AI: Introduction

Course Code: **CSC4226** Course Title: **Artificial Intelligence and Expert System**

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**Theory-01**	**Week No:**	**One**	**Semester:**	
Instructor	*Dr. Muhammad Firoz Mridha, Associate Professor, Dept of CS* *Email: firoz.mridha@aiub.edu*				

Lecture Outline

1. What is AI?
2. The Foundations of AI.
3. Brief History of AI
4. The State of the Art.
5. Course Outline by Topics and Weeks.

Course Objectives

- Get an overview of artificial intelligence (AI) principles and approaches.
- Develop a basic understanding of the building blocks of AI as presented in terms of intelligent agents: Search, Knowledge representation, inference, logic, learning.
- Develop a brief overview of AI applications: Expert Systems and Planners.
- Follow AI literature with the ability to go on to independent work in the field.

What is Artificial Intelligence ?

- John McCarthy, who coined the term Artificial Intelligence in 1956, defines it as "the science and engineering of making intelligent machines", especially intelligent computer programs.
- Artificial Intelligence (AI) is the intelligence of machines and the branch of computer science that aims to create it.
- Intelligence is the computational part of the ability to achieve goals in the world. Varying kinds and degrees of intelligence occur in people, many animals and some machines.
- AI is the study of the mental faculties through the use of computational models.
- AI is the study of : How to make computers do things which, at the moment, people do better.
- AI is the study and design of intelligent agents, where an intelligent agent is a system that perceives its environment and takes actions that maximize its chances of success.

Definition of AI: Outlined in Textbox

(a) '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)

(b) '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)

(c) '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)

(d) 'A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes' (Schalkoff, 1990)

'The branch of computer science that is concerned with the automation of intelligent behavior' (Luger and Stubblefield, 1993)

Intelligent Behavior

- **Perceiving** one's environment,
- **Acting** in complex environments,
- **Learning** and understanding from experience,
- **Reasoning** to solve problems and discover hidden knowledge,
- **Knowledge** applying successfully in new situations,
- **Thinking** abstractly, using analogies,
- **Communicating** with others, and more like
- **Creativity, Ingenuity, Expressive-ness, Curiosity.**

Understanding AI

- How knowledge is acquired, represented, and stored;
- How intelligent behavior is generated and learned;
- How motives, emotions, and priorities are developed and used;
- How sensory signals are transformed into symbols;
- How symbols are manipulated to perform logic, to reason about past, and plan for future;
- How mechanisms of intelligence produce the phenomena of illusion, belief, hope, fear, dreams, kindness and love."

Types of AI

Hard or Strong AI

- Generally, artificial intelligence research aims to create AI that can **replicate human intelligence** completely.
- Strong AI refers to a machine that approaches or supersedes human intelligence,
  - If it can do typically human tasks,
  - If it can apply a wide range of background knowledge and
  - If it has some degree of self-consciousness.
- Strong AI aims to build machines whose overall intellectual ability is indistinguishable from that of a human being.

Types of AI

Soft or Weak AI

- Weak AI refers to the use of software to study or accomplish specific problem solving or reasoning tasks that do not encompass the full range of human cognitive abilities.
- Example : a chess program such as **Deep Blue**.
- Weak AI does not achieve self-awareness; it demonstrates wide range of human-level cognitive abilities; it is merely an intelligent, a specific problem-solver.

Goals of AI

- The definitions of AI gives four possible goals to pursue :

  1. Systems that think like humans.
  2. Systems that think rationally.
  3. Systems that act like humans
  4. Systems that act rationally

- Traditionally, all four goals have been followed and the approaches were:

	Human-like	Rationally
Think	(1) Cognitive science Approach	(2) Laws of thought Approach
Act	(3) Turing Test Approach	(4) Rational agent Approach

- Most of AI work falls into category (2) and (4).

Goal of AI

Continued…

General AI Goal

- Replicate human intelligence : still a distant goal.
- Solve knowledge intensive tasks.
- Make an intelligent connection between perception and action.
- Enhance human-human, human-computer and computer to computer interaction / communication.

Engineering based AI Goal

- Develop concepts, theory and practice of building intelligent machines
- Emphasis is on system building.

Science based AI Goal

- Develop concepts, mechanisms and vocabulary to understand biological intelligent behavior.
- Emphasis is on understanding intelligent behavior.

AI Approaches

Cognitive Science : Think Humanly

- An exciting new effort to make computers think; that it is, the machines with minds, in the full and literal sense.
- Focus is not just on behavior and I/O, but looks at reasoning process.
- Computational model as to how results were obtained.
- Goal is not just to produce human-like behavior but to produce a sequence of steps of the reasoning process, similar to the steps followed by a human in solving the same task.

AI Approaches

Laws of Thought: Think Rationally

- The study of mental faculties through the use of computational models; that it is, the study of the computations that make it possible to perceive, reason, and act.
- Focus is on inference mechanisms that are provably correct and guarantee an optimal solution.
- Develop systems of representation to allow inferences to be like *"Socrates is a man. All men are mortal. Therefore Socrates is mortal."*
- Goal is to formalize the reasoning process as a system of logical rules and procedures for inference.
- The issue is, not all problems can be solved just by reasoning and inferences.

AI Approaches

Turing Test: Act Humanly

- The art of creating machines that perform functions requiring intelligence when performed by people; that it is the study of, how to make computers do things which at the moment people do better.
- Focus is on action, and not intelligent behavior centered around representation of the world.
- A Behaviorist approach, is not concerned with how to get results but to the similarity to what human results are.

AI Approaches

Turing Test

- 3 rooms contain: a person, a computer, and an interrogator.
- The interrogator can communicate with the other 2 by teletype (to avoid the machine imitate the appearance or voice of the person).
- The interrogator tries to determine which is the person and which is the machine.
- The machine tries to fool the interrogator to believe that it is the human, and the person also tries to convince the interrogator that it is the human.
- If the machine succeeds in fooling the interrogator, then conclude that the machine is intelligent.

AI Approaches

Turing Test : Capabilities Required to Pass Complete Turing Test

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

AI Approaches

Rational Agent: Act Rationally

- Tries to explain and emulate intelligent behavior in terms of computational processes; that it is concerned with the automation of intelligence.
- Focus is on systems that act sufficiently if not optimally in all situations;
- It is passable to have imperfect reasoning if the job gets done.
- Goal is to develop systems that are rational and sufficient.

Course Code: **CSC4226** Course Title: **Artificial Intelligence and Expert System**

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**Two (2)**	**Week No:**	**Two (2)**	**Semester:**	
Lecturer:	*Dr. Muhammad Firoz Mridha, Associate Professor, Dept. of CS* *Firoz.mridha@aiub.edu*				

Lecture Outline

1. Agents and Environments
2. Good Behavior: The Concept of Rationality
3. The Nature of Environments
4. The Structure of Agents

INTELLIGENT AGENT

Agent: entity in a program or environment capable of generating action.

An agent uses perception of the environment to make decisions about actions to take.

The perception capability is usually called a sensor.

The actions can depend on the most recent perception or on the entire history (percept sequence).

AGENT V/S PROGRAM

Size - an agent is usually smaller than a program.

Purpose - an agent has a specific purpose while programs are multi-functional.

Persistence - an agent's life span is not entirely dependent on a user launching and quitting it.

Autonomy - an agent doesn't need the user's input to function.

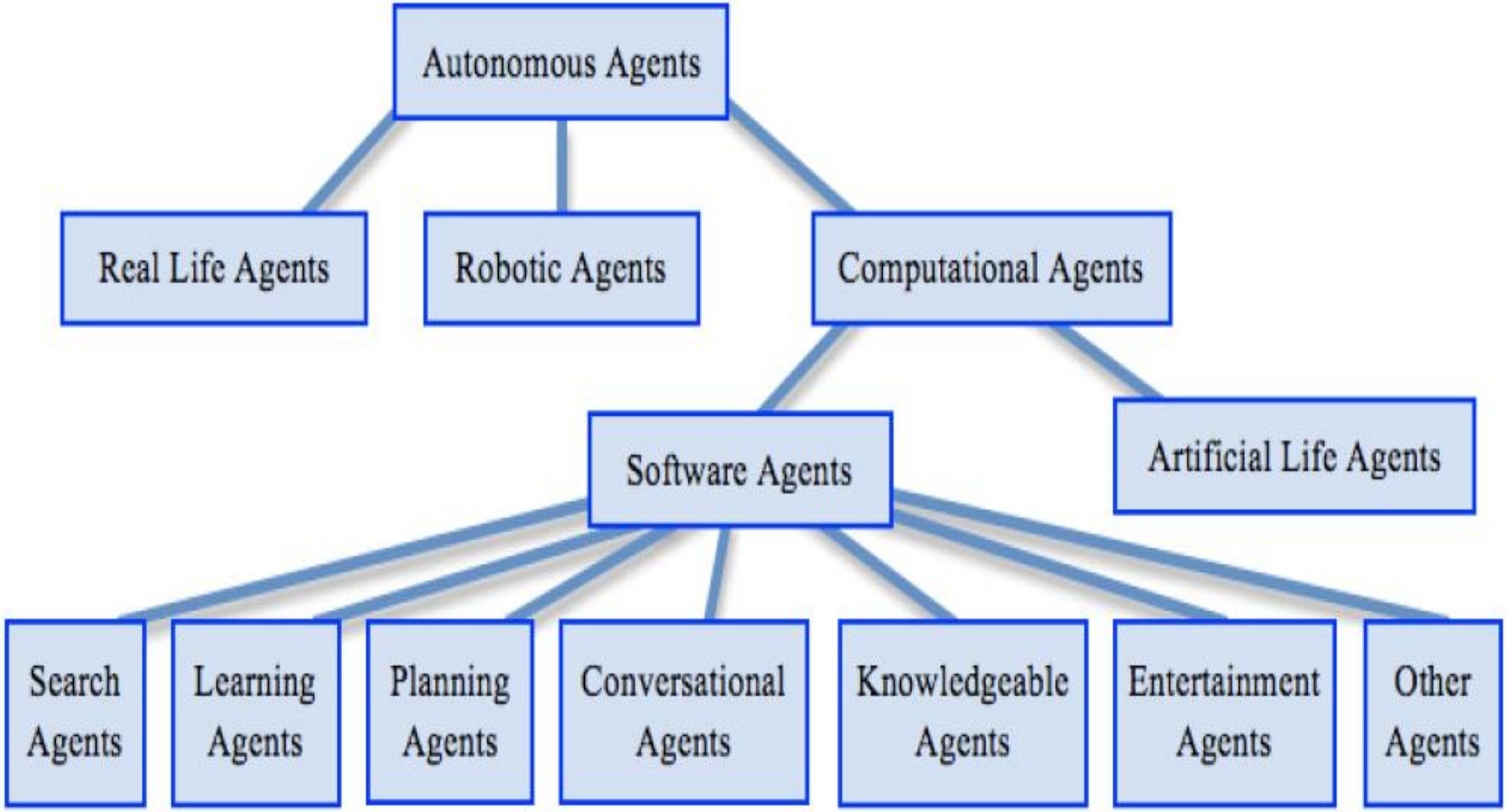
WHAT IS AN AGENT ?

Perspective	Key Ideas	Some Application Areas
Artificial Intelligence	An agent is embodied (i.e. situated) in an environment and makes it own decisions. It perceives the environment through sensors and acts on the environment through actuators.	Intelligent Agents. Intelligent Systems. Robotics.
Distributed Computing	An agent is an autonomous software process or thread.	3-Tier model (using agents). Peer-to-peer networks. Parallel and Grid Computing.
Internet-based Computing	The agent performs a task on behalf of a user. i.e. The agent acts as a proxy; the user cannot perform (or chooses not to perform) the task themselves.	Web spiders and crawlers. Web scrapers. Information Gathering, Filtering and Retrieval.
Simulation and Modelling	An agent provides a model for simulating the actions and interactions of autonomous individuals in a network.	Game Theory. Complex Systems. Multi-agent systems. Evolutionary Programming.

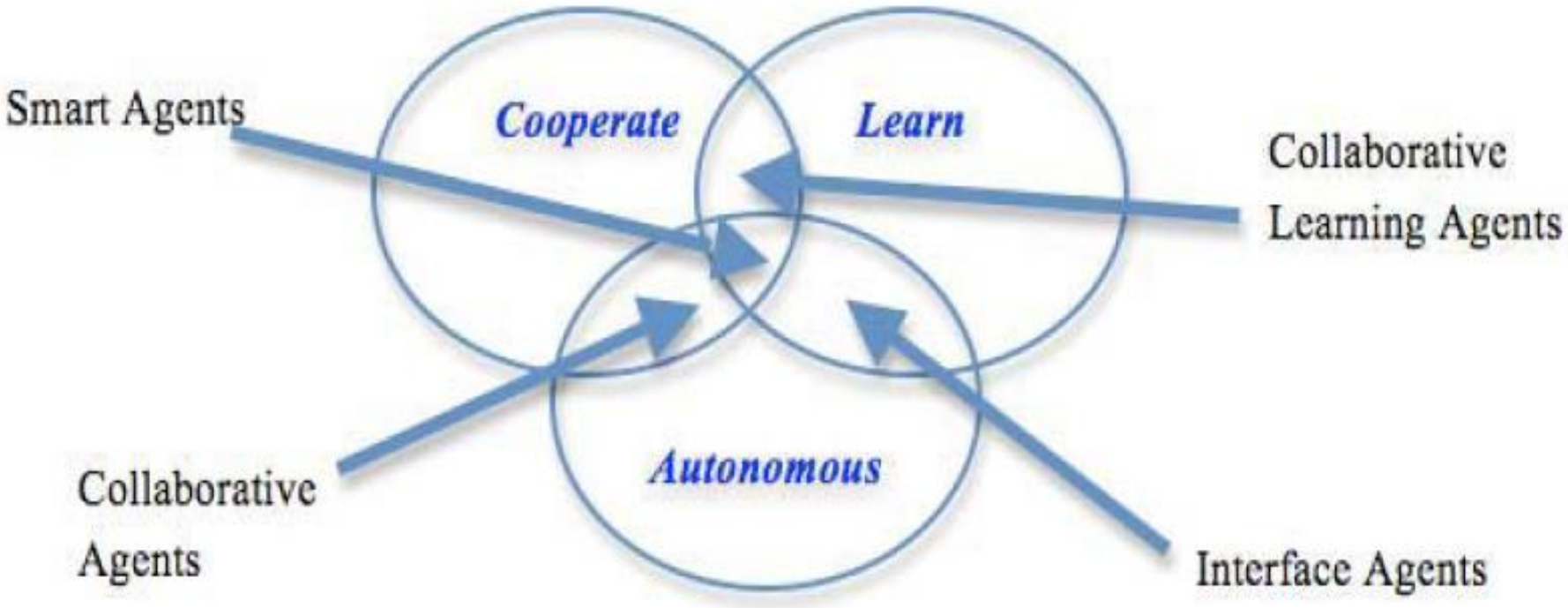
ROBOTS AND THEIR APPLICATION

Bot name(s)	Description
Chatterbots	Agents that are used for chatting on the Web.
Annoybots	Agents that are used to disrupt chat rooms and newsgroups.
Spambots	Agents that generate junk email ('spam') after collecting Web email addresses.
Mailbots	Agents that manage and filter e-mail (e.g. to remove spam).
Spiderbots	Agents that crawl the Web to scrape content into a database (e.g. Googlebot). For search engines (e.g. Google) this is then indexed in some manner.
Infobots	Agents that collect information. e.g. 'Newsbots' collect news; 'Hotbots' find the hottest or latest site for information; 'Jobbots' collect job information.
Knowbots or Knowledgebots	Agents that seek specific knowledge. e.g. 'Shopbots' locate the best prices; 'Musicbots' locate pieces of music, or audio files that contain music.

TAXONOMY OF AUTONOMOUS AGENT



AGENT TOPOLOGY



DESIRABLE PROPERTIES OF AGENT

Property	Description
Autonomy	The agent exercises control over its own actions; it runs asynchronously.
Reactivity	The agent responds in a timely fashion to changes in the environment and decides for itself when to act.
Proactivity	The agent responds in the best possible way to possible future actions that are anticipated to happen.
Social ability (Ability to communicate)	The agent has the ability to communicate in a complex manner with other agents, including people, in order to obtain information or elicit help in achieving its goals.
Ability to set goals	The agent has a purpose.
Temporal continuity	The agent is a continually running process.

DESIRABLE PROPERTIES OF AGENT

Continued..

Property	Description
Mobility	The agent is able to transport itself around its environment.
Adaptivity	The agent has the ability to learn. It is able to change its behaviour based on the basis of its previous experience.
Benevolence	The agent performs its actions for the benefit of others.
Rationality	The agent makes rational, informed decisions.
Collaborative ability	The agent collaborates with other agents or humans to perform its tasks.
Flexibility	The agent is able to dynamically respond to the external state of the environment by choosing its own actions.
Personality	The agent has a well-defined, believable personality and emotional state.
Cognitive ability	The agent is able to explicitly reason about its own intentions or the state and plans of other agents.
Versatility	The agent is able to have multiple goals at the same time.
Veracity	The agent will not knowingly communicate false information.
Persistency	The agent will continue steadfastly in pursuit of any plan.

DESIRABLE PROPERTIES OF AGENT

Continued..

Property	Description
Coordination	The agent has the ability to manage resources when they need to be distributed or synchronised.
Cooperation	The agent makes use of interaction protocols beyond simple dialogues, for example negotiations on finding a common position, solving conflicts or distributing tasks
Ability to plan	The agent has the ability to pro-actively plan and coordinating its reactive behavior in the presence of the dynamical environment formed by the other acting agents.

AGENT AND ENVIRONMENT

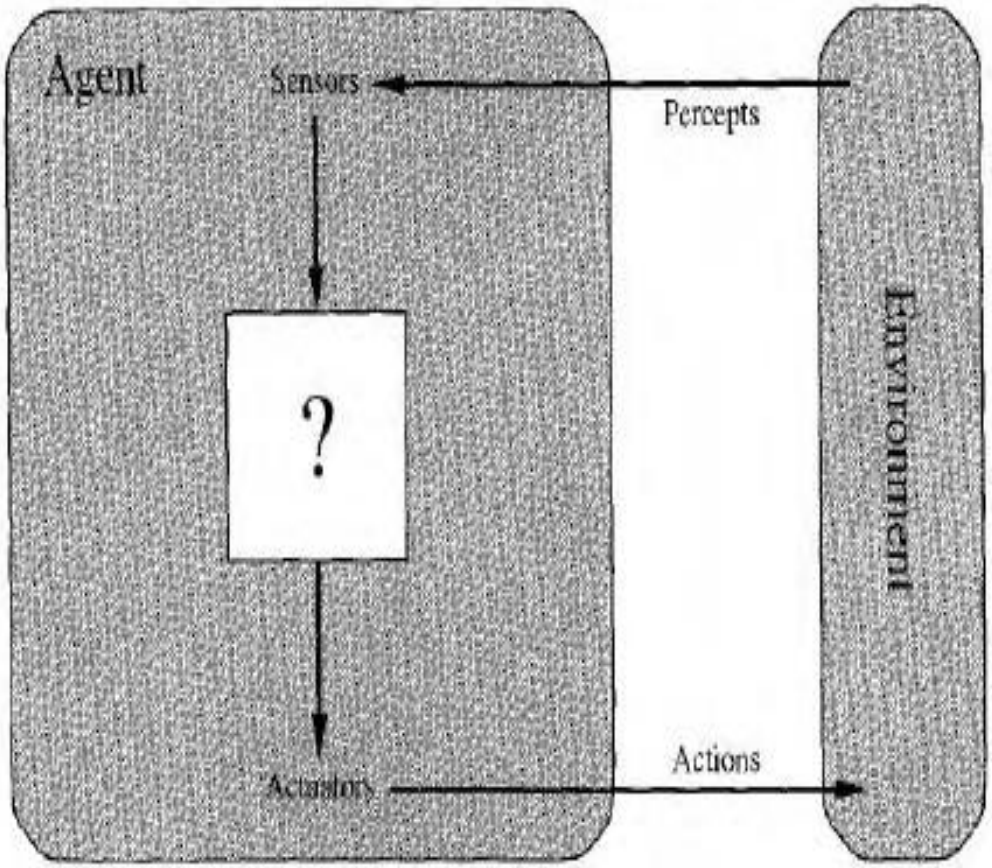


Figure 2.1 Agents interact with environments through sensors and actuators.

