Artificial Intelligence

Unit –I

We call ourselves *Homo sapiens*—Humans with modern INTELLIGENCE and the wise—because our **intelligence** is so important to us. For thousands of years, we 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 ARTIFICIAL more complicated than itself. The field of **artificial intelligence**, or AI, goes further still: it INTELLIGENCE attempts not just to understand but also to *build* intelligent entities.

1.1 What is AI?

In Figure 1.1 we see eight definitions of AI, laid out along two dimensions. The definitions on top are concerned with *thought processes* and *reasoning*, whereas the ones on the bottom address *behavior*. The definitions on the left measure success in terms of fidelity to *human* performance, whereas RATIONALITY the ones on the right measure against an *ideal* performance measure, called **rationality**. A system is rational if it does the "right thing," given what it knows.

	Human-centered	Rationality-centered
Thought-centered	Systems that think like humans.	Systems that think rationally.
Behaviour-centered	Systems that act like humans.	Systems that act rationally.

1.1.1 Acting humanly: The Turing Test approach

The **Turing Test**, proposed by Alan Turing TURING TEST (1950), was designed to provide a satisfactory operational definition of intelligence. A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer. 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.

so-called **total Turing Test** includes a video signal so that the interrogator can test the subject's perceptual abilities, To pass the total Turing Test, the computer will need

• **computer vision** to perceive objects, and
• **robotics** to manipulate objects and move about.

1.1.2 Thinking humanly: The cognitive modeling approach

we must have some way of determining how humans think.- 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.

For example, Allen Newell and Herbert Simon, who developed GPS, the “General Problem Solver”.- were not content merely to have their program solve problems correctly. They were more concerned with comparing the trace of its reasoning steps to traces of human subjects solving the same problems.
cognitive science =computer models from AI + experimental techniques from psychology -> to construct precise and testable theories of the human mind.

1.1.3 Thinking rationally: The “laws of thought” approach

The Greek philosopher Aristotle was one of the first to attempt to codify “right thinking,” that is, irrefutable reasoning processes.
for example, “Socrates is a man; all men are mortal; therefore, Socrates is mortal.”
These laws of thought were supposed to govern the operation of the mind-- their study initiated the field called **logic.**

1.1.4 Acting rationally: The rational agent approach

An **agent** is just something that acts ? additionally, operate autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals.
A **rational agent** is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.
In the “laws of thought” - the emphasis was on correct inferences. - sometimes *part* of being a rational agent, because , act rationally is to reason logically to the conclusion that a given action will achieve one’s goals and then to act on that conclusion. On the other hand, correct inference is not *all* of rationality;
in some situations, there is no provably correct thing to do, but something must still be done. There are also ways of acting rationally that cannot be said to involve inference.
All the skills needed for the Turing Test also allow an agent to act rationally.
Knowledge representation + reasoning -> agents to reach good decisions.
The rational-agent - advantages :
more general than the “laws of thought” approach because correct inference is just one of several possible mechanisms for achieving rationality.
more amenable to scientific development than are approaches based on human behavior or human thought.

2.Intelligent Agents

2.1 AGENTS AND ENVIRONMENTS

Def: An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators**.

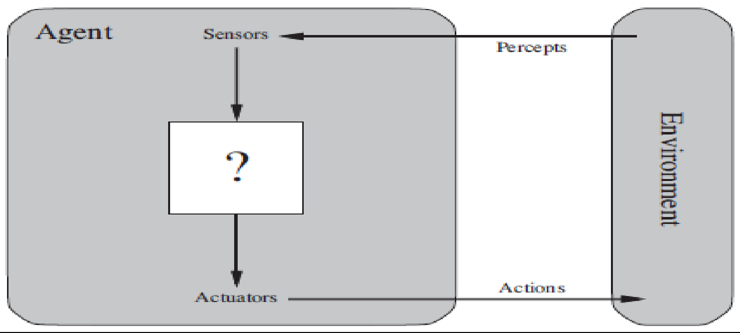


Figure 2.1 Agents interact with environments through sensors and actuators.

A human agent ---sensors : has eyes, ears ; actuators.: hands, legs, vocal tract, and so on

robotic agent -- sensors :cameras and infrared range finders;   actuators :various motors
A software agent -- sensory inputs : receives keystrokes, file contents, and network packets; acts on the environment by displaying on the screen, writing files, and sending network packets.
percept : the agent's perceptual inputs at any given instant.
percept sequence : complete history of everything the agent has ever perceived.
agent function :describes an agent's behavior that maps any given percept sequence to an action. Thus, agent function is an abstract mathematical description; the agent program is a concrete implementation, running within some physical system.
Given an agent to experiment with, we can, in principle, construct this table by trying out all possible percept sequences and recording which actions the agent does in response. The table is, of course, an *external* characterization of the agent. *Internally*, the agent function for an artificial agent will be implemented by an **agent program**. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running within some physical system.
simple example—the vacuum-cleaner world shown in Figure 2.2. 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 whether 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 clean; otherwise, move to the other square. A partial tabulation of this agent function is shown in Figure 2.3 and an agent program that implements it appears in Figure 2.8.

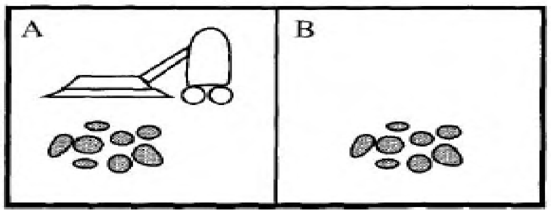
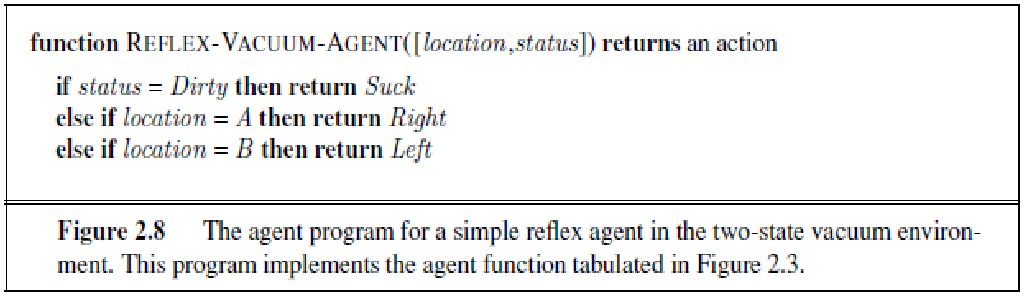


Figure 2.2 A vacuum-cleaner world with just two locations.

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

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2.



```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

Figure 2.8 The agent program for a simple reflex agent in the two-state vacuum environment. This program implements the agent function tabulated in Figure 2.3.

2.2 GOOD BEHAVIOR: THE CONCEPT OF RATIONALITY

A **rational agent** is one that does the right thing—conceptually speaking, every entry in the table for the agent function is filled out correctly.

Obviously, there is not one fixed performance measure for all tasks and agents; typically, a designer will devise one appropriate to the circumstances. This is not as easy as it sounds.

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.*

2.2.1 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.

2.2.2 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 Elys´ees 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,2 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.

Our definition requires a rational agent not only to gather information but also to **learn** as much as possible from what it perceives. The agent's initial configuration could reflect some prior knowledge of the environment, but as the agent gains experience this may be modified and augmented. There are extreme cases in which the environment is completely known *a priori*. In such cases, the agent need not perceive or learn; it simply acts correctly. Of course, such agents are fragile.
To the extent that an agent relies on the prior knowledge of its designer rather than AUTONOMY on its own percepts, we say that the agent lacks **autonomy**. A rational agent should be autonomous—it should learn what it can to compensate for partial or incorrect prior knowledge.

2.3 THE NATURE OF ENVIRONMENTS

2.3.1 Specifying the task environment

task environment. Includes the performance measure, the environment, and the agent's actuators and sensors. For the acronymically minded, we call
PEAS this the **PEAS** (**P**erformance, **E**nvironment, **A**ctuators, **S**ensors) description.
Ex:

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.

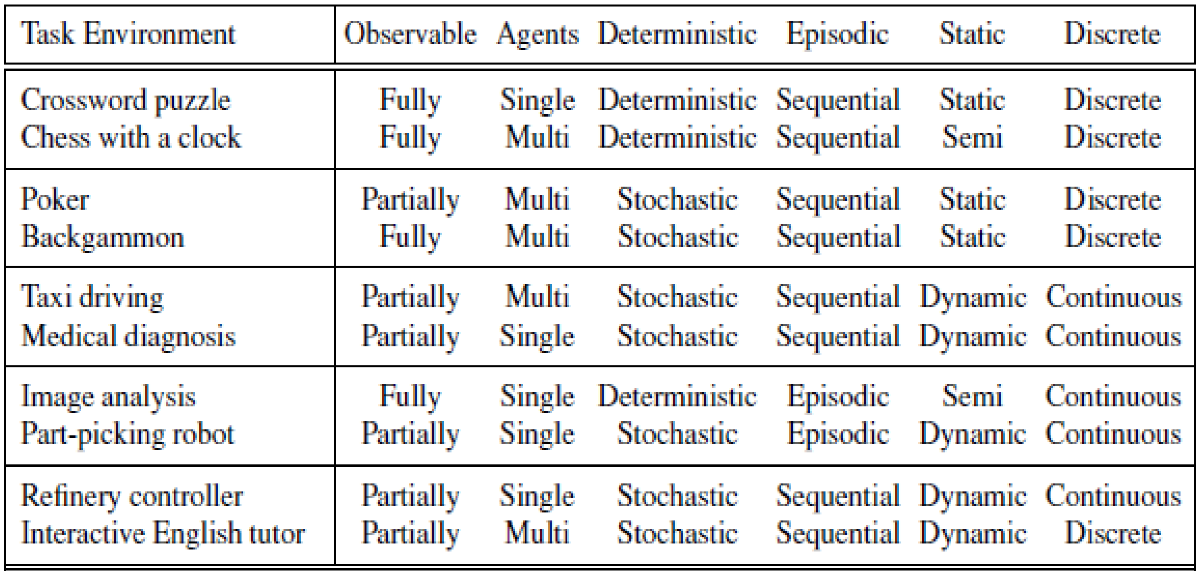
Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image 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.

2.3.2 Properties of task environments

Fully observable vs. **partially observable**: If sensors give it access to the complete state of the environment at each point in time, then the task environment is fully observable.
It is effectively fully observable, if the sensors detect all aspects that are *relevant* to the choice of action; *depends on the performance measure*.
convenient because the agent need not maintain any internal state to keep track of the world.
- might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data
Eg: a vacuum agent with only a local dirt sensor cannot tell whether there is dirt in other squares, an automated taxi cannot see what other drivers are thinking.
Unobservable: If the agent has no sensors at all then the environment
Single agent vs. **multiagent**: Eg: an agent solving a crossword puzzle by itself - in a single-agent environment, agent playing chess – two agent environment.
[entity *may* be viewed as an agent, but we have not explained which entities *must* be viewed as agents. Does an agent A (the taxi driver for example) have to treat an object B (another vehicle) as an agent, or can it be treated merely as an object behaving according to the laws of physics
The key distinction is whether B's behavior is best described as maximizing a performance measure whose value depends on agent A's behavior.]
competitive multiagent environment: in chess, the opponent entity B is trying to maximize its performance measure, which, by the rules of chess, minimizes agent A's performance measure
partially **cooperative** multiagent environment: In the taxi-driving environment, avoiding collisions maximizes the performance
It is also partially competitive :because, for example, only one car can occupy a parking space.
Deterministic vs. **stochastic:** the next state of the environment is completely determined by the current state and the action executed by the agent is then the environment deterministic; otherwise, it is stochastic.
Accessible vs. **inaccessible.**
If an agent's sensory apparatus gives it access to the complete state of the environment, then we say that the environment is accessible to that agent. An environment is effectively accessible if the sensors detect all aspects that are relevant to the choice of action. An accessible environment is convenient because the agent need not maintain any internal state to keep track of the world.
Deterministic vs. **nondeterministic.**
If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic. In principle, an agent need not worry about uncertainty in an accessible, deterministic environment. If the environment is inaccessible or partially observable, then it may *appear to* be nondeterministic. This is particularly true if the environment is complex, making it hard to keep track of all the inaccessible aspects. When the probability of next state cannot be calculated by the current state, then it is nondeterministic environment.
Episodic vs. **nonepisodic.**
In an episodic environment, the agent's experience is divided into "episodes." Each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself, because subsequent episodes do not depend on what actions occur in previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.
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. If the environment does not change with the passage of time but the agent's performance score does, then we say the environment is **semidynamic.**

Discrete vs. **continuous.**
If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Chess is discrete—there are a fixed number of possible moves on each turn. Taxi driving is continuous—the speed and location of the taxi and the other vehicles sweep **through a range of**

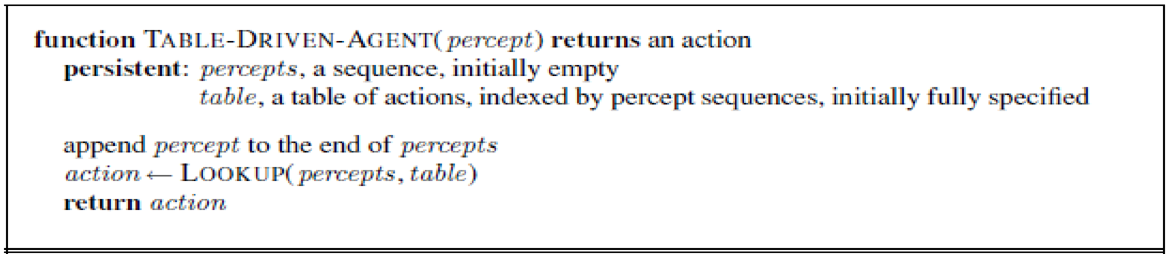


Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	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 tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.6 Examples of task environments and their characteristics.

2.4 THE STRUCTURE OF AGENTS
2.4.1 Agent programs

The agent programs all have the same skeleton: they take the current percept as input from the sensors and return an action to the actuator. 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 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
```

