

# **Introduction to Artificial Intelligence**



***Dr. Pulak Sahoo***

Associate Professor

Silicon Institute of Technology

# **Syllabus - 18CS2T29 : Artificial Intelligence**

## **B.Tech – CSE-A – 6<sup>th</sup> sem**

- **Module I**
- **Artificial Intelligence:** Introduction
- **Intelligent Agents:** Agents, Behavior, Nature of Env., Structure of Agents
- **Problem Solving by Searching** - Problem-Solving Agents, Example Problems, Searching for Solutions, Uninformed search strategies, Searching with Partial Info
- **Module II**
- **Informed Search & Exploration:** Informed (Heuristic) search strategies, Heuristic functions, Local Search Algorithms & Optimization Problems
- **Constraint Satisfaction Problems:** Introduction, Backtracking search for CSPs, Local Search for CSPs
- **Adversarial Search:** Games, Optimal Decisions in Games, Alpha-Beta Pruning;
- **Knowledge & Reasoning:** Knowledge-Based Agents, TheWumpusWorld.

# **Syllabus - 18CS2T29 : Artificial Intelligence**

## **B.Tech – CSE – 6<sup>th</sup> sem**

- **Module III**
- **Knowledge and Reasoning:** Logic, Propositional Logic, Reasoning Patterns in Propositional Logic
- **First-Order Logic:** Syntax and Semantics of First-Order Logic, Using First-Order Logic, Knowledge Engineering in First-Order Logic
- **Inference in First-Order Logic:** Propositional vs. First-Order Logic, Unification and Lifting, Forward Chaining, Backward Chaining, Resolution
- **Knowledge Representation:** Ontological Engineering , Categories & Objects, Semantic Nets, Frames

# Syllabus - 18CS2T29 : Artificial Intelligence

## B.Tech – CSE – 6<sup>th</sup> sem

- **Module IV**
- **Planning:** The Planning Problem, Planning with State-Space Search, Partial-Order Planning, Planning Graphs
- **Uncertain Knowledge & Reasoning:** Acting under Uncertainty, Bayes Rule & its use
- **Probabilistic Reasoning:** Representing Knowledge in an Uncertain Domain, Semantics of Bayesian Networks
- **Module V**
- **Learning:** Learning from Observations, Forms of Learning, Inductive Learning,
- **Learning Decision Trees:** Statistical Learning, Instance Based Learning, Neural Networks
- **Reinforcement Learning:** Passive & Active Reinforcement Learning
- **Expert Systems:** Introduction, Architecture, Representations.

# Books

- 1. Stuart Russell & Peter Norvig**, Artificial Intelligence: A Modern Approach, 2<sup>nd</sup> Edition, Pearson Education.
2. Elaine Rich, Kevin Knight, Shivshankar B Nair, Artificial Intelligence, McGraw Hill, 3rd Edition.
3. Nills J. Nilson, “Artificial Intelligence: A New Synthesis”, 2nd Edition, 2000, Elsevier India Publications, New Delhi.
4. Michael Negnevitsky, “Artificial Intelligence: A Guide to Intelligent Systems”, Second Edition, 2005, Pearson Education, Inc. New Delhi.
- 5. Dan W. Patterson**, “Introduction to Artificial Intelligence and Expert Systems”, 1<sup>st</sup> Edition, 1996, PHI Learning Pvt. Ltd., New Delhi.
6. E. Charniak and D. McDermott, Introduction to AI, 1st Edition, Addison-Wesley, 1985



# Introduction to AI - Topics

- **What is AI**
- **Four Approaches to AI**
  - Acting Humanly
  - Thinking Humanly
  - Acting Rationally
  - Thinking Rationally
- **Foundations of AI**
- **History of AI**





# What is AI?



# What is AI?

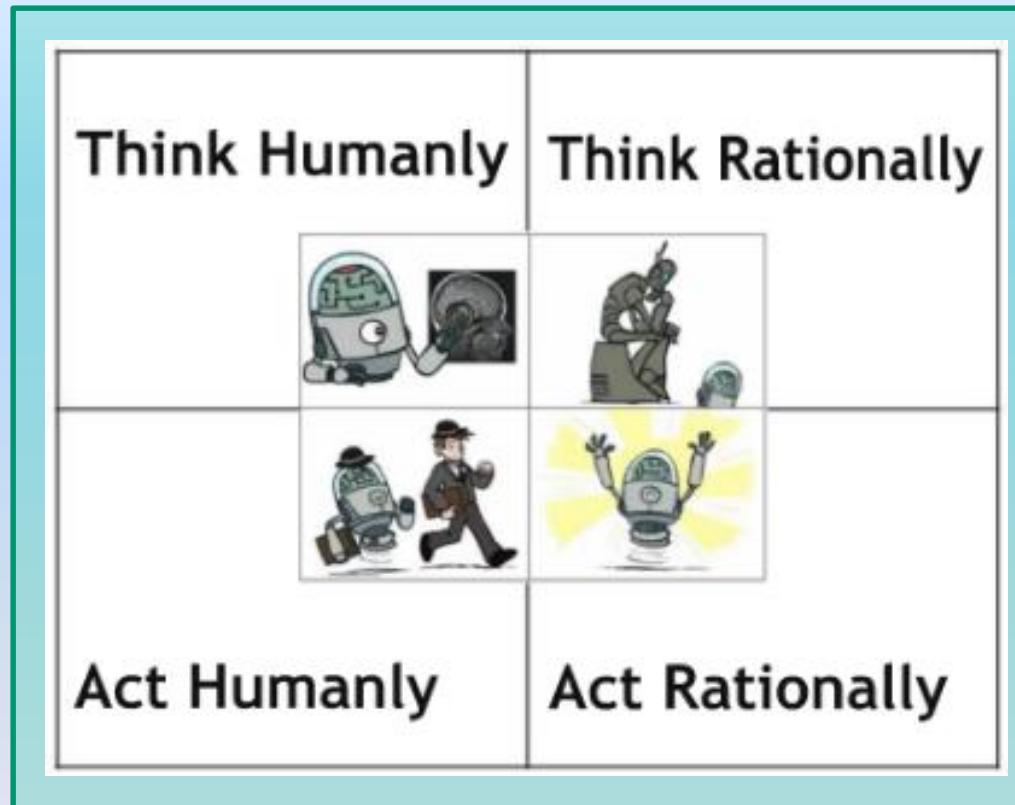
- Artificial intelligence is an area of computer science
  - that emphasizes the creation of intelligent machines
    - that think, work & react like humans
- Computers with artificial intelligence can do activities like:
  - Speech recognition
    - Car responding to master's voice
  - Learning
    - Amazon Alexa
  - Planning
    - Decision making by a Robot
  - Problem solving
    - Tic-tac-toe, TSP or n-Queen problem





# Four Approaches to AI

- Computers with AI can :



# Four Approaches to AI

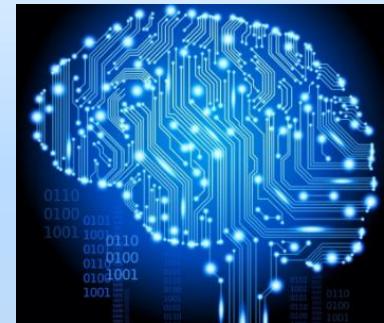
<b>Thinking Humanly</b>	<b>Thinking Rationally</b>
"The exciting new effort to make computers think ... <i>machines with minds</i> , in the full and literal sense" (Haugeland, 1985)  "The automation of activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985)  "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)
<b>Acting Humanly</b>	<b>Acting Rationally</b>
"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)  "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)	"A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990)  "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)



# Four Approaches to AI

- **(1) Think humanly – Cognitive modeling approach**

- **AI application** thinks like a **human**
  - *For this it needs to know “working of human mind”*
- **3 ways to do this:**
  - (1) By **Introspection** – *catching your “own thoughts”*
  - (2) By **Psychological experiments** – *observing a “person’s action”*
  - (3) By **Brain imaging** – *observing “brains in action”*
- **Cognitive Science** - scientific study of the mind & its processes
  - The study of **how we do, what we do** (*Ex: Car driving*)
  - How to make the **computer do it in the same way**

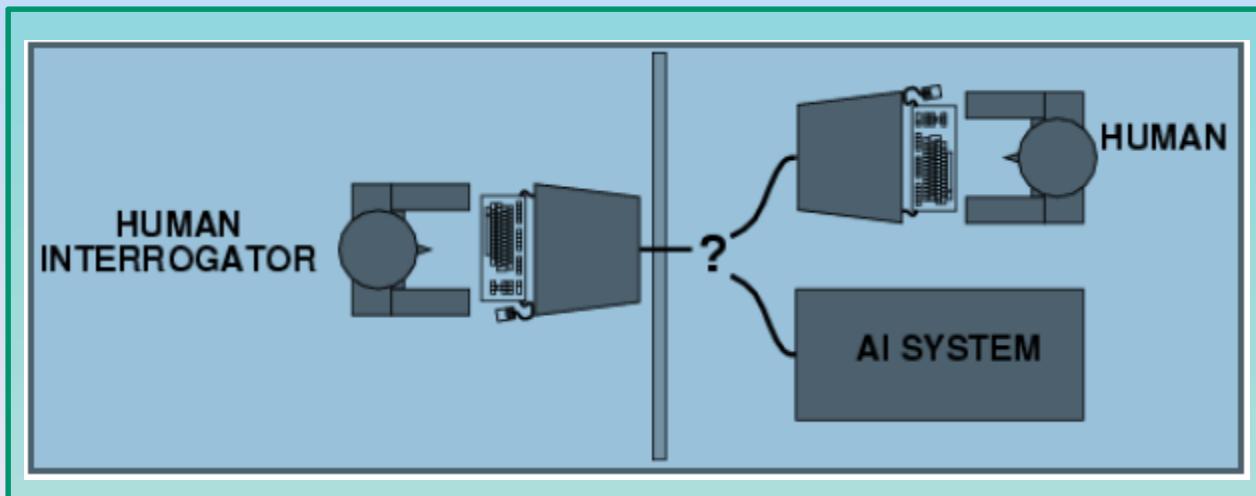




- **(2) Act humanly - Turing Test approach**

- **Turing Test (1950) of a computer with AI (by Alan Turing)**

- The **computer** & a **human** are interrogated by another **human** behind a barrier via **written questions**
- Computer passes the test if the human cannot tell if the **written response** is from a **computer** or **human**





## A Computer with AI needs to have below capabilities :

- Natural Language Processing – *to communicate in English*
- Knowledge Representation – *to store what it knows or hears*
- Automated Reasoning – *to use the stored info to answer questions*
- Machine Learning – *to adapt to new/changing circumstances*

## To pass the Total Turing Test, a machine needs :

- Computer vision – *to perceive objects (sensor)*
- Robotics – *to manipulate objects (actuator)*

# **Introduction to Artificial Intelligence**



***Dr. Pulak Sahoo***

Associate Professor

Silicon Institute of Technology



# Introduction to AI - Topics



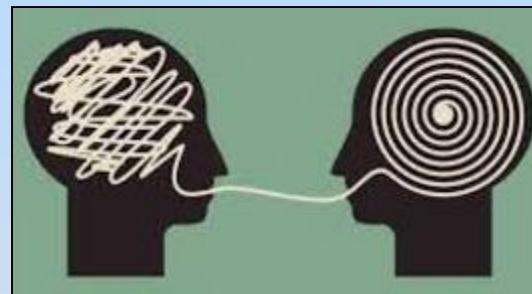
- **What is AI**
- **Four Approaches to AI**
  - Thinking Humanly
  - Acting Humanly
  - Thinking Rationally
  - Acting Rationally
- **Foundations of AI**
- **History of AI**





- **(3) Think Rationally – The “Laws of thought” approach**

- **Rational thinking** – “Right Thinking” with unquestionable reasoning or Logic
- A program can solve any **solvable problem** described in “logical notation” (Ex: TSP, 8-Puzzle, Chess...)
  - **Ex** - Socrates is a man; All men are moral; Socrates is moral

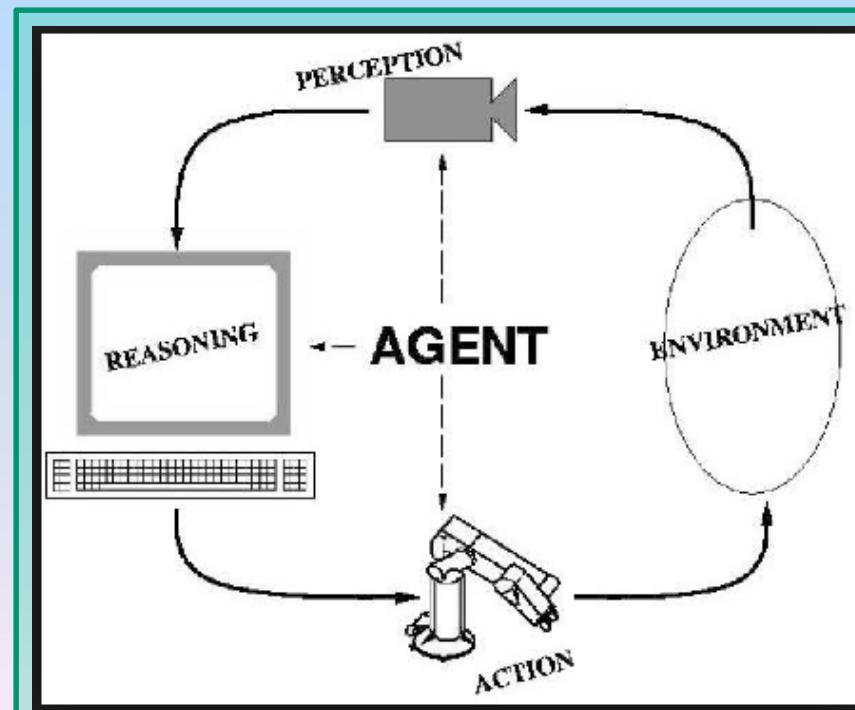


- Two obstacles to this approach –
  - (1) Not easy to represent all problems in logical notations
  - (2) Big difference between solving a problem “in principle” than “in practice”



## • (4) Act Rationally – The “Rational Agent” approach

- **Agent** – “One that Acts” – It **perceives** the env. through **sensors** & acts on the env. through **actuators** (*Ex: Robot, Auto-pilot, Vacuum cleaner...*)
- **Rational Agent** – The Agent that acts to achieve the “best outcome”
  - In case of uncertainty, the “best expected outcome”
- All the skills needed for Turing test also applies here

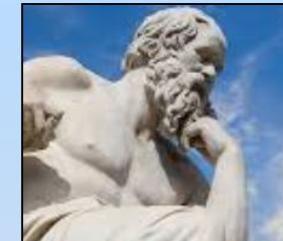


# Foundations of AI

- **Following disciplines** have contributed to ideas, viewpoints & techniques to AI

- **Philosophy**

- Made AI conceivable with the idea that “Mind is in someway like a machine”
- It operates based on “knowledge”
- “Thought” can be used to choose “action”



$$\begin{aligned} \sin \alpha &= BC = \frac{a}{c}, & \sin \alpha &= \text{opposite side} \\ \cos \alpha &= CB = \frac{b}{c}, & \cos \alpha &= \text{adjacent side} \\ \tan \alpha &= AB = \frac{a}{b}; & \tan \alpha &= \text{opposite side / adjacent side} \\ \operatorname{cosec} \alpha &= AO = \frac{c}{a}; & & \end{aligned}$$
$$\begin{aligned} \sin^2 \alpha + \cos^2 \alpha &= 1, & \sin^2 \alpha + \cos^2 \alpha &= 1 - \tan^2 \alpha \\ \sin^2 \alpha &= 1 - \cos^2 \alpha = 1 - \frac{b^2}{c^2}, & \sin^2 \alpha &= \frac{a^2}{c^2} \\ \sin \alpha &= \sqrt{1 - \frac{b^2}{c^2}} = \frac{a}{\sqrt{c^2 - b^2}}; & & \end{aligned}$$
$$360^\circ = 4 \pi, 180^\circ = \pi, \quad \begin{aligned} \sin^2 \alpha &= \cos^2 \alpha = 1 - \frac{b^2}{c^2}, & \sin \alpha &= \sqrt{1 - \frac{b^2}{c^2}} \\ \cos^2 \alpha &= \sin^2 \alpha = 1 - \frac{a^2}{c^2}, & \cos \alpha &= \sqrt{1 - \frac{a^2}{c^2}} \\ \cos \alpha &= \sqrt{1 - \frac{a^2}{c^2}} = \frac{b}{\sqrt{c^2 - a^2}}, & & \end{aligned}$$
$$\begin{aligned} \alpha &= 180^\circ - \beta, & \alpha &= 180^\circ - \beta \\ \sin \alpha &= \sin(180^\circ - \beta) = \sin \beta, & \sin \alpha &= \text{opposite side} \\ \cos \alpha &= -\cos(180^\circ - \beta) = -\cos \beta, & \cos \alpha &= \text{adjacent side} \\ \tan \alpha &= \tan(180^\circ - \beta) = -\tan \beta; & \tan \alpha &= \text{opposite side / adjacent side} \end{aligned}$$
$$\begin{aligned} x &= \frac{\pi}{2} - \alpha, & x &= \frac{\pi}{2} - \alpha \\ \sin x &= \cos \alpha, & \sin x &= \text{opposite side} \\ \cos x &= \sin \alpha, & \cos x &= \text{adjacent side} \\ \tan x &= \operatorname{cosec} \alpha, & \tan x &= \text{opposite side / adjacent side} \end{aligned}$$
$$\begin{aligned} x &= \frac{\pi}{2} + \alpha, & x &= \frac{\pi}{2} + \alpha \\ \sin x &= -\cos \alpha, & \sin x &= \text{opposite side} \\ \cos x &= -\sin \alpha, & \cos x &= \text{adjacent side} \\ \tan x &= \operatorname{cosec} \alpha, & \tan x &= \text{opposite side / adjacent side} \end{aligned}$$
$$\begin{aligned} x &= \pi - \alpha, & x &= \pi - \alpha \\ \sin x &= -\sin \alpha, & \sin x &= \text{opposite side} \\ \cos x &= -\cos \alpha, & \cos x &= \text{adjacent side} \\ \tan x &= \operatorname{cosec} \alpha, & \tan x &= \text{opposite side / adjacent side} \end{aligned}$$
$$\begin{aligned} x &= \pi + \alpha, & x &= \pi + \alpha \\ \sin x &= \sin \alpha, & \sin x &= \text{opposite side} \\ \cos x &= \cos \alpha, & \cos x &= \text{adjacent side} \\ \tan x &= \operatorname{cosec} \alpha, & \tan x &= \text{opposite side / adjacent side} \end{aligned}$$

- **Mathematics**

- Provided “tools” to represent & manipulate logically certain or uncertain statements
- Provided algorithms & computations



- **Economics**

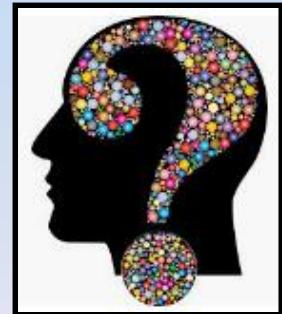
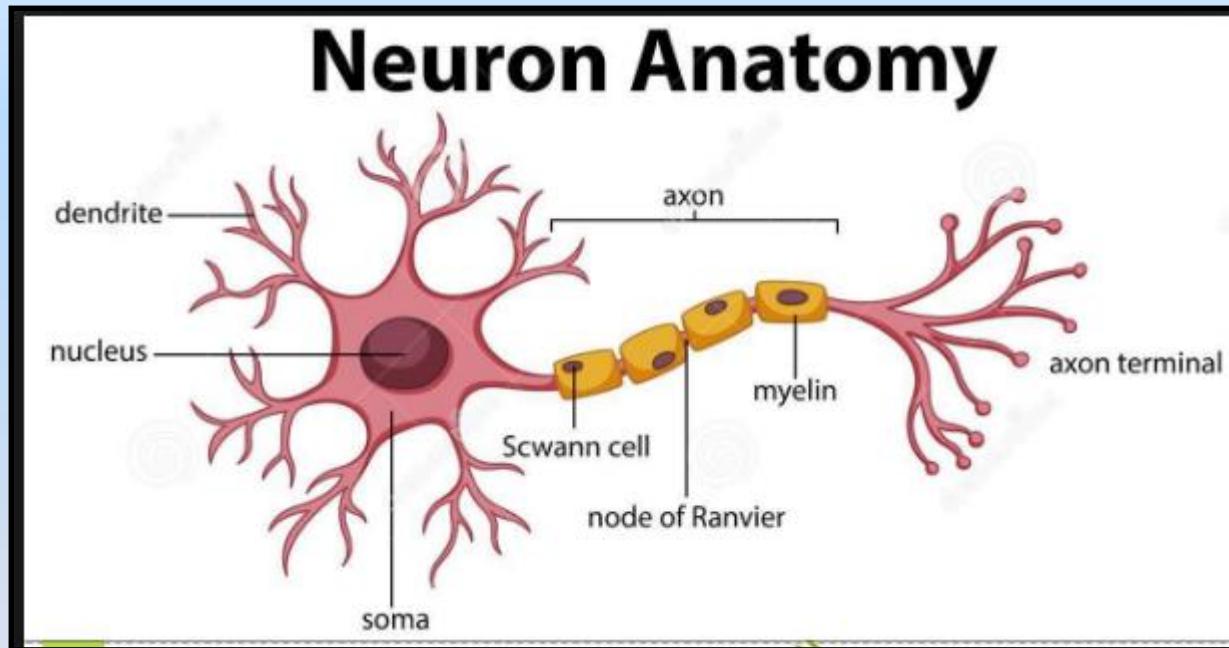
- Helped in making decision that Maximize the expected outcome



# Foundations of AI

- **Neuroscience**

- Provided the knowledge about “how brain works”
- How brain is **similar & different** from computers



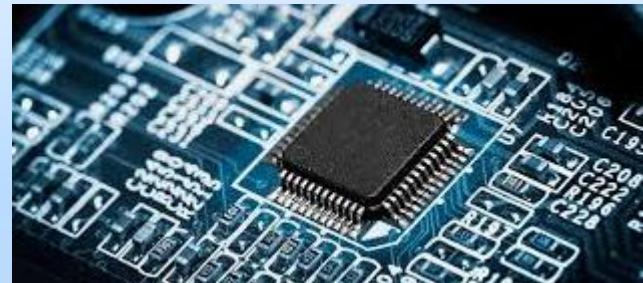
- **Psychology**

- Provided the idea that humans & animals can be considered as “info. processing machines”

# Foundations of AI

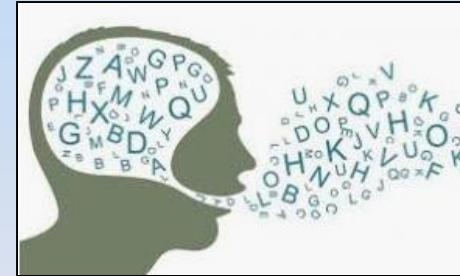
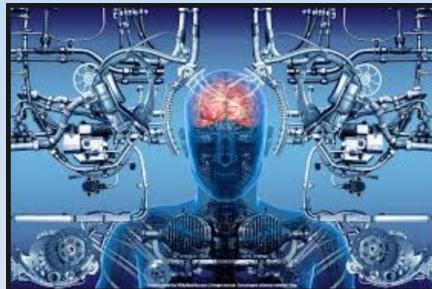
## ▪ Computer Engg.

- Provided highly efficient & powerful machines to “implement AI applications”



## ▪ Control theory and cybernetics

- Helped in designing devices that act optimally based on “feedback from the env.”



## ■ Linguistics

- Provided the idea that language for communication can fit into AI models

# History of AI

## Gestation of AI (1943-55)

- McCulloch & Pitt propose AI based on knowledge of physiology & neurons in brain
- Russel proposed formal analysis
- Turing proposed theory of computation
- Hebb (1949) proposed **Hebbian learning** on connection between neurons

## Birth of AI (1956)

- John McCarthy (Dartmouth College) – official birth place of AI – **2 month workshop**
- Newell & Simon – Created a reasoning program “**Logic Theorist**”

# History of AI

## Early enthusiasm, Great Expectations (1952-69)

- Early success but in limited way by Newell & Simons
- General Problem Solver (GPS) – program designed to imitate human problem-solving
- Physical symbol system hypothesis by Newell & Simons
- Lisp AI programming language by McCarthy
- Neural Network – adalines & perceptrons – by Widrow & Rosenblatt

## A dose of reality (1966-73)

- Simons prediction of extraordinary success could not be achieved
- Computer chess champion & theorem proving by computer took 40 years than prediction of 10 years
- A number of difficulties were encountered while implementing complicated AI applications

# History of AI

## Knowledge-based systems (1969-79)

- General purpose applications (called **weak methods**) were unable to handle **complex problems**
- Powerful applications using **domain-specific knowledge** like **DENDRAL** were needed

## AI becomes an Industry (1980-present)

- R.I – 1<sup>st</sup> commercial Expert System started in 1982 – saved \$40m per year for the org
- DEC's AI group installed 40 E.S.s
- 5<sup>th</sup> Generation project (10 yrs) of Intelligent AI system – started in Japan using prolog

## The return of neural networks (1986-present)

- Back-Propagation – reinvented in 1980's by 4 different books
- Parallel Distributed processing was introduced

# History of AI

## AI adopts the scientific method (1987-present)

- **Revolution** in both content & methodology of AI applications
- **Hidden Markov model (HMMs)** – Dominates AI applications
- **Data-Mining technology** – spread into AI area
- **Bayesian network** – dominates uncertain reasoning AI & ES

## Emergence of Intelligent Agents (1995-present)

- **Intelligent Agents architecture** by **Newell, Laird & Rosenbloom**
- **Internet** – Most important Intelligent agent env.
- **AI systems** have become common in **Web-based applications**
- **Human-level AI (HLAI)** – Machines that think

## Availability of very large data sets (2001-present)

- In many AI applications “**Data**” is more important than “**Algorithm**”
- In recent times, availability of **large data sources** have improved AI applications
- Ex: Images from web, Genomic sequences, words of English

# Course Outline - **Module I**

- Introduction to AI
- AI Problems & AI techniques
- Solving problems by searching
- Problem Formulation
- Intelligent Agents:
  - Structure of Intelligent agents
  - Types of Agents
  - Agent Environments
- Uninformed Search Techniques:
  - DFS, BFS, Uniform cost search, Depth. Limited Search,
  - Iterative Deepening, Bidirectional search
- Comparing Different Techniques



# Chapter 2

## Intelligent Agents

*Dr. Pulak Sahoo*

Associate Professor

Silicon Institute of Technology

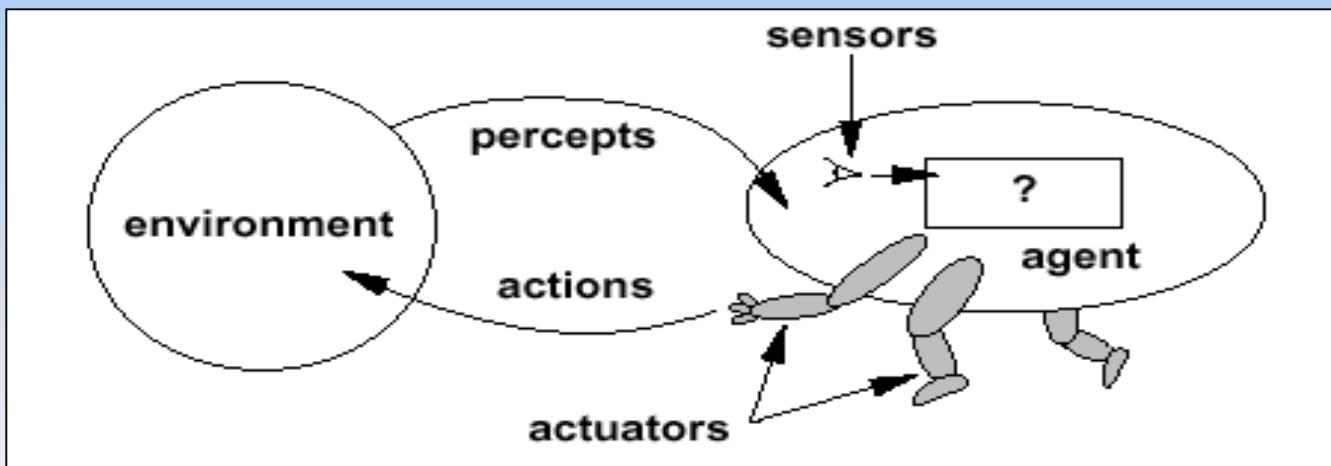


# Intelligent Agent

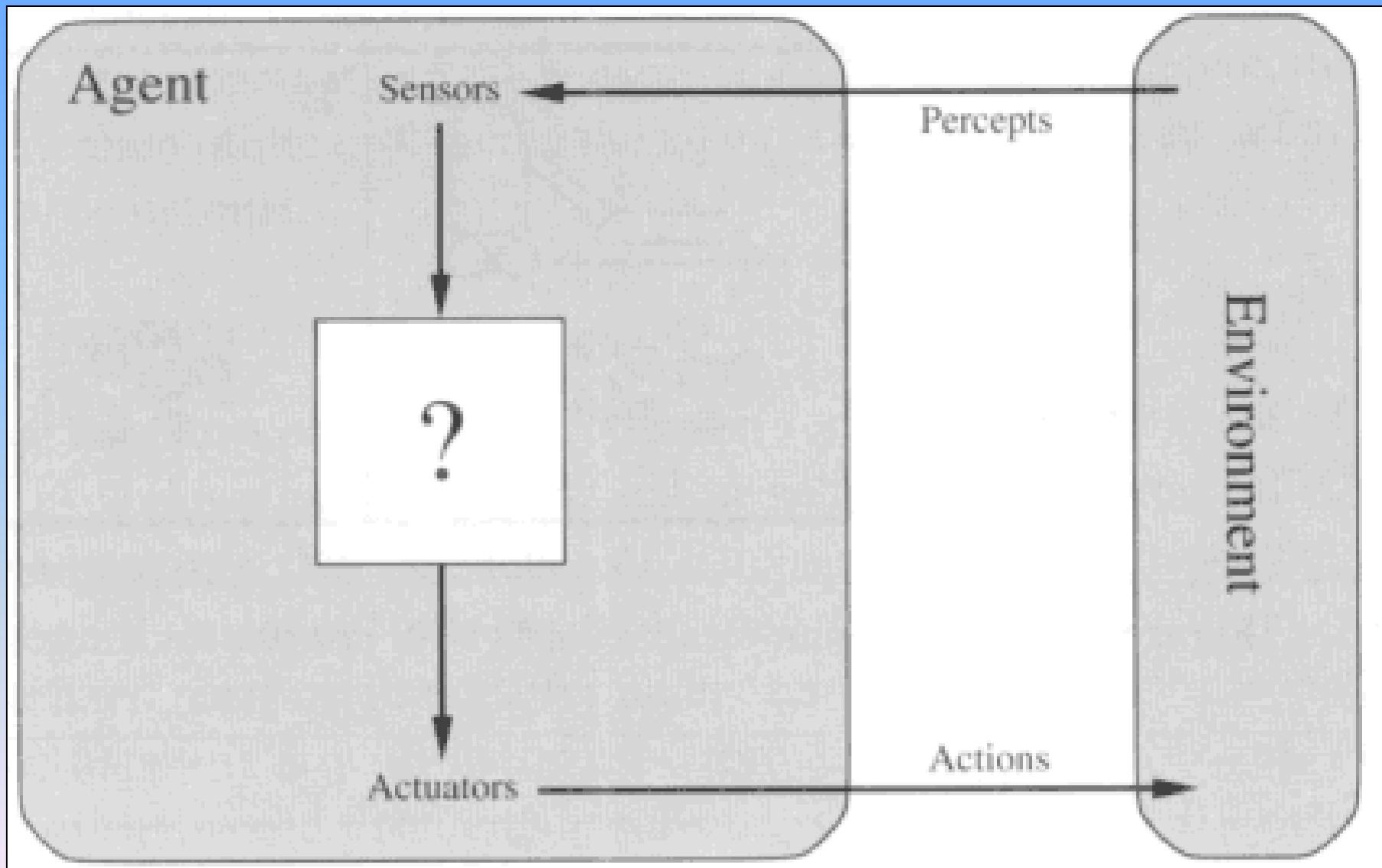
- An **agent** is something that **perceives** its environment through **sensors** & **acts** upon it through **actuators**

- Ex:**

- A **robot** with cameras (sensors) & motors (actuators)
- An **AC** detecting and regulating room temperature
- An **automatic Vacuum Cleaner** detecting dirt & cleaning



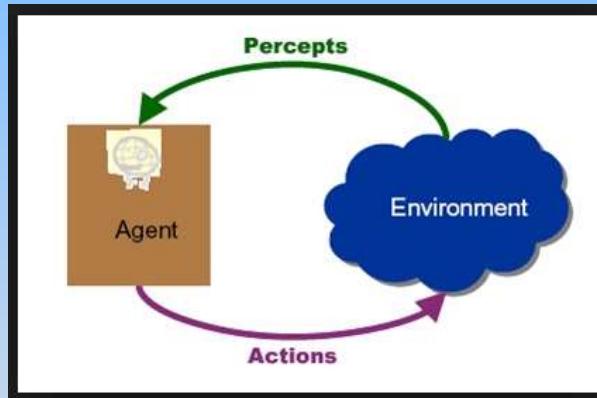
# ★ Intelligent Agent - Diagram



# Some simple terms

## Percept

- Agent's **perceptual inputs from env.** at any given instant



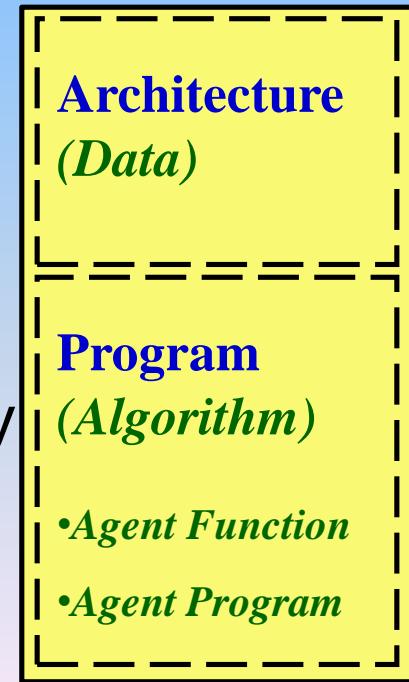
## Percept sequence

- Complete **history** of everything that the agent has ever perceived (*Ex: temp or location reading at short time intervals...*)

# Agent function & program

- Agent's behavior is mathematically described by
  - Agent function
    - A **function** mapping any given percept sequence to an action

Percept Seq.	Action



- Agent's behavior is Practically described by
  - Agent program
    - The real implementation

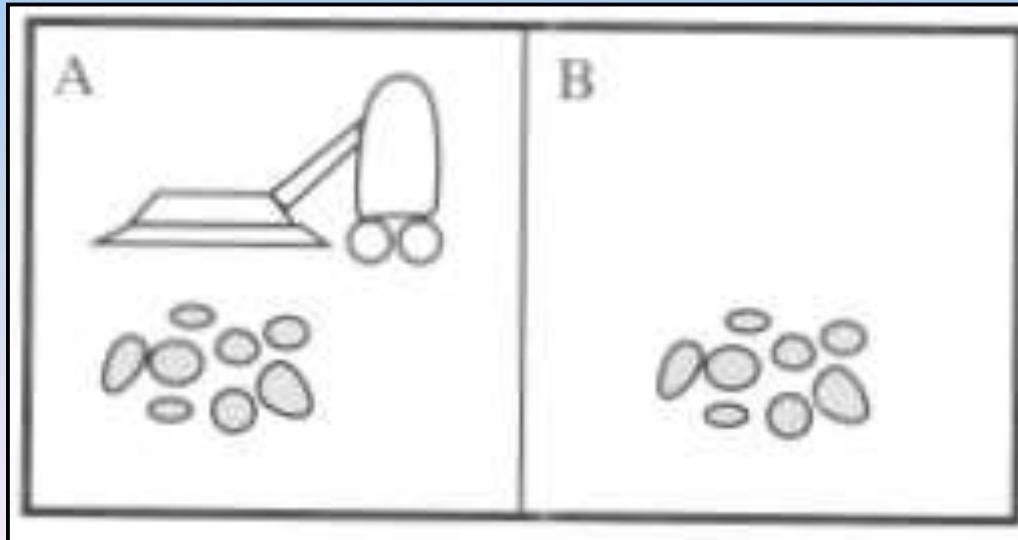


# Vacuum-World

## Perception:

- *Clean or Dirty?*
- *Where it is in? (location A & B)*

## Actions: *Move left, Move right, Suck, Do nothing*





# Vacuum-World

- **Agent function** mapping percept sequences to actions partial tabulation

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

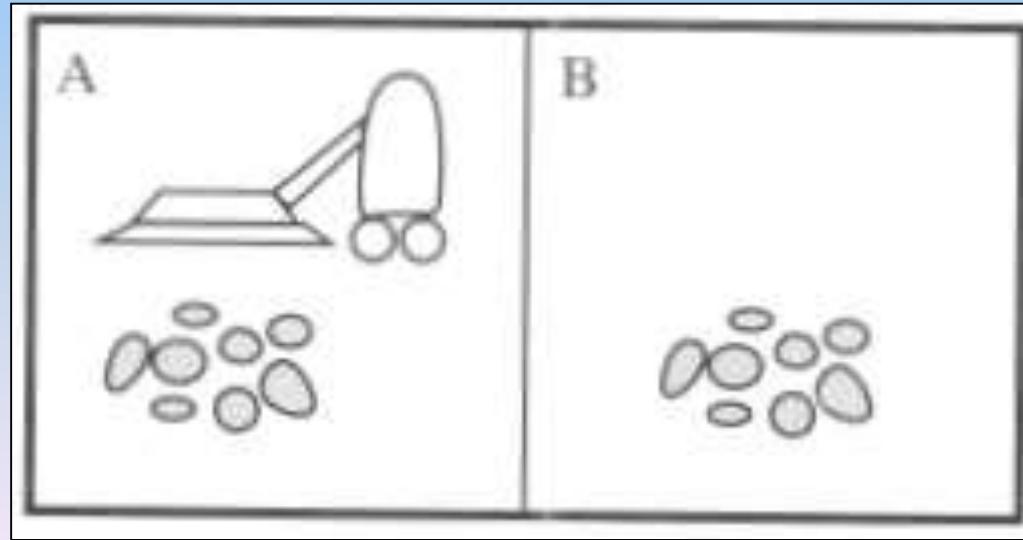
# Agent Program - implementation

**Function Reflex-Vacuum-Agent ([*location*,*state*]) return an action**

**If *state* = Dirty then return *Suck***

**else if *location* = A then return *Right* // *state* = Clean**

**else if *location* = B then return *Left***



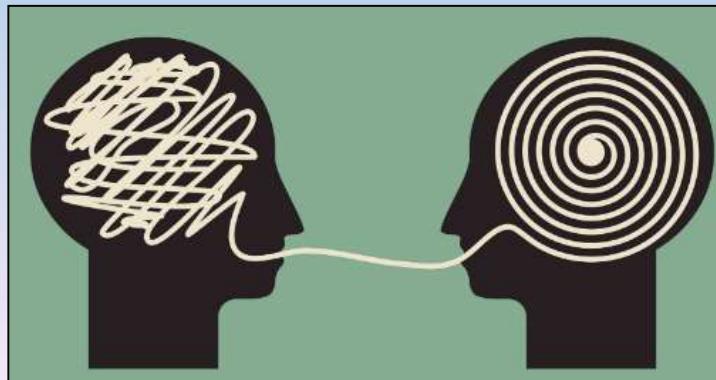
# Concept of Rationality

## ➊ Rational agent

- One that does the right thing (see example below)
- Every entry in the agent function table is **correct (rational)**

## ➋ What is correct?

- The actions that cause the agent to be most successful
- Need ways to measure success



# Performance measure

- **Performance measure**
  - An Objective function determines:
    - Criteria of Success for an agent
      - Ex: Passed the exam if scored 45%
        - Success if 90% clean
- An agent, based on its percepts performs action sequence if result is desirable, it is said to be performing well
- No “universal performance measure” exists for all agents





## ● A general rule:

- Design “**performance measures**” according to
  - “What is required to be achieved” in the env.
  - Rather than “How the agent should behave”

## ● Ex. in vacuum-cleaner

- We want the “**floor clean**”
- We don’t restrict “**how the agent behaves**”



# Rationality

- **What is rational** behavior at any given time depends on Four factors:

1. The “performance measure” defining the “criterion of success”
2. The agent’s “prior knowledge” of the env
3. The “actions” that the agent can perform
4. The agent’s “percept sequence” up to now

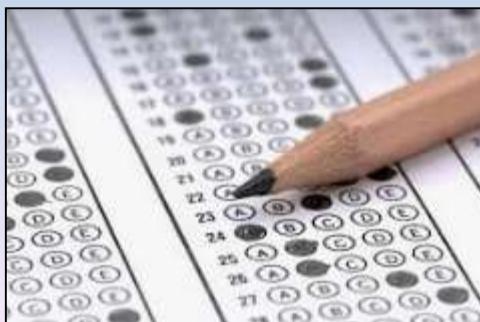


- For each possible “percept sequence”:
  - A **rational agent** should select
    - An action expected to “**maximize performance**” with whatever “**built-in knowledge**” the agent has

---

- Ex.** An exam

- Maximize marks, based on the given questions & own knowledge*



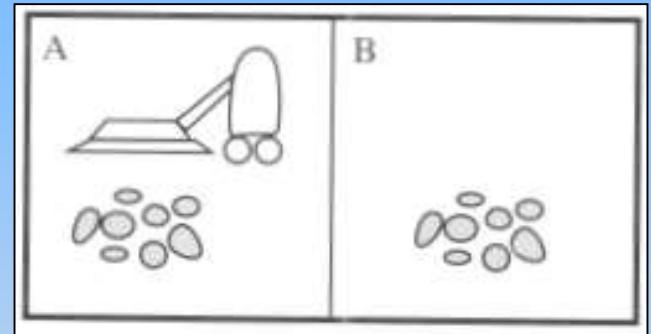
# Example rational agent – Vacuum Cleaner

## Performance measure

- Awards “one point” for each “clean square” at each time step, measured over 10000 time steps

## Prior knowledge about the env

- The “geography” of the env
  - Only two squares
- The “effect” of the actions



## Actions it can perform

- **Left, Right, Suck & NoOp**

## Percept sequences

- “Where” is the agent? (A or B)
- Whether the “location contains dirt”? (Dirty or Clean)

## *Under this circumstance, the agent is rational*

# Omniscience Agent: different from rational

- Knows the actual outcome of it's actions in advance
- This is impossible in real world
- Ex** – *While crossing an empty street, man dies of the fallen cargo door from an aircraft → was the man irrational?*
- Outcome depends only on current percept not past percept sequence*
- Rationality** maximizes - *Expected performance*
- Perfection** maximizes - *Actual performance*
- Rational agents are not omniscient

# Learning Agent : different from rational

- A **rational agent** depends on “current percept” as well as “past percept sequence”
  - This is called learning
  - After “*experiencing an episode*”, the agent should adjust it's behaviors to “*perform better*” next time

# Autonomous Agent

- If an agent just relies on the “prior knowledge of its designer” rather than
  - its “own percepts” then the agent lacks **autonomy**
- A **rational agent** is **autonomous** - it should learn to compensate for partial or incorrect prior knowledge

## Software Agents

- In some cases the env is not real world but “artificial”
  - *Ex: flight simulator, video games, Internet*
- Those agents working in these env.s are called “**Software agents**” (**softbots**) - *All parts of the agent are software*

# Task environment (PEAS Description)

- **Task env.** describes the problem While the **rational agent** provides the solution
  - In **designing** an **agent**, the first step is to **describe** the **task env.** fully
- 
- **Specifying the task env (PEAS description)-Ex-Automated taxi driver**
- **Performance** (the measure success)  
How to judge the performance of a automated driver?  
**Factors** - Reaching correct destination, Minimizing fuel consumption, Trip time, Non-violations of traffic laws, Maximizing the safety etc.
  - **Environment** (the external world in which the agent operates)  
A taxi must deal with **Road condition**, **Traffic lights**, other **Vehicles**, **Pedestrians**, **Animals**, **Road works** etc. & also interact with the **customer**
  - **Actuators** (Implements agents action on the env) – **the output**  
Provides control over the **steering**, **gear**, **brake** & **communicates** with the customer
  - **Sensors** (Reads or takes inputs from the env)  
Detect other vehicles, road situation (camera), **GPS** to know where the taxi is etc...

# Task environments - Automated taxi driver

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, Fast, Legal, Comfortable trip, Maximize profit	Roads, Other Traffic, Pedestrians, Customers	Steering, Accelerators, Brake, Signal, Horn, Display	Cameras, Sonar, Speedometer, Accelerometer, Engine Sensors, Keyboard

PEAS Description of the task environment for an Automated Taxi

## Properties of Task environments

- Fully observable vs. Partially observable
  - If an **agent's sensors** give it access to the **complete state of the env** at each point in time then the env is fully observable
    - **Sensors** detect **all aspects relevant to the choice of action**

- **Partially observable** - An env is partially observable because of noisy & inaccurate sensors or missing sensor data
- Ex: A local dirt sensor of the cleaner cannot tell squares are clean or not

Agent type	P	E	A	S
Medical Diagnosis System ★	Healthy patient, minimize costs, lawsuits	Patient, Hospital Staff	Display questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's Answers.
Interactive English Tutor	Maximize student's score on test.	Set of students, testing agency.	Display exercises, suggestions, corrections	Typed words.
Satellite Image Analysis System ★	Correct categorization of images	Downlink from orbiting satellite	Display categorization of scene.	Colour pixel arrays.
Refinery Controller	Maximize purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical 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,

# Task Env Examples- PEAS description

Agent	Performance Measure	Environment	Actuators	Sensors
Playing Soccer	Score, Injuries, Team work.	Players, Referees, Field, Crowd, Goals, Ball.	Strength, Stamina, Coordination.	Eyes, Ears, Mouth, Ears, Touch.
Exploring Titan	Underwater mobility, Safety, Data, Navigation.	Shuttle, Rover, Atmosphere, Surface, Ocean.	Communication, Sustainability, Reliability.	Camera, GPS, Temperature, Pressure.
AI Book Shopping	Prices, Ease of site, Shipping time.	Websites, Internet, PC.	Correct Information, User.	Pictures, Information, Eyes.
Playing Tennis	Scoring, Stamina, Team work, Strategy.	Players, Referees, Crowd, Net, Court, Ball.	Strength, Stamina, skill Coordination.	Eyes, Footwork.
Practicing Tennis	Stamina, Lowering missed balls.	Player, Wall, Racket, Ball.	Stance, Racket Placement, Speed.	Eyes, Skill, Footwork.
High Jump	Form, Height, Landing.	Height bar, Padded Mat, Judge, Field.	Speed, Form, Leg Strength, Flexibility.	Eyes, Touch.
Knitting a Sweater	Correct Dimension, Reducing mistakes.	Yarn, Needles, Instructions, Room.	Speed, Yarn type, Sweater size, Precision.	Eyes, Hands.
Auction Bidding	Winning, Paying lowest price.	Opponents, Item, Auctioneer.	Budget, Item Value, Eagerness.	Eyes, Ears, Mouth, Knowledge of item.

# Properties Task environments

## ● Deterministic vs. Stochastic

- Next state of the env Completely determined by the Current state & the actions, then the env is **Deterministic**, otherwise, it is **Stochastic**
- **Auto-Pilot Car** - Stochastic because of some unobservable aspects  
→ noise or unknown

## ● Episodic vs. Sequential

- An episode - “A single pair of perception & action”
- Eposodic - The “quality of an agent’s action” does not depend on “other episodes” (*Every episode is independent of other*) – Ex:  
*Smoke detection*
- **Episodic env** is **simpler** - The agent does not need to think ahead
- Sequential – “Current action” may affect “all future decisions”
  - **Ex.** *Taxi driving & Chess*

# Properties Task environments

## Static vs. Dynamic

- **Dynamic env** – is changing over time (*Ex: The no. of people in street*)
  - **Static env** – don't change with time (*Ex: The destination*)
  - **Semidynamic env** - env is not changed over time, but the agent's performance score does (Soccer player)
- 

## Discrete vs. Continuous

- **Discrete env** - There are a limited no. of distinct states, clearly defined percepts & actions (*Ex: Vacuum World*)
  - **Continuous env:** *Taxi driving*
- 

## Single agent VS. Multiagent

- **Single agent** - *Playing a crossword puzzle*
- **Two agent** (Competitive multi-agent env) - *Chess playing*
- Cooperative multi-agent env – *Automated tutors* (english & comm skill)

# Properties Task environments

## Known vs. Unknown

- This distinction refers not to the env itself but to the agent's (or designer's) state of knowledge about the env
- **Known env** - The outcomes for all actions are known (Ex: *Solitaire*)
- **Unknown env** - the agent will have to learn how make good decisions (Ex: *new video game*)

# Structure of agents

Agent

Architecture  
(Components)

Program  
(Algorithm)

- Agent Function
- Agent Program

# Structure of agents

## Agent = architecture + program

- **Architecture** = the components of agent (*sensors, actuators..*)
- **(Agent) Program** = the functions (job of AI) that implement the agent actions based on the current percept

## Agent programs

- Input for “**Agent Program**” - only the current percept
- Input for “**Agent Function**” - the entire percept sequence
- The agent must remember all of them
- Can be implemented as a **look up table**

**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*  $\leftarrow$  LOOKUP(*percepts, table*)

**return** *action*



# Vacuum-World

- **Agent function** mapping percept sequences to actions  
partial tabulation

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

# Agent programs

- $P$  = the set of possible percepts (*Ex: 2 (Clean or Dirty)*)
- $T$  = max length of the percept sequence (*Ex- 150*)
  - The total number of percepts it receives
- Size of the look up table  $\sum_{t=1}^T |P|^t$  (*Ex-  $2^1 + 2^2 + \dots + 2^{150}$* )

- *Ex: Chess*
  - $P=10, T=150$
  - *Look up table contain  $>10^{150}$  entries*
- Despite of huge size, look up table does “**what we want it to do**”
- The key challenge of AI
  - How to write programs that produces rational behaviour from a small amount of code
  - It is ok to have large amount of table entries

# Chapter 2

## Intelligent Agents

***Dr. Pulak Sahoo***

Associate Professor

Silicon Institute of Technology



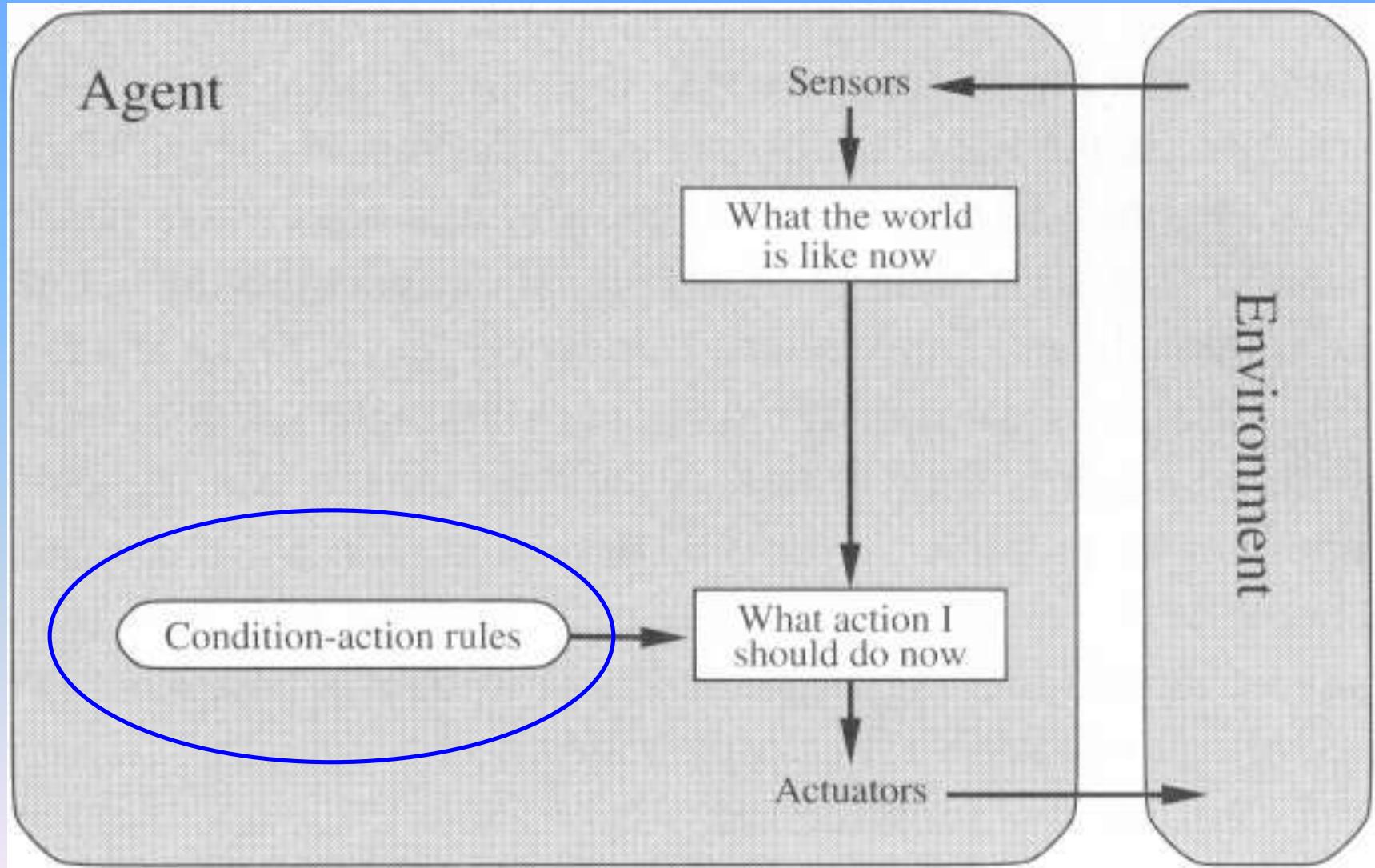
# Types of Agent programs



## Four types

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

# 1. Simple reflex agents



# 1. Simple reflex agents (for fully observable env)

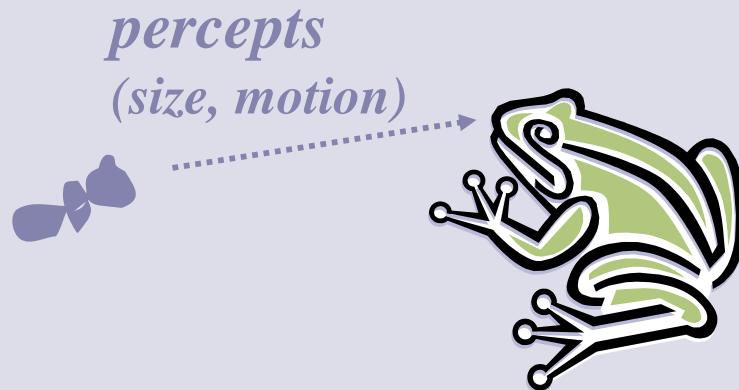
Lab

- It uses just **condition-action rules**
  - The rules are in form of “if ... then ...”
  - Efficient but have narrow range of applicability
  - Because knowledge sometimes “cannot be stated explicitly”
  - Works only if the environment is “fully observable”

**function** SIMPLE-REFLEX-AGENT(*percept*) **returns** *action*  
**static:** *rules*, a set of condition-action rules

```
state  $\leftarrow$  INTERPRET-INPUT(percept) // From percept get state  
rule  $\leftarrow$  RULE-MATCH(state, rules) // From State get rule  
action  $\leftarrow$  RULE-ACTION[rule] // From rule find action  
return action
```

# A Simple Reflex Agent in Nature



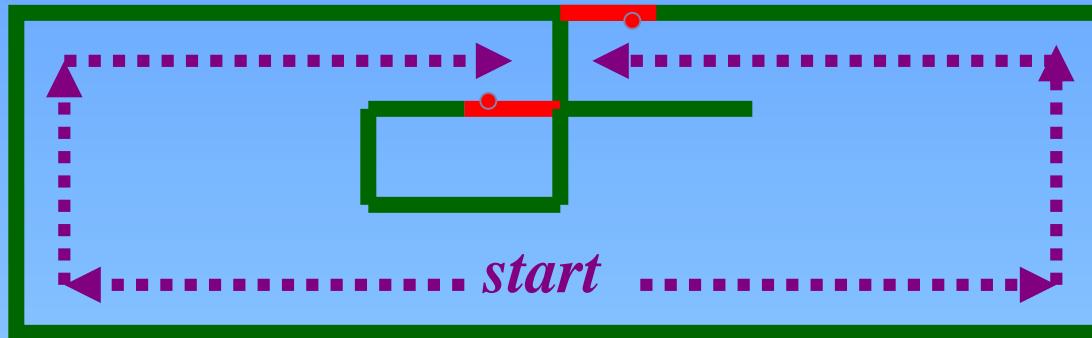
## RULES:

- (1) If small moving object,  
then activate SNAP
  - (2) If large moving object,  
then activate AVOID and inhibit SNAP
- ELSE** (not moving) then NOOP

needed for  
completeness

Action: SNAP or AVOID or NOOP

# Example Reflex Agent With Internal State: Wall-Following

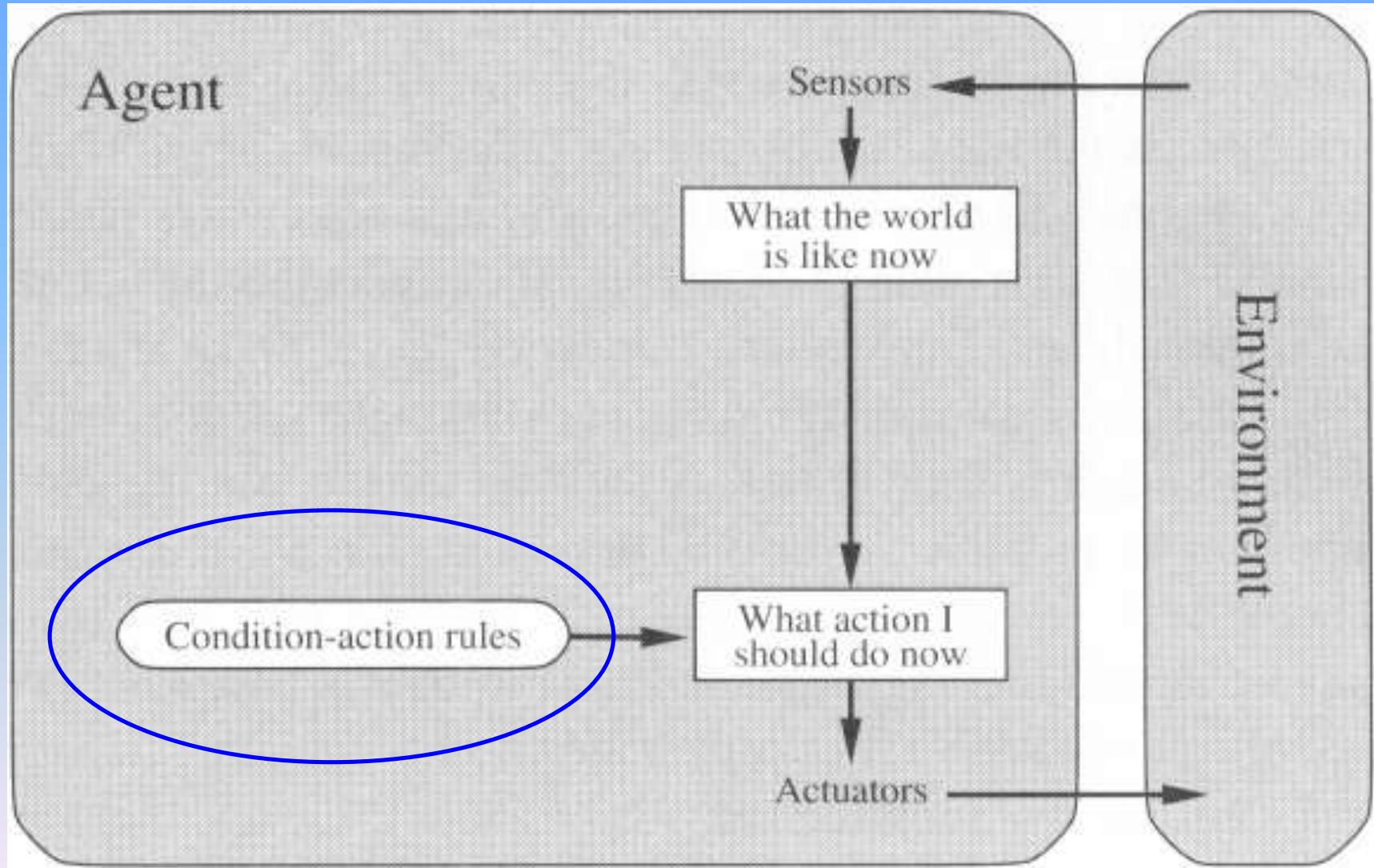


**Actions:** left, right, straight, open-door

**Rules:**

1. If open(left) & open(right) and open(straight) then  
choose **randomly between right & left**
2. If wall(left) & open(right) & open(straight) then **straight**
3. If wall(right) & open(left) & open(straight) then **straight**
4. If wall(right) & open(left) & wall(straight) then **left**
5. If wall(left) & open(right) & wall(straight) then **right**
6. If wall(left) and door(right) and wall(straight) then **open-door**
7. If wall(right) and wall(left) and open(straight) then **straight**
8. (Default) **Move randomly**

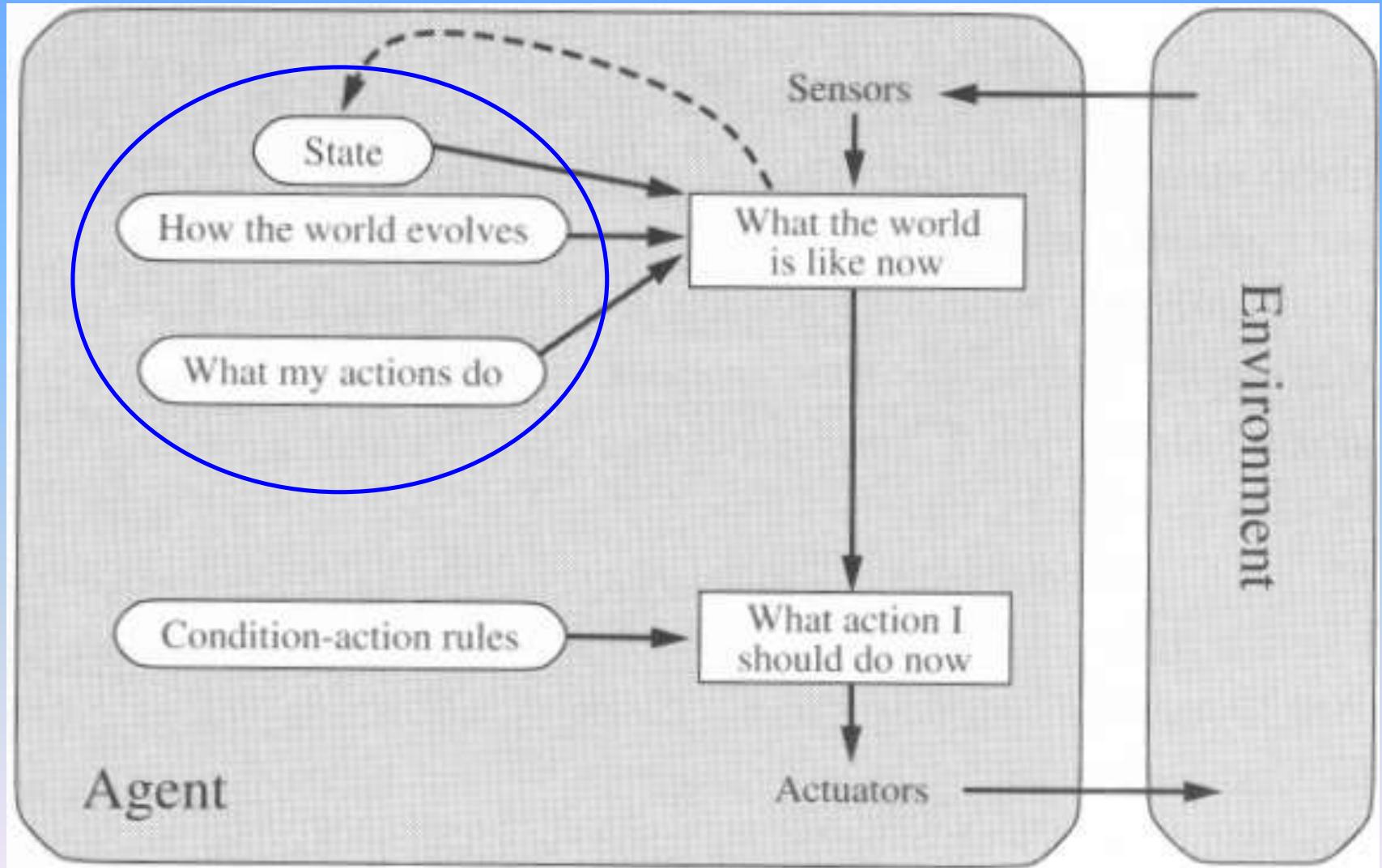
# 1. Simple reflex agents





## 2. Model-based Reflex Agents

(for partially observable env)



## 2. Model-based Reflex Agents (for partially observable env)



- For the world that is “**partially observable**”

- the agent has to keep track of the state of the env.
  - Ex- Driving a car & changing lane (traffic situation is imp)*

- We also require two more types of knowledge

- How the world evolves independent of the agent
- How the agent’s actions affect the world

### Example – Self-driven car

Example Mapping Table

**State** – Road Clear,  
Object ahead, Dead end

IF	THEN
Saw an object ahead, turned right, and it's now <b>road clear</b> ahead	<b>Go straight</b>
Saw an object ahead, turned right, and <b>another object ahead</b>	<b>Slow down</b>
See <b>no objects</b> ahead	<b>Go straight</b>
See <b>dead end</b>	<b>Take U Turn</b>

