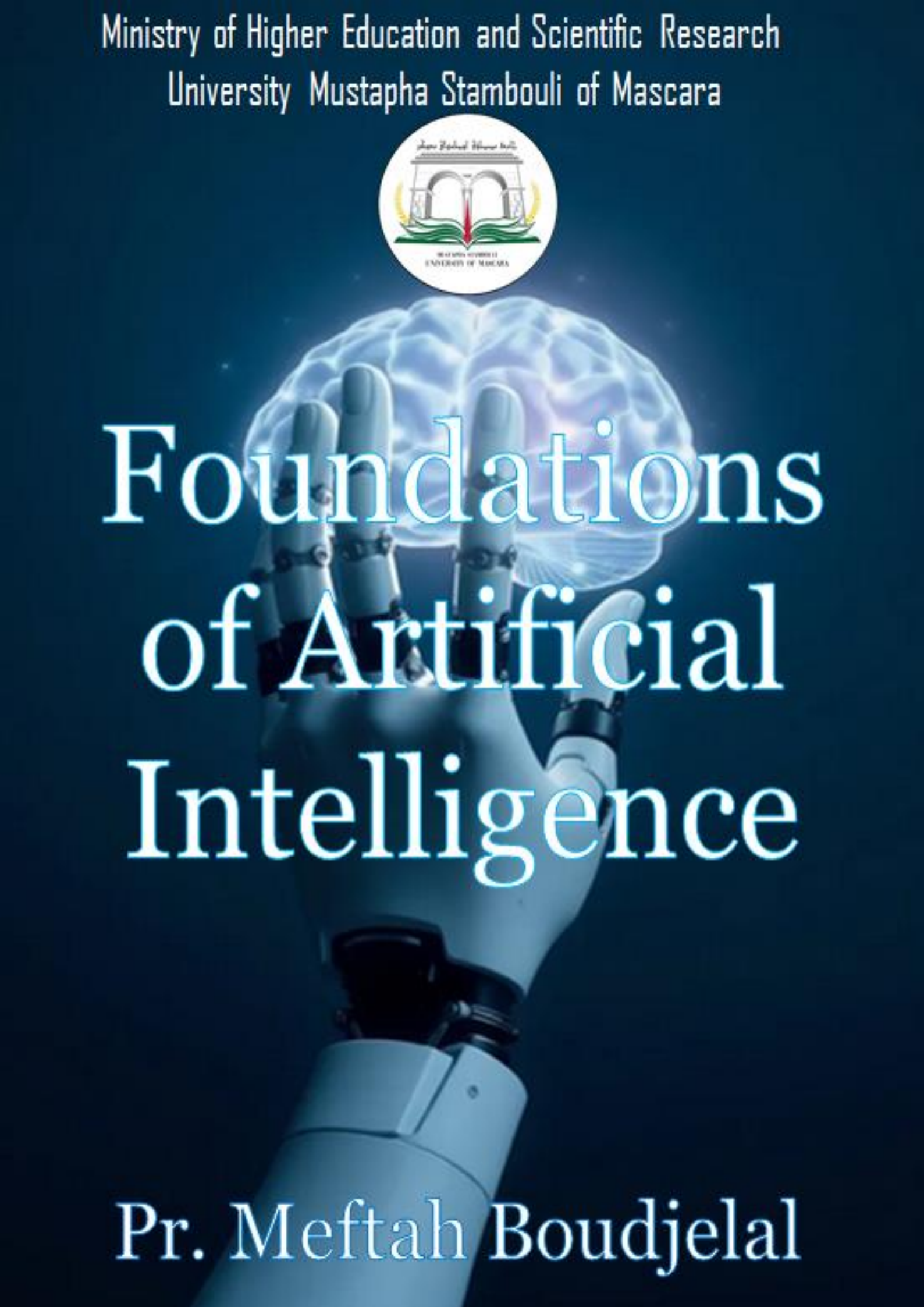
Ministry of Higher Education and Scientific Research
University Mustapha Stambouli of Mascara



Foundations of Artificial Intelligence

Pr. Meftah Boudjelal

Contents

Chapter 6. Machine Learning and Neural Networks

General Introduction

General Introduction

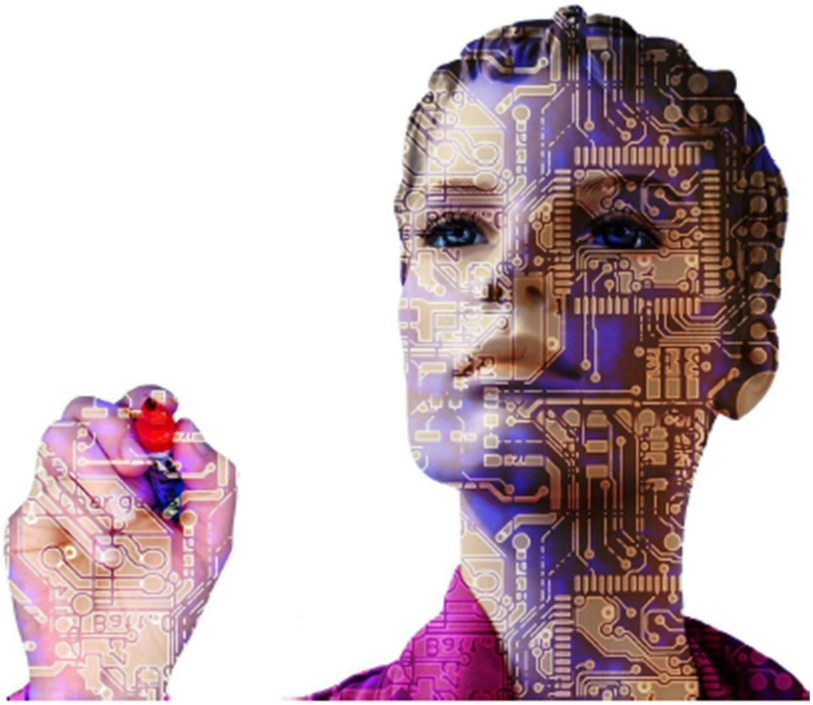
Artificial Intelligence (AI) has emerged as one of the most transformative technologies of the 21st century, reshaping industries, enhancing decision-making, and pushing the boundaries of what machines can achieve. From virtual assistants and recommendation systems to autonomous vehicles and advanced medical diagnostics, AI is revolutionizing the way we interact with technology and process information.

This lecture notes provides a structured and comprehensive introduction to the fundamental concepts of AI, guiding readers through its core principles, methodologies, and applications. The chapters are designed to build a strong theoretical foundation while also highlighting practical implementations.

- Chapter 1: Introduction to Artificial Intelligence – Offers an overview of AI, its history, key milestones, and different branches, including narrow AI vs. general AI.
- Chapter 2: Logics and Reasoning – Explores symbolic AI, propositional and first-order logic, and rule-based systems that enable machines to perform deductive and inductive reasoning.
- Chapter 3: Search Algorithms – Covers classical search techniques such as breadth-first, depth-first, A, and heuristic-based methods used in problem-solving and pathfinding.
- Chapter 4: Game Theory – Examines strategic decision-making in competitive environments, including Nash equilibrium, minimax algorithms, and applications in economics and AI.
- Chapter 5: Metaheuristics – Introduces optimization techniques like genetic algorithms, simulated annealing, and swarm intelligence for solving complex, real-world problems.
- Chapter 6: Machine Learning and Neural Networks– Delves into supervised, unsupervised, and reinforcement learning, along with neural networks forming the backbone of modern AI applications.

By the end of this lecture notes, students will have acquired a well-rounded understanding of AI's key components, enabling them to explore advanced topics or apply these concepts in research and industry.

Chapter 1
Introduction to Artificial Intelligence



1. General definition of intelligence

Intelligence can be defined as the ability to **understand**, **learn**, and **adapt to new situations**. This includes the ability to solve problems, reason logically, perceive relationships, and use knowledge effectively. Intelligence is not just about acquired knowledge, but also about how this knowledge is applied in different situations. It can also be perceived as the faculty to process information to achieve specific goals.

- According to A. Turing: what makes it difficult to distinguish between a task performed by a human or by a machine,
- According to C. Darwin: what allows the survival of the fittest individual, perfectly adapted to its environment,
- According to Yam (1998): An exact definition of intelligence is probably impossible; the most likely: the ability to manage complexity and solve problems in a useful context.
- According to Voss (2004): The ability of an entity to achieve goals. Greater intelligence.

2. Definition of artificial intelligence

Defining precisely and definitively what artificial intelligence (AI) corresponds to is a challenging task. This term generally refers to the set of techniques that enable the integration of intelligent behaviors into a machine. The conditions necessary for creating such a machine thus involve a wide range of knowledge domains, including robotics, biology, signal processing, logic, statistics, and constrained optimization.
The debate on the definition of intelligent behavior and the means to reproduce it is lively within the research community: Can simple logical principles explaining intelligent behaviors be formalized? Can intelligence be reproduced through symbolic manipulations? Is knowledge of human or animal biology necessary for understanding intelligence? These are questions about which no consensus has emerged in recent decades, given the complexity of the subject of intelligence.
Artificial intelligence (AI) is a set of theories and techniques aimed at creating machines capable of simulating human intelligence. It includes devices that imitate or replace humans in certain implementations of their cognitive functions. The term 'artificial intelligence' was coined by John McCarthy, who defined it as 'the science and engineering of making intelligent machines, especially intelligent computer programs.
Artificial intelligence (AI) produces machines that imitate humans:

- Simulates the intelligent processes of humans
- Reproduces the methods or results of human reasoning or intuition.

According to Marvin Minsky: « ... the science of making machines do things that would require intelligence if done by humans»
According to E. Feigenbaum: «AI is the part of computer science concerned with designing intelligent computer systems»

2.1. AI: Definitions According to four Orientations

AI is a discipline that systematizes and automates intellectual tasks to create machines capable of:

THINK like a human (cognitive science → cognitive modeling)	think **RATIONALLY** (logical approach)
"The automation activities that we associate with human thinking, activities such as decision-making, problem-solving, learning."(Bellman, 1978)	"The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985)
ACT like a human (test of Turing)	Act **RATIONALLY** (achieve a goal)
"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)	"Computational Intelligence is the study of the design on intelligent agents" (Pool, 1998)

3. Applications of Artificial Intelligence

Artificial intelligence is a versatile technology that finds applications in almost every sector of our society. Its potential to improve efficiency, reduce costs, and provide innovative solutions is immense, and its adoption continues to grow at a rapid pace.

3.1. Healthcare

AI is revolutionizing the healthcare sector by improving diagnosis, treatment, and patient management. Machine learning algorithms are used to analyze medical images, predict diseases, and personalize treatments. For example, AI helps detect anomalies in X-rays and predict epidemics.

3.2. Finance

In the financial sector, AI is used to automate tasks, improve decision-making, and reduce risks. Applications include fraud detection, portfolio management, and predictive analysis of financial markets. AI algorithms also enable better account management and the generation of financial reports.

3.3. Transport

AI plays a crucial role in the development of autonomous vehicles and the optimization of transportation systems. AI technologies are used for navigation, traffic management, and predictive maintenance of vehicles. Intelligent transportation systems improve the efficiency and safety of travel.

3.4. Commerce and Customer Service

Conversational agents (chatbots) and personalized recommendation systems are examples of AI in commerce. These technologies enhance the customer experience by making service more accessible and personalized. AI also helps analyze purchasing behaviors and optimize marketing strategies.

3.5. Agriculture

AI is used to optimize agricultural yields, manage resources, and monitor crop health. Drones equipped with sensors and data analysis systems enable precision agriculture, thereby reducing costs and increasing productivity.

3.6. Security

In the field of security, AI is used for facial recognition, video surveillance, and the detection of cyber threats. AI systems can analyze real-time data streams to identify suspicious behaviors and prevent security incidents.

3.7. Education

AI is transforming education by personalizing learning paths and providing automated assessment tools. Online learning platforms use algorithms to adapt content to the individual needs of students, thereby improving teaching efficiency.

3.8. Industry

In the industrial sector, AI is used for process automation, predictive maintenance, and supply chain optimization. Industrial robots equipped with AI can perform complex tasks with high precision, increasing productivity and reducing errors.

3.9. Justice and Public Safety

AI is used to analyze criminal data, predict crimes, and improve the efficiency of law enforcement. AI systems can help identify criminal patterns and optimize resources for crime prevention and resolution.

3.10. Environment

AI contributes to the fight against climate change by optimizing the management of natural resources and predicting extreme weather phenomena. AI technologies are also used to monitor biodiversity and manage ecosystems."

4. Importance of Artificial Intelligence

AI has the potential to transform many aspects of society and the economy, improving efficiency, stimulating innovation, and solving complex problems.

4.1. Improvement of Efficiency and Productivity

- Automation: AI can automate repetitive and tedious tasks, freeing up time for humans to focus on more creative and strategic activities.
- Optimization: AI algorithms can optimize production, logistics, and management processes, reducing costs and increasing efficiency.

4.2. Improved Decision-Making

- Data Analysis: AI can process and analyze large amounts of data quickly and accurately, providing valuable insights for decision-making.
- Predictions: AI models can make predictions based on historical data and trends, helping businesses anticipate future needs and plan accordingly.

4.3. Innovation and Creativity

- New Products and Services: AI can help develop new products and services by identifying market opportunities and quickly testing ideas.
- Personalization: AI systems can offer personalized experiences to customers, thereby improving satisfaction and loyalty.

4.4. Health and Well-being

- Medical Diagnosis: AI can help diagnose diseases more quickly and accurately by analyzing medical images and patient data.
- Medical Research: AI algorithms can accelerate medical research by identifying patterns and correlations in health data.

4.5. Security and Surveillance

- Cybersecurity: AI can detect and respond to cybersecurity threats in real-time, thereby protecting sensitive systems and data.
- Surveillance: AI systems can monitor critical infrastructures, transportation networks, and public environments to ensure safety.

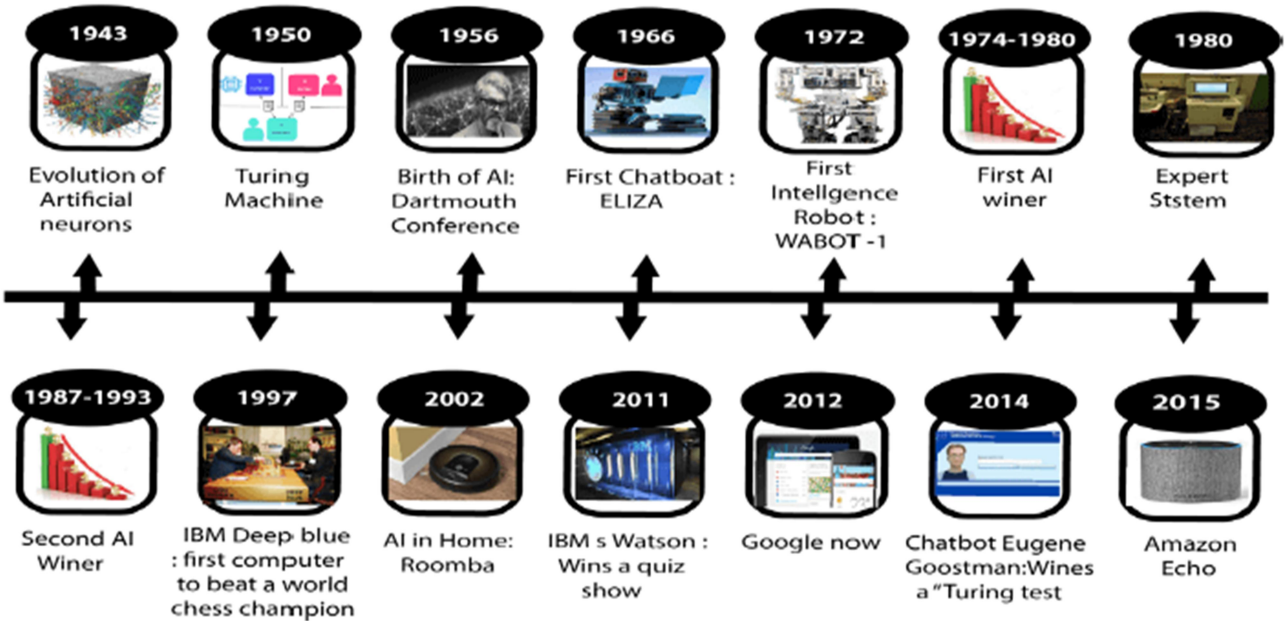
4.6. Education and Learning

- Personalized Learning: AI systems can tailor learning programs to the individual needs of students, improving the effectiveness of teaching.
- Intelligent Tutoring: AI-based virtual assistants can provide real-time educational support, helping students better understand concepts.

4.7. Sustainable Development

- Resource Management: AI can help optimize the use of natural resources, thereby reducing environmental impact.
- Renewable Energy: AI algorithms can improve the efficiency of renewable energy systems, contributing to the transition to a more sustainable economy.

5. Evolution of Artificial Intelligence history



5.1. Maturation of Artificial Intelligence (1943-1952)

- Year 1943: The first work which is now recognized as AI was done by Warren McCulloch and Walter pits in 1943. They proposed a model of artificial neurons.
- Year 1949: Donald Hebb demonstrated an updating rule for modifying the connection strength between neurons. His rule is now called Hebbian learning.
- Year 1950: The Alan Turing who was an English mathematician and pioneered Machine learning in 1950. Alan Turing publishes "Computing Machinery and Intelligence" in which he proposed a test. The test can check the machine's ability to exhibit intelligent behavior equivalent to human intelligence, called a Turing test.

5.2. The birth of Artificial Intelligence (1952-1956)

- **Year 1955:** An Allen Newell and Herbert A. Simon created the "first artificial intelligence program" Which was named as **"Logic Theorist"**. This program

had proved 38 of 52 Mathematics theorems, and find new and more elegant proofs for some theorems.

- **Year 1956:** The word "Artificial Intelligence" first adopted by American Computer scientist John McCarthy at the Dartmouth Conference. For the first time, AI coined as an academic field.
- At that time high-level computer languages such as FORTRAN, LISP, or COBOL were invented. And the enthusiasm for AI was very high at that time.

5.3. The golden years-Early enthusiasm (1956-1974)

- **Year 1966:** The researchers emphasized developing algorithms which can solve mathematical problems. Joseph Weizenbaum created the first chatbot in 1966, which was named as ELIZA.
- **Year 1972:** The first intelligent humanoid robot was built in Japan which was named as WABOT-1.

5.4. The first AI winter (1974-1980)

- The duration between years 1974 to 1980 was the first AI winter duration. AI winter refers to the time period where computer scientist dealt with a severe shortage of funding from government for AI researches.
- During AI winters, an interest of publicity on artificial intelligence was decreased.

5.5. A boom of AI (1980-1987)

- **Year 1980:** After AI winter duration, AI came back with "Expert System". Expert systems were programmed that emulate the decision-making ability of a human expert.
- In the Year 1980, the first national conference of the American Association of Artificial Intelligence was held at Stanford University.

5.6 The second AI winter (1987-1993)

- The duration between the years 1987 to 1993 was the second AI Winter duration.
- Again Investors and government stopped in funding for AI research as due to high cost but not efficient result. The expert system such as XCON was very cost effective.

5.7. The emergence of intelligent agents (1993-2011)

- **Year 1997:** In the year 1997, IBM Deep Blue beats world chess champion, Gary Kasparov, and became the first computer to beat a world chess champion.

- **Year 2002:** for the first time, AI entered the home in the form of Roomba, a vacuum cleaner.
- **Year 2006:** AI came in the Business world till the year 2006. Companies like Facebook, Twitter, and Netflix also started using AI.

5.8 Deep learning, big data and artificial general intelligence (2011-present)

- **Year 2011:** In the year 2011, IBM's Watson won jeopardy, a quiz show, where it had to solve the complex questions as well as riddles. Watson had proved that it could understand natural language and can solve tricky questions quickly.
- **Year 2012:** Google has launched an Android app feature "Google now", which was able to provide information to the user as a prediction.
- **Year 2014:** In the year 2014, Chatbot "Eugene Goostman" won a competition in the infamous "Turing test."
- **Year 2018:** The "Project Debater" from IBM debated on complex topics with two master debaters and also performed extremely well.
- Google has demonstrated an AI program "Duplex" which was a virtual assistant and which had taken hairdresser appointment on call, and lady on other side didn't notice that she was talking with the machine. Now AI has developed to a remarkable level. The concept of Deep learning, big data, and data science are now trending like a boom. Nowadays companies like Google, Face book, IBM, and Amazon are working with AI and creating amazing devices. The future of Artificial Intelligence is inspiring and will come with high intelligence.

6. Artificial Intelligence Approaches

Artificial intelligence is a constantly evolving field, with a multitude of approaches tailored to different types of problems. Each approach has its own advantages and disadvantages, and the choice of approach often depends on the specific context and objectives of the application.

6.1. Symbolic Approach

The symbolic approach, also known as symbolic AI, relies on the manipulation of symbols and explicit rules to represent knowledge and solve problems. This approach is often associated with expert systems, which use predefined rules to make decisions.

Advantages:

- Transparency and explainability of decisions.

- Effective for well-defined tasks with clear rules.

Disadvantages:

- Difficulty in handling unexpected situations or unstructured data.
- Requires deep domain knowledge to define the rules.

6.2. Statistical Approach (Machine Learning)

Machine learning is a statistical approach that enables systems to learn from data. This approach includes techniques such as neural networks, decision trees, and Bayesian methods.

Advantages:

- Ability to handle large amounts of data and adapt to new information.
- Used in various fields such as image recognition, fraud detection, and trend prediction.

Disadvantages:

- Opacity of complex models (black box).
- Requires large amounts of data for effective learning.

6.3. Cognitivist Approach

The cognitivist approach aims to reproduce human mental processes using computational models. This approach is influenced by cognitive sciences and seeks to explain intelligent behavior through mental processes that manipulate representations.

Advantages:

- Allows for a better understanding of the mechanisms of human thought.
- Used for applications requiring human interaction, such as virtual assistants.

Disadvantages:

- Complexity of models and difficulty in implementing them.
- Epistemological and philosophical challenges related to modeling human cognition."

6.4. Bioinspired Approach

Bioinspired approaches draw inspiration from biological mechanisms to develop AI algorithms. This includes techniques such as genetic algorithms, artificial neural networks, and swarm systems.

Advantages:

- Ability to solve complex and non-linear problems.

- Flexibility and adaptability of algorithms.

Disadvantages:

- Requires a deep understanding of biological systems.
- Can be computationally expensive.

6.5. Generative Approach

Generative approaches, such as Generative Adversarial Networks (GANs), are used to create new data from existing models. These techniques are particularly effective for generating images, texts, and other types of data.

Advantages:

- Ability to generate realistic and diverse data.
- Used in creative fields such as art, music, and design.

Disadvantages:

- Ethical risks associated with generating deceptive content.
- Complexity of models and the need for significant computational resources.

Exercises

A set of multiple-choice questions (MCQs) on Introduction to Artificial Intelligence (AI). These questions cover fundamental concepts of AI, its applications, and key techniques.

1. What is Artificial Intelligence (AI)?

a) A branch of computer science that focuses on building machines capable of performing tasks that typically require human intelligence.
b) A type of software used only for gaming.
c) A programming language for building websites.
d) A tool for creating hardware devices.

2. Which of the following is NOT a goal of AI?

a) To create systems that can learn and adapt.
b) To develop machines that can perform tasks without human intervention.
c) To replace all human jobs with machines.
d) To enable machines to understand and process natural language.

3. What is the difference between Narrow AI and General AI?

a) Narrow AI is designed for specific tasks, while General AI can perform any intellectual task that a human can.
b) Narrow AI is used in robotics, while General AI is used in gaming.
c) Narrow AI requires human supervision, while General AI does not.
d) Narrow AI is a subset of General AI.

4. Which of the following is an example of Narrow AI?

a) A self-driving car.
b) A robot that can cook, clean, and write novels.
c) A machine that can solve any mathematical problem.
d) A system that can understand and replicate human emotions.

5. Which of the following is an application of Natural Language Processing (NLP)?

a) Speech recognition.
b) Image classification.
c) Predicting stock prices.
d) Clustering customer data.

6. Which of the following is a key challenge in AI?

a) Lack of data.
b) Overfitting in machine learning models.
c) Ethical concerns about AI decision-making.
d) All of the above.

7. Which of the following is a key component of an AI system?

a) Data.
b) Algorithms.
c) Computational power.
d) All of the above.

Chapter 2

Logics and Reasoning

1. Introduction

It is common to hear "it's logical" when someone tries to share their point of view. This seemingly innocuous expression actually asks those who hear it to adopt the reasoning of the person saying it. Typically, this phrase implies an obviousness in the statement that precedes it, but in reality, behind it lies an entire chain of reasoning—whether by deduction, proof by contradiction, or elimination.

1.1 Brief History

The philosophers of Ancient Greece laid the foundations of logic. In particular, Aristotle established the basics of logic. Aristotelian logic was taught for a very long time and dominated thought at least until the Middle Ages. It was only very recently that modern logic emerged.

It was Frege who laid the foundations of modern logic. The essential difference compared to Aristotelian logic is that Frege took a mathematical approach to logic, whereas Aristotle's logic was infused with philosophy. Frege thus developed propositional logic and predicate logic.

While Aristotle used everyday language to conduct logical reasoning, Frege used a symbolic language: ideography.

Leibniz had already attempted to create a logical language he called the "universal characteristic," but unfortunately without success, as he did not achieve something that satisfied him. Today, modern logical language is no longer ideography, which is no longer used, but some symbols in modern logic are derived from this language.

1.2. The Importance of Logic

Logic is the study of the art of thinking correctly. It has become a cornerstone of philosophy and mathematics, and more recently, it has also become essential to linguistics and computer science.

In philosophy, logic plays a central role by offering rigorous methods to evaluate ideas, theories, and arguments. Philosophers rely on logical principles to clarify concepts, identify inconsistencies, and build coherent systems of thought. Logic ensures that philosophical discourse remains grounded in reason rather than mere speculation or subjective opinion.

Similarly, in mathematics, logic forms the foundation upon which all mathematical truths are built. Mathematical proofs depend heavily on logical deduction, allowing mathematicians to establish results with absolute certainty.

More recently, logic has expanded its influence into other domains, including linguistics and computer science. In linguistics, logic helps analyze the structure of language, enabling researchers to understand how meaning is conveyed through syntax and semantics. For instance, formal logic provides tools for studying sentence formation, quantifiers, and relationships between statements. Meanwhile, in

computer science, logic is integral to algorithm design, artificial intelligence, and programming languages. Computers operate based on binary logic (true/false), making logical operations the backbone of computation. Concepts like Boolean algebra, propositional logic, and predicate logic are essential for developing software, designing circuits, and creating intelligent systems capable of reasoning.

Beyond academia, logic is also crucial in everyday life. Critical thinking—a skill rooted in logical analysis—empowers individuals to make informed decisions, solve complex problems, and communicate ideas persuasively. Whether evaluating news sources, debating ethical issues, or planning strategic solutions, logical reasoning enhances clarity of thought and fosters intellectual rigor. In this sense, logic transcends specific disciplines, serving as a universal guide for rational inquiry and effective problem-solving.

2. Definitions

2.1. What is logic?

Logic comes from the Greek word "logos," which means "speech, discourse," and by extension "rationality." Thus, logic is the science of reason. More specifically, it is the science that studies the rules that must be followed for any valid reasoning, enabling one to distinguish between valid reasoning and reasoning that is not valid.

2.2. Propositional Logic

Propositional logic, also known as **sentential logic** or **statement logic**, is the simplest and most fundamental branch of classical logic. It deals with the study of **logical relationships between propositions** (statements or declarative sentences) that can be either **true** or **false**.

In propositional logic, propositions are treated as indivisible units, meaning their internal structure (e.g., subjects, predicates) is not analyzed. Instead, the focus is on how these propositions are combined using **logical connectives** (such as AND, OR, NOT, IMPLIES) to form more complex statements.

2.3. Definition of a Proposition

A **proposition** is a **declarative statement** that is either **true** or **false**, but not both. In other words, a proposition is a sentence that expresses a fact or a claim that can be objectively evaluated as either true or false.

Characteristics of a Proposition

1. **Declarative:**
    - A proposition must be a statement that declares something, not a question, command, or exclamation.
    - Example: "The sky is blue." (Declarative)

- Non-example: "Is the sky blue?" (Interrogative)

2. **Truth Value:**
   - A proposition must have a definite truth value: it is either **true** or **false**.
   - Example: "2 + 2 = 4" is true, while "2 + 2 = 5" is false.
3. **Objective:**
   - The truth value of a proposition should not depend on personal opinion or perspective.
   - Example: "Water boils at 100°C at sea level." (Objective fact)
   - Non-example: "Chocolate ice cream is the best." (Subjective opinion)
4. **Atomic:**
   - In propositional logic, a proposition is treated as an **indivisible unit**. Its internal structure (e.g., subjects, predicates) is not analyzed.
   - Example: "It is raining." (Atomic proposition)

This excludes, among others, questions (a), imperatives (b), exclamations (c), and more generally all so-called non-assertive statements, such as certain performatives (d), certain phatic function statements (e), or the entire class of modalized statements (f).

a) Is Said attending the class?
b) Close the door!
c) How kind she is!
d) I promise you I will come.
e) Can you hear me?
f) The exam will take place tomorrow.

2.4. Truth table

A truth table is a mathematical table used in logic to determine the truth value of a compound statement based on the truth values of its components. It systematically lists all possible combinations of truth values for the input propositions (variables) and shows the resulting truth value of the compound statement for each combination.

2.5. Tautology

A **tautology** is a logical statement or formula that is **always true**, regardless of the truth values of the individual propositions (variables) within it. In other words, a tautology is a statement that is true in every possible interpretation or scenario.

2.6. Satisfiable Proposition

A satisfiable proposition (or satisfiable formula) is a logical statement or expression that can be true under at least one interpretation of its variables. In other words,

there exists at least one assignment of truth values to the variables that makes the entire proposition true.

2.7. Unsatisfiable Proposition (Contradiction)

An unsatisfiable proposition (or contradiction) is a logical statement or formula that is always false, regardless of the truth values of its variables. In other words, there is no possible assignment of truth values to the variables that makes the proposition true.

Example: $P \wedge \neg P$

2.8. Consistent Proposition

A **consistent proposition** (or **consistent set of propositions**) is a logical statement or collection of statements that **can all be true simultaneously** under at least one interpretation of their variables. In other words, there exists at least one assignment of truth values to the variables that makes all the propositions in the set true at the same time.

3. Logical Connectives

Logical connectives are symbols or words used to combine or modify propositions in logic. They form the basis of constructing complex logical expressions from simpler ones.

3.1. Negation (NOT)

The **negation** connective, represented by the symbol ¬ or ~, inverts the truth value of a proposition. If a proposition P is true, then ¬P (read as "not P") is false, and vice versa. For example:
if P represents "It is raining," then ¬P means "It is not raining."
Negation is a unary connective, meaning it operates on a single proposition.

P	$\overline{P}$
T	F
F	T

3.2. Disjunction (OR)

The **disjunction** connective, represented by the symbol ∨, combines two propositions and is true if at least one of the propositions is true.

For example:

If P represents "It is raining" and Q represents "I am at home," then P∨Q (read as "P or Q") means "It is raining, or I am at home."
Disjunction is inclusive, meaning it allows for the possibility that both propositions are true.

P	**Q**	$\boldsymbol{P \vee Q}$
T	T	T
T	F	T
F	T	T
F	F	F

3.3. Conjunction (AND)

The **conjunction** connective, represented by the symbol ∧, combines two propositions and is true only if both propositions are true.
For example:
If P represents "It is raining" and Q represents "I am at home," then P∧Q (read as "P and Q") means "It is raining, and I am at home."
Conjunction is a binary connective, meaning it operates on two propositions.

P	**Q**	$\boldsymbol{P \wedge Q}$
T	T	T
T	F	F
F	T	F
F	F	F

3.4. Implication (IF...THEN)

The **implication** connective, represented by the symbol →, expresses a conditional relationship between two propositions. The statement P→Q (read as "If P, then Q") is false only when P is true and Q is false. In all other cases, the implication is true.
For example:
If P represents "It is raining" and Q represents "I will stay at home," then P→Q means "If it is raining, then I will stay at home."

P	**Q**	$\boldsymbol{P \Rightarrow Q}$
T	T	T
T	F	F
F	T	T
F	F	T

3.5. Biconditional (IF AND ONLY IF)

The **biconditional** connective, represented by the symbol ↔, expresses a two-way conditional relationship between two propositions. The statement P↔Q (read as "P if and only if Q") is true if both propositions have the same truth value.

For example:

If P represents "It is raining" and Q represents "I will stay at home," then P↔Q means "It is raining if and only if I will stay at home."

P	**Q**	$\boldsymbol{P \Leftrightarrow Q}$
T	T	T
T	F	F
F	T	F
F	F	T

3.6. Exclusive OR (XOR)

The **exclusive OR** connective, represented by the symbol ⊕ or ⊻, combines two propositions and is true if exactly one of the propositions is true, but not both.

For example:

If P represents "I will go to the park" and Q represents "I will go to the cinema," then P⊕Q (read as "P xor Q") means "I will go to the park or the cinema, but not both."

P	**Q**	$\boldsymbol{P \oplus Q}$
T	T	F
T	F	T
F	T	T
F	F	F

3.7. NAND (NOT AND)

The **NAND** connective, represented by the symbol ↑, is a combination of negation and conjunction. The statement P↑Q (read as "P nand Q") is true unless both P and Q are true.

For example:

If P represents "It is sunny" and Q represents "I am outside," then P↑Q means "It is not the case that it is sunny and I am outside."

