Artificial Intelligence & Neural Network

Module 1
Introduction

...

- Why study AI?
- What is AI?
- The Turing test
- Rationality
- Branches of AI
- Brief history of AI
- Challenges for the future
- What is an intelligent agent?
- Doing the right thing (rational action)
- Performance measure
- Autonomy
- Environment and agent design.
- Structure of Agents
- Agent types.

Introduction

Why AI?

- AI adds intelligence to existing products
- AI adapts through progressive learning algorithms
- AI analyses more and deeper data
- AI achieves incredible accuracy
- AI gets the most out of data

What is AI?

- Humans – Homo Sapiens- man the Wise
- Intelligence: 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
- The field of artificial intelligence, or AI, attempts not just to understand but also to build intelligent entities. It is one of the newest fields in science and engineering.
- Work started in earnest soon after World War II, and the name itself was coined in 1956.
- AI has openings for several full-time Einsteins and Edisons.
- 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.
- AI is relevant to any intellectual task; it is truly a universal field.

What is AI?

Views of AI fall into four categories:

- The definitions of Thinking Humanly and Thinking Rationality are concerned with thought processes and reasoning.
- The definitions of Acting Humanly and Acting Rationally are address behaviour.
- The definitions of Thinking Humanly and Acting Humanly measure success in terms of fidelity to human performance.
- The definitions of Thinking Rationally and Acting Rationally measure against an ideal performance measure, called rationality.
- **A system is rational if it does the "right thing," given what it knows.**

Thinking Humanly	**Thinking Rationally**
"The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"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)	"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)

Figure 1.1 Some definitions of artificial intelligence, organized into four categories.

Acting humanly: The Turing Test approach

- The Turing Test, proposed by Alan Turing (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
- Total Turing Test includes a video signal so that the interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects "through the hatch."
- To pass the total Turing Test, the computer will need
  - **computer vision** to perceive objects,
  - **robotics** to manipulate objects and move about

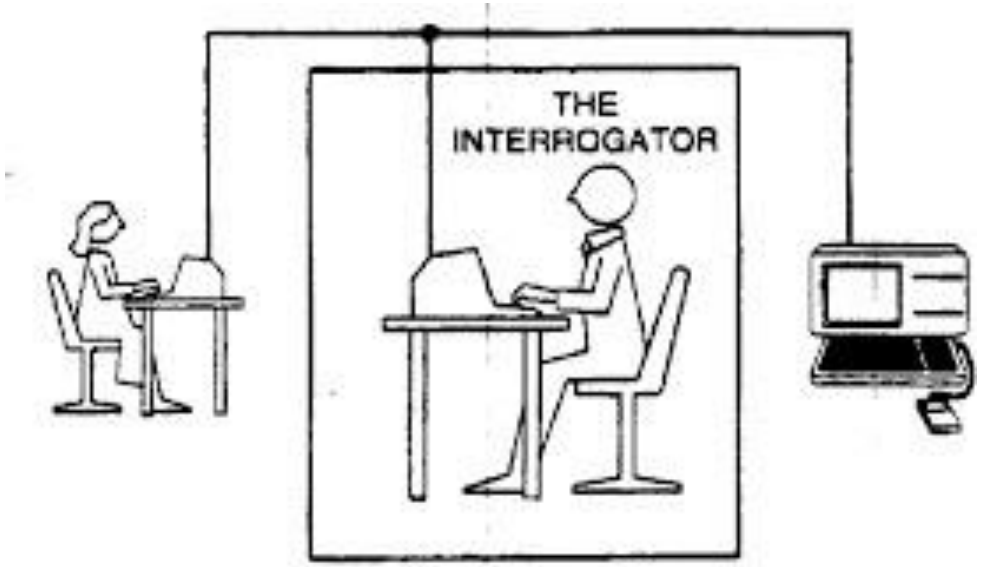
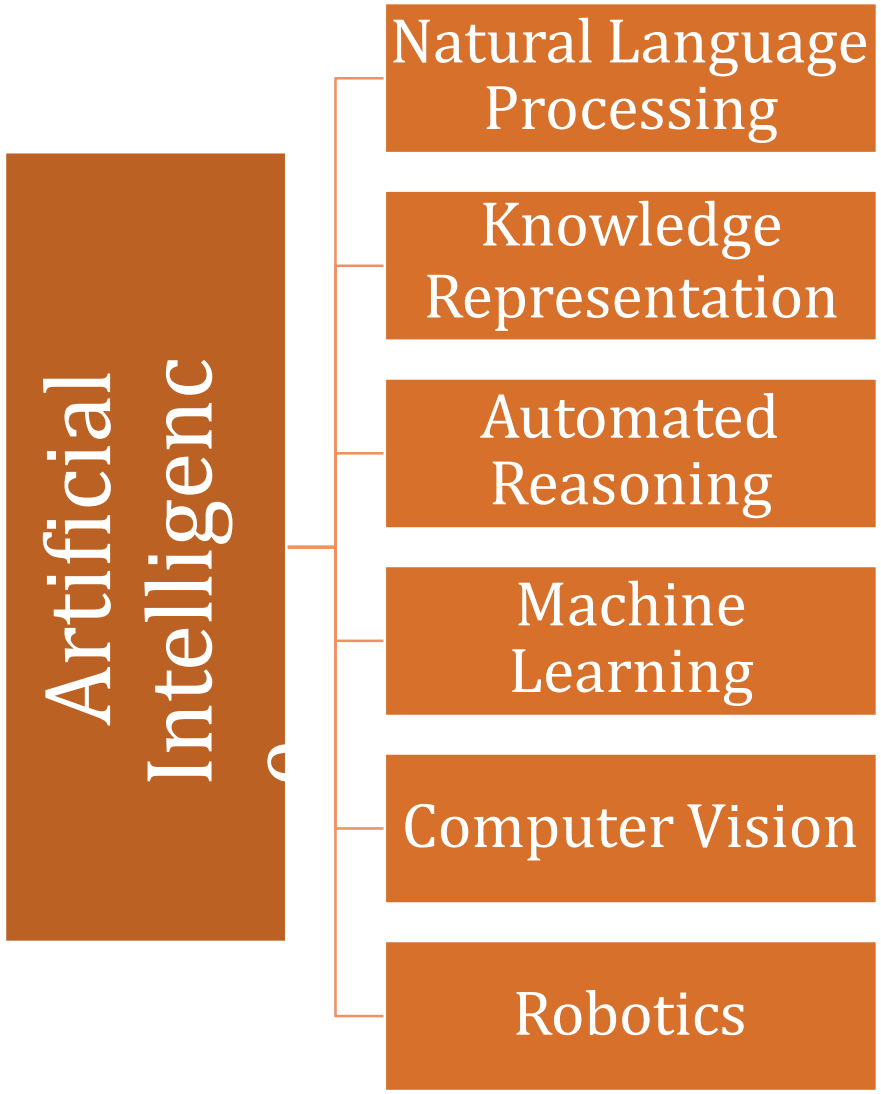


Figure 1.1 The Turing test.

Acting humanly: The Turing Test approach



Thinking humanly: The cognitive modeling approach

We need to get inside the actual workings of human minds.

- Introspection—trying to catch our own thoughts as they go by;
- Psychological experiments—observing a person in action;
- 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.
- The interdisciplinary field of cognitive science brings together computer models from AI and experimental techniques from psychology to construct precise and testable theories of the human mind

Thinking rationally: The “laws of thought” approach

- By 1965, programs existed that could, in principle, solve any solvable problem described in logical notation.
- The so-called logicist tradition within artificial intelligence hopes to build on such programs to create intelligent systems.
- There are two main obstacles to this approach
  - First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain.
  - Second, there is a big difference between solving a problem “in principle” and solving it in practice

Acting rationally: The rational agent approach

- An agent is just something that acts
- All computer programs do something, but computer agents are expected to do more:
  - operate autonomously,
  - perceive their environment,
  - persist over a prolonged time period
  - adapt to change
  - 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" approach to AI, the emphasis was on correct inferences
- Making correct inferences is sometimes part of being a rational agent, because one way to act rationally is to reason logically to the conclusion that a given action will achieve one's goals and then to act on that conclusion.
- On the other hand, correct inference is not all of rationality; in some situations, there is no provably correct thing to do, but something must still be done.

Acting rationally: The rational agent approach

- There are also ways of acting rationally that cannot be said to involve inference. For example, recoiling from a hot stove is a reflex action that is usually more successful than a slower action taken after careful deliberation.
- The rational-agent approach has two advantages over the other approaches:
  - It is more general than the “laws of thought” approach because correct inference is just one of several possible mechanisms for achieving rationality.
  - It is more amenable to scientific development than are approaches based on human behavior or human thought.

Artificial Intelligence & Neural Network

Chapter -2
Intelligent Agents

Agents and Environments

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

- A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators.
- A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.
- A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.
- **PERCEPT:** We use the term percept to refer to the agent's perceptual inputs at any given instant
- An agent's percept sequence is the complete history of everything the agent has ever perceived.

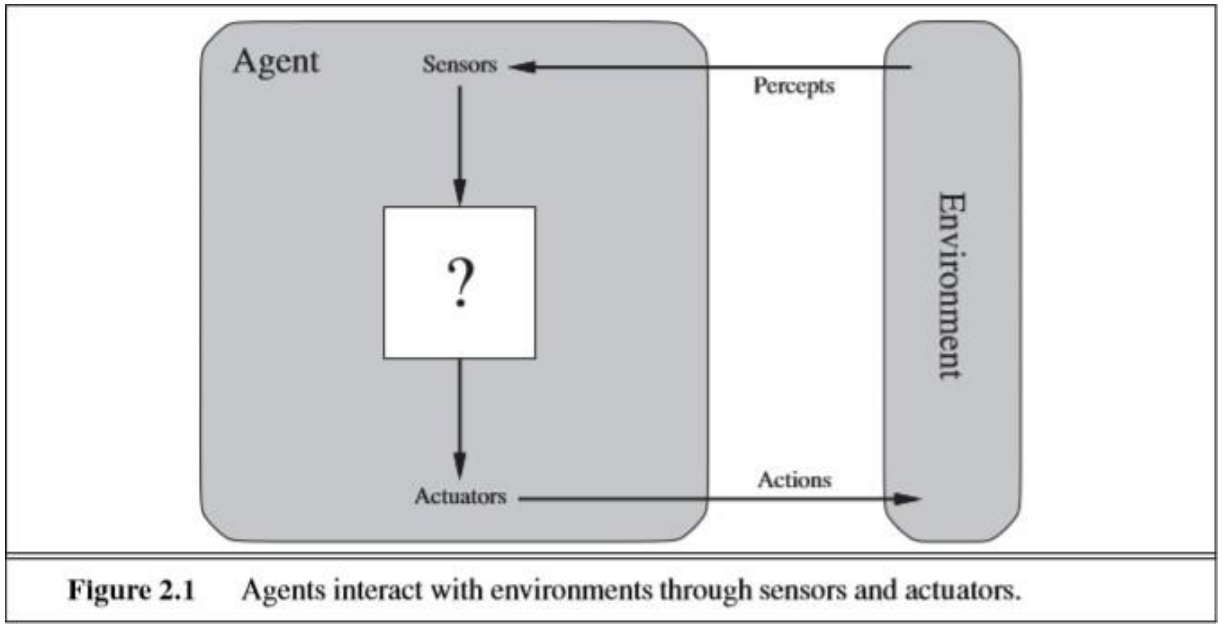


Figure 2.1 Agents interact with environments through sensors and actuators.

- In general, an agent's choice of action at any given instant can depend on the entire percept sequence observed to date, but not on anything it hasn't perceived.
- An agent's behavior is described by the **agent function** that maps any given percept sequence to an action

Tabulating Agent Function

- 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 an external characterization of the agent. Internally, the agent function for an artificial agent will be implemented by an **agent program**.
  - The agent function is an abstract mathematical description;
  - The agent program is a concrete implementation, running within some physical system.
- Example: Vacuum Cleaner World

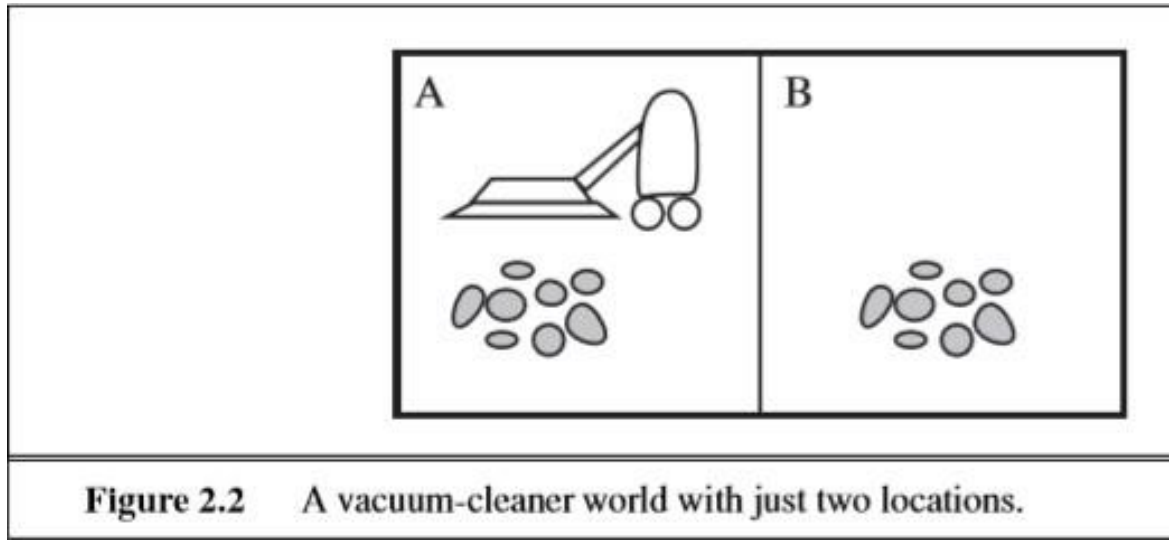


Figure 2.2 A vacuum-cleaner world with just two locations.

- 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 suck; otherwise, move to the other square.

A partial tabulation of this agent function is shown in Figure 2.3

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.