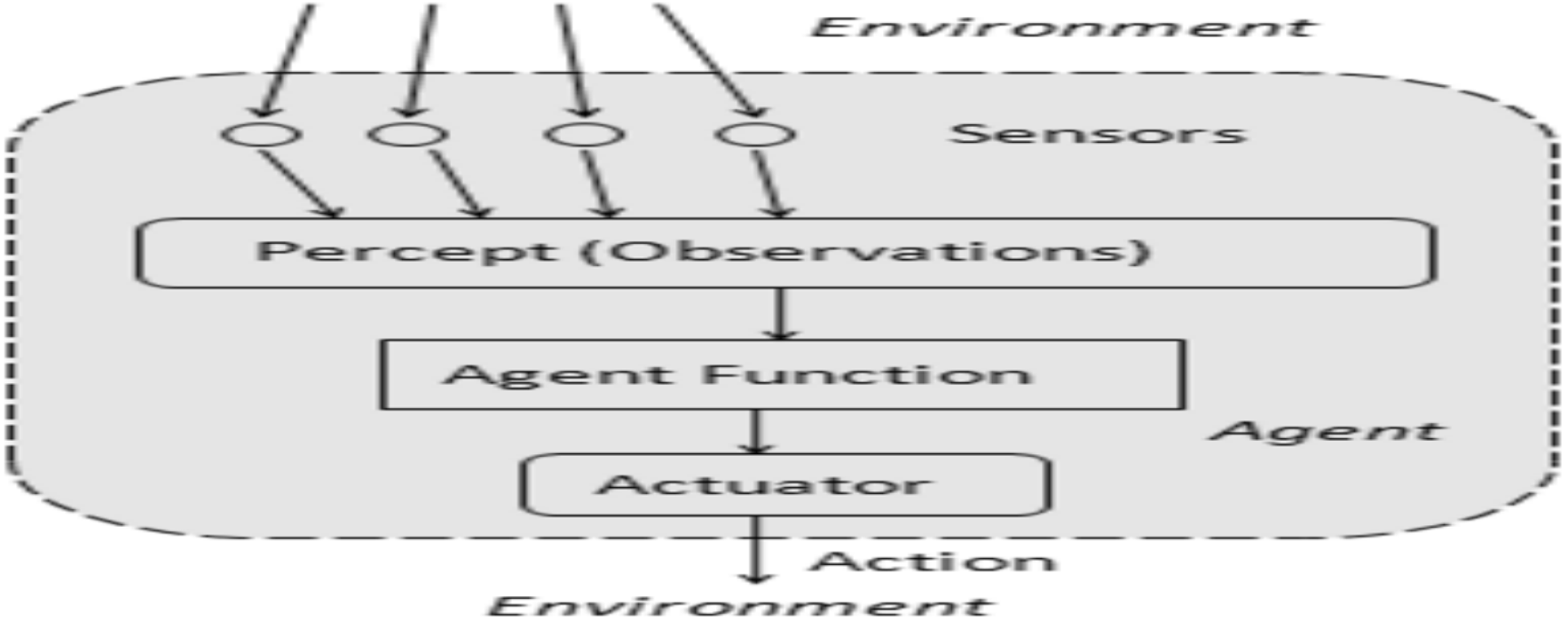
ENVIRONMENT
SENSOR
ACTUATORS

PERCEPT
PERCEPT SEQUENCE

AGENT FUNCTION
AGENT PROGRAM

AGENT FUNCTION

The agent function is a mathematical function that maps a sequence of perceptions into action.
The function is implemented as the agent program.
The part of the agent taking an action is called an actuator.
environment -> sensors -> agent function -> actuators -> environment



VACUUM CLEANING AGENT

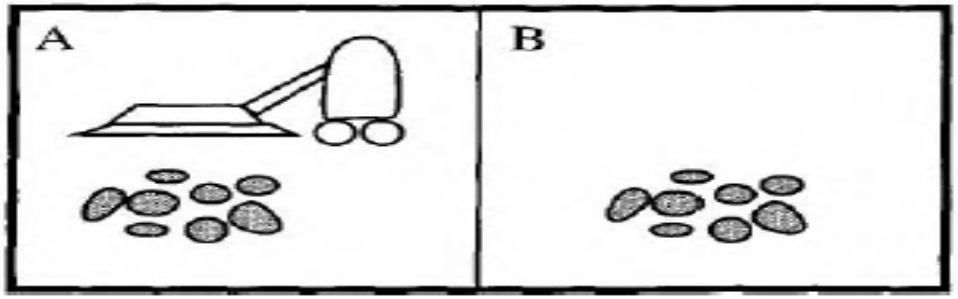


Figure 2.2 A vacuum-cleaner world with just two locations.

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

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

GOOD BEHAVIOR:
THE CONCEPT OF RATIONALITY

Rational Agent

- one does the right thing
- Every entry in the table for the agent function is filled out correctly

What does it mean to do the right thing ?

by considering the *consequences* of the agent's behavior

Agent - >plunked down in an environment

-> generates a sequence of actions

-> according to the percepts it receives.

sequence of actions -> causes

-> the environment

-> to go through a sequence of states.

If the sequence of states is desirable, then the agent has performed well.

performance Measure –

evaluates any given sequence of environment states

RATIONAL AGENT

A rational agent is one that can take the right decision in every situation.

Performance measure: a set of criteria/test bed for the success of the agent's behavior.

The performance measures should be based on the desired effect of the agent on the environment.

RATIONALITY

The agent's rational behavior depends on:

- the **performance measure** that **defines success**
- the agent's **knowledge** of the **environment**
- the **action** that it is capable of **performing**
- the **current sequence** of **perceptions**.
- **Definition:** for every possible percept sequence, the agent is expected to take an action that will maximize its performance measure.

SPECIFYING THE TASK ENVIRONMENT: PEAS DESCRIPTION

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

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

PEAS: Examples

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, beaters, 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 ENVIRONMENT

Fully v/s Partially Observable
Unobservable

Single Agent v/s Multi Agent

Deterministic v/s Stochastic

Nondeterministic

Uncertain Environment

Episodic V/s Sequential

Static v/s Dynamic

Discrete v/s Continuous

Known v/s Unknown

PROPERTIES OF TASK ENVIRONMENT

Attributes	Description
Observable and partially observable.	An agent can be considered to be an agent only if it has the ability to observe its environment (and conversely, the environment itself must therefore be observable). In some cases, usually simple environments, or software-generated environments, all of the environment may be observable. Usually, however, the environment may only be partially observable.
Deterministic, stochastic and strategic.	A fully deterministic environment is one where any future state of the environment can be completely determined from a preceding state and the actions of the agent. An environment is stochastic if there is some element of uncertainty or outside influence involved. Note that if a deterministic environment is only partially observable to the agent, it will appear to be stochastic from the agent's point of view. A strategic environment is fully determined by the preceding state combined with the actions of multiple agents.
Episodic and sequential.	The task environment is episodic if each of the agent's tasks do not rely on past performance, or cannot affect future performance. If not, then it is sequential.
Static and dynamic.	A static environment does not change. In a dynamic environment, if an agent does not respond to the change, this is considered as a choice to do nothing.
Discrete and continuous.	A discrete environment has a finite number of possible states, whereas the number of states in a continuous environment is infinite.
Single-agent and multiple-agent.	A multiple-agent environment the agent that acts cooperatively or competitively with another agent. If this is not the case, then from the perspective of the agent, the other agents can be viewed as part of the environment that is behaving stochastically.

TASK ENVIRONMENT: EXAMPLES

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

Architecture: computing device + physical sensors and actuators

architecture makes the percepts from the sensors available to the agent program,

runs the program,

and feeds the program's action choices to the actuators as they are generated.

- **Agent programs**
  - take the current percept as input from the sensors and return an action to the actuators.

AGENT EXAMPLE: TABLE DRIVEN AGENT

Table-driven agents: the function consists in a lookup table of actions to be taken for every possible state of the environment.

If the environment has **n** variables, each with **t** possible states, then the table size is $\mathbf{t^n}$.

Only works for a small number of possible states for the environment.

Simple reflex agents: deciding on the action to take based only on the current perception and not on the history of perceptions.

Based on the condition-action rule: **(if (condition) action)**

Works if the environment is fully observable

TABLE-DRIVEN-AGENT

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.

VACUUM CLEANING AGENT: Table Driven

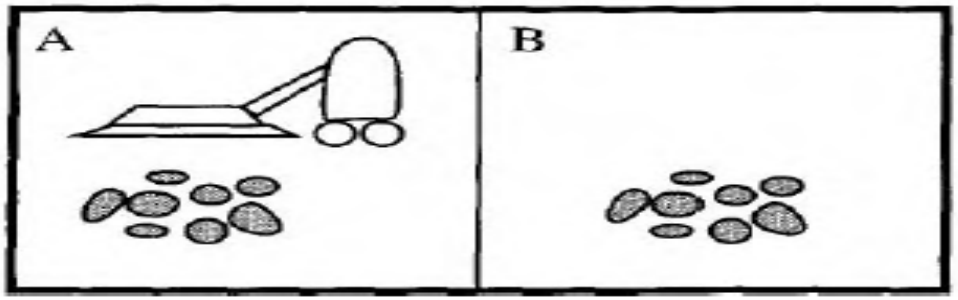


Figure 2.2 A vacuum-cleaner world with just two locations.

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

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

Table Drives

```
percepts = []
table = {}

def table_agent (percept):
        action = True
        percepts.append(percept)
        action = lookup(percepts, table)
        return action
```

LIMITATION OF TABLE-DRIVEN AGENT

(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

BASIC KINDS OF AGENT PROGRAMS

1. Simple reflex agents
2. Model-based reflex agents
3. Goal-based agents
4. Utility-based agents.

SIMPLE REFLEX AGENTS

select actions on the basis of the *current percept, ignoring the rest of the percept history.*

a ***condition-action rule***, written as

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

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.

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.

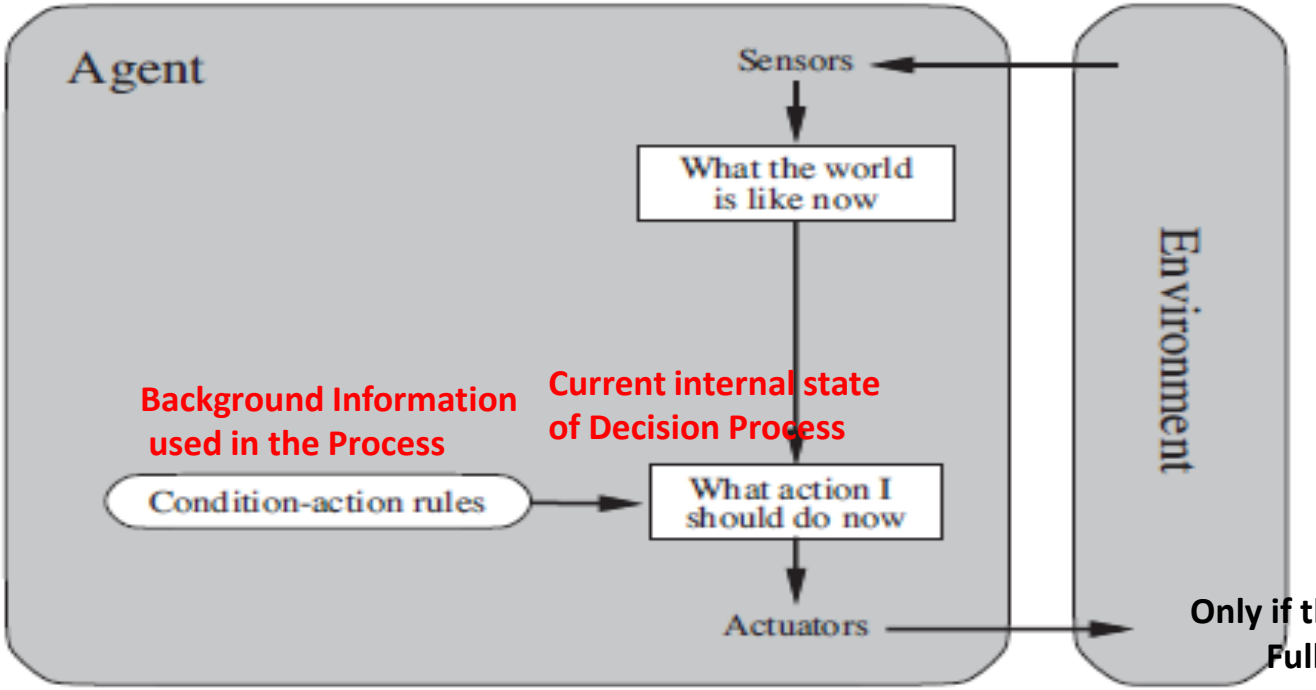


Figure 2.9 Schematic diagram of a simple reflex agent.

```
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
```

What will happen? when,
Vacuum Cleaner with poor perception!
Without location sensor !

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

*keep track of the part of the world it can't see now. [*handle partial observability]

model of the world

That is, the agent should maintain some sort of **internal state** that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.

Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program.

First, we need some information about how the world evolves independently of the agent [ex. overtaking car]

Second, we need some information about how the agent's own actions affect the world [ex. Turn steering wheel clockwise..]

This knowledge about "how the world works"—whether implemented in simple Boolean circuits or in complete scientific theories - is called a **model** of the world**.**

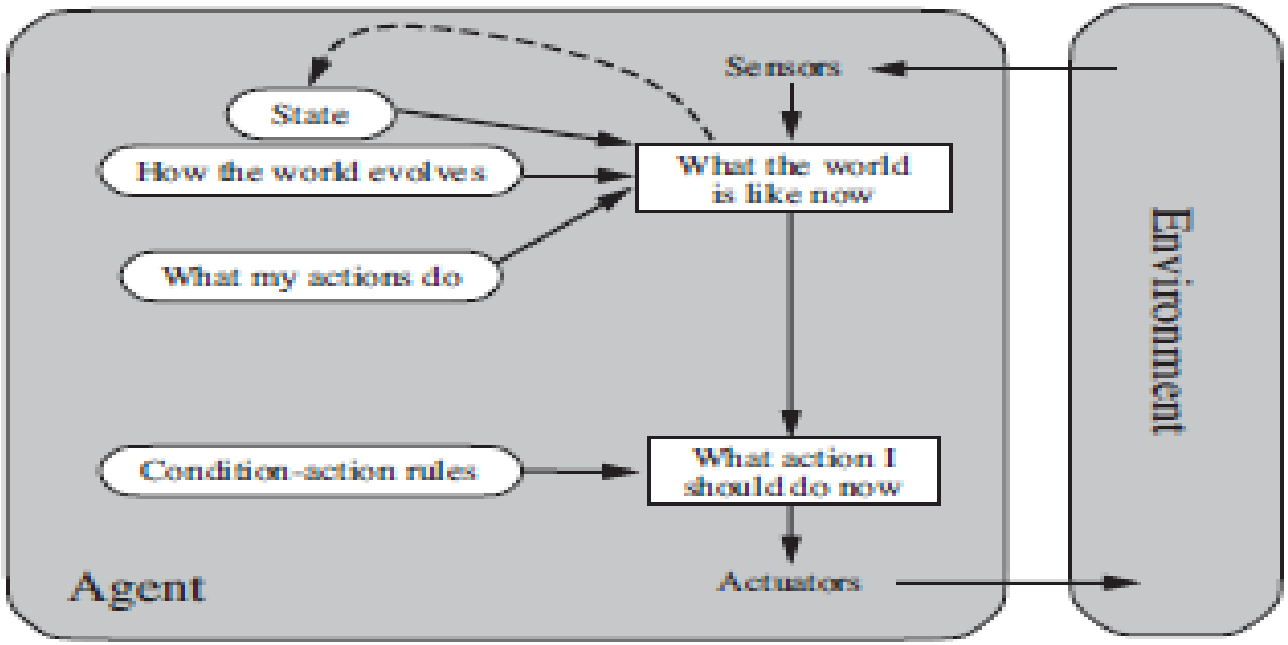


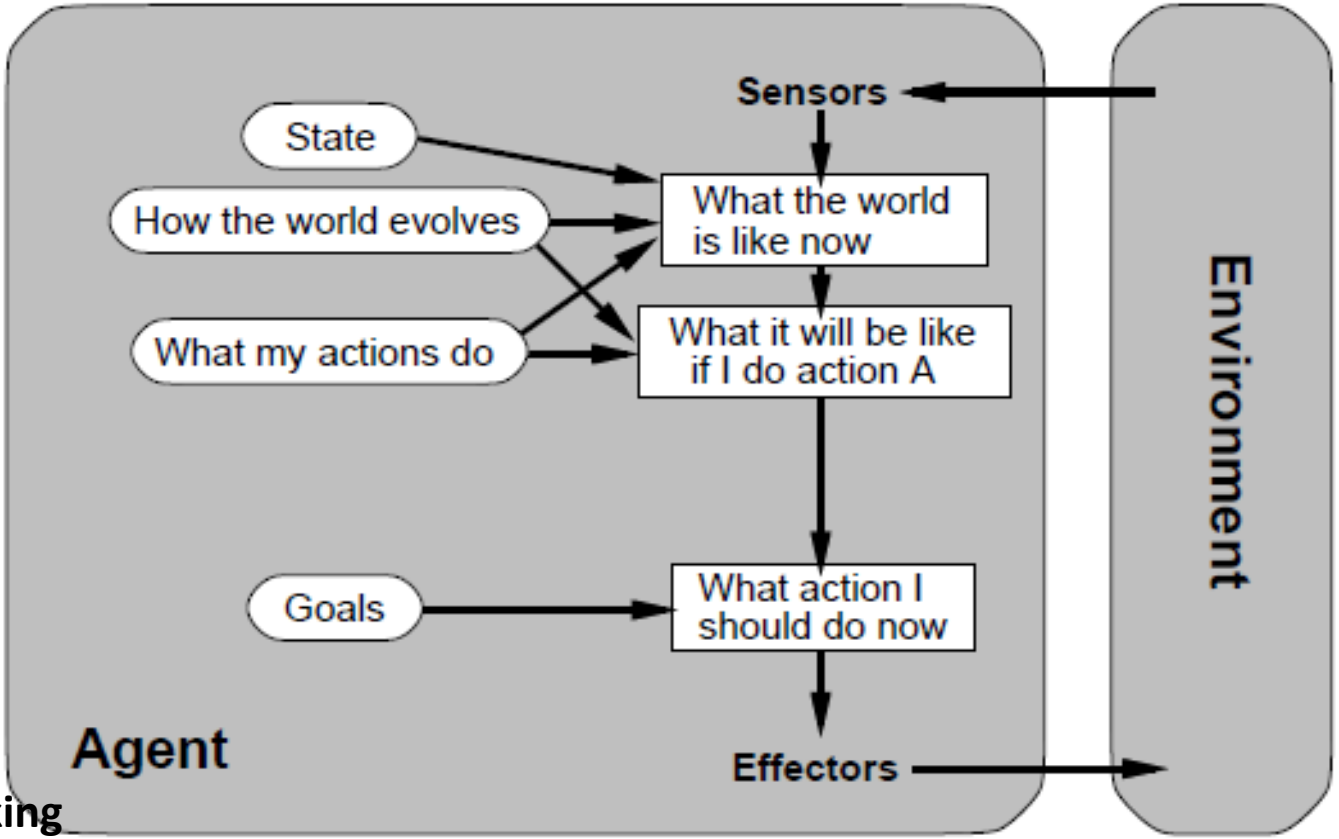
Figure 2.11 A model-based reflex agent.

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 AGENT



Searching

Planning

decision making

Figure 2.11 An agent with explicit goals.

UTILITY-BASED AGENTS

Goals alone are not enough to generate high-quality behavior in most environments.

An agent's **utility function** is essentially an internalization of the performance measure.

flexibility and learning

In two kinds of cases, goals are inadequate, but a utility-based agent can still make rational decisions

1. when there are conflicting goals, only some of which can *be achieved (for example, speed and safety), the utility function specifies the appropriate tradeoff.*

2. 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.