P	**Q**	$\boldsymbol{P \wedge Q}$	$\boldsymbol{P \uparrow Q}$
T	T	T	F
T	F	F	T
F	T	F	T
F	F	F	T

3.8. NOR (NOT OR)

The **N**OR connective, represented by the symbol ↓, is a combination of negation and disjunction. The statement P Q (read as "P nor Q") is true only if both P and Q are false. For example:

If P represents "It is sunny" and Q represents "I am outside," then P↓Q means "It is neither sunny nor am I outside."

P	**Q**	$P \vee Q$	$P \downarrow Q$
T	T	T	F
T	F	T	F
F	T	T	F
F	F	F	T

3.9. De Morgan's Laws

De Morgan's Laws are fundamental rules in logic and set theory that describe the relationship between logical connectives (AND, OR) and negation (NOT). These laws are essential for simplifying logical expressions and are widely used in mathematics, computer science, and digital circuit design.

- **First Law (Negation of a Conjunction):**
  The negation of a conjunction (AND) is equivalent to the disjunction (OR) of the negations.

$$\overline{P \wedge Q} \Leftrightarrow \overline{P} \vee \overline{Q}$$

- **Second Law (Negation of a Disjunction):**
  The negation of a disjunction (OR) is equivalent to the conjunction (AND) of the negations.

$$\overline{P \vee Q} \Leftrightarrow \overline{P} \wedge \overline{Q}$$

4. Language of Propositional Logic

The **language of propositional logic** (also called **sentential logic**) is a formal system used to represent and analyze logical relationships between propositions. It consists of a set of rules for constructing well-formed formulas (WFFs) using **propositional variables**, **logical connectives**, and **parentheses**.

Well-Formed Formulas (WFFs)

A **well-formed formula** is a syntactically correct expression in propositional logic. The rules for constructing WFFs are:

1. Every propositional variable is a WFF.

- Example: P, Q, R.

2. If P is a WFF, then ¬P is a WFF.
   - Example: ¬P, ¬(P∧Q).
3. If P and Q are WFFs, then (P∧Q), (P∨Q), (P→Q), and (P↔Q) are WFFs.

5. Limits of Propositional Logic

Propositional logic is a fundamental and powerful tool for reasoning about simple logical relationships, but it has several limitations when it comes to expressing more complex or nuanced ideas.

- **Cannot Analyze Internal Structure of Propositions:** Propositional logic treats propositions as indivisible units (atomic statements). It cannot analyze the internal structure of propositions, such as subjects, predicates, or quantifiers.

- **Limited Expressiveness:** Propositional logic is limited to expressing relationships between simple propositions using logical connectives (e.g., AND, OR, NOT). It cannot express more complex relationships or dependencies.

- **No Support for Quantifiers:** Propositional logic cannot express statements that involve quantifiers like "all," "some," or "none."

- **Cannot Represent Relationships Between Objects:** Propositional logic cannot represent relationships or properties of objects.

- **No Support for Modalities:** Propositional logic cannot express modalities like "necessarily," "possibly," "in the future," or "in the past."

- **Scalability Issues:** As the number of propositions increases, the complexity of propositional logic grows exponentially. This makes it impractical for large-scale problems.

- **No Support for Reasoning About Change:** Propositional logic is static and cannot represent or reason about changes over time.

- **Limited to Binary Truth Values:** Propositional logic assumes that propositions are either true or false. It cannot handle uncertainty, partial truth, or degrees of truth.

Extensions to Overcome Limitations

To address these limitations, more advanced logical systems have been developed:

- **Predicate Logic:** Extends propositional logic to include quantifiers, variables, and predicates.
- **Modal Logic:** Adds modalities like "necessarily" and "possibly."
- **Temporal Logic:** Incorporates time and reasoning about change.
- **Fuzzy Logic:** Allows for degrees of truth between 0 and 1.
- **First-Order Logic:** Combines predicate logic with quantifiers and relations.

6. First-order logic

First-order logic, also known as predicate logic or first-order predicate calculus, is a formal system used in mathematics, philosophy, linguistics, and computer science to express statements about objects and their relationships. It extends propositional logic by introducing quantifiers, variables, and predicates, allowing for a more expressive and precise representation of real-world phenomena.

First-order logic builds upon the foundation of propositional logic but goes further by enabling the representation of individual objects, properties, and relations between those objects. In propositional logic, statements are treated as atomic units (true or false), whereas first-order logic allows for the decomposition of these statements into smaller components. This is achieved through the use of **predicates**, which describe properties or relations, and **terms**, which represent objects or entities in the domain of discourse.

One of the main advantages of first-order logic is its balance between expressiveness and formal rigor. Unlike natural language, which can be ambiguous and context-dependent, first-order logic provides a precise and unambiguous framework for representing knowledge. This makes it particularly useful in fields such as artificial intelligence, where automated reasoning systems rely on formal representations to draw conclusions from given premises. However, first-order logic also has limitations, such as its inability to directly express certain higher-level concepts like sets of sets or self-referential statements. These limitations led to the development of second-order logic and other extensions, but first-order logic remains a cornerstone of modern logic due to its wide applicability and well-understood properties.

6.1. Predicate

A **predicate** is a fundamental concept in logic that represents a property or relation involving one or more objects (or variables) within a domain of discourse. It is used to make statements about these objects by expressing whether a particular property holds true for them or whether certain relationships exist between them.

In formal terms, a predicate is a function that maps elements (or tuples of elements) from a domain to truth values: **true** or **false**. Predicates are typically denoted by uppercase letters such as P, Q, or R, and they may take one or more arguments, which are usually represented by variables or constants.

Examples:

1. **Unary Predicate**:
    - T(x): "x is tall." Here, T is a predicate that evaluates to true if the object x has the property of being tall, and false otherwise.
2. **Binary Predicate**:
    - L(x,y): "x meets y." This predicate expresses a relationship between two individuals x and y. For example, L(Ahmed, Ali) means "Ahmed meets Ali."
3. **Ternary Predicate:**
    - G(x,y,z): "x is the sum of y and z." This predicate describes a three-way relationship, where x is the result, y and z are numbers.

6.2. Quantifiers

Quantifiers are essential components of **first-order logic** that allow us to make generalizations or specify the existence of elements within a domain of discourse. They extend the expressiveness of propositional logic by enabling statements about entire sets of objects, rather than just individual ones. There are two primary types of quantifiers: the **universal quantifier** (∀) and the **existential quantifier** (∃). Below is a detailed explanation of these quantifiers.

6.2.1. Universal Quantifier (∀)

The universal quantifier (∀) is used to express that a property or relation holds for **all** elements in a given domain. It can be read as "for all," "for every," or "for any."

- The universal quantifier asserts a property universally across the entire domain.
- If the domain is empty, $\forall x P(x)$ is vacuously true because there are no counterexamples.

Definition:

- $\forall x P(x)$: This means "For all x, P(x) is true."
- In other words, the predicate P(x) must hold for every possible value of x in the domain.

Example:

- Let H(x) mean "x is human" and M(x) mean "x is mortal." The statement "All humans are mortal" can be written as:

$$\forall x (H(x) \rightarrow M(x))$$

This reads: "For all x, if x is human, then x is mortal."

Existential Quantifier (∃)

The existential quantifier (∃) is used to express that there exists **at least one** element in the domain for which a property or relation holds. It can be read as "there exists," "for some," or "there is."

- The existential quantifier asserts the existence of at least one element satisfying the property.
- If the domain is empty, ∃xP(x) is always false because there are no elements to satisfy the property.

Definition:

- $\exists x P(x)$: This means "There exists at least one x such that P(x) is true."
- In other words, the predicate P(x) must hold for at least one value of x in the domain.

Example:

- Let P(x) mean "x is a prime number." The statement "There exists a prime number" can be written as:

$$\exists x P(x)$$

This reads: "There exists an x such that x is a prime number."

6.2.3. Combining Quantifiers

Quantifiers can be combined to express more complex statements. The order of quantifiers matters and can significantly affect the meaning of a statement.

Example 1: Universal Quantifier Followed by Existential Quantifier

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

"For every x, there exists a y such that R(x,y) holds." This means that for each x, we can find at least one y related to x.

Example 2: Existential Quantifier Followed by Universal Quantifier

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

"There exists a y such that for all x, R(x,y) holds." This means there is a single y that is related to every x.

Difference Between the Two:

- In $\forall x \exists y R(x, y)$, the y can depend on x.
- In $\exists y \forall x R(x, y)$, the y is fixed and independent of x.

6.2.4. Negation of Quantifiers

The negation of quantifiers involves switching between universal and existential quantifiers and applying De Morgan's laws.

Negation Rules:

1. $\neg(\forall x P(x)) \equiv \exists x \neg P(x)$

   "Not all x satisfy P(x)" is equivalent to "There exists an x such that P(x) is false."

2. $\neg(\exists x P(x)) \equiv \forall x \neg P(x)$

   "There does not exist an x such that P(x) is true" is equivalent to "For all x, P(x) is false."

Example:

Let E(x) mean "x is even." The negation of "All numbers are even" (∀xE(x)) is:

$$\neg(\forall x E(x)) \equiv \exists x \neg E(x)$$

This reads: "There exists a number that is not even."

7. Non-Classical Logics

Non-classical logics represent a family of formal systems that deviate from the fundamental principles of classical logic, particularly the laws of excluded middle and non-contradiction. While classical logic is grounded in a strict binary distinction between true and false, non-classical logics aim to address situations where this dichotomy is insufficient for adequately modeling reality or complex human reasoning. These systems expand the traditional logical framework to meet specific needs in various fields, such as philosophy, cognitive science, computer science, and even social sciences.

Classical logic is based on two fundamental postulates: every proposition is either true or false (the law of excluded middle), and no proposition can be both true and false simultaneously (the law of non-contradiction). However, there are many contexts where these assumptions do not align well with real-world phenomena or human thought processes. For instance, vague statements, uncertain information, incomplete knowledge, or paradoxical situations cannot always be adequately captured within the rigid structure of classical logic. Non-classical logics were developed to provide alternative frameworks that better accommodate such complexities.

One of the defining characteristics of non-classical logics is their flexibility in relaxing or modifying the principles of classical logic. These non-classical logics provide

powerful tools for addressing complexities that classical logic cannot handle, making them essential in fields ranging from philosophy and mathematics to computer science and artificial intelligence. This can take several forms:

7.1. Many-Valued Logics

- **Description**: Extend the binary true/false values of classical logic to include additional truth values, such as "unknown," "partially true," or values along a continuum.
- **Examples**:
    - **Fuzzy Logic**: Allows truth values to range between 0 (completely false) and 1 (completely true), enabling reasoning about vagueness and uncertainty.
    - **Three-Valued Logic (e.g., Łukasiewicz Logic)**: Introduces a third truth value, often interpreted as "unknown" or "indeterminate."

7.2. Paraconsistent Logics

- **Description**: Allow for the existence of contradictions without leading to triviality (i.e., not everything becomes true when a contradiction is present).
- **Examples**:
    - **Dialetheism**: Accepts that some statements can be both true and false simultaneously.
    - **Relevant Paraconsistent Logic (e.g., LP Logic)** : Ensures that contradictions do not spread uncontrollably throughout the system.

7.3. Intuitionistic Logic

- **Description**: Rejects the law of excluded middle and double negation elimination, emphasizing constructivist principles where truth is tied to provability rather than mere existence.
- **Applications**: Used in mathematical constructivism and computer science (e.g., type theory and proof assistants like Coq).

7.4. Modal Logics

- **Description**: Introduce modal operators (e.g., necessity □ and possibility ◊) to reason about concepts such as time, belief, obligation, or possible worlds.
- **Examples**:
    - **Alethic Modal Logic**: Deals with necessity and possibility.
    - **Epistemic Logic**: Models knowledge and belief.
    - **Deontic Logic**: Studies obligation, permission, and prohibition.
    - **Temporal Logic**: Reasons about time and temporal relationships.

7.5. Relevance Logics

- **Description**: Require that the premises of an argument are relevant to its conclusion, addressing issues with vacuous truths in classical logic.
- **Example**: Systems like **R** or **RW** ensure that irrelevant premises cannot lead to valid conclusions.

7.6. Fuzzy Logic

- **Description**: A form of many-valued logic specifically designed to handle vagueness and uncertainty by allowing partial membership in sets.
- **Applications**: Widely used in artificial intelligence, control systems, and decision-making processes.

7.7. Quantum Logics

- **Description**: Developed to model the peculiarities of quantum mechanics, where classical logical principles (e.g., distributivity) may fail.
- **Example**: Quantum propositional logic replaces classical Boolean algebra with orthomodular lattices.

7.8. Substructural Logics

- **Description**: Relax structural rules of classical logic, such as contraction, weakening, or exchange, to model resource-sensitive reasoning.
- **Examples**:
    - **Linear Logic**: Focuses on resource management and consumption.
    - **Lambek Calculus**: Used in linguistics to model syntactic structures.

7.9. Default Logic

- **Description**: A formalism for reasoning with incomplete information, allowing for default assumptions unless contradicted by evidence.
- **Applications**: Common in artificial intelligence and expert systems.

7.10. Non-Monotonic Logics

- **Description**: Enable reasoning where adding new information can invalidate previous conclusions, mimicking human common-sense reasoning.
- **Examples**:
    - **Autoepistemic Logic**: Models self-referential reasoning about knowledge.
    - **Circumscription**: Minimizes the extension of predicates to avoid unnecessary assumptions.

Exercises

A) Exercises on Propositional Logic

1. Identify Propositions

Determine whether the following sentences are propositions. If they are, indicate their truth value (if possible).

a) "Paris is the capital of France."
b) "What is your name?"
c) "2 + 2 = 5."
d) "This statement is false."
e) "It will rain tomorrow."

2. Negation of Propositions

Write the negation of the following propositions:

a) "The number 10 is even."
b) "All birds can fly."
c) "Some students like mathematics."
d) "If it rains, then the ground will be wet."

3. Logical Operators

Let P be "It is raining," and Q be "I will stay at home." Express the following statements using logical operators:

a) "It is raining, and I will stay at home."
b) "If it is raining, then I will stay at home."
c) "I will stay at home if and only if it is raining."
d) "It is not raining, or I will not stay at home."

4. Truth Tables

Construct truth tables for the following compound propositions:

a) $P \land \neg Q$
b) $P \lor (Q \rightarrow R)$
c) $(P \leftrightarrow Q) \land \neg R$
d) $\neg(P \lor Q) \rightarrow (P \land Q)$

5. Logical Equivalence

Determine whether the following pairs of propositions are logically equivalent:

a) $P \rightarrow Q \ and \ \neg P \lor Q$
b) $\neg(P \land Q) \ and \ \neg P \lor \neg Q$
c) $P \leftrightarrow Q \ and \ (P \rightarrow Q) \land (Q \rightarrow P)$
d) $P \lor (Q \land R) \ and \ (P \lor Q) \land (P \lor R)$

6. Translating Sentences into Propositional Logic

Translate the following sentences into propositional logic:

a) "If it is sunny, then I will go to the beach."
b) "I will go to the party only if I finish my homework."
c) "Either I will study for the exam, or I will fail."
d) "It is not true that both John and Mary are coming to the meeting."

7. Constructing Compound Propositions

Given the propositions:

- P: "It is cold."
- Q: "It is snowing."
- R: "I will wear a coat."

Construct compound propositions for the following scenarios:

a) "If it is cold and snowing, then I will wear a coat."
b) "I will wear a coat if and only if it is cold or snowing."
c) "It is not cold, but it is snowing, and I will not wear a coat."
d) "If it is not snowing, then I will wear a coat only if it is cold."

8. Analyzing Logical Statements

Analyze the following statements and determine whether they are tautologies, contradictions, or contingencies:

a) $P \vee \neg P$

b) $P \wedge \neg P$

c) $(P \rightarrow Q) \leftrightarrow (\neg Q \rightarrow \neg P)$

d) $(P \wedge Q) \vee (\neg P \wedge \neg Q)$

B) Exercises on Predicate Logic

1. Translating Sentences into Predicate Logic

Translate the following sentences into predicate logic. Define the predicates and domains clearly.

a) "Every student in the class is intelligent."
b) "Some birds cannot fly."
c) "All cats are mammals, and some mammals are pets."
d) "There exists a person who is loved by everyone."
e) "No two people have the same birthday."

2. Evaluating Truth Values

Given the following predicates and domains, determine whether the predicate logic statements are true or false.

- Domain: All integers.
- P(x): "x is even."
- Q(x): "x is positive."
- R(x,y): "x is greater than y."

a) $\forall x\ (P(x) \rightarrow Q(x))$
b) $\exists x\ (P(x) \wedge Q(x))$
c) $\forall x \exists y\ R(x, y)$
d) $\exists x \forall y\ R(x, y)$

3. Constructing Predicate Logic Statements

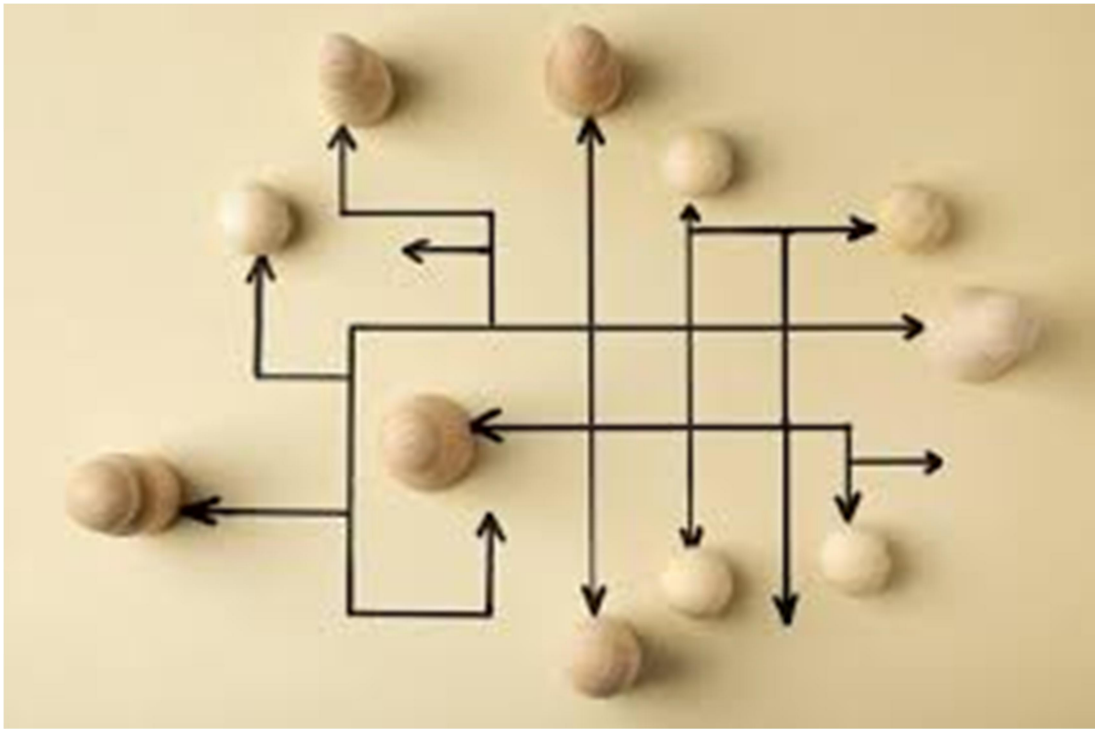
Given the following predicates and domains, construct predicate logic statements for the given scenarios.

- Domain: All people.
- P(x): "x is a student."
- Q(x): "x is hardworking."
- R(x,y): "x likes y."

a) "Every student is hardworking."
b) "Some hardworking students like everyone."
c) "There is a student who is liked by all hardworking students."
d) "No student likes every hardworking student."

Chapter 3

Search Algorithms for Problem Solving



1. Introduction

Many AI problems are hard to solve because they are difficult to be characterized. Not only the problem but a path to the solution is also hard to be characterized. Problem solving basically involves doing the right thing at the right time. Given a problem to solve the task is to select the right moves that would lead to the solution.
In this chapter we will learn the process of solving AI problems using state space search. We will also see the difficulties a designer might face.

Importance of search algorithms for problem-solving

- **Efficiency and Optimization:** Search algorithms are essential for efficiently managing resources and improving system performance. They help in minimizing the use of computational resources like time and memory, making systems faster and more reliable. By finding optimal solutions, search algorithms enable efficient resource management and enhance overall performance.

- **Decision Making and Strategic Planning:** Search algorithms play a crucial role in informed decision-making by providing the best possible solutions based on available data and constraints. They are used to explore different scenarios and evaluate outcomes, aiding in strategic planning and risk assessment. This capability is particularly valuable in fields like healthcare, finance, and manufacturing, where optimal decisions can have significant impacts.

- **Problem-Solving in AI and Complex Problems:** In artificial intelligence, search algorithms are fundamental for pathfinding in robotics, game development, and navigation systems. They are also used in planning and scheduling tasks, such as job scheduling and logistics optimization. Additionally, search algorithms are essential for solving complex combinatorial problems like the traveling salesman problem and constraint satisfaction problems, where the goal is to find solutions that satisfy all given constraints.

- **Data Retrieval and Pattern Recognition:** Search algorithms are at the core of information retrieval systems, such as search engines, databases, and recommendation systems. They are also used in pattern recognition tasks, such as image and speech recognition, where the goal is to find patterns in data. This makes them indispensable for applications that rely on efficient data retrieval and accurate pattern recognition.

- **Optimization and Real-World Applications:** Search algorithms are used in various optimization techniques, such as linear programming, dynamic programming, and genetic algorithms. They help in finding global optima in complex optimization problems, ensuring that the best possible solution is found. In real-world applications, search algorithms are used in healthcare for diagnosis and treatment planning, in finance for portfolio optimization and risk management, and in manufacturing for supply chain optimization and production planning.

- **Scalability and Adaptability:** Search algorithms can handle large-scale problems efficiently, making them suitable for big data applications and real-time systems. They can be designed to take advantage of parallel processing, further enhancing their scalability and performance. Additionally, search algorithms can adapt to dynamic environments, making them suitable for real-time decision-making and problem-solving in changing conditions. They are robust and can handle uncertainty and variability in data and problem constraints.

2. Definitions

2.1. Problem definition

A problem is a situation, condition, or issue that is perceived as difficult, confusing, or undesirable and requires a resolution or solution.

2.2. Problem for AI

A problem is a well-defined, often formalized, challenge or task that an AI system is designed to address or solve. The definition and formulation of the problem are crucial steps in designing and implementing effective AI solutions.

AI problems have several key characteristics:

- **Well-Defined Objective**: An AI problem typically has a clear, quantifiable goal or objective that the AI system aims to achieve, such as maximizing a reward signal, minimizing a cost function, or finding an optimal solution.
- **Formalized Input and Output**: AI problems are usually formalized with a specific input format (e.g., data, state, or environment) and a desired output format (e.g., prediction, decision, or action).
- **Constraints and Rules**: AI problems often have constraints, rules, or limitations that the system must adhere to while solving the problem, such as time limits, resource constraints, or domain-specific rules.

- **Data-Driven**: Many AI problems involve processing and analyzing data to extract insights, make predictions, or support decision-making.
- **Uncertainty and Complexity**: AI problems often involve dealing with uncertainty, incomplete information, or complex environments, requiring the AI system to make probabilistic inferences or learn from experience.

2.3. Examples of AI problems

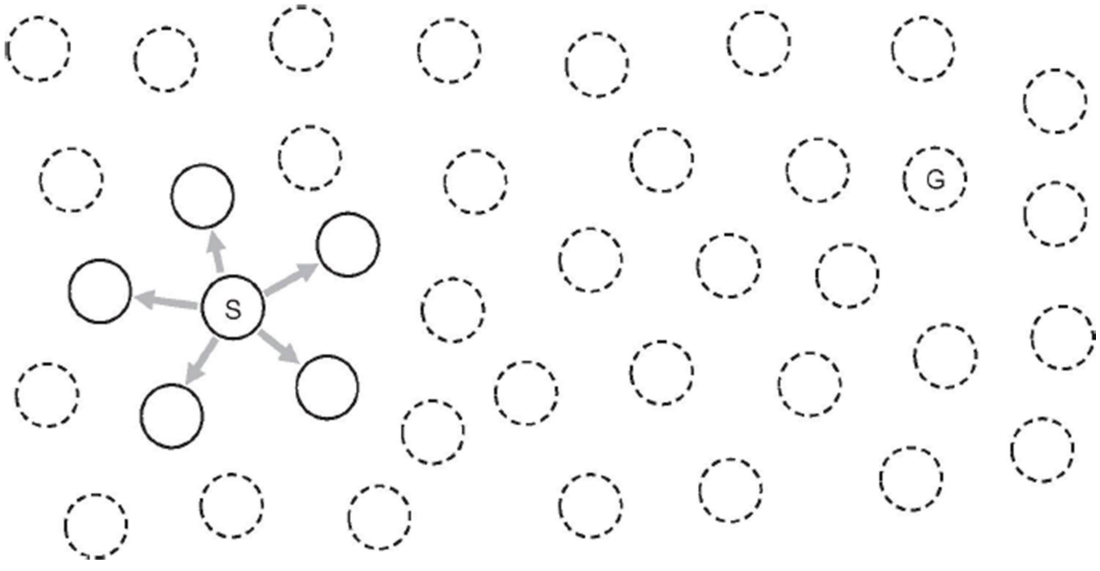
- **Classification**: Assigning a label or category to input data, such as spam detection or image recognition.
- **Regression**: Predicting a continuous value based on input data, such as house price prediction.
- **Planning and Scheduling**: Finding an optimal sequence of actions to achieve a goal, such as route planning or job scheduling.
- **Optimization**: Finding the best solution from a set of possible solutions, such as resource allocation or portfolio optimization.

3. The State Space

3.1. Definition

A state space is a set of all possible configurations or states that a system can be in. Each state represents a specific situation or condition of the system, and the state space encompasses the entire range of these situations. The concept of a state space is fundamental to various AI techniques, including search algorithms, planning, and optimization.

The problem solver begins in a given or start state and has some desired state as a goal state. The desired state can be a single state that is completely specified, or it may be a set of states described by a common property.



Understanding and effectively navigating the state space is a crucial aspect of solving many AI problems. The design of the state space, the choice of operators, and the

selection of search strategies are all critical factors in developing efficient and effective AI solutions.

3.2. Components of a State Space

a) States: Individual configurations or conditions of the system. Each state is a unique snapshot of the system at a particular point in time.
b) Initial State: The starting point or the initial configuration of the system. This is where the search or problem-solving process begins.
c) Goal State: The desired outcome or the final configuration that the system aims to reach. The objective of the search or problem-solving process is to find a path from the initial state to the goal state.
d) Operators or Actions: The set of actions or transitions that can be applied to move from one state to another. These operators define how the system can change from one state to another.
e) Transitions: The rules or functions that describe how applying an operator changes the state of the system. Transitions define the relationships between states.

Example

Consider a real-world example of a state space search problem: **Navigating a Maze**.

- **State**: The current position of the agent (e.g., a robot) within the maze.
- **Initial State**: The starting position of the agent at the entrance of the maze.
- **Goal State**: The target position, usually the exit of the maze.
- **Transition**: Moving from one position to an adjacent position (up, down, left, or right) within the maze.
- **Path**: A sequence of moves from the initial state to the goal state.

Imagine a simple grid maze where the agent starts at the top-left corner (0,0) and needs to reach the bottom-right corner (3,3). The agent can move in four directions: up, down, left, and right, but cannot pass through walls.

1. **Initial State**: (0,0)
2. **Goal State**: (3,3)
3. **Possible Transitions**:
    - From (0,0) to (0,1) or (1,0)
    - From (0,1) to (0,2) or (1,1) or back to (0,0)

- And so on…

The agent will explore different paths, keeping track of visited states to avoid cycles, until it finds a path that leads to the goal state.

3.3. Problem solving by State space search

State space search in AI is a fundamental technique used to solve problems by navigating through a series of states and transitions. In this approach, a problem is represented as a collection of states, each depicting a specific configuration, and the transitions represent possible actions or moves between these states. The objective is to find a sequence of actions that leads from an initial state to a goal state.

This concept is analogous to finding a path through a complex maze: each decision or action leads to a new state, and the goal is to discover the optimal sequence of actions that leads to a desired outcome.

By applying state space search, AI systems can effectively tackle a diverse array of problems, ranging from robotics and game-playing to natural language processing and scheduling. It serves as a crucial tool for enabling machines to make intelligent decisions and find optimal solutions in complex, dynamic environments.

3.4. Principles and Features of State Space Search

The efficiency and effectiveness of state space search are heavily dependent on several principles and characteristics. Understanding these elements is crucial for selecting the right search strategy and optimizing the search process.

a) **Expansiveness**: The number of successors that each state can generate. This impacts how many new states are explored from a given state.
b) **Branching Factor**: The average number of successors in each state. It influences the width of the search tree and the overall complexity of the search.
c) **Depth**: The length from the initial state to the goal state in the search tree. Deeper search trees can increase the time required to find a solution.
d) **Completeness**: A search strategy is complete if it guarantees finding a solution, assuming one exists.
e) **Optimality**: A search strategy is optimal if it guarantees finding the best solution according to a specified criterion.
f) **Time Complexity**: The duration of the state space exploration. It is influenced by the branching factor and the depth of the search tree.
g) **Space Complexity**: The amount of memory required to carry out the search. This depends on the number of states that need to be stored in memory simultaneously.

3.5. Problem Representation

A number of factors need to be taken into consideration when developing a state space representation. Factors that must be addressed are:

- What is the goal to be achieved?
- What are the legal moves or actions?
- What knowledge needs to be represented in the state description?
- Type of problem - There are basically three types of problems. Some problems only need a representation, e.g. crossword puzzles. Other problems require a yes or no response indicating whether a solution can be found or not. Finally, the last type problem are those that require a solution path as an output, e.g. mathematical theorems, Towers of Hanoi. In these cases, we know the goal state and we need to know how to attain this state
- Best solution vs. Good enough solution - For some problems a good enough solution is sufficient. For example, theorem proving, eight squares.
- However, some problems require a best or optimal solution, e.g. the traveling salesman problem.

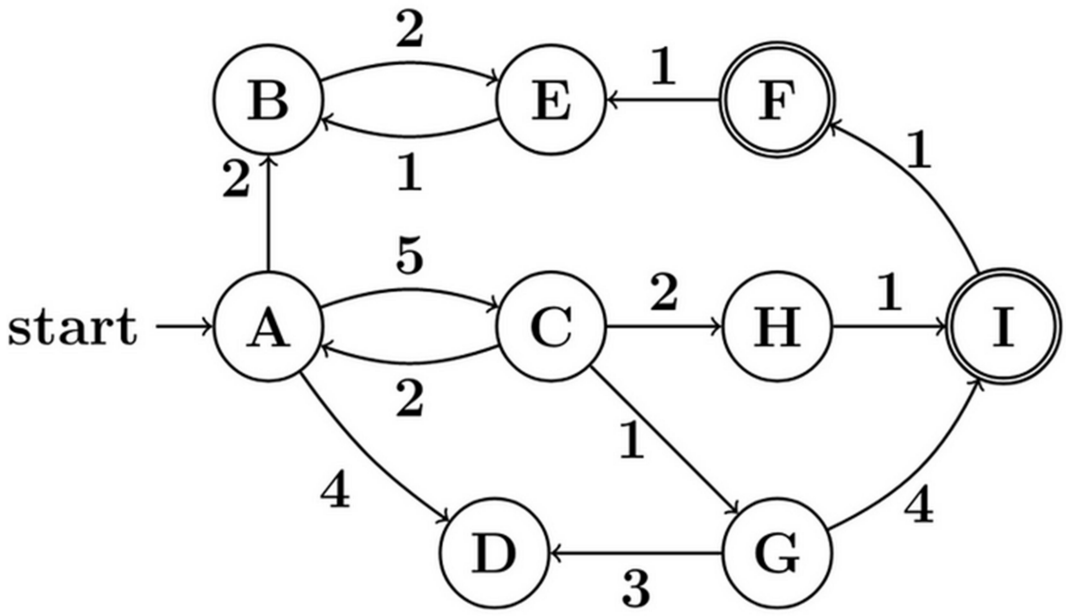
3.6. Representation of a State Space

Representing a state space is a fundamental step in designing algorithms for problem-solving, planning, and optimization in Artificial Intelligence (AI). The representation of a state space involves defining the states, the transitions between states, and the structure of the space. The choice of representation depends on the nature of the problem and the specific requirements of the AI system.

3.6.1. Graph Representation

A graph is a natural and widely used representation for a state space. In this representation:

- Nodes: Represent the states of the system.
- Edges: Represent the transitions or actions that move the system from one state to another.
- Labels: Edges can be labeled with the actions or costs associated with the transitions.