An agent program for Vacuum Cleaner Agent

```
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.

Good Behaviour: 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, doing the right thing is better than doing the wrong thing, but what does it mean to do the right thing?

Ans: We answer this by considering the consequences of the agent's behavior. 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.

If we define success in terms of agent's opinion of its own performance, an agent could achieve perfect rationality simply by deluding itself that its performance was perfect.

There is not one fixed performance measure for all tasks and agents. For Example, the vacuum cleaner agent, we might propose to measure performance by the amount of dirt cleaned up in a single eight-hour shift

Good Behaviour: The Concept of Rationality

- 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.

RATIONALITY

- **Example:**

Consider the simple vacuum-cleaner agent that cleans a square if it is dirty and moves to the other square if not; this is the agent function tabulated in Figure 2.3. Is this a rational agent?

Ans: That depends!

First, we need to say what the performance measure is, what is known about the environment, and what sensors and actuators the agent has. Let us assume the following:

- The performance measure awards one point for each clean square at each time step, over a "lifetime" of 1000 time steps.
- The "geography" of the environment is known a priori(Figure 2.2) but the dirt distribution and the initial location of the agent are not. Clean squares stay clean and sucking cleans the current square. The Left and Right actions move the agent left and right except when this would take the agent outside the environment, in which case the agent remains where it is.
- The only available actions are Left, Right, and Suck.
- The agent correctly perceives its location and whether that location contains dirt.

We claim that under these circumstances the agent is indeed rational; its expected performance is at least as high as any other agent's

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.

RATIONALITY

- How the same agent would be Irrational?
- The same agent would be irrational under different circumstances. For example, once all the dirt is cleaned up, the agent will oscillate needlessly back and forth; if the performance measure includes a penalty of one point for each movement left or right, the agent will fare poorly.
- A better agent for this case would do nothing once it is sure that all the squares are clean. If clean squares can become dirty again, the agent should occasionally check and re-clean them if needed. If the geography of the environment is unknown, the agent will need to explore it rather than stick to squares A and B.

Omniscience, learning, and autonomy

- An **omniscient** agent knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.
- Rationality is not the same as perfection.
- Rationality maximizes expected performance, while perfection maximizes actual performance.
- Doing actions in order to modify future percepts—sometimes called information gathering—is an important part of rationality .

Example: exploration that must be undertaken by a vacuum-cleaning agent in an initially unknown environment.

- **Learning:** 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.

Omniscience, learning, and autonomy

- **Autonomy:** To the extent that an agent relies on the prior knowledge of its designer rather than 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. For example, a vacuum-cleaning agent that learns to foresee where and when additional dirt will appear will do better than one that does not.
- When the agent has had little or no experience, it would have to act randomly unless the designer gave some assistance.
- After sufficient experience of its environment, the behavior of a rational agent can become effectively independent of its prior knowledge. Hence, the incorporation of learning allows one to design a single rational agent that will succeed in a vast variety of environments.

THE NATURE OF ENVIRONMENTS- Specifying the task environment

- Task environments, which are essentially the "problems" to which rational agents are the "solutions."
- Acronym for task environment is PEAS (Performance, Environment, Actuators, Sensors)

Example:

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.

THE NATURE OF ENVIRONMENTS- Specifying the task

What is the performance measure to which we would like our automated driver to aspire?

Ans: Desirable qualities include getting to the correct destination; minimizing fuel consumption and wear and tear; minimizing the trip time or cost; minimizing violations of traffic laws and disturbances to other drivers; maximizing safety and passenger comfort; maximizing profits. Obviously, some of these goals conflict, so tradeoffs will be required.

What is the driving environment that the taxi will face?

Any taxi driver must deal with a variety of roads, ranging from rural lanes and urban alleys to 12-lane freeways.

- The roads contain other traffic, pedestrians, stray animals, road works, police cars, puddles and potholes.
- The taxi must also interact with potential and actual passengers.
- Driving on the left or right
- The more restricted the environment, the easier the design problem.

THE NATURE OF ENVIRONMENTS- Specifying the task

- The **actuators** for an automated taxi include those available to a human driver: control over the engine through the accelerator and control over steering and braking. In addition, it will need output to a display screen or voice synthesizer to talk back to the passengers, and perhaps some way to communicate with other vehicles, politely or otherwise.
- The basic **sensors** for the taxi will include one or more controllable video cameras so that it can see the road; it might augment these with infrared or sonar sensors to detect distances to other cars and obstacles. To avoid speeding tickets, the taxi should have a speedometer, and to control the vehicle properly, especially on curves, it should have an accelerometer. To determine the mechanical state of the vehicle, it will need the usual array of engine, fuel, and electrical system sensors. Like many human drivers, it might want a global positioning system (GPS) so that it doesn't get lost. Finally, it will need a keyboard or microphone for the passenger to request a destination.

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

- Fully observable vs. partially observable
- Single agent vs. multiagent
- Deterministic vs. stochastic
- Episodic vs. sequential
- Static vs. dynamic
- Discrete vs. continuous
- Known vs. unknown

Properties of Task Environment

- 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. One might think that in such cases the agent's plight is hopeless, but the agent's goals may still be achievable, sometimes with certainty.

Properties of Task Environment

- Single agent v/s. multiagent:
- An agent solving a crossword puzzle by itself is clearly in a single-agent environment
- An agent playing chess is in a two agent environment.

How an entity may be viewed as an agent?

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 lawsof physics?

- The key distinction is whether B's behavior is best described as maximizing a performance measure whose value depends on agent A's behaviour.
- For example, 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. Thus, chess is a **competitive multiagent** environment.
- In the taxi-driving environment, on the other hand, avoiding collisions maximizes the performance measure of all agents, so it is a partially cooperative multiagent environment. It is also partially competitive because, for example, only one car can occupy a parking space.
- Communication often emerges as a rational behavior in multiagent environments;
- In some competitive environments, randomized behavior is rational because it avoids the pitfalls of predictability.

Deterministic V/S Stochastic

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.
- In principle, an agent need not worry about uncertainty in a fully observable, deterministic environment.
- If the environment is partially observable, however, then it could appear to be stochastic.
- Most real situations are so complex that it is impossible to keep track of all the unobserved aspects; for practical purposes, they must be treated as stochastic.

Example 1: Taxi driving is clearly stochastic, because one can never predict the behavior of traffic exactly.

Example 2: The vacuum world is deterministic, but variations can include stochastic elements such as randomly appearing dirt and an unreliable suction mechanism.

An environment is uncertain if it is not fully observable or not deterministic.

The word “stochastic” generally implies that uncertainty about outcomes is quantified in terms of probabilities;

A nondeterministic environment is one in which actions are characterized by their possible outcomes, but no probabilities are attached to them.

Nondeterministic environment descriptions are usually associated with performance measures that require the agent to succeed for all possible outcomes of its actions.

Episodic v/s. Sequential

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.
- For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions; moreover, the current decision doesn't affect whether the next part is defective.
- In **sequential environments**, on the other hand, the current decision could affect all future decisions.

Example: 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

Static v/s dynamic

Static v/s 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.
- If the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is semidynamic.

Example: Taxi driving is clearly dynamic: the other cars and the taxi itself keep moving while the driving algorithm dithers about what to do next.

Chess, when played with a clock, is semidynamic.

Crossword puzzles are static.

Discrete V/S Continuous

Discrete V/S 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 (excluding the clock). Chess also has a discrete set of percepts and actions.
- Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time.
- Taxi-driving actions are also continuous (steering angles, etc.).
- Input from digital cameras is discrete, strictly speaking, but is typically treated as representing continuously varying intensities and locations.

Known V/S Unknown

Known V/S Unknown

- This distinction refers not to the environment itself but to the agent's (or designer's) state of knowledge about the "laws of physics" of the environment.
- In a known environment, the outcomes (or outcome probabilities if the environment is stochastic) for all actions are given.
- If the environment is unknown, the agent will have to learn how it works in order to make good decisions.
- It is quite possible for a known environment to be partially observable—for example, in solitaire card games, I know the rules but am still unable to see the cards that have not yet been turned over.
- An unknown environment can be fully observable—in a new video game, the screen may show the entire game state but I still don't know what the buttons do until I try them.

- The hardest case is
  - partially observable
  - Multi-agent
  - Stochastic
  - Sequential
  - Dynamic
  - Continuous and unknown.
- Taxi driving is hard in all these senses, except that for the most part the driver's environment is known.
- Driving a rented car in a new country with unfamiliar geography and traffic laws is a lot more exciting

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.

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

Agent = architecture + program

The program we choose has to be one that is appropriate for the architecture.

- The architecture might be just an ordinary PC, or it might be a robotic car with several on board computers, cameras, and other sensors.
- In general, the architecture makes the percepts from the sensors available to the program, runs the program, and feeds the program's action choices to the actuators as they are generated.

THE STRUCTURE OF AGENTS

- Agent programs: Many agent programs have the same skeleton: they take the current percept as input from the sensors and return an action to the actuators.
- Agent program takes the current percept as Input
- Agent function takes the entire percept history.

TRIVIAL AGENT Program

- Figure 2.7 shows a trivial agent program that keeps track of the percept sequence and then uses it to index into a table of actions to decide what to do.
- The designers must construct a table that contains the appropriate action for every possible percept sequence.

```
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.

Why the table-driven approach to agent construction is doomed to failure?

- Let P be the set of possible percepts and let T be the lifetime of the agent (the total number of percepts it will receive).
- The lookup table will contain $\sum_{t=1}^{T} |P|^t$ entries.
- Consider the automated taxi: the visual input from a single camera comes in at the rate of roughly 27 megabytes per second (30 frames per second, 640×480 pixels with 24 bits of color information).
- This gives a lookup table with over $10^{250,000,000,000}$ entries for an hour's driving.
- Even the lookup table for chess—a tiny, well-behaved fragment of the real world—would have at least 10^{150} entries.
- The daunting size of these tables (the number of atoms in the observable universe is less than 10^{80}) means that
  - (a) no physical agent in this universe will have the space to store the table,
  - (b) the designer would not have time to create the table,
  - (c) no agent could ever learn all the right table entries from its experience, and
  - (d) even if the environment is simple enough to yield a feasible table size, the designer still has no guidance about how to fill in the table entries.

The key challenge for AI is to find out how to write programs that, to the extent possible, produce rational behavior from a small program rather than from a vast table.

Types of Intelligent Agents

1. Simple Reflex Agents
2. Model Based Reflex Agents
3. Goal Based Agents
4. Utility Based Agents

Simple Reflex Agents

- 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.

Example: The vacuum agent whose agent function is tabulated in Figure 2.3 is a simple reflex agent, because its decision is based only on the current location and on whether that location contains dirt.

An agent program for this agent is shown in Figure 2.8.

```
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.

Notice that the vacuum agent program is very small indeed compared to the corresponding table. The most obvious reduction comes from ignoring the percept history, which cuts down the number of possibilities from 4^T to just 4.

Simple Reflex Agents

Condition – Action Rule:

Example: Imagine yourself as the driver of the automated taxi. If the car in front brakes and its brake lights come on, then you should notice this and initiate braking. In other words, some processing is done on the visual input to establish the condition we call "The car in front is braking." Then, this triggers some established connection in the agent program to the action "initiate braking." We call such a connection a condition–action rule, written as

if car-in-front-is-braking then initiate-braking

A more general and flexible approach is first to build a general-purpose interpreter for condition–action rules and then to create rule sets for specific task environments.

Figure 2.9 gives the structure of this general program in schematic form, showing how the condition–action rules allow the agent to make the connection from percept to action.

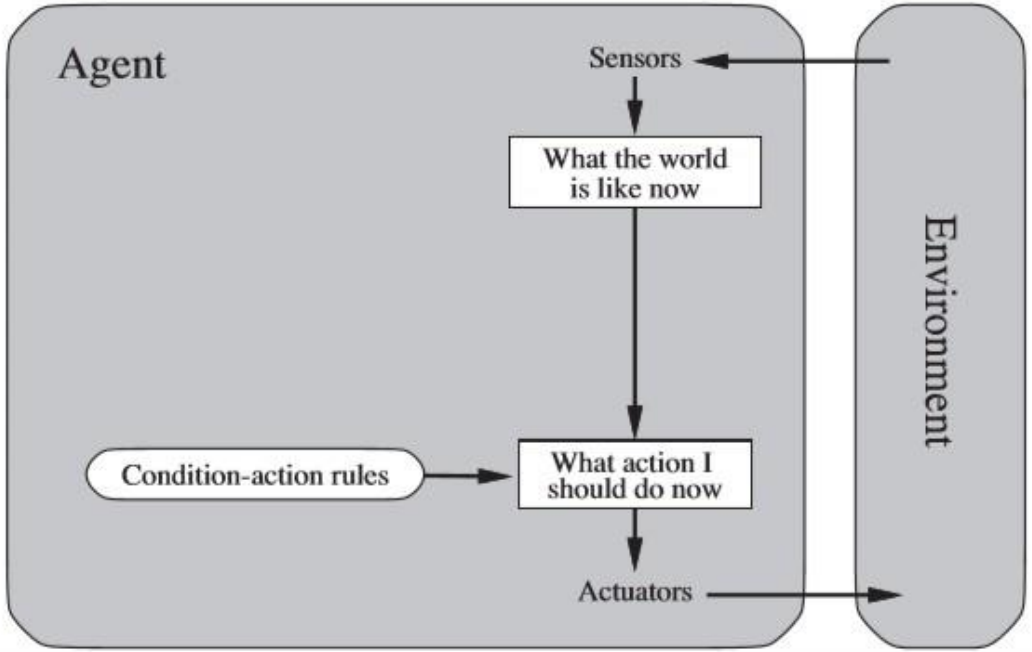


Figure 2.9 Schematic diagram of a simple reflex agent.

The **INTERPRET-INPUT** function generates an abstracted description of the current state from the percept, and the **RULE-MATCH** function returns the first rule in the set of rules that matches the given state description.

```
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.

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.
- 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 works" is called a Model of the world. An agent uses such a model is called a Model Based Agent.

Model Based Reflex Agents

- Fig. gives the structure of the model-based reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state, based on the agent's model of how the world works.

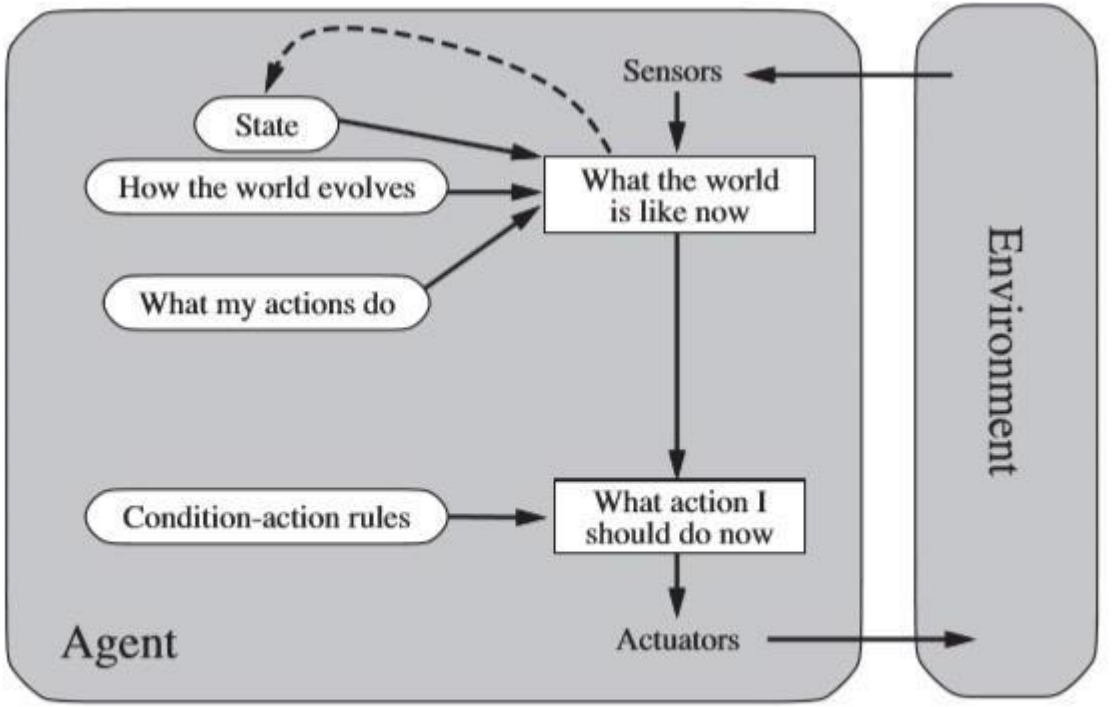


Figure 2.11 A model-based reflex agent.

