



Dr. Vishwanath Karad

**MIT WORLD PEACE  
UNIVERSITY** | PUNE

TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

## AI UNIT - I

## UNIT-I

# Introduction to Artificial Intelligence and Search Strategies

- Part-I
- History and Introduction to AI,
- Intelligent Agent,
- Types of agents,
- Environment and types,
- Typical AI problems

# Artificial Intelligence

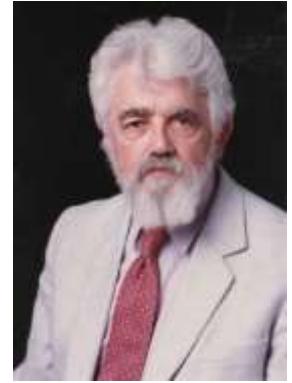
What is AI ?

Artificial Intelligence is concerned with the design of intelligence in an artificial device.

The term was coined by McCarthy in 1956.

There are two ideas in the definition.

1. Intelligence
2. Artificial device



**John McCarthy**

# Definition of AI

“The exciting new effort to make computers think ... machine with minds, ... ” (**Haugeland, 1985**)

“Activities that we associated with human thinking, activities such as decision-making, problem solving, learning ... ” (**Bellman, 1978**)

“The art of creating machines that perform functions that require intelligence when performed by people” (**Kurzweil, 1990**)

**“The study of how to make computers do things at which, at the moment, people are better”**  
(**Rich and Knight, 1991**)

“The study of mental faculties through the use of computational models” (**Charniak and McDermott, 1985**)

“ The study of the computations that make it possible to perceive, reason, and act” (**Winston, 1992**)

“A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes” (**Schalkoff, 1990**)

“The branch of computer science that is concerned with the automation of intelligent behavior” (**Luger and Stubblefield, 1993**)

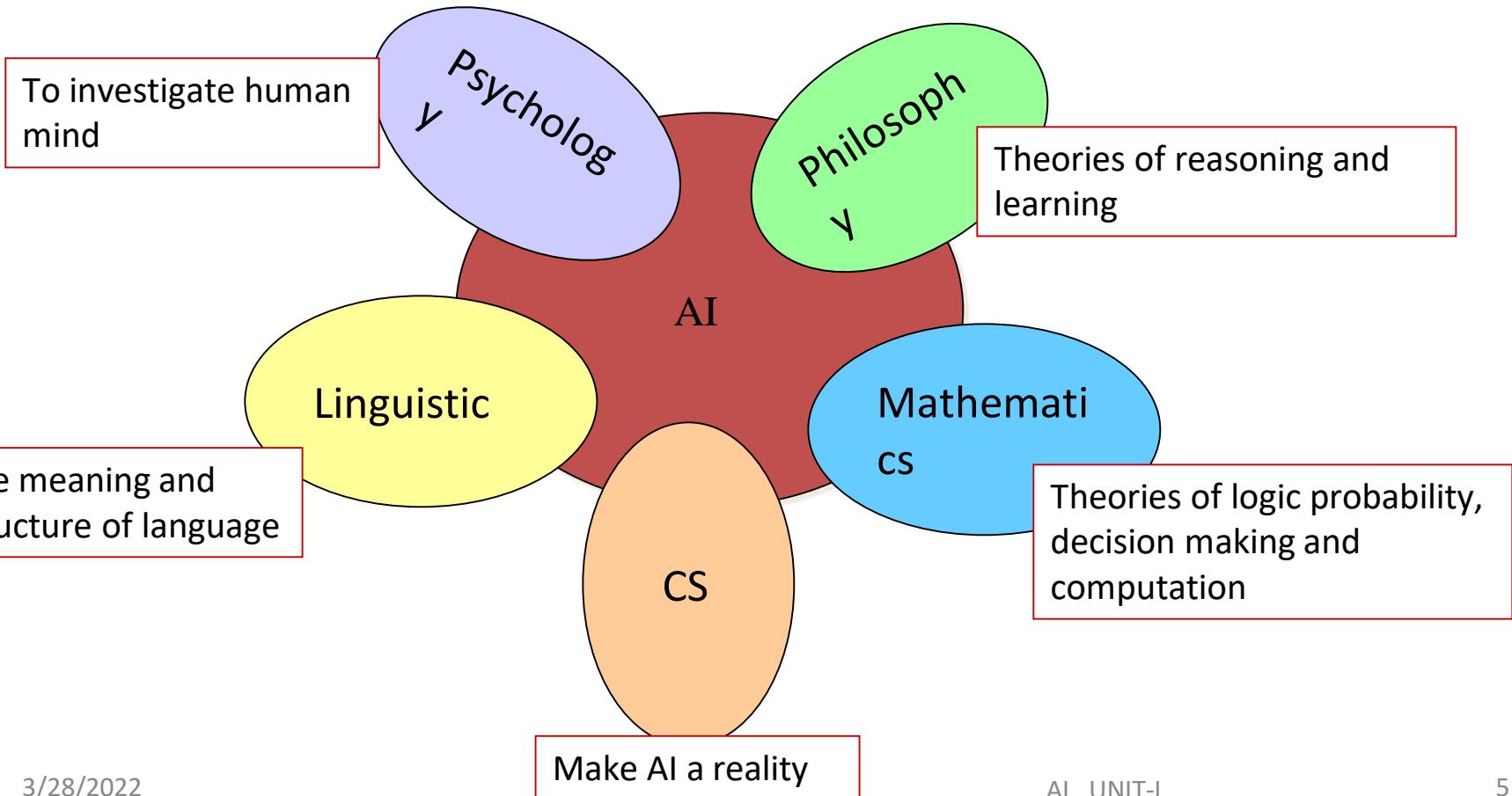
In conclusion, they falls into four categories: Systems that think like human, act like human, think rationally, or act rationally.



What is your definition of AI?

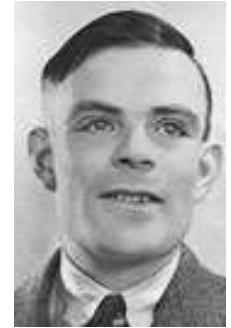
# AI Foundations?

AI **inherited** many ideas, viewpoints and techniques from other **disciplines**.

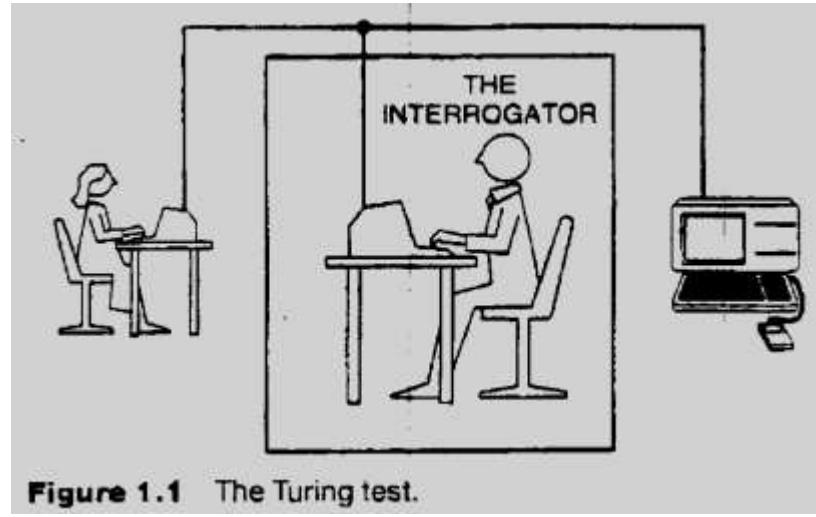


# The Turing Test

([Can Machine think? A. M. Turing, 1950](#))



- Requires:
  - Natural language Processing
  - Knowledge representation
  - Automated reasoning
  - Machine learning
  - (vision, robotics) for full test



**Figure 1.1** The Turing test.

# Acting/Thinking Humanly/Rationally

- Turing test (1950)
- Requires:
  - Natural language
  - Knowledge representation
  - automated reasoning
  - machine learning
  - (vision, robotics.) for full test
- Methods for Thinking Humanly:
  - Introspection, the general problem solver
  - Cognitive sciences

Thinking rationally:

Logic Problems: how to represent and reason in a domain

Acting rationally:

Agents: Perceive and act

# History of AI

- McCulloch and Pitts (1943)
  - Neural networks that learn
- Minsky and Edmonds (1951)
  - Built a neural net computer
- Dartmouth conference (1956):
  - McCarthy, Minsky, Newell, Simon met,
  - Logic theorist (LT)- Of Newell and Simon proves a theorem in Principia Mathematica-Russel.
  - The name “Artificial Intelligence” was coined.
- 1952-1969
  - GPS- Newell and Simon
  - Geometry theorem prover - Gelernter (1959)
  - Samuel Checkers that learns (1952)
  - McCarthy - Lisp (1958), Advice Taker, Robinson’s resolution

# History.... continued

## 1969-1979 Knowledge-based systems

- Expert systems:
  - **Dendral:** Inferring molecular structures
  - **Mycin:** diagnosing blood infections
  - **Prospector:** recommending exploratory drilling.

## • 1980-1988: AI becomes an industry

- R1: McDermott, 1982, order configurations of computer systems
- 1981: Fifth generation

## • 1986-present: return to neural networks

## • Recent event:

- AI becomes a science: planning, belief network

## State of the art

- **Deep Blue** defeated the world chess champion Garry Kasparov in 1997
- During the 1991 Gulf War, US forces deployed an AI logistics planning and scheduling program that involved up to 50,000 vehicles, cargo, and people
- NASA's on-board autonomous planning program controlled the scheduling of operations for a spacecraft
- **Proverb** solves crossword puzzles better than most humans

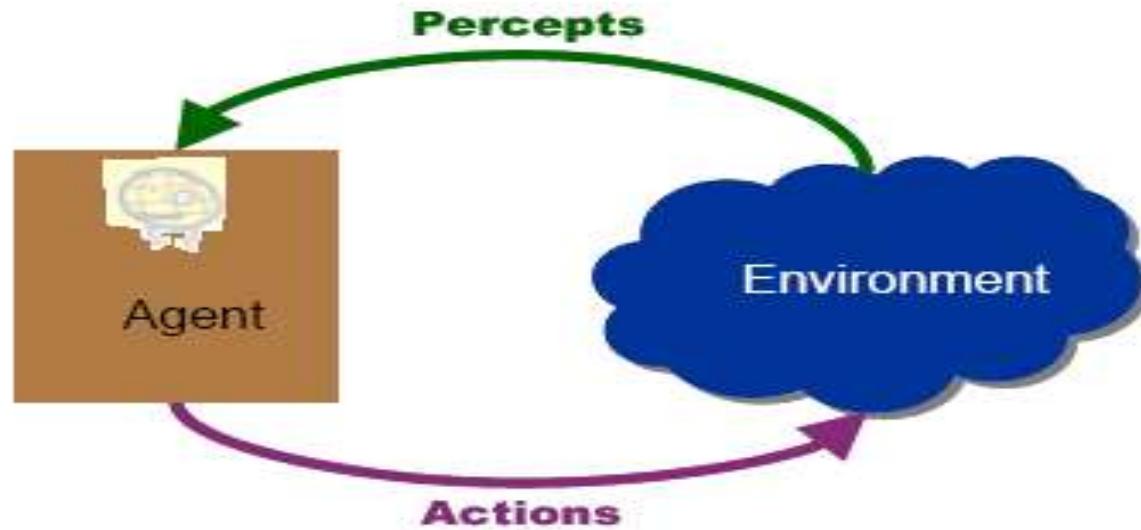
# Summary

- Definition of AI
- Turing Test
- Foundations of AI
- History

# Intelligent Agents

- Agents and environments
- Rationality
- PEAS (Performance measure, Environment, Actuators, Sensors)
- Environment types
- Agent types Or The Structure of Agents

# Agents



An **agent** is any thing that can be viewed as **perceiving** its **environment** through **sensors** and **acting** upon that environment through **actuators/ effectors**

Ex: Human Being , Calculator etc

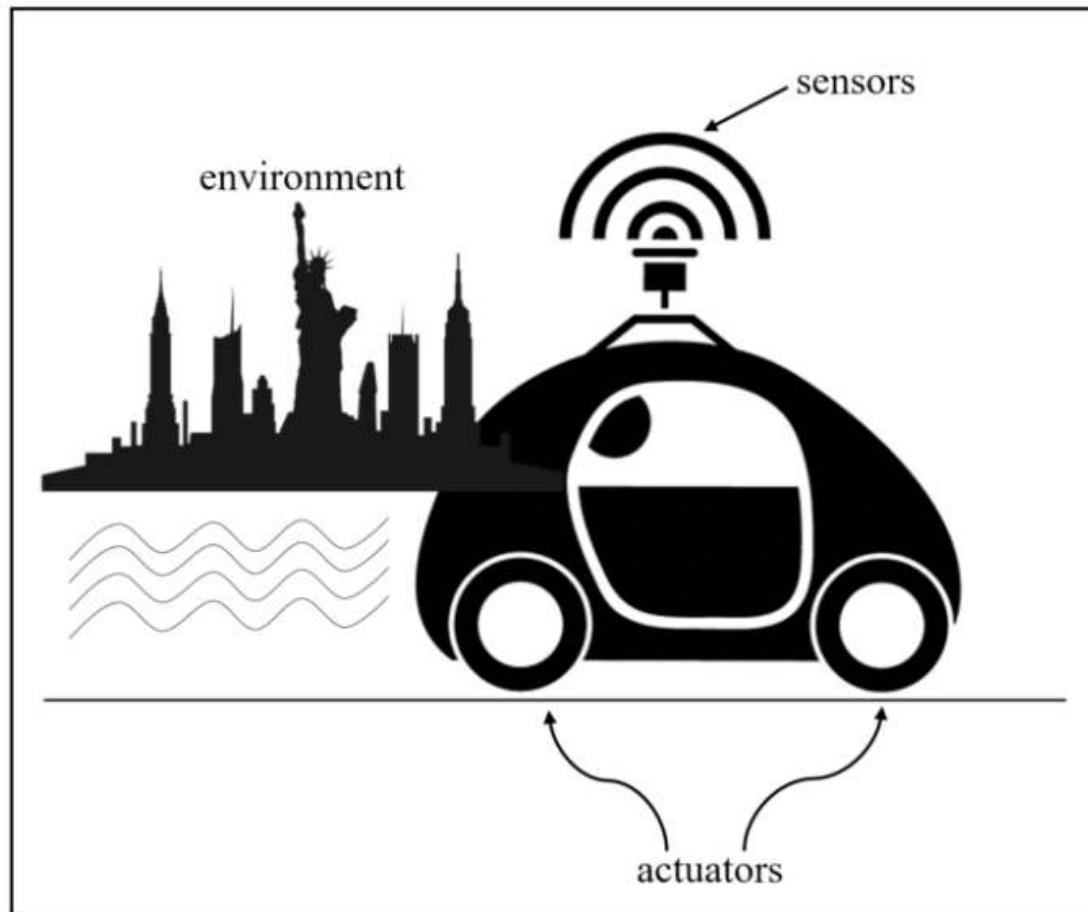
**Agent has goal** –the objective which agent has to satisfy

Actions can potentially change the environment

Agent perceive current percept or sequence of perceptions

# Agent-Driverless Taxi

## Sensors, Actuators, Environment



# Agents

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators/effectors

**1. Human agent:** eyes, ears, and other organs used as **sensors**;

- hands, legs, mouth, and other body parts used as **actuators/ Effector**

**2. Robotic agent:**

- **Sensors**:- cameras (picture Analysis) and infrared range finders for sensors, Solar Sensor.
- **Actuators**- various motors, speakers, Wheels

**3. Software Agent(Soft Bot)**

- Functions as sensors
- Functions as actuators
- **Ex. Askjeeves.com, google.com**

**4. Expert System**

Ex- Cardiologist

# What is an Intelligent Agent

- **Fundamental Facilities of Intelligent Agent**
  - Acting
  - Sensing
  - Understanding, reasoning, learning
- In order to act one must sense , Blind actions is not characteristics of Intelligence.
- **Goal:** Design **rational** agents that do a “good job” of acting in their environments
- **success determined based on some objective performance measure**

# What is an Intelligent Agent

- Rational Agents
  - An agent should strive to "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.
    - **Perfect Rationality** ( Agent knows all & correct action)
      - Humans do not satisfy this rationality
    - **Bounded Rationality-**
      - Human use approximations

Definition of Rational Agent:

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure.

**Rational=best? Yes, but best of its knowledge.**

**Rational=Optimal?**

**Yes, to the best of its abilities & constraints (Subject to resources)**

# What is an Intelligent Agent - Agent Function

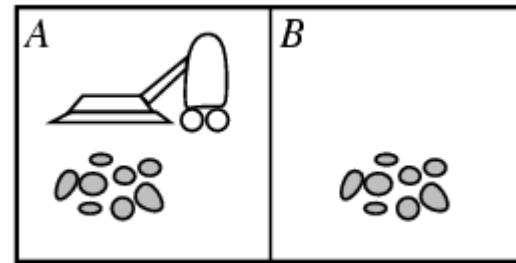
- Agent Function (percepts ==> actions)
  - Maps from percept histories to actions  $f: \mathcal{P}^* \rightarrow \mathcal{A}$
  - The **agent program** runs on the physical **architecture** to produce the function  $f$

agent = architecture + program

```
Action := Function(Percept Sequence)
If (Percept Sequence) then do Action
```

- Example: A Simple Agent Function for Vacuum World
  - If (current square is dirty) then suck
  - Else move to adjacent square

# Example: Vacuum Cleaner Agent



- i **Percepts:** location and contents, e.g., [A, Dirty]
- i **Actions:** Left, Right, Suck, NoOp

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
:	:

# What is an Intelligent Agent

- Limited Rationality
  - limited sensors, actuators, and computing power may make Rationality impossible
  - Theory of NP-completeness: some problems are likely impossible to solve quickly on ANY computer
  - Both natural and artificial intelligence are always limited

**Degree of Rationality:** the degree to which the agent's internal "thinking" maximizes its performance measure, given

- the available sensors
- the available actuators
- the available computing power
- the available built-in knowledge

# PEAS Analysis

- To design a rational agent, we must specify the **task environment**.
- **PEAS Analysis:**
  - Specify Performance Measure
  - Environment
  - Actuators
  - Sensors

# PEAS Analysis – Examples

- Agent: Medical diagnosis system
  - **Performance measure:** Healthy patient, minimize costs
  - **Environment:** Patient, hospital, staff
  - **Actuators:** Screen display (questions, tests, diagnoses, treatments, referrals)
  - **Sensors:** Keyboard (entry of symptoms, findings, patient's answers)
- Agent: Part-picking robot
  - **Performance measure:** Percentage of parts in correct bins
  - **Environment:** Conveyor belt with parts, bins
  - **Actuators:** Jointed arm and hand
  - **Sensors:** Camera, joint angle sensors

## PEAS

To design a rational agent, we must specify the **task environment**

Consider, e.g., the task of designing an automated taxi:

Performance measure??

Environment??

Actuators??

Sensors??

## PEAS

To design a rational agent, we must specify the **task environment**

Consider, e.g., the task of designing an automated taxi:

Performance measure?? safety, destination, profits, legality, comfort, ...

Environment?? US streets/freeways, traffic, pedestrians, weather, ...

Actuators?? steering, accelerator, brake, horn, speaker/display, ...

Sensors?? video, accelerometers, gauges, engine sensors, keyboard, GPS, ...

## PEAS Analysis – More Examples

- Agent: Internet Shopping Agent

- Performance measure??
- Environment??
- Actuators??
- Sensors??

# Environment

- Environment in which agent operates
  - Absolute
  - Agents point of view

# Environment Types

## Fully observable (vs. partially observable):

- An agent's sensors give it access to the complete state of the environment at each point in time.
- It is convenient bcoz agent need not maintain any internal state to keep track of the world.
- **Ex. Chess (Ex: Deep Blue)**
- **Partially Observable**:: When Noisy & inaccurate sensors or part of state r missing from the sensor data. (ex- Vacuum agent with only a local dirt sensor cannot tell whether there is dirt in other squares, & automated taxi cannot see what other drivers are thinking)
- **Ex. Poker**

## Deterministic (vs. stochastic):

- Deterministic AI environments are those on which the outcome can be determined base on a specific state. In other words, deterministic environments ignore uncertainty.
- Most real world AI environments are not deterministic. Instead, they can be classified as stochastic.
- **Ex (stochastic ): Self-driving vehicles are a classic example of stochastic AI processes.**

## Environment Types (cont.)

- **Episodic** (vs. sequential):
- Episodic is an environment where each state is independent of each other. The action on a state has nothing to do with the next state.
- **Real-life Example:** A support bot (agent) answer to a question and then answer to another question and so on. So each question-answer is a single episode.
- The sequential environment is an environment where the next state is dependent on the current action. So agent current action can change all of the future states of the environment.
- **Real-life Example:** Playing tennis is a perfect example where a player observes the opponent's shot and takes action.

- **Static** (vs. dynamic):
- The **Static** environment is completely unchanged while an agent is perceiving the environment.
- **Real-life Example:** Cleaning a room (Environment) by a dry-cleaner Agent is an example of a static environment where the room is static while cleaning.
- **Dynamic** Environment could be changed while an agent is perceiving the environment. So agents keep looking at the environment while taking action.
- **Real-life Example:** Playing soccer is a dynamic environment where players' positions keep changing throughout the game. So a player hits the ball by observing the opposite team.

- **Discrete** (vs. continuous):
- **Discrete Environment** consists of a finite number of states and agents have a finite number of actions.
- **Real-life Example:** Choices of a move (action) in a tic-tac game are finite on a finite number of boxes on the board (Environment).
- While in a **Continuous environment**, the environment can have an infinite number of states. So the possibilities of taking an action are also infinite.
- **Real-life Example:** In a basketball game, the position of players (**Environment**) keeps changing continuously and hitting (**Action**) the ball towards the basket can have different angles and speed so infinite possibilities.

# Environment Types (cont.)

- **Complete vs. Incomplete**

Complete AI environments are those on which, at any give time, we have enough information to complete a branch of the problem.

Ex: Chess is a classic example of a complete AI environment.

Ex: Poker, on the other hand, is an **incomplete environments** as AI strategies can't anticipate many moves in advance and, instead, they focus on finding a good 'equilibrium' at any given time.

- **Single agent** (vs. multi-agent):

- An agent operating by itself in an environment.

# Environment Types (cont.)

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>	Yes	No	Partly	No
<u>Episodic??</u>	No	No	No	No
<u>Static??</u>	Yes	Semi	Semi	No
<u>Discrete??</u>	Yes	Yes	Yes	No
<u>Single-agent??</u>	Yes	No	Yes (except auctions)	No

The environment type largely determines the agent design.

The real world is (of course) partially observable, stochastic, sequential, dynamic, continuous, multi-agent

Examples	Fully vs Partially Observable	Deterministic vs Stochastic	Episodic vs Sequential	Static vs Dynamic	Discrete vs Continuous	Single vs Multi Agents
Brushing Your Teeth	Fully	Stochastic	Sequential	Static	Continuous	Single
Playing Chess	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi-Agent
Playing Cards	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi-Agent
Playing	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi Agent
Autonomous Vehicles	Fully	Stochastic	Sequential	Dynamic	Continuous	Multi-Agent
Order in Restaurant	Fully	Deterministic	Episodic	Static	Discrete	Single Agent

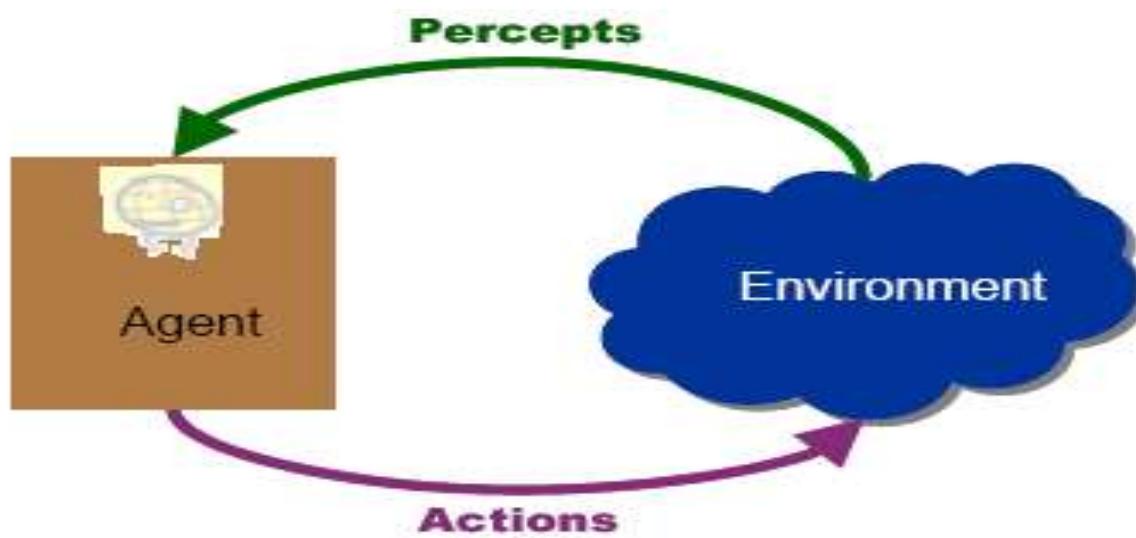
# End of First Lecture

- Thanks

# Agent types

- Four basic types:
  - Simple reflex agents
  - Model-based reflex agents
  - Goal-based agents
  - Utility-based agents

# Agents



An **agent** is any thing that can be viewed as **perceiving** its **environment** through **sensors** and **acting** upon that environment through **actuators/ effectors**

Ex: Human Being , Calculator etc

Agent has goal –the objective which agent has to satisfy

Actions can potentially change the environment

Agent perceive current percept or sequence of perceptions

# Structure of an Intelligent Agent

- All agents have the same basic structure:
  - accept percepts from environment
  - generate actions
- A Skeleton Agent:

```
function Skeleton-Agent(percept) returns action
    static: memory, the agent's memory of the
           world

    memory  $\leftarrow$  Update-Memory(memory, percept)
    action  $\leftarrow$  Choose-Best-Action(memory)
    memory  $\leftarrow$  Update-Memory(memory, action)
```

- Observations: *return action*
  - Agent may or may not build percept sequence in memory (depends on domain)
  - Performance measure is not part of the agent; it is applied externally to judge the success of the agent

# Agent Types

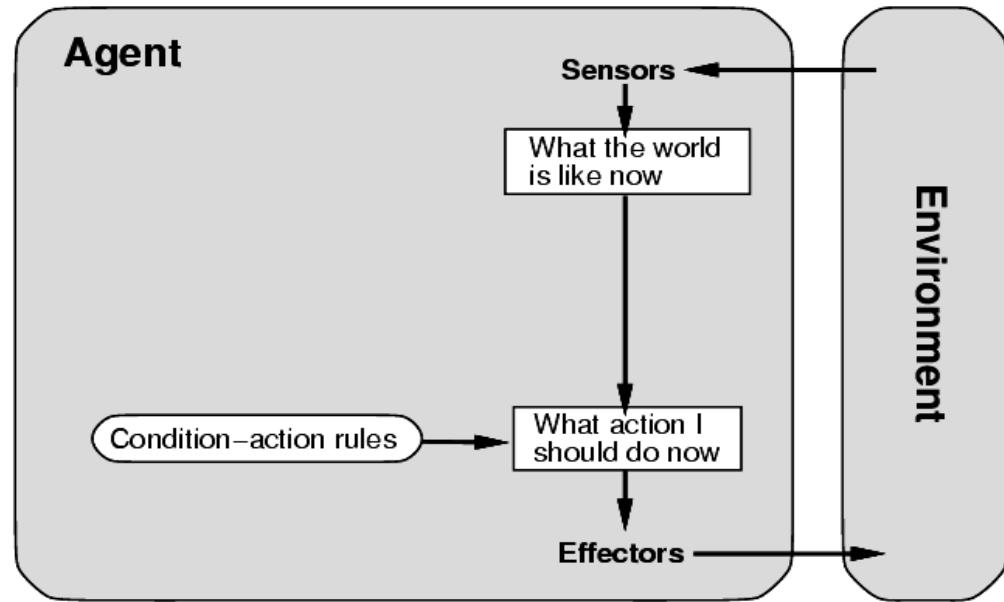
- Simple reflex agents
  - These are based on **condition-action rules** and implemented with an appropriate production system. They are **stateless devices which do not have memory** of past world states.
- Reflex Agents with memory (Model-Based)
  - have internal state which is used to keep track of past states of the world.
- Agents with goals
  - are agents which in addition to state information have a kind of goal information which describes desirable situations.
  - Agents of this kind take **future events into consideration**.
- Utility-based agents
  - base their decision on classic axiomatic utility-theory in order to **act rationally**.

Note: All of these can be turned into “learning”  
agents

# A Simple Reflex Agent

- We can summarize part of the table by formulating commonly occurring patterns as condition-action rules:
- Example:
 

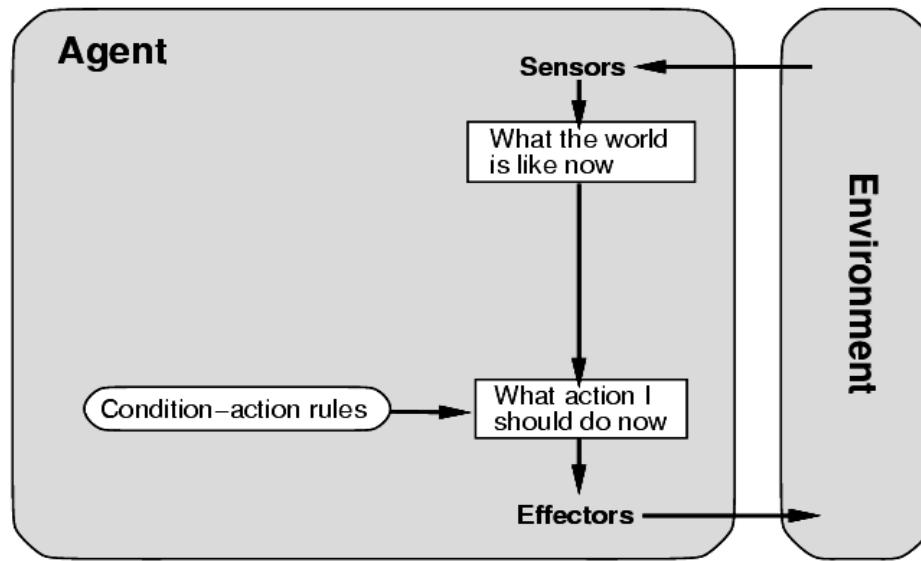
if *car-in-front-brakes*  
then *initiate braking*
- Agent works by finding a rule whose condition matches the current situation
  - rule-based systems
- **But, this only works if the current percept is sufficient for making the correct decision**



```

function Simple-Reflex-Agent(percept) returns action
  static: rules, a set of condition-action rules
  state  $\leftarrow$  Interpret-Input(percept)
  rule  $\leftarrow$  Rule-Match(state, rules)
  action  $\leftarrow$  Rule-Action[rule]
  return action
  
```

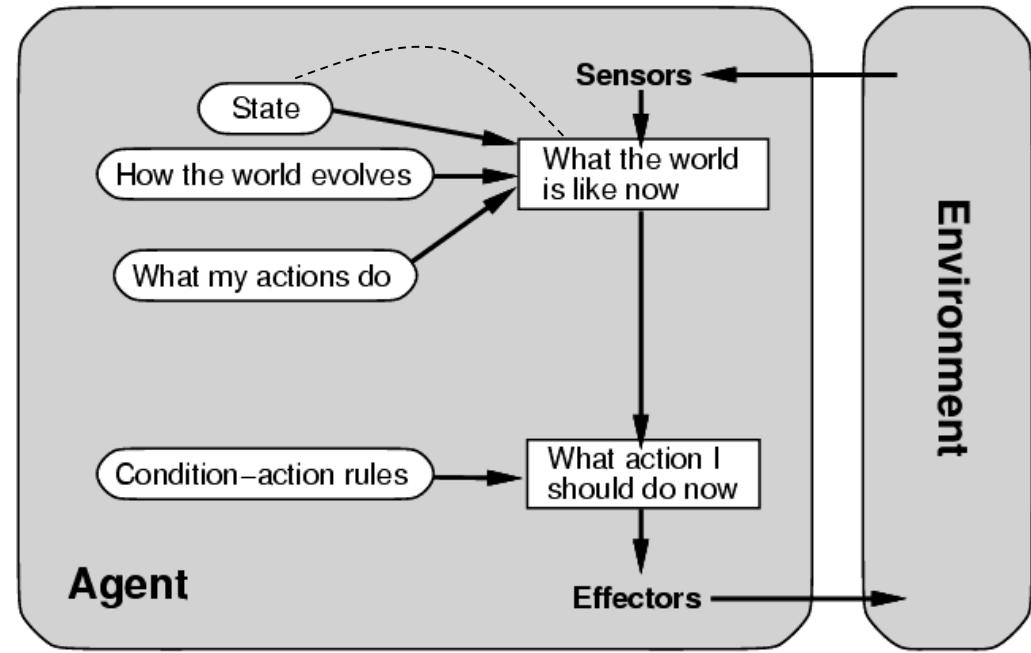
## Example: Simple Reflex Vacuum Agent



```
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
```

# Agents that Keep Track of the World

- Updating internal state requires two kinds of encoded knowledge
  - knowledge about how the world changes (independent of the agents' actions)
  - knowledge about how the agents' actions affect the world
- But, knowledge of the internal state is not always enough
  - how to choose among alternative decision paths (e.g., **where should the car go at an intersection?**)
  - Requires knowledge of the **goal** to be achieved

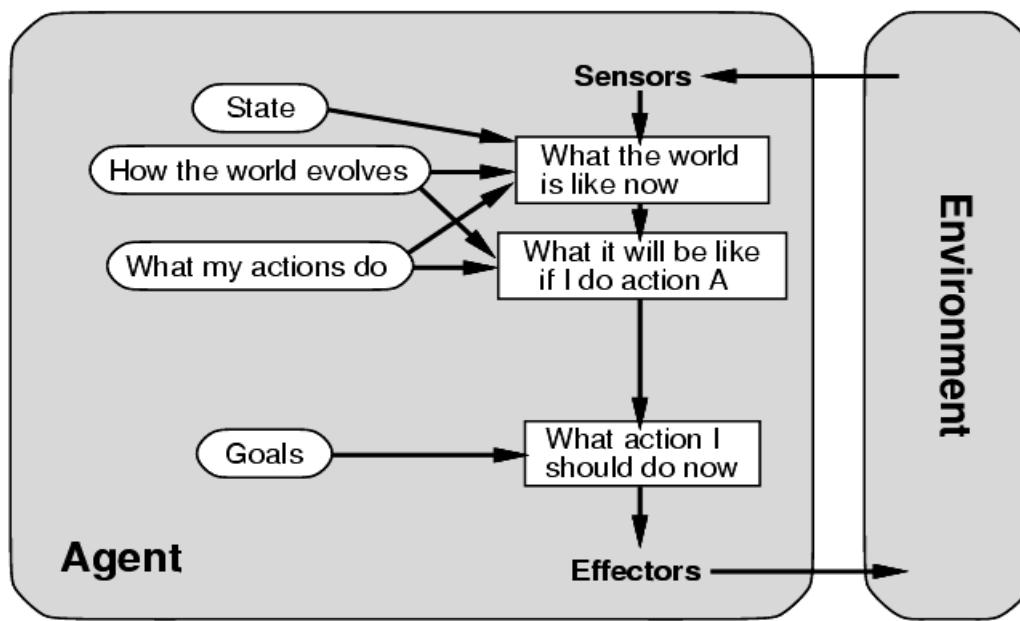


```

function Reflex-Agent-With-State(percept) returns action
static: rules, a set of condition-action rules
        state, a description of the current world

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
  
```

# Agents with Explicit Goals



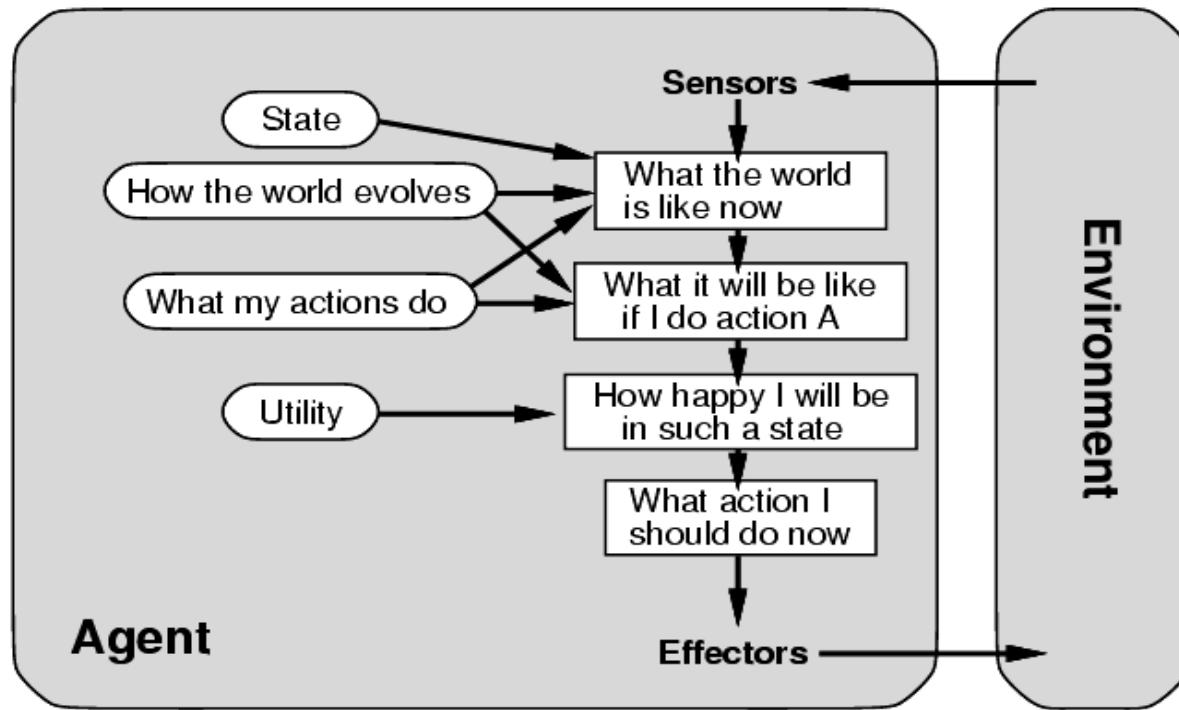
## i Reasoning about actions

- 4 reflex agents only act based on pre-computed knowledge (rules)
- 4 goal-based (planning) act by reasoning about which actions achieve the goal
- 4 more adaptive and flexible

# Agents with Explicit Goals

- Knowing current state is not always enough.
  - State allows an agent to keep track of unseen parts of the world, but the agent must update state based on knowledge of changes in the world and of effects of own actions.
  - Goal = description of desired situation
- Examples:
  - Decision to change lanes depends on a goal to go somewhere (and other factors);
  - Decision to put an item in shopping basket depends on a shopping list, map of store, knowledge of menu
- Notes:
  - **Search** (Russell Chapters 3-5) and **Planning** (Chapters 11-13) are concerned with finding sequences of actions to satisfy a goal.
  - Reflexive agent concerned with one action at a time.
  - Classical Planning: finding a sequence of actions that achieves a goal.
  - Contrast with condition-action rules: involves consideration of future "what will happen if I do ..." (fundamental difference).

