Module-01

Introduction | Problem-solving

Chapter: - Introduction: What is AI? Foundations and History of AI

1. Define Artificial Intelligence and list the task domains of Artificial Intelligence.

Ans: -

"It is a branch of computer science by which we can create intelligent machines which can behave like a human, think like humans, and able to make decisions."

OR

“The science and engineering of making intelligent machines, especially intelligent computer programs**”.**

- Perception
- Machine vision
- Speech understanding
- Touch (tactile or haptic) sensation
- Robotics
- Natural Language Processing
- Natural Language Understanding
- Speech Understanding
- Language Generation
- Machine Translation
- Planning
- Expert Systems
- Machine Learning
- Theorem Proving
- Symbolic Mathematics
- Game Playing

SEARCH CREATORS

2. State and explain algorithm for Best First Search Algorithm with an example.

Ans: -

Best First Search (Informed Search)

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node.

So, both BFS and DFS blindly explore paths without considering any cost function.

The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore.

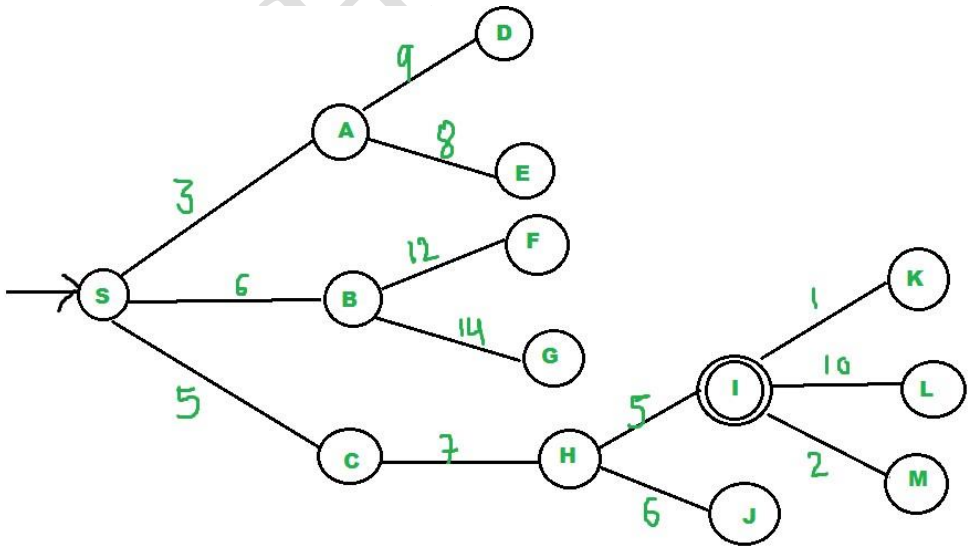
Best First Search falls under the category of Heuristic Search or Informed Search.

Algorithm: Best-First-Search (Grah g, Node start)

1) Create an empty PriorityQueue pq;

2) Insert "start" in pq. pq. insert(start)

3) Until PriorityQueue is empty u = PriorityQueue.DeleteMin



SEARCH CREATORS

- We start from source "S" and search for goal "I" using given costs and Best First search.
- pq initially contains S We remove s from and process unvisited neighbors of S to pq.
- pq now contains {A, C, B} (C is put before B because C has lesser cost) We remove A from pq and process unvisited neighbors of A to pq.
- pq now contains {C, B, E, D} 43 We remove C from pq and process unvisited neighbors of C to pq.
- pq now contains {B, H, E, D} We remove B from pq and process unvisited neighbors of B to pq.
- pq now contains {H, E, D, F, G} We remove H from pq.
- Since our goal "I" is a neighbor of H, we return. Analysis: The worst-case time complexity for Best First Search is O (n * Log n) where n is number of nodes.

3. **A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug.**
    a. **Write down the production rules for the above problem**
    b. **Write any one solution to the above problem**

Ans: -

Water Jug Problem

State: (x, y)

where x represents the quantity of water in a 4-liter jug and y represents the quantity of water in a 3-liter jug.

That is, x = 0, 1, 2, 3, or 4 y = 0, 1, 2, 3

Start state: (0, 0).

Goal state: (2, n) for any n.

Here need to start from the current state and end up in a goal state.

SEARCH CREATORS

Production Rules for Water Jug Problem in Artificial Intelligence

1	(x, y) is X<4 ->(4, Y)	Fill the 4-liter jug
2	(x, y) if Y<3 -> (x, 3)	Fill the 3-liter jug
3	(x, y) if x>0 -> (x-d, d)	Pour some water out of the 4-liter jug.
4	(x, y) if Y>0 -> (d, y-d)	Pour some water out of the 3-liter jug.
5	(x, y) if x>0 -> (0, y)	Empty the 4-liter jug on the ground
6	(x, y) if y>0 -> (x,0)	Empty the 3-liter jug on the ground
7	(x, y) if X+Y >= 4 and y>0 -> (4, y-(4-x))	Pour water from the 3-liter jug into the 4-liter jug until the 4-liter jug is full
8	(x, y) if X+Y>=3 and x>0 -> (x-(3-y), 3))	Pour water from the 4-liter jug into the 3-liter jug until the 3-liter jug is full.
9	(x, y) if X+Y <=4 and y>0 -> (x+y, 0)	Pour all the water from the 3-liter jug into the 4-liter jug.
10	(x, y) if X+Y<=3 and x>0 -> (0, x+y)	Pour all the water from the 4-liter jug into the 3-liter jug.
11	(0, 2) -> (2, 0)	Pour the 2-liter water from the 3-liter jug into the 4-liter jug.
12	(2, Y) -> (0, y)	Empty the 2-liter in the 4-liter jug on the ground.

The solution to Water Jug Problem in Artificial Intelligence

1. Current state = (0, 0)

2. Loop until the goal state (2, 0) reached

– Apply a rule whose left side matches the current state

– Set the new current state to be the resulting state

(0, 0) – Start State

(0, 3) – Rule 2, Fill the 3-liter jug

(3, 0) – Rule 9, Pour all the water from the 3-liter jug into the 4-liter jug.

(3, 3) – Rule 2, Fill the 3-liter jug

(4, 2) – Rule 7, Pour water from the 3-liter jug into the 4-liter jug until the 4-liter jug is full.

(0, 2) – Rule 5, Empty the 4-liter jug on the ground

- Production systems provide appropriate structures for performing and describing search processes.
- A production system has four basic components: A set of rules each consisting of a left side that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.
- A database of current facts established during the process of inference.
- A control strategy that specifies the order in which the rules will be compared with facts in the database and also specifies how to resolve conflicts in selection of several rules or selection of more facts.
- A rule firing module.
- The production rules operate on the knowledge database.
- Each rule has a precondition—that is, either satisfied or not by the knowledge database. If the precondition is satisfied, the rule can be applied.

