Automatic Reasoning and Planning

Course Summary - Master's Degree

Carlos Alberto Botina Carpio
Universidad Internacional de la Rioja
carlos.botina621@comunidadunir.net

January 2, 2026

Abstract

This document contains a summary of the Automatic Reasoning and Planning course syllabus for the Master's Degree. It includes a summary of the main topics covered during the sessions, as well as additional explanations and extensions of the concepts and techniques referenced in class. The purpose of this document is to serve as study material and reference for the course contents.

Notice that this document is fully customized to the author's needs, placing greater emphasis on topics that the author struggles with or has not yet mastered, while covering more briefly those topics that are already well understood by the author.

Contents

Lecture 001

1 Introduction to Decision Making

Decision making is a complex process that requires evaluating and understanding environmental conditions. In many cases, decisions become so intricate that we need to exhaustively evaluate every possible scenario. This is where mathematics becomes useful in helping us make informed decisions.

When making decisions, the following elements are involved:

- **Future effect**: What is the duration of the decision's impact? (short-term, long-term)
- **Reversibility**: Is it easy to reverse the decision?
- **Impact**: How many areas or domains are affected?
- **Quality**: Ethics, legality, behavioral principles, workplace relationships, etc.
- **Periodicity**: How frequently must this decision be made?

1.1 Decision Classification

Decisions can be classified into two main categories: **high-level** and **low-level** decisions. High-level decisions are strategic, have long-term consequences, are difficult to reverse, and affect multiple areas of an organization or system. They are typically exceptional and require careful consideration of various quality factors. In contrast, low-level decisions are operational, have minimal future impact, are easily reversible, and affect fewer areas. They are made frequently and have limited impact on important quality factors.

Table 1 summarizes the characteristics that distinguish high-level from low-level decisions based on the elements involved in decision making.

Table 1: Classification of decisions: High-level vs. Low-level

	High Level	**Low Level**
Future Effect	Affect the future	Don't affect the future
Reversibility	Difficult reversibility	Reversible
Impact	Broad impact	Little impact
Quality Factors	Affect many important	Affect few important
Periodicity	Exceptional	Frequent

Decisions can also be classified as **programmed** or **non-programmed**. Programmed decisions have a well-defined step-by-step sequence that is known and can be followed. For example, in case of an emergency, one calls the emergency number. Non-programmed decisions are unique and specific to the situation, with no defined rules or steps to follow.

1.2 Problem Classification

Problems can be classified as **structured** or **non-structured**:

- **Structured problems**: The problem contains all the information needed to solve it. All necessary data, constraints, and conditions are available from the start.
  - *Example*: Solving a system of linear equations where all coefficients and constants are given.
- **Non-structured problems**: The problem does not contain all the information needed to solve it. To solve it, we need to search for additional information.
  - *Example*: Diagnosing a medical condition where symptoms are present but additional tests, patient history, or expert consultation are required to reach a diagnosis.

2 Problem Solving Stages

2.1 First Stage: Understand the Problem's Complexity

There are many techniques to understand a problem's complexity, which can be organized into two main categories:

- **Identify the problem**: Self-questioning about the problem (origin, magnitude, focus, or history), SWOT analysis (Strengths, Weaknesses, Opportunities, Threats), discussion meetings (Scrum), etc.
- **Explain the problem**: Go beyond the surface and investigate the underlying causes of the problem. This can be done using various techniques such as ERIM problem classification, the 20 causes technique, the Ishikawa diagram (fishbone diagram), and other root cause analysis methods.

We must, therefore, create problem definitions that present characteristics that allow us to work with them efficiently. Some characteristics or assumptions that can be made in simple environments (well-defined problems) are:

- **Discrete**: The world can be conceived in states. In each state there is a finite set of perceptions and actions.
- **Accessible**: The agent can access the relevant characteristics of the environment. It can determine the **current state** of the world and the **state it would like to reach**.
- **Static and deterministic**: There is no temporal pressure nor uncertainty. The world changes only when the agent acts. The result of each action is totally defined and predictable.

2.2 Second Stage: Create a Strategy

The steps for creating a strategy are:

1. **Define strategies**: Generate potential strategies using techniques such as brainstorming, 4x4x4, etc.
2. **Choose a strategy**: Select a strategy by evaluating:
    - Benefits
    - Probability of success
    - Dependencies
    - Resources needed (time, cost)
3. **Design the strategy**: Create a roadmap defining which actions will be performed. This involves planning the sequence and details of the actions to be executed.

2.3 Third Stage: Solve the Problem

This stage is achieved by implementing the strategy, evaluating the outcome, and if necessary, refining the strategy.

3 Intelligent Agents

An **intelligent agent** is an autonomous entity that perceives its environment through sensors and acts upon that environment through actuators to achieve its goals or objectives. Intelligent agents are characterized by their ability to operate independently, make decisions based on their perceptions, and take actions that affect their environment in pursuit of their goals.

An agent is considered intelligent based on its **autonomy** and **rationality**:

- **Autonomy**: The agent operates independently without direct human intervention or control. It has control over its own actions and internal state.
- **Rationality**: The agent acts in a way that maximizes its performance measure, given the available information and its knowledge. A rational agent selects actions that are expected to achieve its goals most effectively.

3.1 Phases of Agent Decision Making

An agent must go through three fundamental phases: **feel**, **think**, and **act**.

- **Feel**: Using perception of the environment through their sensors, the agent extracts and processes information. This phase involves gathering raw data from the environment and converting it into a usable format.
- **Think**: The agent reasons and decides through a deliberative process, using the information obtained from the environment and its internal memory. This phase involves analyzing the current situation, considering possible actions, and selecting the best course of action to achieve its goals.

- **Act**: Through actuators, the agent produces changes in the environment that will help it achieve its goal. For this, the agent must convert its decisions into information that the actuators can understand and execute.

Example: Consider a robotic vacuum cleaner agent. In the **feel** phase, it uses sensors (cameras, bump sensors, dirt detectors) to perceive the room layout, detect obstacles, and identify dirty areas. In the **think** phase, it processes this information along with its internal map and battery level, deciding which areas to clean next and planning an efficient path. In the **act** phase, it converts these decisions into motor commands that control its wheels and vacuum mechanism, moving through the room and cleaning the identified areas.

4 Intelligent Agents Architectures

Agent architectures define the internal structure and organization of an intelligent agent, determining how it processes information, makes decisions, and acts. There are several main types of agent architectures: **deliberative**, **reactive**, **hybrid**, and **cognitive**.

4.1 Deliberative Architecture

A **deliberative architecture** (also known as a *symbolic* or *thinking* architecture) is characterized by the agent's ability to maintain an internal model of the world and engage in explicit reasoning and planning before taking action. The agent uses symbolic representations of knowledge and performs logical reasoning to determine the best course of action.

Key characteristics:

- Maintains an internal model or representation of the world
- Uses symbolic knowledge representation
- Performs explicit reasoning and planning
- Makes decisions based on logical inference
- Typically slower to respond but more thoughtful

This process involves explicit reasoning and planning, making it a deliberative approach. The agent "thinks before it acts," considering multiple possibilities and their outcomes.

Figure 1 illustrates the classic deliberative agent architecture, which follows a Sense-Plan-Act cycle. The architecture consists of three main internal components:

- **Deliberative Planner**: Generates high-level plans based on goals and current state information. It receives failure signals from the monitoring component and creates or refines plans accordingly.
- **Monitoring**: Monitors the execution of the plan, compares the current state with the expected state, and sends failure signals to the planner if deviations occur. It receives plans from the planner and sends specific actions to the execution component.

- **Execution**: Interacts directly with the environment through sensors and actuators. It receives sensor data from the environment, updates the monitoring component with the current state, receives action commands from monitoring, and executes actions in the environment.

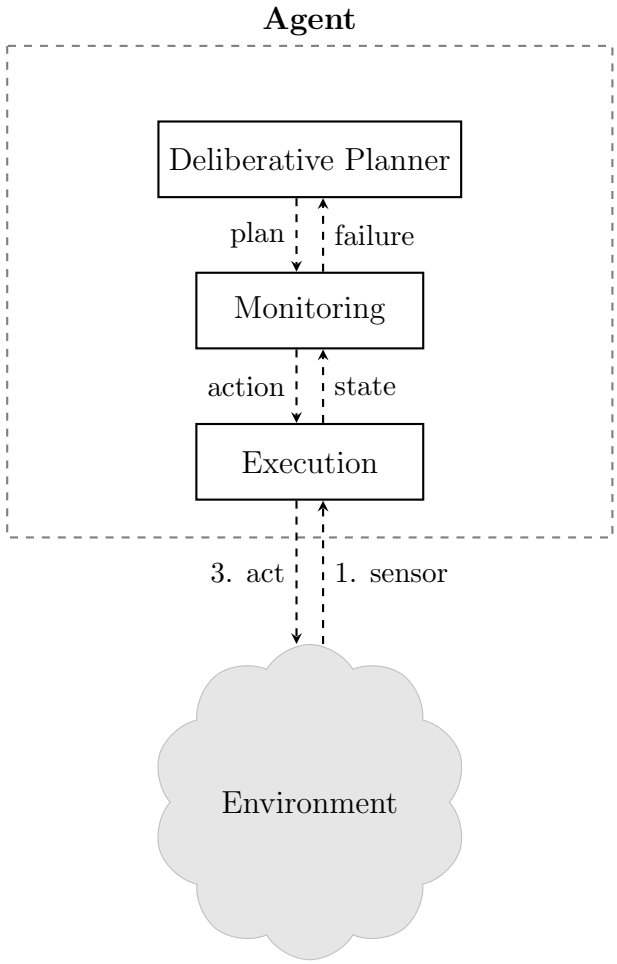

Figure 1: Classic deliberative agent architecture (Sense-Plan-Act cycle)

The flow operates as a continuous cycle:

1. The agent **senses** the environment through sensors (1. sensor), providing raw data to the Execution component.
2. Execution updates Monitoring with the current **state**.
3. Monitoring compares the state with the **plan** received from the Deliberative Planner and determines the next **action** for Execution. If the plan is not progressing as expected, Monitoring sends a **failure** signal to the Deliberative Planner.
4. The Deliberative Planner receives failure signals and generates or refines a **plan**, which is sent to Monitoring.
5. Execution **acts** (3. act) upon the environment based on the action received from Monitoring.

This cycle represents how a deliberative agent continuously perceives, plans, and acts to achieve its goals, with a mechanism for detecting and responding to plan failures.

4.1.1 Examples of Deliberative Architectures

- **Planning agents**: These agents use automated planning algorithms to generate sequences of actions that achieve specific goals. They maintain a symbolic representation of the world state and use search algorithms to find optimal or near-optimal plans. Planning agents are commonly used in robotics, autonomous systems, and game AI where complex sequences of actions need to be coordinated.

- **Belief Desire & Intention (BDI)**: This architecture models agents based on three mental attitudes: **Beliefs** (what the agent knows about the world), **Desires** (the agent's goals or objectives), and **Intentions** (the commitments to specific plans of action). BDI agents reason about their beliefs, select desires to pursue, and commit to intentions (plans) to achieve those desires. This architecture is particularly useful for modeling complex, goal-oriented behavior in multi-agent systems and autonomous agents.

4.2 Reactive Architecture

A **reactive architecture** (also known as a *behavior-based* architecture) is characterized by the agent's direct mapping from perceptions to actions without maintaining an internal world model or engaging in complex reasoning. The agent responds quickly to environmental stimuli through simple stimulus-response rules.

Key characteristics:

- No internal world model or symbolic representation
- Direct mapping from sensors to actuators
- Simple stimulus-response rules or behaviors
- Fast response time
- Emergent behavior from simple rules

Example: Consider a simple obstacle-avoidance robot with a reactive architecture. The robot has sensors on its front and sides, and it follows these simple rules:

- If the front sensor detects an obstacle, turn right
- If the right sensor detects an obstacle, turn left
- If no obstacles are detected, move forward
- If both sensors detect obstacles, move backward

The robot doesn't maintain a map of its environment or plan a path. It simply reacts to what it perceives at each moment. This makes it very fast and efficient for simple tasks, though it may not find the optimal path. The robot's navigation behavior emerges from these simple reactive rules.

Figure 2 illustrates a reactive agent architecture that incorporates a **Reactive Planner** component:

- **Reactive Planner**: Positioned at the top, this component generates reactive action sequences in direct response to the current state. Unlike deliberative planners, it does not maintain a complex world model but instead quickly generates actions based on immediate perceptions. It receives failure signals from Monitoring and sends actions back to Monitoring.
- **Monitoring**: Positioned in the middle, it monitors the execution of actions and the current state. It receives state information from Execution, sends failure signals to the Reactive Planner when deviations occur, receives actions from the Reactive Planner, and sends action commands to Execution. It also receives external plans.
- **Execution**: Located at the bottom, it interacts directly with the environment through sensors and actuators. It receives sensor data from the environment, updates Monitoring with the current state, receives action commands from Monitoring, and executes actions in the environment.

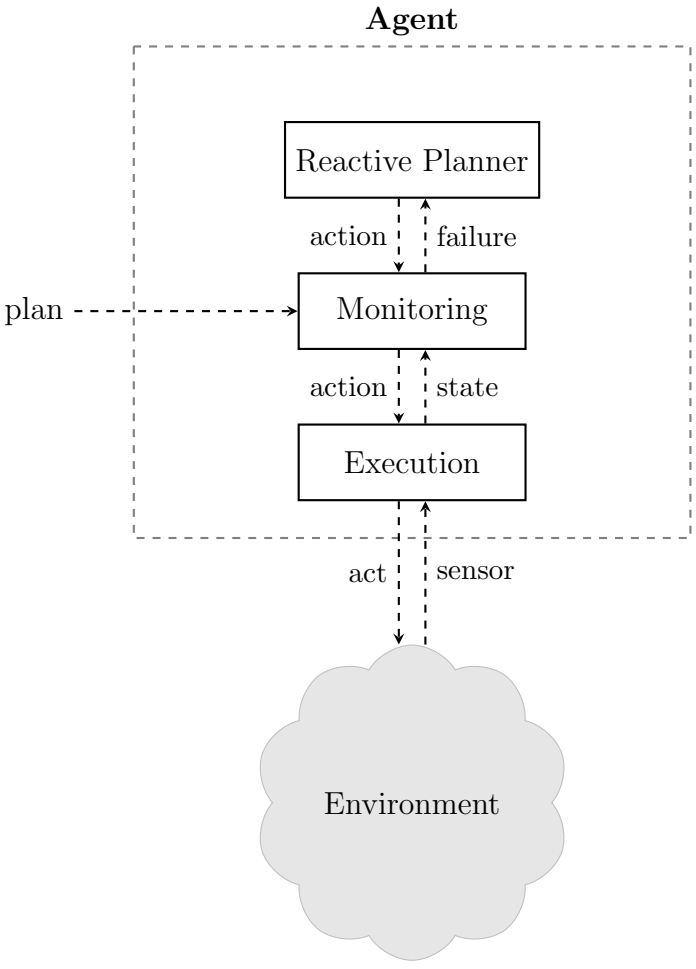

Figure 2: Reactive agent architecture with Reactive Planner

The flow operates as follows:

1. An external **plan** is provided to the Monitoring component (this could come from a higher-level planner or user input).

2. The Environment sends **sensor** data to the Execution component, providing raw information about the current state of the environment.

3. Execution processes the sensor data and updates Monitoring with the current **state**.

4. Monitoring evaluates the state against the plan. If there is a deviation or problem, it sends a **failure** signal to the Reactive Planner.

5. The Reactive Planner receives the failure signal and immediately generates a reactive **action** response, which it sends back to Monitoring.

6. Monitoring receives the action from the Reactive Planner and forwards the **action** command to Execution.

7. Execution **acts** upon the environment based on the action received from Monitoring.

This architecture emphasizes fast, reactive responses to environmental changes while still allowing for external plan guidance. The Reactive Planner provides quick adaptation without the overhead of maintaining complex internal models. The bidirectional flow between Monitoring and the Reactive Planner enables rapid failure detection and action generation.

4.2.1 Examples of Reactive Architectures

- **Subsumption architecture**: A layered architecture where behaviors are organized in levels of increasing complexity. Lower-level behaviors (like obstacle avoidance) can subsume or override higher-level behaviors (like exploration) when triggered by environmental conditions. Each layer operates independently and reactively, with no central control or world model.

- **Agent network architecture from Pattie Maes**: A distributed reactive architecture where multiple simple agents (or behaviors) are connected in a network. Each agent has local rules and can activate or inhibit other agents. The overall behavior emerges from the interactions between these agents, without centralized planning.

- **Reactive execution model**: Is domain independent and operates with structures precalculated at runtime.

4.3 Hybrid Architecture

A **hybrid architecture** combines elements of both deliberative and reactive architectures. It typically has multiple layers: a reactive layer for fast, immediate responses to critical situations, and a deliberative layer for complex planning and reasoning. This architecture attempts to get the best of both worlds: the speed of reactive systems and the intelligence of deliberative systems.

Key characteristics:

- Combines reactive and deliberative components

- Multiple layers of control (reactive at the bottom, deliberative at the top)

- Fast response for urgent situations (reactive layer)
- Complex reasoning and planning for strategic decisions (deliberative layer)
- Coordination between layers

Example: Consider an autonomous vehicle with a hybrid architecture. The vehicle has two main layers:

- **Reactive layer**: Handles immediate, critical situations. For example:
  - If a pedestrian suddenly appears in front, immediately apply brakes (no time for planning)
  - If another vehicle swerves into the lane, quickly adjust steering
- **Deliberative layer**: Handles strategic planning and navigation. For example:
  - Plans the route from origin to destination
  - Analyzes traffic conditions and selects the best path
  - Decides when to change lanes based on traffic patterns
  - Maintains a map and tracks the vehicle's position

The reactive layer ensures safety by responding instantly to immediate threats, while the deliberative layer handles the overall navigation strategy. The layers work together: the deliberative layer sets the general plan, and the reactive layer handles unexpected situations that require immediate action.

4.3.1 Examples of Hybrid Architectures

- **Procedural Reasoning System (PRS)**: A BDI (Belief-Desire-Intention) architecture that combines deliberative reasoning with reactive capabilities. The agent maintains **Beliefs** (facts about the world expressed in first-order logic), **Desires** (system behaviors or goals), and **Intentions** (the current set of active plans). PRS includes a library of partially specified plans called Knowledge Areas (KAs), each with an activation condition. KAs can be activated by goals or by data, and can be reactive, allowing PRS to respond quickly to environmental changes while also reasoning about which plans to execute.
- **COSY (Cooperative System)**: A BDI architecture that combines elements from both PRS and IRMA architectures. It has five main components: **Sensors** (receive perceptual inputs), **Actuators** (perform actions), **Communications** (send messages), **Cognition** (mediates between intentions and knowledge to choose actions), and **Intention** (contains long-term goals and control elements). COSY combines deliberative reasoning with reactive communication and action capabilities, making it suitable for interactive and collaborative environments.

