

Artificial Intelligence

Unit I

1.1 Introduction

1.1.1 What is Artificial Intelligence?

1.1.2 Foundations of AI

1.1.3 History

1.1.4 The state of art AI today

1.2 Intelligent Agents

1.2.1 Agents and environment

1.2.2 Good behavior

1.2.3 Nature of environment

1.2.4 The structure of agents.

1.1 Introduction

We human beings have tried to understand how we think; that is, how a mere handful of matter can perceive, understand, predict, and manipulate a world far larger and more complicated than itself. Whereas Artificial intelligence, or AI attempts not just to understand but also to build intelligent entities.

AI is one of the newest fields in science and engineering. AI currently encompasses a huge variety of subfields, ranging from the general (learning and perception) to the specific, such as playing chess, proving mathematical theorems, writing poetry, driving a car on a crowded street, and diagnosing diseases.

Source: <https://www.youtube.com/watch?v=mJeNghZXtMo&t=263s>

1.1.1 What is Artificial Intelligence?

Thinking Humanly "The exciting new effort to make computers think ... machines with minds, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning" (Bellman, 1978)	Thinking Rationally "The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly "The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)	Acting Rationally "Computational Intelligence is the study of the design of intelligent agents." (Poole et al., 1998) "AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Some definitions of artificial intelligence, organized into four categories.

Categories/Approaches of AI are-

1) Acting humanly: The Turing Test approach

The computer would need to possess the following capabilities:

- natural language processing to enable it to communicate successfully in English;
- knowledge representation to store what it knows or hears;
- automated reasoning to use the stored information to answer questions and to draw new conclusions;
- machine learning to adapt to new circumstances and to detect and extrapolate patterns.
- computer vision to perceive objects, and
- robotics to manipulate objects and move about.

2) Thinking humanly: The cognitive modeling approach

Here, we must have some way of determining how humans think. We need to get inside the actual workings of human minds.

There are three ways to do this: through introspection—trying to catch our own thoughts as they go by; through psychological experiments—observing a person in action; and through brain imaging—observing the brain in action.

Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program.

If the program's input-output behavior matches corresponding human behavior, that is evidence that some of the program's mechanisms could also be operating in humans.

3) Thinking rationally: The “laws of thought” approach

“Socrates is a man; all men are mortal; therefore, Socrates is mortal.” These laws of thought were supposed to govern the operation of the mind; their study initiated the field called logic.

There are two main obstacles to this approach.

First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain.

Second, there is a big difference between solving a problem “in principle” and solving it in practice.

4) Acting rationally: The rational agent approach

An agent is just something that acts. A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

The rational-agent approach has two advantages over the other approaches.

First, it is more general than the “laws of thought” approach because correct inference is just one of several possible mechanisms for achieving rationality.

Second, it is more amenable to scientific development than are approaches based on human behavior or human thought.

1.1.2 Foundations of AI

A. Philosophy

Philosophy answers the following questions-

- Can formal rules be used to draw valid conclusions?
- How does the mind arise from a physical brain?
- Where does knowledge come from?
- How does knowledge lead to action?

- ❖ Logic, methods of reasoning.
- ❖ Mind as a physical system.
- ❖ Foundations of learning, language, and rationality.

Thus, using this field, AI will be having the idea of where exactly the knowledge comes from and how does it leads to an action.

B. Mathematics

Mathematics provided the tasks to manipulate statements of logical certainty as well as uncertain probabilistic statements.

It also sets the groundwork for understanding computation and reasoning about algorithm.

A) Formal representation and proof Development of Formal logic

- 1) Propositional or Boolean logic

- 2) Development of First-Order logic by extending the Boolean logic to include objects and relations
- B) Algorithms, computation, decidability, tractability First algorithm was developed - Euclid's algorithm for computing GCD
- C) Probability
 - Baye's rule is the underlying approach for uncertain reasoning in AI systems.

C. Economics

Economics, formalizes the problem of making decision that maximize the expected outcome to the decision maker.

- How should we make decisions so as to maximize payoff?
- How should we do this when others may not go along?
- How should we do this when the payoff may be far in the future?
 - ☞ Decision theory Game theory
 - ☞ Operations Research
 - ☞ Markov Decision Processes

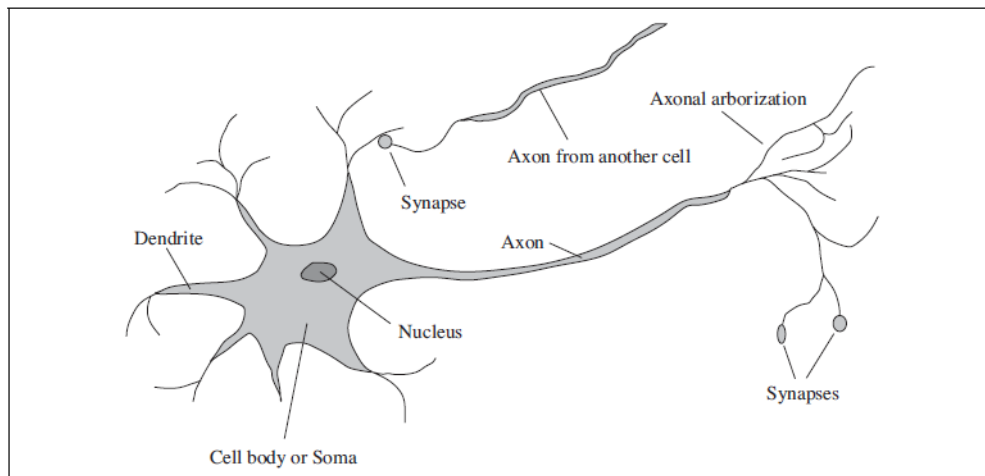
D. Neuroscience

The study of nervous system, particularly the brain. The exact way in which the brain enables thought is unknown. However, it does enable thought has the evidence "A strong blow to the head can lead to mental incapacitation"

Neurons

– Brain consists of nerve cells or neurons. There is no theory on how an individual memory is stored. A collection of simple cells can lead to thought, action, consciousness that "brains cause minds".

Each neuron consists of a cell body or soma, that contains a cell nucleus. Branching out from the cell body are a number of fibers called dendrites and a single long fiber called the axon. A neuron makes connections with 10 to 100,000 other neurons at junctions called synapses. Signals are propagated from neuron to neuron by a complicated electrochemical reaction. The signals control brain activity in the short term and also enable long-term changes in the connectivity of neuron. These mechanisms are thought to form the basis for learning in the brain.



A comparison of resources available at IBM BLUE GENE Supercomputer, a PC of 2008 and the human brain

	Supercomputer	Personal Computer	Human Brain
Computational units	10^4 CPUs, 10^{12} transistors	4 CPUs, 10^9 transistors	10^{11} neurons
Storage units	10^{14} bits RAM 10^{15} bits disk	10^{11} bits RAM 10^{13} bits disk	10^{11} neurons 10^{14} synapses
Cycle time	10^{-9} sec	10^{-9} sec	10^{-3} sec
Operations/sec	10^{15}	10^{10}	10^{17}
Memory updates/sec	10^{14}	10^{10}	10^{14}

Note: The brain's numbers are fixed whereas the supercomputer's numbers increase by a factor of 10 every 5 year or so. The PC lags behind on all metrics except cycle time. Brains and digital computers have somewhat different properties. Computers have a cycle time that is a million times faster than a brain. The brain makes

up for that with far more storage and interconnection than even a high-end personal computer. The largest supercomputers have a capacity that is similar to the brain's. Even with a computer of virtually unlimited capacity, we still would not know how to achieve the brain's level of intelligence

- Early studies (1824) relied on injured and abnormal people to understand what parts of brain work. More recent studies use accurate sensors to correlate brain activity to human thought
- The measurement of intact brain activity began in 1929 with the invention by Hans Berger of the electroencephalograph (EEG).
- The recent development of functional magnetic resonance imaging (fMRI) is giving neuroscientists detailed images of brain activity
- Individual neurons can be stimulated electrically, chemically, or even optically allowing neuronal input output relationships to be mapped.

Despite these advances, we are still a long way from understanding how cognitive processes actually work.

E. Psychology

Psychologists adopted the idea that humans and animals can be considered as information processing machines as well as it will search how human and animals think and act. Cognitive psychology, which views the brain as an information-processing device, can be traced back at least to the works of William James (1842–1910).

Craik specified the three key steps of a knowledge-based agent: (1) the stimulus must be translated into an internal representation, (2) the representation is manipulated by cognitive processes to derive new internal representations, and (3) these are in turn retranslated back into action. He clearly explained why this was a good design for an agent:

If the organism carries a “small-scale model” of external reality and of its own possible actions within its head, it is able to try out various alternatives, conclude which is the best of them, react to future situations before they arise, utilize the knowledge of past events in dealing with the present and future, and in every way to react in a much fuller, safer, and more competent manner to the emergencies which face it. (Craik, 1943)

F. Computer Engineering

How can we build an efficient computer?

For artificial intelligence to succeed, we need two things:
intelligence and an artifact.

The computer has been the artifact of choice.

Each generation of computer hardware has brought an increase in speed and capacity and a decrease in price.

In 2005 power dissipation problems led to multiply the number of CPU cores rather than the clock speed.

Current expectations are that future increases in power will come from massive parallelism

- Convergence with the properties of the brain.

The software side of computer science has supplied AI with:

- The operating systems, programming languages, and tools needed to write modem programs

G. Control theory and cybernetics

How can artifacts operate under their own control?

- Previous Assumption: Only living things could modify their behavior in response to changes in the environment
- Machines can modify their behavior in response to the environment (sense/action loop)

Ex: Water-flow regulator, steam engine governor, thermostat

- The theory of stable feedback systems (1894)
- Build systems that transition from initial state to goal state with minimum energy

In 1950, control theory could only describe linear systems.

AI largely rose as a response to this shortcoming.

H. Linguistics (1957 - present)

How does language relate to thought?

- Knowledge representation: the study of how to put knowledge into a form that a computer can reason with
- Grammar
- Speech demonstrates human intelligence
 - Analysis of human language reveals thought taking place in ways not understood in other settings

Language and thought are believed to be tightly intertwined.

Modern linguistics and AI intersect in a hybrid field called computational linguistics or natural language processing.

1.1.3 History

- The gestation of artificial intelligence (1943–1955)
The first work that is now generally recognized as AI was done by Warren McCulloch and Walter Pitts (1943). They proposed a model of artificial neurons in which each neuron is characterized as being “on” nor “off,” with a switch to “on” occurring in response to stimulation by a sufficient number of neighbouring neurons. The state of a neuron was conceived of as “factually equivalent to a proposition which proposed its adequate stimulus.”
Donald Hebb (1949) demonstrated a simple updating rule for modifying the connection strengths between neurons. His rule, now called Hebbian HEBBIAN LEARNING learning,
The birth of artificial intelligence (1956) Princeton was home to another influential figure in AI, John McCarthy. McCarthy convinced Minsky, Claude Shannon, and Nathaniel Rochester to help him bring together U.S. researchers interested in automata theory, neural nets, and the study of intelligence.
Two researchers from Carnegie Tech, Allen Newell and Herbert Simon, had ideas and in some cases programs for particular applications such as checkers.
Newell and Simon already had a reasoning program, the Logic Theorist (LT), about which Simon claimed, “We have invented a computer program capable of thinking non-numerically, and thereby solved the venerable mind–body problem.”
Early enthusiasm, great expectations (1952–1969) Newell and Simon’s early success was followed up with the General Problem Solver, or GPS.
Thus, GPS was probably the first program to embody the “thinking humanly” approach. The success of GPS and subsequent programs as models of cognition led Newell and Simon (1976) to formulate the famous physical symbol system hypothesis, which states that “a physical symbol system has the necessary and PHYSICAL SYMBOL SYSTEM sufficient means for general intelligent action.”
- A dose of reality (1966–1973)
The first kind of difficulty arose because most early programs knew nothing of their subject matter; they succeeded by means of simple syntactic manipulations. In 1966, a report by an advisory committee found that “there has been no machine translation of general scientific text, and none is in immediate prospect.” All U.S. government funding for academic translation projects was canceled. Today, machine translation is an imperfect but widely used tool for technical, commercial, government, and Internet documents.
The illusion of unlimited computational power was not confined to problem-solving MACHINE EVOLUTION programs. Early experiments in machine evolution (now called genetic algorithms) (Friedberg, 1958; Friedberg et al., 1959) were based on the undoubtedly correct belief that by making an appropriate series of small mutations to a machine-code program, one can generate a program with good performance for any particular task.
- Knowledge-based systems: The key to power? (1969–1979)
The picture of problem solving that had arisen during the first decade of AI research. Such approaches have been called weak methods because, although general, they do not scale up to large or difficult problem instances.
The DENDRAL researchers consulted analytical chemists and found that they worked by looking for well-known patterns of peaks in the spectrum. Example: if there are two peaks at x_1 and x_2 such that (a) $x_1 + x_2 = M + 28$ (M is the mass of the whole molecule); (b) $x_1 - 28$ is a high peak; (c) $x_2 - 28$ is a high peak; (d) At least one of x_1 and x_2 is high.
- There is a ketone subgroup AI becomes an industry (1980–present)
In 1981, the Japanese announced the “Fifth Generation” project, a 10-year plan to build intelligent computers running Prolog. Overall, the AI industry boomed from a few million dollars in 1980 to billions of dollars in 1988, including hundreds of companies building expert systems, vision systems, robots, and software and hardware specialized for these purposes.
- AI adopts the scientific method (1987–present)
In terms of methodology, AI has finally come firmly under the scientific method. To be accepted, hypotheses must be subjected to rigorous empirical experiments, and the results must be analysed statistically for their importance (Cohen, 1995)

- The emergence of intelligent agents (1995–present)
A related idea is the subfield of Artificial General Intelligence or AGI (Goertzel and ARTIFICIAL GENERAL INTELLIGENCE Pennachin, 2007), which held its first conference and organized the Journal of Artificial General Intelligence in 2008. AGI looks for a universal algorithm for learning and acting in any environment, and has its roots in the work of Ray Solomonoff (1964), one of the attendees of the original 1956 Dartmouth conference. Guaranteeing that what we create is really FRIENDLY AI Friendly AI is also a concern (Yudkowsky, 2008; Omohundro, 2008).
- The availability of very large data sets (2001–present)
“Knowledge bottleneck” in AI—the problem of how to express all the knowledge that a system needs—may be solved in many applications by learning methods rather than hand-coded knowledge engineering, provided the learning algorithms have enough data to go on (Halevy et al., 2009). Reporters have noticed the surge of new applications and have written that “AI Winter” may be yielding to a new Spring (Havenstein, 2005). As Kurzweil (2005) writes, “today, many thousands of AI applications are deeply embedded in the infrastructure of every industry.”

1.1.4 The state of art AI today

There are so many activities in so many subfields. Here we sample a few applications.

- **Robotic vehicles:** A driverless robotic car named STANLEY sped through the rough terrain of the Mojave desert at 22 mph, finishing the 132-mile course first to win the 2005 DARPA Grand Challenge. STANLEY is a Volkswagen Touareg outfitted with cameras, radar, and laser rangefinders to sense the environment and onboard software to command the steering, braking, and acceleration. The following year CMU’s BOSS won the Urban Challenge, safely driving in traffic through the streets of a closed Air Force base, obeying traffic rules and avoiding pedestrians and other vehicles.
- **Speech recognition:** A traveler calling United Airlines to book a flight can have the entire conversation guided by an automated speech recognition and dialog management system.
- **Robotics:** The iRobot Corporation has sold over two million Roomba robotic vacuum cleaners for home use. The company also deploys the more rugged PackBot to Iraq and Afghanistan, where it is used to handle hazardous materials, clear explosives, and identify the location of snipers.
- **Spam fighting:** Each day, learning algorithms classify over a billion messages as spam, saving the recipient from having to waste time deleting what, for many users, could comprise 80% or 90% of all messages, if not classified away by algorithms. Because the spammers are continually updating their tactics, it is difficult for a static programmed approach to keep up, and learning algorithms work best.
- **Game playing:** IBM’s DEEP BLUE became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match. Kasparov said that he felt a “new kind of intelligence” across the board from him. Newsweek magazine described the match as “The brain’s last stand.” Human champions studied Kasparov’s loss and were able to draw a few matches in subsequent years, but the most recent human-computer matches have been won convincingly by the computer.
- **Machine Translation:** A computer program automatically translates from Arabic to English, allowing an English speaker to see the headline “Ardogan Confirms That Turkey Would Not Accept Any Pressure, Urging Them to Recognize Cyprus.” The program uses a statistical model built from examples of Arabic-to-English translations and from examples of English text totaling two trillion words (Brants et al., 2007). None of the computer scientists on the team speak Arabic, but they do understand statistics and machine learning algorithms.