SEARCH CREATORS

4. List all task domains of Artificial Intelligence.?

Ans: - Refer Question No. 01

5. Explain Minimax procedure of tic — tac — toe.?

Ans: -

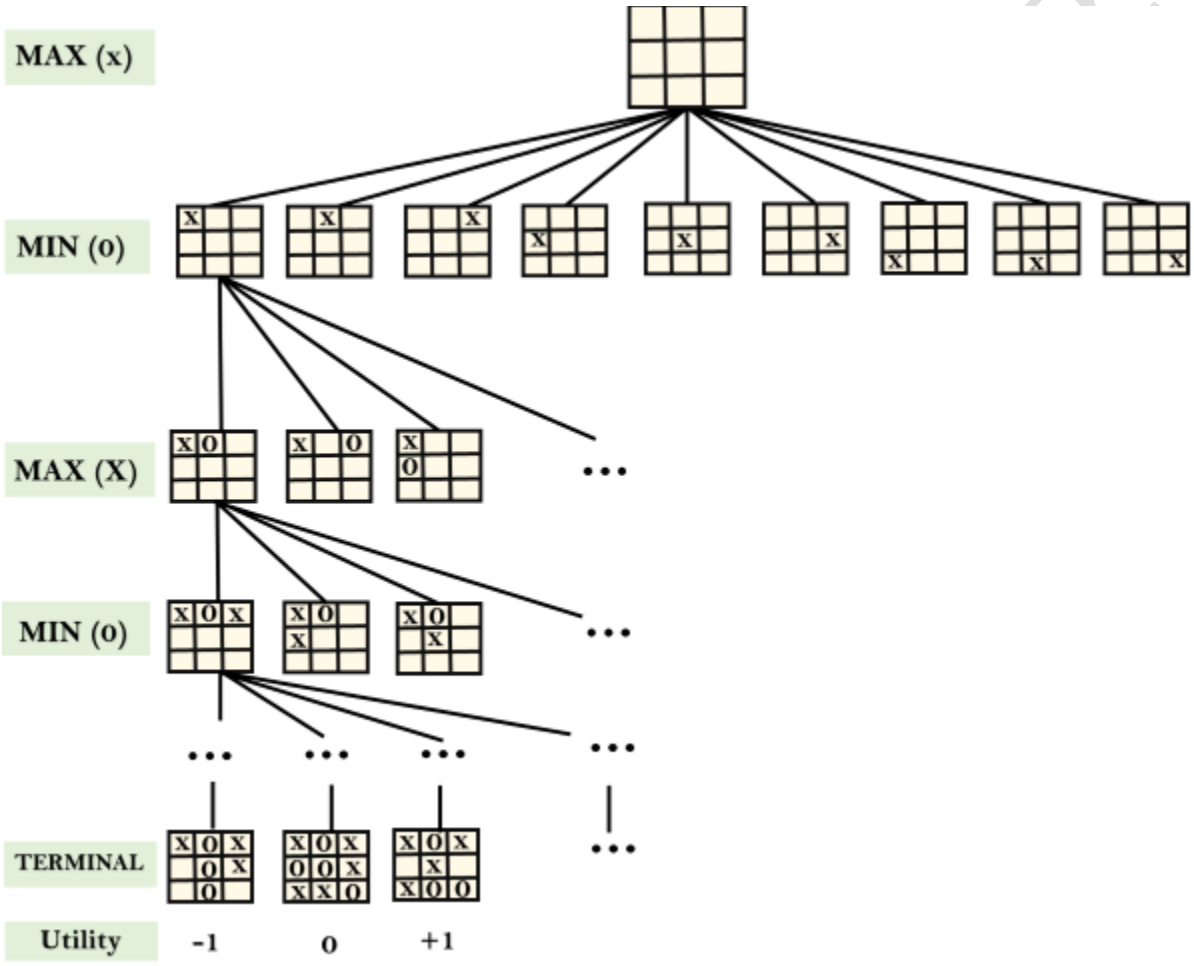


Fig: - Tic – Tac -Toe Game

The Above figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:

- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.

Explanation:

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.
- So, in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as Ply. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

6. List all production rules for the water jug problem.?

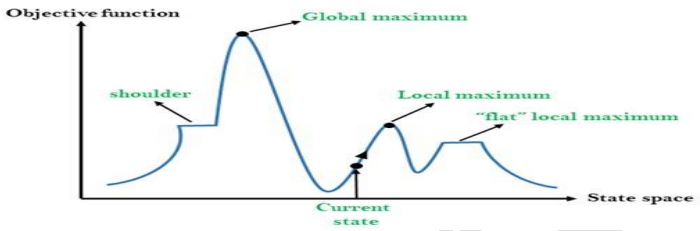
Ans: - Refer Question No. 03

SEARCH CREATORS

7. Explain Hill climbing issues which terminates algorithm without finding a goal state or getting to state from which no better state can be generated?

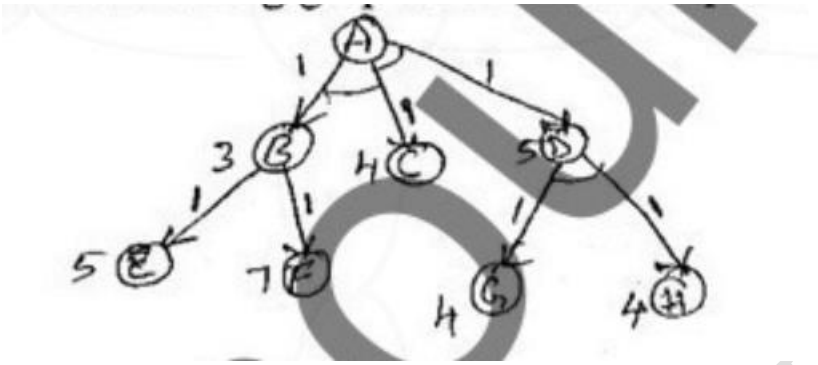
Ans: -

- Hill Climbing is heuristic search used for mathematical optimization problems in the field of Artificial Intelligence.



1. **Local maximum**
   - It is a state which is better than its neighboring state however there exists a state which is better than it (global maximum).
   - This state is better because here value of objective function is higher than its neighbors.
2. **Global maximum**
   - It is the best possible state in the state space diagram. This because at this state, objective function has highest value.
3. **Plateau/flat local maximum**
   - It is a flat region of state space where neighboring states have the same value.
4. **Ridge**
   - It is region which is higher than its neighbors but itself has a slope. It is a special kind of local maximum.
5. **Current state**
   - The region of state space diagram where we are currently present during the search.
6. **Shoulder**
   - It is a plateau that has an uphill edge.
   - It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as next node.

8. Apply AO* algorithm for the following graph and find final path?



Ans: -

Algorithm:

Step 1: Place the starting node into OPEN.

Step 2: Compute the most promising solution tree say T0.

Step 3: Select a node n that is both on OPEN and a member of T0. Remove it from OPEN and place it in

CLOSE

Step 4: If n is the terminal goal node, then leveled n as solved and leveled all the ancestors of n as solved. If the starting node is marked as solved then success and exit.

Step 5: If n is not a solvable node, then mark n as unsolvable. If starting node is marked as unsolvable, then return failure and exit.

Step 6: Expand n. Find all its successors and find their h (n) value, push them into OPEN.