4.4 Cognitive Architecture

A **cognitive architecture** is a computational framework that models the structure and processes of human cognition. It provides a unified theory of how the mind works, including perception, memory, reasoning, learning, and decision-making. Cognitive architectures can be defined as a hypothesis about the fixed structures that provide a mind, whether in natural or artificial systems, and how they work together—along with the knowledge and skills incorporated within the architecture—to produce intelligent behavior in a diversity of complex environments. Cognitive architectures aim to create artificial agents that can exhibit human-like intelligence and behavior.

Key characteristics:

- Models human cognitive processes and structures
- Provides unified framework for multiple cognitive functions
- Includes memory systems (short-term and long-term)
- Supports learning and adaptation
- Integrates perception, reasoning, and action
- Based on cognitive science and psychology principles

4.4.1 Examples of Cognitive Architectures

- **ACT-R**: A cognitive architecture developed primarily by John Robert Anderson at Carnegie Mellon University. The most important assumption of ACT-R is that human knowledge can be divided into two irreducible types of representations:
  - **Declarative knowledge**: Represented as **chunks** (vector representations of individual properties, each accessible from a labeled slot)
  - **Procedural knowledge**: Represented as production rules

  Chunks are maintained and accessed through **buffers**, which are the front-end of **modules** (specialized and largely independent brain structures). ACT-R has two main types of modules:
  - **Perceptual-motor module**: Manages interaction with the environment, handling the flow of perception and action to connect the agent with the world.
  - **Memory module**: Divided into:
    - **Long-term memory** (production memory): Contains production rules
    - **Short-term memory** (working memory or declarative memory): Contains current facts about the world
- **SOAR**: A cognitive architecture created at Carnegie Mellon University by Laird, Newell, and Rosenbloom (1987). The ultimate goal of SOAR is to provide a foundation for a system capable of general intelligent behavior, supporting the full range of cognitive tasks, problem-solving methods, and knowledge representations. SOAR is both a theory of cognition and a computational implementation of that theory.

The design of SOAR is based on the hypothesis that all goal-oriented deliberate behavior can be understood as the selection and application of **operators** to a **state**:

- **State**: A representation of the current problem situation
- **Operator**: Transforms a state (performs changes in the representation)
- **Goal**: A desired result for the problem

SOAR runs continuously, attempting to apply the current operator and select the next operator (a state can have only one operator at a time) until the goal is achieved.

SOAR has separate memories with different representation modes:

- **Short-term memory**: Stores sensor data, intermediate inferences about current data, currently active goals, and active operators (actions and plans)
- **Long-term memory** (production memory): Maintains knowledge for responding to situations through procedures. It stores problem-solving knowledge, inference rules, and knowledge for selecting and applying operators in specific states

4.5 Other Architectures

- Three layer architecture
- Multilayer architecture
- Three tower architecture

5 Summary - Lecture 001

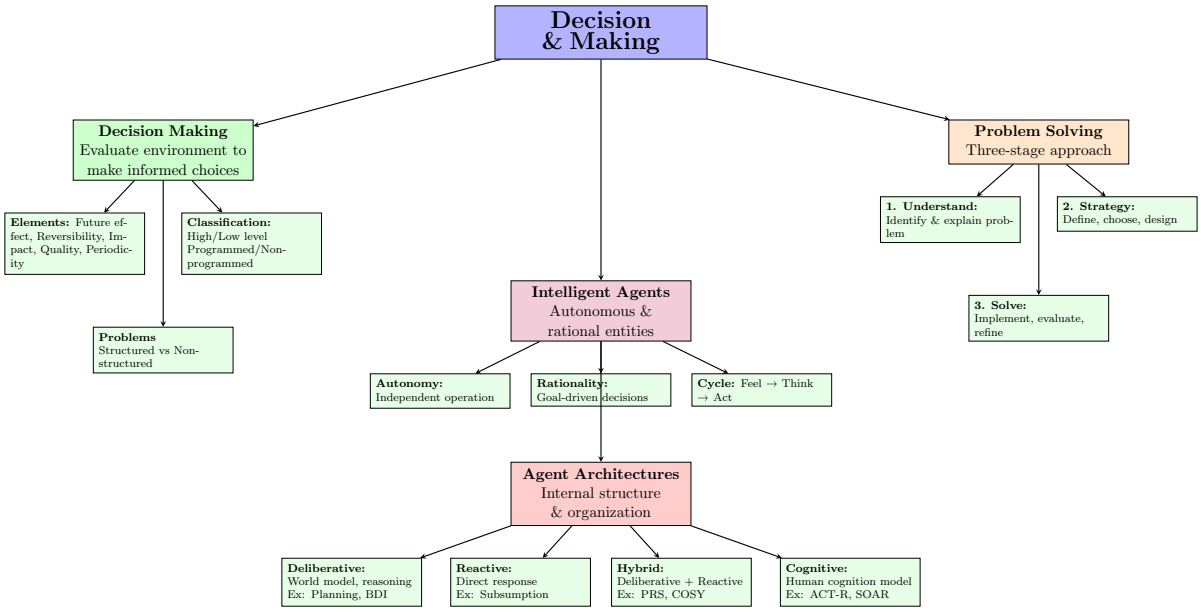

Figure 3: Conceptual map of the Reasoning and Planning course - Lecture 001

Lecture 002

6 Symbolic Representation of Knowledge

When agents make decisions based on changes in the environment, a fundamental question arises: **how do we represent the environment and the problem's information?** Agents need a way to understand and work with information about the world, which is where symbolic representation becomes essential.

Reasoning is an internal process that operates on external entities. The key insight is that **operations with representations substitute operations with the real world**. Instead of directly interacting with the environment, agents can manipulate symbolic representations of it, allowing them to reason about possible actions and outcomes before actually taking action.

Symbolic representation allows agents to create **internal models of the external world using symbols** (such as variables, predicates, logical formulas, or other formal structures). These representations enable reasoning processes that can:

- **Simulate** different scenarios and outcomes
- **Predict** the consequences of actions
- **Plan** sequences of actions before execution
- **Learn** from experience without direct interaction

This makes decision-making more efficient and safer, as agents can explore possibilities and evaluate strategies in a controlled, internal environment before committing to actions

in the real world. It is important to note that reasoning and action are **complementary** rather than substitutes: reasoning often precedes and guides action, enabling agents to make informed decisions rather than acting randomly.

6.1 Requirements for Knowledge Representation

According to (Molina González, 2006) a knowledge representation must satisfy:

1. **Formal**: The representation must be unambiguous. Natural language is not considered a knowledge representation because of its ambiguities.

   Example: The sentence "I saw the man with binoculars" is ambiguous—it could mean "I used binoculars to see the man" or "I saw the man who had binoculars." A formal representation would explicitly distinguish these meanings, such as `Saw(I, man)` $\wedge$ `Used(I, binoculars)` versus `Saw(I, man)` $\wedge$ `Has(man, binoculars)`.

2. **Expressive**: The representation must be rich enough to capture the different aspects that need to be distinguished. For example, first-order predicate logic formulas are more expressive than propositional calculus.

   Example: Propositional logic can only express simple statements like "It is raining" (P) or "The ground is wet" (Q). Predicate logic can express relationships and quantification, such as $\forall$`x (Bird(x)` $\rightarrow$ `CanFly(x))` meaning "All birds can fly," or `Loves(john, mary)` expressing the relationship "John loves Mary."

3. **Natural**: The representation should be sufficiently analogous to natural ways of expressing knowledge. Traditional quantitative mathematical representations (e.g., matrices) can be too artificial for emulating reasoning processes.

   Example: Representing "John loves Mary" as a matrix entry (row 1, column 2 = 1) is mathematically precise but doesn't match how humans naturally think about relationships. A more natural representation would be a semantic network with nodes and edges: `[John] --loves--> [Mary]`, or a frame structure with properties, which aligns better with human cognitive patterns.

4. **Tractable**: The representation must be computationally tractable, meaning there must exist sufficiently efficient procedures to generate answers through manipulation of the knowledge base elements.

   Example: While a representation might be perfectly formal and expressive, if answering queries requires exponential time or is undecidable, it becomes impractical. For instance, certain logical formalisms may be too complex to reason about efficiently, making them unsuitable for real-time applications despite their theoretical expressiveness.

It is important to note that "natural" in this context refers to **cognitive naturalness**—how well the representation aligns with human thinking patterns—not to natural language. Natural language is ambiguous and therefore not formal, while we seek representations that are both formal (unambiguous) and natural (intuitive). The ideal representation balances both requirements.

In general, it is convenient to create information representations based on a **single representation**. This approach improves knowledge base maintenance, as it provides consistency and simplifies updates across the system.

Example: A system might use only first-order predicate logic to represent all knowledge:

- Facts: `Student(john)`, `Course(cs101)`, `Enrolled(john, cs101)`
- Rules: $\forall$`x (Student(x)` $\wedge$ `Enrolled(x, y)` $\rightarrow$ `TakesCourse(x, y))`
- Relationships: `Teaches(profSmith, cs101)`, `Prerequisite(cs101, cs201)`

Alternatively, a system might use only semantic networks, representing everything as nodes and edges: `[John] --is_a--> [Student]`, `[John] --enrolled_in--> [CS101]`, etc.

However, in some cases, attempting to represent all knowledge in a single representation can **limit the application of techniques and algorithms**. Different problems and subtasks may require different representation formalisms or specialized techniques that are not well-suited to a unified representation.

For complex systems that require making decisions at different levels, it is common to employ the idea of generating **multi-layer agents** (Molina González, 2006) that decompose the problem into levels. For each level, a treatment is established oriented to a specialized agent that needs a **concrete representation of part of the environment information**. Each agent at its respective layer uses a representation tailored to its specific needs and the techniques it employs.

6.2 Types of Knowledge

To solve problems in a natural way, it is necessary to carry out a precise analysis of knowledge. For this, we must take into account the different classifications of knowledge used in artificial intelligence (Molina González, 2006). The main types of knowledge are:

1. **Domain Knowledge**: Knowledge about a specific context or problem, represented in a **declarative** way (describes what exists, properties, and constraints). It can be incomplete and does not require order or relationships between elements.
2. **Explicit Knowledge**: **Knowledge extracted from introspective analysis of one's own reasoning and problem-solving processes**. It is usually expressed through frames or rules about how problems are solved.
3. **Implicit Knowledge**: Knowledge about innate capacities or abilities that are not easily expressed verbally. Bayesian networks or neural networks are used for its representation or modeling.
4. **Superficial Knowledge**: Knowledge obtained through experience in solving similar problems. It uses practical rules or heuristics that work but don't explain the underlying theoretical principles.
5. **Deep Knowledge**: Knowledge based on a well-structured theoretical framework that explains the underlying principles and mechanisms in detail. However, it is not present in many problems because it is not easy to have a theoretical analysis of the environment's functioning in all problems.

6. **Control Knowledge**: The **strategy/organization for the problem-solving process** — the execution order and approach for how to solve the problem, not just the steps of the task itself. In many cases, it can lead to coding a program in the expansion sequence of search algorithms.

7. **Metaknowledge**: **Knowledge about how to generate, transfer, and learn from knowledge**. It allows generating new models from previous problem models and establishes relationships between levels of knowledge bases.

7 Introduction to Reasoning

Reasoning refers to a set of mental activities that connect ideas based on rules that justify an idea, allowing problem-solving through conclusions.

As can be observed in these definitions, all authors refer to the same concepts: **premises** (initial propositions, what is already known) and **conclusion** (final proposition obtained from the premises, representing new knowledge).

7.1 Elements of Reasoning: Content and Form

In all reasoning, there exist two elements: **content** and **form**.

- **Content**: What makes a proposition true or false. It is the reference to objects and properties. Content deals with the actual meaning and truth value of statements in the real world.

  Example: In the proposition "It is raining," the content refers to the actual weather condition. This proposition can be true or false depending on whether it is actually raining.

- **Form**: The logical connection between the antecedents (what is already known, the premises) and the consequents (the conclusion inferred from the antecedents). This connection that implies inference is expressed through conjunctions. Form is what makes the proposition **valid**, and consists of using symbols to express the validity of propositions. Form deals with the logical structure of reasoning, independent of whether the statements are actually true or false.

  Example: Consider the following reasoning:

  - Premise 1: "If it rains, then the ground gets wet"
  - Premise 2: "It is raining"
  - Conclusion: "Therefore, the ground gets wet"

  The form of this reasoning is: `If P, then Q. P. Therefore, Q.` This logical structure is valid regardless of whether it is actually raining or not. The same form can be applied to different content:

  - Premise 1: "If John is a student, then John studies"
  - Premise 2: "John is a student"
  - Conclusion: "Therefore, John studies"

Both examples share the same valid logical form, even though they have different content.

The distinction between content and form is crucial: **content** determines whether propositions are true or false in the real world, while **form** determines whether the reasoning structure is valid (whether the conclusion follows logically from the premises).

We speak of **valid reasoning** when the conclusion follows from the premises. A reasoning is considered valid based on its logical form, even if the conclusion or the premises are false. On the other hand, **invalid reasoning** occurs when, from true premises, a false conclusion is obtained.

Table 2 shows the different scenarios for reasoning validity based on the truth values of premises and conclusion:

Table 2: Validity or Non-Validity of a Reasoning

If the premises are...	**And the conclusion is...**	**The reasoning is...**
True	True	Valid
True	False	Invalid
False	True	Valid
False	False	Valid

The concept of reasoning validity is directly analogous to the truth conditions of a conditional proposition. Table 3 shows the truth table for a conditional proposition ($p \rightarrow q$), where p represents the premises (antecedent) and q represents the conclusion (consequent):

Table 3: Truth Table of a Conditional Proposition

p	**q**	**p → q**
1 (True)	1 (True)	1 (True)
1 (True)	0 (False)	0 (False)
0 (False)	1 (True)	1 (True)
0 (False)	0 (False)	1 (True)

As can be observed, the highlighted row in both tables (premises True, conclusion False) represents the only scenario where the reasoning is invalid and the conditional proposition is false. This emphasizes that an argument is invalid if and only if it is possible for its premises to be true and its conclusion false.

The following examples illustrate each scenario from Table 2:

1. **True premises → True conclusion (Valid)**:
    - Premise 1: "If it rains, then the ground gets wet" (True)
    - Premise 2: "It is raining" (True)

- Conclusion: "Therefore, the ground is wet" (True)

This reasoning is **valid** because the conclusion follows logically from the premises (modus ponens), and all statements are true.

2. **True premises → False conclusion (Invalid)**:

   - Premise 1: "All dogs are animals" (True)
   - Premise 2: "Lassie is an animal" (True)
   - Conclusion: "Therefore, Lassie is a dog" (False - Lassie could be any animal)

   This reasoning is **invalid** because the conclusion does not follow from the premises (fallacy of affirming the consequent). Even though the premises are true, the logical form is incorrect, making it possible for the conclusion to be false.

3. **False premises → True conclusion (Valid)**:

   - Premise 1: "If it is July, then it is winter" (False - in the Northern Hemisphere)
   - Premise 2: "It is July" (False - assume it is actually January)
   - Conclusion: "Therefore, it is winter" (True - it is winter in January)

   This reasoning is **valid** because the logical form (modus ponens) is correct, even though both premises are false and the conclusion happens to be true. The validity depends on the form, not the truth values.

4. **False premises → False conclusion (Valid)**:

   - Premise 1: "All insects are mammals" (False)
   - Premise 2: "Spiders are insects" (False - spiders are arachnids)
   - Conclusion: "Therefore, spiders are mammals" (False)

   This reasoning is **valid** because the conclusion follows logically from the premises using a valid syllogistic form. Even though both premises and the conclusion are false, the logical structure is correct—if the premises were true, the conclusion would necessarily follow.

8 Reasoning Classification

Although there are many types of reasoning, we will focus on the most important ones for artificial intelligence: **deductive**, **inductive**, and **abductive** reasoning.

8.1 Deductive Reasoning

Reasoning is **deductive** when it requires that the conclusion necessarily and forcibly derives from the premises. For this reason, it is considered rigorous.

Example of deductive reasoning:

- "If it snows, then it is cold"
- "It is snowing"

- "Therefore, I am cold"

It is understood that validity exists when, from true premises, a false conclusion cannot be obtained. From false premises, true conclusions can be derived, and yet the argument can still be valid.

Truth occurs when what is described in the premises corresponds to reality. This type of reasoning goes from **general to particular**.

8.1.1 Types of Deductive Reasoning

Within deductive reasoning, several types are distinguished:

- **Categorical deductive reasoning**: Starts from two true premises that will lead to a true conclusion.

  Example:

  - Premise 1: "All humans are mortal" (True)
  - Premise 2: "Socrates is a human" (True)
  - Conclusion: "Therefore, Socrates is mortal" (True)

- **Propositional deductive reasoning**: Relates two premises where one is a condition of the other, antecedent and consequent.

  Example:

  - Premise 1: "If it rains, then the ground gets wet" (antecedent: it rains, consequent: ground gets wet)
  - Premise 2: "It is raining" (antecedent is true)
  - Conclusion: "Therefore, the ground is wet" (consequent follows)

- **Disjunction or dilemma**: The relationship between the premises is one of contraries, therefore the conclusion discards one of them.

  Example:

  - Premise 1: "Either it is day or it is night"
  - Premise 2: "It is not day"
  - Conclusion: "Therefore, it is night" (discards the first option)

8.1.2 Forms of Deductive Reasoning

There are two forms of deductive reasoning:

- **Immediate**: The only logical operation is the change of judgment.

  Example:

  - Original judgment: "All students are learners"
  - Immediate conclusion: "No students are non-learners" (direct conversion/-transformation of the same judgment)

In immediate reasoning, the conclusion is obtained directly from a single premise by changing its form, without needing additional premises.

- **Mediate**: A mediation relationship is established between judgments to reach the conclusion.

  Example:

  - Premise 1: "All mammals are warm-blooded"
  - Premise 2: "All dogs are mammals" (middle term: "mammals")
  - Conclusion: "Therefore, all dogs are warm-blooded"

  In mediate reasoning, the conclusion is reached by connecting two premises through a middle term (in this case, "mammals"), which mediates the relationship between the other terms.

The deductive method goes from **general to particular**. Table 4 illustrates a classic syllogism example:

Table 4: Example of a syllogism

Part	**Abbreviation**	**Statement**
Major Premise	MP	Humans are mortal.
Minor Premise	SM	Greeks are humans.
Conclusion	SP	Greeks are mortal.

