

# UNIT- I

**Introduction:** AI problems, Agents and Environments, Structure of Agents, Problem Solving Agents **Basic Search Strategies:** Problem Spaces, Uninformed Search (Breadth First, Depth-First Search, Depth-first with Iterative Deepening), Heuristic Search (Hill Climbing, Generic Best-First, A\*), Constraint Satisfaction (Backtracking, Local Search)

## Introduction:

- Artificial Intelligence is concerned with the design of intelligence in an artificial device. The term was coined by John McCarthy in 1956.
- Intelligence is the ability to acquire, understand and apply the knowledge to achieve goals in the world.
- AI is the study of the mental faculties through the use of computational models
- AI is the study of intellectual/mental processes as computational processes.
- AI program will demonstrate a high level of intelligence to a degree that equals or exceeds the intelligence required of a human in performing some task.
- AI is unique, sharing borders with Mathematics, Computer Science, Philosophy, Psychology, Biology, Cognitive Science and many others.
- Although there is no clear definition of AI or even Intelligence, it can be described as an attempt to build machines that like humans can think and act, able to learn and use knowledge to solve problems on their own.

## Sub Areas of AI:

### 1) Game Playing

Deep Blue Chess program beat world champion Gary Kasparov

### 2) Speech Recognition

PEGASUS spoken language interface to American Airlines' EASY SABRE reservation system, which allows users to obtain flight information and make reservations over the

telephone. The 1990s has seen significant advances in speech recognition so that limited systems are now successful.

### 3) **Computer Vision**

Face recognition programs in use by banks, government, etc. The ALVINN system from CMU autonomously drove a van from Washington, D.C. to San Diego (all but 52 of 2,849 miles), averaging 63 mph day and night, and in all weather conditions. Handwriting recognition, electronics and manufacturing inspection, photo interpretation, baggage inspection, reverse engineering to automatically construct a 3D geometric model.

### 4) **Expert Systems**

Application-specific systems that rely on obtaining the knowledge of human experts in an area and programming that knowledge into a system.

- a. **Diagnostic Systems:** MYCIN system for diagnosing bacterial infections of the blood and suggesting treatments. Intellipath pathology diagnosis system (AMA approved). Pathfinder medical diagnosis system, which suggests tests and makes diagnoses. Whirlpool customer assistance center.
- b. **System Configuration**  
DEC's XCON system for custom hardware configuration. Radiotherapy treatment planning.
- c. **Financial Decision Making**  
Credit card companies, mortgage companies, banks, and the U.S. government employ AI systems to detect fraud and expedite financial transactions. For example, AMEX credit check.
- d. **Classification Systems**  
Put information into one of a fixed set of categories using several sources of information. E.g., financial decision making systems. NASA developed a system for classifying very faint areas in astronomical images into either stars or galaxies with very high accuracy by learning from human experts' classifications.

### 5) **Mathematical Theorem Proving**

Use inference methods to prove new theorems.

### 6) **Natural Language Understanding**

AltaVista's translation of web pages. Translation of Caterpillar Truck manuals into 20 languages.

## **7) Scheduling and Planning**

Automatic scheduling for manufacturing. DARPA's DART system used in Desert Storm and Desert Shield operations to plan logistics of people and supplies. American Airlines rerouting contingency planner. European space agency planning and scheduling of spacecraft assembly, integration and verification.

## **8) Artificial Neural Networks:**

## **9) Machine Learning**

### **Applications of AI:**

AI algorithms have attracted close attention of researchers and have also been applied successfully to solve problems in engineering. Nevertheless, for large and complex problems, AI algorithms consume considerable computation time due to stochastic feature of the search approaches

1. Business; financial strategies
2. Engineering: check design, offer suggestions to create new product, expert systems for all engineering problems
3. Manufacturing: assembly, inspection and maintenance
4. Medicine: monitoring, diagnosing
5. Education: in teaching
6. Fraud detection
7. Object identification
8. Information retrieval
9. Space shuttle scheduling

### **Building AI Systems:**

#### **1) Perception**

Intelligent biological systems are physically embodied in the world and experience the world through their sensors (senses). For an autonomous vehicle, input might be images from a camera and range information from a rangefinder. For a medical diagnosis

system, perception is the set of symptoms and test results that have been obtained and input to the system manually.

## 2) Reasoning

Inference, decision-making, classification from what is sensed and what the internal "model" is of the world. Might be a neural network, logical deduction system, Hidden Markov Model induction, heuristic searching a problem space, Bayes Network inference, genetic algorithms, etc. Includes areas of knowledge representation, problem solving, decision theory, planning, game theory, machine learning, uncertainty reasoning, etc.

## 3) Action

Biological systems interact within their environment by actuation, speech, etc. All behavior is centered around actions in the world. Examples include controlling the steering of a Mars rover or autonomous vehicle, or suggesting tests and making diagnoses for a medical diagnosis system. Includes areas of robot actuation, natural language generation, and speech synthesis.

### The definitions of AI:

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

<p>c) "The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)</p> <p>"The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)</p>	<p>d) "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990)</p> <p>"The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)</p>
--	--

The definitions on the top, **(a)** and **(b)** are concerned with **reasoning**, whereas those on the bottom, **(c)** and **(d)** address **behavior**. The definitions on the left, **(a)** and **(c)** measure success in terms of human performance, and those on the right, **(b)** and **(d)** measure the ideal concept of intelligence called rationality

**Intelligent Systems:**

In order to design intelligent systems, it is important to categorize them into four categories (Luger and Stubblefield 1993), (Russell and Norvig, 2003)

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

	Human-Like	Rationality
Think:	<p><b>Cognitive Science Approach</b></p> <p><i>"Machines that think like humans"</i></p>	<p><b>Laws of thought Approach</b></p> <p><i>"Machines that think Rationally"</i></p>
Act:	<p><b>Turing Test Approach</b></p> <p><i>"Machines that behave like humans"</i></p>	<p><b>Rational Agent Approach</b></p> <p><i>"Machines that behave Rationally"</i></p>

### **Cognitive Science: Think Human-Like**

- a. Requires a model for human cognition. Precise enough models allowsimulation by computers.
- b. Focus is not just on behavior and I/O, but looks like reasoning process.
- c. Goal is not just to produce human-like behavior but to produce a sequence of steps of thereasoning process, similar to the steps followed by a human in solving the same task.

### **Laws of thought: Think Rationally**

- a. The study of mental faculties through the use of computational models; that it is, thestudy of computations that make it possible to perceive reason and act.
- b. Focus is on inference mechanisms that are probably correct and guarantee an optimal solution.
- c. Goal is to formalize the reasoning process as a system of logical rules and procedures ofinference.
- d. Develop systems of representation to allow inferences to be like

*“Socrates is a man. All men are mortal. Therefore Socrates is mortal”*

### **Turing Test: Act Human-Like**

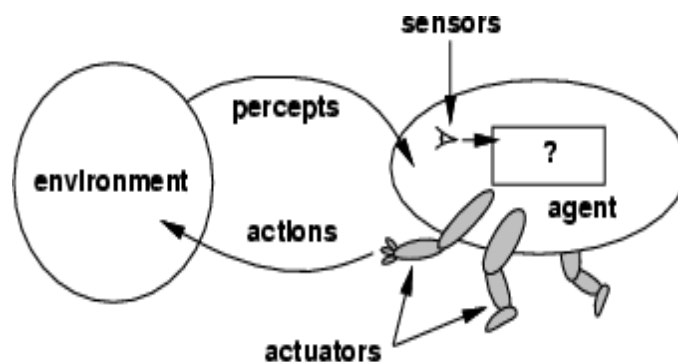
- a. The art of creating machines that perform functions requiring intelligence when performed by people; that it is the study of, how to make computers do things which, at the moment, people do better.
- b. Focus is on action, and not intelligent behavior centered around the representation of the world
- c. Example: Turing Test
  - 3 rooms contain: a person, a computer and an interrogator.
  - The interrogator can communicate with the other 2 by teletype (to avoidthe machine imitate the appearance of voice of the person)
  - The interrogator tries to determine which the person is and which themachine is.

- The machine tries to fool the interrogator to believe that it is the human, and the person also tries to convince the interrogator that it is the human.
- If the machine succeeds in fooling the interrogator, then conclude that the machine is intelligent.

### Rational agent: Act Rationally

- Tries to explain and emulate intelligent behavior in terms of computational process; that it is concerned with the automation of the intelligence.
- Focus is on systems that act sufficiently if not optimally in all situations.
- Goal is to develop systems that are rational and sufficient

### Agents and Environments:



**Fig 2.1:** Agents and Environments

### Agent:

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

- ✓ A *human agent* has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- ✓ A *robotic agent* might have cameras and infrared range finders for sensors and various motors for actuators.
- ✓ A *software agent* receives keystrokes, file contents, and network packets as

sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

**Percept:**

We use the term percept to refer to the agent's perceptual inputs at any given instant.

