



ARTIFICIAL INTELLIGENCE

PREPARED BY

-ANOOJA JOY

ARTIFICIAL INTELLIGENCE INTRODUCTION

- AI Applications
- What is AI ?
- Intelligent Agents
- Agent Environments
- Problem Formulation



COURSE OBJECTIVES



To introduce AI and key paradigms of AI



Understand core techniques and algorithms of AI: Deep Learning, Machine Learning, Natural Language Processing, Reinforcement Learning, Q-learning, Intelligent Agents, Various Search Algorithms



Understand the basic of knowledge presentation



Gain knowledge of problem solving techniques



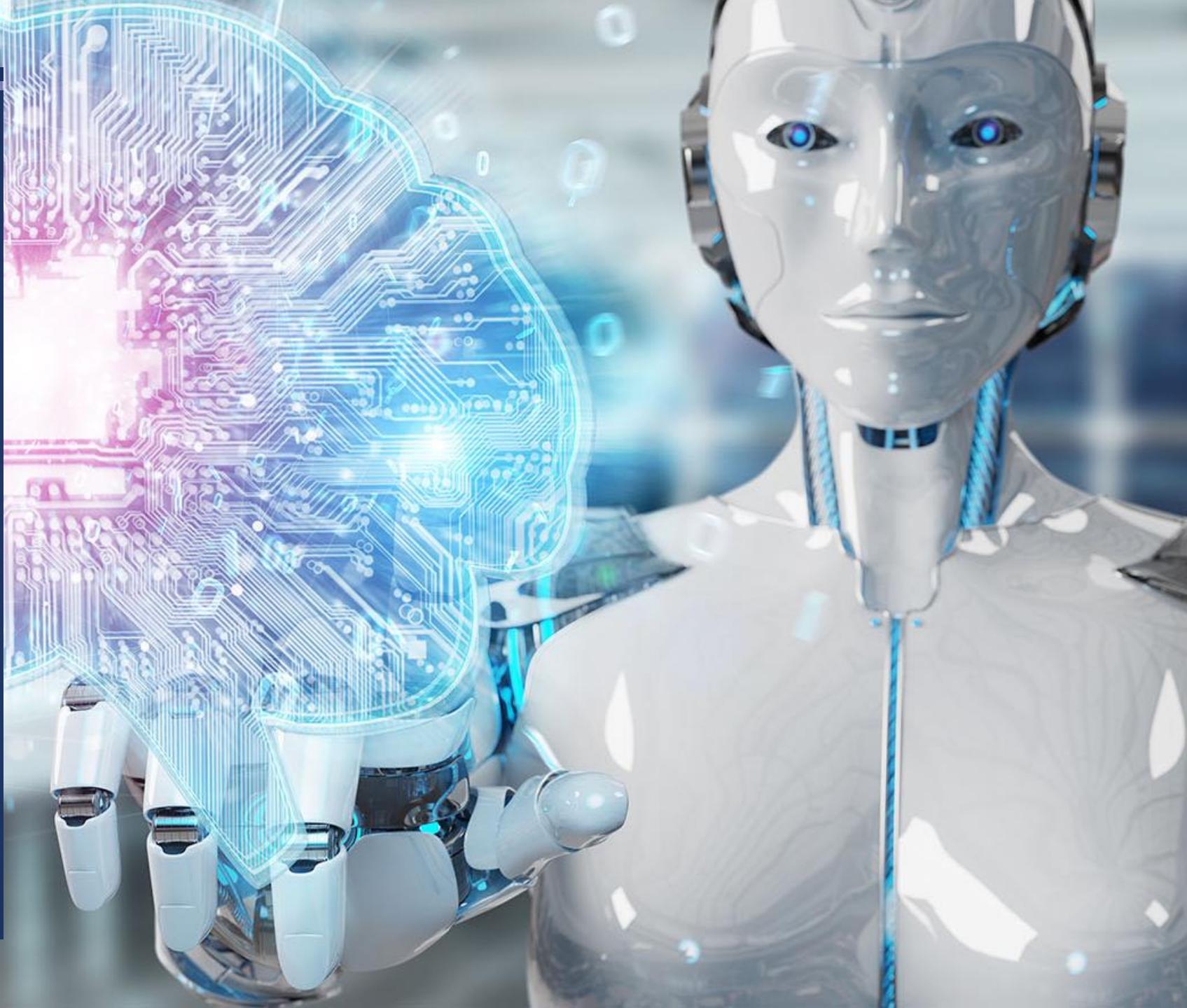
How to build an intelligent system

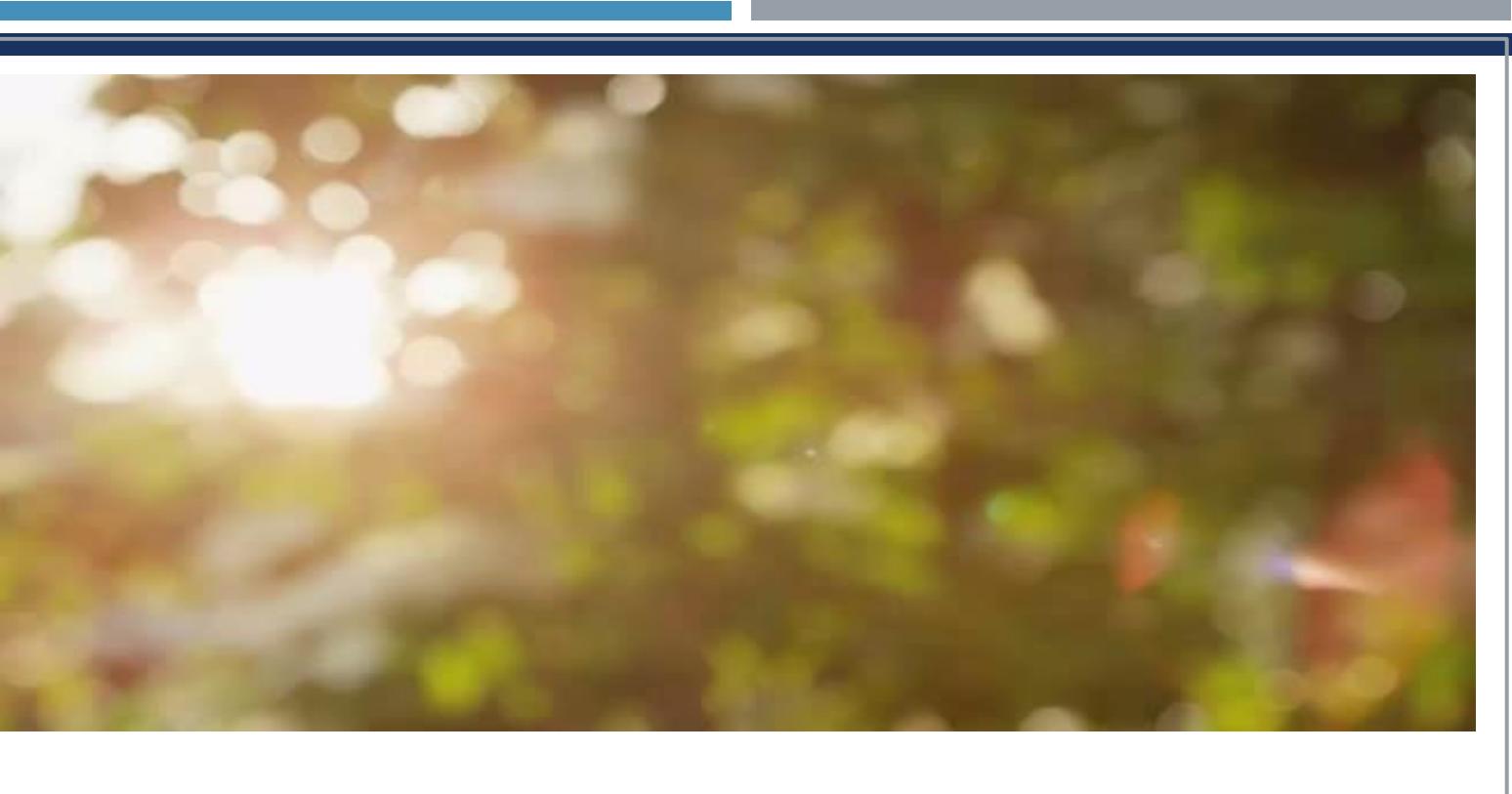
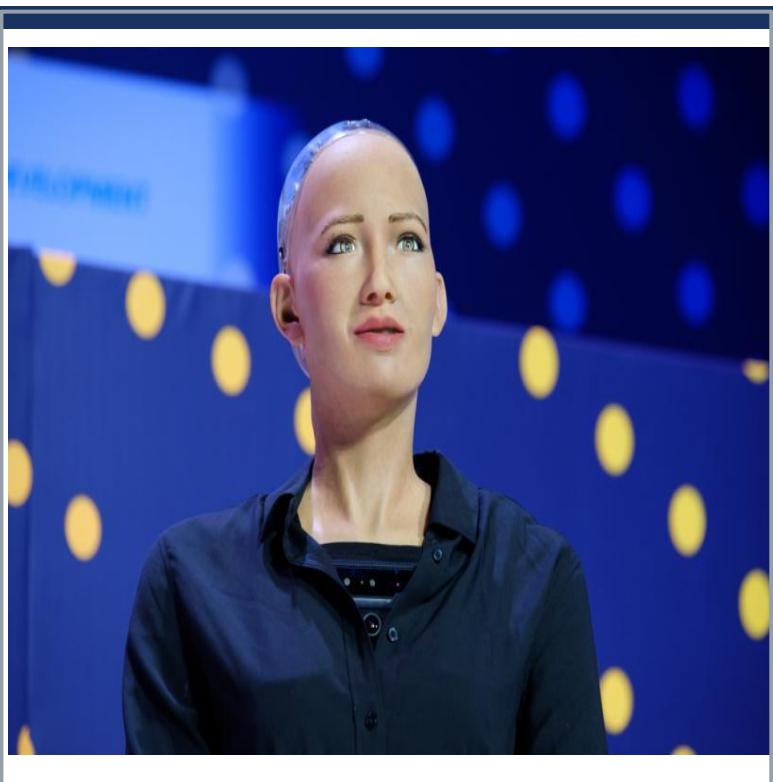
TEXT BOOKS

1. Artificial Intelligence A modern Approach, 3rd edition,
Stuart Russel and Peter Norwig
2. *Artificial intelligence : structures and strategies for complex problem*, 6th Edition, Luger, George F, Pearson Education, 2009
3. *Master Machine Learning Algorithms*, Edition, v1.12, Jason Brownlee, eBook, 2017
4. *Artificial Intelligence*, Patrick H. Winston, 3rd edition, Pearson Education, 1992

AI APPLICATIONS

A GLIMPSE INTO THE FUTURE





SOPHIA

HUMANOID ROBOTS



NADINE



ASIMO



ERICA

NON-HUMANOID ROBOTS



PARO



AIBO



MIRO

IS AI ONLY ABOUT ROBOTS?

- HDFC Bank has developed an AI-based chatbot called EVA (Electronic Virtual Assistant), built by Bengaluru-based Senseforth AI Research.
- Apple's personal assistant, Siri.
- Amazon's Alexa
- Tesla Vehicles
- Google Maps
- Ride Sharing apps like Uber, Lyft
- Jarvis by Facebook
- Photo recognition apps

SOME DOMAINS WHERE AI IS INFUSED WITH

- Steering a driver-less car, AI autopilots for commercial flights
- Spam Filters
- Beating Gary Kasparov in a chess match
- Understanding language
- Healthcare: Robotic assistants in surgery
- Automated customer support: chatbots
- Personalized shopping experience
- Finance: Stockbrkerage prediction
- Travel and navigation
- Social Media
- Smart home devices
- Creative arts: watson BEAT
- Security and surveillance



INTRODUCTION TO AI

HOW TO DEFINE AI

WHAT IS ARTIFICIAL INTELLIGENCE?

- AI a term coined by McCortey
- **What is intelligence?**
- **Are humans intelligent?**
- ***“Can a machine think and behave like humans do?***

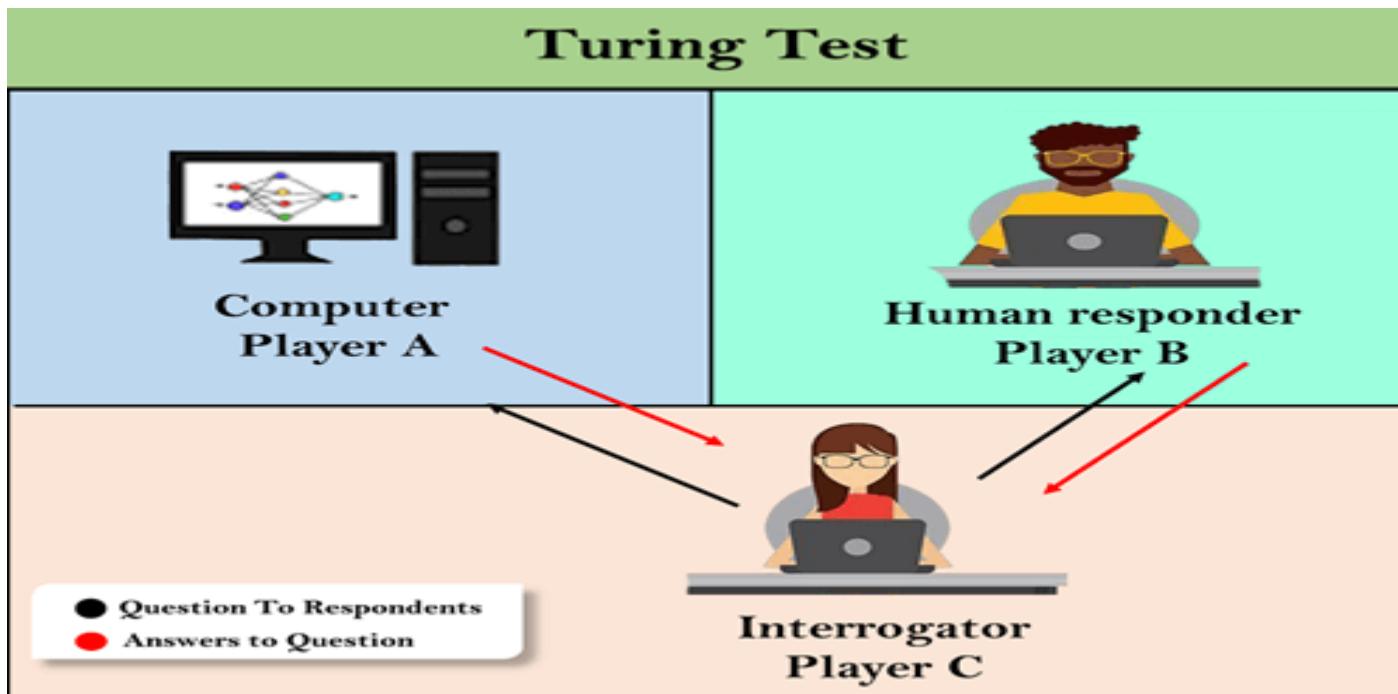
• **Artificial Intelligence** is a branch of computer science by which we can create intelligent machines which can behave like a human, think like humans, and able to make decisions

• Artificial Intelligence is the study of systems that

Think Humanly	Think Rationally
Acting/behaving Humanly	Acting/behaving Rationally

ACTING HUMANLY:TURING TEST

- Human beings are intelligent
- To be called intelligent, a machine must produce responses that are indistinguishable from those of a human



Alan Turing

- If interrogator cannot reliably distinguish human from computer, then computer possess intelligence.

AI TECHNIQUES: REQUIRED CAPABILITIES TO BEHAVE HUMANLY(TURING TEST)

Natural Language Processing:
Enabling the computer to communicate successfully in English

Knowledge Presentation: Storing what the computer knows or hear

Automated Reasoning: Using the stored information to answer questions and to draw new conclusions

Machine Learning:
Adapting to new circumstances and to detect and extrapolate patterns

Computer Vision: To perceive objects

Robotics: To manipulate objects and move about.

COGNITIVE MODELING APPROACH

To get inside the actual workings of human minds



Three ways to get inside the mind:

Introspection: Trying to catch our own thoughts as they go by

Psychological Experiments: observing a person in action

Brain Imaging: observing the brain in action



The interdisciplinary field of cognitive science brings together computer models from AI and experimental techniques from psychology to construct precise and testable theories of the human mind.

AI PROBLEMS



PLAYING CHESS



PROVING
MATHEMATICAL
THEOREMS



WRITING POETRY



DRIVING A CAR
ON A CROWDED
STREET



DIAGNOSING
DISEASES



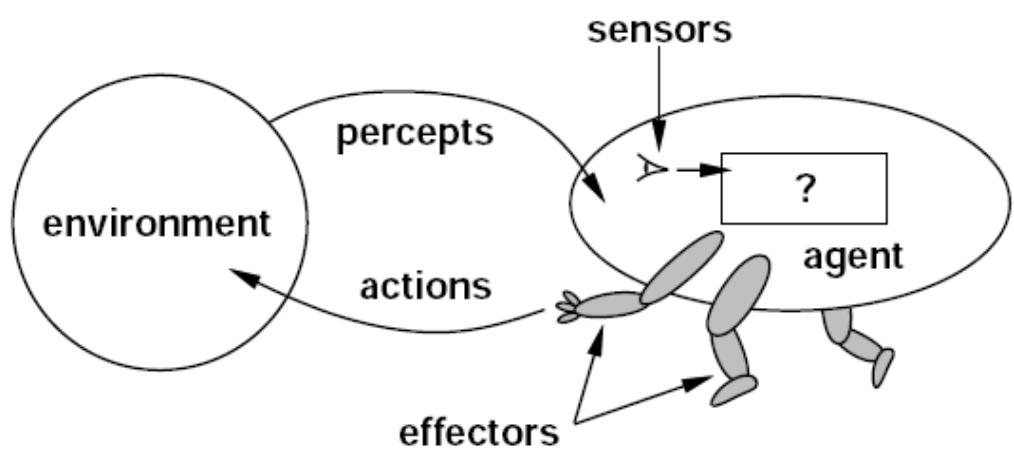
AI AGENTS

AGENTS

Agent = Architecture +
Program

- An **agent** is just something that acts.
- Agents can be classified according to the environment in which they are like Intelligent Agents, Human Agent, Robotic agent, Biological Agents, Software/computer Agents, Hardware Agents, Interface Agents, Mobile Agents, Reactive Agents, Information Agents, Distributed Artificial Intelligence Agents ...
- Computer Agents does:
 - operate autonomously
 - perceive their environment
 - persist over a prolonged time period
 - adapt to change
 - create and pursue goals

AI AGENTS



Agent-->perceives-->decides--> Acts

*An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators -
(Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norvig)*

Intelligent Agent must

- Must **sense/percept(current & memory)**
- Must **act/react with high performance and optimized result**
- Must **autonomous** (to some extend)
- Must take **rational action**

AI AGENT COMPONENTS

Sensors: An agent perceives its environment through sensors. Eg: cameras, infrared range finders

Actuators: Agent can change the environment through actuators/ effectors Eg: Motors

Percept: The complete set of inputs at a given time for an agent is called a percept

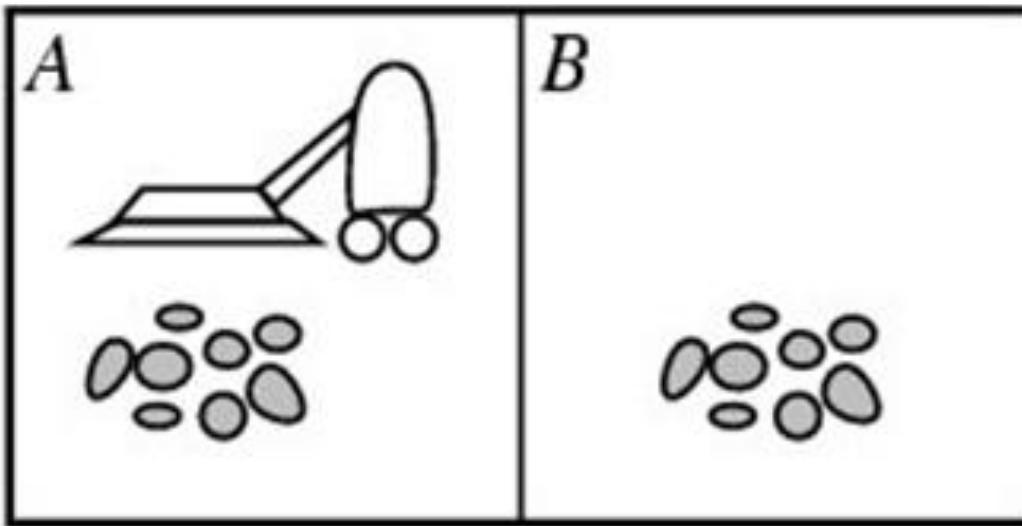
Percept sequence: The complete history of everything the agent has perceived

Agent function(agent's behaviour) maps from percept histories to actions: $[f: p^* \rightarrow A]$

Agent program: runs on the physical architecture to produce f. It is a concrete implementation

Action: An operation involving an actuator is called an action

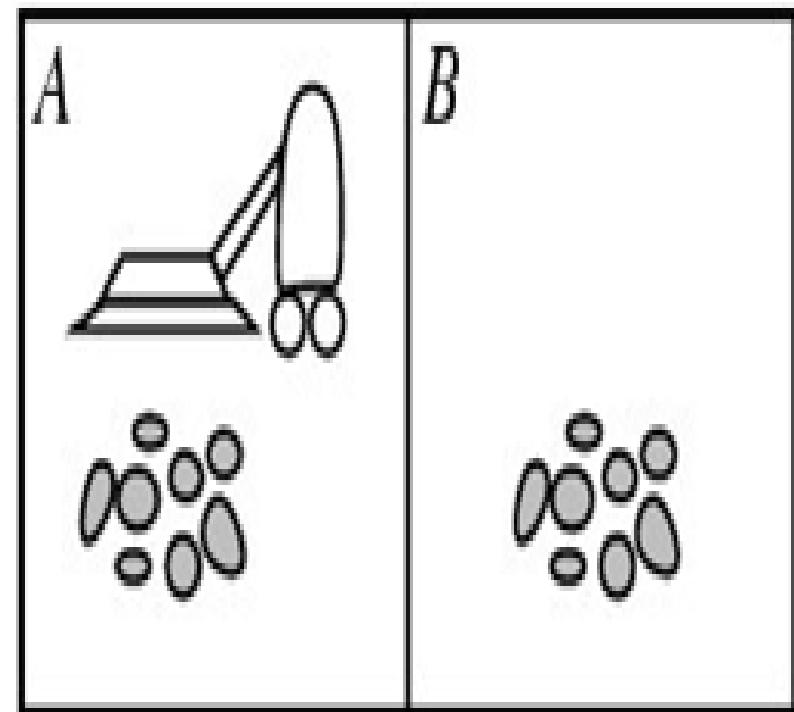
VACUUM-CLEANER WORLD



- **Percepts:** location and contents,
e.g., [A, dirty]
- **Actions:** Left, Right, Suck, NoOp
- **Agents Function:** *look-up table*

A SIMPLE AGENT FUNCTION

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



PROPERTIES OF AGENT

Rationality

Autonomy

Reactivity

RATIONALITY

An agent should "do the right thing", based on what it can perceive and the actions it can perform. The right action is the one that will cause the agent to be most successful

Definition Rationality: For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

Rational is different from omniscience

Percepts may not supply all relevant information. **We can behave rationally even when faced with incomplete information.**
E.g., in card game, don't know cards of others.



Rational is different from being perfect

Rationality maximizes expected outcome while perfection maximizes actual outcome.

RATIONALITY

DEFINITION: The **autonomy** of an agent is the extent to which its behavior is determined by its own experience, rather than knowledge of designer.

- Agents can perform actions in order to modify future percepts so as to obtain useful information: information gathering, exploration.

Extremes of rationality

- No autonomy – ignores environment/data
- Complete autonomy – must act randomly/no program

Example: baby learning to crawl

Ideal Rationalism

- Design agents to have some autonomy
- Possibly become more autonomous with experience

AUTONOMY

PEAS ANALYSIS



How to design agent? Must first specify the setting for intelligent agent design



Agents can be described by their PEAS.



A rational agent maximizes the performance measure for their PEAS



Specifying the task environment is
always the first step in designing agent



PEAS:

Performance
Environment
Actuators
Sensors

PEAS

PERFORMANCE MEASURES



Performance measure: *An objective criterion for success of an agent's behavior. The performance measure depends on the agent function.*



Performance measures of a vacuum-cleaner agent: amount of dirt cleaned up, amount of time taken, amount of electricity consumed, level of noise generated, etc.



Performance measures self-driving car: time to reach destination (minimize), safety, predictability of behavior for other agents, reliability, etc.

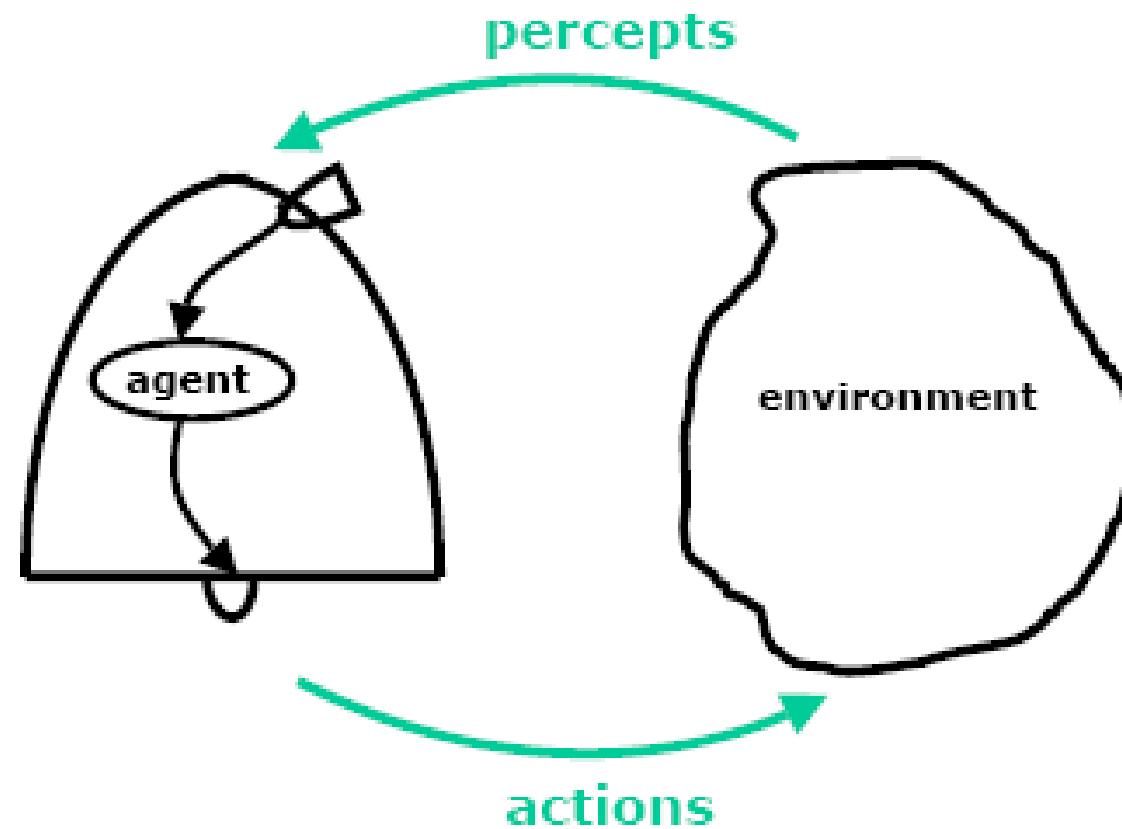


Performance measure of game-playing agent: win/loss percentage (maximize), robustness, unpredictability (to “confuse” opponent), etc.

THE ENVIRONMENT

- What all do we need to specify?
 - The action space
 - The percept space

The environment is a string of mappings from the action space to the percept space



TAXI DRIVER EXAMPLE

Performance Measure	Environment	Actuators	Sensors
safe, fast, legal, comfortable trip, maximize profits	roads, other traffic, pedestrians, customers	steering, accelerator, brake, signal, horn, display	camera, sonar, speedometer, GPS, odometer, engine sensors, keyboard, accelerator

MEDICAL DIAGNOSIS SYSTEM

Performance Measure	Environment	Actuators	Sensors
Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Screen display (questions, tests, diagnoses, treatments, referrals)	Keyboard (entry of symptoms, findings, patient's answers)

PART-PICKING ROBOT

Performance Measure	Environment	Actuators	Sensors
Percentage of parts in correct bins	Conveyor belt with parts, bins	Jointed arm and hand	Camera, joint angle sensors

INTERACTIVE ENGLISH TUTOR

Performance Measure	Environment	Actuators	Sensors
Maximize student's score on test	Set of students	Screen display (exercises, suggestions, corrections)	Keyboard

AI AGENT ENVIRONMENT

WORLD OUTSIDE OR THAT SURROUNDS AGENT

AGENT ENVIRONMENT TYPES

Fully Observable(vs. partially observable)

Deterministic(vs. stochastic)

Episodic (vs. sequential)

Static Environment (vs. dynamic)

Discrete(vs.continuous)

Single Agent(vs. multiagent)



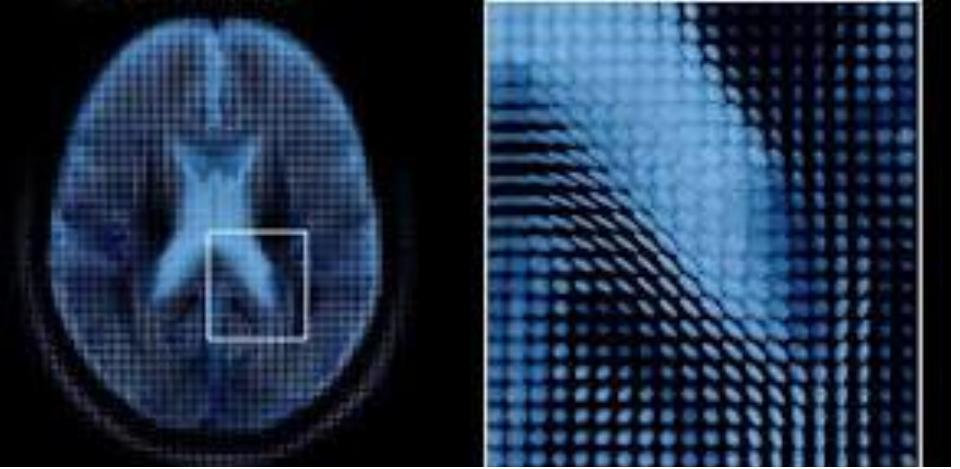
BASED ON OBSERVABILITY

- Is everything an agent requires to choose its actions available to it via its sensors? Perfect or Full information.
- **Fully observable** : An agent's sensors give it access to the complete state of the environment at each point in time. An accessible environment is one in which the agent can obtain complete, accurate, up-to-date information about the environment's state. The more accessible an environment is, the simpler it is to build agents to operate in it

Eg: Chess

- **Partially Observable**: The relevant features are partially observable. Need to track how to deliberate.

Eg: Poker



BASED ON DETERMINISM

- **Deterministic:** The next state of the environment is completely determined by the current state and the action executed by the agent.

Eg: Image analysis

- **Stochastic:** If an element of uncertainty occurs then it is stochastic may be partially observable. Non-deterministic environments present greater problems for the agent designer. Stochastic is random in nature. Does environment have aspects beyond the control of the agent?
- Utility functions have to guess at changes in world

Eg: Ludo

- **Strategic:** If the environment is dependent on preceding state and action of other agents

Eg: Chess



BASED ON DYNAMISM

- **Static (vs. dynamic):** The environment is unchanged while an agent is deliberating. A static environment is unchanged while an agent is reflecting.

Eg: Speech Analysis, Crossword Puzzle

- **Dynamic :** A dynamic environment is one that has other processes operating on it, and which hence changes in ways beyond the agent's control. Other processes can interfere with the agent's actions (as in concurrent systems theory). The physical world is a highly dynamic environment. Semi-Dynamic: The environment is semi-dynamic if the environment itself does not change with the passage of time, but the agent's performance score does

Eg: Vision systems in drones



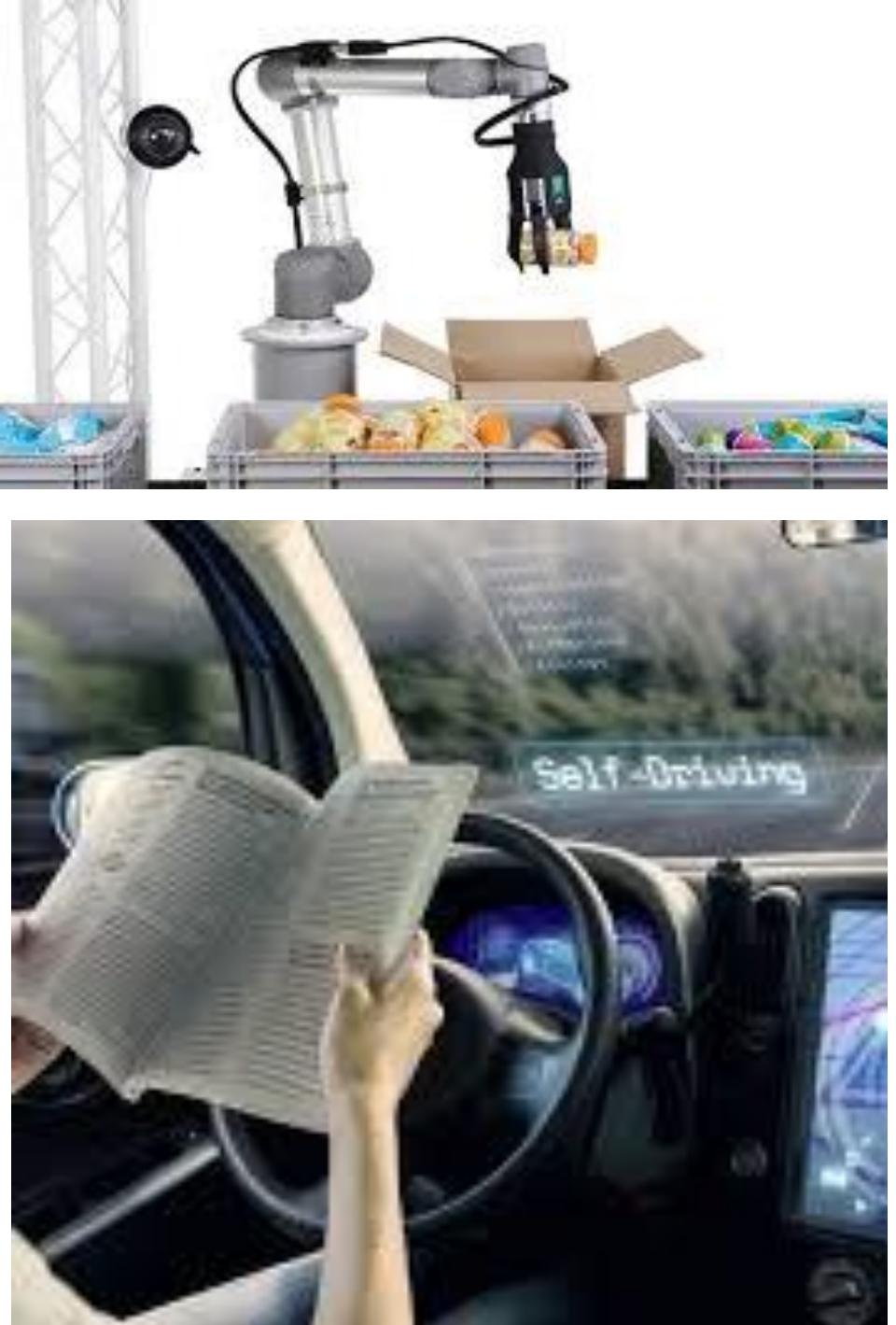
BASED ON CONTINUITY

- **Discrete (vs. continuous):** A limited number of distinct, clearly defined percepts and actions. An environment is discrete if there are a fixed, finite number of actions and percepts in it . Discrete environments could in principle be handled by a kind of “lookup table”

Eg: chess game, crossword

- **Continuous:** Continuous environments have a certain level of mismatch with computer systems

Eg: taxi driving.



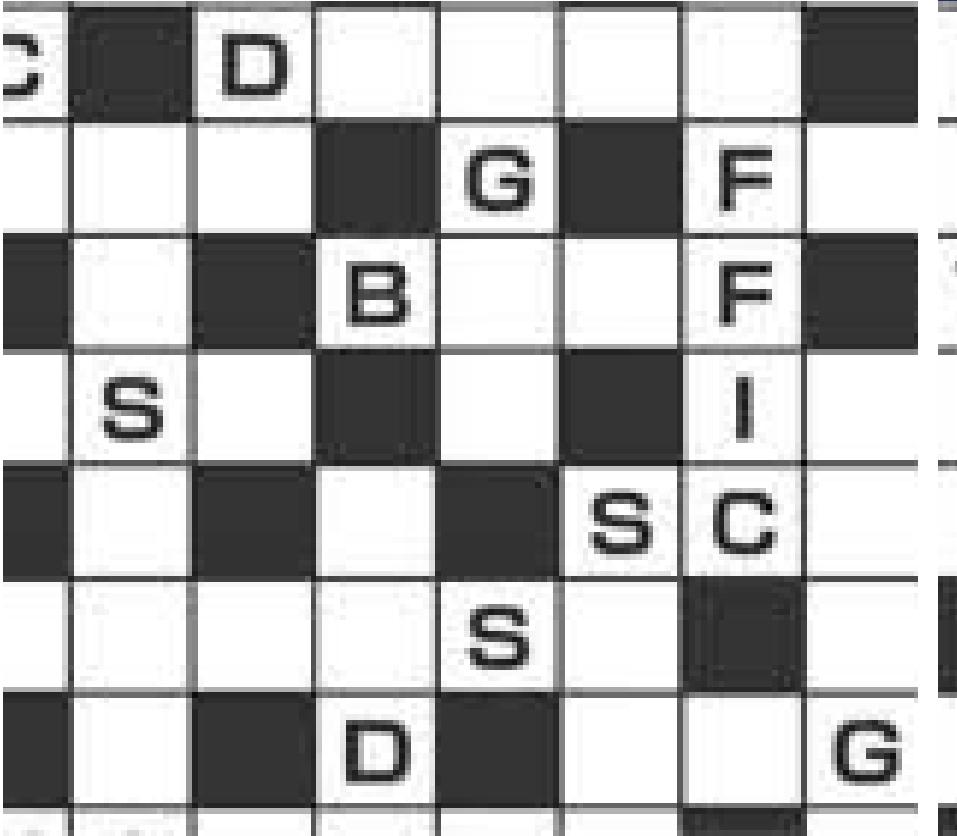
BASED ON AGENTS EXPERIENCE

- **Episodic :** The agent's experience is divided into discrete "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself. In an episodic environment, the performance of an agent is dependent on a number of discrete episodes, with no link between the performance of an agent in different scenarios. Episodic environments are simpler from the agent developer's perspective because the agent can decide what action to perform based only on the current episode — it need not reason about the interactions between this and future episodes

Eg: Part Picking Robot, Image Analysis

- **Sequential:** In a Sequential environment, the agent has to take decisions considering its previous decisions.

Eg: Chess, Automated car



BASED ON NUMBER OF AGENTS

- **Single agent:** An agent operating by itself in an environment.

Eg:Crossword

- **Multiagent:** A multi-agent system is a computerized system composed of multiple interacting intelligent

Eg: Chess

EXAMPLE

Environment	Accessible	Deterministic	Episodic	Static	Discrete
Chess with a clock	Yes	Yes	No	Semi	Yes
Chess without a clock	Yes	Yes	No	Yes	Yes
Poker	No	No	No	Yes	Yes
Backgammon	Yes	No	No	Yes	Yes
Taxi driving	No	No	No	No	No
Medical diagnosis system	No	No	No	No	No
Image-analysis system	Yes	Yes	Yes	Semi	No
Part-picking robot	No	No	Yes	No	No
Refinery controller	No	No	No	No	No
Interactive English tutor	No	No	No	No	Yes

AGENT TYPES

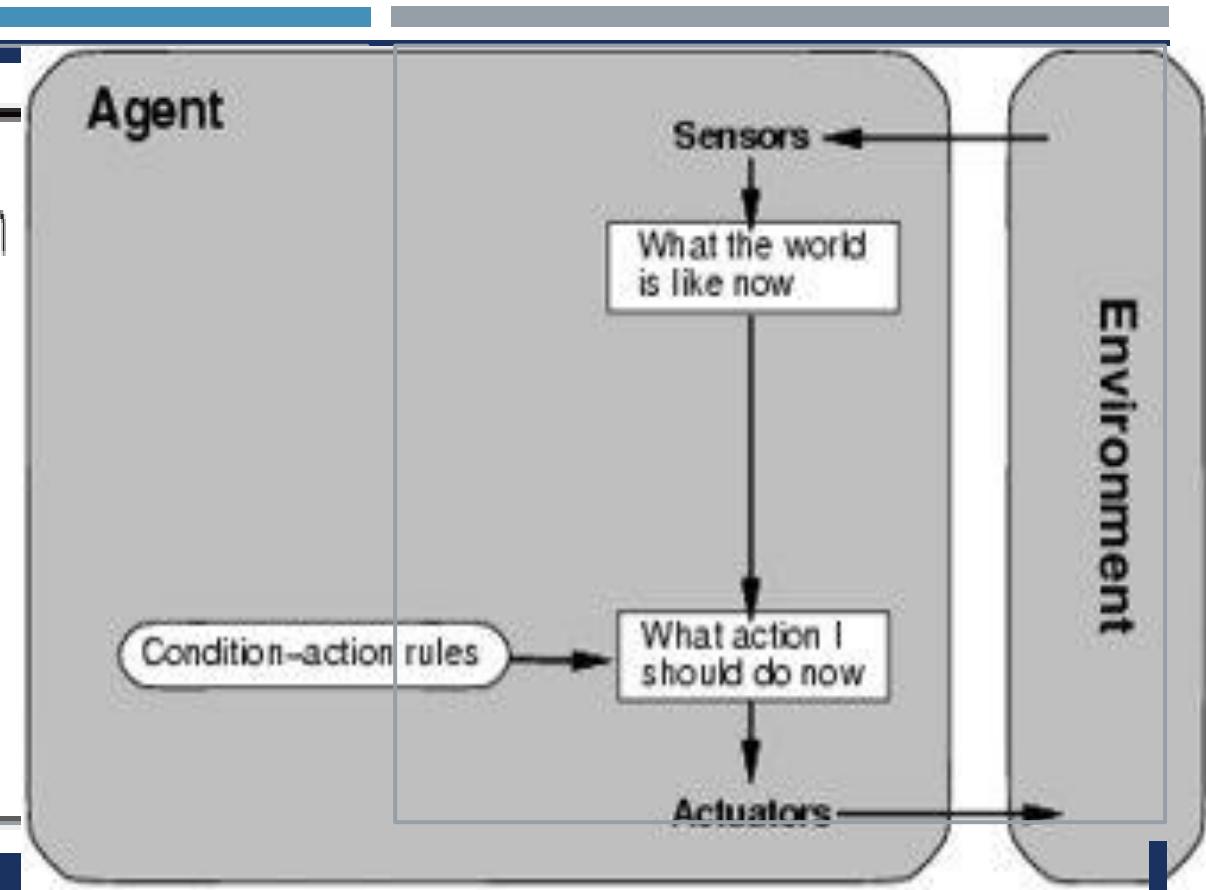
- Four basic types in order of increasing generality:

1. **Simple reflex agents**
2. **Reflex agents with state/model**
3. **Goal-based agents**
4. **Utility-based agents**
5. **Learning Agents**



```
function REFLEX-VACUUM-AGENT([location,status]) returns an action
```

```
if status = Dirty then return Suck  
else if location = A then return Right  
else if location = B then return Left
```



SIMPLE REFLEX AGENTS

SIMPLE REFLEX AGENTS

Action **does not depend on percept history, only on current percept.**

Environment should be **fully observable**

Agent function uses **Condition-Action-rule.**

Condition-Action Rule: It is a rule that maps a state (condition) to an action.

They are rational only if a correct decision is made only on the basis of current precept.

Eg: **Robotic Vacuum Cleaner, smart thermostat**

If each internal state includes all information relevant to information making, the state space is **Markovian**

SIMPLE REFLEX AGENTS

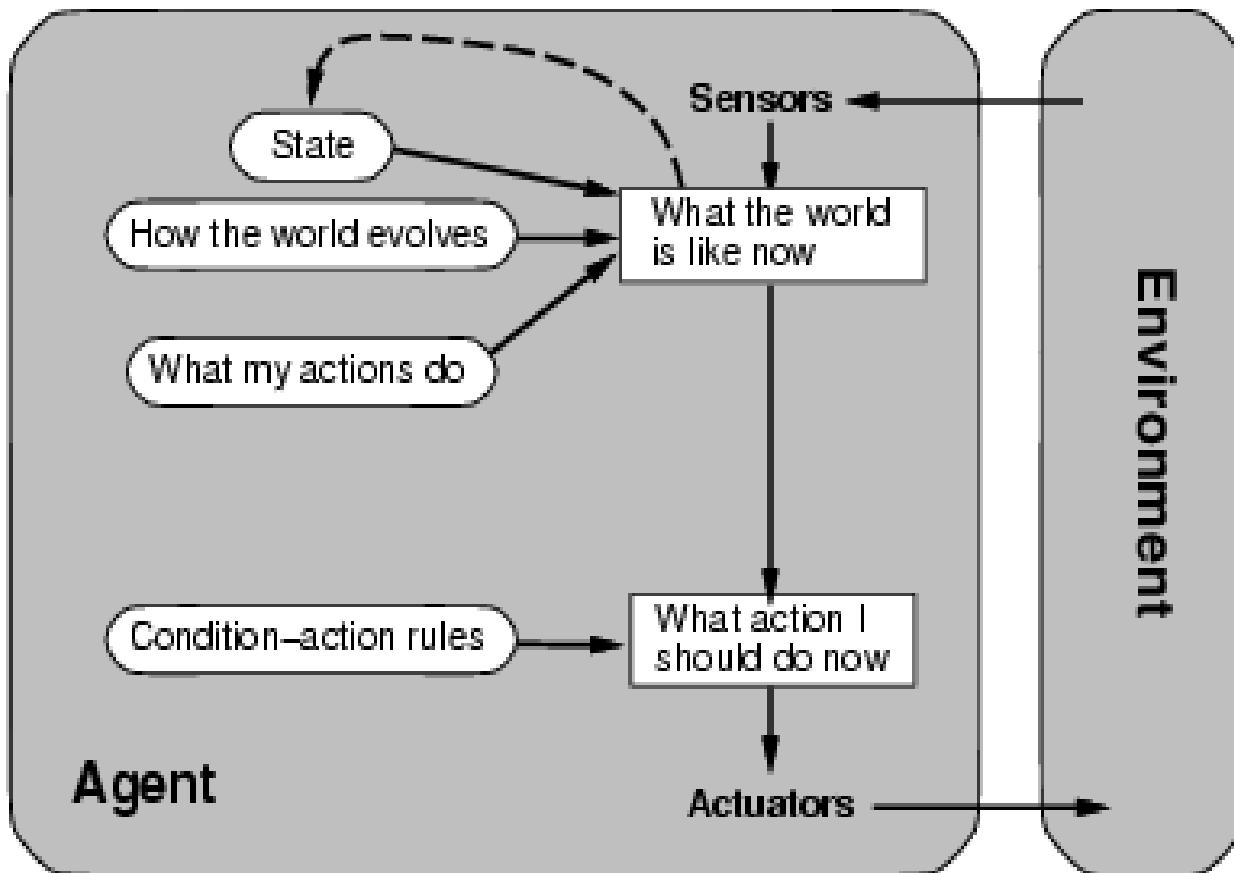
ADVANTAGES

- Simple but very limited intelligence.
- Therefore no memory requirements.

DISADVANTAGES

- They have very low intelligence capability as they don't have the ability to store past state.
- No knowledge of non-perceptual parts of state.
- No autonomy as actions are predetermined and everything is done by designer.
- Usually too big to generate and store.
 - Lookup table (not a good idea in general) as 35100 entries required for the entire game
- If there occurs any change in the environment, then the collection of rules need to be updated.
- Infinite loops due to partially observable environments
 - Suppose vacuum cleaner does not observe location. What do you do given location = clean? Left of A or right on B -> infinite loop.
- Possible Solution: Randomize action.

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)
  rule  $\leftarrow$  RULE-MATCH(state, rules)
  action  $\leftarrow$  RULE-ACTION(rule)
  state  $\leftarrow$  UPDATE-STATE(state, action)
  return action
```

MODEL BASED AGENT



Can handle **Partially observable Environment**.



It also has the capability to **store the internal state (past information)** based on previous events. Model-Based Agents **updates the internal state at each step**.



Agent perceives from history and has to keep **track of internal state** which is adjusted by each percept depending on percept history.



Knowledge is used to **build model and hence requires Knowledge Database** and becomes knowledge based agent

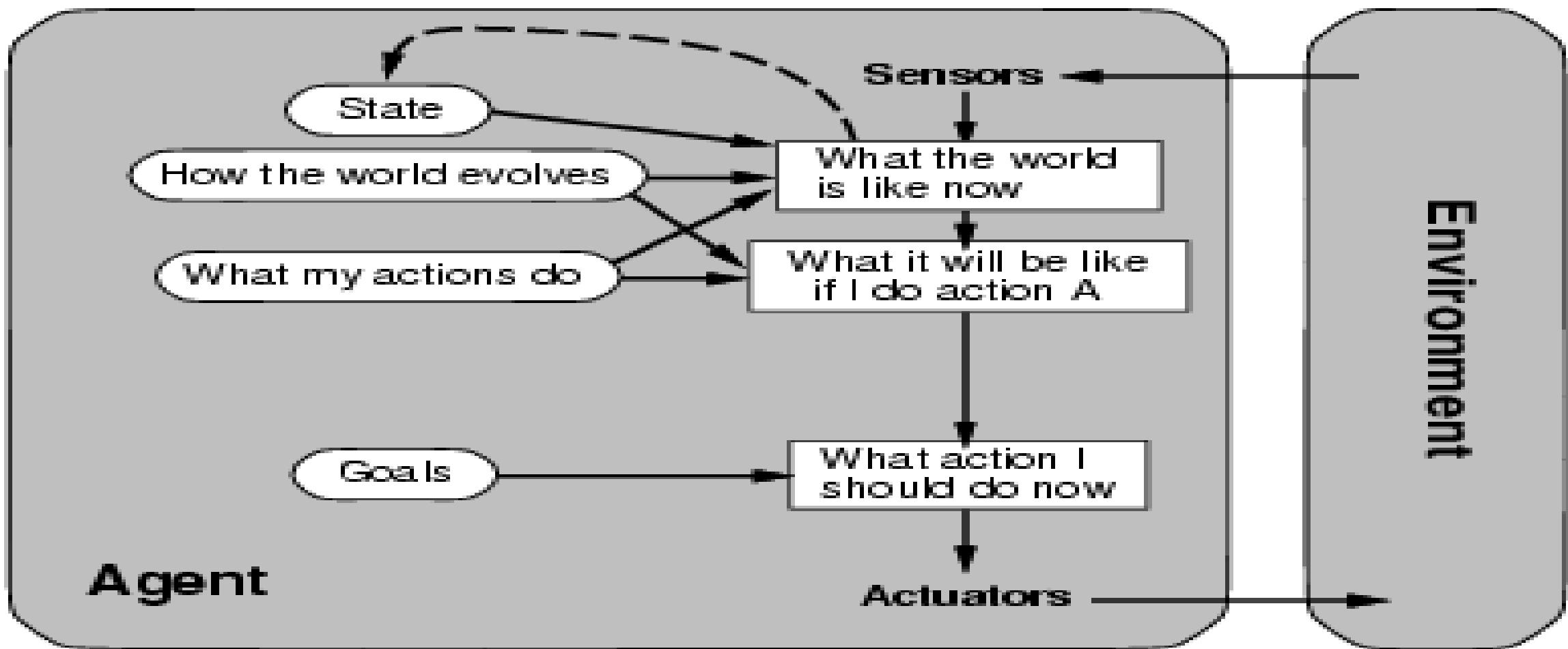


Condition action rules based on perception



Eg: **self-steering mobile vision**

GOAL-BASED AGENTS



GOAL BASED AGENTS

Depends on **current percept, previous percept & goal or desirable situation.**

It has a **goal and has a strategy to reach that goal.** A **goal** is a description of a desirable situation.

All actions are taken to reach this goal. From a set of possible actions, it selects the one that improves the progress towards the goal.

In order to attain its goal, it makes use of the **search and planning algorithm.**

The sequence of steps required to solve a problem is not known priori and is determined by **systematic exploration.**

Eg: **searching robot**

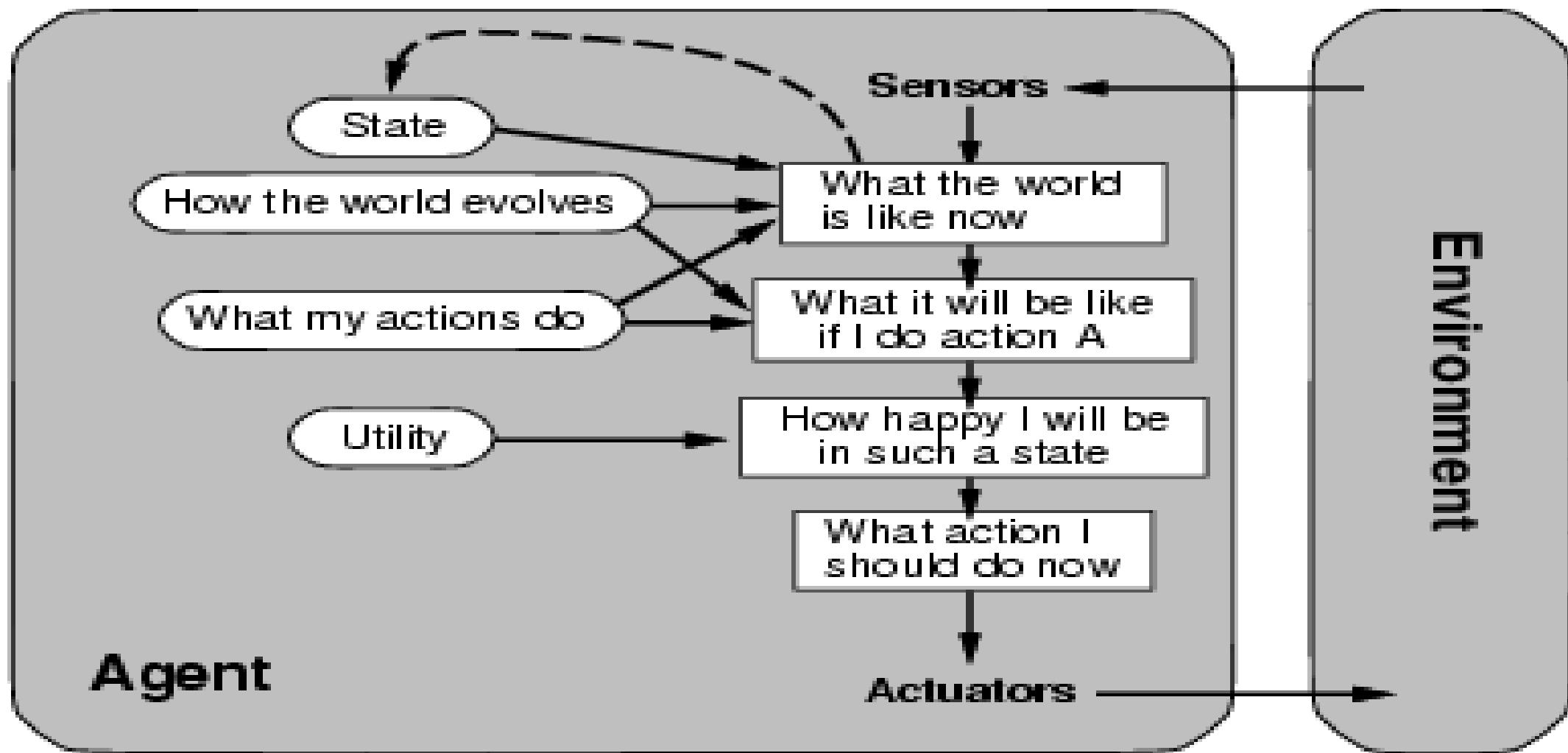
Keeping track of the current state is often not enough - need to add goals to decide which situations are good

Deliberative instead of **reactive.**

May have to consider long sequences of possible actions before deciding if goal is achieved – involves consideration of the future, “*what will happen if I do...?*”

One drawback of Goal-Based Agents is that **they don't always select the most optimized path to reach the final goal**

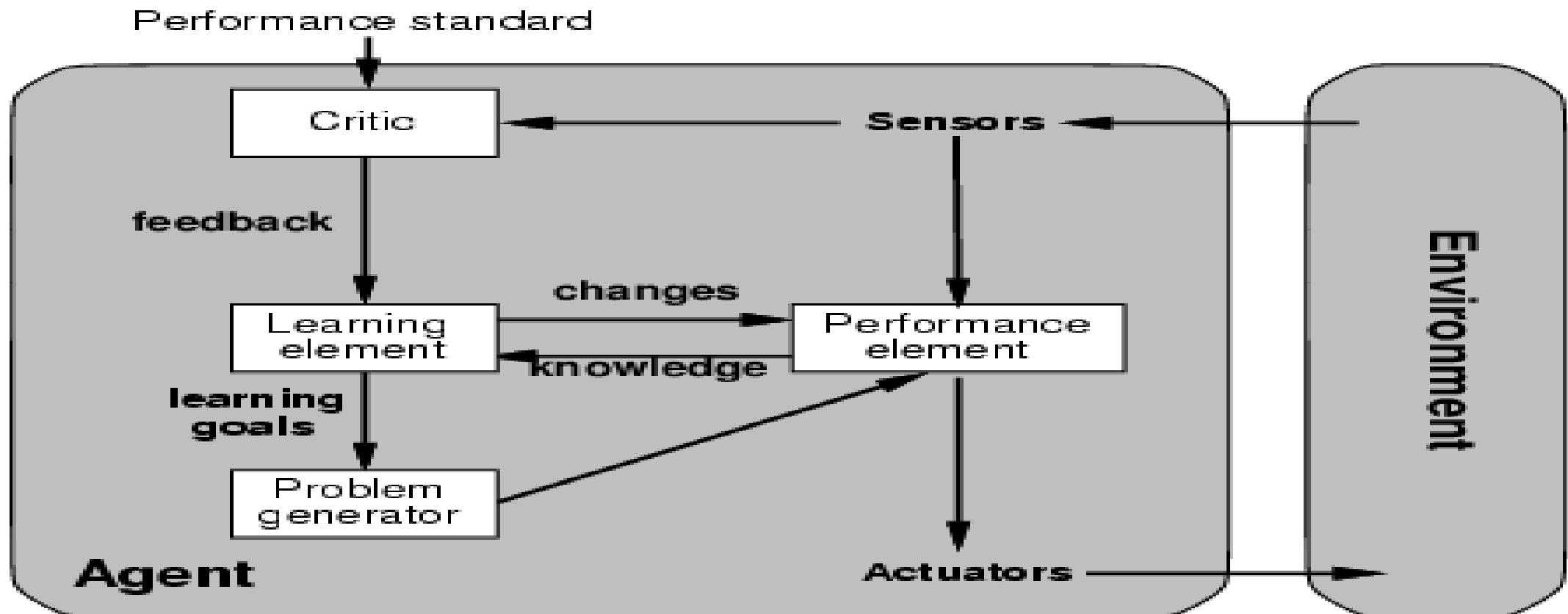
UTILITY-BASED AGENTS



UTILITY BASED AGENTS

-  Utility Agents are used when there are **multiple solutions to a problem and the best possible alternative** has to be chosen.
-  Action Depends on **utility function(state based)** and not goal.
-  A **utility function** maps a state onto a real number which describes the associated degree of “happiness”, “goodness”, “success”.
-  It is like the goal-based agent but with a measure of cost-benefit analysis of each solution and select the one which can achieve the goal in minimum cost.
-  Status of each state is checked, and decision is made.
-  Eg: **route recommendation system**

LEARNING AGENTS



LEARNING AGENTS

It is capable of **learning from past experience** which is achieved through **four major components**

It can operate on **unknown environments**

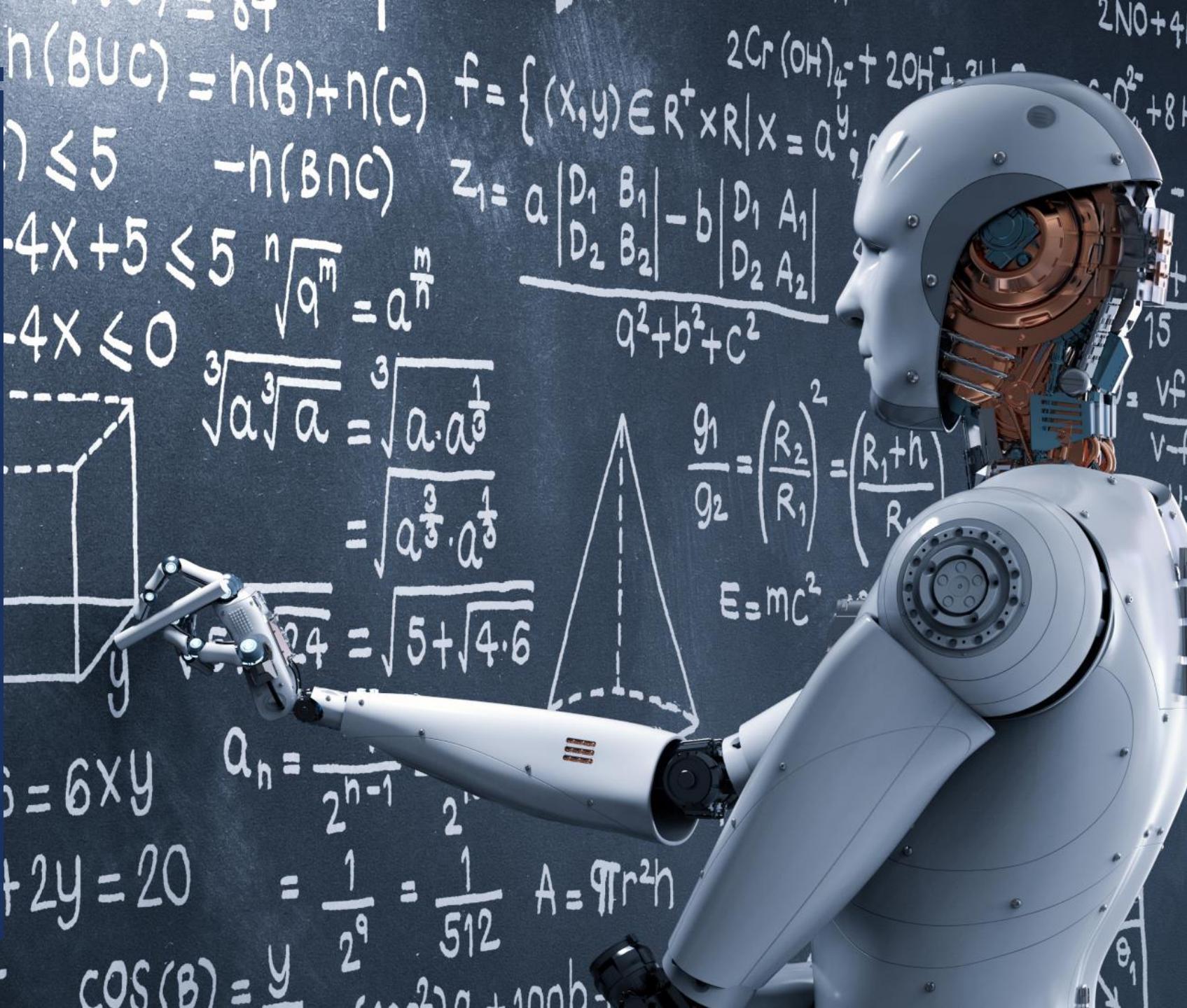
The learning agents have which enable to achieve it

- **Critic:** The Critic evaluates how well is the agent performing vis-à-vis the set performance benchmark.
- **Learning Elements:** It takes input from the Critic and helps Agent in performance improvement by learning from the environment.
- **Performance Element:** This component decides on the action to be taken to improve the performance.
- **Problem Generator:** Problem Generator takes input from other component and suggests actions which will result in a better experience.

It has the capability of **automatic information acquisition and integration** into the system.

These types of agents can start from scratch and over time can acquire significant knowledge from their environment.

AI PROBLEM FORMULATION



AI PROBLEM FORMULATION CONCERNS



How to **represent the problem precisely** so that can be easily analyzed?



How to **represent knowledge which has been isolated particularly for solving problem?**



Which **approach** to be used to solve a problem?



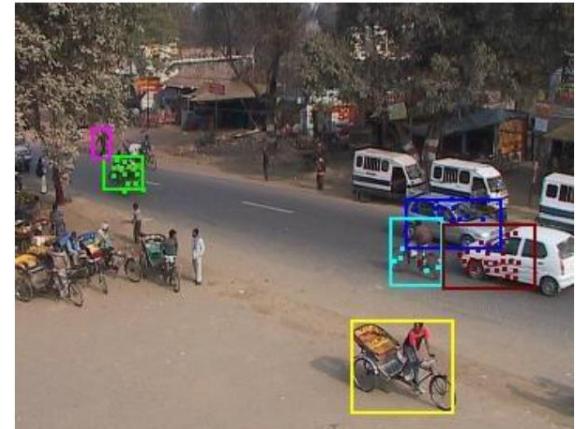
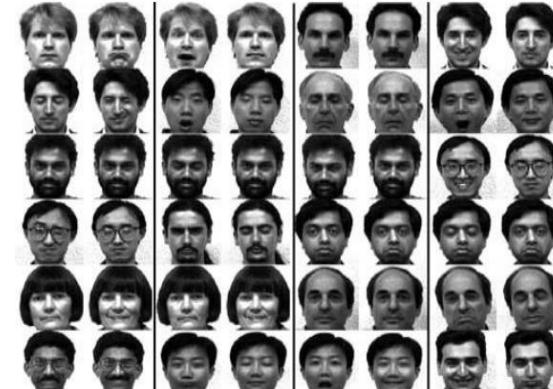
Problem-solving agents in AI mostly uses **search strategies or algorithms** to solve a specific problem and provide the best result.



AI agents should achieve **autonomy, adaptability, rationality, social ability, coperativity...**

KNOWLEDGE REPRESENTATION ISSUES

- **Structured Data**
 - Features already extracted as Data + tags; (Relational Databases)
 - e.g. Movie Preference matrix (Netflix)
- **Unstructured Data**
 - How to identify objects and their relations from unstructured data?
 - e.g. **Text:** Newspapers, blogs, technical papers
 - **Images:** ImageNet,
 - Which features to use?



Example : Face Recognition Events in Video

AI PROBLEMS ASSOCIATED TYPES

-  **Reasoning and Problem Solving:** How to work with incomplete and uncertain information? How to develop fast, intuitive algorithms for complex problems
-  **Knowledge Representation:** To store what it knows or receives. AI can use **Logical Representation**(Propositional logic, first order logic), **Production Rule**, **Semantic Network**: (graphical networks), **Frame representation**: (records with slots and attributes)
-  **Automated reasoning:** To use the stored information to answer questions and to draw new conclusions.
-  **Planning:** How to set goals, achieve them, and visualize the future? Accessibility to its environment, make predictions, evaluate predictions, and adapt according to its assessment
-  **Learning:** the ability to take a stream of input and find patterns in it. This includes classification and numerical regression.
-  **Natural Language Processing:** How to read human language and understand it
-  **Perception:** speech recognition, facial recognition, and object recognition
-  **Motion and Manipulation**
-  **Social Intelligence**

PROBLEM SOLVING AGENTS

- Intelligent agents adopt a goal and try to achieve it for maximizing performance measures.
- **Goal formulation**, based on the current situation and the agent's performance measure, is the **first step in problem solving**. A **goal is a description desirable state**. Goal directed agents need to achieve goal.
- **Goals of an agent** can be modelled using **state space search**. Each **state** is an **abstract representation of a physical configuration of agent's environment**.
- Agent moves from current state to successor state through some actions. It is known as **plan/ path** which is a sequence of actions to reach its goal and is examined using **goal test**.
- **Problem formulation**: choosing relevant set of states together and feasible set of operators to move from one state to another state, given a goal. Problem formulation is the process of deciding what sorts of actions and states to consider, given a goal
- **State space**: problem is divided into intermediates states to reach goal state, set of all possible states reachable from initial state
- An **operator** is a function that expands a node

PROBLEM SOLVING AGENTS

- An agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence
- Looking for such a sequence is called **search**
- A search algorithm takes a problem as input and returns a solution in the form of action sequence
- Once a **solution** is found the actions it recommends can be carried out – execution phase
- “**formulate, search, execute**” design for the agent
- A **node** is a data structure constituting part of a search tree includes state, parent node, action, path cost $g(x)$, depth

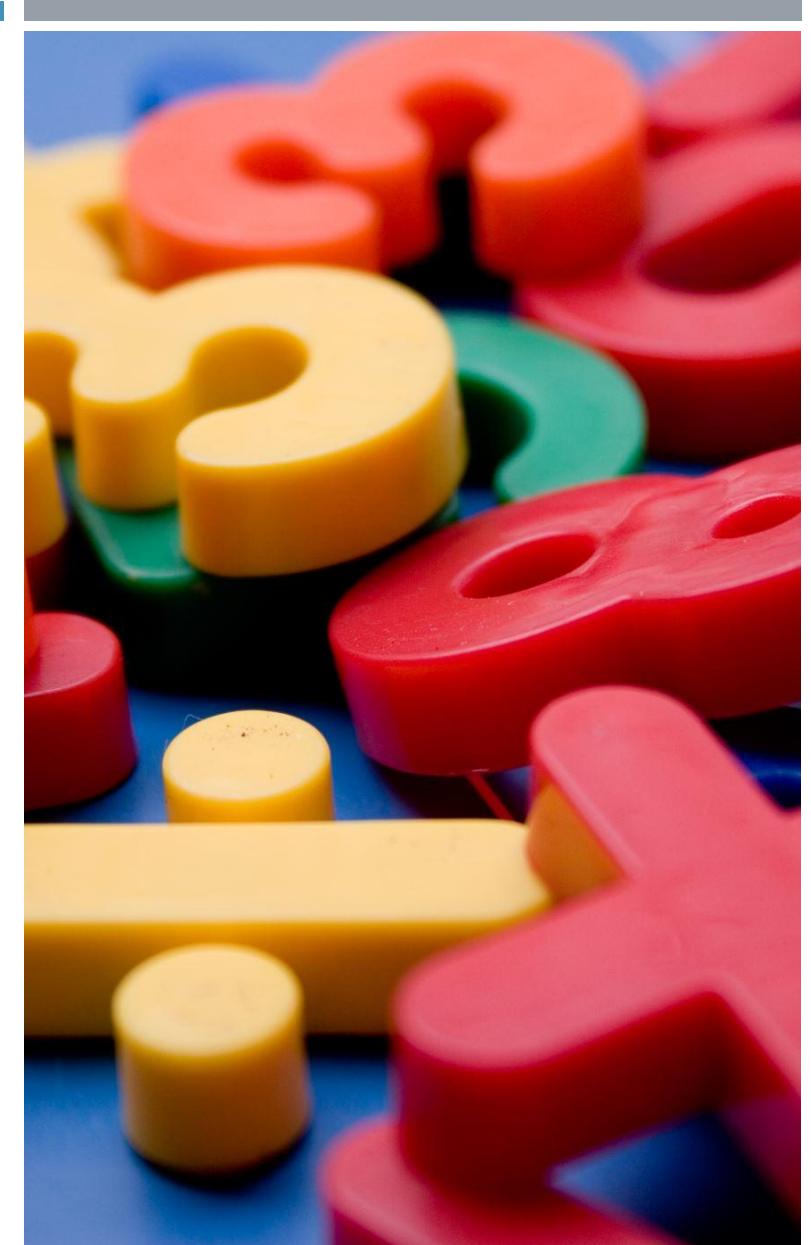
HOW TO FORMULATE A PROBLEM?

A problem can be defined formally by 5 components:

1. The **initial state** of the agent. In this case, the initial state can be described as *In:Arad*
2. The **possible actions** available to the agent, **corresponding to each of the state the agent resides in.**
3. The **transition model** describing what each action does. A successor function (transition model) Given a state, generates its successor states
4. The **goal test**, determining whether the current state is a goal state. Here, the goal state is *{In: Bucharest}*
5. The **path cost** function, which determine the cost of each path, which is reflecting in the performance measure.

REPRESENTING STATE

- The **size of a problem** is usually described in terms of the **number of states** that are possible.
 - The 8-puzzle has 181,440 states.
 - Tic-Tac-Toe has about 3^9 states.
 - Rubik's Cube has about 10^{19} states.
 - Checkers has about 10^{40} states.
 - Chess has about 10^{120} states in a typical game
- The number of actions / operators depends on the **representation** used in describing a state.
- **Closed World Assumption:** All necessary information about a problem domain is available in each percept so that each state is a complete description of the world.



AI problem can be represented as a state space in order to solve it which can be either **tree or graph**. The state space is then searched to find a solution to the problem.



A state space is set of all possible states where the problem can lead to from current state.



A search tree has following

Root = Start state	Set of nodes = representing each state of the problem	goal state.	Children = successor states	Edges = actions and costs. The legal moves from one state to another
--------------------	---	-------------	-----------------------------	--

HOW TO SOLVE A PROBLEM?

STATE SPACE

TREE: A *state space tree* is a tree constructed from all of the possible states of the problem as nodes, connected via state transitions from some initial state as root to some terminal state as leaf.

FORMAL REPRESENTATION OF STATE SPACE

- a state space can be graphically represented using **tree or graph** and formally represented as a tuple **{S,A,Action(s), Result(s,a), Cost(s,a)}**, in which:
 - **{ S }** is the set of all possible states;
 - **{A}** is the set of possible action, not related to a state but regarding all the state space;
 - **{Action(s)}** is the function that establish which action is possible to perform in a certain state;
 - **{Result(s,a)}** is the function that return the state reached performing action { a } in state {s}
 - **{ Cost(s,a) }** is the cost of performing an action {a} in state {s}. In many state spaces is a constant, but this is not true in general.

FEW TERMINOLOGIES



The initial state, the actions and the transition model together define the **state space** of the problem — the set of all states reachable by any sequence of actions



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



The **solution** to the given problem is defined as the sequence of actions from the initial state to the goal states.



The quality of the solution is measured by the cost function of the path, and an **optimal solution** has the lowest path cost among all the solutions.



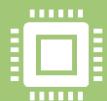
Completeness: Is the algorithm guaranteed to find a solution if there exist one?



Optimality: Does the algorithm find the optimal solution?



Time complexity: How long does it take for the algorithm to find a solution?



Space complexity: How much memory is consumed in finding the solution?

PERFORMANCE MEASURE OF PROBLEM- SOLVING AGENTS

CLASSICAL AI PROBLEMS

Toy problems are well-defined problems that have concise, exact description which are intended to illustrate or exercise various problem-solving methods. Following are the classical Toy problems.

- **TRAVELLING SALESMAN PROBLEM**
- **CHESS**
- **TOWER OF HANOI**
- **8-PUZZLE PROBLEM**
- **WATER JUG PROBLEM**
- **MISSIONARY-CANNIBAL PROBLEM**
- **N-QUEENS PROBLEM**
- **CRYPTARITHMETIC**
- **REMOVE 5 STICKS**
- **BLOCK WORD PROBLEM**

EXAMPLE I: 8 PUZZLE PROBLEM

Initial State

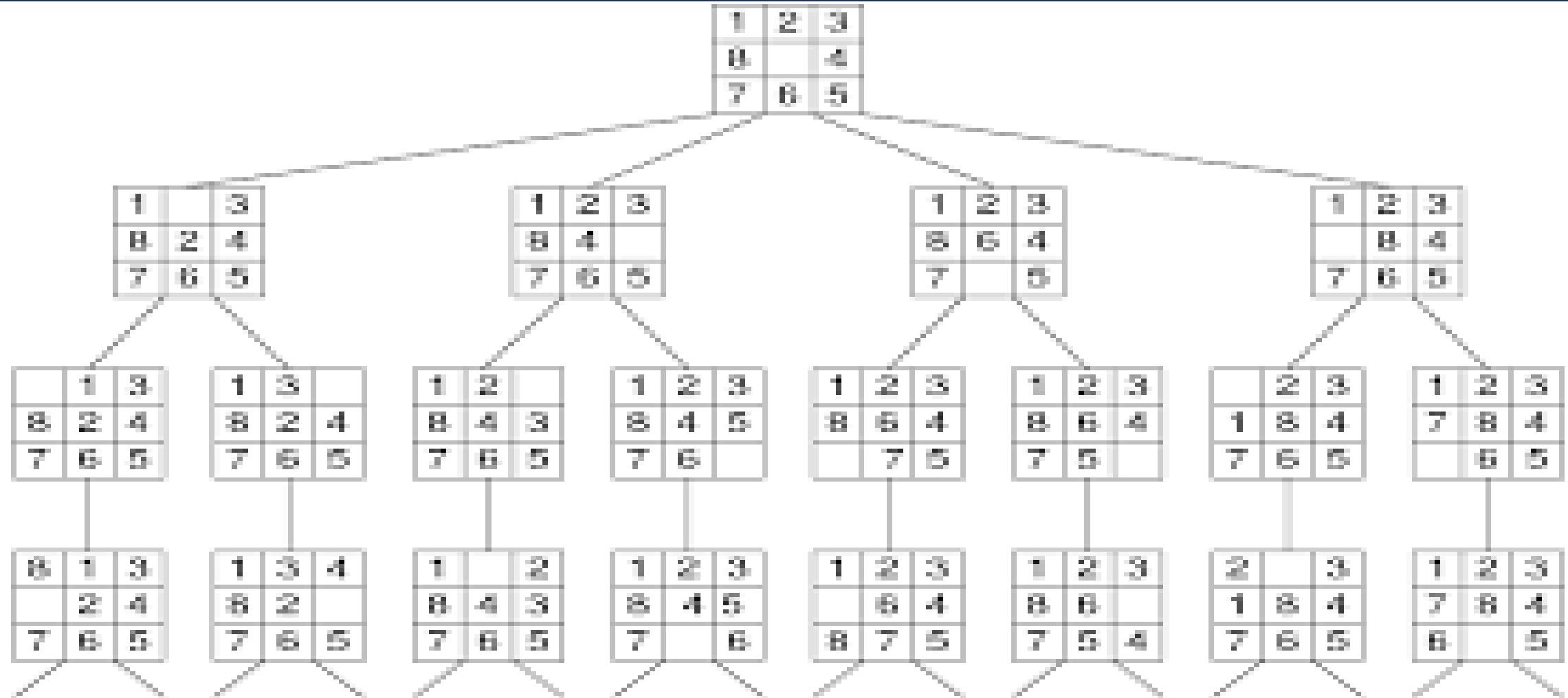
1	2	3
8		4
7	6	5

Goal State

2	8	1
	4	3
7	6	5

- **PROBLEM DEFINITION:** Given a 3-by-3 grid with 8 square blocks labeled from 1 to 8 and a blank square. Goal is to rearrange the blocks to **produce a desired goal configuration of the tiles**. Player permitted to slide blocks horizontally or vertically into the blank square, if the blank square is in adjacent space.
- **State:** Specification of each of the eight tiles in the nine squares (the blank is in the remaining square).
- **Initial state:** Any state.
- **Successor function (Actions):** Blank moves *Left, Right, Up, or Down*.
- **Goal test:** Check whether the goal state has been reached.
- **Path cost:** Each move costs 1. The path cost = the number of moves.

STATE SPACE TREE FOR 8 PUZZLE PROBLEM

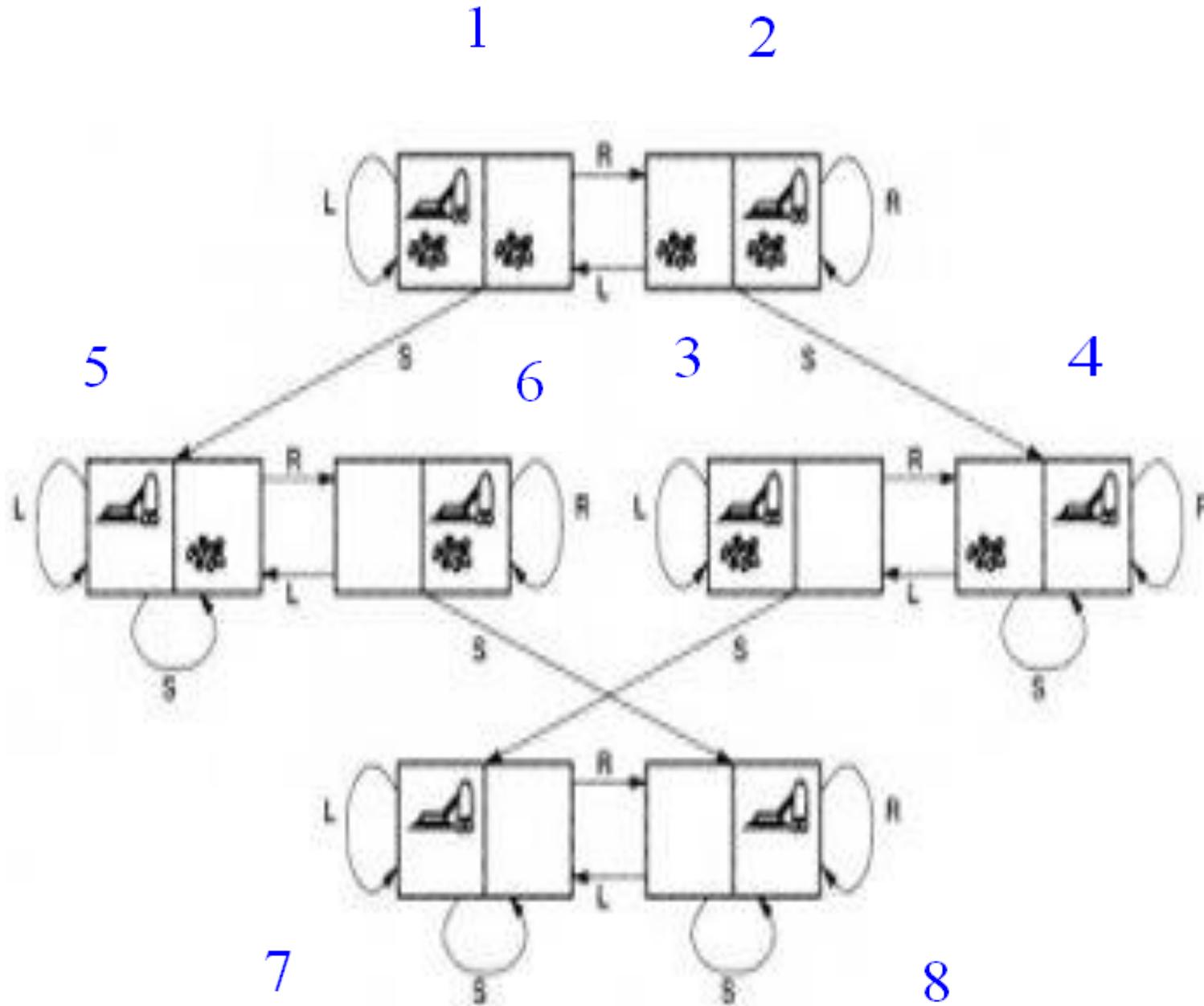


EXAMPLE 2: VACCUM CLEANER WORLD

- **States** integer dirt and robot locations. The agent is in one of two locations, each of which might or might not contain dirt – 8 possible states
- **Initial state:** any state
- **Actions:** left, right, suck, noOp
- **Goal:** test not dirty?
- **Path cost:** 1 per operation (0 for noOp)

STATE SPACE TREE

- Actions: Left, Right, Suck
- Goal state: 7, 8
- Initial state: one of {1,2,3,4,5,6,7,8}
- Right: one of {2,4,6,8}
- Solution: [Right, Suck, Left, Suck]



EXAMPLE 3: WATER JUG PROBLEM



- **PROBLEM DEFINITION:** given two jugs, a 4-gallon one and a 3-gallon one, Neither jug has any measuring markings on it. There is a pump that can be used to fill jugs with water. How can you get exactly 2 gallons of water in the 4-gallon jug?
- **State Representation:** Represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. where $0 \leq x \leq 4$, and $0 \leq y \leq 3$.
- **Initial state:** $(0,0)$
- **Goal state:** $(2,y)$ where $0 \leq y \leq 3$.

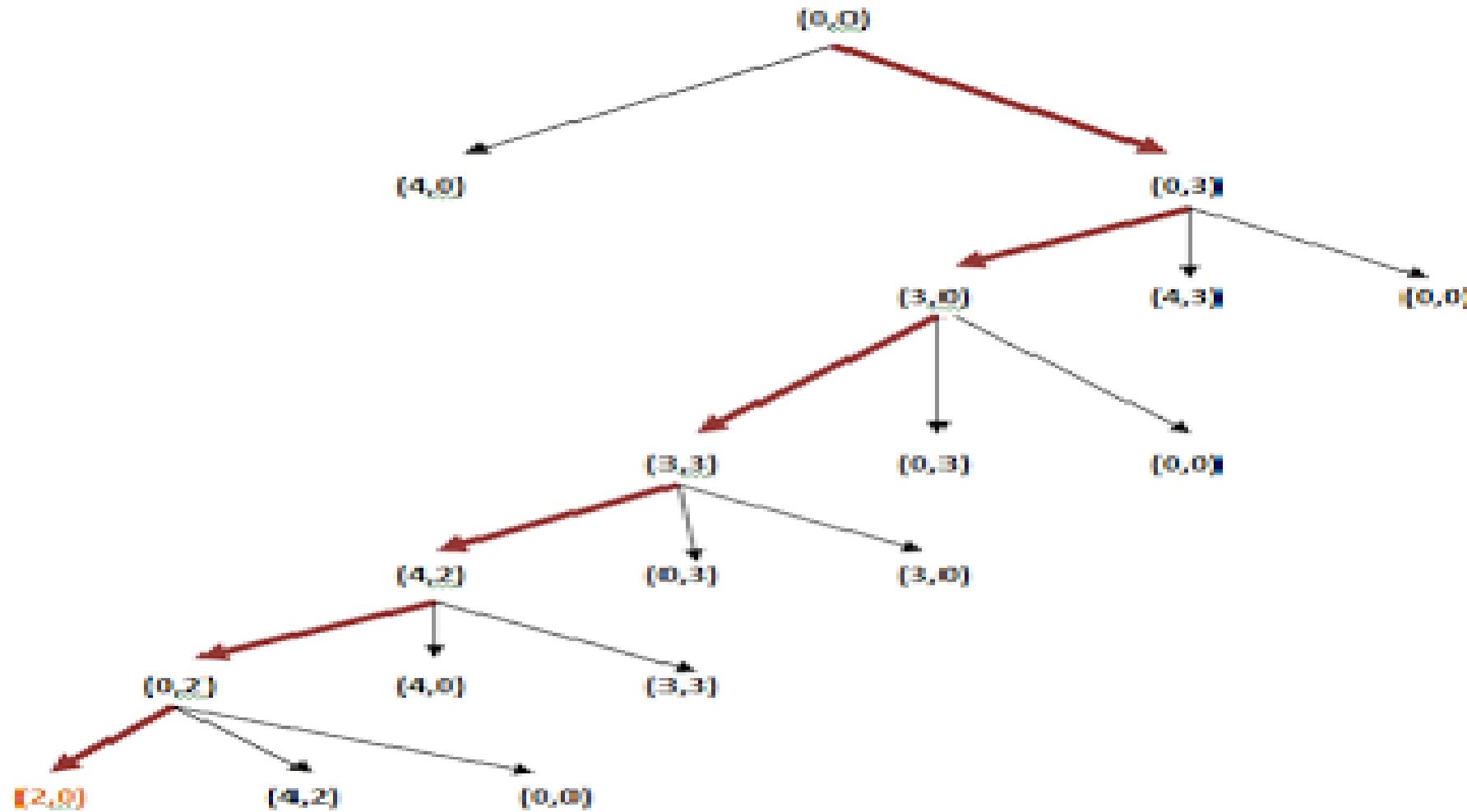
PRODUCTION RULES FOR SOLVING THE WATER JUG PROBLEM

No.	Initial State	Condition	Final state	Description of action taken
1.	(x,y)	If $x < 4$	$(4,y)$	Fill the 4 gallon jug completely
2.	(x,y)	if $y < 3$	$(x,3)$	Fill the 3 gallon jug completely
3.	(x,y)	If $x > 0$	$(x-d,y)$	Pour/spill some part from the 4 gallon jug
4.	(x,y)	If $y > 0$	$(x,y-d)$	Pour/spiil some part from the 3 gallon jug
5.	(x,y)	If $x > 0$	$(0,y)$	Empty the 4 gallon jug
6.	(x,y)	If $y > 0$	$(x,0)$	Empty the 3 gallon jug
7.	(x,y)	If $(x+y) < 7$	$(4, y-[4-x])$	Pour some water from the 3 gallon jug to fill the four gallon jug
8.	(x,y)	If $(x+y) < 7$	$(x-[3-y],3)$	Pour some water from the 4 gallon jug to fill the 3 gallon jug.
9.	(x,y)	If $(x+y) < 4$	$(x+y,0)$	Pour all water from 3 gallon jug to the 4 gallon jug
10.	(x,y)	if $(x+y) < 3$	$(0, x+y)$	Pour all water from the 4 gallon jug to the 3 gallon jug

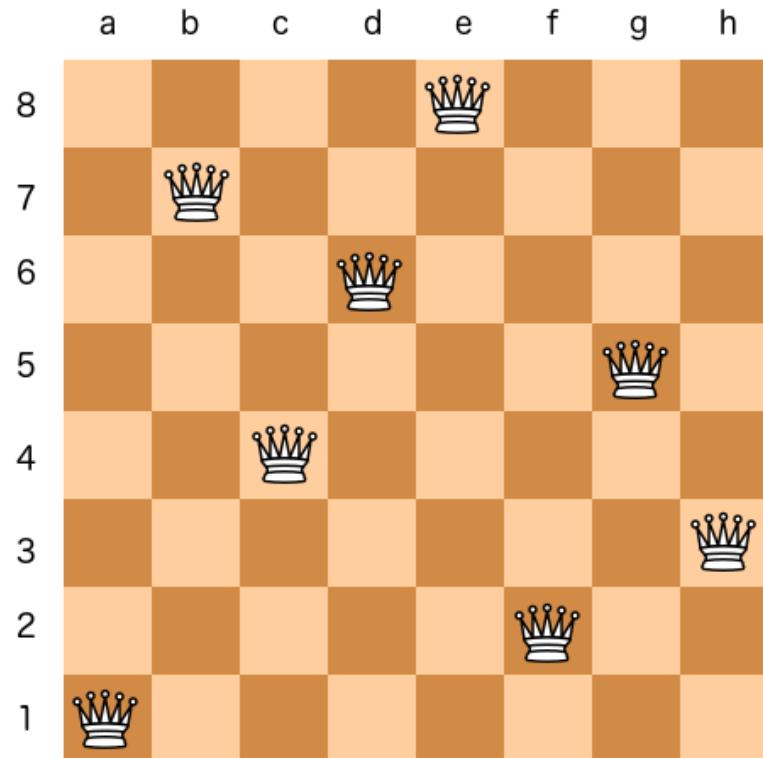
SOLUTION OF WATER JUG PROBLEM ACCORDING TO THE PRODUCTION RULES

S.No.	4 gallon jug contents	3 gallon jug contents	Rule followed
1.	0 gallon	0 gallon	Initial state
2.	0 gallon	3 gallons	Rule no.2
3.	3 gallons	0 gallon	Rule no. 9
4.	3 gallons	3 gallons	Rule no. 2
5.	4 gallons	2 gallons	Rule no. 7
6.	0 gallon	2 gallons	Rule no. 5
7.	2 gallons	0 gallon	Rule no. 9

STATE SPACE TREE FOR WATER JUG PROBLEM



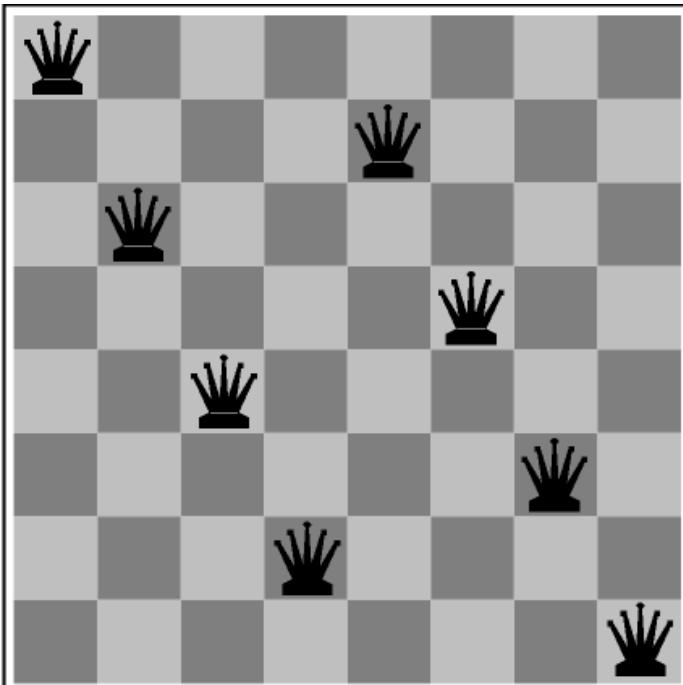
EXAMPLE 4: 8 QUEENS PROBLEM FORMULATION I



Naïve formulation/ Incremental formulation

- **PROBLEM DEFINITION:** Place 8 queens on an (8 by 8) chess board such that none of the queens attacks any of the others. This problem can be solved by searching for a solution.
- **states:** any arrangement of 0-8 queens on the board is a state
- **Initial state:** no queens on the board
- **actions:** add a queen to any empty square
- **goal test:** 8 queens are on the board, none attacked
- Problem: $64 \cdot 63 \cdot \dots \cdot 57 \gg 10^{14}$ possible states

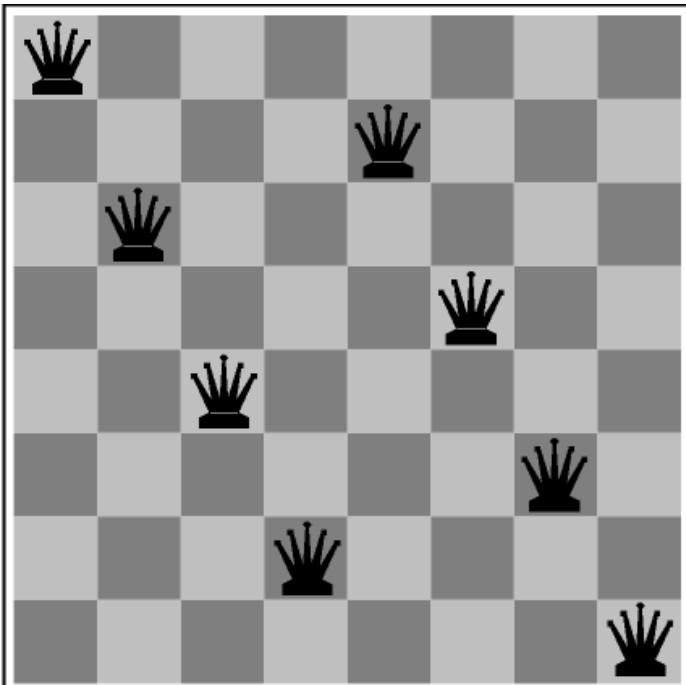
8 QUEENS PROBLEM ANOTHER FORMULATION



Complete Formulation

- **States:** Any arrangement of n queens ($0 \leq n \leq 8$) one per column in the leftmost n columns such that no queen attacks another. Any arrangement of 8 queens on the board.
- **Initial state:** All queens at column 1.
- **Successor function:** Change the position of any one queen. Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
- **Actions:** Change the position of any one queen
- **Problem:** 2,057 states. Sometimes no admissible states can be found.

8 QUEENS PROBLEM ANOTHER FORMULATION



Better formulation

- **States:** Any arrangement of k queens on k rows so that none are attacked.
- **Successor function:** Add a queen to $(k+1)$ the row so that none are attacked each other.
- **Initial state:** 0 Queens are on the board.

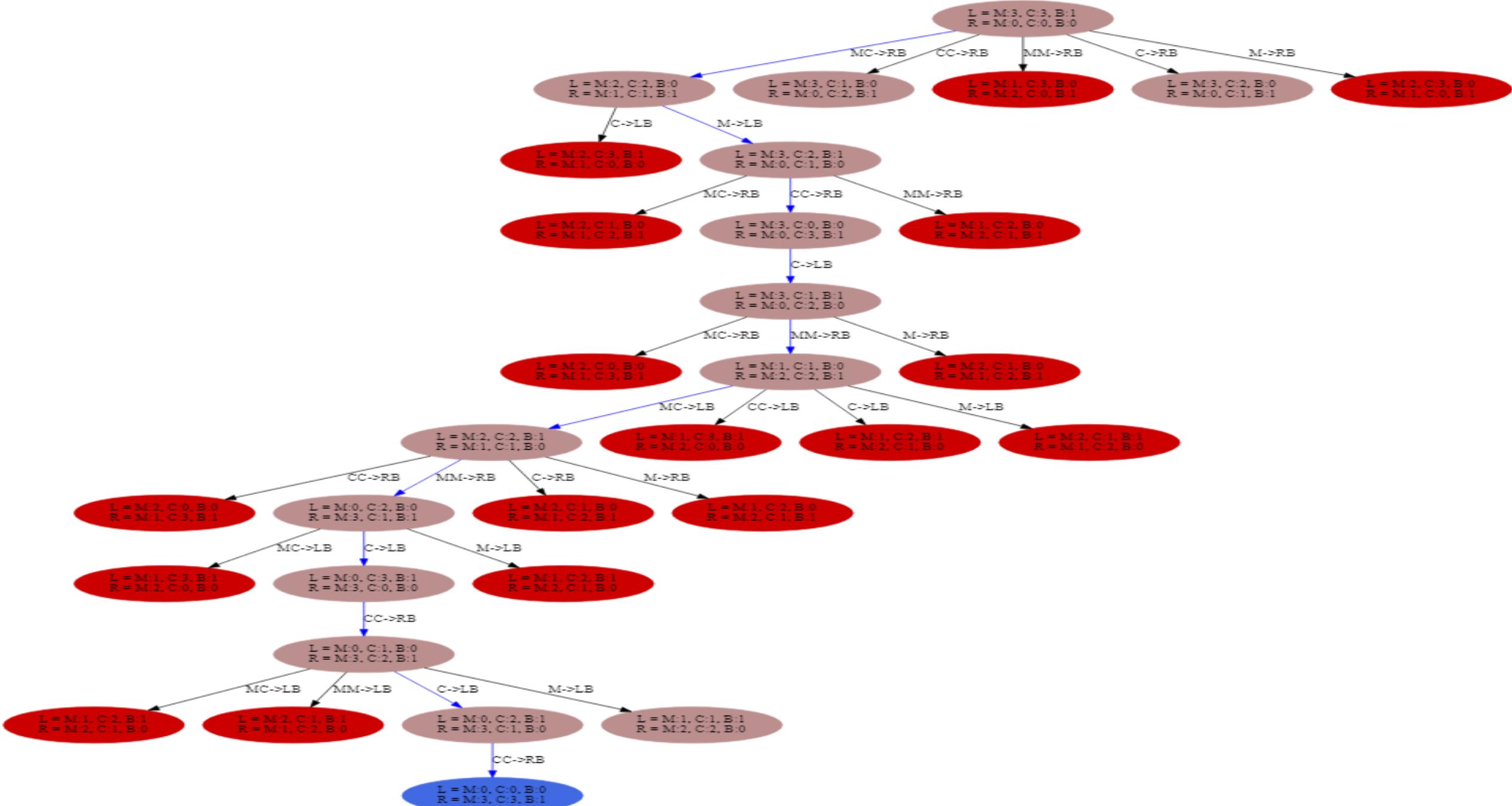
EXAMPLE 5: MISSIONARIES AND CANNIBALS

- **PROBLEM DEFINITION:** Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. **Find the smallest number of crossings that will allow everyone to cross the river safely**, without ever leaving a group of missionaries outnumbered by cannibals.
- **State:** $(m, c, l/t)$ where m: number of missionaries in the first/left bank c: number of cannibals in the first/left bank The last bit indicates whether the boat t is in the first bank. is in the first bank.
- **Start state:** $(3, 3, l)$ $(3, 3, l)$
- **Goal state:** $(0, 0, 0)$ $(0, 0, 0)$
- **Operators:** Boat carries $(1, 0)$ or $(0, 1)$ or $(1, 1)$ or $(2, 0)$ or $(0, 2)$

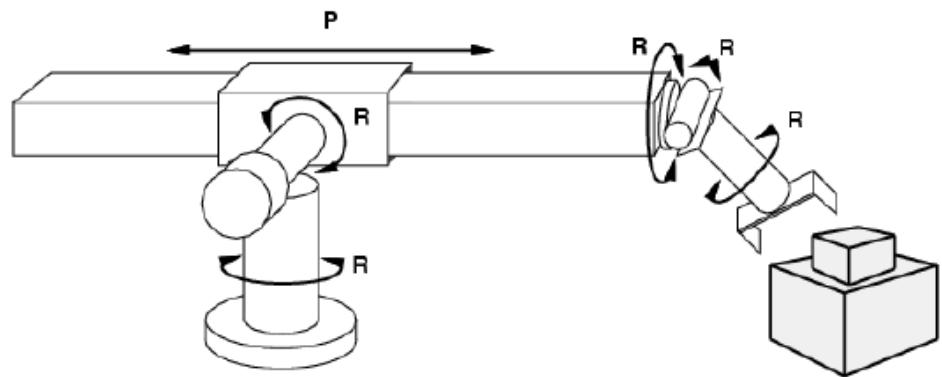
MISSIONARY AND CANNIBAL SOLUTION

	<u>Near side</u>		<u>Far side</u>
0 Initial setup:	MMMCCC	B	-
1 Two cannibals cross over:	MMMC		B CC
2 One comes back:	MMMCC	B	C
3 Two cannibals go over again:	MMM		B CCC
4 One comes back:	MMMC	B	CC
5 Two missionaries cross:	MC		B MMCC
6 A missionary & cannibal return:	MMCC	B	MC
7 Two missionaries cross again:	CC		B MMMC
8 A cannibal returns:	CCC	B	MMM
9 Two cannibals cross:	C		B MMMCC
10 One returns:	CC	B	MMMC
11 And brings over the third:	-		B MMMCCCC

SOLUTION TREE



EXAMPLE 6: ROBOTIC ASSEMBLY



- **states:** real-valued coordinates of robot joint angles parts of the object to be assembled
- **actions:** continuous motions of robot joints
- **goal test:** complete assembly
- **path cost:** time to execute

EXAMPLE 7:CRYPTARITMETIC

- **PROBLEM DEFINITION:**Find an assignment of digits (0, ..., 9) to letters so that a given arithmetic expression is true.

- **Examples:**

1. Dudeney's puzzle reads: SEND + MORE = MONEY.

- Cryptarithms are solved by deducing numerical values from the mathematical relationships indicated by the letter arrangements (i.e.) . S=9, E=5, N=6, M=1, O=0,....
- The only solution to Dudeney's problem: $9567 + 1085 = 10,652$.

2. **FORTY**

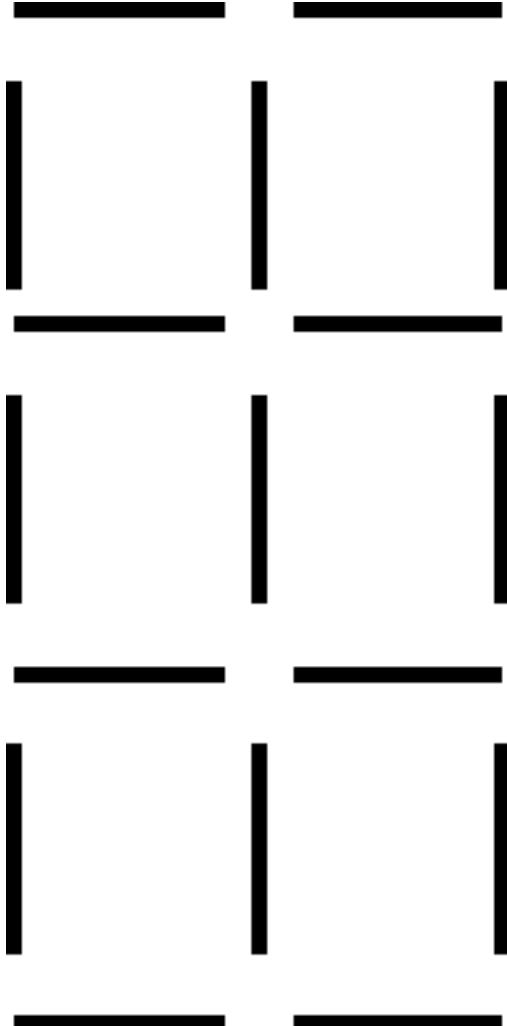
+ **TEN**

+ **TEN**

SIXTY

STATE SPACE REPRESENTATION

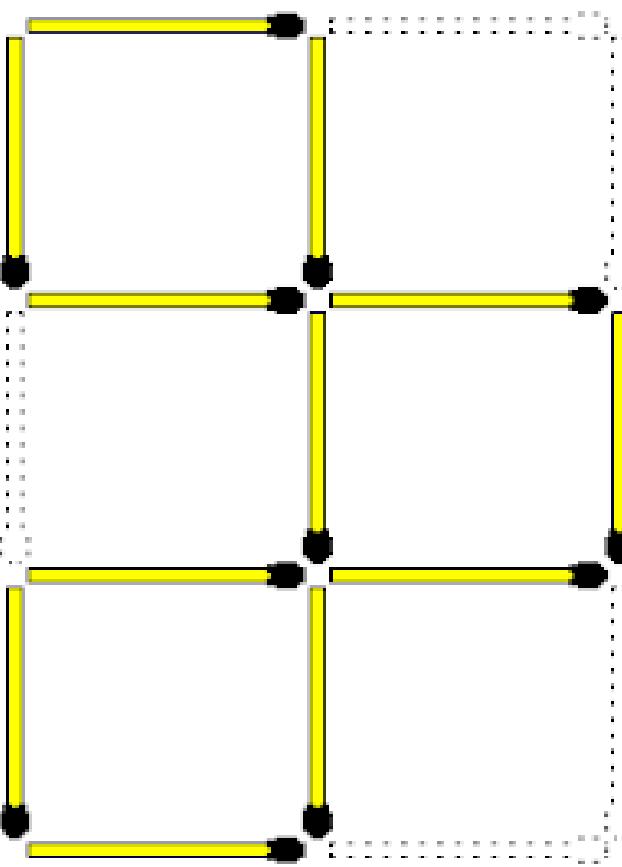
- **State:** mapping from letters to digits
- **Initial State:** empty mapping
- **Operators:** assign a digit to a letter ie no 2 letters have same digit
- **Goal Test:** whether the expression is true given the complete mapping
ie it represents correct sum
- **Note:** In this problem, the solution is NOT a sequence of actions that transforms the initial state into the goal state; rather, the solution is a goal node that includes an assignment of a digit to each letter in the given problem.
- **APPROACH:** adopt a fixed order, e.g., alphabetical order. A better choice is to do whichever is the most constrained substitution, that is, the letter that has the fewest legal possibilities given the constraints of the puzzle.



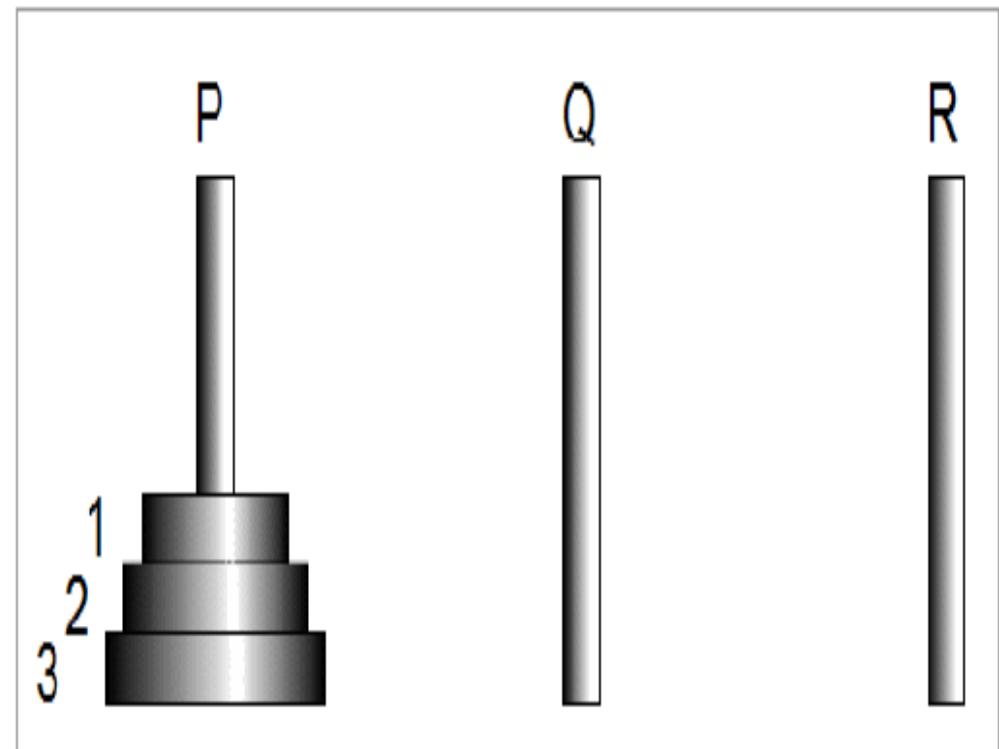
EXAMPLE 8: REMOVE 5 STICKS

- Given the following configuration of sticks, remove exactly 5 sticks in such a way that the remaining configuration forms exactly 3 squares.
- **State:** ?
- **Initial State:** ?
- **Operators:** ?
- **Goal Test:** ?

SOLUTION



EXAMPLE 9:TOWER OF HANOI PROBLEM

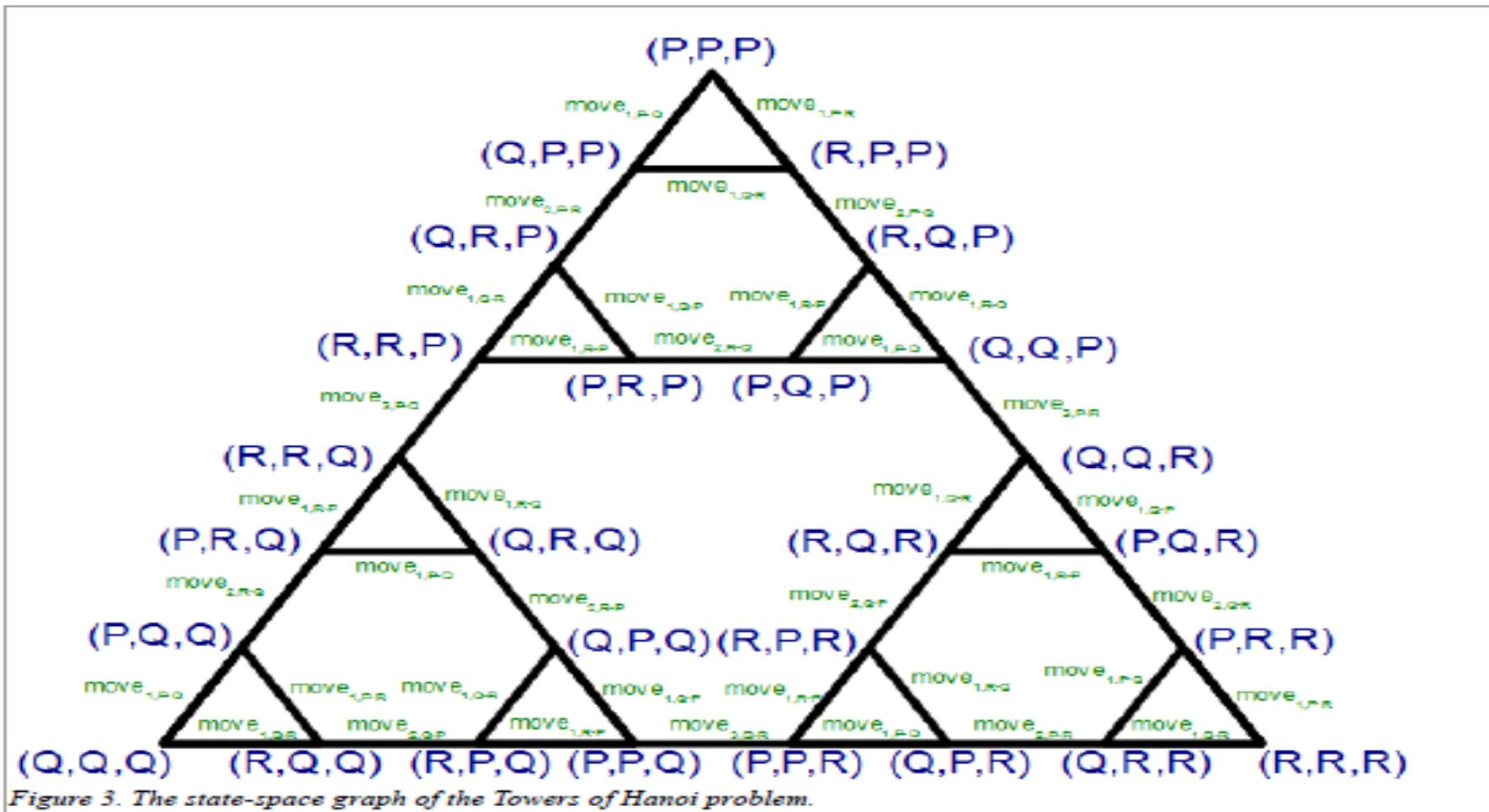


- **PROBLEM DEFINITION:** There are 3 discs with different diameters. We can slide these discs onto 3 perpendicular rods. It's important that if there is a disc under another one then it must be bigger in diameter. We denote the rods with „P”, „Q”, and „R”, respectively. The discs are denoted by „1”, „2”, and „3”, respectively, in ascending order of diameter. We can slide a disc onto another rod
 - 1. if the disc is on the top of its current rod, and
 - 2. the discs on the goal rod will be in ascending order by size after the replacing.
- Our goal is to move all the discs to rod R.

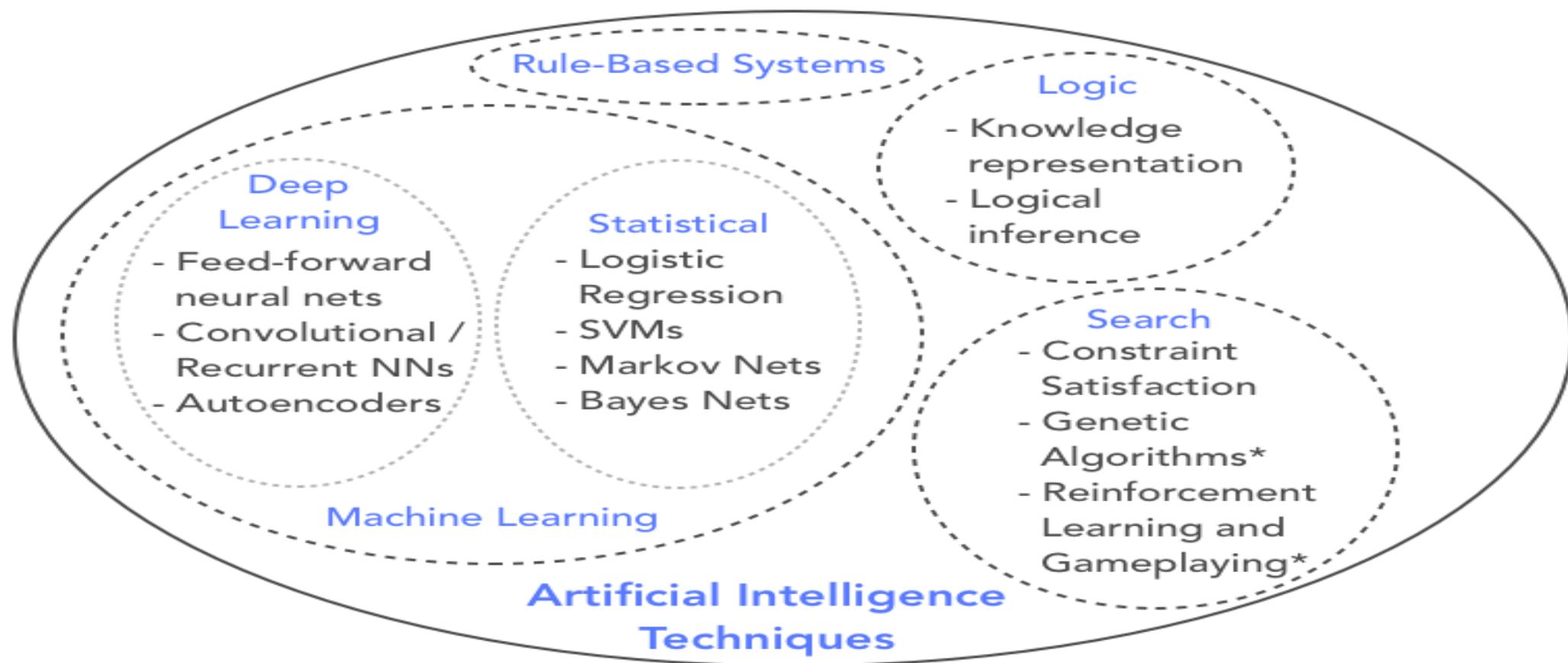
STATE SPACE REPRESENTATION

- **SATATES:** a state is a vector (a_1, a_2, a_3) where a_i is the position of disc i (i.e., either P, Q, or R)
- **Initial state:** Initially, all the discs are on rod P, i. e.: $k=(P, P, P)$
- **Goal state:** The goal is to move all the three discs to rod R. So, in this problem, we have only one goal state, namely: $C=\{R, R, R\}$
- **Set of operators:**

STATE SPACE GRAPH



AI TOOLS AND TECHNIQUES



AI TECHNIQUES

1. Machine Learning

- It is one of the applications of AI where machines are not explicitly programmed to perform certain tasks rather they **learn and improve from experience automatically**. **Deep Learning** is a subset of machine learning based on **artificial neural networks for predictive analysis**. There are various machine learning algorithms such as **Unsupervised Learning**, **Supervised Learning**, and **Reinforcement Learning**. In **Unsupervised Learning**, the algorithm does not use classified information to act on it without any guidance. In **Supervised Learning**, it deduces a function from the training data which consists of a set of an input object and the desired output. **Reinforcement learning** is used by machines to take suitable actions to increase the reward to find the best possibility which should be taken into account.

2. NLP (Natural Language Processing)

- It is the interactions between computers and human language where the computers are programmed to process natural languages. **Machine Learning** is a reliable technology for Natural Language Processing to obtain meaning from human languages. In NLP, the audio of a human talk is captured by the machine. Then the audio to text conversion occurs and then the text is processed where the data is converted into audio. Then the machine uses the audio to respond to humans. Applications of Natural Language Processing can be found in **IVR (Interactive Voice Response)** applications used in call centers, language translation applications like **Google Translate** and word processors such as **Microsoft Word** to check the accuracy of grammar in text. However, the nature of human languages makes the Natural Language Processing difficult because of the rules which are involved in the passing of information using natural language and they are not easy for the computers to understand. So NLP uses algorithms to recognize and abstract the rules of the natural languages where the unstructured data from the human languages can be converted to a format that is understood by the computer.

AI TECHNIQUES

3. Automation and Robotics

- The purpose of Automation is to get the monotonous and repetitive tasks done by machines which also improve productivity and in receiving cost-effective and more efficient results. Many organizations use machine learning, neural networks, and graphs in automation. Such automation can prevent **fraud issues** while **financial transactions online** by using **CAPTCHA technology**. Robotic process automation is programmed to perform high volume repetitive tasks which can adapt to the change in different circumstances.

4. Machine Vision

- Machines can capture visual information and then analyze it. Here cameras are used to capture the visual information, the analog to digital conversion is used to convert the image to digital data and digital signal processing is employed to process the data. Then the resulting data is fed to a computer. In machine vision, two vital aspects are sensitivity, which is the ability of the machine to perceive impulses that are weak and resolution, the range to which the machine can distinguish the objects. The usage of machine vision can be found in signature identification, pattern recognition, and medical image analysis, etc.

PROBLEM SOLVING TYPES

Criteria	Supervised Learning	Unsupervised Learning	Reinforcement Learning
Definition	The machine learns by using labeled data	The machine is trained on unlabeled data without any guidance	An agent interacts with its environment by performing actions & learning from errors or rewards
Type of problems	Regression & classification	Association & clustering	Reward-based
Type of data	Labeled data	Unlabeled data	No predefined data
Training	External supervision	No supervision	No supervision
Approach	Maps the labeled inputs to the known outputs	Understands patterns & discovers the output	Follows the trial-and-error method

TASK DOMAINS OF ARTIFICIAL INTELLIGENCE

Task Domain of Artificial Intelligence		
Mundane (Ordinary) Tasks	Formal Tasks	Expert Tasks
Perception	Mathematics	Engineering
Computer Vision	Geometry	Fault Finding
Speech, Voice	Logic	Manufacturing
Speech, Voice	Integration and Differentiation	Monitoring
Natural Language Processing	Games	Scientific Analysis
Understanding	Go	
Language Generation	Chess (Deep Blue)	
Language Translation	Checkers	
Common Sense	Verification	Financial Analysis
Reasoning	Theorem Proving	Medical Diagnosis
Planning		Creativity
Robotics		
Locomotive		

HISTORY OF AI

- 1923
- Karel Čapek play named “Rossum's Universal Robots” (RUR) opens in London, first use of the word "robot" in English.
- 1943
- Foundations for neural networks laid.
- 1945
- Isaac Asimov, a Columbia University alumni, coined the term Robotics.
- 1950
- Alan Turing introduced Turing Test for evaluation of intelligence and published Computing Machinery and Intelligence.
- Claude Shannon published Detailed Analysis of Chess Playing as a search.
- 1956
- John McCarthy coined the term Artificial Intelligence. Demonstration of the first running AI program at Carnegie Mellon University.

HISTORY OF AI

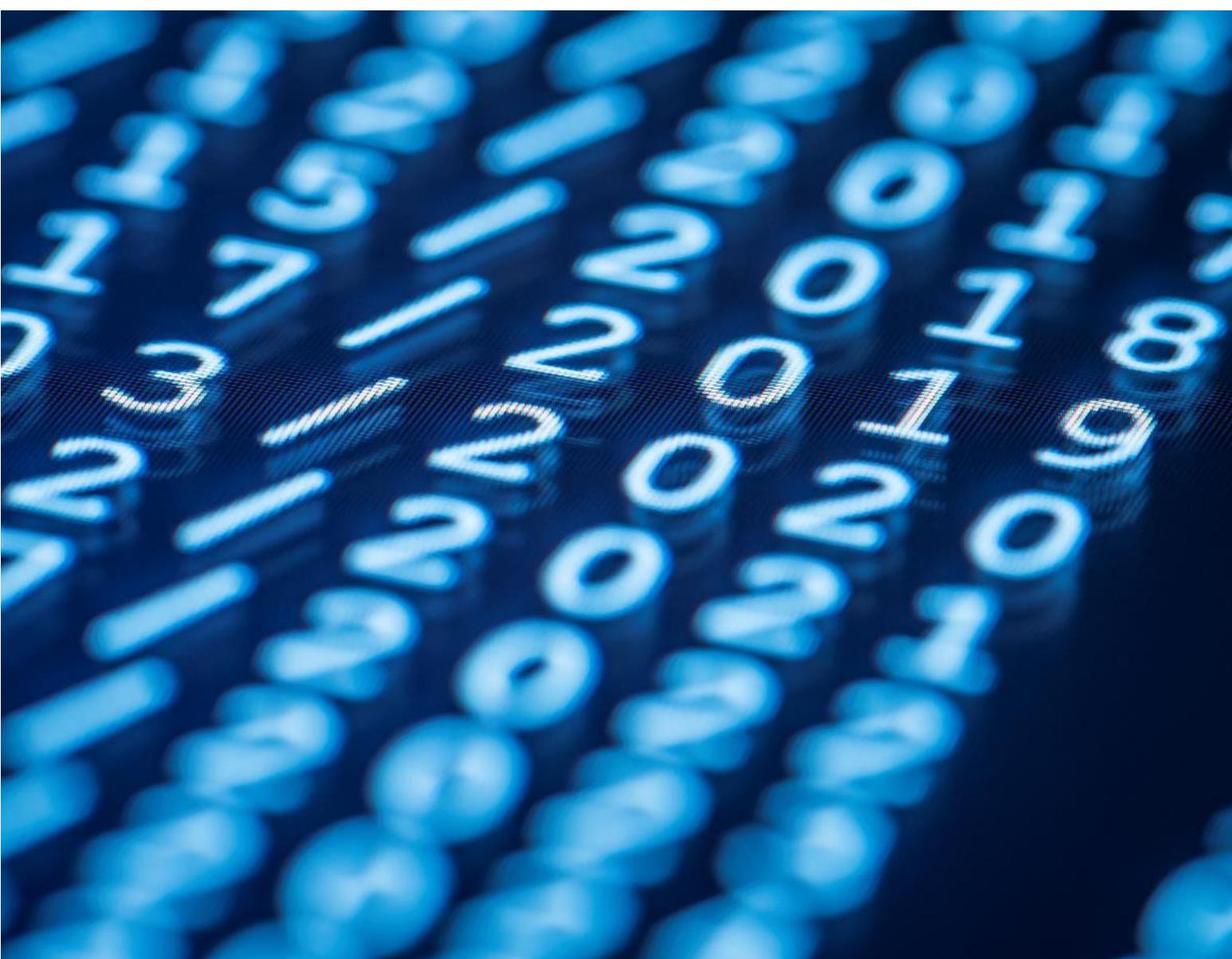
- 1958
- John McCarthy invents LISP programming language for AI.
- 1964
- Danny Bobrow's dissertation at MIT showed that computers can understand natural language well enough to solve algebra word problems correctly.
- 1965
- Joseph Weizenbaum at MIT built ELIZA, an interactive program that carries on a dialogue in English.
- 1969
- Scientists at Stanford Research Institute Developed Shakey, a robot, equipped with locomotion, perception, and problem solving.
- 1973
- The Assembly Robotics group at Edinburgh University built Freddy, the Famous Scottish Robot, capable of using vision to locate and assemble models.
- 1979
- The first computer-controlled autonomous vehicle, Stanford Cart, was built.

HISTORY OF AI

- 1985
- Harold Cohen created and demonstrated the drawing program, Aaron.
- 1990
- Major advances in all areas of AI –
- Significant demonstrations in machine learning
- Case-based reasoning
- Multi-agent planning
- Scheduling
- Data mining, Web Crawler
- natural language understanding and translation
- Vision, Virtual Reality
- Games

HISTORY OF AI

- 1997
- The Deep Blue Chess Program beats the then world chess champion, Garry Kasparov.
- 2000
- Interactive robot pets become commercially available.
- MIT displays Kismet, a robot with a face that expresses emotions.
- The robot Nomad explores remote regions of Antarctica and locates meteorites.



THANK YOU

anooha@somaiya.edu



AI SEARCH TECHNIQUES

PREPARED BY
- ANOOJA JOY

PROBLEM SOLVING

- Problem solving is a process of generating solutions from observed data. It is selection of relevant set of states to consider and feasible set of operators for moving from one state to another state.
- A ‘problem’ is characterized by a set of goals, a set of objects, and a set of operations. Goals help in organizing behavior of AI agent by limiting the objectives that the agent is trying to achieve
- **Goal formulation** is the first step in problem solving.
- **Problem formulation** is the process of deciding what actions and states to consider and follows goal formulation.
- **A ‘problem space’** is an abstract space. A solution is a combination of operations and objects that achieve the goals.
- A ‘search’ refers to the search for a solution in a problem space. The process of generating sequence of states by applying operators to initial state and checking which sequence has reached goal state.
- Search proceeds with different types of ‘search control strategies’. A search algorithm takes a problem as input and returns a solution in the form of an action sequence.

PROBLEM SOLVING AGENTS

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

  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then do
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
  action  $\leftarrow$  FIRST(seq)
  seq  $\leftarrow$  REST(seq)
  return action
```