In syllogistic logic, the abbreviations follow a standard terminology:

- **S (Subject)**: The subject of the conclusion ("Greeks")
- **P (Predicate)**: The predicate of the conclusion ("mortal")
- **M (Middle)**: The term that appears in both premises but not in the conclusion ("humans")

In this syllogism:

- **Major Premise (MP)**: Contains the Major Term (P = "mortal") and Middle Term (M = "humans")
- **Minor Premise (SM)**: Contains the Minor Term (S = "Greeks") and Middle Term (M = "humans")
- **Conclusion (SP)**: Contains the Minor Term (S = "Greeks") and Major Term (P = "mortal")

The conclusion is labeled **SP** because it contains the Subject (S) and Predicate (P) terms. The middle term (M = "humans") connects the premises but does not appear in the conclusion.

8.2 Inductive Reasoning

Inductive reasoning creates probable conclusions according to the given premises. It is based on the idea that if various events present the same situation as their premises, there is a probability that the result will be identical. To induce means precisely to extract general conclusions from particular experiences.

The difference with deductive reasoning is that the conclusion is **not necessarily obtained** from the premises. The conclusion of inductive reasoning is obtained through the direct observation of particular cases.

8.2.1 Types of Inductive Reasoning

Within inductive reasoning, there are different types:

- **Complete inductive reasoning** (also called perfect inductive reasoning): Occurs when all particular cases are included in the premises.

  Example:

  - Observation: "Student 1 passed the exam"
  - Observation: "Student 2 passed the exam"
  - Observation: "Student 3 passed the exam"
  - Observation: "Student 4 passed the exam"
  - Observation: "Student 5 passed the exam"
  - (These are all the students in the class)
  - Conclusion: "Therefore, all students in the class passed the exam"

  Since all particular cases (all students) have been observed, this is complete inductive reasoning.

- **Incomplete inductive reasoning** or imperfect inductive reasoning: Only certain particular cases are included in the premises.

  Example:

  - Observation: "The swan I saw in the park is white"
  - Observation: "The swan I saw at the lake is white"
  - Observation: "The swan I saw in the zoo is white"
  - Observation: "The swan I saw in the river is white"
  - (These are only some of all the swans that exist)
  - Conclusion: "Therefore, all swans are white"

  Since only some particular cases (some swans) have been observed, this is incomplete inductive reasoning. The conclusion is probable but not certain, as there might be swans that haven't been observed (e.g., black swans in Australia).

The inductive method goes from **particular to general**. Table 5 illustrates how inductive reasoning works, showing the reverse direction compared to deductive reasoning:

Table 5: Example of inductive reasoning

Part	**Abbreviation**	**Statement**
Minor Premise	SM	Greeks are human beings.
Conclusion	SP	Greeks are mortal.
Major Premise	MP	Human beings are mortal.

In inductive reasoning, we start by observing particular cases (Greeks are humans, Greeks are mortal) and then infer the general rule (Human beings are mortal). This is the opposite direction of deductive reasoning, which starts with the general rule and applies it to particular cases.

8.3 Abductive Reasoning

Abductive reasoning (also called retroduction) is a method used to find explanations for observed facts. From a fact, we arrive at the actions that caused it. Aristotle was the first to describe this type of reasoning.

Key characteristics:

- Starts from **facts** and seeks a **theory** (from effect to cause)
- The key concept is the **syllogism**, where:
  - Major premise is considered **certain**
  - Minor premise is considered **probable**
  - Conclusion has the same level of **probability** as the minor premise

  Example:

  - Major Premise (certain): "If it rains, then the ground gets wet"
  - Minor Premise (probable): "The ground is wet" (observed fact)
  - Conclusion (probable): "Therefore, it probably rained" (inferred cause from the effect)
- Relates the observable with something that cannot be directly observed
- For Charles S. Peirce (Peirce, 1867), it is an inferential process related to the **generation of hypotheses**

Abductive reasoning process (three steps):

1. The object or fact (observation)
2. Hypothesis of why the object or fact occurs
3. Affirm that the cause was responsible for the object or fact

Scheme: "I see A with characteristic Z. Since all A I see are Z, then any element A has characteristic Z."

Importance:

- Allows thinking in an **alternative way**, without following usual reasoning paths
- Leads to **disruptive and novel solutions**
- Contrary to deductive reasoning, which keeps us in the comfort zone
- **Innovation** is strongly linked to abductive reasoning
- Enriches processes in the testing phase, providing a perspective of change

9 Summary - Lecture 002

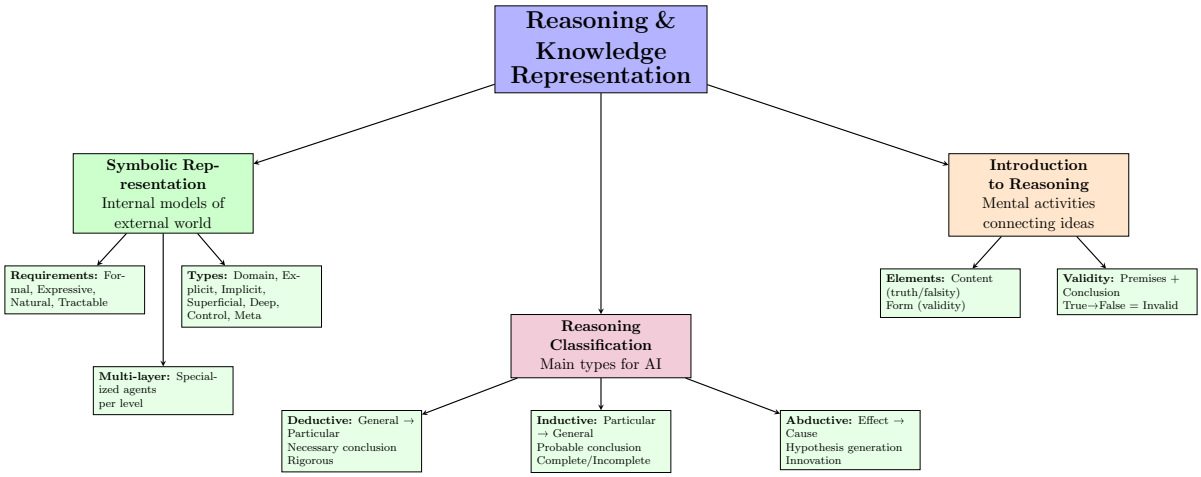

Figure 4: Conceptual map of the Reasoning and Planning course - Lecture 002

Lecture 003

10 Introduction to Logic

Logic is key for representing human thinking and building systems that emulate thought processes. It's a tool for explaining knowledge based on reasoning elements: categories, definitions, judgments, and propositions. From Aristotle to today, logic has evolved into a powerful tool for modeling behavior in AI.

Logic is a **formal science** (not empirical). **Formal sciences** like logic and mathematics work with abstract structures, symbols, and inference rules through **deductive** reasoning—starting from axioms and deriving conclusions via formal rules, no empirical observation needed. **Empirical sciences** like physics and biology observe the world and test hypotheses through experimentation using **inductive** reasoning. Logic studies the **structure of reasoning itself**, focusing on abstract relationships and formal rules, regardless of whether statements are actually true in reality.

Logic enables **formal representation of relationships** between objects and their properties, useful for emulating human thinking, modeling AI behavior, and identifying valid reasoning patterns.

Aristotle is considered the father of logic. He developed the **syllogism**—two premises leading to one conclusion. Example: "All humans are mortal. Socrates is a human. Therefore, Socrates is mortal."

10.1 Types of Logic

There are several types of logic, all focused on understanding reasoning and determining if it's correct or incorrect. They study statements beyond natural language, extending to mathematics and computer science with very different structures.

- **Propositional logic**: Uses propositions (statements that can be true or false) connected with logical operators ($\wedge$ [and], $\vee$ [or], $\neg$ [not]) and rules with implication ($\rightarrow$). Uses inference mechanisms like modus ponens and modus tollens.
- **Predicate logic**: Extends propositional logic by adding quantifiers:
  - $\forall$ (for all)
  - $\exists$ (there exists)

  Uses traditional inference mechanisms (modus ponens, modus tollens). Example: PROLOG programming language.
- **First-order logic**: Formal system for studying inference in first-order languages. Uses quantifiers over individual variables with predicates and functions. Establishes objects and relationships between them. Foundation of computational logic.
- **Formal logic** (classical/Aristotelian): Studies propositions and arguments from a structural perspective. Focuses on argument structure, not content truth/falsity. Includes:
  - **Deductive logic**
  - **Inductive logic**
- **Symbolic/Mathematical logic**: Uses symbols to build a new language for expressing arguments. Translates human thought to mathematical language, converting abstract thinking into formal structures. Used in mathematics to prove theorems.
- **Class logic**: Based on set theory. Analyzes logical propositions about membership (or non-membership) of an element to a class (set of elements sharing a characteristic).
- **Material logic**: Studied from epistemology. Includes uncertainty—conclusions involve some degree of doubt. Proves validity of reasoning based on reality.
- **Natural logic**: Related to empiricism, learning through trial and error. Innate reason that prevents humans from repeating the same mistake.

- **Scientific logic**: Extends natural logic by including reason, creating frameworks for everything that exists. Based on finding reasons or justifications for why facts occur.
- **Informal logic**: Focuses on language and meaning of semantic constructions and arguments. Differs from formal logic by focusing on sentence content rather than structure.
- **Modern logic**: Born in the 19th century, differs from classical logic by including mathematical and symbolic elements, theorems that replace formal logic limitations. Includes:
  - **Modal logic**: Adds modal operators to determine if statements are true/false. Considers expressions like "always", "very likely", "sometimes", "maybe".
  - **Mathematical logic**
  - **Trivalent logic**
- **Computational logic**: Derives from symbolic/mathematical logic (first-order) and applies to computer science. Enables working with programming languages for specific verification tasks.

The logics most relevant to artificial intelligence (which will be covered in detail later) are: mathematical logic, description logic (ALC), higher-order logic, multivalued logic, and fuzzy logic.

11 Propositional Logic

Propositional logic (also called sentential logic or statement logic) is the most basic form of logic. It deals with simple propositions that can be true or false, and how they can be combined using logical connectives.

11.1 Basic Concepts

A **proposition** is any statement or expression that has meaning and of which we can say whether it is false (F/0) or true (V/1). Propositions can be linked together using logical connectives to form structures with precise meaning.

Not propositions: imperative sentences (commands like "read this"), exclamatory and interrogative sentences (questions like "what's your name?"), and instructions (like "go back").

Propositions are usually represented with lowercase letters, e.g., p, q, r. By relating propositions, it's possible to obtain other propositions. All logical reasoning must necessarily start from an adequate connection of some elementary propositions.

Propositions can be classified into:

- **Tautologies**: A compound proposition is a tautology if it's true for all assignments of truth values to its component propositions. Its truth value doesn't depend on the truth values of the propositions that form it, but on the way syntactic relationships are established between propositions.

- **Contradictions**: Propositions that, in all possible cases of their truth table, are always false. Their false value doesn't depend on the truth values of the propositions that form them, but on the way syntactic relationships are established between propositions.
- **Contingencies, fallacies, or inconsistencies**: Also called "truth of fact," these are propositions that can be true or false, combining tautology and contradiction, depending on the values of the propositions that compose them.

A **truth table** is a table that shows all possible combinations of truth values for the component propositions and the resulting truth value of the compound proposition.

11.2 Logical Connectives

The main logical connectives and their truth tables are:

- **Negation** ($\neg$): Denies a proposition. The negation of p is true when p is false, and false when p is true.

Table 6: Truth Table for Negation

p	$\neg p$
1	0
0	1

- **Conjunction** ($\wedge$): "And". The conjunction $p \wedge q$ is true only when both p and q are true.

Table 7: Truth Table for Conjunction

p	q	$p \wedge q$
1	1	1
1	0	0
0	1	0
0	0	0

- **Non-exclusive disjunction** ($\vee$): "Or" (inclusive). The disjunction $p \vee q$ is true when at least one of p or q is true (or both).

Table 8: Truth Table for Non-exclusive Disjunction

p	q	$p \vee q$
1	1	1
1	0	1
0	1	1
0	0	0

- **Exclusive disjunction** ($\oplus$ or $\veebar$): "Either...or" (exclusive). The exclusive disjunction $p \oplus q$ is true when exactly one of p or q is true, but not both.

Table 9: Truth Table for Exclusive Disjunction

p	q	$p \oplus q$
1	1	0
1	0	1
0	1	1
0	0	0

- **Conditional** ($\rightarrow$): "If...then". The conditional $p \rightarrow q$ is false only when p is true and q is false. Otherwise, it's true.

Table 10: Truth Table for Conditional

p	q	$p \rightarrow q$
1	1	1
1	0	0
0	1	1
0	0	1

- **Biconditional** ($\leftrightarrow$): "If and only if". The biconditional $p \leftrightarrow q$ is true when both p and q have the same truth value (both true or both false).

Table 11: Truth Table for Biconditional

p	q	$p \leftrightarrow q$
1	1	1
1	0	0
0	1	0
0	0	1

Key concepts for relationships between propositions:

- **Logical implication**: Any conditional that is a tautology. When a conditional statement ($p \rightarrow q$) is always true regardless of the truth values of its component propositions, it represents a logical implication.

  Example: $(p \wedge q) \rightarrow p$ (if both p and q are true, then p is true). This is a logical implication because the conditional is always true, as shown in Table 12:

Table 12: Truth Table for Logical Implication: $(p \land q) \rightarrow p$

p	q	$p \land q$	$(p \land q) \rightarrow p$
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	1

- **Logical equivalence**: Any biconditional that is a tautology. When a biconditional statement $(p \leftrightarrow q)$ is always true regardless of the truth values of its component propositions, the propositions are logically equivalent.

  Example: $\neg(\neg p) \leftrightarrow p$ (double negation). This is a logical equivalence because the biconditional is always true, as shown in Table 13:

Table 13: Truth Table for Logical Equivalence: $\neg(\neg p) \leftrightarrow p$

p	$\neg(\neg p)$	$\neg(\neg p) \leftrightarrow p$
1	1	1
0	0	1

11.2.1 Example: Robbery Investigation

Problem statement: A robbery has occurred and it's known that the perpetrators fled in a car. Three known criminals (Par, Qun, and Rag) are interrogated. The police obtain the following information:

1. Par, Qun, and Rag are the only possible culprits
2. Rag never does a job without Par as an accomplice (doesn't exclude others)
3. Qun doesn't know how to drive

How can we identify the guilty persons?
Solution using Truth Tables:
Step 1: Define propositions
Let's represent each suspect with a proposition:

- P: Par participated in the robbery
- Q: Qun participated in the robbery
- R: Rag participated in the robbery

Step 2: Translate information into logical statements

1. At least one person participated (there was a robbery): $P \lor Q \lor R$
2. Rag never works without Par: $R \rightarrow P$
3. Qun doesn't know how to drive, someone must drive: $Q \rightarrow (P \lor R)$

Let C represent all constraints combined:

$$C = (P \vee Q \vee R) \wedge (R \to P) \wedge (Q \to (P \vee R))$$

Step 3: Build the truth table for all combinations

Table 14: Truth Table for Constraints

P	Q	R	$P \vee Q \vee R$	$R \to P$	$Q \to (P \vee R)$	C
0	0	0	0	1	1	0
0	0	1	1	0	1	0
0	1	0	1	1	0	0
0	1	1	1	0	1	0
1	0	0	1	1	1	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

Notice that $C = 1$ (constraints are satisfied) only when $P = 1$ in rows 5, 6, 7, and 8.

Step 4: Check which suspects must be guilty using logical implication

To determine who must be guilty, we check if $C \to X$ is a tautology for each suspect X:

Table 15: Testing Logical Implications

P	Q	R	C	$C \to P$	$C \to Q$	$C \to R$
0	0	0	0	1	1	1
0	0	1	0	1	1	1
0	1	0	0	1	1	1
0	1	1	0	1	1	1
1	0	0	1	1	0	0
1	0	1	1	1	0	1
1	1	0	1	1	1	0
1	1	1	1	1	1	1

Step 5: Conclusion

- $C \to P$ is a **tautology** (all values are 1) $\to$ **Par is definitely guilty**
- $C \to Q$ is **not** a tautology (contains 0) $\to$ Qun may or may not be guilty
- $C \to R$ is **not** a tautology (contains 0) $\to$ Rag may or may not be guilty

Answer: **Par is definitely guilty**. We cannot determine with certainty whether Qun and/or Rag participated. Since $C \to P$ is a tautology, P is a **logical consequence** of the constraints.

Alternative Solution using Logical Reasoning:

We can also solve this by analyzing scenarios:

Notice that in **all valid scenarios (where $C = 1$), Par must be true**:

- Par alone: $P \wedge \neg Q \wedge \neg R$ (row 5)
- Par and Rag: $P \wedge \neg Q \wedge R$ (row 6)
- Par and Qun: $P \wedge Q \wedge \neg R$ (row 7)
- All three: $P \wedge Q \wedge R$ (row 8)

We can prove Par's guilt by contradiction:

- Suppose $\neg P$ (Par didn't participate)
- From $R \to P$, by contrapositive: $\neg P \to \neg R$, so Rag didn't participate
- If only Qun participated ($Q \wedge \neg P \wedge \neg R$), no one could drive $\to$ impossible
- Therefore, P must be true

11.3 Propositional Logic - Detailed Concept Map

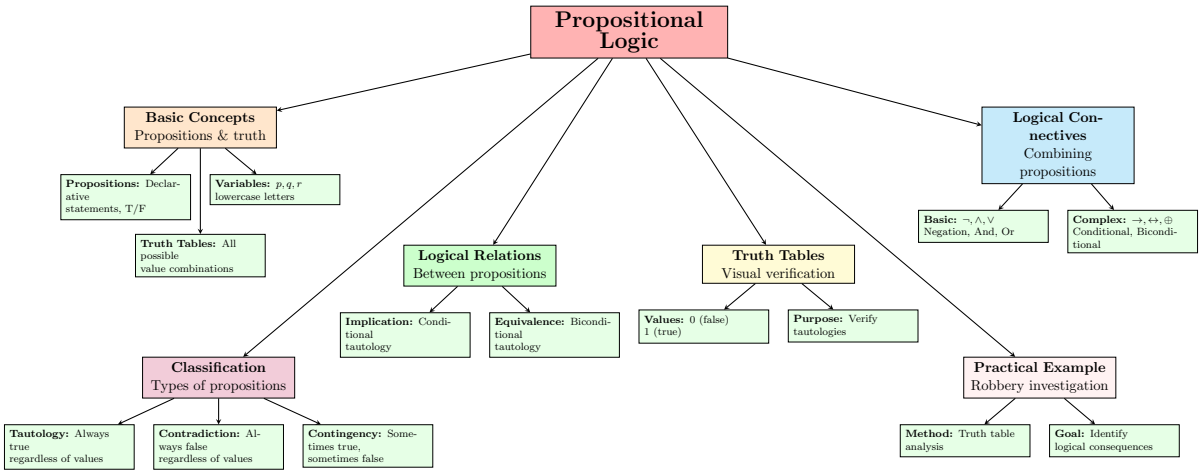

Figure 5: Detailed conceptual map of Propositional Logic

12 First-Order Logic

First-order logic (also called predicate logic) extends propositional logic by allowing quantification over individuals and the use of predicates, functions, and variables. It's more expressive than propositional logic and can represent relationships between objects and properties.