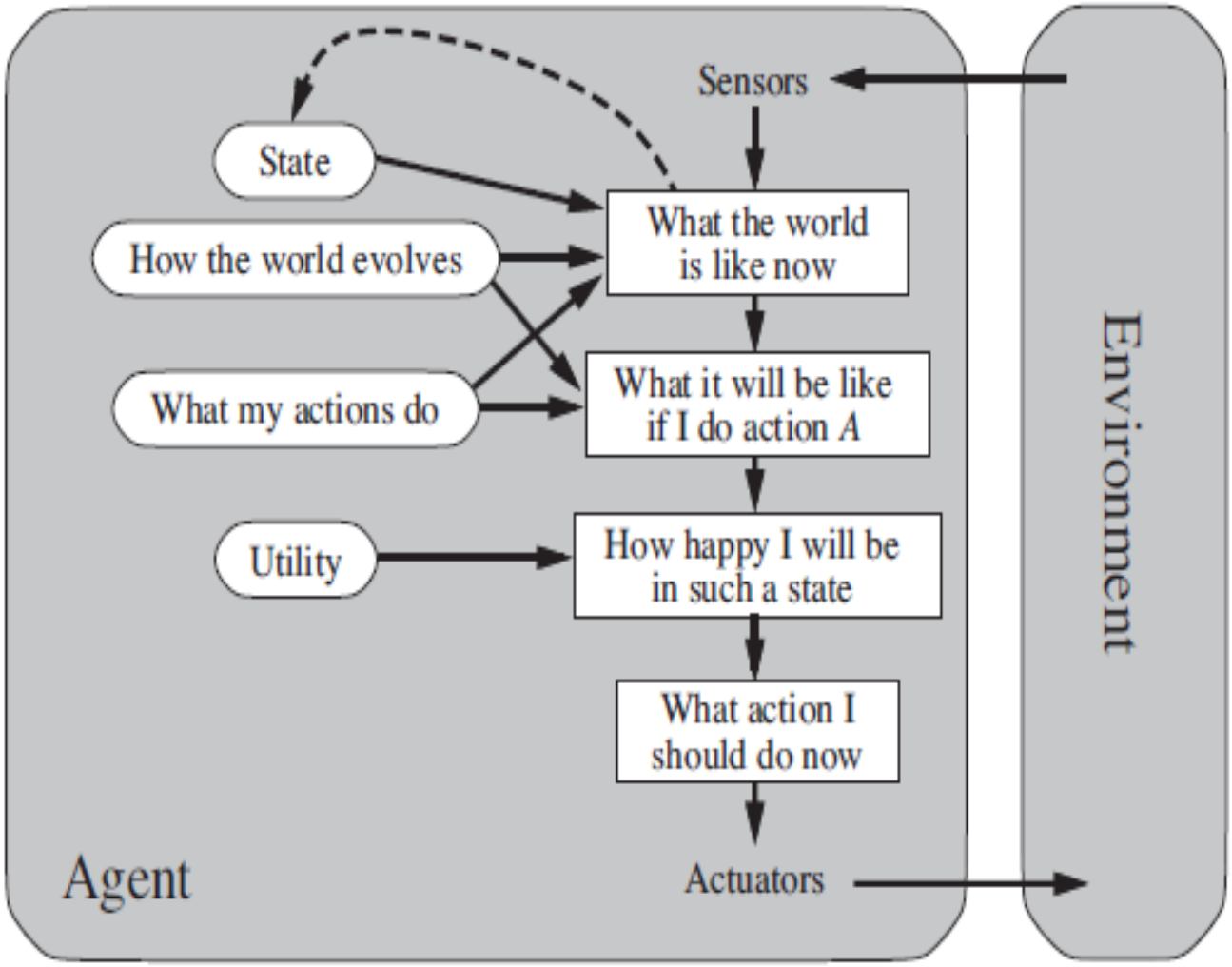


Figure 2.14 A model-based, utility-based agent. It uses a model of the world, along with a utility function that measures its preferences among states of the world. Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome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

LEARNING AGENT

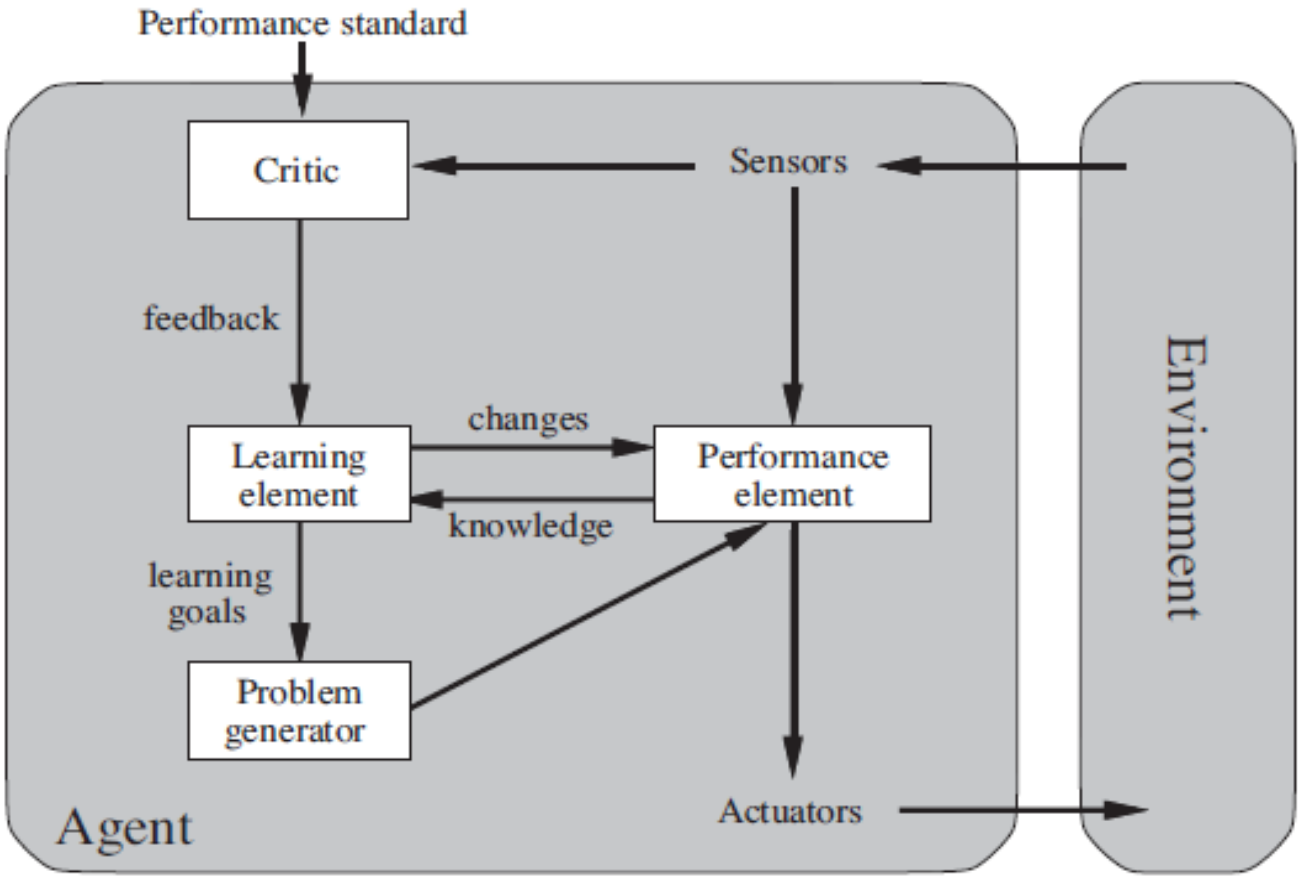


Figure 2.15 A general learning agent.

HOW COMPONENT OF AGENT PROGRAMS WORK

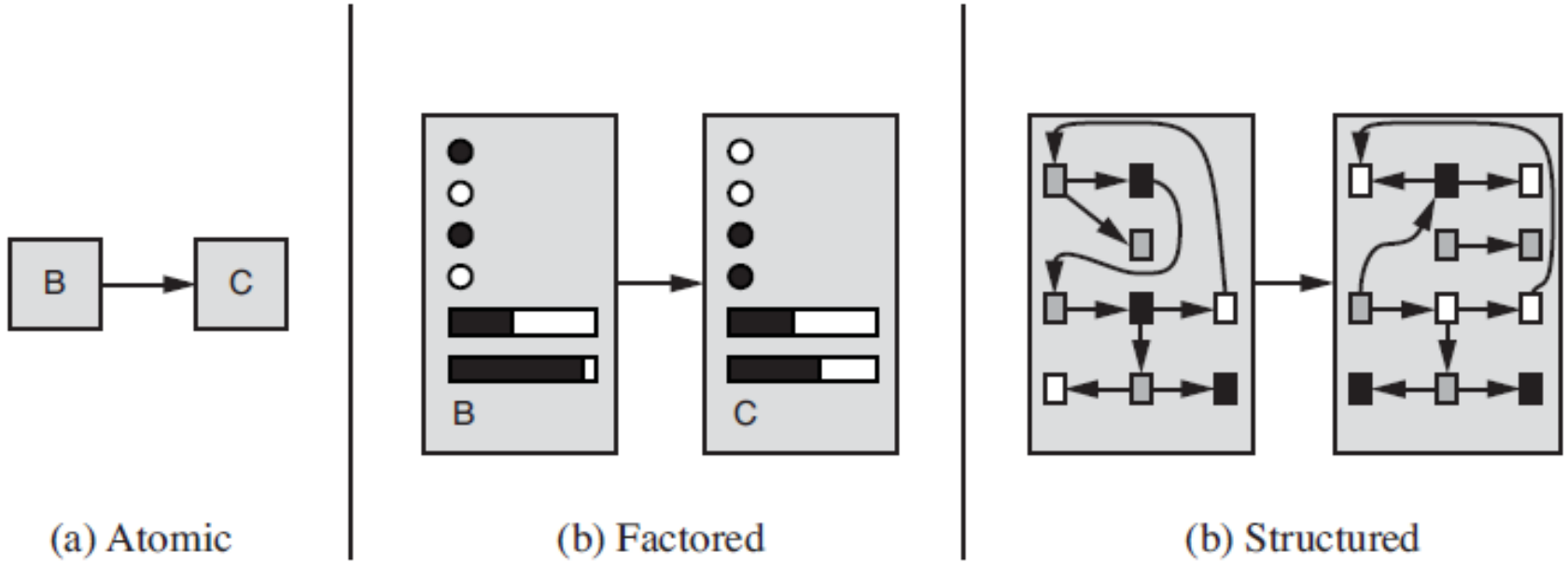


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed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

References

1. Chapter 2: Intelligent Agents , Pages 34-58
"Artificial Intelligence: A Modern Approach," by Stuart J. Russell and Peter Norvig,

Solving Problem by Searching : Uninformed Search

Course Code: **CSC4226** Course Title: **Artificial Intelligence and Expert System**

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**Three (3)**	**Week No:**	**Three (3)**	**Semester:**	
Lecturer:	*Dr. Muhammad Firoz Mridha, Associate Professor, Dept of CS* *Email: firoz.mridha@aiub.edu*				

Lecture Outline

1. Problem-Solving Agents
2. Example Problems
3. Searching for Solutions
4. Uninformed Search Strategies

PROBLEM SOLVING AGENTS

Simple Reflexive Agent

Base their action on a direct mapping from

STATEs to ACTIONs

Goal Based Agent

Consider future Actions and the desirability of their outcomes

Problem Solving Agent

Goal based Agent

Use Atomic representation of Environment

Planning Agent

Goal based Agent

Use more advanced factored/structured representation

FORMULATE-SEARCH-EXECUTE

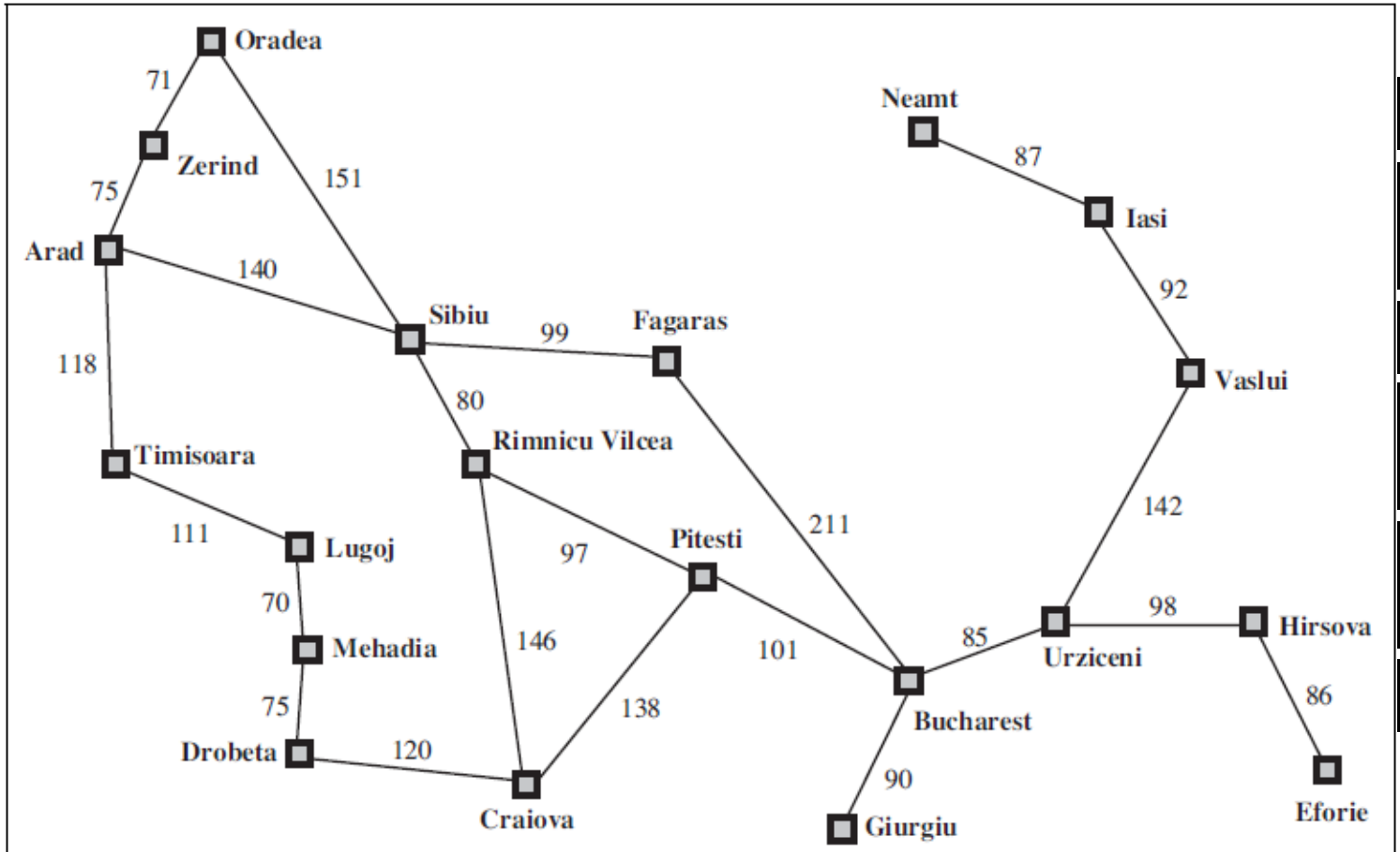


Figure 3.2 A simplified road map of part of Romania.

- Goal Formulation
- Problem Formulation
- Environment
- examining future actions
- Formulate, Search, Execution
- Open Loop System

GOAL FORMULATION

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

Goal: 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.

PROBLEM FORMULATION: BASED ON ENVIRONMENT

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

In an **Unknown** environment the agent will not know which of its possible actions is best, because it does not yet know enough about the state that results from taking each action.

If the agent has no additional information—i.e., if the environment is **unknown** in the sense defined in Section 2.3—then it is having no choice but to try one of the actions at **random**.

*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.***

SEARCH-SOLUTION-EXECUTE: OPEN LOOP SYSTEM

The process of looking for a sequence of actions that reaches the goal is called **search**.

Solution is the sequence of actions that takes any agent to the goal state, exactly those state the agent is satisfied.

A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence.

While the agent is executing the solution sequence it *ignores its percepts* when choosing an action because it knows in advance what they will be. An agent that carries out its plans with its eyes closed.

Control theorists call this an **open-loop** system, because ignoring the percepts breaks the loop between agent and environment.

WELL-DEFINED PROBLEMS

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.[2] For example, we have RESULT(In(Arad),Go(Zerind)) = In(Zerind) .

4. The **goal test**, which determines whether a given state is a goal state.

5. A **path cost** function that assigns a numeric cost to each path.

TOY PROBLEMS

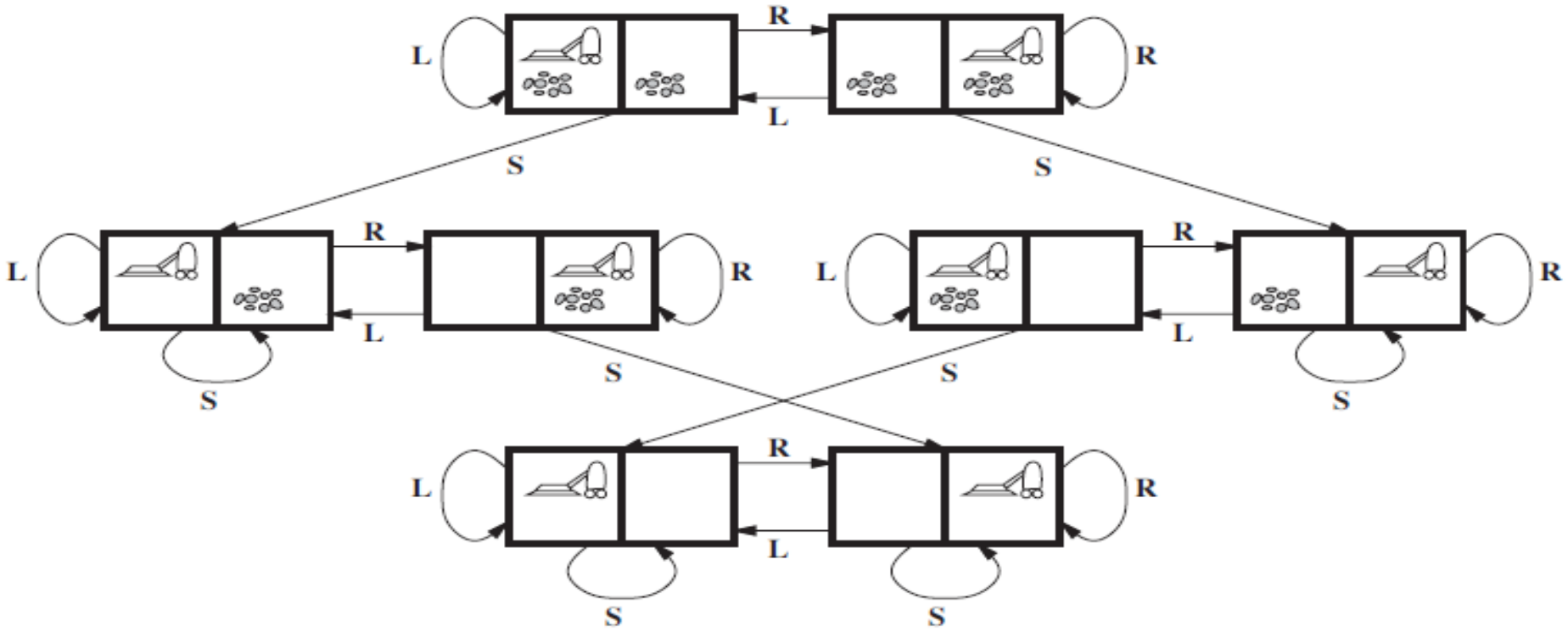


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

VACUUM WORLD: PROBLEM FORMULATION

- **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 $n \cdot 2^n$ 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.

8-PUZZLE: PROBLEM FORMULATION

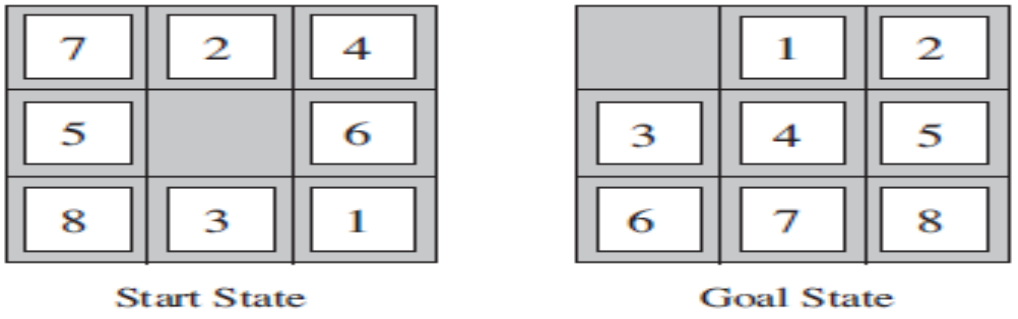


Figure 3.4 A typical instance of the 8-puzzle.