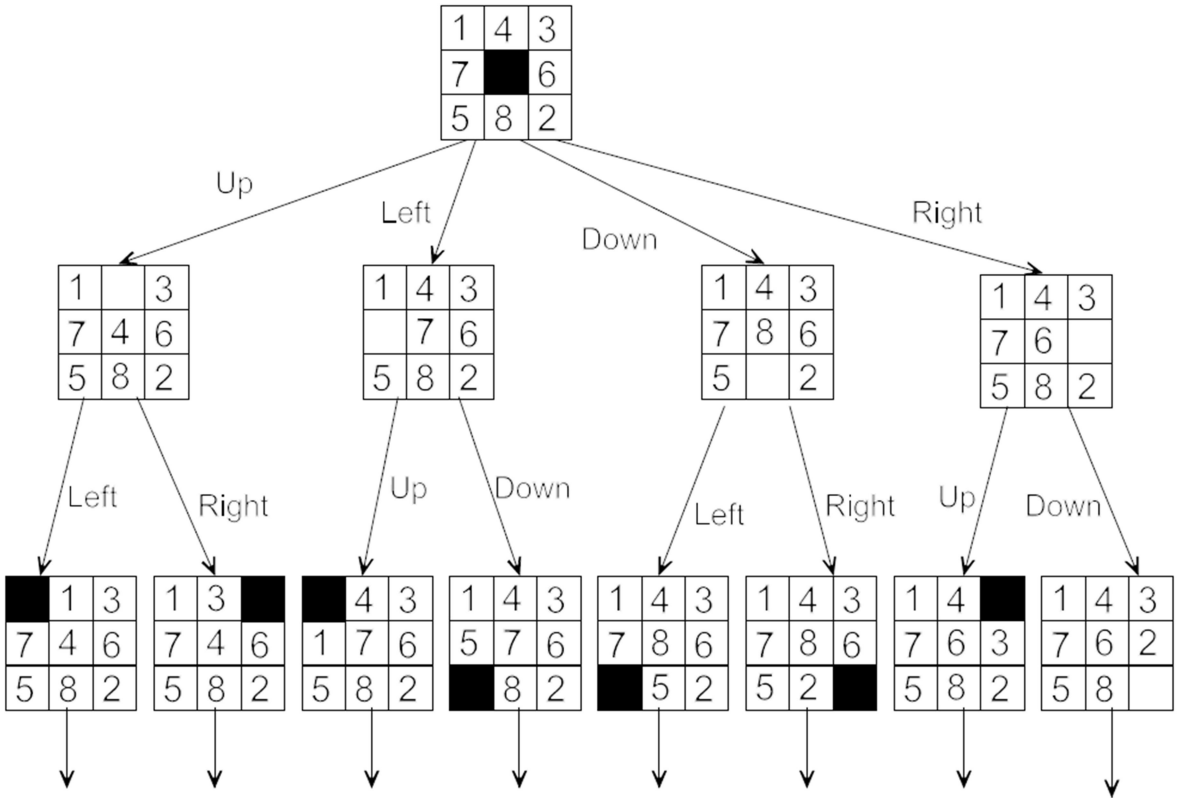


3.6.2. Tree Representation

A tree is a special type of graph where each node has a single parent (except the root node) and can have multiple children. Trees are useful for representing hierarchical or recursive state spaces.

- Root Node: Represents the initial state.
- Leaf Nodes: Represent terminal states, including the goal state.
- Internal Nodes: Represent intermediate states.
- Edges: Represent transitions between states.

Example: Eight-Puzzle Problem state space representation by a tree



3.6.3. State Vectors

In some cases, states can be represented as vectors of attributes or features. This is common in problems where the state can be described by a fixed set of variables.

- State Vector: A tuple or array of values representing the state.
- Transitions: Functions that modify the state vector based on actions.

3.6.4. Matrix Representation

For problems with a fixed, discrete state space, a matrix can be used to represent the states and transitions.

- Matrix: A 2D array where each cell represents a state.
- Transitions: Rules or functions that define how to move from one cell to another.

3.6.5. Symbolic Representation

In some cases, states can be represented symbolically using logical expressions or other formal languages.

- States: Symbolic expressions or formulas.
- Transitions: Rules or axioms that define how to transform one expression into another.

3.6.6. Object-Oriented Representation

In object-oriented programming, states can be represented as objects with attributes and methods.

- State Object: An object with attributes representing the state.
- Methods: Functions that define transitions between states.

4. Steps in State Space Search

The following steps are often involved in the state space search process:

Step 1: Define the State Space

Determine the collection of all potential states and their interchanging states. To do this, the problem must be modelled in a fashion that encompasses all pertinent configurations and actions.

Step 2: Pick a Search Strategy

Decide how to comb over the state space. Typical tactics consist of:

- Before going on to nodes at the following depth level, the **Breadth-First Search (BFS)** method investigates every node at the current depth level. Full and ideal for graphs without weights.
- **Depth-First Search (DFS) i**nvestigates a branch as far as it can go before turning around. less memory-intensive, although completeness and optimality are not assured.
- The best method for locating the lowest-cost solution is **Uniform Cost Search (UCS)**, which expands the least expensive node first.
- **Greedy Best-First Search e**xpands the node that seems to be closest to the objective using a heuristic.
- **A* Search Algorithm** assures completeness and optimality with an admissible heuristic by combining the cost to reach the node with a heuristic calculating the cost to the target.

Step 3: Start the Search

Add the initial state to the frontier (the collection of states to be investigated) by starting there.

Step 4: Extend the Nodes

Using the selected search technique, iteratively expands nodes from the frontier, producing successor states and appending them to the frontier. After each node has been expanded, determine whether it now corresponds to the desired state. If so, retrace your route to the objective and call off the hunt.

Step 5: Address State Repetition

Put in place safeguards to prevent revisiting the same state, including keeping track of the states you've been to.

Step 6: End the Search

The search comes to an end when the desired state is discovered or, in the event that no viable solution is identified, when every state has been investigated.

AI systems are able to tackle complicated issues in an organized and methodical manner by employing these methods to systematically explore the state space.

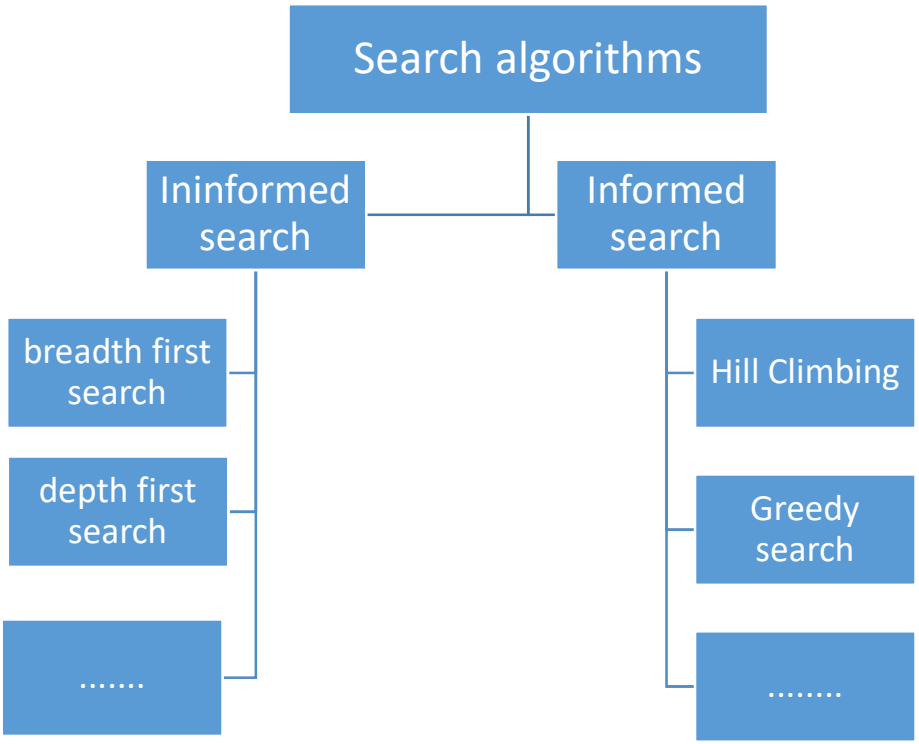
5. Measuring problem-solving performance

The performance of the search algorithms is evaluated on the following four criteria.

- **Completeness**. An algorithm is said to be complete if it is guaranteed to find a goal state if one is reachable. We also call such algorithms as being systematic. By this we mean that the algorithm searches the entire space and returns fail only if a goal state is not reachable.
- **Quality of solution (optimality)**. We may optionally specify a quality measure for the algorithm. We begin with the length of the path found as a measure of quality. Later, we will associate edge costs with each move, and then the total cost of the path will be the measure.
- **Space complexity**. This looks at the amount of space the algorithm requires to execute. We will see that this will be a critical measure, as the number of candidates in the search space often grows exponentially.
- **Time complexity**. This describes the amount of time needed by the algorithm, measured by the number of candidates inspected. The most desirable complexity will be linear in path length, but one will have to often contend with exponential complexity.

6. Types of search algorithms

Search algorithms are fundamental tools in computer science and artificial intelligence used to find specific data or solutions within a dataset or problem space. They can be broadly categorized into two main types: **uninformed search algorithms** and **informed search algorithms**. Each type has its own characteristics, advantages, and use cases, depending on the nature of the problem and the available information.



1. **Uninformed search algorithm (Blind search)**: is a search that has no information about its domain or nature of the problem.
2. **Informed search algorithm (Heuristic search)**: have further information about the cost of the path between any state in search space and the goal state.

6.1. Uninformed search algorithms

Uninformed search algorithms are a fundamental class of search strategies in artificial intelligence (AI). These algorithms operate without any additional information about the problem domain beyond what is explicitly provided in the problem definition. With these approaches, nothing is presumed to be known about the state space. Instead, they rely only on the structure of the problem space and the goal state.

The principal algorithms that fall under this heading are the **depth first search (DFS) and Breadth first search (BFS)**. These algorithms share two properties:

- They do not use heuristic measures in which the search would proceed along the most promising path.
- Their aim is to find some solution to the given problem.

Challenges and limitations

Despite their simplicity and versatility, uninformed search algorithms face several challenges and limitations.

- inefficient in complex problems with large search spaces, leading to an exponential increase in the number of states explored. They may also consume significant computational resources and memory.
- **Lack of Optimality:** Uninformed search algorithms do not guarantee an optimal solution, as they do not consider the cost of reaching the goal or other relevant information.
- **Potential for Infinite Loops:** Algorithms like DFS can become trapped in infinite loops if the search space is too deep or if there are cycles in the graph.

6.1.1. Breadth First Search (BFS)

Breadth-first search (BFS) is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

In BFS, nodes are visited level by level from the top of the tree to the bottom, in left to right. All nodes at level *i* are visited before any nodes at level *i+1* are encountered.

BFS is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.

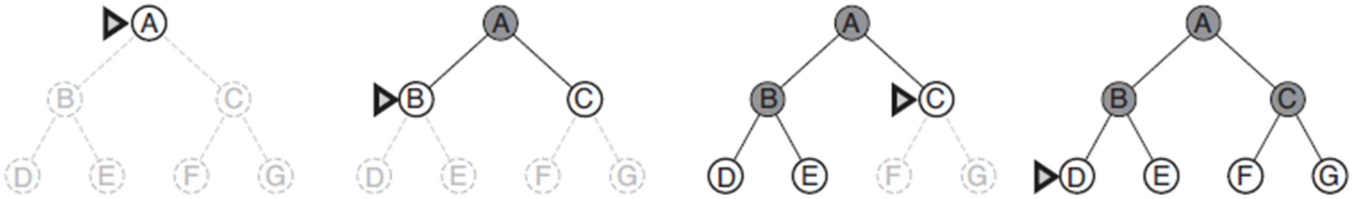
Breadth-first search Pseudocode

function BREADTH-FIRST-SEARCH(problem) **returns** a solution, or failure

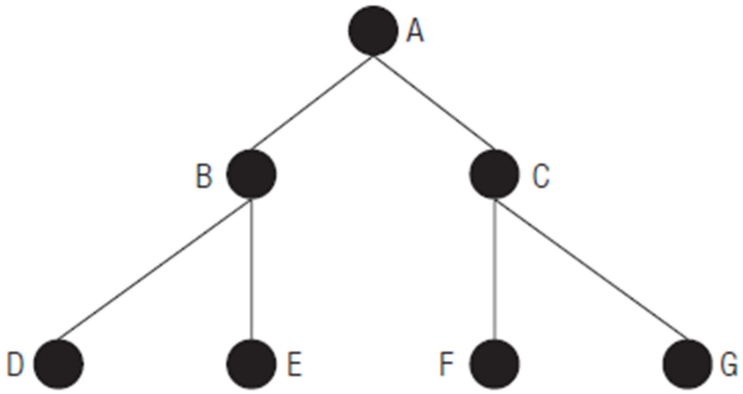
```
node ←a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
frontier ←a FIFO queue with node as the only element
explored ←an empty set
loop do
    if EMPTY?( frontier) then return failure
    node←POP( frontier ) /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
            child ←CHILD-NODE(problem, node, action)
            if child .STATE is not in explored or frontier then
```

if problem.GOAL-TEST(child .STATE) **then return** SOLUTION(child)
frontier ←INSERT(child , frontier)

Figure bellow shows the progress of the search on a simple binary tree.



The Figure below shows Breadth-first traversal of a tree.



The nodes will be visited in the order: **A, B, C, D, E, F, G.**

How does breadth-first search rate according to the **four criteria** from the previous section? We can easily see that it is ***complete***—if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite). Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test. Now, the *shallowest* goal node is not necessarily the *optimal* one; technically, breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node. The most common such scenario is that all actions have the same cost.

So far, the news about breadth-first search has been good. The news about time and space is not so good. Imagine searching a uniform tree where every state has b successors.

The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of *these* generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is

$$b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

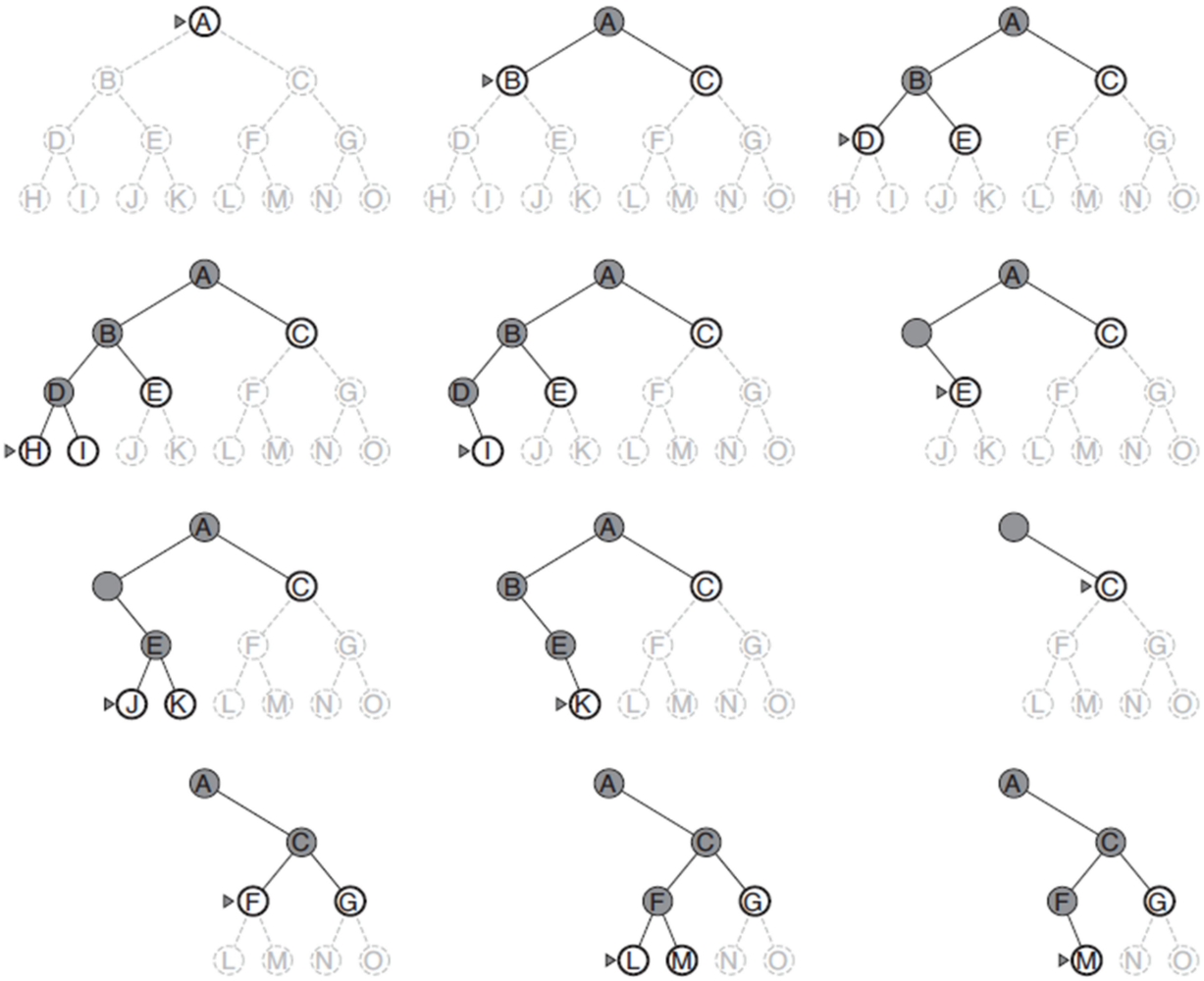
An exponential complexity bound such as $O(b^d)$ is scary. Table shows why. It lists, for various values of the solution depth d, the time and memory required for a breadth first search with branching factor b = 10. The table assumes that 1 million nodes can be
generated per second and that a node requires 1000 bytes of storage. Many search problems fit roughly within these assumptions (give or take a factor of 100) when run on a modern personal computer.

Depth	Nodes	Time	Memory
2	110	0.11 milliseconds	107 kilobytes
4	11100	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

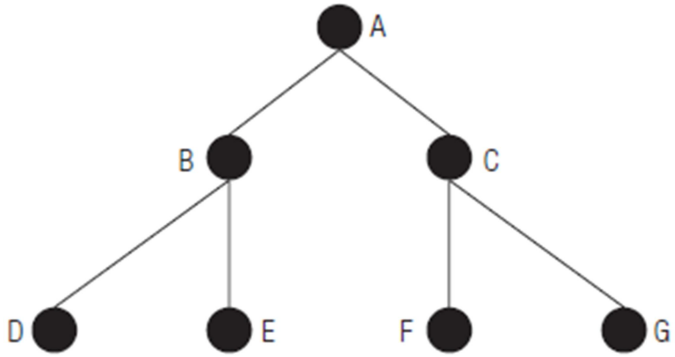
6.1.2 Depth First Search (DFS)

Depth first search (DFS) attempts to plunge as deeply into a tree as quickly as possible.
Whenever the search can make a choice, it selects the far left (or far right).
A DFS plunges depth first into a graph without regard for which edge it takes next until it cannot go any further at which point it backtracks and continues.

The progress of the search is illustrated in Figure bellow. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.



As an example of DFS, consider the tree in the Figure below. Tree traversal algorithms often "visit" a node several times.



The DFS encounters nodes in the following order: **A, B, D, E, C, F, G**.

whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion.

Depth-first search Pseudocode

```
DFS-iterative (G, s):                    //Where G is graph and s is source vertex
    let S be stack
    S.push( s )          //Inserting s in stack
    mark s as visited.
    while ( S is not empty):
      //Pop a vertex from stack to visit next
      v  =  S.top( )
     S.pop( )
     //Push all the neighbours of v in stack that are not visited
     for all neighbours w of v in Graph G:
       if w is not visited :
             S.push( w )
            mark w as visited


  DFS-recursive(G, s):
     mark s as visited
     for all neighbours w of s in Graph G:
       if w is not visited:
          DFS-recursive(G, w)
```

The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used. The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node. **The tree-search version, on the other hand, is *not* complete**.

For similar reasons, both versions are nonoptimal. For example, in the first Figure, depth first search will explore the entire left subtree even if node C is a goal node. If node J were also a goal node, then depth-first search would return it as a solution instead of C, which would be a better solution; hence, depth-first search is not optimal.

The time complexity of depth-first graph search is bounded by the size of the state space (which may be infinite, of course). A depth-first tree search, on the other hand, may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space. Note that

m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.

6.1.3. Comparison between DFS and BFS

The choice between DFS and BFS depends on the specific problem you are trying to solve. DFS is often used for tasks like cycle detection and topological sorting, while BFS is preferred for finding the shortest path in unweighted graphs and for pathfinding tasks in networks.

Traversal Order

- **DFS**: Explores nodes as deep as possible along each branch before backtracking.
- **BFS**: Explores all neighboring nodes at the present depth level before moving on to nodes at the next depth level.

Memory Usage

- **DFS**: Uses a stack (or recursion), which can be problematic for very deep graphs due to the depth of recursion.
- **BFS**: Uses a queue, which can be problematic for very wide graphs due to the width of the queue.

Time Complexity

- **DFS and BFS**: Both algorithms have a time complexity of $O(V+E)$, where V is the number of nodes and E is the number of edges.

Applications

- **DFS**:
  - Cycle detection in a graph.
  - Topological sorting.
  - Traversing undirected graphs.
- **BFS**:
  - Finding the shortest path in an unweighted graph.
  - Detecting connected components in an undirected graph.
  - Pathfinding algorithms in networks.

Advantages and Disadvantages

- **DFS**:
  - **Advantages**: Less memory required for deep graphs.
  - **Disadvantages**: Can enter infinite loops if the graph contains cycles.

- **BFS**:
    - **Advantages**: Finds the shortest path in an unweighted graph.
    - **Disadvantages**: Requires more memory for wide graphs.

6.2. Informed search algorithms

Informed search algorithms, also known as **heuristic search algorithms**, are a crucial component of artificial intelligence (AI). These algorithms leverage additional information, often in the form of heuristics, to guide the search process more efficiently towards a solution.

Informed search algorithms use domain-specific knowledge to improve the efficiency of the search process. This additional information, typically provided by heuristic functions, helps the algorithm make educated guesses about which paths to explore, thereby reducing the time and computational resources required to find a solution.

The goal of heuristic search methods is to greatly reduce the number of nodes considered in order to reach a goal state. They are ideally suited for problems whose **combinatorial complexity** grows very quickly.

Heuristic search is useful in solving problem which:

- Could not be solved any other way
- Solution takes an infinite time or very long time to compute

Local search algorithms

The search algorithms that we have seen before are designed to explore search spaces systematically.

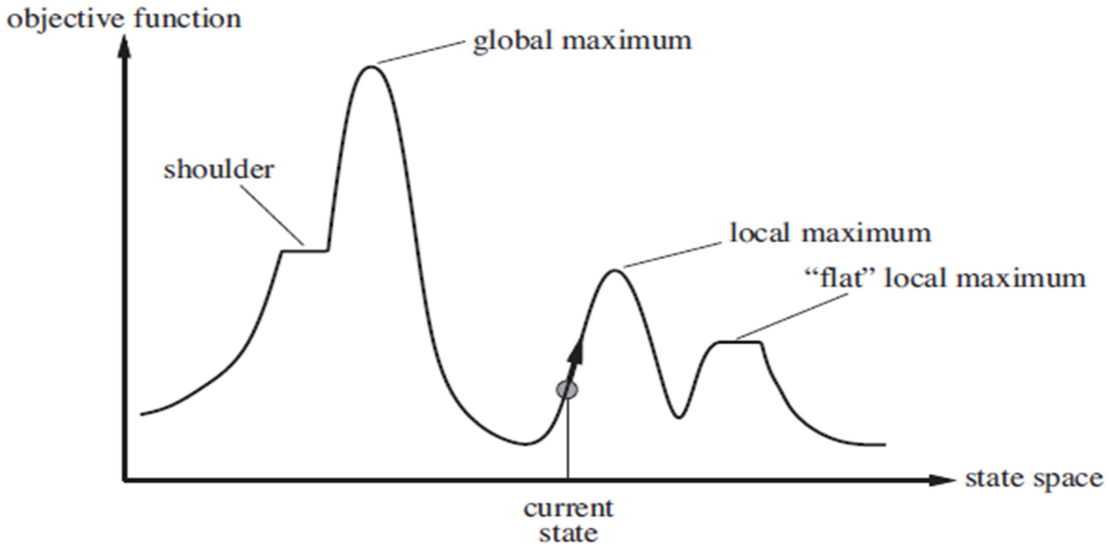
This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path. When a goal is found, the path to that goal also constitutes a solution to the problem. In many problems, however, the path to the goal is irrelevant.

If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all. **Local search** algorithms operate using a single **current node** (rather than multiple paths) and generally move only to neighbors of that node. Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages:

- they use very little memory—usually a constant amount; and
- they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

To understand local search, we find it useful to consider the **state-space landscape** (as in Figure below). A landscape has both "location" (defined by the state) and

"elevation" (defined by the value of the heuristic cost function or objective function). If elevation corresponds to cost, then the aim is to find the lowest valley—a **global minimum**; if elevation corresponds to an objective function, then the aim is to find the highest peak—a **global maximum**. Local search algorithms explore this landscape. A **complete** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a global minimum/maximum.



- Global Maximum: It is the highest point on the hill, which is the goal state.
- Local Maximum: It is the peak higher than all other peaks but lower than the goal maximum.
- Flat Local Maximum: It is a region in the search space where the objective function reaches a maximum value and remains relatively constant over a range of input values.
- Shoulder (Plateau) : It refers to a region in the search space where the objective function has a relatively flat gradient, meaning that small changes in the variables do not significantly affect the function's value.

6.2.1. Hill Climbing

The **hill-climbing** search is an iterative algorithm that continually moves in the direction of increasing value. It terminates when it reaches a "peak" where no neighbor has a higher value.

It starts with an arbitrary solution to a problem and iteratively makes small changes to improve the solution. It only considers the local neighborhood of the current solution.

The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function. Hill climbing

does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount in a thick fog.
Hill climbing is very useful in routing-related problems like travelling salesmen problem, job scheduling, chip designing, and portfolio management.

Algorithm for Simple Hill Climbing:

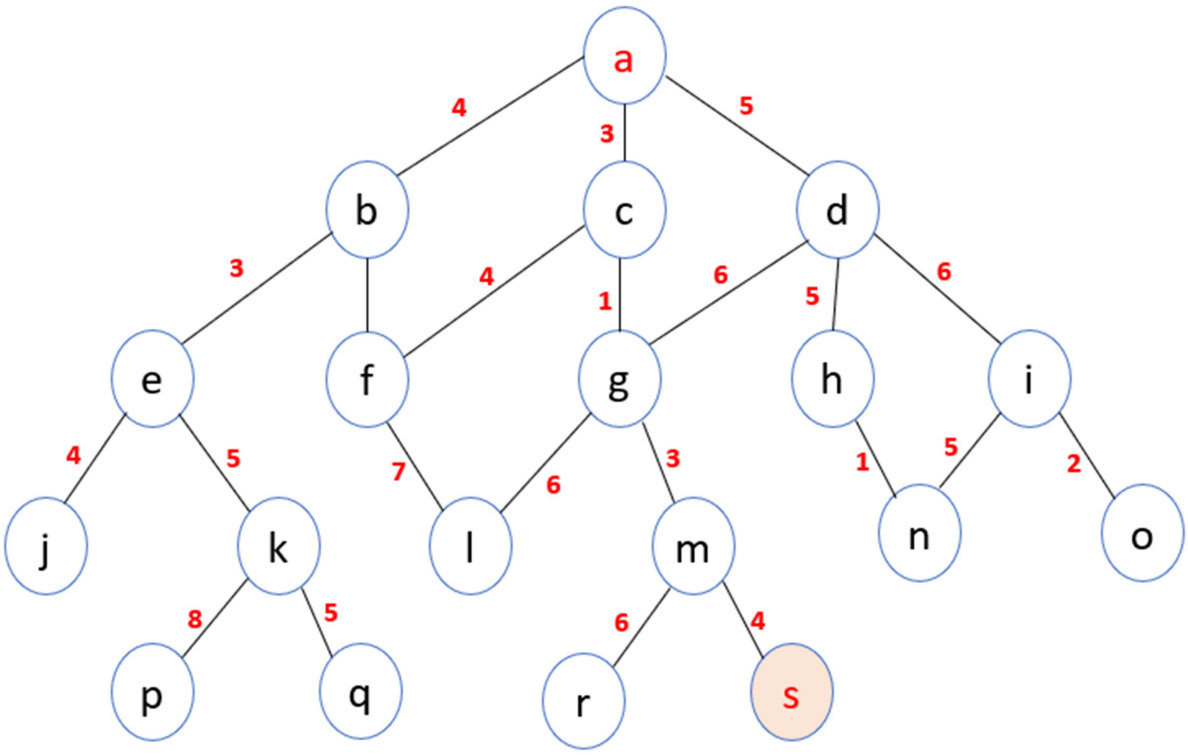
Step 1: Start with an initial state.
Step 2: Check if the initial state is the goal. If so, return success and exit.
Step 3: Enter a loop to search for a better state continuously.

- Select a neighboring state within the loop by applying an operator to the current state.
- Evaluate this new state:
  - If it's the goal state, return success and exit.
  - If it's better than the current state, update the current state to this new state.
  - If it's not better, discard it and continue the loop.

Step 4: End the process if no better state is found and the goal isn't achieved.

Pseudocode for Hill Climbing

```
function HillClimbing(initial_solution):
  current_solution = initial_solution
  while True:
    neighbors = generate_neighbors(current_solution)
    best_neighbor = current_solution
    for neighbor in neighbors:
      if evaluate(neighbor) > evaluate(best_neighbor):
        best_neighbor = neighbor
    if evaluate(best_neighbor) <= evaluate(current_solution):
      break
    current_solution = best_neighbor
  return current_solution
```



Open	Stack	Closed
[a]	a	[ ]
[b4, c3, d5]	c	[a]
[f4, g1, b4, d5]	g	[a, c]
[l6, m3, f4, b4, d5]	m	[a, c, g]
[r6, s4, l6, f4, b4, d5]	s	[a, c, g, m]
		[a, c, g, m, s]

Goal found: [a, c, g, m, s]

Variants of Hill Climbing

- Simple Hill Climbing: The basic version described above, where the first better neighbor found is accepted.

- Steepest Ascent Hill Climbing: Instead of accepting the first better neighbor, this algorithm examines all the neighboring nodes of the current state and selects the one that provides the greatest improvement in the objective function. This algorithm consumes more time as it searches for multiple neighbors.

- Stochastic Hill Climbing: This variant introduces randomness by selecting a random neighbor and accepting it if it is better than the current solution.

Advantages and Disadvantages

Advantages:

- Simple and easy to implement.
- Effective for finding local optima.

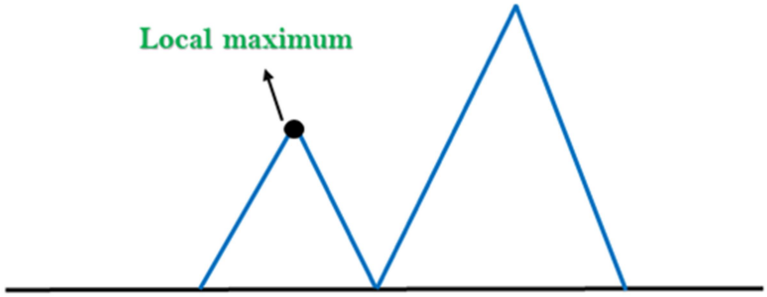
Disadvantages:

- Can get stuck in local optima and fail to find the global optimum.
- Performance depends heavily on the initial solution and the neighborhood function.

Problems in Hill Climbing Algorithm

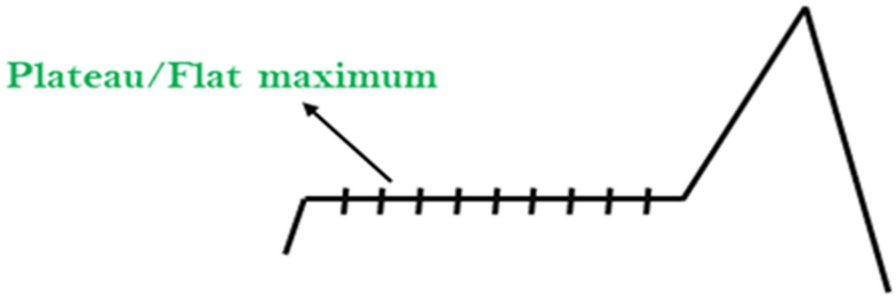
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



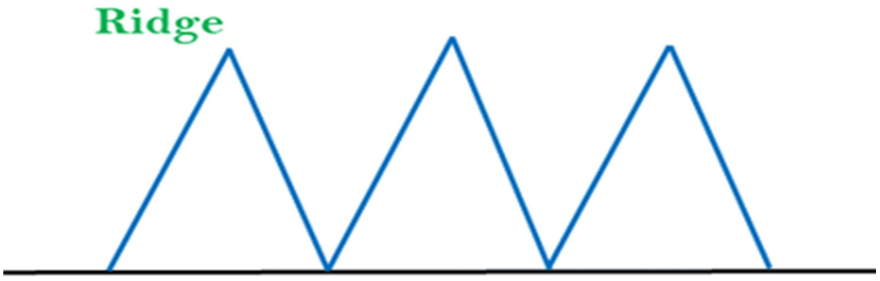
2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



3. Ridges: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.



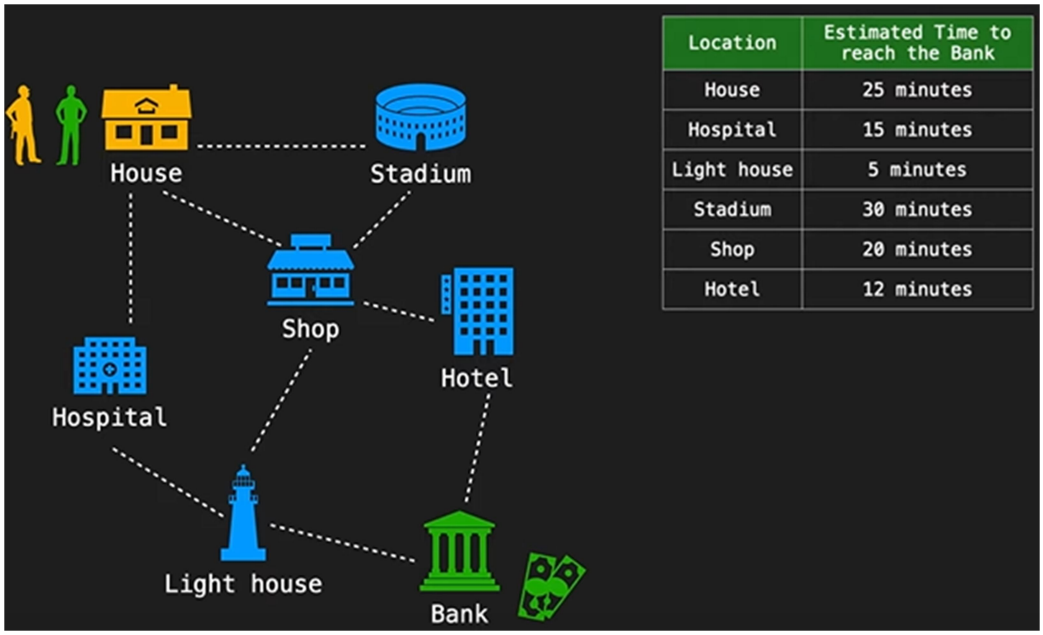
6.2.2. Greedy Search

Greedy Best-First Search attempts to find the most promising path from a given starting point to a goal. It prioritizes paths that appear to be the most promising, regardless of whether or not they are actually the shortest path. The algorithm works by evaluating the cost of each possible path and then expanding the path with the lowest cost. This process is repeated until the goal is reached.