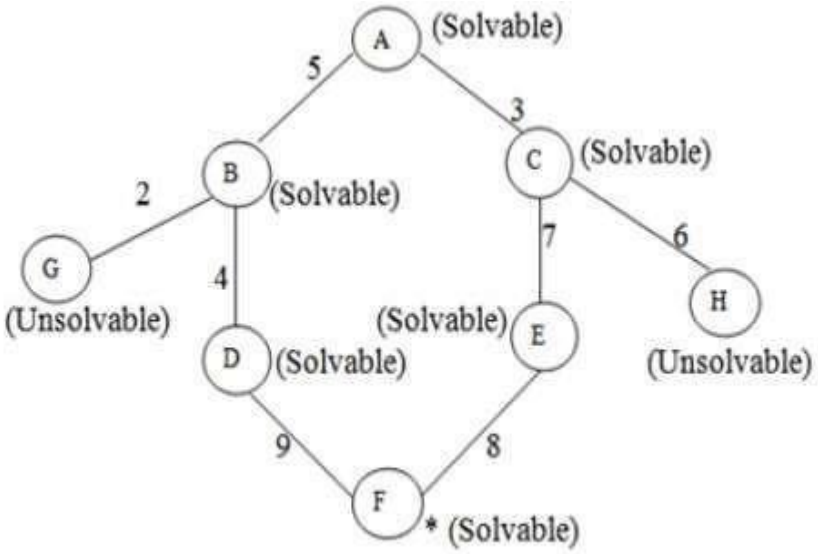
Step 7: Return to Step 2.

Step 8: Exit.

Implementation:

SEARCH CREATORS

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



Figure

Step 1:

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

i.e. OPEN = [A] CLOSE = (NULL) [ϕ] (A)

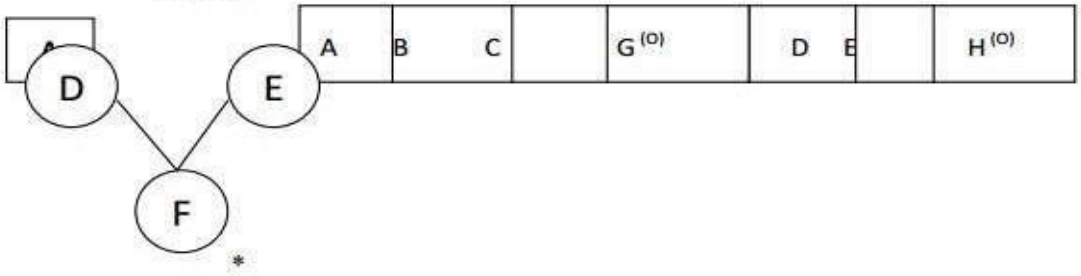
Step 2:

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

Step 4:

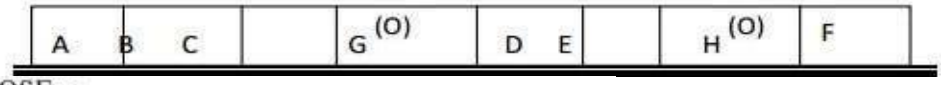
As the nodes G and H are unsolvable, so place them into CLOSE directly and process the nodes D and E.

i.e. OPEN = CLOSE =



Step 5:

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

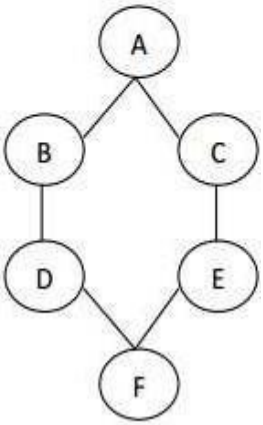


i.e. CLOSE =

Step 6:

Success and Exit

AO* Graph:



Figure

9. What is Artificial Intelligence? Discuss the branches of Artificial Intelligence?

Ans: - Refer Question No. 01

Branches of AI:

1. **Logical AI**
   - In general, the facts of the specific situation in which it must act, and its goals are all represented by sentences of some mathematical logical language.
   - The program decides what to do by inferring that certain actions are appropriate for achieving its goals.
2. **Search**
   - Artificial Intelligence programs often examine large numbers of possibilities for example, moves in a chess game and inferences by a theorem proving program.
   - Discoveries are frequently made about how to do this more efficiently in various domains.
3. **Pattern Recognition**
   - When a program makes observations of some kind, it is often planned to compare what it sees with a pattern.
   - For example, a vision program may try to match a pattern of eyes and a nose in a scene in order to find a face.
   - More complex patterns are like a natural language text, a chess position or in the history of some event.
4. **Representation**
   - Usually, languages of mathematical logic are used to represent the facts about the world.
5. **Inference**
   - Others can be inferred from some facts.
   - Mathematical logical deduction is sufficient for some purposes, but new methods of *non-monotonic* inference have been added to the logic since the 1970s.
   - The simplest kind of non-monotonic reasoning is default reasoning in which a conclusion is to be inferred by default.

SEARCH CREATORS

6. **Common sense knowledge and Reasoning**
    - This is the area in which AI is farthest from the human level, in spite of the fact that it has been an active research area since the 1950s.

7. **Learning from experience**
    - Programs can only learn what facts or behavior their formalisms can represent, and unfortunately learning systems are almost all based on very limited abilities to represent information.

8. **Planning**
    - Planning starts with general facts about the world (especially facts about the effects of actions), facts about the particular situation and a statement of a goal. From these, planning programs generate a strategy for achieving the goal.

9. **Epistemology**
    - This is a study of the kinds of knowledge that are required for solving problems in the world.

10. **Ontology**
    - Ontology is the study of the kinds of things that exist.
    - In AI the programs and sentences deal with various kinds of objects and we study what these kinds are and what their basic properties are.
    - Ontology assumed importance from the 1990s.

11. **Heuristics**
    - A heuristic is a way of trying to discover something or an idea embedded in a program. The term is used variously in AI.
    - *Heuristic functions* are used in some approaches to search or to measure how far a node in a search tree seems to be from a goal.

12. **Genetic programming**
    - Genetic programming is an automated method for creating a working computer program from a high-level problem statement of a problem.
    - Genetic programming starts from a high-level statement of 'what needs to be done' and automatically creates a computer program to solve the problem.

SEARCH CREATORS

10. Explain any two Al techniques for solving tie-tac-toe problem.?

Ans: -

The first approach (simple)

The Tic-Tac-Toe game consists of a nine-element vector called BOARD; it represents the numbers 1 to 9 in three rows.

1	2	3
4	5	6
7	8	9

An element contains the value 0 for blank, 1 for X and 2 for O. A MOVETABLE vector consists of 19,683 elements (3^9) and is needed where each element is a nine-element vector. The contents of the vector are especially chosen to help the algorithm.

The algorithm makes moves by pursuing the following:

1. View the vector as a ternary number. Convert it to a decimal number.
2. Use the decimal number as an index in MOVETABLE and access the vector.
3. Set BOARD to this vector indicating how the board looks after the move. This approach is capable in time but it has several disadvantages.
4. It takes more space and requires stunning effort to calculate the decimal numbers. This method is specific to this game and cannot be completed.

1.2.2 **The second approach**

- The structure of the data is as before but we use 2 for a blank, 3 for an X and 5 for an O.
- A variable called TURN indicates 1 for the first move and 9 for the last.

  The algorithm consists of three actions:

  - MAKE2 which returns 5 if the center square is blank; otherwise, it returns any blank non- corner square, i.e. 2, 4, 6 or 8.

SEARCH CREATORS

- POSSWIN (p) returns 0 if player p cannot win on the next move and otherwise returns the number of the square that gives a winning move.
- It checks each line using products 3*3*2 = 18 gives a win for X, 5*5*2=50 gives a win for O, and the winning move is the holder of the blank.
- GO (n) makes a move to square n setting BOARD[n] to 3 or 5.
- This algorithm is more involved and takes longer but it is more efficient in storage which compensates for its longer time.
- It depends on the programmer's skill.

