Artificial Intelligence

MD. SHAHARIAR SARKAR
Assistant Professor,
Department of Computer Science and Engineering
Military Institute of Science and Technology (MIST)

What is intelligence?

Intelligence is the

- Ability to perceive and act in the world.
- Reasoning: proving theorems, medical diagnosis
- Planning: take decisions
- Learning and Adaptation: recommend movies, learn traffic patterns
- Understanding: text, speech, visual scene

What is artificial intelligence?

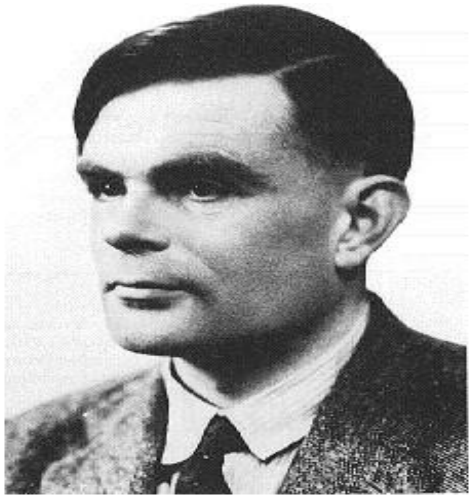
Thought *vs.* Behavior	Human *vs.*	Rational
Thought	"[Automation of] activities that we associate with human thinking and activities such as decision making, problem solving, learning" (Bellman 1978)	"The study of mental faculties through the use of computational models" (Charniak & McDertmott 1985)
Behavior	"The study of how to make computers to things at which, at the moment, people are better" (Rich & Knight 1991)	"The branch of computer science that is concerned with the automation of intelligent behavior" (Luger & Stubblefield 1993)

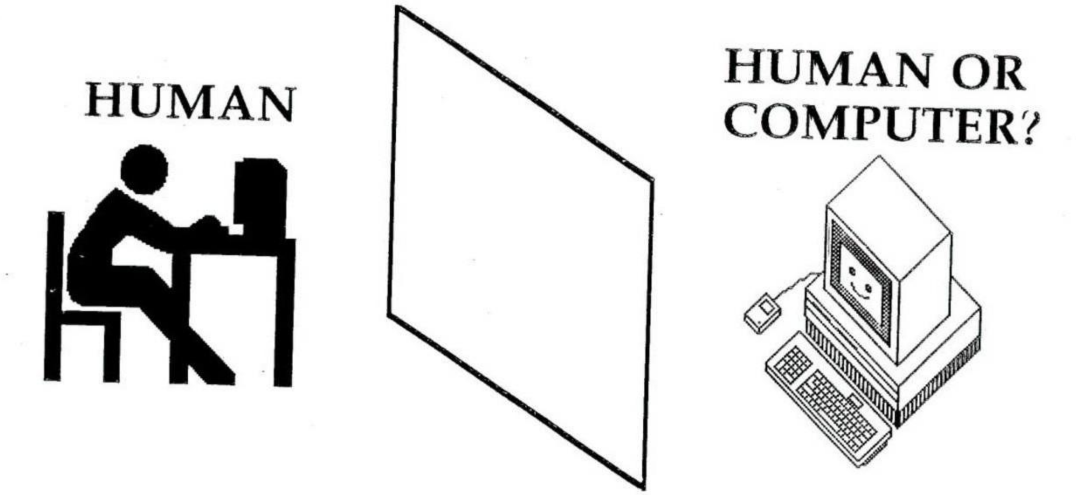
What is artificial intelligence?

	Humanly *vs.*	Rationally
Thought *vs.*	Systems that think like humans	Systems that think rationally
Behavior	Systems that act like humans	Systems that act rationally

Acting Humanly: Turing Test Approach

- The Turing test, proposed by Alan Turing in 1950.
- If the human cannot tell whether the responses from the other side of a wall are coming from a human or computer, then the computer is intelligent.





Thinking humanly: The cognitive modeling approach

To say that a program thinks like a human, we must know how humans think.

- ❑ We can learn about human thought in three ways:
  - 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.

Thinking humanly: The cognitive modeling approach

- 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.

Thinking rationally: The “laws of thought” approach

- Aristotle was one of the first to attempt to codify “right thinking”
- Example: Socrates is a man and all men are mortal.
- So, conclusion: Socrates is mortal.
- This study initiated the field called logic.
- We simply don’t know the rules of, say, politics or warfare in the same way that we know the rules of chess or arithmetic.

Thinking rationally: The “laws of thought” approach

- The theory of probability fills this gap, allowing rigorous reasoning with uncertain information.
- In Probability principle, it allows the construction of a comprehensive model of rational thought, leading from raw perceptual information to an understanding of how the world works to predictions about the future.

Acting rationally: The rational agent approach

- An agent is just something that are expected to operate autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals.
- A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

The Foundations of Artificial Intelligence

- Philosophy
- Mathematics
- Economics
- Neuroscience
- Psychology

The State of the Art

Stanford University’s One Hundred Year Study on AI (also known as AI100) report concludes that

- “Substantial increases in the future uses of AI applications, including more self-driving cars, healthcare diagnostics and targeted treatment, and physical assistance for elder care can be expected”

and that

- “Society is now at a crucial juncture in determining how to deploy AI-based technologies in ways that promote rather than hinder democratic values such as freedom, equality, and transparency.”

Risks of AI

- Lethal autonomous weapons
- Surveillance and persuasion
- Biased decision making
- Impact on employment

Benefits of AI

- Safety-critical applications
- Cybersecurity

Agents and Environments

Agent: An agent is anything that can perceive its environment through sensors and act upon that environment through actuators.

Example:

- 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 file contents, network packets, and human input (keyboard/mouse/touchscreen/voice) as sensory inputs and acts on the environment by writing files, sending network packets, and displaying information or generating sounds.

Agents and Environments

In general, an agent's choice of action at any given instant can depend on its built-in knowledge and on the entire percept sequence observed to date, but not on anything it hasn't perceived.

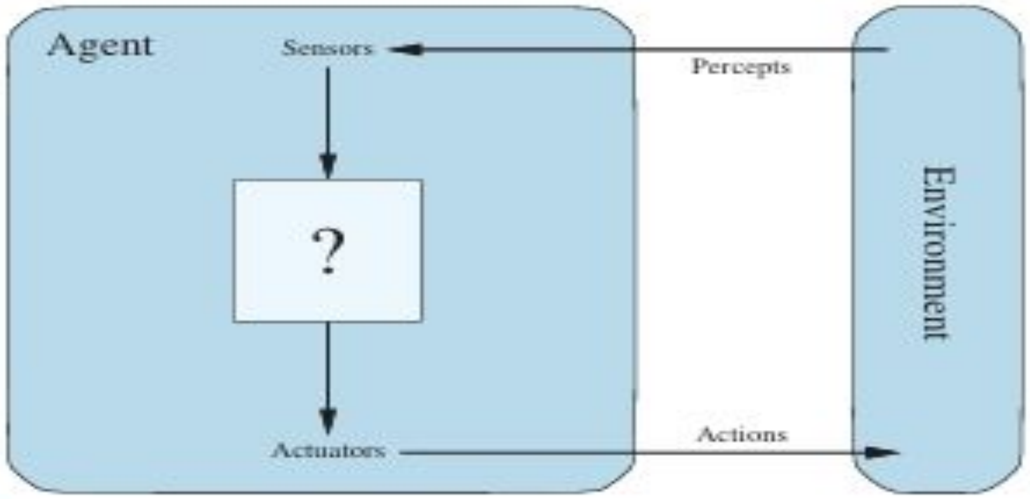


Figure 2.1 Agents interact with environments through sensors and actuators.

Rational Agent Example

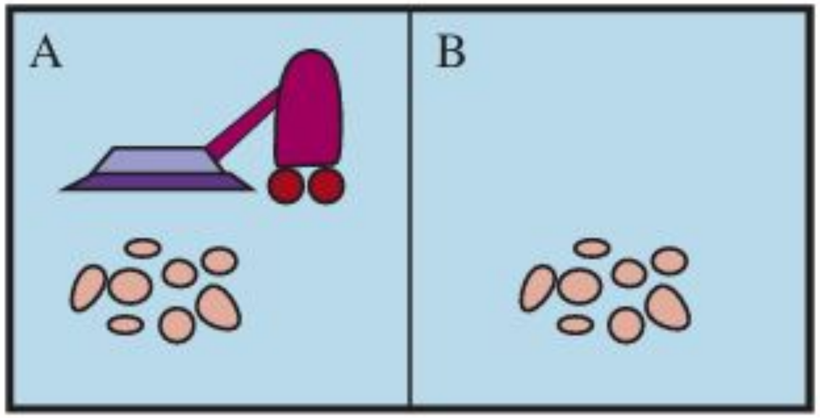
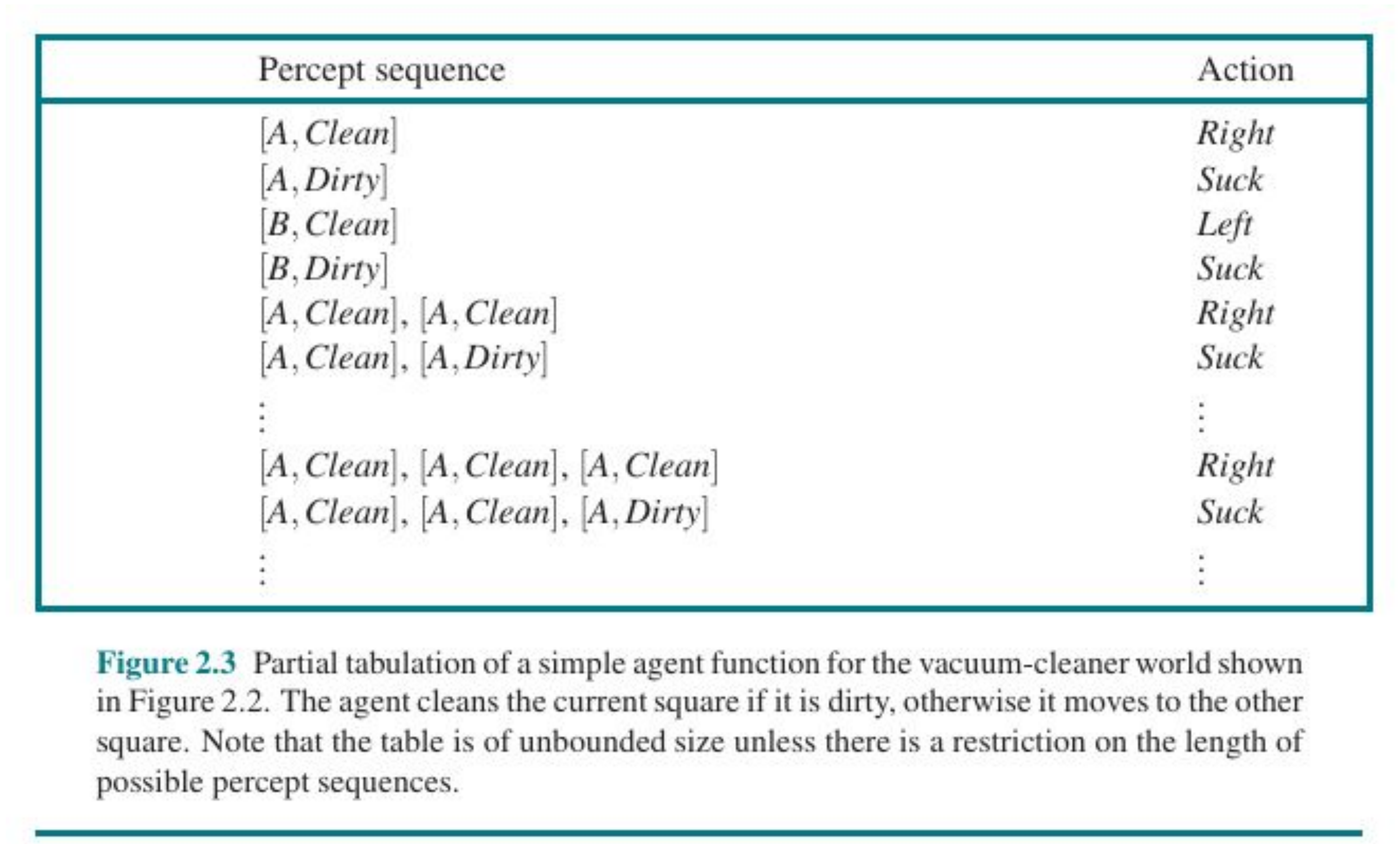


Figure 2.2 A vacuum-cleaner world with just two locations. Each location can be clean or dirty, and the agent can move left or right and can clean the square that it occupies. Different versions of the vacuum world allow for different rules about what the agent can perceive, whether its actions always succeed, and so on.

Partial Percept Sequence for Vacuum Cleaner



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. The agent cleans the current square if it is dirty, otherwise it moves to the other square. Note that the table is of unbounded size unless there is a restriction on the length of possible percept sequences.