The algorithm works by using a heuristic function $\boldsymbol{f(n) = h(n)}$ to determine which path is the most promising. The heuristic function takes into account the cost of the current path and the estimated cost of the remaining paths.

If the cost of the current path is lower than the estimated cost of the remaining paths, then the current path is chosen. This process is repeated until the goal is reached.

Example of heuristic function (estimated time to reach the bank).



Location	Estimated Time to reach the Bank
House	25 minutes
Hospital	15 minutes
Light house	5 minutes
Stadium	30 minutes
Shop	20 minutes
Hotel	12 minutes

Algorithm

1. Initialize a tree with the root node being the start node in the open list.
2. If the open list is empty, return a failure, otherwise, add the current node to the closed list.
3. Remove the node with the lowest **h(x)** value from the open list for exploration.

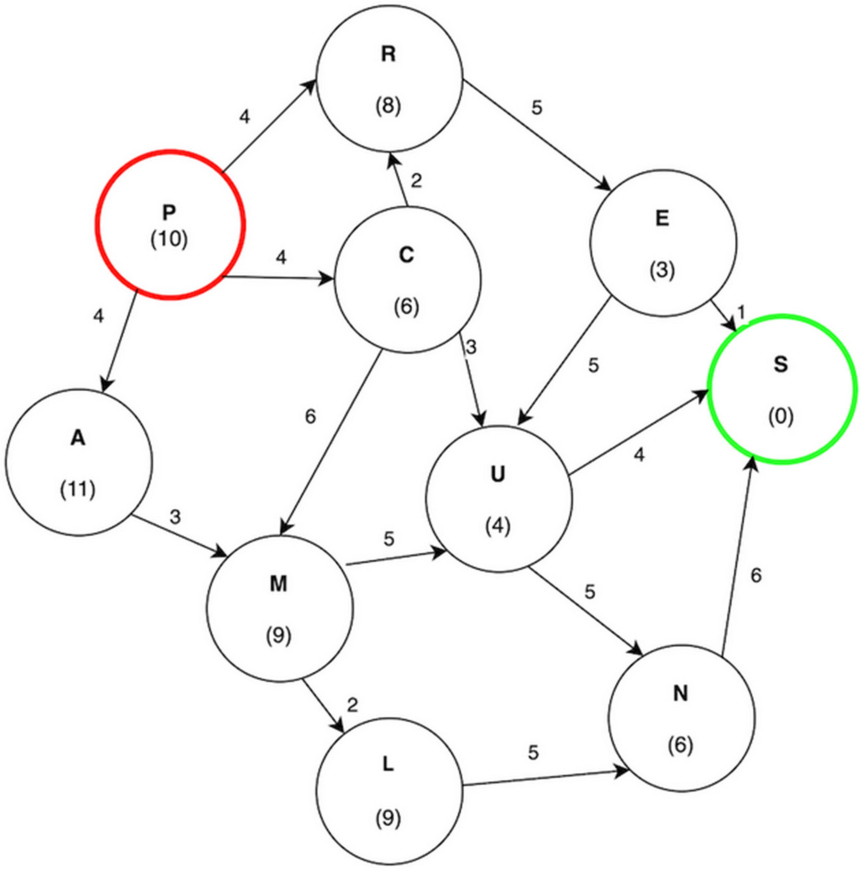
4. If a child node is the target, return a success. Otherwise, if the node has not been in either the open or closed list, add it to the open list for exploration.

Pseudocode

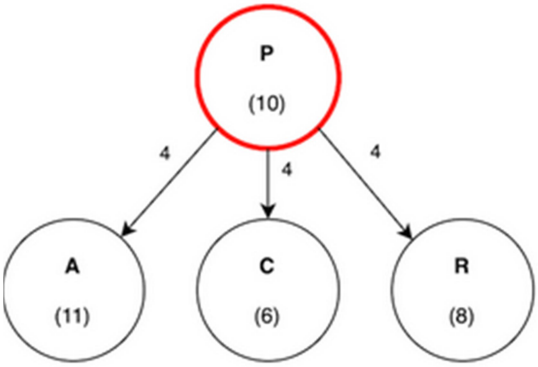
```
function GreedyBestFirstSearch(start, goal, h):
   open_set = PriorityQueue()
   open_set.add(start, h(start))
   came_from = {}

   while not open_set.is_empty():
      current = open_set.pop()

      if current == goal:
         return reconstruct_path(came_from, current)

      for neighbor in current.neighbors():
         if neighbor not in came_from:
            came_from[neighbor] = current
            open_set.add(neighbor, h(neighbor))
   return failure

function reconstruct_path(came_from, current):
   total_path = [current]
   while current in came_from:
      current = came_from[current]
      total_path.append(current)
   return total_path.reverse()
```

Exemple

Consider finding the path from **P** to **S** in the following graph:



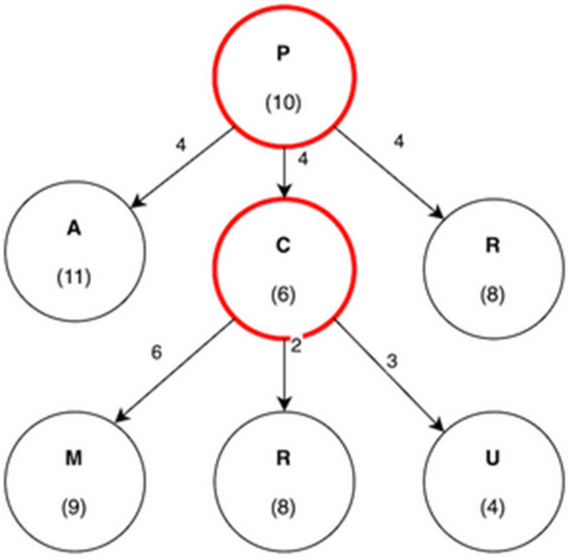
In this example, the cost is measured strictly using the heuristic value.



Node[cost]
A[11]
C[6]
R[8]

Closed List
P

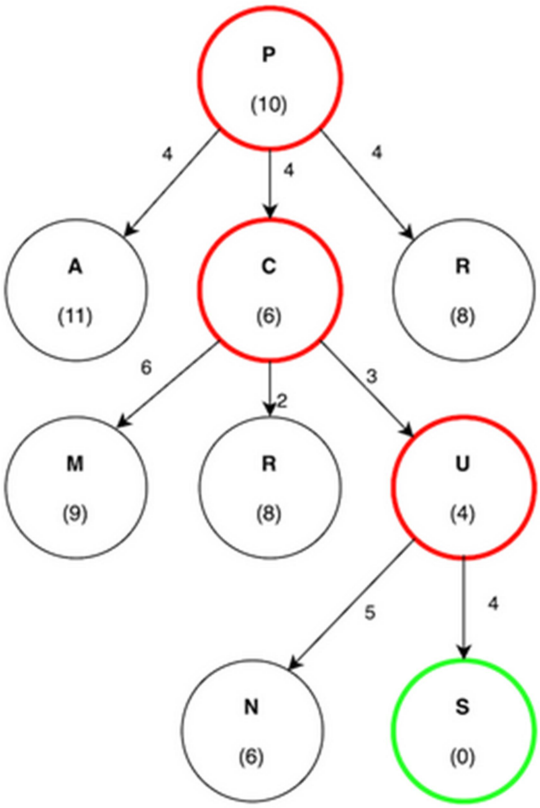
C has the lowest cost of 6. Therefore, the search will continue like so:



Node[cost]
M[9]
R[8]
U[4]

Closed List
P
C

U has the lowest cost compared to **M** and **R**, so the search will continue by exploring **U**. Finally, **S** has a heuristic value of 0 since that is the target node:



Node[cost]
N[6]
S[0]

Closed List
P
C
U

The total cost for the path (**P -> C -> U -> S**) evaluates to 11. The potential problem with a greedy best-first search is revealed by the path (**P -> R -> E -> S**) having a cost of 10, which is lower than (**P -> C -> U -> S**). Greedy best-first search ignored this path because it does not consider the edge weights.

Advantages of Greedy Best-First Search

- **Simple and Easy to Implement:** Greedy Best-First Search is a relatively straightforward algorithm, making it easy to implement.

- **Fast and Efficient:** Greedy Best-First Search is a very fast algorithm, making it ideal for applications where speed is essential.
- **Low Memory Requirements:** Greedy Best-First Search requires only a small amount of memory, making it suitable for applications with limited memory.
- **Flexible:** Greedy Best-First Search can be adapted to different types of problems and can be easily extended to more complex problems.
- **Efficiency:** If the heuristic function used in Greedy Best-First Search is good to estimate, how close a node is to the solution, this algorithm can be a very efficient and find a solution quickly, even in large search spaces.

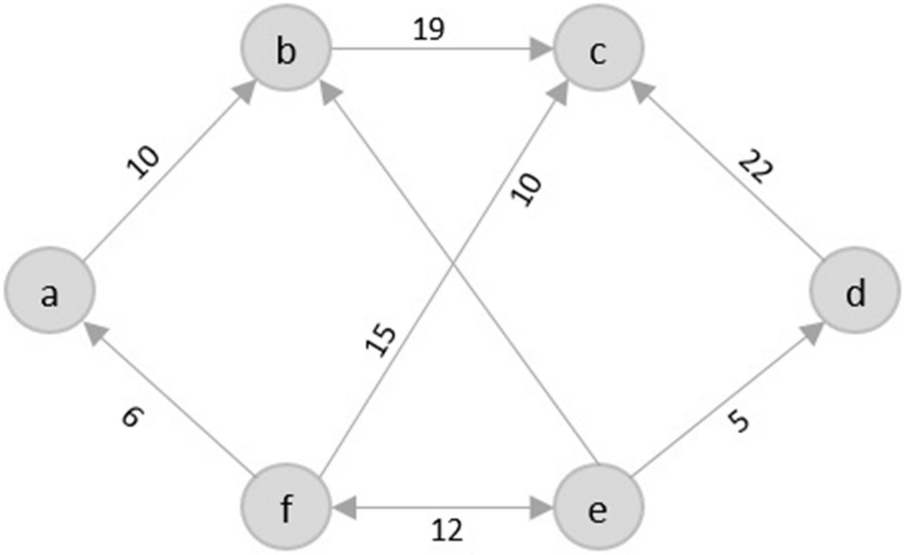
Disadvantages of Greedy Best-First Search

- **Inaccurate Results:** Greedy Best-First Search is not always guaranteed to find the optimal solution, as it is only concerned with finding the most promising path.
- **Local Optima:** Greedy Best-First Search can get stuck in local optima, meaning that the path chosen may not be the best possible path.
- **Heuristic Function:** Greedy Best-First Search requires a heuristic function in order to work, which adds complexity to the algorithm**.**
- **Lack of Completeness:** Greedy Best-First Search is not a complete algorithm, meaning it may not always find a solution if one exists. This can happen if the algorithm gets stuck in a cycle or if the search space is a too much complex.

Exercises

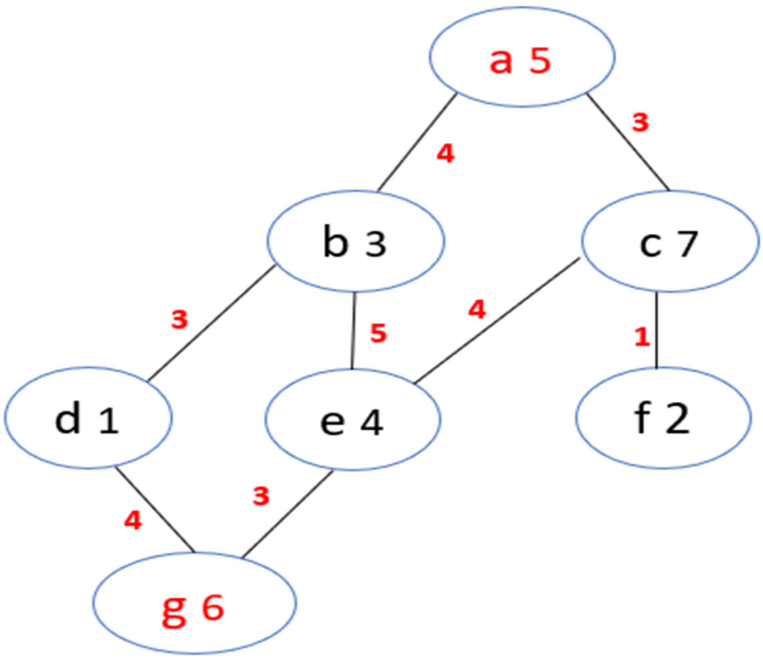
Exercise 1

Apply Hill Climbing Search to find the highest-value path in the following graph, where each node has an associated heuristic value (estimating how close it is to the goal d).



Exercise 2

Compare Hill Climbing (HC) and Greedy Best-First Search (GBFS) on the graph bellow to observe their differences in path selection and optimality.



Chapter 4

Game Theory

1. Introduction

Introduced by Von Neumann and Morgenstern in 1940, game theory is a tool used by both economists and strategists to predict and analyze the behaviors and choices of rational individuals seeking to maximize their gains and minimize their losses in strategic interaction situations, without knowing the choices of others. Indeed, the choices of some determine and influence the gains of others in strategic interaction situations, and game theory will allow for predicting the strategy to be adopted by the agent.

In this chapter we cover **competitive** environments, in which the agents' goals are in conflict, giving rise GAME to **adversarial search** problems—often known as **games**.

Mathematical **game theory**, a branch of economics, views any multiagent environment

as a game, provided that the impact of each agent on the others is "significant," regardless of whether the agents are cooperative or competitive.

Games are interesting because they are too hard to solve. For example, chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about 35^{100} or 10^{154} nodes (although the search graph has "only" about 10^{40} distinct nodes). Games, like the real world, therefore require the ability to make some decision even when calculating the optimal decision is infeasible.

2. What Is Game Theory?

Game theory is the study of how and why individuals and entities (called players) make decisions about their situations. It is a theoretical framework for conceiving social scenarios among competing players.

The goal of game theory is to explain the strategic actions of two or more players in a given situation with set rules and outcomes. Any time a situation with two or more players involves known payouts or quantifiable consequences, we can use game theory to help determine the most likely outcomes.

The key to game theory is that one player's payoff is contingent on the strategy implemented by the other player.

Game theory is the science of strategy, or at least of the optimal decision-making of independent and competing actors in a strategic setting.

Game theory has several main objectives, which can be summarized as follows:

1. **Modeling Strategic Interactions**: Game theory aims to model situations where the actions of one individual or entity affect the outcomes of others. This is often referred to as strategic interaction.
2. **Predicting Behavior**: By analyzing the incentives and possible actions of players, game theory seeks to predict how individuals or entities will behave in strategic situations.
3. **Identifying Optimal Strategies**: One of the primary goals is to determine the best strategies for players to maximize their payoffs or minimize their losses, given the strategies of other players.
4. **Understanding Equilibria**: Game theory aims to identify and understand equilibrium points, such as the Nash equilibrium, where no player has an incentive to deviate from their chosen strategy.
5. **Analyzing Conflict and Cooperation**: It provides tools to analyze situations of conflict and cooperation, helping to understand when and why players might choose to cooperate or compete.
6. **Designing Mechanisms**: Game theory is used to design mechanisms and institutions that can achieve desired outcomes, such as auctions, voting systems, and market structures.
7. **Applied to Various Fields**: The theory is applied across various disciplines, including economics, political science, biology, computer science, and military strategy, to provide insights and solutions to real-world problems.
8. **Normative and Descriptive Analysis**: Game theory can be used both normatively (to prescribe what players should do) and descriptively (to describe what players actually do) in strategic situations.

By achieving these objectives, game theory offers a powerful framework for understanding and influencing strategic decision-making in a wide range of contexts.

3. Formulation of game theory

Game theory is a mathematical framework used to model and analyze strategic interactions between rational decision-makers. It can be formally defined as a kind of search problem with the following elements:

1. **Players:** The individuals or entities making decisions in the game. Typically denoted by P1,P2,…,Pn.

2. **Strategies:** The set of actions or choices available to each player. For player i, the set of strategies is denoted by Si.

3. **Payoffs:** The payoff matrix provides a clear visualization of possible outcomes or rewards that players receive based on the combination of strategies chosen

by all players. Each cell in the matrix contains the payoffs for each player, typically in the form of a pair of numbers.

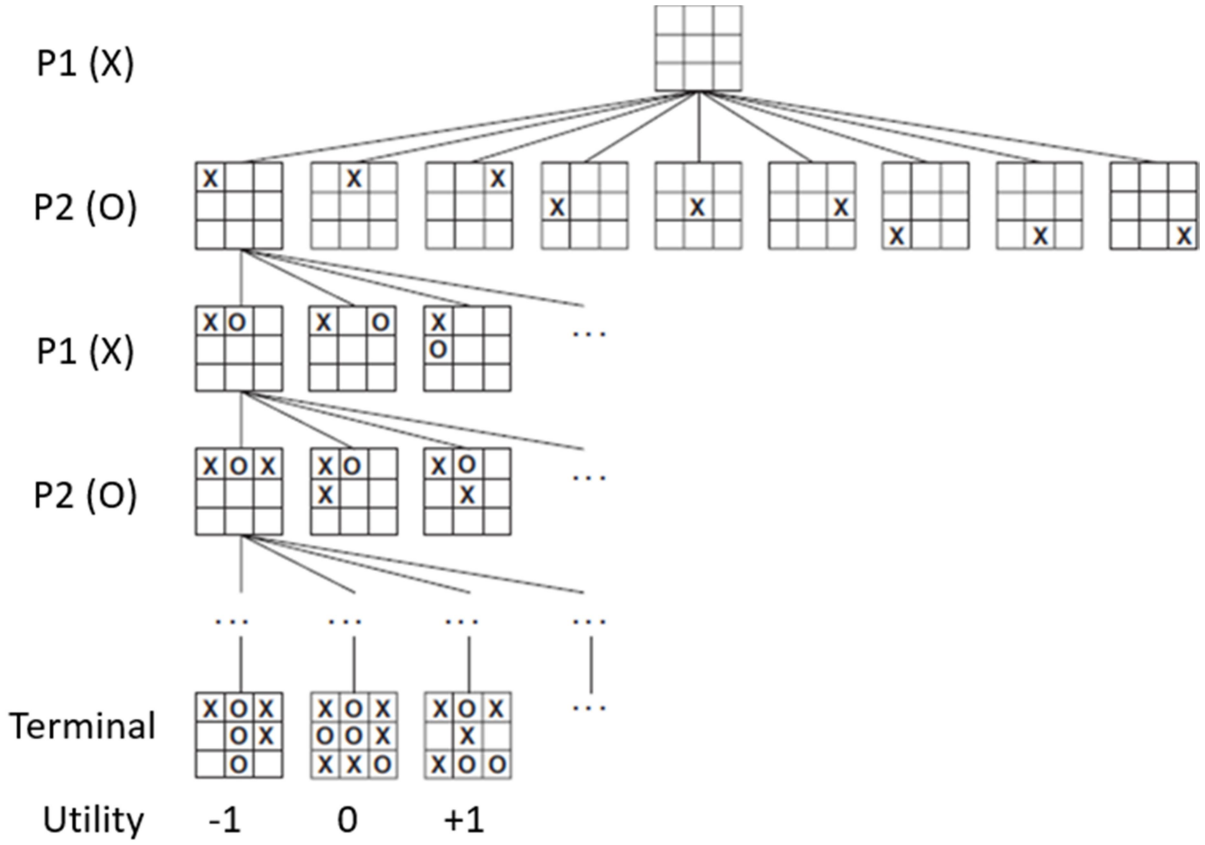
Structure

- **Rows**: Represent the strategies of Player 1.
- **Columns**: Represent the strategies of Player 2.
- **Cells**: Contain the payoffs for each combination of strategies, usually in the form (Player 1's payoff, Player 2's payoff).

4. Game Representation

- **Normal Form (Strategic Form)**: A matrix or table that lists the strategies of each player and the corresponding payoffs for each combination of strategies.
- **Extensive Form**: A tree diagram that represents the sequence of moves and the information available to players at each decision point.

Figure below shows part of the game tree for tic-tac-toe (noughts and crosses). Players take turns, with Player 1 placing an X and Player 2 placing an O. From the initial state, Player 1 has nine possible moves.



Example: The Prisoner's Dilemma

The prisoner's dilemma is the most well-known example of game theory. Consider the example of two criminals arrested for a crime. Prosecutors have no hard evidence

to convict them. However, to gain a confession, officials remove the prisoners from their solitary cells and question each one in separate chambers. Neither prisoner has the means to communicate with the other. Officials present four deals, often displayed as a 2 x 2 box.

- If both confess, they will each receive a three-year prison sentence.
- If Prisoner 1 confesses, but Prisoner 2 does not, Prisoner 1 will be free and Prisoner 2 will get five years.
- If Prisoner 2 confesses, but Prisoner 1 does not, Prisoner 1 will get five years, and Prisoner 2 will be free.
- If neither confesses, each will serve one year in prison.

	Cooperate (Player 2)	**Defect (Player 2)**
Cooperate (Player 1)	(3, 3)	(0, 5)
Defect (Player 1)	(5, 0)	(1, 1)

The payoff matrix is used to analyze optimal strategies and determine **Nash equilibrium**.

The **Nash equilibrium** is for both players to defect, as this is the best strategy for each player regardless of the other's choice.

5. Types of Game Theory

Although there are many types of game theory, such as symmetric/asymmetric, simultaneous/sequential, and so on, cooperative and non-cooperative game theories are the most common.

5.1. Cooperative vs. Non-Cooperative Games

Cooperative game theory deals with how coalitions, or cooperative groups, interact when only the payoffs are known. It is a game between coalitions of players rather than between individuals, and it questions how groups form and how they allocate the payoff among players.

Non-cooperative game theory deals with how rational economic agents deal with each other to achieve their own goals. The most common non-cooperative game is the strategic game, in which only the available strategies and the outcomes that result from a combination of choices are listed. A simplistic example of a real-world non-cooperative game is rock-paper-scissors.

5.2. Complete Information vs. Incomplete Information

A game of complete information is a game where you know the structure of the game, that is, the identity of all players, the number of players, the order of decisions, the possible actions or strategies, and the payoffs; typically, the game of chess is a game of complete information.

If one of the indicated elements is not present, the game is of incomplete information. Poker is an example of a game of incomplete information.

5.3. Symmetric vs.Asymmetric

A game is said to be symmetric if the roles of the players are interchangeable, meaning that the payoffs and available strategies are the same for all players. In a symmetric game, no player has an intrinsic advantage or disadvantage over the others.

The Prisoner's Dilemma is an example of a symmetric game. Both players have the same choices (cooperate or defect) and the same payoffs associated with these choices.

A game is said to be asymmetric if the roles of the players are not interchangeable, meaning that the payoffs and available strategies differ among the players. In an asymmetric game, some players may have specific advantages or disadvantages.

A negotiation game between an employer and an employee can be asymmetric. The employer and the employee have different positions and different payoffs based on their actions.

5.4. Zero-Sum vs. Non-Zero-Sum Games

When multiple parties are in direct competition for the same result, it is often referred to as a zero-sum game.

This means that for every winner, there is a loser. Alternatively, it means that the collective net benefit received is equal to the collective net benefit lost. Lots of sporting events are a zero-sum game as one team wins and another team loses.

A non-zero-sum game is one in which all participants can win or lose at the same time. Consider business partnerships that are mutually beneficial and foster value for both entities. Instead of competing and attempting to win at the expense of the other, both parties benefit.

5.5. Simultaneous Move vs. Sequential Move Games

Simultaneous move situations, which occur frequently in life, mean each participant must continually make decisions at the same time that their opponent is making decisions. As companies devise their marketing, product development, and operational plans, competing companies are doing the same thing at the same time.

In some cases, there is an intentional staggering of decision-making steps, enabling one party to see the other party's moves before making their own. This is usually present in negotiations; one party lists their demands, then the other party has a designated amount of time to respond and list their own.

5.6. One Shot vs. Repeated Games

These are games that are played only once. Players make their decisions simultaneously and independently, without knowledge of the other players' choices. This is often the case with equity traders, who must wisely choose their entry point and exit point, as their decision may not easily be undone or retried.

On the other hand, repeated games are games that are played multiple times by the same players. It could be a finite number of times (known or unknown) or an infinite number of times.
For example, consider rival companies trying to price their goods. Whenever one makes a price adjustment, so may the other. This circular competition repeats itself across product cycles or sale seasonality.

6. Search and decision-making algorithms

In the context of game theory, search and decision-making algorithms are crucial for modeling and solving strategic situations where players interact competitively or cooperatively.
The **Minimax algorithm** and its optimizations, such as **Alpha-Beta Pruning**, are fundamental for perfect information games. For games with larger search spaces or imperfect information, techniques like **Monte Carlo Tree Search** and **reinforcement learning algorithms** are often used. These algorithms enable optimal decision-making in complex and dynamic environments.

6.1. The Minimax algorithm

The Minimax is a tree search algorithm used in two-player, zero-sum games with alternating friendly and enemy moves. It assumes the existence of an evaluation function called at a given depth and common to both players. The friendly player seeks to maximize the evaluation, and the enemy player seeks to minimize it. At a friendly node (respectively enemy node), the minimax value is the maximum (respectively minimum) of the minimax values of the child nodes.

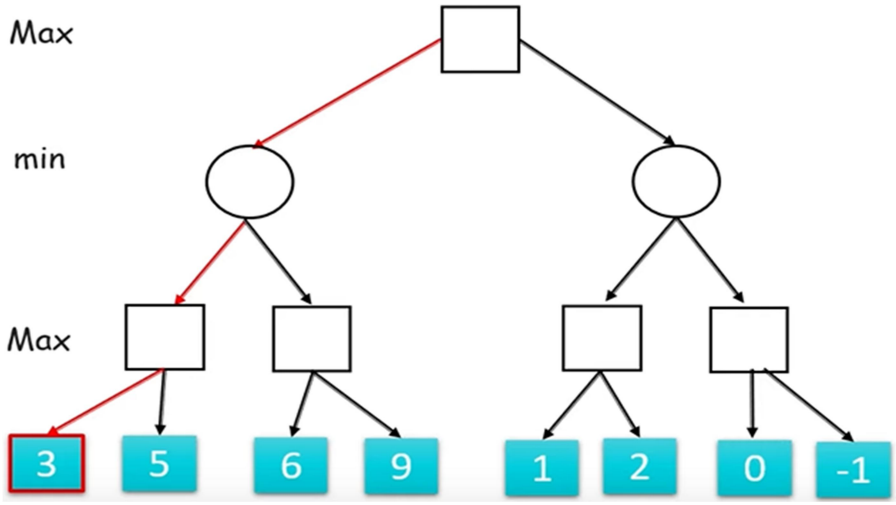
1. **Maximizing Player (Maximizer)**: The player who aims to maximize their gain.

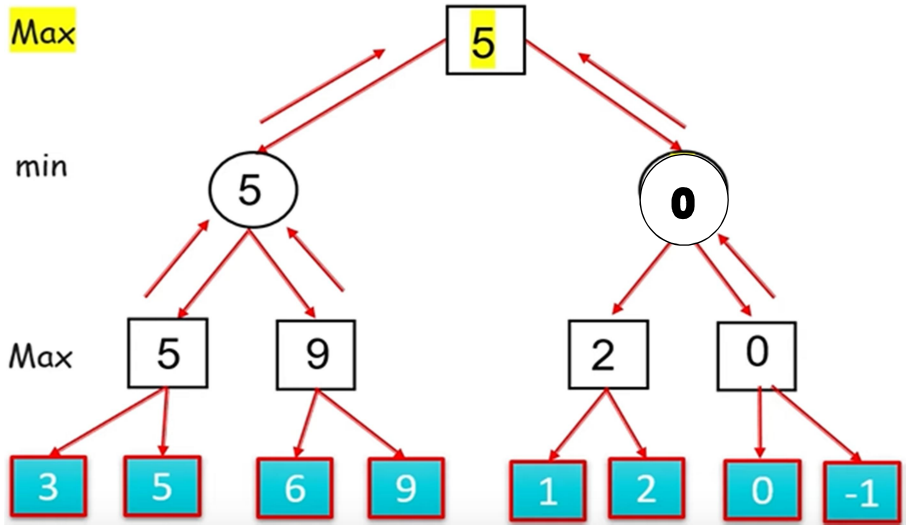
2. **Minimizing Player (Minimizer)**: The player who aims to minimize the gain of the maximizing player.
3. **Game Tree**: A tree structure representing all possible moves and countermoves in the game.
4. **Terminal Nodes**: Nodes in the game tree that represent the end of the game, with associated payoffs.

How the Minimax Algorithm Works

1. **Generate the Game Tree**: Create a tree of all possible moves and countermoves.
2. **Evaluate Terminal Nodes**: Assign a value to each terminal node based on the payoff for the maximizing player.
3. **Propagate Values Up the Tree**:
    - For the maximizing player's turn, choose the maximum value among the child nodes.
    - For the minimizing player's turn, choose the minimum value among the child nodes.
4. **Determine the Optimal Move**: The value at the root of the tree represents the best possible outcome for the maximizing player, assuming optimal play by both players.

Example





Pseudocode

```
def minimax(node, depth, maximizingPlayer):
    if node is a terminal node or depth == 0:
        return evaluate(node)

    if maximizingPlayer:
        maxEval = -infinity
        for child in node.children:
            eval = minimax(child, depth - 1, False)
            maxEval = max(maxEval, eval)
        return maxEval
    else:
        minEval = +infinity
        for child in node.children:
            eval = minimax(child, depth - 1, True)
            minEval = min(minEval, eval)
        return minEval
```

Limitations

- **Computational Complexity**: The Minimax algorithm can be computationally expensive for large game trees.
- **Perfect Information**: It assumes perfect information, which may not be the case in many real-world scenarios.

Optimizations

- **Alpha-Beta Pruning**: An optimization technique that reduces the number of nodes evaluated by the Minimax algorithm, making it more efficient.

6.2. Alpha–Beta pruning

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.
For tic-tac-toe the game tree is relatively small—fewer than 9! = 362 880 terminal nodes. But for chess there are over 10^{40} nodes, so the game tree is best thought of as a theoretical construct that we cannot realize in the physical world.

Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half. The trick is that it is possible to compute the correct minimax decision without looking at every node in the game tree. That is, we can borrow the idea of **pruning** to eliminate large parts of the tree from consideration. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Alpha-Beta Pruning is an optimization technique for the Minimax algorithm that reduces the number of nodes evaluated in the game tree. It eliminates branches that cannot influence the final decision, thereby improving the efficiency of the search process. This involves two threshold parameters alpha and beta for future expansion, so it is called **alpha-beta pruning.**
Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prunes the tree leaves but also entire sub-tree.

The two-parameter can be defined as:

1. **Alpha:** Represents the minimum score that the maximizing player is assured of.
2. **Beta:** Represents the maximum score that the minimizing player is assured of.

The main condition which required for alpha-beta pruning is: **α>=β**

How Alpha-Beta Pruning Works

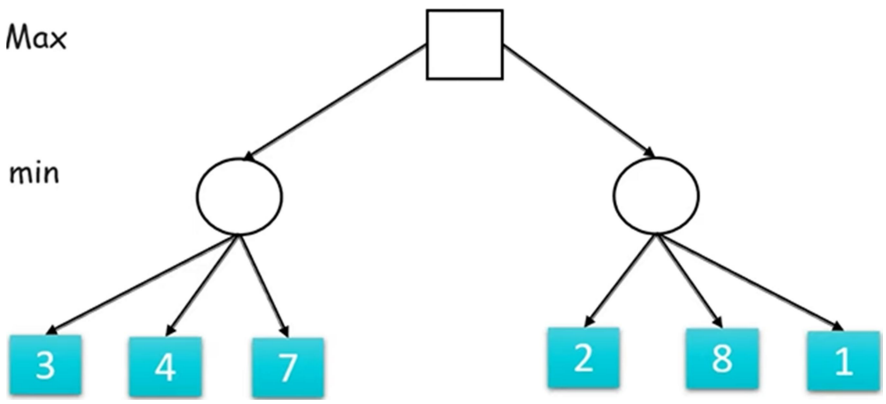
1. **Initialize Alpha and Beta**: Start with alpha as negative infinity (−∞) and beta as positive infinity (+∞).

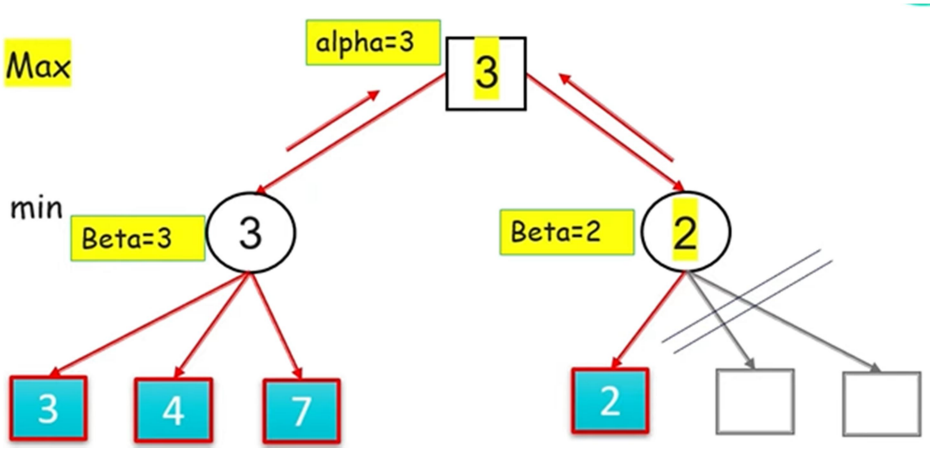
2. **Propagate Alpha and Beta**:
   - At a maximizing node, update alpha to the maximum of alpha and the current node's value.
   - At a minimizing node, update beta to the minimum of beta and the current node's value.
3. **Prune Branches**:

- If at any point, beta ≤ alpha, prune the remaining branches because they cannot affect the final decision.