# A Complete Utility-Based Agent



## i Utility Function

- 4 a mapping of states onto real numbers
- 4 allows rational decisions in two kinds of situations
  - h evaluation of the tradeoffs among conflicting goals
  - h evaluation of competing goals

# Utility-Based Agents (Cont.)

- Preferred world state has higher utility for agent = quality of being useful
- Examples
  - quicker, safer, more reliable ways to get where going;
  - price comparison shopping
  - bidding on items in an auction
  - evaluating bids in an auction
- Utility function: state ==>  $U(\text{state})$  = measure of happiness

# Shopping Agent Example

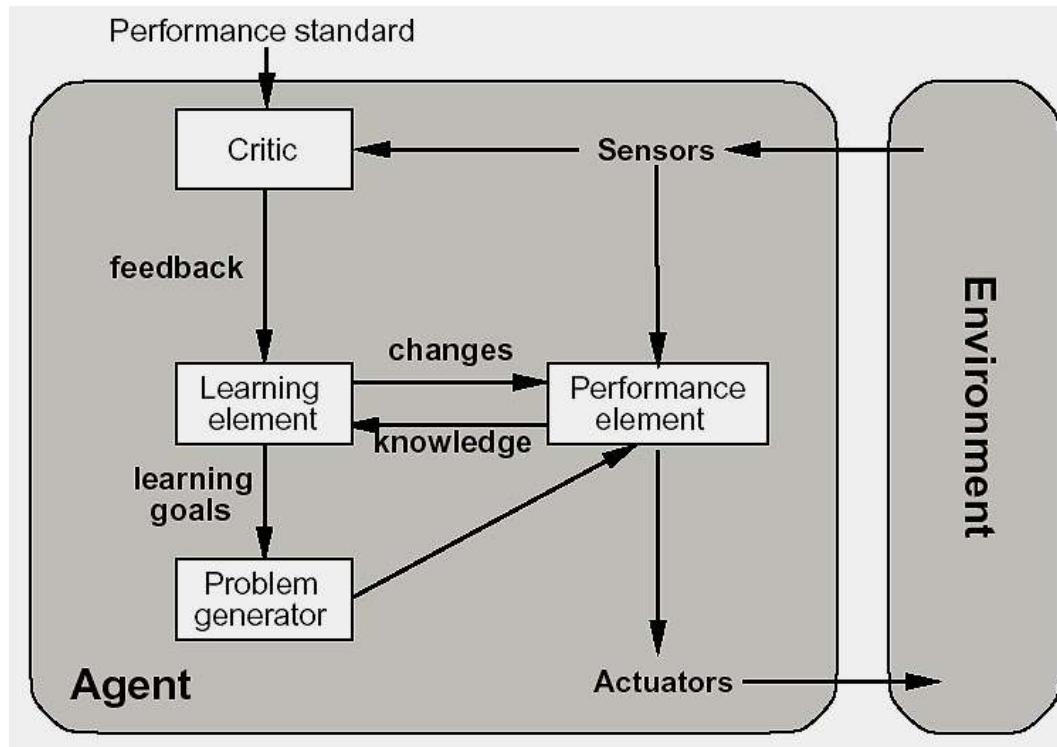
- Navigating: Move around store; avoid obstacles
  - Reflex agent: store map precompiled.
  - Goal-based agent: create an internal map, reason explicitly about it, use signs and adapt to changes () .
- Gathering: Find and put into cart groceries it wants, need to induce objects from percepts.
  - Reflex agent: wander and grab items that look good.
  - Goal-based agent: shopping list.
- Menu-planning: Generate shopping list, modify list if store is out of some item.
  - Goal-based agent: required; what happens when a needed item is not there? Achieve the goal some other way. e.g., no milk cartons: get canned milk or powdered milk.

# General Architecture for Goal-Based Agents

```
Input percept
state ← Update-State(state, percept)
goal ← Formulate-Goal(state, perf-measure)
search-space ← Formulate-Problem (state,
goal)
plan ← Search(search-space , goal)
while (plan not empty) do
    action ← Recommendation(plan, state)
    plan ← Remainder(plan, state)
    output action
end
```

- Simple agents do not have access to their own performance measure
  - In this case the designer will "hard wire" a goal for the agent, i.e. the designer will choose the goal and build it into the agent
- Similarly, unintelligent agents cannot formulate their own problem
  - this formulation must be built-in also
- The while loop above is the "execution phase" of this agent's behavior
  - Note that this architecture assumes that the execution phase does not require monitoring of the environment

# Learning Agents



## | Four main components:

- 4 Performance element: the agent function
- 4 Learning element: responsible for making improvements by observing performance
- 4 Critic: gives feedback to learning element by measuring agent's performance
- 4 Problem generator: suggest other possible courses of actions (exploration)

# Intelligent Agent Summary

- An agent perceives and acts in an environment. It has an architecture and is implemented by a program.
- An ideal agent always chooses the action which maximizes its expected performance, given the percept sequence received so far.
- An autonomous agent uses its own experience rather than built-in knowledge of the environment by the designer.
- An agent program maps from a percept to an action and updates its internal state.
- Reflex agents respond immediately to percepts.
- Goal-based agents act in order to achieve their goal(s).
- Utility-based agents maximize their own utility function.

# Exercise

- News Filtering Internet Agent
  - uses a static user profile (e.g., a set of keywords specified by the user)
  - on a regular basis, searches a specified news site (e.g., Reuters or AP) for news stories that match the user profile
  - can search through the site by following links from page to page
  - presents a set of links to the matching stories that have not been read before (matching based on the number of words from the profile occurring in the news story)
- (1) Give a detailed PEAS description for the news filtering agent
- (2) Characterize the environment type (as being observable, deterministic, episodic, static, etc).

# AI Problems

- water jug problem in Artificial Intelligence
- cannibals and missionaries problem in AI
- tic tac toe problem in artificial intelligence
- 8/16 puzzle problem in artificial intelligence
- tower of hanoi problem in artificial intelligence

Thanks End of Lecture-2

- Agent Types
- Environment types

# Search Strategies

- Problem solving and formulating a problem State Space Search- Uninformed and Informed Search Techniques,
- Heuristic function,
- A\*,
- AO\* algorithms ,
- Hill climbing,
- simulated annealing,
- genetic algorithms ,
- Constraint satisfaction method

# Search Strategies

- Uninformed Search
  - breadth-first
  - depth-first
  - uniform-cost search
  - depth-limited search
  - iterative deepening
  - bi-directional search
  - constraint satisfaction
- Informed Search
  - best-first search
  - search with heuristics
  - ~~memory bounded search~~
  - ~~iterative improvement search~~

# Introduction to State Space Search

## 2.2 State space search

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

# State Space Search Notations

Let us begin by introducing certain terms.

An initial state is the description of the starting configuration of the agent

An action or an operator takes the agent from one state to another state which is called a successor state. A state can have a number of successor states.

A plan is a sequence of actions. The cost of a plan is referred to as the path cost. The path cost is a positive number, and a common path cost may be the sum of the costs of the steps in the path.

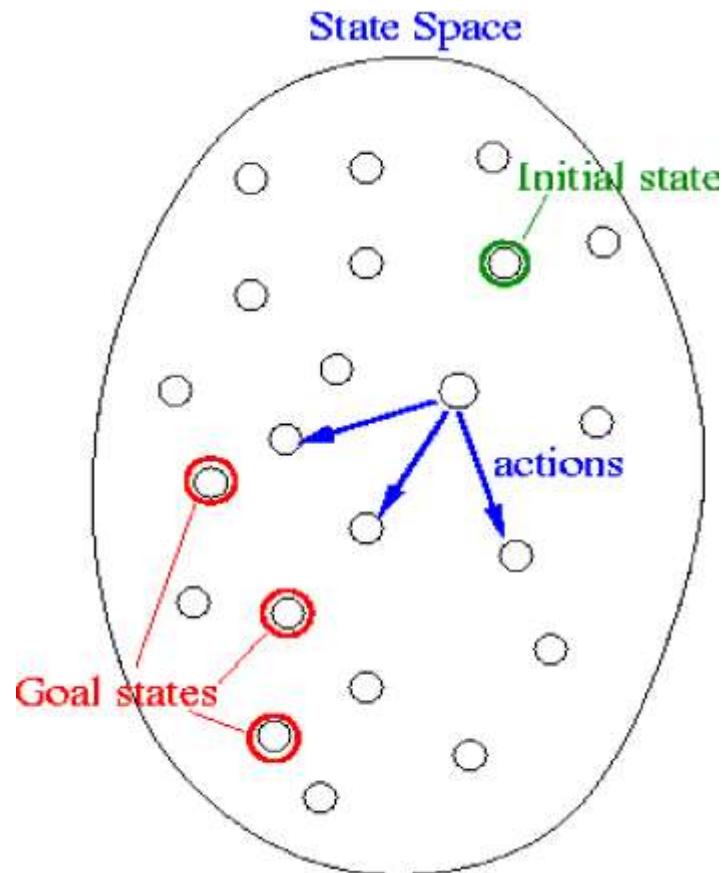
*Search* is the process of considering various possible sequences of operators applied to the initial state, and finding out a sequence which culminates in a goal state.

# Search Problem

We are now ready to formally describe a search problem.

A search problem consists of the following:

- $S$ : the full set of states
- $s^0$ : the initial state
- $A:S \rightarrow S$  is a set of operators
- $G$  is the set of final states. Note that  $G \subseteq S$



These are schematically depicted in above Figure

# Search Problem

The search problem is to find a sequence of actions which transforms the agent from the initial state to a goal state  $g \in G$ .

A search problem is represented by a 4-tuple  $\{S, s^0, A, G\}$ .

S: set of states

$s^0 \in S$  : initial state

A:  $S \times S$  operators/ actions that transform one state to another state

G : goal, a set of states.  $G \subseteq S$

This sequence of actions is called a solution plan. It is a path from the initial state to a goal state. A *plan P* is a sequence of actions.

$P = \{a^0, a^1, \dots, a^N\}$  which leads to traversing a number of states  $\{s^0, s^1, \dots, s^{N+1} \in G\}$ .

A sequence of states is called a path. The cost of a path is a positive number. In many cases the path cost is computed by taking the sum of the costs of each action.

# Representation of search problems

A search problem is represented using a directed graph.

- The states are represented as nodes.
- The allowed actions are represented as arcs.

# Searching process

The steps for generic searching process :

Do until a solution is found or the state space is exhausted.

1. Check the current state
2. Execute allowable actions to find the successor states.
3. Pick one of the new states.
4. Check if the new state is a solution state

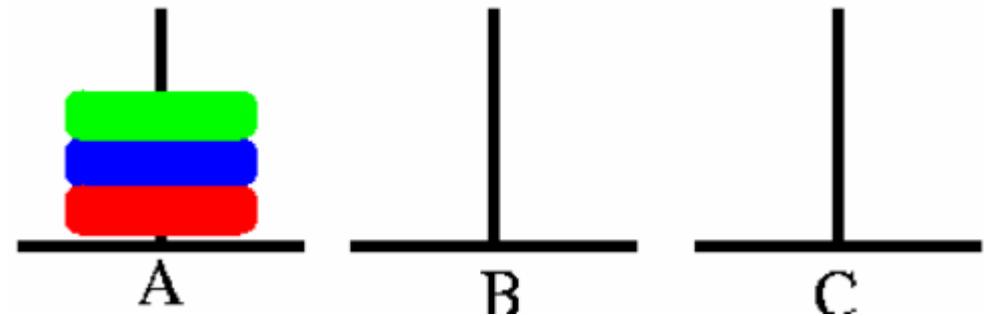
If it is not, the new state becomes the current state and the process is repeated

# Examples

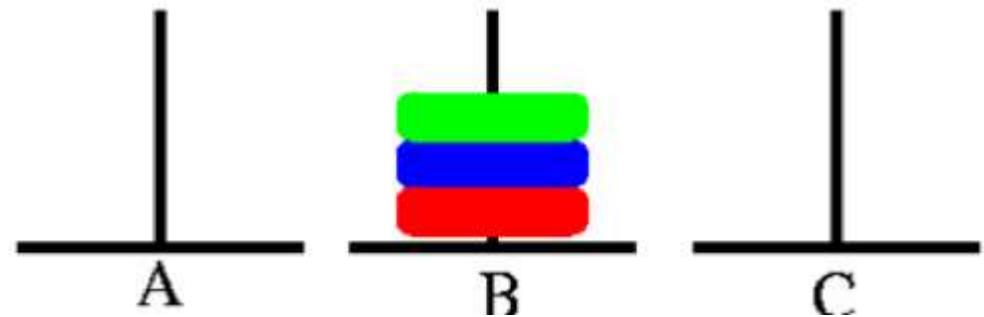
Illustration of a search process

## Example problem: Pegs and Disks problem

The initial state



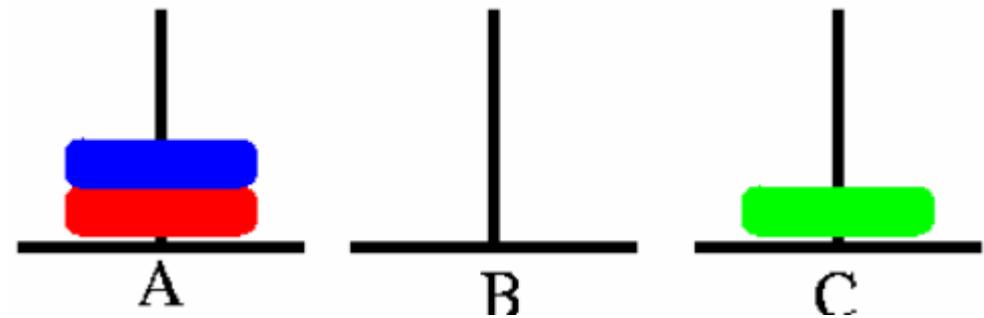
Goal State



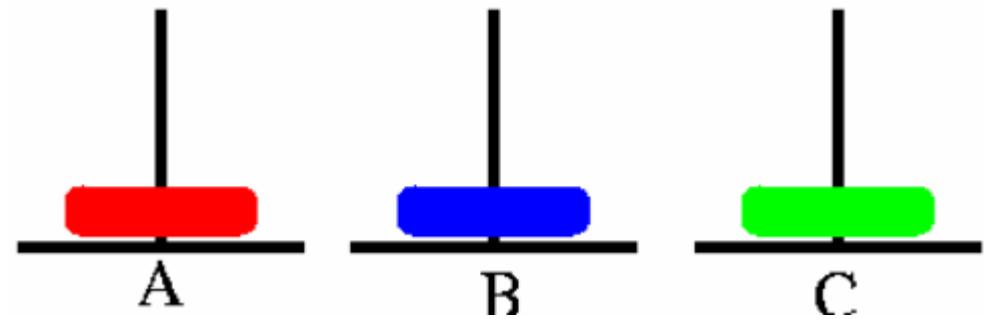
# Example problem: Pegs and Disks problem

Now we will describe a sequence of actions that can be applied on the initial state.

Step 1: Move A → C

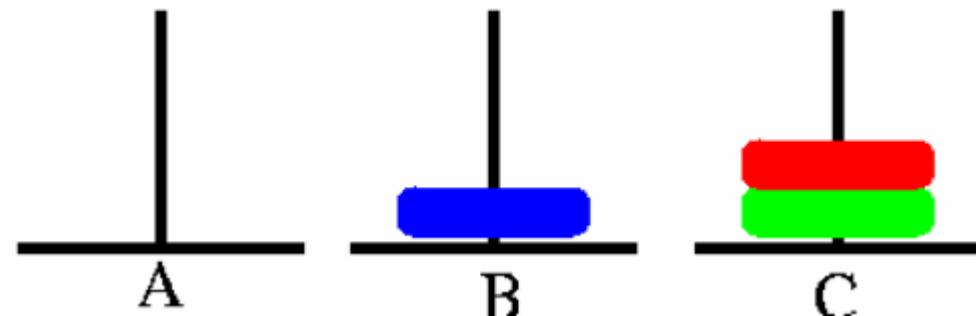


Step 2: Move A → B

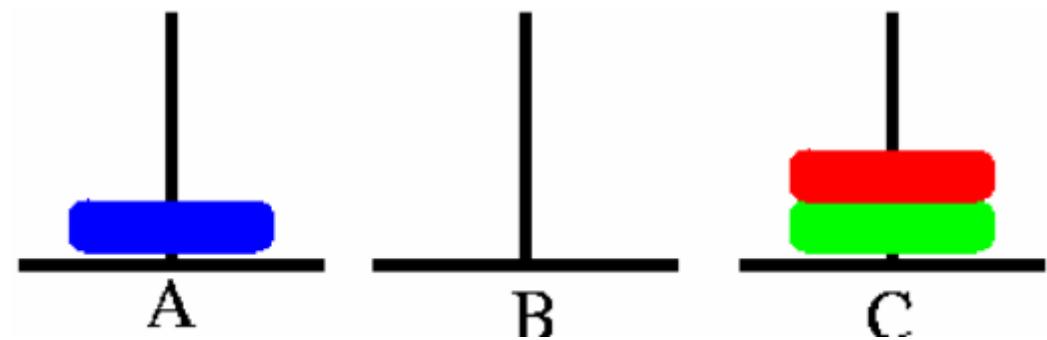


## Example problem: Pegs and Disks problem

Step 3: Move A → C

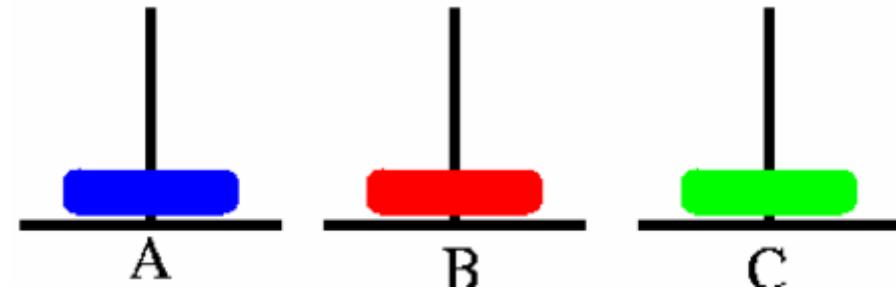


Step 4: Move B → A

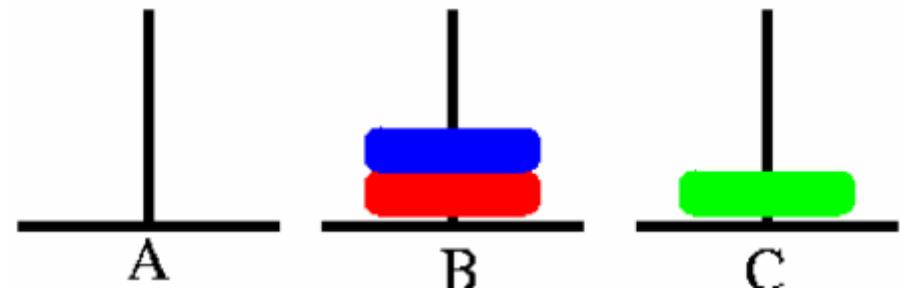


## Example problem: Pegs and Disks problem

- Step 5: Move C → B

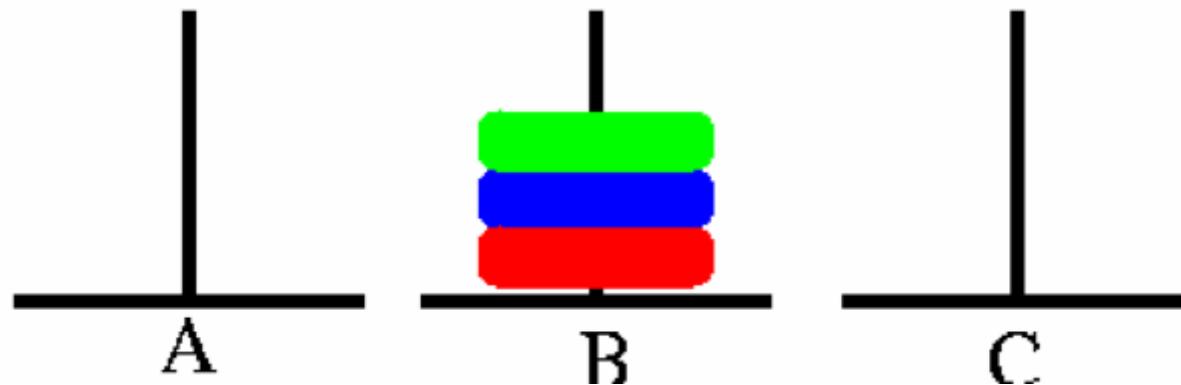


Step 6: Move A → B

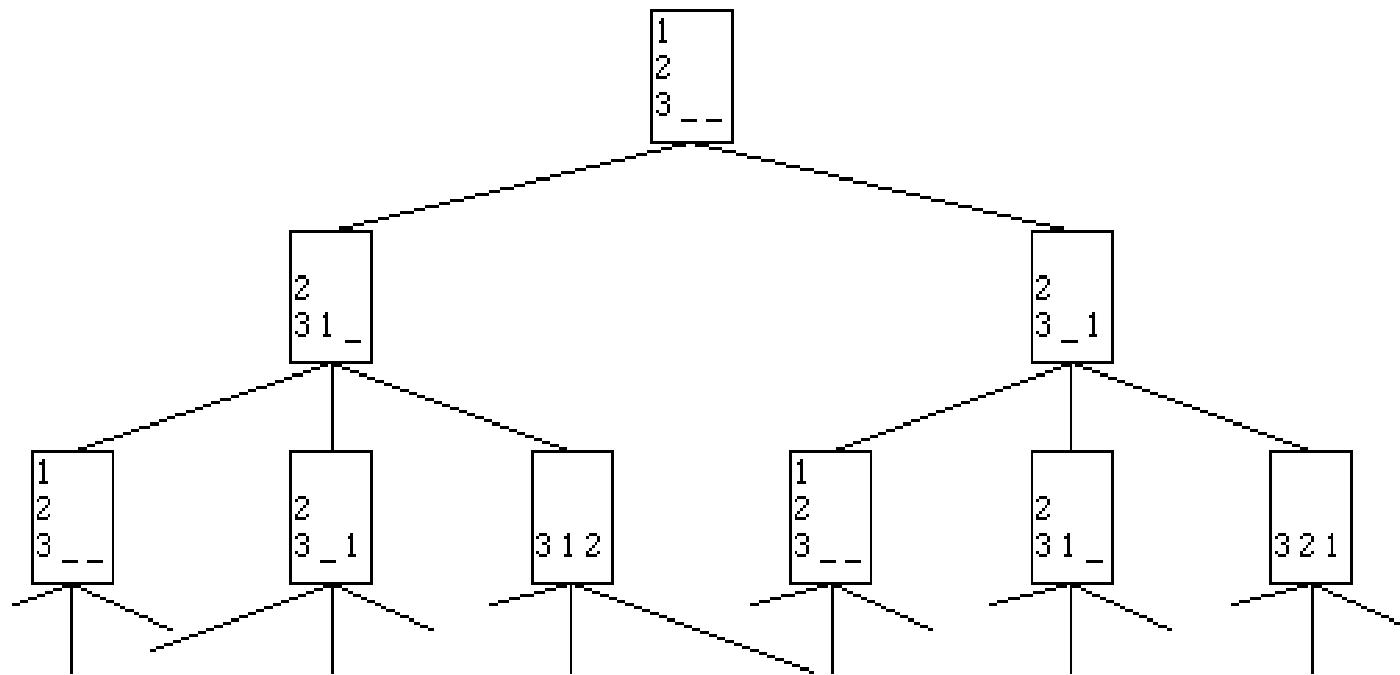


## Example problem: Pegs and Disks problem

- Step 7: Move C → B



# Example problem: Pegs and Disks problem



# Search

Searching through a state space involves the following:

1. A set of states
2. Operators and their costs
3. Start state
4. A test to check for goal state

We will now outline the basic search algorithm, and then consider various variations of this algorithm.

# The basic search algorithm

Let L be a list containing the initial state (L= the fringe)

Loop if L is empty return failure

    Node  $\leftarrow$  select (L)

        if Node is a goal

            then return Node

                (the path from initial state to Node)

        else

            generate all successors of Node, and  
            merge the newly generated states into L

End Loop

In addition the search algorithm maintains a list of nodes called the fringe(open list). The fringe keeps track of the nodes that have been generated but are yet to be explored.

# Evaluating Search strategies

What are the characteristics of the different search algorithms and what is their efficiency? We will look at the following three factors to measure this.

1. **Completeness:** Is the strategy guaranteed to find a solution if one exists?
2. **Optimality:** Does the solution have low cost or the minimal cost?
3. What is the search cost associated with the **time and memory required** to find a solution?
  - a. Time complexity: Time taken (number of nodes expanded) (worst or average case) to find a solution.
  - b. Space complexity: Space used by the algorithm measured in terms of the maximum size of fringe

# The different search strategies

## 1. Blind Search strategies or Uninformed search

- a. Depth first search
- b. Breadth first search
- c. Depth Limited search
- d. Iterative deepening search

## 2. Informed Search

## 3. Constraint Satisfaction Search

## 4. Adversary Search

# Search Tree – Terminology

- **Root Node:** The node from which the search starts, initial state of state space.
- **Leaf Node:** A node in the search tree having no children.
  - not yet expanded (i.e., in fringe) or
  - having no successors (i.e., “dead-ends”)
- **Ancestor/Descendant:** X is an ancestor of Y if either X is Y’s parent or X is an ancestor of the parent of Y. If X is an ancestor of Y, Y is said to be a descendant of X.
- **Branching factor:** the maximum number of children of a non-leaf node in the search tree
- **Path:** A path in the search tree is a complete path if it begins with the start node and ends with a goal node. Otherwise it is a partial path.

We also need to introduce some data structures that will be used in the search algorithms we call it Fringe/open list & Closed List.

Search tree may be infinite because of loops inside the state space even if state space is small

# Node data structure

A node used in the search algorithm is a data structure which contains the following:

1. A state description
2. A pointer to the parent of the node
3. Depth of the node
4. The operator that generated this node
5. Cost of this path (sum of operator costs) from the start state

The nodes that the algorithm has generated are kept in a data structure called OPEN or fringe. Initially only the start node is in OPEN.

# Comparing Uninformed Search Strategies

- Completeness
  - Will a solution always be found if one exists?
- Time
  - How long does it take to find the solution?
  - Often represented as the number of nodes searched
- Space
  - How much memory is needed to perform the search?
  - Often represented as the maximum number of nodes stored at once
- Optimal
  - Will the optimal (least cost) solution be found?

# Comparing Uninformed Search Strategies

- Time and space complexity are measured in
  - $b$  – maximum branching factor of the search tree
  - $m$  – maximum depth of the state space
  - $d$  – depth of the least cost solution

# Breadth-First Search

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

# Breadth First Search

## Algorithm Breadth first search

Let *fringe* be a list containing the initial state

Loop

    if *fringe* is empty return failure

        Node  $\leftarrow$  remove-first (*fringe*)

        if Node is a goal

            then return the path from initial state to Node

            else generate all successors of Node, and

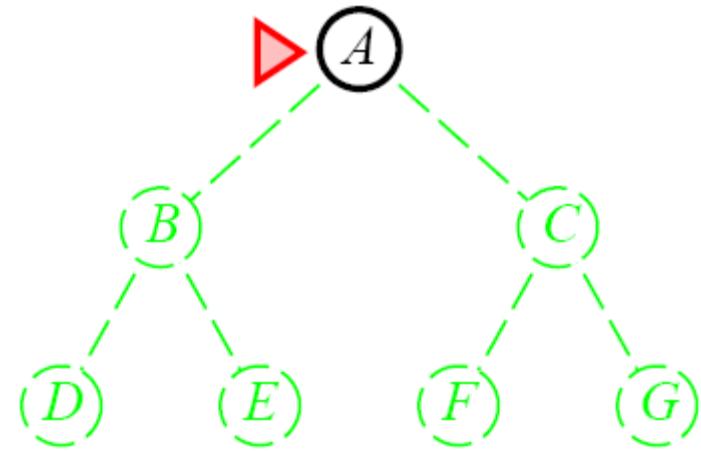
                (merge the newly generated nodes into *fringe*)

                add generated nodes to the back of *fringe*

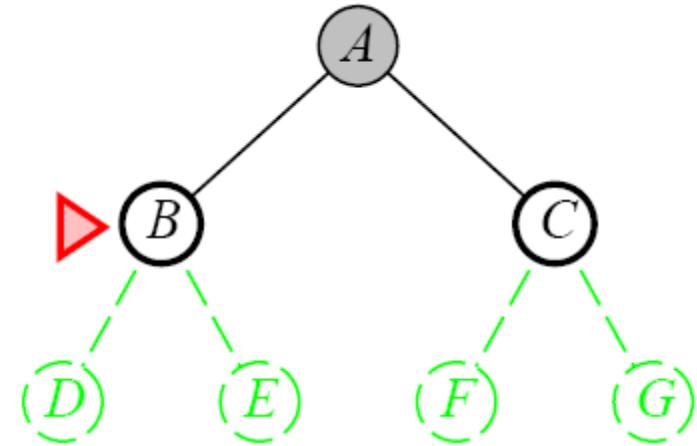
End Loop

Note that in breadth first search the newly generated nodes are put at the back of fringe or the OPEN list. The nodes will be expanded in a FIFO (First In First Out) order. The node that enters OPEN earlier will be expanded earlier.

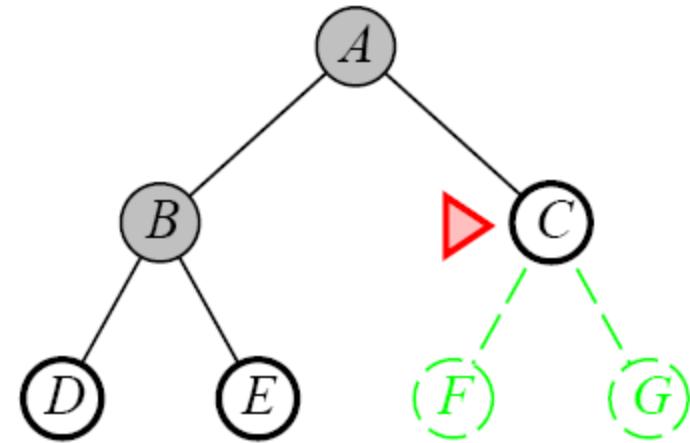
# Breadth-First Search



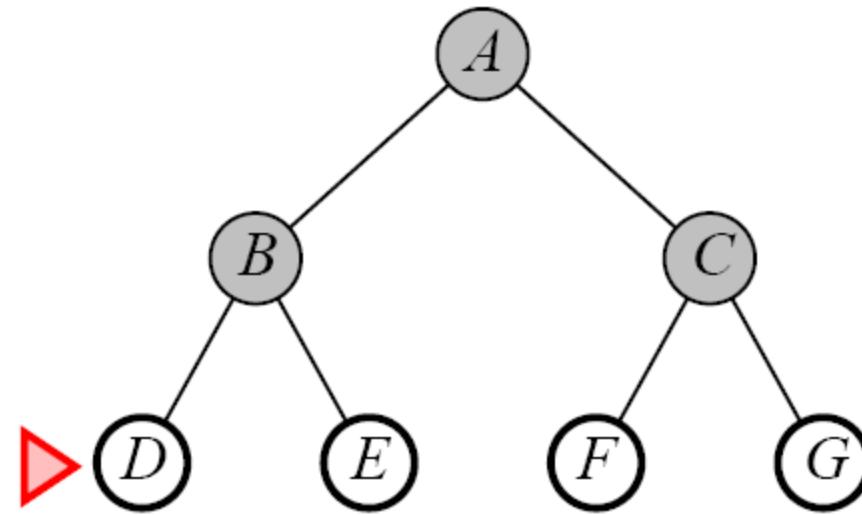
# Breadth-First Search



# Breadth-First Search

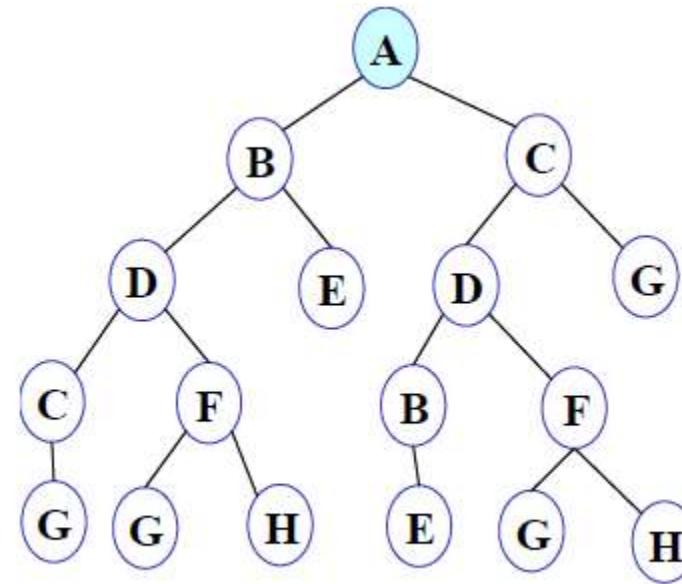


# Breadth-First Search



# BFS illustrated

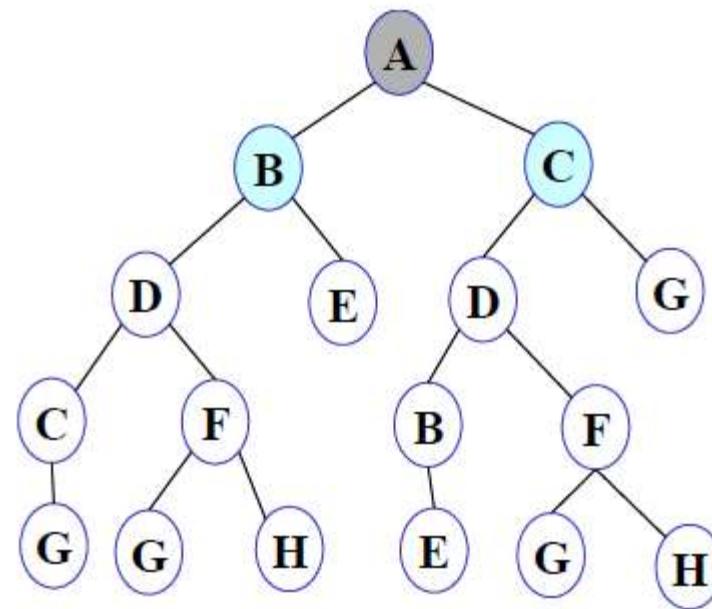
Step 1: Initially fringe contains only one node corresponding to the source state A.



FRINGE: A

Figure 3

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

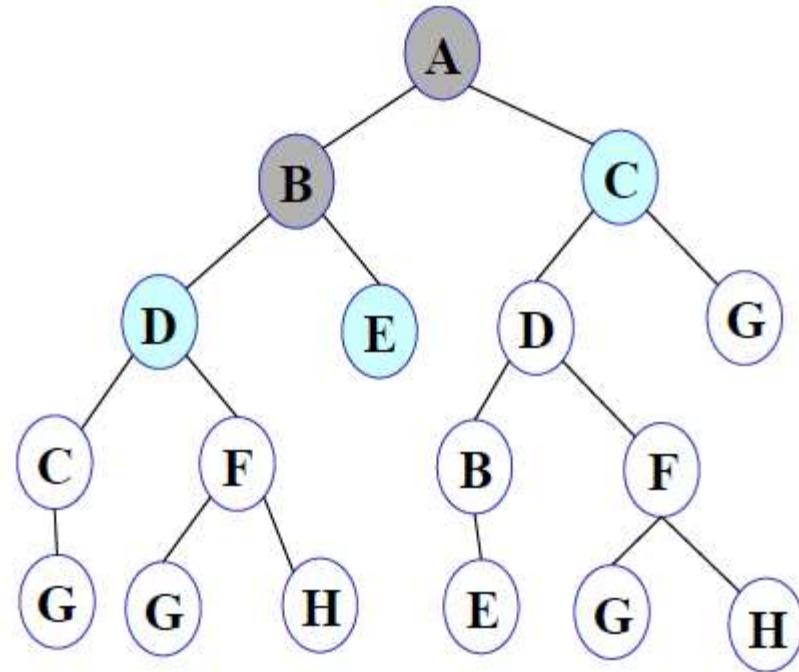


**Figure 4**

FRINGE: B C

Step 3: Node B is removed from fringe and is expanded. Its children D, E are generated and put at the back of fringe.

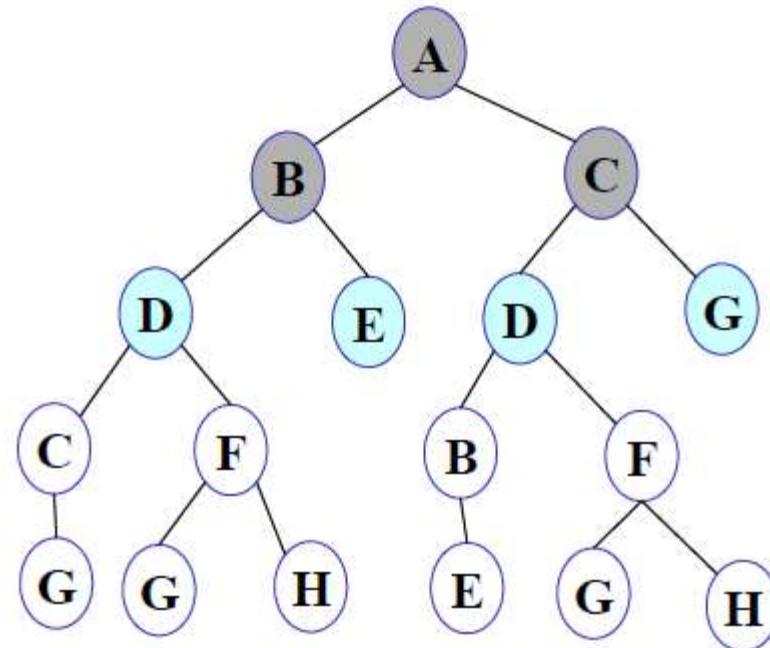
Version



**Figure 5**

FRINGE: C D E

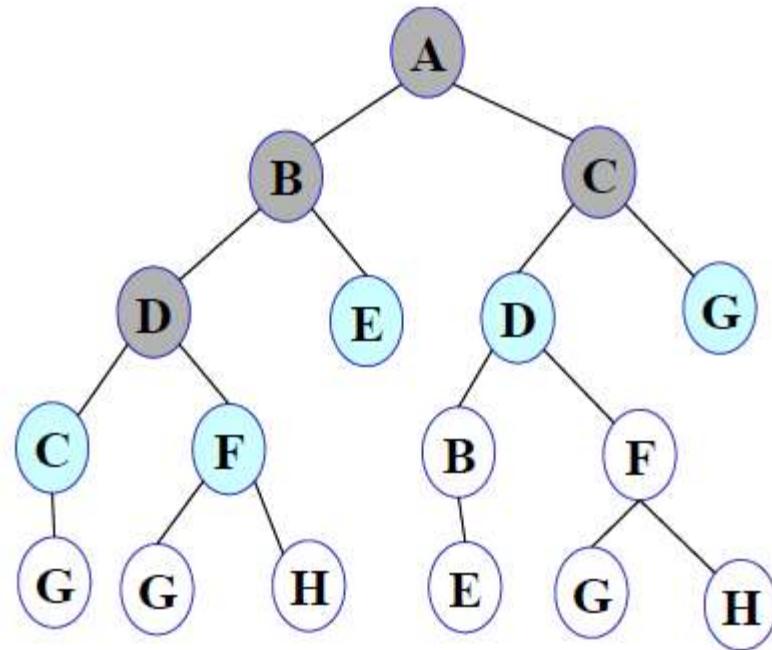
Step 4: Node C is removed from fringe and is expanded. Its children D and G are added to the back of fringe.