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

SEARCHING FOR SOLUTIONS

A **solution** is an action sequence, so search algorithms work by considering various possible action sequences

The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the branches are actions and the **nodes** correspond to states in the state space of the problem.

Expanding the current state is, applying each legal action to the current state, thereby **generating** a new set of states

SEARCH TREE

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.

SEARCH STRATEGY

the essence of search—following up one option now and putting the others aside for later, in case the first choice does not lead to a solution.

The set of all leaf nodes available for expansion at any given point is called the **frontier**.

The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.

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

TREE SEARCH / GRAPH SEARCH

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.

INFRASTRUCTURE FOR SEARCH ALGORITHMS

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

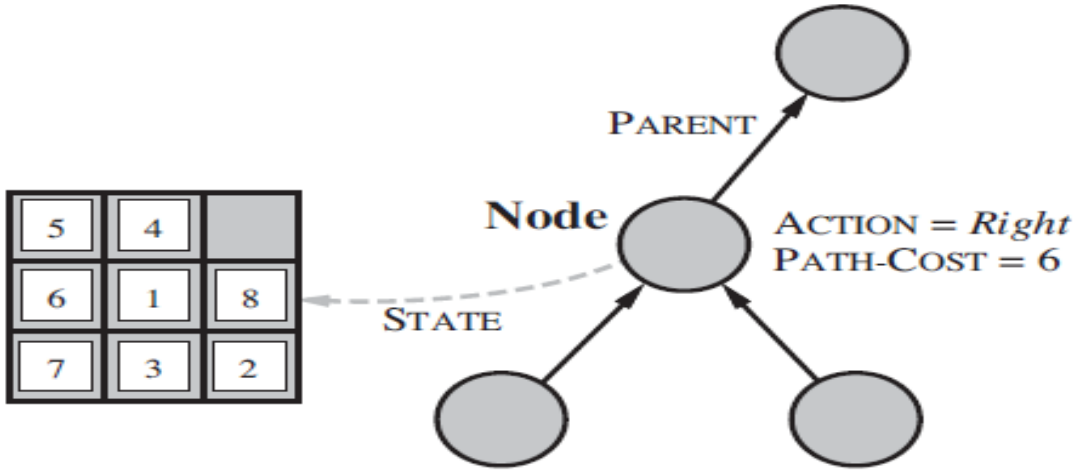


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

FRONTIER [FRINGE]

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. 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; and
3. the **priority queue**,

MEASURING PROBLEM-SOLVING PERFORMANCE

- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution, as defined on page 68?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

SEARCH ALGORITHMS

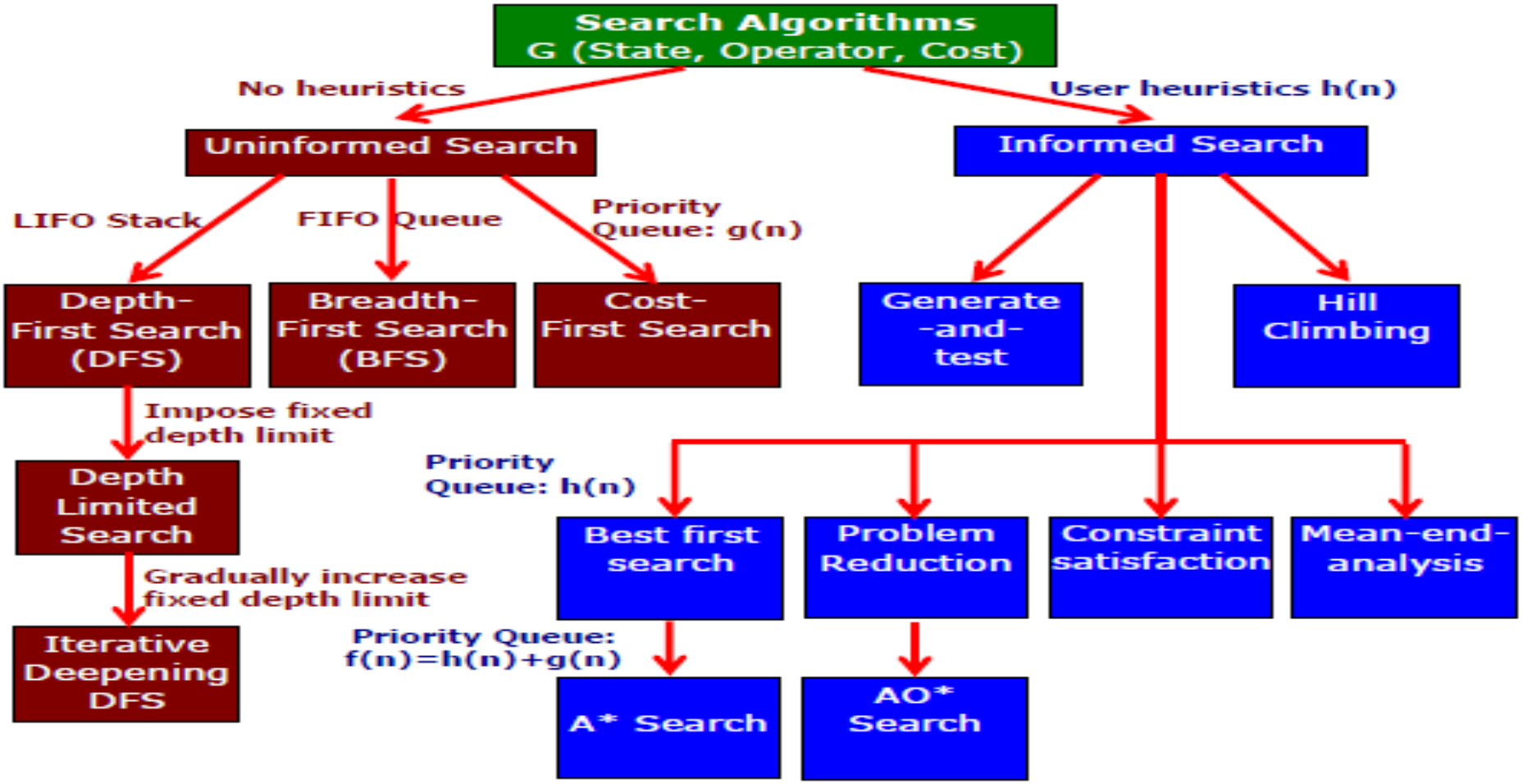
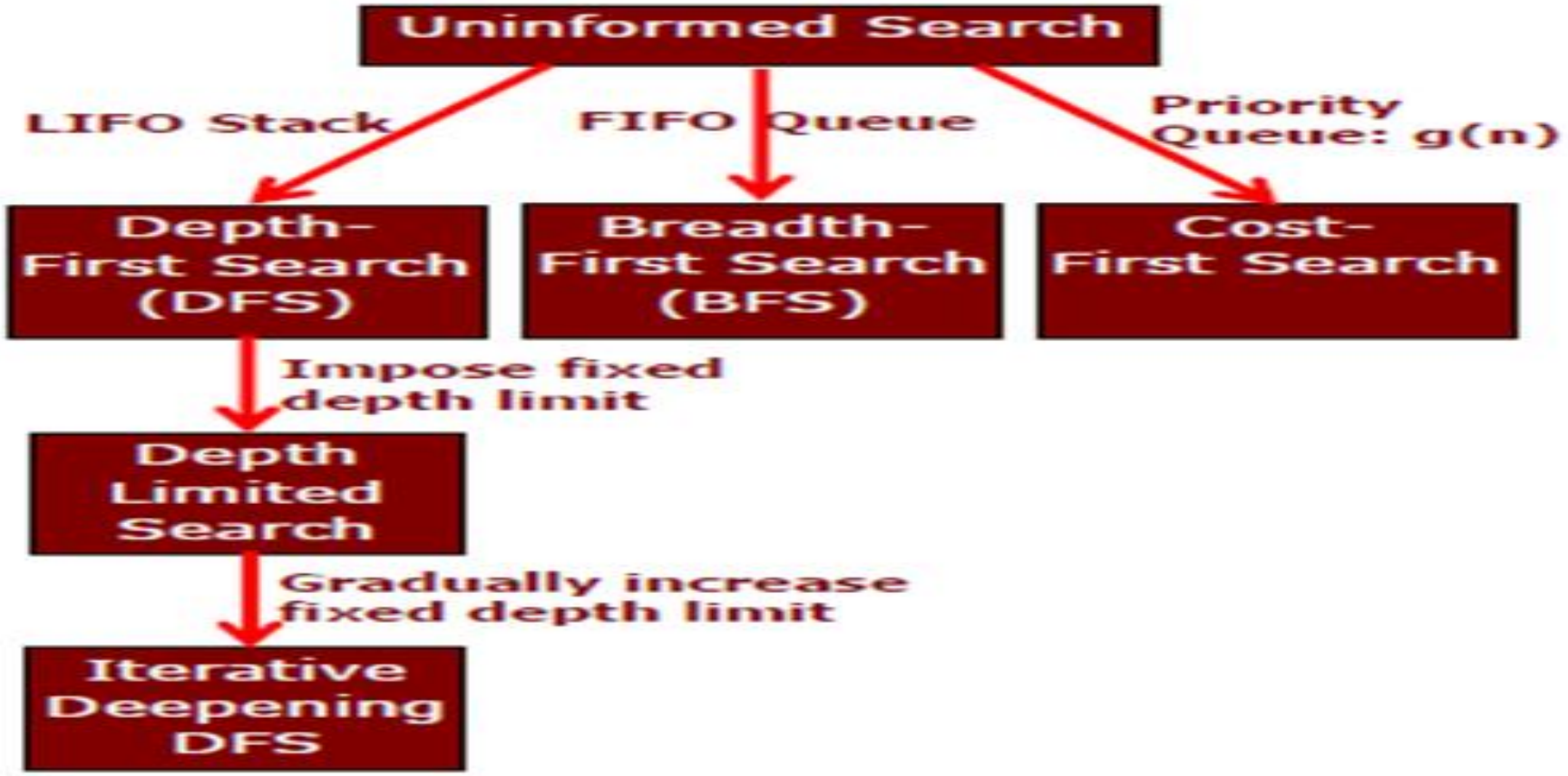


Fig. Different Search Algorithms

UNINFORMED SEARCH STRATEGIES



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 (Figure 3.7) 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 **goal test** is applied to each node when it is ***generated*** rather than when it is **selected** for expansion.
- Following the general template for graph search, discards any new path to a state already in the frontier or explored set

BREADTH-FIRST SEARCH: PSEUDOCODE

```
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: FOUR CRITERIA

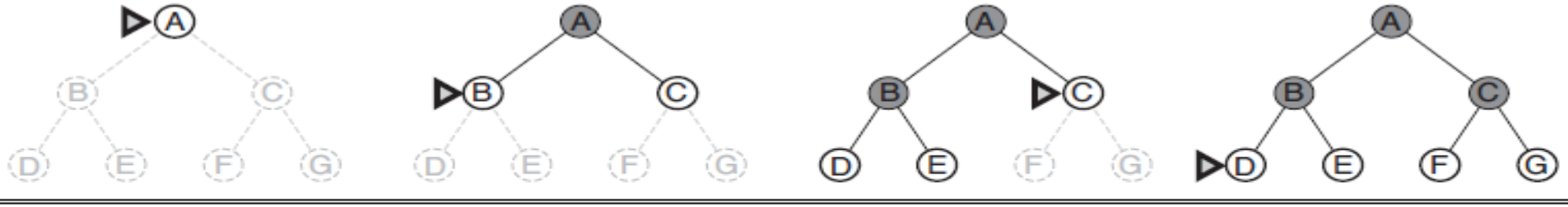


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

- ***complete***—if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes.
- Breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node.
- For uniform tree, the total number of nodes generated is $b + b^2 + b^3 + \cdots + b^d = O(b^d)$
- The time complexity would be $O(b^{d+1})$
- The space complexity is $O(b^d)$

BREADTH-FIRST SEARCH: MEMORY REQUIREMENTS

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

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

UNIFORM-COST SEARCH

- 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**
- Two other significant differences from breadth-first search

1. The **goal test** is applied to a node when it is ***selected for expansion*** rather than when it is first **generated**

2. A test is added in case a better path is found to a node currently on the frontier.

UNIFORM-COST SEARCH: PSEUDOCODE


```
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: ILLUSTRATION

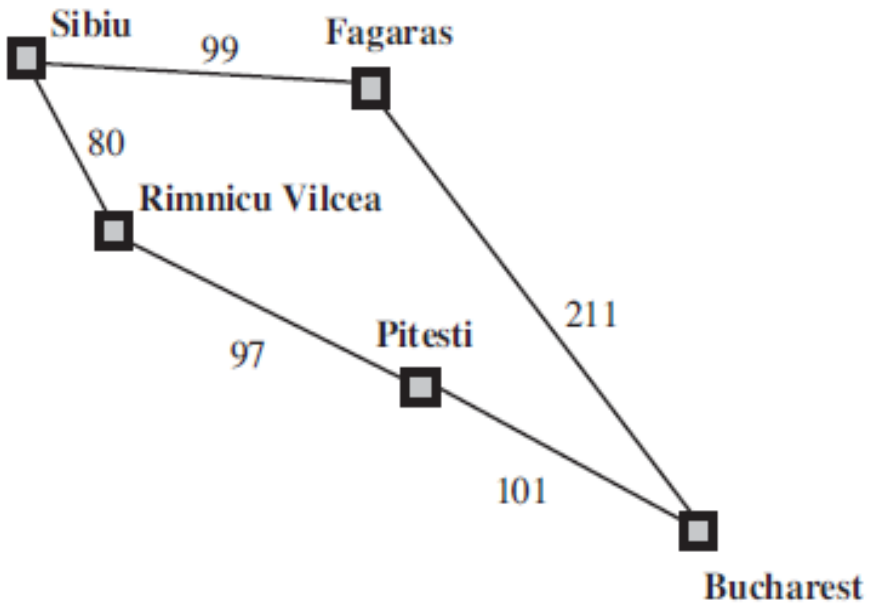


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

UNIFORM-COST SEARCH: OPTIMALITY

- Observe that whenever uniform-cost search selects a **node n** for expansion, the optimal path to that node has been found.
  - Were this not the case, there would have to be another frontier **node n'** on the optimal path from the **start node to n, n'** would have lower g-cost than n and would have been selected first

- 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.

UNIFORM-COST SEARCH: COMPLETENESS & COMPLEXITY

- Completeness is guaranteed provided the cost of every step exceeds some small positive constant ε.
- Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d.
- Instead, let C* be the cost of the optimal solution, and assume that every action costs at least ε .
- Then the algorithm's worst-case time and space complexity is $\mathbf{O(b^{1+[C^*/\varepsilon]})}$, which can be much greater than b^d.
- When all step costs are equal, $\mathbf{b^{1+[C*/\varepsilon]}}$ is just $\mathbf{b^{d+1}}$.

DEPTH-FIRST SEARCH

Depth-first search always expands the *deepest* node in the current frontier of the search tree.

The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors

As those nodes are expanded, they are dropped from the frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.

The depth-first search algorithm is an instance of the graph-search algorithm, where it uses a LIFO queue

DEPTH-FIRST SEARCH: SEARCH TREE

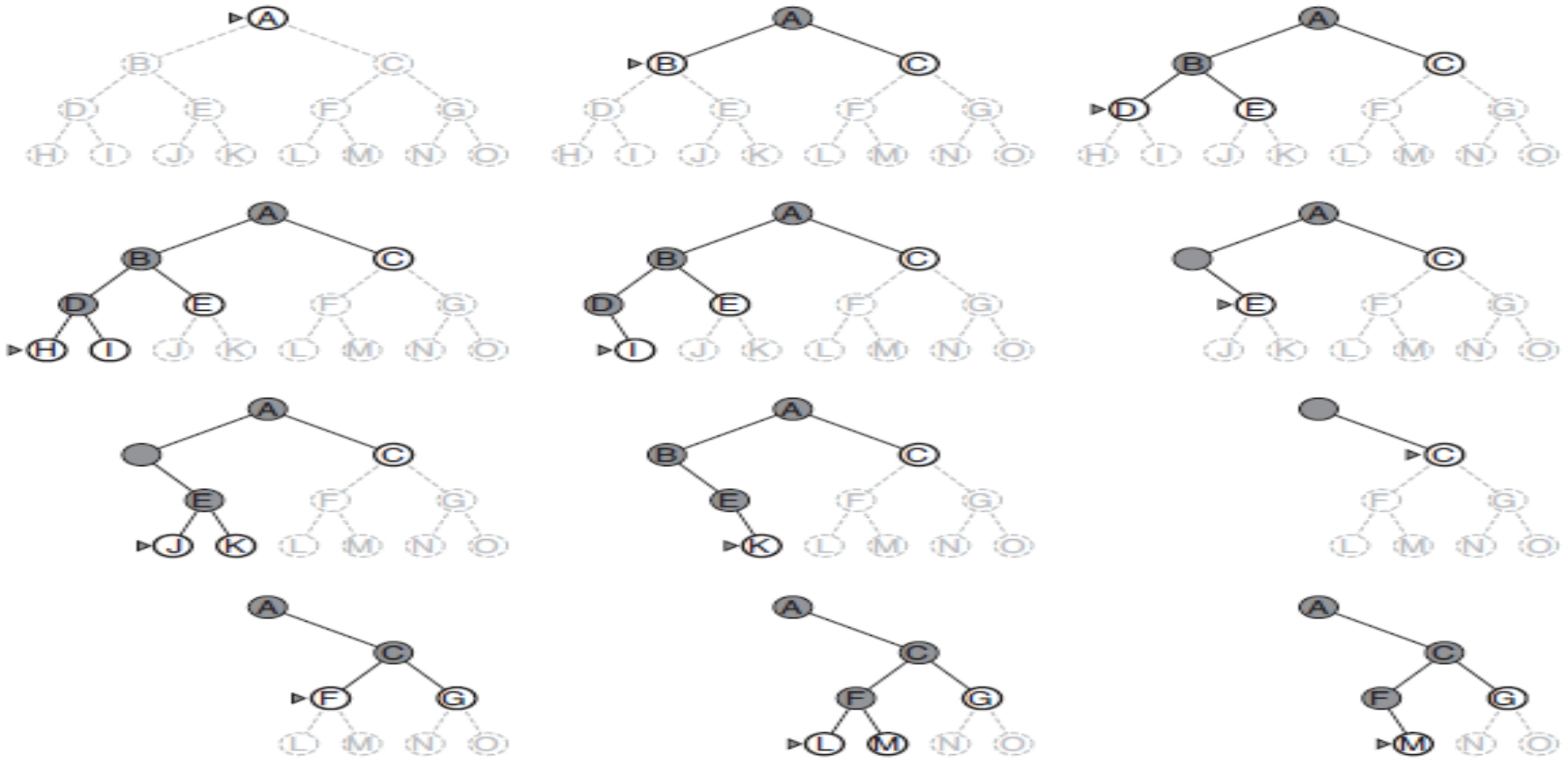


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

DEPTH-FIRST SEARCH:
Optimality, Complexity, Completeness

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** In infinite state spaces, both versions fail if an infinite non-goal path is encountered.

Both versions are **nonoptimal**. For example, in Figure 3.16, depth first search will explore the entire left subtree even if node C is a goal node. If node J were also a goal node, then depth-first search would return it as a solution instead of C, which would be a better solution; hence, depth-first search is not optimal.

A depth-first tree search, may generate all of the $\mathbf{O(b^m)}$ 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.

For a state **space** with branching factor b and maximum depth m, depth-first search requires storage of only **O(bm)** nodes.

DFS v/s BFS

- **Compare Algorithms**

Depth-first search

```
Put the root node on a stack;
while (stack is not empty)
{
   remove a node from the stack;

   if (node is a goal node)
      return  success;

   put all children of node
   onto the stack;
}
return failure;
```

Breadth-first search

```
Put the root node on a queue;
while (queue is not empty)
{
   remove a node from the  queue;

   if (node is a goal node)
      return  success;

   put all children of node
   onto the queue;
}
return failure;
```

DFS v/s BFS

- **Compare Features**
  - When succeeds, the goal node found is not necessarily minimum depth
  - Large tree, may take excessive long time to find even a nearby goal node
  - When succeeds, it finds a minimum-depth (nearest to root) goal node
  - Large tree, may require excessive memory
- **How to over come limitations of DFS and BFS ?**
  - Requires, how to combine the advantages and avoid disadvantages?
  - The answer is *"Depth-limited search"* .