SEARCH ALGORITHMS



WHY SEARCH IS IMPORTANT?

Problem-solving agents are the **goal-based agents** and use atomic representation. Agent will have set of **actions to achieve goals**.

Rational agents or **Problem-solving agents** in AI mostly used these **search strategies** or algorithms to solve a specific problem and provide the best result.

Search techniques are universal problem-solving methods and is a key computational mechanism in many AI agents

Problem: Find a path from **START** to **GOAL** and find the minimum number of transitions

A search problem finds **sequence of actions** which transforms agent from **initial state to goal state**

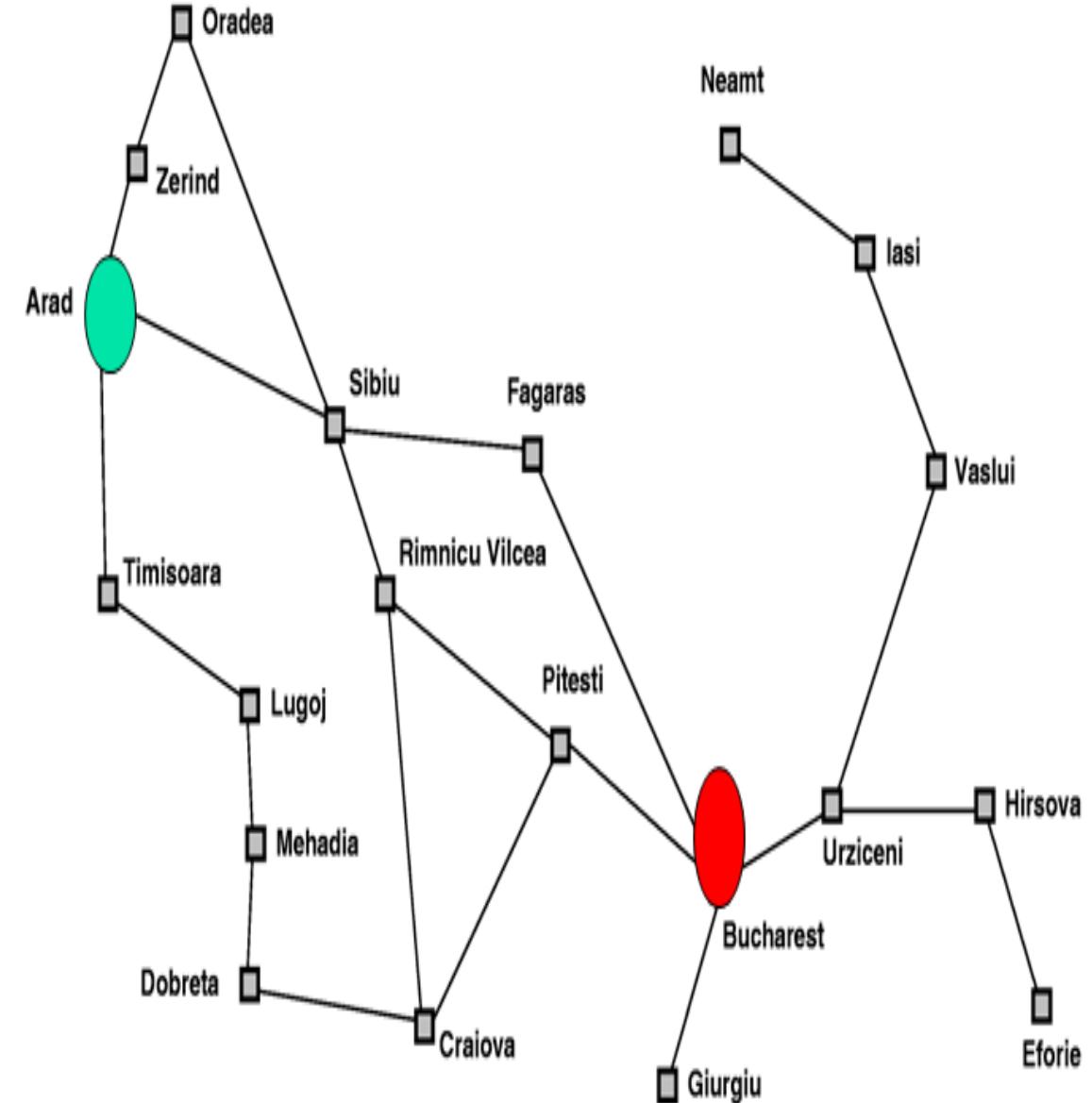
A search problem is represented using **directed graph** where **edges** are the **operators** and **nodes** are **states**. Searching process checks **allowable operation from one state to other states**.

ELEMENTS OF PROBLEM SOLVING

- **Problems** can be represented as **set of states** and how **state changes happens by application of some rules**.
- A problem space **encompasses all valid states that can be generated by the application of any combination of operators on any combination of objects**. The problem space may contain one or more solutions.
- **State space of a problem** is the set of all states reachable from the initial state by any sequence of actions. A path in the state space is simply any sequence of actions leading from one state to another.
- **Problem State space representation have following things:**
 - **State(S)** is an abstract representation of agents environment.
 - **Initial State(S₀)**: description of starting configuration of agent.
 - **Action/Operator(A: S->Set of actions)**: takes agent from one state to another state.
 - **Plan**: sequence of actions
 - **Goal**: description of desirable set of states. Its often specified by **Goal test** which any goal must satisfy.
 - **Goal test**: A function that looks at the current state returns whether or not it is the goal state.
 - **Path Cost**: A path cost function is a function that assigns a cost to a path.
 - **Solution**: a path from initial state to goal state

ROUTE FINDING/ MAP SEARCHING

- Imagine an agent on touring holiday in the city of Arad in Romania. The agent has a ticket to fly to Bucharest tomorrow so he can go to his home country from there. The ticket is nonrefundable, the agent's visa is about to expire after tomorrow, there are no seats available for six weeks. How to reach Bucharest?



STATE SPACE REPRESENTATION

- **State space:** Cities
- **Successor function:** Roads: Go to adjacent city with cost = distance

$S(x)$ = set of action–state pairs

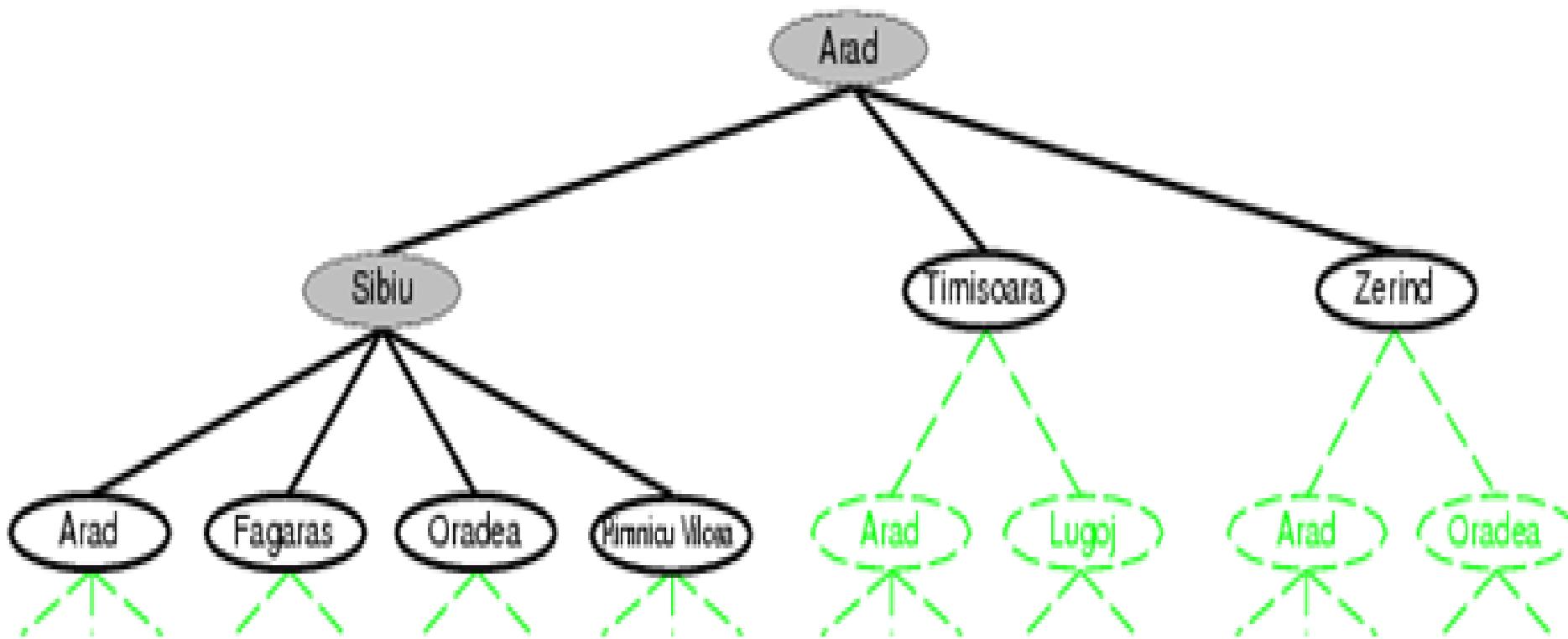
e.g., $S(Arad) = \{<Arad \rightarrow Zerind, Zerind>, \dots\}$

- **Start state:** Arad
- **Goal test:** Is state == Bucharest?
- **Solution?**

ABSTRACTION/MODELING

- **State space** must be **abstracted** for problem solving
 - n(Abstract) state = set of real states
 - n(Abstract) action = complex combination of real actions
- *Abstraction is the process of removing irrelevant detail to create an abstract representation: ``high-level'', ignores irrelevant details*
- Abstraction is critical for automated problem solving as good abstractions retain all important details.
- Abstraction is to create an approximate, simplified, model of the world for the computer to deal with: real-world is too detailed to model exactly

