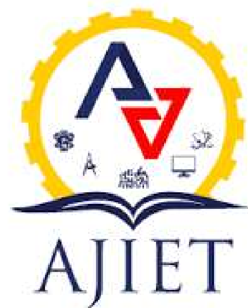
VISVESVARAYA TECHNOLOGICAL UNIVERSITY

BELGAUM

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING (Subject Code: 21CS54)

V-SEMESTER



A J INSTITUTE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

(A unit of Laxmi Memorial Education Trust. (R))

NH - 66, Kottara Chowki, Kodical Cross - 575006

VISION OF THE INSTITUTE

"To produce top-quality engineers who are groomed for attaining excellence in their profession and competitive enough to help in the growth of nation and global society."

MISSION OF THE INSTITUTE

M1: To offer affordable high-quality graduate program in engineering with value education and make the students socially responsible.

M2: To support and enhance the institutional environment to attain research excellence in both faculty and students and to inspire them to push the boundaries of knowledge base.

M3: To identify the common areas of interest amongst the individuals for the effective industry- institute partnership in a sustainable way by systematically working together.

M4: To promote the entrepreneurial attitude and inculcate innovative ideas among the engineering professionals.

VISION OF THE DEPARTMENT

To be a centre of excellence in Information Science & Engineering education, research and training to meet the growing needs of the industry and society.

MISSION OF THE DEPARTMENT

M1: To impart theoretical and practical knowledge through the concepts and technologies in Information Science and Engineering

M2: To foster research, collaboration and higher education with premier institutions and industries.

M3: Promote innovation and entrepreneurship to fulfil the needs of the society and industry

Program Educational Objectives (PEOs)

After 4 years of graduation, graduates will be able to

PEO1: Analyze, design and implement solutions to the real-world problems in the field of Information Science and Engineering with multidisciplinary setup.

PEO2: Keep abreast with the technology, innovation and pursue higher education with high standards of

social and professional ethics.

PEO3: Develop professional and entrepreneurship skills to work effectively as an individual and in a team to meet the ever-changing goals of the organization.

Program Outcomes (POs)

PO1: Engineering Knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem Analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/Development of Solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct Investigations of Complex Problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern Tool Usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6: The Engineer and Society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and Sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and Team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project Management and Finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-Long Learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

[As per Choice Based Credit System (CBCS) scheme] (Effective from the academic year 2022-23) SEMESTER – V

Course Code	21CS54	CIE Marks	50
Teaching Hours/Week	3:0:0:0	SEE Marks	50
Total Hour of Pedagogy	40	Exam Hours	3 Hrs.

CREDITS – 03

Course Objectives

- CLO 1. Gain a historical perspective of AI and its foundations
- CLO 2. Become familiar with basic principles of AI toward problem solving
- CLO 3. Familiarize with the basics of Machine Learning & Machine Learning process, basics of Decision Tree, and probability learning
- CLO 4. Understand the working of Artificial Neural Networks and basic concepts of clustering algorithms

Course Outcomes

- CO 1. Apply the knowledge of searching and reasoning techniques for different applications.
- CO 2. Have a good understanding of machine leaning in relation to other fields and fundamental issues and challenges of machine learning.
- CO 3. Apply the knowledge of classification algorithms on various dataset and compare results
- CO 4. Model the neuron and Neural Network, and to analyze ANN learning and its applications.
- CO 5. Identifying the suitable clustering algorithm for different pattern

Teaching-Learning Process (General Instructions)

These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes.

1. Lecturer method (L) need not to be only a traditional lecture method, but alternative effective teaching methods could be adopted to attain the outcomes.
2. Use of Video/Animation to explain functioning of various concepts.
3. Encourage collaborative (Group Learning) Learning in the class.
4. Ask at least three HOT (Higher order Thinking) questions in the class, which promotes critical thinking.

AJIET

5. **5.** Adopt Problem Based Learning (PBL), which fosters students' Analytical skills, develop design thinking skills such as the ability to design, evaluate, generalize, and analyze information rather than simply recall it.
6. **6.** Introduce Topics in manifold representations.
7. **7.** Show the different ways to solve the same problem with different circuits/logic and encourage the students to come up with their own creative ways to solve them.
8. **8.** Discuss how every concept can be applied to the real world - and when that's possible, it helps improve the students' understanding.

MODULE 1
Introduction: What is AI? Foundations and History of AI Problem‐solving: Problem‐solving agents, Example problems, Searching for Solutions, Uninformed Search Strategies: Breadth First search, Depth First Search, Textbook 1: Chapter 1- 1.1, 1.2, 1.3 Textbook 1: Chapter 3- 3.1, 3.2, 3.3, 3.4.1, 3.4.3
MODULE 2
Informed Search Strategies: Greedy best-first search, A*search, Heuristic functions. Introduction to Machine Learning , Understanding Data Textbook 1: Chapter 3 - 3.5, 3.5.1, 3.5.2, 3.6 Textbook 2: Chapter 1 and 2
MODULE 3
Basics of Learning theory Similarity Based Learning Regression Analysis Textbook 2: Chapter 3 - 3.1 to 3.4, Chapter 4, chapter 5.1 to 5.4
MODULE 4
Decision Tree learning Bayesian Learning Textbook 2: Chapter 6 and 8
MODULE 5
Artificial neural Network Clustering Algorithms Textbook 2: Chapter 10 and 13

	Assessment Details (both CIE and SEE)
	• The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks). A student shall be deemed to have satisfied the academic requirements and earned

the credits allotted to each subject/ course if the student secures not less than 35% (18 Marks out of 50) in the semester-end examination (SEE), and a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together

Continuous Internal Evaluation:

- Three Unit Tests each of 20 Marks (duration 01 hour)
  1. First test at the end of 5th week of the semester
  2. Second test at the end of the 10th week of the semester
  3. Third test at the end of the 15th week of the semester
- Two assignments each of 10 Marks
  4. First assignment at the end of 4th week of the semester
  5. Second assignment at the end of 9th week of the semester
  - Group discussion/Seminar/quiz any one of three suitably planned to attain the COs and POs for 20 Marks (duration 01 hours) OR Suitable Programming experiments based on the syllabus contents can be given to the students to submit the same as laboratory work(for example; Implementation of concept learning, implementation of decision tree learning algorithm for suitable data set, etc…)

  6. At the end of the 13th week of the semester.

The sum of three tests, two assignments, and quiz/seminar/group discussion will be out of 100 marks and will be scaled down to 50 marks (to have less stressed CIE, the portion of the syllabus should not be common /repeated for any of the methods of the CIE. Each method of CIE should have a different syllabus portion of the course).

CIE methods /question paper has to be designed to attain the different levels of Bloom's taxonomy as per the outcome defined for the course.

Semester End Examination:

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the subject (duration 03 hours)

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), should have a mix of topics under that module.