11. Write the algorithms for breadth first search and depth-first search. Enlist the advantages of each?

Ans: -

Algorithm BFS: Breadth First Search

- Create a variable called NODE_LIST and set it to the initial state.
- Until a goal state is found or NODE_LIST is empty:
- Remove the first element from NODE_LIST and call it E. If NODE_LIST was empty, quit.
- For each way that each rule can match the state described in E do:
- Apply the rule to generate a new state.
- If the new state is a goal state, quit and return this state.
- Otherwise, add the new state to the end of NODE_LIST.

Algorithm DFS: Depth First Search

- If the initial state is a goal state, quit and return success.
- Otherwise, do the following until success or failure is signaled.
- Generate successor, E of the initial state. If there are no more successors, signal failure.
- Call Depth-First Search with E as the initial state.
- If success is returned, signal success. Otherwise continue in this loop.

Advantages of BFS:

SEARCH CREATORS

3. Used to find the shortest path between states.
4. Always finds optimal solutions.
5. There is nothing like useless path in BFS, since it searches level by level.
6. Finds the closest goal state in less time.

Disadvantages of BFS:

All of the connected vertices must be stored in memory. So consumes more memory

Advantages of DFS:

1. Consumes less memory
2. Finds the larger distant element (from initial state) in less time.

Disadvantages of DFS:

1. May not find optimal solution to the problem.
2. May get trapped in searching useless path.

SEARCH CREATORS

Additional Topic from The Syllabus -2021

Chapter-01: - Foundations and History of AI

THE F OUNDATIONS OF A RTIFICIAL I NTELLIGENCE

Philosophy

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

Mathematics

- What are the formal rules to draw valid conclusions?
- What can be computed?
- How do we reason with uncertain information?

History of AI

The gestation of artificial intelligence (1943–1955)

- The first work that is now generally recognized as AI was done by Warren McCulloch and Walter Pitts (1943).
- They drew on three sources: knowledge of the basic physiology and function of neurons in the brain; a formal analysis of propositional logic due to Russell and Whitehead; and Turing's theory of computation.

The birth of artificial intelligence (1956)

- Princeton was home to another influential figure in AI, John McCarthy.
- After receiving his PhD there in 1951 and working for two years as an instructor, McCarthy moved to Stanford and then to Dartmouth College, which was to become the official birthplace of the field.

Early enthusiasm, great expectations (1952–1969)

SEARCH CREATORS

- The early years of AI were full of successes in a limited way.
- Given the primitive computers and programming tools of the time and the fact that only a few years earlier computers were seen as things that could do arithmetic and no more, it was astonishing whenever a computer did anything remotely clever.

A dose of reality (1966–1973)

Knowledge-based systems: The key to power? (1969–1979)

AI becomes an industry (1980–present)

The return of neural networks (1986–present)

AI adopts the scientific method (1987–present)

The emergence of intelligent agents (1995–present)

The availability of very large data sets (2001–present)

SEARCH CREATORS

Chapter-02: - Problem-solving agents

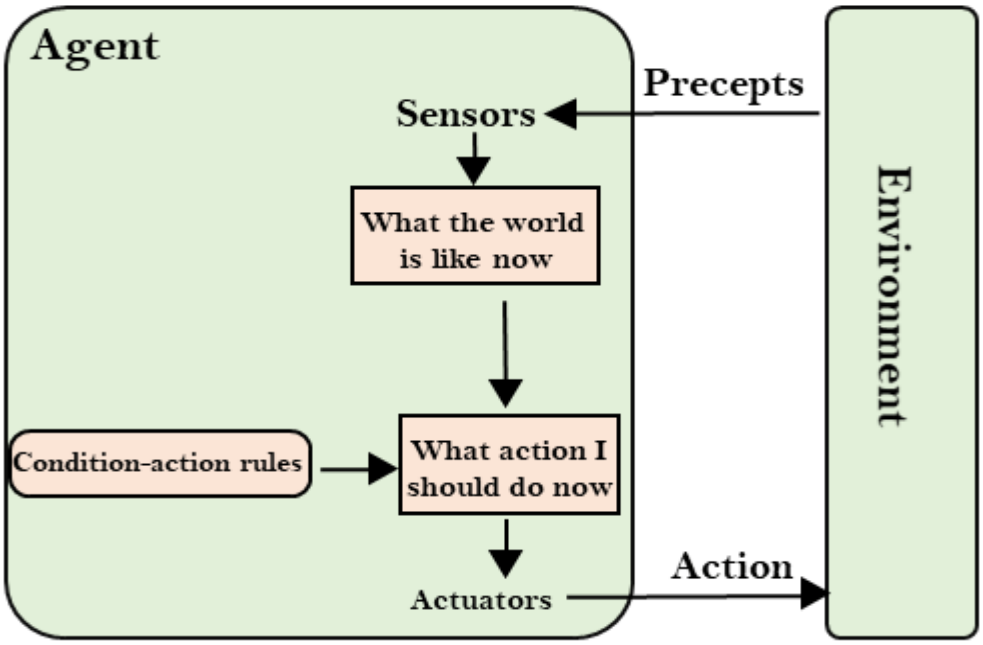
- Agents can be grouped into five classes based on their degree of perceived intelligence and capability.
- All these agents can improve their performance and generate better action over the time. These are given below:

1) Simple Reflex Agent

- The Simple reflex agents are the simplest agents. These agents take decisions on the basis of the current precepts and ignore the rest of the percept history.

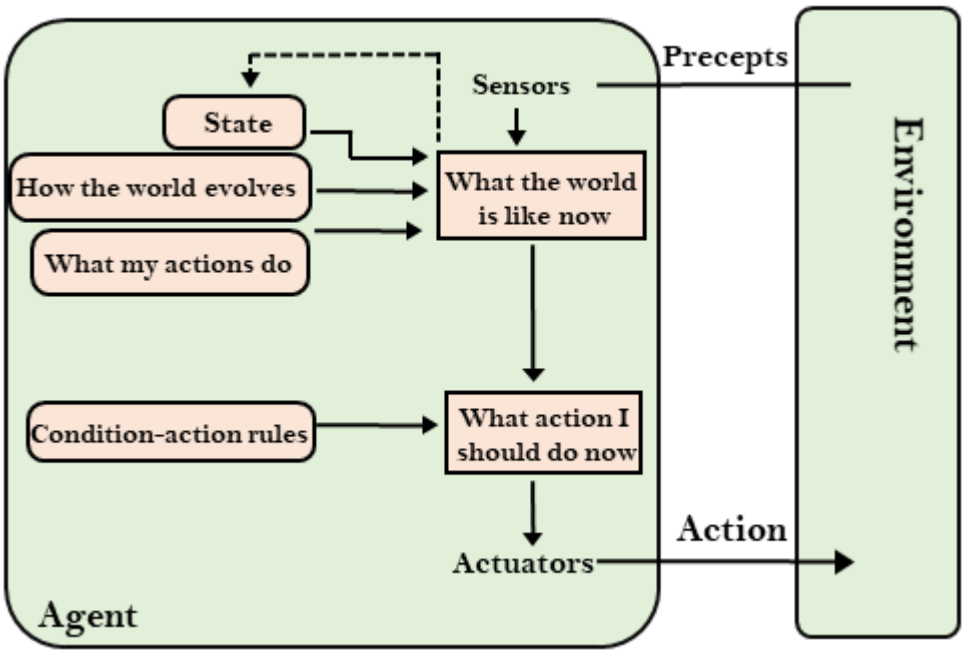
- These agents only succeed in the fully observable environment.

- The Simple reflex agent does not consider any part of precepts history during their decision and action process.