STATE SPACE GRAPH

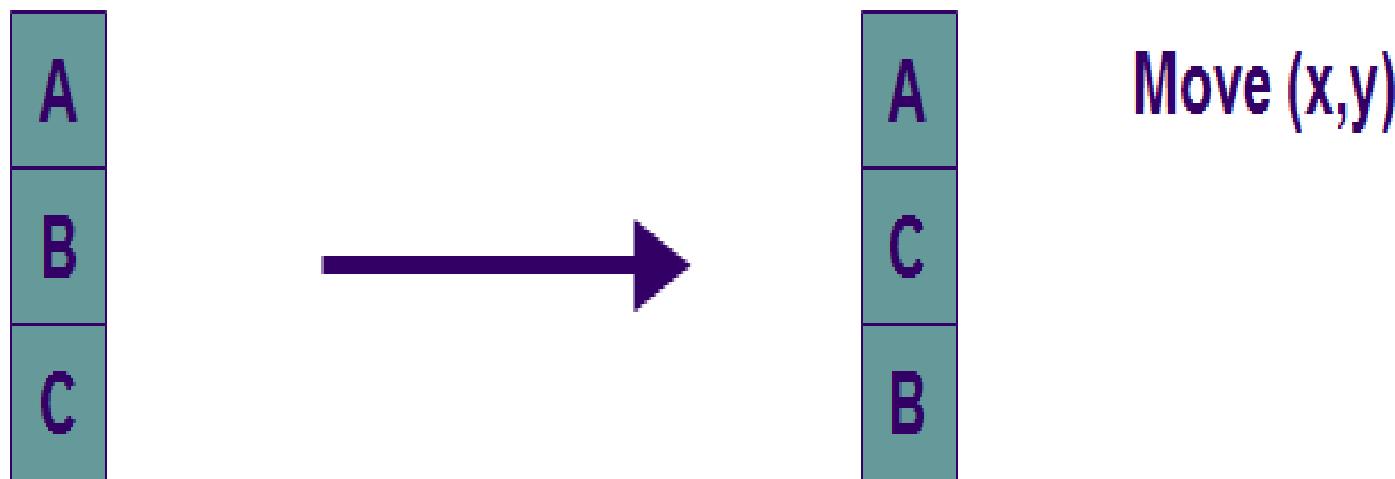


GENERAL SEARCH ALGORITHM

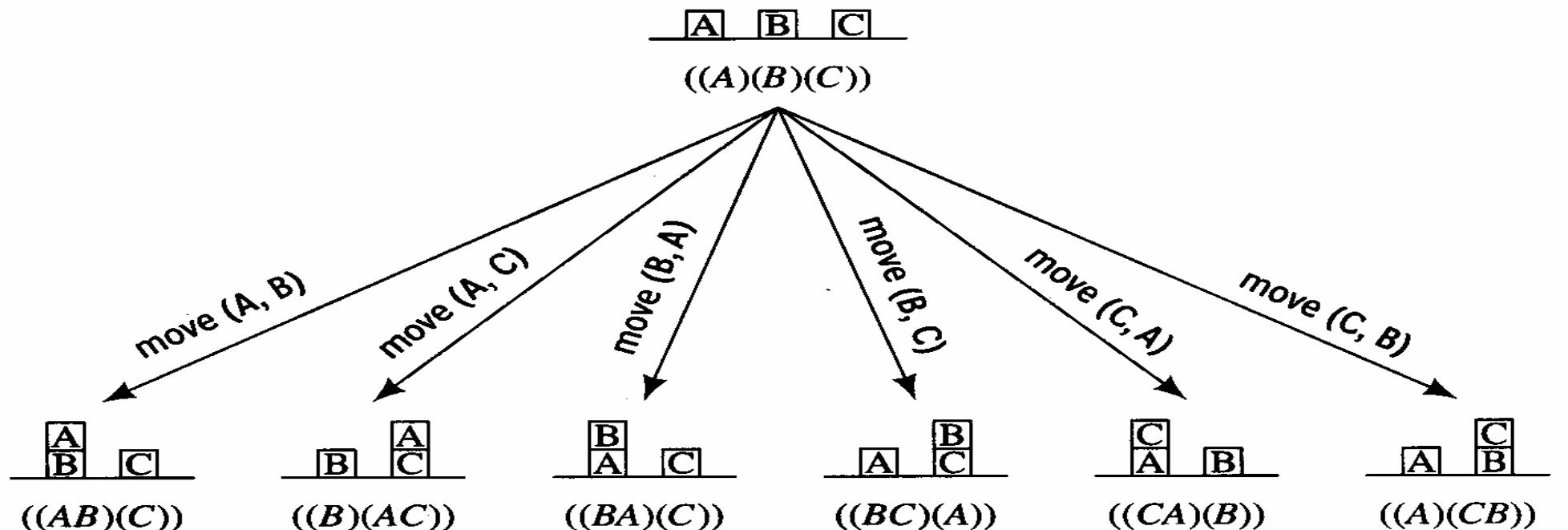
```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

ROBOT BLOCK WORLD PROBLEM

- Given a set of blocks in a certain configuration,
- Move the blocks into a goal configuration.
- Example : (c,b,a) \rightarrow (b,c,a)



STATE SPACE GRAPH



Effects of Moving a Block

STATE SPACE GRAPHS

State space graph: A mathematical representation of a search problem

Nodes are (abstracted) world configurations

Arcs represent successors (action results)

Goal test is a set of goal nodes (maybe only one)



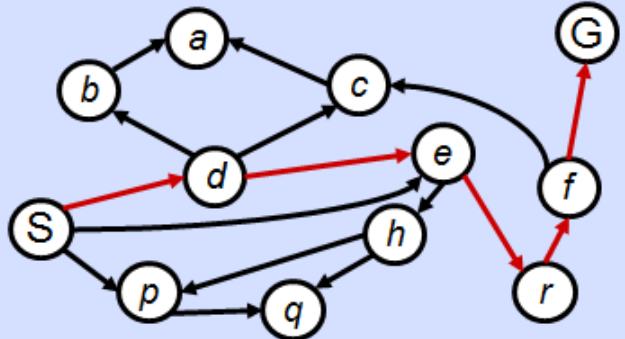
In a state space graph, each state occurs only once!



We can rarely build this full graph in memory (it's too big), but it's a useful idea

STATE SPACE GRAPHS VS. SEARCH TREES

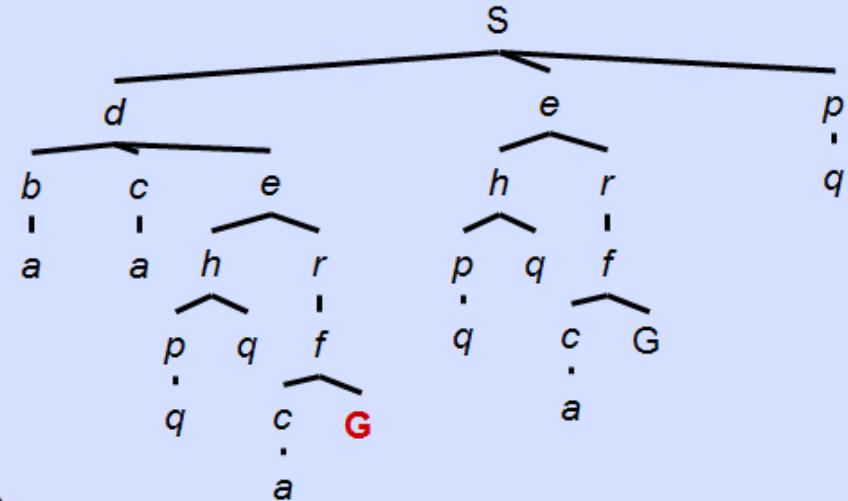
State Space Graph



*Each NODE in in
the search tree is
an entire PATH in
the state space
graph.*

*We construct both
on demand – and
we construct as
little as possible.*

Search Tree

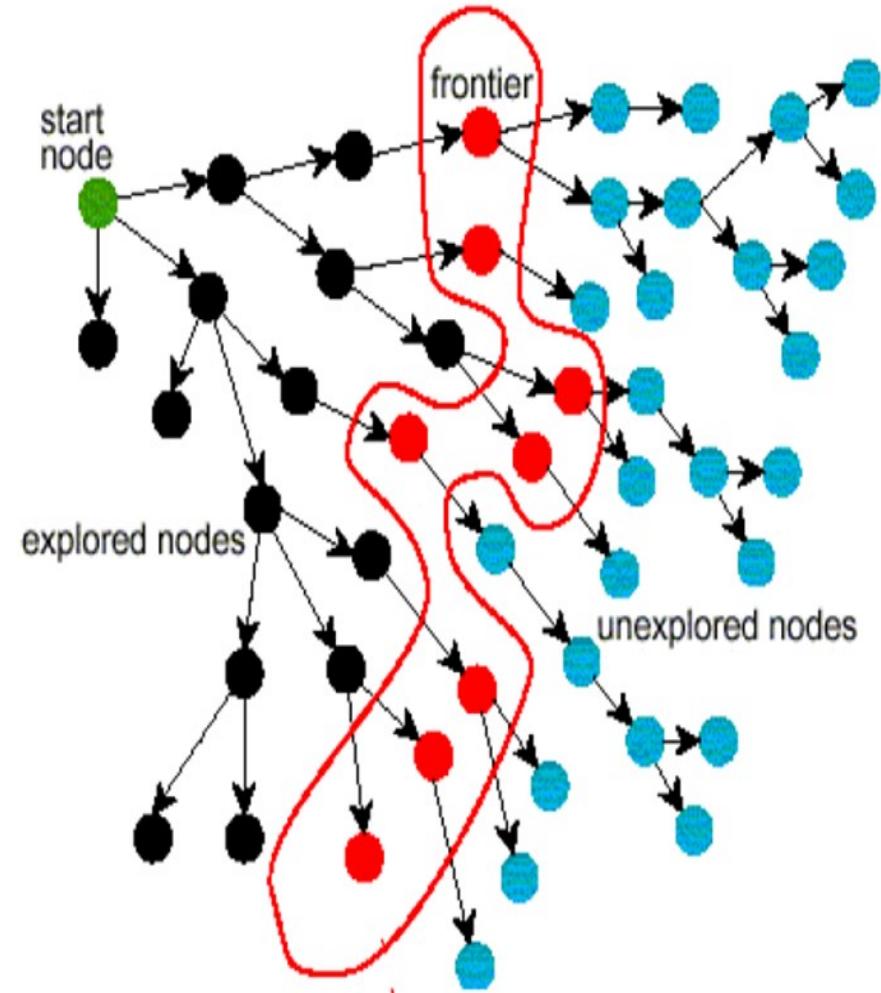


Lots of repeated structure in the search tree!

CONSTITUENTS OF SEARCH ALGORITHMS

- A problem **graph**, containing the start node S and the goal node G.
- A **strategy**, describing the way the graph will be traversed to get to G .
- **Frontier**: - The set of paths which could be explored next. The way in which the frontier is expanded defines the search strategy
- A **fringe**, which is a data structure used to store all the possible states (nodes) that you can go from the current states

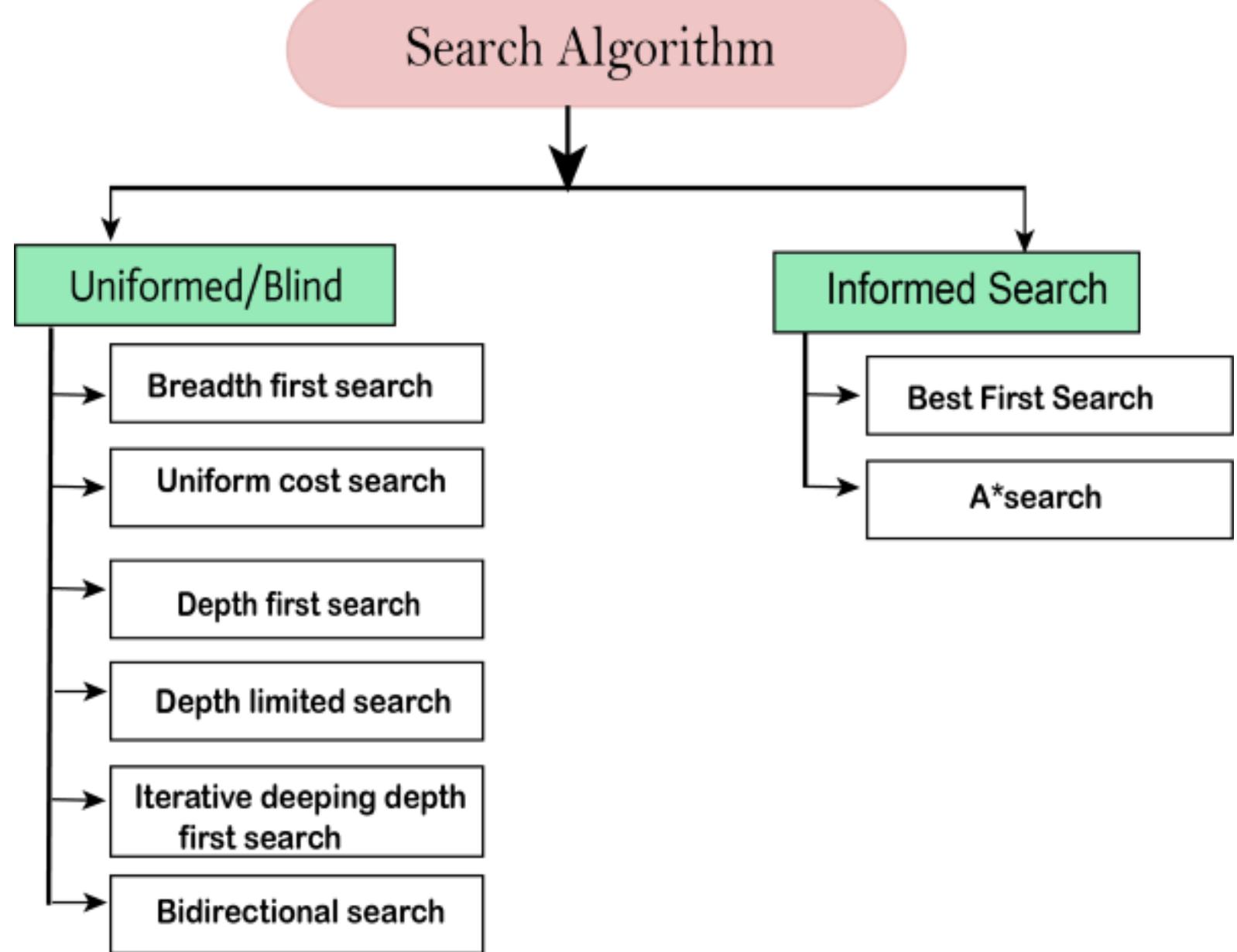
Problem Solving by Graph Searching



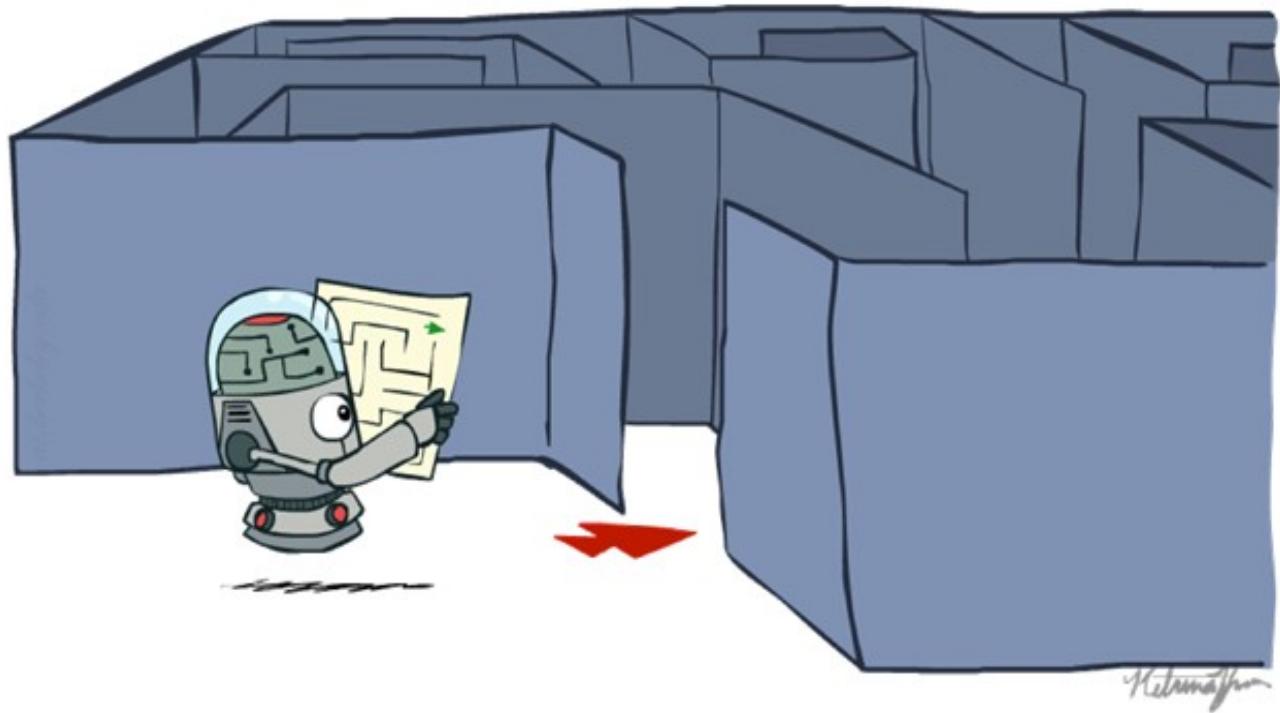
- The **total cost** of the search is the sum of the **path cost** and the **search cost**
- **Search cost** associated is the **time** and **memory** required to find a solution
 - **Time complexity**—number of nodes generated-expanded
 - **Space complexity**—maximum number of nodes in memory.
- Time and space complexity are measured in terms of
 - **b**—maximum **branching factor** of the search tree
 - **d**—**depth** of the **least-cost solution**
 - **m**—maximum depth of the state space (may be ∞)

MEASURING PROBLEM-SOLVING PERFORMANCE

TYPES OF SEARCH ALGORITHMS



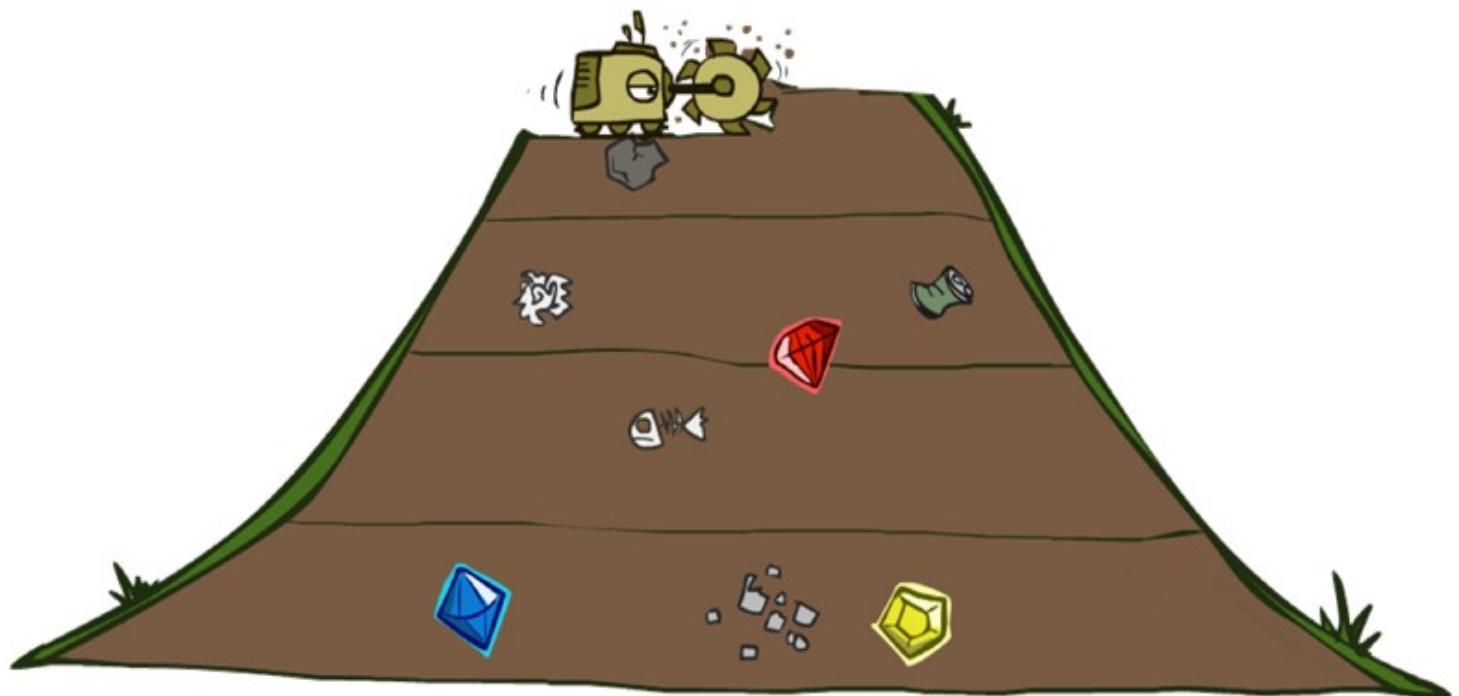
UNINFORMED SEARCH STRATEGIES



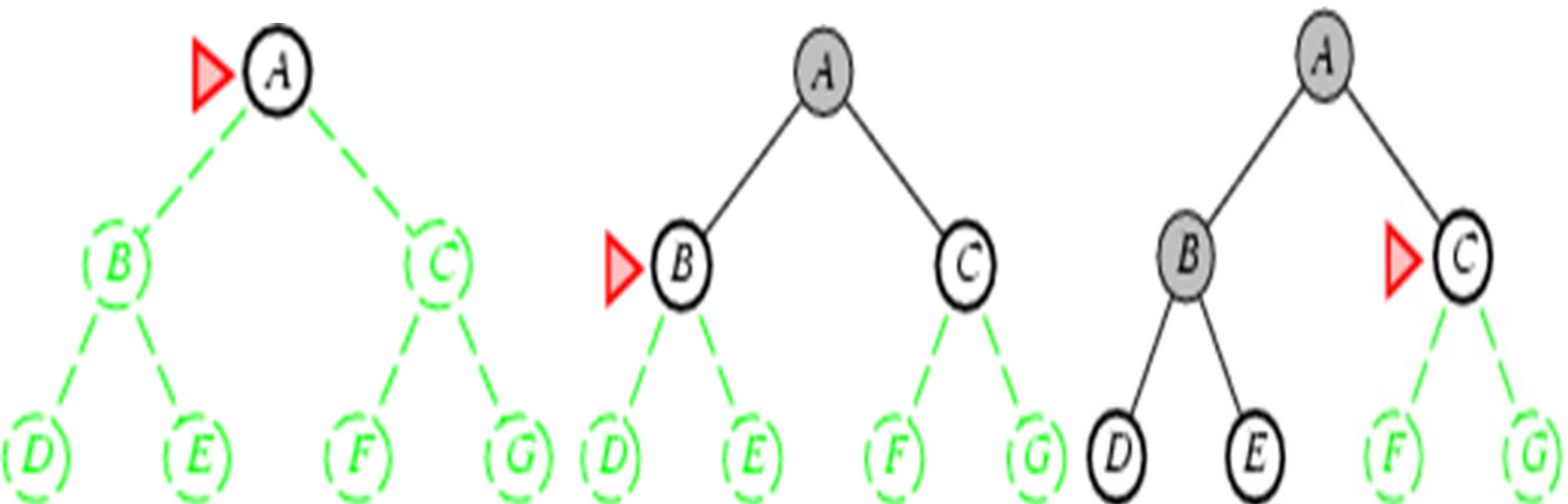
UNINFORMED SEARCH

- Uninformed search have no information about the number of steps or the path cost from the current state to the goal—all they can do is distinguish a goal state from a nongoal state.
- Uninformed search is also sometimes called blind search.
- The uninformed search **does not contain any domain knowledge** such as **closeness, the location of the goal ie** no additional information on the goal node other than the one provided in the problem definition.
- **Uninformed (blind) search strategies use only the information available in the problem definition.** All they can do is generate successors and differentiate between goal and non-goal states.
- Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goa.
- It operates in a **brute-force way** as it only includes information about **how to traverse the tree and how to identify leaf and goal nodes.**

BREADTH FIRST SEARCH



BREADTH FIRST SEARCH



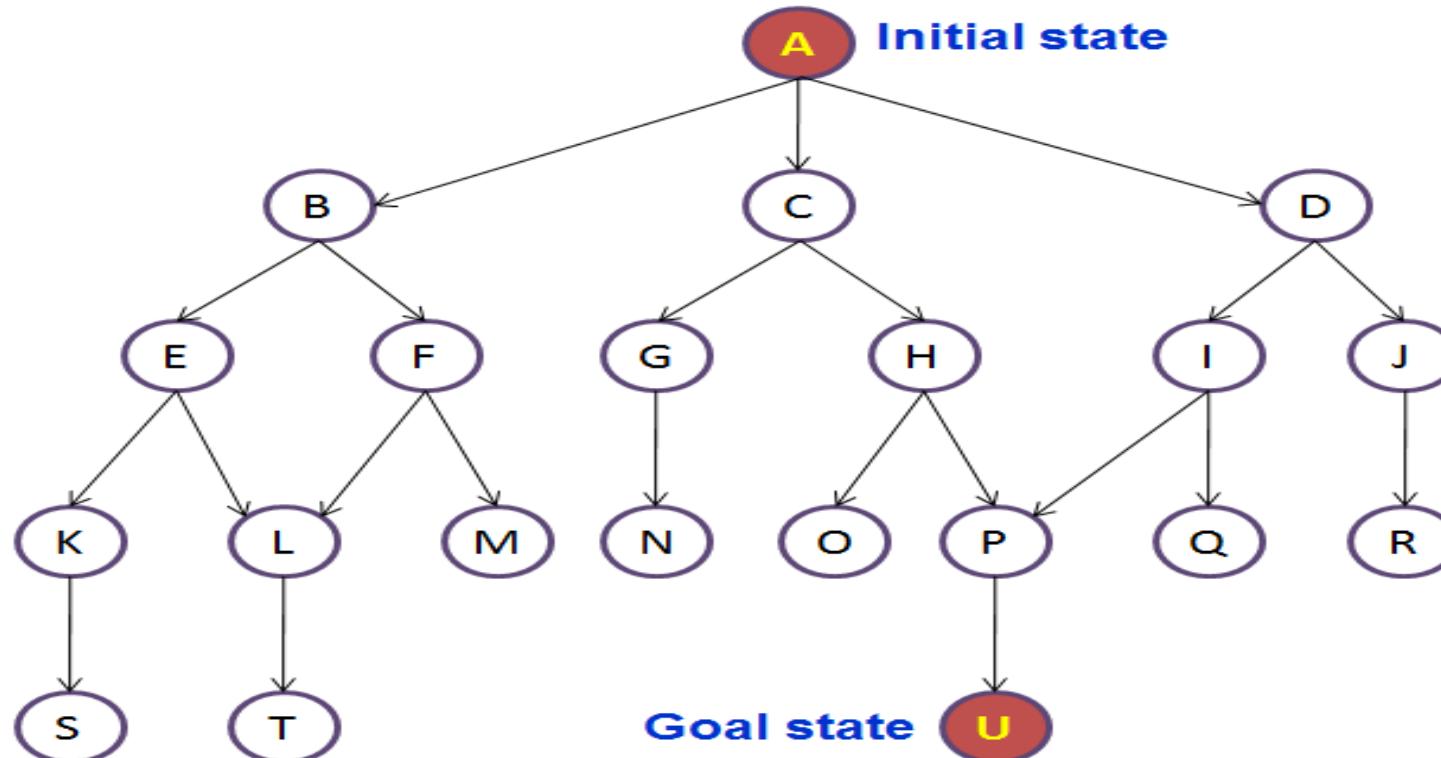
OPEN AND CLOSED LIST IN SEARCH STRATEGIES

- During search, **a node** can be in one of the **three categories**:
 1. **Not generated yet** (has not been made explicit yet)
 2. **OPEN**: generated but not expanded
 3. **CLOSED**: expanded
- **Search strategies differ** mainly on **how to select an OPEN node for expansion** at each step of search

BFS

Breadth First Search examines the nodes in the following order:

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U

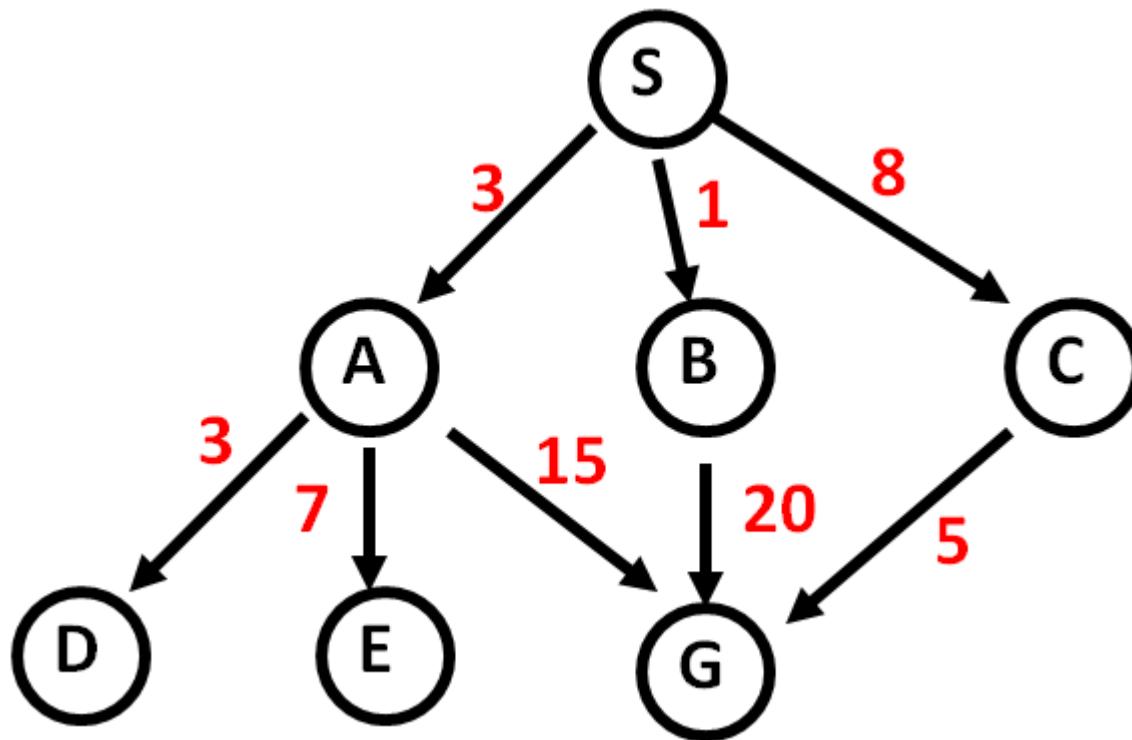


OPEN	CLOSE
A	
BCD	A
CDEF	BA
DEFGH	CBA
EFGHIJ	DCBA
FGHIJKL	EDCBA
GHIJKLMNOP	FEDCBA
HJKLMNOPMN	GFEDCBA
IJKLMNOPQ	HGFEDCBA
JJKLMNOPQ	IHGfedcba
KLMNOPQR	Jihgfedcba
LMNOPQRS	Kjihgfedcba
MNOPQRST	Lkjihgfedcba
NOPQRST	mlkjihgfedcba
OPQRST	nmlkjihgfedcba
PQRST	onmlkjihgfedcba
QRSTU	ponmlkjihgfedcba
RSTU	qpomnlkjihgfedcba
STU	rqpomnlkjihgfedcba
TU	srqpomnlkjihgfedcba
U	tsrqponmlkjihgfedcba
U	hold

BREADTH FIRST SEARCH

- Root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on.
- All the nodes at depth d in the search tree are expanded before the nodes at depth $d + 1$.
- Expand the shallowest unexpanded node
- Implementation: frontier/fringe is a **FIFO queue**, i.e., new successors go at end
- Problems involving search of **exponential complexity** (like chess game) **cannot be solved** by uninformed methods since the size of the data being too big.

BFS EXAMPLE



BFS

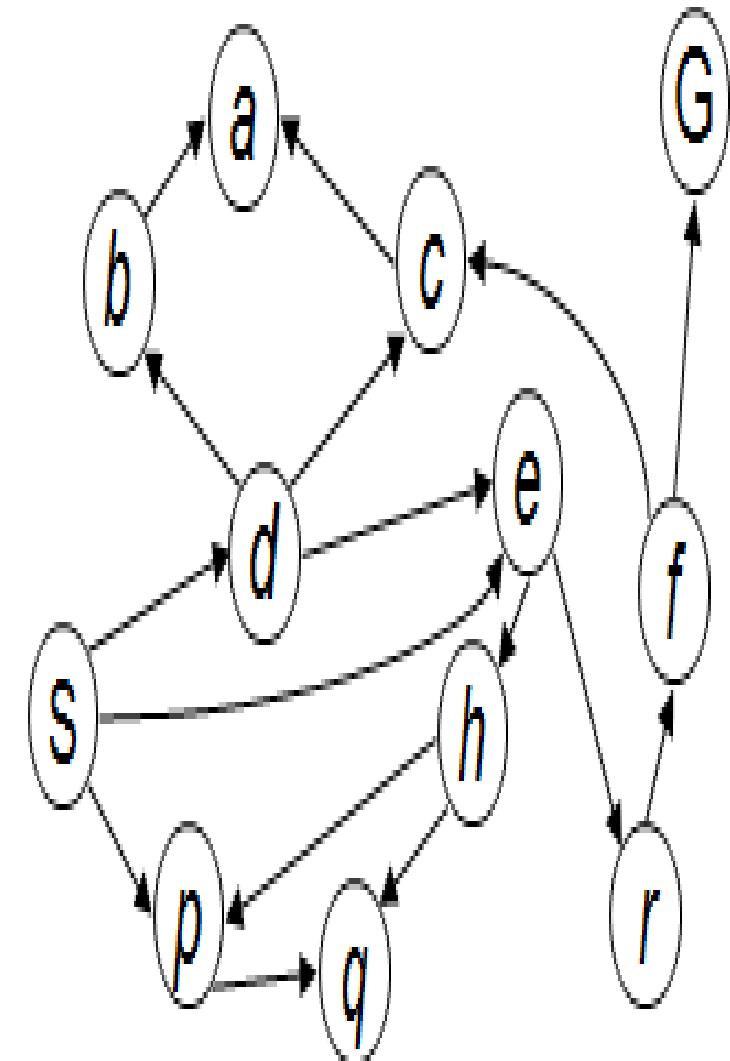
	Expanded node	Nodes list
		{ S ⁰ }
1	S ⁰	{ A ³ B ¹ C ⁸ }
2	A ³	{ B ¹ C ⁸ D ⁶ E ¹⁰ G ¹⁸ }
3	B ¹	{ C ⁸ D ⁶ E ¹⁰ G ¹⁸ G ²¹ }
4	C ⁸	{ D ⁶ E ¹⁰ G ¹⁸ G ²¹ G ¹³ }
5	D ⁶	{ E ¹⁰ G ¹⁸ G ²¹ G ¹³ }
6	E ¹⁰	{ G ¹⁸ G ²¹ G ¹³ }
7	G ¹⁸	{ G ²¹ G ¹³ }

- Solution path found is S A G , cost 18
- Number of nodes expanded (including goal node) = 7

BFS

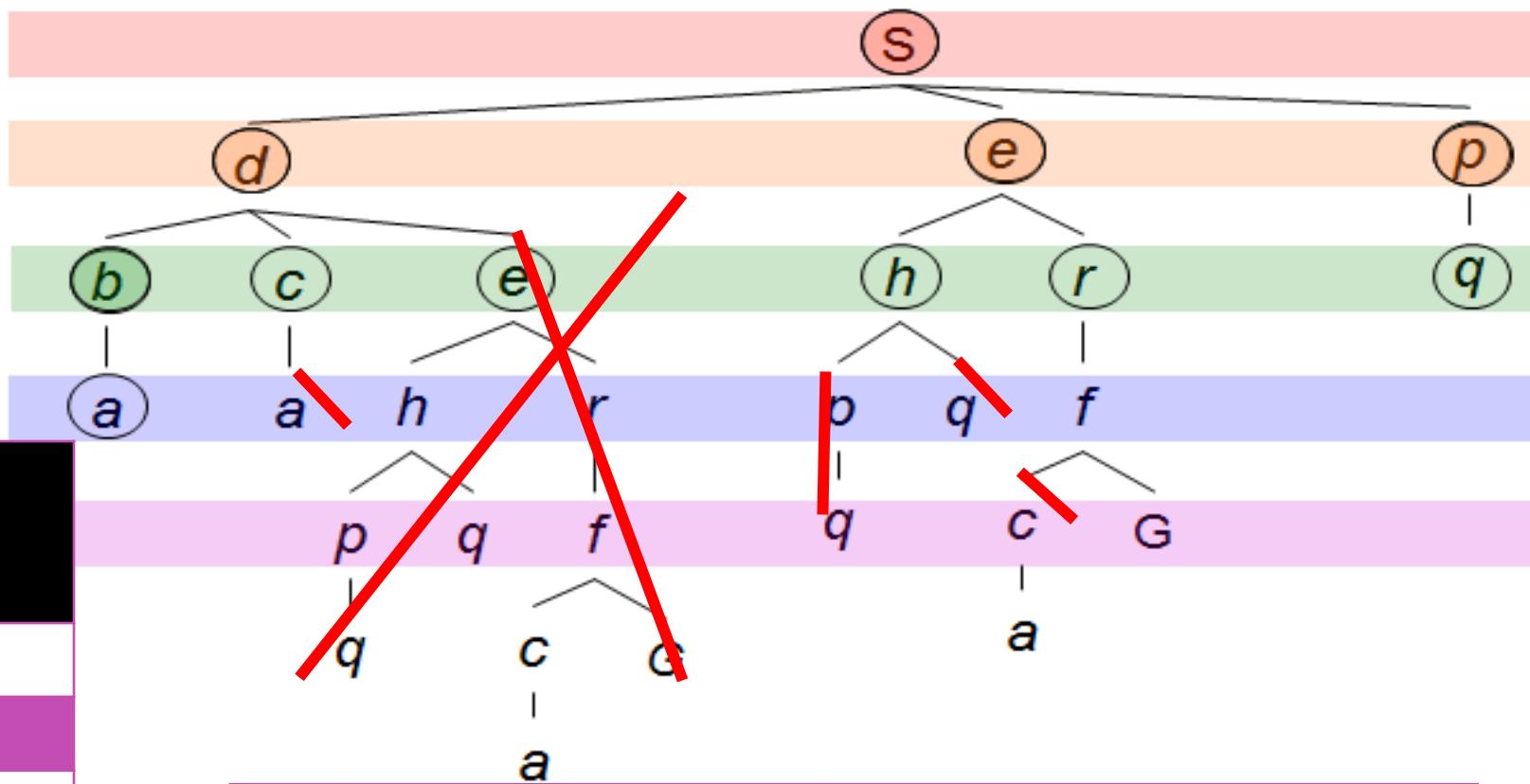
*Strategy: expand a
shallowest node first*

*Implementation: Fringe
is a FIFO queue*



BFS

Search
Tiers



	Expanded node	Nodes list/ open list
		{ S ⁰ }
1	s ⁰	{ d ¹ e ¹ p ¹ }
2	d ¹	{ e ¹ p ¹ b ² c ² e ² }
3	e ¹	{ p ¹ b ² c ² e ² h ³ r ³ }
4	p ¹	{ b ² c ² e ² h ² r ² q ² }
5	b ²	{ c ² e ² h ² r ² q ² a ³ }
6	c ²	{ e ² h ² r ² q ² a ³ a ³ }
7	h ²	{ r ² q ² a ³ a ³ p ³ q ³ }

	Expanded node	Nodes list/ open list
8	r ²	{ q ² a ³ a ³ p ³ q ³ f ³ }
9	q ²	{ a ³ a ³ p ³ q ³ f ³ }
10	a ³	{ a ³ p ³ q ³ f ³ }
11	f ³	{ c ⁴ g ⁴ }
12	g ⁴	{ }

PROPERTIES OF BREADTH-FIRST SEARCH

Complete??

- Yes (If there is a solution that exist, breadth-first search is guaranteed to find it, and if there are several solutions, breadth-first search will always find the shallowest goal state first. It fails if graph or tree is cyclic and if child node have infinite successors.)

Space Complexity?

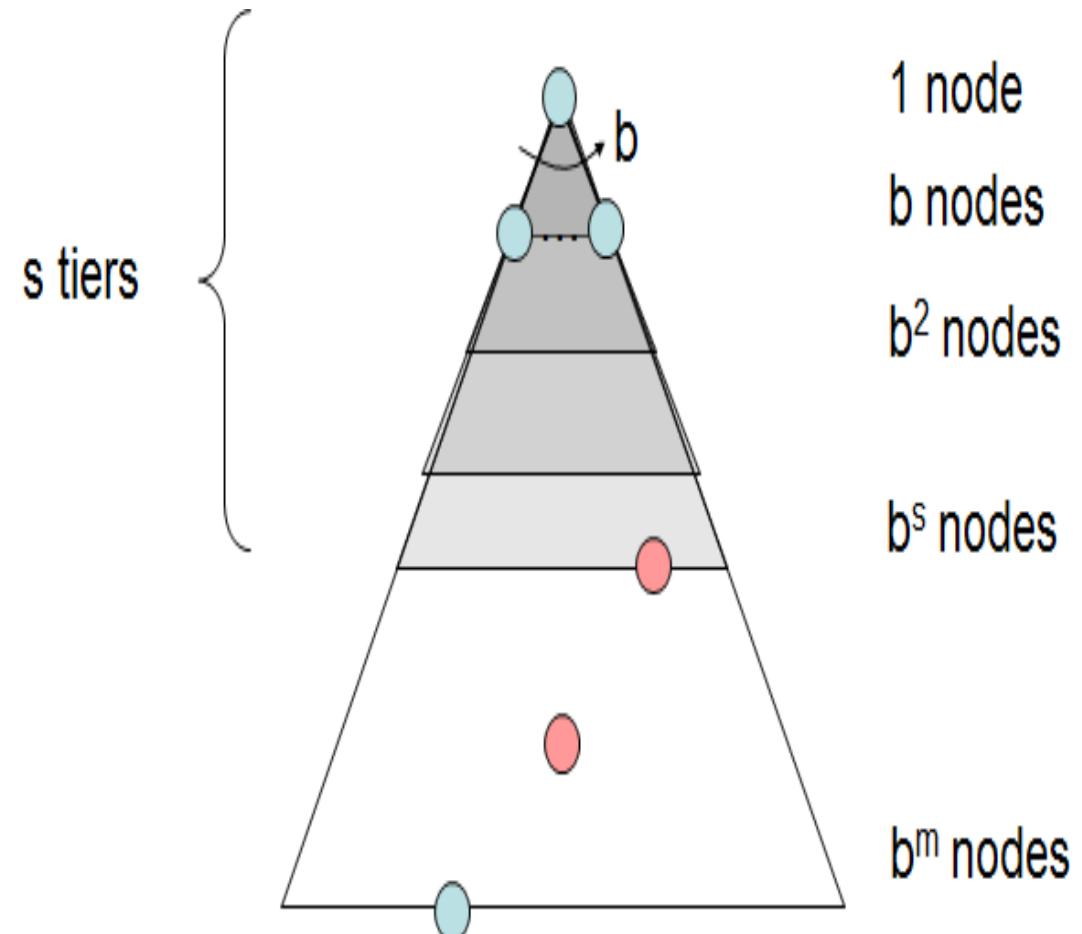
- Has roughly the last tier, so $O(b^s)$. Keeps every node in memory. s is the depth of the shallowest solution and b is the branching factor (i.e., number of children per node) at each node. Space is the big problem: it can easily generate 1M nodes/second so after 24hrs it has used 86,000GB (and then it has only reached depth 9 in the search tree)

Time??

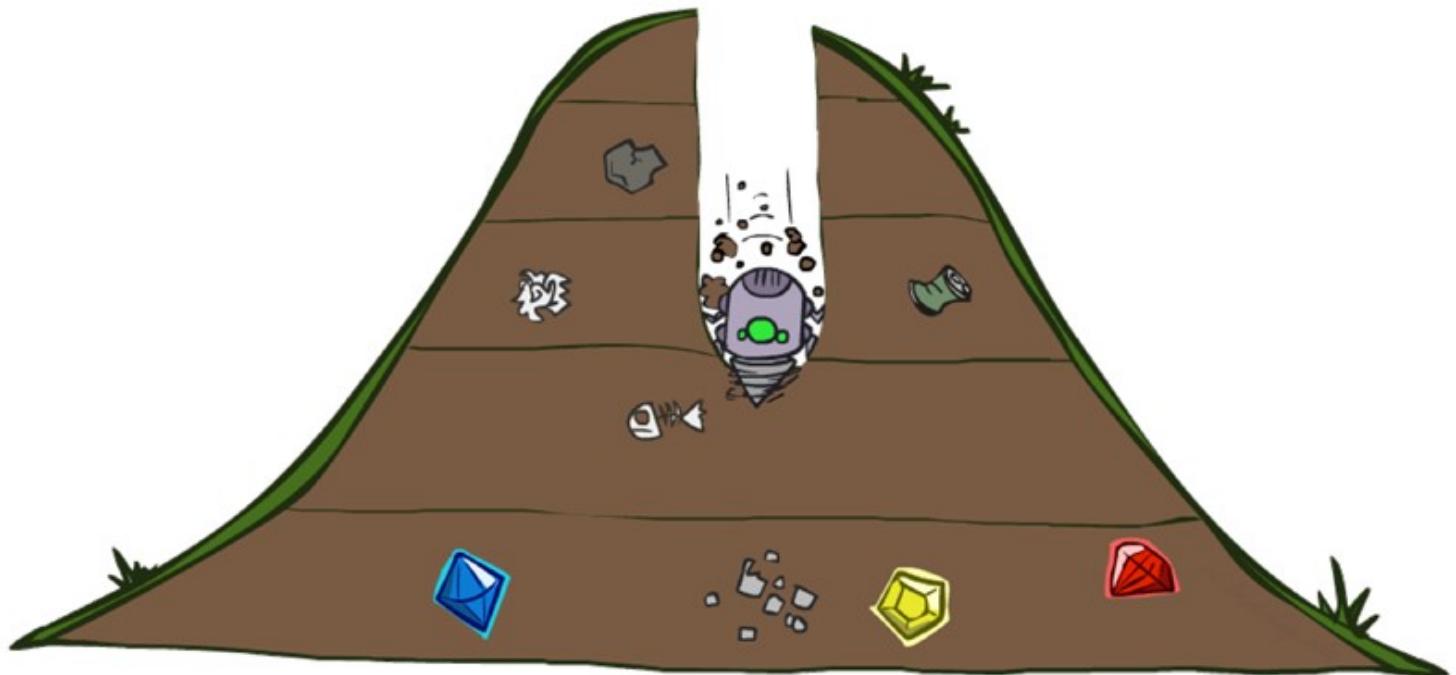
$$1 + b + b^2 + b^3 + \dots + b^s = b(b^{s-1}) = O(b^s)$$

Is it optimal?

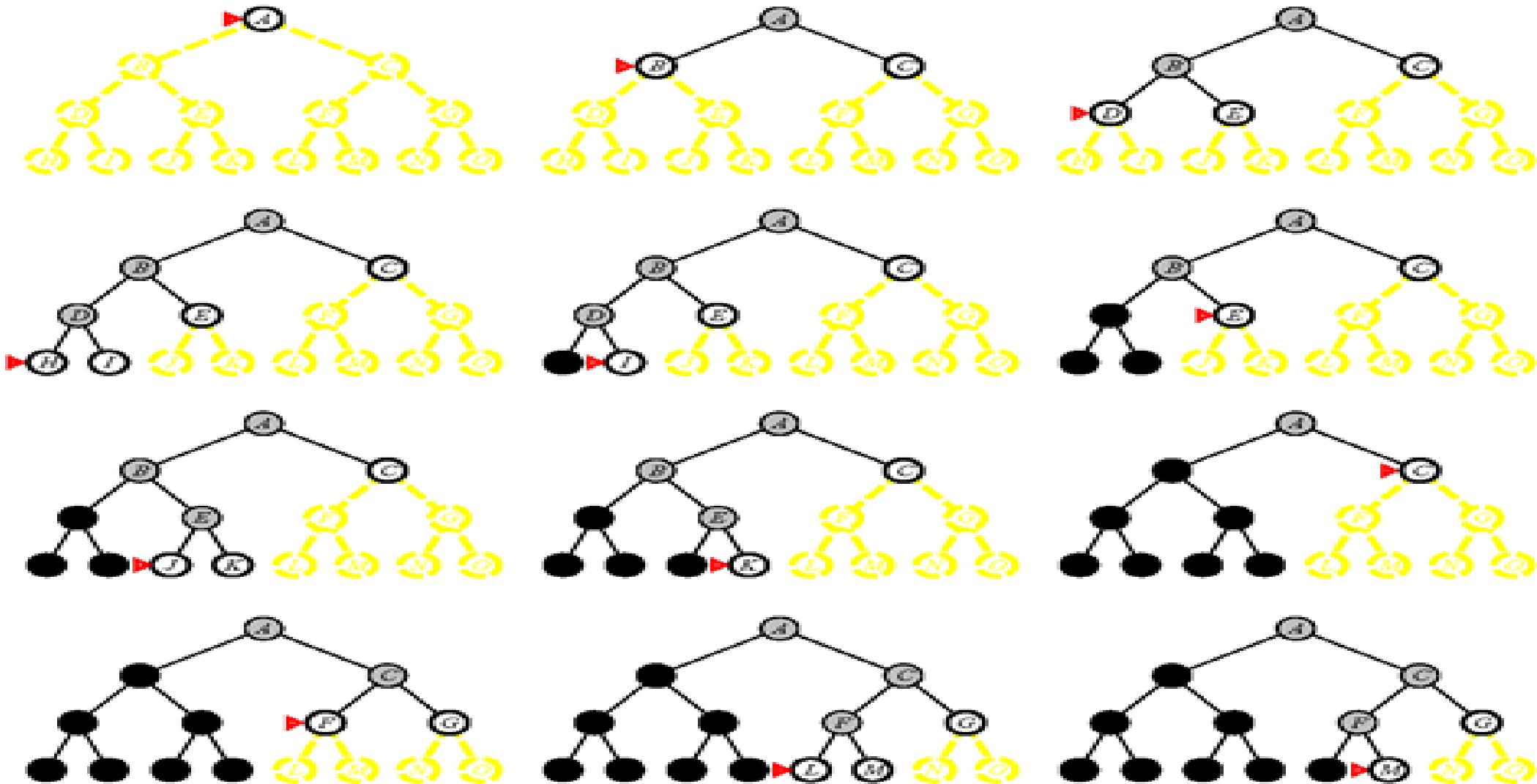
- Yes. it is optimal provided the path cost is a nondecreasing function of the depth of the node. Only if costs are all 1 (for all operators have the same cost)



DEPTH FIRST SEARCH



DEPTH FIRST SEARCH



DEPTH FIRST SEARCH

- Depth-first search always expands one of the **nodes at the deepest level of the tree**.
- Enqueue nodes on nodes in **LIFO** (last-in, first-out) order. That is, nodes used as a **stack data structure to order nodes**.
- Depth-first search has very **modest memory requirements**.
- **Implementation:** frontier = **LIFO queue**, i.e., put successors at front
- **May not terminate** without a "depth bound," i.e., cutting off search below a fixed depth D

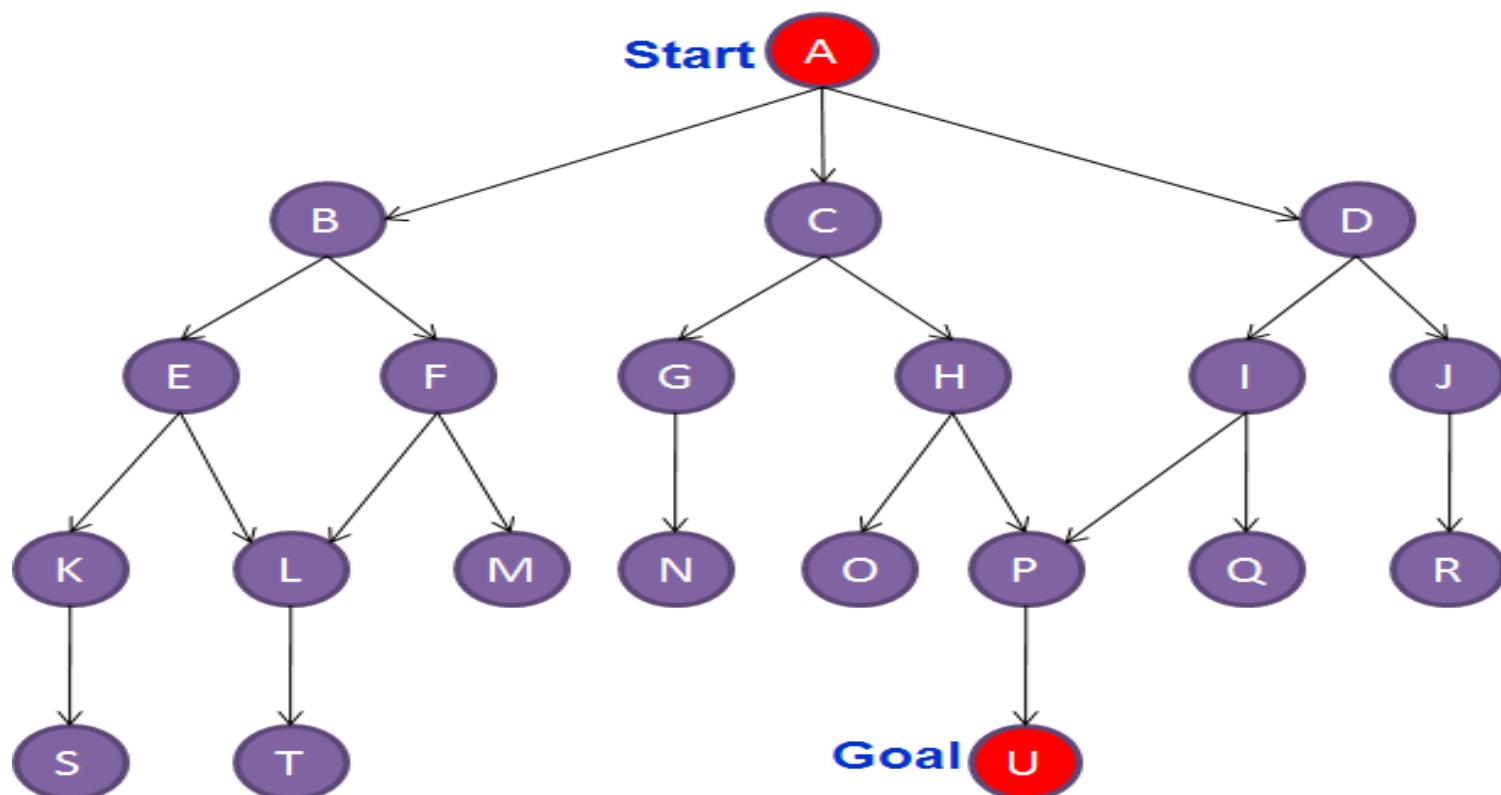
Algorithm outline:

1. Always select from the OPEN the node with the greatest depth for expansion, and put all newly generated nodes into OPEN
2. OPEN is organized as **LIFO** (last-in, first-out) list.
3. Terminate if a node selected for expansion is a goal

DFS

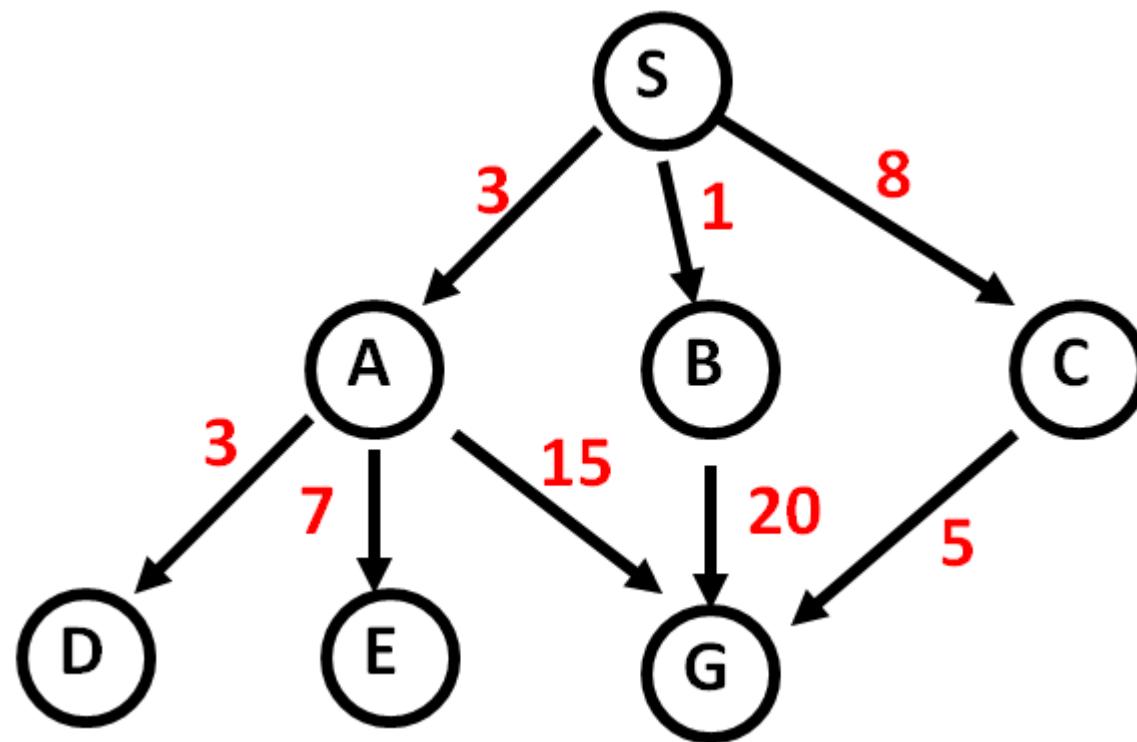
Depth First Search examines the nodes in the following order:

A, B, E, K, S, L, T, F, M, C, G, N, H, O, P, U, D, I, Q, J, R



OPEN	CLOSED
A	
B C D	A
E F C D	B A
K L F C D	E B A
S L F C D	K E B A
L F C D	S K E B A
T F C D	L S K E B A
F C D	T L S K E B A
M C D	F T L S K E B A
C D	M F T L S K E B A
G H D	C M F T L S K E B A
N H D	G C M F T L S K E B A
H D	N G C M F T L S K E B A
O P D	H N G C M F T L S K E B A
P D	O H N G C M F T L S K E B A
U D	P O H N G C M F T L S K E B A

DFS EXAMPLE



DFS

	Expanded node	Nodes list
		{ S ⁰ }
1	S ⁰	{ A ³ B ¹ C ⁸ }
2	A ³	{ D ⁶ E ¹⁰ G ¹⁸ B ¹ C ⁸ }
3	D ⁶	{ E ¹⁰ G ¹⁸ B ¹ C ⁸ }
4	E ¹⁰	{ G ¹⁸ B ¹ C ⁸ }
5	G ¹⁸	{ B ¹ C ⁸ }

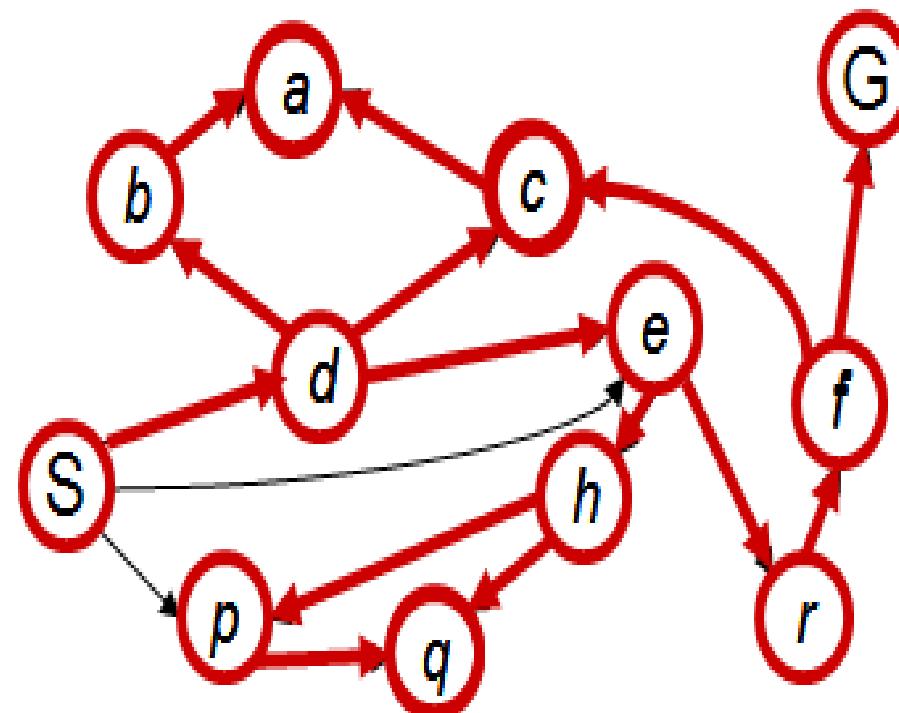
- **Frontier: LIFO Queue(Equivalent to stack)**
- **Solution path found is S A G, cost 18**
- **Number of nodes expanded (including goal node) = 5**

DEPTH FIRST SEARCH

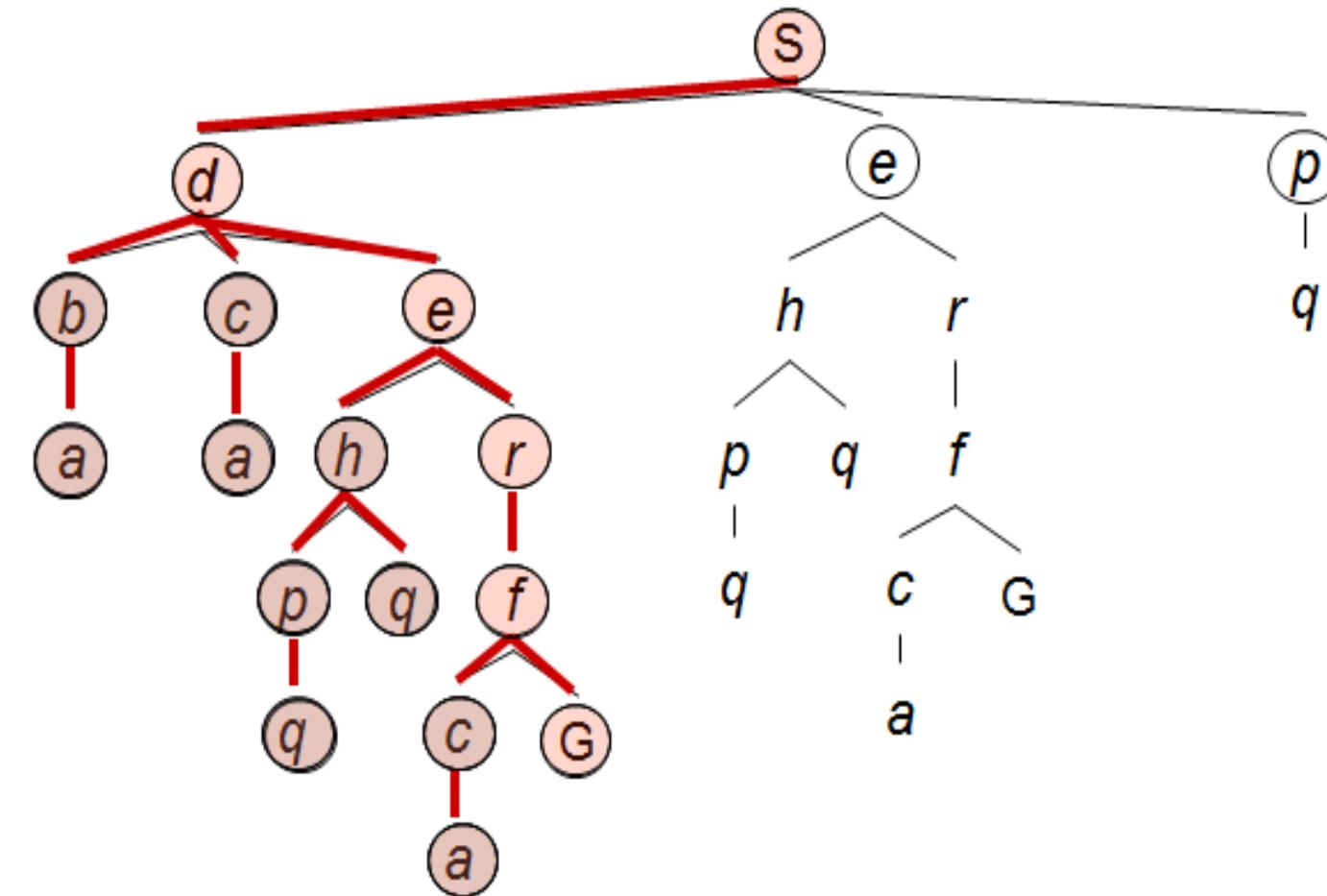
Strategy: expand a deepest node first

Implementation:

Fringe is a LIFO stack



DFS



	Expanded node	Nodes list/ open list
		{ s⁰ }
1	s⁰	{ d¹ e¹ p¹ }
2	d¹	{ b² c²e² e¹ p¹ }
3	b²	{ a³ c²e² e¹ p¹ }
4	a³	{ c²e² e¹ p¹ }
5	c²	{ a³ e² e¹ p¹ }
6	e²	{ h³ r³ e¹ p¹ }
7	h³	{ p³ q³ r³ e¹ p¹ }
8	p³	{ q⁴ q³ r³ e¹ p¹ }
9	q⁴	{ q³ r³ e¹ p¹ }
10	r³	{ f⁴ e¹ p¹ }
11	f⁴	{ c⁵g⁵ e¹ p¹ }
12	g⁵	{ e¹ p¹ }

DFS PROPERTIES

What nodes DFS expand?

Some left prefix of the tree.

Could process the whole tree!

If m is finite, takes time $O(b^m)$

How much space does the fringe take?

how many nodes can be in the queue (worst-case)?

At depth $l < d$ (max) we have $b-1$ nodes

So $O(b^*d)$ linear space. terrible if m is much larger than d but if solutions are dense, it may be much faster than breadth-first

Time?

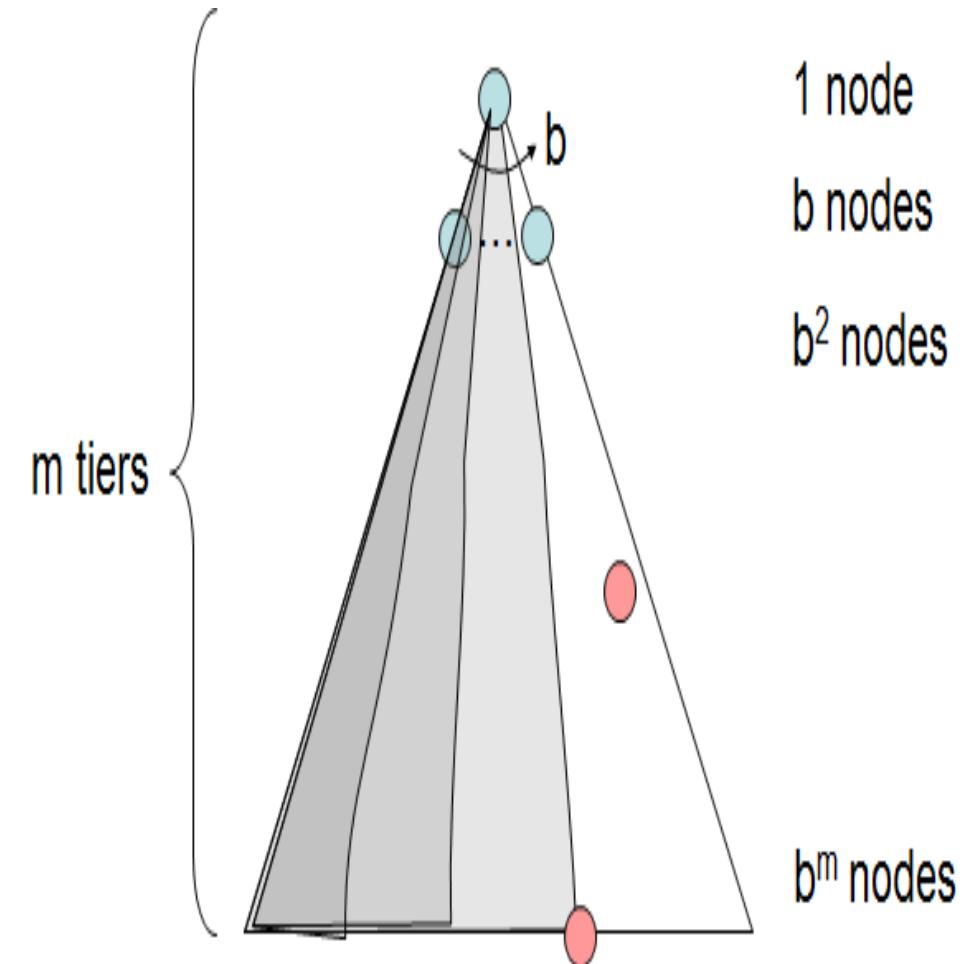
$O(b^m)$

Is it complete?

No. It fails in infinite-depth spaces it also fails in finite spaces with loops. but if we modify the search to avoid repeated states \Rightarrow complete in finite spaces (even with loops)

Is it optimal?

No, it finds the “leftmost” solution, regardless of depth or cost



Criterion	Breadth-First	Depth-First
Time	b^d	b^m
Space	b^d	bm
Optimal?	Yes if we guarantee that deeper solutions are less optimal, e.g. step-cost=1	No It may find a non-optimal goal first
Complete?	Yes it always reaches goal (if b is finite)	No Fails in infinite-depth spaces
Advantages	Guaranteed to find a solution (if one exists) - complete; Depending on the problem, can be guaranteed to find an <i>optimal</i> solution.	Simple to implement; Needs relatively small memory for storing the state-space.
Disadvantages	Needs a lot of memory for storing the state space if the search space has a high branching factor. More complex to implement;	Sometimes fail to find a solution (may be get stuck in an infinite long branch) - not complete ; Not guaranteed to find an <i>optimal</i> solution (may not find the shortest path solution); Can take a lot longer to find a solution.

DEPTH LIMITED SEARCH

- Depth-limited search avoids the pitfalls of depth-first search which is infinite path by **imposing a cutoff on the maximum depth of a path**.
- Depth-first search with depth limit l . Algorithm treats the node at the depth limit l as it has no successor nodes further.
- In this algorithm, Depth-limited search can be terminated with two Conditions of failure:
 - **Standard failure value**: It indicates that problem does not have any solution.
 - **Cutoff failure value**: It defines no solution for the problem within a given depth limit.

Advantages:

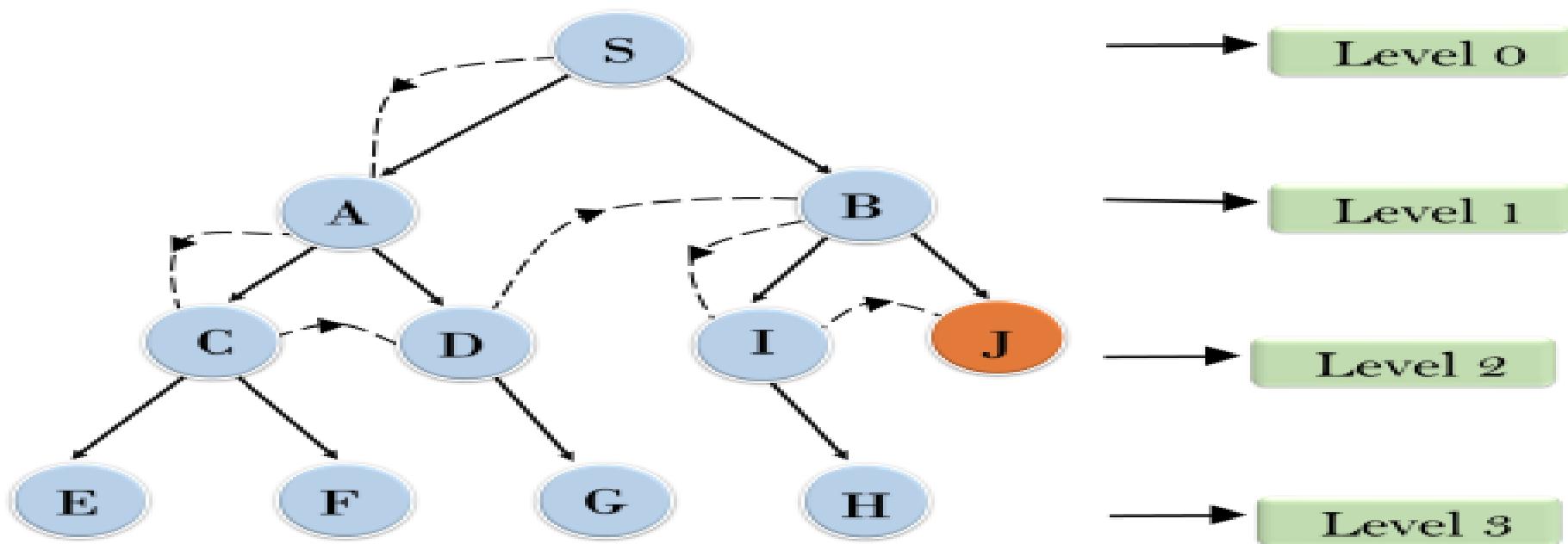
- Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

DEPTH LIMITED SEARCH

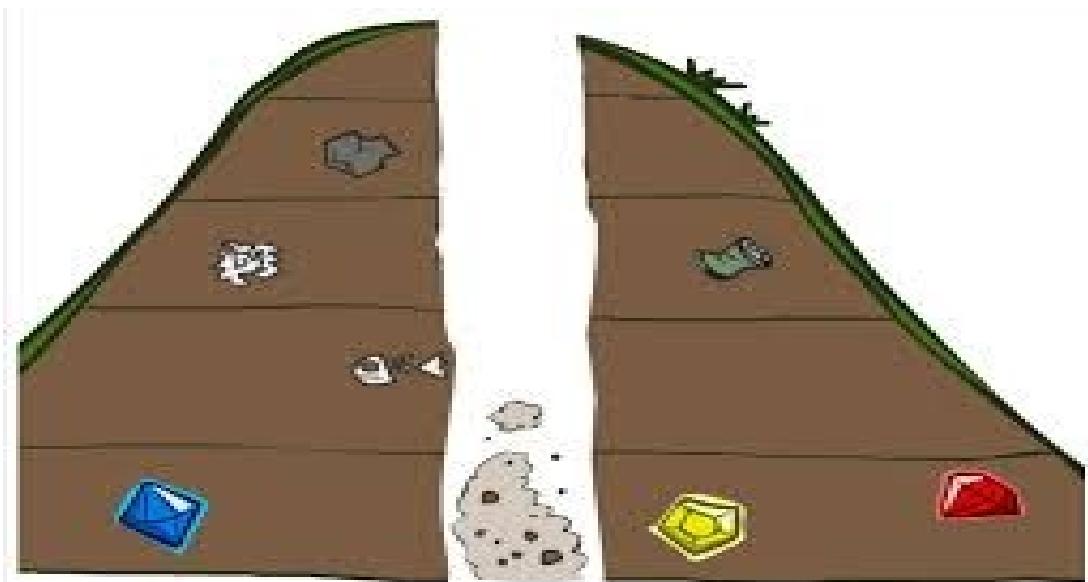
Depth Limited Search



PROPERTIES OF DLS

- Complete?
- Yes (unless the node is within the depth l)
- Time?
- $O(b^l)$ Exponential
- Space?
- $O(bl)$ Keeps all nodes in memory
- Optimal?
- No (depending upon search algo and heuristic property)

ITERATIVE DEEPENING SEARCH



ITERATIVE DEEPENING SEARCH

- The hard part about depth-limited search is picking a good limit, which is known as diameter of the state space. for most problems, we will not know a good depth limit until we have solved the problem.
- Iterative deepening search is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits: first depth 0, then depth 1, then depth 2, and so on.
- The iterative deepening algorithm is a combination of **DFS** and **BFS** algorithms. This search algorithm **finds out the best depth limit** and does it by iteratively increasing the depth limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- To avoid the infinite depth problem of DFS, we can decide to only search until depth L, i.e. we don't expand beyond depth L.

ITERATIVE DEEPENING SEARCH ALGORITHM

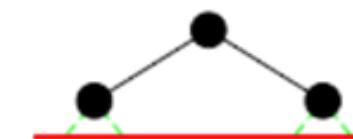
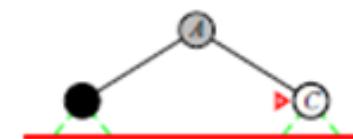
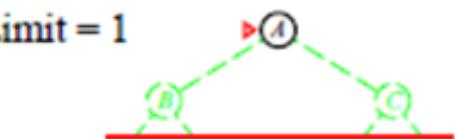
- Explore the nodes in DFS order.
- Set a LIMIT variable with a limit value.
- Loop each node up to the limit value and further increase the limit value accordingly.
- Terminate the search when the goal state is found.

ITERATIVE DEEPENING SEARCH

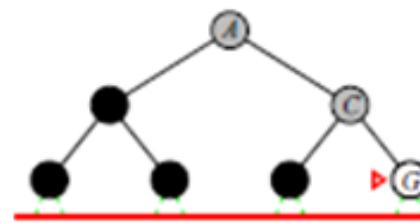
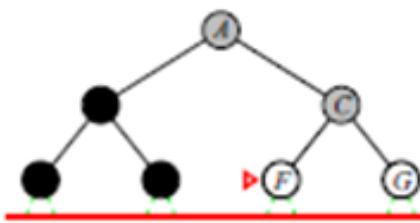
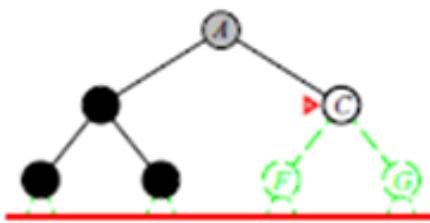
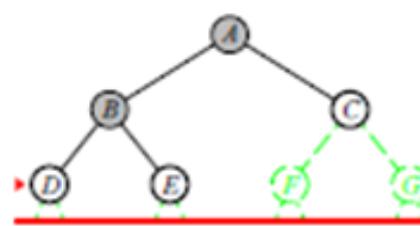
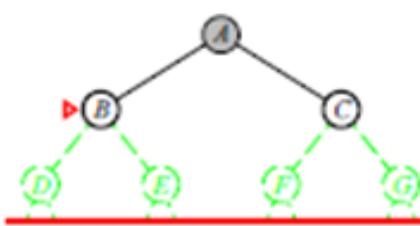
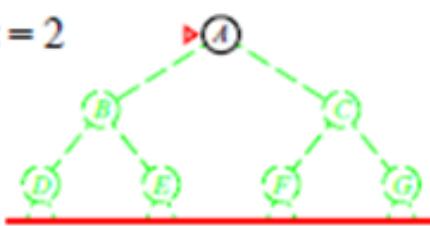
Limit = 0



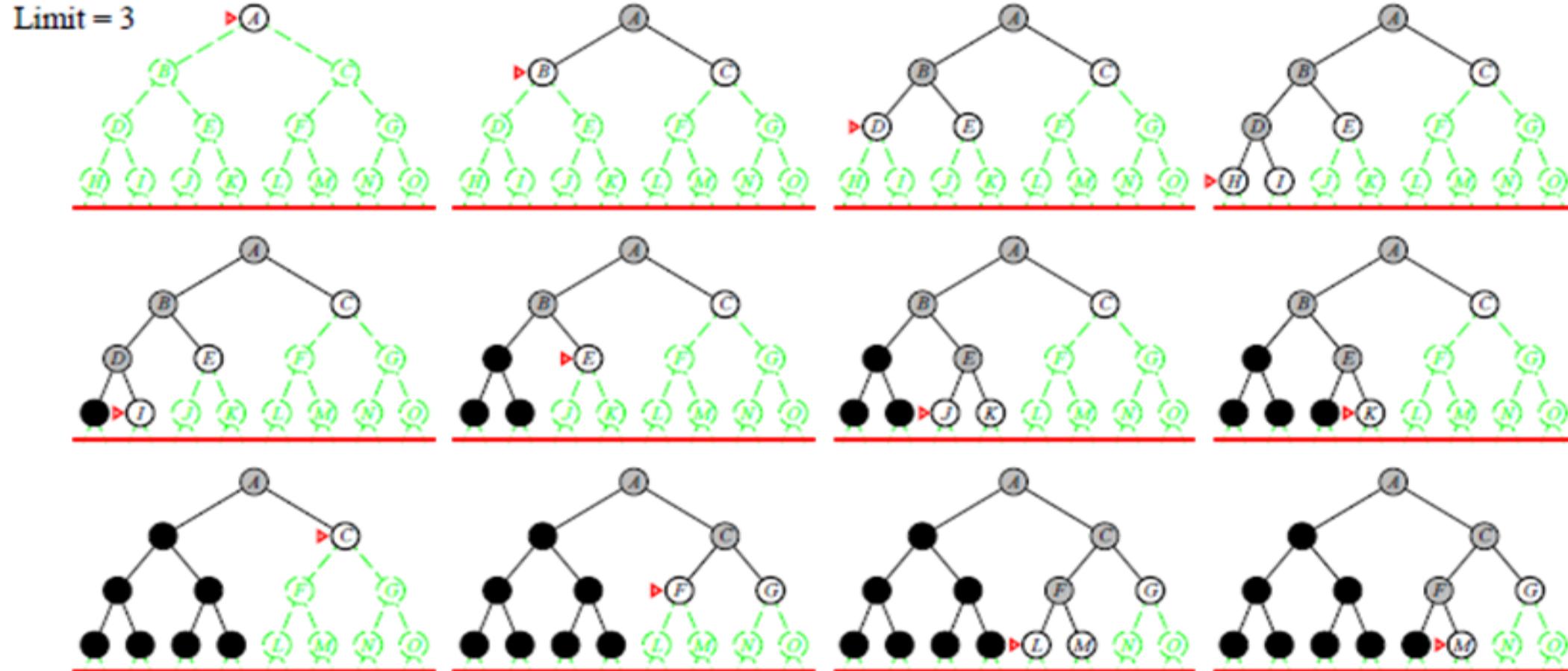
Limit = 1



Limit = 2

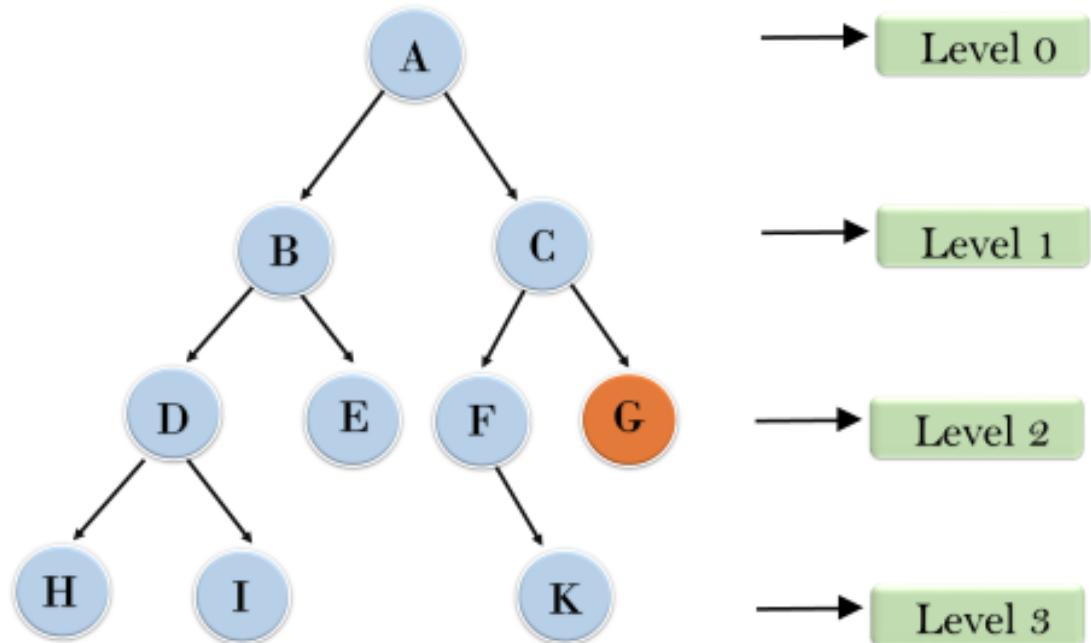


ITERATIVE DEEPENING SEARCH



ITERATIVE DEEPENING SEARCH

Iterative deepening depth first search



1'st Iteration----> A

2'nd Iteration----> A, B, C

3'rd Iteration----->A, B, D, E, C, F, G

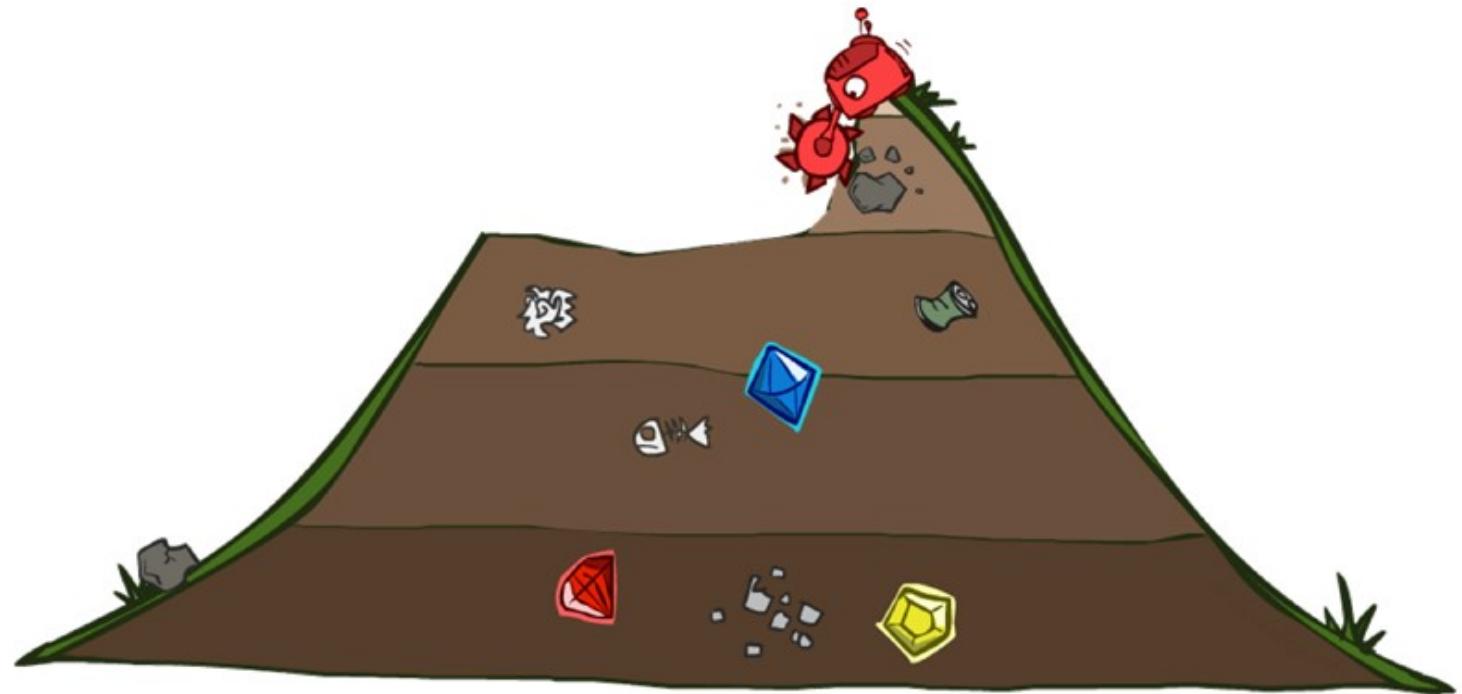
4'th Iteration----->A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.

PROPERTIES OF ITERATIVE DEEPENING SEARCH

- Complete?? **Yes**
- Time?? $(d + 1) b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$
- Space?? **$O(bd)$**
- Optimal?? **Yes**, if step cost = 1 it can be modified to explore a uniform-cost tree. Otherwise, not optimal but guarantees finding solution of shortest length (like BFS).
- **Disadvantages of Iterative deepening search**
 - The drawback of iterative deepening search is that it seems wasteful because it generates states multiple times.
 - **Note:** Generally, iterative deepening search is required when the search space is large, and the depth of the solution is unknown.

UNIFORM COST SEARCH



UNIFORM COST SEARCH

- The primary goal of the uniform-cost search is to **find a path** to the goal node which has the **lowest cumulative cost** ie sort by the **cost-so-far**.
- A uniform-cost search algorithm is implemented by the **priority queue**. It gives **maximum priority to the lowest cumulative cost** and Enqueue nodes by **path cost**.
- Uniform cost search modifies the breadth-first strategy by always expanding the lowest-cost node on the fringe. Uniform cost search is **equivalent to BFS algorithm** if the **path cost of all edges is the same**.
- **Algorithm outline:** Let $g(n) = \text{cost of the path from the start node to the current node } n$. Sort nodes by increasing value of g
 - Always select from the OPEN the node with the least $g(.)$ value for expansion, and put all newly generated nodes into OPEN
 - Nodes in OPEN are sorted by their $g(.)$ values (in ascending order)
 - Terminate if a node selected for expansion is a goal
- Called “*Dijkstra's Algorithm*” in the algorithm's literature and similar to “*Branch and Bound Algorithm*” in operations research literature

UNIFORM COST SEARCH

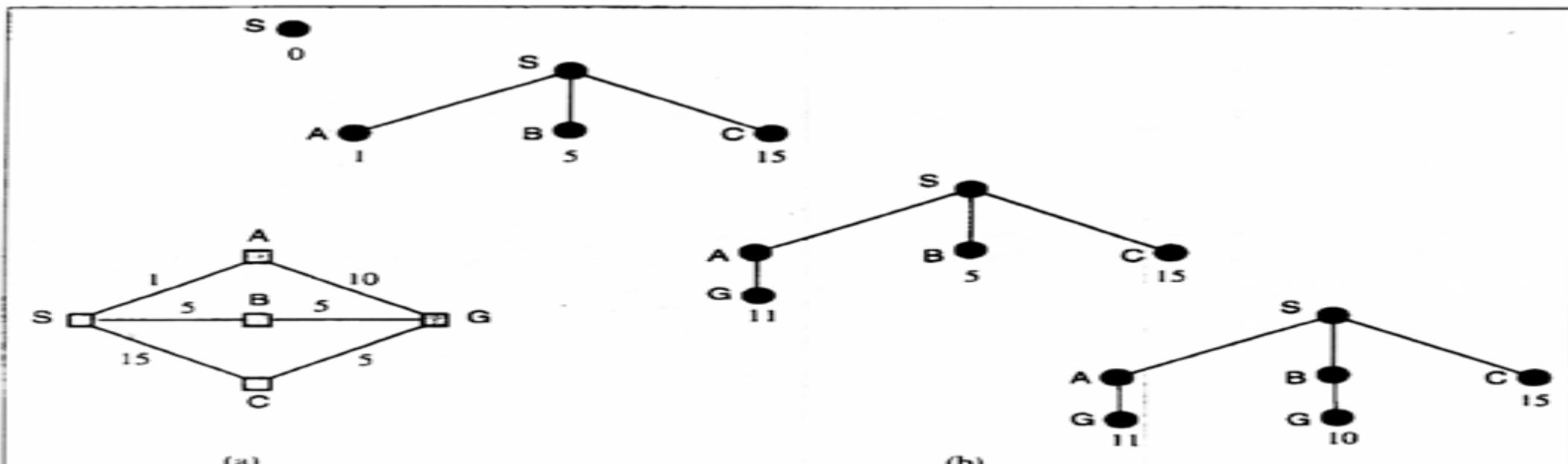
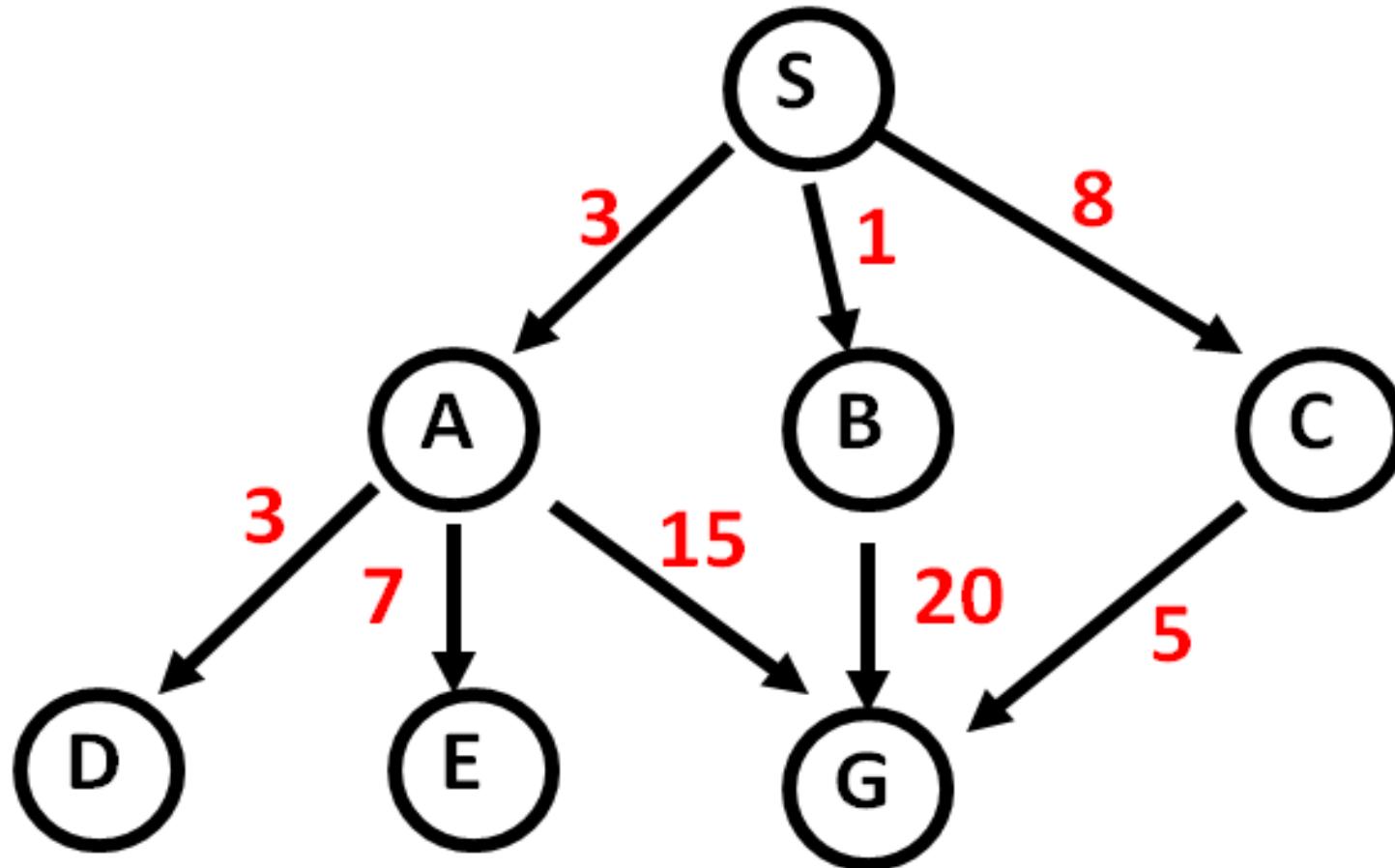


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

UNIFORM COST SEARCH



UNIFORM COST SEARCH

	Expanded node	Nodes list/open list
		{ S ⁰ }
1	S ⁰	{ B ¹ A ³ C ⁸ }
2	B ¹	{ A ³ C ⁸ G ²¹ }
3	A ³	{ D ⁶ C ⁸ E ¹⁰ G ¹⁸ G ²¹ }
4	C ⁸	{ C ⁸ E ¹⁰ G ¹⁸ G ²¹ }
5	C ⁸	{ E ¹⁰ G ¹³ G ¹⁸ G ²¹ }
6	E ¹⁰	{ G ¹³ G ¹⁸ G ²¹ }
7	G ¹⁸	{ G ²¹ G ¹³ }

- Solution path found is S B G, cost 13
- Number of nodes expanded (including goal node) = 7

UNIFORM COST SEARCH

Strategy: expand a cheapest node first:

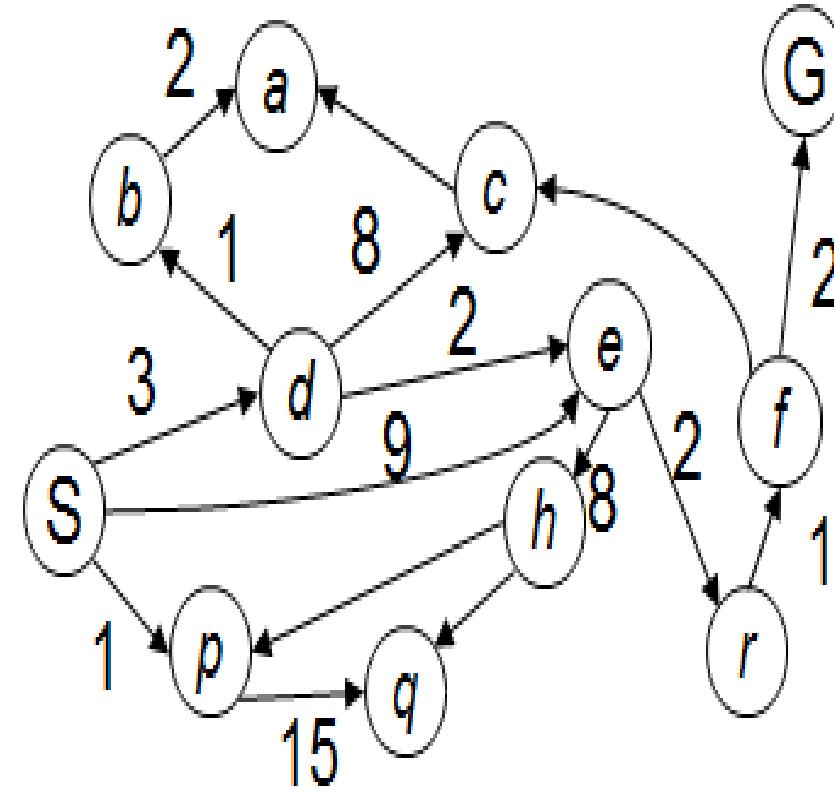
Fringe is a priority queue
(priority: cumulative cost)

The good:

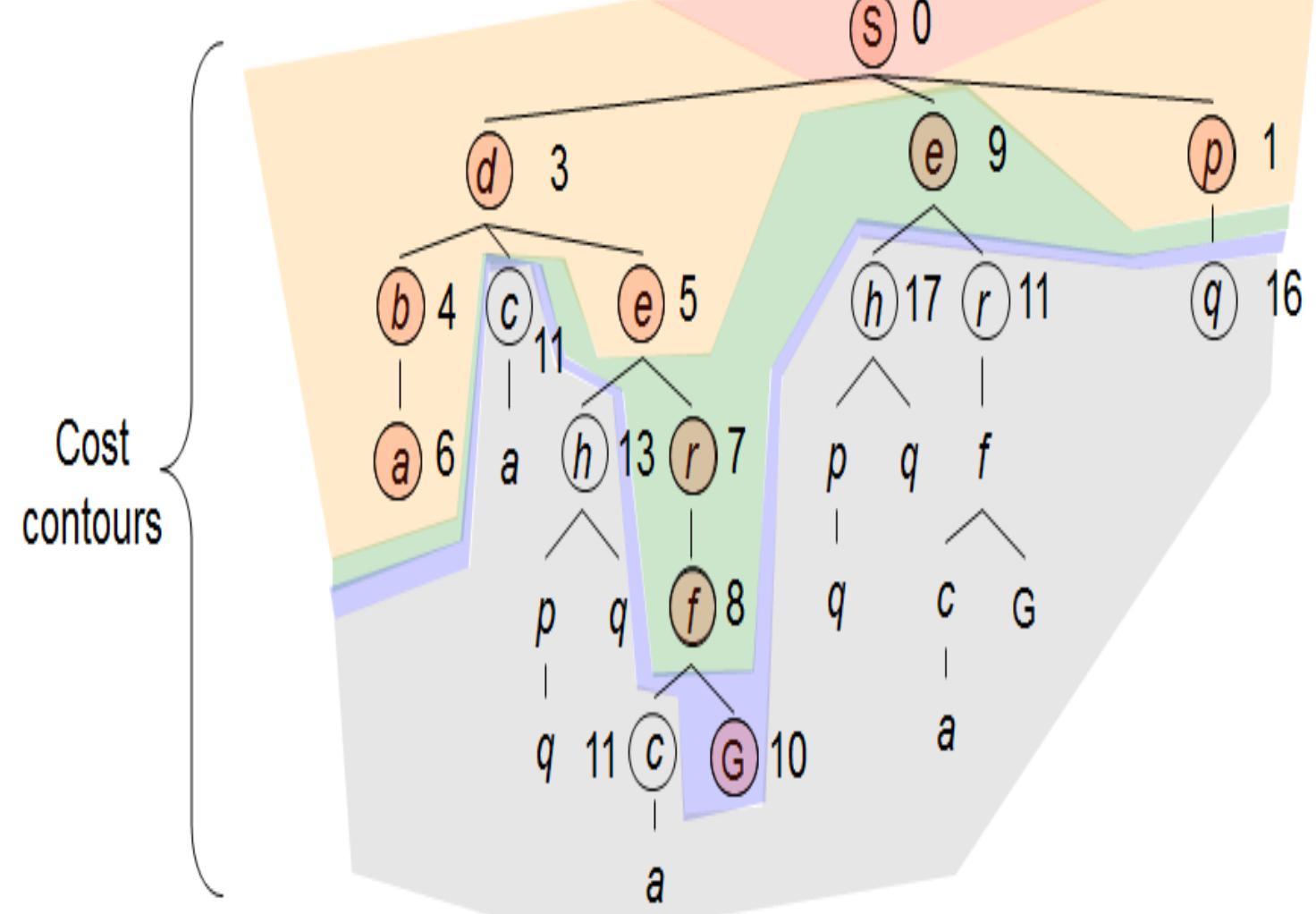
UCS is complete and optimal!

The bad:

Explores options in every “direction”. No information about goal location



UNIFORM COST SEARCH



	Expanded node	Nodes list/ open list
		{ S ⁰ }
1	s ⁰	{ p ¹ d ³ e ⁹ }
2	p ¹	{ d ³ e ⁹ q ¹⁶ }
3	d ³	{ b ⁴ e ⁵ e ⁹ c ¹¹ q ¹⁶ }
4	b ⁴	{ e ⁵ a ⁶ e ⁹ c ¹¹ q ¹⁶ }
5	e ⁵	{ a ⁶ r ⁷ e ⁹ c ¹¹ h ¹³ q ¹⁶ }
6	a ⁶	{ r ⁷ e ⁹ c ¹¹ h ¹³ q ¹⁶ }
7	r ⁷	{ f ⁸ e ⁹ c ¹¹ h ¹³ q ¹⁶ }
8	f ⁸	{ e ⁹ g ¹⁰ c ¹¹ h ¹³ q ¹⁶ }
9	g ¹⁰	{ c ¹¹ h ¹³ q ¹⁶ }

UCS PROPERTIES

What nodes does UCS expand?

Processes all nodes with cost less than cheapest solution!

If that solution costs C^* and arcs cost at least ε , then the “effective depth” is roughly C^*/ε

Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)

Optimal?

Yes. Optimality depends on the goal test being applied when a node is removed from the nodes list, not when its parent node is expanded and the node is first generated

Time?

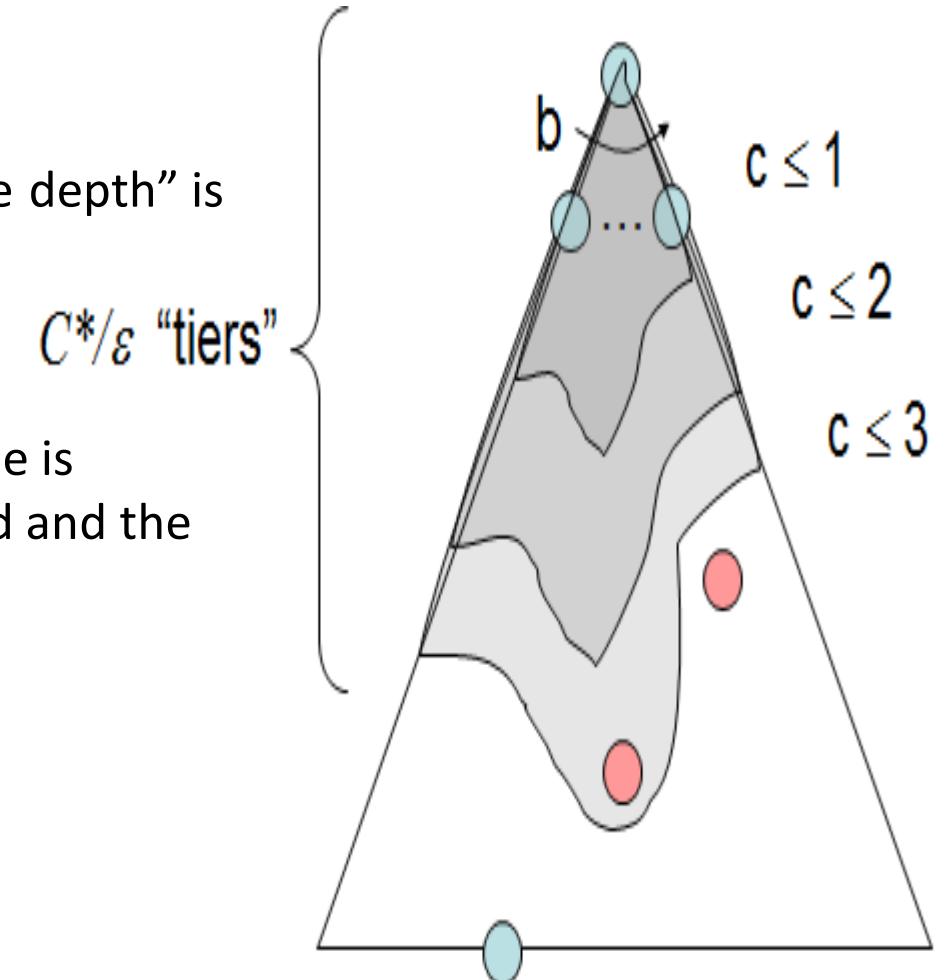
Exponential time and space complexity, $O(b^{C^*/\varepsilon})$

How much space does the fringe take?

Has roughly the last tier, so $O(b^{C^*/\varepsilon})$

Is it complete?

Assuming best solution has a finite cost and minimum arc cost is positive, yes!



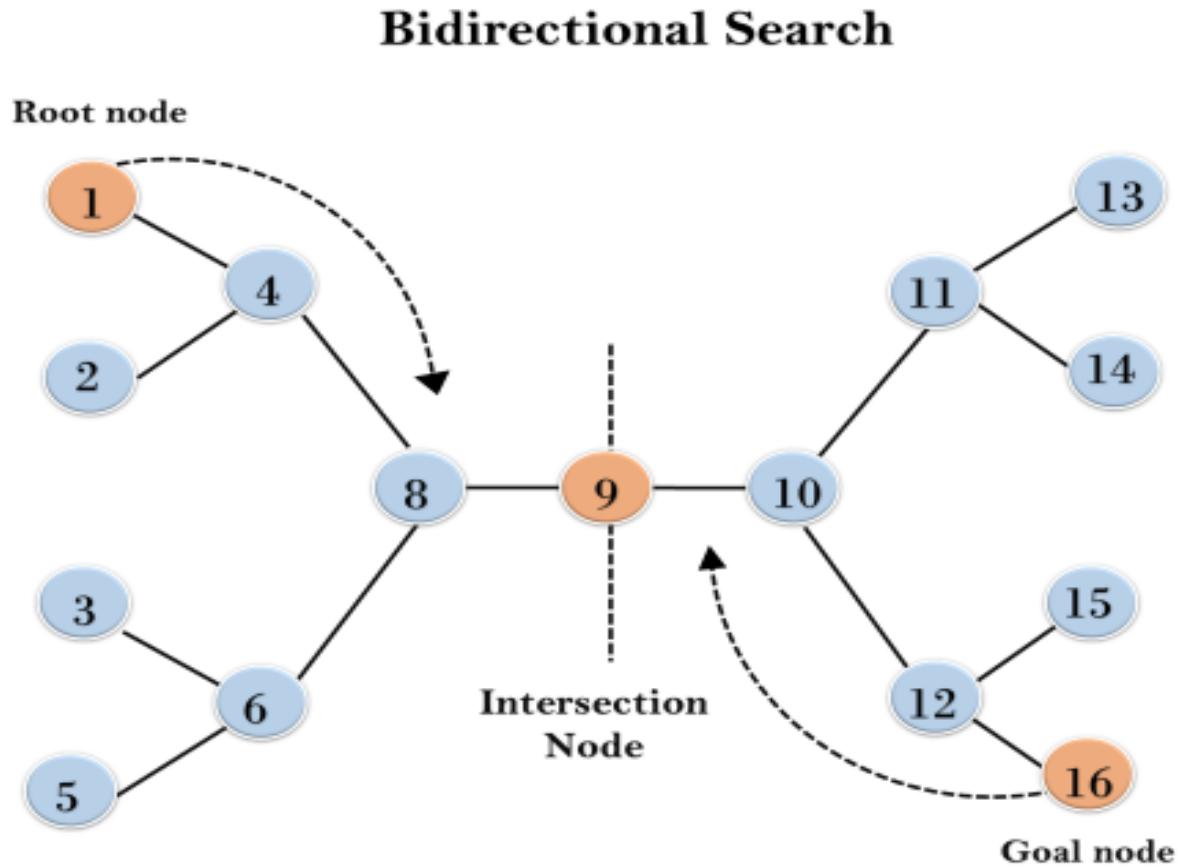
BIDIRECTIONAL SEARCH

- Bidirectional search algorithm runs **two simultaneous searches, one from initial state** called as **forward-search** and **other from goal node** called as **backward-search**, to find the goal node. The search stops when these two graphs intersect each other.
- It is implemented by replacing goal test with a test to see whether **frontiers of 2 searches intersect**.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, Iterative deepening search etc.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex.
- **Idea**
 - simultaneously search forward from S and backwards from G
 - stop when both “meet in the middle”
 - need to keep track of the intersection of 2 open sets of nodes. need a way to specify the predecessors of G this can be difficult

When to use bidirectional approach?

- Both initial and goal states are unique and completely defined.
- The branching factor is same in both directions.

BIDIRECTIONAL SEARCH



PROPERTIES OF BIDIRECTIONAL SEARCH

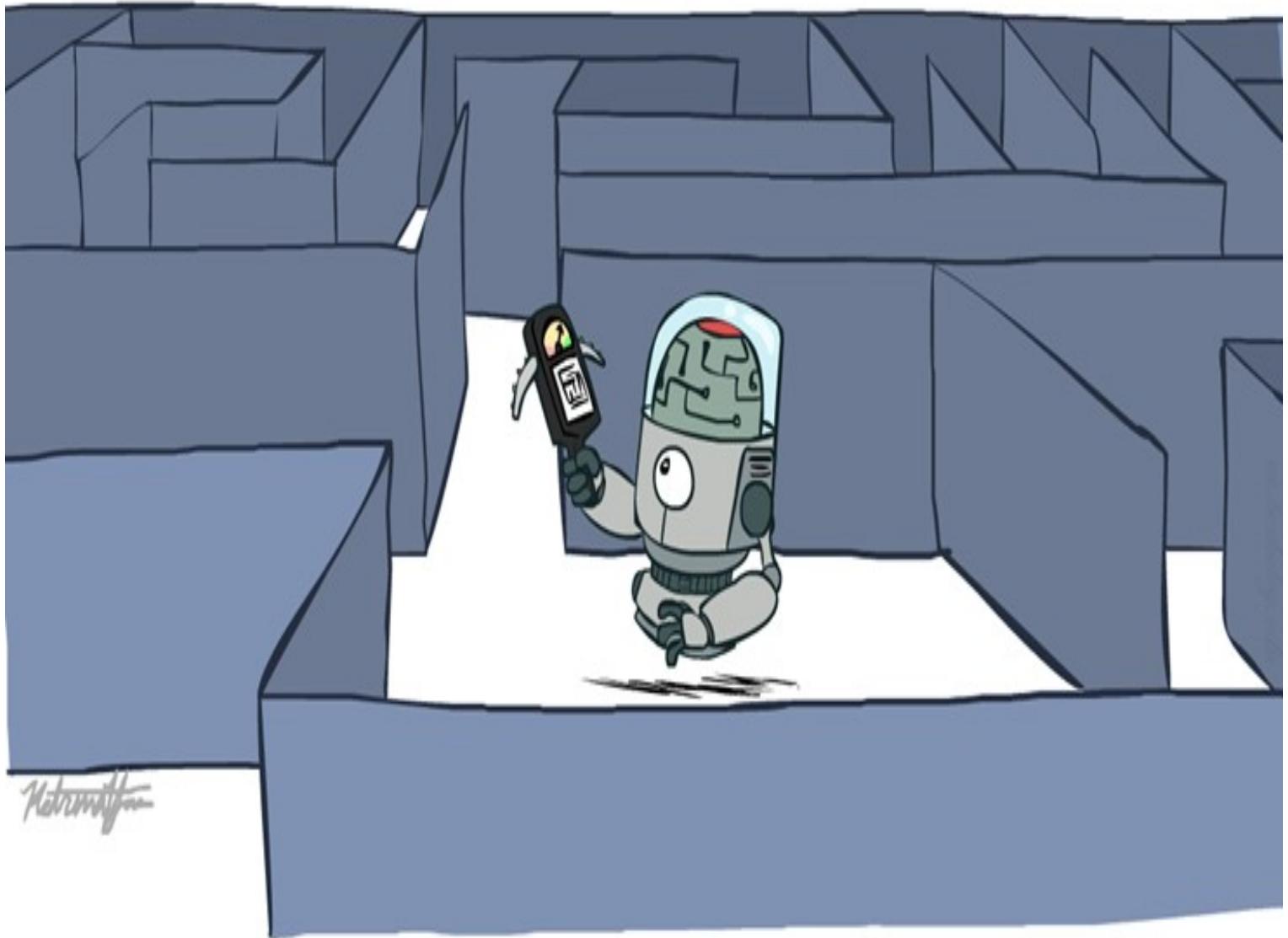
- Complete: Yes. Bidirectional search is complete.
- Optimal: Yes. It gives an optimal solution.
- Time and space complexity: Bidirectional search has $O(b^{d/2})$
- Disadvantage of Bidirectional Search
- **Though time complexity is reduced increased search complexity is a weakness owing to intersection check and how to search backward.**
- If the goal is an abstract description like for N-Queens problem its very difficult to use.

COMPARISON

Criterion	BFS	DFS	Limited Depth	Iterative deepening	Bidirectional	Uniform Cost Search
Time	B^d	B^m	B^l	B^d	$B^{d/2}$	$B^{c^*/\square}$
Space	B^d	B^{*m}	B^{*l}	B^d	$B^{d/2}$	$B^{c^*/\square}$
Optimality?	Yes	No	No	Yes	Yes	Yes
Completeness	Yes	No	Yes if $l \geq d$	Yes	Yes if $e \geq 0$	Yes if $\square \geq 0$

- m – maximum depth of the tree, l – depth limit of search (AdMax)
- \square -smallest step cost
- C^* -cost of optimal solution

INFORMED SEARCH STRATEGIES



INFORMED SEARCH

- Informed search algorithms **use domain knowledge**. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution **more efficiently than an uninformed search strategy**. Informed search is also called a **Heuristic search**.
- A **heuristic** is a way which **might not always be guaranteed for best solutions** but **guaranteed to find a good solution in reasonable time**.
- A node is selected for expansion in informed search algorithm based on an evaluation function that estimates cost to goal.
- Informed search can solve much complex problem which could not be solved in another way.
- It contains the problem description as well as extra information like how far is the goal node.
- It might not give the optimal solution always but it will definitely give a good solution in a reasonable time. It can solve complex problems more easily than uninformed.

Example: traveling salesman problem, Greedy Search, A* Search

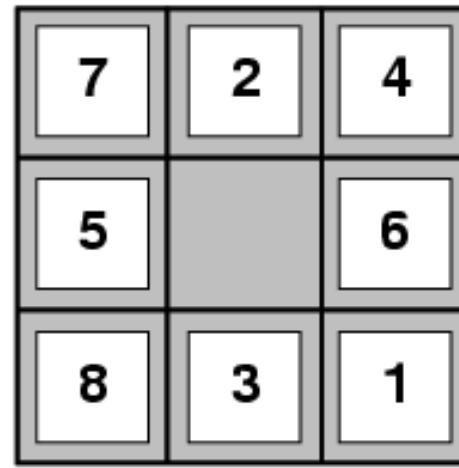
HEURISTIC FUNCTIONS

- A **heuristic function**, $h(n)$, is the **estimated cost of the cheapest path from the state at node n, to a goal state**.
- The value of the heuristic function is **always positive**.
- A heuristic is:
 - A function that *estimates* how close a state is to a goal
 - Designed for a particular search problem
- **Examples:** Manhattan distance, Euclidean distance for pathing
- Heuristic is a function which is used in Informed Search finds the most promising path.
- Heuristic functions are very much dependent on the domain used. $h(n)$ might be the estimated number of moves needed to complete a puzzle, or the estimated straight-line distance to some town in a route finder.
- Choosing an appropriate function greatly affects the effectiveness of the state-space search, since it tells us which parts of the state-space to search next.
- A heuristic evaluation function which accurately represents the actual cost of getting to a goal state, tells us very clearly which nodes in the state-space to expand next, and leads us quickly to the goal state.