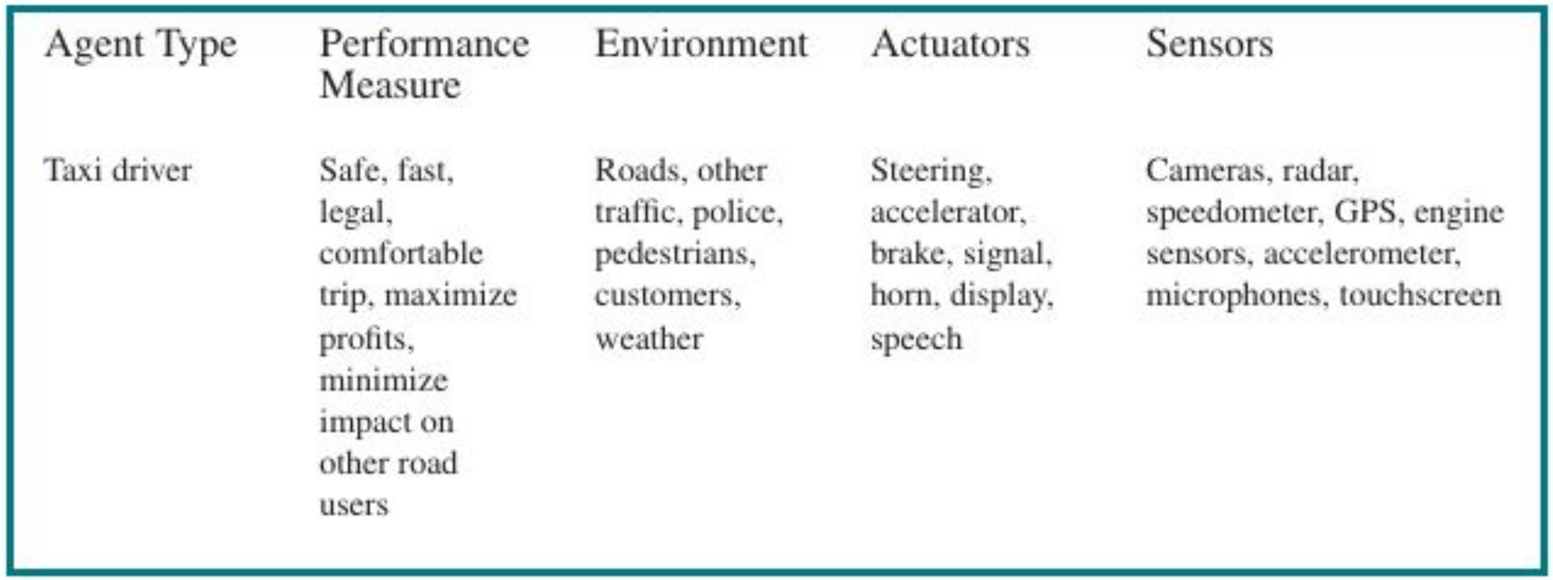
Task environment

In designing an agent, the first step must always be to specify the **task environment** as fully as possible.

The task environment can be defined by measuring or specifying **PEAS**

- Performance.
- Environment.
- Actuators.
- Sensors.

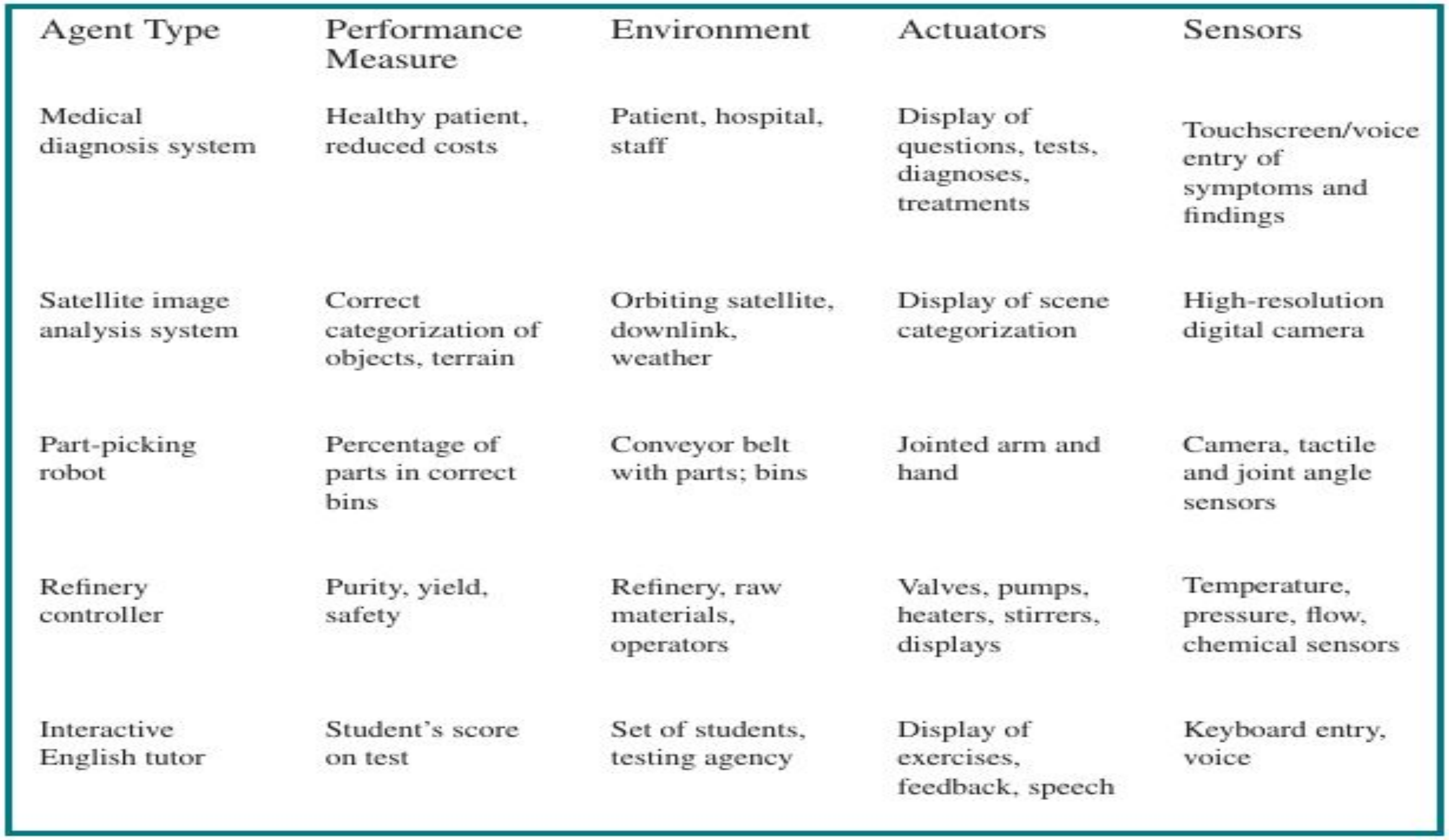
Specifying the task environment an automated taxi driver



Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits, minimize impact on other road users	Roads, other traffic, police, pedestrians, customers, weather	Steering, accelerator, brake, signal, horn, display, speech	Cameras, radar, speedometer, GPS, engine sensors, accelerometer, microphones, touchscreen

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

Example of Agent Types and PEAS description



Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments	Touchscreen/voice entry of symptoms and findings
Satellite image analysis system	Correct categorization of objects, terrain	Orbiting satellite, downlink, weather	Display of scene categorization	High-resolution digital camera
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, tactile and joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, raw materials, operators	Valves, pumps, heaters, stirrers, displays	Temperature, pressure, flow, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, feedback, speech	Keyboard entry, voice

Figure 2.5 Examples of agent types and their PEAS descriptions.

Properties and Classification of Task environment

Task environments vary along several significant dimensions. They can be -

- Fully observable or partially observable,
- Single-agent or Multi-agent,
- Deterministic or Nondeterministic,
- Episodic or Sequential,
- Static or Dynamic,
- Discrete or Continuous,
- Known or Unknown.

Fully observable vs Partially observable

Fully observable: If an agent’s sensors give it access to the Fully observable 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.

Fully observable vs Partially observable

Partially observable: 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.

Unobservable: If the agent has no sensors at all then the environment is unobservable.

Single-agent vs. multi-agent

Single-Agent: A **single-agent environment** is one in which **only one agent** is actively operating to achieve a goal. There are **no other agents** whose goals influence or conflict with the agent's performance.

- Example: Solving a **crossword puzzle** → The puzzle does not act against you. It does not try to "win".
- There is **no competition** or **cooperative behavior** with other agents.
- The agent focuses solely on maximizing its own performance.

Single-agent vs. multi-agent

Multi-agent : A **multi-agent environment** contains **two or more agents**, whose actions can affect one another. Each agent may have **its own performance measure**, and these measures may be:

- Competitive (one agent's gain is another's loss), or
- Cooperative (agents benefit from working together), or
- Mixed (sometimes cooperative, sometimes competitive).

Deterministic vs. Nondeterministic (Stochastic)

Deterministic: If the next state of the environment is completely determined by the current state and the action executed by the agent(s), then we say the environment is deterministic.

Nondeterministic(stochastic): If the next state of the environment is not completely determined by the current state and the action executed by the agent(s), then we say the environment is deterministic.

- We say that a model of the environment is stochastic if it explicitly deals with probabilities.

Episodic vs. sequential

Episodic: 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.

Example: An agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions.

Episodic vs. Sequential

Sequential: In sequential environments, 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 vs. dynamic

If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise, it is static.

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

Static vs. dynamic

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 semi dynamic.

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 semi dynamic.
- Crossword puzzles are static.

Discrete vs. Continuous

Discrete : The discrete has a finite number of distinct states percepts and actions

- Example: The chess environment has a finite number of distinct states (excluding the clock). Chess also has a discrete set of percepts and actions.

Continuous: It has continuous states, percepts and actions.

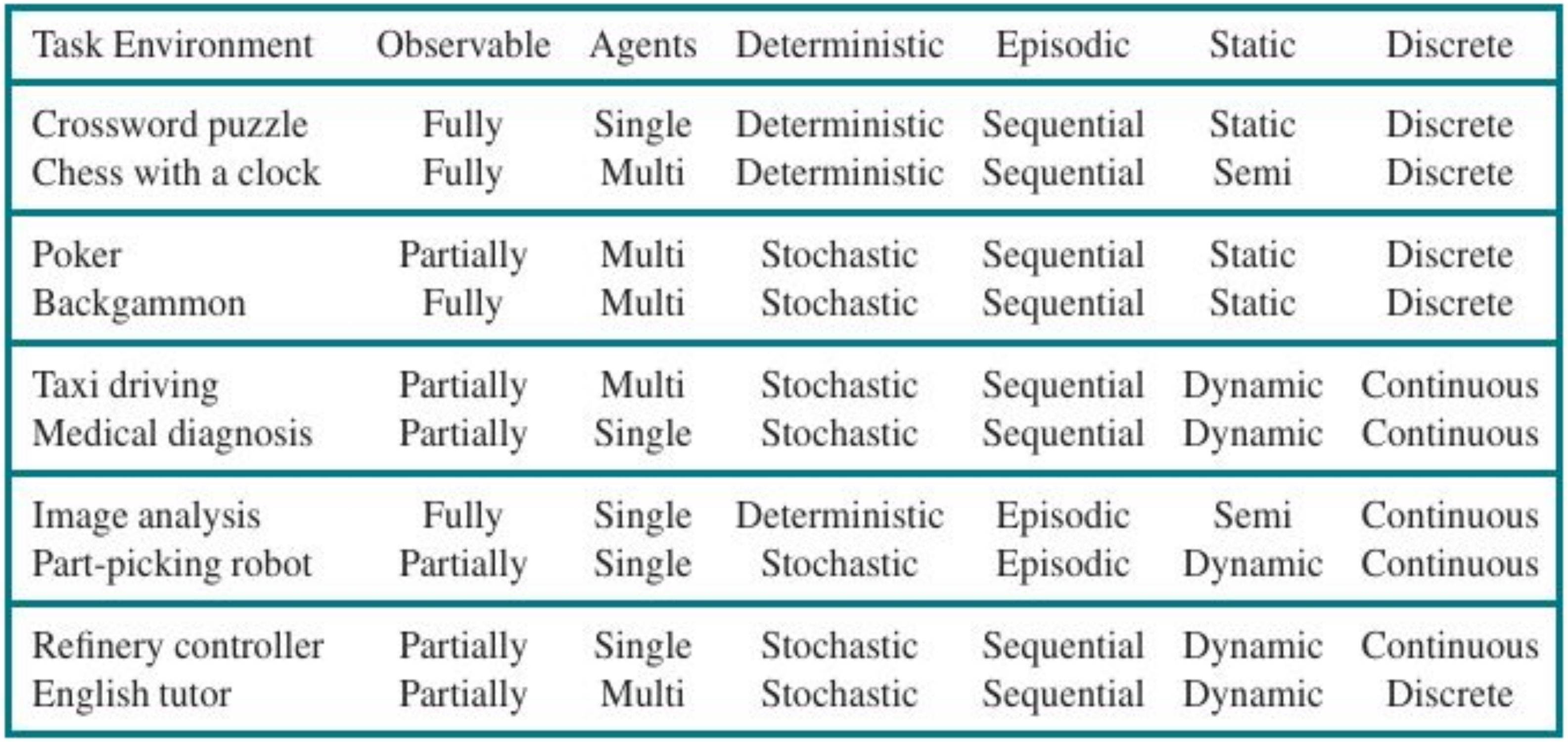
- Example: Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time. Taxi-driving actions are also continuous.

Known vs. Unknown

Known: In a known environment, the outcomes(or outcome probabilities if the environment is nondeterministic) for all actions are given.

Unknown: Obviously, if the environment is unknown, the agent will have to learn how it works in order to make good decisions.

Example of Task Environment



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
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 agent architecture:

agent = Agent architecture + Agent program.

- Obviously, the program we choose has to be one that is appropriate for the architecture.

Agent Program

Four basic kinds of agent programs for all intelligent systems:

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

Simple reflex agents

- The simplest kind of agent.
- These agents select actions on the basis of the current percept.
- Ignores the rest of the percept history.

Example: The vacuum Cleaner agent.