**Percept Sequence:**

An agent's percept sequence is the complete history of everything the agent has ever perceived.

**Agent function:**

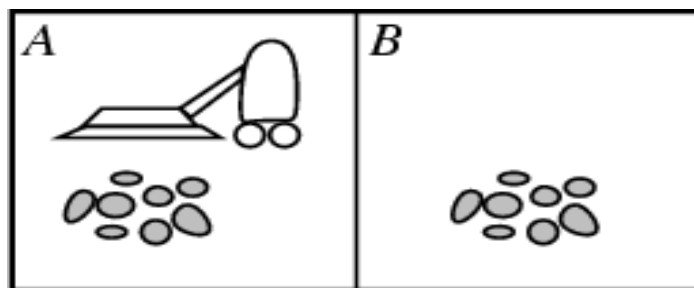
Mathematically speaking, we say that an agent's behavior is described by the agent function that maps any **given percept sequence to an action**.

**Agent program**

Internally, the agent function for an artificial agent will be implemented by an agent program. It is important to keep these two ideas distinct. The agent function is an abstract

mathematical description; the agent program is a concrete implementation, running on the agent architecture.

To illustrate these ideas, we will use a very simple example—the vacuum-cleaner world shown in **Fig 2.1.5**. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square. A partial tabulation of this agent function is shown in **Fig 2.1.6**.



**Fig 2.1.5:** A vacuum-cleaner world with just two locations.

## Agent function

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
...	

**Fig 2.1.6:** Partial tabulation of a simple agent function for the example: vacuum-cleaner world shown in the Fig2.1.5

Function REFLEX-VACCUM-AGENT ([location, status]) returns an

action If status=Dirty then return Suck

else if location = A then return Right

else if location = B then return Left

**Fig 2.1.6(i):** The REFLEX-VACCUM-AGENT program is invoked for each new percept (location, status) and returns an action each time

- A **Rational agent** is one that does the right thing. we say that the right action is the one that will cause the agent to be most successful. That leaves us with the problem of deciding how and when to evaluate the agent's success. We use the term performance measure for the how—the criteria that determine how successful an agent is.

- ✓ Ex-Agent cleaning the dirty floor
- ✓ Performance Measure-Amount of dirt collected
- ✓ When to measure-Weekly for better results

**What is rational at any given time depends on four things:**

- The performance measure defining the criterion of success
- The agent’s prior knowledge of the environment
- The actions that the agent can perform
- The agent’s percept sequence up to now.

**Omniscience ,Learning and Autonomy:**

- We need to distinguish between rationality and omniscience. An **Omniscient agent** knows the actual outcome of its actions and can act accordingly but omniscience is impossible in reality.
- Rational agent not only gathers information but also **learns** as much as possible from what it perceives.
- If an agent just relies on the prior knowledge of its designer rather than its own percepts then the agent lacks **autonomy**.
- A system is autonomous to the extent that its behavior is determined by its own experience.
- A rational agent should be autonomous.
  - E.g., a clock(lacks autonomy)
- No input (percepts)
- Run only by its own algorithm (prior knowledge)
- No learning, no experience, etc.

**ENVIRONMENTS:**

The Performance measure, the environment and the agents actuators and sensors comes under the heading task environment. We also call this as PEAS(Performance,Environment,Actuators,Sensors)

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

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

## Other PEAS Examples

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	healthy patient, costs, lawsuits	patient, hospital, staff	display questions, tests, diagnoses, treatments, referrals	keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	correct image categorization	downlink from orbiting satellite	display categorization of scene	color pixel arrays
Part-picking robot	percentage of parts in correct bins	conveyor belt with parts, bins	jointed arm and hand	camera, joint angle sensors
Refinery controller	purity, yield, safety	refinery, operators	valves pumps, heaters displays	temperature, pressure, chemical sensors
Interactive English tutor	student's score on test	set of students, testing agency	display exercises, suggestions, corrections	keyboard entry

### Environment-Types:

#### 1. Accessible vs. inaccessible or Fully observable vs Partially Observable:

If an agent sensor can sense or access the complete state of an environment at each point of time then it is a fully observable environment, else it is partially observable.

#### 2. Deterministic vs. Stochastic:

If the next state of the environment is completely determined by the current state and the actions selected by the agents, then we say the environment is deterministic

#### 3. Episodic vs. nonepisodic:

- The agent's experience is divided into "episodes." Each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself, because subsequent episodes do not depend on what actions occur in previous episodes.
- Episodic environments are much simpler because the agent does not need to think ahead.

#### 4. 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.

## 5. Discrete vs. continuous:

If there are a limited number of distinct, clearly defined percepts and actions we say that the environment is discrete. Otherwise, it is continuous.

**Examples of task environments**

Task Environment	Observable	Deterministic	Episodic	Static	Discrete	Agents
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Strategic	Sequential	Semi	Discrete	Multi
Poker	Partially	Strategic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi driving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diagnosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Image-analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

**Figure 2.6** Examples of task environments and their characteristics.

## STRUCTURE OF INTELLIGENT AGENTS

- The job of AI is to design the agent program: a function that implements the agent mapping from percepts to actions. We assume this program will run on some sort of ARCHITECTURE computing device, which we will call the architecture.
- The architecture might be a plain computer, or it might include special-purpose hardware for certain tasks, such as processing camera images or filtering audio input. It might also include software that provides a degree of insulation between the raw computer and the agent program, so that we can program at a higher level. In general, the architecture makes the percepts from the sensors available to the program, runs the program, and feeds the program's action choices to the effectors as they are generated.
- The relationship among agents, architectures, and programs can be summed up as follows: agent = architecture + program

Agent Type	Percepts	Actions	Goals	Environment
Medical diagnosis system	Symptoms, findings, patient's answers	Questions, tests, treatments	Healthy patient, minimize costs	Patient, hospital
Satellite image analysis system	Pixels of varying intensity, color	Print a categorization of scene	Correct categorization	Images from orbiting satellite
Part-picking robot	Pixels of varying intensity	Pick up parts and sort into bins	Place parts in correct bins	Conveyor belt with parts
Refinery controller	Temperature, pressure readings	Open, close valves; adjust temperature	Maximize purity, yield, safety	Refinery
Interactive English tutor	Typed words	Print exercises, suggestions, corrections	Maximize student's score on test	Set of students

Figure 2.3 Examples of agent types and their PAGE descriptions.

**Agent programs:**

- Intelligent agents accept percepts from an environment and generates actions. The early versions of agent programs will have a very simple form (Figure 2.4)
- Each will use some internal data structures that will be updated as new percepts arrive.
- These data structures are operated on by the agent's decision-making procedures to generate an action choice, which is then passed to the architecture to be executed

```

function TABLE-DRIVEN-AGENT(percept) returns action
  static: percepts, a sequence, initially empty
           table, a table, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(perceptstable)
  return action

```

Figure 2.5 An agent based on a prespecified lookup table. It keeps track of the percept sequence and just looks up the best action.

**Types of agents:**

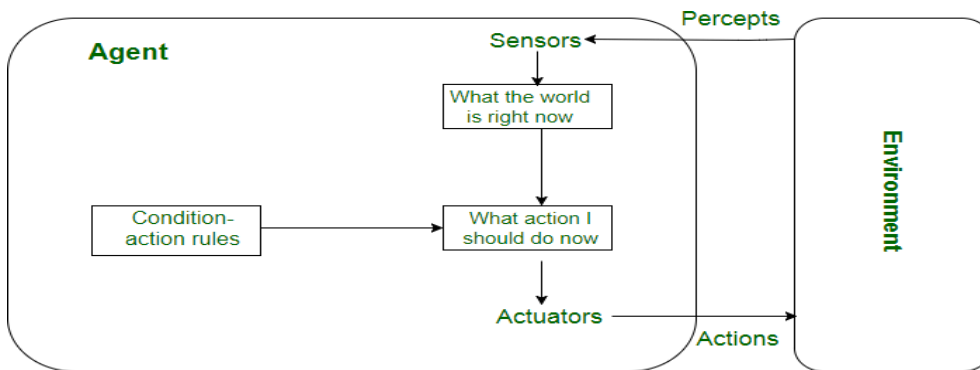
Agents can be grouped into four classes based on their degree of perceived intelligence and capability :

- Simple Reflex Agents

- Model-Based Reflex Agents
- Goal-Based Agents
- Utility-Based Agents

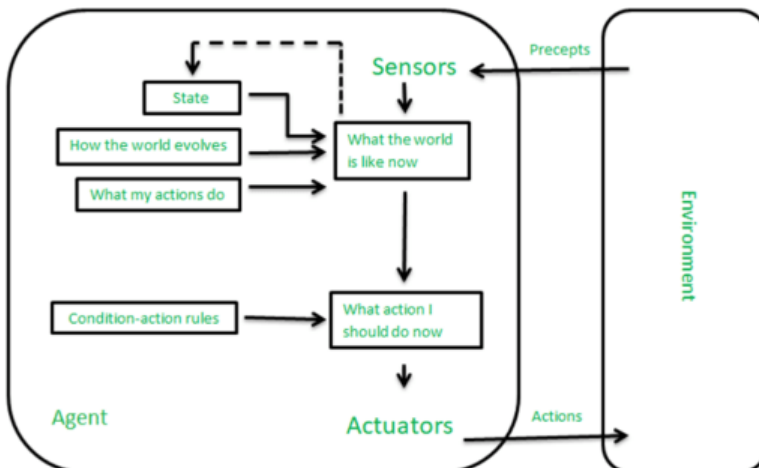
**Simple reflex agents:**

- Simple reflex agents ignore the rest of the percept history and act only on the basis of the current percept.
- The agent function is based on the condition-action rule.
- If the condition is true, then the action is taken, else not. This agent function only succeeds when the environment is fully observable.