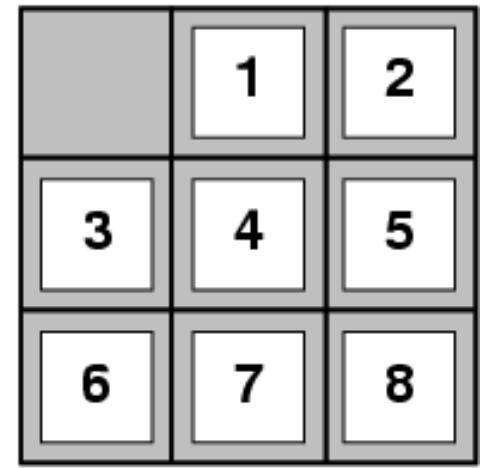
EXAMPLE HEURISTICS

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance(i.e., no. of squares from desired location of each tile)
- $h_1(S) = ?$
- $h_2(S) = ?$



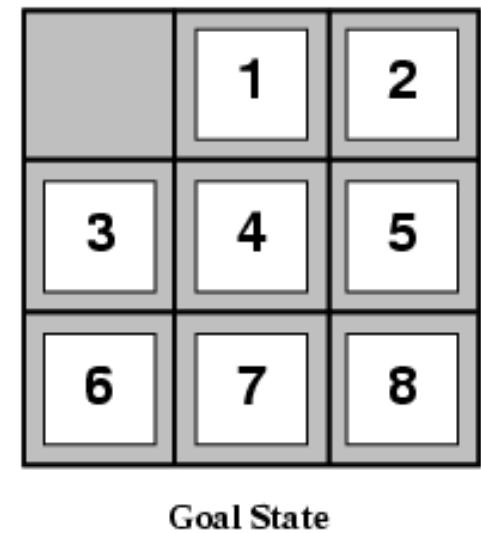
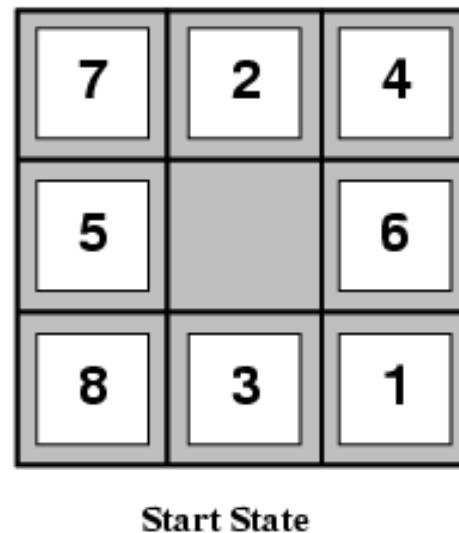
Start State



Goal State

EXAMPLE HEURISTICS

- E.g., for the 8-puzzle:
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance(i.e., no. of squares from desired location of each tile)
- $h_1(S) = 8$
- $h_2(S) = 3+1+2+2+2+3+3+2 = 18$



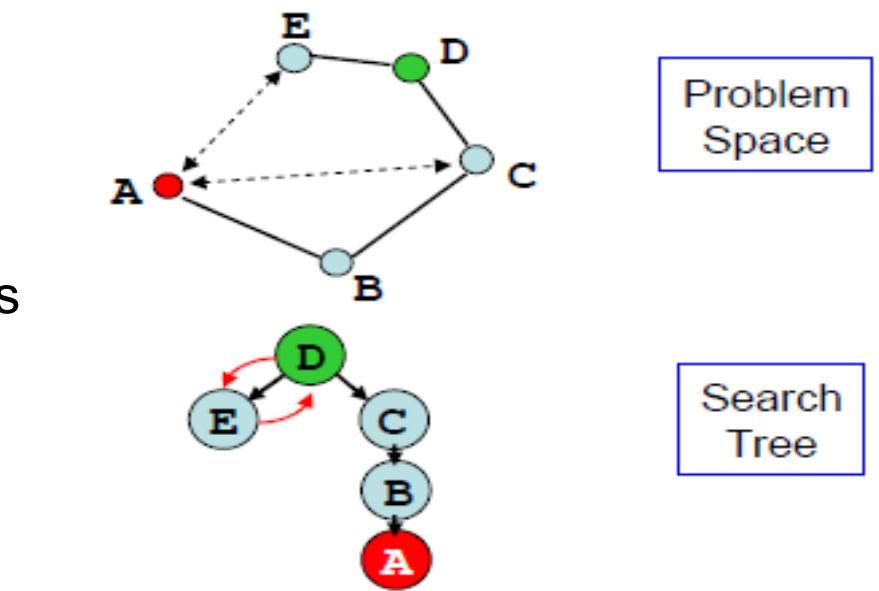
EXAMPLE HEURISTICS

- For Graph Search problem
- **Straight-line distance** : The distance between two locations on a map can be known without knowing how they are linked by roads (i.e. the absolute path to the goal).

$$d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}.$$

- This is known as Euclidian distance formula
- In three-dimensional Euclidean space, the distance is

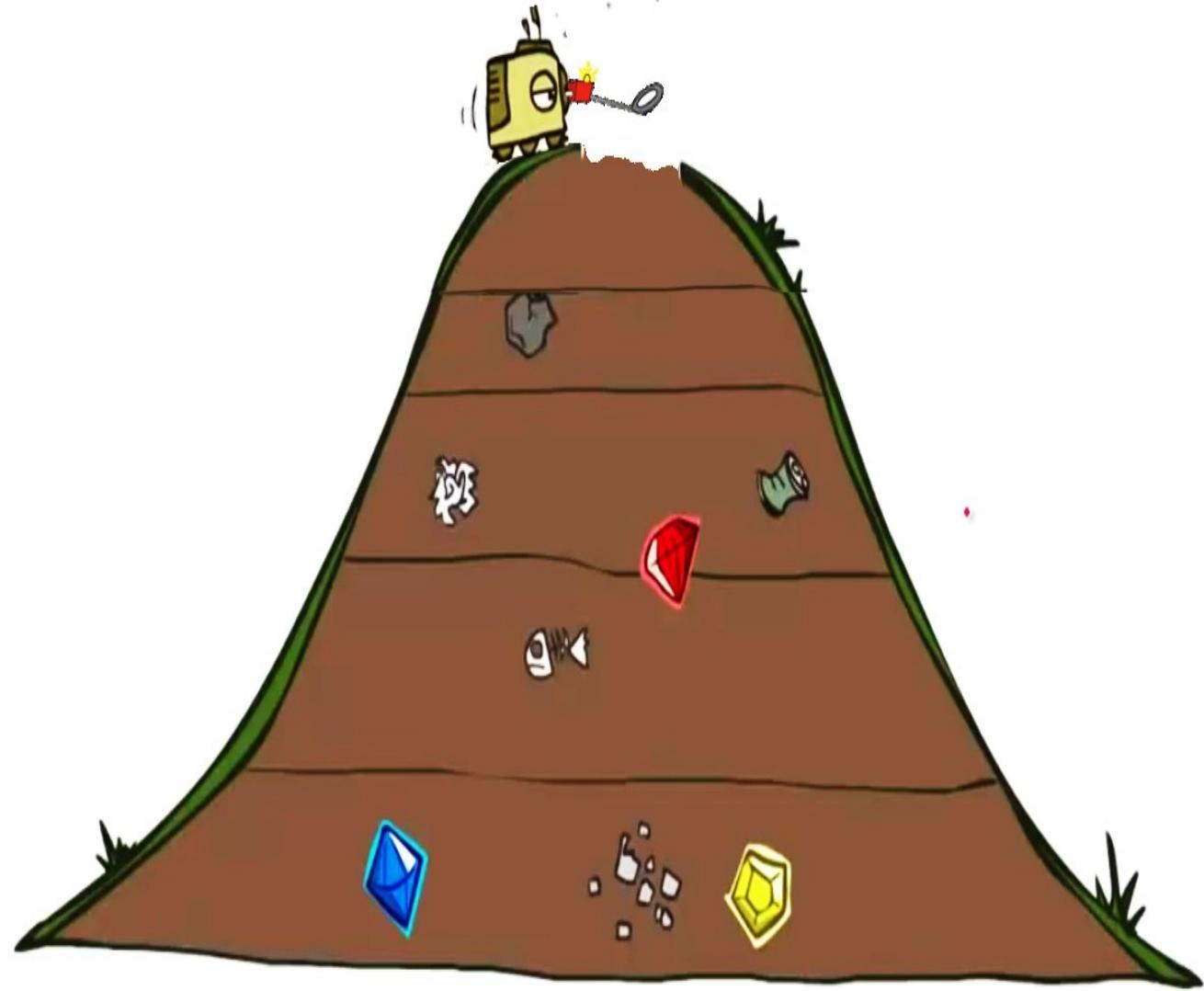
$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2}.$$



EFFICIENT HEURISTIC ACCURACY ON PERFORMANCE

- Quality of a heuristic is determined by the effective branching factor b^* .
- Total no: of nodes generated by A* is N at a depth d with a branching factor b^* then,
- $N+1=1+b^*+(b^*)^2+\dots+(b^*)^d$
- A well defined heuristic should have value of b^* close to 1.
- A dominant heuristic will be a better choice.
- A problem with fewer restrictions on the graph is called relaxed problem and cost of an optimal solution to a relaxed problem is admissible heuristic for original problem.

BEST
FIRST
SEARCH



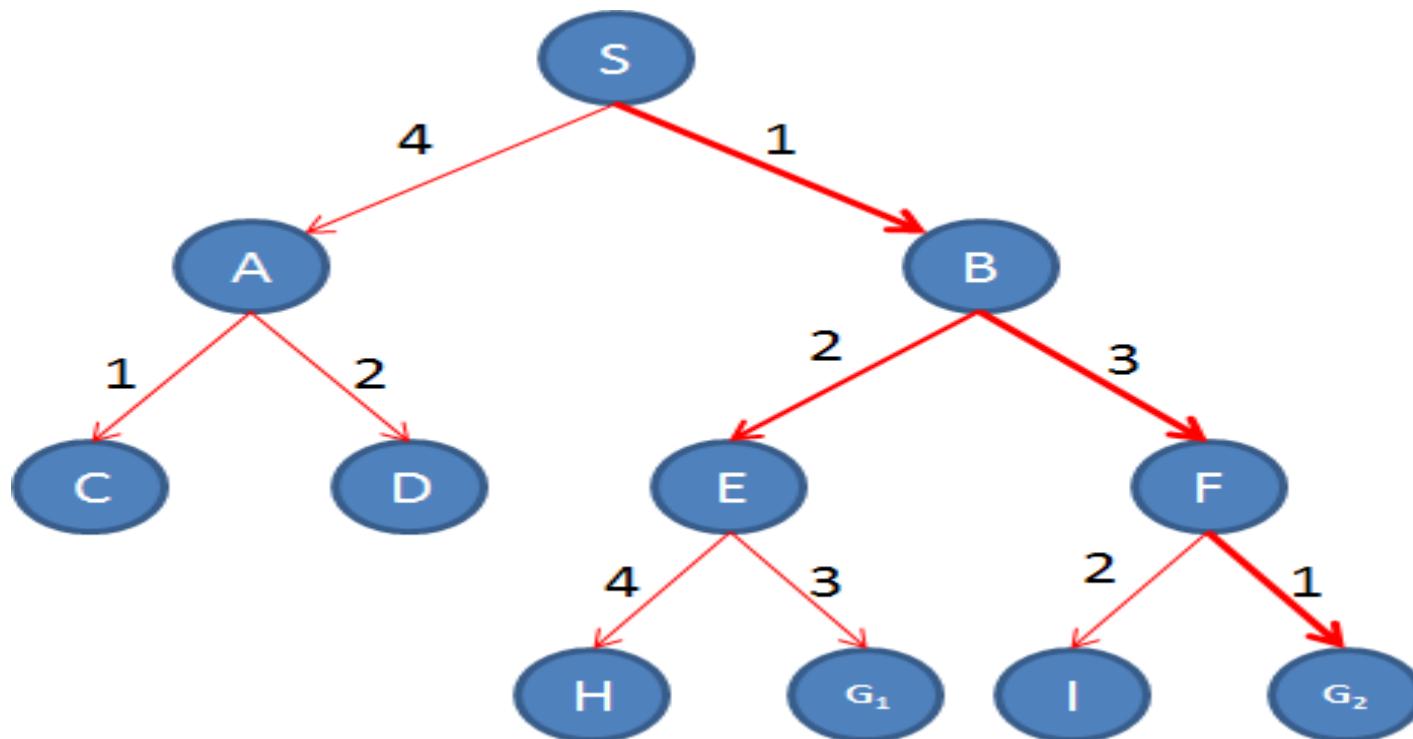
BEST FIRST SEARCH

- A search strategy is defined by picking the **order of node expansion** by using an **evaluation function $f(n)$** . It is used to assign score for each node.
- **Best-first search** is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function, $f(n)$**
- The evaluation function **$f(n)$** is a cost estimate that provides an **estimate for the total cost** also known as **estimate of "desirability"**. Expand the **node n with smallest $f(n)$** ie most desirable unexpanded node
- Implementation is like uniform cost search where f replaces g in priority queue.
- Choice of f determines search strategy, and most searches use heuristic function as f .

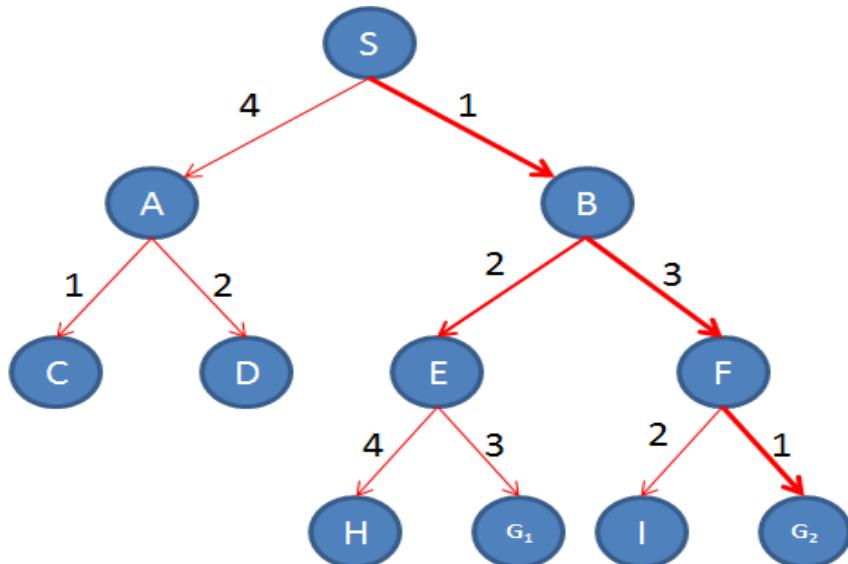
Implementation:

- Order the nodes in fringe **increasing order of cost**.
- The algorithm maintains two lists, one containing a list of candidates yet to explore (**OPEN**), and one containing a list of already visited nodes (**CLOSED**). States in OPEN are ordered according to some heuristic estimate of their “closeness” to a goal. This ordered OPEN list is referred to as **priority queue**.
- The algorithm always chooses the best of all unvisited nodes that have been graphed
- The advantage of this strategy is that if the algorithm reaches a dead-end node, it will continue to try other nodes. `

BEST FIRST SEARCH EXAMPLE



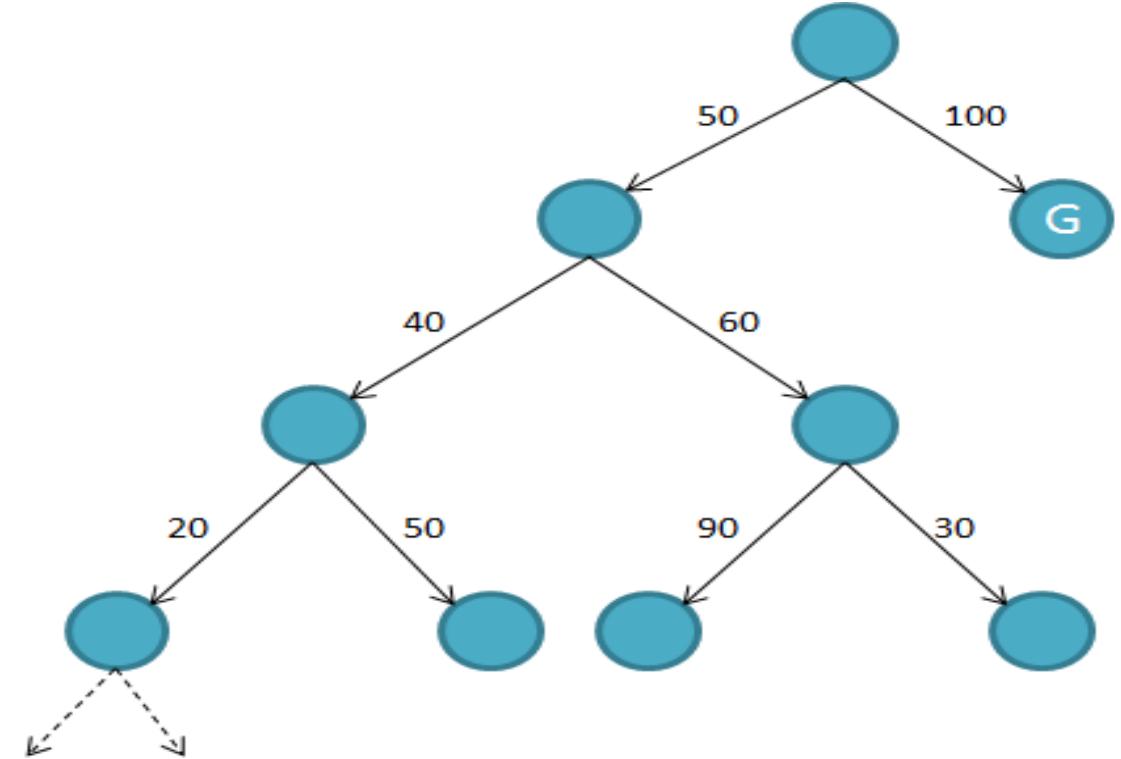
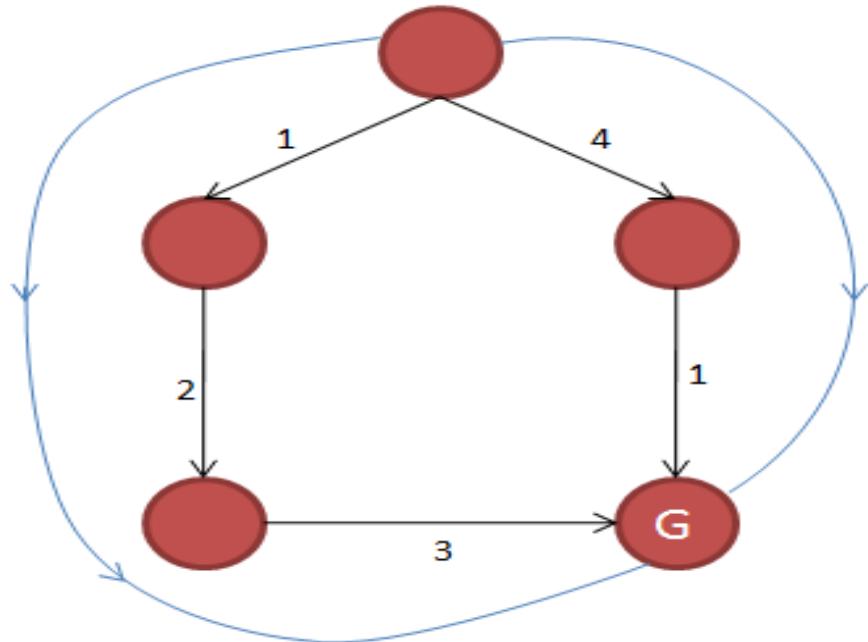
BEST FIRST SEARCH SOLUTION



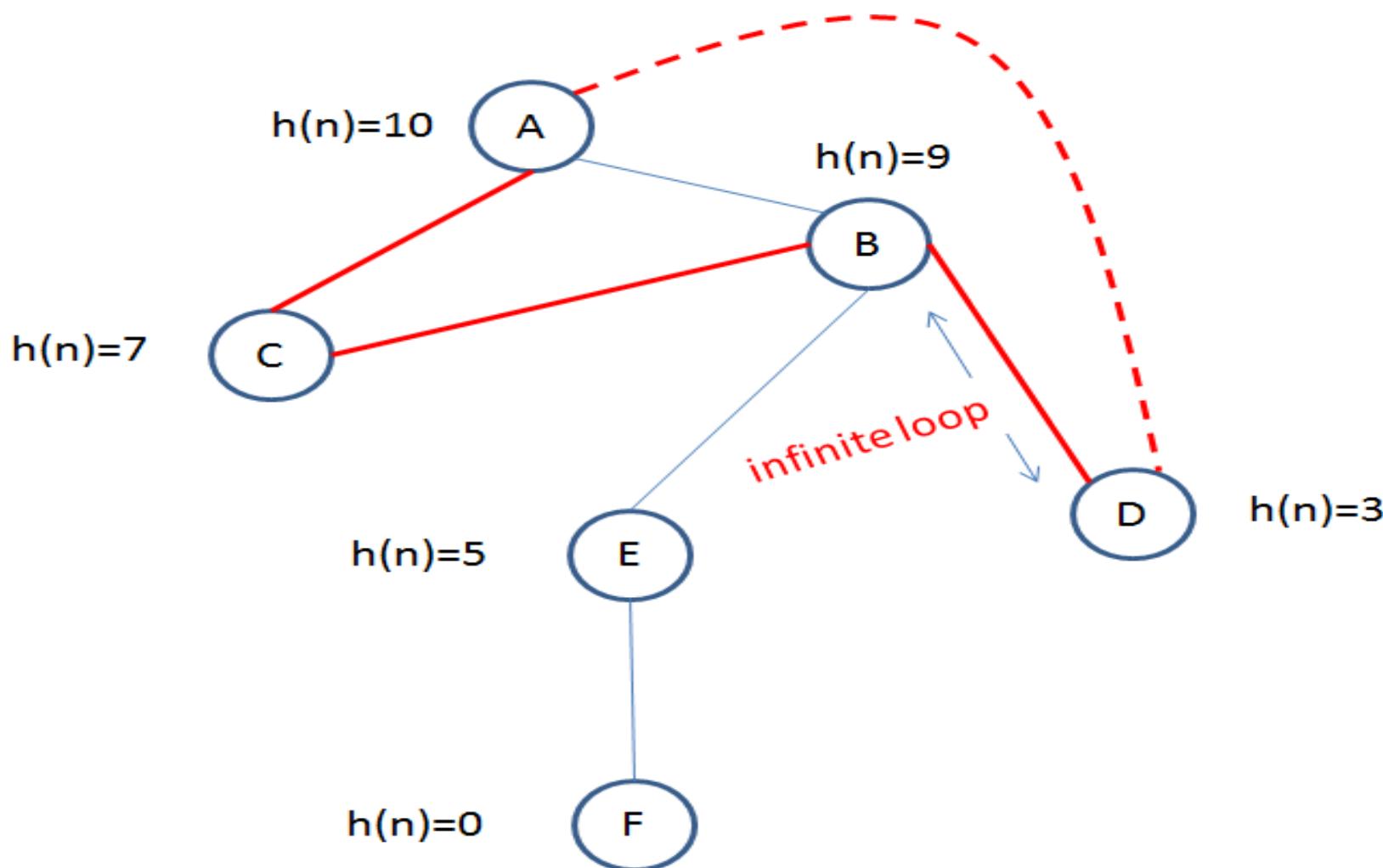
```
open=[S0]; closed=[ ]  
open=[B1, A4]; closed=[S0]  
open=[E3, A4, F4]; closed=[S0, B1]  
open=[A4, F4, G16, H7]; closed=[S0, B1, E3]  
open=[F4, C5, G16, D6, H7]; closed=[S0, B1, E3, A4]  
open=[C5, G25, G16, I6, D6, H7]; closed=[S0, B1, E3, A4, F4]  
open=[G25, G16, I6, D6, H7]; closed=[S0, B1, E3, A4, F4, C5]
```

Cost = 1+3+1=5

DOES BEST FIRST ALGORITHM ALWAYS GUARANTEE TO FIND SHORTEST PATH?



GBFS INFINITE LOOP



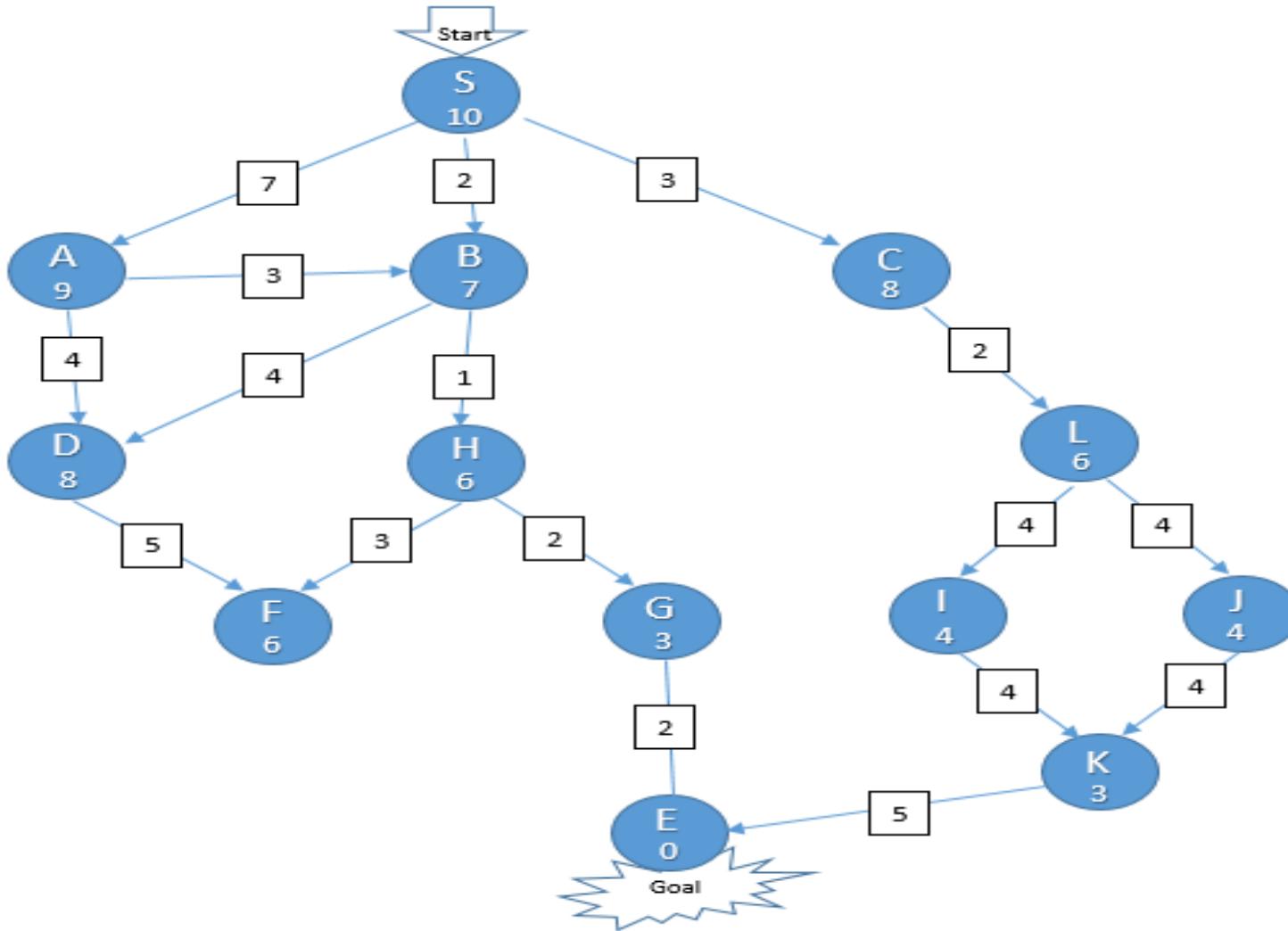
PROPERTIES OF BEST FIRST SEARCH

- It may **get stuck** in an infinite branch that doesn't contain the goal .
- It does **not guarantee** to find the shortest path solution .

Memory requirement :

- **In best case** : as depth first search.
- **In average case** : between depth and breadth.
- **In worst case** : as breadth first search

PROBLEM



GREEDY BFS



GREEDY BEST FIRST SEARCH

- Greedy best-first search expands the node that appears to be closest to goal.
- **Greedy best-first search is a best first search but uses heuristic estimate $h(n)$ rather than cost function. It sorts by the **cost of getting to the goal from that state****
- **Evaluation function $f(n) = h(n)$ (heuristic) = estimate of cost from n to goal**

E.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest

- Greedy best-first search algorithm always selects the path which appears best at that moment.

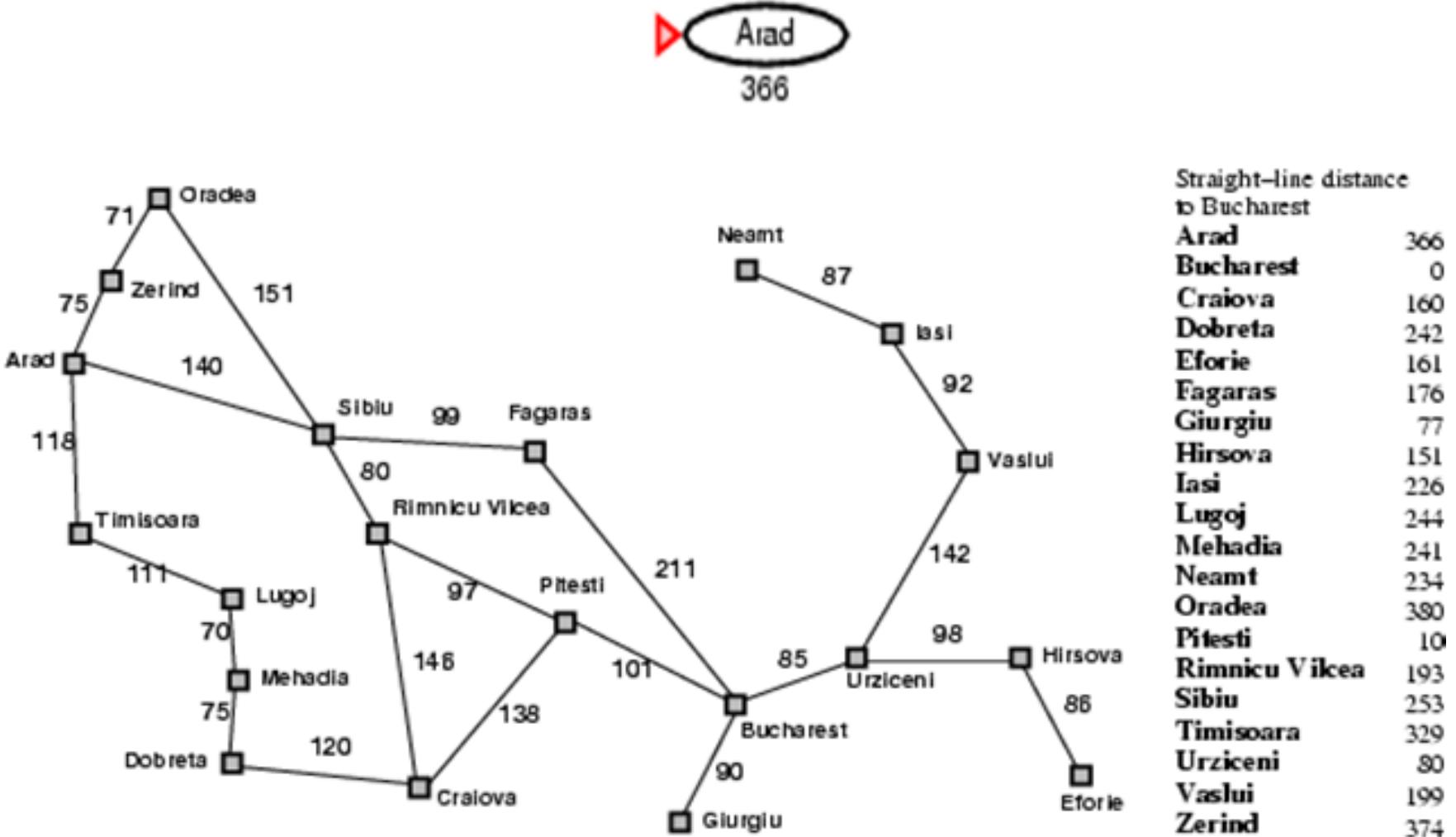
PROBLEM

- It doesn't take account of the cost so far like best first instead uses heuristic function, so it **isn't optimal**, and can **wander into dead-ends**, like depth-first search.
- In most domains, we also don't know the cost of getting to the goal from a state. So we have to guess, using a heuristic evaluation function.
 - If we knew how far we were from the goal state we wouldn't need to search for it!

GREEDY BEST-FIRST SEARCH EXAMPLE

Frontier
queue:

Arad 366



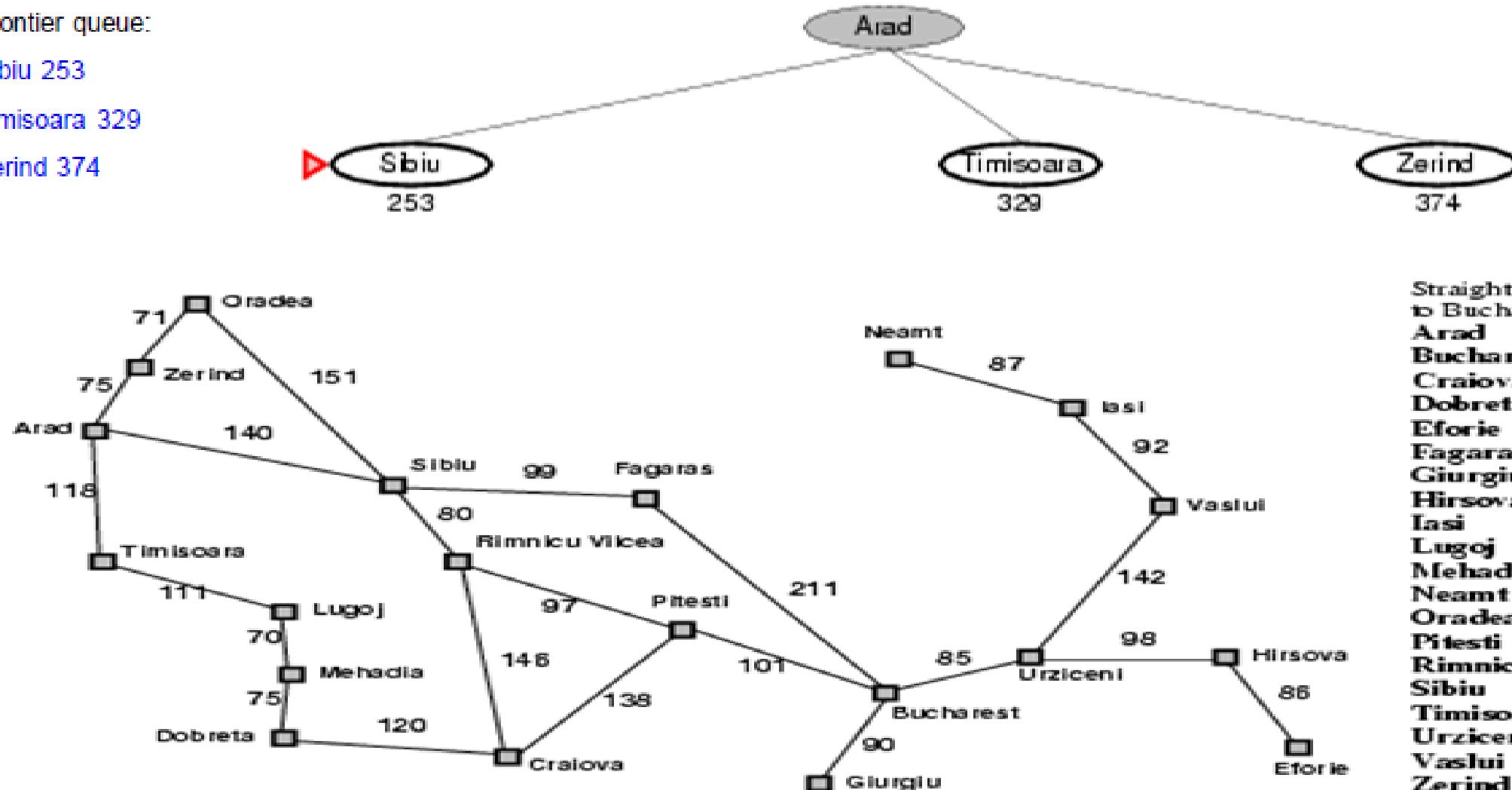
GREEDY BEST-FIRST SEARCH EXAMPLE

Frontier queue:

Sibiu 253

Timisoara 329

Zerind 374



GREEDY BEST-FIRST SEARCH EXAMPLE

Frontier queue:

Fagaras 176

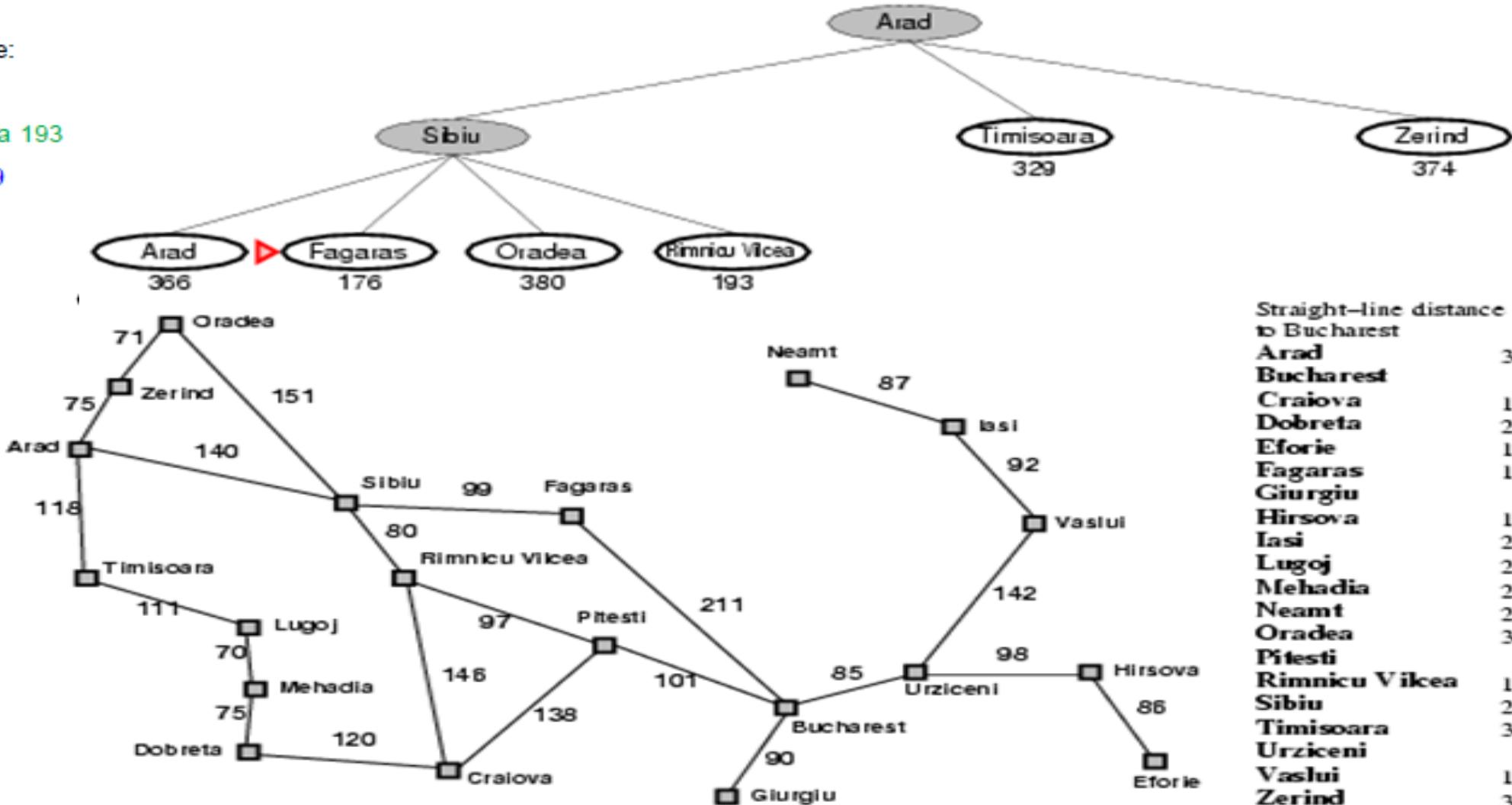
Rimnicu Vilcea 193

Timisoara 329

Arad 366

Zerind 374

Oradea 380



GREEDY BEST-FIRST SEARCH EXAMPLE

Frontier queue:

Bucharest 0

Rimnicu Vilcea 193

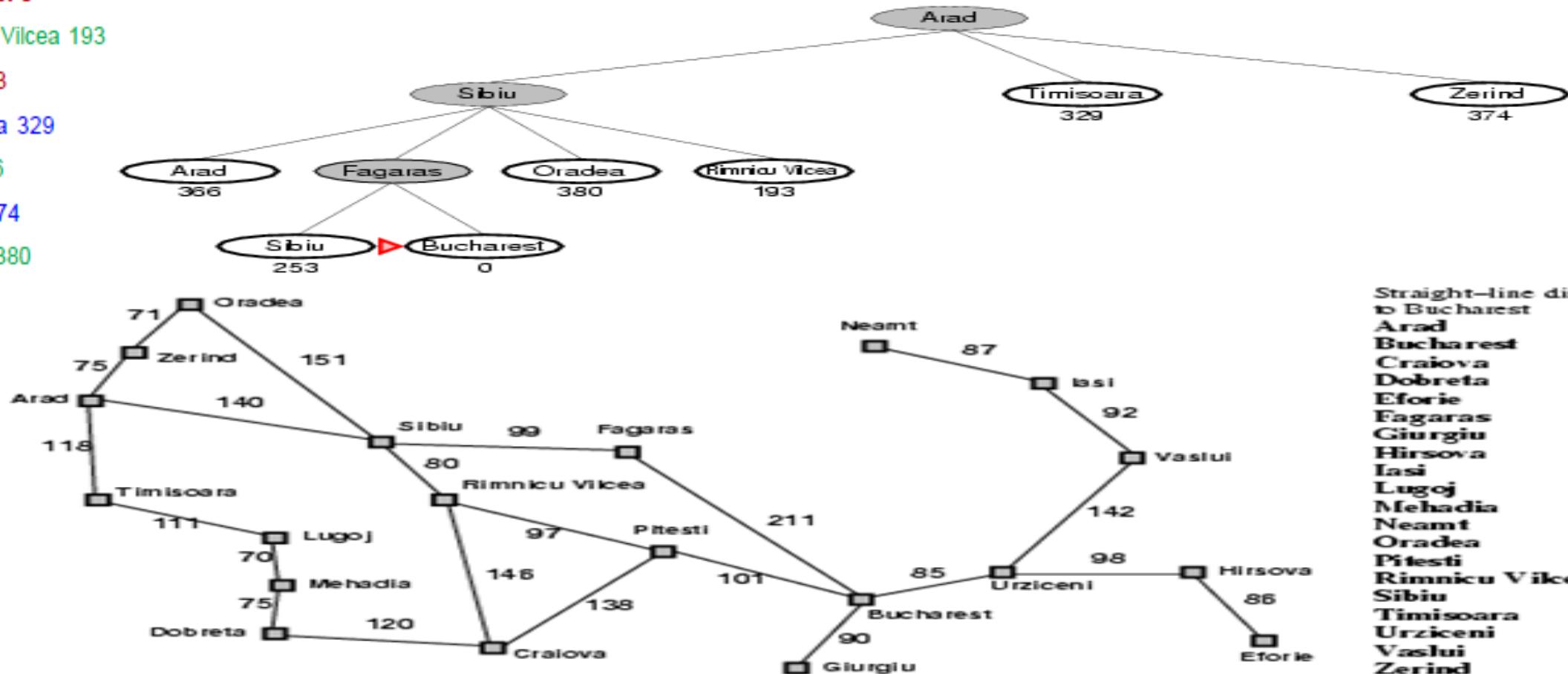
Sibiu 253

Timisoara 329

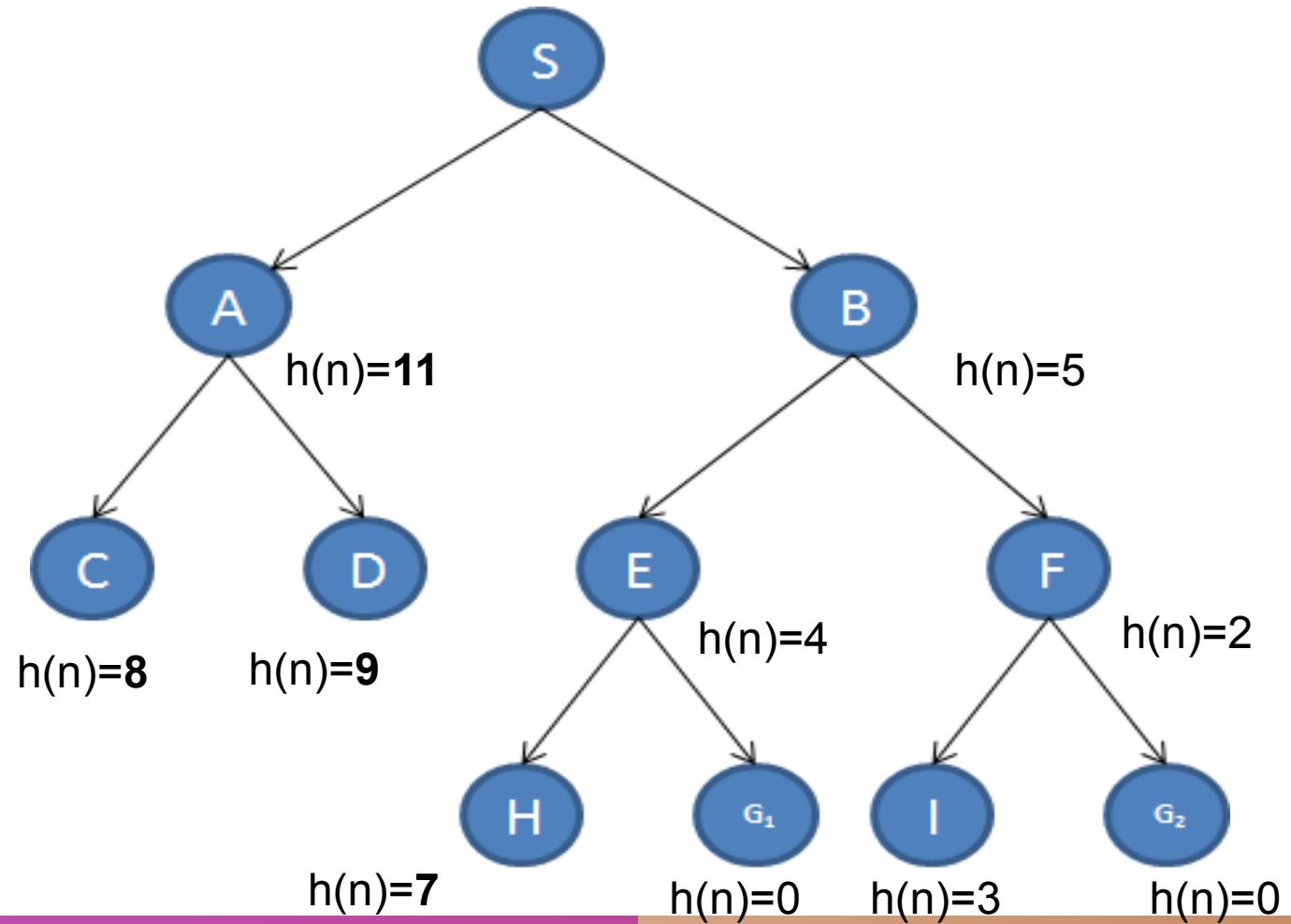
Arad 366

Zerind 374

Oradea 380

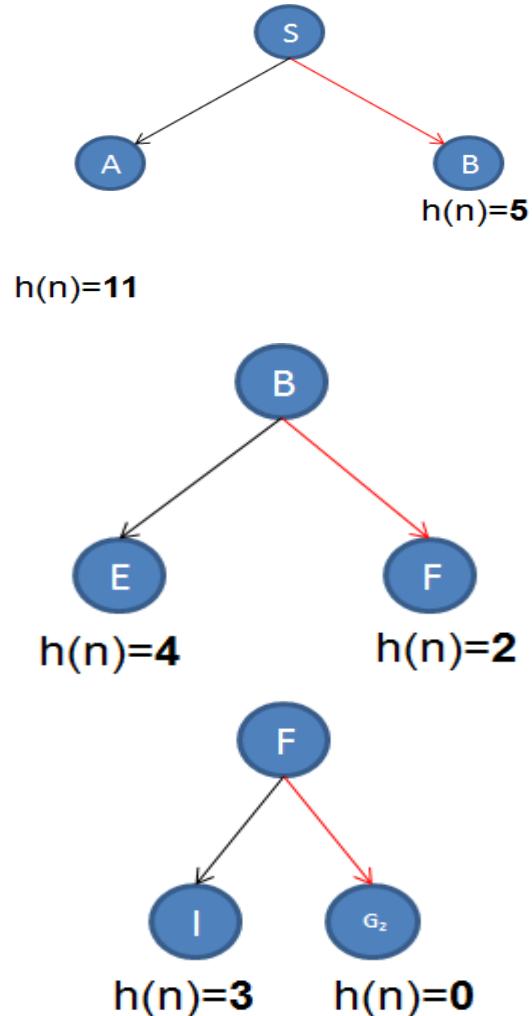


GREEDY BFS EXAMPLE



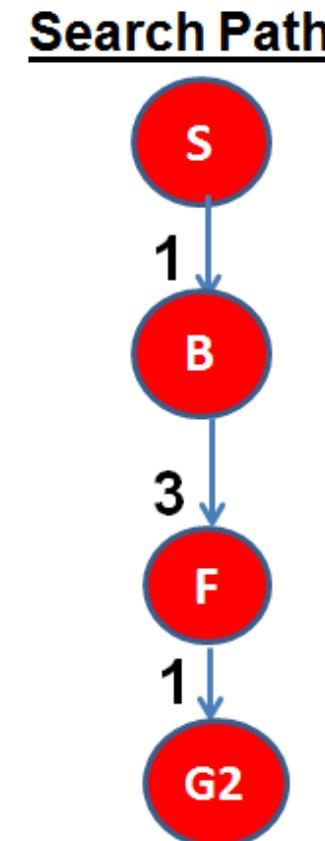
node	$h(n)$
A	11
B	5
C	9
D	8
E	4
F	2
H	7
I	3

GREEDY BFS SOLUTION



open=[S]; closed=[]
open=[B₅, A₁₁]; closed=[S]
open=[F₂, E₄, A₁₁]; closed=[S, B]
open=[G2₀, I₃, E₄, A₁₁]; closed=[S, B, F]
open=[I₃, E₄, A₁₁]; closed=[S, B, F, G2]

Solution:
S , B , F , G₂
Cost = 1+3+1=5



PROPERTIES OF GREEDY BEST-FIRST SEARCH

Complete?

- No – can get stuck in loops,

Time?

- $O(bm)$, but a good heuristic can give dramatic improvement

Space?

- $O(bm)$ -- keeps all nodes in memory

Optimal?

- No
- **Obtain best solution than best-first. But not guaranteed the optimum solution**

A* SEARCH



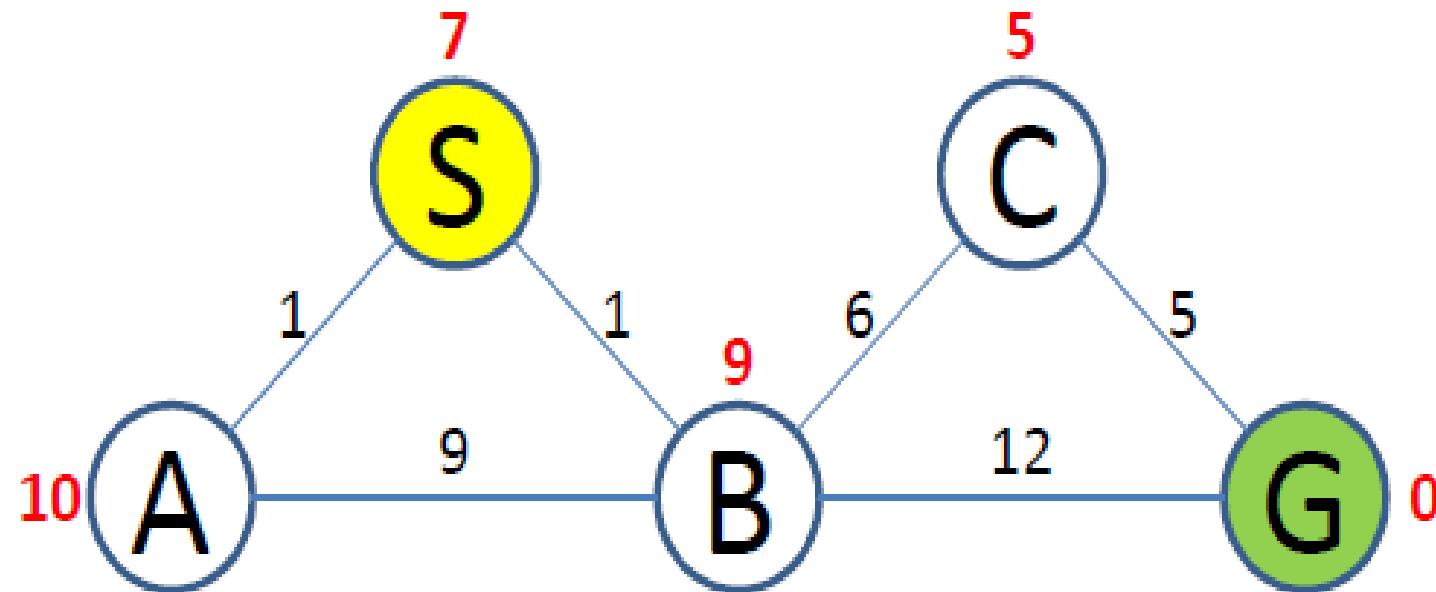
A* SEARCH ALGORITHM

- GBFS uses minimal estimated cost $h(n)$ to goal node reduce search cost considerably. Unfortunately, it is neither optimal nor complete cost to the goal. Uniform-cost search, uses minimal cost of the path so far, $g(r_i)$; it is optimal and complete, but can be very inefficient.
- A* combines the two evaluation functions. A* is the best search for minimizing the total estimated solution cost.
- **Idea:** avoid expanding paths that are already expensive
- **Evaluation function $f(n) = g(n) + h(n)$** // Equals to traveled distance plus predicted distance ahead.
 - **$g(n)$** = cost so far to reach n . $g(n)$ is called the **backward cost**
 - **$h(n)$** = estimated cost from n to goal . $h(n)$ is called the **forward cost**
 - **$f(n)$** = estimated total cost of path through n to goal

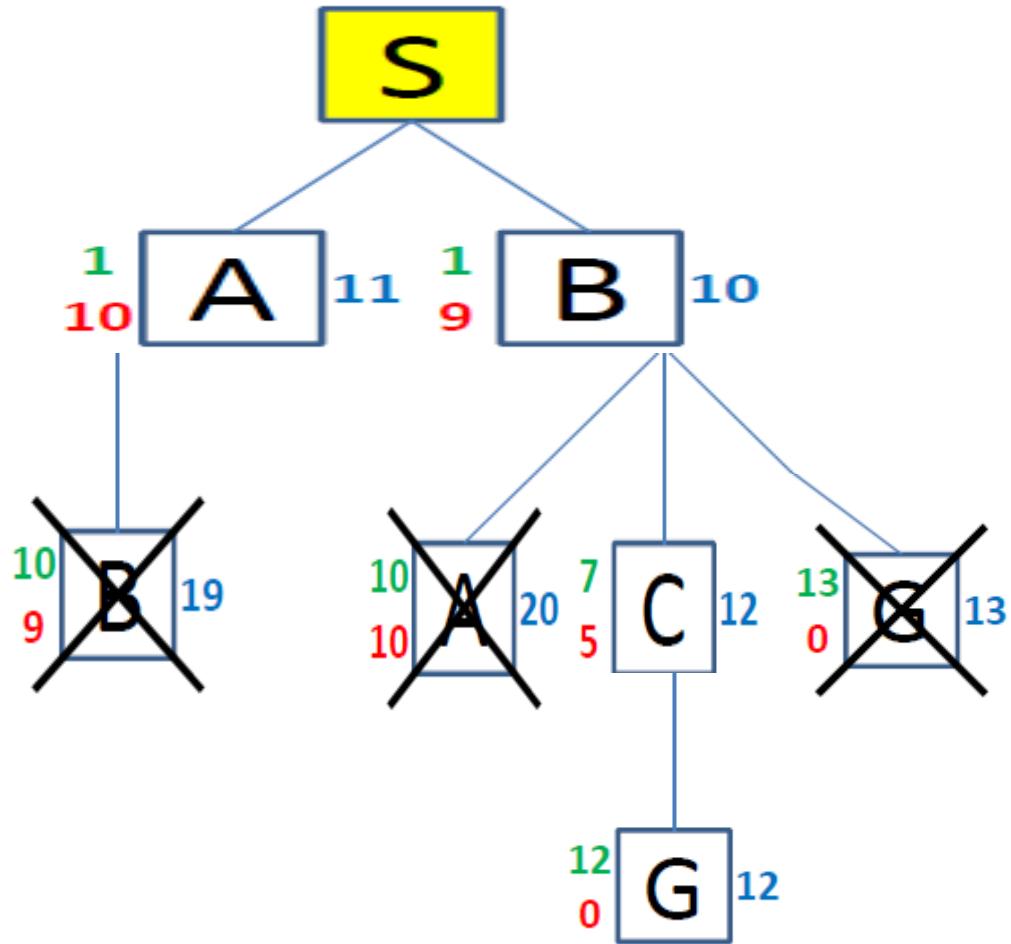
A* AND ADMISSIBILITY OF HEURISTIC FUNCTION

- The choice of an appropriate heuristic evaluation function, $h(n)$, is still crucial to the behavior of this algorithm.
- If its guaranteed that there is **no overestimate** for $h(n)$ ie If $h(n) \leq h^*(n)$ for all n , then $f(n) = h(n) + g(n)$ gives the **shortest path**, where $h(n)$ is theis the **heuristic estimate of the cost of getting to a goal node from n** and $h^*(n)$ is the **cost of an optimal path from n to a goal node**. If $h(n) \leq h^*(n)$ then $h(n)$ is called **Admissible heuristic**
- Choose a heuristic evaluation function, $h(n)$ which is as close as possible to the actual cost of getting to a goal state. Ie, choose a function $h(n)$ which never overestimates the actual cost of getting to the goal state.
- An **admissible heuristic never overestimates the cost to reach the goal**, i.e., it is optimistic
- Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs
- Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe

EXERCISE-1 ON A* ALGORITHM



SOLUTION



QUEUE:

S7

B10 A11

A11 C12 G13

C12 G12 G13 B19 A20

G12 G13 B19 A20

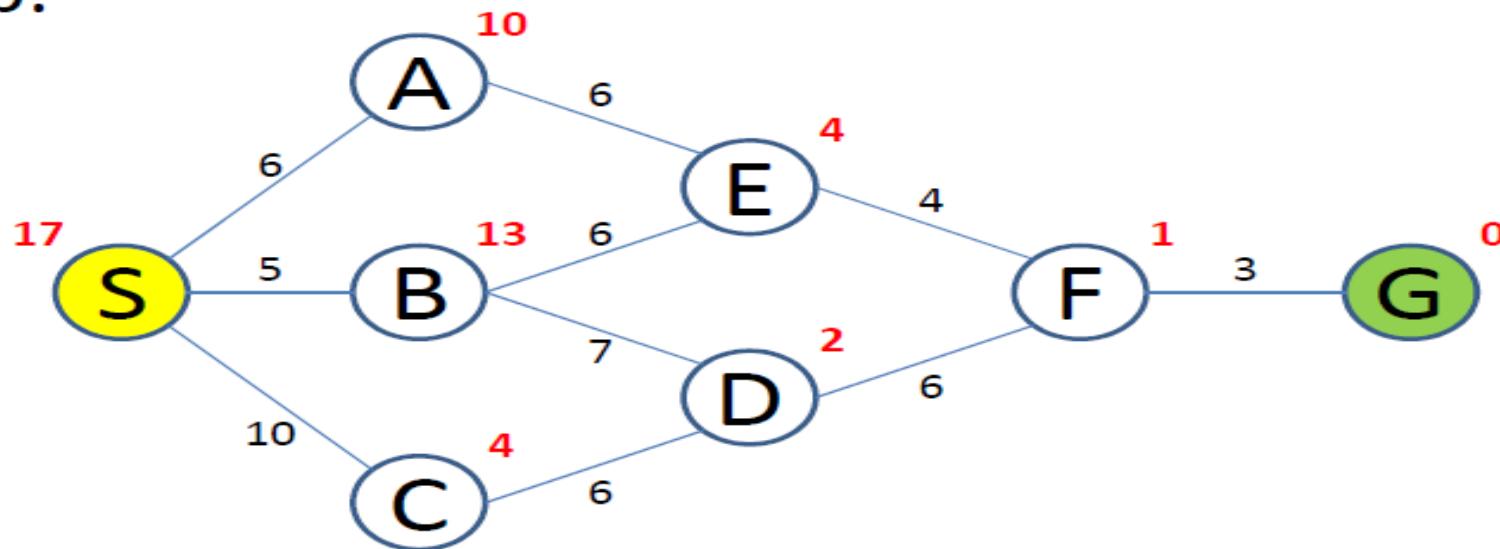
- **Frontier: Priority Queue (with priority to least $F(n)$)**
- **Solution path found is S B C G, cost 12**
- **Number of nodes expanded (including goal node) = 4**

THINGS TO BE TAKEN CARE WHILE IMPLEMENTING A*

1. Push a node n which is already visited to fringe if the $f(n)$ value is lesser than previous $f(n)$ value. Else optimality wont be achieved.(else G19 since E15 cant be pushed in)ie We must not exclude states we have seen before, but update their information when their information when the new path is better.
2. Always check the expanded list and see if the node is visited and expand a node if $F(n)$ value is less(refer case of D in coming)
3. If its undirected graph all possible paths should be explored out even explored nodes, but exclude the predecessors in a branch. Else infinite path will arise(completeness can't be achieved)

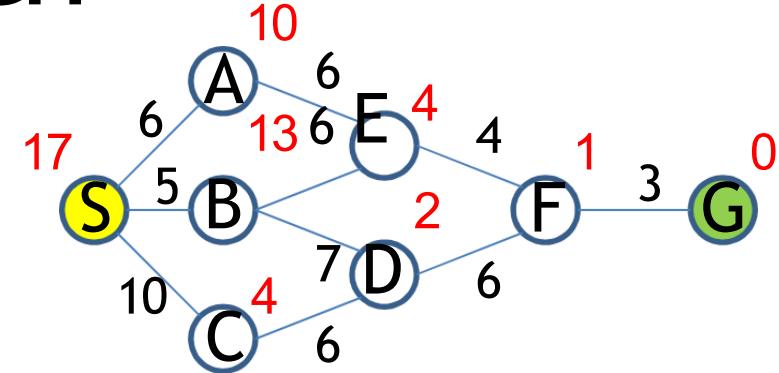
EXERCISE-2 ON A* ALGORITHM

Perform the A* Algorithm on the following figure. Explicitly write down the queue at each step.



A* Search

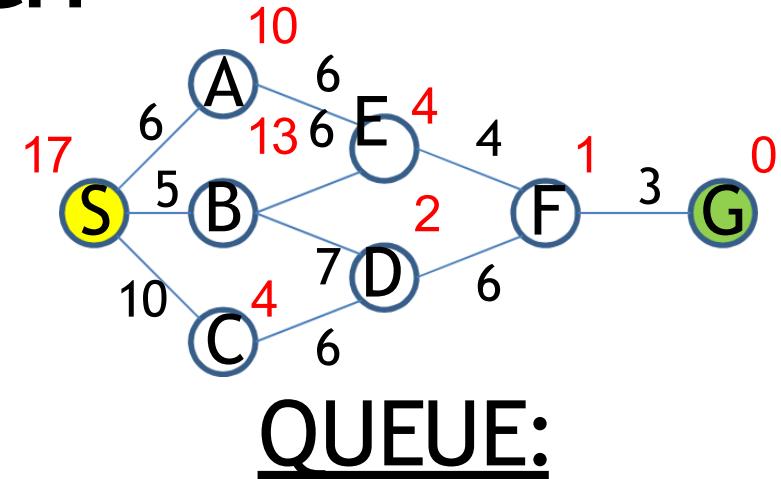
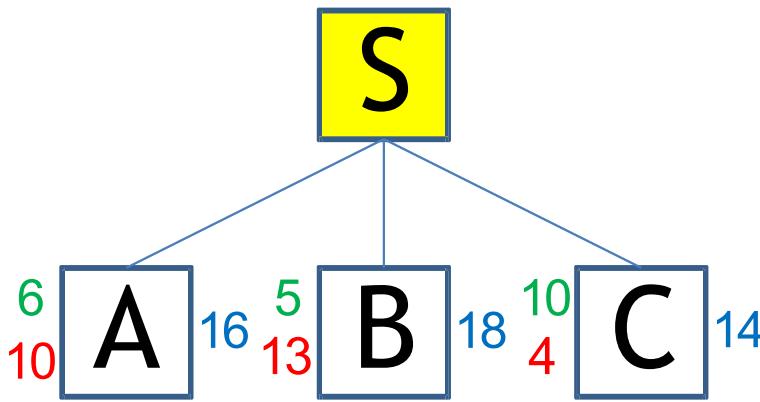
0
17 **S** 17



QUEUE:
 S^{17}

Expanded Node	Fringe Queue
--	S^{17}

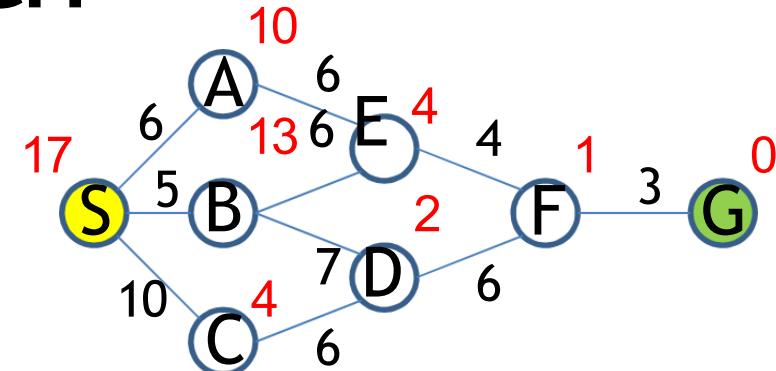
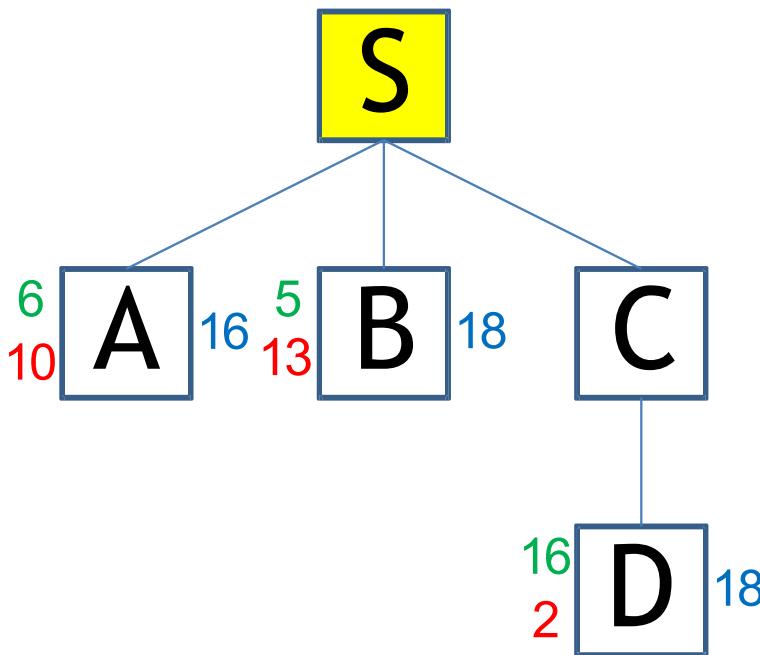
A* Search



S17 C14 A16 B18

Expanded Node	Fringe Queue
--	S ₁₇
S ₁₇	C ₁₄ A ₁₆ B ₁₈

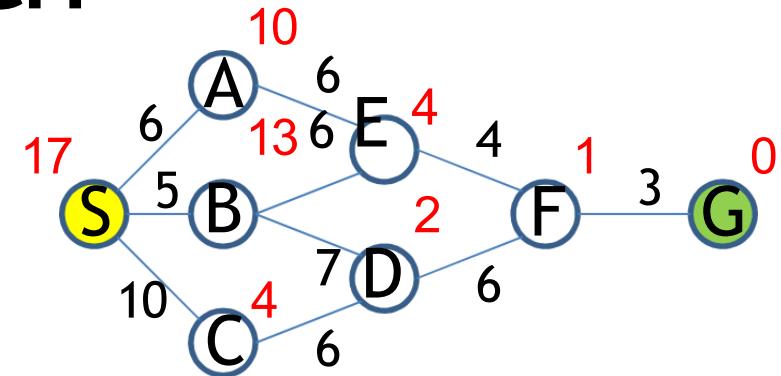
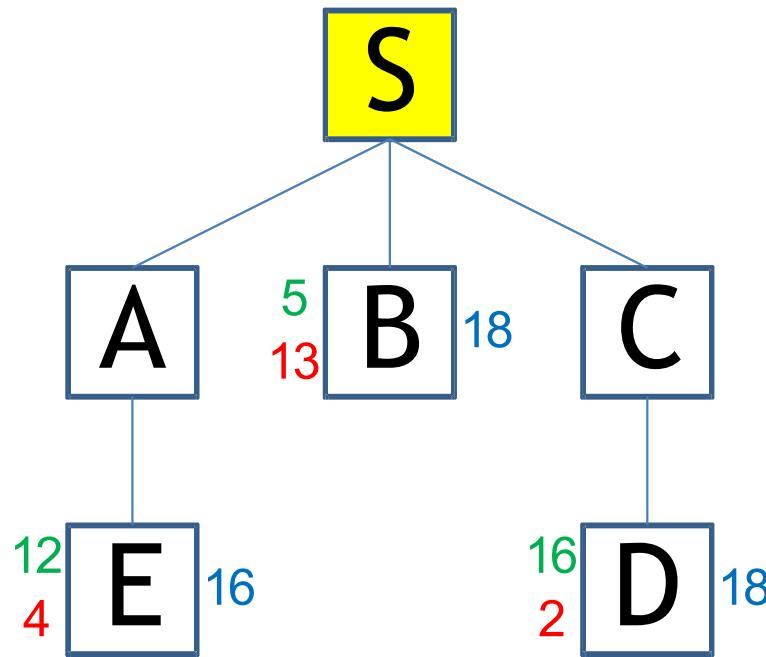
A* Search



QUEUE:
 $S^{17} C^{14} A^{16} B^{18} D^{18}$

Expanded Node	Fringe Queue
--	S^{17}
S^{17}	$C^{14} A^{16} B^{18}$
C^{14}	$A^{16} B^{18} D^{18}$

A* Search

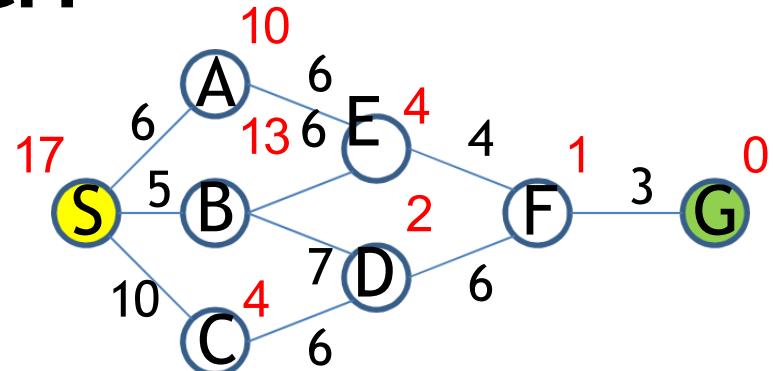
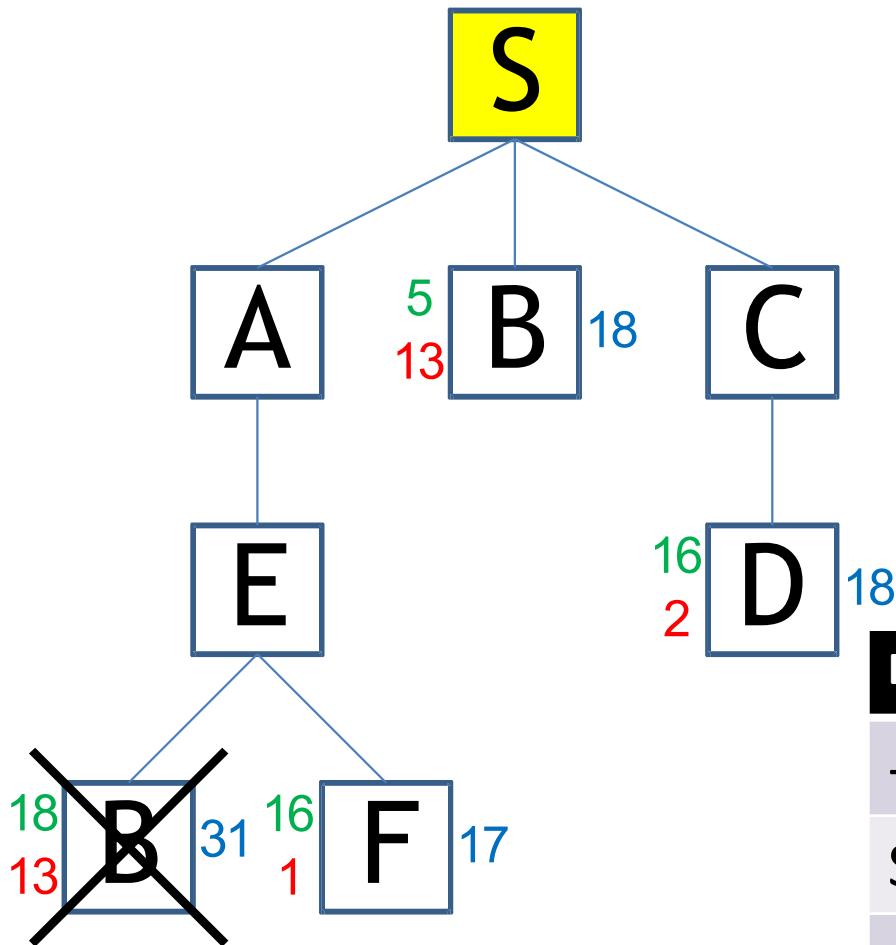


QUEUE:

$S^{17} C^{14} A^{16} E^{16} B^{18} D^{18}$

Expanded Node	Fringe Queue
--	S^{17}
S^{17}	$C^{14} A^{16} B^{18}$
C^{14}	$A^{16} B^{18} D^{18}$
A^{16}	$E^{16} B^{18} D^{18}$

A* Search

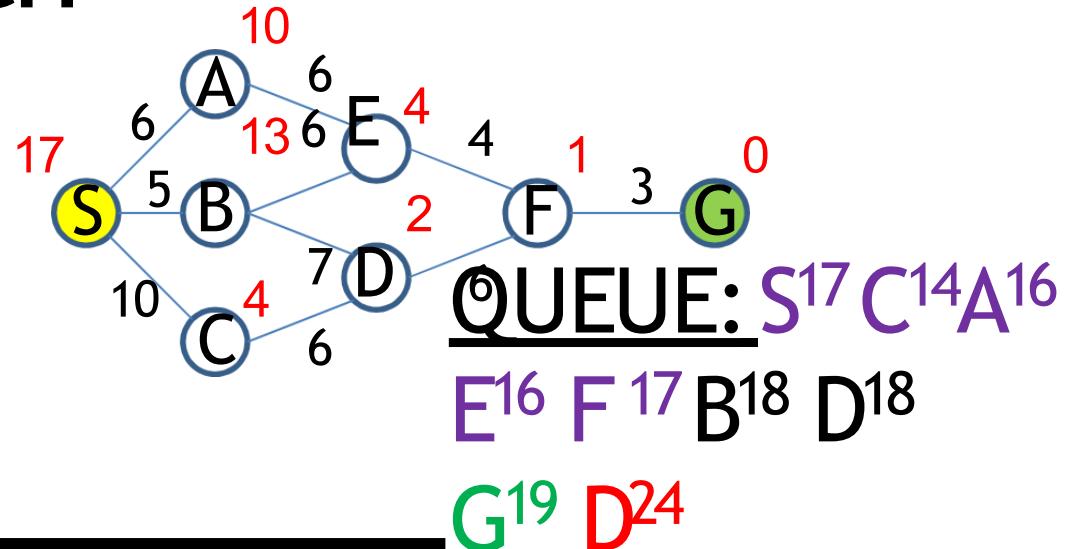
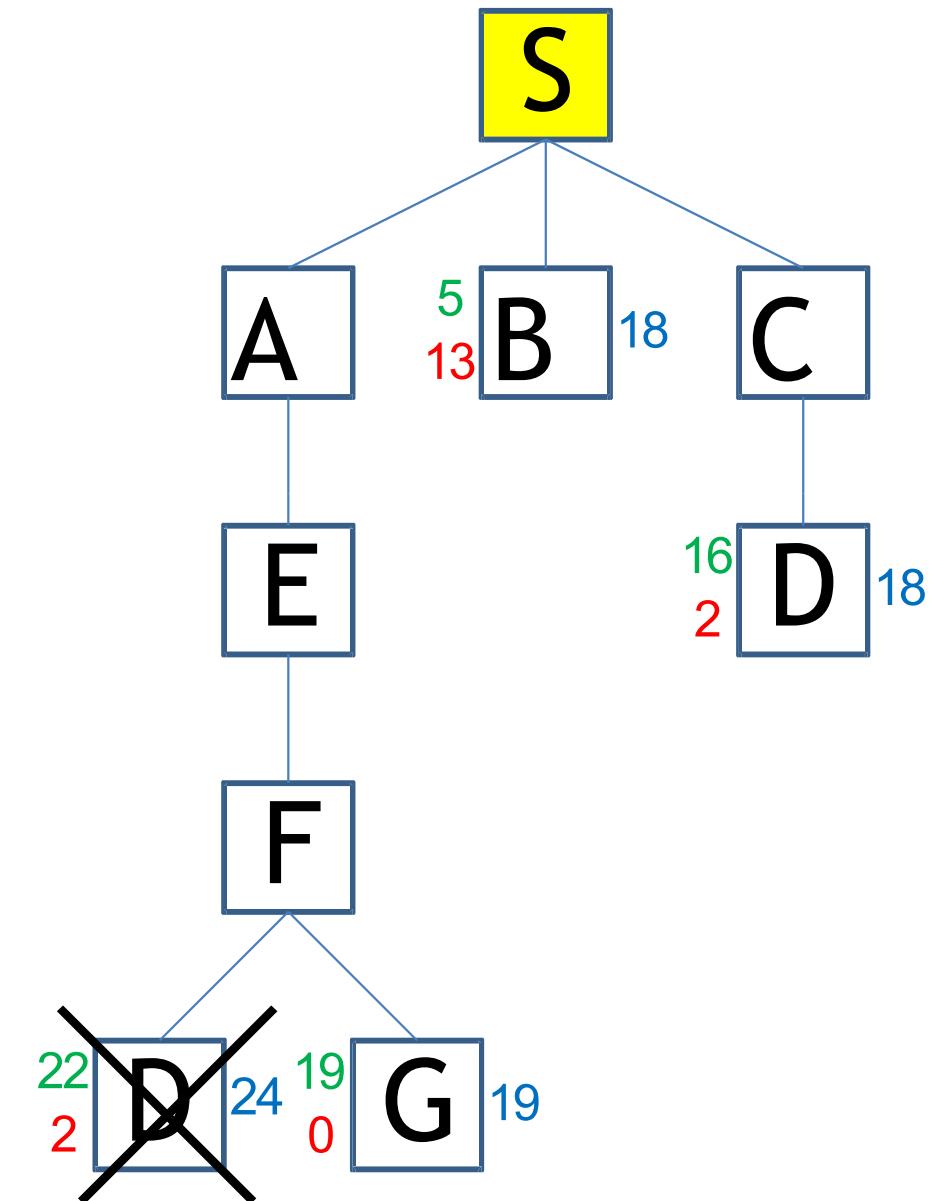


QUEUE:

$S^{17} C^{14} A^{16} E^{16} F^{17} B^{18} D^{18} B^{31}$

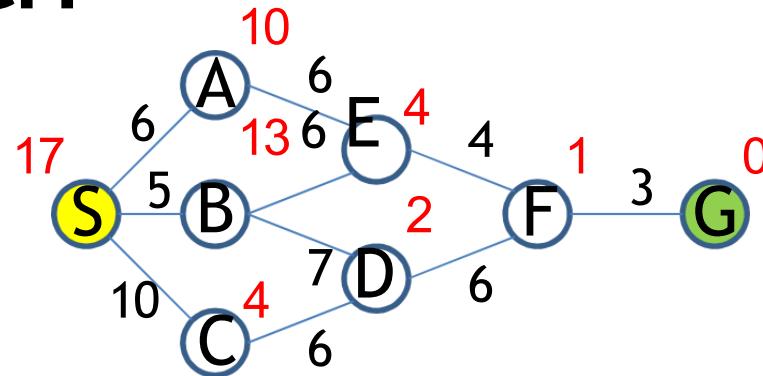
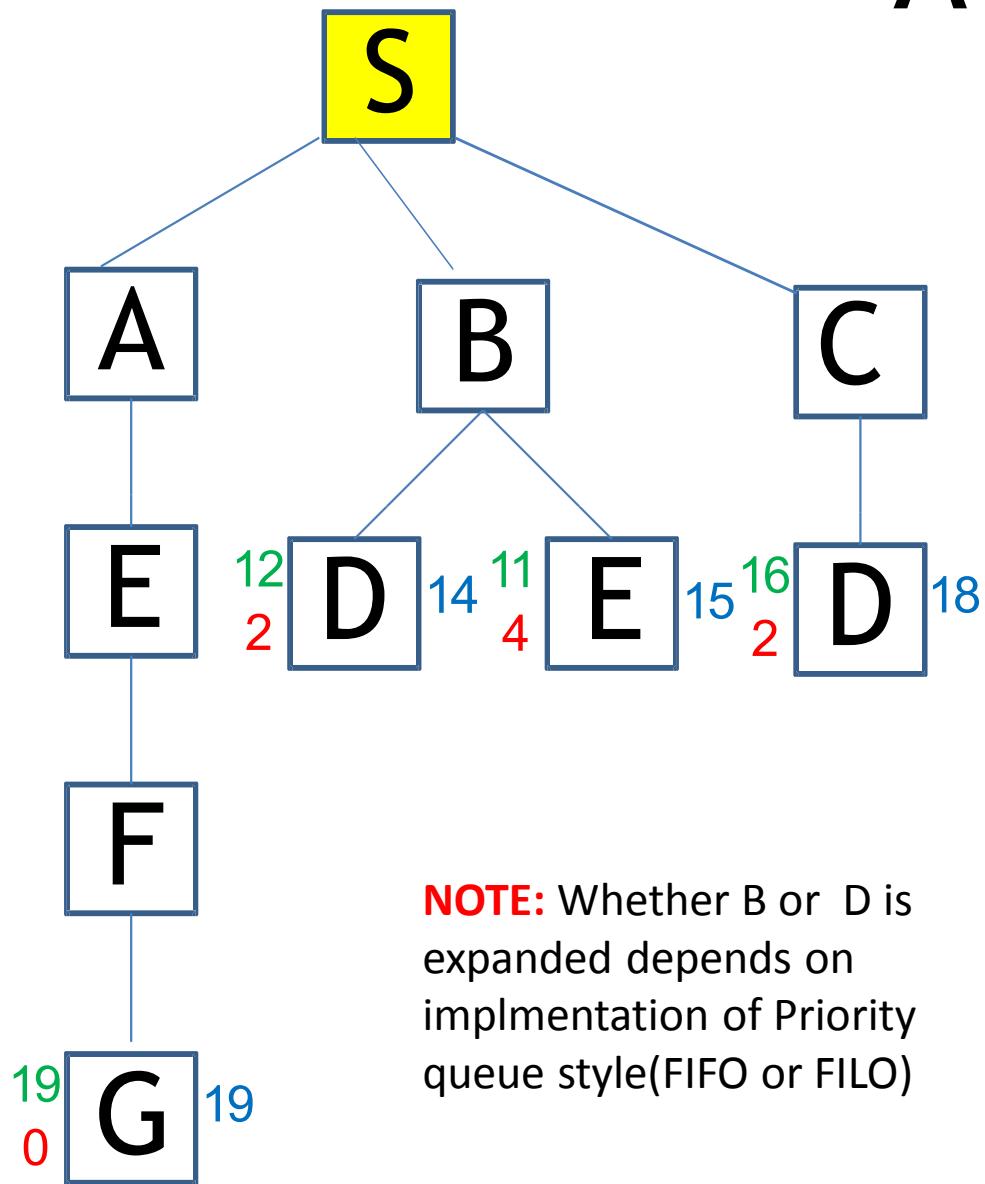
Expanded Node	Fringe Queue
--	S^{17}
S^{17}	$C^{14} A^{16} B^{18}$
C^{14}	$A^{16} B^{18} D^{18}$
A^{16}	$E^{16} B^{18} D^{18}$
E^{16}	$F^{17} B^{18} D^{18}$

A* Search



Expanded Node	Fringe Queue
--	S^{17}
S^{17}	$C^{14}A^{16} B^{18}$
C^{14}	$A^{16} B^{18} D^{18}$
A^{16}	$E^{16} B^{18} D^{18}$
E^{16}	$F^{17} B^{18} D^{18}$
F^{17}	$B^{18} D^{18} G^{19} D^{24}$

A* Search

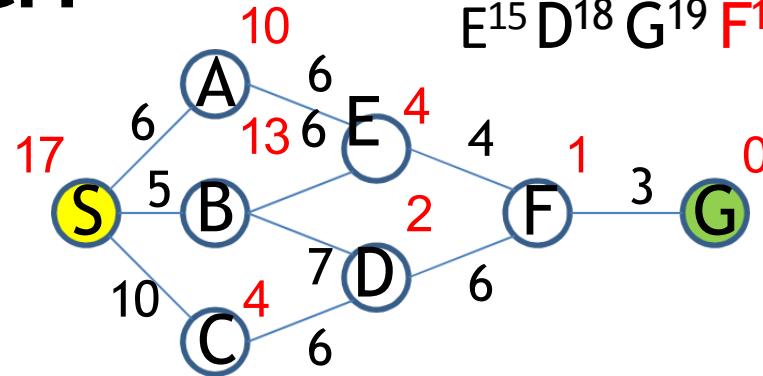
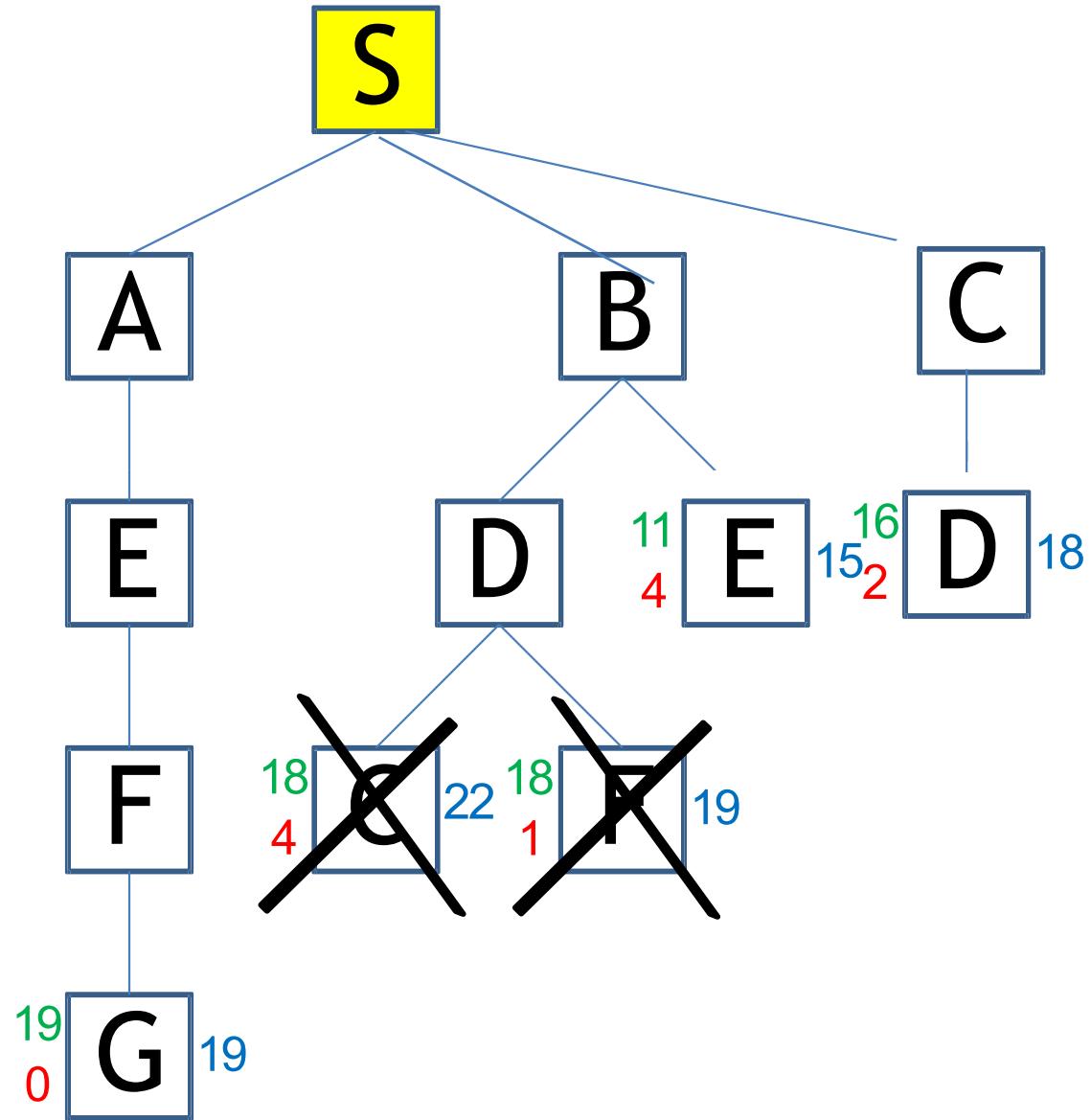


Expanded Node	Fringe Queue
--	S ₁₇
S ₁₇	C ₁₄ A ₁₆ B ₁₈
C ₁₄	A ₁₆ B ₁₈ D ₁₈
A ₁₆	E ₁₆ B ₁₈ D ₁₈
E ₁₆	F ₁₇ B ₁₈ D ₁₈
F ₁₇	B ₁₈ D ₁₈ G ₁₉
B ₁₈	D ₁₄ E ₁₅ D ₁₈ G ₁₉

QUEUE:

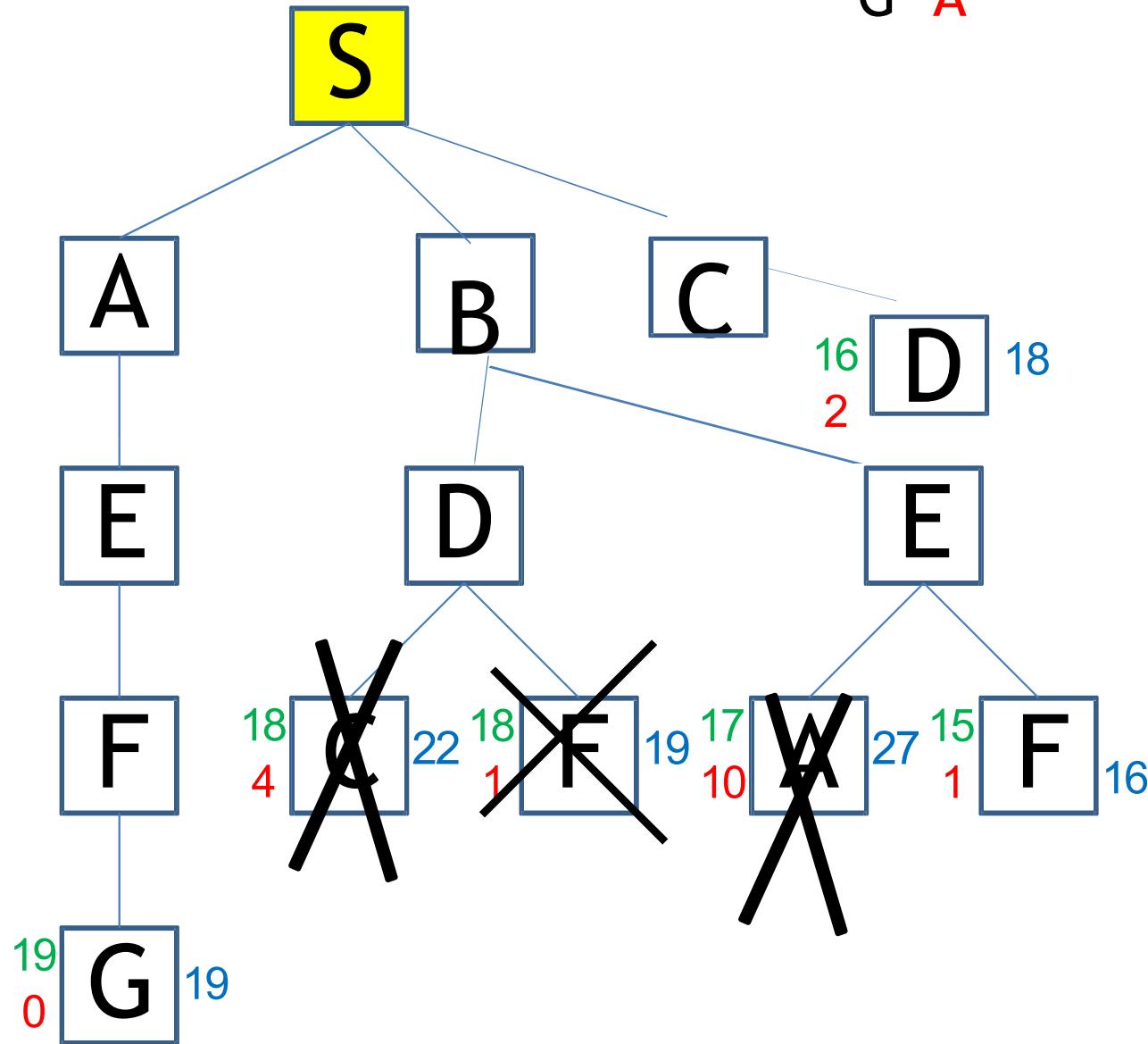
S₁
 7 C₁₄ A₁₆ E₁₆
 F₁₇ B₁₈ D₁₄
 E₁₅ D₁₈ G₁₉

A* Search

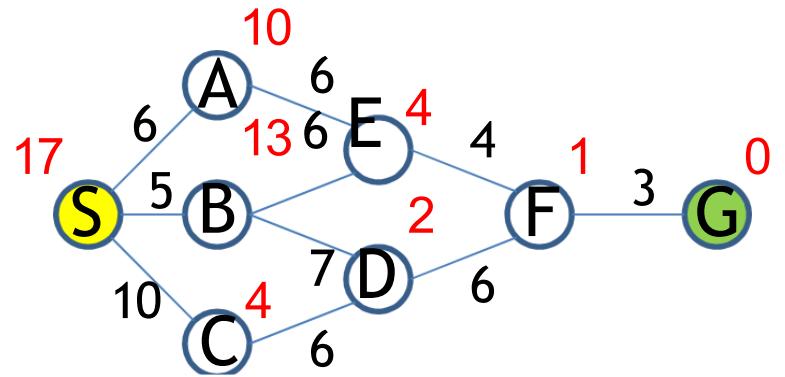


Expanded Node	Fringe Queue
--	S^{17}
S^{17}	$C^{14}A^{16} B^{18}$
C^{14}	$A^{16} B^{18} D^{18}$
A^{16}	$E^{16} B^{18} D^{18}$
E^{16}	$F^{17} B^{18} D^{18}$
F^{17}	$B^{18} D^{18} G^{19}$
B^{18}	$D^{14} E^{15} D^{18} G^{19}$
D^{14}	$E^{15} D^{18} G^{19}$

A* Search

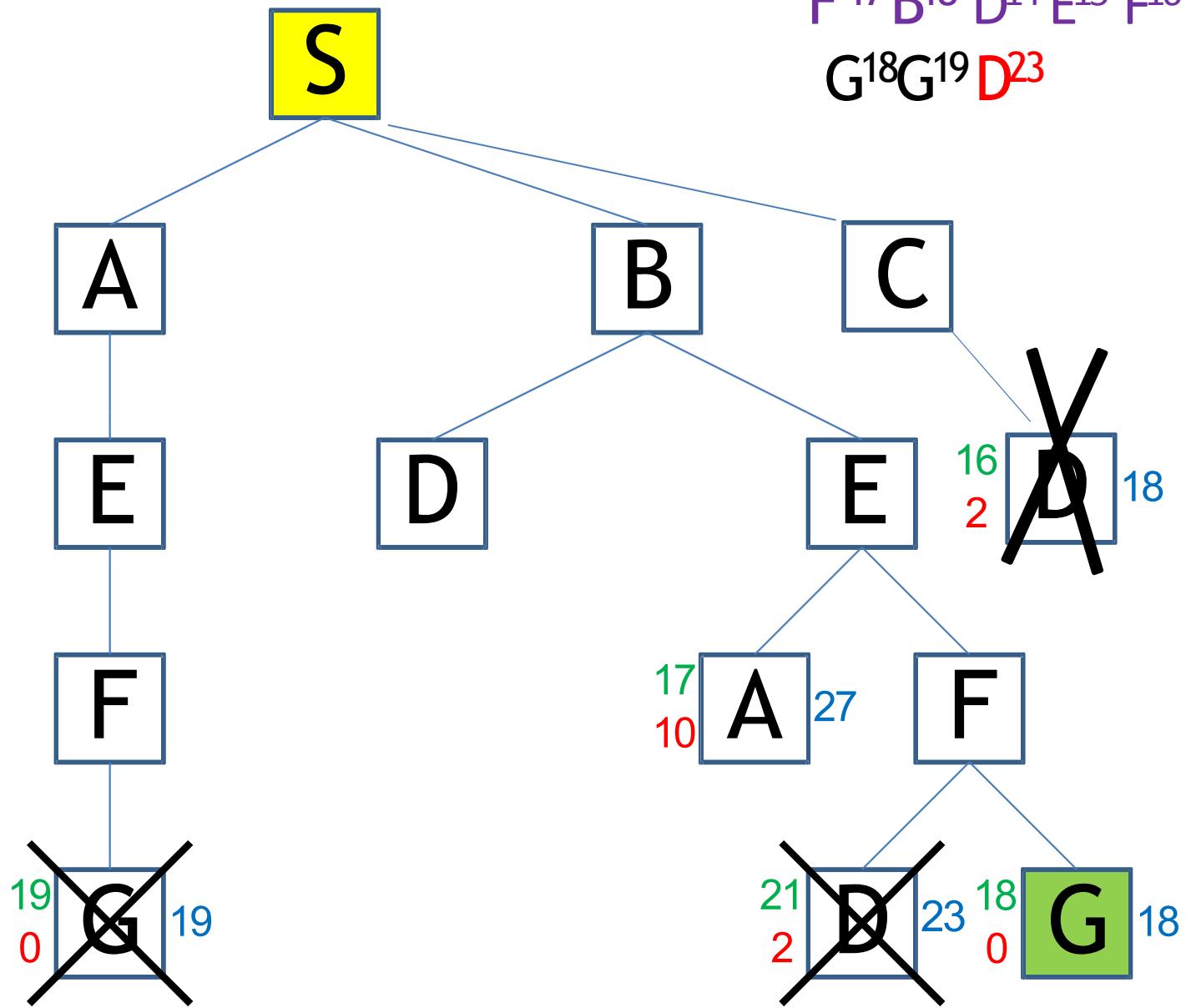


QUEUE: $S^{17}C^{14}A^{16}E^1$
 $6F^{17}B^{18}D^{14}E^{15}F^{16}D^{18}$
 $G^{19}A^{27}$



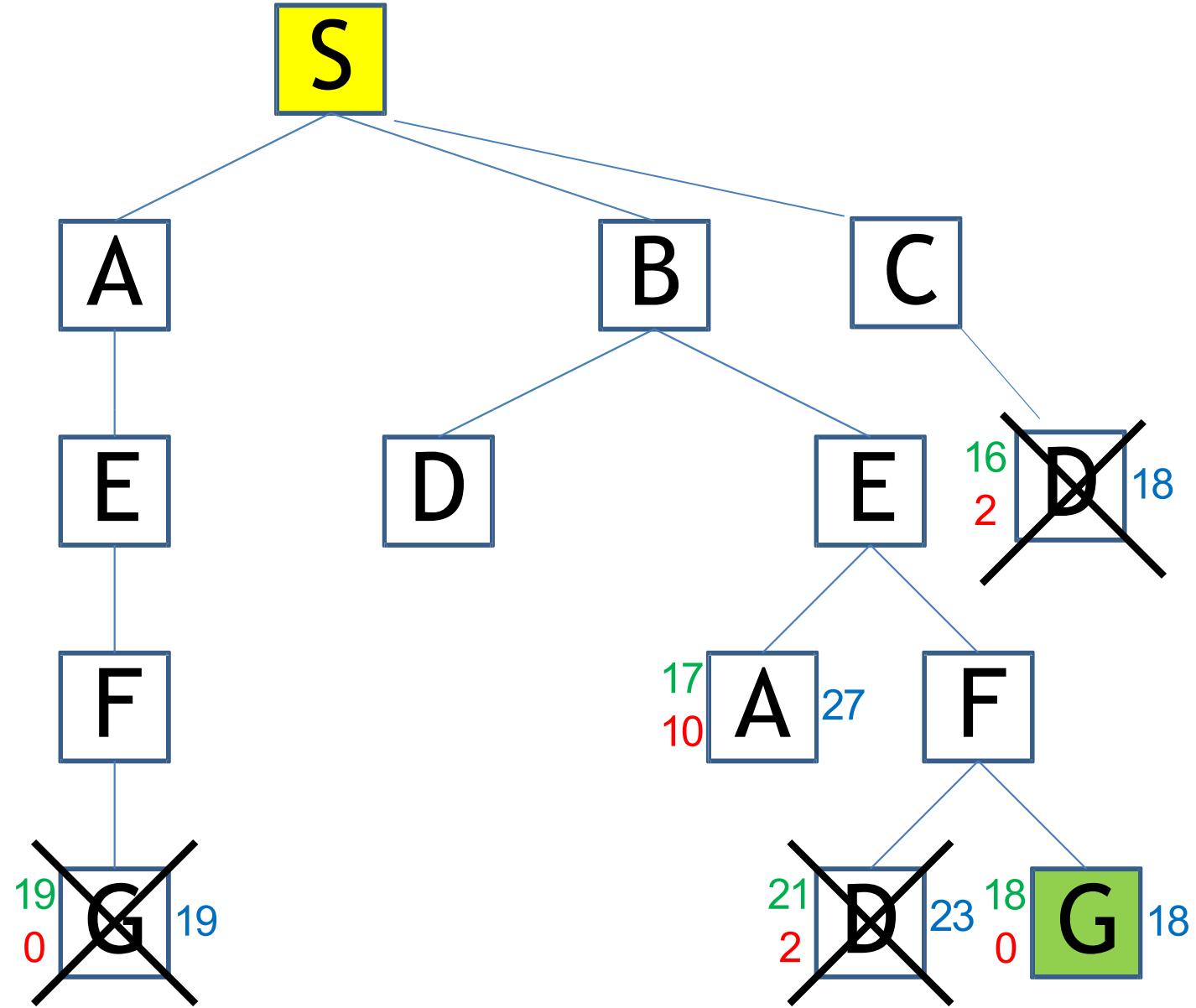
Expanded Node	Fringe Queue
--	S^{17}
S^{17}	$C^{14}A^{16}B^{18}$
C^{14}	$A^{16}B^{18}D^{18}$
A^{16}	$E^{16}B^{18}D^{18}$
E^{16}	$F^{17}B^{18}D^{18}$
F^{17}	$B^{18}D^{18}G^{19}$
B^{18}	$D^{14}E^{15}D^{18}G^{19}$
D^{14}	$E^{15}D^{18}G^{19}$
E^{15}	$F^{16}D^{18}G^{19}$

A* Search



Expanded Node	Fringe Queue
--	S^{17}
S^{17}	$C^{14} A^{16} B^{18}$
C^{14}	$A^{16} B^{18} D^{18}$
A^{16}	$E^{16} B^{18} D^{18}$
E^{16}	$F^{17} B^{18} D^{18}$
F^{17}	$B^{18} D^{18} G^{19}$
B^{18}	$D^{14} E^{15} D^{18} G^{19}$
D^{14}	$E^{15} D^{18} G^{19}$
E^{15}	$F^{16} D^{18} G^{19}$
F^{16}	$D^{18} G^{18} G^{19}$

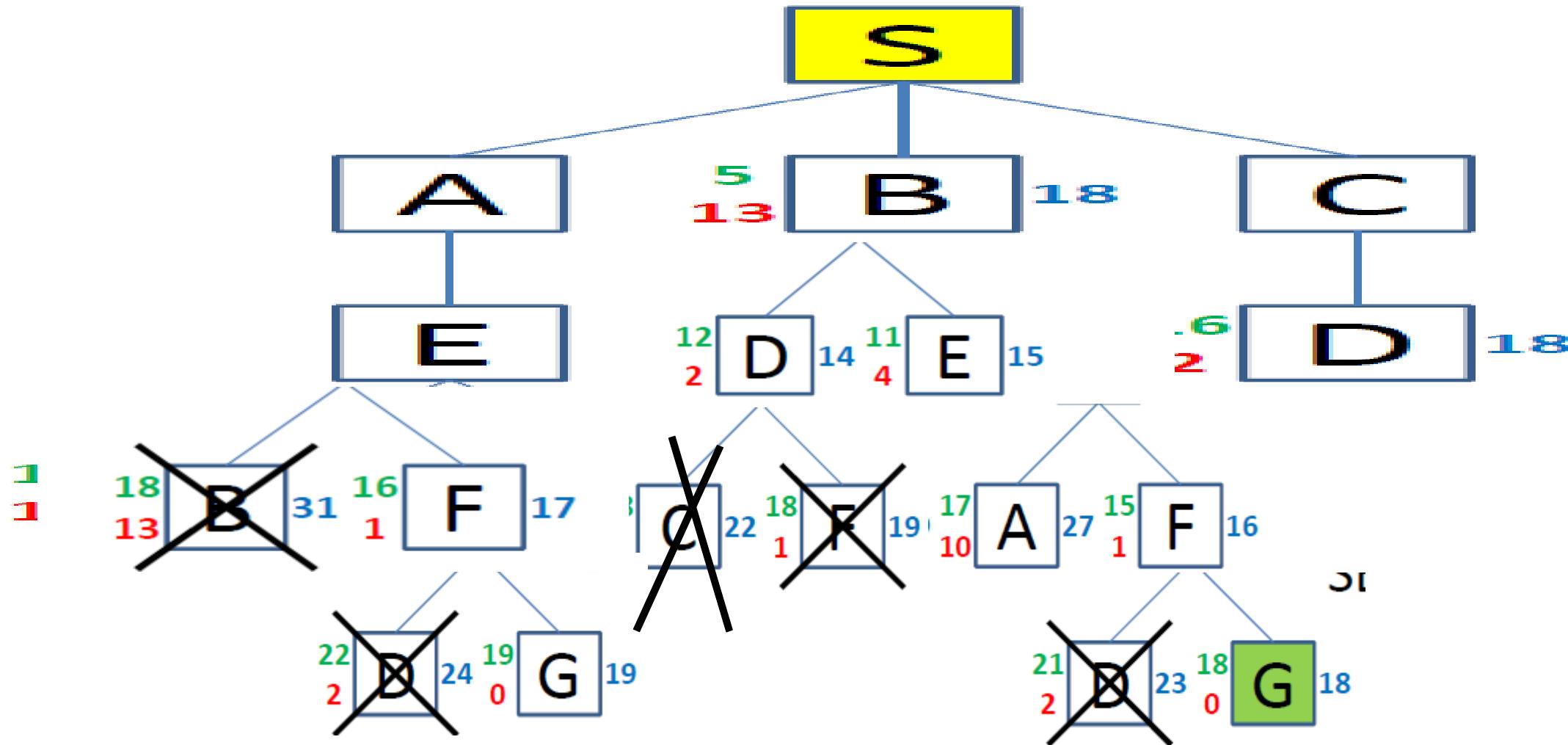
A* Search



QUEUE: SBEF
G SEFGSBD
SBEFDSBEA

Expanded Node	Fringe Queue
--	S ¹⁷
S ¹⁷	C ¹⁴ A ¹⁶ B ¹⁸
C ¹⁴	A ¹⁶ B ¹⁸ D ¹⁸
A ¹⁶	E ¹⁶ B ¹⁸ D ¹⁸
E ¹⁶	F ¹⁷ B ¹⁸ D ¹⁸
F ¹⁷	B ¹⁸ D ¹⁸ G ¹⁹
B ¹⁸	D ¹⁴ E ¹⁵ D ¹⁸ G ¹⁹
D ¹⁴	E ¹⁵ D ¹⁸ G ¹⁹
E ¹⁵	F ¹⁶ D ¹⁸ G ¹⁹
F ¹⁶	D ¹⁸ G ¹⁸ G ¹⁹
G ¹⁸	

SOLUTION

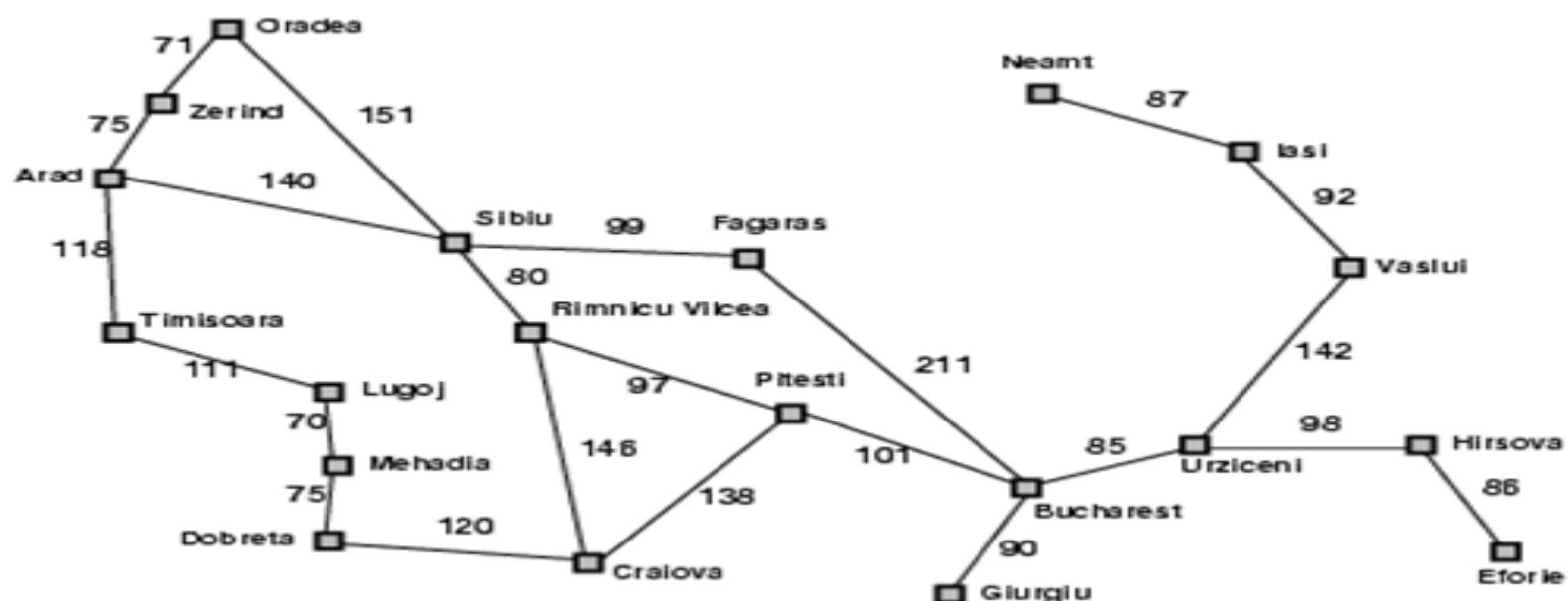


A* SEARCH EXAMPLE

Frontier queue:

Arad 366

Arad
366=0+366



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

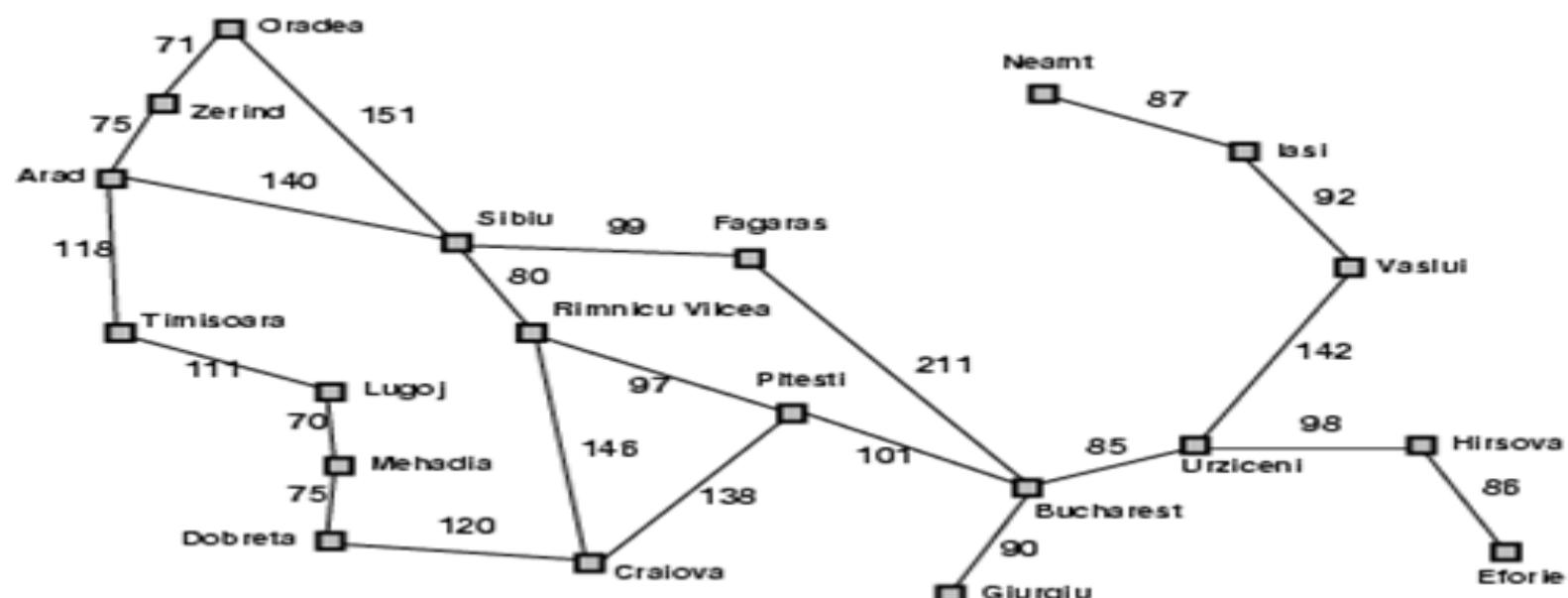
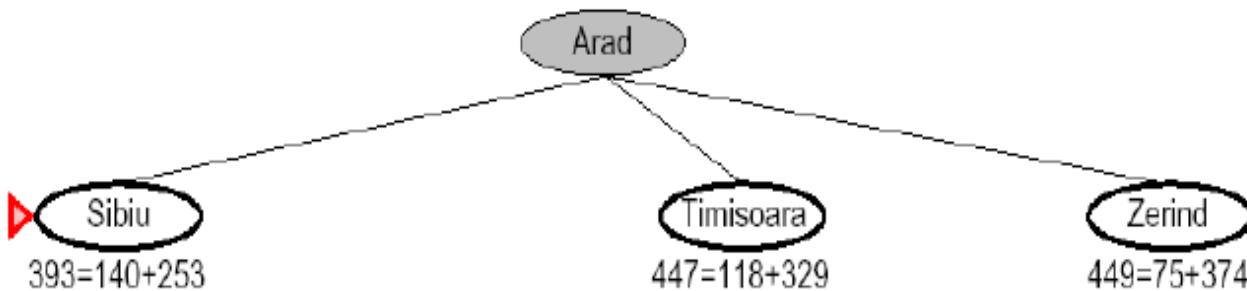
A* SEARCH EXAMPLE

Frontier queue:

Sibiu 393

Timisoara 447

Zerind 449



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

A* SEARCH EXAMPLE

Frontier queue:

Rimnicu Vilcea 413

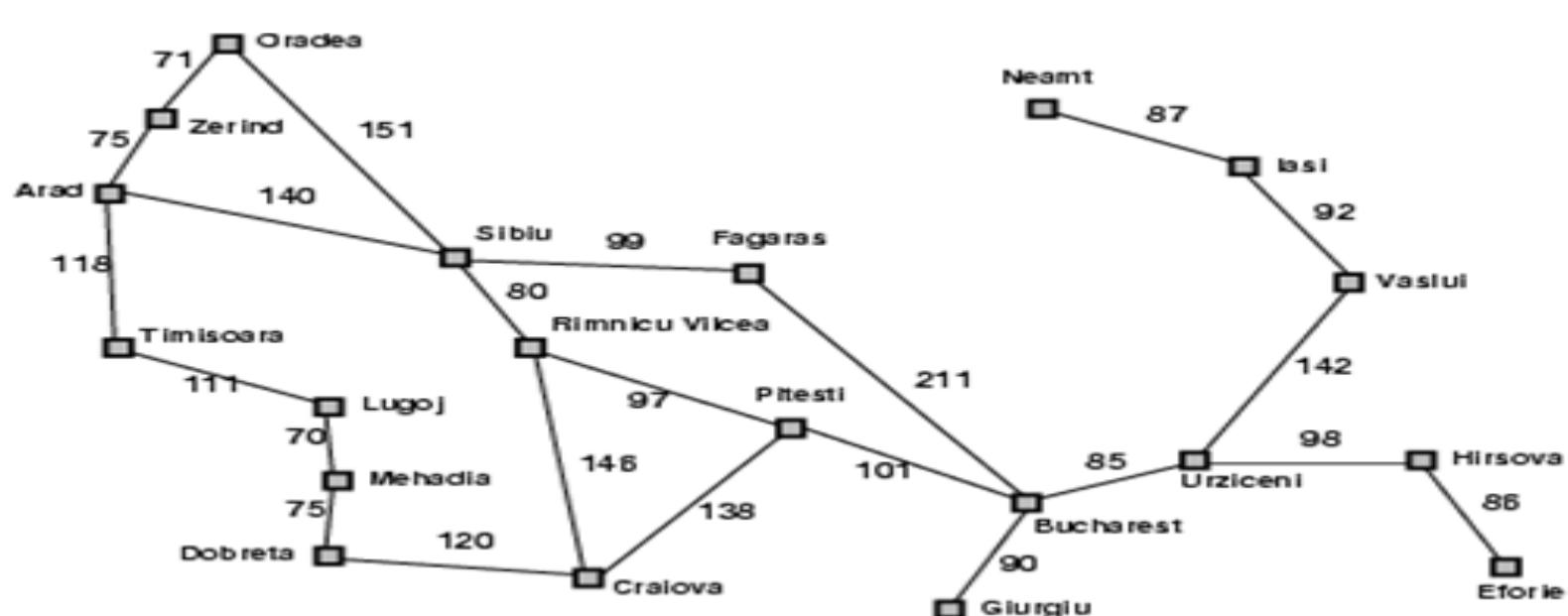
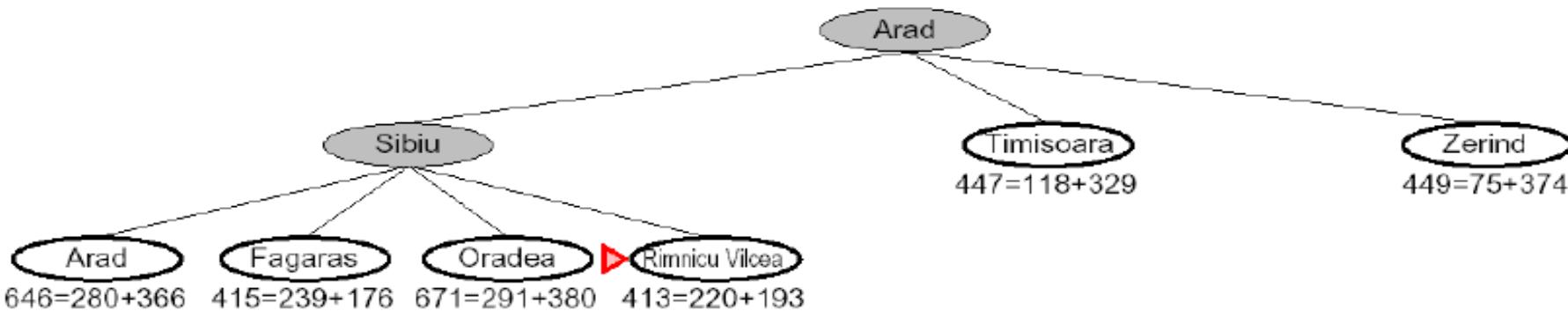
Fagaras 415

Timisoara 447

Zerind 449

Arad 646 ←

Oradea 671



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

A* SEARCH EXAMPLE

Frontier queue:

Fagaras 415

Pitesti 417

Timisoara 447

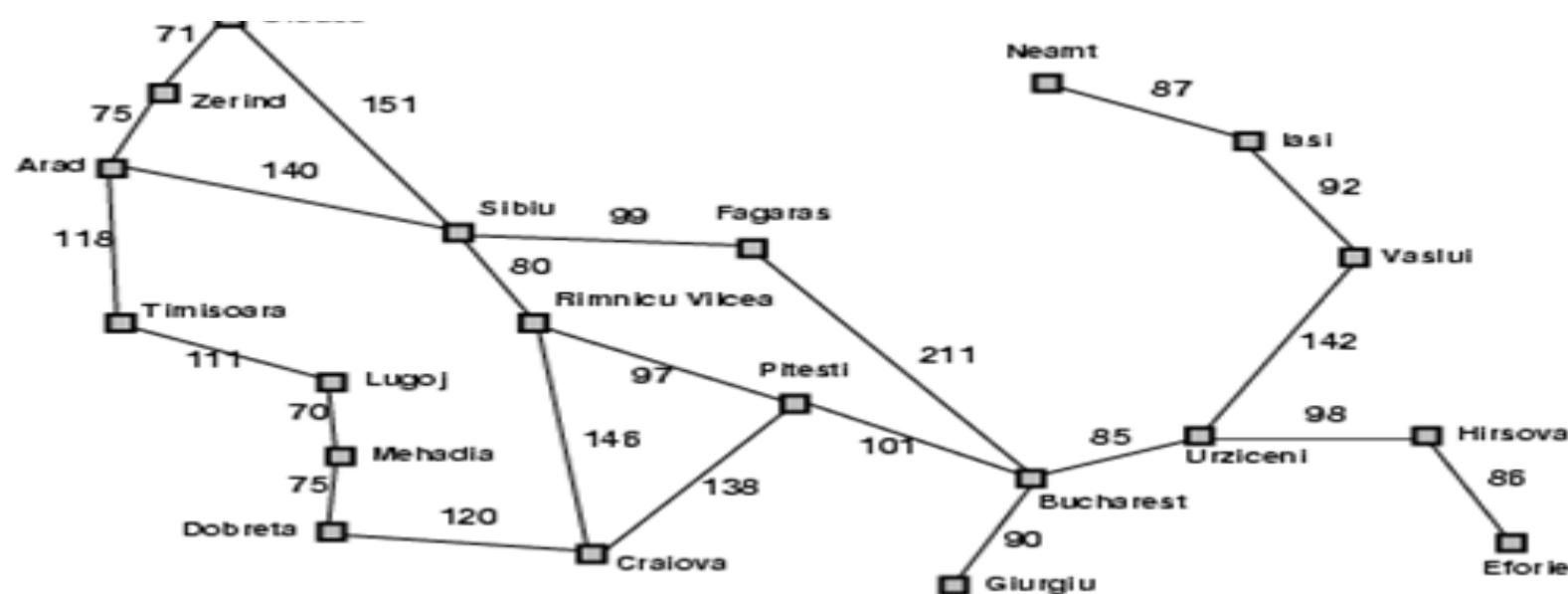
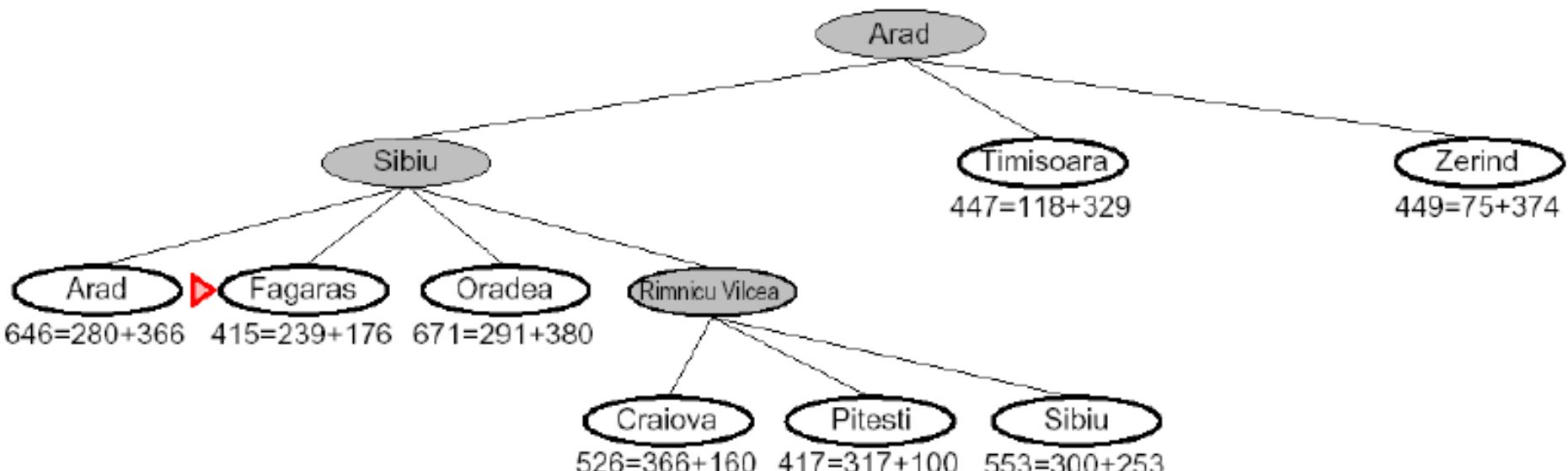
Zerind 449

Craiova 526

Sibiu 553

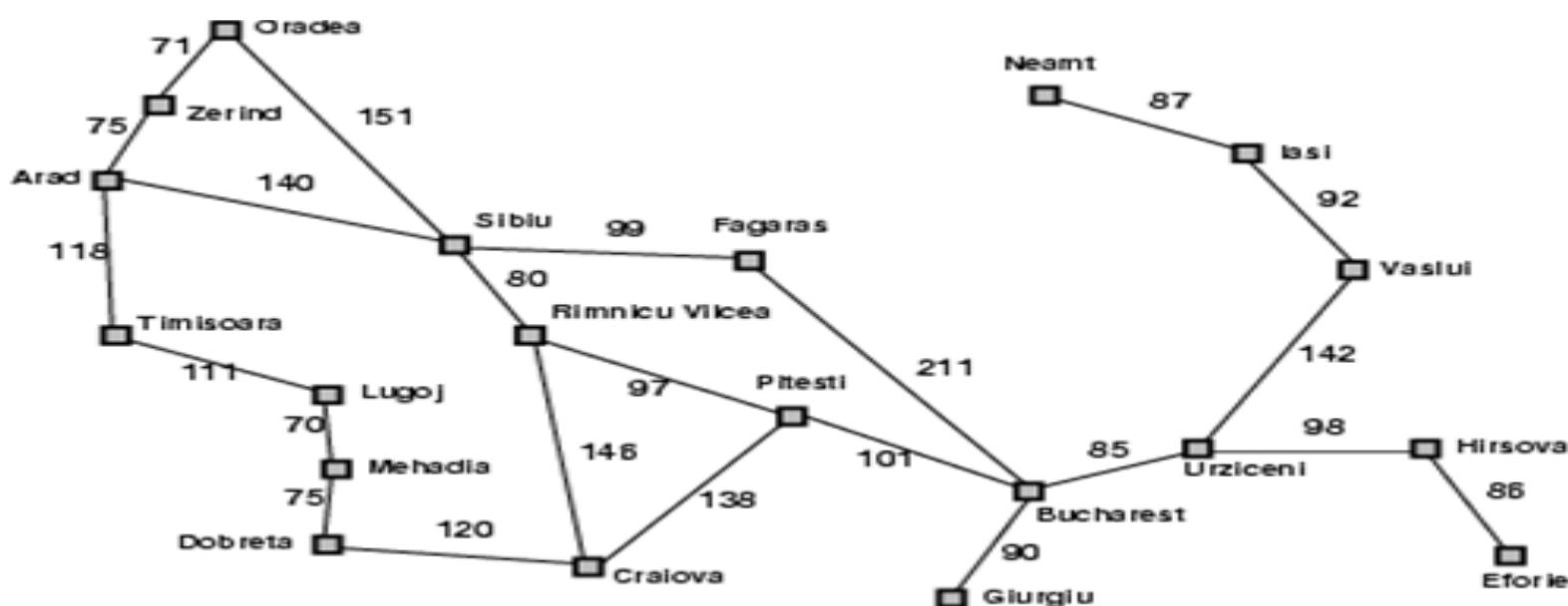
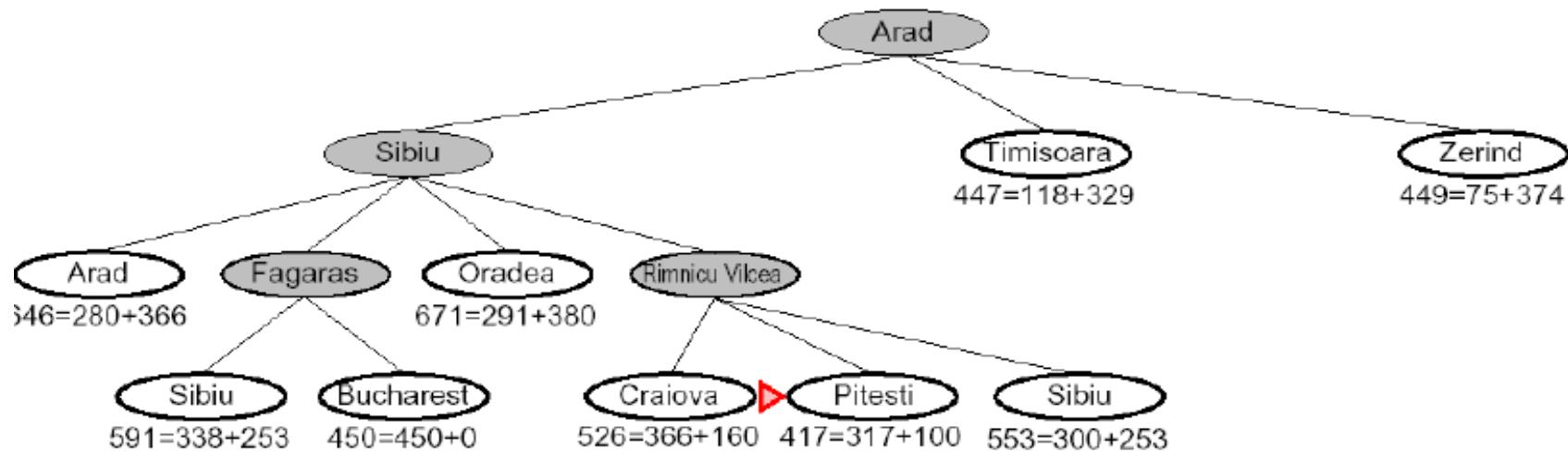
Arad 646

Oradea 671



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

A* SEARCH EXAMPLE

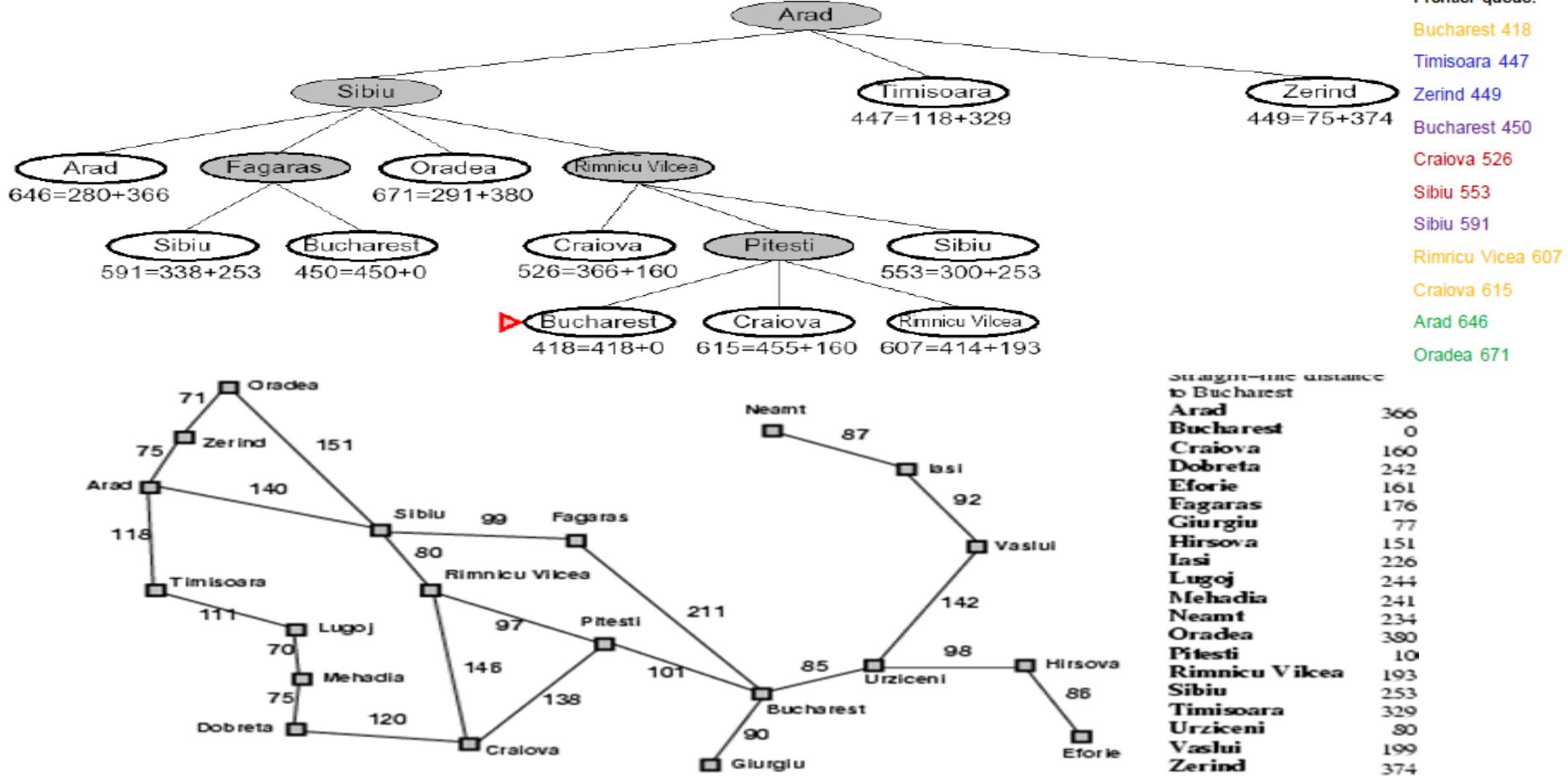


Frontier queue:

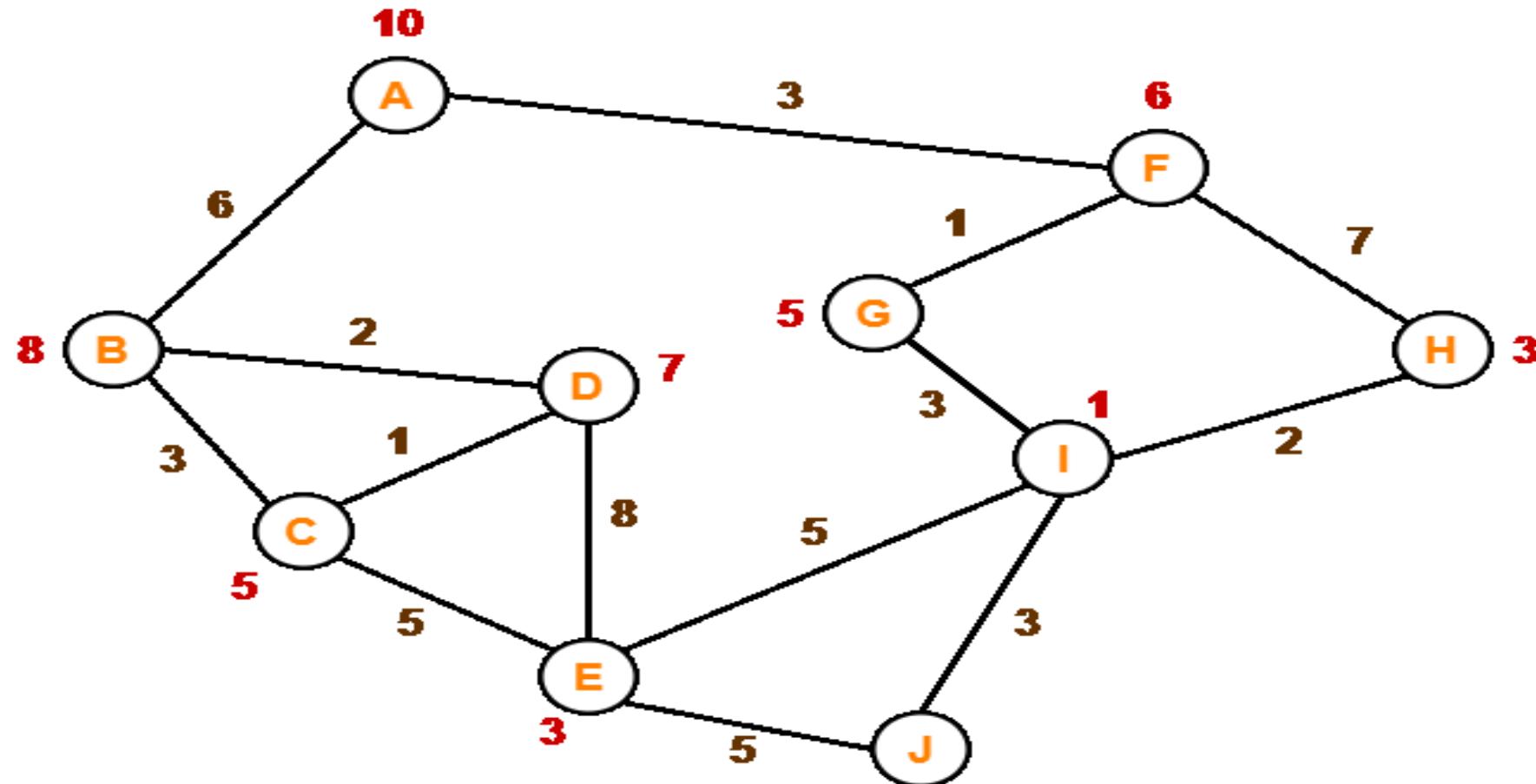
- Pitesti 417
- Timisoara 447
- Zerind 449
- Bucharest 450
- Craiova 526
- Sibiu 553
- Sibiu 591
- Arad 646
- Oradea 671

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

A* SEARCH EXAMPLE



A* CHALLENGE



PROPERTIES OF A*

- **Complete?**
- Yes (unless there are infinitely many nodes with $f \leq f(G)$)
- **Time?**
- Exponential
- **Space?**
- Keeps all nodes in memory
- **Optimal?**
- Yes (depending upon search algo and heuristic property)

PROPERTIES OF HEURISTIC FUNCTION

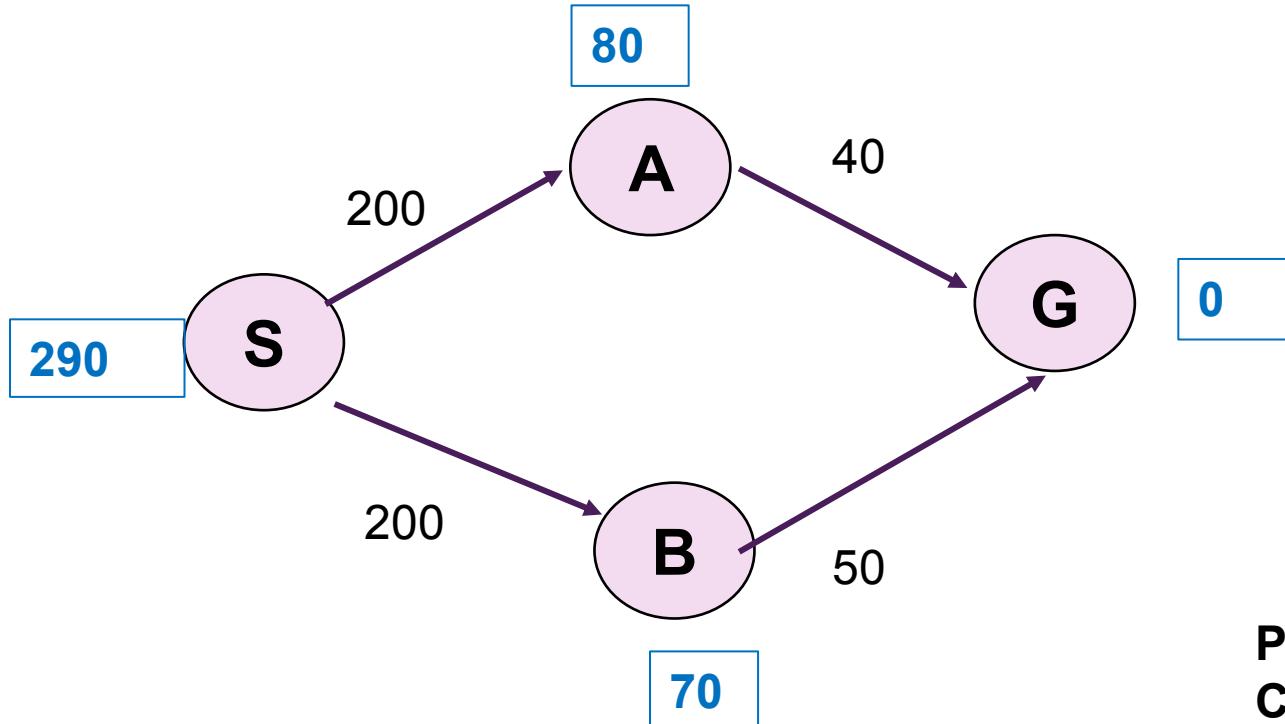
1. OPTIMALITY

"Is it guaranteed to find the shortest path?"

- The tree search version of A* is **optimal** if $h(n)$ is **admissible**, while graph search version is optimal if $h(n)$ is **consistent**.
- If h never overestimates the real distance then h is admissible, A* is guaranteed to find the **optimal solution**. An admissible heuristic is optimistic
- Example admissible heuristic: straight-line distance.
- A* with an admissible heuristic is **optimal** for a tree search, meaning when there is only one path to any given state.
- A search algorithm is **admissible** if it is guaranteed to find a minimal path to a solution whenever such a solution exists. **Breadth first search** is admissible, because it looks at every state at level n before considering any state at level $n+1$.
- A* is **complete** if the search graph is finite or when there is no infinite path in the graph having a finite total cost.

ADMISSIBILITY EXAMPLE

Overestimation : will it guarantee optimality?

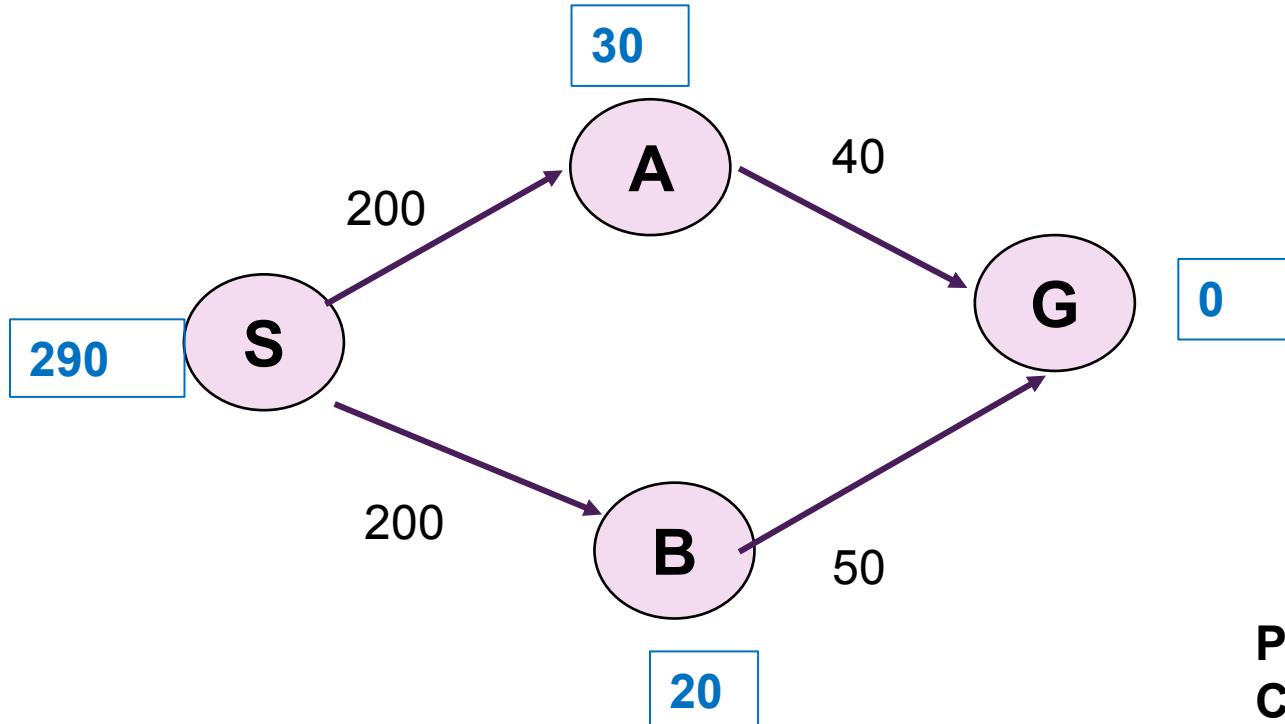


Expanded Node	Fringe Queue
--	S ²⁹⁰
S ²⁹⁰	A ²⁸⁰ B ²⁷⁰
B ²⁷⁰	G ²⁵⁰ A ²⁸⁰
G ²⁵⁰	A ²⁸⁰

Path: S-B-G
Cost: 250
Optimal: NO

ADMISSIBILITY EXAMPLE

Underestimation : will it guarantee optimality?

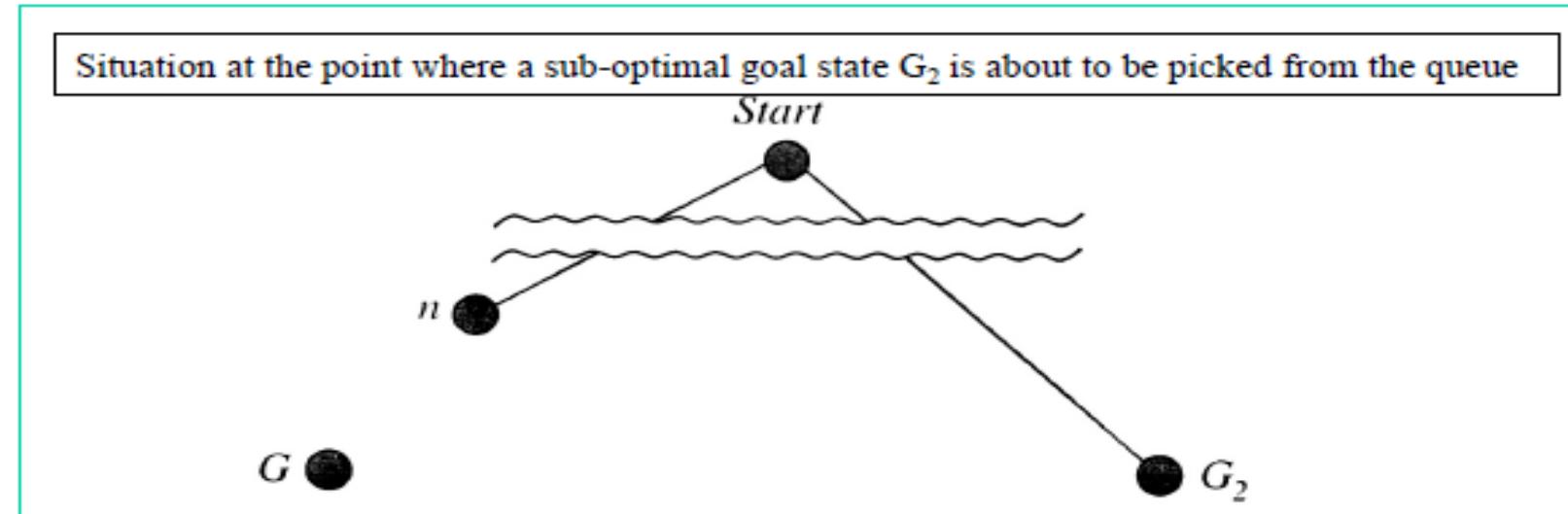


Expanded Node	Fringe Queue
--	S ²⁹⁰
S ²⁹⁰	A ²³⁰ B ²²⁰
B ²²⁰	A ²³⁰ G ²⁵⁰
A ²³⁰	G ²⁴⁰ G ²⁵⁰
G ²⁴⁰	

Path: S-A-G
Cost: 240
Optimal: YES

PROOF OF OPTIMALITY IN A*

- Let G be an optimal goal state, and $f(G) = f^* = g(G)$. Let G_2 be a suboptimal goal state, i.e. $f(G_2) = g(G_2) > f^*$. Suppose for contradiction that A^* has selected G_2 from the queue. (This would terminate A^* with a suboptimal solution) Let n be a node that is currently a leaf node on an optimal path to G .



- Because h is admissible, $f^* \geq f(n)$. If n is not chosen for expansion over G_2 , we must have $f(n) \geq f(G_2)$. So, $f^* \geq f(G_2)$. Because $h(G_2)=0$, we have $f^* \geq g(G_2)$, contradiction

2. INFORMEDNESS

"Is it better than another heuristic?"

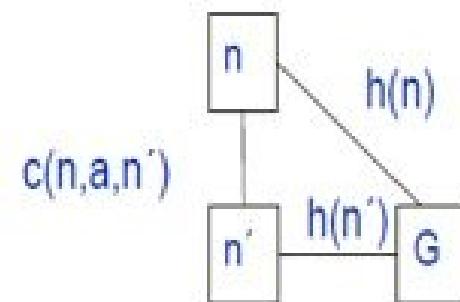
- **Informedness** A search strategy which searches less of the state space in order to find a goal state is more informed.
- Ideally, a search strategy which is both admissible (so it will find us an optimal path to the goal state), and informed (so it will find the optimal path quickly.) is preferred.
- Formally, for two admissible heuristics h_1 and h_2 , if $h_1(n) \leq h_2(n)$ for all states n in the state-space, then heuristic h_2 is said to be more informed than h_1 .
- **Dominance:** If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 dominates h_1 .
- h_2 is better for search: it is guaranteed to expand less or equal nr of nodes.

3. MONOTONICITY/CONSISTENCY

"Does it make steady progress toward a goal?"

- Monotonicity ensures consistency in finding minimal path to each state encountered in the search.
- **If along any path in the search tree of A* the f-cost never decreases, then the heuristic is said to be monotonic.**
- $h(n)$ is monotonic if, for every node n and every successor n' of n generated by any action a , the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n' : $h(n) \leq c(n,a,n') + h(n')$. This is known as the triangular inequality.

for example if $g(n) = 3$ and $h(n) = 4$. Then $f(n) = g(n) + h(n)$ - that is, we know that the true cost of a solution path through n is at least 7. Suppose also that $g(n') = 4$ and $h(n') = 2$, so that $f(n') = 6$. Clearly, this is an example of a nonmonotonic heuristic as cost should be at least 7.



3. MONOTONICITY

- f is *monotonously non-decreasing if* a heuristic function h satisfies the **monotonicity restriction**: Let n' be a descendant of n and if $h(n) - h(n') \leq \text{cost}(n \dots n')$ and the heuristic evaluation of the goal state is 0: $h(\text{Goal}) = 0$.
- This property asks if an algorithm is **locally admissible**---that is, it always **underestimates the cost between any two states in the search space**

Consequences of Non-Monotonous Heuristic function

- Monotonous Heuristic deletes later-created duplicates
- **A search method that reaches a given node at different depths in the search tree is not monotone.**
- A* with a consistent heuristic is optimal for any kind of search (graph search).
- When a state is discovered by using heuristic search, is there any guarantee that the same state won't be found later in the search at a cheaper cost (with a shorter path from the start state)? This is the property of monotonicity.

ITERATIVE IMPROVE MENT ALG ORITHMS



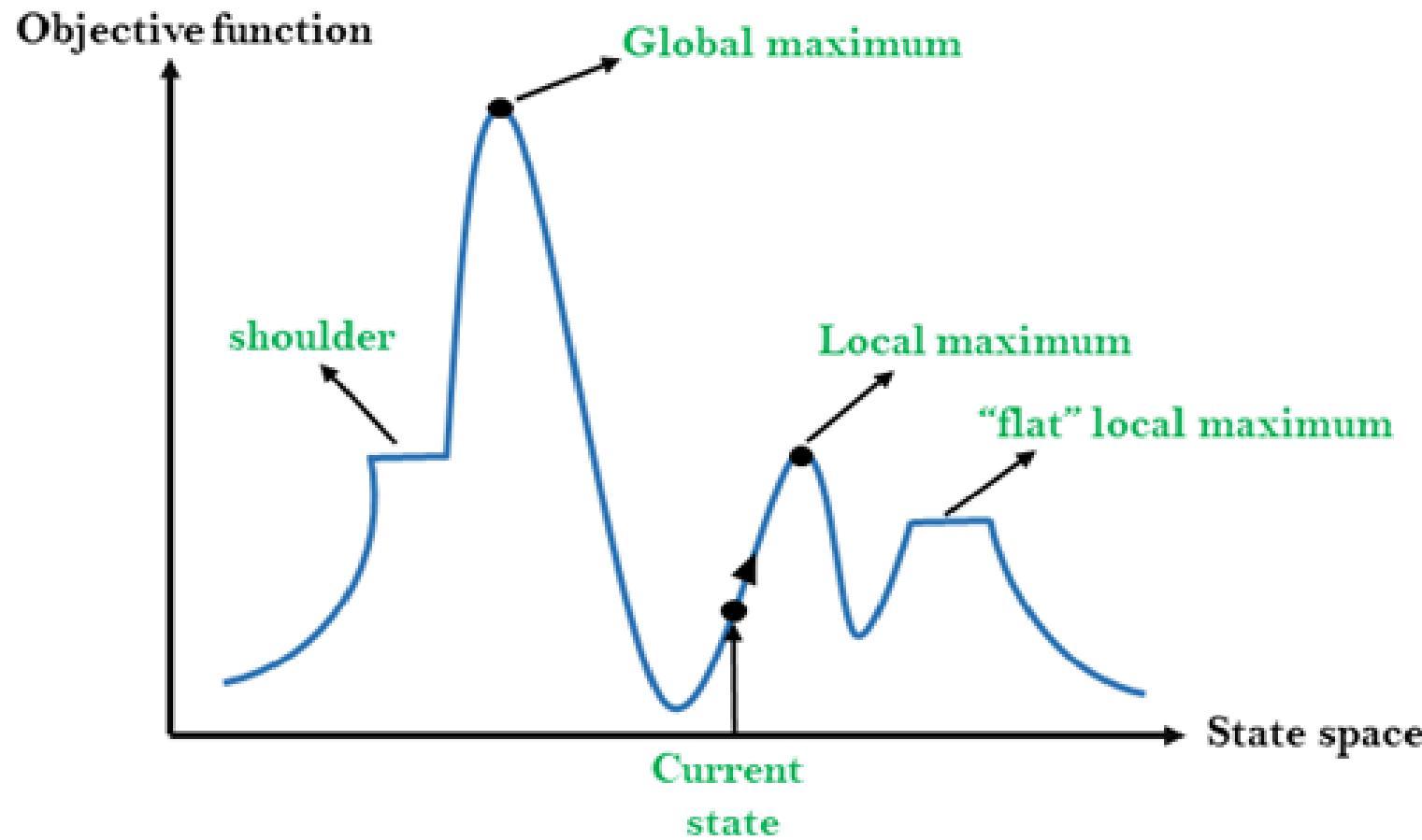
ITERATIVE IMPROVEMENT ALGORITHMS

- Iterative improvement algorithms provide the most practical approach for problems in which **all the information needed for a solution are contained in the state description itself** but performs purely a local search in state space, evaluating and modifying one or more current states rather than systematically exploring paths from initial state.
- The general idea is to start with a complete configuration and to make modifications to improve its quality.
- The idea of iterative improvement is to move around the landscape trying to find the highest peaks, which are the optimal solutions.
- The height of any point on the landscape corresponds to the evaluation function of the state at that point.
- Iterative improvement algorithms usually keep track of only the current state, and do not look ahead beyond the immediate neighbors of that state so can save some memory space
- The iterative improvement algorithms divide into two major classes. The first one is called the **Hill-climbing search**, also called **gradient descent**. The second one is called the **Simulated annealing**.
- Suitable for problem for which solution state matters rather than path cost to reach it.

LOCAL SEARCH ALGORITHMS & OPTIMIZATION

- Local search algorithm searches in the state space, **evaluates and modifies one or more current states rather than systematically exploring paths from an initial state.**
- Suitable for problems in which all that **matters is the solution state**, not the path cost to reach it.
- **Eg:** 8-Queens problem, integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.
- Local search algorithms operate on a single current node (rather than multiple paths) and generally move only to neighbors of that node. **Paths followed by the search are not retained.**
- **Advantages:**
 1. Use very **little memory**—usually a constant amount;
 2. Often find **reasonable solutions** in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.
 3. Useful for **solving pure optimization problems**, which the aim to find the best state according to an objective function.

DIFFERENT REGIONS IN THE STATE SPACE LANDSCAPE



Local Maximum: a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum.

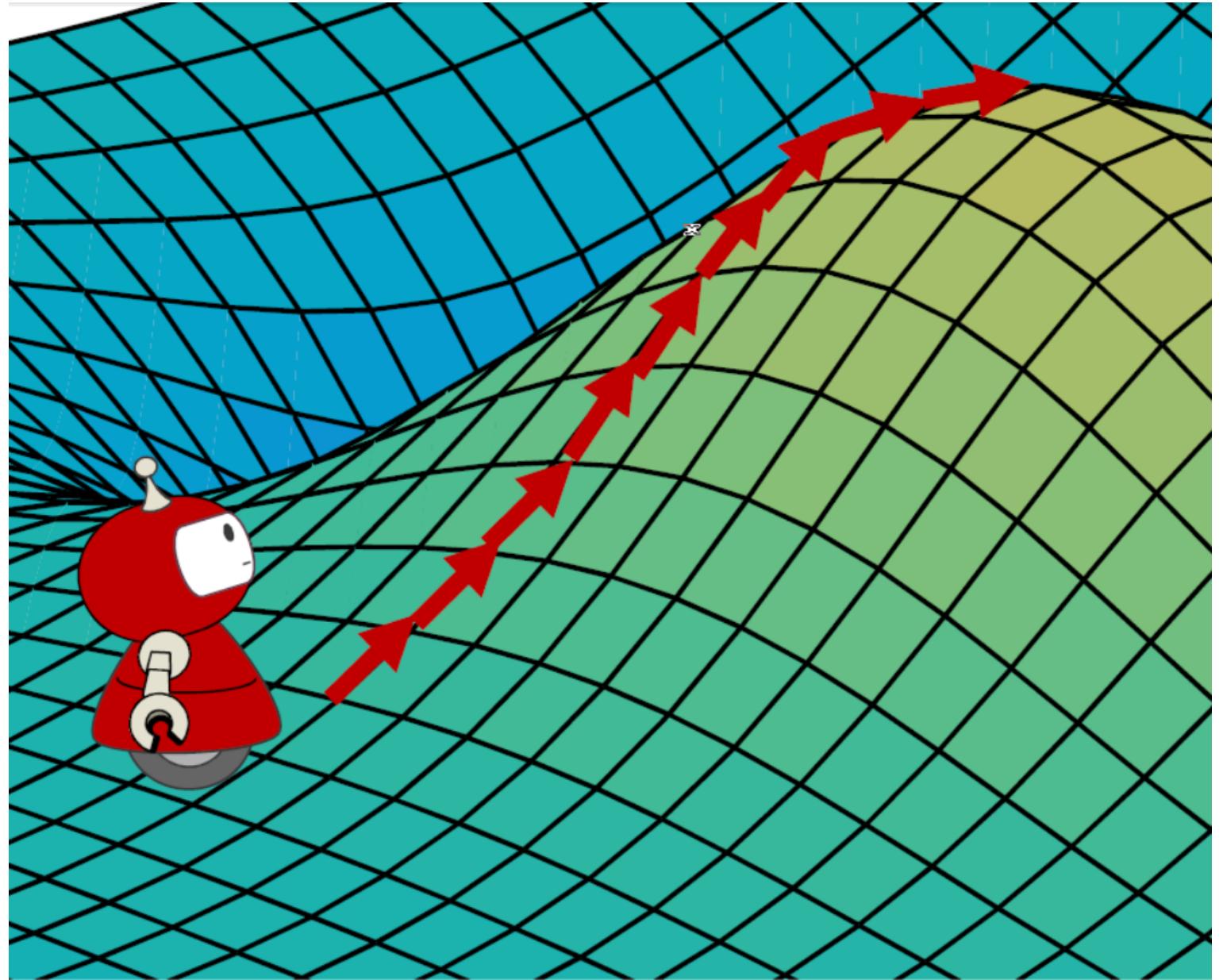
- **Global Maximum:** Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