Schematic Diagram of Simple reflex agents

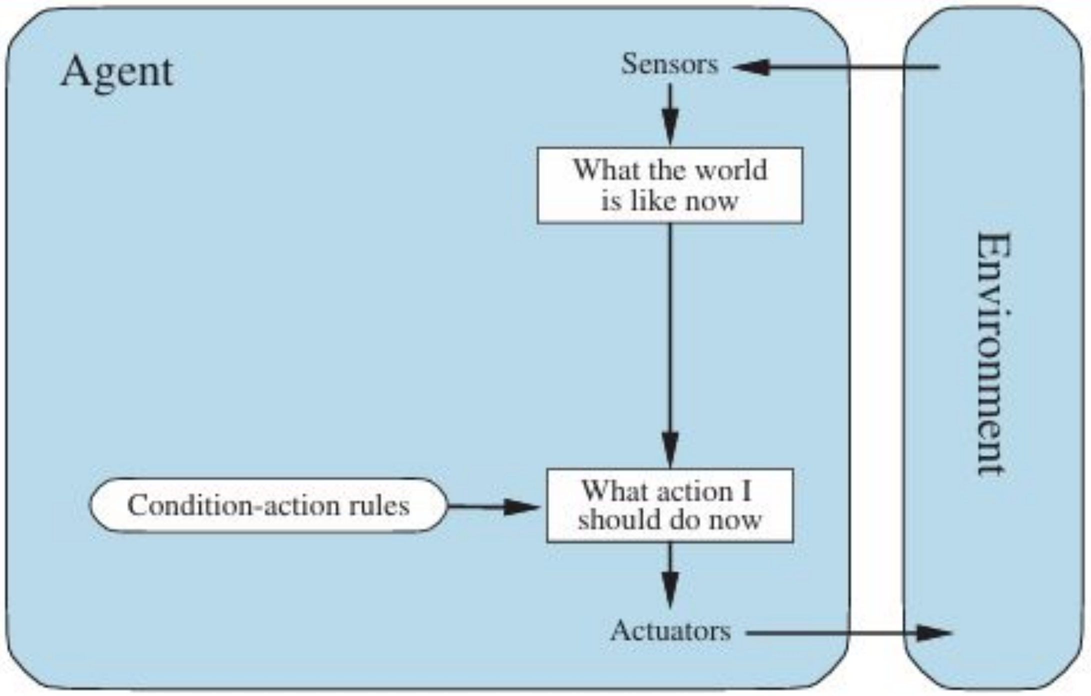
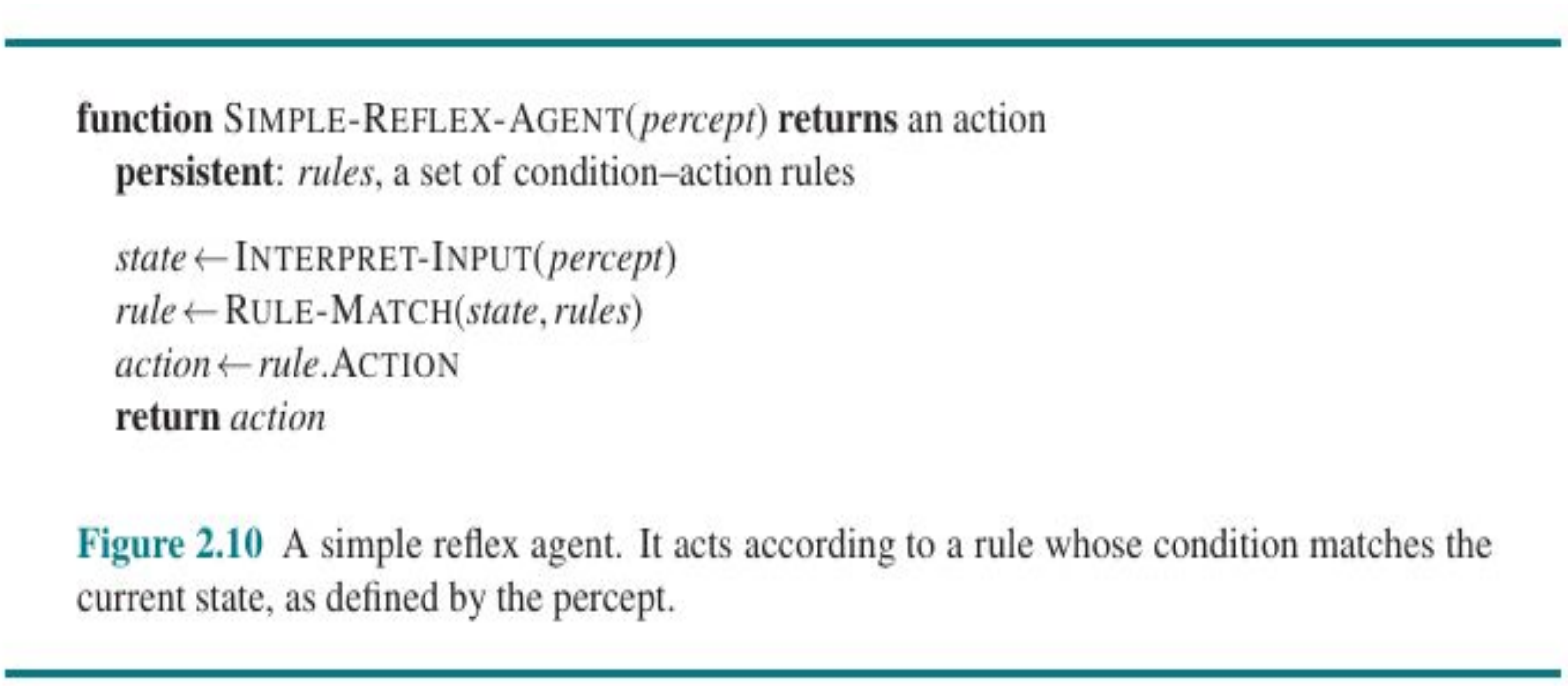


Figure 2.9 Schematic diagram of a simple reflex agent. We use rectangles to denote the current internal state of the agent's decision process, and ovals to represent the background information used in the process.