These are just a few examples of artificial intelligence systems that exist today. Not magic or science fiction—but rather science, engineering, and mathematics.

Source: <https://www.youtube.com/watch?v=oxYhGVHihhA&t=335s>

1.2 Intelligent Agents

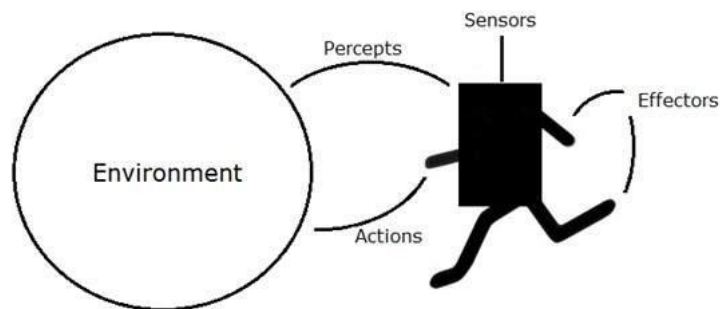
Agent and environment

An agent is anything that can perceive its environment through sensors and acts upon that environment through actuators.

-A human agent has sensory organs such as eyes, ears, nose, tongue and skin parallel to the sensors, and other organs such as hands, legs, mouth, for effectors.

-A robotic agent replaces cameras and infrared range finders for the sensors, and various motors and actuators for effectors.

-A software agent has encoded bit strings as its programs and actions.



Agent Terminology

- Performance Measure of Agent – It is the criteria, which determines how successful an agent is.
- Behavior of Agent – It is the action that agent performs after any given sequence of percepts.
- Percept – It is agent's perceptual inputs at a given instance.
- Percept Sequence – It is the history of all that an agent has perceived till date.
- Agent Function – It is a map from the precept sequence to an action.

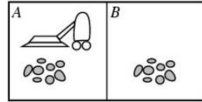
1.2.1 Agents and environment

Agent and environment with respect to vacuum cleaner world

The vacuum-cleaner world:

This world is so simple that we can describe everything that happens; it's also a made-up world, so we can invent many variations. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and where there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square. A partial tabulation of this agent function:

Vacuum-cleaner world



- Percepts: location and contents, e.g., [A,Dirty]
- Actions: *Left*, *Right*, *Suck*
- Function-table (table look-up agent)

Percept	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck

1.2.2 Good behaviour: The concept of rationality

When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well. This notion of desirability is captured by a performance measure that evaluates any given sequence of environment states.

Consider, for example, the vacuum-cleaner agent from the preceding section. We might propose to measure performance by the amount of dirt cleaned up in a single eight-hour shift.

With a rational agent, of course, what you ask for is what you get. A rational agent can maximize this performance measure by cleaning up the dirt, then dumping it all on the floor, then cleaning it up again, and so on. A more suitable performance measure would reward the agent for having a clean floor. For example, one point could be awarded for each clean square at each time step (perhaps with a penalty for electricity consumed and noise generated). As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave.

Rationality

What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

This leads to a definition of a rational agent:

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

Omniscience, learning, and autonomy

We need to be careful to distinguish between rationality and omniscience. An omniscient agent knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality. Consider the following example: I am walking along the Champs Elysées one day and I see an old friend across the street. There is no traffic nearby and I'm not otherwise engaged, so, being rational, I start to cross the street. Meanwhile, at 33,000 feet, a cargo door falls off a passing airliner, and before I make it to the other side of the street I am flattened. Was I irrational to cross the street? It is unlikely that my obituary would read "Idiot attempts to cross street."

This example shows that rationality is not the same as perfection. Rationality maximizes expected performance, while perfection maximizes actual performance. Retreating from a requirement of perfection is not just a question of being fair to agents. The point is that if we expect an agent to do what turns out to be the best action after the fact, it will be impossible to design an agent to fulfill this specification—unless we improve the performance of crystal balls or time machines.

First, it would not be rational to cross the road given this uninformative percept sequence: the risk of accident from crossing without looking is too great. Second, a rational agent should choose the “looking” action before stepping into the street, because looking helps maximize the expected performance. Doing actions in order to modify future percepts—sometimes called information gathering— is an important part of rationality.

1.2.3 Nature of environment

Specify the task environment

We had to specify the performance measure, the environment, and the agent’s actuators and sensors. We group all these under the heading of the task environment.

-For the acronymically minded, we call PEAS this the PEAS (Performance, Environment, Actuators, Sensors) description. In designing an agent, the first step must always be to specify the task environment as fully as possible.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Figure 2.4 PEAS description of the task environment for an automated taxi.

PEAS

- PEAS stands for Performance Measure, Environment, Actuators, and Sensors. It is the short form used for performance issues grouped under Task Environment.
- Performance Measure: It is the objective function to judge the performance of the agent. For example, in case of pick and place robot, number of correct parts in a bin can be the performance measure.
- Environment: It is the real environment where the agent needs to deliberate actions.
- Actuators: These are the tools, equipment or organs using which agent performs actions in the environment. This works as the output of the agent.
- Sensors: These are the tools, equipment or organs using which agent captures the state of the environment. This works as the input to the agent.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

Figure 2.5 Examples of agent types and their PEAS descriptions.

Properties of task environments: -

a) Fully observable vs. partially observable:

- If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable. A task environment is effectively fully observable if the sensors detect all aspects that are relevant to the choice of action; relevance, in turn, depends on the performance measure. Fully observable environments are convenient because the agent need not maintain any internal state to keep track of the world.
- An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data—for example, a vacuum agent with only a local dirt sensor cannot tell whether there is dirt in other squares, and an automated taxi cannot see what other drivers are thinking. If the agent has no sensors at all then the environment is unobservable.

b) Single agent vs. multiagent:

- The distinction between single-agent and multiagent environments may seem simple enough.

c) Deterministic vs stochastic:

- If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic.

d) Episodic vs. sequential:

- In an episodic task environment, the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and then performs a single action. Crucially, the next episode does not depend on the actions taken in previous episodes. Many classification tasks are episodic.
- In sequential environments, on the other hand, the current decision could affect all future decisions. Chess and taxi driving are sequential: in both cases, short-term actions can have long-term consequences. Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

e) Static vs. dynamic:

- If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise, it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time.
- Dynamic environments, on the other hand, are continuously asking the agent what it wants to do; if it hasn't decided yet, that counts as deciding to do nothing

f) Discrete vs. continuous:

- The discrete/continuous distinction applies to the state of the environment, to the way time is handled, and to the percepts and actions of the agent. For example, the chess environment has a finite number of distinct states

g) Known vs. unknown:

- In a known environment, the outcomes (or outcome probabilities if the environment is stochastic) for all actions are given.
- Obviously, if the environment is unknown, the agent will have to learn how it works in order to make good decisions.

Examples of task environments and their characteristics: -

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi Driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical Diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image Analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-Picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery Controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English Controller	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

1.2.4 The structure of agents.

The job of AI is to design an agent program that implements the agent function—the mapping from percepts to actions. We assume this program will run on some sort of computing device with physical sensors and actuators—we call this the architecture:

$$\text{agent} = \text{architecture} + \text{program}$$

Agent programs:

The agent programs that we design in this book all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuators. Notice the difference between the agent program, which takes the current percept as input, and the agent function, which takes the entire percept history. The agent program takes just the current percept as input because nothing more is available from the environment; if the agent's actions need to depend on the entire percept sequence, the agent will have to remember the percepts.

function TABLE-DRIVEN-AGENT (percept) returns an action persistent: percepts, a sequence, initially empty

```

        table, a table of actions, indexed by percept sequences, initially fully
        specified
    append percept to the end of percepts
    action ← LOOKUP (percepts, table)
    return action

```

The TABLE-DRIVEN-AGENT Program is invoked foreach new percept and returns an action each time. It retains the complete percept sequence in memory.

The job of AI is to design an agent program that implements the agent function—the mapping from percepts to actions. We assume this program will run on some sort of computing device with physical sensors and actuators—we call this the architecture :

agent = architecture + program .

Four basic kinds of agent programs that embody the principles underlying almost all intelligent systems:

- Simple reflex agents;
- Model-based reflex agents;
- Goal-based agents;
- Utility-based agents.

Each kind of agent program combines particular components in particular ways to generate actions.

Simple reflex agent:

The simplest kind of agent is the simple reflex agent. These agents select actions on the basis of the current percept, ignoring the rest of the percept history.

Simple reflex agents have the admirable property of being simple, but they turn out to be of limited intelligence.

```

function SIMPLE-REFLEX-AGENT(percept) returns an action
    persistent: rules, a set of condition–action rules

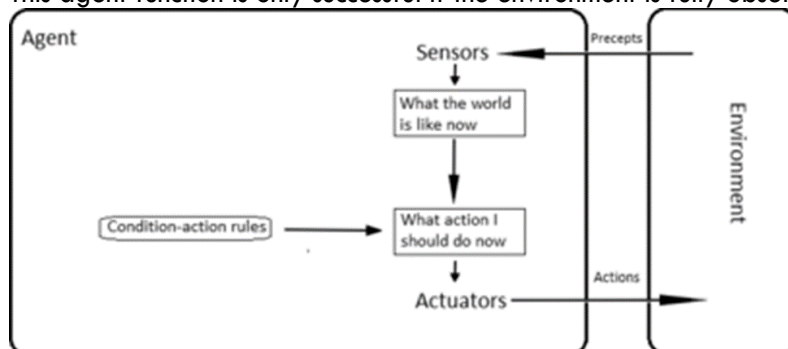
    state ← INTERPRET-INPUT(percept)
    rule ← RULE-MATCH(state, rules)
    action ← rule.ACTION
    return action

```

Figure 2.10 A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

The agent function is based on condition-action rule.

This agent function is only successful if the environment is fully observable.



Problems with simple reflex agents:-

- Very limited intelligence.
- No knowledge of non-perceptual parts of state.
- Usually too big to generate and store.
- If there occurs any change in the environment, then the collection of rules need to be updated.

Model based reflex agents:

It works by finding a rule whose condition matches the current situation.

A model-based agent can handle partially observable environments by use of model about the world.

The agent has to keep track of internal state which is adjusted by each percept and that depends on the percept history.

The current state is stored inside the agent which maintains some kind of structure describing the part of the world which cannot be seen.