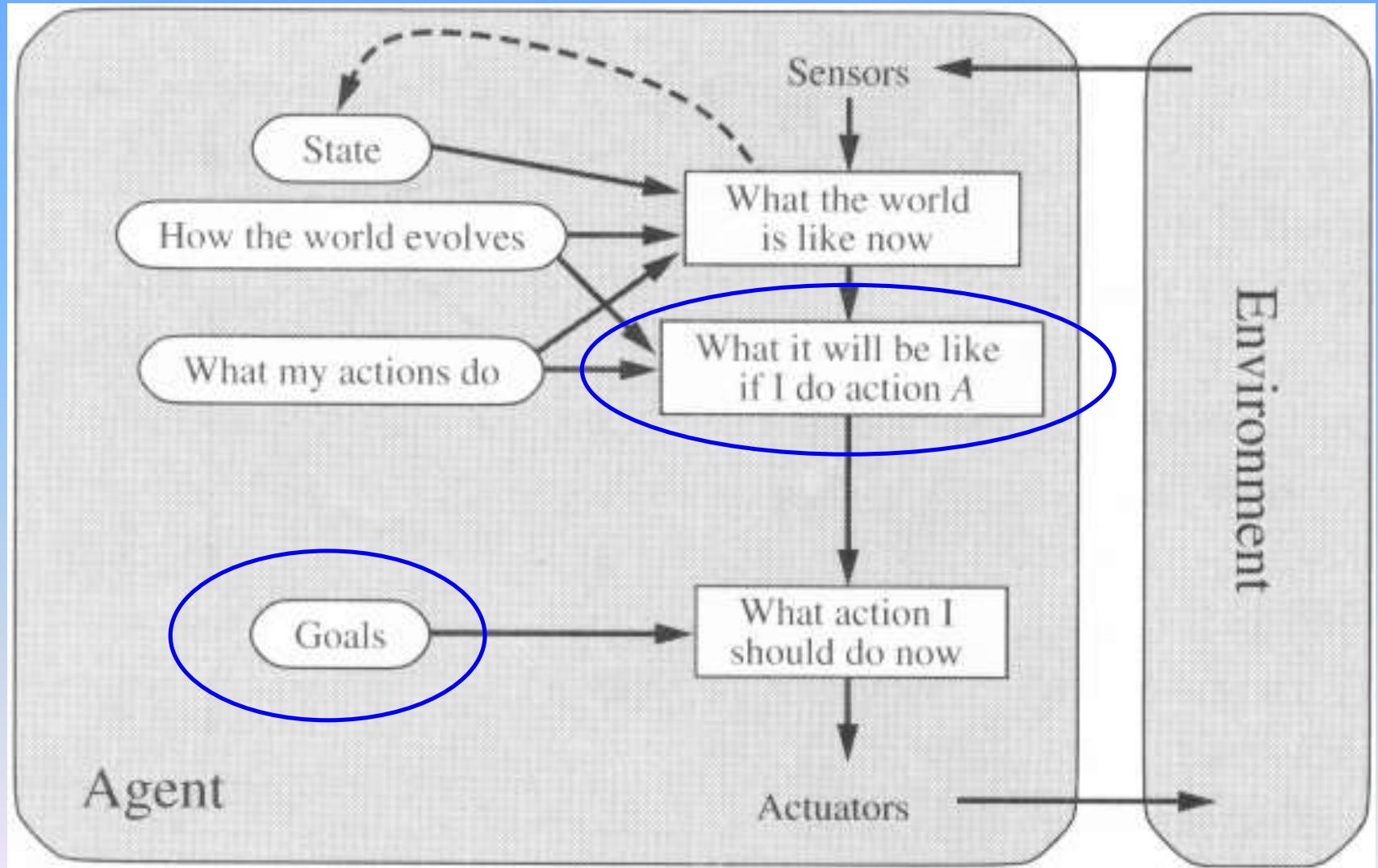
# Model-based Reflex Agents

```
function REFLEX-AGENT-WITH-STATE(percept) returns action
    static: state, a description of the current world state
          rules, a set of condition-action rules
```

```
state  $\leftarrow$  UPDATE-STATE(state, percept) // From current state & percept get next state.
rule  $\leftarrow$  RULE-MATCH(state, rules) // From State get rule.
action  $\leftarrow$  RULE-ACTION[rule] // From rule find the action.
state  $\leftarrow$  UPDATE-STATE(state, action) // From new state & action update next state.
return action
```



### 3. Goal-based agents





### 3. Goal-based agents

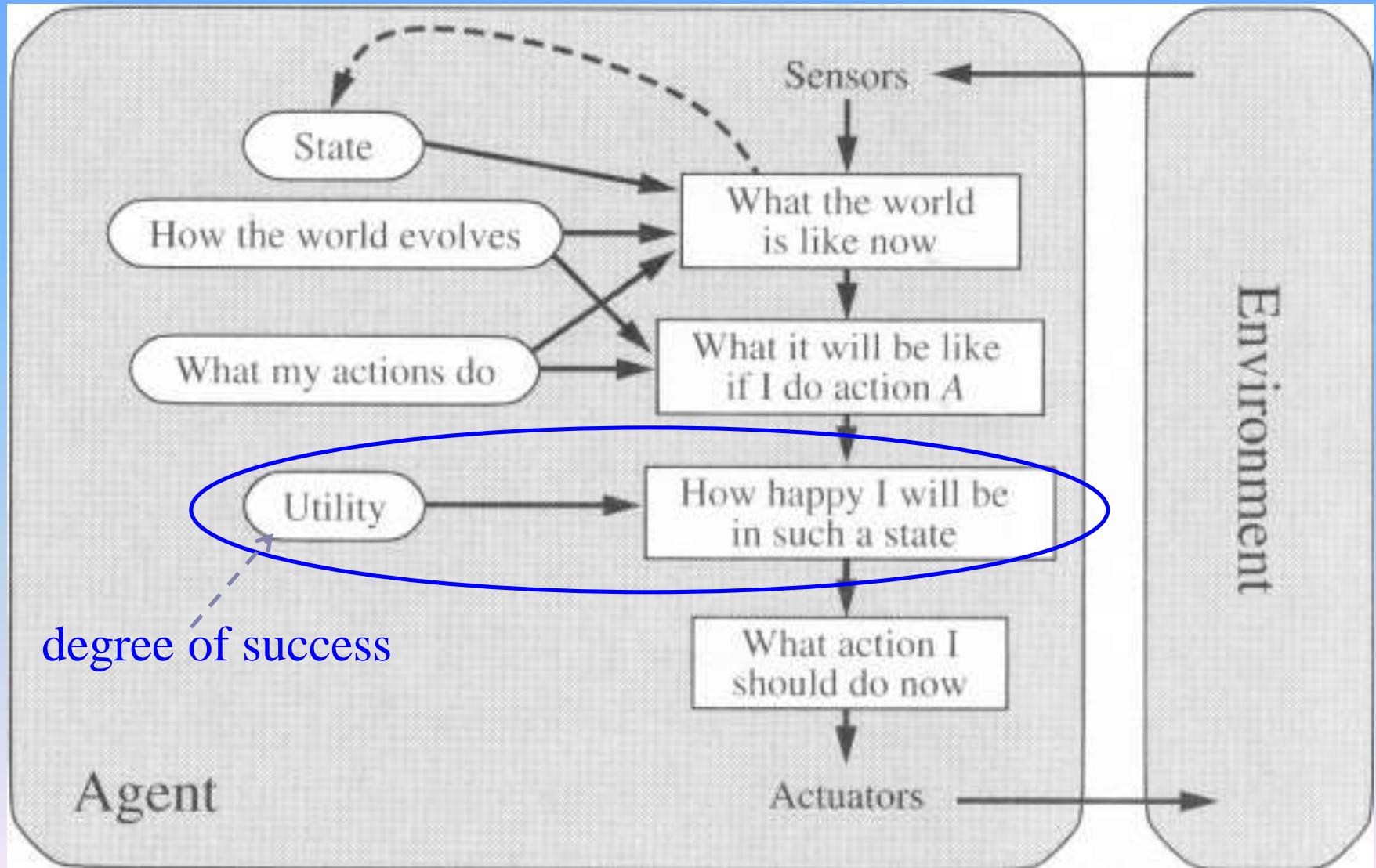
- The **objective** is to achieve the **goal**
- Current state of the env.& **action** are not enough to reach the **goal**
- Need to find out the **action sequences** required to achieve the **goal**
- **Choose Actions** that will lead to the **goal**, based on
  - the **current state**
  - the **current percept**

---

- **Conclusion**
  - Goal-based agents are **less efficient** but **more flexible**
  - **Searching & Planning** (Two other sub-fields in AI )

## 4. Utility-based agents

A2

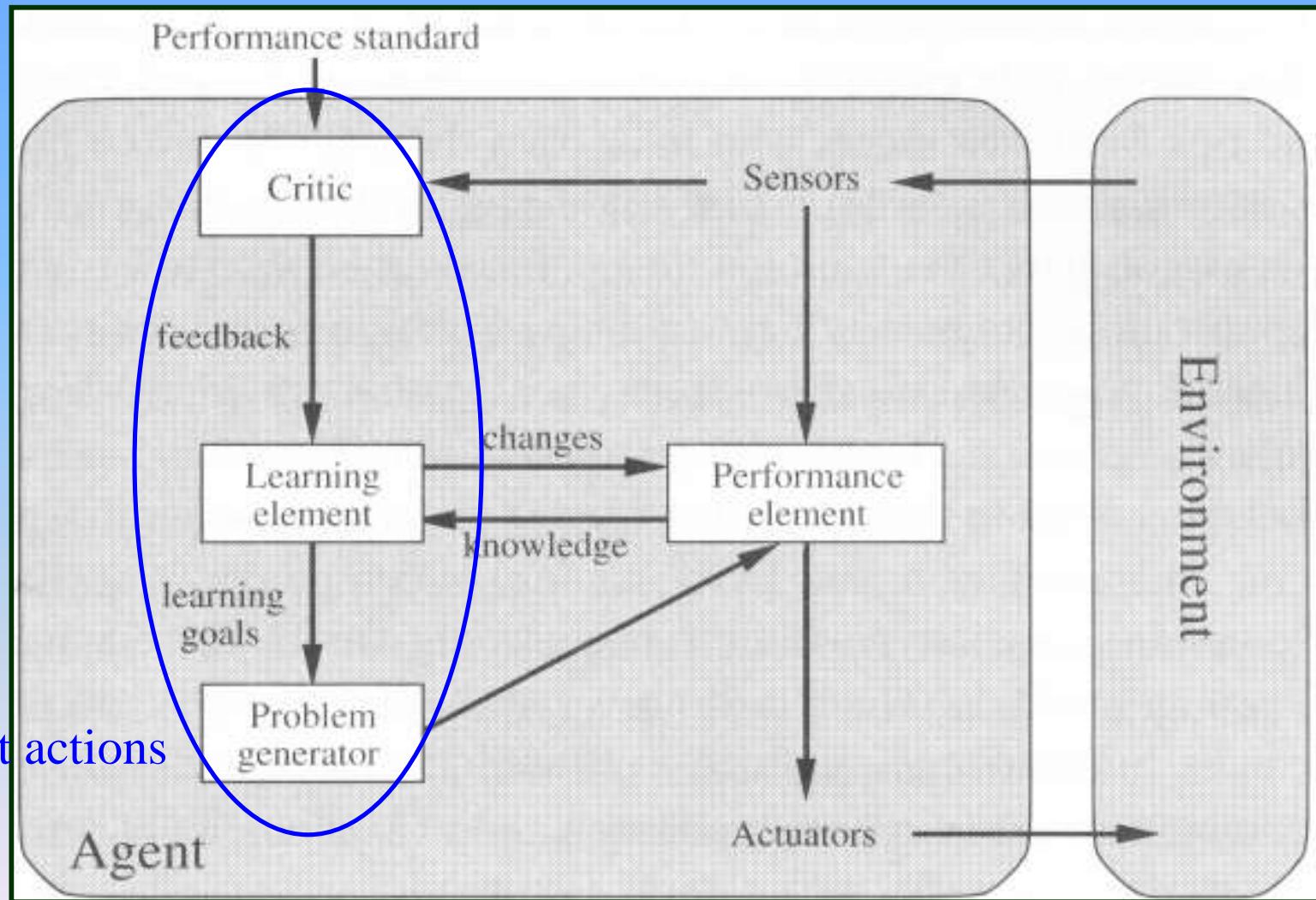


## 4. Utility-based agents

- Reaching the **Goals** is not enough. Achieving **high-quality** behavior (reaching goal in a better approach) is important
- A no. of action sequences required to achieve the goal
  - If **goal** => success, then **utility** => degree of success
- State A** has higher utility if it is preferred more than other states
- Utility** is a function that maps a state onto a real number indicating the degree of success
- Utility** has below advantages:
  - In case of **conflicting goals**, only some of the goals can be achieved
  - Utility** describes the appropriate trade-off
- When there are **several goals & none of them can be achieved with certainty**
  - Utility** provides a way for the decision-making

# Learning Agents

A2



# Learning Agents

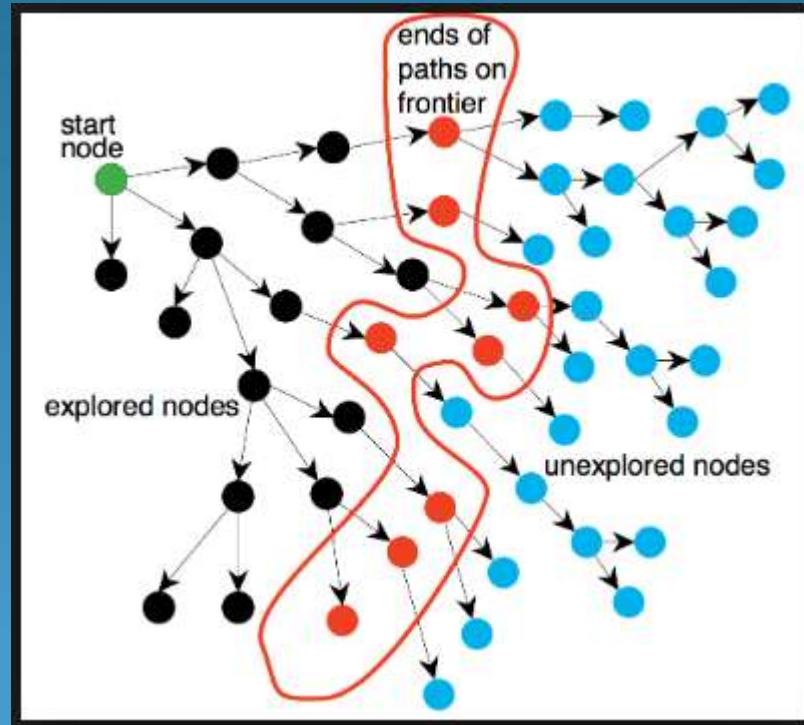
- After an agent is **programmed**, it can not work immediately
  - It **needs to taught**
  - Teach** it by giving it a set of examples
  - Test** it by using another set of examples
  - We then say the agent - **A learning agent**

## Four conceptual components

- Learning element – For Making continuous improvements
- Performance element - Selecting proper actions
- Critic**
  - Tell how well the agent is doing with respect to a fixed performance std
  - Feedback from users
- Problem generator
  - Suggest actions that will lead to new & informative experiences



# Problem solving by search



*Dr. Pulak Sahoo*

Associate Professor

Silicon Institute of Technology

# Contents

- **Introduction**
- **Problem Solving Agents**
  - Goal Formulation
  - Problem Formulation (Defining a problem)
- **Searching**
  - (1) Uninformed (Blind) Search
  - (2) Informed Search
- **Example problems**
  - Traveler's route finding problem (Travelling Romania)
  - Vacuum World
- **Toy problems**
  - 8-Puzzle & 8-Queen

# Contents

- **Example problems**
  - Missionaries & Cannibals
  - **Real-World Problems**
    - Route Finding Problem
- **Uninformed (Blind) Search**
  - BREADTH-FIRST Search
  - DEPTH-FIRST Search
  - UNIFORM-COST Search
  - DEPTH-LIMITED Search
  - Iterative Deepening Depth-First Search
  - Bidirectional Search
- Comparison of Uninformed search techs

# Introduction

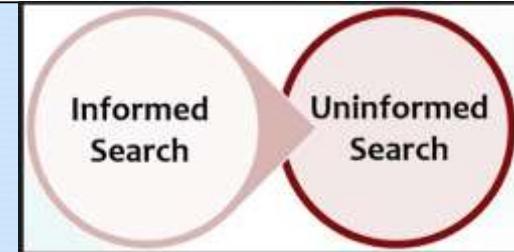
- **Problem Solving Agent**
- **Problem solving by searching:**
  - How an agent find a “sequence of actions” to achieve it’s goal
- **Goal-based Agents: 2 Types**
  - Problem-solving agents (use atomic representation)
  - Planning agents (use advanced structured representation)



# Introduction

- We will discuss :

- A no. of example **problem definitions & their solutions**
- A no. of general purpose **search algorithms** (2 types)
  1. **Uninformed search algorithm** – Have no other info than **problem definition**
  2. **Informed search algorithm** – Guidance provided for finding solution
- In “Simple task envs” – Solution is a “**Fixed sequence of actions**”



# Water Jug problem

**Problem:** You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring mark on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug.

## Solution:

The state space for this problem can be described as the set of ordered pairs of integers  $(x,y)$

Where,

X represents the quantity of water in the 4-gallon jug  $X = 0, 1, 2, 3, 4$

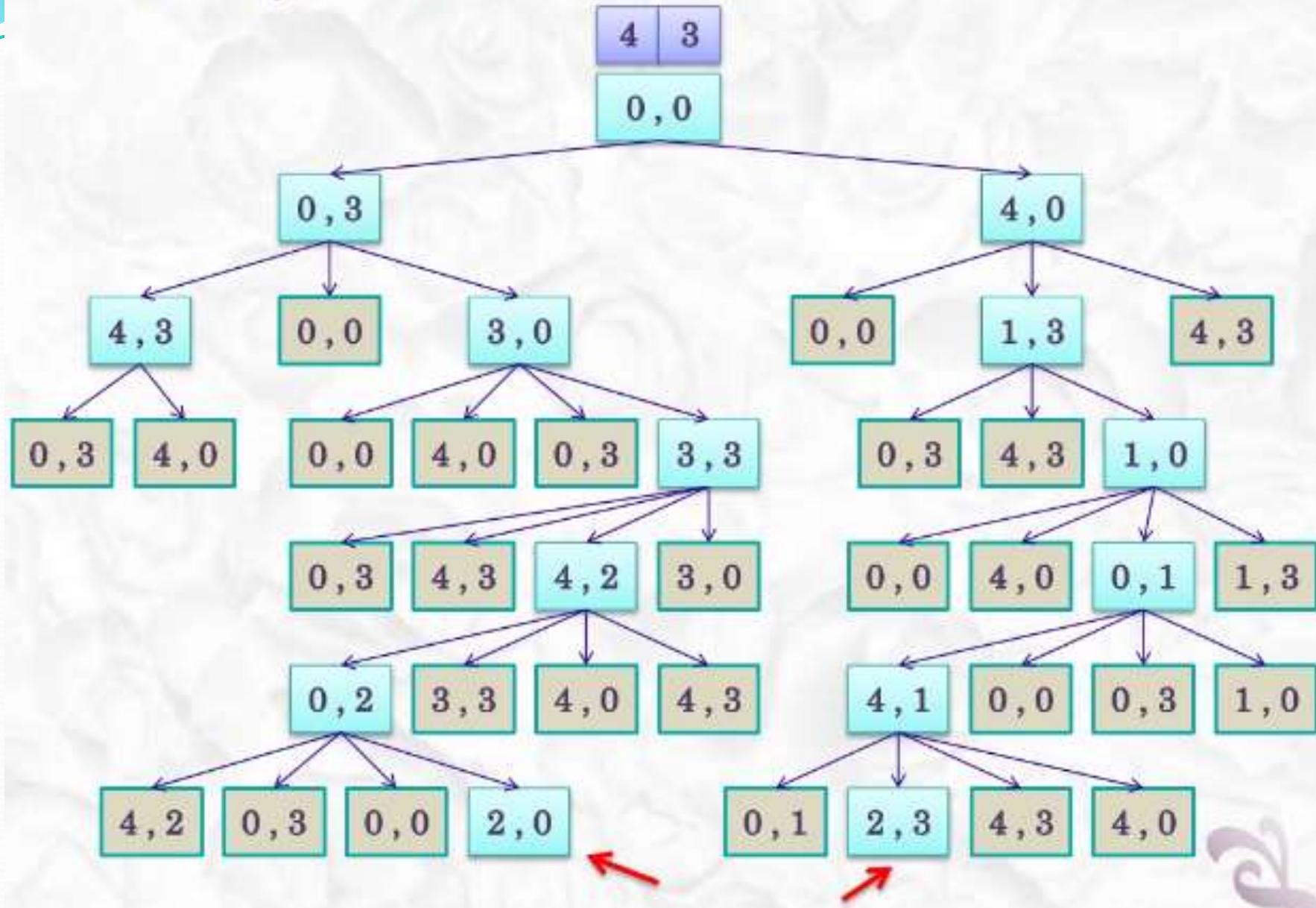
Y represents the quantity of water in 3-gallon jug  $Y = 0, 1, 2, 3$

**Start State:**  $(0,0)$

**Goal State:**  $(2,0)$

Generate production rules for the water jug problem

# THE WATER JUGS PROBLEM - SEARCH TREE



# Water Jug problem

## Production Rules:

Rule	State	Process
1	(X,Y   X<4)	(4,Y) {Fill 4-gallon jug}
2	(X,Y   Y<3)	(X,3) {Fill 3-gallon jug}
3	(X,Y   X>0)	(0,Y) {Empty 4-gallon jug}
4	(X,Y   Y>0)	(X,0) {Empty 3-gallon jug}
5	(X,Y   X+Y>=4 ^ Y>0)	(4,Y-(4-X)) {Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}
6	(X,Y   X+Y>=3 ^ X>0)	(X-(3-Y),3) {Pour water from 4-gallon jug into 3-gallon jug until 3-gallon jug is full}
7	(X,Y   X+Y<=4 ^ Y>0)	(X+Y,0) {Pour all water from 3-gallon jug into 4-gallon jug}
8	(X,Y   X+Y <=3 ^ X>0)	(0,X+Y) {Pour all water from 4-gallon jug into 3-gallon jug}
9	(0,2)	(2,0) {Pour 2 gallon water from 3 gallon jug into 4 gallon jug}

# Water Jug problem - Solution

1

**Initialization:**

Start State: (0,0)

**Apply Rule 2:**  
{Fill 3-gallon jug}

Now the state is (X,3)

2

Current State: (X,3)

**Apply Rule 7:**

Now the state is (3,0)

{Pour all water from 3-gallon jug into 4-gallon jug}

3

Current State : (3,0)

**Apply Rule 2:**  
{Fill 3-gallon jug}

Now the state is (3,3)

4

Current State: (3,3)

**Apply Rule 5:**  
{Pour water from 3-gallon jug into 4-gallon jug until 4-gallon jug is full}

Now the state is (4,2)

5

Current State : (4,2)

**Apply Rule 3:**  
{Empty 4-gallon jug}

Now state is (0,2)

6

Current State : (0,2)

**Apply Rule 9:**  
{Pour 2 gallon water from 3 gallon jug into 4 gallon jug}

Now the state is (2,0)

**Goal Achieved.**

# Traveler's path finding problem

## (Travelling Romania)

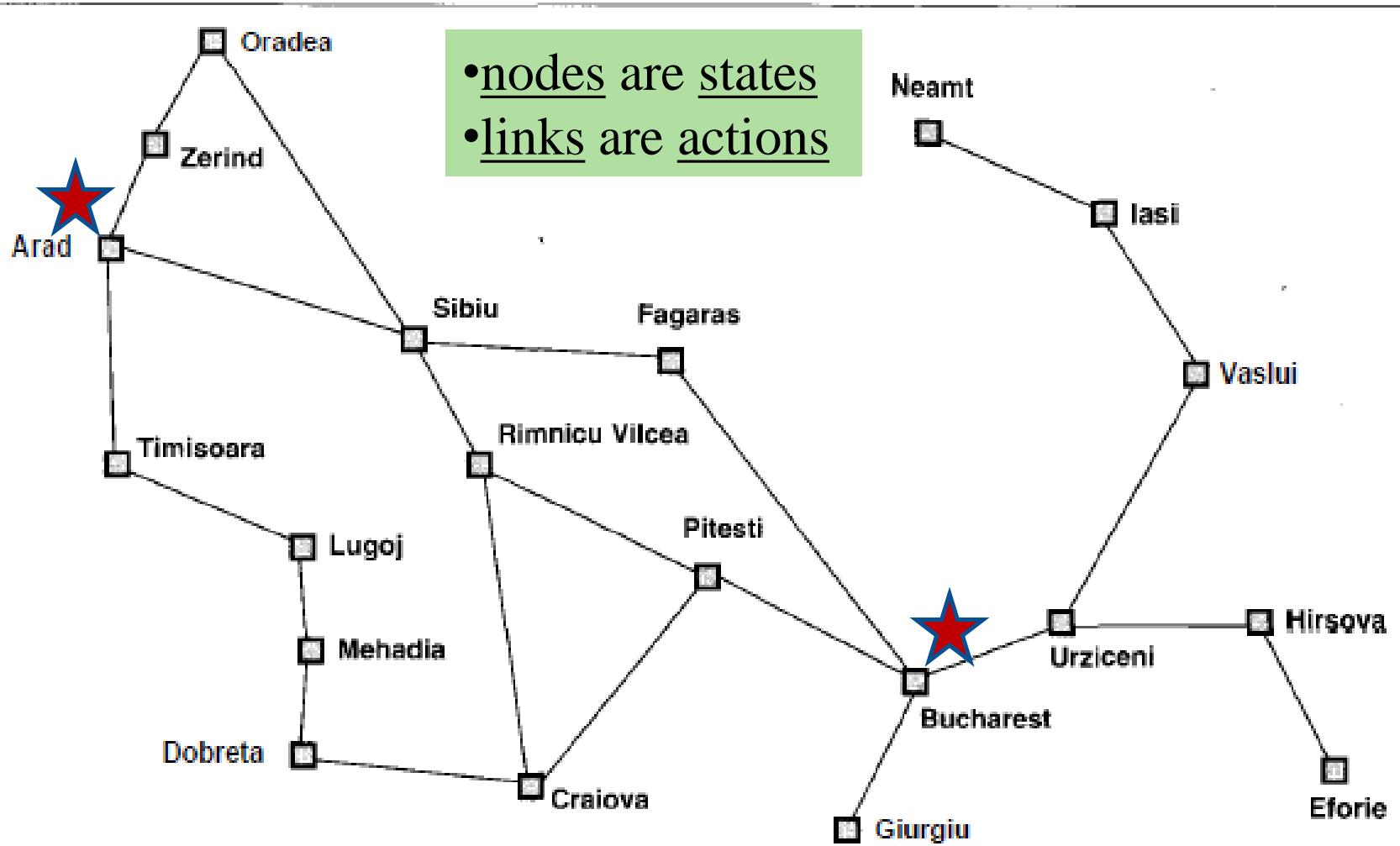


Figure 3.3 A simplified road map of Romania.

# Problem Solving Agents

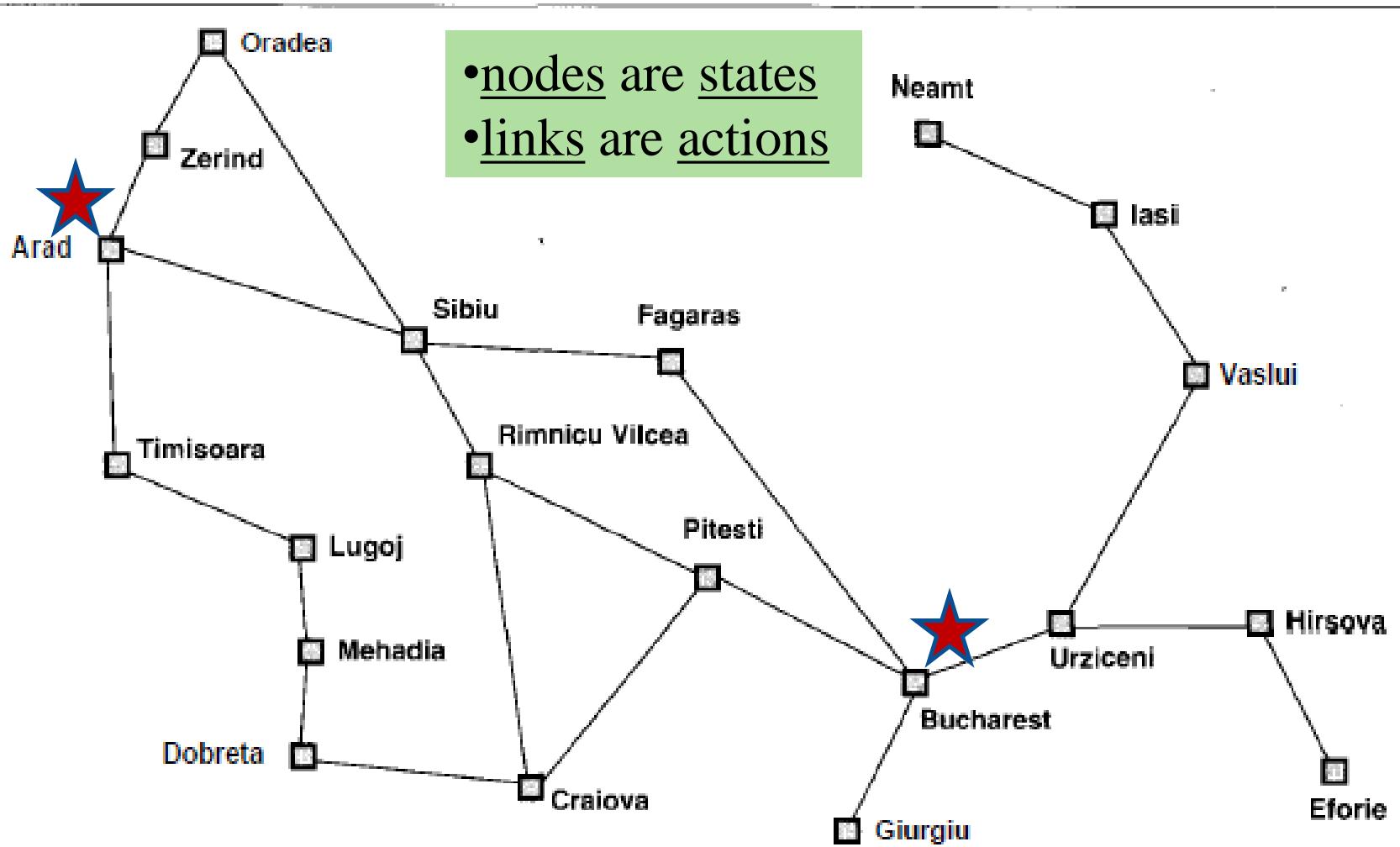
- **Traveler's path finding problem**
  - A traveler is visiting **Romania** with performance measures like *distance traveled, sight seeing, shopping & nightlife*
- **Goal Formulation:**
  - An agent's task is simplified if it can formulate a goal based on current situation & performance measure
  - **Goals limit objectives** of the agent (simplifies the problem)
  - Goal is a “set of world states” (*Cities to be visited by a traveler*)
  - An agent must act to reach a goal state (Reach **Bucharest**)
- **Problem Formulation:**
  - Is the process of deciding what type of **Actions** (*driving*) & **States** (*Cities*) to consider for a given **goal**  
*(Driving from one city to another & reach destination city)*



# Problem Solving Agents

- Let us assume that the env is:
  - Observable** (agent knows **current state** - *current city*)
  - Discrete** (finite set of **actions** to choose from - *connected cities*)
  - Known** (which **states** are reached by which **action** - *having map of cities*)
  - Deterministic** (Each action has only one outcome)
- Then the **solution** to any problem is a **<fixed sequence of actions>**
- The process of looking for a **sequence of actions** that reaches the goal is called **SEARCH**
- When a **solution** is found, the **recommended action sequence** is carried out (called **EXECUTION**)
- Agent Design - **<Formulate -> Search -> Execute>**

# Traveler's path finding problem (Travelling Romania)



**Figure 3.3** A simplified road map of Romania.

# Problem Solving Agents

## Ex – Problem – Getting to Bucharest from Arad (in Romania)

### Step-1: Defining the problem

- A problem can be defined using 5 components

**1. Initial State** – starting state (Ex: In(Arad))

**2. Actions** (available to the agent) – **ACTIONS (s)** – set of actions available in state ‘s’ (Ex: Go(Sibiu), Go(Timisoara), Go(Zerind))

**3. Transition Model** – **RESULT (s, a)** – Returns the “**state reached**” after action ‘a’ in state ‘s’ – (Ex: RESULT(In(Arad), Go(Zerind)) = In(Zerind))

- **State Space** - <Initial state, Actions, Transition model> - A directed graph with nodes are states and links are actions

**4. Goal Test** – Determines if a given state is a goal state (Ex: Goal is {In(Bucharest)})

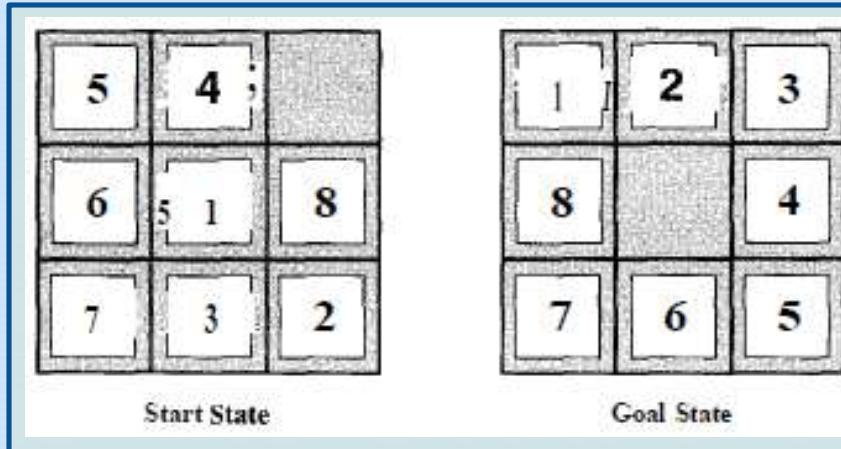
# Problem Solving Agents

## 5. Path Cost – Assigns a “numeric cost” to each path (Ex: Length in km)

- **Path** – A sequence of states connected by a sequence of actions
  - **Step Cost** – sum of costs -  $c(s_1, a, s_2)$  - cost of taking action **a** in state **s<sub>1</sub>** to reach state **s<sub>2</sub>**
  - **Optimal Solution** – Has the lowest path cost among all solutions
- 
- **Step-2: Formulating a problem**
  - **Problem** – Getting to Bucharest from Arad (in Romania)
    - **Formulation** – In terms of <Initial state, Actions, Transition Model, Goal Test, Path Cost> - Is an abstraction not the real thing – Irrelevant details are removed
    - **Ex:** In (Arad) is an abstract description excluding many things like Road condition, Weather, Travel companions, Driving Rules etc..

# Example Problems – 8 - Puzzle

- **State** – A state desc. specifies the location of 8 tiles & the blank space
- **Initial State** – Any state can be the initial state
- **Actions** – Movement of the blank space *Left, Right, Up & Down*
- **Transition Model** – For a given state & action, this returns the resulting state
- **Goal Test** – Checks if the resulting state matches goal state given below
- **Path Cost** – Each step costs 1 (Path cost is the ‘no. of steps in the path’)



# State space of the 8 puzzle problem

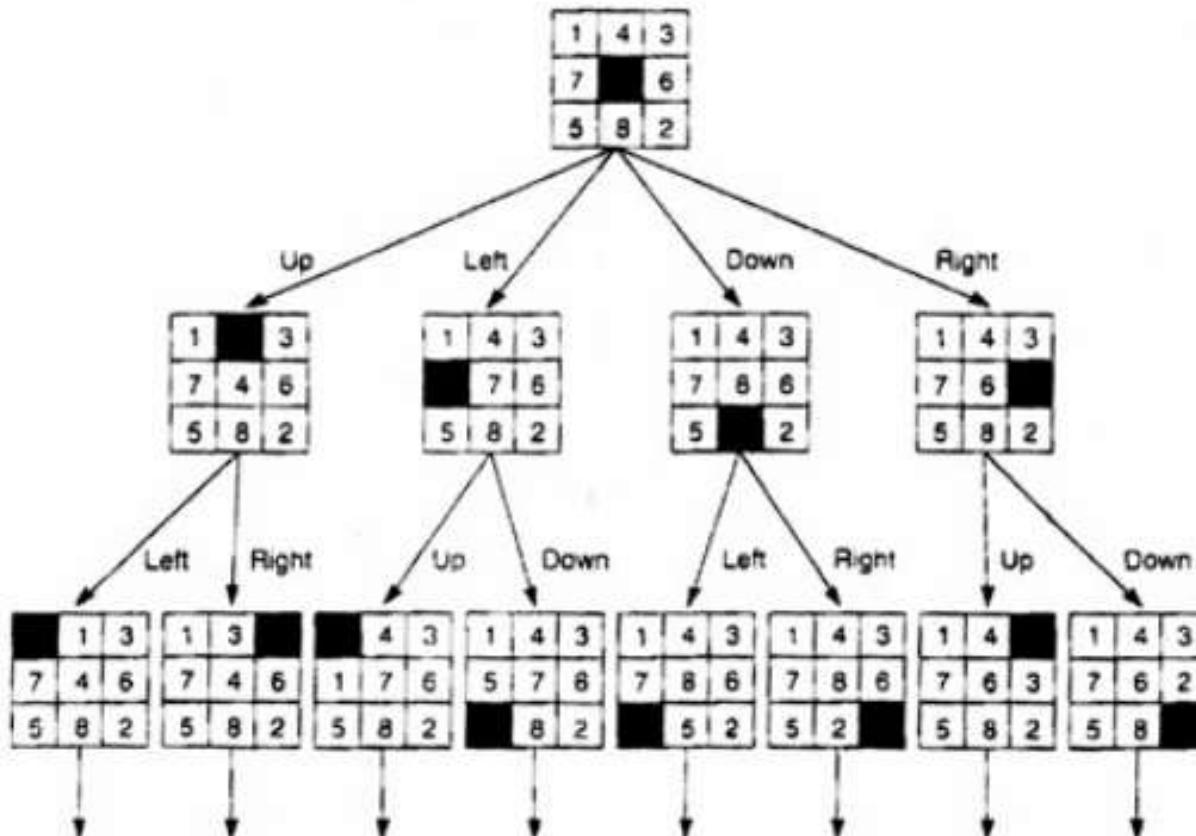


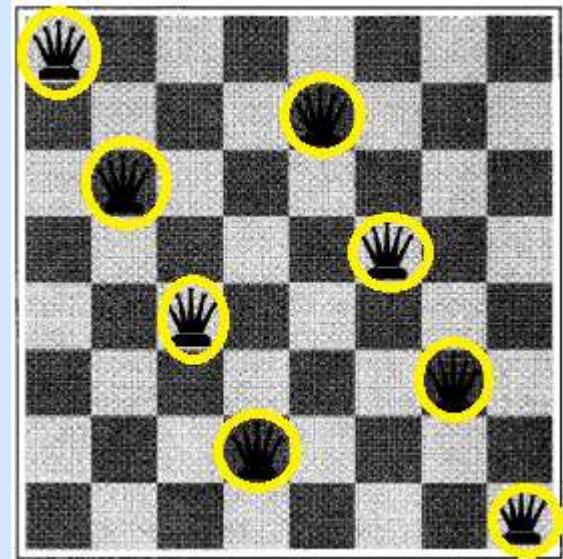
Figure 3.6 State space of the 8-puzzle generated by "move blank" operations.

# 8-Queens problem

Placing **8 queens** on a chess board, so that **none attacks the other**

## Problem Formulation

- **State** - Any arrangement of **0 to 8 queens** on board
- **Initial State** – **No queen** on the board
- **Actions** - **Add a queen** to any empty square
- **Transition Model** –
  - Returns the board with  
**a queen added to the specific square**
- **Goal Test**– 8 queens on board, none attacked



# Missionaries & Cannibals

- **3 missionaries & 3 cannibals** are on one side of a river
- The **boat** can hold **1 or 2 people**
- Find a way to get everyone to the other side, without ever leaving a **group of missionaries outnumbered by cannibals**

- **State:** (#m, #c, 1/o)

- **#m** - denotes the no. of missionaries in the first bank
  - **#c** - denotes the no. of cannibals in the first bank
  - **The last bit** - indicates whether the boat is in the first bank

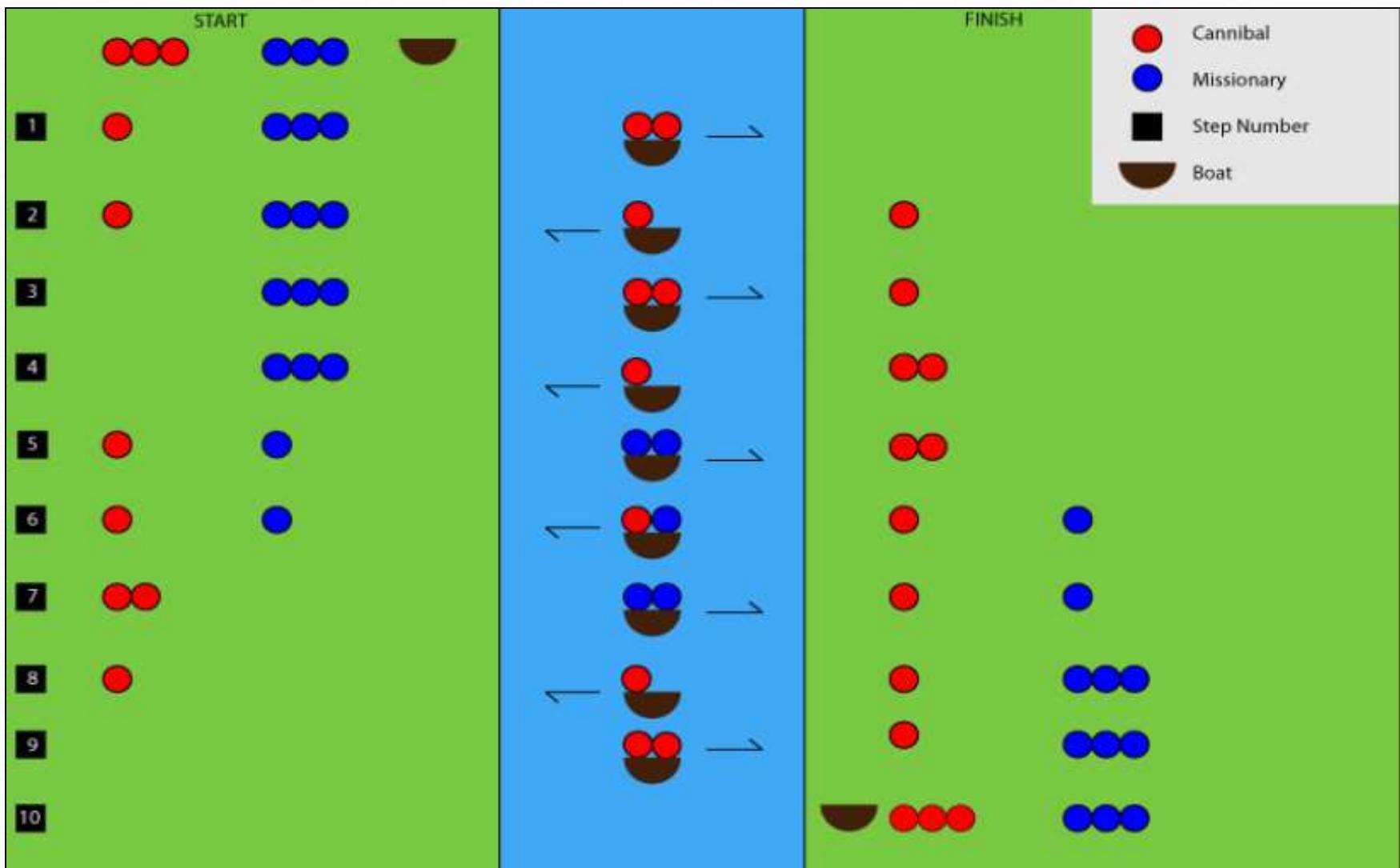
- **Initial state:** (3, 3, 1)

- **Goal state:** (0, 0, 0)

$m - o \text{ or } o \geq c$

- **Actions:** Boat carries (1, 0) or (0, 1) or (1, 1) or (2, 0) or (0, 2)

**Initial state:** (3, 3, 1)

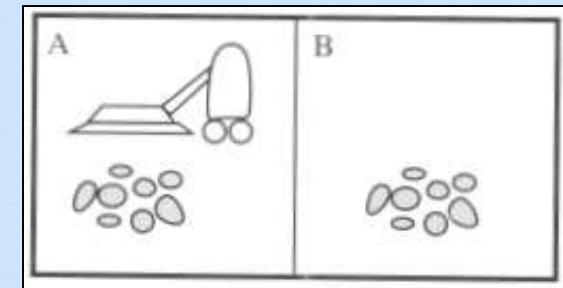


**Goal state:** (0, 0, 0)

# Example Problems

## Vacuum World - Problem formulation

- **States**
- If Agent has  $n$  ( $=2$ ) locations, each location has 2 possibilities (Dirt or No Dirt), Possible world states =  $n \cdot 2^n$  ( $= 2 \cdot 2^2 = 8$ )
- **Initial State**
- Any state can be the starting state
- **Actions** – Each state has 3 actions (Left, Right, Suck)
- **Transition Model**
- All actions have expected effects except below which have no effect : (1) Moving Left in leftmost square (2) Moving Right in rightmost square (3) Sucking in a clean square
- **Goal Test** – Checks if all squares are clean
- **Path Cost** – Each step costs 1 (Path cost is the ‘no. of steps in the path’)



# Real-World Problems

## Route Finding Problem - Problem formulation

- Used in applications like Driving Directions, Touring, Travelling Salesman (TSP), ROBOT Navigation etc.. **Described by  $\langle S, s, O, G \rangle$**

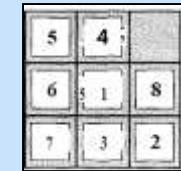
- **Set of States (S)**
- Each state is a combination of **< location & current time >**
- **Initial State (s)**
- **Starting location** as specified by user
- **Actions (O)** – **Taking flight** from current location (*seat, class ...*)
- **Transition Model**
- State resulting from taking a flight -> **Destination** (current location), **Arrival time** (current time)
- **Goal Test** – **Final destinations (G – Goal Set)** as specified by the user
- **Path Cost** – **Monetary cost**, Travel time, Airplane details etc..

# Problem space & search

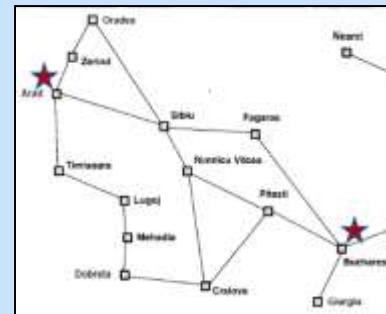
- To build a system for problem solving we need to do the following:
  1. Define the problem precisely
  2. Analyze the problem
  3. Represent the knowledge necessary for the problem
  4. Choose the best problem solving technique & apply it to solve the problem

# Definitions – State & State Space

- **State** - is an instance of the problem
  - Ex: *current city, state of 8-puzzle board*



- **State Space:**
  - Is a graph :
    - Whose nodes are set of states
    - Whose links are actions that transform one state to another state



# State space search

- **Basic Search Problem:**
  - Given:  $[S, s, O, G]$  where
    - **S** - is the (implicitly specified) set of states
    - **s** - is the Initial state
    - **O** - is the set of state transition operators
    - **G** - is the set of goal states
- To find **a sequence of state transitions** leading from Initial state **s** to a goal state **G**
- **Variants (to be considered later)**
  - Finding the shortest sequence
  - When operators have an associated cost

# Outline of a search algorithm

1. **Initialize:** Set  $\text{OPEN} = \{s\}$  //Set of unexplored state(s)
  2. **Fail:** If  $\text{OPEN} = \{ \}$ , then terminate with failure // no more states
  3. **Select:** Select a state,  $n$ , from  $\text{OPEN}$
  4. **Terminate:** If  $n \in G$ , terminate with success //If reached a Goal state
  5. **Expand:** Generate the successors of  $n$  using  $O$  and insert them in  $\text{OPEN}$
  6. **Loop:** Go To Step 2
- **Compare:**  $\text{OPEN}$  is a queue (FIFO) versus a stack (LIFO)

# Uninformed (Blind) Search

Searching with no additional info available about states other than the problem definition

- Approach – Generate successors from current state & distinguish a goal state from non-goal state

## BREADTH-FIRST Search

- Root node of the search tree is expanded first
- All the **successors** of Root node are **expanded next** & then their **successors**...so on
- All nodes at a given level (**depth**) are expanded before next level
- BFS is an instance of general Graph-Search Algorithm
- Lowest unexpanded node is chosen for expansion by using **FIFO queue**  
Shallowest node from unexplored node set (frontier) is explored next

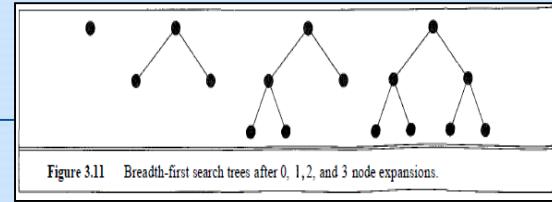


Figure 3.11 Breadth-first search trees after 0, 1, 2, and 3 node expansions.

# Traveler's path finding problem

## (Travelling Romania)

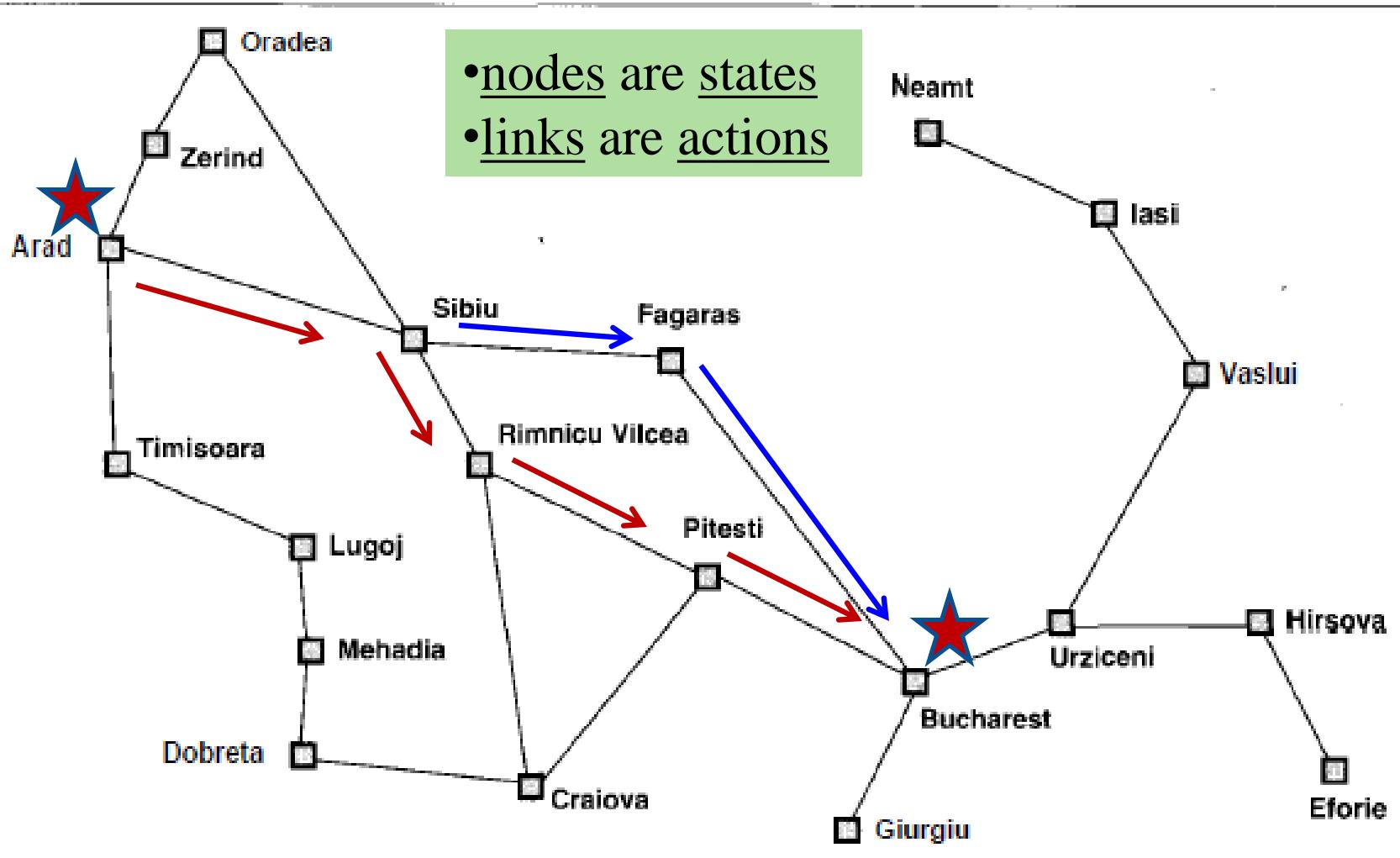


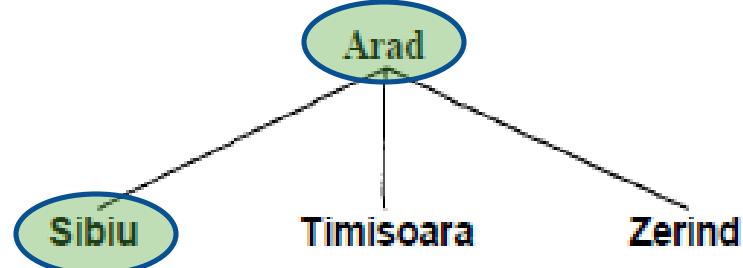
Figure 3.3 A simplified road map of Romania.

# Search Tree – Arad to Bucharest

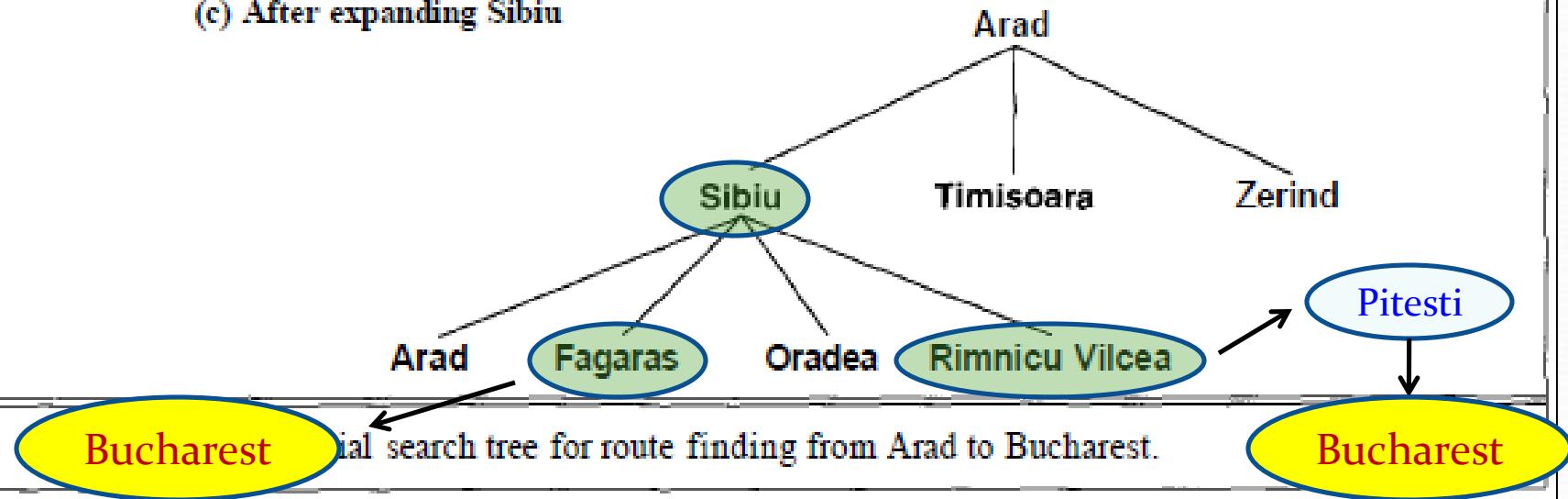
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



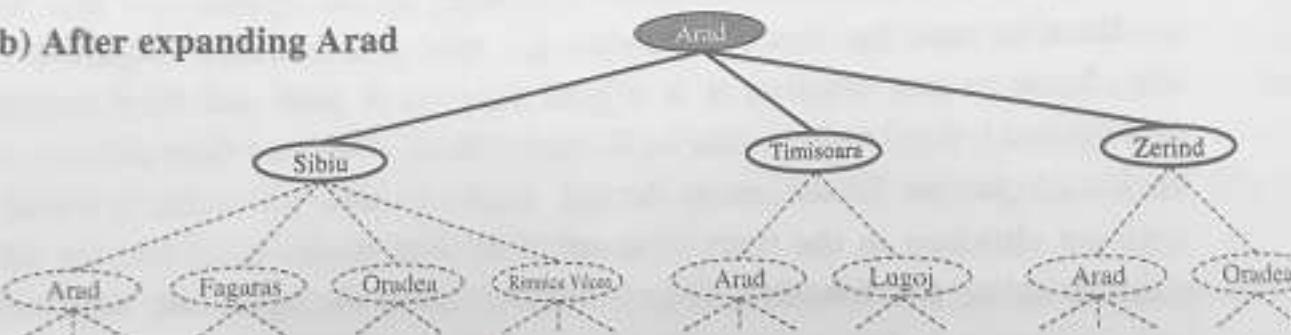
Final search tree for route finding from Arad to Bucharest.