Key components:

- **Constants**: Specific individuals (e.g., $john$, $mary$)
- **Variables**: Placeholders for individuals (e.g., x, y, z)
- **Predicates**: Properties or relations (e.g., $Student(x)$, $Loves(x, y)$)
- **Functions**: Map individuals to individuals (e.g., $mother(x)$)

- **Quantifiers**:
  - $\forall x$ (universal): "For all x"
  - $\exists x$ (existential): "There exists an x"

Example: "All humans are mortal" can be expressed as:

$$\forall x \ (Human(x) \rightarrow Mortal(x))$$

This reads: "For all x, if x is a human, then x is mortal."
Another example: "Some students study" can be expressed as:

$$\exists x \ (Student(x) \wedge Studies(x))$$

This reads: "There exists an x such that x is a student and x studies."
More examples:
"Every mother loves her children" (using a binary relation):

$$\forall x \forall y \ (Mother(x, y) \rightarrow Loves(x, y))$$

This reads: "For all x and y, if x is the mother of y, then x loves y."
"All students have at least one teacher":

$$\forall x \ (Student(x) \rightarrow \exists y \ (Teacher(y) \wedge Teaches(y, x)))$$

This reads: "For all x, if x is a student, then there exists a y such that y is a teacher and y teaches x."
"Nobody loves everyone" (combining negation and quantifiers):

$$\neg \exists x \forall y \ Loves(x, y)$$

This reads: "There does not exist an x such that for all y, x loves y." Or equivalently: "For all x, there exists a y such that x does not love y."
"Everyone has a mother" (using a function):

$$\forall x \exists y \ (Mother(y, x))$$

This reads: "For all x, there exists a y such that y is the mother of x." Alternatively, using a function: $\forall x \ (Mother(mother(x), x))$ where $mother(x)$ is a function that returns the mother of x.
"All teachers who teach students are professors":

$$\forall x \ (Teacher(x) \wedge \exists y \ (Student(y) \wedge Teaches(x, y)) \rightarrow Professor(x))$$

This reads: "For all x, if x is a teacher and there exists a y such that y is a student and x teaches y, then x is a professor."

First-order logic is the foundation for many advanced logics, including Description Logic ALC, which is a specialized fragment of first-order logic designed for knowledge representation.

12.1 First-Order Logic - Detailed Concept Map

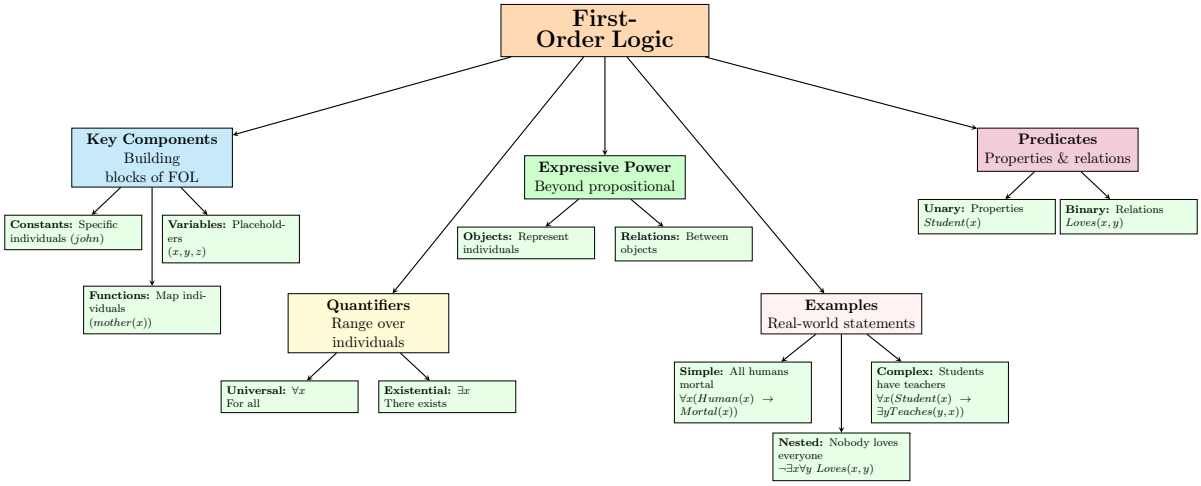

Figure 6: Detailed conceptual map of First-Order Logic

13 Description Logic ALC

Description logics are more than languages for formalizing concepts—they are a set of knowledge representation languages used to represent the terminological knowledge of a specific domain in a structured and formal way, ensuring clear understanding.

They are used to represent **ontologies** (formalization of a domain) and enable reasoning about them. They introduce new language and semantic elements necessary to formalize:

Description logics are particularly appropriate for the **semantic web** because they are useful for adding reasoning capabilities to the web. They have a formal syntax that allows describing:

- Important concepts of a universe or domain
- Relationships that arise or exist between them
- Constructors for building new concepts

Like all formal logics, they make it possible to reason based on knowledge that has been defined as such. Description logics are **variants of first-order logic**.

The formal characteristics of description logic include:

- **Ontology modeling**: They model ontologies, provide descriptions to domains, and formalize the elements of a terminology or descriptions of an ontology. The syntax and semantics have no ambiguity, as they are formal.
- **Descriptive formalism**: Uses three main components:
  - **Roles**: Represent relationships between individuals (e.g., "hasChild", "teaches")
  - **Constructors**: Build complex concepts from simpler ones (e.g., conjunction, disjunction, negation, quantifiers)
  - **Concepts**: Represent classes or categories of individuals (e.g., "Student", "Teacher", "Person")

13.1 ALC Building Blocks

ALC (Attributive Language with Complements) is built from atomic types and constructors that allow building complex concepts.

13.1.1 Atomic Types

- **Concept names**: $A, B, \ldots$ — Individual concept identifiers (e.g., "Student", "Teacher", "Person")
- **Special concepts**:
  - $\top$ — **Top** (universal concept): Represents the concept that encompasses all individuals
  - $\bot$ — **Bottom** (empty concept): Represents the empty or contradictory concept that encompasses no individuals
- **Role names**: $R, S, \ldots$ — Individual role identifiers representing relationships (e.g., "hasChild", "teaches", "loves")

13.1.2 Constructors

Constructors are used to build complex concepts from simpler ones:

- **Negation**: $\neg C$ — The negation of concept C (everything that is not C)
- **Conjunction**: $C \sqcap D$ — The intersection of concepts C and D (individuals that belong to both C and D)
- **Disjunction**: $C \sqcup D$ — The union of concepts C and D (individuals that belong to C or D or both)
- **Existential quantifier**: $\exists R.C$ — "There exists" — Individuals that have at least one R-relationship to an individual in concept C
- **Universal quantifier**: $\forall R.C$ — "For all" — Individuals where all R-relationships lead to individuals in concept C

13.1.3 Class Relationships

ALC allows expressing relationships between concepts:

- **Class Inclusion**: $C \sqsubseteq D$ — Concept C is a subclass of D (every C is a D)

  Example: $Professor \sqsubseteq FacultyMember$ means "every Professor is a Faculty Member"

  In first-order logic: $\forall x\ (Professor(x) \rightarrow FacultyMember(x))$
- **Class Equivalence**: $C \equiv D$ — Concepts C and D are equivalent (they represent the same set of individuals)

  Example: $Professor \equiv FacultyMember$ means "the Faculty Members are exactly the Professors"

  In first-order logic: $\forall x\ (Professor(x) \leftrightarrow FacultyMember(x))$

13.2 Quantifiers on Roles

Quantifiers can be used to bind the range of a role to a class, restricting what types of individuals can be related through that role. There are two main types of binding:

13.2.1 Strict Binding

Strict binding uses the universal quantifier ($\forall$) to require that **all** relationships through a role must satisfy a constraint.

Example: $Examination \sqsubseteq \forall hasSupervisor.Professor$

- **Natural language**: "An Examination must be supervised by a Professor" (all supervisors must be professors)
- **First-order logic**: $\forall x\,(Examination(x) \rightarrow (\forall y\,(hasSupervisor(x, y) \rightarrow Professor(y))))$

  This reads: "For all x, if x is an Examination, then for all y, if x has supervisor y, then y is a Professor."

Strict binding means that if an examination has any supervisor, that supervisor **must** be a professor. It doesn't require that an examination has a supervisor, but if it does, all supervisors must be professors.

13.2.2 Open Binding

Open binding uses the existential quantifier ($\exists$) to require that **at least one** relationship through a role exists and satisfies a constraint.

Example: $Examination \sqsubseteq \exists hasSupervisor.Person$

- **Natural language**: "Every Examination has at least one supervisor (who is a person)"
- **First-order logic**: $\forall x\,(Examination(x) \rightarrow (\exists y\,(hasSupervisor(x, y) \wedge Person(y))))$

  This reads: "For all x, if x is an Examination, then there exists a y such that x has supervisor y and y is a Person."

Open binding means that every examination **must have** at least one supervisor, and that supervisor must be a person. However, it doesn't restrict what other supervisors (if any) the examination might have.

13.3 ALC - Formal Syntax

The formal syntax of ALC defines how concepts (classes) can be constructed and how knowledge bases are structured.

13.3.1 Production Rules for Classes

The production rules define how complex classes can be built from atomic components. Let A be an atomic class, C and D be complex classes, and R be a role.

The formal grammar for creating classes in ALC is:

$$C, D ::= A \mid \top \mid \bot \mid \neg C \mid C \sqcap D \mid C \sqcup D \mid \exists R.C \mid \forall R.C$$

This means that a class C or D can be:

- An atomic class A
- The top concept $\top$ (universal concept)
- The bottom concept $\bot$ (empty concept)
- The negation of a class $\neg C$
- The conjunction of two classes $C \sqcap D$
- The disjunction of two classes $C \sqcup D$
- An existential restriction $\exists R.C$ (there exists an R-relationship to a C)
- A universal restriction $\forall R.C$ (all R-relationships lead to a C)

13.3.2 TBox (Terminological Box)

A **TBox** contains terminological knowledge—assertions about relationships between concepts (classes). It defines the vocabulary and structure of the domain.

An ALC TBox contains assertions of the form:

- $C \sqsubseteq D$ — Concept inclusion (subsumption): C is a subclass of D
- $C \equiv D$ — Concept equivalence: C and D are equivalent

where C and D are complex classes.
Examples:

- $Student \sqsubseteq Person$ — "Every student is a person"
- $Professor \equiv FacultyMember$ — "Professors and faculty members are the same"
- $Examination \sqsubseteq \forall hasSupervisor.Professor$ — "Examinations must be supervised by professors"

13.3.3 ABox (Assertional Box)

An **ABox** contains assertional knowledge—facts about specific individuals in the domain.

An ALC ABox contains assertions of the form:

- $C(a)$ — Concept assertion: Individual a belongs to class C
- $R(a, b)$ — Role assertion: Individual a is related to individual b through role R

where C is a complex class, R is a role, and a, b are individuals.
Examples:

- $Student(john)$ — "John is a student"
- $Professor(mary)$ — "Mary is a professor"
- $teaches(mary, john)$ — "Mary teaches John"
- $(Student \sqcap \exists hasSupervisor.Professor)(alice)$ — "Alice is a student who has a professor as supervisor"

Together, the TBox and ABox form a complete knowledge base that allows reasoning about both the structure of concepts and specific instances in the domain.

Curious Fact: What are Semantics?

Semantics refers to the meaning or interpretation of formal expressions. While **syntax** defines the structure and rules for forming valid expressions (like grammar rules), semantics assigns meaning to those expressions by interpreting them in a specific domain or model.
In logic, semantics provides a way to determine whether statements are true or false by mapping formal symbols to real-world objects, properties, and relationships. For example, in ALC, semantics tells us what it means for an individual to belong to a concept or for two individuals to be related through a role.
Real-world example: Consider the expression "$Student(john)$". The **syntax** tells us this is a valid ALC expression (a concept assertion). The **semantics** tells us what it means: we need an interpretation that maps "$john$" to an actual person (say, John Smith) and "$Student$" to the set of all students. Only then can we determine if the statement is true (is John Smith actually a student?) or false (is he not a student?). Without semantics, "$Student(john)$" is just a string of symbols with no meaning.
Model-theoretic semantics (used in ALC) defines meaning through interpretations that map formal expressions to sets and relations in a mathematical domain, allowing us to reason about truth and entailment.

13.4 ALC - Semantics (Interpretation)

ALC uses a **model-theoretic semantics**, meaning that entailment (logical consequence) is defined through interpretations. This provides a formal way to assign meaning to ALC expressions.

An **interpretation** $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of:

- **Domain** $\Delta^{\mathcal{I}}$: A non-empty set of individuals (the universe of discourse). This represents all the objects in the domain we're modeling.
- **Interpretation function** $\cdot^{\mathcal{I}}$: A function that maps:
  - **Individual names** a to domain elements: $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
  - **Class names** C to sets of domain elements: $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
  - **Role names** R to sets of pairs of domain elements: $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$

The interpretation function $\cdot^{\mathcal{I}}$ provides the semantic meaning by mapping abstract symbols (names) to concrete mathematical objects (elements, sets, and relations) in the domain.

Example: Consider a university domain with the following interpretation $\mathcal{I}$:

- **Domain**: $\Delta^{\mathcal{I}} = \{john, mary, alice, bob\}$
- **Individual names**:
  - $john^{\mathcal{I}} = john$ (the person John)
  - $mary^{\mathcal{I}} = mary$ (the person Mary)
  - $alice^{\mathcal{I}} = alice$ (the person Alice)
  - $bob^{\mathcal{I}} = bob$ (the person Bob)
- **Class names**:
  - $Student^{\mathcal{I}} = \{john, alice, bob\}$ (John, Alice, and Bob are students)
  - $Professor^{\mathcal{I}} = \{mary\}$ (Mary is a professor)
  - $Person^{\mathcal{I}} = \{john, mary, alice, bob\}$ (all individuals are persons)
- **Role names**:
  - $teaches^{\mathcal{I}} = \{(mary, john), (mary, alice)\}$ (Mary teaches John and Alice)
  - $hasSupervisor^{\mathcal{I}} = \{(alice, mary)\}$ (Alice has Mary as supervisor)

With this interpretation, we can determine the truth of assertions:

- $Student(john)$ is **true** because $john^{\mathcal{I}} \in Student^{\mathcal{I}}$
- $Professor(john)$ is **false** because $john^{\mathcal{I}} \notin Professor^{\mathcal{I}}$
- $teaches(mary, john)$ is **true** because $(mary^{\mathcal{I}}, john^{\mathcal{I}}) \in teaches^{\mathcal{I}}$
- $teaches(mary, bob)$ is **false** because $(mary^{\mathcal{I}}, bob^{\mathcal{I}}) \notin teaches^{\mathcal{I}}$

13.4.1 Semantic Interpretation of Constructors

The interpretation function extends to complex concepts built using constructors. The semantic interpretation rules are:

- **Top and Bottom concepts**:
  - $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$ — The top concept is interpreted as the entire domain
  - $\bot^{\mathcal{I}} = \emptyset$ — The bottom concept is interpreted as the empty set
- **Conjunction and Disjunction**:
  - $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$ — Disjunction is interpreted as set union
  - $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$ — Conjunction is interpreted as set intersection
- **Negation**:

- $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$ — Negation is interpreted as set difference (complement)

- **Universal quantifier**:
  - $(\forall R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid (\forall b \in \Delta^{\mathcal{I}})((a, b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}})\}$
    An individual a belongs to $\forall R.C$ if, for all individuals b related to a through role R, b belongs to concept C.

- **Existential quantifier**:
  - $(\exists R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid (\exists b \in \Delta^{\mathcal{I}})((a, b) \in R^{\mathcal{I}} \wedge b \in C^{\mathcal{I}})\}$
    An individual a belongs to $\exists R.C$ if there exists at least one individual b related to a through role R such that b belongs to concept C.

13.5 Properties

The following properties describe how individuals belong to concepts based on their relationships through roles:

- **Existential** ($\exists R.C$):

  An individual x belongs to $\exists R.C$ if there exists some value $y \in C$ such that $R(x, y)$.

  Example: An individual belongs to $\exists hasChild.Student$ if they have at least one child who is a student.

- **Universal** ($\forall R.C$):

  An individual x belongs to $\forall R.C$ if for all y, if $R(x, y)$, then $y \in C$.

  Example: An individual belongs to $\forall hasChild.Student$ if all their children are students.

- **Cardinality** ($= nR.C$):

  An individual x belongs to $(= nR.C)$ if there exist exactly n individuals $y \in C$ such that $R(x, y)$.

  Example: An individual belongs to $(= 2hasChild.Person)$ if they have exactly 2 children.

- **Maximum cardinality** ($\leq nR.C$):

  An individual x belongs to $(\leq nR.C)$ if there exist n or fewer individuals $y \in C$ such that $R(x, y)$.

  Example: An individual belongs to $(\leq 3hasChild.Person)$ if they have at most 3 children.

- **Minimum cardinality** ($\geq nR.C$):

  An individual x belongs to $(\geq nR.C)$ if there exist n or more individuals $y \in C$ such that $R(x, y)$.

  Example: An individual belongs to $(\geq 1hasChild.Student)$ if they have at least 1 child who is a student.

13.5.1 Property Attributes

Properties (relations/roles) can have the following attributes that describe how they behave. These attributes are important for understanding the structure and constraints of relationships in a knowledge base.

- **Reflexive** ($\forall x\ P(x,x)$):

  A property P is reflexive if every element is related to itself through P.

  Example: The relation "is equal to" is reflexive because every element is equal to itself: $\forall x\ (x = x)$. In a university domain, "hasSameAge" could be reflexive if we consider that everyone has the same age as themselves.

- **Irreflexive** ($\forall x\ \neg P(x,x)$):

  A property P is irreflexive if no element is related to itself through P.

  Example: The relation "is parent of" is irreflexive because no one is their own parent: $\forall x\ \neg Parent(x,x)$. Similarly, "teaches" is typically irreflexive (a professor doesn't teach themselves).