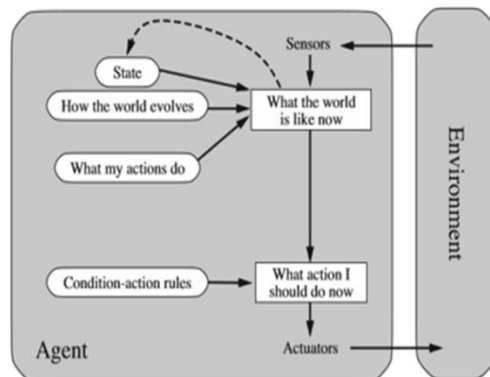
```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
               model, a description of how the next state depends on current state and action
               rules, a set of condition–action rules
               action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept, model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Figure 2.12 A model-based reflex agent. It keeps track of the current state of the world, using an internal model. It then chooses an action in the same way as the reflex agent.

Updating the state requires the information about :

- how the world evolves in-dependently from the agent, and
- how the agent actions affects the world



Goal based agents:

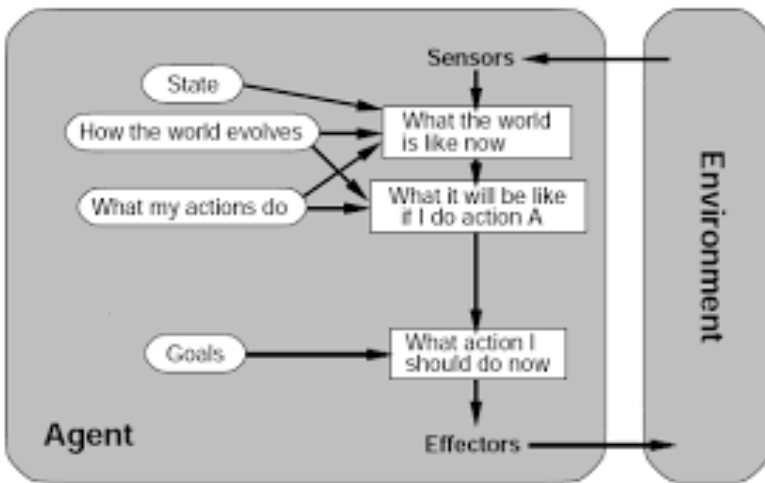
They choose their actions in order to achieve goals.

Goal-based approach is more flexible than reflex agent since the knowledge supporting a decision is explicitly modelled, thereby allowing for modifications. □

Goal – It is the description of desirable situations.

The goal-based agent's behaviour can easily be changed to go to a different destination, simply by specifying that destination as the goal.

Sometimes goal-based action selection is straightforward—for example, when goal satisfaction results immediately from a single action. Sometimes it will be more tricky—for example, when the agent has to consider long sequences of twists and turns in order to find a way to achieve the goal.



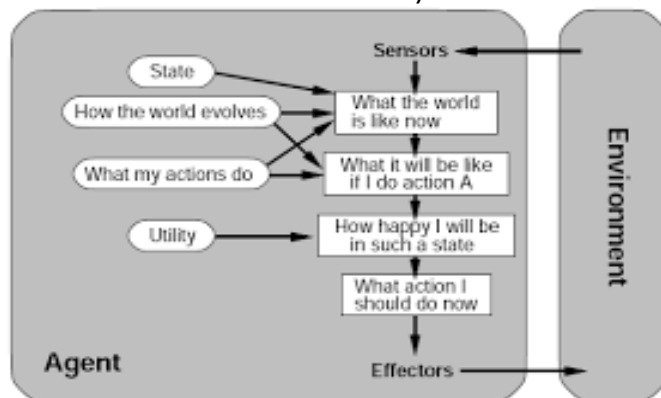
Utility based agents:

The agents which are developed having their end uses as building blocks are called utility based agents. They choose actions based on a preference (utility) for each state.

Sometimes achieving the desired goal is not enough. Agent happiness should be taken into consideration. Utility describes how “happy” the agent is. Because of the uncertainty in the world, a utility agent chooses the action that maximizes the expected utility.

A utility function maps a state onto a real number which describes the associated degree of happiness.

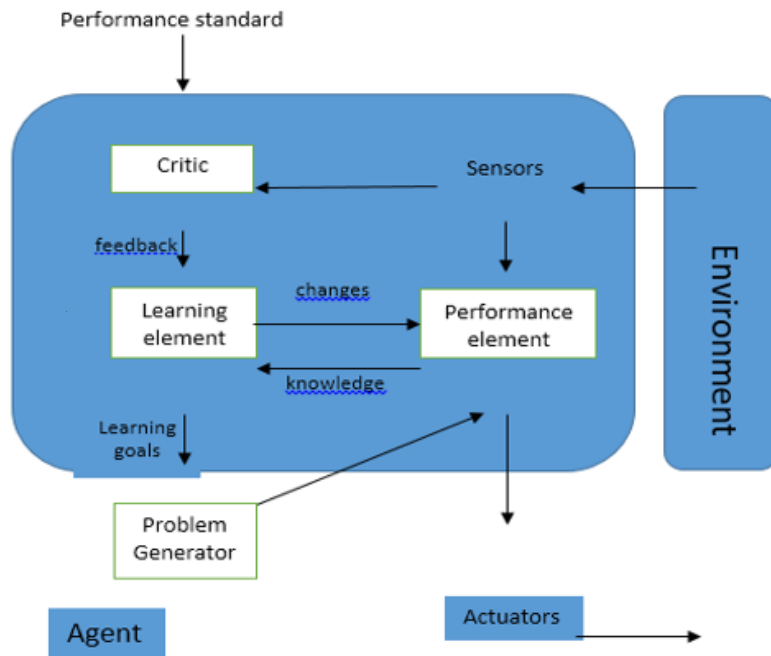
If the internal utility function and the external performance measure are in agreement, then an agent that chooses actions to maximize its utility will be rational according to the external performance measure.



Learning agent:

A learning agent can be divided into four conceptual components which are the basic building blocks of Learning Agent.

1. Performance element
2. Learning element
3. Critic
4. Problem generator



1. Performance element

- Responsible for choosing actions that are known to offer good outcomes
- Corresponds to the agents discussed earlier

2. Learning element

- Responsible for improving the performance element
- Requires feedback on how well the agent is doing

3. Critic

- Responsible for providing feedback
- Compares outcomes with some objective performance standard from outside the agent

4. Problem generator

- Responsible for generating new experience
- Requires exploration –trying unknown actions which may be sub-optimal

Example:

Consider a taxi driver agent

Performance element: you want to go into Mumbai CST? Let's take Highway, it's worked well previously

Problem generator: nah, let's try Freeway for a change –it may be better

Critic element: great, it was 15 minutes quicker, and what a nice view!

Learning element: yeah, in future we'll take Freeway

How the components of agent programs work

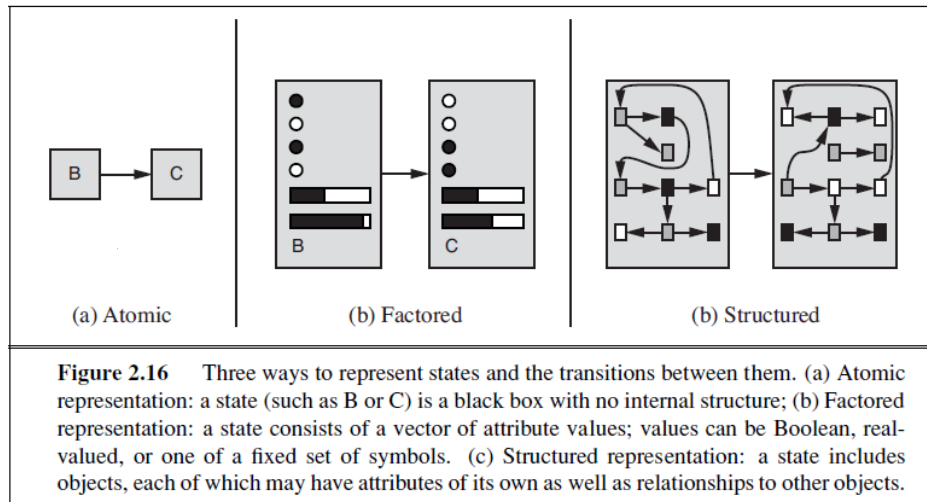


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

The components of agent program works in three ways i.e. atomic, factored, structured.

1. In an atomic representation, each state of the world is indivisible-it has no internal structure. Consider the problem of finding a driving route from one end of a country to the other via some sequence of cities. For the purposes of solving this problem, it may suffice to reduce the state of world to just the name of city we are in-a single atom of knowledge; the algorithm underlying search and game playing, Hidden Markov models and Markov decision processes all work with atomic representations or they treat representations as if they were atomic.
2. A factored representation splits up each state into a fixed set of variables or attributes, each of which can have a value. Many important areas of AI are based on factored representations, including constraint satisfaction algorithms, propositional logic, planning, Bayesian networks and the machine learning algorithms.
3. A structured representation, in which objects and their various and varying relationships can be described explicitly. Structured representations underlie relational databases and first order logic, first order probability models, knowledge based learning and natural language understanding.

Practice exercises

Graded Questions

- Q.1 What is AI? [Nov 2018, May 2019]
- Q.2 What are the categories of AI?
- Q.3 What is Turing Test? Explain [Nov 2018, May 2019]
- Q.4 Explain the foundations of AI
- Q.5 Write short note on history of AI
- Q.6 Write short note on State of Art
- Q.7 What are the application areas of AI? Explain [Nov 2018]
- Q.8 Define agent and environment [Nov 2018, May 2019]
- Q.9 Define agent and environment with respect to vacuum cleaner world
- Q.10 Write short note on concept of rationality. [Nov 2018, May 2019]
- Q.11 How do you specify the task environment? Explain
- Q.12 Explain the term PEAS
- Q.13 Explain PEAS description of task environment for automated taxi [Nov 2018, May 2019]
- Q.14 Explain the properties of task environments
- Q.15 Explain different types of task environments.
- Q.16 Give some examples of task environments with their characteristics
- Q.17 Give comparison between Full observable and partially observable agent [Nov 2018]
- Q.18 Explain the structure of Agents
- Q.19 Explain different types of agent programs
- Q.20 Explain Simple reflex agent [May 2019]
- Q.21 Explain model based reflex agents
- Q.22 Explain Goal based agents
- Q.23 Explain utility based agents
- Q.24 Explain learning agents
- Q.25 How the components of agent programs work?

Unit II

Solving Problems by Searching & Beyond Classical Search

2.1 Solving Problems by Searching

- 2.1.1 Problem solving agents
- 2.1.2 Example problems
- 2.1.3 Searching for solutions
- 2.1.4 Uninformed search
- 2.1.5 Informed search strategies
- 2.1.6 Heuristic functions

2.2 Beyond Classical Search

- 2.2.1 Local search algorithms
- 2.2.2 Searching with non-deterministic action
- 2.2.3 Searching with partial observations
- 2.2.4 Online search agents
- 2.2.5 Unknown environments

2.1 Solving Problems by Searching

Problem-solving agents use atomic representations—that is, states of the world are considered as wholes, with no internal structure visible to the problem solving algorithms. Goal-based agents that use more advanced factored or structured representations are usually called planning agents.

Our discussion of problem solving begins with precise definitions of problems and their solutions and gives several examples to illustrate these definitions. We then describe several general-purpose search algorithms that can be used to solve these problems. We will see several uninformed search algorithms—algorithms that are given no information about the problem other than its definition. Although some of these algorithms can solve any solvable problem, none of them can do so efficiently. Informed search algorithms, on the other hand, can do quite well given some guidance on where to look for solutions.

2.1.1 Problem solving agents

1. Intelligent agents are supposed to maximize their performance measure.
2. This can be simplified if the agent can adopt a goal and aim at satisfying it.
3. Goals help organize behaviour by limiting the objectives that the agent is trying to achieve.
4. Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.
5. Goal is a set of states. The agent's task is to find out which sequence of actions will get it to a goal state.
6. Problem formulation is the process of deciding what sorts of actions and states to consider, given a goal.
7. An agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence. Looking for such a sequence is called search.
8. A search algorithm takes a problem as input and returns a solution in the form of action sequence. Once a solution is found the actions it recommends can be carried out – execution phase “formulate, search, execute” design for the agent. After formulating a goal and a problem to solve; the agent calls a search procedure to solve it.
9. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do (typically the first action in the sequence). Then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal.

Well-defined problems and solutions

A problem can be defined formally by five components:

1. INITIAL STATE: The initial state that the agent starts in. For example, the initial state for our agent in Romania might be described as $ln(Arad)$.

```

function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
               state, some description of the current world state
               goal, a goal, initially null
               problem, a problem formulation

  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
    if seq = failure then return a null action
  action  $\leftarrow$  FIRST(seq)
  seq  $\leftarrow$  REST(seq)
  return action

```

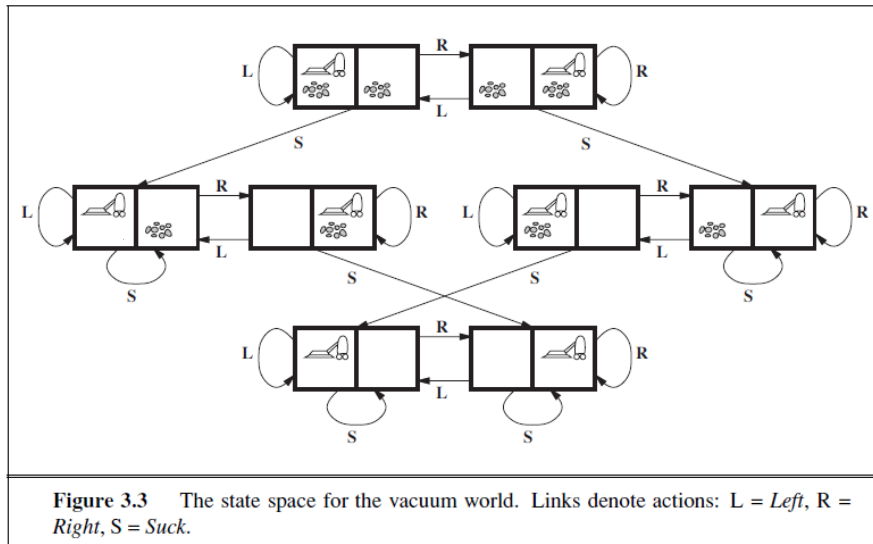
Figure: A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

2. **ACTIONS:** A description of the possible actions available to the agent. Given a particular states, ACTIONS(*s*) returns the set of actions that can be executed in *s*. We say that each of these actions is applicable in *s*. For example, from the state *In(Arad)*, the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}.
3. **TRANSITION MODEL:** A description of what each action does; the formal name for this is the transition model, specified by a function RESULT(*s*, *a*) that returns the state that results from doing action *a* in states. We also use the term successor to refer to any state reachable from a given state by a single action. For example, we have RESULT(*In(Arad)*, Go(Zerind)) = *In(Zerind)* . Together, the initial state, actions, and transition model implicitly define the state space of the problem—the set of all states reachable from the initial state by any sequence of actions. The state space forms a directed network or graph in which the nodes are states and the links between nodes are actions. A path in the state space is a sequence of states connected by a sequence of actions.
4. **GOAL STATE:** The goal test, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the singleton set {*In(Bucharest)*}.
5. **PATH COST:** A path cost function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometres. In this chapter, we assume that the cost of a path can be described as the sum of the costs of the individual actions along the path. The step cost of taking action (*a*) in state (*s*) to reach state (*s'*) is denoted by $c(s, a, s')$.

The preceding elements define a problem and can be gathered into a single data structure that is given as input to a problem-solving algorithm. A solution to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.

2.1.2 Example problems

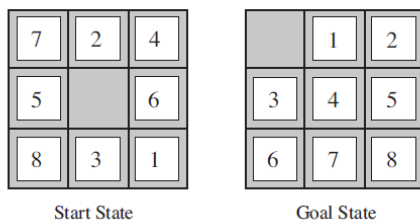
Toy problems



The problem is formulated as:

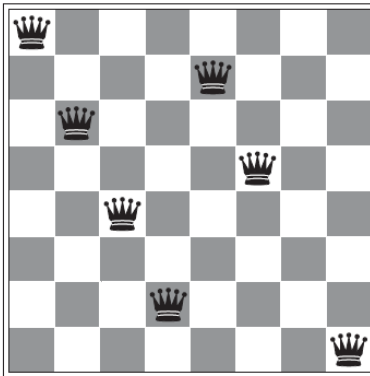
1. States: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are 2×2^2 possible world states. A larger environment with n locations has $n \cdot 2^n$ states.
2. Initial state: Any state can be designated as the initial state.
3. Actions: In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.
4. Transition model: The actions have their expected effects, except that moving Left in the leftmost square, moving right in the rightmost square, and Sucking in a clean square have no effect. The complete state space is shown in the above figure.
5. Goal test: This checks whether all the squares are clean.
6. Path cost: Each step costs 1, so the path cost is the number of steps in the path.

8 - puzzle problem



1. States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
2. Initial state: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
3. Actions: The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.
4. Transition model: Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure, the resulting state has the 5 and the blank switched.
5. Goal test: This checks whether the state matches the goal configuration shown in Figure. (Other goal configurations are possible.)
6. Path cost: Each step costs 1, so the path cost is the number of steps in the path.

8 - queens problem



An incremental formulation involves operators that augment the state INCREMENTAL FORMULATION description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state. A complete-state formulation starts with all 8 queens on COMPLETE-STATE FORMULATION the board and moves them around. In either case, the path cost is of no interest because only the final state counts. The first incremental formulation one might try is the following:

1. States: Any arrangement of 0 to 8 queens on the board is a state.
2. Initial state: No queens on the board.
3. Actions: Add a queen to any empty square.
4. Transition model: Returns the board with a queen added to the specified square.
5. Goal test: 8 queens are on the board, none attacked.

In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate. A better formulation would prohibit placing a queen in any square that is already attacked:

- States: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.
- Actions: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

Real world problems

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

1. States: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these “historical” aspects.
 2. Initial state: This is specified by the user’s query.
 3. Actions: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
 4. Transition model: The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time.
 5. Goal test: Are we at the final destination specified by the user?
 6. Path cost: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.
- **Touring problems** are closely related to route-finding problems, but with an important difference. Consider, for example, the problem “Visit every city in Figure 3.2 at least once, starting and ending in Bucharest.” As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the set of cities the agent has visited.
 - The **traveling salesperson problem** (TSP) is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.
 - A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase and is usually split into two parts: cell layout and channel

routing. In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

- **Robot navigation** is a generalization of the route-finding problem described earlier. Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite.
- **Automatic assembly sequencing** of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972). Progress since then has been slow but sure, to the point where the assembly of intricate objects such as electric motors is economically feasible. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the generation of legal actions is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space.
- Another important assembly problem is **protein design**, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

2.1.3 Searching for solutions

A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem.

The set of all leaf nodes available for expansion at any given point is called the frontier. (Many authors call it the open list, which is both geographically less evocative and less accurate, because other data structures are better, suited than a list.)

The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand. Search algorithms all share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called search strategy. If a repeated state gets generated in the search tree, it is called a loopy path. Considering such loopy paths means that the complete search tree is infinite because there is no limit to how often one can traverse a loop.

Loopy paths are a special case of the more general concept of redundant paths, which exist whenever there is more than one way to get from one state to another.

If you are concerned about reaching the goal, there's never any reason to keep more than one path to any given state, because any goal state that is reachable by extending one path is also reachable by extending the other.

In some cases, it is possible to define the problem itself so as to eliminate redundant paths. For example, if we formulate the 8-queens problem so that a queen can be placed in any column, then each state with n queens can be reached by $n!$ different paths; but if we reformulate the problem so that each new queen is placed in the leftmost empty column, then each state can be reached only through one path.