**Figure 6**

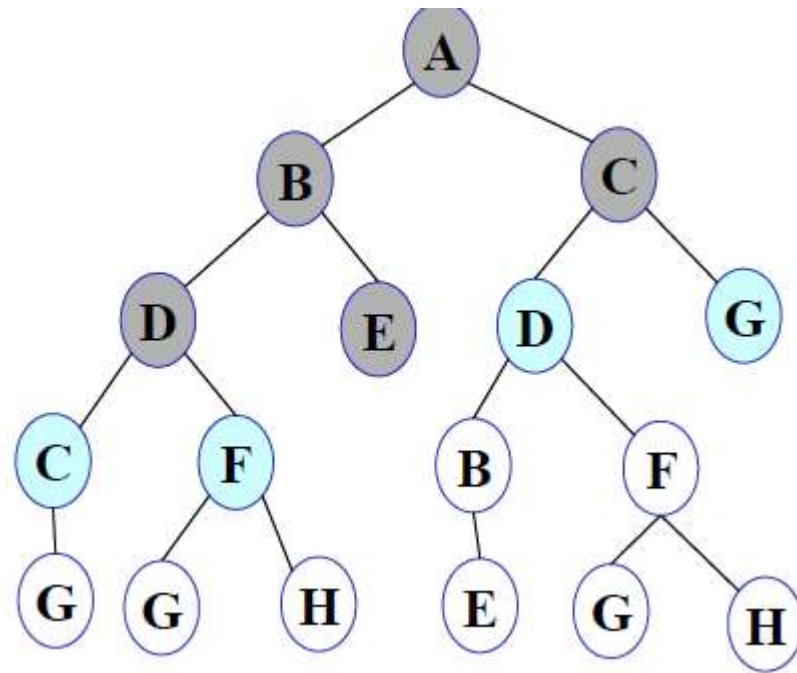
FRINGE: D E D G

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



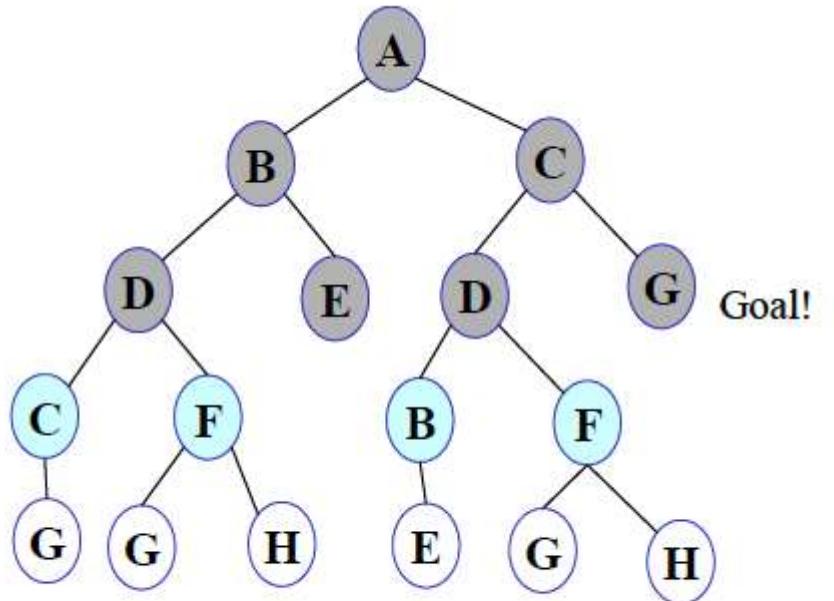
FRINGE: E D G C F

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



FRINGE: D G C F

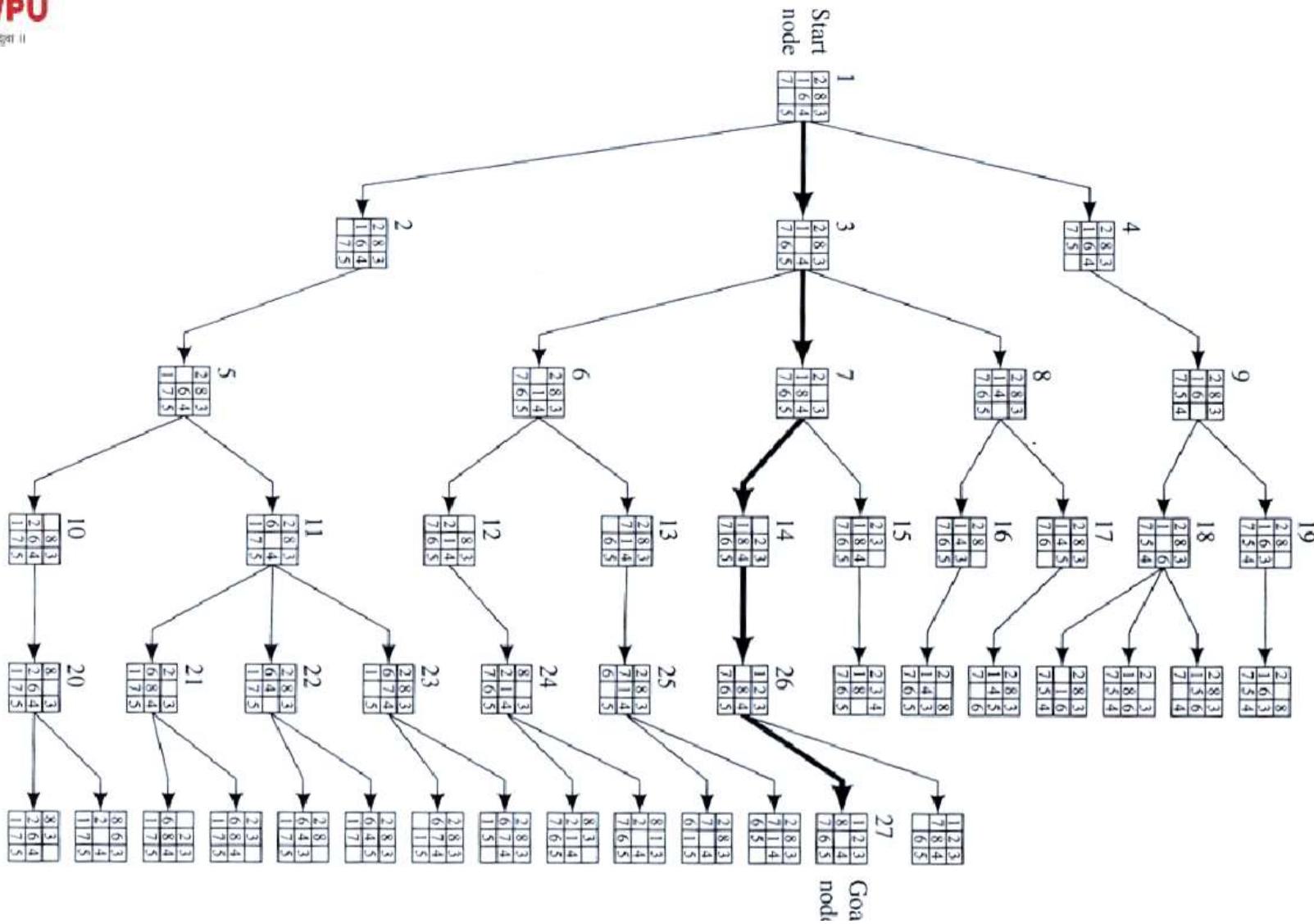
Step 7: D is expanded, B and F are put in OPEN.



**Figure 8**

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

# Example BFS

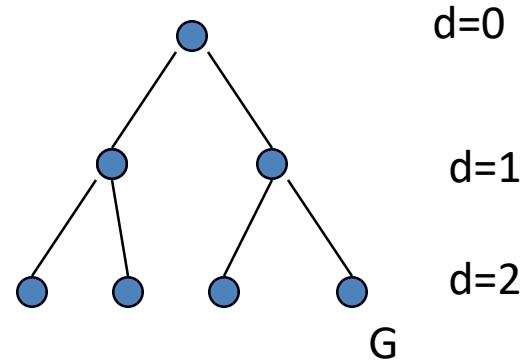


## What is the Complexity of Breadth-First Search?

- **Time Complexity**

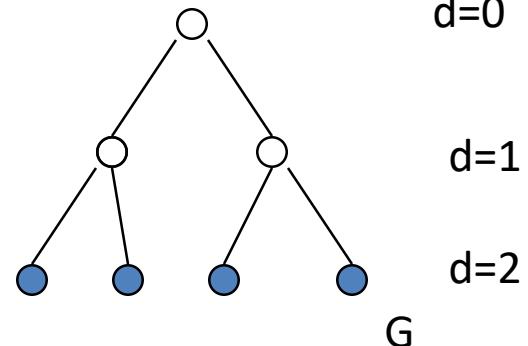
- assume (worst case) that there is 1 goal leaf at the RHS
- so BFS will expand all nodes

$$\begin{aligned}
 &= 1 + b + b^2 + \dots + b^d \\
 &= O(b^d)
 \end{aligned}$$



- **Space Complexity**

- how many nodes can be in the queue (worst-case)?
- at depth  $d-1$  there are  $b^d$  unexpanded nodes in the Q =  $O(b^d)$



## Properties of Breadth-First Search

- **Time Complexity:** Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the  $d$ = depth of shallowest solution and  $b$  is a node at every state.  
$$T(b) = 1+b^2+b^3+\dots+b^d= O(b^d)$$
- **Space Complexity:** Space complexity of BFS algorithm is given by the Memory size of frontier which is  $O(b^d)$ .
- **Completeness:** BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.
- **Optimality:** BFS is optimal if path cost is a non-decreasing function of the depth of the node.

# Advantages & Disadvantages of Breadth First Search

## Advantages of Breadth First Search

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

## Disadvantages of Breadth First Search

1. It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
2. BFS needs lots of time if the solution is far away from the root node.

## Lessons From Breadth First Search

- The memory requirements are a bigger problem for breadth-first search than is execution time
- Exponential-complexity search problems cannot be solved by uninformed methods

# Depth-First Search

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a **stack data structure** for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

# Depth first Search

## Algorithm

Let *fringe* be a list containing the initial state

Loop

    if *fringe* is empty return failure

    Node $\leftarrow$  remove-first (*fringe*)

    if Node is a goal

        then return the path from initial state to Node

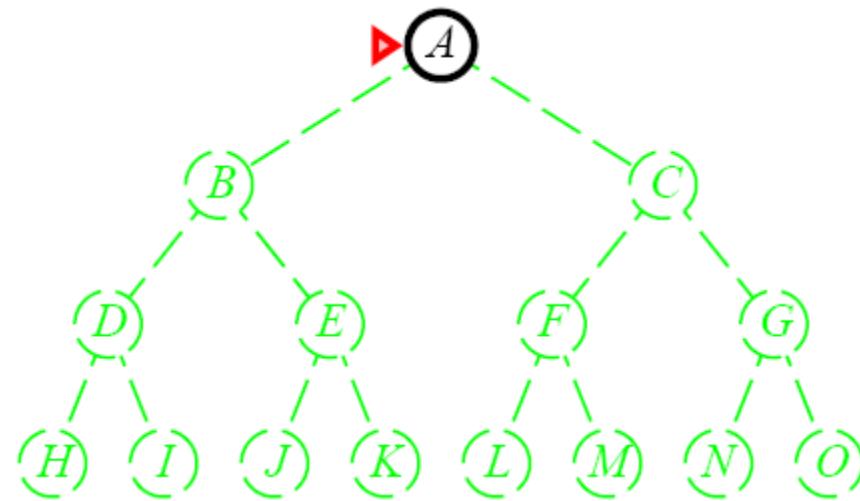
    else generate all successors of Node, and

        merge the newly generated nodes into *fringe*

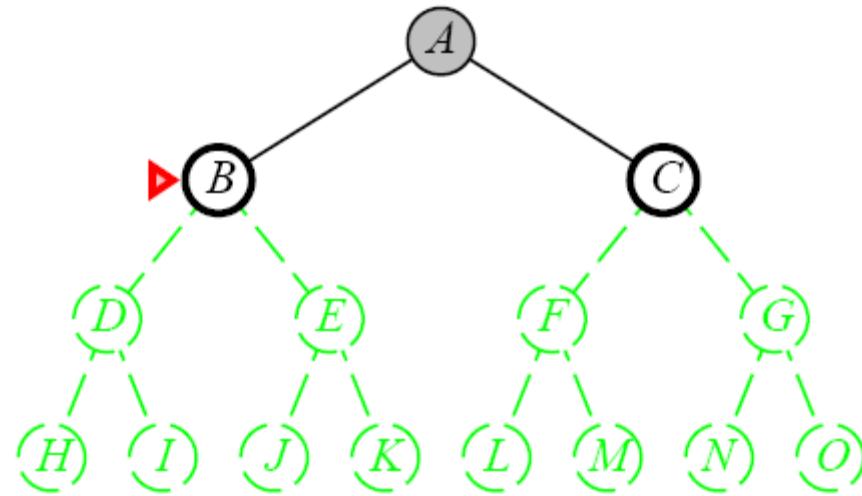
        add generated nodes to the front of *fringe*

End Loop

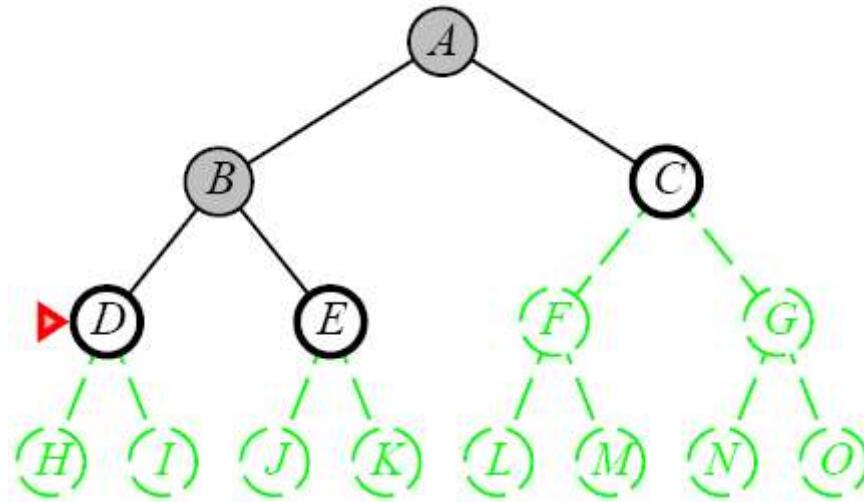
# Depth-First Search



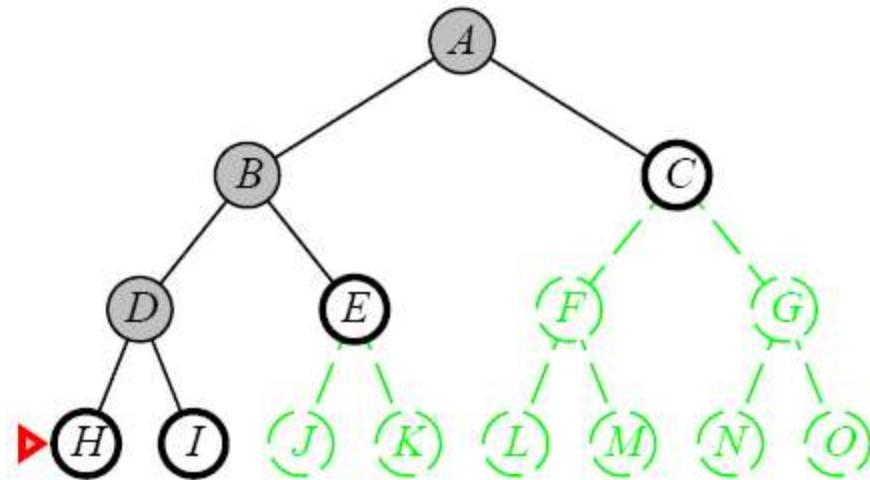
# Depth-First Search



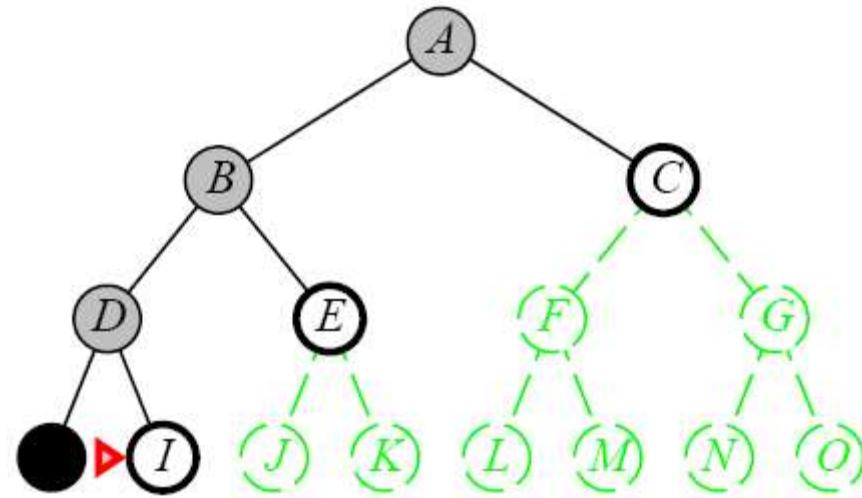
# Depth-First Search



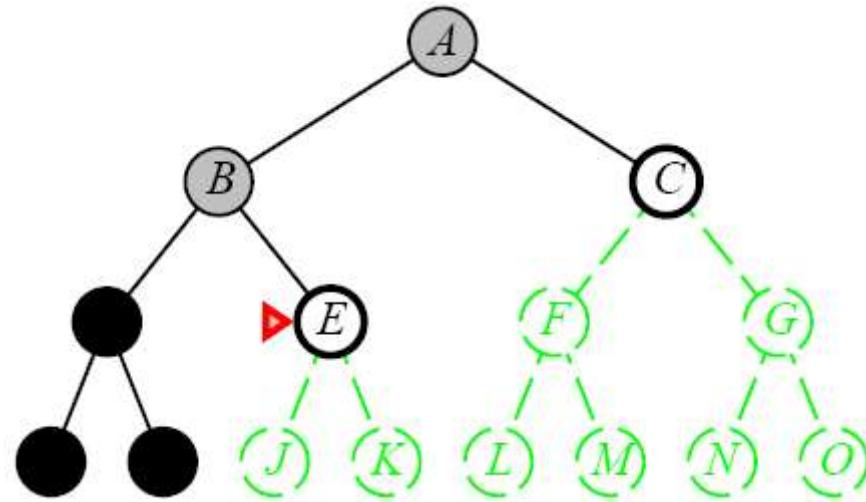
# Depth-First Search



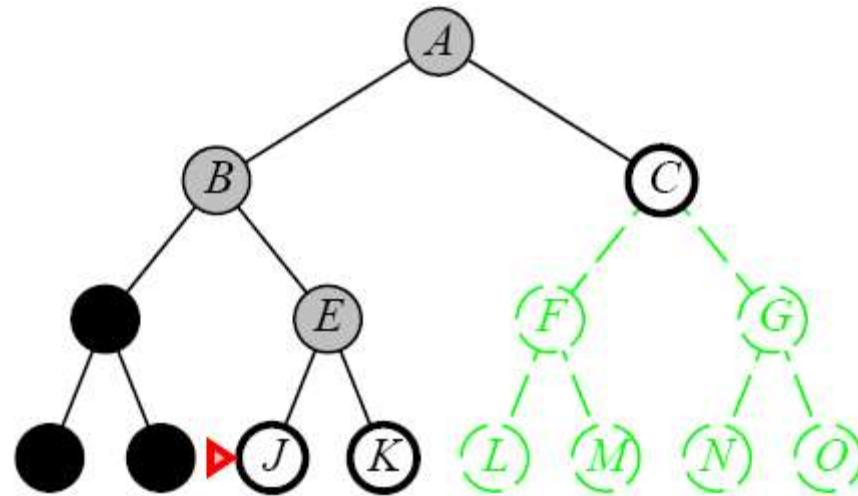
# Depth-First Search



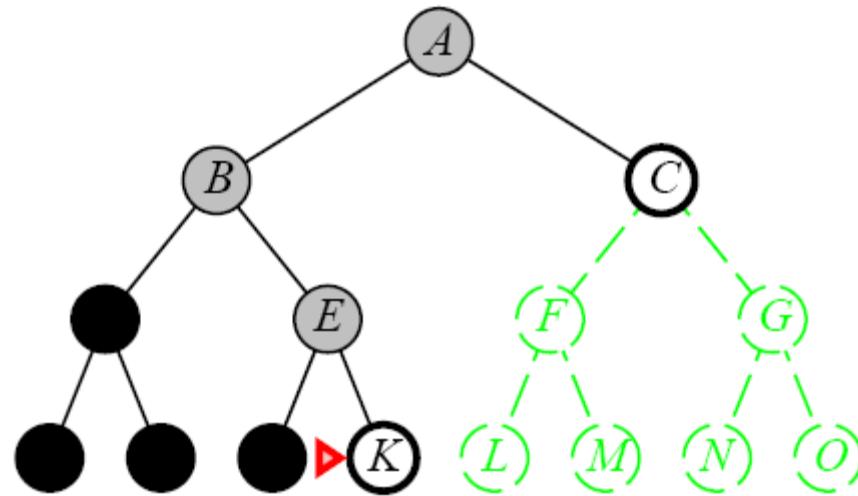
# Depth-First Search



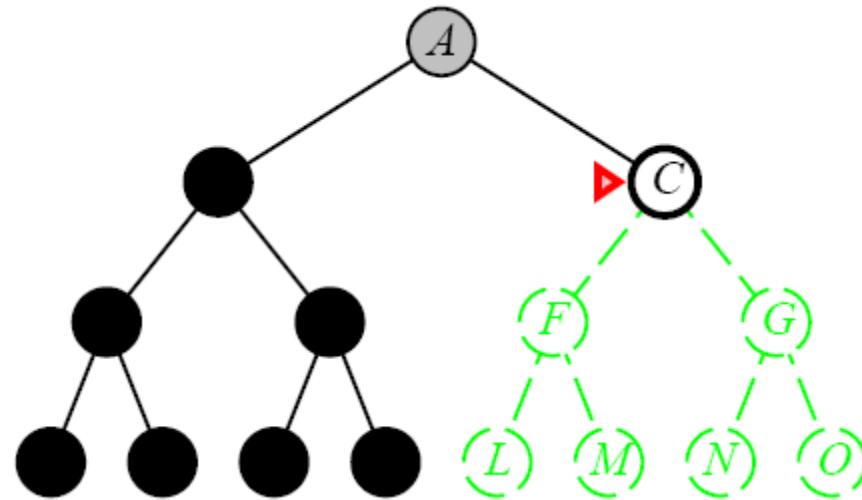
# Depth-First Search



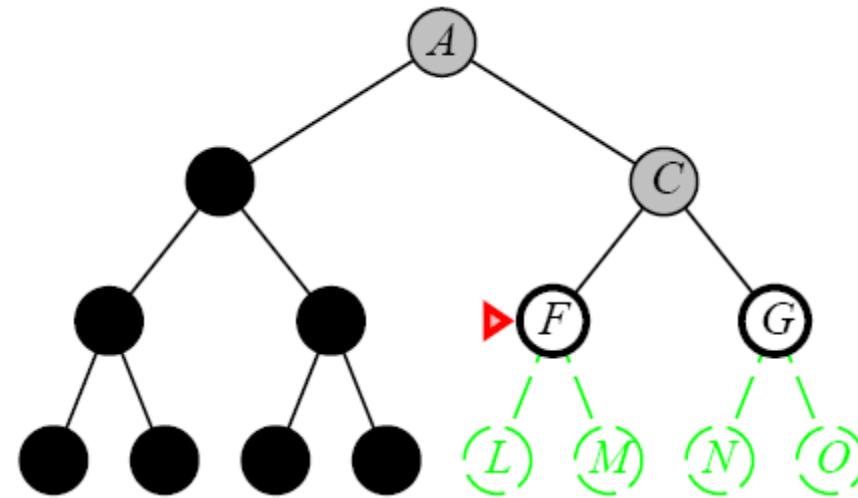
# Depth-First Search



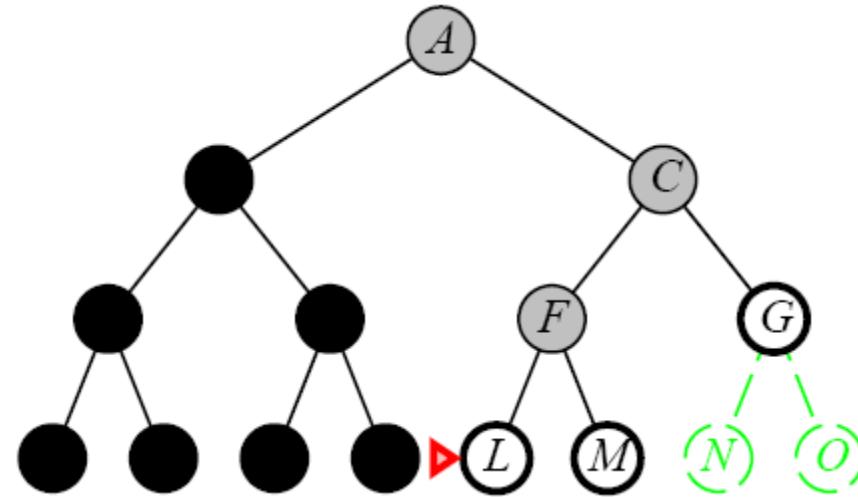
# Depth-First Search



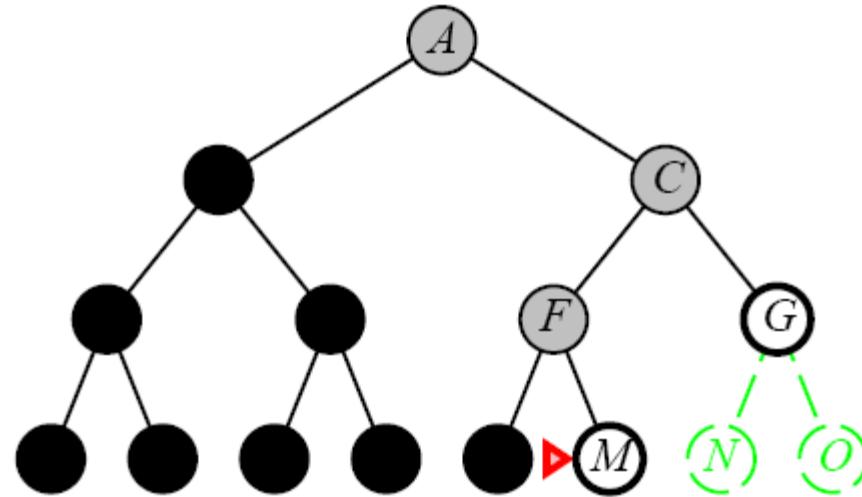
# Depth-First Search



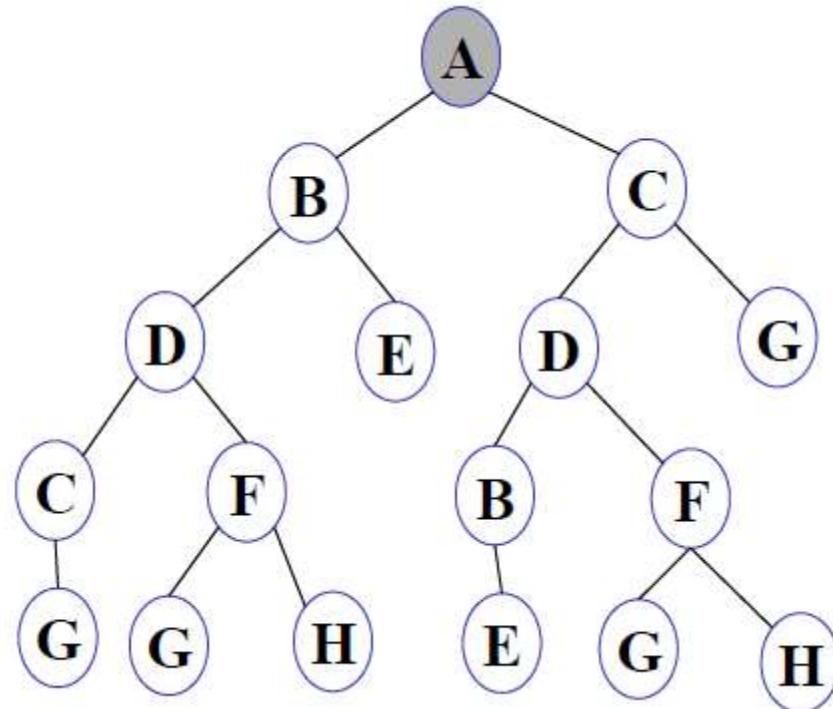
# Depth-First Search



# Depth-First Search



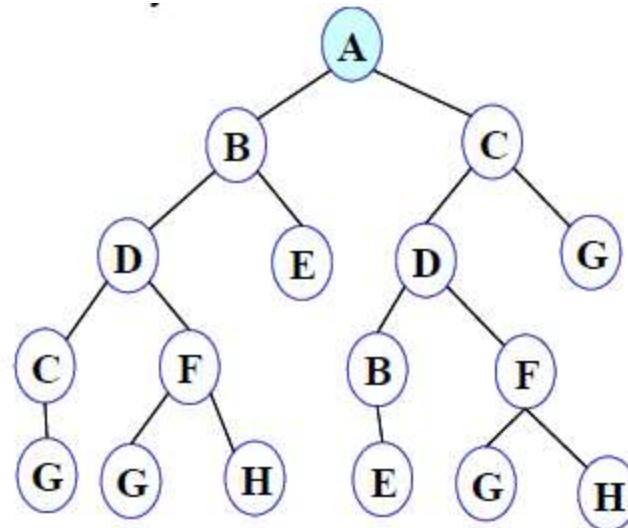
Let us now run Depth First Search on the search space given in Figure 34, and trace its progress.



**Figure 11: Search tree for the state space**

# Depth-First Search

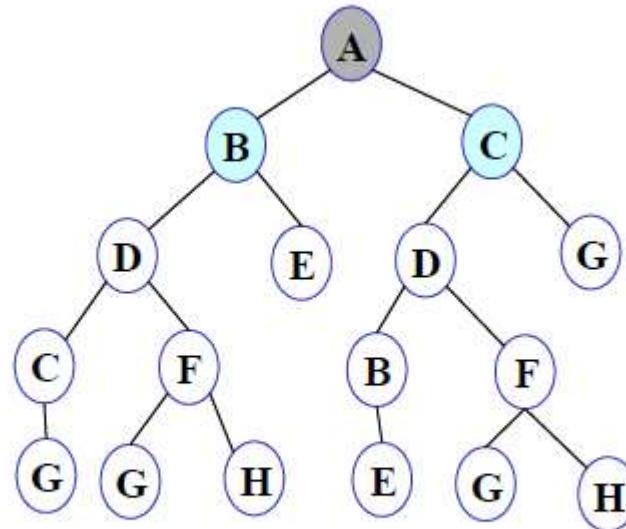
Step 1: Initially fringe contains only the node for A.



FRINGE: A

# Depth-First Search

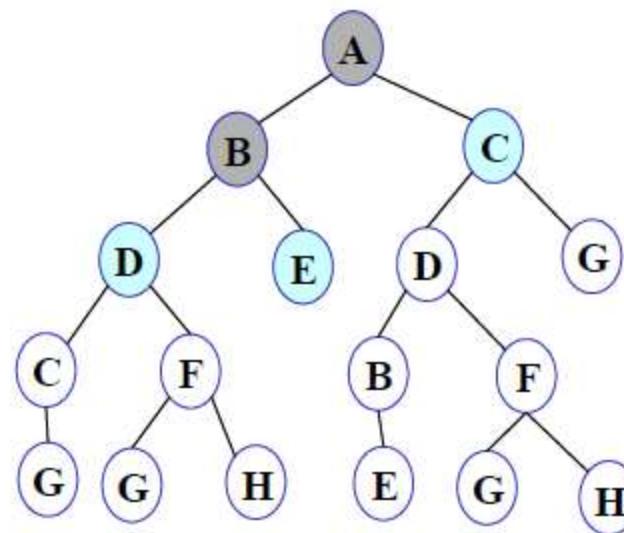
Step 2: A is removed from fringe. A is expanded and its children B and C are put in front of fringe.



FRINGE: B C

# Depth-First Search

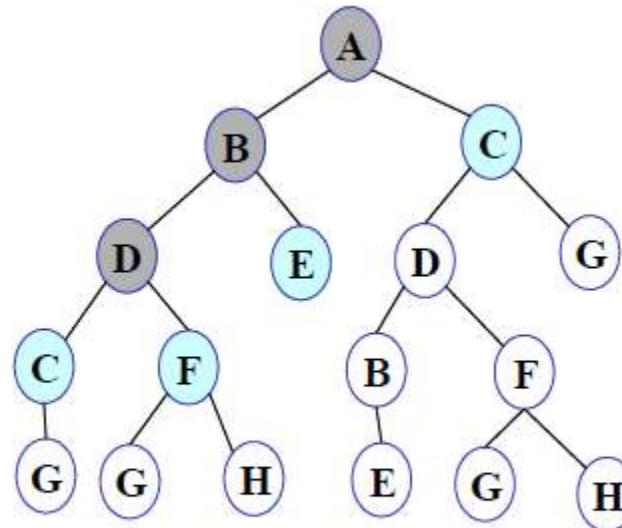
Step 3: Node B is removed from fringe, and its children D and E are pushed in front of fringe.



FRINGE: D E C

# Depth-First Search

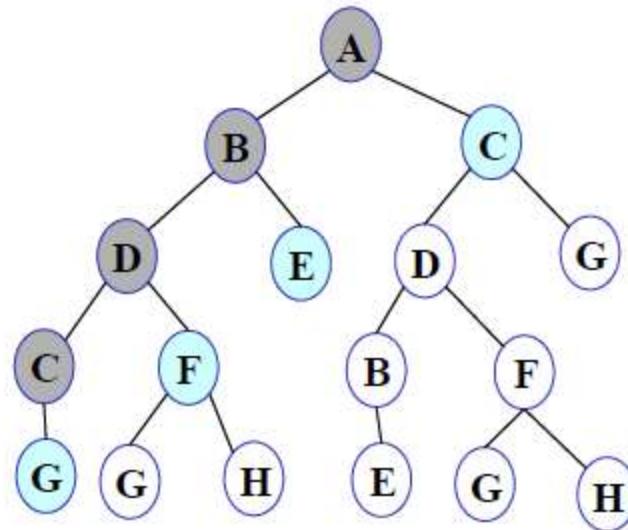
Step 4: Node D is removed from fringe. C and F are pushed in front of fringe.



FRINGE: C F E C

# Depth-First Search

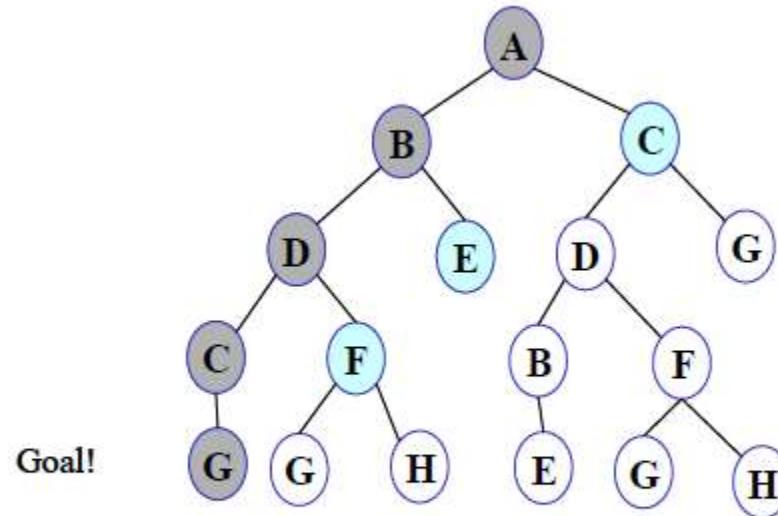
Step 5: Node C is removed from fringe. Its child G is pushed in front of fringe



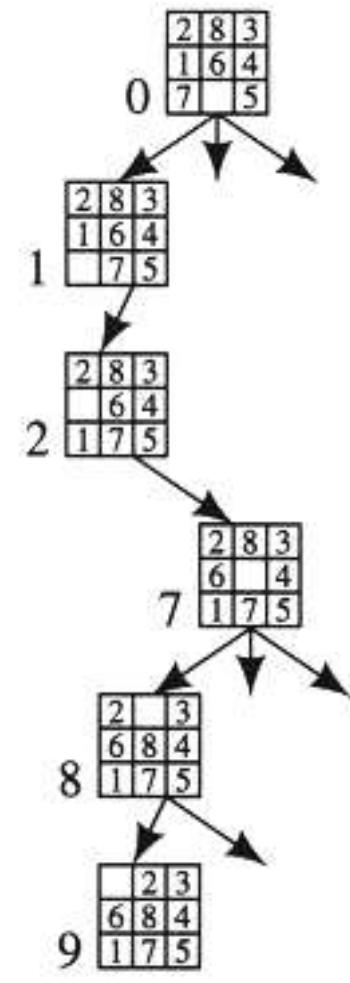
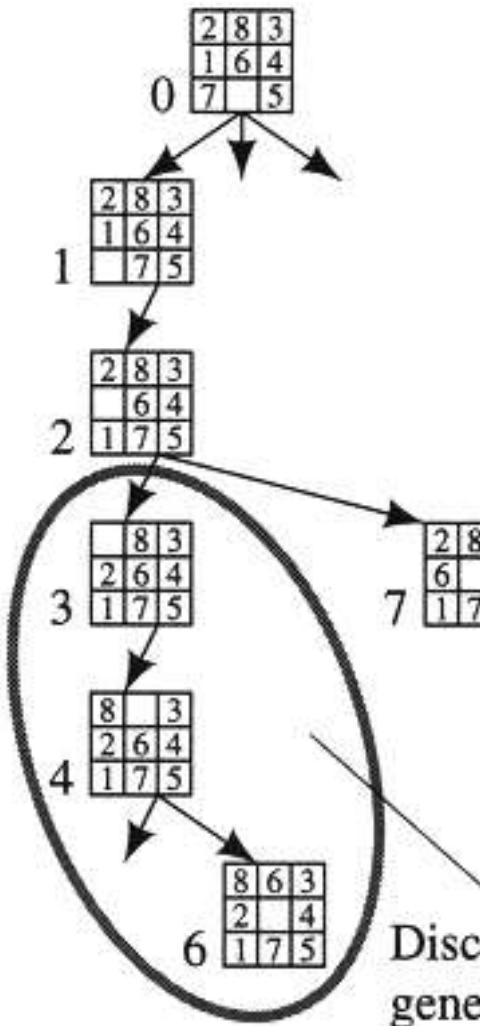
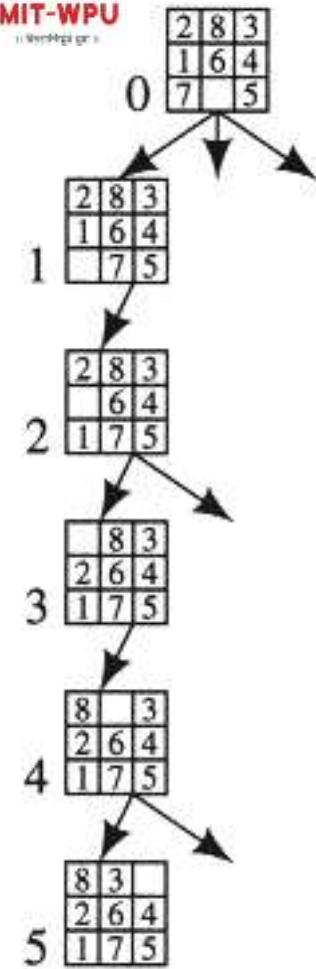
FRINGE: G F E C

# Depth-First Search

Step 6: Node G is expanded and found to be a goal node. The solution path A-B-D-C-G is returned and the algorithm terminates.



## Depth-First Search on 8 Puzzle

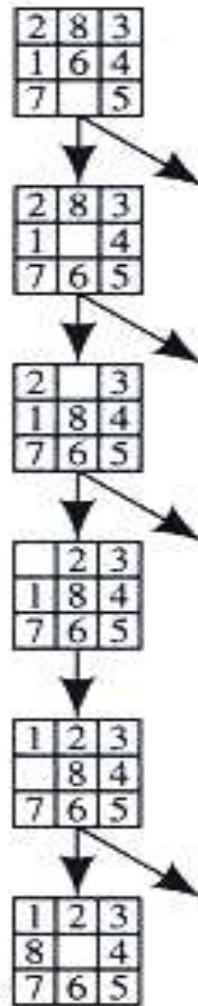


Generation of the First Few Nodes in a Depth-First Search



MIT-WPU

॥ रामानिष्ठुरं हुया ॥



Goal node

The Graph When the Goal Is Reached in Depth-First Search

## Advantages and Disadvantages of DFS

### Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

### Disadvantage:

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

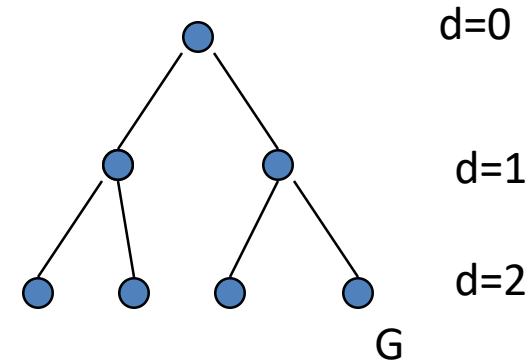
# What is the Complexity of Depth-First Search?