Model Based Reflex Agents

```
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.

Goal Based Agents

- Knowing something 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.
- Along with current state description, the agent needs some sort of goal information that describes situations that are desirable Ex: Being at the passenger's destination.

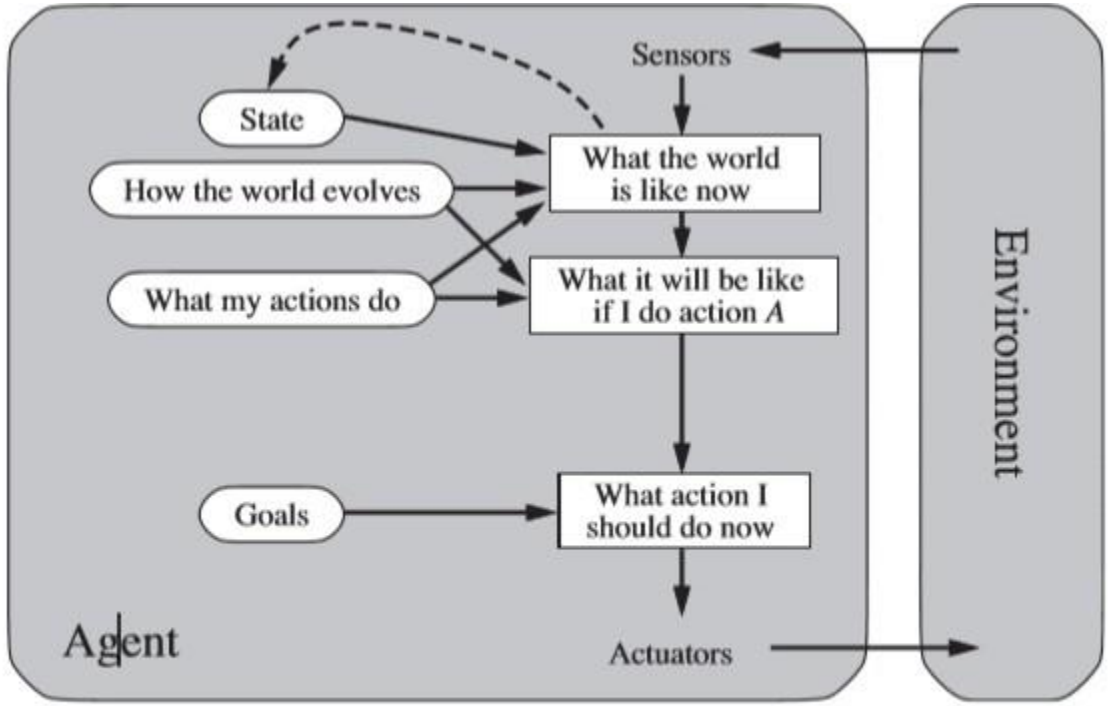


Figure 2.13 A model-based, goal-based agent. It keeps track of the world state as well as a set of goals it is trying to achieve, and chooses an action that will (eventually) lead to the achievement of its goals.

- 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.
- Search and planning are the subfields of AI devoted to finding action sequences that achieve the agent's goals

Goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified

Utility-based agents

- Goals alone are not enough to generate high-quality behavior in most environments. For example, many action sequences 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.
- A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent.
- Because “happy” does not sound very scientific, economists and computer scientists use the term utility instead.

- An agent’s utility function is essentially an internalization of the performance measure.
- 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.

Utility-based agents

- In two kinds of cases, goals are inadequate but a utility-based agent can still make rational decisions.
  - First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate tradeoff.
  - Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals.
- A rational utility-based agent chooses the action that maximizes the expected utility of the action outcomes—that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome.

Utility Based Agent

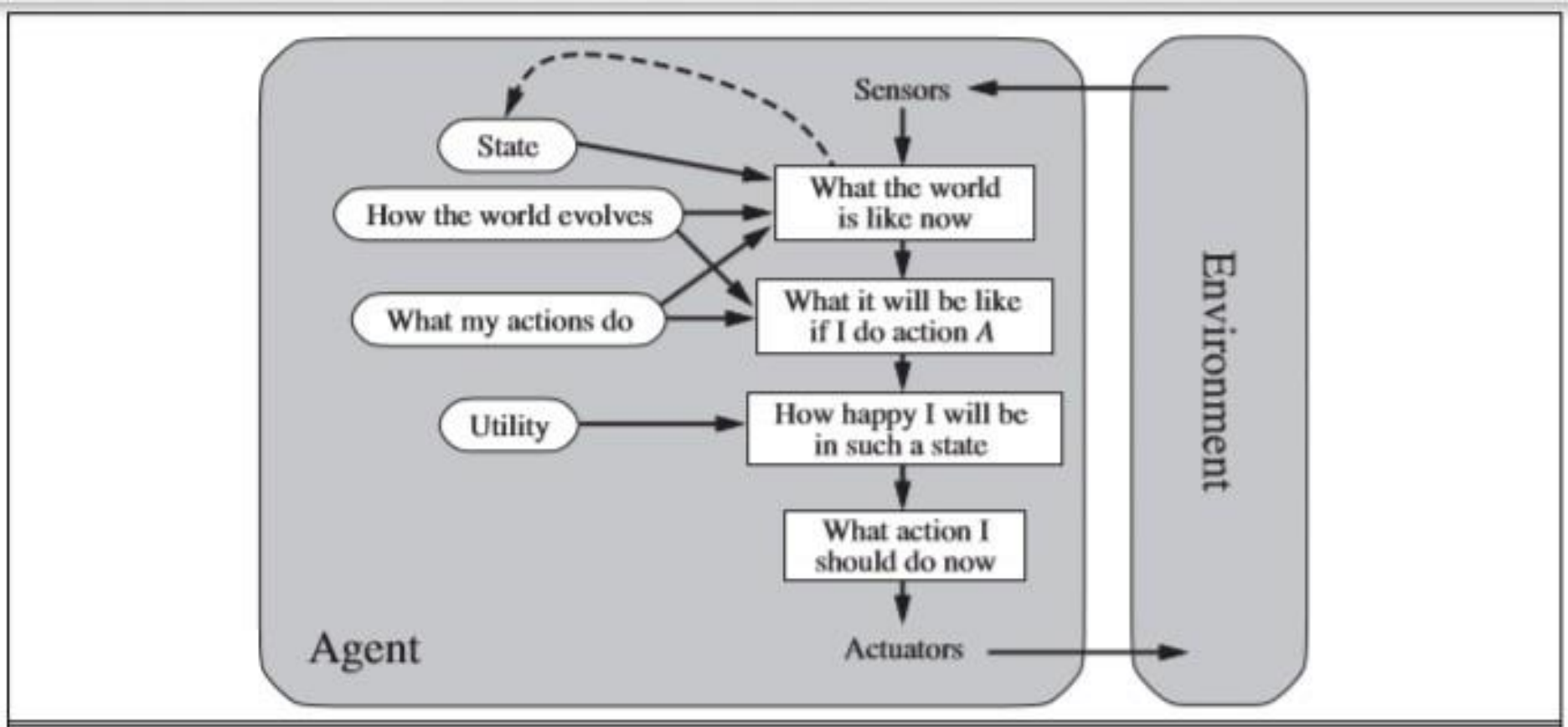


Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.

Learning Agents

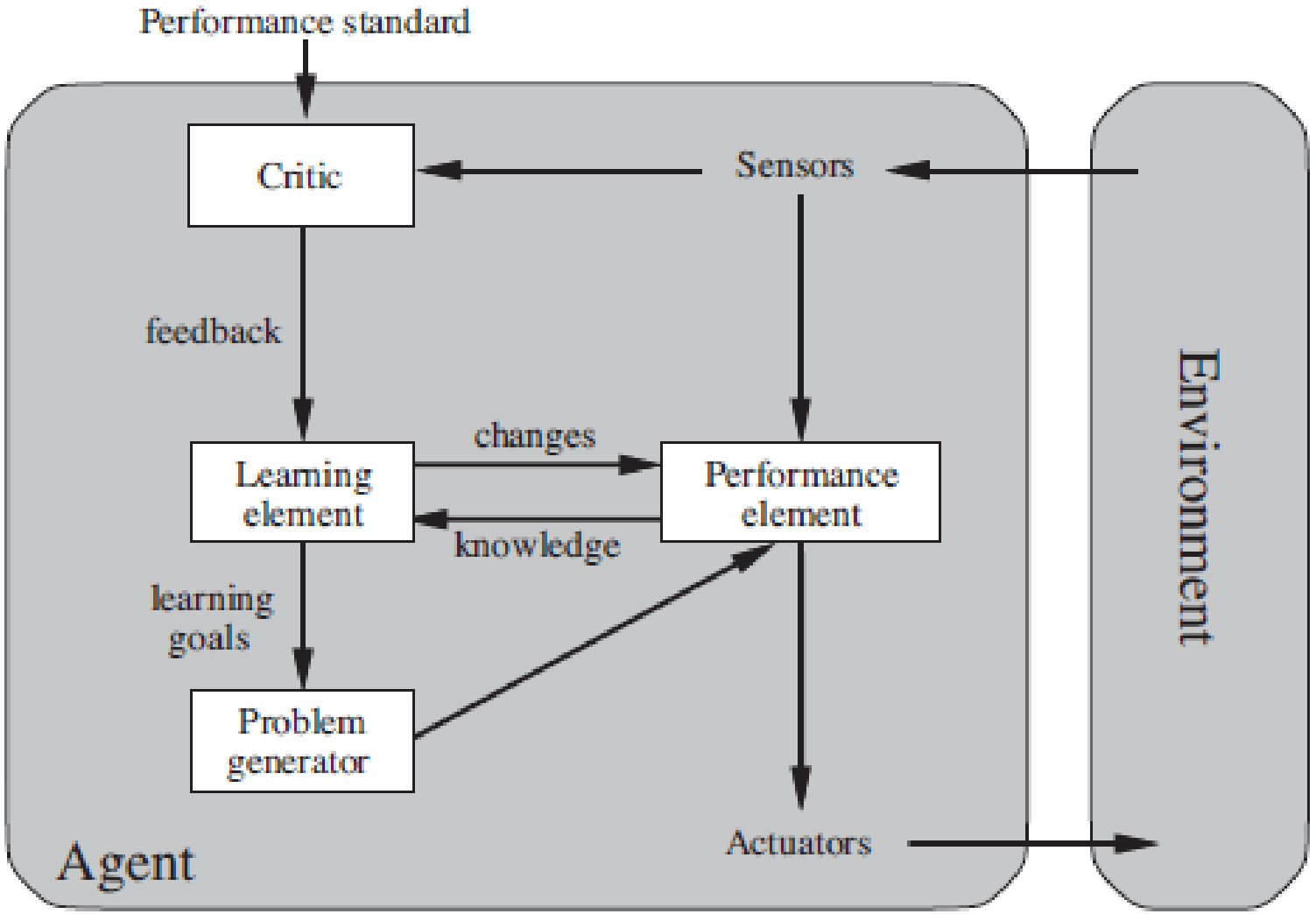


Figure 2.15 A general learning agent.

Learning Agents

- A learning agent in AI is the type of agent which can learn from its past experiences, or it has learning capabilities.
- It starts to act with basic knowledge and then able to act and adapt automatically through learning.
- There is distinction is between the
  - **learning element -** responsible for making improvements
  - **performance element-** responsible for selecting external 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.

Solving Problems by Searching

Solving problems by searching:

The agent can find a sequence of actions that achieves its goals when no single action will do.

Problem Solving Agents:

It is a goal based agent. 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 structured representations are called planning agents.

- Uninformed Search: —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: can do quite well given some guidance on where to look for solutions.

Solving problems by searching

- Intelligent agents are supposed to maximize their performance measure. Achieving this is sometimes simplified if the agent can adopt a goal and aim at satisfying it.

- Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.

Goal Formulation based on the current situation and the agent's performance measure, is the first step in problem solving.

Consider a goal to be a set of world states—exactly those states in which the goal is satisfied. The agent's task is to find out how to act, now and in the future, so that it reaches a goal state.

Solving problems by searching

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

Example: Assume that the agent will consider actions at the level of driving from one major town to another. Each state therefore corresponds to being in a particular town.

An agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.

Solving problems by searching

- To be more specific about what we mean by “examining future actions,” we have to be more specific about properties of the environment.
- We assume that,
  - Environment is **observable**, so the agent always knows the current state.
  - Environment is **discrete**, so at any given state there are only finitely many actions to choose from.
  - Environment is **known**, so that agent knows which states are reached by each action.
  - Environment is **deterministic**, so each action has exactly one outcome.

Under these assumptions, the solution to any problem is a **fixed sequence of actions.**

Solving problems by searching

- The process of looking for a sequence of actions that reaches the goal is called SEARCH.
- A search algorithm takes a problem as input and returns a solution in the form of an action sequence.
- Once a solution is found, the action it recommends can be carried out. This is called the execution phase.

- The simple design of an agent involves,
  - **FORMULATE**
  - **SEARCH**
  - **EXECUTE**

After formulating a goal and a problem to solve, the 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.

Solving problems by searching

```
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.

Well Defined Problems and Solutions

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

1. The **initial state** that the agent starts in.
   For example, the initial state for our agent in Romania might be described as In(Arad).
2. A 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)}**
3. 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, **RESULT(In(Arad),Go(Zerind)) = In(Zerind)**

- 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.

Well Defined Problems and Solutions

4. 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.

Example: 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.

5. 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.

Example: For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers.

- 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 step costs for Romania are shown in Figure 3.2 as route distances.