```

function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

```

```

function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
    only if not in the frontier or explored set

```

Figure: An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed.

For each node n of the tree, we have a structure that contains four components:

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

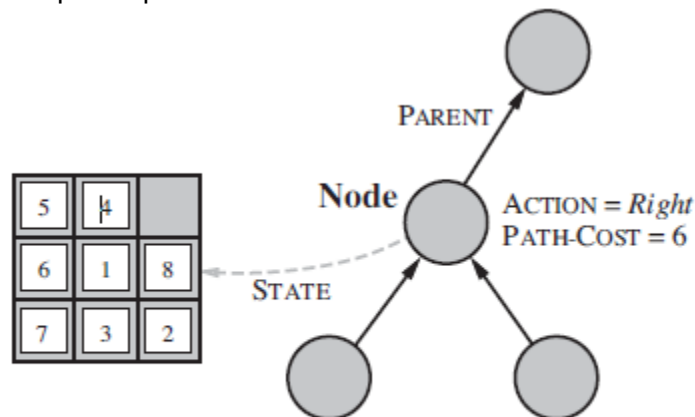


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD-NODE takes a parent node and an action and returns the resulting child node:

```

function CHILD-NODE(problem, parent, action) returns a node
  return a node with
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)

```

The appropriate data structure for this is a queue. The operations on a queue are as follows:

- **EMPTY?(queue)** returns true only if there are no more elements in the queue.
- **POP(queue)** removes the first element of the queue and returns it.
- **INSERT(element, queue)** inserts an element and returns the resulting queue.

Three common variants are:

- the first-in, first-out or FIFO queue, which pops the oldest element of the queue;
- the last-in, first-out or LIFO queue (also known as a stack), which pops the newest element of the queue; and
- the priority queue, which pops the element of the queue with the highest priority according to some ordering function.

Measuring problem-solving performance

We can evaluate an algorithm's performance in four ways:

1. **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
2. **Optimality:** Does the strategy find the optimal solution,
3. **Time complexity:** How long does it take to find a solution?
4. **Space complexity:** How much memory is needed to perform the search?

Time and space complexity are always considered with respect to some measure of the problem difficulty.

In AI, the graph is often represented implicitly by the initial state, actions, and transition model and is frequently infinite. For these reasons, complexity is expressed in terms of three quantities:

1. b , the branching factor or maximum number of successors of any node;
2. d , the depth of the shallowest goal node (i.e., the number of steps along the path from the root);
3. m , the maximum length of any path in the state space.

2.1.4 Uninformed search

The term means that the strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the order in which nodes are expanded. Strategies that know whether one non-goal state is "more promising" than another are called informed search or heuristic SEARCH strategies.

1. **Breadth-first search:** Is an instance of the general graph-search algorithm in which the shallowest unexpanded node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is generated rather than when it is selected for expansion. The general template for graph search, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found. Thus, breadth-first search always has the shallowest path to every node on the frontier.

```

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

```

Figure: Breadth-first search on a graph.

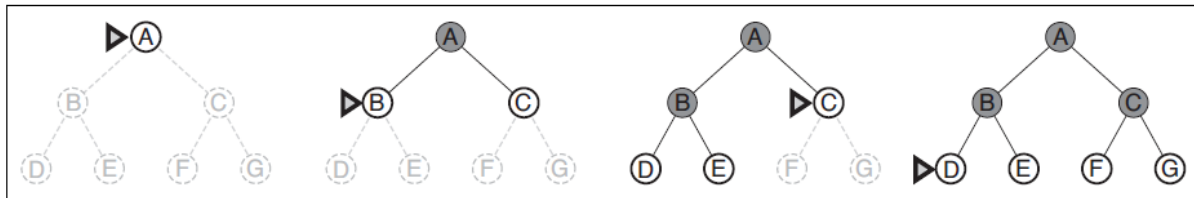


Figure: Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

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

Figure: Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

2. Uniform cost: It does not care about the number of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions—for example, a sequence of NoOp actions. Completeness is guaranteed provided the cost of every step exceeds some small positive constant. Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d . Instead, let C^* be the cost of the optimal solution, and assume that every action cost at least. Then the algorithm's worst-case time and space complexity is $O(b(1+C^*/\epsilon))$, which can be much greater than bd . This is because uniform cost search can explore large trees of small steps before exploring paths involving large and perhaps useful steps. When all step costs are equal, $b(1+C^*/\epsilon)$ is just $bd+1$. When all step costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost; thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily.

```

function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier  $\leftarrow$  a priority queue ordered by PATH-COST, with node as the only element
  explored  $\leftarrow$  an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node  $\leftarrow$  POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier  $\leftarrow$  INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child

```

Figure: Uniform-cost search on a graph. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

3. Depth-first search: Depth-first search always expands the deepest node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors. The depth-first search algorithm is an instance of the graph-search algorithm whereas breadth-first-search uses a FIFO queue, depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected. As an alternative to the GRAPH-SEARCH-style implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.

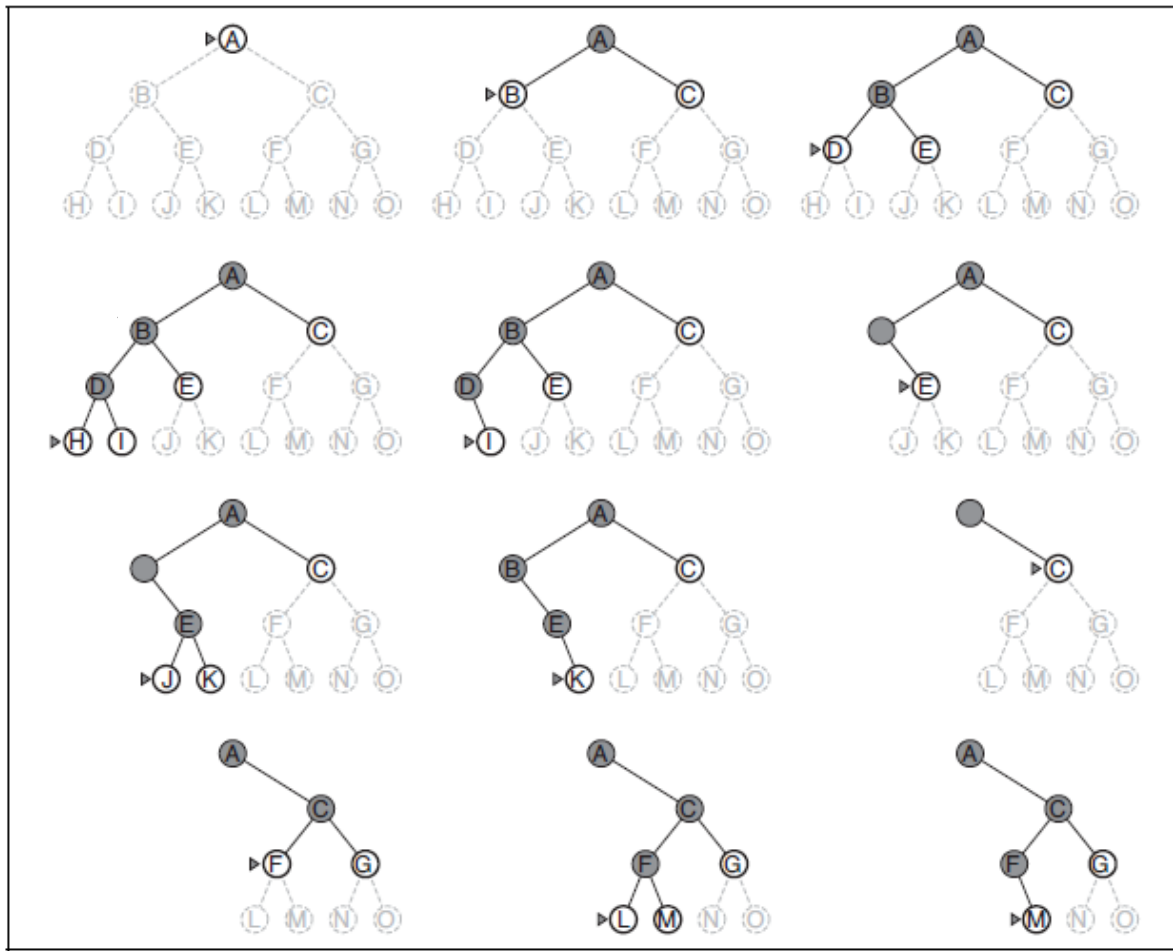


Figure: Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

4. Depth-limited search:

The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit. That is, nodes at depth are treated as if they have no successors. This approach is called depth-limited search. The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose $< d$, that is, the shallowest goal is beyond the depth limit. (This is likely when d is unknown.) Depth-limited search will also be non-optimal if we choose $> d$. Its time complexity is $O(b^d)$ and its space complexity is $O(b^d)$. Depth-first search can be viewed as a special case of depth-limited search with $d = \infty$.

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution, or failure/cutoff
return RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff

if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

else if *limit* = 0 **then return** cutoff

else

cutoff_occurred? $\leftarrow$ false

for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

child $\leftarrow$ CHILD-NODE(*problem*, *node*, *action*)

result $\leftarrow$ RECURSIVE-DLS(*child*, *problem*, *limit* - 1)

if *result* = cutoff **then** *cutoff_occurred?* $\leftarrow$ true

else if *result* $\neq$ failure **then return** *result*

if *cutoff_occurred?* **then return** cutoff **else return** failure

Figure: A recursive implementation of depth-limited tree search.

Depth-limited search can be implemented as a simple modification to the general tree or graph-search algorithm. Alternatively, it can be implemented as a simple recursive algorithm. Notice that depth-limited search can terminate with two kinds of failure: the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit.

5. Iterative deepening search (or iterative deepening depth-first search): It is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found. This will occur when the depth limit reaches d , the depth of the shallowest goal node. Iterative deepening combines the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest: $O(bd)$ to be precise. Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a non-decreasing function of the depth of the node. Figure below shows four iterations on a binary search tree, where the solution is found on the fourth iteration.

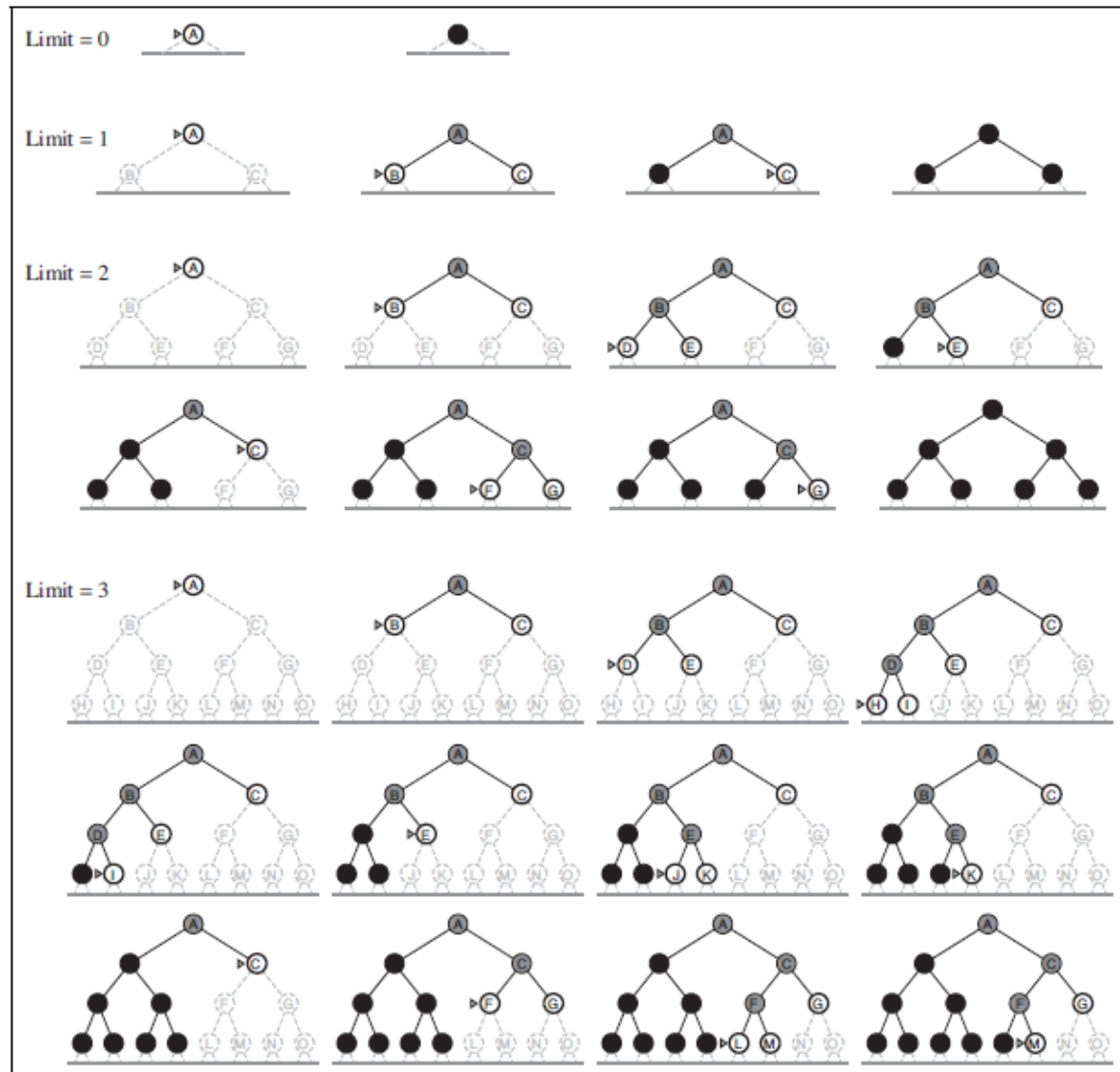


Figure: Four iterations of iterative deepening search on a binary tree.