Figure 2.7 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

Drawbacks:

- **Table lookup** of percept-action pairs defining all possible condition-action rules necessary to interact in an environment

- **Problems**
  - Too big to generate and to store (Chess has about 10^120 states, for example)
  - No knowledge of non-perceptual parts of the current state
  - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
  - Looping: Can't make actions conditional
- Take a long time to build the table
- No autonomy
- Even with learning, need a long time to learn the table entries

Some Agent Types

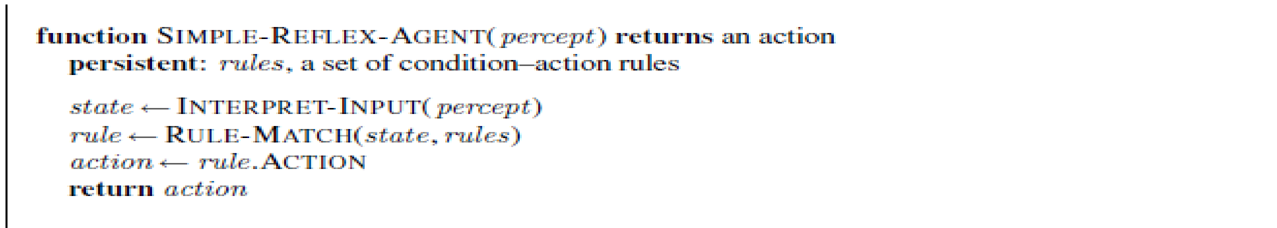
- **Table-driven agents**
  - use a percept sequence/action table in memory to find the next action. They are implemented by a (large) **lookup table**.
- **Simple reflex agents**
  - are based on **condition-action rules**, implemented with an appropriate production system. They are stateless devices which do not have memory of past world states.
- **Agents with memory**
  - have **internal state**, which is used to keep track of past states of the world.
- **Agents with goals**
  - are agents that, in addition to state information, have **goal information** that describes desirable situations. Agents of this kind take future events into consideration.
- **Utility-based agents**
  - base their decisions on **classic axiomatic utility theory** in order to act rationally.

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. For example, the vacuum agent whose agent function is tabulated in Figure 1.10 is a simple reflex agent, because its decision is based only on the current location and on whether that contains dirt.

- Implemented through *condition-action rules*
  If dirty then suck

2.4.2 A Simple Reflex Agent: Schema



```
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.

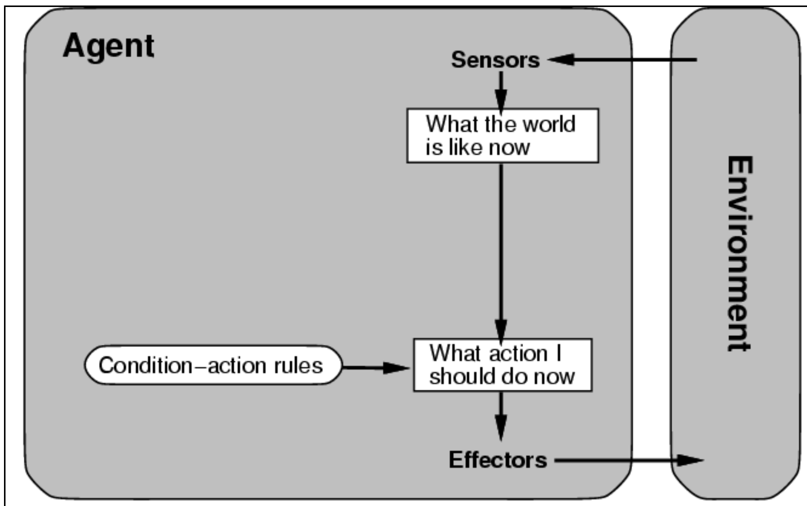


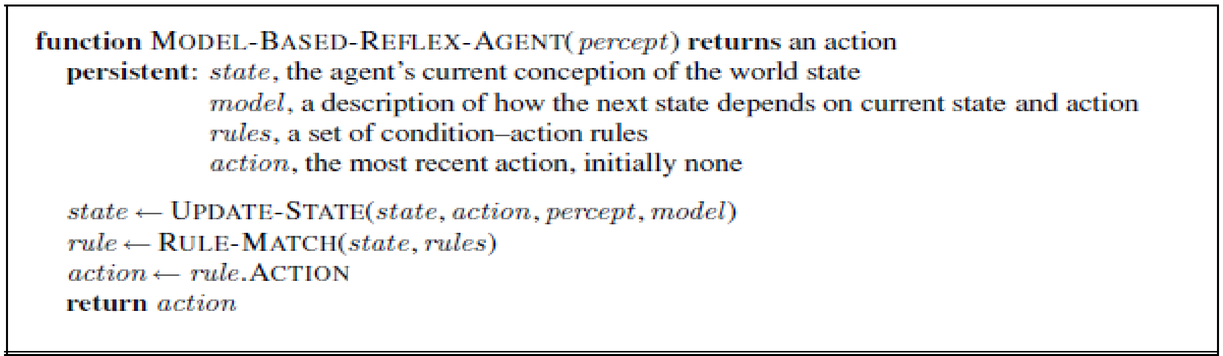
Figure 1.9 Schematic diagram of a simple reflex agent.

- ❖ **Characteristics**
  - o Only works if the environment is fully observable.
  - o Lacking history, easily get stuck in infinite loops
  - o One solution is to randomize actions

Model-based reflex agents

The most effective way to handle partial observability is for the agent to *keep track of the part of the world it can't see now*. That is, the agent should maintain some sort of **internal state** that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.

Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program. First, we need some information about how the world evolves independently of the agent-for example, that an overtaking car generally will be closer behind than it was a moment ago. Second, we need some information about how the agent's own actions affect the world-for example, that when the agent turns the steering wheel clockwise, the car turns to the right or that after driving for five minutes northbound on the freeway one is usually about five miles north of where one was five minutes ago. This knowledge about "how the world working - whether implemented in simple Boolean circuits or in complete scientific theories-is called a **model** of the world.


```
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.

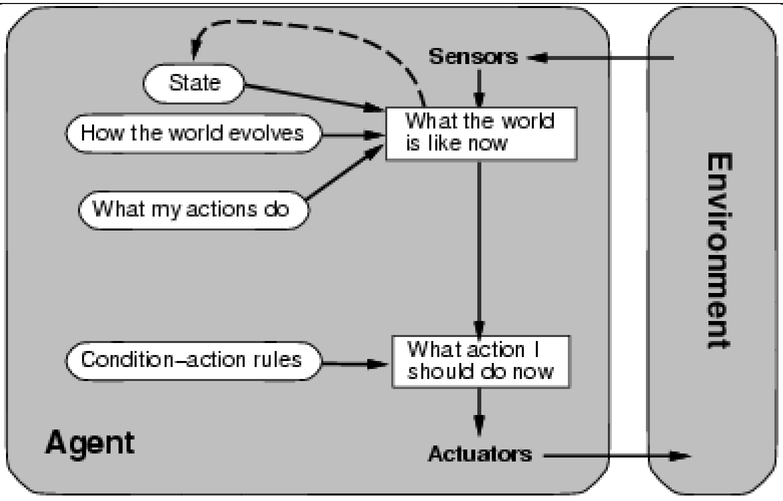


Figure 1.12 A model based reflex agent

2.4.3 Goal-based agents

Knowing about the current state of the environment is not always enough to decide what to do. For example, at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to. In other words, as well as a current state description, the agent needs some sort of **goal** information that describes situations that are desirable-for example, being at the passenger's destination. The agent program can combine this with information about the results of possible actions (the same information as was used to update internal state in the reflex agent) in order to choose actions that achieve the goal. Figure 1.13 shows the goal-based agent's structure. Some times it will be tricky with long sequence and twist to their goals, then they have to use 'search' & 'planning'. Different from condition-action rule. Simple agent- breaks when sees red light; model- could reason, if the front car red lights are on, it will slow down. Goal based: rainy day- how the breaks will operate- behaviors are altered to suit new conditions

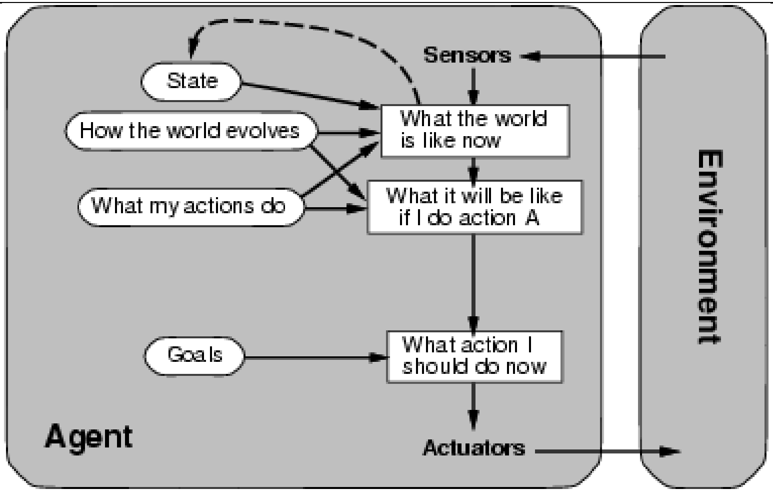

Sensors
State
How the world evolves
What the world is like now
What my actions do
What it will be like if I do action A
Environment
Goals
What action I should do now
Agent
Actuators

Figure 1.14 A goal based agent

2.4.4 Utility-based agents

Goals alone are not really enough to generate high-quality behavior in most environments. For example, there are many action sequences that will get the taxi to its destination (thereby achieving the goal) but some are quicker, safer, more reliable, or cheaper than others. Goals just provide a crude binary distinction between "happy" and "unhappy" states, whereas a more general **performance measure** should allow a comparison of different world states according to exactly how happy they would make the agent if they could be achieved. Because "happy" does not sound very scientific, the customary terminology is to say that if one world state is preferred to another, then it has higher **utility** for the agent.