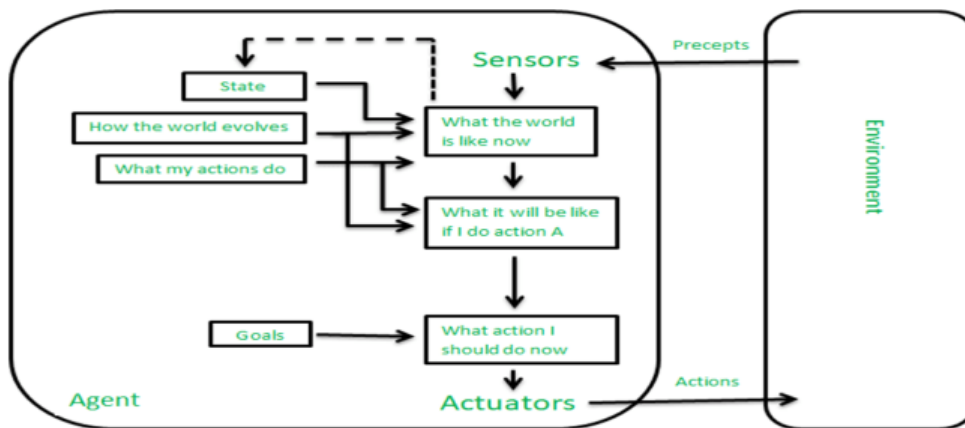
**Model-based reflex agents:**

- 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.



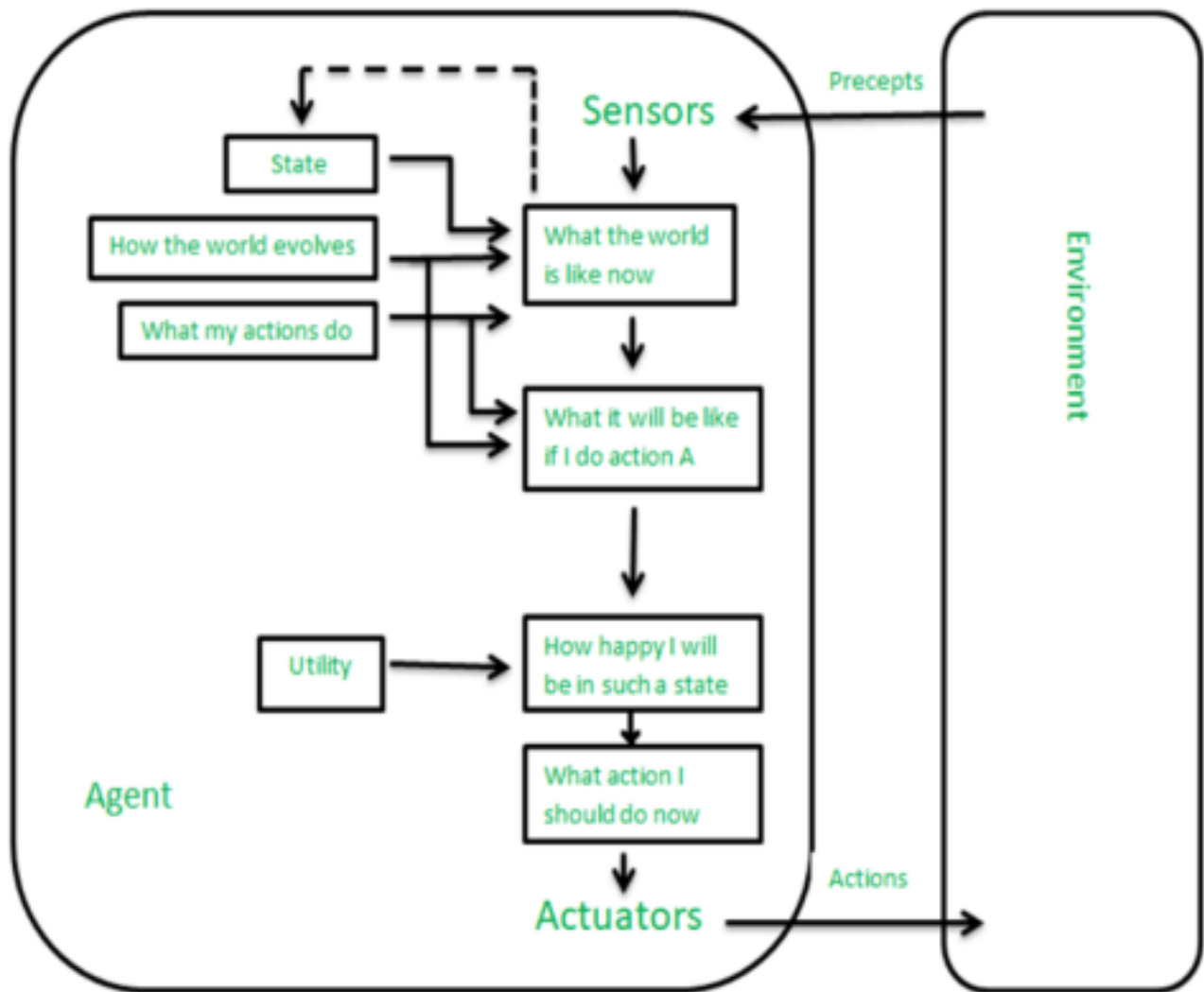
### Goal-based agents:

- A goal-based agent has an agenda.
- It operates based on a goal in front of it and makes decisions based on how best to reach that goal.
- A goal-based agent operates as a search and planning function, meaning it targets the goal ahead and finds the right action in order to reach it.
- Expansion of model-based agent.



### Utility-based agents:

- A utility-based agent is an agent that acts based not only on what the goal is, but the best way to reach that 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 term utility can be used to describe how "happy" the agent is.



### Problem Solving Agents:

- Problem solving agent is a goal-based agent.
- Problem solving agents decide what to do by finding sequence of actions that lead to desirable states.

### Goal Formulation:

It organizes the steps required to formulate/ prepare one goal out of multiple goals available.

### Problem Formulation:

It is a process of deciding what actions and states to consider to follow goal formulation. The process of looking for a best sequence to achieve a goal is called

### Search.

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

Once the solution is found the action it recommends can be carried out. This is called **Execution**

### phase. Well Defined problems and solutions:

A problem can be defined formally by 4 components:

- The **initial state** of the agent is the state where the agent starts in. In this case, the initial state can be

described as In: Arad

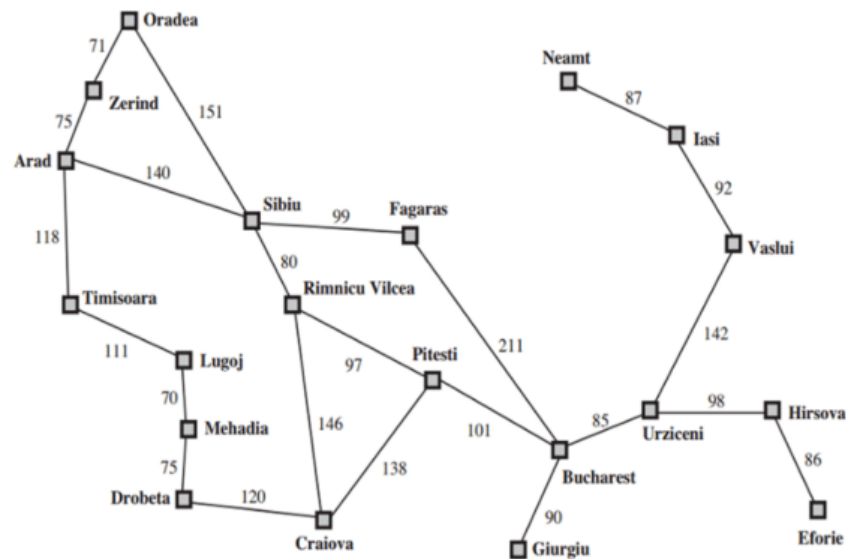
- The possible **actions** available to the agent, corresponding to each of the state the agent resides in.