- **Current state:** It is a state in a landscape diagram where an agent is currently present.
- **Flat local maximum:** It is a flat space in the landscape where all the neighbor states of current states have the same value.
- **Shoulder:** It is a plateau region which has an uphill edge.

TRIVIAL ALGORITHMS

- **Random Sampling**
 - Generate a state randomly
- **Random Walk**
 - Randomly pick a neighbor of the current state
- Both algorithms asymptotically complete.
- Local search can do quite well on optimization problems.

HILL CLIMBING SEARCH

"Like climbing
everest in
thick fog with
amnesia"



HILL CLIMBING SEARCH

- **Analog**

- Go uphill along the steepest possible path until no farther up
- A **greedy local search algorithm**
- It is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where **no neighbor has a higher value**

- **Principle**

- Expand **the current state** of the search and **evaluate** its children
- Select the **best child**, ignore its **siblings** and **parent**
- **No history for backtracking**
- The basic idea is to **proceed** according to some **heuristic measurement of the remaining distance to the goal.**
- attempts to find a better solution by incrementally changing a



HILL CLIMBING SEARCH

- The *Hill-climbing search algorithm* is a loop that continually moves in the direction of increasing value. The algorithm **only records the state and its evaluation instead of maintaining a search tree**. It takes a *problem* as an input, and it keeps comparing the values of the *current* and the *next* nodes. The **next node** is the **highest-valued successor** of the **current node**. If the value of the *current* node is greater than the next node, then the *current* node will be returned. Otherwise, it will go deeper to look at the *next* node of the *next* node.
- **Hill climbing** is a mathematical optimization technique which belongs to the family of **local search**. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by making an incremental change to the solution. If the change produces a better solution, another incremental change is made to the new solution, and so on until no further improvements can be found.
- It is also called **greedy local search** as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are **state** and **value**.
- Hill Climbing is mostly used **when a good heuristic is available**.

HILL CLIMBING ALGORITHM

1. Looks one step ahead to determine if any successor is better than the current state; then move to the best successor.
2. If there exists a **successor s** for the **current state n** such that
 1. $h(s) < h(n)$
 2. $h(s) \leq h(t)$ for all the successors t of n,
3. Then **move from n to s**. Otherwise, **halt at n**.
4. Similar to Greedy search in that it uses h but **does not allow backtracking or jumping to an alternative path since it doesn't "remember" where it has been**.
5. Not complete since the search **will terminate at "local minima," "plateaus," and "ridges."**

FEATURES OF HILL CLIMBING

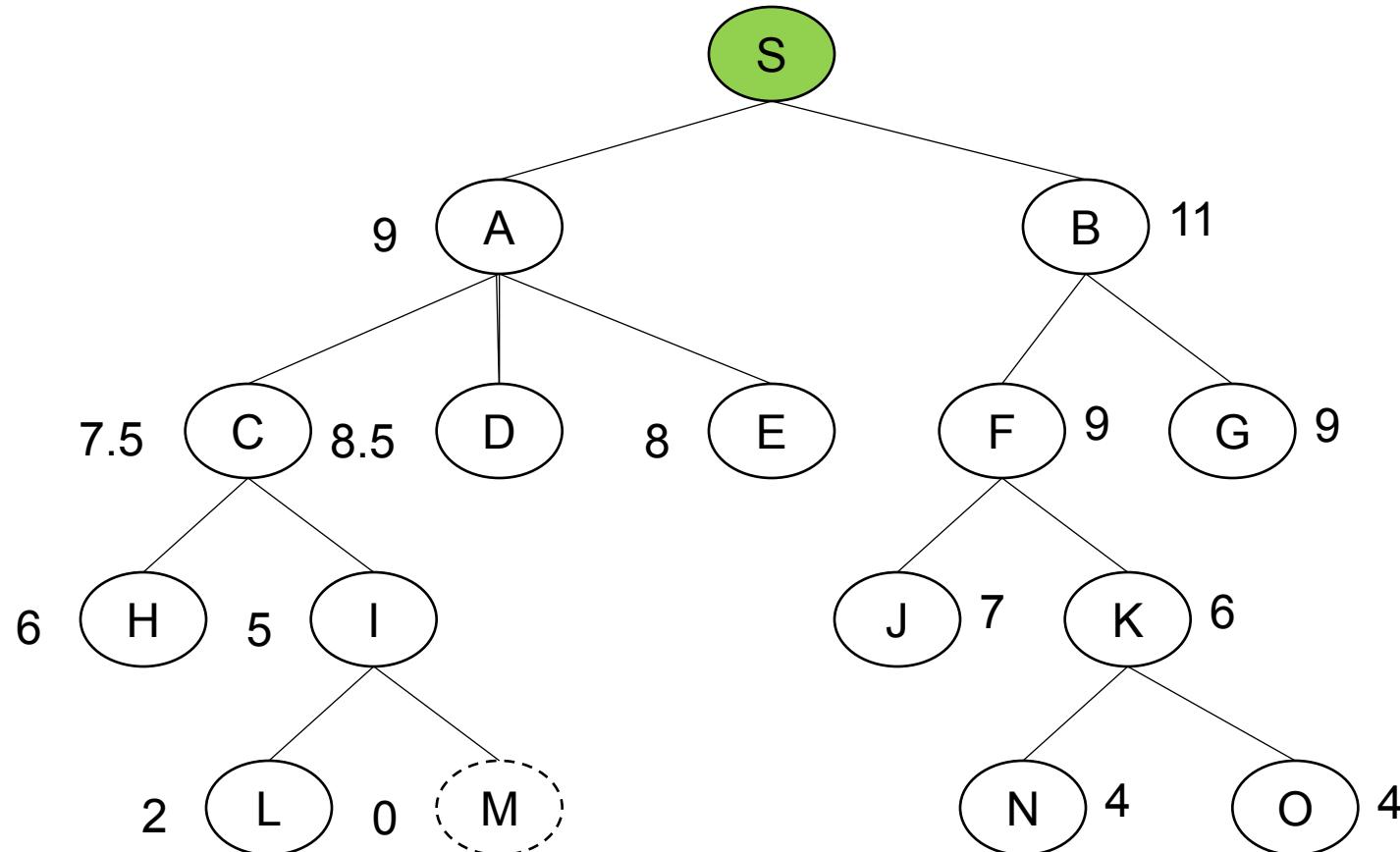
- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

STATE SPACE DIAGRAM FOR HILL CLIMBING

- The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.
- On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.

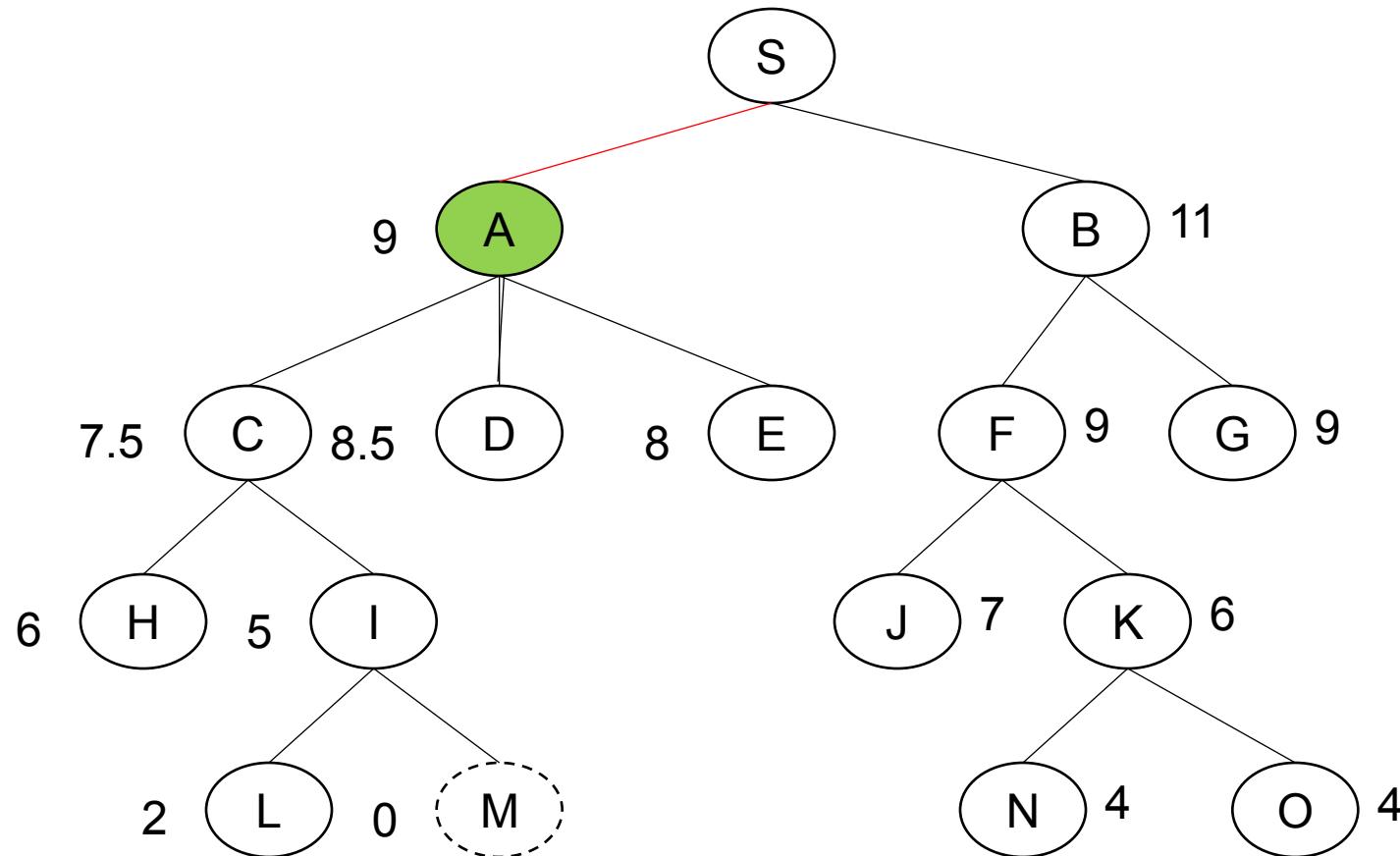
Hill-climbing search example

our aim is to find a path from **S** to **M**

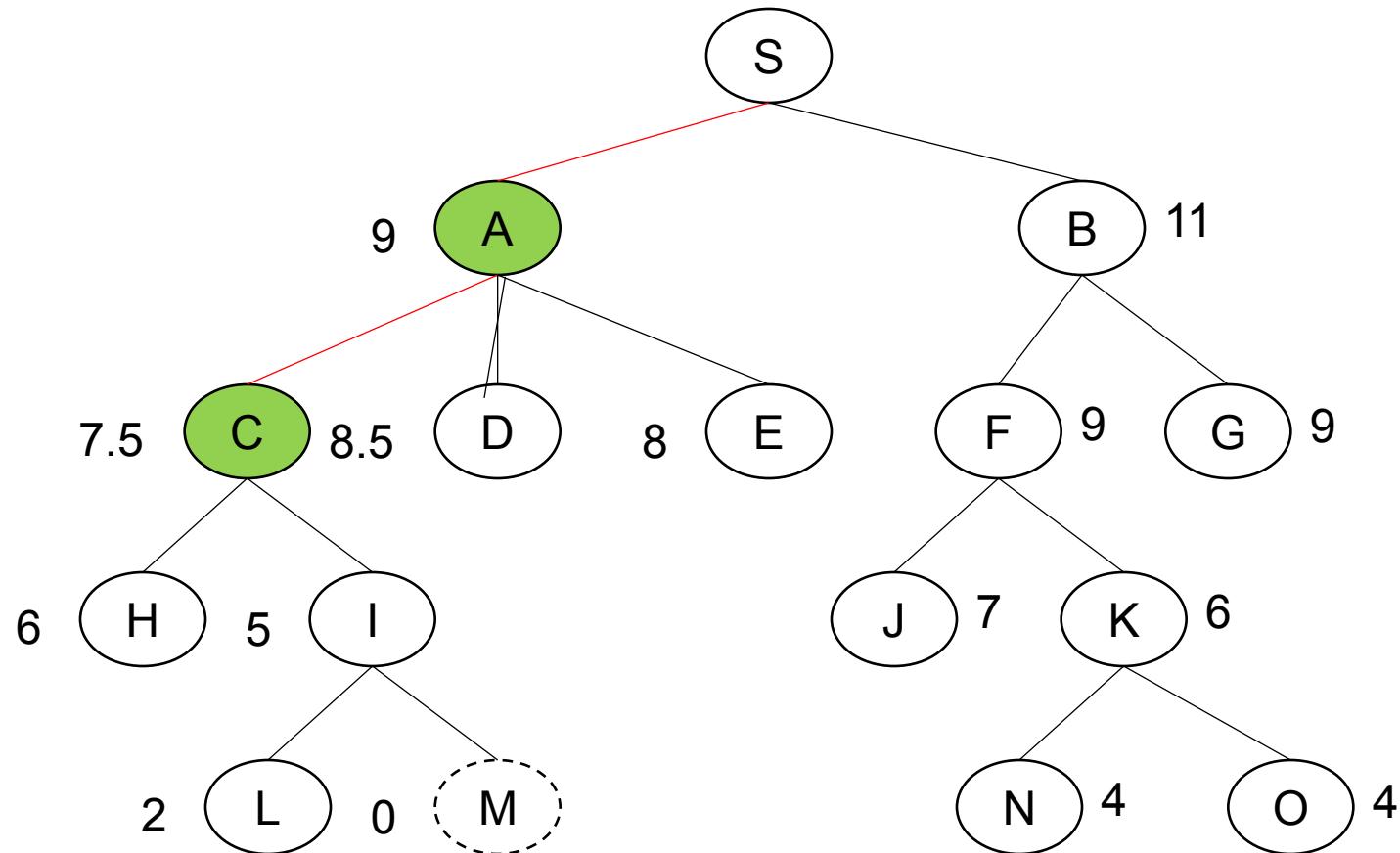


associate heuristics with every node, that is the straight line distance from the path terminating city to the goal city

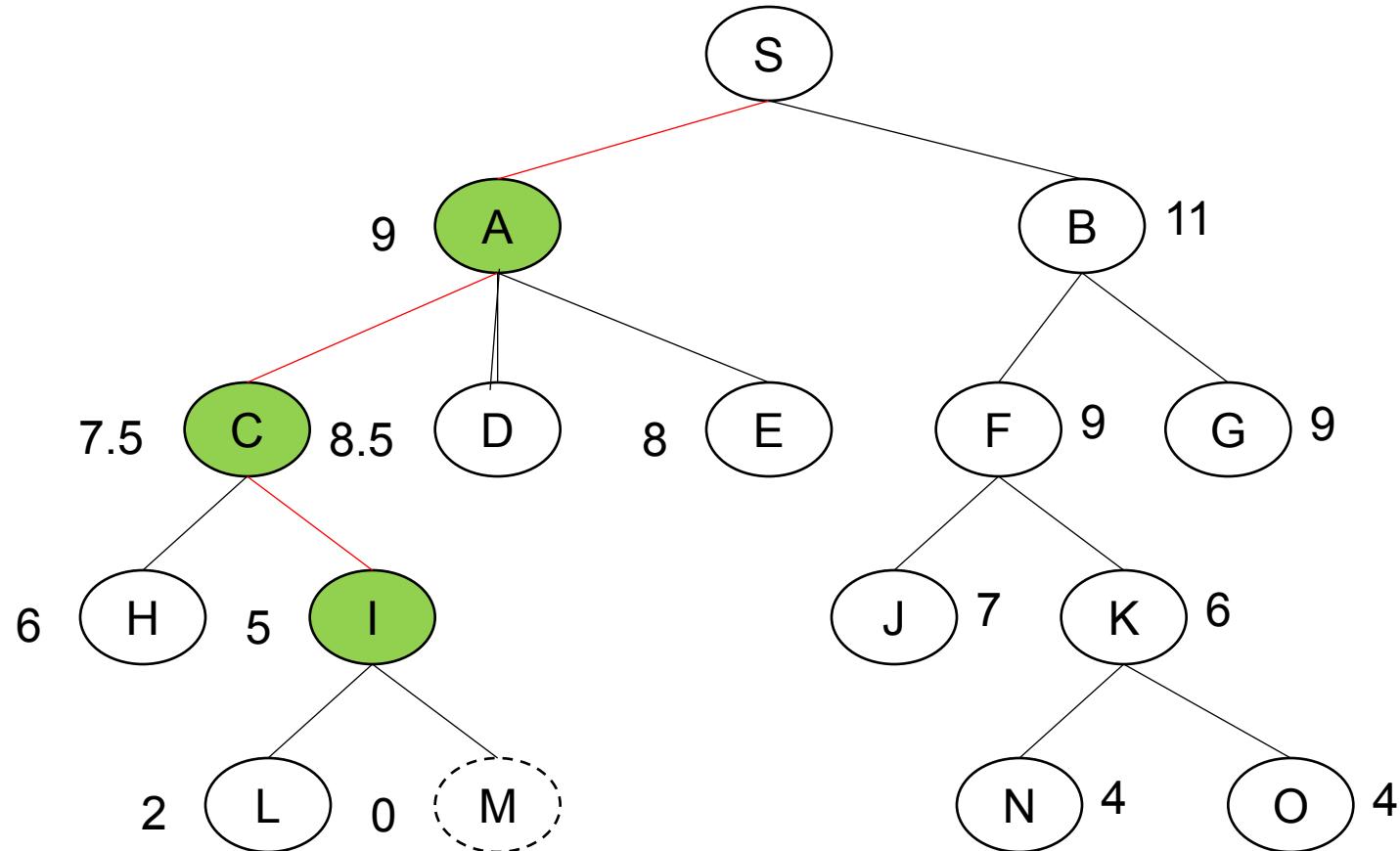
Hill-climbing search example



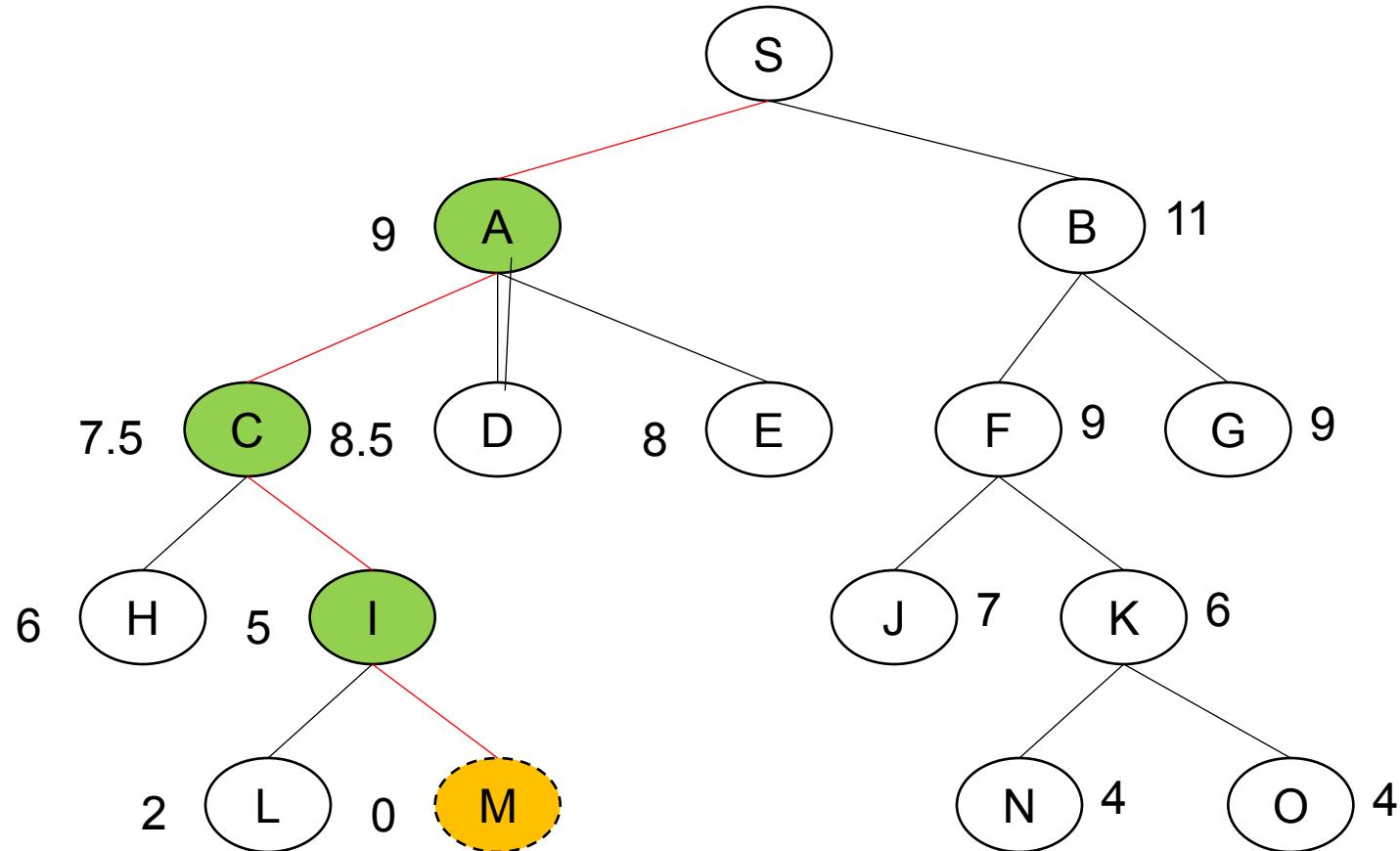
Hill-climbing search example



Hill-climbing search example

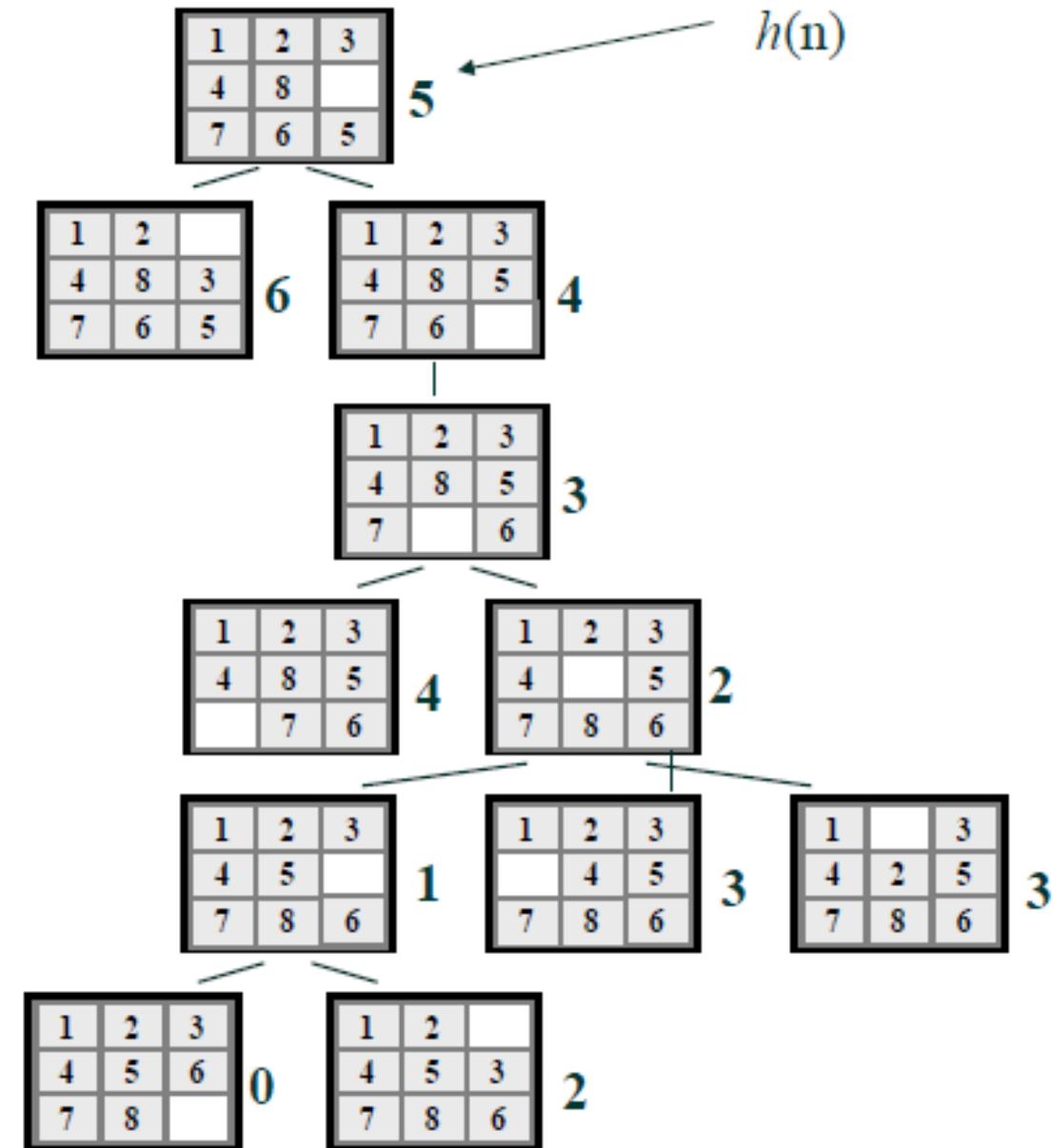


Hill-climbing search example



HILL CLIMBING EXAMPLE

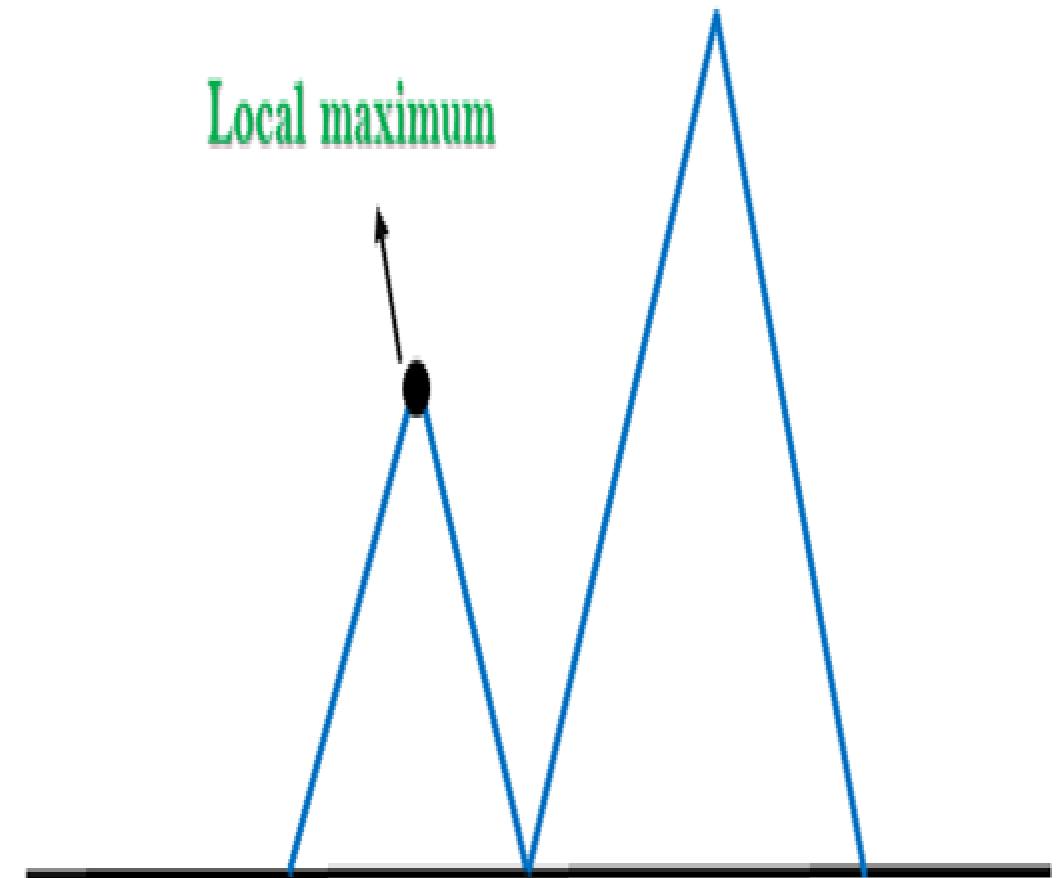
MANHATTAN DISTANCE
IS USED AS
HEURISTIC



PROBLEMS IN HILL CLIMBING

1. LOCAL MAXIMUM

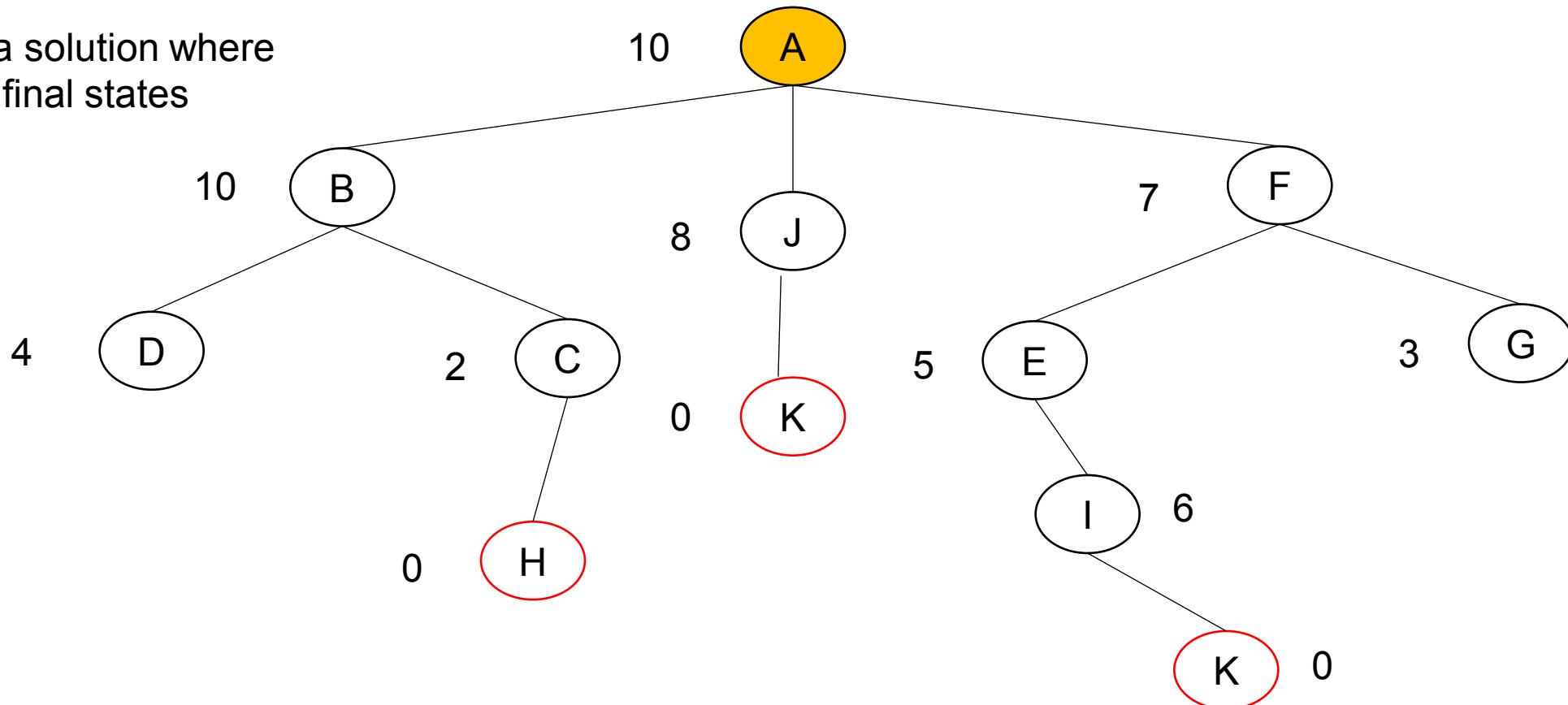
- A local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum..
- **PROBLEM:** When a local maxima is reached, the algorithm will stuck with no where else to go even a solution has not been reached yet.
- **Solution:** Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.



Hill-climbing search example

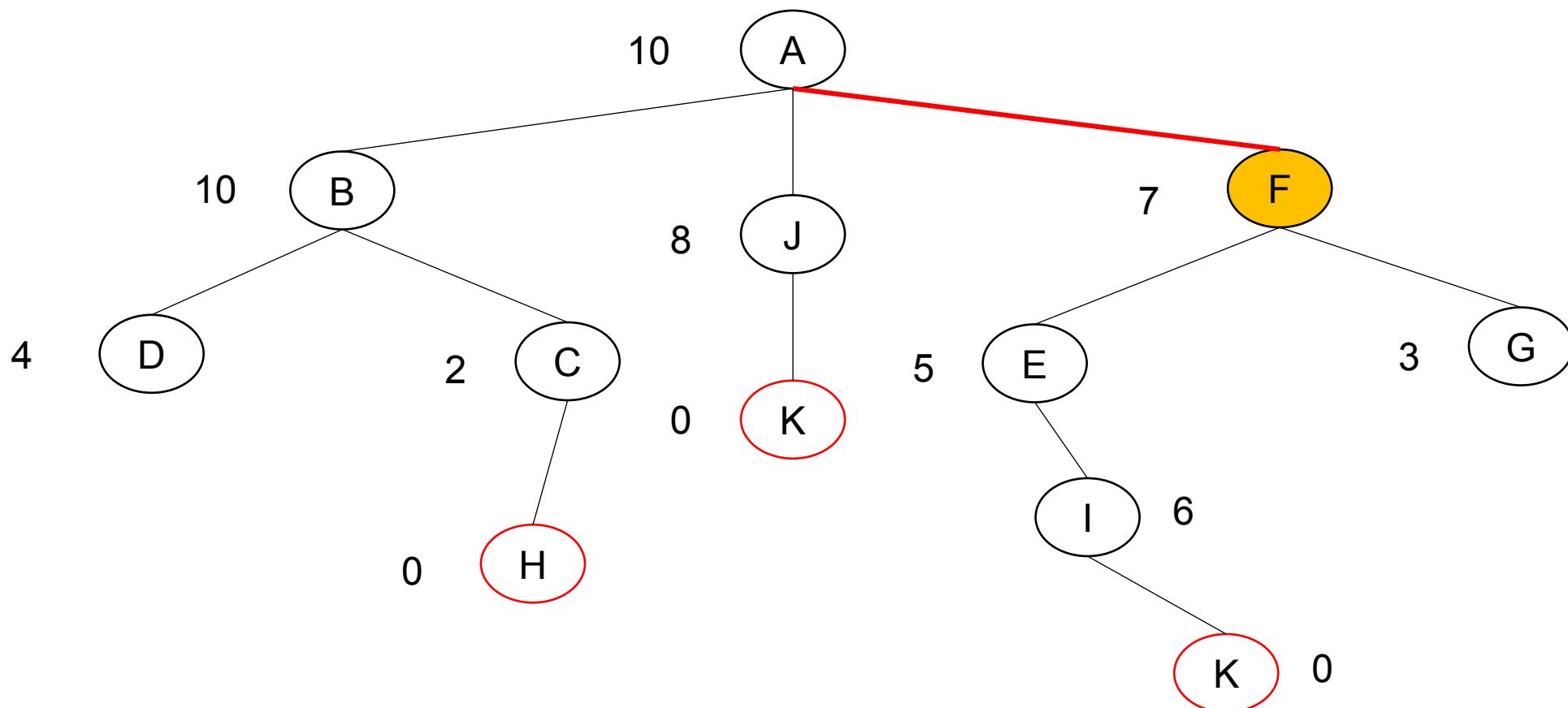
Local maximum

From A find a solution where
H and K are final states



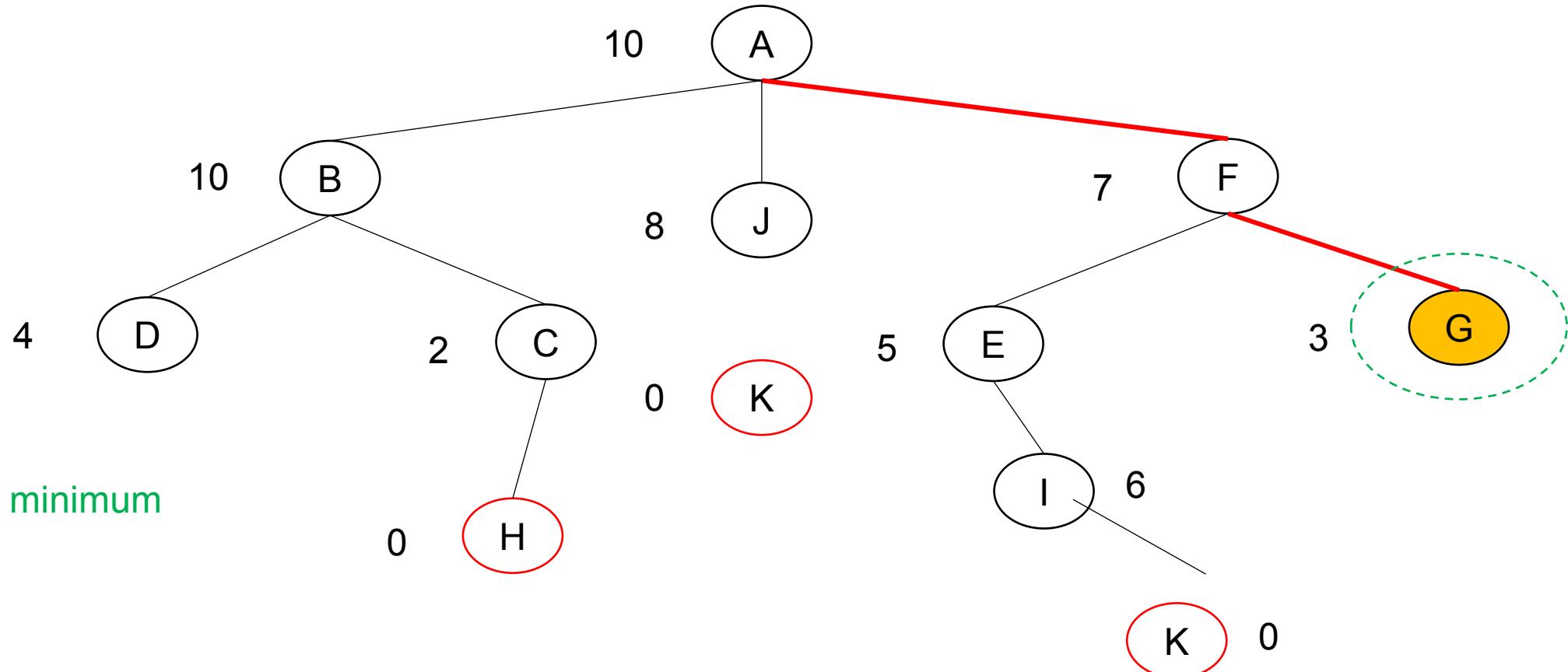
Hill-climbing search example

Local maximum



Hill-climbing search example

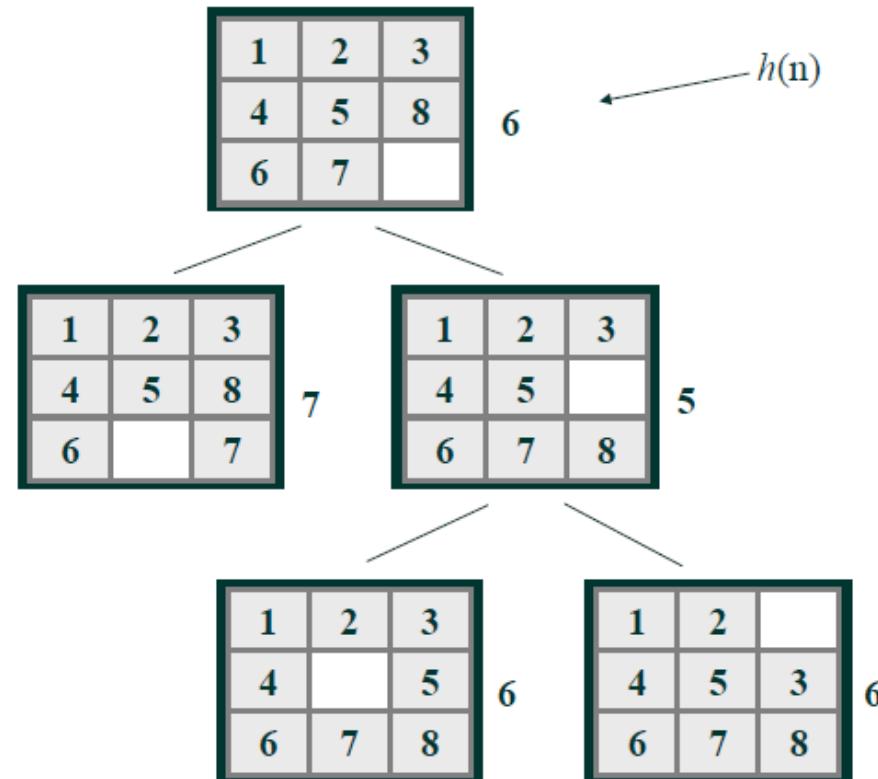
Local minimum



Hill climbing is sometimes called **greedy local search** because it grabs a good neighbor state without thinking ahead about where to go next.

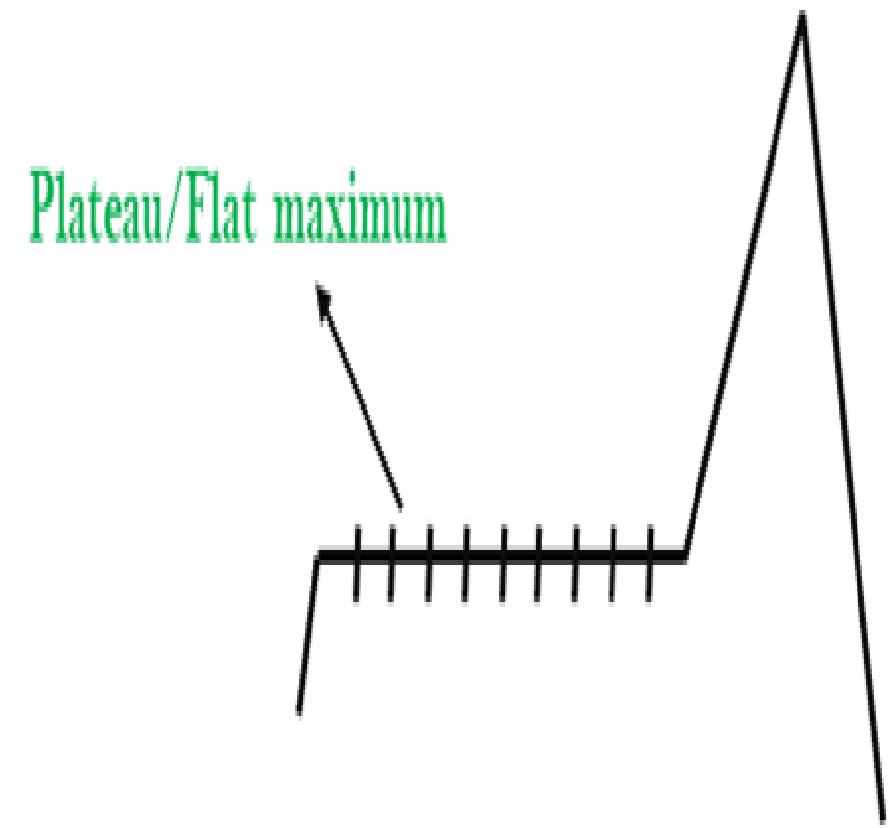
PROBLEMS OF HILL CLIMBING

All the nodes on the fringe
are taking a step
“backwards”
(local minima)



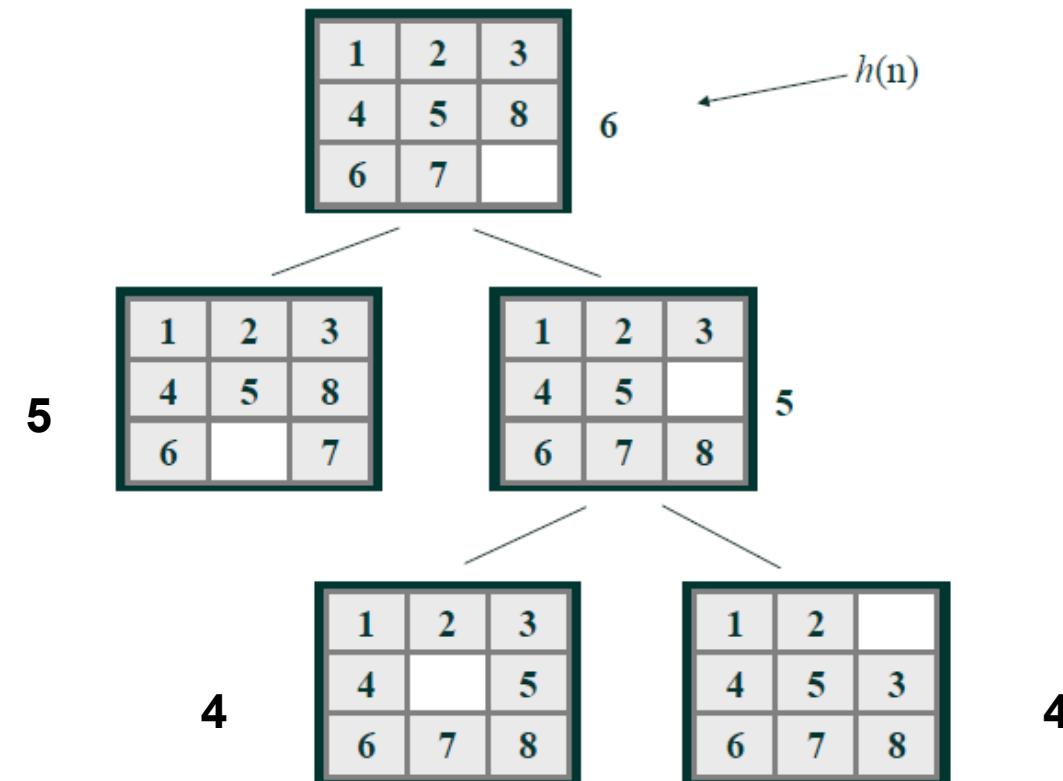
2. PLATEAU

- A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area. : a plateau is an area of the state space landscape where the evaluation function is flat.
- On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.
- A plateau is an area of the state space where the neighbors are about the same height. In such a situation, a random walk will be generated.
- **Solution:** The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-



PROBLEMS OF HILL CLIMBING

All the nodes on the fringe are having same heuristic value 4 (plateau)

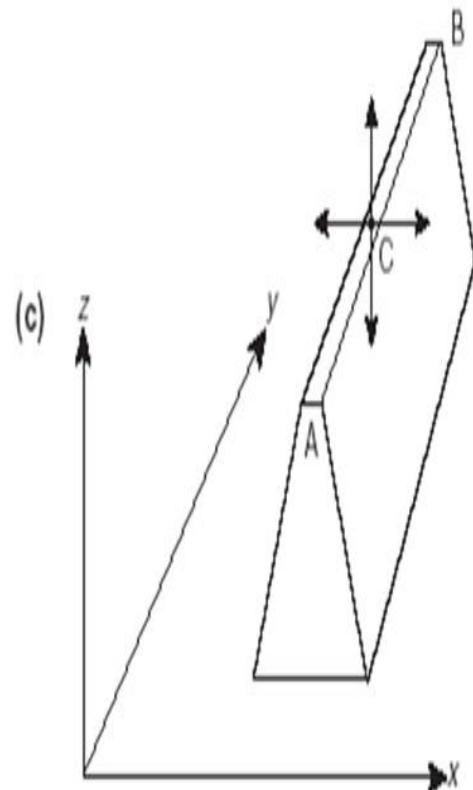


3. RIDGES

- Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate. The search direction is not towards the top but towards the side
- A ridge may have steeply sloping sides towards the top, but the top only slopes gently towards a peak. In this case, the search makes little progress unless the top is directly reached, because it has to go back and forth from side to side.
- A ridge is a special kind of local maximum. It is an area of the search space that is higher than the surrounding areas and that itself has a slope. But the orientation of the high region, compared to the set of available moves and the directions in which they move, makes it impossible to traverse a ridge by single moves. Any point on a ridge can look like a peak because movement in all probe directions is downward.
- **Solution:** With the use of bidirectional search, or by moving in different directions, we can improve this problem.

■ B is higher than A.

■ At C, the hill-climber can't find a higher point North, South, East or West, so it stops.



HOW HILL CLIMBING RECOVERS?

- If algorithm encounters a situation that there is **more than one best successor to choose from**, it will randomly select a certain one.
- If algorithm reaches a point at which no progress is being made further from a certain starting point, algorithm start again from a different starting point. It is known as **Random-restart hill-climbing**
- Conducting a series of hill-climbing searches from randomly generated initial states, running each until it halts or makes no discernible progress. It saves the best result found so far from any of the searches. It can use a fixed number of iterations, or can continue until the best saved result has not been improved for a certain number of iterations.
- As a matter of fact, and obviously, the fewer local maxima, the quicker it finds a good solution. And it can eventually find out the optimal solution if enough iterations are allowed. But usually, a reasonably good solution can be found after a small number of iterations.

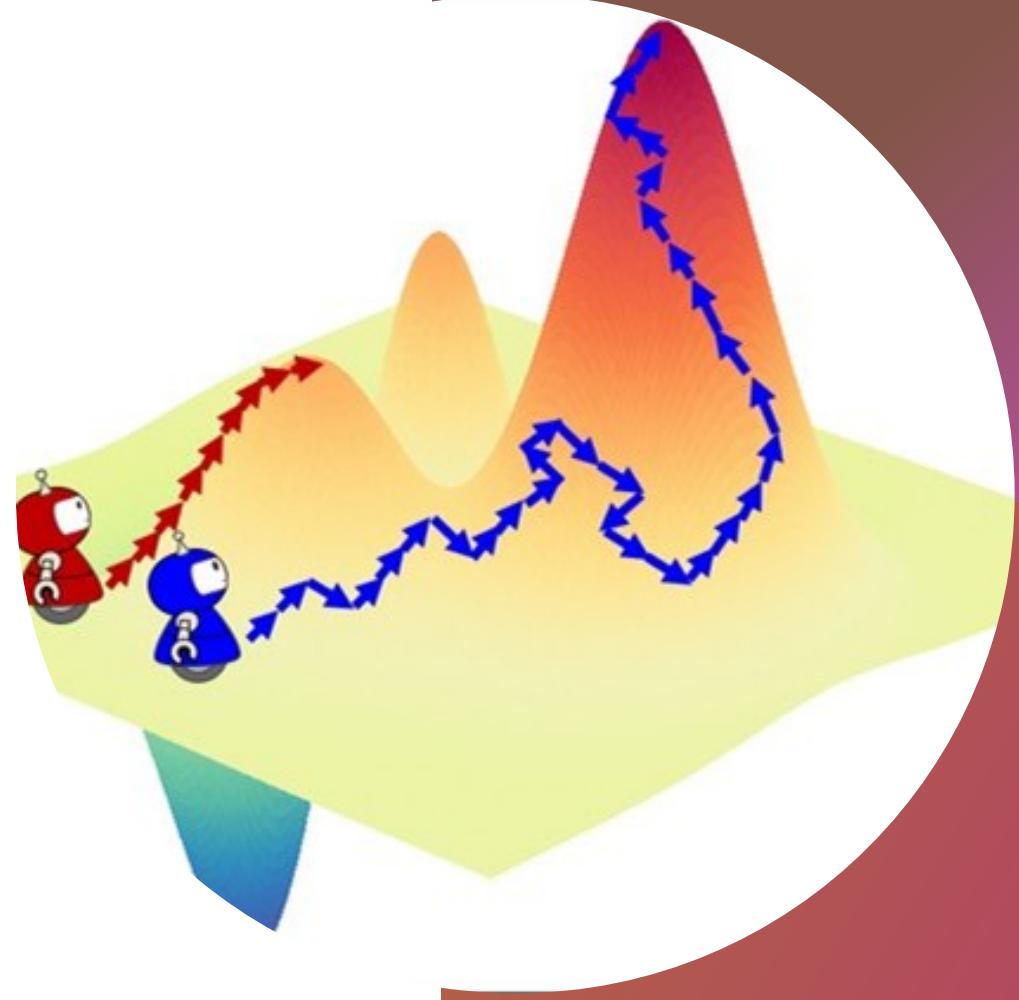
VARIANTS OF HILL CLIMBING ALGORITHMS

1. **Stochastic hill Climbing:** It does not examine all the neighboring nodes before deciding which node to select .It just **selects a neighboring node at random** and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.
2. **Simple hill Climbing:** It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as next node. | It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state.
3. **Steepest-Ascent hill-climbing:** It first examines all the neighboring nodes and then selects the node closest to the solution state as of next node. This algorithm consumes more time as it searches for multiple neighbors
4. **First Choice Hill Climbing:** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.
5. **Random-restart Hill Climbing:** It conducts a series of hill-climbing searches from randomly generated initial states,1 until a goal is found. It is trivially complete with probability approaching 1, because it will eventually generate a goal state as the initial state. If each hill-climbing search has a probability p of success, then the expected number of restarts required is $1/p$.

APPLICATIONS

- Hill Climbing technique can be used to solve many problems, where the current state allows for an accurate evaluation function, such as Network-Flow, Travelling Salesman problem, 8-Queens problem, Integrated Circuit design, etc.
- Hill Climbing is used in inductive learning methods too. This technique is used in robotics for coordination among multiple robots in a team. There are many other problems where this technique is used.

SIMULATED ANNEALING



SIMULATED ANNEALING

- **Hill Climbing Major drawbacks**
- A **hill-climbing algorithm** never makes “downhill” moves toward states with lower value (or higher cost) turns out to be **incomplete**, since it get stuck on a local maximum. In contrast, **a purely random walk**, moving to a successor chosen uniformly at random from the set of successors is **complete** but **extremely inefficient**.
- **Solution:**
- **Simulated Annealing:** Combine hill climbing with a random walk in some way that yields both efficiency and completeness
- **Analog**
- Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1.

SIMULATED ANNEALING

- Motivated by the physical annealing process used to **temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low energy crystalline state.**
- The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.
- When the end the searching is close, it starts behaving like *hill-climbing*. At the start, make lots of moves and then gradually slow down
- More formally...
 - Instead of picking the best move (as in Hill Climbing), **Generate a random new neighbor from current state, If it's better take it,**
 - **If it's worse then take it with some probability proportional to the temperature ie the delta between the new and old states. Probability gets smaller as time passes and by the amount of "badness" of the move,**
- Compared to hill climbing the main difference is that **SA allows downwards steps**; (moves to higher cost successors).
- **SA can avoid becoming trapped at local maxima.** SA uses a random search that occasionally accepts changes that decrease objective function f. SA uses a control parameter T, which by analogy with the original application is known as the system "temperature." T starts out high and gradually decreases toward 0.

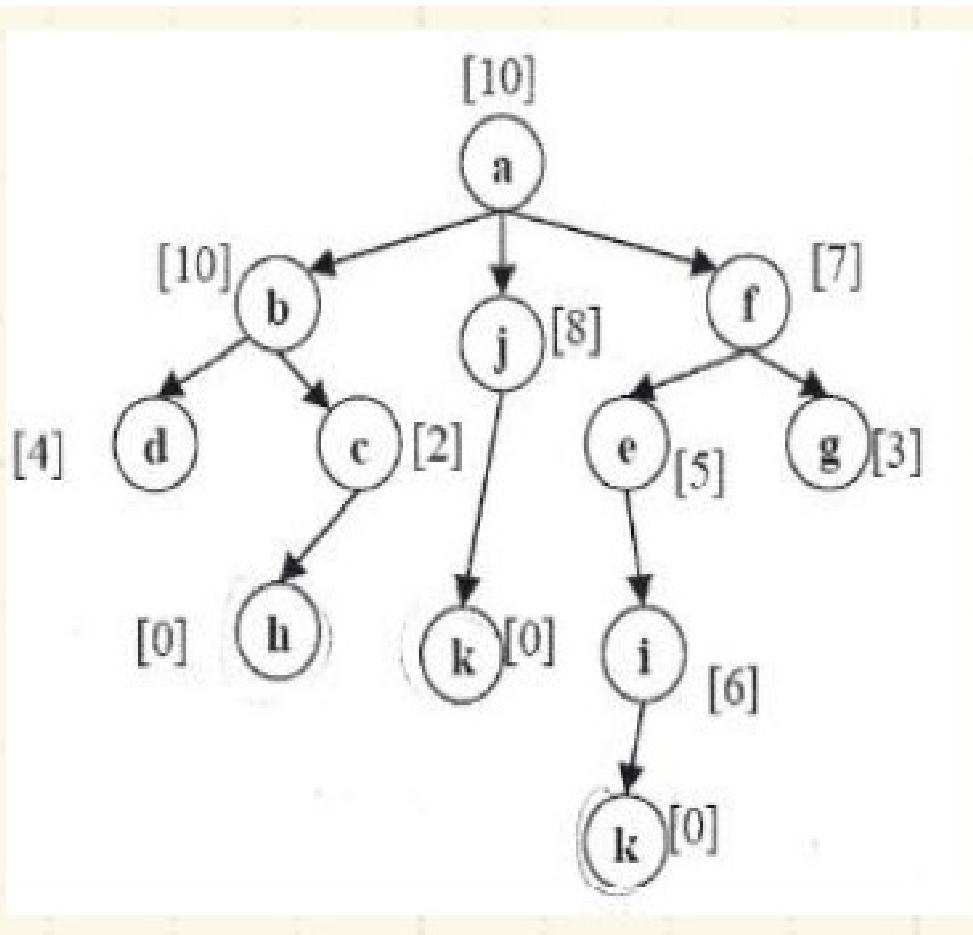
SIMULATED ANNEALING PROBABILITY FACTOR

- By checking the value of ΔE the algorithm can determine the probability of the next move. If ΔE is greater than zero, then the *next* node will be looked at. Otherwise, the probability for the *next* node to be looked at is e to the power ΔE over T .
- subtracting the values of the current node from the next node to obtain the difference **$\Delta E = f(B) - f(A)$**
- **$P(\text{move}_{A \rightarrow B}) = e^{(\Delta E) / T}$**
- ΔE is actually the amount by which the evaluation is worsened
- A local variable T which is the temperature controlling the probability of downward steps. The higher T , the more likely a bad move will be made. As T tends to zero, this probability tends to zero, and SA becomes more like hill climbing. If T is lowered slowly enough, SA is complete and admissible.
- **Simulated Annealing** is an algorithm which yields both efficiency and completeness.

SIMULATED ANNEALING STEPS

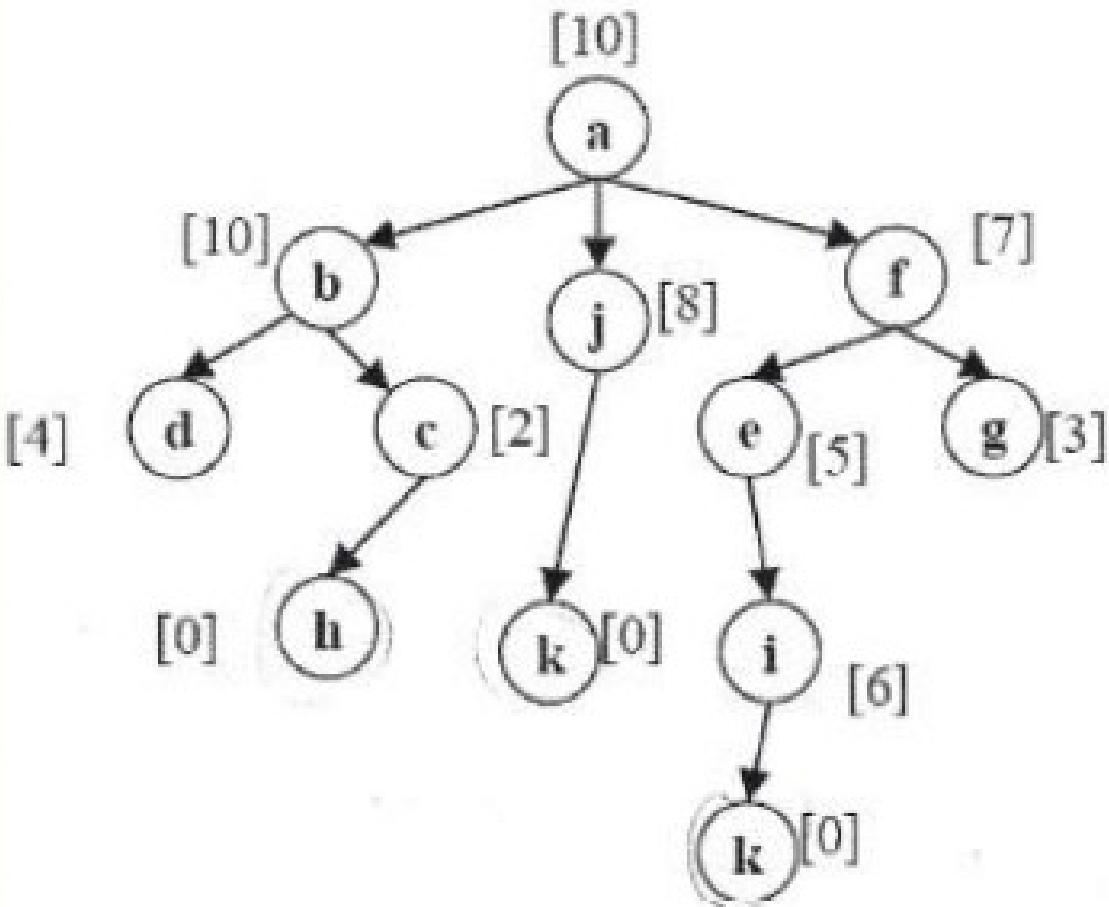
1. Select a start node(rootnode).
2. Randomly select a child of the current node, calculate a value reflecting how good such child is like $\text{valu}(\text{node}) = -\text{heuristic}(\text{node})$.
3. Select the child if it is better than the current node. Else try another child.
A node is better than the current node if $\Delta E = \text{valu}(\text{next}) - \text{valu}(\text{current}) > 0$.
Else if $\Delta E < 0$, then try to find another child.
4. If the child was not better than the current node then it will be selected with probability equal to $p = e^{\Delta E/T}$ where $\Delta E = \text{valu}(\text{next}) - \text{valu}(\text{current})$ and T is a temperature.
5. Stop if no improvement can be found or after a fixed time.

EXAMPLE



Current	Children
a	---
a	f_7, j_8, b_{10}
Randomly Select a Child	
$\Delta E > 0$	
Check if next node f_7 is better than current node	
$\Delta E = \text{value}(\text{next}) - \text{value}(\text{current})$	
$\Delta E = \text{value}(f_7) - \text{value}(a_{10})$	
$\text{value}(f_7) = -\text{heuristic}(f_7) = -7$	
$\text{value}(a_{10}) = -\text{heuristic}(a_{10}) = -10$	
$\Delta E = -7 - (-10) = +3$	
$\because \Delta E > 0$	
$\therefore f_7$ will be selected with probability 1	

EXAMPLE



Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3

Randomly Select a Child

Check if next node e_5 is better than current node

$\Delta E = \text{value(next)} - \text{value(current)}$

$\Delta E = \text{value}(e_5) - \text{value}(f_7)$

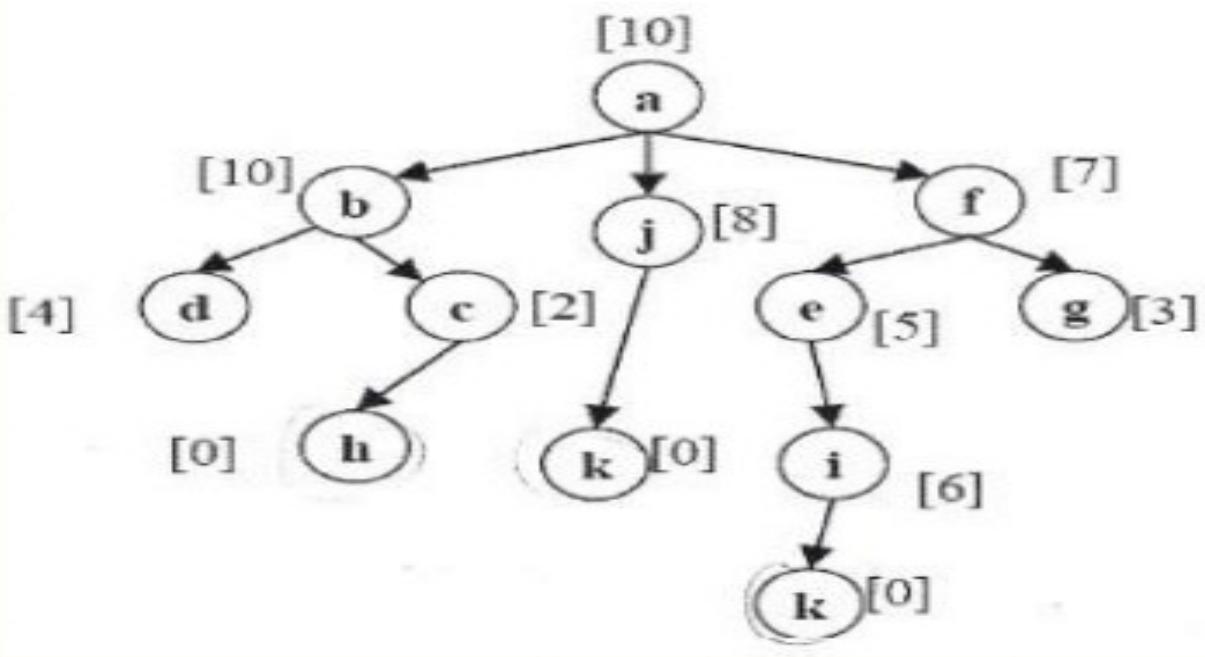
$\text{value}(e_5) = -\text{heuristic}(e_5) = -5$

$\text{value}(f_7) = -\text{heuristic}(f_7) = -7$

$\Delta E = -5 - (-7) = +2$

$\because \Delta E > 0$

$\therefore e_5$ will be selected with probability 1



$\because \Delta E < 0$
 $\therefore i_6$ can be selected with
 probability $p = e^{\frac{\Delta E}{T}}$

$$p = e^{\frac{-1}{10}} = e^{\frac{-1}{10}} = .905$$

Because the only child of e_5 is i_6
 then it will be selected even if its
 probability is not 1.

Randomly Select a Child

Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3
e	i_6

Check if next node i_6 is better than current node

$$\Delta E > 0$$

$$\Delta E = \text{value(next)} - \text{value(current)}$$

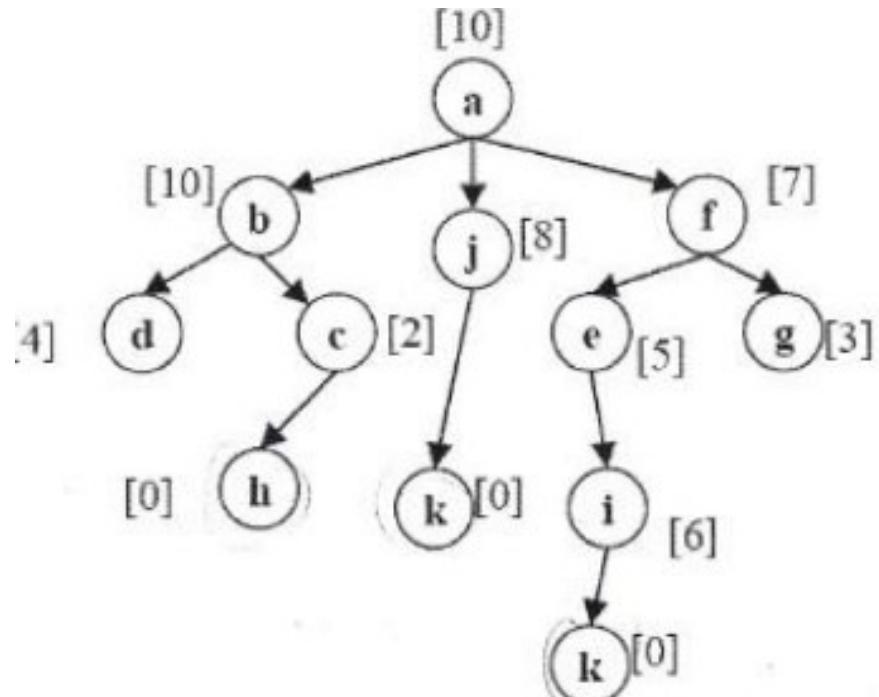
$$\Delta E = \text{value}(i_6) - \text{value}(e_5)$$

$$\text{value}(i_6) = -\text{heuristic}(i_6) = -6$$

$$\text{value}(e_5) = -\text{heuristic}(e_5) = -5$$

$$\Delta E = -6 - (-5) = -1$$

EXAMPLE



Randomly Select a Child

Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3
e	i_6
i	k_0

Check if next node k_0 is better than current node

$$\Delta E > 0$$

$$\Delta E = \text{value}(k_0) - \text{value}(i_6)$$

$$\text{value}(k_0) = -\text{heuristic}(k_0) = 0$$

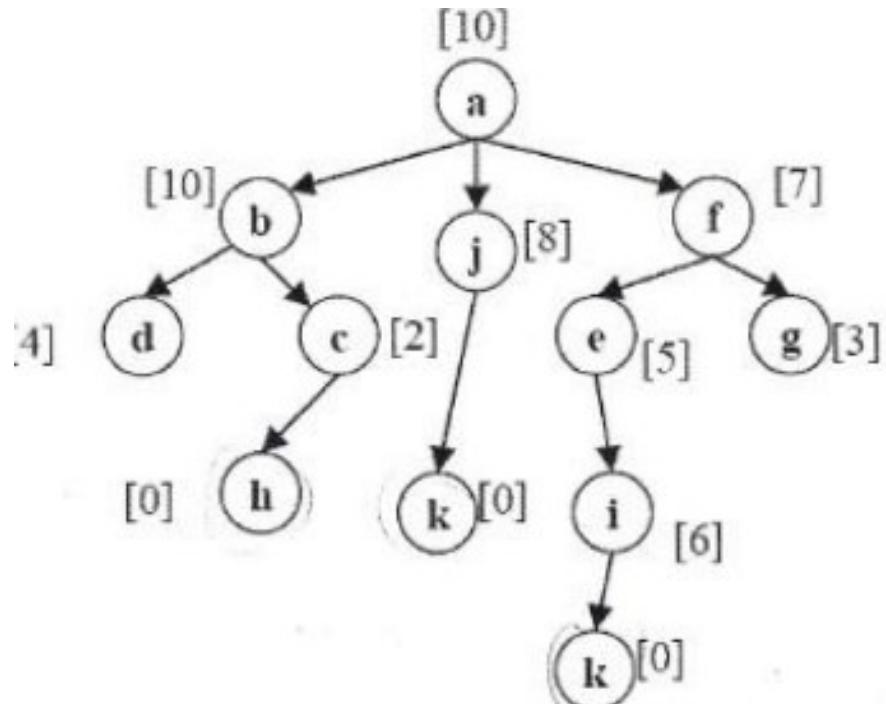
$$\text{value}(i_6) = -\text{heuristic}(i_6) = -6$$

$$\Delta E = 0 - (-6) = +6$$

$\because \Delta E > 0$

$\therefore k_0$ will be selected with probability 1

EXAMPLE



Current	Children
a	---
a	f_7, j_8, b_{10}
f	e_5, g_3
e	i_6
i	k_0
k	
GOAL	



GENETIC ALGORITHM

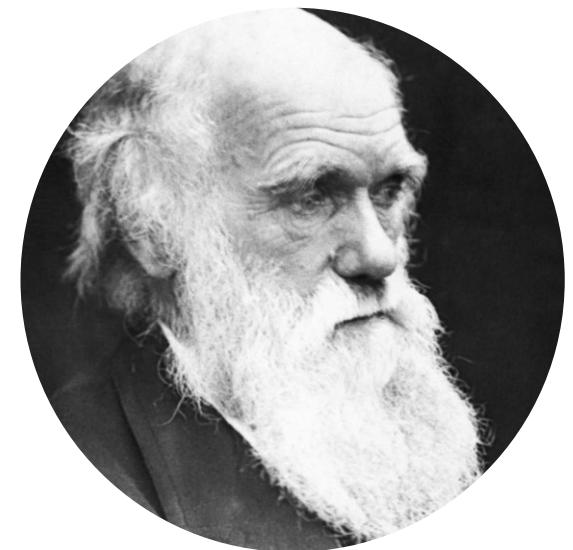
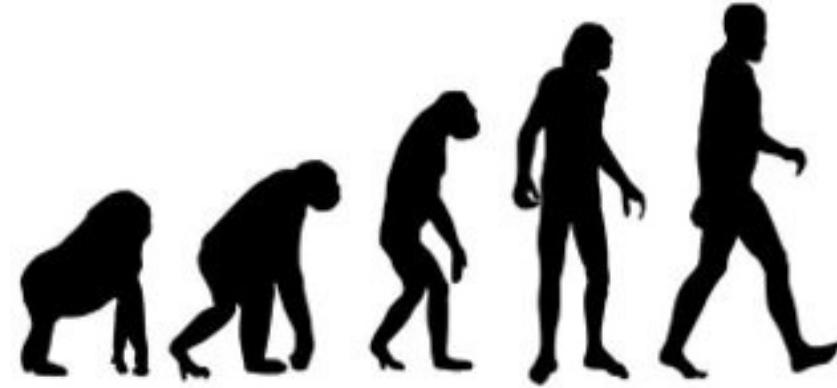
GENETICALGORITHM

- A **genetic algorithm** is an **adaptive heuristic search algorithm** that is inspired by **Charles Darwin's theory of natural evolution** which is **survival of the fittest**.
- **Basic idea of Darwin's theorem:** process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation.

"If a population want to thrive, it must improve by itself constantly, it's the survival of the fittest. The best element of the population should inspire the offspring, but the other individuals must not be forgotten in order to maintain some diversity and be able to adapt in case of a variation of the natural environment." -Charles Darwin

Why GA?

GA's do not break easily even if inputs changed slightly or even in the presence of reasonable noise. Offer more robust results than other typical search optimization techniques.



GENETIC ALGORITHM

- Developed by **John Holland(1975)**.
- GA's are categorized as **global search heuristics**.
- **GA** are used for **solving optimization problems** generally **maximization or minimization** kind of problems in order to gets best output from a set of input.
- GA's are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as **inheritance, mutation, selection, and crossover**
- GAs have 2 essential components: **1. Survival of the fittest (selection) 2. Variation**



BASICS OF GENETIC ALGORITHM

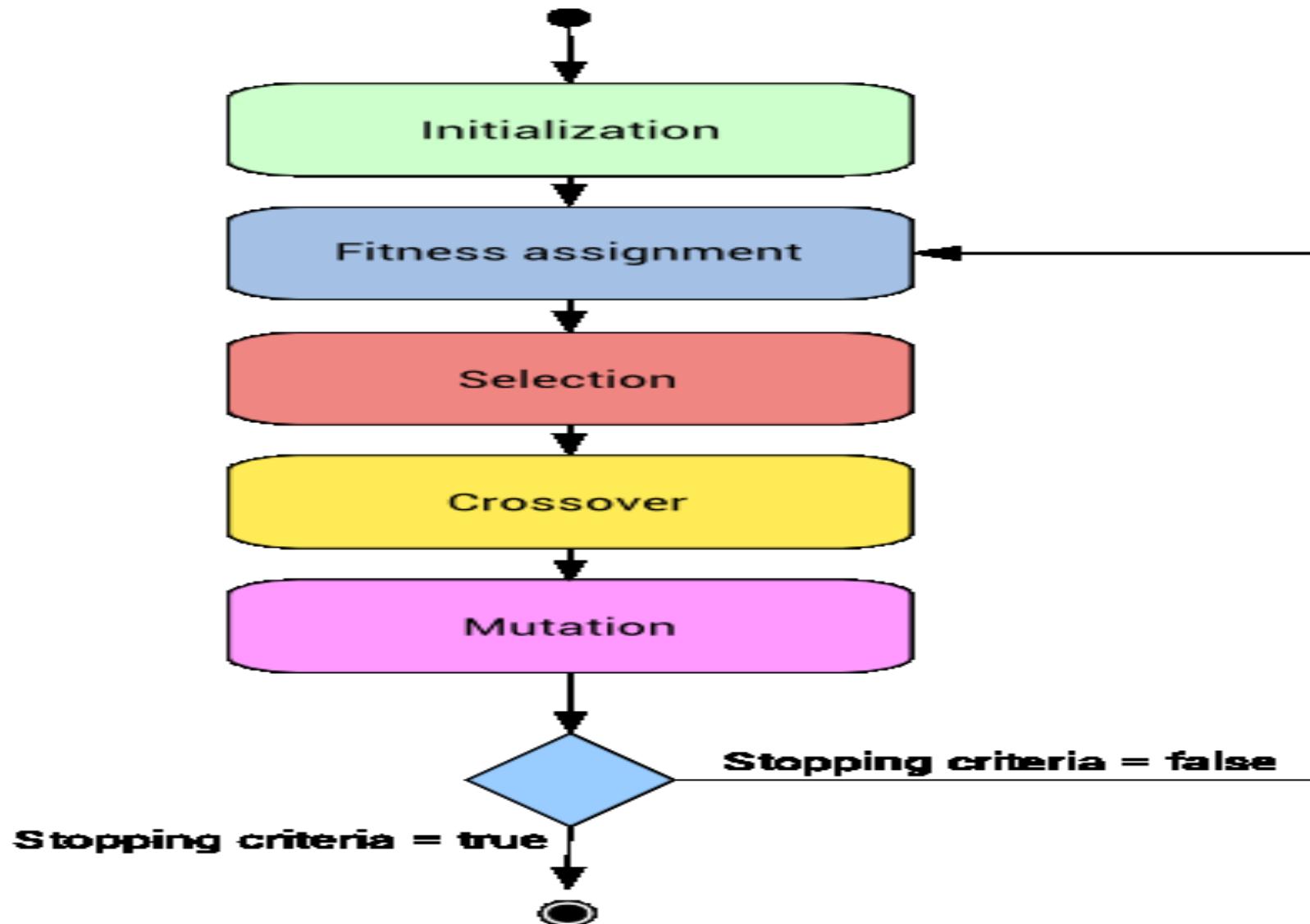
- GAs begin with a set of k randomly generated states, called the **population**. Each **state, or individual**, is represented as a **string over a finite alphabet—most commonly, a string of 0s and 1s**.
- Each state is rated by the objective function, or (in GA terminology) the **fitness function**. A fitness function should return higher values for better states. In each generation, the fitness of every individual in the population is evaluated.
- A genetic algorithm (or GA) **generates successor states by combining two parent states** rather than by modifying a single state.
- The algorithm **terminates** when either a **maximum number of generations has been produced**, or a **satisfactory fitness level has been reached for the population**.
- Produce the next generation of states by “simulated evolution”
 - **Random selection**
 - **Crossover**
 - **Random mutation**

STOCHASTIC OPERATORS OF GENETIC ALGORITHM

A genetic algorithm maintains a population of candidate solutions for the problem at hand, and makes it evolve by iteratively applying a set of stochastic operators

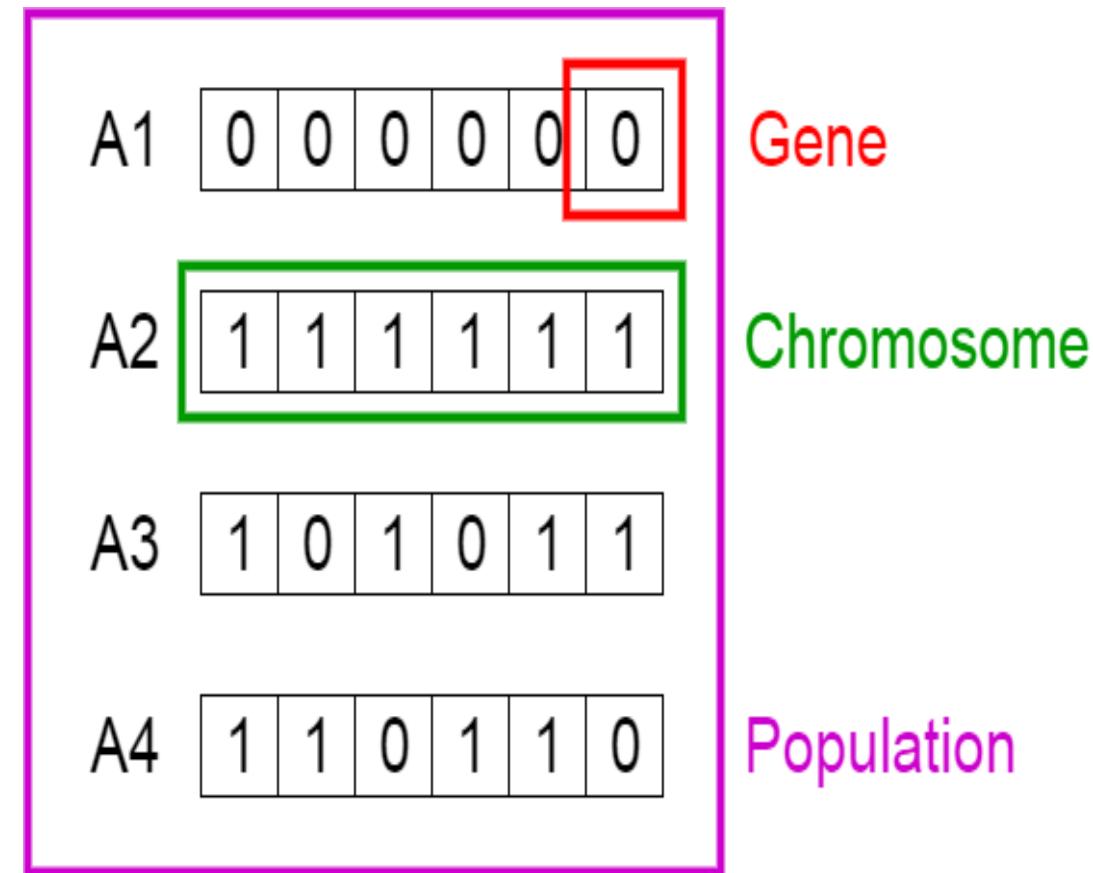
- 1) Selection Operator:** The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to the successive generations.
- 2) Crossover Operator(recombination):** This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring).
- 3) Mutation Operator:** The key idea is to insert random genes in offspring to maintain the diversity in population to avoid the premature convergence.
- 4)Fitness function:** Numerical function estimate the quality of each individual.

STAGES OF GENETIC ALGORITHM



INITIAL POPULATION

- Process begins with a set of individuals which is called a **Population**.
- An individual is characterized by a set of parameters (variables/attributes) known as **Genes(solution component)**. Genes are joined into a string to form a **Chromosome** (solution).
- **Individual:** A possible solution of the problem.
- **Allele:** A gene value [eye color=green]
- In a genetic algorithm, the set of genes of an



individual is represented using a **string**, in terms of an alphabet. Usually, binary values are used (string of 1s and 0s).

- Initially many individual solutions are **randomly generated** to form an **initial population**. The **population size** depends on the **nature of the problem**, but typically contains several hundreds or thousands of possible solutions

CHROMOSOMES REPRESENTATION

- Chromosome representation has great influence in the problem resolution
- Initially represented as binary string
- Nowadays, other possibilities:
 - **Bit strings (0101 ... 1100)**
 - **Real numbers (43.2 -33.1 ... 0.0 89.2)**
 - **Permutations of element (E11 E3 E7 ... E1 E15)**
 - **Lists of rules (R1 R2 R3 ... R22 R23)**
 - **Program elements (genetic programming)**
 - **... any data structure ...**

FITNESS FUNCTION

- **Fitness function** determines the fitness of individuals, in general terms, its ability to compete with other individuals and be the final desired output. It gives *fitness scores* to everyone and the probability that an individual will be selected for reproduction is dependent on its fitness score.
- Eg: fitness score = (number of char correct) / (total number of char)
- Evaluation function (fitness function) has Higher values for better states. It is opposite to heuristic function, e.g., # non-attacking pairs in 8-queens

SELECTION

- The idea of **selection** phase is to select the fittest individuals and let them pass their genes to the next generation.
- During each successive generation, **a proportion of the existing population** is selected to breed a new generation.
- Individual solutions are selected through a ***fitness-based process***, where fitter solutions (as measured by a fitness function) are typically more likely to be selected.
- There are many variants of selection rule
 - **Culling:** All individuals below a threshold is discarded(converge faster)
- Most **selection functions are stochastic** and designed so that a **small proportion of less fit solutions are selected**. This helps keep the **diversity of the population large**, preventing **premature convergence** on poor solutions. Popular and well-studied selection methods include **roulette wheel selection** and **tournament selection**.

Crossover

A1

0	0	0	0	0	0
---	---	---	---	---	---

A2

1	1	1	1	1	1
---	---	---	---	---	---

Crossover
point

A1

0	0	0	0	0	0
---	---	---	---	---	---

A2

1	1	1	1	1	1
---	---	---	---	---	---

A5

1	1	1	0	0	0
---	---	---	---	---	---

A6

0	0	0	1	1	1
---	---	---	---	---	---