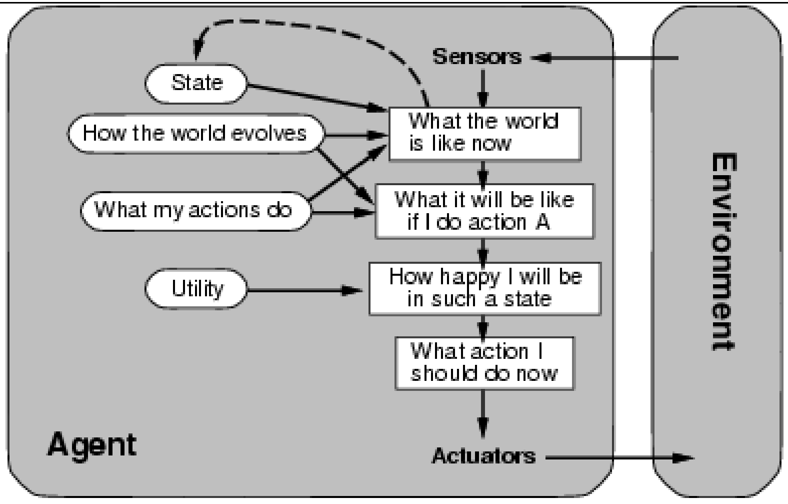

Sensors
State
What the world is like now
How the world evolves
What my actions do
What it will be like if I do action A
Utility
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Figure 1.15 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

- Certain goals can be reached in different ways.
  - Some are better, have a higher utility.
- If utility fun and performance measure are in agreement- agent chooses action to maximize its utility- rational
- Conflicting goals: (speed & safety)- utility function specifies the trade off-
- Select appropriately between several goals based on likelihood of success.

2.4.5 Learning agents

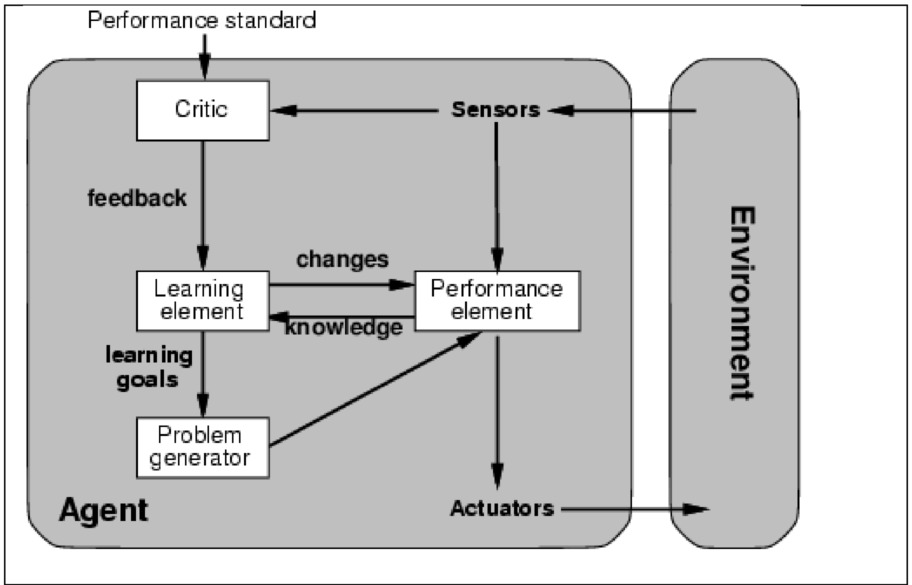


Figure 1.16 A general model of learning agents.

- All agents can improve their performance through **learning.**

A learning agent can be divided into four conceptual components, as shown in Figure 1.15 The most important distinction is between the **learning element**, which is responsible for making improvements, and the **performance element**, which is responsible for selecting external actions. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions. The learning element uses feedback from the **critic** on how the agent is doing and determines how the performance element should be modified to do better in the future.

The last component of the learning agent is the **problem generator**. It is responsible for suggesting actions that will lead to new and **informative experiences**. But if the agent is willing to explore a little, it might discover much better actions for the long run. The problem generator's job is to suggest these **exploratory actions**. This is what scientists do when they carry out experiments.

Summary: Intelligent Agents

- An **agent** perceives and acts in an environment, has an architecture, and is implemented by an agent program.
- Task environment – **PEAS (P**erformance**, E**nvironment, **A**ctuators, **S**ensors**)**
- The most challenging environments are inaccessible, nondeterministic, dynamic, and continuous.
- An **ideal agent** always chooses the action which maximizes its expected performance, given its percept sequence so far.
- An **agent program** maps from percept to action and updates internal state.
  - **Reflex agents** respond immediately to percepts.
    - simple reflex agents
    - model-based reflex agents
  - **Goal-based agents** act in order to achieve their goal(s).
  - **Utility-based agents** maximize their own utility function.
- All agents can improve their performance through **learning.**

Solving Problems by Searching

3.1 PROBLEM-SOLVING AGENTS

An important aspect of intelligence is ***goal-based*** problem solving.

The **solution** of many **problems** can be described by finding a **sequence of actions** that lead to a desirable **goal.** Each action changes the ***state*** and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.
A well-defined problem can be described by:

- **Initial state**
- **Operator or successor function** - for any state x returns s(x), the set of states reachable from x with one action
- **State space** - all states reachable from initial by any sequence of actions
- **Path** - sequence through state space
- **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
- **Goal test** - test to determine if at goal state

What is Search?
Search is the systematic examination of **states** to find path from the **start/root state** to the **goal state.**
The set of possible states, together with *operators* defining their connectivity constitute the *search space*.
The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.
Problem-solving agents

A Problem solving agent is a **goal-based** agent . It decide what to do by finding sequence of actions that lead to desirable states. The agent can adopt a goal and aim at satisfying it.

To illustrate the agent's behavior ,let us take an example where our agent is in the city of Arad,which is in Romania. The agent has to adopt a **goal** of getting to Bucharest.

Goal formulation,based on the current situation and the agent's performance measure,is the first step in problem solving. The agent's task is to find out which sequence of actions will get to a goal state.

Problem formulation is the process of deciding what actions and states to consider given a goal.
Search

An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that leads to the states of known value,and then choosing the best sequence. The process of looking for sequences actions from the current state to reach the goal state is called **search.**

The **search algorithm** takes a **problem** as **input** and returns a **solution** in the form of **action sequence.** Once a solution is found,the **execution phase** consists of carrying out the recommended action.

agent calls a search procedure to solve it. It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do—typically, the first action of the sequence—and then removing that step from the sequence. Once the solution has been executed, the agent will formulate a new goal.

while the agent is executing the solution sequence it *ignores its percepts* when choosing an action because it knows in advance what they will be. Control theorists call this an **open-loop** system, because ignoring the percepts breaks the loop between agent and environment.

"formulate, search, execute"

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

Figure 3.1 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.

3.1.1 Well-defined problems and solutions

A **problem** can be defined formally by five components:

The **initial state** that the agent starts in (initial state for our agent in Romania might be described as In(Arad)

description of the possible **actions** available to the agent. Given a particular state s, 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)}.)

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 state s.

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.

path in the state space is a sequence of states connected by a sequence of actions.

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)}. Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. (For example, in chess, the goal is to reach a state called "checkmate," where the opponent's king is under attack and can't escape.)

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.

Ex: travel time is of essence - the cost of a path might be its length in kilometers.

cost of a path = *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^1 is denoted by c(s, a, s^1).

Travel: route distances. assume that step costs are nonnegative.

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.
optimal solution has the lowest path cost among all solutions.

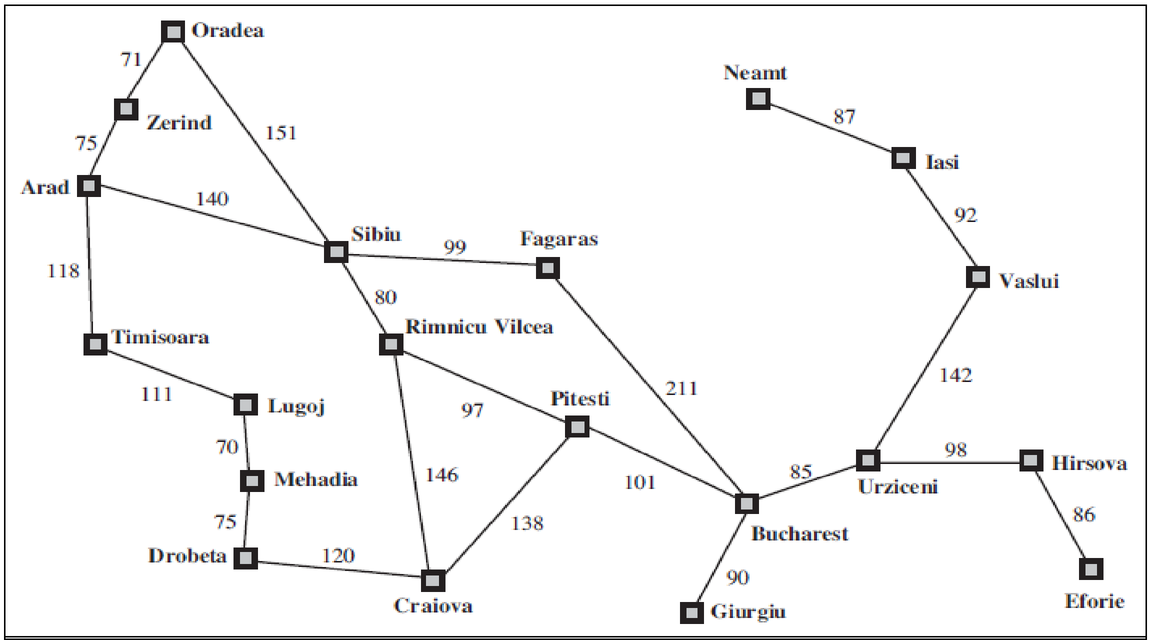


Figure 3.2 A simplified road map of part of Romania.

