UNIT I

Introduction – Foundations of AI, the History of AI –Intelligent Agent –Agent and Environment, Good Behaviour: The Concept ofRationality, Nature of Environments, Structure of Agents- Problem Solving Agents -Example Problems.

UNIT II

Uninformed Searching strategies-Breadth First Search, Depth First search, Depth limited search, Iterative deepening search, Bidirectional Search - Avoiding repeated States - Searching with Partial information –Informed search strategies – Greedy Best First Search-A* Search-Heuristic Functions- Local Search Algorithms for Optimization Problems-Local search in Continuous Spaces. Online Search Agents and Unknown Environments-Online Search Problems, Online Search Agents- Online Local search, learning in Online Search

UNIT-1

Introduction – Foundations of AI, the History of AI –Intelligent Agent – Agent and Environment, Good Behaviour: The Concept of Rationality, Nature of Environments, Structure of Agents- Problem Solving Agents -Example Problems.

AIM & OBJECTIVES

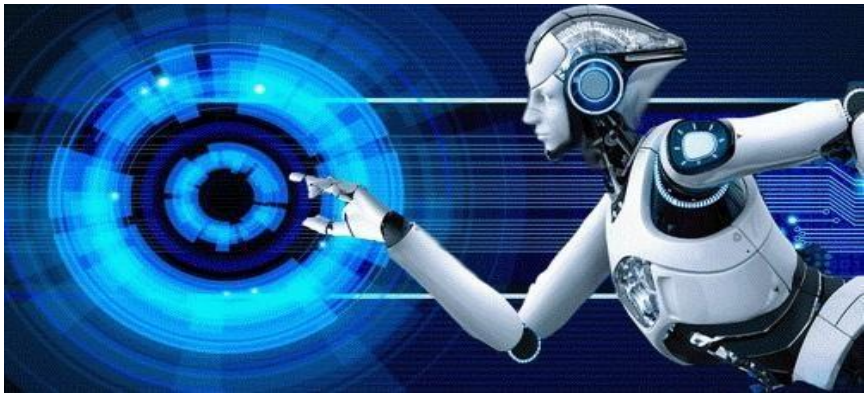
To understand some fundamentals of AI and algorithms required to produce AI systems.

PRE- REQUISITE: Basic knowledge of Computer Architecture.

In today's world, technology is growing very fast, and we are getting in touch with different new technologies day by day.

Here, one of the booming technologies of computer science is Artificial Intelligence which is ready to create a new revolution in the world by making intelligent machines. The Artificial Intelligence is now all around us. It is currently working with a variety of subfields, ranging from general to specific, such as self-driving cars, playing chess, proving theorems, playing music, Painting, etc.

AI is one of the fascinating and universal fields of Computer science which has a great scope in future. AI holds a tendency to cause a machine to work as a human.



Artificial Intelligence is composed of two words Artificial and Intelligence, where Artificial defines "man- made," and intelligence defines "thinking power", hence AI means "a man-made thinking power."

So, we can define AI as:

"It is a branch of computer science by which we can create intelligent machines which can behave like a human, think like humans, and able to make decisions."

Artificial Intelligence exists when a machine can have human based skills such as learning, reasoning, and solving problems.

With Artificial Intelligence you do not need to preprogram a machine to do some work, despite that you can create a machine with programmed algorithms which can work with own intelligence, and that is the awesomeness of AI.
It is believed that AI is not a new technology, and some people says that as per Greek myth, there were Mechanical men in early days which can work and behave like humans.

Why Artificial Intelligence?

Before Learning about Artificial Intelligence, we should know that what is the importance of AI and why should we learn it.

Following are some main reasons to learn about AI:

- With the help of AI, you can create such software or devices which can solve real-world problems very easily and with accuracy such as health issues, marketing, traffic issues, etc.
- With the help of AI, you can create your personal virtual Assistant, such as Cortana, Google Assistant, Siri, etc.
- With the help of AI, you can build such Robots which can work in an environment where survival of humans can be at risk.
- AI opens a path for other new technologies, new devices, and new Opportunities.

Goals of Artificial Intelligence

Following are the main goals of Artificial Intelligence:

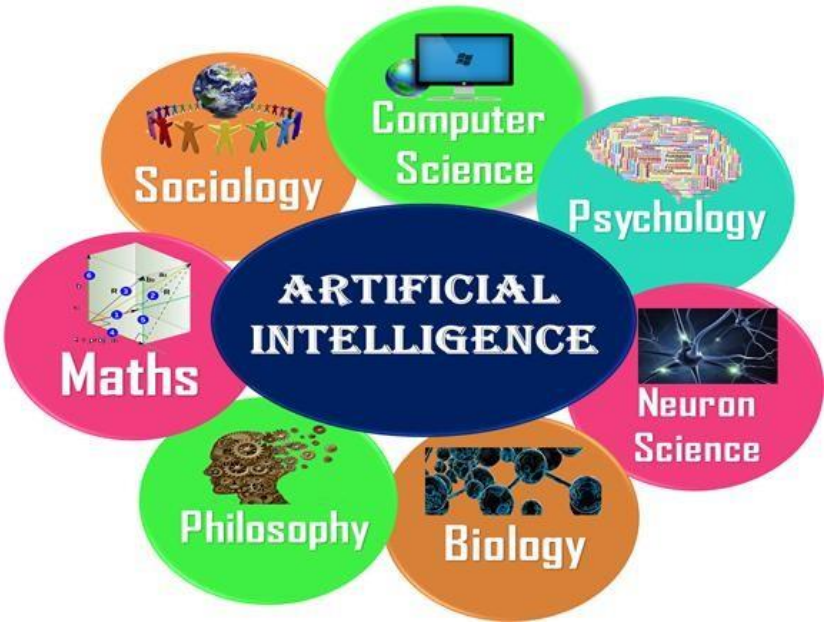
1. Replicate human intelligence
2. Solve Knowledge-intensive tasks
3. An intelligent connection of perception and action
4. Building a machine which can perform tasks that requires human intelligence such as:
    - Proving a theorem
    - Playing chess
    - Plan some surgical operation
    - Driving a car in traffic
5. Creating some system which can exhibit intelligent behavior, learn new things by itself, demonstrate, explain, and can advise to its user.

What Comprises to Artificial Intelligence?

Artificial Intelligence is not just a part of computer science even it's so vast and requires lots of other factors which can contribute to it. To create the AI first we should know that how intelligence iscomposed, so the Intelligence is an intangible part of our brain whichis a combination of Reasoning, learning, problem-solving perception, language understanding, etc.

To achieve the above factors for a machine or software Artificial Intelligence requires the following discipline:

- Mathematics
- Biology
- Psychology
- Sociology
- Computer Science
- Neurons Study
- Statistics



Advantages of Artificial Intelligence

Following are some main advantages of Artificial Intelligence:

- High Accuracy with less error: AI machines or systems are prone to less errors and high accuracy as it takes decisions asper pre-experience or information.
- High-Speed: AI systems can be of very high-speed and fast-decision making; because of that AI systems can beat a chess champion in the Chess game.
- High reliability: AI machines are highly reliable and can perform the same action multiple times with high accuracy.
- Useful for risky areas: AI machines can be helpful in situations such as defusing a bomb, exploring the ocean floor, where to employ a human can be risky.
- Digital Assistant: AI can be very useful to provide digitalassistant to the users such as AI technology is currently usedby various E-commerce websites to show the products as per customer requirement.
- Useful as a public utility: AI can be very useful for public utilities such as a self-driving car which can make our journey safer and hassle-free, facial recognition for security purpose, Natural language processing to communicate with the humanin human-language, etc.

Disadvantages of Artificial Intelligence

Every technology has some disadvantages, and the same goes for Artificial intelligence. Being so advantageous technology still, it has some disadvantages which we need to keep in our mind while creating an AI system.

Following are the disadvantages of AI:

- High Cost: The hardware and software requirement of AI is very costly as it requires lots of maintenance to meet current world requirements.
- Can't think out of the box: Even we are making smarter machines with AI, but still they cannot work out of the box, as the robot will only do that work for which they are trained, or programmed.
- No feelings and emotions: AI machines can be an outstanding performer, but still it does not have the feeling so it cannot

make any kind of emotional attachment with human, and may sometime be harmful for users if the proper care is not taken.

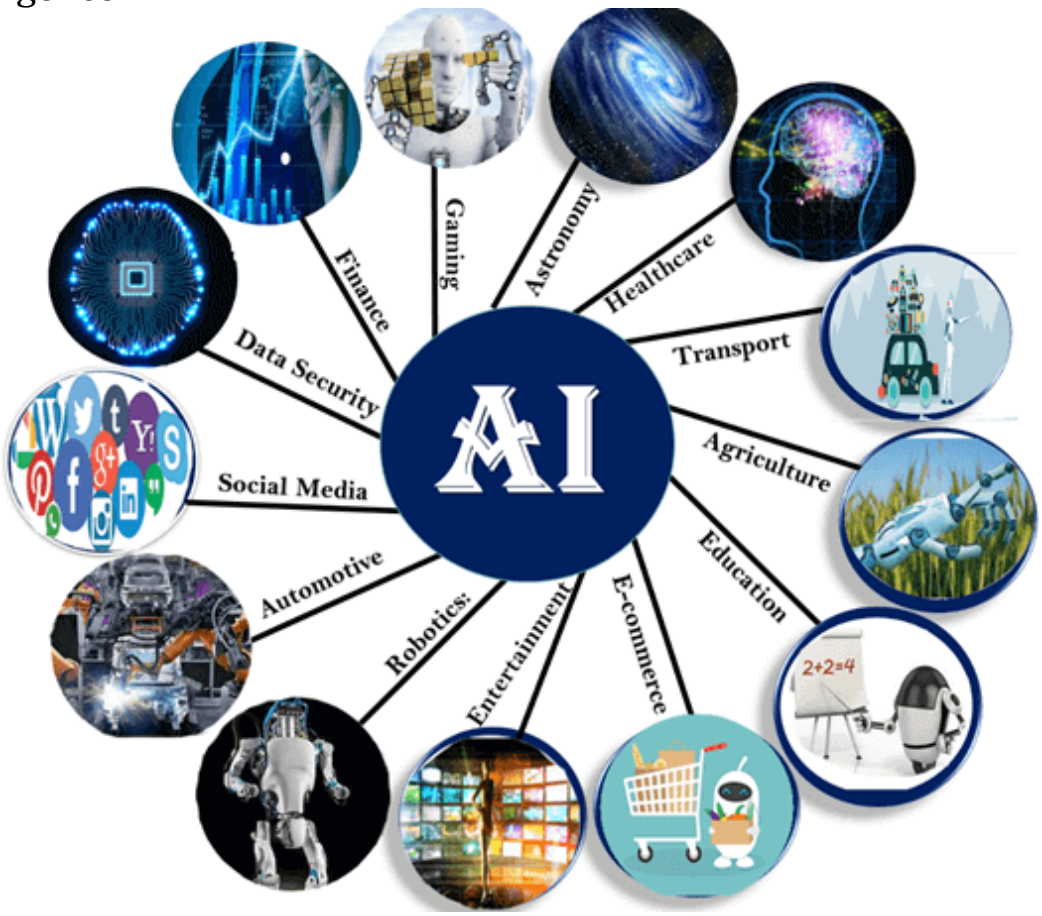
- Increase dependency on machines: With the increment of technology, people are getting more dependent on devices and hence they are losing their mental capabilities.
- No Original Creativity: As humans are so creative and can imagine some new ideas but still AI machines cannot beat this power of human intelligence and cannot be creative and imaginative.

Application of AI

Artificial Intelligence has various applications in today's society.

It is becoming essential for today's time because it can solve complex problems with an efficient way in multiple industries, such as Healthcare, entertainment, finance, education, etc.
AI is making our daily life more comfortable and fast.

Following are some sectors which have the application of Artificial Intelligence:



1. AI in Astronomy
    - Artificial Intelligence can be very useful to solve complex universe problems. AI technology can be helpful for understanding the universe such as how it works, origin, etc.

2. AI in Healthcare
    - In the last, five to ten years, AI becoming more advantageous for the healthcare industry and going to have a significant impact on this industry.

    - Healthcare Industries are applying AI to make a better and faster diagnosis than humans. AI can help doctors with diagnoses and can inform when patients are worsening so that medical help can reach to the patient before hospitalization.

3. AI in Gaming
    - AI can be used for gaming purpose. The AI machines can play strategic games like chess, where the machine needs to think of a large number of possible places.

4. AI in Finance
    - AI and finance industries are the best matches for each other. The finance industry is implementing automation, chatbot, adaptive intelligence, algorithm trading, and machine learning into financial processes.

5. AI in Data Security
    - The security of data is crucial for every company and cyber-attacks are growing very rapidly in the digital world. AI can be used to make your data more safe and secure. Some examples such as AEG bot, AI2 Platform, are used to determine software bug and cyber-attacks in a better way.

6. AI in Social Media
    - Social Media sites such as Facebook, Twitter, and Snapchatcontain billions of user profiles, which need to be stored and managed in a very efficient way. AI can organize and manage massive amounts of data. AI can analyze lots of data to identify the latest trends, hashtag, and requirement of different users.

7. AI in Travel & Transport
    - AI is becoming highly demanding for travel industries. AI is capable of doing various travel related works such as from making travel arrangement to suggesting the hotels, flights, and best routes to the customers. Travel industries are using AI-powered chatbots which can make human-like interaction with customers for better and fast response.

8. AI in Automotive Industry
    - Some Automotive industries are using AI to provide virtual assistant to their user for better performance. Such as Teslahas introduced TeslaBot, an intelligent virtual assistant.

    - Various Industries are currently working for developing self-driven cars which can make your journey more safe and secure.

9. AI in Robotics:
    - Artificial Intelligence has a remarkable role in Robotics. Usually, general robots are programmed such that they can perform some repetitive task, but with the help of AI, we can createintelligent robots which can perform tasks with their own experiences without pre-programmed.

    - Humanoid Robots are best examples for AI in robotics, recently the intelligent Humanoid robot named as Erica and Sophia has been developed which can talk and behave like humans.

10. AI in Entertainment
    - We are currently using some AI based applications in our daily life with some entertainment services such as Netflix or Amazon. With the help of ML/AI algorithms, these services show the recommendations for programs or shows.

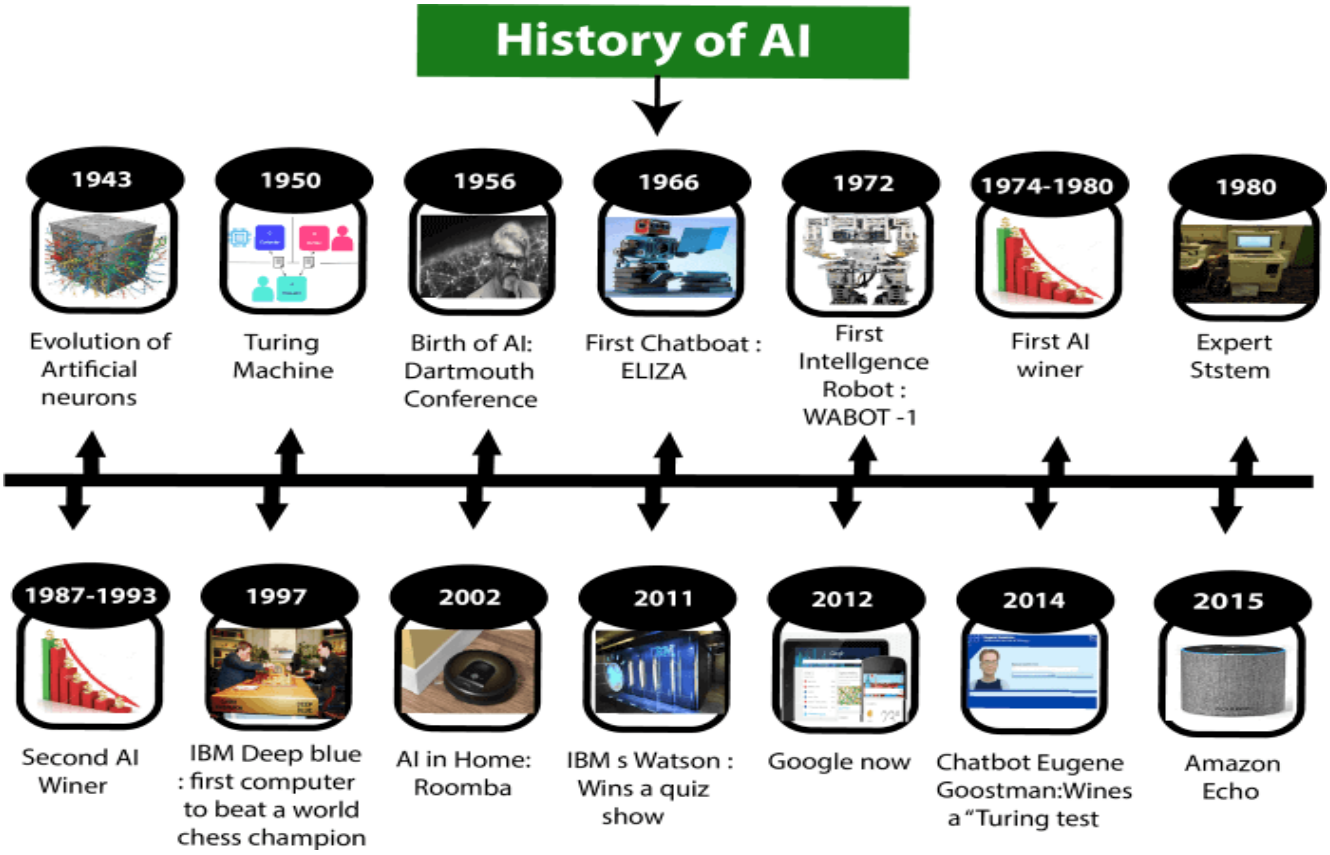
11. AI in Agriculture
    - Agriculture is an area which requires various resources, labor, money, and time for best result. Now a day's agriculture is becoming digital, and AI is emerging in this field. Agriculture is applying AI as agriculture robotics, solid and crop monitoring, predictive analysis. AI in agriculture can be very helpful for farmers.

12. AI in E-commerce
    - AI is providing a competitive edge to the e-commerce industry, and it is becoming more demanding in the e-commerce business. AI is helping shoppers to discover associated products with recommended size, color, or even brand.

13. AI in education:
    - AI can automate grading so that the tutor can have more time to teach. AI chatbot can communicate with students as a teaching assistant.
    - AI in the future can be work as a personal virtual tutor for students, which will be accessible easily at any time and any place.

History of Artificial Intelligence

Artificial Intelligence is not a new word and not a new technology for researchers. This technology is much older than you would imagine. Even there are the myths of Mechanical men in Ancient Greek and Egyptian Myths. Following are some milestones in the history of AI which defines the journey from the AI generation to till date development.



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 Turingpublishes "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.

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.

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.

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.

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.

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.

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 tobeat 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.

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.

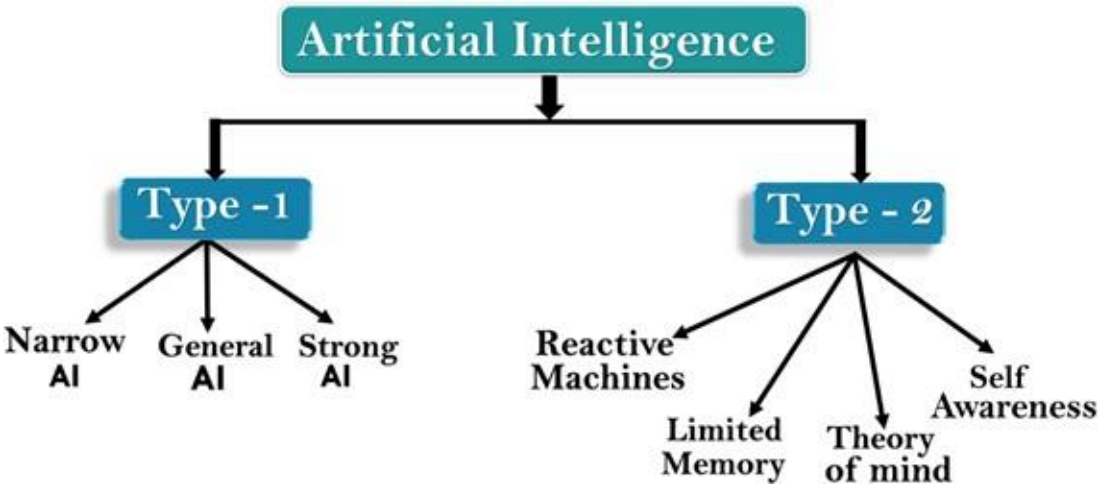
Now AI has developed to a remarkable level. The concept of Deep learning, big data, and data science are now trending like a boom. Nowadays companies like Google, Facebook, IBM, and Amazon are working with AI and creating amazing devices.

The future of Artificial Intelligence is inspiring and will come with high intelligence.

Types of Artificial Intelligence:

Artificial Intelligence can be divided in various types, there are mainly two types of main categorization which are based on capabilities and based on functionally of AI.

Following is flow diagram which explains the types of AI.



AI type-1: Based on Capabilities

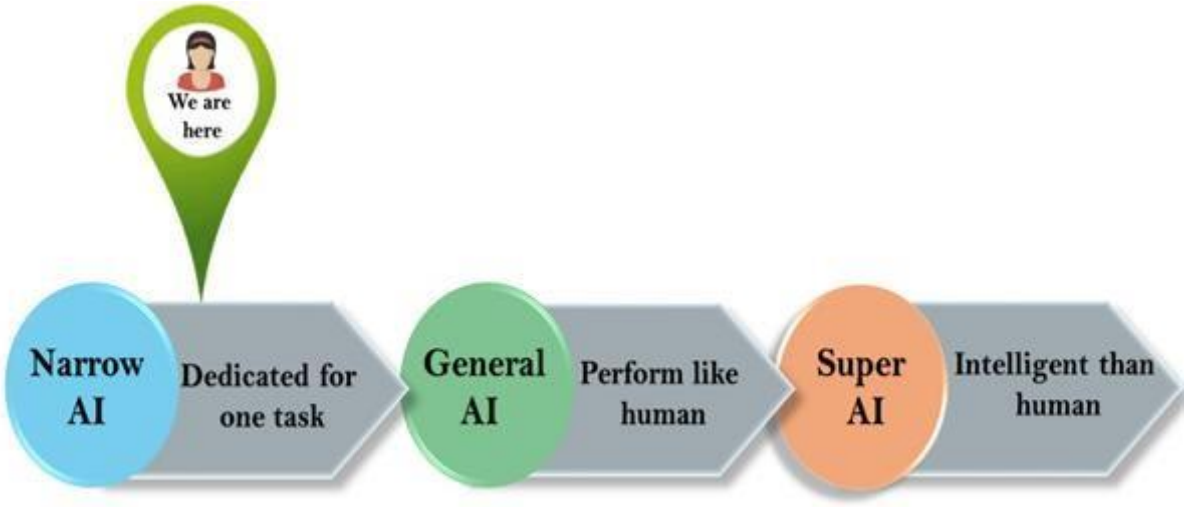
1. Weak AI or Narrow AI:

   - Narrow AI is a type of AI which is able to perform a dedicated task with intelligence. The most common and currently available AI is Narrow AI in the world of Artificial Intelligence.
   - Narrow AI cannot perform beyond its field or limitations, as it is only trained for one specific task. Hence it is also termed as weak AI. Narrow AI can fail in unpredictable ways if it goes beyond its limits.
   - Apple Siriis a good example of Narrow AI, but it operates with a limited pre-defined range of functions.
   - IBM's Watson supercomputer also comes under Narrow AI, as it uses an Expert system approach combined with Machinelearning and natural language processing.
   - Some Examples of Narrow AI are playing chess, purchasing suggestions on e-commerce site, self-driving cars, speech recognition, and image recognition.

2. General AI:
   - General AI is a type of intelligence which could perform any intellectual task with efficiency like a human.
   - The idea behind the general AI to make such a system which could be smarter and think like a human by its own.
   - Currently, there is no such system exist which could come under general AI and can perform any task as perfect as a human.

- The worldwide researchers are now focused on developing machines with General AI.
- As systems with general AI are still under research, and it will take lots of efforts and time to develop such systems.

3. Super AI:
   - Super AI is a level of Intelligence of Systems at which machines could surpass human intelligence, and can perform any task better than human with cognitive properties. It is an outcome of general AI.
   - Some key characteristics of strong AI include capability include the ability to think, to reason, solve the puzzle, make judgments, plan, learn, and communicate by its own.
   - Super AI is still a hypothetical concept of Artificial Intelligence. Development of such systems in real is still world changing task.



Artificial Intelligence type-2:

Based on functionality

1. Reactive Machines
   - Purely reactive machines are the most basic types of Artificial Intelligence.
   - Such AI systems do not store memories or past experiences for future actions.
   - These machines only focus on current scenarios and react on it as per possible best action.
   - IBM's Deep Blue system is an example of reactive machines.
   - Google's AlphaGo is also an example of reactive machines.

2. Limited Memory
    - Limited memory machines can store past experiences or some data for a short period of time.
    - These machines can use stored data for a limited time period only.
    - Self-driving cars are one of the best examples of Limited Memory systems. These cars can store recent speed of nearby cars, the distance of other cars, speed limit, and other information to navigate the road.

3. Theory of Mind
    - Theory of Mind AI should understand the human emotions, people, beliefs, and be able to interact socially like humans.
    - This type of AI machines is still not developed, but researchers are making lots of efforts and improvement for developing such AI machines.

4. Self-Awareness
    - Self-awareness AI is the future of Artificial Intelligence. These machines will be super intelligent, and will have their own consciousness, sentiments, and self-awareness.
    - These machines will be smarter than human mind.
    - Self-Awareness AI does not exist in reality still and it is a hypothetical concept.

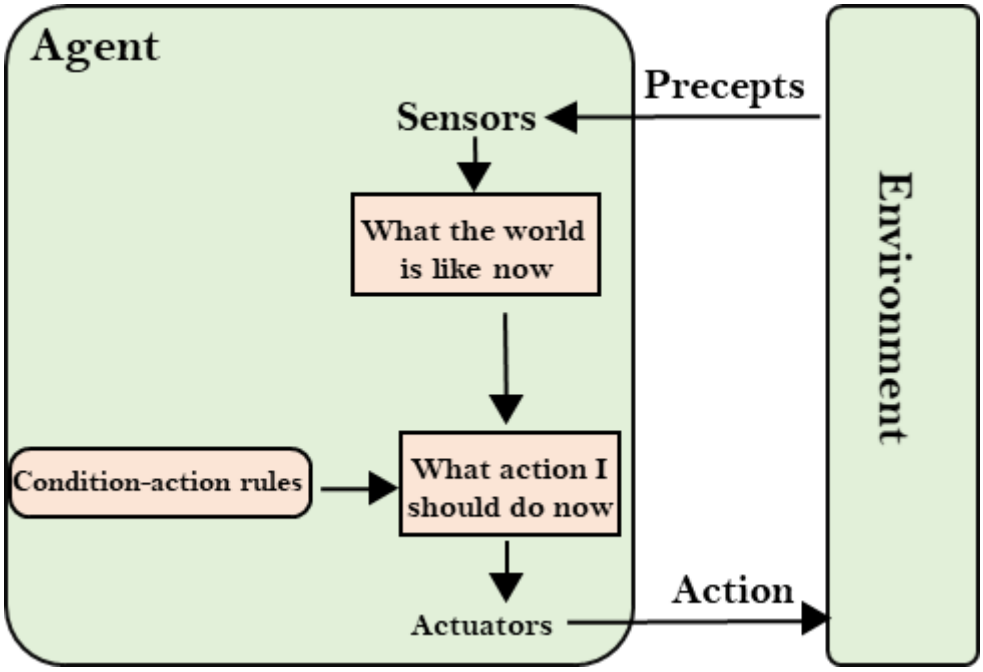
Types of AI Agents

Agents can be grouped into five classes based on their degree of perceived intelligence and capability. All these agents can improve their performance and generate better action over the time.

These are given below:

- Simple Reflex Agent
- Model-based reflex agent
- Goal-based agents
- Utility-based agent
- Learning agent

1. Simple Reflex agent:



- The Simple reflex agents are the simplest agents. These agents take decisions on the basis of the current percepts and ignore the rest of the percept history.
- These agents only succeed in the fully observable environment.
- The Simple reflex agent does not consider any part of percepts history during their decision and action process.
- The Simple reflex agent works on Condition-action rule, which means it maps the current state to action. Such as a Room Cleaner agent, it works only if there is dirt in the room.

- Problems for the simple reflex agent design approach:
  - They have very limited intelligence
  - They do not have knowledge of non-perceptual parts of the current state
  - Mostly too big to generate and to store.
  - Not adaptive to changes in the environment.

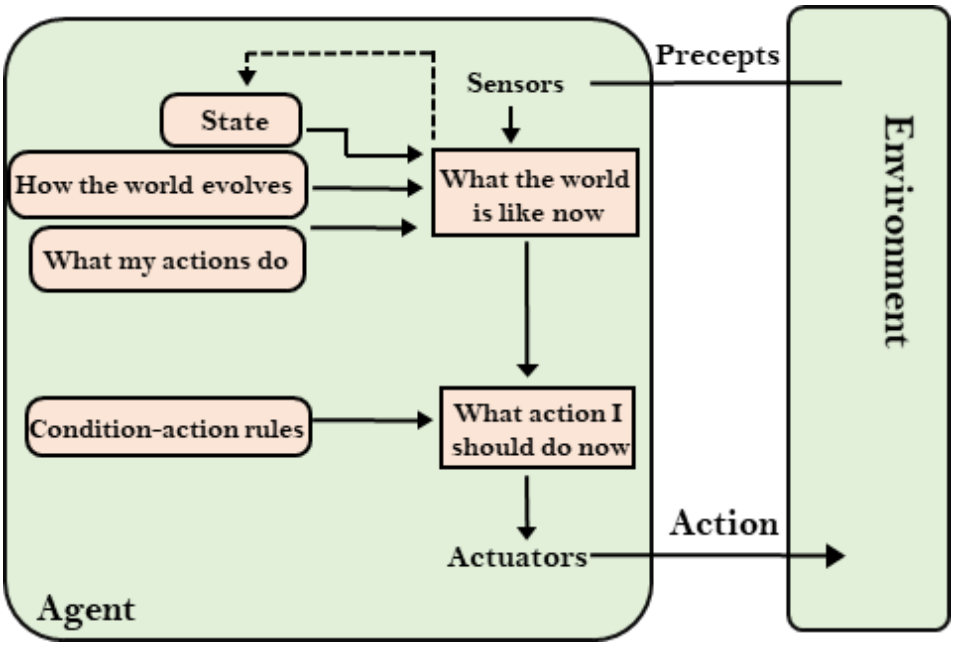
2. Model-based reflex agent

- The Model-based agent can work in a partially observable environment, and track the situation.

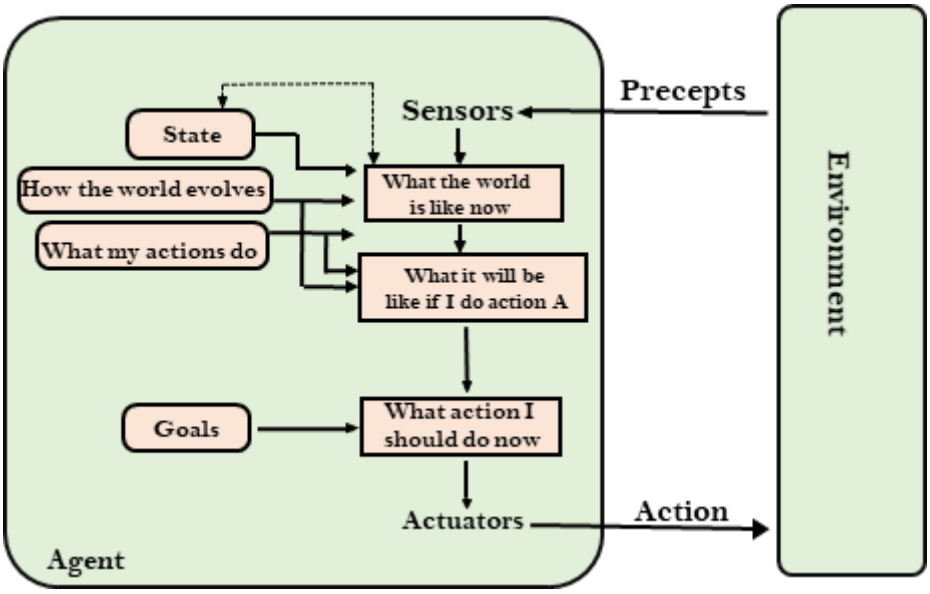
- A model-based agent has two important factors:
  - Model: It is knowledge about "how things happen in the world," so it is called a Model-based agent.
  - Internal State: It is a representation of the current state based on percept history.

- These agents have the model, "which is knowledge of the world" and based on the model they perform actions.

- Updating the agent state requires information about:
  a. How the world evolves
  b. How the agent's action affects the world.

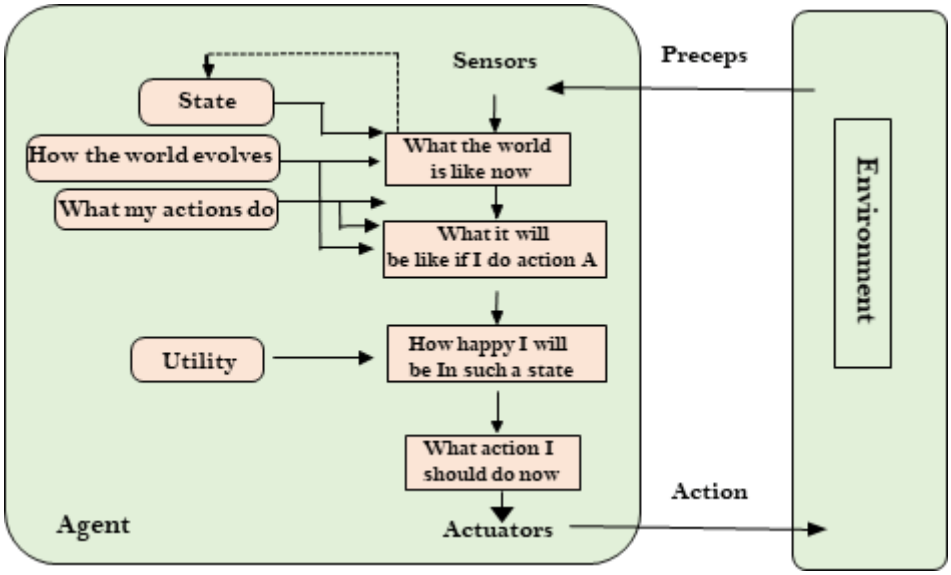


3. Goal-based agents



- The knowledge of the current state environment is not always sufficient to decide for an agent to what to do.
- The agent needs to know its goal which describes desirable situations.
- Goal-based agents expand the capabilities of the model-based agent by having the "goal" information.
- They choose an action, so that they can achieve the goal.
- These agents may have to consider a long sequence of possible actions before deciding whether the goal is achieved or not.Such considerations of different scenario are called searching and planning, which makes an agent proactive.

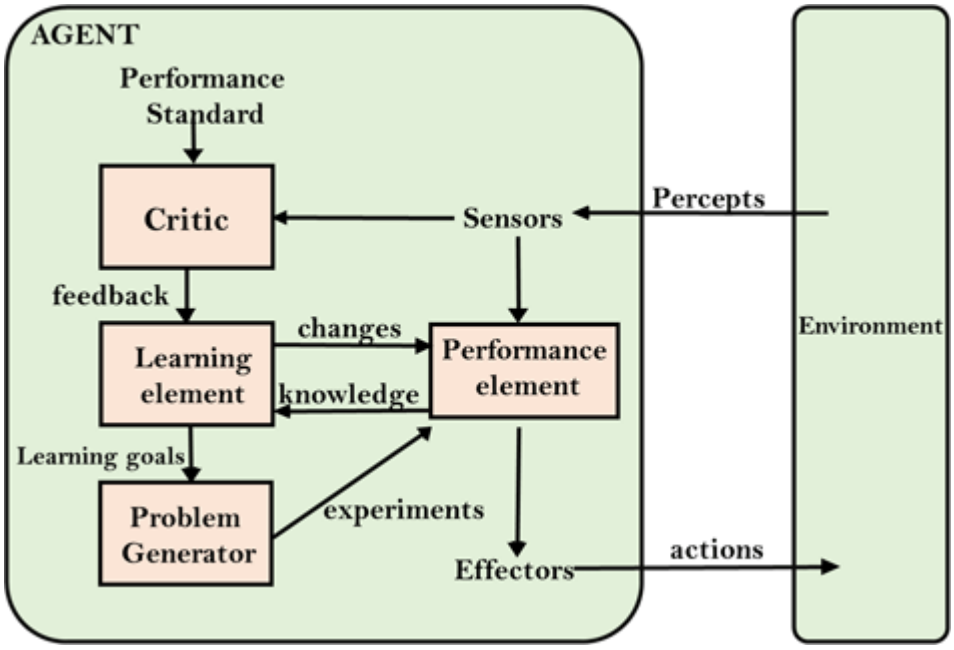
4. Utility-based agents
   - These agents are similar to the goal-based agent but provide an extra component of utility measurement which makes them different by providing a measure of success at a given state.
   - Utility-based agent act based not only goals but also the best way to achieve the goal.
   - The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.
   - The utility function maps each state to a real number to check how efficiently each action achieves the goals.



5. Learning Agents

- A learning agent in AI is the type of agent which can learn from its past experiences, or it has learning capabilities.
- It starts to act with basic knowledge and then able to act and adapt automatically through learning.
- A learning agent has mainly four conceptual components, which are:
    a. Learning element: It is responsible for making improvements by learning from environment
    b. Critic: Learning element takes feedback from critic which describes that how well the agent is doing with respect to a fixed performance standard.
    c. Performance element: It is responsible for selecting external action
    d. Problem generator: This component is responsible for suggesting actions that will lead to new and informative experiences.

- Hence, learning agents are able to learn, analyze performance, and look for new ways to improve the performance.



Agents in Artificial Intelligence

An AI system can be defined as the study of the rational agent and its environment. The agents sense the environment through sensors and act on their environment through actuators. An AI agent can have mental properties such as knowledge, belief, intention, etc.

What is an Agent?

An agent can be anything that perceive its environment through sensors and act upon that environment through actuators.

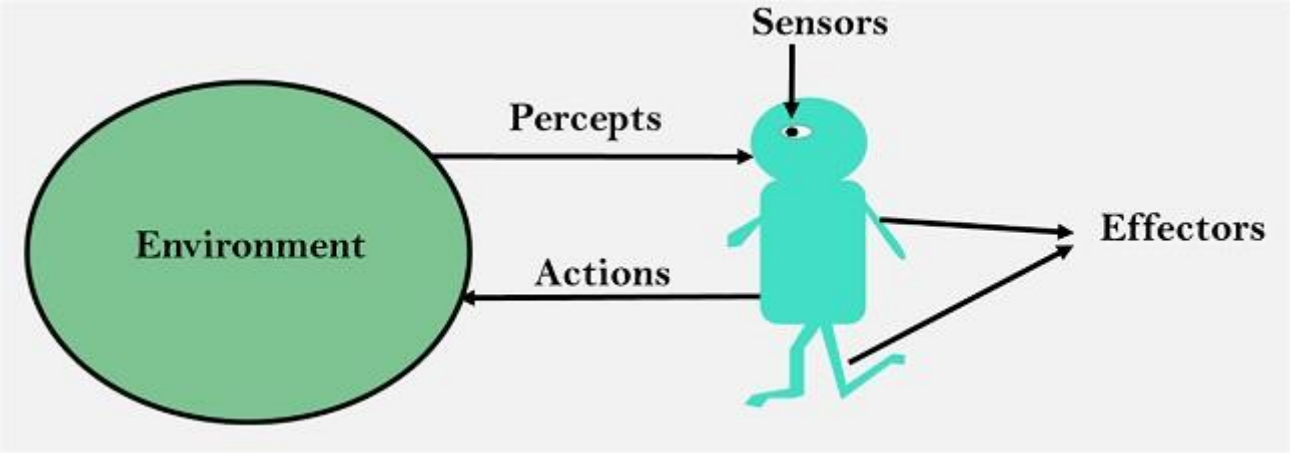
An Agent runs in the cycle of perceiving, thinking, and acting. An agent can be:

- Human-Agent: A human agent has eyes, ears, and other organs which work for sensors and hand, legs, vocal tract work for actuators.
- Robotic Agent: A robotic agent can have cameras, infrared range finder, NLP for sensors and various motors for actuators.
- Software Agent: Software agent can have keystrokes, file contents as sensory input and act on those inputs and display output on the screen.