- The students have to answer 5 full questions, selecting one full question from each module

Textbooks:

1. Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson,2015

	2. S. Sridhar, M Vijayalakshmi "Machine Learning". Oxford ,2021 **Reference Books:** 1. Elaine Rich, Kevin Knight, Artificial Intelligence, 3rdedition, Tata McGraw Hill,2013 2. George F Lugar, Artificial Intelligence Structure and strategies for complex, Pearson Education, 5th Edition, 2011 3. Tom Michel, Machine Learning, McGrawHill Publication.
	Weblinks and Video Lectures (e-Resources): 1. https://www.kdnuggets.com/2019/11/10-free-must-read-books-ai.html 2. https://www.udacity.com/course/knowledge-based-ai-cognitive-systems--ud409 3. https://nptel.ac.in/courses/106/105/106105077/ 4. https://www.javatpoint.com/history-of-artificial-intelligence 5. https://www.tutorialandexample.com/problem-solving-in-artificial-intelligence 6. https://techvidvan.com/tutorials/ai-heuristic-search/ 7. https://www.analyticsvidhya.com/machine-learning/ 8. https://www.javatpoint.com/decision-tree-induction 9. https://www.hackerearth.com/practice/machine-learning/machine-learning-algorithms/mldecision-tree/tutorial/ 10. https://www.javatpoint.com/unsupervised-artificial-neural-networks **Activity Based Learning (Suggested Activities in Class)/ Practical Based learning** 1. Role play for strategies– DFS & BFS, Outlier detection in Banking and insurance transaction for identifying fraudulent behaviour etc. Uncertainty and reasoning Problem- reliability of sensor used to detect pedestrians using Bayes Rule

AJIET

MODULE 1

Introduction: What is AI? Foundations and History of AI

Problem‐solving: Problem‐solving agents, Example problems, Searching for Solutions, Uninformed Search Strategies: Breadth First search, Depth First Search,

Textbook 1: Chapter 1- 1.1, 1.2, 1.3 Textbook 1: Chapter 3- 3.1, 3.2, 3.3, 3.4.1, 3.4.3

Chapter 1.

Introduction

In which we try to explain why we consider artificial intelligence to be a subject most worthy of study, and in which we try to decide what exactly it is, this being a good thing to decide before embarking.

Humankind has given itself the scientific name **homo sapiens**--man the wise--because our mental capacities are so important to our everyday lives and our sense of self. The field of **artificial intelligence**, or AI, attempts to understand intelligent entities. Thus, one reason to study it is to learn more about ourselves. But unlike philosophy and psychology, which are also concerned with intelligence, AI strives to *build* intelligent entities as well as understand them. Another reason to study AI is that these constructed intelligent entities are interesting and useful in their own right. AI has produced many significant and impressive products even at this early stage in its development. Although no one can predict the future in detail, it is clear that computers with human-level intelligence (or better) would have a huge impact on our everyday lives and on the future course of civilization.

AI addresses one of the ultimate puzzles. How is it possible for a slow, tiny brain{brain}, whether biological or electronic, to perceive, understand, predict, and manipulate a world far larger and more complicated than itself? How do we go about making something with those properties? These are hard questions, but unlike the search for faster-than-light travel or an antigravity device, the researcher in AI has solid evidence that the quest is possible. All the researcher has to do is look in the mirror to see an example of an intelligent system.

AI is one of the newest disciplines. It was formally initiated in 1956, when the name was coined, although at that point work had been under way for about five years. Along with modern genetics, it is regularly cited as the ``field I would most like to be in'' by scientists in other disciplines. A student in physics might reasonably feel that all the good ideas have already been taken by Galileo, Newton, Einstein, and the rest, and that it takes many years of study before one can contribute new ideas. AI, on the other hand, still has openings for a full-time Einstein.

The study of intelligence is also one of the oldest disciplines. For over 2000 years, philosophers have tried to understand how seeing, learning, remembering, and reasoning could, or should, be done. The

advent of usable computers in the early 1950s turned the learned but armchair speculation concerning these mental faculties into a real experimental and theoretical discipline. Many felt that the new ``Electronic Super-Brains" had unlimited potential for intelligence. ``Faster Than Einstein" was a typical headline. But as well as providing a vehicle for creating artificially intelligent entities, the computer provides a tool for testing theories of intelligence, and many theories failed to withstand the test--a case of ``out of the armchair, into the fire." AI has turned out to be more difficult than many at first imagined, and modern ideas are much richer, more subtle, and more interesting as a result.

AI currently encompasses a huge variety of subfields, from general-purpose areas such as perception and logical reasoning, to specific tasks such as playing chess, proving mathematical theorems, writing poetry{poetry}, and diagnosing diseases. Often, scientists in other fields move gradually into artificial intelligence, where they find the tools and vocabulary to systematize and automate the intellectual tasks on which they have been working all their lives. Similarly, workers in AI can choose to apply their methods to any area of human intellectual endeavor. In this sense, it is truly a universal field.

What is AI?

We have now explained why AI is exciting, but we have not said what it *is*. We could just say, ``Well, it has to do with smart programs, so let's get on and write some." But the history of science shows that it is helpful to aim at the right goals. Early alchemists, looking for a potion for eternal life and a method to turn lead into gold, were probably off on the wrong foot. Only when the aim changed, to that of finding explicit theories that gave accurate predictions of the terrestrial world, in the same way that early astronomy predicted the apparent motions of the stars and planets, could the scientific method emerge and productive science take place. Definitions of artificial intelligence according to eight recent textbooks are shown in the table below. These definitions vary along two main dimensions. The ones on top are concerned with *thought processes* and *reasoning*, whereas the ones on the bottom address *behavior*. Also, the definitions on the left measure success in terms of *human* performance, whereas the ones on the right measure against an *ideal* concept of intelligence, which we will call **rationality**. A system is rational if it does the right thing.

``The exciting new effort to make computers think ... *machines with minds*, in the full and literal sense" (Haugeland, 1985) ``The automation of 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) ``The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)

``The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) ``The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)	``A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990) ``The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

This gives us four possible goals to pursue in artificial intelligence:

Systems that think like humans.	Systems that think rationally.
Systems that act like humans	Systems that act rationally

Historically, all four approaches have been followed. As one might expect, a tension exists between approaches centered around humans and approaches centered around rationality. (We should point out that by distinguishing between *human* and *rational* behavior, we are not suggesting that humans are necessarily ``irrational" in the sense of ``emotionally unstable" or ``insane." One merely need note that we often make mistakes; we are not all chess grandmasters even though we may know all the rules of chess; and unfortunately, not everyone gets an A on the exam. Some systematic errors in human reasoning are cataloged by Kahneman *et al.*.) A human-centered approach must be an empirical science, involving hypothesis and experimental confirmation. A rationalist approach involves a combination of mathematics and engineering. People in each group sometimes cast aspersions on work done in the other groups, but the truth is that each direction has yielded valuable insights. Let us look at each in more detail.

Acting humanly: The Turing Test approach

The **Turing Test**, proposed by Alan Turing (Turing, 1950), was designed to provide a satisfactory operational definition of intelligence. Turing defined intelligent behavior as the ability to achieve human-level performance in all cognitive tasks, sufficient to fool an interrogator. Roughly speaking, the test he proposed is that the computer should be interrogated by a human via a teletype, and passes the test if the interrogator cannot tell if there is a computer or a human at the other end. Chapter 26 discusses the details of the test, and whether or not a computer is really intelligent if it passes. For now, programming a computer to pass the test provides plenty to work on. The computer would need to possess the following capabilities:

- **natural language processing** to enable it to communicate successfully in English (or some other human language);

- **knowledge representation** to store information provided before or during the interrogation;
- **automated reasoning** to use the stored information to answer questions and to draw new conclusions;
- **machine learning** to adapt to new circumstances and to detect and extrapolate patterns.

Turing's test deliberately avoided direct physical interaction between the interrogator and the computer, because *physical* simulation of a person is unnecessary for intelligence. However, the so-called **total Turing Test**includes a video signal so that the interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects ``through the hatch.'' To pass the total Turing Test, the computer will need

- **computer vision** to perceive objects, and
- **robotics** to move them about.

Within AI, there has not been a big effort to try to pass the Turing test. The issue of acting like a human comes up primarily when AI programs have to interact with people, as when an expert system explains how it came to its diagnosis, or a natural language processing system has a dialogue with a user. These programs must behave according to certain normal conventions of human interaction in order to make themselves understood. The underlying representation and reasoning in such a system may or may not be based on a human model.