Pseudo-code for Alpha-Beta Pruning

```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
  maxEva= -infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, False)
 maxEva= max(maxEva, eva)
 alpha= max(alpha, maxEva)
  if beta<=alpha
 break
 return maxEva

else                    // for Minimizer player
  minEva= +infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, true)
  minEva= min(minEva, eva)
  beta= min(beta, eva)
   if beta<=alpha
 break
 return minEva
```





Advantages

- **Efficiency**: Significantly reduces the number of nodes evaluated compared to the standard Minimax algorithm.
- **Optimal Decisions**: Maintains the optimality of the Minimax algorithm while improving performance.

Limitations

- **Complexity**: Still computationally intensive for very large game trees.
- **Perfect Information**: Assumes perfect information, which may not be the case in many real-world scenarios.

7. Limitations of Game Theory

While game theory provides a valuable framework for analyzing strategic interactions, it is important to recognize its limitations and apply it judiciously. Combining game theory with other analytical tools and considering the specific context and constraints of the situation can help to mitigate some of these limitations and provide more robust insights.

1. **Assumption of Rationality**
   Game theory often assumes that all players are perfectly rational and will always make decisions that maximize their utility. In reality, human behavior is often influenced by emotions, biases, and limited information, leading to decisions that may not be strictly rational.

2. **Perfect Information**
   Many game theory models assume that players have perfect information about the game's structure, the actions of other players, and the possible outcomes. In real-world scenarios, information is often incomplete or uncertain, making it difficult to apply these models directly.

3. **Complexity**
   Game theory can become extremely complex, especially in games with many players, multiple rounds, or a large number of possible actions. Analyzing such games can be computationally intensive and may not be feasible in practical applications.

4. **Static vs. Dynamic Environments**
   Game theory often focuses on static or one-shot games, where players make a single decision. However, many real-world situations are dynamic, with players making multiple decisions over time. This dynamic nature can introduce additional complexities that are not fully captured by static models.

5. **Bounded Rationality**
   Players in real-world situations often have limited cognitive abilities and resources, leading to bounded rationality. This means they may not be able to consider all possible outcomes and may make decisions based on heuristics or rules of thumb rather than optimal strategies.

6. **Simplified Models**
   Game theory models often simplify real-world situations to make them controllable. This simplification can lead to models that do not fully capture the nuances and complexities of real-world interactions. For example, models may assume that players have fixed preferences or that the game has a fixed number of rounds.

8. **Equilibrium Assumptions**
   Many game theory models focus on finding equilibrium points, such as Nash equilibria, where no player can improve their outcome by unilaterally changing their strategy. However, in real-world situations, players may not always reach or maintain equilibrium, especially if the game is repeated or if players can learn and adapt over time.

7. **Ethical and Social Considerations**
   Game theory often focuses on individual utility maximization and may not fully account for ethical, social, or cooperative considerations. In real-world situations, players may be influenced by factors such as justice, mutuality, and social norms, which are not always captured by game theory models.

8. **Limited Predictive Power**
   Game theory models can provide insights into possible outcomes and strategies, but they often have limited predictive power. Real-world outcomes

are influenced by a wide range of factors, many of which may not be captured by the model.

9. **Assumption of Common Knowledge**
   Many game theory models assume that players have common knowledge of the game's structure, the rules, and the possible outcomes. In real-world situations, players may have different levels of knowledge or may interpret the rules and outcomes differently.

Exercises

Exercise 1: Quiz on Game Theory

1. **Game theory studies how people make decisions in competitive situations.**
   a. ( ) True
   b. ( ) False

2. **In the Prisoner's Dilemma, both players cooperating is the best outcome for society.**
   a. ( ) True
   b. ( ) False

3. **A Nash Equilibrium is when all players use the best strategy possible, no matter what others do.**
   a. ( ) True
   b. ( ) False

4. **Which of these is an example of a zero-sum game?**
   a. ( ) Chess
   b. ( ) The Prisoner's Dilemma
   c. ( ) Both

5. **In the Prisoner's Dilemma, what happens if both players defect?**
   a. ( ) They both get the worst possible outcome.
   b. ( ) They both get a medium outcome.
   c. ( ) One wins, and the other loses.

6. **What is a "dominant strategy"?**
   a. ( ) A strategy that always works, no matter what the opponent does.
   b. ( ) A strategy that only works if the opponent makes a mistake.
   c. ( ) A strategy that changes based on the opponent's moves.

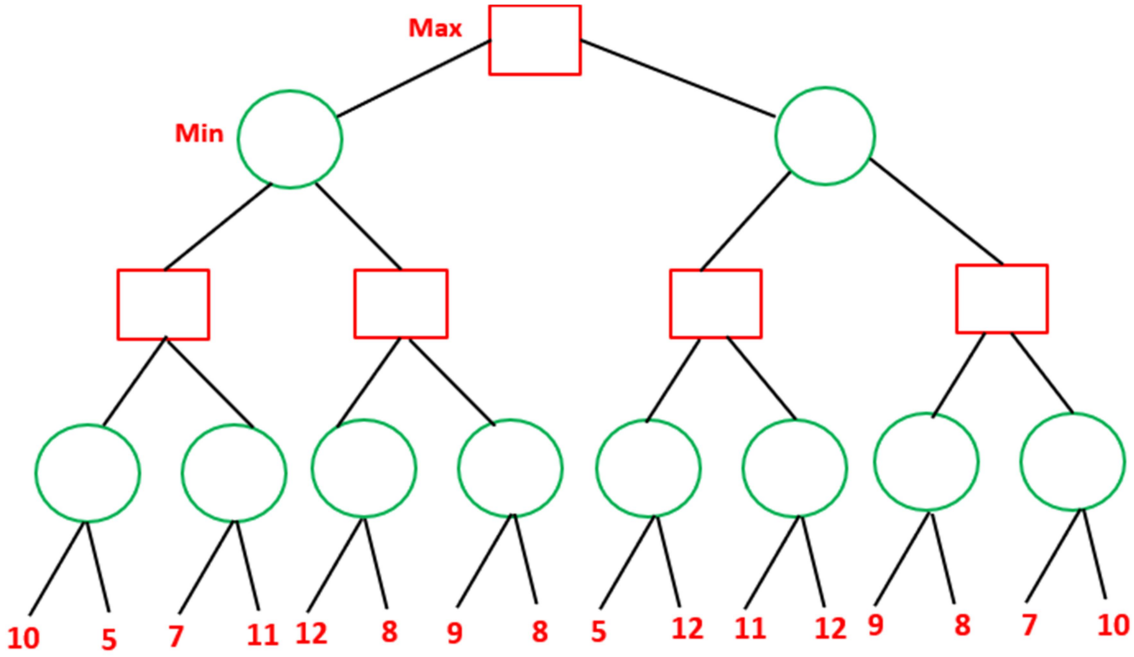
7. **What is a "Nash Equilibrium"?**
   a) A strategy where no player can benefit by unilaterally changing their strategy.
   b) A strategy that maximizes the total payoff of all players.
   c) A strategy where players take turns optimizing their moves.

8. **In the Prisoner's Dilemma, why do players often defect (confess)?**
   a) Because cooperation is irrational in a one-shot game.
   b) Because the police force them to confess.
   c) Because it maximizes social welfare.

9. **What is a "dominant strategy"?**
   a) A strategy that always gives a higher payoff, regardless of opponents' choices.
   b) A strategy that only works against weak opponents.
   c) A strategy that requires cooperation.

Exercise 2

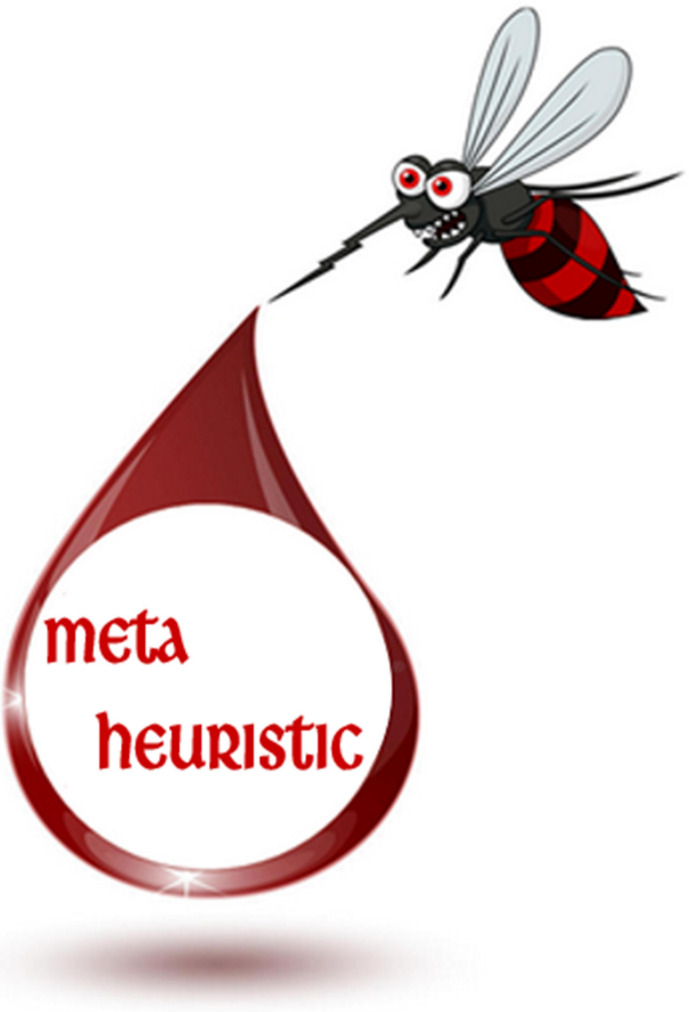
Consider the following game tree where the goal is to maximize the score.



1. Use the minimax algorithm to determine the best move for the maximizing player
2. Use the alpha-beta pruning algorithm to determine the best move for the maximizing player

Chapter 5

Metaheuristic



1. Introduction

Heuristic and metaheuristic methods are both used to solve optimization problems, particularly when finding an exact solution is computationally expensive or impractical. While they share some similarities, they differ mainly in their generality, approach, and adaptability to different types of problems.

A **metaheuristic search** refers to a high-level, generalized method or strategy designed to find approximate solutions to complex optimization problems. Unlike problem-specific algorithms (like exact optimization methods), metaheuristics are designed to be adaptable and applicable to a wide range of problems, often without requiring detailed knowledge of the problem's specific structure. The key advantage of metaheuristics is their ability to explore large solution spaces and avoid getting stuck in local optima, potentially finding near-optimal solutions in reasonable time.

Metaheuristics provide "acceptable" solutions in a reasonable time for solving hard and complex problems in science and engineering.
Unlike exact optimization algorithms, metaheuristics do not guarantee the optimality of the obtained solutions. Instead of approximation algorithms, metaheuristics do not define how close are the obtained solutions from the optimal ones.

The word heuristic has its origin in the old Greek word heuriskein, which means the art of discovering new strategies (rules) to solve problems. The suffix meta, also a Greek word, means "upper-level methodology." The term metaheuristic was introduced by F. Glover. Metaheuristic search methods can be defined as upper-level general methodologies (templates) that can be used as guiding strategies in designing underlying heuristics to solve specific optimization problems.

2. Metaheuristic Search Methods

Heuristic search methods and metaheuristic search methods are both valuable tools for solving optimization problems, but they differ in their approach and scope. Heuristic methods are problem-specific and aim to find acceptable solutions quickly, while metaheuristics offer a general and flexible approach to exploring the search space and finding near-optimal solutions.

2.1. Definition

Metaheuristic search methods are high-level, problem-independent strategies designed to guide the search process for finding near-optimal solutions in complex

search spaces. They provide a general framework that can be applied to a wide range of optimization problems.
Unlike heuristics, which are problem-specific and often provide quick but not necessarily optimal solutions, metaheuristics offer a more general approach that can be applied to a wide range of problems.

Generality
Metaheuristics are not specific to any particular problem but can be adapted to various types of problems. They offer a flexible approach to exploring the search space and avoiding local optima.

Objective
The objective of metaheuristic search methods is to find near-optimal solutions by effectively exploring the search space. They aim to balance exploration (searching new areas) and exploitation (refining known good solutions).

Approach
Metaheuristics use a combination of exploration and exploitation strategies to navigate the search space. They often incorporate randomness and stochastic elements to escape local optima and explore different regions of the solution space.

2.2. Characteristics of Metaheuristics

1. **Problem-Independent**: Metaheuristics are not tailored to a specific problem but can be adapted to various optimization problems. This makes them versatile and applicable to a broad spectrum of challenges.

2. **Global Search**: They aim to explore the entire solution space to avoid getting trapped in local optima. This global search capability is crucial for finding near-optimal solutions in complex landscapes.

3. **Iterative Improvement**: Metaheuristics typically involve iterative processes that improve the solution over time. This iterative nature allows for continuous refinement and exploration of the solution space.

4. **Stochastic Nature**: Many metaheuristics incorporate randomness to explore the solution space more effectively. This stochastic element helps in escaping local optima and discovering new regions of the solution space.

5. **Balance Between Exploration and Exploitation**: Metaheuristics balance between exploring new areas of the solution space (exploration) and refining known good solutions (exploitation). This balance is essential for finding high-quality solutions efficiently.

2.3. Advantages of Metaheuristics

1. **Versatility**: Metaheuristics can be applied to a wide range of optimization problems, making them a valuable tool in various fields such as engineering, computer science, operations research, and finance.

2. **Effectiveness in Complex Problems**: They are particularly effective in solving complex, high-dimensional problems where traditional optimization methods may fail. Metaheuristics can handle non-linear, non-convex, and discontinuous solution spaces.

3. **Flexibility**: Metaheuristics can be combined with other optimization techniques to enhance performance. This flexibility allows for the development of hybrid methods that leverage the strengths of multiple approaches.

4. **Robustness**: Metaheuristics are robust to changes in the problem formulation and can adapt to different types of constraints and objectives. This robustness makes them suitable for real-world applications where the problem may evolve over time.

2.4. Limitations of Metaheuristics

1. **Computational Complexity**: Metaheuristics can be computationally intensive, especially for large-scale problems. The iterative nature and the need to explore a vast solution space can require significant computational resources.

2. **Parameter Sensitivity**: The performance of metaheuristics can be sensitive to the choice of parameters and initial conditions. Fine-tuning these parameters can be challenging and may require extensive experimentation.

3. **Convergence Issues**: Metaheuristics may require a large number of iterations to converge to a good solution. Premature convergence to local optima can be a problem, especially if the balance between exploration and exploitation is not well-managed.

4. **Limited Theoretical Guarantees**: Unlike some traditional optimization methods, metaheuristics often lack theoretical guarantees of convergence to the global optimum. This can make it difficult to assess the quality of the solutions obtained.

2.5. Example of Metaheuristics algorithms

- **Genetic Algorithms (GA)**: Use techniques such as selection, crossover, and mutation to evolve a population of solutions.
- **Simulated Annealing (SA)**: Allows for occasional moves to worse solutions to escape local optima, with a decreasing probability over time.
- **Particle Swarm Optimization (PSO)**: Uses a population of particles that move through the solution space, adjusting their positions based on their own best-known positions and the best-known positions of their neighbors.
- **Ant Colony Optimization (ACO)**: Inspired by the pheromone trail laying and following behavior of real ants, used to find optimal paths in graphs.
- **Tabu Search (TS)**: Uses memory structures to avoid revisiting previously explored solutions and to escape local optima.

2.6. Applications of Metaheuristics

Metaheuristics are widely used in various fields to solve complex optimization problems.

Engineering:

- Design Optimization: Optimizing the design parameters of mechanical, electrical, and civil engineering systems to improve performance, reduce costs, and enhance reliability.
- Scheduling and Resource Allocation: Optimizing the scheduling of tasks, allocation of resources, and management of projects to maximize efficiency and minimize costs.

Computer Science:

- Machine Learning: Optimizing the parameters of machine learning models to improve accuracy and generalization.
- Data Mining: Finding patterns and structures in large datasets to extract valuable insights.
- Network Optimization: Designing and optimizing communication networks, routing protocols, and data transmission paths.

Operations Research:

- Supply Chain Management: Optimizing the flow of goods, information, and finances from the point of origin to the point of consumption to maximize efficiency and minimize costs.
- Logistics: Planning and optimizing the transportation and storage of goods to ensure timely and cost-effective delivery.
- Inventory Control: Managing the stock levels of products to meet demand while minimizing holding and shortage costs.

Finance:

- Portfolio Optimization: Selecting and allocating investments to maximize returns while minimizing risk.
- Risk Management: Identifying and mitigating financial risks to protect investments and ensure stability.
- Trading Strategies: Developing and optimizing trading algorithms to maximize profits and minimize losses in financial markets.

Biology:

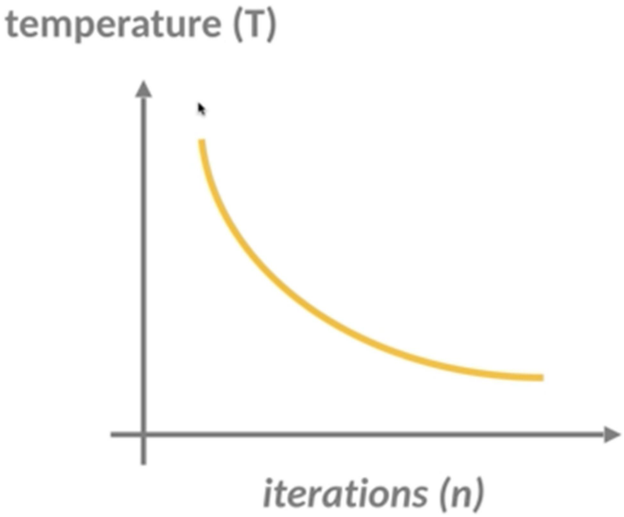
- Protein Folding: Predicting the three-dimensional structure of proteins to understand their function and behavior.
- Drug Design: Optimizing the molecular structure of drugs to enhance their efficacy and reduce side effects.
- Genetic Sequencing: Analyzing and interpreting genetic data to understand biological processes and diseases.

3. Simulated Annealing algorithm

Simulated Annealing (SA) is a probabilistic optimization algorithm inspired by the annealing process in metallurgy. It is used to find near-optimal solutions to complex optimization problems, especially those with large search spaces. The algorithm mimics the process of heating and then slowly cooling a material to decrease defects and minimize the system's energy. While it has some limitations, such as computational complexity and parameter sensitivity, its flexibility and robustness make it suitable for a wide range of optimization problems.

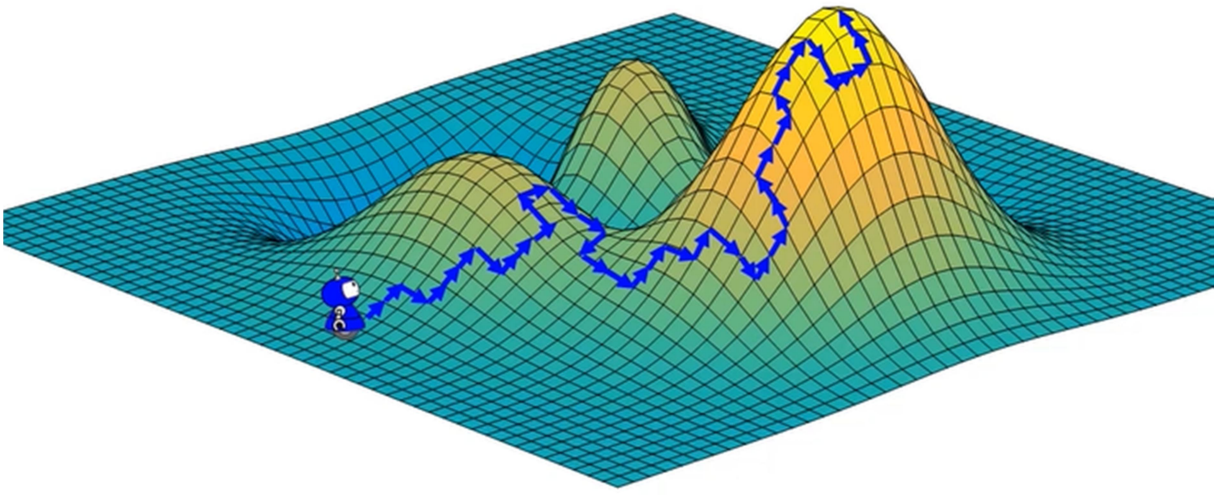
3.1. Annealing Process

In metallurgy, annealing involves heating the material to a high temperature and then slowly cooling it, which allows the atoms to rearrange themselves into a more stable crystalline structure. This reduces internal stresses and defects, resulting in a material with improved properties.



3.2. How Simulated Annealing works?

Simulated Annealing (SA) is a metaheuristic algorithm inspired by the annealing process in metallurgy. It is designed to find near-optimal solutions to complex optimization problems, especially those with large search spaces.



Simulated Annealing introduces a probability of accepting worse solutions (downhill step), especially early in the search process. This helps the algorithm escape local optima and explore the solution space more effectively.
It starts with a high "temperature" (which controls the probability of accepting worse solutions) and gradually reducing it.

Temperature Schedule
The temperature schedule defines how the temperature decreases over time. The initial temperature is set high enough to allow for significant exploration of the solution space, while the final temperature is low enough to focus on exploitation.

Acceptance Probability

The Metropolis criterion determines whether a new solution should be accepted based on the change in the objective function value and the current temperature. The criterion allows for occasional moves to worse solutions (uphill moves), which helps the algorithm escape local optima and explore the solution space more effectively.

The acceptance probability determines whether a new solution is accepted. It is given by the Metropolis criterion:

$$P(accept) = \begin{cases} 1 & if\ \Delta E \leq 0 \\ e^{\frac{-\Delta E}{T}} & if\ \Delta E > 0 \end{cases}$$

where ΔE is the change in the objective function value, and T is the current temperature.

- If the new solution improves the objective function value ($\Delta E \leq 0$), it is always accepted.
- If the new solution worsens the objective function value ($\Delta E > 0$), it is accepted with a probability that **decreases** exponentially with the magnitude of (ΔE and the current temperature T).

Cooling Schedule

The cooling schedule specifies how the temperature is reduced over time. It is crucial for the performance of the algorithm.

Common cooling schedules include exponential decay, linear decay, and logarithmic decay:

- **Exponential Decay**: $T_{new} = \alpha . T_{old}$ where $0 < \alpha < 1$.
- **Linear Decay**: $T_{new} = T_{old} - \beta . t$ where β is a constant and t is the iteration number.
- **Logarithmic Decay**: $T_{new} = \frac{T_{old}}{1+\gamma . t}$ where γ is a constant.

3.3. Algorithm Steps

1. **Initialization**
    - Start with an initial solution S_0 and an initial temperature T_0.
    - Set the cooling schedule parameters (e.g., α for exponential decay).

2. **Iterative Process**
    - While the temperature T is above a minimum threshold T_{min}:
        1. **Perturbation**: Generate a new solution S_{new} by making a small random change to the current solution S.

2. **Evaluation**: Calculate the change in the objective function value $\Delta E = f(S_{new}) - f(S)$
3. **Acceptance**: Accept the new solution S_{new} with a probability given by the Metropolis criterion.
4. **Update**: If the new solution is accepted, update the current solution $S = S_{new}$
5. **Cooling**: Reduce the temperature according to the cooling schedule.

3. **Termination**
   - The algorithm terminates when the temperature reaches the minimum threshold T_{min} or a maximum number of iterations is reached.

```
T0 = 1000  (or other large value)
T = T0
α = 0.99 (or other values  < 1)
A = initial solution

while the end condition is not met
Generate a B ∈ Neighbours(A)
    Δ = f(A) − f(B)
    if Δ > 0
        B = A
    else
        P = e^(−Δ/T)
        Generate a random r in [0,1]
        if r < P
            B = A
        end if

    end if

    T = αT
end while
```

3.4. Pseudocode

```
def simulated_annealing(initial_solution, initial_temperature, cooling_schedule, objective_function, max_iterations):

    current_solution = initial_solution
    current_temperature = initial_temperature
    best_solution = current_solution
    best_value = objective_function(current_solution)

    for iteration in range(max_iterations):
```

```
        new_solution = perturb(current_solution)
        delta_e = objective_function(new_solution) -objective_function(current_solution)

        if delta_e < 0 or random.uniform(0, 1) < math.exp(-delta_e / current_temperature):
            current_solution = new_solution

        if objective_function(current_solution) < best_value:
            best_solution = current_solution
            best_value = objective_function(current_solution)

        current_temperature = cooling_schedule(current_temperature, iteration)
    return best_solution, best_value
def perturb(solution):
    # Generate a new solution by making a small random change to the current solution
    # This function is problem-specific and needs to be implemented accordingly
    pass

def cooling_schedule(current_temperature, iteration):
    # Define the cooling schedule
    # Example: Exponential decay
    alpha = 0.9
    return alpha * current_temperature
```

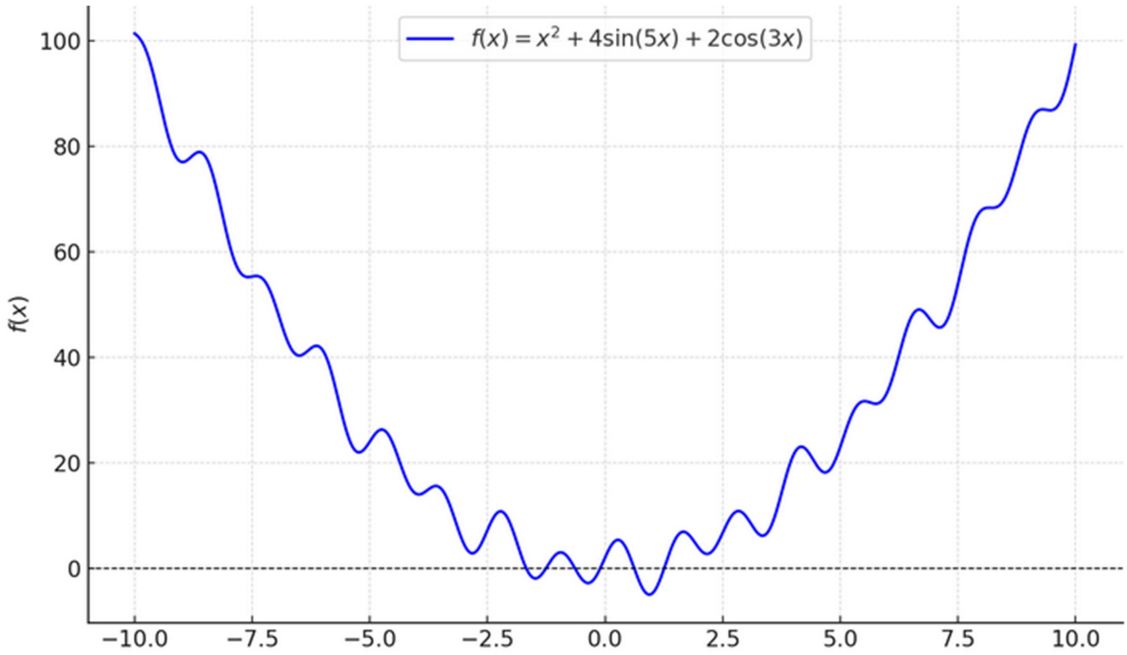
3.5. Example of Problem: Minimize a Mathematical Function

We aim to minimize the function:

$$f(x) = x^2 + 4\sin(5x) + 2\cos(3x)$$

where $x \in [-10,10]$. This function has several local minima and maxima, making it an ideal candidate for demonstrating how simulated annealing can escape local minima to find a better solution.

The graph below clearly shows a series of local minima and maxima within the range $[-10,10]$. This makes the function a good example to demonstrate how the simulated annealing algorithm can navigate such a landscape to find a global minimum.



Steps of the Simulated Annealing Algorithm

1. Initialization

- Start with a random solution, e.g., $x_{current} = 0$
- Set a high initial temperature, $T = T_{init} = 100$
- Define a cooling schedule, such as $T_{new} = 0.95 * T_{current}$
- Decide on the stopping criteria (e.g., when the temperature T drops below a small value, such as $T_{min} = 0.01$

2. Exploration

At each temperature:

1. Generate a new solution, x_{new} by making a small random change near the current solution,

$$x_{current}(\text{e.g. } x_{new} = x_{current} + \Delta, where\ \Delta\ is\ a\ small\ random\ step$$

2. Evaluate the change in the objective function:

$$\Delta E = f(x_{new}) - f(x_{current})$$

3. Acceptance Rule

- **If the new solution is better** (ΔE <0), accept it immediately:

$$x_{current} = x_{new}$$

- **If the new solution is worse** (ΔE >0), accept it with a probability:

$$P = e^{-\Delta E/T}$$

This probabilistic acceptance helps the algorithm escape local minima early in the process when T is high.

4. Cooling

- After a certain number of iterations, reduce the temperature T according to the cooling schedule.

5. Stopping

- The algorithm stops when the temperature becomes very small ($T < T_{min}$) or after a fixed number of iterations.

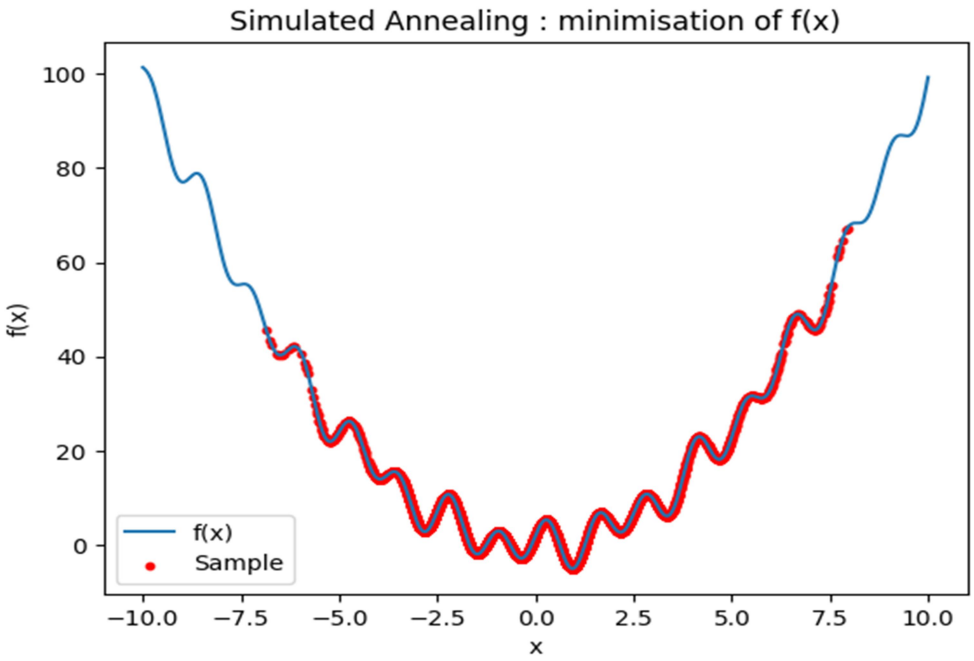
Example Walkthrough

1. **Starting Point**: Suppose $x_{current} = 0$, and the function value is $f(0) = 0^2 + 4\sin(5 * 0) + 2\cos(3 * 0) = 2$

2. **First Move**: Generate a new $x_{new} = 0.3$. Calculate $f(0.3)$ and $\Delta E = f(0.3) - f(0)$. Decide whether to accept or reject x_{new} based on the acceptance rule.

3. **Iterate**: Continue exploring and updating $x_{current}$ as the temperature T decreases, allowing the algorithm to transition from broad exploration to fine exploitation.

4. **Final Result**: After many iterations, the algorithm converges to a solution near a global minimum, such as $x \approx -2.5$



3.6. Advantages of Simulated Annealing

1. **Global Optimization**: Simulated Annealing is effective in finding near-optimal solutions in complex, high-dimensional search spaces. It can handle non-linear, non-convex, and discontinuous solution spaces.

2. **Flexibility**: The algorithm can be applied to a wide range of optimization problems, including continuous, discrete, and combinatorial problems.
3. **Robustness**: Simulated Annealing is robust to changes in the problem formulation and can adapt to different types of constraints and objectives.
4. **Escape from Local Optima**: The ability to accept worse solutions early in the process helps in escaping local optima and exploring the solution space more effectively.

3.7. Limitations of Simulated Annealing

1. **Computational Complexity**: Simulated Annealing can be computationally intensive, especially for large-scale problems. The iterative nature and the need to explore a vast solution space can require significant computational resources.
2. **Parameter Sensitivity**: The performance of Simulated Annealing is sensitive to the choice of parameters, such as the initial temperature, cooling schedule, and acceptance probability. Fine-tuning these parameters can be challenging and may require extensive experimentation.
3. **Convergence Issues**: The algorithm may require a large number of iterations to converge to a good solution. Premature convergence to local optima can be a problem if the cooling schedule is not well-managed.
4. **Limited Theoretical Guarantees**: Unlike some traditional optimization methods, Simulated Annealing often lacks theoretical guarantees of convergence to the global optimum. This can make it difficult to assess the quality of the solutions obtained.