- **Crossover** is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a **crossover point** is chosen at random from within the genes.
- There are different types of cross overs like **single point crossover, two point, uniform, heuristic, arithmetic**
- **Offspring** are created by exchanging the genes of parents among themselves until the crossover point is reached. The offspring are added to the population.

MUTATION

- In certain new offspring formed, some of their genes can be subjected to a **mutation** with a low random probability. This implies that some of the bits in the bit string can be flipped.
- Mutation occurs to **maintain diversity within the population** and **prevent premature convergence**.
- Mutation occurs according to user definable mutation probability which is set to a **fairly lower value that is 0.01**.
- It prevent population from **stagnating at a local optima**.
- **Types:** Flip Bit, Boundary, Uniform, Non-Uniform, Gaussian.

Before Mutation

A5	1	1	1	0	0	0
----	---	---	---	---	---	---

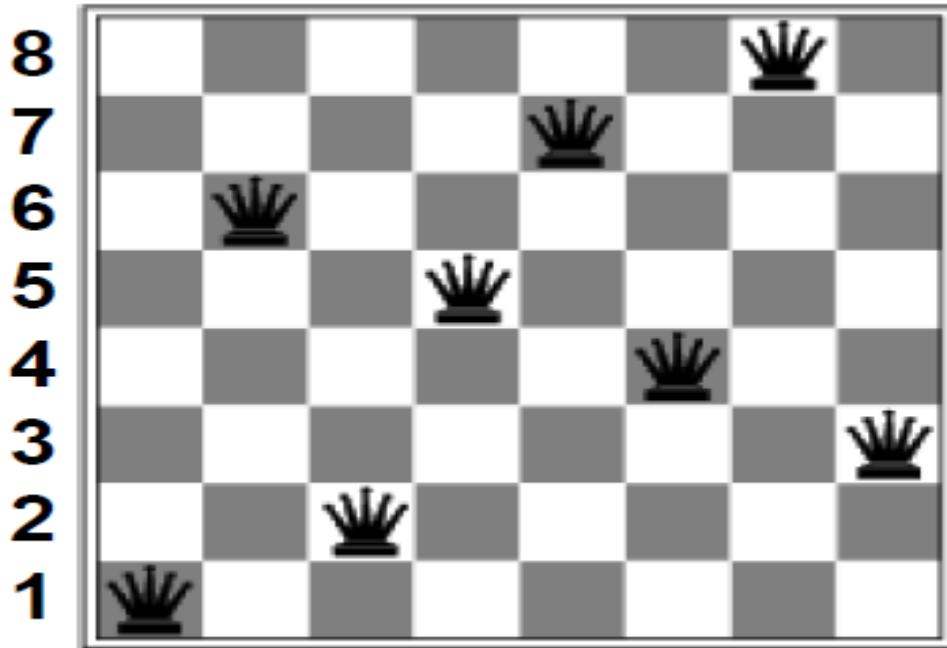
After Mutation

A5	1	1	0	1	1	0
----	---	---	---	---	---	---

TERMINATION

- The algorithm terminates when either a **maximum number of generations** has been produced, or a **satisfactory fitness level has been reached for the population**.
- The algorithm terminates if the **population has converged** (does not produce offspring which are significantly different from the previous generation).

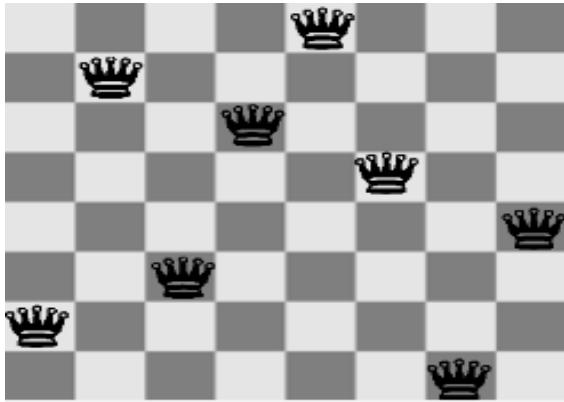
8-QUEENS USING GENETIC ALGORITHMS



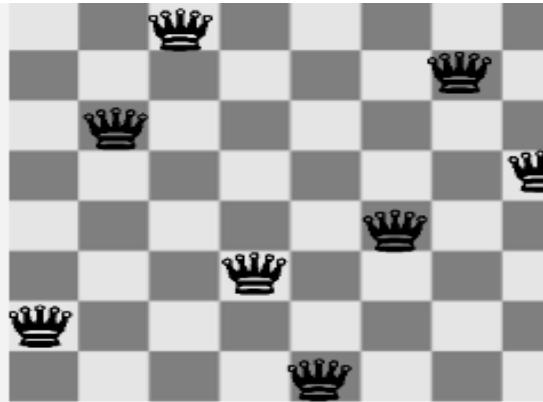
String representation
16257483

States: assume each queen has its own column, represent a state by listing a row where the queen is in each column (digits 1 to 8)

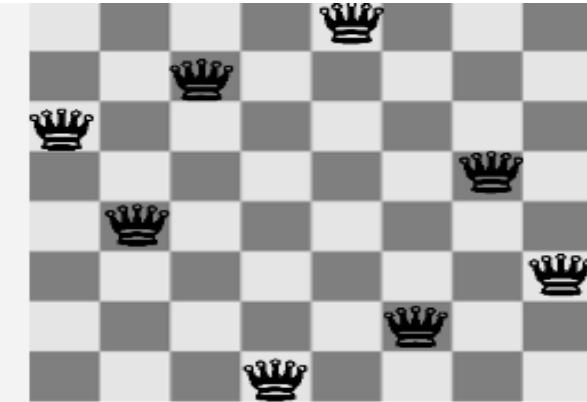
DIFFERENT STATE REPRESENTATION



[1 6 2 5 7 4 0 3]

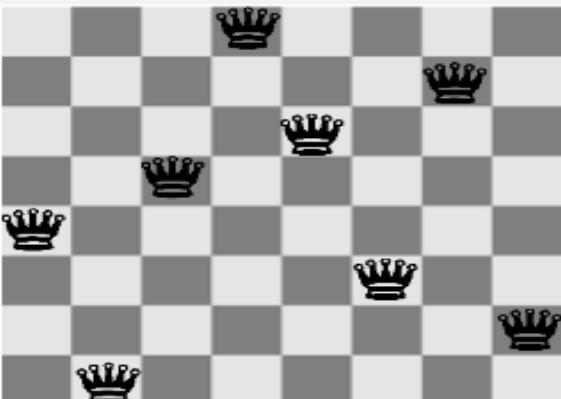


[1 5 7 2 0 3 6 4]

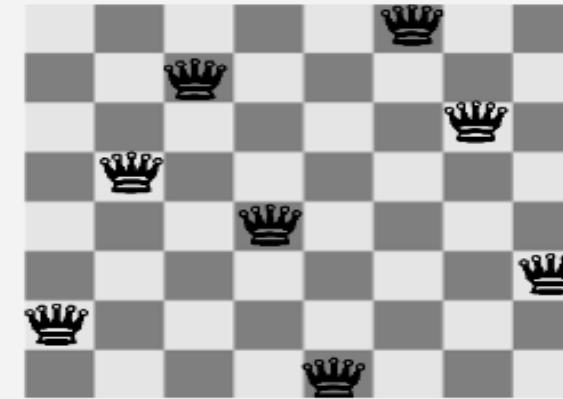


[5 3 6 0 7 1 4 2]

[3 0 4 7 5 2 6 1]



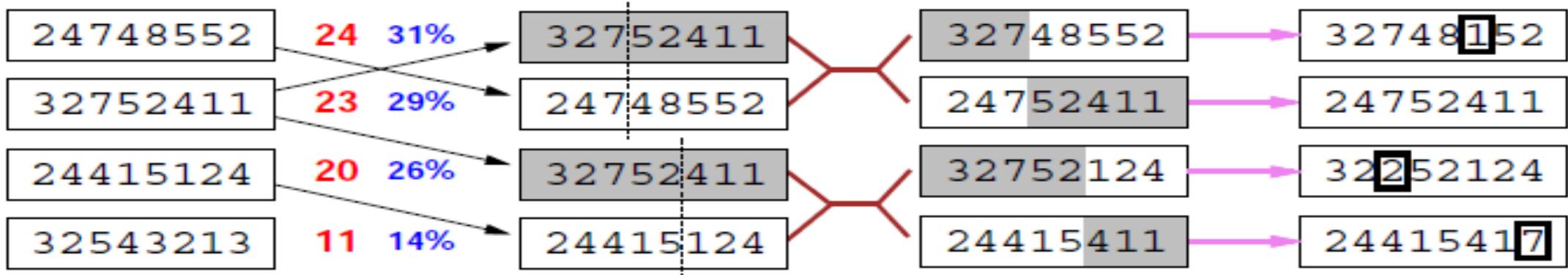
[1 4 6 3 0 7 5 2]



FITNESS FUNCTION OF 8-QUEENS

- Fitness function: number of non-attacking pairs of queens (min = 0, max = $8 \times 7/2 = 28$)
- **Fitness fn** is proportional to the number of clashes amongst the queens. There are 28 clashes possible in an 8×8 chessboard. Let clashes=number of clashing queens
- To determine clashes, each queen must be checked for
 - Row clashes
 - Column clashes
 - Diagonal clashes
- **fitness=28-clashes**
- Therefore, if an individual has high fitness, it will have lower number of clashes. Then, any individual with the maximum fitness will be having least number of clashes.
- Fitness of the state(16257483) given is 27
- Find fitness of following states
 - 24748552
 - 32752411
 - 24415124
 - 32543213

GENETIC ALGORITHMS



Fitness Selection Pairs Cross-Over Mutation

**2 pairs of 2 states randomly selected based
on fitness. Random crossover points are
selected.**

FITNESS SCORE PERCENTAGE CALCULATION

$$24/(24+23+20+11) = 31\%$$

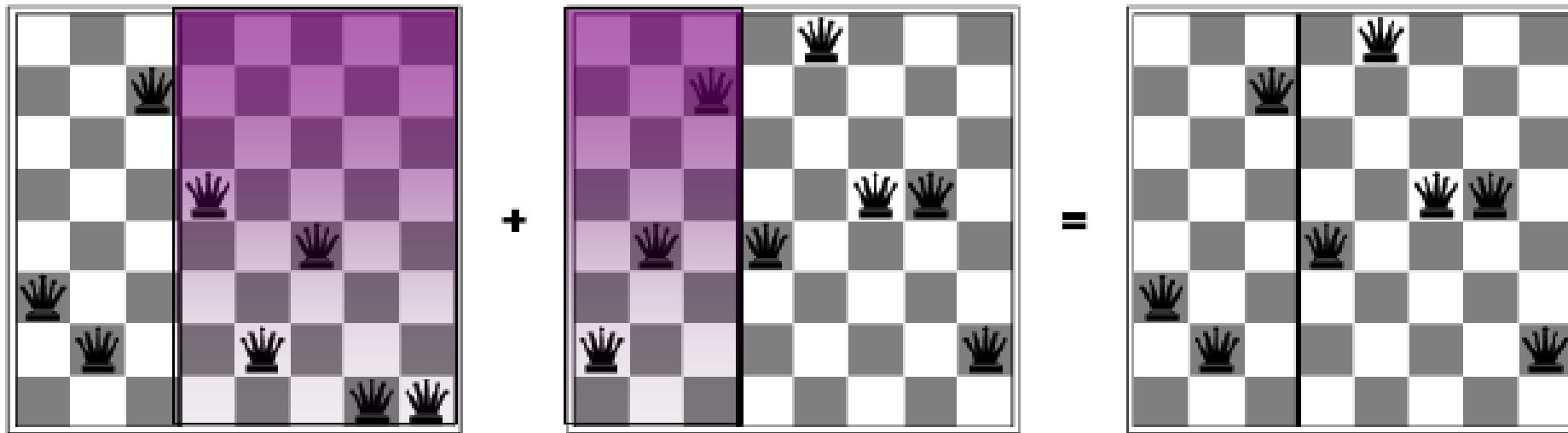
$$23/(24+23+20+11) = 29\% \text{ etc}$$

- 2 Pairs randomly selected for reproduction according to probabilities
- Crossover point is chosen randomly from positions in string(3rd and 5th)
- Mutation corresponds to choosing a queen at random and moving it to a random square in its column.

TERMINATION

keep a limit of iteration. most optimal solution can be described as the **most fit board position**.

EFFECT OF GA



IMPORTANCE OF REPRESENTATION

- Has the effect of “jumping” to a completely different new part of the search space (quite non-local)
- Parts we swap in crossover should result in a well-formed solution (and in addition better be meaningful)
- consider what would happen with binary representation (where position requires 3 digits)
- also, chosen representation reduced search space considerably (compared to representing each square for example)

EXAMPLE 2: MAXONE PROBLEM

- **PROBLEM DEFINITION:** maximize the number of ones in a string of l binary digits
- An individual is encoded (naturally) as a string of l binary digits
- The **fitness function f** of a candidate solution to the MAXONE problem is the number of ones in its genetic code
- We start with a population of n random strings. Suppose that $l = 10$ and $n = 6$

INITIALIZATION

- We toss a fair coin 60 times and get the following initial population:

$s_1 = 1111010101 \ f(s_1) = 7$

$s_2 = 0111000101 \ f(s_2) = 5$

$s_3 = 1110110101 \ f(s_3) = 7$

$s_4 = 0100010011 \ f(s_4) = 4$

$s_5 = 1110111101 \ f(s_5) = 8$

$s_6 = 0100110000 \ f(s_6) = 3$

SELECTION

- we apply fitness proportionate selection with the roulette wheel method:
- Individual i will have a $\frac{f(i)}{\sum_i f(i)}$ probability to be chosen
- repeat the extraction as many times as the number of individuals we need to have the same parent population size (6 in our case)
- after performing selection, we get the following population:

$s_1` = 1111010101 \quad (s_1)=7/7+5+7+4+8+3=7/34$

$s_2` = 1110110101 \quad (s_3)=7/34$

$s_3` = 1110111101 \quad (s_5)=8/34$

$s_4` = 0111000101 \quad (s_2)=5/34$

$s_5` = 0100010011 \quad (s_4)=4/34$

$s_6` = 1110111101 \quad (s_5)=8/34$

CROSSOVER

- mate strings for crossover. For each couple we decide according to crossover probability (for instance 0.6) whether to actually perform crossover or not
- Suppose that we decide to actually perform crossover only for couples (s_1^-, s_2^-) and (s_5^-, s_6^-) . For each couple, we randomly extract a crossover point, for instance 2 for the first and 5 for the second
- Before crossover:

$s_1^- = 11\textcolor{red}{11010101}$

$s_5^- = 01000\textcolor{red}{10011}$

$s_2^- = 11\textcolor{blue}{10110101}$

$s_6^- = 11101\textcolor{blue}{11101}$

- After crossover:

$s_1'' = \textcolor{blue}{1110110101}$

$s_5'' = 01000\textcolor{blue}{11101}$

$s_2'' = 11\textcolor{red}{11010101}$

$s_6'' = 11101\textcolor{red}{10011}$

MUTATION

- The final step is to apply random mutation: for each bit that we are to copy to the new population we allow a small probability of error (for instance 0.1)
- Before applying mutation:

$s_1 = 1110110101$

$s_2 = 1111010101$

$s_3 = 1110111101$

$s_4 = 0111000101$

$s_5 = 0100011101$

$s_6 = 1110110011$

MUTATION

- After applying mutation:

$$s_1''' = 1110100101 \quad f(s_1''') = 6$$

$$s_2''' = 1111110100 \quad f(s_2''') = 7$$

$$s_3''' = 1110101111 \quad f(s_3''') = 8$$

$$s_4''' = 0111000101 \quad f(s_4''') = 5$$

$$s_5''' = 0100011101 \quad f(s_5''') = 5$$

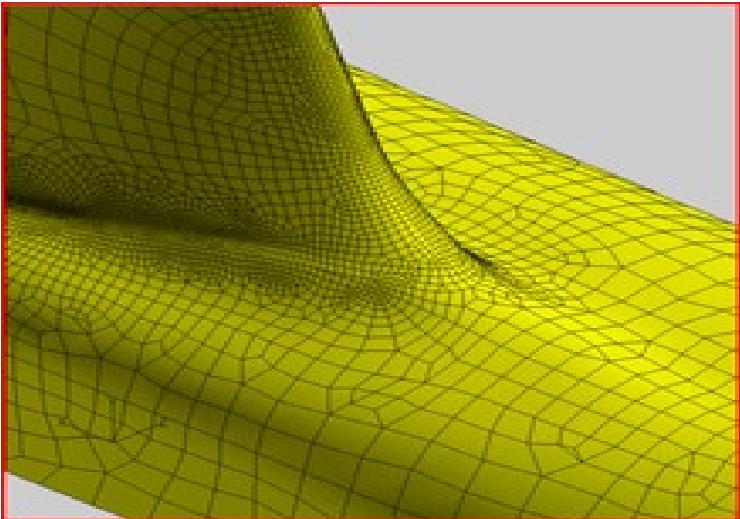
$$s_6''' = 1110110001 \quad f(s_6''') = 6$$

In one generation, the total population fitness changed from 34 to 37, thus improved by ~9%

At this point, we go through the same process all over again, until a stopping criterion is met

APPLICATIONS

- Design jet engines.
- Draw criminals.
- Program computers.
- Criminal-likeness Reconstruction
- Image compression – evolution



THANK
YOU



ADVERSARIAL SEARCH



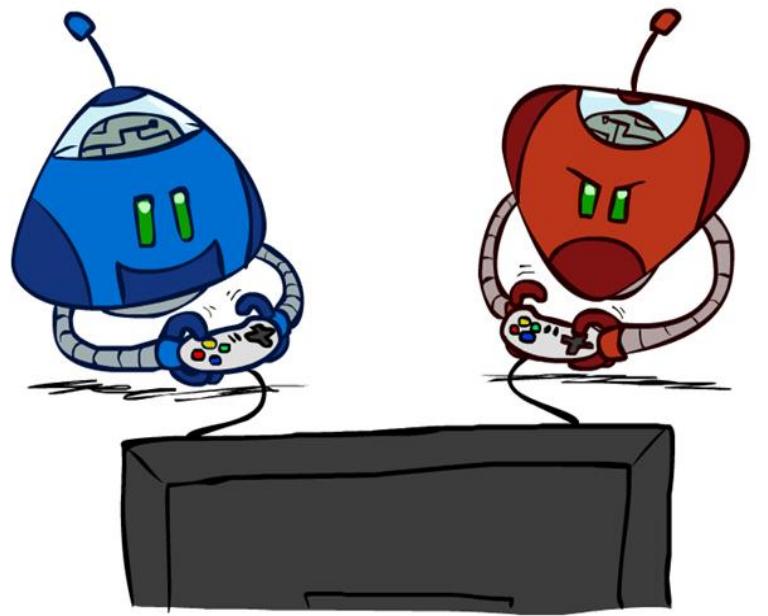
Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us

GAMES AND TYPES

- AI researchers study game playing as it's a **good reasoning problem** which is **formal(well defined rules)**, **unbiased** and **nontrivial** as it require **intelligence** in direct comparison with humans. This can be utilized in development strategies of other computer programs too.

GAME TYPES

1. **Deterministic**(follow a strict pattern and set of rules for the games, and there is no randomness associated.Eg: chess, Checkers,) vs **stochastic**(have various unpredictable events and has a factor of chance or luck Eg: monopoly, bridge, poker)
2. **Single Agent or Multi Agent**
3. **Zero sum**(Agents have opposite utilities (values on outcomes), one maximizes and the other minimizes)
4. **Turn Taking(Poker, monopoly)**
5. **Perfect information** (Gives info abt states, players, actions, transition, terminal test and utilities.Eg: Chess, checkers) vs **Imperfect Information**(agents do not have all information about the game and not aware with what's going on Eg: battleships, bridge, poker)



Game-playing programs like chess, checkers, GO, pacman etc were developed by AI researchers s (1950s)

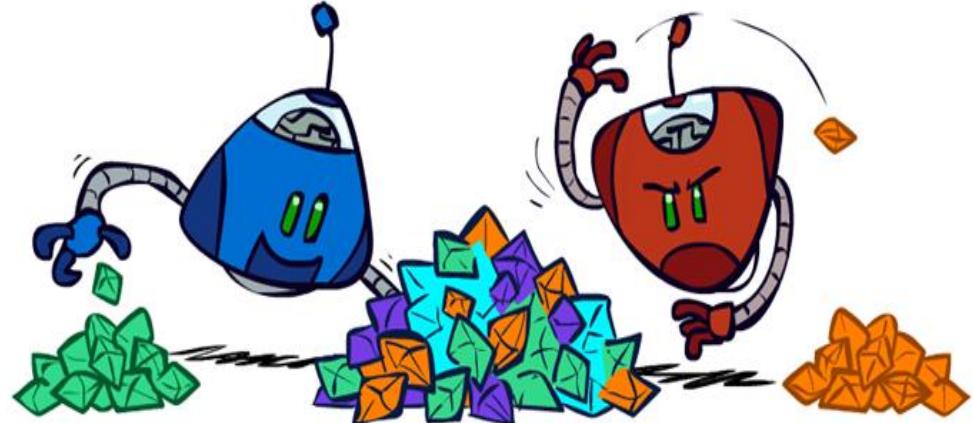
Zero-Sum Games



- Agents have opposite utilities (values on outcomes)
- A single value that one maximizes and the other minimizes
- Adversarial game search is pure competition
- a mathematical representation of a situation in which a participant's gain (or loss) of utility is exactly balanced by the losses (or gains) of the utility of other participant(s)

Eg:Chess and tic-tac-toe

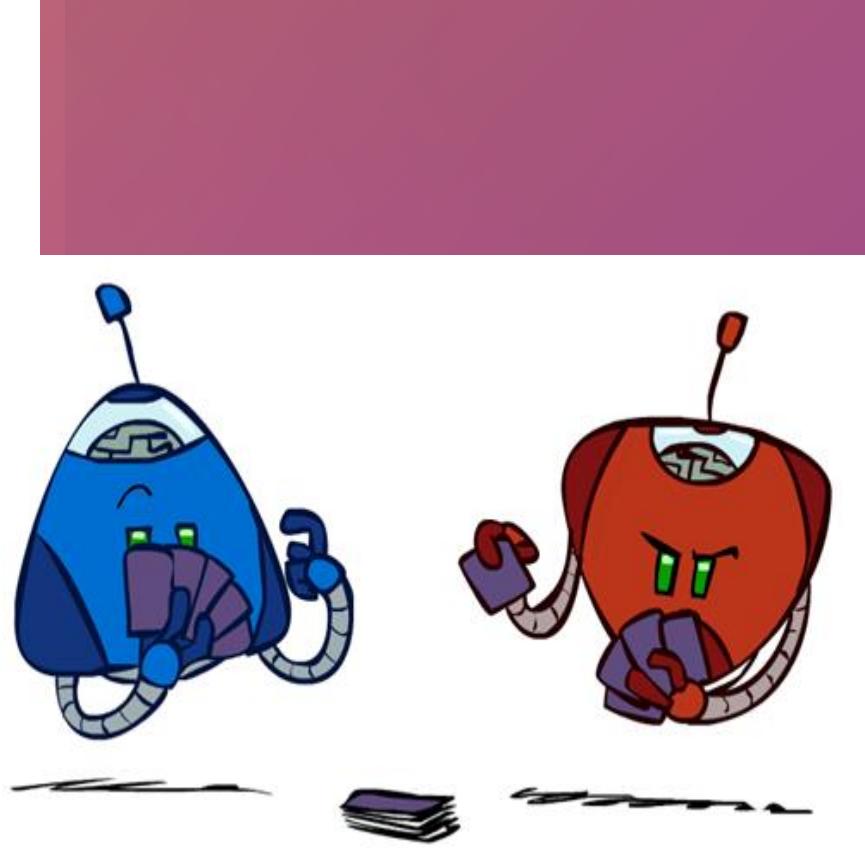
General Games



- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible
- More later on non-zero-sum games

GAME SEARCH

- Competitive environments, in which the agent's goals are in conflict require **adversarial search** are often known as **games**.
- Games are modeled as a Search problem and heuristic evaluation function
- Games require generally **multiagent (MA) environments**
- **Specifics:**
 - Sequences of player's decisions we control
 - Decisions of other player(s) we do not control
- **Contingency problem:** many possible opponent's moves must be “covered” by the solution
- Contingency problem is because of **uncertainty of Opponent's behavior.**
- **Rational opponent** – maximizes its own **utility(payoff) function**



GAMES VS SEARCH

- **Search – no adversary**

- **Solution** is (heuristic) method for **finding goal**
- **Heuristic techniques** can find *optimal* solution
- **Evaluation function:** estimate of cost from start to goal through given node
- The machine is “exploring” the search space by itself.

Examples: path planning, scheduling activities

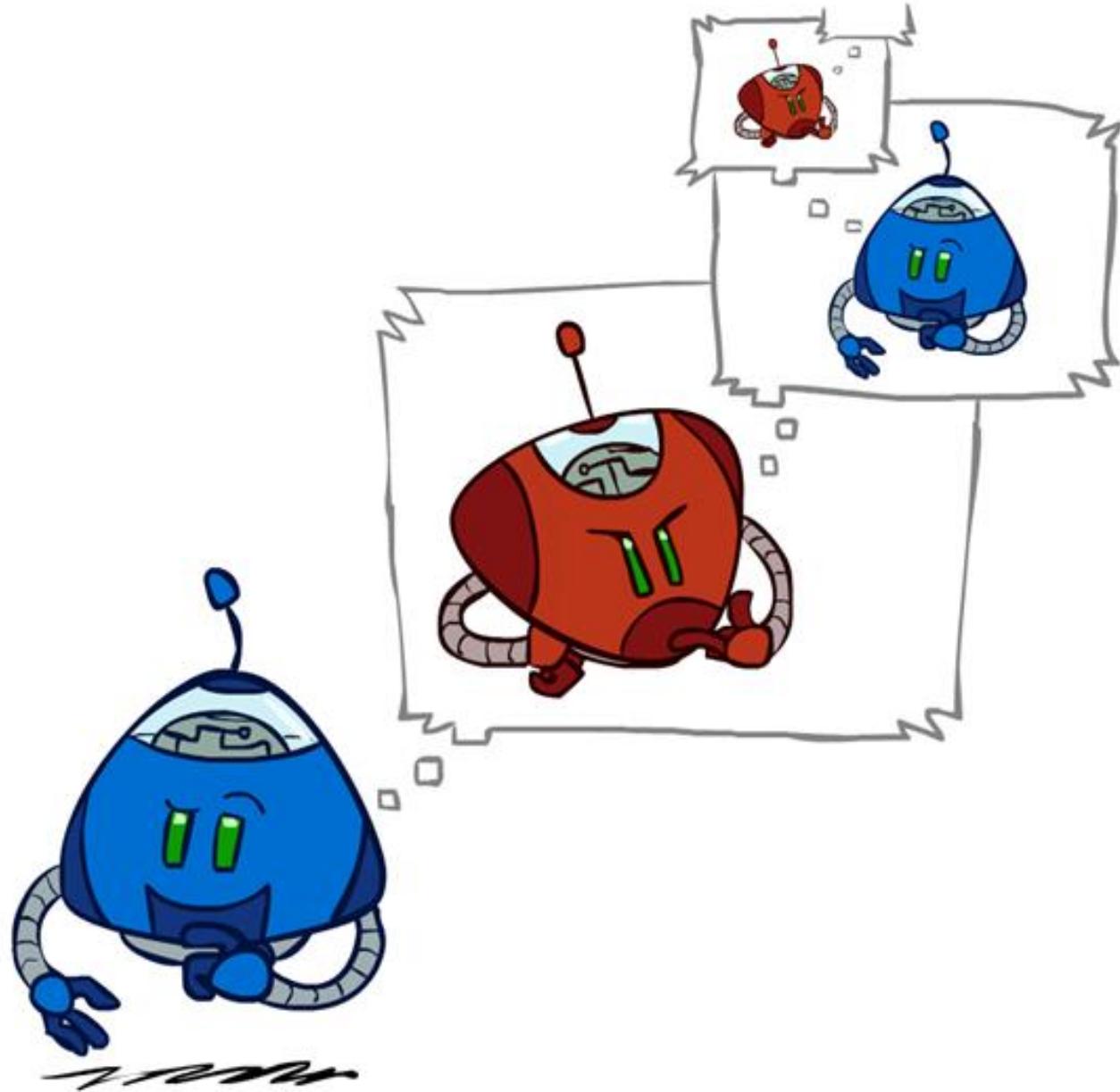
- **Games – adversary**

- Solution is **strategy** (strategy specifies move for every possible opponent reply).
- **Optimality depends on opponent.** Why?
- Time limits force an *approximate* solution
- **Evaluation function:** evaluate “goodness” of game position
- **“Unpredictable” opponent** --> specifying a move for every possible opponent reply
- **Time limits -->** unlikely to find goal, must approximate

Examples: chess, checkers, Othello, backgammon

ADVERSARIAL SEARCH

- Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.
- Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games.



GAME SEARCH PROBLEM FORMULATION AS ADVERSARIAL SEARCH

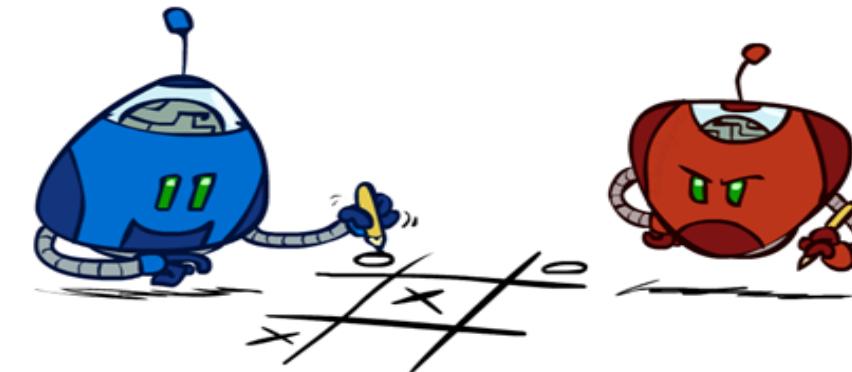
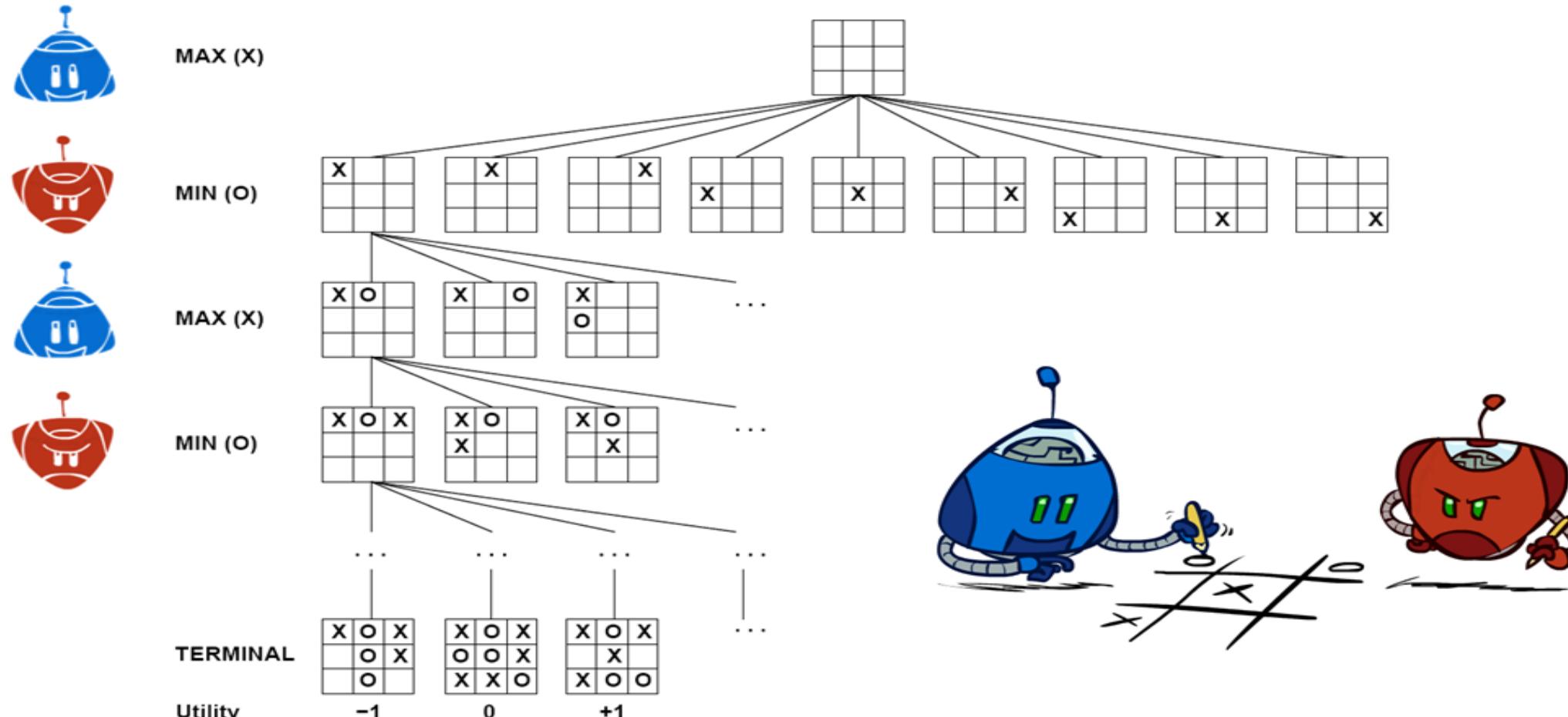
1. **Initial state(S_0):** initial board position + whose move it is. How game is set up at start.
2. **Operators:** legal moves a player can make. **Successor function** returns list of (move, state) pairs, each indicating a legal move and the resulting state.
3. **Goal (terminal test):** It returns true when game is over.
4. **Utility (payoff) function:** measures the outcome of the game and its desirability. A utility function gives the final numeric value for a game that ends in terminal states s for player p . Utility can be thought of as a way to “score” each possible move based on its potential to result in a win.
5. **States:** board configurations.
6. **Result(s,a):** Transition model which defines result of moves.
7. **Player(S):** Defines which player has move in a state.
8. **Actions(S):** Returns the set of legal moves in a state.

Search objective:

- Find the sequence of player’s decisions (moves) maximizing its utility (payoff)
- Consider the opponent’s moves and their utility

GAME TREE

A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function.



- At the leaf nodes, the utility function is employed. Big value means good, small is bad.
- Each layer of game tree is called as Ply.

MIN MAX SEARCH



MIN MAX SEARCH

- Min-max algorithm is a **recursive or backtracking algorithm** which is used in decision-making and game theory
- A strategy/solution for **optimal** decisions
- Min-Max Algorithm uses the **game tree** where each level has the additional information as:
 - **Max** if player wants to **maximize utility/reward**(Agent) (**Best move for player 1**)
 - **Min** if player wants to **minimize utility**(opponent)(**worst move player 1**)
- **Best move strategy** is used by both the players. They fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Min max algorithm calculate values of leaf node and propagate to root node so as to decide which path should be opted for.
- Perfect play for **deterministic environments with perfect information**

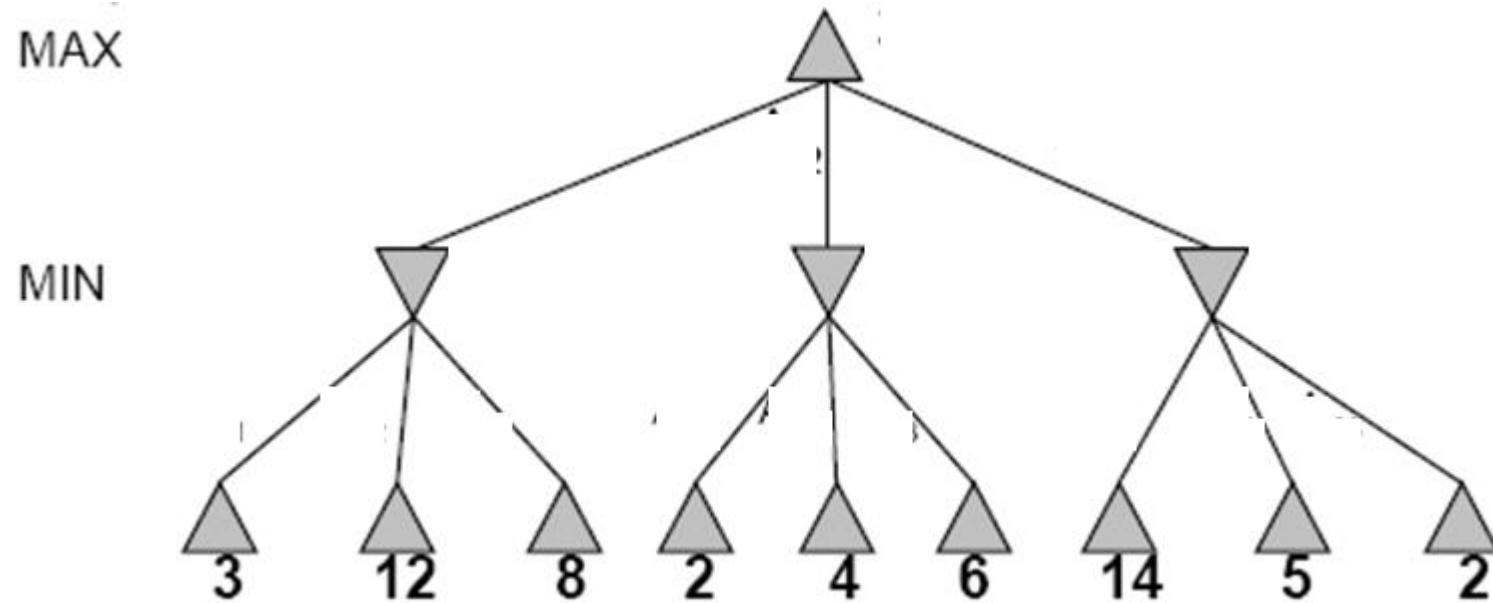
MIN-MAX SEARCH

- **Basic idea:** choose move with highest minimax value = best achievable payoff against best play
- A complete depth-first, recursive exploration of the game tree
- Initialize **max** and **min** with worst values ie, **Max=∞ Min=-∞**

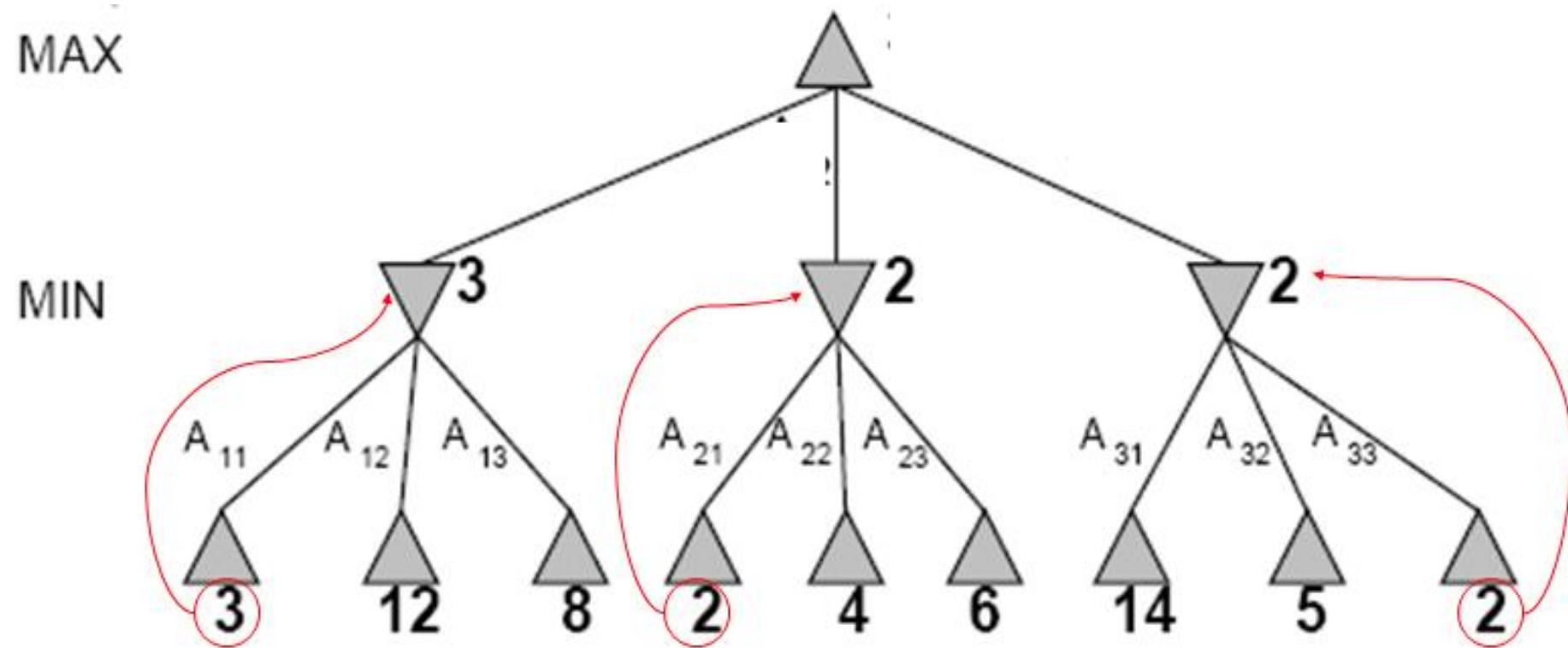
Algorithm:

1. Generate game tree completely
 2. Determine utility of each terminal state
 3. Propagate the utility values upward in the tree by applying MIN and MAX operators on the nodes in the current level
 4. At the root node use minimax decision to select the move with the max (of the min) utility value
- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.

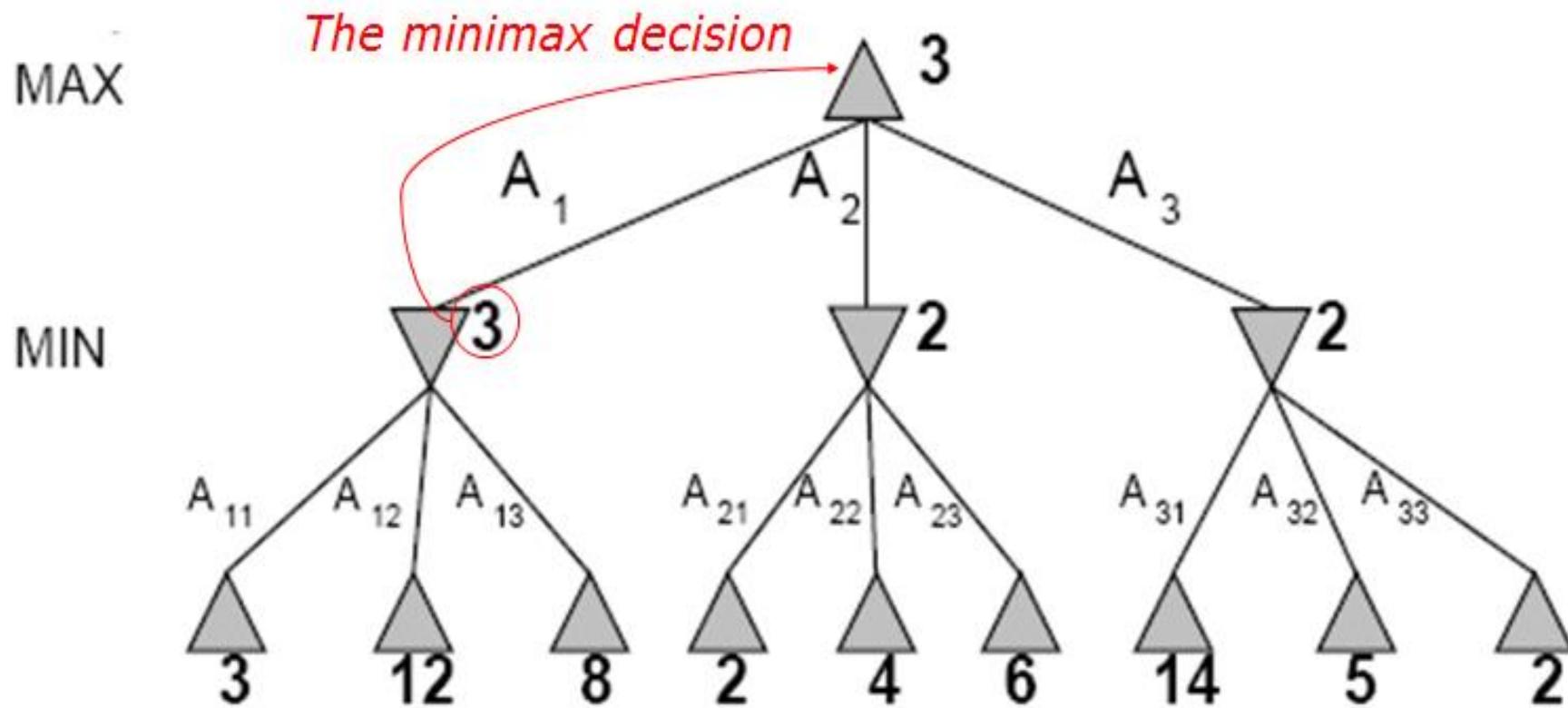
MIN-MAX SEARCH EXAMPLE



MIN-MAX SEARCH EXAMPLE

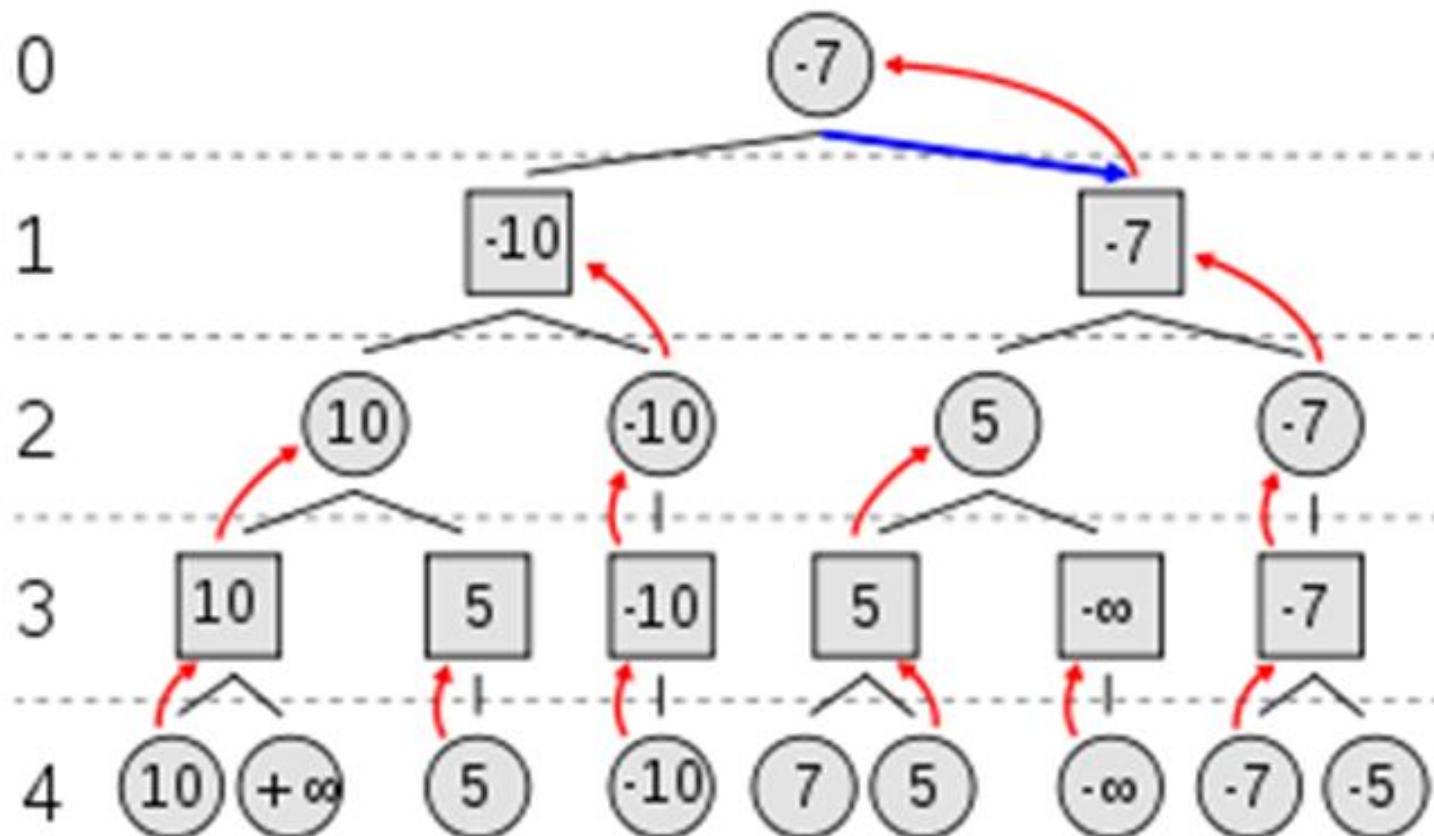


MIN-MAX SEARCH EXAMPLE



EXAMPLE

MAX to move

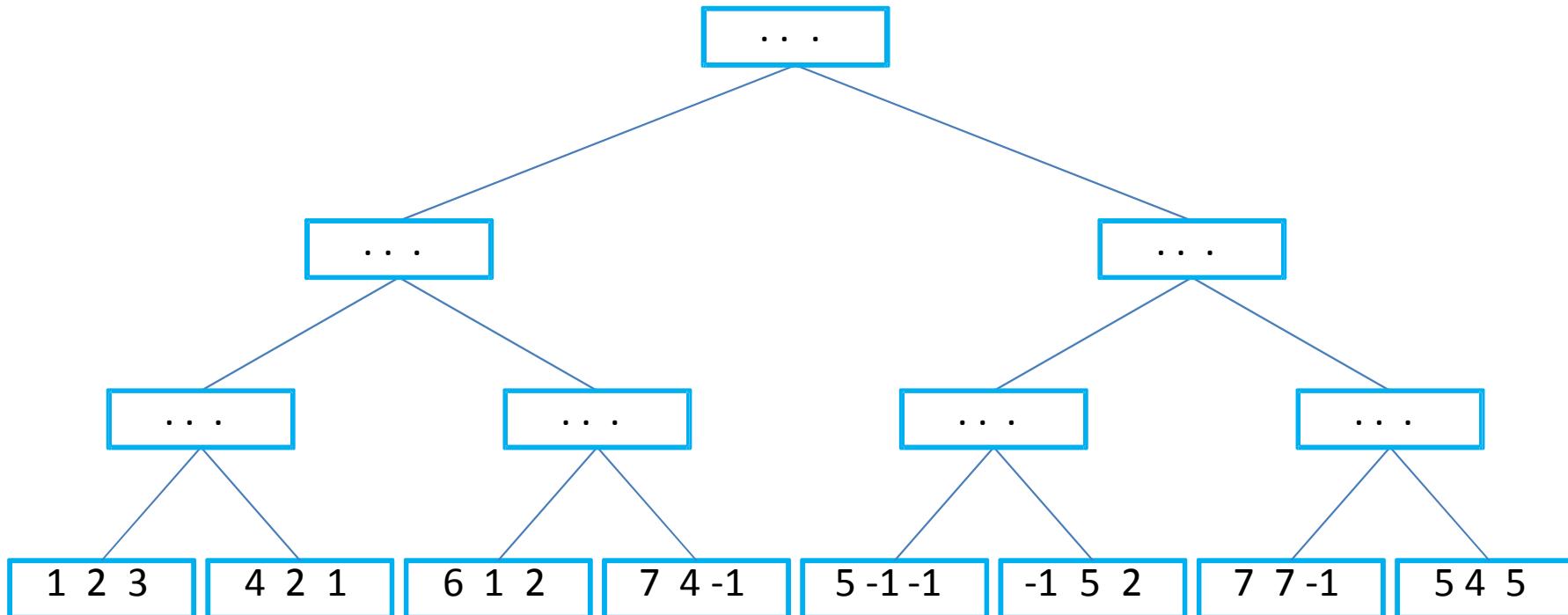


OPTIMAL DECISION IN MULTIPLAYER GAME

- Extend the minimax idea to multiplayer games
- Replace the single value for each node with a vector of values (utility vector)
- Extend the minimax idea to multiplayer games
 - Replace the single value for each node with a vector of values (utility vector)

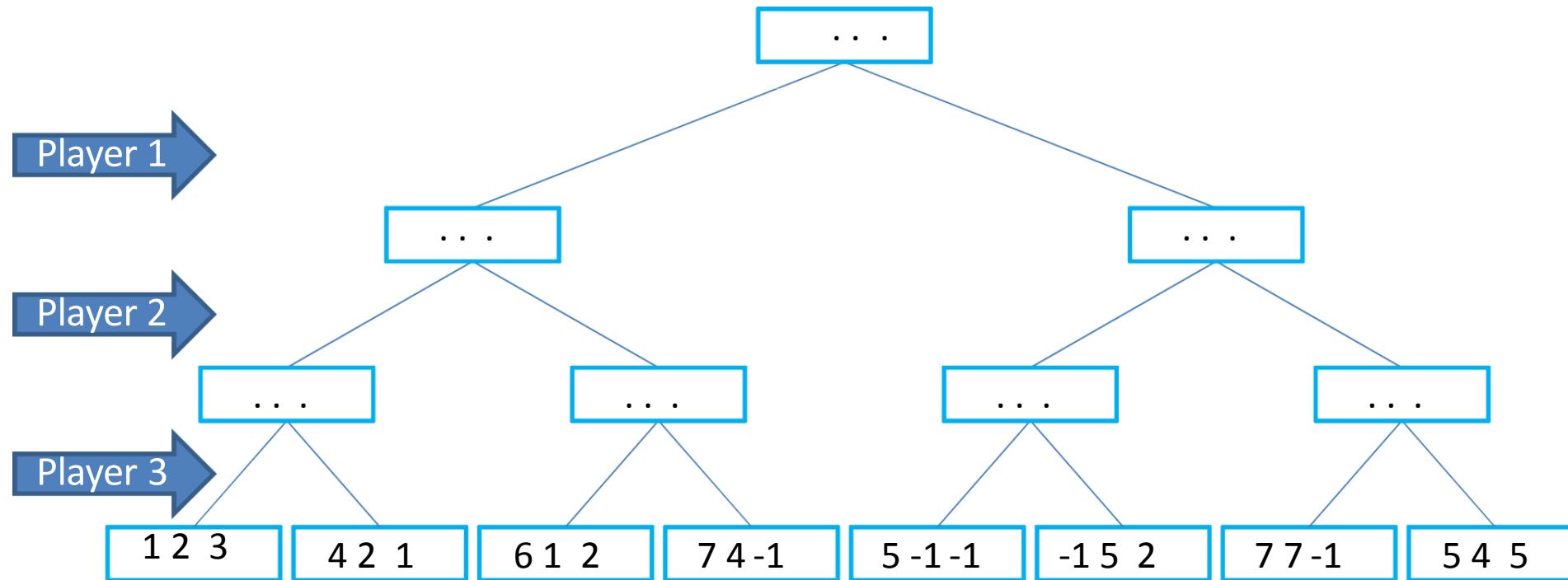
MiniMax For 3 Players

- All players are Max
- Evaluation function given by vector



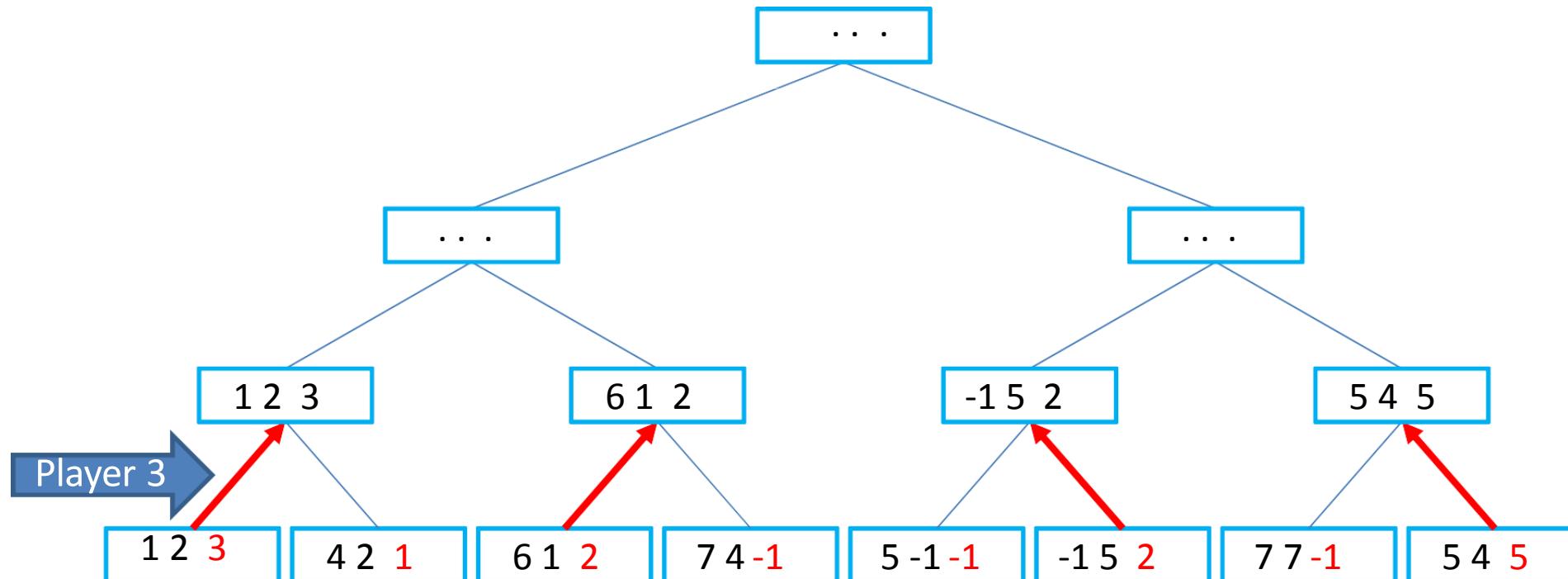
MiniMax For 3 Players

- Each layer assigned to 1 player
- Turn: every 3 layers



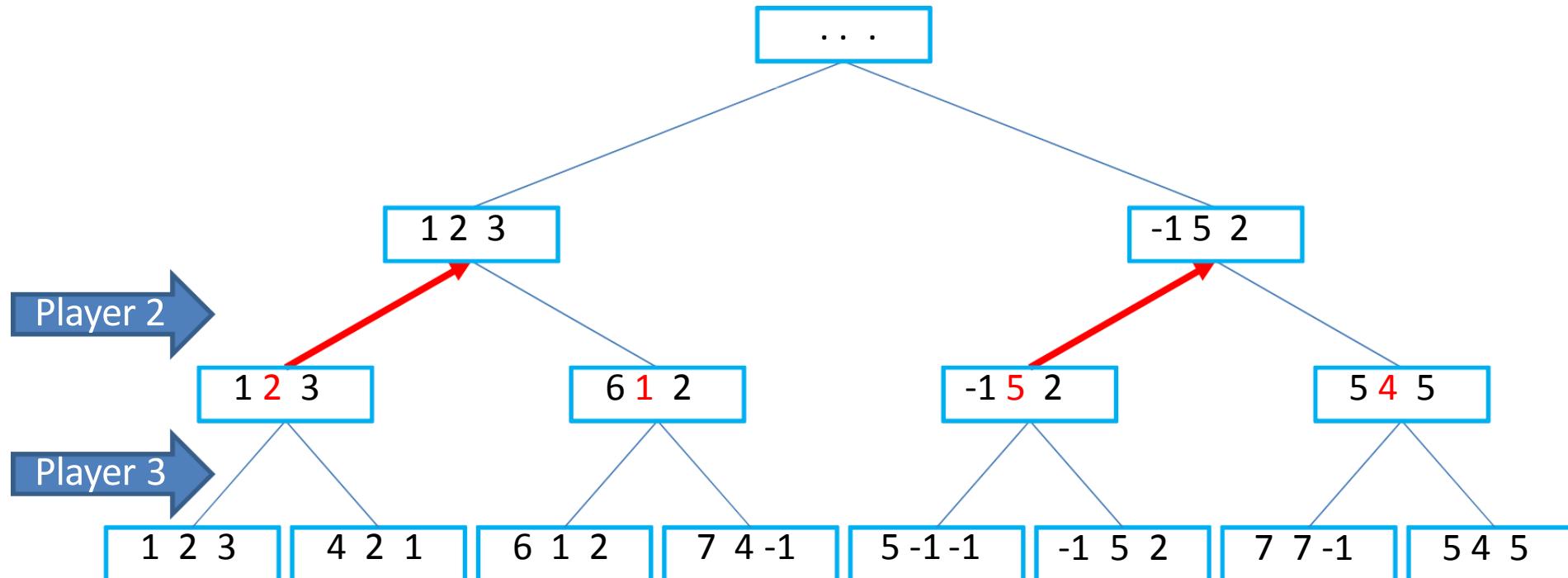
MiniMax For 3 Players

- $\text{MaxThirdPlayer}([1,2,3],[4,2,1]) = [1,2,3]$
- $\text{MaxThirdPlayer}([6,1,2],[7,4,-1]) = [6,1,2]$
- $\text{MaxThirdPlayer}([5,-1,-1],[-1,5,2]) = [-1,5,2]$
- $\text{MaxThirdPlayer}([7,7,-1],[5,4,5]) = [5,4,5]$



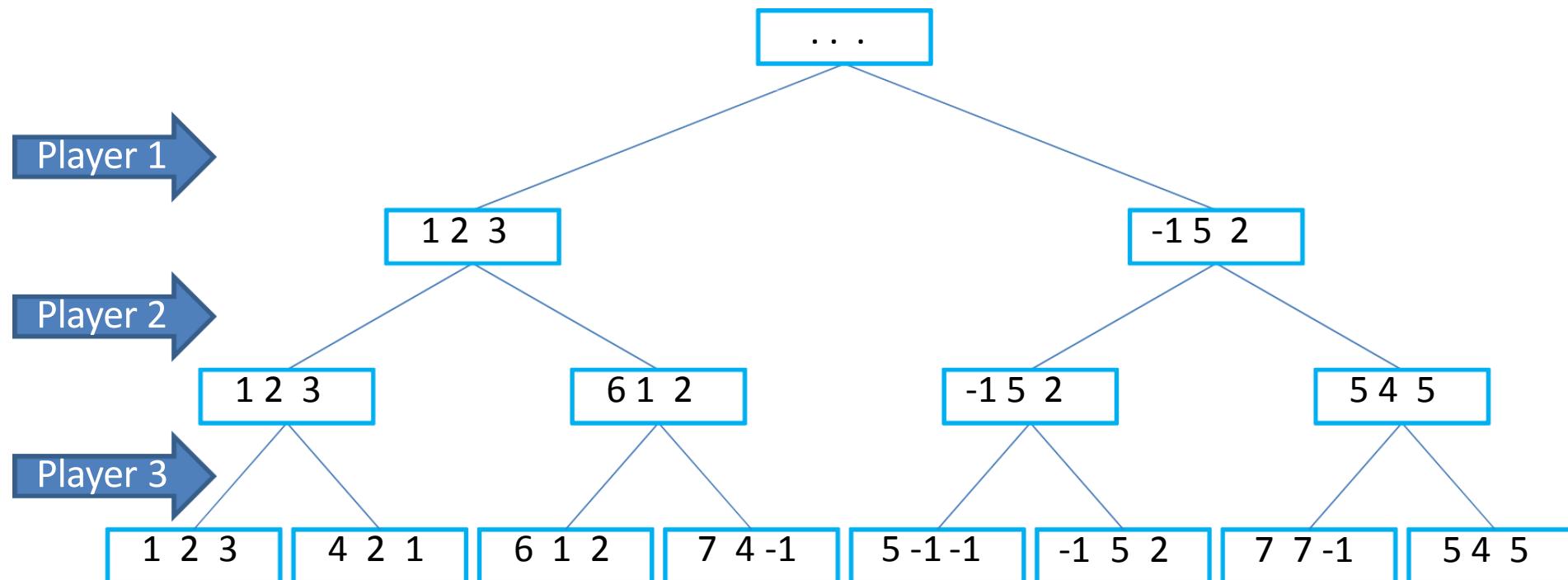
MiniMax For 3 Players

- $\text{MaxSecondPlayer}([1,2,3],[6,1,2]) = [1,2,3]$
- $\text{MaxSecondPlayer}([-1,5,2],[5,4,5]) = [-1,5,2]$



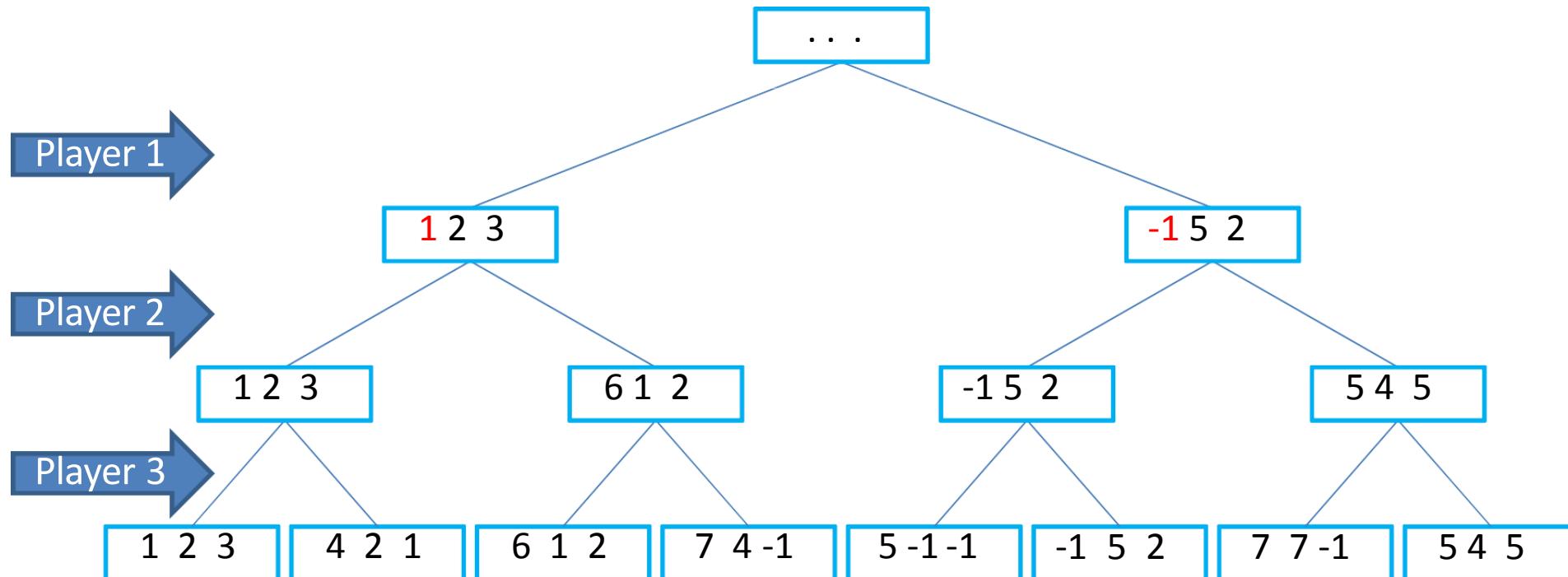
MiniMax For 3 Players

- First player's move



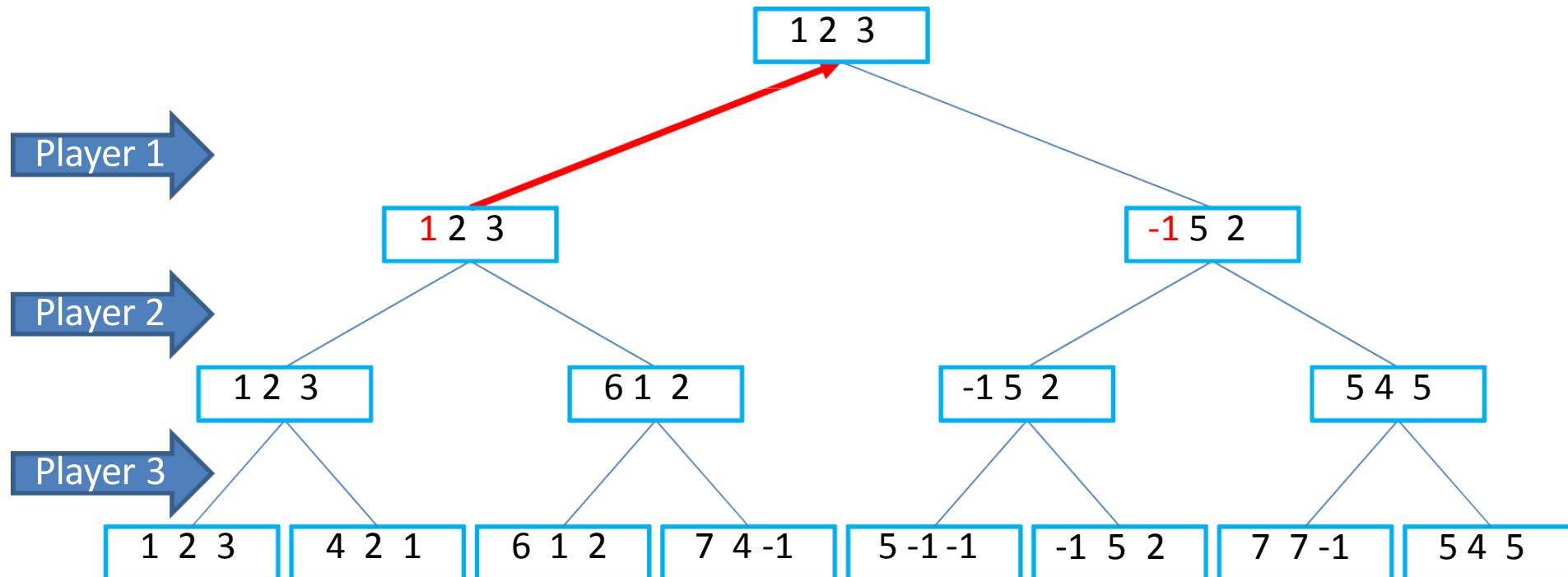
MiniMax For 3 Players

- Max first player: first position of vector



MiniMax For 3 Players

- $\text{MaxFirstPlayer}([1,2,3], [-1,5,4]) = [1,2,3]$



MULTIPLAYER GAMES

MAX

to move
A

MIN

B

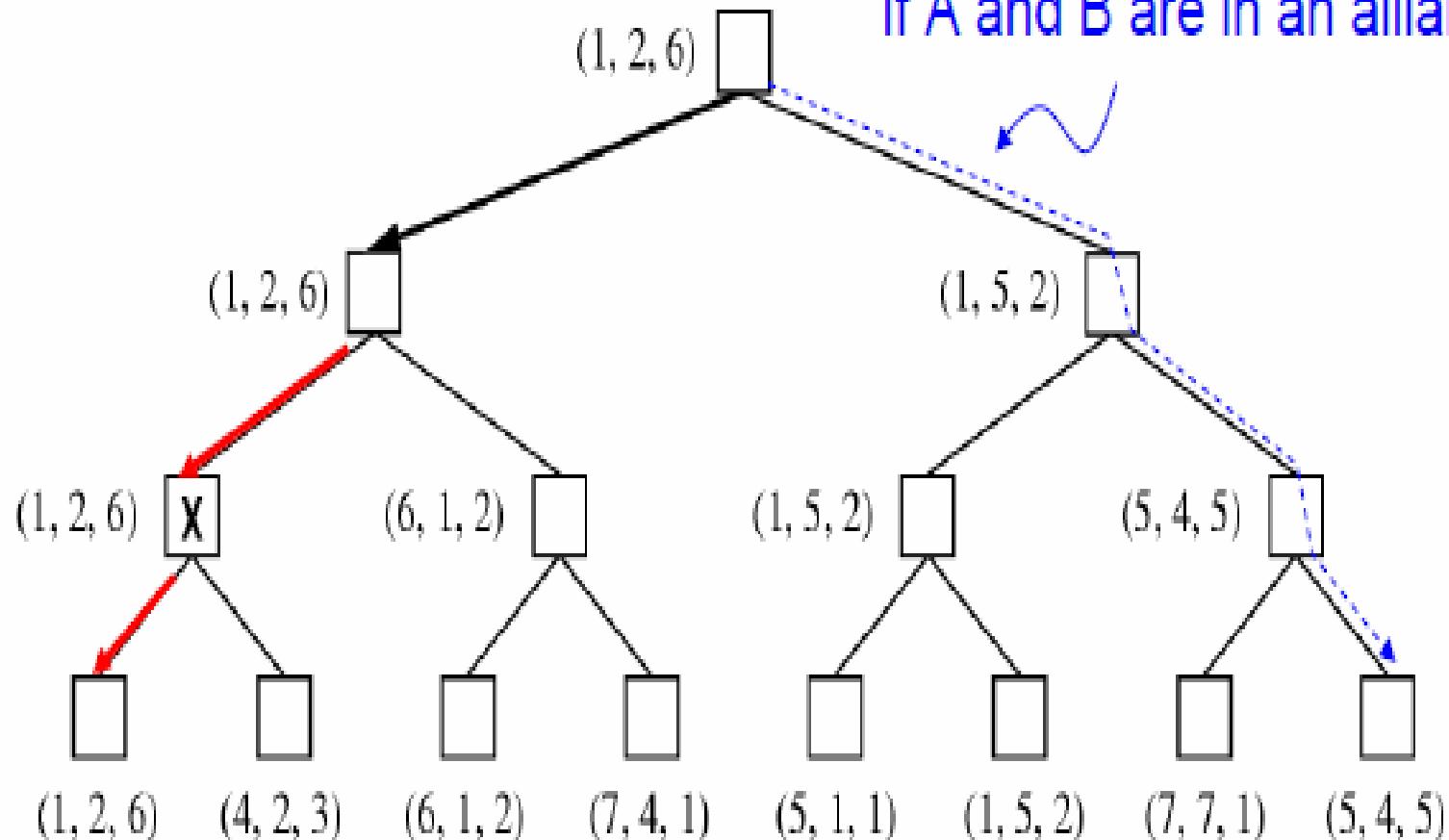
MAX

C

MIN

A

If A and B are in an alliance



MIN MAX EFFICIENCY

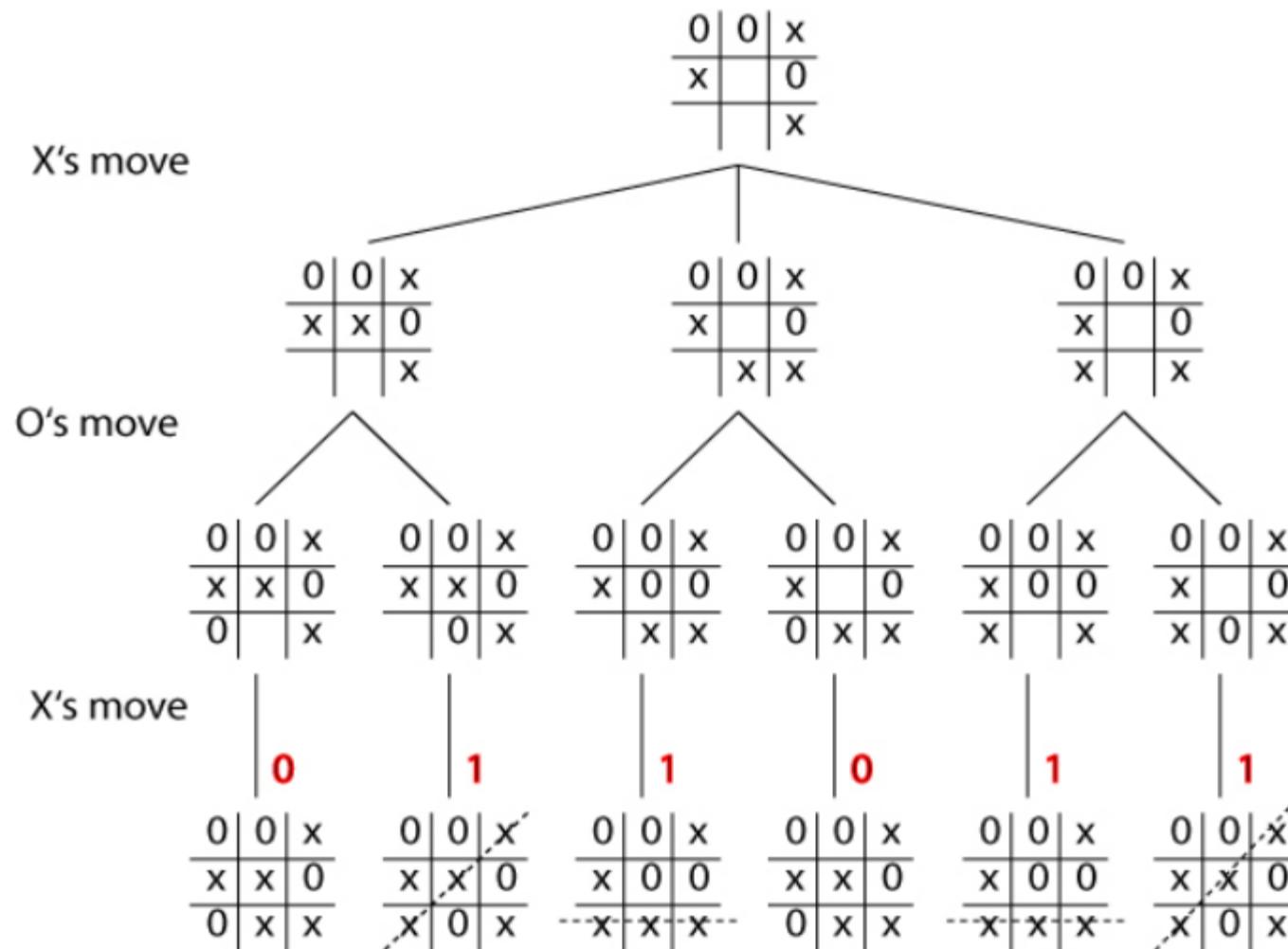
- How efficient is minimax?
 - **Complete?** Yes (if tree is finite)
 - **Optimal?** Yes (against an optimal opponent)
 - **Time:** $O(b^m)$ Just like (exhaustive) DFS
 - **Space:** $O(bm)$
- **Disadvantage:** Evaluate all nodes
- It has a huge branching factor, and the player has lots of choices to decide.
This limitation of the minimax algorithm can be improved from **alpha-beta pruning**

Static Evaluation:

Game playing with Mini-Max

'+1' for a win, '0' for a draw

■ Criteria '+1' for a Win, '0' for a Draw

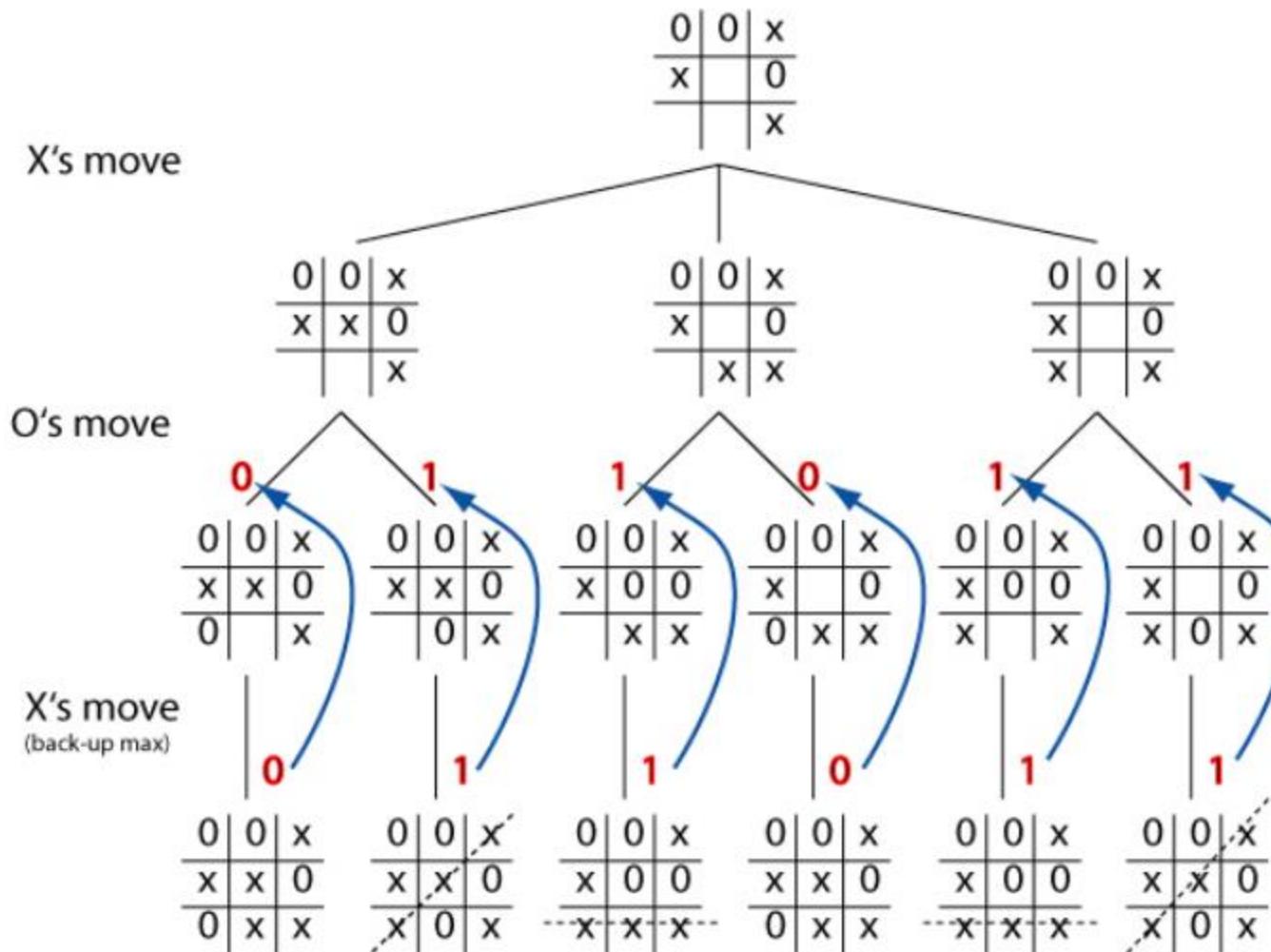


Back-up the Evaluations:

Game playing with Mini-Max

Level by level, on the basis of opponent's turn

■ Up : One Level



MAX

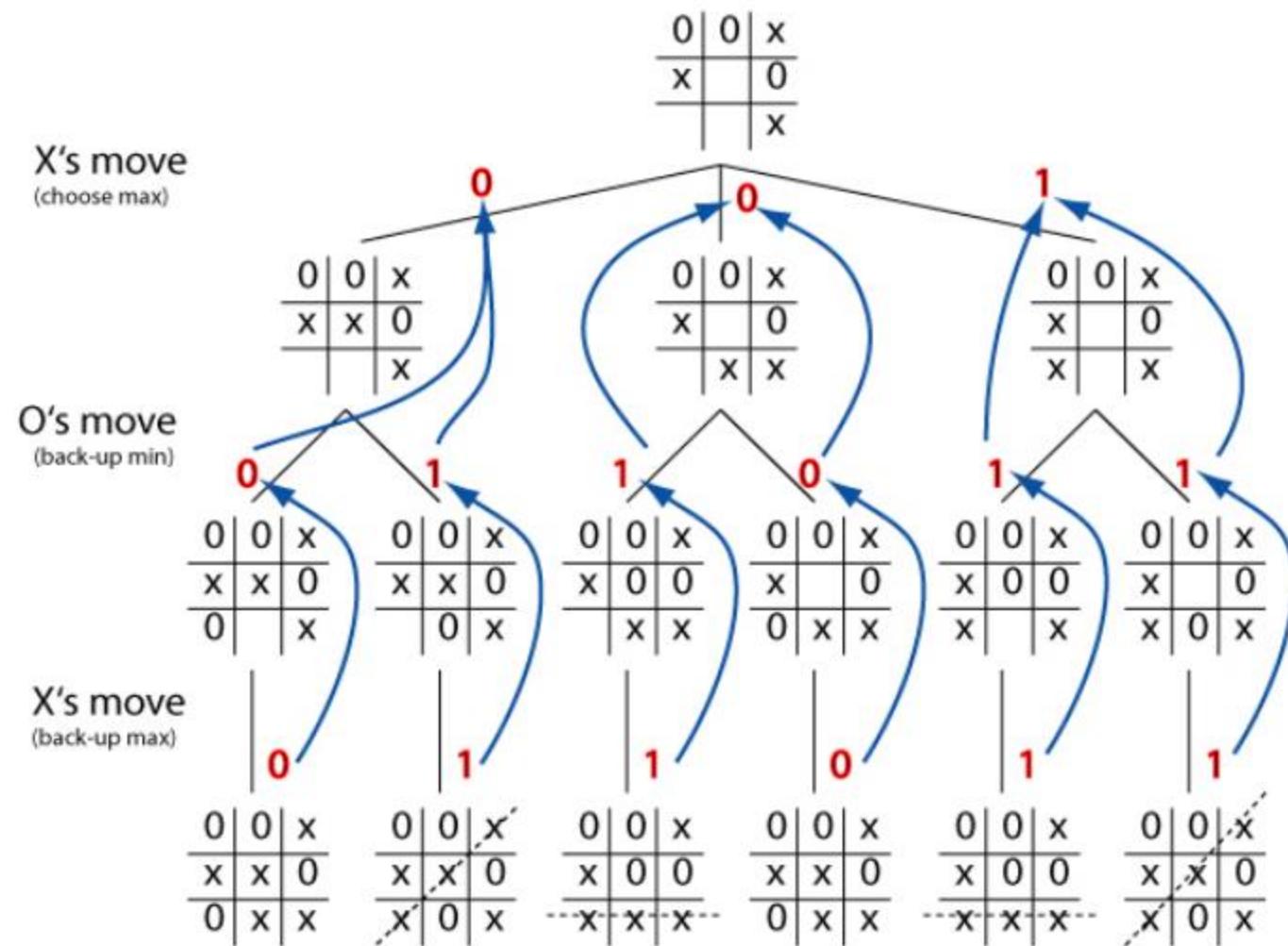
MIN

MAX

MIN

■ Up : Two Levels

Game playing with Mini-Max



MAX

MIN

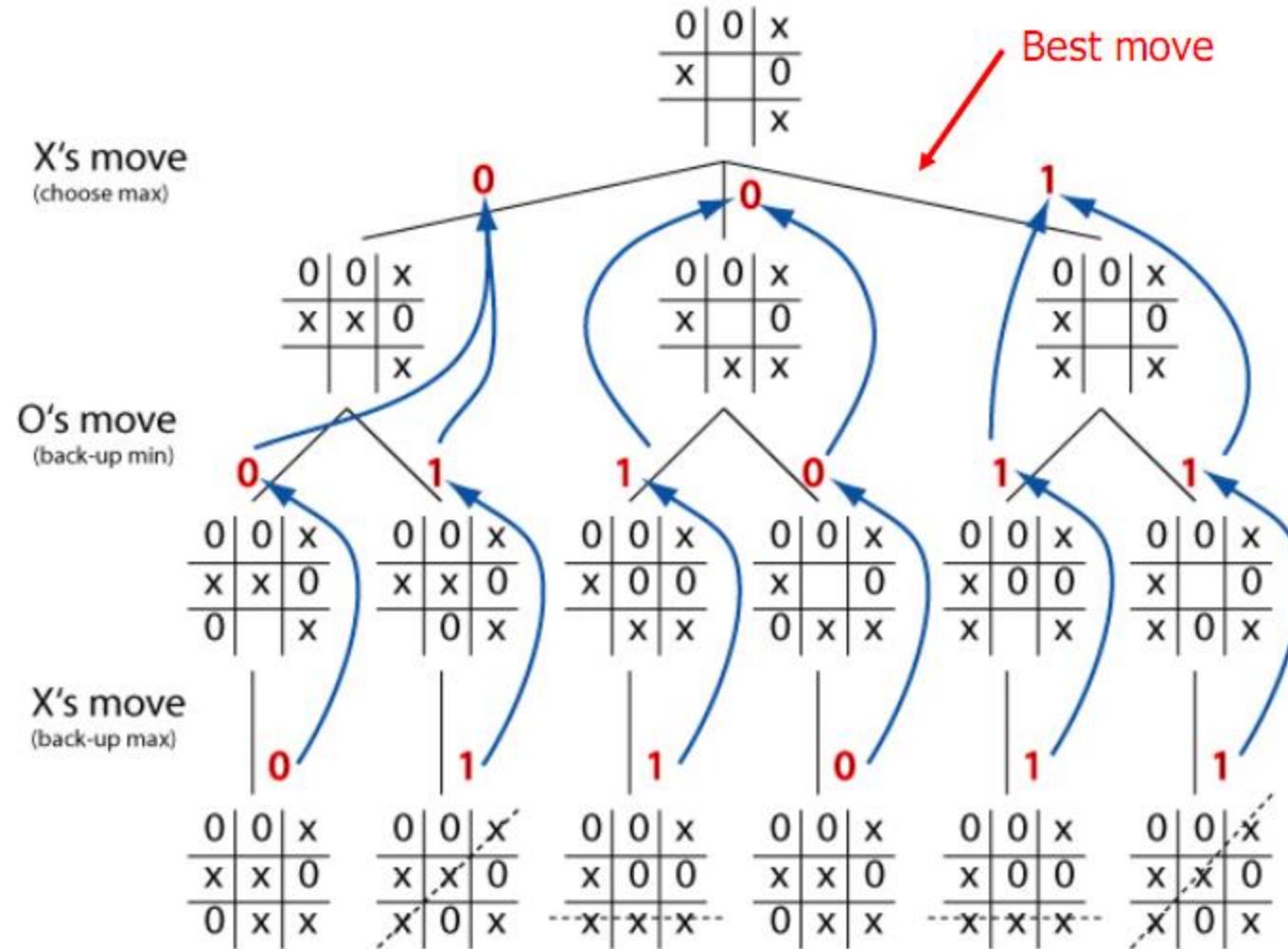
MAX

MIN

Evaluation obtained :

Game playing with Mini-Max

Choose best move which is maximum



MAX

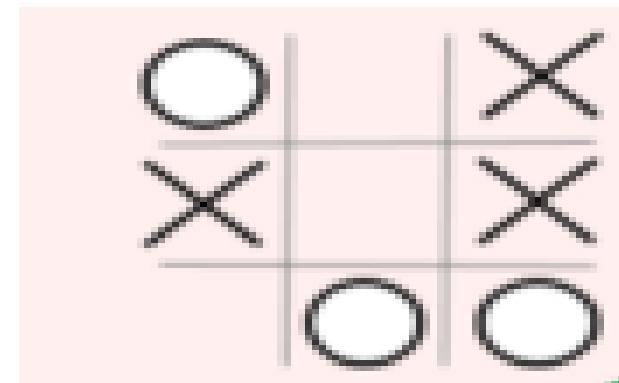
MIN

MAX

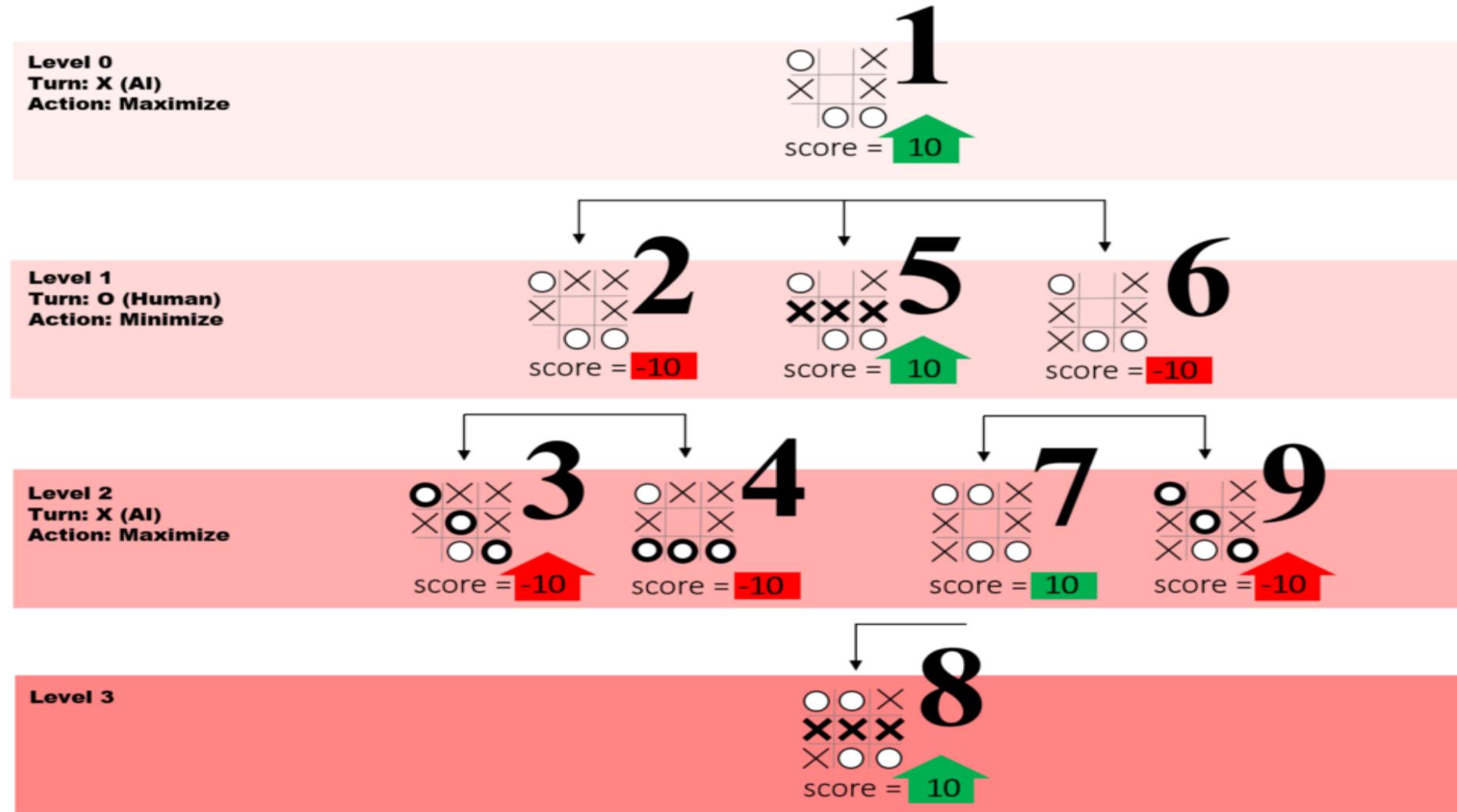
MIN

Challenge

- Let AI is the one who marks X and the human player marks O. Identify which step is to be taken further for the given state if AI player has next turn . Assume If O wins you should return -10, if X wins you should return +10.



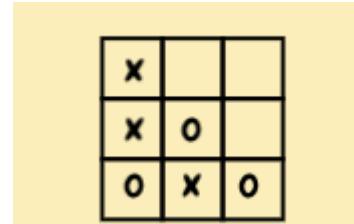
SOLUTION



SOLUTION

- Identify which step is to be taken further for the given state if X player has next turn .

Assume If X wins you should return +1, O wins returns -1 and wilbe 0 if it's a draw.



SOLUTION

X wins: +1

O wins: -1

Draw: 0

x		
x	o	
o	x	o

max

x	x	
x	o	
o	x	o

x		x
x	o	
o	x	o

x		
x	o	x
o	x	o

min

x	x	o
x	o	
o	x	o

-1

x	x	
x	o	o
o	x	o

+1

x	o	x
x	o	
o	x	o

0

x		x
x	o	o
o	x	o

+1

x	o	
x	o	x
o	x	o

0

x	o	
x	o	x
o	x	o

-1

x	x	x
x	o	o
o	x	o

+1

x	o	x
x	o	x
o	x	o

0

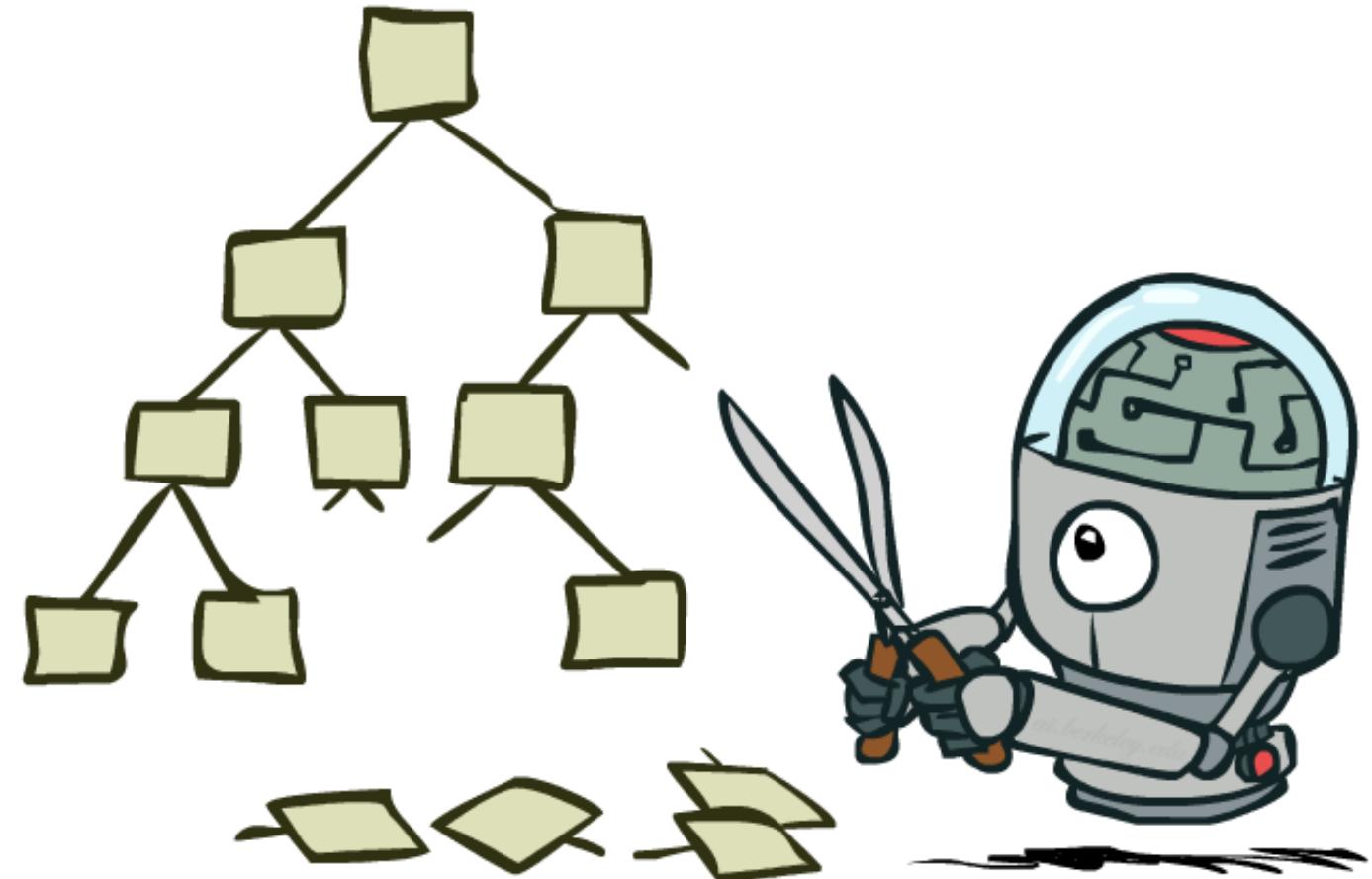
x	x	x
x	o	o
o	x	o

+1

x	o	x
x	o	x
o	x	o

0

GAME TREE PRUNING



MIN MAX CONCERN

- The number of game nodes to examine with minimax search is **exponential to number of moves**.
- Is it possible to **compute the correct minimax decision without looking at every node** in the game tree?

SOLUTION

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an **optimization technique for the minimax algorithm**.
- Same as that of minmax but **prune** away branches that cannot **possibly influence the final decision**.

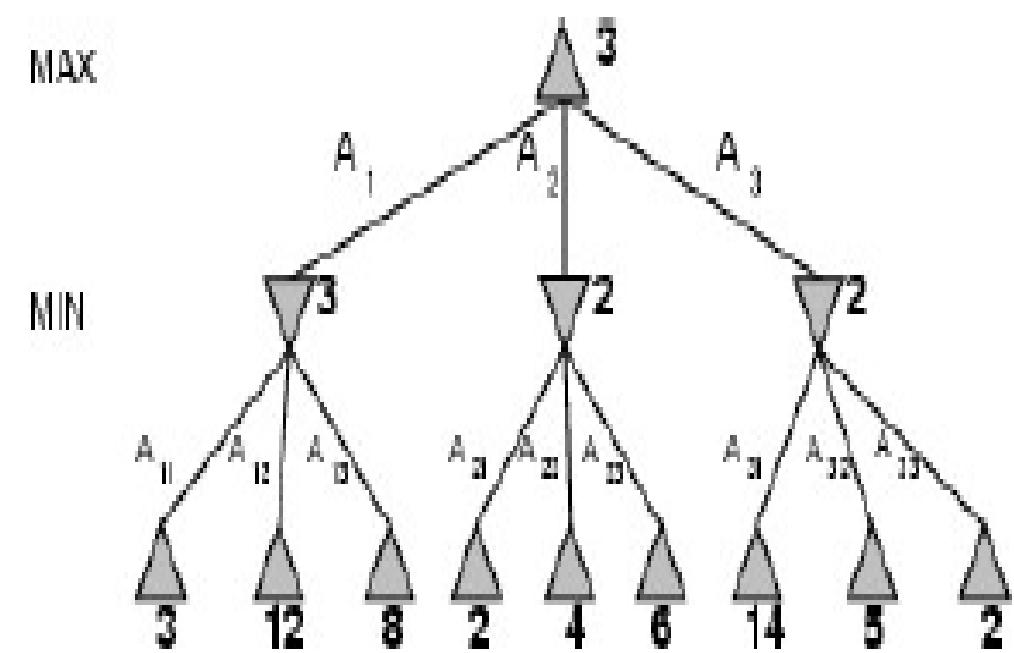
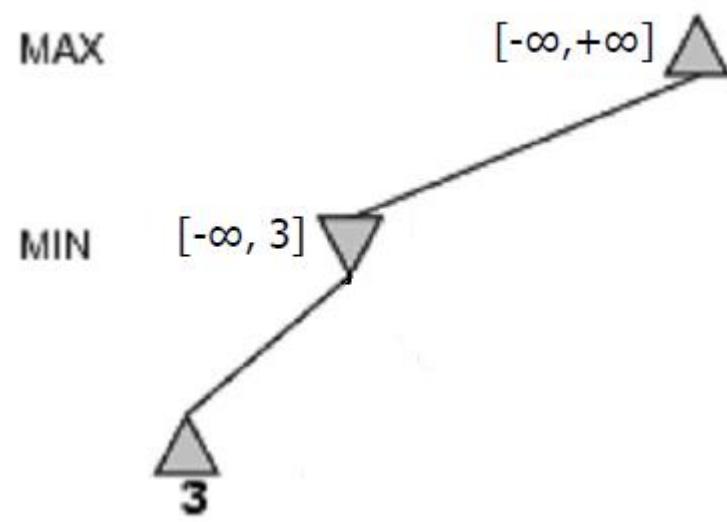
DEFINITION OF APLPHA BETA

- α = the value of the **best (highest-value) choice we have found so far** at any choice point along the path for **MAX**. The initial value of alpha is $-\infty$.
- β = the value of the **best (lowest-value) choice we have found so far** at any choice point along the path for **MIN**. The initial value of beta is $+\infty$.
- If v is worse than α , max will avoid it-->prune that branch
- **α - β Search updates** the values of α and β as it goes along and **prunes** the remaining branches at a node as soon as the value of the current node is worse than the current α or β for MAX or MIN respectively

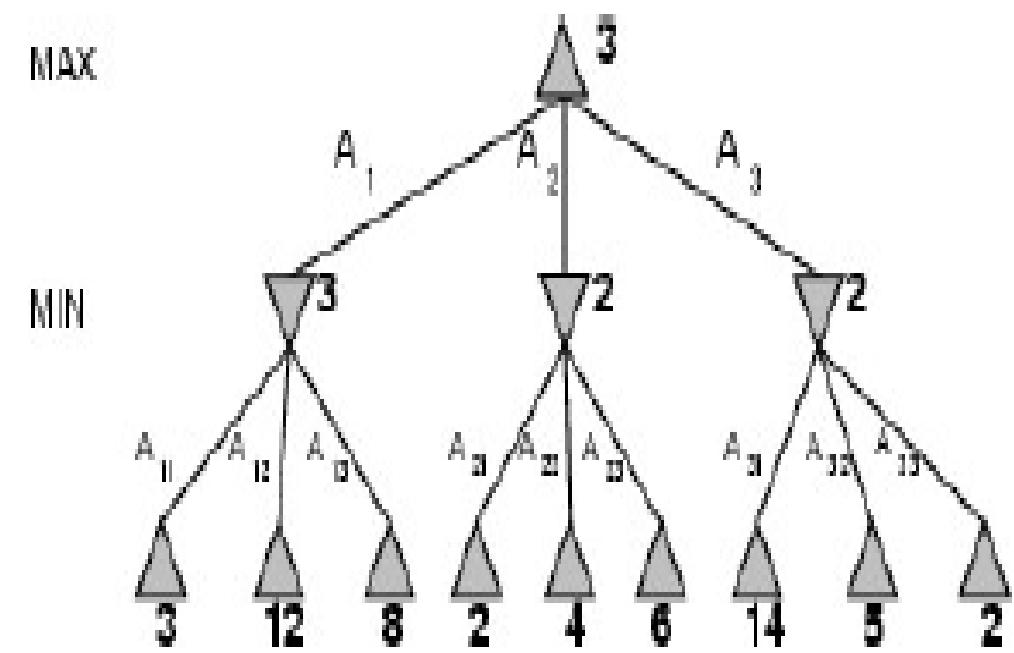
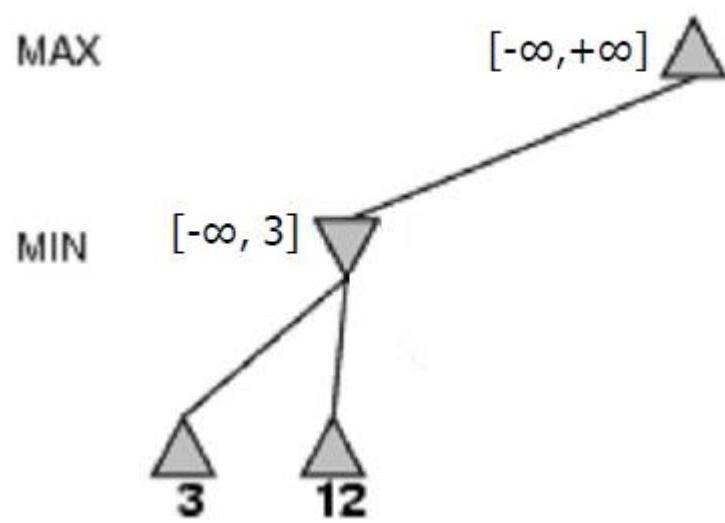
ALPHA BETA PRUNING

- **α - β pruning Search** cannot eliminate the exponent, but we can cut it to half. It cuts off search cost by exploring less no of nodes
- The **effectiveness** of alpha-beta pruning is highly dependent on the **order in which the successors are examined**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- Always $\alpha \geq \beta$

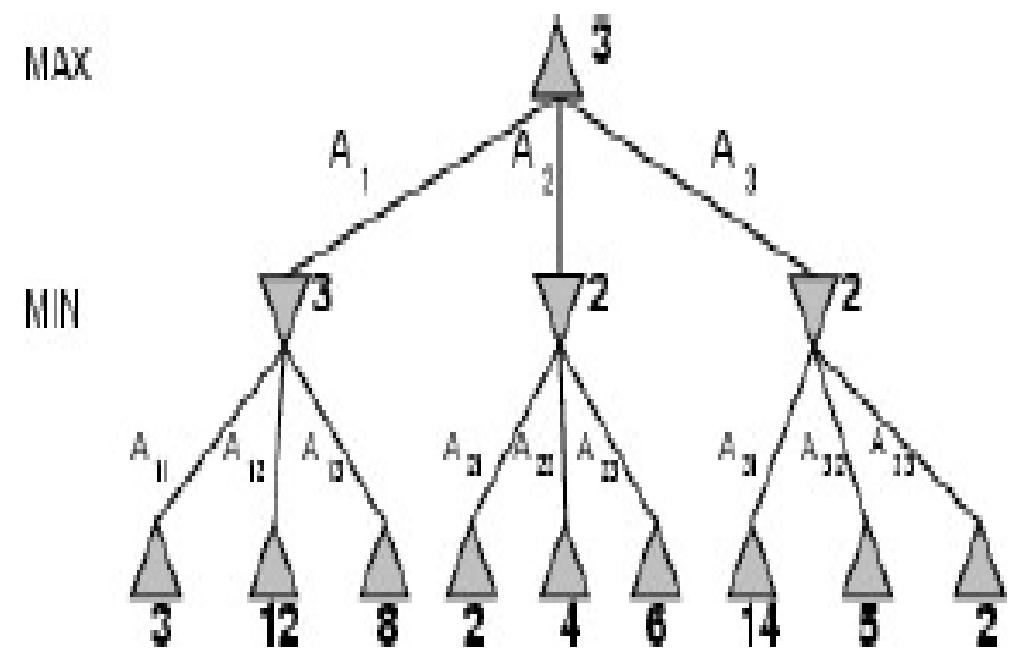
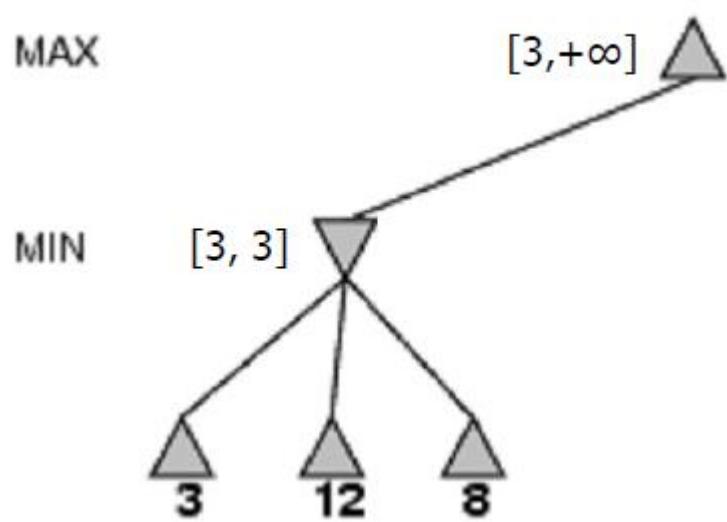
EXAMPLE



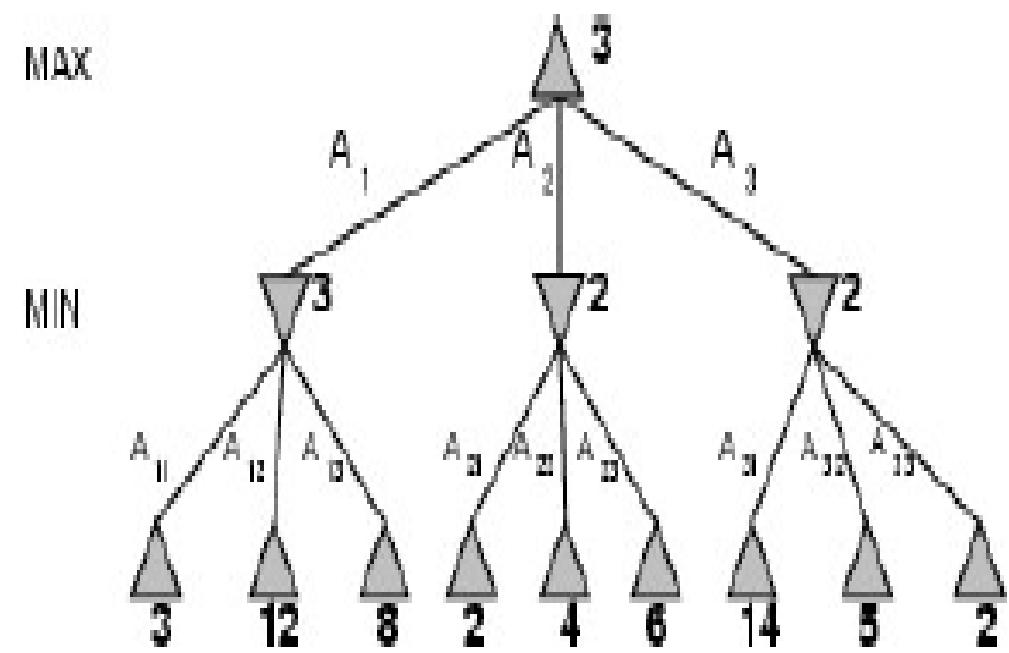
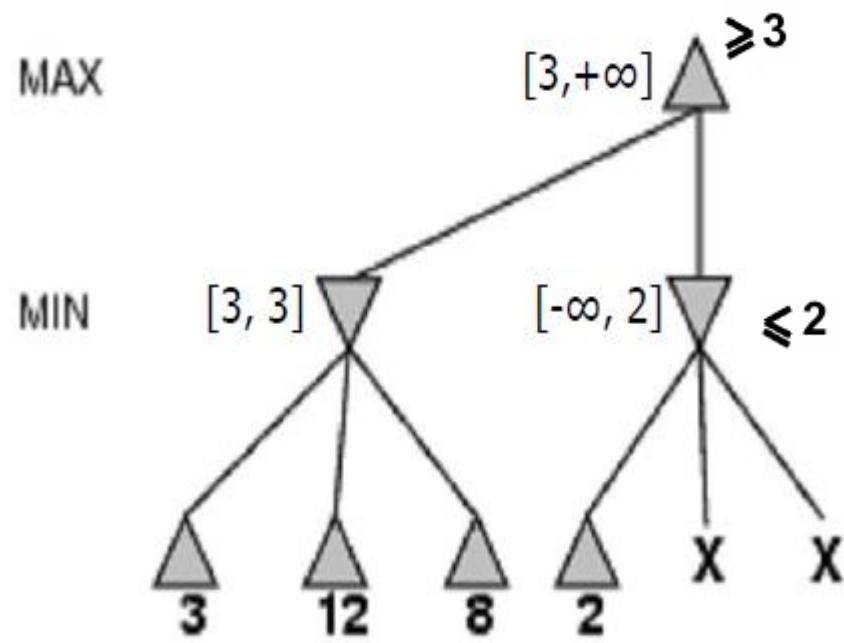
EXAMPLE



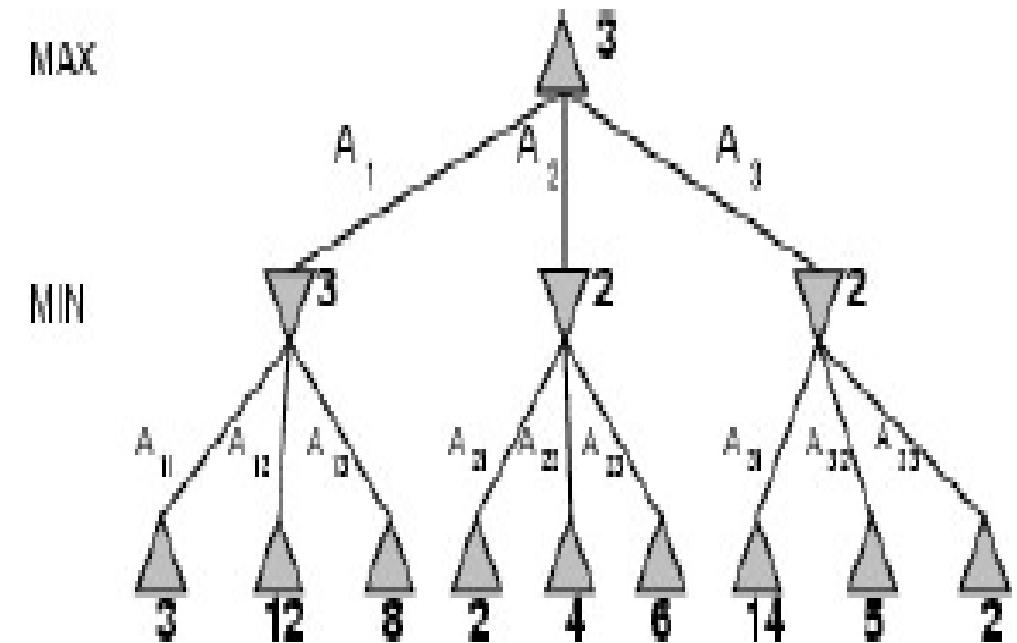
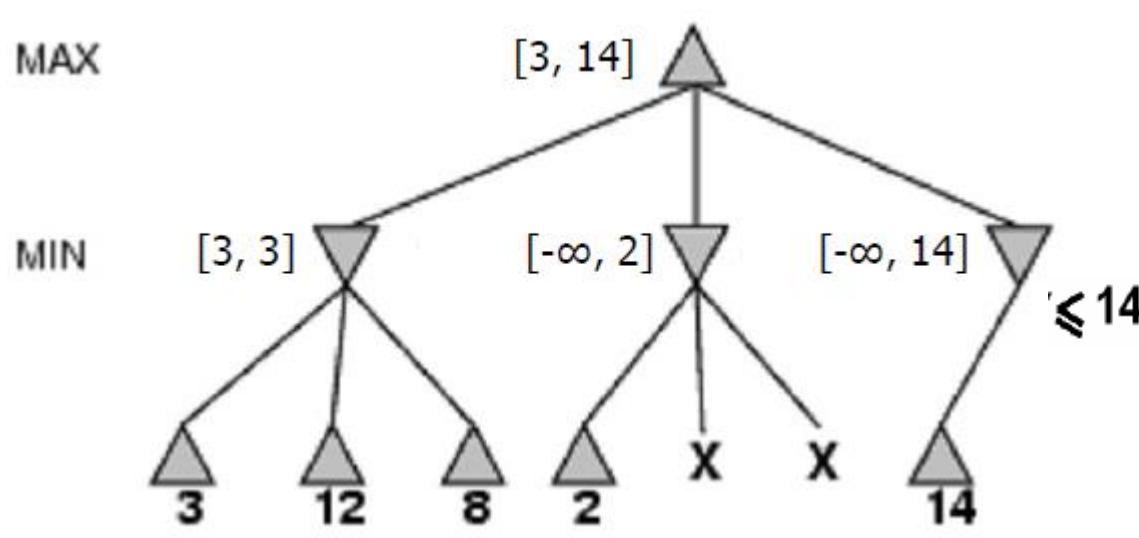
EXAMPLE



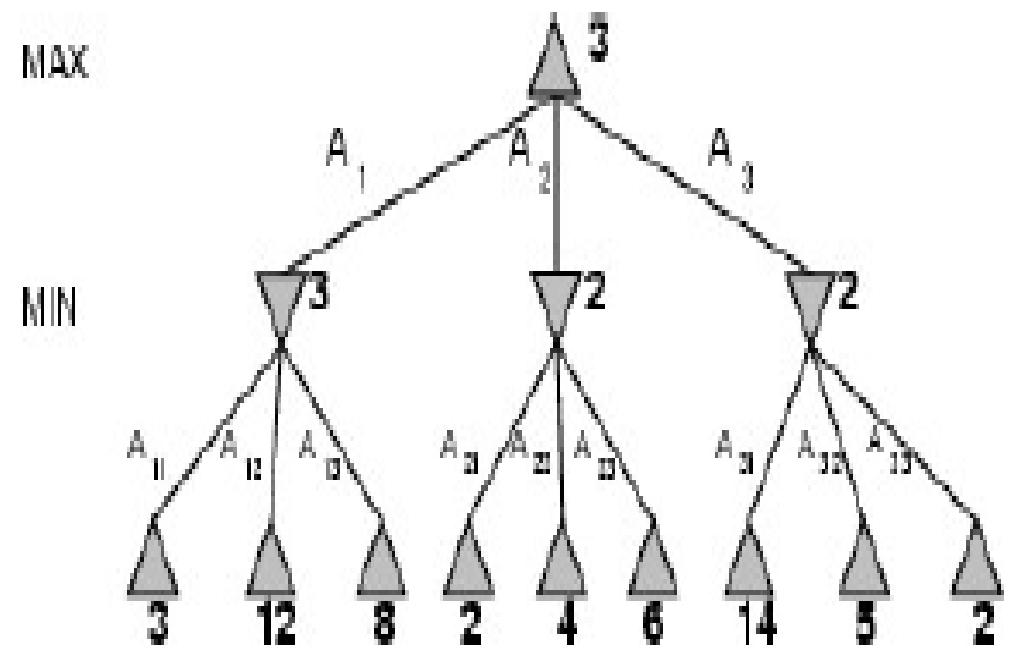
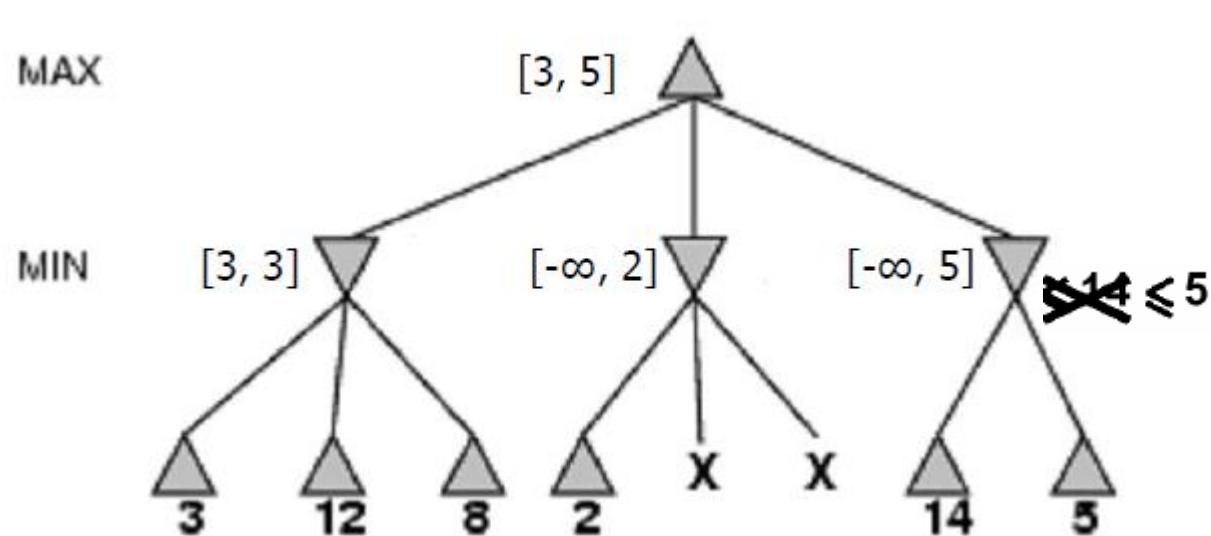
EXAMPLE



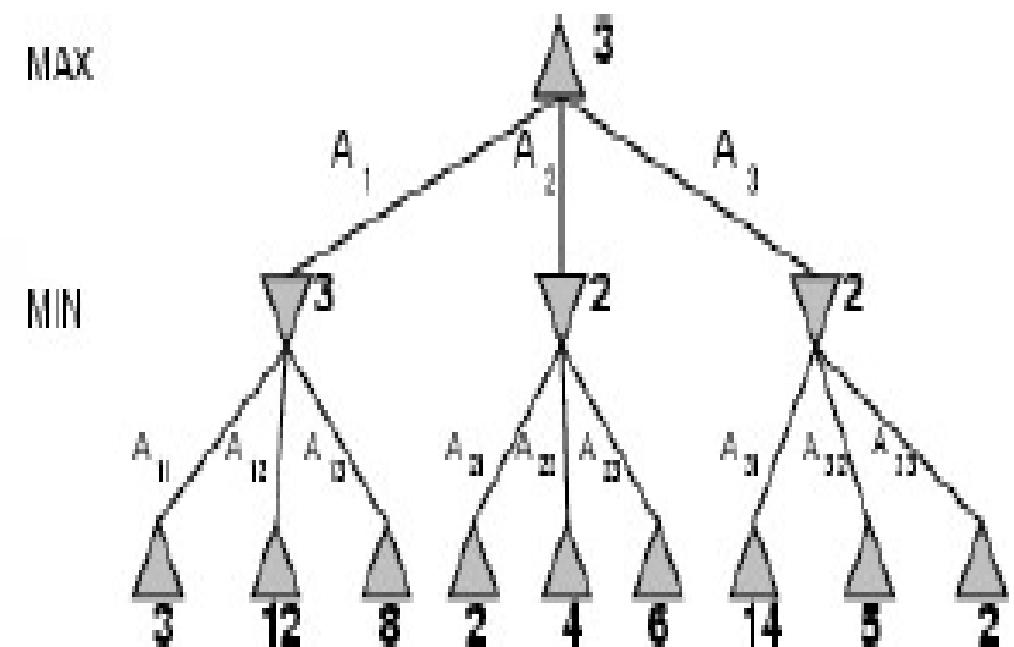
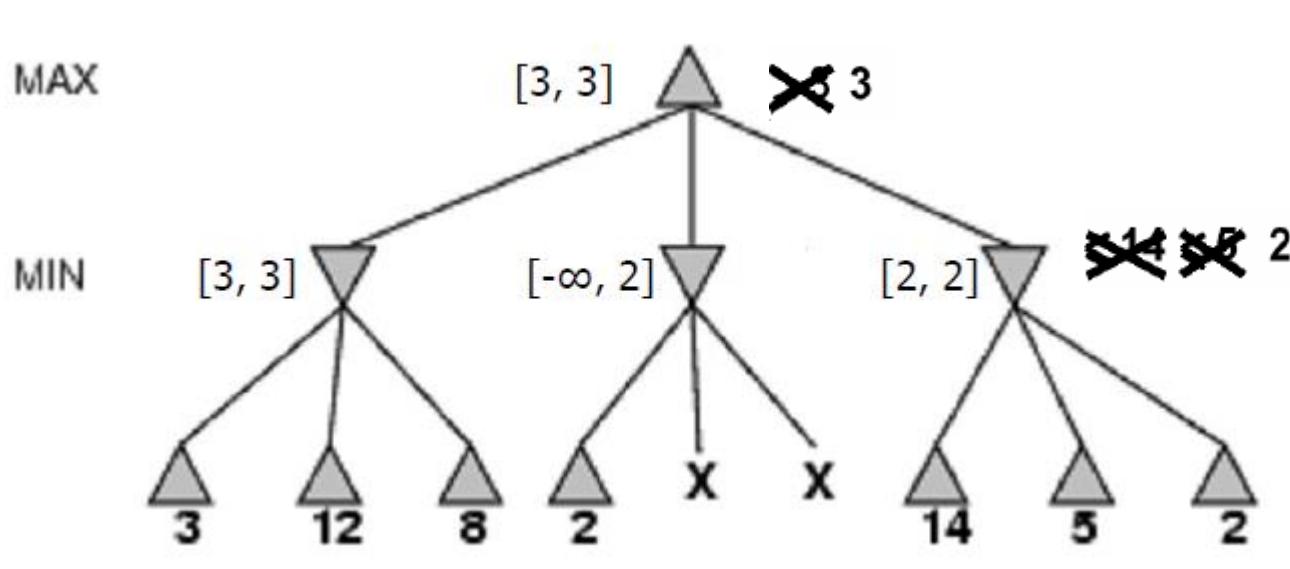
EXAMPLE



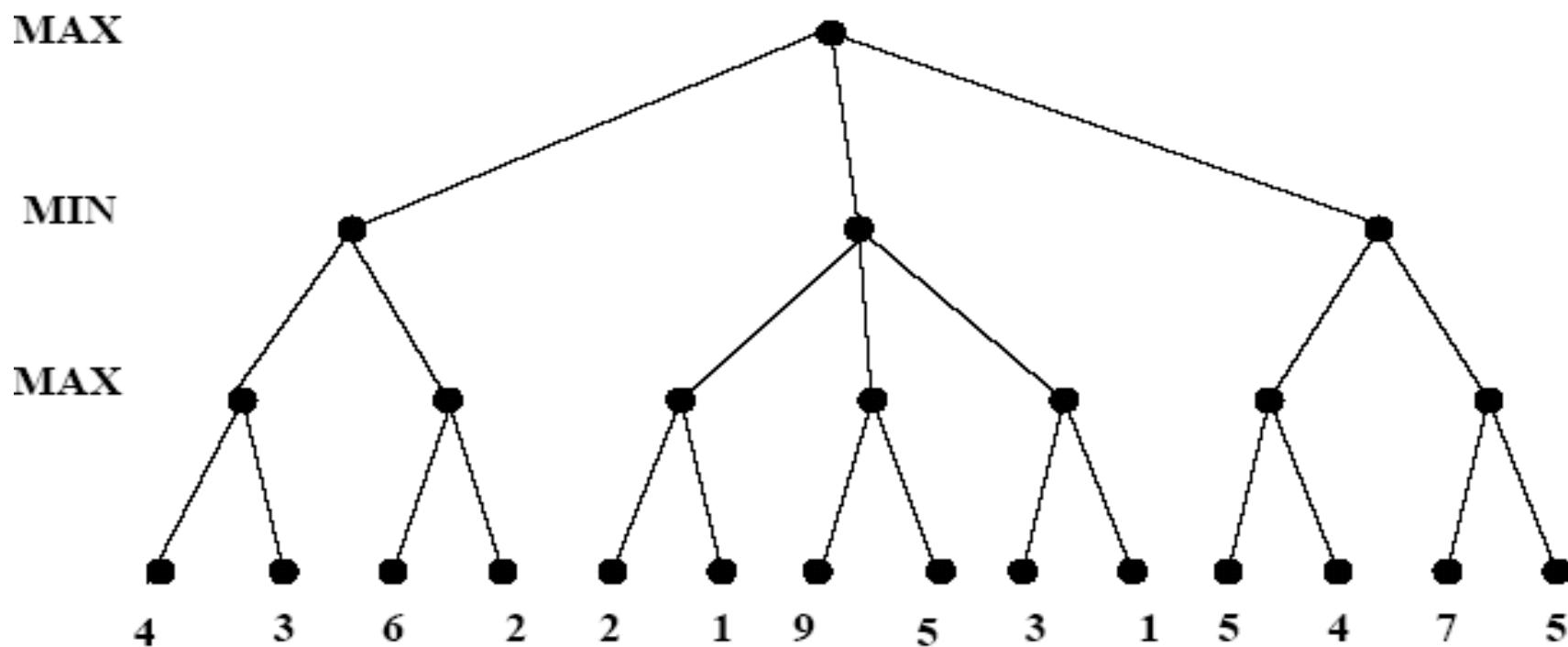
EXAMPLE



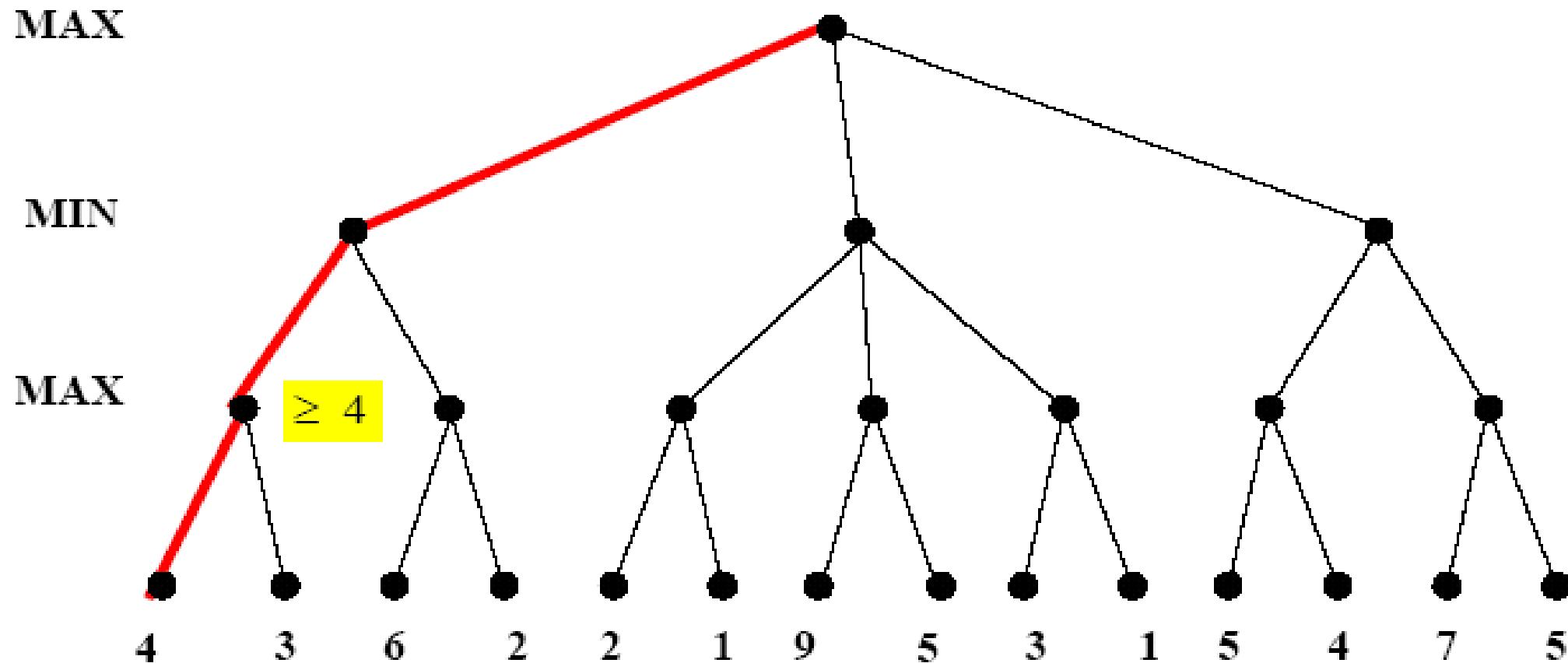
EXAMPLE



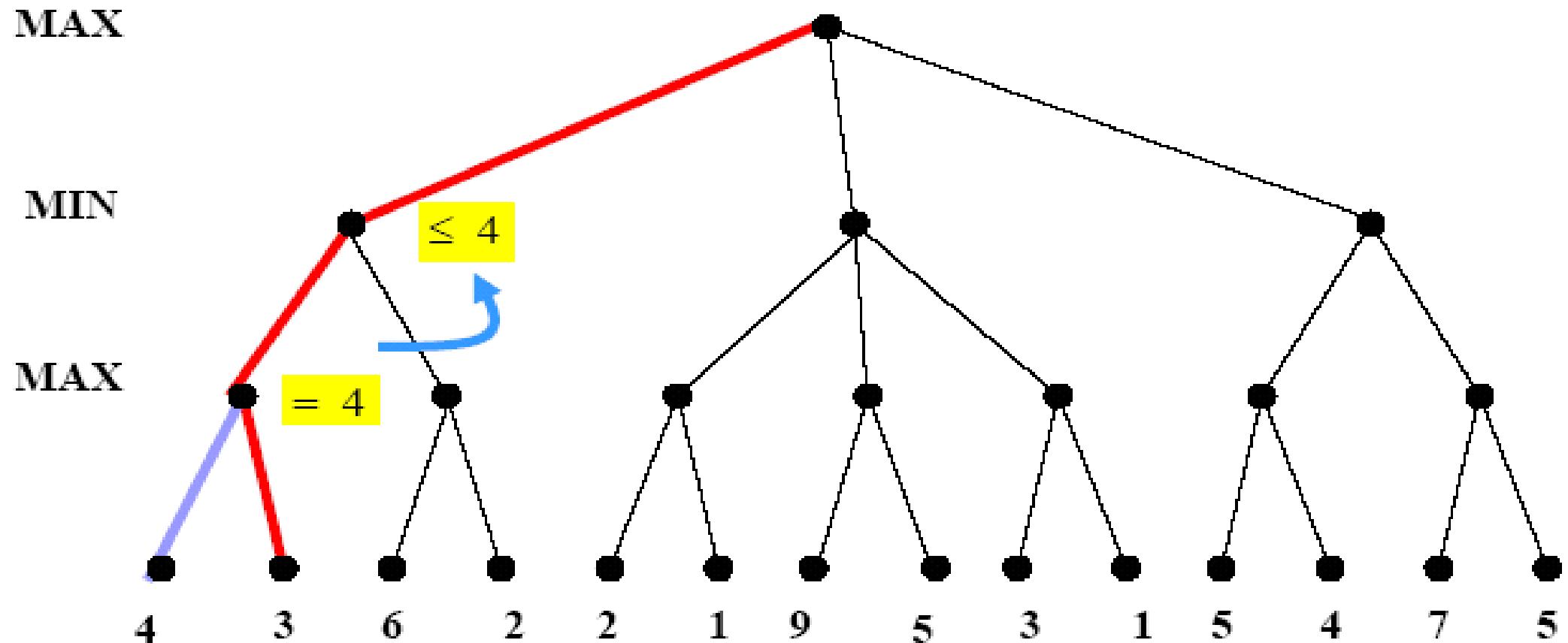
Alpha beta pruning. Example



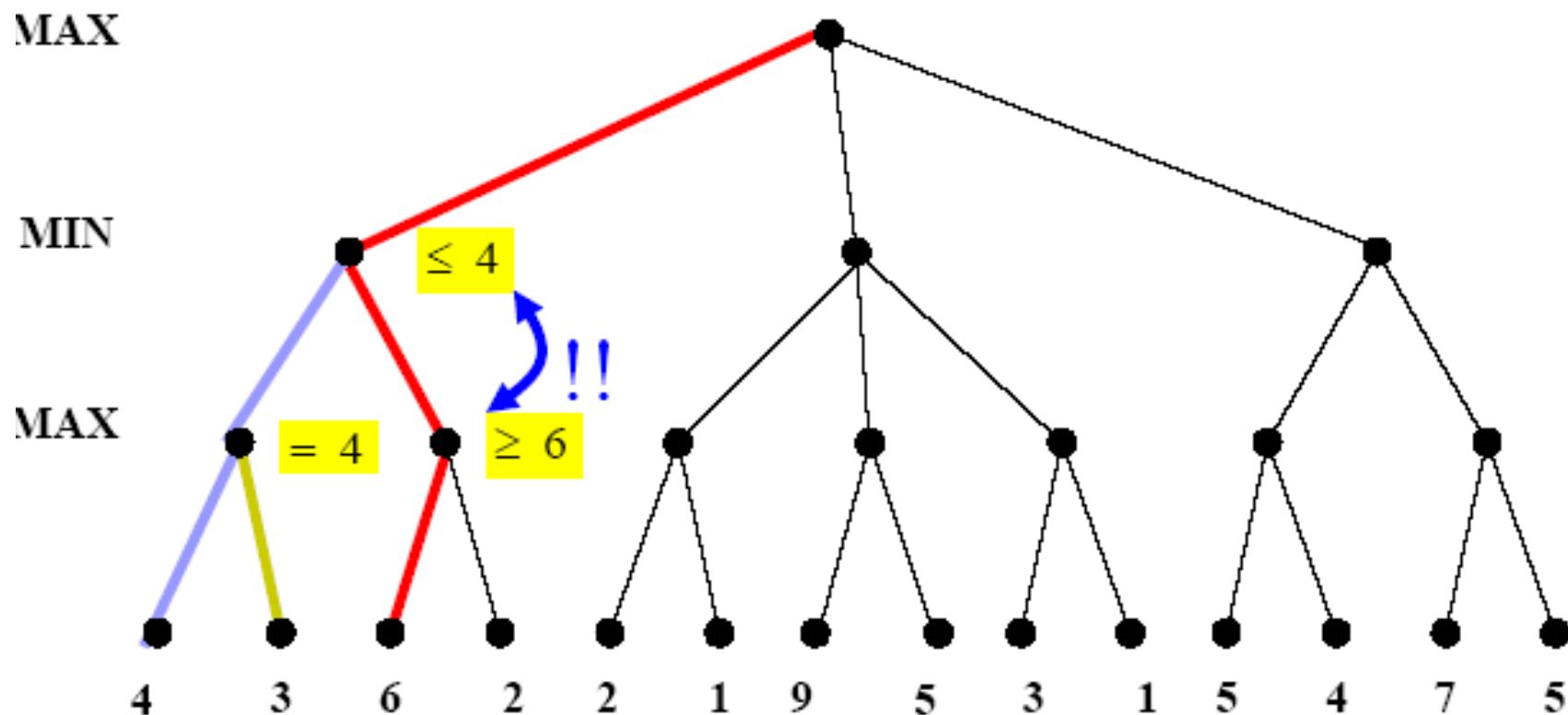
Alpha beta pruning. Example



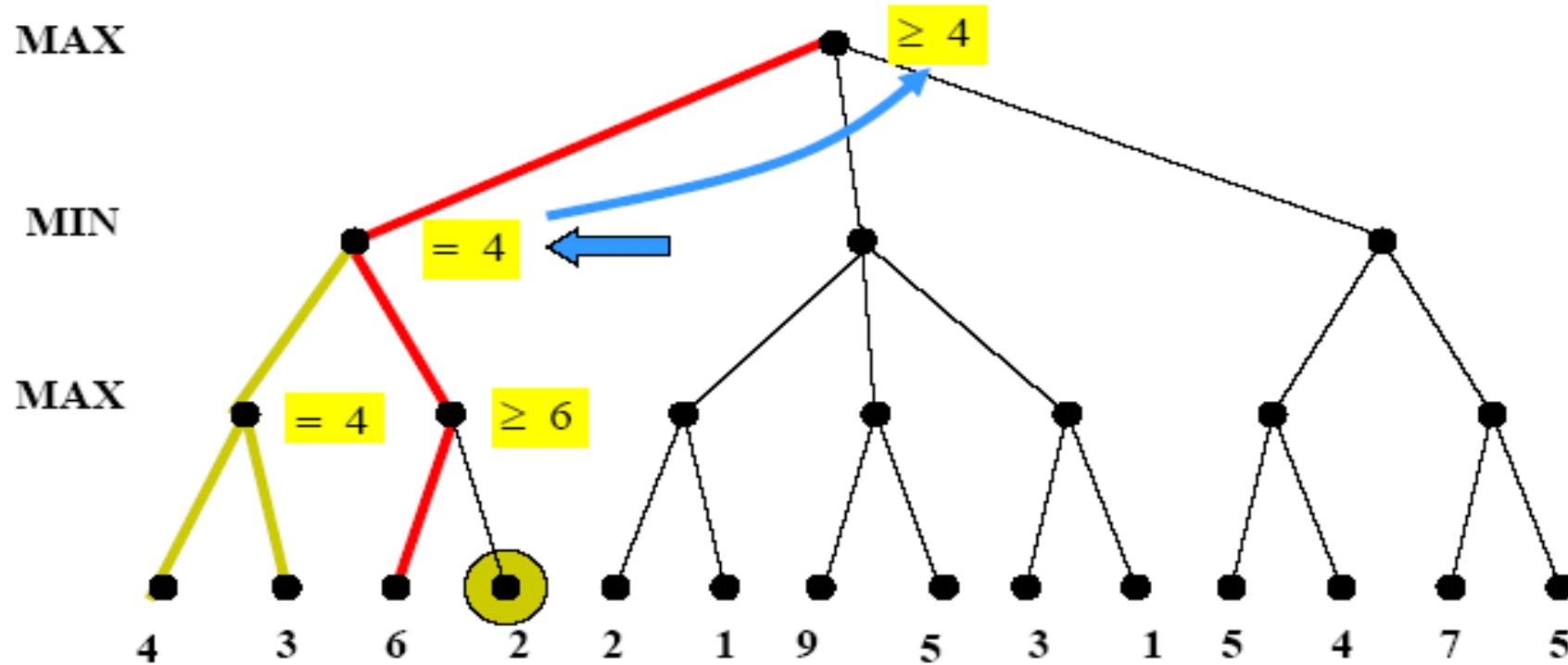
Alpha beta pruning. Example



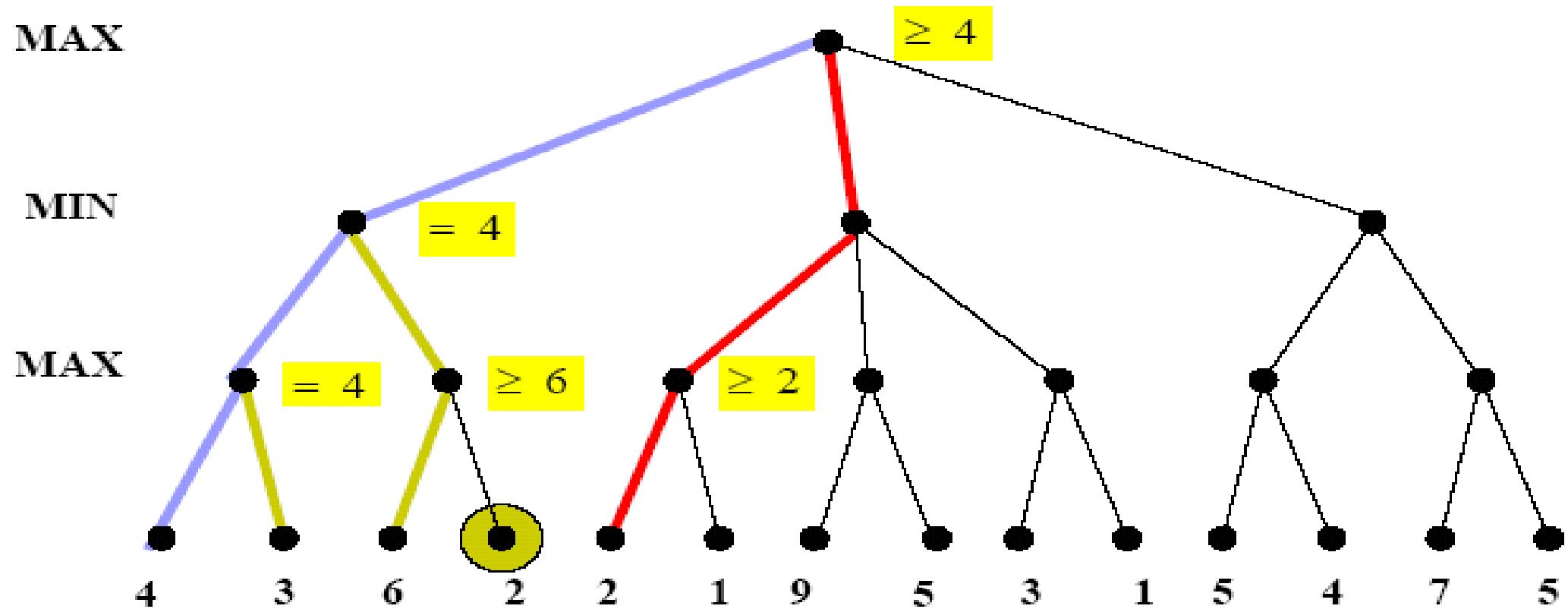
Alpha beta pruning. Example



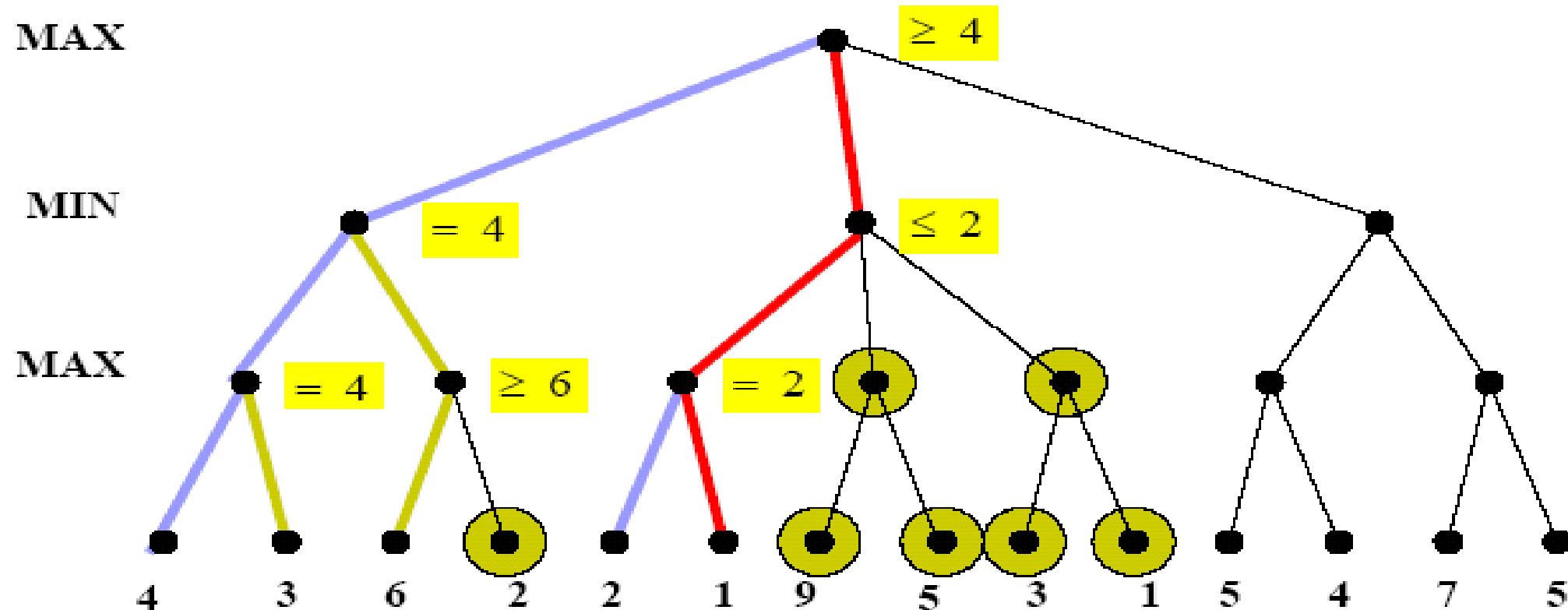
Alpha beta pruning. Example



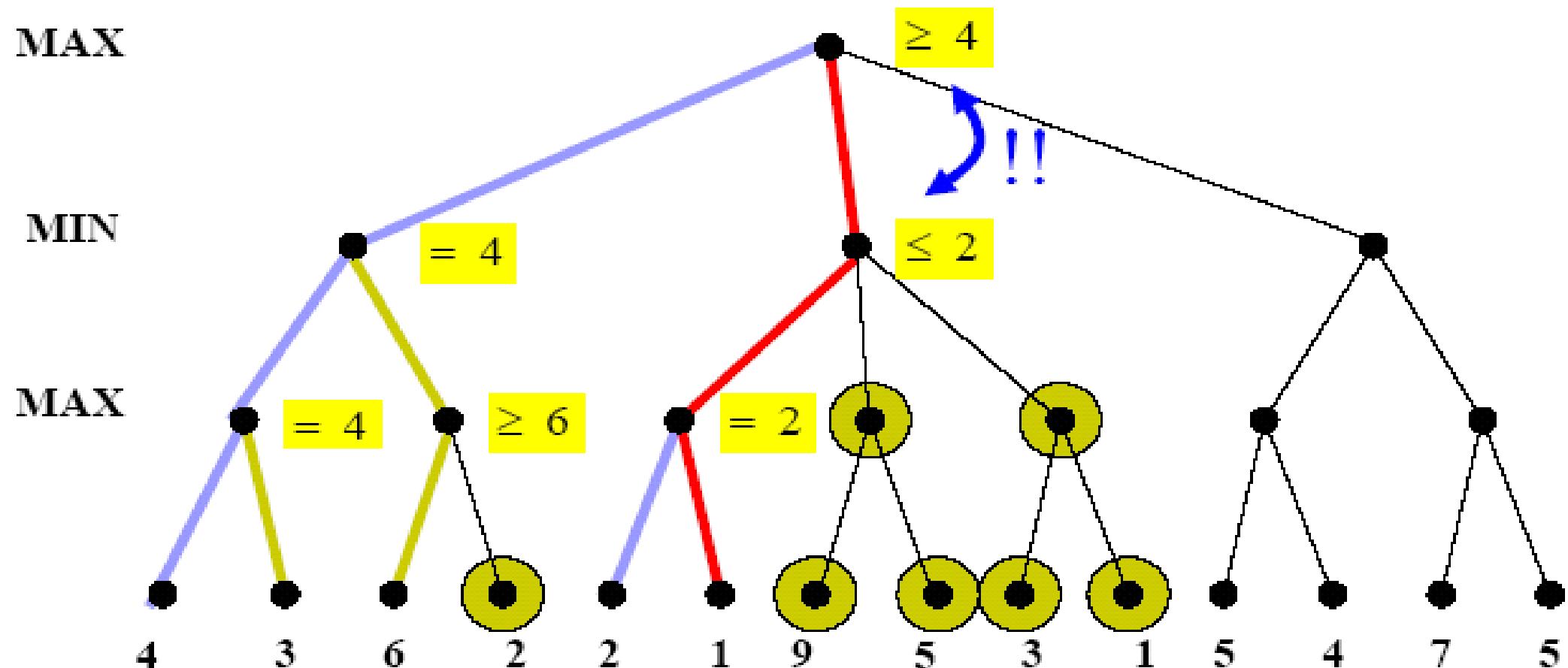
Alpha beta pruning. Example



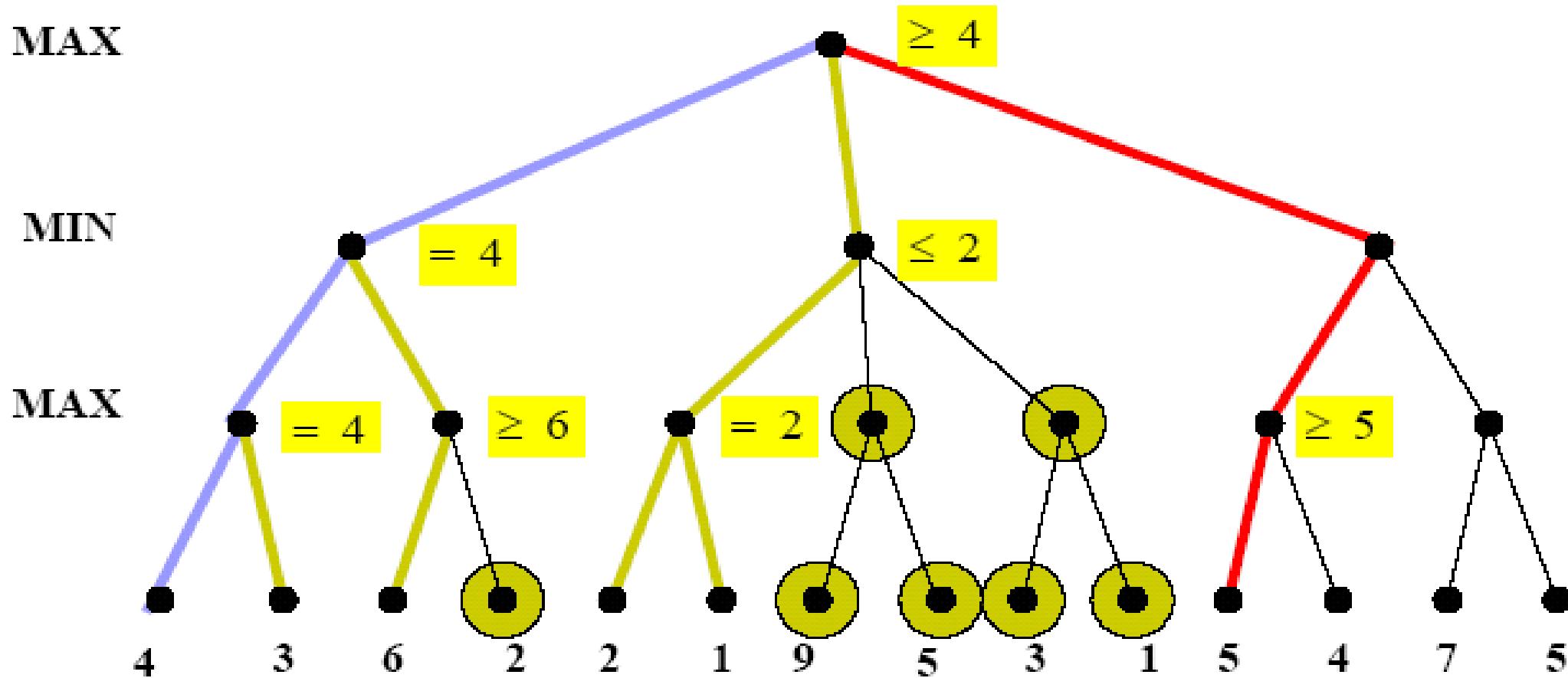
Alpha beta pruning. Example



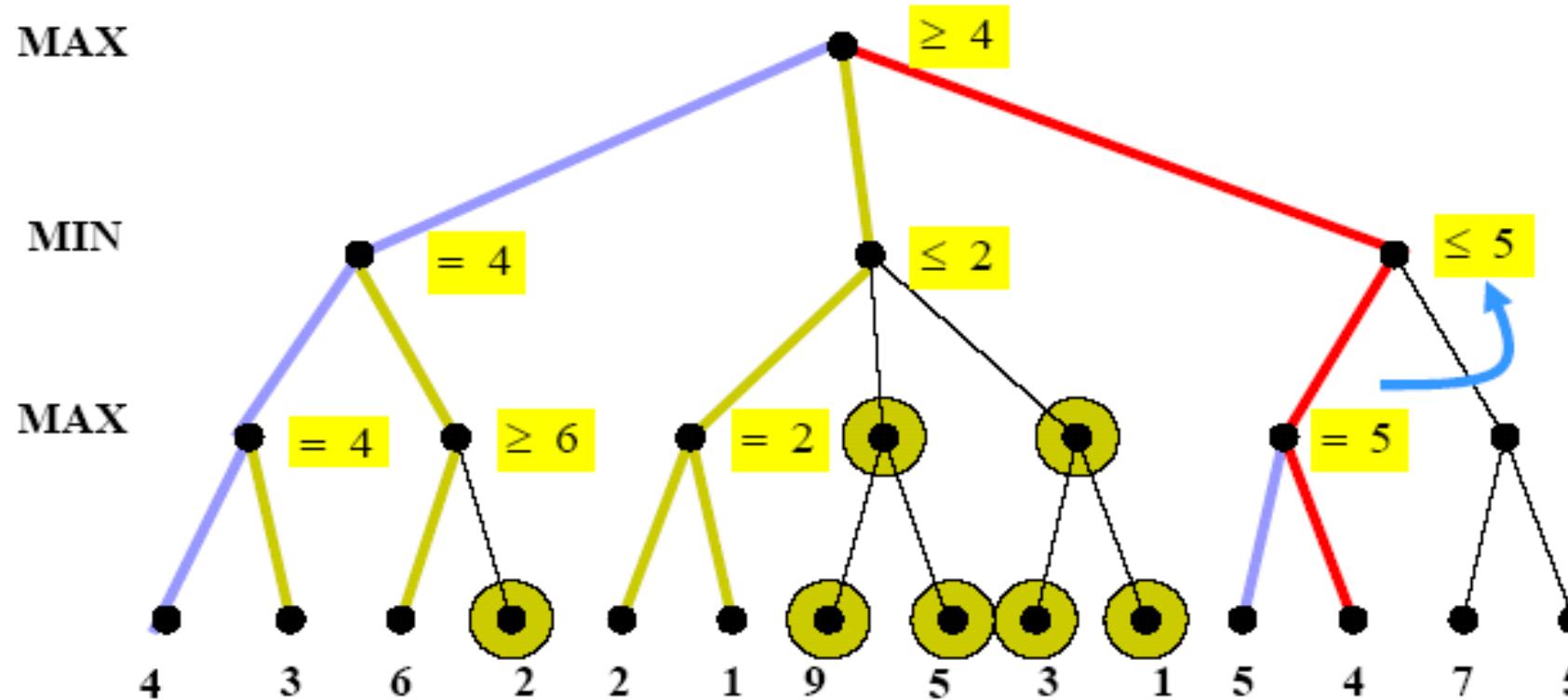
Alpha beta pruning. Example



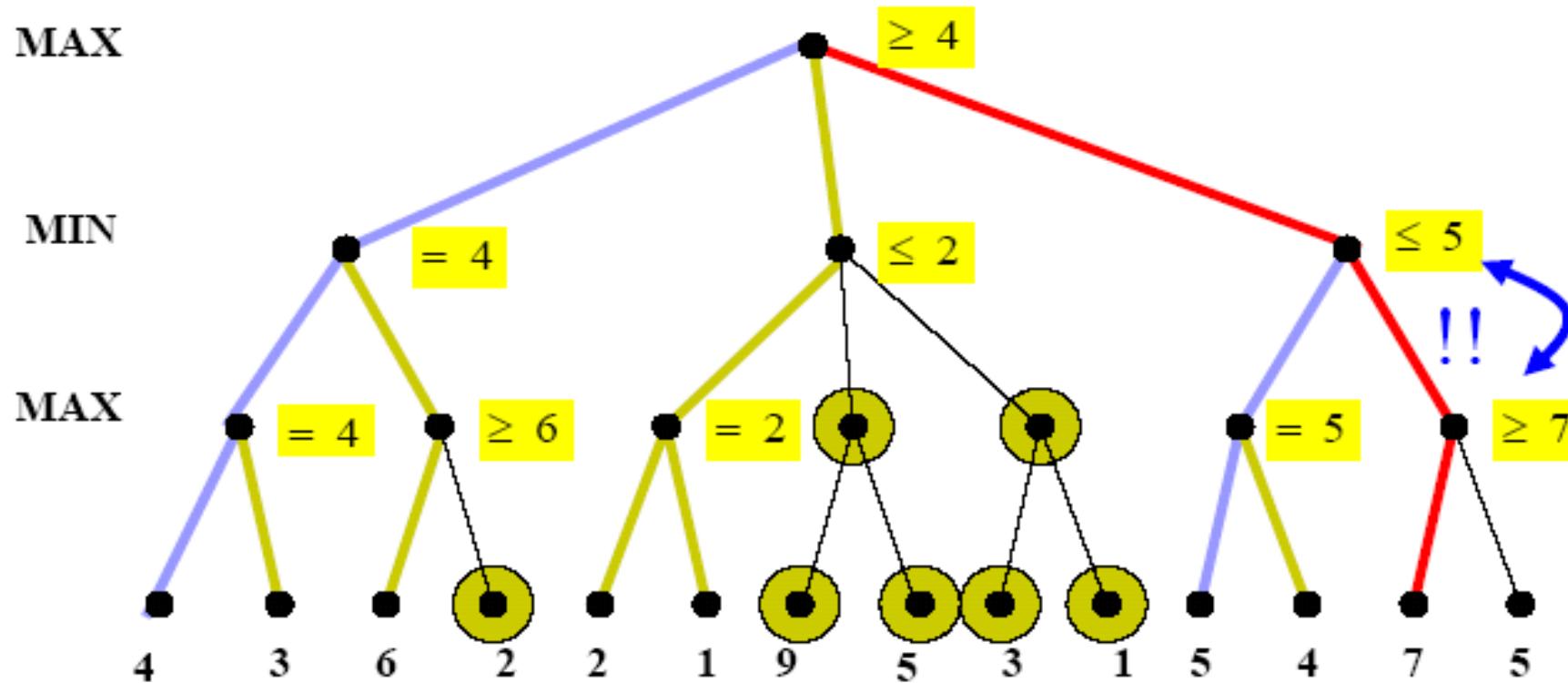
Alpha beta pruning. Example



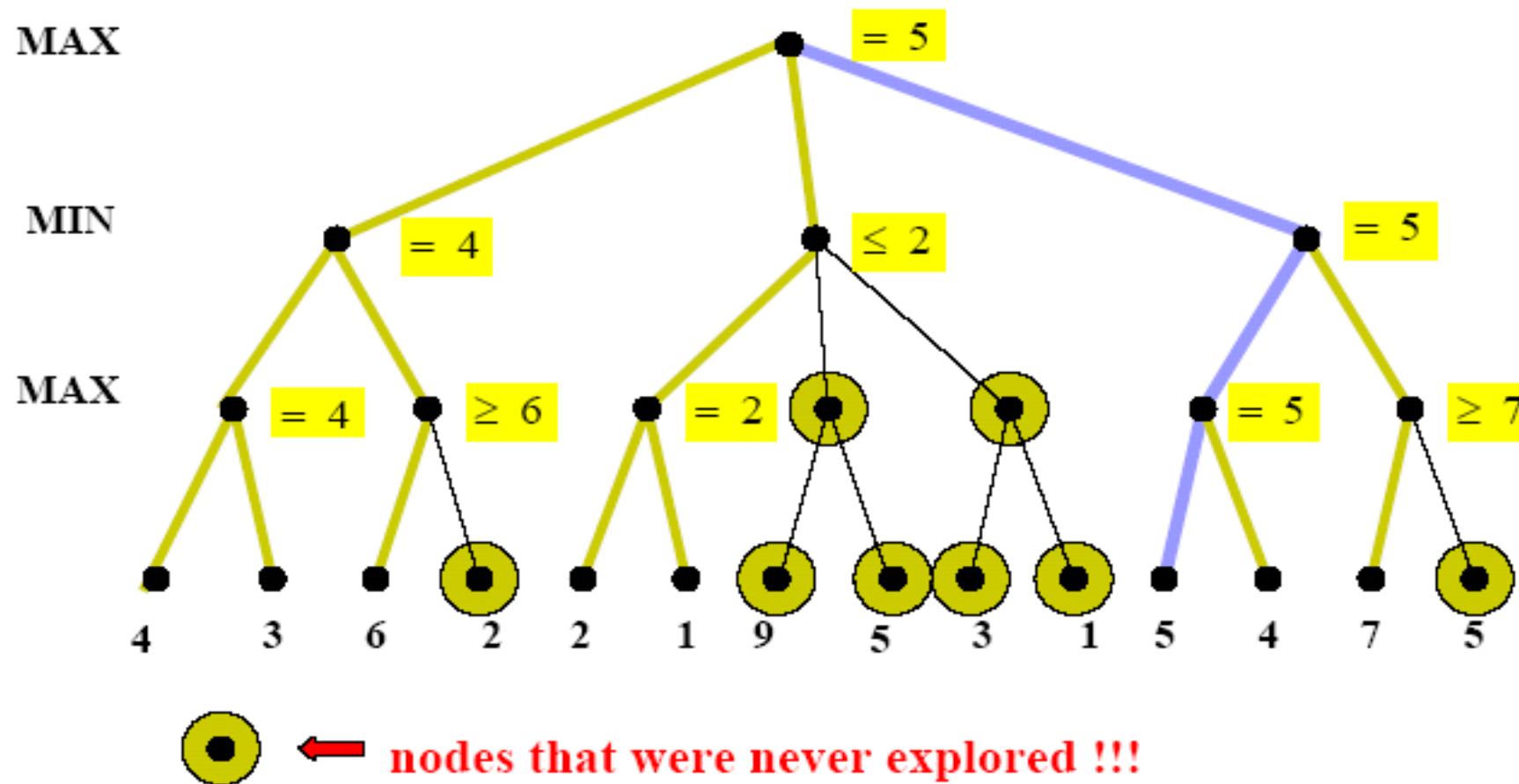
Alpha beta pruning. Example



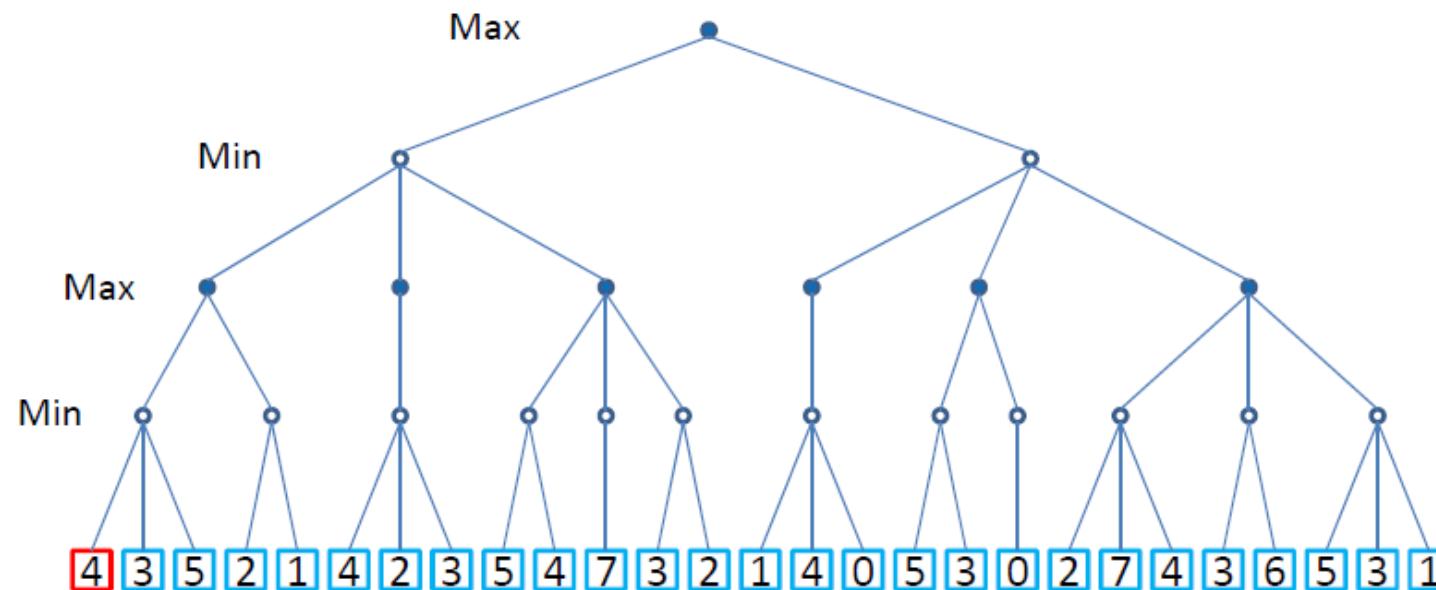
Alpha beta pruning. Example



Alpha beta pruning. Example

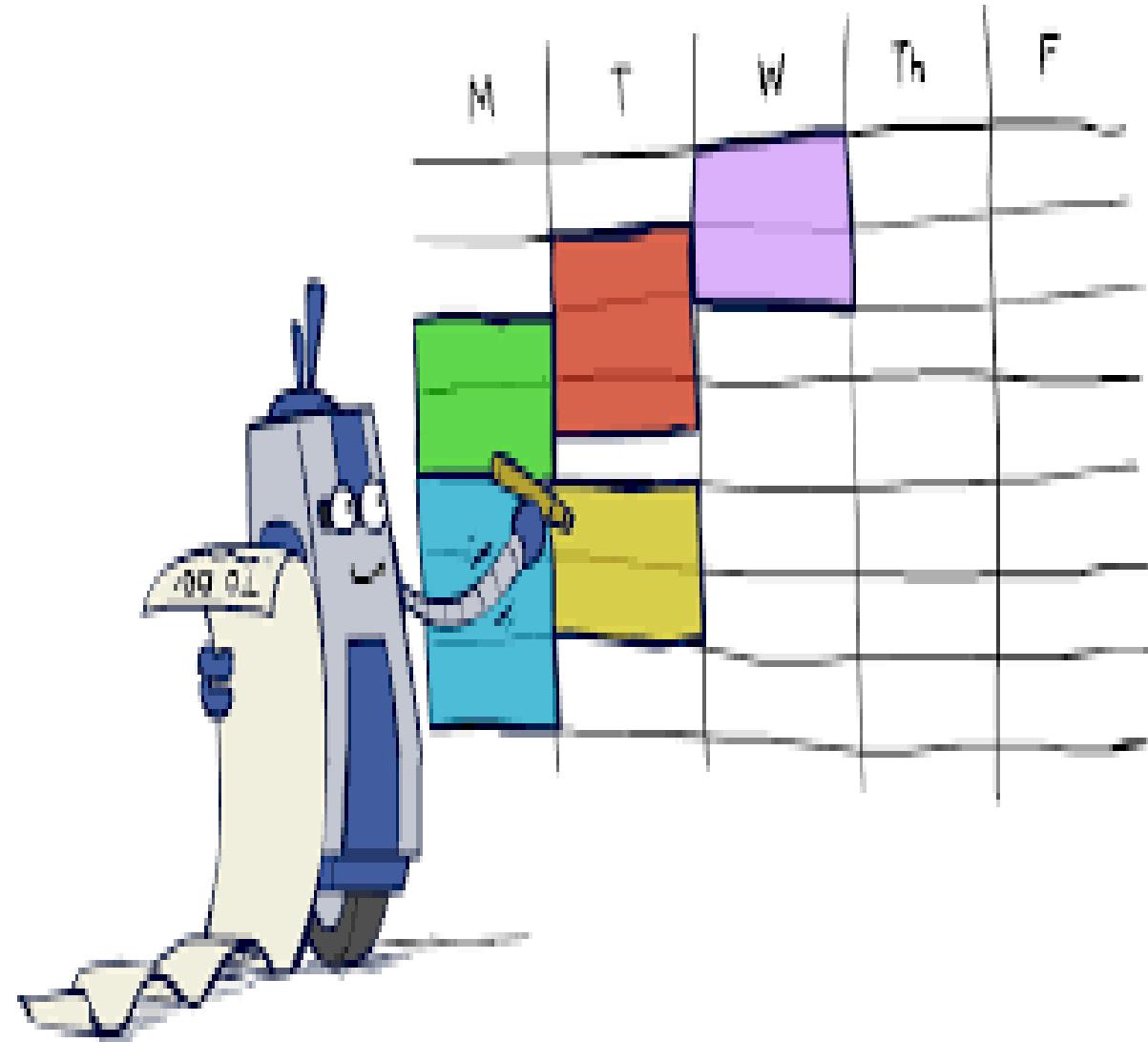


CHALLENGE



CONSTRAINT SATISFACTION PROBLEM

CSP is a problem consisting of finite set of variables, which are associated with finite domain, and constraints, which restrict the values that the variables can simultaneously take. The challenge is to assign a value to each variable satisfying all the constraints.



CONSTRAINT SATISFACTION PROBLEM COMPONENTS

- CSPs represented using 3 components **X**, **D** and **C**:
 1. **X** is a set of **variables** $\{X_1, X_2, \dots, X_n\}$ and a fixed values for each X_i represent a **State**.
 2. Each variable X_i has a nonempty **domain** $\{D_1, D_2, \dots, D_n\}$ of **possible values** where $D_i = \{v_1, \dots, v_k\}$ set of allowable values. Each **constraint** C_i limits the values that variables can take.
 3. **C** is a set of **constraints** that specify set of allowable values for variable $V_i \{C_1, C_2, \dots, C_m\}$. **Goal Test** defined by set of constraints. Each constraint C_i is mentioned as a pair $\langle \text{scope}, \text{rel} \rangle$ where **scope** is set of variables that participate in constraint definition and **relation** defines values that variables can take.