Thinking humanly: The cognitive modelling approach

If we are going to say that a given program thinks like a human, we must have some way of determining how humans think. We need to get *inside* the actual workings of human minds. There are two ways to do this: through introspection--trying to catch our own thoughts as they go by--or through psychological experiments. Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program. If the program's input/output and timing behavior matches human behavior, that is evidence that some of the program's mechanisms may also be operating in humans. For example, Newell and Simon, who developed GPS, the ``General Problem Solver'' (Newell and Simon, 1961), were not content to have their program correctly solve problems. They were more concerned with comparing the trace of its reasoning steps to traces of human subjects solving the same problems. This is in contrast to other researchers of the same time (such as Wang (1960)), who were concerned with getting the right answers regardless of how humans might do it. The interdisciplinary field of **cognitive science** brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind. Although cognitive science is a fascinating field in itself, we are not going to be discussing it all that much in this book. We will occasionally comment on

similarities or differences between AI techniques and human cognition. Real cognitive science, however, is necessarily based on experimental investigation of actual humans or animals, and we assume that the reader only has access to a computer for experimentation. We will simply note that AI and cognitive science continue to fertilize each other, especially in the areas of vision, natural language, and learning. The history of psychological theories of cognition is briefly covered on page 12.

Thinking rationally: The laws of thought approach

The Greek philosopher Aristotle was one of the first to attempt to codify ``right thinking,'' that is, irrefutable reasoning processes. His famous **syllogisms** provided patterns for argument structures that always gave correct conclusions given correct premises. For example, ``Socrates is a man; all men are mortal; therefore Socrates is mortal.'' These laws of thought were supposed to govern the operation of the mind, and initiated the field of **logic**.

The development of formal logic in the late nineteenth and early twentieth centuries, which we describe in more detail in Chapter 6, provided a precise notation for statements about all kinds of things in the world and the relations between them. (Contrast this with ordinary arithmetic notation, which provides mainly for equality and inequality statements about numbers.) By 1965, programs existed that could, given enough time and memory, take a description of a problem in logical notation and find the solution to the problem, if one exists. (If there is no solution, the program might never stop looking for it.) The so-called **logicist** tradition within artificial intelligence hopes to build on such programs to create intelligent systems.

There are two main obstacles to this approach. First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain. Second, there is a big difference between being able to solve a problem ``in principle'' and doing so in practice. Even problems with just a few dozen facts can exhaust the computational resources of any computer unless it has some guidance as to which reasoning steps to try first. Although both of these obstacles apply to *any* attempt to build computational reasoning systems, they appeared first in the logicist tradition because the power of the representation and reasoning systems are well-defined and fairly well understood.

Acting rationally: The rational agent approach

Acting rationally means acting so as to achieve one's goals, given one's beliefs. An **agent** is just something that perceives and acts. (This may be an unusual use of the word, but you will get used to it.) In this approach, AI is viewed as the study and construction of rational agents.

In the ``laws of thought'' approach to AI, the whole emphasis was on correct inferences. Making correct inferences is sometimes *part* of being a rational agent, because one way to act rationally is to reason logically to the conclusion that a given action will achieve one's goals, and then to act on that

conclusion. On the other hand, correct inference is not *all* of rationality, because there are often situations where there is no provably correct thing to do, yet something must still be done. There are also ways of acting rationally that cannot be reasonably said to involve inference. For example, pulling one's hand off of a hot stove is a reflex action that is more successful than a slower action taken after careful deliberation.

All the ``cognitive skills'' needed for the Turing Test are there to allow rational actions. Thus, we need the ability to represent knowledge and reason with it because this enables us to reach good decisions in a wide variety of situations. We need to be able to generate comprehensible sentences in natural language because saying those sentences helps us get by in a complex society. We need learning not just for erudition, but because having a better idea of how the world works enables us to generate more effective strategies for dealing with it. We need visual perception not just because seeing is fun, but in order to get a better idea of what an action might achieve--for example, being able to see a tasty morsel helps one to move toward it.

The study of AI as rational agent design therefore has two advantages. First, it is more general than the ``laws of thought'' approach, because correct inference is only a useful mechanism for achieving rationality, and not a necessary one. Second, it is more amenable to scientific development than approaches based on human behavior or human thought, because the standard of rationality is clearly defined and completely general. Human behavior, on the other hand, is well-adapted for one specific environment and is the product, in part, of a complicated and largely unknown evolutionary process that still may be far from achieving perfection. *This book will therefore concentrate on general principles of rational agents, and on components for constructing them.* We will see that despite the apparent simplicity with which the problem can be stated, an enormous variety of issues come up when we try to solve it. Chapter 2 outlines some of these issues in more detail. One important point to keep in mind: we will see before too long that achieving perfect rationality--always doing the right thing--is not possible in complicated environments. The computational demands are just too high. However, for most of the book, we will adopt the working hypothesis that understanding perfect decision making is a good place to start. It simplifies the problem and provides the appropriate setting for most of the foundational material in the field. Chapters 5 and 17 deal explicitly with the issue of **limited rationality**--acting appropriately when there is not enough time to do all the computations one might like.

AI Foundations

AI prehistory	
Philosophy	logic, methods of reasoning
	mind as physical system
	foundations of learning, language, rationality
Mathematics	formal representation and proof
	algorithms
	computation, (un)decidability, (in)tractability
	probability
Psychology	adaptation
	phenomena of perception and motor control
	experimental techniques (psychophysics, etc.)
Linguistics	knowledge representation
	grammar
Neuroscience	physical substrate for mental activity
Control theory	homeostatic systems, stability
	simple optimal agent designs

History of AI

Potted history of AI	
1943	McCulloch & Pitts: Boolean circuit model of brain
1950	Turing's "Computing Machinery and Intelligence"
1952–69	Look, Ma, no hands!
1950s	Early AI programs, including Samuel's checkers program,
	Newell & Simon's Logic Theorist, Gelernter's Geometry Engine
1956	Dartmouth meeting: "Artificial Intelligence" adopted
1965	Robinson's complete algorithm for logical reasoning
1966–74	AI discovers computational complexity
	Neural network research almost disappears
1969–79	Early development of knowledge-based systems
1980–88	Expert systems industry booms
1988–93	Expert systems industry busts: "AI Winter"
1985–95	Neural networks return to popularity
1988–	Resurgence of probabilistic and decision-theoretic methods
	Rapid increase in technical depth of mainstream AI
	"Nouvelle AI": ALife, GAs, soft computing

Summary

This chapter defines AI and establishes the cultural background against which it has developed. Some of the important points are as follows:

- Different people think of AI differently. Two important questions to ask are: Are you concerned with thinking or behavior? Do you want to model humans, or work from an ideal standard?

- In this book, we adopt the view that intelligence is concerned mainly with **rational action**. Ideally, an **intelligent agent** takes the best possible action in a situation. We will study the problem of building agents that are intelligent in this sense.
- Philosophers (going back to 400 B.C.) made AI conceivable by considering the ideas that the mind is in some ways like a machine, that it operates on knowledge encoded in some internal language, and that thought can be used to help arrive at the right actions to take.
- Mathematicians provided the tools to manipulate statements of logical certainty as well as uncertain, probabilistic statements. They also set the groundwork for reasoning about algorithms.
- Psychologists strengthened the idea that humans and other animals can be considered information processing machines. Linguists showed that language use fits into this model.
- Computer engineering provided the artifact that makes AI applications possible. AI programs tend to be large, and they could not work without the great advances in speed and memory that the computer industry has provided.
- The history of AI has had cycles of success, misplaced optimism, and resulting cutbacks in enthusiasm and funding. There have also been cycles of introducing new creative approaches and systematically refining the best ones.
- Recent progress in understanding the theoretical basis for intelligence has gone hand in hand with improvements in the capabilities of real systems.