Hence the world around us is full of agents such as thermostat, cellphone, camera, and even we are also agents.

Before moving forward, we should first know about sensors, effectors, and actuators.
Sensor: Sensor is a device which detects the change in the environment and sends the information to other electronic devices. An agent observes its environment through sensors.
Actuators: Actuators are the component of machines that converts energy into motion. The actuators are only responsible for moving and controlling a system. An actuator can be an electric motor, gears, rails, etc.

Effectors: Effectors are the devices which affect the environment. Effectors can be legs, wheels, arms, fingers, wings, fins, and display screen.



Intelligent Agents:

An intelligent agent is autonomous entities which act upon an environment using sensors and actuators for achieving goals. An intelligent agent may learn from the environment to achieve their goals. A thermostat is an example of an intelligent agent.
Following are the main four rules for an AI agent:

- Rule 1: An AI agent must have the ability to perceive the environment.
- Rule 2: The observation must be used to make decisions.
- Rule 3: Decision should result in an action.
- Rule 4: The action taken by an AI agent must be a rational action.

Rational Agent:

A rational agent is an agent which has clear preference, models uncertainty, and acts in a way to maximize its performance measure with all possible actions.

A rational agent is said to perform the right things. AI is about creating rational agents to use for game theory and decision theoryfor various real-world scenarios.

For an AI agent, the rational action is most important because in AI reinforcement learning algorithm, for each best possible action, agent gets the positive reward and for each wrong action, an agent gets a negative reward.

Rationality:

The rationality of an agent is measured by its performance measure. Rationality can be judged on the basis of following points:

- Performance measure which defines the success criterion.
- Agent prior knowledge of its environment.
- Best possible actions that an agent can perform.
- The sequence of percepts.

Structure of an AI Agent

The task of AI is to design an agent program which implements the agent function. The structure of an intelligent agent is a combination of architecture and agent program. It can be viewed as:

Agent = Architecture + Agent program

Following are the main three terms involved in the structure of an AI agent:
Architecture: Architecture is machinery that an AI agent executes on.

Agent Function: Agent function is used to map a percept to an action.

$$f:P^* \rightarrow A$$

Agent program: Agent program is an implementation of agent function.

An agent program executes on the physical architecture to produce function f.

PEAS Representation

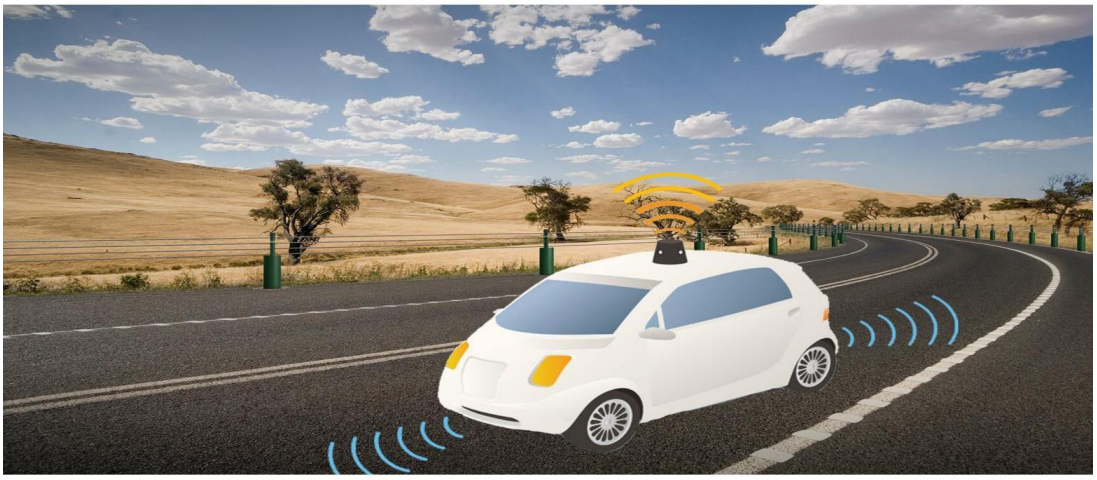
PEAS is a type of model on which an AI agent works upon. When we define an AI agent or rational agent, then we can group its properties under PEAS representation model.

It is made up of four words:

- P: Performance measure
- E: Environment
- A: Actuators
- S: Sensors

Here performance measure is the objective for the success of an agent's behavior.

PEAS for self-driving cars:



Let's suppose a self-driving car then PEAS representation will be:

Performance: Safety, time, legal drive, comfort

Environment: Roads, other vehicles, road signs, pedestrian

Actuators: Steering, accelerator, brake, signal, horn

Sensors: Camera, GPS, speedometer, odometer, accelerometer, sonar.

Example of Agents with their PEAS representation

Agent	Performance measure	Environment	Actuators	Sensors
Medical Diagnose	Healthy patient Minimized cost	Patient Hospital Staff	Tests Treatments	Keyboard (Entry of symptoms)
Vacuum Cleaner	Cleanness Efficiency Battery life Security	Room Table Wood floor Carpet Various obstacles	Wheels Brushes Vacuum Extractor	Camera Dirt detection sensor Cliff sensor Bump Sensor Infrared Wall Sensor
Part picking Robot	Percentage of parts in correct bins.	Conveyor belt with parts, Bins	Jointed Arms Hand	Camera Joint angle sensors.

Agent Environment in AI

An environment is everything in the world which surrounds the agent, but it is not a part of an agent itself. An environment can be described as a situation in which an agent is present.

The environment is where agent lives, operate and provide the agent with something to sense and act upon it. An environment is mostly said to be non-feministic.

Features of Environment

As per Russell and Norvig, an environment can have various features from the point of view of an agent:

1. Fully observable vs Partially Observable
2. Static vs Dynamic
3. Discrete vs Continuous
4. Deterministic vs Stochastic
5. Single-agent vs Multi-agent
6. Episodic vs sequential
7. Known vs Unknown
8. Accessible vs Inaccessible

1. Fully observable vs Partially Observable:
   - If an agent sensor can sense or access the complete state of an environment at each point of time then it is a fully observable environment, else it is partially observable.
   - A fully observable environment is easy as there is no need to maintain the internal state to keep track history of the world.
   - An agent with no sensors in all environments then such an environment is called as unobservable.

2. Deterministic vs Stochastic:
   - If an agent's current state and selected action can completely determine the next state of the environment, then such environment is called a deterministic environment.
   - A stochastic environment is random in nature and cannot be determined completely by an agent.
   - In a deterministic, fully observable environment, agent does not need to worry about uncertainty.

3. Episodic vs Sequential:
   - In an episodic environment, there is a series of one-shot actions, and only the current percept is required for the action.
   - However, in Sequential environment, an agent requires memory of past actions to determine the next best actions.

4. Single-agent vs Multi-agent
    - If only one agent is involved in an environment, and operating by itself then such an environment is called single agent environment.
    - However, if multiple agents are operating in an environment, then such an environment is called a multi-agent environment.
    - The agent design problems in the multi-agent environment are different from single agent environment.

5. Static vs Dynamic:
    - If the environment can change itself while an agent is deliberating then such environment is called a dynamic environment else it is called a static environment.
    - Static environments are easy to deal because an agent does not need to continue looking at the world while deciding for an action.
    - However for dynamic environment, agents need to keep looking at the world at each action.
    - Taxi driving is an example of a dynamic environment whereas Crossword puzzles are an example of a static environment.

6. Discrete vs Continuous:
    - If in an environment there are a finite number of percepts and actions that can be performed within it, then such an environment is called a discrete environment else it is called continuous environment.
    - A chess gamecomes under discrete environment as there is a finite number of moves that can be performed.
    - A self-driving car is an example of a continuous environment.

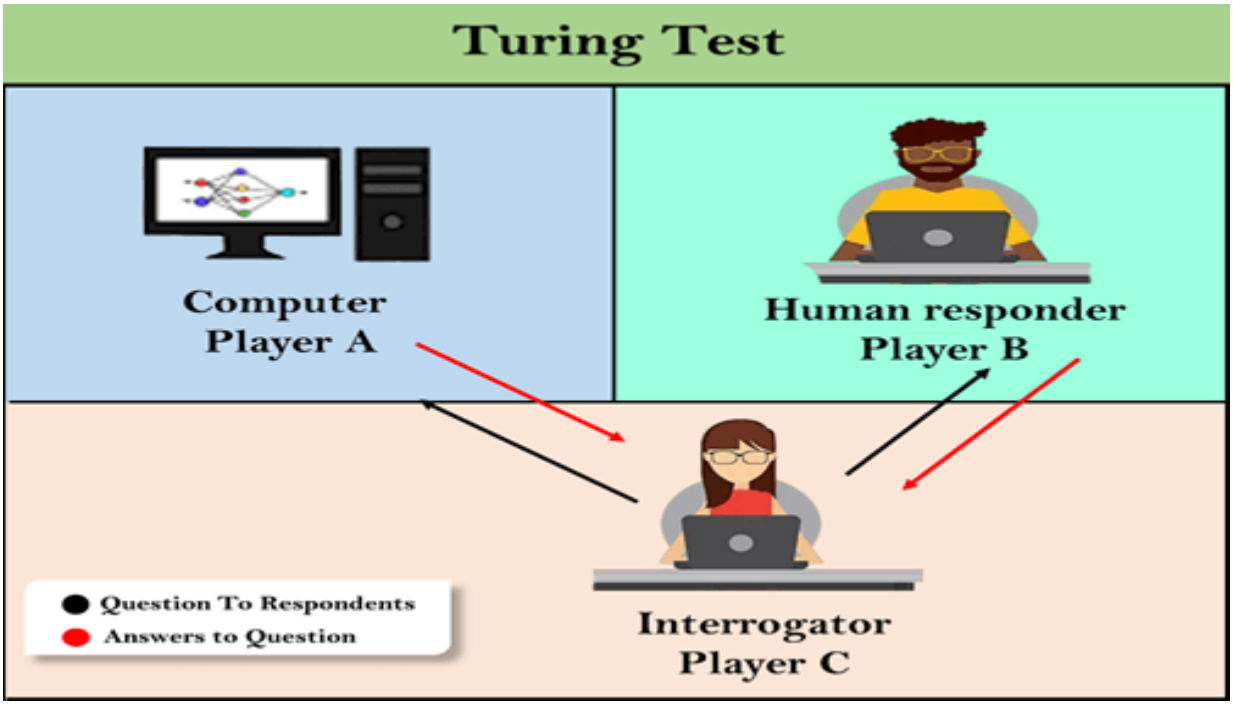
7. Known vs Unknown
    - Known and unknown are not actually a feature of an environment, but it is an agent's state of knowledge to perform an action.
    - In a known environment, the results for all actions are known to the agent. While in unknown environment, agent needs to learn how it works in order to perform an action.
    - It is quite possible that a known environment to be partially observable and an Unknown environment to be fully observable.

8. Accessible vs Inaccessible
    - If an agent can obtain complete and accurate information about the state's environment, then such an environment is called an Accessible environment else it is called inaccessible.
    - An empty room whose state can be defined by its temperature is an example of an accessible environment.
    - Information about an event on earth is an example of Inaccessible environment.

Turing Test in AI

In 1950, Alan Turing introduced a test to check whether a machine can think like a human or not, this test is known as the Turing Test.In this test, Turing proposed that the computer can be said to be an intelligent if it can mimic human response under specific conditions.

Turing Test was introduced by Turing in his 1950 paper, "Computing Machinery and Intelligence," which considered the question, "Can Machine think?"



The Turing test is based on a party game "Imitation game," with some modifications. This game involves three players in which one player is Computer, another player is human responder, and the third player is a human Interrogator, who is isolated from other two

players and his job is to find that which player is machine among two of them.

Consider, Player A is a computer, Player B is human, and Player C is an interrogator. Interrogator is aware that one of them is machine, but he needs to identify this on the basis of questions and their responses.

The conversation between all players is via keyboard and screen so the result would not depend on the machine's ability to convert words as speech.

The test result does not depend on each correct answer, but only how closely its responses like a human answer. The computer is permitted to do everything possible to force a wrong identification by the interrogator. The questions and answers can be like:

Interrogator: Are you a computer?

PlayerA (Computer): No

Interrogator: Multiply two large numbers such as (256896489*456725896)

Player A: Long pause and give the wrong answer.

In this game, if an interrogator would not be able to identify which is a machine and which is human, then the computer passes the test successfully, and the machine is said to be intelligent and can think like a human.

"In 1991, the New York businessman Hugh Loebner announces the prize competition, offering a $100,000 prize for the first computer to pass the Turing test. However, no AI program to till date, come close to passing an undiluted Turing test".

Chatbots to attempt the Turing test:

ELIZA: ELIZA was a Natural language processing computer program created by Joseph Weizenbaum. It was created to demonstrate the ability of communication between machine and humans. It was one of the first chatterbots, which has attempted the Turing Test.

Parry: Parry was a chatterbot created by Kenneth Colby in 1972.Parry was designed to simulate a person with Paranoid schizophrenia(most common chronic mental disorder). Parry was described as "ELIZA with attitude." Parry was tested using a variation of the Turing Test in the early 1970s.

Eugene Goostman: Eugene Goostman was a chatbot developed in Saint Petersburg in 2001. This bot has competed in the various number of Turing Test. In June 2012, at an event, Goostman won the competition promoted as largest-ever Turing test content, in which it has convinced 29% of judges that it was a human. Goostman resembled as a 13-year old virtual boy.

The Chinese Room Argument:

There were many philosophers who really disagreed with the complete concept of Artificial Intelligence. The most famous argument in this list was "Chinese Room."

In the year 1980, John Searle presented "Chinese Room" thought experiment, in his paper "Mind, Brains, and Program," which was against the validity of Turing's Test. According to his argument, "Programming a computer may make it to understand a language, but it will not produce a real understanding of language or consciousness in a computer."
He argued that Machine such as ELIZA and Parry could easily pass the Turing test by manipulating keywords and symbol, but they had no real understanding of language.

So it cannot be described as "thinking" capability of a machine such as a human.

Features required for a machine to pass the Turing test:

- Natural language processing: NLP is required to communicate with Interrogator in general human language like English.
- Knowledge representation: To store and retrieve information during the test.
- Automated reasoning: To use the previously stored information for answering the questions.
- Machine learning: To adapt new changes and can detect generalized patterns.

- Vision (For total Turing test): To recognize the interrogator actions and other objects during a test.
- Motor Control (For total Turing test): To act upon objects if requested.

MCQ

1. Who is known as the -Father of AI"?
   a. Fisher Ada
   b. Alan Turing
   c. John McCarthy
   d. Allen Newell

2. The state-space of the problem includes
   a. Initial state
   b. Action
   c. Transition model
   d. All the above

3. An AI system is composed of
   a. Agent
   b. Environment
   c. Agent and Environment
   d. None of the above

4. Agents can be grouped into classes based on their degree of perceived

   a. Intelligence
   b. Capability
   c. Intelligence and capability
   d. Performance

5. Which agent can work in a partially observable environment, and track the situation?

   a) Simple Reflex Agent
   b) Model-based reflex agent
   c) Goal-based agents
   d) Utility-based agent

6. Which type of agent acts not only for goals but also for the best way to achieve the goal?

    a. Simple Reflex Agent
    b. Model-based reflex agent
    c. Goal-based agents
    d. Utility-based agent

7. Which agent is useful when there are multiple possible alternatives?

    a. Simple Reflex Agent
    b. Model-based reflex agent
    c. Goal-based agents
    d. Utility-based agent

8. Which type of agent works on Condition-action rule?

    a. Simple Reflex Agent
    b. Model-based reflex agent
    c. Goal-based agents
    d. Utility-based agent

9. Rationality can be judged on the basis of

    a. Performance measure which defines the success criterion.
    b. Agent prior knowledge of its environment.
    c. The sequence of percepts.
    d. All the above

10. Which device detects the change in the environment and sends the information to other electronic devices?

    a. Sensors
    b. Actuators
    c. Effectors
    d. All the above

CONCLUSION:

Upon completion of this, Students should be able to

To understand the some fundamentals of AI and AI systems

REFERENCES

1. David Poole, Alan Mackworth, Randy Goebel, -Computational Intelligence: a Logical Approach||, Oxford University Press, 2004.

2. G. Luger, —Artificial Intelligence: Structures and Strategies for Complex Problem Solving||, Fourth Edition, Pearson Education, 2002.

ASSIGNMENT

1. Define intelligent agent.
2. Explain about the foundations of AI.
3. Explain the types of Agent and its environment
4. Explain the structure of agents.
5. Explain about the problem solving agent.

UNIT-II

Uninformed Searching strategies-Breadth First Search, Depth First search, Depth limited search, Iterative deepening search, Bidirectional Search - Avoiding repeated States - Searching with Partial information –Informed search strategies – Greedy Best First Search-A* Search-Heuristic Functions Local Search Algorithms for Optimization Problems-Local search in Continuous Spaces

AIM & OBJECTIVES

- To understand the fundamental concepts of Propagation.
- To understand fading techniques and its types.
- To understand about Antenna Diversity

PRE- REQUISITE: Basic knowledge of Wireless Communication

Search Algorithms in Artificial Intelligence

Search algorithms are one of the most important areas of Artificial Intelligence. This topic will explain all about the search algorithms in AI.

Problem-solving agents:

In Artificial Intelligence, Search techniques are universal problem-solving methods. Rational agents or Problem-solving agents in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

Search Algorithm Terminologies:

- Search: Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:

    a. Search Space: Search space represents a set of possible solutions, which a system may have.

b. Start State: It is a state from where agent begins the search.

c. Goal test: It is a function which observe the current state and returns whether the goal state is achieved or not.

- Search tree: A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.

- Actions: It gives the description of all the available actions to the agent.

- Transition model: A description of what each action do, can be represented as a transition model.

- Path Cost: It is a function which assigns a numeric cost to each path.

- Solution: It is an action sequence which leads from the startnode to the goal node.

- Optimal Solution: If a solution has the lowest cost among all solutions.

Properties of Search Algorithms:

Following are the four essential properties of search algorithms to compare the efficiency of these algorithms:

Completeness: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.

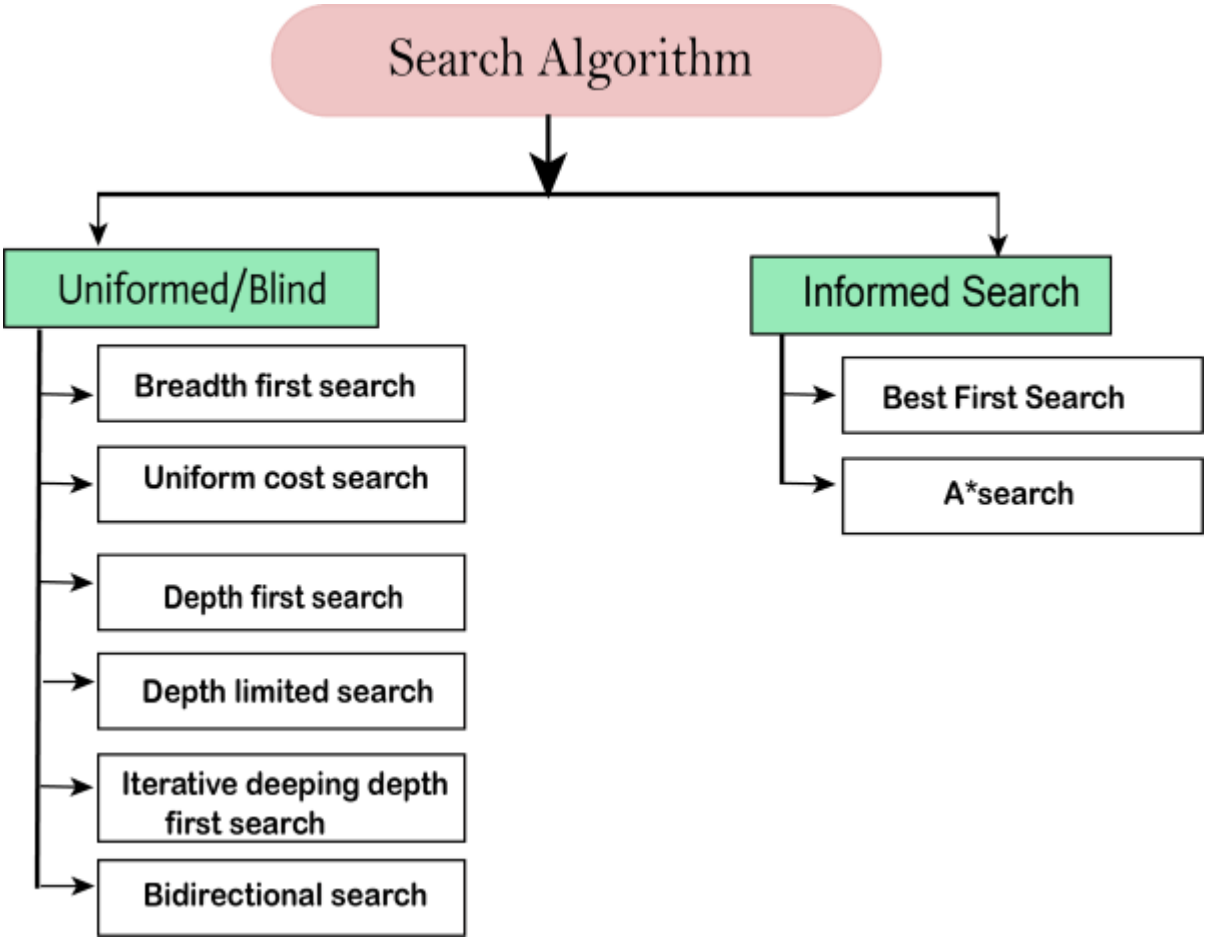
Optimality: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.

Time Complexity: Time complexity is a measure of time for an algorithm to complete its task.

Space Complexity: It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



Uninformed/Blind Search:

The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force wayas it only includes information about how to traverse the tree and how to identify leaf and goal nodes. Uninformed search applies a wayin which search tree is searched without any information about thesearch space like initial state operators and test for the goal, so it isalso called blind search. It examines each node of the tree until it achieves the goal node.

It can be divided into five main types:

- Breadth-first search
- Uniform cost search
- Depth-first search
- Iterative deepening depth-first search
- Bidirectional Search

Informed Search

Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy. Informed search is also called a Heuristic search.

A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.

Informed search can solve much complex problem which could not be solved in another way.

An example of informed search algorithms is a traveling salesman problem.

1. Greedy Search
2. A* Search

Uninformed Search Algorithms

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way.

Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Following are the various types of uninformed search algorithms:

1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search

5. Uniform cost search
6. Bidirectional Search

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.

- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.

- The breadth-first search algorithm is an example of a general-graph search algorithm.

- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.

- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages:

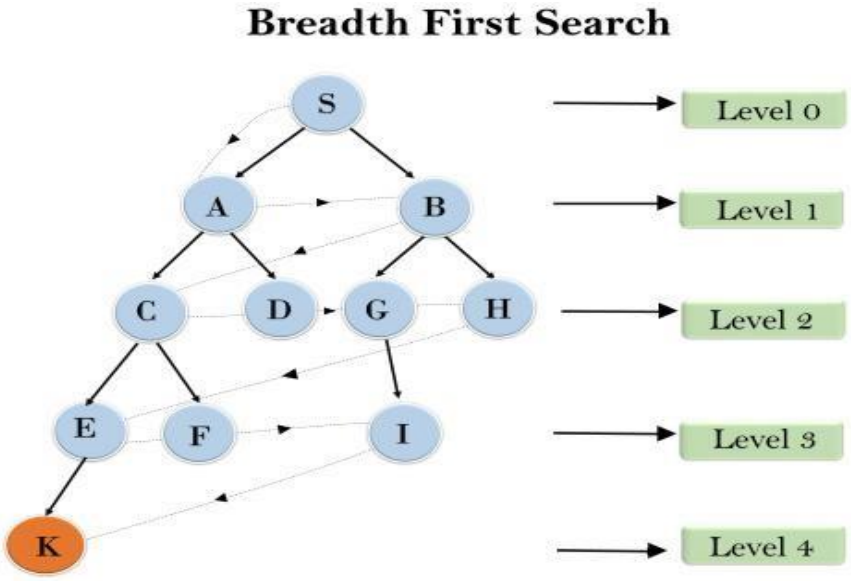
- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.

- BFS needs lots of time if the solution is far away from the root node.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

1. S---> A--->B---->C--->D---->G--->H--->E---->F---->I >K



Time Complexity:

Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.
T (b) = $1+b^2+b^3+.....+ b^d= O (b^d)$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFSalgorithm.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

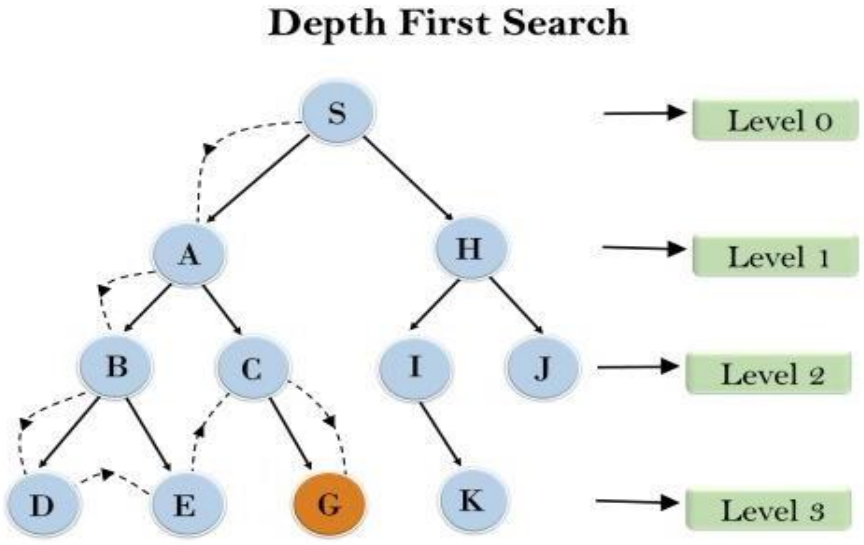
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm.

It is given by:

$$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$$

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is O(bm).

Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that problem does not have any solution.

- Cutoff failure value: It defines no solution for the problem within a given depth limit.

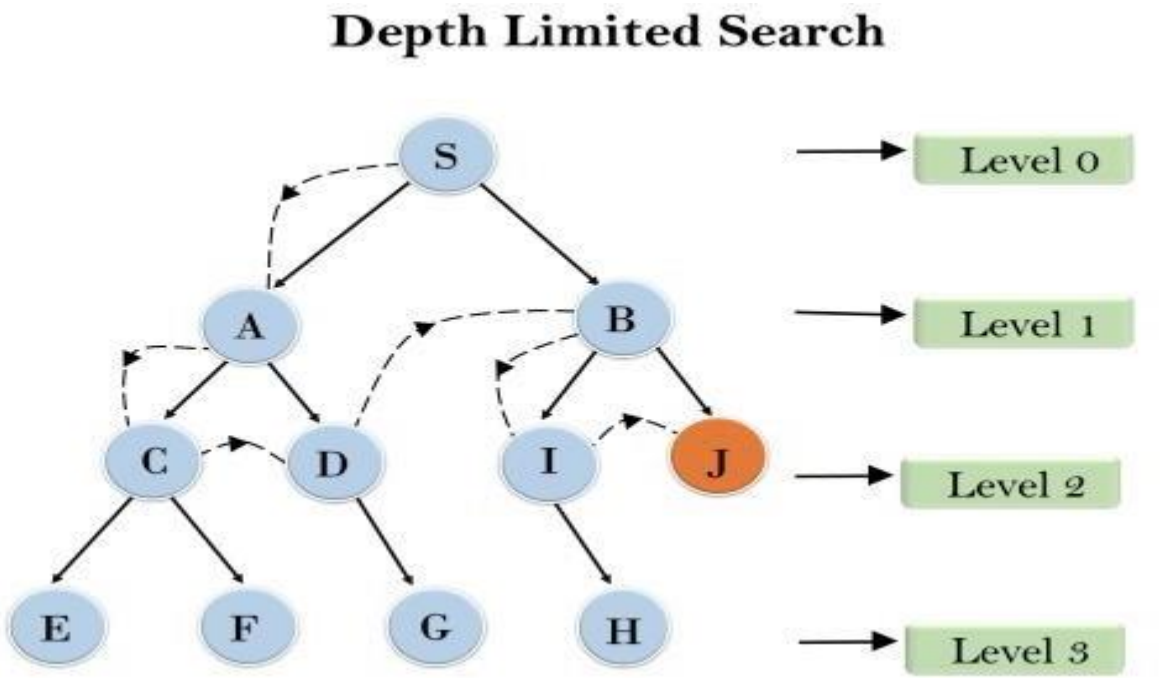
Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example:



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is $O(b^{\ell})$.

Space Complexity: Space complexity of DLS algorithm is $O(b \times \ell)$.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.

4. Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

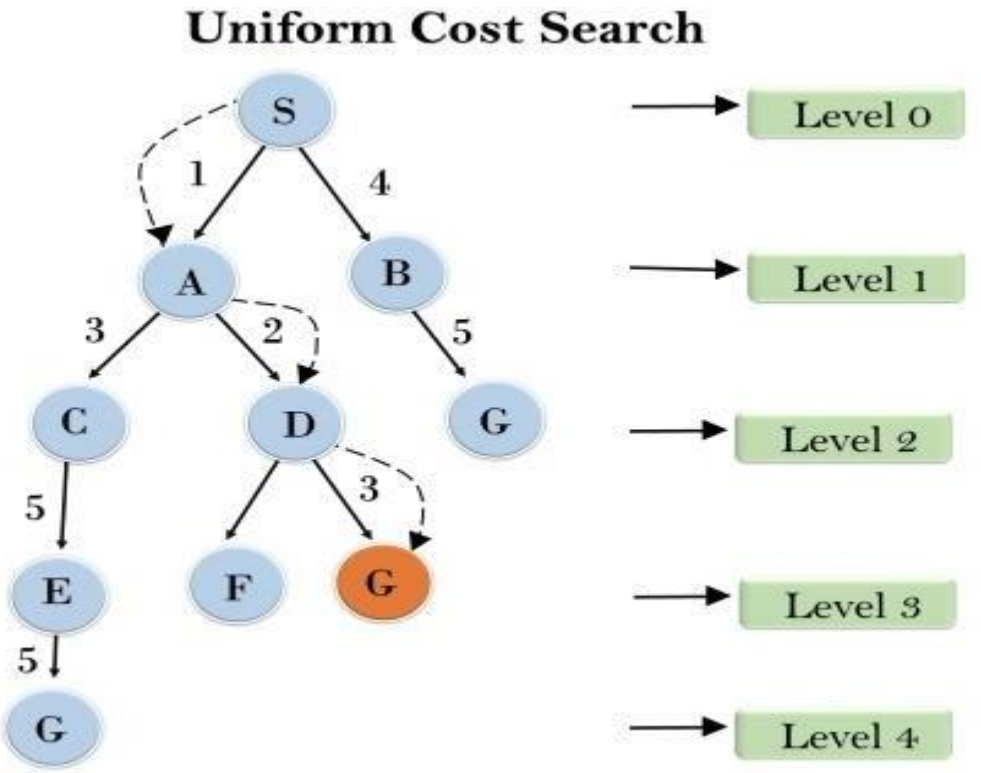
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Let C* is Cost of the optimal solution, and ε is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is$O(b^{1 + [C^*/\varepsilon]})$/.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $O(b^{1 + [C^*/\varepsilon]})$.

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. Iterative deepening depth-first Search:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

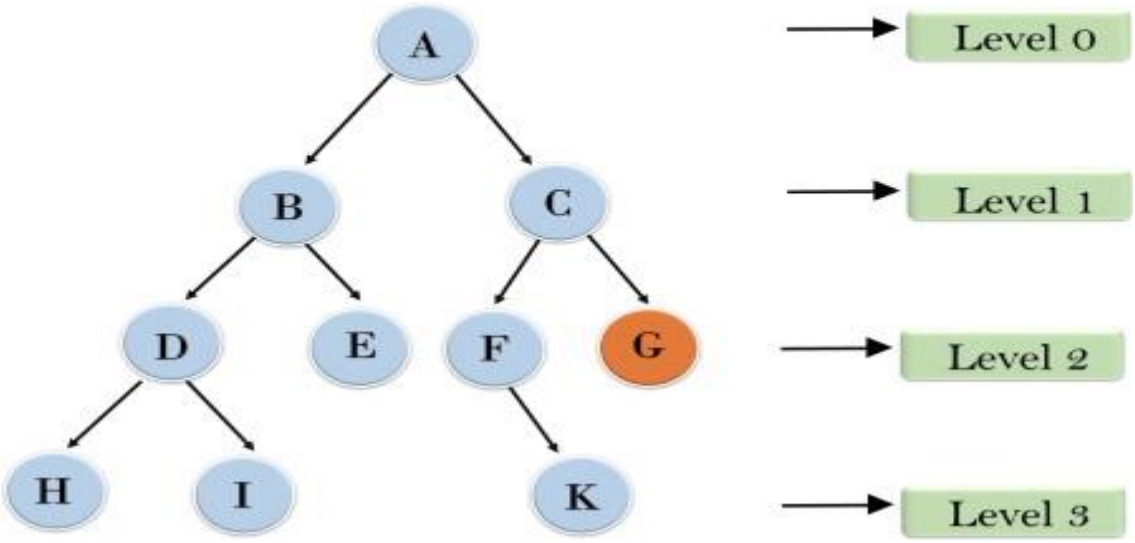
Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure is showing the iterative deepening depth- first search. IDDFS algorithm performs various iterations until itdoes not find the goal node. The iteration performed by the algorithm is given as:

Iterative deepening depth first search



1'stIteration----- >A
2'ndIteration--- >A,B,C
3'rdIteration ----->A,B,D,E,C,F,G
4'thIteration ----->A,B,D,H,I,E,C,F,K,G

In the fourth iteration, the algorithm will find the goal node.

Completeness:
This algorithm is complete is if the branching factor is finite.

Time Complexity:
Let's suppose b is the branching factor and depth is d then the worst-case time complexity is $O(b^d)$.

Space Complexity:
The space complexity of IDDFS will be O(bd).

Optimal:
IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

6. Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.Bidirectional search can use search techniques such as BFS,DFS, DLS, etc.

Advantages:

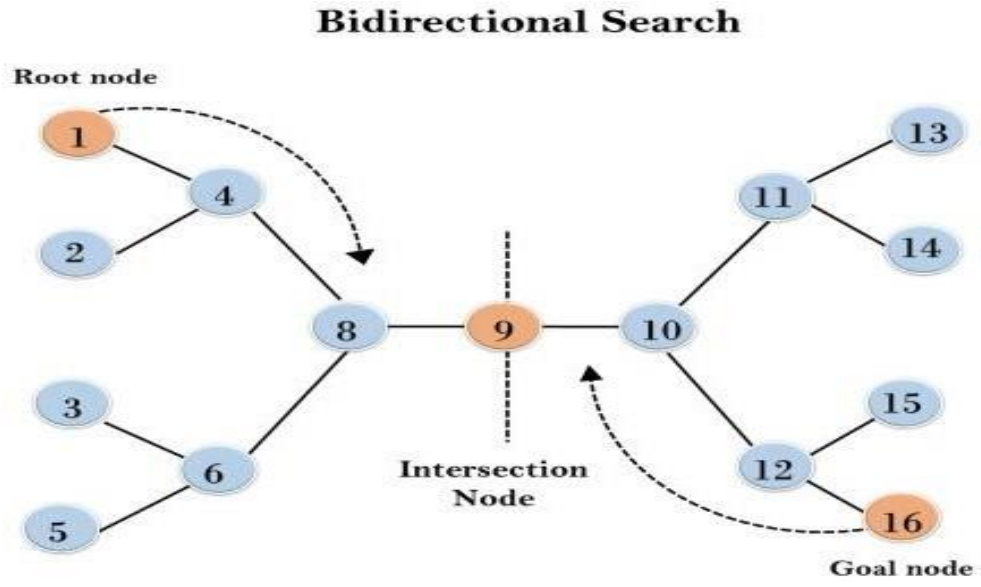
- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It startstraversing from node 1 in the forward direction and starts from goal node 16 in the backward direction. The algorithm terminates at node9 where two searches meet.



Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is $O(b^d)$.

Space Complexity: Space complexity of bidirectional search is $O(b^d)$.

Optimal: Bidirectional search is Optimal.

Informed Search Algorithms

So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space. But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge helps agents to explore less to the search space and find more efficiently the goal node.

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.

The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as: $h(n) <= h^*(n)$

Here h(n) is heuristic cost, and $h^*(n)$ is the estimated cost.

Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPENlist, it places nodes which have yet not been expanded.

On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

In the informed search we will discuss two main algorithms which are given below:

- Best First Search Algorithm(Greedy search)
- A* Search Algorithm

1.) Best-first Search Algorithm (Greedy Search):

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-firstsearch and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

$$f(n)= g(n).$$

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

Best first search algorithm:

- Step 1: Place the starting node into the OPEN list.
- Step 2: If the OPEN list is empty, Stop and return failure.
- Step 3: Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.

- Step 4: Expand the node n, and generate the successors of node n.

- Step 5: Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.

- Step 6: For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.

- Step 7: Return to Step 2.

Advantages:

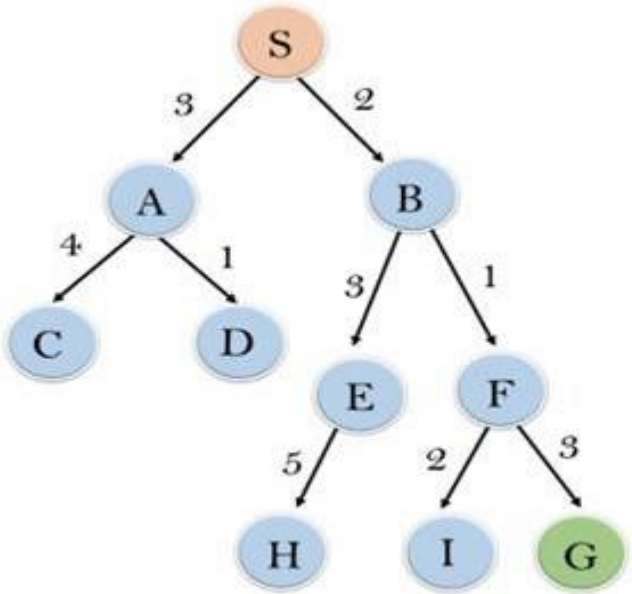
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

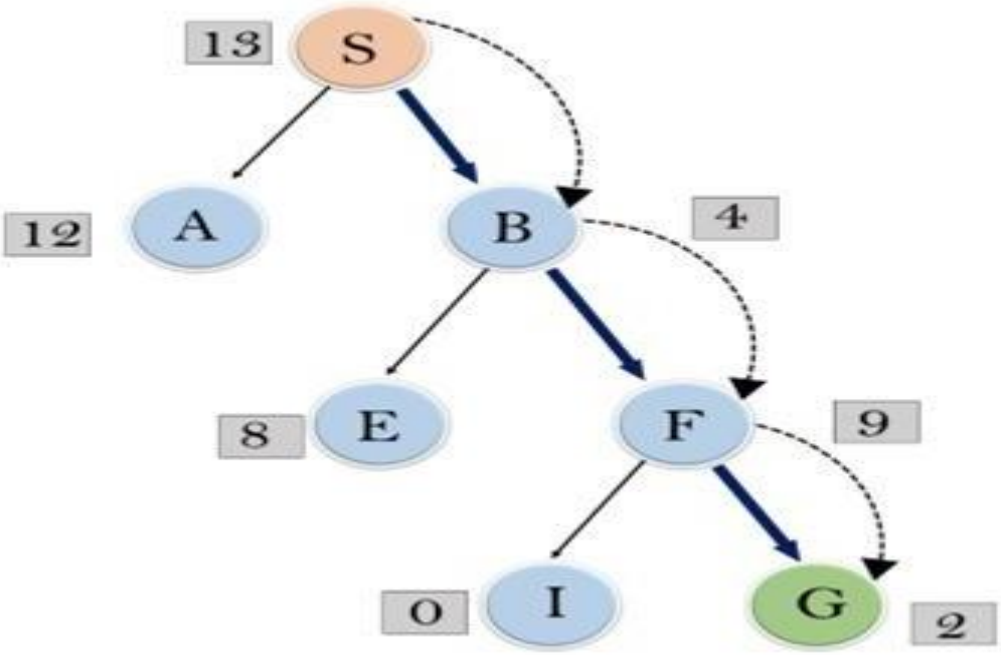
Example:

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

In this search example, we are using two lists which are OPEN and CLOSED Lists. Following are the iteration for traversing the above example.