For example,  $ACTIONS(In: Arad) = \{Go: Sibiu, Go: Timisoara, Go: Zerind\}$ .

Actions are also known as operations.

- A description of what each action does. the formal name for this is **Transition model**, Specified by the function  $Result(s,a)$  that returns the state that results from the action  $a$  in state  $s$ .

We also use the term Successor to refer to any state reachable from a given state by a single action. For EX:  $Result(In(Arad),GO(Zerind))=In(Zerind)$



Together the initial state, actions and transition model implicitly defines the **state space** of the problem. State space: set of all states reachable from the initial state by any sequence of actions

- **The goal test**, determining whether the current state is a goal state. Here, the goal state is  $\{In: Bucharest\}$
- **The path cost function**, which determine the cost of each path, which is reflecting in the performance measure.

we define the cost function as  $c(s, a, s')$ , where  $s$  is the current state and  $a$  is the action performed by the agent to reach state  $s'$ .

```

function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
  inputs: p, a percept
  static: s, an action sequence, initially empty
           state, some description of the current world state
           g, a goal, initially null
           problem, a problem formulation

  state ← UPDATE-STATE(state, p)
  if s is empty then
    g ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, g)
    s ← SEARCH(problem)
  action ← RECOMMENDATION(s, state)
  s ← REMAINDER(s, state)
  return action

```

**Figure 3.1** A simple problem-solving agent.

Example –

8 puzzle problem

**Initial State**

2	8	3
1	6	4
7		5

**Goal State**

1	2	3
8		4
7	6	5

- **States:** a state description specifies the location of each of the eight tiles in one of the ninesquares. For efficiency, it is useful to include the location of the blank.
- **Actions:** blank moves left, right, up, or down.
- **Transition Model:** Given a state and action, this returns the resulting state. For example if we apply left to the start state the resulting state has the 5 and the blank switched.
- **Goal test:** state matches the goal configuration shown in fig.
- **Path cost:** each step costs 1, so the path cost is just the length of the path.

### State Space Search/Problem Space Search:

The state space representation forms the basis of most of the AI methods.

- Formulate a problem as a **state space search** by showing the legal problem states, the legal operators, and the initial and goal states.
- A **state** is defined by the specification of the values of all attributes of interest in the world
- An **operator** changes one state into the other; it has a precondition which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator
- The **initial state** is where you start
- The **goal state** is the partial description of the solution

### Formal Description of the problem:

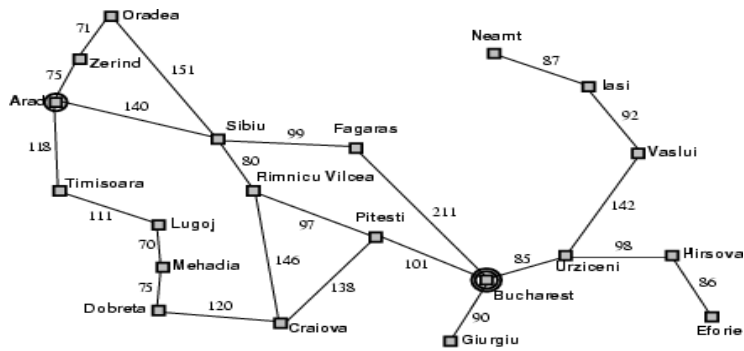
1. Define a state space that contains all the possible configurations of the relevant objects.
  2. Specify one or more states within that space that describe possible situations from which the problem solving process may start (**initial state**)
  3. Specify one or more states that would be acceptable as solutions to the problem. (**goal states**)
- Specify a set of rules that describe the actions (**operations**) available

### State-Space Problem Formulation:

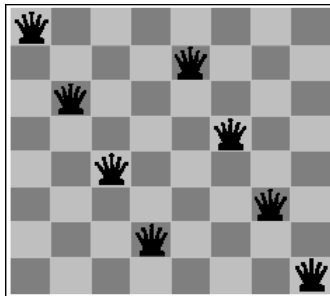
Example: A problem is defined by four items:

1. **initial state** e.g., "at Arad"
2. **actions or successor function** :  $S(x)$  = set of action–state pairs e.g.,  $S(\text{Arad}) = \{ \langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots \}$
3. **goal test (or set of goal states)**  
e.g.,  $x = \text{"at Bucharest"}$ ,  $\text{Checkmate}(x)$
4. **path cost (additive)**  
e.g., sum of distances, number of actions executed, etc.  
 $c(x, a, y)$  is the step cost, assumed to be  $\geq 0$

A solution is a sequence of actions leading from the initial state to a goal state



### Example: 8-queens problem



1. **Initial State:** Any arrangement of 0 to 8 queens on board.
2. **Operators:** add a queen to any square.
3. **Goal Test:** 8 queens on board, none attacked.
4. **Path cost:** not applicable or Zero (because only the final state counts, search cost might be of interest).

### Search strategies:

**Search:** Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:

**Search Space:** Search space represents a set of possible solutions, which a system may

have. **Start State:** It is a state from where agent begins the search.

**Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.

## Properties of Search Algorithms

Which search algorithm one should use will generally depend on the problem domain. There are four important factors to consider:

1. **Completeness** – Is a solution guaranteed to be found if at least one solution exists?
2. **Optimality** – Is the solution found guaranteed to be the best (or lowest cost) solution if there exists more than one solution?
3. **Time Complexity** – The upper bound on the time required to find a solution, as a function of the complexity of the problem.
4. **Space Complexity** – The upper bound on the storage space (memory) required at any point during the search, as a function of the complexity of the problem.

### State Spaces versus Search Trees:

- State Space
  - Set of valid states for a problem
  - Linked by operators
  - e.g., 20 valid states (cities) in the Romanian travel problem
- Search Tree
  - Root node = initial state
  - Child nodes = states that can be visited from parent
  - Note that the depth of the tree can be infinite
    - E.g., via repeated states
  - Partial search tree
    - Portion of tree that has been expanded so far
  - Fringe
    - Leaves of partial search tree, candidates

for expansion Search trees = data structure to search state-space

## Searching

Many traditional search algorithms are used in AI applications. For complex problems, the traditional algorithms are unable to find the solution within some practical time and space limits. Consequently, many special techniques are developed; using heuristic functions. The algorithms that use heuristic

functions are called heuristic algorithms. Heuristic algorithms are not really intelligent; they appear to be intelligent because they achieve better performance.

Heuristic algorithms are more efficient because they take advantage of feedback from the data to direct the search path.

### Uninformed search

Also called blind, exhaustive or brute-force search, uses no information about the problem to guide the search and therefore may not be very efficient.

### Informed Search:

Also called heuristic or intelligent search, uses information about the problem to guide the search, usually guesses the distance to a goal state and therefore efficient, but the search may not be always possible.