- Time Complexity

- assume (worst case) that there is 1 goal leaf at the RHS
- so DFS will expand all nodes

$$= 1 + b + b^2 + \dots + b^d$$

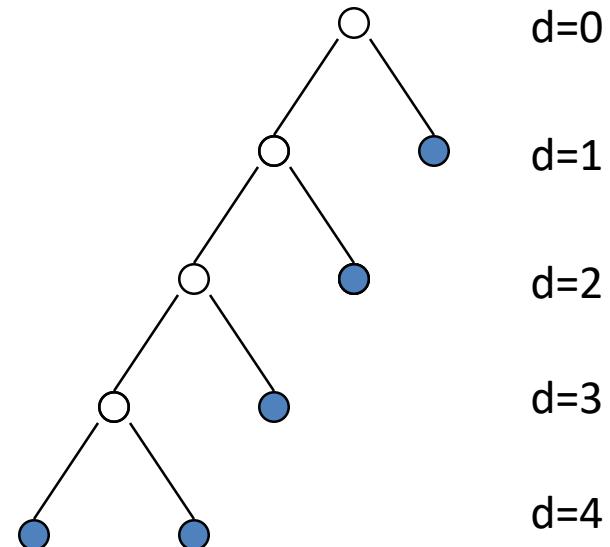
$$= O(b^d) \text{ (Best case)}$$



—

- Space Complexity

- how many nodes can be in the queue (worst-case)?
- at depth  $l < d$  we have  $b-1$  nodes
- at depth  $d$  we have  $b$  nodes
- total =  $(d-1)*(b-1) + b = O(bd)$



# Depth-First Search

- **Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- **Time Complexity:** Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$$T(n) = 1 + n^2 + n^3 + \dots + n^m = O(n^m)$$

Where, m= maximum depth of any node and this can be much larger than d  
(Shallowest solution depth)

- **Space Complexity:** DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.
- **Optimal:** DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

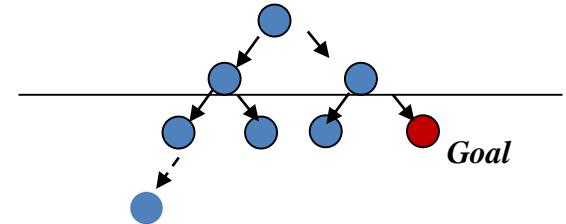
# Advantages and Disadvantages of Depth limited search

- **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 (DLS):

*To avoid infinite search of DFS*

- Stop DFS at a *fixed* depth  $l$ , no matter what
- Goal, may NOT be found: *Incomplete Algorithm*
  - If goal depth  $d > l$
- Why DLS? To avoid getting stuck at infinite (read: large) depth
- Time:  $O(b^l)$ , Memory:  $O(bl)$



# Depth-Limited Search

- A depth-limited search algorithm is similar to depth-first search with a predetermined limit.
- Depth-limited search can solve the drawback of the infinite path in the Depth-first search.
- In this algorithm, the node at the depth limit will treat as it has no successor nodes further.
- 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.

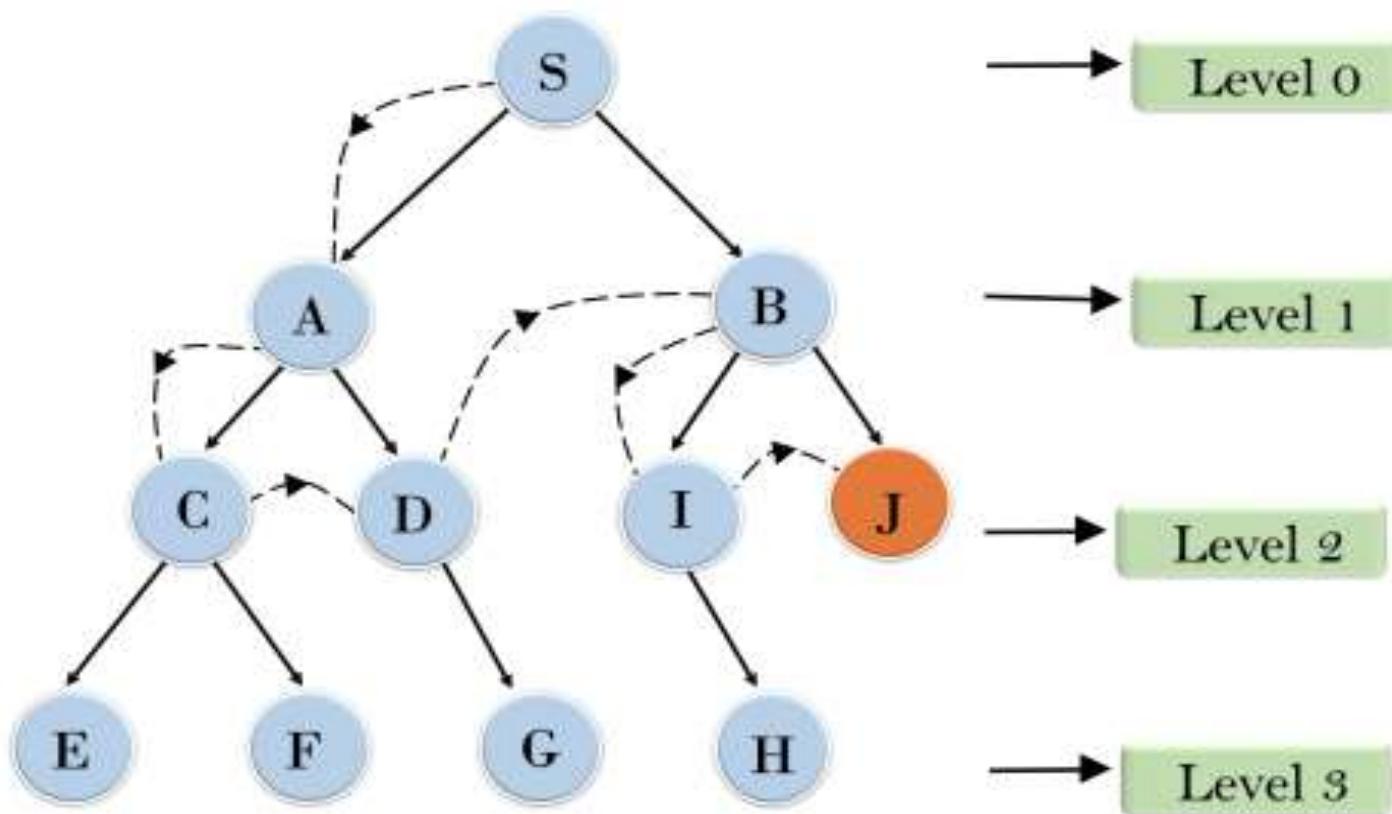
# Depth-Limited Search

Recursive implementation:

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred?  $\leftarrow$  false
    if GOAL-TEST[problem](STATE[node]) then return node
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred?  $\leftarrow$  true
        else if result  $\neq$  failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Example:

## Depth Limited Search



# Depth-Limited Search Properties

- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Time Complexity:** Time complexity of DLS algorithm is  $O(b^\ell)$ .
- **Space Complexity:** Space complexity of DLS algorithm is  $O(b \times \ell)$ .
- **Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if  $\ell > d$ .

# Uninformed : Iterative Deepening Search(IDS)

- The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.
- This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.
- This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.
- The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

# Advantages and Disadvantages of IDS

## Advantages:

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

## Disadvantages:

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

# Iterative Deepening Search

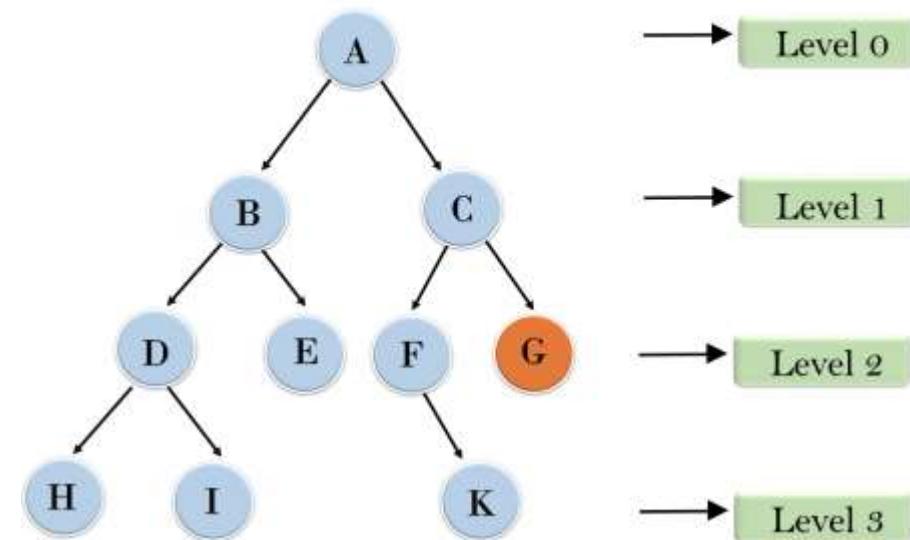
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem

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

## Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

### Iterative deepening depth first search



1<sup>st</sup> Iteration----> A

2<sup>nd</sup> Iteration----> A, B, C

3<sup>rd</sup> Iteration----> A, B, D, E, C, F, G

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

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

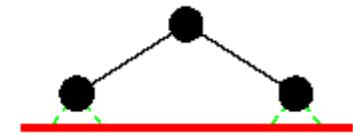
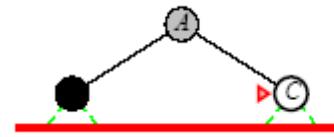
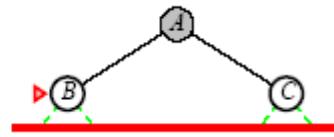
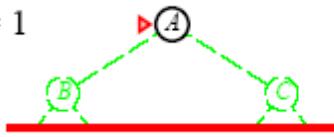
# Iterative Deepening Search

Limit = 0



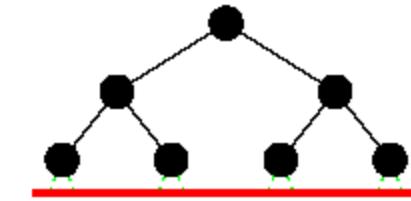
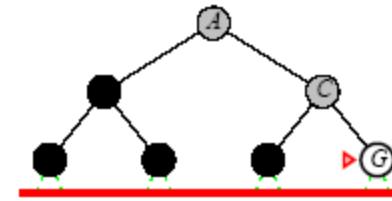
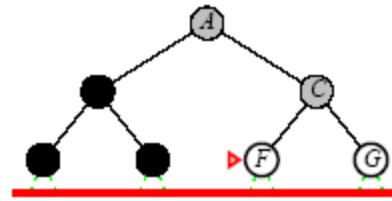
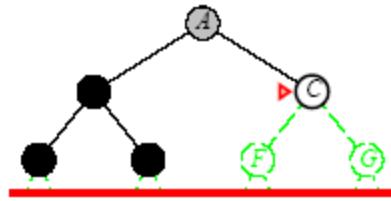
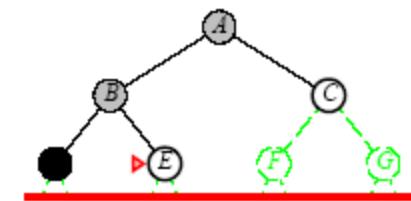
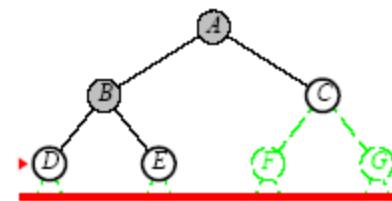
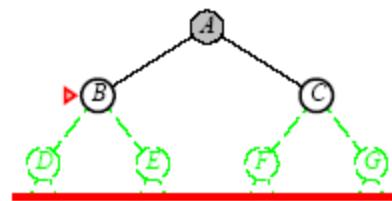
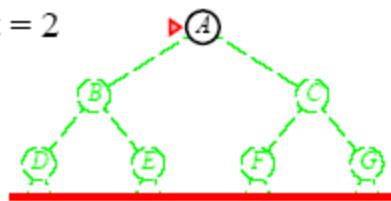
# Iterative Deepening Search

Limit = 1



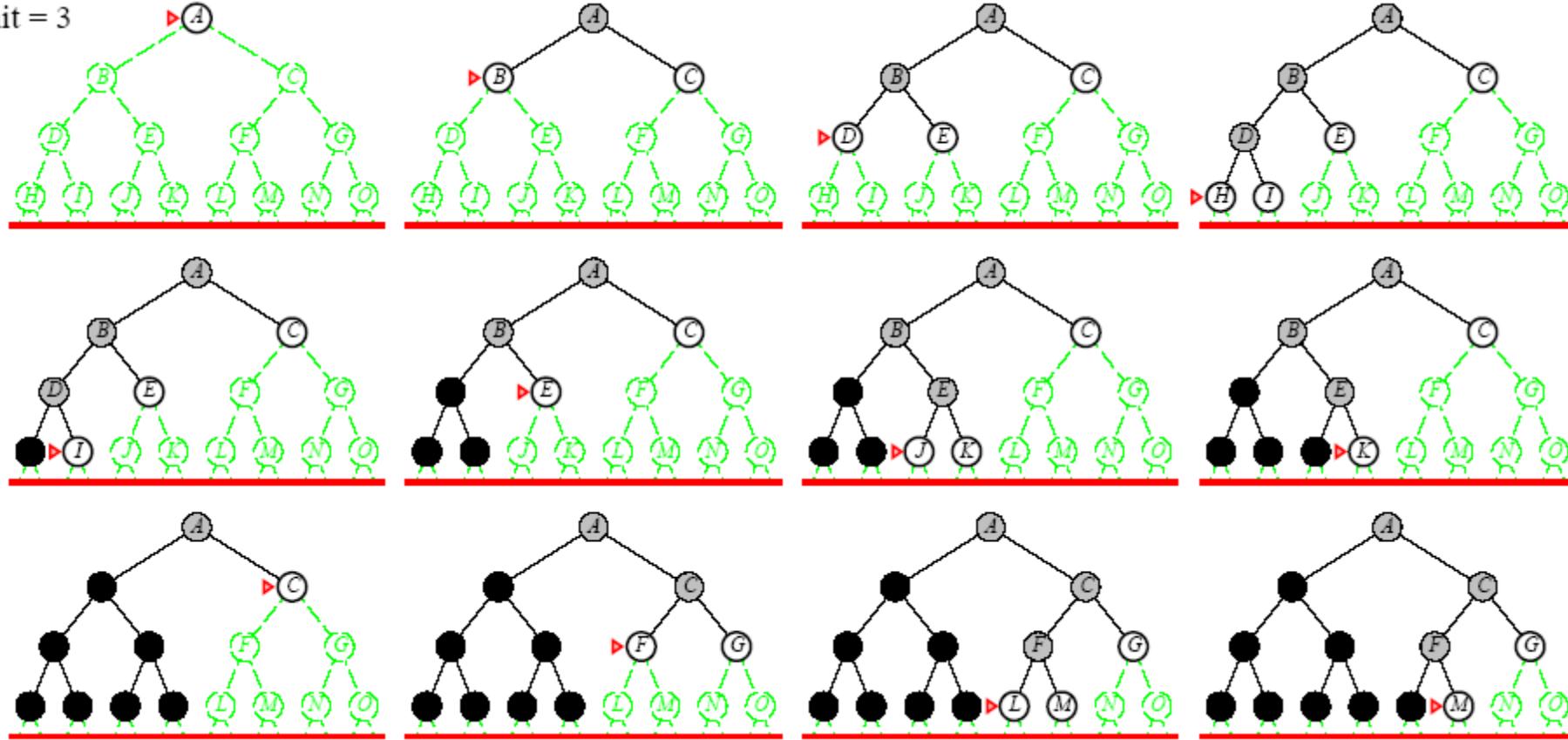
# Iterative Deepening Search

Limit = 2



# Iterative Deepening Search

Limit = 3



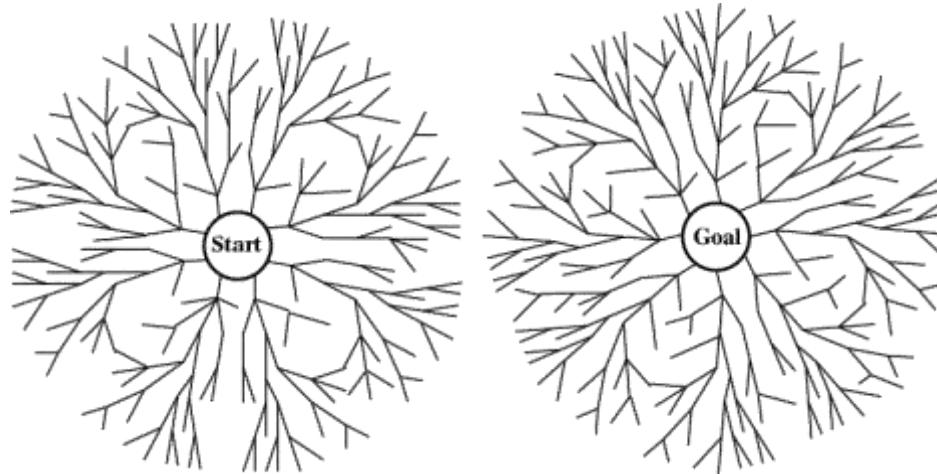
# Iterative Deepening Search

- **Completeness:**
- This algorithm is complete if the branching factor is finite.
- **Time Complexity:**
- Let's suppose  $b$  is the branching factor and depth is  $d$  then the worst-case time complexity is  $\mathbf{O}(b^d)$ .
- **Space Complexity:**
- The space complexity of IDDFS will be  $\mathbf{O}(bd)$ .
- **Optimal:**
- IDDFS algorithm is optimal if path cost is a non-decreasing function of the depth of the node.

# Lessons From Iterative Deepening Search

- Faster than BFS even though IDS generates repeated states
  - BFS generates nodes up to level  $d+1$
  - IDS only generates nodes up to level  $d$
- In general, iterative deepening search is the preferred uninformed search method when there is a large search space and the depth of the solution is not known

# Bi-directional search



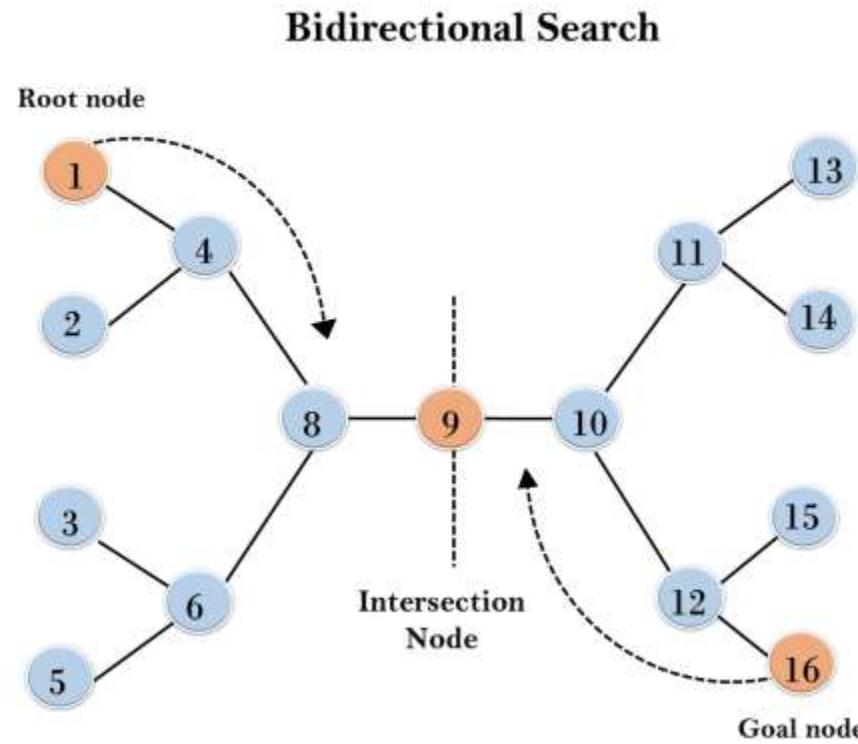
- Bidirectional search algorithm runs two simultaneous searches, one from initial state called as forward-search and other from goal node called as backward-search, to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex.
- The search stops when these two graphs intersect each other.
- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

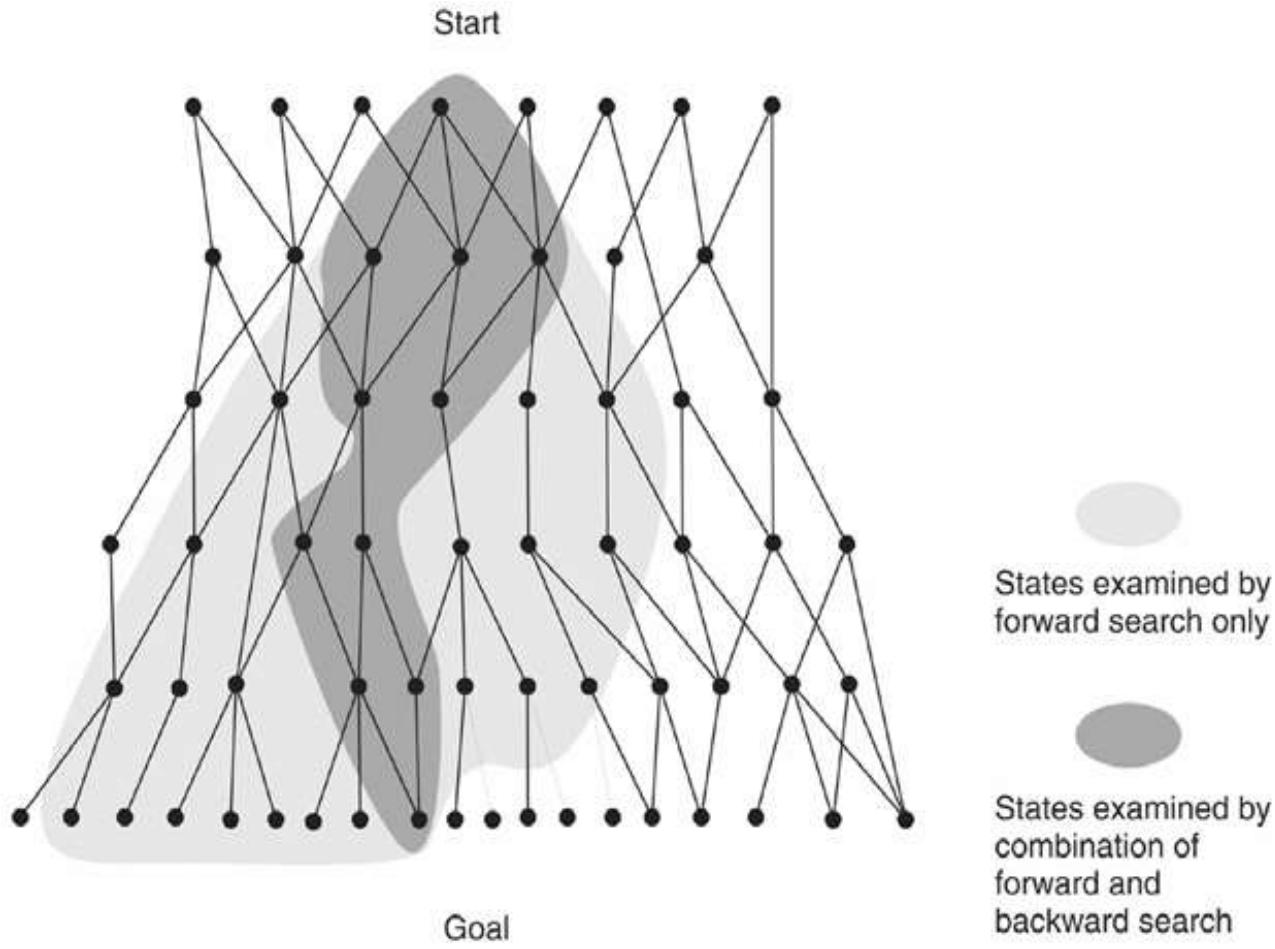
# Bi-directional search

- Works well only when there are unique start and goal states.
- Requires the ability to generate “predecessor” states.
- Can (sometimes) lead to finding a solution more quickly.
- Time complexity:  $O(b^{d/2})$ . Space complexity:  $O(b^{d/2})$ .
- **Advantages:**
  - Bidirectional search is fast.
  - Bidirectional search requires less memory
- **Disadvantages:**
  - Implementation of the bidirectional search tree is difficult.
  - **In bidirectional search, one should know the goal state in advance.**

# Example: BDS

- In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.
- The algorithm terminates at node 9 where two searches meet.





Also note that the algorithm works well only when there are unique start and goal states.

# Time and Space Complexities

- **Completeness:** Bidirectional Search is complete if we use BFS in both searches.
- **Time Complexity:** Time complexity of bidirectional search using BFS is  $O(b^d)$ .
- **Space Complexity:** Space complexity of bidirectional search is  $O(b^d)$ .
- **Optimal:** Bidirectional search is Optimal.

# Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

There are a number of footnotes, caveats, and assumptions.

See Fig. 3.21, p. 91.

[a] complete if  $b$  is finite

[b] complete if step costs  $\geq \varepsilon > 0$

[c] optimal if step costs are all identical

(also if path cost non-decreasing function of depth only)

[d] if both directions use breadth-first search

(also if both directions use uniform-cost search with step costs  $\geq \varepsilon > 0$ )



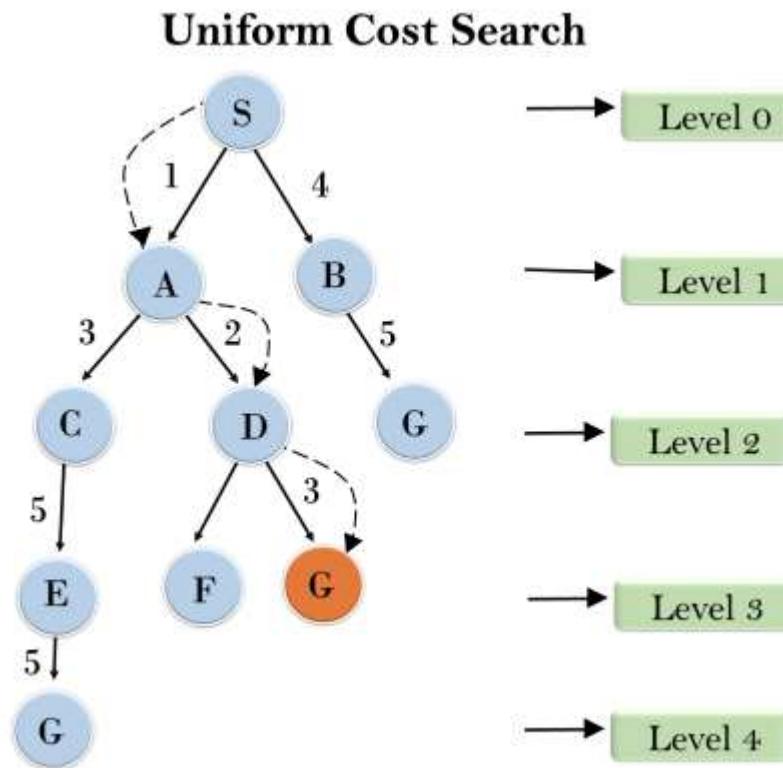
Generally the preferred uninformed search strategy

## 5.Uniform-cost search

- Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph.
- This algorithm comes into play when a different cost is available for each edge.
- The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost.
- Uniform-cost search expands nodes according to their path costs from the root node.
- It can be used to solve any graph/tree where the optimal cost is in demand.
- A uniform-cost search algorithm is implemented by the priority queue.
- It gives maximum priority to the lowest cumulative cost.
- Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.
- This ensures that when a node is selected for expansion it is a node with the cheapest cost among the nodes in **OPEN**, “priority queue”
- **Let  $g(n)$  = cost of the path from the start node to the current node n. Sort nodes by increasing value of g.**

# Uniform-cost search

- **Advantages:**
- Uniform cost search is optimal because at every state the path with the least cost is chosen.
- **Disadvantages:**
- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop



# Uniform Cost Search in Tree

1.  $\text{fringe} \leftarrow \text{MAKE-EMPTY-QUEUE}()$
2.  $\text{fringe} \leftarrow \text{INSERT( root\_node ) // with } g=0$
3. loop {
  1. if fringe is empty then return false // *finished without goal*
  2. node  $\leftarrow \text{REMOVE-SMALLEST-COST}(\text{fringe})$
  3. if node is a goal
    1. print node and g
    2. return true // *that found a goal*
  4.  $L_g \leftarrow \text{EXPAND(node)} // L_g \text{ is set of children with their } g \text{ costs}$   
*// NOTE: do not check  $L_g$  for goals here!!*
  5.  $\text{fringe} \leftarrow \text{INSERT-ALL}(L_g, \text{fringe} )$
- }

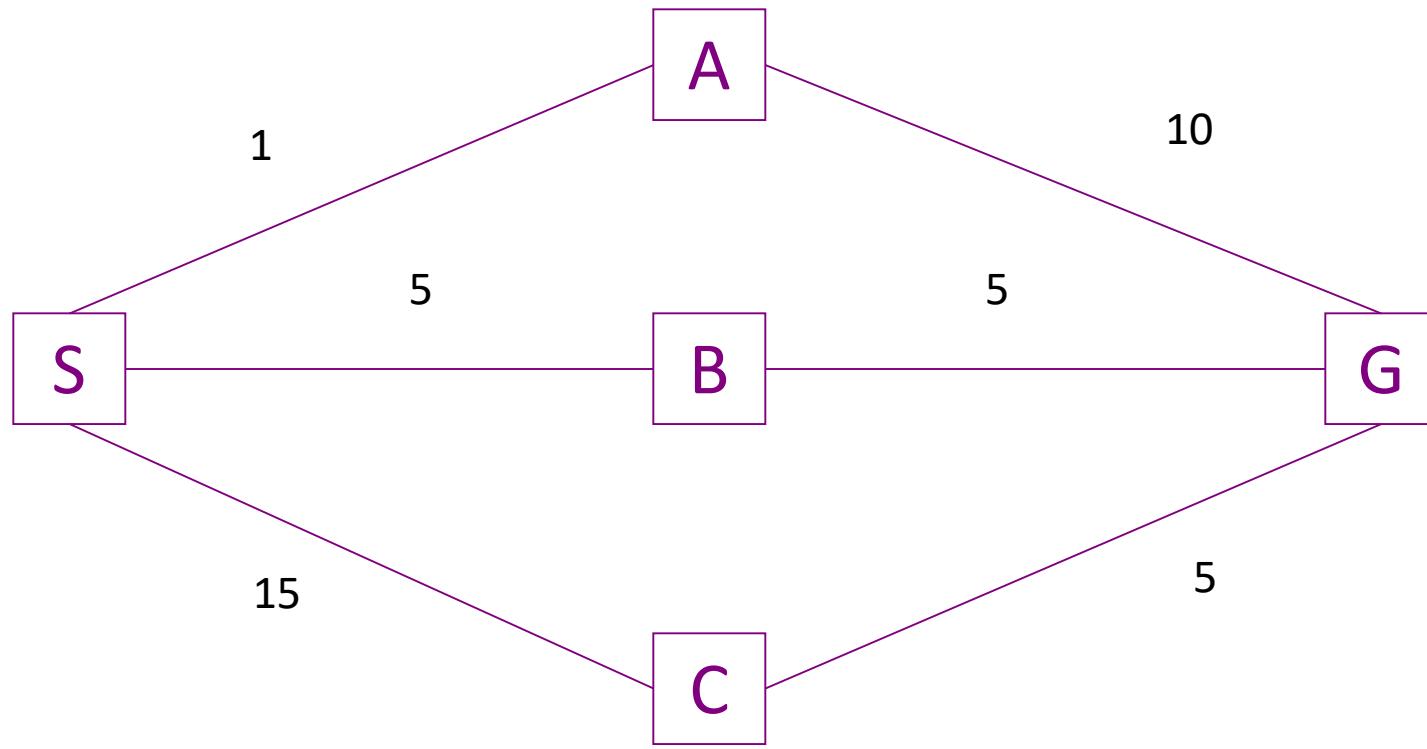
# Uniform Cost Search in Graph

```
1. fringe ⊂ MAKE-EMPTY-QUEUE()
2. fringe ⊂ INSERT( root_node ) // with g=0
3. loop {
    1. if fringe is empty then return false // finished without goal
    2. node ⊂ REMOVE-SMALLEST-COST(fringe)
    3. if node is a goal
        1. print node and g
        2. return true // that found a goal
    4. Lg ⊂ EXPAND(node) // Lg is set of neighbours with their g costs
                           // NOTE: do not check Lg for goals here!!
    5. fringe ⊂ INSERT-IF-NEW(Lg, fringe ) // ignore revisited nodes
                                               // unless is with new better g
}
```

# Uniform cost search Properties

- **Completeness:**
- Uniform-cost search is complete, such as if there is a solution, UCS will find it.
- **Time Complexity:**
- Let  $C^*$  is **Cost of the optimal solution**, and  $\epsilon$  is each step to get closer to the goal node. Then the number of steps is  $= C^*/\epsilon+1$ . Here we have taken  $+1$ , as we start from state 0 and end to  $C^*/\epsilon$ .
- Hence, the worst-case time complexity of Uniform-cost search is  $O(b^{1 + \lceil C^*/\epsilon \rceil})$ .
- **Space Complexity:**
- The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is  $O(b^{1 + \lceil C^*/\epsilon \rceil})$ .
- **Optimal:**
- Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

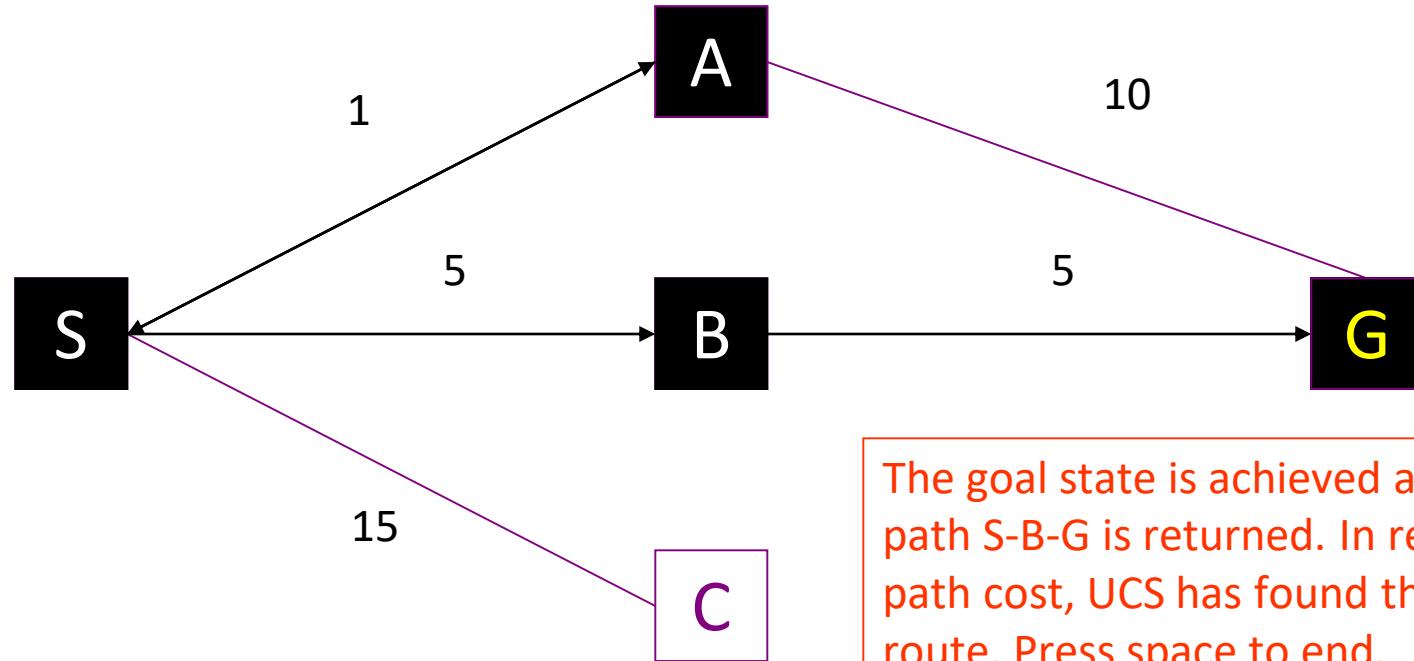
der the following problem...



We wish to find the shortest route from node S to node G; that is, node S is the initial state and node G is the goal state. In terms of path cost, we can clearly see that the route *SBG* is the cheapest route. However, if we let breadth-first search chose on the problem it will find the non-optimal path *SAG*, assuming that A is the first node to be expanded at level 1. Press space to see a UCS of the same node set...

**UNIFORM COST SEARCH PATTERN** is a search pattern that finds the shortest path from start node (S) to goal node (G). The queue (Q) is implemented using priority queue which has the following properties. It contains all nodes in the graph, sorted by their current total cost. Nodes with lower cost have higher priority than nodes with higher cost. If two nodes have the same cost, they are added to the queue in the order they were generated. In case that there are two paths to the goal node G (one with cost 5, one with cost 10), the algorithm prefers the one with the lower cost.

Press space to goal state. Press space.



Press space to begin the search

Size of Queue: 0

Queue: Empty

Nodes expanded: 3

FINISHED SEARCH

Current level: 2

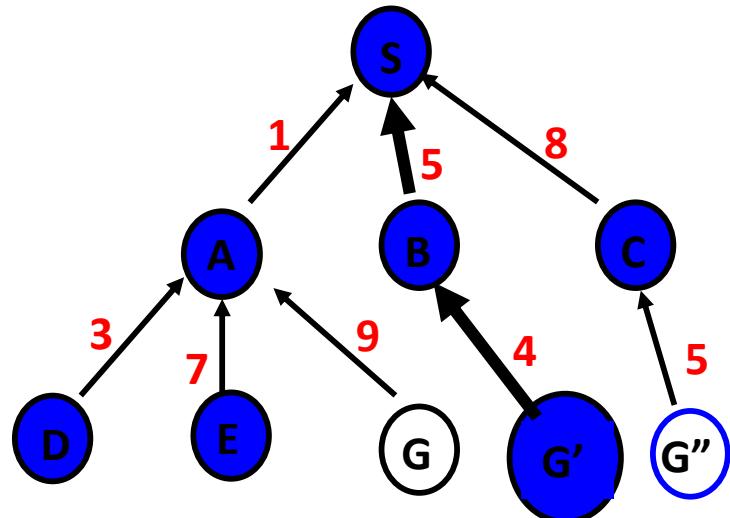
UNIFORM COST SEARCH PATTERN

# Uniform-Cost Search

GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)

exp. node nodes list

CLOSED list



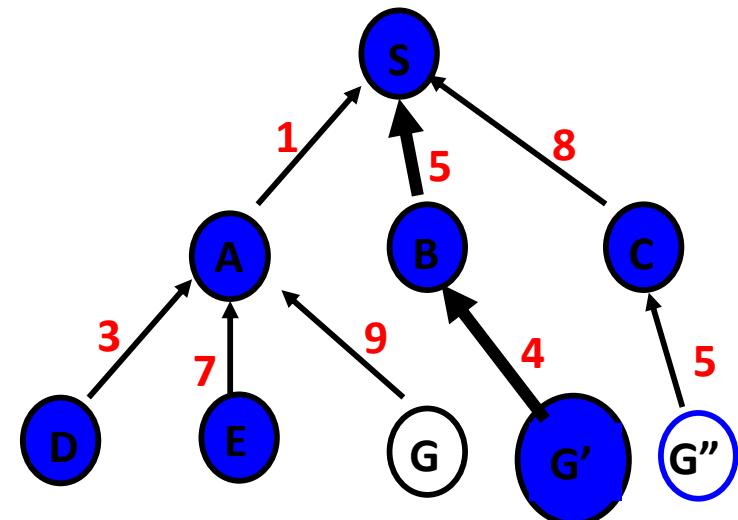
# Uniform-Cost Search

GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)

**exp. node nodes list**

	{S(0)}
S	{A(1) B(5) C(8)}
A	{D(4) B(5) C(8) E(8) G(10)}
D	{B(5) C(8) E(8) G(10)}
B	{C(8) E(8) G'(9) G(10)}
C	{E(8) G'(9) G(10) G''(13)}
E	{G'(9) G(10) G''(13)}
G'	{G(10) G''(13)}

**CLOSED list**



Solution path found is S B G  $\leftarrow$  this G has cost 9, not 10

Number of nodes expanded (including goal node) = 7

# Uniform-Cost (UCS)

- **It is complete** (if cost of each action is not infinitesimal)
  - The total # of nodes  $n$  with  $g(n) \leq g(\text{goal})$  in the state space is **finite**
- **Optimal/Admissible**
  - It is admissible if the goal test is done **when a node is removed from the OPEN list** (delayed goal testing), not when its parent node is expanded and the node is first generated
- **Exponential time and space complexity**,  $O(b^d)$  where  $d$  is the depth of the solution path of the least cost solution

# Comparing Search Strategies

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

And how on our  
small example? 