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration2: Open[E,F,A],Closed[S,B]
: Open [E, A], Closed [S, B, F]

Iteration3: Open[I,G,E,A],Closed[S,B,F]
: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: S----> B----->F----> G

Time Complexity: The worst case time complexity of Greedy best first search is $O(b^m)$.

Space Complexity: The worst case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

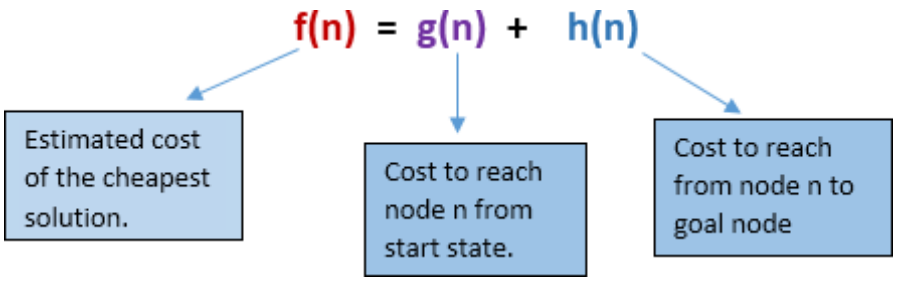
Optimal: Greedy best first search algorithm is not optimal.

2.) A* Search Algorithm:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster.

A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n). In A* search algorithm, we use search heuristic as well as the cost to reach the node.

Hence we can combine both costs as following, and this sum is called as a fitness number.



Algorithm of A* search:

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n')value.

Step 6: Return to Step 2.

Advantages:

- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

Disadvantages:

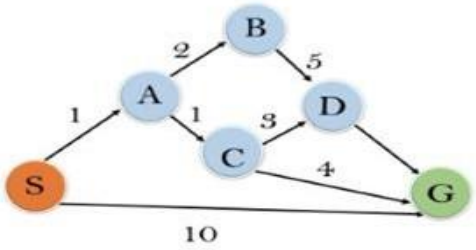
- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.

- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example:

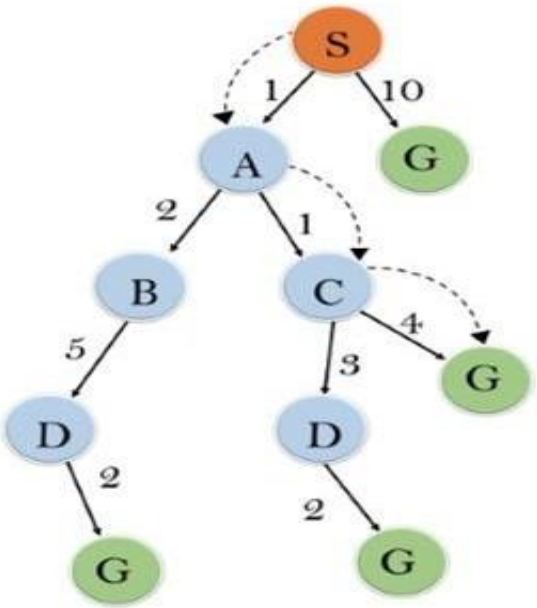
In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

Here we will use OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution :



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as S--->A--->C--->G it provides the optimal path with cost 6.

Points to remember:

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)<="" li="">

Complete: A* algorithm is complete as long as:

- Branching factor is finite.
- Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

- Admissible: the first condition requires for optimality is thath(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- Consistency: Second required condition is consistency for only A* graph-search.

If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is O(b^d)

Hill Climbing Algorithm in Artificial Intelligence

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.

- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.

- A node of hill climbing algorithm has two components which are state and value.

- Hill Climbing is mostly used when a good heuristic is available.

- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing:

Following are some main features of Hill Climbing Algorithm:

- Generate and Test variant: Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.

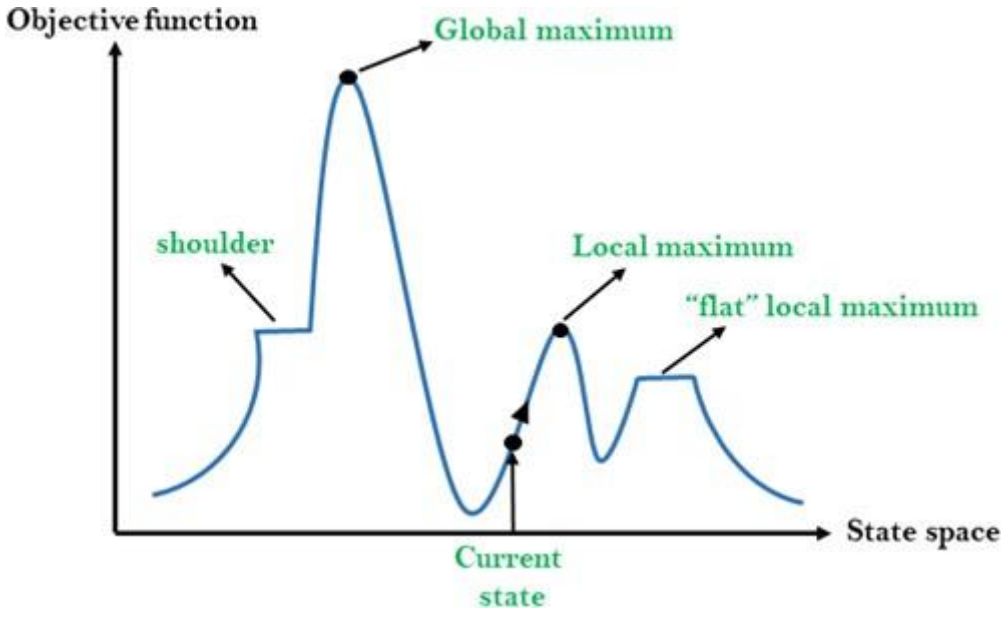
- Greedy approach: Hill-climbing algorithm search moves in the direction which optimizes the cost.
- No backtracking: It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.

On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum.

If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.



Different regions in the state space landscape:

Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.
C++ vs Java

Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

Current state: It is a state in a landscape diagram where an agent is currently present.

Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.

Shoulder: It is a plateau region which has an uphill edge.

Types of Hill Climbing Algorithm:

- Simple hill Climbing:
- Steepest-Ascent hill-climbing:
- Stochastic hill Climbing:

1. Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as acurrent state. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state.

This algorithm has the following features:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

Algorithm for Simple Hill Climbing:

- Step 1: Evaluate the initial state, if it is goal state then return success and Stop.
- Step 2: Loop Until a solution is found or there is no new operator left to apply.
- Step 3: Select and apply an operator to the current state.
- Step 4: Check new state:
    a. If it is goal state, then return success and quit.
    b. Else if it is better than the current state then assign new state as a current state.
    c. Else if not better than the current state, then return to step2.
- Step 5: Exit.

2. Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches formultiple neighbors

Algorithm for Steepest-Ascent hill climbing:

- Step 1: Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- Step 2: Loop until a solution is found or the current state does not change.
    a. Let SUCC be a state such that any successor of the current state will be better than it.
    b. For each operator that applies to the current state:
        a. Apply the new operator and generate a new state.
        b. Evaluate the new state.
        c. If it is goal state, then return it and quit, else compare it to the SUCC.
        d. If it is better than SUCC, then set new state as SUCC.
        e. If the SUCC is better than the current state, then set current state to SUCC.
- Step 5: Exit.

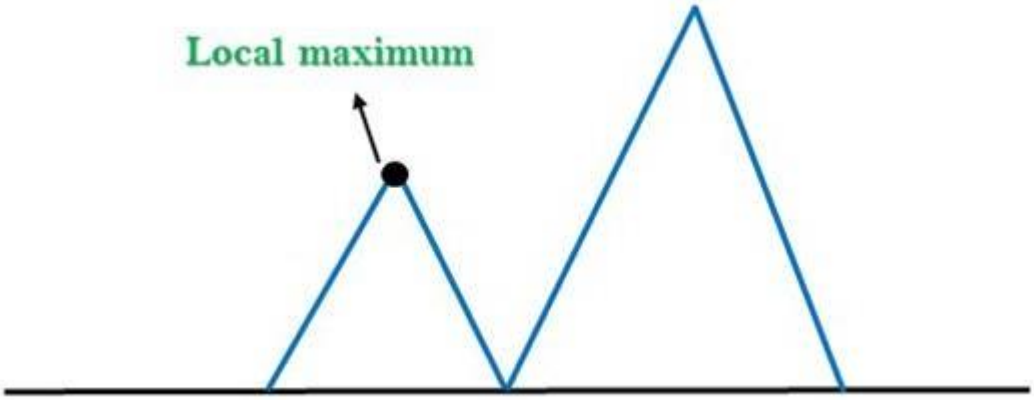
3. Stochastic hill climbing:

Stochastic hill climbing does not examine for all its neighbor before moving. Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Problems in Hill Climbing Algorithm:

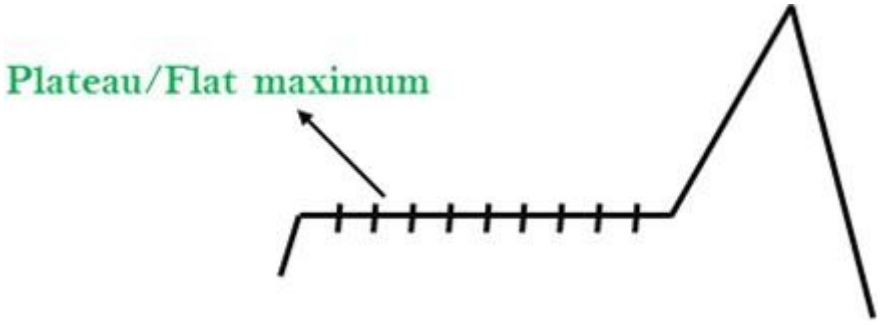
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

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.



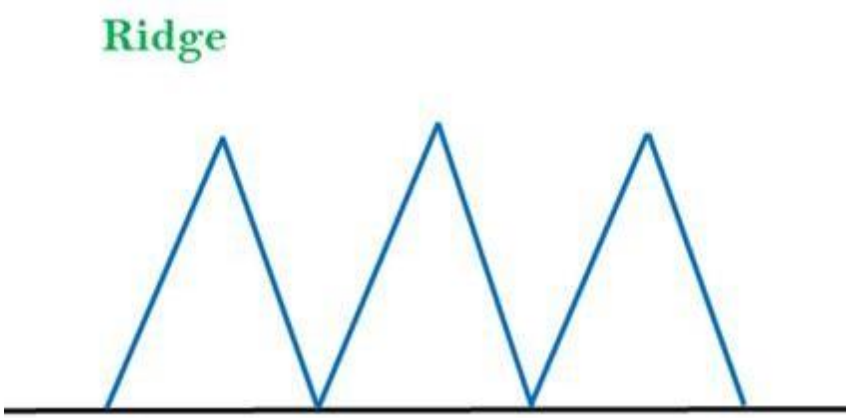
2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

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.



Simulated Annealing:

A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum. And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient.

Simulated Annealing is an algorithm which yields both efficiency and completeness.

In mechanical term Annealing is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state. The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path.
Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

MCQ

1. Problem solving agents are also called as

    a. Simple agent
    b. Reflex agent
    c. Rational agent
    d. Goal based agent

2. Which represents a set of possible solutions, which a system may have?

    a) Search space
    b) Start state
    c) Search tree
    d) Goal test

3. If a solution has the lowest cost among all solutions, then it is called as

    a) Optimal solution
    b) Path cost
    c) Transition model
    d) None of the above

4. Which search does not contain any domain knowledge such as closeness, the location of the goal?

    a) Uninformed search
    b) Informed search
    c) Blind search
    d) Both A and C

5. Which algorithm is a combination of DFS and BFS algorithms?

    a) Iterative deepening depth-first Search
    b) Simple Search
    c) Complex search
    d) Bidirectional search

6. If the environment is not fully observable or deterministic, then which type of problems will occur?

   a) Contingency problem
   b) Conformant problem
   c) Sensorless problems
   d) All the above

7. The Estimated cost of cheapest solution f(n) =

   a) h(n)
   b) g(n)
   c) h(n) * g(n)
   d) h(n) + g(n)

8. Which is defined by the value of the objective function or heuristic cost function?

   a) Location
   b) Elevation
   c) Both
   d) None of the Above

9. Which type of Search Algorithm requires less computation?

   a) Informed search
   b) Uninformed search
   c) Both
   d) None of the above

10. A node of hill climbing algorithm has

a) State components
b) Value components
c) Both
d) None of the above

CONCLUSION:

Upon completion of this, Students should be able to

- ❖ Understand the AI systems able to exhibit limited human-like abilities, particularly in the form of problem solving by search

REFERENCES

1. David Poole, Alan Mackworth, Randy Goebel, -Computational Intelligence: a Logical Approach||, Oxford University Press, 2004.
2. G. Luger, —Artificial Intelligence: Structures and Strategies for Complex Problem Solving||, Fourth Edition, Pearson Education, 2002.

ASSIGNMENT

1. Explain about the Informed Search Algorithm.
2. Explain about the Uninformed Search Algorithm.
3. Explain about Local search Algorithm.
4. Explain about Local search in continuous spaces.
5. Explain about optimization problems.

Online Search Agents and Unknown Environments

An online search agent operates by interleaving computation and action: first it takes an action and then it observes the environment and computes the next action. Online search is a good idea in dynamic or semi dynamic domains-domains where there is a penalty for sitting around and computing too long. Online search is an even better idea for stochastic domains.
(The term "online" is commonly used in computer science to refer to algorithms that must process input data as they are received, rather than waiting for the entire input data set to become available.)

In general, an offline search would have to come up with an exponentially large contingency plan that considers all possible happenings, while an online search need only consider what actually does happen.

For example,
A chess playing agent is well-advised to make its first move long before it has figured out the complete course of the game. Online search is a necessary idea for an exploration problem, where the states and actions are unknown to the agent. An agent in this state

of Ignorance must use its actions as experiments to determine what to do next, and hence must interleave computation and action.
The canonical example of online search is a robot that is placed in a new building and must explore it to build a map that it can use for getting from A to B. Methods for escaping from labyrinths-required knowledge for aspiring heroes of antiquity-are also examples of online search algorithms. Spatial exploration is not the only form of exploration, however.

Consider a newborn baby: it has many possible actions, but knows the outcomes of none of them, and it has experienced only a few ofthe possible states that it can reach. The baby's gradual discovery ofhow the world works is, in part, an online search process.

Online search problems

An online search problem can be solved only by an agent executing actions, rather than by a purely computational process. We will assume that the agent knows just the following:

ACTIONS(S), which returns a list of actions allowed in state s;

The step-cost function c(s, a, s^l)-note that this cannot be used until the agent knows that s^l is the outcome; and

GOAL-TEST(S).

Note in particular that the agent cannot access the successors of a state except by actually trying all the actions in that state. For example, in the maze problem shown in Figure, the agent does not know that going Up from (1,l**)** leads to (1,2); nor, having done that, does it know that going Down will take it back to (1,l). This degree of ignorance can be reduced in some applications-for example, a robot explorer might know how its movement actions work and be ignorantonly of the locations of obstacles.
We will assume that the agent can always recognize a state that it has visited before, and we will assume that the actions are deterministic. Finally, the agent might have access to an, admissible heuristic function h(s) that estimates the distance from the current state to a goal state. For example, in Figure, the agent might knowthe location of the goal and be able to use the Manhattan distance heuristic.

Typically, the agent's objective is to reach a goal state while minimizing cost. (Another possible objective is simply to explore the entire environment.) The cost is the total path cost of the path that the agent actually travels. It is common to compare this cost with the path cost of the path the agent would follow if it knew the search space in advance- that is, the actual shortest path (or shortest complete exploration). In the language of online algorithms
this is called the competitive ratio; we would like it to be as small as possible. Although this sounds like a reasonable request, it is easy to see that the best achievable competitive ratio is infinite in some cases. For example, if some actions are irreversible, the online search might accidentally reach a dead-end state from which no goal state is reachable.

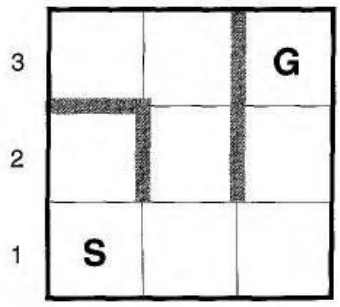
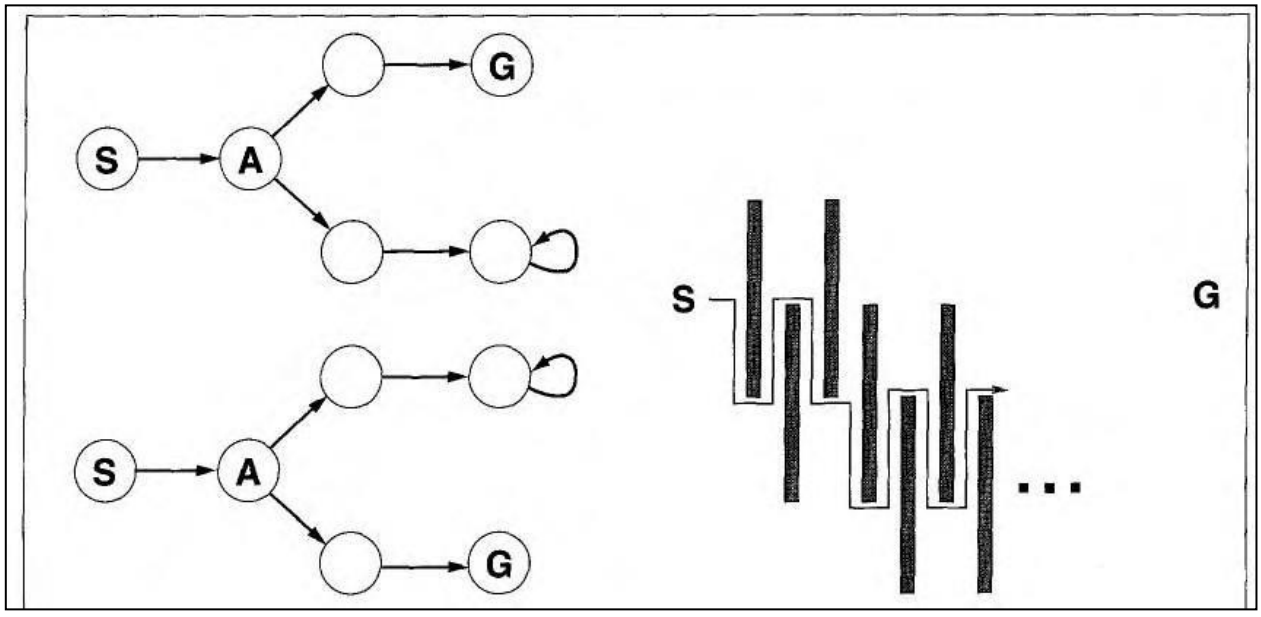


Figure: A simple maze problem.

The agent starts at S and must reach G, but knows nothing of the environment.



(a) (b)

(a) Two state spaces that might lead an online search agent into a dead end.Any given agent will fail in at least one of these spaces.

(b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

Perhaps you find the term "accidentally" unconvincing-after all, there might be an algorithm that happens not to take the dead-end path as it explores. Our claim, to be more precise, is that no algorithm can avoid dead ends in all state spaces.
Consider the two dead-end state spaces in Figure (a). To an online search algorithm that has visited states S and A, the two state spaces look identical, so it must make the same decision in both. Therefore, it will fail in one of them. This is an example of an adversary argument-we can imagine an adversary that constructs the state space while the agent explores it and can put the goals and dead ends wherever it likes.

Dead ends are a real difficulty for robot exploration--staircases, ramps, cliffs, and all kinds of natural terrain present opportunities for irreversible actions. To make progress, we will simply assume that the state space is safely explorable-that is, some goal state is reachable from every reachable state. State spaces with reversible actions, such as mazes and 8-puzzles, can be viewed as undirected graphs and are clearly safely explorable.

Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost. This is easy to show in environments with irreversible actions, but in fact it remains true for the reversible case as well, as Figure (b) shows. For this reason, it is common to describe the performance of online search algorithms in terms of the size of the entire state space rather than just the depth of the shallowest goal.

Online search agents

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The current map is used to decide where to go next.

This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms wehave seen previously.

For example, offline algorithms such as A* have the ability to expand a node in one part of the space and then immediately expand a node in another part of the space, because node expansion involves simulated rather than real actions. An online algorithm, on the other hand, can expand only a node that it physically occupies. To avoid traveling all the way across the tree to expand the next node, it seems better to expand nodes in a local order.

Depth-first search has exactly this property, because (except when backtracking) the next node expanded is a child of the previous node expanded.

An online depth-first search agent is shown in Figure. This agentstores its map in a table, result [a, s], that records the state resulting from executing action a in state s. whenever an action from the current state has not been explored, the agent tries that action.

The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack physically.

In depth-first search, this means going back to the state from which the agent entered the current state most recently. That is achieved by keeping a table that lists, for each state, the predecessor states to which the agent has riot yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

The progress of ONLINE-DFS-AGENT can be traced when applied to the maze given in Figure. It is fairly easy to see that the agent will, in the worst case, end up traversing every link in the state space exactly twice.

For exploration, this is optimal; for finding a goal, on the other hand, the agent's competitive ratio could be arbitrarily bad if it goes off on a long excursion when there is a goal right next to the initial state. An online variant of iterative deepening solves this problem; for an environment that is a uniform tree, the competitive ratio of such an agent is a small constant.

Because of its method of backtracking, ONLINE-DFS-AGENT works only in state spaces where the actions are reversible. There are slightly more complex algorithms that work in general state spaces, but no such algorithm has a bounded competitive ratio.

```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          unexplored, a table that lists, for each visited state, the actions not yet tried
          unbacktracked, a table that lists, for each visited state, the backtracks not yet tried
          s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state then unexplored[s'] ← ACTIONS(s')
  if s is not null then do
      result[a, s] ← s'
      add s to the front of unbacktracked[s']
  if unexplored[s] is empty then
      if unbacktracked[s] is empty then return stop
      else a ← an action b such that result[b, s'] = POP(unbacktracked[s'])
  else a ← POP(unexplored[s'])
  s ← s'
  return a
```

Figure: An online search agent that uses depth-first exploration.

The agent is applicable only in bidirected search spaces.

Online local search

Like depth-first search, **hill-climbing search** has the property of locality in its node expansions. In fact, because it keeps just one current state in memory, hill-climbing search is already an online search algorithm! Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go. Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

Instead of random restarts, one might consider using a random walk to explore the environment. A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried. It is easy to prove that a random walk will eventually find a goal or complete its exploration, provided that the space is finite.15 On the other hand, the process can be very slow. Figure shows an environment in which a random

walk will take exponentially many steps to find the goal, because, at each step, backward progress is twice as likely as forward progress.

The example is contrived, of course, but there are many real-world state spaces whose topology causes these kinds of "traps" for random walks.
Augmenting hill climbing with memory rather than randomness turns out to be a more effective approach. The basic idea is to store a "current best estimate" H(s) of the cost to reach the goal from each state that has been visited. H(s) starts out being just the heuristic

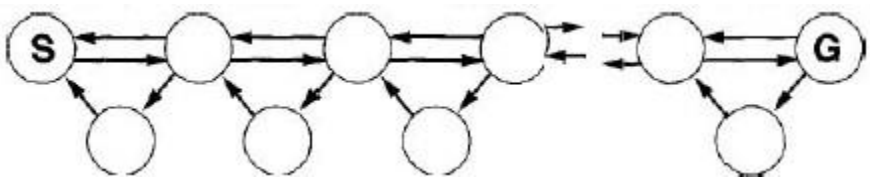


Figure - An environment in which a random walk will take exponentially many steps to find the goal.

estimate h(s) and is updated as the agent gains experience in the state space. Figure shows a simple example in a one-dimensional state space. In (a), the agent seems to be stuck in a flat local minimum at the shaded state. Rather than staying where it is, the agent should follow what seems to be the best path to the goal based on the current cost estimates for its neighbors. The estimated cost to reach the goal through a neighbor s is the cost to get to s plus the estimated cost to get to a goal from there-that is, c(s, a, s) + H(st).

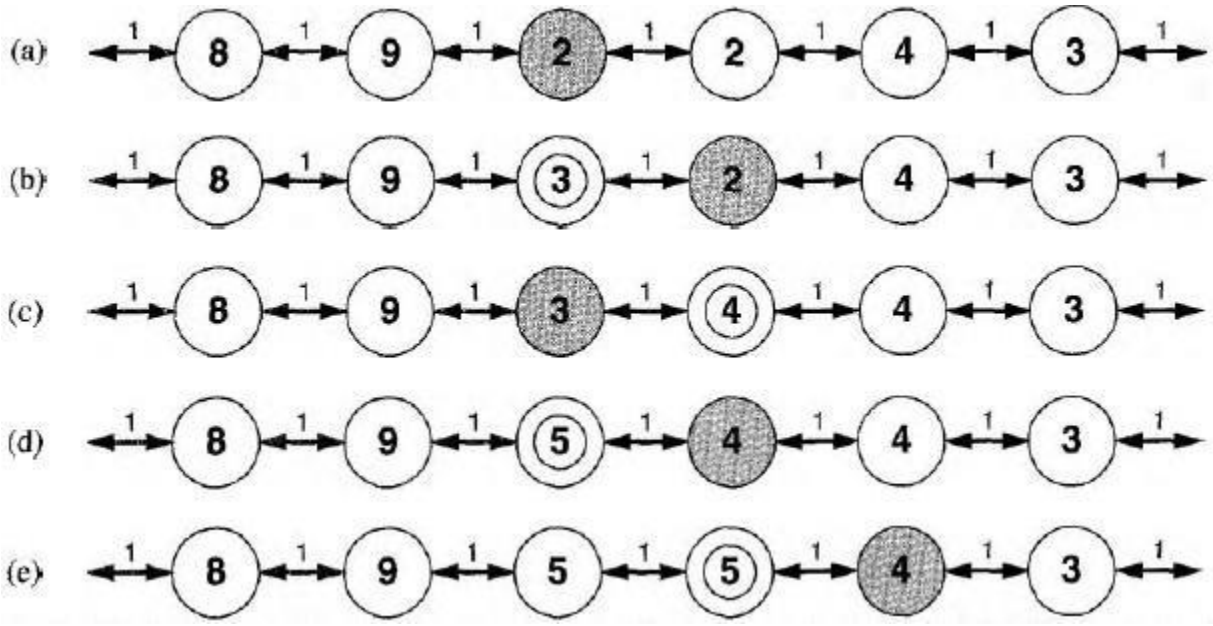
In the example, there are two actions with estimated costs 1 + 9 and 1 + 2, so it seems best to move right. Now, it is clear that the cost estimate of 2 for the shaded state was overly optimistic. Since the best move cost 1 and led to a state that is at least 2 steps from a goal, the shaded state must be at least 3 steps from a goal, so its H should be updated accordingly, as shown in Figure. Continuing this process, the agent will move back and forth twice more, updating H each time and "flattening out" the local minimum until it escapes to the right.

An agent implementing this scheme, which is called learning real- time A* (LRTA*), is shown in Figure. Like ONLINE-DFS-AGENT, itbuilds a map of the environment using the result table. It updates the cost estimate for the state it has just left and then chooses the

"apparently best" move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state s are always assumed to lead immediately to the goal with the least possible cost, namely h(s). This optimism under uncertainty encourages the agent to explore new, possibly promising paths.

Learning in online search

The initial ignorance of online search agents provides several opportunities for learning. First, the agents learn a "map" of the environment-more precisely, the outcome of each action in eachstate-simply by recording each of their experiences. (Notice that the assumption of deterministic environments means that one experience is enough for each action.) Second, the local search agents acquire more accurate estimates of the value of each state by using local updating rules.



Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with H(s), the current cost estimate to reach a goal, and each arc is labeled with its step cost. The shaded state marks the location of the agent, and the updated values at each iteration are circled.

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  static: result, a table, indexed by action and state, initially empty
          H, a table of cost estimates indexed by state, initially empty
          s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  unless s is null
      result[a, s] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[b, s], H)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[b, s'], H)
  s ← s'
  return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
  if s' is undefined then return h(s)
  else return c(s, a, s') + H[s']
```

LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

These updates eventually converge to exact values for every state, provided that the agent explores the state space in the right way. Once exact values are known, optimal decisions can be taken simply by moving to the highest-valued successor-that is, pure hill climbing is then an optimal strategy.

If you followed our suggestion to trace the behavior of ONLINE-DFS-AGENT in the environment, you will have noticed that the agent is not very bright. For example, after it has seen that the Up action goes from (1,l) to (1,2), the agent still has no idea that the Down action goes back to (1,1), or that the Up action also goes from (2,l) to (2,2), from (2,2) to (2,3), and so on. In general, we would like the agent to learn that Up increases the y-coordinate unless there is a wall in the way, which Down reduces it, and so on. For this to happen, we needtwo things. First, we need a formal and explicitly representation for these kinds of general rules; so far, we have hidden the information inside the black box called the successor function.

ARTIFICIAL INTELLIGENCE

UNIT – III

Syllabus

Reinforcement Learning: Introduction, Passive Reinforcement Learning, Active Reinforcement Learning, Generalization in Reinforcement Learning, Policy Search, applications of RL .

Natural Language Processing: Language Models, Text Classification, Information Retrieval, Information Extraction.

Chapter – 1

Reinforcement Learning

3.1.1. INTRODUCTION:

- A supervised learning agent needs to be told the correct move for each position it encounters, but such feedback is seldom available.
- In the absence of feedback from a teacher, an agent can learn a transition model for its own moves and can perhaps learn to predict the opponent's moves, but *without some feedback about what is good and what is bad, the agent will have no grounds for deciding which move to make.*
- The agent needs to know that something good has happened when it (accidentally) checkmates the opponent, and that something bad has happened when it is checkmated—or vice versa, if the game is suicide chess.

  This kind of feedback is called a reward, or reinforcement.
- In games like chess, the reinforcement is received only at the end of the game. In other environments, the rewards come more frequently.
- In ping-pong, each point scored can be considered a reward; when learning to crawl, any forward motion is an achievement.
- In animals seem to be hardwired to recognize pain and hunger as negative rewards and pleasure and food intake as positive rewards.
- Rewards, where they served to define optimal policies in Markov decision processes (MDPs).
- An optimal policy is a policy that maximizes the expected total reward. The task of **reinforcement learning** is to use observed rewards to learn an optimal (or nearly optimal) policy for the environment.
- Imagine playing a new game whose rules you don't know; after a hundred or so moves, your opponent announces, "You lose." This is reinforcement learning in a nutshell.
- In many complex domains, reinforcement learning is the only feasible way to train a program to perform at high levels.

  Reinforcement learning might be considered to encompass all of AI: an agent is placed in an environment and must learn to behave successfully therein. To keep the concept manageable, we will concentrate on simple environments and simple agent designs.
- Thus, the agent faces an unknown Markov decision process. We will consider three of the agent designs first :

→ A **utility-based agent** learns a utility function on states and uses it to select actions that maximize the expected outcome utility.

→ A **Q-learning** agent learns an **action-utility function**, or **Q-function**, giving the ex-pected utility of taking a given action in a given state.

A **reflex agent** learns a policy that maps directly from states to actions.

- A utility-based agent must also have a model of the environment in order to make decisions, because it must know the states to which its actions will lead.
- A Q-learning agent, on the other hand, can compare the expected utilities for its available choices without needing to know their outcomes, so it does not need a model of the environment.
- On the other hand, because they do not know where their actions lead, Q-learning agents cannot look ahead; this can seriously restrict their ability to learn

(i) Passive learning, where the agent's policy is fixed and the task is to learn the utilities of states (or state–action pairs); this could also involve learning a model of the environment.

(ii) Active learning, where the agent must also learn what to do.

→ The principal issue is **exploration**: an agent must experience as much as possible of its environment in order to learn how to behave in it.

3.1.2. PASSIVE REINFORCEMENT LEARNING :

- To keep things simple, we start with the case of a passive learning agent using a state-based representation in a fully observable environment.
- In passive learning, the agent's policy π is fixed: in state s, it always executes the action π(s).

Its goal is simply to learn how good the policy is—that is, to learn the utility function U π(s).

- We will use as our example the 4 × 3 world shows a policy for that world and the corresponding utilities.
- Clearly, the passive learning task is similar to the **policy evaluation** task, part of the **policy iteration** algorithm.

The main difference is that the passive learning agent does not know the **transition model** P(s | s, a), which specifies the probability of reaching state s from state s after doing action a; nor does it now the **reward function** R(s), which specifies the reward for each state.

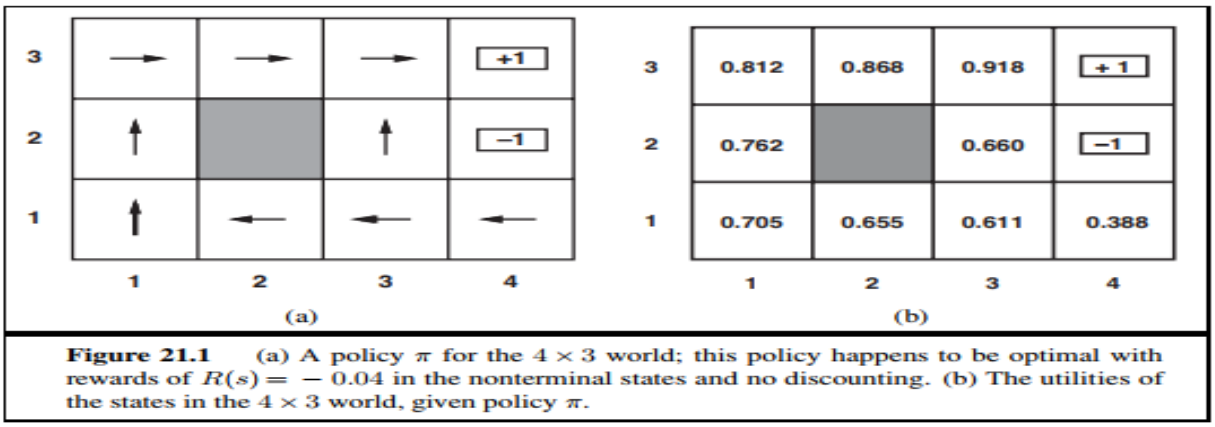


Figure 21.1 (a) A policy π for the 4×3 world; this policy happens to be optimal with rewards of $R(s) = -0.04$ in the nonterminal states and no discounting. (b) The utilities of the states in the 4×3 world, given policy π.

- The agent executes a set of trials in the environment using its policy π. In each trial, the agent starts in state (1,1) and experiences a sequence of state transitions until it reaches one of the terminal states, (4,2) or (4,3). Its percepts supply both the current state and the reward received in that state.

- The utility is defined to be the expected sum of (discounted) rewards obtained if policy π is followed. As in Equation we write :

where R(s) is the reward for a state, St (a random variable) is the state reached at time t when executing policy π, and S0 = s. We will include a discount factor γ in all of our equations, but for the 4 × 3 world we will set γ = 1.

(i) Direct utility estimation :

- A simple method for **direct utility estimation** was invented in the late 1950s in the area of **adaptive control theory** by Widrow and Hoff (1960).

- The idea is that the utility of a state is the expected total reward from that state onward (called the expected **reward-to-go**), and each trial provides a *sample* of this quantity for each state visited.

- For example, the first trial in the set of three given earlier provides a sample total reward of 0.72 for state (1,1) and so on.

Thus, at the end of each sequence, the algorithm calculates the observed reward-to-go for each state and updates the estimated utility for that state accordingly, just by keeping a running average for each state in a table.

- It is clear that direct utility estimation is just an instance of supervised learning where each example has the state as input and the observed reward-to-go as output.

- Direct utility estimation succeeds in reducing the reinforcement learning problem to an inductive learning problem, about which much is known.

- Unfortunately, it misses a very important source of information, namely, the fact that the utilities of states are not independent! *The utility of each state equals its own reward plus the expected utility of its successor states.*

That is, the utility values obey the Bellman equations for a fixed policy :

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) U^{\pi}(s')$$

(ii) Adaptive dynamic programming :

- An **adaptive dynamic programming** (or ADP) agent takes advantage of the constraints among the utilities of states by learning the transition model that connects them and solving the corresponding Markov decision process using a dynamic programming method.

For a passive learning agent, this means plugging the learned transition model P(s |s, π(s)) and the observed rewards R(s) into the Bellman equations to calculate the utilities of the states.

- Alternatively, we can adopt the approach of **modified policy iteration** , using a simplified value iteration process to update the utility estimates after each change to the learned model.

- Because the model usually changes only slightly with each observation, the value iteration process can use the previous utility estimates as initial values and should converge quite quickly.
- The process of learning the model itself is easy, because the environment is fully observable. This means that we have a supervised learning task where the input is a state–action pair and the output is the resulting state.

In the simplest case, we can represent the transition model as a table of probabilities.

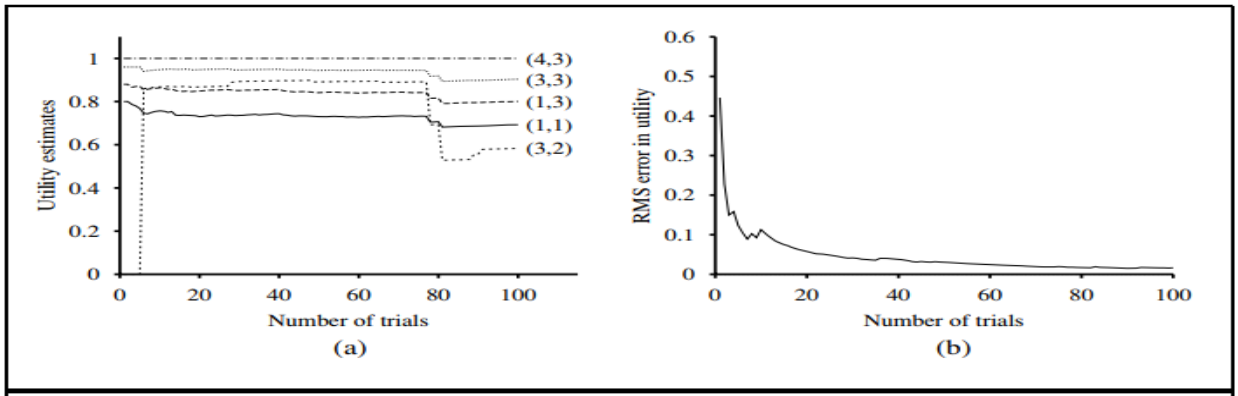


Figure 21.3 The passive ADP learning curves for the 4×3 world, given the optimal policy

- There are two mathematical approaches that have this flavor.
- The first approach, **Bayesian reinforcement learning**, assumes a prior probability P (h) for each hypothesis h about what the true model is; the posterior probability P (h | **e**) is obtained in the usual way by Bayes' rule given the observations to date.

The second approach, derived from **robust control theory**, allows for a *set* of possible models H and defines an optimal robust policy as one that gives the best outcome in the *worst case* over H:

(iii) Temporal-difference learning :

- Solving the underlying MDP as in the preceding section is not the only way to bring the Bellman equations to bear on the learning problem.
- Another way is to use the observed transitions to adjust the utilities of the observed states so that they agree with the constraint equations.
- Consider, for example, the transition from (1,3) to (2,3) Suppose that, as a result of the first trial, the utility estimates are U π(1, 3) = 0.84 and U π(2, 3) = 0.92.

Now, if this transition occurred all the time, we would expect the utilities to obey the equation

$$U^{\pi}(1,3) = -0.04 + U^{\pi}(2,3)$$

- So U π(1, 3) would be 0.88. Thus, its current estimate of 0.84 might be a little low and should be increased. More generally, when a transition occurs from state s to state s, we apply the following update to U π(s):
- Here, α is the **learning rate** parameter. Because this update rule uses the difference in utilities between successive states, it is often called the **temporal-difference**, or TD, equation.

3.1.3. ACTIVE REINFORCEMENT LEARNING:

- A passive learning agent has a fixed policy that determines its behavior. An active agent must decide what actions to take.
- Let us begin with the adaptive dynamic programming agent and consider how it must be modified to handle this new freedom.
- The utilities it needs to learn are those defined by the *optimal* policy; they obey the Bellman equations , which we repeat here for convenience:

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

(i) Exploration :

- In Figure it shows the results of one sequence of trials for an ADP agent that follows the recommendation of the optimal policy for the learned model at each step.
- The agent *does not* learn the true utilities or the true optimal policy! What happens instead is that, in the 39th trial, it finds a policy that reaches the +1 reward along the lower route via (2,1), (3,1),(3,2), and (3,3).
- After experimenting with minor variations, from the 276th trial onward it sticks to that policy, never learning the utilities of the other states and never finding the optimal route via (1,2), (1,3), and (2,3).
- We call this agent the **greedy agent**. Repeated experiments show that the greedy agent *very seldom* converges to the optimal policy for this environment and sometimes converges to really horrendous policies.

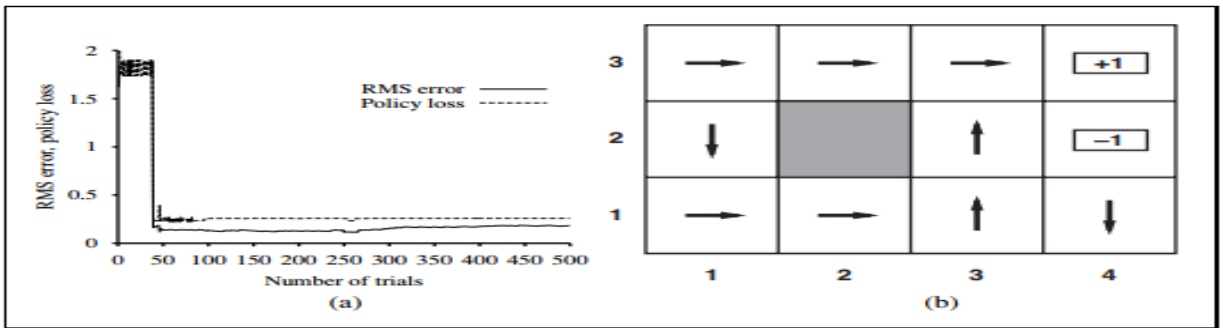


Figure 21.6 Performance of a greedy ADP agent that executes the action recommended by the optimal policy for the learned model. (a) RMS error in the utility estimates averaged over the nine nonterminal squares. (b) The suboptimal policy to which the greedy agent converges in this particular sequence of trials.

- By improving the model, the agent will receive greater rewards in the future. An agent therefore must make a tradeoff between **exploitation** to maximize its reward.
- Pure exploitation risks getting stuck in a rut. Pure exploration to improve one's knowledge is of no use if one never puts that nowledge into practice. With greater understanding, less exploration is necessary.
- A GLIE scheme must try each action in each state an unbounded number of times to avoid having a finite probability that an optimal action is missed because of an unusually bad series of outcomes.

  An ADP agent using such a scheme will eventually learn the true environment model.

- In Las Vegas, a *one-armed bandit* is a slot machine. A gambler can insert a coin, pull the lever, and collect the winnings (if any).
- An n-**armed bandit** has n levers. The gambler must choose which lever to play on each successive coin—the one that has paid off best, or maybe one that has not been tried?
- This is because the agent stops exploring the unrewarding parts of the state space fairly soon, visiting them only "by accident" thereafter.

  However, it makes perfect sense for the agent not to care about the exact utilities of states that it knows are undesirable and can be avoided.

(ii) Learning an action-utility function :

- Now that we have an active ADP agent, let us consider how to construct an active temporal difference learning agent.
- The most obvious change from the passive case is that the agent is no longer equipped with a fixed policy, so, if it learns a utility function U, it will need to learn a model in order to be able to choose an action based on U via one-step look-ahead.

  The model acquisition problem for the TD agent is identical to that for the ADP agent.
- Suppose the agent takes a step that normally leads to a good destination, but because of non-determinism in the environment the agent ends up in a catastrophic state. The TD update rule will take this as seriously.
- There is an alternative TD method, called **Q-learning**, which learns an action-utility representation instead of learning utilities. We will use the notation Q(s, a) to denote the value of doing action a in state s. Q-values are directly related to utility values as follows:

$$U(s) = \max_a Q(s, a) \ .$$

- Q-functions may seem like just another way of storing utility information, but they have avery important property:

  "a TD agent that learns a Q-function does not need a model of the form P(s | s, a)*, either for learning or for action selection."*
- For this reason, Q-learning is called a **model-free** method.
- As with utilities, we can write a constraint equation :

$$Q(s, a) = R(s) + \gamma \sum_{s'} P(s' \mid s, a) \max_{a'} Q(s', a') \ .$$