## Uninformed Search (Blind searches):

### 1. Breadth First Search:

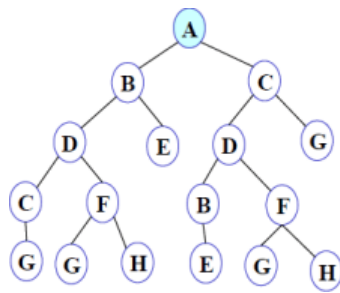
- One simple search strategy is a breadth-first search. In this strategy, the root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on.
- In general, all the nodes at depth  $d$  in the search tree are expanded before the nodes at depth  $d + 1$ .

```
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.

### BFS illustrated:

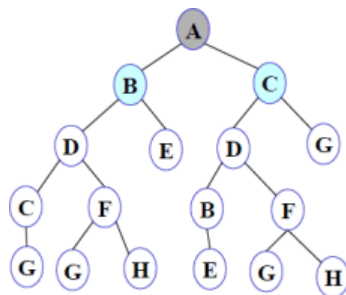
**Step 1:** Initially frontier contains only one node corresponding to the source state A.



**Figure 1**

**Frontier:** A

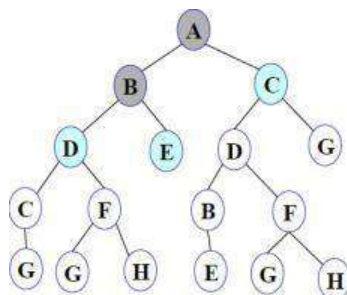
**Step 2:** A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.



**Figure 2**

**Frontier:** B C

**Step 3:** Node B is removed from fringe and is expanded. Its children D, E are generated and



putat the back of fringe.

**Figure 3**

**Frontier:** C D E

**Step 4:** Node C is removed from fringe and is expanded. Its children D and G are added to theback of fringe.

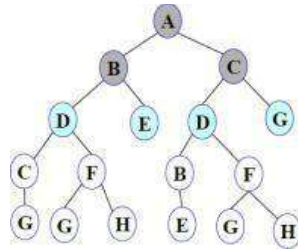


Figure 4

Frontier: D E D G

Step 5: Node D is removed from fringe. Its children C and F are generated and added to the back of fringe.

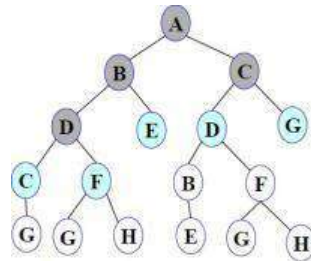


Figure 5

Frontier: E D G C F

Step 6: Node E is removed from fringe. It has no children.

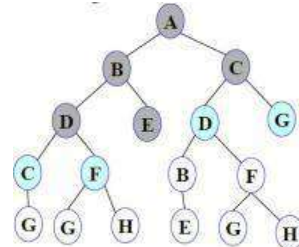


Figure 6

Frontier: D G C F

Step 7: D is expanded; B and F are put in OPEN.

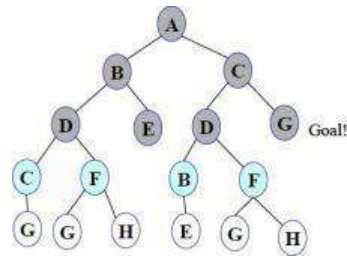


Figure 7

Frontier: G C F B F

**Step 8:** G is selected for expansion. **It is found to be a goal node.** So the algorithm returns the path A C G by following the parent pointers of the node corresponding to G. The algorithm terminates.

**Breadth first search is:**

- One of the simplest search strategies
- Complete. If there is a solution, BFS is guaranteed to find it.
- If there are multiple solutions, then a minimal solution will be found
- The algorithm is optimal (i.e., admissible) if all operators have the same cost. Otherwise, breadth first search finds a solution with the shortest path length.
- **Time complexity** :  $O(b^d)$
- **Space complexity** :  $O(b^d)$
- **Optimality** : Yes

***b - branching factor (maximum no of successors of any node), d – Depth of the shallowest goal node***

**Advantages:** ***Maximum length of any path (m) in search space***

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

#### Disadvantages:

- Requires the generation and storage of a tree whose size is exponential the depth of the shallowest goal node.
- The breadth first search algorithm cannot be effectively used unless the search space is quite small.

#### Applications Of Breadth-First Search Algorithm

**GPS Navigation systems:** Breadth-First Search is one of the best algorithms used to find neighboring locations by using the GPS system.

**Broadcasting:** Networking makes use of what we call as packets for communication. These packets follow a traversal method to reach various networking nodes. One of the most commonly used traversal

methods is Breadth-First Search. It is being used as an algorithm that is used to communicate broadcasted packets across all the nodes in a network.

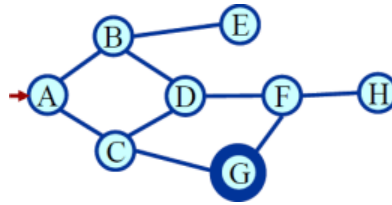
#### Depth- First- Search.

We may sometimes search the goal along the largest depth of the tree, and move up only when further traversal along the depth is not possible. We then attempt to find alternative offspring of the parent of the node (state) last visited. If we visit the nodes of a tree using the above principle to search the goal, the traversal made is called depth first traversal and consequently the search strategy is called *depth first search*.

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

```
function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
    cutoff_occurred? ← false
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      result ← RECURSIVE-DLS(child, problem, limit - 1)
      if result = cutoff then cutoff_occurred? ← true
      else if result ≠ failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

DFS illustrated:



A State Space Graph

Step 1: Initially fringe contains only the node for A.

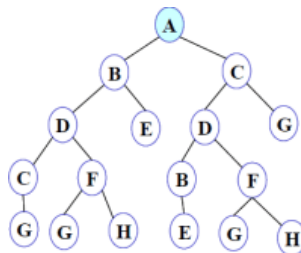


Figure 1

FRINGE: A

Step 2: A is removed from fringe. A is expanded and its children B and C are put in front of fringe.

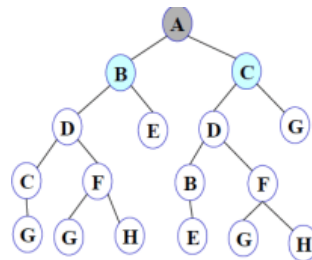


Figure 2

FRINGE: B C

Step 3: Node B is removed from fringe, and its children D and E are pushed in front of fringe.

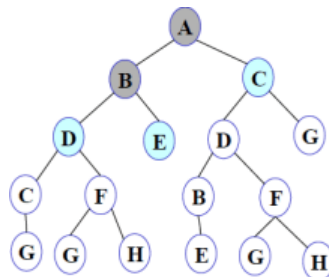


Figure 3

FRINGE: D E C

Step 4: Node D is removed from fringe. C and F are pushed in front of fringe.

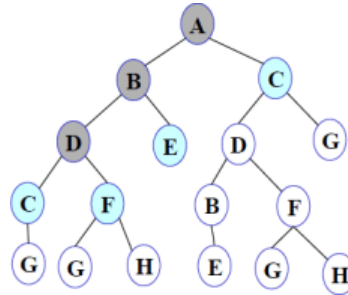


Figure 4

FRINGE: C F E C

Step 5: Node C is removed from fringe. Its child G is pushed in front of fringe.

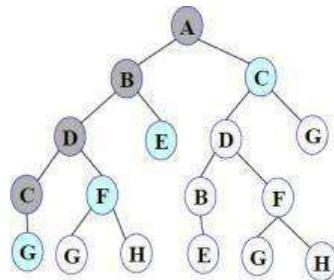


Figure 5

Figure 5

FRINGE: G F E C

Step 6: Node G is expanded and found to be a goal node.

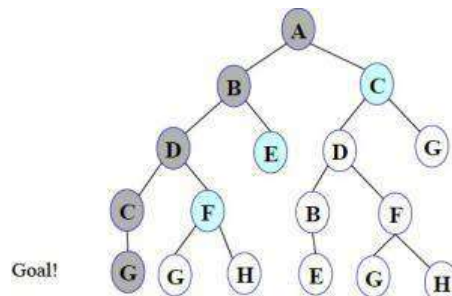


Figure 6

## FRINGE: G F E C

The solution path A-B-D-C-G is returned and the algorithm terminates.

### Depth first search

1. takes exponential time.
2. If  $N$  is the maximum depth of a node in the search space, in the worst case the algorithm will take time  $O(b^d)$ .
3. The space taken is linear in the depth of the search tree,  $O(bN)$ .

Note that the time taken by the algorithm is related to the maximum depth of the search tree. If the search tree has infinite depth, the algorithm may not terminate. This can happen if the search space is infinite. It can also happen if the search space contains cycles. The latter case can be handled by checking for cycles in the algorithm. Thus **Depth First Search is not complete**.

## Iterative Deeping DFS

- The iterative deepening algorithm is a combination of DFS and BFS algorithms.
- This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

### Advantages:

- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

### Disadvantages:

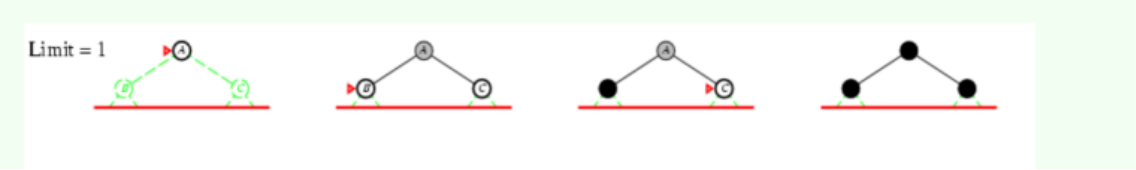
- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Iterative deepening search  $L=0$