4. Genetic algorithms

4.1. Definition

Genetic algorithms (GAs) are adaptive search algorithms based on the principles of natural selection and genetics. By simulating the process of evolution, GAs utilize mechanisms such as selection, crossover, and mutation to iteratively refine potential solutions, making them particularly effective for optimization tasks that traditional methods struggle to address.

4.2. Terminologies of Evolutionary Computation

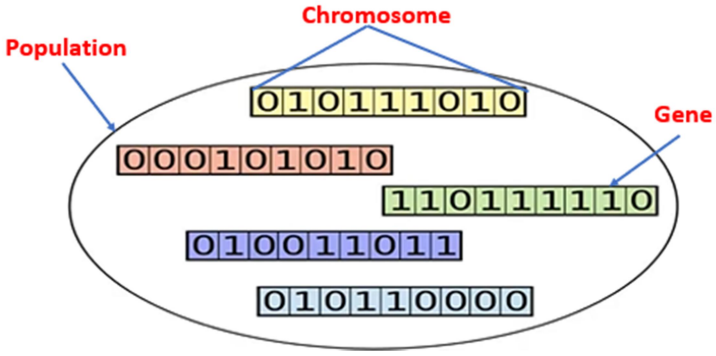
- **Population**. A set of individuals in a generation is called a population, $P(t) = \{x_1, x_2, \dots, x_n\}$, where x_i, is the ith individual.

- **Chromosome**. Each individual x_i in a population is a single chromosome. A chromosome, sometimes called a genome, is a set of parameters that define a solution to the problem.
- **Gene**. Each chromosome x comprises of a string of elements g_i , called genes, i.e., $x = (g_1, g_2, \dots, g_n)$, where n is the number of genes in the chromosome. Each gene encodes a parameter of the problem into the chromosome. A gene is usually encoded as a binary string or a real number.

- **Alleles** are the smallest information units in a chromosome

- **Genotype**. A genotype is biologically referred to the underlying genetic coding of a living organism, usually in the form of DNA. In EAs, a genotype represents a coded solution, that is, an individual's chromosome.

- **Fitness**. Fitness in biology refers to the ability of an individual of certain genotype to reproduce. The set of all possible genotypes and their respective fitness values is called a fitness landscape.



4.3. Encoding/Decoding

coding and encoding are fundamental aspects of genetic algorithms that determine how solutions are represented and manipulated. The choice of encoding scheme can greatly influence the performance and effectiveness of the GA.

Coding: The general process of transforming a problem's solution into a format suitable for a genetic algorithm.
Encoding: The specific method or scheme used to represent the solution. Encoding is a more detailed aspect of coding, specifying how the solution is structured and what data types are used.

Types of Encoding

- **Binary coding:** Solutions are represented as binary strings (sequences of 0 and 1). Example: For a problem with three binary parameters, a solution might be encoded as [1, 0, 1].

Advantages: Simple to implement, easy to apply genetic operators.
Disadvantages: May not be efficient for problems with large or continuous parameter spaces.

- **Real-Valued Encoding:** Solutions are represented as vectors of real numbers. Example: For a problem with three real-valued parameters, a solution might be encoded as [0.5, 1.2, 0.8].
  Advantages: More natural for problems with continuous parameters, can handle a wider range of values.
  Disadvantages: Requires more complex genetic operators.

- **Integer Encoding:** Solutions are represented as vectors of integers. Example: For a problem with three integer parameters, a solution might be encoded as [3, 7, 2].
  Advantages: Suitable for problems with discrete parameters.
  Disadvantages: May require special handling for genetic operators.

Example of Encoding in a Genetic Algorithm
Let's consider a simple optimization problem where we want to maximize the function $f(x, y) = x^2 + y^2$ over the range $-10 \leq x, y \leq 10$

- **Binary Encoding**: If we use 10 bits for each parameter, a solution might be encoded as [1010101010, 0101010101]. For decoding, convert the binary strings back to real numbers within the range [-10, 10].
- **Real-Valued Encoding**: A solution might be encoded as [3.5, -2.7].

4.4. Selection
Selection, also known as reproduction, is a critical step in genetic algorithms (GAs) that determines which individuals (solutions) from the current population will contribute to the next generation. The goal of selection is to favor the reproduction of fitter individuals, thereby increasing the likelihood that their beneficial traits will be passed on to future generations. This process mimics natural selection in biological evolution.

Purpose of Selection

- Fitness Propagation: Ensures that individuals with higher fitness scores have a higher probability of contributing to the next generation, thereby propagating their beneficial traits.
- Population Improvement: Over successive generations, selection helps improve the overall fitness of the population by favoring better solutions.

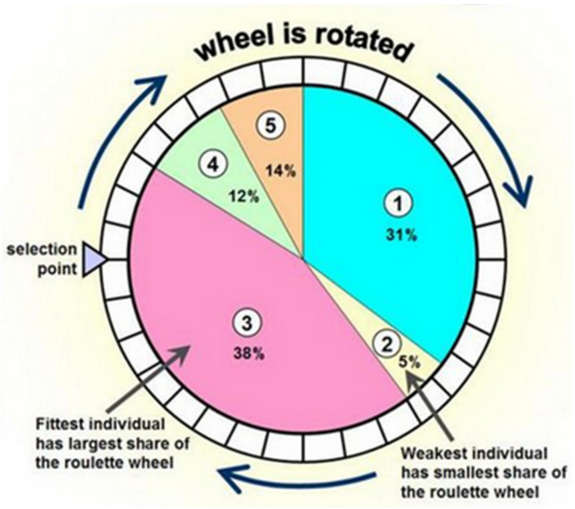
- Diversity Maintenance: Balances the need to exploit good solutions with the need to explore new areas of the search space, maintaining genetic diversity.

Types of Selection Methods

1. **Roulette Wheel Selection (Fitness Proportionate Selection):**
    - Each individual's probability of being selected is proportional to its fitness score.
        - Calculate the total fitness of the population. Assign a portion of the roulette wheel to each individual proportional to its fitness.

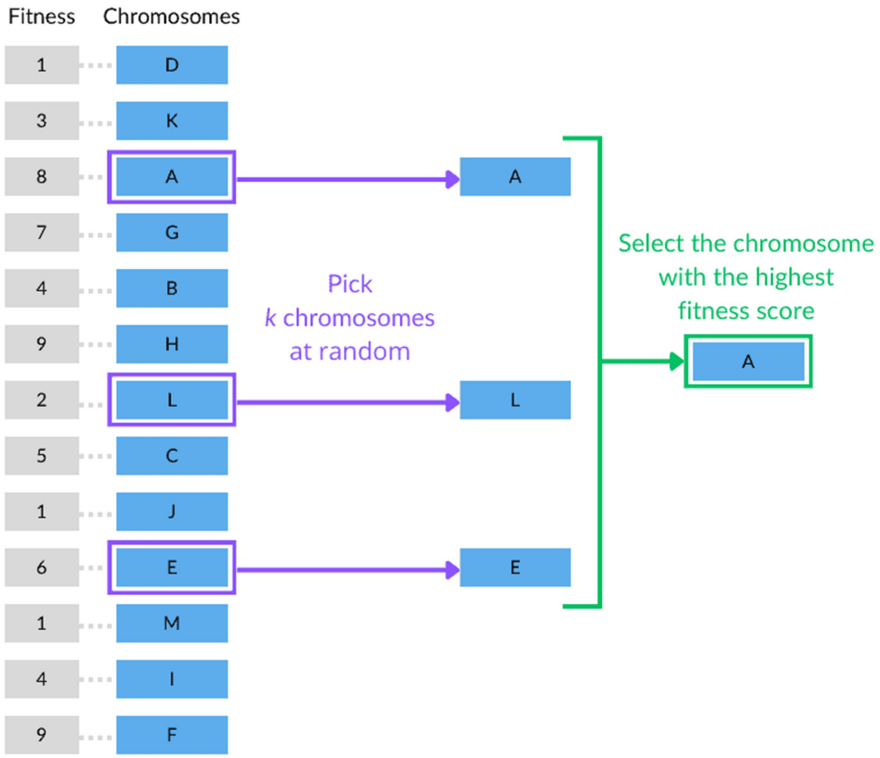
$$P_i = \frac{f(x_i)}{\sum_{i=1}^{N} f(x_i)}$$

        - Spin the roulette wheel to select individuals.



    - **Advantages**: Simple to implement.
    - **Disadvantages**: Can lead to premature convergence if a few individuals have significantly higher fitness.

2. **Tournament Selection:**
    - Randomly select a subset of individuals (tournament size) and choose the one with the highest fitness.
        - Randomly pick a subset of individuals from the population.
        - Select the individual with the highest fitness from the subset.



In this case, the tournament size is set to 3

- **Advantages**: Easy to implement, allows control over selection pressure by adjusting the tournament size.
- **Disadvantages**: May not always select the best individuals if the tournament size is small.

3. **Rank Selection:**
   - Individuals are ranked based on their fitness, and selection probabilities are assigned based on these ranks.
     - Rank individuals from best to worst in descending order.
     - Convert ranks into selection probabilities. Commonly, higher ranks are assigned higher probabilities, often using linear or exponential scaling
     - Assign selection probabilities based on ranks (e.g., linear or exponential ranking).

Example: Linear Rank Probability

$$P_i = \frac{1}{N}\left(\beta - 2(\beta - 1)\frac{i-1}{N-1}\right), i = 1, 2, \dots, N$$

Where β is selected in [0, 2], i is the rank, and N is the population size

- **Advantages**: Reduces the risk of premature convergence, as it spreads selection pressure more evenly.
- **Disadvantages**: May not exploit the best solutions as effectively as other methods.

4. **Elitism:**
    - The best individuals from the current generation are directly copied to the next generation without modification.
        - Identify the top-performing individuals.
        - Copy these individuals to the next generation.
    - **Advantages**: Ensures that the best solutions are preserved, preventing loss of good solutions.
    - **Disadvantages**: May reduce genetic diversity if too many elite individuals are preserved.

5. **Truncation Selection:**
    - Only the top fraction of the population is selected for reproduction.
        - Rank individuals based on fitness in descending order.
        - Select the top fraction (e.g., top 50%) for reproduction.
    - **Advantages**: Strong selection pressure, ensures that only the best individuals contribute to the next generation.
    - **Disadvantages**: May lead to premature convergence and loss of genetic diversity.

4.5. Crossover

Crossover, also known as recombination, is a fundamental genetic operator in genetic algorithms (GAs) that combines parts of two parent solutions to produce offspring. This process mimics the biological recombination of genetic material during reproduction. Crossover is crucial for exploring the search space and generating new solutions that inherit characteristics from both parents.

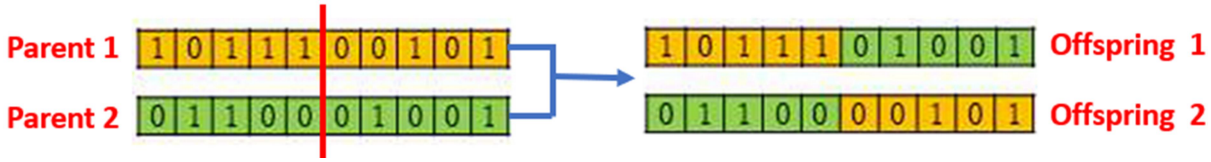
Purpose of Crossover

- **Exploration**: Crossover helps explore the search space by combining different parts of parent solutions, potentially leading to better solutions.
- **Inheritance**: It allows offspring to inherit beneficial traits from both parents, which can improve the overall fitness of the population.
- **Diversity**: Crossover maintains genetic diversity in the population, preventing premature convergence to suboptimal solutions.

Types of Crossover

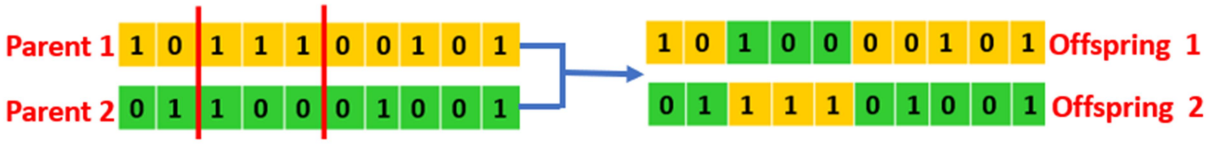
1. **Single-Point Crossover** :
    - A single crossover point is randomly chosen, and the genetic material is exchanged between the two parents at this point.

    Example: For binary-encoded chromosomes. Crossover point: 5



2. **Two-Point Crossover**:
   - Two crossover points are randomly chosen, and the genetic material between these points is exchanged between the two parents.

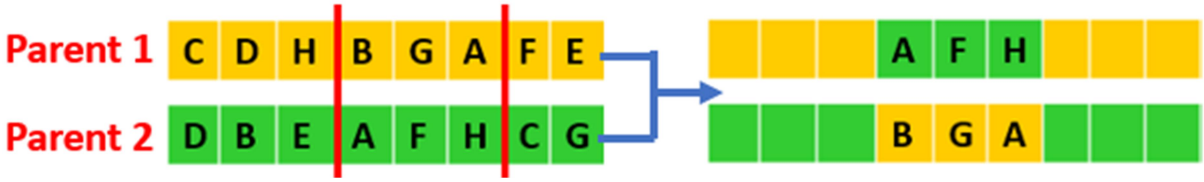
Example: For binary-encoded chromosomes. Crossover points: 2 and 5



3. **Uniform Crossover**:
   - Each gene in the offspring is randomly chosen from one of the two parents with a certain probability.

Example: For binary-encoded chromosomes:

4. **Order Crossover (OX)**:
   - Used for permutation-encoded chromosomes, such as in scheduling problems. A segment of one parent is copied to the offspring, and the remaining genes are filled in the order they appear in the other parent.



From Parent 1: F-E-C-D-H-B-G-A
Remove A-F-H
Remain: E-C-D-B-G

From Parent 2: C-G-D-B-E-A-F-H
Remove B-G-A
Remain: C-E-D-E-F-H

4.6. Mutation

Mutation is a fundamental genetic operator in genetic algorithms (GAs) that introduces random changes to the genetic material of individuals (solutions) in the population. This process mimics the biological phenomenon of mutation, where random changes occur in the DNA of organisms. In the context of GAs, mutation helps maintain genetic diversity, explore new areas of the search space, and prevent premature convergence to suboptimal solutions.

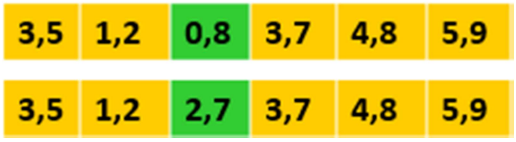
Purpose of Mutation

- **Diversity**: Introduces new genetic material into the population, maintaining diversity and preventing the population from becoming too homogeneous.
- **Exploration**: Helps explore new areas of the search space by randomly altering solutions, potentially leading to the discovery of better solutions.
- **Escape from Local Optima**: Allows the algorithm to escape from local optima by introducing random changes, which can help the population move towards the global optimum.

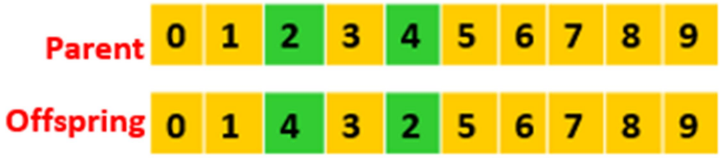
Types of Mutation

1. **Bit-Flip Mutation (Binary Encoding)**:
    - Randomly flips one or more bits in a binary-encoded chromosome.
        - For each bit in the chromosome, flip the bit (change 0 to 1 or 1 to 0) with a certain probability (mutation rate).

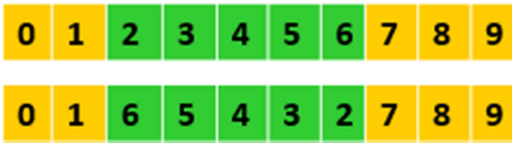
2. **Uniform Mutation (Real-Valued Encoding)**:
    - Randomly selects a gene and replaces it with a uniformly random value within the allowed range.
        - For each gene in the chromosome, replace the gene with a random value within the specified range with a certain probability.



3. **Swap Mutation (Permutation Encoding)**:
    - Randomly selects two genes and swaps their positions.
        - Randomly select two positions in the chromosome and swap the genes at these positions.



4. **Inversion Mutation (Permutation Encoding)**:
    - Randomly selects a segment of the chromosome and reverses the order of the genes within that segment.



Mutation Rate

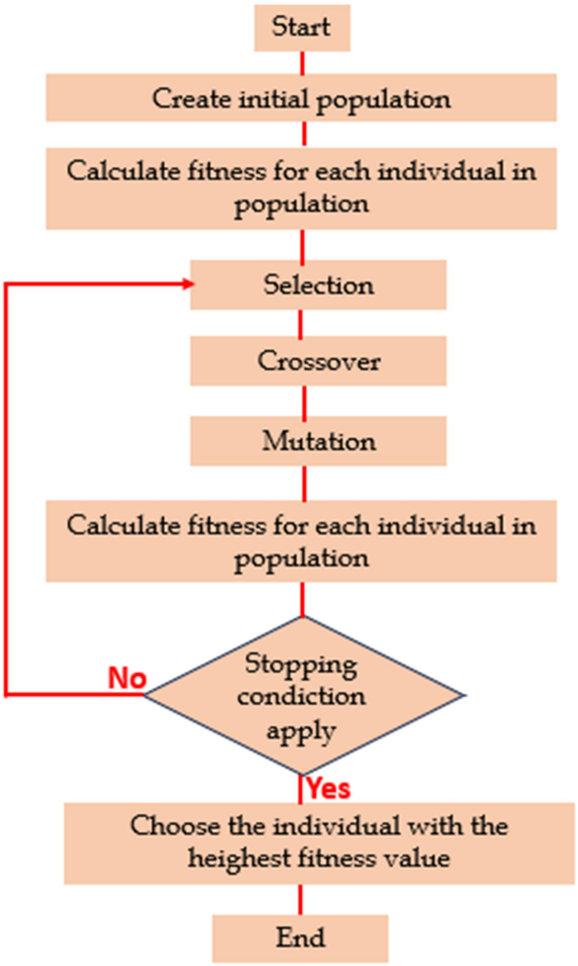
The probability that a gene in a chromosome will be mutated.

- High Mutation Rate: Can introduce too much randomness, leading to a loss of good solutions and slowing down convergence.
- Low Mutation Rate: May not introduce enough diversity, leading to premature convergence and stagnation.
- Balance: Finding the right mutation rate is crucial for balancing exploration and exploitation in the genetic algorithm.

4.7. Algorithm Structure

The structure of a genetic algorithm consists of several key components and steps that guide the evolution of potential solutions.

4.7.1. Basic flow



4.7.2. Steps of a Genetic Algorithm

1. **Initialization:**

   The first step in the genetic algorithm process involves the initialization of a random population of individuals, each representing a potential solution to the problem at hand. Each solution is represented as a chromosome, which can be encoded. The number of individuals in the population is typically fixed but can vary based on the problem and computational resources.

   The diversity of this initial population is crucial, as it allows the algorithm to explore a broad range of possible solutions, some of which may be closer to the optimal solution than others.

2. **Fitness Evaluation:**

   Once the population is initialized, each individual's fitness is evaluated using a predefined fitness function. This function assesses how well each solution addresses the problem's objectives. Each individual is assigned a fitness score based on its performance, which helps guide the algorithm toward better solutions by favoring individuals with higher scores.

3. **Selection:**
   The selection process involves choosing individuals from the current population to serve as parents for the next generation. This is typically based on their fitness scores, with fitter individuals having a higher chance of being selected. This process is commonly referred to as "survival of the fittest," which emphasizes the principle that only the best solutions will contribute to the next generation.

4. **Crossover (Recombination):**
   Once the parents are selected, the crossover operator is applied. This operator combines the genetic information of two parent solutions to produce one or more offspring. The offspring inherit a mixture of genes from both parents, potentially leading to new and improved solutions.

5. **Mutation:**
   A mutation operator may introduce random changes to the genes of the offspring, maintaining genetic diversity within the population and helping the algorithm to escape local optima.

6. **Termination:**
   The algorithm stops when one of the defined termination conditions is met. These conditions can include a maximum number of generations produced or the attainment of a fitness score that meets or exceeds a set threshold. This iterative process ensures that with each generation, the population evolves, becoming increasingly adapted to solve the problem at hand.

7. **Iteration:**
   The genetic algorithm follows an iterative cycle of evaluation, selection, crossover, and mutation. This cycle continues until a termination condition is met. Common termination conditions include reaching a predefined number of generations or achieving a satisfactory fitness level, indicating that an optimal or near-optimal solution has been found.

4.8. Advantages of Genetic Algorithms

- Flexibility: Genetic algorithms can be applied to a wide range of optimization problems, including those with non-linear, discontinuous, or multimodal objective functions.
- Global Optimization: Genetic algorithms can explore the search space more effectively than traditional optimization methods, reducing the risk of getting stuck in local optima.

- Parallelism: Genetic algorithms can be easily parallelized, allowing for efficient computation on modern hardware.
- Adaptability: Genetic algorithms can adapt to changing environments or problem constraints by modifying the fitness function or the genetic operators.

4.9. Limitations of Genetic Algorithms

- Computational Complexity: Genetic algorithms can be computationally intensive, especially for large-scale problems.
- Parameter Tuning: The performance of genetic algorithms can be sensitive to the choice of parameters, such as population size, crossover rate, and mutation rate.
- Premature Convergence: Genetic algorithms may converge to a suboptimal solution if the population lacks diversity or the fitness function is not well-designed.

Exercises

Exercise 1

Use Simulated Annealing (SA) to find the global maximum of the function:

$$f(x) = -x2 + 4x + 10$$

within the range $x \in [0,5]$

Exercise 2

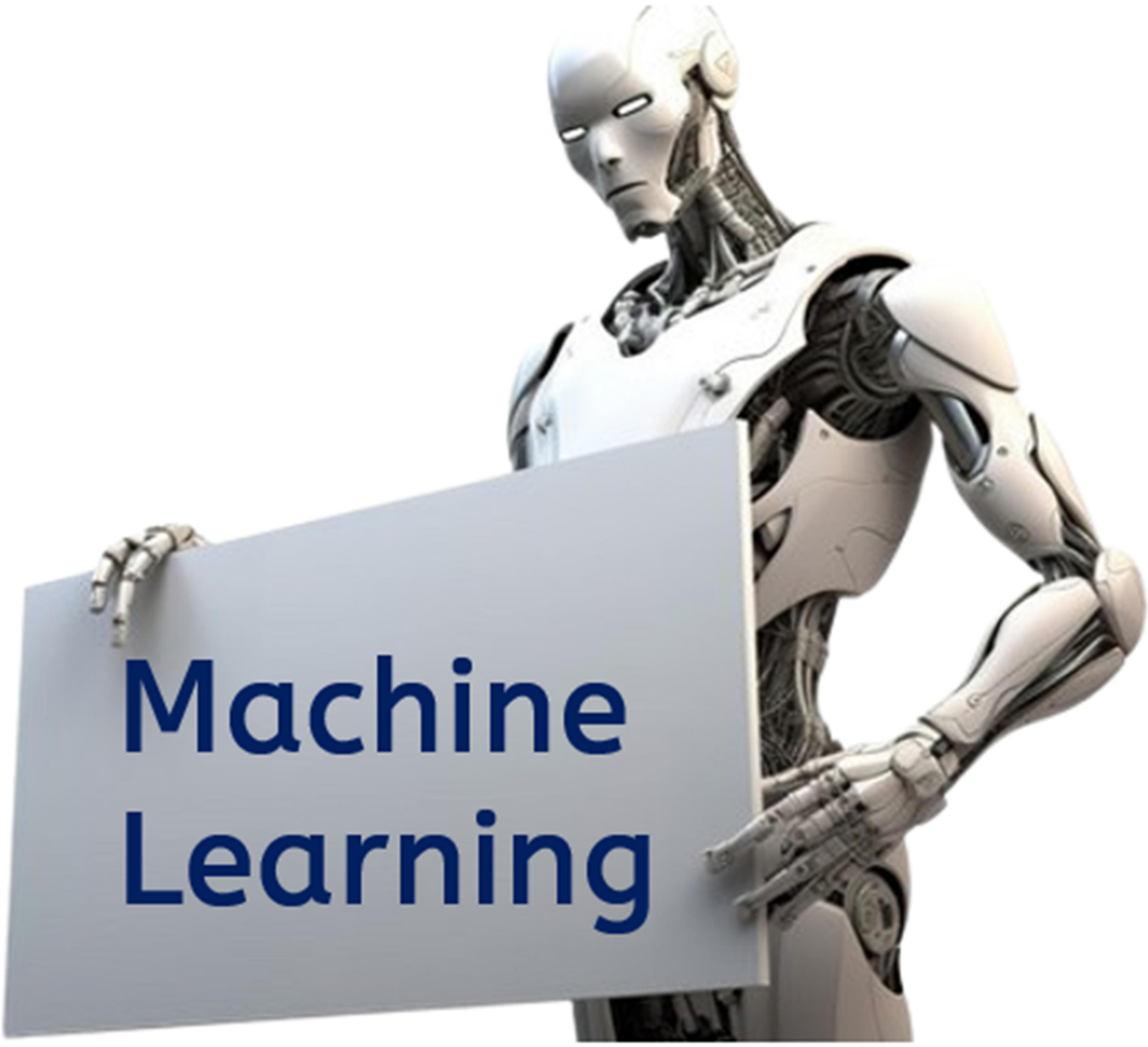
Use Simulated Annealing to find a near-optimal solution for a 5-city TSP. The distances between cities are given in the table below:

	A	**B**	**C**	**D**	**E**
A	0	2	9	10	7
B	2	0	6	8	3
C	9	6	0	5	4
D	10	8	5	0	6
E	7	3	4	6	0

Exercise 3

Use a Genetic Algorithm (GA) to find a near-optimal solution for the same 5-city TSP from the previous exercise.

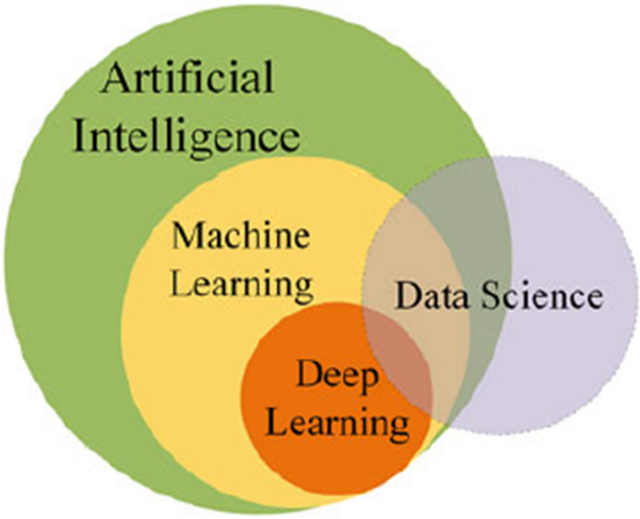
Chapter 6
Machine Learning and Neural Networks



1. Introduction

Difference Between Data Science, Machine Learning, Artificial Intelligence, Deep Learning

Data science, artificial intelligence, machine learning, and deep learning are closely related terminologies. However, these are distinctly separate fields of technology. Machine learning falls within the subset of artificial intelligence, while deep learning is considered to fall within the subset of machine learning, as is demonstrated by Figure below.



The difference between ML and deep learning is in the fact that deep learning requires more computing resources and very large datasets. Deep learning is especially helpful in handling large volumes of text or images.

Data science is an interdisciplinary field that involves identifying data patterns and making inferences, predictions, or insights from the data. Data science is closely related to deep learning, data mining, and big data. Here, data mining is the field that
deals with identifying patterns and extracting information from big datasets using techniques that combine ML, statistics, and database systems, and by definition, big data refers to vast and complex data that are too huge to be processed by traditional systems using traditional algorithms. ML is one of the primary tools used to aid the data analysis process in data science, particularly for making extrapolations or predictions on future data trends.

2. What Is Machine Learning?

Machine Learning (ML) is a field of artificial intelligence that focuses on the development of algorithms and statistical models that enable computers to perform specific tasks without explicit instructions, relying on patterns and inference instead.

The primary goal of machine learning is to allow computers to learn from and make predictions or decisions based on data.

- The computer scientist and machine learning pioneer Arthur Samuel defined machine learning as the "field of study that gives computers the ability to learn without being explicitly programmed".
- Tom Mitchell's defined machine learning as "the study of computer algorithms that allows computer programs to automatically improve through experience".
- He defines learning as follows: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

3. Goals of Machine Learning

The primary goal of Machine Learning (ML) is to enable computers to learn from and make predictions or decisions based on data, without being explicitly programmed for each specific task.

- Pattern Recognition: Identify and understand patterns within data to make informed decisions or predictions.
- Automation: Automate tasks that would typically require human intelligence, such as image recognition, speech recognition, and natural language processing.
- Generalization: Develop models that can generalize well from training data to new, unseen data, ensuring robust performance in real-world applications.
- Optimization: Improve the efficiency and effectiveness of processes by optimizing parameters and making data-driven decisions.
- Adaptation: Create systems that can adapt and improve over time as they are exposed to more data, allowing for continuous learning and improvement.
- Insight Generation: Extract valuable insights and knowledge from large datasets, helping to uncover hidden trends, correlations, and anomalies.
- Personalization: Tailor experiences and recommendations to individual users based on their preferences and behaviors, enhancing user satisfaction and engagement.

By achieving these goals, machine learning enables the development of intelligent systems that can solve complex problems, enhance decision-making, and drive innovation across various industries.

4. Types of Machine Learning

Machine Learning (ML) can be categorized into several types based on the nature of the learning process and the data used.

Each type of machine learning has its own strengths and is suited to different kinds of problems and datasets. The choice of which type to use depends on the specific application and the nature of the data available.

4.1. Supervised Learning

In this approach, the algorithm learns from **labeled data**, where each training example is paired with an **output label**. The goal is to learn a mapping from inputs to outputs so that the model can accurately predict the output for new, unseen data.

Common algorithms used in supervised learning include Linear Regression, Logistic Regression, Support Vector Machines (SVM), Decision Trees, Random Forests, and Neural Networks.

4.2. Unsupervised Learning

Unsupervised Learning involves training algorithms on data **without labeled responses**. The goal is to infer the natural structure present within a set of data points. This type of learning is useful for tasks such as clustering and dimensionality reduction.

- Clustering aims to group similar data points together based on their features, which is useful in applications like customer segmentation and anomaly detection.
- Dimensionality reduction techniques, such as Principal Component Analysis (PCA), help in reducing the number of input variables while retaining as much variance as possible, making the data easier to visualize and analyze.

Common algorithms in unsupervised learning include K-Means Clustering, Hierarchical Clustering, and t-SNE.

4.3. Semi-Supervised Learning

Semi-Supervised Learning combines a small amount of labeled data with a large amount of unlabeled data. This approach is particularly useful when obtaining labeled data is expensive or time-consuming. By leveraging both labeled and unlabeled data, semi-supervised learning can improve the accuracy and robustness of the model. Common algorithms include self-training, co-training, and multi-view training, where the model iteratively labels the unlabeled data and uses it to improve its predictions.

4.4. Reinforcement Learning

Reinforcement Learning focuses on training algorithms to make a sequence of decisions by performing actions in an environment to achieve a goal. The agent receives rewards or penalties for the actions it performs, learning to optimize its behavior over time.

This type of learning is widely used in applications such as game playing (e.g., AlphaGo), robotics, and autonomous vehicles.

Common algorithms in reinforcement learning include Q-Learning, Deep Q-Networks (DQN), Proximal Policy Optimization (PPO), and Actor-Critic Methods.

4.5. Transfer Learning

Transfer Learning involves applying knowledge gained from one task to improve learning on a related but different task. This approach is particularly useful in scenarios where there is a limited amount of data for the target task. By leveraging pre-trained models, transfer learning can significantly reduce the time and data required to train a new model.

For example, a model pre-trained on a large dataset of images can be fine-tuned for a specific task, such as medical image analysis. Common techniques include fine-tuning and feature extraction.

4.6. Ensemble Learning

Ensemble Learning combines the predictions from multiple models to improve overall performance. The idea is that a group of weak learners can collectively make better predictions than any single model. Ensemble methods can be categorized into bagging, boosting, and stacking.

- Bagging involves training multiple models on different subsets of the data and averaging their predictions, as seen in Random Forests.
- Boosting sequentially trains models to correct the errors of previous models, as in AdaBoost and Gradient Boosting.
- Stacking combines the predictions of multiple models using a meta-learner.

4.7. Deep Learning

Deep Learning is a subset of machine learning that uses neural networks with many layers to model complex patterns in data. These networks, known as deep neural networks, can automatically learn and extract features from raw data, making them highly effective for tasks such as image recognition, speech recognition, and natural language processing.

Common algorithms in deep learning include Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Long Short-Term Memory (LSTM) networks, and Generative Adversarial Networks (GANs).

Deep learning has revolutionized various fields by achieving state-of-the-art performance in many applications.

5. Machine Learning Tasks

5.1. Regression

Regression is a statistical method used to model and analyze the relationship between a dependent variable (output) and one or more independent variables (inputs).

The primary goal of regression is to find a mathematical function that best describes this relationship, allowing for the prediction of continuous output values.

For example, linear regression models the relationship as a linear equation, while polynomial regression includes polynomial terms to capture more complex relationships.

Other types of regression, such as ridge and lasso regression, incorporate regularization techniques to prevent overfitting and improve model generalization.

Regression is widely used in various fields, including finance for predicting stock prices, healthcare for predicting patient outcomes, and real estate for estimating property values.

5.2. Classification

Classification is a supervised learning task that involves assigning input data into predefined categories or classes.

The primary goal of classification is to learn a mapping from input features to output labels, allowing the model to predict the class of new, unseen data points.

For example, in email spam detection, a classification algorithm learns to distinguish between "spam" and "not spam" emails based on their content and metadata.

Common classification algorithms include logistic regression, decision trees, support vector machines (SVM), and neural networks.