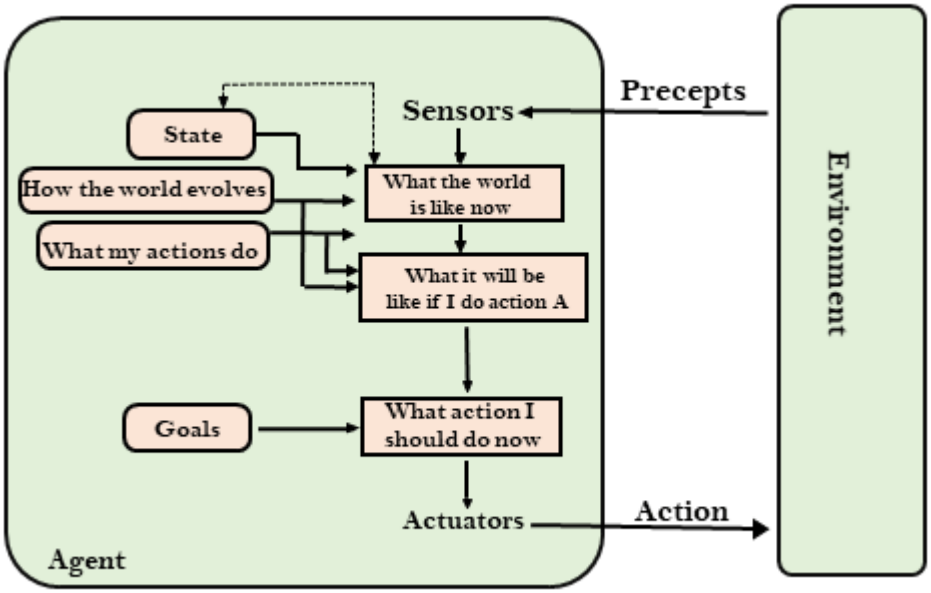
- **Model-based reflex agent**

- The Model-based agent can work in a partially observable environment, and track the situation.
- A model-based agent has two important factors:
- Model: It is knowledge about "how things happen in the world," so it is called a Model-based agent.
- Internal State: It is a representation of the current state based on percept history.
- These agents have the model, "which is knowledge of the world" and based on the model they perform actions.
- Updating the agent state requires information about:
- How the world evolves
- How the agent's action affects the world.



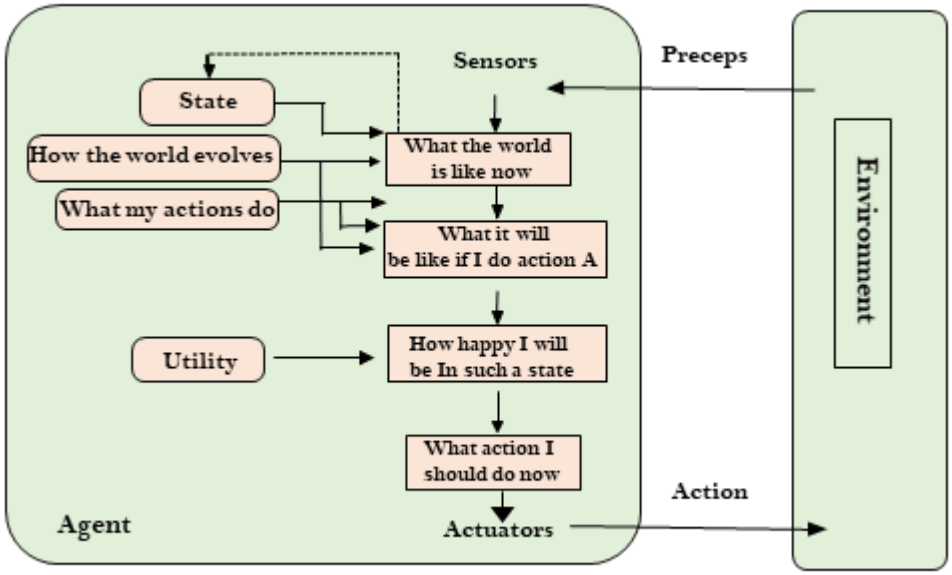
2) Goal-based agents

- The knowledge of the current state environment is not always sufficient to decide for an agent to what to do.
- The agent needs to know its goal which describes desirable situations.
- Goal-based agents expand the capabilities of the model-based agent by having the "goal" information.
- They choose an action, so that they can achieve the goal.
- These agents may have to consider a long sequence of possible actions before deciding whether the goal is achieved or not. Such considerations of different scenario are called searching and planning, which makes an agent proactive.



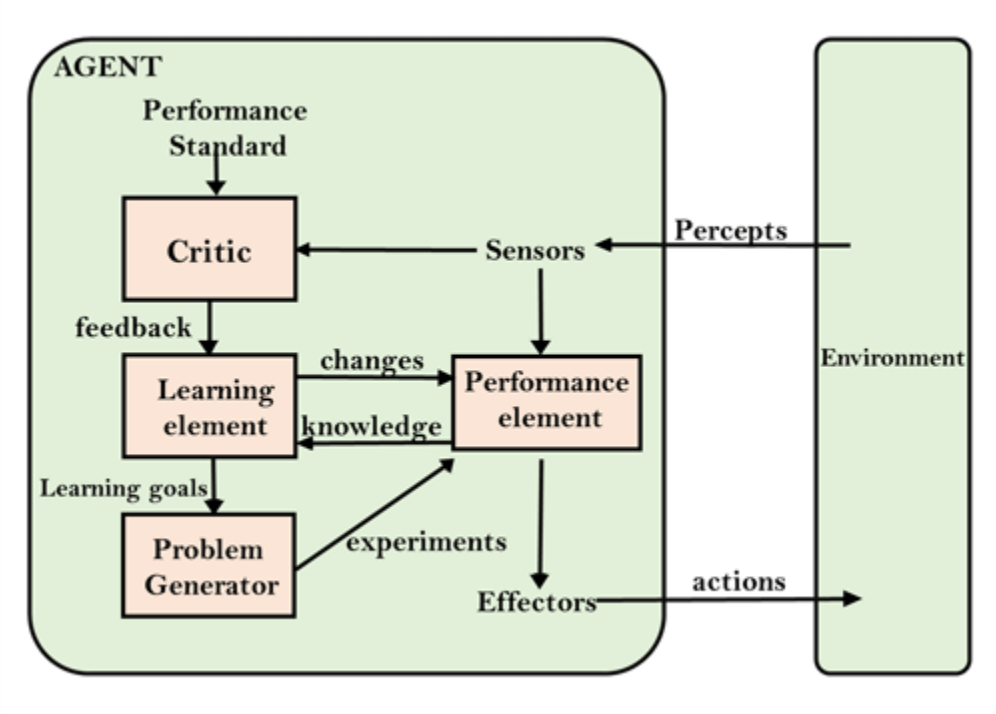
3) Utility-based agent

- These agents are similar to the goal-based agent but provide an extra component of utility measurement which makes them different by providing a measure of success at a given state.
- Utility-based agent act based not only goals but also the best way to achieve the goal.
- The Utility-based agent is useful when there are multiple possible alternatives, and an agent has to choose in order to perform the best action.
- The utility function maps each state to a real number to check how efficiently each action achieves the goals.