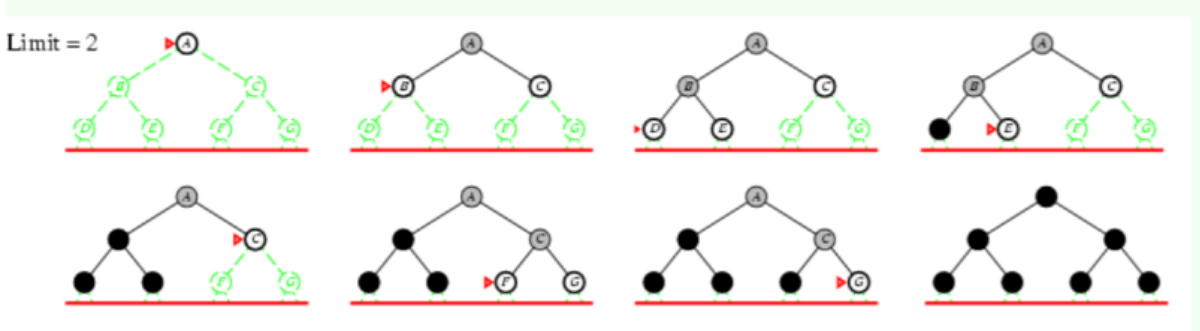
Limit = 0



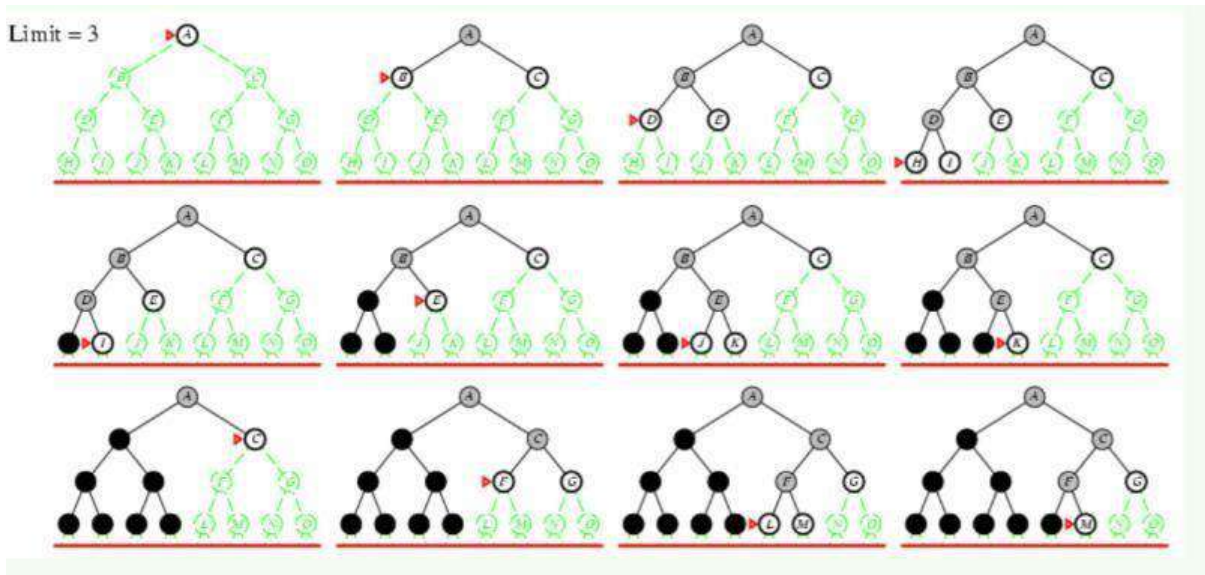
Iterative deepening search L=1



Iterative deepening search L=2



Iterative Deepening Search L=3



**M is the goal node. So we stop there.**

Complete: Yes

Time:  $O(bd)$

Space:  $O(bd)$

Optimal: Yes, if step cost = 1 or increasing function of depth.

Conclusion:

- We can conclude that IDS is a hybrid search strategy between BFS and DFS inheriting their advantages.
- IDS is faster than BFS and DFS.
- It is said that “IDS is the preferred uniformed search method when there is a large search space and the depth of the solution is not known”

A comparison table between DFS, BFS and IDDFS

	Time Complexity	Space Complexity	When to Use ?
DFS	$O(b^d)$	$O(d)$	=> Don't care if the answer is closest to the starting vertex/root. => When graph/tree is not very big/infinite.
BFS	$O(b^d)$	$O(b^d)$	=> When space is not an issue => When we do care/want the closest answer to the root.
IDDFS	$O(b^d)$	$O(bd)$	=> You want a BFS, you don't have enough memory, and somewhat slower performance is accepted. In short, you want a BFS + DFS.

### Informed search/Heuristic search

A *heuristic* is a method that

- might not always find the best solution **but** is guaranteed to find a good solution in reasonable time. By sacrificing completeness it increases efficiency.
- Useful in solving tough problems which
  - could not be solved any other way.
  - solutions take an infinite time or very long time to compute.

#### Calculating Heuristic Value:

- 1. Euclidian distance- used to calculate straight line distance.
- 2. Manhattan distance- If we want to calculate vertical or horizontal

distance For ex: 8 puzzle problem

#### Source state

1	3	2
---	---	---

6	5	4
	8	7

**destination state**

1	2	3
4	5	6
7	8	

Then the Manhattan distance would be sum of the no of moves required to move each number from source state to destination state.

<b>Number in 8 puzzle</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>
No. of moves to reach destination	0	2	1	2	0	2	2	0

3. No. of misplaced tiles for 8 puzzle problem

**Source state**

1	3	2
6	5	4
	8	7

**Destination state**

1	2	3
4	5	6
7	8	

Here just calculate the number of tiles that have to be changed to reach goal state Here 1,5,8 need not be changed

2,3,4,6,7 should be changed, so the heuristic value will be 5(because 5 tiles have to be changed)

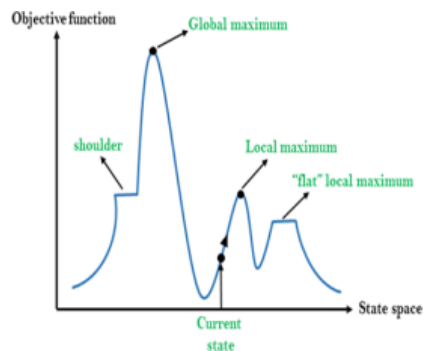
### **Hill Climbing Algorithm**

- ✓ Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

- ✓ It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- ✓ Hill Climbing is mostly used when a good heuristic is available.
- ✓ In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

The idea behind hill climbing is as follows.

1. Pick a random point in the search space.
2. Consider all the neighbors of the current state.
3. Choose the neighbor with the best quality and move to that state.
4. Repeat 2 thru 4 until all the neighboring states are of lower quality.
5. Return the current state as the solution state.



### Different regions in the state space landscape:

**Local Maximum:** Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.

**Global Maximum:** Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

**Current state:** It is a state in a landscape diagram where an agent is currently present.

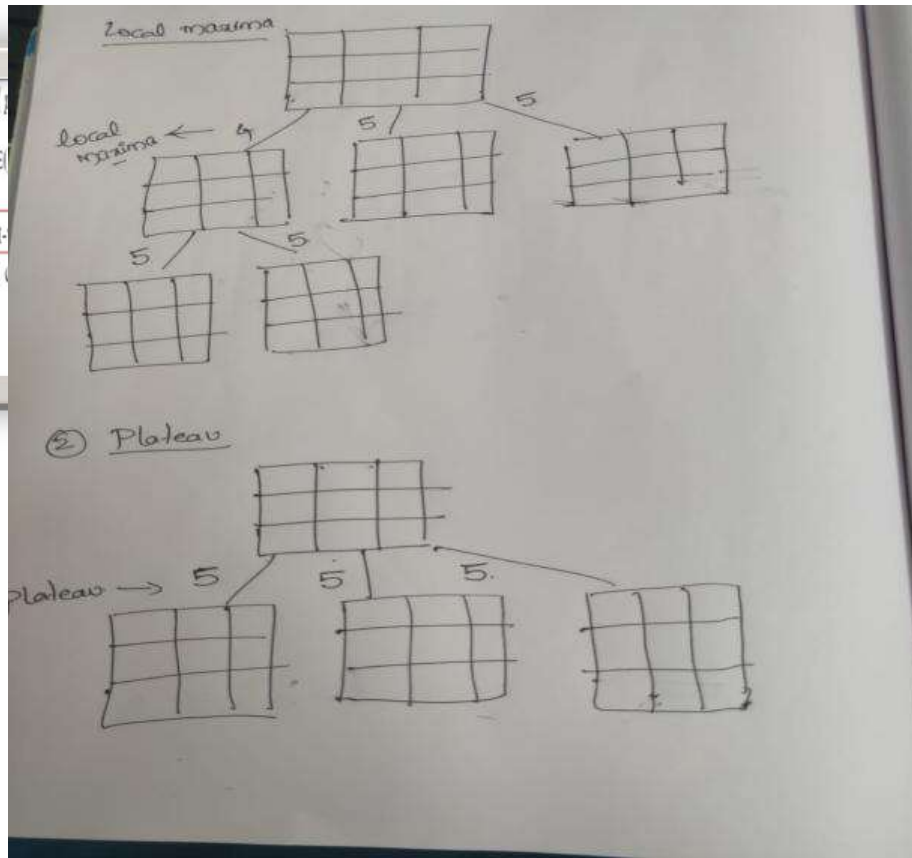
**Flat local maximum:** It is a flat space in the landscape where all the neighbor states of current states have the same value.