SOLVING PROBLEMS BY SEARCHING

In which we see how an agent can look ahead to find a sequence of actions that will eventually achieve its goal. When the correct action to take is not immediately obvious, an agent may need to plan ahead: to consider a sequence of actions that form a path to a goal state. Such an agent is called a problem-solving agent, and the computational process it undertakes is called search. Problem-solving agent Problem-solving agents use atomic representations, as described in Section 2.4.7—that Search is, states of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms. Agents that use factored or structured representations of states are called planning agents and are discussed in Chapters 7 and 11. We will cover several search algorithms. In this chapter, we consider only the simplest environments: episodic, single agent, fully observable, deterministic, static, discrete, and known. We distinguish between informed algorithms, in which the agent can estimate how far it is from the goal, and uninformed algorithms, where no such estimate is available. Chapter 4 relaxes the constraints on environments, and Chapter 5 considers multiple

agents. This chapter uses the concepts of asymptotic complexity (that is, O(n) notation). Readers unfamiliar with these concepts should consult Appendix A.

3.1 Problem-Solving Agents

Imagine an agent enjoying a touring vacation in Romania. The agent wants to take in the sights, improve its Romanian, enjoy the nightlife, avoid hangovers, and so on. The decision problem is a complex one. Now, suppose the agent is currently in the city of Arad and has a nonrefundable ticket to fly out of Bucharest the following day. The agent observes street signs and sees that there are three roads leading out of Arad: one toward Sibiu, one to Timisoara, and one to Zerind. None of these are the goal, so unless the agent is familiar with the geography of Romania, it will not know which road to follow.1 If the agent has no additional information—that is, if the environment is unknown—then the agent can do no better than to execute one of the actions at random. This sad situation is discussed in Chapter 4. In this chapter, we will assume our agents always have access to information about the world, such as the map in Figure 3.1. With that information, the agent can follow this four-phase problem-solving process:

• **Goal formulation**: The agent adopts the goal of reaching Bucharest. Goals organize Goal formulation behavior by limiting the objectives and hence the actions to be considered.

Problem Formulation: Problem formulation involves defining the problem that the agent needs to solve. It includes identifying the possible states, actions, and transition model that describes how the agent can move from one state to another by taking specific actions. Problem formulation lays the foundation for the agent's search process to find a solution.

Search: Search is the process of applying a search algorithm to explore the state space and find a solution to the problem. The process of looking for a sequence of actions that reaches the goal is called search. A search algorithm takes a problem as input and returns a solution in the form of an action sequence.

Execution: the agent can now execute the action in the solution one at the time. Once a solution is found, the actions it recommends can be carried out. This is called the execution phase. Thus, we have a simple "formulate, search, execute" design for the agent, as shown in Figure 3.1.

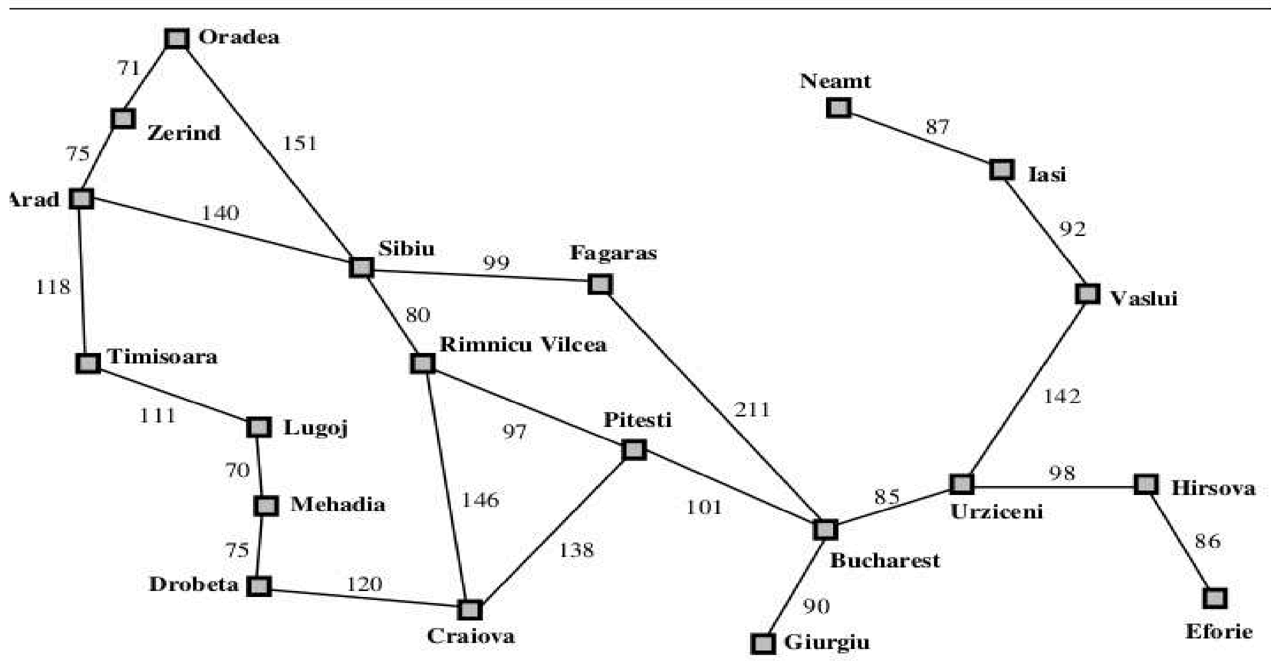


Figure 3.2 A simplified road map of part of Romania.

3.1.1 Search Problems and solutions:

A search problem can be defined as follows:

1. State Space (S):

The state space, denoted as S, is the set of all possible states that the agent can be in during its interaction with the environment. Each state in the state space represents a distinct configuration or situation.

2. Initial State (s0):

The initial state, denoted as s0, is the starting state from which the search begins. It is the state where the agent begins its exploration or problem-solving process.

In(Arad).

3. Actions (A):

Actions, denoted as A, represent the set of possible moves or actions that the agent can take to transition from one state to another. Each action is associated with a specific state transition. For example, from the state In(Arad), the possible actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.

4. Transition Model (T):

The transition model, denoted as T, describes the possible outcomes of actions. It is a function that maps a state-action pair (s, a) to the resulting state s' in the state space S. In other words, T(s, a) = s'.

R ESULT (In(Arad), Go(Zerind)) = In(Zerind) .

5. Goal Test (Goal(s)):

The goal test is a condition that determines whether a given state s is a goal state or not. It defines the desired outcome or condition that the agent wants to achieve. It can be a single state or a set of states that represent the goal.

6. Path Cost (optional, c(s, a, s')):

The path cost function, denoted as c(s, a, s'), assigns a numerical cost to each action. It represents the cost of taking action a from state s to reach state s'. The path cost function is often used in optimization problems to find the lowest-cost path to the goal.

The goal of the agent in a search problem is to find a sequence of actions or states that lead from the initial state s0 to a goal state that satisfies the goal test. The process of finding this sequence is known as search, and there are various search algorithms designed to explore the state space and find solutions efficiently based on the problem's characteristics.

By defining these components of a search problem, we provide a formal and well-defined framework for designing intelligent agents and solving various AI problems, such as pathfinding, puzzle-solving, game playing, and planning tasks.