# Search Tree – Arad to Bucharest

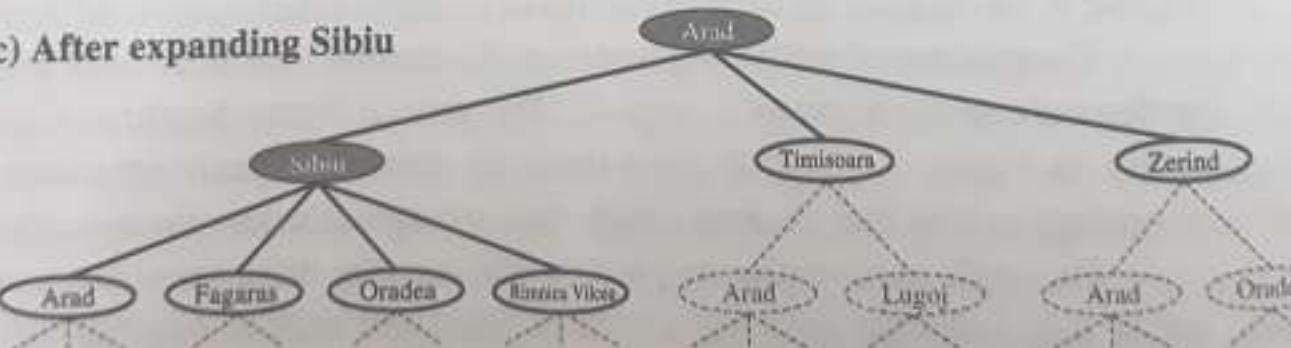
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



# BREADTH-FIRST Search

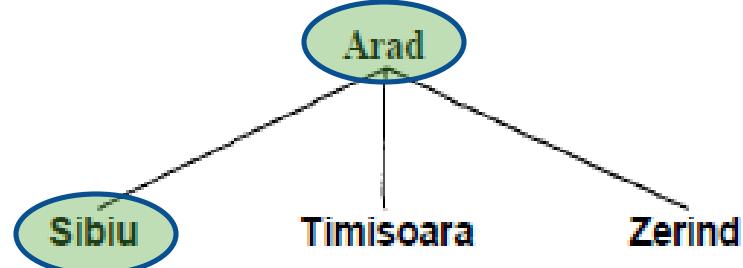
- New nodes go to the back of the queue & Older nodes are Expended first  
**FIFO** queue 
- Goal test is applied to each node when generated
- Complete – Shallowest goal node is at a finite depth ‘d’, provided branching factor ‘b’ is finite
- Shallowest (least depth) goal node may not be optimal  
Check diagram in next slide
- BFS search is optimal if path cost is increasing with depth of node
- Searching an “**Uniform tree**” where each node has same ‘b’ successors
  - Total no. of nodes generated :  $b \text{ (1st level)} + b^2 \text{ (2nd level)} + b^3 + \dots + b^d = O(b^d)$
  - Time Complexity:  **$O(b^d)$**
  - Every generated node remains in memory –  $O(b^{d-1})$  in explored set &  $O(b^d)$  in frontier (unexplored)
  - Space Complexity:  **$O(b^d)$**

# Search Tree – Arad to Bucharest

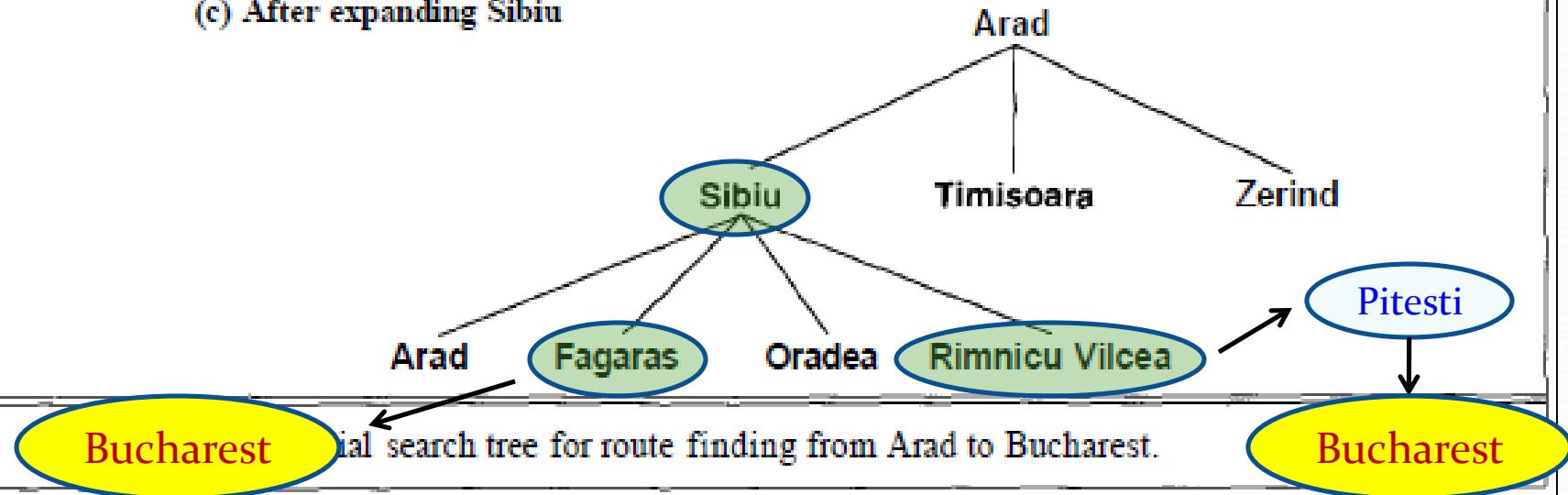
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu



# BFS on a Graph

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier  $\leftarrow$  a FIFO queue with node as the only element // List of nodes to explored next
    explored  $\leftarrow$  an empty set // List of nodes already explored
    loop do
        if EMPTY?(frontier) then return failure // If no nodes to explore
        node  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */
        add node.STATE to explored // node is added to explored list
        for each action in problem.ACTIONS(node.STATE) do
            child  $\leftarrow$  CHILD-NODE(problem, node, action) // Generate successors
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION // Reached goal state
                frontier  $\leftarrow$  INSERT(child, frontier) // Add node to frontier
```

Start node

Initialization

Iter-1

Iter-2

Reached  
goal state

# BFS on a Binary tree

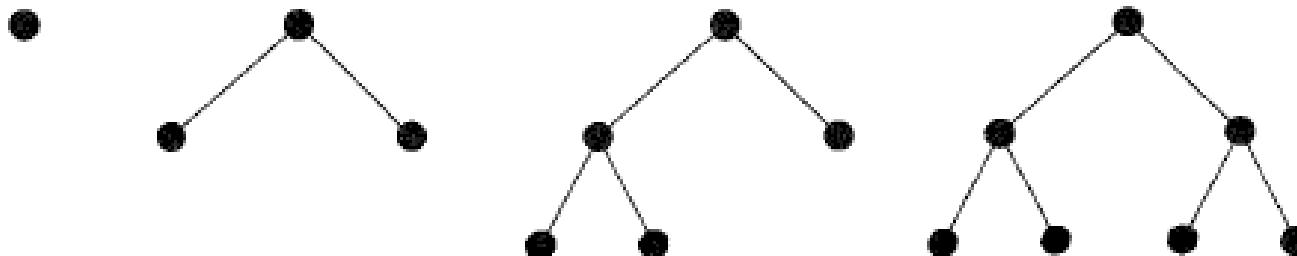


Figure 3.11 Breadth-first search trees after 0, 1, 2, and 3 node expansions.

## Time & Memory Requirements for BFS

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	$10^6$	18 minutes	111 megabytes
8	$10^8$	31 hours	11 gigabytes
10	$10^{10}$	128 days	1 terabyte
12	$10^{12}$	35 years	111 terabytes
14	$10^{14}$	3500 years	11,111 terabytes

Figure 3.12 Time and memory requirements for breadth-first search. The figures shown assume branching factor  $b = 10$ ; 1000 nodes/second; 100 bytes/node.

# BREADTH-FIRST Search

- Memory requirements are a **bigger problem** than Execution time
  - For problem with depth 12, Petabytes of memory is required
- Execution time is still a major factor
  - 350 years for solution for problem with depth 16
- Exponential Complexity search problems cannot be solved by uninformed methods



# DEPTH-FIRST Search

- DFS expands the **deepest node** in the **frontier** of the search tree 1st
- The expanded nodes are then dropped from the frontier
- Search “Backs up” to the next deepest node with unexplored successors
- DFS uses **LIFO queue** (most recently generated node is expanded)
- **Implementations:** (1) Graph-Search (2) Recursive function call ↑
  - DFS is an non-optimal search technique
  - Let ‘b’ be the branching factor & ‘m’ be max depth of any node
    - Time Complexity:  $O(b^m)$  ( m can be  $\gg$  d of BFS)
  - DFS have no clear adv. over BFS except for space complexity (for Tree search)
  - Once a node is expanded it can be removed from memory
    - Space Complexity:  $O(bm)$
  - For d=16, DFS needs 156 kb, but BFS - 10 exabytes (7 Trillion times more)



# DEPTH-FIRST SEARCH

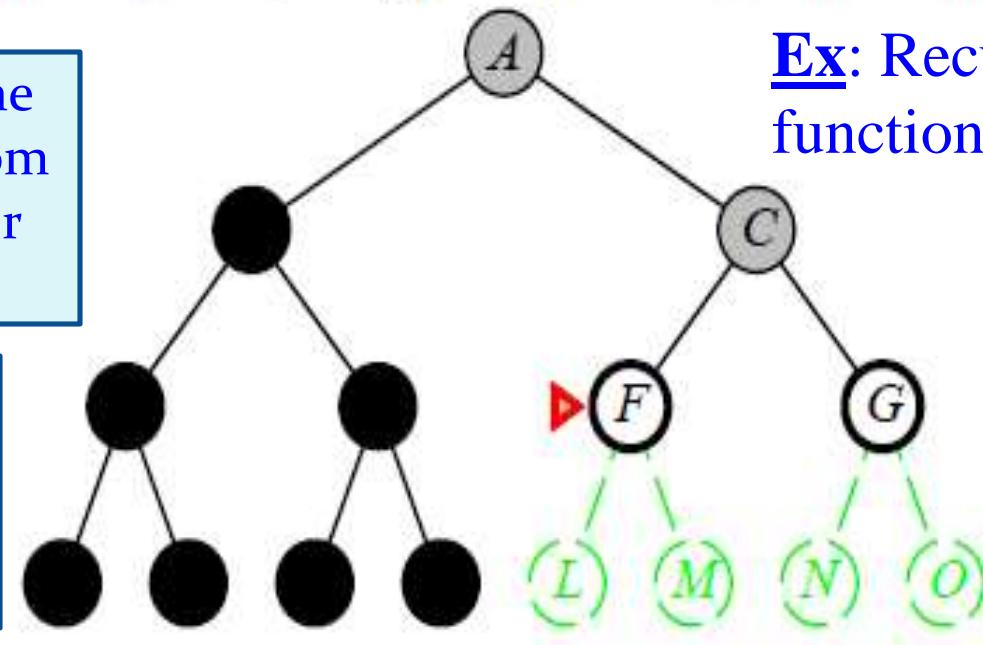
Expand deepest unexpanded node

Implementation:

*fringe* = LIFO queue, i.e., put successors at front

Once Expanded, the node is removed from memory & Frontier (Open set)

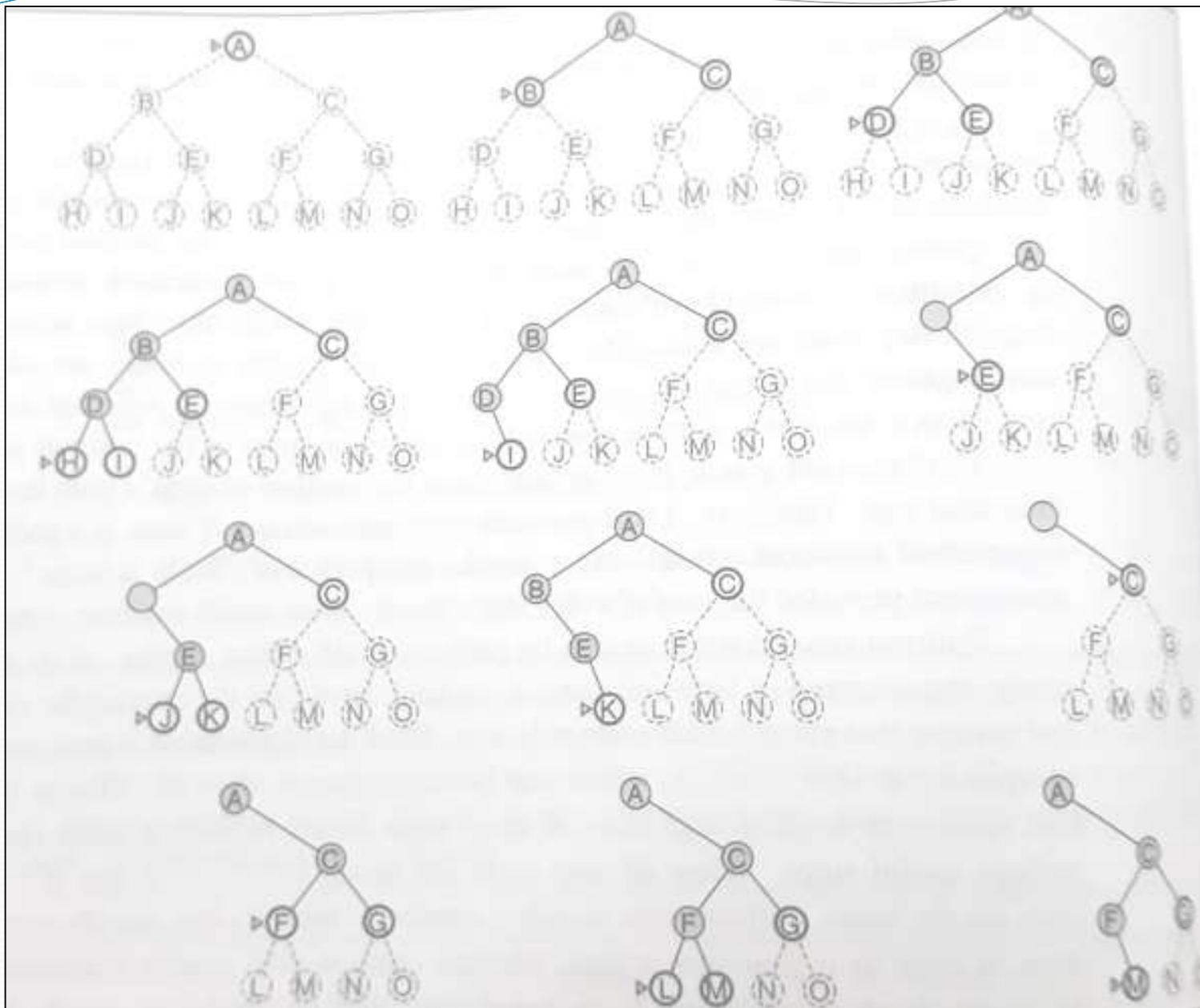
Implemented using LIFO queue



Ex: Recursive function call



# DEPTH-FIRST Search



# DFS on a Graph

```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier  $\leftarrow$  a LIFO queue with node as the only element // List of nodes to explored next
  explored  $\leftarrow$  an empty set // List of nodes already explored
  loop do
    if EMPTY?(frontier) then return failure // If no nodes to explore
    node  $\leftarrow$  POP(frontier) /* chooses the shallowest node in frontier */
    add node.STATE to explored // node is added to explored list
    for each action in problem.ACTIONS(node.STATE) do
      child  $\leftarrow$  CHILD-NODE(problem, node, action) // Generate successors
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier  $\leftarrow$  INSERT(child, frontier) // Add node to frontier
```

Start node

Initialization

Iter-1

Iter-2

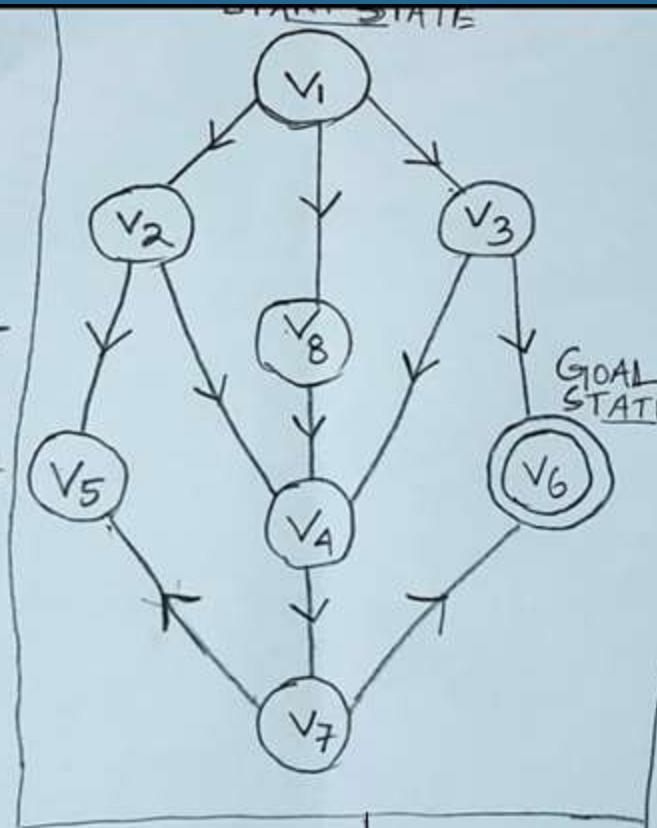
// Reached goal state

# Example - BFS-DFS Search

DFS

Frontier (LIFO) · Stack

- (1)  $v_1$  Exploded set
- (2)  $v_1 \rightarrow v_1$   
 $v_2, v_3, v_8$   $\downarrow$   
 $v_8 \rightarrow v_8$
- (3)  $v_2, v_3, v_4$   $\downarrow$   
 $v_4 \rightarrow v_4$
- (4)  $v_2, v_3, v_7$   $\downarrow$   
 $v_7 \rightarrow v_7$
- (5)  $v_2, v_3, v_5, v_6$   $\downarrow$   
 $v_6 \rightarrow v_6$



BFS

Frontier (FIFO) · Queue

[Insert from REAR] [Remove from FRONT] Explored set

- (1)  $\xrightarrow{v_1}$
- (2)  $\xrightarrow{v_1} \xrightarrow{v_2}$
- (3)  $\xrightarrow{v_2} \xrightarrow{v_8}$
- (4)  $\xrightarrow{v_8} \xrightarrow{v_4}$
- (5)  $\xrightarrow{v_4} \xrightarrow{v_3}$
- (6)  $\xrightarrow{v_3} \xrightarrow{v_6}$
- (7)  $\xrightarrow{v_6} \xrightarrow{v_5}$
- (8)  $\xrightarrow{v_5} \xrightarrow{v_6}$

# DEPTH-LIMITED Search

- **Backtracking Search**

- Only one successor is generated at a time
- Uses less memory –  $O(m)$       ←  $m$  – max depth

← To save memory

## Depth-Limited Search

- Searches with a predetermined depth limit of 'l' // Ex:  $l = 19$  for Romania
- Assumed that **no successors** exist after **level-l**
- This solves the **indefinite path problem** faced by DFS & BFS
- **DFS** is a special case of **DLS** with  $l = \text{infinitive}$
- But introduces 'incompleteness' if  $l < d$  (shallowest goal is beyond the depth limit)
- **DLS** is non-optimal if  $l > d$ 
  - Time Complexity:  $O(b^l)$
  - Space Complexity:  $O(bl)$
- Ex: Romania has 20 cities with solution length  $\leq 19$ . So,  $l = 19$  can be chosen

# Iterative Deepening Depth-First Search

- Finds the “Best depth limit”
  - By gradually increasing the **depth limit 0, 1, 2.. until goal is found**
  - **Goal** – Shallowest goal node (d)

## Algorithm

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
    inputs: problem, a problem

    for depth  $\leftarrow$  0 to  $\infty$  do
        if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
    end
    return failure
```

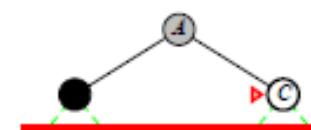
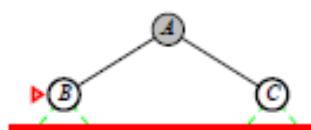
- Combines the benefits of **DFS & BFS**
- Like **DFS** - Space Complexity:  $O(bd)$
- Like **BFS** - Time Complexity:  $O(b^d)$
- Preferred uninformed search method when the search space is large & depth of the solution is not known

# ITERATIVE DEEPENING SEARCH (ID-DFS): EXAMPLE

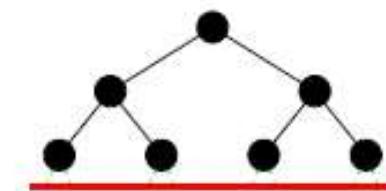
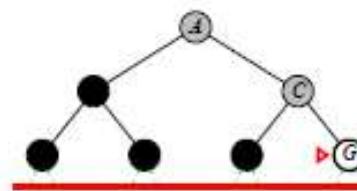
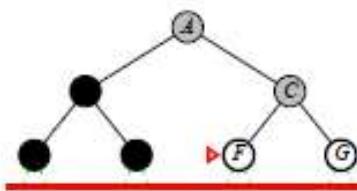
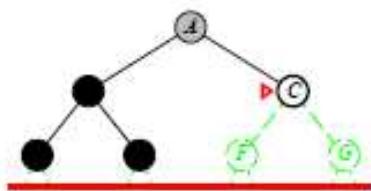
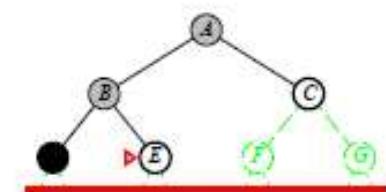
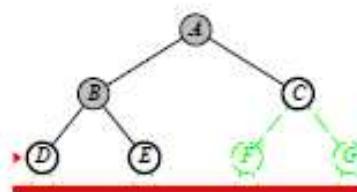
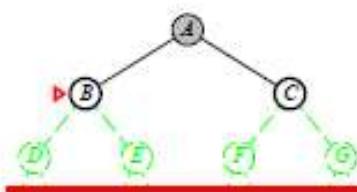
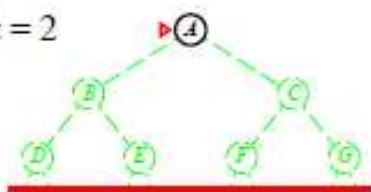
Limit = 0



Limit = 1



Limit = 2



# ITERATIVE DEEPENING SEARCH (ID-DFS):

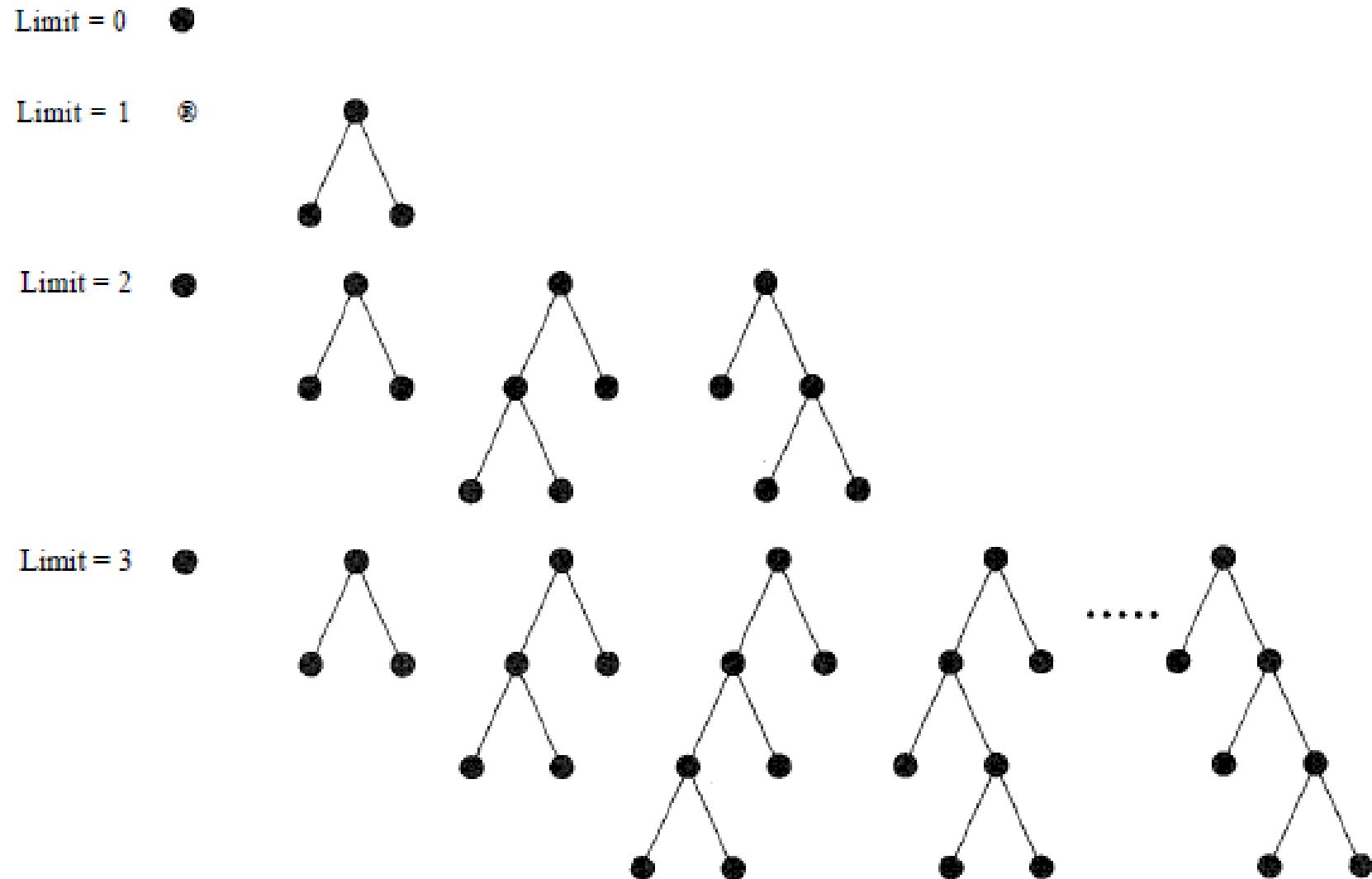


Figure 3.16 Four iterations of iterative deepening search on a binary tree.

# ITERATIVE DEEPENING SEARCH (ID-DFS): PROPERTIES



Time

$$db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$$

Space

$$O(bd)$$

Numerical comparison for  $b = 10$  and  $d = 5$ , solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,100$$

IDS does better because other nodes at depth  $d$  are not expanded

BFS can be modified to apply goal test when a node is generated

# Bidirectional Search

## ● Idea behind BS

- \* Search “from both ends”
- \* Search “Forward from initial state” & “Backwards from goal” – till intersect

## ● Analysis

- \* Solution depth (in levels from root, i.e., edge depth):  $d$ 
  - ⇒  $b^i$  nodes generated at level  $i$
  - ⇒ At least this many nodes to test
  - ⇒ Total:  $\Sigma b^i = 1 + b + b^2 + \dots + b^{d/2} = O(b^{d/2})$

- Space Complexity:  $O(b^{d/2})$       & Time Complexity:  $O(b^{d/2})$

# Bidirectional Search

## Properties

- \* Space complexity is a weakness of BS
- \* But, Time complexity makes BS attractive

## \* *Backward search*

- \* **8-Puzzle & Romania** problems have only one goal state  
*(backward search is similar to forward search)*
- \* In case of many goal states a new dummy goal state is  
created (predecessor to all goal states)

# UNIFORM-COST Search

- When all step(path) costs are equal => BFS search is the optimal search
- When steps costs are unequal => Bcoz, shallowest goal node is found first
  - Uniform-cost search expands the **node 'n'** with the lowest path cost g(n) instead of the shallowest node

function UNIFORM-COST-SEARCH(*problem*) returns a solution, or failure

*node*  $\leftarrow$  a node with STATE = *problem.INITIAL-STATE*, PATH-COST = 0  
*frontier*  $\leftarrow$  a priority queue ordered by PATH-COST, with *node* as the only element

*explored*  $\leftarrow$  an empty set

loop do

if EMPTY?(*frontier*) then return failure

*node*  $\leftarrow$  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*  $\leftarrow$  CHILD-NODE(*problem, node, action*)

if *child.STATE* is not in *explored* or *frontier* then

*frontier*  $\leftarrow$  INSERT(*child, frontier*)

else if *child.STATE* is in *frontier* with higher PATH-COST then

replace that *frontier* node with *child*

Initialization

Iteration-1

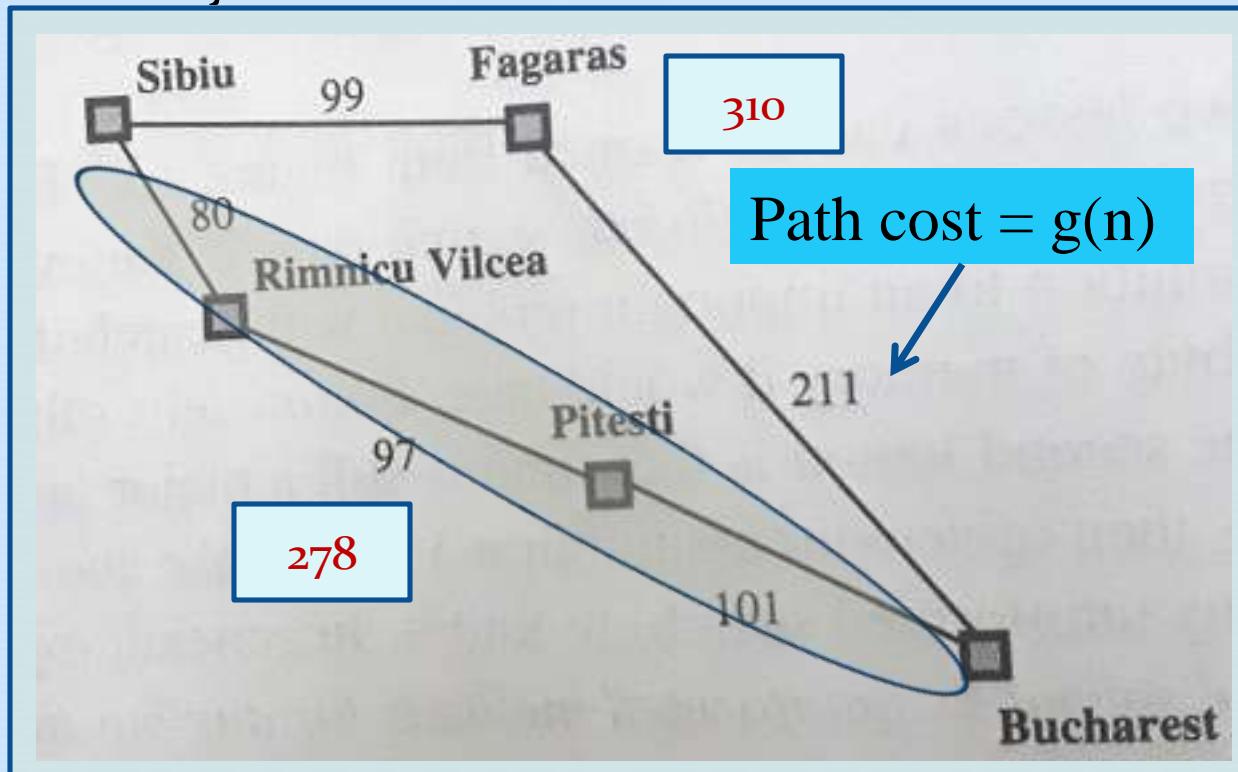
Iter-2

If frontier node has  
higher cost, replace it

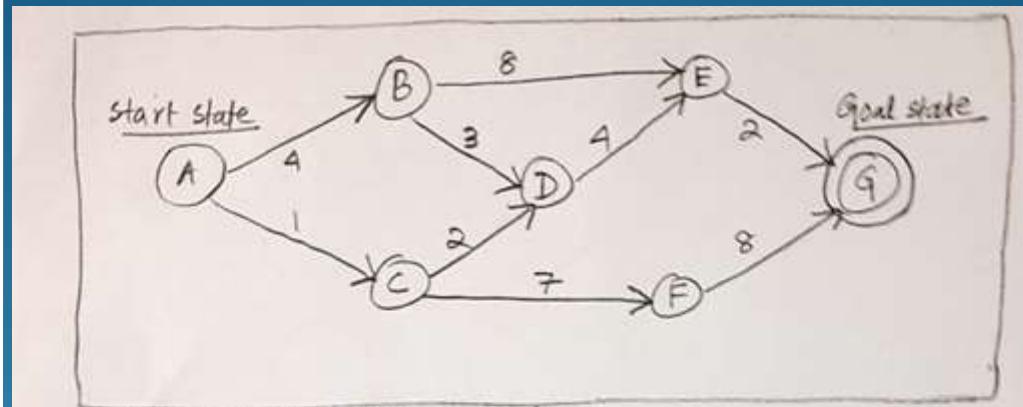
# UNIFORM-COST Search

- Difference with BFS

- Frontier (Open set) – Is a priority queue ordered by ‘ $g$ ’ (**Path cost**)
- Goal Test is applied on node when it is selected for expansion (rather than on generation)
- A Cost Test (path cost calc) is added to check if a **better path is found** to a node currently in frontier



# Example Uniform Cost Search



(\* Least Path Cost - LPC)

Frontier (Priority Queue ordered by Path Cost)

Explored set

- |   |  |                                   |                  |
|---|--|-----------------------------------|------------------|
| ① | A <sup>0</sup>   | → A <sup>0</sup>                  | → A <sup>0</sup> |
| ② | B <sup>4</sup> , C <sup>1</sup>  | → C <sup>1</sup><br>(LPC)         | → C <sup>1</sup> |
| ③ | B <sup>4</sup> , D <sup>3(1+2)</sup> , F <sup>8(1+7)</sup>                 | → D <sup>3</sup><br>(LPC)         | → D <sup>3</sup> |
| ④ | B <sup>4</sup> , F <sup>8</sup> , E <sup>7(1+2+4)</sup><br>(A → C → D → E) | → B <sup>4</sup><br>(LPC)         | → B <sup>4</sup> |
| ⑤ | F <sup>8</sup> , E <sup>7</sup>  | → E <sup>7</sup> → E <sup>7</sup> |                  |
| ⑥ | F <sup>8</sup> , G <sup>9(1+2+4+2)</sup>                                   | → F <sup>8</sup> → F <sup>8</sup> |                  |
| ⑦ | G <sup>9</sup>   | → G <sup>9</sup>                  |                  |

Final Path: A, C, D, E, G

# Comparison of Uninformed search techs

- See table below:
- **Complete** => Solution is guaranteed

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	$b^d$	$b^d$	$b^m$	$b^l$	$b^d$	$b^{d/2}$
Space	$b^d$	$b^d$	$bm$	$bl$	$bd$	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l > d$	Yes	Yes

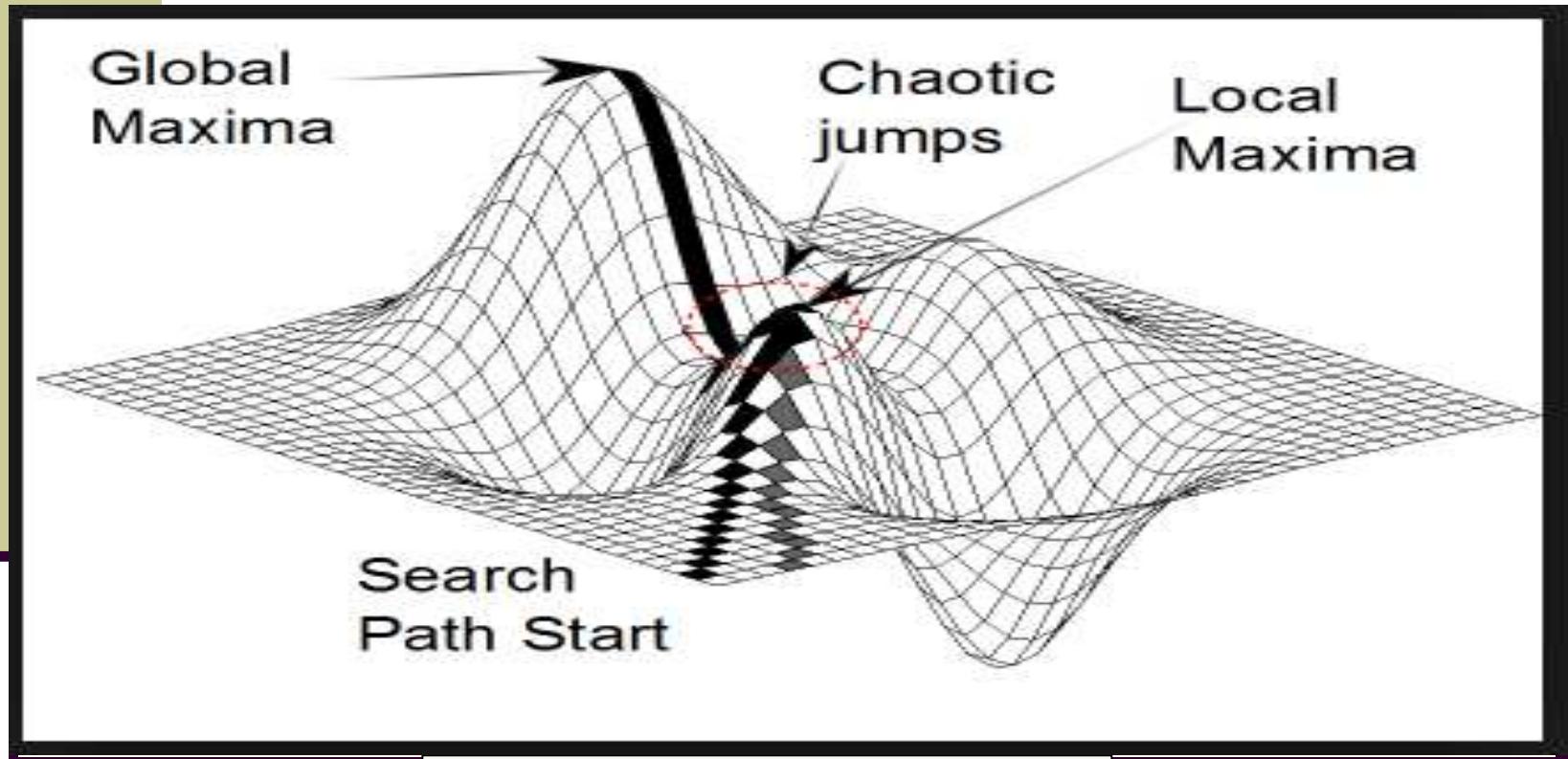
- b – branching factor
- d – depth of the shallowest solution
- m – max depth of search tree
- l – depth limit

# Tradeoff between space and time

- **Iterative deepening**
  - Perform DFS repeatedly using increasing depth bounds
  - Works in  $O(b^d)$  time and  $O(bd)$  space
- **Bi-directional search**
  - Possible only if the operators are reversible
  - Works in  $O(b^{d/2})$  time and  $O(b^{d/2})$  space



# Beyond Classical Search – Hill Climbing & Simulated Annealing



*Dr. Pulak Sahoo*

Associate Professor

Silicon Institute of Technology



# Topics

- **Beyond Classical Search**

- **Local Search Algorithms**

- (1) Hill-climbing search (Local Greedy Search)**

- HCS: n-queens problem
    - HCS: Algorithm
    - HCS: TSP, 8-Puzzle Problems

- (2) Simulated Annealing**

- HCS: n-queens problem
    - HCS: Algorithm



# Local search algorithms

- Previous search algorithms use to explore search space systematically to find the **optimal path to the goal**
  - Ex: *TSP, Route finding problems etc.*
- But, in many large & complex search problems, the **path** to the goal is **irrelevant**; the **goal state** itself is the **solution**
  - Ex: *8-puzzle, n-queens, Chess etc.*

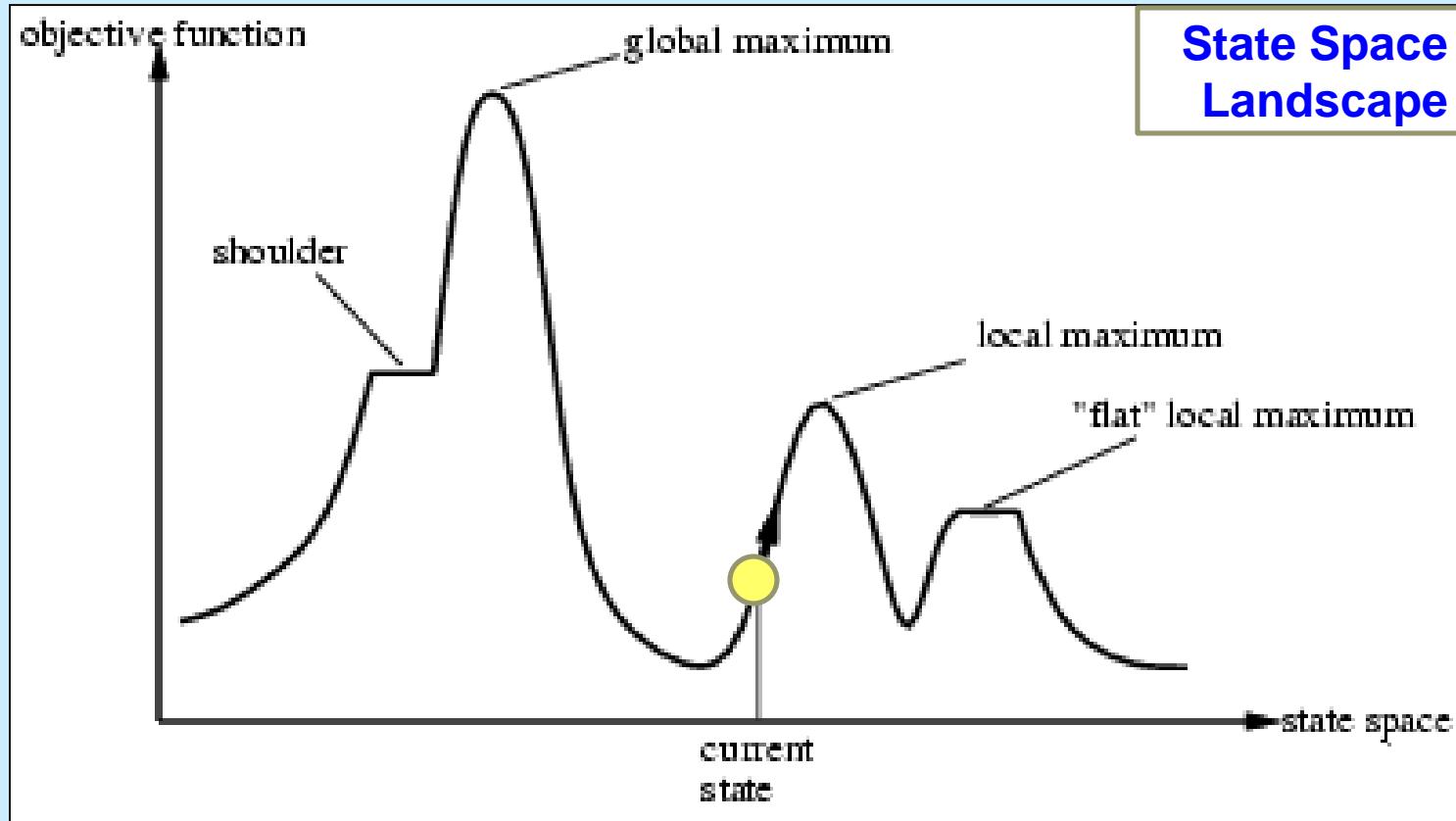


# Local search algorithms

- In such cases: we can use **local search algorithms**
  - LCA's keep track of the "**current**" state
  - Then try to **improve it** by comparing it with its **neighbors** or **successors** (*if current state is better, it is a peak*)
  - Comparison is done based on “**Objective Function**” (*elevation – similar to evaluation function*)
    - *Ex: No. of conflicts for n-Queen, Manhattan distance for 8-Puzzle...*
  - **Advantages** –
    1. Requires **small memory** – a constant amount (*to store states & values*)
    2. Can find a **reasonable solution** in **large search space** where systematic algorithms are unsuitable

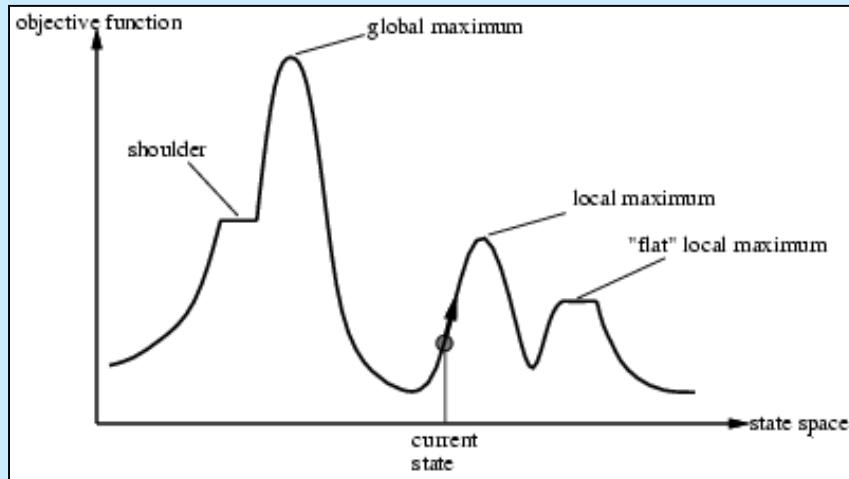
# Local search algorithms

- Are useful for solving large search space problems fast
  - By finding the **best state** according to “Objective Function”
  - Ex: Darwinian Evolution, Fitness Function – Reproductive fitness



# Local search algorithms

- LSAs explore the “State-space landscape” containing 2 things
- Which has both **location** (state) & **elevation** (cost defined by objective function)
- **Aim** – to find **lowest valley** (global minimum) or **highest peak** (global maximum)
- **Complete LSA** – always finds a **goal**
- **Optimal LSA** - always finds a **global maximum/minimum**





# Hill-climbing search (steepest ascent)

- **HCS** algorithm is a **loop** that continuously moves in the direction of increasing value (**uphill**)
- **Terminates** when reaches a “**Peak**” where **no neighbor** has a higher value
  - *Like climbing Everest in thick fog*
- **Data structure:** Stores **State** & **Value** of **objective function** (not search tree)
- Also called “**Greedy Local Search**”
- Makes rapid progress towards a **solution** by improving on previous state
  - *Ex: n-Queen problem*



objective function

global maximum

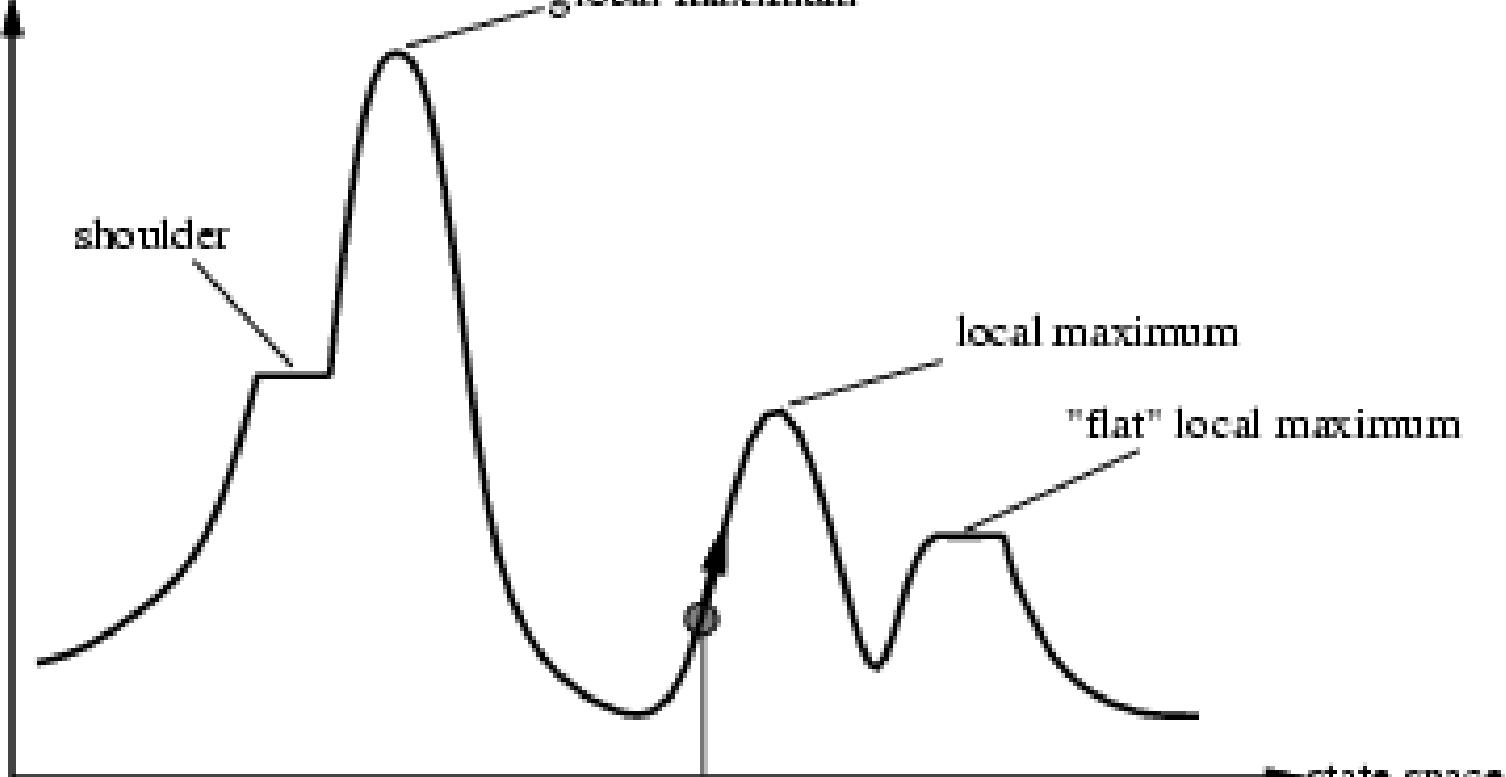
shoulder

local maximum

"flat" local maximum

current  
state

state space





# HCS: Algorithm

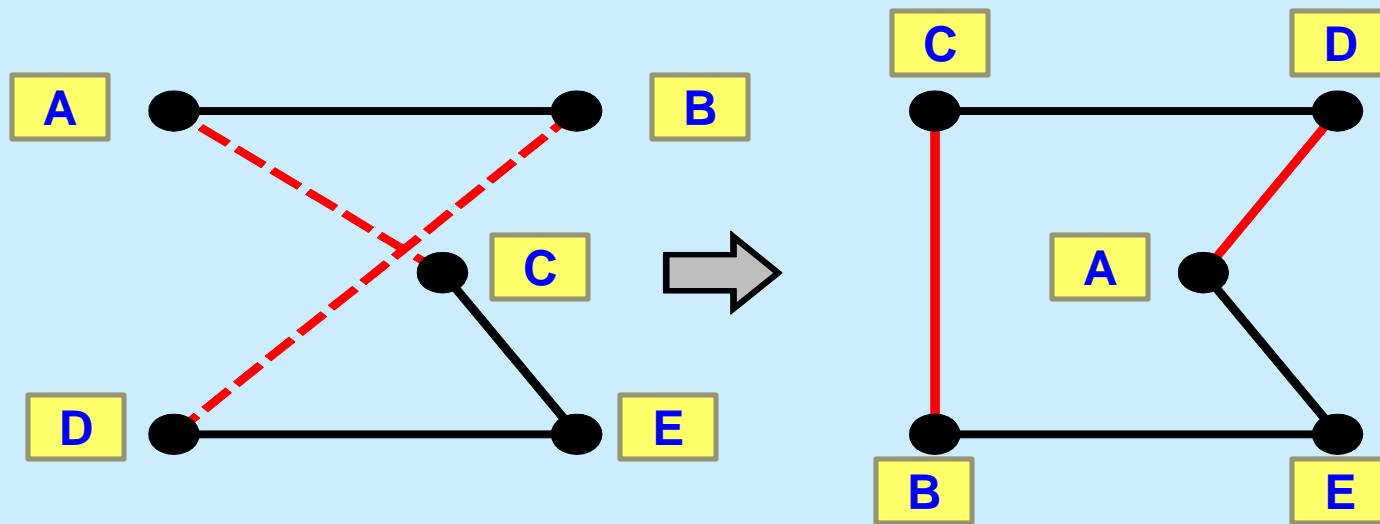
- At each step current node is replaced by it's best neighbor (with highest/lowest VALUE)

```

function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                    neighbor, a node
  current  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])    \\\ Current node set to start node
  loop do \\\ Loop through each neighbor
    neighbor  $\leftarrow$  a highest-valued successor of current    \\\ Select the nbr with best obj .function
    if VALUE[neighbor]  $\leq$  VALUE[current] then return STATE[current]
    current  $\leftarrow$  neighbor      \\\ If current node's obj function is higher, return current node
                                else set nbr node as the current node
  
```

# Example: TSP

- Start with any complete tour, perform pairwise exchange



- Variants of this approach get within 1% of optimal very quickly with thousands of cities



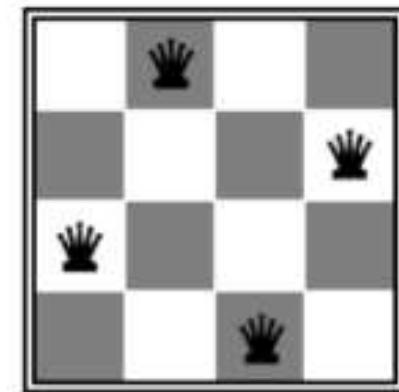
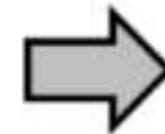
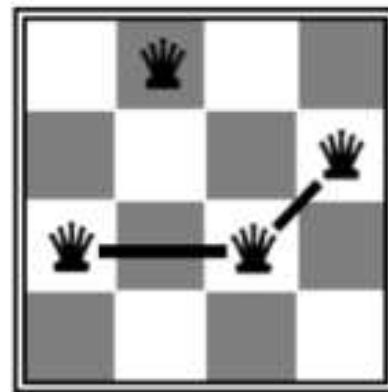
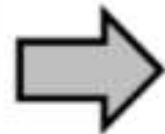
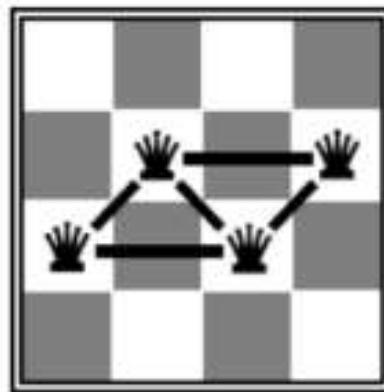
# Example: n-queens

- Put **n queens** on an **n × n board** with no two queens attacking each other (*i.e not on the same row, column or diagonal*)
- **Objective function:** **Number of conflicts** (*solution is global minimum – preferably 0 conflicts(perfect solution))* or *Manhattan distance (how many positions away)*
- **Conflict =>** the number of pairs of queens that are attacking each other



# Example: n-queens

- Reached a solution in 2 steps





# HCS: 8-queens problem

- Complete state formation
  - *Each state has 8 queens, one per column*
  - **Successor states** – *All possible states by moving a queen to another square in the same column ( $8 \times 7 = 56$  successors)*
- Heuristic cost function ( $h$ ) – No. of queen pairs that attack each other
- Global minimum = 0 (perfect solutions)
- **Fig-1** in next page shows initial state with  **$h=17$**
- Value of best successors  **$h=12$**
- It takes only **5 steps** to reach the state in **Fig-2** with  **$h=1$**  (very nearly a solution)
  - HCS makes rapid progress towards solution



# HCS: 8-queens problem

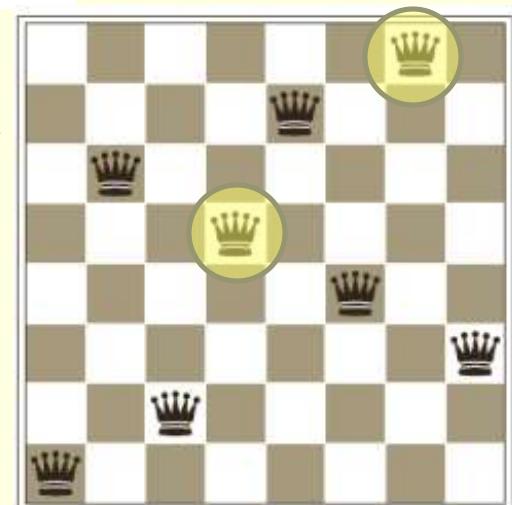
- Hill Climbing may NOT reach to a goal state for n-queens problem.

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	13	16	13	16
14	14	17	15	15	14	16	16
17	15	16	18	15	15	15	15
18	14	15	15	15	14	15	16
14	14	13	17	12	14	12	18

Min h

five steps to reach this state

local minimum with  $h = 1$



- A local minimum in the 8-queens state space; the state has  $h=1$
- but every successor has a higher cost.
- Hill Climbing will stuck here

- $h$  = no. of queen pairs attacking each other, either directly or indirectly
- $h = 17$  for the above state
- Each square contains  $h$  values of the successors ( $h=12$  is next best state)

# HCS: 8-queens problem

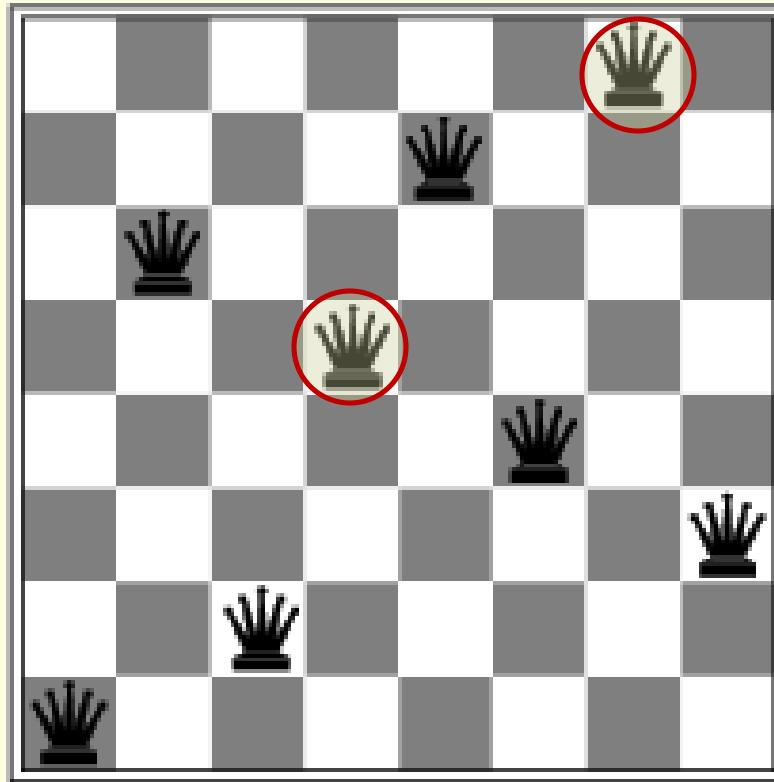
Fig-1

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	13	16	13	16
14	14	17	15	14	16	16	16
17	15	16	18	15	14	15	15
18	14	15	15	15	14	15	16
14	14	13	17	12	14	12	18

- $h$  = no. of queen pairs attacking each other, either directly or indirectly
- $h = 17$  for the above state
- Each square contains  $h$  values of the successors ( $h=12$  is next best state)

# HCS: 8-queens problem

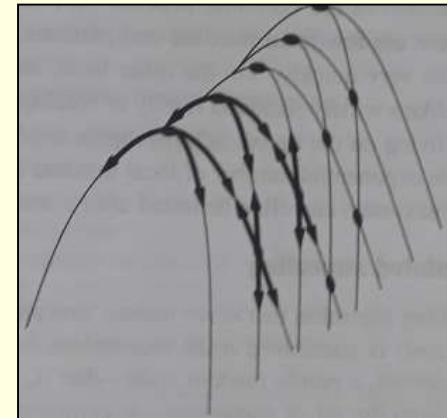
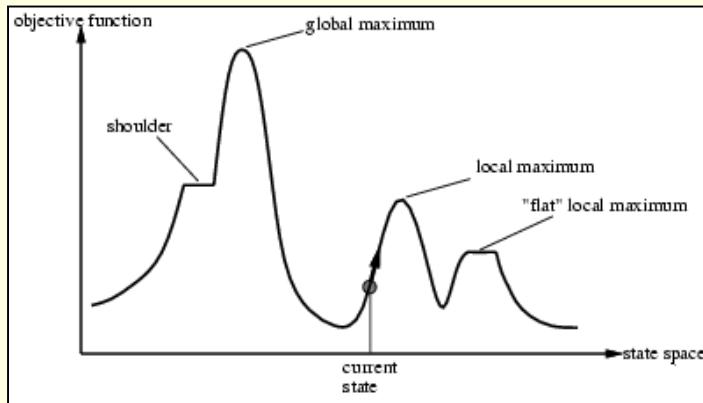
Fig-2



- A local minimum with  $h = 1$
- Achieved in 5 steps

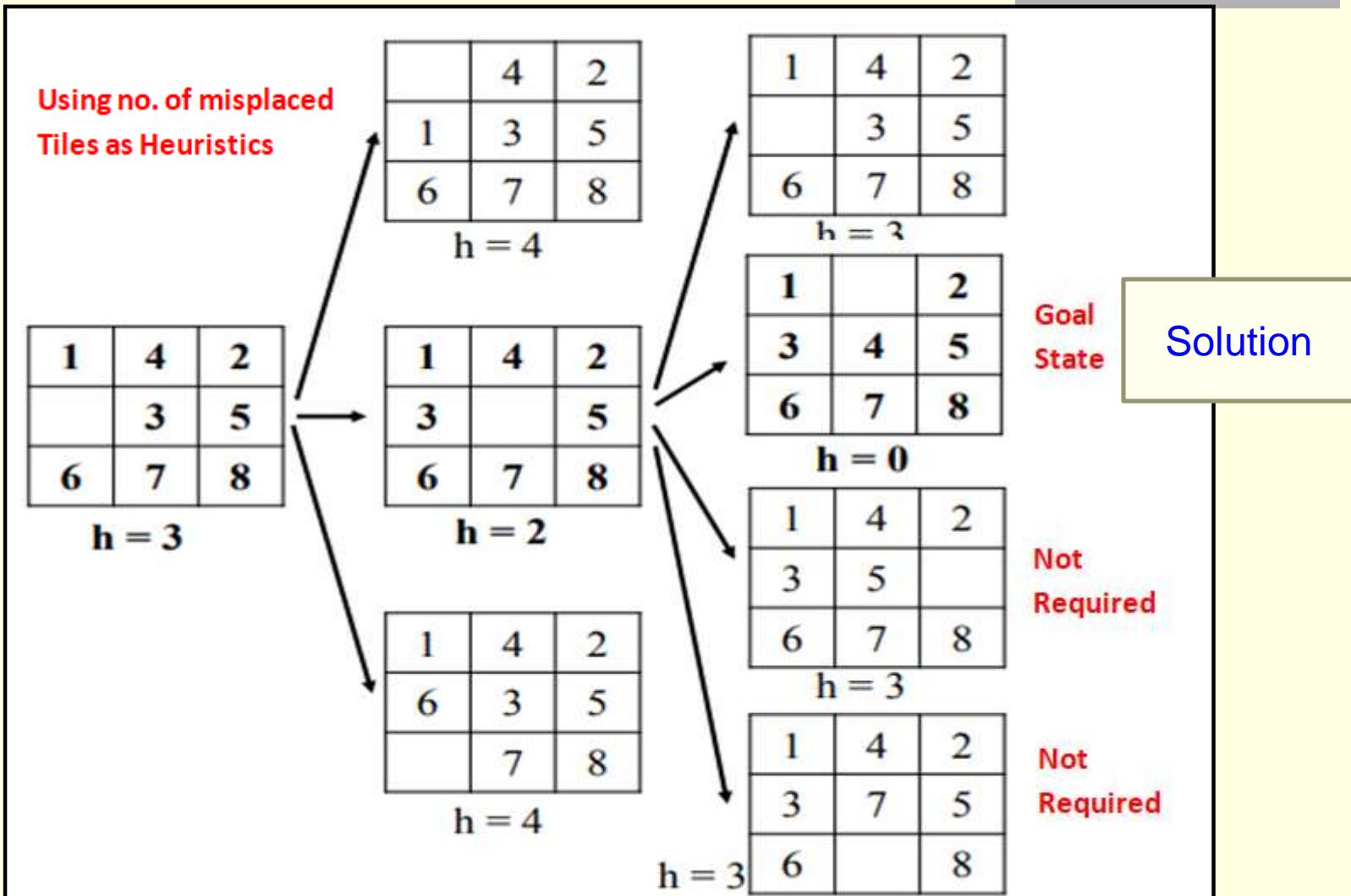
# HCS-Problems

- Local maxima: is a peak that is **higher** than it's neighboring states but **lower** than **global maximum**
  - HCS can get stuck in local maxima

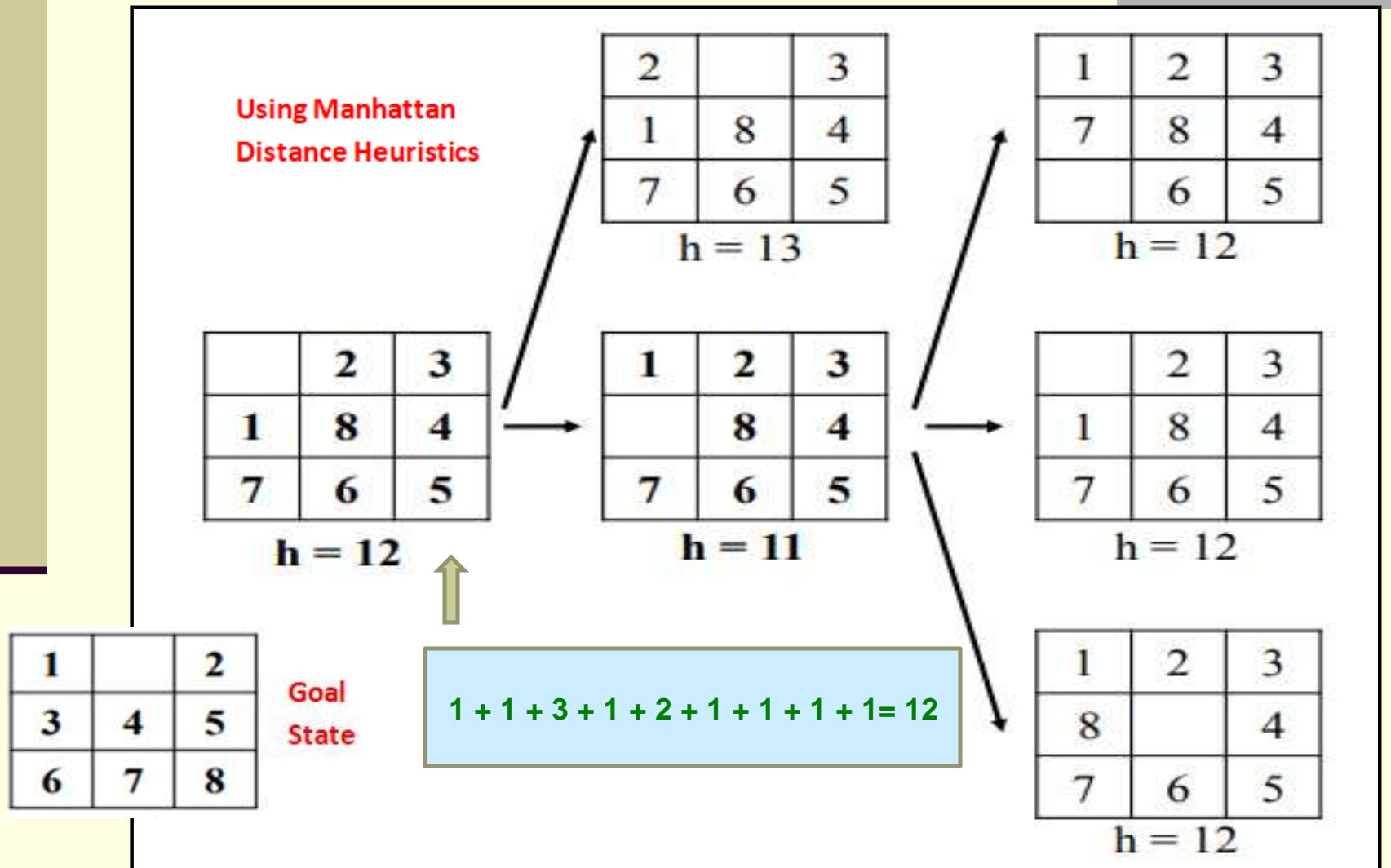


- Ridges – Sequence of local maxima that are difficult to navigate
- Plateaux – Flat local maximum from which **no uphill exit exists**

# An 8-puzzle problem solved by a HCS



# 8-puzzle: HCS stuck at local maximum





# HCS: Algorithm

- At each step current node is replaced by it's best neighbor (with highest/lowest VALUE)

```

function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem, a problem
  local variables: current, a node
                    neighbor, a node
  current  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])    \\\ Current node set to start node
  loop do \\\ Loop through each neighbor
    neighbor  $\leftarrow$  a highest-valued successor of current    \\\ Select the nbr with best obj .function
    if VALUE[neighbor]  $\leq$  VALUE[current] then return STATE[current]
    current  $\leftarrow$  neighbor      \\\ If current node's obj function is higher, return current node
                                else set nbr node as the current node
  
```



# Simulated Annealing

- **HCS algorithm** - can “get stuck in local maxima” as it never makes “downhill” moves (*hence remains incomplete*)
- **Solution**
- **Random walk** – “Randomly selects a successor” from the neighborhood
- Is complete but inefficient
- **Simulated Annealing** algorithm combines both **HCS** & **Random Walk**



objective function

global maximum

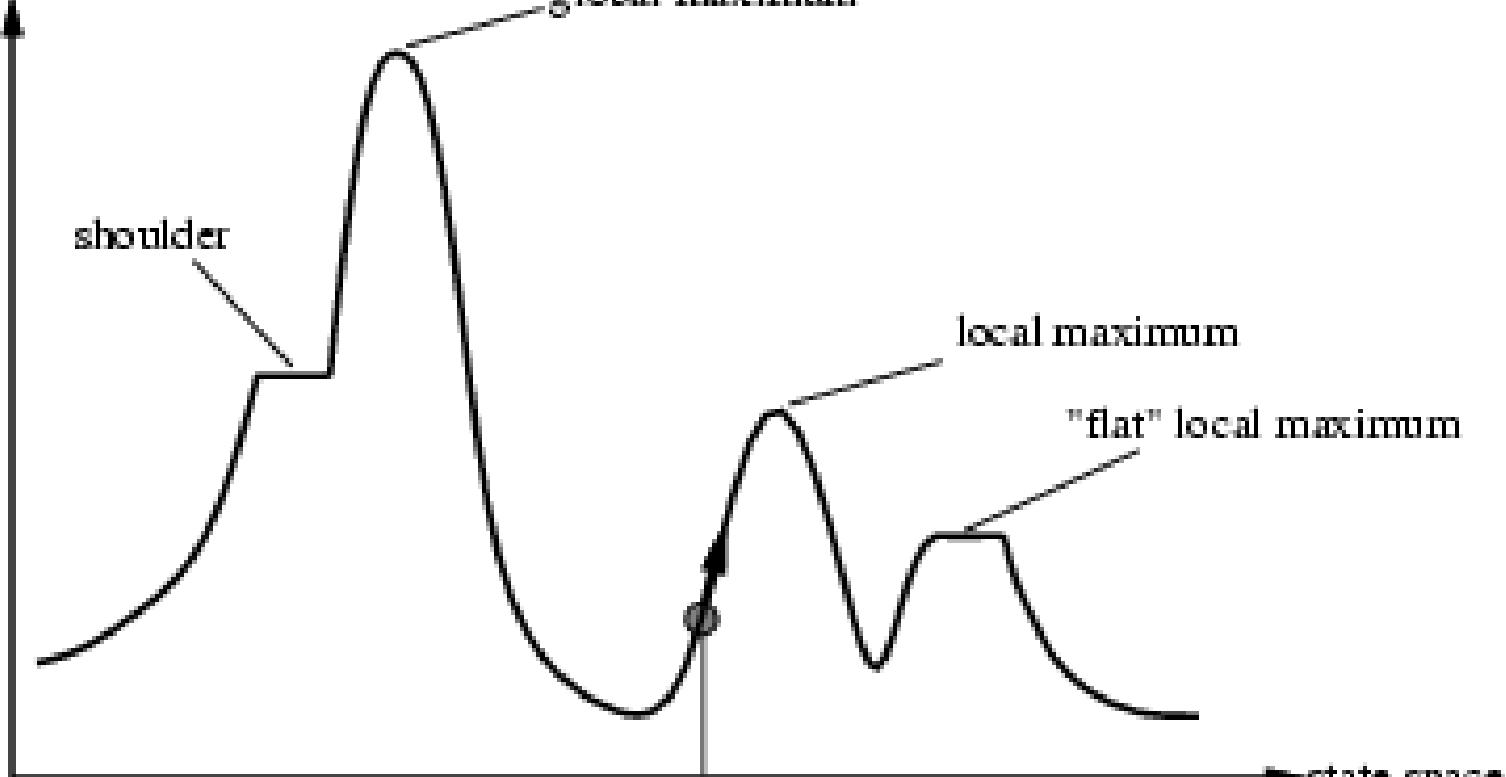
shoulder

local maximum

"flat" local maximum

current  
state

state space





# Simulated Annealing

- **Gradient descent:**
- Roll a *ping pong ball* on *bumpy surface* it will *rest at a local minimum*. Then shake the ball hard enough to dislodge it from *local minimum* to *global minimum*
  