 - Each **state** in a **CSP** is an **assignment of values** to some or all **variables**, $\{X_i = V_i, X_j = V_j, \dots\}$. CSP search algorithms take advantage of this structure of states to solve complex problems.
-

CSP SOLUTION

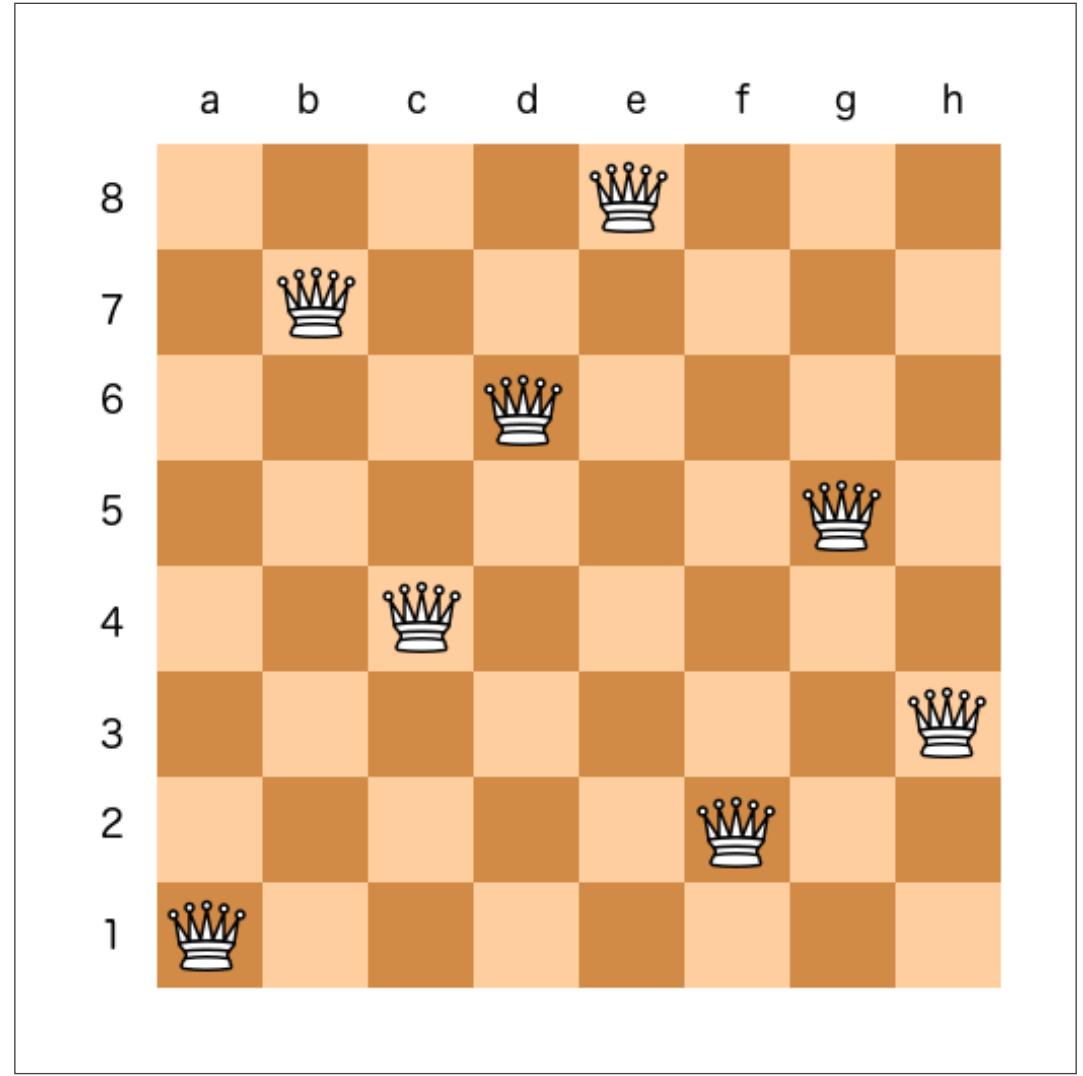
- *Consistent assignment or Legal Assignment:* An assignment does not violate any constraints.
 - *Complete Assignment:* An assignment in which **every variable is assigned a value.**
 - *Partial Assignment:* An assignment in which some variables have no values.
 - A **solution** to a CSP is a **complete assignment** that satisfies all constraints ie **consistent assignment**.
 - Some CSPs require an **objective function** that **maximizes** or **minimizes** solution.
-

CSP TO BE FORMULATED AS A STANDARD SEARCH PROBLEM

- A CSP can easily be expressed as a standard search problem.
- GOAL is to find a CONSISTENT ASSIGNMENT (if one exists)
 - *Initial State*: the empty assignment {}.
 - *Operators*: Assign value to unassigned variable provided that there is no conflict.
 - *Goal test*: assignment consistent and complete.
 - *Path cost*: constant cost for every step.
 - Solution is found at depth n , for n variables
 - Hence depth first search can be used

Popular Problems with CSP

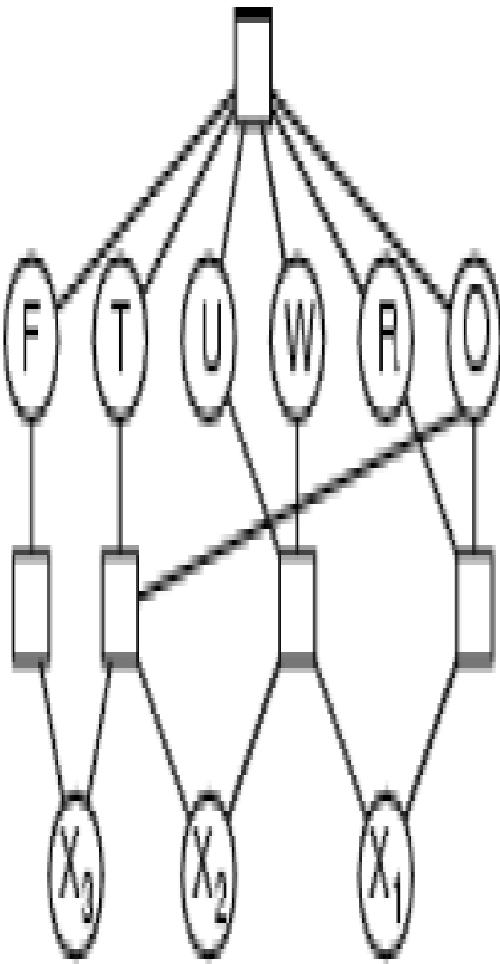
- **CryptArithmetic** (Coding alphabets to numbers.)
- **n-Queen** (In an n-queen problem, n queens should be placed in an nXn matrix such that no queen shares the same row, column or diagonal.)
- **Map Coloring** (coloring different regions of map, ensuring no adjacent regions have the same color)
- **Crossword** (everyday puzzles appearing in newspapers)
- **Sudoku** (a number grid)
- **Scheduling problems**
 - **Job shop scheduling**
 - **Scheduling the Hubble Space Telescope**



EXAMPLE: 8-Queens

- **Variables:** Queens, one per column
 - Q_1, Q_2, \dots, Q_8
- **Domains:** row placement, $\{1, 2, \dots, 8\}$
- **Constraints:**
 - $Q_i \neq Q_j$ ($j \neq i$)
 - $|Q_i - Q_j| \neq |i - j|$

$$\begin{array}{r}
 X_3 \quad X_2 \quad X_1 \\
 T \quad W \quad O \\
 + \quad T \quad W \quad O \\
 \hline
 F \quad O \quad U \quad R
 \end{array}$$



EXAMPLE: CRYPT ARITHMATIC

- **Variables:** F T U W R O, X1 X2 X3
- **Domain:** {0,1,2,3,4,5,6,7,8,9}
- **Constraints:**
 - Alldiff (F,T,U,W,R,O)
 - $O + O = R + 10 \cdot X_1$
 - $X_1 + W + W = U + 10 \cdot X_2$
 - $X_2 + T + T = O + 10 \cdot X_3$
 - $X_3 = F, T \neq 0, F \neq 0$

EXAMPLE: SUDOKU

	1	2	3	4	5	6	7	8	9
A		6	1		4		5		
B			8	3	5	6			
C	2								1
D	8		4	7					6
E		6				3			
F	7		9	1					4
G	5								2
H		7	2	6	9				
I	4	5	8			7			

- **Variables:**
 - A1, A2, A3, …, A7, A8, A9
 - Letters index rows, top to bottom
 - Digits index columns, left to right
- **Domains: The nine positive digits**
 - A1 --> {1, 2, 3, 4, 5, 6, 7, 8, 9}
- **Constraints:**
 - $AllDiff(A1, A2, A3, A4, A5, A6, A7, A8, A9)$

GRAPH COLORING PROBLEM TYPES

- There are 2 kinds of graph coloring problems
 - **M-colorability Decision problem:** Graph and colors given you should tell in how many possible ways the graph can be colored with no 2 adjacent vertices having same color with the given 'm' set of colors.
 - **M-colorability Optimization problem:** Graph is given colors are not given you should be able to find what is the minimum number 'm' of colors required to color the graph in such a way that no 2 adjacent vertices having same color.
 - The smallest integer m with which a graph G can be colored in such a way that no two adjacent vertices have same color. M is referred to as **Chromatic number of the graph**. Its known to be **chromatic number problem**.
 - For a **planar graph**(if a graph can be drawn in a plane in such a way no 2 edges cross each other) chromatic number can't be more than **4 (Four color theorem)**.
 - In a connected graph in which every vertex has at most Δ neighbors, the vertices can be colored with only Δ colors, except for two cases, complete graphs and cycle graphs of odd length, which require $\Delta + 1$ colors.(**Brook's Theorem**)
-

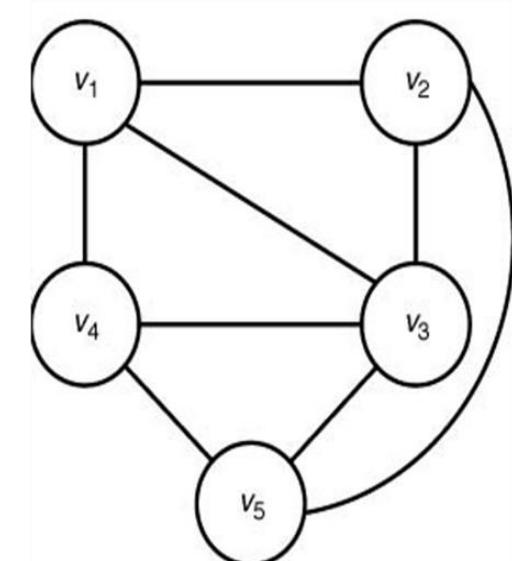
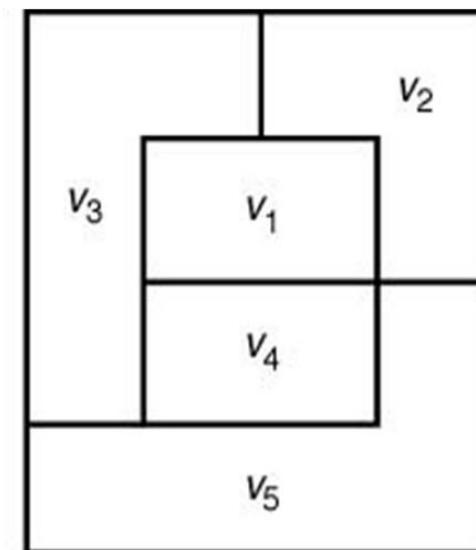
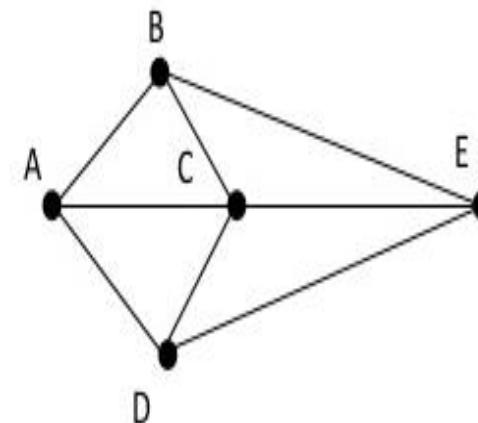
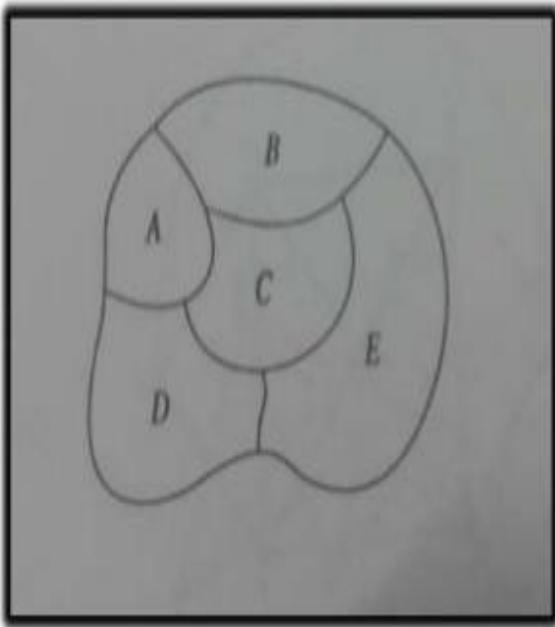


EXAMPLE: MAP COLORING PROBLEM

- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:** $D_i = \{\text{red}, \text{green}, \text{blue}\}$
- **Constraints:** adjacent regions must have different colors
 - e.g., $WA \neq NT$
 - So (WA, NT) must be in $\{(red, green), (red, blue), (green, red), \dots\}$

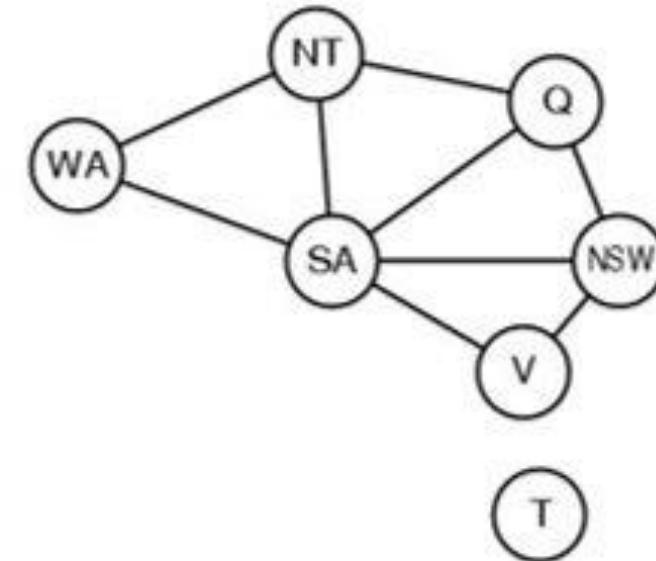
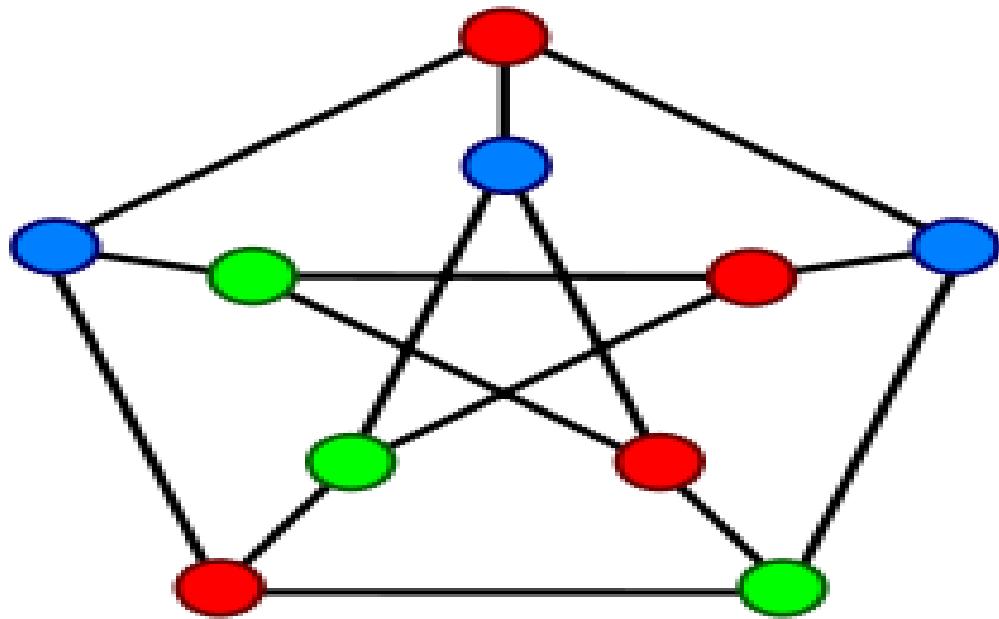
MAP COLORING PROBLEM

- A map can be transformed into a **constraint graph** by representing each region of map into a node and if two regions are adjacent, then the corresponding nodes are joined by an edge.
- Given graph can be colored using 3 colors



CONSTRAINT GRAPH

- *Constraint Graph* helps to visualize *CSP*.
 - *nodes* are variables
 - *arcs* are constraints or link that connects 2 variables that participate in a constraint.



CONSTRAINT TYPES

- *Unary* constraints involve a single variable,
 - e.g., SA \neq 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
 - *Preference* (soft constraints) e.g. *red* is better than *green* can be represented by a cost for each variable assignment => *Constrained optimization* problems.
-

VARIABLE TYPES

- *Discrete variables*
 - **Finite domains:**
 - n variables, domain size $d \rightarrow O(d^n)$ complete assignments
 - e.g., Boolean CSPs, includes Boolean satisfiability (NP-complete)
 - **Infinite domains:**
 - integers, strings, etc.
 - e.g., job scheduling, variables are start/end days for each job
 - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
 - *Continuous variables*
 - e.g., start/end times for Hubble Space Telescope observations
 - linear constraints solvable in polynomial time by linear programming
-

REAL-WORLD CSPS

- Assignment problems: e.g., who teaches what class
 - Timetabling problems: e.g., which class is offered when and where?
 - Hardware configuration
 - Transportation scheduling
 - Factory scheduling
 - Circuit layout
 - Fault diagnosis
-

HOW TO SOLVE CONSTRAINT SATISFACTION PROBLEM

- A CSP can easily be expressed as a **standard search problem**.
- The main idea is to exploit the constraints to **eliminate large portions of search space**.

SOLUTION:

- **Incremental formulation**
 - Assign a value to an unassigned variable provided that it does not violate a constraint
 - End up with $n! \cdot d^n$ leafs even though there are only d^n complete assignments.
- **Backtracking Search** over Assignments
 - Depth- first search that chooses **values for one variable at a time** and **backtracks** when a variable has no legal values left to assign.
 - **Advantage: Solve CSP efficiently even without domain specific knowledge.**

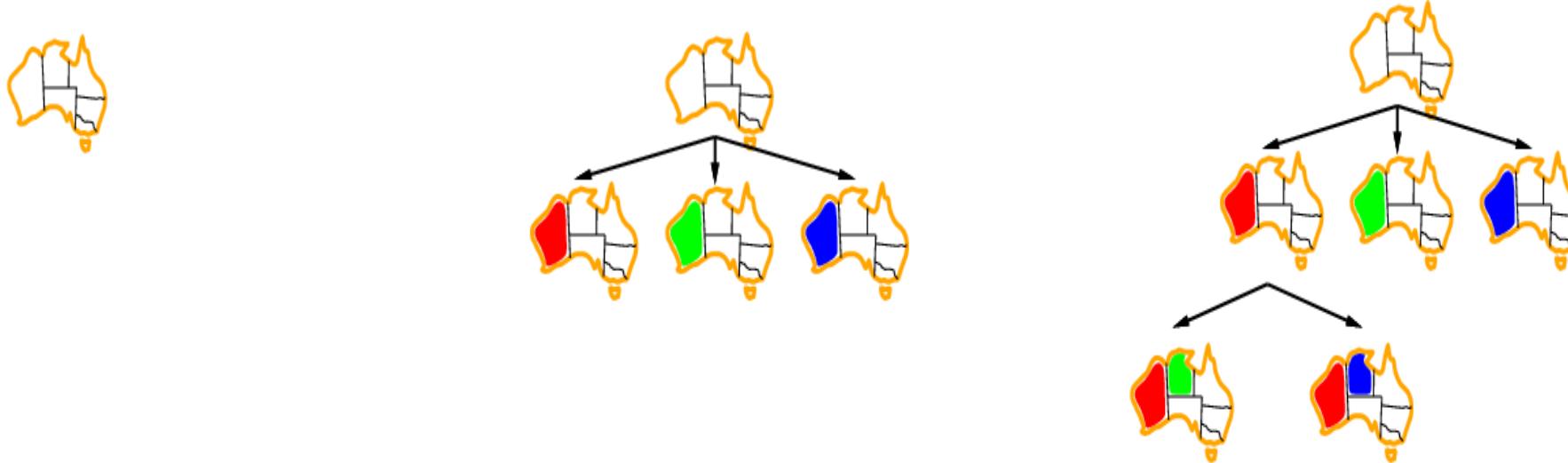
BACKTRACKING SEARCH

- Intelligent Backtracking = DFS + variable-ordering + fail-on-violation
- A **depth-first search** that chooses values for **one variable at a time** and **backtracks** when a **variable has no legal values left to assign**.
- When a **node** is expanded, check that **each successor state is consistent before adding it to the queue**. If its **not consistent or legal** go back to the previous legal state and **start solving by selecting remaining domaining values**.

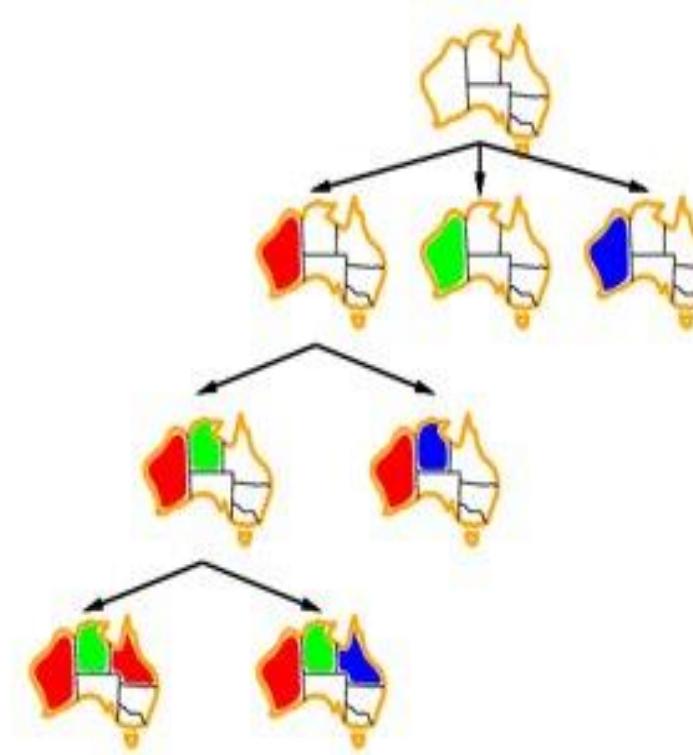
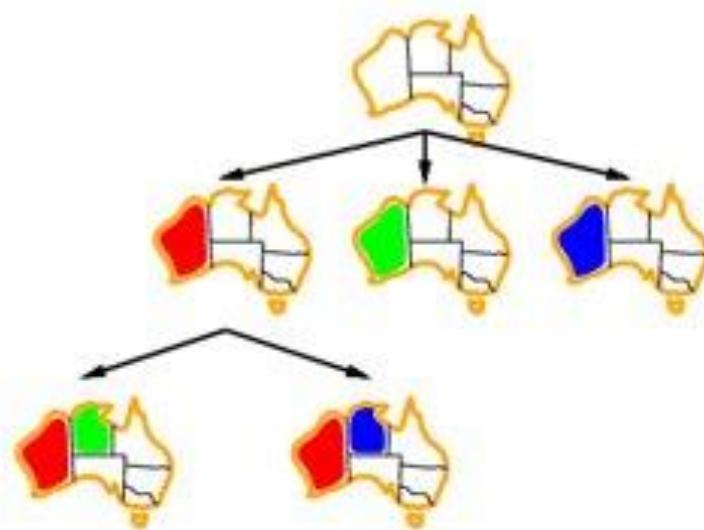
BACKTRACKING SEARCH

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/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 given 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 SEARCH FOR MAP COLORING PROBLEM



BACKTRACKING SEARCH FOR MAP COLORING PROBLEM



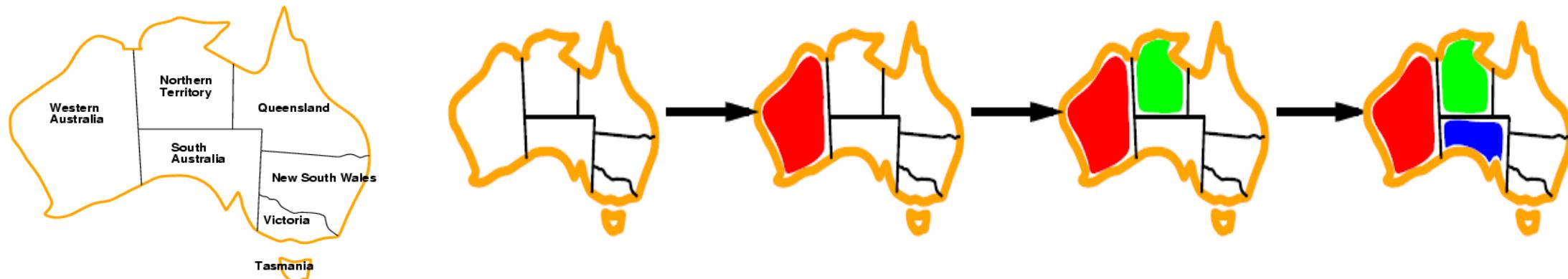
IMPROVING BACKTRACKING EFFICIENCY: CSP

HEURISTICS & PRUNING TECHNIQUES

- In what order should variable should be assigned next ? In which order variable's value should be tried next?
- Variable ordering and value selection heuristics help significantly.
- **Variable ordering:** Which variable should be assigned next?
 - Most constrained variable
 - Minimum Remaining value Heuristics
 - Most constraining variable
 - Degree heuristics
- **Value ordering:**
 - Least constraining value
- Can we detect failures early?
 - Forward checking prevents assignments that guarantee later failure.
- Can we exploit problem structure
 - Arc Consistency

VARIABLE ORDERING: MINIMUM REMAINING VALUES (MRV)

- Expand variables with **minimum size domain** first.
- The idea of choosing the variable with the **fewest “legal” value** or **“most constrained variable”** or **“fail-first” heuristic**, is to pick a variable that is most likely to cause a failure soon thereby pruning the search tree. If some variable X has no legal values left, the MRV heuristic will select X and failure will be detected immediately—avoiding pointless searches through other variables.
- Before the assignment to the rightmost state: one region has one remaining; one region has two; three regions have three. Choose the region with only one remaining



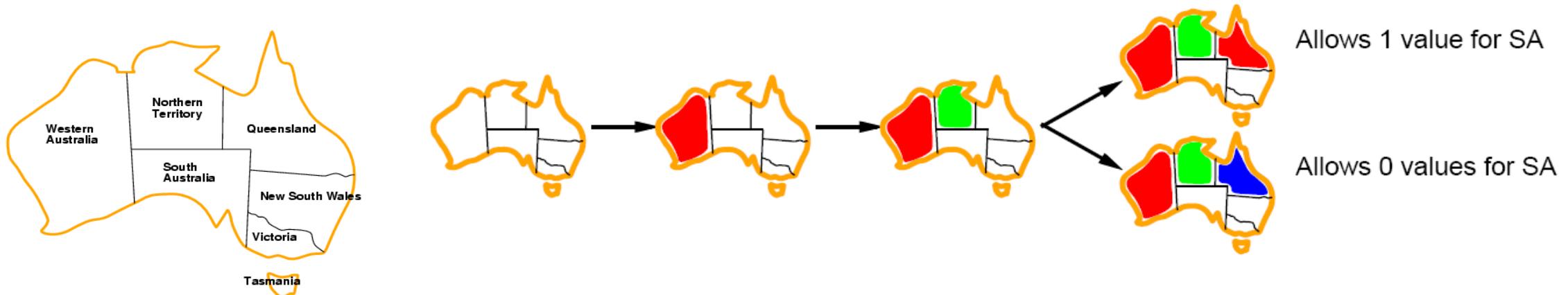
VARIABLE ORDERING: DEGREE HEURISTIC FOR RESOLVING TIES AMONG VARIABLES

- Degree heuristic can be useful as a tie breaker and reduce branching factor.
- The degree heuristic attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables
- Before the assignment to the rightmost state, WA and Q have the same number of remaining values ($\{R\}$). So, choose the one adjacent to the most states. This will cut down on the number of legal successor states to it.
- **Most constraining variable:** choose the variable with the most constraints on remaining variables



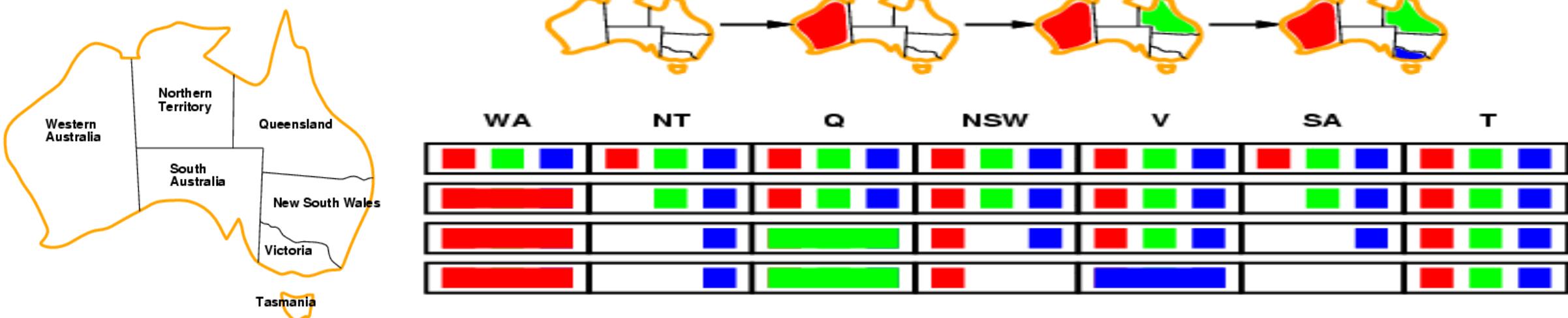
VALUE-ORDERING: LEAST CONSTRAINING VALUE

- Value ordering prefers the value that rules out the fewest choice for the neighboring variables in the remaining variables of constraint graph. **This leave the maximum flexibility for subsequent variable assignments.**



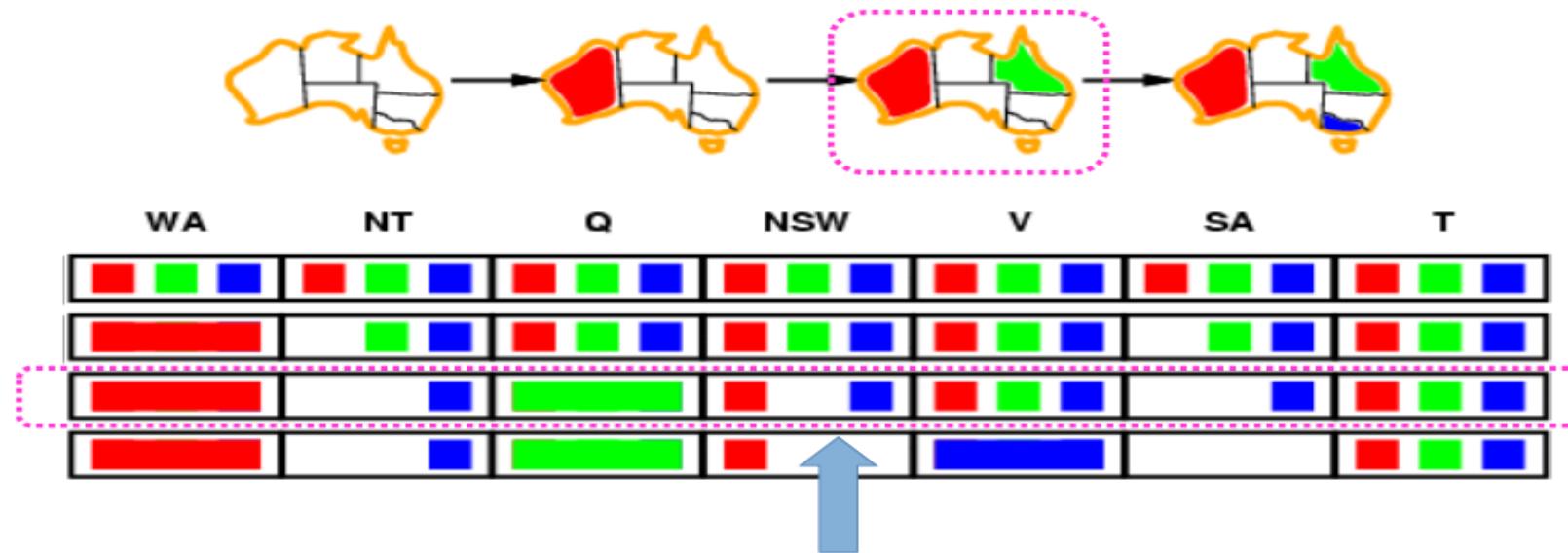
FORWARD CHECKING

- Whenever a variable X is assigned, the forward-checking process establishes **arc consistency** for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X.
- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no legal values



FORWARD CHECKING

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



But, failure was inevitable here!
– what did we miss?

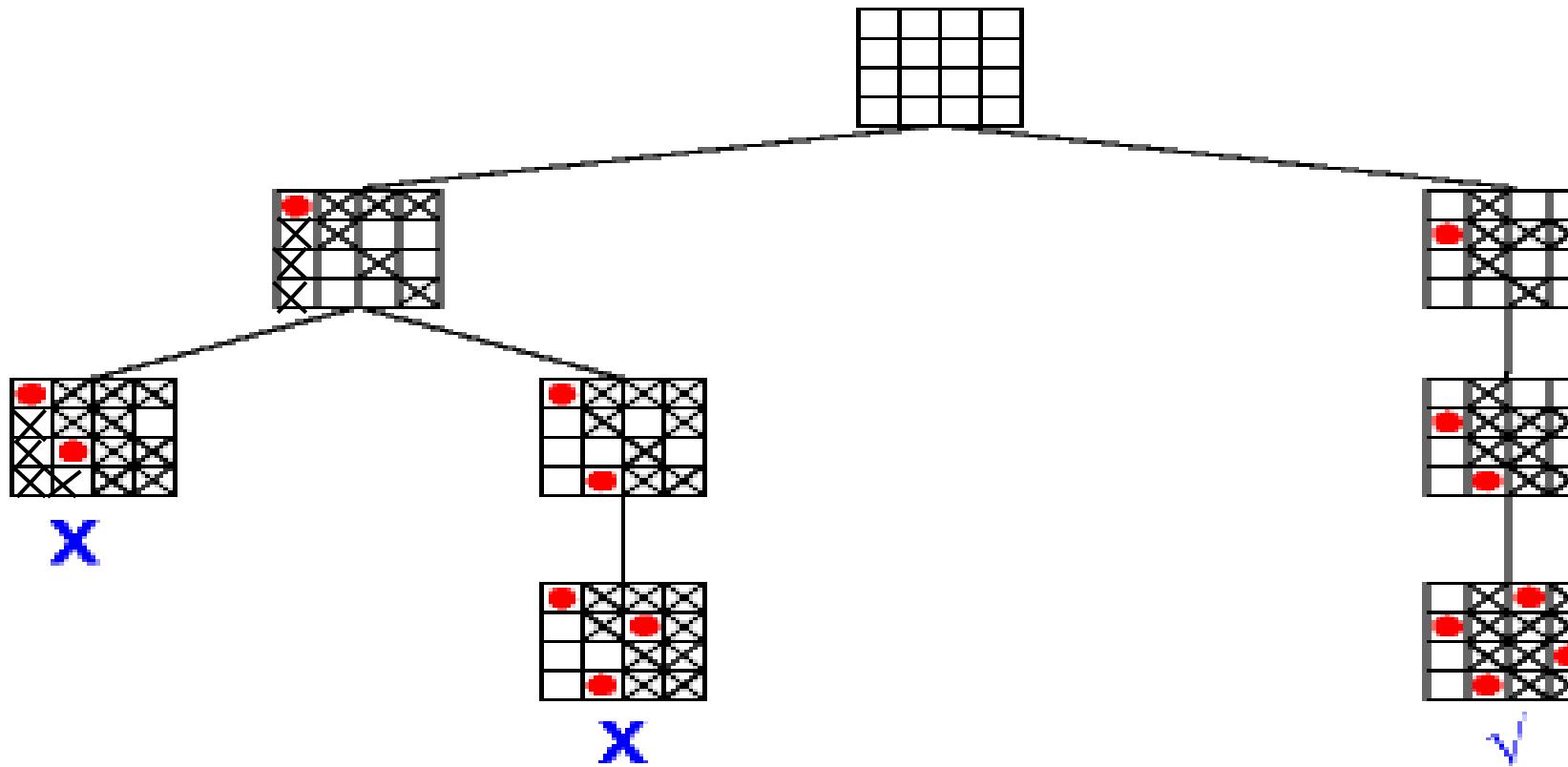
Constraint propagation repeatedly enforces constraints locally

FORWARD CHECKING

	WA	NT	Q	NSW	V	SA	T
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After $WA=red$	(R)	G B	R G B	R G B	R G B	G B	R G B
After $Q=green$	(R)	B	(G)	R B	R G B	B	R G B
After $V=blue$	(R)	B	(G)	R	(B)		R G B

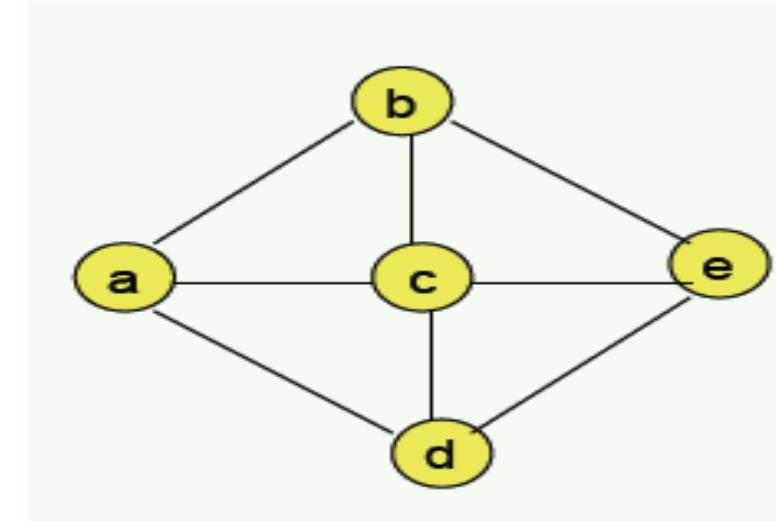
Figure 6.7 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA . After $Q = green$ is assigned, *green* is deleted from the domains of NT , SA , and NSW . After $V = blue$ is assigned, *blue* is deleted from the domains of NSW and SA , leaving SA with no legal values.

4-QUEENS PROBLEM AND FC



PROBLEM

- Find a solution for this CSP by using the following heuristics: minimum value heuristic, degree heuristic, forward checking., least constraining value Explain each step of your answer. The domain for every variable is [1,2,3,4]. There are 2 unary constraints:
 - variable “a” cannot take values 3 and 4.
 - variable “b” cannot take value 4.



SOLUTION

- There are 8 binary constraints stating that variables connected by an edge cannot have the same value.

MVH--> a=1 (for example)

FC+MVH -->b=2

FC+MVH+MD -->c=3

FC+LCV -->d=2

FC -->e=1

CONSTRAINT PROPAGATION

- Forward checking (FC) is in effect eliminating parts of the search space
 - Constraint propagation goes further than FC by repeatedly enforcing constraints locally
 - Needs to be faster than actually searching to be effective
 - Constraint propagation is the process of communicating the domain reduction of a decision variable to all of the constraints that are stated over this variable.
 - Constraint propagation is the process of communicating the domain reduction of a decision variable to all of the constraints that are stated over this variable.
 - Arc-consistency (AC) is a systematic procedure for constraint propagation
 - Simplest form of propagation makes each arch consistent
-

TYPES OF CONSISTENCY

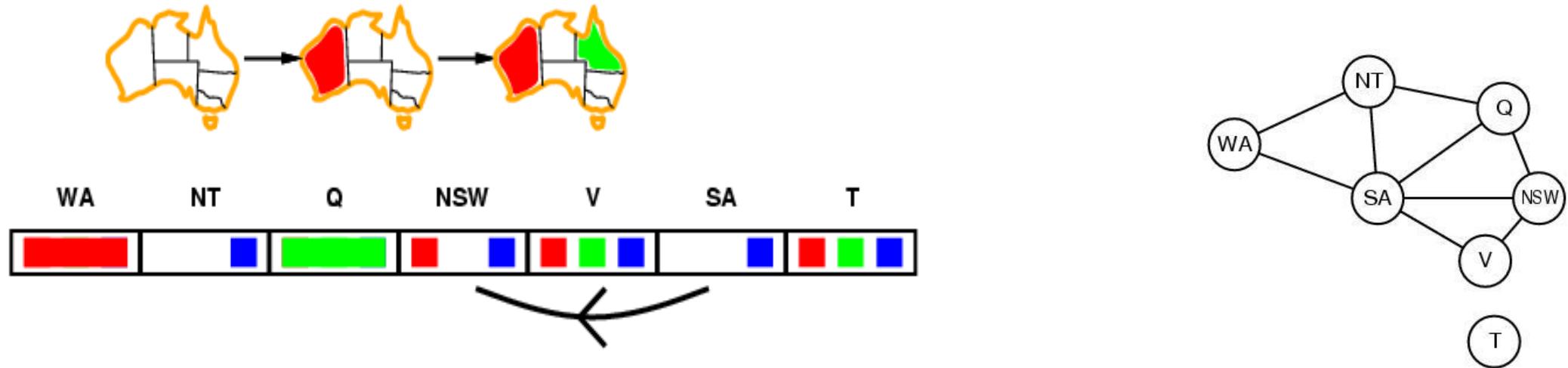
- Maintaining NODE and ARC consistency further reduces the potential DOMAINS of variables, thereby reducing the amount of searching.
 - **NODE CONSISTENCY (1-consistency)**
 - a node is consistent if and only if all values in its domain satisfy all unary constraints on the corresponding variable. (note change here)
 - Unary constraint contains only one variable, e.g., $x_1 \neq R$
 - **ARC CONSISTENCY (2-consistency)**
 - an arc, or edge, $(x_i \rightarrow x_j)$ in the constraint graph is consistent if and only if for every value “a” in the domain of x_i , there is some value “b” in the domain of x_j such that the assignment $\{x_i, x_j\} = \{a, b\}$ is permitted by the constraint between x_i and x_j .
-

ARC CONSISTENCY CHECKING

- **ARC must be run until no inconsistency remains**
- **Trade-off**
 - Requires some overhead to do, but generally more effective than direct search
 - In effect it can eliminate large (inconsistent) parts of the state space more effectively than search can
- **Need a systematic method for arc-checking**
 - If X loses a value, neighbors of X need to be rechecked:
 - **i.e. incoming arcs can become inconsistent again outgoing arcs will stay consistent).**

ARC CONSISTENCY

- *Arc $X \rightarrow Y$ (link in constraint graph) is consistent iff for **every** value x of X there is **some** allowed y . Delete values from tail in order to make each arc consistent*
- 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
- Consider state of search after WA and Q are assigned:
 - $SA \rightarrow NSW$ is consistent: if $SA = \text{blue}$ NSW could be =red



ARC CONSISTENCY

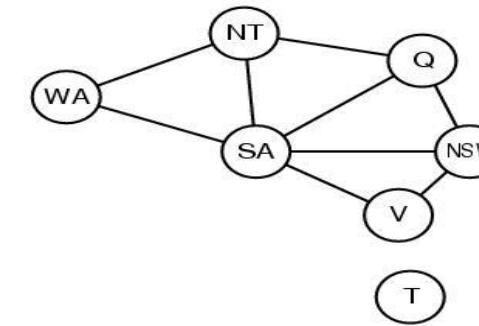
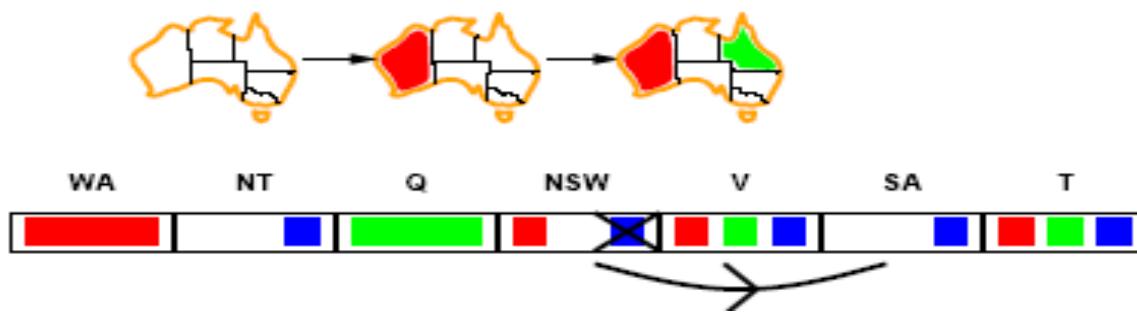
- We will try to make the arc consistent by deleting x's for which there is no y (and then check to see if anything else has been affected – algorithm is in a few slides)
- NSW \rightarrow SA: if NSW=red SA could be =blue

But, if NSW=blue, there is no color for SA.

So, remove blue from the domain of NSW

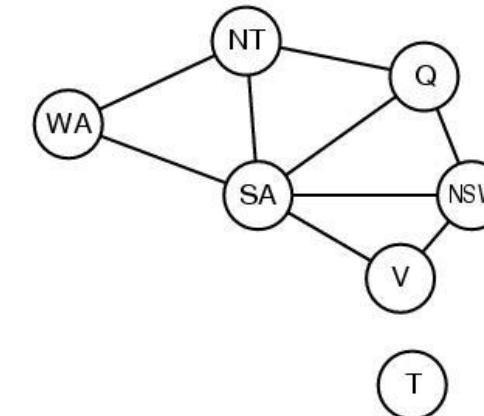
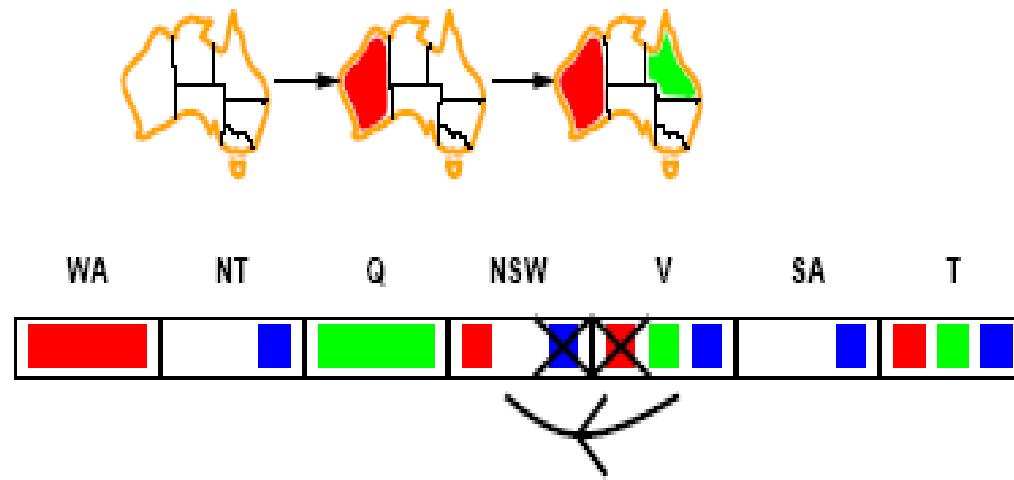
Propagate the constraint: need to check Q \rightarrow NSW SA \rightarrow NSW V \rightarrow NSW

If we remove values from any of Q, SA, or V's domains, we will need to check THEIR neighbors



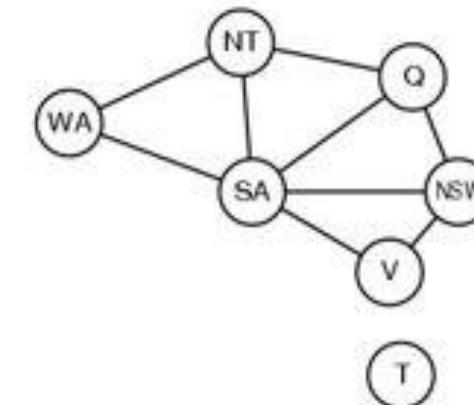
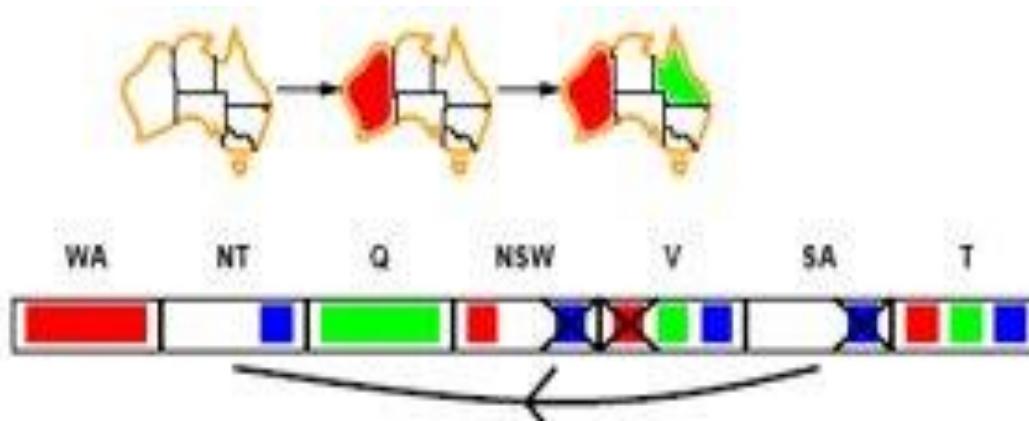
ARC CONSISTENCY

- After removing red from domain of V to make $V \rightarrow NSW$ arc consistent
- $SA \rightarrow V, NSW \rightarrow V$ check out; no changes
- *Check the remaining arcs: most check out, until we check $SA \rightarrow NT, NT \rightarrow SA$. Whichever is checked first will result in failure.*



ARC CONSISTENCY

- $SA \rightarrow NT$ is not consistent and cannot be made consistent
- Arc consistency detects failure earlier than FC
- This process was all in one call to the INFERENCE function right after we assigned $Q=\text{green}$. Forward checking proceeded in the search, assigning a value to V.



ARC CONSISTENCY ALGORITHM(AC-2)

function AC-2(csp) returns false if inconsistency found, else true, may reduce csp domains

local variables: $queue$, a queue of arcs, initially all the arcs in csp

while queue is not empty do /* *initial queue must contain both (X_i, X_j) and (X_j, X_i)* */

$(X_i, X_j) \leftarrow REMOVE-FIRST(queue)$

if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then

 if size of $D_i = 0$ then return false

 for each X_k in NEIGHBORS[X_i] – { X_j } do

 add (X_k, X_i) to queue if not already there

return true

function REMOVE-INCONSISTENT-VALUES(X_i, X_j) returns *true* iff we delete a

value from the domain of X_i

$removed \leftarrow false$

for each x in DOMAIN[X_i] do

 if no value y in DOMAIN[X_j] allows (x, y) to satisfy the constraints

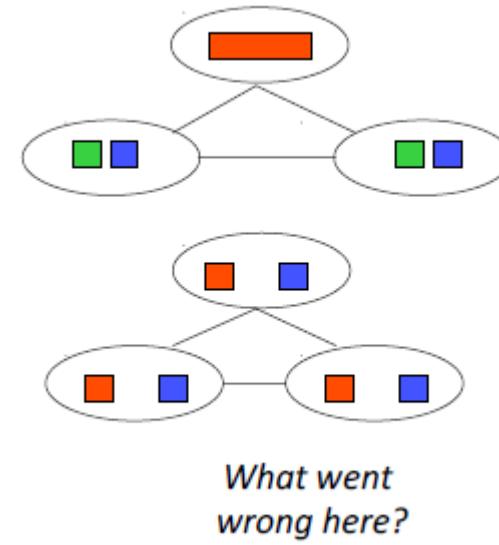
 between X_i and X_j

 then delete x from DOMAIN[X_i]; $removed \leftarrow true$

return $removed$

ARC CONSISTENCY DOES NOT DETECT ALL INCONSISTENCIES

- After enforcing arc consistency:
- Can have one solution left
- Can have multiple solutions left
- Can have no solutions left (and not know it)
- Arc consistency still runs inside a backtracking search!



INTELLIGENT BACKTRACKING

- **Chronological Backtracking** At dead end, backup to the most recent variable.

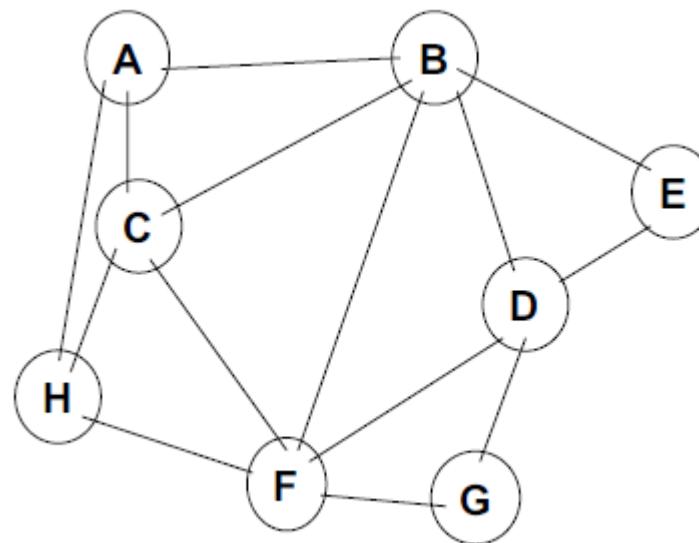
Eg: {Q-red, NSW=green, V=blue, T=red}. For next variable SA no values are left out and backtracking to T cannot resolve problem with SA.

- **Backjumping** At dead end, backup to the most recent variable that eliminated some value in the domain of the dead end variable. Ie backtrack variable that fix problem
- Keep track of conflict set(set of assignments that are in conflict with values of SA. Eg: {Q-red, NSW=green, V=blue})
- **Conflict-directed Backjumping** A backjumping algorithm that uses conflict sets to backtrack.i.e backjumping occurs when current value in domain is in conflict with current assignment.
- **Forward checking** supplies **conflict set**. **Backjumping** occurs when every value in a domain is in conflict with current assignment ie for one variable domain becomes empty.
- **Constraint Learning** is idea of finding minimum set of variables from conflict set that causes problem.



CHALLENGE

Find a solution for this CSP by using the following heuristics: minimum value heuristic, degree heuristic, forward checking., least constraining value Explain each step of your answer. The domain for every variable is [1,2,3,4].



Thank You

anooja@somaiya.edu

KNOWLEDGE AND REASONING



Prepared By
-Anooa Joy



KNOWLEDGE-BASED AGENTS

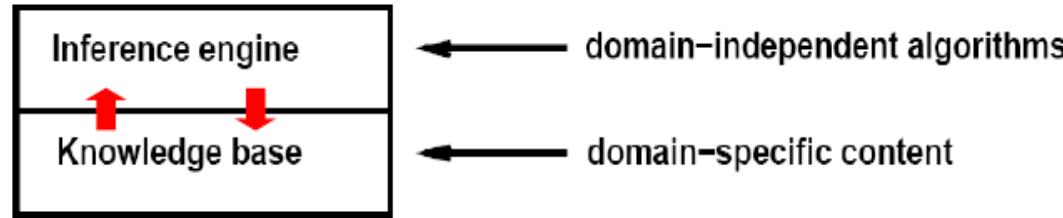
"An agent can represent knowledge of its world, its goals and the current situation by sentences in logic and decide what to do by inferring that a certain action or course of action is appropriate to achieve its goals."

LOGICAL SYSTEM

- Intelligent agents need knowledge about the world to choose good actions/decisions.
 - A **logical system** is a system that knows about its **partially observable environment** and can reason about possible actions by **inferring from the hidden information**. **Reasoning** is also known as **inferencing**. An agent that acts upon logical system is known as **Knowledge based Agent**.
 - Issues in construction of a logical system:
 1. **Knowledge representation:** how do we represent information? Knowledge representation should be somewhat natural, expressive and efficient.
 2. **Knowledge reasoning:** how do we use information to reach decisions and/or derive new facts?
 - Knowledge in the form of a set of facts about our environment are stored in a knowledge base (KB).
 - **Facts** are claims about the environment which are either **true** or **false**. **Facts** are represented by sentences
 - A **sentence** is an assertion about the world. **Sentences** are expressed in a representation language.
-

KNOWLEDGE BASED AGENT

- A knowledge-based agent(Logic Agent)comprises of 2 features:
 1. **Knowledge base:** domain-specific content ie a list of facts that are known to the agent.
 2. **Inference engine:** domain-independent algorithms for inferencing new knowledge. *Current percepts* to infer hidden aspects of the current state using *Rules of inference* .
- **Knowledge base:** A set of sentences in in a formal knowledge representation language that encodes assertions about the world.



A KNOWLEDGE BASED AGENT

- The agent must be able to:
 - Represent states, actions, etc.
 - Incorporate new percepts
 - Update internal representations of the world
 - Deduce hidden properties of the world
 - Deduce appropriate actions
-

A KNOWLEDGE BASED AGENT

- **Declarative approach** to build a knowledge based agent
- The agent operates as follows:
 - **Add new sentences:** It TELLS the knowledge base what it perceives based on what it wants to know.
 - **Query what is known:** It ASKS the knowledge base what action it should perform. The answers should follow from the KB.
 - **Execute Action:** It performs the chosen action.
- **Procedural approach** to build a knowledge based agent
 - Encode desired behaviors directly as program code
 - Minimizing the role of explicit representation and reasoning can result in a much more efficient system. In this approach, knowledge is stored into an empty system in the form of program code. It designs the behavior of the system via coding



Mechanism of an Agent Program

LEVELS OF A KNOWLEDGE-BASED AGENT

- **Knowledge Level:** In this level, the behavior of an agent is decided by specifying the following :
 - The agent's current knowledge it has perceived.
 - The goal of an agent.
 - **Implementation Level:** This level is the physical representation of the knowledge level. Here, it is understood that “how the knowledge-based agent actually implements its stored knowledge.”
-

KNOWLEDGE BASED AGENT