Classification is used in a wide range of applications, such as image recognition for identifying objects in images, sentiment analysis for categorizing text sentiments, and medical diagnosis for classifying diseases based on patient data.

The performance of classification models is often evaluated using metrics like accuracy, precision, recall, and the F1 score.

5.3. Prediction

Prediction is a broad term that encompasses various methods used to forecast future values or outcomes based on historical data.

It includes both regression and classification tasks, as well as other types of forecasting techniques.

The goal of prediction is to make informed decisions by leveraging data to anticipate future trends, events, or behaviors.
For example, time series prediction involves analyzing historical time series data to forecast future values, such as stock prices or weather conditions.
Anomaly detection is another form of prediction that identifies unusual patterns or outliers in data, which is crucial in fields like fraud detection and network security. Prediction models can produce continuous values (regression), discrete labels (classification), probabilities, or other types of outputs, making them versatile tools for data-driven decision-making.

6. K-Nearest Neighbors (KNN)

6.1. Definition

K-Nearest Neighbors (KNN) is a simple and effective machine learning method used for classification and regression tasks. It is particularly popular for classification tasks due to its simplicity and efficiency.
KNN is a supervised learning method that classifies a data point based on the majority class of its k nearest neighbors in the feature space.

6.2. Distance Metrics

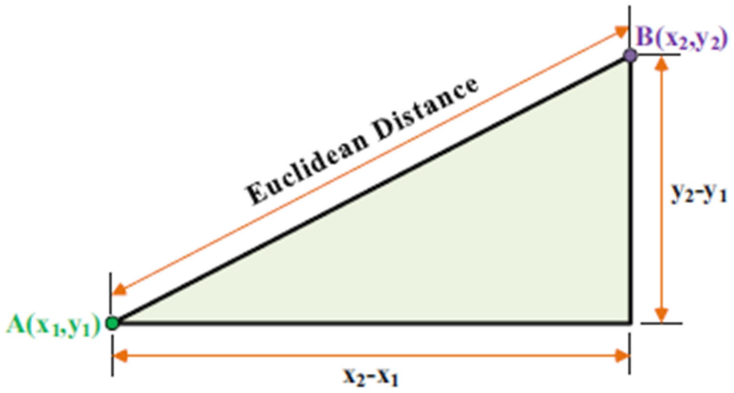
Distance metrics can be used to get an idea about the distance between two places, points, objects, etc. For example, distance metrics can tell how close or how far two points are from each other.
Furthermore, some machine learning algorithms, such as KNN classification, K-means clustering, self-organizing map (SOM), SVM, etc., rely on the closeness between data points. Therefore, this "closeness" estimation can be considered a "distance" calculation.
Many algorithms such as the k-nearest neighbors algorithm and k-means algorithm use Euclidean distance as the objective function where the data dimension is low. However, in the case of higher dimensionality, the Euclidean distance may not perform effectively.

6.2.1. Euclidean Distance

The Euclidean function calculates the square root of the sum of the differences between two different data objects. Assume two data objects a and b, where each data object has n attributes, $a = (x_1, x_2, x_3, \ldots, x_n)$ and $b = (y_1, y_2, y_3, \ldots, y_n)$. The function to calculate the Euclidean distance between a and b is given by:

$$d(a,b) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + (x_3 - y_3)^2 + \cdots + (x_n - y_n)^2}$$

Figure below depicts the Euclidean distance between two data points A(x1, y1) and B(x2, y2) with two features x and y.

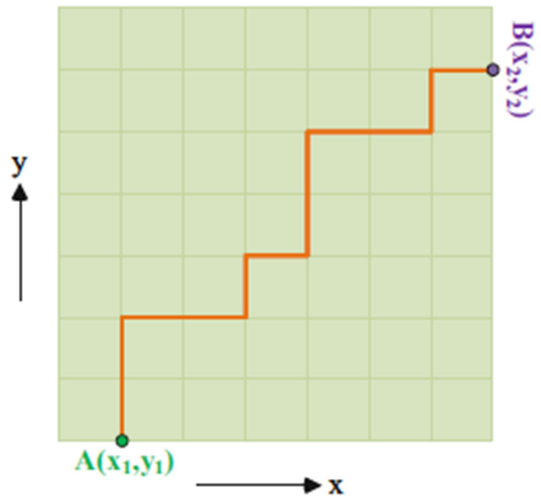


6.2.2. Manhattan Distance

Another popular distance metric is the Manhattan distance, also referred to as city block distance or taxicab metric. The intuition behind these names derives from calculating the shortest distance between any two blocks along the vertical and horizontal axes. The summation of absolute differences between two points is the Manhattan distance. The formula for calculating the Manhattan distance between two data points $a = (x_1, x_2, x_3, \ldots, x_n)$ and $b = (y_1, y_2, y_3, \ldots, y_n)$ is given by Equation below:

$$d(a, b) = |x_1 - y_1| + |x_2 - y_2| + \cdots + |x_n - y_n|$$

Figure below depicts the Manhattan distance between two data points A and B with two features, x and y. The Manhattan distance is preferred over the Euclidean distance in the case of higher dimensionality. It also works best for discrete attributes.

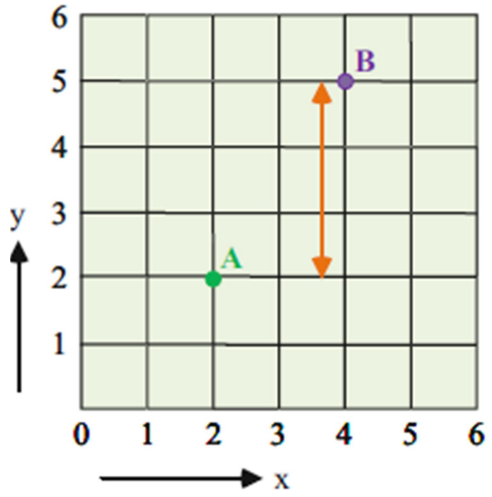


6.2.3. Chebyshev Distance

Chebyshev distance, also referred to as the supremum distance, is the maximum difference in the attributes between two data objects. The formula for calculating the Chebyshev distance between two data points $a = (x_1, x_2, x_3, \ldots, x_n)$ and $b = (y_1, y_2, y_3, \ldots, y_n)$ is given by Equation below:

$$d(a,b) = max(|x_1 - y_1|, |x_2 - y_2|, \ldots, |x_n - y_n|)$$

Figure below depicts the Chebyshev distance for two points A and B. The Chebyshev distance can be applied to specific logistical problems only. For instance,
it can determine the minimum moves required by the king on a chessboard to move from one square to another.



6.2.4. Cosine Similarity and Cosine Distance

Cosine similarity gives an estimation of the similarity between two or more vectors. The mathematical concept of cosine similarity is quite simple. It is the cosine of the angle between two vectors. Cosine similarity can be calculated by dividing the dot product of two vectors by the product of the magnitude of the two vectors. If θ is the angle between two vectors $\vec{A}$ and $\vec{B}$, the cosine similarity can be calculated using Equation below:

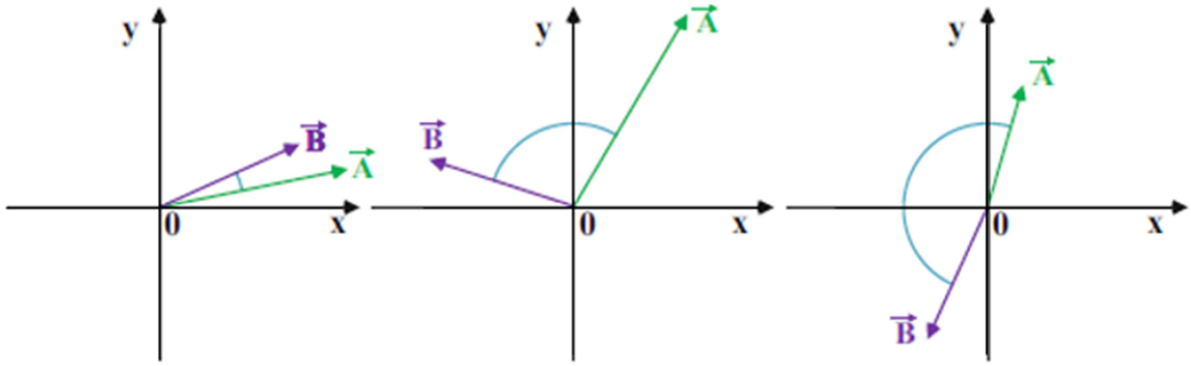
$$cos(\theta) = \frac{\vec{A}.\vec{B}}{\|\vec{A}\| \, \|\vec{B}\|} = \frac{\sum_{i=1}^{n} A_i \, B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \; \sqrt{\sum_{i=1}^{n} B_i^2}}$$

The two vectors $\vec{A}$ and $\vec{B}$ are non-zero, and they exist within an inner product space. The value of the cosine similarity metric ranges from −1 to 1, where −1 means total dissimilarity and 1 means total similarity between the two vectors.
Cosine distance can be calculated from cosine similarity. Cosine distance is opposite to cosine similarity. The greater the distance, the less the similarity between data

objects. The cosine distance is calculated as shown in Equation below:

$$d(\vec{A},\vec{B}) = 1 - cos(\theta) = 1 - \frac{\vec{A}.\vec{B}}{\|\vec{A}\|\,\|\vec{B}\|}$$

Figure below depicts the different cases of determining cosine similarity. Cosine similarity is preferable when two data objects vary in their size. The Euclidean distance between these two data objects could be huge due to their length mismatch, even if they have similarities. This similarity can be captured by cosine similarity because the angle remains constant regardless of the size of the data objects.



(a) Similar Vectors (b) Unrelated Vectors (c) Opposite Vectors

Cosine similarity between two (**a**) similar, (**b**) unrelated, and (**c**) opposite vectors. The similar vectors have nearly 0◦ angle between them. The angle between unrelated vectors would be around 90◦ or a bit more. Two opposite vectors would have nearly 180◦ angle between them. There is no correlation between the length of the vectors and their cosine similarity

Example

Suppose we have two vectors A=[1, 2, 3] and B=[4, 5, 6]

Dot Product:

$$A.B = 1 \times 4 + 2 \times 5 + 3 \times 6 = 32$$

Magnitudes:

$$\|A\| = \sqrt{1^2 + 2^2 + 3^2} = \sqrt{14}$$

$$\|B\| = \sqrt{4^2 + 5^2 + 6^2} = \sqrt{77}$$

Cosine Similarity:

$$cos(\theta) = \frac{32}{\sqrt{14}\sqrt{77}} = \frac{32}{\sqrt{1078}}$$

Cosine Distance:

$$d(A,B) = 1 - cos(\theta) = 1 - \frac{32}{\sqrt{1078}}$$

Euclidean distance

$$de(A,B) = \sqrt{(1-4)^2 + (2-5)^2 + (3-6)^2} = \sqrt{9+9+9} = \sqrt{27}$$

Manhattan distance

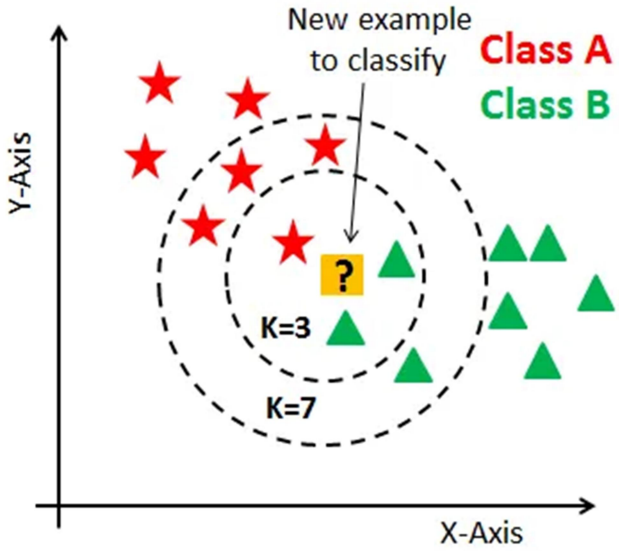
$$dm(A,B) = |1 - 4| + |2 - 5| + |3 - 6| = 9$$

Chebyshev distance

$$dc(A,B)dm(A,B) = \max(|1 - 2| + |2 - 4| + |3 - 6|) = 3$$

6.3. How KNN Works

1. Distance Metric: KNN uses a distance metric (such as Euclidean distance, Manhattan distance, or Minkowski distance) to measure the similarity between data points.
2. Neighbor Selection: For a given data point, KNN identifies the k nearest neighbors in the training dataset.
3. Voting (for Classification): In classification tasks, KNN assigns the most common class among the k nearest neighbors to the data point.
4. Averaging (for Regression): In regression tasks, KNN predicts the value by averaging the values of the k nearest neighbors.



Guidelines for Choosing k

Choosing the optimal value of k in K-Nearest Neighbors (KNN) is crucial for the performance of the algorithm. The value of k determines how many neighbors influence the prediction for a given data point.

1. Small k Values:
   Pros: Captures local patterns and variations in the data.
   Cons: More sensitive to noise and outliers, which can lead to overfitting.
2. Large k Values:
   Pros: Smooths out the effect of noise and outliers, leading to more stable predictions.
   Cons: May overlook local patterns and variations, leading to underfitting.
3. Odd vs. Even k:
   For classification tasks, it is often beneficial to choose an odd value of k to avoid ties in voting.
   For regression tasks, the choice of odd or even k is less critical.

6.4. Example

A dataset is given with two features "A" and "B" and respective labels as True or False. Using the KNN algorithm, classify the **new data point (6, 5)** where $k = 3$.

Feature A	Feature B	Label
5	2	true
5	4	true
7	4	false
8	6	false
7	3	false

We will calculate the Euclidean distance of the new data point (6, 5) from every data point in the dataset.

Feature A	Feature B	Label	Euclidian distance
5	2	true	$\sqrt{(5-6)^2+(2-5)^2}=3.16$
5	4	true	$\sqrt{(5-6)^2+(4-5)^2}=1.41$
7	4	false	$\sqrt{(7-6)^2+(4-5)^2}=1.41$
8	6	false	$\sqrt{(8-6)^2+(6-5)^2}=2.24$
7	3	false	$\sqrt{(7-6)^2+(3-5)^2}=2.24$

Now we will rank it in ascending order of the Euclidean distances.

Feature A	Feature B	Label	Euclidian distance
5	4	true	$\sqrt{(5-6)^2+(4-5)^2}=1.41$
7	4	false	$\sqrt{(7-6)^2+(4-5)^2}=1.41$
8	6	false	$\sqrt{(8-6)^2+(6-5)^2}=2.24$
7	3	false	$\sqrt{(7-6)^2+(3-5)^2}=2.24$
5	2	true	$\sqrt{(5-6)^2+(2-5)^2}=3.16$

Since **k = 3**, we will pick the three nearest data points according to their Euclidean distance.
The three nearest data points to the new data point are **(5, 4), (7, 4),** and **(8, 6)**.
Now according to their respective labels, we have to classify the new data point.
Data point (5, 4) classifies the data point as "True," whereas both the data points (7, 4) and (8, 6) classify the new data point as "False."
Since most of the k-nearest neighbor is classifying the new data point as "False," the new data point (6, 5) is "**False**."

6.5. Advantages of KNN

1. Simplicity: KNN is easy to understand and implement. It does not require an explicit training phase, making it quick to deploy.
2. Non-Parametric: KNN is a non-parametric method, meaning it does not make assumptions about the underlying data distribution.
3. Efficiency for Small Datasets: KNN can be very efficient for small datasets where the relationships between data points are clear.
4. Flexibility: KNN can be used for both classification and regression tasks and can handle multidimensional data.

6.6. Limitations of KNN

1. Computational Complexity: KNN can be computationally expensive for large datasets, as it requires calculating the distance between the test data point and all training data points.
2. Sensitivity to Irrelevant Features: KNN can be sensitive to irrelevant or noisy features, which can affect the accuracy of the classification.
3. Choice of k: The choice of the number of neighbors (k) is crucial and can impact the model's performance. A small k can lead to overfitting, while a large k can lead to underfitting.
4. Sensitivity to Feature Scaling: KNN is sensitive to the scale of the features. Features need to be normalized or standardized to prevent certain features from dominating others.

7. Neural Networks

7.1. Biological Neuron

The total number of neurons in the human brain is estimated to be between 86 and 100 billion. Each neuron can be described as a very simple processor that receives signals in the form of electrical potentials, computes the summation of the received potentials relative to a threshold, and, if the threshold is exceeded, emits a signal in turn. These billions of elementary processors are highly interconnected and operate in parallel.

The neuron is composed of a cell body called the soma and two types of extensions: the axon, which is singular and acts as the transmitter, conducting the action potential in a centrifugal manner, and the dendrites, which average around 7,000 per neuron and serve as the receptors for the action potential.

Between neurons, there exists a discontinuity called the synapse, which is the primary relay ensuring the transmission of the nerve impulse (action potential). Each neuron has between 1 and over 100,000 synapses, with an average of 10,000.

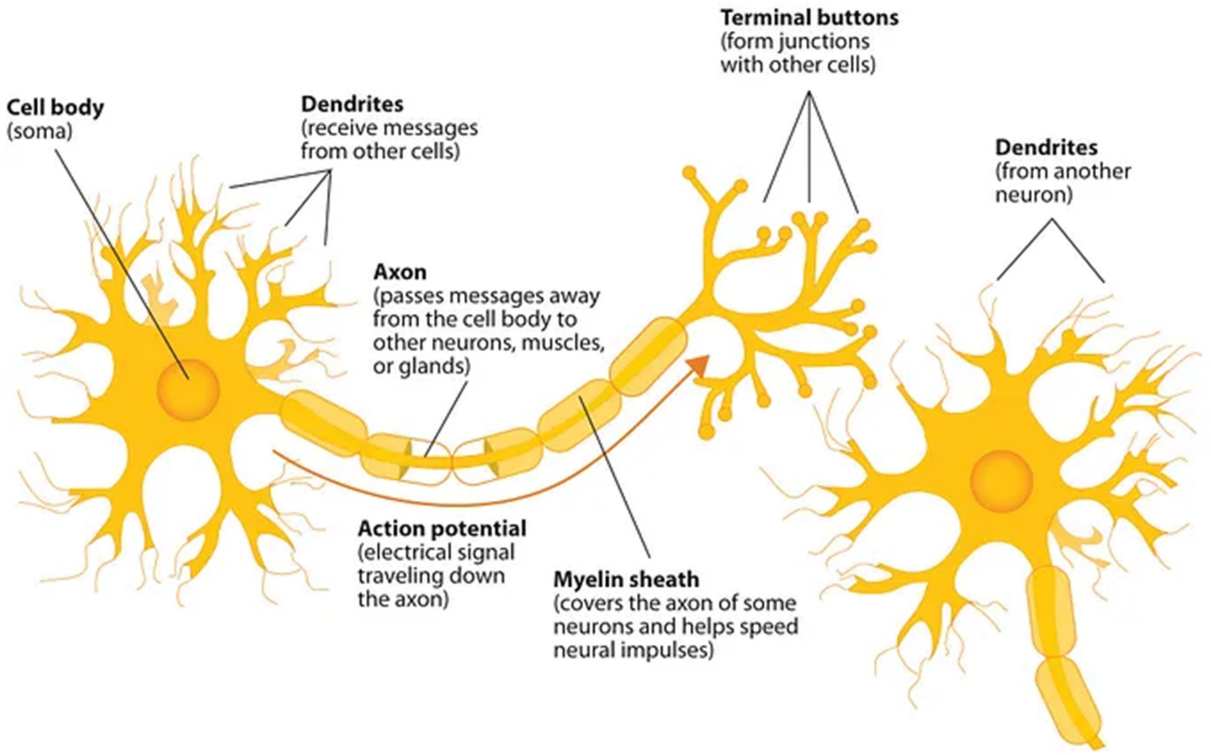
Dendrites and synapses work together to facilitate communication between neurons. A single dendrite can host hundreds or even thousands of synapses, depending on its branching complexity. Synapses can also form on the **soma (cell body)** and **axon**, not just on dendrites.

The high number of dendrites and synapses reflects the brain's need for **massive parallel processing** and **information integration**.

This complexity allows neurons to:

- Receive and integrate inputs from thousands of other neurons.
- Adapt and rewire connections (synaptic plasticity), which is the basis of learning and memory.
- Maintain redundancy and robustness in neural circuits, ensuring functionality even if some connections are lost.

The morphology, location, and number of these extensions, as well as the shape of the soma, vary and contribute to defining different morphological families of neurons.



When a relatively weak current is injected into a neuron, its potential changes in proportion to the current. However, if the current is sufficiently strong, a sudden rise in potential (depolarization) is observed, reaching a typical value of 40 mV, followed by a rapid decline (hyperpolarization). This phenomenon is referred to as the action potential.

7.2. History

Artificial Intelligence (AI) has evolved with the objective of simulating the behaviors of the human brain. Early attempts to model the brain date back to a time even before the computer era.

The origin of the inspiration for artificial neural networks can be traced to 1890, when William James introduced the concept of associative memory. He proposed what would later become a fundamental principle for the learning process of neural networks, known as Hebb's rule. A few decades later, in 1949, Warren McCulloch and Walter Pitts lent their names to a model of the biological neuron (a neuron with binary behavior). They were the first to demonstrate that simple formal neural networks could perform complex logical, arithmetic, and symbolic functions.

The first successes in this field date back to 1957, when Frank Rosenblatt developed the Perceptron model. He built the first neurocomputer based on this model and applied it to the field of pattern recognition. Shortly thereafter, in 1960, the control engineer Bernard Widrow introduced the Adaline (Adaptive Linear Element) model.

In its structure, Adaline resembles the Perceptron; however, its learning rule is different.

In 1969, Marvin Minsky and Seymour Papert published a seminal work that highlighted the theoretical limitations of the Perceptron. These limitations primarily concern the model's inability to handle non-linear problems, which significantly constrained its applicability to more complex tasks.

Following a period of relative stagnation from 1967 to 1982, the field experienced a revival in 1982 thanks to John J. Hopfield. He introduced a theory on the functioning and capabilities of neural networks. Hopfield first defined the desired behavior of his model and then constructed the corresponding structure and learning rule to achieve the intended outcome.

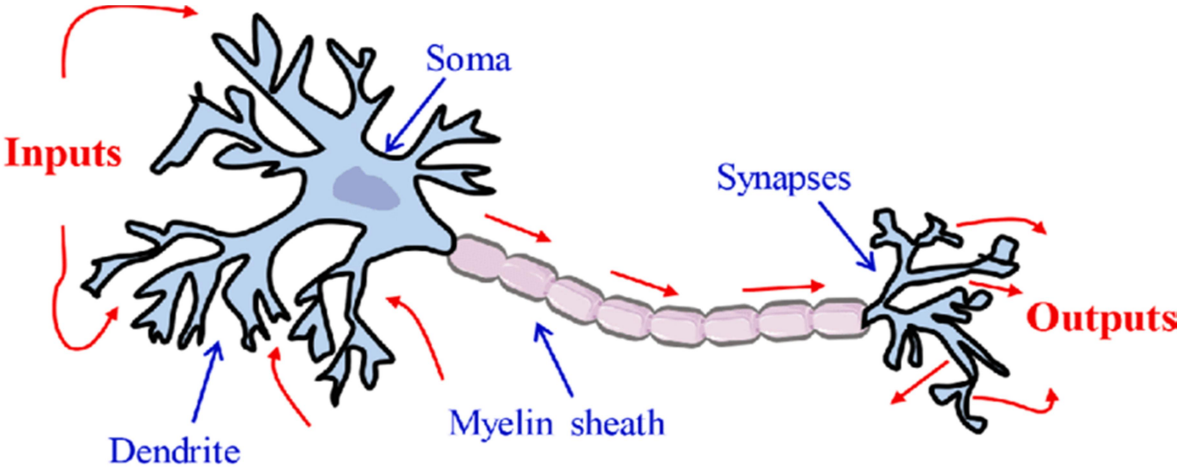
Subsequently, in 1985, the backpropagation algorithm emerged. This learning algorithm was specifically designed for Multi-Layer Perceptrons. With this breakthrough, it became possible to implement a non-linear input/output function on a network by decomposing the function into a series of linearly separable steps.

After being overshadowed in the mid-1990s by other machine learning or statistical algorithms—such as boosting and support vector machines—neural networks have experienced a resurgence of interest and even widespread media attention under the name of deep learning. The availability of large-scale datasets, particularly image datasets sourced from the internet, combined with significant advances in computational power, has enabled the estimation of millions of parameters in perceptrons that stack dozens or even hundreds of layers of neurons with highly specialized properties. This media success is a direct result of the remarkable achievements of these networks in areas such as image recognition, Go gameplay, natural language processing, and more.

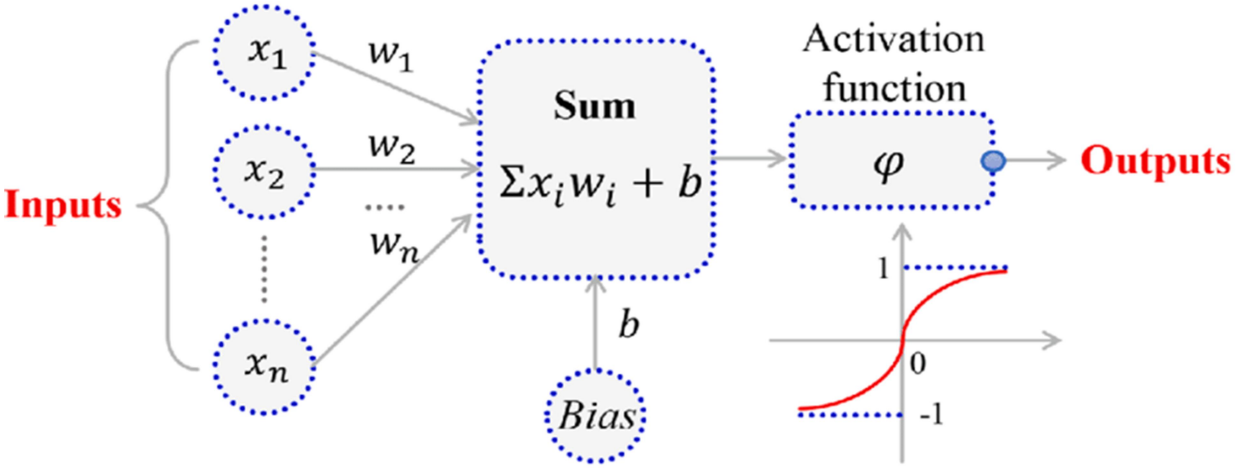
7.3. Artificial Neuron

7.3.1. Correspondence Between Biological Neurons and Artificial Neurons

Figure below illustrates the structure of an artificial neuron. Each artificial neuron functions as an elementary processor. It receives a variable number of inputs from upstream neurons. Each of these inputs is associated with a weight w (weight), which represents the strength of the connection. Each elementary processor has a single output, which then branches out to feed a variable number of downstream neurons.



(a) Biological neuron



(b) Artificial neuron

The structure and functionality of artificial neurons are inspired by their biological counterparts.

- Inputs (Dendrites)
- Biological Neuron: Dendrites receive electrical signals (action potentials) from other neurons via synapses.
- Artificial Neuron: Inputs represent the signals received from other artificial neurons, analogous to the role of dendrites.

- Weights (Synaptic Strength)
- Biological Neuron: The strength of the connection between neurons is determined by the efficiency of synaptic transmission, which can be modulated (e.g., through synaptic plasticity).

- Artificial Neuron: Each input is associated with a weight w, which represents the strength or importance of the connection. These weights are adjustable and are optimized during the learning process.

- Soma (Cell Body)
- Biological Neuron: The soma integrates incoming signals from dendrites. If the summed signal exceeds a certain threshold, the neuron generates an action potential.
- Artificial Neuron: The processing unit of the artificial neuron performs a weighted sum of the inputs and applies an activation function to determine the output.

- Activation Function (Threshold Mechanism)
- Biological Neuron: The neuron's firing mechanism is governed by a threshold; if the membrane potential exceeds this threshold, an action potential is triggered.
- Artificial Neuron: An activation function (e.g., sigmoid) is applied to the weighted sum of inputs to produce the output. This function introduces non-linearity and determines whether the neuron "fires."

- Output (Axon)
- Biological Neuron: The axon transmits the action potential to other neurons via synaptic connections.
- Artificial Neuron: The output of the artificial neuron is sent to other neurons in the network, mimicking the role of the axon.

- Synapses (Connections)
- Biological Neuron: Synapses are the junctions between neurons where neurotransmitters facilitate signal transmission.
- Artificial Neuron: The connections between artificial neurons, represented by weights, simulate the role of synapses in transmitting and modulating signals.

7.3.2. Formal Neuron

A neuron is essentially composed of an integrator that performs the weighted sum of its inputs. The result S of this summation is then transformed by a transfer function f, which produces the output y of the neuron. The inputs X of the neuron correspond to the vector $X[x_1, x_2, \dots, x_n]^T$, while $W[w_1, w_2, \dots, w_n]^T$ represents the vector of the neuron's weights.

The output S is given by the following equation:

$$S = \sum_{i=1}^{n} w_i x_i + b$$
$$= w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

which can also be expressed in matrix form as:

$$S = W^T X + b$$

This output corresponds to a weighted sum of the weights and inputs, plus what is called the bias b of the neuron. The result S of the weighted sum is referred to as the activation level of the neuron.

A bias b can be defined as the constant added to the product of the inputs and weights. It is used to adjust the output, helping models to shift the activation function toward the positive or negative side. This adjustment allows the model to better fit the data by providing additional flexibility in the learning process.

Finally, the output y of the neuron is computed by applying the activation function f.

$$y = f(S)$$
$$= f(W^T X + b)$$

The activation function *f* acts as a filter that adjusts the value of the previous sum to align with the characteristics of the desired output.

7.3.3. Bias

Bias refers to an additional learnable parameter in a model that allows the output of a neuron or layer to be shifted independently of the inputs. It is added to the weighted sum of inputs before applying the activation function, enabling the model to better fit the data.

The role of bias in neural networks is crucial for several reasons:

1. Shifts the Activation Function: Allows the model to adjust the position of the decision boundary or activation curve.
2. Enables Non-Zero Outputs: Ensures that neurons can produce meaningful outputs even when all inputs are zero.
3. Improves Flexibility: Increases the model's ability to represent complex relationships in the data.
4. It enhances convergence during training and aids in better generalization.

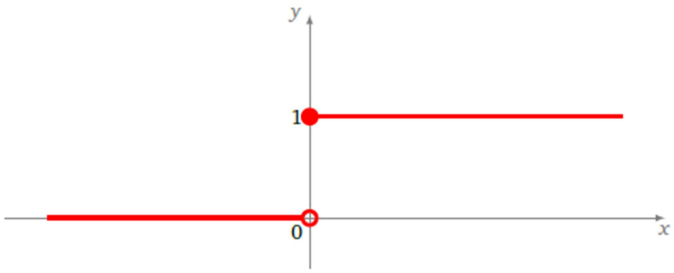
7.3.4. Activation Functions (Transfer Functions)

An activation function in neural networks is a mathematical function applied to the output of a neuron in order to introduce **non-linearity** into the model. Without activation functions, neural networks would only be able to learn and represent linear relationships between inputs and outputs, which severely limits their ability to solve complex real-world problems.

Different transfer functions can be used as activation functions for the neuron. The choice of activation function depends on the problem being solved and the architecture of the neural network. The most commonly used ones are the threshold, linear, and sigmoid functions.

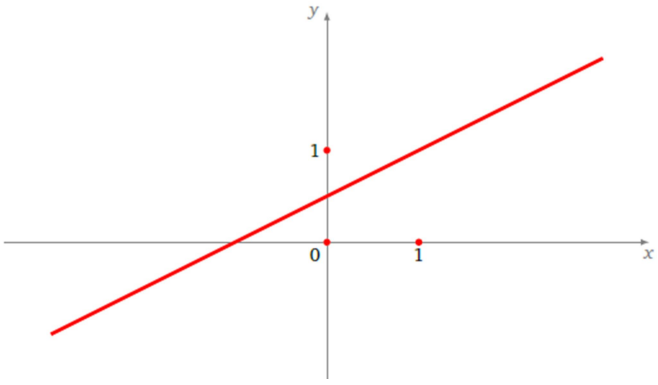
- **Threshold Function (Heaviside Step Function)**
  This function outputs 1 if the input is greater than or equal to a threshold (usually 0), and 0 otherwise. It is rarely used in modern neural networks due to its discontinuity. Rarely used in modern neural networks due to its binary nature and lack of gradient for optimization.

$$\begin{cases} f(x) = 0 & si\ x < 0 \\ f(x) = 1 & si\ x \geq 0 \end{cases}$$



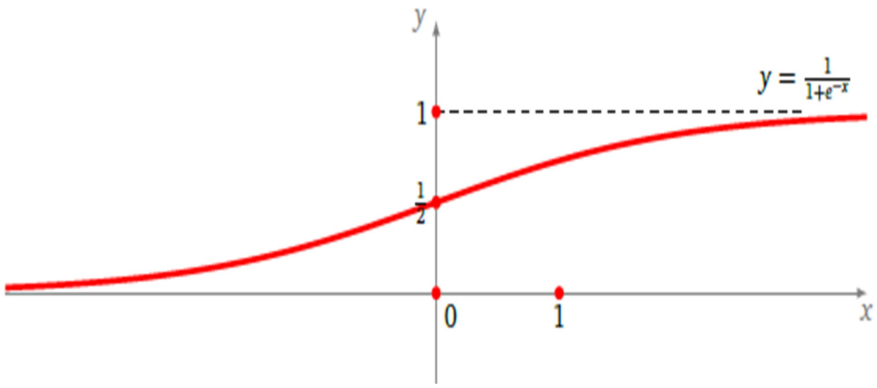
- **Linear Function**
  The output is directly proportional to the input. While simple, it lacks the ability to introduce non-linearity, limiting its use in deep learning

$$f(x) = ax + b$$



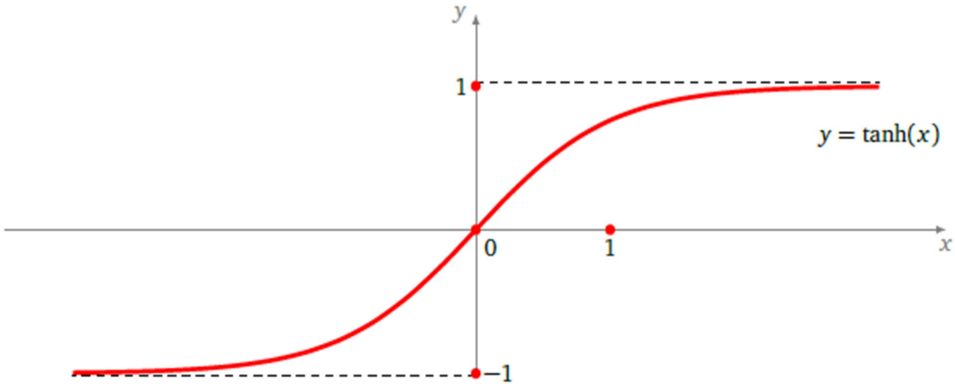
- **Sigmoid Function**
  This function maps input values to a range between 0 and 1, making it useful for binary classification tasks. However, it can suffer from the vanishing gradient problem. It suffers from the "**vanishing gradient**" problem when x is very large or very small.

$$f(x) = \frac{1}{1 + e^{-x}}$$



- **Hyperbolic Tangent (Tanh) Function**
  Similar to the sigmoid but maps inputs to a range between -1 and 1. It is also prone to the vanishing gradient problem. It suffers from the "**vanishing gradient**" problem when x is very large or very small.