```
function Q-LEARNING-AGENT(percept) returns an action
  inputs: percept, a percept indicating the current state s' and reward signal r'
  persistent: Q, a table of action values indexed by state and action, initially zero
              N_sa, a table of frequencies for state–action pairs, initially zero
              s, a, r, the previous state, action, and reward, initially null

  if TERMINAL?(s) then Q[s, None] ← r'
  if s is not null then
     increment N_sa[s, a]
     Q[s, a] ← Q[s, a] + α(N_sa[s, a])(r + γ max_a' Q[s', a'] − Q[s, a])
  s, a, r ← s', argmax_a' f(Q[s', a'], N_sa[s', a']), r'
  return a
```

Figure 21.8 An exploratory Q-learning agent. It is an active learner that learns the value $Q(s, a)$ of each action in each situation. It uses the same exploration function f as the exploratory ADP agent, but avoids having to learn the transition model because the Q-value of a state can be related directly to those of its neighbors.

- Q-learning has a close relative called SARSA (for State-Action-Reward-State-Action).
- The update rule for SARSA is

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma\, Q(s', a') - Q(s, a)) \ ,$$

3.1.4. GENERALIZATION IN REINFORCEMENT LEARNING :

- So far, we have assumed that the utility functions and Q-functions learned by the agents are represented in tabular form with one output value for each input tuple.
- Such an approach works reasonably well for small state spaces, but the time to convergence and (for ADP) the time per iteration increase rapidly as the space gets larger.
- With carefully controlled, approximate ADP methods, it might be possible to handle 10,000 states or more. This suffices for two-dimensional maze-like environments, but more realistic worlds are out of the question.
- Backgammon and chess are tiny subsets of the real world, yet their state spaces contain on the order of 1020 and 1040 states, respectively.
- It would be absurd to suppose that one must visit all these states many times in order to learn how to play the game.
- One way to handle such problems is to use **function approximation**, which simplymeans using any sort of representation for the Q-function other than a lookup table.
- The representation is viewed as approximate because it might not be the case that the *true* utility function or Q-function can be represented in the chosen form.
- For example, we described an **evaluation function** for chess that is represented as a weighted linear

function of a set of **features** (or **basis functions**) f1, . . . , fn:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \cdots + \theta_n f_n(s)$$

- A reinforcement learning algorithm can learn values for the parameters θ = θ1, . . . , θn such that the evaluation function Uˆθ approximates the true utility function.
- Instead of, say, values in a table, this function approximator is characterized by, say, n = 20 parameters an *enormous* compression.
- Although no one knows the true utility function for chess, no one believes that it can be represented exactly in 20 numbers.
- If the approximation is good enough, however, the agent might still play excellent chess.
- Function approximation makes it practical to represent utility functions for very large state spaces, but that is not its principal benefit.
- *The compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited.*
- That is, the most important aspect of function approximation is not that it requires less space, but that it allows for inductive generalization over input states.
- For reinforcement learning, it makes more sense to use an *online* learning algorithm that updates the parameters after each trial.
- Suppose we run a trial and the total reward obtained starting at (1,1) is 0.4. This suggests that Uˆθ(1, 1), currently 0.8, is too large and must be reduced.
- There are more sophisticated algorithms that can avoid these problems, but at present reinforcement learning with general function approximators remains a delicate art.
- Function approximation can also be very helpful for learning a model of the environment. Remember that learning a model for an *observable* environment is a *supervised* learning problem, because the next percept gives the outcome state.
- For a *partially observable* environment, the learning problem is much more difficult.
- If we know what the hidden variables are and how they are causally related to each other and to the observable variables, then we can fix the structure of a dynamic Bayesian network and use the EM algorithm to learn the parameters .
- Inventing the hidden variables and learning the model structure are still open problems.

3.1.5. POLICY SEARCH :

- The final approach we will consider for reinforcement learning problems is called policy search.
- In some ways, policy search is the simplest of all the methods, the idea is to keep twiddling the policy as long as its performance improves, then stop.

- Let us begin with the policies themselves. Remember that a policy π is a function that maps states to actions.
- We are interested primarily in *parameterized* representations of π that have far fewer parameters than there are states in the state space .
- For example, we could represent π by a collection of parameterized Q-functions, one for each action, and take the action with the highest predicted value:

$$\pi(s) = \max_a \hat{Q}_\theta(s, a)$$

- Each Q-function could be a linear function of the parameters θ, or it could be a nonlinear function such as a neural network.
- Policy search will then adjust the parameters θ to improve the policy. Notice that if the policy is represented by Q functions, then policy search results in a process that learns Q-functions.
- *This process is not the same as Q-learning!* In Q-learning with function approximation, the algorithm finds a value of θ such that Qˆθ is "close" to Q∗, the optimal Q-function.
- Policy search, on the other hand, finds a value of θ that results in good performance; the values found by the two methods may differ very substantially.

  One problem with policy representations of the kind is that the policy is a *discontinuous* function of the parameters when the actions are discrete.
- That is, there will be values of θ such that an infinitesimal change in θ causes the policy to switch from one action to another.
- This means that the value of the policy may also change discontinuously, which makes gradient-based search difficult.

  For this reason, policy search methods often use a **stochastic policy** representation πθ(s, a), which specifies the *probability* of selecting action a in state s. One popular representation is the **softmax function**:

$$\pi_\theta(s, a) = e^{\hat{Q}_\theta(s,a)} / \sum_{a'} e^{\hat{Q}_\theta(s,a')}$$

- For the case of a stochastic policy πθ(s, a), it is possible to obtain an unbiased estimate of the gradient at θ, ∇θρ(θ), directly from the results of trials executed at θ.
- In this case, the policy value is just the expected value of the reward, and we have

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a \pi_\theta(s_0, a) R(a) = \sum_a (\nabla_\theta \pi_\theta(s_0, a)) R(a)$$

Suppose that we have N trials in all and the action taken on the jth trial is aj. Then

$$\nabla_\theta \rho(\theta) = \sum_a \pi_\theta(s_0, a) \cdot \frac{(\nabla_\theta \pi_\theta(s_0, a)) R(a)}{\pi_\theta(s_0, a)} \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s_0, a_j)) R(a_j)}{\pi_\theta(s_0, a_j)}$$

- For the sequential case, this generalizes to :

$$\nabla_\theta \rho(\theta) \approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s, a_j)) R_j(s)}{\pi_\theta(s, a_j)}$$

- Policy search is carried out by evaluating each candidate policy using the *same* set of random sequences to determine the action outcomes.

It can be shown that the number of random sequences required to ensure that the value of *every* policy is well estimated depends only on the complexity of the policy space, and not at all on the complexity of the underlying domain.

3.1.6. APPLICATIONS OF REINFORCEMENT LEARNING:

(i) Applications to game playing :

→ The first significant application of reinforcement learning was also the first significant learning program of any kind—the checkers program written by Arthur Samuel (1959, 1967).

- Samuel first used a weighted linear function for the evaluation of positions, using up to 16 terms at any one time. He applied a version of Equation to update the weights.
- There were some significant differences, however, between his program and current methods.
- First, he updated the weights using the difference between the current state and the backed-up value generated by full look-ahead in the search tree.
- Second difference was that the program did *not* use any observed rewards .

→ **Gerry Tesauro's backgammon program TD-GAMMON** (1992) forcefully illustrates the potential of reinforcement learning techniques.

- In earlier work Tesauro tried learning a neural network representation of Q(s, a) directly from examples of moves labeled with relative values by a human expert.
- This approach proved extremely tedious for the expert. It resulted in a program, called NEUROGAMMON, that was strong by computer standards, but not competitive with human experts.
- The TD-GAMMON project was an attempt to learn from self-play alone. The only reward signal was given at the end of each game.

(ii) Application to robot control :

- The setup for the famous cart–pole balancing problem, also known as the **inverted pendulum.**

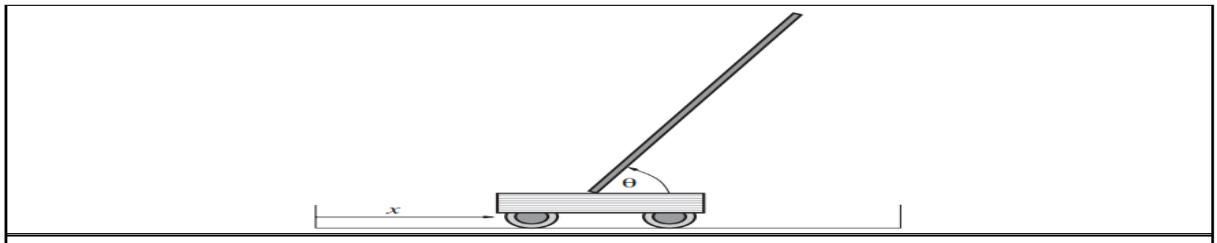


Figure 21.9 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x, θ, $\dot{x}$, and $\dot{\theta}$.

- The problem is to control the position x of the cart so that the pole stays roughly upright ($\theta \approx \pi/2$), while staying within the limits of the cart track as shown.
- The cart–pole problem differs from the problems described earlier in that the state variables x, θ, $\dot{x}$, and $\dot{\theta}$ are continuous.
- The actions are usually discrete: jerk left or jerk right, the so-called **bang-bang control** regime.

The earliest work on learning for this problem was carried out by Michie and Chambers (1968). Their BOXES algorithm was able to balance the pole for over an hour after only about 30 trials.

- Moreover, unlike many subsequent systems, BOXES was implemented with a real cart and pole, not a simulation.
- The algorithm first discretized the four-dimensional state space into boxes—hence the name. It then ran trials until the pole fell over or the cart hit the end of the track.
- Negative reinforcement was associated with the final action in the final box and then propagated back through the sequence.
- It was found that the discretization caused some problems when the apparatus was initialized in a position different from those used in training, suggesting that generalization was not perfect.
- Improved generalization and faster learning can be obtained using an algorithm that *adaptively* partitions the state space according to the observed variation in the reward, or by using a continuous-state, nonlinear function approximator such as a neural network.
- Nowadays, balancing a *triple* inverted pendulum is a common exercise—a feat far beyond the capabilities of most humans.

➔ Still more impressive is the application of reinforcement learning to **helicopter flight.**

Figure 21.10 Superimposed time-lapse images of an autonomous helicopter performing a very difficult "nose-in circle" maneuver. The helicopter is under the control of a policy developed by the PEGASUS policy-search algorithm. A simulator model was developed by observing the effects of various control manipulations on the real helicopter; then the algorithm was run on the simulator model overnight. A variety of controllers were developed for different maneuvers. In all cases, performance far exceeded that of an expert human pilot using remote control. (Image courtesy of Andrew Ng.)

- This work has generally used policy search as well as the PEGASUS algorithm with simulation based on a learned transition model.

ARTIFICIAL INTELLIGENCE

UNIT – III

Chapter – 2

Natural Language Processing

3.2.1. LANGUAGE MODELS :

- Formal languages, such as the programming languages Java or Python, have precisely defined language models.
- A **language** can be defined as a set of strings; "print(2 + 2)" is a legal program in the language Python, whereas "2)+(2 print" is not.
- Since there are an infinite number of legal programs, they cannot be enumerated; instead they are specified by set of rules called a **grammar**.
- Formal languages also have rules that define the meaning or **semantics** of a program; for example, the rules say that the "meaning" of "2 + 2" is 4, and the meaning of "1/0" is that an error is signaled.
- Natural languages, such as English or Spanish, cannot be characterized as a definitive set of sentences.
- Everyone agrees that "Not to be invited is sad" is a sentence of English, but people disagree on the grammaticality of "To be not invited is sad."
- Therefore, it is more fruitful to define a natural language model as a probability distribution over sentences rather than a definitive set.

  P(S = words)
- Natural languages are also **ambiguous**. Because we cannot speak of a single meaning for a sentence, but rather of a probability distribution over possible meanings.
- Finally, natural languages are difficult to deal with because they are very large, and constantly changing.
- Thus, our language models are, at best, an approximation. We start with the simplest possible approximations and move up from there.

(i) *N*-gram character models :

- Ultimately, a written text is composed of **characters**—letters, digits, punctuation, and spaces in English (and more exotic characters in some other languages).
- Thus, one of the simplest language models is a probability distribution over sequences of characters.
- We write P(C1:N) for the probability of a sequence of N characters, C1 through CN.
- In one Web collection, P("the") = 0.027 and P("zgq") = 0.000000002.
- A sequence of written symbols of length n is called an n-gram (from the Greek root for writing or letters), with special case "unigram" for 1-gram, "bigram" for 2-gram, and "trigram" for 3-gram.
- A model of the probability distribution of n-letter sequences is thus called an n-**gram model**. (But be careful: we can have n-gram models over sequences of words, syllables, or other units; not just over characters.)
- An n-gram model is defined as a **Markov chain** of order n - 1.
- In a Markov chain the probability of character ci depends only on the immediately preceding characters, not on any other characters.
- So in a trigram model (Markov chain of order 2) we have :

$$P(c_i \mid c_{1:i-1}) = P(c_i \mid c_{i-2:i-1})$$

- We can define the probability of a sequence of characters P(c1:N) under the trigram model by first

factoring with the chain rule and then using the Markov assumption:

$$P(c_{1:N}) = \prod_{i=1}^{N} P(c_i \mid c_{1:i-1}) = \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1})$$

- We call a body of text a **corpus** (plural *corpora*), from the Latin word for *body*.
- What can we do with n-gram character models? One task for which they are well suited is **language identification** .
- For Example, given a text, determine what natural language it is written in.
- This is a relatively easy task; even with short texts such as “Hello, world” or “Wie geht es dir,” it is easy to identify the first as English and the second as German.
- Computer systems identify languages with greater than 99% accuracy; occasionally, closely related languages, such as Swedish and Norwegian, are confused.
- One approach to language identification is to first build a trigram character model of each candidate language, where the variable L ranges over languages.
- That gives us a model of **P**(Text | Language), but we want to select the most probable language given the text, so we apply Bayes’ rule followed by the Markov assumption to get the most probable language:

$$\begin{aligned} \ell^* &= \operatorname*{argmax}_{\ell} P(\ell \mid c_{1:N}) \\ &= \operatorname*{argmax}_{\ell} P(\ell) P(c_{1:N} \mid \ell) \\ &= \operatorname*{argmax}_{\ell} P(\ell) \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1}, \ell) \end{aligned}$$

(ii) Smoothing *n*-gram models :

- The major complication of n-gram models is that the training corpus provides only an estimate of the true probability distribution.
- For common character sequences such as “ _th” any English corpus will give a good estimate: about 1.5% of all trigrams. On the other hand, “ _ht” is very uncommon—no dictionary words start with ht.

The process of adjusting the probability of low-frequency counts is called **smoothing**.

- The simplest type of smoothing was suggested by Pierre-Simon Laplace in the 18th century: he said that, in the lack of further information, if a random Boolean variable X has been false in all n observations so far then the estimate for P (X = true) should be 1/(n+2).
- That is, he assumes that with two more trials, one might be true and one false. Laplace smoothing (also called add-one smoothing) is a step in the right direction, but performs relatively poorly.
- A better approach is a **backoff model**, in which we start by estimating n-gram counts, but for any particular sequence that has a low (or zero) count, we back off to (n - 1)-grams.
- **Linear interpolation smoothing** is a backoff model that combines trigram, bigram, and unigram models by linear interpolation. It defines the probability estimate as :

$$\widehat{P}(c_i|c_{i-2:i-1}) = \lambda_3 P(c_i|c_{i-2:i-1}) + \lambda_2 P(c_i|c_{i-1}) + \lambda_1 P(c_i)$$

- where λ3 + λ2 + λ1 = 1. The parameter values λi can be fixed, or they can be trained with an expectation–maximization algorithm.
- It is also possible to have the values of λi depend on the counts: if we have a high count of trigrams, then we weigh them relatively more; if only a low count, then we put more weight on the bigram and unigram models.

(iii) Model evaluation :

- With so many possible n-gram models—unigram, bigram, trigram, interpolated smoothing with different values of λ, etc.—how do we know what model to choose? We can evaluate a model with cross-validation.
- Split the corpus into a training corpus and a validation corpus.Determine the parameters of the model from the training data. Then evaluate the model on the validation corpus.
- The evaluation can be a task-specific metric, such as measuring accuracy on language identification.
- Alternatively we can have a task-independent model of language quality: calculate the probability assigned to the validation corpus by the model; the higher the probability the better.
- This metric is inconvenient because the probability of a large corpus will be a very small number, and floating-point underflow becomes an issue.
- A different way of describing the probability of a sequence is with a measure called **perplexity**, defined as :

$$Perplexity(c_{1:N}) = P(c_{1:N})^{-\frac{1}{N}}$$

- Perplexity can be thought of as the reciprocal of probability, normalized by sequence length.
- It can also be thought of as the weighted average branching factor of a model. Suppose there are 100 characters in our language, and our model says they are all equally likely. Then for a sequence of any length, the perplexity will be 100.
- If some characters are more likely than others, and the model reflects that, then the model will have a perplexity less than 100.

(iv) N-gram word models :

- Now we turn to n-gram models over words rather than characters.
- All the same mechanism applies equally to word and character models. The main difference is that the **vocabulary**—the set of symbols that make up the corpus and the model—is larger.
- There are only about 100 characters in most languages, and sometimes we build character models that are even more restrictive, for example by treating “A” and “a” as the same symbol or by treating all punctuation as the same symbol.
- But with word models we have at least tens of thousands of symbols, and sometimes millions.
- The wide range is because it is not clear what constitutes a word.
- In English a sequence of letters surrounded by spaces is a word, but in some languages, like Chinese, words are not separated by spaces, and even in English many decisions must be made to have a clear policy on word boundaries: how many words are in.

- Word n-gram models need to deal with **out of vocabulary** words.
- With character models, we didn't have to worry about someone inventing a new letter of the alphabet.
- But with word models there is always the chance of a new word that was not seen in the training corpus, so we need to model that explicitly in our language model.

3.2.2. TEXT CLASSIFICATION :

- We now consider in depth the task of **text classification**, also known as **categorization**: givena text of some kind, decide which of a predefined set of classes it belongs to.
- Language identification and genre classification are examples of text classification
- **spam detection** classifying an email message as spam or not-spam(ham).
- A training set is readily available: the positive (spam) examples are in my spam folder, the negative (ham) examples are in my inbox.
- Note that we have two complementary ways of talking about classification.
- In the language-modeling approach, we define one n-gram language model for **P**(Message | spam) by training on the spam folder, and one model for **P**(Message | ham) by training on the inbox.
- Then we can classify a new message with an application of Bayes' rule:

$$\underset{c \in \{spam, ham\}}{\operatorname{argmax}} P(c \mid message) = \underset{c \in \{spam, ham\}}{\operatorname{argmax}} P(message \mid c)\, P(c)$$

- where P (c) is estimated just by counting the total number of spam and ham messages. This approach works well for spam detection, just as it did for language identification.
- If there are 100,000 words in the language model, then the feature vector has length 100,000, but for a short email message almost all the features will have count zero.
- This unigram representation has been called the **bag of words** model.
- You can think of the model as putting the words of the training corpus in a bag and then selecting words one at a time.
- The notion of order of the words is lost; a unigram model gives the same probability to any permutation of a text.
- Higher-order n-gram models maintain some local notion of word order.
- It can be expensive to run algorithms on a very large feature vector, so often a process of **feature selection** is used to keep only the features that best discriminate between spam and ham.
- Once we have chosen a set of features, we can apply any of the supervised learning techniques we have seen; popular ones for text categorization include k-nearest-neighbors, support vector machines, decision trees, naive Bayes, and logistic regression.
- All of these have been applied to spam detection, usually with accuracy in the 98%–99% range. With a carefully designed feature set, accuracy can exceed 99.9%.

(i) Classification by data compression :

- Another way to think about classification is as a problem in **data compression**.
- A lossless compression algorithm takes a sequence of symbols, detects repeated patterns in it, and writes a description of the sequence that is more compact than the original.
- For example, the text "0.142857142857142857" might be compressed to "0.[142857]*3."
- To do classification by compression, we first lump together all the spam training messages and compress them as a unit.
- We do the same for the ham. Then when given a new message to classify, we append it to the spam messages and compress the result.
- We also append it to the ham and compress that. Whichever class compresses better—adds the fewer number of additional bytes for the new message—is the predicted class.

3.2.3. INFORMATION RETRIEVAL :

- Information retrieval is the task of finding documents that are relevant to a user's need for information.
- The best-known examples of information retrieval systems are search engines on the World Wide Web.
- A Web user can type a query such as "AI book" into a search engine and see a list of relevant pages.

- An **information retrieval** (henceforth **IR**) system can be **characterized** by :

→**A corpus of documents.** Each system must decide what it wants to treat as a document: a paragraph, a page, or a multipage text.

➔ **Queries posed in a query language**. A query specifies what the user wants to know.The query language can be just a list of words, such as [AI book]; or it can specify a phrase of words that must be adjacent, as in ["AI book"]; it can contain Boolean operators as in [AI AND book]; it can include non-Boolean operators such as [AI NEAR book].

➔ →**A result set.** This is the subset of documents that the IR system judges to be **relevant** to the query.

→ **A presentation of the result set.** This can be as simple as a ranked list of document titles or as complex as a rotating color map of the result set projected onto a three dimensional space, rendered as a two-dimensional display.

- The earliest IR systems worked on a **Boolean keyword model**. Each word in the documentcollection is treated as a Boolean feature that is true of a document if the word occurs in the document and false if it does not.

  The query language is the language of Boolean expressions over features. A document is relevant only if the expression evaluates to true.

- This model has the advantage of being simple to explain and implement.
- However, it has some **disadvantages:**

➔ First, the degree of relevance of a document is a single bit, so there is no guidance as to how to order the relevant documents for presentation.

➔ Second, Boolean expressions are unfamiliar to users who are not programmers or logicians.

➔ Third, it can be hard to formulate an appropriate query, even for a skilled user.

(i) IR scoring functions :

- Most IR systems have abandoned the Boolean model and use models based on the statistics of word counts. We describe the **BM25 scoring function.**
- A scoring function takes a document and a query and returns a numeric score; the most relevant documents have the highest scores.

  In the BM25 function, the score is a linear weighted combination of scores for each of the words that make up the query.
- **Three factors** affect the weight of a query term:

→ First, the frequency with which a query term appears in a document (also known as TF for term frequency). For the query documents that mention "farming" frequently will have higher scores.

→ Second, the inverse document frequency of the term, or IDF. The word "in" appears in almost every document, so it has a high document frequency, and thus a low inverse document frequency, and thus it is not as important to the query.

➔ Third, the length of the document. A million-word document will probably mention all the query words, but may not actually be about the query. A short document that mentions all the words is a much better candidate.

- The BM25 function takes all three of these into account.
- Then, given a document dj and a query consisting of the words q1:N, we have :

$$BM25(d_j, q_{1:N}) = \sum_{i=1}^{N} IDF(q_i) \cdot \frac{TF(q_i, d_j) \cdot (k+1)}{TF(q_i, d_j) + k \cdot (1 - b + b \cdot \frac{|d_j|}{L})}$$

- IDF(qi) is the inverse document frequency of word qi, given by :

$$IDF(q_i) = \log \frac{N - DF(q_i) + 0.5}{DF(q_i) + 0.5} .$$

(ii) IR system evaluation :

- How do we know whether an IR system is performing well? We undertake an experiment in which the system is given a set of queries and the result sets are scored with respect to human relevance judgments.
- Traditionally, there have been two measures used in the scoring:

  → recall

  → precision.
- **Precision** measures the proportion of documents in the result set that are actually relevant.

- In our example, the precision is 30/(30 + 10) = .75. The false positive rate is 1 - .75 = .25.
- **Recall** measures the proportion of all the relevant documents in the collection that are in the result set.
- In our example, recall is 30/(30 + 20) = .60. The false negative rate is 1 - .60 = .40.
- In a very large document collection, such as the World Wide Web, recall is difficult to compute, because there is no easy way to examine every page on the Web for relevance.
- All we can do is either estimate recall by sampling or ignore recall completely and just judge precision.

(iii) IR refinements :

- There are many possible refinements to the system described here, and indeed Web search engines are continually updating their algorithms as they discover new approaches and as the Web grows and changes.
- One common refinement is a better model of the effect of document length on relevance.
- Singhal *et al.* (1996) observed that simple document length normalization schemes tend to favor short documents too much and long documents not enough.
- They propose a *pivoted* document length normalization scheme; the idea is that the pivot is the document length at which the old-style normalization is correct; documents shorter than that get a boost and longer ones get a penalty.
- The BM25 scoring function uses a word model that treats all words as completely independent, but we know that some words are correlated.
- Many IR systems attempt to account for these correlations.
- The next step is to recognize **synonyms**, such as "sofa" for "couch." As with stemming, this has the potential for small gains in recall, but can hurt precision.
- As a final refinement, IR can be improved by considering m**etadata**—data outside of the text of the document. Examples include human-supplied keywords and publication data.
- On the Web, hypertext **links** between documents are a crucial source of information.

(iv) The PageRank algorithm :

- **PageRank** was one of the two original ideas that set Google's search apart from other Web search engines when it was introduced in 1997. (The other innovation was the use of anchor text—the underlined text in a hyperlink).
- PageRank was invented to solve the problem of the tyranny of TF scores: if the query is [IBM], how do we make sure that IBM's home page, ibm.com, is the first result, even if another page mentions the term "IBM" more frequently?

  The idea is that ibm.com has many in-links (links to the page), so it should be ranked higher: each in-link is a vote for the quality of the linked-to page.
- But if we only counted in-links, then it would be possible for a Web spammer to create a network of pages and have them all point to a page of his choosing, increasing the score of that page.

- Therefore, the PageRank algorithm is designed to weight links from high-quality sites more heavily.
- What is a highquality site? One that is linked to by other high-quality sites.
- The definition is recursive, but we will see that the recursion bottoms out properly. The PageRank for a page p is defined as:

$$PR(p) = \frac{1-d}{N} + d\sum_{i} \frac{PR(in_i)}{C(in_i)}$$

- where P R(p) is the PageRank of page p, N is the total number of pages in the corpus, ini are the pages that link in to p, and C(ini) is the count of the total number of out-links on page ini.
- The constant d is a damping factor. It can be understood through the **random surfer model** : imagine a Web surfer who starts at some random page and begins exploring.

(v) The HITS algorithm :

- The Hyperlink-Induced Topic Search algorithm, also known as “Hubs and Authorities” or HITS, is another influential link-analysis algorithm .
- HITS differs from PageRank in several ways.
- First, it is a query-dependent measure: it rates pages with respect to a query.
- Given a query, HITS first finds a set of pages that are relevant to the query. It does that by intersecting hit lists of query words, and then adding pages in the link neighborhood of these pages
- Both PageRank and HITS played important roles in developing our understanding of Web information retrieval.
- These algorithms and their extensions are used in ranking billions of queries daily as search engines steadily develop better ways of extracting yet finer signals of search relevance.

(vi) Question answering :

- Information retrieval is the task of finding documents that are relevant to a query, where the query may be a question, or just a topic area or concept.
- **Question answering** is a somewhat different task, in which the query really is a question, and the answer is not a ranked list of documents but rather a short response—a sentence, or even just a phrase.
- There have been question-answering NLP (natural language processing) systems since the 1960s, but only since 2001 have such systems used Web information retrieval to radically increase their breadth of coverage.

3.2.4. INFORMATION EXTRACTION :

- Information extraction is the process of acquiring knowledge by skimming a text and looking for occurrences of a particular class of object and for relationships among objects.
- A typical task is to extract instances of addresses from Web pages, with database fields for street, city, state, and zip code; or instances of storms from weather reports, with fields for temperature, wind speed, and precipitation.
- In a limited domain, this can be done with high accuracy. As the domain gets more general, more complex linguistic models and more complex learning techniques are necessary.

(i) Finite-state automata for information extraction:

- The simplest type of information extraction system is an **attribute-based extraction** system that assumes that the entire text refers to a single object and the task is to extract attributes of that object.
- For example, the problem of extracting from the text "IBM ThinkBook 970. Our price: $399.00" the set of attributes {Manufacturer=IBM, Model=ThinkBook970, Price=$399.00}.
- We can address this problem by defining a **template** (also known as a pattern) for each attribute we would like to extract. The template is defined by a finite state automaton, the simplest example of which is the **regular expression**, or regex.
- Here we show how to build up a regular expression template for prices in dollars:

`[0-9]`	matches any digit from 0 to 9
`[0-9]+`	matches one or more digits
`[.][0-9][0-9]`	matches a period followed by two digits
`([.][0-9][0-9])?`	matches a period followed by two digits, or nothing
`[$][0-9]+([.][0-9][0-9])?`	matches $249.99 or $1.23 or $1000000 or . . .

- Templates are often defined with three parts: a prefix regex, a target regex, and a postfix regex.
- For prices, the target regex is as above, the prefix would look for strings such as "price:" and the postfix could be empty.
- The idea is that some clues about an attribute come from the attribute value itself and some come from the surrounding text.
- One step up from attribute-based extraction systems are **relational extraction** systems, which deal with multiple objects and the relations among them.
- Thus, when these systems see the text "$249.99," they need to determine not just that it is a price, but also which object has that price.
- A typical relational-based extraction system is FASTUS, which handles news stories about corporate mergers and acquisitions.
- A relational extraction system can be built as a series of **cascaded finite-state transducers**.
- That is, the system consists of a series of small, efficient finite-state automata (FSAs), where each automaton receives text as input, transduces the text into a different format, and passes it along to the next automaton.

- **FASTUS consists of five stages:**

1. Tokenization
2. Complex-word handling
3. Basic-group handling
4. Complex-phrase handling
5. Structure merging

1. FASTUS's first stage is **tokenization**, which segments the stream of characters into tokens (words, numbers, and punctuation). Some tokenizers also deal with markup languages such as HTML, SGML, and XML.
2. The second stage handles **complex words**, including collocations such as "set up" and "joint venture," as well as proper names such as "Bridgestone Sports Co."
3. The third stage handles **basic groups**, meaning noun groups and verb groups. The idea is to chunk these into units that will be managed by the later stages.
4. The fourth stage combines the basic groups into **complex phrases**.
5. The final stage **merges structures** that were built up in the previous step.

(ii) Probabilistic models for information extraction:

- When information extraction must be attempted from noisy or varied input, simple finite-state approaches fare poorly.
- It is too hard to get all the rules and their priorities right; it is better to use a probabilistic model rather than a rule-based model.
- The simplest probabilistic model for sequences with hidden state is the hidden Markov model, or HMM.
- HMM models a progression through a sequence of hidden states, **x**t, with an observation **e**t at each step.
- To apply HMMs to information extraction, we can either build one big HMM for all the attributes or build a separate HMM for each attribute. We'll do the second.
- HMMs have **two big advantages** over FSAs for extraction.

  → First, HMMs are probabilistic, and thus tolerant to noise.

  →Second, HMMs can be trained from data; they don't require laborious engineering of templates, and thus they can more easily be kept up to date as text changes over time.

(iii) Conditional random fields for information extraction :

- One issue with HMMs for the information extraction task is that they model a lot of probabilities that we don't really need.
- Modeling this directly gives us some freedom. We don't need the independence assumptions of the Markov model—we can have an **x**t that is dependent on **x**1.
- A framework for this type of model is the **conditional random field**, or CRF, which models a

conditional probability distribution of a set of target variables given a set of observed variables.

- Like Bayesian networks, CRFs can represent many different structures of dependencies among the variables.
- One common structure is the **linear-chain conditional random field** for representing Markov dependencies among variables in a temporal sequence.
- Thus, HMMs are the temporal version of naive Bayes models, and linear-chain CRFs are the temporal version of logistic regression.

(iv) Ontology extraction from large corpora :

- So far we have thought of information extraction as finding a specific set of relations (e.g.,speaker, time, location) in a specific text (e.g., a talk announcement).
- A different application of extraction technology is building a large knowledge base or ontology of facts from a corpus.
- This is different in three ways:
- First it is open-ended—we want to acquire facts about all types of domains, not just one specific domain.
- Second, with a large corpus, this task is dominated by precision, not recall—just as with question answering on the Web .
- Third, the results can be statistical aggregates gathered from multiple sources, rather than being extracted from one specific text.

(v) Automated template construction :

- Fortunately, it is possible to *learn* templates from a few examples, then use the templates to learn more examples, from which more templates can be learned, and so on.
- In one of the first experiments of this kind, Brin (1999) started with a data set of just five examples:

("Isaac Asimov", "The Robots of Dawn")
("David Brin", "Startide Rising")
("James Gleick", "Chaos—Making a New Science")
("Charles Dickens", "Great Expectations")
("William Shakespeare", "The Comedy of Errors")

- Clearly these are examples of the author–title relation, but the learning system had no knowledge of authors or titles.
- The words in these examples were used in a search over a Web corpus, resulting in 199 matches. Each match is defined as a tuple of seven strings,

(*Author, Title, Order, Prefix, Middle, Postfix, URL*) ,

- where *Order* is true if the author came first and false if the title came first, *Middle* is the characters between the author and title, *Prefix* is the 10 characters before the match, *Suffix* is the 10 characters

after the match, and *URL* is the Web address where the match was made.

(vi) Machine reading :

- Automated template construction is a big step up from handcrafted template construction, but it still requires a handful of labeled examples of each relation to get started.
- To build a large ontology with many thousands of relations, even that amount of work would be onerous; we would like to have an extraction system with *no* human input of any kind—a system that could read on its own and build up its own database.
- Such a system would be relation-independent; would work for any relation. In practice, these systems work on *all* relations in parallel, because of the I/O demands of large corpora.
- They behave less like a traditional information extraction system that is targeted at a few relations and more like a human reader who learns from the text itself; because of this the field has been called **machine reading**.

ARTIFICIAL INTELLIGENCE

UNIT – IV

Syllabus

Natural Language for Communication: Phrase structure grammars, Syntactic Analysis, Augmented Grammars and semantic Interpretation, Machine Translation, Speech Recognition.

Perception: Image Formation, Early Image Processing Operations, Object Recognition by appearance, Reconstructing the 3D World, Object Recognition from Structural information, Using Vision.

Chapter – 1

Natural Language for Communication

INTRODUCTION

Communication is the intentional exchange of information brought about by the production SIGN and perception of signs drawn from a shared system of conventional signs. Most animals use signs to represent important messages: food here, predator nearby, approach, withdraw, let's mate.

4.1.1. PHRASE STRUCTURE GRAMMARS :

- The n-gram language models were based on sequences of words.
- The big issue for these models is **data sparsity**—with a vocabulary of, say, trigram probabilities to estimate, and so a corpus of even a trillion words will not be able to supply reliable estimates for all of them.
- We can address the problem of sparsity through generalization.
- Despite the exceptions, the notion of a **lexical category** (also known as a **part of speech**) such as *noun* or *adjective* is a useful generalization—useful in its own right, but more so when we string together lexical categories to form **syntactic categories** such as *noun phrase* or *verb phrase*, and combine these syntactic categories into trees representing the **phrase structure** of sentences: nested phrases, each marked with a category .

GENERATIVE CAPACITY :

- Grammatical formalisms can be classified by their **generative capacity**: the set of languages they can represent.
- Chomsky (1957) describes four classes of grammatical formalisms that differ only in the form of the rewrite rules.
- The classes can be arranged in a hierarchy, where each class can be used to describe all the languages that can be described by a less powerful class, as well as some additional languages.
- Here we list the hierarchy, most powerful class first:

1. **Recursively enumerable** grammars use unrestricted rules: both sides of the rewrite rules can have any number of terminal and nonterminal symbols, as in the rule A B C → D E.
    - These grammars are equivalent to Turing machines in their expressive power.
2. **Context-sensitive grammars** are restricted only in that the right-hand side must contain at least as many symbols as the left-hand side.
    - The name "contextsensitive" comes from the fact that a rule such as A X B → A Y B says that an X can be rewritten as a Y in the context of a preceding A and a following B.

    Context-sensitive grammars can represent languages such as (a sequence of n copies of a followed by the same number of bs and then cs).

3. In **context-free grammars** (or **CFG**s), the left-hand side consists of a single nonterminal symbol. Thus, each rule licenses rewriting the nonterminal as the right-hand side in *any* context.

   CFGs are popular for natural-language and programming-language grammars, although it is now widely accepted that at least some natural languages have constructions that are not context-free (Pullum, 1991).

   Context-free grammars can represent $a^n b^n$, but not $a^n b^n c^n$.

4. **Regular** grammars are the most restricted class. Every rule has a single nonterminal on the left-hand side and a terminal symbol optionally followed by a nonterminal on the right- hand side.

   Regular grammars are equivalent in power to finite state machines. They are poorly suited for programming languages, because they cannot represent constructs such as balanced opening and closing parentheses .

   The closest they can come is representing a□b□, a sequence of any number of as followed by any number of bs.

- There have been many competing language models based on the idea of phrase structure; we will describe a popular model called the **probabilistic context-free grammar**, or PCFG.
- A **grammar** is a collection of rules that defines a **language** as a set of allowable strings of words. Probabilistic means that the grammar assigns a probability to every string.
- Here is a PCFG rule:

  VP → Verb [0.70]
  VP NP [0.30]

  - Here VP (*verb phrase*) and NP (*noun phrase*) are **non-terminal symbols**. The grammar also refers to actual words, which are called **terminal symbols**.
  - This rule is saying that with probability 0.70 a verb phrase consists solely of a verb, and with probability 0.30 it is a VP followed by an NP.

(i) **The lexicon of**

- First we define the **lexicon**, or list of allowable words. The words are grouped into the lexical categories familiar to dictionary users: nouns, pronouns, and names to denote things; verbs to denote events; adjectives to modify nouns; adverbs to modify verbs; and function words: articles (such as *the*), prepositions (*in*), and conjunctions (*and*).
- Each of the categories ends in . . . to indicate that there are other words in the category.
- For nouns, names, verbs, adjectives, and adverbs, it is infeasible even in principle to list all the words. Not only are there tens of thousands of members in each class, but new ones–like *iPod* or

biodiesel—are being added constantly.

- These five categories are called **open classes**.
- For the categories of pronoun, relative pronoun, article, preposition, and conjunction we could have listed all the words with a little more work. These are called **closed classes**; they have a small number of words (a dozen or so).
- Closed classes change over the course of centuries, not months. For example, "thee" and "thou" were commonly used pronouns in the 17th century, were on the decline in the 19th, and are seen today only in poetry and some regional dialects.

(ii) **The Grammar of** $\mathcal{E}_0$

- The next step is to combine the words into phrases.

A grammar for $\mathcal{E}_0$ with rules for each of the six syntactic categories and an example for each rewrite rule.

$\mathcal{E}_0$:	S	→	$NP\ VP$	[0.90]	I + feel a breeze
		|	$S\ Conj\ S$	[0.10]	I feel a breeze + and + It stinks
	NP	→	$Pronoun$	[0.30]	I
		|	$Name$	[0.10]	John
		|	$Noun$	[0.10]	pits
		|	$Article\ Noun$	[0.25]	the + wumpus
		|	$Article\ Adjs\ Noun$	[0.05]	the + smelly dead + wumpus
		|	$Digit\ Digit$	[0.05]	3 4
		|	$NP\ PP$	[0.10]	the wumpus + in 1 3
		|	$NP\ RelClause$	[0.05]	the wumpus + that is smelly
	VP	→	$Verb$	[0.40]	stinks
		|	$VP\ NP$	[0.35]	feel + a breeze
		|	$VP\ Adjective$	[0.05]	smells + dead
		|	$VP\ PP$	[0.10]	is + in 1 3
		|	$VP\ Adverb$	[0.10]	go + ahead
	$Adjs$	→	$Adjective$	[0.80]	smelly
		|	$Adjective\ Adjs$	[0.20]	smelly + dead
	PP	→	$Prep\ NP$	[1.00]	to + the east
	$RelClause$	→	$RelPro\ VP$	[1.00]	that + is smelly

Figure 23.2 The grammar for $\mathcal{E}_0$, with example phrases for each rule. The syntactic categories are sentence (S), noun phrase (NP), verb phrase (VP), list of adjectives ($Adjs$), prepositional phrase (PP), and relative clause ($RelClause$).

4.1.2. SYNTACTIC ANALYSIS (PARSING) :

- **Parsing** is the process of analyzing a string of words to uncover its phrase structure, according to the rules of a grammar.
- Consider the following two sentences:

1. Have the students in section 2 of Computer Science 101 take the exam.

2. Have the students in section 2 of Computer Science 101 taken the exam?

- Even though they share the first 10 words, these sentences have very different parses, because the first is a command and the second is a question.
- By using left-to-right parsing algorithm would have to guess whether the first word is part of a command or a question and will not be able to tell if the guess is correct until at least the eleventh word, *take* or *taken.*
- If the algorithm guesses wrong, it will have to backtrack all the way to the first word and reanalyze the whole sentence under the other interpretation.
- To avoid this source of inefficiency we can use dynamic programming: *every time we analyze a substring, store the results so we won't have to reanalyze it later.*
- For example, once we discover that "the students in section 2 of Computer Science 101" is an NP, we can record that result in a data structure known as a **chart**.
- Algorithms that do this are called **chart parsers**.
- There are many types of chart parsers; we describe a bottom-up version called the **CYK algorithm**, after its inventors, **John Cocke, Daniel Younger, and Tadeo Kasami.**

CYK algorithm :