3.1.2 Formulating Problems:

In the preceding section we proposed a formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost. This formulation seems reasonable, but it is still a model—an abstract mathematical description—and not the real thing. Compare the simple state description we have chosen, In(Arad), to an actual crosscountry trip, where the state of the world includes so many things: the traveling companions, what is on the radio, the scenery out of the window, whether there are any law enforcement officers nearby, how far it is to the next rest stop, the condition of the road, the weather, and so on. All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest. The process of removing detail ABSTRACTION from a representation is called abstraction. In addition to abstracting the state

description, we must abstract the actions themselves. A driving action has many effects. Besides changing the location of the vehicle and its occupants, it takes up time, consumes fuel, generates pollution, and changes the agent (as they say, travel is broadening). Our formulation takes into account only the change in location. Also, there are many actions that we will omit altogether: turning on the radio, looking out of the window, slowing down for law enforcement officers, and so on. And of course, we don't specify actions at the level of "turn steering wheel to the left by three degrees." Can we be more precise about defining the appropriate level of abstraction? Think of the abstract states and actions we have chosen as corresponding to large sets of detailed world states and detailed action sequences. Now consider a solution to the abstract problem: for example, the path from Arad to Sibiu to Rimnicu Vilcea to Pitesti to Bucharest. This abstract solution corresponds to a large number of more detailed paths. For example, we could drive with the radio on between Sibiu and Rimnicu Vilcea, and then switch it off for the rest of the trip. The abstraction is valid if we can expand any abstract solution into a solution in the more detailed world; a sufficient condition is that for every detailed state that is "in Arad," there is a detailed path to some state that is "in Sibiu," and so on.5 The abstraction is useful if carrying out each of the actions in the solution is easier than the original problem; in this case they are easy enough that they can be carried out without further search or planning by an average driving agent. The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out. Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world

3.2 EXAMPLE PROBLEMS

The problem-solving approach has been applied to a vast array of task environments. We list some of the best known here, distinguishing between toy and real-world problems. A TOY PROBLEM toy problem is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms. A real-world problem is one whose solutions people actually care about. They tend not to have a single agreed-upon description, but we will attempt to give the general flavor of their formulations.

3.2.1 Standardized problems

The grid world problem is a classic concept used in the field of artificial intelligence and reinforcement learning. It is often represented as a two-dimensional grid or array, where each cell in

the grid can be in one of several states. Agents can navigate through the grid, and the goal is typically to find an optimal policy for the agent to achieve a specific objective.

The first example we will examine is the vacuum world first introduced in Chapter 2. (See Figure 2.2.) This can be formulated as a problem as follows:

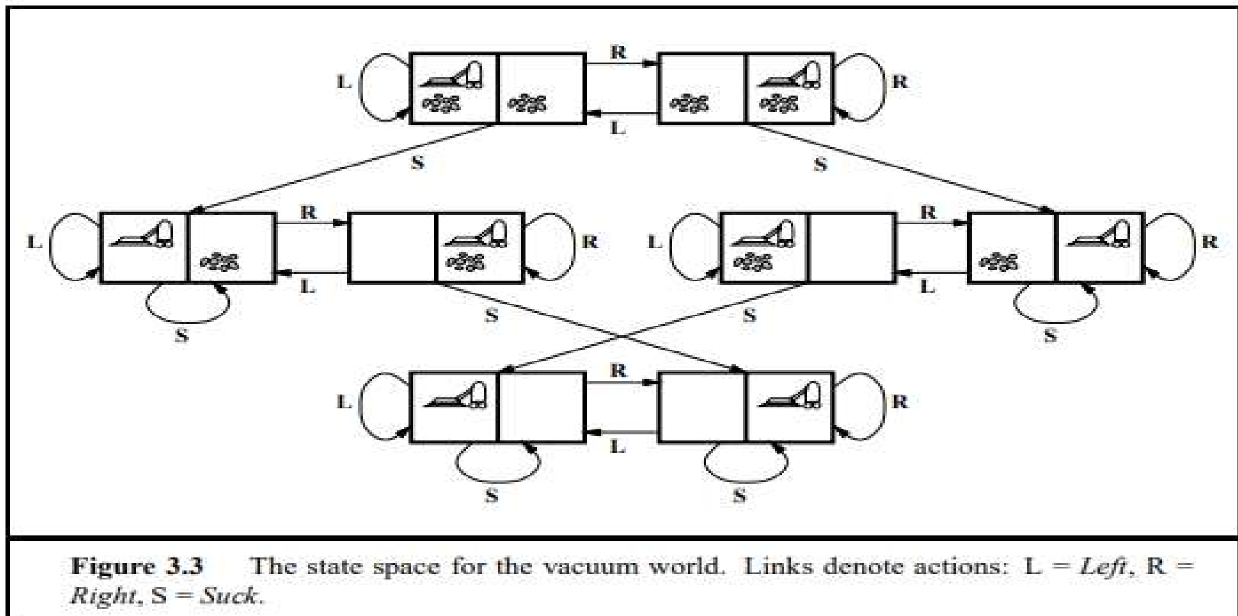


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

• States: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus there are 2 × 2 2 = 8 possible world states. A larger environment with n locations has n · 2 n states.

• Initial state: Any state can be designated as the initial state.

• Actions: In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.

• Transition model: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect. The complete state space is shown in Figure 3.3.

• Goal test: This checks whether all the squares are clean.

• Path cost: Each step costs 1, so the path cost is the number of steps in the path.

The 8-puzzle, an instance of which is shown in Figure 3.4, consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation is as follows:

• States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

• Initial state: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states (Exercise 3.17).

• Actions: The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.

• Transition model: Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.

• Goal test: This checks whether the state matches the goal configuration shown in Figure 3.4. (Other goal configurations are possible.)

• Path cost: Each step costs 1, so the path cost is the number of steps in the path.

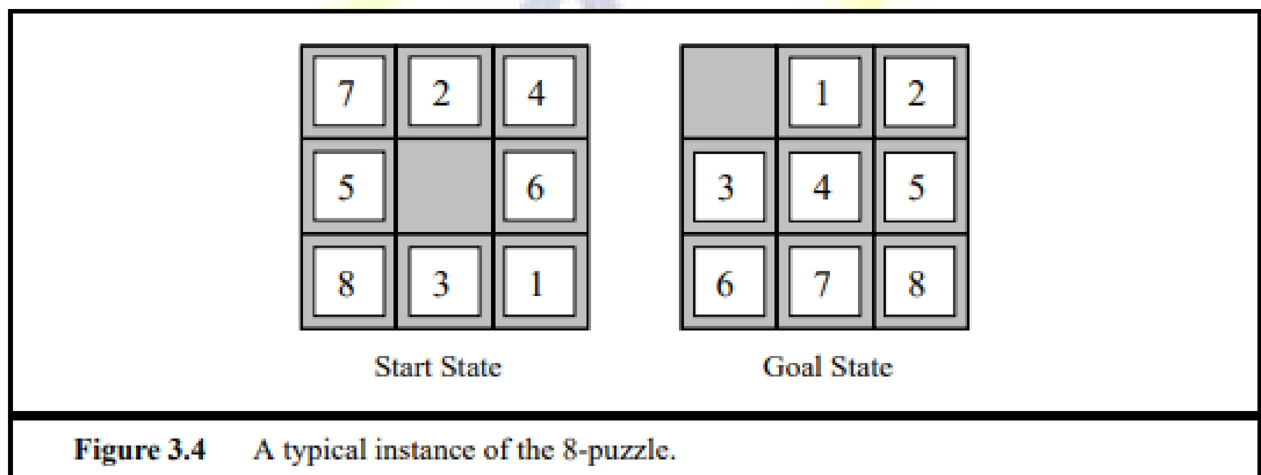


Figure 3.4 A typical instance of the 8-puzzle.