Agent program for 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
```

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

Strength of Simple Reflex Agents

- Simple and easy to implement.
- Simple reflex agents are fast and efficient.
- Suitable for a fully observable environment.

Weakness of Simple Reflex Agents

- Have limited adaptability.
- Cannot learn from past experiences.
- Cannot perform well in partially observable environment.

Model based Reflex Agent

- 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.
- Internal state depends on the percept history and thereby reflects at least some of the unobserved Internal state aspects of the current state.

Model-based reflex agents

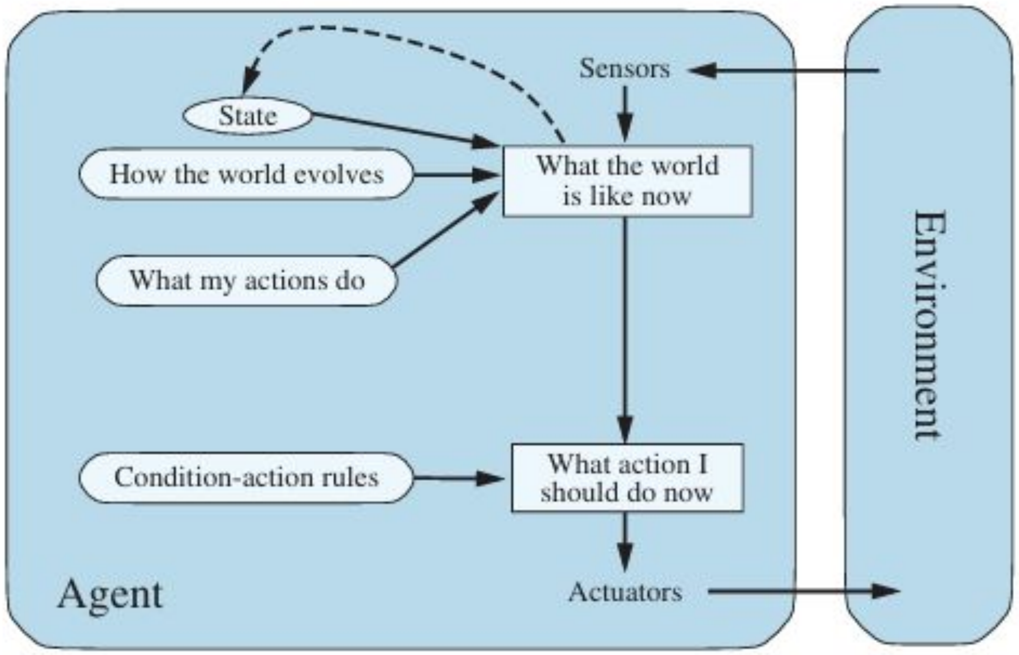
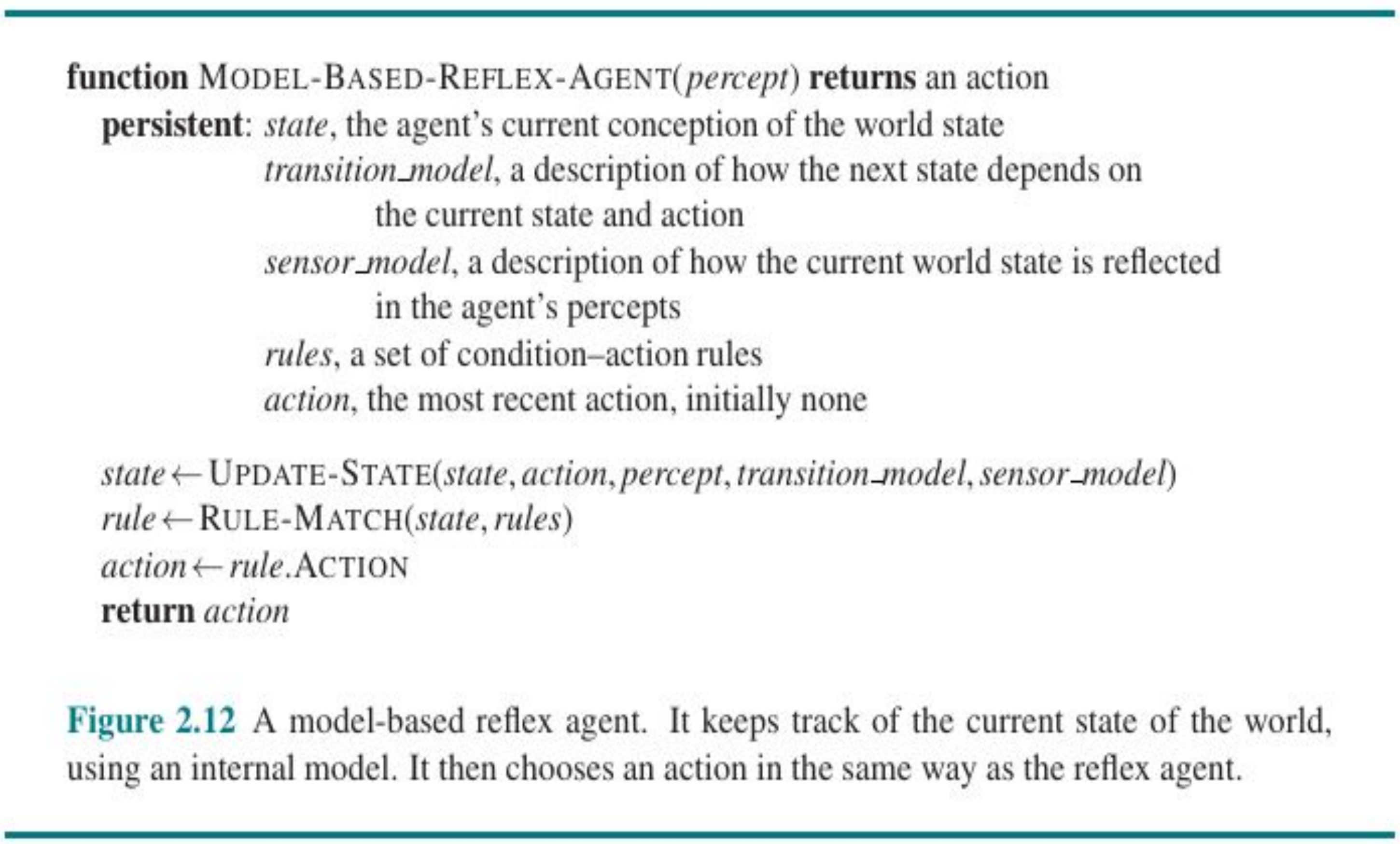


Figure 2.11 A model-based reflex agent.

Agent Program for model-based reflex



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

  state ← UPDATE-STATE(state, action, percept, transition_model, sensor_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.

Strength of Model based Reflex Agents

- Can handle partially observable environments.
- Model based reflex agents are more flexible because they can use the internal model to make predictions about how the environment might react to their actions.

Weakness of Model based Reflex Agents

- There is an increased level of complexity in comparison with simple reflex agent.
- The agent's performance relies heavily on the accuracy of its internal model.
- There is always an issue of limited learning.

Goal Based Agents

- Goal based agents extend the concept of model based agents by incorporating an planning to achieve the goal.
- They have a specific objective or goal and actively plan their action to achieve it.

Example: Navigation apps, Game playing(Chess).

Goal-based agents

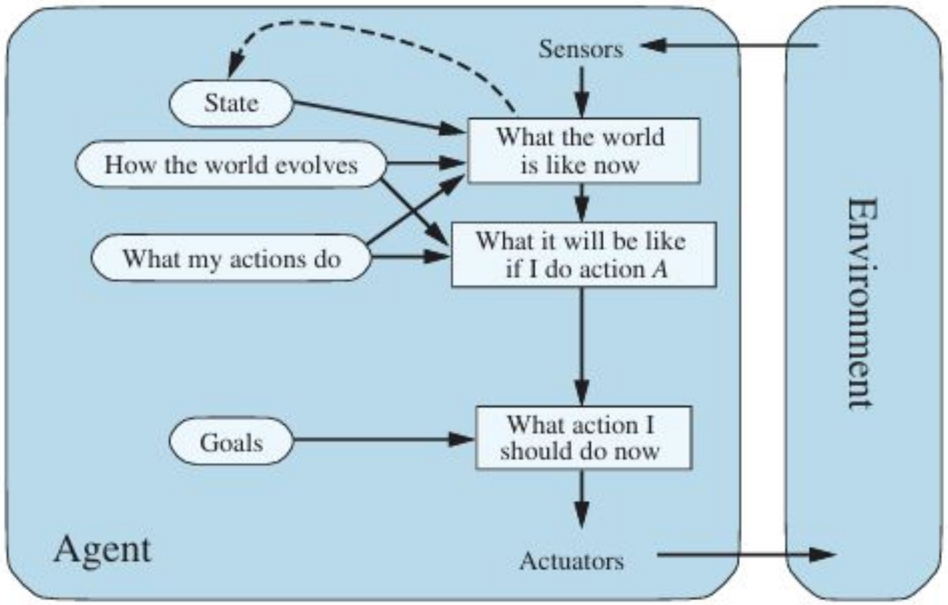

Sensors
State
What the world is like now
How the world evolves
What my actions do
What it will be like if I do action *A*
Goals
What action I should do now
Agent
Actuators
Environment

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.

Ar

Strength of Goal based Agents

- They can adapt their behavior depending on the current situation.
- These AI agents function in environments with multiple possible outcomes.
- They have solid reasoning capability.

Weakness of Goal based Agents

- The planning algorithms can be computationally expensive.
- Defining clear goal is also crucial for the agent’s success.
- If the agent doesn’t have the complete information about the environment, planning is not possible.

Utility-based agents

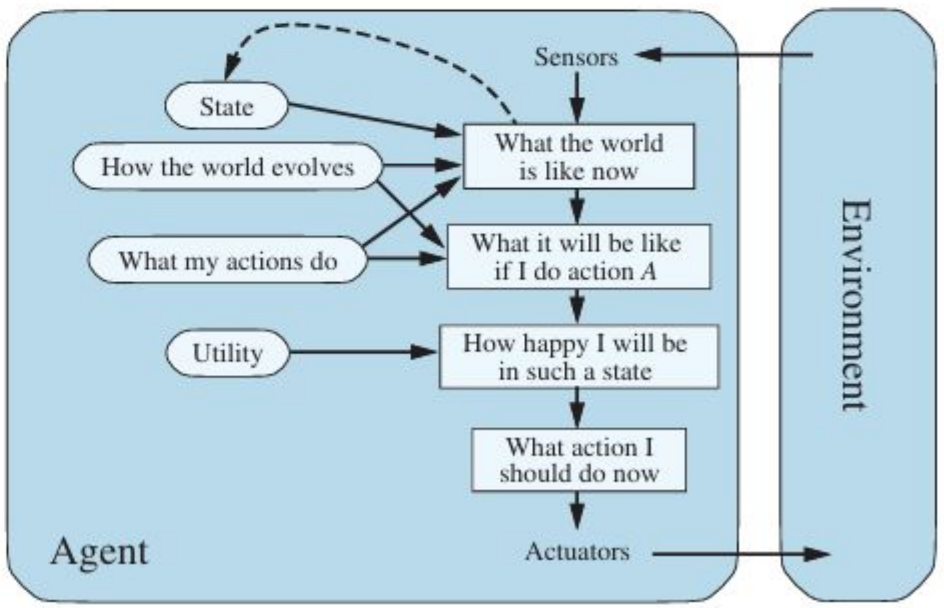

Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action *A*
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

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

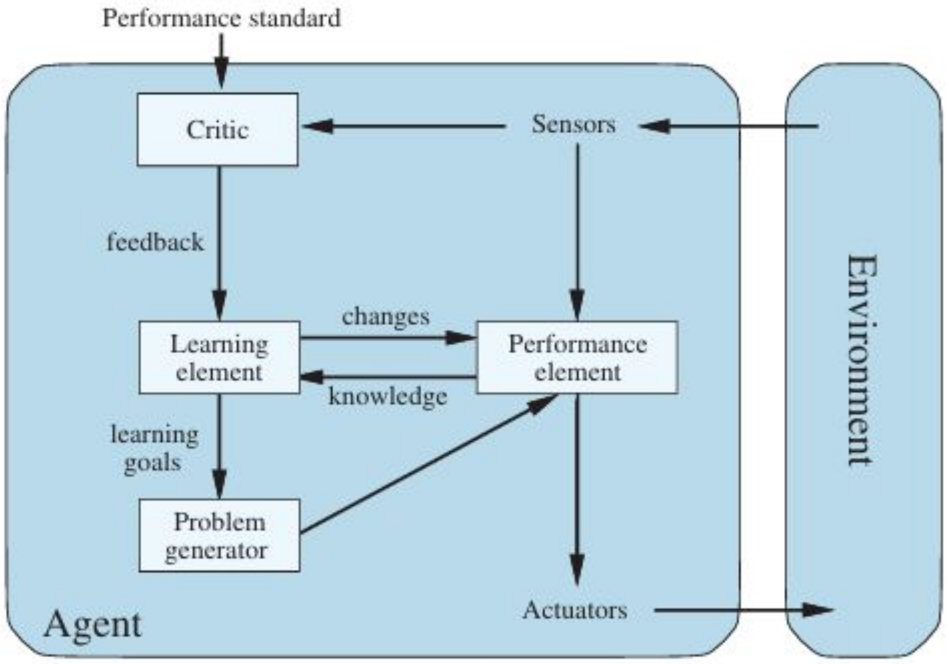


Figure 2.15 A general learning agent. The "performance element" box represents what we have previously considered to be the whole agent program. Now, the "learning element" box gets to modify that program to improve its performance.

How the components of agent programs work

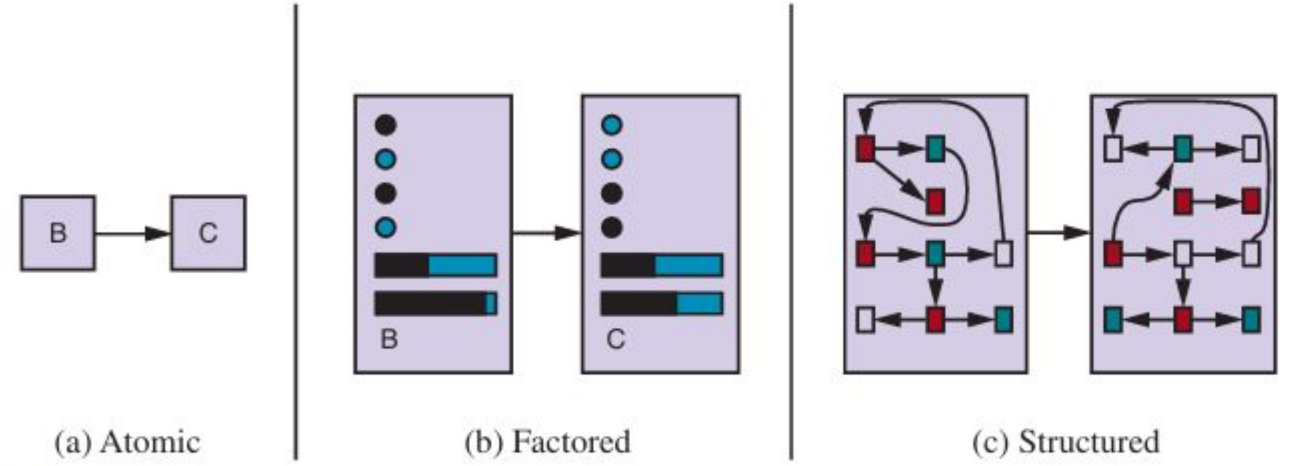


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed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

Uninformed Search

Chapter 3

MD. SHAHARIAR SARKAR
Assistant Professor,
Department of Computer Science and Engineering
Military Institute of Science and Technology (MIST)

Problem Solving Agents

Imagine an agent enjoying a touring vacation in Romania.

Suppose the agent is

- Currently in the city of Arad and
- has a nonrefundable ticket to fly out of Bucharest the following day.
- The agent observes street signs and sees that there are three roads leading out of Arad:
  - One toward Sibiu,
  - One to Timisoara, and
  - One to Zerind.

Tour in Romania

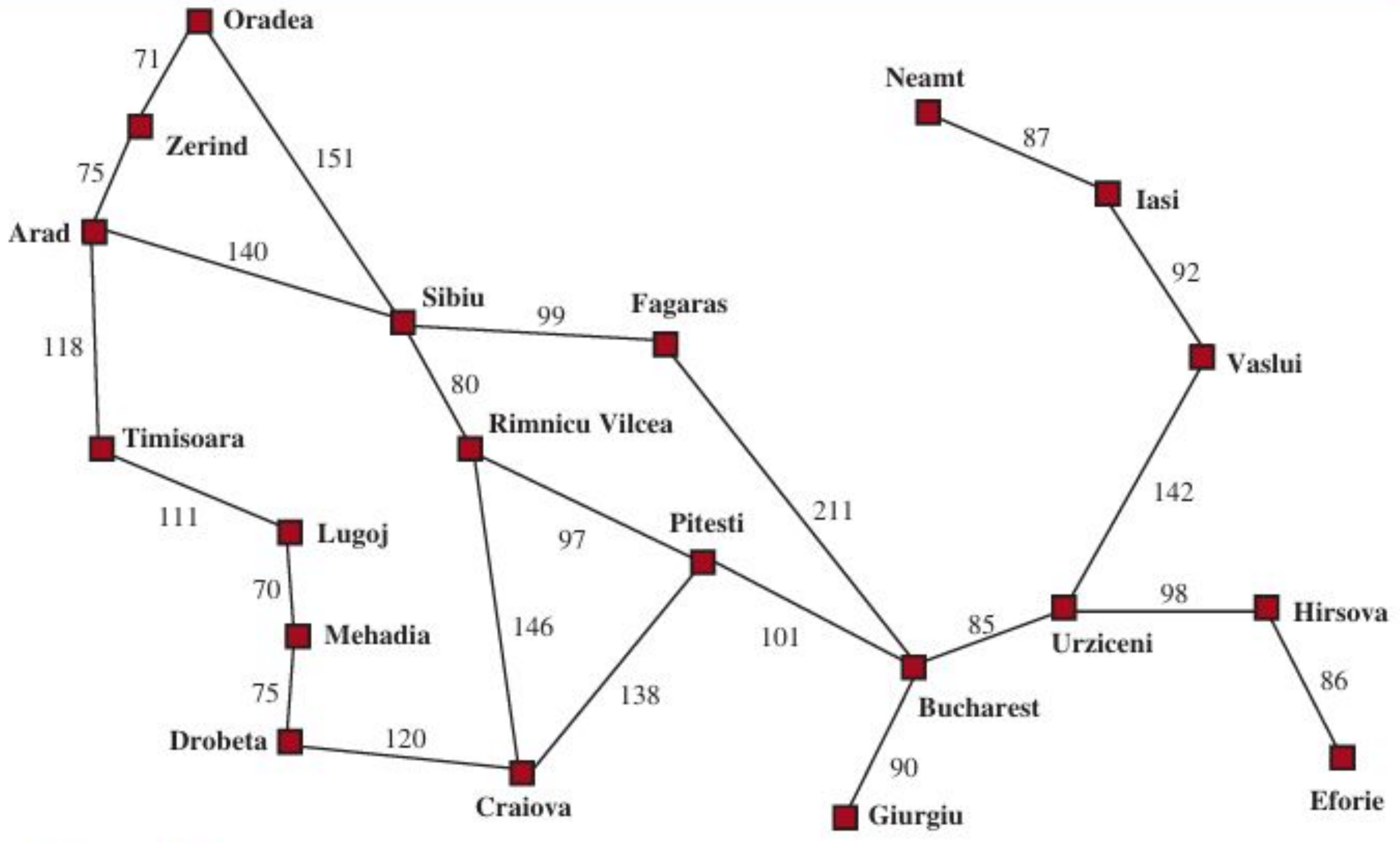


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Problem Solving Agents

If the environment is known, the agent can follow this four-phase problem-solving process:

- Goal formulation: The agent adopts the goal.
- Problem formulation: The agent devises a description of the states and actions necessary to reach the goal—an abstract model of the relevant part of the world.
- Search: Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal. Such a sequence is called a solution.
- Execution: The agent can now execute the actions in the solution, one at a time.

Search problems and solutions

A search problem can be defined formally as follows:

- A set of possible states called state space.
- The initial state that the agent starts in. For example: Arad.
- A set of one or more goal states. For Example: Bucharest.
- The actions available to the agent. Given a state s, ACTIONS(s) returns a finite set of actions that can be executed in s. We say that each of these actions is applicable in s.

  An example: ACTIONS(Arad) = {ToSibiu, ToTimisoara, ToZerind}

Search problems and solutions