3.2 EXAMPLE PROBLEMS

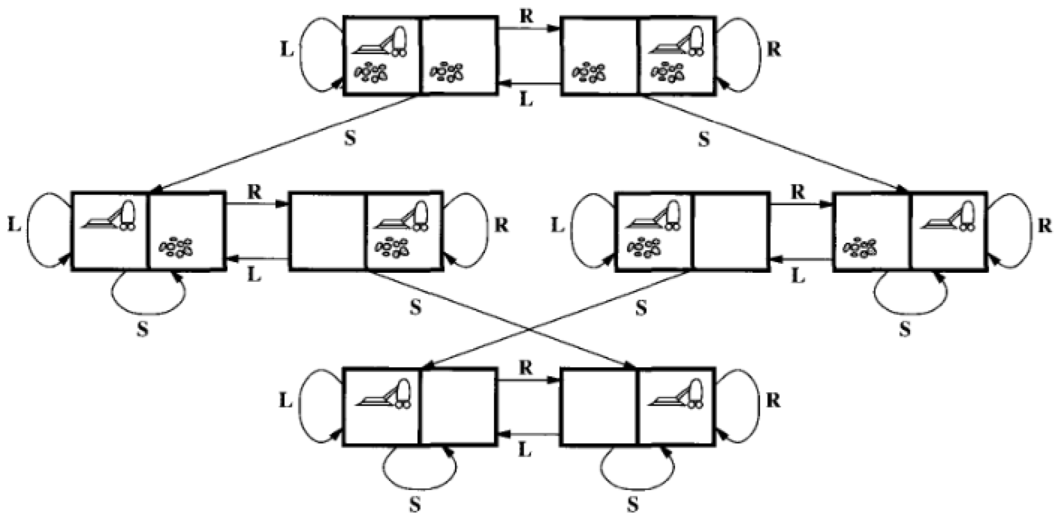
The problem solving approach has been applied to a vast array of task environments. Some best known problems are summarized below. They are distinguished as toy or real-world problems
A **toy problem** is intended to illustrate various problem solving methods. It can be easily used by different researchers to compare the performance of algorithms.
A **real world problem** is one whose solutions people actually care about.

3.2.1 Toy Problems

Vacuum World Example

- **States**: The agent is in one of two locations.,each of which might or might not contain dirt. Thus there are 2 x 2^2 = 8 possible world states.
- **Initial state**: Any state can be designated as initial state.
- **Successor function** : This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in figure 2.3
- **Goal Test** : This tests whether all the squares are clean.
- **Path test** : Each step costs one ,so that the the path cost is the number of steps in the path.

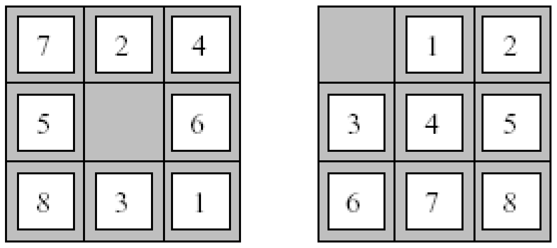
Vacuum World State Space



The 8-puzzle

An 8-puzzle consists of a 3x3 board with eight numbered tiles and a blank space. A tile adjacent to the balank space can slide into the space. The object is to reach the goal state ,as shown in figure 2.4

Example: The 8-puzzle



Start State Goal State

The problem formulation is as follows :

- **States** : A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state** : Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.
- **Successor function** : This generates the legal states that result from trying the four actions(blank moves Left,Right,Up or down).
- **Goal Test** : This checks whether the state matches the goal configuration shown in figure 2.4.(Other goal configurations are possible)
- **Path cost** : Each step costs 1,so the path cost is the number of steps in the path.

  The 8-puzzle belongs to the family **of sliding-block puzzles**,which are often used as test problems for new search algorithms in AI. This general class is known as NP-complete.

  The **8-puzzle** has 9!/2 = 181,440 reachable states and is easily solved.

  The **15 puzzle** (4 x 4 board) has around 1.3 trillion states,an the random instances can be solved optimally in few milli seconds by the best search algorithms.

The **24-puzzle** (on a 5 x 5 board) has around 10^{25} states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

8-queens problem

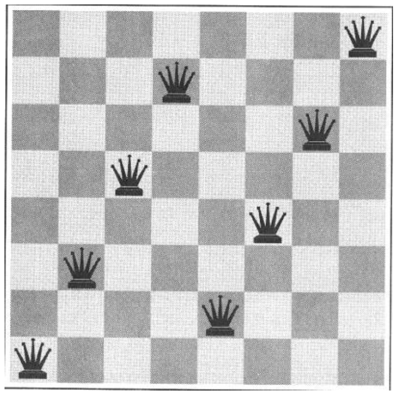
The goal of 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other.(A queen attacks any piece in the same row,column or diagonal).

Figure 2.5 shows an attempted solution that fails: the queen in the right most column is attacked by the queen at the top left.

An **Incremental formulation** involves operators that augments the state description,starting with an empty state.for 8-queens problem,this means each action adds a queen to the state.

A **complete-state formulation** starts with all 8 queens on the board and move 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 :

- **States** : Any arrangement of 0 to 8 queens on board is a state.
- **Initial state** : No queen on the board.
- **Successor function** : Add a queen to any empty square.
- **Goal Test** : 8 queens are on the board,none attacked.

In this formulation,we have 64.63…57 = 3 x 10^{14} possible sequences to investigate.

A better formulation would prohibit placing a queen in any square that is already attacked. :

- **States** : Arrangements of n queens (0 <= n < = 8) ,one per column in the left most columns ,with no queen attacking another are states.
- **Successor function** : Add a queen to any square in the left most empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queen state space from 3 x 10^{14} to just 2057,and solutions are easy to find.

For the 100 queens the initial formulation has roughly 10^{400} states whereas the improved formulation has about 10^{52} states. This is a huge reduction,but the improved state space is still too big for the algorithms to handle.

Infinite State space

Donald Knuth (1964) - how infinite state spaces can arise.

starting with the number 4, a sequence of factorial square root, and floor operations will reach any desired positive integer. For example, we can reach 5 from 4 as follows:

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

The problem definition is very simple:
• **States**: Positive numbers.
• **Initial state**: 4.
• **Actions**: Apply factorial, square root, or floor operation (factorial for integers only).
• **Transition model**: As given by the mathematical definitions of the operations.
• **Goal test**: State is the desired positive integer.
(To our knowledge there is no bound on how large a number might be constructed in the process of reaching a given target—for example, the number 620,448,401,733,239,439,360,000 is generated in the expression for 5—so the state space for this problem is infinite. Such state spaces arise frequently in tasks involving the generation of mathematical expressions, circuits, proofs, programs, and other recursively defined objects.)

3.2.2 Real-World Problems

ROUTE-FINDING PROBLEM

Route-finding problem is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications,such as routing in computer networks,military operations planning,and air line travel planning systems.

AIRLINE TRAVEL PROBLEM

The **airline travel problem** is specifies as follows **:**

- **States :** Each is represented by a location(e.g.,an airport) and the current time.
- **Initial state :** This is specified by the problem.
- **Successor function :** This returns the states resulting from taking any scheduled flight(further specified by seat class and location),leaving later than the current time plus the within-airport transit time,from the current airport to another.
- **Goal Test :** Are we at the destination by some prespecified time?
- **Path cost :** This depends upon the monetary cost,waiting time,flight time,customs and immigration procedures,seat quality,time of dat,type of air plane,frequent-flyer mileage awards, and so on

TOURING PROBLEMS

Touring problems are closely related to route-finding problems,but with an important difference.
Consider for example,the problem,"Visit every city at least once" as shown in Romania map.
As with route-finding the actions correspond to trips between adjacent cities. The state space, however,is quite different.
The initial state would be "In Bucharest; visited{Bucharest}".
A typical intermediate state would be "In Vaslui;visited {Bucharest,Urziceni,Vaslui}".
The goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

THE TRAVELLING 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**. Enormous efforts have been expended to improve the capabilities of TSP algorithms. These algorithms are also used in tasks such as planning movements of **automatic circuit-board drills** and of **stocking machines** on shop floors.

VLSI layout

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 is split into two parts : **cell layout** and **channel routing**.

ROBOT navigation

ROBOT navigation is a generalization of the route-finding problem. Rather than a discrete set of routes,a robot can move in a continuous space with 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 also must be controlled,the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

AUTOMATIC ASSEMBLY SEQUENCING

The example includes assembly of intricate objects such as electric motors. The aim in assembly problems is to find the order in which to assemble the parts of some objects. If the wrong order is choosen,there will be no way to add some part later without undoing somework already done. Another important assembly problem is protein design,in which the goal is to find a sequence of Amino acids that will be fold into a three-dimensional protein with the right properties to cure some disease.

INTERNET SEARCHING

In recent years there has been increased demand for software robots that perform Internet searching.,looking for answers to questions,for related information,or for shopping deals. The searching techniques consider internet as a graph of nodes(pages) connected by links.

3.3 Searching For Solutions

SEARCH TREE

Having formulated some **problems**,we now need to **solve** them. This is done by a **search** through the **state space**. A **search tree** is generated by the **initial state** and the **successor function** that together define the **state space**. In general,we may have a *search graph* rather than a *search tree*,when the same state can be reached from multiple paths.

Figure 1.23 shows some of the expansions in the search tree for finding a route from Arad to Bucharest.

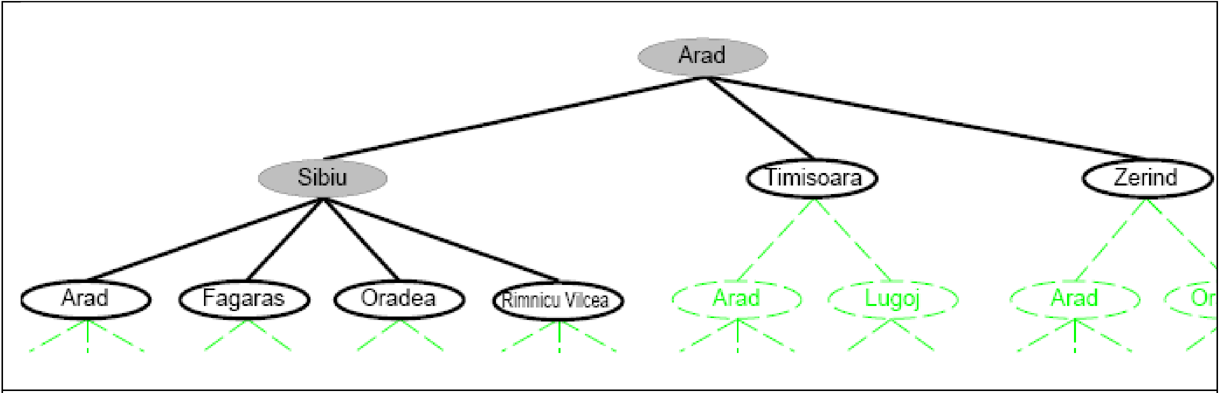


Figure 1.23 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded.; nodes that have been generated but not yet expanded are outlined in bold;nodes that have not yet been generated are shown in faint dashed line

The root of the search tree is a **search node** corresponding to the initial **state**,In(Arad). The first step is to test whether this is a **goal state**. The current state is expanded by applying the successor function to the current state,thereby generating a new set of states. In this case,we get three new states: In(Sibiu),In(Timisoara),and In(Zerind). Now we must choose which of these three possibilities to consider further. This is the essense of search- following up one option now and putting the others aside for latter,in case the first choice does not lead to a solution.
Search strategy . The general tree-search algorithm is described informally in Figure 1.24
.