```

function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  for depth = 0 to  $\infty$  do
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)
    if result  $\neq$  cutoff then return result

```

Figure: The iterative deepening search algorithm, which repeatedly applies depth limited search with increasing limits. It terminates when a solution is found or if the depth limited search returns failure, meaning that no solution exists.

There is some extra cost for generating the upper levels multiple times, but it is not large. For example, if $b = 10$ and $d = 5$, the numbers are

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110.$$

If you are really concerned about repeating the repetition, you can use a hybrid approach that runs breadth-first search until almost all the available memory is consumed, and then runs iterative deepening from all the nodes in the frontier. In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

Iterative deepening search is analogous to breadth-first search in that it explores a complete layer of new nodes at each iteration before going on to the next layer. It would seem worthwhile to develop an iterative analog to uniform-cost search, inheriting the latter algorithm's optimality guarantees while avoiding its memory requirements.

6. Bidirectional search

The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle (Figure below). The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d , or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect; if they do, a solution has been found.

(It is important to realize that the first such solution found may not be optimal, even if the two searches are both breadth-first; some additional search is required to make sure there isn't another short-cut across the gap.) The check can be done when each node is generated or selected for expansion and, with a hash table, will take constant time. For example, if a problem has solution depth $d=6$, and each direction runs breadth-first search one node at a time, then in the worst case the two searches meet when they have generated all of the nodes at depth 3. For $b=10$, this means a total of 2,220 node generations, compared with 1,111,110 for a standard breadth-first search. Thus, the time complexity of bidirectional search using breadth-first searches in both directions is $O(b^{d/2})$. The space complexity is also $O(b^{d/2})$.

We can reduce this by roughly half if one of the two searches is done by iterative deepening, but at least one of the frontiers must be kept in memory so that the intersection check can be done. This space requirement is the most significant weakness of bidirectional search.

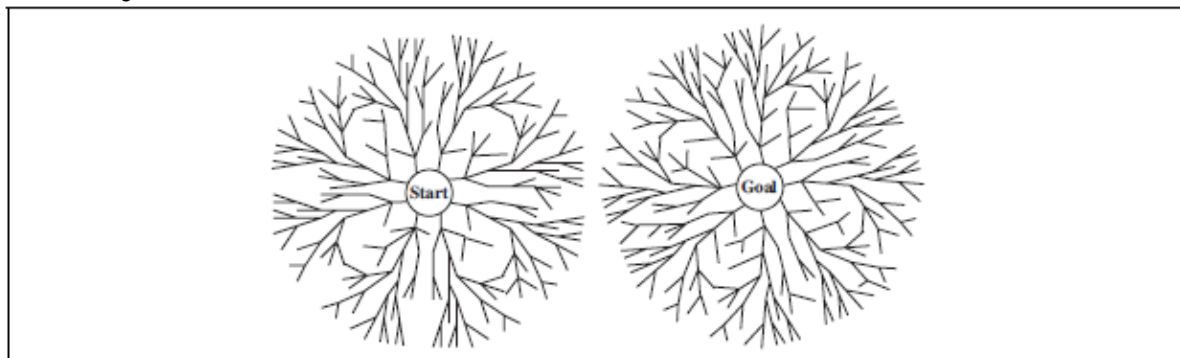


Figure: A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

Comparing uninformed search strategies

For graph searches, the main differences are that depth-first search is complete for finite state spaces and that the space and time complexities are bounded by the size of the state space.

Criteria	Breadth First	Uniform cost	Depth First	Depth Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes ^a	Yes ^{a,b}	No	No	Yes ^a	Yes ^{a,d}

Time	$O(b^d)$	$O(b^1 + \lceil C * \epsilon \rceil)$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^1 + \lceil C * \epsilon \rceil)$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Optimal?	Yes ^c	No	No	No	Yes ^c	Yes ^{c,d}

Table: Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^a complete if b is finite; ^b complete if step costs $\geq \epsilon$ for positive ϵ ; ^c optimal if step costs are all identical; ^d if both directions use breadth-first search.

2.1.5 Informed search strategies

1. Informed search strategy is the one that uses problem specific knowledge beyond the definition of the problem itself can find solutions more efficiently than can an uninformed strategy.
2. The general approach we consider is called best-first search. Best-first search is an instance of the general tree search or graph search algorithm in which a node is selected for expansion based on an evaluation function, $f(n)$.
3. The evaluation function is a cost estimate, so the node with the lowest evaluation is expanded first. The implementation of best-first graph search is identical to that for uniform-cost search. The choice of f determines the search strategy.
4. Most best-first algorithms include as a component of f a heuristic function, denoted $h(n)$:
 $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.
5. Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. For now, we consider them to be arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then $h(n) = 0$.

Types of Informed or heuristic search strategies:

(a) Greedy best-first search- Greedy best-first search tries to expand the node that is closest to the goal, on the ground that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$.

Greedy Best First Search is a basic search that uses path cost to determine the next node to expand. Greedy best-first search expands the node that appears to be closest to goal

Properties of Greedy best first search are as follows:

(a) Space $O(b^m)$ -- keeps all nodes in memory.

(b) Optimal? No.

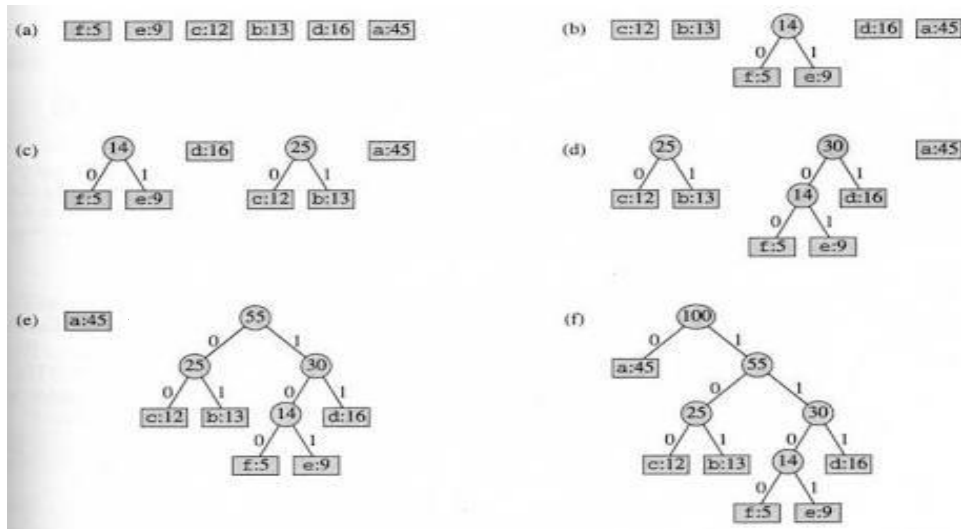
(c) Complete? No – can get stuck in loops, e.g., Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

(d) Time? $O(b^m)$, but a good heuristic can give dramatic improvement.

Evaluation function $h(n)$ (heuristic)

- $h(n)$ = estimate of cost from n to goal
- The better the estimate the better the solution
- e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest

The worst-case time and space complexity for the tree version is $O(b^m)$, where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially. The amount of the reduction depends on the particular problem and on the quality of the heuristic.



(b) A* search: Minimizing the total estimated solution cost. The most widely known form of best-first search is called A* search (pronounced "A-star SEARCH search"). It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal: $f(n)=g(n)+h(n)$. Since $g(n)$ gives the path cost from the start node to node n , and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have $f(n)$ = estimated cost of the cheapest solution through n . A* search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A* uses $g + h$ instead of g .

(a) When $h(n)$ = actual cost to goal

- Only nodes in the correct path are expanded
- Optimal solution is found

(b) When $h(n)$ < actual cost to goal

- Additional nodes are expanded
- Optimal solution is found

(c) When $h(n)$ > actual cost to goal

- Optimal solution can be overlooked

A* is optimal if it uses an **admissible heuristic**

- $h(n) \leq h^*(n)$ the true cost from node n
- if $h(n)$ never overestimates the cost to reach the goal

• Example

- h_{SLD} never overestimates the actual road distance

A* expands nodes in increasing f value

- a. Gradually adds f -contours of nodes (like breadth-first search adding layers)
- b. Contour i has all nodes $f=f_i$ where $f_i < f_{i+1}$

Properties:

- c. Complete- Yes, unless there are infinitely many nodes with $f \leq f(G)$
- d. Time- Exponential in [relative error of $h \times$ length of soln]. The better the heuristic, the better the time
 - Best case h is perfect, $O(d)$
 - Worst case $h = 0$, $O(b^d)$ same as BFS
- e. Space
 - Keeps all nodes in memory and save in case of repetition
 - This is $O(b^d)$ or worse
 - A* usually runs out of space before it runs out of time
- f. Optimal
 - Yes, cannot expand f_{i+1} unless f_i is finished

Conditions for optimality:

(a) Admissibility and consistency

Admissibility:

- A heuristic $h(n)$ is admissible if for every node n , $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n .

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If $h(n)$ is admissible, A* using TREE-SEARCH is optimal.

Consistency:

- A heuristic is consistent if for every node n , every successor n' of n generated by any action a , $h(n) \leq c(n, a, n') + h(n')$
- If h is consistent, we have

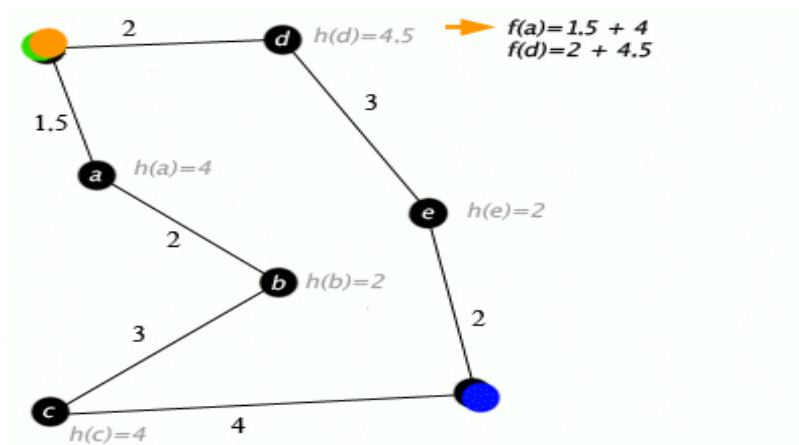
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$
- i.e., $f(n)$ is non-decreasing along any path.
- Theorem: If $h(n)$ is consistent, A* using GRAPH-SEARCH is optimal

(b) Optimality of A*

The tree-search version of A* is optimal if $h(n)$ is admissible, while the graph-search version is optimal if $h(n)$ is consistent. The argument essentially mirrors the argument for the optimality of uniform-cost search, with g replaced by f —just as in the A* algorithm itself.

The first step is to establish the following: if $h(n)$ is consistent, then the values of $f(n)$ along any path are non-decreasing. The proof follows directly from the definition of consistency. Suppose n is a successor of n' ; then $g(n) = g(n') + c(n', a, n)$ for some action a , and we have $f(n) = g(n) + h(n) = g(n') + c(n', a, n) + h(n) \geq g(n') + h(n') = f(n')$.

A* search is complete, optimal, and optimally efficient among all such algorithms is rather satisfying. The number of states within the goal contour search space is still exponential in the length of the solution. The details of the analysis are beyond the scope of this book, but the basic results are as follows. For problems with constant step costs, the growth in run time as a function of the optimal solution depth d is analyzed in terms of the absolute error or the relative error of the heuristic. The absolute error is defined as $\Delta \equiv h^* - h$, where h^* is the actual cost of getting from the root to the goal, and the relative error is defined as $\epsilon \equiv (h^* - h)/h^*$.



Source:

<https://www.youtube.com/watch?v=iKDHfva1Cf8&list=PL90VWpg9kppNhy3esMHFy4KjBGQpzofPI&index=21&t=0s>

c) Memory-bounded heuristic search: The simplest way to reduce memory requirements for A* is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A* (IDA*) algorithm. The main difference between IDA* and standard iterative deepening is that the cut off used is the f-cost ($g + h$) rather than the depth; at each iteration, the cut off value is the smallest f-cost of any node that exceeded the cut off on the previous iteration. IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes. Unfortunately, it suffers from the same difficulties with real-valued costs as does the iterative version of uniform-cost search.

Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space. Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the f limit variable to keep track of the f-value of the best alternative path available from any ancestor of the current node.

2. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up value—the best f-value of its children.

3. In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time. RBFS is somewhat more efficient than IDA*, but still suffers from excessive node re-generation.

4. RBFS follows the path then “changes its mind” and tries and then changes its mind back again. These mind changes occur because every time the current best path is extended, its f-value is likely to increase—h is usually less optimistic for nodes closer to the goal.

5. When this happens, the second-best path might become the best path, so the search has to backtrack to follow it. Each mind change corresponds to an iteration of IDA* and could require many re-expansions of forgotten nodes to recreate the best path and extend it one more node.

6. Like A* tree search, RBFS is an optimal algorithm if the heuristic function $h(n)$ is admissible. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded.

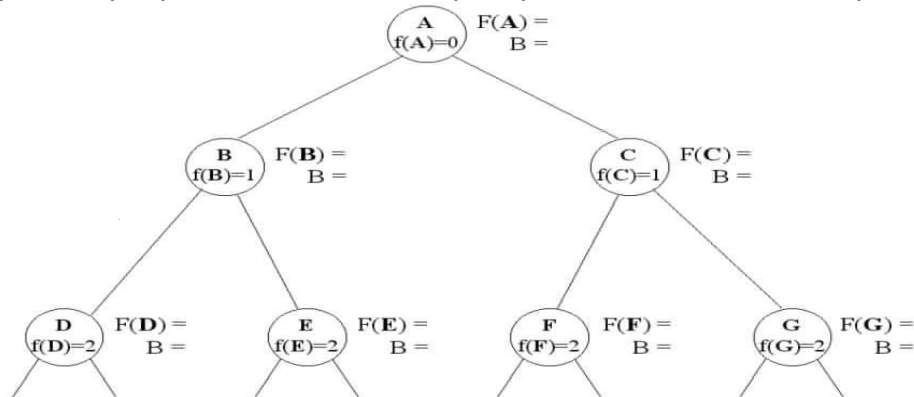
```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE),  $\infty$ )