### **-First Search:**

Expanded nodes: S A D E G

- Solution found: S A G (cost 10)

- Breadth-First Search:**

- Expanded nodes: S A B C D E G
- Solution found: S A G (cost 10)

- Uniform-Cost Search:**

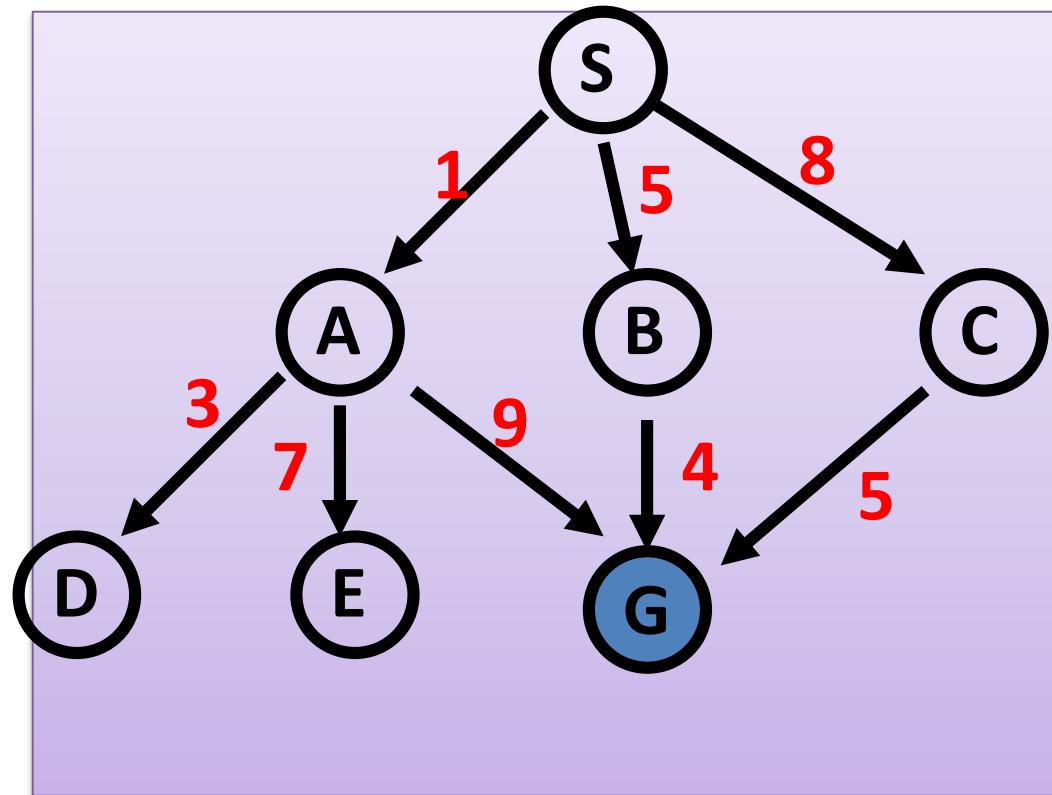
- Expanded nodes: S A D B C E G
- Solution found: S B G (cost 9)

*This is the only uninformed search that worries about costs.*

- Depth First Iterative-Deepening Search:**

- nodes expanded: S S A B C S A D E G
- Solution found: S A G (cost 10)

# How they perform



**Depth-First Search:**

**Breadth-First Search:**

**Uniform-Cost Search:**

**Iterative-Deepening Search:**

# When to use what?

- **Depth-First Search:**

- Many solutions exist
- Know (or have a good estimate of) the depth of solution

- **Breadth-First Search:**

- Some solutions are known to be shallow

- **Uniform-Cost Search:**

- Actions have varying costs
- Least cost solution is the required

*This is the only uninformed search that worries about costs.*

- **Iterative-Deepening Search:**

- Space is limited and the shortest solution path is required

# Search Graphs

- If the search space is not a tree, but a graph, the search tree may contain different nodes corresponding to the same state.
- The way to avoid generating the same state again when not required
- The search algorithm can be modified to check a node when it is being generated.
- we use another list called CLOSED,
- which records all the expanded nodes.
- The newly generated node is checked with the nodes in CLOSED list & open list,

# Algorithm outline-

Graph search algorithm

Let *fringe* be a list containing the initial state

Let *closed* be initially empty

Loop

    if *fringe* is empty return *failure*

    Node  $\leftarrow$  remove-first (*fringe*)

    if Node is a *goal*

        then return the path from initial state to Node

    else put Node in *closed*

        generate all successors of Node S

        for all nodes m in S

            if m is not in *fringe* or *closed*

                merge m into *fringe*

End Loop

- The CLOSED list has to be maintained,
- the algorithm is required to check every generated node to see if it is already there in OPEN or CLOSED. this will require efficient way to index every node.
- $S \triangleq$  Set of successor
- $M \triangleq$  node going to be generated

# End of Part-3

## (Blind Search Algorithms)

# Informed Search

- So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space.
- But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc.
- This knowledge help agents to explore less to the search space and find more efficiently the goal node.
- The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.
- At the heart of such algorithms there is the concept of a **heuristic function**.

# Heuristics

- **Heuristic** “Heuristics are criteria, methods or principles for deciding which among several alternative actions, promises to be the most effective in order to achieve some goal”.

*quote by Judea Pearl*

- In heuristic search or informed search, heuristics are used to identify the most promising search path.
- **Admissibility of the heuristic function is given as:**
- $h(n) \leq h^*(n)$
- Here  $h(n)$  is heuristic cost, and  $h^*(n)$  is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

# Example of Heuristic Function

- A heuristic at a node  $n$  is an estimate of the optimum cost from the current node to a goal. It is denoted by  $h(n)$ .
- $h(n) = \text{estimated cost of the cheapest path from node } n \text{ to a goal node}$
- **Example 1: We want a path from Kolkata to Guwahati**
- Heuristic for Guwahati may be straight-line distance between Kolkata and Guwahati
- $h(\text{Kolkata}) = \text{euclideanDistance}(\text{Kolkata}, \text{Guwahati})$

## 2: 8-puzzle: Misplaced Tiles Heuristics is the number of tiles out of place.

Example 2: 8-puzzle: Misplaced Tiles Heuristics is the number of tiles out of place.

2	8	3
1	6	4
	7	5

Initial State

1	2	3
8		4
7	6	5

Goal state

Figure 1: 8 puzzle

**1. Hamming distance** The first picture shows the current state  $n$ , and the second picture the goal state.

$h(n) = 5$  (because the tiles 2, 8, 1, 6 and 7 are out of place. )

### 2. Manhattan Distance Heuristic:

This heuristic sums the distance that the tiles are out of place.

The distance of a tile is measured by the sum of the differences in the x-positions and the y-positions.

For the above example, using the Manhattan distance heuristic,

$$h(n) = 1 + 1 + 0 + 0 + 0 + 1 + 1 + 2 = 6$$

# Heuristic Search Algorithm Best-First Search.

## Best First Search

Let  $\text{fringe}$  be a priority queue containing the initial state

Loop

    if  $\text{fringe}$  is empty return failure

    Node  $\leftarrow$  remove-first ( $\text{fringe}$ )

        if Node is a goal

            then return the path from initial state to Node

        else generate all successors of Node, and

            put the newly generated nodes into  $\text{fringe}$

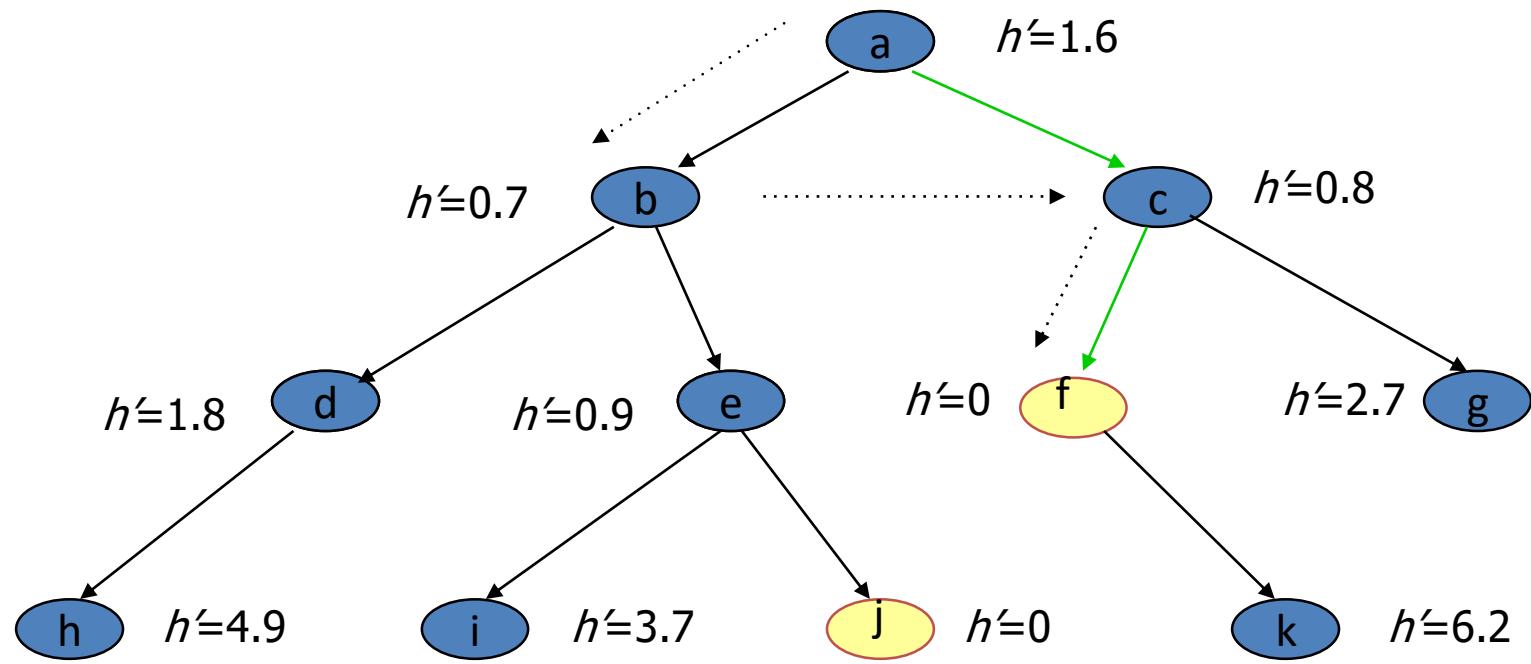
            according to their  $f$  values

End Loop

- The algorithm maintains a priority queue of nodes to be explored
- A cost function  $f(n)$  is applied to each node.
- The nodes are put in OPEN in the order of their  $f$  values.
- Nodes with smaller  $f(n)$  values are expanded earlier.

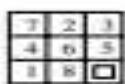
# Example of best first search

- Nodes are visited in the order : a, b, c, f
- Solution path is : a, c, f

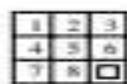


We will now consider different ways of defining the function  $f$ .  
 This leads to different search algorithms

# Applying Best first search to solved 8 puzzle problem

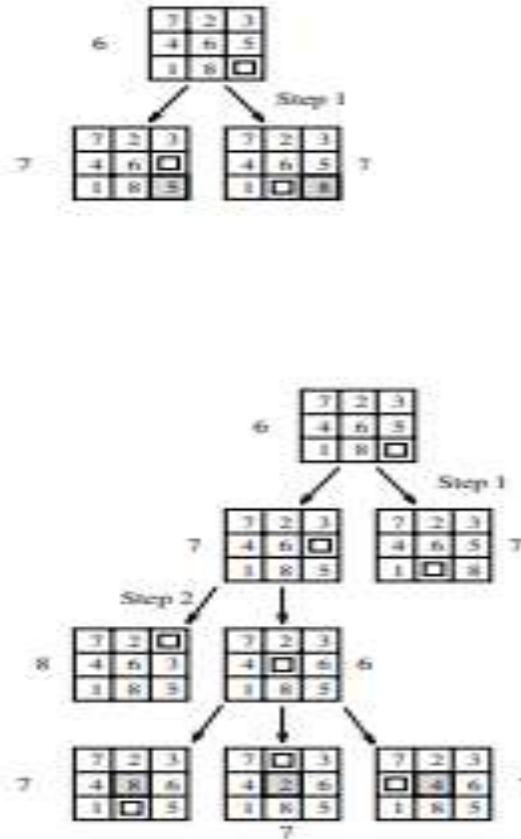


(a)



(b)

Blank Tile  
The last tile moved.



(c)

Figure 11.6 Applying best-first search to the 8-puzzle: (a) initial configuration; (b) final configuration; and (c) states resulting from the first four steps of best-first search. Each state is labeled with its  $f$ -value (that is, the Manhattan distance from the state to the final state).

# Greedy Search

- Greedy best-first search algorithm always selects the path which appears best at that moment.
- It is the combination of depth-first search and breadth-first search algorithms.
- It uses the heuristic function and search. In this search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function,
- i.e.  $f(n) = g(n)$ .
- Were,  $h(n)$ = estimated cost from node n to the goal.
- The greedy best first algorithm is implemented by the priority queue.

# Greedy Best first search algorithm

- Step 1: Place the starting node into the OPEN list.
- Step 2: If the OPEN list is empty, Stop and return failure.
- Step 3: Remove the node  $n$ , from the OPEN list which has the lowest value of  $h(n)$ , and places it in the CLOSED list.
- Step 4: Expand the node  $n$ , and generate the successors of node  $n$ .
- Step 5: Check each successor of node  $n$ , and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
- Step 6: For each successor node, algorithm checks for evaluation function  $f(n)$ , and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list,

# Greedy Search

Advantages:

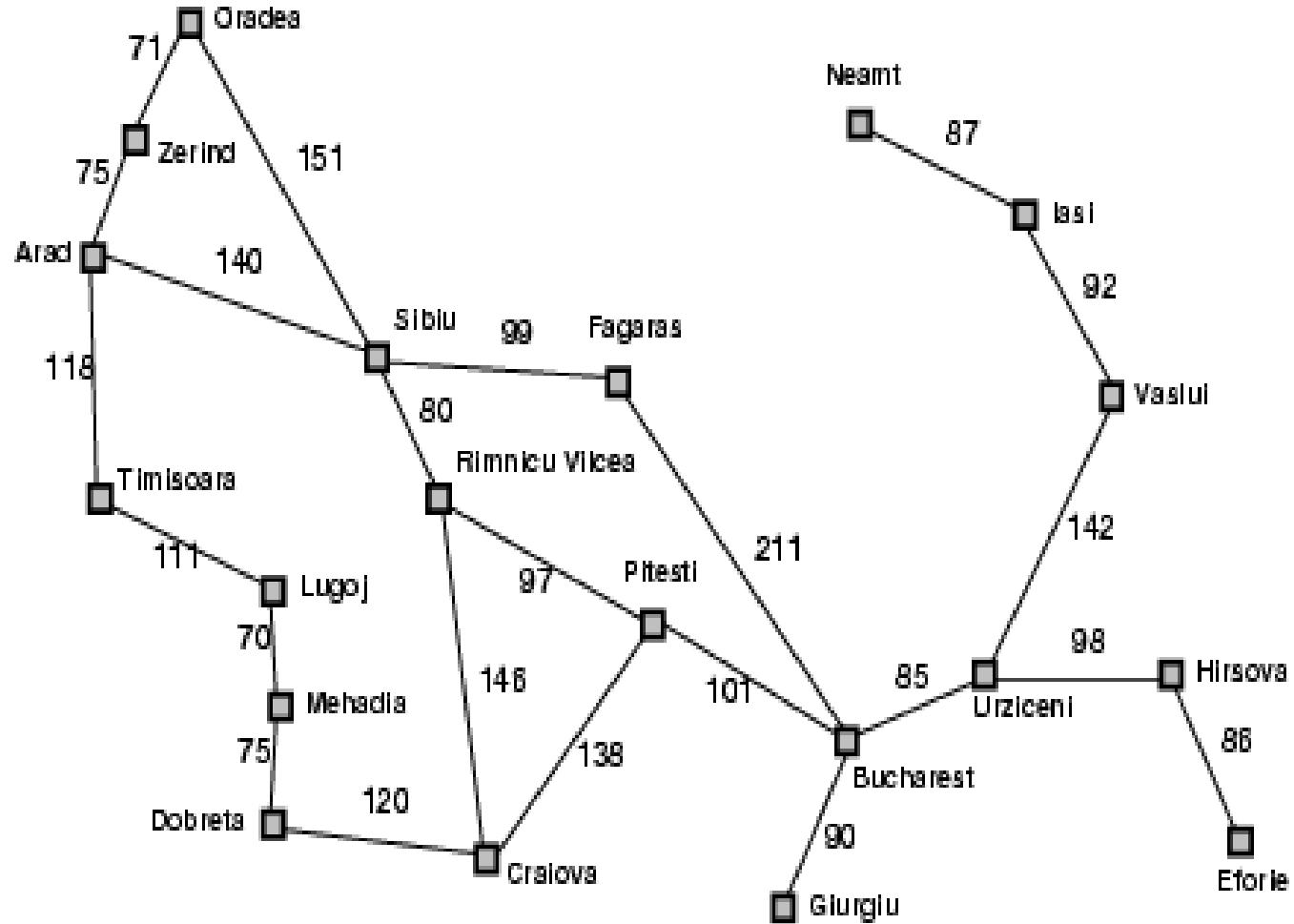
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

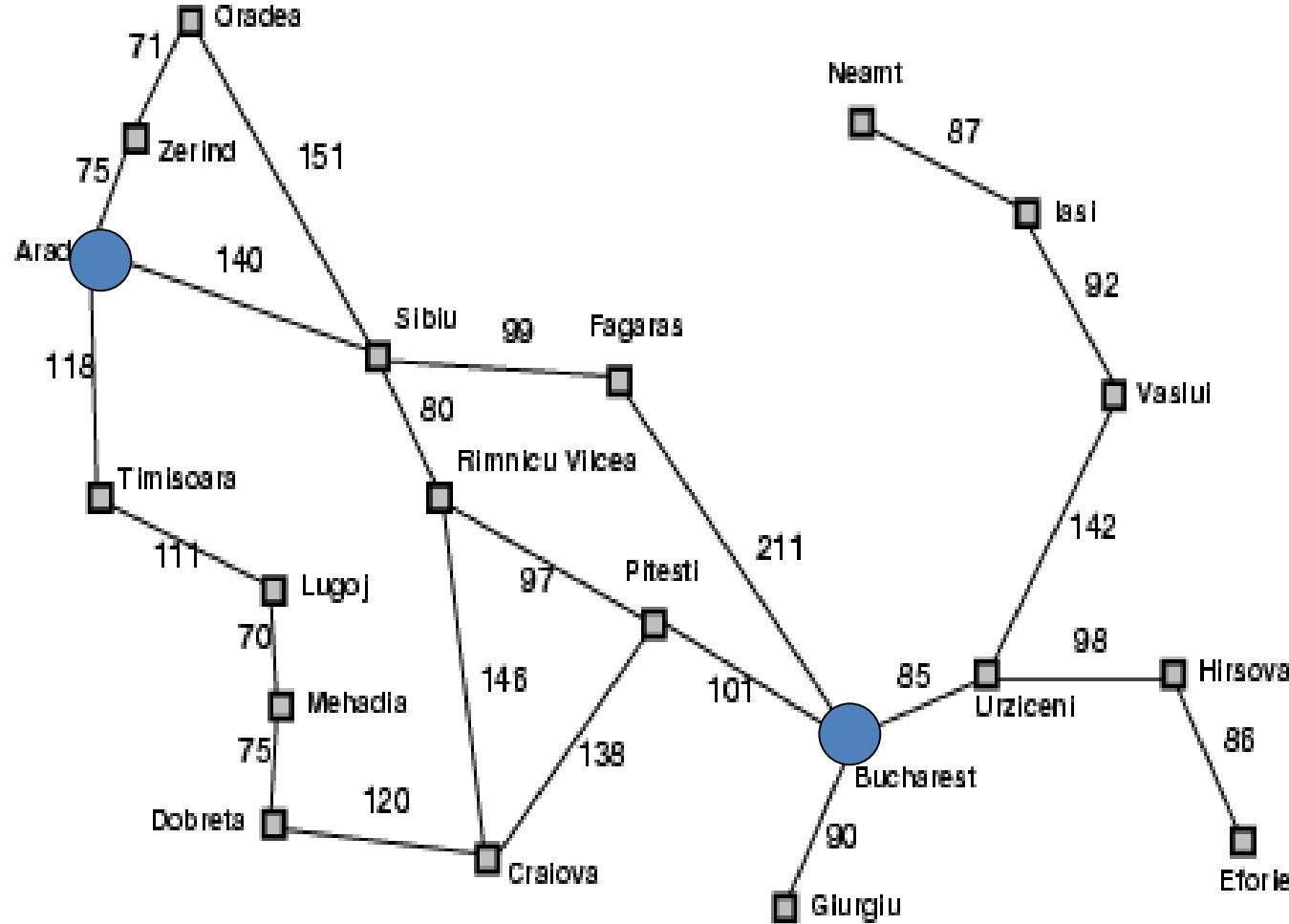


# Romania with step costs in km



# Greedy best-first search

expand the node that is **closest** to the goal : *Straight line distance* heuristic



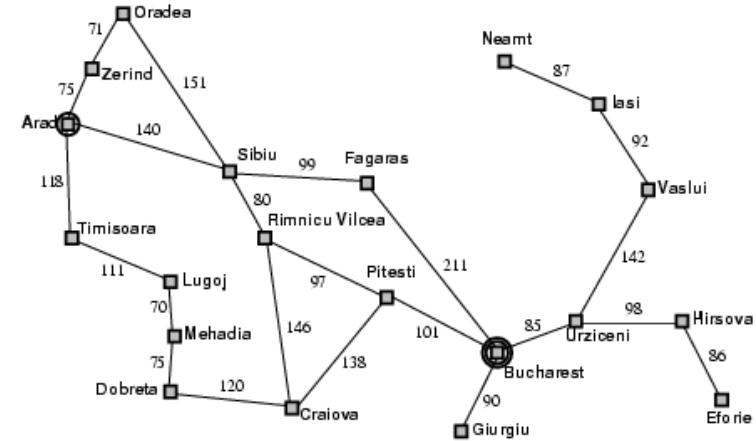
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

# Greedy best-first search properties

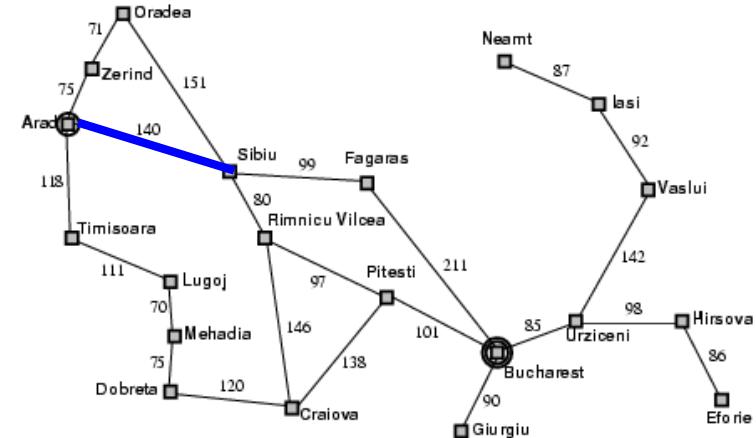
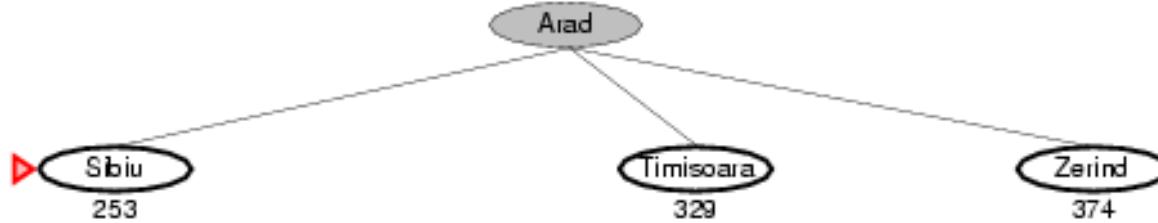
- **Time Complexity:** The worst case time complexity of Greedy best first search is  $O(b^m)$ .
- **Space Complexity:** The worst case space complexity of Greedy best first search is  $O(b^m)$ . Where, m is the maximum depth of the search space.
- **Complete:** Greedy best-first search is also incomplete, even if the given state space is finite.
- **Optimal:** Greedy best first search algorithm is not optimal.

# Greedy best-first search example

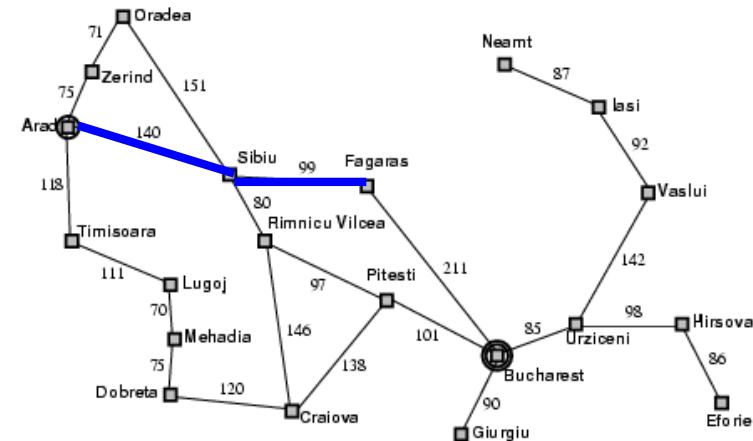
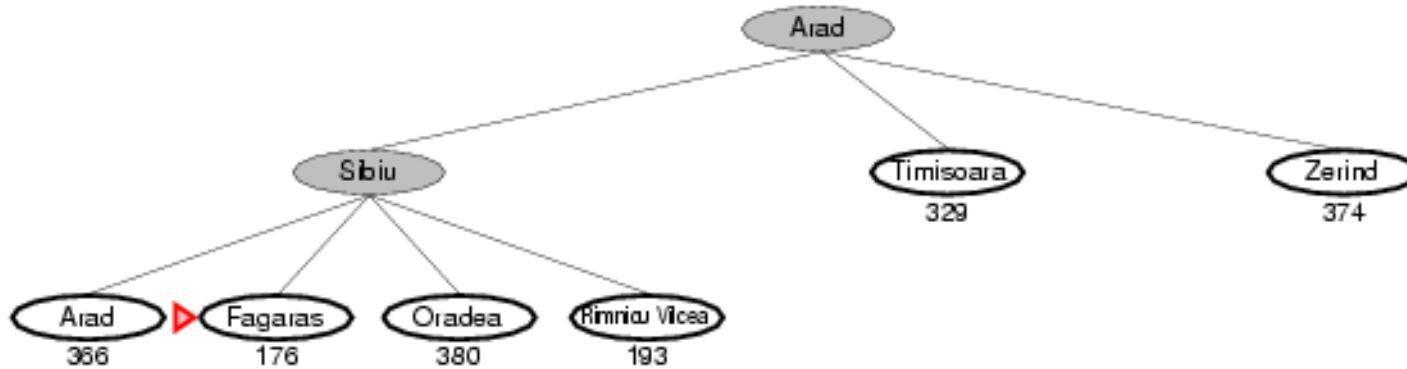
► Arad  
366



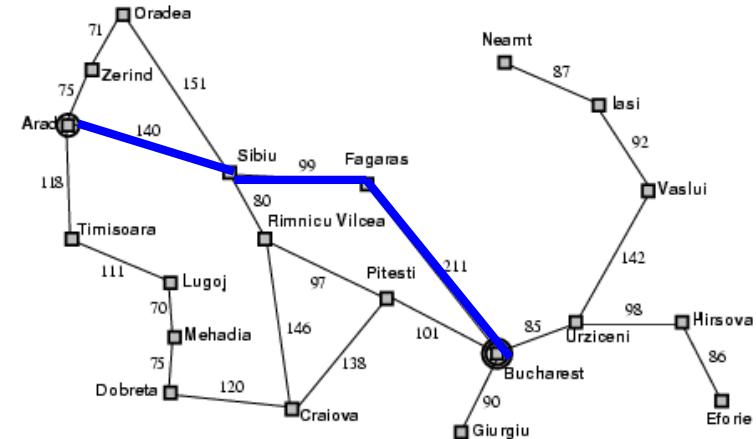
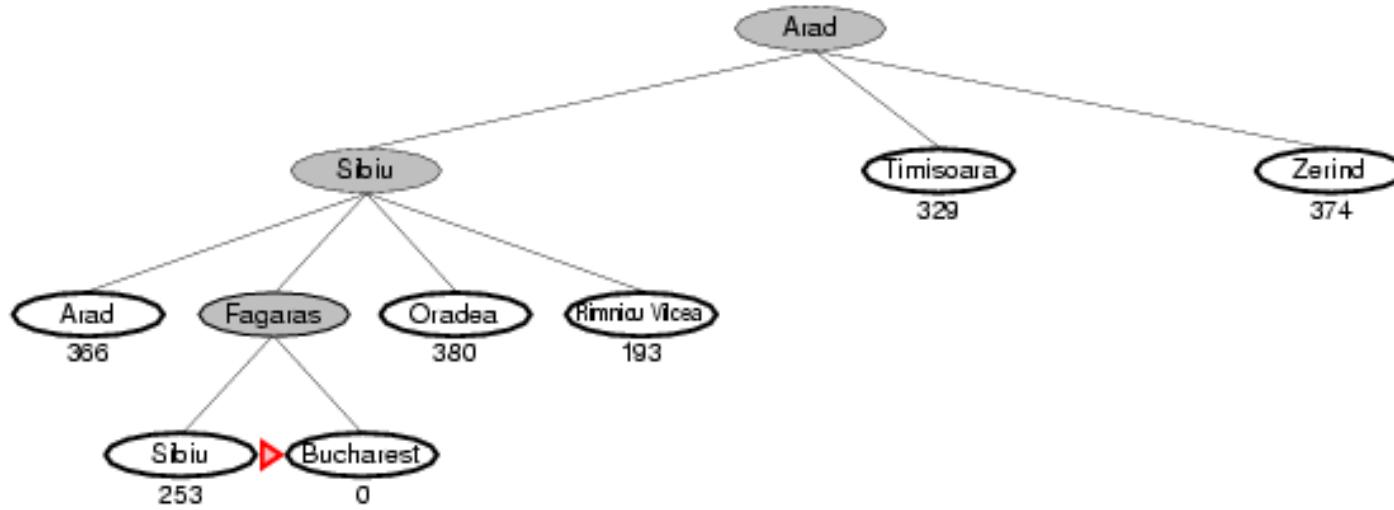
# Greedy best-first search example



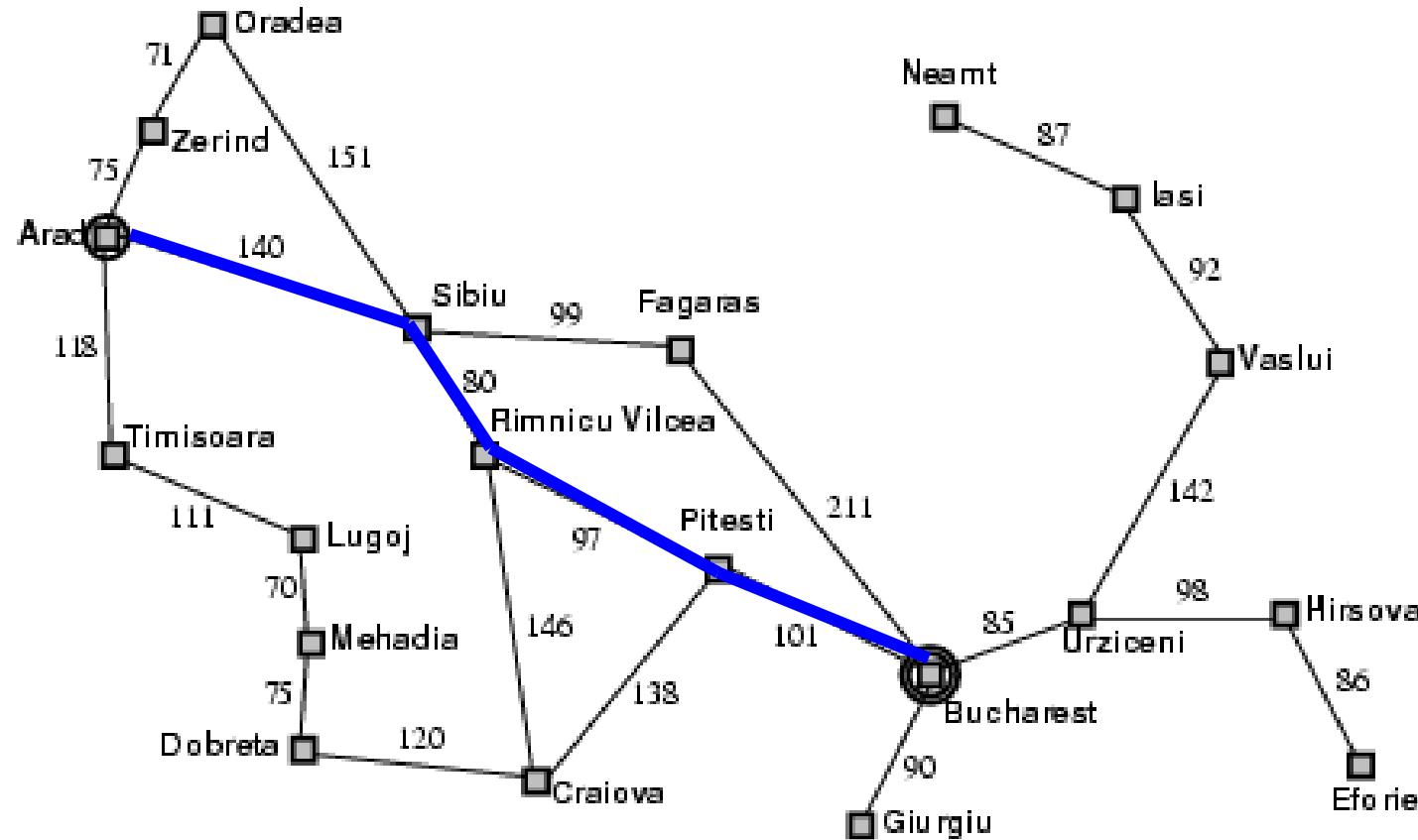
# Greedy best-first search example



# Greedy best-first search example

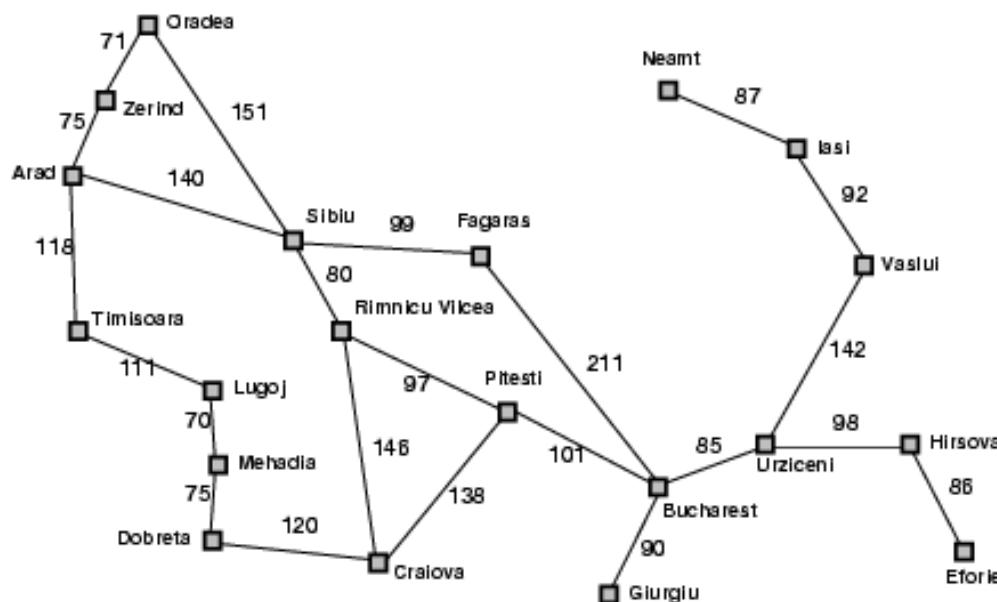


# Optimal Path



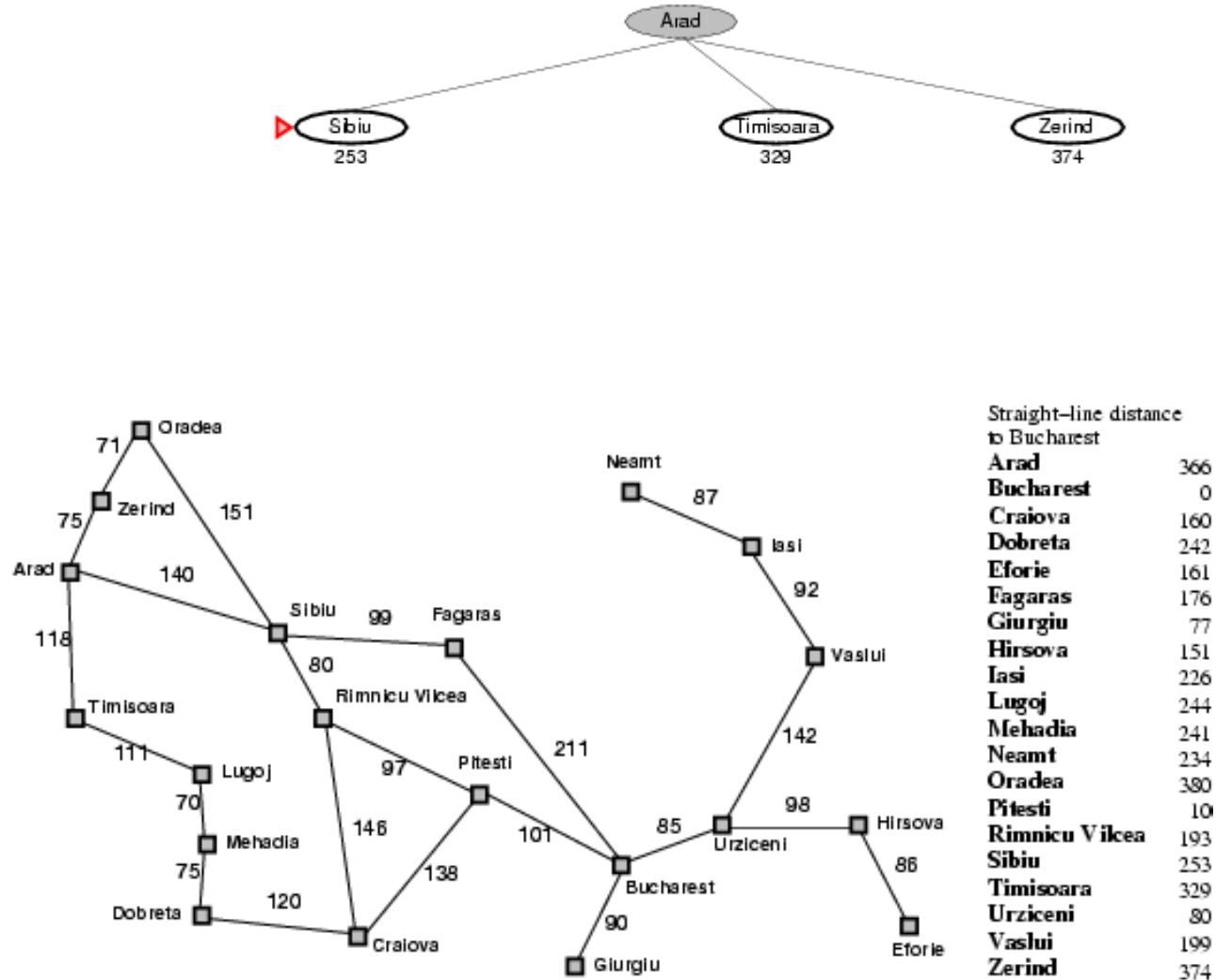
# Greedy best-first search example

► Arad  
366

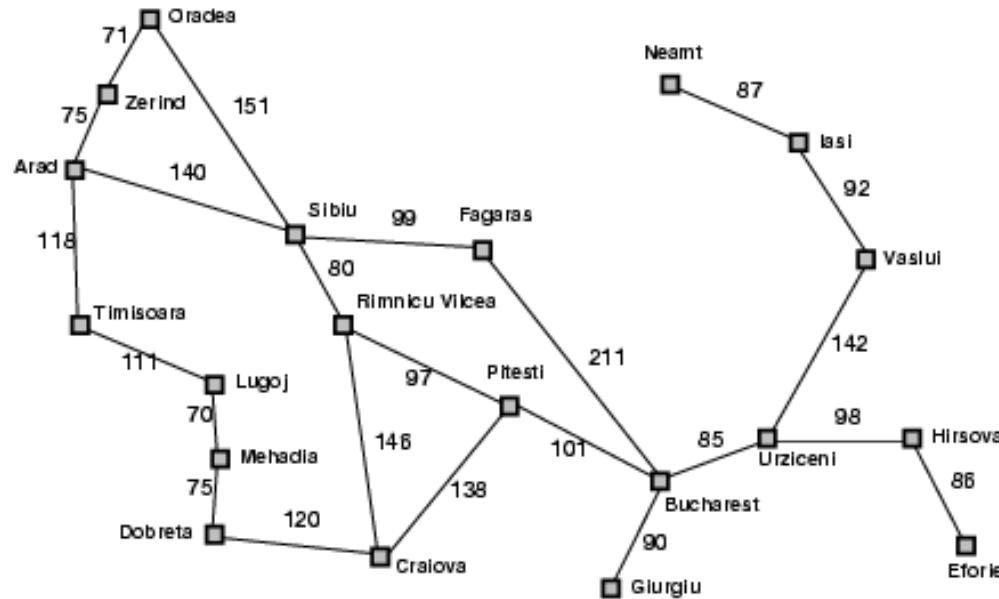
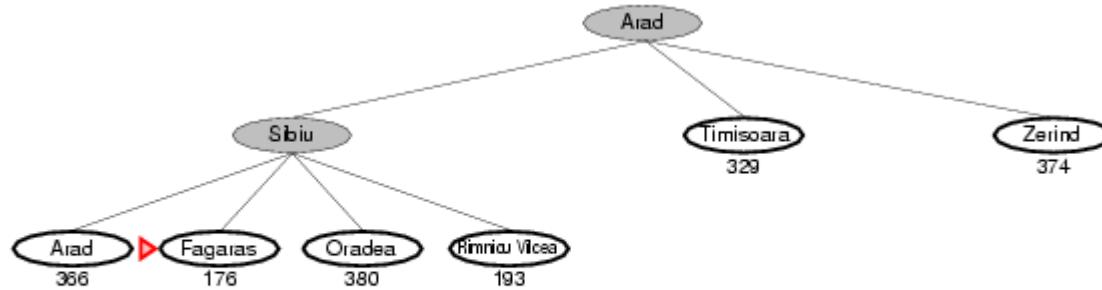