    This means, perform Depth-first searches with a depth limit.

DEPTH-LIMITED SEARCH

- The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit .
- That is, nodes at depth are treated as if they have no successors.
- This approach is called **depth-limited search**
- Depth-limited search will also be nonoptimal if we choose $\ell > d$. Its time complexity is $O(b^{\ell})$ and its space complexity is $O(b\ell)$.
- Depth-first search can be viewed as a special case of depth-limited search with $\ell = \infty$.

DEPTH-LIMITED SEARCH: PSEUDOCODE

```
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
- Iterative deepening combines the benefits of depth-first and breadth-first search.

1. Like depth-first search, its **memory requirements are modest: O(bd)** to be precise.
2. 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

ITERATIVE DEEPENING DEPTH-FIRST SEARCH TREE

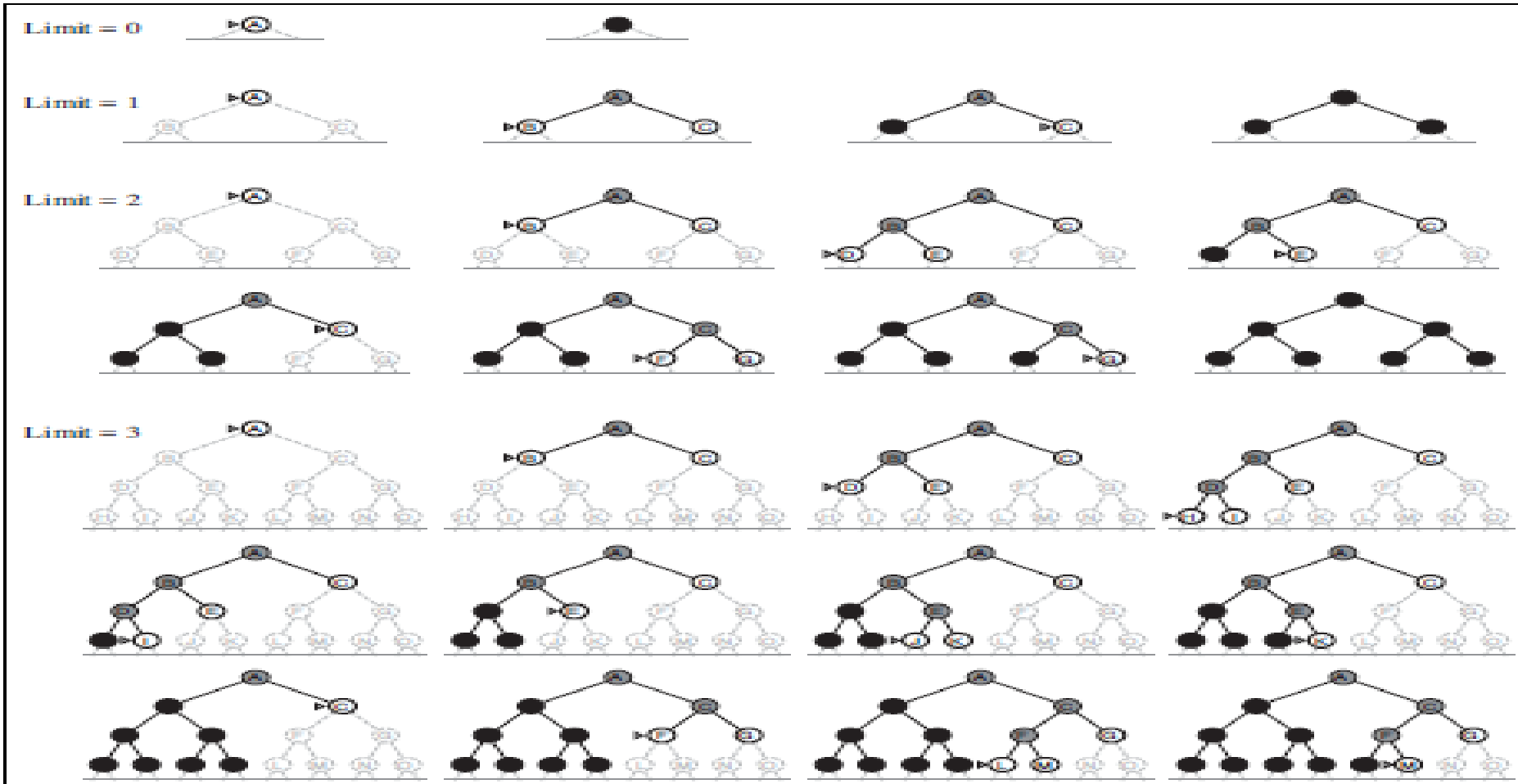


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

IDP: COMPLEXITY

- Iterative deepening search may seem wasteful because states are generated multiple times. It turns out this is not too costly.
- The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.
- In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times.
- So the total number of nodes generated in the worst case is

$$N(\mathrm{IDS}) = (d)b + (d-1)b^2 + \cdots + (1)b^d$$

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
- Bidirectional search is implemented by replacing the goal test with a check to see whether the frontiers of the two searches intersect;

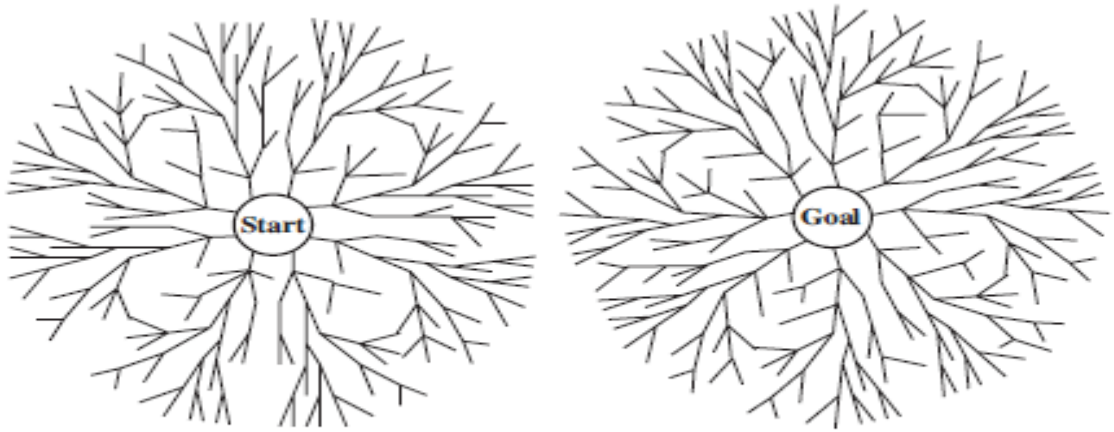


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

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.

INFORMED (HEURISTIC) SEARCH STRATEGIES

Course Code: **CSC4226** Course Title: **Artificial Intelligence and Expert System**

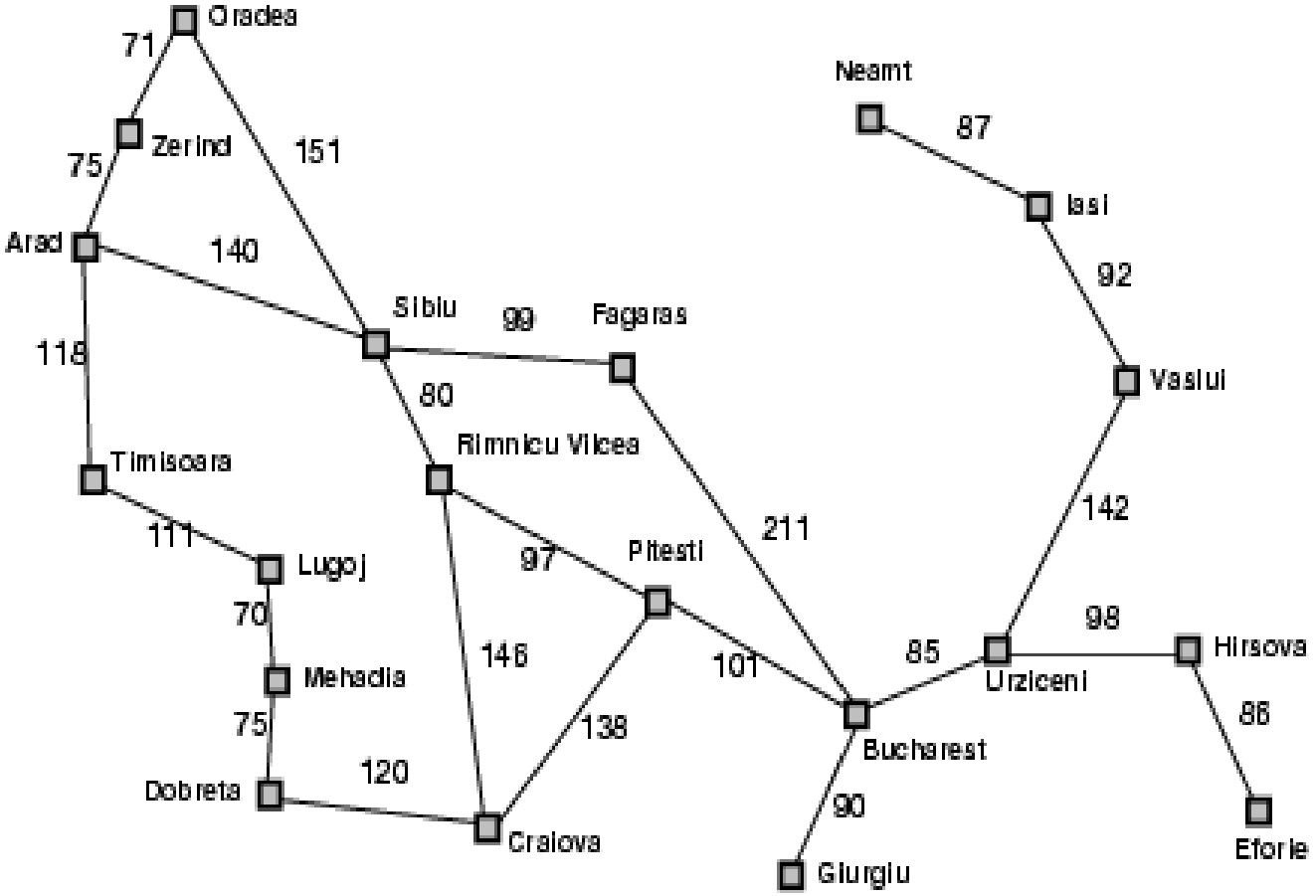
Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**Four (4)**	**Week No:**	**Four (4)**	**Semester:**	
Lecturer:	*Dr. Muhammad Firoz Mridha, Associate Professor, Dept of CS* *Email: firoz.mridha@aiub.edu*				

LECTURE OUTLINE

1. **BEST-FIRST SEARCH**
2. **GREEDY BEST-FIRST SEARCH**
3. **A* SEARCH**
4. **CONDITIONS FOR OPTIMALITY**
5. **OPTIMALITY OF A***

Romania with step costs in km



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

INFORMED SEARCH STRATEGY

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.

problem-specific knowledge

is the extra bit of information the program uses rather than the problem formulation, thus known as **Informed Search**

BEST-FIRST SEARCH

- The general approach to informed search 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.

HEURISTIC SEARCH COMPARED WITH BLIND SEARCH

Brute force / Blind search

- Only have knowledge about already explored nodes
- No knowledge about how far a node is from goal state

Heuristic search

- Estimates "distance" to goal state
- Guides search process toward goal state
- Prefer states (nodes) that lead close to and not away from goal state