<u>Tree Search</u>

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

Figure 1.24 An informal description of the general tree-search algorithm

The choice of which state to expand is determined by the **search strategy**. There are an infinite number paths in this state space ,so the search tree has an infinite number of **nodes.**
A **node** is a data structure with five components :

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

It is important to remember the distinction between nodes and states. A node is a book keeping data structure used to represent the search tree. A state corresponds to configuration of the world.

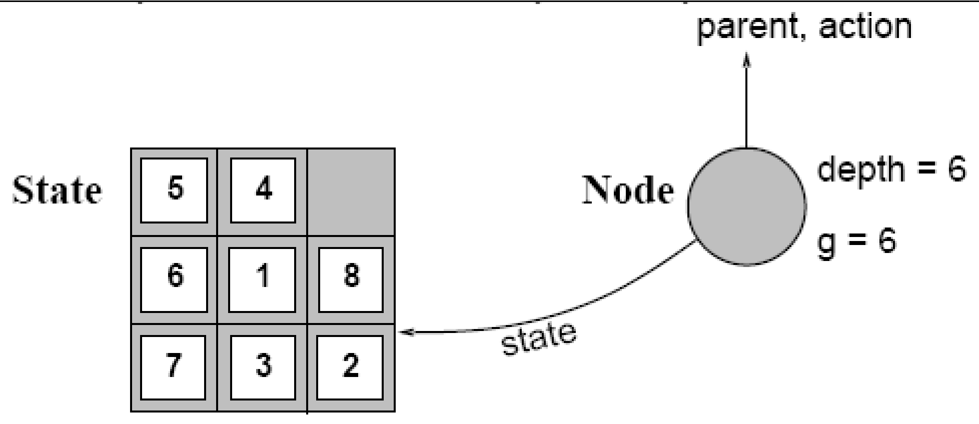


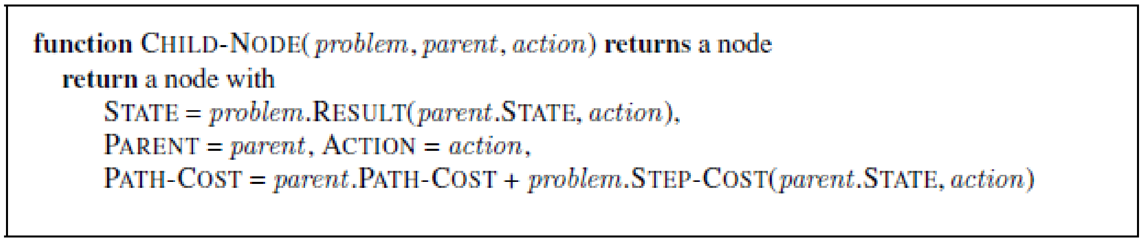
Figure 1.25 Nodes are data structures from which the search tree is constructed. Each has a parent,a state, Arrows point from child to parent.

Fringe

Fringe is a collection of nodes that have been generated but not yet been expanded. Each element of the fringe is a leaf node,that is,a node with no successors in the tree. The fringe of each tree consists of those nodes with bold outlines. The collection of these nodes is implemented as a **queue.**

The operations specified in Figure 1.26 on a queue are as follows:

- **MAKE-QUEUE(element,…)** creates a queue with the given element(s).
- **EMPTY?(queue)** returns true only if there are no more elements in the queue.
- **FIRST(queue)** returns FIRST(queue) and removes it from the queue.
- **INSERT(element,queue)** inserts an element into the queue and returns the resulting queue.
- **INSERT-ALL(elements,queue)** inserts a set of elements into the queue and returns the resulting queue.



```
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)
```

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if EMPTY?(fringe) then return failure
      node ← REMOVE-FIRST(fringe)
      if GOAL-TEST[problem] applied to STATE[node] succeeds
          then return SOLUTION(node)
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes
  successors ← the empty set
  for each ⟨action, result⟩ in SUCCESSOR-FN[problem](STATE[node]) do
      s ← a new NODE
      STATE[s] ← result
      PARENT-NODE[s] ← node
      ACTION[s] ← action
      PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
      DEPTH[s] ← DEPTH[node] + 1
      add s to successors
  return successors
```

3.3.2 MEASURING PROBLEM-SOLVING PERFORMANCE

The output of problem-solving algorithm is either failure or a solution.(Some algorithms might struck in an infinite loop and never return an output.

The algorithm’s performance can be measured in four ways :

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

3.4 UNINFORMED SEARCH STRATEGIES

Uninformed Search Strategies have no additional information about states beyond that provided in the **problem definition**.

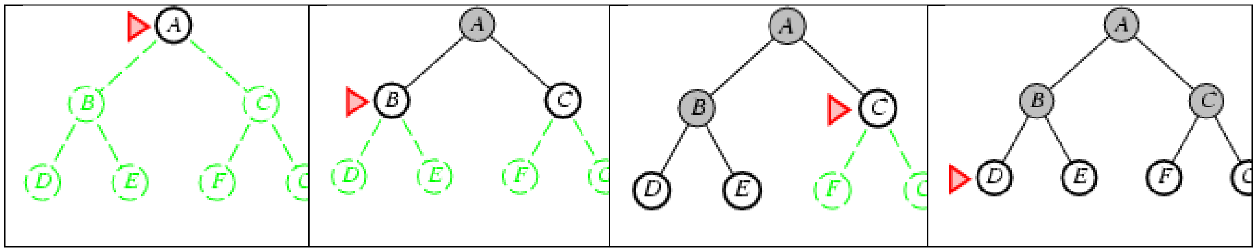
Strategies that know whether one non goal state is “more promising” than another are called **Informed search or heuristic search** strategies.

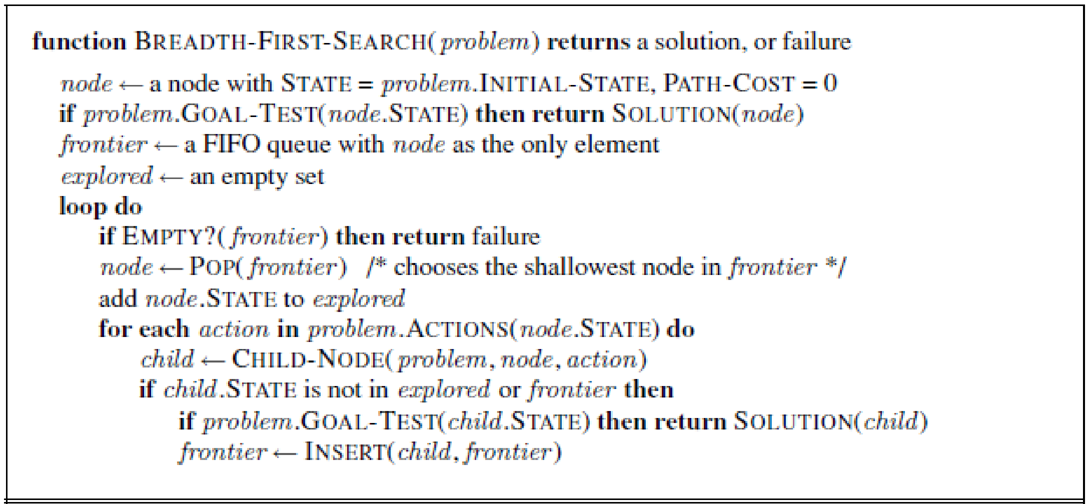
There are five uninformed search strategies as given below.

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional search

3.4.1 Breadth-first search

One simple search strategy is a **breadth-first search.** In this strategy, the root node is expanded first, then all the nodes generated by the root node are expanded next, and then *their* successors, and so on. In general, all the nodes at depth *d* in the search tree are expanded before the nodes at depth *d* + 1.
BFS- implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first.
Calling TREE-SEARCH(problem,FIFO-QUEUE()) ->bfs
FIFO queue puts the newly generated successors at the end of the queue, which means that shallow nodes are expanded before deeper nodes.





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

Figure 3.11 Breadth-first search on a graph.

If 'F' is a goal : FIFO queue

Iteration	**Fringe**	**Closed list (Visited)**	**Goal test**
0	**A**		**A×**
1	**B,C**	**A**	**B×**
2	**C,D,E**	**A,B**	**C×**
3	**D,E,F,G**	**A,B,C**	**D×**
4	**E,F,G**	**A,B,C,D**	**E×**
5	**F,G**	**A,B,C,D,E**	**F✓**

Properties of breadth-first-search

Evaluate using criteria : if the shallowest goal node is at some finite depth d, the branching factor b is finite).-> complete
(The shallowest goal node is not necessarily the optimal one; Technically breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node. (For example, when all actions have the same cost.)- optimal

Every state has b successors, root generate b nodes, the next b more nodes $b+b^2\ldots+b^d)=O(b^d)$ -> memory
Every node should be in memory since, part or ancestor of fringe. ->space

Shallow nodes are expanded before deeper nodes.

Time complexity for BFS

Assume every state has b successors. The root of the search tree generates b nodes at the first level,each of which generates b more nodes,for a total of b^2 at the second level. Each of these generates b more nodes,yielding b^3 nodes at the third level,and so on. Now suppose,that the solution is at depth d. In the worst case,we would expand all but the last node at level d.

Then the total number of nodes generated is

$b + b^2 + b^3 + \ldots + b^d = O(b^{d+1}).$

Every node that is generated must remain in memory,because it is either part of the fringe or is an ancestor of a fringe node. The space compleity is,therefore ,the same as the time complexity

Disadvantage − Since each level of nodes is saved for creating next one, it consumes a lot of memory space. Space requirement to store nodes is exponential.

Its complexity depends on the number of nodes. It can check duplicate nodes

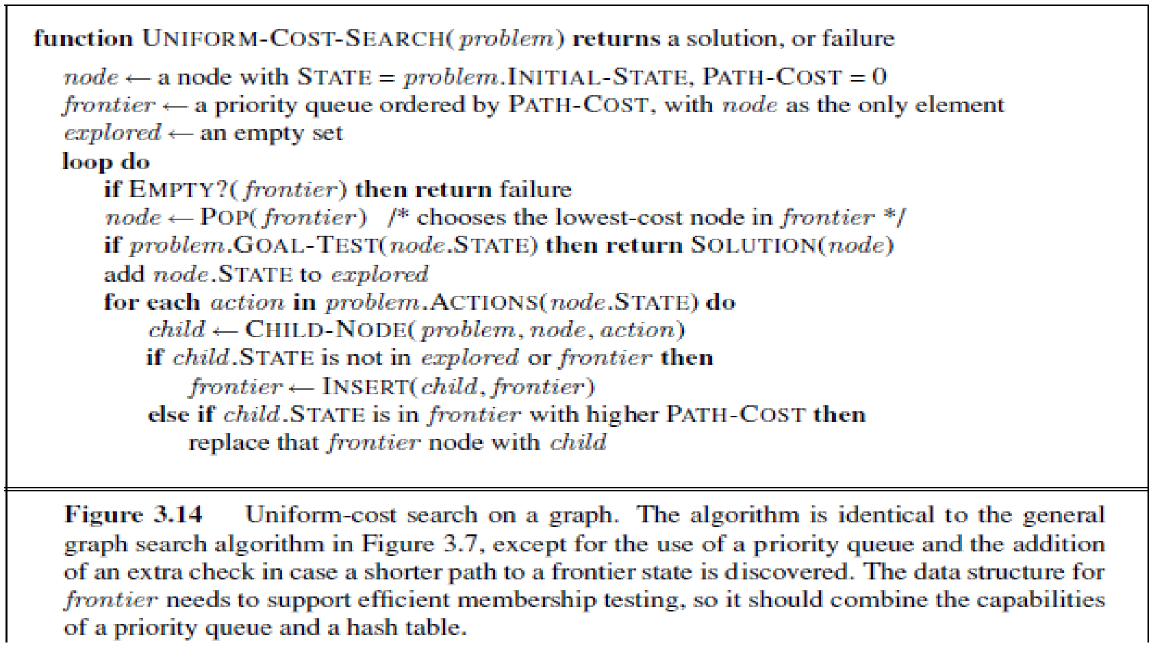
Breadth-first search is useful when

- space is not a problem;
- few solutions may exist, and at least one has a short path length; and
- infinite paths may exist, because it explores all of the search space, even with infinite paths.

It is a poor method when all solutions have a long path length or there is some heuristic knowledge available. It is not used very often because of its space complexity.

3.4.2 Uniform-Cost Search

Instead of expanding the shallowest node,**uniform-cost search** expands the node n with the lowest path cost. uniform-cost search does not care about the number of steps a path has,but only about their total cost.