- A transition model, which describes what each action does. RESULT(s, a) returns the Transition model state that results from doing action a in state s. For example, RESULT(Arad, ToZerind) = Zerind
- An action cost function, denoted by ACTION-COST(s, a, s′). Here, cost of applying action a in state s to reach state s′. For example, for route-finding agents, the cost of an action might be the length in miles or it might be the time it takes to complete the action.

Example of Problems

- Standardized problems.
- Real-world problems.

Example of Standardized problems.

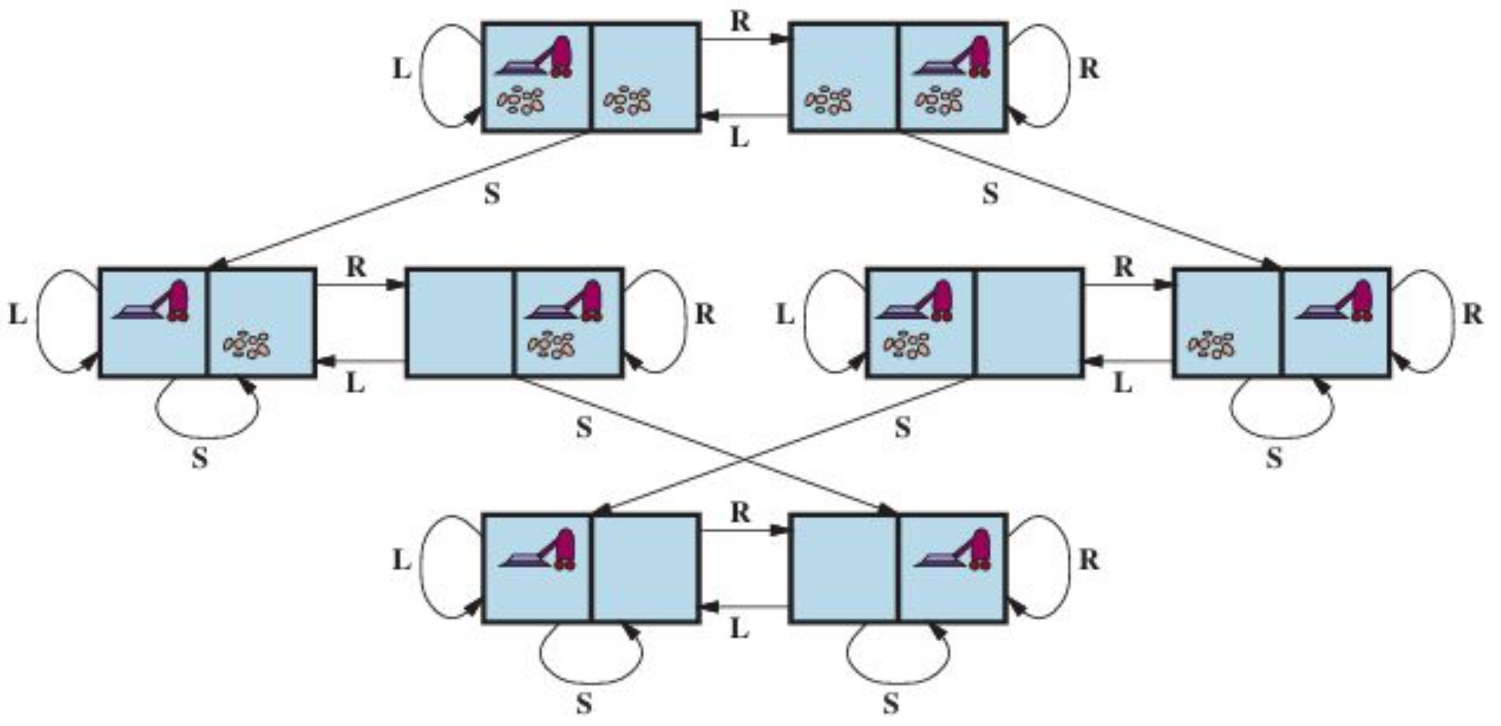


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

Vacuum World Example

- States: In the simple two-cell version, the agent can be in either of the two cells, and each call can either contain dirt or not, so there are 2·2·2 =8 states. In general, a vacuum environment with n cells has 2.2^n states.
- Initial state: Any state can be designated as the initial state.
- Actions: Suck, move Left, and move Right.

Vacuum World

- Transition model: Suck removes any dirt from the agent’s cell;
- Forward moves the agent ahead one cell in the direction it is facing, unless it hits a wall.
- Backward moves the agent in the opposite direction.
- Goal states: The states in which every cell is clean.
- Action cost: Each action costs 1.

8 puzzle Problem Example

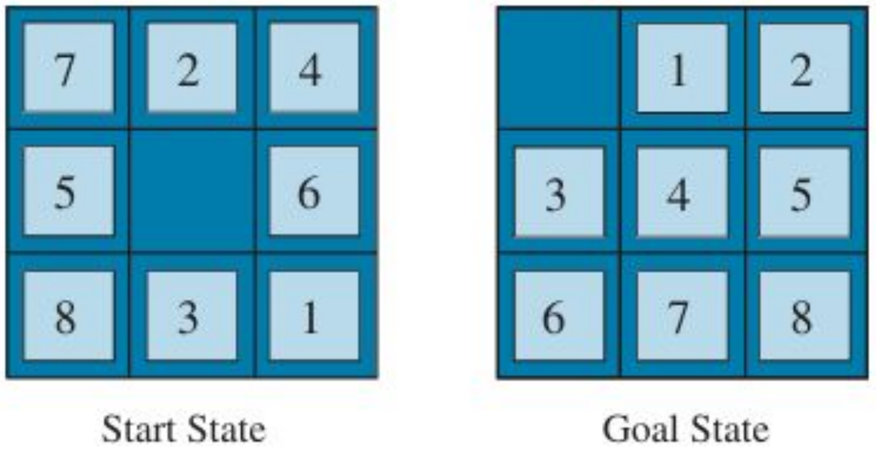


Figure 3.3 A typical instance of the 8-puzzle.

8 puzzle Problem Example

The standard formulation of the 8 puzzle is as follows:

- States: A state description specifies the location of each of the tiles.
- Initial state: Any state can be designated as the initial state.

8 puzzle Problem Example

- Actions: Moving Left, Right, Up, or Down.
- Transition model: Maps a state and action to a resulting state.
- Goal state: Although any state could be the goal, we typically specify a state with the numbers in order.
- Action cost: Each action costs 1.

Real-world problems

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: The user's home airport.
- 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.

Real-world problems

- Transition model: The state resulting from taking a flight will have the flight's destination as the new location and the flight's arrival time as the new time.
- Goal state: A destination city.
- Action cost:
  - A combination of monetary cost,
- waiting time,
  - flight time,
  - customs and immigration procedures, seat quality.
  - time of day, type of airplane, frequent-flyer re ward points, and so on.

Measuring problem-solving performance

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

- Completeness: Is the algorithm guaranteed to find a solution when there is one, and to Completeness correctly report failure when there is not?
- Cost optimality: Does it find a solution with the lowest path cost of all solutions?
- Time complexity: How long does it take to find a solution? This can be measured in Time complexity seconds, or more abstractly by the number of states and actions considered.
- Space complexity: How much memory is needed to perform the search?

Uninformed Search

An uninformed search algorithm is given no clue about how close a state is to the goal(s).

Uninformed Search Algorithms differ based on which node they expand first:–

- Best-first search- selects nodes for expansion using to an evaluation function.
- Breadth-first search- expands the shallowest nodes first; it is complete, optimal for unit action costs, but has exponential space complexity.
- Depth-first search- expands the deepest unexpanded

Uninformed Search

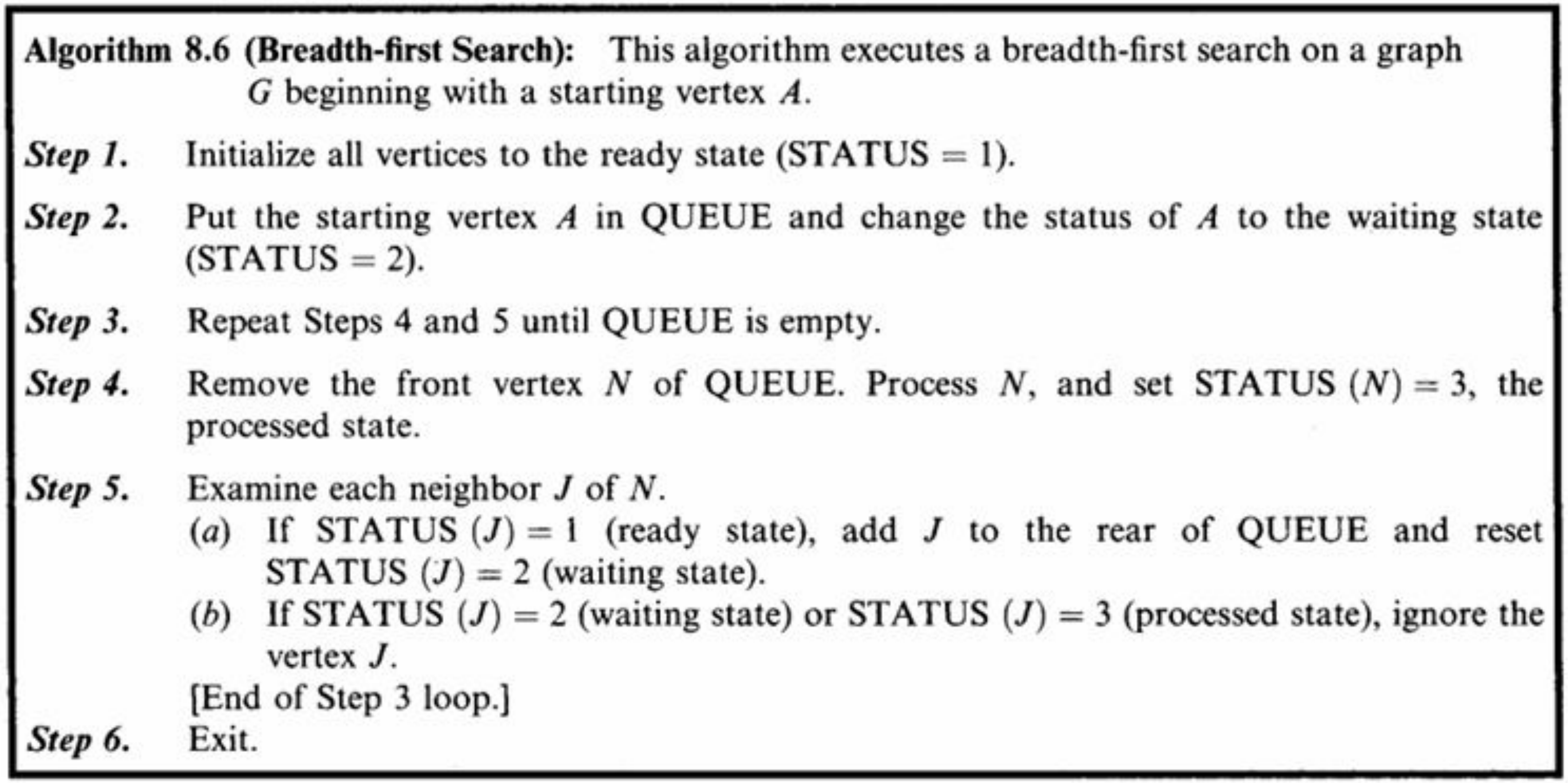
- Depth-limited search- adds a depth bound.
- Uniform-cost search- expands the node with lowest path cost, g(n), and is optimal for general action costs.
- Iterative deepening search- calls depth-first search with increasing depth limits until a goal is found. It is complete when full cycle checking is done, optimal for unit action costs, has time complexity comparable to breadth-first search, and has linear space complexity.
- Bidirectional search- expands two frontiers, one around the initial state and one around the goal, stopping when the two frontiers meet

Best-first search

Best-first search, in which we choose a node, n, with minimum value of Best-first search evaluation function, f(n).

- On each iteration we choose a node on the frontier with minimum f(n) value,
- return it if its state is a goal state, and otherwise apply EXPAND to generate child nodes.
- Each child node is added to the frontier if it has not been reached before, or is re-added if it is now being reached with a path that has a lower path cost than any previous path.
- The algorithm returns either an indication of failure, or a node that represents a path to a goal. By employing different f(n) functions, we get different specific algorithms.

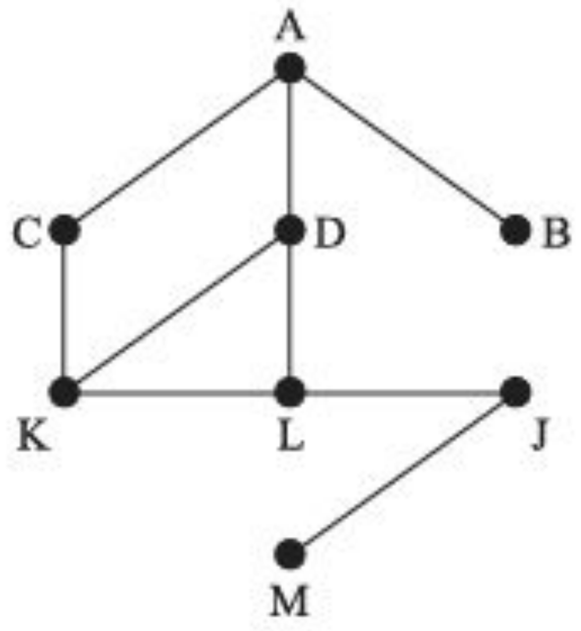
Breadth First Search



Algorithm 8.6 (Breadth-first Search): This algorithm executes a breadth-first search on a graph G beginning with a starting vertex A.