**Shoulder:** It is a plateau region which has an uphill edge.

### Algorithm for Hill Climbing

```

function HILL-CLIMBING()
  current ← MAKE-NODE()
  loop do
    neighbor ← a highest-
    if neighbor.VALUE > current.VALUE
      current ← neighbor
  
```



### Problems in Hill Climbing Algorithm:

#### : Hill-climbing

This simple policy has three well-known drawbacks:

1. **Local Maxima:** a local maximum as opposed to global maximum.
2. **Plateaus:** An area of the search space where evaluation function is flat, thus requiring random walk.
3. **Ridge:** Where there are steep slopes and the search direction is not towards the top but towards the side.

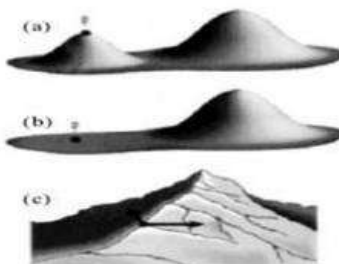


Figure 5.9 Local maxima, Plateaus and ridge situation for Hill Climbing

Simulated annealing search

A hill-climbing algorithm that never makes “downhill” moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk – that is, moving to a successor chosen uniformly at random from the set of successors – is complete, but extremely inefficient. Simulated annealing is an algorithm that combines hill-climbing with a random walk in some way that yields both efficiency and completeness.

The simulated annealing algorithm is quite similar to hill climbing. Instead of picking the best move, however, it picks the random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The

probability decreases exponentially with the “badness” of the move – the amount  $\Delta E$  by which the evaluation is worsened. The probability also decreases as the “temperature”  $T$  goes down: “bad” moves are more likely to be allowed at the start when temperature is high, and they become more unlikely as  $T$  decreases. One can prove that if the schedule lowers  $T$  slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing was first used extensively to solve VLSI layout problems. It has been applied widely to factory scheduling and other large-scale optimization tasks.

### **Best First Search:**

- A combination of depth first and breadth first searches.
- Depth first is good because a solution can be found without computing all nodes and breadth first is good because it does not get trapped in dead ends.
- The best first search allows us to switch between paths thus gaining the benefit of both approaches. At each step the most promising node is chosen. If one of the nodes chosen generates nodes that are less promising it is possible to choose another at the same level and in effect the search changes from depth to breadth. If on analysis these are no better than this previously unexpanded node and branch is not forgotten and the search method reverts to the

**OPEN** is a priority queue of nodes that have been evaluated by the heuristic function but

which have not yet been expanded into successors. The most promising nodes are at the front.

**CLOSED** are nodes that have already been generated and these nodes must be stored because a graph is being used in preference to a tree.

**Algorithm:**

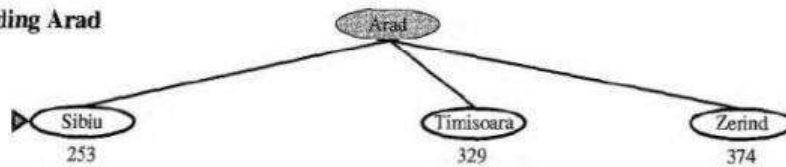
1. Start with OPEN holding the initial state
2. Until a goal is found or there are no nodes left on open do.
  - Pick the best node on OPEN
  - Generate its successors
  - For each successor Do
    - If it has not been generated before ,evaluate it ,add it to OPEN and record its parent
    - If it has been generated before change the parent if this new path is better and in that case update the cost of getting to any successor nodes.
3. If a goal is found or no more nodes left in OPEN, quit, else return to 2.

**Example:**

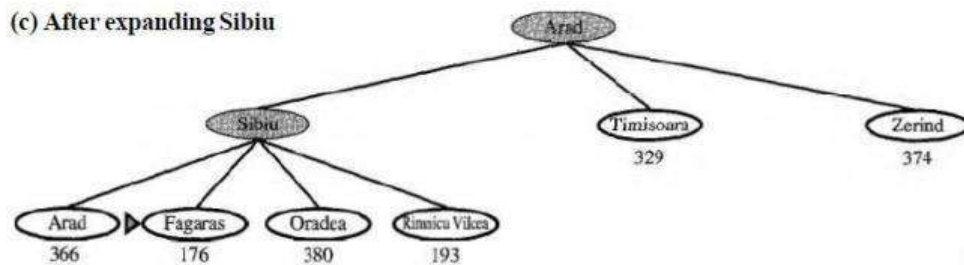
(a) The initial state



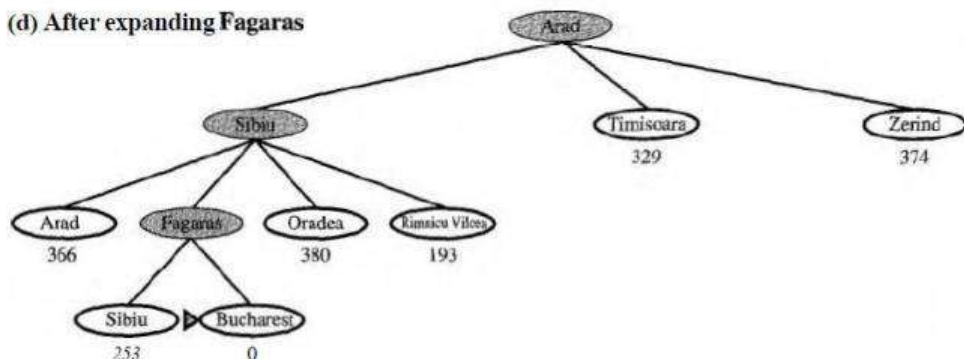
(b) After expanding Arad



(c) After expanding Sibiu



(d) After expanding Fagaras



1. It is not optimal.
2. It is incomplete because it can start down an infinite path and never return to try other possibilities.
3. The worst-case time complexity for greedy search is  $O(b^m)$ , where  $m$  is the maximum depth of the search space.
4. Because greedy search retains all nodes in memory, its space complexity is the same as its time complexity.

### A\* Algorithm

The Best First algorithm is a simplified form of the A\* algorithm.

The **A\* search algorithm** (pronounced "Ay-star") is a tree search algorithm that finds a path from a given initial node to a given goal node (or one passing a given goal test). It employs a "heuristic estimate" which ranks each node by an estimate of the best route that goes through

that node. It visits the nodes in order of this heuristic estimate.

Similar to greedy best-first search but is more accurate because A\* takes into account the nodes that have already been traversed.

From A\* we note that  $f = g + h$  where

$g$  is a measure of the distance/cost to go from the initial node to the current node

$h$  is an estimate of the distance/cost to solution from the current node.

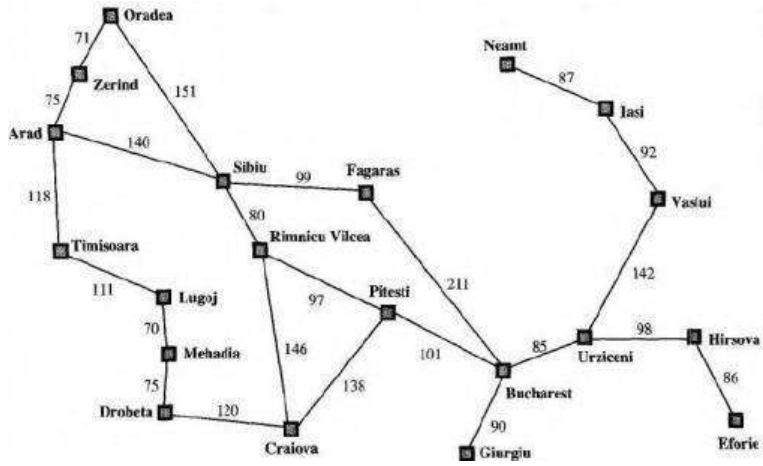
Thus  $f$  is an estimate of how long it takes to go from the initial node to the solution

**Algorithm:**

1. Initialize : Set OPEN = (S); CLOSED = ( )  
 $g(s) = 0, f(s) = h(s)$
2. Fail : If OPEN = ( ), Terminate and fail.
3. Select : select the minimum cost state, n, from OPEN,  
save n in CLOSED
4. Terminate : If  $n \in G$ , Terminate with success and return  $f(n)$
5. Expand : for each successor, m, of n
  - a) If  $m \in [OPEN \cup CLOSED]$  Set  $g(m) = g(n) + c(n, m)$  Set  $f(m) = g(m) + h(m)$   
Insert m in OPEN
  - b) If  $m \in [OPEN \cup CLOSED]$   
Set  $g(m) = \min \{ g(m), g(n) + c(n, m) \}$  Set  $f(m) = g(m) + h(m)$   
If  $f(m)$  has decreased and  $m \in CLOSED$   
Move m to OPEN.