HEURISTIC FUNCTION

- The choice of **f(evaluation function)** determines the search strategy
- 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.
- h(n) takes a *node* as input, but, unlike g(n), it depends only on the *state* at that node.
- Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm
- 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

- **Greedy best-first search**[8] 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).***
- "greedy"—at each step it tries to get as close to the goal as it can.
- Its search cost is minimal. It is not optimal.
- Greedy best-first tree search is also incomplete even in a finite state space, much like depth-first search.

STRAIGHT-LINE DISTANCE HEURISTIC

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.

GREEDY BEST-FIRST TREE SEARCH

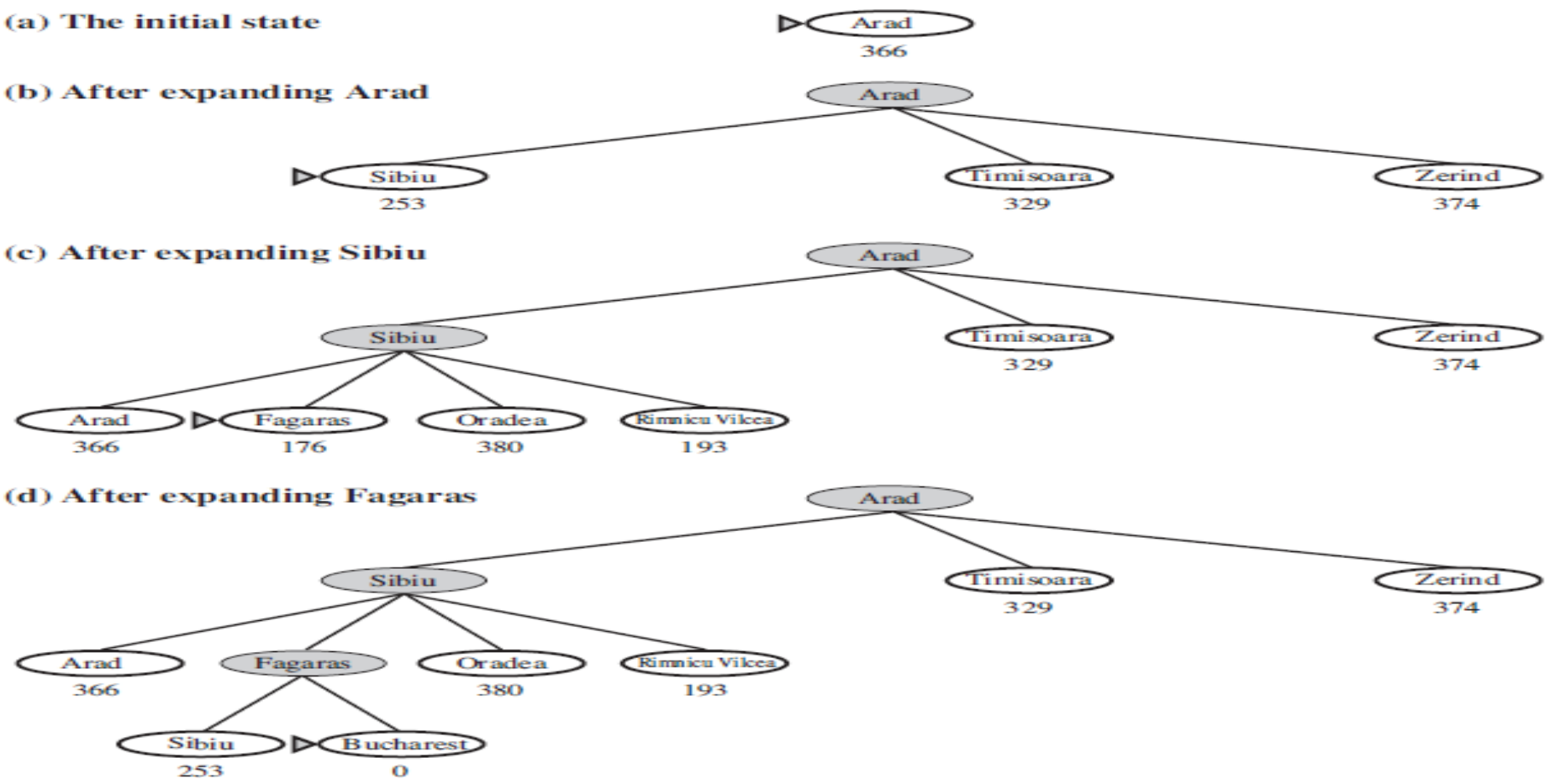


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

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) .

Since

g(n) gives the path cost from the start node to node n, and

h(n) is the estimated cost of the cheapest path from n to the goal,

we have f(n) = estimated cost of the cheapest solution through n .

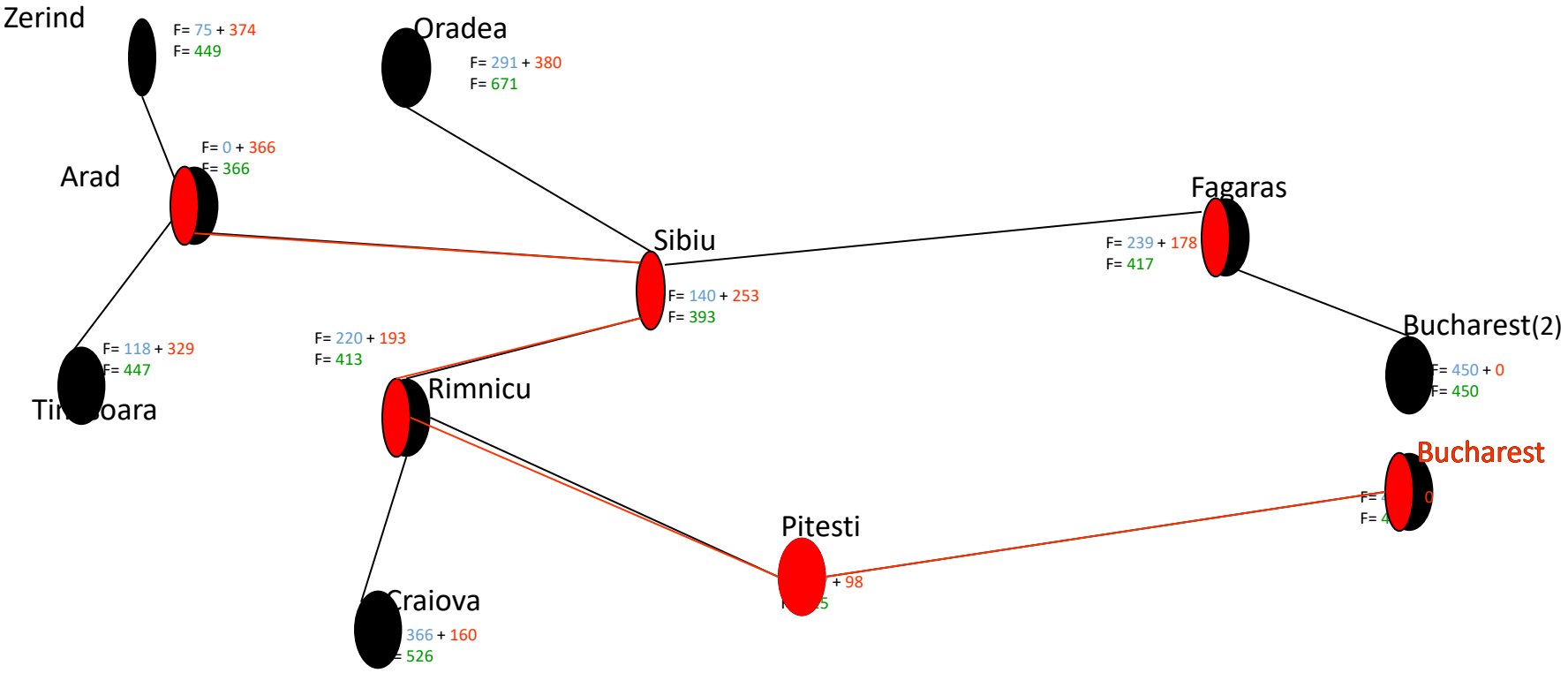
A∗ search is both complete and optimal.

The algorithm is identical to UNIFORM-COST-SEARCH except that

A∗ uses **g + h** instead of **g**.

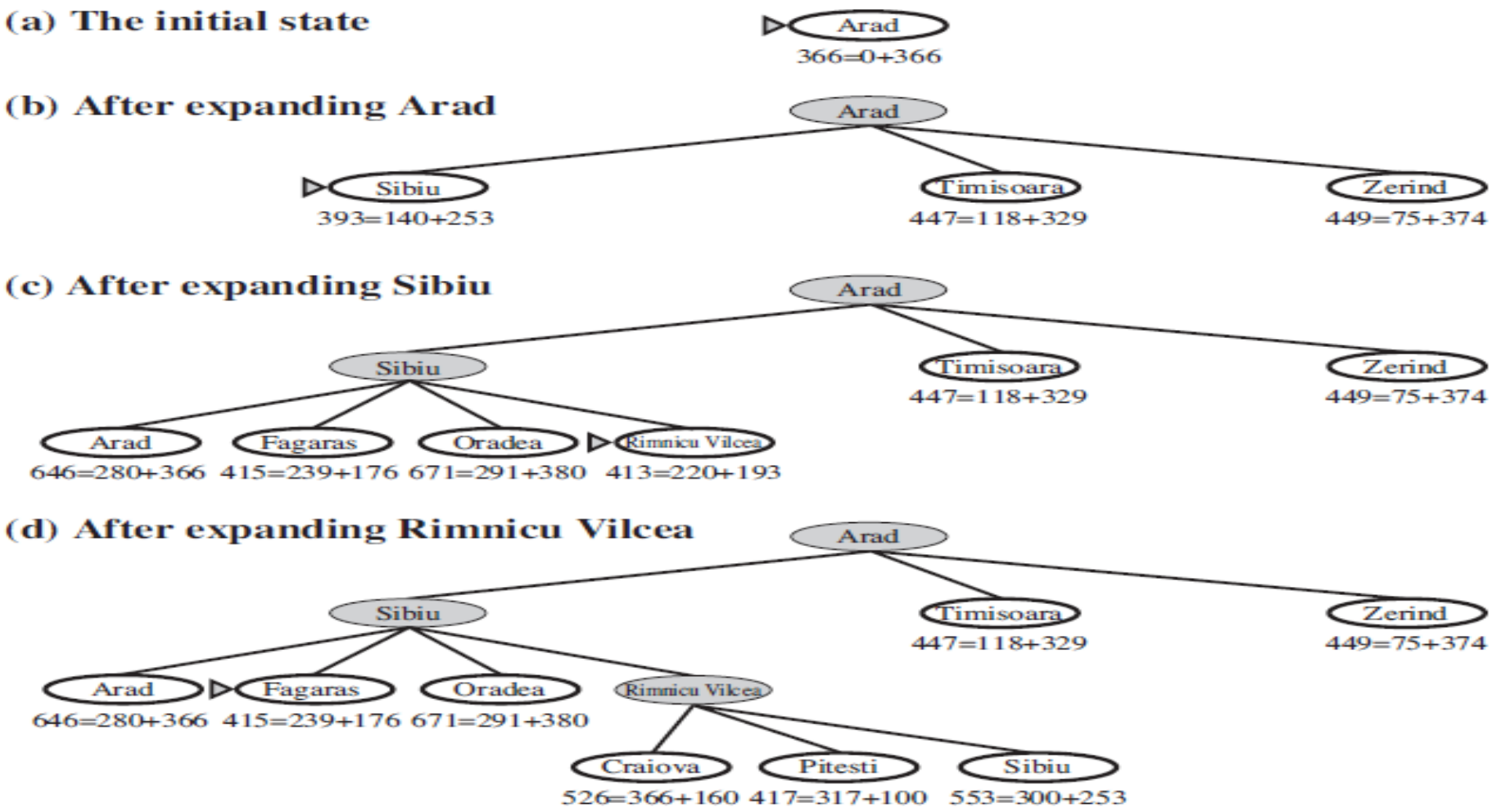
We have now arrived at Bucharest. As this is the lowest cost node AND the goal state we can terminate the search. If you look back over the slides you will see that the solution returned by the A* search pattern (Arad – Sibiu – Rimnicu – Pitesti – Bucharest), is in fact the optimal solution. Press space to continue with the slideshow.

We now expand Fagaras (that is, we expand the node with the lowest value of ***f***). Press space to continue the search.



Once Fagaras is expanded we look for the lowest cost node. As you can see, we now have **two** Bucharest nodes. One of these nodes (Arad – Sibiu – Rimnicu – Pitesti – Bucharest) has an ***f*** value of 418. The other node (Arad – Sibiu – Fagaras – Bucharest(2)) has an ***f*** value of 450. We therefore move to the first Bucharest node and expand it. Press space to continue

A* SEARCH FOR BUCHAREST



A* SEARCH FOR BUCHAREST

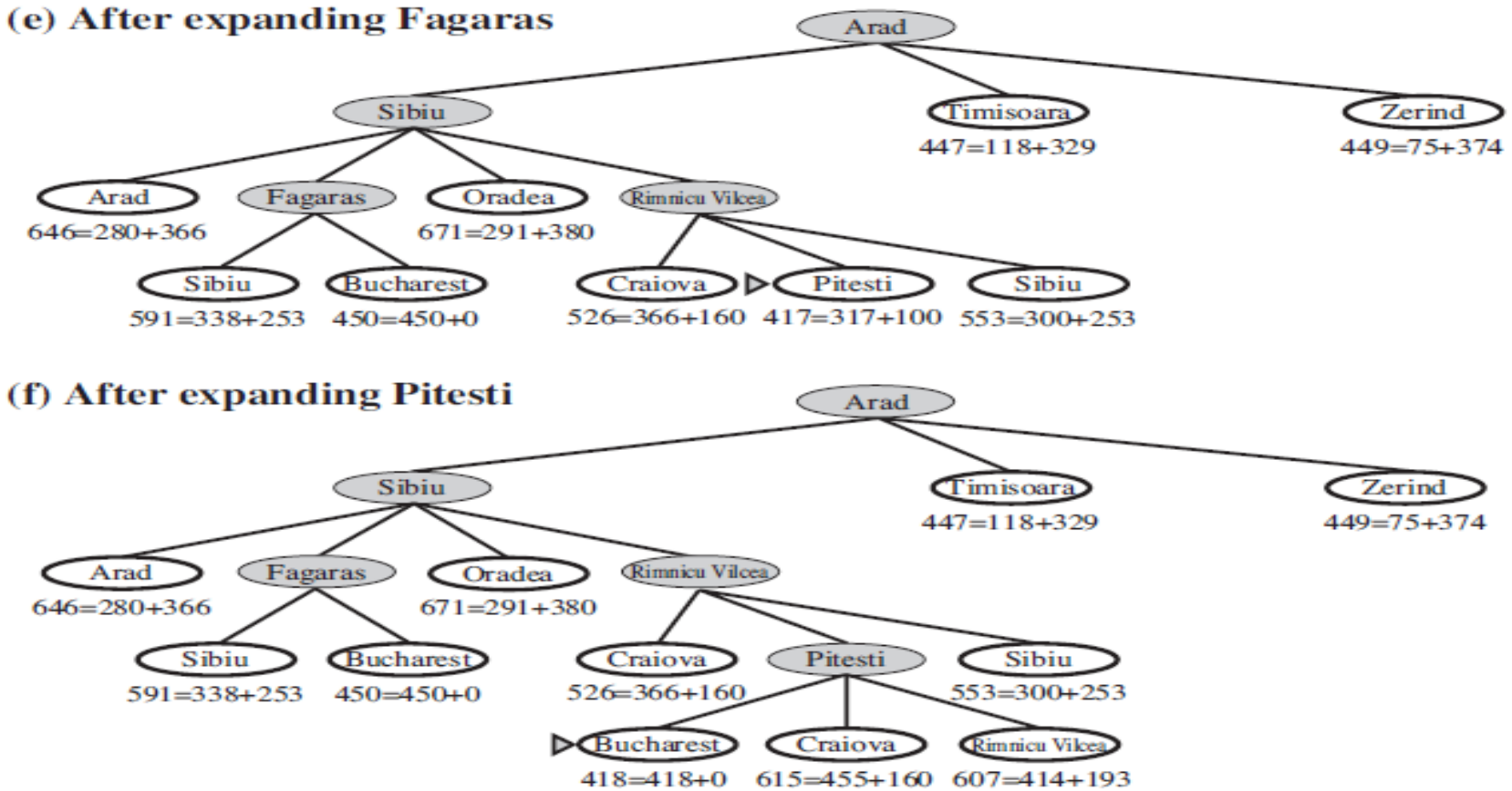


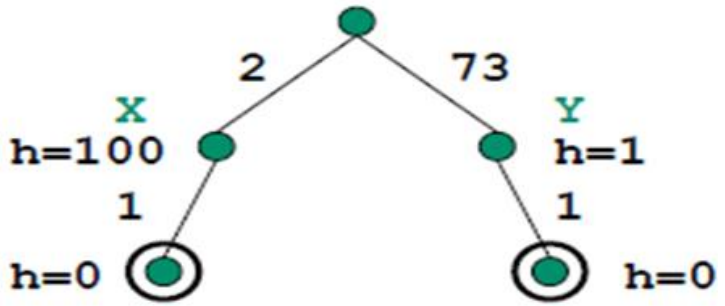
Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

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.

ADMISSIBILITY

- What must be true about h for A* to find optimal path?
- A* finds optimal path if h is admissible; h is admissible when it never overestimates.
- In this example, h is not admissible.
- In route finding problems, straight-line distance to goal is admissible heuristic.



g(X)+h(X) = 102

g(Y)+h(Y) = 74

Optimal path is not found!

CONSISTENCY

- 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) ≤ c(n, a, n') + h(n').