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Lasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

# Greedy best-first search example



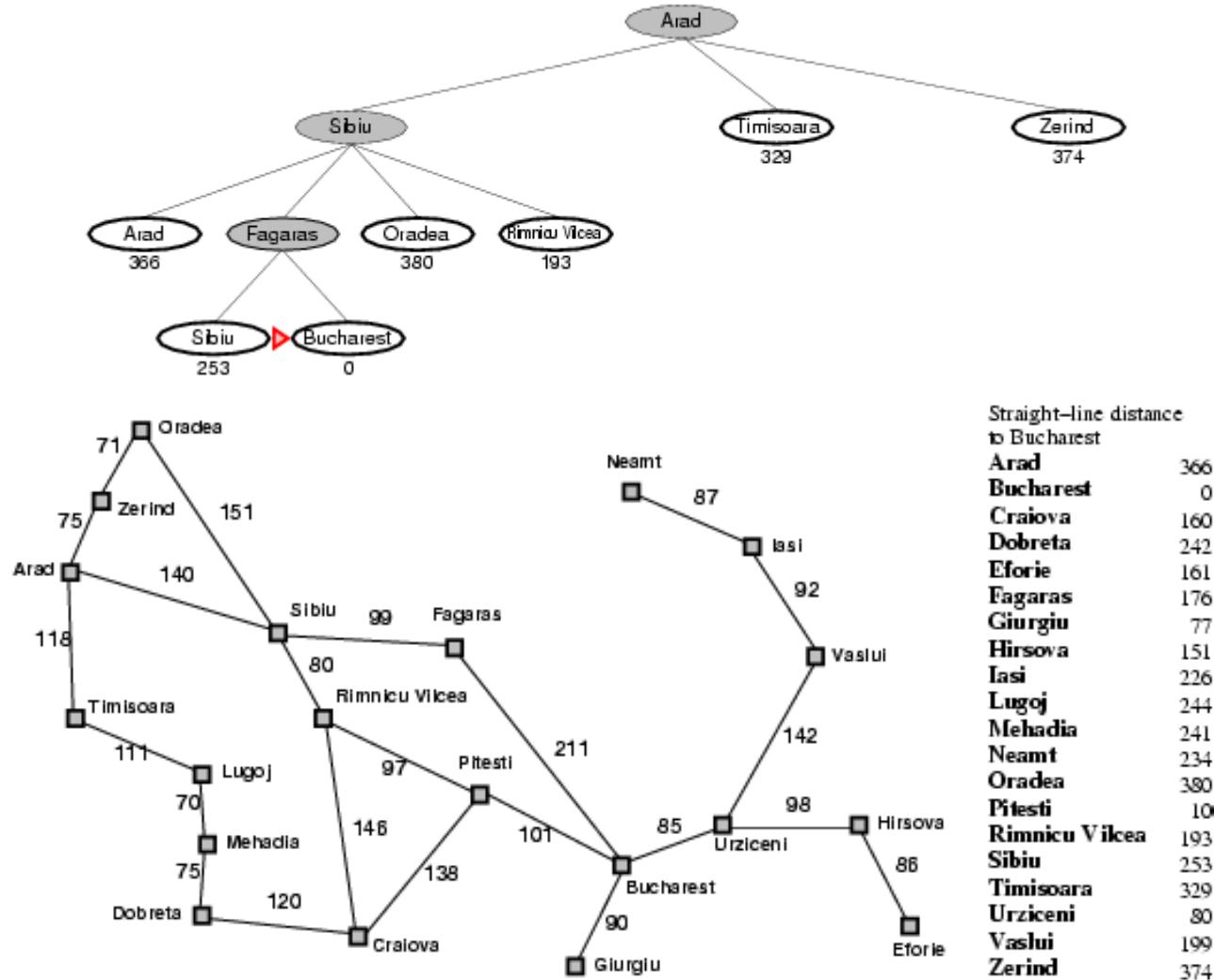
# Greedy best-first search example



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

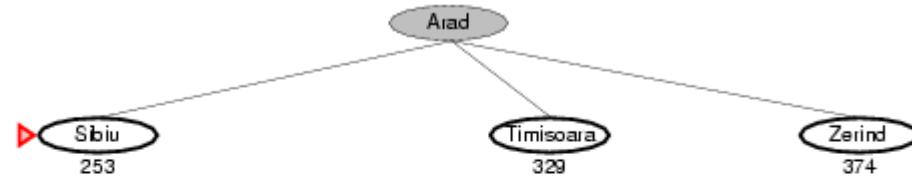
# Greedy best-first search example



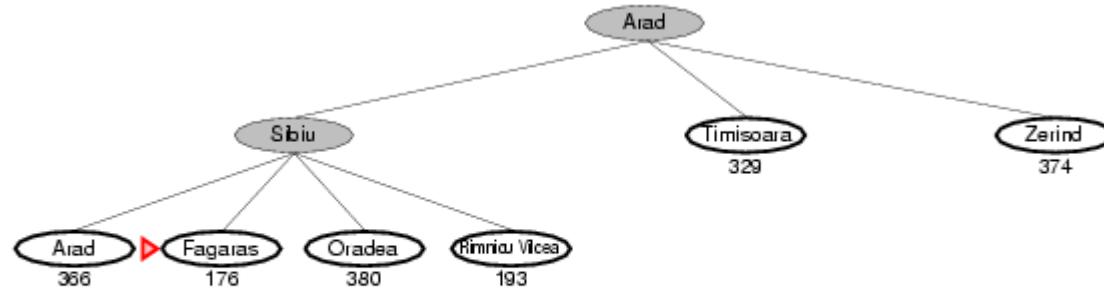
# Greedy best-first search example

► Aiad  
366

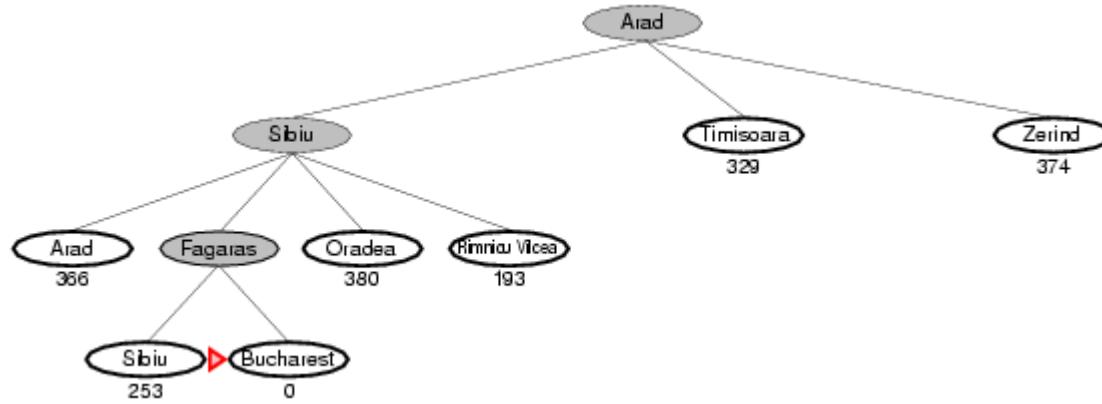
# Greedy best-first search example



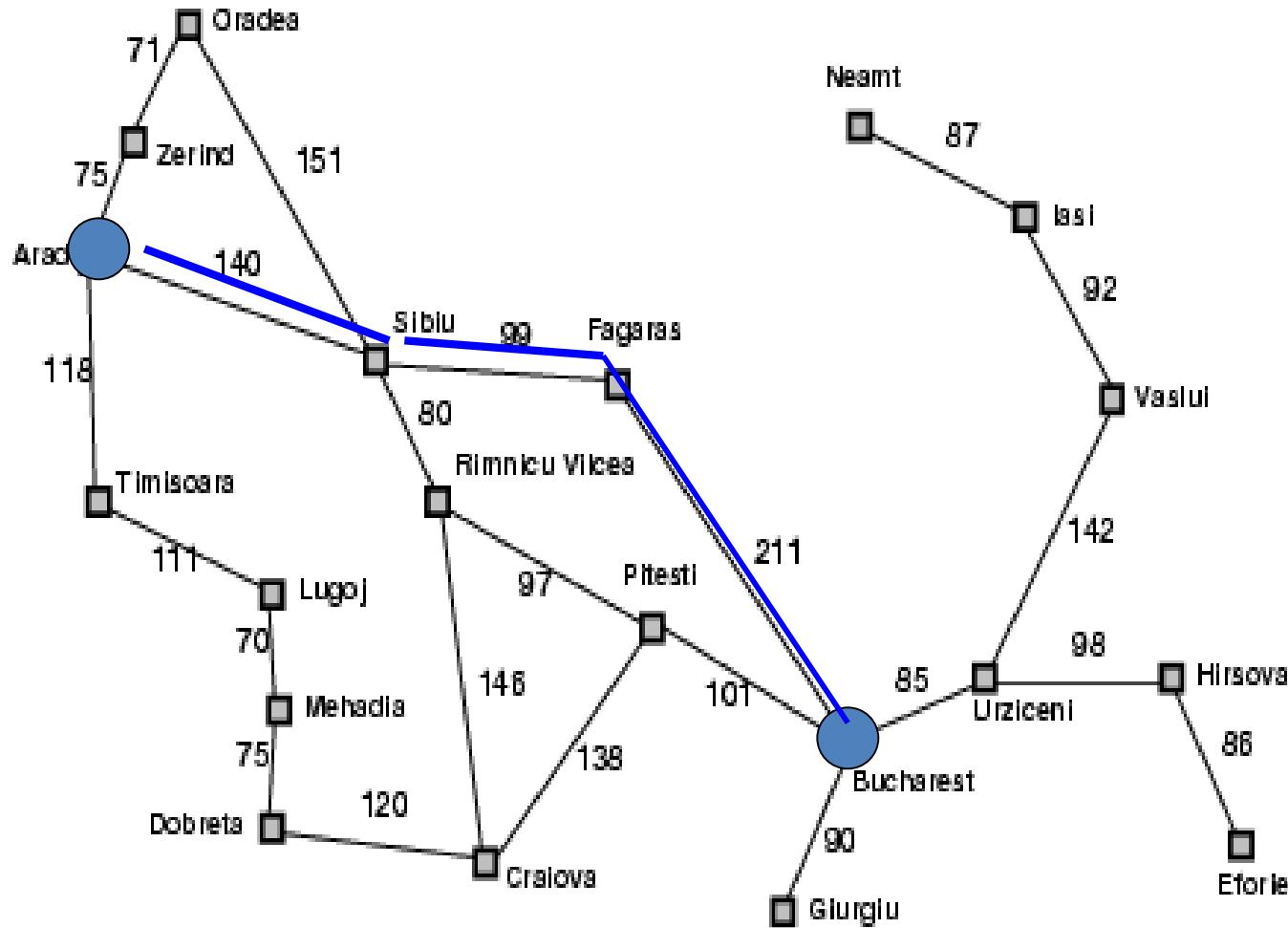
# Greedy best-first search example



# Greedy best-first search example



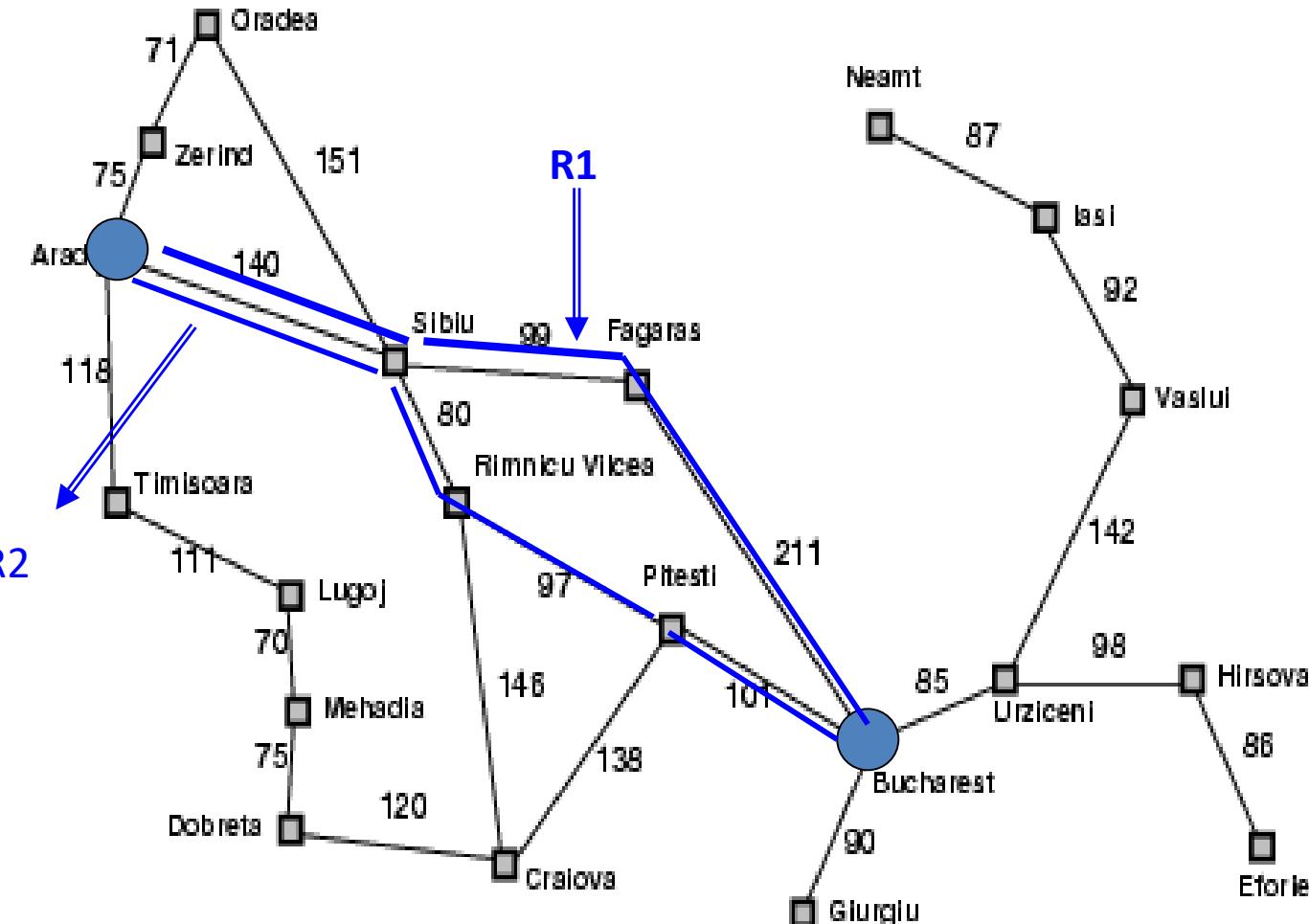
# Romania with step costs in km



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

R1 Arad → sibiu → fagaras → Bucharest =  $140 + 99 + 211 = 450$

# Romania with step costs in km



Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobrete	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

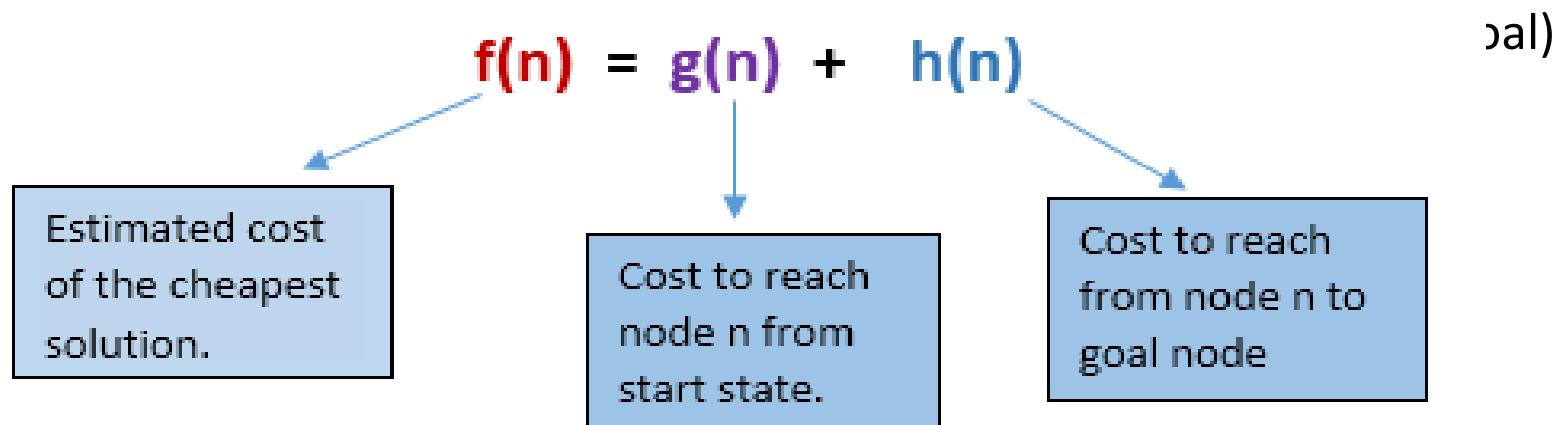
R1: Arad → sibiu → fagaras → Bucharest =  $140 + 99 + 211 = 450$  (Greedy)

R2: Arad → sibiu → Rimnicu vilcea → pitesti → Bucharest =  $140 + 80 + 97 + 101 = 418$

# A\* Algorithm

The famous A\* algorithm. **Nilsson & Rafael** in 1968.

- In A\* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.
- A\* is a best first search  $f(n) = g(n) + h'(n)$ 
  - $g(n)$  = sum of edge costs from start to n(start node ↳ current node)
  - $h'(n)$  = estimate of lowest cost path from n to goal



$h(n)$  is said to be **admissible** if it underestimates the cost of solution that can be reached from  $n$ .

If  $C^*(n)$  is the cost of the cheapest sol path from  $n$  to goal & if  $h'$  is admissible,

$$h'(n) \leq C^*(n).$$

we can prove that if  $h'(n)$  is admissible, then the search will find an optimal solution.

# Advantages and Disadvantages of A\* search

## Advantages:

- A\* search algorithm is the best algorithm than other search algorithms.
- A\* search algorithm is optimal and complete.
- This algorithm can solve very complex problems.

## Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A\* search algorithm has some complexity issues.
- The main drawback of A\* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

## Algorithm A\*

OPEN = nodes on frontier. CLOSED = expanded nodes.

OPEN = { $\langle s, \text{nil} \rangle$ }

while OPEN is not empty

    remove from OPEN the node  $\langle n, p \rangle$  with minimum  $f(n)$

    place  $\langle n, p \rangle$  on CLOSED

    if  $n$  is a goal node,

        return success (path  $p$ )

    for each edge connecting  $n$  &  $m$  with cost  $c$

        if  $\langle m, q \rangle$  is on CLOSED and  $\{p|e\}$  is cheaper than  $q$

            → then remove  $n$  from CLOSED,

                put  $\langle m, \{p|e\} \rangle$  on OPEN

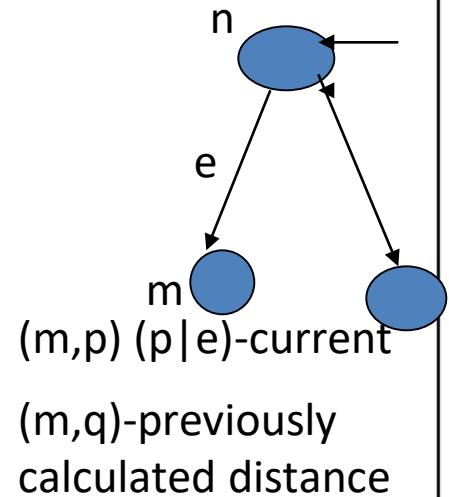
        else if  $\langle m, q \rangle$  is on OPEN and  $\{p|e\}$  is cheaper than  $q$

            → then replace  $q$  with  $\{p|e\}$

        else if  $m$  is not on OPEN

            → then put  $\langle m, \{p|e\} \rangle$  on OPEN

return failure

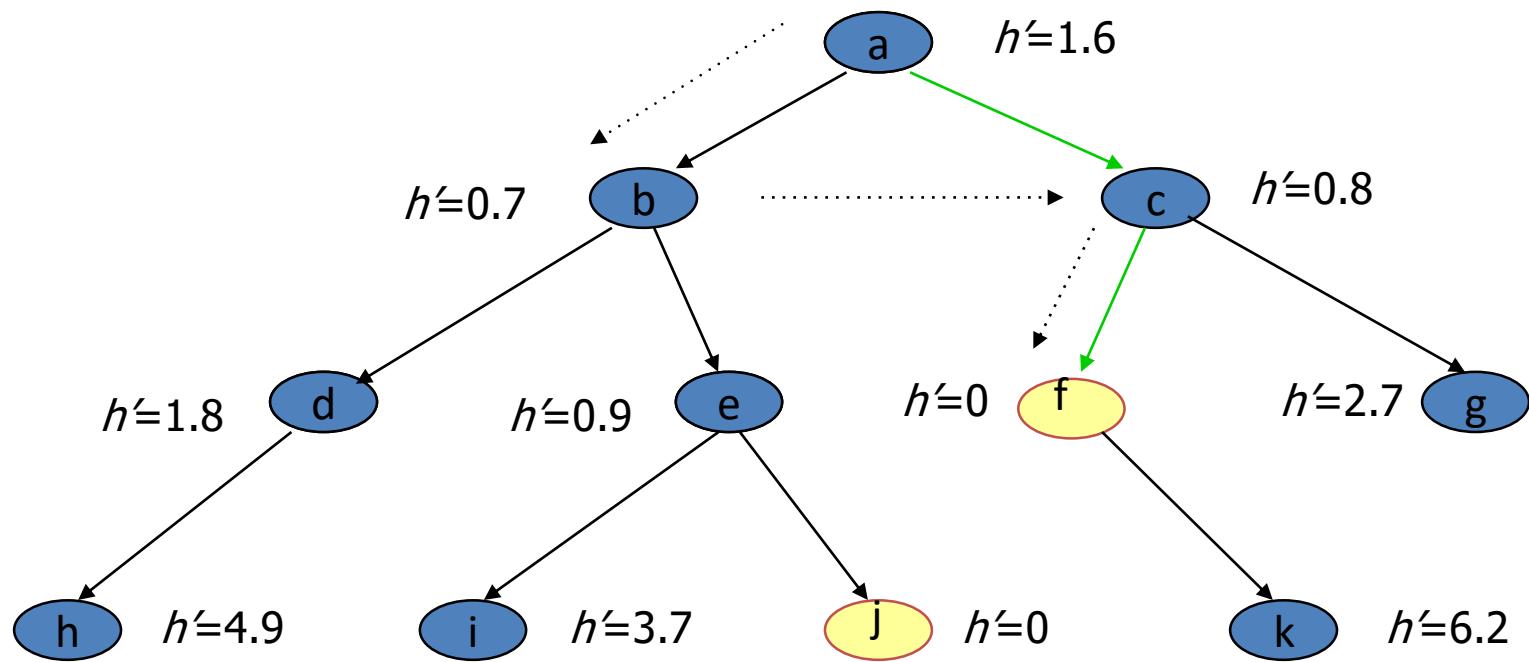


# A\* search: properties

## Points to remember:

- A\* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A\* algorithm depends on the quality of heuristic.
- A\* algorithm expands all nodes which satisfy the condition  $f(n)$
- **Complete:** A\* algorithm is complete as long as:
  - Branching factor is finite, Cost at every action is fixed.
- **Optimal:** A\* search algorithm is optimal if it follows below two conditions:
- **Admissible:** The first condition requires for optimality is that  $h(n)$  should be an admissible heuristic for A\* tree search. An admissible heuristic is optimistic in nature.
- **Consistency:** Second required condition is consistency for only A\* graph-search.
- If the heuristic function is admissible, then A\* tree search will always find the least cost path.
- **Time Complexity:** The time complexity of A\* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution  $d$ . So the time complexity is  $O(b^d)$ , where  $b$  is the branching factor.
- **Space Complexity:** The space complexity of A\* search algorithm is  $O(b^d)$

# Example of A\* search

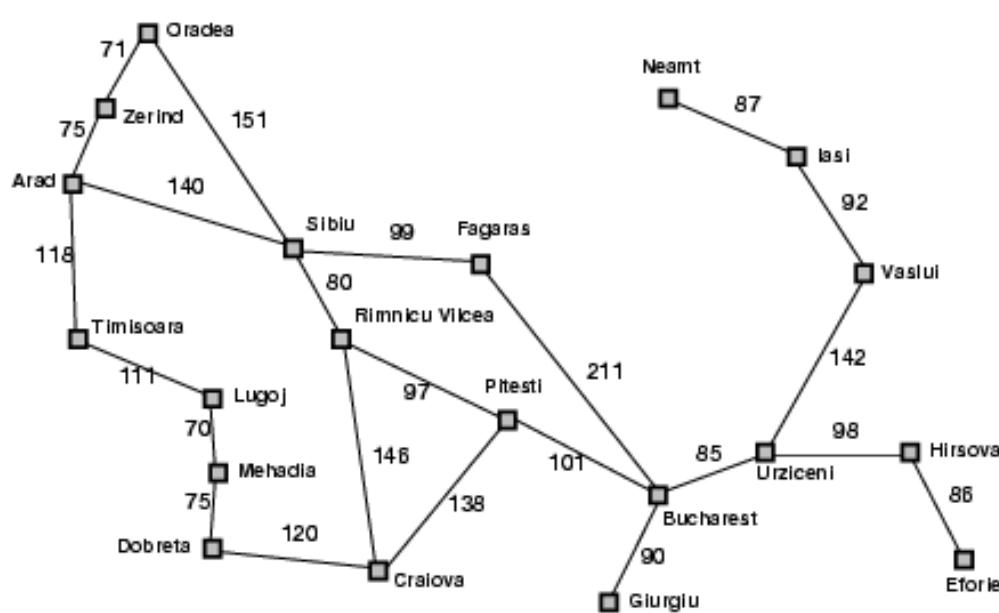


Cost per arc = 1

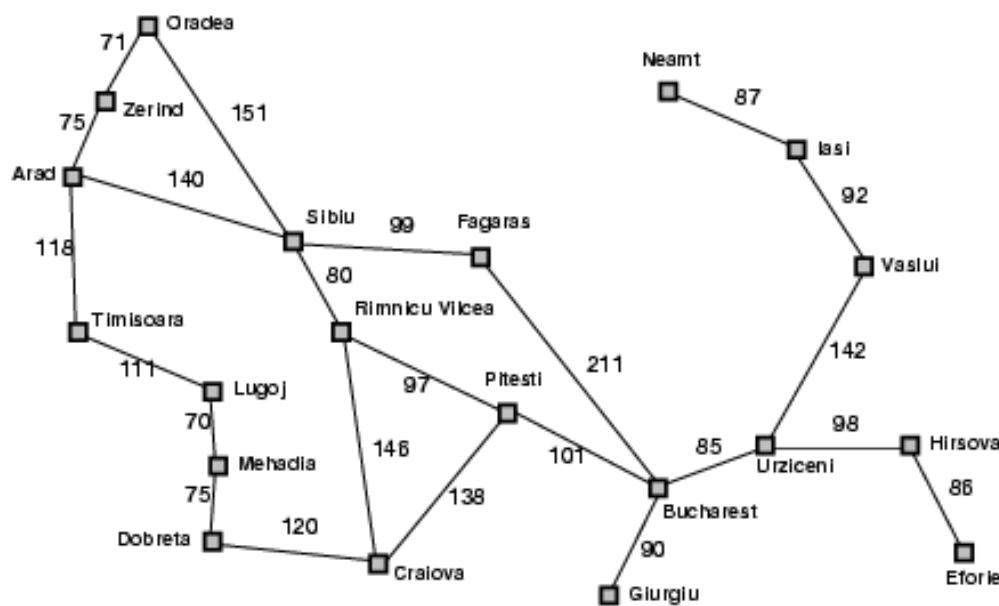
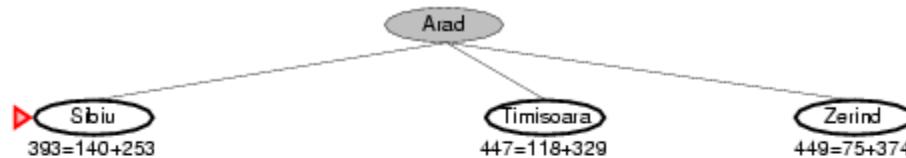
$h'$  values are admissible, e.g. at b, actual cost of reaching goal (j) is  $1+1=2$  but  $h'$  is only 0.7. At b,  $f(b) = g(b) + h'(b) = 1 + 0.7 = 1.7$

# A\* search example

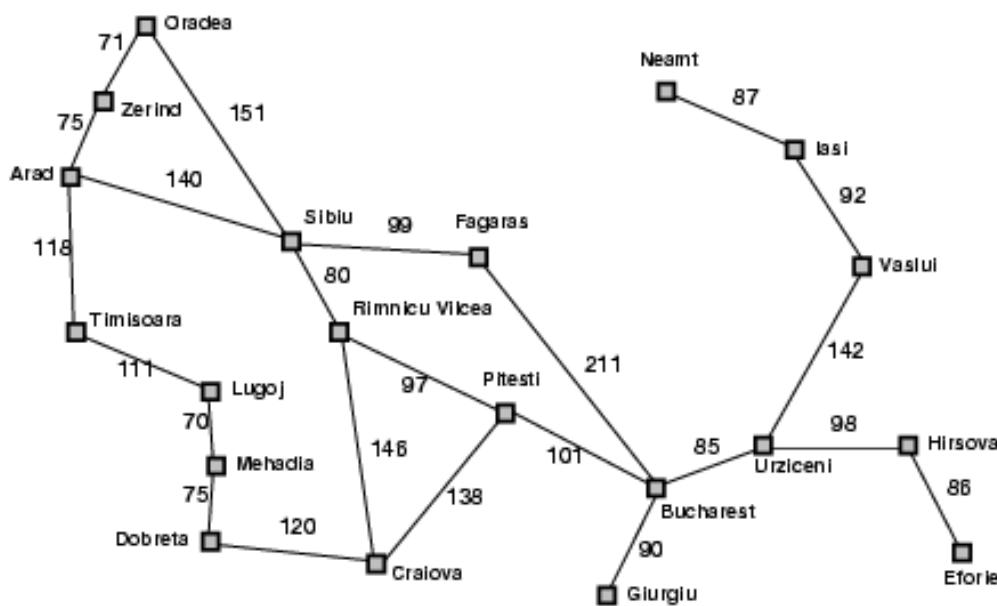
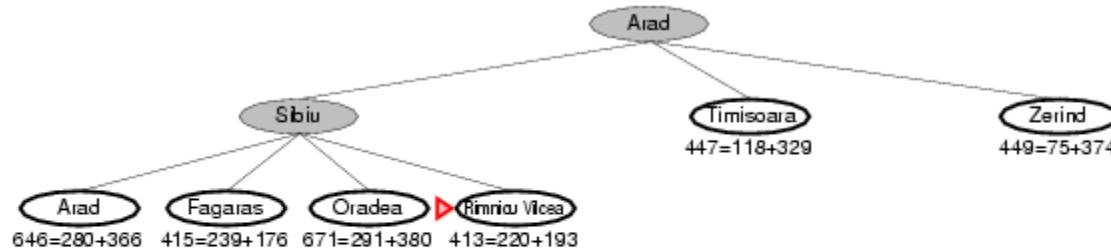
► Arad  
366=0+366



# A\* search example

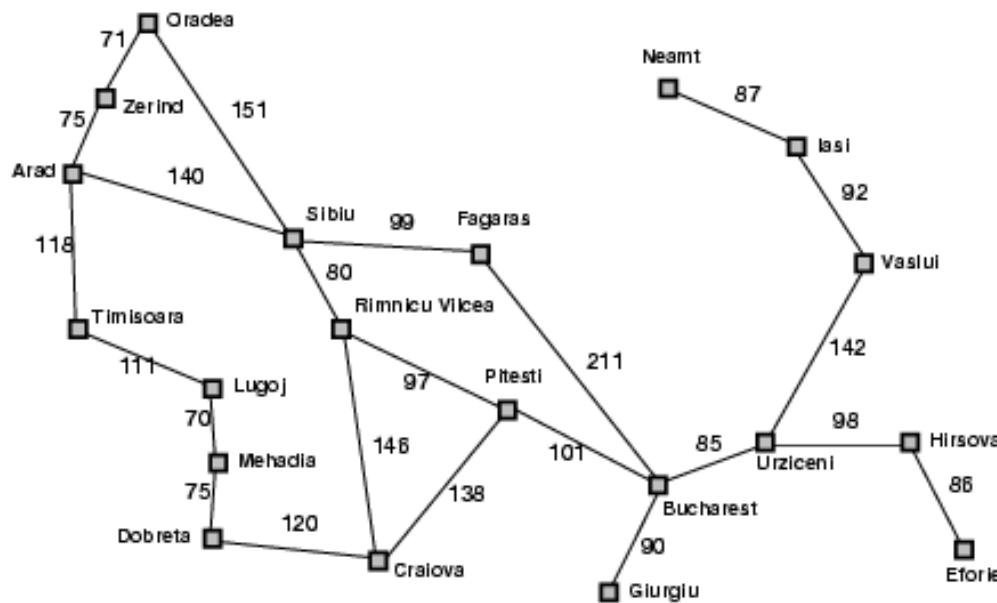
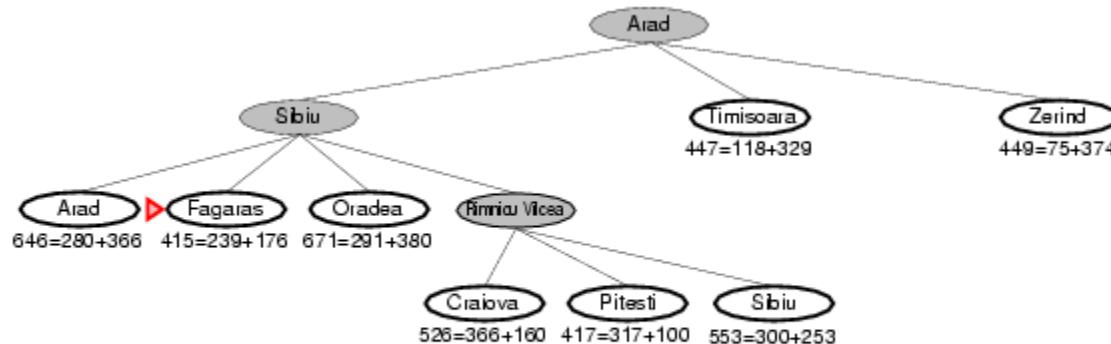