```
function KB-AGENT(percept) returns an action
    static: KB, a knowledge base
            t, a counter, initially 0, indicating time
    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    action  $\leftarrow$  ASK(KB, MAKE-ACTION-QUERY(t))
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t  $\leftarrow$  t + 1
    return action
```

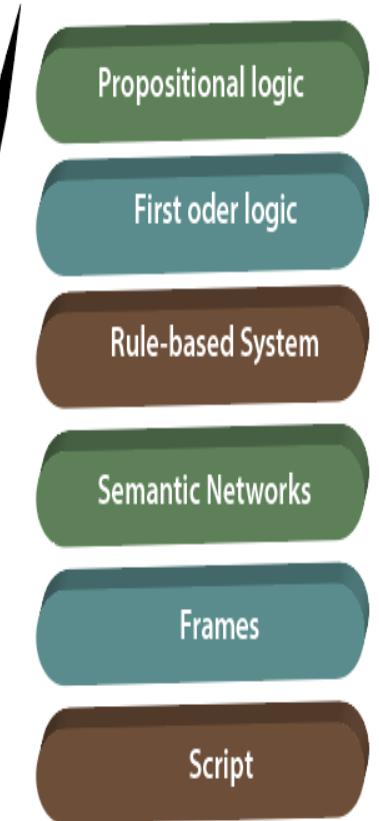
TECHNIQUES USED FOR KNOWLEDGE REPRESENTATION

- **Logic:** It is the basic method used to represent the knowledge of a machine. The term logic means to apply intelligence over the stored knowledge.

Logic can be further divided as:

1. **Propositional Logic:** This technique is also known as **propositional calculus, statement logic, or sentential logic**. It is used for representing the knowledge about **what is true and what is false**.
2. **First-order Logic:** It is also known as **Predicate logic or First-order predicate calculus (FOPL)**. This technique is used to represent the objects in the form of **predicates or quantifiers**. It is different from Propositional logic as it removes the complexity of the sentence represented by it. In short, FOPL is an advance version of propositional logic.

Techniques used for
Knowledge
Representation

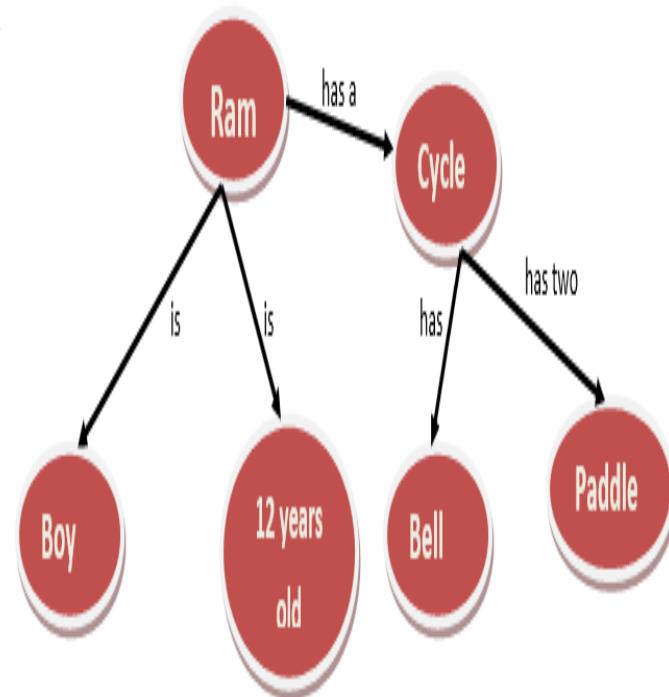


TECHNIQUES USED FOR KNOWLEDGE REPRESENTATION

3. Rule-based System: In the rule-based system, we impose rules over the propositional logic and first-order logic techniques. If-then clause is used for this technique. **For example,** if there are two variables A and B. Value of both A and B is True. Consequently, the result of both should also be True and vice-versa. **It is represented as:** If the value of A and B is True, then the result will be True. So, such a technique makes the propositional as well as FOPL logics bounded in the rules.

4. Semantic Networks: The technique is based on storing the knowledge into the system in the form of a graph. Nodes of a graph represent the objects which exist in the real world, and the arrow represents the relationship between these objects. Such techniques show the connectivity of one object with another object. **For example,** Consider the given knowledge stored in a machine:

- Ram has a cycle.
- Ram is a boy.
- Cycle has a bell.
- Ram is 12 years old.
- Cycle has two paddles.



TECHNIQUES USED FOR KNOWLEDGE REPRESENTATION

5. Frames: In this technique, the knowledge is stored via **slots and fillers**. Slots are the entities and Fillers are its attributes similar to database. They are together stored in a frame. So, whenever there is a requirement, the machine infers the necessary information to take the decision. **For example**, Tomy is a dog having one tail. It can be framed as:

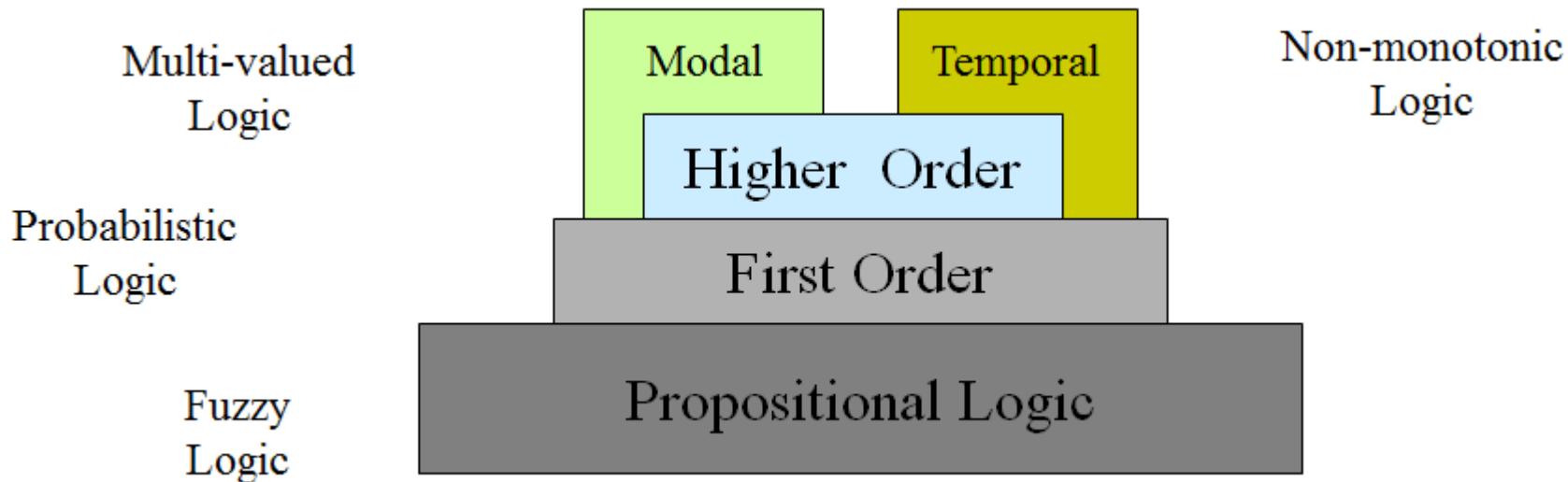
Tomy((Species (Value = Dog))

(Feature (Value = Tail)))

6. Script: It is an advanced technique over the Frames. Here, the information is stored in the form of a script. The script is stored in the system containing all the required information. The system infers the information from that script and solves the problem



LOGIC AS A KR LANGUAGE

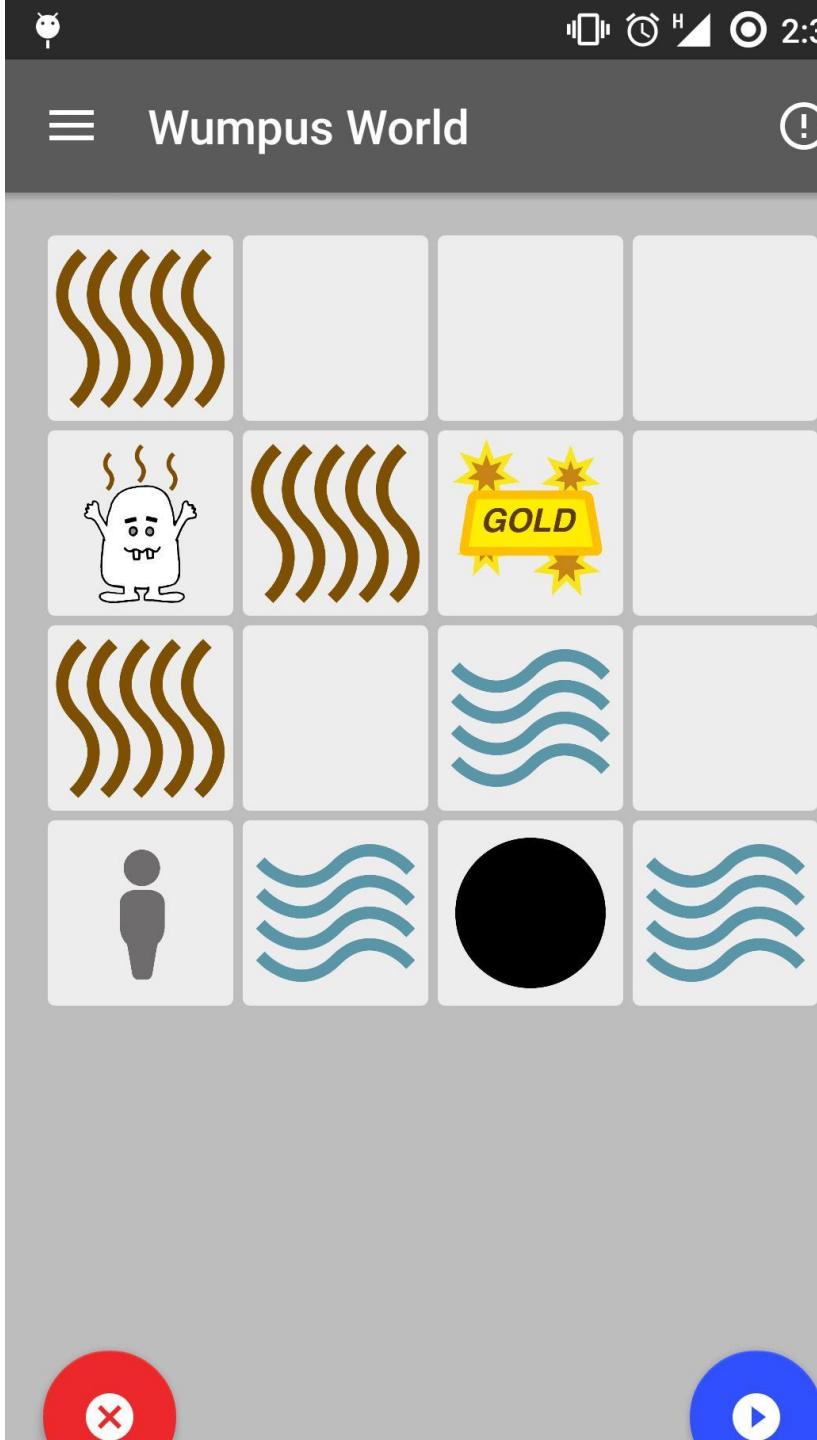


WUMPUS WORLD



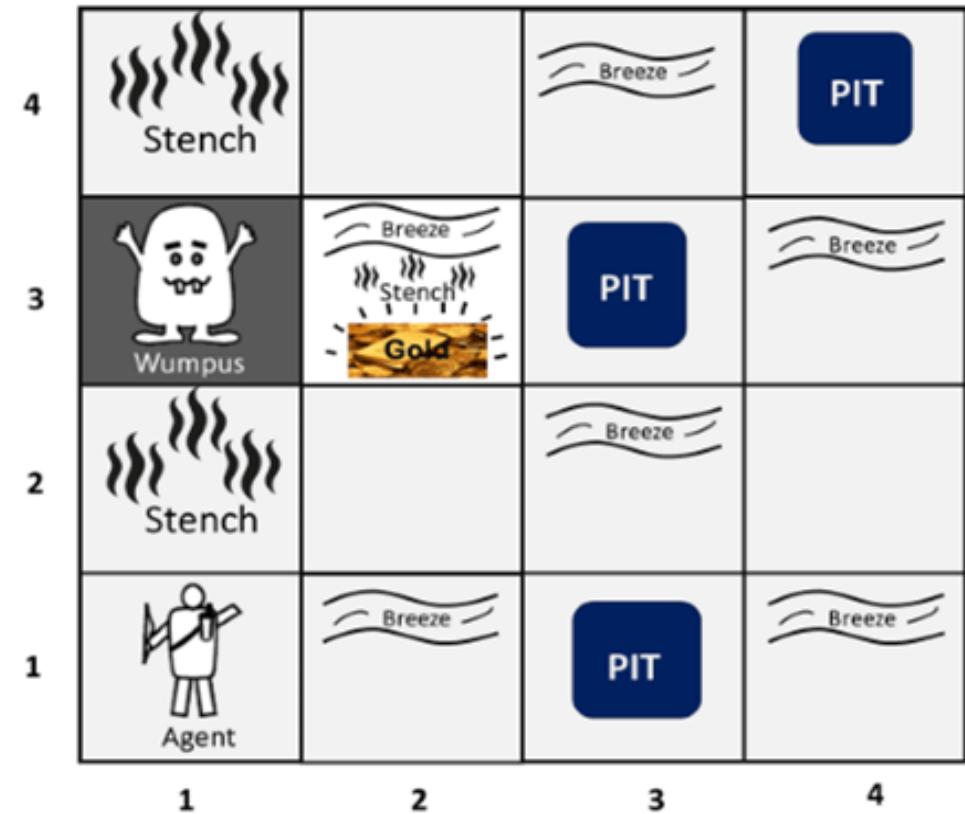
WUMPUS WORLD GAME DESCRIPTION

The Wumpus World is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the Wumpus, a beast that eats any agent that enters its room. The Wumpus can be shot by an agent, but agent has only one arrow. Some rooms contain bottomless pits that trap any agent that wanders into the room. Occasionally, there is a heap of gold in a room. The goal is to collect the gold and exit the world without being eaten.



WUMPUS WORLD ENVIRONMENT

- The agent always starts in the field [1,1].
- The task of the agent is to **find the gold**, **return to the field [1,1]** and **climb out of the cave**.
- Squares adjacent to Wumpus are smelly and squares adjacent to pit are breezy (not diagonal)
- Glitter iff gold is in the same square
- Shooting kills Wumpus if you are facing it
- Wumpus emits a horrible scream when it is killed that can be heard anywhere
- Shooting uses up the only arrow
- Grabbing picks up gold if in same square
- Releasing drops the gold in same square



PEAS DESCRIPTION

- **Performance measure**
 - gold: +1000, death: -1000
 - -1 per step , -10 for using the arrow
 - **Environment**
 - Squares adjacent to Wumpus are smelly
 - Squares adjacent to pit are breezy
 - Glitter iff gold is in the same square
 - Gold is picked up by reflex, can't be dropped
 - Shooting kills Wumpus if you are facing it. It screams
 - Shooting uses up the only arrow
 - Grabbing picks up gold if in same square
 - Releasing drops the gold in same square
 - You bump if you walk into a wall
 - **Actuators:** Face , Move, Grab, Release, Shoot
 - **Sensors:** Stench, Breeze, Glitter, Bump, , Scream
-

WUMPUS WORLD CHARACTERIZATION

1. **Deterministic** **Yes** – outcomes exactly specified
 2. **Static** **Yes** – Wumpus and Pits do not move
 3. **Discrete** **Yes**
 4. **Single-agent** **Yes** – Wumpus is essentially a natural feature
 5. **Fully Observable** **No** – only local perception
 6. **Episodic** **No**—What was observed before (breezes, pits, etc) is very useful.
-

EXPLORING THE WUMPUS WORLD

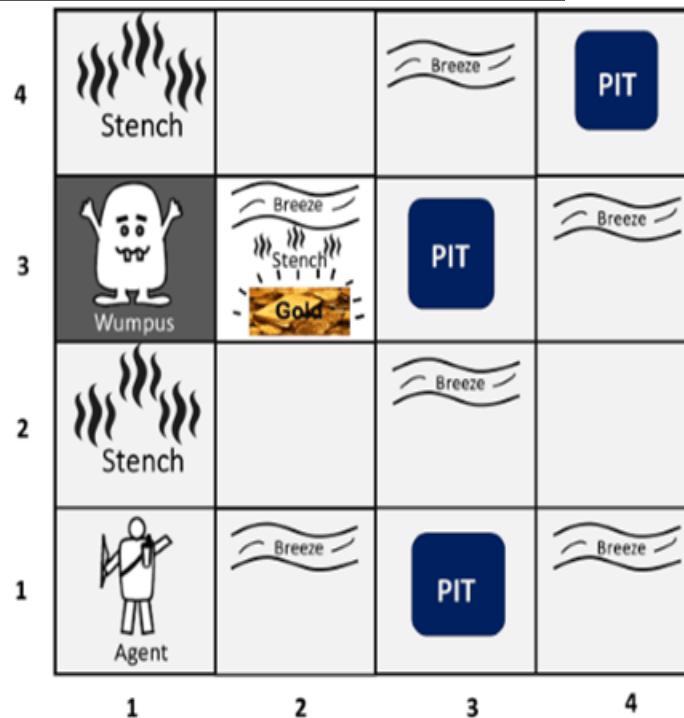
1,4	2,4	3,4	4,4	
1,3	2,3	3,3	4,3	
1,2	2,2	3,2	4,2	
OK				
1,1	A	2,1	3,1	4,1

(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	A	2,1	3,1	4,1

(b)



1. The KB initially contains the rules of the environment.

2. **Location: [1,1]**

Percept: [\neg Stench, \neg Breeze, \neg Glitter, \neg Bump]=[None, None, None, None]

Action: Move to safe cell e.g. 2,1

3. **Location: [2,1]**

Percept: [\neg Stench, Breeze, \neg Glitter, \neg Bump]

INFERENCE: Breeze indicates that there is a pit in [2,2] or [3,1]

Action: Return to [1,1] to try next safe cell

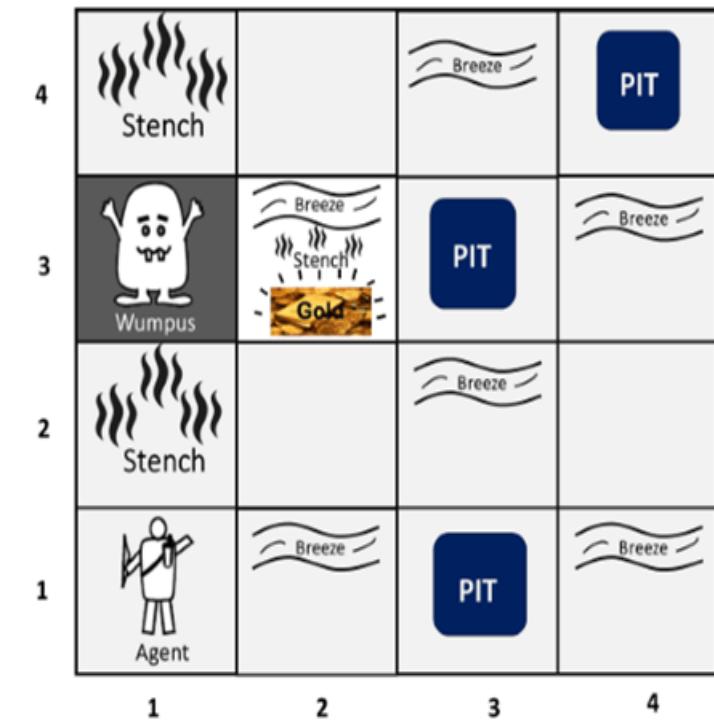
EXPLORING THE WUMPUS WORLD

1,4	2,4	3,4	4,4
1,3 W!	2,3	3,3	4,3
1,2 A S OK	2,2	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

A = Agent
B = Breeze
G = Glitter, Gold
OK = Safe square
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 OK	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

(b)



4. Location: [1,2] (after going through [1,1])

Percept: [Stench, \neg Breeze, \neg Glitter, \neg Bump]

INFER: Wumpus is in [1,1] or [2,2] or [1,3]

INFER... stench not detected in [2,1], thus not in [2,2]

REMEMBER.... Wumpus not in [1,1]

THUS... Wumpus is in [1,3]

THEREFORE [2,2] is safe because of lack of breeze in [1,2]

Action: Move to [2,2]

REMEMBER: Pit in [2,2] or [3,1]

THEREFORE: Pit in [3,1]!

An illustration featuring two profile silhouettes of human heads facing each other. The head on the left is colored red and orange, while the head on the right is blue. Both heads are filled with various colored gears (orange, blue, purple, green) of different sizes, symbolizing thought or logic. A dashed line connects the two profiles. The background is yellow with faint gear patterns.

LOGIC

LOGIC

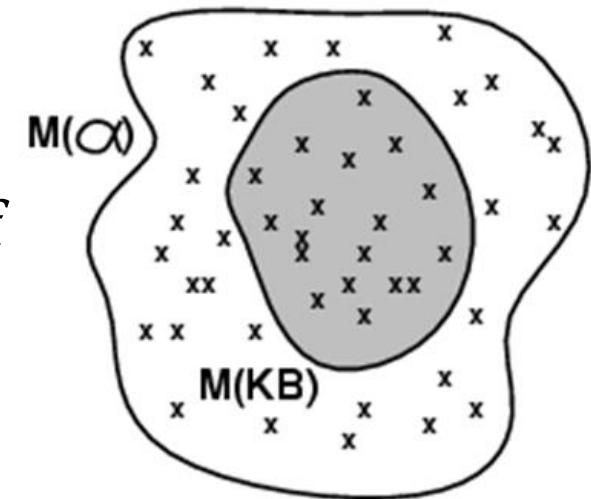
- The objective of **knowledge representation** is to express knowledge in a **computer-tractable form**, so that agents can perform well.
 - *Logics are formal languages for representing information such that conclusions can be drawn.*
 - A **formal knowledge representation language** is defined by:
 - its **syntax**, which **defines all possible sequences of symbols that can be put together to constitute sentences of the language**.
 - its **semantics**, which **determines the facts in the world to which the sentences refer**. It define the "**meaning**" of sentences.
 - Each **sentence** makes a **claim about the world**. An agent is said to believe a sentence about the world.
 - E.g., the language of arithmetic
 - $x+2 \geq y$ is a sentence; $x2y +> \{\}$ is not a sentence
 - $x+2 \geq y$ is true iff the number $x+2$ is no less than the number y
 - $x+2 \geq y$ is true in a world where $x = 7, y = 1$
 - $x+2 \geq y$ is false in a world where $x = 0, y = 6$
-

INFERENCE WITH KNOWLEDGE AND ENTAILMENT

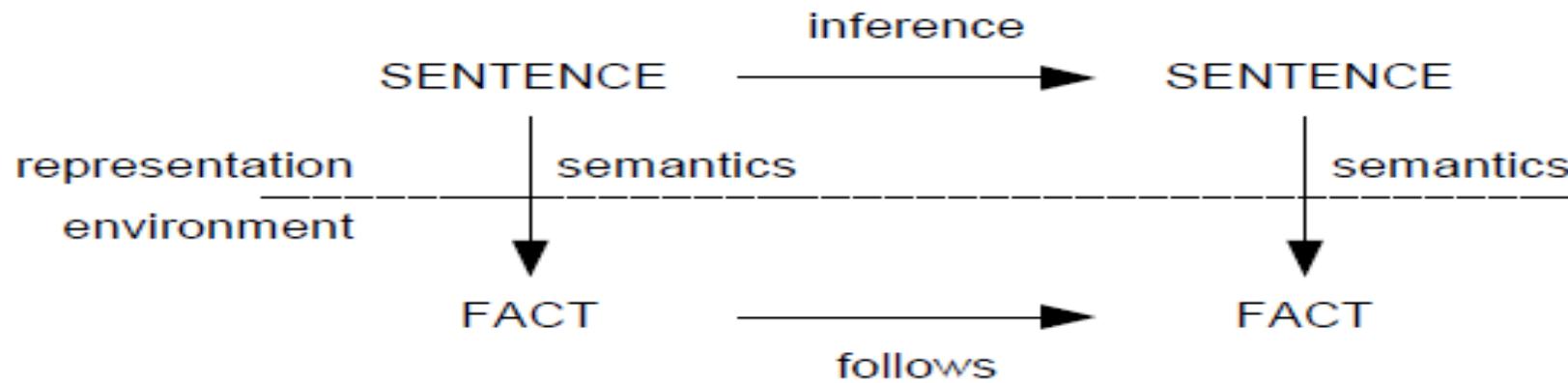
- **Inferencing** is how we derive:
 - **Conclusions** from **existing knowledge**;
 - **New information** from **existing information**. Inferencing might be used in both ASK and TELL operations.
- **Entailment** is the generation or discovery that a new sentence is **TRUE** given existing sentences. Entailment means that one thing follows logically from another. **Entailment** is a relationship between sentences (i.e., syntax) that is **based on semantics**. Knowledge base KB entails sentence α if and only if α is true in all worlds where KB is true ie, $\text{KB} \models \alpha$
- **E.g.**
 1. KB containing “the Phillies won” and “the Reds won” entails “Either the Phillies won or the Reds won”
 2. $x+y = 4$ entails $4 = x+y$

MODELS

- Logicians typically think in terms of **models**, which are **formally structured worlds with respect to which truth can be evaluated**.
- We say m is a model of a sentence α if α is true in m .
- $M(\alpha)$ is the set of all models of α , then $KB \models \alpha$ iff $M(KB) \models M(\alpha)$
- E.g.
 1. $KB = \text{Phillies won and Yankees won}$
 2. $\alpha = \text{Phillies won}$



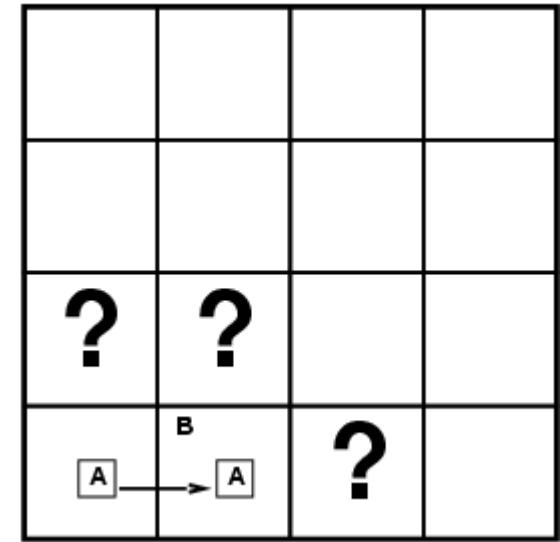
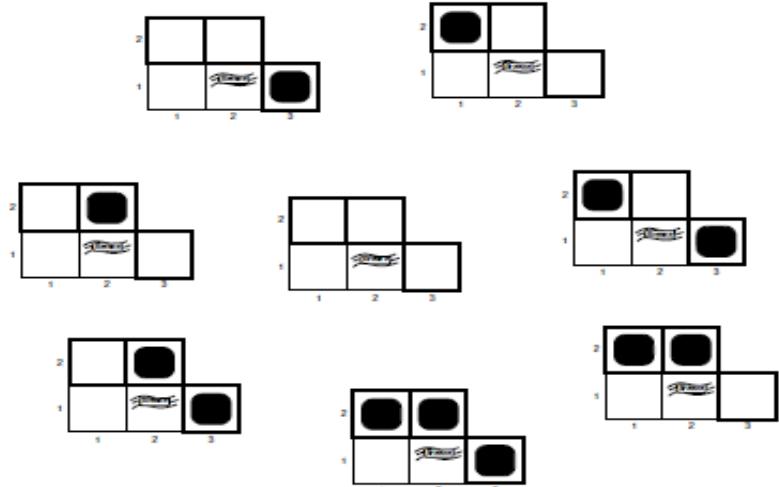
THE CONNECTION BETWEEN SENTENCES AND FACTS



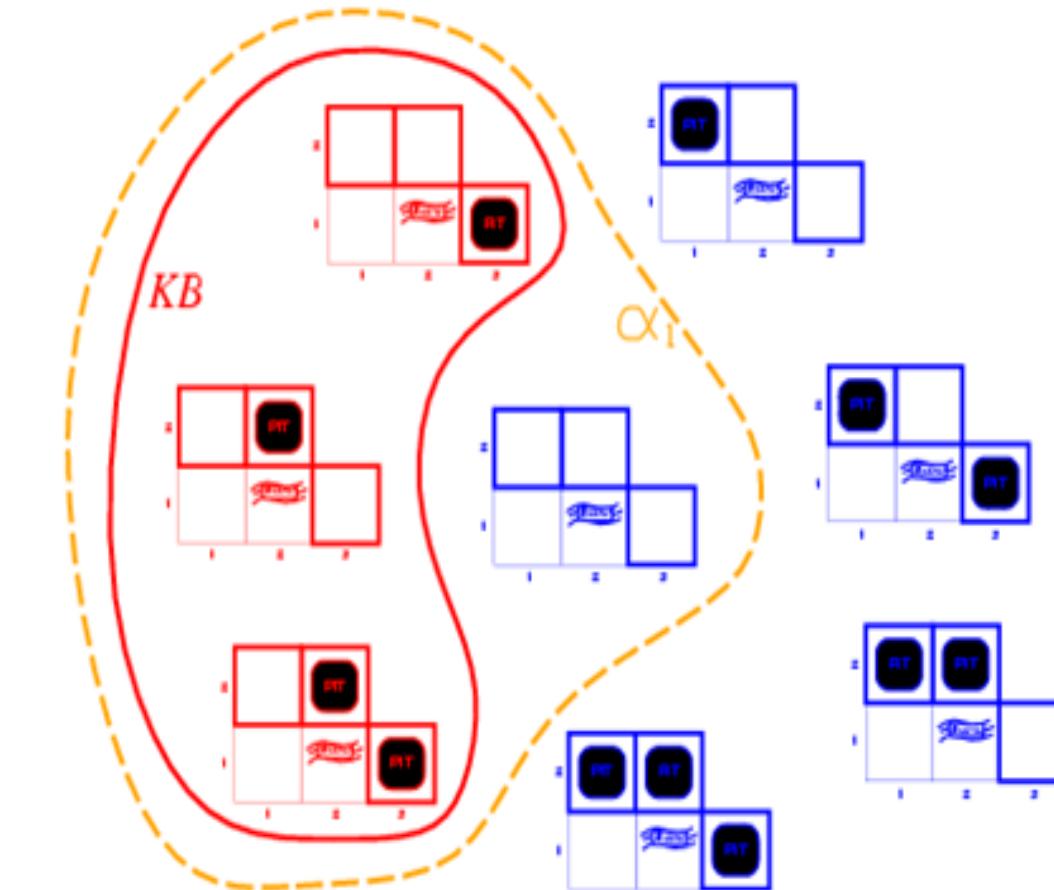
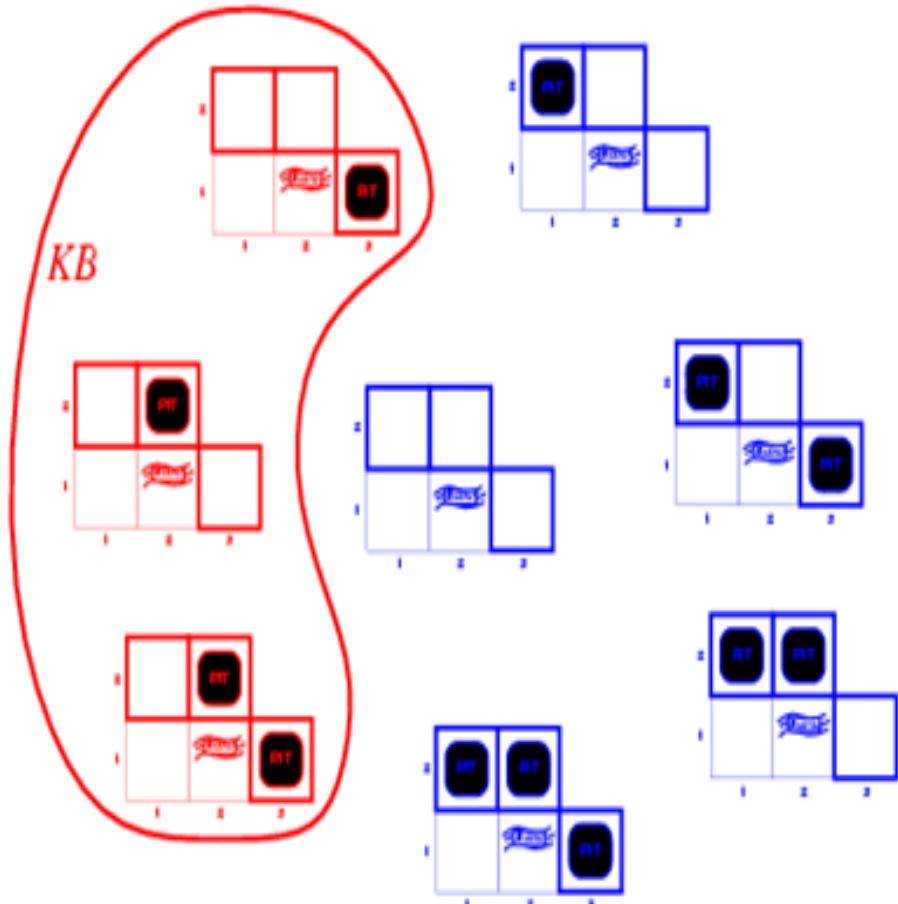
- Semantics maps sentences in logic to facts in the world.
- The property of one fact following from another is mirrored by the property of **one sentence being entailed by another**.
- If KB is true in the real world, then any sentence α derived from KB by a sound inference procedure is also true in the real world

ENTAILMENT IN THE WUMPUS WORLD

- Situation after detecting nothing in [1,1], moving right, breeze in [2,1]
- Consider possible models for KB assuming only pits
- 3 Boolean choices \Rightarrow 8 possible models



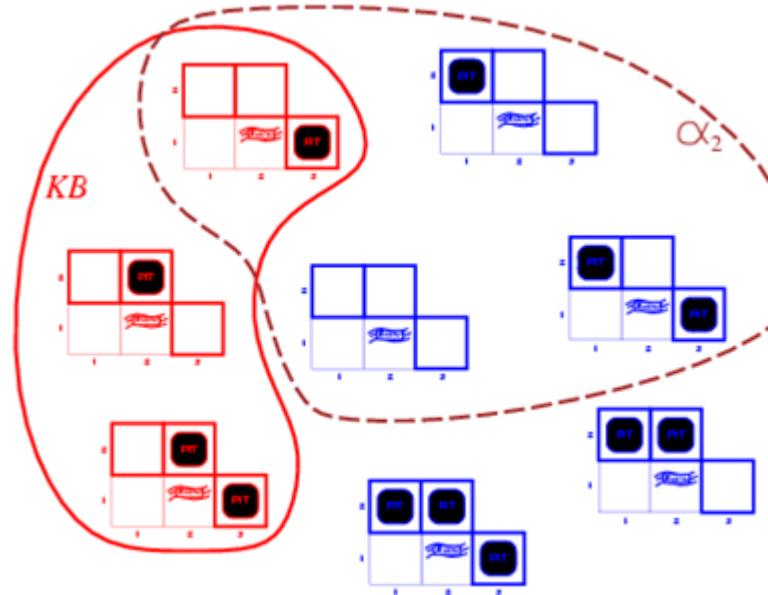
WUMPUS MODELS



- KB = wumpus-world rules + observations

$\alpha_1 = [1,2]$ is safe''

WUMPUS MODELS



- KB = wumpus-world rules + observations
- α_2 = “there is no pit in [2,2]”, $KB \models \alpha_2$

SOUNDNESS AND COMPLETENESS

- A **sound** inference method derives only entailed sentences. Ie, $KB \vdash_I \alpha$ = sentence α can be derived from KB by inference procedure I
- **Soundness:** i is sound if whenever $KB \vdash_i \alpha$, it is also true that $KB \models \alpha$
- Analogous to the property of **completeness** in search, a *complete* inference method can derive any sentence that is entailed.
- **Completeness:** i is complete if whenever $KB \models \alpha$, it is also true that $KB \vdash_i \alpha$
- Preview: we will define a logic (first-order logic) which is expressive enough to say almost anything of interest, and for which there exists a sound and complete inference procedure. That is, the procedure will answer any question whose answer follows from what is known by the KB .

PROPOSITIONAL LOGIC

- **Propositional Logic** also known as simply “Boolean logic” is a method to achieve **knowledge representation** and **logical inferencing**.
- Propositional logic consists of **Syntax** and **Semantics**

SYNTAX

- The **symbols** and the **connectives** together define the **syntax of the language**. Again, syntax is like grammar.
- **TRUTH SYMBOLS:** T (true) and F (false) are provided by the language. Either T or F.
- **PROPOSITIONAL SYMBOLS:** P, Q, R, etc. mean something in the environment. Proposition symbols are **sentences**.
- E.g: **P** means “It is hot”, **Q** means “It is humid”, **R** means “It is raining”, “If it is hot and humid, then it is raining”
 $P \wedge Q \Rightarrow R$
- **Syntax** can have:
 - **ATOMIC SENTENCE:** Truth and propositional symbols are considered ATOMIC SENTENCES. Atomic sentences must have truth assigned (i.e., be assigned T or F).
 - **COMPLEX SENTENCES:** More complex sentences are formed using connectives. Sentences formed in this way can be called **Well-Formed Formula (WFF)**. The evaluation of complex sentences is done using truth tables for the connectives.

SEMANTICS

- Need to be able to evaluate sentences to true or false. The **truth tables** define the **semantics of the language**.
-

LOGICAL CONNECTIVES

- \neg or **NOT** or **NEGATION**: If S_1 is a sentence, then $\neg S_1$ is a sentence
 - \wedge or **AND** or **CONJUNCTION**: If S_1, S_2 are sentences, then $S_1 \wedge S_2$ is a sentence
 - \vee or **OR** or **DISJUNCTION**: If S_1, S_2 are sentences, then $S_1 \vee S_2$ is a sentence
 - \Rightarrow or **IFTHEN** or **IMPLICATION**: If S_1, S_2 are sentences, then $S_1 \Rightarrow S_2$ is a sentence
 - \Leftrightarrow or **IFF** or **BICONDITIONAL**: If S_1, S_2 are sentences, then $S_1 \Leftrightarrow S_2$ is a sentence
 - **Parentheses** can be used to indicate precedence.
 - **KB is conjunction (AND) of all facts.**
-

PROPOSITIONAL LOGIC TRUTH TABLE

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

PRECEDENCE OF OPERATORS

- Just like arithmetic operators, there is an operator precedence when evaluating logical operators as follows:
 1. Expressions in parentheses are processed (inside to outside)
 2. Negation
 3. AND
 4. OR
 5. Implication
 6. Biconditional
 7. Left to right
 - Use parentheses whenever you have any doubt!
-

PROPOSITIONAL LOGIC EXAMPLES

- **Example 1:** If it is humid, then it is raining.
 - P=It is humid. And Q=It is raining.
 - It is represented as $(P \rightarrow Q)$.
 - **Example 2:** It is noon and Ram is sleeping.
 - **Solution:** A= It is noon. And B= Ram is sleeping.
 - It is represented as $(A \wedge B)$.
 - **Example 3:** If it is raining, then it is not sunny.
 - **Solution:** P= It is raining. And Q= It is sunny.
 - It is represented as $P \rightarrow (\sim Q)$
 - **Example 4:** Ram is a man or a boy.
 - **Solution:** X= Ram is a man. And Y= Ram is a boy.
 - It is represented as $(X \vee Y)$.
 - **Example 5:** I will go to Delhi if and only if it is not humid.
 - **Solution:** A= I will go to Delhi. And B= It is humid.
 - It is represented as $(A \Leftrightarrow \sim B)$.
-

HOW CAN WE REPRESENT THE WUMPUS WORLD?

- We can represent the **Wumpus world** (things we know and things we discover) in terms of logic as follows:
- Consider the propositional symbols (partial formulation):
 - $P(i,j)$ is T if there is a **pit in (I,J)** , otherwise F.
 - $B(i,j)$ is T if there is a **breeze in (I,J)** , otherwise F.
- We can update as we explore:
 - $\neg B(1,1)$ – no breeze in square (1,1).
 - $B(2,1)$ – breeze in square (2,1).
 - $\neg P(1,1)$ - no pit in starting square.
- "Pits cause breezes in adjacent squares"
 - $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
 - $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$

LOGICAL EQUIVALENCE

- Two sentences are **logically equivalent**, denoted by $\alpha \equiv \beta$ iff they are true in the same models, i.e., iff: $\alpha \models \beta$ and $\beta \models \alpha$.
- If the **value of P** and **Q** is true in the same set of models, then they are said to be **logically equivalence**.
- It can be used as **inference rules in both directions**.

Example

- $(A \Rightarrow B) \equiv (\neg B \Rightarrow \neg A)$ (contraposition)

INFERENCE RULES WITH LOGICAL EQUIVALENCES

Rule Name	Rule
Idempotency Law	$(A \wedge A) \equiv A$ $(A \vee A) \equiv A$
Commutative Law	$(A \wedge B) \equiv (B \wedge A)$ $(A \vee B) \equiv (B \vee A)$
De morgan's Law	$\sim(A \wedge B) \equiv \sim A \vee \sim B$ $\sim(A \vee B) \equiv (\sim A \wedge \sim B)$
Associative Law	$A \vee (B \vee C) \equiv (A \vee B) \vee C$ $A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$
Distributive Law	$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$ $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
Contrapositive Law	$A \rightarrow B \equiv \sim B \rightarrow \sim A$
Implication Removal	$A \rightarrow B = \sim A \vee B$
Biconditional Removal	$A \leftrightarrow B = (A \rightarrow B) \wedge (B \rightarrow A)$
Absorption Law	$A \wedge (A \vee B) \equiv A$ $A \vee (A \wedge B) \equiv A$
Double-negation elimination	$\sim(\sim A) = A$

INFERENCE RULES IN PROPOSITION LOGIC

- **Inference rules** are those rules which are used to **describe certain conclusions**. The inferred conclusions lead to the desired goal state.
- In propositional logic, there are various inference rules which can be applied to prove the given statements and conclude them.

$$\frac{p \quad p \rightarrow q}{q}$$

Modus Ponens

$$\frac{\neg q \quad p \rightarrow q}{\neg p}$$

Modus Tollens

COMMON RULES

1. Addition:
$$\frac{p}{p \vee q}$$
 2. Simplification:
$$\frac{p \wedge q}{q} \quad p \vee q$$
 3. Disjunctive-syllogism:
$$\frac{\neg p}{q}$$
 4. Hypothetical-syllogism:
$$\frac{\begin{array}{c} p \rightarrow q \\ q \rightarrow r \end{array}}{p \rightarrow r}$$

Hypothetical Syllogism can be represented as: If $(P \rightarrow Q) \wedge (Q \rightarrow R) = (P \rightarrow R)$
 5. Introduction:
$$\frac{\begin{array}{c} A \\ B \\ \hline A \wedge B \end{array}}{A \wedge B}$$
 6. And Elimination:
$$\frac{A \wedge B}{A}$$
-

VALIDITY AND SATISFIABILITY

- **Validity:** If a sentence is **valid in all set of models**, then it is a valid sentence. Validity is also known as **tautology**, where it is necessary to have true value for each set of model.

Eg: $A \vee \neg A$, $A \Rightarrow A$,

- **Satisfiability:** If a sentence is true **atleast for some set of values**, it is a **satisfiable sentence**.
- It can be done by **truthtable enumeration**.
- $(P \vee Q) \rightarrow (P \wedge Q)$

P	Q	$P \vee Q$	$P \wedge Q$	$(P \vee Q) \rightarrow (P \wedge Q)$
False	False	False	False	True
False	True	True	False	False
True	False	True	False	False
True	True	True	True	True

- from the above truth table, it is clear that the given expression is satisfiable but not valid.

EXAMPLE 2:

- $((A \rightarrow B) \wedge A) \rightarrow B$

A	B	$A \rightarrow B$	$(A \rightarrow B) \wedge A$	$((A \rightarrow B) \wedge A) \rightarrow B$
False	False	True	False	True
False	True	True	False	True
True	False	False	False	True
True	True	True	True	True

- the given expression is valid as well as satisfiable.
-

LOGICAL INFERENCE PROBLEM

- Given a **knowledge base KB** (a set of sentences) and a **sentence α** (called a **theorem**). Does a **KB semantically entail α** ? In other words in all interpretations in which sentences in the **KB** are true, is also α true? Ie, **$KB \models \alpha$** ?
- Three approaches:
 - **Truth-table approach**
 - **Deduction using Inference rules**
 - **Proof by Contradiction or Resolution-refutation**

DEDUCTION THEOREM & PROOF BY CONTRADICTION

Deduction Theorem (connects inference and validity)

- $\text{KB} \models \alpha$ if and only if $\text{KB} \Rightarrow \alpha$ is valid

Proof By Contradiction or Refutation or reductio ad absurdum

- $\text{KB} \models \alpha$ is valid if and only if the sentence $\text{KB} \wedge \neg \alpha$ is a contradiction.
- **Monotonic**
 - If we have a proof, adding information to the DB will not invalidate the proof ie set of entailed sentences can only increase information to KB.

DEDUCTION EXAMPLE

- **P:** “It is hot”, **Q :** “It is humid” and **R :** “It is raining”. (SYMBOLS).
 - Given **KB** as:
 1. “If it is hot and humid, then it is raining”: $P \wedge Q \Rightarrow R$
 2. "If it is humid, then it is hot": $Q \Rightarrow P$
 3. “It is humid”: Q
 - **Question:** Is it raining? (i.e., is R entailed by KB?)
-

SOLUTION

Step		Reason
1	Q	(premise)
2	$Q \Rightarrow P$	(premise)
3	P	(modus ponens) (1,2)
4	$(P \wedge Q) \Rightarrow R$	(premise)
5	$P \wedge Q$	(and-intro) (1,3)
6	R	(and-elim) (4,5)

CHALLENGE

- Given KB.
 - $P \wedge Q$
 - $P \rightarrow R$
 - $Q \wedge R \rightarrow S$
 - Can you conclude S
-

SOLUTION

Step	Formula	Derivation
1	$P \wedge Q$	Given
2	$P \rightarrow R$	Given
3	$(Q \wedge R) \rightarrow S$	Given
4	P	1 And-Elim
5	R	4,2 Modus Ponens
6	Q	1 And-Elim
7	$Q \wedge R$	5,6 And-Intro
8	S	7,3 Modus Ponens

PROOF BY CONTRADICTION

- Assume our conclusion is false, and look for a contradiction. If found, the opposite of our assumption must be true.
1. Arrive at conclusion R

1	$P \vee Q$
2	$P \rightarrow R$
3	$Q \rightarrow R$

SOLUTION

Step	Formula	Derivation
1	$P \vee Q$	Given
2	$P \rightarrow R$	Given
3	$Q \rightarrow R$	Given
4	$\neg R$	Negated Conclusion
5	$Q \vee R$	1,2
6	$\neg P$	2,4
7	$\neg Q$	3,4
8	R	5,7
9	F	4,8

FORMALIZING THE WW IN PL

- The Wumpus World knowledge base:
 - There is no pit in [1, 1] (agent percept): $R_1 : \neg P_{11}$
 - A square is breezy if and only if there is a pit in a neighboring square. (Rule of the WW).
We state this for the square $B11$ only: $R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
 - There is no breeze in square [1; 1]. (agent percept) $R_3 : \neg B_{11}$
 - The agent can now use the PL inference rules and logical equivalences to prove the following: **There is no pit in squares [1,2] or [2, 1]**
 - **Theorem:** $\neg P_{12} \wedge \neg P_{21}$
-

FORMALIZING THE WW IN PL

- Apply biconditional elimination to R_2 :
 - $\textcolor{violet}{R4} : (\text{B11} \Rightarrow (\text{P12} \vee \text{P21})) \wedge ((\text{P12} \vee \text{P21}) \Rightarrow \text{B11})$
 - Apply And-elimination to R_4 :
 - $\textcolor{violet}{R5} : (\text{P12} \vee \text{P21}) \Rightarrow \text{B11}$
 - Apply logical equivalence for contrapositives to R_5 :
 - $\textcolor{violet}{R6} : \neg \text{B11} \Rightarrow \neg(\text{P12} \vee \text{P21})$
 - Apply modus ponens to R_6 and R_3 :
 - $\textcolor{violet}{R7} : \neg(\text{P12} \vee \text{P21})$
 - Apply de Morgan's rule to $R7$:
 - $\textcolor{violet}{R8} : \neg \text{P12} \wedge \neg \text{P21}$
-

KB IN RESTRICTED FORMS

- If the sentences in the KB are restricted to some special forms some of the sound inference rules may become complete
- **Example:**
- **Horn form (Horn normal form)**
- **CNF (Conjunctive Normal Forms)**

PROPOSITIONAL THEOREM PROVING

- Search for proofs is a more efficient way than enumerating models (We can ignore irrelevant information). Truth tables have an exponential number of models.
- The idea of inference is to repeat applying inference rules to the KB.
- Inference can be applied whenever suitable premises are found in the KB
- **Theorem proving means to apply rules of inference directly to the sentences.**
- Two ways to ensure completeness:
 1. **Proof by resolution:** use sequence of powerful inference rules (resolution rule) and construction of / search for a proof. **Resolution** works best when the formula is of the special form **CNF**.
Properties
 - Typically requires translation of sentences into a normal form.
 2. **Forward or Backward chaining:** use of modus ponens on a restricted form of propositions (**Horn clauses**)

NORMAL FORMS

- **Literal:** A literal is an atomic sentence (propositional symbol), or the negation of an atomic sentence. Eg:- p (positive literal), $\neg p$ (negative literal)
- **Clause:** A disjunction of literals. Eg:- $\neg p \vee q$
- **Conjunctive Normal Form (CNF):** A conjunction of disjunctions of literals, i.e., a conjunction of clauses Eg:- $(AV\neg B) \wedge (BV\neg C) \wedge (CV\neg D)$
- **DNF(Disjunctive Normal Form):** This is a reverse approach of CNF which is disjunction of conjunction of literals. Eg:- $(A_1 \wedge B_1) \vee (A_2 \wedge B_2) \vee \dots \vee (A_n \wedge B_n)$
- In **DNF**, it is **OR of AND's**, a **sum of products**, or a **cluster concept**, whereas, in **CNF**, it is **ANDs of OR's** a **product of sums**.

CNF TRANSFORMATION

- In propositional logic, the resolution method is applied only to those clauses which are disjunction of literals. **There are following steps used to convert into CNF:**

1) Eliminate **bi-conditional implication** by replacing $A \Leftrightarrow B$ with $(A \rightarrow B) \wedge (B \rightarrow A)$

2) Eliminate **implication** by replacing $A \rightarrow B$ with $\neg A \vee B$.

3) In CNF, negation(\neg) appears only in literals, therefore we **move negation inwards** as:

$$\neg(\neg A) \equiv A \text{ (double-negation elimination)}$$

$$\neg(A \wedge B) \equiv (\neg A \vee \neg B) \text{ (De Morgan)}$$

$$\neg(A \vee B) \equiv (\neg A \wedge \neg B) \text{ (De Morgan)}$$

4) Finally, **using distributive law** on the sentences, and form the CNF as:

$$(A_1 \vee B_1) \wedge (A_2 \vee B_2) \wedge \dots \wedge (A_n \vee B_n).$$

- **Note: CNF can also be described as AND of ORS**
 - Transform to CNF: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
-

CNF TRANSFORMATION EXAMPLE

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

1. Eliminate \Leftrightarrow , replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$

2. Eliminate \Rightarrow , replacing $\alpha \Rightarrow \beta$ with $\neg \alpha \vee \beta$.

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$$

3. Move \neg inwards using de Morgan's rules and double-negation:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$$

4. Apply distributivity law (\wedge over \vee) and flatten:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$$

METHOD 1: RESOLUTION METHOD IN FOL

- In propositional logic, **resolution method is by application of inference rule gives a new clause when two or more clauses and are coupled together to prove theorem.**

The resolution algorithm tries to prove:

$KB \models \alpha$ equivalent to

$KB \wedge \neg \alpha$ unsatisfiable

- Using propositional resolution, it becomes easy to make a theorem prover sound and complete for all. **The process followed to convert the propositional logic into resolution method is known as Resolution refutation contains the below steps:**

1. Convert the given axiom(all sentences) into clausal form, CNF.
 2. Negate the desired conclusion (converted to CNF)
 3. Apply resolution rule until either – Derive false (a contradiction) – Can't apply any more
 4. If we derive a contradiction, then the conclusion follows from the axioms
 5. If we can't apply any more, then the conclusion cannot be proved from the axioms.
- This is known as **resolution Algorithm**.
 - **Resolution**

APPLYING RESOLUTION

$$(A \vee B \vee C)$$

$$(\neg A)$$

$$\therefore (B \vee C)$$

“If A or B or C is true, but not A, then B or C must be true.”

$$(A \vee B \vee C)$$

$$(\neg A \vee D \vee E)$$

$$\therefore (B \vee C \vee D \vee E)$$

“If A is false then B or C must be true, or if A is true then D or E must be true, hence since A is either true or false, B or C or D or E must be true.”

$$(A \vee B)$$

$$(\neg A \vee B)$$

$$\therefore (B \vee B) = B$$

Simplification

EXAMPLE

- Prove R from:

1	$(P \rightarrow Q) \rightarrow Q$
2	$(P \rightarrow P) \rightarrow R$
3	$(R \rightarrow S) \rightarrow \neg(S \rightarrow Q)$

SOLUTION

- Convert to CNF

$$1. \quad (\mathbf{P} \rightarrow \mathbf{Q}) \rightarrow \mathbf{Q} \equiv \neg(\neg \mathbf{P} \vee \mathbf{Q}) \vee \mathbf{Q}$$

$$\equiv (\mathbf{P} \neg \mathbf{Q}) \vee \mathbf{Q}$$

$$\equiv (\mathbf{P} \vee \mathbf{Q}) \wedge (\neg \mathbf{Q} \vee \mathbf{Q})$$

$$\equiv (\mathbf{P} \vee \mathbf{Q}) \wedge \mathbf{T}$$

$$2. \quad (\mathbf{P} \rightarrow \mathbf{P}) \rightarrow \mathbf{R} \equiv \neg(\neg \mathbf{P} \vee \mathbf{P}) \vee \mathbf{R}$$

$$\equiv (\mathbf{P} \neg \mathbf{P}) \vee \mathbf{R}$$

$$\equiv (\mathbf{P} \vee \mathbf{R}) \wedge (\neg \mathbf{P} \vee \mathbf{R})$$

$$3. \quad (\mathbf{R} \rightarrow \mathbf{S}) \rightarrow \neg(\mathbf{S} \rightarrow \mathbf{Q}) \equiv \neg(\neg \mathbf{R} \vee \mathbf{S}) \vee \neg(\neg \mathbf{S} \vee \mathbf{Q})$$

$$\equiv (\mathbf{R} \wedge \neg \mathbf{S}) \vee (\mathbf{S} \wedge \neg \mathbf{Q})$$

$$\equiv (\mathbf{R} \vee \mathbf{S}) \wedge (\neg \mathbf{S} \vee \mathbf{S}) \wedge (\mathbf{R} \vee \neg \mathbf{Q}) \wedge (\neg \mathbf{S} \vee \neg \mathbf{Q})$$

$$\equiv (\mathbf{R} \vee \mathbf{S}) \wedge \mathbf{T} \wedge (\mathbf{R} \vee \neg \mathbf{Q}) \wedge (\neg \mathbf{S} \vee \neg \mathbf{Q})$$

1	$\mathbf{P} \vee \mathbf{Q}$	
2	$\mathbf{P} \vee \mathbf{R}$	
3	$\neg \mathbf{P} \vee \mathbf{R}$	
4	$\mathbf{R} \vee \mathbf{S}$	
5	$\mathbf{R} \vee \neg \mathbf{Q}$	
6	$\neg \mathbf{S} \vee \neg \mathbf{Q}$	
7	$\neg \mathbf{R}$	Neg
8	\mathbf{S}	4,7
9	$\neg \mathbf{Q}$	6,8
10	\mathbf{P}	1,9
11	\mathbf{R}	3,10
12	\mathbf{F}	7,11

PROPOSITIONAL RESOLUTION EXAMPLE

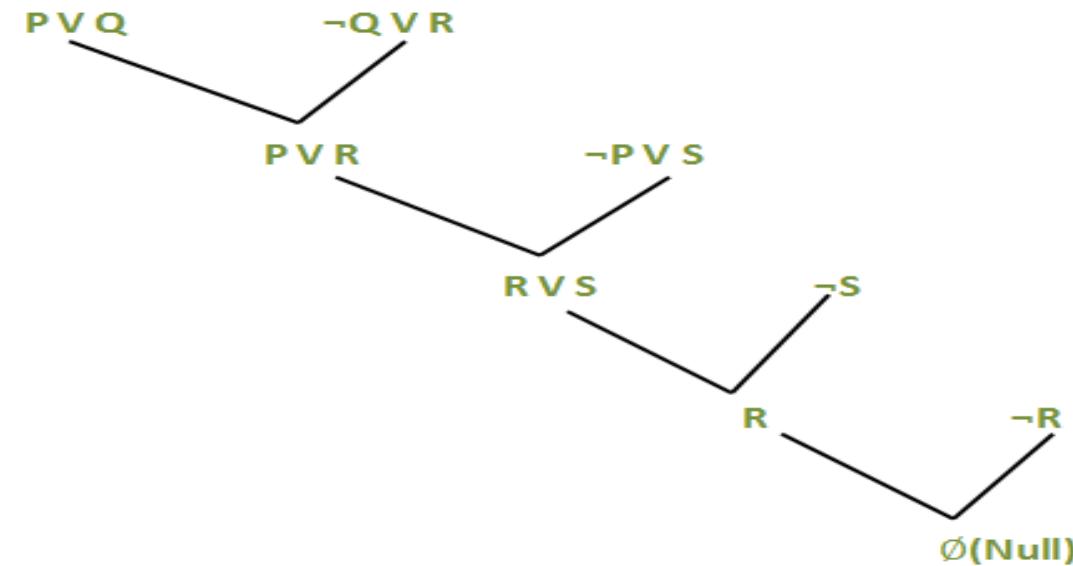
- Consider the following Knowledge Base:
 1. The humidity is high or the sky is cloudy.
 2. If the sky is cloudy, then it will rain.
 3. If the humidity is high, then it is hot.
 4. It is not hot.
- **Goal:** It will rain.
- Use propositional logic and apply resolution method to prove that the goal is derivable from the given knowledge base.

SOLUTION

- **Solution:** Let's construct propositions of the given sentences one by one:

Let, **P**: Humidity is high. **Q**: Sky is cloudy. **R**: It will rain **S**: It is hot.

1. It will be represented as **P V Q**.
2. It will be represented as **Q → R**.
3. It will be represented as **P → S**.
4. It will be represented as **¬S**.



CHALLENGES

- Given $\text{KB} = (\mathbf{B}_{1,1} \Leftrightarrow (\mathbf{P}_{1,2} \vee \mathbf{P}_{2,1})) \wedge \neg \mathbf{B}_{1,1}$ Prove $\alpha: \neg \mathbf{P}_{1,2}$

SOLUTION

Given KB

- **R1:** $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$
- **R2:** $\neg B_{1,1}$
- **R3:** Negation of theorem = $\neg(\neg P_{1,2}) = P_{1,2}$
- Given R1 can be split up as **R4:** $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ **R5:** $(\neg P_{1,2} \vee B_{1,1})$ **R6:** $(\neg P_{2,1} \vee B_{1,1})$
- Consider **R5** and **R2** apply Modus Ponens **R6:** $\neg P_{1,2}$
- Consider R6 and R3 which leads to a negation

HORN CLAUSES AND DEFINITE CLAUSES

- **DEFINITE CLAUSE:** A disjunction of literals of which **exactly one** is positive.
 - $(\neg L_{1,1} \vee \neg \text{breeze} \vee B_{1,1})$ **Yes**
 - $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ **No**
 - **HORN CLAUSE:** A disjunction of literals of which **atmost one** is positive, ie is a CNF clause with exactly one positive literal. The positive literal is called the **head**. The negative literals are called the **body**. All **definite clauses are Horn Clauses**.
 - $(\neg L_{1,1} \vee \neg \text{breeze} \vee B_{1,1})$ **Yes**
 - $(\neg B_{1,1} \vee \neg P_{1,2} \vee \neg P_{2,1})$ **Yes**
 - $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ **No**
 - Horn clauses are **closed under resolution**, ie if 2 Horn closes are resolved we get back a horn clause.
 - Not all sentences in propositional logic can be converted into the Horn form
 - **GOAL CLAUSE:** A clause with **no** positive literal.
 - $(\neg L_{1,1} \vee \text{breeze} \vee B_{1,1})$ **No**
 - $(\neg B_{1,1} \vee \neg P_{1,2} \vee \neg P_{2,1})$ **Yes**
-

HORN CLAUSES

- **Horn clauses** can be re-written as **implications ie**, logic proposition of the form: $p_1 \wedge \dots \wedge p_n \rightarrow q$.
- Eg: $\neg C \vee \neg B \vee A$ can be written as $C \wedge B \rightarrow A$
- **KB = conjunction of Horn clauses.**
- Modus Ponens (for Horn Form)

$$\frac{B \Rightarrow A, \quad B}{A}$$

– More general version of the rule:

$$\frac{(B_1 \wedge B_2 \wedge \dots \wedge B_k \Rightarrow A), B_1, B_2, \dots, B_k}{A}$$

- Inference with **Horn Clauses** can be done using **forward and backward chaining algorithms**.
 - The **Prolog language** is based on **Horn Clauses**.
 - Deciding entailment with Horn Clauses is *linear in the size of the knowledge base*.
-

FORWARD AND BACKWARD CHAINING

- These algorithms are very natural and run in linear time

FORWARD CHAINING:

- Based on rule of **modus ponens**. If know P_1, \dots, P_n & know $(P_1 \wedge \dots \wedge P_n) \rightarrow Q$. Then can conclude Q . Whenever the premises of a rule are satisfied, infer the conclusion. Continue with rules that became satisfied.
- Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine. Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached.

BACKWARD CHAINING:

- In Backward chaining, we will start with our goal predicate and then infer further rules.
 - Search start from the query and go backwards.
-

FORWARD CHAINING

- **IDEA:** It begins from facts(positive literals) in knowledge base and determines if the query can be entailed by knowledge base of definite clauses. If all premises of an implication are known its conclusion is added to set of known facts. Eg: Given $L_{1,1}$ and **Breeze** and $(L_{1,1} \wedge \text{Breeze}) \rightarrow B_{1,1}$ is in knowledge base then $B_{1,1}$ can be added.
- Every inference is an application of **modus ponens** ie
$$\frac{p_1, \dots, p_n}{q} \quad p_1 \wedge \dots \wedge p_n \rightarrow q$$
 Can be used with forward chaining.

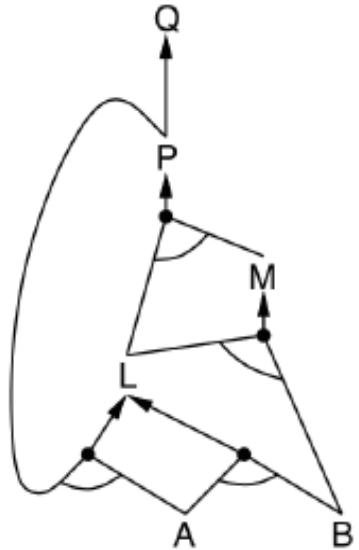
FORWARD CHAINING STEPS

1. Start with given proposition symbols (atomic sentence).
 2. Iteratively try to infer truth of additional proposition symbols
 3. Continue until
 - no more inference can be carried out, or
 - goal is reached
-

FORWARD CHAINING

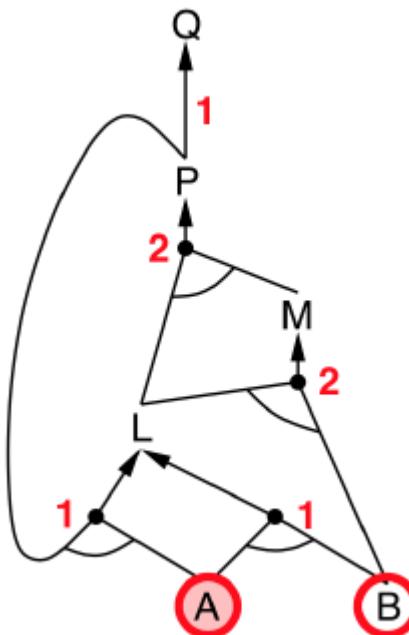
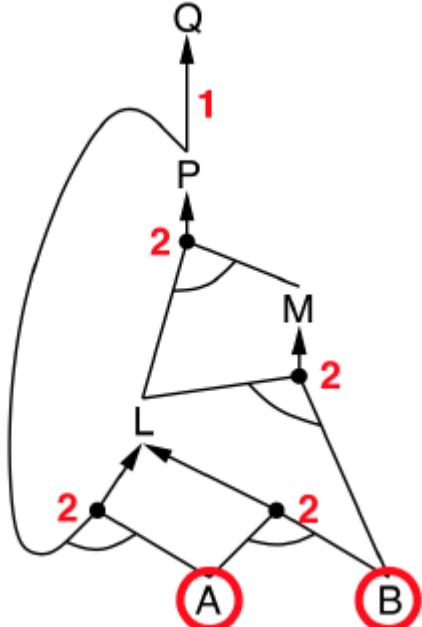
- Fire any rule whose premises are satisfied in the KB, add its conclusion to the KB, until query is found
- **AND-OR graph:** Multiple links joined by an arc indicates a conjunction where every link has to be proved, while multiple links without an arc indicates disjunction, where any link has to be proved.
- This proves that $\text{KB} \rightarrow Q$ is true in all possible worlds (i.e. trivial), and hence it proves entailment.

$P \Rightarrow Q$
 $L \wedge M \Rightarrow P$
 $B \wedge L \Rightarrow M$
 $A \wedge P \Rightarrow L$
 $A \wedge B \Rightarrow L$
 A
 B

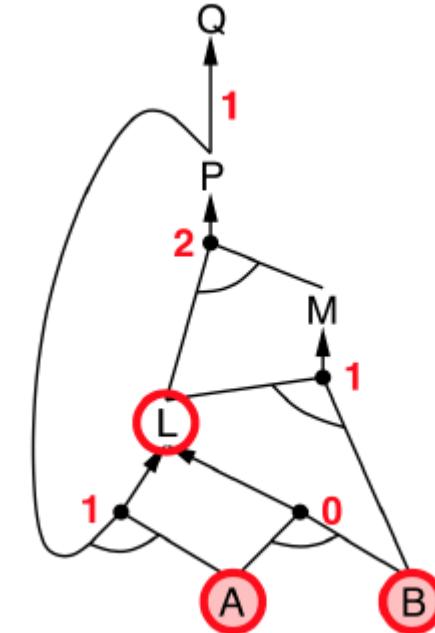


**AND-OR
GRAPH**

FORWARD CHAINING

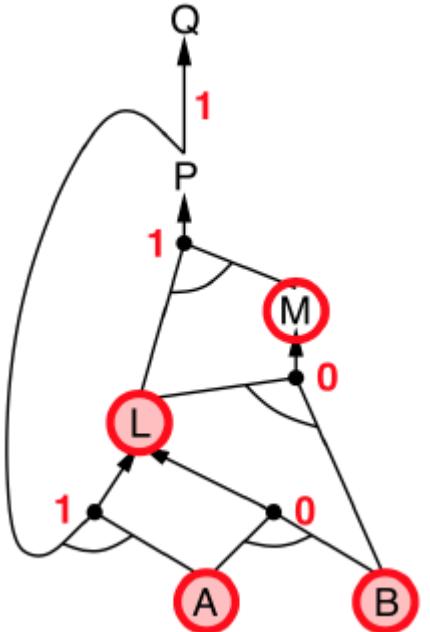


- Process agenda item A
- Decrease count for horn clauses in which A is premise

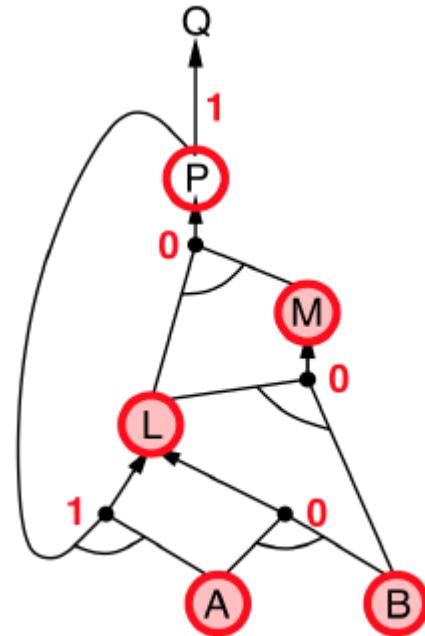


- Process agenda item B
- Decrease count for horn clauses in which B is premise
- $A \wedge B \rightarrow L$ has now fulfilled premise
- Add L to agenda

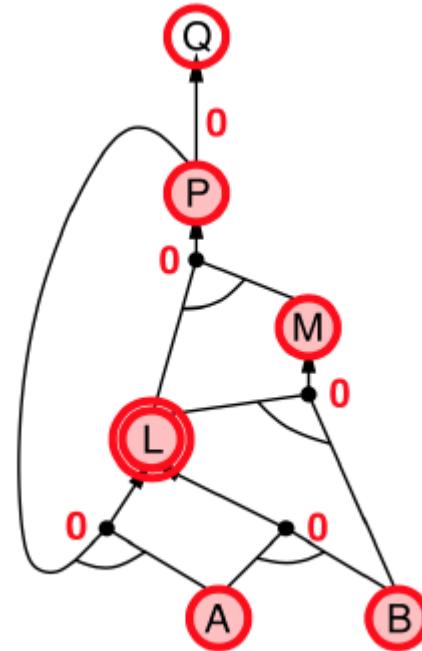
FORWARD CHAINING



- Process agenda item L
- Decrease count for horn clauses in which L is premise
- $B \wedge L \rightarrow M$ has now fulfilled premise
- Add M to agenda

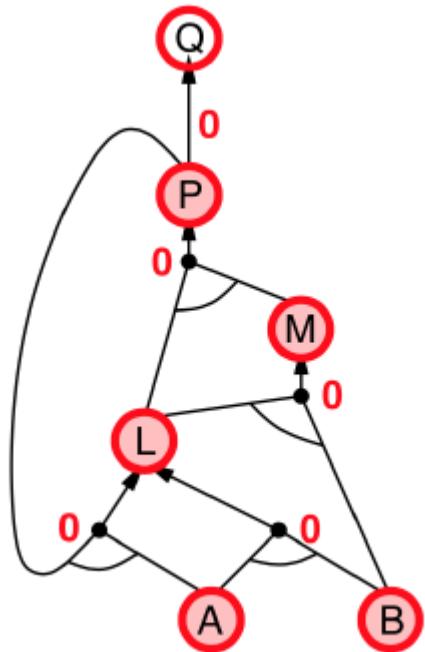


- Process agenda item M
- Decrease count for horn clauses in which M is premise
- $L \wedge M \rightarrow P$ has now fulfilled premise
- Add P to agenda

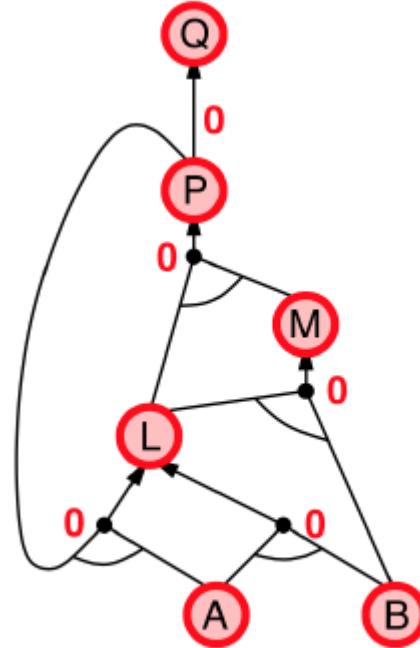


- Process agenda item P
- Decrease count for horn clauses in which P is premise
- $P \rightarrow Q$ has now fulfilled premise
- Add Q to agenda
- $A \wedge P \rightarrow L$ has now fulfilled premise

FORWARD CHAINING



- Process agenda item P
- Decrease count for horn clauses in which P is premise
- $P \rightarrow Q$ has now fulfilled premise
- Add Q to agenda
- $A \wedge P \rightarrow L$ has now fulfilled premise
- But L is already inferred



- Process agenda item Q
- Q is inferred
- Done

FORWARD CHAINING CHALLENGE

Assume the KB with the following rules and facts:

KB: R1: $A \wedge B \Rightarrow C$

R2: $C \wedge D \Rightarrow E$

R3: $C \wedge F \Rightarrow G$

F1: A

F2: B

F3: D

Theorem: E ?

SOLUTION

Theorem: E

KB: R1: $A \wedge B \Rightarrow C$

R2: $C \wedge D \Rightarrow E$

R3: $C \wedge F \Rightarrow G$

F1: A

F2: B

F3: D

Rule R1 is satisfied.

F4: C

Rule R2 is satisfied.

F5: E

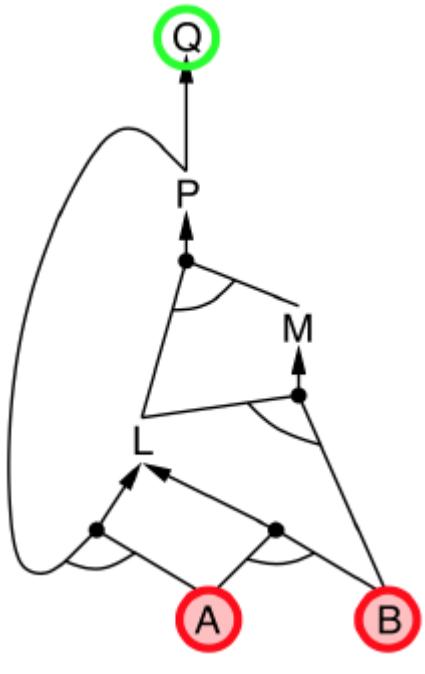


BACKWARD CHAINING

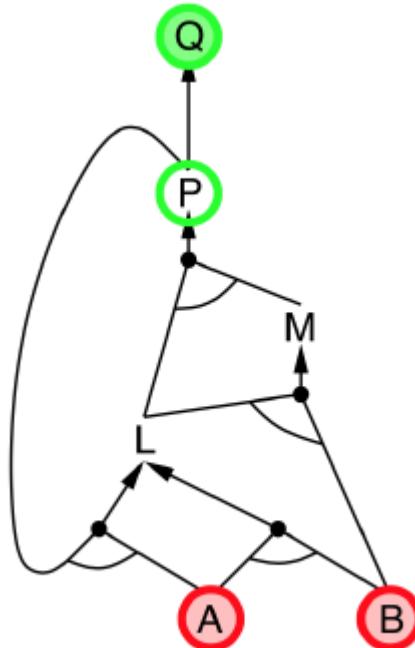
- **Idea:** Works backwards from the query q
- To prove q by **Backward Chaining**:
 1. Check if q is known already or exist in KB, if so return True or
 2. Prove by Backward Chaining all premises of some rule concluding q ie, . Find all implications, I , whose conclusion “matches” q
 3. Avoid loops: check if new subgoal is already on the goal stack
 4. Avoid repeated work: check if new subgoal has already been proved true, or has already failed. Recursively establish the premises of all i in I via backward chaining.

NOTE: Avoids inferring unrelated facts.

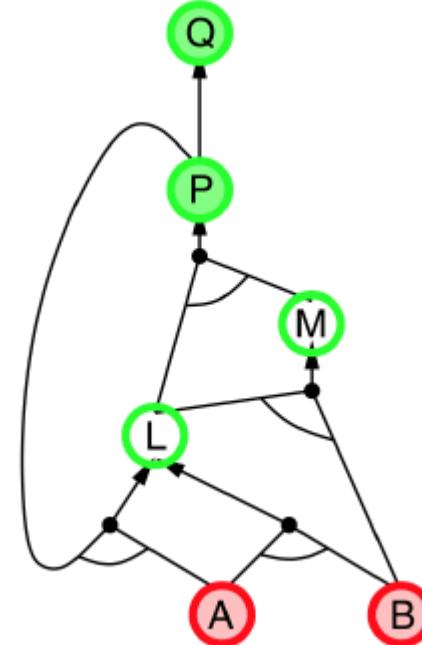
BACKWARD CHAINING



$P \Rightarrow Q$ ←
 $L \wedge M \Rightarrow P$
 $B \wedge L \Rightarrow M$
 $A \wedge P \Rightarrow L$
 $A \wedge B \Rightarrow L$
 A ←
 B ←



$P \Rightarrow Q$ ←
 $L \wedge M \Rightarrow P$ ←
 $B \wedge L \Rightarrow M$
 $A \wedge P \Rightarrow L$
 $A \wedge B \Rightarrow L$
 A ←
 B ←



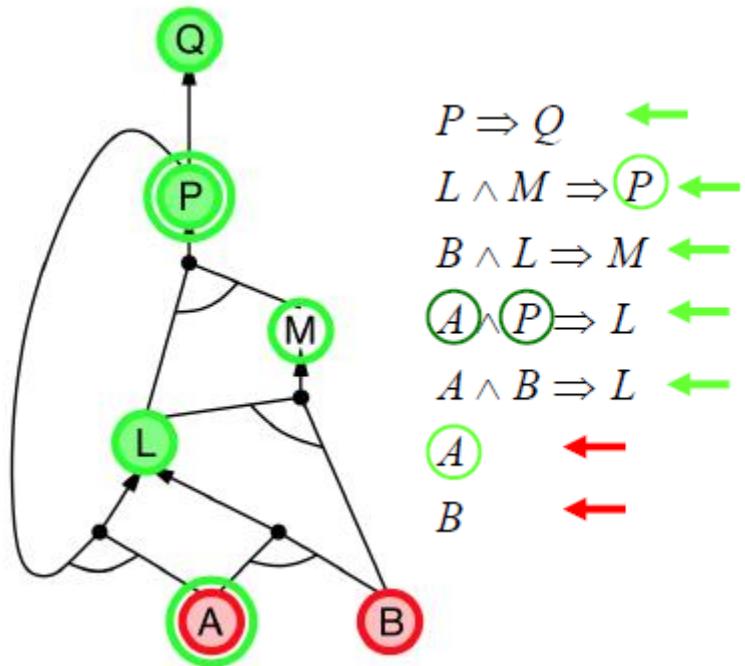
$P \Rightarrow Q$ ←
 $L \wedge M \Rightarrow P$ ←
 $B \wedge L \Rightarrow M$ ←
 $A \wedge P \Rightarrow L$ ←
 $A \wedge B \Rightarrow L$ ←
 A ←
 B ←

- A and B are known to be true
- Q needs to be proven

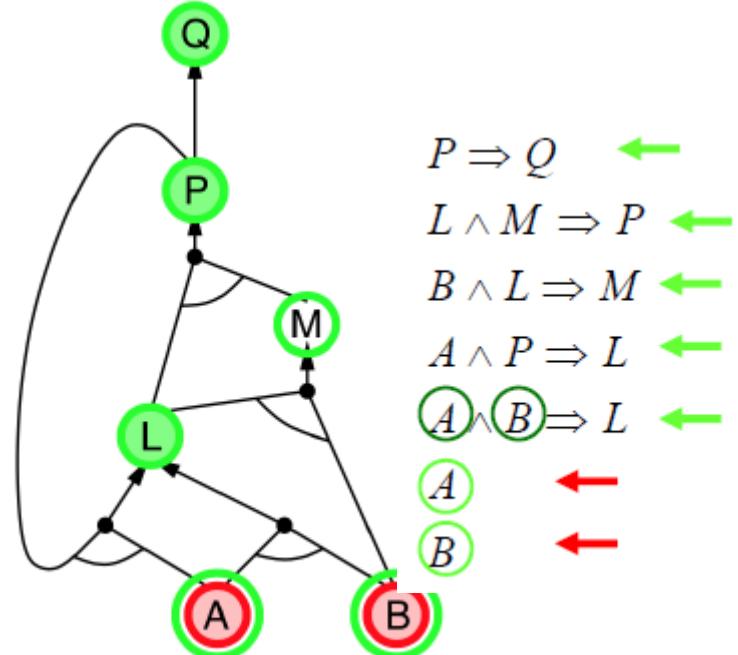
- Current goal: Q
- Q can be inferred by $P \rightarrow Q$
- P needs to be proven

- Current goal: P
- P can be inferred by $L \wedge M \rightarrow P$
- L and M need to be proven

BACKWARD CHAINING

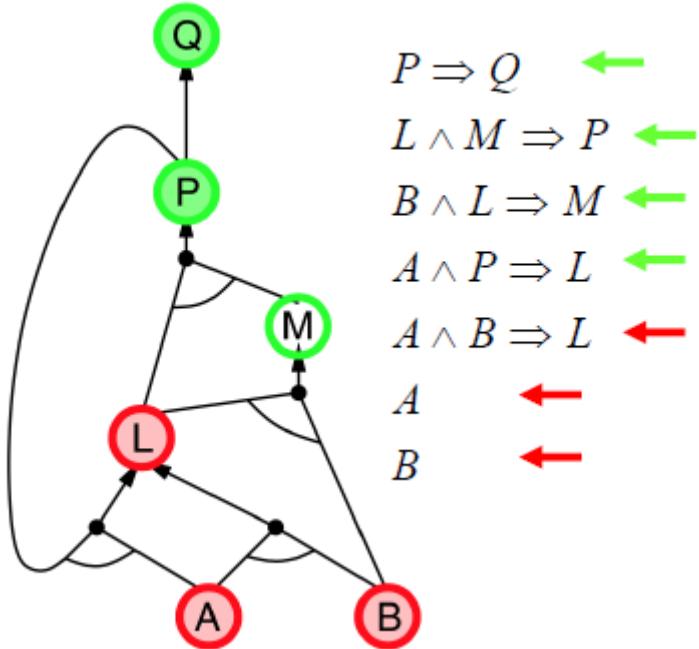


- Current goal: L
- L can be inferred by $A \wedge P \rightarrow L$
- A is already true
- P is already a goal
- repeated sub-goal

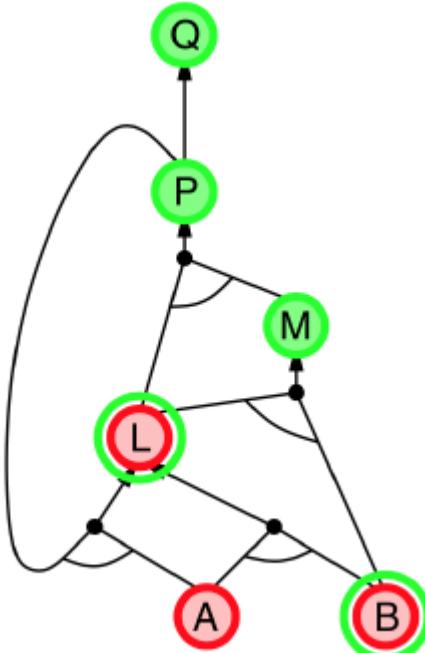


- Current goal: L
- L can be inferred by $A \wedge B \rightarrow L$
- Both are true

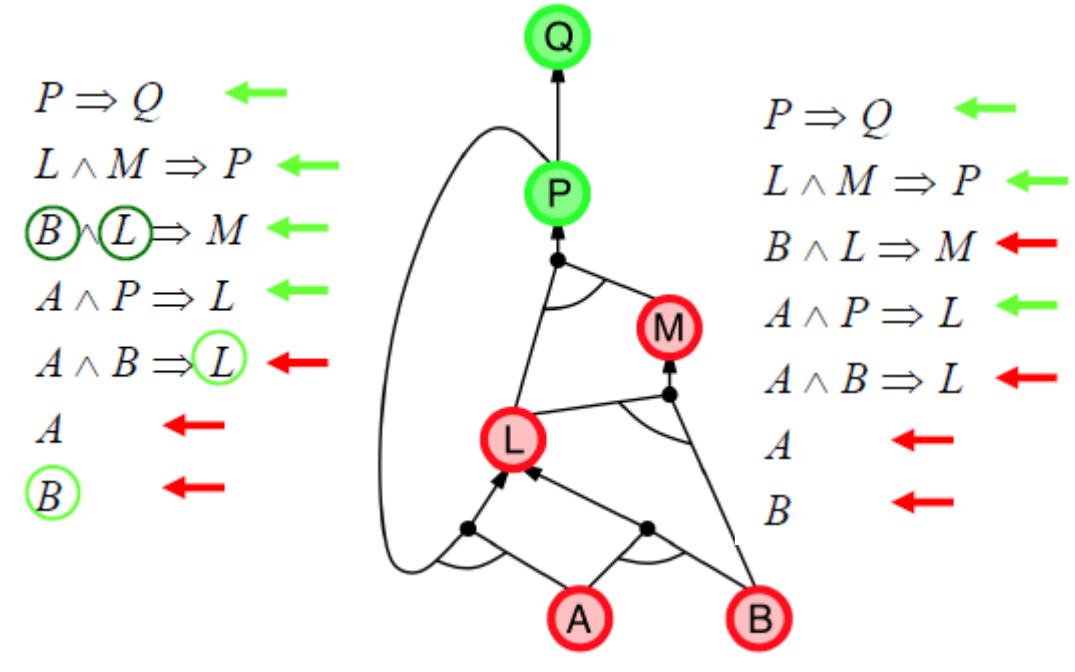
BACKWARD CHAINING



- Current goal: L
- L can be inferred by $A \wedge B \rightarrow L$
- Both are true
- L is true
- Current goal: M

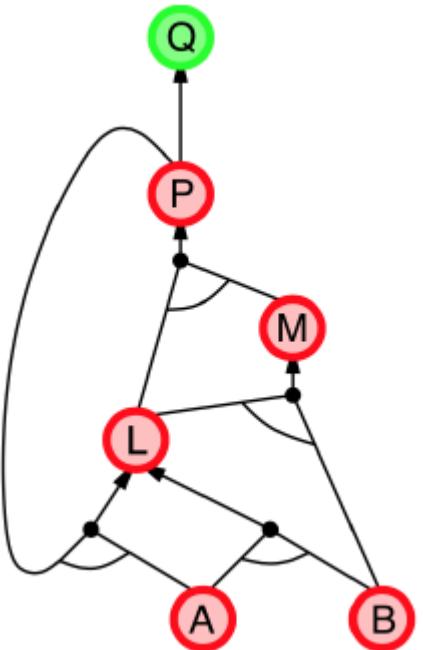


- Current goal: M
- M can be inferred by $B \wedge L \rightarrow M$

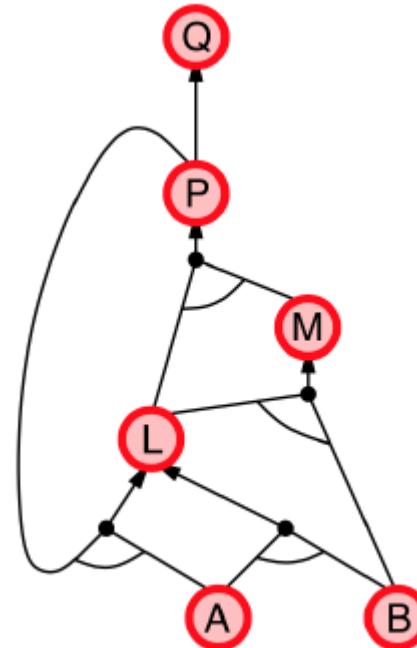


- Current goal: M
- M can be inferred by $B \wedge L \rightarrow M$
- Both are true
- M is true

BACKWARD CHAINING



$P \Rightarrow Q$ ←
 $L \wedge M \Rightarrow P$ ←
 $B \wedge L \Rightarrow M$ ←
 $A \wedge P \Rightarrow L$ ←
 $A \wedge B \Rightarrow L$ ←
 A ←
 B ←



$P \Rightarrow Q$ ←
 $L \wedge M \Rightarrow P$ ←
 $B \wedge L \Rightarrow M$ ←
 $A \wedge P \Rightarrow L$ ←
 $A \wedge B \Rightarrow L$ ←
 A ←
 B ←

- Current goal: P
- P can be inferred by $L \wedge M \rightarrow P$
- Both are true
- P is true

- Current goal: Q
- Q can be inferred by $P \rightarrow Q$
- P is true
- Q is true

BACKWARD CHAINING EXAMPLE

Problem KB:

R1: $A \wedge B \Rightarrow C$

R2: $C \wedge D \Rightarrow E$

R3: $C \wedge F \Rightarrow G$

F1: A

F2: B

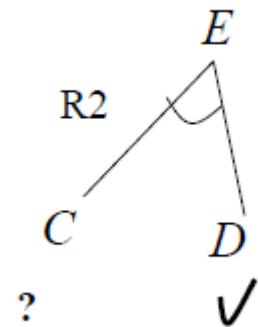
F3: D

Theorem: E

- Goal: prove the theorem, try to be more theorem driven
-

BACKWARD CHAINING EXAMPLE

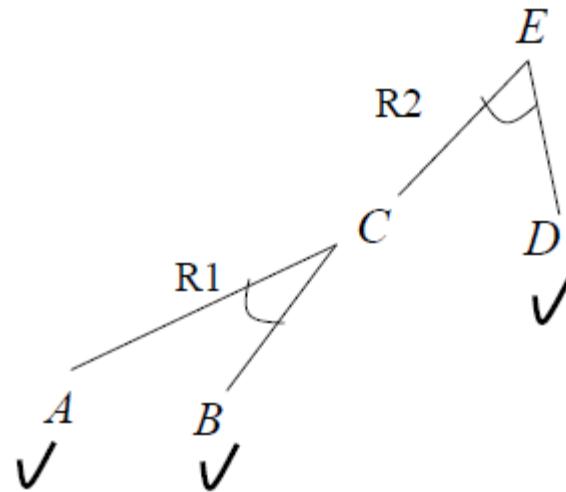
- Backward chaining tries to prove a theorem
- Procedure idea:
 - Checks if the theorem is true
 - If not, find the rule with the theorem in its conclusion and try to prove its premises



R1: $A \wedge B \Rightarrow C$
R2: $C \wedge D \Rightarrow E$
R3: $C \wedge F \Rightarrow G$
F1: A
F2: B
F3: D
Theorem: E

BACKWARD CHAINING EXAMPLE

- Backward Chaining is Theorem driven.



FORWARD VS BACKWARD

Forward chaining:

- Data-driven, automatic, unconscious processing.
- May do lots of work that is irrelevant to the goal

Backward chaining:

- Goal-driven, appropriate for problem-solving.
 - Complexity of BC can be much less than linear in size of KB
-

DISADVANTAGES OF PL

- Consider now the following WW rule: If a square has no smell, then neither the square nor any of its adjacent squares can house a Wumpus. How can we formalize this rule in PL?
- We have to write one rule for every relevant square! For example: $\neg S11 \Rightarrow \neg W11 \wedge \neg W12 \wedge \neg W21$
- For an example having large environment say we have a vacuum cleaner (Roomba) to clean a 1010 squares in the classroom. Use PL to express information about the squares.
- This is a very disappointing feature of PL. There is no way in PL to make a statement referring to all objects of some kind (e.g., to all squares).

LIMITATION

1. PL is not expressive enough to describe all the world around us. It can't express information about different object and the relation between objects.
 2. PL is not compact. It can't express a fact for a set of objects without enumerating all of them which is sometimes impossible.
 3. Propositional logic is declarative: pieces of syntax correspond to facts
 - Not to worry: this can be done in First order logic!
-

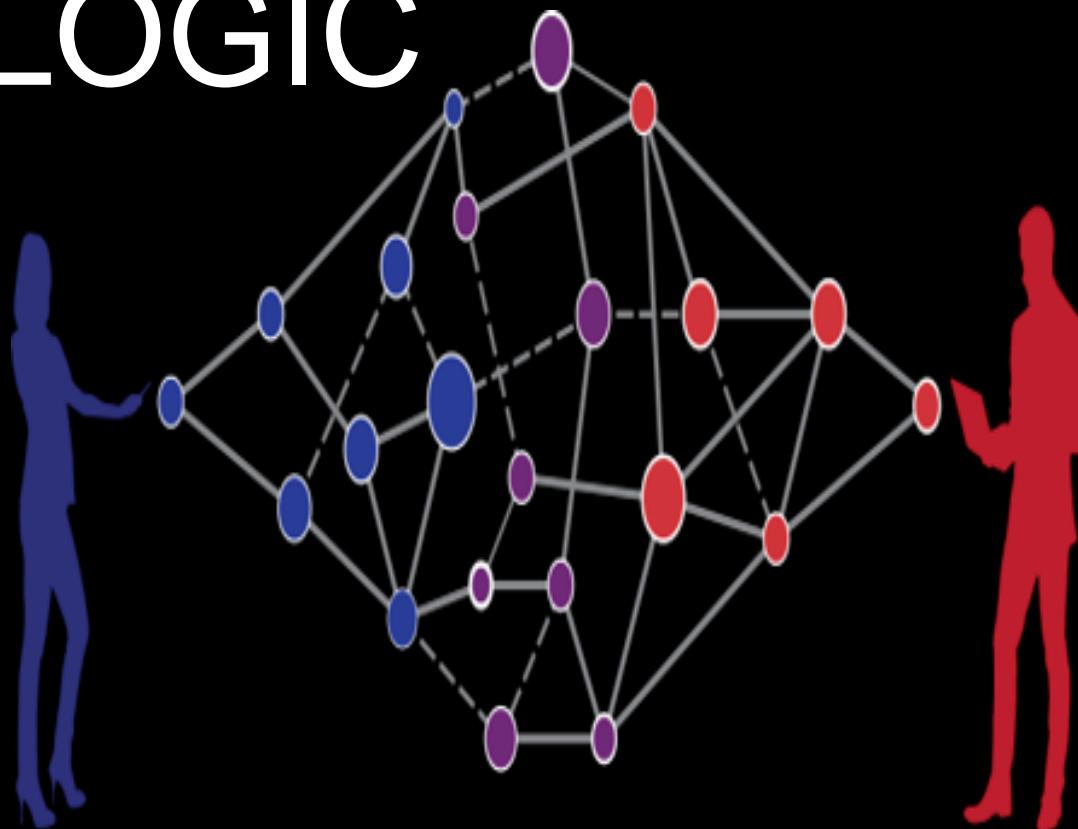
anooja@somaiya.edu

**THANK YOU
ANY QUESTIONS?**

THANK YOU



FIRST ORDER LOGIC



Prepared By
-Anooja Joy



FIRST-ORDER LOGIC

- **Propositional logic** Assumes that the world contains **facts** of from **yes or no**.
 - There is a need for a language that is **declarative**(knowledge + inference based on truth relation between sentences), **compositional**(meaning of a sentence is meaning of its parts), **context-independent**, **more expressive**(to deal with partial information using disjunction and negation) and **unambiguous**.
 - First-order logic is also called **Predicate logic** and **First-order predicate calculus (FOPL)**. It is a formal representation of logic in the form of **quantifiers**. In predicate logic, the **input** is taken as an **entity**, and the **output** it gives is either **true or false**.
 - **First-order logic** Assumes that the world contains
 - **Objects** people, houses, numbers, theories, Donald Duck, colors, centuries, ...
 - **Relations** red, round, prime, multistoried, ... brother of, bigger than, part of, has color, occurred after, owns
 - **Functions** +, middle of, father of, one more than, beginning of, ...
-

FORMAL LANGUAGES: ONTOLOGY AND EPISTEMOLOGY

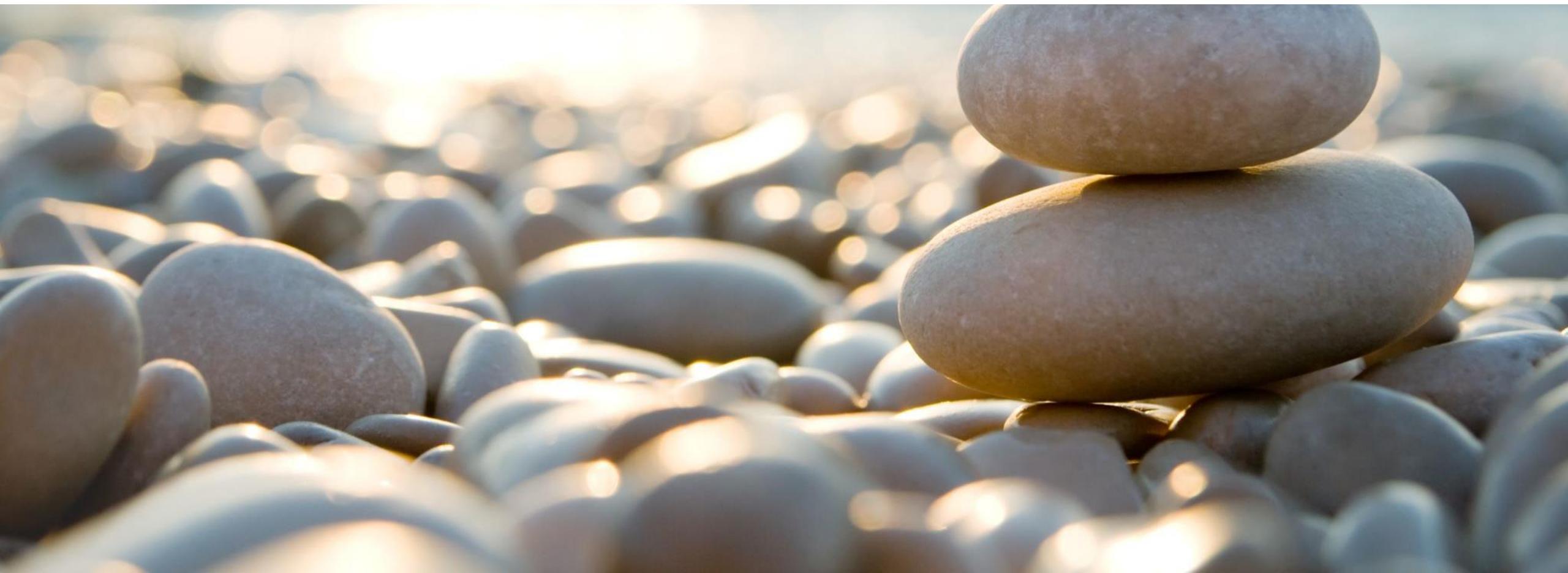
- **Ontology** is the study of existence claim ie, what it assumes about **nature of reality**. Ie, an inventory of what exists. Ontological commitment means **“WHAT EXISTS IN THE WORLD”**.
- **Epistemology** is a major branch of philosophy that concerns the forms, nature, and preconditions of knowledge. Epistological commitment is a commitment that considers **possible states of knowledge** with respect to each fact. Epistemological commitment means **“WHAT AN AGENT BELIEVES ABOUT FACTS”**.

Language	Ontological Commitment	Epistemological Commitment
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 $\in [0, 1]$
Fuzzy logic	degree of truth $\in [0, 1]$	known interval value

CONCEPTUALIZATION

- One plus two equals three
 - **Objects:** one, two, three, one plus two
 - **Function:** plus
 - **Relation:** equals
 - Squares neighboring the wumpus are smelly
 - **Objects:** wumpus, square
 - **Property:** smelly
 - **Relations:** neighboring
 - Evil King John ruled England in 1200
 - **Objects:** John, England, 1200
 - **Properties:** evil, king
 - **Relations:** ruled
-

SYNTAX AND SEMANTICS OF FOL



MODELS FOR FOL

- **Model** contains **domain elements** and **relations** among them.
 - Models for **propositional logic** are just **sets of truth values** for the proposition symbols.
 - Models for **first-order logic** have **objects, relations** and **functions**.
 - **Syntax:** It defines the way of representing the **given predicates**. As these predicates are represented via **quantifiers**, there are different types of quantifiers used.
 - **Semantics:** It defines the sense of the given predicate. It allows to make more logical expression by devising its semantics. Semantics allow us to understand the sentence meaning.
 - **Predicates:** express relationships between objects
 - **Quantifiers:** Quantifiers are used to express properties of entire collections of objects, instead of enumerating the objects by name if a logic that allows object is found. It can restrict the scope of variables
 - In FOL, **variables** refer to things in the world and can quantify over the-talk about all or some of them without naming them explicitly.
-

MODELS FOR FOL

Constants: *KingJohn, Richard*

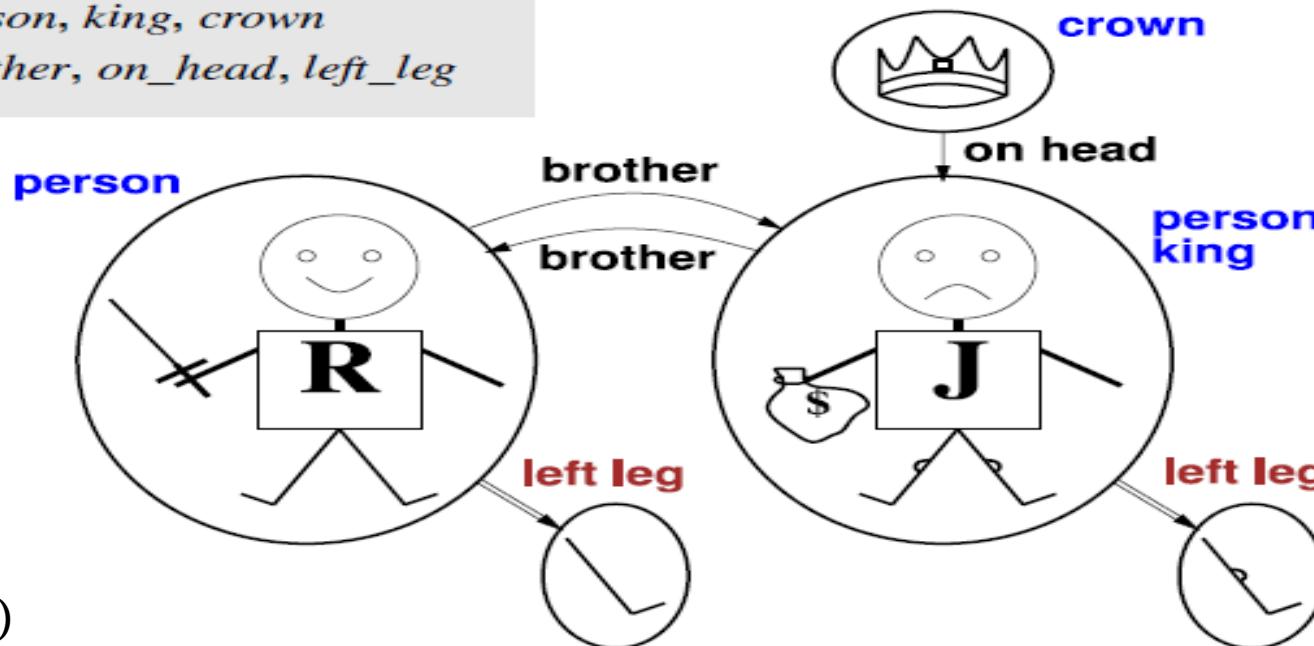
Predicates: *person, king, crown*

Functions: *brother, on_head, left_leg*

Objects (Ex:- Richard)

Relations (Ex:- King)

functions (Ex:- LeftLeg)



- The picture depicts Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown.
-

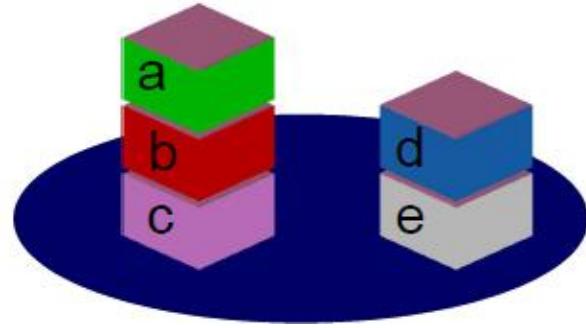
MODELS FOR FOL: OBJECTS

- The model contain **5 objects**, 2 **binary relations**(brother, onhead), 3 **unary relations**(person, person king, crown) and 1 **unary function**(left leg)
- **Objects** refers to an entity that exists in the real world.
- The picture shows a model with **five objects**:
 - Richard the Lionheart
 - His younger brother
 - The evil King John
 - The left legs of Richard and John
 - A crown

MODELS FOR FOL: RELATIONS

The objects in the model may be related in various ways. In the figure Richard and John are **brothers**. Formally speaking, **a relation** is just the **set of tuples of objects that are related**.

- A **tuple** is a collection of Objects arranged in a fixed order and is written with angle brackets surrounding the objects.
- **Eg:** The **brotherhood** relation in this model is the set {**<Richard the Lionheart, King John>**,**<King John, Richard the Lionheart>**} The crown is on King John's head, so the “**on head**” relation contains just one tuple, **<the crown, King John>**.
- The relation can be:
 - **binary relation** relating pairs of objects (**Eg: “Brother”**)
 - **unary relation** representing a common object (**Eg:“Person”** representing both **Richard** and **John**)



Eg: Set of blocks {a, b, c, d, e}. The “On” relation includes: On = {<a,b>, <b,c> , <d,e>} <a,b> represents the predicate On(A,B)

MODELS FOR FOL: FUNCTIONS

- Certain kinds of relationships are best considered as functions that relates an object to exactly one object.
- **Eg:-** each person has one left leg, so the model has a **unary “left leg” function** that includes the following mappings (Richard the Lionheart) ----> Richard’s left leg

SYNTAX: ELEMENTS AND SYMBOLS OF

FOL

- The elements for which different symbols are defined are:

- Objects(Constant Symbols):** It refers to an entity that exists in the real world. For example, Ram, John, etc. are referred to as Objects.
- Functions(Function Symbols):** Any function performed by the object/on the object. For example, LeftLeg, writes, eats, Sqrt, etc. are some of the functions.
- Relations(Predicate Symbols):** The relation of an object with the other object defines its relation. For example, brother, mother, king,>, =, ... etc. are some types of relations which exist in the real world.

- Constant Symbols:** These symbols are used to represent the objects. Eg: KingJohn, 2, Koblenz, C,...
 - Function Symbols:** These symbols are used to represent the functions. Eg: Sqrt, LeftLegOf,...
 - Predicate Symbols** are used to represent relations. Eg: brother, mother, king,>,
 - Variable Symbols** x, y, a, b, ...
 - Connectives** $\wedge \vee \neg \Rightarrow \Leftrightarrow$
 - Equality:** ==
 - Quantifiers** $\forall \exists$
-

SYNTAX: ATOMIC AND COMPLEX

SENTENCES

- **Atomic Sentences:** These sentences are formed via predicate symbols may or may not be followed by a list of terms.

Example:

Parents(Ram, Sita) where Ram and Sita are the parents.

Brother(KingJohn, RichardTheLionheart)

Length(LeftLegOf(KingJohn)))

- **Complex Sentences:** These sentences make use of logical connectives to construct more complex sentences from atomic sentences .

Example:

Sibling(KingJohn, Richard) \Rightarrow Sibling(Richard, KingJohn)

\neg Brother (LeftLeg(Richard), John)

King (Richard) \vee King (John)

King (Richard) \Rightarrow King(John)

EXAMPLES: ATOMIC SENTENCES

Brother (KingJohn, RichardTheLionheart)

The diagram shows the atomic sentence *Brother (KingJohn, RichardTheLionheart)*. It is annotated with green curly braces and labels. The first brace groups the word *Brother* as the **predicate**. The second brace groups *KingJohn* and *RichardTheLionheart* as **constant**. The third brace groups the entire expression as an **atomic sentence**. Below each brace, the corresponding label is repeated.

> *(Length(LeftLegOf(Richard)), Length(LeftLegOf(KingJohn)))*

The diagram shows the atomic sentence *> (Length(LeftLegOf(Richard)), Length(LeftLegOf(KingJohn)))*. It is annotated with green curly braces and labels. The first brace groups the character *>* as the **predicate**. The second and third braces group *Length* and *LeftLegOf* as **function**. The fourth and fifth braces group *Richard* and *KingJohn* as **constant**. The sixth brace groups the entire expression as an **atomic sentence**. Below each brace, the corresponding label is repeated.

EXAMPLES: COMPLEX SENTENCES

Sibling(KingJohn, Richard) \Rightarrow *Sibling(Richard, KingJohn)*

The diagram illustrates the structure of atomic and complex sentences. On the left, the atomic sentence *Sibling(KingJohn, Richard)* is shown with its components: *Sibling* is the predicate, and *KingJohn* and *Richard* are terms. These three elements are grouped by a bracket labeled "atomic sentence". This entire atomic sentence is then grouped by a larger bracket labeled "complex sentence". An arrow points to the right, indicating a transformation. On the right, the atomic sentence *Sibling(Richard, KingJohn)* is shown with the same structure: *Sibling* is the predicate, and *Richard* and *KingJohn* are terms. These three elements are grouped by a bracket labeled "atomic sentence". This entire atomic sentence is then grouped by a larger bracket labeled "complex sentence".

SEMANTICS: INTENDED INTERPRETATION

- The semantics must relate sentences to models in order to determine truth. To do this, an interpretation is needed.
 - Interpretation specifying exactly which objects, relations and functions are referred to by the constant, predicate and function symbols. Interpretation specifies referents for ie, it maps constant symbols → objects, predicate symbols → relations, function symbols → functional relations
 - One possible interpretation called as the intended interpretation- is as follows;
 - Richard refers to Richard the Lionheart and John refers to the evil King John.
 - Brother refers to the brotherhood relation, that is the set of tuples of objects given in equation $\{(Richard\ the\ Lionheart, King\ John), (King\ John, Richard\ the\ Lionheart)\}$
 - OnHead refers to the “on head” relation that holds between the crown and King John; Person, King and Crown refer to the set of objects that are persons, kings and crowns.
 - Leftleg refers to the “left leg” function, that is, the mapping given in $\{(Richard\ the\ Lionheart, King\ John), (King\ John, Richard\ the\ Lionheart)\}$
 - There are many other possible interpretations relating these symbols to this particular model.
 - The truth of any sentence is determined by a model and an interpretation for the sentence symbols.
-

SEMANTICS: QUANTIFIERS

- Once we have a logic that allows objects, it is natural to want to express properties of entire collections of objects, instead of enumerating the objects by name. **Quantifiers** is a way for that.
- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
- Quantifiers are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. These are the types of quantifier:
 1. Universal Quantifier, (for all, everyone, everything)
 2. Existential quantifier, (for some, at least one)
 3. Nested Quantifiers

UNIVERSAL QUANTIFIER

- Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing. The Universal quantifier is represented by a symbol \forall , which resembles an inverted A.
- In general, **$\forall x P$ is true** in a given model under a given interpretation **if P is true in all possible extended interpretations.**
- **Universal quantification (\forall) Eg**
 - “**All kings are person**”: $\forall x \text{ King}(x) \Rightarrow \text{Person}(x)$ which means For all x, if x is a king, then x is a person. Here x could be one of the following(5 extended interpretations): Richard, John, Richard’s left leg, John’s left leg, Crown

EXISTENTIAL QUANTIFIER

- Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something. It is denoted by the logical operator \exists , which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.
- In general, $\exists x P$ is true in a given model under a given interpretation **if P is true in at least one extended interpretation that assigns x to a domain element.**
- Existential quantification (\exists) Eg
 - $\exists x \text{ Crown}(x) \wedge \text{OnHead}(x, \text{John})$ which means there is an x such that x is a crown and x is on the John's head. Here , “Crown(crown) \wedge OnHead(crown, John)” is true

NESTED QUANTIFIERS

- It is the nesting of the same type of quantifier. One predicate is nested under the other predicate.
- Nested quantifiers Example
 - $\forall x \forall y [\text{Brother}(x, y) \Rightarrow \text{Sibling}(x, y)]$
 - $\forall x \exists y \text{ Loves}(x, y)$
 - $\exists y \forall x \text{ Loves}(x, y)$
- The **order of quantification is important.**
- $\exists x \forall y$ is not the same as $\forall y \exists x$.

PROPERTIES OF QUANTIFIERS

Quantifier duality: each can be expressed using the other

- $\forall x \text{Likes}(x, \text{IceCream})$ is the same as $\neg \exists x \neg \text{Likes}(x, \text{IceCream})$
- $\exists x \text{Likes}(x, \text{Broccoli})$ is the same as $\neg \forall x \neg \text{Likes}(x, \text{Broccoli})$
- **De Morgan's rules for quantifiers**
 - $\forall x \neg P = \neg \exists x P$
 - $\neg \forall x P = \exists x \neg P$
 - $\forall x P = \neg \exists x \neg P$
 - $\neg \forall x \neg P = \exists x P$

PROPERTIES OF QUANTIFIERS

Quantifiers of same type commute

- $\forall x \forall y$ is the same as $\forall y \forall x$
- $\exists x \exists y$ is the same as $\exists y \exists x$

Quantifiers of different type do NOT commute

- $\exists x \forall y$ is not the same as $\forall y \exists x$

Example

- $\exists x \forall y \text{Loves}(x, y)$: “There is a person who loves everyone in the world”
 - $\forall y \exists x \text{Loves}(x, y)$: “Everyone in the world is loved by at least one person”
-

THINGS TO BE TAKEN CARE

- \rightarrow is the main connective with \forall where as \wedge as the main connective with \exists
- It's a mistake to use \wedge as the main connective with \forall and Using \rightarrow as the main connective with \exists .
- Example
- Correct:
 - $\forall x (\text{StudiesAt}(x, \text{Koblenz})) \rightarrow \text{Smart}(x)$: Everyone who studies at Koblenz is smart”
 - $\exists x (\text{StudiesAt}(x, \text{Landau}) \wedge \text{Smart}(x))$: There is someone who studies at Landau and is smart”
- Wrong:
 - $\forall x (\text{StudiesAt}(x, \text{Koblenz}) \wedge \text{Smart}(x))$: Everyone studies at Koblenz and is smart,”i.e., Everyone studies at Koblenz and everyone is smart”
 - $\exists x (\text{StudiesAt}(x; \text{Landau})) \rightarrow \text{Smart}(x)$: There is someone who, if he/she studies at Landau, is smart’This is true if there is anyone not studying at Landau

EQUALITY

- **term1 = term2** is true under a given interpretation if and only if **term1 and term2 refer to the same object .**
- Can be used to **state facts** about a given function

E.g., Father(John) = Henry

- Can be **used with negation to insist that two terms are not the same object**

USING FOL:

- **Example 1:** Lipton is a tea.
 - **Solution:** Here, the object is Lipton. It will be represented as **Tea(Lipton)**. There is no requirement of quantifiers because the quantity is not specified in the given predicate
 - **Example 2:** Every man is mortal.
 - **Solution:** Here, the quantifier is the universal identifier, and the object is man. Let x be the man. Thus, it will be represented as **$\forall x: \text{man}(x) \rightarrow \text{mortal}(x)$** .
 - **Example 3:** All girls are beautiful.
 - **Solution:** Here, we are talking about all girls. It means universal quantifier will be used. The object is girls. Let, y be the girls. Therefore, it will be represented as **$\forall y: \text{girls}(y) \rightarrow \text{beautiful}(y)$** .
 - **Example 4:** All that glitters is not gold.
 - **Solution:** Here, we will represent gold as x. Therefore, it will be represented as **$\forall x: \text{glitters}(x) \rightarrow \neg \text{gold}(x)$** .
 - **Example 5:** Some boys are obedient.
 - **Solution:** Here, boys are objects. The quantifier used will be existential quantifier. Let x be the boys. Thus, it will be represented as **$\exists x: \text{boys}(x) \rightarrow \text{obedient}(x)$** .
 - **Example 6:** Some cows are black and some cows are white.
 - **Solution:** Let, x be the cows. Therefore, it will be represented as: **$\exists x: \text{cows}(x) \rightarrow \text{black}(x) \wedge \text{white}(x)$** .
-

USING FOL:

1. **All birds fly:** $\forall x \text{ bird}(x) \rightarrow \text{fly}(x)$.

In this question the predicate is "fly(bird)." And since there are all birds who fly so it will be represented as follows

2. **Every man respects his parent.** $\forall x \text{ man}(x) \rightarrow \text{respects}(x, \text{parent})$.

In this question, the predicate is "respect(x, y)," where x=man, and y= parent. Since there is every man so will use \forall , and it will be represented as follows

3. **Some boys play cricket.** $\exists x \text{ boys}(x) \rightarrow \text{play}(x, \text{cricket})$.

In this question, the predicate is "play(x, y)," where x= boys, and y= game. Since there are some boys so we will use \exists , and it will be represented as:

4. **Not all students like both Mathematics and Science** $\neg \forall (x) [\text{student}(x) \rightarrow \text{like}(x, \text{Mathematics}) \wedge \text{like}(x, \text{Science})]$.

In this question, the predicate is "like(x, y)," where x= student, and y= subject. Since there are not all students, so we will use \forall with negation, so following representation for this:

5. **Only one student failed in Mathematics.** $\exists(x) [\text{student}(x) \rightarrow \text{failed}(x, \text{Mathematics}) \wedge \forall (y) [\neg(x==y) \wedge \text{student}(y) \rightarrow \neg\text{failed}(x, \text{Mathematics})]]$.

In this question, the predicate is "failed(x, y)," where x= student, and y= subject. Since there is only one student who failed in Mathematics, so we will use following representation for this.

USING FOL: KINSHIP DOMAIN

- One's mom is one's female parent: $\forall x, y (\text{Mother}(x, y) \Leftrightarrow (\text{Female}(x) \wedge \text{Parent}(x, y)))$
- One's husband is one's male spouse: $\forall w, h \text{ Husband}(h, w) \Leftrightarrow \text{Male}(h) \wedge \text{Spouse}(h, w)$
- Brothers are siblings: $\forall x, y (\text{Brother}(x, y)) \Leftrightarrow \text{Sibling}(x, y)$
- Parent and child are inverse relations: $\forall p, c \text{ Parent}(p, c) \Leftrightarrow \text{Child}(c, p)$
- A Grandparent is a Parent of one's Parent: $\forall g, c \text{ Grandparent}(g, c) \Leftrightarrow \exists p \text{ Parent}(g, p) \wedge \text{Parent}(p, c)$
- A first cousin is a child of a parent's sibling: $\forall x, y (\text{FirstCousin}(x, y) \Leftrightarrow \exists p, ps (\text{Parent}(p, x) \wedge \text{Sibling}(ps, p) \wedge \text{Parent}(ps, y)))$

Practice:

- Male and female are disjoint categories
 - A sibling is another child of one's parents
-

SOLUTION

- Male and female are disjoint categories:
 - $\forall x \text{ Male}(x) \Leftrightarrow \neg \text{Female}(x)$
 - A sibling is another child of one's parent:
 - $\forall x, y \text{ Sibling}(x, y) \Leftrightarrow x \neq y \wedge \exists p \text{ Parent}(p, x) \wedge \text{Parent}(p, y)$
 - Definition of (full) sibling in terms of Parent
 - $\forall x, y \text{ Sibling}(x, y) \Leftrightarrow (\neg(x = y) \wedge \exists m, f (\neg(m = f) \wedge \text{Parent}(m, x) \wedge \text{Parent}(f, x) \wedge \text{Parent}(m, y) \wedge \text{Parent}(f, y)))$
-

FREE AND BOUND VARIABLES

- The quantifiers interact with variables which appear in a suitable way. There are two types of variables in First-order logic which are given below:
- **Free Variable:** A variable is said to be a free variable in a formula if it occurs outside the scope of the quantifier.

Example: $\forall x \exists(y)[P(x, y, z)]$, where z is a free variable.

- **Bound Variable:** A variable is said to be a bound variable in a formula if it occurs within the scope of the quantifier.

Example: $\forall x [A(x) B(y)]$, here x and y are the bound variables.



INFERENCE IN FOL

INFERENCE IN FOL

- Inference in First-Order Logic is used to deduce new facts or sentences from existing sentences.
- Two ideas:
 - convert the KB to propositional logic and use propositional inference
 - a shortcut that manipulates on first-order sentences directly

FOL INFERENCE RULES FOR QUANTIFIER:

As propositional logic we also have inference rules in first-order logic, so following are some basic inference rules in FOL:

- 1. Universal Generalization**
 - 2. Universal Instantiation**
 - 3. Existential Instantiation**
 - 4. Existential introduction**
-

1. UNIVERSAL GENERALIZATION:

- Universal generalization is a valid inference rule which states that if premise $P(c)$ is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as $\forall x P(x)$.

- It can be represented as: $P(c)$

$\forall x P(x)$

- This rule can be used if we want to show that every element has a similar property.
- In this rule, x must not appear as a free variable.
- Example:** Let's represent, $P(c)$: "A byte contains 8 bits", so for $\forall x P(x)$ "All bytes contain 8 bits.", it will also be true.

2. UNIVERSAL INSTANTIATION (UI):

- In this, we can infer any sentence by substituting a ground term (a term without variables) for the variables. In short, when we create the FOPL of the given statement, we can easily infer any other statement using that FOPL . It can be represented as:

$\forall x P(x)$

$P(c)$

- **Notation:** Let, $\text{SUBST}(\theta, \alpha) = \forall v \alpha / \text{Subst}(\{v/g\}, \alpha)$
- means replace All occurrences of “v” with “g” in expression “ α ” be the result θ , where v is the variable and g is the ground term.
- **For example:** Every man is mortal. It is represented as $\forall x: \text{man}(x) \rightarrow \text{mortal}(x)$. In UI, we can infer different sentences as: $\text{man(John)} \rightarrow \text{mortal(John)}$ also $\text{man(Aakash)} \rightarrow \text{mortal(Aakash)}$, etc.
- **Universal instantiation** is also called as **universal elimination** or **UI** is a valid inference rule. It can be applied multiple times to add new sentences. The new KB is logically equivalent to the previous KB. As per UI, **we can infer any sentence obtained by substituting a ground term for the variable**. The UI rule state that we can infer any sentence $P(c)$ by substituting a ground term c (a constant within domain x) from $\forall x P(x)$ for any object in the universe of discourse.

EXAMPLES

- **Example:1.**
 - IF "Every person like ice-cream" $\Rightarrow \forall x P(x)$ so we can infer that "John likes ice-cream" $\Rightarrow P(c)$
 - **Example: 2.**
 - Let's take a famous example,
 - "All kings who are greedy are Evil." So let our knowledge base contains this detail as in the form of FOL:
 - **$\forall x \text{ king}(x) \wedge \text{greedy}(x) \rightarrow \text{Evil}(x)$,**
 - So from this information, we can infer any of the following statements using Universal Instantiation:
 - **$\text{King}(\text{John}) \wedge \text{Greedy}(\text{John}) \rightarrow \text{Evil}(\text{John})$,**
 - **$\text{King}(\text{Richard}) \wedge \text{Greedy}(\text{Richard}) \rightarrow \text{Evil}(\text{Richard})$,**
 - **$\text{King}(\text{Father}(\text{John})) \wedge \text{Greedy}(\text{Father}(\text{John})) \rightarrow \text{Evil}(\text{Father}(\text{John}))$,**
-

3. EXISTENTIAL INSTANCEATION(EI):

- In EI, the **variable is substituted** by a **single new constant symbol**:

$\exists x P(x)$

$P(c)$

- This rule states that one can infer $P(c)$ from the formula given in the form of $\exists x P(x)$ for a new constant symbol c .
 - **Notation:** Let, the variable be v which is replaced by a constant symbol k for any sentence α . $\exists v \alpha / \text{Subst}(\{v/k\}, \alpha)$
 - The value of k is unique as it does not appear for any other sentence in the knowledge base. Such type of constant symbols are known as **Skolem constant**. As a result, EI is a special case of **Skolemization process**.
 - **Note:** **UI can be applied several times to produce many sentences**, whereas **EI can be applied once, and then the existentially quantified sentences can be discarded.**
 - **For example:** $\exists x: \text{steal}(x, \text{Money})$. We can infer from this: **steal(Thief, Money)**
 - Existential instantiation is also called as **Existential Elimination**, which is a valid inference rule in first-order logic.
-

EXAMPLE

- From the given sentence: $\exists x \text{ Crown}(x) \wedge \text{OnHead}(x, \text{John})$,
 - So we can infer: $\text{Crown}(K) \wedge \text{OnHead}(K, \text{John})$, as long as K does not appear in the knowledge base.
 - The above used K is a constant symbol, which is called **Skolem constant**.
 - The Existential instantiation is a special case of **Skolemization process**.
-

4. EXISTENTIAL INTRODUCTION

- An existential introduction is also known as an existential generalization, which is a valid inference rule in first-order logic. This rule states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.
- It can be represented as:

P(c)

$\exists x P(x)$

- **Example:** Let's say that,
"Priyanka got good marks in English."
"Therefore, someone got good marks in English."

EXAMPLE

- For example, the following argument can be proven correct using the Universal Instantiation: "**No humans can fly. John Doe is human. Therefore John Doe can not fly.**"

First let us express this using the following notation:

$F(x)$ means "**x can fly.**"

$H(x)$ means "**x is human.**"

d is a symbol representing **John Doe.**

Then the argument is: $\forall x [H(x) \rightarrow \neg F(x)] \wedge H(d) \rightarrow \neg F(d)$

1. $\forall x [H(x) \rightarrow \neg F(x)]$	Hypothesis
2. $H(d)$	Hypothesis
3. $H(d) \rightarrow \neg F(d)$	Universal instantiation on 1.
4. $\neg F(d)$	Modus ponens on 2 and 3.

REDUCTION TO PROPOSITIONAL

• INFERENCE

- Suppose the KB contains just the following:

$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$

King(John)

Greedy(John)

Brother(Richard, John)

- Instantiating the universal sentence in all possible ways, we have:

King(John) \wedge Greedy(John) \Rightarrow Evil(John)

King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)

King(John)

Greedy(John)

Brother(Richard, John)

- Discarding quantifiers and applying inference rules to make KB propositional is known as **propositionalization**.**
 - Proposition symbols are : King(John), Greedy(John), Evil(John), King(Richard), etc.
 - What conclusion can you get?
-

PROBLEMS WITH

PROPOSITIONALIZATION

Propositionalization seems to generate lots of irrelevant sentences

- E.g., from:

$$\exists x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$$

King(John)

$\forall y \text{ Greedy}(y)$

Brother(Richard, John)

- It seems obvious that Evil(John) is true, but propositionalization produces lots of facts such as Greedy(Richard) that are irrelevant
 - When the KB includes a function symbol, the set of possible ground term substitutions is infinite like Father(Father(...(John)...))
 - Theorem:
 - If a sentence is entailed by the original, first-order KB, then there is a proof involving just a finite subset of the propositionalized KB
 - First instantiation with constant symbols
 - Then terms with depth 1 (Father(John)) Then terms with depth 2 ... Until the sentence is entailed
-

PROBLEMS WITH PROPOSITIONALIZATION

- Every FOL KB can be propositionalized so as to preserve entailment
 - (A ground sentence is entailed by new KB iff entailed by original KB)
 - Idea: propositionalize KB and query, apply resolution, return result
 - Here the issue is **set of possible ground-term substitution is infinite**. This causes halting problem in Turing Machine
 - *Entailment of FOL is semidecidable-i.e Algorithm says yes to every entailed sentences, but no algorithm exists that says no to every non-entailed sentence*
 - **Hence propositionalization is inefficient.**
-

UNIFICATION AND LIFTING



FIRST ORDER INFERENCE RULE:

GENERALIZED MODUS PONEN

- For the inference process in FOL, **we have a single inference rule** which is called **Generalized Modus Ponens**. Generalized Modus Ponens can be summarized as, " P implies Q and P is asserted to be true, therefore Q must be True."
- According to Modus Ponens, for atomic sentences p_i , p'_i , q . Where there is a substitution θ such that $\text{SUBST}(\theta, p'_i) = \text{SUBST}(\theta, p_i)$, it can be represented as:

$$\frac{p'_1, p'_2, \dots, p'_n, (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

- Here the **conclusion** is the result of **applying substitution to consequent q**.
- GMP is **lifted version of Modus ponens**-it raises Modus ponens from **propositional logic to FOL**.

GENERALIZED MODUS PONEN EXAMPLE

- If there is some substitution θ that makes the premise of the implication identical to sentences already in the KB, then we assert the conclusion of the implication, after applying θ . Given query **Evil(x)**?

$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$

King(John)

Greedy(John) What's θ here? $\theta=(x/\text{John})$

- Instead of **Greedy(John)**, we know every one is greedy

$\forall x \text{ King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)$

King(John)

∀y Greedy(y) What's θ here? $\theta=(x/\text{John}, y/\text{john})$

- **Example:** We will use this rule for **Kings are evil**, so we will find some x such that x is king, and x is greedy so we can infer that **x is evil**.

- Here let say, p1' is **king(John)** p1 is **king(x)**

- p2' is **Greedy(y)** p2 is **Greedy(x)**

- θ is $\{x/\text{John}, y/\text{John}\}$ q is **evil(x)**

- **SUBST(θ ,q)** is **Evil(John)**.
-

UNIFICATION AND LIFTING

- Unification is a process of making **two different logical atomic expressions identical by finding a substitution**. Unification depends on the substitution process and is the **key component of First-order inference algorithms**.
- Unification is the process used by the lifted inference rules to find substituents that could give identical but different logical expressions. It means the meaning of the sentence should not be changed, but it should be expressed in multiple ways. The **UNIFY** algorithm in unification takes two sentences as input and then returns a unifier if one exists:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q)$$

- Unification is a lifted version of Modus Ponen as it uplifts the Modus Ponens from ground propositions to FOPL. It takes two literals as input and makes them identical using substitution.
 - **Unification** is needed for **resolution** to work.
-

UNIFICATION

- **Unifier:** a substitution that makes two clauses resolvable
- E.g. Let's say there are two different expressions, $P(x, y)$, and $P(a, f(z))$.
- In this example, we need **to make both above statements identical to each other**. For this, we will **perform the substitution**.

$P(x, y) \dots\dots\dots$ (i)

$P(a, f(z)) \dots\dots\dots$ (ii)

- Substitute x with a , and y with $f(z)$ in the first expression, and it will be represented as a/x or x/a and $f(z)/y$ or $y/f(z)$. There are different interpretation for this notation.

In Russel an Norwig the representation is $x/a, y/f(z)$

- With both the substitutions, the first expression will be identical to the second expression and the substitution set will be: $[x/a, y/f(z)]$
-

UNIFICATION EXAMPLE

- Given KB:

- **Knows(John, Jane)**
- **Knows(y, Bill)**
- **Knows(y, Mother(y))**
- **Knows(x, Elizabeth)**

p	q	θ
UNIFY(Knows(John, x),	Knows(John, Jane))	{x/Jane}
UNIFY(Knows(John, x)	Knows(y, Bill))	{x/Bill, y/John}
UNIFY(Knows(John, x)	Knows(y, Mother(y)))	{y/John, x/Mother(John)}
UNIFY(Knows(John, x)	Knows(x, Elizabeth))	fail

- a query **Knows(John, x)** means **Every one knows John** The question is- **Whom does John know?** Ie like **Does John know Jane? Does John know Bill?**
 - Answer to this query can be found by finding all sentences in KB that unifies **Knows(John,x)**. The UNIFY algorithm will search all the related sentences in the knowledge base, which could unify with **Knows(John,x)**.
 - UNIFY (Knows(John, x), Knows(John, Jane)) $\equiv\{x/Jane\}$
 - UNIFY (Knows{John,x}, Knows{y, Bill}) $\equiv\{x/Bill, y/John\}$
 - UNIFY (Knows{John, x}, Knows{y, Mother(y)}) $\equiv\{ y/John, x/Mother(John)\}$
-

UNIFICATION EXAMPLE

- UNIFY (Knows{John,x}, Knows{x, Elizabeth}) \equiv fails. The last one failed because we have used the **same variable for two persons at the same time** ie x cannot take both values John and Elizabeth.
 - It can be avoided by standardizing apart ie renaming
 - UNIFY (Knows{John,x}, Knows{x, Elizabeth}) \equiv UNIFY (Knows{John,x}, Knows{x17, Elizabeth}) \equiv { x/Elizabeth, x17/ John }
 - Consider the following two unifications
 - UNIFY (Knows (John, x), Knows (y, z)) = { y/John, x/ z }
 - UNIFY (Knows (John, x), Knows (y, z)) = { y/John, x/John , z/John }
 - We say the first unifier is more general than the second . **It places fewer restrictions on the values of variables.** For every unifiable pairs of expressions, there is a single **most generalized unifier (MGU)** – E.g., the former unifier, { y/John, x/ z }, shown above
-

UNIFICATION RULES

- Following are some basic conditions for unification:
 1. Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
 2. Number of Arguments in both expressions must be identical.
 3. Unification will fail if there are two similar variables present in the same expression.
 - Rules for substitutions:
 1. Can replace a variable by a constant.
 2. Can replace a variable by a variable.
 3. Can replace a variable by a function expression, as long as the function expression does not contain the variable.
-

MOST GENERAL UNIFIER

- In cases **where there is more than one substitution choose the one that makes the least commitment (most general) about the bindings. The substitution variables are called Most General Unifier or MGU.**

= {y / John, x / z} not {y / John, x / John, z / John} not {y / John, x / z, z / Freda} ...

- For each pair of atomic sentences, give the most general unifier if it exists.

1. Find the MGU of $\{p(f(a), g(Y))\}$ and $p(X, X)\}$

- Sol: $S_0 \Rightarrow \{\}$ Here, $\Psi_1 = p(f(a), g(Y))$, and $\Psi_2 = p(X, X)$

SUBST $\theta = \{X/ f(a)\}$

$S1 \Rightarrow \Psi_1 = p(f(a), g(Y))$, and $\Psi_2 = p(f(a), f(a))$

SUBST $\theta = \{g(y)/f(a)\}$, Unification failed.

- Unification is not possible for these expressions.

MGU EXAMPLES

2. Find the MGU of $\{p(b, X, f(g(Z)))\}$ and $\{p(Z, f(Y), f(Y))\}$

- Here, $\Psi_1 = p(b, X, f(g(Z)))$, and $\Psi_2 = p(Z, f(Y), f(Y))$
 $S_0 \Rightarrow \{ p(b, X, f(g(Z))); p(Z, f(Y), f(Y)) \}$
SUBST $\theta = \{Z/b\}$
- $S_1 \Rightarrow \{ p(b, X, f(g(b))); p(b, f(Y), f(Y)) \}$
SUBST $\theta = \{X/f(Y)\}$
- $S_2 \Rightarrow \{ p(b, f(Y), f(g(b))); p(b, f(Y), f(Y)) \}$
SUBST $\theta = \{Y/g(b)\}$
- $S_2 \Rightarrow \{ p(b, f(g(b)), f(g(b)); p(b, f(g(b)), f(g(b))) \}$ **Unified Successfully.**
And Unifier = { Z/b , $X/f(Y)$, $Y/g(b)$ }.

3. UNIFY(knows(Richard, x) , $\text{knows(Richard, John)}$)

- Here, $\Psi_1 = \text{knows(Richard, x)}$, and $\Psi_2 = \text{knows(Richard, John)}$
 $S_0 \Rightarrow \{ \text{knows(Richard, x)}; \text{knows(Richard, John)} \}$
SUBST $\theta = \{x/John\}$
 $S_1 \Rightarrow \{ \text{knows(Richard, John)}; \text{knows(Richard, John)} \}$, **Successfully Unified.**
Unifier: { $x/John$ }.
-

MGU EXAMPLES

4. Find the MGU of { $p(X, X)$, and $p(Z, f(Z))$ }

- Here, $\Psi_1 = \{p(X, X)$, and $\Psi_2 = p(Z, f(Z))\}$
 $S_0 \Rightarrow \{p(X, X), p(Z, f(Z))\}$
SUBST $\theta = \{X/Z\}$
 $S_1 \Rightarrow \{p(Z, Z), p(Z, f(Z))\}$
SUBST $\theta = \{f(Z) / Z\}$, Unification Failed.
- Hence, unification is not possible for these expressions.

5. Find the MGU of UNIFY(prime(11), prime(y))

- Here, $\Psi_1 = \{\text{prime}(11)$, and $\Psi_2 = \text{prime}(y)\}$
 $S_0 \Rightarrow \{\text{prime}(11), \text{prime}(y)\}$
SUBST $\theta = \{y/11\}$
- $S_1 \Rightarrow \{\text{prime}(11), \text{prime}(11)\}$, Successfully unified.
Unifier: $\{y/11\}$.

MGU EXAMPLES

6. Find the MGU of $\{Q(a, g(x, a), f(y)), Q(a, g(f(b), a), x)\}$

- Here, $\Psi_1 = Q(a, g(x, a), f(y))$, and $\Psi_2 = Q(a, g(f(b), a), x)$

$S_0 \Rightarrow \{Q(a, g(x, a), f(y)); Q(a, g(f(b), a), x)\}$

$SUBST\ \theta = \{x/f(b)\}$

$S_1 \Rightarrow \{Q(a, g(f(b), a), f(y)); Q(a, g(f(b), a), f(b))\}$

- $SUBST\ \theta = \{y/b\}$

$S_1 \Rightarrow \{Q(a, g(f(b), a), f(b)); Q(a, g(f(b), a), f(b))\}$, Successfully Unified.

- Unifier: $[a/a, x/f(b), y/b]$.

- CHALLENGES

1. $P(A, B, B), P(x, y, z)$

2. $Q(y, G(A, B)), Q(G(x, x), y)$

3. $Older(Father(y), y), Older(Father(x), John)$

4. $Knows(Father(y), y), Knows(x, x)$

RESOLUTION IN FOL

- Resolution means proving a conclusion S from given premises expressed in FOL.
- Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. Unification is a key concept in proofs by resolutions.

RESOLUTION STEPS

1. Conversion of facts into first-order logic.
2. Convert all sentences to CNF(clausal form)
3. Negate conclusion S & convert result to CNF
4. Add negated conclusion S to the premise clauses
5. Draw resolution graph (unification), to better understand all the above steps, we will take an example in which we will apply resolution.Repeat until contradiction or no progress is made:
 - a. Select 2 clauses (call them parent clauses)
 - b. Resolve them together, performing all required unifications
 - c. If resolvent is the empty clause, a contradiction has been found (i.e., S follows from the premises)
 - d. If not, add resolvent to the premises

The Resolution Rule

Resolution relies on the following rule:

$$\frac{\neg\alpha \Rightarrow \beta, \beta \Rightarrow \gamma}{\neg\alpha \Rightarrow \gamma} \quad \text{Resolution rule}$$

equivalently,

$$\frac{\alpha \vee \beta, \neg\beta \vee \gamma}{\alpha \vee \gamma} \quad \text{Resolution rule}$$

Applying the resolution rule:

1. Find two sentences that contain the same literal, once in its positive form & once in its negative form:

CNF sentences → $\boxed{\text{summer} \vee \text{winter}, \neg\text{winter} \vee \text{cold}}$

2. Use the resolution rule to eliminate the literal from both sentences

→ $\boxed{\text{summer} \vee \text{cold}}$

RESOLUTION INFERENCE RULE IN FOL

- The resolution inference rule: The resolution rule for first-order logic is simply a lifted version of the propositional rule. **Resolution can resolve two clauses if they contain complementary literals**, which are assumed to be standardized apart so that they share no variables.
- Where l_i and m_j are complementary literals.

$$\frac{l_1 \vee \dots \vee l_k \quad m_1 \vee \dots \vee m_n}{\text{SUBST}(\emptyset, l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n)}$$

- This rule is also called the **binary resolution rule** because it only resolves exactly two literals.
-

Resolution: brief summary

Full first-order version:

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \quad m_1 \vee \cdots \vee m_n}{(\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)\theta}$$

where $\text{UNIFY}(\ell_i, \neg m_j) = \theta$.

For example,

$$\frac{\begin{array}{l} \neg \text{Rich}(x) \vee \text{Unhappy}(x) \\ \text{Rich}(\text{Ken}) \end{array}}{\text{Unhappy}(\text{Ken})}$$

with $\theta = \{x/\text{Ken}\}$

Apply resolution steps to $CNF(KB \wedge \neg \alpha)$; complete for FOL

EXAMPLE FOR RESOLUTION INFERENCE RULE

- Example: We can resolve two clauses which are given below:
- **[Animal (g(x) V Loves (f(x), x)] and [¬ Loves(a, b) V ¬ Kills(a, b)]**
- Where two complimentary literals are: **Loves (f(x), x)** and **¬ Loves (a, b)**
- These literals can be unified with unifier $\theta = [a/f(x), \text{ and } b/x]$, and it will generate a resolvent clause:
- **[Animal (g(x) V ¬ Kills(f(x), x)].**

EXAMPLE 1:

1. If something is intelligent, it has common sense
 2. Deep Blue does not have common sense
- Prove that Deep Blue is not intelligent

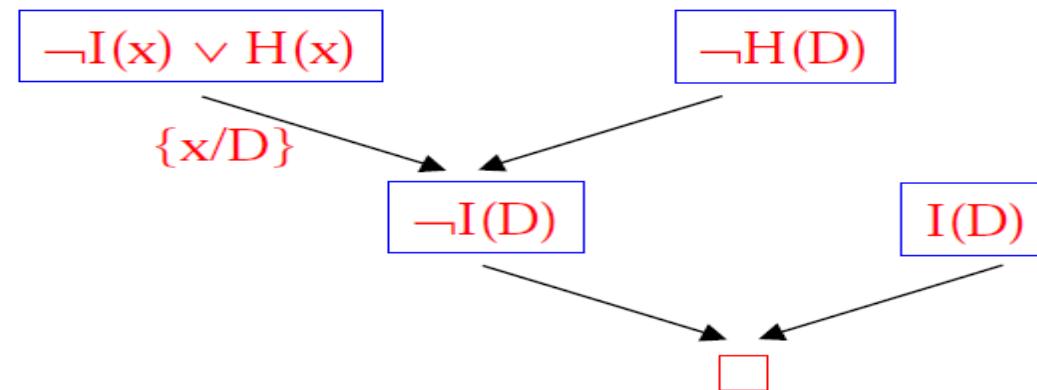
1. $\forall x.I(x) \Rightarrow H(x)$
2. $\neg H(D)$
Conclusion: $\neg I(D)$
Denial: C3: $I(D)$



C1: $\neg I(x) \vee H(x)$
C2: $\neg H(D)$

CNF

A resolution proof of $\neg I(D)$:



Proof also written as:

C4: $\neg I(D)$

C5: \square

2nd literal,
1st clause
r[C1b, C2]
r[C3, C4]