- This is a form of the general **triangle inequality**, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides

OPTIMALITY OF A*

A∗ has the following properties: *the tree-search version of* A* *is optimal if* h(n) *is admissible, while the graph-search version is optimal if* h(n) *is consistent.*

if h(n) *is consistent, then the values of* f(n) *along any path are nondecreasing.*
The proof follows directly from the definition of consistency. Suppose n is a successor of n; then g(n')=g(n) + c(n, a, n') for some action a, and we have
f(n) = g(n) + h(n) = g(n) + c(n, a, n) + h(n) ≥ g(n) + h(n) = f(n) .

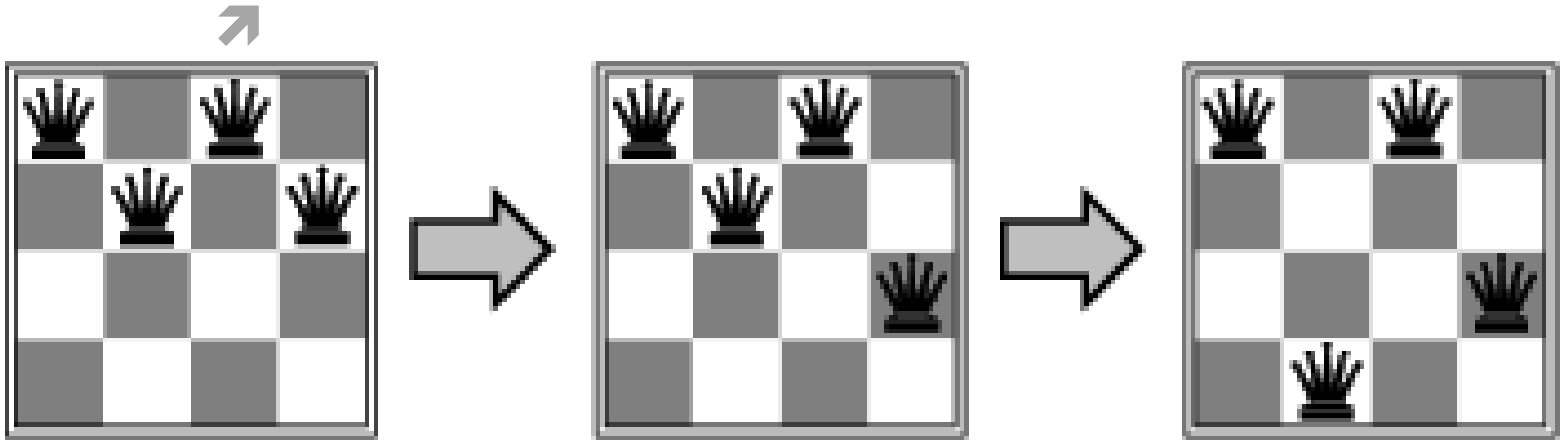
The next step is to prove that *whenever* A* *selects a node* n *for expansion, the optimal path to that node has been found.* Were this not the case, there would have to be another frontier node n on the optimal path from the start node to n. because f is nondecreasing along any path, n would have lower f-cost than n and would have been selected first.

Local search algorithms

- The search algorithms that we have seen so far are designed to explore search space systematically
- Systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path and which have not
- When a goal is found, the path to that goal also constitute a solution to the problem
- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it

Example: *n*-queens

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal



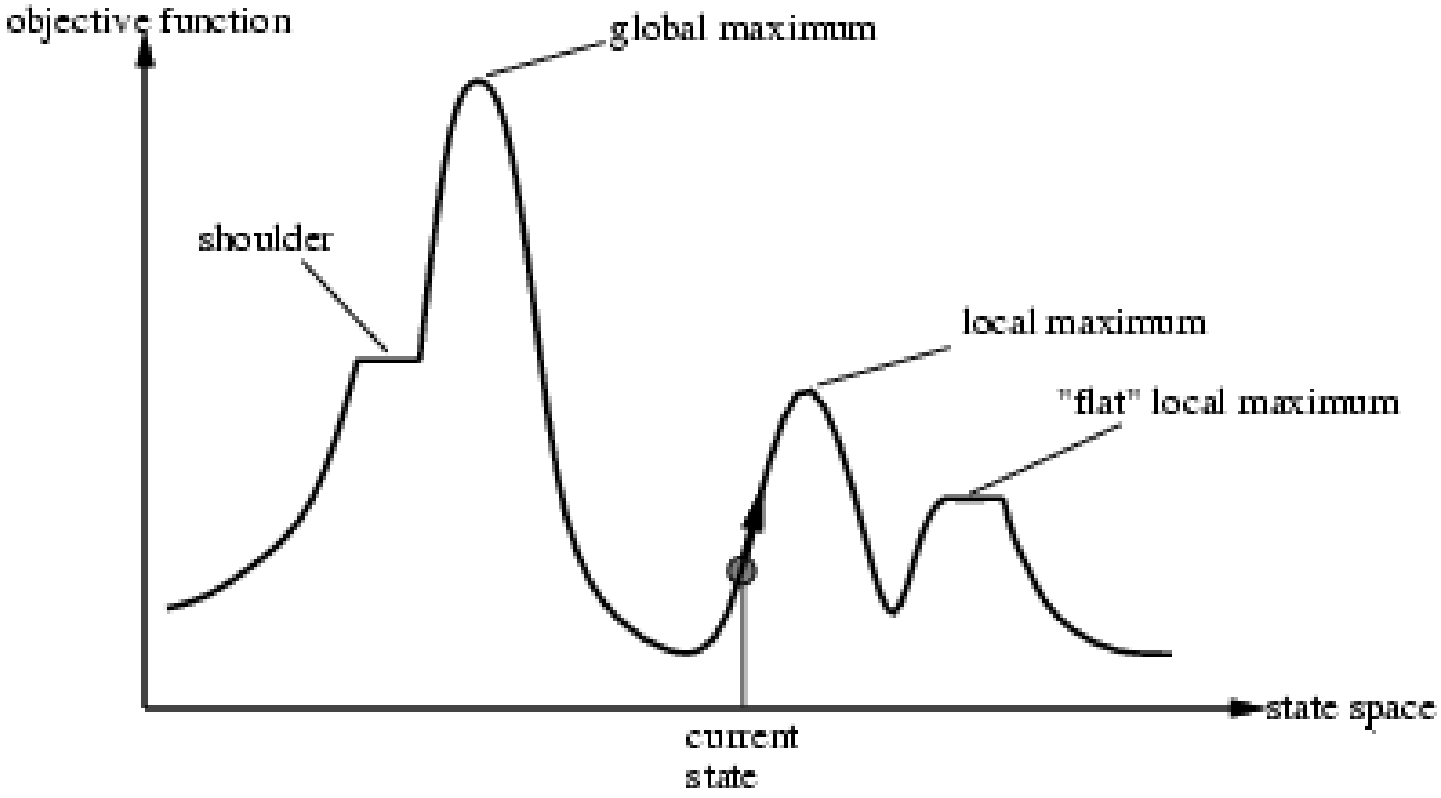
Hill-climbing search

- "Like climbing Everest in thick fog with amnesia"

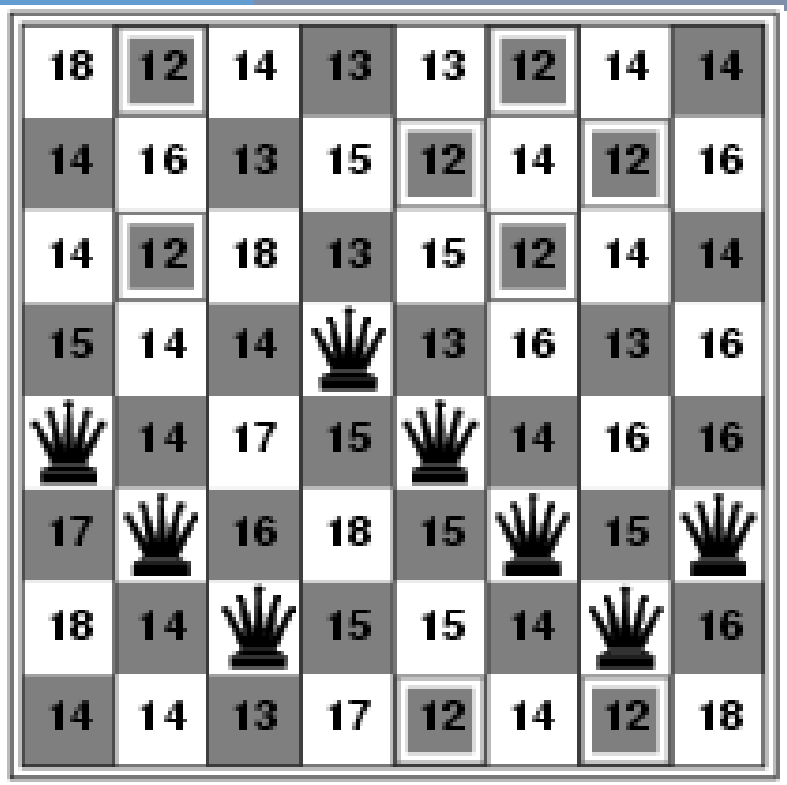
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima

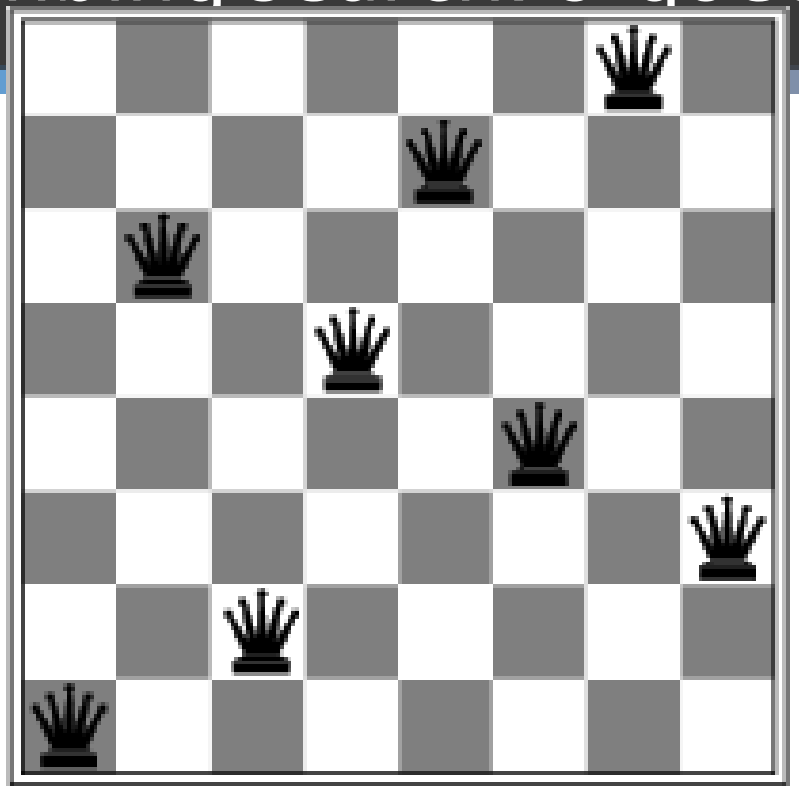


Hill-climbing search: 8-queens problem



- *h* = number of pairs of queens that are attacking each other, either directly or indirectly
- *h = 17* for the above state
-

Hill-climbing search: 8-queens problem



- A local minimum with *h = 1*
-

AI - HEURISTIC SEARCH

- **Heuristic search** refers to a **search** strategy that attempts to optimize a problem by iteratively improving the solution based on a given **heuristic** function or a cost measure.

- A Heuristic is a technique to solve a problem faster than classic methods, or to find an approximate solution when classic methods cannot.

- This is a kind of a shortcut as we often trade one of optimality, completeness, accuracy, or precision for speed.

Fig 4.1 First three levels of the tic-tac-toe state space reduced by symmetry

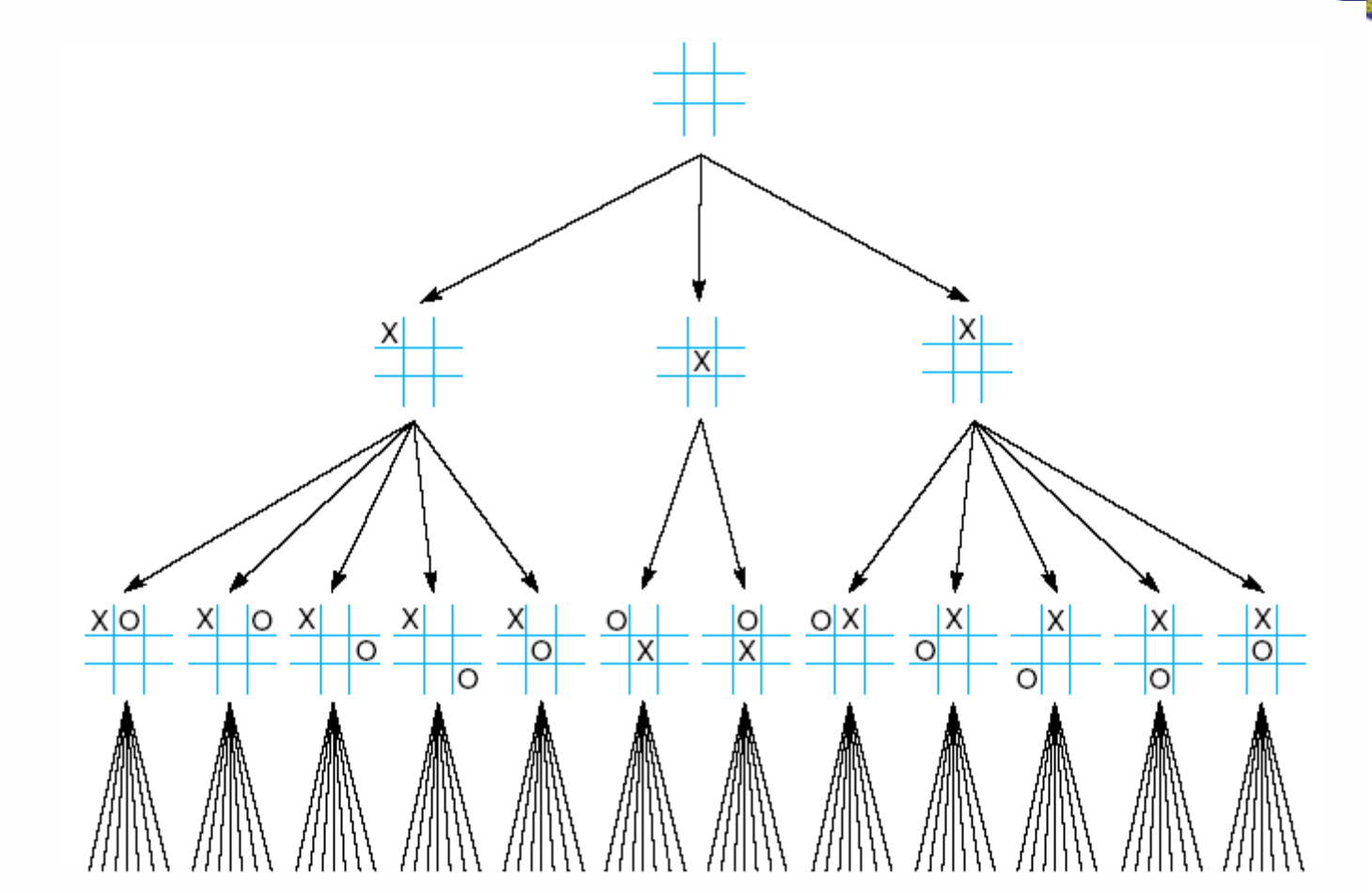


Fig 4.2 The "most wins" heuristic applied to the first children in tic-tac-toe.

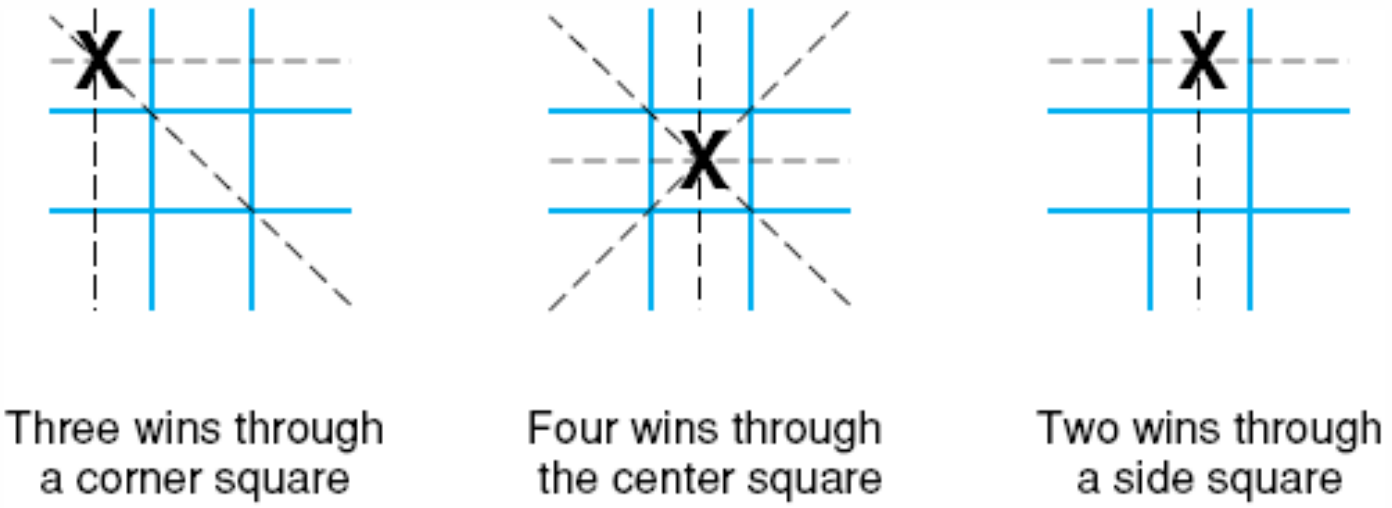


Fig 4.3 Heuristically reduced state space for tic-tac-toe.

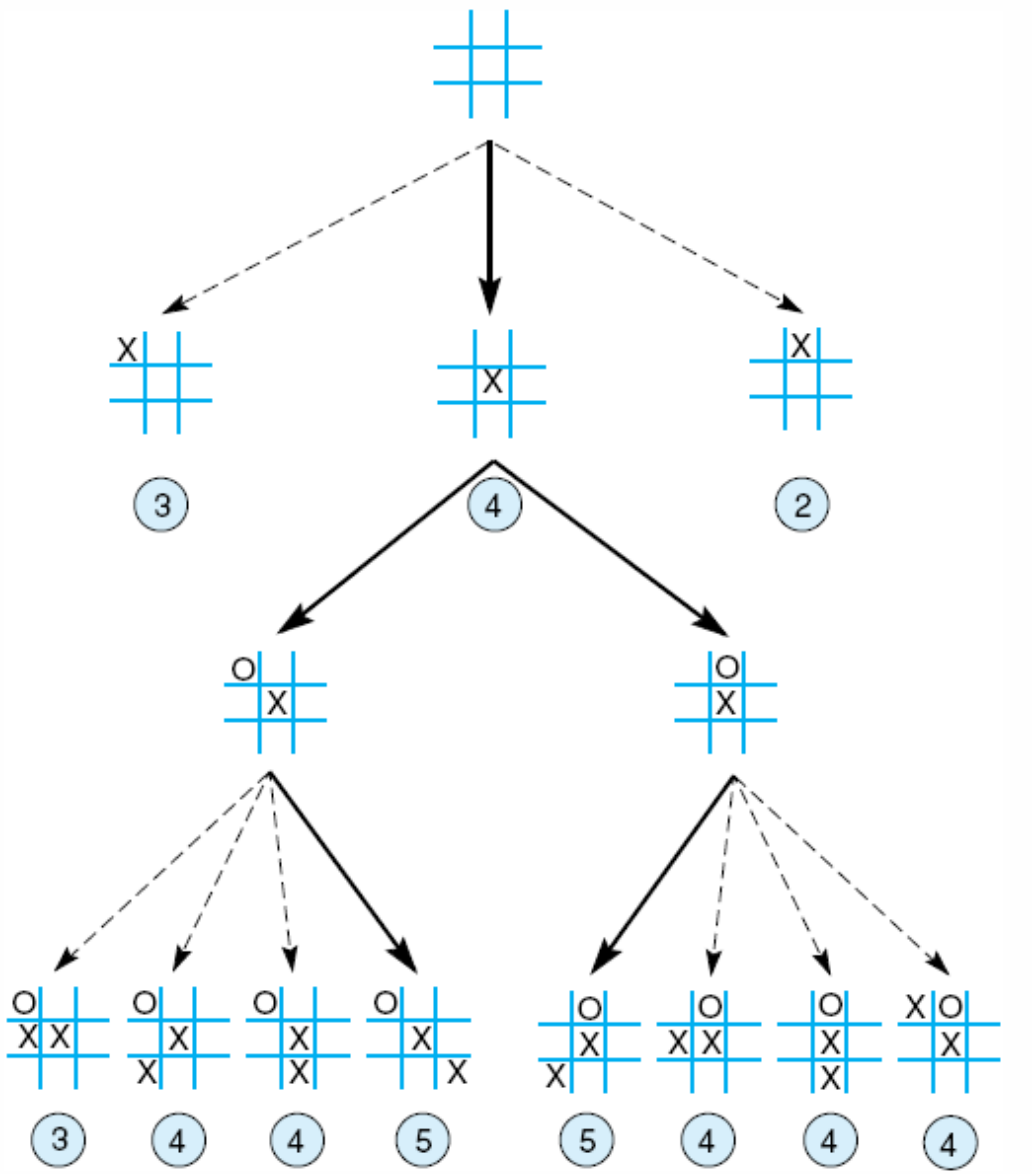


Fig 4.12 The start state, first moves, and goal state for an example-8 puzzle.

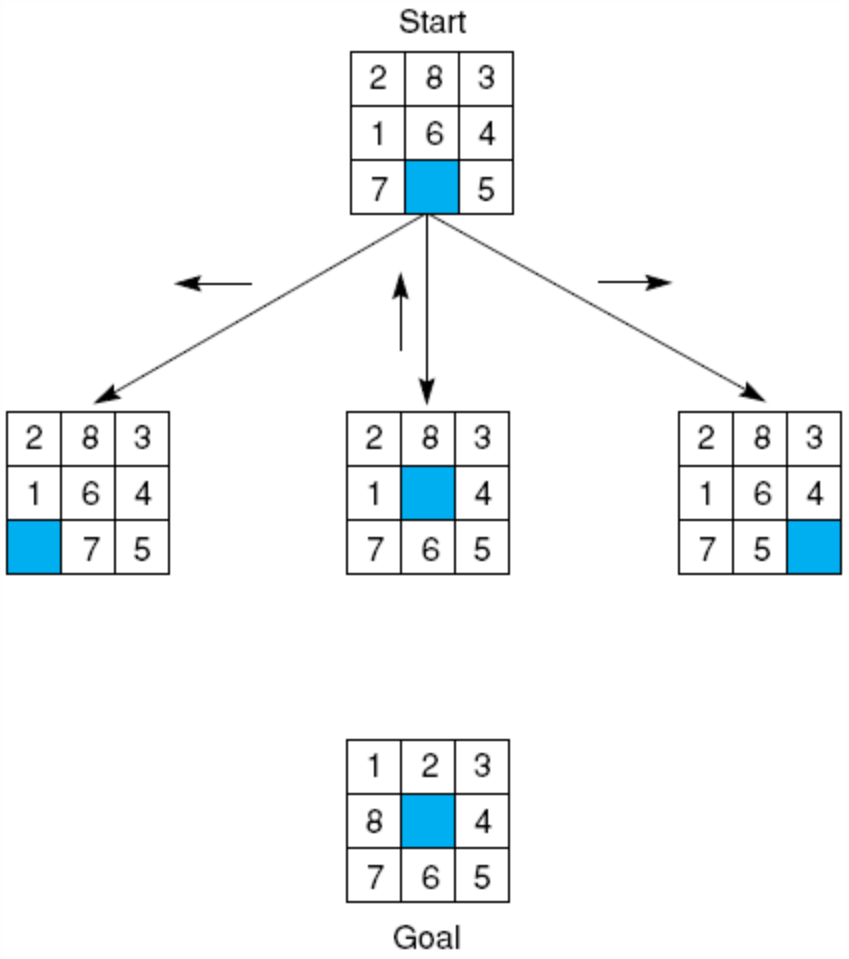


Fig 4.14 Three heuristics applied to states in the 8-puzzle.

2 8 3 1 6 4 _ 7 5	5	6	0
2 8 3 1 _ 4 7 6 5	3	4	0
2 8 3 1 6 4 7 5 _	5	6	0