The 8-puzzle belongs to the family of sliding-block puzzles, which are often used as SLIDING-BLOCK PUZZLES test problems for new search algorithms in AI. This family is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms described in this chapter and the next. The 8-puzzle has 9!/2 = 181, 440 reachable states and is easily solved. The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms.

$$\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \rfloor = 5 .$$

The problem definition is very simple:

• States: Positive numbers.
• Initial state: 4.
• Actions: Apply factorial, square root, or floor operation. Factorial can be applied only to integers.
• Transition model: As given by the mathematical definitions of the operations.
• Goal test: State is the desired positive integer.

To our knowledge there is no bound on how large a number might be constructed in the process of reaching a given target—for example, the number 620,448,401,733,239,439,360,000 is generated in the expression for 5—so the state space for this problem is infinite. Such state spaces arise very frequently in tasks involving the generation of mathematical expressions, circuits, proofs, programs, and other recursively defined objects.

3.2.2 Real-world problems

Consider the airline travel problems that must be solved by a travel planning Web site:

• States: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and whether they were domestic or international, the state must record extra information about these "historical" aspects.
• Initial state: This is specified by the user's query
• Actions: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if there is a preceding flight segment.
• Transition model: The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
• Goal test: Are we at the final destination specified by the user?
• Path cost: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

Commercial travel advice systems use a problem formulation of this kind, with many additional complications to handle the byzantine fare structures that airlines impose. Any seasoned traveler knows, however, that not all air travel goes according to plan. A really good system should include contingency plans—such as backup reservations on alternate flights— to the extent that these are justified by the cost and likelihood of failure of the original plan.

Touring problems are closely related to route-finding problems, but with an important difference. Consider, for example, the problem "Visit every city in Figure 3.2 at least once, starting and ending in Bucharest." As with route finding, the actions correspond to trips between adjacent cities. The state

space, however, is quite different. Each state must include not just the current location but also the set of cities the agent has visited. So the initial state would be In(Bucharest), Visited({Bucharest}), a typical intermediate state would be In(Vaslui), Visited({Bucharest, Urziceni, Vaslui}), and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

The traveling salesperson problem (TSP) is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

A VLSI layout problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase, and is usually split into two parts: cell layout and channel routing. In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving. Later in this chapter, we will see some algorithms capable of solving them.

Robot navigation is a generalization of the route-finding problem described earlier. Rather than a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite. We examine some of these methods in Chapter 25. In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls.

Automatic assembly sequencing of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972). Progress since then has been slow but sure, to the point where the assembly of intricate objects such as electric motors is economically feasible. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the generation of legal actions is the expensive part of assembly

sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space. Another important assembly problem is **protein design**, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

3.3 SEARCHING FOR SOLUTIONS

A search algorithm takes a search problem as input and returns a solution or an indication of failure.

A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem. Figure 3.6 shows the first few steps in growing the search tree for finding a route from Arad to Bucharest. The root node of the tree corresponds to the initial state, In(Arad).

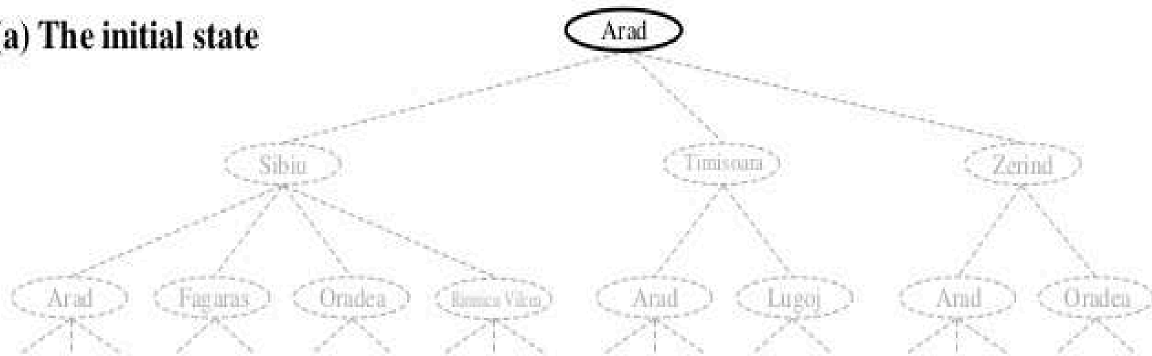
The first step is to test whether this is a goal state. Then we need to consider taking various actions. This is done by expanding the current state; that is, applying each legal action to the current state, thereby generating a new set of states. In this case, we add three branches from the parent node In(Arad) leading to three new child nodes: In(Sibiu), In(Timisoara), and In(Zerind).

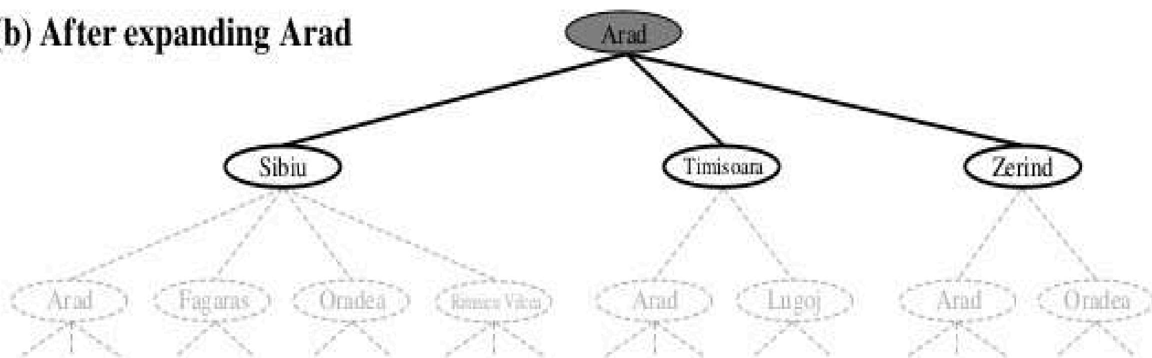
Now we must choose which of these three possibilities to consider further. This is the essence of search—following up one option now and putting the others aside for later, in case the first choice does not lead to a solution. Suppose we choose Sibiu first.

We check to see whether it is a goal state (it is not) and then expand it to get In(Arad),

In(Fagaras), In(Oradea), and In(RimnicuVilcea). We can then choose any of these four, or go back and choose Timisoara or Zerind. Each of these six nodes is a leaf node, that is, a node with no children in the tree. The set of all leaf nodes available for expansion at any given point is called the frontier (open list).

In Figure 3.6, the frontier of each tree consists of those nodes with bold outlines.





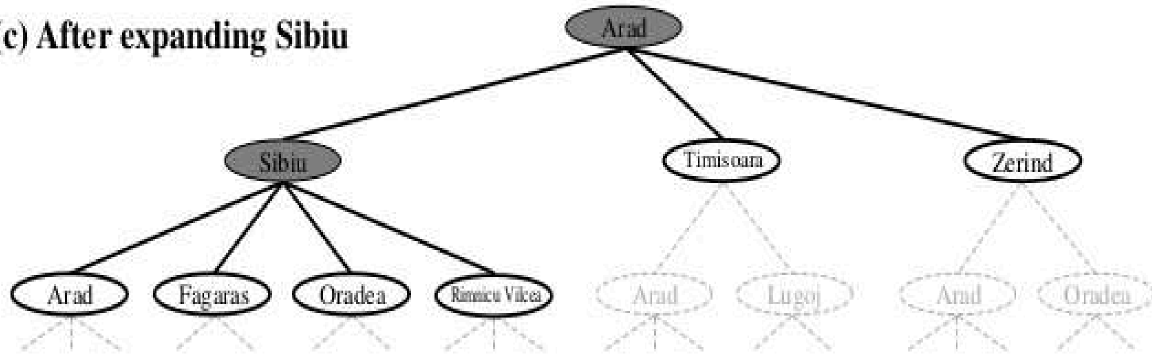


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

The process of choosing and expanding nodes in the frontier continues until either a solution is found or there are no more states to be expanded. The general TREE -S EARCH algorithm is shown informally in Figure 3.7.

Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called **search strategy.**

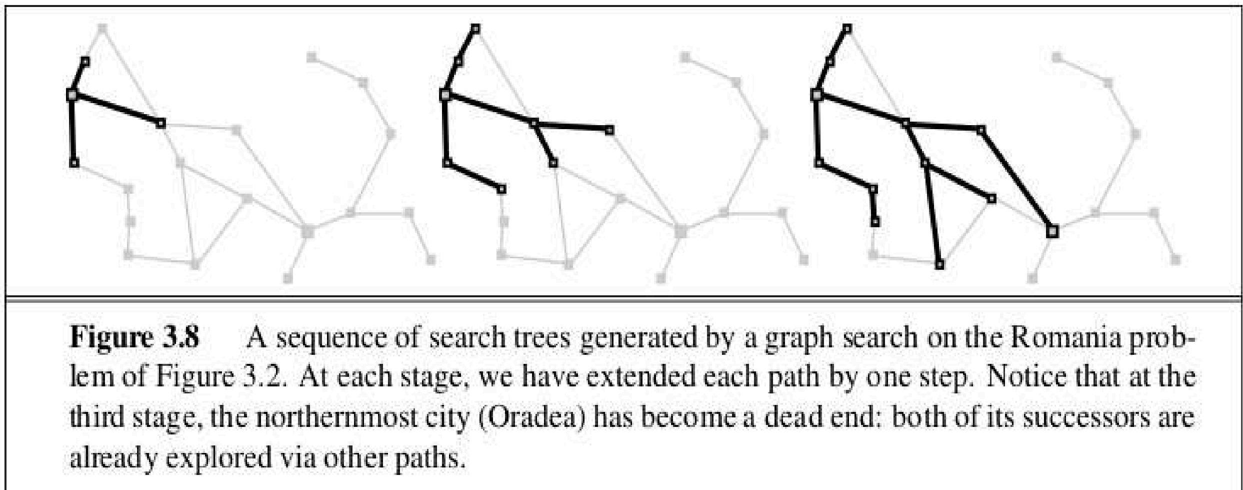


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

Clearly, the search tree constructed by the GRAPH -S SEARCH algorithm contains at most one copy of any given state, so we can think of it as growing a tree directly on the state-space graph itself, as shown in Figure 3.8.

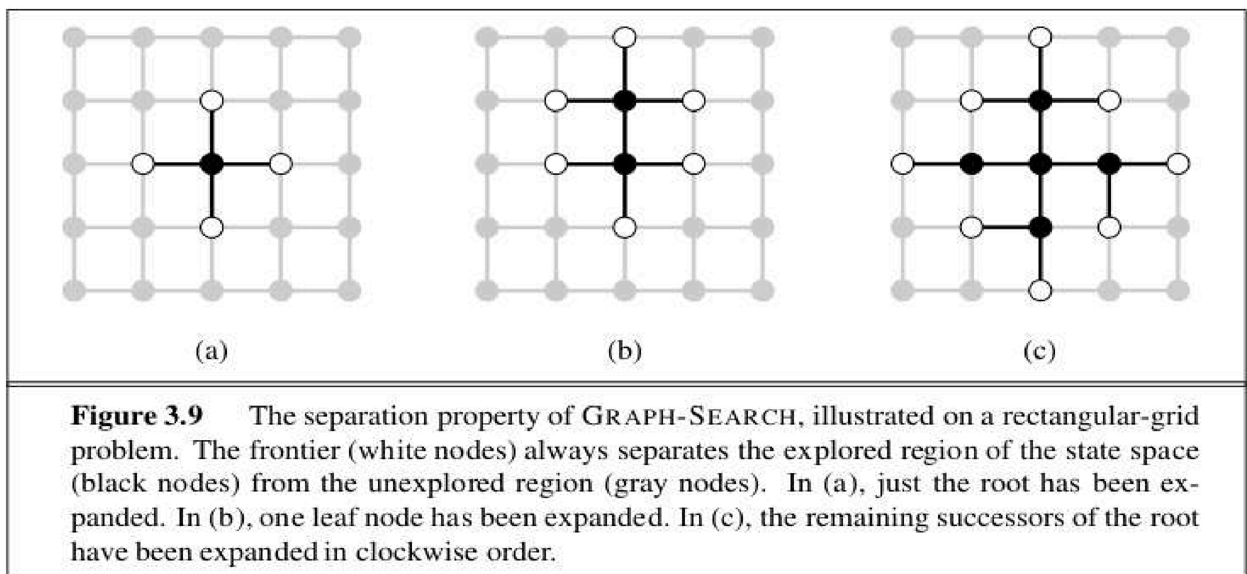


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

The frontier separates the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier. (If this seems completely obvious, try Exercise 3.20 now.) This property is illustrated in Figure 3.9. As every step moves a state from the frontier into the explored region, while moving some states from the unexplored region into the frontier, we see that the algorithm is systematically examining the states in the state space, one by one, until it finds a solution.

3.3.1 Best-first search

Best-first search is an instance of the general TREE -SEARCH or GRAPH -SEARCH algorithm in which a node is selected for expansion based on an evaluation function, f (n).

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Figure 3.7 The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for **yield**.

3.3.2 Search data structures

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we will have a structure that contains the following four components:

- n.STATE : the state in the state space to which the node corresponds;
- n.PARENT : the node in the search tree that generated this node;
- n.ACTION : the action that was applied to the parent to generate the node;
- n.PATH -COST : the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

We need a data structure to store the **frontier**. The appropriate choice is a **queue** of some kind, because the operations on a frontier are:

- IS-EMPTY(*frontier*) returns true only if there are no nodes in the frontier.
- POP(*frontier*) removes the top node from the frontier and returns it.
- TOP(*frontier*) returns (but does not remove) the top node of the frontier.
- ADD(*node*, *frontier*) inserts node into its proper place in the queue.

queue used for search algorithms:

Queues are characterized by the order in which they store the inserted nodes. Three common variants are the first-in, first-out or **FIFO** queue, which pops the oldest element of the queue; the last-in, first-out or **LIFO** queue (also known as a stack), which pops the newest element of the queue; and the **priority** queue, which pops the element of the queue with the highest priority according to some ordering function.

3.3.3 Redundant paths

The eagle-eyed reader will notice one peculiar thing about the search tree shown in Fig-
ure 3.6: it includes the path from Arad to Sibiu and back to Arad again! We say that In(Arad) is a **repeated state** in the search tree, generated in this case by a **loopy path.**
Loopy paths are a special case of the more general concept of redundant paths, which
exist whenever there is more than one way to get from one state to another. Consider the paths Arad–Sibiu (140km long) and Arad–Zerind–Oradea–Sibiu (297km long). Obviously, the second path is redundant—it's just a worse way to get to the same state. If you are concerned about reaching the goal, there's never any reason to keep around more than one path to any given state, because any goal state that is reachable by extending one path is also reachable by extending the other.