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    successors  $\leftarrow$  []
    for each action in problem.ACTIONS(node.STATE) do
        add CHILD-NODE(problem, node, action) into successors
    if successors is empty then return failure,  $\infty$ 
    for each s in successors do /* update f with value from previous search, if any */
        s.f  $\leftarrow$  max(s.g + s.h, node.f)
    loop do
        best  $\leftarrow$  the lowest f-value node in successors
        if best.f > f_limit then return failure, best.f
        alternative  $\leftarrow$  the second-lowest f-value among successors
        result, best.f  $\leftarrow$  RBFS(problem, best, min(f_limit, alternative))
        if result  $\neq$  failure then return result
```

Figure: The algorithm for recursive best-first search

7. IDA* and RBFS suffer from using too little memory. Between iterations, IDA* retains only a single number: the current f-cost limit. RBFS retains more information in memory, but it uses only linear space: even if more

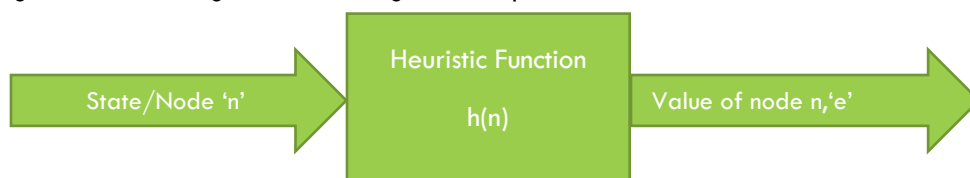
memory were available, RBFS has no way to make use of it. Because they forget most of what they have done, both algorithms may end up re-expanding the same states many times over. Furthermore, they suffer the potentially exponential increase in complexity associated with redundant paths in graphs.



(d) Learning to search better: The method rests on an important concept called the Meta level state space. Each state in a state space captures the internal (computational) state of a program that is searching in an object-level state space such as Romania. For example, the internal state of the A* algorithm consists of the current search tree. Each action in the meta level state space is a computation step that alters the internal state; for example, each computation step in A* expands a leaf node and adds its successors to the tree.

2.1.6 Heuristic functions

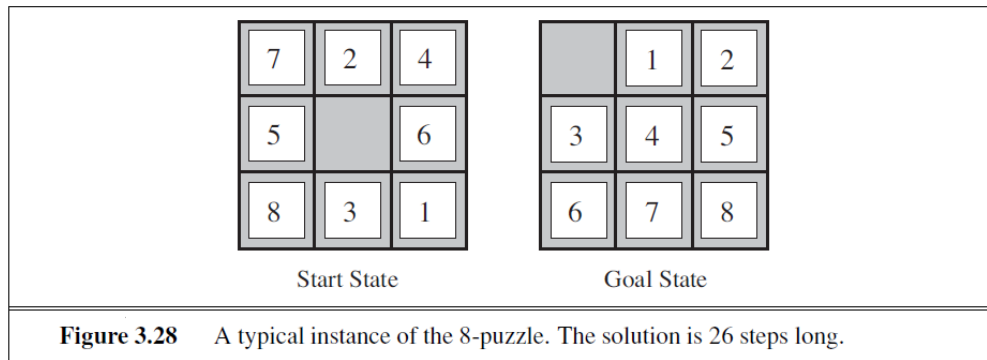
- A heuristic function is an evaluation function, to which the search state is given as input and it generates the tangible representation of the state as output.
- It maps the problem state description to measures of desirability, usually represented as number weights. The value of a heuristic function at a given node in the search process gives a good estimate of that node being on the desired path to solution.
- It evaluates individual problem state and determines how much promising the state is.
- Heuristic functions are the most common way of imparting additional knowledge of the problem states to the search algorithm. Below figure shows the general representation of heuristic function.



- The representation may be the approximate cost of path from the goal node or number of hops required to reach the goal node, etc.
- The heuristic function that we are considering in this syllabus, for a node n is, $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.
- Example: For the Travelling Salesman Problem, the sum of the distances travelled so far can be a simple heuristic function.
- Heuristic function can be of two types depending on the problem domain. It can be a Maximization Function or Function Minimization of the path cost.

If we want to find the shortest solutions by using A*, we need a heuristic function that never overestimates the number of steps to the goal. There is a long history of such heuristics for the 15-puzzle; here are two commonly used candidates:

- h_1 = the number of misplaced tiles. For Figure below, all of the eight tiles are out of position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristic because it is clear that any tile that is out of place must be moved at least once.



• h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the city block distance or Manhattan distance. h_2 is also admissible because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

$$h_2 = 3+1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$$

As expected, neither of these overestimates the true solution cost, which is 26.

d	Search Cost (nodes generated)			Effective Branching Factor		
	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	—	539	113	—	1.44	1.23
16	—	1301	211	—	1.45	1.25
18	—	3056	363	—	1.46	1.26
20	—	7276	676	—	1.47	1.27
22	—	18094	1219	—	1.48	1.28
24	—	39135	1641	—	1.48	1.26

Figure: Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A^* algorithms with h_1 , h_2 . Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths d .

Generating admissible heuristics from relaxed problems

h_1 and h_2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for simplified versions of the puzzle. If the rules of the puzzle were changed so that a tile could move anywhere instead of just to the adjacent empty square, then h_1 would give the exact number of steps in the shortest solution. Similarly, if a tile could move one square in any direction, even onto an occupied square, then h_2 would give the exact number of steps in the shortest solution. A problem with fewer restrictions on the actions is called a relaxed problem. The state-space graph of the relaxed problem is a supergraph of the original state space because the removal of restrictions creates added edges in the graph.

Because the relaxed problem adds edges to the state space, any optimal solution in the original problem is, by definition, also a solution in the relaxed problem; but the relaxed problem may have better solutions if the added edges provide short cuts. Hence, the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.

Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore consistent. If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically. For example, if the 8-puzzle actions are described as A tile can move from square A to square B if A is horizontally or vertically adjacent to B and B is blank, we can generate three relaxed problems by removing one or both of the conditions:

- A tile can move from square A to square B if A is adjacent to B.
- A tile can move from square A to square B if B is blank.
- A tile can move from square A to square B.

From (a), we can derive h_2 (Manhattan distance). The reasoning is that h_2 would be the proper score if we moved each tile in turn to its destination. The heuristic is derived from (b). From (c), we can derive h_1 (misplaced tiles) because it would be the proper score if tiles could move to their intended destination in one step. Notice that it is crucial that the relaxed problems generated by this technique can be solved essentially without search, because the relaxed rules allow the problem to be decomposed into eight independent subproblems. If the relaxed problem is hard to solve, then the values of the corresponding heuristic will be expensive to obtain.

A program called ABSOLVER can generate heuristics automatically from problem definitions, using the “relaxed problem” method and various other techniques (Prieditis, 1993). ABSOLVER generated a new heuristic for the 8-puzzle that was better than any preexisting heuristic and found the first useful heuristic for the famous Rubik’s Cube puzzle.

2.2 Beyond Classical Search

We will cover algorithms that perform local search in the state space, evaluate and modify one or more current states rather than systematically exploring paths from an initial state.

These algorithms are suitable for problems in which all that matters is the solution state, not the path cost to reach it. The family of local search algorithms includes methods inspired by statistical physics (simulated annealing) and evolutionary biology (genetic algorithms).

The key idea is that if an agent cannot predict exactly what percept it will receive, then it will need to consider what to do under each contingency that its percepts may reveal. With partial observability, the agent will also need to keep track of the states it might be in.

2.2.1 Local search algorithms

Local search algorithms operate using a single current node (rather than multiple paths) and generally move only to neighbors of that node. Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages: (1) they use very little memory—usually a constant amount; and (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

In addition to finding goals, local search algorithms are useful for solving pure optimization problems, in which the aim is to find the best state according to an objective function. For example, nature provides an objective function—reproductive fitness—that Darwinian evolution could be seen as attempting to optimize, but there is no “goal test” and no “path cost” for this problem. To understand local search, we find it useful to consider the state-space landscape. A landscape has both “location” (defined by the state) and “elevation” (defined by the value of the heuristic cost function or objective function). If elevation corresponds to cost, then the aim is to find the lowest valley—a global minimum; if elevation corresponds to an objective function, then the aim is to find the highest peak—a global maximum. (You can convert from one to the other just by inserting a minus sign.) Local search algorithms explore this landscape. A complete local search algorithm always finds a goal if one exists; an optimal algorithm always finds a global minimum/maximum.

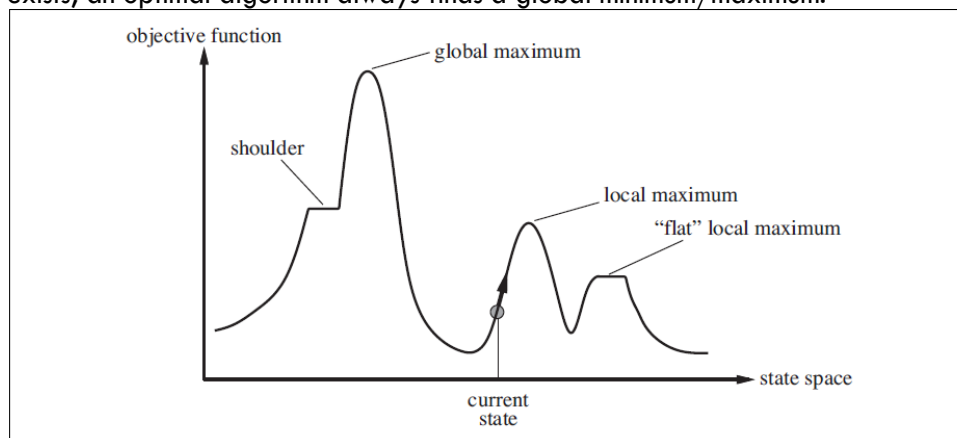


Figure: A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill Climbing Search

Hill climbing is simply a combination of depth first with generate and test where a feedback is used here to decide on the direction of motion in the search paper.

Hill climbing technique is used widely in artificial intelligence, to solve solving computationally hard problems, which has multiple possible solutions.

In the depth first search, the test function will merely accept or reject a solution. But in hill climbing the test function is provided with a heuristic function which provides an estimate of how close a given state is to goal state.

In Hill climbing, each state is provided with the additional information needed to find the solution. i.e. the heuristic value. The algorithm is memory efficient since it does not maintain a complete search tree. Rather, it looks only at the current state and immediate level states.

For example, if you want to find a mall from your current location. There are n possible paths with different directions to reach to the mall. The heuristic function will just give you the distance of each path which is reaching to the mall, so that it becomes very simple and time efficient for you to reach to the mall.

Hill climbing always attempts to make changes that improve the current state. In other words, hill climbing can only advance if there is a higher point in the adjacent landscape.

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

current $\leftarrow$ MAKE-NODE(*problem*.INITIAL-STATE)

loop do

neighbor $\leftarrow$ a highest-valued successor of *current*

if *neighbor*.VALUE $\leq$ *current*.VALUE **then return** *current*.STATE

current $\leftarrow$ *neighbor*

Figure: The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h .

Unfortunately, hill climbing often gets stuck for the following reasons:

- Local maxima: a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upward toward the peak but will then be stuck with nowhere else to go. Figure below illustrates the problem schematically. More concretely, the state in Figure.

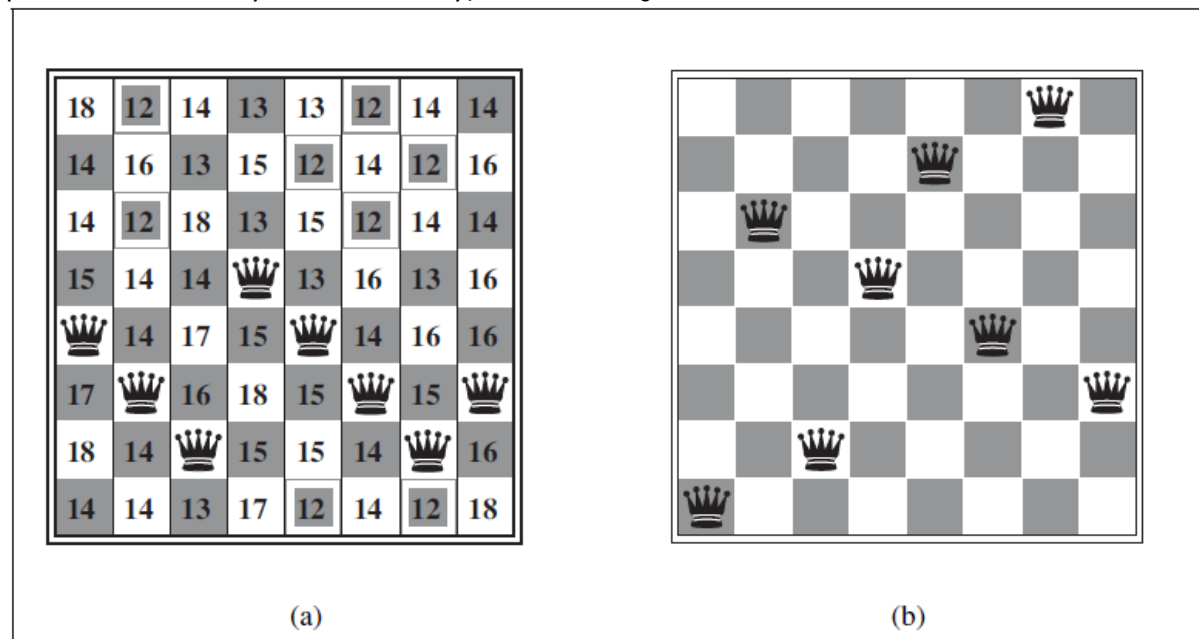
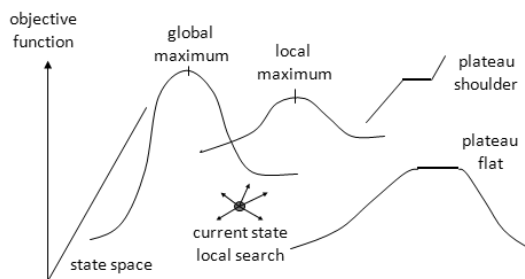


Figure: (a) An 8-queens state with heuristic cost estimate $h=17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h=1$ but every successor has a higher cost.

- Ridges: A ridge is shown in Figure below. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.



- **Plateau:** A plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible. A hill-climbing search might get lost on the plateau.



Variants of Hill Climbing search

Many variants of hill climbing have been invented.

- **Stochastic hill climbing:** Stochastic hill climbing chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions.
- **First-choice hill climbing:** First-choice hill climbing implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.
The hill-climbing algorithms described so far are incomplete—they often fail to find a goal when one exists because they can get stuck on local maxima.
- **Random-restart hill climbing:** Random-restart hill climbing adopts the well-known adage, “If at first you don’t succeed, try, try again.” It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found. It is trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state.

Simulated Annealing

A hill-climbing algorithm that never makes “downhill” moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness. Simulated annealing is such an algorithm. In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state. To explain simulated annealing, we switch our point of view from hill climbing to gradient descent (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high

temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature). The innermost loop of the simulated-annealing algorithm is quite similar to hill climbing. Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the “badness” of the move—the amount ΔE by which the evaluation is worsened. The probability also decreases as the “temperature” T goes down: “bad” moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases. If the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1. Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

```

function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
           schedule, a mapping from time to “temperature”