- **Simulated Annealing –**
- Shake the ball hard (by increasing the temp in the beginning) & then gradually reduce the intensity of shaking (reduce the temp)



# Simulated Annealing - Algorithm

- Innermost loop is similar to HCS algorithm
- Instead of picking the nearest best move it picks a random move
- If it improves the situation, it is accepted
- Otherwise accept the move with a probability  $< 1$
- Probability decreases exponentially with badness of the move
- Probability also decreases as temp goes down
- Used extensively to solve VLSI layout problems

**Thanks!!!**

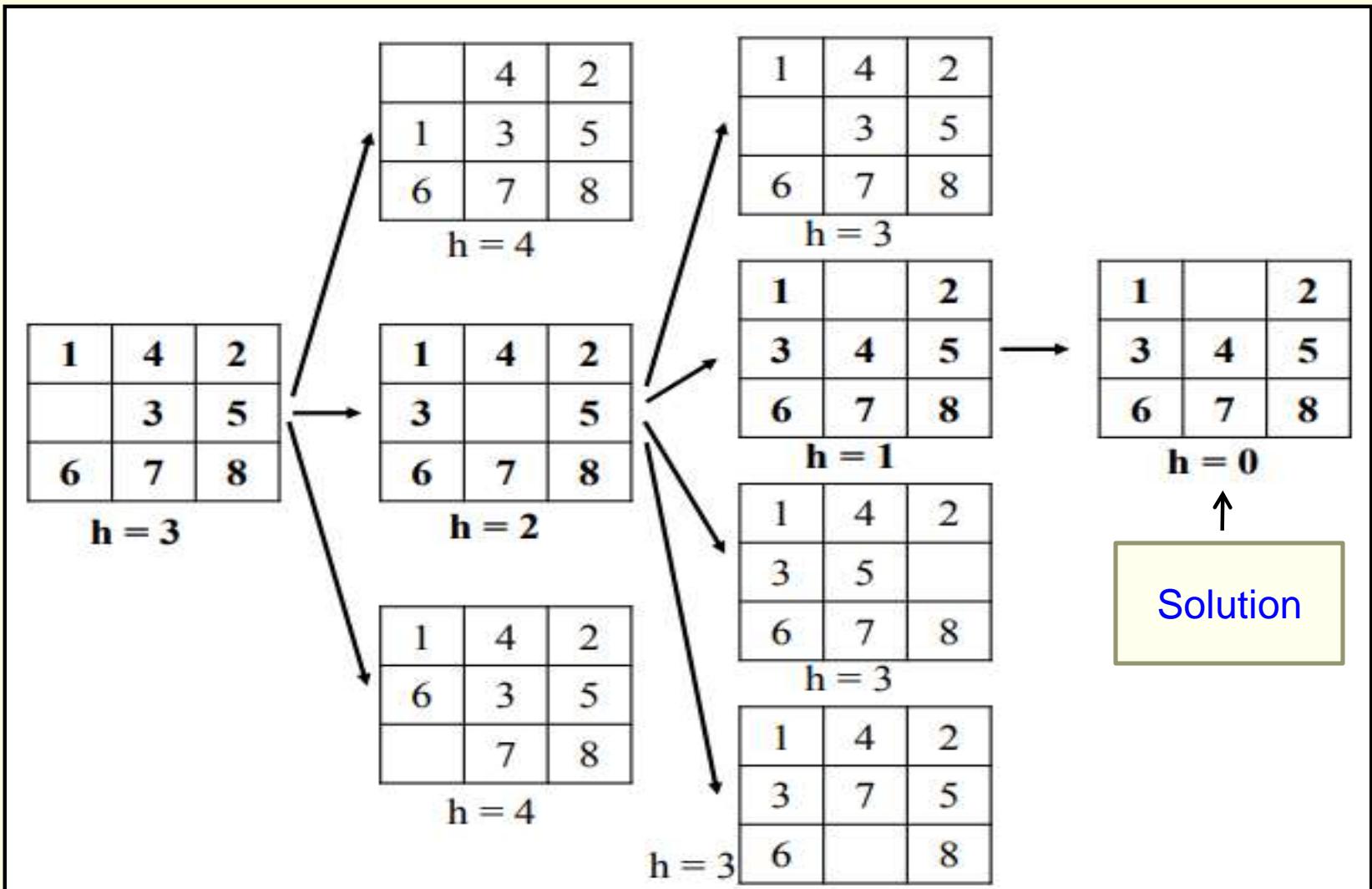
# Simulated Annealing - Algorithm

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to “temperature”

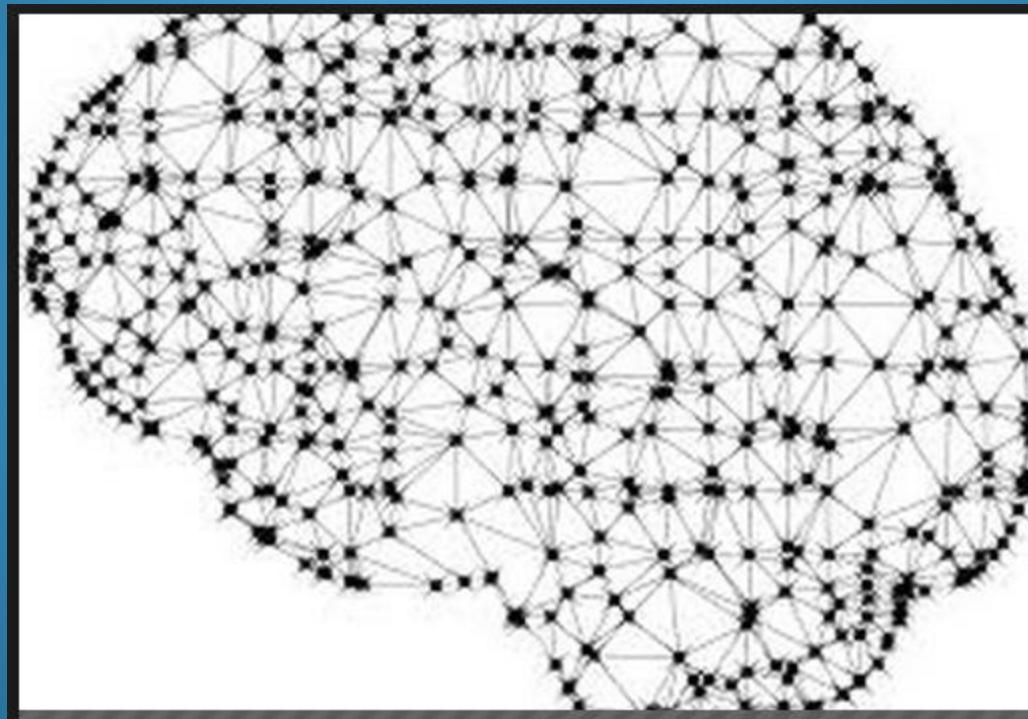
  current  $\leftarrow$  MAKE-NODE(problem.INITIAL-STATE)
  for  $t = 1$  to  $\infty$  do
     $T \leftarrow \text{schedule}(t)$ 
    if  $T = 0$  then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow \text{next.VALUE} - \text{current.VALUE}$ 
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$ 
```

**Figure 5** The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The *schedule* input determines the value of  $T$  as a function of time.

# An 8-puzzle problem solved by a HCS



# Informed Search - Exploration & Strategies



*Dr. Pulak Sahoo*

Associate Professor

Silicon Institute of Technology

# Contents

- **Introduction**
  - Heuristic Function
- **Informed Search Strategies**
  - Best First Search
    - Greedy Search
  - **A\* Search**
    - A\* Search – Algorithm
    - A\* Search – Conditions for Optimality
    - A\* Algorithm for 8-Puzzle problem
    - Memory bounded A\*: MA\*

# Introduction

- **Informed Search Strategy** uses
  - **Problem-Specific Knowledge** or **Domain Knowledge**
    - along with the **Problem definition**
      - to find **Solutions More Efficiently**
        - than **Uninformed search**
  - **Best-First Search** is an example of **Informed search**

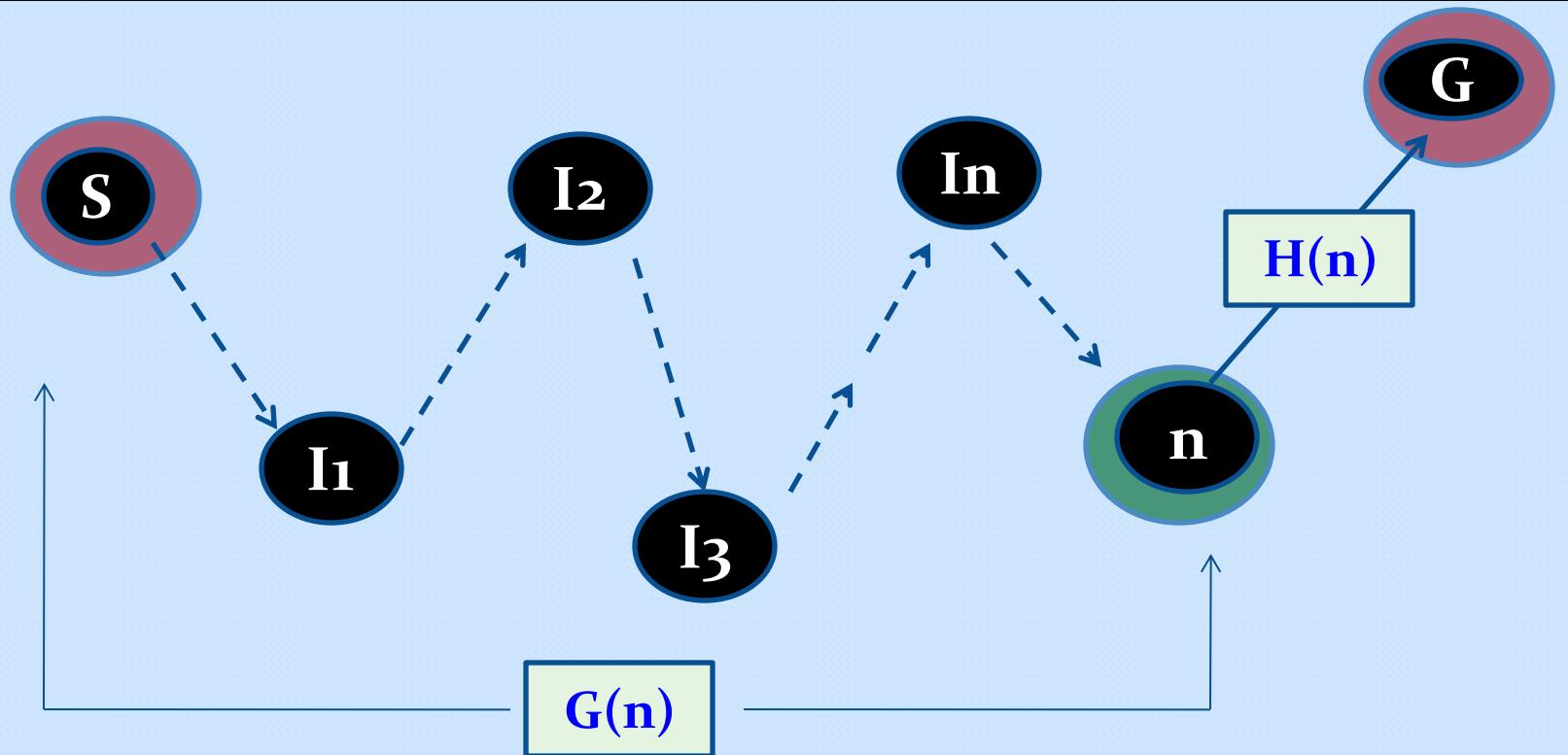
# Heuristic Function

- A **node** is selected for expansion based on the **Evaluation Function  $F(n)$**  <cost estimate>
  - The **node** with “lowest evaluation” is expanded first
  - Choice of “ **$F$** ” determines the **search strategy**
  - **Heuristic function** “ **$H(n)$** ” is a component of  **$F(n)$**
- **$H(n)$**  – Estimated cost of **cheapest path** to **goal** from node ‘**n**’
- If ( **$n = goal\ node$** ) then  **$H(n) = 0$**

- **$F(n) = H(n) + G(n)$**
- **$H(n)$**  – How far the goal is
- **$G(n)$**  – No. of nodes travelled from start node to current node

# Heuristic Function

- $F(n) = H(n) + G(n)$
- $H(n)$  – How far the goal **G** is
- $G(n)$  – No. of nodes travelled from **start node S** to **current node n**



# Domain Knowledge & Heuristic Function

## ○ Domain Knowledge is used for :

- For guiding the search & Generating next states
- Heuristics uses “domain specific knowledge” to estimate the “quality of potential solutions”

## • Examples of Heuristic Functions:

- Manhattan distance heuristic for “8 puzzle”
- Minimum Spanning Tree heuristic for “TSP”
- Heuristics are fundamental to “Chess programs”

- A strategy is defined by picking *the order of node expansion*

# 8-Puzzle (Initial & Goal States)

## Manhattan distance heuristic

Initial State		
1	2	3
	4	6
7	5	8

Goal State		
1	2	3
4	5	6
7	8	

### Operations

1. Up
2. Down
3. Right or
4. Left

Step cost - 1

- $F(n) = H(n) + G(n)$
- $H(n)$  – How far the goal is – **NO. OF MISPLACED TILES (4)**
- $G(n)$  – Number of nodes travelled from start node to current node

# The informed search problem

- **Problem definition:**

- Given: **[S, s, O, G, h]** where
  - **S** is the set of states
  - **s** is the start state
  - **O** is the set of state transition operators having some cost
  - **G** is the set of goal states
  - **h( )** is a heuristic function estimating the distance to a goal

- **Goal is to find:**
  - A minimum cost “sequence of transitions” to a goal state

# Evaluation Function ( $f(n)$ )

## for different search algorithms

- $f(n)$  – **Evaluation function** (Estimated cost from start node to goal node thru current node ‘n’)
- $h(n)$  – **Heuristics** – Estimated cost of how far the goal node is from current node ‘n’
- $g(n)$  – Cost so far from start node to node ‘n’

### • Greedy Search

- $f(n) = h(n)$  // heuristic (estimated cost) to goal node

### • Uniform Cost Search

- $f(n) = g(n)$  // path cost so far

### • A\* Search

- $f(n) = h(n) + g(n)$  // path cost so far + heuristic to goal node

# Greedy – Best first search

(pg-93)

- **Greedy search** is one of the “**Best First Search**” Algorithm
- Greedy best-first search algorithm always selects the path which appears **best** at that moment
  - *May not always find the optimal solution*
- A greedy search algorithm uses a **heuristic** for making locally optimal choices at each stage with the hope of finding a global optimum
- We will use of **Greedy search** to solve the **route-finding problem** from **Arad** to **Bucharest** (in Romania)

# Route Finding problem (Travelling Romania)

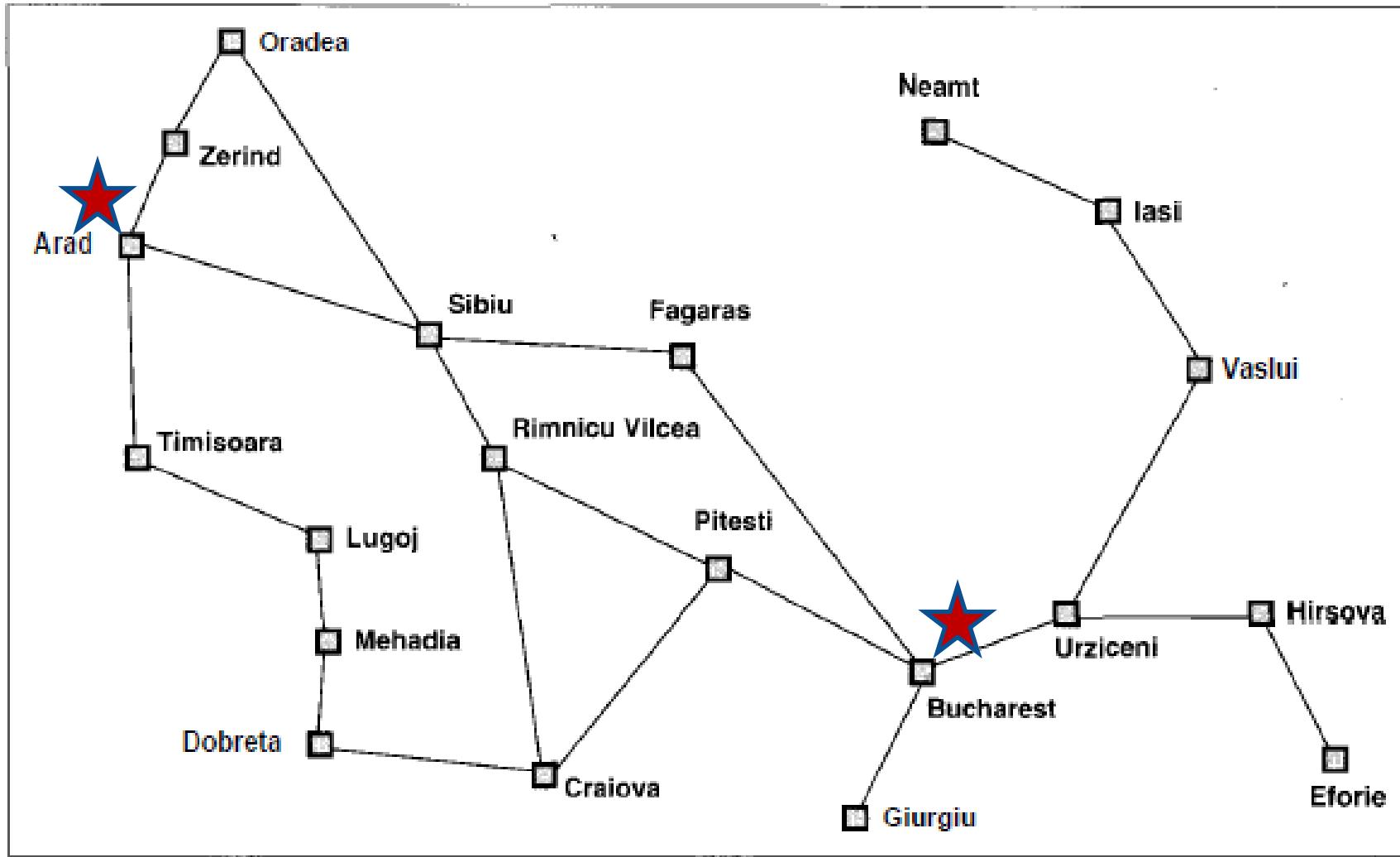


Figure 3.3 A simplified road map of Romania.

# ★ Greedy – Best first search (pg-93)

- Use of **Greedy search** to solve the route-finding problem from **Arad** to **Bucharest** (in Romania)
- **Heuristic** used =  $h_{sld}$  = straight line distance (based on experience)

- The initial state=**Arad**



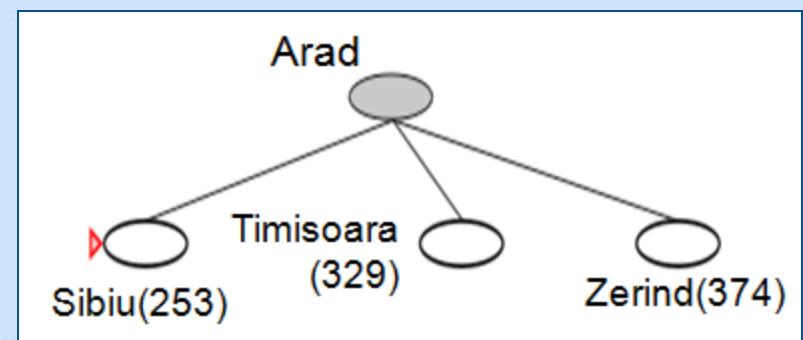
- $h_{sld}(In(Arad)) = 366$

- The first expansion step produces:

- Sibiu, Timisoara & Zerind

- **Greedy best-first** will select **Sibiu**
  - As the heuristic ( $h_{sld}$ ) is minimum

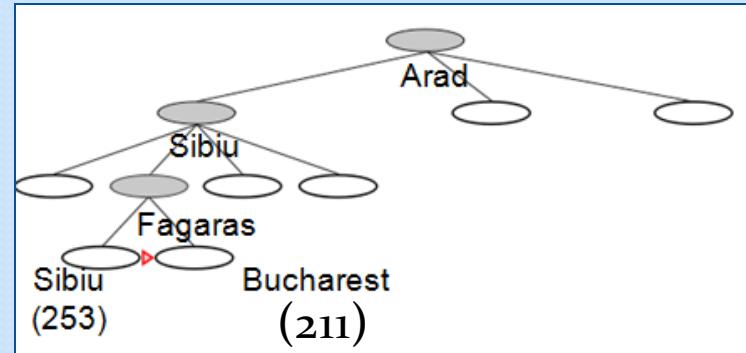
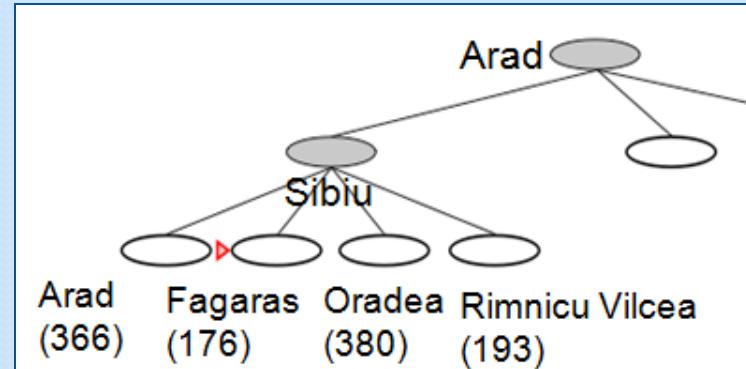
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	248
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374





# Greedy – Best first search

- Greedy best-first will select **Sibiu**
- If **Sibiu** is expanded we get:
  - Arad, Fagaras, Oradea & Rimnicu Vilcea
- Greedy best-first search will select: **Fagaras**
- If **Fagaras** is expanded we get:
  - Sibiu & Bucharest



# Route Finding problem (Travelling Romania)

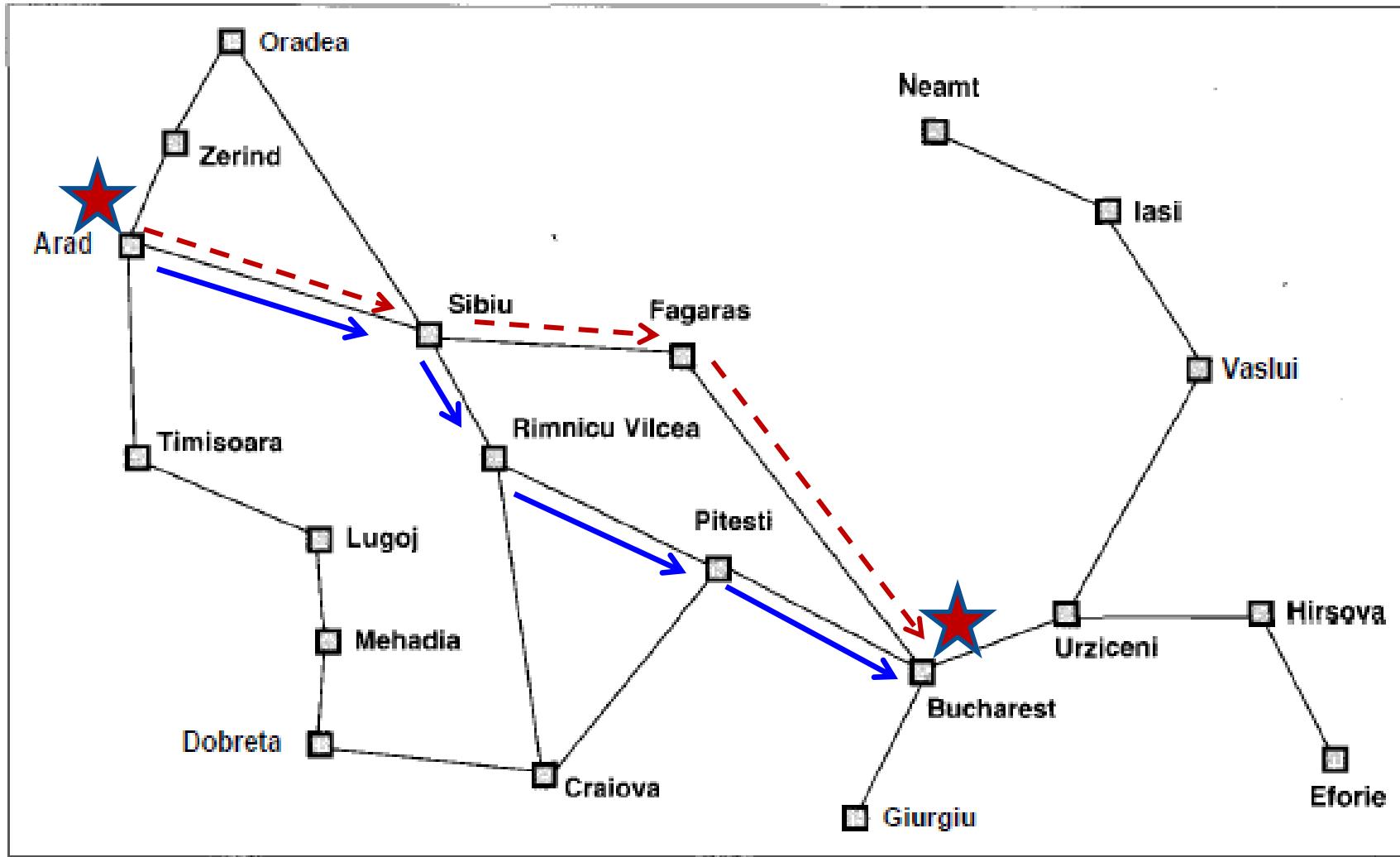


Figure 3.3 A simplified road map of Romania.

# Greedy – Best first search

- Goal reached !!!
  - Search **cost** is minimal (only expanded the nodes on solution path)
  - But not optimal (also incomplete) (also leads to false paths)
  - *Ex: Path Arad, Sibiu, Rimnicu Vilcea, Pitesti is 32 km shorter*

- Time & Space complexity for Tree is :  $O(b^m)$

- $m$  – max depth of the search tree
- With quality heuristics, space & time complexity can be reduced substantially



# A\* Search - Algorithm

- A\* Search is Best-known form of best-first search
- Idea: Avoid expanding already expensive paths
- Evaluation function  $f(n) = g(n) + h(n)$ 
  - $f(n)$  - Total cost of path from start node to “goal” through “n”
  - $g(n)$  - The cost of reaching the node “n” from “start”
  - $h(n)$  - Estimated cost of the cheapest solution through node “n” to the goal
- In order to get the **cheapest solution**, try the **node** with lowest value of  $g(n) + h(n)$
- *A\* search is – Complete & Optimal*

# A\* Search - Algorithm

**STEP-1 (Initialize)**: <Initialize the node lists - **OPEN** (currently known but not evaluated) & **CLOSED** (already evaluated)>

// Initially, only the start node is known & no node is evaluated

- Set **OPEN** = {s} & **CLOSED** = { }
- **g(s)** = 0 // The cost of reaching from start node to start node is zero
- **f(s)** = **h(s)** // For first node, the evaluation function is completely based on heuristic

**STEP-2 (Fail)**: <If nothing in **OPEN** node list, then terminate with failure>

- If **OPEN** = { }, then terminate with failure

**STEP-3 (Select)**: <If **OPEN** list has nodes, then select the cheapest cost node>

- Select the **minimum cost node n** from **OPEN** //like done in route finding problem
- Save **n** in **CLOSED** // Move n into already evaluated list 'CLOSED'

**STEP-4 (Goal test)**: <if node n is the **goal**, stop search, return Success & f(n)>

- If **n ∈ G**, terminate with success, return f(n)



## STEP-5 (Expand): <Generate the successors of node n>

- For each successor m of n // For each of the successors do below steps

// New Node - If m is neither in OPEN or CLOSED => neither known nor evaluated before

- If  $m \notin [\text{OPEN} \text{ or } \text{CLOSED}]$  // g(m) is not yet calculated
  - // Calculate evaluation function f(m)
    - Set  $g(m) = g(n) + \text{Cost}(n,m)$  // Find the distance from start to node m
    - Set  $f(m) = g(m) + h(m)$  // Evaluation function for node m is updated
    - Insert m in OPEN // Insert m in OPEN (known nodes list)

// Old Node - If m is either in OPEN or in CLOSED => either known or evaluated before

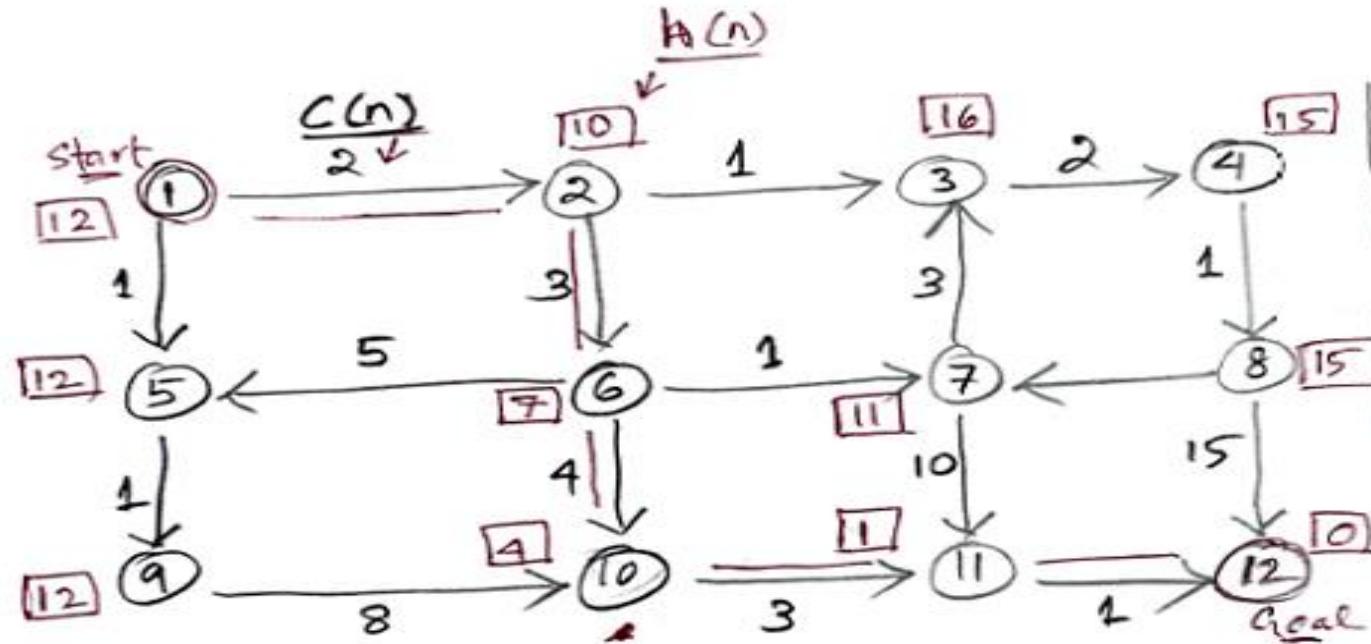
- If  $m \in [\text{OPEN} \text{ or } \text{CLOSED}]$  // g(m) is already calculated
  - Set  $g(m) = \min \{g(m), g(n) + \text{Cost}(n,m)\}$  // set g(m) to lowest value till now
  - Set  $f(m) = g(m) + h(m)$  // Calculate evaluation function f(m)

// If f(m) value of this node is less than earlier nodes at same level

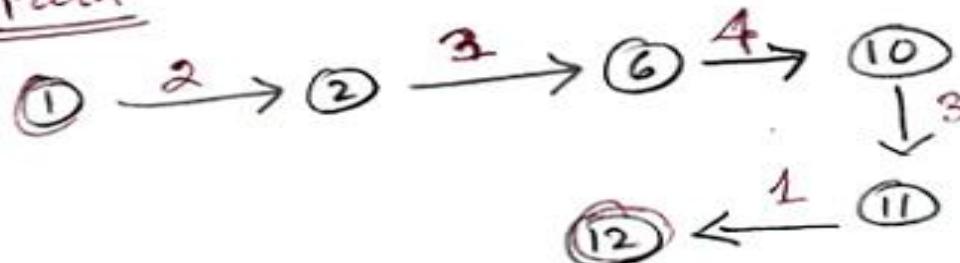
- If f(m) has decreased &  $m \in \text{CLOSED}$ 
  - move it to OPEN

**STEP-6 Loop:** Go To Step 2.

# A\* Search – Example-1

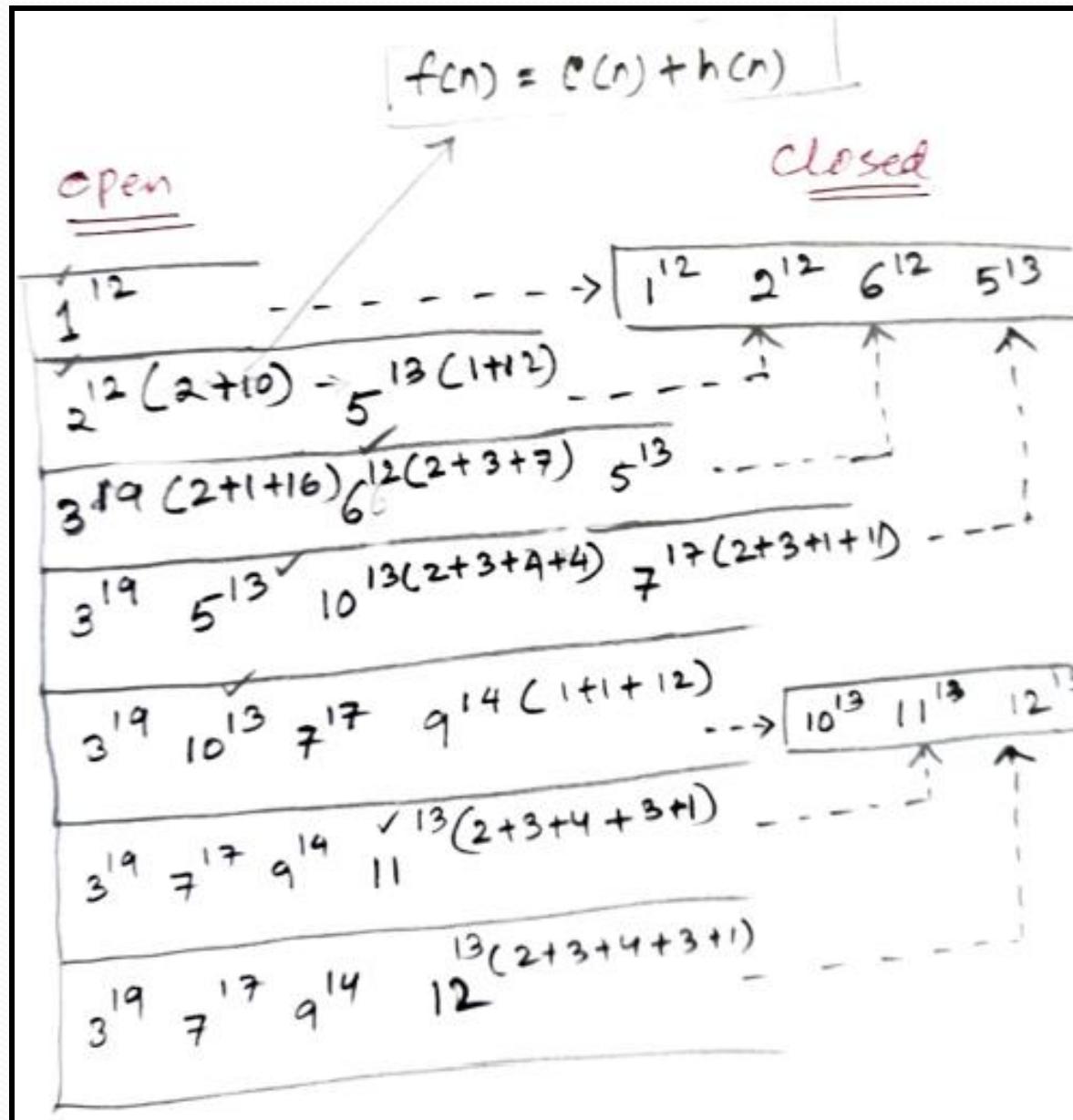


Path

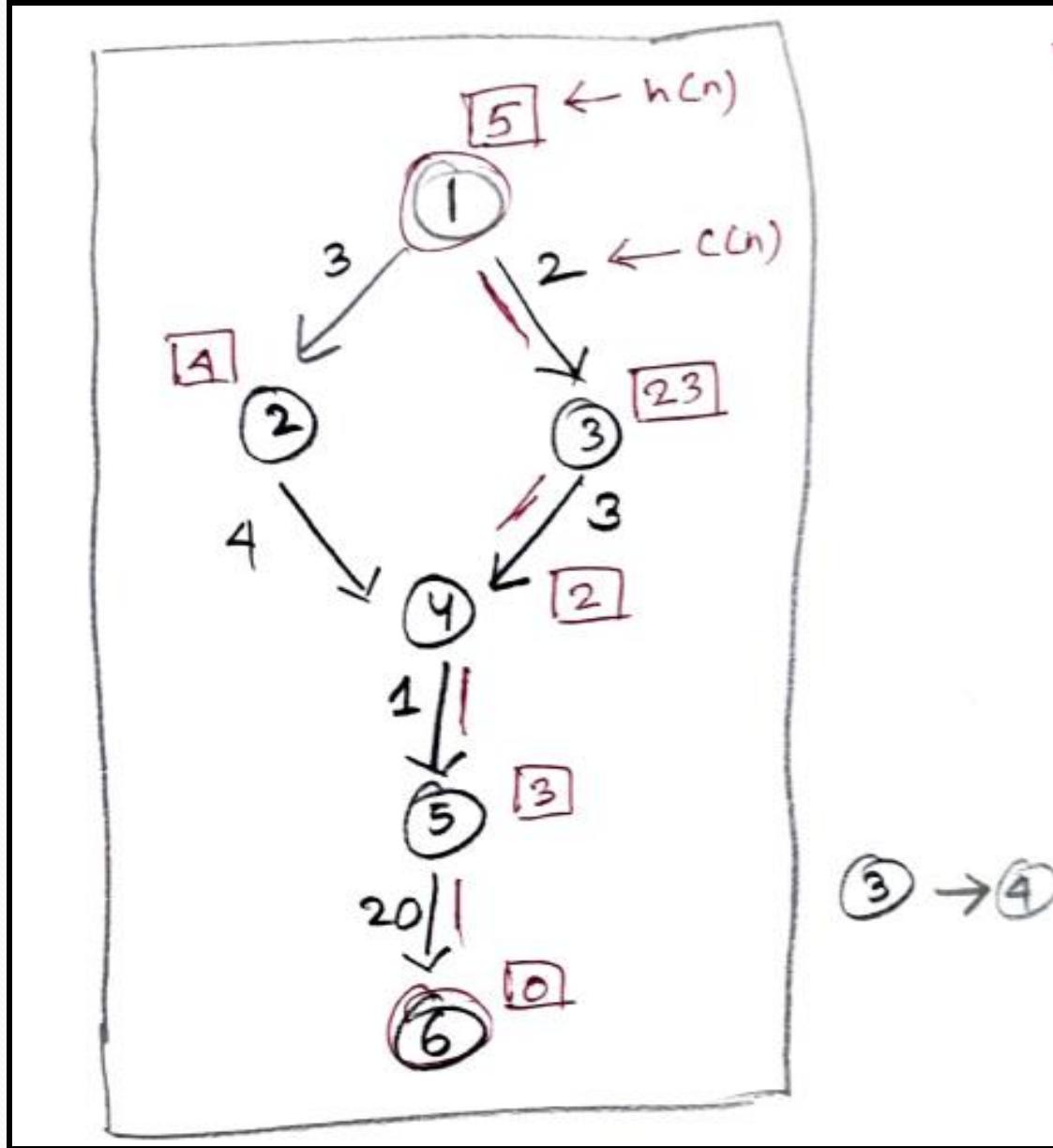


[Path cost = 13]

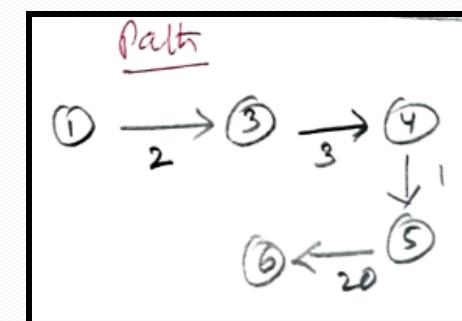
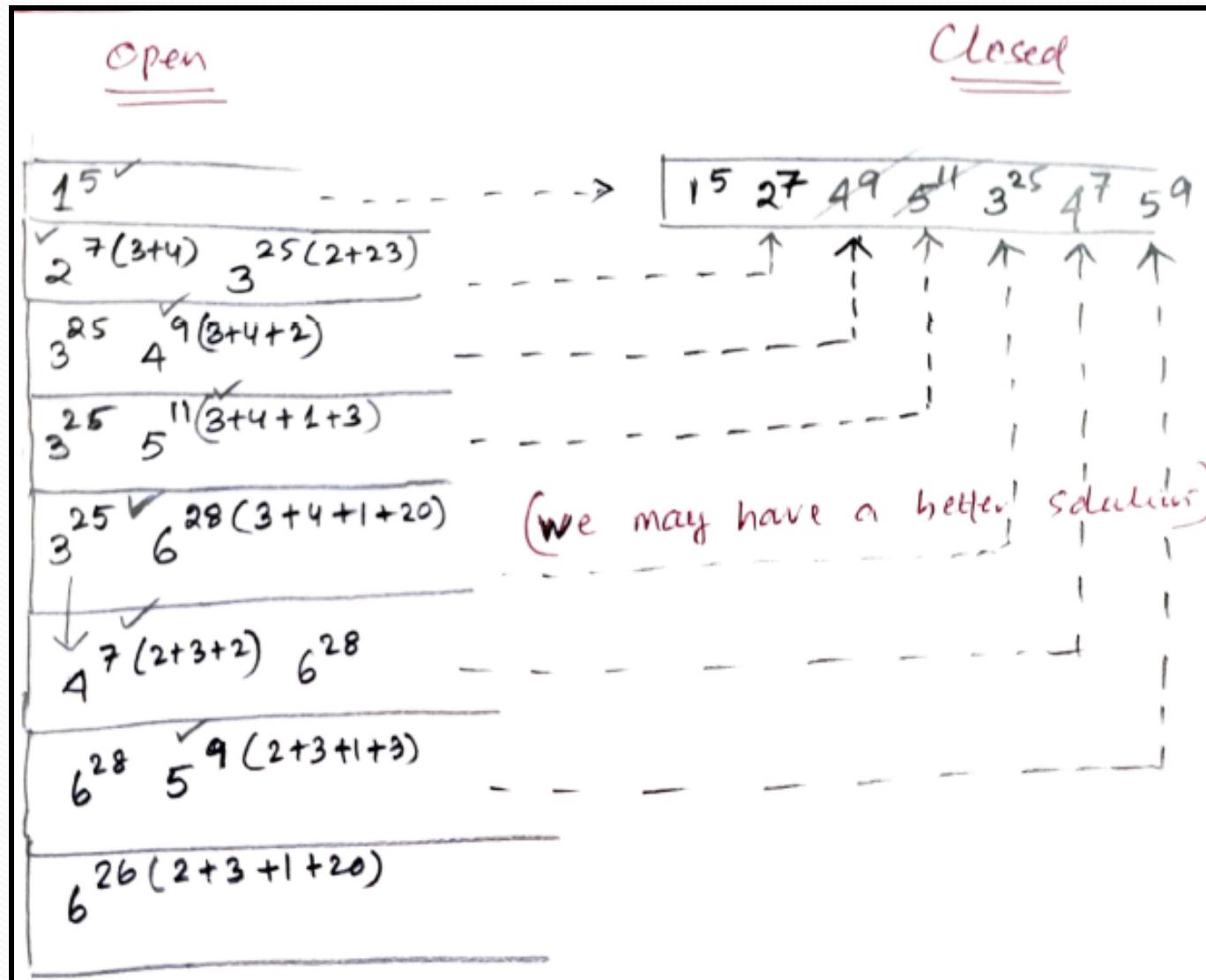
# A\* Search – Example-1



# A\* Search – Example-2



# A\* Search – Example-2



# **A\* Algorithm for 8-Puzzle problem**

# 8-Puzzle (Initial & Goal States)

Initial State	Goal State																		
<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td></td><td>4</td><td>6</td></tr><tr><td>7</td><td>5</td><td>8</td></tr></table>	1	2	3		4	6	7	5	8	<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr><tr><td>7</td><td>8</td><td></td></tr></table>	1	2	3	4	5	6	7	8	
1	2	3																	
	4	6																	
7	5	8																	
1	2	3																	
4	5	6																	
7	8																		

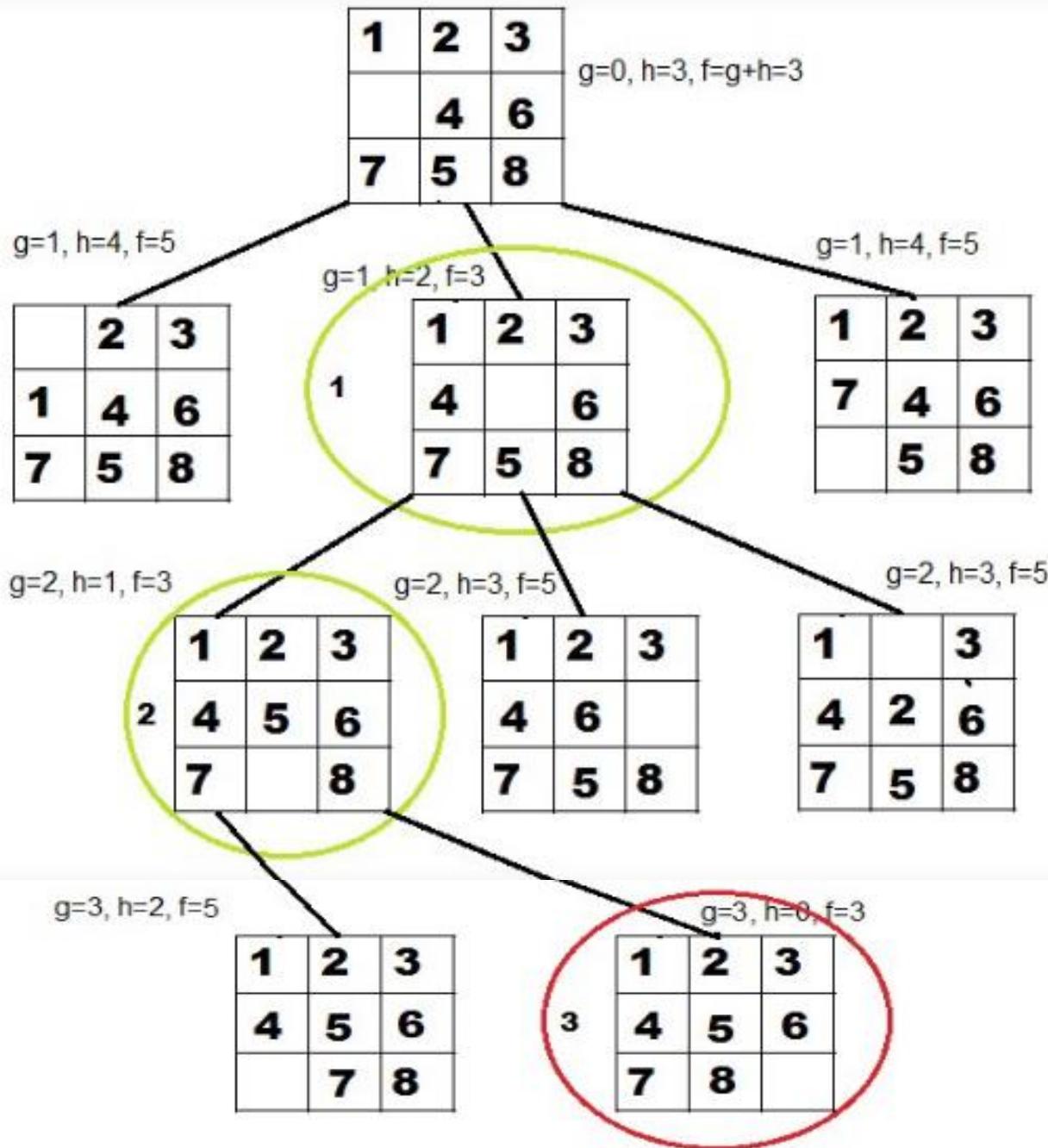
## Operations

1. Up
2. Down
3. Right or
4. Left

Step cost - 1

- $F(n) = H(n) + G(n)$
- $H(n)$  – How far the goal is – No. of misplaced tiles
- $G(n)$  – Number of nodes travelled from start node to current node

## 8-Puzzle (Solving steps)



# Memory bounded A\*: MA\*

- Working A\* within a given memory bound, M
  - Whenever  $|\text{OPEN} \cup \text{CLOSED}|$  approaches M
    - Some of the least promising states are removed
    - To guarantee that the algorithm terminates
      - we need to back up the cost of the most promising leaf of the subtree being deleted at the root of that subtree
    - Many variants of this algorithm have been studied
      - Recursive Best-First Search (RBFS) is a linear space version of this algorithm

A\* search for an instance of 8-puzzle with  $h_1$  (sum of misplaced tiles).

$g(n)$  assumes each move has a cost of 1.

Here we assume repeated state checking.

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



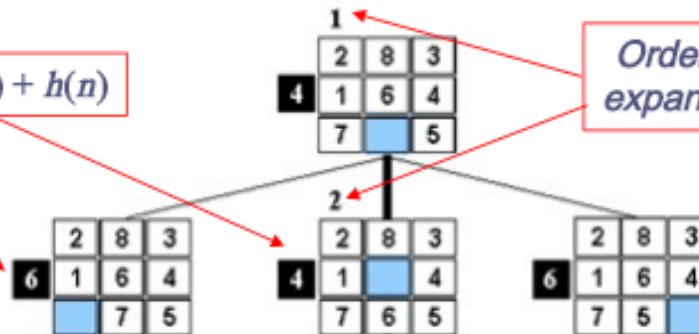
1		
2	8	3
1	6	4
7		5

A\* search for an instance of 8-puzzle with  $h_1$  (sum of misplaced tiles).

$g(n)$  assumes each move has a cost of 1.

Here we assume repeated state checking.

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

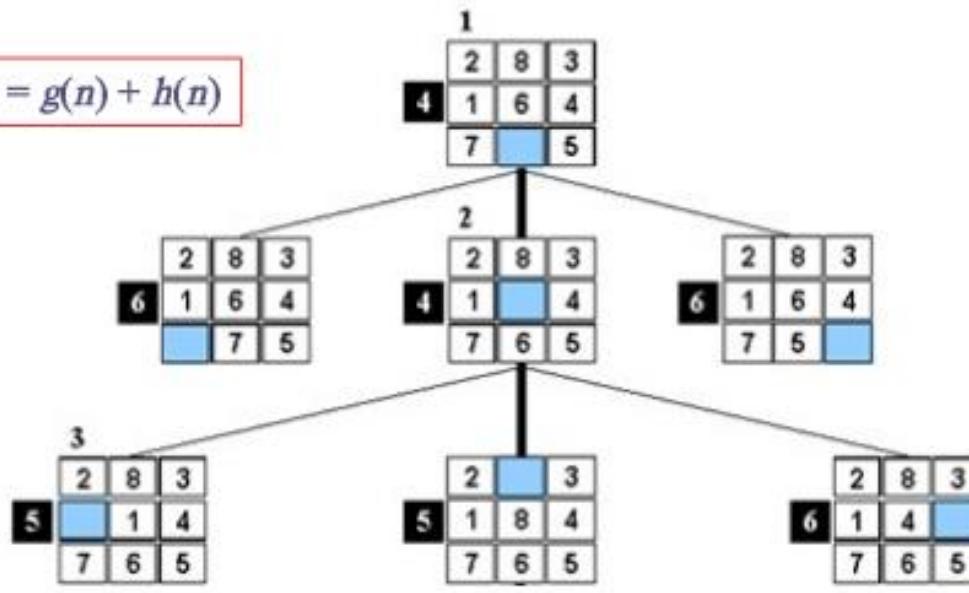


A\* search for an instance of 8-puzzle with  $h_1$  (sum of misplaced tiles).

$g(n)$  assumes each move has a cost of 1.

Here we assume repeated state checking.

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

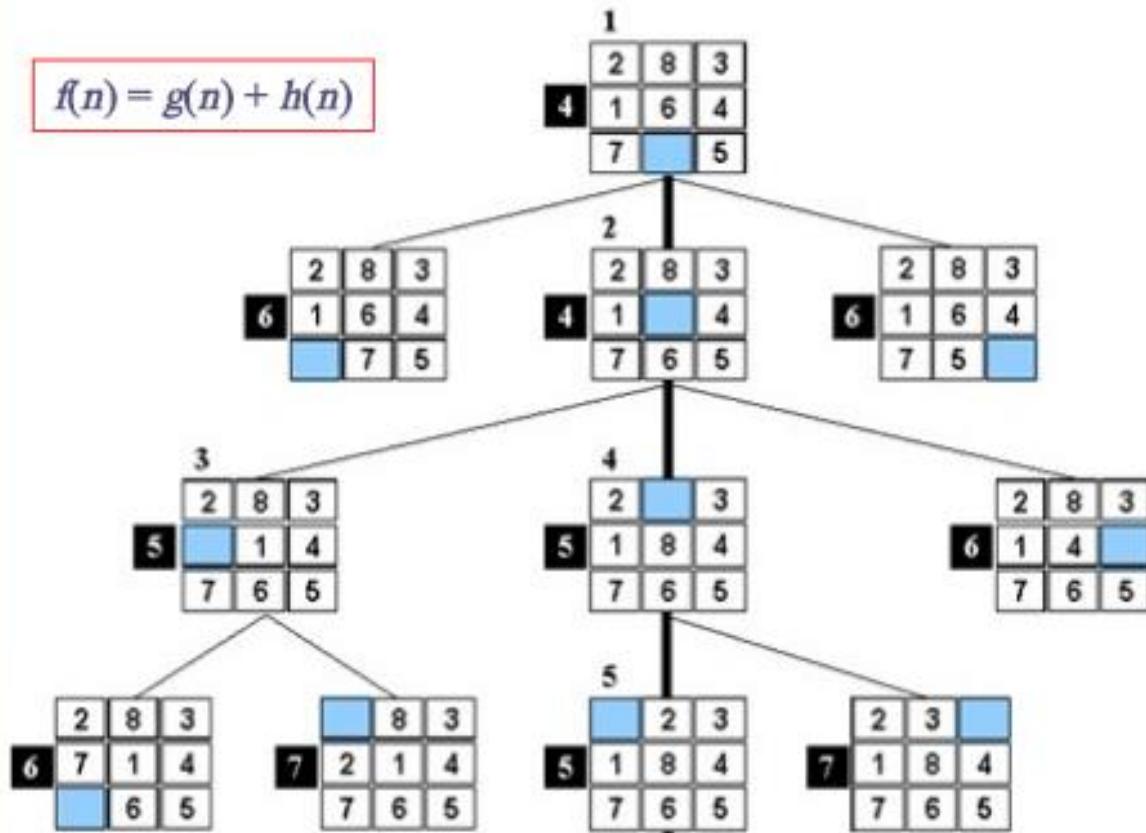


A\* search for an instance of 8-puzzle with  $h_1$  (sum of misplaced tiles).

$g(n)$  assumes each move has a cost of 1.

Here we assume repeated state checking.

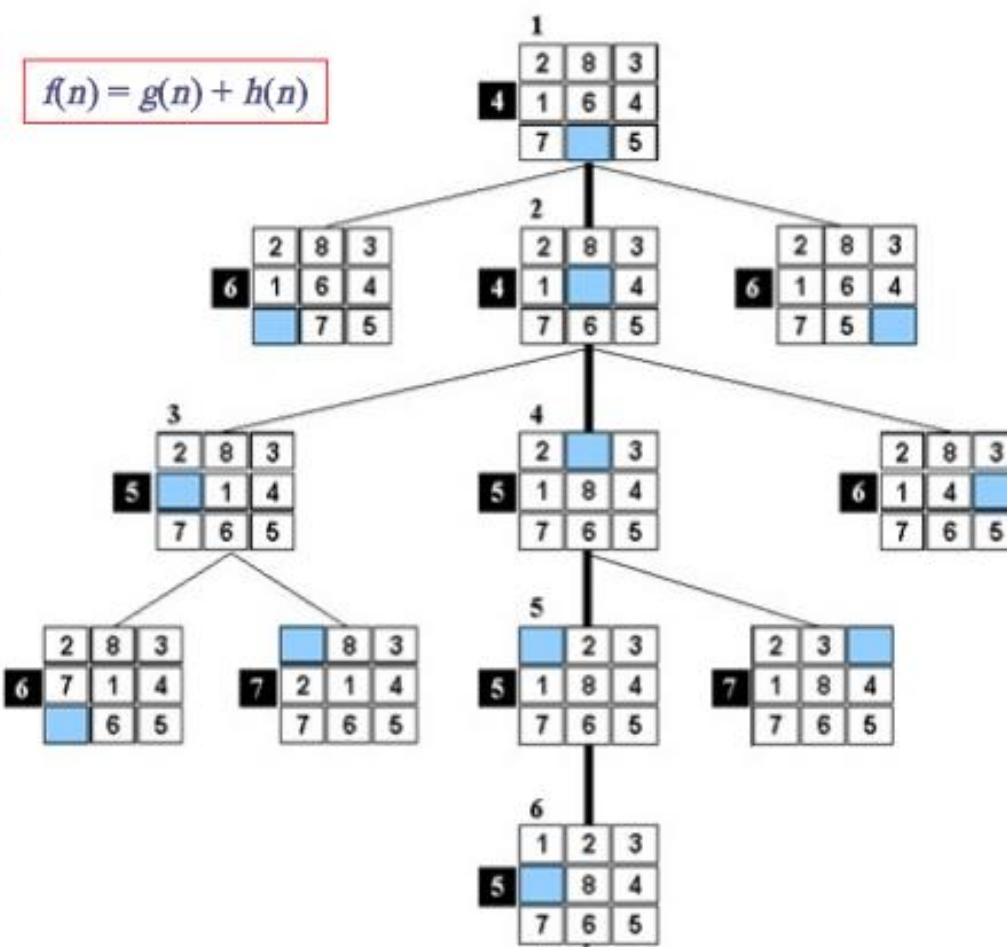
$$f(n) = g(n) + h(n)$$



A\* search for an instance of 8-puzzle with  $h_1$  (sum of misplaced tiles).

$g(n)$  assumes each move has a cost of 1.

Here we assume repeated state checking.



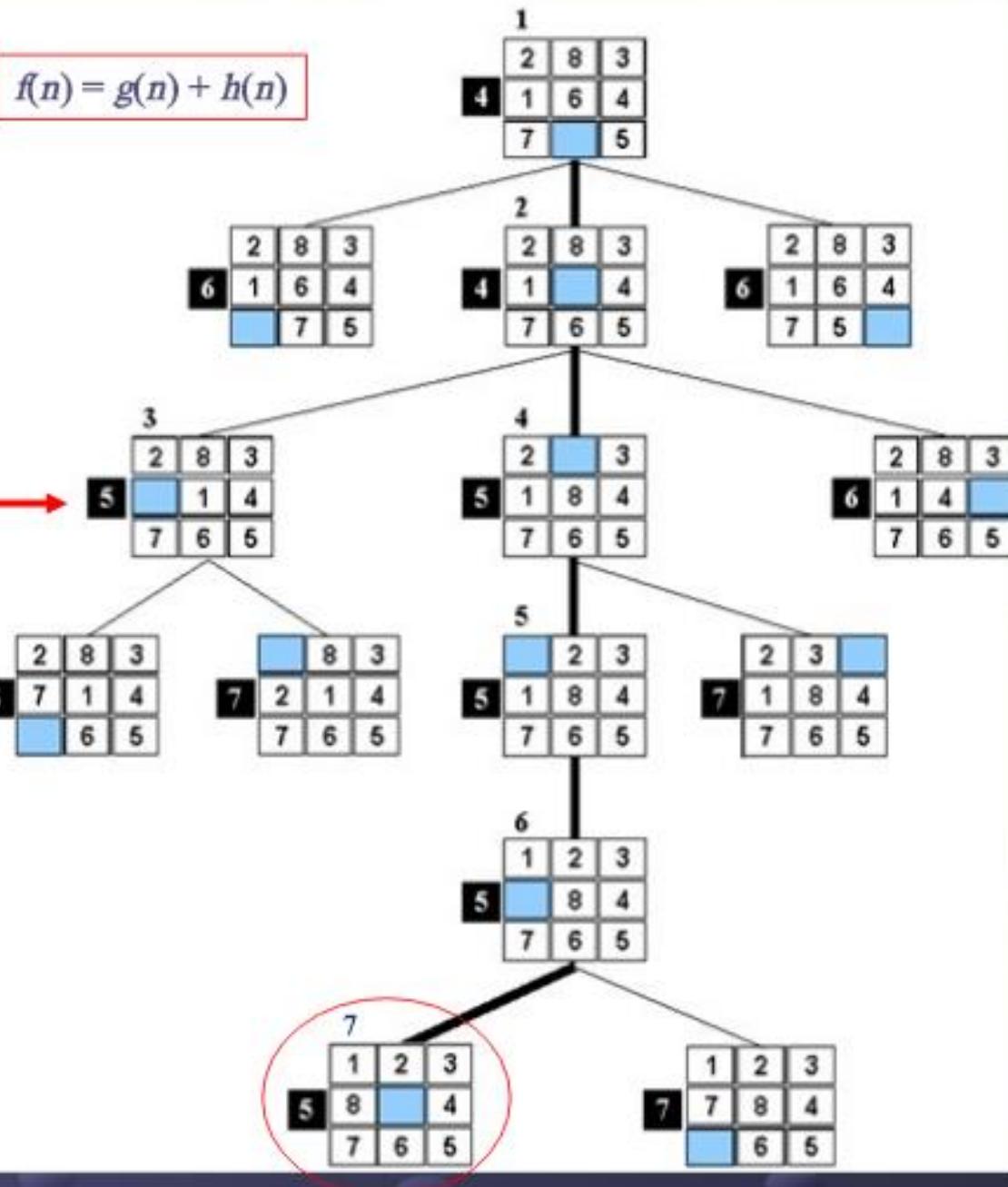
A\* search for an instance of 8-puzzle with  $h_1$  (sum of misplaced tiles).

$g(n)$  assumes each move has a cost of 1.

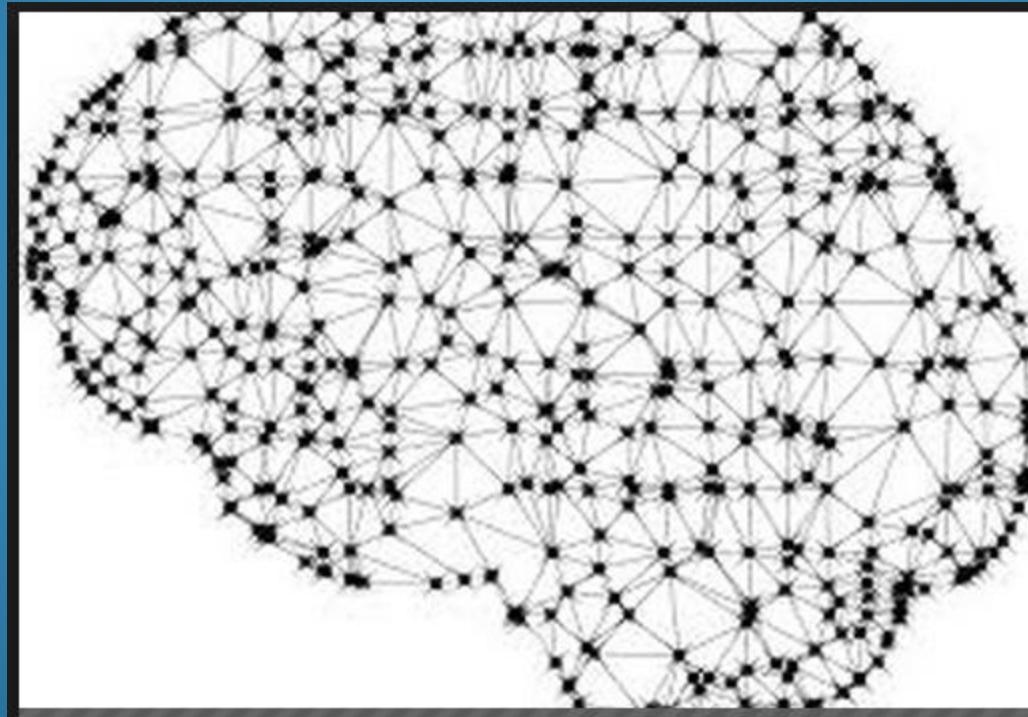
Here we assume repeated state checking.

Note: at level 2 there are two nodes listed with  $f(n) = 5$ . Depending on which node is we put in front of the queue, the algorithm will either expand 6 or 7 nodes. Here we have assumed the worse case, and thus the tree shows that 6 nodes were expanded

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



# Informed Search - Exploration & Strategies



*Dr. Pulak Sahoo*

Associate Professor

Silicon Institute of Technology

# Contents

- **Introduction**

- Heuristic Function & Evaluation Function

- **Informed Search Strategies**

- Best First Search

- Greedy Search

- **A\* Search**

- A\* Search – Algorithm

- A\* Search – Conditions for Optimality

- A\* Algorithm for 8-Puzzle problem

- Memory bounded A\*: MA\*

- Searching with partial information

# A\* Search – Conditions for Optimality

- **1<sup>st</sup> Condition for Optimality – Admissibility**
  - $h(n)$  should be an **admissible** heuristic
  - **Optimistic** - it never over-estimates cost to reach the goal
  - $\Rightarrow f(n) = g(n) + h(n)$  – never overestimates true cost of solution
  - Ex: Straight Line distance to reach Bucharest  $h_{SLD}$
- **2<sup>nd</sup> Condition for Optimality – Consistency (monotonicity)**
  - Required only for A\* Graph Search applications
  - $h()$  is monotonic if:  $h(n) - h(m) \leq c(n,m)$ , where m is a successor of n
  - If monotonicity is satisfied, then A\* has already found an optimal path to the current node
  - The f-values of the nodes expanded by A\* is non-decreasing

# Searching with partial information

This occurs If the environment is not fully observable or deterministic.

Given below are two reasons for this:

## 1. Sensorless problems

If the agent has no sensors:

Then the agent cannot know it's current state

It would have to take repeated action paths to ensure that  
the goal state is reached regardless of it's initial state.

## 2. Contingency problems

Occurs when the env. is partially observable or when actions are uncertain

After each action the agent needs to verify what effects that action has caused

Rather than planning for every possible contingency after an action, it is usually better to start acting and see which contingencies do arise.



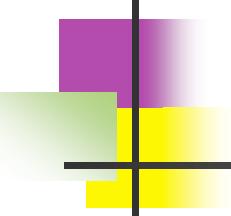
# Constraint Satisfaction Problems



***Dr. Pulak Sahoo***

Associate Professor

Silicon Institute of Technology



# Contents

---

- Constraint Satisfaction Problems (CSP)
- Backtracking search for CSPs
- Local search for CSPs

# Constraint satisfaction problems (CSPs)

- **In Standard search problem:**

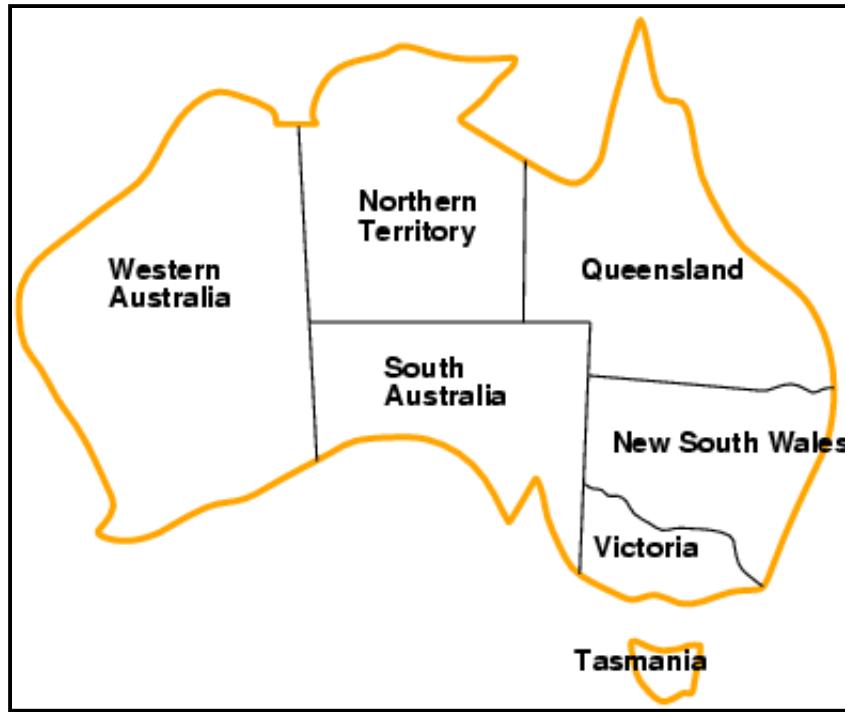
- A **State** (*Ex: city (Arad) or water jug (3,2)*) contains all of the info necessary :
  - to predict the effects of an action
    - *Transition operator – Go (Zerind)*
  - to determine if it is a goal state
    - *In (Bucharest) ?*
- Requires a **Data structure** that supports *state info, heuristic function & goal test*

# Constraint satisfaction problems (CSPs)

- A **Constraint Satisfaction Problem (CSP)** consists of
  - **States** - a set of variables –  $X_i$ ,
  - **Domain** for each variable –  $D_i$  ( $X_i$  can take values from  $D_i$ )
  - **Goal test** - a set of constraints
- The aim is to **choose a value** for each variable so that the resulting possible world **satisfies the constraints**
  - **Example:** *Map-Coloring*

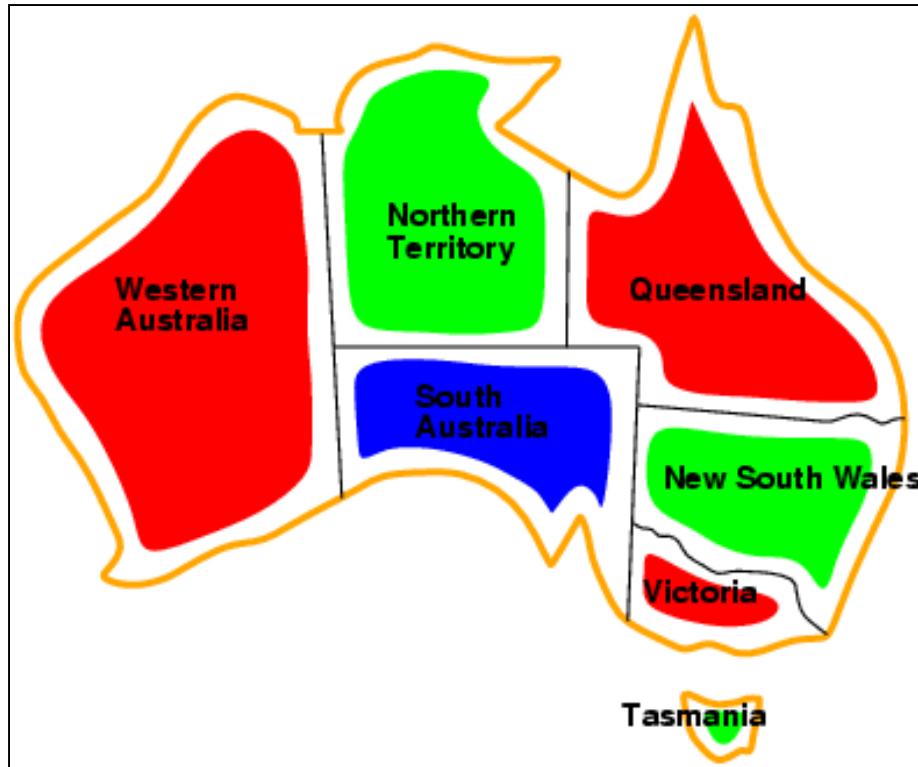
# Example: Map-Coloring

Australia



- **Variables**  $WA, NT, Q, NSW, V, SA, T$
- **Domains**  $D_i = \{\text{red, green, blue}\}$
- **Constraints:** Adjacent regions must have different colors
- **Ex:**  $WA \neq NT$ , or  $(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}$

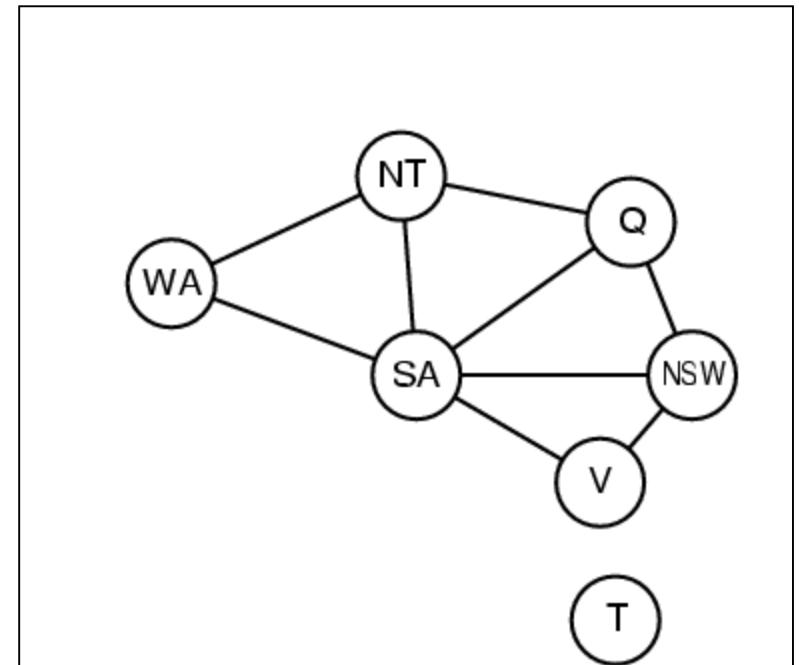
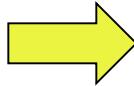
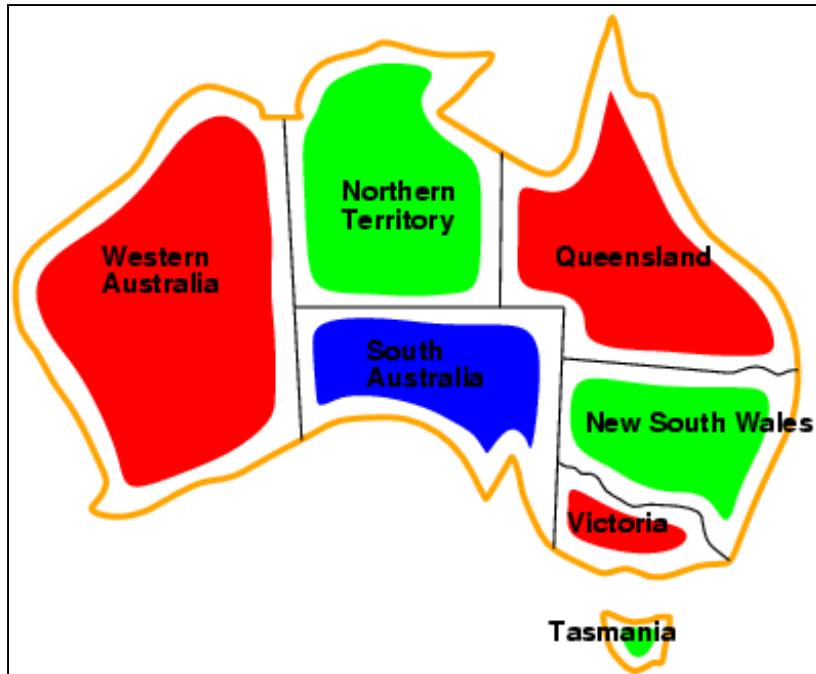
# Ex: Map-Coloring - solution



- **Solutions** are **complete** & **consistent** assignments
- **Ex:** WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

# Constraint graph

- **Binary CSP:** each constraint relates two variables (*Ex: map coloring problem*)
- **Constraint graph:** nodes are variables, arcs are constraints



# Varieties of CSPs

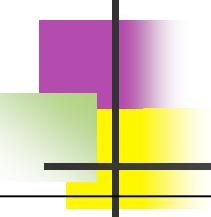
## ■ Discrete variables

### ■ **Finite domains:**

- $n$  variables, domain size  $d$  (*Ex: map coloring problem*)
  - $O(d^n)$  complete assignments
- **Ex:** Boolean CSPs (NP-complete)
- Formula " $a$  AND NOT  $b$ " is satisfiable because one can find the values  $a = \text{TRUE}$  and  $b = \text{FALSE}$ , which make  $(a \text{ AND NOT } b) = \text{TRUE}$

### ■ **Infinite domains:**

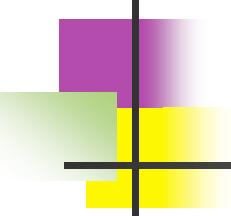
- Integers, strings variables etc.
- **Ex:** job scheduling, variables are start/end days for each job
  - need a constraint language, e.g.,  $\text{StartJob}_1 + 5 \leq \text{StartJob}_3$



# Varieties of CSPs

- **Continuous variables**

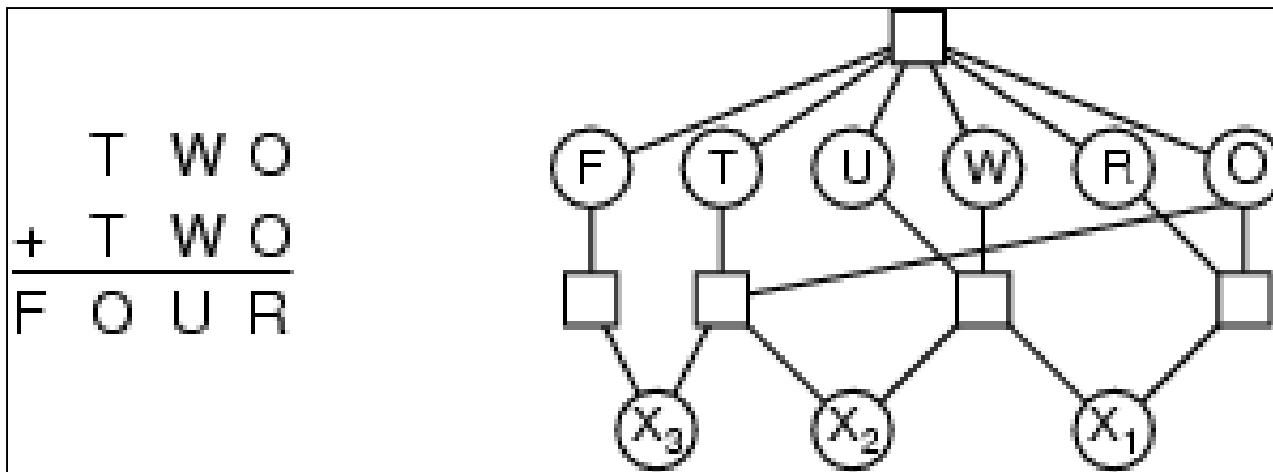
- **Ex1:** start/end times for Hubble Space Telescope observations (*time is variable in a continuous domain*)
- **Ex2:** linear constraints solvable in polynomial time by linear programming
- **Ex3: Finding the intersections of two circles**
- Find the Cartesian coordinates of all the intersection points of the two circles
- **Hubble** is solar-powered telescope that takes sharp pictures of objects in the sky such as planets, stars & galaxies. **Hubble** has made more than 1 million observations including pictures of the birth & death of stars, galaxies billions of light years away.



# Varieties of constraints

- **Unary** constraints involve a single variable,
  - *e.g.,  $SA \neq \text{green}$*
- **Binary** constraints involve pairs of variables,
  - *e.g.,  $SA \neq WA$*
- **Higher-order** constraints involve 3 or more variables,
  - *e.g., cryptarithmetic column constraints*

# Example: Cryptarithmetic

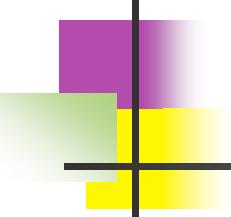


**Variables:**  $F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3$

**Domains:**  $\{0,1,2,3,4,5,6,7,8,9\}$

**Constraints:**  $\text{Alldiff}(F, T, U, W, R, O)$

- $O + O = R + 10 \cdot X_1$
- $X_1 + W + W = U + 10 \cdot X_2$
- $X_2 + T + T = O + 10 \cdot X_3$
- $X_3 = F, T \neq 0, F \neq 0$



# Real-world CSPs

- Assignment problems
  - *Ex: who teaches in what class*
- Timetabling problems
  - *Ex: which class is offered when and where?*
- Transportation scheduling
- Factory scheduling

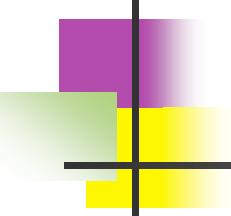
Notice that many real-world problems involve real-valued variables

# Standard search formulation (incremental)

Let's start with the straightforward approach, then fix it

States are defined by the values assigned so far

- **Initial state:** Empty assignment { }
- **Successor function:** Assign a value to an unassigned variable that does not conflict with current assignments → fail if no legal assignments
- **Goal test:** The current assignment is complete
- *Ex: Which teacher teaches in which class*
  1. Every solution appears at depth  $n$  with  $n$  variables → use DFS
  2.  $b = (n - \ell)d$  at depth  $\ell$ , hence  $n! \cdot d^n$  leaves



# Backtracking search

- Variable assignments are **commutative**
  - *Ex: [WA = red then NT = green] same as [NT = green then WA = red]*
- Only need to consider assignments to a single variable at each node
- **Depth-first search** for CSPs with single-variable assignments is called **Backtracking** search
- **Backtracking search** is the basic uninformed algorithm for **CSPs**
- Can solve  $n$ -queens for  $n \approx 25$

# Backtracking search - Algorithm

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({}, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or
failure
    if assignment is complete then return assignment
    var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(Variables[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to Constraints[csp] then
            add { var = value } to assignment
            result  $\leftarrow$  RECURSIVE-BACKTRACKING(assignment, csp)
            if result  $\neq$  failure then return result
            remove { var = value } from assignment
    return failure
```