**Description:**

- A\* begins at a selected node. Applied to this node is the "cost" of entering this node (usually zero for the initial node). A\* then estimates the distance to the goal node from the current node. This estimate and the cost added together are the heuristic which is assigned to the path leading to this node. The node is then added to a priority queue, oftencalled "open".
- The algorithm then removes the next node from the priority queue (because of the way a priority queue works, the node removed will have the lowest heuristic). If the queue is empty, there is no path from the initial node to the goal node and the algorithm stops. If the node is the goal node, A\* constructs and outputs the successful path and stops.
- If the node is not the goal node, new nodes are created for all admissible adjoining nodes;the exact way of doing this depends on the problem at hand. For each successive node, A\* calculates the "cost" of entering the node and saves it with the node. This cost is calculated from the cumulative sum of costs stored with its ancestors, plus the cost of the operation which reached this new node.
- The algorithm also maintains a 'closed' list of nodes whose adjoining nodes have been checked. If a newly generated node is already in this list with an equal or lower cost, no further processing is done on that node or with the path associated with it. If a node in the closed list matches the new one, but has been stored with a *higher* cost, it is removed from the closed list, and processing continues on the new node.
- Next, an estimate of the new node's distance to the goal is added to the cost to form the heuristic for that node. This is then added to the 'open' priority queue, unless an identical node is found there.
- Once the above three steps have been repeated for each new adjoining node, the original node taken from the priority queue is added to the 'closed' list. The next node is then popped from the priority queue and the process is repeated



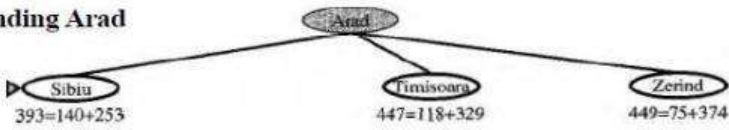
The heuristic costs from each city to Bucharest:

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

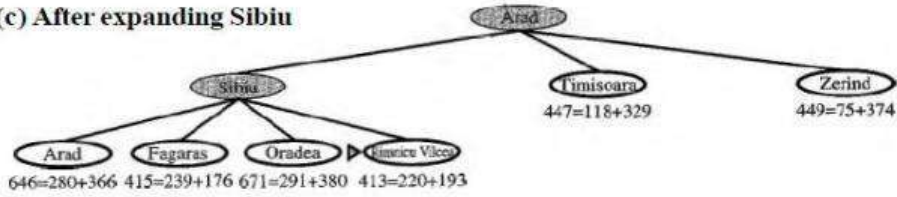
(a) The initial state

$$366=0+366$$

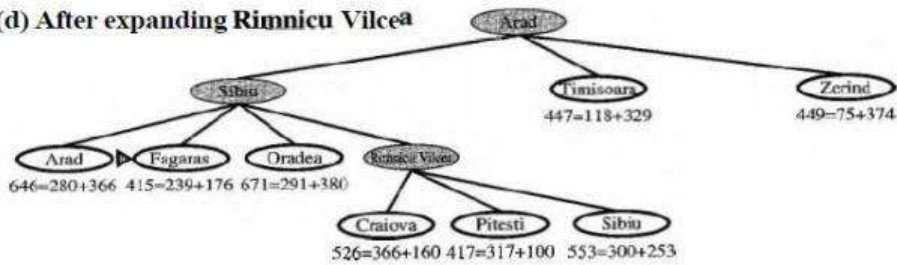
(b) After expanding Arad



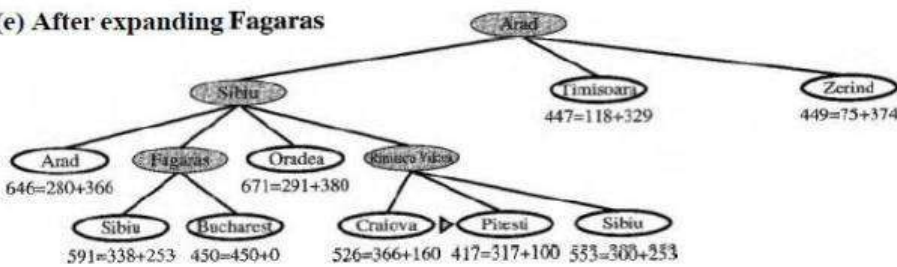
(c) After expanding Sibiu



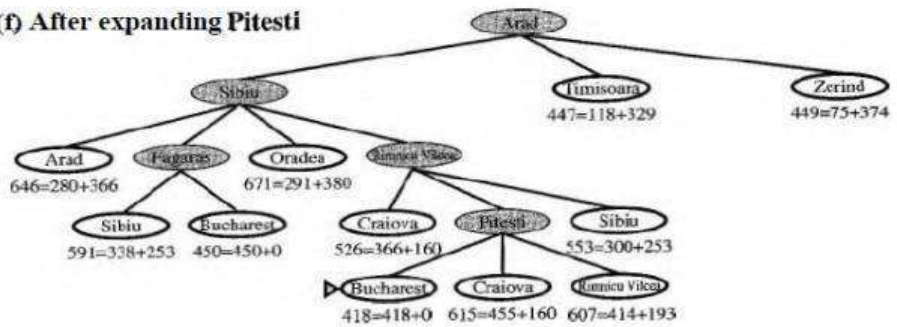
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti



### A\* search properties:

- The algorithm A\* is admissible. This means that provided a solution exists, the first solution found by A\* is an optimal solution. A\* is admissible under the following conditions:
  - Heuristic function: for every node  $n$ ,  $h(n) \leq h^*(n)$ .
    - A\* is also complete.
- A\* is optimally efficient for a given heuristic.
- A\* is much more efficient than uninformed search.

### Constraint Satisfaction Problems

<https://www.cnblogs.com/RDaneelOlivaw/p/8072603.html>

Sometimes a problem is not embedded in a long set of action sequences but requires picking the best option from available choices. A good general-purpose problem solving technique is to list the constraints of a situation (either negative constraints, like limitations, or positive elements that you want in the final solution). Then pick the choice that satisfies most of the constraints.

Formally speaking, a **constraint satisfaction problem (or CSP)** is defined by a set of variables,  $X_1; X_2; \dots$

$; X_n$ , and a set of constraints,  $C_1; C_2; \dots; C_m$ . Each variable  $X_i$  has a nonempty domain  $D_i$  of possible values. Each constraint  $C_i$  involves some subset of variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all of the variables,  $\{X_i = v_i; X_j = v_j; \dots\}$ . An assignment that does not violate any constraints is called a consistent or

legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a CSP is a complete assignment that satisfies all the constraints. Some CSPs also require a solution that maximizes an objective function.

CSP can be given an **incremental formulation** as a standard search problem as follows:

1. **Initial state:** the empty assignment  $fg$ , in which all variables are unassigned.
2. **Successor function:** a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.

3. **Goal test:** the current assignment is complete.
4. **Path cost:** a constant cost for every step

**Examples:**

1. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear inequalities forming a *convex* region.

$$\begin{array}{r}
 T \ W \ O \\
 + \ T \ W \ O \\
 \hline
 F \ O \ U \ R
 \end{array}$$

2. **Crypt arithmetic** puzzles.

**Example: The map coloring problem.**

The task of coloring each region red, green or blue in such a way that no neighboring regions have the same color.

We are given the task of coloring each region red, green, or blue in such a way that the neighboring regions must not have the same color.

To formulate this as CSP, we define the variable to be the regions: WA, NT, Q, NSW, V, SA, and T. The domain of each variable is the set {red, green, blue}. The constraints require neighboring regions to have distinct colors: for example, the allowable combinations for WA and NT are the pairs {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}. (The constraint can also be represented as the inequality  $WA \neq NT$ ). There are many possible solutions,

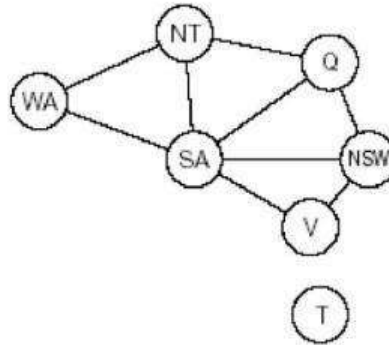


Variables WA, NT, Q, NSW, V, SA, T  
 Domains  $D_i = \{red, green, blue\}$   
 Constraints: adjacent regions must have different colors  
 e.g.,  $WA \neq NT$  (if the language allows this), or  
 $(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \dots\}$

such as {WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red}. Map of Australia showing each of its states and territories

Constraint Graph: A CSP is usually represented as an undirected graph, called constraint graph where the nodes are the variables and the edges are the binary constraints.

Constraint graph: nodes are variables, arcs show constraints



The map-coloring problem represented as a constraint

graph. CSP can be viewed as a standard search

problem as follows:

- > **Initial state** : the empty assignment {}, in which all variables are unassigned.
- > **Successor function**: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.
- > **Goal test**: the current assignment is complete.
- > **Path cost**: a constant cost (E.g., 1) for every step.