  current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
  for  $t = 1$  to  $\infty$  do
     $T \leftarrow$  schedule( $t$ )
    if  $T = 0$  then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow$  next.VALUE – current.VALUE
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$ 

```

Local Beam search

Keeping just one node in memory

might seem to be an extreme reaction to the problem of memory limitations. The local beam search algorithm³ keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If anyone is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.

At first sight, a local beam search with k states might seem to be nothing more than running k random restarts in parallel instead of in sequence. In fact, the two algorithms are quite different. In a random-restart search, each search process runs independently of the others. In a local beam search, useful information is passed among the parallel search threads. In effect, the states that generate the best successors say to the others, “Come over here, the grass is greener!” The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made. In its simplest form, local beam search can suffer from a lack of diversity among the k states—they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing. A variant called stochastic beam search, analogous to stochastic hill climbing, helps alleviate this problem. Instead of choosing the best k from the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value. Stochastic beam search bears some resemblance to the process of natural selection, whereby the “successors” (offspring) of a “state” (organism) populate the next generation according to its “value” (fitness).

Genetic algorithms

A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state. The analogy to natural selection is the same as in stochastic beam search, except that now we are dealing with sexual rather than asexual reproduction. Like beam searches, GAs begin with a set of k randomly generated states, called the population. Each state, or individual, is represented as a string over a finite alphabet most commonly, a string of 0s and 1s.

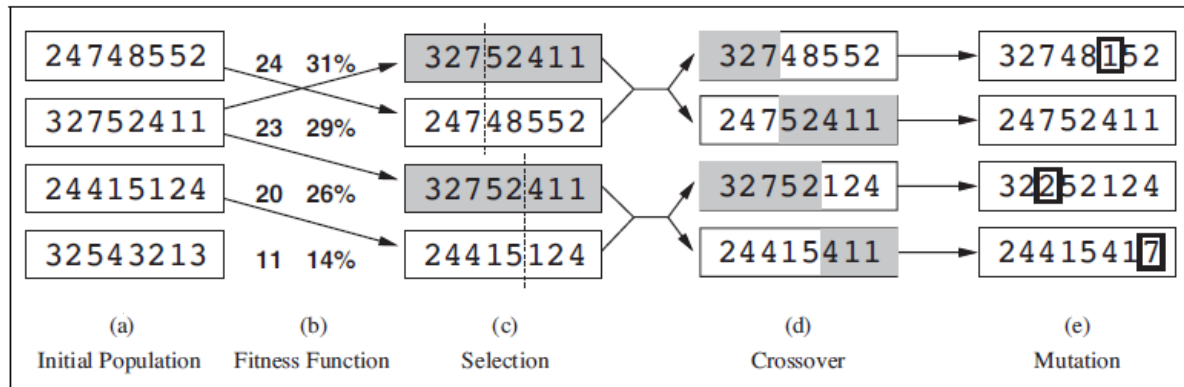


Figure: The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

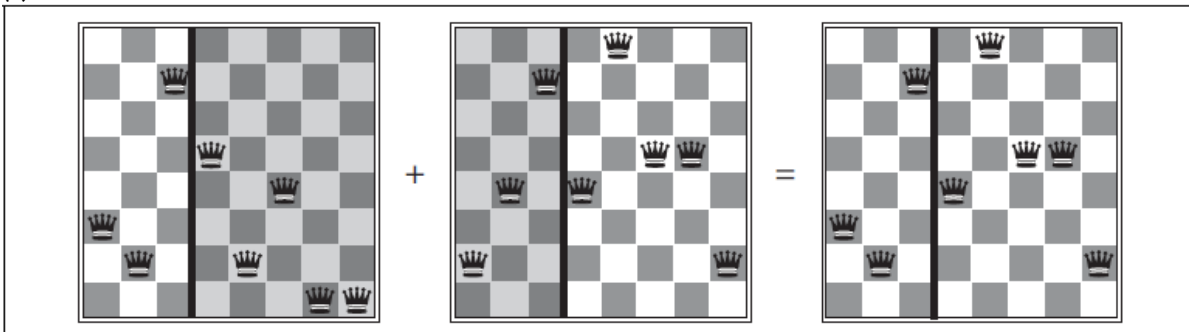


Figure: The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

function GENETIC-ALGORITHM(*population*, FITNESS-FN) **returns** an individual

inputs: *population*, a set of individuals

FITNESS-FN, a function that measures the fitness of an individual

repeat

new_population $\leftarrow$ empty set

for $i = 1$ **to** SIZE(*population*) **do**

$x \leftarrow$ RANDOM-SELECTION(*population*, FITNESS-FN)

$y \leftarrow$ RANDOM-SELECTION(*population*, FITNESS-FN)

child $\leftarrow$ REPRODUCE(x, y)

if (small random probability) **then** *child* $\leftarrow$ MUTATE(*child*)

add *child* **to** *new_population*

population $\leftarrow$ *new_population*

until some individual is fit enough, or enough time has elapsed

return the best individual in *population*, according to FITNESS-FN

function REPRODUCE(x, y) **returns** an individual

inputs: x, y , parent individuals

$n \leftarrow$ LENGTH(x); $c \leftarrow$ random number from 1 to n

return APPEND(SUBSTRING($x, 1, c$), SUBSTRING($y, c + 1, n$))

Figure: A genetic algorithm. The algorithm is with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

The theory of genetic algorithms explains how this works using the idea of a schema, which is a substring in which some of the positions can be left unspecified.

Genetic algorithms work best when schemata correspond to meaningful components of a solution. For example, if the string is a representation of an antenna, then the schemata may represent components of the antenna,

such as reflectors and deflectors. A good component is likely to be good in a variety of different designs. This suggests that successful use of genetic algorithms requires careful engineering of the representation. In practice, genetic algorithms have had a widespread impact on optimization problems, such as circuit layout and job-shop scheduling. At present, it is not clear whether the appeal of genetic algorithms arises from their performance or from their aesthetically pleasing origins in the theory of evolution. Much work remains to be done to identify the conditions under which genetic algorithms perform well.

Genetic algorithm with 8-queens problem

An 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires $8 \times \log_2 8 = 24$ bits. Alternatively, the state could be represented as 8 digits, each in the range from 1 to 8. (We demonstrate later that the two encodings behave differently.) Figure below (a) shows a population of four 8-digit strings representing 8-queens states.

The production of the next generation of states is shown in Figure (b)–(e). In (b), each state is rated by the objective function, or (in GA terminology) the fitness function. The fitness function should return higher values for better states, so, for the 8-queens problem we use the number of non-attacking pairs of queens, which has a value of 28 for a solution. The values of the four states are 24, 23, 20, and 11. In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.

In (c), two pairs are selected at random for reproduction, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all. For each pair to be mated, a crossover point is chosen randomly from the positions in the string. In CROSSOVER Figure, the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.

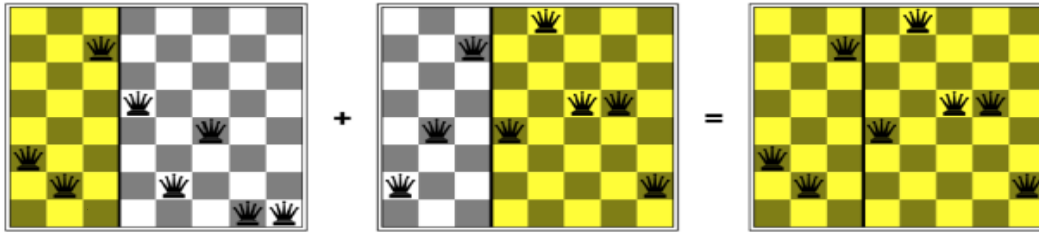
In (d), the offspring themselves are created by crossing over the parent strings at the crossover point. For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent. The 8-queens states involved in this reproduction step are shown below. The example shows that when two parent states are quite different, the crossover operation can produce a state that is a long way from either parent state. It is often the case that the population is quite diverse early on in the process, so crossover (like simulated annealing) frequently takes large steps in the state space early in the search process and smaller steps later on when most individuals are quite similar.

Finally, in (e), each location is subject to random mutation with a small independent MUTATION probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column. Figure describes an algorithm that implements all these steps. Like stochastic beam search, genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads. The primary advantage, if any, of genetic algorithms comes from the crossover operation. Yet it can be shown mathematically that, if the positions of the genetic code are permuted initially in a random order, crossover conveys no advantage. Intuitively, the advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates. For example, it could be that putting the first three queens in positions 2, 4, and 6 (where they do not attack each other) constitutes a useful block that can be combined with other blocks to construct a solution.

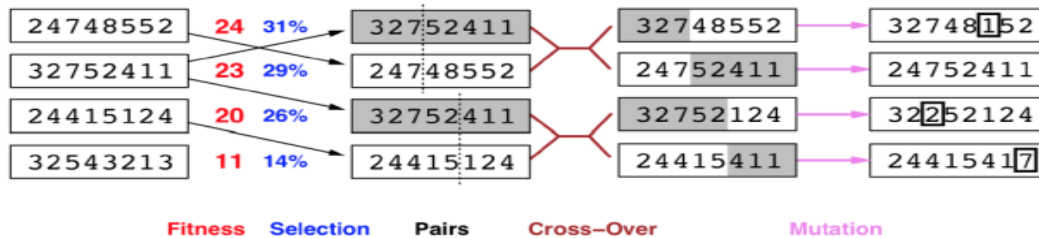
Genetic algorithms

Crossover

Taken from the edX course ColumbiaX: CSMM.101x Artificial Intelligence (AI)



Generate successors from pairs of states.



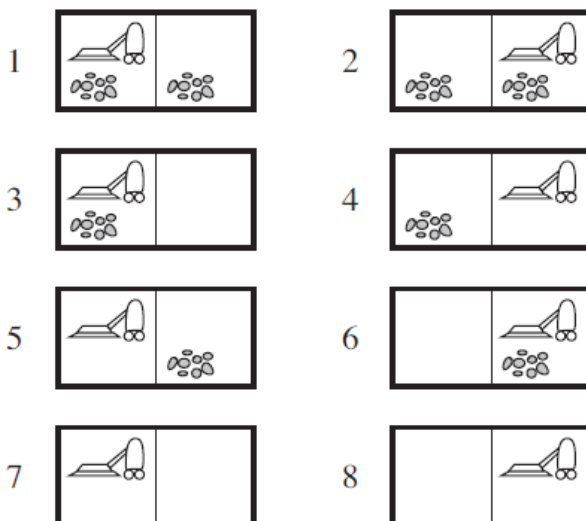
The theory of genetic algorithms explains how this works using the idea of a schema, SCHEMA which is a substring in which some of the positions can be left unspecified. For example, the schema 246***** describes all 8-queens states in which the first three queens are in positions 2, 4, and 6, respectively. Strings that match the schema (such as 24613578) are called instances of the schema. It can be shown that if the average fitness of the instances of a schema is above the mean, then the number of instances of the schema in the population will grow over time. Clearly, this effect is unlikely to be significant if adjacent bits are totally unrelated to each other, because then there will be few contiguous blocks that provide a consistent benefit. Genetic algorithms work best when schemata correspond to meaningful components of a solution.

2.2.2 Searching with non-deterministic action

When the environment is either partially observable or nondeterministic (or both), percepts become useful. In a partially observable environment, every percept helps narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals. When the environment is nondeterministic, percepts tell the agent which of the possible outcomes of its actions has actually occurred. So the solution to a problem is not a sequence but a contingency plan (also known as a strategy) that specifies what to do depending on what percepts are received.

The erratic vacuum world

Eight states, as shown in fig. There are three actions—Left, Right, and Suck—and the goal is to clean up all the dirt (states 7 and 8).



In the erratic vacuum world, the Suck action works as follows:

1. When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.
 2. When applied to a clean square the action sometimes deposits dirt on the carpet. To provide formulation of this problem, we need to generalize the notion of a transition model.
 Through the RESULT function that returns a single state, we use a RESULTS function that returns a set of possible outcome states.

For example, in the erratic vacuum world, the Suck action in state 1 leads to a state in the set {5, 7}—the dirt in the right-hand square may or may not be vacuumed up. We also need to generalize the notion of a solution to the problem. For example, if we start in state 1, there is no single sequence of actions that solves the problem.

Instead, we need a contingency plan such as the following:

[Suck, if State = 5 then [Right, Suck] else []] .

Thus, solutions for nondeterministic problems can contain nested if—then—else statements; this means that they are trees rather than sequences. Many problems in the real, physical world are contingency problems because exact prediction is impossible. For this reason, many people keep their eyes open while walking around or driving.

And/Or or AO* search

Amarel (1967) applied the idea to propositional theorem proving, and introduced a search algorithm similar to AND-OR-GRAPH-SEARCH. The algorithm was further developed and formalized by Nilsson (1971), who also described AO*—which, as its name suggests, finds optimal solutions given an admissible heuristic. AO* was analyzed and improved by Martelli and Montanan (1973). AO* is a top-down algorithm; a bottom-up generalization of A* is A*LD, for A* Lightest Derivation (Felzenszwalb and McAllester, 2007).

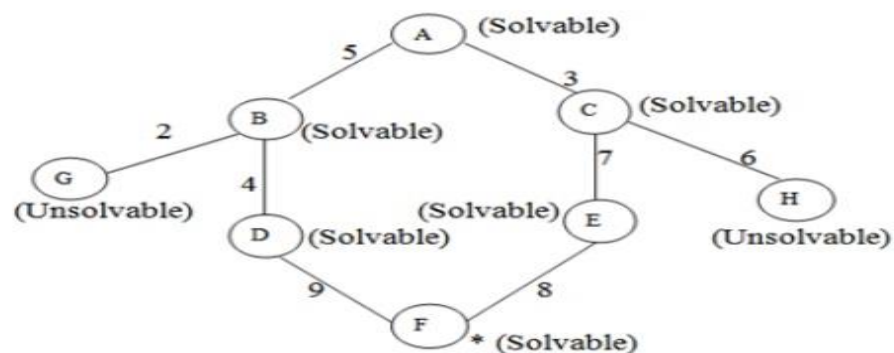
The Depth first search and Breadth first search given earlier for OR trees or graphs can be easily adopted by AND-OR graph. The main difference lies in the way termination conditions are determined, since all goals following an AND nodes must be realized; whereas a single goal node following an OR node will do. So for this purpose we are using AO* algorithm.

Like A* algorithm here we will use two arrays and one heuristic function.

- OPEN: It contains the nodes that has been traversed but yet not been marked solvable or unsolvable.
- CLOSE: It contains the nodes that have already been processed.

Let us take the following example to implement the AO* algorithm.

Eg:



Figure

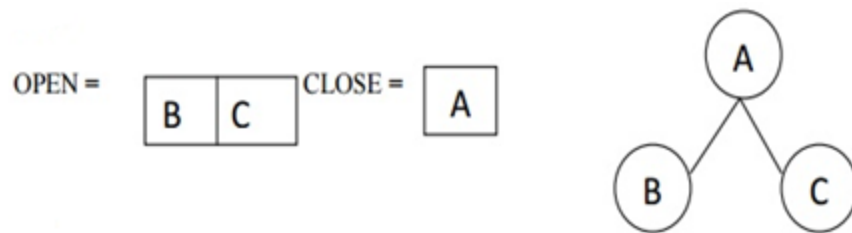
Step 1:

In the above graph, the solvable nodes are A, B, C, D, E, F and the unsolvable nodes are G, H. Take A as the starting node. So place A into OPEN.

i.e. OPEN = A CLOSE = (NULL) ϕ A

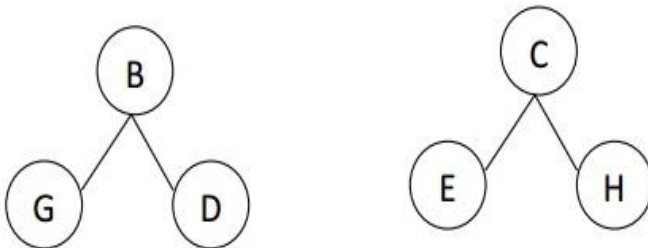
Step 2:

The children of A are B and C which are solvable. So place them into OPEN and place A into the close.



Step 3:

Now process the nodes B and C. The children of B and C are to be placed into OPEN. Also remove B and c from OPEN and place them into CLOSE.

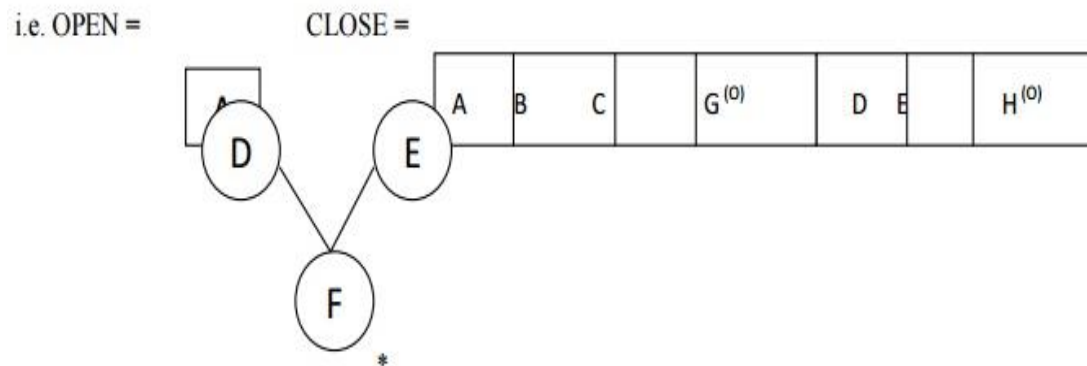


(O)

'O' indicates that the nodes G and H are unsolvable.

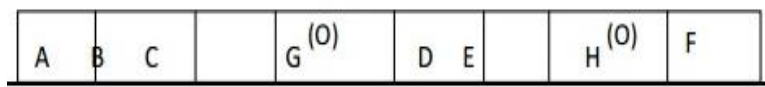
Step 4:

As the nodes G and H are unsolvable, so place them directly into CLOSE and process the nodes D and E.



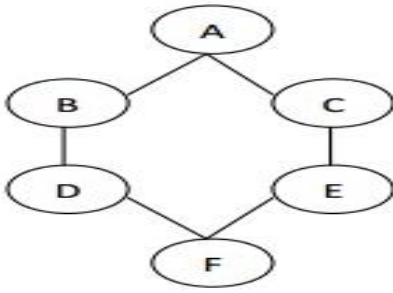
Step 5:

Now we have been reached at our goal state. So place F into CLOSE.



Step 6:

Success and exit



AO* graph

function AND-OR-GRAPH-SEARCH(*problem*) **returns** a conditional plan, or failure
 OR-SEARCH(*problem*.INITIAL-STATE, *problem*, [])

function OR-SEARCH(*state*, *problem*, *path*) **returns** a conditional plan, or failure
 if *problem*.GOAL-TEST(*state*) **then** **return** the empty plan
 if *state* is on *path* **then** **return** failure
 for each *action* in *problem*.ACTIONS(*state*) **do**
 plan ← AND-SEARCH(RESULTS(*state*, *action*), *problem*, [*state* | *path*])
 if *plan* ≠ failure **then** **return** [*action* | *plan*]
return failure

function AND-SEARCH(*states*, *problem*, *path*) **returns** a conditional plan, or failure
 for each *s_i* in *states* **do**
 plan_i ← OR-SEARCH(*s_i*, *problem*, *path*)
 if *plan_i* = failure **then** **return** failure
return [if *s₁* **then** *plan₁* **else** if *s₂* **then** *plan₂* **else** ... if *s_{n-1}* **then** *plan_{n-1}* **else** *plan_n*]

An algorithm for searching AND-OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation [*x* | *l*] refers to the list formed by adding object *x* to the front of list *l*.)

2.2.3 Searching with partial observations

The key concept required for solving partially observable problems is the belief state, representing the agent's current belief about the possible physical states it might be in, given the sequence of actions and percepts up to that point. We begin with the simplest scenario for studying belief states, which is when the agent has no sensors at all; then we add in partial sensing as well as nondeterministic actions.

Searching with no observation

When the agent's percepts provide no information at all, we have what is called a sensorless problem or sometimes a conformant problem. At first, one might think the sensorless agent has no hope of solving a problem if it has no idea what state it's in; in fact, sensorless problems are quite often solvable. Moreover, sensorless agents can be surprisingly useful, primarily because they don't rely on sensors working properly. In manufacturing systems, for example, many ingenious methods have been developed for orienting parts correctly from an unknown initial position by using a sequence of actions with no sensing at all. The high cost of sensing is another reason to avoid it: for example, doctors often prescribe a broadspectrum antibiotic rather than using the contingent plan of doing an expensive blood test, then waiting for the results to come back, and then prescribing a more specific antibiotic and perhaps hospitalization because the infection has progressed too far.

We can make a sensorless version of the vacuum world. Assume that the agent knows the geography of its world, but doesn't know its location or the distribution of dirt. In that case, its initial state could be any element of the set {1, 2, 3, 4, 5, 6, 7, 8}. Now, consider what happens if it tries the action Right. This will cause it to be in one of the states {2, 4, 6, 8}—the agent now has more information! Furthermore, the action sequence [Right,Suck] will always end up in one of the states {4, 8}. Finally, the sequence [Right,Suck,Left,Suck] is guaranteed to reach the goal state 7 no matter what the start state. We say that the agent can coerce the world into state 7.

It is instructive to see how the belief-state search problem is constructed. Suppose the underlying physical problem P is defined by $ACTIONS_P$, $RESULT_P$, $GOAL-TEST_P$, and $STEP-COST_P$. Then we can define the corresponding sensorless problem as follows:

- **Belief states:** The entire belief-state space contains every possible set of physical states. If P has N states, then the sensorless problem has up to 2^N states, although many may be unreachable from the initial state.
- **Initial state:** Typically the set of all states in P , although in some cases the agent will have more knowledge than this.
- **Actions:** This is slightly tricky. Suppose the agent is in belief state $b = \{s_1, s_2\}$, but $ACTIONS_P(s_1) \neq ACTIONS_P(s_2)$; then the agent is unsure of which actions are legal. If we assume that illegal actions have no effect on the environment, then it is safe to take the *union* of all the actions in any of the physical states in the current belief state b :

$$ACTIONS(b) = \bigcup_{s \in b} ACTIONS_P(s)$$

On the other hand, if an illegal action might be the end of the world, it is safer to allow only the intersection, that is, the set of actions legal in all the states. For the vacuum world, every state has the same legal actions, so both methods give the same result.

- **Transition model:** The agent doesn't know which state in the belief state is the right one; so as far as it knows, it might get to any of the states resulting from applying the action to one of the physical states in the belief state. For deterministic actions, the set of states that might be reached is

$$b' = RESULT(b, a) = \{s' : s' = RESULT_P(s, a) \text{ and } s \in b\}$$

With deterministic actions, b' is never larger than b . With nondeterminism, we have

$$b' = RESULT(b, a) = \{s' : s' = RESULT_P(s, a) \text{ and } s \in b\} \\ = \bigcup_{s \in b} RESULT_P(s, a)$$

which may be larger than b . The process of generating the new belief state after the action is called the **prediction** step; the notation $b' = PREDICT_P(b, a)$ will come in handy.

- **Goal test:** The agent wants a plan that is sure to work, which means that a belief state satisfies the goal only if all the physical states in it satisfy $GOAL-TEST_P$. The agent may accidentally achieve the goal earlier, but it won't know that it has done so.

- **Path cost:** This is also tricky. If the same action can have different costs in different states, then the cost of taking an action in a given belief state could be one of several values. For now we assume that the cost of an action is the same in all states and so can be transferred directly from the underlying physical problem.

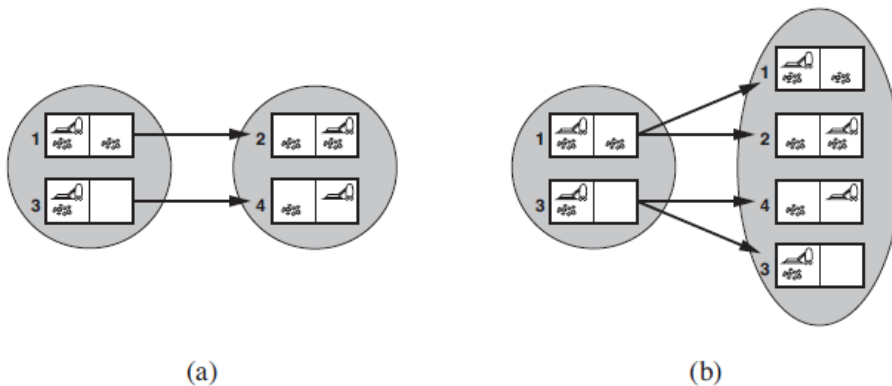


Figure: (a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, Right. (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

Searching with observations

When observations are partial, it will usually be the case that several states could have produced any given percept. For example, the percept $[A, \text{Dirty}]$ is produced by state 3 as well as by state 1. Hence, given this as the initial percept, the initial belief state for the local-sensing vacuum world will be $\{1, 3\}$. The $ACTIONS$, $STEP-COST$, and $GOAL-TEST$ are constructed from the underlying physical problem just as for sensorless problems, but the transition model is a bit more complicated. We can think of transitions from one belief state to the next for a particular action as occurring in three stages, as shown in Figure 4.15:

- The **prediction** stage is the same as for sensorless problems: given the action a in belief state b , the predicted belief state is $b' = PREDICT(b, a)$.

- The **observation prediction** stage determines the set of percepts o that could be observed in the predicted belief state:

$\text{POSSIBLE-PERCEPTS}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\}.$

- The **update** stage determines, for each possible percept, the belief state that would result from the percept. The new belief state b_o is just the set of states in $\hat{b}$ that could have produced the percept:

$b_o = \text{UPDATE}(\hat{b}, o) = \{s : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\}.$

Notice that each updated belief state b_o can be no larger than the predicted belief state $\hat{b}$; observations can only help reduce uncertainty compared to the sensorless case. Moreover, for deterministic sensing, the belief states for the different possible percepts will be disjoint, forming a *partition* of the original predicted belief state.

2.2.4 Online search agents and Unknown Environments

Online search problem

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment, but we stipulate that the agent knows only the following:

- $\text{ACTIONS}(s)$, which returns a list of actions allowed in state s ;
- The step-cost function $c(s, a, s')$ —note that this cannot be used until the agent knows that s' is the outcome.
- $\text{GOAL-TEST}(s)$.

Note in particular that the agent cannot determine $\text{RESULT}(s, a)$ except by actually being in s and doing a . For example, in the maze problem shown in Figure below, the agent does not know that going Up from (1,1) leads to (1,2); nor, having done that, does it know that going Down will take it back to (1,1). This degree of ignorance can be reduced in some applications—for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.

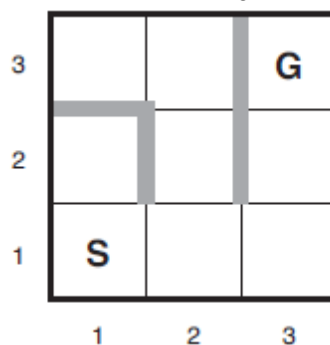


Figure: A simple maze problem. The agent starts at S and must reach G but knows nothing of the environment.

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 above, the agent might know the 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—i.e., they lead to a state from which no action leads back to the previous state—the online search might accidentally reach a dead-end state from which no goal state is reachable. Perhaps the term “accidentally” is 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 below(a).

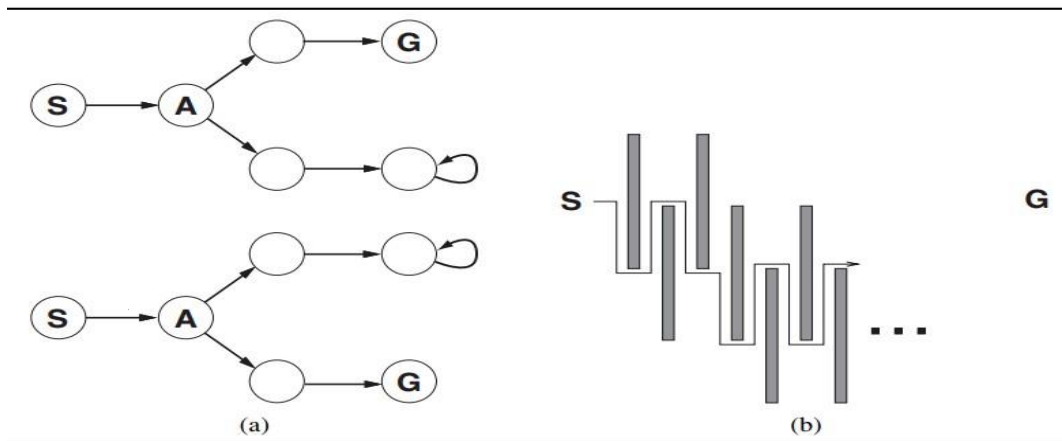


Figure: (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

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 constructing the state space while the agent explores it and putting the goals and dead ends wherever it chooses. Dead ends are a real difficulty for robot exploration—staircases, ramps, cliffs, one-way streets, and all kinds of natural terrain present opportunities for irreversible actions. To make progress, we 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 above(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 we have seen previously.

function ONLINE-DFS-AGENT(s') **returns** an action

inputs: s' , a percept that identifies the current state

persistent: *result*, a table indexed by state and action, initially empty

untried, a table that lists, for each state, the actions not yet tried

unbacktracked, a table that lists, for each 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 (not in *untried*) **then** *untried*[s'] $\leftarrow$ ACTIONS(s')

if s is not null **then**

result[s , a] $\leftarrow s'$

add s to the front of *unbacktracked*[s']

if *untried*[s'] is empty **then**

if *unbacktracked*[s'] is empty **then return** stop

else $a \leftarrow$ an action b such that *result*[s' , b] = POP(*unbacktracked*[s'])

else $a \leftarrow$ POP(*untried*[s'])

$s \leftarrow s'$

return a

Figure: An online search agent that uses depth-first exploration. The agent is applicable only in state spaces in which every action can be “undone” by some other action.

For example, offline algorithms such as A* can 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 discover successors only for 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.

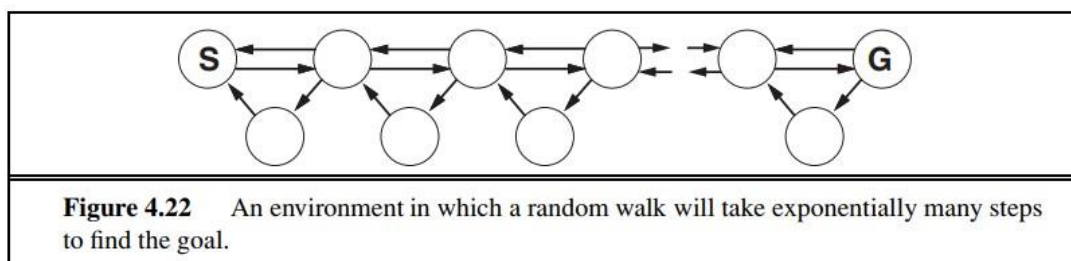
An online depth-first search agent is shown in algorithm above. This agent stores its map in a table, $RESULT[s, a]$, that records the state resulting from executing action a in state s .

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 most recently entered the current state.

To achieve that, the algorithm keeps a table that lists, for each state, the predecessor states to which the agent has not yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete. We recommend that the reader trace through the progress of ONLINE-DFS-AGENT when applied to the maze given in Figure Maze problem. 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.

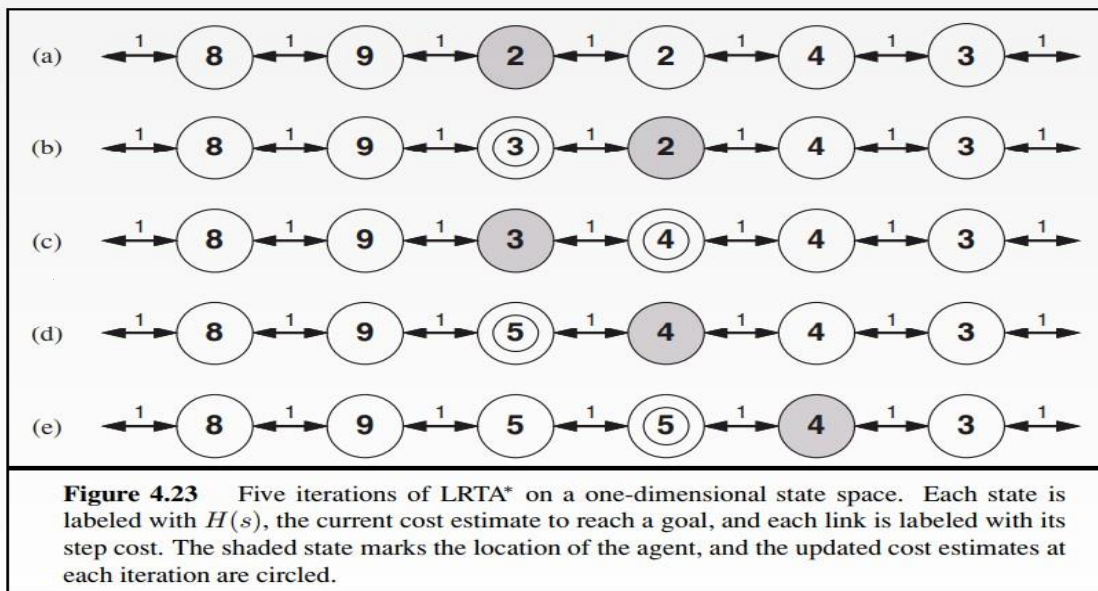
Online local search

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. On the other hand, the process can be very slow. Figure below 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 estimate $h(s)$ and is updated as the agent gains experience in the state space. Figure 4.23 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 given 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, $c(s, a, s) + H(s)$. 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 4.23(b). 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* LRTA*), is shown in Figure 4.24. Like ONLINE-DFS-AGENT, it builds a map of the environment in 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.