# A\* search example



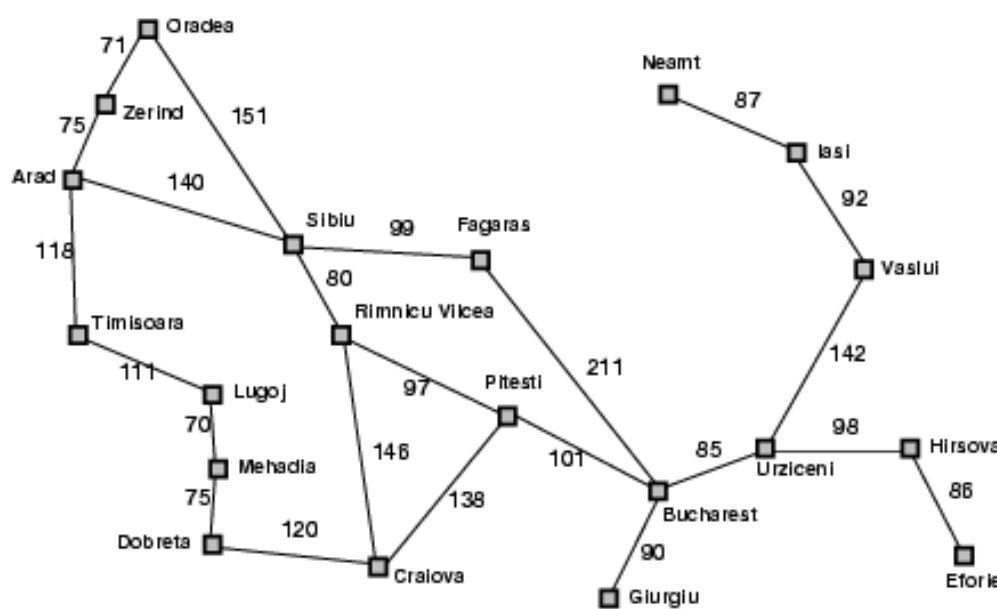
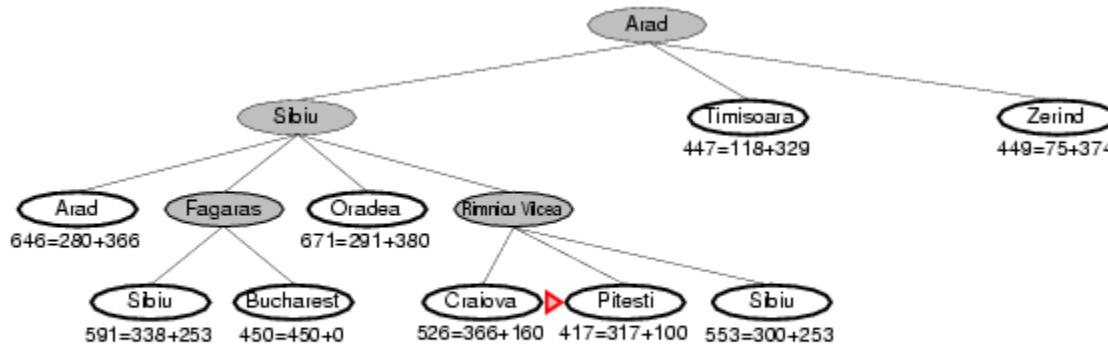
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

# A\* search example



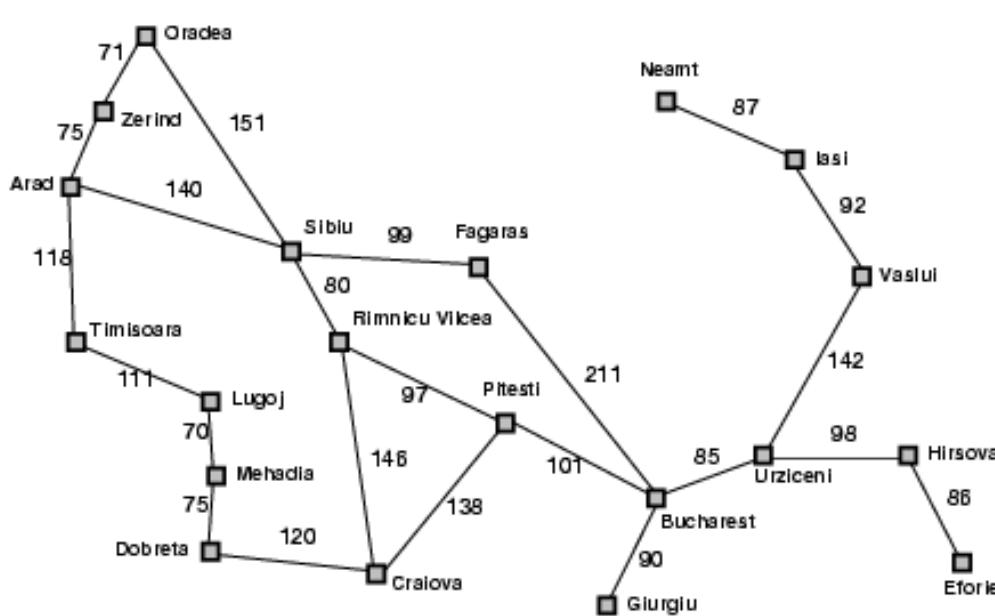
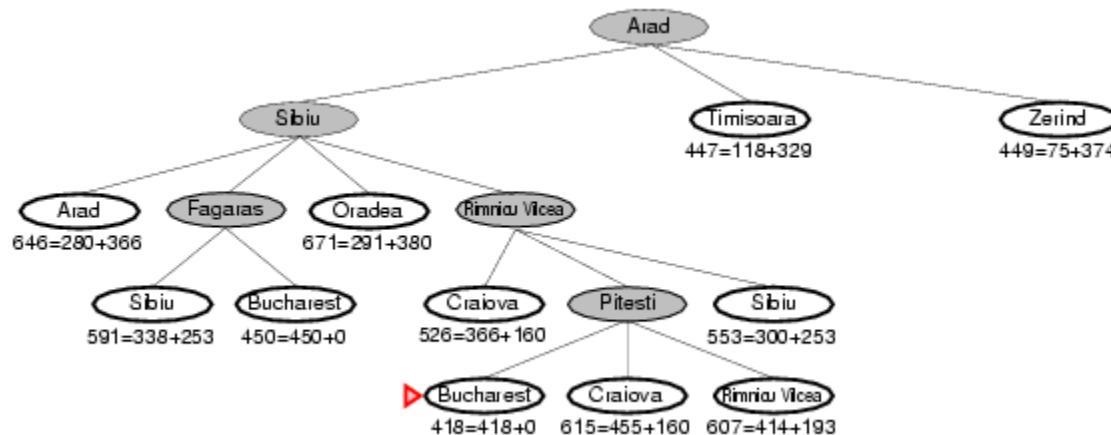
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

# A\* search example



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

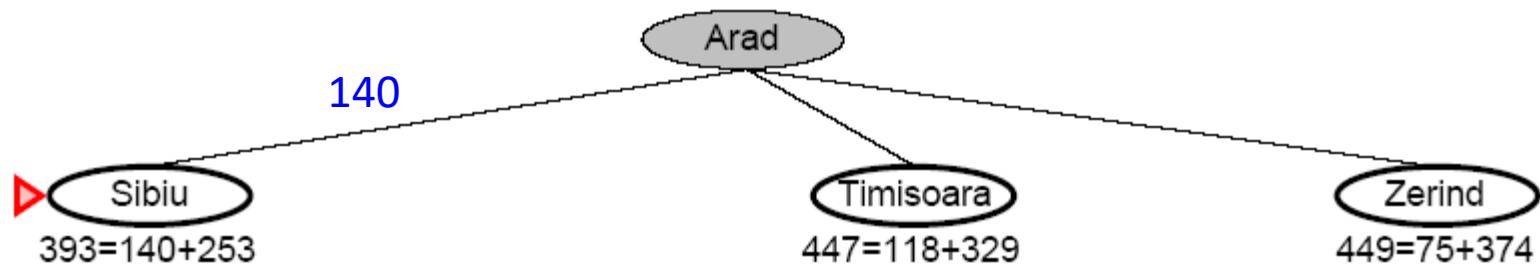
# A\* search example



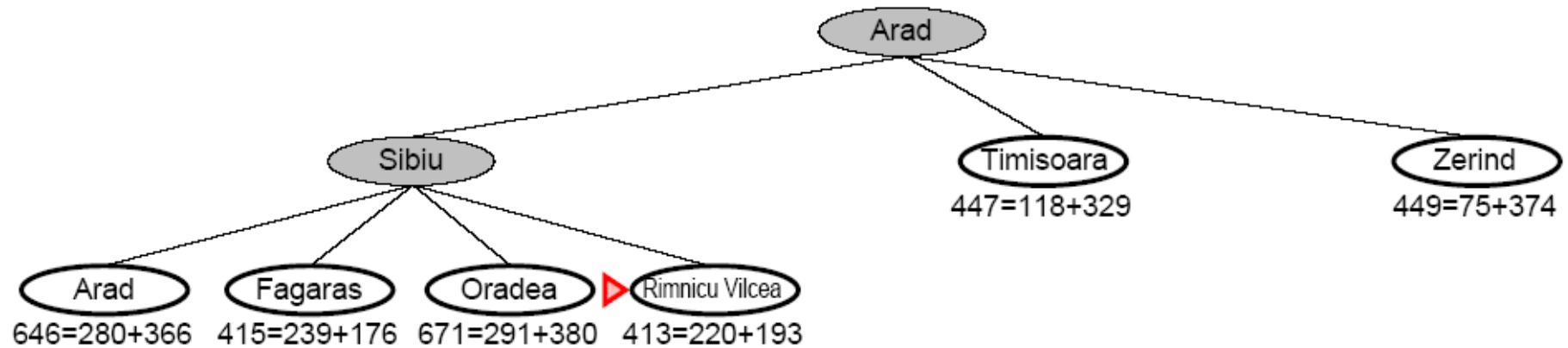
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

# A\* Search

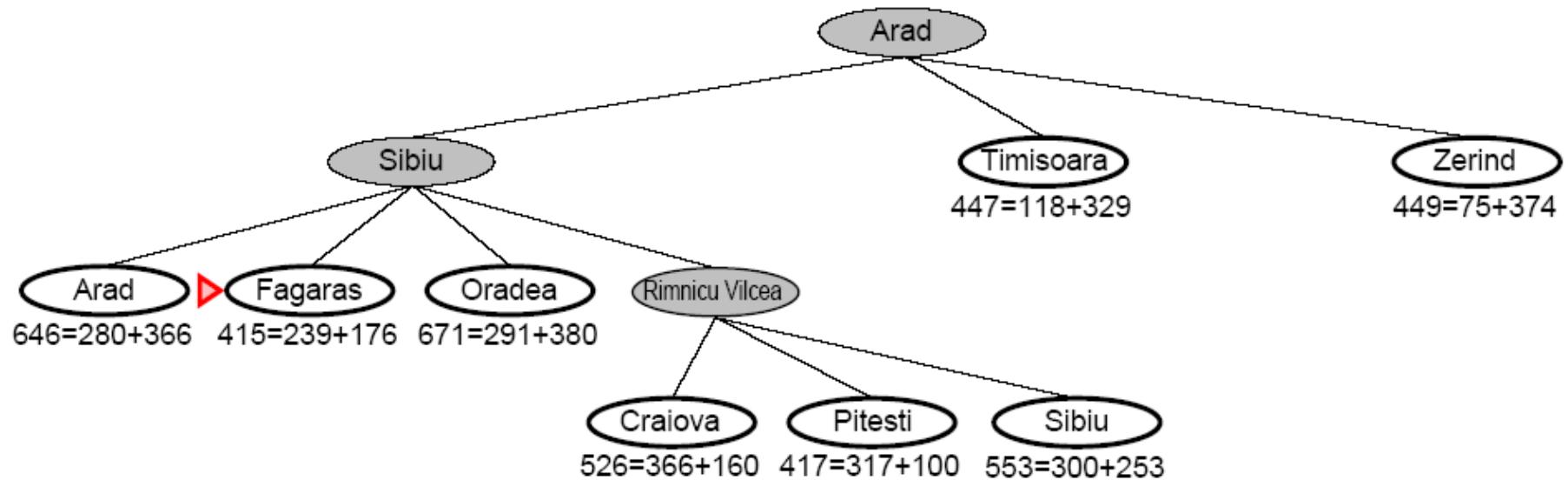
► Arad  
 $366=0+366$



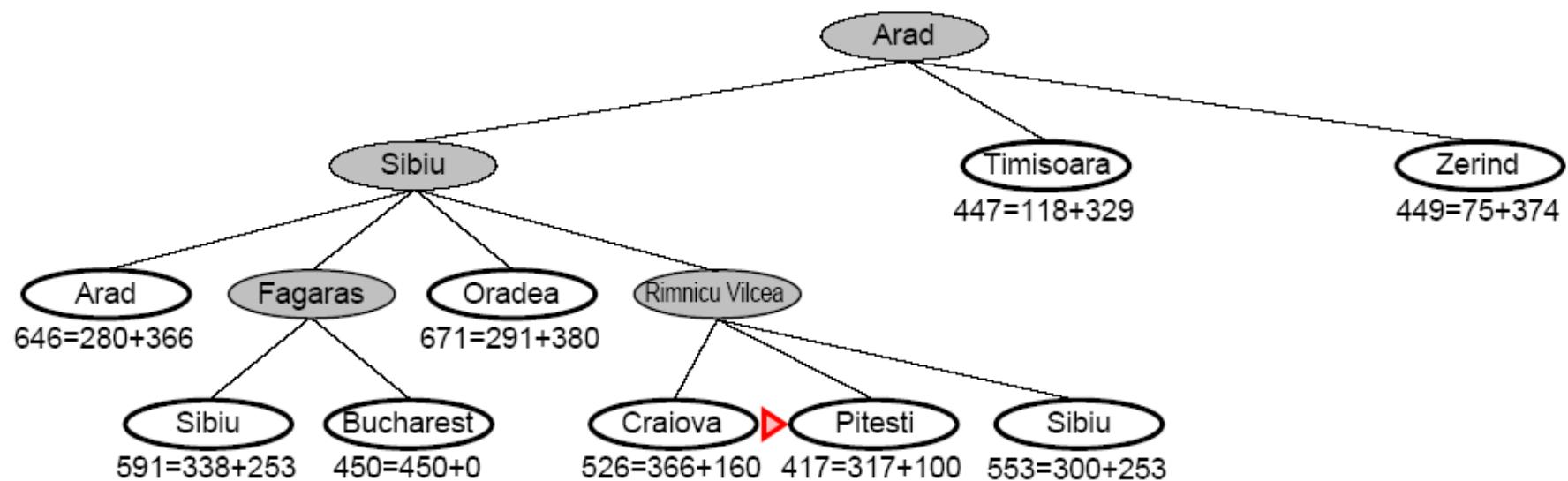
# A\* Search



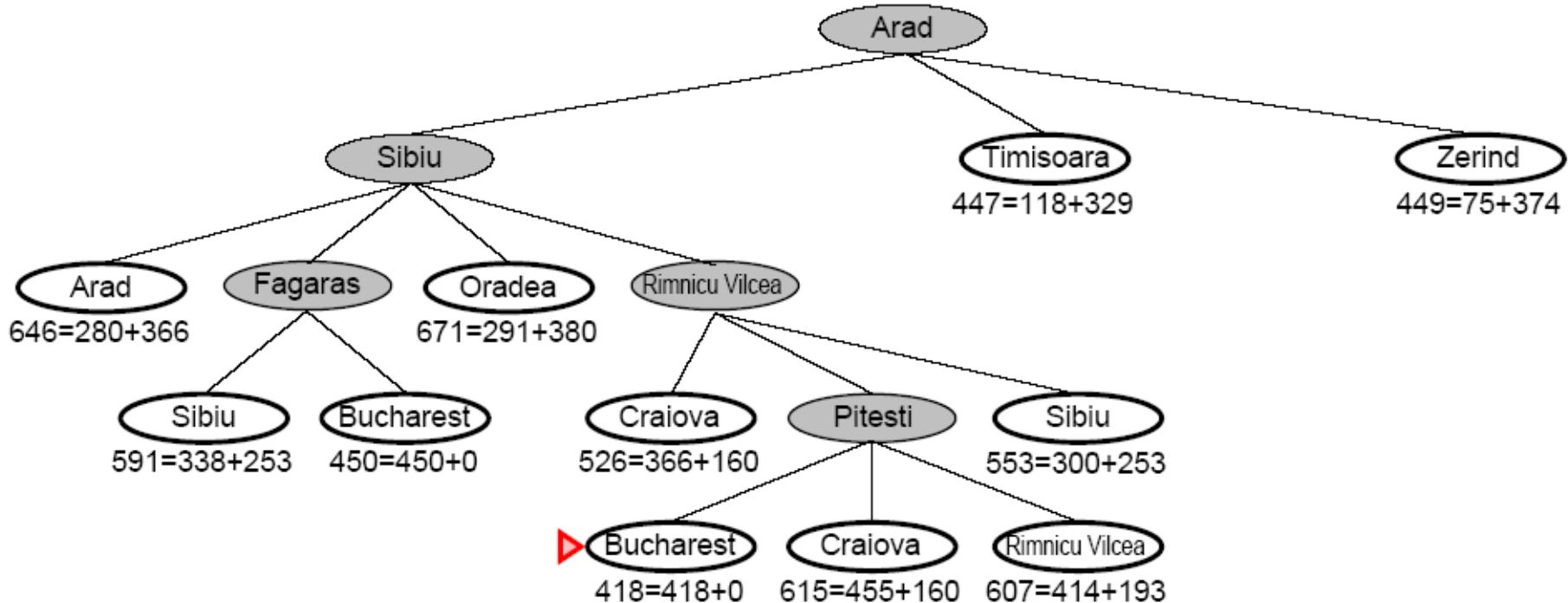
# A\* Search



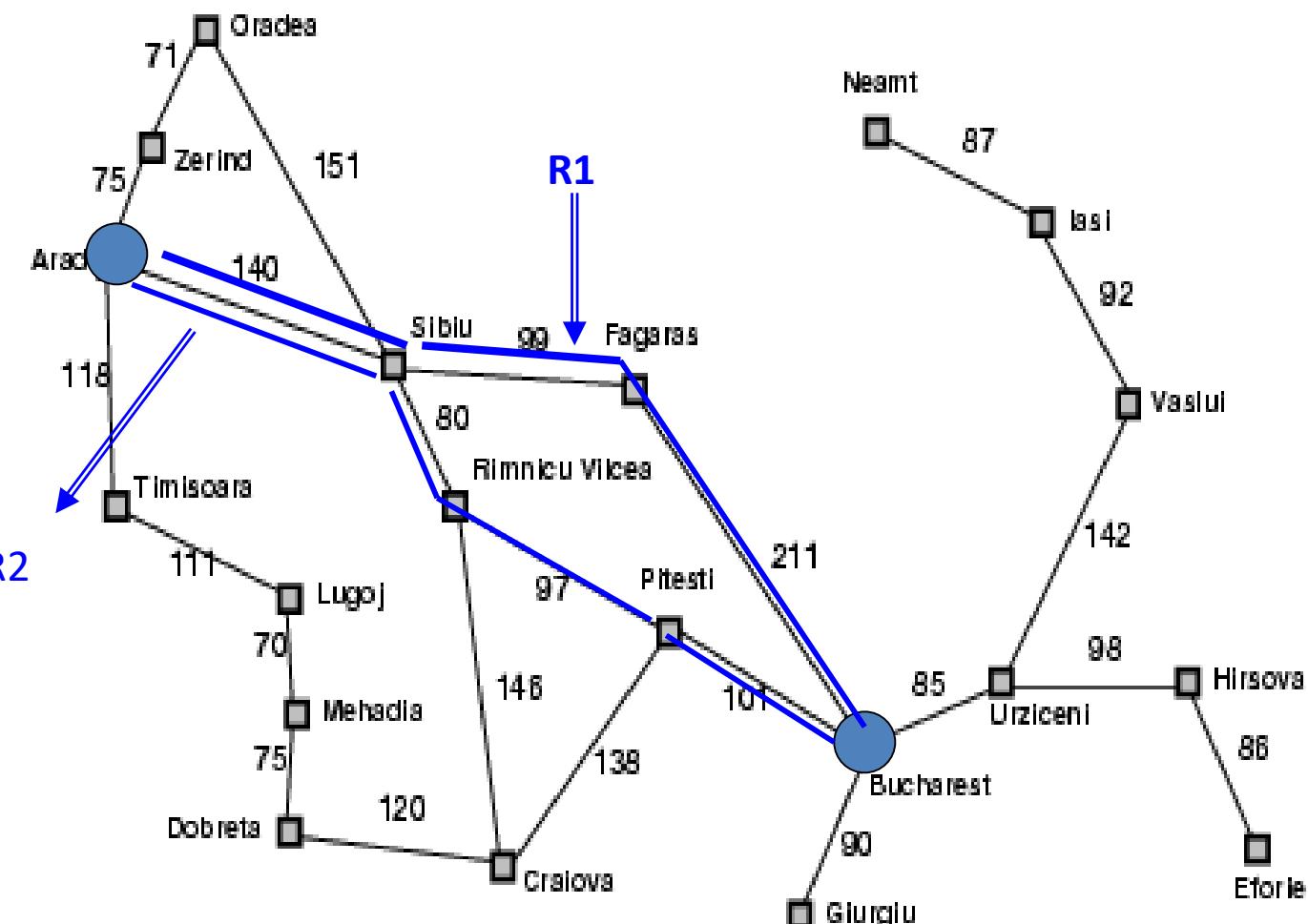
# A\* Search



# A\* Search



# Romania with step costs in km



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

R1: Arad → sibiu → fagaras → Bucharest =  $140 + 99 + 211 = 450$  (Greedy)

R2: Arad → sibiu → Rimnicu vilcea → pitesti → Bucharest =  $140 + 80 + 97 + 101 = 418$  (A\*)

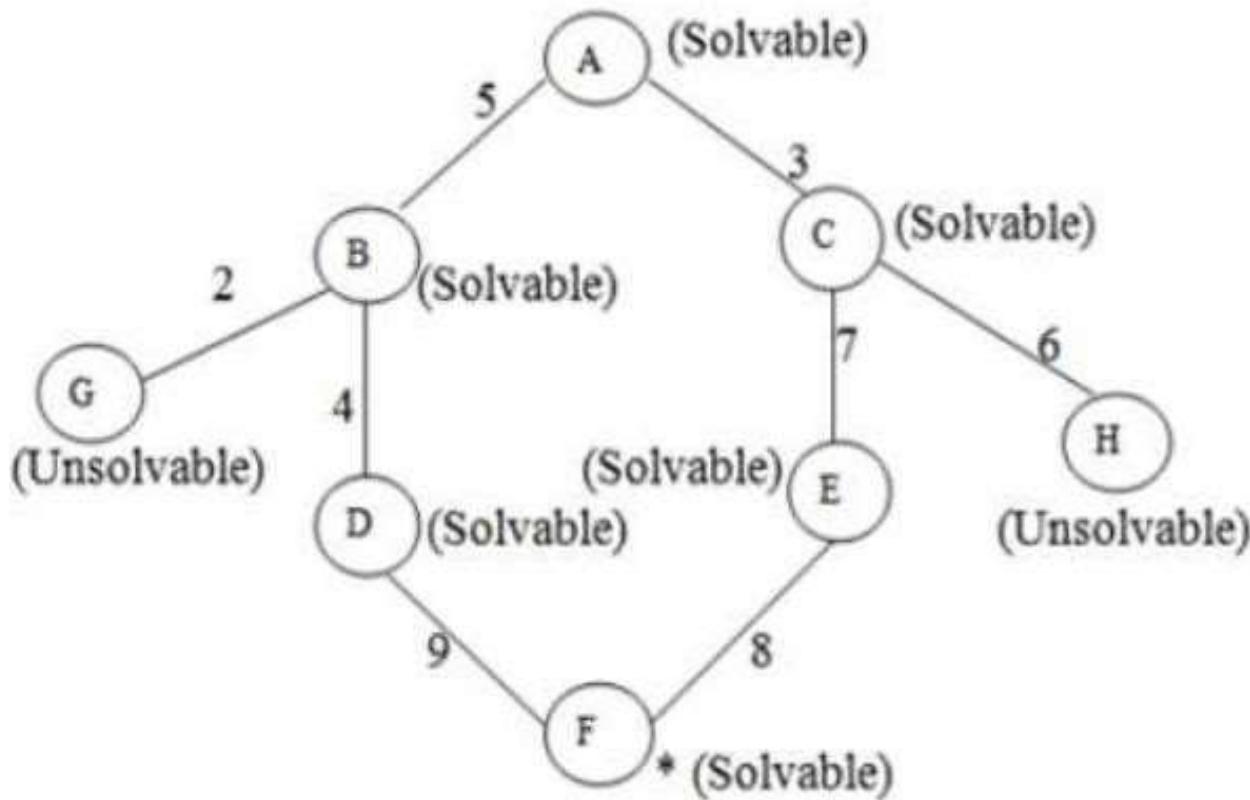
## AO\* GRAPH AND-OR GRAPH

- The Depth first search and Breadth first search given earlier for OR trees or graphs can be easily adopted by AND-OR graph.
  - The main difference lies in the way termination conditions are determined, since all goals following an AND nodes must be realized; whereas a single goal node following an OR node will do.
  - So for this purpose we are using AO\* algorithm.
  - Like A\* algorithm here we will use two arrays and one heuristic function.
  - OPEN:
    - It contains the nodes that has been traversed but yet not been marked solvable or unsolvable.
  - CLOSE:
    - It contains the nodes that have already been processed.

# ALGORITHM

- Step 1: Place the starting node into OPEN.
- Step 2: Compute the most promising solution tree say T<sub>0</sub>.
- Step 3: Select a node n that is both on OPEN and a member of T<sub>0</sub>. Remove it from OPEN and place it in CLOSE
- Step 4: If n is the terminal goal node then level n as solved and level . all the ancestors of n as solved.  
If the starting node is marked as solved then success and exit.
- Step 5: If n is not a solvable node, then mark n as unsolvable.  
If starting node is marked as unsolvable, then return failure and exit.
- Step 6: Expand n. Find all its successors and find their h (n) value, push them into OPEN.
- Step 7: Return to Step 2.
- Step 8: Exit

# EXAMPLE FOR IMPLEMENTATION



- Let us take the following example to implement the AO\* algorithm

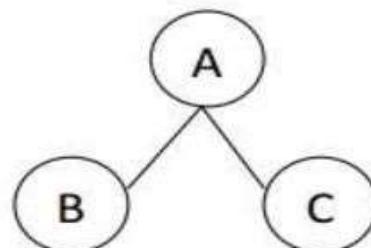
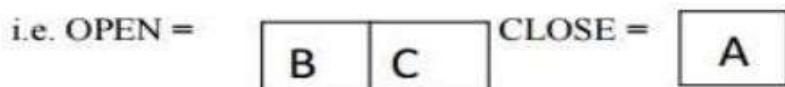
**Step 1:**

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



**Step 2:**

The children of A are B and C which are solvable. So place them into OPEN and place A into the CLOSE.



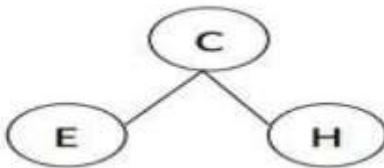
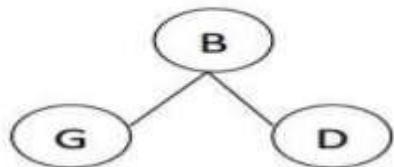
**Step 3:**

Now process the nodes B and C. The children of B and C are to be placed into OPEN. Also remove B and C from OPEN and place them into CLOSE.

So OPEN =

G	D	E	
---	---	---	--

C[



**Step 4:**

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

i.e. OPEN =

CLOSE =

D		E	A	B	C		G <sup>(O)</sup>	D	E	H <sup>(O)</sup>
---	--	---	---	---	---	--	------------------	---	---	------------------

```
graph TD; D((D)) --- E((E)); E --- F((F))
```

\*

**Step 5:**

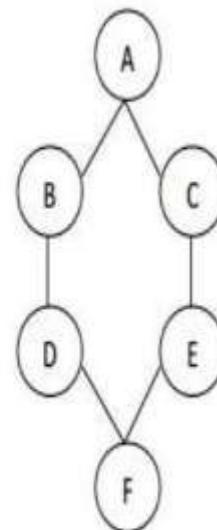
Now we have been reached at our goal state. So place F into CLOSE.

A	B	C		G <sup>(0)</sup>	D	E		H <sup>(0)</sup>	F
---	---	---	--	------------------	---	---	--	------------------	---

AO\* Graph:

**Step 6:**

Success and Exit

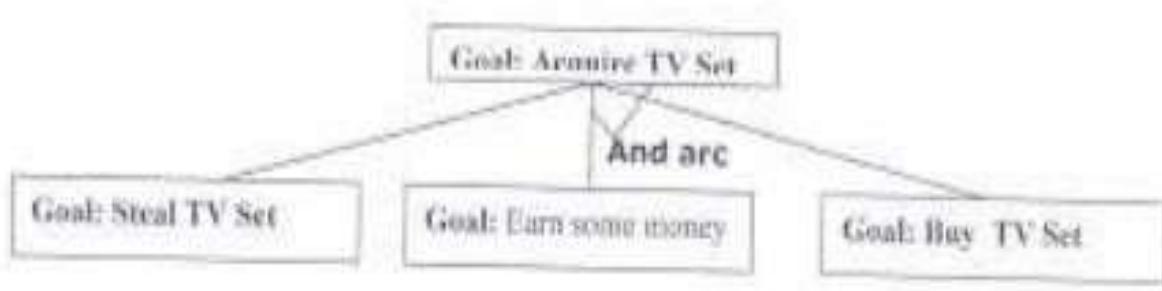


# ADVANTAGES AND DISADVANTAGES OF AO\*

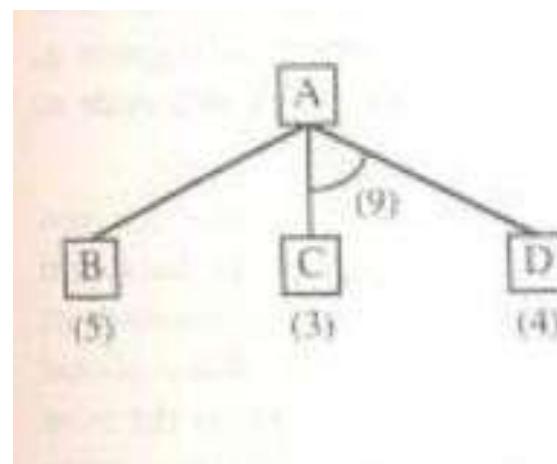
- **Advantages:**
  - It is an optimal algorithm.
  - If traverse according to the ordering of nodes. It can be used for both OR and AND graph.
- **Disadvantages:**
  - Sometimes for unsolvable nodes, it can't find the optimal path.

# Problem Reduction/Decomposition with AO\*

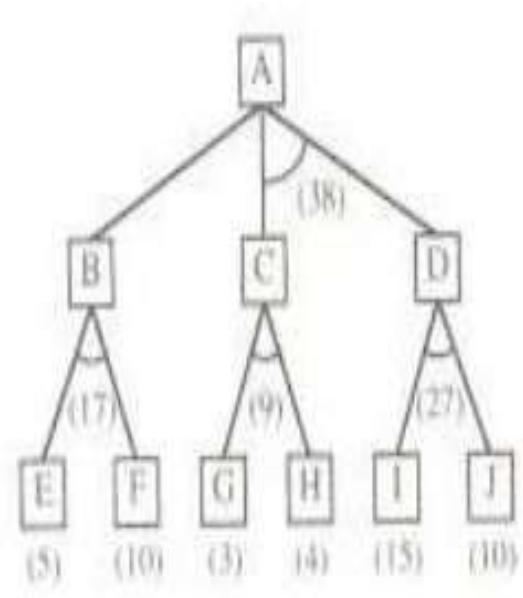
- When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND – OR trees are used for representing the solution.
- The decomposition of the problem or problem reduction generates AND arcs.
- One AND arc may point to any number of successor nodes, these must be solved so that the arc will rise to many arcs, indicating several possible solutions.
- Hence the graph is known as AND - OR instead of AND



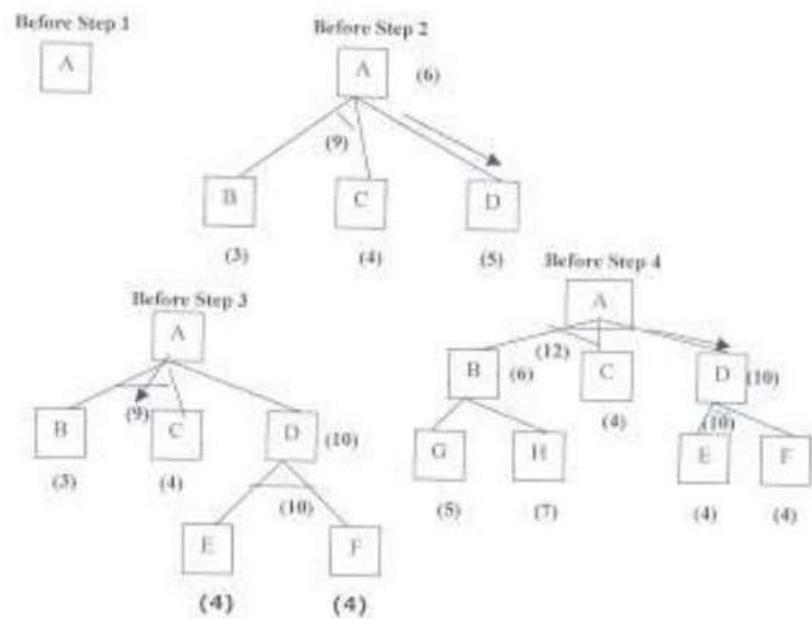
- An algorithm to find a solution in an AND - OR graph must handle AND area appropriately. A\* algorithm can not search AND - OR graphs efficiently
- In figure (a) the top node A has been expanded producing two area one leading to B and leading to C-D .
- the numbers at each node represent the value of  $f'$  at that node (cost of getting to the goal state from current state).
- For simplicity, it is assumed that every operation(i.e. applying a rule) has unit cost, i.e., each are with single successor will have a cost of 1 and each of its components.
- With the available information till now , it appears that C is the most promising node to expand since its  $f' = 3$  , the lowest but going through B would be better since to use C we must also use D' and the cost would be  $9(3+4+1+1)$ . Through B it would be  $6(5+1)$ .



- Thus the choice of the next node to expand depends not only on a value but also on whether that node is part of the current best path from the initial mode
- the node G appears to be the most promising node, with the least f' value. But G is not on the current best path, since to use G we must use GH with a cost of 9 and again this demands that arcs be used (with a cost of 27).
- The path from A through B, E-F is better with a total cost of  $(17+1=18)$ .
- Thus we can see that to search an AND-OR graph, the following three things must be done
  - 1. traverse the graph starting at the initial node and following the current best path, accumulate the set of nodes that are on the path and have not yet been expanded.
  - 2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and compute f' (cost of the remaining distance) for each of them
  - 3. Change the f' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path



- The initial node is expanded and D is Marked initially as promising node.
- D is expanded producing an AND arc E-F. f' value of D is updated to 10.
- Going backwards we can see that the AND arc B-C is better, it is now marked as current best path.
- B and C have to be expanded next. This process continues until a solution is found or all paths have led to dead ends, indicating that there is no solution.



- An A\* algorithm the path from one node to the other is always that of the lowest cost and it is independent of the paths through other nodes

# Problem decomposition into an and-or graph

- Example 2
- "The function of a financial advisor is to help the user decide whether to invest in a savings account, or the stock market, or both. The recommended investment depends on the investor's income and the current amount they have saved:

# Problem decomposition into an and-or graph

- Individuals with inadequate savings should always increase the amount saved as their first priority, regardless of income.
- Individuals with adequate savings and an adequate income should consider riskier but potentially more profitable investment in the stock market.
- Individuals with low income who already have adequate savings may want to consider splitting their surplus income between savings and stocks, to increase the cushion in savings while attempting to increase their income through stocks.
- The adequacy of both savings and income is determined by the number of dependants an individual must support.  
There must be at least £3000 in the bank for each dependant. An adequate income is a steady income, and it must supply at least £9000 per year, plus £2500 for each dependant."

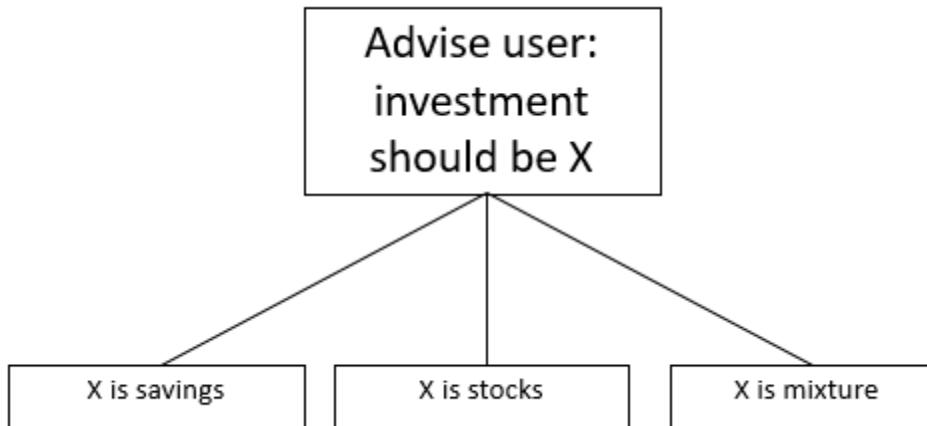
## Problem decomposition into an and-or graph

- How can we turn this information into an and-or graph?
- Step 1: decide what the ultimate advice that the system should provide is.

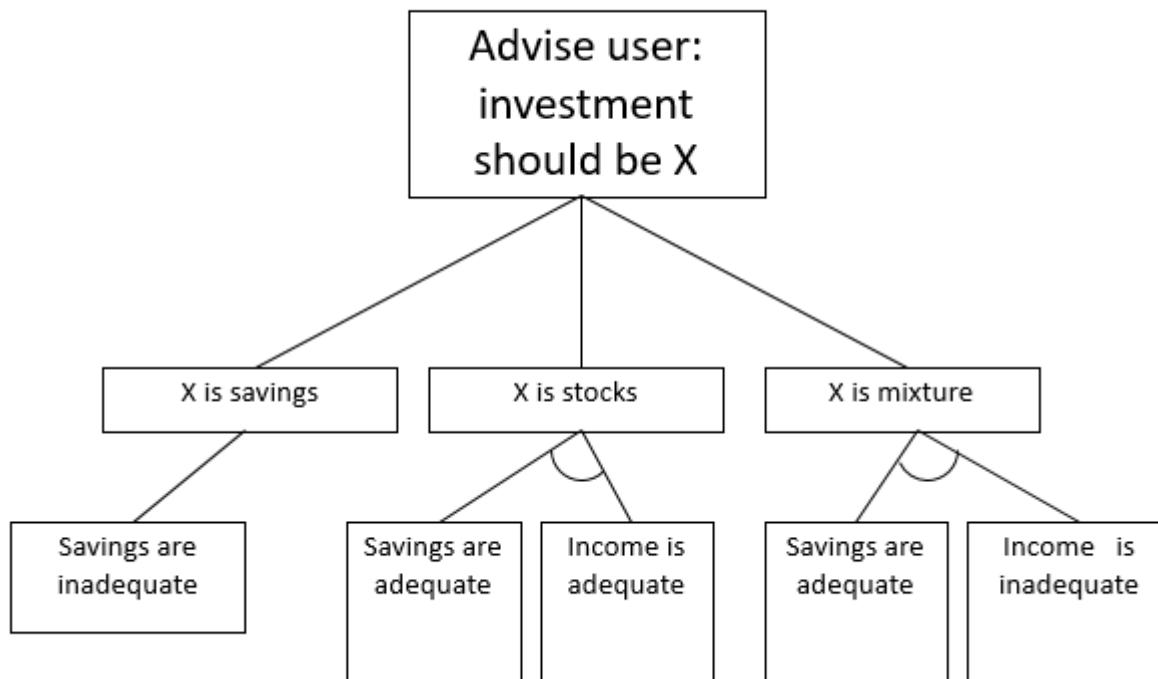
It's a statement along the lines of "The investment should be X", where X can be any one of several things.

Advise user:  
investment  
should be X

- Step 2: decide what sub-goals this goal can be split into.  
In this case, X can be one of three things: savings, stocks or a mixture.  
Add three sub-goals to the graph. Make sure the links indicate “or” rather than “and”.