- **Symmetry** (If $P(x,y)$ then $P(y,x)$):

  A property P is symmetric if whenever x is related to y through P, then y is also related to x through P.

  Example: The relation "is sibling of" is symmetric because if x is a sibling of y, then y is a sibling of x: if $Sibling(x,y)$ then $Sibling(y,x)$. Another example: "is married to" is symmetric.

- **Asymmetry** (If $P(x,y)$ then $\neg P(y,x)$):

  A property P is asymmetric if whenever x is related to y through P, then y cannot be related to x through P.

  Example: The relation "is parent of" is asymmetric because if x is a parent of y, then y cannot be a parent of x: if $Parent(x,y)$ then $\neg Parent(y,x)$. Another example: "teaches" is typically asymmetric (if Mary teaches John, John doesn't teach Mary).

- **Transitivity** (If $P(x,y)$ and $P(y,z)$ then $P(x,z)$):

  A property P is transitive if whenever x is related to y and y is related to z through P, then x is also related to z through P.

  Example: The relation "is ancestor of" is transitive because if x is an ancestor of y and y is an ancestor of z, then x is an ancestor of z: if $Ancestor(x,y)$ and $Ancestor(y,z)$ then $Ancestor(x,z)$. Another example: "is part of" is transitive (if A is part of B and B is part of C, then A is part of C).

13.5.2 Relationships between Properties

Properties (relations) can have relationships with each other. These relationships define how different properties are related in the knowledge base.

- **Inverse** (P is inverse of $Q \Leftrightarrow P(x, y) \Leftrightarrow Q(y, x)$):

  A property P is the inverse of property Q if whenever x is related to y through P, then y is related to x through Q, and vice versa.

  Example: If P is "hasChild" and Q is "hasParent", then P is the inverse of Q because if $hasChild(x, y)$ (x has child y), then $hasParent(y, x)$ (y has parent x), and if $hasParent(y, x)$ (y has parent x), then $hasChild(x, y)$ (x has child y). So $hasChild(x, y) \Leftrightarrow hasParent(y, x)$. Another example: "teaches" and "isTaughtBy" are inverses: if $teaches(x, y)$ then $isTaughtBy(y, x)$.

- **Subproperty** (P is subproperty of Q if $P(x, y) \Rightarrow Q(x, y)$):

  A property P is a subproperty of property Q if whenever x is related to y through P, then x is also related to y through Q.

  Example: If P is "hasSon" and Q is "hasChild", then P is a subproperty of Q because if $hasSon(x, y)$ (x has son y), then $hasChild(x, y)$ (x has child y). However, the reverse is not necessarily true: if $hasChild(x, y)$, it doesn't mean $hasSon(x, y)$ (y could be a daughter). So $hasSon(x, y) \Rightarrow hasChild(x, y)$. Another example: "isMotherOf" is a subproperty of "isParentOf" because if someone is a mother of someone, they are also a parent of that person.

13.5.3 Functionality Properties

Properties can have a series of functionality properties that constrain how many values they can relate to.

- **Functional property** (If $P(x, y)$ and $P(x, z)$ then $y = z$):

  A property is functional if each individual can be related to at most one value through that property.

  Example: "hasMother" is functional because each person has exactly one biological mother. If $hasMother(x, y)$ and $hasMother(x, z)$, then $y = z$ (the same person).

- **Inverse functional property** (If $P(x, y)$ and $P(z, y)$ then $x = z$):

  A property is inverse functional if each value can be related to at most one individual through that property.

  Example: "hasSSN" (has Social Security Number) is inverse functional because each SSN belongs to exactly one person. If $hasSSN(x, y)$ and $hasSSN(z, y)$ (both x and z have the same SSN y), then $x = z$ (they are the same person).

- **Keys** (If $P(x, y)$ and $P(z, y)$ then $x = z$):

  Keys are properties that uniquely identify individuals. They have the same constraint as inverse functional properties. Keys are often used in databases and knowledge bases to uniquely identify entities.

  Example: "hasEmail" can be a key if each email address belongs to exactly one person. If $hasEmail(x, y)$ and $hasEmail(z, y)$ (both x and z have the same email y), then $x = z$ (they are the same person).

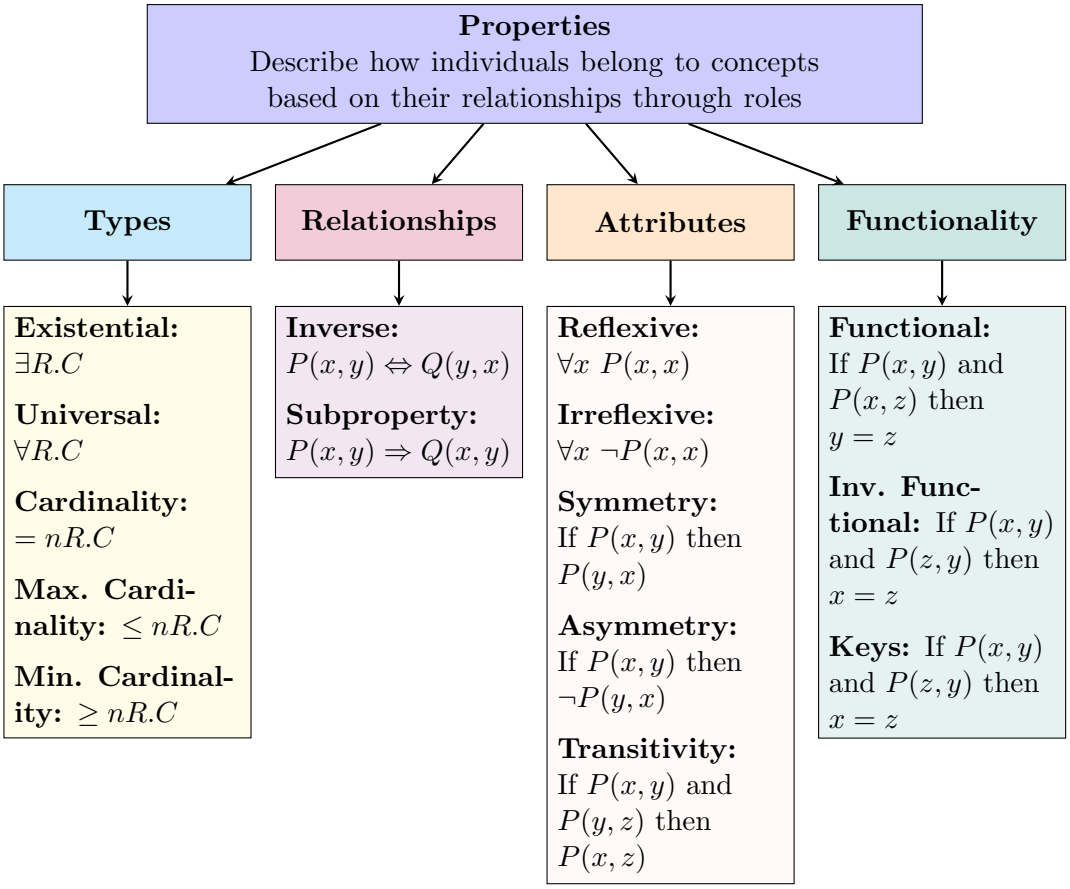

Figure 7: Concept map of Properties

13.6 Reasoning

From a knowledge base, reasoning (inference) can be performed. Different types of reasoning tasks allow us to extract information and answer questions about the domain.

Table 16: Types of Reasoning

Type of Reasoning	Description
Concept Satisfaction	From Σ it is not deduced that $C \equiv \bot$
Subsumption	$\Sigma \Rightarrow C \sqsubseteq D$
Instantiation	$\Sigma \Rightarrow a \in C$
Information Retrieval	Given a concept C, obtain individuals a such that $a \in C$
Comprehension	Given an individual a, obtain the most specific concept C such that $a \in C$

Explanation of each type:

- **Concept Satisfaction**: Checks if a given concept C is satisfiable within the knowledge base Σ, meaning it's not equivalent to the bottom concept ($\bot$), which repre-

sents an inconsistent or empty concept. If a concept is unsatisfiable, it means there can be no individuals that belong to it.

Example: Check if the concept $Student \sqcap \neg Person$ is satisfiable. If the knowledge base states that all students are persons, then this concept is unsatisfiable (equivalent to $\bot$).

- **Subsumption**: Determines if concept C is subsumed by concept D within the knowledge base Σ, meaning all instances of C are also instances of D. This checks if C is a subclass of D.

  Example: Check if $Student \sqsubseteq Person$ holds in the knowledge base. If true, it means every student is also a person.

- **Instantiation**: Checks if an individual a is an instance of concept C according to the knowledge base Σ. This verifies whether a specific individual belongs to a concept.

  Example: Check if $Student(john)$ holds, i.e., whether John is a student according to the knowledge base.

- **Information Retrieval**: Given a concept C, retrieves all individuals a such that $a \in C$. This is a query that finds all instances of a concept.

  Example: Query "find all students" would return all individuals that belong to the $Student$ concept, such as $\{john, alice, bob\}$.

- **Comprehension**: Given an individual a, finds the most specific concept(s) C such that $a \in C$. This determines what concepts an individual belongs to, focusing on the most specific ones.

  Example: Given individual $john$, find the most specific concepts he belongs to. If John is a graduate student, the result might be $\{GraduateStudent, Student, Person\}$, but the most specific would be $GraduateStudent$.

13.7 Description Logic ALC - Summary

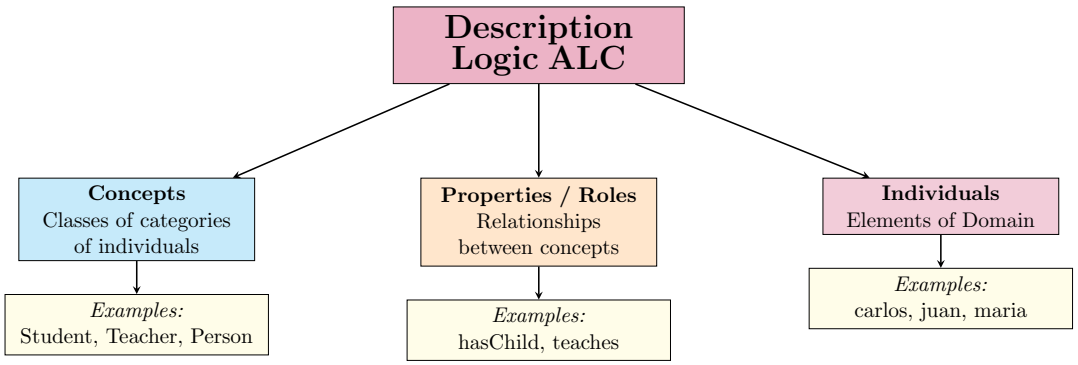

Figure 8: Summary concept map of Description Logic ALC

14 Higher-Order Logic

Higher-order logic (or second-order logic) is an extension of first-order logic that adds variables for properties, functions, and relations, along with quantifiers that operate over

those variables. This expands the expressive power of the language without having to add new logical symbols.

The need for second-order logic is reflected in Giuseppe Peano's induction axiom for arithmetic, which requires quantifiers that can range not only over concrete elements (like numbers), but also over relations and functions (like properties of numbers).

Key distinction:

- **First-order logic**: Quantifiers only apply to objects. Example: "All students are smart" — $\forall x(Student(x) \to Smart(x))$
- **Higher-order logic**: Quantifiers can also apply to predicates/properties. Example: "All properties that hold for 0 and are preserved by succession hold for all numbers" (mathematical induction).

Second-order logic has greater expressive power than first-order logic, allowing the formalization of complex mathematical systems that cannot be captured in first-order logic.

Types of Second-Order Logic:

- **Monadic Second-Order Logic (MSOL)**: Only allows quantification over unary predicates (sets).

  Example: "There exists a group of people who all speak Spanish" — $\exists Group \forall p(p \in Group \to SpeaksSpanish(p))$. Here, $Group$ is a variable representing a set of people.

- **Full Second-Order Logic (FSOL)**: Allows quantification over predicates of any arity (unary, binary, etc.) and functions.

  Example: "There exists a relation that is symmetric" — $\exists R \forall x \forall y(R(x, y) \to R(y, x))$. Here, R is a variable representing a binary relation, like "is friend of".

Table 17: First-Order vs Second-Order Logic

Aspect	**First-Order Logic**	**Second-Order Logic**
Quantifiers range over	Objects/individuals	Objects and predicates/relations
Expressive power	Limited	Greater
Completeness	Complete	Incomplete
Example	$\forall x P(x)$ (all objects satisfy P)	$\forall P \forall x P(x)$ (all properties hold for all objects)

14.1 Syntax of Second-Order Logic

Given a vocabulary $\mathcal{L}$, second-order logic (SOL) over $\mathcal{L}$ is defined as an extension of first-order logic that includes the following rules:

Formation Rules:

- **Rule 1**: If $t_1, \ldots, t_k$ are $\mathcal{L}$-terms and X is a second-order variable of arity k (that is, a relation with $k \geq 1$ arguments), then $X(t_1, \ldots, t_k)$ is a formula in SOL.

  Explanation: We can apply second-order variables (which represent predicates/relations) to first-order terms.

  Example: If X is a second-order variable of arity 2 (binary relation) and $john$, $mary$ are constants, then $X(john, mary)$ is a valid SOL formula. This could represent "there exists some relation X between John and Mary" (e.g., X could be "knows", "likes", etc.).

- **Rule 2**: If φ is a formula in SOL and X is a second-order variable of arity k, then $\exists X\varphi$ and $\forall X\varphi$ are formulas in SOL.

  Explanation: We can quantify over second-order variables (predicates/relations).

  Examples:

  - $\exists X \forall x(Student(x) \rightarrow X(x))$ — "There exists a property X that all students have." (X could be "intelligent", "young", etc.)
  - $\forall R \forall x \forall y(R(x, y) \rightarrow R(y, x))$ — "For all binary relations R, if $R(x, y)$ holds, then $R(y, x)$ holds." This describes only symmetric relations.
  - $\exists P(P(alice) \wedge \neg P(bob))$ — "There exists a property P that Alice has but Bob doesn't."

Note: The key difference from first-order logic is that in SOL, variables can represent not just objects (like people, numbers), but also predicates and relations (like "is tall", "is greater than", "knows"). This allows us to make statements about properties and relations themselves, not just about individual objects.

14.2 Semantics of Second-Order Logic

Given a structure $\mathfrak{A}$ with domain A, an assignment σ is a function that assigns:

- **A value in A to each first-order variable** x: $\sigma(x) \in A$

  This is the same as in first-order logic: each variable representing an object gets assigned a concrete object from the domain.

  Example: If $A = \{john, mary, alice\}$ (set of people), then $\sigma(x) = john$ assigns the person John to variable x.

- **A subset of A^k to each second-order variable X with k arguments**: $\sigma(X) \subseteq A^k$

  Each second-order variable (representing a predicate or relation) gets assigned a set of tuples. For a unary predicate ($k = 1$), this is a subset of A. For a binary relation ($k = 2$), this is a subset of $A \times A$ (pairs of elements).

  Examples:

  - If P is a unary second-order variable (property), then $\sigma(P) = \{john, alice\} \subseteq A$ means that the property P holds for John and Alice, but not for Mary.

    For instance, if P represents "speaks Spanish", this assignment says John and Alice speak Spanish, but Mary doesn't.

– If R is a binary second-order variable (relation with 2 arguments), then $\sigma(R) = \{(john, mary), (alice, john)\} \subseteq A \times A$ means that R relates John to Mary and Alice to John.

  For instance, if R represents "knows", this assignment says John knows Mary, and Alice knows John.

Key insight: In first-order logic, we only assign concrete objects to variables. In second-order logic, we also assign sets of tuples to second-order variables, which represent the extensions of predicates and relations. This allows us to quantify over all possible predicates and relations, giving SOL its greater expressive power.

14.3 Additional Semantic Cases for SOL

The definition of SOL includes three additional cases for evaluating formulas with second-order variables:

For a second-order variable X with arity k:

1. **Application of second-order variable**:

$$(\mathfrak{A}, \sigma) \models X(t_1, \ldots, t_k) \text{ if and only if } (\sigma(t_1), \ldots, \sigma(t_k)) \in \sigma(X)$$

   Meaning: The formula $X(t_1, \ldots, t_k)$ is satisfied if the tuple of values assigned to the terms $(t_1, \ldots, t_k)$ belongs to the set assigned to the second-order variable X.

   Example: Suppose X is a binary relation, $\sigma(X) = \{(john, mary), (alice, bob)\}$, and $\sigma(t_1) = john$, $\sigma(t_2) = mary$. Then $(\mathfrak{A}, \sigma) \models X(t_1, t_2)$ because $(john, mary) \in \sigma(X)$.

   If we think of X as "knows", this says "John knows Mary" is true in this interpretation.

2. **Existential quantification over second-order variables**:

$$(\mathfrak{A}, \sigma) \models \exists X \varphi \text{ if and only if there exists } S \subseteq A^k \text{ such that } (\mathfrak{A}, \sigma[X/S]) \models \varphi$$

   Meaning: The formula $\exists X \varphi$ is satisfied if there exists some subset S of A^k such that when we assign S to X, the formula φ becomes true. Here, $\sigma[X/S]$ denotes the assignment that is identical to σ except that it assigns S to X.

   Example: Consider $\exists P \forall x(Student(x) \rightarrow P(x))$ — "There exists a property P that all students have."

   This is satisfied if we can find some set $S \subseteq A$ (e.g., $S = \{john, mary, alice, bob\}$ representing "all people") such that when we assign P to this set, every student is in that set.

3. **Universal quantification over second-order variables**:

$$(\mathfrak{A}, \sigma) \models \forall X \varphi \text{ if and only if for every } S \subseteq A^k, (\mathfrak{A}, \sigma[X/S]) \models \varphi$$

   Meaning: The formula $\forall X \varphi$ is satisfied if for every possible subset S of A^k, when we assign S to X, the formula φ is true.

Example: Consider $\forall P(P(john) \rightarrow P(mary))$ — "For all properties, if John has the property, then Mary also has it."

This is satisfied if, no matter which property we choose (tall, smart, Spanish-speaking, etc.), whenever John has it, Mary has it too. This would only be true if Mary has all the properties that John has.

Summary: These three cases extend the standard first-order semantics to handle second-order variables. The key difference is that quantifiers in SOL range over sets (subsets of A^k), not just individual elements, allowing us to express statements about all possible properties and relations.

15 Multivalued Logic and Fuzzy Logic

15.1 Multivalued Logic

Multivalued logic is a logic that allows intermediate values (large, warm, far, few, many, etc.) and uses more than two truth values to describe concepts that go beyond true and false. Multivalued logics provide conceptual tools that make it possible to formally describe fuzzy, vague, or uncertain information.