```

function LRTA*-AGENT( $s'$ ) returns an action
inputs:  $s'$ , a percept that identifies the current state
persistent: result, a table, indexed by state and action, 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  $H[s'] \leftarrow h(s')$ 
if  $s$  is not null
     $result[s, a] \leftarrow s'$ 
     $H[s] \leftarrow \min_{b \in ACTIONS(s)} LRTA^*-COST(s, b, result[s, b], H)$ 
     $a \leftarrow$  an action  $b$  in  $ACTIONS(s')$  that minimizes  $LRTA^*-COST(s', b, result[s', b], H)$ 
     $s \leftarrow s'$ 
return  $a$ 

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

```

Figure 4.24 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

An LRTA* agent is guaranteed to find a goal in any finite, safely explorable environment. Unlike A*, however, it is not complete for infinite state spaces—there are cases where it can be led infinitely astray. It can explore an environment of n states in $O(n^2)$ steps in the worst case, but often does much better. The LRTA* agent is just one of a large family of online agents that one can define by specifying the action selection rule and the update rule in different ways.

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 each state—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 cost of each state by using local updating rules, as in LRTA*. It shows that 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 lowest-cost 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 of following figure, you will have noticed that the agent is not very bright. For example, after it has seen that the Up action goes from (1,1) 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 (2,1) 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, that Down reduces it, and so on. For this to happen, we need two things. First, we need a formal and explicitly manipulable representation for these kinds of general rules; so far, we have hidden the information inside the black box called the RESULT function. Part III is devoted to this issue. Second, we need algorithms that can construct suitable general rules from the specific observations made by the agent.

Practice Exercise

Graded Questions

Q.1 Write short note on problem solving agents

Q.2 What are the various components of a problem? Explain

Q.3 Explain how problem is formulated for toy problem/ vacuum world problem [May 2019]

Or

Discuss in brief the formulation of single state problem [Nov 2018]

Q.4 Explain how problem is formulated for 8 - puzzle problem

Q.5 Explain how problem is formulated for 8 - queens problem

Q.6 Explain how problem is formulated for real world problems

Q.7 Explain infrastructure for search algorithms

Q.8 How you will measure problem-solving performance? [May 2019]

Q.9 What are uninformed search strategies? Explain

Q.10 Explain Breadth First Search with an example

Q.11 Write algorithm of Breadth First Search [Nov 2018]

Q.12 Explain Uniform Cost search with an example [May 2019]

Q.13 Explain algorithm of Uniform Cost Search

Q.14 Explain Depth First Search with an example

Q.15 Explain algorithm of Depth limited search with an example

Q.16 Write short note on Iterative Deepening depth-first search

Q.17 What are the differences between Depth First Search, Depth limited search and Iterative Deepening depth-first search?

Q.18 What is bidirectional search? How it is advantageous over other searches?

Q.19 Compare all Uniformed search strategies

Q.20 Give the outline of tree search algorithm [Nov 2018]

Q.20 What do you mean by Informed or heuristic search strategies? Explain

Q.21 Explain Greedy best-first search with an example

Q.22 Explain A* algorithm with an example

Q.23 Explain RBFS with an example

Q.24 What is IDA*? Explain

Q.25 Write short notes on MA* and SMA*

Q.26 What do you mean by heuristic function?

Q.27 What are local search problems? Explain

Q.28 Write short note on Hill Climbing search

Q.29 What are the problems of Hill Climbing search? Explain

OR

With suitable diagram explain the following concepts

i. shoulder ii. Global maximum iii. Local maximum [May 2019]

Q.30 What are the various variants of Hill Climbing search? Explain

Q.31 Give the illustration of 8 queen problem using hill climbing algorithm. [Nov 2018]

Q.31 Write short note on Simulated Annealing

Q.32 What do you mean by Local Beam search?

Q.33 Write short note on Genetic algorithms [Nov 2018, May 2019]

Q.34 Explain Genetic algorithm with 8-queens problem

Q.35 Write short note on search with nondeterministic actions with respect to vacuum cleaner world

Or

Explain how transition model is used for sensing in vacuum cleaner problem [Nov 2018]

Q.36 Explain And/Or or AO* Search with an example

[May 2019]

Q.37 Write short note on searching with no observations

Q.38 Write short note on searching with observations

Q.39 How you will solve partially observable problems

Q.40 Write short note on online search problems

Q.41 What do you mean by online search agents?

Q.42 Write short note on Online local search

Q.43 How learning happens in online search?