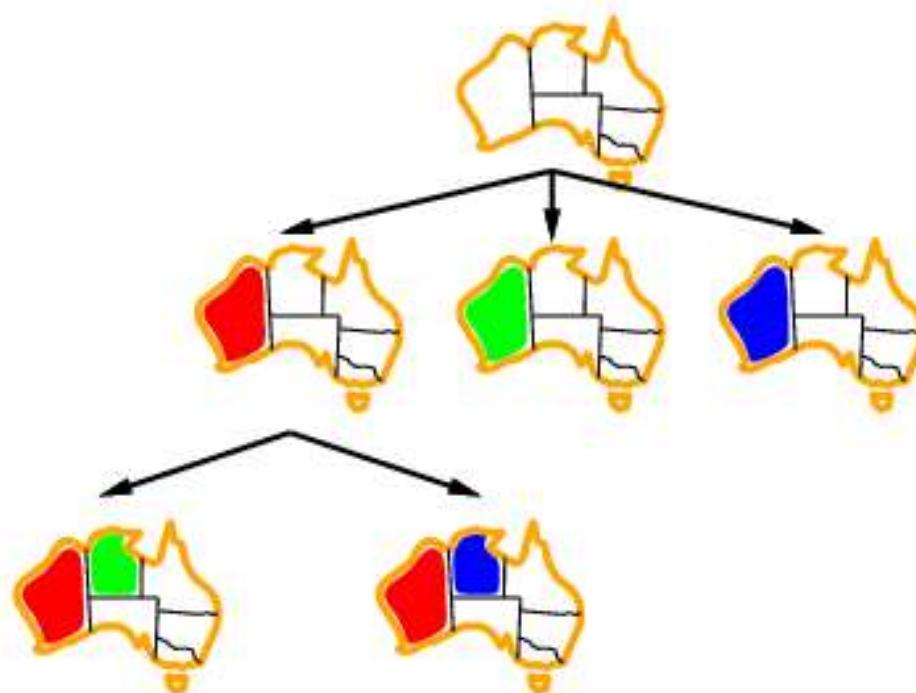
# Backtracking example



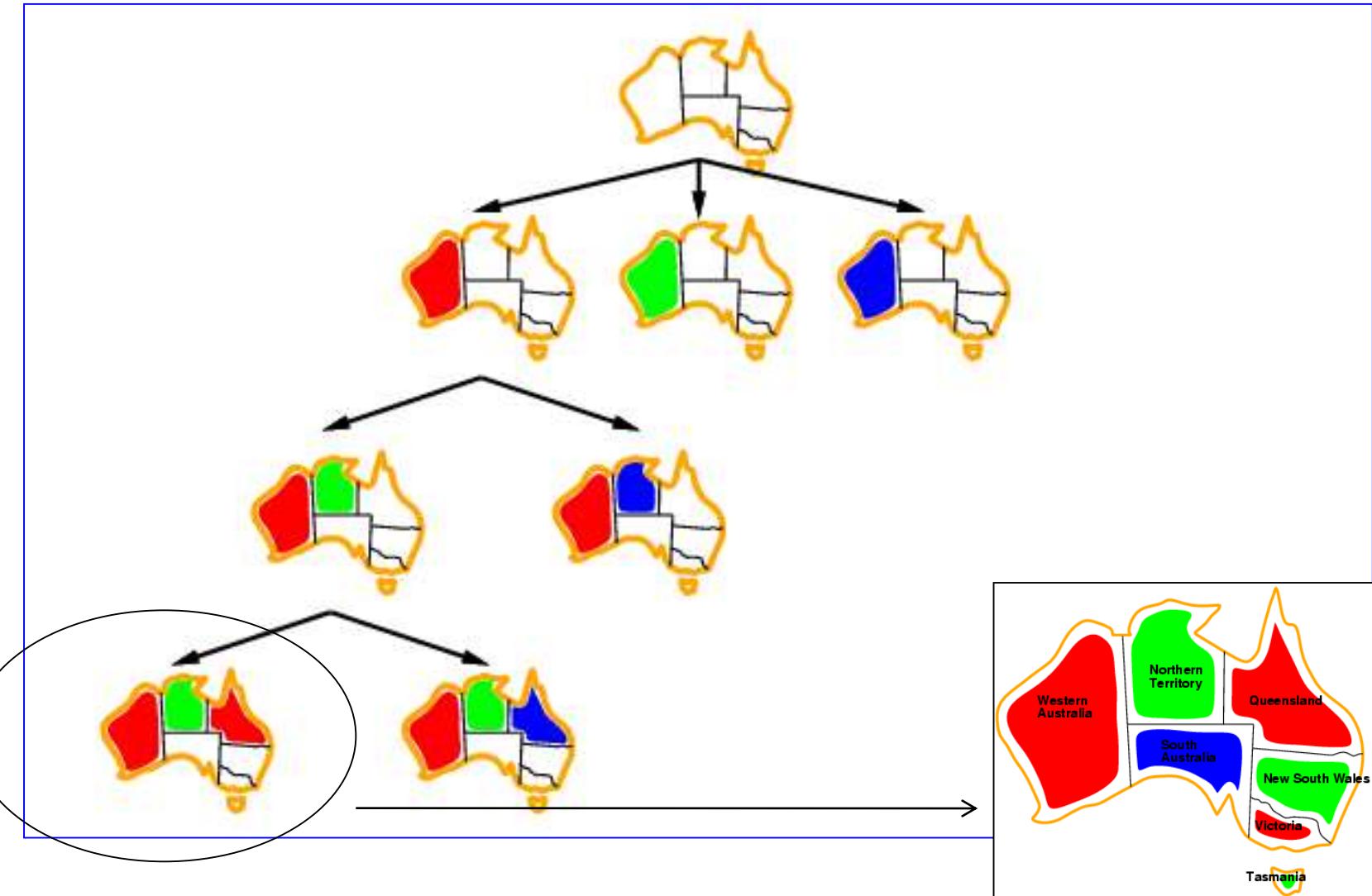
# Backtracking example

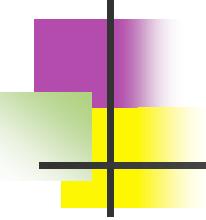


# Backtracking example



# Backtracking example



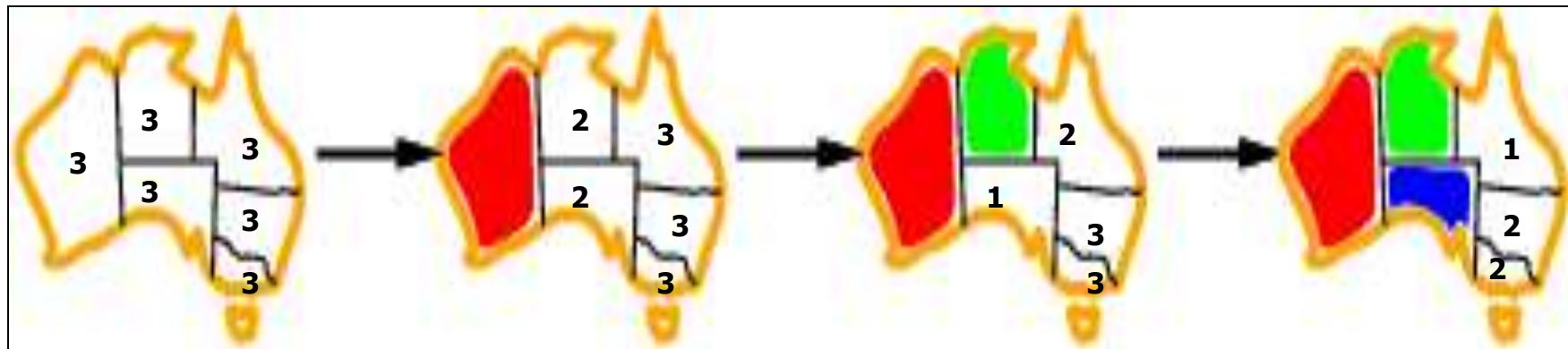


# Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
  - Which variable (province) should be assigned next?
  - In what order (color) should its values be tried?
  - Can we detect inevitable failure early?

# Most constrained variable

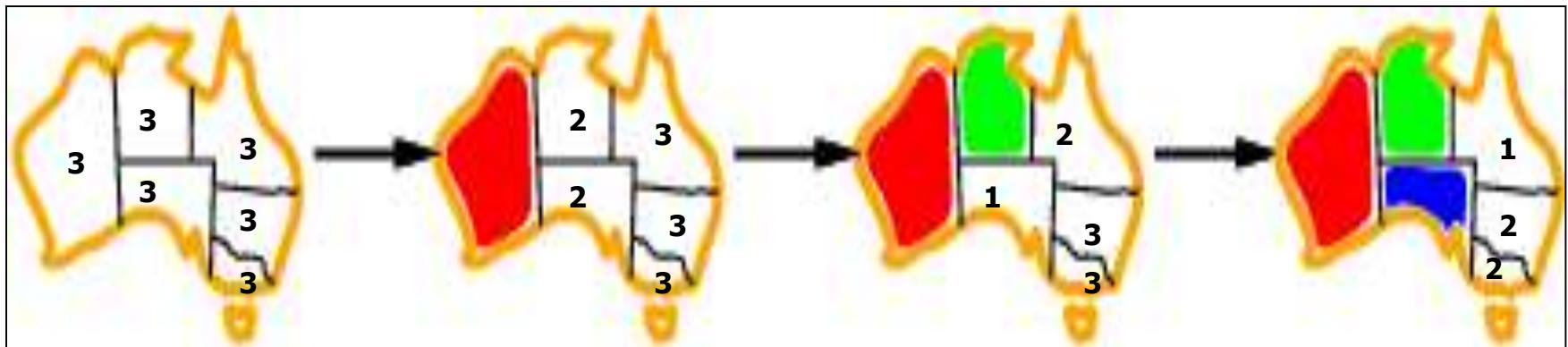
- Most constrained variable:  
choose the variable with the **fewest legal values**



- Ex: **minimum remaining values (MRV)** heuristic

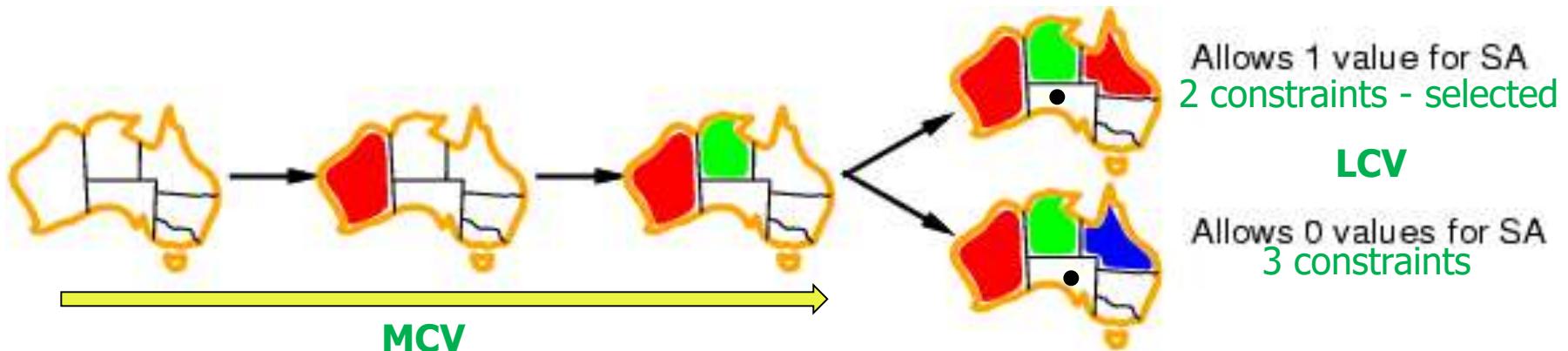
# Most constraining variable

- Tie-breaker among most constrained variables
- Most constraining variable:
  - Choose the variable with the most constraints on remaining variables



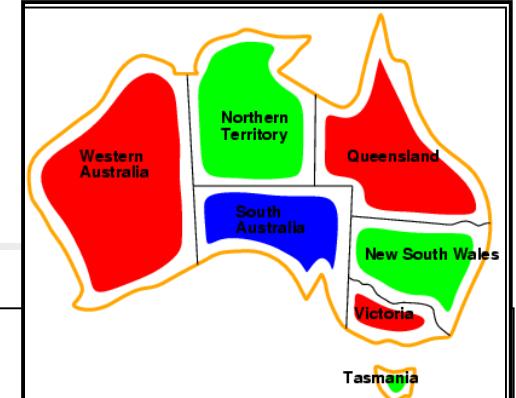
# Least constraining value

- Given a variable, choose the **least constraining value**: the one that rules out the fewest values in the remaining variables



- Combining these heuristics makes 1000 queens feasible

# Forward checking



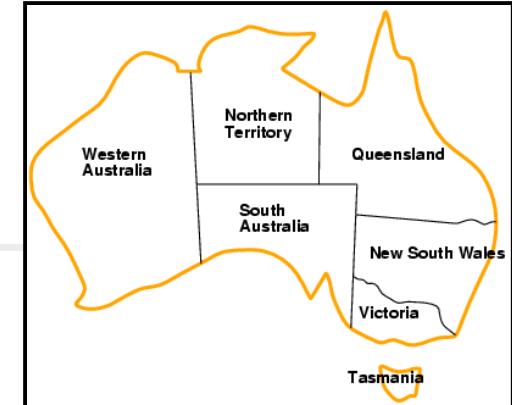
## Idea:

- Keep track of **remaining legal values** for **unassigned variables**
- Terminate search** when **any variable has no legal values**



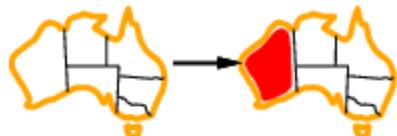
Possible colors in each province

# Forward checking



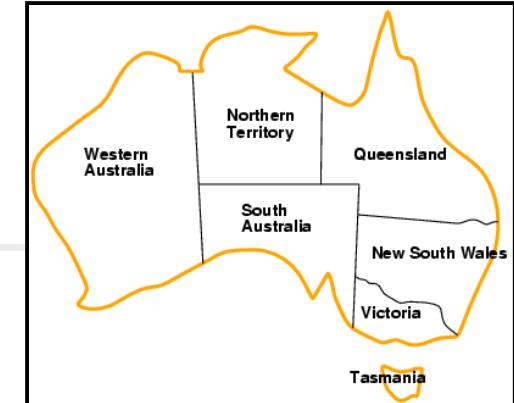
## Idea:

- Keep track of **remaining legal values** for **unassigned variables**
- Terminate search** when **any variable has no legal values**



WA	NT	Q	NSW	V	SA	T
█ Red	█ Green	█ Blue	█ Red	█ Green	█ Blue	█ Red
█ Red	█ Green	█ Blue	█ Red	█ Green	█ Blue	█ Red

# Forward checking



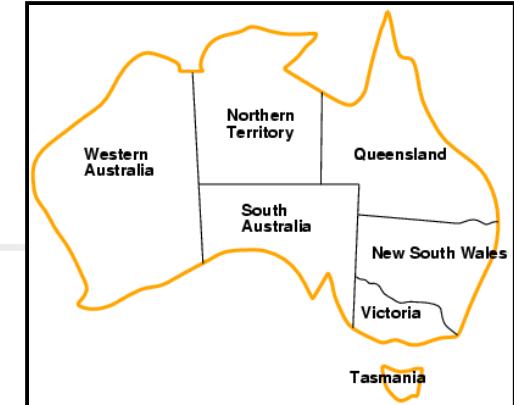
## Idea:

- Keep track of **remaining legal values** for **unassigned variables**
- Terminate search** when **any variable has no legal values**



WA	NT	Q	NSW	V	SA	T
■ Red ■ Green ■ Blue						
■ Red		■ Green ■ Blue	■ Red ■ Green ■ Blue	■ Red ■ Green ■ Blue	■ Green ■ Blue	■ Red ■ Green ■ Blue
■ Red			■ Red	■ Red ■ Green ■ Blue		■ Red ■ Green ■ Blue

# Forward checking



## Idea:

- Keep track of **remaining legal values** for **unassigned variables**
- Terminate search** when **any variable has no legal values**

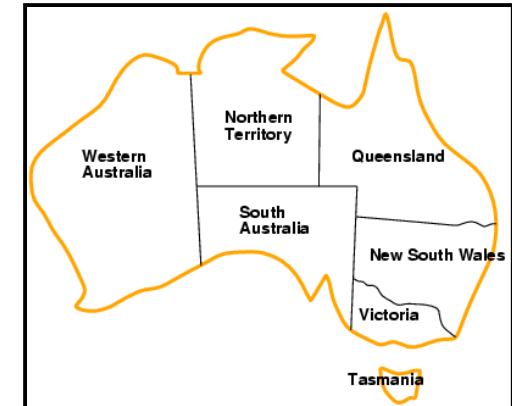
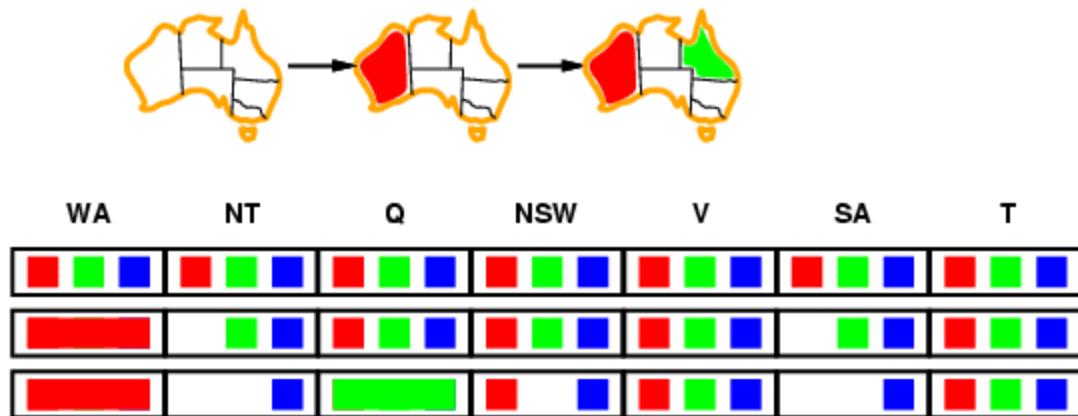


WA	NT	Q	NSW	V	SA	T
Red, Green, Blue						
Red	Green, Blue	Red, Green, Blue	Red, Green, Blue	Red, Green, Blue	Green, Blue	Red, Green, Blue
Red	Blue	Green	Red	Red, Green, Blue	Blue	Red, Green, Blue
Red	Blue	Green	Red	Blue		Red, Green, Blue

Search terminated with “no legal values” for “SA”

# Constraint propagation

- **Forward checking** propagates information from assigned to unassigned variables, but doesn't provide **early detection for all failures**:



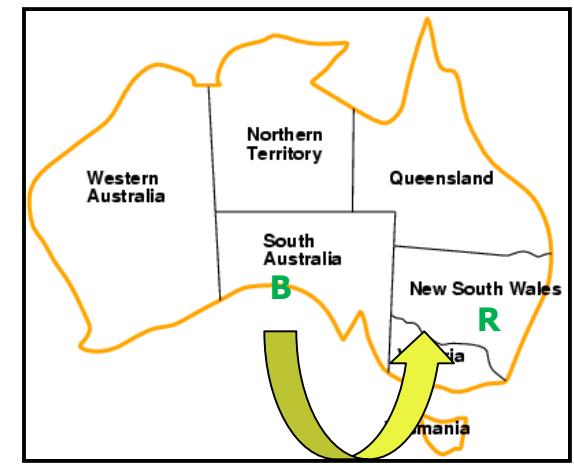
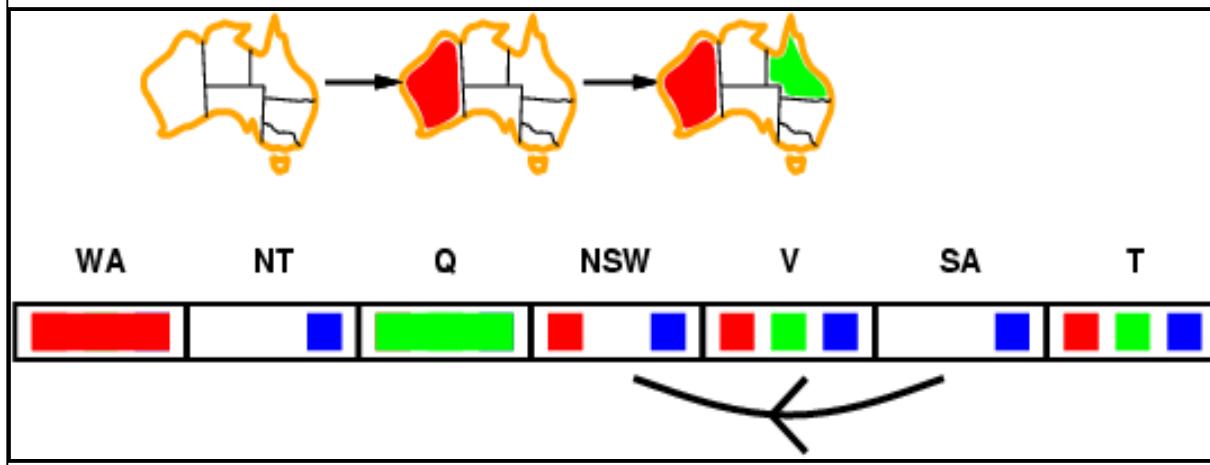
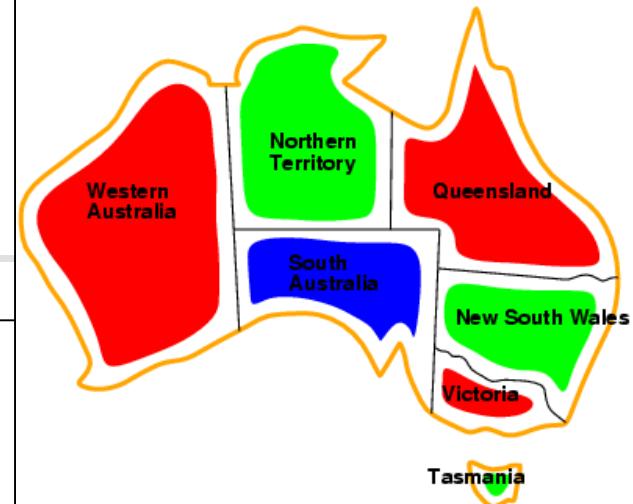
- **NT** and **SA** cannot both be blue!
- **Constraint propagation** repeatedly enforces constraints locally

# Arc consistency

- Simplest form of propagation makes each arc **consistent** with another

$X \rightarrow Y$  is consistent

Iff for every value  $x$  of  $X$  there is some allowed  $y$  of  $Y$

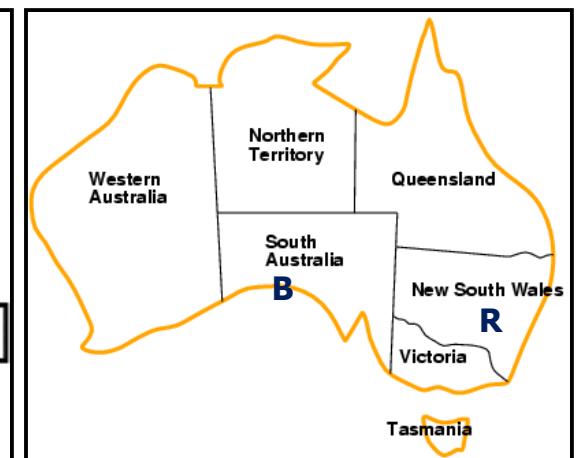
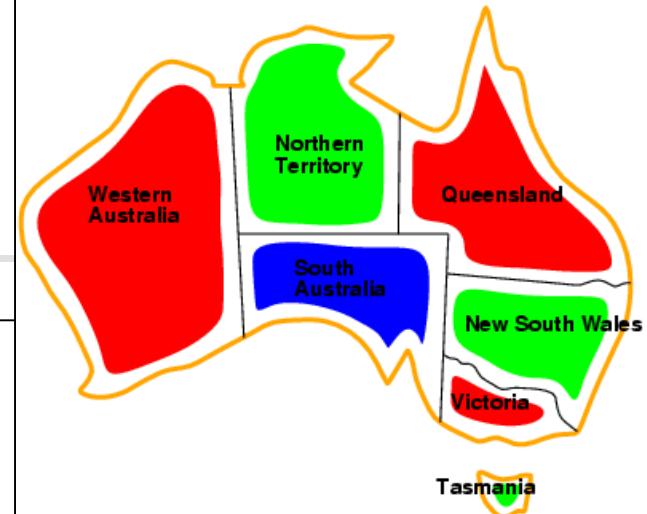
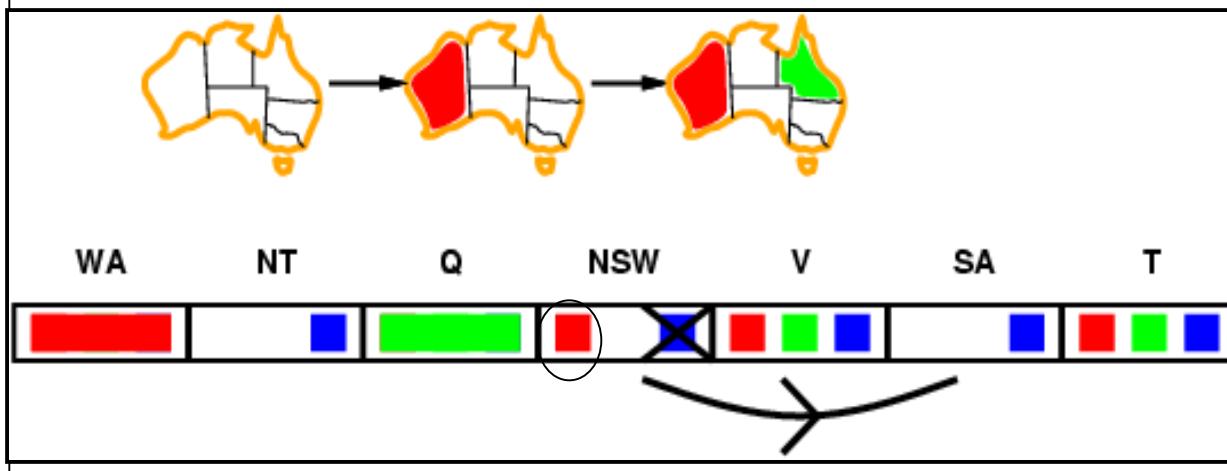


# Arc consistency

- Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$  is consistent

Iff for every value  $x$  of  $X$  there is some allowed  $y$  of  $Y$

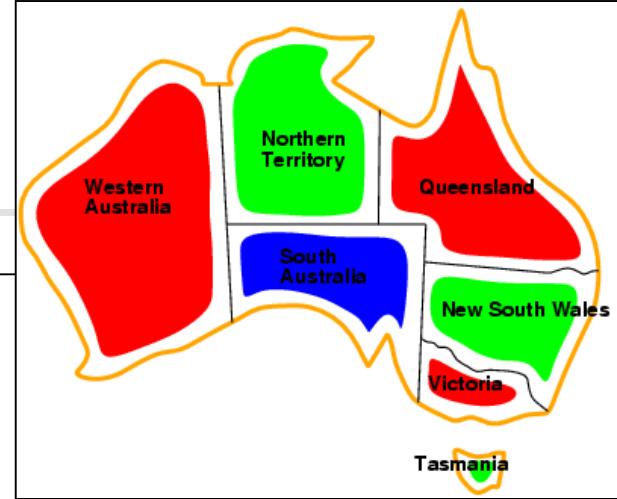
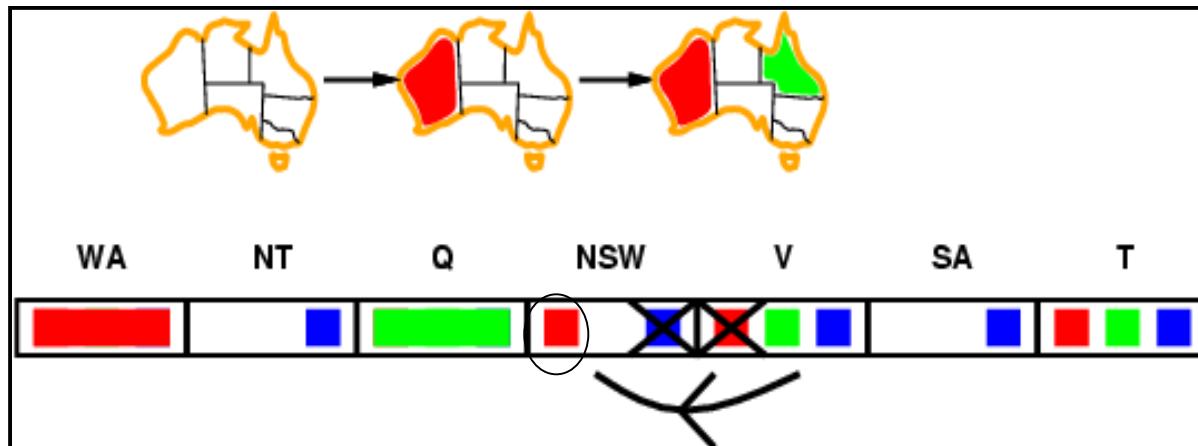


# Arc consistency

- Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$  is consistent

Iff for every value  $x$  of  $X$  there is some allowed  $y$  of  $Y$



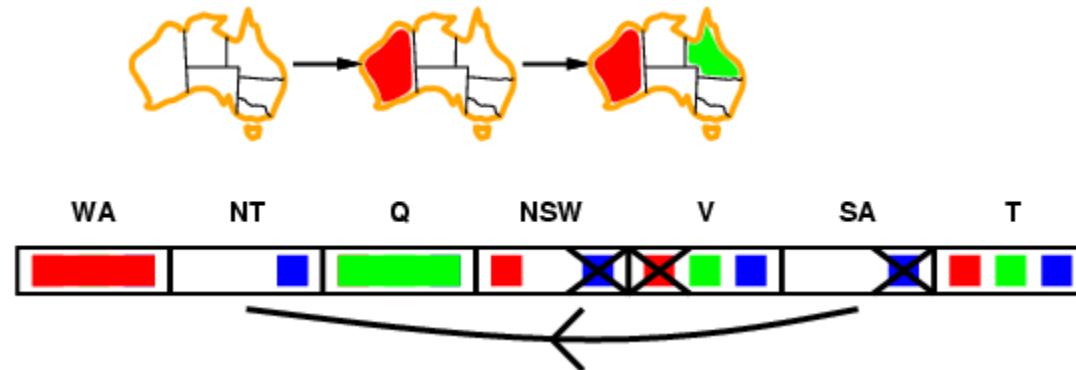
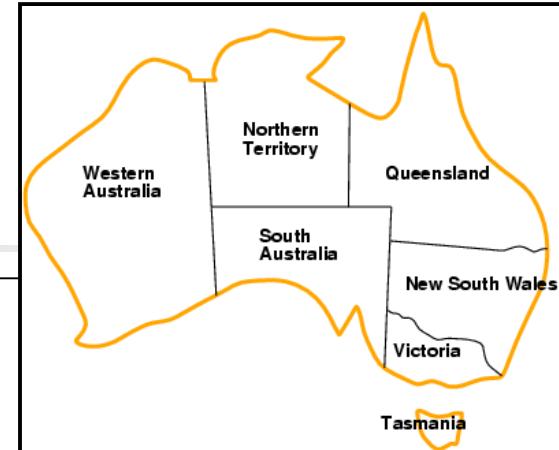
- If  $X$  loses a value, neighbors of  $X$  need to be rechecked

# Arc consistency

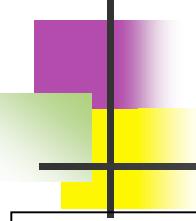
- Simplest form of propagation makes each arc **consistent**

$X \rightarrow Y$  is consistent

Iff for every value  $x$  of  $X$  there is some allowed  $y$  of  $Y$



- If  $X$  loses a value, neighbors of  $X$  need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

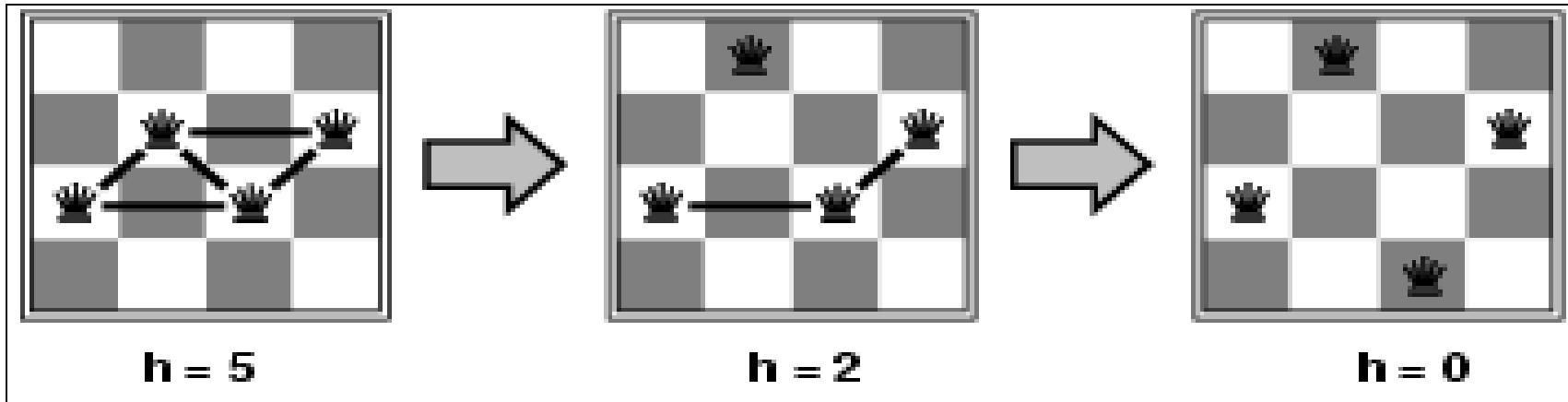


# Local search for CSPs

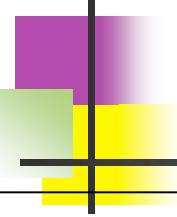
- Hill-climbing, simulated annealing typically work with "**complete**" states, i.e., all variables assigned
- **To apply to CSPs:**
  - Allow **states** with **unsatisfied constraints**
  - 
  - Operators **reassign** variable values
- **Variable selection:** randomly select any conflicted variable
- **Value selection** by **min-conflicts** heuristic:
  - **Choose value that violates the fewest constraints**
  - i.e., Hill-climb with  $h(n)$  = total number of violated constraints

# Example: 4-Queens

- **States:** 4 queens in 4 columns ( $4^4 = 256$  states)
- **Actions:** move queen in column
- **Goal test:** no attacks
- **Evaluation:**  $h(n)$  = number of attacks

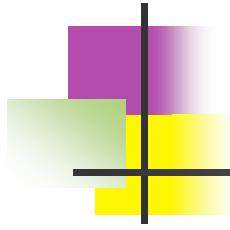


- Given random initial state, can solve  $n$ -queens in almost constant time for arbitrary  $n$  with high probability (e.g.,  $n = 10,000,000$ )



# Summary

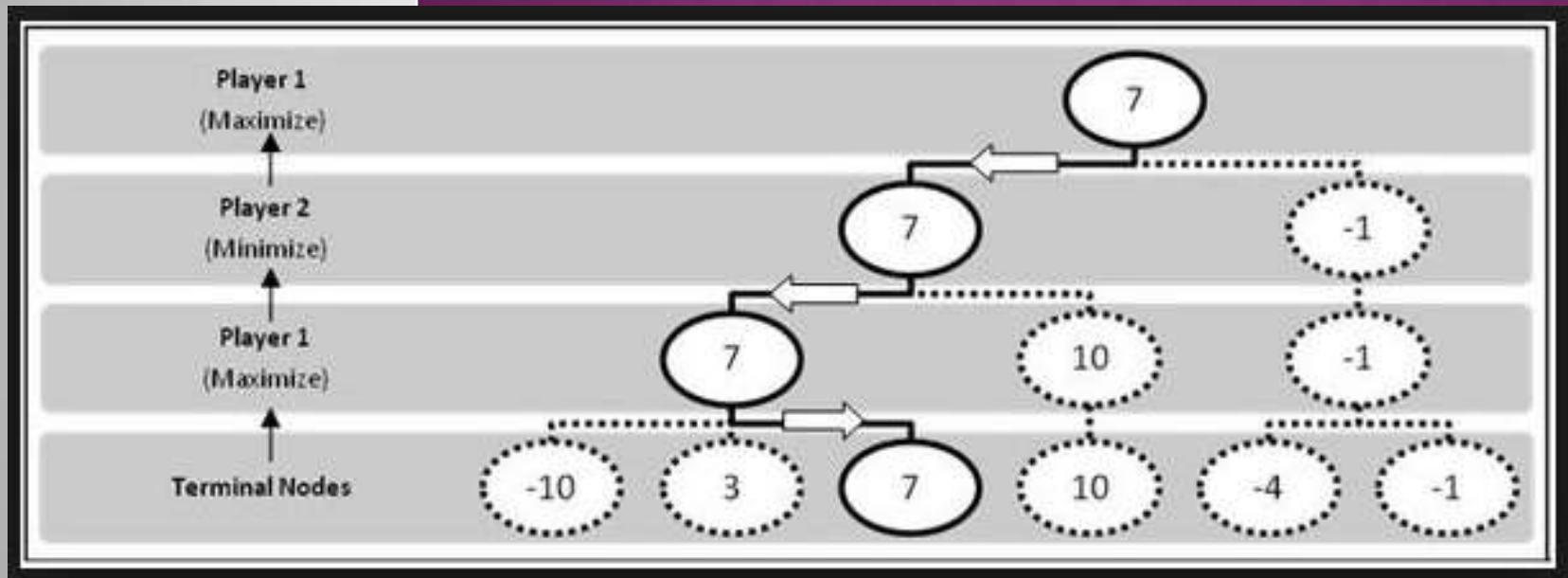
- CSPs are a special kind of problem:
  - states defined by values of a fixed set of variables
  - goal test defined by constraints on variable values
- Backtracking = depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- Iterative min-conflicts is usually effective in practice



---

The End

# ADVERSARIAL SEARCH / GAME PLAYING



*Dr. Pulak Sahoo*

Associate Professor

Silicon Institute of Technology

# CONTENTS

## ⦿ Adversarial Search

- Introduction
- Game Playing
- Min-Max Search
- Alpha Beta Pruning

# INTRODUCTION

- **Adversarial Search problems (Game Playing)**
  - Multi-Agent env - Need to consider action of other agents
  - Competitive env - The agent's goals are in conflict (adversarial)
- **Ex: Chess or Tic-Tac-Toe**
  - Deterministic, turn-taking, two-player, zero-sum games
  - **Zero sum game** - Total payoff to all players is same ( $0+1$ ,  $1+0$ ,  $\frac{1}{2} + \frac{1}{2}$ )
  - One agents action has equal & opposite effect on other  
(One wins, other loses)



# INTRODUCTION

## Games vs. Search problems

- More than one agent
- Harder to solve (search tree of chess has  $35^{100}$  nodes)
- Deal with "**Unpredictable**" opponent → Determine a **counter-move** for every possible opponent move
- **Time limits** → If can not find the exact goal within time, **must approximate**
- Ex: Chess & Tic-Tac-Toe

# GAME PLAYING - DEFINITION

## ○ Formally defining a game

- ***S<sub>0</sub>* : Initial State** - How the game is setup at the start
- **PLAYER (s)**- Defines which player has the move in a state
- **ACTION (s)**- Returns the set of legal moves in a state
- **RESULT (s,a)**- The transition model that define result of a move
- **TERMINAL\_TEST (s)** - True if game is over & false otherwise
- **UTILITY (s,p)** - (objective / payoff function) - Final numeric value of a game that ends in terminal state ‘s’ for player ‘p’
  - Ex: In Chess, the outcomes are win(1), loss(0) or draw(1/2)

# GAME PLAYING

## ○ GAME TREE

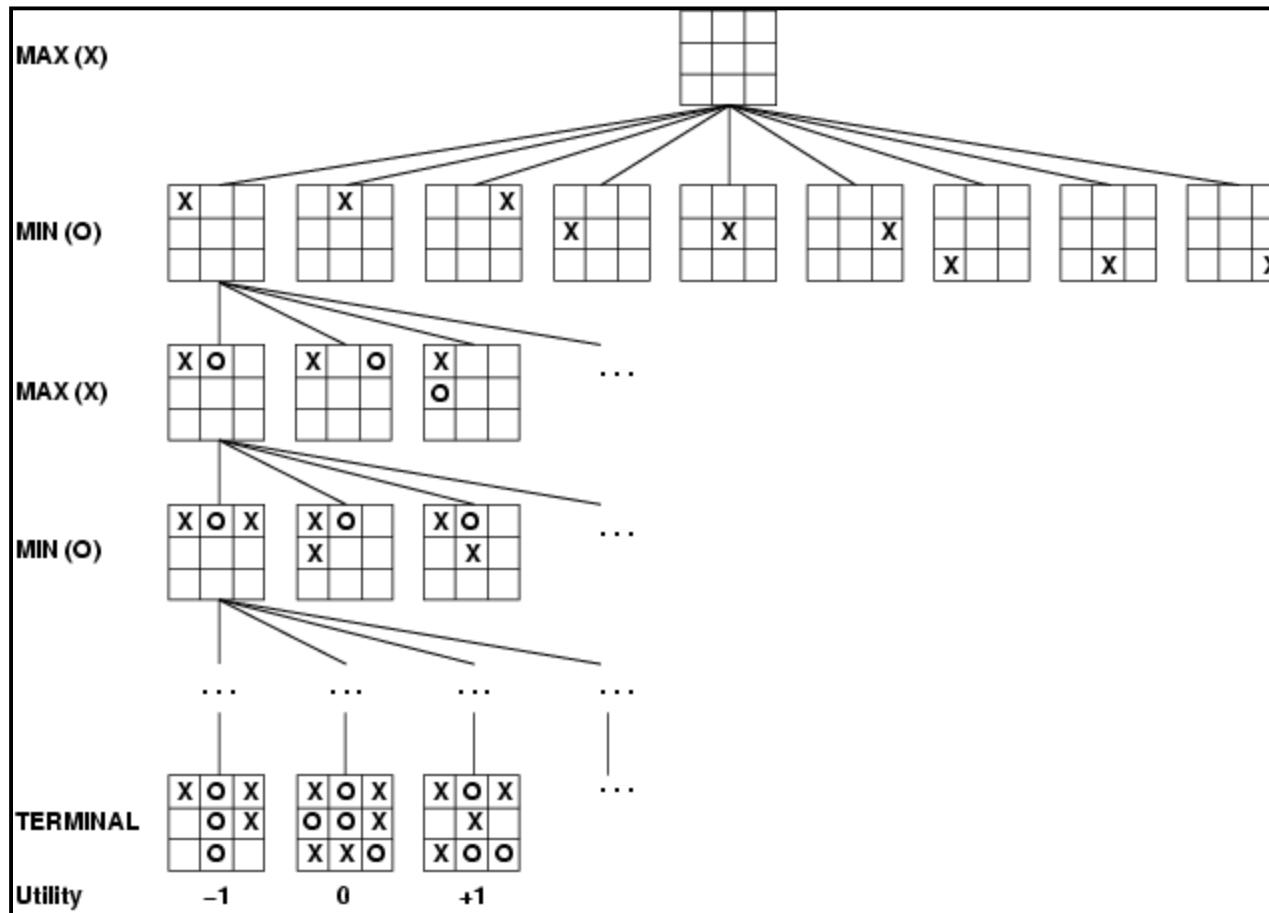
- Tree with nodes as game states & edges are moves (could be very large)
- Combination of **Initial state**, **ACTION** & **RESULT** functions
- See diagram on next page

# GAME TREE FOR TIC-TAC-TOE GAME (2-PLAYER, DETERMINISTIC, TURNS)

**MAX** places 'X' & **MIN** places 'O' alternatively

Terminal state - One player has 3 'X' or 3 'O' in a row

Tic-Tac-Toe has  $< 9!$  (362880) terminal nodes



MAX has 9  
possible moves

MIN has 8  
possible moves

•  
•  
•

# MINIMAX

## Optimal Decisions in games

- MAX & MIN are playing, MAX responds to all possible moves from MIN
- Optimal Strategy - Respond to “*best possible move*” of opponent
- Idea: choose the move with highest minimax value (given by utility/ payoff function) = best achievable move against best play by opponent
- Ex: 2-Ply Game Tree (See diagram on next page)

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if } \text{TERMINAL-TEST}(s) \\ & \text{Ex: In Chess, the Utility is win(1), loss(0) or draw(1/2)} \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

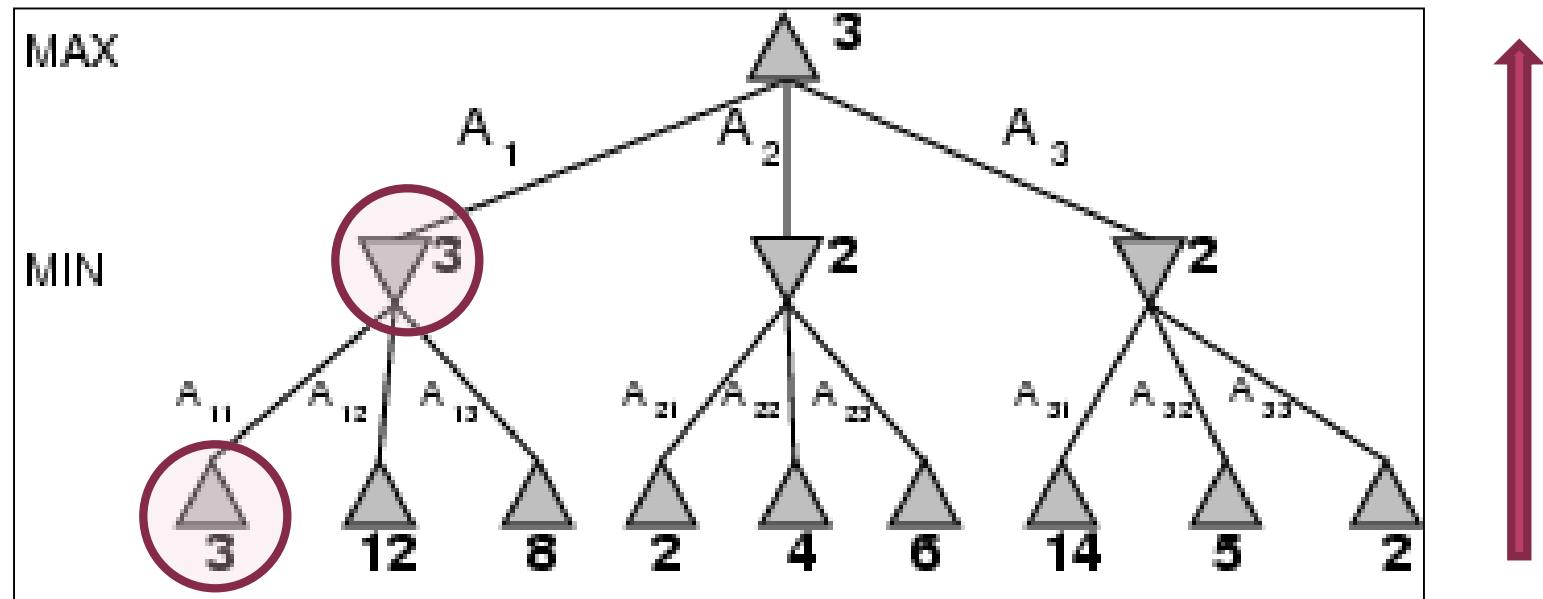
# EX: 2-PLY GAME TREE

Three possible moves by MAX (at root) - A<sub>1</sub>, A<sub>2</sub>, A<sub>3</sub>

Possible replies by MIN to A<sub>1</sub> - A<sub>11</sub>, A<sub>12</sub>, A<sub>13</sub> and so on..