Key characteristics:

- Uses more than two truth values (unlike classical logic which only has true/false)
- Allows for degrees of truth (e.g., 0.0 to 1.0, or linguistic values like "very true", "somewhat true", "false")
- Suitable for representing imprecise or uncertain information

Example: Instead of saying "The temperature is hot" (true/false), multivalued logic allows us to say "The temperature is 0.8 hot" or "The temperature is somewhat hot", representing degrees of truth.

15.2 Fuzzy Logic

Fuzzy logic (also called fuzzy logic) is a multivalued logic that allows the mathematical representation of uncertainty and vagueness, providing formal tools for their treatment. Lotfi A. Zadeh is considered the father of fuzzy logic. His career focused on work on fuzzy sets and the application of fuzzy logic in approximate reasoning. The term "fuzzy logic" first appeared in 1974.

Curious Fact: Zadeh's Principle

Zadeh's Principle (1973): "As complexity increases, precise statements lose their meaning and useful statements lose precision."
This can be summarized as "you can't see the forest for the trees" — when dealing with complex systems, trying to be too precise can make statements meaningless, while useful statements often need to sacrifice some precision to remain comprehensible and applicable.

Key insight: The model of characterizing a problem through fuzzy logic is based on the prerogative that the mapping between concepts is done through semantics, not numerical precision. It is very suitable for modeling problems from expert knowledge, which normally details their knowledge base in the form of imprecise expressions.

Applications in Artificial Intelligence:

- Handling reasoning under uncertainty and with imprecise notions
- Management of databases and knowledge-based systems when information is known to be imprecise
- Automation of data mining techniques, which are often linked to fuzzy or multivalued sets
- Automatic reasoning methods for these logics

15.3 Example: Air Conditioning System

Imagine a fuzzy system controlling an air conditioner that regulates temperature according to needs.

Inputs: The fuzzy chips of the air conditioner collect input data, which in this case could be simply:

- Temperature
- Humidity

Processing: These data are subject to the rules of the inference engine (in the form IF... THEN...), which derives a results area.

Example rules:

- IF temperature is *very hot* AND humidity is *high*, THEN cooling is *maximum*
- IF temperature is *warm* AND humidity is *moderate*, THEN cooling is *moderate*
- IF temperature is *cool* AND humidity is *low*, THEN cooling is *minimum*

Output: From the results area, the center of gravity is chosen, providing it as an output. According to the result, the air conditioner could increase or decrease the temperature based on the output degree.

Key advantage: Unlike traditional systems that use precise thresholds (e.g., "if temperature ¿ 25°C, turn on"), fuzzy systems handle gradual transitions and imprecise concepts naturally, making them more suitable for human-like reasoning and expert knowledge representation.

16 Summary - Lecture 003

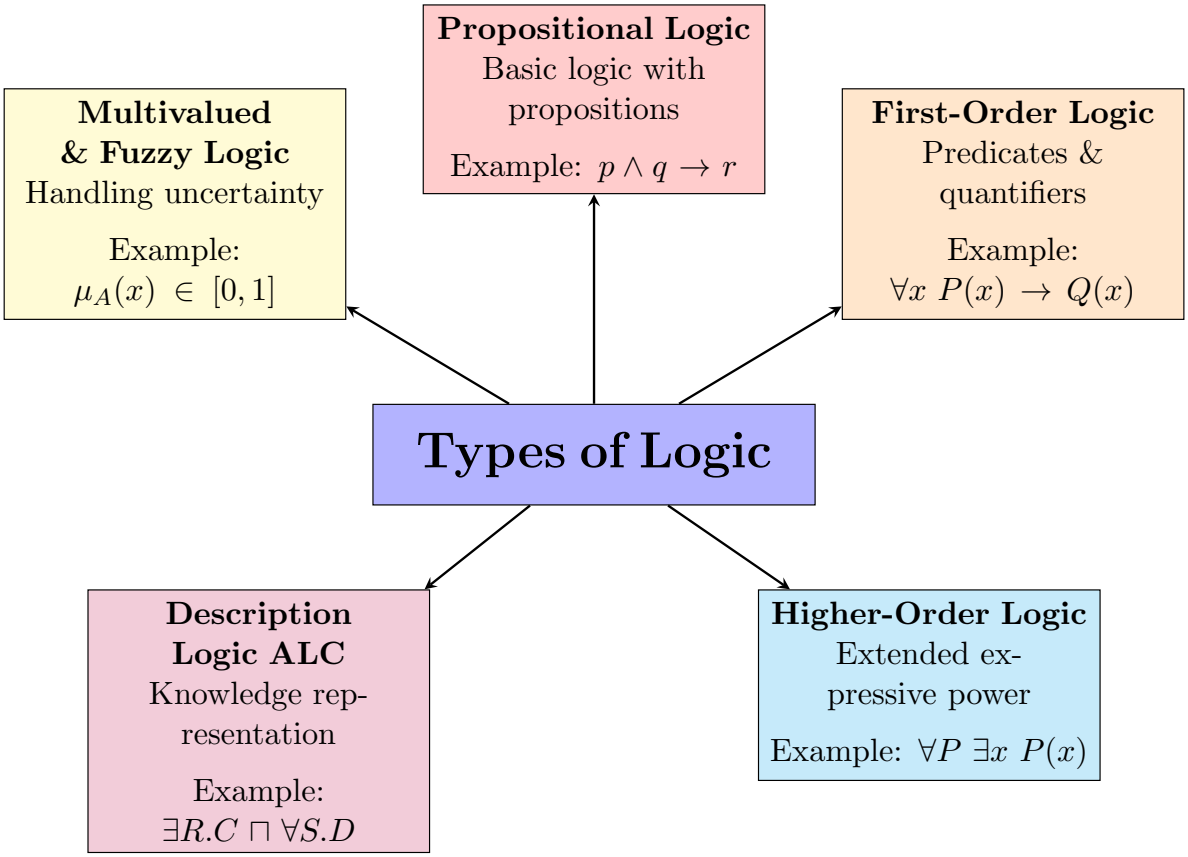

Figure 9: Conceptual map of the Reasoning and Planning course - Lecture 003

Lecture 004

17 Overview of a Search Problem

Search problems can be classified according to two main criteria: the use of heuristics and the execution timing. These classifications help us understand the characteristics and behavior of different search algorithms.

17.1 Uninformed vs. Informed Search

The first classification depends on whether the algorithm uses a **heuristic function** (a function that estimates the cost or distance to the goal):

- **Uninformed search**: Algorithms that do not use any heuristic information. They explore the search space systematically without prior knowledge about which states are more promising. Each state is evaluated equally, without knowing if it brings us closer to the goal.

- **Informed search**: Algorithms that employ a heuristic function to guide the search process. The heuristic provides an estimate of how close a state is to the goal, allowing the algorithm to prioritize more promising paths and typically find solutions more efficiently.

17.2 Offline vs. Online Search

The second classification relates to when the agent executes actions relative to the search process:

- **Offline search**: The agent first completes the entire search process to find a complete solution plan (from initial state to goal), and only then begins executing the actions. This approach is typical of deliberative agents that have sufficient time to plan before acting.
- **Online search**: The agent interleaves search and execution—it searches for a short-term plan, executes some actions, then searches again from the new state. This approach is used by reactive agents operating under time constraints or in dynamic environments. Typically employs local search algorithms that find partial solutions incrementally.

18 Search-Based Agents

Search-based agents are intelligent agents that use search algorithms to find sequences of actions that achieve their goals. These agents operate by maintaining an internal model of their environment and systematically exploring possible action sequences.

18.1 Key Characteristics

Search-based agents are characterized by two fundamental properties:

- **Symbolic environment model**: They maintain a symbolic representation of the environment that captures only the information relevant to the problem at hand. This model defines the parameters that distinguish one state from another, allowing the agent to reason about different configurations of the environment.
- **Goal-oriented state modification**: They aim to modify the environment state according to their objectives by applying actions that transform the current state into a goal state—one that satisfies the agent's objectives.

18.2 How They Work

To achieve their goals, search-based agents follow a systematic approach:

1. **Action anticipation**: Using their environment model, they predict the effects that their actions would have on the world. This allows them to simulate different action sequences without actually executing them.

2. **Plan generation**: Through a search process, they explore possible sequences of actions that lead from the current state to a goal state. The search algorithm systematically evaluates different paths through the state space.

3. **Plan execution**: Once a valid sequence is found, the agent executes the plan to transform the environment from its current state to the desired goal state.

The search process is central to these agents—it is the mechanism that allows them to find the appropriate sequence of actions to achieve their objectives.

18.3 Deliberative vs. Reactive Search-Based Agents

The timing of search and execution differs based on the agent type:

- **Deliberative agents**: Complete the entire search process *before* executing any actions. Once a solution plan is found, execution begins. This approach minimizes the risk of costly mistakes.

- **Reactive agents**: Interleave search and execution—they search while executing actions. This carries the risk of making mistakes that may require undoing costly actions.

Note: Both deliberative and reactive agents can use either **informed** or **uninformed** search algorithms.

19 Domain-Specific Algorithms

Domain-specific algorithms are problem-solving techniques where the agent designer encodes a known method for solving problems in a particular domain into a specialized algorithm.

```
PROCEDURE MoverDiscos(n:integer;
                      origen,destino,auxiliar:char);
{ Pre: n > 0
  Post: output = [movimientos para pasar n
                  discos de la aguja origen
                  a la aguja destino] }
BEGIN
  IF n = 0 THEN  {Caso base}
    writeln
  ELSE BEGIN     {Caso recurrente}
    MoverDiscos(n-1,origen,auxiliar,destino);
    write('Pasar disco',n,'de',origen,'a',destino);
    MoverDiscos(n-1,auxiliar,destino,origen)
  END; {fin ELSE}
END; {fin MoverDiscos}
```

Figure 10: Example of a domain-specific algorithm: Recursive solution for the Towers of Hanoi problem. This algorithm is specifically designed for this puzzle and cannot be directly applied to other domains.

19.1 How They Work

- The designer identifies a solution method for a specific problem domain.
- This method is encoded into a specialized algorithm that can solve any problem instance within that domain.
- Flexibility can be improved by using **parameters** that configure the problem, allowing the same code to handle different problem instances within the domain.

19.2 Limitations

The main drawbacks of this approach are:

- **Exhaustive anticipation required**: The designer must anticipate all possible scenarios the agent might encounter. In real-world environments, this is often too complex to achieve.

- **Domain restriction**: The algorithm only works for the specific domain it was designed for. It cannot be applied to problems outside that domain without significant modification.

20 Domain-Independent Methods

Domain-independent methods use a symbolic model of the domain and problem to solve tasks. Unlike domain-specific algorithms, they employ generic search algorithms that work across different problem domains.

20.1 Problem Representation

For example, in the Towers of Hanoi problem, we would define the problem in general terms as follows:

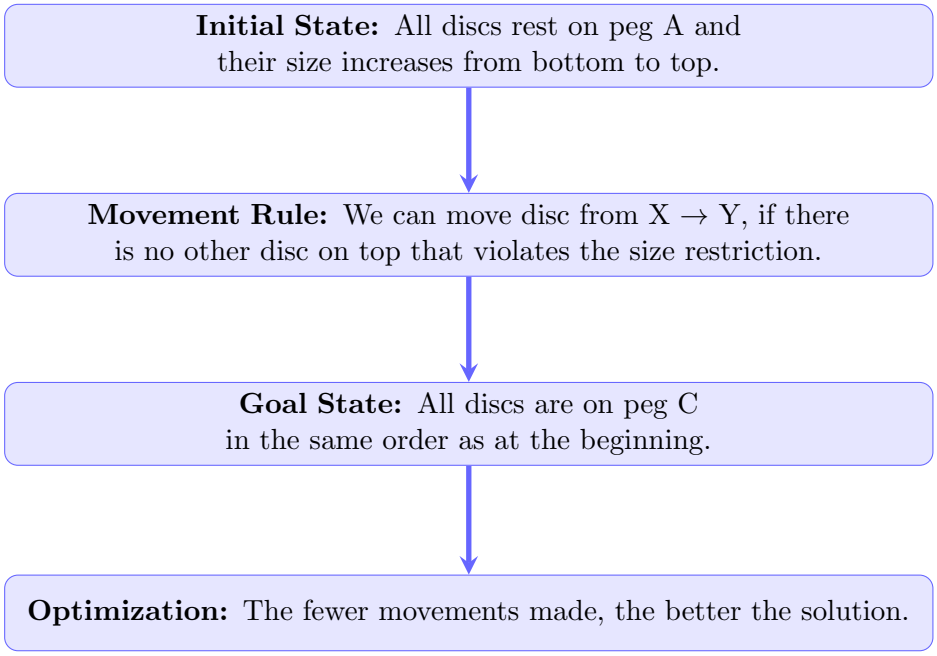

Figure 11: Symbolic representation of the Towers of Hanoi problem using domain-independent methods.

20.2 Advantages

To solve problems, this method employs a **generic search algorithm**, representing the domain and problem through the symbolic model. This approach offers several advantages:

- **Greater flexibility**: We do not need to know the solution beforehand. The search algorithm explores the solution space automatically.
- **Easy extensibility**: It is straightforward to add new features or constraints to the problem by modifying the symbolic model, without changing the search algorithm itself.

- **Reusability**: The same generic search algorithm can be applied to different problem domains by simply changing the symbolic representation.

21 State-Space Search Problems

In state-space search problems, the environment is represented through **states** that must be uniquely distinguishable from each other, but lack accessible characteristics that allow direct differentiation. For the agent, they are simply different "labels" representing distinct states.

21.1 Problem Components

This type of problem is defined by three essential elements:

- **State space**: A model of the world represented by a graph, where nodes represent components of the world. These components are translated into elements of the symbolic model, symbolized in a specific way within the graph.
- **Search problem**: Through a problem-independent mechanism, we explore the state space by applying the agent's attitude, which represents the rationality component in the exploration process.
- **Objective**: To find the most efficient plan that leads from the initial state to a goal state.

21.2 The Search Challenge

Therefore, this type of problem aims to find the best path within a directed graph, as shown in Figure 12. However, we do not know the graph structure in advance. If we knew it, we could perform a minimum path search using algorithms like Dijkstra's algorithm.

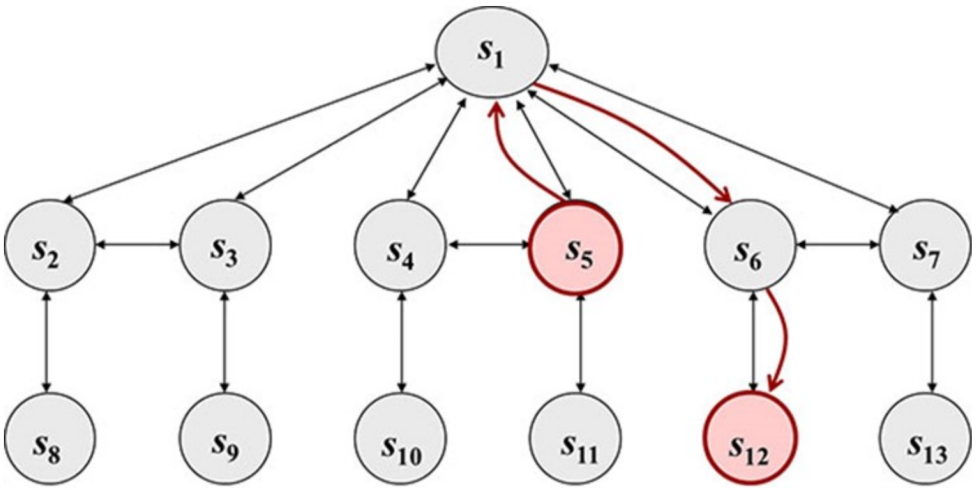

Figure 12: Example of a state-space search graph. The red nodes and path highlight a possible solution path from the initial state (s1) to a goal state (s12).

Since the graph is unknown, we must explore it incrementally through search algorithms, discovering states and transitions as we explore the state space.

21.3 The Search Process

The search method operates through an iterative cycle, as illustrated in Figure 13:

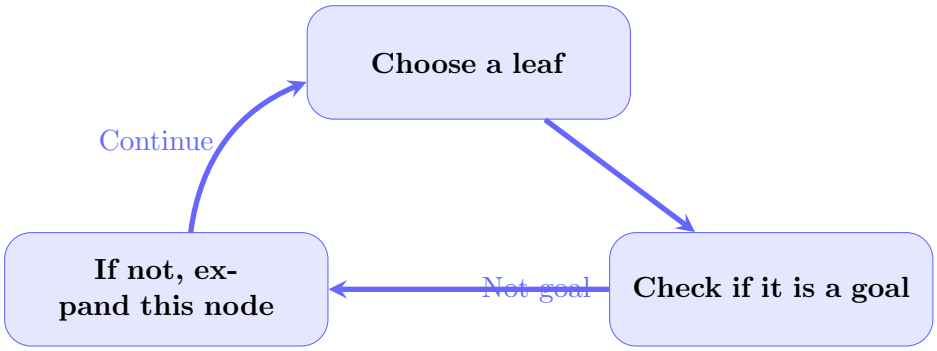

Figure 13: Search method based on the selection of a leaf representing a state.

The process repeats until a goal state is found. During this exploration, we build a **search tree** that represents the states we have discovered and explored, as shown in Figure 14.

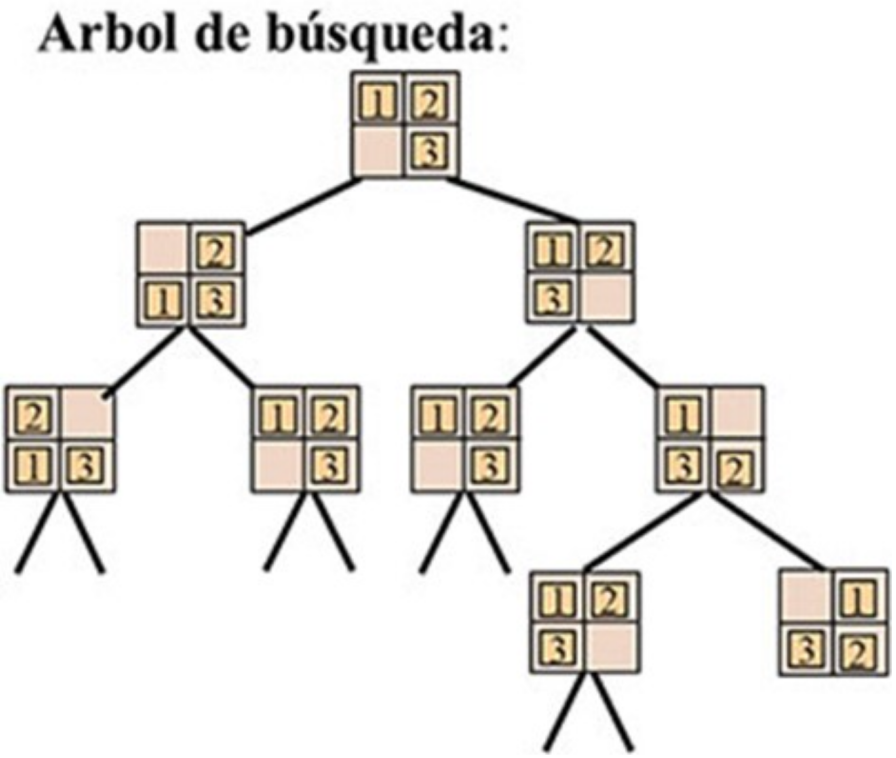

Figure 14: Example of a search tree showing the exploration of states. Each node represents a state configuration, and edges represent transitions between states.