```
function CYK-PARSE(words, grammar) returns P, a table of probabilities
  N ← LENGTH(words)
  M ← the number of nonterminal symbols in grammar
  P ← an array of size [M, N, N], initially all 0
  /* Insert lexical rules for each word */
  for i = 1 to N do
    for each rule of form (X → words_i [p]) do
      P[X, i, 1] ← p
  /* Combine first and second parts of right-hand sides of rules, from short to long */
  for length = 2 to N do
    for start = 1 to N − length + 1 do
      for len1 = 1 to N − 1 do
        len2 ← length − len1
        for each rule of the form (X → Y Z [p]) do
          P[X, start, length] ← MAX(P[X, start, length],
                                    P[Y, start, len1] × P[Z, start + len1, len2] × p)
  return P
```

Figure 23.5 The CYK algorithm for parsing. Given a sequence of words, it finds the most probable derivation for the whole sequence and for each subsequence. It returns the whole table, P, in which an entry $P[X, start, len]$ is the probability of the most probable X of length len starting at position $start$. If there is no X of that size at that location, the probability is 0.

- The CYK algorithm requires a grammar with all rules in one of two very specific formats: lexical rules of the form X → **word**, and syntactic rules of the form X → Y Z .
- This grammar format, called **Chomsky Normal Form**, may seem restrictive, but it is not: any context-free grammar can be automatically transformed into Chomsky Normal Form.

(i) **Learning probabilities for PCFGs :**

- A PCFG has many rules, with a probability for each rule.
- This suggests that **learning** the grammar from data might be better than a knowledge engineering approach.
- Learning is easiest if we are given a corpus of correctly parsed sentences, commonly called a **treebank**.

The Penn Treebank is the best known; it consists of 3 million words which have been annotated with part of speech and parse-tree structure, using human labor assisted by some automated tools.

- Annotated tree from the Penn Treebank :