4) Learning agent

- A learning agent in AI is the type of agent which can learn from its past experiences, or it has learning capabilities.
- It starts to act with basic knowledge and then able to act and adapt automatically through learning.
- A learning agent has mainly four conceptual components, which are:
    - **Learning element:** It is responsible for making improvements by learning from environment
    - **Critic:** Learning element takes feedback from critic which describes that how well the agent is doing with respect to a fixed performance standard.
    - **Performance element:** It is responsible for selecting external action
    - **Problem generator:** This component is responsible for suggesting actions that will lead to new and informative experiences.



Well-defined problems and solutions

A problem can be defined formally by five components:

• The initial state that the agent starts in. For example, the initial state for our agent in Romania might be described as in (Arad).

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

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

Figure: - A simple problem-solving agent.

- A description of the possible actions available to the agent. Given a particular state, ACTIONS return the set of actions that can be executed in s.
- A description of what each action does; the formal name for this is the transition model, specified by a function R ESULT (s, a) that returns the state that results from TRANSITION MODEL doing action an in states.
- Together, the initial state, actions, and transition model implicitly define the state space STATE SPACE of the problem—the set of all states reachable from the initial state by any sequence of actions.

  RESULT (*In* (*Arad*), *Go* (*Zerind*)) = *In* (*Zerind*).
- Together, the initial state, actions, and transition model implicitly define the state space of the problem—the set of all states reachable from the initial state by any sequence of actions.

- The state space forms a directed network or graph in which the nodes are states and the links between nodes are actions. (The map of Romania shown in Figure can be interpreted as a state-space graph if we view each road as standing for two driving actions, one in each

SEARCH CRL

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

- The goal test, which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them.
- The agent's goal in Romania is the singleton set {In (Bucharest)}.

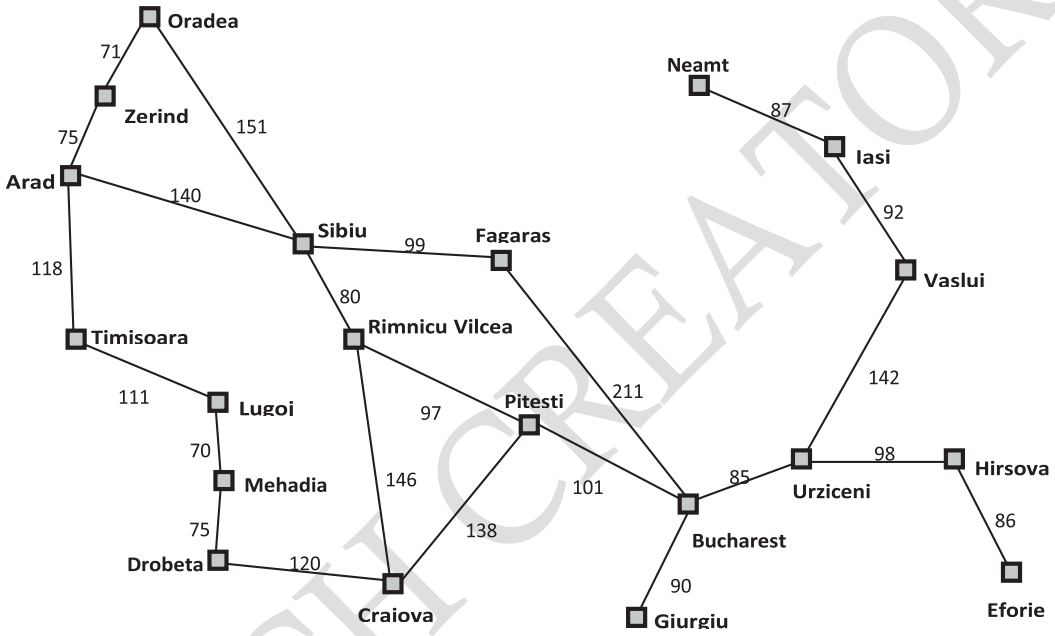


Figure: - A simplified road map of part of Romania.

- A **path cost** function that assigns a numeric cost to each path.
- The problem-solving agent chooses a cost function that reflects its own performance measure.
- For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers.

Formulating problems

- we proposed a formulation of the problem of getting to Bucharest in terms of the initial state, actions, transition model, goal test, and path cost. This formulation seems reasonable, but it is still a model—an abstract mathematical description—and not the real thing.

SEARCH CREATORS

- All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest.
- The process of removing detail from a representation is called abstraction.
- Can we be more precise about defining the appropriate level of abstraction? Think of the abstract states and actions we have chosen as corresponding to large sets of detailed world states and detailed action sequences.

EXAMPLE PROBLEMS

- The problem-solving approach has been applied to a vast array of task environments.
- We list some of the best known here, distinguishing between **toy and real-world problems.**
- A **toy problem** is intended to illustrate or exercise various problem-solving methods.
- A **real-world problem** is one whose solutions people actually care about. Such problems tend not to have a single agreed-upon description, but we can give the general flavor of their formulations.

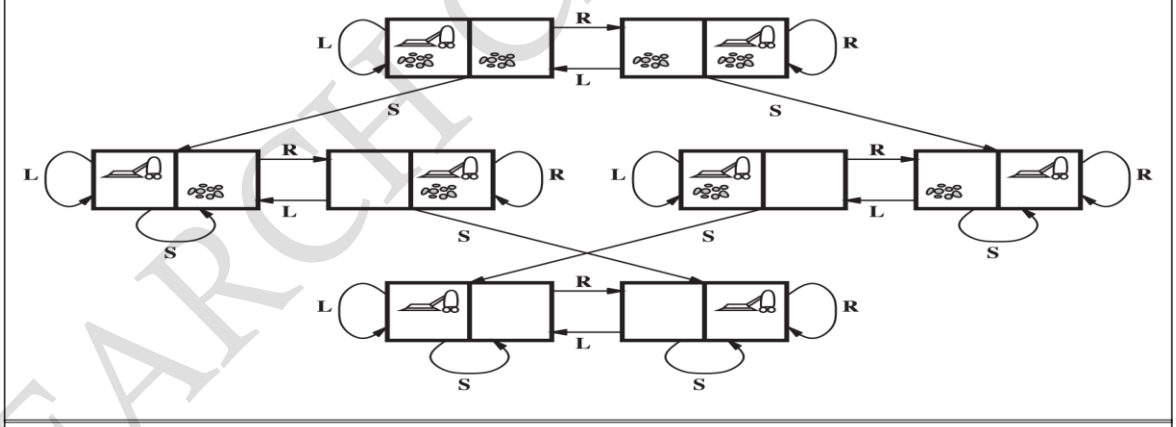


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Toy problems

This can be formulated as a problem as follows:

- **States:** The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are 2 × 22 = 8 possible world states. A larger environment with n locations has n · 2n 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 Sucking in a clean square have no effect. The complete state space is shown in Figure
- **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