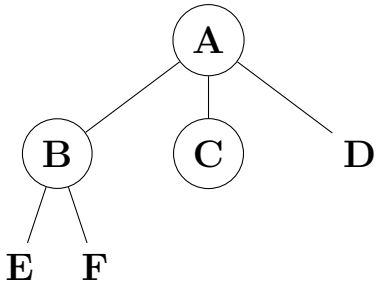

Figure 15: Nomenclature for graph traversal in search algorithms. **Notation:** When expanding, we note the successors from left to right, but the algorithm does not always expand them in that order (another algorithm could have chosen D before, and would then continue on that side of the figure).

It is a particular way of traversing a graph (it will be different for each algorithm we will see).

- **A** is a node with initial state (expanded) with three successors B, C, D.
- **B** is a node with state B, expanded, with two successors.
- **C** is a node with state C, expanded, without successors.
- **D** is a node pending expansion (in open list).
- **E** is a node pending expansion (in open list).
- **F** is a node pending expansion (in open list).

21.4 General Search Algorithm

We can define a general search algorithm as shown in the following Python implementation:

```
def general_search(initial_state, goal_state):
    """
␣␣␣␣General␣search␣algorithm␣for␣state-space␣problems.

␣␣␣␣Args:
␣␣␣␣␣␣␣␣initial_state:␣The␣starting␣state␣S0
␣␣␣␣␣␣␣␣goal_state:␣The␣goal␣state␣G

␣␣␣␣Returns:
␣␣␣␣␣␣␣␣Path␣from␣initial_state␣to␣goal_state,␣or␣None␣if␣no␣
   solution␣exists
␣␣␣␣"""
    # Line 1: Initialize open list with initial state
    open_list = [Node(initial_state, parent=None)]

    # Line 2: Main search loop
    while open_list:  # While open_list is not empty
        # Line 3: Extract first node from open list
```

```
18          node = open_list.pop(0)  # FIFO for BFS
19
20          # Line 4-5: Check if node is goal
21          if is_goal(node.state, goal_state):
22              # Return path by backtracking through parent nodes
23              return reconstruct_path(node)
24
25          # Line 7: Expand node to get successors
26          successors = expand(node.state)
27
28          # Line 8-10: Add successors to open list
29          for successor_state in successors:
30              successor_node = Node(successor_state, parent=node)
31              open_list.append(successor_node)
32
33      # Line 13: No solution found
34      return None
35
36  class Node:
37      """Node representation in the search tree."""
38      def __init__(self, state, parent=None):
39          self.state = state  # State reached at this point
40          self.parent = parent  # Reference to parent node (line 9)
```

Listing 1: General search algorithm

21.4.1 Algorithm Components

The search tree is represented using a **Node** record type. In its simplest form, a node is linked to its predecessor (parent) through a reference (line 112 in the code, where the parent is set in the Node constructor). Each node stores the state reached at that point in the exploration.

- **Open list** (line 85): A list of nodes containing the current leaves of the tree—those states and paths that have been expanded for exploration.
- **Empty list check** (line 88): If the list is empty, we have encountered a problem with no solution. Otherwise, we extract the first element from the list of open leaves (line 90).
- **Goal check** (lines 93-95): If the node is a goal state, we return the path by performing backtracking through the parent nodes of the found goal node.
- **Expansion** (lines 98, 101-103): We expand the node to get successors (line 98), then add the new states obtained from expansion to the open list (lines 101-103).

In the following sections, we will show how different uninformed search algorithms work, which are used as the foundation for many intelligent agent problems.

21.4.2 Repeated States Problem

In all search mechanisms, we face a potential problem: **repeated states**. This can cause serious errors in some cases, such as entering infinite loops. We have different strategies to resolve this:

- **Ignore it**: As strange as it may seem, some algorithms do not have problems with this solution due to their own exploration order.
- **Avoid simple cycles**: Prevent adding the parent of a node to the set of successors.
- **Avoid general cycles**: Prevent any ancestor of a node from being added to the set of successors.
- **Avoid all repeated states**: Do not allow adding any node that already exists in the tree to the set of successors.

These strategies must consider the cost of both exploring too much and searching for repeated elements to explore less. The choice depends on the specific problem and the trade-off between memory usage and computational efficiency.

21.5 Algorithm Classification Criteria

In the following sections, we will present different search algorithms. For all of them, we will use a classification mechanism based on the following concepts and characteristics:

21.5.1 Completeness

An algorithm is **complete** if it is guaranteed to find a solution whenever one exists. If no solution exists, a complete algorithm will correctly report that no solution was found.

Example: Imagine you're searching for your keys in your house. A complete search method would guarantee that if your keys are somewhere in the house, you will eventually find them. If you search room by room systematically, you're using a complete method. However, if you only check the living room and give up, that's an incomplete method—you might miss the keys if they're in the bedroom.

21.5.2 Optimality

An algorithm is **optimal** if, when multiple solutions exist, it always finds the best one according to some cost measure (e.g., shortest path, minimum number of steps, lowest cost).

Example: Consider finding a route from your home to a restaurant. There might be three routes: Route A (5 km, 10 minutes), Route B (8 km, 8 minutes), and Route C (12 km, 15 minutes). An optimal algorithm would find Route B if you want the fastest route, or Route A if you want the shortest distance. A non-optimal algorithm might find Route C first and stop there, even though better options exist.

21.5.3 Time Complexity

Time complexity measures how long an algorithm takes to find a solution, typically expressed in terms of the problem size (e.g., number of nodes, depth of the search tree).

Example: Think of searching for a specific book in a library. If the library has 1,000 books and you check them one by one, you might need to check all 1,000 in the worst case. If the library has 10,000 books, you might need to check all 10,000. The time complexity describes how the search time grows as the library size increases. Some algorithms might need to explore exponentially more states as the problem grows, while others scale more efficiently.

21.5.4 Space Complexity

Space complexity measures how much memory an algorithm requires to find a solution. This includes the storage needed for the open list, closed list, and other data structures.

Example: Imagine you're exploring a maze and want to remember all the paths you've tried. A method that stores every path you've explored requires more memory than one that only remembers the current path. If the maze is very large, storing all explored paths might require more memory than your computer has available. Space complexity helps us understand these memory requirements and choose algorithms that fit within available resources.

22 Breadth-First Search

Breadth-First Search (BFS) is a search strategy that generates the search tree level by level, expanding all nodes at depth level i before expanding nodes at level $i + 1$.

BFS considers, first, all states that are reachable in paths of length 1 (i.e., paths requiring only one action), then those of length 2 (paths requiring two actions), and so on. In this way, it finds the goal state at the minimum depth.

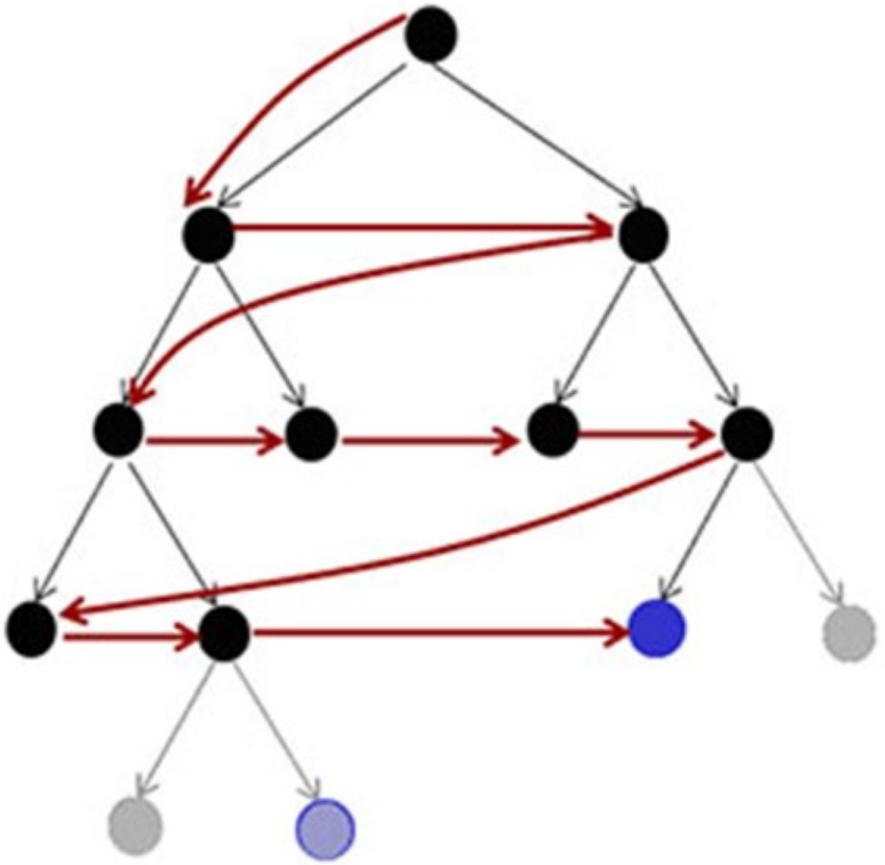

Figure 16: Breadth-first search exploration schema. The algorithm explores all nodes at each depth level before moving to the next level.

The algorithm develops a search mechanism that, for example, in the 8-puzzle problem, would result in the search tree shown in Figure 17.

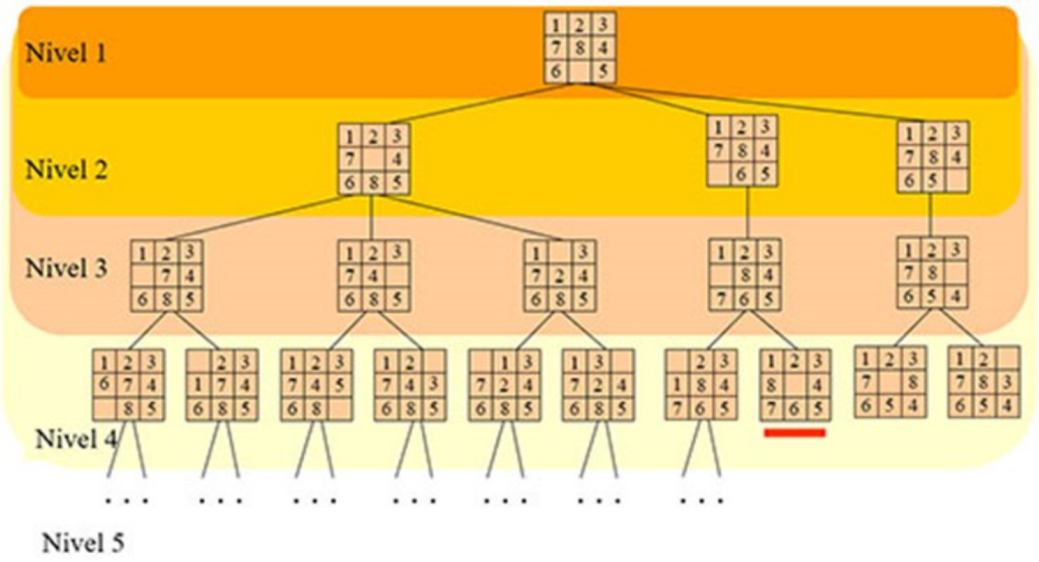

Figure 17: Breadth-first search tree for the 8-puzzle problem. The tree is expanded level by level, showing all possible states at each depth.

22.1 Algorithm Implementation

The breadth-first search algorithm can be derived from the general search algorithm previously presented, with the following modifications:

- **Adding successors**: New successors are added to the *end* of the open list.
- **Queue behavior**: The open list functions as a queue (inserting at the end and retrieving from the beginning), which ensures that the oldest nodes (i.e., those at the shallowest depth) are always expanded first.
- **Visited nodes control**: Additionally, we control nodes that have been visited previously to avoid revisiting states.

22.2 Properties

This algorithm is **complete** and **optimal** (finds the shortest path when all actions have the same cost), but it presents very poor time and space complexity, as it depends proportionally on the depth level d of the solution and, therefore, on the number of expanded nodes or branching factor b.

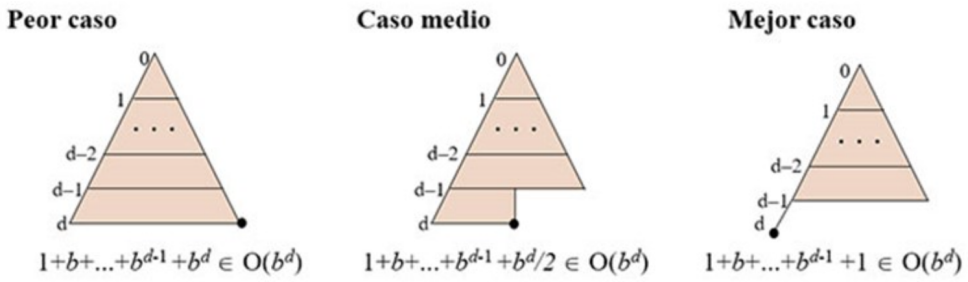

Figure 18: Time complexity analysis for breadth-first search in worst case, average case, and best case scenarios. The complexity is $O(b^d)$ where b is the branching factor and d is the depth of the solution.

As shown in Figure 18, the time complexity is $O(b^d)$ in all cases:

- **Worst case**: Explores all nodes up to depth d: $1 + b + \ldots + b^{d-1} + b^d \in O(b^d)$
- **Average case**: Explores approximately half of the nodes at depth d: $1 + b + \ldots + b^{d-1} + b^d/2 \in O(b^d)$
- **Best case**: Finds the goal as the first node at depth d: $1+b+\ldots+b^{d-1}+1 \in O(b^d)$

The space complexity is also $O(b^d)$ because the algorithm must store all nodes at the deepest level in the open list.

22.3 Example

Let us illustrate BFS with a concrete example. Consider the weighted graph shown in Figure 19, where we want to find a path from the initial state S to one of the goal states ($G1$ or $G2$).

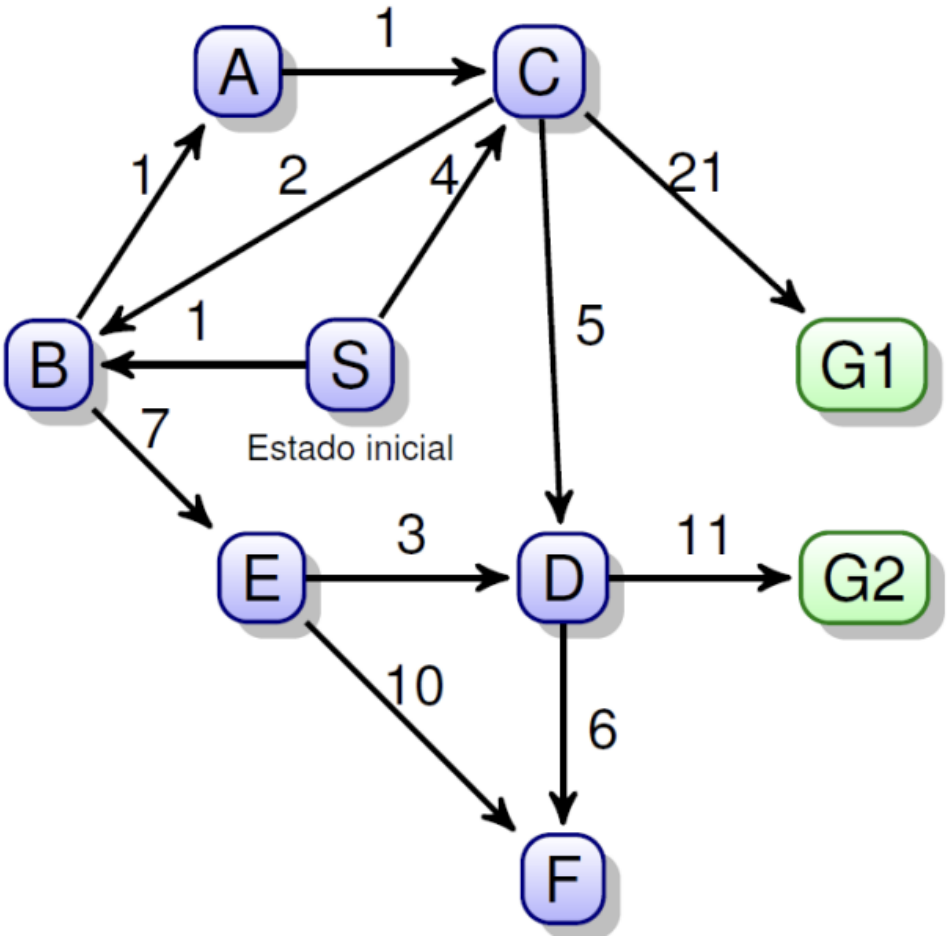

Figure 19: Example search problem: finding a path from initial state S to goal states $G1$ or $G2$. Edge weights represent the cost of transitions. **Note:** In this BFS example, all edge costs are treated as 1 (uniform cost), so the algorithm finds the shortest path in terms of number of steps, not total cost.

Table 18 shows the step-by-step execution of BFS on this problem. The algorithm expands nodes level by level, and we can see how repeated states are avoided.

Expanded	**Generates**	**Open**
S	B, C	$B(SB), C(SC)$
$B(SB)$	A, E	$C(SC), A(SBA), E(SBE)$
$C(SC)$	$D, G1, \cancel{B}$	$A(SBA), E(SBE), D(SCD), G1(SCG1)$
$A(SBA)$	$\cancel{C}$	$E(SBE), D(SCD), G1(SCG1)$
$E(SBE)$	$F, \cancel{D}$	$D(SCD), G1(SCG1), F(SBEF)$
$D(SCD)$	$G2, \cancel{F}$	$G1(SCG1), F(SBEF), G2(SCDG2)$
$G1(SCG1)$	**FIN**	—

Table 18: BFS expansion process for the example problem. The path notation (e.g., SB) indicates the path from S to the node. Nodes with strikethrough ($\cancel{X}$) represent states that would be generated but are skipped because they have already been visited or are in the open list, preventing cycles and redundant exploration. **Note:** Generated nodes are listed in alphabetical order for convention. All edge costs are treated as 1 in this BFS example.