Well Defined Problems and Solutions

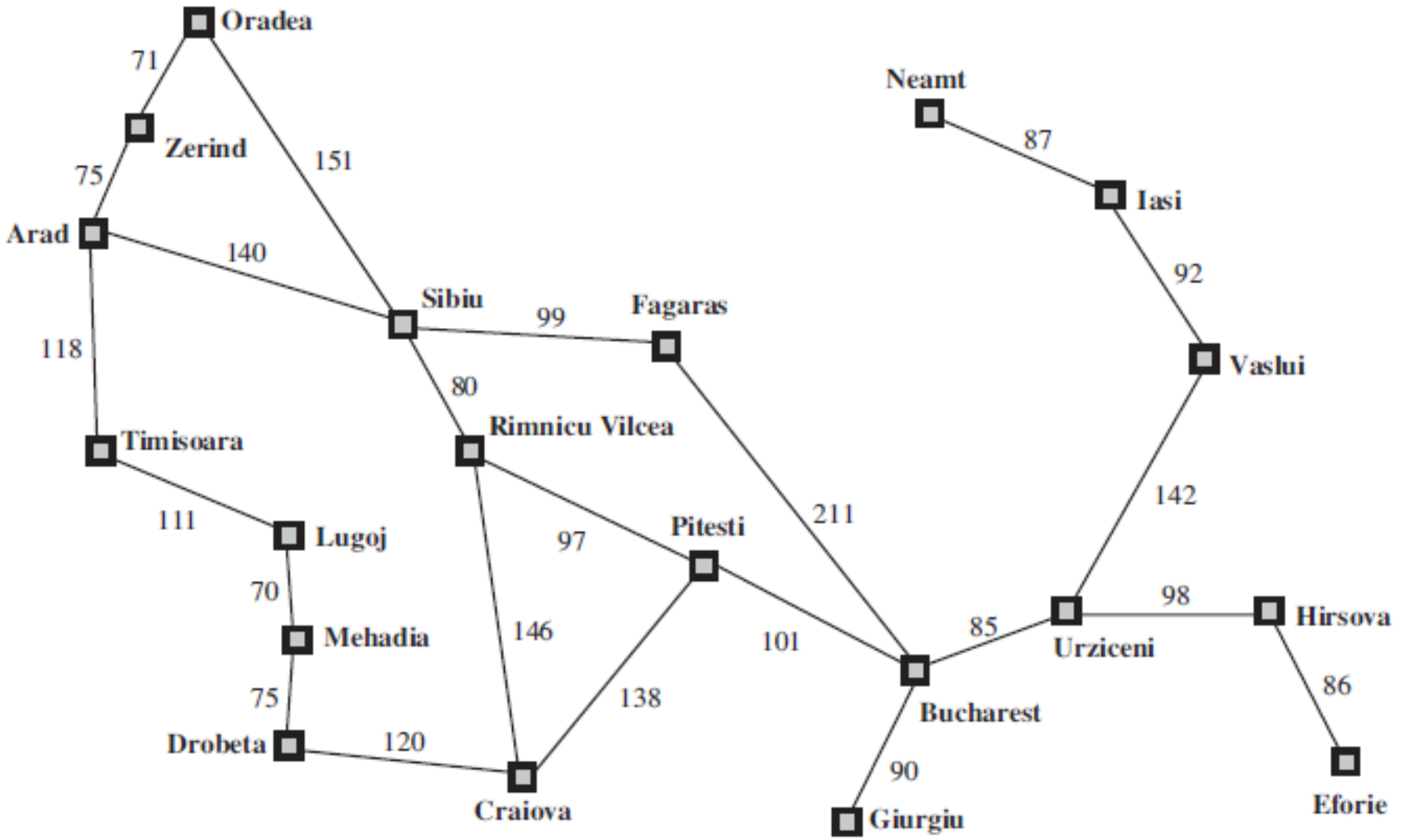


Figure 3.2 A simplified road map of part of Romania.

Formulating Problems

Solution

- 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
- **optimal solution** has the lowest path cost among all solutions.
- The process of removing detail from a representation is called **abstraction.**

Example Problems

A **toy problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is usable by different researchers to compare the performance of algorithms.

A **real-world problem** is one whose solutions people actually care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.

Example Problem – 1: Vacuum World

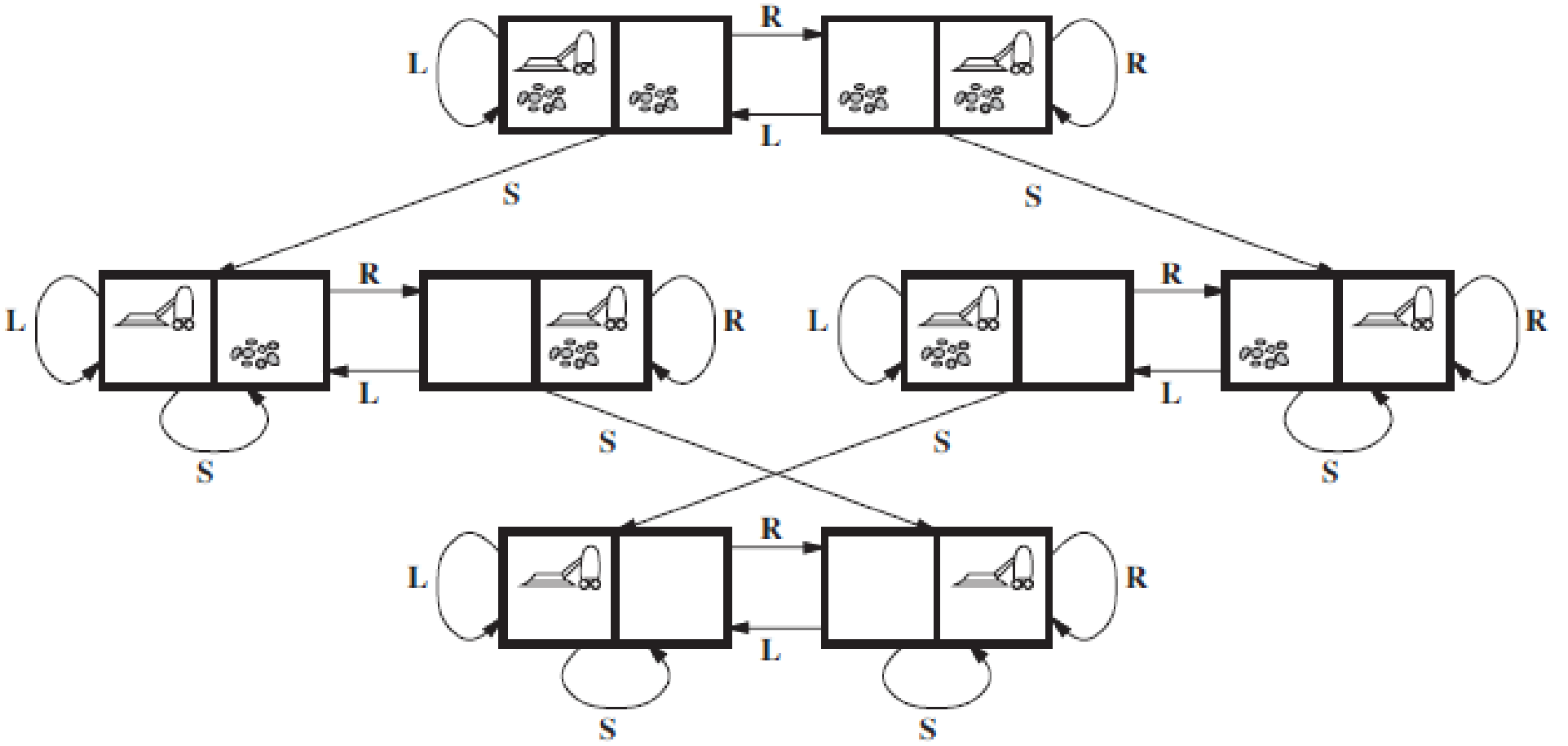


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

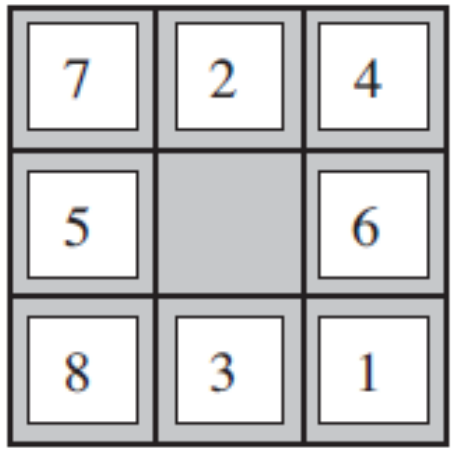
Example Problem – 1: Vacuum World

The standard formulation of Vacuum World problem is as follows:

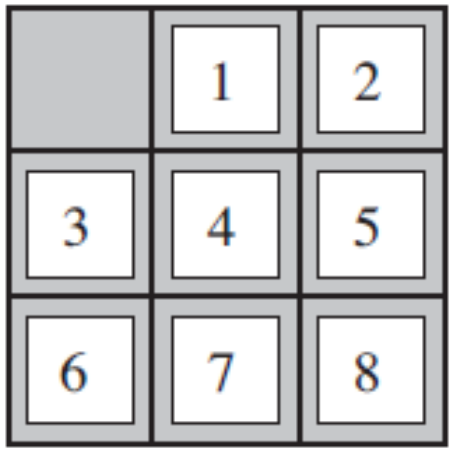
- **States**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $\mathbf{n*2^n}$ states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.
- **Transition model**: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect. The complete state space is shown in Figure 3.3.
- **Goal test**: This checks whether all the squares are clean.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

Example Problem 2: 8 Puzzle Example

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



Start State



Goal State

Figure 3.4 A typical instance of the 8-puzzle.

Example Problem 2: 8 Puzzle Example

- **The standard formulation is as follows:**
- **States**: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state**: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
- **Actions**: The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.
- **Transition model**: Given a state and action, this returns the resulting state; for example, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.
- **Goal test**: This checks whether the state matches the goal configuration shown in Figure 3.4.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.

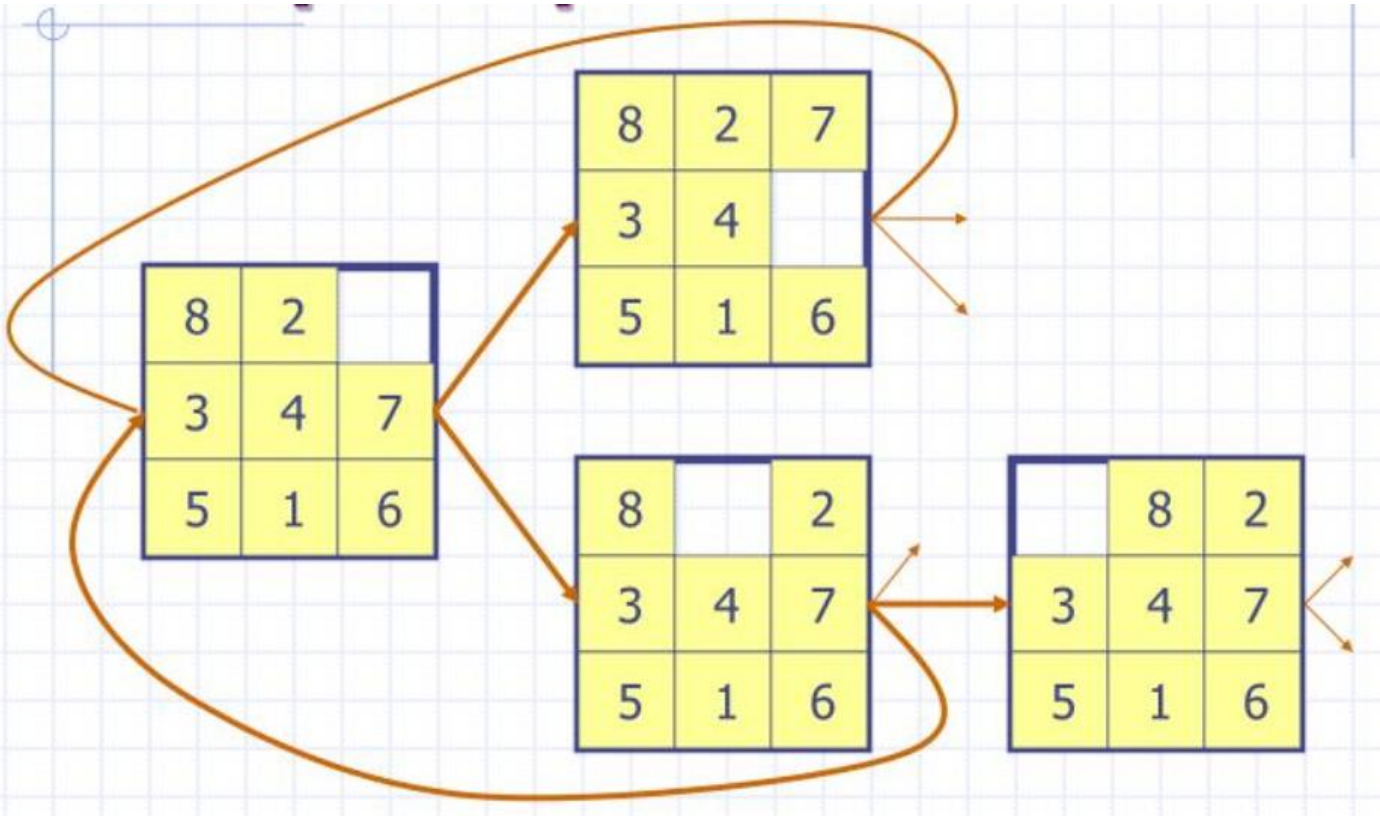
8 Puzzle Example Contd.