	Tiles out of place	Sum of distances out of place	2 x the number of direct tile reversals

1 2 3
8 _ 4
7 6 5

Goal

Fig 4.15 The heuristic f applied to states in the 8-puzzle.

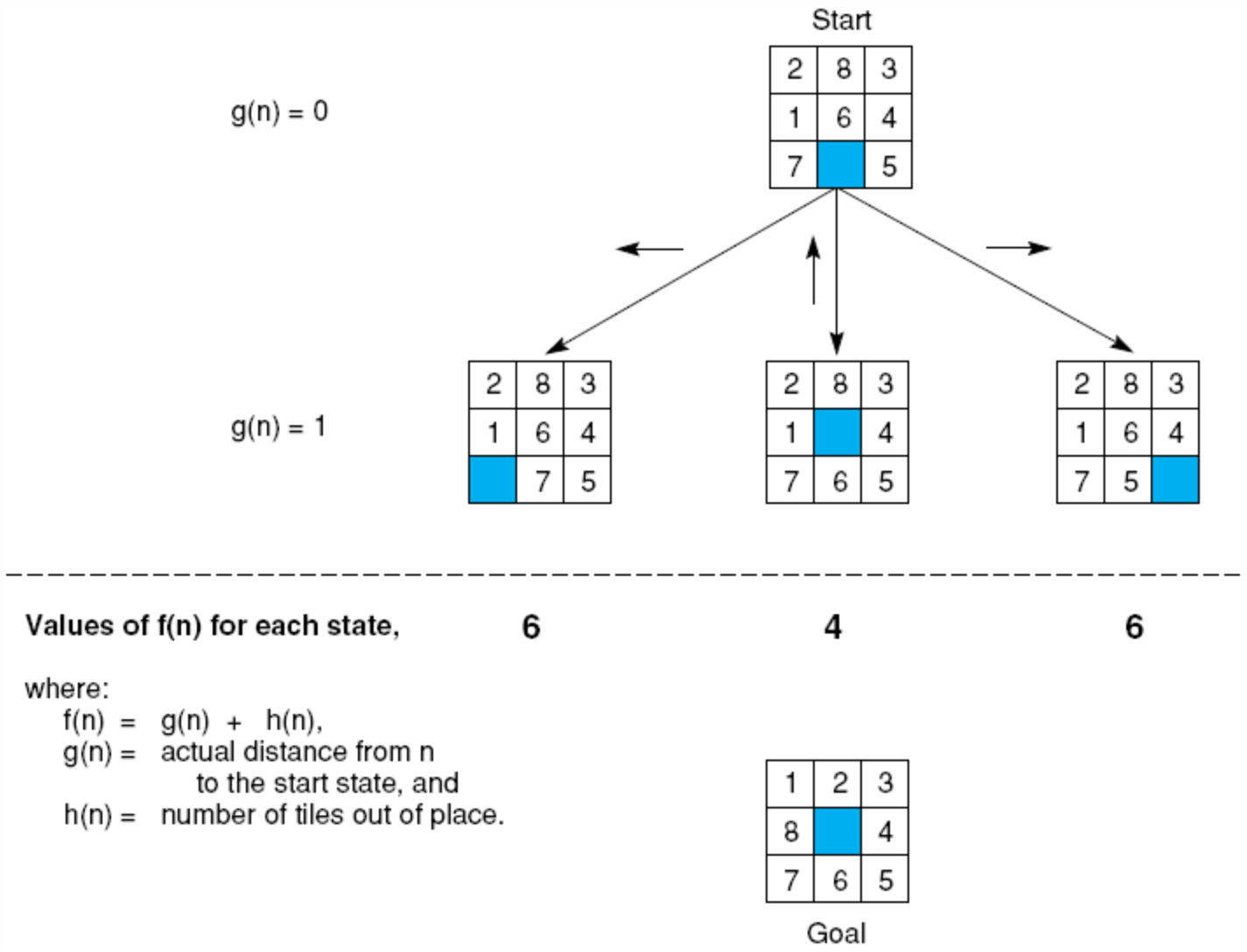


Fig 4.16 State space generated in heuristic search of the 8-puzzle graph.

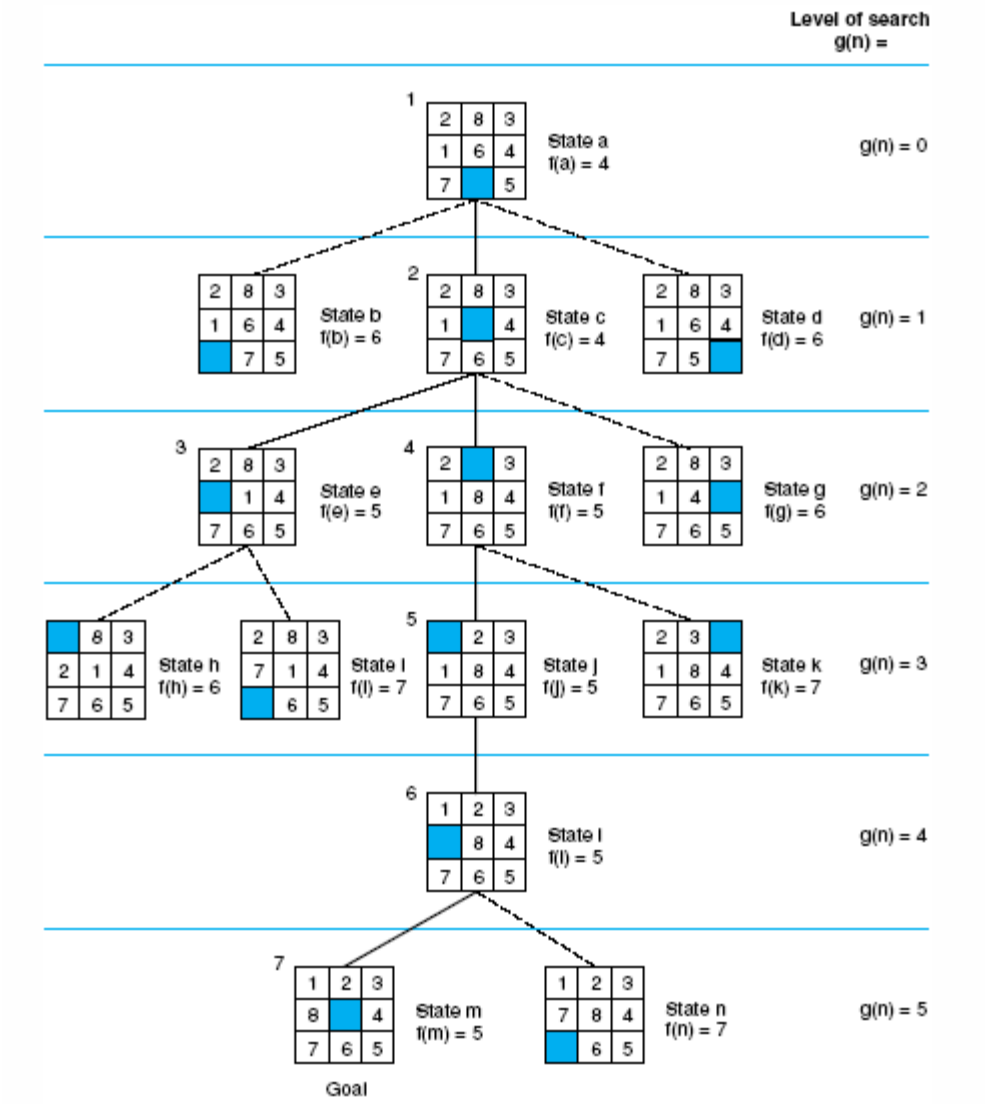
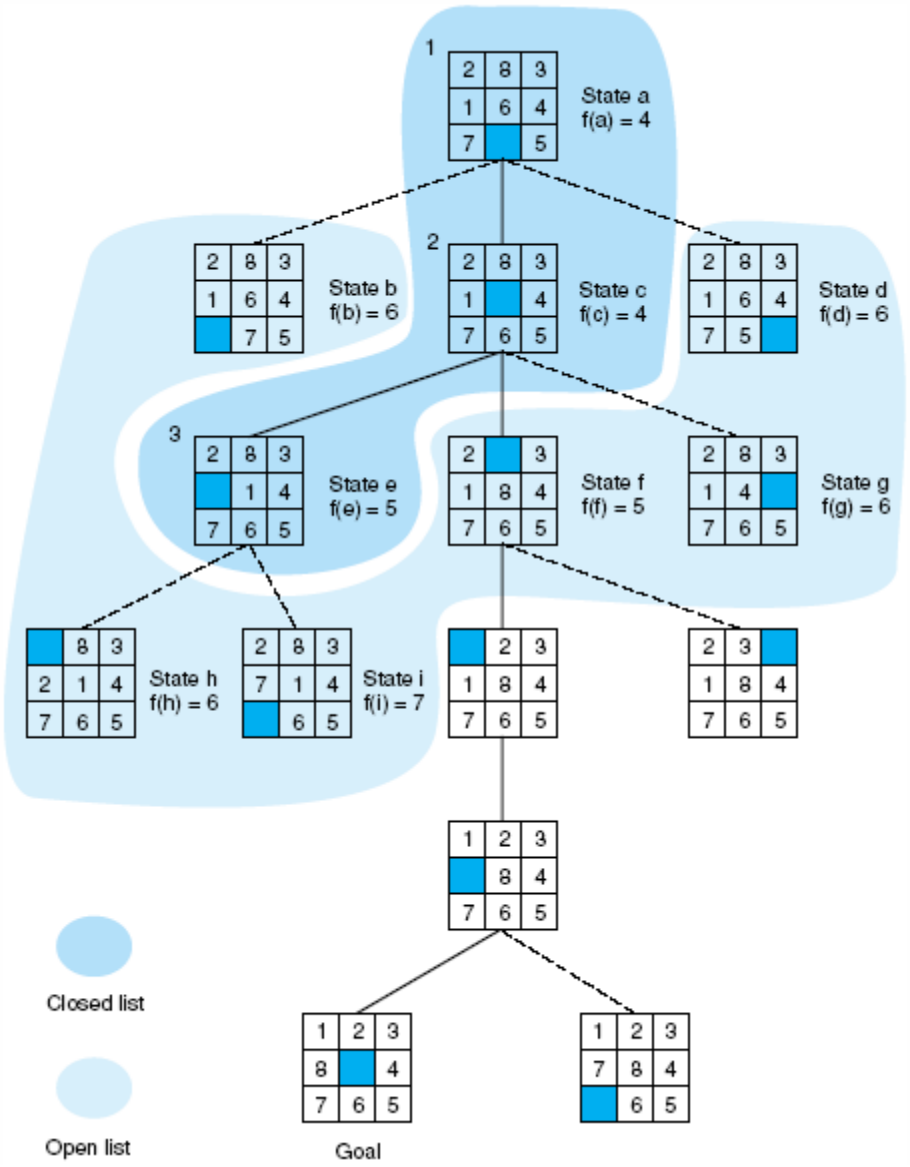


Fig 4.17 Open and closed as they appear after the 3rd iteration of heuristic search



DEFINITION

ALGORITHM A, ADMISSIBILITY, ALGORITHM A*

Consider the evaluation function f(n) = g(n) + h(n), where

n is any state encountered in the search.

g(n) is the cost of n from the start state.

h(n) is the heuristic estimate of the cost of going from n to a goal.

If this evaluation function is used with the best_first_search algorithm of Section 4.1, the result is called *algorithm A*.

A search algorithm is *admissible* if, for any graph, it always terminates in the optimal solution path whenever a path from the start to a goal state exists.

If algorithm A is used with an evaluation function in which h(n) is less than or equal to the cost of the minimal path from n to the goal, the resulting search algorithm is called *algorithm* A* (pronounced "A STAR").

It is now possible to state a property of A* algorithms:

All A* algorithms are admissible.

DEFINITION

MONOTONICITY

A heuristic function h is monotone if

1. For all states n_i and n_j, where n_j is a descendant of n_i,

   $h(n_i) - h(n_j) \leq cost(n_i,n_j)$,

   where $cost(n_i,n_j)$ is the actual cost (in number of moves) of going from state n_i to n_j.

2. The heuristic evaluation of the goal state is zero, or h(Goal) = 0.

DEFINITION

INFORMEDNESS

For two A* heuristics h_1 and h_2, if $h_1(n) \leq h_2(n)$, for all states n in the search space, heuristic h_2 is said to be *more informed* than h_1.

Fig 4.18 Comparison of state space searched using heuristic search with space searched by breadth-first search. The proportion of the graph searched heuristically is shaded. The optimal search selection is in bold. Heuristic used is f(n) = g(n) + h(n) where h(n) is tiles out of place.

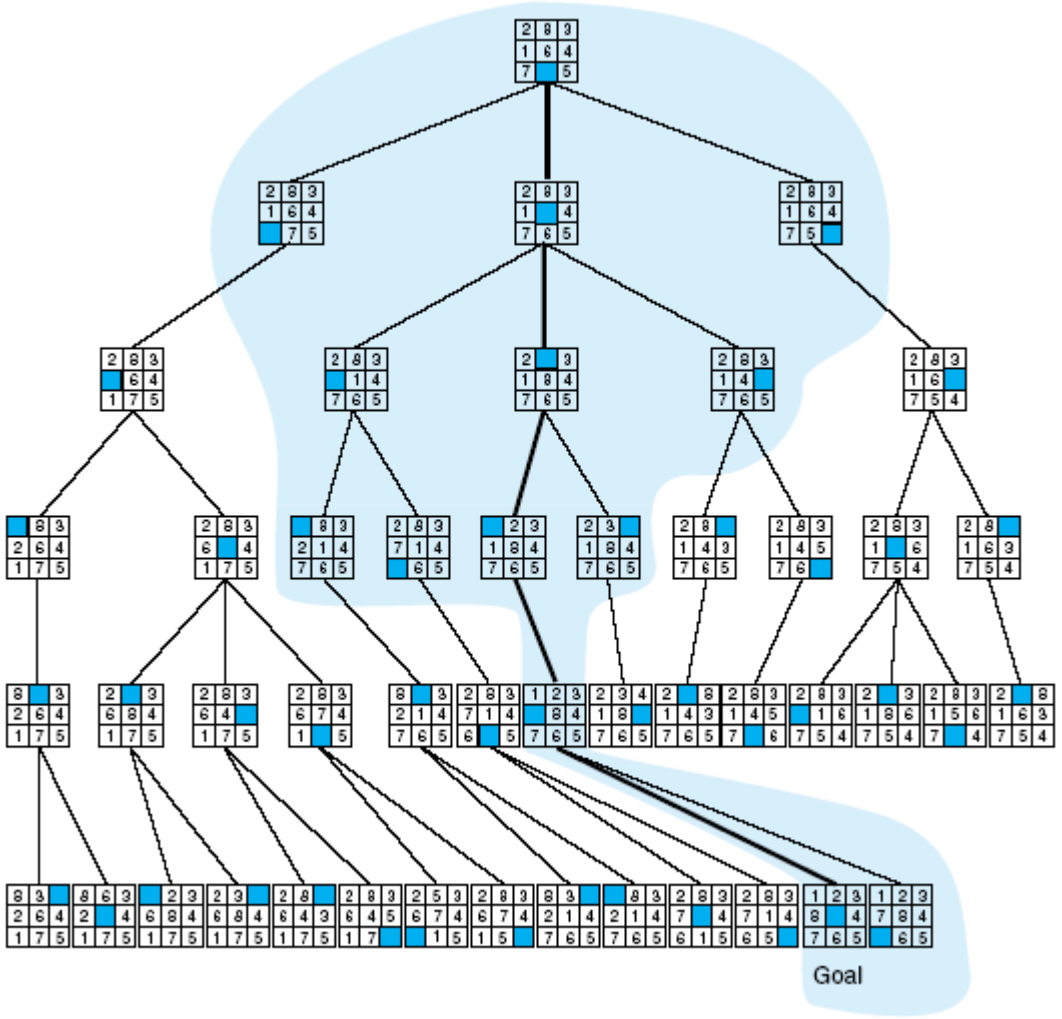


Fig 4.19 State space for a variant of nim. Each state partitions the seven matches into one or more piles.

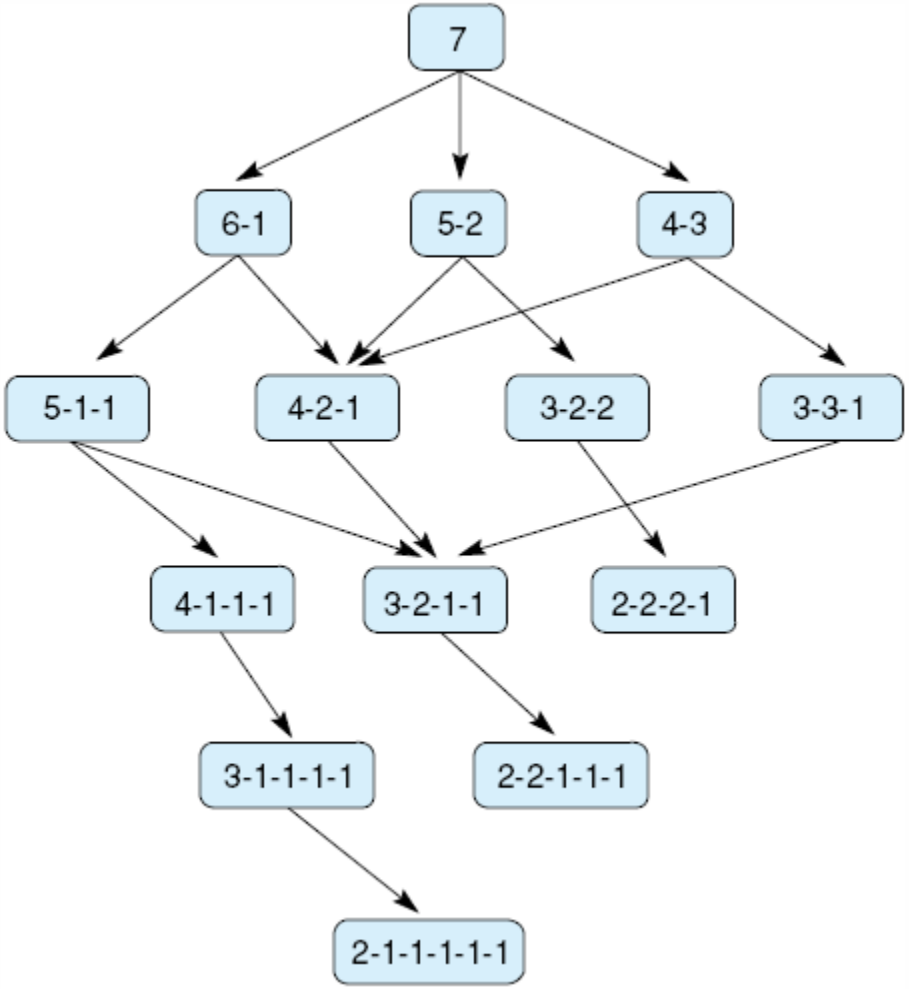
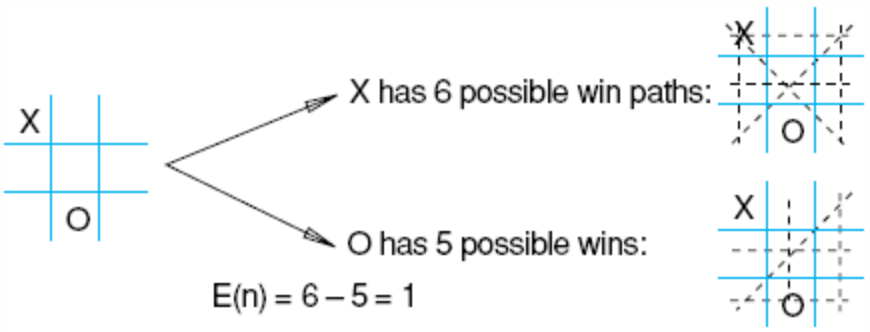
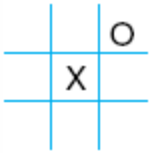


Fig 4.22 Heuristic measuring conflict applied to states of tic-tac-toe.





Heuristic is E(n) = M(n) – O(n)

where M(n) is the total of My possible winning lines

O(n) is total of Opponent's possible winning lines

E(n) is the total Evaluation for state n

Fig 4.23 Two-ply minimax applied to the opening move of tic-tac-toe, from Nilsson (1971).

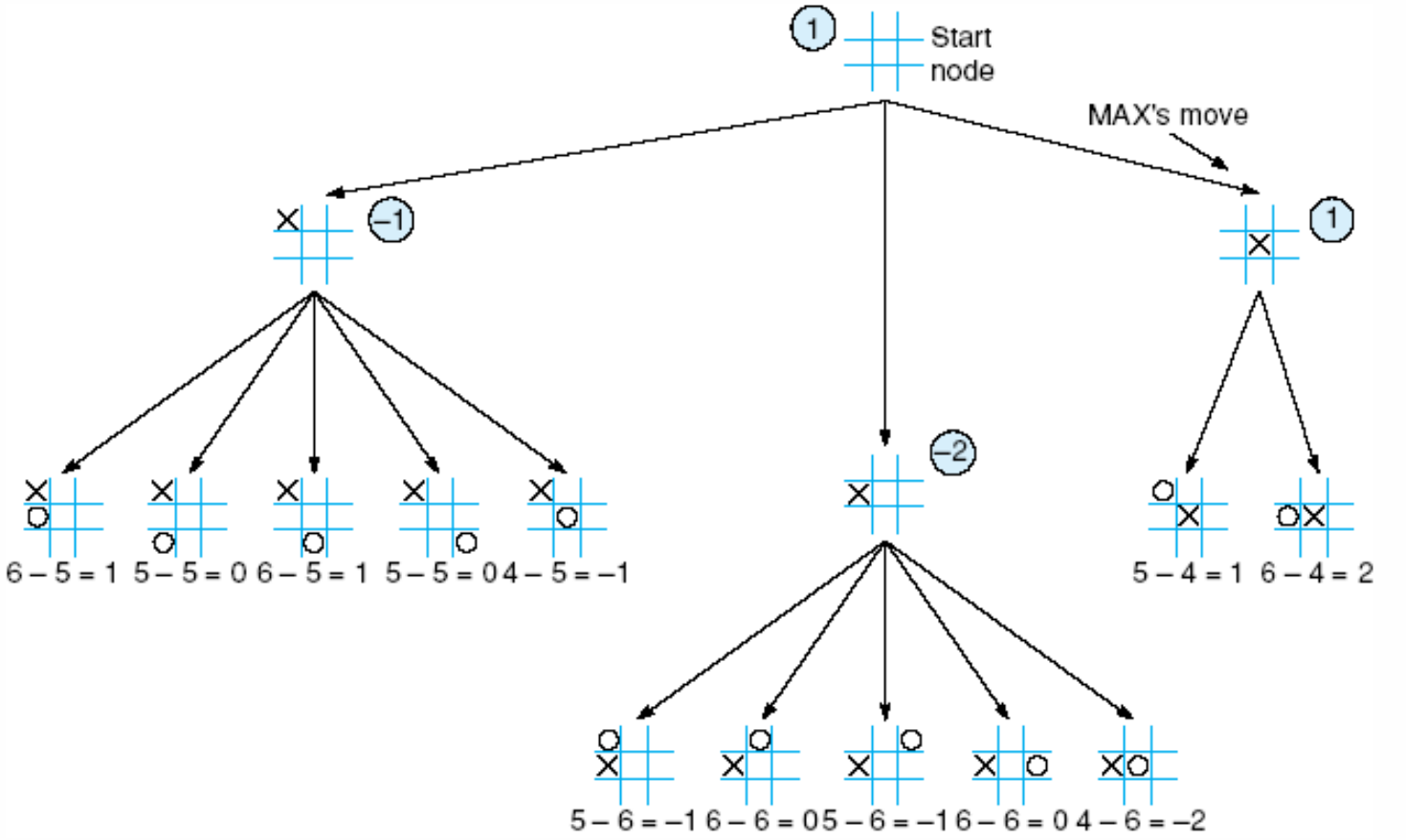


Fig 4.24 Two ply minimax, and one of two possible MAX second moves, from Nilsson (1971).

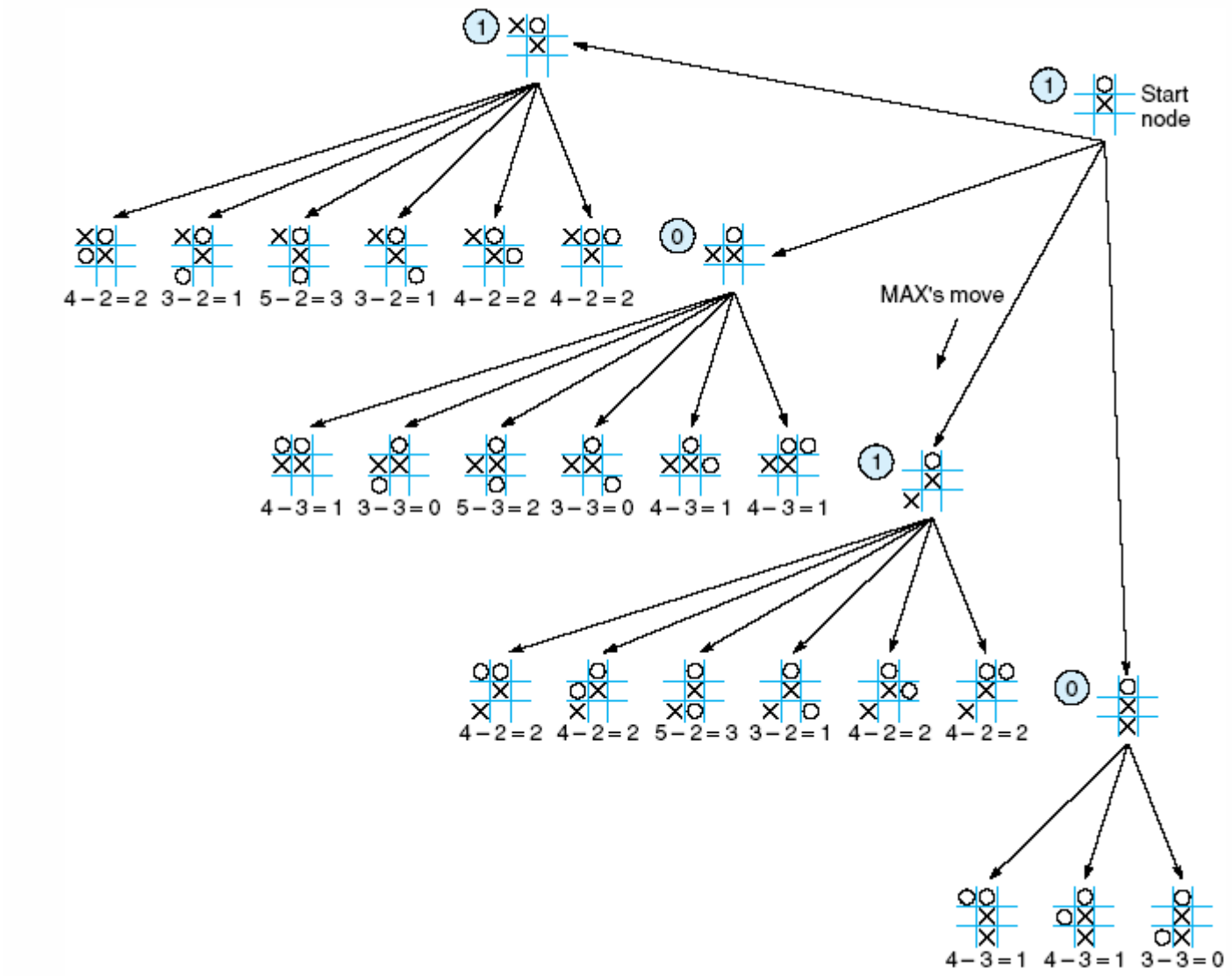


Fig 4.25 Two-ply minimax applied to X's move near the end of the game, from Nilsson (1971).