Solution summary:

- **Solution:** $SC, CG1$
- **Length:** 2
- **Expanded:** 6 nodes
- **Generated:** 8 nodes (excluding repeated states)
- **Iterations:** 7

In this example, BFS finds the goal $G1$ through the path $S \to C \to G1$ (2 steps). Notice how the algorithm systematically explores all nodes at depth 1 (B, C) before moving to depth 2 $(A, E, D, G1)$, and how it avoids generating states that have already been visited.

23 Depth-First Search

Depth-First Search (DFS) is another uninformed search strategy (without additional information). Unlike breadth-first search, DFS attempts to develop a path of indeterminate length, trying to reach deep goals (those that have a long path to reach them) by developing the fewest possible branches.

In general, DFS works well when combined with additional information, but it can solve problems in the same way as a breadth-first algorithm.

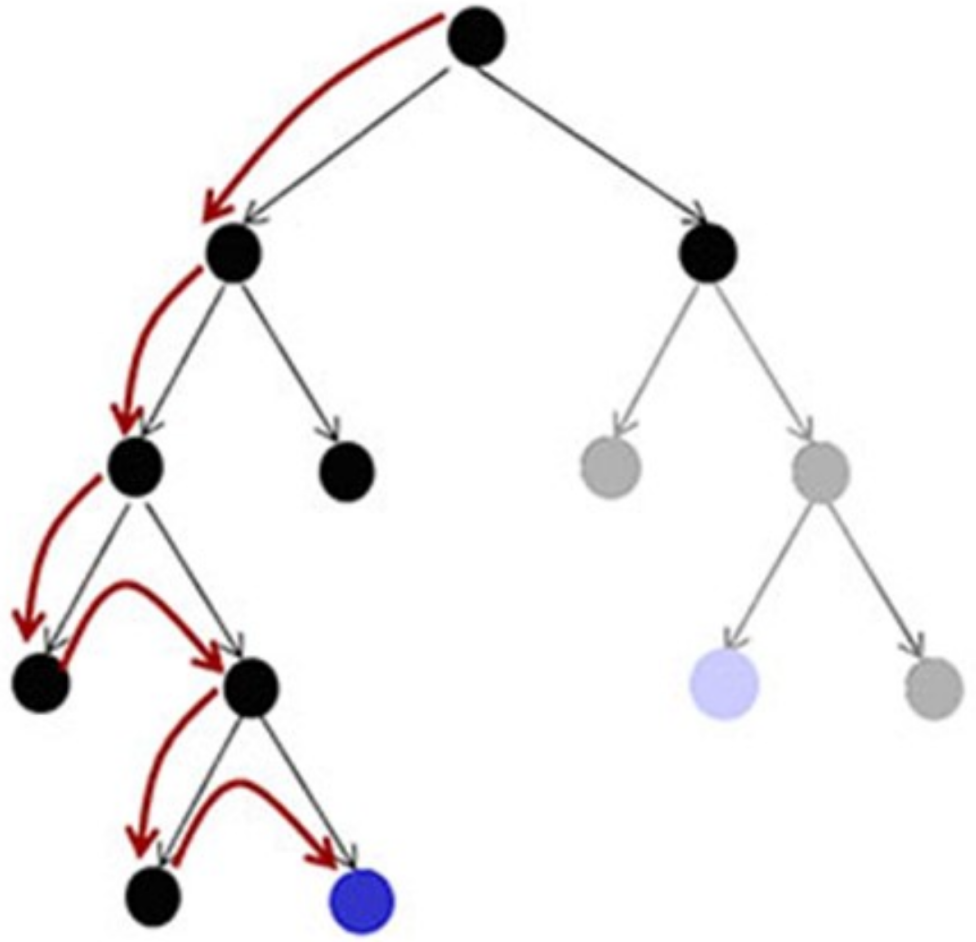

Figure 20: Example of how the search tree is generated with the depth-first search algorithm. The red path highlights the exploration sequence, showing how DFS goes deep before backtracking.

In this strategy, the tree is expanded from "left to right," so the deepest nodes are expanded first. If a node without successors is reached, the algorithm backtracks and expands the next deepest node.

As a result, the method explores a "current path" and does not always find the node at minimum depth. The general algorithm is adapted with the following considerations:

- **Adding successors**: New successors are added to the *beginning* of the open list.
- **Stack behavior**: The open list functions as a stack (inserting at the beginning and extracting from the beginning), so we always extract the deepest node. By storing all successors of an expanded node in the open list, we allow "backtracking."
- **Visited nodes control**: We only process a node from the stack if it has not been visited yet.

23.1 Properties

The analysis of this algorithm shows that it is **complete** (if and only if we guarantee the elimination of repeated states within the same branch), but it is **not optimal** (for cost-one operators), since it does not guarantee that it will always find the solution at the minimum depth.

DFS has the following characteristics:

- **Completeness**: Complete only if repeated states within the same branch are eliminated. Without this control, DFS can get stuck in infinite loops in graphs with cycles.
- **Optimality**: Not optimal—it may find a solution, but not necessarily the shortest one, as it explores deep paths first rather than systematically exploring by depth level.
- **Time complexity**: $O(b^m)$ in the worst case, where b is the branching factor and m is the maximum depth of the search tree. This can be much worse than BFS if the solution is shallow.
- **Space complexity**: $O(bm)$—much better than BFS, as it only needs to store the current path from root to leaf, not all nodes at a given depth level.

23.2 Example

Let us illustrate DFS with the same example graph used for BFS. Consider the weighted graph shown in Figure 21, where we want to find a path from the initial state S to one of the goal states ($G1$ or $G2$).

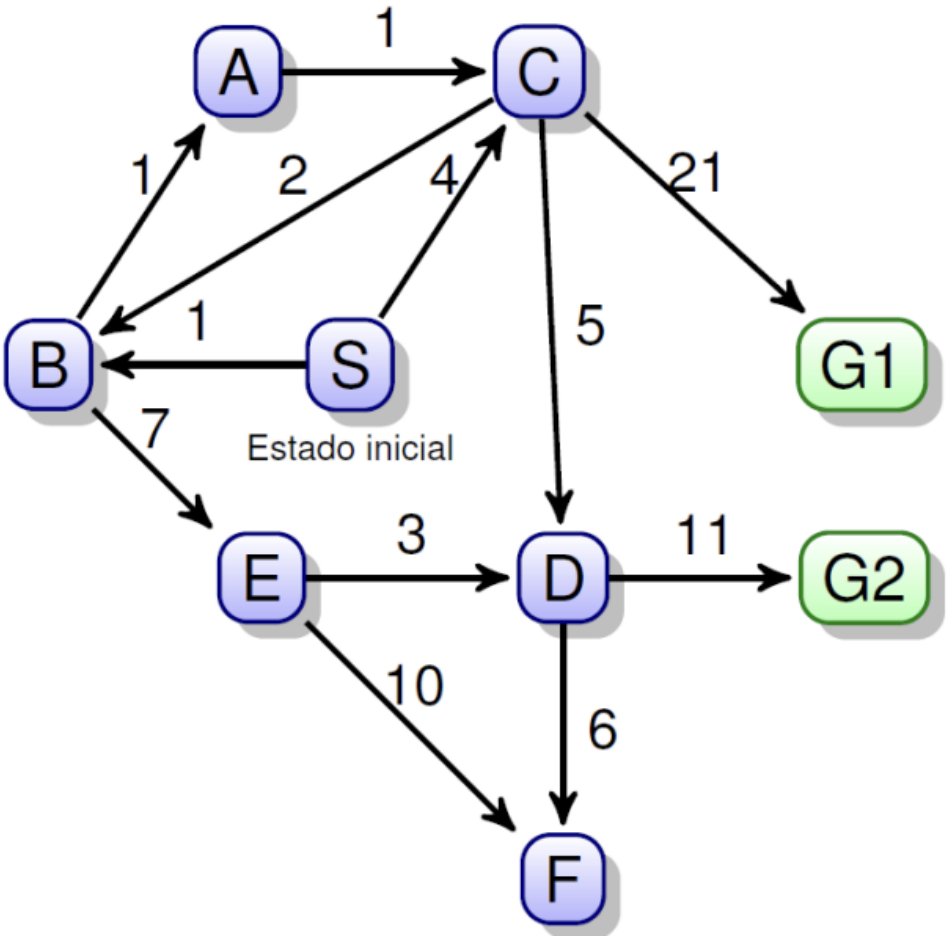

Figure 21: Example search problem: finding a path from initial state S to goal states $G1$ or $G2$. Edge weights represent the cost of transitions. **Note:** In this DFS example, all edge costs are treated as 1 (uniform cost), and nodes are generated in inverse alphabetical order for convention.

Table 19 shows the step-by-step execution of DFS on this problem. Notice how DFS explores deep paths first, going as far as possible before backtracking.

Expanded	Generates	Open
S	C, B	$B(SB), C(SC)$
$B(SB)$	E, A	$A(SBA), E(SBE), C(SC)$
$A(SBA)$	$\not{C}$	$E(SBE), C(SC)$
$E(SBE)$	F, D	$D(SBED), F(SBEF), C(SC)$
$D(SBED)$	$G2, \not{F}$	$G2(SBEDG2), F(SBEF), C(SC)$
$G2(SBEDG2)$	**FIN**	—

Table 19: DFS expansion process for the example problem. The path notation (e.g., SB) indicates the path from S to the node. Nodes with strikethrough ($\not{X}$) represent states that would be generated but are skipped because they have already been visited or are in the open list, preventing cycles and redundant exploration. **Note:** Generated nodes are listed in inverse alphabetical order for convention. All edge costs are treated as 1 in this DFS example.

Solution summary:

- **Solution:** $SB, BE, ED, DG2$
- **Length:** 4
- **Expanded:** 5 nodes
- **Generated:** 7 nodes (excluding repeated states)
- **Iterations:** 6

In this example, DFS finds the goal $G2$ through the path $S \to B \to E \to D \to G2$ (4 steps). Notice how DFS explores deep into the tree (following the path $S \to B \to E \to D$) before finding the goal, unlike BFS which would have found a shorter path. This demonstrates that DFS is not optimal—it finds a solution, but not necessarily the shortest one.

24 Uniform Cost Search

In the previous cases of breadth-first and depth-first search, we assumed that the cost of applying any action (and therefore, the cost of choosing a branch) was always equal to 1, so the total cost of a path was the number of levels at which a particular node was located. But what happens when the cost of transitioning from one node to another is not equal for all actions in the environment?

To solve this type of scenario where the cost is not equal for all actions (but is positive in all cases), we have the **Uniform Cost Search** (UCS) algorithm.

24.1 Algorithm Overview

Using the same general search algorithm, we apply the idea of directing the search by the cost of the operators. We assume there exists a utility function $f(n) = g(n)$ that allows us to calculate the real cost to reach node n from the initial node. In each iteration of the algorithm, we will expand first the node with the lowest cost f.

For simplicity, in the rest of the explanation of this algorithm, we will refer to the utility function interchangeably as f or g.

24.2 Algorithm Modifications

The modifications made to the general search algorithm are:

- **Priority-based storage**: We store each node by priority based on its g value. Therefore, the insertion of new nodes in the open list will be ordered in ascending order according to their g value.
- **Priority queue**: The above makes the open list a priority queue ordered by the g value.
- **Duplicate prevention**: We only add a node to the open list if it is not already in it.
- **Cost update**: If a node is found in the open list, we replace it if and only if the new cost f is lower.

24.3 Properties

This algorithm is **complete** and **optimal** when all costs have positive integer values. Therefore, the sequence of g values is unbounded, and we always expand according to the insertion order based on this same function.

UCS has the following characteristics:

- **Completeness**: Complete when all edge costs are positive. If costs can be zero, completeness is guaranteed only if we ensure no infinite paths with zero cost exist.
- **Optimality**: Optimal—always finds the path with the lowest total cost from the initial state to the goal, as it expands nodes in order of increasing path cost.
- **Time complexity**: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, where C^* is the cost of the optimal solution and ϵ is the minimum edge cost. In practice, this can be exponential.
- **Space complexity**: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$—same as time complexity, as all nodes with cost less than the optimal solution cost may be stored.

UCS can be seen as a generalization of BFS: when all edge costs are equal to 1, UCS behaves exactly like BFS.

24.4 Example

Let us illustrate UCS with the same example graph. Consider the weighted graph shown in Figure 22, where we want to find the lowest-cost path from the initial state S to one of the goal states ($G1$ or $G2$). Unlike BFS and DFS, UCS considers the actual edge costs.

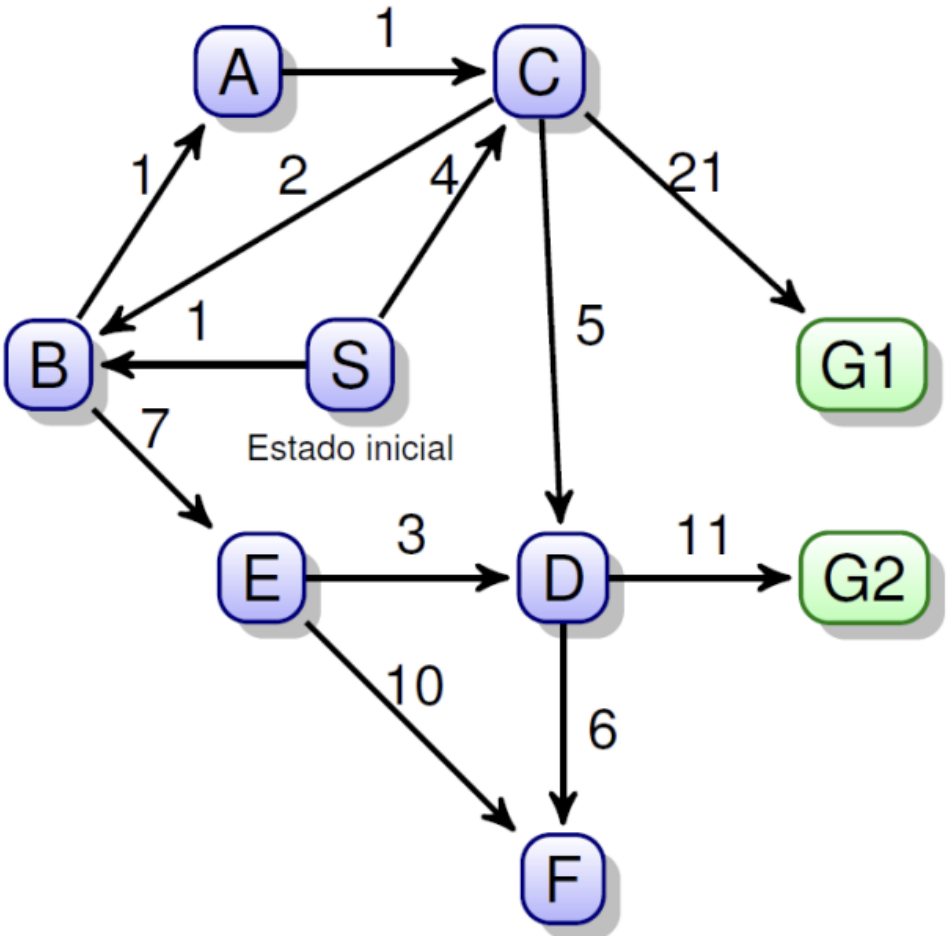

Figure 22: Example search problem: finding the lowest-cost path from initial state S to goal states $G1$ or $G2$. Edge weights represent the actual costs of transitions. **Note:** In UCS, nodes are generated in alphabetical order, and this order is also used for tie-breaking in the open list when g values are equal.

Table 20 shows the step-by-step execution of UCS on this problem. Notice how UCS expands nodes in order of increasing path cost g, and updates nodes in the open list when better paths are found.

Expanded	**Generates**	**Open**	**Closed**
S	$B(g=1), C(g=4)$	$B(g=1), C(g=4)$	S
$B(g=1)$	$A(g=2), E(g=8)$	$A(g=2), C(g=4), E(g=8)$	$S, B(g=1)$
$A(g=2)$	$C(g=3)$	$C(g=3), \cancel{C(g=4)}, E(g=8)$	$S, B(g=1), A(g=2)$
$C(g=3)$	$D(g=8), G1(g=24)$ (B repeated in path to C)	$D(g=8), E(g=8), G1(g=24)$	$S, B(g=1), A(g=2), C(g=3)$
$D(g=8)$	$F(g=14), G2(g=19)$	$E(g=8), F(g=14), G2(g=19), G1(g=24)$	$S, B(g=1), A(g=2), C(g=3), D(g=8)$
$E(g=8)$	$D(g=11), F(g=18)$ (D and F worse than existing paths)	$F(g=14), G2(g=19), G1(g=24)$	$S, B(g=1), A(g=2), C(g=3), D(g=8), E(g=8)$
$F(g=14)$	(no successors)	$G2(g=19), G1(g=24)$	$S, B(g=1), A(g=2), C(g=3), D(g=8), E(g=8), F(g=14)$
$G2(g=19)$	**FIN**	—	$S, B(g=1), A(g=2), C(g=3), D(g=8), E(g=8), F(g=14)$

Table 20: UCS expansion process for the example problem. The path notation shows the path from S to the node, and g values represent the total cost. Nodes with strikethrough ($\cancel{X}$) represent states that are replaced when a better path is found. The closed list tracks all expanded nodes. **Note:** Generated nodes are listed in alphabetical order, and this order is used for tie-breaking when g values are equal.

Solution summary:

- **Solution:** $SB(1), BA(1), AC(1), CD(5), DG2(11)$
- **Cost:** $1+1+1+5+11=19$
- **Expanded:** 7 nodes
- **Generated:** 11 nodes (some were never inserted into the open list due to worse costs)

In this example, UCS finds the goal $G2$ through the path $S \to B \to A \to C \to D \to G2$ with total cost 19. Notice how UCS:

- Expands nodes in order of increasing cost (first B with $g=1$, then A with $g=2$, then C with $g=3$, etc.)
- Updates node C when a better path is found (replacing $C(g=4)$ with $C(g=3)$)
- Ignores worse paths (e.g., $D(g=11)$ and $F(g=18)$ from E are not added because better paths already exist)
- Finds the optimal solution with the lowest total cost, unlike DFS which found a longer path

References

Martín Molina González. *Métodos de resolución de problemas: Aplicación al diseño de sistemas inteligentes*. Fundación General de la UPM, 2006. URL `https://oa.upm.es/14207/`.