```
Step 1. Initialize all vertices to the ready state (STATUS = 1).
Step 2. Put the starting vertex A in QUEUE and change the status of A to the waiting state
        (STATUS = 2).
Step 3. Repeat Steps 4 and 5 until QUEUE is empty.
Step 4.     Remove the front vertex N of QUEUE. Process N, and set STATUS (N) = 3, the
            processed state.
Step 5.     Examine each neighbor J of N.
            (a) If STATUS (J) = 1 (ready state), add J to the rear of QUEUE and reset
                STATUS (J) = 2 (waiting state).
            (b) If STATUS (J) = 2 (waiting state) or STATUS (J) = 3 (processed state), ignore the
                vertex J.
        [End of Step 3 loop.]
Step 6. Exit.
```

Breadth First Search



(a)

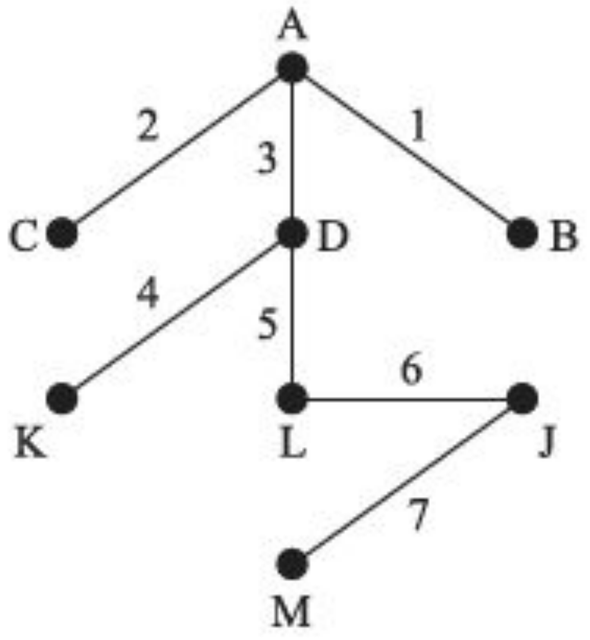
Vertex	Adjacency list
A	B, C, D
B	A
C	A, K
D	A, K, L
J	L, M
K	C, D, L
L	D, J, K
M	J

(b)

Breadth First Search

QUEUE	Vertex
A	A
D, C, B	B
D, C	C
D	D
L, K	K
L	L
J	J
M	M
∅	

(*a*)



(*b*)

Complexity of BFS

In terms of time and space, imagine searching a uniform tree where every state has b successors. The root of the search tree generates b nodes, 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. Then the total number of nodes generated is

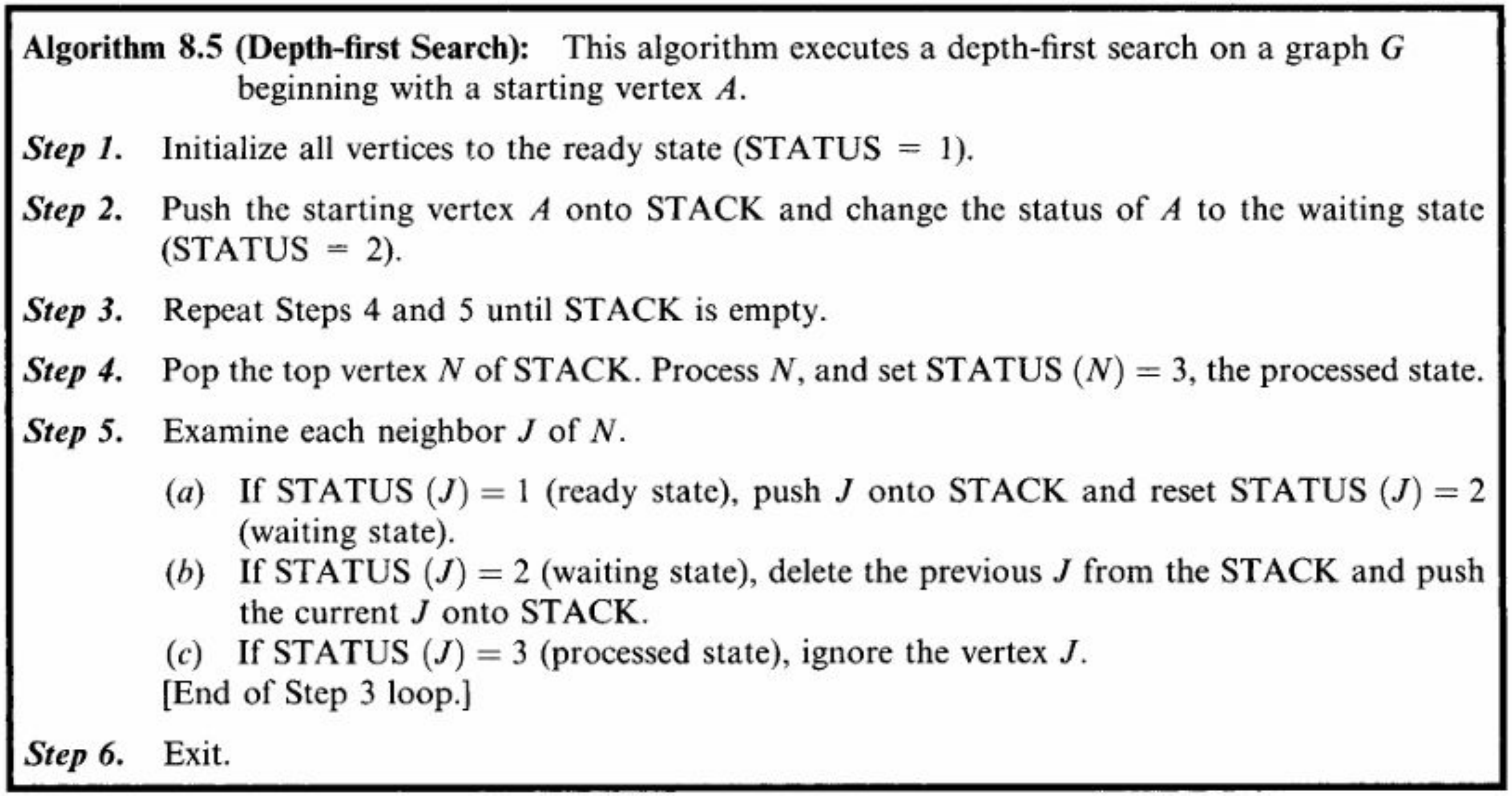
$$1+b+b^2+b^3+\cdots+b^d=O(b^d)$$

All the nodes remain in memory, so both time and space complexity are $O(b^d)$

Complexity of BFS

As a typical real-world example, consider a problem with branching factor b = 10, processing speed 1 million nodes/second, and memory requirements of 1 Kbyte/node. A search to depth d = 10 would take less than 3 hours, but would require 10 terabytes of memory. The memory requirements are a bigger problem for breadth-first search than the execution time. But time is still an important factor. At depth d = 14, even with infinite memory, the search would take 3.5 years. In general, exponential-complexity search problems cannot be solved by uninformed search for any but the smallest instances.

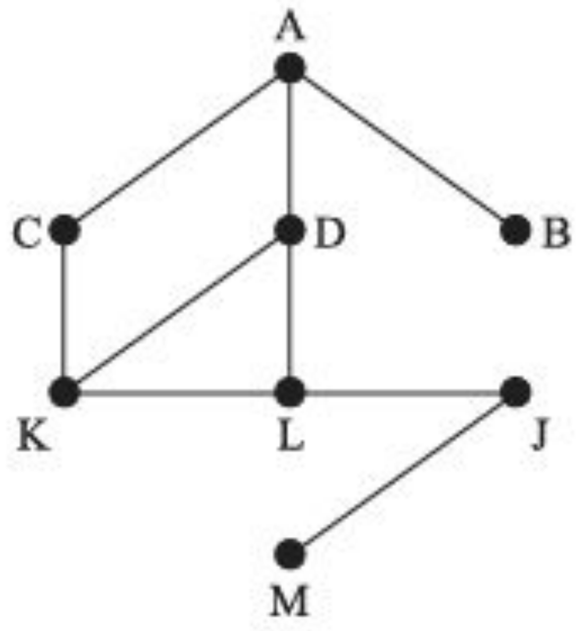
Depth First Search



Algorithm 8.5 (Depth-first Search): This algorithm executes a depth-first search on a graph G beginning with a starting vertex A.