3.3.4 Measuring problem-solving performance

We will evaluate an algorithm's performance in four ways:
COMPLETENESS
OPTIMALITY
TIME COMPLEXITY
SPACE COMPLEXITY

- Completeness: Is the algorithm guaranteed to find a solution when there is one?
- Optimality: Does the strategy find the optimal solution, as defined on page 69?
- Time complexity: How long does it take to find a solution?
- Space complexity: How much memory is needed to perform the search?

Time and space complexity are always considered with respect to some measure of the problem's difficulty. In theoretical computer science, the typical measure is the size of the state space graph, |V | + |E|. where V is the set of vertices (nodes) of the graph and E is the set of edges (links).
complexity is expressed in terms of three quantities: b, the branching factor or maximum number of successors of any node; d, the depth of the shallowest goal node (i.e., the number of steps along the path from the root); and m, the maximum length of
any path in the state space.

Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory.

3.4 UNINFORMED SEARCH STRATEGIES

3.4.1 Breadth-first search

Breadth-first search 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.

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Figure 3.9 Breadth-first search and uniform-cost search algorithms.

Breadth-first search is an instance of the general graph search algorithm in which the shallowest unexpanded node is chosen for expansion. This 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.
goal test is applied to each node when it is generated, rather than when it is selected for expansion.
Figure 3.12 shows the progress of the search on a simple binary tree.

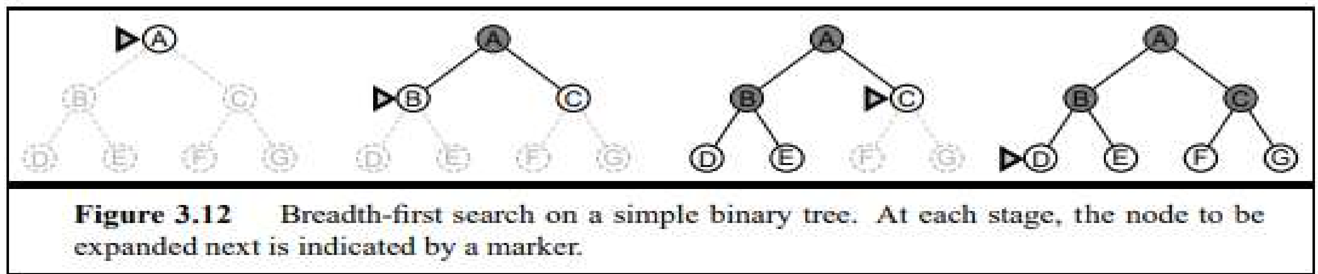


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

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

$1+ b + b^2 + b^3 + \cdots + b^d = O(b^d)$.

Depth	Nodes	Time	Memory
2	110	1.1 milliseconds	107 kilobytes
4	11,110	111 milliseconds	10.6 megabytes
6	10^6	11 seconds	1 gigabytes
8	10^8	19 minutes	103 gigabytes
10	10^{10}	31 hours	10 terabytes
12	10^{12}	129 days	1 petabytes
14	10^{14}	35 years	99 petabytes
16	10^{16}	3,500 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 100,000 nodes/second; 1000 bytes/node.

3.4.3 Depth-first search and the problem of memory

Depth-first search always expands the deepest node in the current frontier of the search tree. The progress of the search is illustrated in Figure 3.16.

AJIET

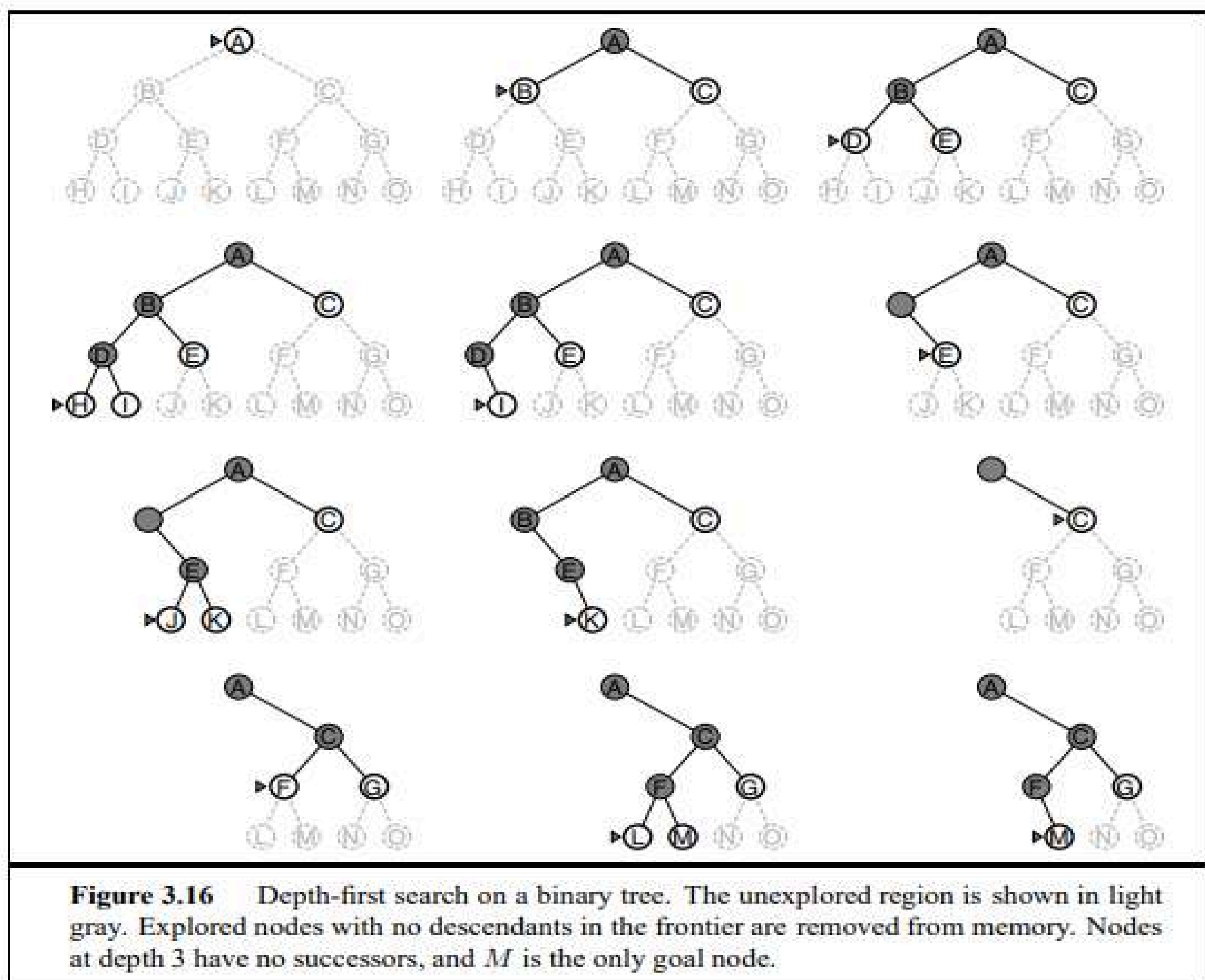


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors, and M is the only goal node.

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.

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. This must be the deepest unexpanded node, because it is one deeper that its parent—which, in turn, was the deepest unexpanded node when it was selected.

AJIET

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.

A variant of depth-first search called backtracking search uses still less memory. In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In this way, only $O(m)$ memory is needed rather than

$O(b^m)$. Backtracking search facilitates yet another memory-saving (and time-saving) trick: the idea of generating a successor by modifying the current state description directly rather than copying it first. This reduces the memory requirements to just one state description and $O(m)$ actions. For this to work, we must be able to undo each modification when we go back to generate the next successor. For problems with large state descriptions, such as robotic assembly, these techniques are critical to success.