```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← 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 ← CHILD-NODE(problem, node, action)
         if child.STATE is not in explored or frontier then
             frontier ← INSERT(child, frontier)
         else if child.STATE is in frontier with higher PATH-COST then
             replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. 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.

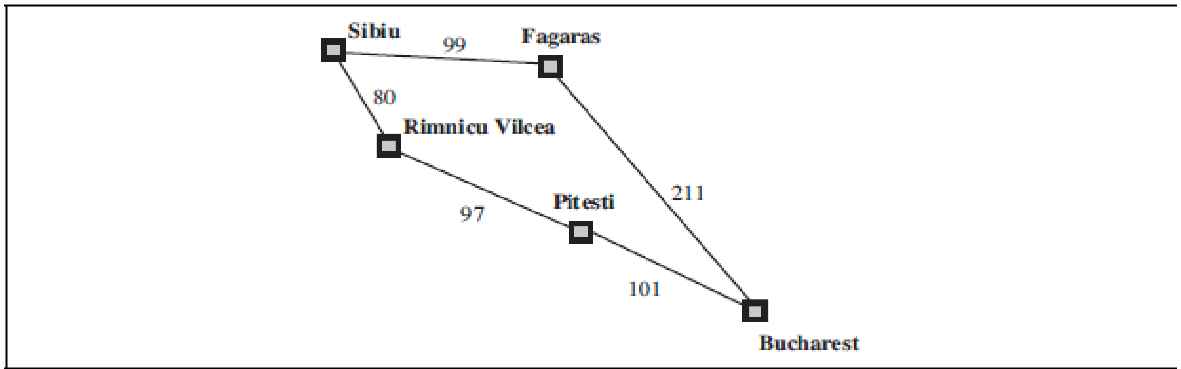


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

Both of these modifications come into play in the example shown in Figure 3.15, where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97=177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99+211=310.

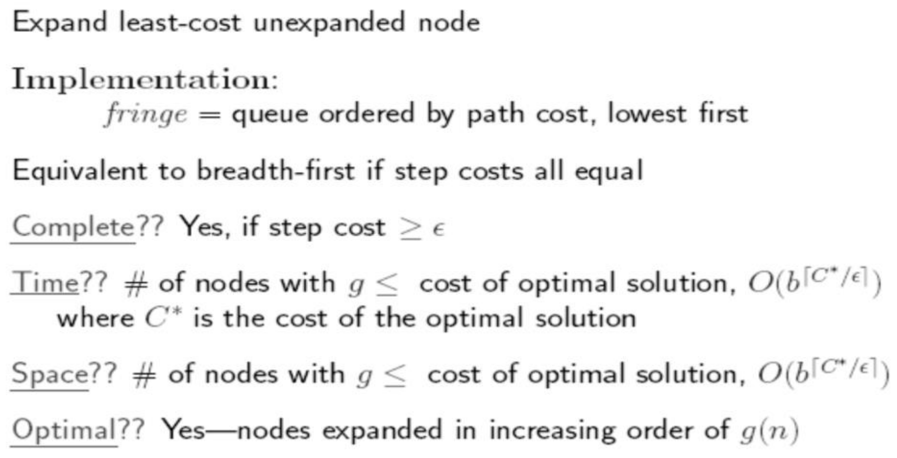
Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+97+101= 278. Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

If 'B' is a goal

Fringe	**Goal test**	**Close list**	**Comparison**
S	S×		
R,F **80,99**	R×	S	S-R<S-F; 80<99; so select S-R
P,F **80+97,99**	F×	S,R	S->R->P=177>S-F 177>99; goto F
P,B **80+97,99+211**	P×	S,R,F	S->F->B=310> S->R->P=177; goto P Though goal is reached still searches for least cost
B,B **117+101,99+211**	B✓	S,R,F,P	S->R->P->B=278 < S->F->B=310; Goal reached in less cost

Note that if all step costs are equal, this is identical to breadth-first search.
Uniform-cost search does not care about the *number* of steps a path has, but total cost.
Therefore, stuck in an infinite loop if it ever expands a node - zero-cost action leading back to the same state (for example, a *NoOp* action).
completeness provided the cost of every step is greater than or equal to some small positive constant **c.** **->**to ensure ***optimality.***
It means that the cost of a path always increases as we go along the path. -algorithm expands nodes in order of increasing path cost.
Eg: We recommend trying the algorithm out to find the shortest path to Bucharest.
Uniform-cost search is guided by path costs rather than depths, so its complexity cannot easily be characterized in terms of *b* and d.

Instead, let C* be the cost of the optimal solution, and assume that every action costs at least *e*. Then the algorithm's worst-case time and space complexity is $O(b^{1+[C^*/e]})$, which can be much greater than b^d.



Expand least-cost unexpanded node

Implementation:
$fringe$ = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

3.4.3 Depth-First-Search

- always expands the *deepest* node in the current fringe 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 fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.
- implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.
. (A recursive depth-first algorithm incorporating a depth limit is shown in Figure 3.13.)
modest memory requirements. It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
Once a node has been expanded, it can be removed from memory as soon
as all its descendants have been fully explored.
branching factor *b* and maximum depth *m*, - exploring nodes $O(b^m)$

The drawback –DFS: can make a wrong choice and get stuck going down a very long (or even infinite) path when a different choice would lead to a solution near the root of the search tree.
(For example, in Figure 3.12, depth-first search will explore the entire left subtree even if node C is a goal node.
If node J were also a goal node, then depth-first search would return it as a solution; hence, depth-first search is **not optimal**.
If the left subtree were of unbounded depth but contained no solutions, depth-first search would never terminate; hence, it is **not complete**. (In the worst case, depth-first search will generate all of the O(bm) nodes in the search tree, where m is the maximum depth of any niode. Note that m can be much larger than d (the depth of the shallowest solution), and is infinite if the tree is unbounded.)

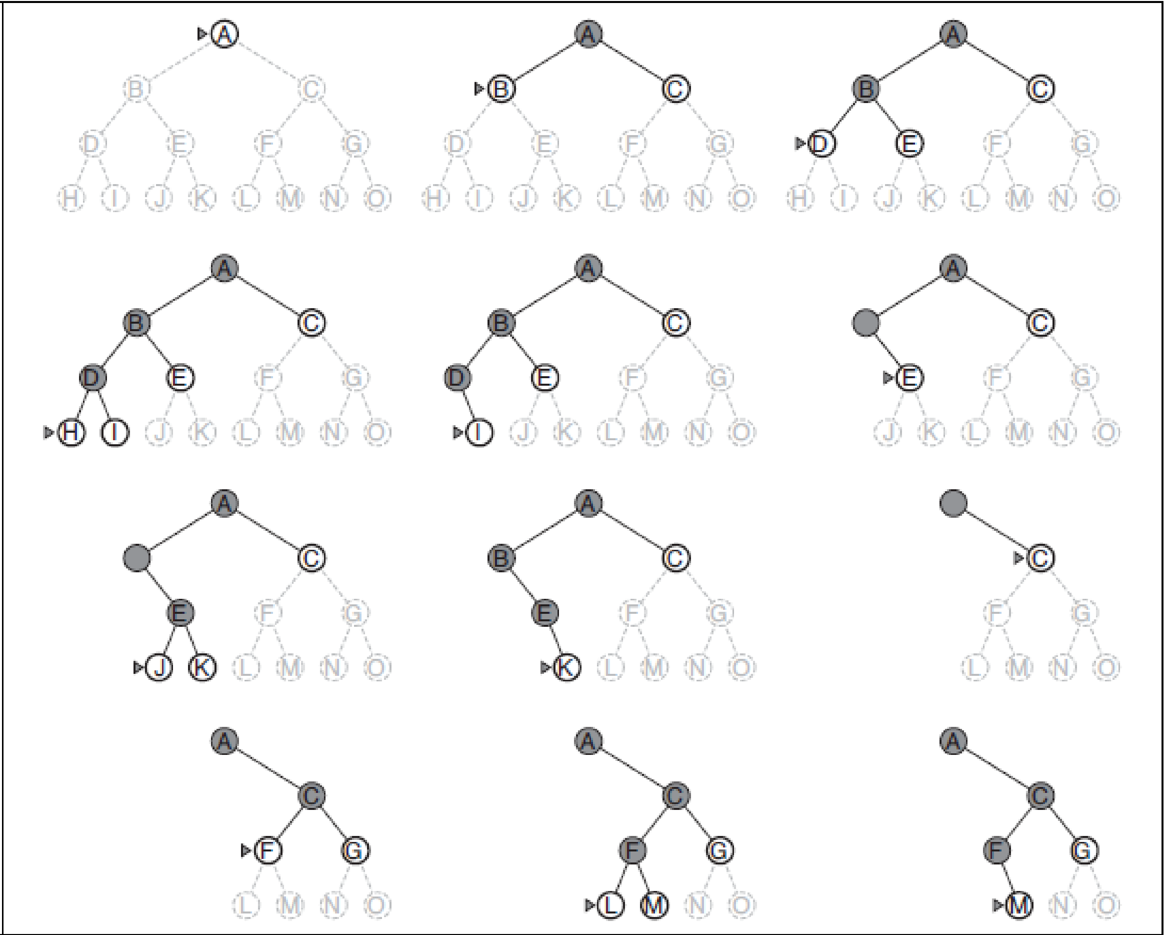


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

If 'M' is a goal : LIFO queue

Iteration	Fringe	Closed list (Visited)	Goal test	Removed nodes from memory	Back track to
0	**A**		**A ×**		
1	**C,B**	**A**	**B×**		
2	**C,E,D**	**A,B**	**D×**		
3	**C,E,I,H**	**A,B,D**	**H×**	**H**	**D**
4	**C,E,I**	**A,B,D**	**I×**	**I**	**D**
5	**C,E**	**A,B**	**E×**	**D**	**B**
6	**C,K,J**	**A,B,E**	**J×**	**J**	**E**
7	**C,K**	**A,B,E**	**K×**	**K**	**E**
	C	**A,B**		**E**	**B**
	C	**A**		**B**	**A**
8	**C**	**A**	**C×**		
9	**G,F**	**A,C**	**F×**		
10	**G,M,L**	**A,C,F**	**L×**	**L**	**F**
	G	**A,C,F**	**M- Goal reached ✓**		

Now third column A-C-F set the path to Goal M.