```
Step 1. Initialize all vertices to the ready state (STATUS = 1).
Step 2. Push the starting vertex $A$ onto STACK and change the status of $A$ to the waiting state
        (STATUS = 2).
Step 3. Repeat Steps 4 and 5 until STACK is empty.
Step 4.     Pop the top vertex $N$ of STACK. Process $N$, and set STATUS ($N$) = 3, the processed state.
Step 5.     Examine each neighbor $J$ of $N$.
                (a) If STATUS ($J$) = 1 (ready state), push $J$ onto STACK and reset STATUS ($J$) = 2
                    (waiting state).
                (b) If STATUS ($J$) = 2 (waiting state), delete the previous $J$ from the STACK and push
                    the current $J$ onto STACK.
                (c) If STATUS ($J$) = 3 (processed state), ignore the vertex $J$.
            [End of Step 3 loop.]
Step 6. Exit.
```

Breadth First Search



(*a*)

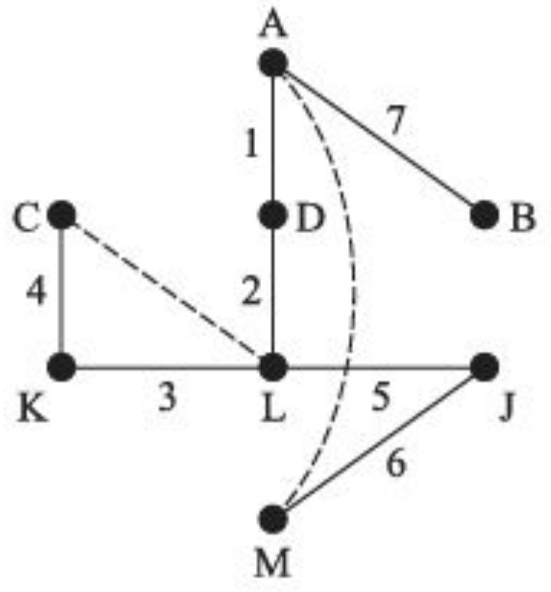
Vertex	Adjacency list
A	B, C, D
B	A
C	A, K
D	A, K, L
J	L, M
K	C, D, L
L	D, J, K
M	J

(*b*)

Depth First Search

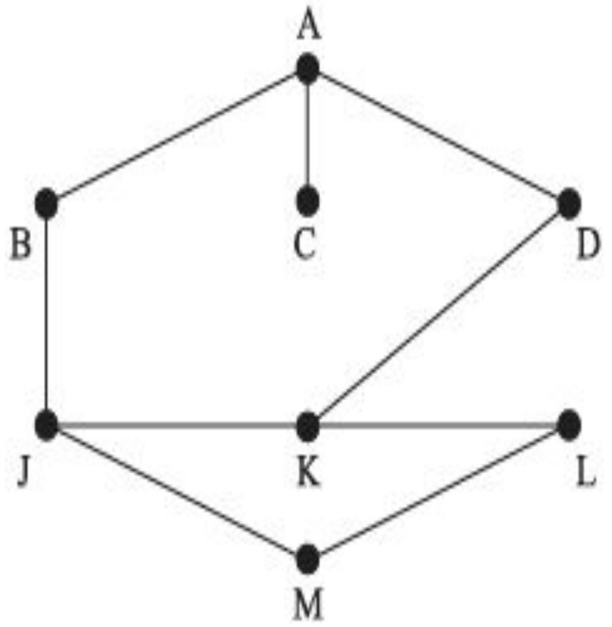
STACK	Vertex
A	A
D, C, B	D
L, K, C, B	L
K, J, $\cancel{K}$, C, B	K
C, J, $\cancel{C}$, B	C
J, B	J
M, B	M
B	B
∅	

(*a*)

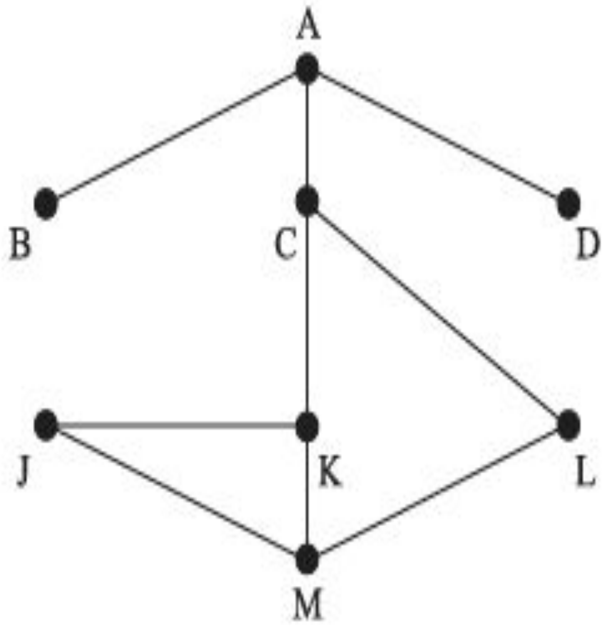


(*b*)

Practice for BFS and DFS



(a)



(b)

Complexity of Depth First Search

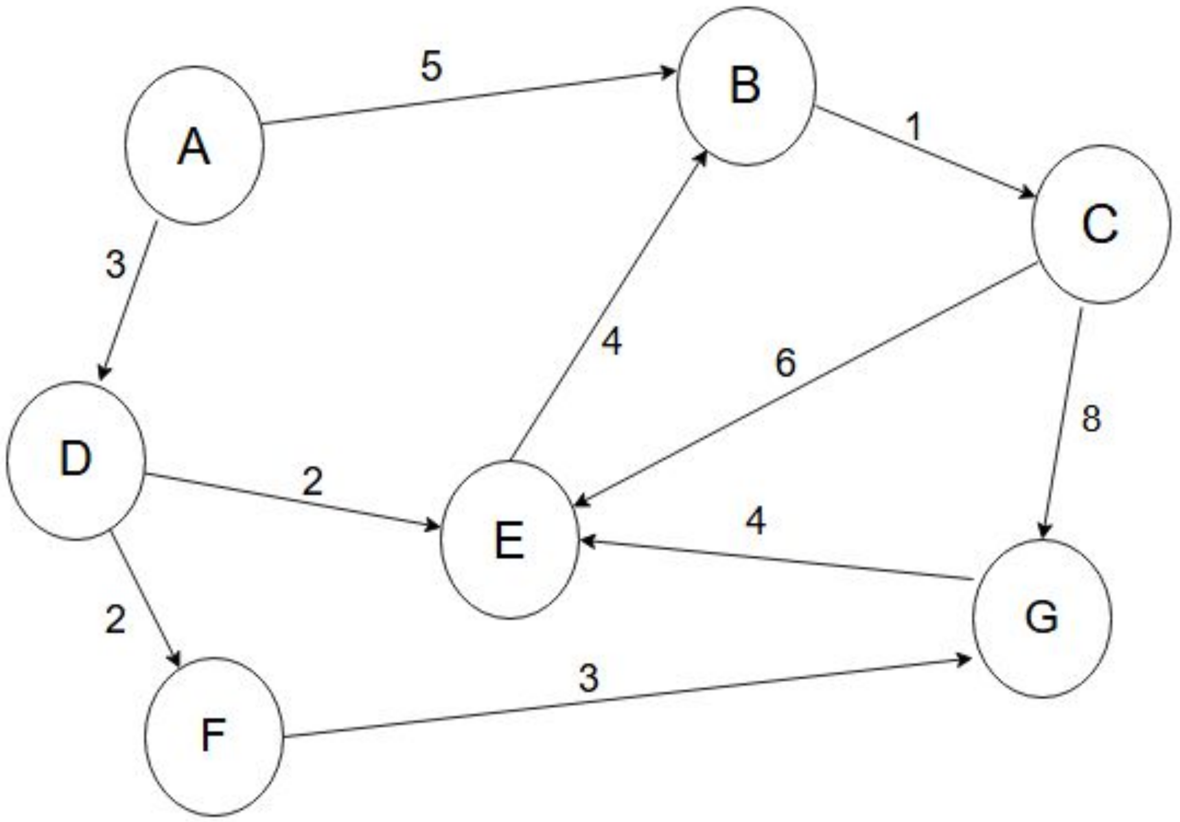
- A depth-first tree-like search takes time proportional to the number of states, and has memory complexity of only O(bm), where b is the branching factor and m is the maximum depth of the tree.

Answer of the Self Work

DFS for (a): A, D, K, L, M, J, C, B.
DFS for (b): A, D, C, L, M, K, J, B.

BFS for (a): A, B, C, D, J, K, M, L.
BFS for (b): A, B, C, D, K, L, J, M.

Dijkstra’s algorithm or uniform-cost search



Finding path A to G using uniform cost search

Dijkstra’s algorithm or uniform-cost search

Concept:

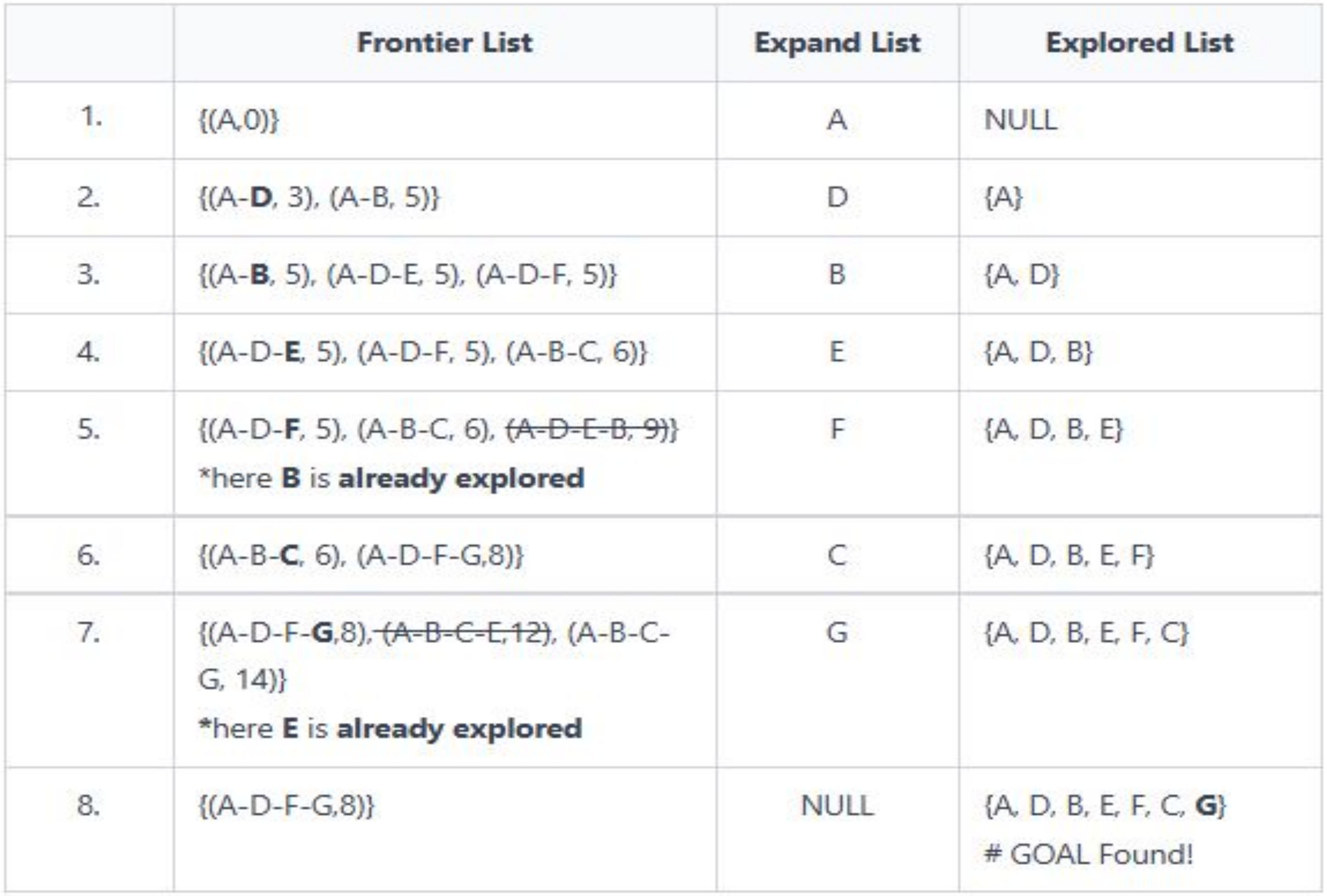
- Frontier list will be based on the priority queue. Every new node will be added at the end of the list and the list will give priority to the least cost path.
- The node at the top of the **frontier list** will be added to the expand list, which shows that this node is going to be explored in the next step. It will not repeat any node. If the node has already been explored, you can discard it.
- Explored list will be having the nodes list, which will be completely explored.

Dijkstra’s algorithm or uniform-cost search

Algorithm:

- Add the Starting node S in the frontier list with the path cost g(n) = 0 (starting point is at 0 path cost).
- Add this node to the Explored list, as we only have a single node in the frontier list. If we have multiple nodes, then we will add that one node at the top of the frontier.
- Now, explore this node by visiting all of its child nodes. After that, add this node to the Explored list, as it is now fully explored.
- Check if the added node is the goal node or not. Stop if the goal node is found, or else move on to the next step**.**
- Since new nodes are added to the frontier list, we need to compare and set the priority queue again, depending upon the priority, that is, the minimum path cost g(n).
- Now, move to back to step **2** and repeat the steps until the goal node is not added to the explored list.

Dijkstra’s algorithm or uniform-cost search



	Frontier List	Expand List	Explored List
1.	{(A,0)}	A	NULL
2.	{(A-**D**, 3), (A-B, 5)}	D	{A}
3.	{(A-**B**, 5), (A-D-E, 5), (A-D-F, 5)}	B	{A, D}
4.	{(A-D-**E**, 5), (A-D-F, 5), (A-B-C, 6)}	E	{A, D, B}
5.	{(A-D-**F**, 5), (A-B-C, 6), ~~(A-D-E-B, 9)~~} *here **B** is **already explored**	F	{A, D, B, E}
6.	{(A-B-**C**, 6), (A-D-F-G,8)}	C	{A, D, B, E, F}
7.	{(A-D-F-**G**,8), ~~(A-B-C-E,12)~~, (A-B-C-G, 14)} *here **E** is **already explored**	G	{A, D, B, E, F, C}
8.	{(A-D-F-G,8)}	NULL	{A, D, B, E, F, C, **G**} # GOAL Found!

Dijkstra’s algorithm or uniform-cost search