```
[ [S [NP-SBJ-2 Her eyes]
    [VP were
        [VP glazed
            [NP *-2]
            [SBAR-ADV as if
                [S [NP-SBJ she]
                    [VP did n't
                        [VP [VP hear [NP *-1]]
                            or
                            [VP [ADVP even] see [NP *-1]]
                            [NP-1 him]]]]]]]
 .]
```

Figure 23.6 Annotated tree for the sentence "Her eyes were glazed as if she didn't hear or even see him." from the Penn Treebank. Note that in this grammar there is a distinction between an object noun phrase (*NP*) and a subject noun phrase (*NP-SBJ*). Note also a grammatical phenomenon we have not covered yet: the movement of a phrase from one part of the tree to another. This tree analyzes the phrase "hear or even see him" as consisting of two constituent *VP*s, [VP hear [NP *-1]] and [VP [ADVP even] see [NP *-1]], both of which have a missing object, denoted *-1, which refers to the *NP* labeled elsewhere in the tree as [NP-1 him].

- Given a corpus of trees, we can create a PCFG just by counting (and smoothing).
- In the example above, there are two nodes of the form [S[NP . . .][VP . . .]]. We would count these, and all the other subtrees with root S in the corpus.

If there are 100,000 S nodes of which 60,000 are of this form, then we create the rule:

S → NP VP [0.60] .

(ii) Comparing context-free and Markov models :

- The problem with PCFGs is that they are context-free.
- That means that the difference between P ("eat a banana") and P ("eat a bandanna") depends only on P (Noun → "banana") versus
  P (Noun → "bandanna") and not on the relation between "eat" and the respective objects.
- A Markov model of order two or more, given a sufficiently large corpus, *will* know that "eat a banana" is more probable.
- We can combine a PCFG and Markov model to get the best of both. The simplest approach is to estimate the probability of a sentence with the geometric mean of the probabilities computed by both models.

Another problem with PCFGs is that they tend to have too strong a preference for shorter sentences.

4.1.3. AUGMENTED GRAMMARS AND SEMANTIC INTERPRETATION :

- In this concept, we see how to extend context-free grammars.

(i) **Lexicalized PCFGs** :

To get at the relationship between the verb "eat" and the nouns "banana" versus "bandanna," we can use a **lexicalized PCFG**, in which the probabilities for a rule depend on the relationship between words in the parse tree, not just on the adjacency of words in a sentence.

- Of course, we can't have the probability depend on every word in the tree, because we won't have enough training data to estimate all those probabilities.
- It is useful to introduce the notion of the **head** of a phrase—the most important word. Thus, "eat" is the head of the VP "eat a banana" and "banana" is the head of the NP "a banana."
- We use the notation VP(v) to denote a phrase with category VP whose head word is v. We say that the category VP is **augmented** with the head variable v.

Here is an **augmented grammar** that describes theverb–object relation:

$$
\begin{array}{lll}
VP(v) \rightarrow Verb(v)\ NP(n) & [P_1(v,n)] \\
VP(v) \rightarrow Verb(v) & [P_2(v)] \\
NP(n) \rightarrow Article(a)\ Adjs(j)\ Noun(n) & [P_3(n,a)] \\
Noun(\mathbf{banana}) \rightarrow \mathbf{banana} & [p_n] \\
\ldots & \ldots
\end{array}
$$

(ii) **Formal definition of augmented grammar rules:**

- Augmented rules are complicated, so we will give them a formal definition by showing how an augmented rule can be translated into a logical sentence.

- The sentence will have the form of a definite clause, so the result is called a **definite clause grammar**, or DCG.

  That gives us

$$Article(a, s_1) \wedge Adjs(j, s_2) \wedge Noun(n, s_3) \wedge Compatible(j, n) \Rightarrow NP(n, Append(s_1, s_2, s_3)) .$$

- This definite clause says that if the predicate Article is true of a head word a and a string s1, and Adjs is similarly true of a head word j and a string s2, and Noun is true of a head word n and a string s3, and if j and n are compatible, then the predicate NP is true of the head
  word n and the result of appending strings s1, s2, and s3.
- The translation from grammar rule to definite clause allows us to talk about parsing as logical inference.
- This makes it possible to reason about languages and strings in many different ways.

  For example, it means we can do bottom-up parsing using forward chaining or top-down parsing using backward chaining .
- In fact, parsing natural language with DCGs was one of the first applications of (and motivations for) the Prolog logic programming language.
- It is sometimes possible to run the process backward and do **language generation** as well as parsing.

(iii) **Case agreement and subject–verb agreement:**

- We splitting NP into two categories, NPS and NPO, to stand for noun phrases in the subjective and objective case, respectively.
- We would also need to split the category Pronoun into the two categories PronounS (which includes “I”) and PronounO (which includes “me”).
- The top part of Figure shows the grammar for **case agreement**; we call the resulting language $\mathcal{E}_1$

$\mathcal{E}_1$:

$$
\begin{aligned}
S &\rightarrow NP_S\ VP \mid \ldots \\
NP_S &\rightarrow Pronoun_S \mid Name \mid Noun \mid \ldots \\
NP_O &\rightarrow Pronoun_O \mid Name \mid Noun \mid \ldots \\
VP &\rightarrow VP\ NP_O \mid \ldots \\
PP &\rightarrow Prep\ NP_O \\
Pronoun_S &\rightarrow \textbf{I} \mid \textbf{you} \mid \textbf{he} \mid \textbf{she} \mid \textbf{it} \mid \ldots \\
Pronoun_O &\rightarrow \textbf{me} \mid \textbf{you} \mid \textbf{him} \mid \textbf{her} \mid \textbf{it} \mid \ldots \\
&\ldots
\end{aligned}
$$

$\mathcal{E}_2$:

$$
\begin{aligned}
S(head) &\rightarrow NP(Sbj, pn, h)\ VP(pn, head) \mid \ldots \\
NP(c, pn, head) &\rightarrow Pronoun(c, pn, head) \mid Noun(c, pn, head) \mid \ldots \\
VP(pn, head) &\rightarrow VP(pn, head)\ NP(Obj, p, h) \mid \ldots \\
PP(head) &\rightarrow Prep(head)\ NP(Obj, pn, h) \\
Pronoun(Sbj, 1S, \textbf{I}) &\rightarrow \textbf{I} \\
Pronoun(Sbj, 1P, \textbf{we}) &\rightarrow \textbf{we} \\
Pronoun(Obj, 1S, \textbf{me}) &\rightarrow \textbf{me} \\
Pronoun(Obj, 3P, \textbf{them}) &\rightarrow \textbf{them} \\
&\ldots
\end{aligned}
$$

Figure 23.7 Top: part of a grammar for the language $\mathcal{E}_1$, which handles subjective and objective cases in noun phrases and thus does not overgenerate quite as badly as $\mathcal{E}_0$. The portions that are identical to $\mathcal{E}_0$ have been omitted. Bottom: part of an augmented grammar for $\mathcal{E}_2$, with three augmentations: case agreement, subject–verb agreement, and head word. *Sbj*, *Obj*, *1S*, *1P* and *3P* are constants, and lowercase names are variables.

- Unfortunately, E1 still overgenerates. English requires **subject–verb agreement** for person and number of the subject and main verb of a sentence.

  For example, if "I" is the subject, then "I smell" is grammatical, but "I smells" is not. If "it" is the subject, we get the reverse.

(iv) Semantic interpretation :

To show how to add semantics to a grammar, we start with an example that is simpler than English: the semantics of arithmetic expressions.

$$
\begin{aligned}
Exp(x) &\rightarrow Exp(x_1)\ Operator(op)\ Exp(x_2)\ \{x = Apply(op, x_1, x_2)\} \\
Exp(x) &\rightarrow (\ Exp(x)\ ) \\
Exp(x) &\rightarrow Number(x) \\
Number(x) &\rightarrow Digit(x) \\
Number(x) &\rightarrow Number(x_1)\ Digit(x_2)\ \{x = 10 \times x_1 + x_2\} \\
Digit(x) &\rightarrow x\ \{0 \leq x \leq 9\} \\
Operator(x) &\rightarrow x\ \{x \in \{+, -, \div, \times\}\}
\end{aligned}
$$

Figure 23.8 A grammar for arithmetic expressions, augmented with semantics. Each variable x_i represents the semantics of a constituent. Note the use of the $\{test\}$ notation to define logical predicates that must be satisfied, but that are not constituents.

- Figure shows a grammar for arithmetic expressions, where each rule is augmented with a variable indicating the semantic interpretation of the phrase.
- The semantics of a digit such as "3" is the digit itself. The semantics of an expression such as "3 + 4" is the operator "+" applied to the semantics of the phrase "3" and the phrase "4."

The rules obey the principle of **compositional semantics.**

- The semantics of a phrase is a function of the semantics of the subphrases. Figure shows the parse tree for 3 + (4 ÷ 2) according to this grammar. The root of the parse tree is Exp(5), an expression whose semantic interpretation is 5.

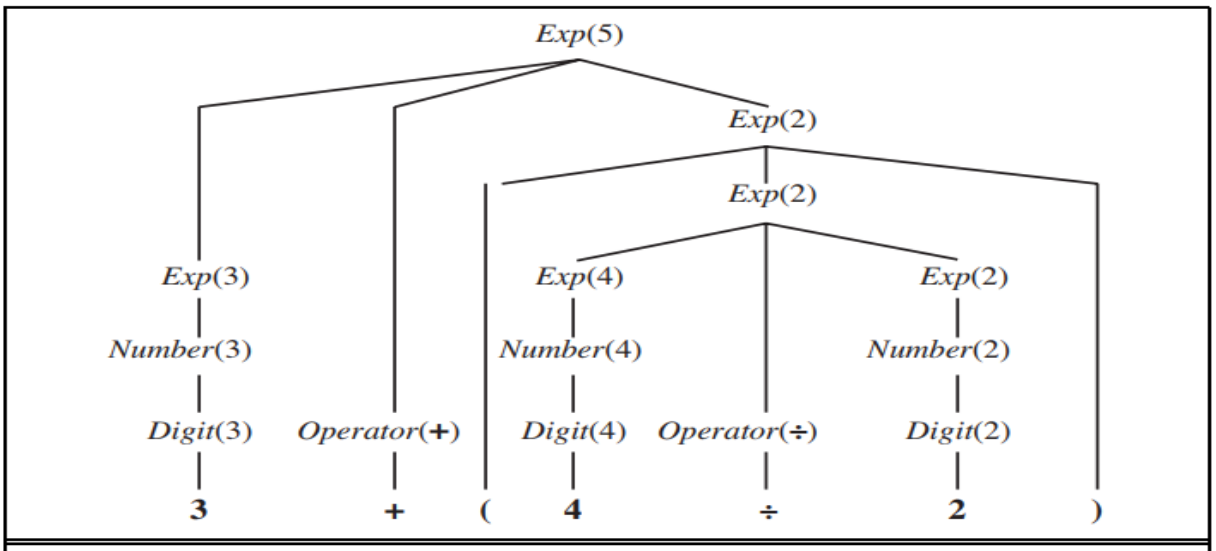


Figure 23.9 Parse tree with semantic interpretations for the string "3 + (4 ÷ 2)".

(v) Complications :

- The grammar of real English is endlessly complex. We will briefly mention some examples.

1. **Time and tense:**

Suppose we want to represent the difference between "John loves Mary" and "John loved Mary."

- English uses verb tenses (past, present, and future) to indicate the relative time of an event. One good choice to represent the time of events is the event calculus notation
- In event calculus we have

  John loves mary: E1 □ Loves(John, Mary) □ During(Now, Extent(E1))
  John loved mary: E2 □ Loves(John, Mary) □ After(Now, Extent(E2)) .

2. **Quantification** :

- Consider the sentence "Every agent feels a breeze."

The sentence has only one syntactic parse under E0, but it is actually semantically ambiguous; the preferred meaning is "For every agent there exists a breeze that the agent feels," but an acceptable alternative meaning is "There exists a breeze that every agent feels.

3. Pragmatics:

- We have shown how an agent can perceive a string of words and use a grammar to derive a set of possible semantic interpretations.

4. Long-distance dependencies:

Questions introduce a new grammatical complexity. In "Who did the agent tell you to give the gold to?" the final word "to" should be parsed as [PP to], where the " " denotes a gap or **trace** where an NP is missing; the missing NP is licensed by the first word of the sentence, "who."

5. Ambiguity :

In some cases, hearers are consciously aware of ambiguity in an utterance.

Types of ambiguities :

→ **Lexical ambiguity**, in which a word has more than one meaning.

→ **Syntactic ambiguity,** refers to a phrase that has multiple parses.

→**Semantic ambiguity,** The syntactic ambiguity leads to a **semantic ambiguity**, because one parse means the other.

- **Disambiguation,** is the process of recovering the most probable intended meaning of an utterance.
- To do disambiguation properly, we need to combine four models:

→ **world model**

→ **mental model**

→ **language model**

→ **acoustic mode**

4.1.4. MACHINE TRANSLATION :

- Machine translation is the automatic translation of text from one natural language (the source) to another (the target).

It was one of the first application areas envisioned for computers (Weaver, 1949), but it is only in the past decade that the technology has seen widespread usage.

- Historically, there have been **three main applications** of machine translation.

→ ***Rough translation***, as provided by free online services, gives the "gist" of a foreign sentence or document, but contains errors.

→ ***Pre-edited translation*** is used by companies to publish their documentation and sales materials in multiple languages.

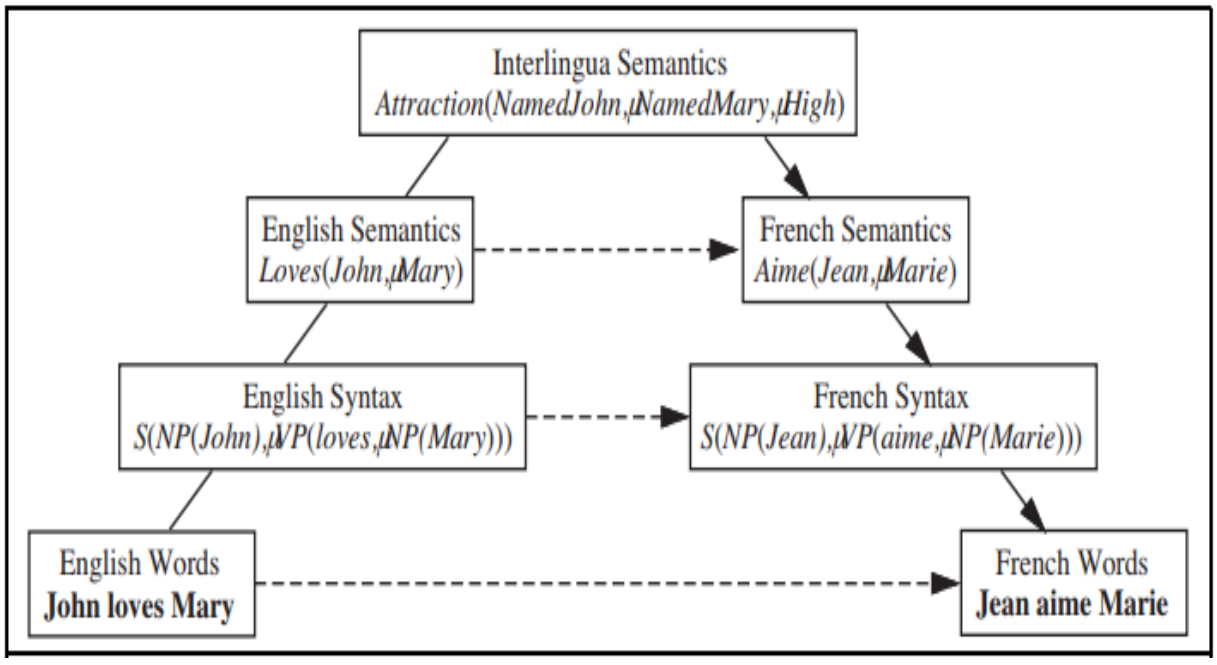
The original source text is written in a constrained language that is easier to translate automatically, and the results are usually edited by a human to correct any errors.

→ ***Restricted-source translation*** works fully automatically, but only on highly stereotypical language, such as a weather report.

- Translation is difficult because, in the fully general case, it requires in-depth understanding of the text. This is true even for very simple texts—even "texts" of one word.

(i) **Machine translation systems** :

- All translation systems must model the source and target languages, but systems vary in the type of models they use.
- Some systems attempt to analyze the source language text all the way into an interlingua knowledge representation and then generate sentences in the target language from that representation.
- This is difficult because it involves **three unsolved problems**:
    - creating a complete knowledge representation of everything;
    - parsing into that representation; and
    - generating sentences from that representation.
- Other systems are based on a **transfer model**.
- They keep a database of translation rules (or examples), and whenever the rule (or example) matches, they translate directly.
- Transfer can occur at the lexical, syntactic, or semantic level.
- For example, a strictly syntactic rule maps English [*Adjective Noun*] to French [*Noun Adjective*]. A mixed syntactic and lexical rule maps French [S1 "et puis" S2] to English [S1 "and then" S2].



(ii) **Statistical machine translation** :

- Now that we have seen how complex the translation task can be, it should come as no surprise that the most successful machine translation systems are built by training a probabilistic model using statistics gathered from a large corpus of text.
- This approach does not need a complex ontology of interlingua concepts, nor does it need handcrafted grammars of the source and target languages, nor a hand-labeled treebank.
- All it needs is data—sample translations from which a translation model can be learned. To translate a sentence in, say, English (e) into French (f), we find the string of words f □ that maximizes

$$f^* = \underset{f}{\operatorname{argmax}}\, P(f \mid e) = \operatorname{argmax}\, P(e \mid f)\, P(f)\ .$$

- Here the factor P (f) is the target **language model** for French; it says how probable a given sentence is in French. P (e|f) is the **translation mode.**
- All that remains is to learn the phrasal and distortion probabilities. We sketch the procedure;

➔ **Find parallel texts**:

First, gather a parallel bilingual corpus. For example, a **Hansard** is a record of parliamentary debate. Canada, Hong Kong, and other countries produce bilingual Hansards, the European Union publishes its official documents in 11 languages, and the United Nations publishes multilingual documents.

➔ **Segment into sentences**: The unit of translation is a sentence, so we will have to break the corpus into sentences. Periods are strong indicators of the end of a sentence.

➔ **Align sentences**: For each sentence in the English version, determine what sentence(s) it corresponds to in the French version.

Usually, the next sentence of English corresponds to the next sentence of French in a 1:1 match, but sometimes there is variation: one sentence in one language will be split into a 2:1 match, or the order of two sentences will be swapped, resulting in a 2:2 match.

➔ **Align phrases**: Within a sentence, phrases can be aligned by a process that is similar to that used for sentence alignment, but requiring iterative improvement.

➔ **Extract distortions**: Once we have an alignment of phrases we can define distortion probabilities. Simply count how often distortion occurs in the corpus for each distance d = 0, ±1, ±2, . . ., and apply smoothing.

➔ **Improve estimates with EM**: Use expectation–maximization to improve the estimates of P(f | e) and P(d) values.

We compute the best alignments with the current values of these parameters in the E step, then

update the estimates in the M step and iterate the process until convergence.

4.1.5. SPEECH RECOGNITION :

- **Speech recognition** is the task of identifying a sequence of words uttered by a speaker, given the acoustic signal.
- It has become one of the mainstream applications of AI—millions of people interact with speech recognition systems every day to navigate voice mail systems, search the Web from mobile phones, and other applications.
- Speech is an attractive option when hands-free operation is necessary, as when operating machinery.

  Speech recognition is difficult because the sounds made by a speaker are ambiguous and, well, noisy.
- Several issues that make speech problematic :
- First, **segmentation**: written words in English have spaces between them, but in fast speech there are no pauses in "wreck a nice" that would distinguish it as a multiword phrase as opposed to the single word "recognize."
- Second, **coarticulation**: when speaking quickly the "s" sound at the end of "nice" merges with the "b" sound at the beginning of "beach," yielding something that is close to a "sp."
- Another problem that does not show up in this example is **homophones**—words like "to," "too," and "two" that sound the same but differ in meaning.
- As usual, the most likely sequence can be computed with the help of Bayes' rule to be:

$$\underset{word_{1:t}}{\operatorname{argmax}} P(word_{1:t} \mid sound_{1:t}) = \underset{word_{1:t}}{\operatorname{argmax}} P(sound_{1:t} \mid word_{1:t}) P(word_{1:t}) \ .$$

- Here P(sound 1:t|word 1:t) is the **acoustic model**. It describes the sounds of words—that "ceiling" begins with a soft "c" and sounds the same as "sealing."

  P(word 1:t) is known as the **language model**. It specifies the prior probability of each utterance—for example, that "ceiling fan" is about 500 times more likely as a word sequence than "sealing fan."
- This approach was named the **noisy channel model** by Claude Shannon (1948).
- He described a situation in which an original message (the *words* in our example) is transmitted over a noisy channel (such as a telephone line) such that a corrupted message (the *sounds* in our example) are received at the other end.
- Once we define the acoustic and language models, we can solve for the most likely sequence of

words using the Viterbi algorithm .

- Most speech recognition systems use a language model that makes the Markov assumption—that the current state Word t depends only on a fixed number n of previous states—and represent Word t as a single random variable taking on a finite set of values, which makes it a Hidden Markov Model (HMM).

Thus, speech recognition becomes a simple application of the HMM methodology,

(i) **Acoustic model** :

- Sound waves are periodic changes in pressure that propagate through the air. When these waves strike the diaphragm of a microphone, the back-and-forth movement generates an electric current.
- An analog-to-digital converter measures the size of the current—which approximates the amplitude of the sound wave—at discrete intervals called the **sampling rate**.

  Speech sounds, which are mostly in the range of 100 Hz (100 cycles per second) to 1000 Hz, are typically sampled at a rate of 8 kHz. (CDs and mp3 files are sampled at 44.1 kHz.)
- The precision of each measurement is determined by the **quantization factor**; speech recognizers typically keep 8 to 12 bits.
- That means that a low-end system, sampling at 8 kHz with 8-bit quantization, would require nearly half a megabyte per minute of speech.
- Since we only want to know what words were spoken, not exactly what they sounded like, we don't need to keep all that information.
- We only need to distinguish between different speech sounds. Linguists have identified about 100 speech sounds, or **phones**, that can be composed to form all the words in all known human languages.
- Roughly speaking, a phone is the sound that corresponds to a single vowel or consonant, but there are some complications: combinations of letters, such as "th" and "ng" produce single phones, and some letters produce different phones in different contexts all the phones that are used in English.
- A **phoneme** is the smallest unit of sound that has a distinct meaning to speakers of a particular language.
- For example, the "t" in "stick" sounds similar enough to the "t" in "tick" that speakers of English consider them the same phoneme.
- First, we observe that although the sound frequencies in speech may be several kHz, the *changes* in the content of the signal occur much less often, perhaps at no more than 100 Hz.

- Therefore, speech systems summarize the properties of the signal over time slices called **frames**.

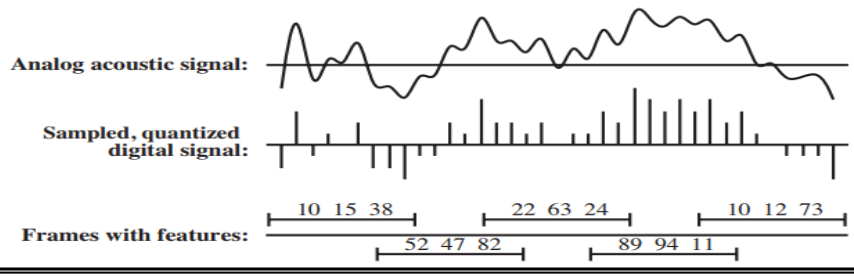


Figure 23.15 Translating the acoustic signal into a sequence of frames. In this diagram each frame is described by the discretized values of three acoustic features; a real system would have dozens of features.

(ii) Language model :

- For general-purpose speech recognition, the language model can be an n-gram model of text learned from a corpus of written sentences.
- However, spoken language has different characteristics than written language, so it is better to get a corpus of transcripts of spoken language.
- For task-specific speech recognition, the corpus should be task-specific: to build your airline reservation system, get transcripts of prior calls.

  It also helps to have task-specific vocabulary, such as a list of all the airports and cities served, and all the flight numbers.

(iii) **Building a speech recognizer** :

- The quality of a speech recognition system depends on the quality of all of its components— the language model, the word-pronunciation models, the phone models, and the signal processing algorithms used to extract spectral features from the acoustic signal.
- The accuracy of a system depends on a number of factors. First, the quality of the signal matters: a high-quality directional microphone aimed at a stationary mouth in a padded room will do much better than a cheap microphone transmitting a signal over phone lines from a car in traffic with the radio playing.

PERCEPTION:

Perception provides agents with information about the world they inhabit by interpreting the response of **sensors**.

A sensor measures some aspect of the environment in a form that can be used as input by an agent program. The sensor could be as simple as a switch, which gives one bit telling whether it is on or off, or as complex as the eye. A variety of sensory modalities are available to artificial agents.

Some robots do **active sensing**, meaning they send out a signal, such as radar or ultrasound, and sense the reflection of this signal off of the environment.

The problem for a vision-capable agent then is: Which aspects of the rich visual stimulus should be considered to help the agent make good action choices, and which aspects should be ignored? Vision—and all perception—serves to further the agent's goals, not as an end to itself.

In the **recognition** approach an agent draws distinctions among the objects it encounters based on visual and other information. Recognition could mean labelling each image with a yes or no as to whether it contains food that we should forage, or contains Grandma's face.

IMAGE FORMATION

Imaging distorts the appearance of objects. For example, a picture taken looking down a long straight set of railway tracks will suggest that the rails converge and meet. As another example, if you hold your hand in front of your eye, you can block out the moon, which is not smaller than your hand. As you move your hand back and forth or tilt it, your hand will seem to shrink and grow in the image, but it is not doing so in reality. Models of these effects are essential for both recognition and reconstruction.

Images without lenses: The pinhole camera

Image sensors gather light scattered from objects in a **scene** and create a two-dimensional **image**. In the eye, the image is formed on the retina, which consists of two types of cells: about 100 million rods, which are sensitive to light at a wide range of wavelengths, and 5 million cones. Cones, which are essential for colour vision, are of three main types, each of which is sensitive to a different set of wavelengths.

In cameras, the image is formed on an image plane, which can be a piece of film coated with silver halides or a rectangular grid of a few million photosensitive **pixels**, each a complementary metal-oxide semiconductor (CMOS) or charge-coupled device (CCD).

Each photon arriving at the sensor produces an effect, whose strength depends on the wavelength of the photon. The output of the sensor is the sum of all effects due to photons observed in some time

window, meaning that image sensors report a weighted average of the intensity of light arriving at the sensor.

To see a focused image, we must ensure that all the photons from approximately the same spot in the scene arrive at approximately the same point in the image plane.

The simplest way to form a focused image is to view stationary objects with a **pinhole camera**, which consists of a pinhole opening, O, at the front of a box, and an image plane at the back of the box. Photons from the scene must pass through the pinhole, so if it is small enough then nearby photons in the scene will be nearby in the image plane, and the image will be in focus.

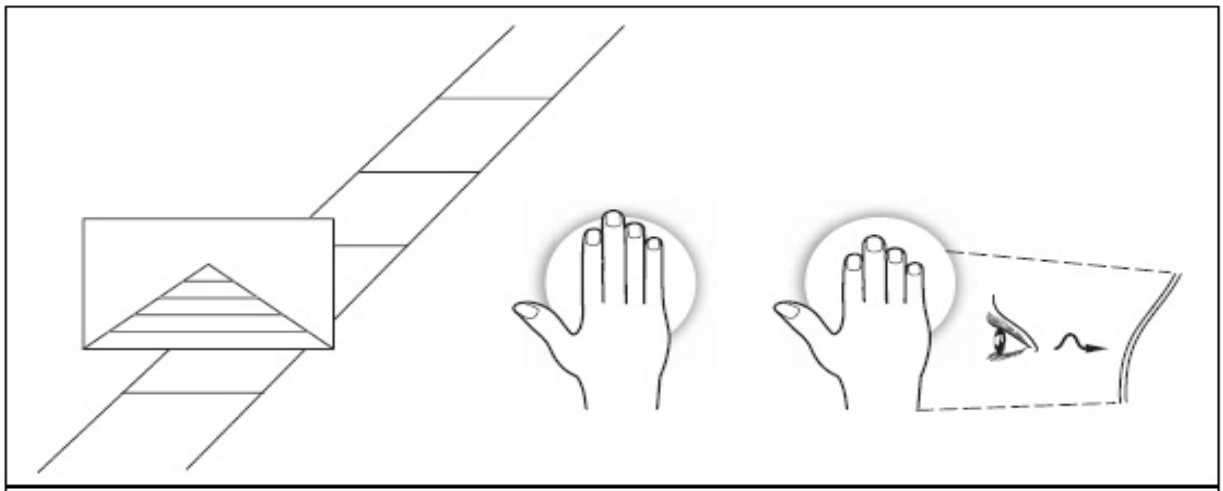
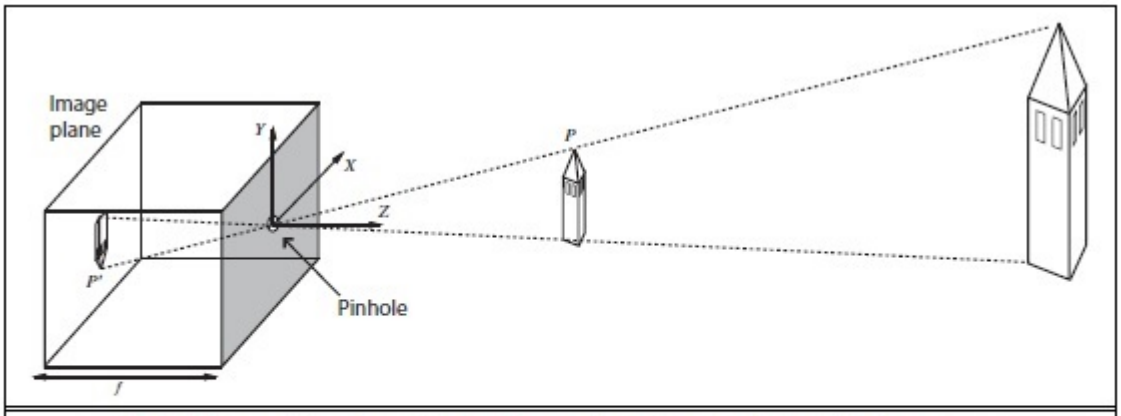


Figure 24.1 Imaging distorts geometry. Parallel lines appear to meet in the distance, as in the image of the railway tracks on the left. In the center, a small hand blocks out most of a large moon. On the right is a foreshortening effect: the hand is tilted away from the eye, making it appear shorter than in the center figure.

The geometry of scene and image is easiest to understand with the pinhole camera. We use a three-dimensional coordinate system with the origin at the pinhole, and consider a point P in the scene, with coordinates (X, Y,Z). P gets projected to the point P'_ in the image plane with coordinates (x, y, z). If f is the distance from the pinhole to the image plane, then by similar triangles, we can derive the following equations.

$$\frac{-x}{f} = \frac{X}{Z}, \frac{-y}{f} = \frac{Y}{Z} \quad \Rightarrow \quad x = \frac{-fX}{Z}, y = \frac{-fY}{Z} .$$

These equations define an image-formation process known as **perspective projection**.



Each light-sensitive element in the image plane at the back of a pinhole camera receives light from a the small range of directions that passes through the pinhole. If the pinhole is small enough, the result is a focused image at the back of the pinhole. The process of projection means that large, distant objects look the same as smaller, nearby objects. Note that the image is projected upside down.

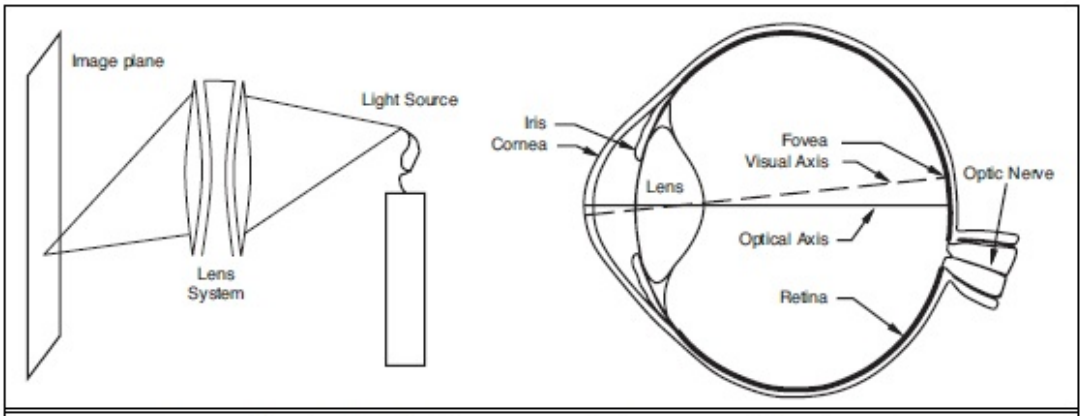
Under perspective projection, distant objects look small. This is what allows you to cover the moon with your hand (Figure 24.1). An important result of this effect is that parallel lines converge to a point on the horizon. (Think of railway tracks, Figure 24.1.) A line in the scene in the direction (U, V,W) and passing through the point (X0, Y0, Z0) can be described as the set of points (X0 + λU, Y0 + λV,Z0 + λW), with λ varying between −∞ and +∞. Different choices of (X0, Y0, Z0) yield different lines parallel to one another. The projection of a point Pλ from this line onto the image plane is given by

$$\left(f\frac{X_0 + \lambda U}{Z_0 + \lambda W}, f\frac{Y_0 + \lambda V}{Z_0 + \lambda W} \right)$$

As λ→ ∞or λ → −∞, this becomes p∞ = (fU/W, fV/W) if W _= 0. This means that two parallel lines leaving different points in space will converge in the image—for large λ, the image points are nearly the same, whatever the value of (X0, Y0, Z0) (again, think railway tracks,). We call p∞ the **vanishing point** associated with the family of straight lines with direction (U, V,W). Lines with the same direction share the same vanishing point.

Lens systems

The drawback of the pinhole camera is that we need a small pinhole to keep the image in focus. But the smaller the pinhole, the fewer photons get through, meaning the image will be dark. We can gather more photons by keeping the pinhole open longer, but then we will get **motion blur**—objects in the scene that move will appear blurred because they send photons to multiple locations on the image plane. If we can’t keep the pinhole open longer, we can try to make it bigger. More light will enter, but light from a small patch of object in the scene will now be spread over a patch on the image plane, causing a blurred image.



Lenses collect the light leaving a scene point in a range of directions, and steer it all to arrive at a single point on the image plane. Focusing works for points lying close to a focal plane in space; other points will not be focused properly. In cameras, elements of the lens system move to change the focal plane, whereas in the eye, the shape of the lens is changed by specialized muscles.

Vertebrate eyes and modern cameras use a **lens** system to gather sufficient light while keeping the image in focus. A large opening is covered with a lens that focuses light from nearby object locations down to nearby locations in the image plane. However, lens systems have a limited **depth of field**: they can focus light only from points that lie within a range of depths (centered around a **focal plane**). Objects outside this range will be out of focus in the image. To move the focal plane, the lens in the eye can change shape in a camera, the lenses move back and forth.

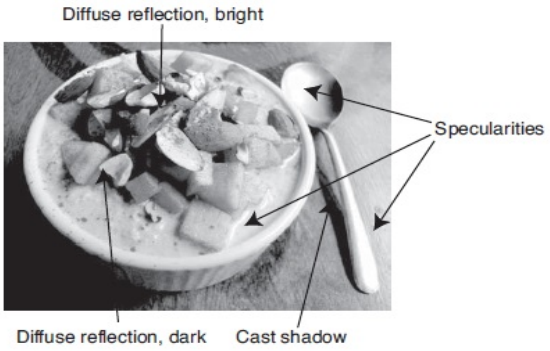
Scaled orthographic projection

Perspective effects aren't always pronounced. For example, spots on a distant leopard may look small because the leopard is far away, but two spots that are next to each other will have about the same size. This is because the difference in distance to the spots is small compared to the distance to them, and so we can simplify the projection model. The appropriate model is **scaled orthographic projection**. The idea is as follows: If the depth Z of points on the object varies within some range Z0 +- ΔZ, with ΔZ * Z0, then the perspective scaling factor f/Z can be approximated by a constant s = f/Z0. The equations for projection from the scene coordinates (X, Y,Z) to the image plane become x = sX and y = sY . Scaled orthographic projection is an approximation that is valid only for those parts of the scene with not much internal depth variation. For example, scaled orthographic projection can be a good model for the features on the front of a distant building.

Light and shading

The brightness of a pixel in the image is a function of the brightness of the surface patch in the scene that projects to the pixel. We will assume a linear model (current cameras have nonlinearities at the extremes of light and dark, but are linear in the middle). Image brightness is a strong, if ambiguous, cue to the shape of an object, and from there to its identity. People are usually able to distinguish the three main causes of varying brightness and reverse-engineer the object's properties.

The first cause is **overall intensity** of the light. Even though a white object in shadow may be less bright than a black object in direct sunlight, the eye can distinguish relative brightness well, and perceive the white object as white. Second, different points in the scene may **reflect** more or less of the light. Third, surface patches facing the light are brighter than surface patches tilted away from the light, an effect known as **shading**. Most surfaces reflect light by a process of **diffuse reflection**. Diffuse reflection scatters light evenly across the directions leaving a surface, so the brightness of a diffuse surface doesn't depend on the viewing direction. The behaviour of a perfect mirror is known as specular **reflection**. Some surfaces—such as brushed metal, plastic, or a wet floor—display small patches where specular reflection has occurred, called **secularities**. These are easy to identify, because they are small and bright. For almost all purposes, it is enough to model all surfaces as being diffuse with secularities.



A variety of illumination effects. There are specularities on the metal spoon and on the milk. The bright diffuse surface is bright because it faces the light direction. The dark diffuse surface is dark because it is tangential to the illumination direction. The shadows appear at surface points that cannot see the light source. Photo by Mike Linksvayer (mlinksva on flickr).

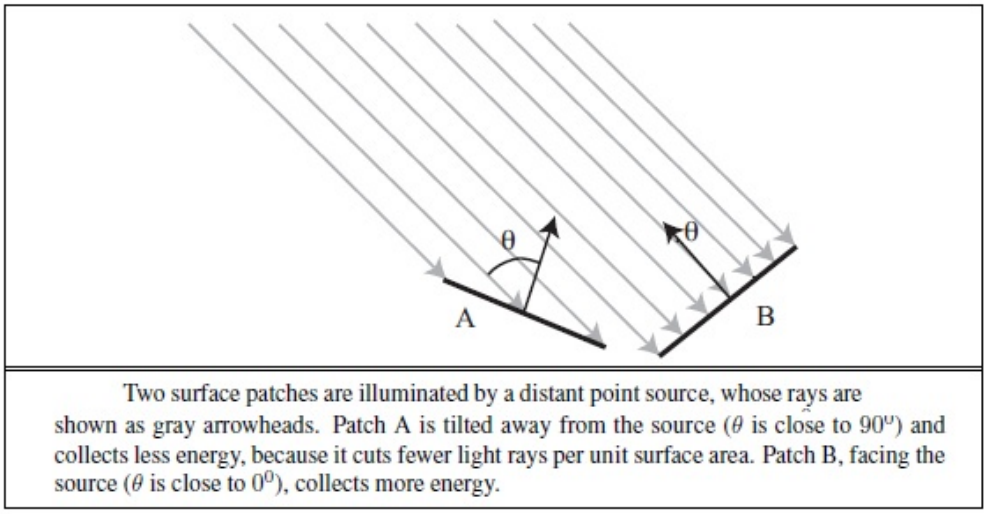
The main source of illumination outside is the sun, whose rays all travel parallel to one another. We model this behaviour as a **distant point light source**. This is the most important model of lighting, and is quite effective for indoor scenes as well as outdoor scenes. The amount of light collected by a surface patch in this model depends on the angle θ between the illumination direction and the normal to the surface.

A diffuse surface patch illuminated by a distant point light source will reflect some fraction of the light it collects; this fraction is called the **diffuse albedo**. White paper and snow have a high albedo, about 0.90, whereas flat black velvet and charcoal have a low albedo of about 0.05 (which means that 95% of

the incoming light is absorbed within the fibres of the velvet or the pores of the charcoal). **Lambert's cosine law** states that the brightness of a diffuse patch is given by

$$I = \rho I_0 \cos\theta$$

where ρ is the diffuse albedo, I0 is the intensity of the light source and θ is the angle between the light source direction and the surface Lampert's law predicts bright image pixels come from surface patches that face the light directly and dark pixels come from patches that see the light only tangentially, so that the shading on a surface provides some shape information. If the surface is not reached by the light source, then it is in **shadow**. Shadows are very seldom a uniform black, because the shadowed surface receives some light from other sources. Outdoors, the most important such source is the sky, which is quite bright. Indoors, light reflected from other S surfaces illuminates shadowed patches. These **interreflections** can have a significant effect on the brightness of other surfaces, too. These effects are sometimes modelled by adding a constant **ambient illumination** term to the predicted intensity.



Two surface patches are illuminated by a distant point source, whose rays are shown as gray arrowheads. Patch A is tilted away from the source (θ is close to 90^0) and collects less energy, because it cuts fewer light rays per unit surface area. Patch B, facing the source (θ is close to 0^0), collects more energy.

Colour

Fruit is a bribe that a tree offers to animals to carry its seeds around. Trees have evolved to have fruit that turns red or yellow when ripe, and animals have evolved to detect these colour changes. Light arriving at the eye has different amounts of energy at different wavelengths; this can be represented by a spectral energy density function. Human eyes respond to light in the 380–750nm wavelength region, with three different types of colour receptor cells, which have peak receptiveness at 420mm (blue), 540nm (green), and 570nm (red). The human eye can capture only a small fraction of the full spectral energy density function—but it is enough to tell when the fruit is ripe.

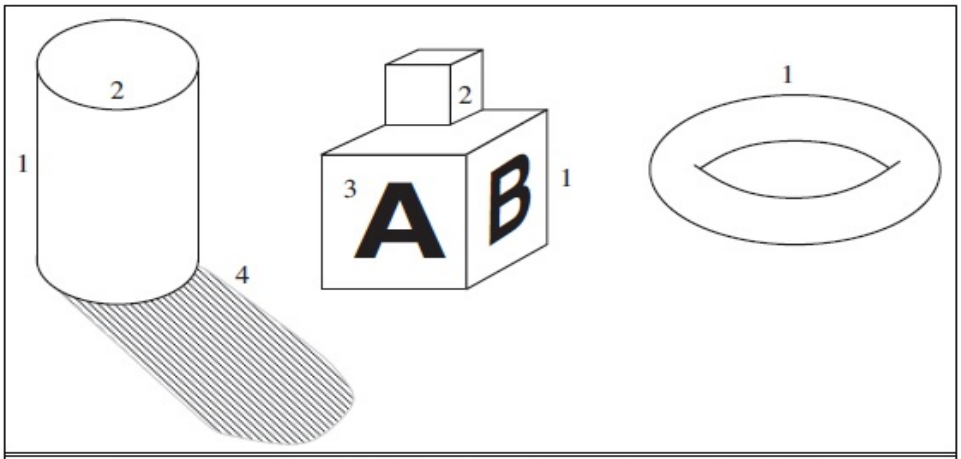
The **principle of trichromacy** states that for any spectral energy density, no matter how complicated, it is possible to construct another spectral energy density consisting of a mixture of just three colours—usually red, green, and blue—such that a human can't tell the difference between the

two. That means that our TVs and computer displays can get by with just the three red/green/blue (or R/G/B) colour elements. It makes our computer vision algorithms easier, too. Each surface can be modelled with three different albedos for R/G/B. Similarly, each light source can be modelled with three R/G/B intensities. We then apply Lambert's cosine law to each to get three R/G/B pixel values. This model predicts, correctly, that the same surface will produce different coloured image patches under different-coloured lights. In fact, human observers are quite good at ignoring the effects of different coloured lights and are able to estimate the colour of the surface under white light, an effect known as **colour constancy**. Quite accurate colour constancy algorithms are now available; simple versions show up in the "auto white balance" function of your camera. Note that if we wanted to build a camera for mantis shrimp, we would need 12 different pixel colours, corresponding to the 12 types of colour receptors of the crustacean.

EARLY IMAGE-PROCESSING OPERATIONS

We have seen how light reflects off objects in the scene to form an image consisting of, say, five million 3-byte pixels. With all sensors there will be noise in the image, and in any case there is a lot of data to deal with. So how do we get started on analysing this data?

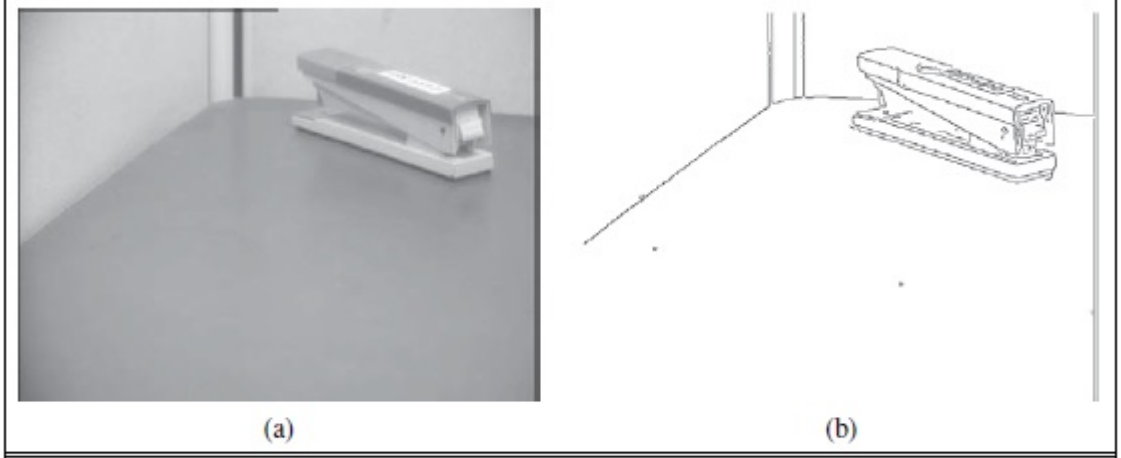
we will study three useful image-processing operations: edge detection, texture analysis, and computation of optical flow. These are called "early" or "low-level" operations because they are the first in a pipeline of operations. Early vision operations are characterized by their local nature (they can be carried out in one part of the image without regard for anything more than a few pixels away) and by their lack of knowledge: we can perform these operations without consideration of the objects that might be present in the scene. This makes the low-level operations good candidates for implementation in parallel hardware—either in a graphics processor unit (GPU) or an eye. We will then look at one mid-level operation: segmenting the image into regions.



Different kinds of edges: (1) depth discontinuities; (2) surface orientation discontinuities; (3) reflectance discontinuities; (4) illumination discontinuities (shadows).

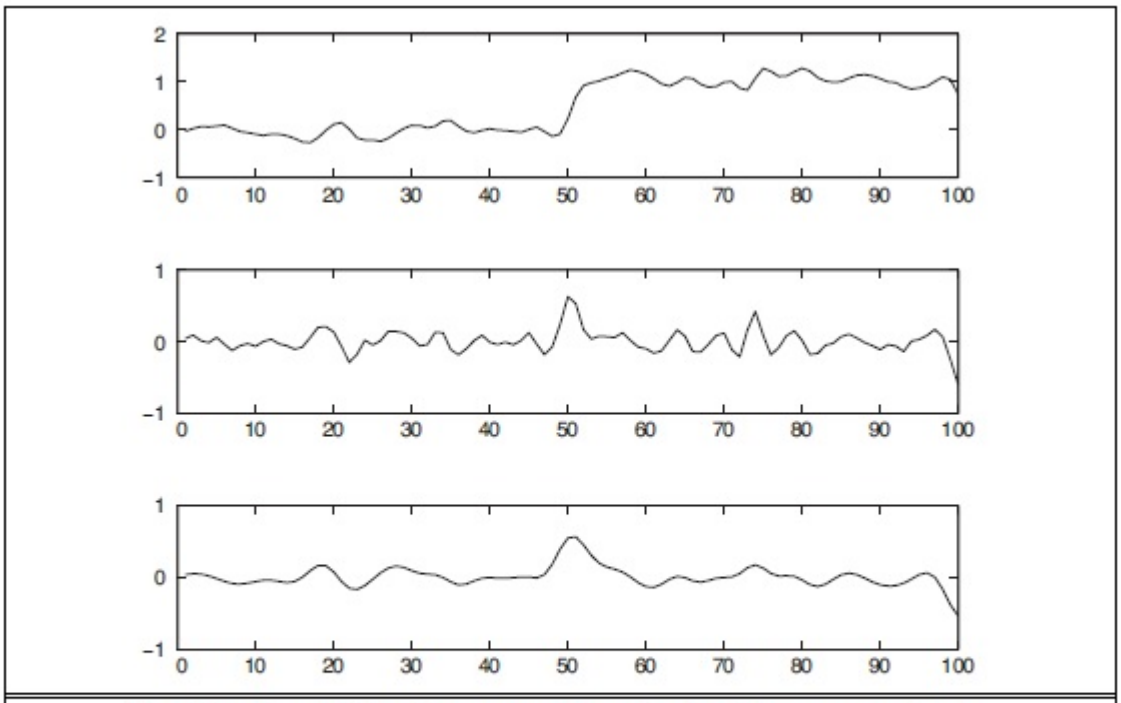
Edge detection

Edges are straight lines or curves EDGE in the image plane across which there is a “significant” change in image brightness. The goal of edge detection is to abstract away from the messy, multimega byte image and toward a more compact, abstract representation, the motivation is that edge contours in the image correspond to important scene contours. In the figure we have three examples of depth discontinuity, labelled 1; two surface-normal discontinuities, labelled 2; a reflectance discontinuity, labelled 3; and an illumination discontinuity (shadow), labelled 4. Edge detection is concerned only with the image, and thus does not distinguish between these different types of scene discontinuities; later processing will.



(a) Photograph of a stapler. (b) Edges computed from (a).

(a) shows an image of a scene containing a stapler resting on a desk, and
(b) shows the output of an edge-detection algorithm on this image. As you can see, there is a difference between the output and an ideal line drawing. There are gaps where no edge appears, and there are “noise” edges that do not correspond to anything of significance in the scene. Later stages of processing will have to correct for these errors.



Top: Intensity profile $I(x)$ along a one-dimensional section across an edge at $x = 50$. Middle: The derivative of intensity, $I'(x)$. Large values of this function correspond to edges, but the function is noisy. Bottom: The derivative of a smoothed version of the intensity, $(I * G_\sigma)'$, which can be computed in one step as the convolution $I * G'_\sigma$. The noisy candidate edge at $x = 75$ has disappeared.

How do we detect edges in an image? Consider the profile of image brightness along a one-dimensional cross-section perpendicular to an edge—for example, the one between the left edge of the desk and the wall. Edges correspond to locations in images where the brightness undergoes a sharp change, so a naive idea would be to differentiate the image and look for places where the magnitude of the derivative I'(x) is large.

One good answer is a weighted average that weights the nearest pixels the most, then gradually decreases the weight for more distant pixels. The **Gaussian filter** does just that. (Users of Photoshop recognize this as the *Gaussian blur* operation.) Recall that the Gaussian function with standard deviation σ and mean 0 is

$$N_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-x^2/2\sigma^2} \quad \text{in one dimension, or}$$
$$N_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \quad \text{in two dimensions.}$$

The application of the Gaussian filter replaces the intensity I(x0, y0) with the sum, overall (x, y) pixels, of I(x, y)Nσ(d), where d is the distance from (x0, y0) to (x, y). This kind of weighted sum is so common that there is a special name and notation for it. We say that the function h is the **convolution** of two functions f and g (denoted f ⍰ g) if we have

$$h(x) = (f * g)(x) = \sum_{u=-\infty}^{+\infty} f(u)\, g(x-u) \qquad \text{in one dimension, or}$$

$$h(x,y) = (f * g)(x,y) = \sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} f(u,v)\, g(x-u, y-v) \qquad \text{in two.}$$

There is a natural generalization of this algorithm from one-dimensional cross sections to general two-dimensional images. In two dimensions edges may be at any angle θ. Considering the image brightness as a scalar function of the variables x, y, its gradient is a vector

$$\nabla I = \begin{pmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{pmatrix} = \begin{pmatrix} I_x \\ I_y \end{pmatrix}$$

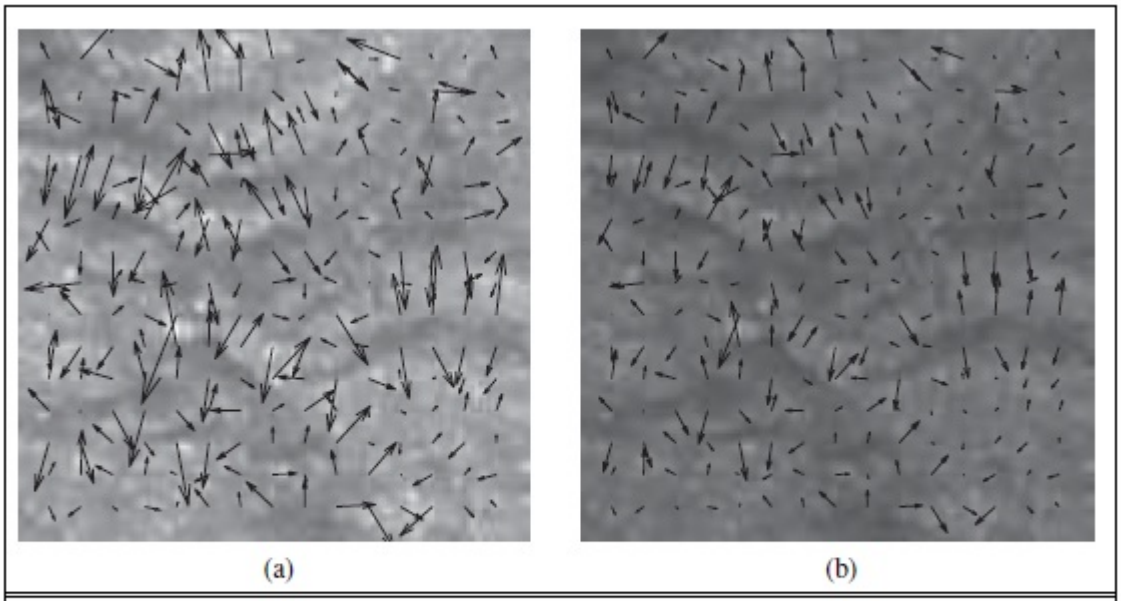
This gives us a θ = θ(x, y) at every pixel, which defines the edge **orientation** at that pixel.

$$\frac{\nabla I}{\|\nabla I\|} = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$$

Once we have marked edge pixels by this algorithm, the next stage is to link those pixels that belong to the same edge curves. This can be done by assuming that any two neighbouring edge pixels with consistent orientations must belong to the same edge curve.