Drawback of Depth-first-search

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree. For example ,depth-first-search will explore the entire left subtree even if node C is a goal node.

Depth-first search is appropriate when either

- space is restricted;
- many solutions exist, perhaps with long path lengths, particularly for the case where nearly all paths lead to a solution;
- It is a poor method when it is possible to get caught in infinite paths; this occurs when the graph is infinite or when there are cycles in the graph; or
- Solutions exist at shallow depth, because in this case the search may look at many long paths before finding the short solutions.

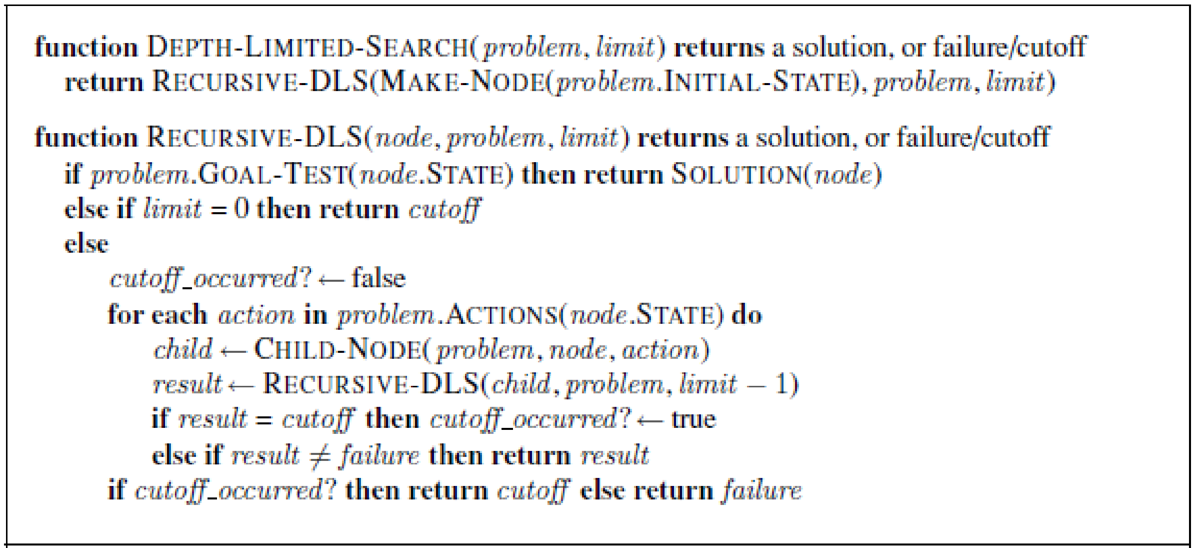
Compare BFS and DFS

1.	BFS stands for Breadth First Search	DFS stands for Depth First Search.
2.	FIFO Queue for traversal and to find goal	Uses LIFO, Stack for traversal and to find goal
3.	more suitable for searching vertices which are closer to the given source.	more suitable when there are solutions away from source.
4.	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop.
5.	Here, siblings are visited before the children	Here, children are visited before the siblings
6.	The Time complexity of BFS is $O(b^{d+1})$	The Time complexity of DFS is $O(b^m)$
7.	Space Complexity is $O(b^{d+1})$ BFS has to keep track of all the nodes on the same level	Space Complexity is O(bm) DFS needs less memory as it only has to keep track of the nodes in a chain from the top to the bottom
8.	Complete if branching factor is finite	DFS **can get stuck in an infinite loop**, which is why it is not "complete". Graph search keeps track of the nodes it has already searched, so it can avoid following infinite loops. "Redundant paths" are different paths which lead from the same start node to the same end node..

9.	Optimal , if step cost are constant	**DFS is not optimal**, -the number of steps in reaching the solution, or the cost spent in reaching it is high.

Depth Limited Search

The problem of unbounded trees can be alleviated by supplying depth-first search with a predetermined
depth limit *l*.
-That is, nodes at depth *l* are treated as if they have no successors- **depth-limited search**
-solves the infinite-path problem.
But incompleteness if we choose *l* < d, that is, the shallowest goal is beyond the depth limit. (This is not unlikeky when d is unknown.)
Depth-limited search will also be nonoptimal if we choose *l* > d.
 Its time complexity is $O(b^l)$ and its space complexity is $O(bl)$.
Depth-first search can be viewed as a special case of depth-limited search with $l=\alpha$
(Sometimes, depth limits can be based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solutiom, it must be of length 19 at the longest, so *l* = 19 is a possible choice.)
 But any city can be reached from any other city ity at most 9 steps. This number, known as the **diameter** of the state space, gives us a better depth limit,-> more efficient depth-limited search.
Notice that depth-limited search can terminate with two kinds of failure:
 the standard failure value indicates no solution;
the *cutclff* value indicates no solution within the depth limit.



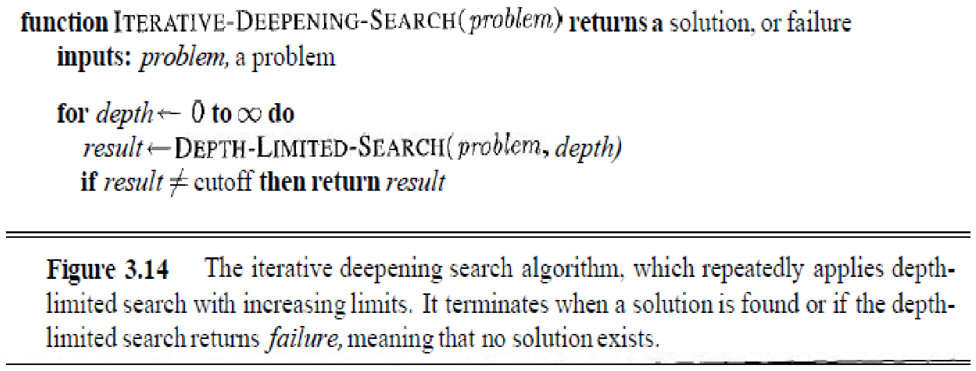
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

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? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

3.4.5 Iterative deepening depth-first search- combination with depth-first search, that finds the best depth limit.
-gradually increasing the limit-first 0, then 1, then 2, and so on-until a goal is found (depth limit reaches d, the depth of the shallowest goal node)
-benefits of depth-first + breadth-first search
Like DFS-memory requirements are very modest:O(bd) to be precise