This one-move game ends after  $\frac{1}{2}$  moves (called Ply) each by MAX & MIN

Optimal move is determined by Utility (Minimax value) of each node (A<sub>1</sub> - 3, A<sub>2</sub> - 2, A<sub>3</sub> - 2) - MAX chooses maximum value i.e. 3, MIN chooses minimum value i.e. 3

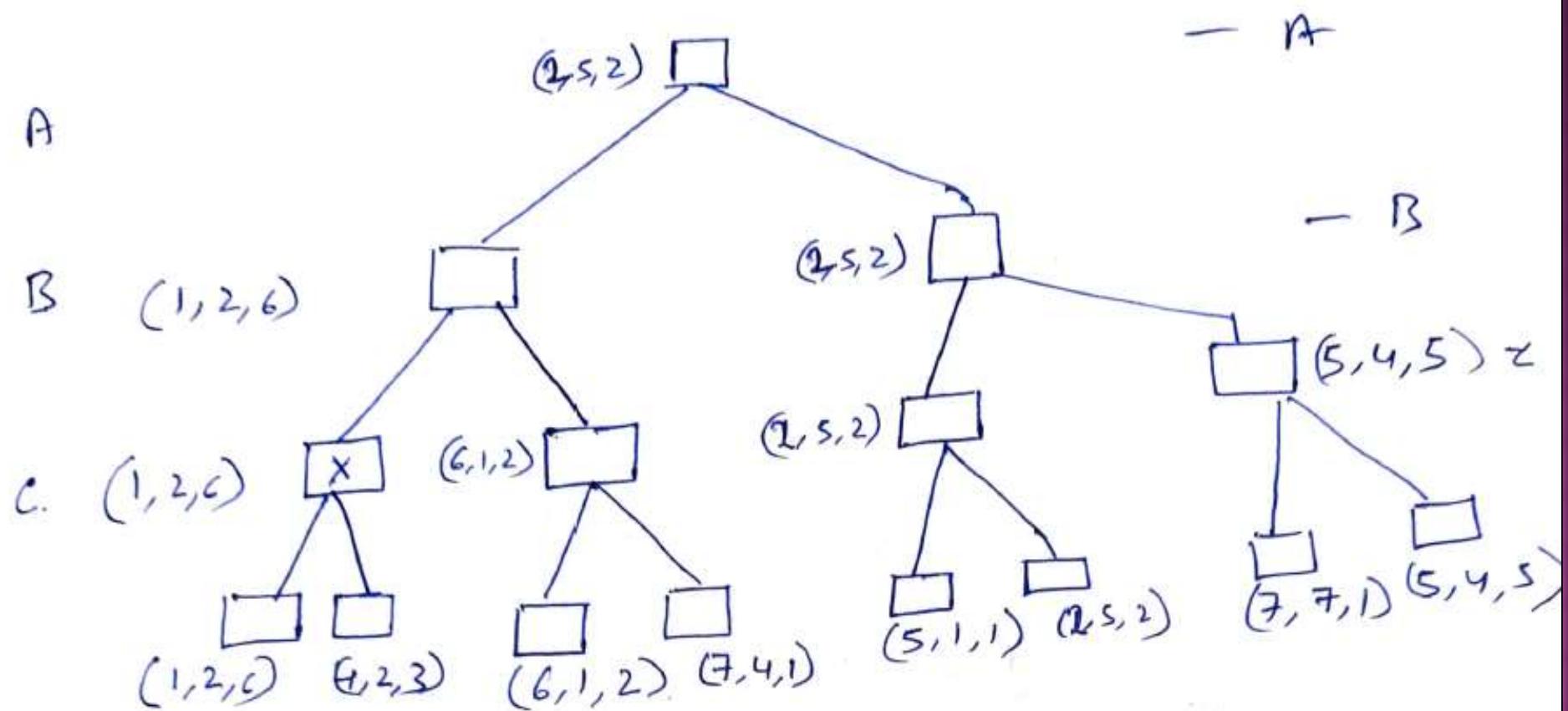


# EX: 3-PLY GAME TREE

3 Players are there - A, B, C with utilities ( $u_1, u_2, u_3$ ) - Not 0 sum game

( $u_1$  - utility of A,  $u_2$  - utility of B &  $u_3$  - utility of C)

We move upwards by selecting “Max” utilities at each level



# MINIMAX ALGORITHM

- Computes the **minimax** decision from the **current state**
- Uses **Recursive Computation** of minimax values of each successor state
- Recursion proceeds **down the leaves** of the tree & minimax values are **backed up** (*in the example it returns value 3*)
- Performs a **complete DFS** of the game tree
  - Time Complexity -  $O(b^m)$
  - Space Complexly -  $O(bm)$  or  $O(m)$  (*all actions generated at once or one at a time*)
  - Note: m - max depth of the tree, b - no. of legal moves at each point
  - For chess, b ≈ 35, m ≈ 100 for "reasonable" games → exact solution completely infeasible

# MINIMAX ALGORITHM

// Action – a, State – s, v - utility

**function** MINIMAX-DECISION(*state*) **returns** an action

*v*  $\leftarrow$  MAX-VALUE(*state*)

// *v* = max utility value of the state

**return** the *action* in SUCCESSORS(*state*) with value *v*

// Returns action corresponding to the best utility value

**function** MAX-VALUE(*state*) **returns** a utility value

**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*) // if terminal node return it's utility

*v*  $\leftarrow$   $-\infty$

// *v* initialized to -infinity

**for** *a, s* in SUCCESSORS(*state*) **do**

// Visit successor level (apply action on states) & find the max utility value (*v*) & return it

*v*  $\leftarrow$  MAX(*v*, MIN-VALUE(*s*))

**return** *v*

**function** MIN-VALUE(*state*) **returns** a utility value

**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

// if terminal node return it's utility

*v*  $\leftarrow$   $\infty$

// *v* initialized to infinity

**for** *a, s* in SUCCESSORS(*state*) **do**

// Visit successor level (apply action on states) & find the min utility value (*v*) & return it

*v*  $\leftarrow$  MIN(*v*, MAX-VALUE(*s*))

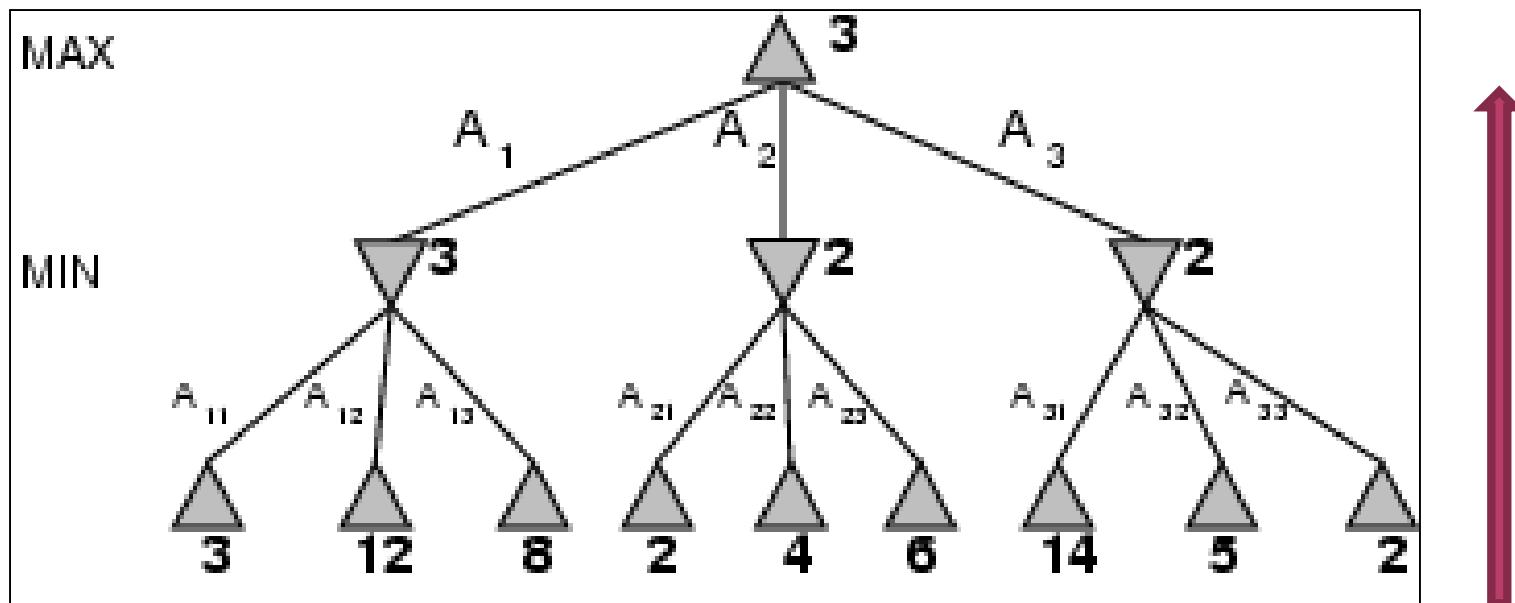
**return** *v*

# PROPERTIES OF MINIMAX

- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)

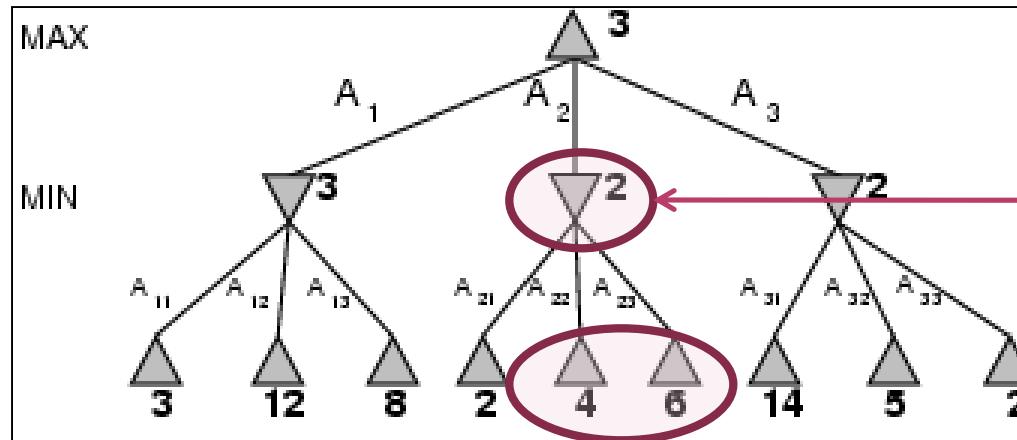
# ALPHA-BETA PRUNING

- Problem with **minimax** - no. of game states to be explored is exponential in the depth of the game tree
- This exponent can be cut down (in half) by **pruning** to eliminate large parts of the tree (called **Alpha-Beta Pruning**)
- Consider the same 2-ply game



# ALPHA-BETA PRUNING

- We can identify the **minimax** decision without evaluating 2 leaf nodes



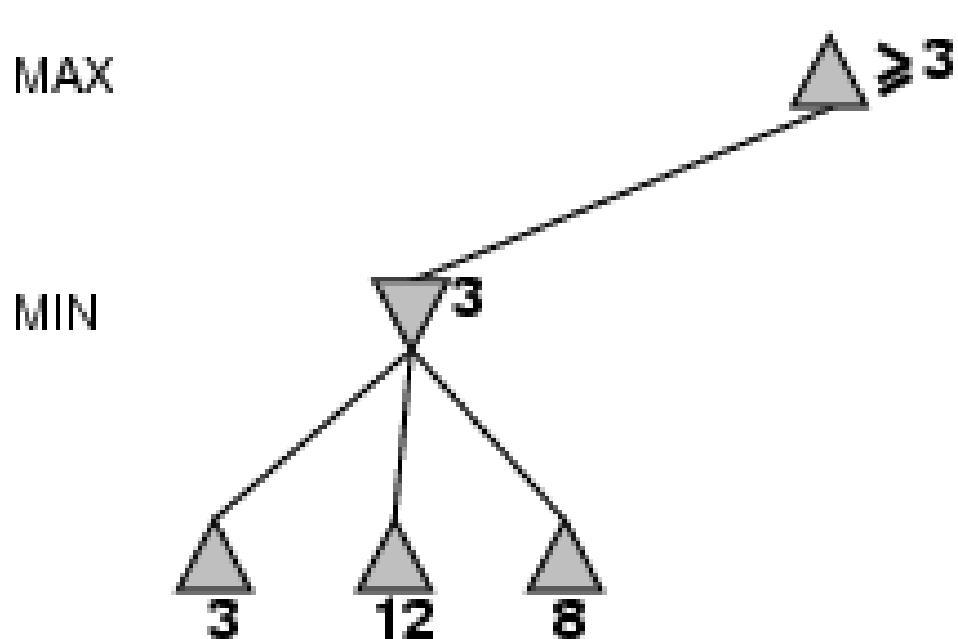
- Simplified** minimax formula is given below
- Consider the 2 unevaluated successors of node 'A2' have values 'x' & 'y'
- The **value** of the **root node** is given below (root value does not depend on pruned nodes x & y)

$$\begin{aligned}\text{MINIMAX (root)} &= \max (\min (3, 12, 8), \min (2, x, y), \min (14, 5, 2)) \\ &= \max (3, \min (2, x, y), 2) \\ &= \max (3, z, 2) \text{ where } z = \min (2, x, y) \leq 2 \\ &= 3\end{aligned}$$

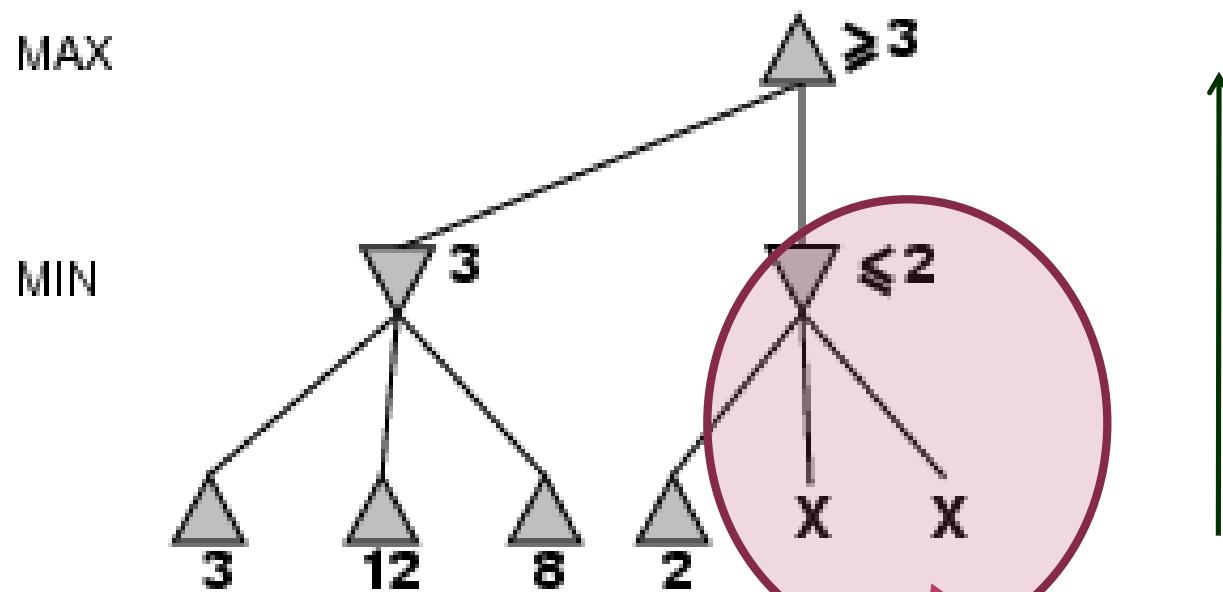
# ALPHA-BETA PRUNING

- Alpha-Beta Pruning can be applied to tree of any depth
- Often it is possible to prune entire sub tree than just leaves
- **Alpha** - *the value of the best (highest value) choice we have found so far at any choice point along the path for MAX*
- **Beta** - *the value of the best (lowest value) choice we have found so far at any choice point along the path for MIN*
- Alpha-Beta search updates values of **alpha** & **beta** as it goes along
- It prunes the remaining branches at a node
  - When value of current node is worse than alpha (for MAX) & beta (for MIN)

# ALPHA-BETA PRUNING EXAMPLE

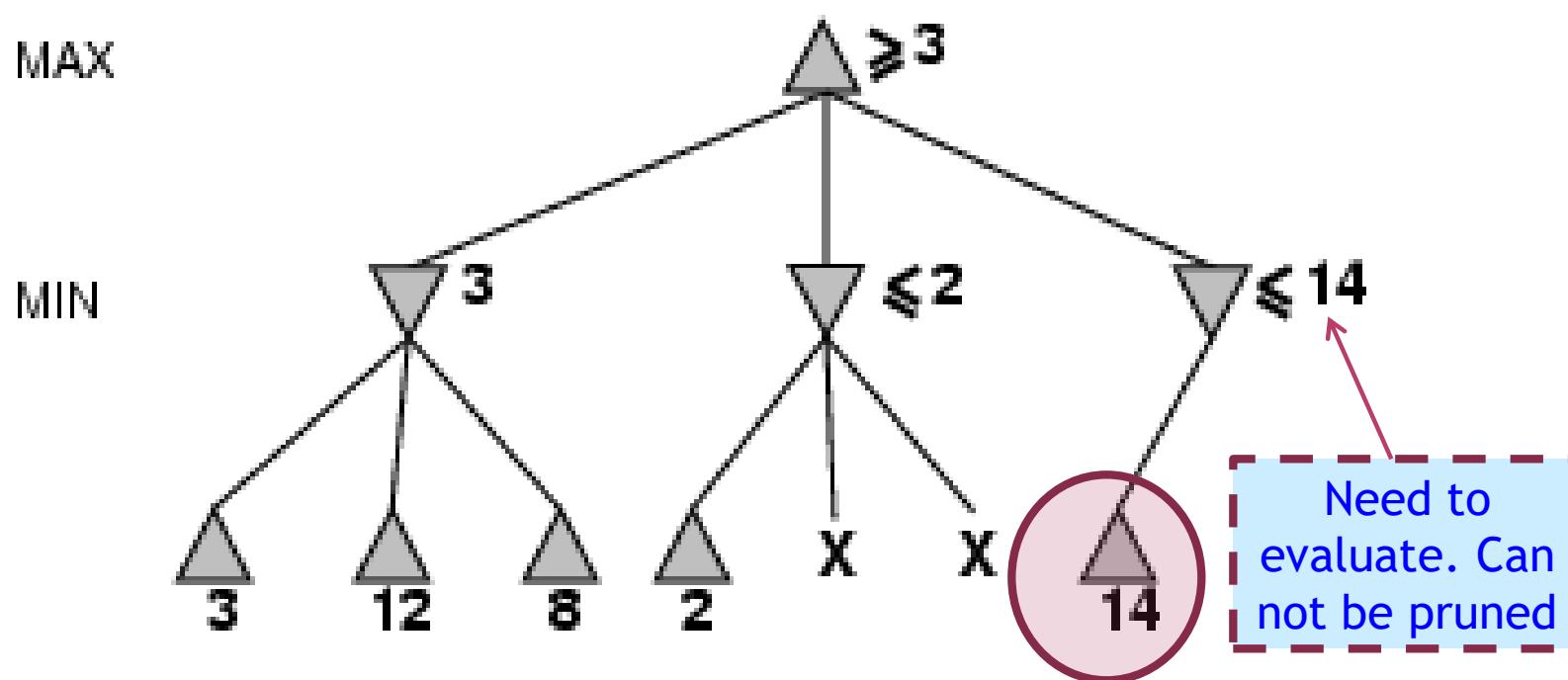


# ALPHA-BETA PRUNING EXAMPLE

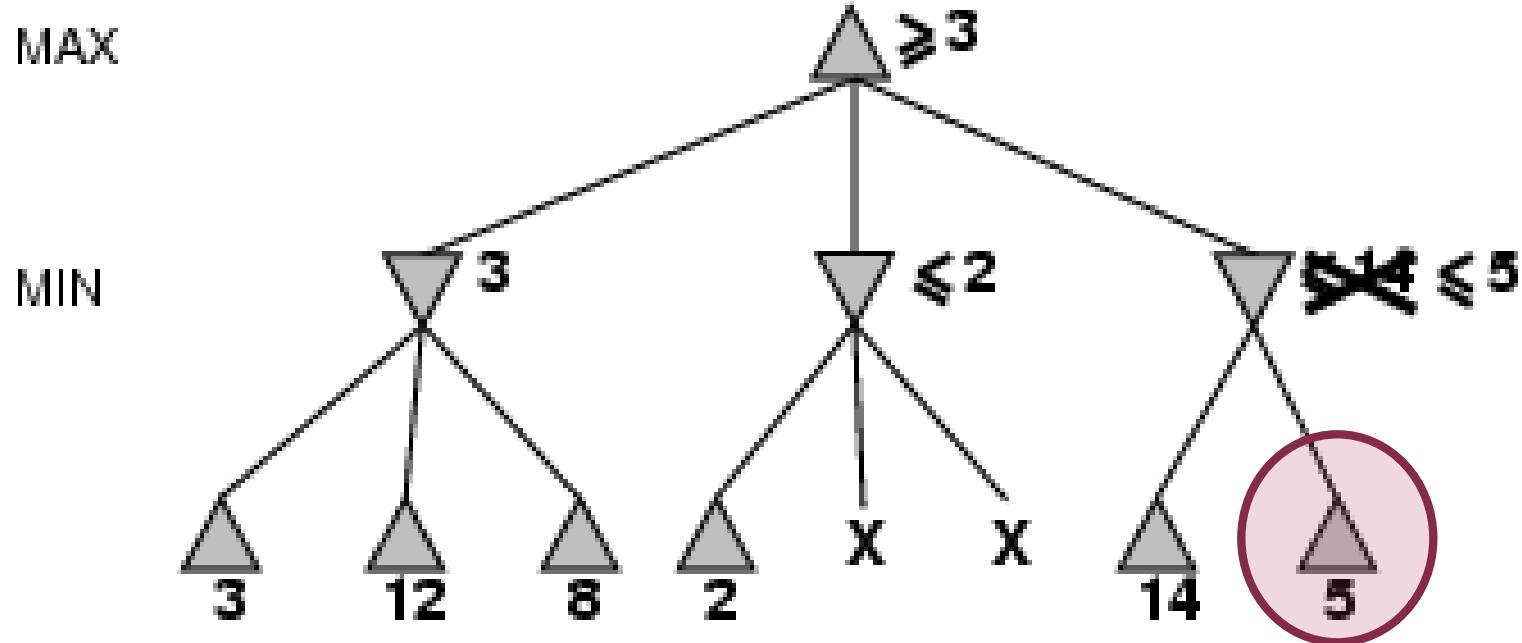


No need to  
evaluate. Can  
be pruned

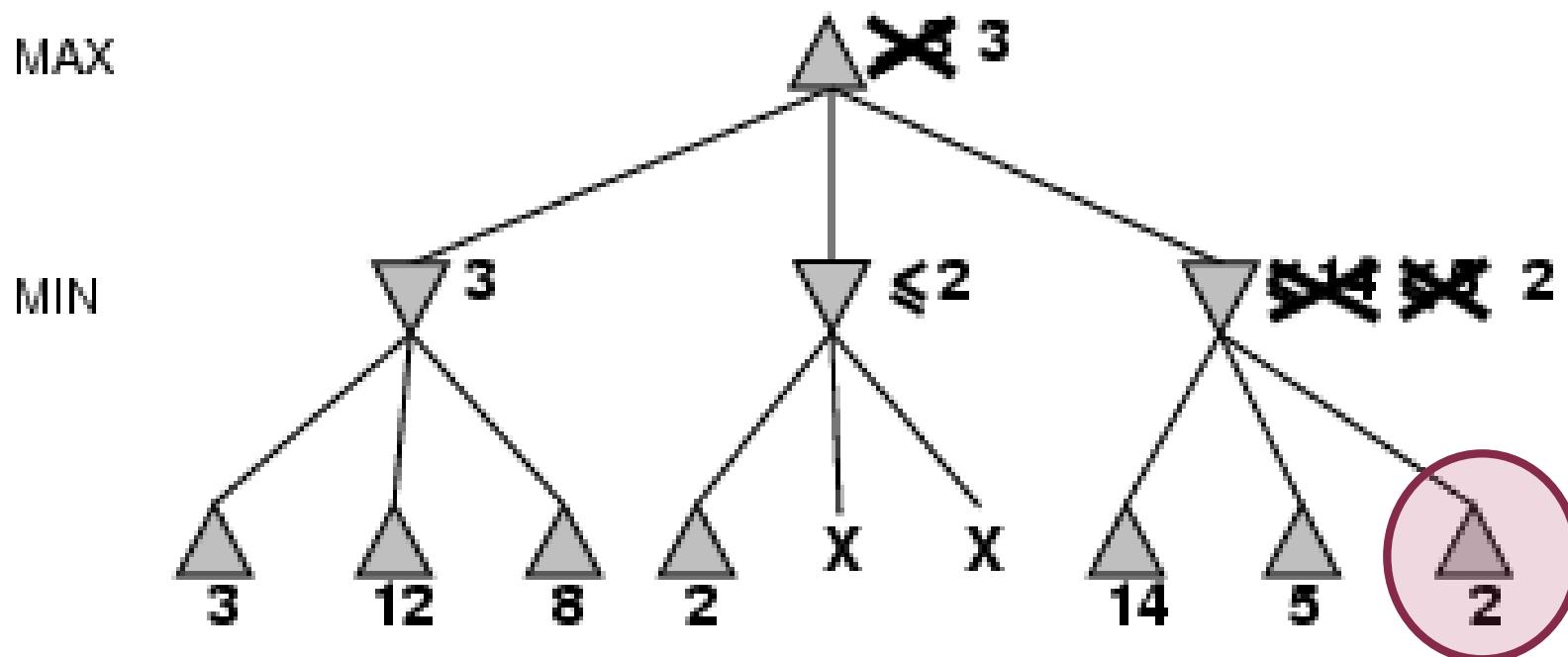
# ALPHA-BETA PRUNING EXAMPLE



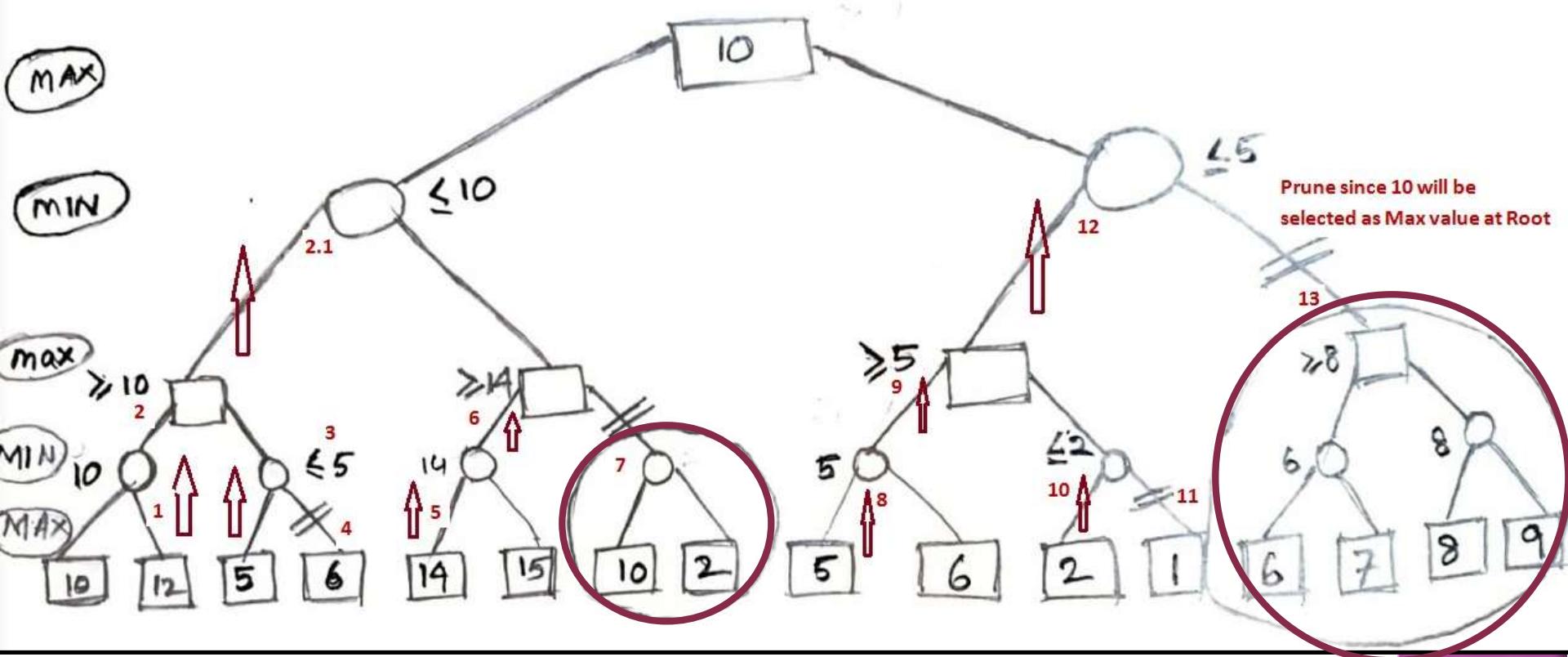
# ALPHA-BETA PRUNING EXAMPLE



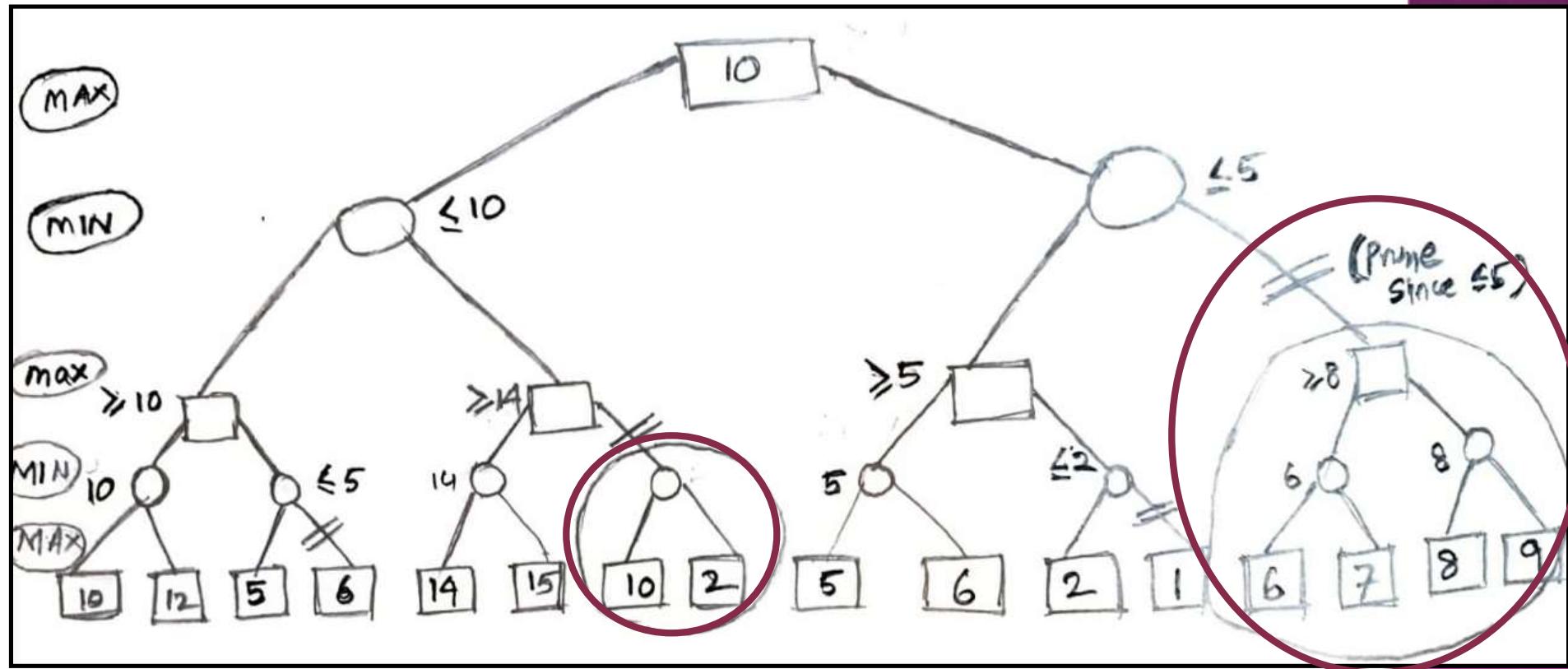
# ALPHA-BETA PRUNING EXAMPLE



# ALPHA-BETA PRUNING EXAMPLE-2



# ALPHA-BETA PRUNING EXAMPLE-2



# PROPERTIES OF ALPHA-BETA

- Pruning **does not** affect final result
- Good move ordering improves effectiveness of pruning
- With "perfect ordering" **time complexity =  $O(b^{m/2})$**

# THE ALPHA-BETA PRUNING ALGORITHM

**function** ALPHA-BETA-SEARCH(*state*) **returns** *an action*

**inputs:** *state*, current state in game

*v*  $\leftarrow$  MAX-VALUE(*state*,  $-\infty$ ,  $+\infty$ )

**return** the *action* in SUCCESSORS(*state*) with value *v*

---

**function** MAX-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** *a utility value*

**inputs:** *state*, current state in game

$\alpha$ , the value of the best alternative for MAX along the path to *state*

$\beta$ , the value of the best alternative for MIN along the path to *state*

**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

*v*  $\leftarrow -\infty$

**for** *a, s* in SUCCESSORS(*state*) **do**

*v*  $\leftarrow$  MAX(*v*, MIN-VALUE(*s*,  $\alpha$ ,  $\beta$ ))

**if** *v*  $\geq \beta$  **then return** *v*

$\alpha \leftarrow \text{MAX}(\alpha, v)$

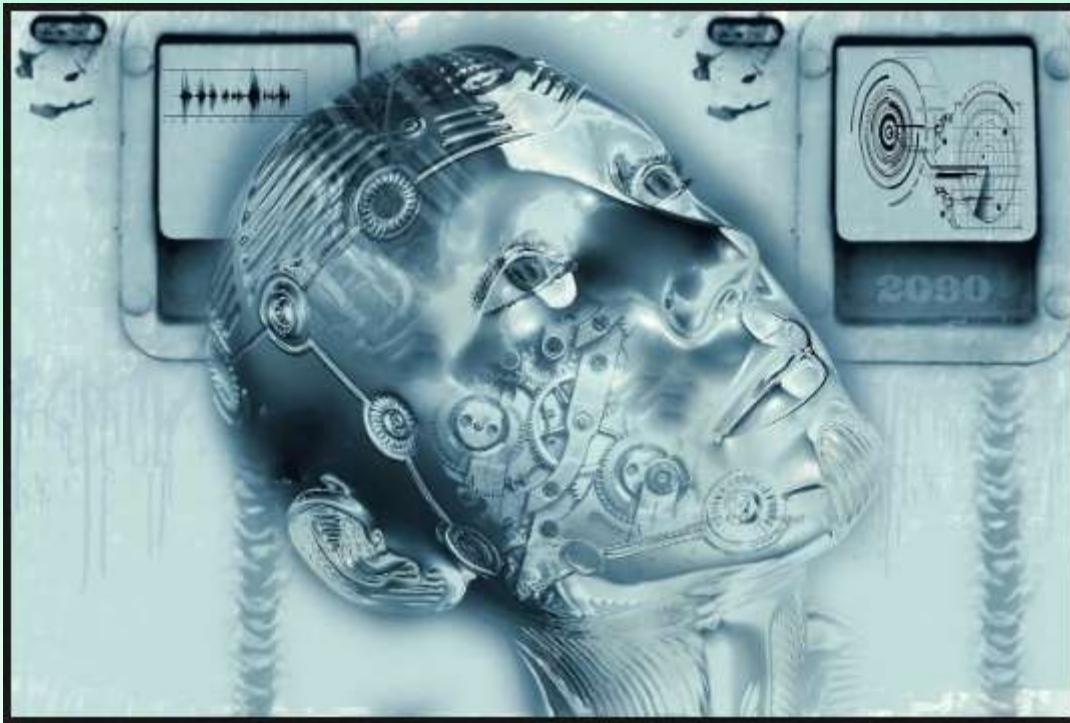
**return** *v*

# DETERMINISTIC AI GAMES IN PRACTICE

- **Checkers:** Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a pre-computed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions
- **Chess:** Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply
- **Othello:** human champions refuse to compete against computers, who are too good



# Knowledge-based Agents Logic



***Dr. Pulak Sahoo***

Associate Professor

Silicon Institute of Technology



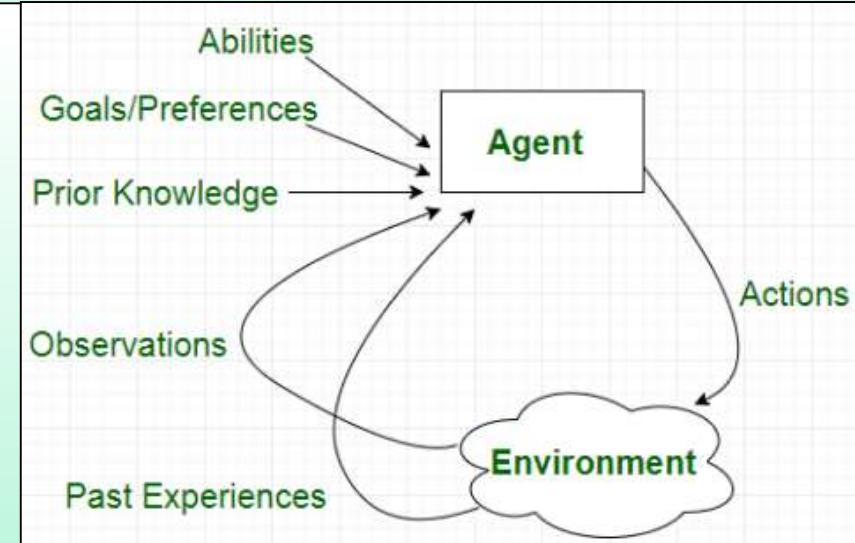
# **Knowledge-Based Agents & Logic**

## **- Topics**

- **Introduction**
- **Knowledge-Based Agents**
- **WUMPUS WORLD Environment**
- **Propositional Logic**
- **First Order Predicate Logic**
- **Forward and Backward Chaining**

# Introduction

- Human beings **know things**
- This helps them to **do things intelligently** based on **reasoning**
- Process of **reasoning** operates based on **internal representation (storage) of knowledge**
- Same approach is followed by **Knowledge-based Agents**
- **Logic** is a class of representation that supports **Knowledge-based Agents**
- They can **adopt to changes in env.** by updating knowledge



# Knowledge-based Agents (Design)

- **Central component** – knowledge base (KB)
- **Knowledge Base** – Set of sentences expressed in **Knowledge Representation Language**
- **Operations**
  - **TELL** – Add new sentence to KB
  - **ASK** – Query what is known
- An **KB Agent program** takes a percept as **input** & **returns an action**
- The **KB** initially contains some “**background knowledge**”
- The **Agent program** does 3 things
  - **TELLs** the **KB** what it **perceives**
  - **ASKs** the **KB** what **action should be performed**
  - **TELLs** the **KB** what **action was chosen & executes the action**

# KB Agents Program

Agent **KB-Agent (Percept)** Returns an **action**

```
Persistent: KB – a knowledge base // Maintain a KB
t (time) = 0 //time is initialized to 0

// Input percept sequence & time to KB
TELL ( KB, Make-Percept-Sentence ( percept, t ) )

// Find suitable action to be taken from KB
action = ASK (KB, Make-Action-Query ( t ) )

// Update KB with action corresponding to the percept seq at time t
TELL (KB, Make-Action-Sentence ( percept, t )

t = t + 1 // Increment time

return action // Return action
```

# KB Agents Program

- Two System building approaches employed by a designer to an empty KB

## 1. Declarative approach

- TELL sentences one-by-one until the agent knows how to operate

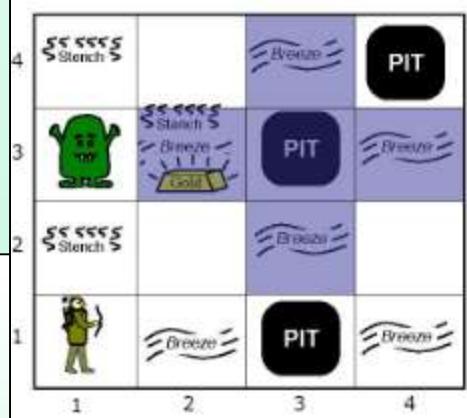
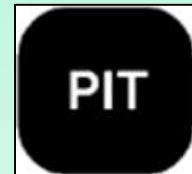
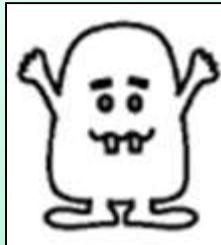
## 2. Procedural approach

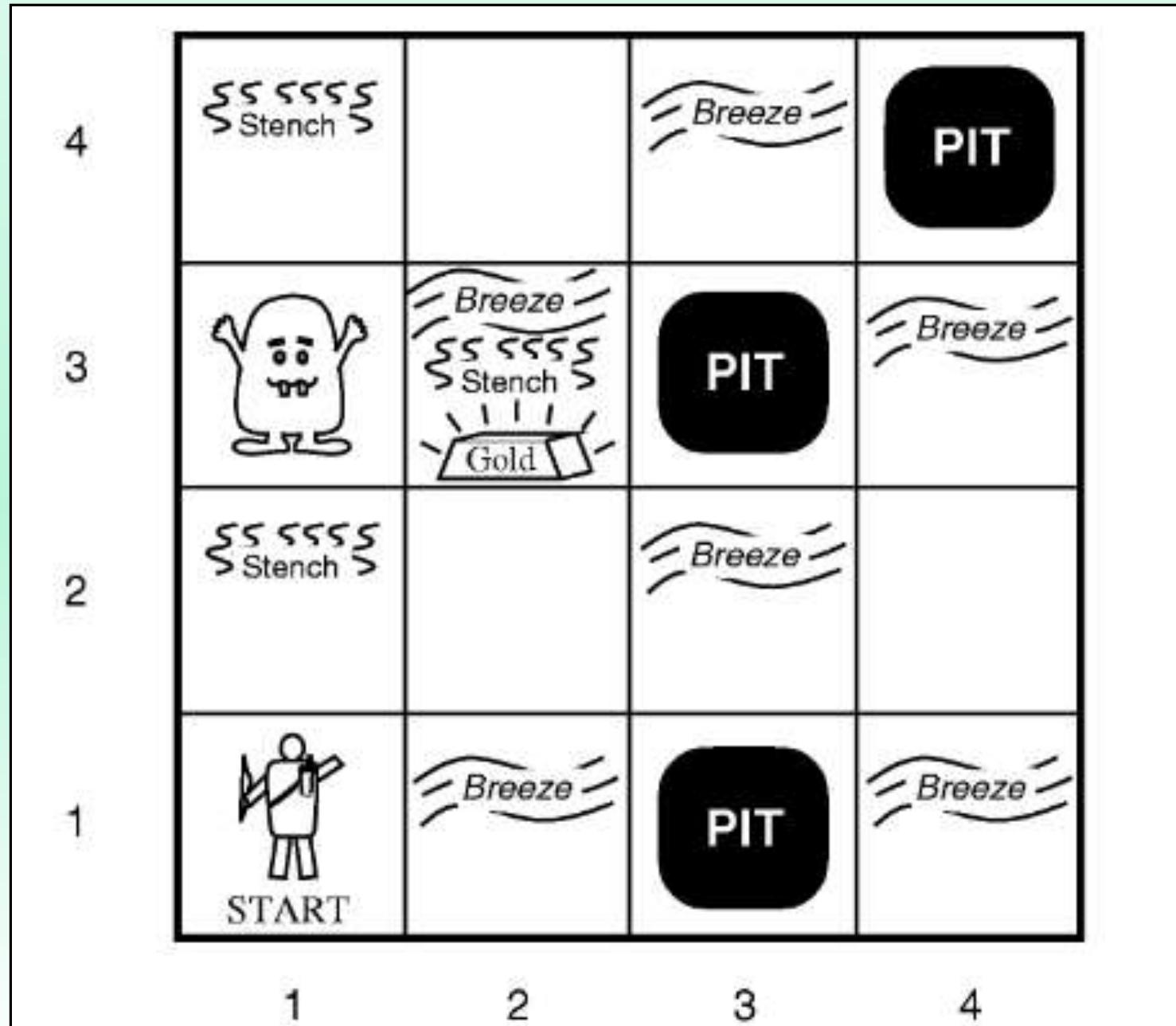
- Encodes desired behavior directly into program code
- A successful agent must combine both approaches

# The Wumpus World Environment

## ■ Wumpus World

- A **cave** containing **rooms** connected by **passageways**
- The **Wumpus (beast)** hidden in the cave – **Eats anyone** entering the room
- The **Agent** has only one arrow to shoot
- Some rooms have bottom-less **pits** to trap anyone entering
- **Only Reward** - Possibility of finding a **gold heap**





# Task Environment Description - PEAS

- **Performance measure**

- **+1000** – Coming out of cave with gold
- **-1000** – Falling into Pit or Eaten by Wumpus
- **-1** – For each action
- **-10** – For using the arrow
- **End of game** – Agent dies or climbs out of cave

- **Env**

- A **4X4 grid** of rooms
- **Agent** starts in **[1,1]**
- Location of **Gold** & **Wumpus** chosen **randomly** (except starting one )
- Each square (except starting one) can be a **pit** with **probability 0.2**

# Task Environment Description - PEAS

- **Actuators**
  - **Agent Moves** – *Forward, TurnLeft, TurnRight*
  - **Death** – Falling into Pit or Eaten by Wumpus (Safe to enter room with dead wumpus)
  - **Forward move against wall** – Not allowed
  - **Actions** – **Grab** (pickup gold), **Shoot** (one Arrow), **Climb** (out of cave from [1,1])
  - **End of game** – Agent dies or climbs out of cave

- 
- **Sensors**
    - **Stench**: Perceived in squares **containing & adjacent** to **wumpus**
    - **Breeze**: Perceived in squares **adjacent** to a **pit**
    - **Glitter**: Perceived in squares containing **Gold**
    - **Bump**: Perceived when walking into a **Wall**
    - **Kill Wumpus**: Perceived **Scream** anywhere in the cave

# Wumpus World - Steps

- Challenges for Agent - Initial ignorance of env configuration (require logical reasoning)
- *Good possibility of agent getting out **with gold***
- *Sometimes, agent will have to **choose** between **empty-hand return** or **death***
- *21% times **gold** is in a **pit** or **surrounded by pits***
- Knowledge Representation Language (KRL) used – writing **symbols** in the grids
- Initial KB – contains **rules** of the game

- **Start** grid [1,1] & it is **safe** – denoted by **A** (agent) & **OK**
- 1<sup>st</sup> percept is **[None, None, None, None]** => neibouring grids **[1,2]** & **[2,1]** are safe (**OK**)
- If **Agent** moves to **[2,1]** => Perceives **breeze** (**B**)=>**Pit(s)** present in **[2,2]** or **[3,1]** or **both (P?)**
- Only safe square is **[1,2]**. Hence agent should move back to **[1,1]** & then to **[1,2]**

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
OK			
1,1 A	2,1	3,1	4,1
OK	OK		

(a)

**A** = Agent  
**B** = Breeze  
**G** = Glitter, Gold  
**OK** = Safe square  
**P** = Pit  
**S** = Stench  
**V** = Visited  
**W** = Wumpus

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
OK			
1,1 V OK	2,1 B OK	3,1 P? OK	4,1

(b)

- In [1,2] perceived **Stench** & **No breeze** – denoted by **S - [Stench, None, None, None, None]**
- After 5<sup>th</sup> move perceived **[Stench, Breeze, Glitter, None, None]** => **Found Gold**

1,4	2,4	3,4	4,4
1,3 W	2,3	3,3	4,3
1,2 A ok	2,2 P?	3,2	4,2
1,1 v ok	2,1 B	3,1 P? P?	4,1

(a)

Perceived  
stench ,  
No Breeze

**A** = Agent  
**B** = Agent  
**G** = Glitter, Gold  
**ok** = Safe,  
**P** = Pit  
**S** = Stench  
**V** = Visited  
**W** = Wumpus

1,4	2,4 P?	3,4	4,4
1,3 W?	2,3 A S-G B	3,3 P?	4,3
1,2 S V ok	2,2 V P?	3,2	4,2
1,1 v ok	2,1 B S V ok	3,1 P?	4,1

(b)

Found gold

# Logic & Deduction

- A formal system for describing **states of affairs**, consisting of:
  - The syntax of the language describing *how to make sentences*
  - The semantics of the language describing *the relation between the sentences & the states of affairs*
  - A proof theory – *a set of rules for logically deducing entailments of a set of sentences*
- Improper definition of **logic** or incorrect **proof theory** can result in absurd reasoning

# Types of Logics

Language	What exists	Belief of agent
Propositional Logic	Facts	True/False/Unknown
First-Order Logic	Facts, Objects, Relations	True/False/Unknown
Temporal Logic	Facts, Objects, Relations, Times	True/False/Unknown
Probability Theory	Facts	Degree of belief 0..1
Fuzzy Logic	Degree of truth	Degree of belief 0..1

# 1. Propositional Logic

- **Simple but Powerful**
- Contains a set of **atomic propositions  $\mathcal{AP}$**
- It contains **Syntax, Semantics & Entailment -**
  - **Syntax** - Defines allowable sentences
  - **Sentences** – 2 types
  - **Atomic sentence** – Single symbol that can be **True | False |  $\mathcal{AP}$**   
*Ex-  $P, Q, R, W_{1,3}$  (means Wumpus in [1,3]), North...*
  - **Complex sentence** - ( Sentence ) | [Sentence]
    - | : Logical Connective like
    - ( $\neg$  (negation),  $\wedge$  (and) ,  $\vee$ (or),  $\Leftrightarrow$  (if & only if),  $\Rightarrow$ (implies))  
*Ex:  $W_{1,3} \Leftrightarrow \neg W_{2,2}$*
  - **Semantics** – *Defines the rules for determining the truth of the statement in a given model*
  - **Entailment** – *Relation between 2 successive sentences*

# Inference Rules

- **Inference rule** - transformation rule - is a logical form that takes premises, analyzes their syntax, and returns a conclusion

## 1. Modus Ponens or Implication Elimination:

- Premise-1 : "If  $\alpha$  then  $\beta$ "    $\alpha \Rightarrow \beta$ ,
- Premise-2 :  $\alpha$ ,
- Conclusion :  $\beta$

$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$$

$\alpha \Rightarrow \beta$  Given  $\alpha$   
Conclusion  $\beta$

=> if the premises are true, then so is the conclusion.

# Inference Rules

## 2. Unit Resolution:

$$\frac{\alpha \vee \beta, \neg \beta}{\alpha}$$

- If  $\alpha \vee \beta$  is True &  $\neg \beta$  is True, Then  $\alpha$  is True

## 3. Resolution:

$$\frac{\alpha \vee \beta, \neg \beta \vee \gamma}{\alpha \vee \gamma}$$

or

$$\frac{\neg \alpha \Rightarrow \beta, \beta \Rightarrow \gamma}{\neg \alpha \Rightarrow \gamma}$$

- The 2 premises are said to be **resolved** and the variable  $\beta$  is said to be **resolved away**.

.... and several other rules

- A **sentence/premise** may have:
  - **Validity** (always true)
  - **Satisfiability** (sometimes true)
  - **No Satisfiability** (always false)

# Propositional Logic

- **Semantics** (*Defines the rules for determining the truth of the statement*)
- **Atomic sentence** – 2 rules (True & False)
- **Complex sentence** – 5 rules
  - $\neg P$  is true iff  $P$  is false
  - $P \wedge Q$  is true iff both  $P$  &  $Q$  are true
  - $P \vee Q$  is true iff either  $P$  or  $Q$  is true
  - $P \Rightarrow Q$  is true unless  $P$  is true &  $Q$  is false
  - $P \Leftrightarrow Q$  is true iff  $P$  &  $Q$  are both true or both false
- **Truth Tables** - Specify truth value of complex sentence for each possible value

**Ex:**  $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

A square is breezy if the neighboring squares have pit and vice versa

$P$	$Q$	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

# Propositional Logic – Example – Wumpus world

- A Simple **Knowledge Base**
  - Example KB for Wumpus world
    - $P_{x,y}$  – True, if **pit** is there in  $[x,y]$
    - $W_{x,y}$  - True, if **wumpus** is there in  $[x,y]$
    - $B_{x,y}$  - True, if **breeze** is there in  $[x,y]$
    - $S_{x,y}$  - True, if **stench** is there in  $[x,y]$
  - Sentences (Enumerate)
    - $R_1 : \neg P_{1,1}$  // There is **no pit** in  $[1,1]$
    - $R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$  // A square is breezy if the neighboring squares have pit & vice versa

# Propositional Logic

- A Simple **Inference Procedure**
    - Models are assignments of True or False to every symbol
    - Check the sentences are true in every model
    - Example Inference Procedure (Wumpus world)
      - Seven symbols –  $B_{1,1}, B_{2,2}, P_{1,1}, P_{1,2}, P_{2,1}, P_{2,2}, P_{3,1}$
      - $2^7 = 128$  possible models
      - In three cases, KB is true
    - Time Complexity =  $O(2^n)$
    - Space Complexity =  $O(n)$
- $n$  = no. of symbols in KB

# Examples - Automated Reasoning

**Example-1:** *Deducing the position of the wumpus based on information like Stench, Breeze etc..*

**Example-2:**

- If the **unicorn** is **mythical**, then it is **immortal**, (premises)
- But, if it is **not mythical**, then it is a **mortal mammal**.
- If the **unicorn** is either **immortal** ( $P_1$ ) or a **mammal** ( $P_2$ ) , then it is **horned** ( $Q$ ) .
- The **unicorn** is **magical** if it is **horned**

**Q:** Can we prove that the unicorn is **mythical?** **Magical?** **Horned?**

- In general, the inference problem is NP-complete (Cook's Theorem)
- If we restrict ourselves to Horn sentences, then repeated use of Modus Ponens gives us a polytime procedure. Horn sentences are of the form:

$$P_1 \wedge P_2 \wedge \dots \wedge P_n \Rightarrow Q$$

# Conjunctive Normal Form (CNF)

- **Conjunctive normal form (CNF)** is an approach to Boolean logic that expresses **formulas** as :
- **Conjunctions of clauses with an AND or OR**
- Each **clause** connected by a **conjunction**, (**AND**) must be either a **literal** or **contain a disjunction (OR)** operator.

$$\begin{array}{c} a \wedge b \\ (a \vee \neg b) \wedge (c \vee d) \\ \neg a \wedge (b \vee \neg c \vee d) \wedge (a \vee \neg d) \end{array}$$

- **CNF** is useful for **automated theorem proving**

# Conjunctive Normal Form (CNF)

## Conversion Procedure to Normal Form

**STEP I:** Eliminate implication and biconditionals. We use the following laws

$$(P \Rightarrow Q) = \neg P \vee Q$$

$$(P \Leftrightarrow Q) = (P \Rightarrow Q) \wedge (Q \Rightarrow P)$$

$$= (\neg P \vee Q) \wedge (\neg Q \vee P)$$

// Replace all  $\Rightarrow$

**STEP II:** Reduce the NOT symbol by the formula  $(\neg(\neg P)) = P$  and apply De Morgan's theorem to bring negations before the atoms.

$$\neg(P \vee Q) = \neg P \wedge \neg Q$$

// De Morgan's theorem

$$\neg(P \wedge Q) = \neg P \vee \neg Q$$

**STEP III:** Use Distributive laws and other equivalent formula given in table III to obtain the normal form

$$P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$$

// in normal form

$$P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$$

# Propositional Logic – Example-1

## 3<sup>rd</sup> inference rule (resolution)

Resolution Rule:

$$(A \vee \underline{B}) \quad \wedge \quad (\underline{B} \vee C)$$

is  $A \vee C$

// B is resolved away

example:

$$C_1 \wedge C_2$$

$$C_1: P \vee \underline{Q} \vee \neg R.$$

$$C_2: \neg Q \vee W$$

$$\underline{P} \vee \underline{Q} \vee \neg R$$

$$\neg \underline{Q} \vee W$$

// Q is resolved away

$$P \vee \neg R \vee W$$

(Resolution tree)



# Propositional Logic – Example-2

## Problem:

- If it is “Hot”, Then it is “Humid”
- If it is “Humid”, Then it will “Rain”

Q: If it is “Hot”, Show that it will “Rain”

**Solution:** H: It is “Humid” (sentences)

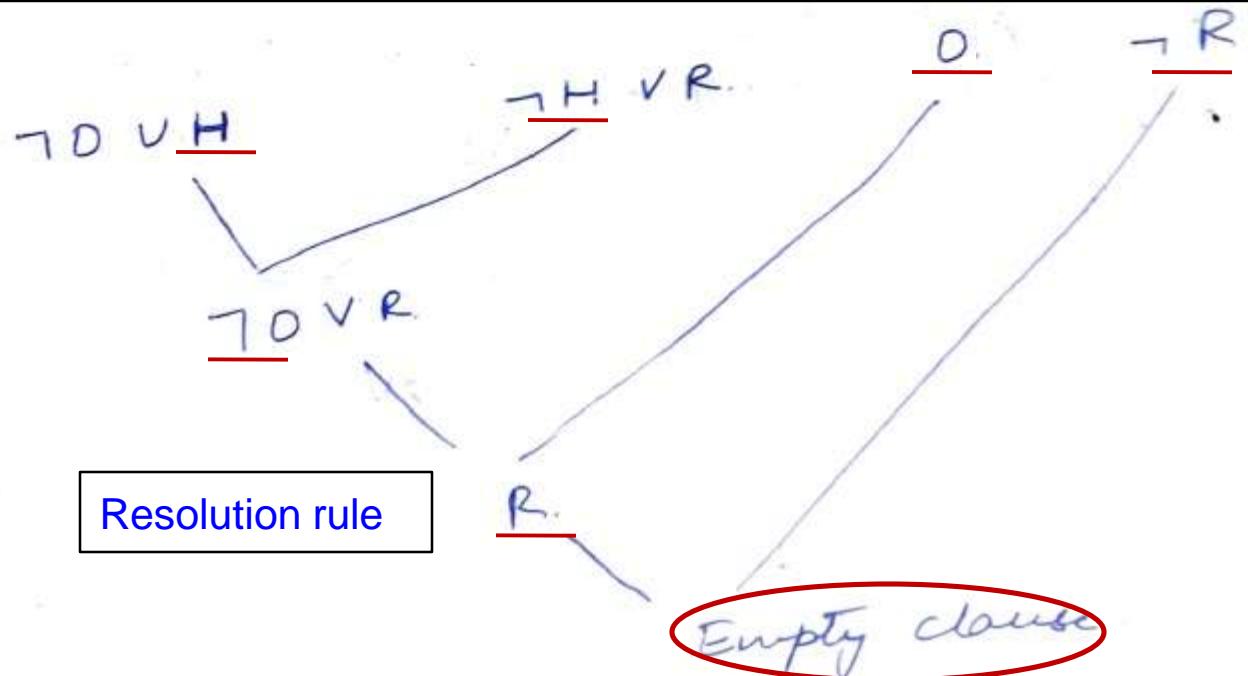
R: it will “Rain”

O: It is “Hot”

- If it is “Hot”, Then it is “Humid”:  $O \Rightarrow H$
- If it is “Humid”, Then it will “Rain” :  $H \Rightarrow R$
- It is “Hot” : O
- Add “Negation of Goal”:  $\neg R$

**CNF: Step-1**  
(eliminate  $\Rightarrow$ )

Apply Resolution  
Inference rule on  
H, O & R



# ★ Propositional Logic – Example-3

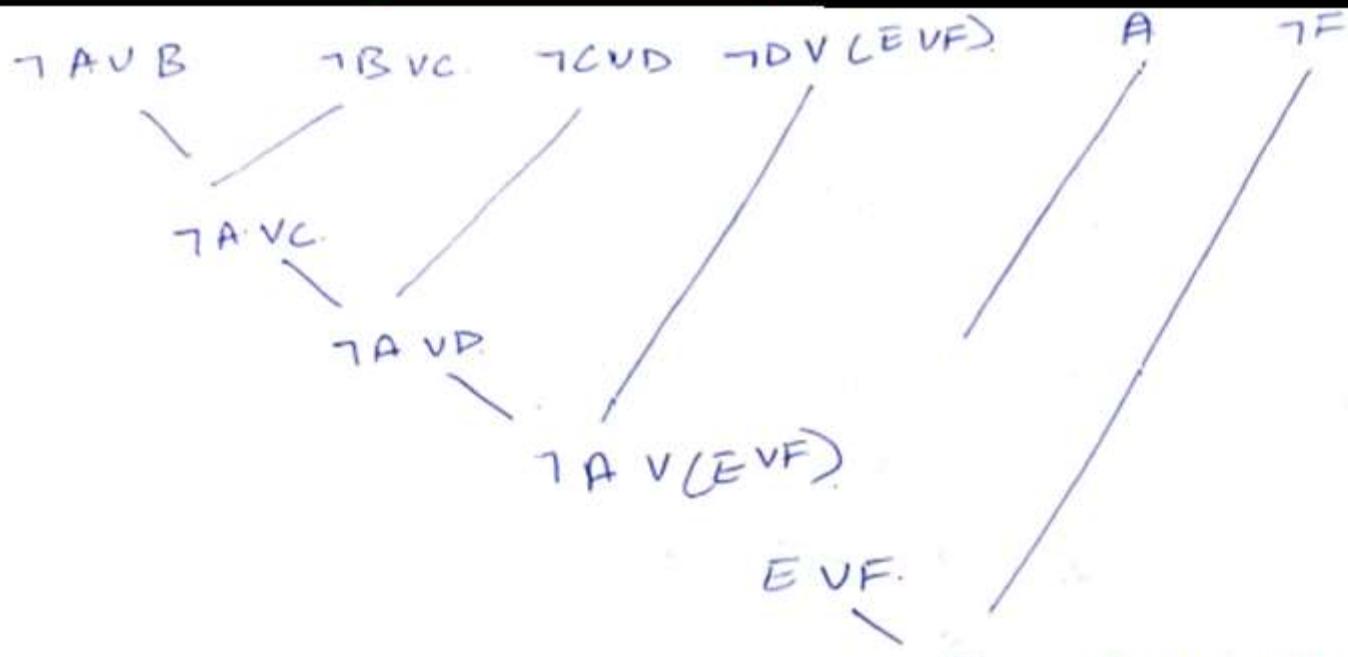
Example 3

Consider the following statements

$$A \rightarrow B, B \rightarrow C, C \rightarrow D, D \rightarrow E \vee F$$

(conclude  $A \rightarrow F$ )

Add "Negation of Goal":  $\neg(A \rightarrow F) = A \wedge \neg F$



we are left with a single clause 'E'  
which is "Not Empty Clause"

Hence  $A \rightarrow F$  is not valid

## 2. First-order Predicate Logic (FOPL)

- FOPL is a **symbolized reasoning system** in which each sentence is broken down into **(1) a subject** (a variable) & **(2) a predicate** (a function)
- The **predicate** modifies or defines the properties of the **subject**
- A **predicate** can only refer to a **single subject**
- A **sentence** in **FOPL** is written in the form
  - **Px** or **P(x)**, where **P** is the predicate & **x** is the subject (a variable)
- Complete sentences are logically combined & manipulated as done in Boolean algebra

# First-order Predicate Logic (FOPL)

- **Sentence** →      Atomic Sentence       $(P(x) \text{ or } x = y)$ 
  - | Sentence
  - | Connective Sentence ( $\Rightarrow | \wedge | \vee | \Leftrightarrow$ )
  - | Quantifier Variable ( $\forall | \exists$ ), ... Sentence
  - |  $\neg$  Sentence
- **Atomic Sentence** →  $\text{Predicate}(\text{Term}, \dots) | \text{Term} = \text{Term}$
- **Term** →      Function( $\text{Term}, \dots$ )    | Constant    | Variable
- **Connective** →  $\Rightarrow | \wedge | \vee | \Leftrightarrow$
- **Quantifier** →       $\forall | \exists$
- **Constant** →   A | 5 | Kolkata | ...
- **Variable** → a | x | s | ...
- **Predicate** →      Before | HasColor | ...
- **Function** → Is-Prof () | Is\_Person () | Is\_Dean ()| ...

# First-order Predicate Logic (FOPL)

- Consider a **subject** as a variable represented by **x**
  - Let **A** be a predicate "is an apple"
  - F** be a predicate "is a fruit"
  - S** be a predicate "is sour"
  - M** be a predicate "is mushy"
- Then we can say -

$\forall x \Rightarrow$  For All x  
 $\exists x \Rightarrow$  Some x

$$\forall x : Ax \Rightarrow Fx$$

which translates to "For all x, if x is an apple, then x is a fruit." We can also say such things as where the existential quantifier translates as "For some."

$$\exists x : Fx \Rightarrow Ax$$

$$\exists x : Ax \Rightarrow Sx$$

$$\exists x : Ax \Rightarrow Mx$$

# Example-1

 $\forall x : Ax \Rightarrow Fx$ 

which translates to "For all x, if x is an apple, then x is a fruit." We can also say such things as

 $\exists x : Fx \Rightarrow Ax$ 

where the existential quantifier translates as "For some."

 $\exists x : Ax \Rightarrow Sx$  $\exists x : Ax \Rightarrow Mx$ 

**1<sup>st</sup>** – If x is a apple  $\Rightarrow$  All x are fruits,

**2<sup>nd</sup>** – If x is a fruit  $\Rightarrow$  **some** x are apple

**3<sup>rd</sup>** – Some apples are sour

**4<sup>th</sup>** – Some apples are mushy

$\forall x \Rightarrow$  For All x

$\exists x \Rightarrow$  Some x

## Examples-2

### Formal definition – using FOPL

1. Lucy\* is a professor                           $\text{is-prof}(\text{lucy})$
2. All professors are people.                 $\forall x (\text{is-prof}(x) \rightarrow \text{is-person}(x))$
3. John is the dean.                               $\text{is-dean}(\text{John})$
4. Deans are professors.                         $\forall x (\text{is-dean}(x) \rightarrow \text{is-prof}(x))$
5. All professors consider the dean a friend or don't know him.  
     $\forall x (\forall y (\text{is-prof}(x) \wedge \text{is-dean}(y) \rightarrow \text{is-friend-of}(y,x) \vee \neg \text{knows}(x,y)))$
6. Everyone is a friend of someone.             $\forall x (\exists y (\text{is-friend-of}(y,x)))$
7. People only criticize people that are not their friends.  
     $\forall x (\forall y (\text{is-person}(x) \wedge \text{is-person}(y) \wedge \text{criticize}(x,y) \rightarrow \neg \text{is-friend-of}(y,x)))$
8. Lucy criticized John .                          $\text{criticize}(\text{lucy}, \text{John})$

Question: Is John no friend of Lucy?  
 $\neg \text{is-friend-of}(\text{John}, \text{lucy})$



# FOPL – Example-3

## Problem:

- Show the validity of the following sentence
- All men are mortal. John is a man. Therefore John is Mortal.

$\text{Man}(n)$  –  $n$  is a man

$\text{Mortal}(n)$  –  $n$  is mortal

$(\forall n)(\text{Man}(n) \rightarrow \text{Mortal}(n))$  // For all 'x', if 'x' is man  $\Rightarrow$  'x' is mortal

$\neg (\forall n) \text{Man} \vee \text{Mortal}(n)$  // Replacing  $\Rightarrow$

$((\exists n) \neg \text{Man}(n) \vee \text{Mortal}) \wedge \text{Man}(\text{John})$  // All men are mortal.