Texture

In everyday language, **texture** is the visual TEXTURE feel of a surface—what you see evokes what the surface might feel like if you touched it (“texture” has the same root as “textile”). In computational vision, texture refers to a spatially repeating pattern on a surface that can be sensed visually. Examples include the pattern of windows on a building, stitches on a sweater, spots on a leopard, blades of grass on a lawn, pebbles on a beach, and people in a stadium. Sometimes the arrangement is quite periodic, as in the stitches on a sweater; in other cases, such as pebbles on a beach, the regularity is only statistical.



Two images of the same texture of crumpled rice paper, with different illumination levels. The gradient vector field (at every eighth pixel) is plotted on top of each one. Notice that, as the light gets darker, all the gradient vectors get shorter. The vectors do not rotate, so the gradient orientations do not change.

In images of textured objects, edge detection does not work as well as it does for smooth objects. This is because the most important edges can be lost among the texture elements. Quite literally, we may miss the tiger for the stripes. The solution is to look for differences in texture properties, just the way we look for differences in brightness. A patch on a tiger and a patch on the grassy background will have very different orientation histograms, allowing us to find the boundary curve between them.

Optical flow

Next, let us consider what happens when we have a video sequence, instead of just a single static image. When an object in the video is moving, or when the camera is moving relative to an object, the resulting apparent motion in the image is called **optical flow**. Optical flow describes the direction and speed of motion of features in the image—the optical flow of a video of a race car would be measured in pixels per second, not miles per hour. The optical flow encodes useful information about scene structure. For example, in a video of scenery taken from a moving train, distant objects have slower apparent motion than close objects; thus, the rate of apparent motion can tell us something about distance. Optical flow also enables us to recognize actions.

This block of pixels is to be compared with pixel blocks cantered at various candidate pixels at ($x_0 + D_x$, $y_0 + D_y$) at time $t_0 + D_t$. One possible measure of similarity is the **sum of squared differences** (SSD):

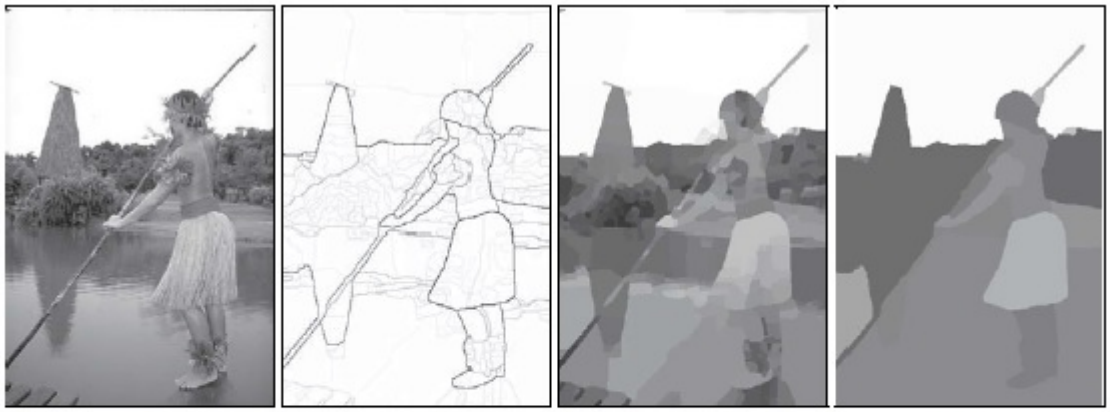
$$\mathrm{SSD}(D_x, D_y) = \sum_{(x,y)} (I(x, y, t) - I(x + D_x, y + D_y, t + D_t))^2 .$$

Two frames of a video sequence. On the right is the optical flow field corresponding to the displacement from one frame to the other. Note how the movement of the tennis racket and the front leg is captured by the directions of the arrows. (Courtesy of Thomas Brox.)

Segmentation of images

Segmentation is the process of breaking an image into **regions** of similar pixels. Each image pixel can be associated with certain visual properties, such as brightness, colour, and texture. Within an object, or a single part of an object, these attributes vary relatively little, whereas across an inter-object boundary there is typically a large change in one or more of these attributes.

Boundaries detected by this technique turn out to be significantly better than those found using the simple edge-detection technique described previously. But still there are two limitations. (1) The boundary pixels formed by thresholding Pb(x, y, θ) are not guaranteed to form closed curves, so this approach doesn't deliver regions, and (2) the decision making exploits only local context and does not use global consistency constraints.



(a) (b) (c) (d)

(a) Original image. (b) Boundary contours, where the higher the P_b value, the darker the contour. (c) Segmentation into regions, corresponding to a fine partition of the image. Regions are rendered in their mean colors. (d) Segmentation into regions, corresponding to a coarser partition of the image, resulting in fewer regions. (Courtesy of Pablo Arbelaez, Michael Maire, Charles Fowlkes, and Jitendra Malik)

Segmentation based purely on low-level, local attributes such as brightness and colour cannot be expected to deliver the final correct boundaries of all the objects in the scene. To reliably find object boundaries we need high-level knowledge of the likely kinds of objects in the scene. Representing this knowledge is a topic of active research. A popular strategy is to produce an over-segmentation of an image, containing hundreds of homogeneous regions known as **super pixels**. From there, knowledge-based algorithms can take over; they will find it easier to deal with hundreds of super pixels rather than millions of raw pixels. How to exploit high-level knowledge of objects is the subject of the next section.

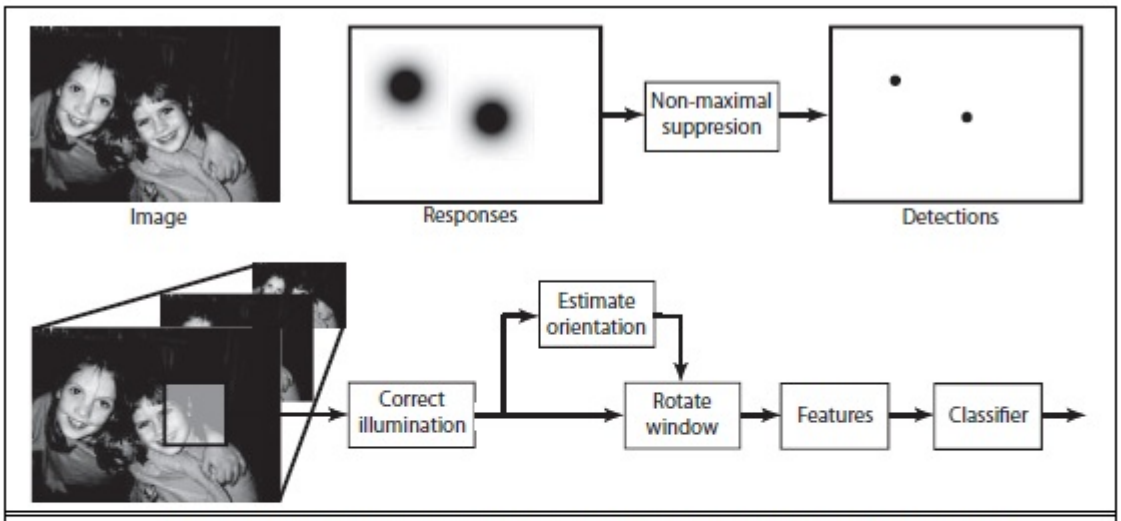
OBJECT RECOGNITION BY APPEARANCE

Appearance is shorthand for what an object tends to look like. Some object categories—for example, baseballs—vary rather little in appearance; all of the objects in the category look about the same under most circumstances. In this case, we can compute a set of features describing each class of images likely to contain the object, then test it with a classifier.

Testing each class of images with a learned classifier is an important general recipe. It works extremely well for faces looking directly at the camera, because at low resolution and under reasonable lighting, all such faces look quite similar. The face is round, and quite bright compared to the eye sockets; these are dark, because they are sunken, and the mouth is a dark slash, as are the eyebrows. Major changes of illumination can cause some variations in this pattern, but the range of variation is quite manageable. That makes it possible to detect face positions in an image that contains faces. Once a computational challenge, this feature is now commonplace in even inexpensive digital cameras.

For the moment, we will consider only faces where the nose is oriented vertically; we will deal with rotated faces below. We sweep a round window of fixed size over the image, compute features for it, and present the features to a classifier. This strategy is sometimes called the **sliding window**. Features need to be robust to shadows and to changes in brightness caused by illumination changes. One strategy is to build features out of gradient orientations. Another is to estimate and correct the illumination in each image window. To find faces of different sizes, repeat the sweep over larger or smaller versions of the image. Finally, we postprocess the responses across scales and locations to produce the final set of detections.

Training data is quite easily obtained. There are several data sets of marked-up face images, and rotated face windows are easy to build (just rotate a window from a training data set). One trick that is widely used is to take each example window, then produce new examples by changing the orientation of the window, the center of the window, or the scale very slightly. This is an easy way of getting a bigger data set that reflects real images fairly well; the trick usually improves performance significantly.

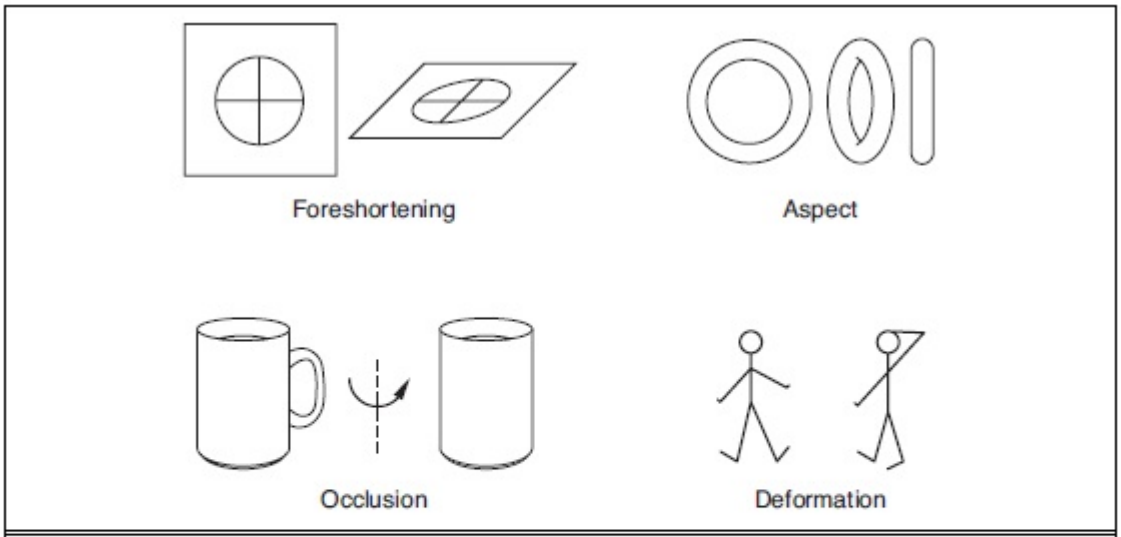


Face finding systems vary, but most follow the architecture illustrated in two parts here. On the top, we go from images to responses, then apply non-maximum suppression to find the strongest local response. The responses are obtained by the process illustrated on the bottom. We sweep a window of fixed size over larger and smaller versions of the image, so as to find smaller or larger faces, respectively. The illumination in the window is corrected, and then a regression engine (quite often, a neural net) predicts the orientation of the face. The window is corrected to this orientation and then presented to a classifier. Classifier outputs are then postprocessed to ensure that only one face is placed at each location in the image.

Complex appearance and pattern elements

Many objects produce much more complex patterns than faces do. This is because several effects can move features around in an image of the object.

- **Foreshortening**, which causes a pattern viewed at a slant to be significantly distorted.
- **Aspect**, which causes objects to look different when seen from different directions. Even as simple an object as a doughnut has several aspects; seen from the side, it looks like a flattened oval, but from above it is an annulus.
- **Occlusion**, where some parts are hidden from some viewing directions. Objects can occlude one another, or parts of an object can occlude other parts, an effect known as self-occlusion.
- **Deformation**, where internal degrees of freedom of the object change its appearance. For example, people can move their arms and legs around, generating a very wide range of different body configurations.

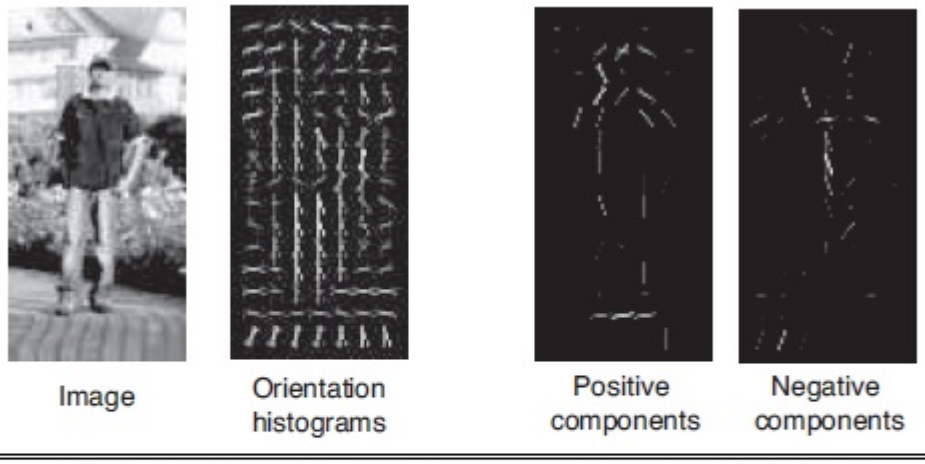


Sources of appearance variation. First, elements can foreshorten, like the circular patch on the top left. This patch is viewed at a slant, and so is elliptical in the image. Second, objects viewed from different directions can change shape quite dramatically, a phenomenon known as aspect. On the top right are three different aspects of a doughnut. Occlusion causes the handle of the mug on the bottom left to disappear when the mug is rotated. In this case, because the body and handle belong to the same mug, we have self-occlusion. Finally, on the bottom right, some objects can deform dramatically.

The most obvious approach is to represent the image window with a histogram of the pattern elements that appear there. This approach does not work particularly well, because too many patterns get confused with one another. For example, if the pattern elements are color pixels, the French, UK, and Netherlands flags will get confused because they have approximately the same color histograms, though the colors are arranged in very different ways. Quite simple modifications of histograms yield very useful features. The trick is to preserve some spatial detail in the representation; for example, headlights tend to be at the front of a car and wheels tend to be at the bottom. Histogram-based features have been successful in a wide variety of recognition applications; we will survey pedestrian detection.

Pedestrian detection with HOG features

The World Bank estimates that each year car accidents kill about 1.2 million people, of whom about two thirds are pedestrians. This means that detecting pedestrians is an important application problem, because cars that can automatically detect and avoid pedestrians might save many lives. Pedestrians wear many different kinds of clothing and appear in many different configurations, but, at relatively low resolution, pedestrians can have a fairly characteristic appearance. The most usual cases are lateral or frontal views of a walk. In these cases, we see either a “lollipop” shape — the torso is wider than the legs, which are together in the stance phase of the walk — or a “scissor” shape — where the legs are swinging in the walk. We expect to see some evidence of arms and legs, and the curve around the shoulders and head also tends to visible and quite distinctive. This means that, with a careful feature construction, we can build a useful moving-window pedestrian detector.



Local orientation histograms are a powerful feature for recognizing even quite complex objects. On the left, an image of a pedestrian. On the center left, local orientation histograms for patches. We then apply a classifier such as a support vector machine to find the weights for each histogram that best separate the positive examples of pedestrians from non-pedestrians. We see that the positively weighted components look like the outline of a person. The negative components are less clear; they represent all the patterns that are not pedestrians. Figure from Dalal and Triggs (2005) © IEEE.

One further trick is required to make a good feature. Because orientation features are not affected by illumination brightness, we cannot treat high-contrast edges specially. This means that the distinctive curves on the boundary of a pedestrian are treated in the same way as fine texture detail in clothing or in the background, and so the signal may be submerged in noise. We can recover contrast information by counting gradient orientations with weights that reflect how significant a gradient is compared to other gradients in the same cell.

Another example of object recognition, this one using the SIFT feature (Scale Invariant Feature Transform), an earlier version of the HOG feature. On the **left**, images of a shoe and a telephone that serve as object models. In the **center**, a test image. On the **right**, the shoe and the telephone have been detected by: finding points in the image whose SIFT feature descriptions match a model; computing an estimate of pose of the model; and verifying that estimate. A strong match is usually verified with rare false positives. Images from Lowe (1999) © IEEE.

orientation at **x** for this cell. A natural choice of weight is

$$w_{\mathbf{x},\mathcal{C}} = \frac{\|\nabla I_{\mathbf{x}}\|}{\sum_{\mathbf{u}\in\mathcal{C}} \|\nabla I_{\mathbf{u}}\|} .$$

This compares the gradient magnitude to others in the cell, so gradients that are large compared to their neighbours get a large weight. The resulting feature is usually called a **HOG feature**

This feature construction is the main way in which pedestrian detection differs from face detection. Otherwise, building a pedestrian detector is very like building a face detector. The detector sweeps a window across the image, computes features for that window, then presents it to a classifier. Non-maximum suppression needs to be applied to the output. In most applications, the scale and orientation of typical pedestrians is known. For example, in driving applications in which a camera is fixed to the car, we expect to view mainly vertical pedestrians, and we are interested only in nearby pedestrians. Several pedestrian data sets have been published, and these can be used for training the classifier.

RECONSTRUCTING THE 3D WORLD

In this section we show how to go from the two-dimensional image to a three-dimensional representation of the scene. The fundamental question is this: Given that all points in the scene that fall along a ray to the pinhole are projected to the same point in the image, how do we recover three-dimensional information? Two ideas come to our rescue

- If we have two (or more) images from different camera positions, then we can triangulate to find the position of a point in the scene.
- We can exploit background knowledge about the physical scene that gave rise to the image. Given an object model **P**(Scene) and a rendering model **P**(Image | Scene), we can compute a posterior distribution **P**(Scene | Image).

There is as yet no single unified theory for scene reconstruction. We survey eight commonly used visual cues: **motion**, **binocular stereopsis**, **multiple views**, **texture**, **shading**, **contour**, and **familiar objects**.

Motion parallax

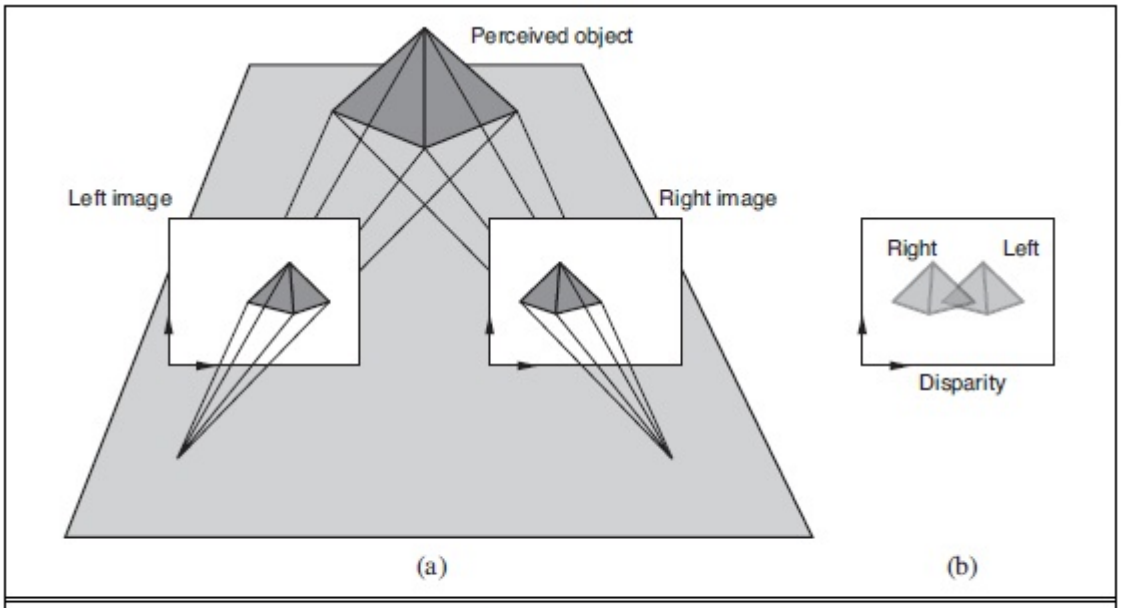
If the camera moves relative to the three-dimensional scene, the resulting apparent motion in the image, optical flow, can be a source of information for both the movement of the camera and depth in the scene. To understand this, we state (without proof) an equation that relates the optical flow to the viewer's translational velocity **T** and the depth in the scene. The components of the optical flow field are

$$v_x(x,y) = \frac{-T_x + xT_z}{Z(x,y)}, \qquad v_y(x,y) = \frac{-T_y + yT_z}{Z(x,y)} ,$$

where Z(x, y) is the z-coordinate of the point in the scene corresponding to the point in the image at (x, y).

Note that both components of the optical flow, $v_x(x, y)$ and $v_y(x, y)$, are zero at the point $x = T_x/T_z, y = T_y/T_z$. This point is called the **focus of expansion** of the flow field. Suppose we change the origin in the x–y plane to lie at the focus of expansion; then the expressions for optical flow take on a particularly simple form. Let (x', y') be the new coordinates defined by $x' = x - T_x/T_z, y' = y - T_y/T_z$. Then

$$v_x(x', y') = \frac{x'T_z}{Z(x', y')}, \qquad v_y(x', y') = \frac{y'T_z}{Z(x', y')}.$$



Translating a camera parallel to the image plane causes image features to move in the camera plane. The disparity in positions that results is a cue to depth. If we superimpose left and right image, as in (b), we see the disparity.

OBJECT RECOGNITION FROM STRUCTURAL INFORMATION

Putting a box around pedestrians in an image may well be enough to avoid driving into them. We have seen that we can find a box by pooling the evidence provided by orientations, using histogram methods to suppress potentially confusing spatial detail. If we want to know more about what someone is doing, we will need to know where their arms, legs, body, and head lie in the picture. Individual body parts are quite difficult to detect on their own using a moving window method, because their color and texture can vary widely and because they are usually small in images. Often, forearms and shins are as small as two to three pixels wide. Body parts do not usually appear on their own, and representing what is connected to what could be quite powerful, because parts that are easy to find might tell us where to look for parts that are small and hard to detect.

Inferring the layout of human bodies in pictures is an important task in vision, because the layout of the body often reveals what people are doing. A model called a **deformable template** can tell us which configurations are acceptable: the elbow can bend but the head is never joined to the foot. The simplest deformable template model of a person connects lower arms to upper arms, upper arms to the torso, and so on.

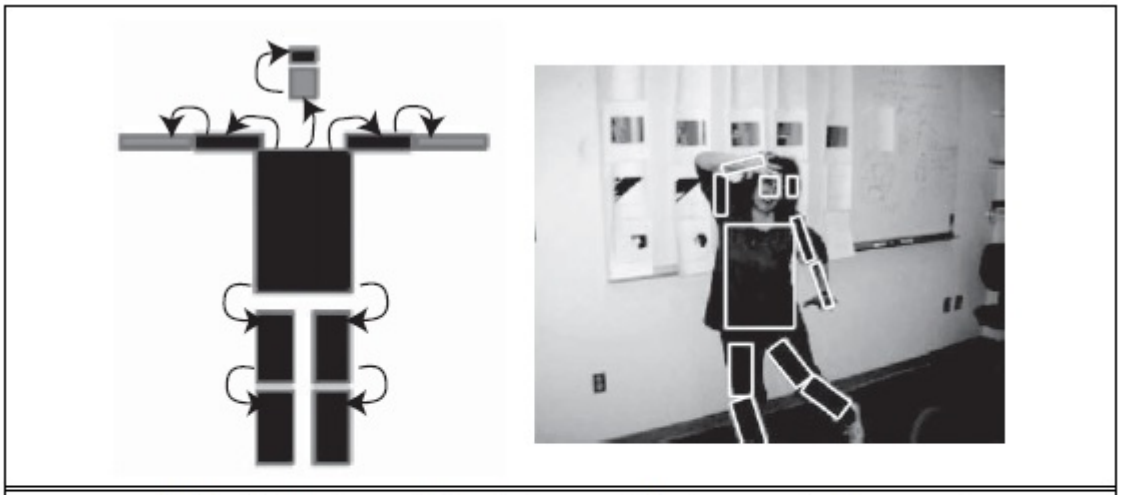
we could represent the fact that left and right upper arms tend to have the same color and texture, as do left and right legs.

The geometry of bodies: Finding arms and legs

For the moment, we assume that we know what the person's body parts look like (e.g., we know the color and texture of the person's clothing). We can model the geometry of the body as a tree of eleven segments (upper and lower left and right arms and legs respectively, a torso, a face, and hair on top of the face) each of which is rectangular. We assume that the position and orientation (**pose**) of POSE the left lower arm is independent of all other segments given the pose of the left upper arm; that the pose of the left upper arm is independent of all segments given the pose of the torso; and extend these assumptions in the obvious way to include the right arm and the legs, the face, and the hair. Such models are often called "cardboard people" models.

There are two criteria for evaluating a configuration. First, an image rectangle should look like its segment. For the moment, we will remain vague about precisely what that means, but we assume we have a function φi that scores how well an image rectangle matches a body segment. For each pair of related segments, we have another function ψ that scores how well relations between a pair of image rectangles match those to be expected from the body segments. The dependencies between segments form a tree, so each segment has only one parent, and we could write ψi,pa(i).

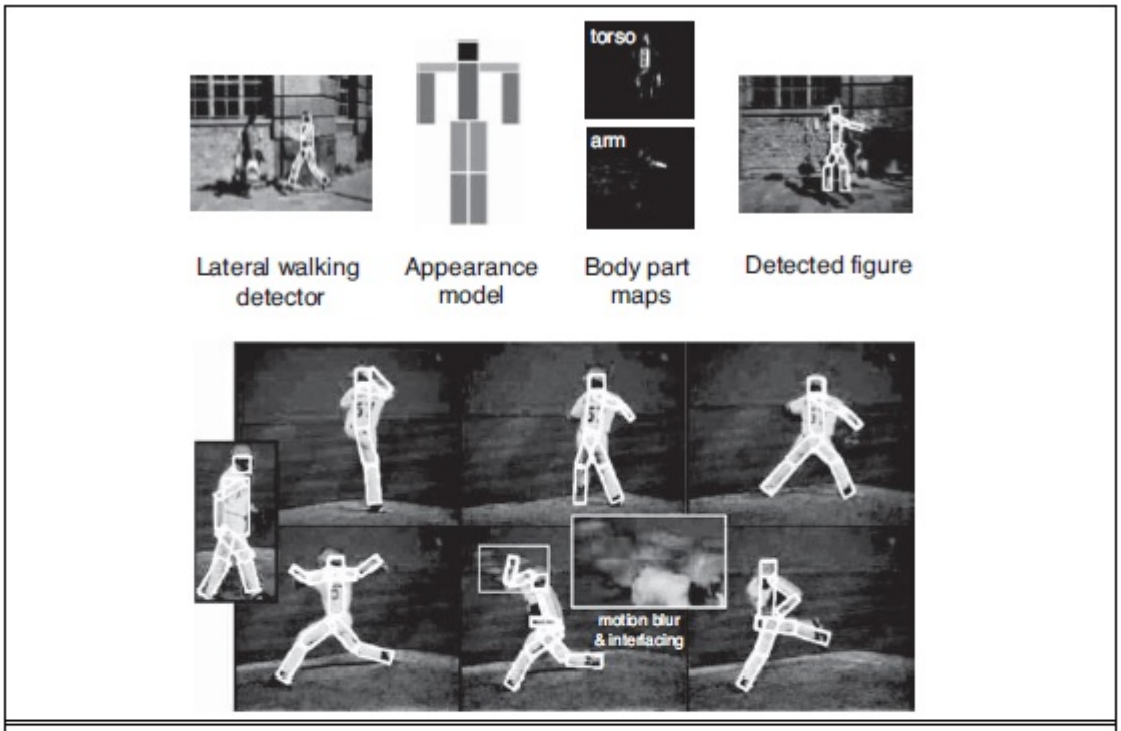
$$\sum_{i \in \text{segments}} \phi_i(m_i) + \sum_{i \in \text{segments}} \psi_{i,\text{pa}(i)}(m_i, m_{\text{pa}(i)}) .$$



A pictorial structure model evaluates a match between a set of image rectangles and a cardboard person (shown on the left) by scoring the similarity in appearance between body segments and image segments and the spatial relations between the image segments. Generally, a match is better if the image segments have about the right appearance and are in about the right place with respect to one another. The appearance model uses average colors for hair, head, torso, and upper and lower arms and legs. The relevant relations are shown as arrows. On the right, the best match for a particular image, obtained using dynamic programming. The match is a fair estimate of the configuration of the body. Figure from Felzenszwalb and Huttenlocher (2000) © IEEE.

Coherent appearance: Tracking people in video

Tracking people in video is an important practical problem. If we could reliably report the location of arms, legs, torso, and head in video sequences, we could build much improved game interfaces and surveillance systems. Filtering methods have not had much success with this problem, because people can produce large accelerations and move quite fast. This means that for 30 Hz video, the configuration of the body in frame i doesn't constrain the configuration of the body in frame i+1 all that strongly. Currently, the most effective methods exploit the fact that appearance changes very slowly from frame to frame. If we can infer an appearance model of an individual from the video, then we can use this information in a pictorial structure model to detect that person in each frame of the video. We can then link these locations across time to make a track.



We can track moving people with a pictorial structure model by first obtaining an appearance model, then applying it. To obtain the appearance model, we scan the image to find a lateral walking pose. The detector does not need to be very accurate, but should produce few false positives. From the detector response, we can read off pixels that lie on each body segment, and others that do not lie on that segment. This makes it possible to build a discriminative model of the appearance of each body part, and these are tied together into a pictorial structure model of the person being tracked. Finally, we can reliably track by detecting this model in each frame. As the frames in the lower part of the image suggest, this procedure can track complicated, fast-changing body configurations, despite degradation of the video signal due to motion blur. Figure from Ramanan *et al.* (2007) © IEEE.

USING VISION

Some problems are well understood. If people are relatively small in the video frame, and the background is stable, it is easy to detect the people by subtracting a background image from the current frame. If the absolute value of the difference is large, this **background subtraction** declares the pixel to be a foreground pixel; by linking foreground blobs over time, we obtain a track.

Words and pictures

Many Web sites offer collections of images for viewing. How can we find the images we want? Let's suppose the user enters a text query, such as "bicycle race." Some of the images will have keywords or captions attached, or will come from Web pages that contain text near the image.

In the most straightforward version of this task, we have a set of correctly tagged example images, and we wish to tag some test images. This problem is sometimes known as auto-annotation. The most accurate solutions are obtained using nearest-neighbours' methods. One finds the training images

that are closest to the test image in a feature space metric that is trained using examples, then reports their tags.

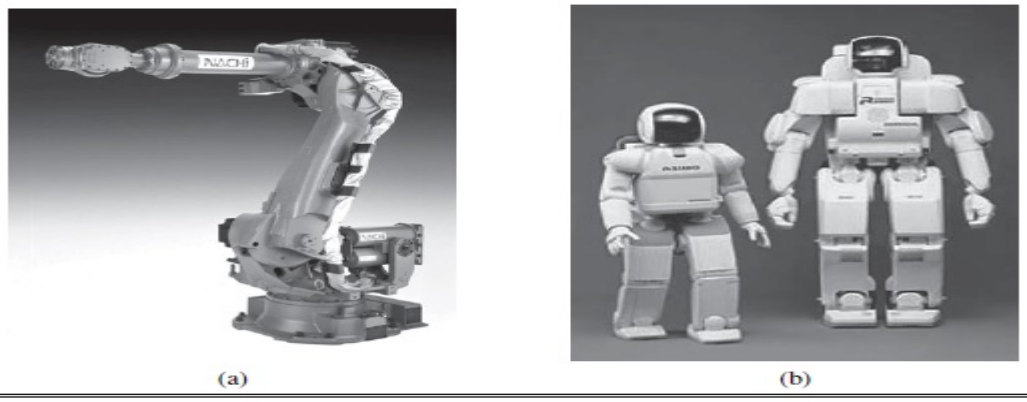
UNIT-V

Robotics: Introduction, Robot Hardware, Robotic Perception, planning to move, planning uncertain movements, Moving, Robotic software architectures, application domains

Philosophical foundations: Weak AI, Strong AI, Ethics and Risks of AI, Agent Components, Agent Architectures, are we going in the right direction, what if AI does succeed.

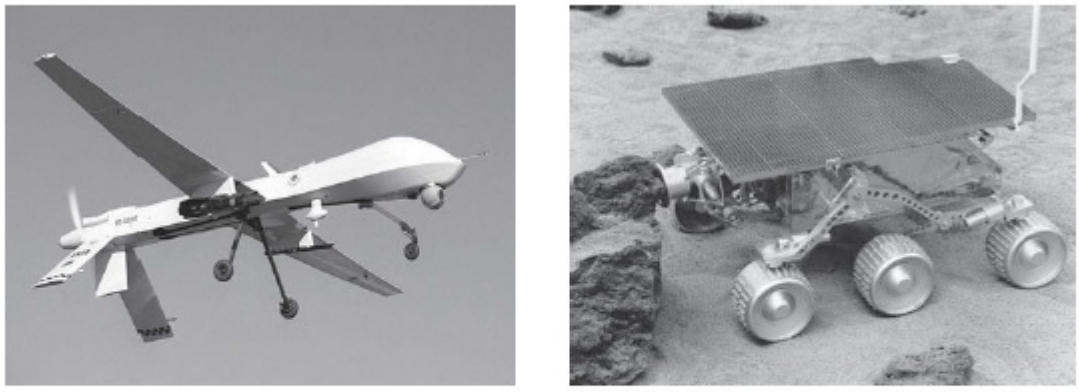
Introduction

- **Robots** are physical agents that perform tasks by manipulating the physical world. To do so they are equipped with **effectors** such as legs, wheels, joints, and grippers. Effectors have a single purpose: to assert physical forces on the environment. Robots are also equipped with **sensors**, which allow them to perceive their environment.
- Present day robotics employs a diverse set of sensors, including cameras and lasers to measure the environment, and gyroscopes and accelerometers to measure the robot's own motion.
- Most of today's robots fall into one of three primary categories. **Manipulators**, or robot arms, are physically anchored to their workplace, for example in a factory assembly line or on the International Space Station. Manipulator motion usually involves a chain of controllable joints, enabling such robots to place their effectors in any position within the workplace. Manipulators are by far the most common type of industrial robots, with approximately one million units installed worldwide. Some mobile manipulators are used in hospitals to assist surgeons.
- The second category is the **mobile robot**. Mobile robots move about their environment using wheels, legs, or similar mechanisms.
- They have been put to use delivering food in hospitals, moving containers at loading docks, and similar tasks. **Unmanned ground vehicles**, or UGVs, drive autonomously on streets, highways, and off-road.
- The **planetary rover** shown in Figure 25.2(b) explored Mars for a period of 3 months in 1997.
- Other types of mobile robots include **unmanned air vehicles** (UAVs), commonly used for surveillance, crop-spraying,



(a) (b)

Figure 25.1 (a) An industrial robotic manipulator for stacking bags on a pallet. Image courtesy of Nachi Robotic Systems. (b) Honda's P3 and Asimo humanoid robots.



(a) (b)

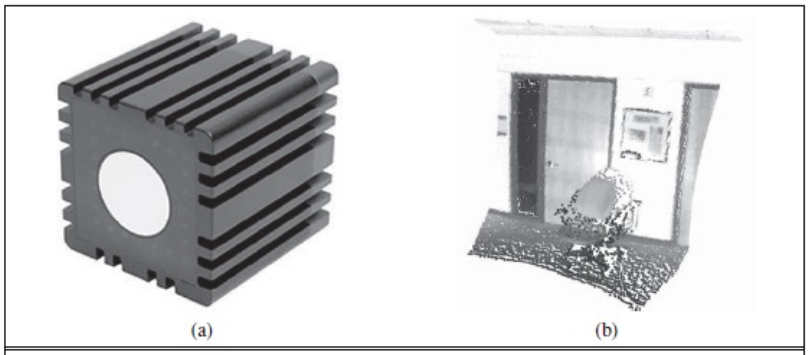
(a) Predator, an unmanned aerial vehicle (UAV) used by the U.S. Military. Image courtesy of General Atomics Aeronautical Systems. (b) NASA's Sojourner, a mobile robot that explored the surface of Mars in July 1997.

- UAV commonly used by the U.S. military. **Autonomous underwater vehicles** (AUVs) are used in deep sea exploration. Mobile robots deliver packages in the workplace and vacuum the floors at home.
- The third type of robot combines mobility with manipulation, and is often called a **mobile manipulator**. **Humanoid robots** mimic the human torso. shows two early humanoid robots, both manufactured by Honda Corp. in Japan. Mobile manipulators can apply their effectors further afield than anchored manipulators can, but their task is made harder because they don't have the rigidity that the anchor provides.

ROBOT HARDWARE

- **Sensors:** Sensors are the perceptual interface between robot and environment

- **Passive sensors**, such as cameras, are true observers of the environment: they capture signals that are generated by other sources in the environment.
- **Active sensors**, such as sonar, send energy into the environment. They rely on the fact that this energy is reflected back to the sensor. Active sensors tend to provide more information than passive sensors, but at the expense of increased power consumption and with a danger of interference when multiple active sensors are used at the same time. Whether active or passive, sensors can be divided into three types, depending on whether they sense the environment, the robot's location, or the robot's internal configuration.
- **Range finders** are sensors that measure the distance to nearby objects. In the early days of robotics, robots were commonly equipped with **sonar sensors**. Sonar sensors emit directional sound waves, which are reflected by objects, with some of the sound making it back into the sensor. The time and intensity of the returning signal indicates the distance to nearby objects. Sonar is the technology of choice for autonomous underwater vehicles



(a) Time of flight camera; image courtesy of Mesa Imaging GmbH. (b) 3D range image obtained with this camera. The range image makes it possible to detect obstacles and objects in a robot's vicinity.

- **Stereo vision** relies STEREO VISION on multiple cameras to image the environment from slightly different viewpoints, analysing the resulting parallax in these images to compute the range of surrounding objects. For mobile ground robots, sonar and stereo vision are now rarely used, because they are not reliably accurate.
- Most ground robots are now equipped with optical range finders. Just like sonar sensors, optical range sensors emit active signals (light) and measure the time until a reflection of this signal arrives back at the sensor. shows a **time of flight camera**. This camera acquires range images like the one at up to 60 frames per second.

- Other range sensors use laser beams and special 1-pixel cameras that can be directed using complex arrangements of mirrors or rotating elements. These sensors are called **scanning lidars** (short for *light detection and ranging*). Scanning lidars tend to provide longer ranges than time of flight cameras, and tend to perform better in bright daylight.
- Other common range sensors include radar, which is often the sensor of choice for UAVs. Radar sensors can measure distances of multiple kilometres. On the other extreme end of range sensing are **tactile sensors** such as whiskers, bump panels, and touch-sensitive skin. These sensors measure range based on physical contact, and can be deployed only for sensing objects very close to the robot.
- A second important class of sensors is **location sensors**. Most location sensors use range sensing as a primary component to determine location. Outdoors, the **Global Positioning System** (GPS) is the most common solution to the localization problem. GPS measures the distance to satellites that emit pulsed signals. At present, there are 31 satellites in orbit, transmitting signals on multiple frequencies. GPS receivers can recover the distance to these satellites by analysing phase shifts. By triangulating signals from multiple satellites, GPS receivers can determine their absolute location on Earth to within a few meters.
- **Differential GPS** involves a second ground receiver with known location, providing millimetre accuracy under ideal conditions. Unfortunately, GPS does not work indoors or underwater. Indoors, localization is often achieved by attaching beacons in the environment at known locations. Many indoor environments are full of wireless base stations, which can help robots localize through the analysis of the wireless signal.
- The third important class is **proprioceptive sensors**, which inform the robot of its own motion. To measure the exact configuration of a robotic joint, motors are often equipped with **shaft decoders** that count the revolution of motors in small increments.
- On robot arms, shaft decoders can provide accurate information over any period of time. On mobile robots, shaft decoders that report wheel revolutions can be used for **odometry**—the measurement of distance travelled.
- **Inertial sensors**, such as gyroscopes, rely on the resistance of mass to the change of velocity. They can help reduce uncertainty.
- Other important aspects of robot state are measured by **force sensors** and **torque sensors**. These are indispensable when robots handle fragile objects or objects whose exact shape and location is unknown. Imagine a one-ton robotic manipulator screwing in a light bulb. It would be all too easy to

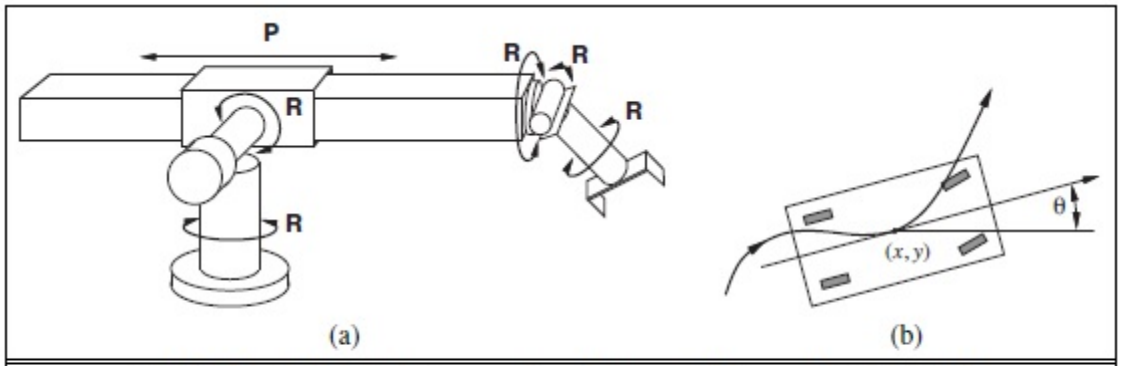
apply too much force and break the bulb. Force sensors allow the robot to sense how hard it is gripping the bulb, and torque sensors allow it to sense how hard it is turning. Good sensors can measure forces in all three translational and three rotational directions. They do this at a frequency of several hundred times a second, so that a robot can quickly detect unexpected forces and correct its actions before it breaks a light bulb.

25.2.2 EFFECTORS

Effectors are the means by which robots move and change the shape of their bodies. To understand the design of effectors, it will help to talk about motion and shape in the abstract, using the concept of a **degree of freedom** (DOF) We count one degree of freedom for each independent direction in which a robot, or one of its effectors, can move. For example, a rigid mobile robot such as an AUV has six degrees of freedom, three for its (x, y, z) location in space and three for its angular orientation, known as *yaw*, *roll*, and *pitch*. These six degrees define the **kinematic state**2 or **pose** of the robot. The **dynamic state** of a robot includes these six plus an additional six dimensions for the rate of change of each kinematic dimension, that is, their velocities.

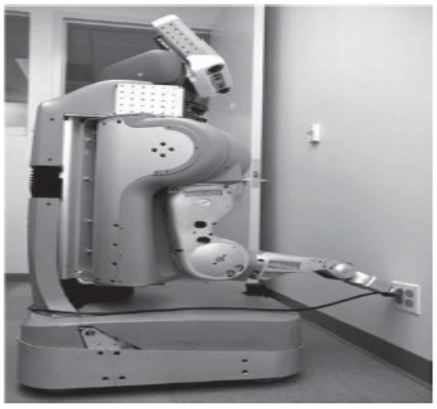
For nonrigid bodies, there are additional degrees of freedom within the robot itself. For example, the elbow of a human arm possesses two degree of freedom. It can flex the upper arm towards or away, and can rotate right or left. The wrist has three degrees of freedom. It can move up and down, side to side, and can also rotate.

created by five **revolute joints** that generate rotational motion and one **prismatic joint** that generates sliding motion. You can verify that the human arm as a whole has more than six degrees of freedom by a simple experiment: put your hand on the table and notice that you still have the freedom to rotate your elbow without changing the configuration of your hand. Manipulators that have extra degrees of freedom are easier to control than robots with only the minimum number of DOFs. Many industrial manipulators therefore have seven DOFs, not six.

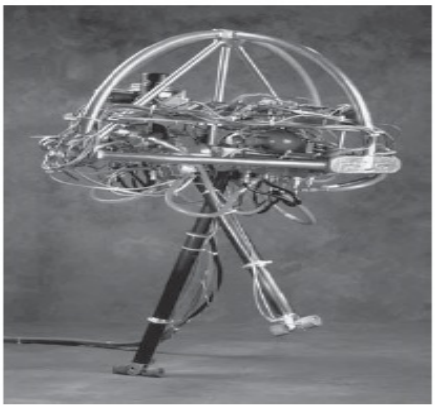


(a) The Stanford Manipulator, an early robot arm with five revolute joints (R) and one prismatic joint (P), for a total of six degrees of freedom. (b) Motion of a nonholonomic four-wheeled vehicle with front-wheel steering.

For mobile robots, the DOFs are not necessarily the same as the number of actuated elements. Consider, for example, your average car: it can move forward or backward, and it can turn, giving it two DOFs. In contrast, a car's kinematic configuration is three-dimensional:
on an open flat surface, one can easily maneuver a car to any (x, y) point, in any orientation. Thus, the car has three **effective degrees of freedom** but two **control label degrees of freedom**. We say a robot is **nonholonomic** if it has more effective DOFs than controllable DOFs and **holonomic** if the two numbers are the same. Holonomic robots are easier to control—it would be much easier to park a car that could move sideways as well as forward and backward—but holonomic robots are also mechanically more complex. Most robot arms are holonomic, and most mobile robots are nonholonomic. Mobile robots have a range of mechanisms for locomotion, including wheels, tracks, and legs. **Differential drive** robots possess two independently actuated wheels (or tracks), one on each side, as on a military tank. If both wheels move at the same velocity, the robot moves on a straight line. If they move in opposite directions, the robot turns on the spot. An alternative is the **synchro drive**, in which each wheel can move and turn around its own axis. To avoid chaos, the wheels are tightly coordinated. When moving straight, for example, all wheels point in the same direction and move at the same speed. Both differential and synchro drives are nonholonomic. Some more expensive robots use holonomic drives, which have three or more wheels that can be oriented and moved independently external forces.



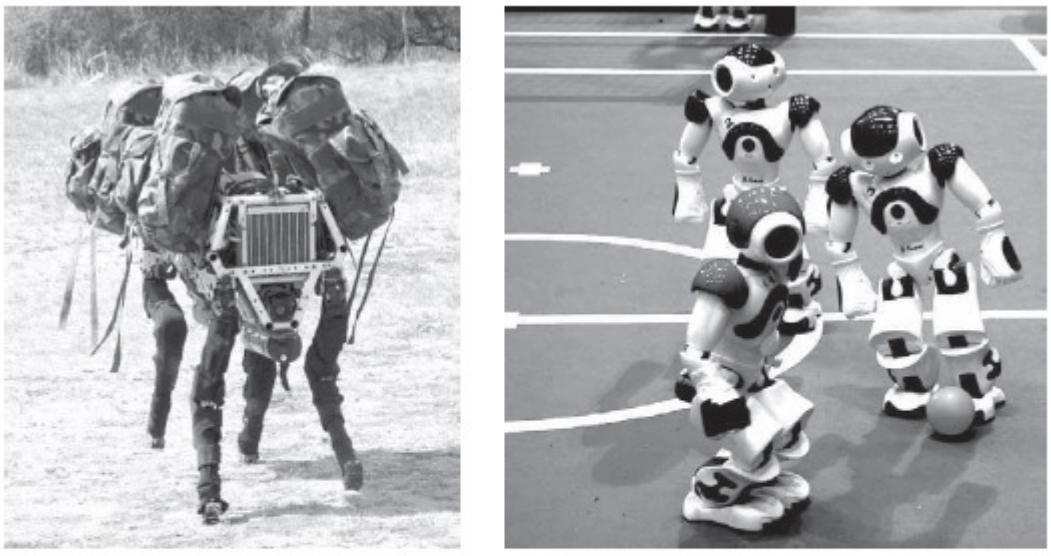
(a)



(b)

(a) Mobile manipulator plugging its charge cable into a wall outlet. Image courtesy of Willow Garage, © 2009. (b) One of Marc Raibert's legged robots in motion.

Legs, unlike wheels, can handle rough terrain. However, legs are notoriously slow on flat surfaces, and they are mechanically difficult to build. Robotics researchers have tried designs ranging from one leg up to opens of legs. Legged robots have been made to walk, run, and even hop—as we see with the legged robot. This robot is **dynamically stable**, meaning that it can remain upright while hopping around. A robot that can remain upright without moving its legs is called **statically stable**. A robot is statically stable if its center of gravity is above the polygon spanned by its legs. The quadruped (four-legged) robot may appear statically stable. However, it walks by lifting multiple legs at the same time, which renders it dynamically stable. The robot can walk on snow and ice, and it will not fall over even if you kick it (as demonstrated in videos available online). Two-legged robots are dynamically stable.



(a) (b)

(a) Four-legged dynamically-stable robot "Big Dog." Image courtesy Boston Dynamics, © 2009. (b) 2009 RoboCup Standard Platform League competition, showing the winning team, B-Human, from the DFKI center at the University of Bremen. Throughout the match, B-Human outscored their opponents 64:1. Their success was built on probabilistic state estimation using particle filters and Kalman filters; on machine-learning models for gait optimization; and on dynamic kicking moves. Image courtesy DFKI, © 2009.

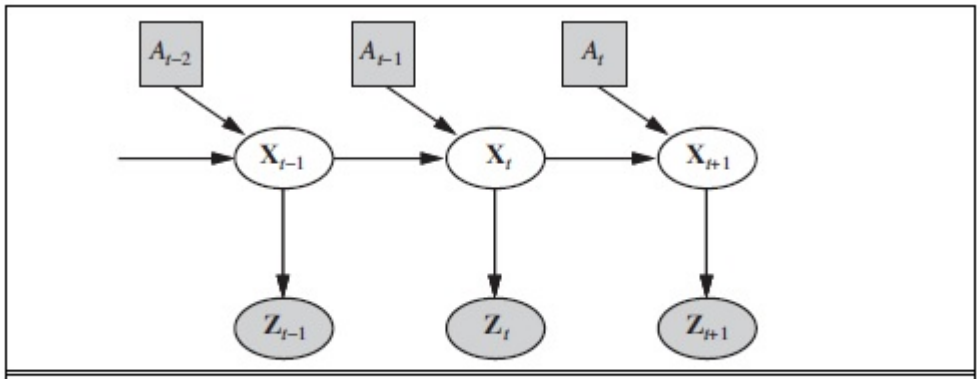
The **electric motor** is the most popular mechanism for both manipulator actuation and locomotion, but **pneumatic actuation** using compressed gas and **hydraulic actuation** using pressurized fluids also have their application niches.

ROBOTIC PERCEPTION

Perception is the process by which robots map sensor measurements into internal representations of the environment. Perception is difficult because sensors are noisy, and the environment is partially observable, unpredictable, and often dynamic. In other words, robots have all the problems of **state estimation** (or **filtering**). As a rule of thumb, good internal representations for robots have three properties: they contain enough information for the robot to make good decisions, they are structured so that they can be updated efficiently, and they are natural in the sense that internal variables correspond to natural state variables in the physical world.

we saw that Kalman filters, HMMs, and dynamic Bayes nets can represent the transition and sensor models of a partially observable environment, and we described both exact and approximate algorithms for updating the **belief state**—the posterior probability distribution over the environment state variables. Several dynamic Bayes net models. For robotics problems, we include the robot's own past actions as observed

variables in the model. the notation used in this chapter: **X**t is the state of the environment (including the robot) at time t, **Z**t is the observation received at time t, and At is the action taken after the observation is received.



Robot perception can be viewed as temporal inference from sequences of actions and measurements, as illustrated by this dynamic Bayes network.

We would like to compute the new belief state, **P**(**X**t+1 | **z**1:t+1, a1:t), from the current belief state **P**(**X**t | **z**1:t, a1:t−1) and the new observation **z**t+1. We did this in Section 15.2, but here there are two differences: we condition explicitly on the actions as well as the observations, and we deal with *continuous* rather than *discrete* variables.

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{z}_{1:t+1}, a_{1:t}) = \alpha \mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1}) \int \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, a_t)\, P(\mathbf{x}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})\, d\mathbf{x}_t \,.$$

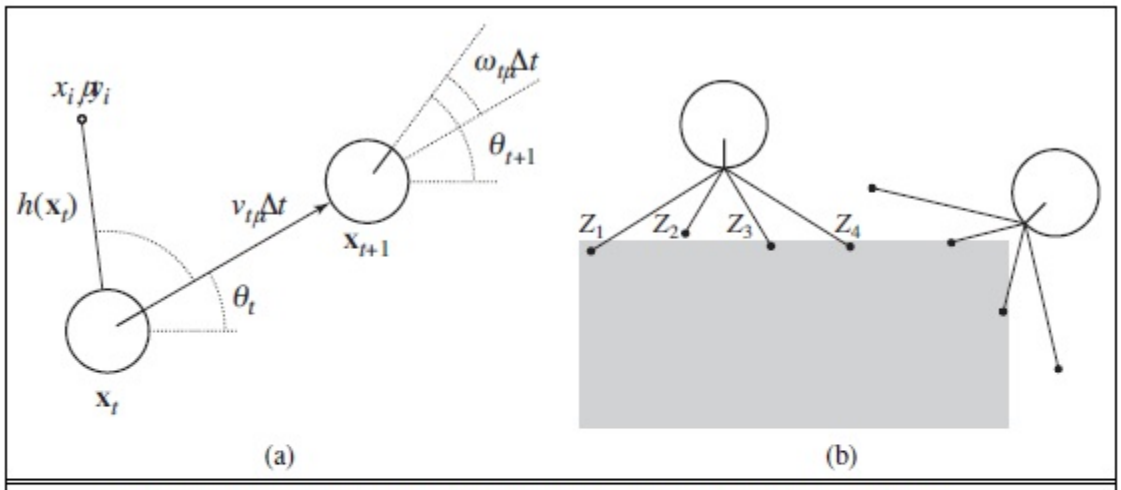
This equation states that the posterior over the state variables **X** at time t + 1 is calculated recursively from the corresponding estimate one-time step earlier. This calculation involves the previous action at and the current sensor measurement **z**t+1. For example, if our goal is to develop a soccer-playing robot, **X**t+1 might be the location of the soccer ball relative to the robot. The posterior **P**(**X**t | **z**1:t, a1:t−1) is a probability distribution over all states that captures what we know from past sensor measurements and controls. how to recursively estimate this location, by incrementally folding in sensor measurements (e.g., camera images) and robot motion commands. The probability **P**(**X**t+1 | **x**t, at) is called the **transition model** or **motion model**, and **P**(**z**t+1 | **X**t+1) is the **sensor model**.

25.3.1 LOCALIZATION AND MAPPING

Localization is the problem of finding out where things are—including the robot itself. Knowledge about where things are is at the core of any successful physical interaction with the environment. For

example, robot manipulators must know the location of objects they seek to manipulate; navigating robots must know where they are to find their way around.

To keep things simple, let us consider a mobile robot that moves slowly in a flat 2D world. Let us also assume the robot is given an exact map of the environment. (An example of such a map appears in Figure 25.10.) The pose of such a mobile robot is defined by its two Cartesian coordinates with values x and y and its heading with value θ, as illustrated. If we arrange those three values in a vector, then any particular state is given by **X**t =(xt, yt, θt)_.