8	2	
3	4	7
5	1	6

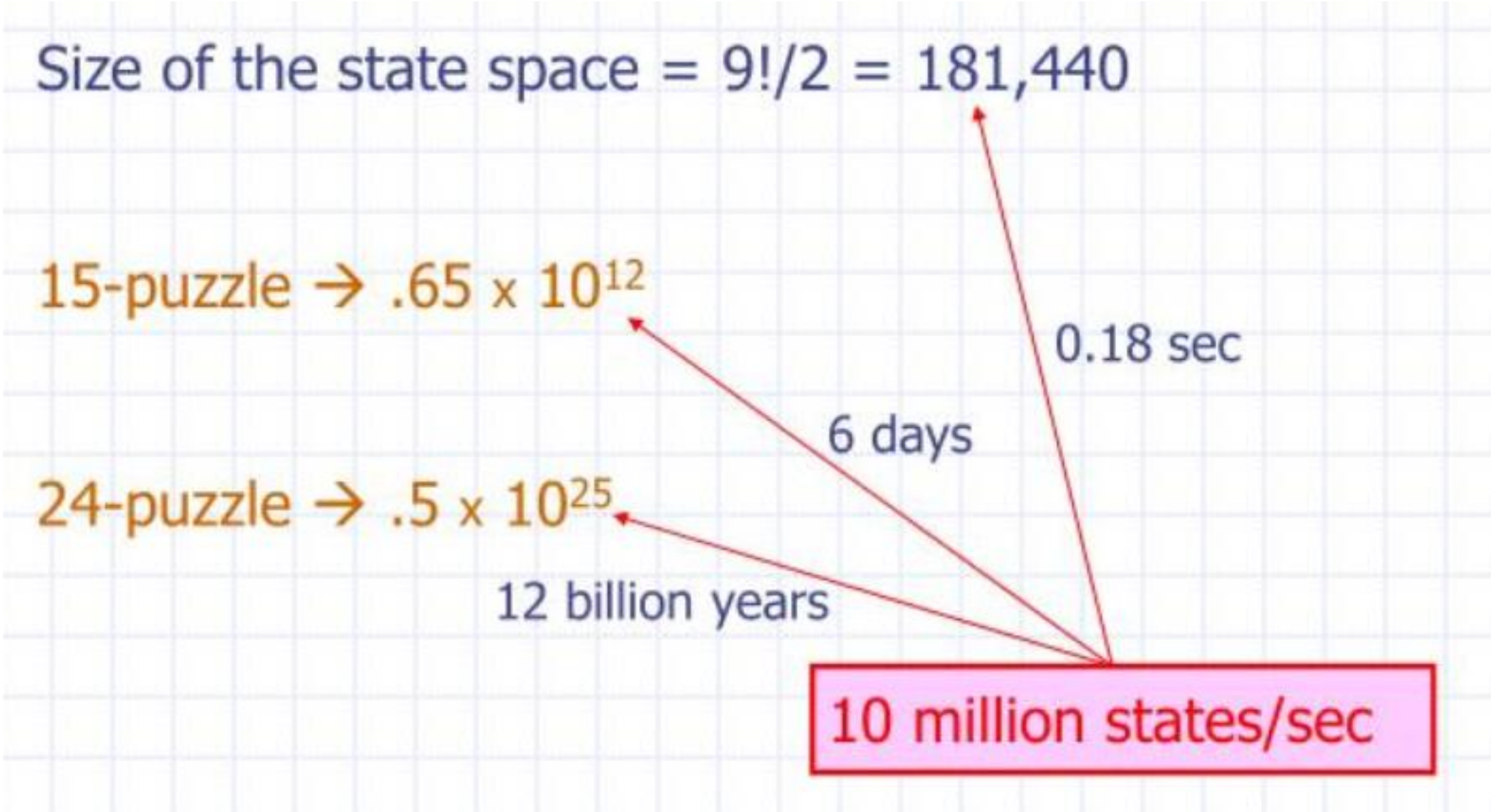
Initial state

1	2	3
4	5	6
7	8	

Goal state



Puzzle game


Size of the state space = 9!/2 = 181,440
15-puzzle → .65 x 10^12
24-puzzle → .5 x 10^25
0.18 sec
6 days
12 billion years
10 million states/sec

Puzzle game

this is known to be NP complete problem

8 puzzle has 9!/2 = 181440 reachable states

15-puzzle has(4X4) board 1.3 trillion states best search algorithm can be used optimally

24 puzzle has 10^{25} states, random instances take several hours to search randomly

Example Problem 3 : 8-queens problem

- The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other. Figure shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.

•

There are two main kinds of formulation.

- An incremental formulation involves operators that augment the state 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 the board and moves them around. In either case, the path cost is of no interest because only the final state counts.

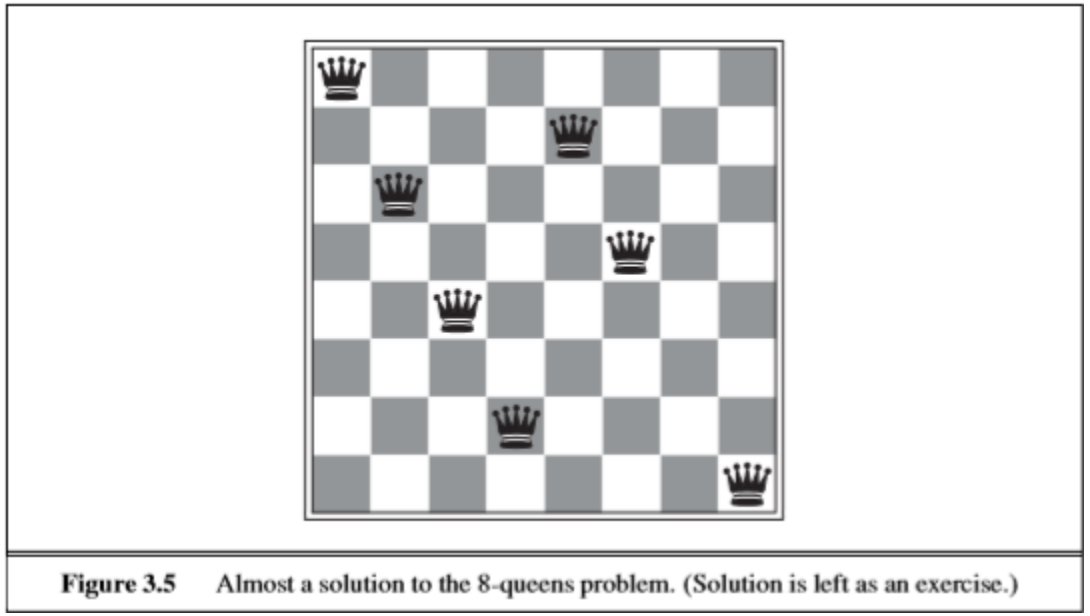


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

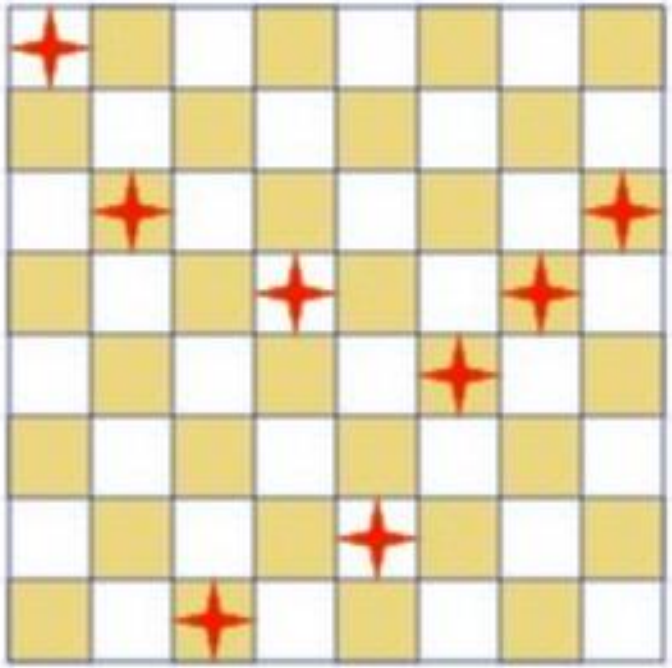
8-queens problem

- The first incremental formulation one might try is the following.
- **States:** Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state:** No queens on the board.
- **Actions:** Add a queen to any empty square.
- **Transition model:** Returns the board with a queen added to the specified square.
- **Goal test:** 8 queens are on the board, none attacked

In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 1014$ 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.

8-queens problem



Formulation #1:

- States: any arrangement of 0 to 8 queens on the board
- Initial state: 0 queens on the board
- Actions: add a queen in any square
- Goal test: 8 queens on the board, none attacked
- Path cost: none

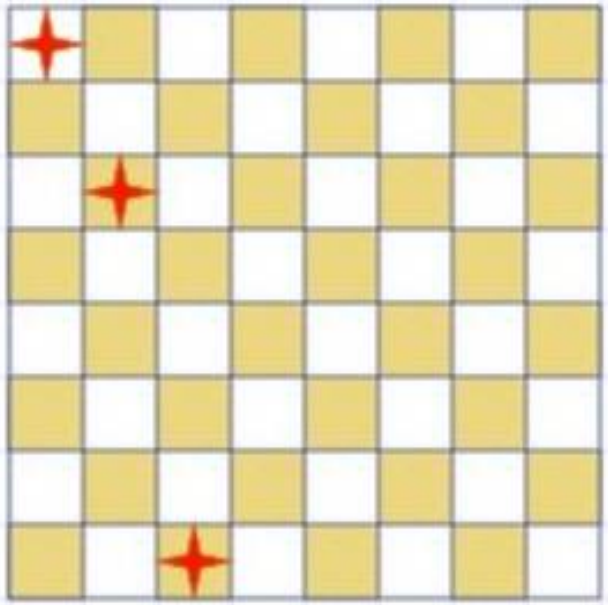
→ 64^8 states with 8 queens

8-queens problem

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.*
- This formulation reduces the 8-queens state space from 1.8×10^{14} to just **2,057**, and solutions are easy to find.
- On the other hand, for 100 queens the reduction is from roughly 10^{400} states to about 10^{52} states

8-queens problem



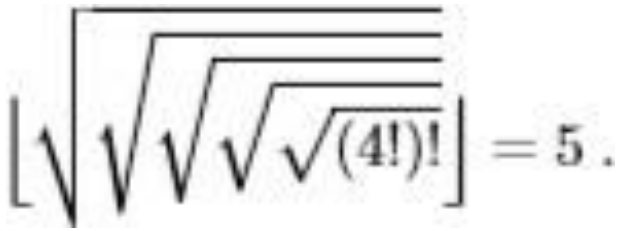
→ 2,067 states

Formulation #2:

- States: any arrangement of $k = 0$ to 8 queens in the k leftmost columns with none attacked
- Initial state: 0 queens on the board
- Successor function: add a queen to any square in the leftmost empty column such that it is not attacked by any other queen
- Goal test: 8 queens on the board

Example 4

- It was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise. Knuth conjectured that, 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

Real-world problems - Route-finding problem

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

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

Real-world problems

- **Touring problems**
- **Traveling salesperson problem (TSP)**
- **VLSI Layout**
- **Robot navigation**
- **Automatic assembly sequencing**
- **Protein design – right sequence of amino acid, with 3-D protein**

SEARCHING FOR SOLUTIONS

- Having formulated some problems, we now need to solve them.
- 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.

SEARCHING FOR SOLUTIONS

- The root node of the tree corresponds to the initial state, In(Arad).
- The first step is to test whether this is a goal state.
- Then we need to consider taking various actions.
- We do this by expanding the current state; that is, applying each legal action to the current state, thereby generating a new set of states.
- In this case, we add three branches from the **parent node** In(Arad) leading to three new **child nodes**: In(Sibiu), In(Timisoara), and In(Zerind). Now we must choose which of these three possibilities to consider further.

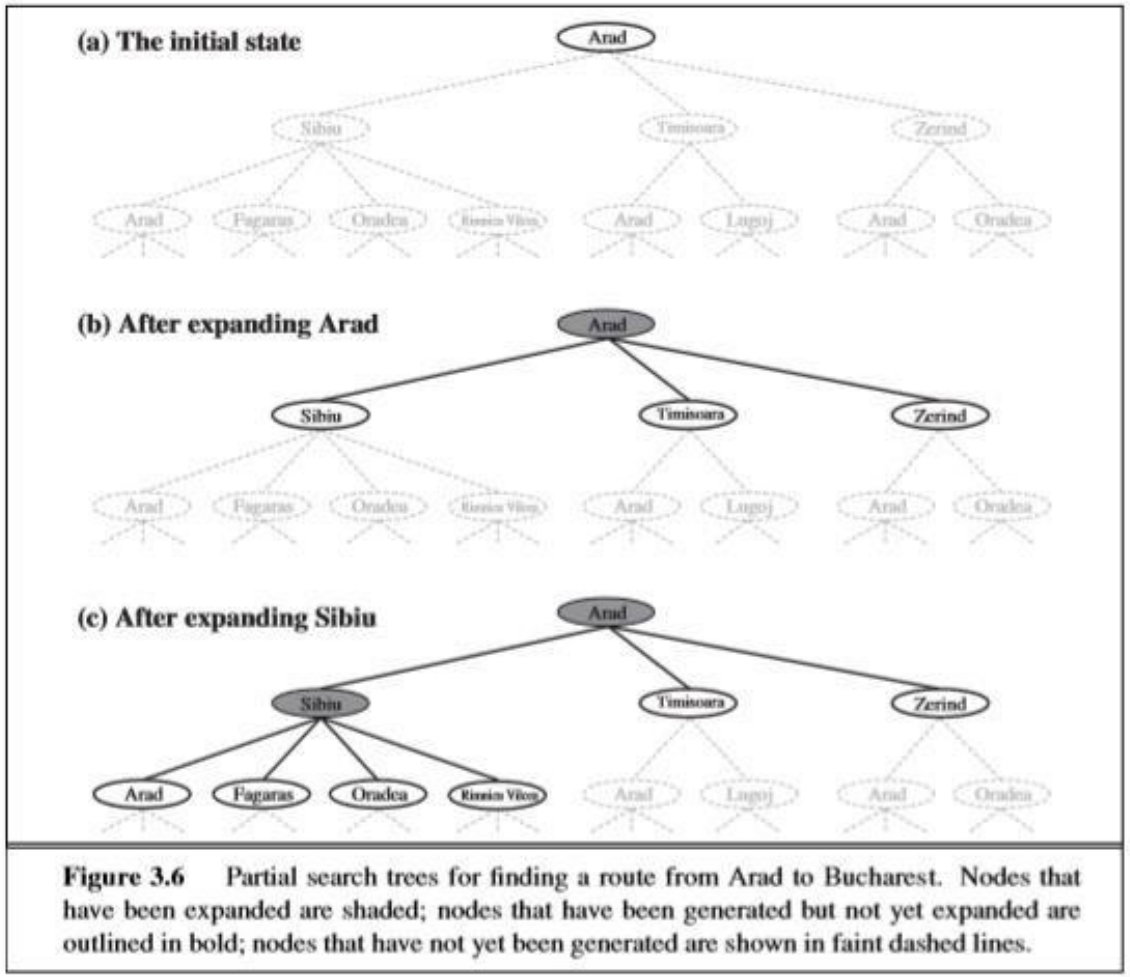


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

SEARCHING FOR SOLUTIONS

- **Leaf node** is a node with no children in the tree.
- The set of all leaf nodes available for expansion at any given point is called the **frontier or open list.**
- In Figure 3.6, the frontier of each tree consists of those nodes with bold outlines.
- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.
- The general **TREE-SEARCH** algorithm is shown informally in Figure 3.7.
- All Search algorithms share this basic structure; they vary primarily according to how they choose which state to expand next—the so-called **search strategy**.

SEARCHING FOR SOLUTIONS

```
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
```

Repeated State and Loopy path

- The search tree shown in Figure 3.6: it includes the path from Arad to Sibiu and back to Arad again! We say that In(Arad) is a **repeated state** in the search tree, generated in this case by a **loopy path**. Loops can cause certain algorithms FAIL, making otherwise solvable problems unsolvable.
- There is no need to consider loopy paths: because path costs are additive and step costs are nonnegative, a loopy path to any given state is never better than the same path with the loop removed.
- Loop 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.