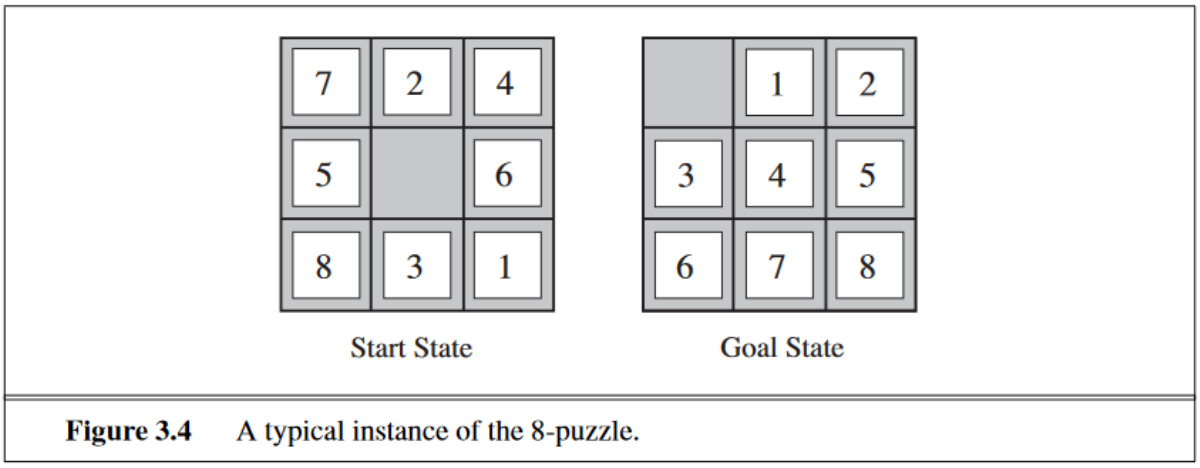


Figure 3.4 A typical instance of the 8-puzzle.

- **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 (Other goal configurations are possible.)
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path. What abstractions have we included here? The actions are abstracted to their beginning and final states, ignoring the intermediate locations where the block is sliding.

SEARCH CREATORS

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

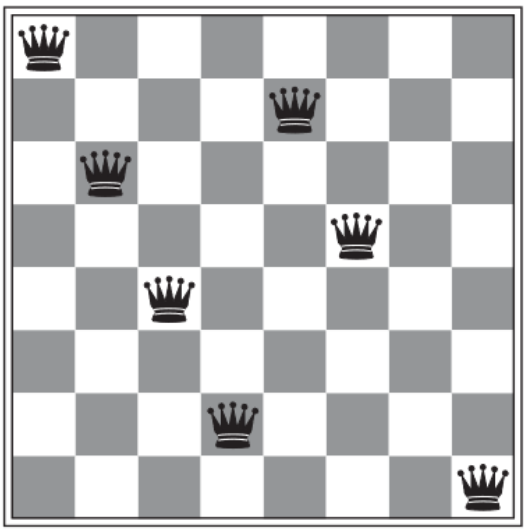


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

There are two main kinds of formulation.

- An incremental formulation involves operators that augment the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state.
- A complete-state formulation starts with all 8 queens on the board and moves them around. In either case, the path cost is of no interest because only the final state counts. The first incremental formulation one might try is the following:
- **States:** Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state:** No queens on the board.
- **Actions:** Add a queen to any empty square.
- **Transition model:** Returns the board with a queen added to the specified square.
- **Goal test:** 8 queens are on the board, none attacked.

SEARCH CRE

- Our final toy problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise.

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

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

The problem definition is very simple:

- States: Positive numbers.
- Initial state: 4.
- Actions: Apply factorial, square root, or floor operation (factorial for integers only).
- Transition model: As given by the mathematical definitions of the operations.
- Goal test: State is the desired positive integer

Real-world problems

- Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example.

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

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

SEARCH CREATORS

- **Path cost:** This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

- **Touring problems** are closely related to route-finding problems, but with an important difference. Consider, for example, the problem "Visit every city in Figure at least once, starting and ending in Bucharest."
- So, the initial state would be *in* (*Bucharest*), *Visited*({*Bucharest*}), a typical intermediate state would be *in* (*Vaslui*), *Visited* ({*Bucharest*, *Urziceni*, *Vaslui*}), and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

- **The traveling salesperson problem (TSP)** is a touring problem in which each city must be visited exactly once.
- The aim is to find the shortest tour.
- The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms.
- In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

- **A VLSI layout problem** requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield.
- The layout problem comes after the logical design phase and is usually split into two parts: cell layout and channel routing. In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells.
- The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells.

SEARCH CREATORS

- Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

- **Robot navigation** is a generalization of the route-finding problem described earlier.
- Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.
- For a circular robot moving on a flat surface, the space is essentially two-dimensional.

- **Automatic assembly** sequencing of complex objects by a robot was first demonstrated by FREDDY (Michie, 1972).
- Progress since then has been slow but sure, to the point where the assembly of intricate objects such as electric motors is economically feasible.
- In assembly problems, the aim is to find an order in which to assemble the parts of some object.