(a) A simplified kinematic model of a mobile robot. The robot is shown as a circle with an interior line marking the forward direction. The state $\mathbf{x}_t$ consists of the (x_t, y_t) position (shown implicitly) and the orientation θ_t. The new state $\mathbf{x}_{t+1}$ is obtained by an update in position of $v_t\Delta_t$ and in orientation of $\omega_t\Delta_t$. Also shown is a landmark at (x_i, y_i) observed at time t. (b) The range-scan sensor model. Two possible robot poses are shown for a given range scan (z_1, z_2, z_3, z_4). It is much more likely that the pose on the left generated the range scan than the pose on the right.

In the kinematic approximation, each action consists of the "instantaneous" specification of two velocities—a translational velocity vt and a rotational velocity ωt. For small time intervals Δt, a crude deterministic model of the motion of such robots is given by

$$\hat{\mathbf{X}}_{t+1} = f(\mathbf{X}_t, \underbrace{v_t, \omega_t}_{a_t}) = \mathbf{X}_t + \begin{pmatrix} v_t\Delta t \cos\theta_t \\ v_t\Delta t \sin\theta_t \\ \omega_t\Delta t \end{pmatrix}$$

The notation ˆ**X** refers to a deterministic state prediction. Of course, physical robots are somewhat unpredictable. This is commonly modelled by a Gaussian distribution with mean f(**X**t, vt, ωt) and covariance **Σ**x. (See Appendix A for a mathematical definition.)

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{X}_t, v_t, \omega_t) = N(\hat{\mathbf{X}}_{t+1}, \boldsymbol{\Sigma}_x) .$$

Next, we need a sensor model. We will consider two kinds of sensor model. The first assumes that the sensors detect stable, recognizable features of the environment called **landmarks**. For each landmark, the range and bearing are reported. Suppose the robot's state is **x**t =(xt, yt, θt)_ and it senses a landmark whose location is known to be (xi, yi)_. Without noise, the range and bearing can be calculated by simple geometry.

$$\hat{\mathbf{z}}_t = h(\mathbf{x}_t) = \begin{pmatrix} \sqrt{(x_t - x_i)^2 + (y_t - y_i)^2} \\ \arctan \frac{y_i - y_t}{x_i - x_t} - \theta_t \end{pmatrix}$$

A somewhat different sensor model is used for an array of range sensors, each of which has a fixed bearing relative to the robot. Such sensors produce a vector of range values **z**t = (z1, . . . , zM)_. Given a pose **x**t, let ˆzj be the exact range along the jth beam direction from **x**t to the nearest obstacle. As before, this will be corrupted by Gaussian noise. Typically, we assume that the errors for the different beam directions are independent and identically distributed, so we have

$$P(\mathbf{z}_t \mid \mathbf{x}_t) = \alpha \prod_{j=1}^{M} e^{-(z_j - \hat{z}_j)/2\sigma^2}.$$

function MONTE-CARLO-LOCALIZATION(a, z, N, $P(X'|X,\ v,\ \omega)$, $P(z|z^*)$, m) **returns** a set of samples for the next time step

inputs: a, robot velocities v and ω
z, range scan $z_1, \ldots, z_M$
$P(X'|X,\ v,\ \omega)$, motion model
$P(z|z^*)$, range sensor noise model
m, 2D map of the environment

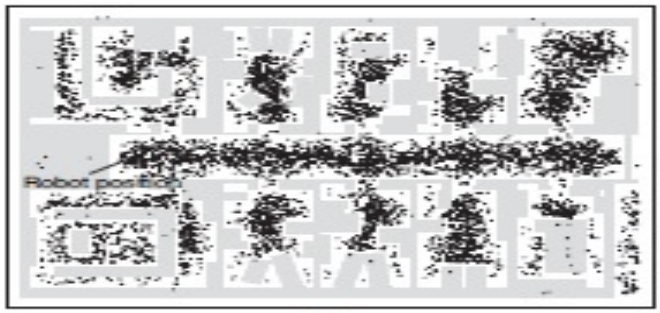
persistent: S, a vector of samples of size N

local variables: W, a vector of weights of size N
S', a temporary vector of particles of size N
W', a vector of weights of size N

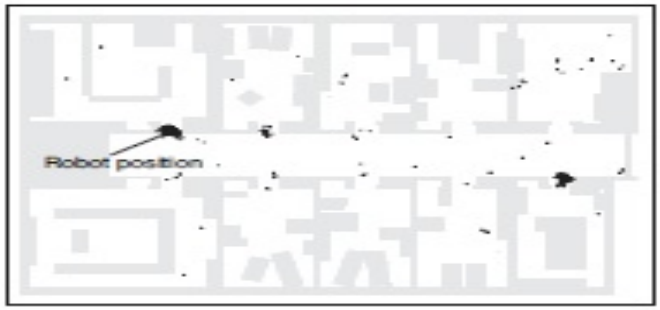
if S is empty **then** /* initialization phase */
  for $i = 1$ to N **do**
    $S[i] \leftarrow$ sample from $P(X_0)$
  for $i = 1$ to N **do** /* update cycle */
    $S'[i] \leftarrow$ sample from $P(X'|X = S[i], v, \omega)$
    $W'[i] \leftarrow 1$
    for $j = 1$ to M **do**
      $z^* \leftarrow$ RAYCAST($j, X = S'[i], m$)
      $W'[i] \leftarrow W'[i] \cdot P(z_j | z^*)$
  $S \leftarrow$ WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S', W')
return S

A Monte Carlo localization algorithm using a range-scan sensor model with independent noise.

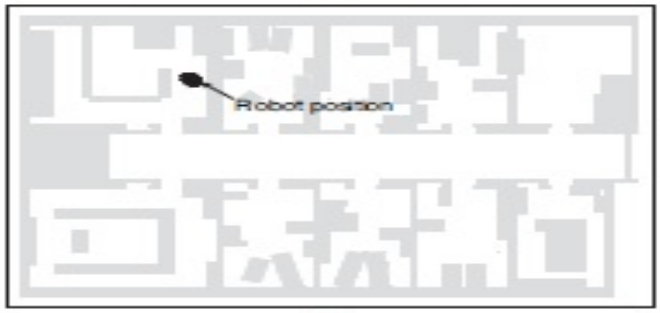
Localization using particle filtering MONTE CARLO is called **Monte Carlo localization**, or MCL. The MCL algorithm is an instance of the particle-filtering algorithm of All we need to do is supply the appropriate motion model and sensor model. shows one version using the range-scan model. The operation of the algorithm is illustrated in as the robot finds out where it is inside an office building. In the first image, the particles are uniformly distributed based on the prior, indicating global uncertainty about the robot's position. In the second image, the first set of measurements arrives and the particles form clusters in the areas of high posterior belief. In the third, enough measurements are available to push all the particles to a single location.



(a)

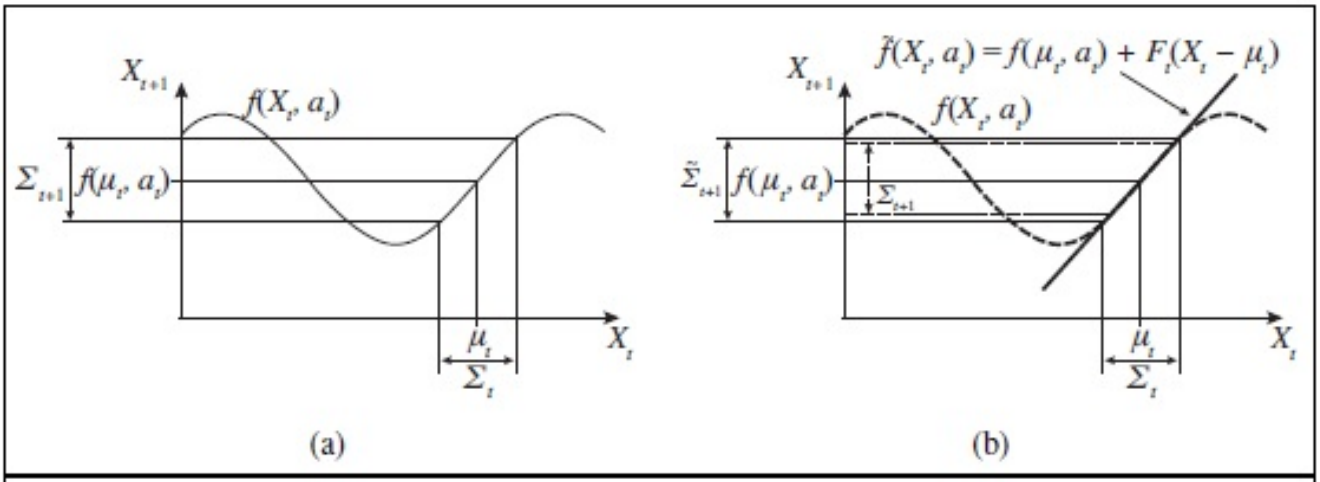


(b)

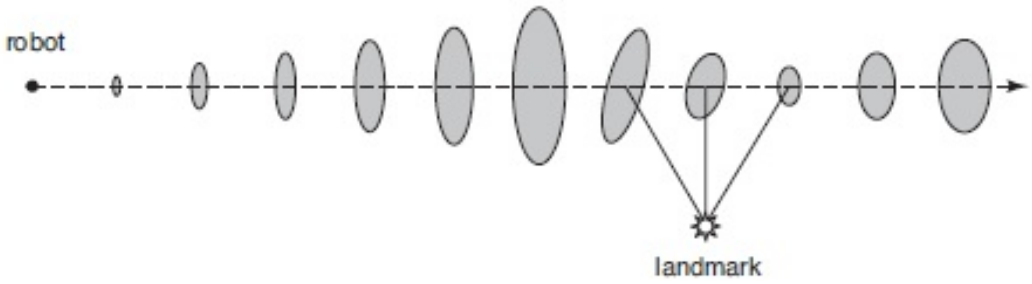


(c)

Monte Carlo localization, a particle filtering algorithm for mobile robot localization. (a) Initial, global uncertainty. (b) Approximately bimodal uncertainty after navigating in the (symmetric) corridor. (c) Unimodal uncertainty after entering a room and finding it to be distinctive.



One-dimensional illustration of a linearized motion model: (a) The function f, and the projection of a mean $\boldsymbol{\mu}_t$ and a covariance interval (based on $\boldsymbol{\Sigma}_t$) into time $t + 1$. (b) The linearized version is the tangent of f at $\boldsymbol{\mu}_t$. The projection of the mean $\boldsymbol{\mu}_t$ is correct. However, the projected covariance $\tilde{\boldsymbol{\Sigma}}_{t+1}$ differs from $\boldsymbol{\Sigma}_{t+1}$.



Example of localization using the extended Kalman filter. The robot moves on a straight line. As it progresses, its uncertainty increases gradually, as illustrated by the error ellipses. When it observes a landmark with known position, the uncertainty is reduced.

the state vector to include the locations of the landmarks in the environment. Luckily, the EKF update scales quadratically, so for small maps (e.g., a few hundred landmarks) the computation is quite feasible. Richer maps are often obtained using graph relaxation methods, similar to the Bayesian network inference techniques.

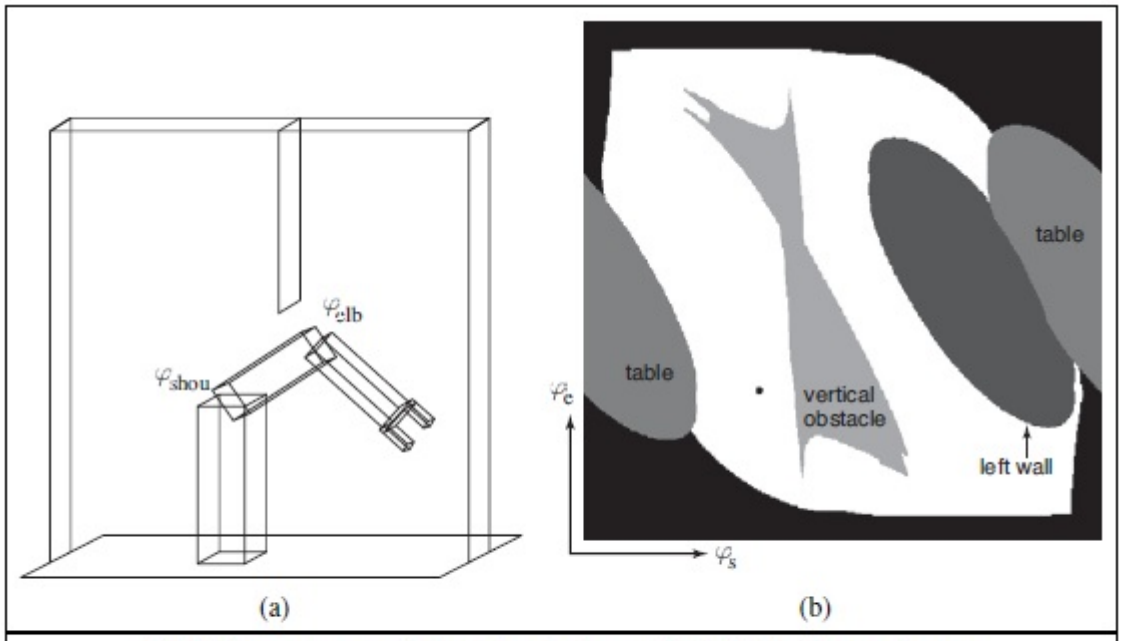
PLANNING TO MOVE

All of a robot's deliberations ultimately come down to deciding how to move effectors. The **point-to-point motion** problem is to deliver the robot or its end effector to a designated target location. A greater challenge is the **compliant motion** problem, in which a robot moves while being in physical contact with an

obstacle. An example of compliant motion is a robot manipulator that screws in a light bulb, or a robot that pushes a box across a table top. We begin by finding a suitable representation in which motion-planning problems can be described and solved. It turns out that the **configuration space**—the space of robot states defined by location, orientation, and joint angles—is a better place to work than the original 3D space. The **path lanning** problem is to find a path from one configuration to another in configuration space. We have already encountered various versions of the path-planning problem throughout this book; the complication added by robotics is that path planning involves continuous spaces. There are two main approaches: **cell decomposition** and **skeletonization**. Each reduces the continuous path-planning problem to a discrete graph-search problem. In this section, we assume that motion is deterministic and that localization of the robot is exact. Subsequent sections will relax these assumptions.

25.4.1 Configuration space

We will start with a simple representation for a simple robot motion problem. Consider the robot arm shown in Figure 25.14(a). It has two joints that move independently. Moving the joints alters the (x, y) coordinates of the elbow and the gripper. (The arm cannot move in the z direction.) This suggests that the robot's configuration can be described by a four-dimensional coordinate: (xe, ye) for the location of the elbow relative to the environment and (xg, yg) for the location of the gripper. Clearly, these four coordinates characterize the full state of the robot. They constitute what is known as **workspace representation**, since the coordinates of the robot are specified in the same coordinate system as the objects it seeks to manipulate (or to avoid). Workspace representations are well-suited for collision checking, especially if the robot and all objects are represented by simple polygonal models. The problem with the workspace representation is that not all workspace coordinates are actually attainable, even in the absence of obstacles. This is because of the **linkage constraints** on the space of attainable workspace coordinates. For example, the elbow position (xe, ye) and the gripper position (xg, yg) are always a fixed distance apart, because they are joined by a rigid forearm. A robot motion planner defined over workspace coordinates faces the challenge of generating paths that adhere to these constraints. This is particularly tricky



(a) Workspace representation of a robot arm with 2 DOFs. The workspace is a box with a flat obstacle hanging from the ceiling. (b) Configuration space of the same robot. Only white regions in the space are configurations that are free of collisions. The dot in this diagram corresponds to the configuration of the robot shown on the left.

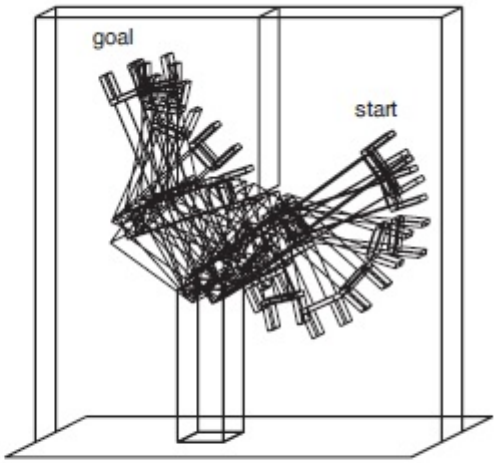
Unfortunately, configuration spaces have their own problems. The task of a robot is usually expressed in workspace coordinates, not in configuration space coordinates. This raises the question of how to map between workspace coordinates and configuration space. Transforming configuration space coordinates into workspace coordinates is simple: it involves a series of straightforward coordinate transformations. These transformations are linear for prismatic joints and trigonometric for revolute joints. This chain of coordinate transformation is known as **kinematics**.

25.4.2 Cell decomposition methods

The first CELL approach to path planning uses **cell decomposition**—that is, it decomposes the free space into a finite number of contiguous regions, called cells. These regions have the important property that the path-planning problem within a single region can be solved by simple means (e.g., moving along a straight line). The path-planning problem then becomes a discrete graph-search problem, very much like the search problems introduced in Chapter 3. The simplest cell decomposition consists of a regularly spaced grid. shows a square grid decomposition of the space and a solution path that is optimal for this grid size. Grayscale shading indicates the *value* of each free-space grid cell—i.e., the cost of the shortest path from that cell to the goal. (These values can be computed by a deterministic form of the VALUE-ITERATION shows the corresponding workspace trajectory for the arm. Of course, we can also use the A

algorithm to find a shortest path. Such a decomposition has the advantage that it is extremely simple to implement, but it also suffers from three limitations. First, it is workable only for low-dimensional configuration spaces, because the number of grid cells increases exponentially with d, the number of dimensions. Sounds familiar? This is the curse! Dimensionality @of dimensionality. Second, there is the problem of what to do with cells that are “mixed”—that is, neither entirely within free space nor entirely within occupied space. A solution path that includes such a cell may not be a real solution, because there may be no way to cross the cell in the desired direction in a straight line. This would make the path planner *unsound*. On the other hand, if we insist that only completely free cells may be used, the planner will be *incomplete*, because it might

(a)



(b)

(a) Value function and path found for a discrete grid cell approximation of the configuration space. (b) The same path visualized in workspace coordinates. Notice how the robot bends its elbow to avoid a collision with the vertical obstacle.

Cell decomposition methods can be improved in a number of ways, to alleviate some of these problems. The first approach allows *further subdivision* of the mixed cells—perhaps using cells of half the original size. This can be continued recursively until a path is found that lies entirely within free cells. (Of course, the method only works if there is a way to decide if a given cell is a mixed cell, which is easy only if the configuration space boundaries have relatively simple mathematical descriptions.) This method is complete provided there is a bound on the smallest passageway through which a solution must pass. Although it focuses most of the computational effort on the tricky areas within the configuration space, it still fails to scale well to high-dimensional problems because each recursive splitting of a cell creates 2d smaller cells. A second way to obtain a complete algorithm is to insist on an **exact cell decomposition** of the free space. This

EXACT CELL method must allow cells to be irregularly shaped where they meet the boundaries of free space, but the shapes must still be “simple” in the sense that it should be easy to compute a traversal of any free cell. This technique requires some quite advanced geometric ideas, so we shall not pursue it further here.

The exact, continuous state that was attained with the cell was first expanded in the search. Assume further, that when propagating information to nearby grid cells, we use this continuous state as a basis, and apply the continuous robot motion model for jumping to nearby cells. In doing so, we can now guarantee that the resulting trajectory is smooth and can indeed be executed by the robot. One algorithm that implements this is **hybrid A***.

25.4.3 Modified cost functions

This problem can be solved by introducing a **potential field**. A potential field is a function defined over state space, whose value grows with the distance to the closest obstacle. shows such a potential field—the darker a configuration state, the closer it is to an obstacle. The potential field can be used as an additional cost term in the shortest-path calculation. This induces an interesting trade off. On the one hand, the robot seeks to minimize path length to the goal. On the other hand, it tries to stay away from obstacles by virtue of minimizing the potential function. With the appropriate weight balancing the two objectives, a resulting path may look like the one shown in Figure 25.17(b). This figure also displays the value function derived from the combined cost function, again calculated by value iteration. Clearly, the resulting path is longer, but it is also safer. There exist many other ways to modify the cost function. For example, it may be desirable to *smooth* the control parameters over time. For example, when driving a car, a smooth path is better than a jerky one. In general, such higher-order constraints are not easy to accommodate in the planning process, unless we make the most recent steering command a part of the state. However, it is often easy to smooth the resulting trajectory after planning, using conjugate gradient methods. Such post-planning smoothing is essential in many real world applications.

PLANNING UNCERTAIN MOVEMENTS

Most of today’s robots use deterministic algorithms for decision making, such as the path-planning algorithms of the previous section. To do so, it is common practice to extract MOST LIKELY STATE the **most likely state** from the probability distribution produced by the state estimation algorithm. The advantage of this approach is purely computational. Planning paths through configuration space is already a challenging problem; it would be worse if we had to work with a full probability distribution over states.

Ignoring uncertainty in this way works when the uncertainty is small. In fact, when the environment model changes over time as the result of incorporating sensor measurements, many robots plan paths online during plan execution. ONLINE REPLANNING This is the **online re-planning** technique.

The field of robotics has adopted a range of techniques for accommodating uncertainty. Some are derived from the algorithms given in for decision making under uncertainty. If the robot faces uncertainty only in its state transition, but its state is fully observable, the problem is best modeled as a Markov decision process (MDP). The solution of an MDP is an optimal **policy**, which tells the robot what to do in every possible state. In this way, it can handle all sorts of motion errors, whereas a single-path solution from a deterministic planner would be much less robust. In robotics, policies are called **navigation** NAVIGATION **functions**. The value FUNCTION

Just as in, partial observability makes the problem much harder. The resulting robot control problem is a partially observable MDP, or POMDP. In such situations, the robot maintains an internal belief state, like the ones discussed. The solution to a POMDP is a policy defined over the robot's belief state. Put differently, the input to the policy is an entire probability distribution. This enables the robot to base its decision not only on what it knows, but also on what it does not know. For example, if it is uncertain INFORMATION about a critical state variable, it can rationally invoke an **information gathering action**. This GATHERING ACTION is impossible in the MDP framework, since MDPs assume full observability. Unfortunately, techniques that solve POMDPs exactly are inapplicable to robotics—there are no known techniques for high-dimensional continuous spaces. Discretization produces POMDPs that are far too large to handle. One remedy is to make the minimization of uncertainty a control object COASTAL For example, the **coastal navigation** heuristic requires the robot to stay near known NAVIGATION landmarks to decrease its uncertainty. Another approach applies variants of the probabilistic roadmap planning method to the belief space representation. Such methods tend to scale better to large discrete POMDPs.

MOVING

So far, we have talked about how to *plan* motions, but not about how to *move*. Our plans particularly those produced by deterministic path planners—assume that the robot can simply follow any path that the algorithm produces. In the real world, of course, this is not the case. Robots have inertia and cannot execute arbitrary paths except at arbitrarily slow speeds. In most cases, the robot gets to exert forces rather than specify positions. This section discusses methods for calculating these forces.

25.6.1 Dynamics and control

Section 25.2 introduced the notion of **dynamic state**, which extends the kinematic state of a robot by its velocity. For example, in addition to the angle of a robot joint, the dynamic state also captures the rate of change of the angle, and possibly even its momentary acceleration. The transition model for a dynamic state representation includes the effect of forces on this rate of change. Such models are typically DIFFERENTIAL expressed via **differential equations**, which are equations that relate a quantity (e.g., a kinematic state) to the change of the quantity over time (e.g., velocity). In principle, we could have chosen to plan robot motion using dynamic models, instead of our kinematic models. Such a methodology would lead to superior robot performance, if we could generate the plans. However, the dynamic state has higher dimension than the kinematic space, and the curse of dimensionality would render many motion planning algorithms inapplicable for all but the most simple robots. For this reason, practical robot system often rely on simpler kinematic path planners. A common technique to compensate for the limitations of kinematic plans is to use a CONTROLLER separate mechanism, a **controller**, for keeping the robot on track. Controllers are techniques for generating robot controls in real time using feedback from the environment, so as to achieve a control objective. If the objective is to keep the robot on a preplanned path, it is often referred to as a **reference controller** and the path is called a **reference path**. Controllers that optimize a global cost function are known as **optimal controllers**. Optimal policies for continuous MDPs are, in effect, optimal controllers. On the surface, the problem of keeping a robot on a prespecified path appears to be relatively straightforward. In practice, however, even this seemingly simple problem has its pitfalls. illustrates what can go wrong; it shows the path of a robot that attempts to follow a kinematic path. Whenever a deviation occurs—whether due to noise or to constraints on the forces the robot can apply—the robot provides an opposing force whose magnitude is proportional to this deviation. Intuitively, this might appear plausible, since deviations should be compensated by a counterforce to keep the robot on track. However, illustrates, our controller causes the robot to vibrate rather violently. The vibration is the result of a natural inertia of the robot arm: once driven back to its reference position the robot then overshoots, which induces a symmetric error with opposite sign. Such overshooting may continue along an entire trajectory, and the resulting robot motion is far from desirable.

ROBOTIC SOFTWARE ARCHITECTURES

A methodology for structuring algorithms is SOFTWARE called a **software architecture**. An architecture

includes languages and tools for writing programs, as well as an overall philosophy for how programs can be brought together. Modern-day software architectures for robotics must decide how to combine reactive control and model-based deliberative planning. In many ways, reactive and deliberate techniques have orthogonal strengths and weaknesses. Reactive control is sensor-driven and appropriate for making low-level decisions in real time. However, it rarely yields a plausible solution at the global level, because global control decisions depend on information that cannot be sensed at the time of decision making. For such problems, deliberate planning is a more appropriate choice.

Consequently, most robot architectures use reactive techniques at the lower levels of control and deliberative techniques at the higher levels. We encountered such a combination in our discussion of PD controllers, where we combined a (reactive) PD controller with a (deliberate) path planner. Architectures that combine reactive and deliberate techniques are called **hybrid architectures**.

25.7.1 Sub Sumption architecture

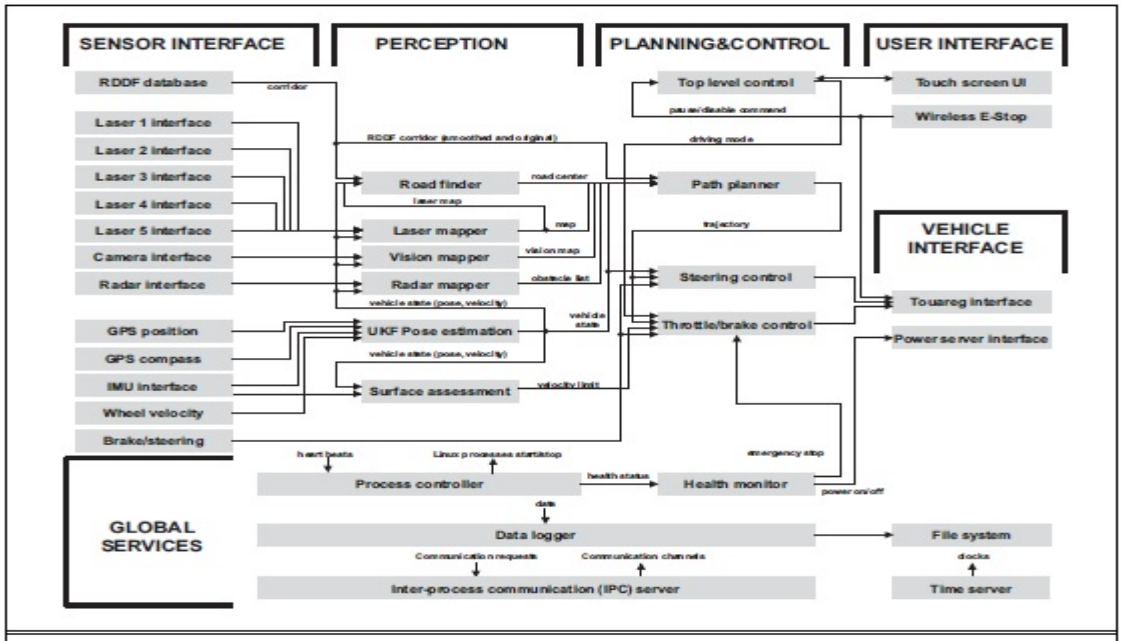
The **sub sumption architecture** (Brooks, 1986) is a framework for assembling reactive con-trollers out of finite state machines. Nodes in these machines may contain tests for certain sensor variables, in which case the execution trace of a finite state machine is conditioned on the outcome of such a test. Arcs can be tagged with messages that will be generated when traversing them, and that are sent to the robot's motors or to other finite state machines. Additionally, finite state machines possess internal timers (clocks) that control the time it takes to traverse an arc. The resulting machines are refereed to as **augmented finite state machines**, or AFSMs, where the augmentation refers to the use of clocks.

An example of a simple AFSM is the four-state machine which generates cyclic leg motion for a hexapod walker. This AFSM implements a cyclic controller, whose execution mostly does not rely on environmental feedback. The forward swing phase, however, does rely on sensor feedback. If the leg is stuck, meaning that it has failed to execute the forward swing, the robot retracts the leg, lifts it up a little higher, and attempts to execute the forward swing once again. Thus, the controller is able to *react* to contingencies arising from the interplay of the robot and its environment.

The sub sumption architecture offers additional primitives for synchronizing AFSMs, and for combining output values of multiple, possibly conflicting AFSMs. In this way, it enables the programmer to compose increasingly complex controllers in a bottom-up fashion.

25.7.2 Three-layer architecture

Hybrid architectures combine reaction with deliberation. The most popular hybrid architecture is the **three-layer architecture**, which THREE-LAYER consists of a reactive layer, an executive layer, and a deliberative layer. The **reactive layer** provides low-level control to the robot. It is characterized by a tight sensor–action loop. Its decision cycle is often on the order of milliseconds. The **executive layer** (or sequencing layer) serves as the glue between the reactive layer and the deliberative layer. It accepts directives by the deliberative layer, and sequences them for the reactive layer. For example, the executive layer might handle a set of via-points generated by a deliberative path planner, and make decisions as to which reactive behaviour to invoke. Decision cycles at the executive layer are usually in the order of a second. The executive layer is also responsible for integrating sensor information into an internal state representation. For example, it may host the robot's localization and online mapping routines.



Software architecture of a robot car. This software implements a data pipeline, in which all modules process data simultaneously.

25.7.3 Pipeline architecture

Another architecture for robots is known as the **pipeline architecture**. Just like the sub sumption architecture, the pipeline architecture executes multiple process in parallel. However, the specific modules in this architecture resemble those in the three-layer architecture. pipeline architecture, which is used to control an autonomous car. Data enters this pipeline at the **sensor interface layer**. The **perception layer**

APPLICATION DOMAINS

Here are some of the prime application domains for robotic technology. **Industry and Agriculture.** Traditionally, robots have been fielded in areas that require difficult human labor, yet are structured enough

to be amenable to robotic automation. The best example is the assembly line, where manipulators routinely perform tasks such as assembly, part placement, material handling, welding, and painting. In many of these tasks, robots have become more cost-effective than human workers. Outdoors, many of the heavy machines that we use to harvest, mine, or excavate earth have been turned into robots. For example, a project at Carnegie Mellon University has demonstrated that robots can strip paint off large ships about 50 times faster than people can, and with a much-reduced environmental impact. Prototypes of autonomous mining robots have been found to be faster and more precise than people in transporting ore in underground mines. Robots have been used to generate high-precision maps of abandoned mines and sewer systems. While many of these systems are still in their prototype stages, it is only a matter of time until robots will take over much of the semi mechanical work that is presently performed by people.

(a) (b)

(a) Robotic car BOSS, which won the DARPA Urban Challenge. Courtesy of Carnegie Mellon University. (b) Surgical robots in the operating room. Image courtesy of da Vinci Surgical Systems.

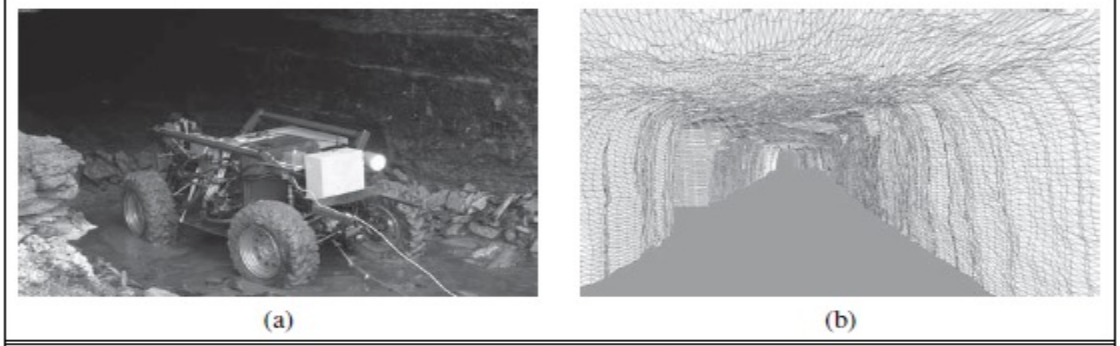
Transportation. Robotic transportation has many facets: from autonomous helicopters that deliver payloads to hard-to-reach locations, to automatic wheelchairs that transport people who are unable to control wheelchairs by themselves, to autonomous straddle carriers that outperform skilled human drivers when transporting containers from ships to trucks on loading docks. A prime example of indoor transportation robots, or gofers, is the Helpmate robot. This robot has been deployed in dozens of hospitals to transport food and other items. In factory settings, autonomous vehicles are now routinely deployed to transport goods in warehouses and between production lines. Many of these robots require environmental modifications for their operation. The most common modifications are localization aids such as inductive loops in the floor, active beacons, or barcode tags. An open challenge in robotics is the design of robots that can use natural cues, instead of artificial devices, to navigate, particularly in environments such as the deep ocean where GPS is unavailable.

Robotic cars. Most of use cars every day. Many of us make cell phone calls while driving. Some of us even text. The sad result: more than a million people die every year in traffic accidents. Robotic cars like BOSS and STANLEY offer hope: Not only will they make driving much safer, but they will also free us from the need to pay attention to the road during our daily commute.

Progress in robotic cars was stimulated by the DARPA Grand Challenge, a race over 100 miles of rehearsed desert terrain, which represented a much more challenging task than had ever been accomplished before. Stanford's STANLEY vehicle completed the course in less than seven hours in 2005, winning a $2 million prize and a place in the National Museum of American History. Figure 25.28(a) depicts BOSS, which in 2007 won the DARPA Urban Challenge, a complicated road race on city streets where robots faced other robots and had to obey traffic rules.

Health care. Robots are increasingly used to assist surgeons with instrument placement when operating on organs as intricate as brains, eyes, and hearts.. Robots have become indispensable tools in a range of surgical procedures, such as hip replacements, thanks to their high precision. In pilot studies, robotic devices have been found to reduce the danger of lesions when performing colonoscopy. Outside the operating room, researchers have begun to develop robotic aides for elderly and handicapped people, such as intelligent robotic walkers and intelligent toys that provide reminders to take medication and provide comfort. Researchers are also working on robotic devices for rehabilitation that aid people in performing certain exercises.

Hazardous environments. Robots have assisted people in cleaning up nuclear waste, most notably in Chernobyl and Three Mile Island. Robots were present after the collapse of the World Trade Center, where they entered structures deemed too dangerous for human search and rescue crews. Some countries have used robots to transport ammunition and to defuse bombs—a notoriously dangerous task. A number of research projects are presently developing prototype robots for clearing minefields, on land and at sea. Most existing robots for these tasks are teleoperated—a human operates them by remote control. Providing such robots with autonomy is an important next step.



(a) (b)

(a) A robot mapping an abandoned coal mine. (b) A 3D map of the mine acquired by the robot.

Entertainment. Robots have begun to conquer the entertainment and toy industry. we see **robotic soccer**, a competitive game very much like human soccer, but played with autonomous mobile robots. Robot soccer provides great opportunities for research in AI, since it raises a range of problems relevant to many other, more serious robot applications. Annual robotic soccer competitions have attracted large numbers of AI researchers and added a lot of excitement to the field of robotics.

Human augmentation. A final application domain of robotic technology is that of human augmentation. Researchers have developed legged walking machines that can carry people around, very much like a wheelchair. Several research efforts presently focus on the development of devices that make it easier for people to walk or move their arms by providing additional forces through extra skeletal attachments.

What is strong AI?

Strong artificial intelligence (AI), also known as artificial general intelligence (AGI) or general AI, is a theoretical form of AI used to describe a certain mindset of AI development. If researchers are able to develop Strong AI, the machine would require an intelligence equal to humans; it would have a self-aware consciousness that has the ability to solve problems, learn, and plan for the future.

Strong AI aims to create intelligent machines that are indistinguishable from the human mind. But just like a child, the AI machine would have to learn through input and experiences, constantly progressing and advancing its abilities over time.

While AI researchers in both academia and private sectors are invested in the creation of artificial general intelligence (AGI), it only exists today as a theoretical concept versus a tangible reality. While some individuals, like Marvin Minsky, have been quoted as being overly optimistic in what we could accomplish in a few decades in the field of AI; others would say that Strong AI systems cannot even be developed. Until

the measures of success, such as intelligence and understanding, are explicitly defined, they are correct in this belief. For now, many use the Turing test to evaluate intelligence of an AI system.

Tests of Strong AI

Turing Test

Alan Turing developed the Turing Test in 1950 and discussed it in his paper, "Computing Machinery and Intelligence" (PDF, 566 KB) (link resides outside IBM). Originally known as the Imitation Game, the test evaluates if a machine's behavior can be distinguished from a human. In this test, there is a person known as the "interrogator" who seeks to identify a difference between computer-generated output and human-generated ones through a series of questions. If the interrogator cannot reliably discern the machines from human subjects, the machine passes the test. However, if the evaluator can identify the human responses correctly, then this eliminates the machine from being categorized as intelligent.

While there are no set evaluation guidelines for the Turing Test, Turing did specify that a human evaluator will only have a 70% chance of correctly predicting a human vs computer-generated conversation after 5 minutes. The Turing Test introduced general acceptance around the idea of machine intelligence. However, the original Turing Test only tests for one skill set — text output or chess as examples. Strong AI needs to perform a variety of tasks equally well, leading to the development of the Extended Turing Test. This test evaluates textual, visual, and auditory performance of the AI and compares it to human-generated output. This version is used in the famous Loebner Prize competition, where a human judge guesses whether the output was created by a human or a computer.

Chinese Room Argument (CRA)

The Chinese Room Argument was created by John Searle in 1980. In his paper, he discusses the definition of understanding and thinking, asserting that computers would never be able to do this. In this excerpt from his paper, from Stanford's website (link resides outside IBM), summarizes his argument well, "Computation is defined purely formally or syntactically, whereas minds have actual mental or semantic contents, and we cannot get from syntactical to the semantic just by having the syntactical operations and nothing else…A system, me, for example, would not acquire an understanding of Chinese just by going through the steps of a computer program that simulated the behavior of a Chinese speaker (p.17)."

The Chinese Room Argument proposes the following scenario:

Imagine a person, who does not speak Chinese, sits in a closed room. In the room, there is a book with Chinese language rules, phrases and instructions. Another person, who is fluent in Chinese, passes

notes written in Chinese into the room. With the help of the language phrasebook, the person inside the room can select the appropriate response and pass it back to the Chinese speaker.

While the person inside the room was able to provide the correct response using a language phrasebook, he or she still does not speak or understand Chinese; it was just a simulation of understanding through matching question or statements with appropriate responses. Searle argues that Strong AI would require an actual mind to have consciousness or understanding. The Chinese Room Argument illustrates the flaws in the Turing Test, demonstrating differences in definitions of artificial intelligence.

Strong AI vs. weak AI

Weak AI, also known as narrow AI, focuses on performing a specific task, such as answering questions based on user input or playing chess. It can perform one type of task, but not both, whereas Strong AI can perform a variety of functions, eventually teaching itself to solve for new problems. Weak AI relies on human interference to define the parameters of its learning algorithms and to provide the relevant training data to ensure accuracy. While human input accelerates the growth phase of Strong AI, it is not required, and over time, it develops a human-like consciousness instead of simulating it, like Weak AI. Self-driving cars and virtual assistants, like Siri, are examples of Weak AI.

Strong AI trends

While there are no clear examples of strong artificial intelligence, the field of AI is rapidly innovating. Another AI theory has emerged, known as artificial superintelligence (ASI), super intelligence, or Super AI. This type of AI surpasses strong AI in human intelligence and ability. However, Super AI is still purely speculative as we have yet to achieve examples of Strong AI.

With that said, there are fields where AI is playing a more important role, such as:

Cybersecurity: Artificial intelligence will take over more roles in organizations' cybersecurity measures, including breach detection, monitoring, threat intelligence, incident response, and risk analysis.

Entertainment and content creation: Computer science programs are already getting better and better at producing content, whether it is copywriting, poetry, video games, or even movies. OpenAI's GBT-3 text generation AI app is already creating content that is almost impossible to distinguish from copy that was written by humans.

Behavioral recognition and prediction: Prediction algorithms will make AI stronger, ranging from applications in weather and stock market predictions to, even more interesting, predictions of human

behavior. This also raises the questions around implicit biases and ethical AI. Some AI researchers in the AI community are pushing for a set of anti-discriminatory rules, which is often associated with the hashtag #responsibleAI.

Strong AI terms and definitions

The terms artificial intelligence, machine learning and deep learning are often used in the wrong context. These terms are frequently used in describing Strong AI, and so it's worth defining each term briefly:

Artificial intelligence defined by John McCarthy (PDF, 109 KB) (link resides outside IBM), is "the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable."

Machine learning is a sub-field of artificial intelligence. Classical (non-deep) machine learning models require more human intervention to segment data into categories (i.e. through feature learning).

Deep learning is also a sub-field of machine learning, which attempts to imitate the interconnectedness of the human brain using neural networks. Its artificial neural networks are made up layers of models, which identify patterns within a given dataset. They leverage a high volume of training data to learn accurately, which subsequently demands more powerful hardware, such as GPUs or TPUs. Deep learning algorithms are most strongly associated with human-level AI.

To read more about the nuanced differences between these technologies, read "AI vs. Machine Learning vs. Deep Learning vs. Neural Networks: What's the Difference?"

Deep learning applications

Deep learning can handle complex problems well, and as a result, it is utilized in many innovative and emerging technologies today. Deep learning algorithms have been applied in a variety of fields. Here are some examples:

Self-driving cars: Google and Elon Musk have shown us that self-driving cars are possible. However, self-driving cars require more training data and testing due to the various activities that it needs to account for, such as giving right of way or identifying debris on the road. As the technology matures, it'll then need to get over the human hurdle of adoption as polls indicate that many drivers are not willing to use one.

Speech recognition: Speech recognition, like AI chatbots and virtual agents, is a big part of natural language processing. Audio-input is much harder to process for an AI, as so many factors, such as background noise, dialects, speech impediments and other influences can make it much harder for the AI to convert the input into something the computer can work with.

Pattern recognition: The use of deep neural networks improves pattern recognition in various applications. By discovering patterns of useful data points, the AI can filter out irrelevant information, draw useful correlations and improve the efficiency of big data computation that may typically be overlooked by human beings.

Computer programming: Weak AI has seen some success in producing meaningful text, leading to advances within coding. Just recently, OpenAI released GPT-3, an open-source software that can actually write code and simple computer programs with very limited instructions, bringing automation to program development.

Image recognition: Categorizing images can be very time consuming when done manually. However, special adaptions of deep neural networks, such as DenseNet, which connects each layer to every other layer in the neural network, have made image recognition much more accurate.

Contextual recommendations: Deep learning apps can take much more context into consideration when making recommendations, including language understanding patterns and behavioral predictions.

Fact checking: The University of Waterloo recently released a tool that can detect fake news by verifying the information in articles by comparing it with other news sources.