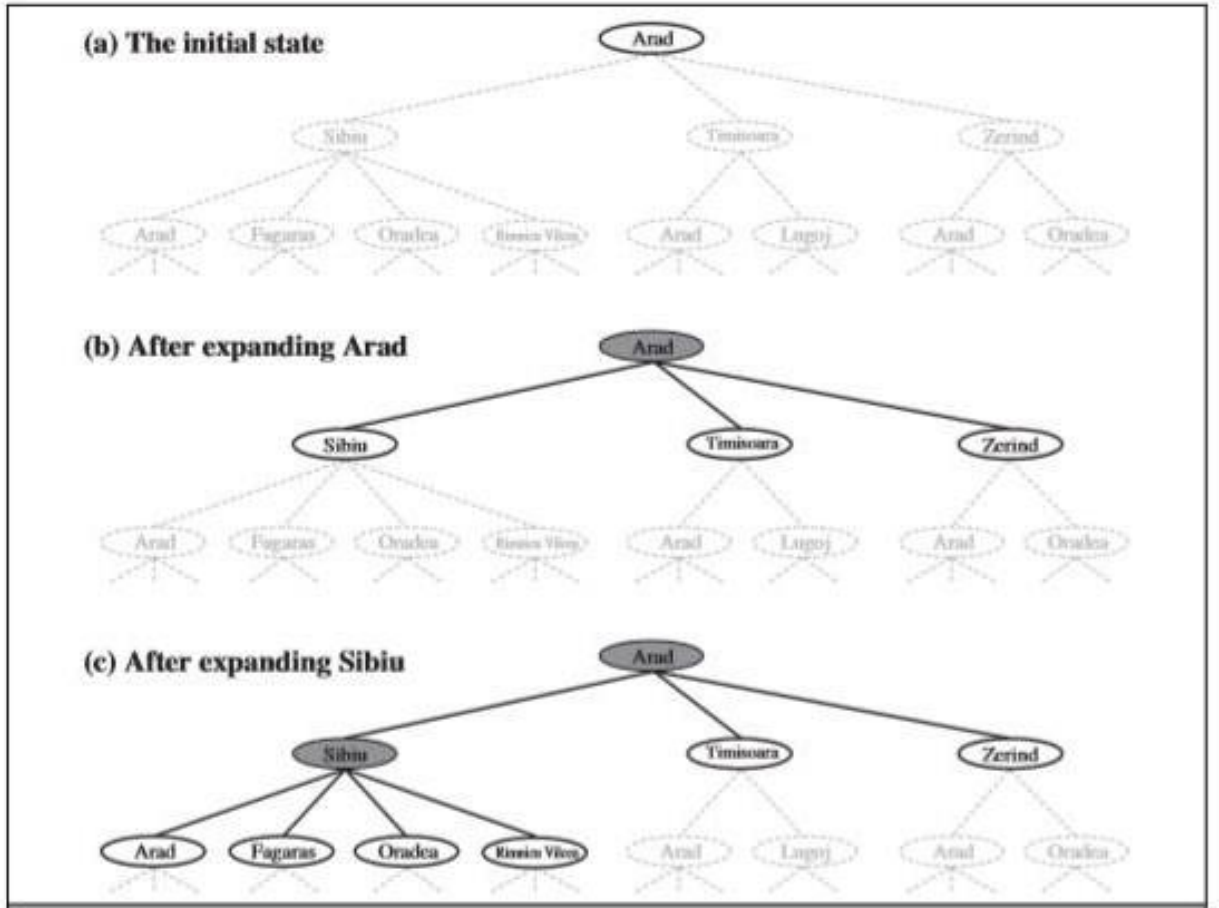
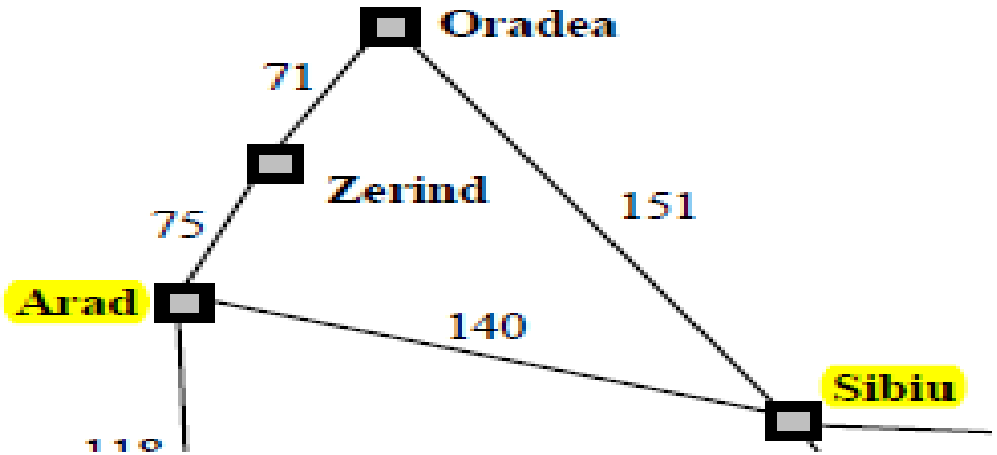


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Repeated State and Loopy path



- **Loopy paths** are a special case of the more general concept of **redundant paths**
- Paths Arad–Sibiu (140km long) and Arad–Zerind–Oradea–Sibiu (297km long)- Redundant
- To avoid exploring redundant paths is to remember where one has been

TREE SEARCH AND GRAPH SEARCH ALGORITHM

- Algorithm that forget their history are doomed to repeat it.
- The way to avoid exploring redundant paths is to remember where one has been.
- To do this, we augment the TREE_SEARCH algorithm with a data structure called **explored set** (also known as the **closed list**), which remembers every expanded node.
- Newly generated nodes that match previously generated nodes—ones in the explored set or the frontier—can be discarded instead of being added to the frontier.
- The new algorithm, called **GRAPH- SEARCH**, is shown informally in Figure.

TREE S

```
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 3.7 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.

TREE SEARCH AND GRAPH SEARCH ALGORITHM

- The search tree constructed by the GRAPH-SEARCH algorithm contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph,as shown in Figure 3.8.
- The algorithm has another nice property: the frontier **separates** the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier.
- This property is illustrated in Figure 3.9.
- As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier, we see that the algorithm is *systematically* examining the states in the state space, one by one, until it finds a solution.

TREE SEARCH AND GRAPH SEARCH ALGORITHM

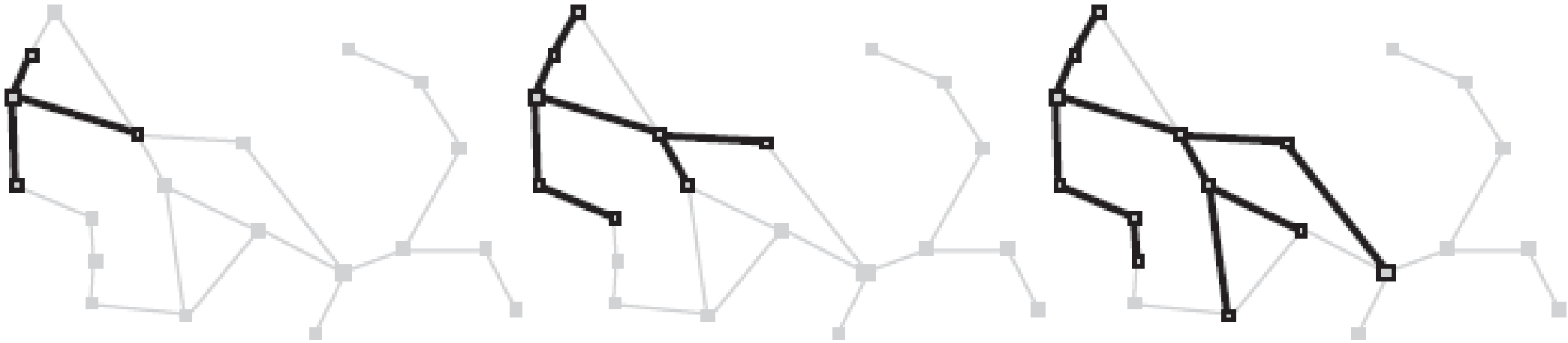


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

TREE SEARCH AND GRAPH SEARCH ALGORITHM

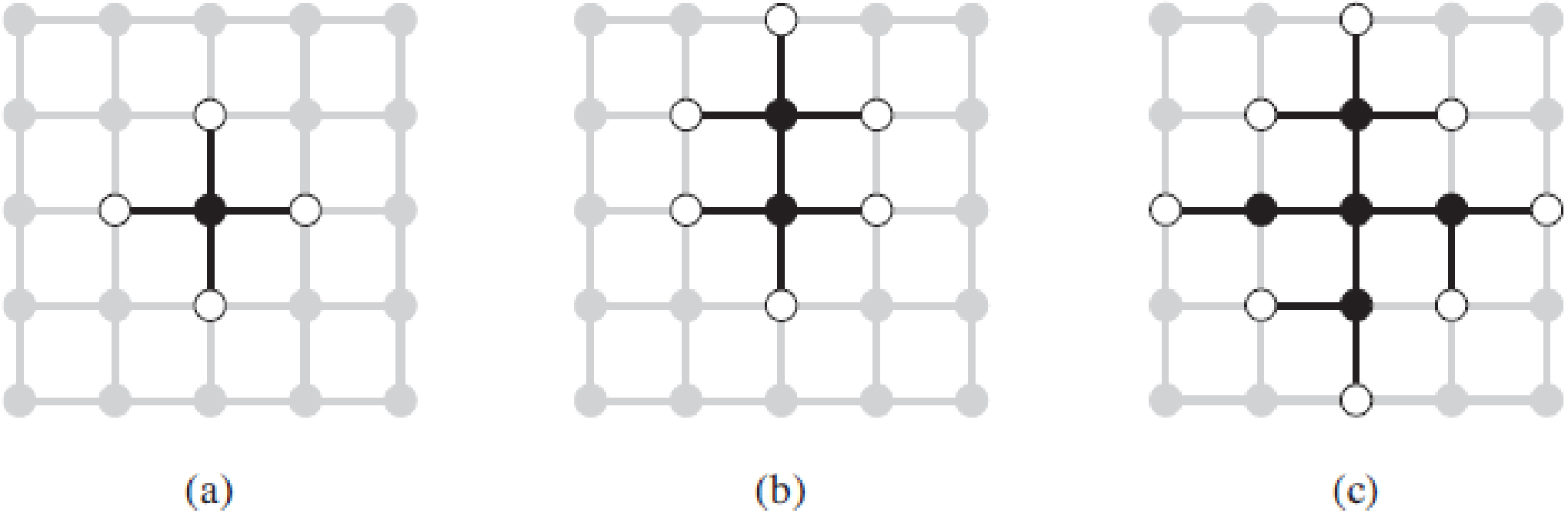


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

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.

Infrastructure for Search Algorithms

- 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)
```

Infrastructure for Search Algorithms

- The **node data structure** is depicted in Figure.

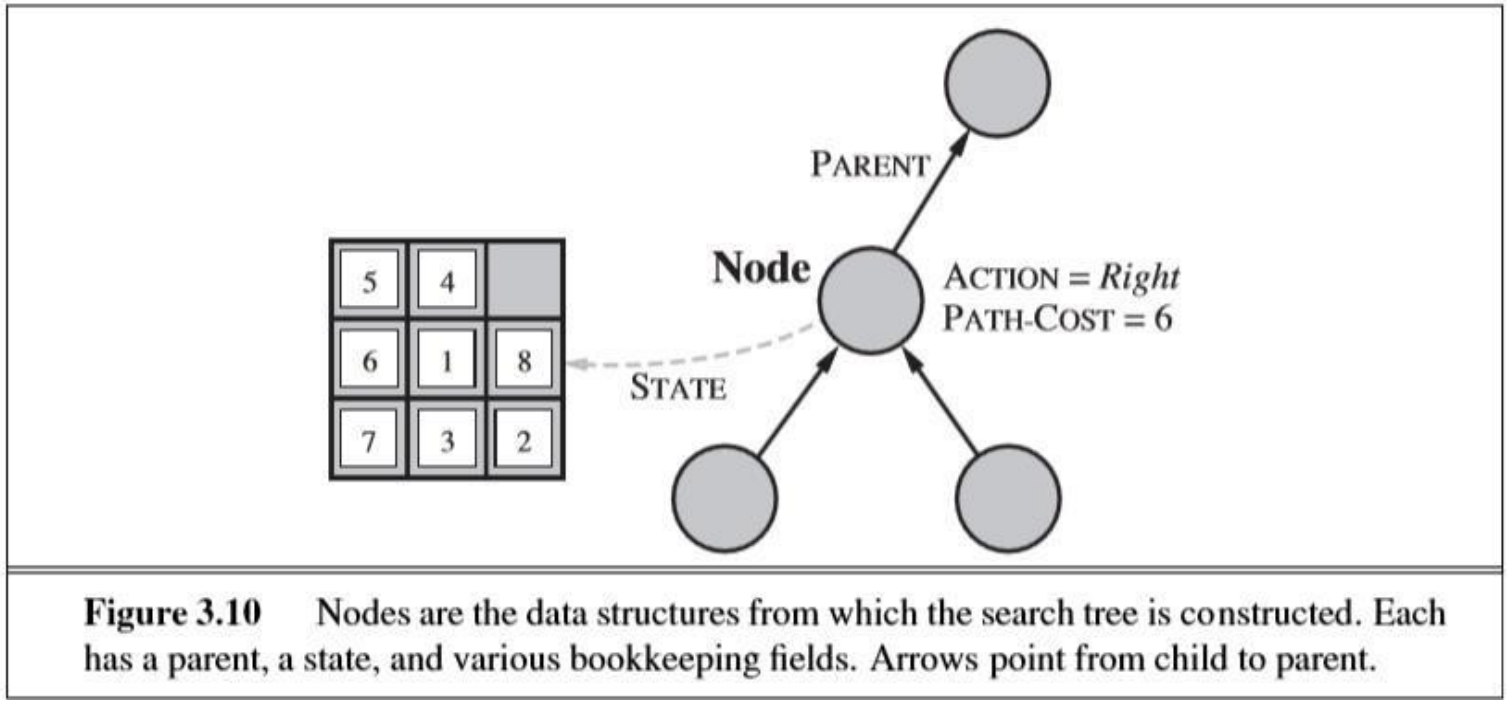


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

- Notice how the PARENT pointers string the nodes together into a tree structure. These pointers also allow the solution path to be extracted when a goal node is found; we use the SOLUTION function to return the sequence of actions obtained by following parent pointers back to the root.

Infrastructure for Search Algorithms

- **Node v/s States**
  - A node is a book keeping data structure used to represent the search tree.
  - A state corresponds to a configuration of the world.
  - Nodes are on particular paths, as defined by PARENT pointers, whereas states are not.
  - Two different nodes can contain the same world state if that state is generated via two different search paths.
- The **frontier** needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy.

Infrastructure for Search Algorithms

- 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.

- Queues are characterized by the order in which they store the inserted nodes.
- **Three common variants** are:

1. the first-in, first-out or **FIFO queue**, which pops the oldest element of the queue;
2. the last-in, first-out or **LIFO queue** (also known as a stack), which pops the newest element of the queue;
3. 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 algorithms performance 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
- Time and space complexity are always considered with respect to some measure of the problem difficulty.
- In theoretical computer science, the typical measure is the size of the state space graph, **|V | + |E|,** where V is the set of vertices (nodes) of the graph and E is the set of edges (links).