Like BFS, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node.



```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  inputs: problem, a problem

  for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Figure 3.14 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.

Iterative search is not as wasteful as it might seem. may seem wasteful, because states are generated multiple times. But not- reason-(with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.)

(In IDS, the nodes on the bottom level (depth d) are generated once, the next to bottom level are generated twice, and so on, up to the children of the root, which are generated d times.)So the total number of nodes generated is
$N(IDS) = (d)\, b + (d - l)b^2 + . . . + (1)\, b^d$,

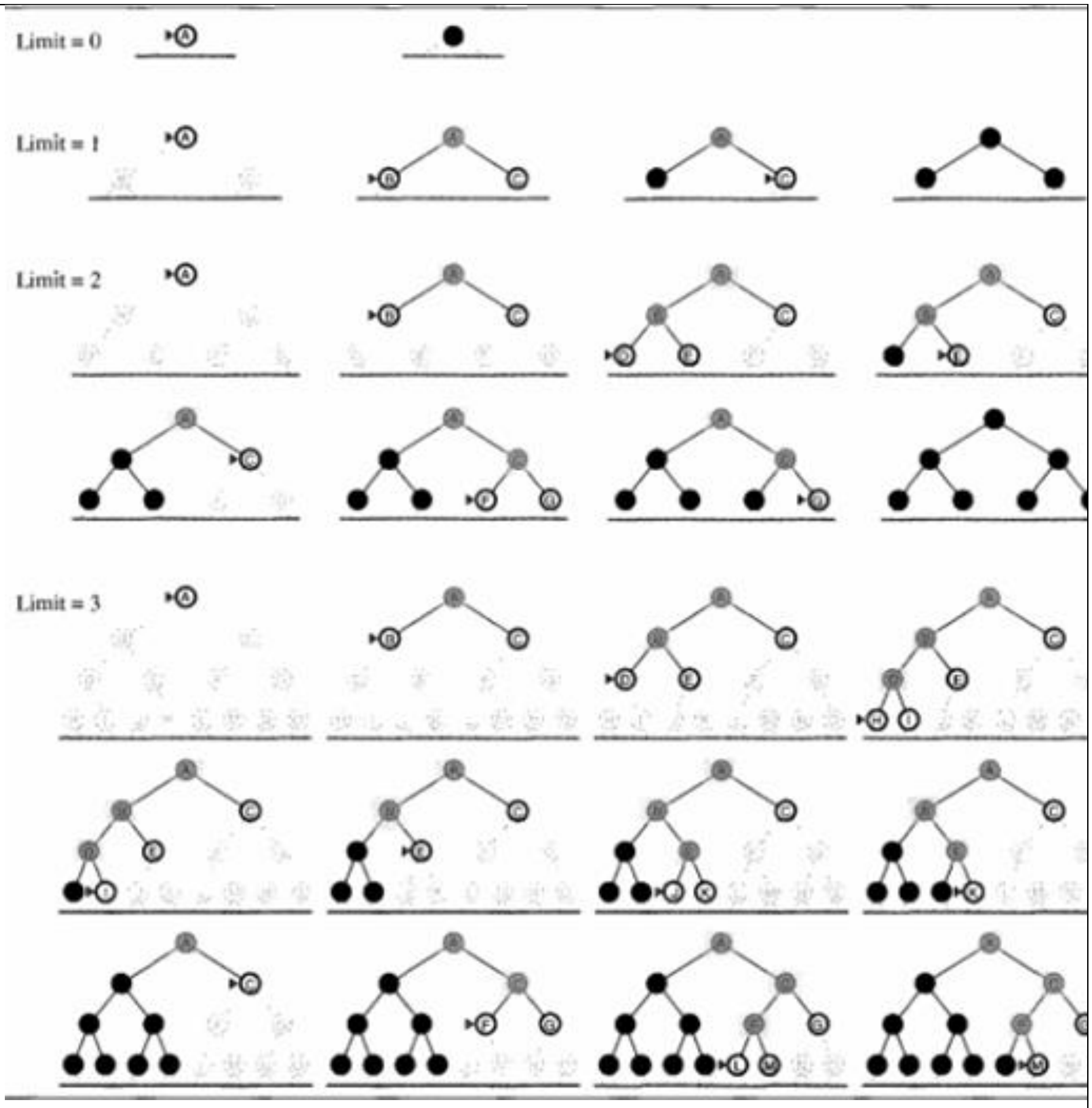
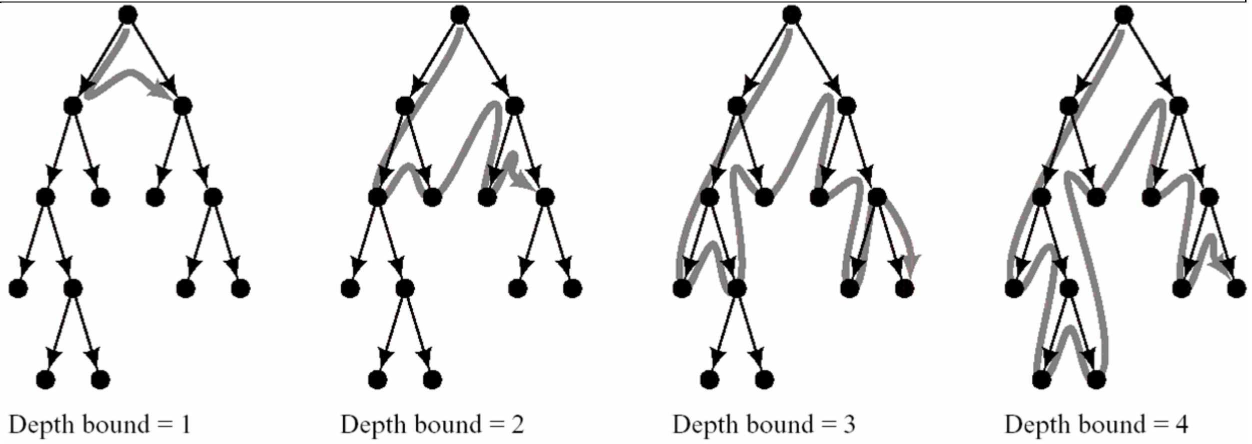
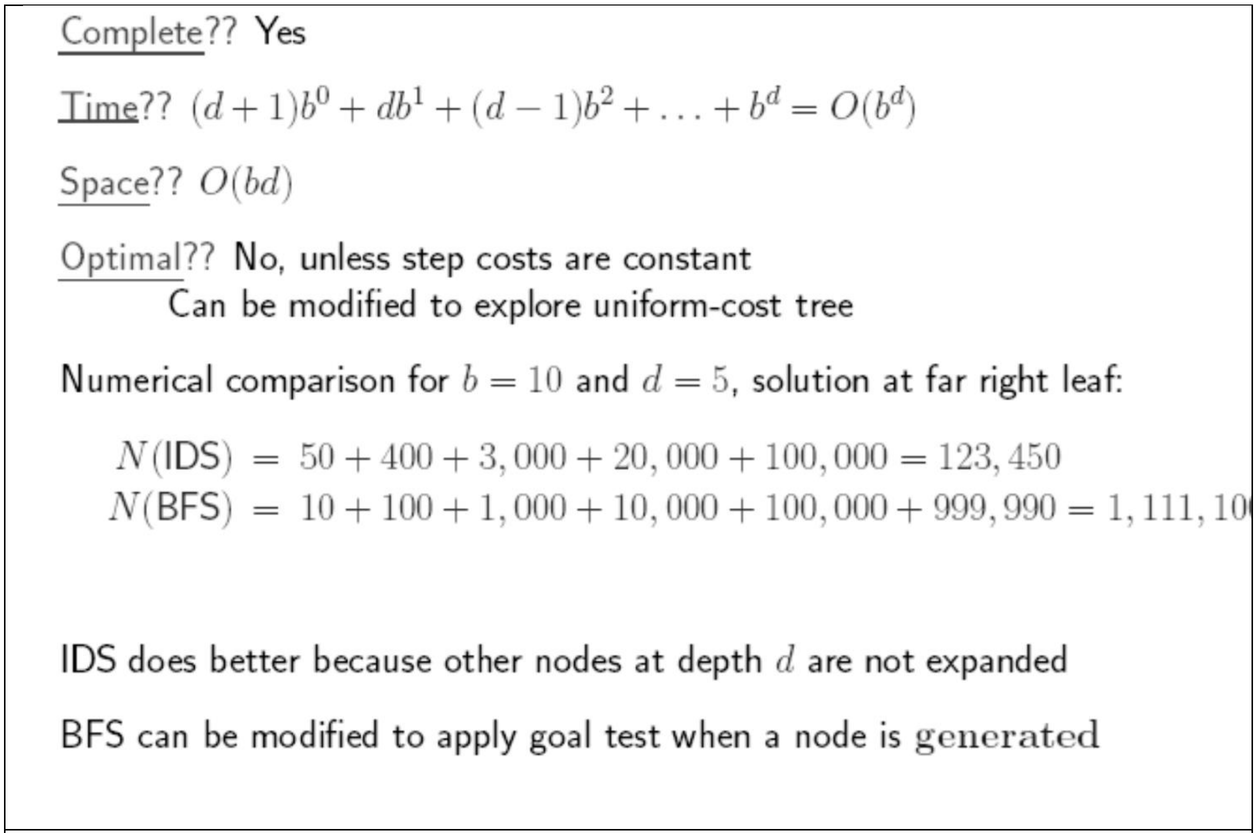


Figure 1.34 Four iterations of iterative deepening search on a binary tree



Properties of iterative deepening search



Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? No, unless step costs are constant
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,10$$

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is generated

Figure 1.36

In general,iterative deepening is the prefered uninformed search method when there is a large search space and the depth of solution is not known.

3.4.6 Bidirectional search

run two simultaneous searches-one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle.

The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d, (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.)

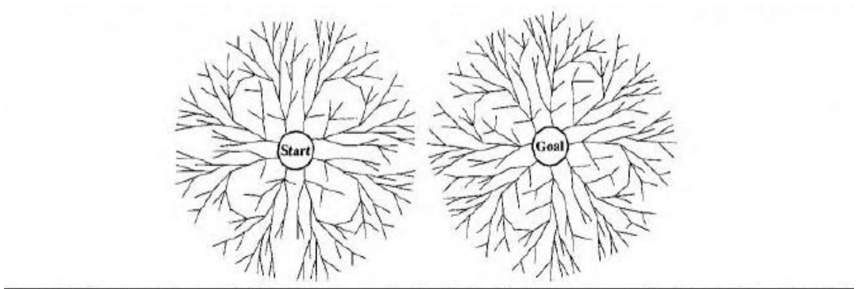
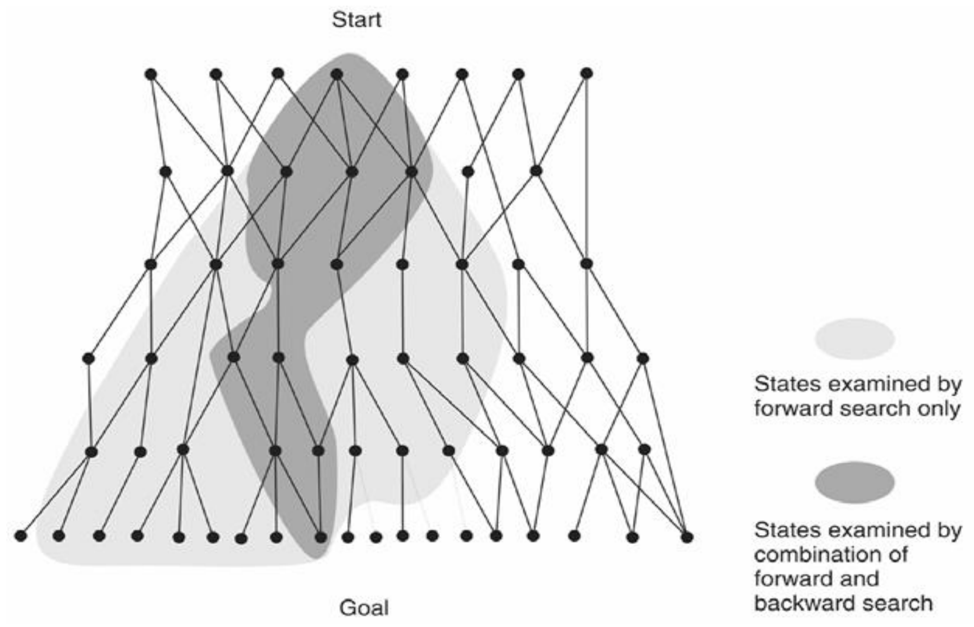


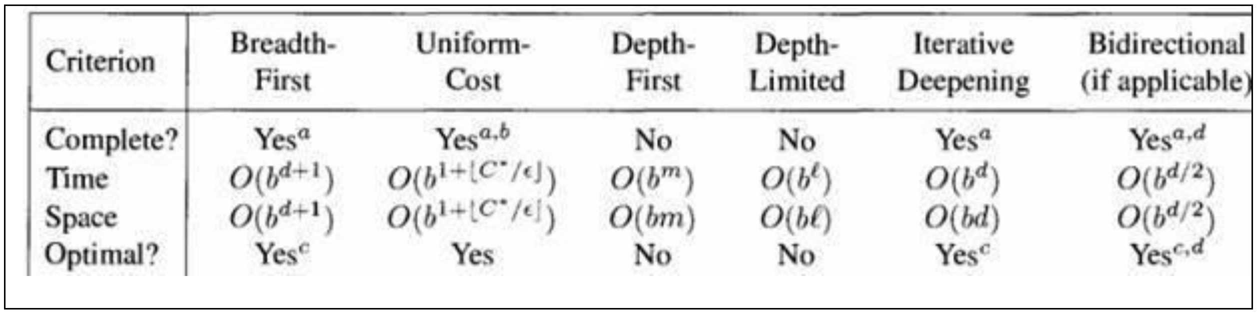
Figure 3.16 A schematic view of a bidirectional search that is about to succeed, when a branch from the start node meets a branch from the goal node.

- implemented by having one or both of the searches check each node before it is expanded to see if it is in the fringe of the other search tree; if so, a solution has been found.

Eg: solution depth d = 6, and each direction runs BFS one node at a time, then in the worst case the two searches meet when each has expanded all but one of the nodes at depth 3.
For b = 10, this means a total of 22,200 node generations, compared with 1 1,111,100 for a standard breadth-first search.
Checking a node for membership in the other search tree can be done in constant time with a hash table,
so the time complexity of bidirectional search is O($b^{d/2}$).
At least one of the search trees must be kept in memory so that the membership check can be done, hence the space complexity is also O($b^{d/2}$). - weakness of bidirectional search.
The algorithm is complete and optimal (for uniform step costs) if both searches are breadth-first; (other combinations may sacrifice completeness, optimality, or both.)
(The most difficult case for bidirectional search is when the goal test gives only an implicit description of some possibly large set of goal states-for example, all the states satisfy ing the "checkmate" goal test in chess. A backward search would need to construct compact descriptions of "all states that lead to checkmate by move *ml"* and so on; and those descriptions would have to be tested against the states generated by the forward search. There is no general way to do this efficiently)
Bidirectional search can sometimes lead to finding a solution more quickly. The reason can be seen from inspecting the following figure.


Start
Goal
States examined by forward search only
States examined by combination of forward and backward search

3.4.7 Comparing Uninformed Search Strategies



Criterion	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	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 1.38 Evaluation of 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 >= E for positive E; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.