John is a man

$(\neg \text{man}(x) \vee \text{Mortal}(x)) \wedge \text{man}(\text{John}) \wedge \neg \text{Mortal}(\text{John})$



if  $x = \text{John}$

Empty Clause

// Negation of Goal

Hence True



# FOPL – Example-3

## Problem:

- Show the validity of the following sentence
- All men are mortal. John is a man. Therefore John is Mortal.

$(\forall n) (\text{Man}(n) \rightarrow \text{Mortal}(n))$

Replace =>

$\neg \text{Man}(n) \text{ Mortal}(n)$

$\text{Man}(\cancel{n})$

$n = \text{John}$

$\text{Mortal}(\text{John})$

Negation of Goal

$\neg \text{Mortal}(\text{John})$



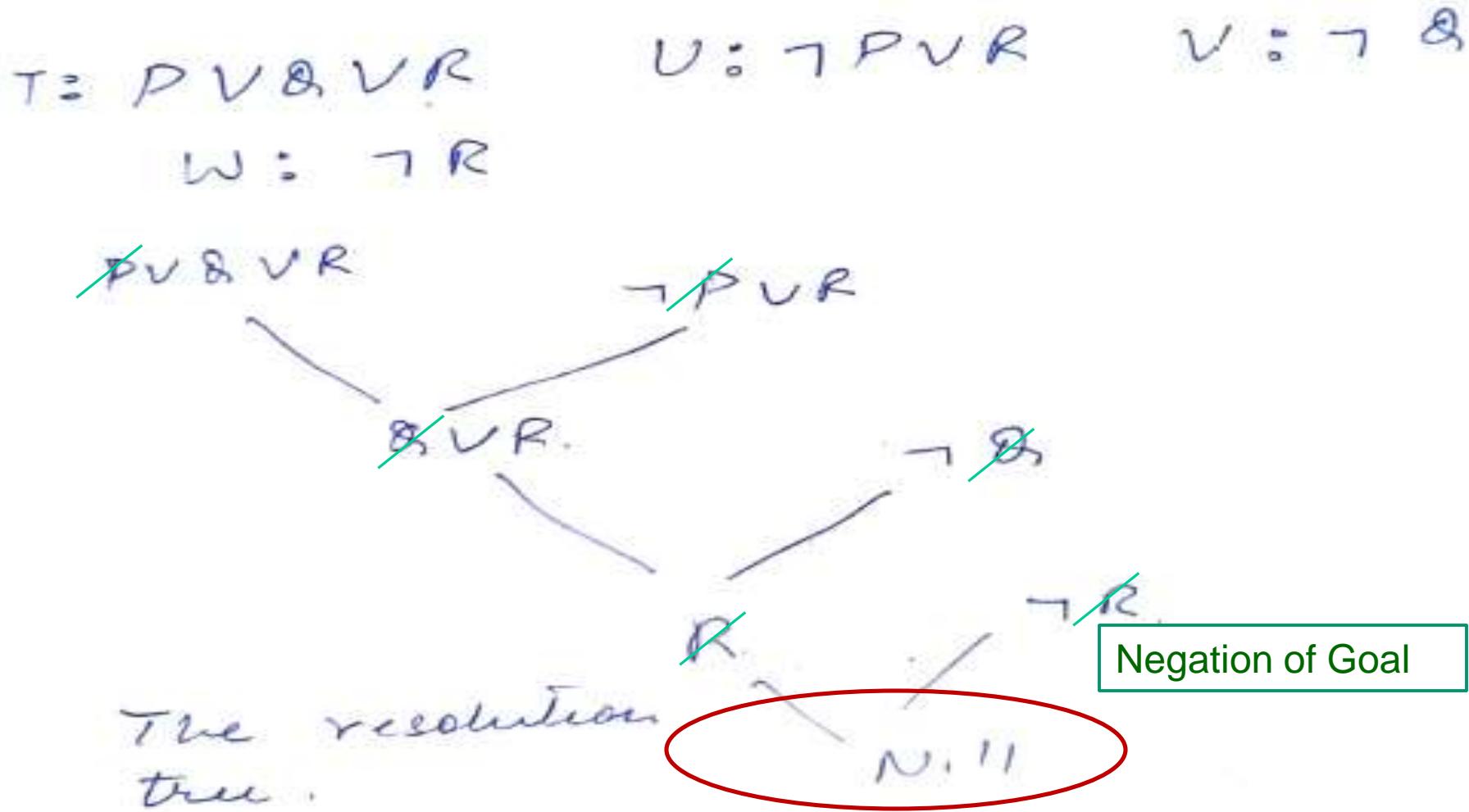
Empty clause

# FOPL – Example-4



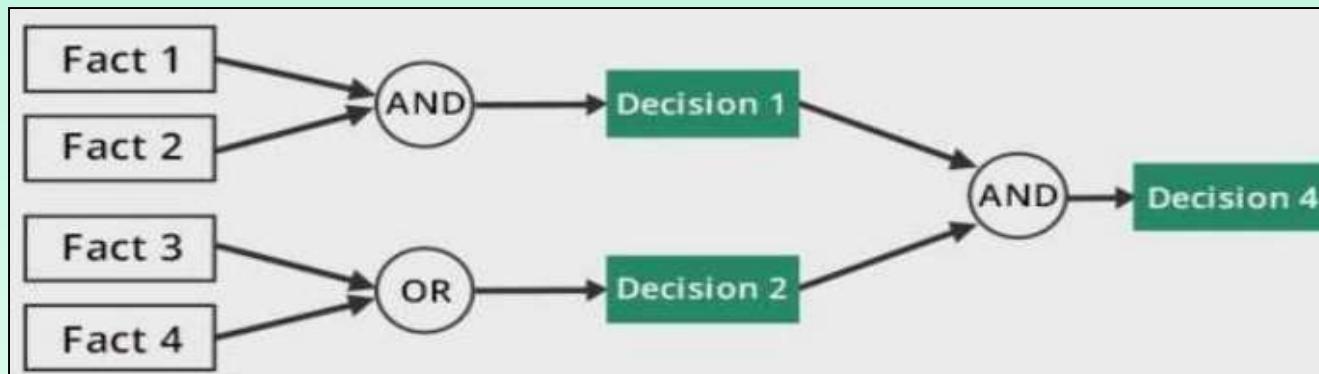
## Problem:

- Given the following predicate show how resolution process can be applied



# Forward Chaining

- **Forward chaining** is a data driven method of
  - Deriving a particular goal
    - from a given knowledge base & a set of inference rules
- The application of inference rules results in new knowledge
  - which is then added to the knowledge base
- Used to answer the question “**What can happen next**”
- The inference engine applies chain of conditions, facts & rules
  - to arrive at a solution (**decision** or **goal**)



# Forward Chaining

- The system starts from a set of facts & a set of rules
  - Tries to find ways of using them to deduce a conclusion (goal)
- Called **data-driven reasoning** because the reasoning starts from a set of data and ends up at the goal
- **1st Step** – Take the facts from the *fact database* & see if any combination of these matches any of the components of rules in the *rule database*
- **2nd Step** – In case of a match, the rule is triggered (fired)
- **3rd Step** – Then it's conclusion is added to the *facts database*
- **4th Step** - If the conclusion is an action, then the system causes that action to take place

# Forward Chaining – Example - Elevator

## Rule 1

IF on first floor and button is pressed on first floor  
THEN open door

## Rule 2

IF        on first floor  
AND      button is pressed on second floor  
THEN     go to second floor

## Rule 3

IF        on first floor                                  // Fact-1  
AND      button is pressed on third floor            // Fact-2  
THEN     go to third floor                              // Conclusion added to KB

# Forward Chaining - Example - Elevator

## Rule 4

IF                   on second floor  
AND                 button is pressed on first floor // Fact-4 added to KB  
AND                 already going to third floor  
THEN               remember to go to first floor later

Let us imagine that we start with the following facts in our database:

### Fact 1

At first floor

### Fact 2

Button pressed on third floor

### Fact 3

Today is Tuesday

# Forward Chaining - Example - Elevator

- The system examines the rules & finds that Facts 1 & 2 match the components of Rule 3
- Rule 3 fired & its conclusion “Go to 3<sup>rd</sup> floor” is added to the facts database
- This results in the elevator heading to the 3<sup>rd</sup> floor
- Note that Fact 3 (*today is Tuesday*) was ignored because it did not match the components of any rules
- Assuming the elevator is going to the 3<sup>rd</sup> floor & has reached the 2<sup>nd</sup> floor, when the button is pressed on the 1<sup>st</sup> floor
- The fact “Button pressed on first floor” is now added to the database, which results in Rule 4 firing (*remember to go to first floor*)



# Forward Chaining - Example – Knowledge Base

- **Question**

- The law says that it is a **crime** for an **American** to **sell weapons to hostile nations**.
  - The country **Nono**, an **enemy America**, has some **missiles**, and all of its missiles were sold to it by **Col. West**, who is an **American**.
- **Prove that Col. West is a criminal.**

# Sentences in FOPL

- **It is a crime for an American to sell weapons to hostile nations:**

$\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x,y,z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$

$< x - \text{person}, y - \text{weapon}, z - \text{country} >$

- **Nono...has some missiles**

$\exists x \text{ Owns}(\text{Nono}, x) \wedge \text{Missiles}(x)$  // some of the weapons owned by Nono are missiles

$\text{Owns}(\text{Nono}, M_1)$  and  $\text{Missle}(M_1)$

$< x - \text{weapon}, y - \text{missile} >$

- **All of its missiles were sold to it by Col. West**

$\forall x \text{ Missle}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono})$

$< x - \text{missile} >$

- **Missiles are weapons**

$\text{Missle}(x) \Rightarrow \text{Weapon}(x)$

- An enemy of America counts as “hostile”

*Enemy( x, America )  $\Rightarrow$  Hostile(x)*

- Col. West who is an American

*American( Col. West )*

- The country Nono, an enemy of America

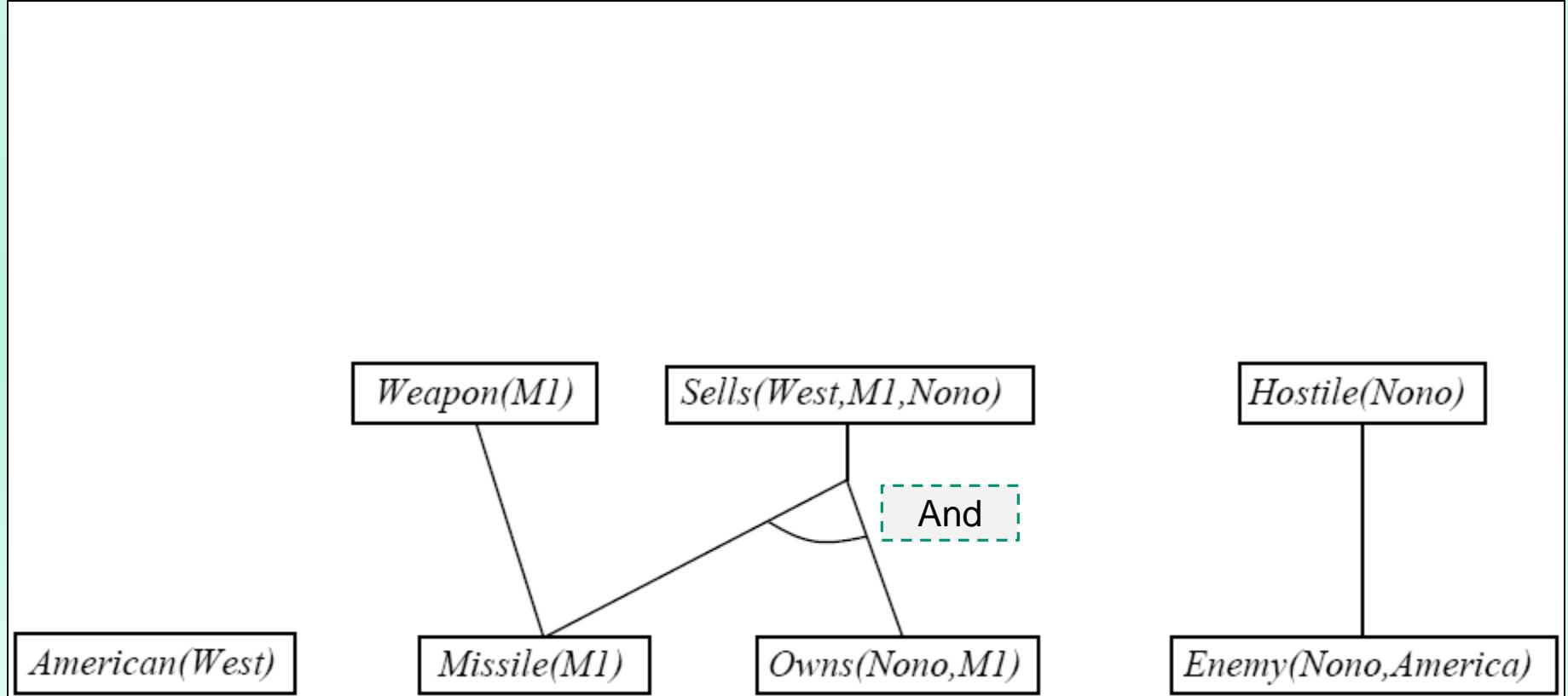
*Enemy(Nono, America)*

*American(West)*

*Missile(M1)*

*Owns(Nono,M1)*

*Enemy(Nono,America)*



## Conclusion



*Criminal(West)*

*Weapon(M1)*

*Sells(West,M1,Nono)*

*Hostile(Nono)*

*American(West)*

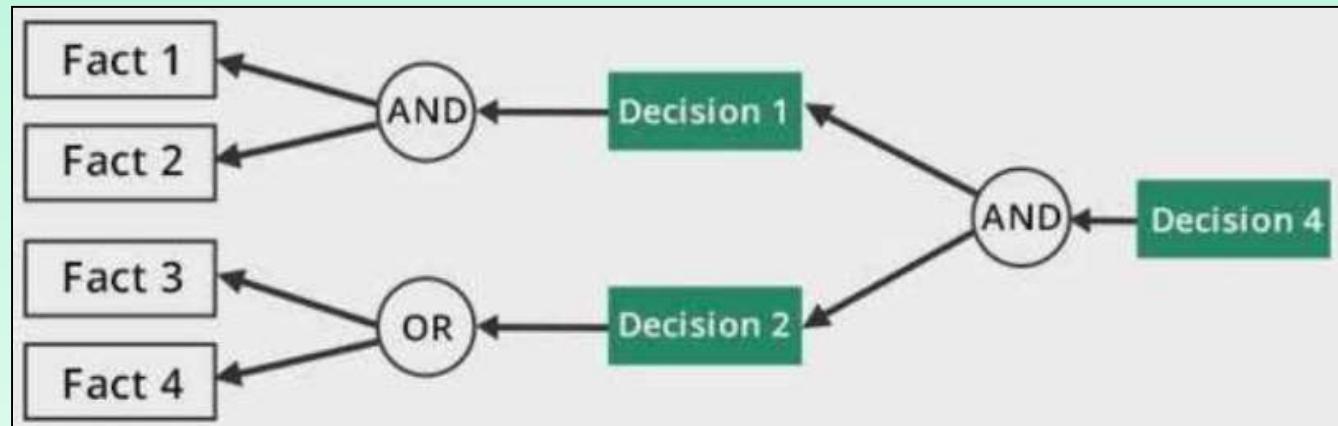
*Missile(M1)*

*Owns(Nono,M1)*

*Enemy(Nono,America)*

# Backward Chaining

- Backward chaining is a goal driven method of
  - Deriving a particular goal from a given knowledge base & a set of inference rules
- Inference rules are applied by matching the goal to the results of the relations stored in the knowledge base
- Used to answer the question “**Why this happened**”
- Based on what has already happened, the inference engine tries to find out which **conditions (causes or reasons)** could have happened for this **result**



# Backward Chaining

- The system starts from a conclusion (hypothesis to prove or goal)
  - Tries to show how the conclusion has been reached from the rules & facts in the database
- Reasoning in this way is called as **goal-driven reasoning**
- **Steps** – Start with the goal state & see what actions could lead to it of the components of rules in the *rule database*
- **Ex:**
  - If the goal state is “blocks arranged on a table”
  - Then one possible action is to “place a block on the table”
  - This action may not be possible from the start state
  - Further actions need to be added before this action
  - In this way, a plan is formulated starting from the goal & working back toward the start state

# Backward Chaining

- Backward chaining ensures that each action that is taken is one that will definitely lead to the goal
- In many cases, Backward Chaining will make the planning process far more efficient compared to Forward Chaining

# Example - WEATHER FORECAST SYSTEM

Rule I

If we suspect temperature is less than  $20^{\circ}$   
AND there is humidity in the air  
Then there are chances of rain

③

// Premise-1 – R1 (conclusion of rule-2)

// Rule-1

Rule II

If Sun is behind the clouds  
AND air is very cool.  
Then we suspect temperature is less than  $20^{\circ}$ .

①

// Premise-1 – R2 (This is ‘Known’)

// Premise-2 – R2 (This is ‘Known’)

// Fire R-2

Rule III If air is very heavy

Then there is humidity in the air.

②

// Premise-1 – R3 (This is ‘Known’)

// Conclusion of R-3

- Suppose we have been given the following facts,
  - a) Sun is behind the clouds
  - b) Air is very heavy & cool
- **Problem:** Use Backward chaining to conclude there are **chances of rain**

# Example - WEATHER FORECAST SYSTEM

Step	Description	Working Memory
1	Goal “There are chances of rain.” Not in Working Memory.	
2	Find rules with our goal “There are chances of rain” in conclusion: It is in Rule 1.	
3	Now see if Rule 1, premise 1 is known “we suspect temperature is less than 20°”.	
4	This is conclusion of rule 2. So going to Rule 2. The premise 1 of rule 2 is “Sun is behind the clouds”.	
5	This is primitive. We ask from user Response: Sun is behind the clouds. Yes	

# Example - WEATHER FORECAST SYSTEM

6	See if Rule 2, premise 2 is known “Air is very cool”.	
7	This is also primitive. We ask its Response: Yes. Both conditions of Rule 2 are met so Fire rule 2	Sun is behind the clouds. Air is very cool. <b>We suspect temperature is less than 20°.</b>
8	So Rule 1 premise 1 is in working memory, coming to Rule 1, premise 2 “There is humidity in the air”	Sun is behind the clouds. Air is very cool. <b>We suspect temperature is less than 20°.</b>
9	This is conclusion of Rule 3. So see if Rule 3, premise 1 is known “Air is very heavy”.	Sun is behind the clouds. Air is very cool. <b>We suspect temperature is less than 20°.</b>

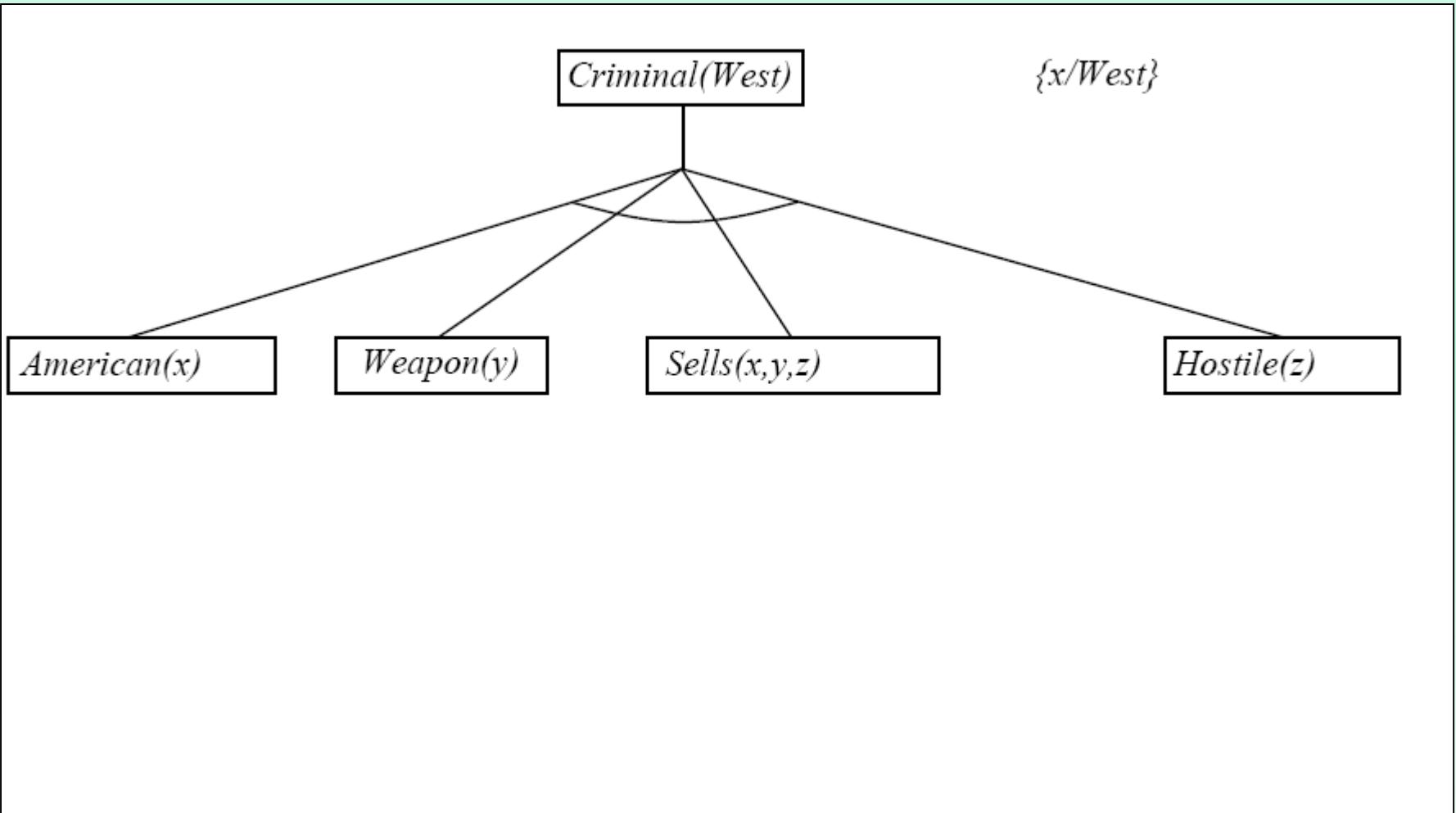
# Example - WEATHER FORECAST SYSTEM

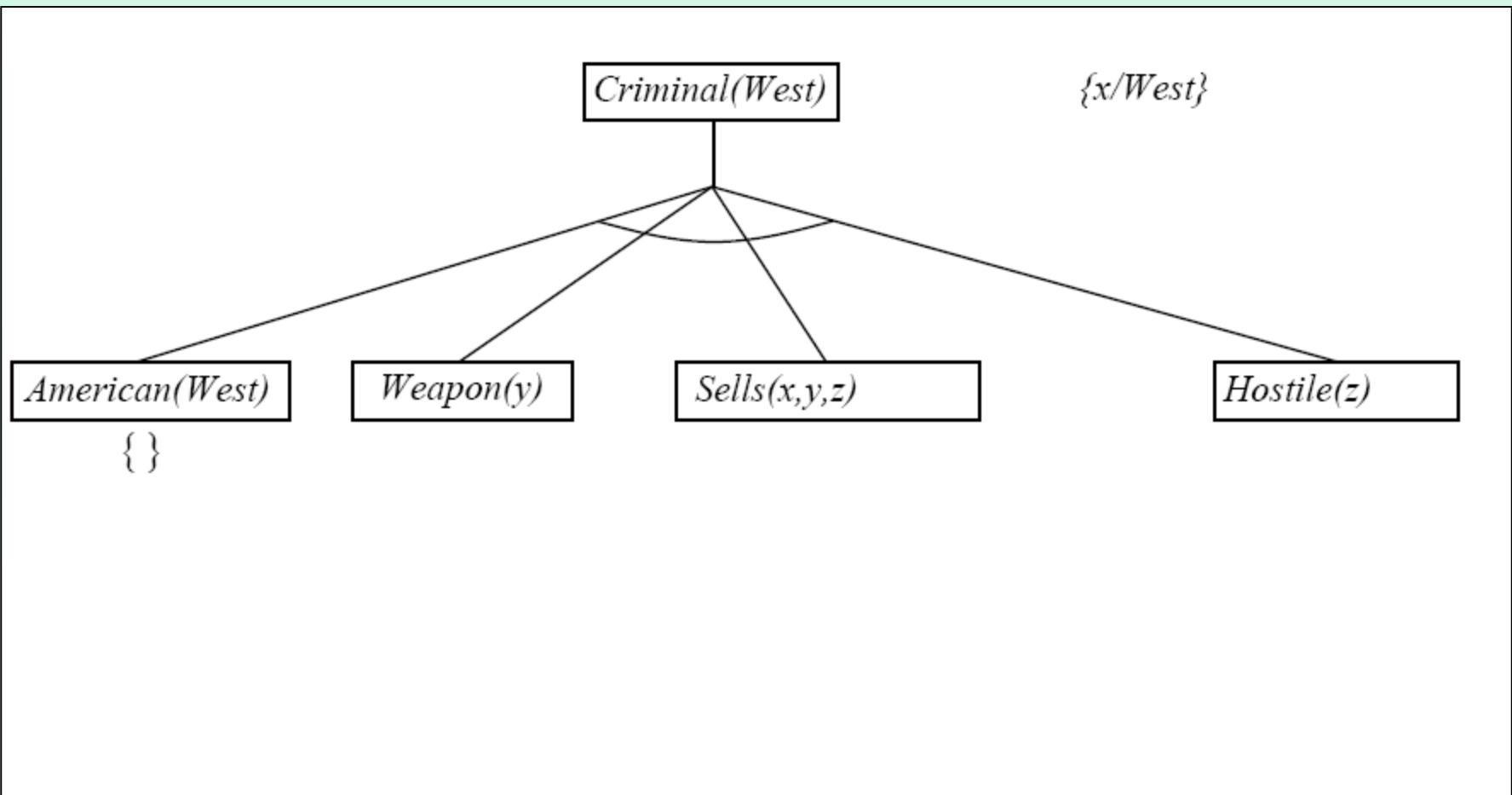
10	This is primitive so asking from user Response: Yes. Fire rule	Sun is behind the clouds. Air is very cool. We suspect temperature is less than 20°. <b>There is humidity in the air.</b>
11	Now Rule 1 premise 1 and 2 both are in working memory so fire Rule 1.	Sun is behind the clouds. Air is very cool. Air is very heavy. We suspect temperature is less than 20°. There is humidity in the air. <b>There are chances of rain.</b>

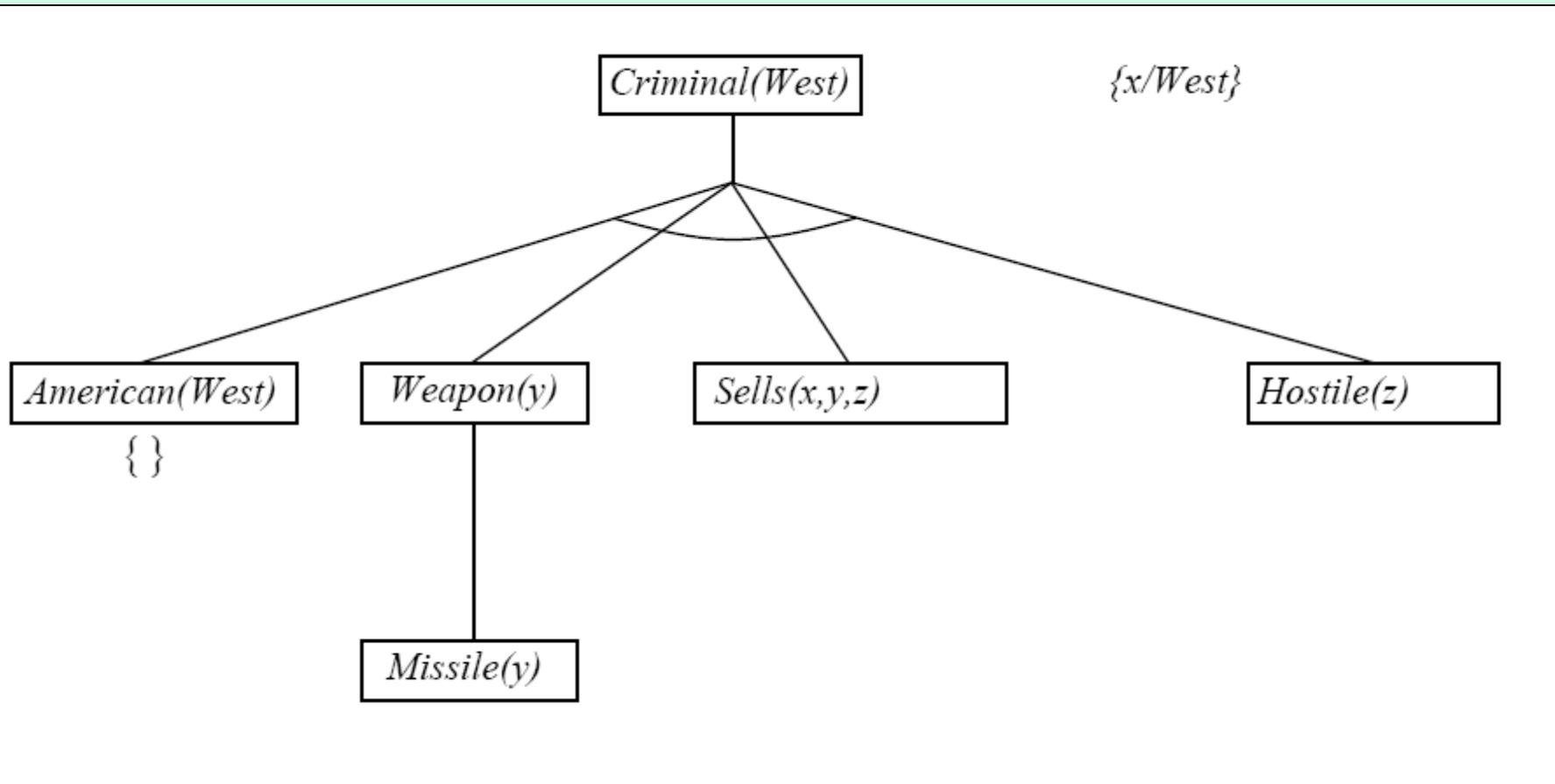


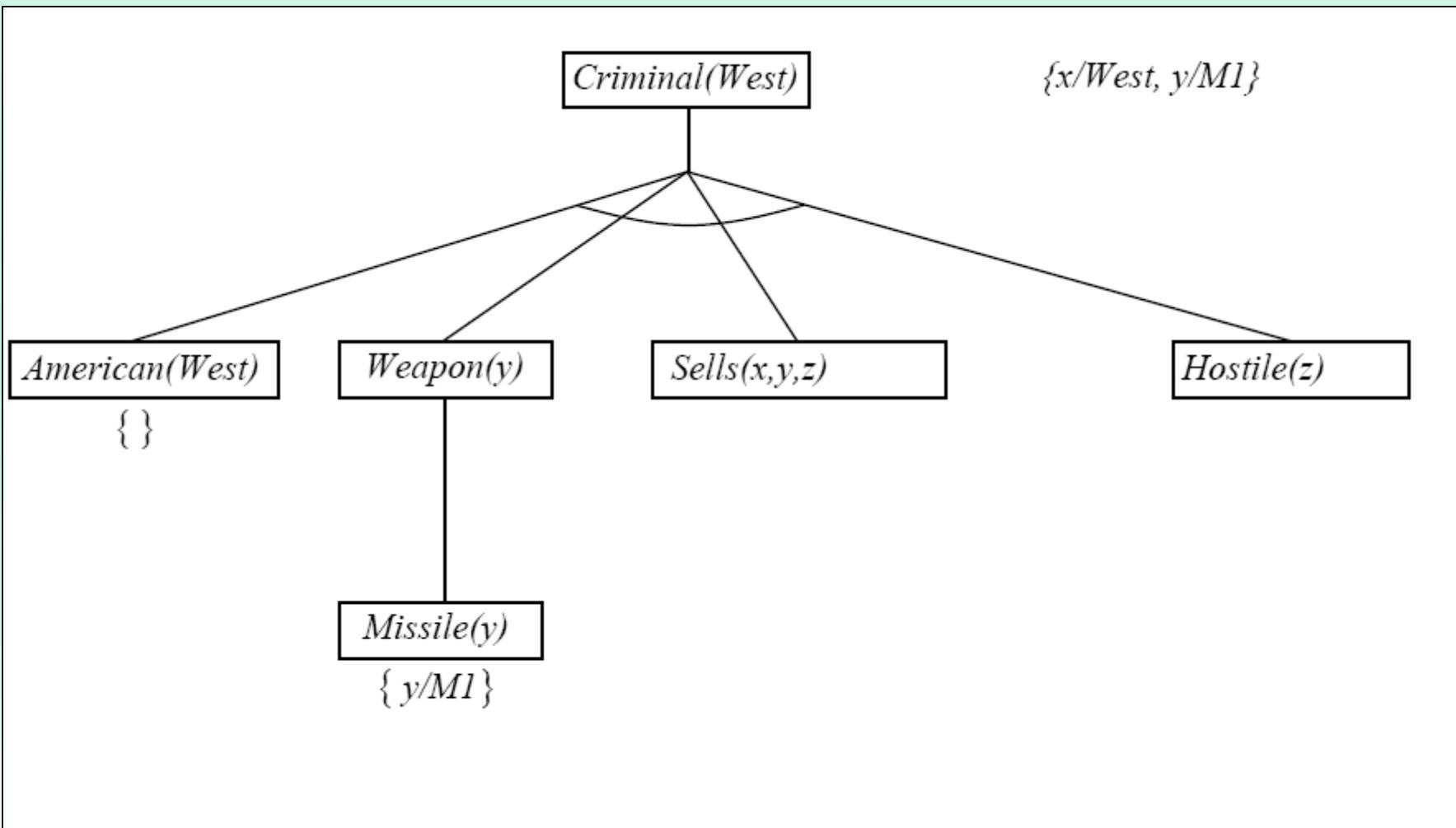
# Backward Chaining Example

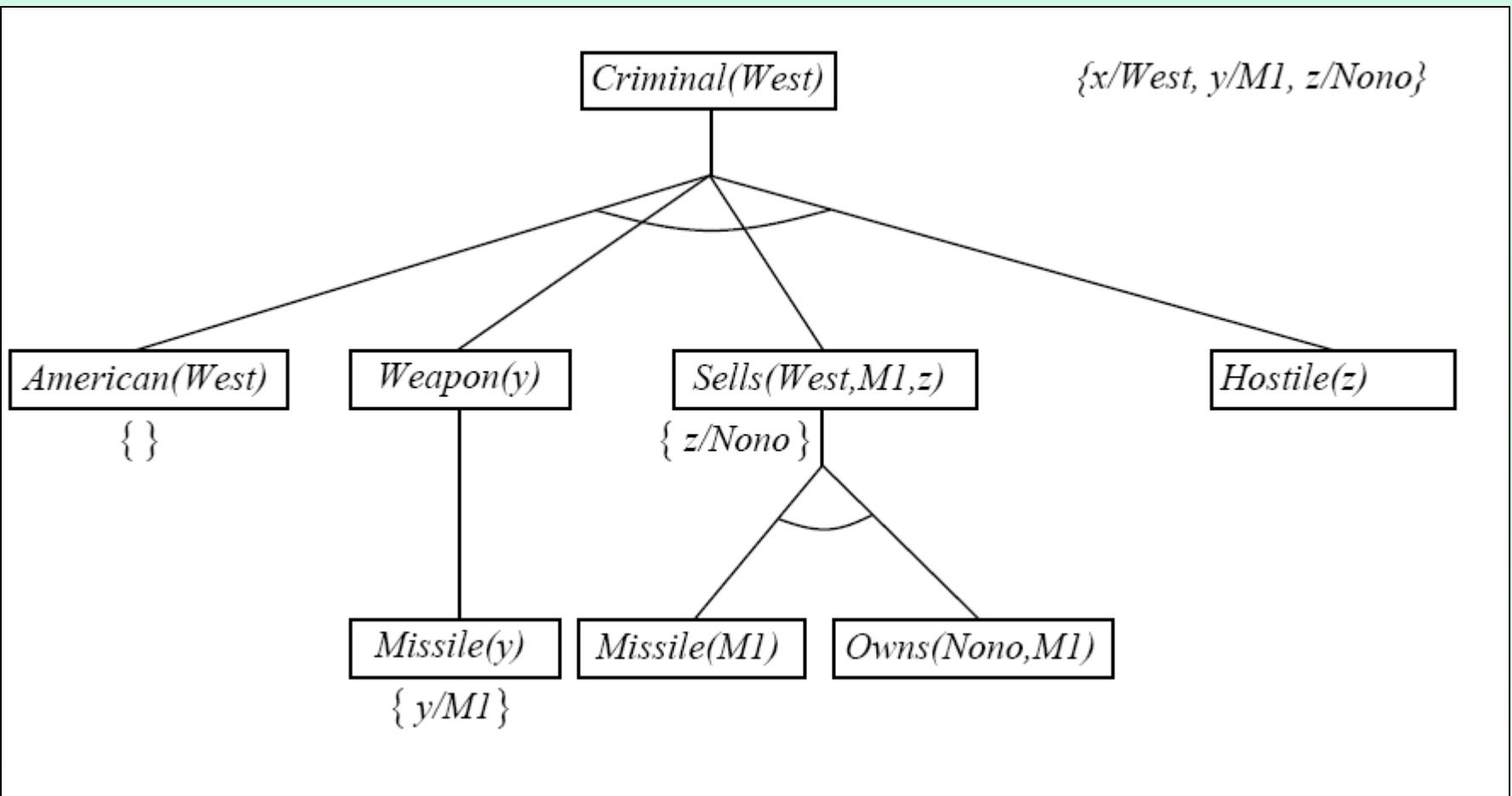
*Criminal(West)*

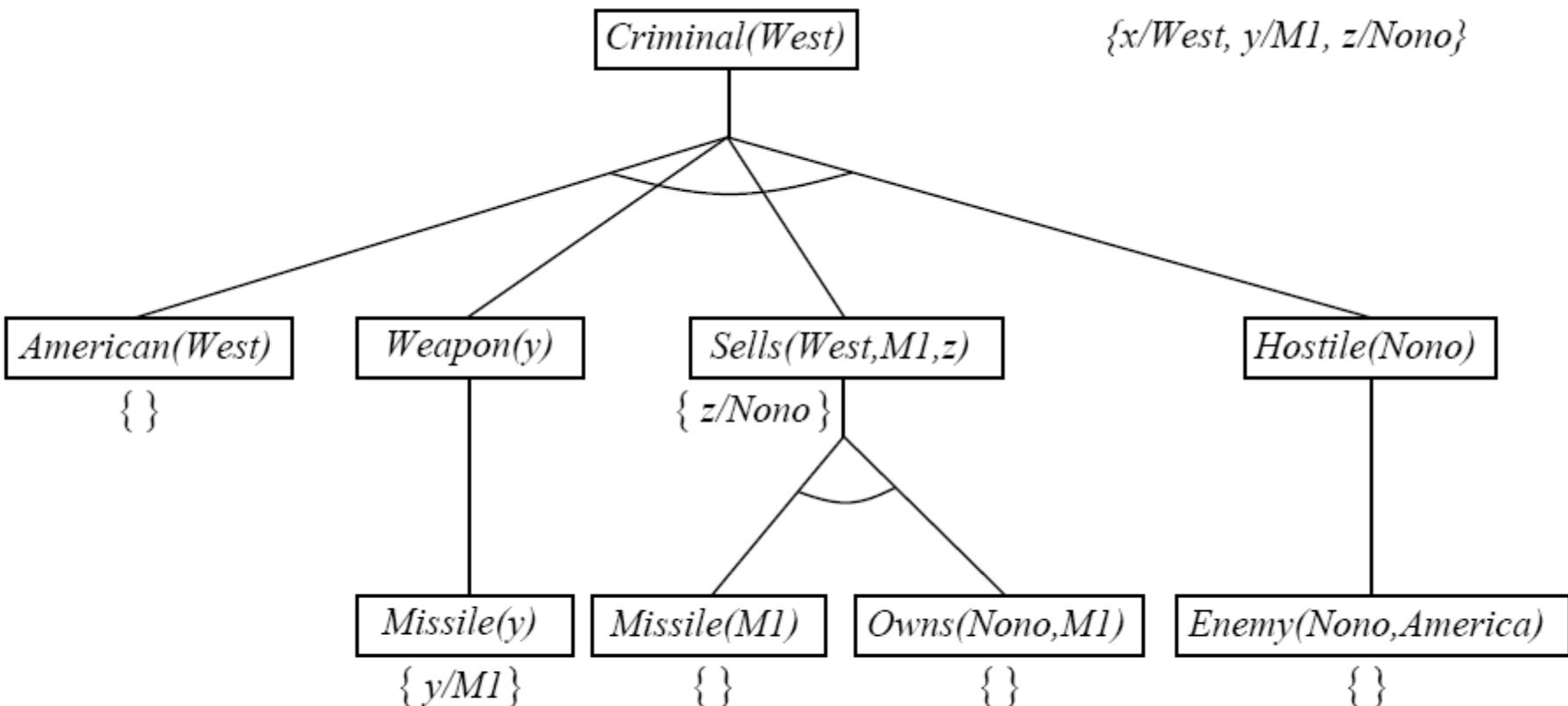












# Backward Chaining Algorithm

```
function FOL-BC-ASK( $KB$ ,  $goals$ ,  $\theta$ ) returns a set of substitutions
  inputs:  $KB$ , a knowledge base
           $goals$ , a list of conjuncts forming a query
           $\theta$ , the current substitution, initially the empty substitution  $\{ \}$ 
  local variables:  $ans$ , a set of substitutions, initially empty
  if  $goals$  is empty then return  $\{\theta\}$ 
   $q' \leftarrow \text{SUBST}(\theta, \text{FIRST}(goals))$ 
  for each  $r$  in  $KB$  where  $\text{STANDARDIZE-APART}(r) = (p_1 \wedge \dots \wedge p_n \Rightarrow q)$ 
    and  $\theta' \leftarrow \text{UNIFY}(q, q')$  succeeds
     $ans \leftarrow \text{FOL-BC-ASK}(KB, [p_1, \dots, p_n | \text{REST}(goals)], \text{COMPOSE}(\theta', \theta)) \cup ans$ 
  return  $ans$ 
```



# Planning



***Dr. Pulak Sahoo***

Associate Professor  
Silicon Institute of Technology



# Planning - Topics

- **Planning: Introduction to Planning**
- **Planning vs Problem Solving**
  - Example – The Blocks World
  - Example: Transportation (Air Cargo) problem
  - Example: Spare Tire problem
- **Representation of Action**
- **Partial Order Planning**
  - Example: Shoes & Socks Problem
- **Hierarchical Planning**
  - Example: House Construction

# Introduction

- **Planning:** *the task of finding a sequence of actions that will achieve a given goal when executed from a given world state*
- Given are:
  - a set of operator descriptions (defining the possible primitive actions by the agent),
  - an initial state description, and
  - a goal state description or predicate,
- The **computed plan** is:
  - a sequence of operator instances, such that executing them in the initial state will change the world to a state satisfying the goal-state description.
- **Classical planning env.s** are *fully observable, deterministic, finite, static & discrete*

# Introduction

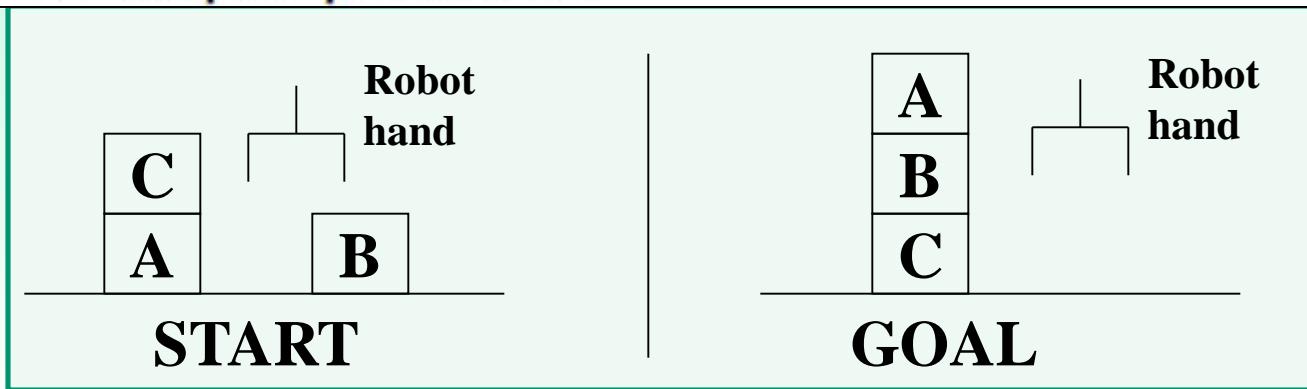
- Planning agents :
  - Construct plans to achieve goals, then executes them
- Planning Agent specifies:
  - States & Goal – as logical sentences
  - Actions – as preconditions/outcomes
  - Plan – as sequence of actions

# Introduction

- **Key ideas:**
  - Use descriptions in a **formal language** – FOPL
  - **Divide-and-conquer** algorithms
  - **States/goals** represented by **sets of sentences**
  - **Actions** represented by logical descriptions of **preconditions** & **effects**
  - **Planner** can **add actions** to plan **whenever needed**
  - Connection between **order of planning** & **order of execution** not **essential**

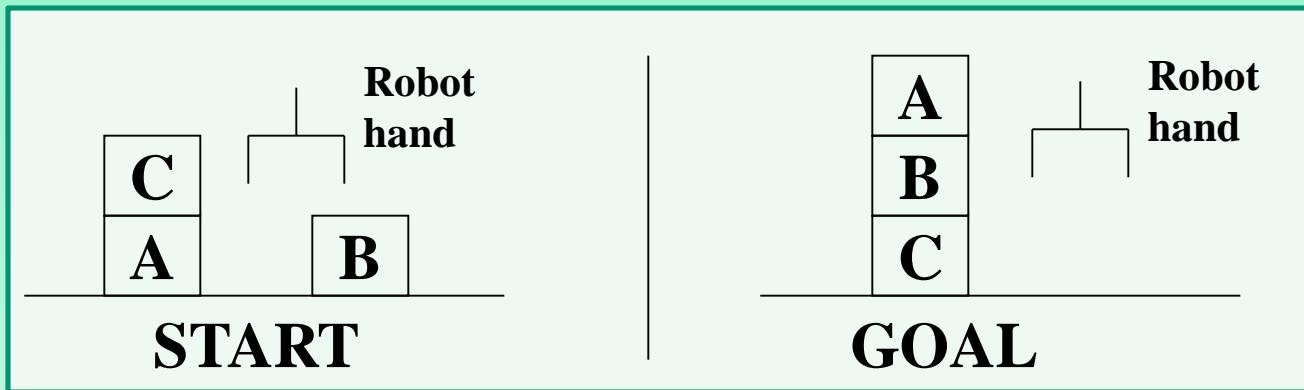
# Example – The Blocks World

- Many planning systems can be illustrated using the blocks world.
- The blocks world consists of a number of blocks and a table.
- The blocks can be picked up and moved around.
- The following shows the start and goal states of a simple problem:



# Example – The Blocks World

- A planning system has rules with precondition deletion & addition list



## Sequence of actions :

1. Grab C
2. Pickup C
3. Place on table C
4. Grab B
5. Pickup B
6. Stack B on C
7. Grab A
8. Pickup A
9. Stack A on B

# Example – The Blocks World

- **Fundamental Problem :**

- The “frame problem” is concerned with the question

“**What piece of knowledge is relevant to the situation?**

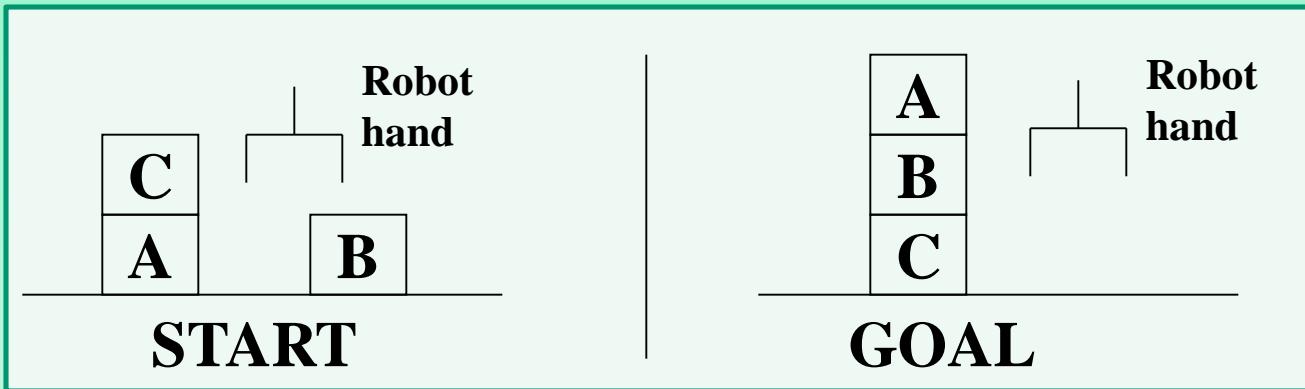
---

- **Fundamental Assumption : “Closed world assumption”**

- If something is not asserted (specified) in the knowledge base, it is assumed to be false
  - Called “**Negation by failure**”

# Example – The Blocks World

- A **planning system** – Has rules with precondition, deletion list & addition list



on(B, table)

on(A, table)

on(C, A)

hand empty

clear(C)

clear(B)

on(C, table)

on(B, C)

on(A, B)

hand empty

clear(A)

# Rules

## *R1 : pickup(x)*

- Precondition & Deletion List: hand empty, on(x,table), clear(x)
- Add List: holding(x)

## *R2 : putdown(x)*

- Precondition & Deletion List: holding(x)
- Add List: hand empty, on(x,table), clear(x)

## *R3 : stack(x,y) // place x on y*

- Precondition & Deletion List: holding(x), clear(y)
- Add List : on(x,y), clear(x)

## *R4 : unstack(x,y) // Remove x placed on y*

- Precondition & Deletion List: on(x,y), clear(x)
- Add List: holding(x), clear(y)

# Plan for the block world problem

- For the given problem, Start → Goal can be achieved by the following sequence:
  1. **Unstack(C,A)** // remove C placed on A
  2. **Putdown(C)** // put C on table
  3. **Pickup(B)**
  4. **Stack(B,C)** // put B on C
  5. **Pickup(A)**
  6. **Stack(A,B)** // put A on B
- **Execution of a plan:** achieved through a data structure called Triangular Table

# Planning vs Problem Solving

- Planning agent is **similar** to problem solving agent as both
  - Construct plans to achieve goals, then executes them
- Note: Issues with the problem solving techniques - Huge state space & large branching factor
- Planning agent is **different** from problem solving agent in:
  - Representation of goals, states, actions
  - Use of logical representations & Way it searches for solutions
- Planning systems follow “Divide-and-conquer” approach

# Example: Transportation (Air Cargo) problem

## Example: Air Cargo Transport

- Loading/unloading cargo onto/off planes and flying it from place to place.
- Actions: Load, Unload, and Fly
- States:
  - $\text{In}(c, p)$ : cargo  $c$  is inside plane  $p$
  - $\text{At}(x, a)$ : object  $x$  (plane/cargo) at airport  $a$

```
Init(At(C1, SFO)  $\wedge$  At(C2, JFK)  $\wedge$  At(P1, SFO)  $\wedge$  At(P2, JFK)
      $\wedge$  Cargo(C1)  $\wedge$  Cargo(C2)  $\wedge$  Plane(P1)  $\wedge$  Plane(P2)
      $\wedge$  Airport(JFK)  $\wedge$  Airport(SFO))
```

```
Goal(At(C1, JFK)  $\wedge$  At(C2, SFO)) \ swap cargos
```

```
Action(Load(c, p, a), \ load cargo c into plane p at airport a
```

PRECOND:  $\text{At}(c, a) \wedge \text{At}(p, a) \wedge \text{Cargo}(c) \wedge \text{Plane}(p) \wedge \text{Airport}(a)$

EFFECT:  $\neg \text{At}(c, a) \wedge \text{In}(c, p)$

```
Action(Unload(c, p, a), \ unload cargo c from plane p at airport a
```

PRECOND:  $\text{In}(c, p) \wedge \text{At}(p, a) \wedge \text{Cargo}(c) \wedge \text{Plane}(p) \wedge \text{Airport}(a)$

EFFECT:  $\text{At}(c, a) \wedge \neg \text{In}(c, p)$

```
Action(Fly(p, from, to), \ plane p flies from airport 'from' to 'to'
```

PRECOND:  $\text{At}(p, from) \wedge \text{Plane}(p) \wedge \text{Airport}(from) \wedge \text{Airport}(to)$

EFFECT:  $\neg \text{At}(p, from) \wedge \text{At}(p, to)$

# Example: Spare Tire problem

## Example: Spare Tire Problem

- **Initial state:** a flat tire on the axle and a good spare tire in the trunk
- **Goal state:** have a good spare tire properly mounted onto the car's axle

```
Init(At(Flat, Axle)  $\wedge$  At(Spare, Trunk))
```

```
Goal(At(Spare, Axle))
```

```
Action(Remove(Spare, Trunk),
```

```
    PRECOND: At(Spare, Trunk)
```

```
    EFFECT:  $\neg$  At(Spare, Trunk)  $\wedge$  At(Spare, Ground))
```

```
Action(Remove(Flat, Axle),
```

```
    PRECOND: At(Flat, Axle)
```

```
    EFFECT:  $\neg$  At(Flat, Axle)  $\wedge$  At(Flat, Ground))
```

```
Action(PutOn(Spare, Axle),
```

```
    PRECOND: At(Spare, Ground)  $\wedge$   $\neg$  At(Flat, Axle)
```

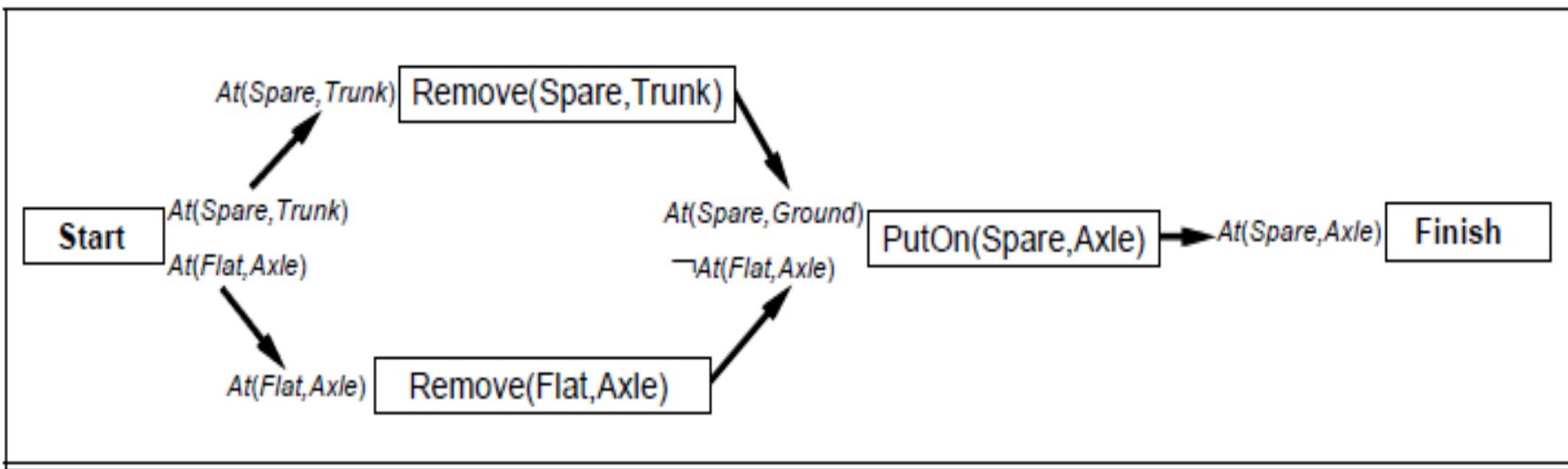
```
    EFFECT:  $\neg$  At(Spare, Ground)  $\wedge$  At(Spare, Axle))
```

```
Action(LeaveOvernight, \ If requirements are not satisfied, leave it for next day
```

```
    PRECOND:
```

```
    EFFECT:  $\neg$  At(Spare, Ground)  $\wedge$   $\neg$  At(Spare, Axle)  $\wedge$   $\neg$  At(Spare, Trunk)  
 $\wedge$   $\neg$  At(Flat, Ground)  $\wedge$   $\neg$  At(Flat, Axle))
```

# Example: Spare Tire problem



**Figure 11.10** The final solution to the tire problem. Note that *Remove(Spare, Trunk)* and *Remove(Flat, Axle)* can be done in either order, as long as they are completed before the *PutOn(Spare, Axle)* action.

# Representation of Action

## Example – Transportation (Air Cargo) problem

- **Representation for Actions**
- **Action description** – serves as a name for possible action
- **Precondition** – a conjunction of positive literals // state required before action can be applied
- **Effect** – a conjunction of literals (positive or negative)

<b>Op ( ACTION:</b>	<b>Go(there),</b>
<b>PRECOND:</b>	<b>At(here) <math>\wedge</math> Path(here, there),</b>
<b>EFFECT:</b>	<b>At(there) <math>\wedge</math> <math>\neg</math>At(here) )</b>

**Representation of actions.** An action is specified in terms of the preconditions that must hold before it can be executed and the effects that ensue when it is executed. For example, an action for flying a plane from one location to another is:

<i>Action(Fly(p, from, to),</i>
<i>  PRECOND:At(p, from) <math>\wedge</math> Plane(p) <math>\wedge</math> Airport(from) <math>\wedge</math> Airport(to)</i>
<i>  EFFECT:<math>\neg</math>At(p, from) <math>\wedge</math> At(p, to))</i>

# Partial Order Planning

**Idea:** (Plan to reach goal is divided into sub-plans with sub-goals)

- Work on several sub-goals independently (*Divide & conquer*)
- Solve them with sub-plans & Combines the sub-plans
- Flexibility in ordering the sub-plans
- Least commitment strategy:
  - Delaying the ordering choice during the search
  - Ex: Leave actions unordered unless they must be sequential

**Strengths of POP:**

- Early pruning of the search space parts in case of irresolvable conflicts
- The solution is partial-order plan (produce flexible plans)

# Partial Order Planning

## Example – Shoes & Socks Problem

### Actions

Op( **ACTION**: RightShoe, **PRECOND**: RightSockOn, **EFFECT**: RightShoeOn)

Op( **ACTION**: RightSock, **EFFECT**: RightSockOn)

Op( **ACTION**: LeftShoe, **PRECOND**: LeftSockOn, **EFFECT**: LeftShoeOn)

Op( **ACTION**: LeftSock, **EFFECT**: LeftSockOn)

### Initial plan

Plan (

**STEPS**: { **S<sub>1</sub>**: Op (ACTION: start),

**S<sub>2</sub>**: Op (ACTION: finish, PRECOND: RightShoeOn  $\wedge$  LeftShoeOn )},

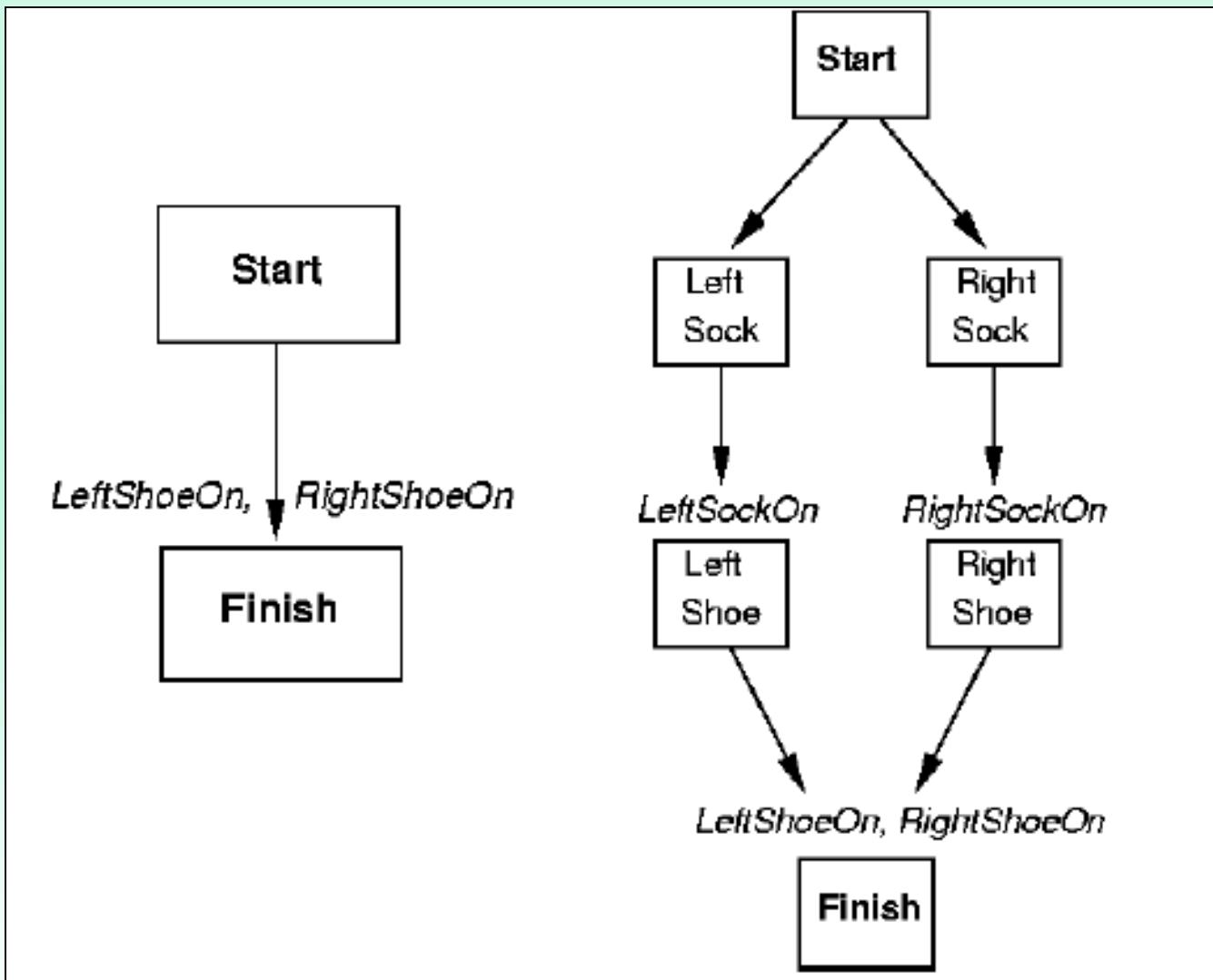
**ORDERINGS**: {S<sub>1</sub>  $\prec$  S<sub>2</sub>}, // S<sub>1</sub> should take place before S<sub>2</sub>

BINDINGS: {},

LINKS: {}

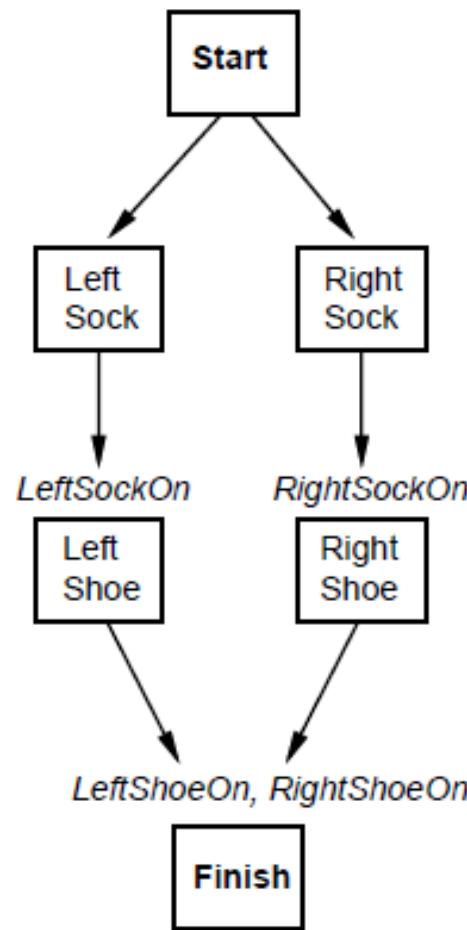
)

# Example – Shoes & Socks Problem



# Example – Shoes & Socks Problem

Partial-Order Plan:



Total-Order Plans:



**Figure 11.6** A partial-order plan for putting on shoes and socks, and the six corresponding linearizations into total-order plans.