Measuring problem-solving performance

- 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**:
  - **b,** the **branching factor** or maximum number of successors of any node;
  - **d**, the **depth** of the shallowest goal node (i.e., the number of steps along the path from the root);
  - **m**, the **maximum length** of any path in the state space
- Time is measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory

UNINFORMED SEARCH STRATERGIES

- Uninformed Search: 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.

Breadth-first search:

- Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- 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.
- The BFS always has the shallowest path to every node on the frontier.

Breadth-first search

```
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.

Breadth-first search

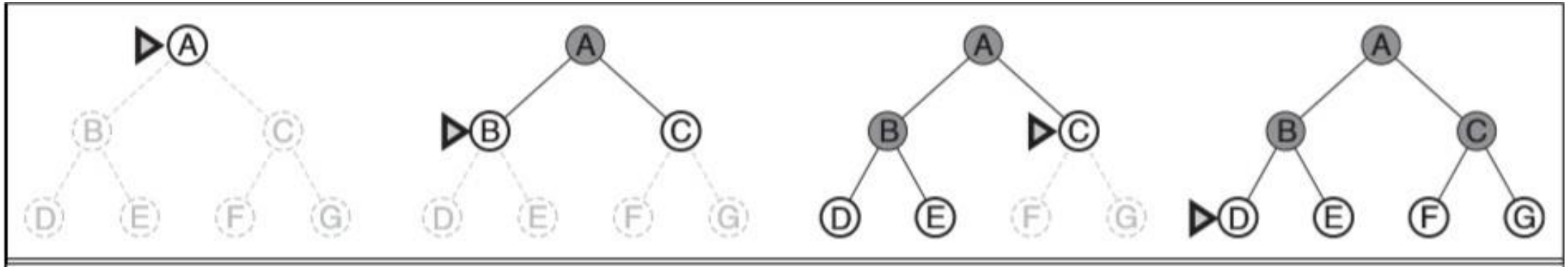


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Breadth-first Search – Space and Time Complexity

- BSF is complete—if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite).
- Breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node. The most common such scenario is that all actions have the same cost.
- With respect to time and space BFS is not good.
- Imagine searching a uniform tree where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b2 at the second level. Each of these generates b more nodes, yielding b3 nodes at the third level, and so on.
- Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodes generated is $b + b^2 + b^3 + \cdots + b^d = O(b^d)$.
- As for space complexity: for any kind of graph search, which stores every expanded node in the explored set, the space complexity is always within a factor of b of the time complexity. For breadth-first graph search in particular, every node generated remains in memory. There will be O(bd−1) nodes in the explored set and O(bd) nodes in the frontier, so the space complexity is O(bd), i.e., it is dominated by the size of the frontier.

Uniform-cost search

- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node.
- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost g(n). This is done by storing the frontier as a priority queue ordered by g.

```
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.

Uniform-cost search

- In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search.
  - The first is that the goal test is applied to a node when it is selected for expansion rather than when it is first generated.
  - The second difference is that a test is added in case a better path is found to a node currently on the frontier.

Example: Problem is to get from Sibiu to Bucharest.

Uniform-cost search is optimal in general.

- First, we observe that whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found.
- Then, because step costs are nonnegative, paths never get shorter as nodes are added. These two facts together imply that uniform-cost search expands nodes in order of their optimal path cost. Hence, the first goal node selected for expansion must be the optimal solution

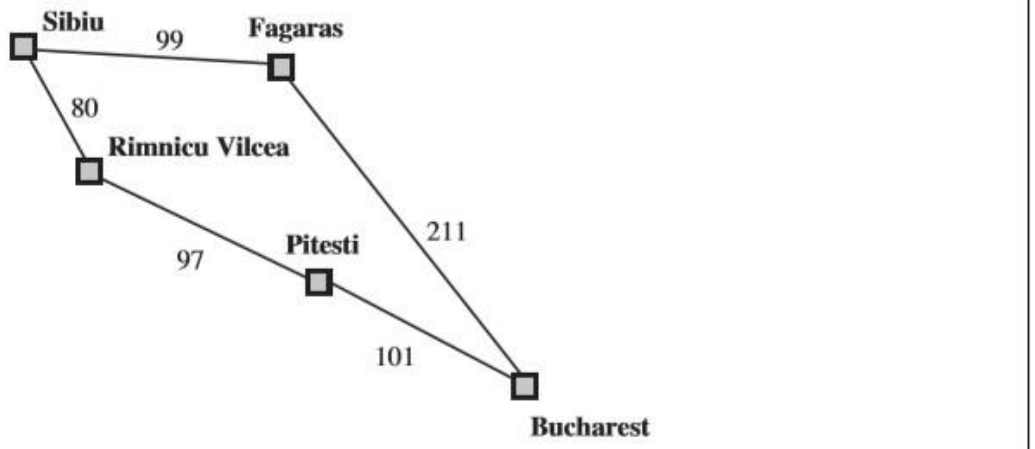


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

Depth-first search

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

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;

- breadth-first-search uses a FIFO queue,
- depth-first search uses a LIFOqueue.
- 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.

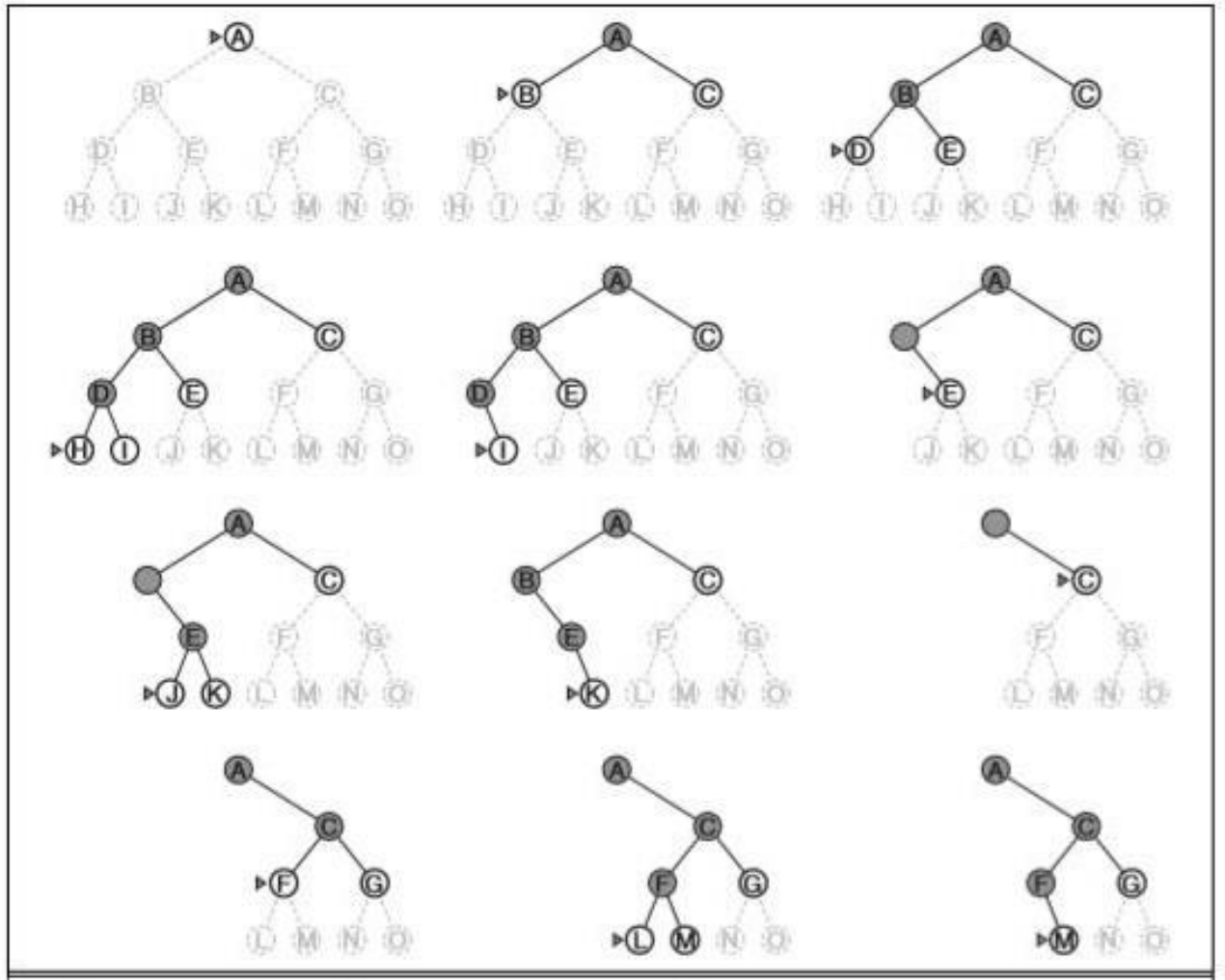


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Depth-first search

- The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used.
- The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node.
- The tree-search version, on the other hand, is not complete—for example, in Figure the algorithm will follow the Arad–Sibiu–Arad–Sibiu loop forever.
- Depth-first tree search can be modified at no extra memory cost so that it checks new states against those on the path from the root to the current node; this avoids infinite loops in finite state spaces but does not avoid the proliferation of redundant paths. In infinite state spaces, both versions fail if an infinite non-goal path is encountered.
- The time complexity of depth-first graph search is bounded by the size of the state space (which may be infinite, of course). A depth-first tree search, on the other hand, may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space