Hence we get:
Actual path => A -- D -- F -- G , with Path Cost = 8.
Traversed path => A -- D -- B -- E -- F -- C – G.

Evaluation

The uniform cost search algorithm can be evaluated by four of the following factors:

- **Completeness**: UCS is complete if the branching factor **b** is finite.
- **Time complexity**: The time complexity of UCS is exponential O($\mathbf{b}^{(1+C/\varepsilon)}$), because every node is checked.
- **Space completeness**: The space complexity is also exponential O($\mathbf{b}^{(1+C/\varepsilon)}$), because all the nodes are added to the list for comparisons of priority.
- **Optimality**: UCS gives the optimal solution or path to the goal node.

Depth Limited Search

To prevent depth-first search from going down an infinite path, we can use depth-limited search, a version of depth-first search.

- In which we supply a depth limit, ℓ, and treat all nodes at depth ℓ as if they had no successors.
- The time complexity is $O(b\ell)$ and the space complexity is $O(b\ell)$.
- Unfortunately, if we make a poor choice for ℓ the algorithm will fail to reach the solution, making it incomplete again.
- Since depth-first search is a tree-like search, we can't keep it from wasting time on redundant paths in general, but we can eliminate cycles at the cost of some computation time.

Depth Limited Search

- If we look only a few links up in the parent chain we can catch most cycles; longer cycles are handled by the depth limit. Sometimes a good depth limit can be chosen based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, $\ell=19$ is a valid limit. But if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 actions. This number, known as the diameter of the state-space graph, gives us a better depth limit, which leads to a more efficient depth-limited search. However, for most problems we will not know a good depth limit until we have solved the problem.

Iterative deepening search

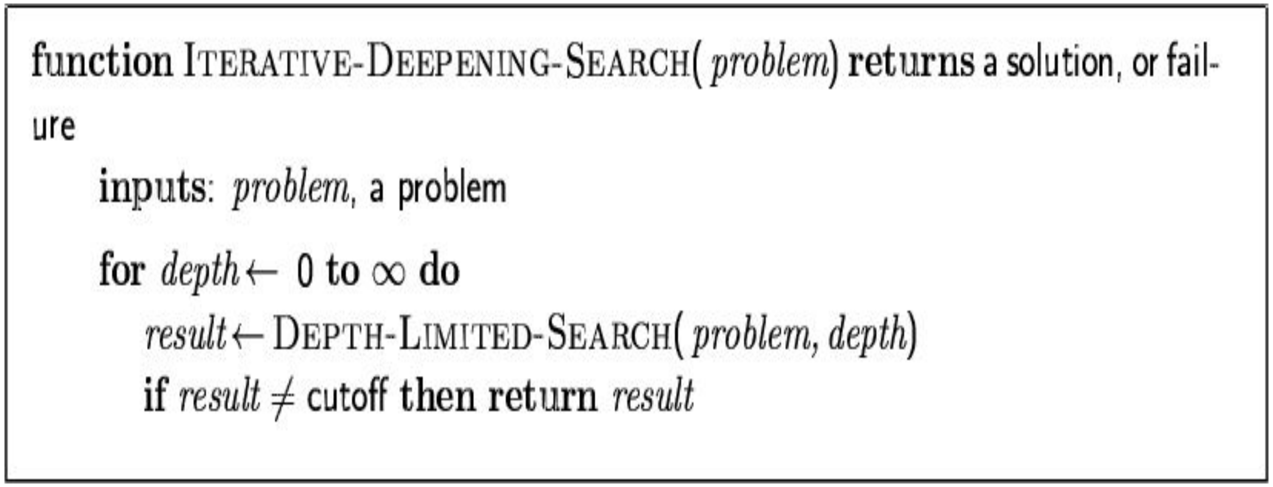
- Iterative deepening search solves the problem of picking a good value for ℓ by trying all values: first 0, then 1, then 2, and so on—until either a solution is found, or the depth limited search returns the failure value rather than the cutoff value.

- Iterative deepening combines many of the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest: $O(bd)$ when there is a solution, or $O(bm)$ on finite state spaces with no solution.

Iterative deepening search

- Like breadth-first search, iterative deepening is optimal for problems where all actions have the same cost, and is complete on finite acyclic state spaces, or on any finite state space when we check nodes for cycles all the way up the path.
- The time complexity is when there is a $O(b^m)$ solution, or $O(b^d)$ when there is none.

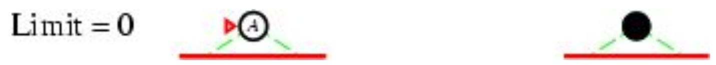
Iterative deepening search



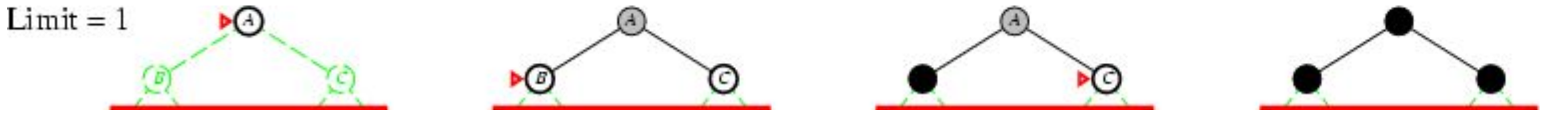
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or fail-
ure
    inputs: problem, a problem

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

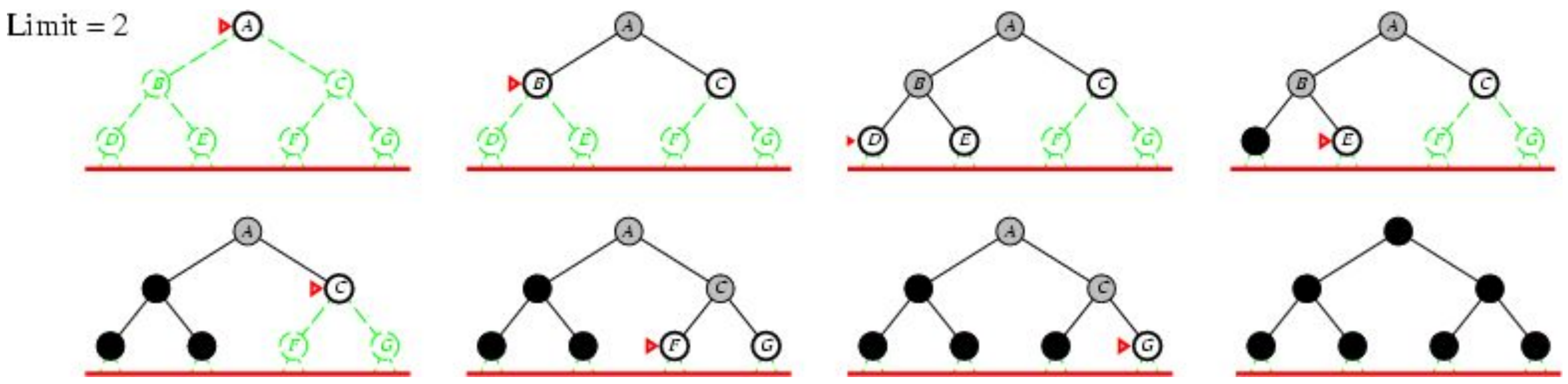
Iterative deepening search $l = 0$



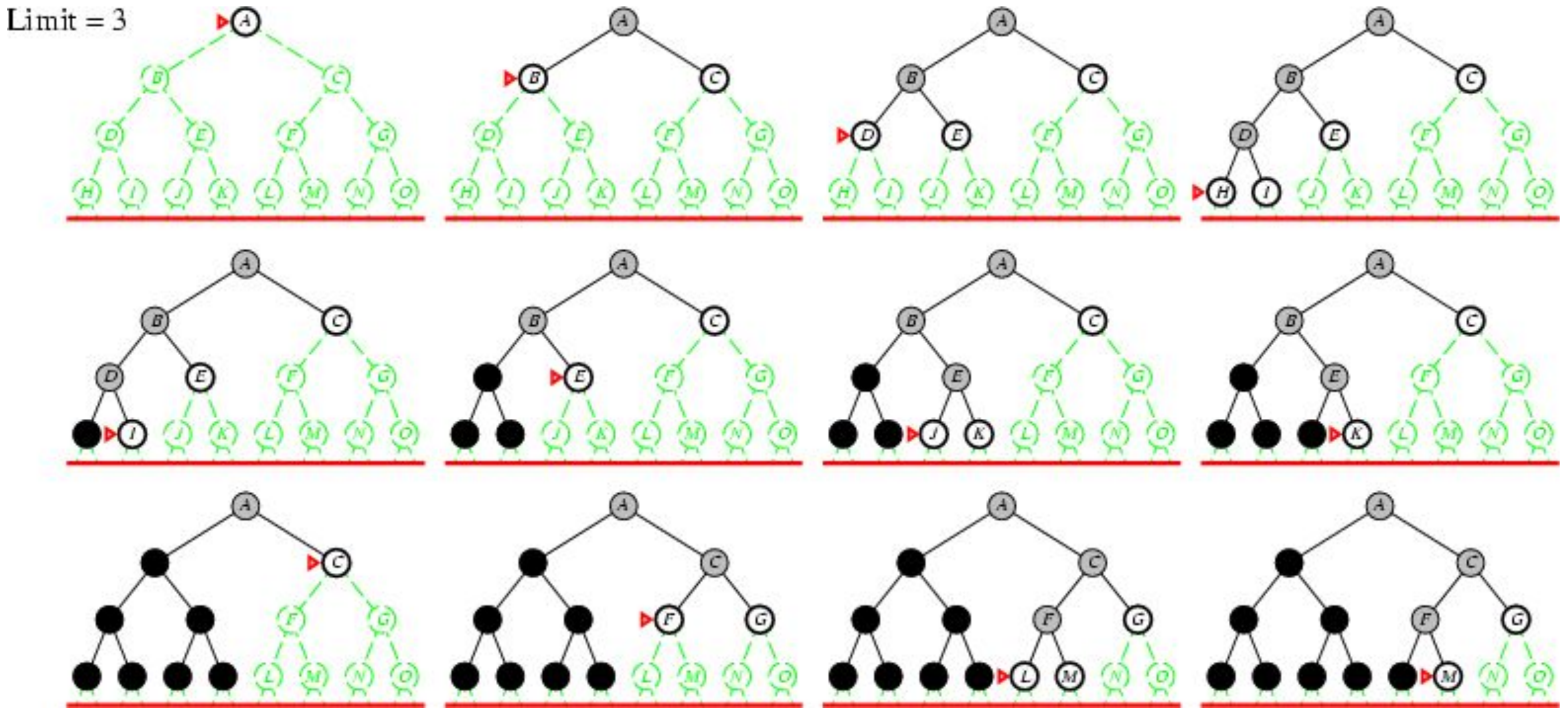
Iterative deepening search l =1



Iterative deepening search *l* =2



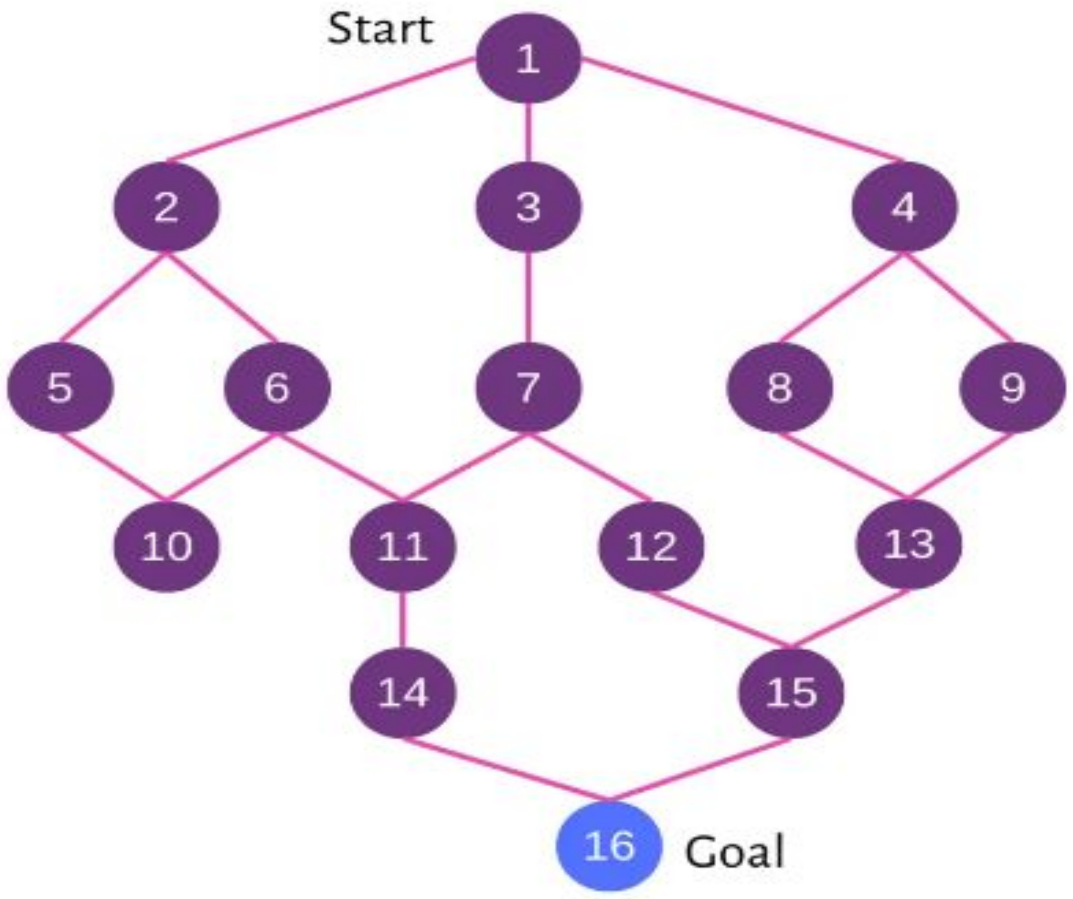
Iterative deepening search l =3



Bidirectional search

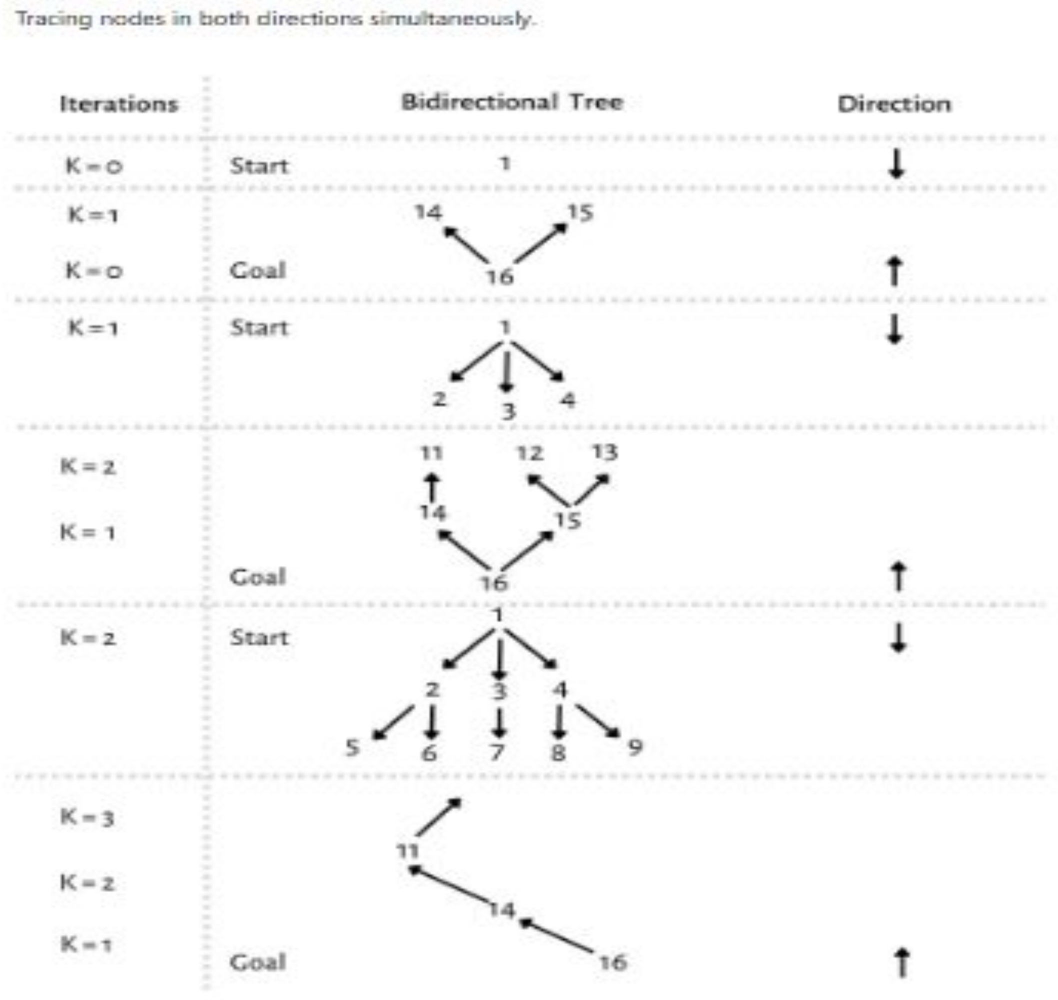
- An alternative approach called bidirectional search simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet.
- The motivation is that is $b^{d/2} + b^{d/2}$ much less than b^d

Bidirectional search

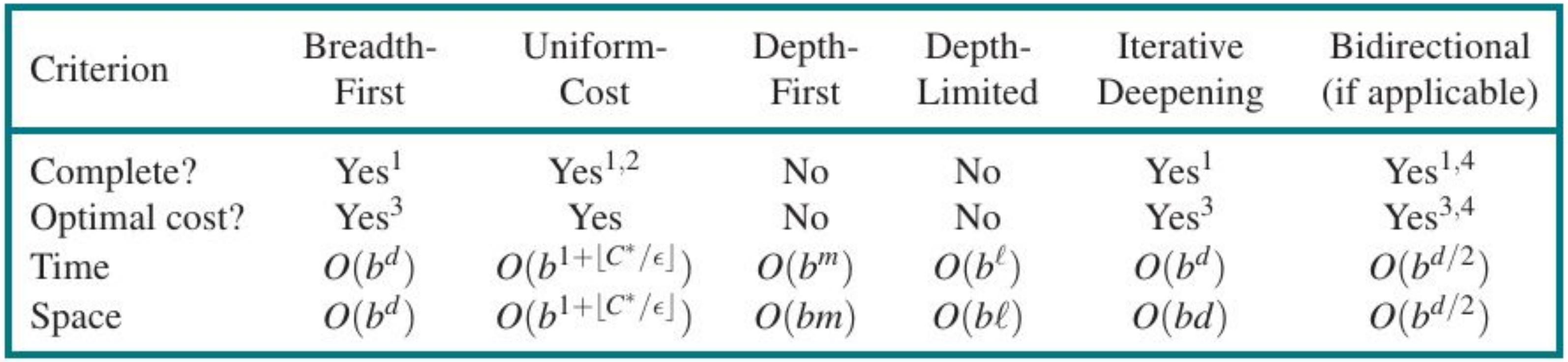


Bidirectional Graph 1

Bidirectional search



Complexity Comparison



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Figure 3.15 Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \epsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.