- **protein design,** in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

SEARCH CREATORS

SEARCHING FOR SOLUTIONS

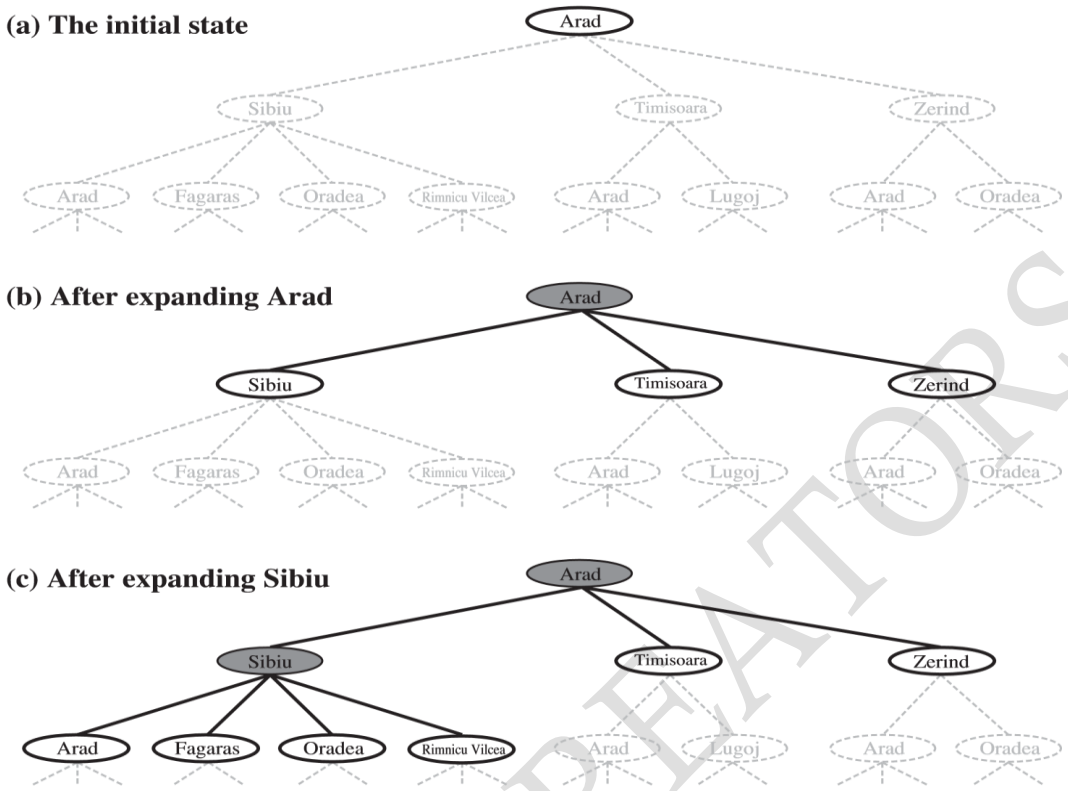


Figure: - Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

SEARCH CREATORS

```
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.

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

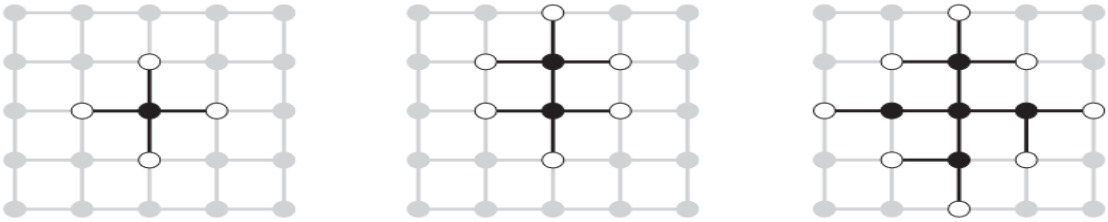


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been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

Infrastructure for search algorithms

- 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.

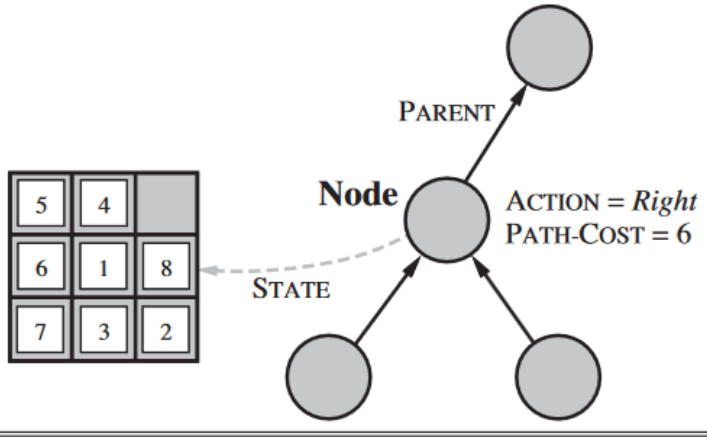


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

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

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?

UNINFORMED SEARCH STRATEGIES

- uninformed search (also called blind search). The term means that the strategies have no additional information about states beyond that provided in the problem definition.
- All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the order in which nodes are expanded.

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.

```
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.

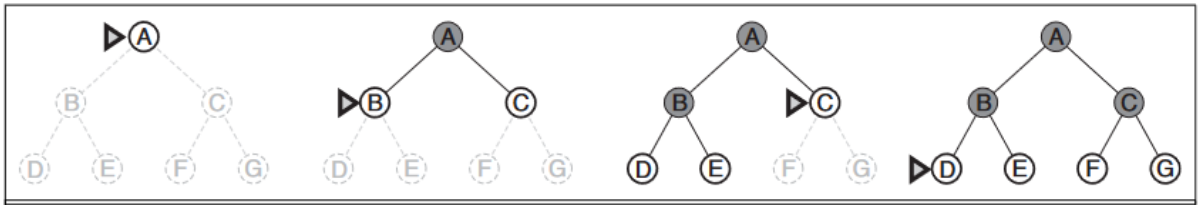


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Uniform-cost search

- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node.
- By a simple extension, we can find an algorithm that is optimal UNIFORM-COSTSEARCH with any step-cost function.
- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost g(n).

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

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

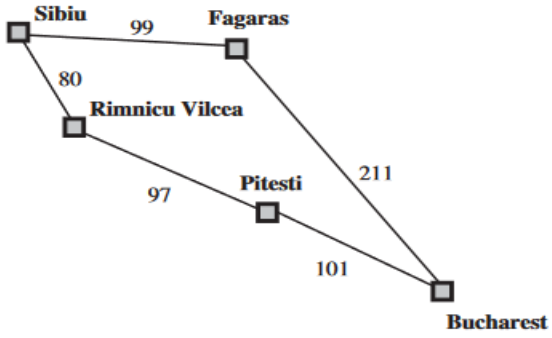


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

Depth-first search

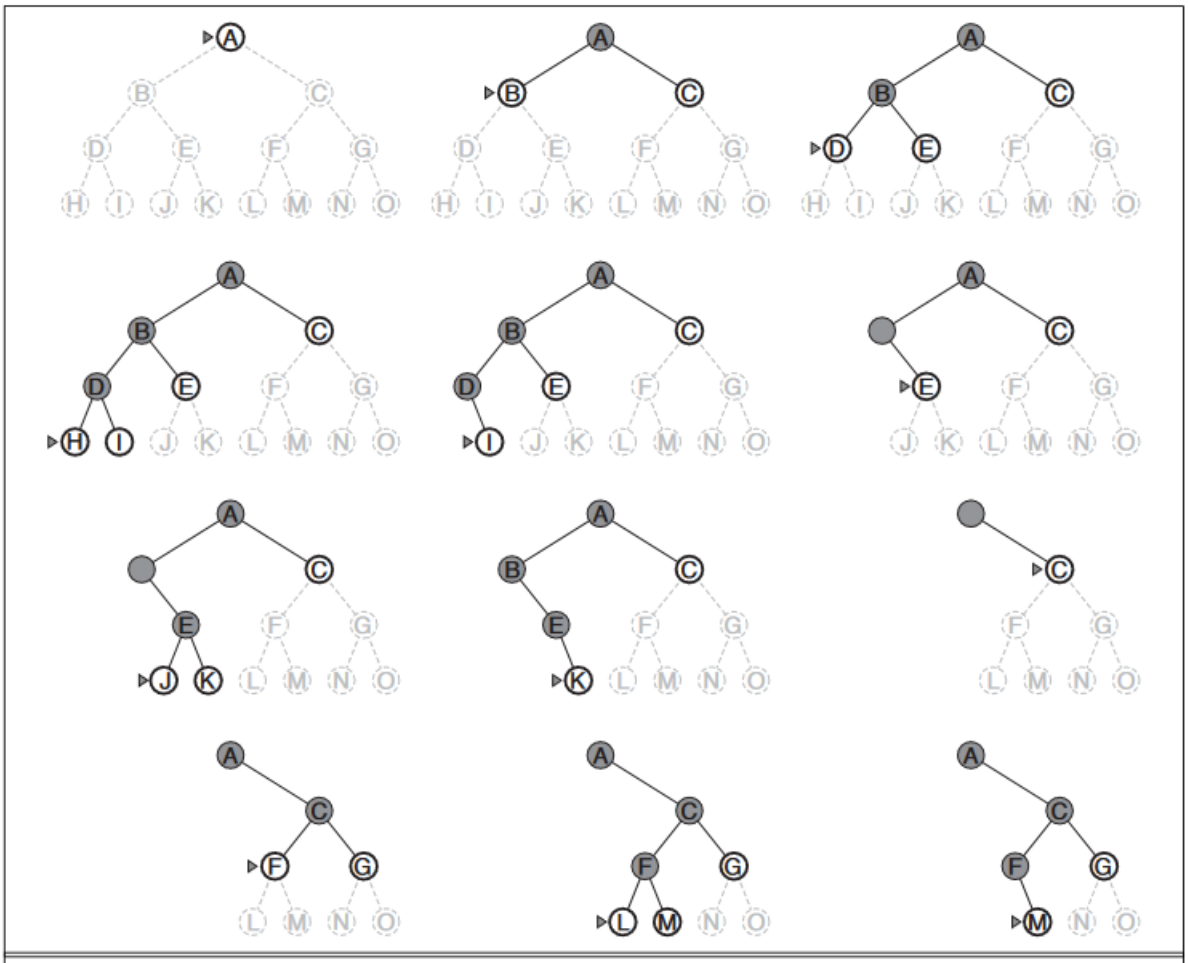


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Depth-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.**