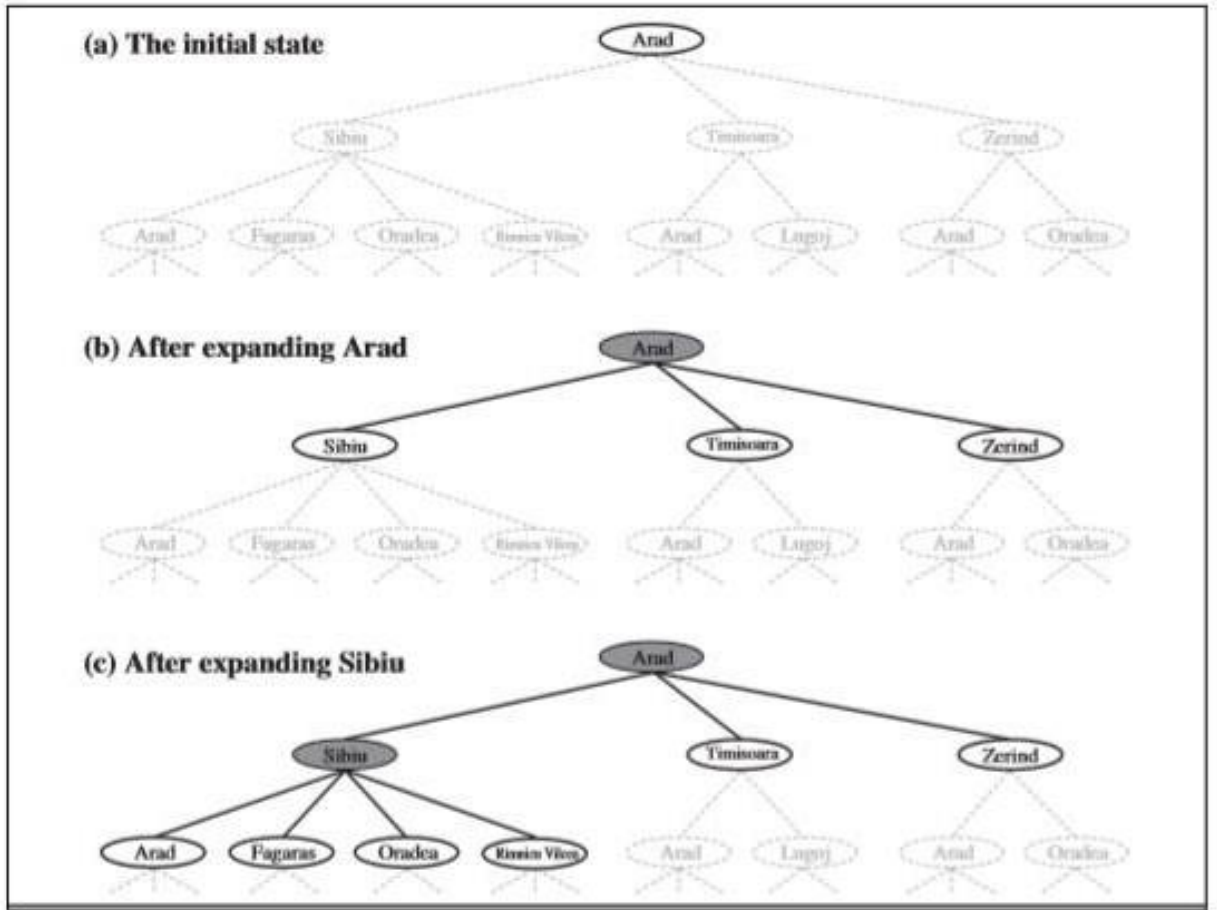


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

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 l. That is, nodes at depth l 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 l<d, that is, the shallowest goal is beyond the depth limit.
- 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 $l=\infty$.
- 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 solution, it must be of length 19 at the longest, so l= 19is a possible choice.
- But in fact if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 steps. This number, known as the diameter of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search.

```
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.

Iterative deepening depth-first search

- Iterative deepening search (or iterative deepening depth-first search) 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.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Figure 3.18 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 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 nondecreasing function of the depth of the node

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.

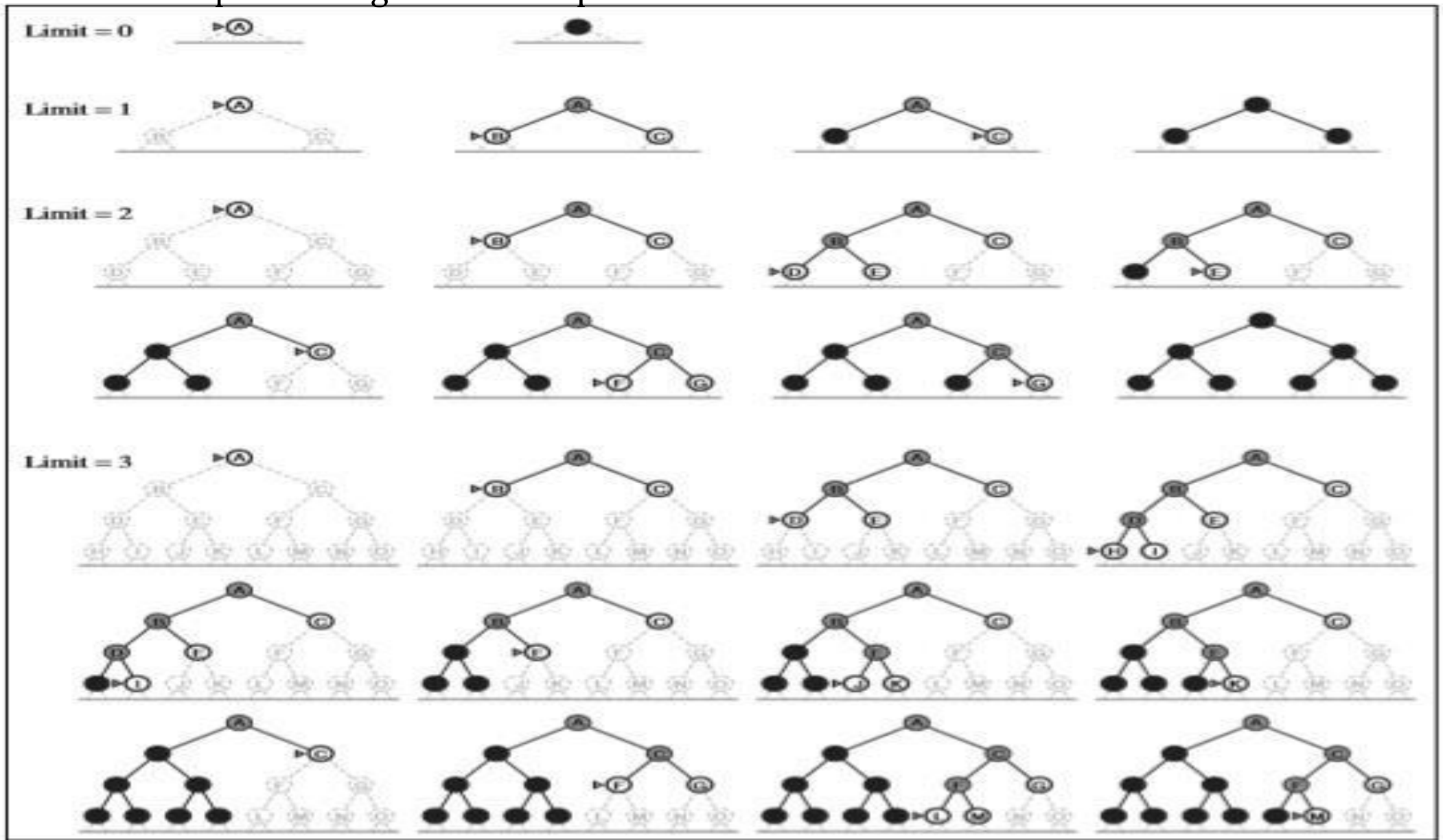


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

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.
- 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.

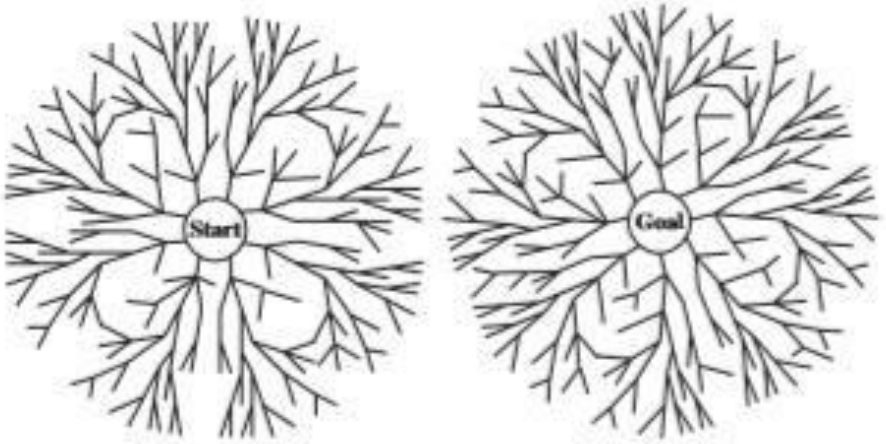


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

Comparing uninformed search strategies

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

Figure 3.21 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.

b = Branching Factor
d = Depth of the shallowest solution
m = the maximum depth of the search tree
l = Depth Limit
a = Complete if b is finite
b = Complete if step cost >= € for positive €
c = Optimal if step costs are all identical
d= if both directions use BSF

Informed (Heuristic) Search Strategies

- Informed search strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy.
- 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).
- The evaluation function is construed as 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, except for the use of f instead of g to order the priority queue.
- The choice of f determines the search strategy
- 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

- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm.
- we consider them to be arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then h(n)=0.

Greedy Best-First Search -GBFS

- Greedy best-first search tries to expand the node that is closest to the goal, on the grounds 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).
- Example: route-finding problems in Romania
- We use the straight-line distance heuristic (h_{SLD})
- If the goal is Bucharest, we need to STRAIGHT-LINE DISTANCE know the straight-line distances to Bucharest, which are shown in Figure 3.22

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

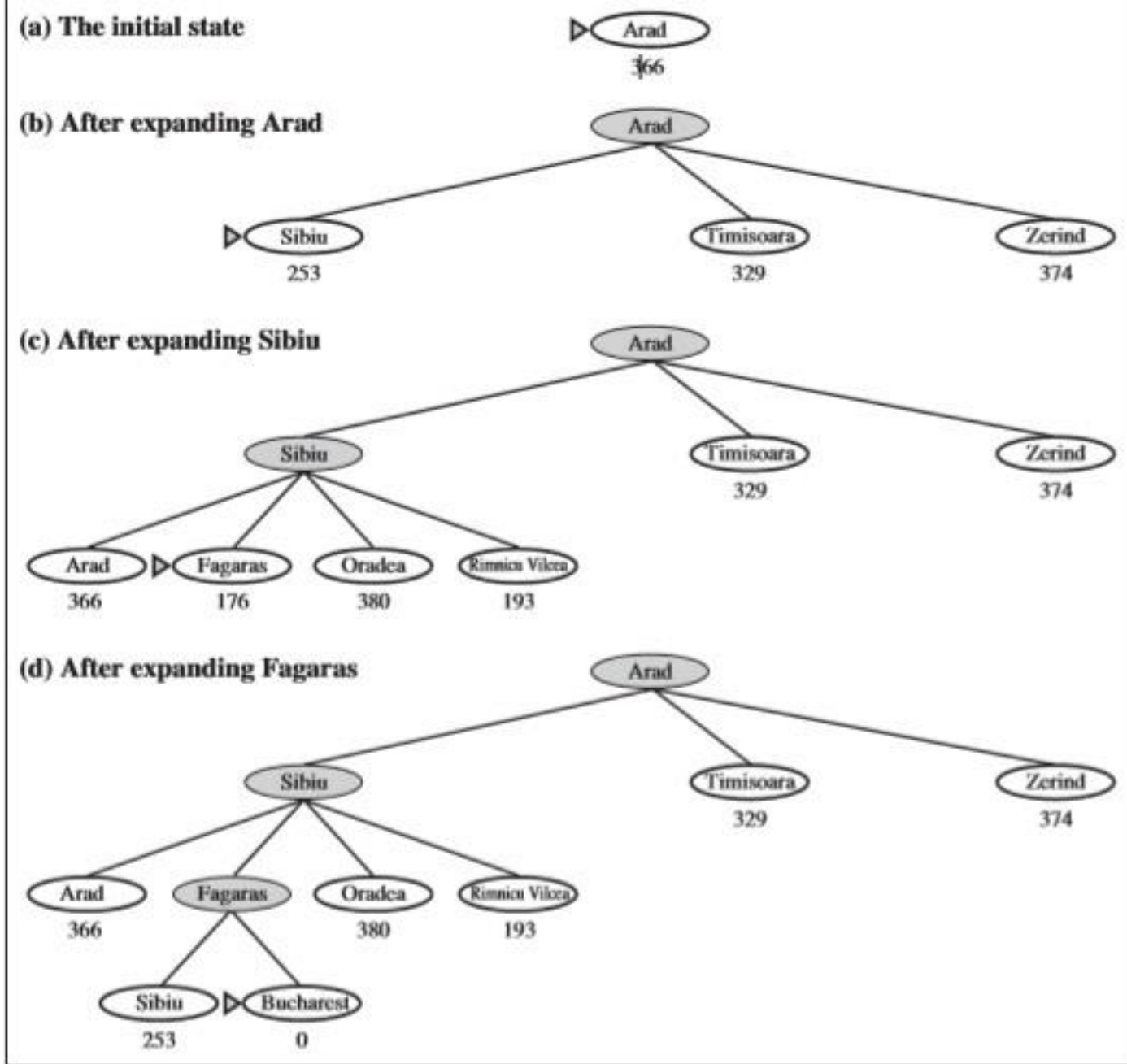


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

A* search: Minimizing the Total Estimated Solution Cost

- The most widely known form of best-first search is called A∗ 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:

$$f(n)=g(n)+h(n)$$

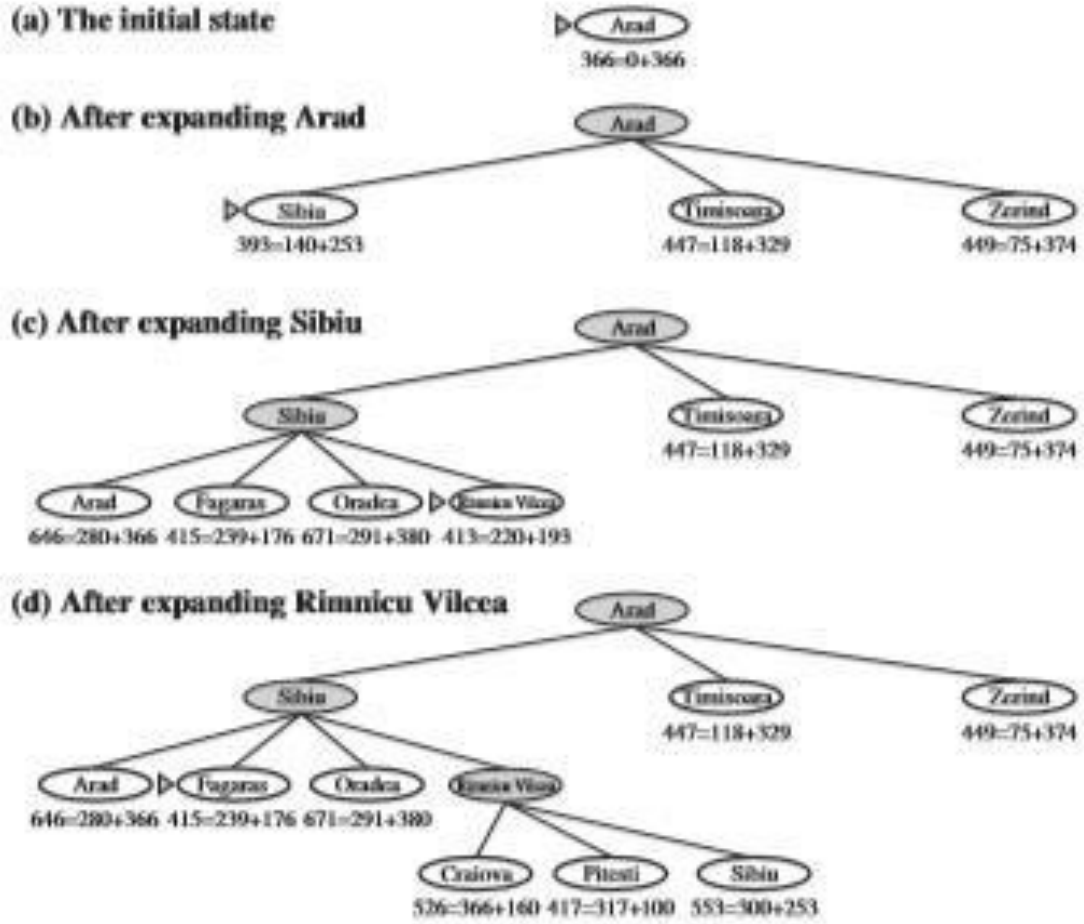
- g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have

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.

Conditions for optimality: Admissibility and consistency

- The first condition we require for optimality is that h(n) be an admissible heuristic. An admissible heuristic is one that never overestimates the cost to reach the goal. Because g(n) is the actual cost to reach n along the current path, and f(n)=g(n)+h(n), we have as an immediate consequence that f(n) never overestimates the true cost of a solution along the current path through n.
- Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is. An obvious example of an admissible heuristic is the straight-line distance hSLD
- Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate.
- A second, slightly stronger condition called **consistency** (or sometimes monotonicity) is required only for applications of A∗ to graph search. A heuristic h(n) is consistent if, for every node n and every successor n of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n:
- $h(n) \leq c(n,a,n') + h(n')$.



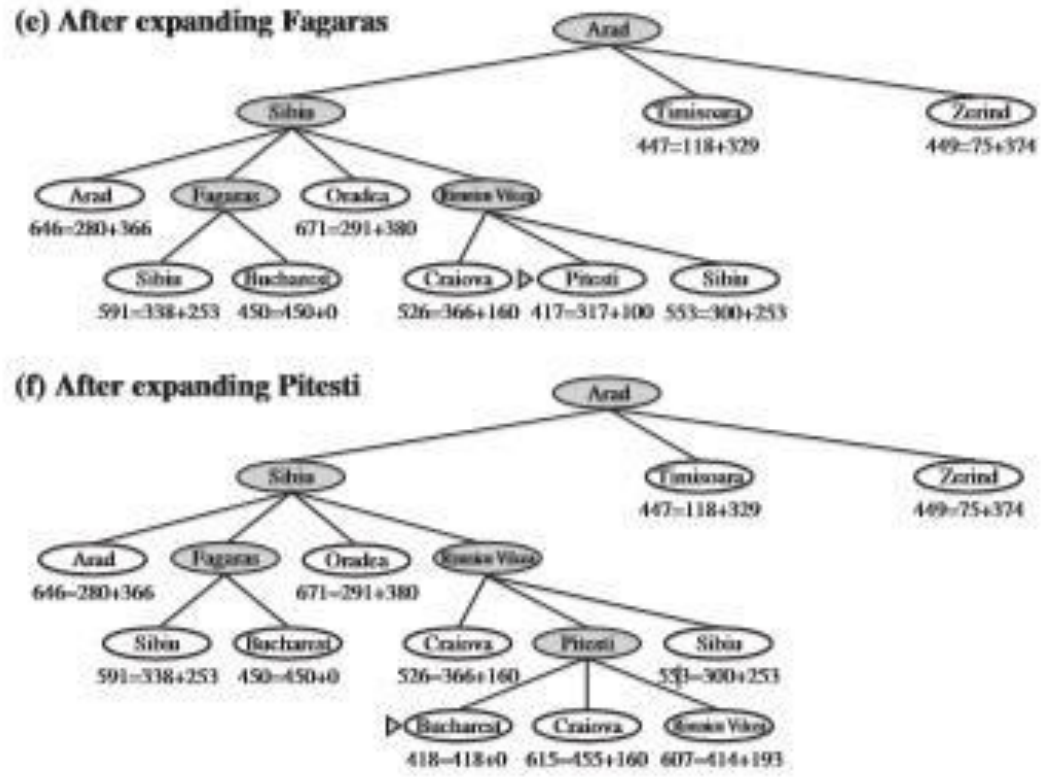


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.