# Hierarchical Planning

- **Hierarchical task network (HTN) planning** is an approach to automated **planning** in which the dependency among **actions** can be given in the form of hierarchically structured networks
- **Hierarchical Task Network**
  - At each “level”, only a small number of planning actions
  - Then descend to lower levels to “solve these” for real
  - At **higher “levels”**, the planner ignores “**internal effects**” of decompositions
  - But these have to be resolved at some (later) level

# HTN Example – House Construction

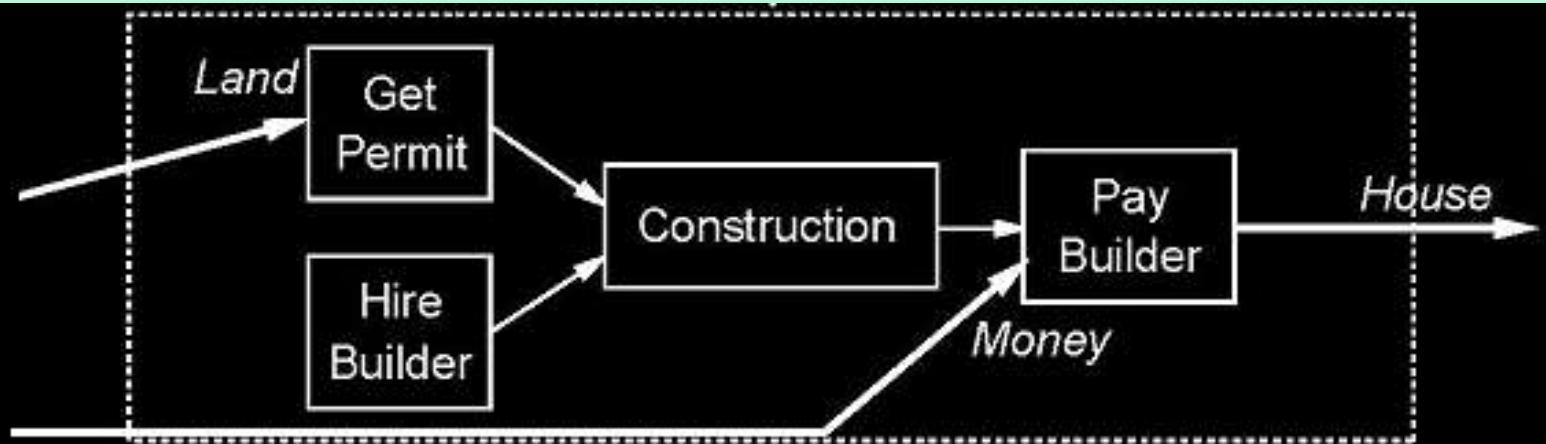
- **Construction Domain:**
  - **Actions:** // arrows show dependency
    - **Buy Land:** Money → Land
    - **Get Loan:** Good Credit → Money
    - **Get Permit:** Land → Permit
    - **Hire Builder:** → Contract
    - **Construction:** Permit ∧ Contract → House Built
    - **Pay Builder:** Money ∧ House Built → House ...



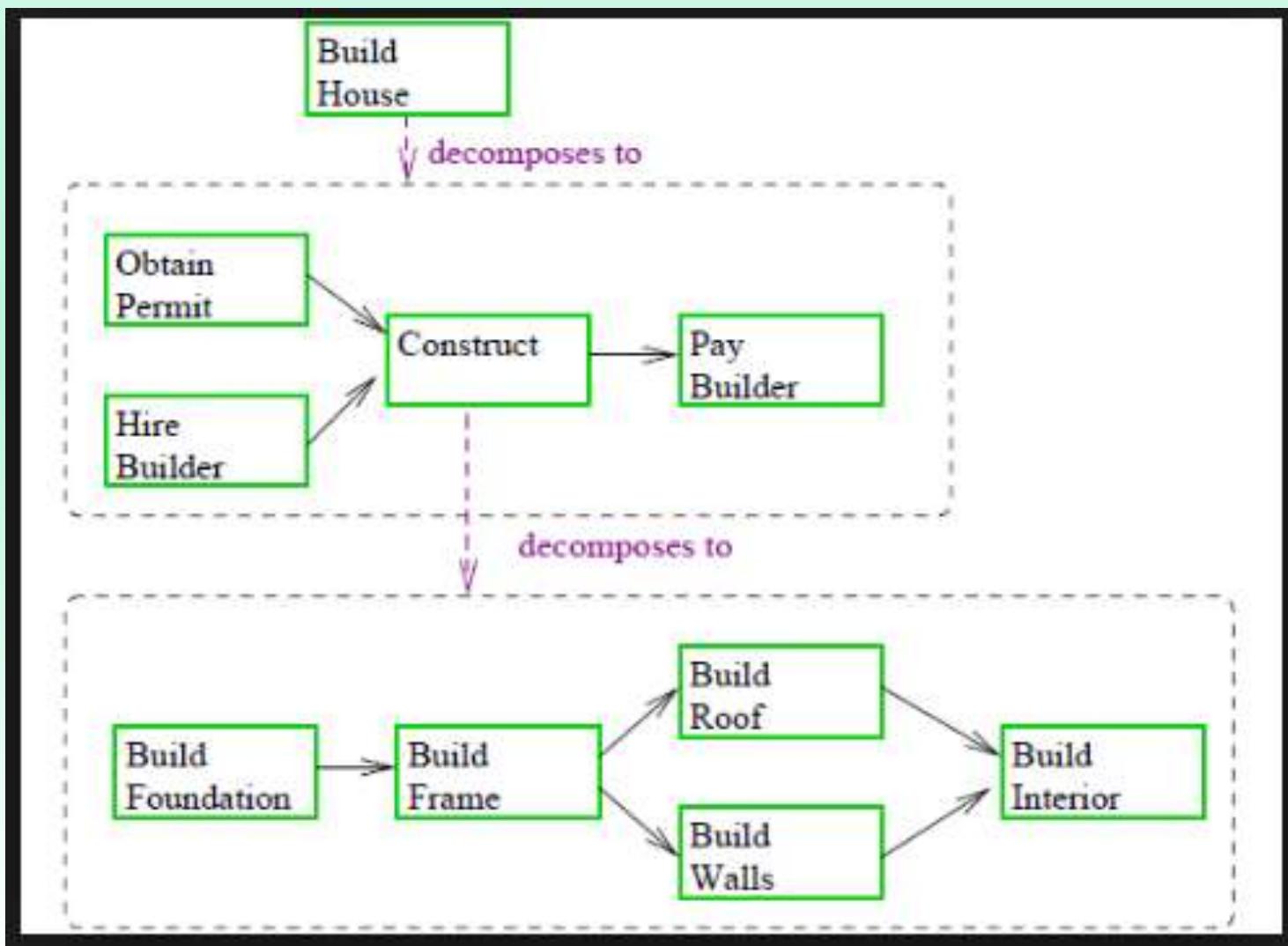
# HTN Example – House Construction

- Macro Action in Library:

- Build House:



# HTN Example – House Construction





# Uncertain Knowledge and Reasoning



*Dr. Pulak Sahoo*

## Associate Professor

Silicon Institute of Technology



# Topics



- Uncertainty-Introduction
- Uncertainty-Example
- Reasons for using probability
- Uncertain knowledge and reasoning
- Probability theory
- Axioms of probability
- Conditional probabilities
- Bayes' Rule, Theorem and its use



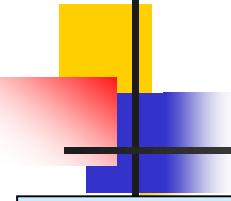
# Topics

- Inferences
- Bayesian networks
- Belief Network for the example
- Bayesian network semantics
- Incremental Network Construction
- Representing the joint probability distribution

# Uncertainty-Introduction



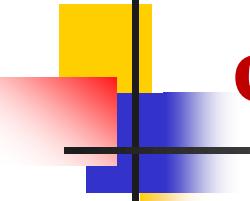
- **Agent** may need to handle **uncertainty** due to :
  - **Partially Observable** Env (*don't know the current state*)
  - **Non-deterministic** Env (*don't know result of a sequence of actions*)
- **Ex:** In up & down economy, whether to invest in stock market
- **Uncertainty** is expressed as **A → B** (*if A is true, then B is true. But not sure whether A is true*)
- **Uncertainty is dealt by:**
  - Keeping track of **belief state** (*all possible current states*)
  - Generating **contingency plan** (*for all possible eventualities*)
- **Drawbacks**
  - Very large & complex belief state representations & Contingency plan



# Uncertainty-Example

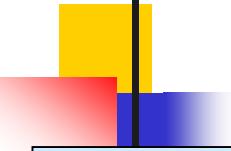


- Uncertainty is expressed as  $A \rightarrow B$
- Example of uncertainty: Diagnosis of a dental patient's toothache
- Consider below rule –
  - $\forall p \text{ symptom}(p, \text{Toothache}) \rightarrow \text{disease}(p, \text{cavity})$
  - *Rule is wrong – because not all patients have cavity*
- Fixing of the above rule –
  - $\forall p \text{ symptom}(p, \text{Toothache}) \rightarrow \text{disease}(p, \text{cavity}) \vee \text{disease}(p, \text{gum\_disease}) \vee \text{disease}(p, \text{abscess}) \dots$
  - *Exhaustive list of possible problems*



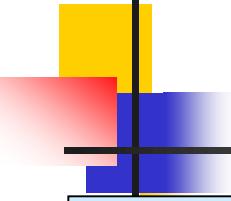
# Uncertain knowledge and reasoning can be dealt using:

- Probability theory
- Bayesian networks
- Certainty factors



# Reasons for using probability

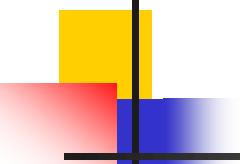
- Three main reasons
- Laziness
  - Too much work to find & list all antecedents (influencing factors).
- Theoretical ignorance
  - Medical science has no complete theory for this domain.
- Practical ignorance
  - Uncertain about a patient condition as **so many tests** are impossible to run.
- In such judgemental domains like law, business, design.., the agent's knowledge is only a **degree of belief**
- Which can be dealt with **probability theory**



# 1. Probability theory



- **Probability** provides a way of *summarizing the uncertainty*
- **Ex-** *The chance of cavity for toothache of a patient with history of gum disease is 40% or 0.4 probability*
- **Probability theory** → degree of **belief** or **plausibility** of a statement – a numerical measure in the range [0,1]



# Probability theory - representation

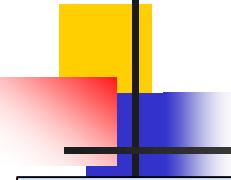
- **Unconditional or prior probability of A** – the degree of belief in A in the absence of any other information –  $P(A)$
- **A** – random variable
- **Probability distribution** –  $P(A)$ ,  $P(A,B)$

## Example-

$P(\text{Weather} = \text{Sunny}) = 0.1$     $P(\text{Weather} = \text{Rain}) = 0.7$

$P(\text{Weather} = \text{Snow}) = 0.2$    Weather – **random variable**

- **P(Weather)** =  $(0.1, 0.7, 0.2)$  – probability distribution
- **Conditional probability** – posterior – The agent has obtained some evidence B (is true) for A -  $P(A|B)$  – Probability of A given B
- **Example-**  $P(\text{Cavity} | \text{Toothache}) = 0.4$



# Axioms of probability

- *The measure of the occurrence of an event (random variable) A* – a function  $P:S \rightarrow R$  satisfying the **axioms**:
  - $0 \leq P(A) \leq 1$
  - $P(S) = 1$  ( or  $P(\text{true}) = 1$  and  $P(\text{false}) = 0$ )
  - $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$
  - $P(A \vee \sim A) = P(A) + P(\cancel{\sim A}) - \cancel{P(\text{false})} = P(\text{true})$
  - $P(\sim A) = 1 - P(A)$
- A & B mutually exclusive  $\rightarrow P(A \vee B) = P(A) + P(B)$  //  $P(A \wedge B)$  is null
- $P(e_1 \vee e_2 \vee e_3 \vee \dots e_n) = P(e_1) + P(e_2) + P(e_3) + \dots + P(e_n)$

The probability of a proposition **a** = the sum of the probabilities of the **atomic events** ( $e(a)$ ) in which **a** holds

$$P(a) = \sum_{e_i \in e(a)} P(e_i)$$

# Uncertain Knowledge and Reasoning



*Dr. Pulak Sahoo*

## Associate Professor

Silicon Institute of Technology



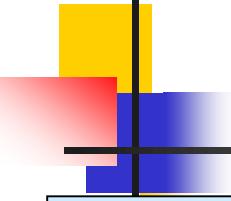
# Topics

- Uncertainty-Introduction
- Uncertainty-Example
- Reasons for using probability
- Uncertain knowledge and reasoning
- Probability theory
- Axioms of probability
- Conditional probabilities
- Bayes' Rule, Theorem and its use



# Topics

- Inferences
- Bayesian networks
- Belief Network for the example
- Bayesian network semantics
- Incremental Network Construction
- Representing the joint probability distribution



# Conditional probabilities

- Conditional probabilities (for A) can be defined in terms of unconditional probabilities (for B)
- *The condition probability of the occurrence of A if event B actually occurs ( or given B (is true))*

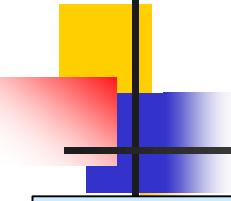
- $P(A|B) = P(A \wedge B) / P(B)$

This can be written also as:

- $P(A \wedge B) = P(A|B) * P(B)$

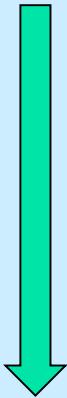
**Probability distributions** (for different outcomes a1, b1, b2...)

- $P(A=a1 \wedge B=b1) = P(A=a1|B=b1) * P(B=b1)$
- $P(A=a1 \wedge B=b2) = P(A=a1|B=b2) * P(B=b2) ....$
- $P(X,Y) = P(X|Y)*P(Y)$



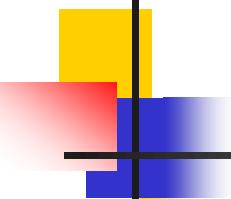
# Bayes' Rule, Theorem and its use

- $P(A \wedge B) = P(A|B) * P(B)$  &  $P(A \wedge B) = P(B|A) * P(A)$



- **Bayes Theorem**

- $P(B|A) = P(A | B) * P(B) / P(A)$  //  $P(B|A)*P(A) = P(A|B)*P(B)$



# Inferences

Probability distribution

$\mathbf{P}(\text{Cavity}, \text{Toothache})$

	Toothache	$\sim$ Toothache
Cavity	0.04	0.06
$\sim$ Cavity	0.01	0.89

// cavity with/wo toothache

// no cavity

- $P(\text{Cavity}) = 0.04 + 0.06 = 0.1$
- $P(\text{Cavity} \vee \text{Toothache}) = 0.04 + 0.01 + 0.06 = 0.11$
- $P(\text{Cavity} \mid \text{Toothache}) = P(\text{Cavity} \wedge \text{Toothache}) / P(\text{Toothache}) = 0.04 / 0.05$

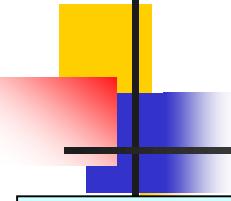
# Inferences

	Toothache		$\sim$ Toothache	
	Catch	$\sim$ Catch	Catch	$\sim$ Catch
Cavity	<b>0.108</b>	<b>0.012</b>	0.072	0.008
$\sim$ Cavity	<b>0.016</b>	<b>0.064</b>	0.144	0.576

## Probability distributions

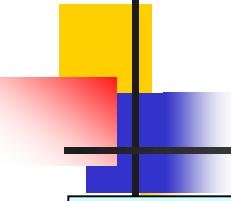
$\mathbf{P}(\text{Cavity}, \text{Toothache}, \text{Catch})$

- $\mathbf{P}(\text{Cavity}) = 0.108 + 0.012 + 0.72 + 0.008 = 0.2$
- $\mathbf{P}(\text{Cavity} \vee \text{Toothache}) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016$   
■  $+ 0.064 = 0.28$
- $\mathbf{P}(\text{Cavity} | \text{Toothache}) = \mathbf{P}(\text{Cavity} \wedge \text{Toothache}) / \mathbf{P}(\text{Toothache})$   
 $= [\mathbf{P}(\text{Cavity} \wedge \text{Toothache} \wedge \text{Catch}) + \mathbf{P}(\text{Cavity} \wedge \text{Toothache} \wedge \sim \text{Catch})]$   
 $* / \mathbf{P}(\text{Toothache})$   
 $= (0.108 + 0.012) / (0.108 + 0.012 + 0.016 + 0.064) = 0.12 / 0.2 = 0.6$



# Bayesian networks

- Represents **dependencies** among **random variables**
- Gives a short specification of **conditional probability distribution**
- Many **random variables** are **conditionally independent**
- **Simplifies** computations
- **Graphical** representation
- **DAG – Causal relationships** among **random variables**
- Allows **inferences** based on the network structure

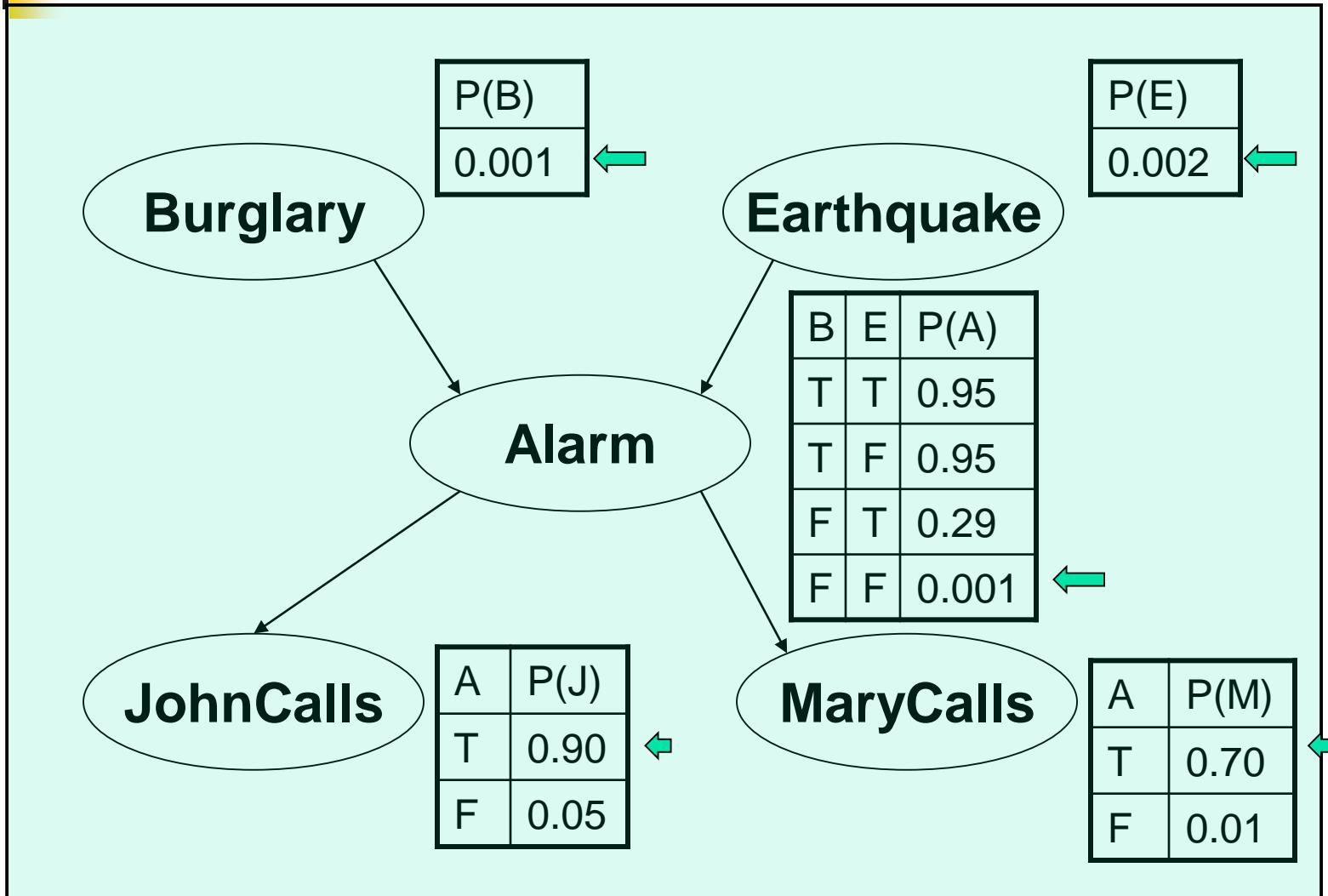


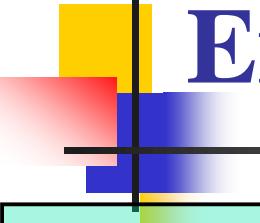
# Bayesian networks

A **BN** is a **DAG** in which each **node** is annotated with quantitative probability information:

- **Nodes** represent **random variables** (discrete or continuous)
- **Directed links  $X \rightarrow Y$ :** X has a direct influence on Y, X is said to be a parent of Y
- Each node X has an associated **conditional probability table,  $P(X_i | Parents(X_i))$**  that quantify the effects of the parents on the node
- **Example:** Weather, Cavity, Toothache, Catch
  - Weather, Cavity → Toothache, Cavity → Catch

# Belief Network for the Burglar alarm example

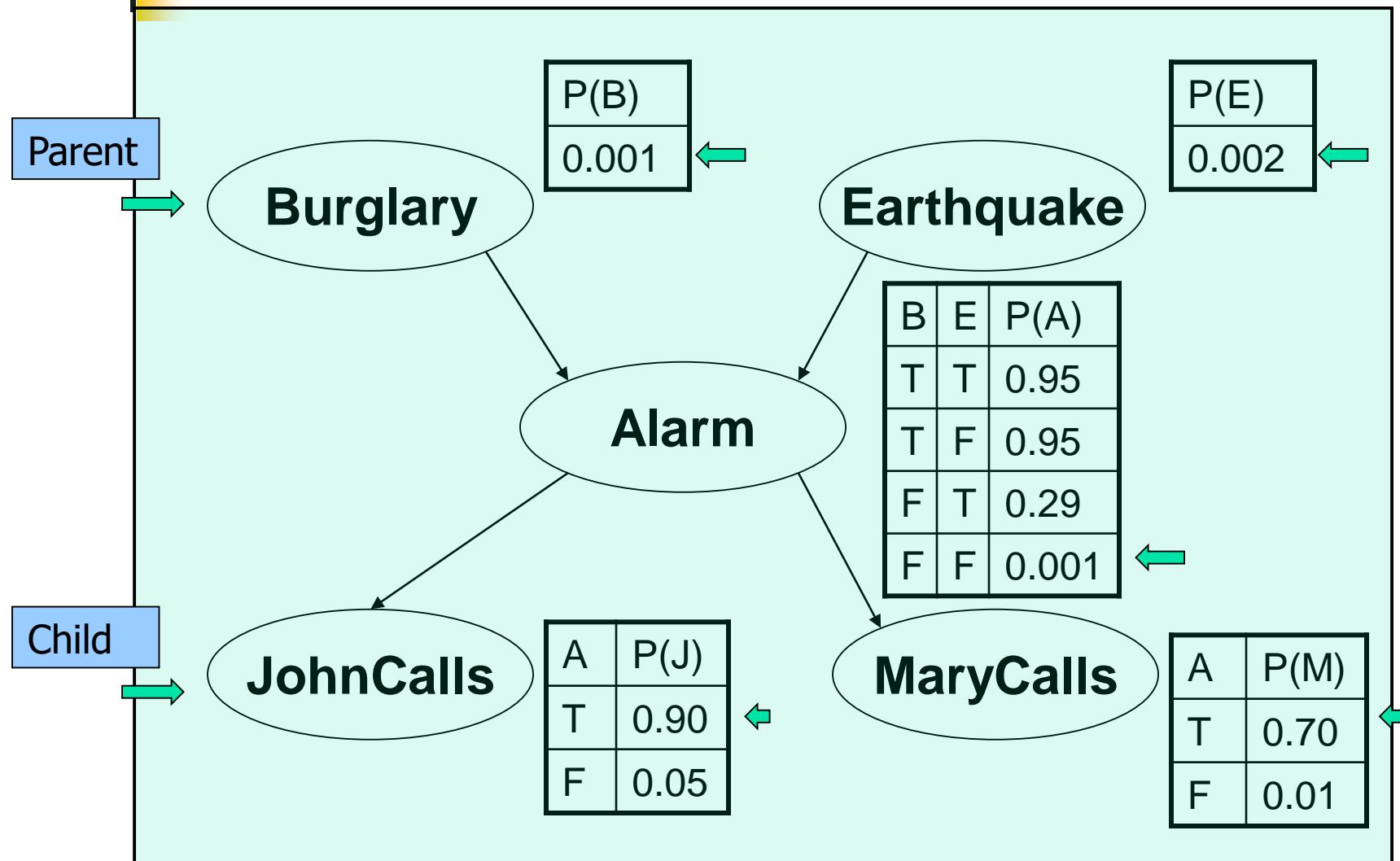




# Example: Burglar alarm at your home

- **Burglar alarm at your home**
  - Fairly reliable at detecting a burglary
  - Also responds on occasion to minor earthquakes
- **Two neighbors** who, on hearing the alarm **calls you at office**
  - **John** always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm & calls
  - **Mary** likes loud music and sometimes misses the alarm altogether

# Belief Network for the Burglar alarm example



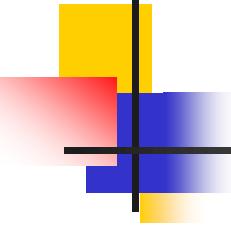
# Representing the joint probability distribution

- A generic entry in the joint probability distribution  $P(x_1, \dots, x_n)$  is given by:

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | Parents(X_i))$$

- Probability** of the event that “**the alarm has sounded but neither a burglary nor an earthquake has occurred, and both Mary and John call**”:

$$\begin{aligned} & P(J \wedge M \wedge A \wedge \neg B \wedge \neg E) \\ &= P(J | A) \ P(M | A) \ P(A | \neg B \wedge \neg E) \ P(\neg B) \ P(\neg E) \\ &= 0.9 \times 0.7 \times 0.001 \times 0.999 \times 0.998 = 0.00062 \end{aligned}$$



# Bayesian network semantics

- A) Represent a probability distribution (table)
- B) Specify conditional independence – **build the network**
- A) Each value of the probability distribution can be computed as:

$$\begin{aligned} P(X_1=x_1 \wedge \dots \wedge X_n=x_n) &= P(x_1, \dots, x_n) = \\ &\prod_{i=1,n} P(x_i \mid \text{Parents}(x_i)) \end{aligned}$$

- where **Parents(xi)** represent the specific values of Parents(Xi)

# Incremental Network Construction

1. Choose the set of relevant variables  $X_i$  that describe the domain
2. Choose an ordering for the variables (very important step)
3. While there are variables left:
  - a) Pick a variable  $X$  & add a node to the network for it
  - b) Set **Parents(X)** to some minimal set of nodes already in the net such that the conditional independence property is satisfied
  - c) Define the conditional probability table for  $X$

**THE END**

# Learning



Dr. Pulak Sahoo  
Associate Professor  
Silicon Institute of Technology

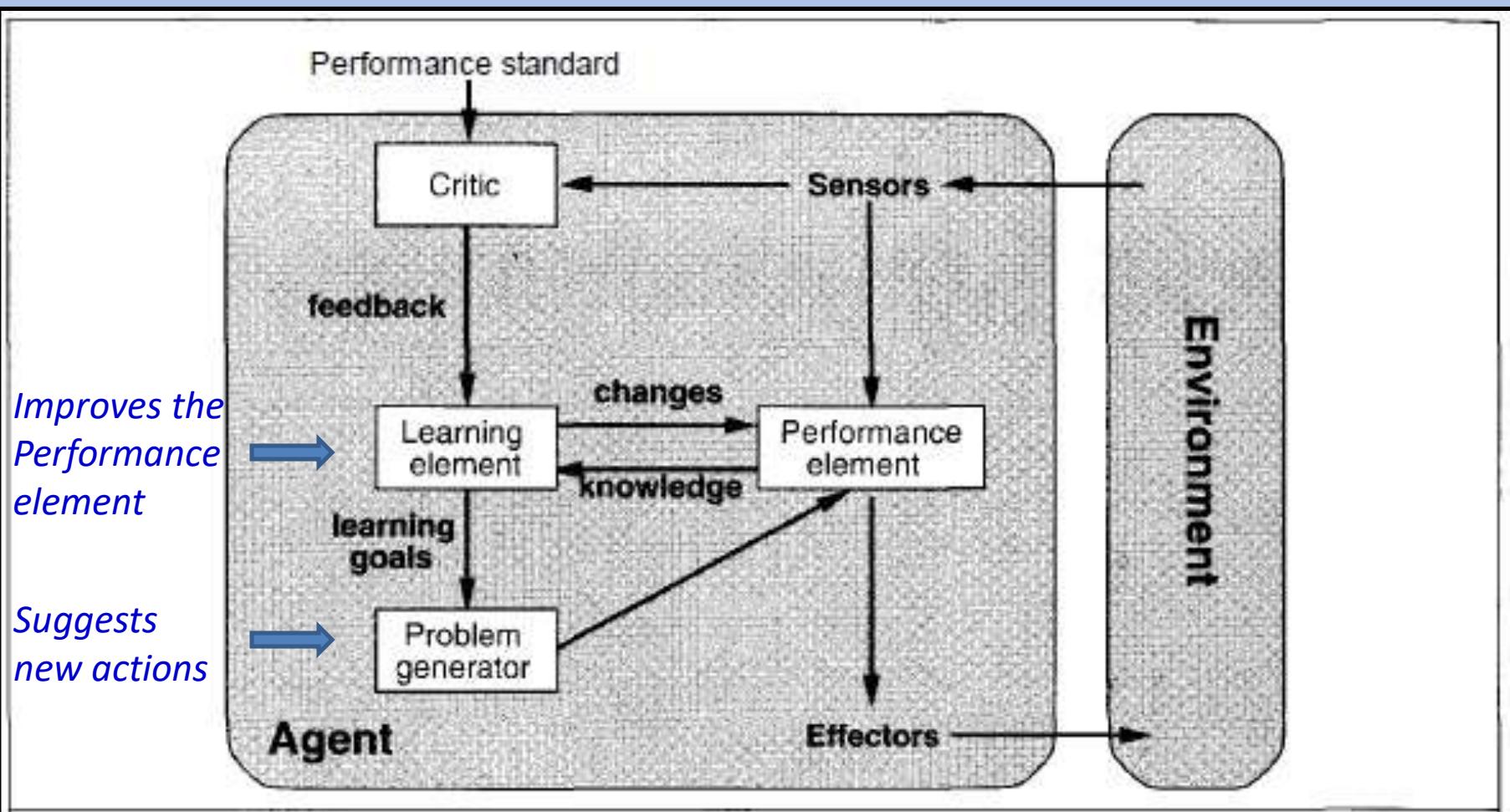
# Contents

- Introduction
- General models of Learning agents
  - Four elements of learning agent
- Paradigms of learning
- Supervised & Unsupervised Learning
- Inductive Learning
  - Decision Tree Induction
- Artificial Neural Networks
  - Perceptron
- Reinforcement Learning

# Introduction

- The idea behind learning:
  - Percepts should be used not only for acting, but also for improving the agent's ability to act in the future
- Learning takes place as a result of :
  - Interaction between the agent & the world
  - From observation by the agent of its own decision-making processes

# GENERAL MODEL OF LEARNING AGENTS



# Four elements of learning agent

## 1. Performance element

- It takes the percepts as input & decides on external actions

## 2. Critic

- Tells the learning element how well the agent is doing

## 3. Learning element

- Takes feedback on how the agent is doing from Critic
- Determines how the performance element can do better in future

## 4. Problem generator

- Suggests actions that will lead to new & informative experiences

# Paradigms of learning

## 1. Supervised Learning

- Uses a labeled training set to teach the **model** to yield the **desired output**.
- This training dataset includes **inputs & correct outputs (labeled)** by a friendly teacher) allowing the model to learn over time

## 2. Unsupervised Learning

- The training dataset are not labeled (*output not provided*)
- The agent has to discover the patterns within the dataset on its own

## 3. Inductive Learning

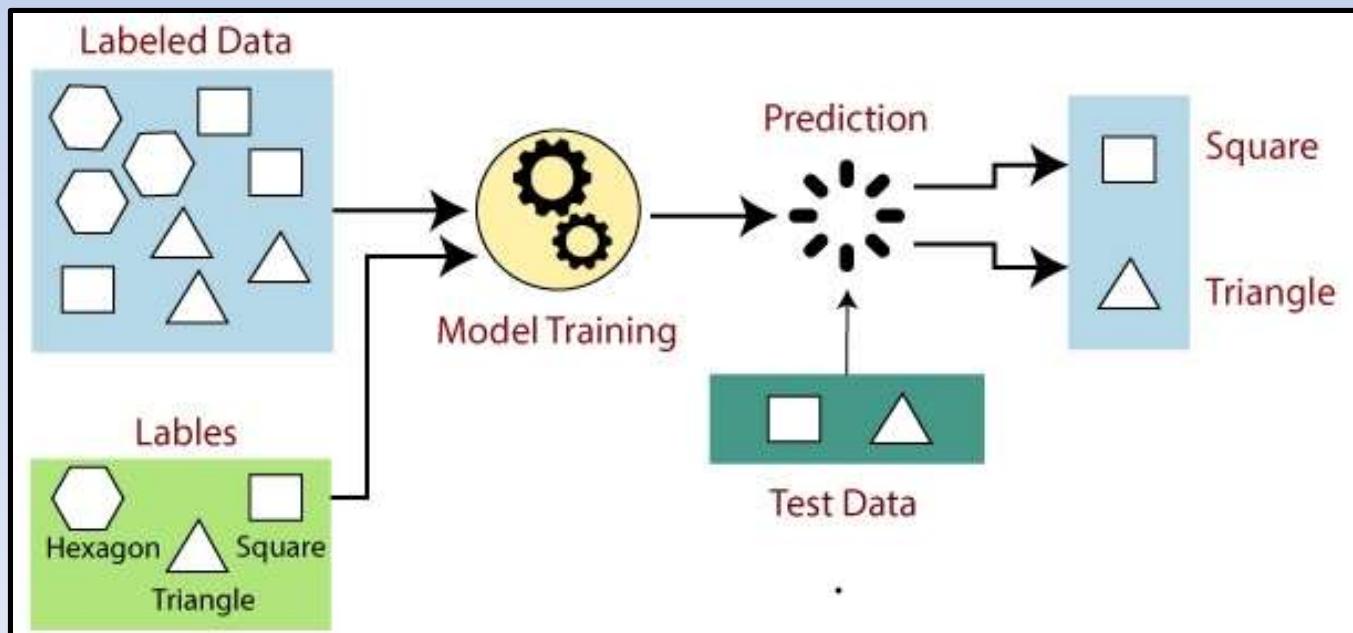
- Is used for generating a set of classification rules which are of the form “IF-THEN”
- These **rules** are **produced** at **each iteration** on the **dataset** & appended to the **existing set of rules**

## 4. Reinforcement Learning

- Helps in taking suitable action to maximize reward in a particular situation
- The **agent** receives some feedbacks (*positive or negative*) on its **action**, but is not told what is the correct action

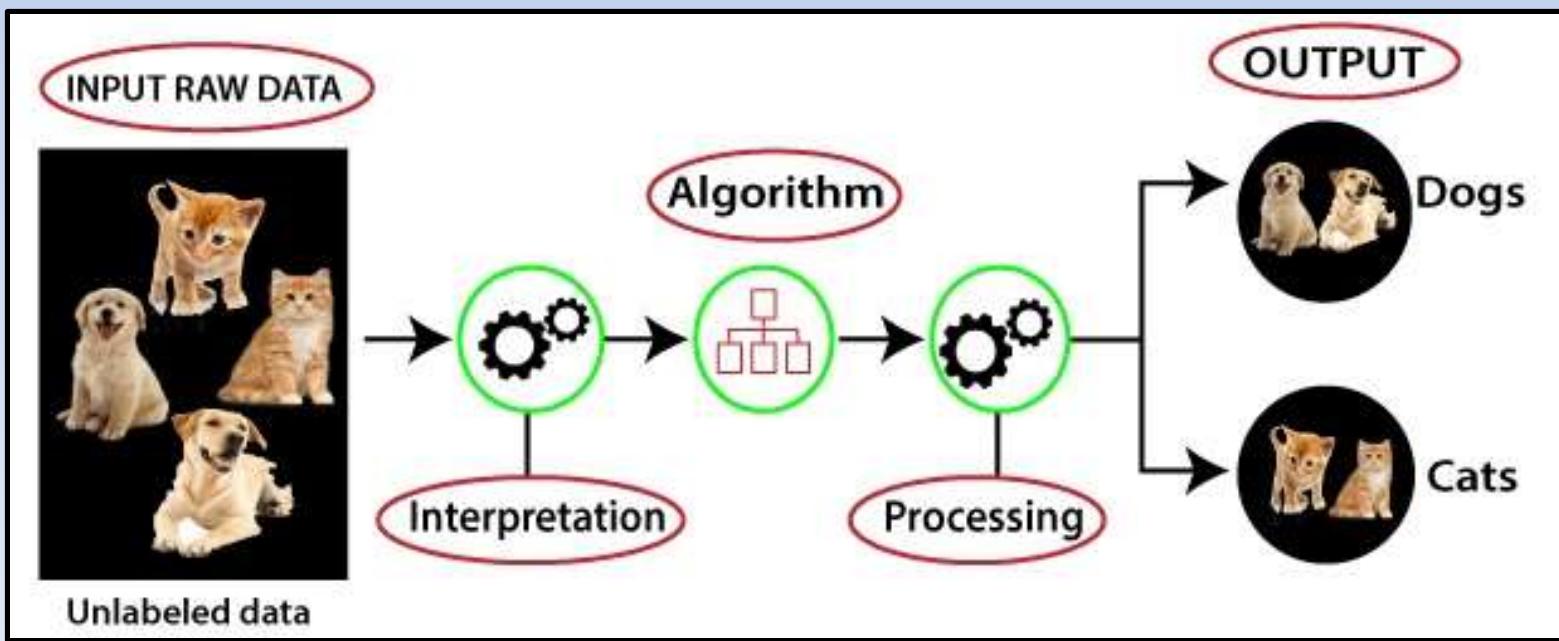
# 1. Supervised learning

- **Supervised learning** method involves the training of the system or model where the output labels are provided
- The model is trained with guidance of already known outputs/labels
  - to facilitate the prediction of outputs/labels for future unknown instances (*Test Data*)
- **SL** is used in Classification & Regression problems



## 2. Unsupervised learning

- This model does not involve the target output/label (*unlabeled data*)
- No training is provided to the **system/model**
- The system has to learn by its own through analysis of the structural characteristics in the input patterns.
- **USL** is used in Clustering problems

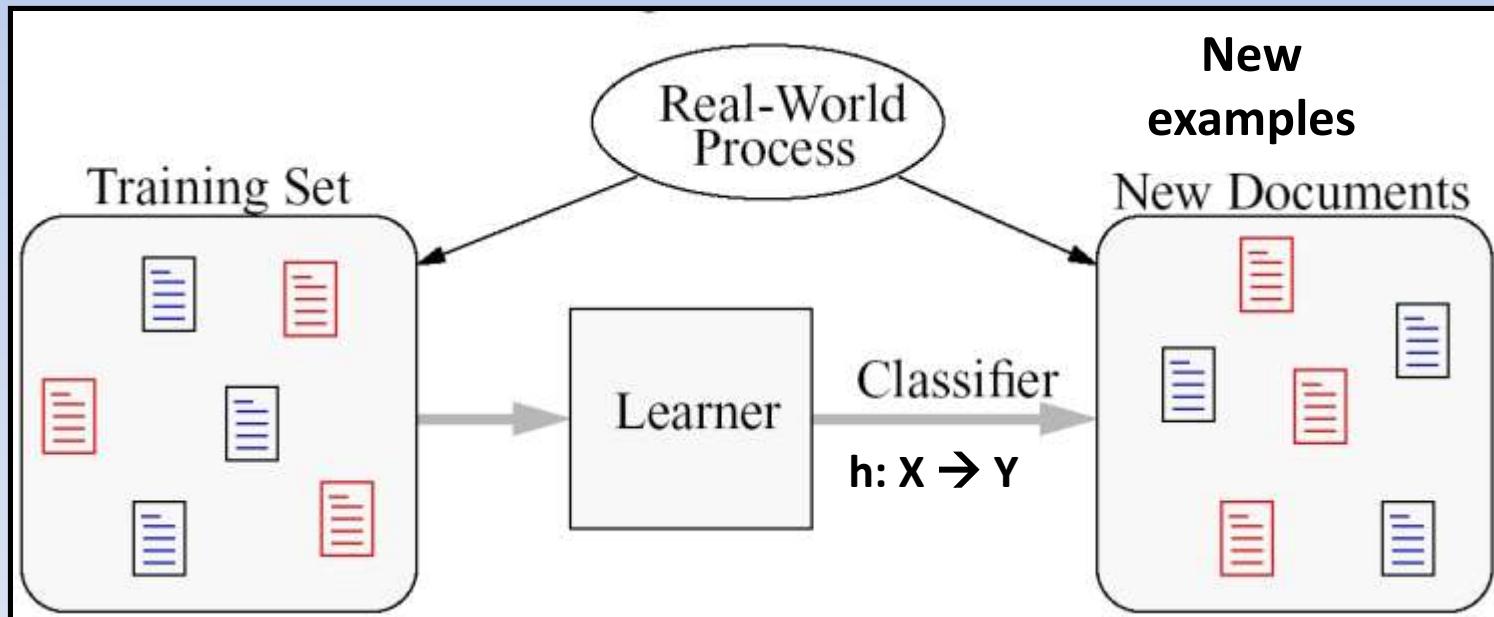


# Differences - Supervised & Unsupervised Learning

1. **SL technique** deals with the **labeled data** where the **output** data patterns are known to the system. But, the **USL technique** works with **unlabeled data**
2. The **SL method** is less complex while **USL method** is more complicated
3. The outcome of **SL technique** is more accurate & reliable. **USL technique** generates moderately reliable results
4. **Classification & Regression** are the types of problems solved under the **SL method**. The **USL method** includes **clustering** problems

# 3. Inductive Learning

- Is an **iterative learning algorithm**:
  - Used for generating a set of **classification rule** of the form “**IF-THEN**”
  - For each set of examples, it **produces rules** at each iteration & appends to the **existing set of rules**



# Inductive Learning Example – Restaurant selection

Food (3)	Can Chat (2)	Fast Service (2)	Price (3)	Bar avl (2)	Should go?
great	yes	yes	normal	no	yes
great	no	yes	normal	no	yes
mediocre	yes	no	high	no	no
great	yes	yes	normal	yes	yes

**Instance Space:** Each row is an instance & each column is a feature (attribute)

**Label:** Last column is the label (or output)

**Training Data :** Set of instances labeled with target values

**Hypothesis Space:** Set of all classification rules  $h_i$  that we allow

# Learning



Dr. Pulak Sahoo  
Associate Professor  
Silicon Institute of Technology

# Contents

- Introduction
- General models of Learning agents
  - Four elements of learning agent
- Paradigms of learning
- Supervised & Unsupervised Learning
- Inductive Learning
  - Decision Tree Induction
- Artificial Neural Networks
  - Perceptron
- Reinforcement Learning

# Classification by Decision Tree Induction

- DT induction is the learning using decision trees created from class-labeled training tuples (*training instance with output value/label provided*)
- A DT is a **flowchart-like tree structure**, where each internal node denotes a test on an attribute (eg:  $age > 18$ ,  $income < 25000\ldots$ )
- *The topmost node in a tree is the root node*
- Each branch represents an outcome of the test
- Each leaf node holds a class label

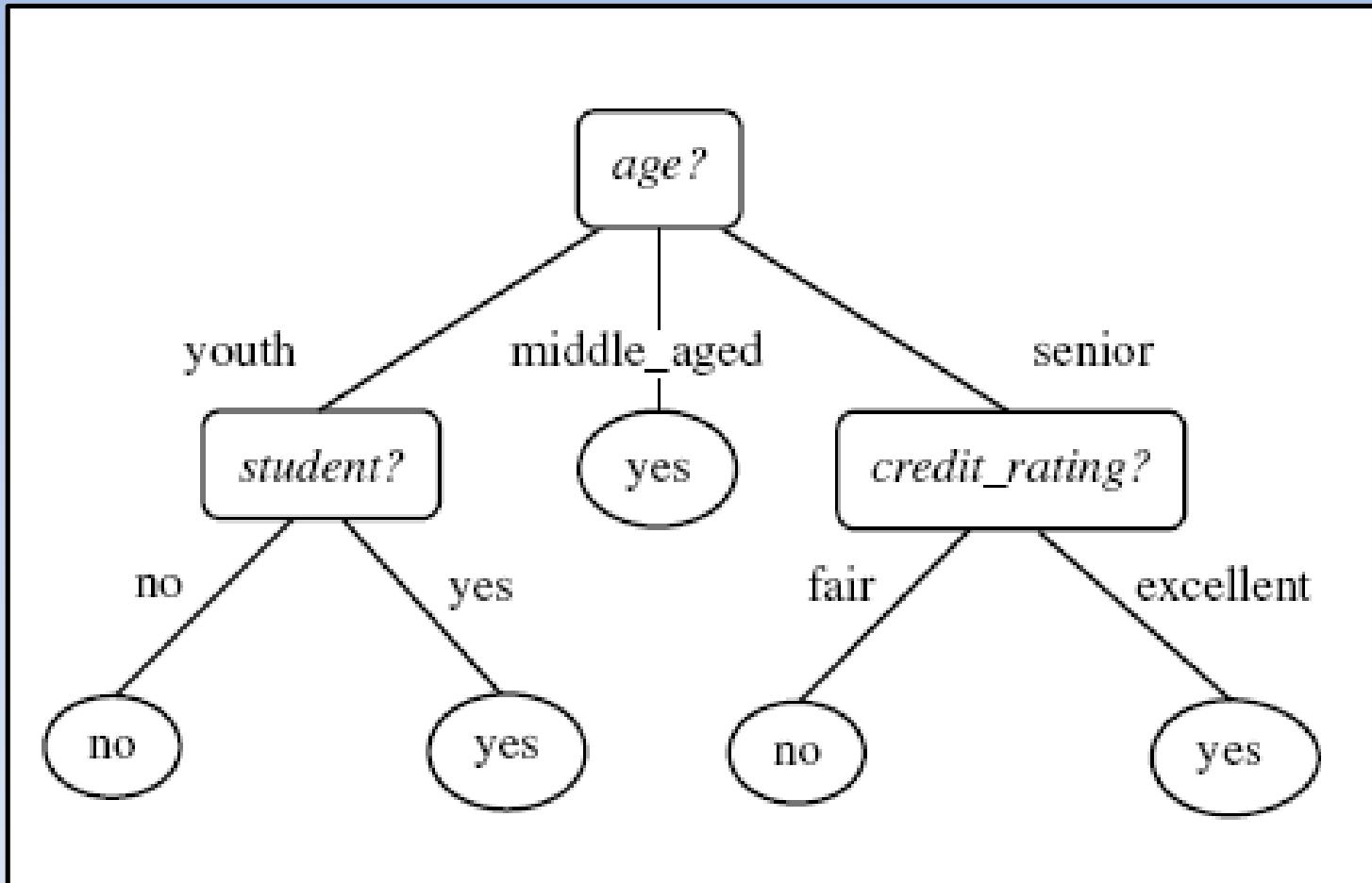
# Example: Computer Buying

RID	age	income	student	credit_rating	Class: buys_computer
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes
6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

Labeled  
Training  
Tuple



# Example: Finding Customers likely to buy a computer



# *How are decision trees used for classification?*

- Given a tuple or instance,  $X$ , the attribute values of the tuple are tested using the decision tree
- A path is traced from the root to a leaf node, which holds the class or label prediction for that tuple

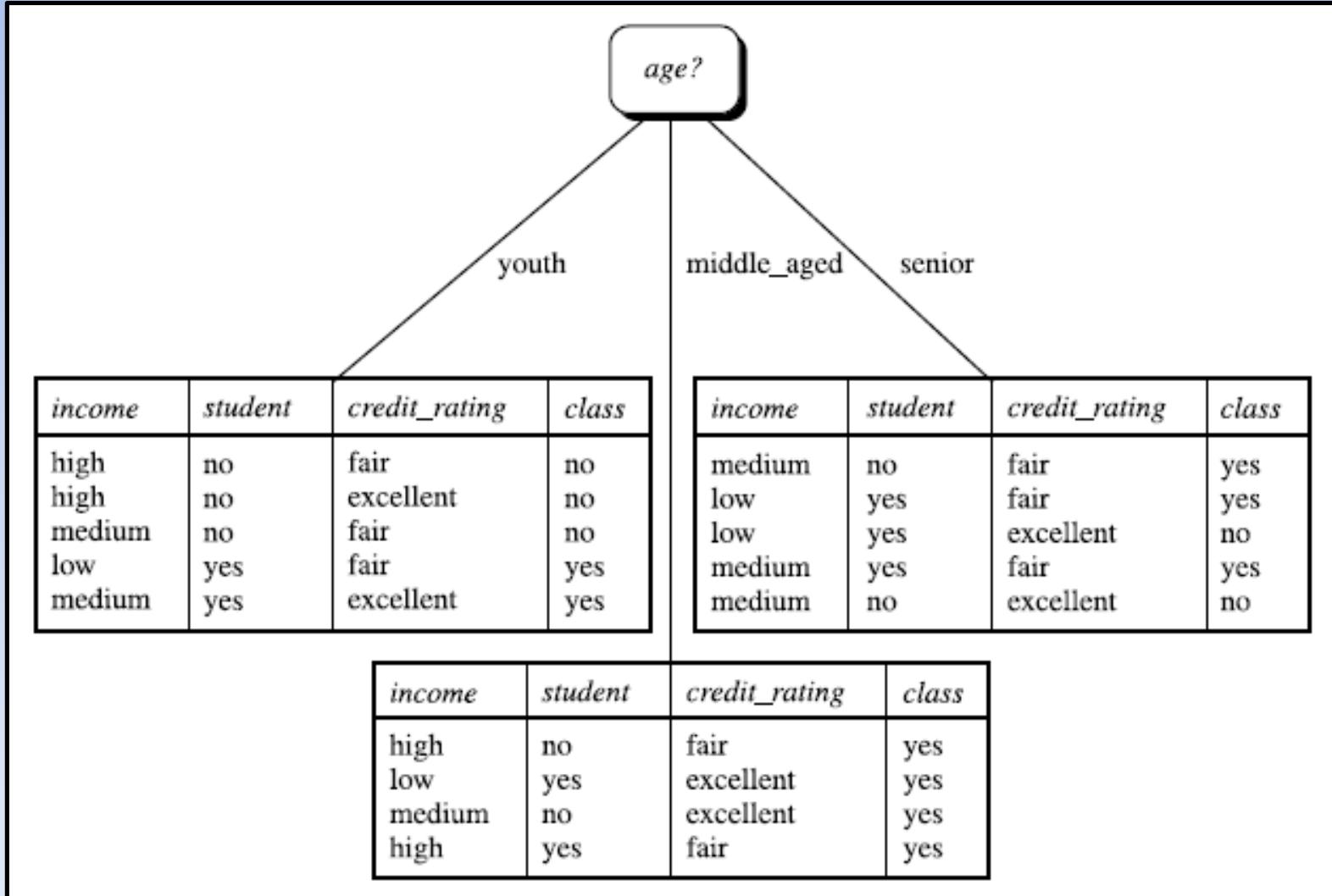
# *Why DT classifiers so popular?*

- The **construction** of DT classifiers does not require any domain knowledge or parameter setting
- Can handle high dimensional data
- The learning & classification steps of DT induction are simple & fast
- Have good accuracy
- Simple to understand & interpret
- Able to handle both numerical & categorical data (*salary figures or low/medium/high scales*)

# Example: Computer Buying

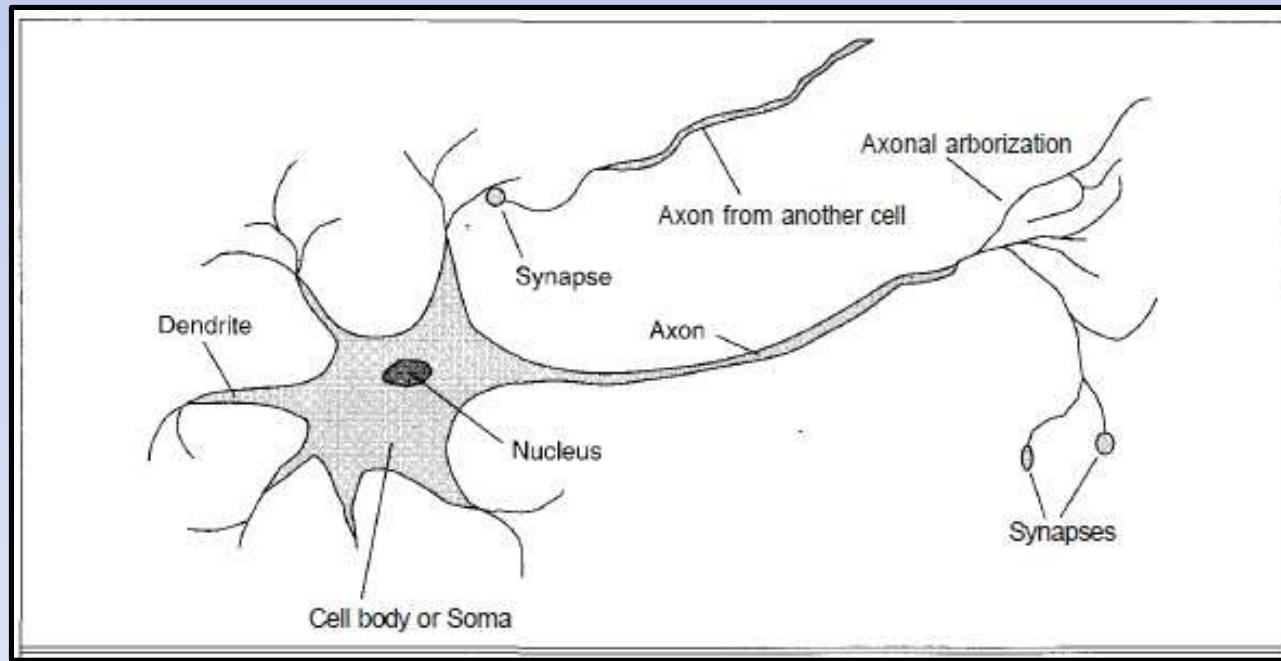
RID	age	income	student	credit_rating	Class: buys_computer
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes
6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

The attribute age has the highest information gain (*has more influence on buying decision*) & therefore becomes the splitting attribute at the root node of the DT. Branches are grown for each outcome of age.



# Artificial Neural Networks (ANN)

- An **artificial neural network (ANN)** is an AI based computing system designed to **simulate** the way the **human brain** analyzes and processes information
- It is used to solve **complex problems** the **same way** human's do

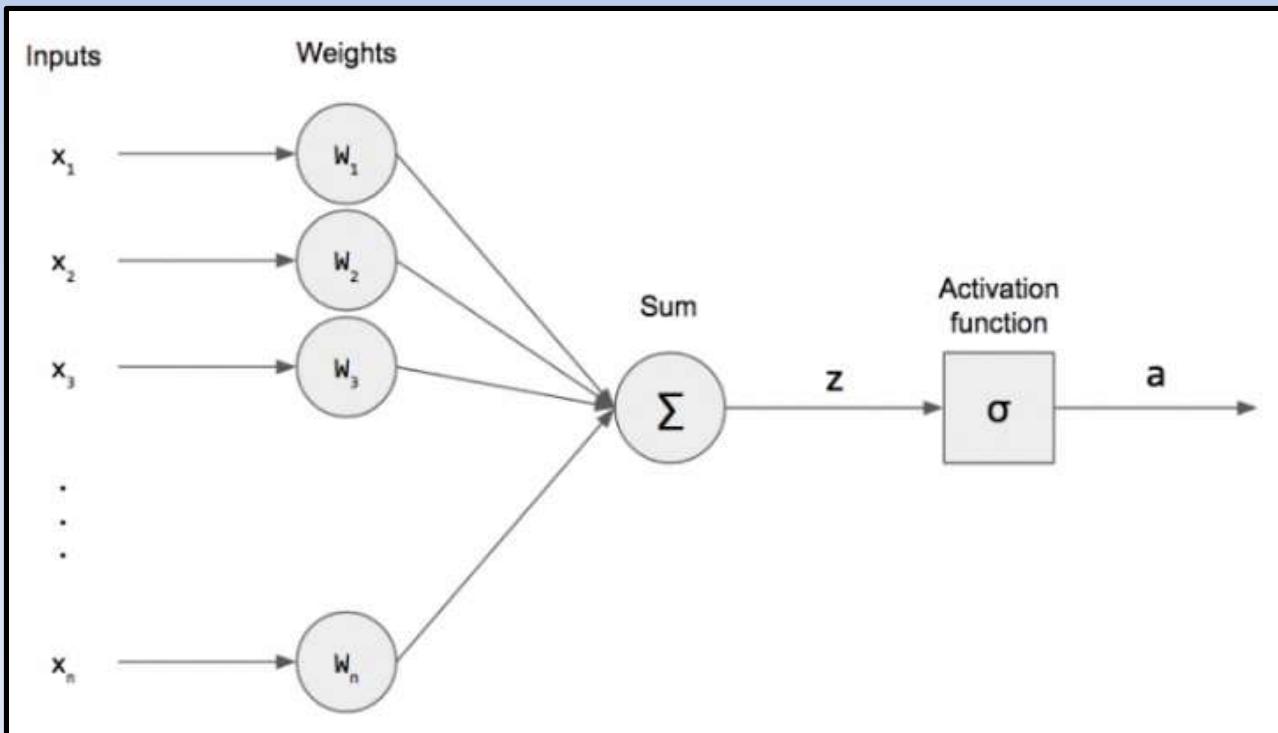


- **ANN** consists of a pool of **processing units (neurons)** which communicate by sending **signals** to each other over a large number of **weighted connections**
- In **Testing stage**, each **unit receives input** from **neighbors** or **external sources** & uses this to **compute** an **output signal** which is propagated to other units
- In **Learning stage**, the **adjustment of the weights** takes place to yield the best result
- The system is **inherently parallel** as many units can carry out their computations at the same time

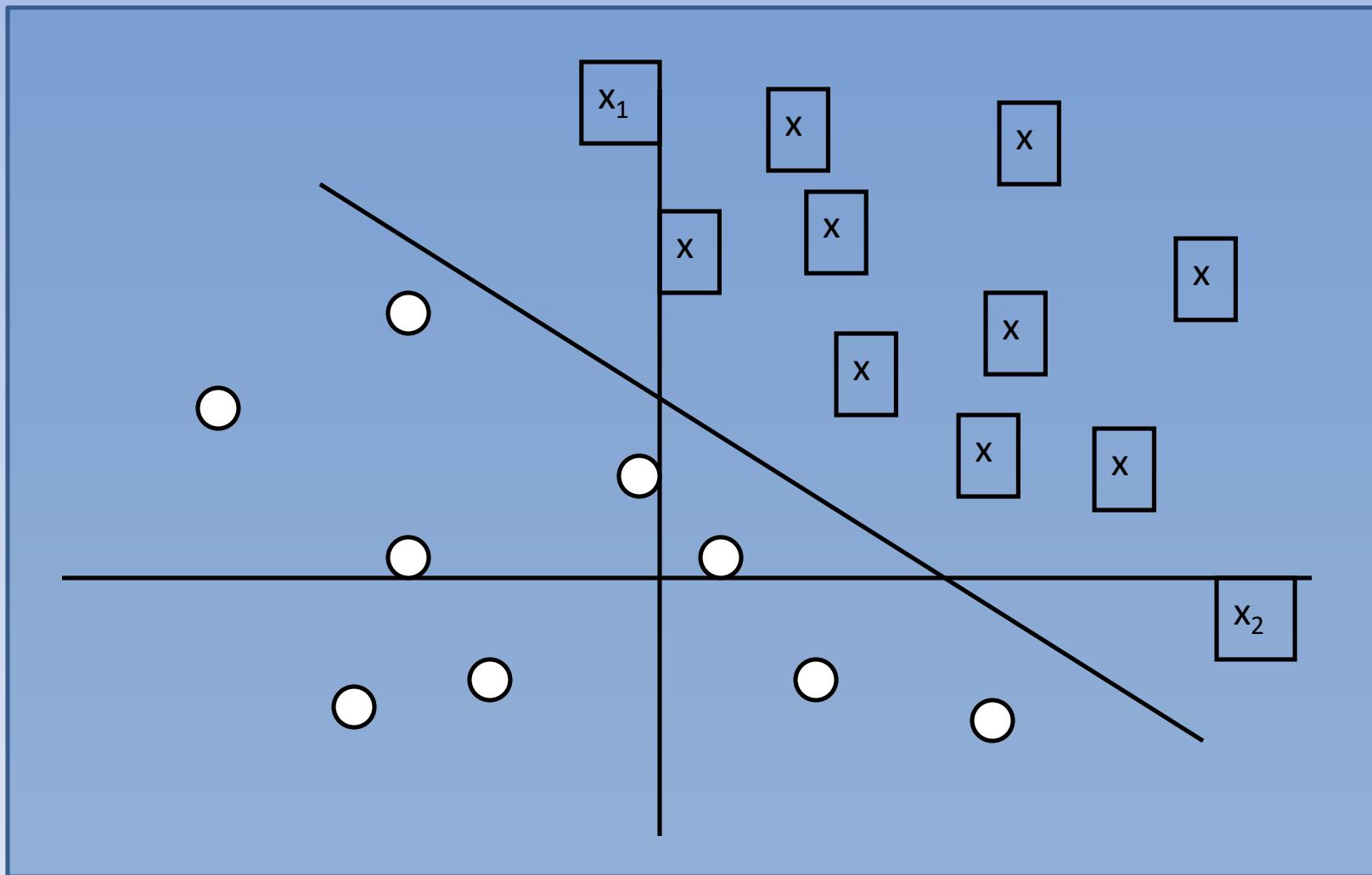
# Perceptron

The **perceptron** is the **simplest form of ANN** that:

- Is a **single layer feed-forward network** of neurons
- Takes the input signals  $x_1, x_2, \dots, x_n$  connected with a **weighting factor  $w_i$**
- Computes a **weighted sum  $z$**  of those inputs
- Then passes it through a **threshold/activation function  $\sigma$**  & outputs the result **a**

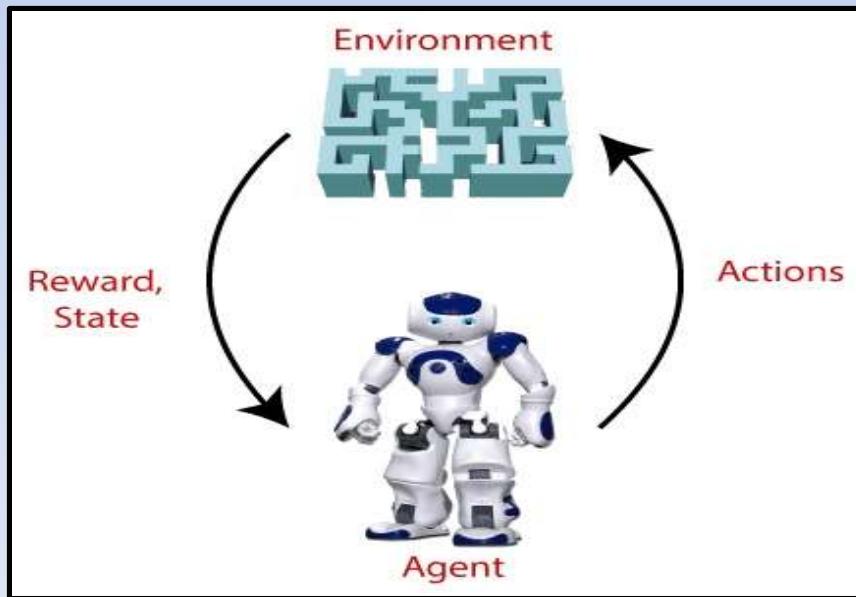


# Perceptron (used for classification problems)

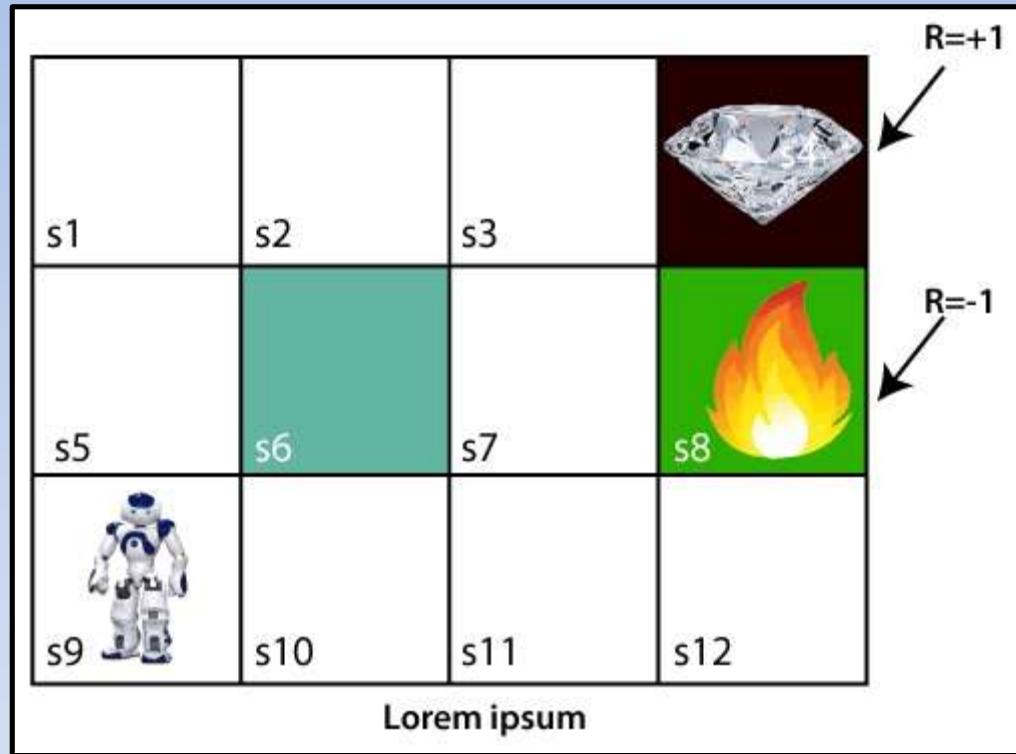


# 4. Reinforcement Learning

- **RL** is a feedback-based learning technique in which an agent learns to behave in an env. by performing the actions & seeing results of actions
- The agent gets **positive feedback** for each **good action** & gets **negative feedback** or penalty for each **bad action**
- The agent **learns automatically** using feedbacks & experience without any labeled data
- It solves problem with long-term goals, such as game-playing, robotics, etc.



# Reinforcement Learning - Example



**The End!!!**