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$



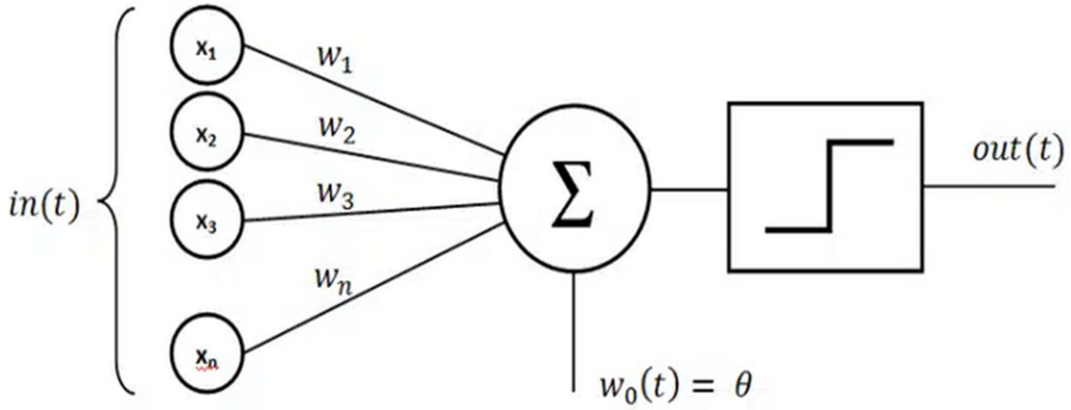
7.4. Architecture of Artificial Neural Networks

The architecture of an artificial neural network is defined by the structure of its neurons and their connectivity. It is specified by the number of inputs, outputs, nodes, and the way these nodes are interconnected and organized. the architecture of an artificial neural network is a critical factor in its performance and applicability.

The choice of architecture depends on the problem at hand, the nature of the data, and the available computational resources. Several architectures are therefore possible.

7.4.1. Single-Layer Perceptron

Developed by Rosenblatt in 1958, it is the first network to have been created. As a feedforward network, it consists of only one input layer and one output layer. All neurons in the input layer are connected to those in the output layer. They process only binary values, and the activation function is a simple threshold. It is capable of simulating simple logical operations such as AND or OR gates, which are linearly separable. Its learning is based on Hebb's reinforcement rule.



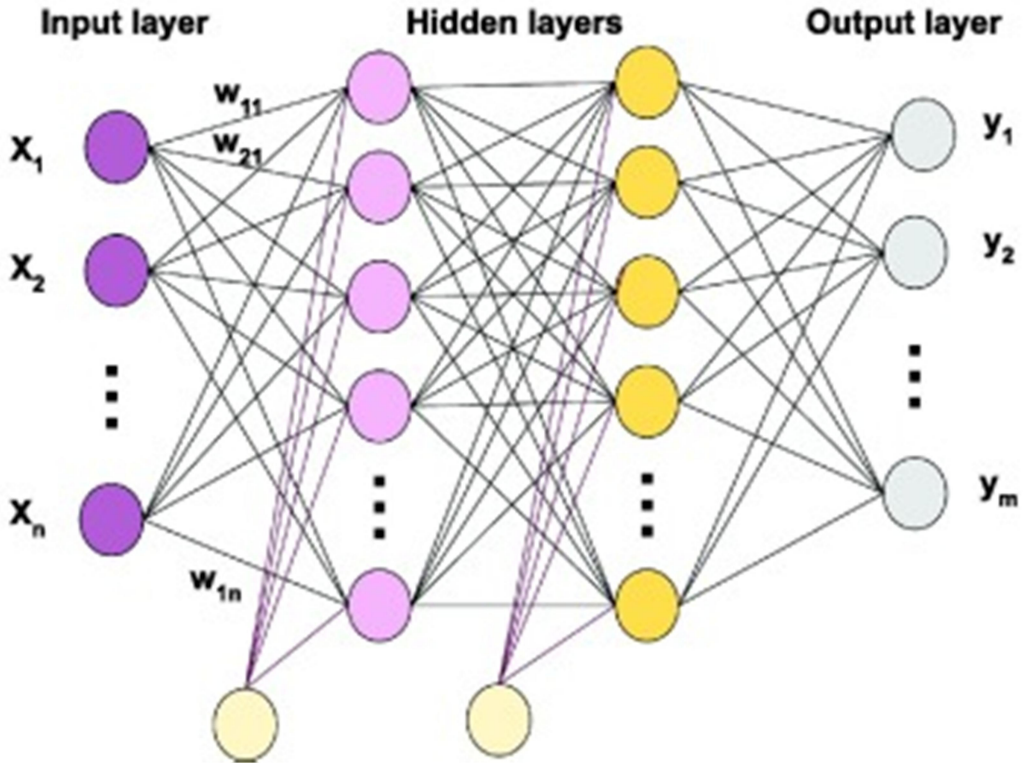
7.4.2. Multilayer Perceptron

The Multilayer Perceptron (MLP) is a multi-layered neural network composed of a sequence of layers, each of which takes its inputs from the outputs of the previous layer. It consists of an input layer, an output layer, and one or more hidden layers with numerous stacked neurons. Information flows from one layer to the next (feedforward network).

The topological characteristic of this network is that all neurons in one layer are connected to all neurons in the next layer. Each neural connection is associated with a weight w.

The multilayer perceptron neural network is composed of:

- Inputs i ranging from 1 to N,
- Outputs j ranging from 1 to M,
- Neuron l in hidden layer k, with k ranging from 1 to P,
- Bias of neuron l in hidden layer k,
- f, an activation function.



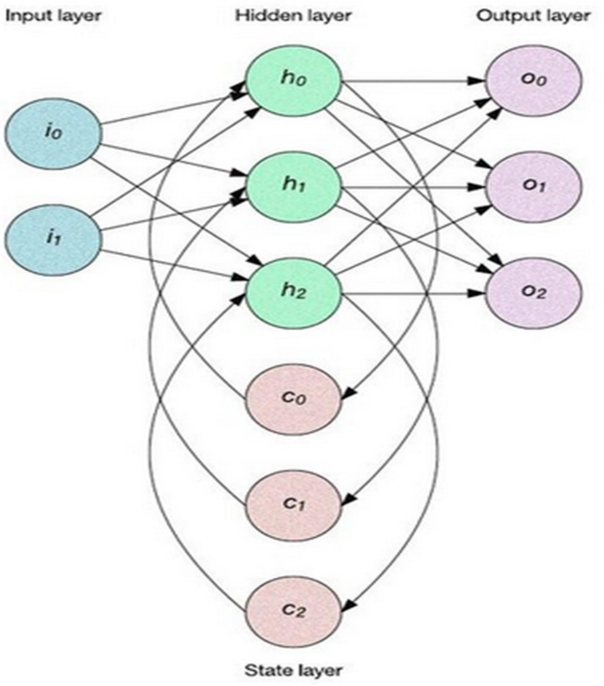
7.4.3. Recurrent neural network

A recurrent neural network (RNN) is composed of interconnected neurons that interact non-linearly, with at least one cycle present in its structure.

Recurrent neural networks are well-suited for input data of variable size. They are particularly effective for analyzing time series data.

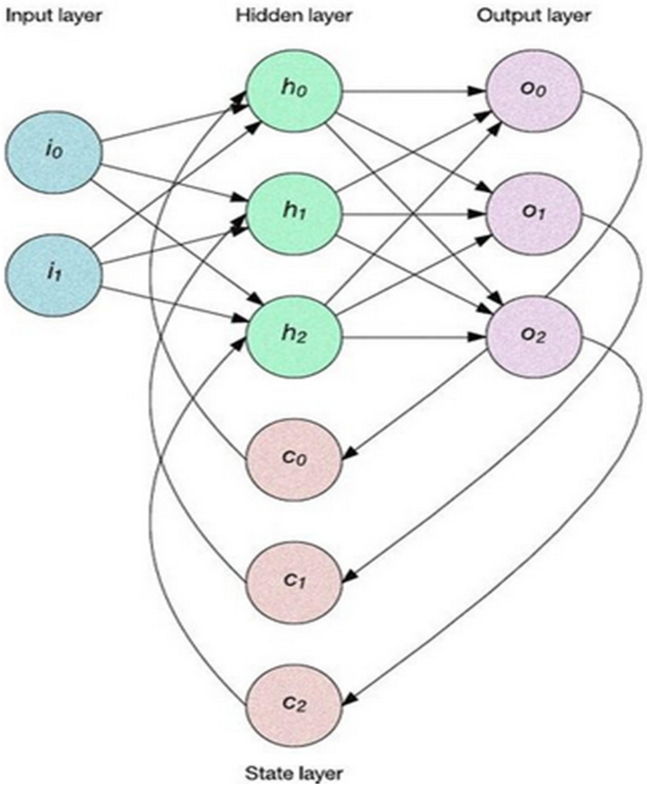
7.4.3.1. Elman Recurrent Neural Network

An Elman network is a three-layer network supplemented with a set of context units. The hidden layer is connected to these context units and has associated weights. At each time step, the input is propagated forward, and a learning rule is applied. The fixed feedback connections store a copy of the previous values of the hidden neurons in the context units (since they propagate the connections before the learning rule is applied). As a result, the network can maintain a kind of state, enabling it to perform tasks such as sequential prediction, which is beyond the capabilities of a standard multilayer perceptron.



7.4.3.2. Jordan Recurrent Neural Network

Jordan networks are similar to Elman networks. However, the context units are fed by the output layer instead of the hidden layer. The context units in a Jordan network also represent the state layer. They have a recurrent connection to themselves.



7.5. Learning

One of the most complex aspects of our brain's functioning is the learning phase. This is a phase during which certain modifications occur in the connections between neurons: some are strengthened, while others are weakened or even become inhibitory.
For artificial neural networks, learning involves calculating the parameters in such a way that the network's outputs, for the examples used during training, are as close as possible to the "desired" outputs. The learning techniques for artificial neural networks are optimization algorithms: they aim to minimize the discrepancy between the network's actual responses and the desired responses by adjusting the parameters through successive steps (called "iterations"). The output of the neural network progressively adapts better to the data as the learning process unfolds. However, the error made by the neural network at the end of the learning process is not zero.

The learning phase of artificial neural networks is a process that determines or adjusts the network's parameters to achieve a desired behavior. Several learning algorithms have been developed since the first learning rule proposed by Hebb in 1949.
Similar to the human brain, artificial neural networks cannot be directly programmed but must learn by studying and analyzing examples. There are three main learning methods:

- Supervised learning,
- Unsupervised learning,
- Reinforcement learning.

7.5.1. Supervised learning

The algorithm trains on labeled data. To perform the task, it adjusts itself until it can process the dataset to produce the expected result. A specific outcome must be defined for each input option. Supervised learning enables modifications to the system to optimize the algorithm's performance.

7.5.2. Unsupervised learning

The neural network must analyze a set of unlabeled data. A specific function informs it to what extent it deviates from or approaches the expected result. The network then adapts accordingly. The outcome of the task is not predetermined, but the system itself makes its diagnosis based on the information obtained. The system relies, among other things, on adaptive resonance theory.

7.5.3. Reinforcement learning

This is a method that involves reinforcement and penalties depending on whether the results are positive or negative. Like the human brain, which learns through trial and error, the neural network gradually learns as it processes the data it is given.

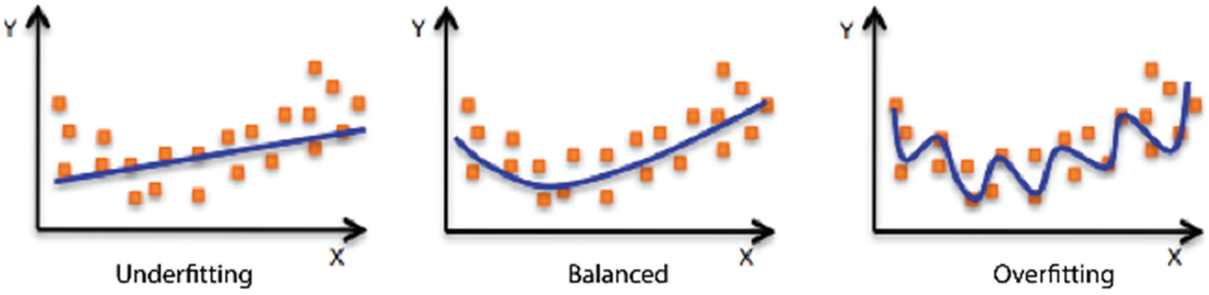
7.5.4. Notion of Generalization

The ability to generalize is one of the key motivations behind the study and development of artificial neural networks. It can be defined as the capacity to extend the knowledge acquired during training to new, previously unseen data encountered by the neural network. This is how neural networks are able to approximate a function based on only a subset of the data or to associate a new input vector, which was not part of the training set, with a given class.

7.5.5. Notion of Overfitting

Overfitting occurs when the network learns the training examples too perfectly, to the point where it becomes incapable of generalizing to new, unseen data.

The following figure clearly illustrates the trade-off between overfitting, underfitting, and good generalization.



7.6. Weight Update in a Neural Network

Updating weights in a neural network is a fundamental step in the learning process. It involves adjusting the parameters of the model (weights w) and biases (b) to minimize a cost (or loss) function that measures the error between the network's predictions and the target values. This update is typically performed using an optimization method such as Widrow-Hoff rule, Gradient Descent or its variants.

7.6.1. Widrow-Hoff rule

The **Perceptron Learning Rule** (also known as the **Widrow-Hoff rule** or **delta rule**) is a supervised learning method used to adjust the weights of a neural network based on the error between the predicted output and the desired output.
This rule was introduced by Frank Rosenblatt for the perceptron and later generalized by Bernard Widrow and Ted Hoff in the context of linear neural networks (ADALINE).
The goal of learning is to minimize the error between the predicted output $\hat{y}$ of the perceptron and the target output y. The learning rule updates the weights w and bias b based on this error.

Algorithm Steps for the Widrow-Hoff Rule

1. Initialize Parameters

- Initialize the weights w_i and bias b to small random values or zeros.

$$w_i = 0 \text{ (or small random values), b=0}$$

- Set the learning rate η (a hyperparameter positive constant that controls the step size during updates).

2. Input Data

- Let the dataset consist of m examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$

Where:

  - $x^{(i)}$ Input vector for example i.
  - $y^{(i)}$: Target output for example i.

3. Forward Pass (Prediction)

For each input $x^{(i)}$, compute the predicted output $\hat{y}^{(i)}$ using the current weights and bias:

$$z^{(i)} = \sum_{j=1}^{n} w_j * x_j^{(i)} + b$$

$$\hat{y}^{(i)} = f(z^{(i)})$$

Where:

- n: Number of inputs.
- $f(z^{(i)})$: Activation function. For the Widrow-Hoff rule, this is typically a **linear activation function** ($f(z) = z$).

4. Compute Error

Calculate the error $e^{(i)}$ between the target output $y^{(i)}$ and the predicted output $\hat{y}^{(i)}$:

$$e^{(i)} = y^{(i)} - \hat{y}^{(i)}$$

5. Update Weights and Bias

Update the weights w_j and bias b using the Delta Rule:

$$w_j = w_j + \eta * e^{(i)} * x_j^{(i)} \ (for\ all\ j = 1,2, \dots, n)$$

$$b = b + \eta * e^{(i)}$$

6. Repeat for All Examples

- Perform steps 3–5 for each example in the dataset (one epoch).
- Optionally, repeat the process for multiple epochs until convergence or until the error is minimized.

7. Check Convergence

- Stop the training if the error is below a predefined threshold or if the weights stabilize (no significant changes after an epoch).

Geometric Interpretation

In an n-dimensional space, the perceptron defines a **decision hyperplane** given by:

$$\sum_{i=1}^{n} w_i * x_i + b$$

When the weights are updated, this hyperplane is adjusted to better separate the classes in the feature space.

Convergence Conditions

The perceptron converges if the data is **linearly separable**, meaning there exists a hyperplane that can perfectly separate the classes. If the data is not linearly separable, the perceptron may not converge, and more advanced techniques (e.g., multi-layer networks or non-linear activation functions) are required.

Batch vs. Stochastic Weight Updates

The difference between **batch weight update** and **stochastic weight update** using the delta rule lies in how the weights are adjusted based on errors.

Batch Weight Update:

- The delta rule is applied to **all examples in the dataset** before updating the weights.

- Gradients are accumulated over the entire dataset, and a **single update** is performed.

For each weight w_j:

$$w_j = w_j + \eta * \frac{1}{m}\sum_{i=1}^{m} e^{(i)} * x_j^{(i)}$$

For the bias b:

$$b = b + \eta * \frac{1}{m}\sum_{i=1}^{m} e^{(i)}$$

Advantages:

- The update direction is more accurate, as it is based on the entire dataset.
- Less fluctuation in convergence.

Disadvantages:

- Slow for large datasets, as all data must be processed before updating the weights.
- Requires a lot of memory to store the entire dataset.

Stochastic Weight Update:

- The delta rule is applied to **one example at a time**.
- Weights are updated after each example.

For each weight w_j:

$$w_j = w_j + \eta * e^{(i)} * x_j^{(i)}$$

For the bias b:

$$b = b + \eta * e^{(i)}$$

Advantages:

- Fast, as weights are updated frequently.
- Can escape local minima due to gradient fluctuations.
- Suitable for large datasets.

Disadvantages:

- Gradient fluctuations can make convergence unstable.
- Less precise, as the gradient direction is calculated on a single example.

Comparison:

Aspect	Batch Update	Stochastic Update
Batch Size	Entire dataset	Single example
Update Frequency	Once per epoch	After each example
Speed	Slow (especially for large datasets)	Fast
Stability	Stable (precise gradient)	Unstable (gradient fluctuations)
Memory Requirement	High (stores entire dataset)	Low (one example at a time)
Convergence	Converges directly to the minimum	May oscillate around the minimum

7.6.2. Backpropagation algorithm

Backpropagation is a fundamental algorithm in training neural networks. It is the process of training a neural network by iteratively updating its weights and biases to minimize the loss function via **gradient descent**.

Backpropagation is an application of the **chain rule** from calculus. It works by propagating the error backward through the network, layer by layer, to compute the gradients of the loss function with respect to each weight.

7.6.2.1. What is a Loss Function?

A loss function (also known as a **cost function** or **objective function**) is a mathematical function that measures the difference between the predicted output of a model and the actual target values. The primary goal during training is to minimize this loss function, which helps the model learn to make better predictions.

1. **Quantify Error**: It evaluates how well the model's predictions match the true values.
2. **Guide Optimization**: It provides a metric for the optimization algorithm (e.g., gradient descent) to adjust the model's parameters (weights and biases) in order to improve performance.

General Form:

For a dataset with n samples, the loss function J can be expressed as:

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, \hat{y}_i)$$

Where:

- y_i: True label for the i-th sample.
- $\hat{y}_i$: Predicted output by the model for the i-th sample.

- L: A specific loss function that calculates the error for a single prediction.
- θ: Model parameters (e.g., weights and biases).

The cost function averages the loss over all samples in the dataset.

Types of loss Functions: There are many types of loss functions to choose from. Popular options include the Mean Squared Error, Squared Error, Root Mean Square Error and Sum of Square Errors. Other loss functions include Cross-Entropy, Exponential, Hellinger Distance, and the Kullback-Leibler Divergence.

7.6.2.2. What is a Partial Derivative?

In neural networks, **partial derivatives** are a fundamental concept used in the process of training the model through an optimization algorithm such as **gradient descent**. They allow us to compute how much each parameter (e.g., weights and biases) affects the overall cost function, enabling the network to learn by adjusting these parameters to minimize the error.

In mathematics, a derivative represents the rate of change of a function at a single point.

A **partial derivative** measures the rate of change of a multivariable function with respect to one of its variables while keeping all other variables constant.

In the context of neural networks, the cost function $J(\theta)$ depends on many parameters (weights w and biases b), and partial derivatives help determine how sensitive the cost function is to small changes in each parameter.

For example:

$$\frac{\partial \mathrm{J}}{\partial w} \; and \; \frac{\partial \mathrm{J}}{\partial b}$$

Why is a partial derivative used?

To update the weights of a neural network we need to know how much a change in a specific weight affects the total error. That is, we want to find the rate of change between two variables: a specific weight and the total error. Within neural networks, the **partial derivative** is often described as a **gradient**.

Partial derivatives are used primarily to compute the gradients of the loss function with respect to the model's parameters (weights and biases). These gradients guide the optimization process, enabling the network to learn by adjusting its parameters to minimize the error.

7.6.2.3. Gradient descent

Gradient descent is a fundamental optimization algorithm used in training neural networks. It works by iteratively adjusting the weights and biases of the network to minimize the loss function.

The goal of training a neural network is to find the optimal values of its parameters (weights W and biases b) that minimize the loss function L.

Gradient descent uses the **gradient** (partial derivatives) of the loss function with respect to each parameter to determine the direction and magnitude of the update. Specifically:

$$\theta = \theta - \eta * \nabla_\theta L$$

Where:

- θ: A parameter (e.g., weight or bias).
- η: Learning rate, controlling the step size of each update.
- $\nabla_\theta L$: Gradient of the loss function with respect to θ.

The negative sign ensures that the parameters are updated in the direction of decreasing loss.

Steps of Gradient Descent

Step 1: Compute Gradients

Using backpropagation, the gradients of the loss function with respect to all parameters (weights and biases) are computed:

- For weights: $\frac{\partial L}{\partial W^{(l)}}$
- For biases: $\frac{\partial L}{\partial b^{(l)}}$

These gradients indicate how much the loss changes with respect to small changes in the parameters.

Step 2: Update Parameters

Once the gradients are computed, the parameters are updated as follows:

$$W^{(l)} = W^{(l)} - \eta * \frac{\partial L}{\partial W^{(l)}}$$

$$b^{(l)} = b^{(l)} - \eta * \frac{\partial L}{\partial b^{(l)}}$$

This process adjusts the weights and biases to reduce the loss.

Step 3: Repeat

The forward pass, backward pass (gradient computation), and parameter update steps are repeated for multiple iterations (epochs) until the loss converges or reaches a predefined threshold.

Learning Rate (η)

The learning rate determines the step size of each parameter update. Choosing an appropriate learning rate is crucial:

- **Too Large**: May cause the loss to oscillate or diverge.
- **Too Small**: May result in slow convergence or getting stuck in local minima.

7.6.3. Weight initialization

Weight initialization is a crucial step in training neural networks, as it sets the starting point for the optimization process and significantly impacts the network's ability to learn effectively. Proper initialization helps avoid common issues such as **vanishing gradients, exploding gradients, and symmetry breaking**, all of which can hinder the training process. When weights are initialized poorly, the network may struggle to converge or may converge very slowly, leading to suboptimal performance. Therefore, choosing the right initialization strategy is essential for ensuring stable and efficient training.

Why is Weight Initialization Important?

- **Symmetry Breaking:** If all weights in a layer are initialized to the same value (e.g., zero), each neuron in that layer will compute the same output during forward propagation and receive the same gradient during backpropagation. This results in all neurons updating identically, effectively making them redundant. Proper initialization breaks this symmetry, allowing neurons to learn different features.

- **Vanishing Gradients:** If weights are initialized too small, the gradients during backpropagation can become extremely small as they propagate through the network. This slows down or even halts learning, especially in deep networks.

- **Exploding Gradients:** If weights are initialized too large, the gradients can grow exponentially during backpropagation, leading to unstable updates and divergence.

- **Faster Convergence:** Good initialization helps the network start in a favorable state, enabling faster convergence to a good solution.

Common Weight Initialization Techniques

1. Zero Initialization

- All weights are initialized to zero.
- **Problem:** Leads to symmetry breaking, as all neurons in a layer behave identically.
- **Not recommended** for neural networks.

2. Random Initialization

- Weights are initialized with small random values, typically sampled from a uniform or normal distribution.
- **Advantage:** Breaks symmetry, allowing neurons to learn different features.
- **Disadvantage:** If the weights are too small or too large, it can lead to vanishing or exploding gradients.

3. Xavier/Glorot Initialization

- Designed for activation functions like **sigmoid** or **tanh**.
- Scales the weights based on the number of input and output neurons to ensure that the variance of the outputs is approximately equal to the variance of the inputs.
- Helps maintain stable gradients throughout the network.
- **Formula:**
  - For uniform distribution:

$$W \sim u\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right)$$

  - For normal distribution:

$$W \sim u\left(0, \sqrt{\frac{6}{n_{in}+n_{out}}}\right)$$

4. He Initialization

- Designed for activation functions like **ReLU** (Rectified Linear Unit) and its variants (e.g., Leaky ReLU, Parametric ReLU).
- Scales the weights based on the number of input neurons only, as ReLU activations zero out negative inputs, which can lead to uneven variance in the outputs.
- **Formula:**
  - For uniform distribution:

$$W \sim u\left(-\sqrt{\frac{6}{n_{in}}}, \sqrt{\frac{6}{n_{in}}}\right)$$

  - For normal distribution:

$$W \sim u\left(0, \sqrt{\frac{2}{n_{in}}}\right)$$

7.7. Advantages and Limitations of Neural Networks

7.7.1. Advantages of Neural Networks

Neural networks offer the following advantages:

- **Empirical Knowledge**: Learning from examples (empirical learning method) is relatively straightforward and often yields better results compared to other machine learning techniques.
- **Fewer Adjustable Parameters**: They generally require fewer adjustable parameters to achieve a nonlinear model of a given accuracy.
- **Graceful Degradation**: The responses provided by neural networks degrade gradually in the presence of noisy inputs. They generalize well from the knowledge present in the training dataset and are less sensitive to disturbances than symbolic systems. Working with a 'numeric' representation of knowledge makes neural networks better suited for handling quantitative data (continuous values). They are less vulnerable to approximate data and the presence of incorrect data in the training dataset.
- **Massive Parallelism**: Neural networks consist of a set of information processing units that can operate in parallel. Although most implementations of neural networks are sequential simulations, it is possible to create (software or hardware) implementations that exploit the ability to activate units simultaneously. Most neural network implementations can be easily converted from a sequential version to a parallel one.

7.7.2. Limitations of Neural Networks

Neural networks also have several disadvantages, such as:

- **Architecture and Parameters**: There is no automatic method to choose the best possible architecture for a given problem. It is quite challenging to determine the optimal network topology and the right parameters for the learning algorithm. The evolution of the learning process is heavily influenced by these two factors (network architecture and parameter settings) and depends significantly on the type of problem being addressed. Simply changing the training dataset may require reconfiguring the entire network.
- **Initialization and Encoding**: Connectionist learning algorithms are generally highly dependent on the initial state of the network (random initialization of

weights) and the configuration of the training dataset. Poor choices in weight initialization, data encoding methods, or even the order of the data can hinder learning or cause issues with the network's convergence to a good solution.

- **Black Box**: The knowledge acquired by the network is encoded in the values of the synaptic weights and the way the units are interconnected. It is very difficult for humans to interpret this directly. Neural networks are black boxes, where knowledge remains enclosed and unintelligible to the user or expert. A network cannot explain the reasoning that led it to a specific solution.

- **Theoretical Knowledge**: Classical neural networks do not allow leveraging theoretical knowledge available about the problem domain. Like decision trees, they are dedicated to handling empirical knowledge. A simplistic way to incorporate theoretical knowledge is to convert rules into examples (prototypes). However, this method does not guarantee that these examples will be well-represented in the network's knowledge after training, as we are forced to go through a learning phase where empirical knowledge and theoretically encoded examples are mixed indiscriminately.

Exercises

Exercise 1

Given two data objects: **A(7, 30, 0, 9, 87)** and **B(4, 67, 2, 54, 5).**
Calculate:

1) Euclidean distance
2) Manhattan distance
3) Chebyshev distance
4) Cosine distance

Exercise 2

Use Euclidean distance, Manhattan distance, Chebyshev distance, Cosine distance for each prediction.

Flower	Petal Length (x1)	Petal Width (x2)	Species
A	1.5	0.3	Setosa
B	4.5	1.5	Versicolor
C	1.4	0.2	
D	4.3	1.3	

1. Calculate the Chebyshev distance between points C and A, then between C and B.
2. Calculate the Chebyshev distance between points D and A, then between D and B.
3. Use the k-NN algorithm with k=1 to predict the species of flowers C and D based on the calculated distances.

Exercise 3

For a neural network composed of:

- **Input layer**: 2 neurons (features x1,x2).
- **Hidden layer**: 2 neurons (**sigmoid** activation).
- **Output layer**: 1 neuron (**sigmoid** activation).
- **Loss function**: Mean Squared Error (MSE).

Given:

- **Input**: x=[0.6, 0.2].
- **True output**: y_{true}=0.9.
- **Initial weights**:

$$W_1 = \begin{bmatrix} 0.1 & -0.3 \\ 0.4 & 0.2 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 0.5 \\ -0.1 \end{bmatrix}$$

Biases: Assume all zeros for simplicity.

Tasks:

1. **Forward Pass**: Compute the predicted output $\hat{y}$.
2. **Backward Pass**:
   - Calculate the loss gradient w.r.t. output ($\partial L/\partial \hat{y}$).
   - Propagate errors backward to update W_2 and W_1 (use learning rate η=0.1).

General Conclusion

General Conclusion

Artificial Intelligence has evolved from a theoretical concept to a driving force behind technological innovation, influencing nearly every aspect of modern life. Throughout this lecture notes, we have explored the foundational principles that make AI possible—from logical reasoning and search algorithms to game theory, metaheuristics, and machine learning. Each of these components plays a crucial role in developing intelligent systems capable of solving complex problems, optimizing decisions, and even mimicking human cognition.

As AI continues to advance, its applications are expanding into areas such as healthcare, finance, robotics, and climate science, offering unprecedented opportunities for automation, efficiency, and discovery. However, with these advancements come significant challenges, including ethical concerns, bias in algorithms, and the societal impact of automation. The future of AI will depend not only on technological progress but also on responsible development, ensuring that these systems are transparent, fair, and aligned with human values.

This lecture notes have provided a structured pathway to understanding AI's core mechanisms, equipping readers with the knowledge to engage critically with the field. Whether you aim to pursue further research, develop AI applications, or simply comprehend the technology shaping our world, the principles discussed here serve as a vital foundation. The journey of AI is far from over, and its next breakthroughs will be driven by those who can harness its potential while navigating its challenges with wisdom and foresight.

As we stand at the forefront of this transformative era, one thing is certain: artificial intelligence will continue to redefine what is possible, and the choices we make today will shape its impact for generations to come.

Bibliography

Bibliography

1. Baum, H. Introduction to artificial intelligence. AG Printing & Publishing, 2023.
2. Bishop, C. M. Pattern Recognition and Machine Learning. Springer, 2006.
3. Castaño, A. P. Practical Artificial Intelligence: Machine Learning, Bots, and Agent Solutions Using C. Apress, 2018.
4. Edelkamp, S., & Schrödl, S. Heuristic Search: Theory and Applications. Morgan Kaufmann, 2012.
5. Eiben, A. E., & Smith, J. E. Introduction to Evolutionary Computing (2nd ed.). Springer, 2015.
6. Fudenberg, D., & Tirole, J. Game Theory. MIT Press, 1991.
7. Goldberg, D. E. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley, 1989.
8. Hastie, T., Tibshirani, R., & Friedman, J. The Elements of Statistical Learning (2nd ed.). Springer, 2009.
9. Koza, J. R. Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, 1992.
10. Leyton-Brown, K., & Shoham, Y. Essentials of Game Theory: A Concise, Multidisciplinary Introduction. Morgan & Claypool, 2008.
11. Mitchell, M. An Introduction to Genetic Algorithms. MIT Press, 1998.
12. Murphy, K. P. Probabilistic Machine Learning: An Introduction. MIT Press, 2022.
13. Nils J. Nilson. Principles of Artificial Intelligence, Springer-Verlag Berlin Heidelberg, 1982.
14. Myerson, R. B. Game Theory: Analysis of Conflict. Harvard University Press, 1991.
15. Pearl, J. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, 1984.
16. Nils J. Nilson. Essentials of Artificial Intelligence, Morgan Kaufmann, 1993.
17. Nils Nilson. Artificial Intelligence: A new synthesis, Elsevier, 1997.
18. Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach, Ed. Pearson, 2016