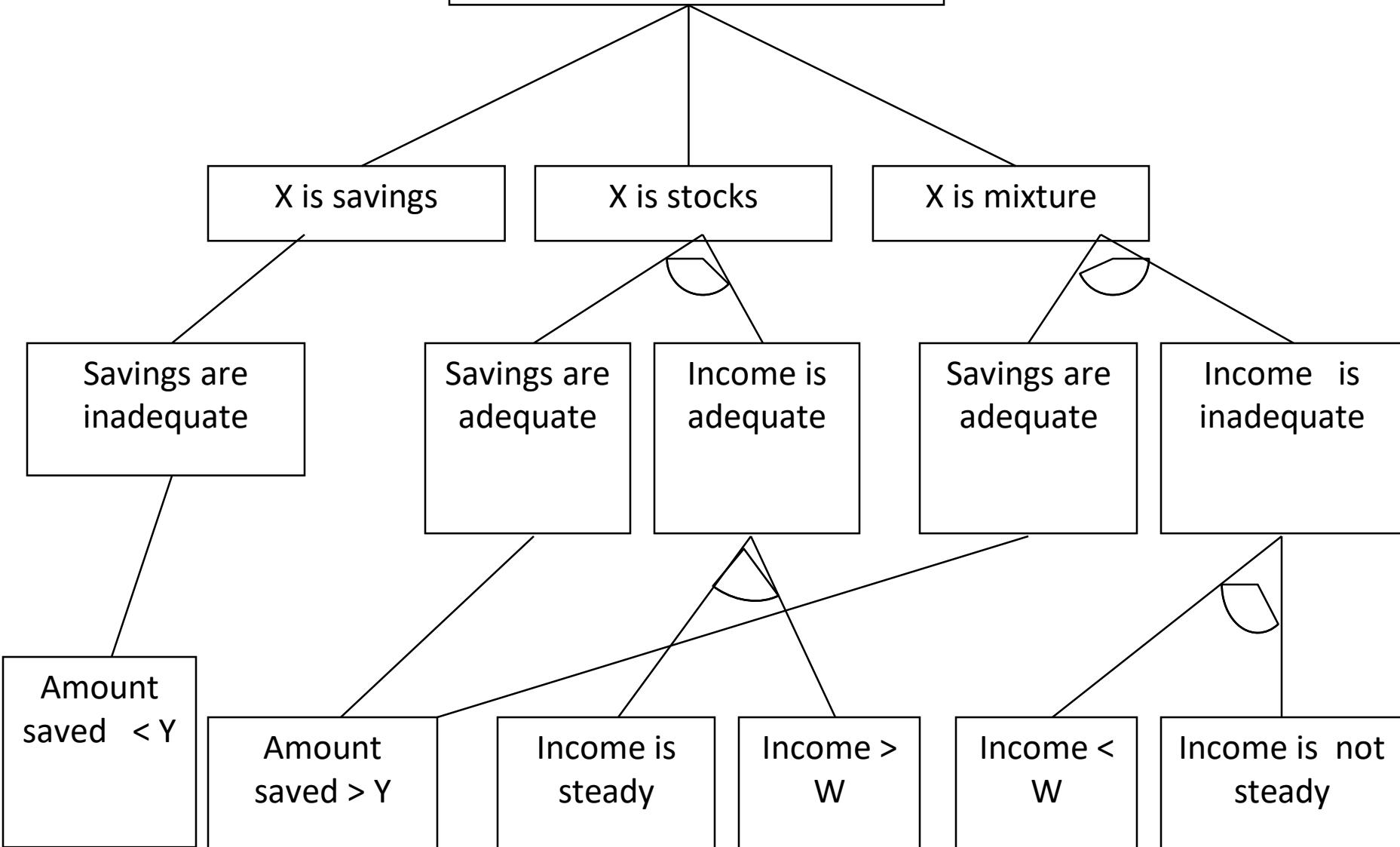


- Steps 3a, 3b and 3c: decide what sub-goals each of the goals at the bottom of the graph can be split into.
  - It's only true that "X is savings" if "savings are inadequate". That provides a subgoal under "X is savings"
  - It's only true that "X is stocks" if "savings are adequate" and "income is adequate". That provides two subgoals under "X is stocks" joined by "and" links.
  - Similarly, there are two subgoals under "X is mixture" joined by "and" links.



- The next steps (4a,4b,4c,4d & 4e) mainly involve deciding whether something's big enough.
  - Step 4a: savings are only inadequate if they are smaller than a certain figure (let's call it Y).
  - Step 4b: savings are only adequate if they are bigger than this figure (Y).
  - Step 4c: income is only adequate if it is bigger than a certain figure (let's call it W), and also steady.
  - Step 4d is the same as 4b. Step 4e is like 4c, but “inadequate”, “smaller” and “not steady”.

Advise user: investment should  
be X



Now we need a box in which the value of Y & W is calculated:

Y is Z times 3000

W is 9000 plus 2500 times Z

Z is the number of dependants, so we need a box in which this value is obtained:

Client has Z dependants

the bottom layers of the graph in the same way as we've added all the others:

advise user:  
investment  
should be X

X is savings

X is stocks

X is mixture

savings  
inadequate

savings  
adequate

income  
adequate

savings  
adequate

income  
inadequate

amount saved  
 $< Y$

amount saved  
 $> Y$

income  $> W$

income  $< W$

$Y = Z * 3000$

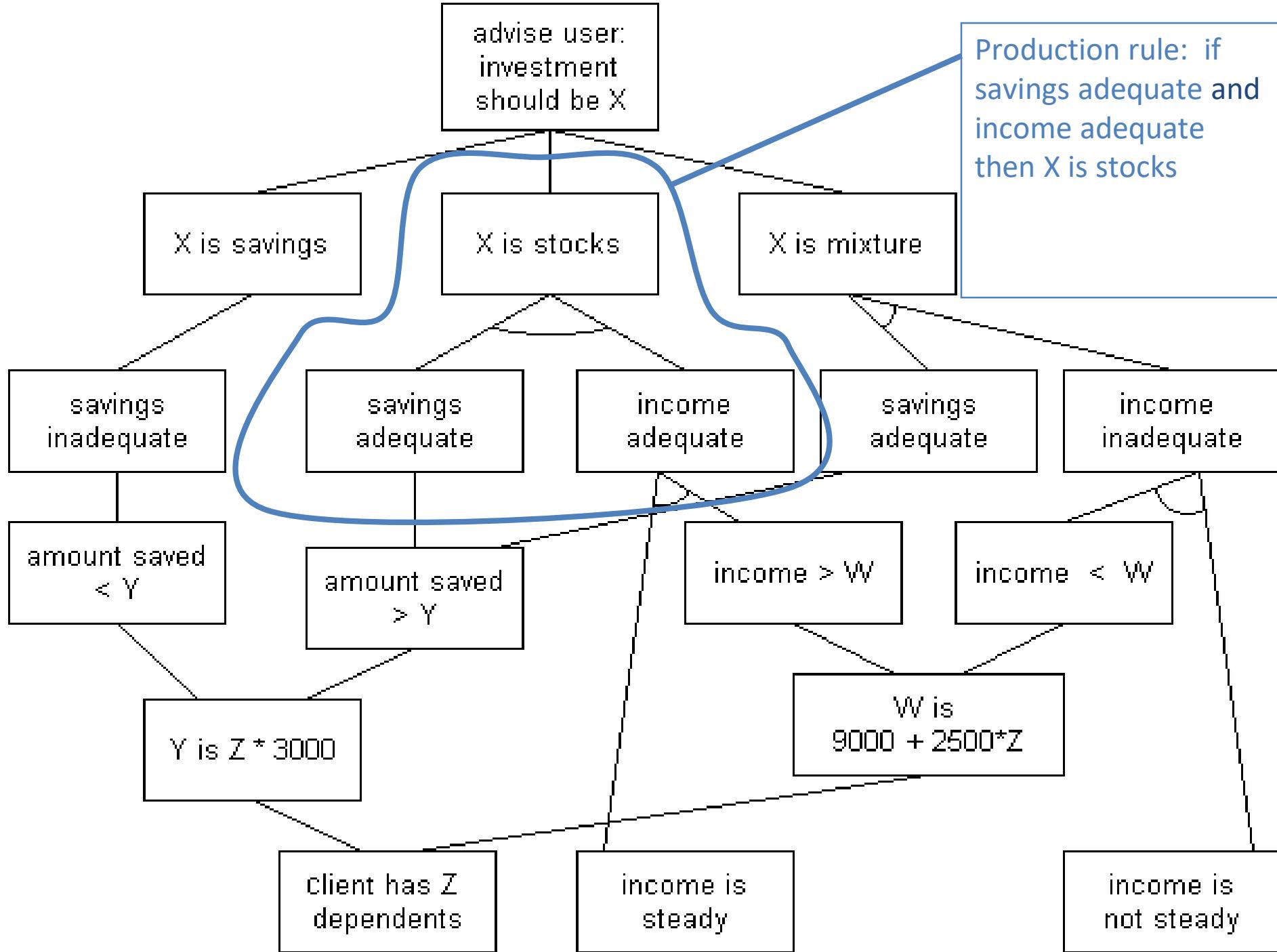
$W = 9000 + 2500 * Z$

client has  $Z$   
dependents

income is  
steady

income is  
not steady

Production rule: if savings adequate and income adequate then X is stocks



advise user:  
investment  
should be X

Production rule: if income  
< W and income is not  
steady  
then income is  
inadequate

X is savings

X is stocks

X is mixture

savings  
inadequate

savings  
adequate

income  
adequate

savings  
adequate

income  
inadequate

amount saved  
< Y

amount saved  
> Y

income > W

income < W

Y is Z \* 3000

W is  
 $9000 + 2500 \cdot Z$

client has Z  
dependents

income is  
steady

income is  
not steady

# **Local Search Algorithm**

## MORE SEARCH ALGORITHMS

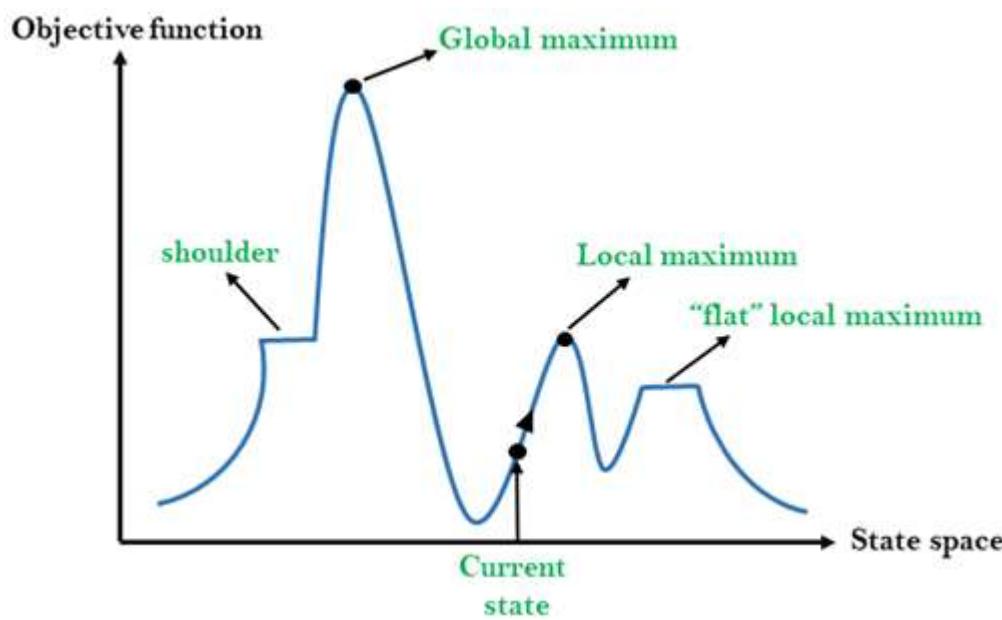
- Local search: Only achieving a *goal* is needed,
  - not the path to the goal
- Possibilities:
  - Non-determinism in state space (graph)
  - Partially-observable state space
  - On-line search (full search space is not available)

# Local search algorithms and optimization

- Systematic search algorithms to find the goal
- Local search algorithms
  - the path to the goal is irrelevant, e.g.,  $n$ -queens problem
  - state space = set of “complete” configurations
  - keep a single “current” state and try to improve it, e.g., move to its neighbors
  - Key advantages:
    - use very little (constant) memory
    - find reasonable solutions in large or infinite (continuous) state spaces
  - Optimization problem (pure)-
    - to find the best state (optimal configuration ) based on an objective function, e.g. reproductive fitness – no goal test and path cost
- Instead of considering the whole state space, consider only the current state
- Limits necessary memory; paths not retained
- Amenable to large or continuous (infinite) state spaces where exhaustive algorithms aren't possible
- Local search algorithms can't backtrack!

# State-space Diagram/landscape for Hill Climbing

Elevation = the value of the objective function or heuristic cost function



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.

# 1. Hill-Climbing Search

What we learn hill-climbing is  
Usually like



What we think hill-  
climbing looks like



# Hill-Climbing Algorithm

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- 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.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

# Features of Hill Climbing:

- Following are some main features of Hill Climbing Algorithm:
- **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.

# Hill-climbing search

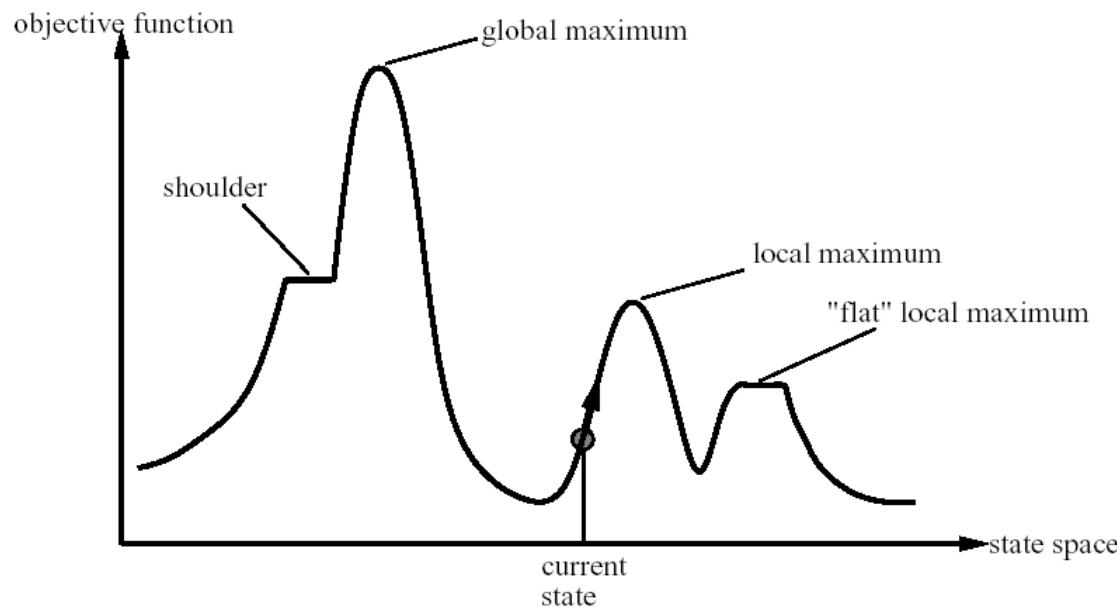
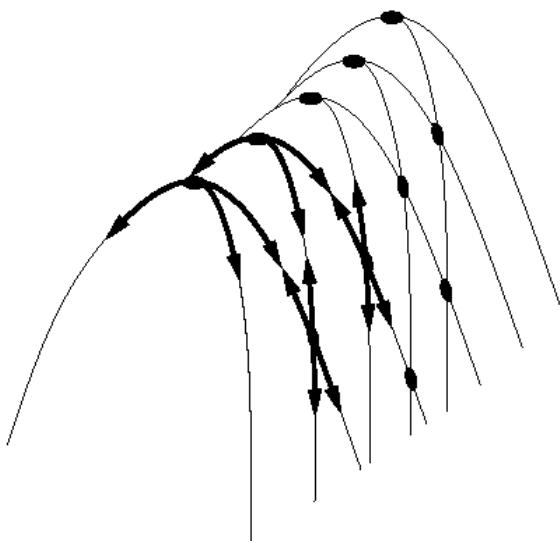
- moves in the direction of increasing value until a “peak”
  - current node data structure only records the state and its objective function
  - neither remember the history nor look beyond the immediate neighbors

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
inputs: problem, a problem
local variables: current, a node
                  neighbor, a node
current  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
loop do
    neighbor  $\leftarrow$  a highest-valued successor of current
    if VALUE[neighbor] < VALUE[current] then return STATE[current]
    current  $\leftarrow$  neighbor
end
```

- Algorithm for Simple Hill Climbing:
- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
  - If it is goal state, then return success and quit.
  - Else if it is better than the current state then assign new state as a current state.
  - Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

# Hill-climbing search – greedy local search

- Hill climbing, the greedy local search, often gets stuck
  - **Local maxima**: a **peak** that is higher than each of its neighboring states, but lower than the global maximum
  - **Ridges**: a sequence of local maxima that is difficult to navigate



- **Plateau**: a **flat** area of the state space landscape
  - a **flat local maximum**: no uphill exit exists
  - a **shoulder**: possible to make progress

# Drawbacks of hill climbing

**1. Local Maximum:** A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

**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.

**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.

**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-plateau region.

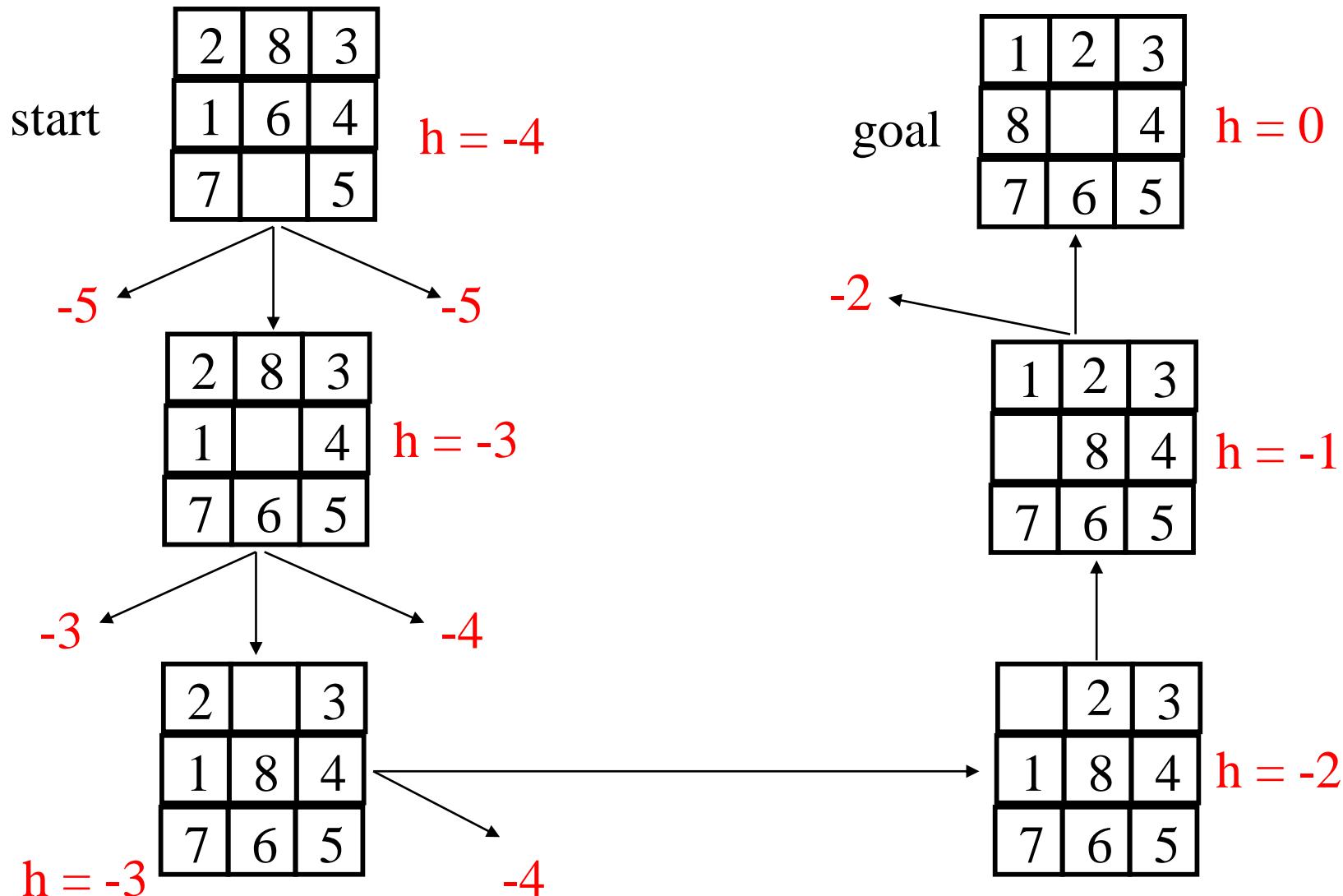
**3. Ridges:** A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

**Solution:** With the use of bidirectional search, or by moving in different directions, we can improve this problem.

# Hill-climbing search

- Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state.** It only checks its one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:
  - Less time consuming
  - Less optimal solution and the solution is not guaranteed

# Hill climbing example



$$f(n) = -( \text{number of tiles out of place} )$$

# Hill Climbing Search

- Variants of Hill climbing
  - Stochastic Hill Climbing
  - First Choice Hill Climbing
  - Random restart hill climbing
  - Evolutionary Hill Climbing

## – Stochastic Hill Climbing

- Basic hill climbing selects always up hill moves,
- This selects random from available uphill moves
- This help in addressing issues with simple hill climbing like ridge.

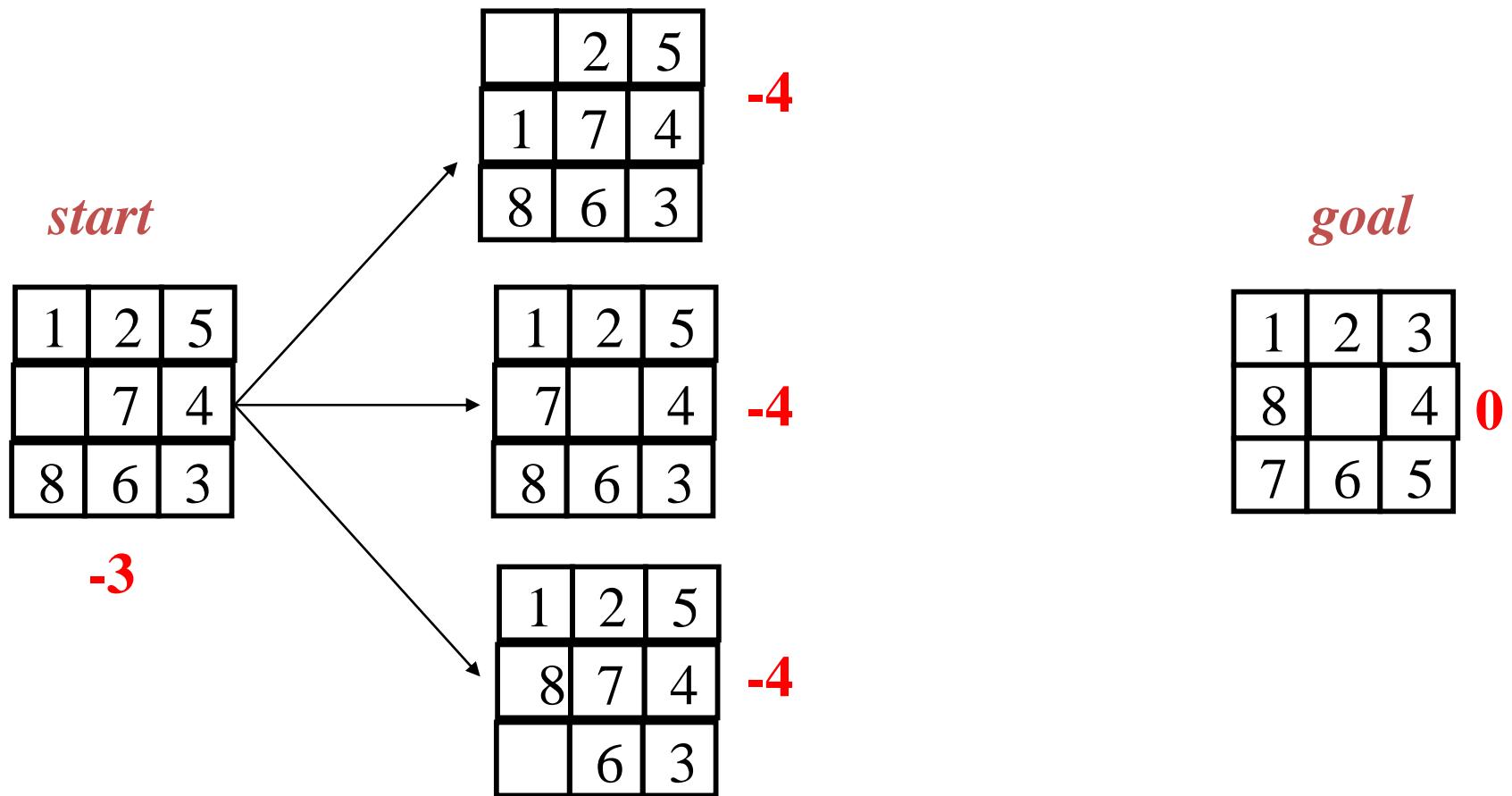
## – Random restart hill climbing

- It tries to overcome other problem with hill climbing
- Initial state is randomly generated
- Reaches to a position from where no progressive state is possible
- Local maxima problem is handled by RRHC

## – Evolutionary Hill Climbing

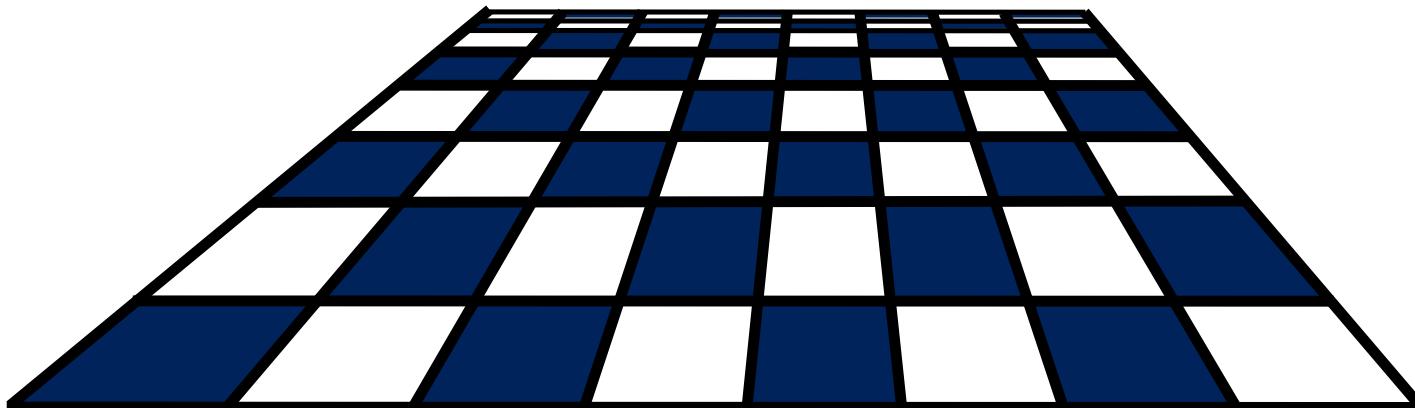
- Performs random mutations
- Genetic algorithm base search

# Example of a local optimum



# The N-Queens Problem

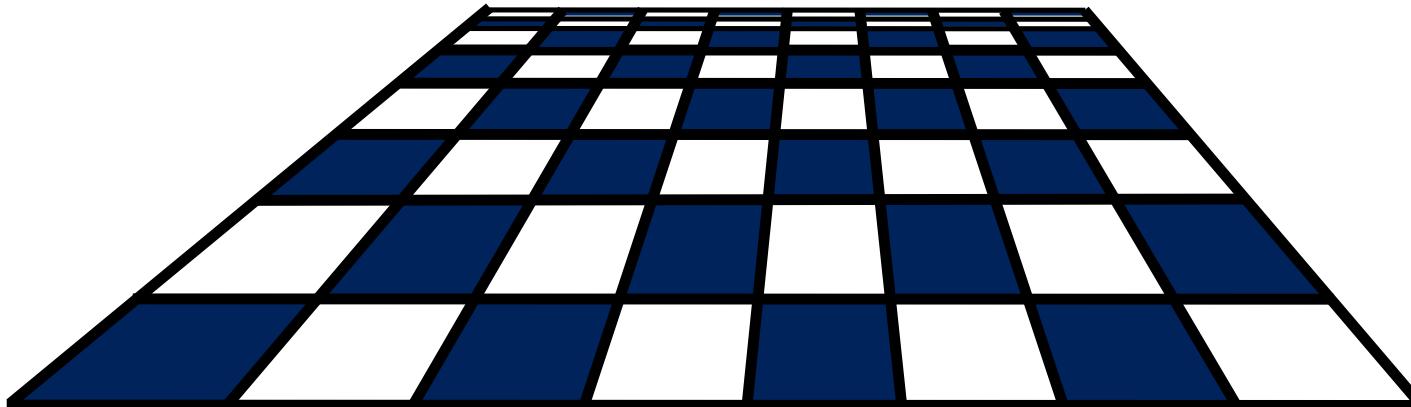
- Suppose you have 8 chess queens...
- ...and a chess board



# The N-Queens Problem

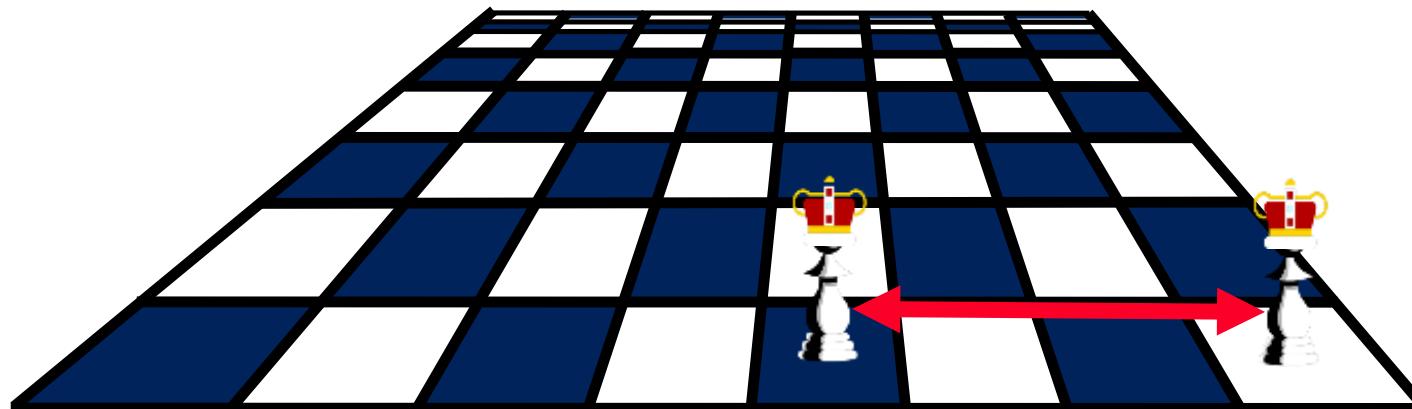
*Can the queens be placed on the board so that no two queens are attacking each other*

?



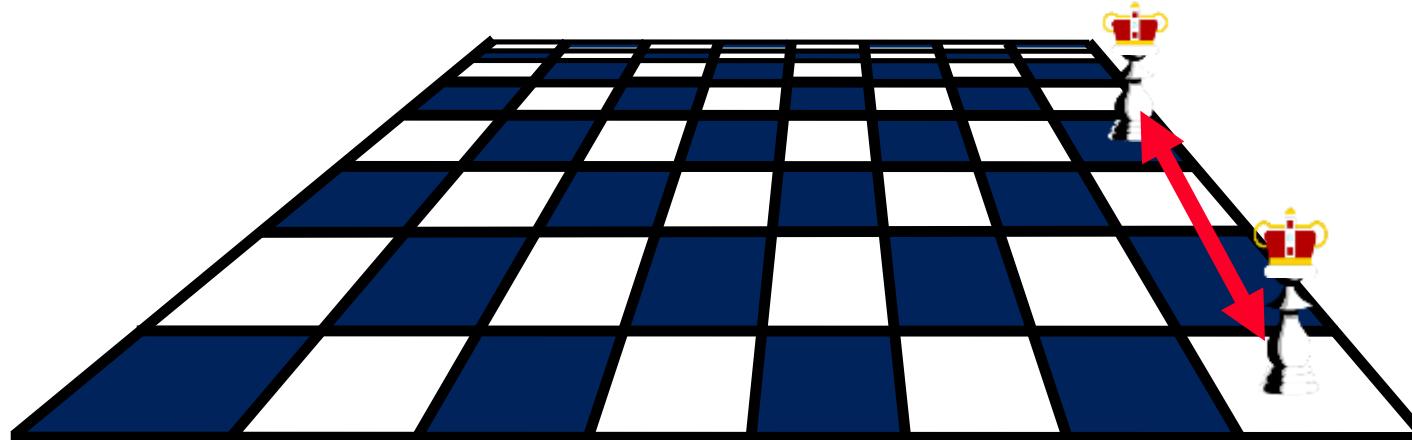
# The N-Queens Problem

Two queens are not allowed in the same row...



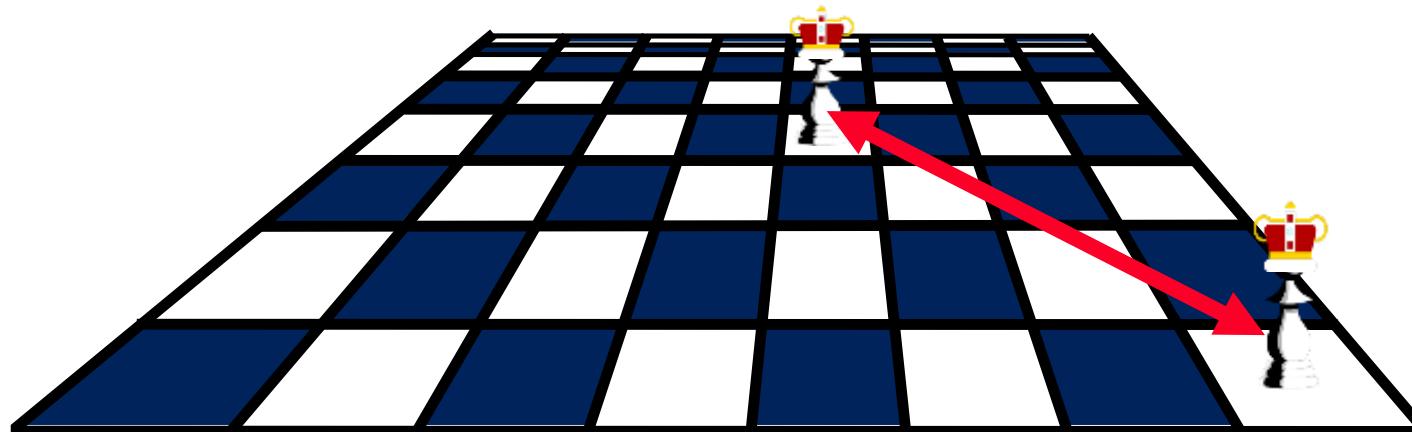
# The N-Queens Problem

Two queens are not allowed in the same row, or in the same column...



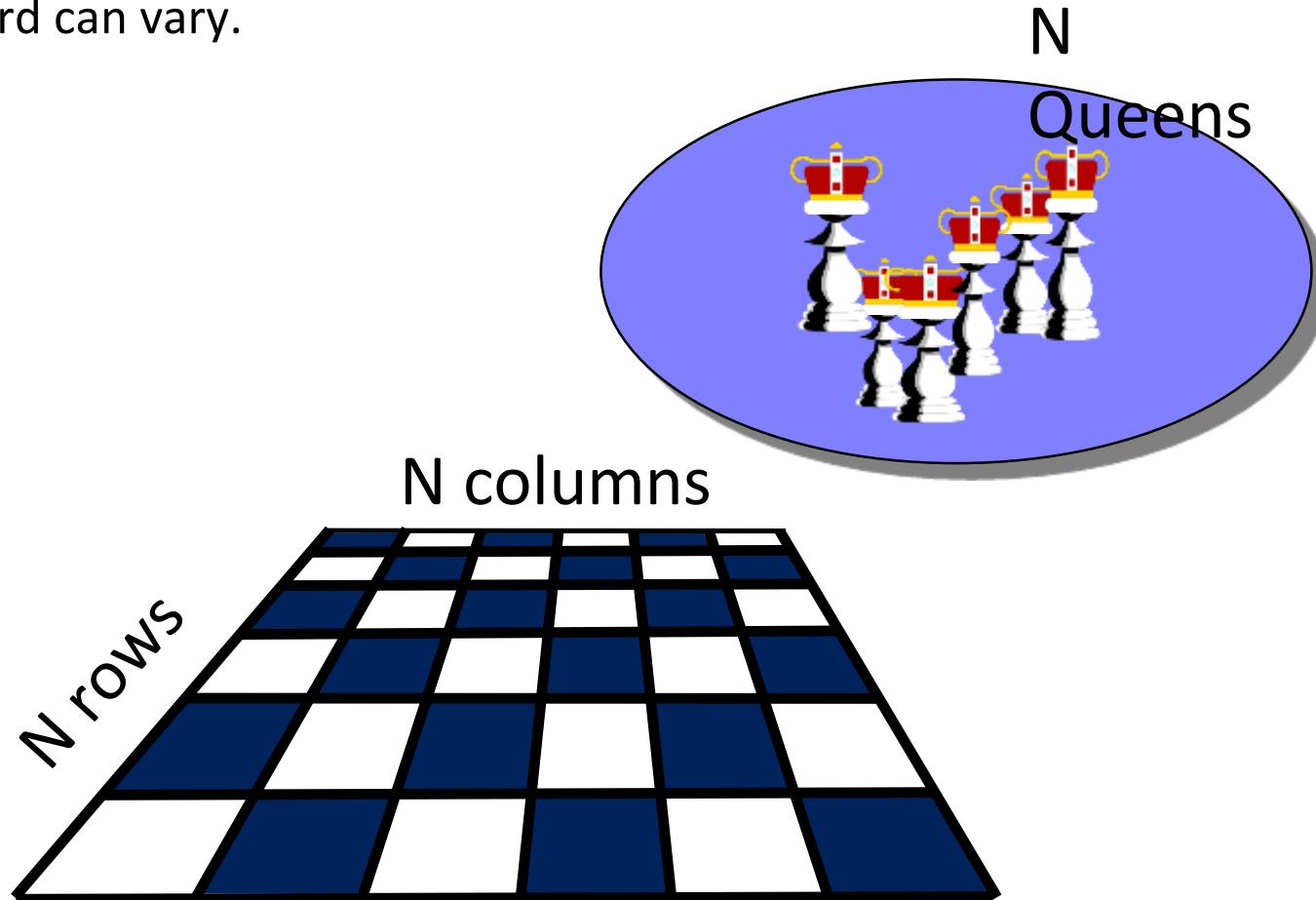
# The N-Queens Problem

Two queens are not allowed in the same row, or in the same column, or along the same diagonal.



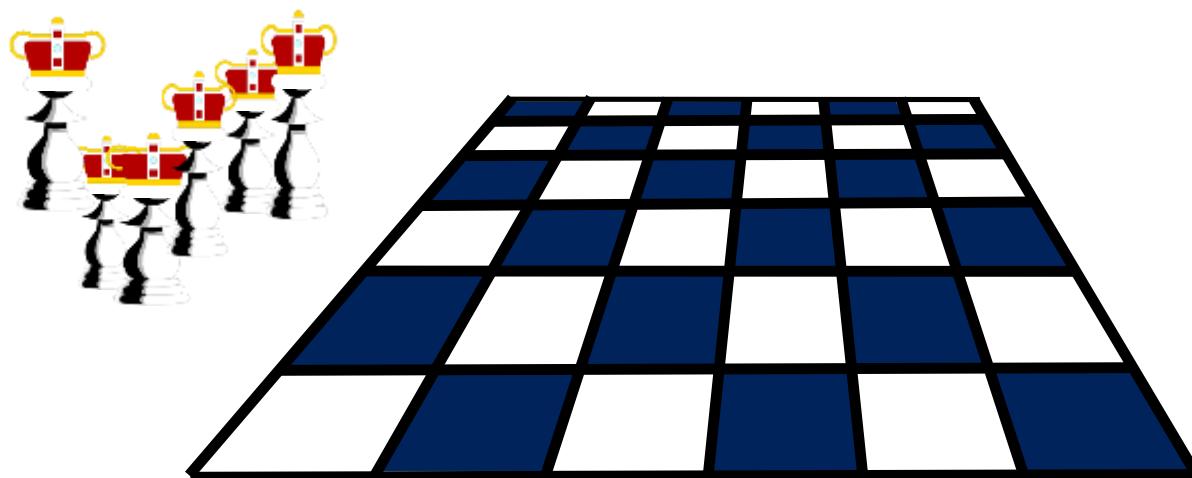
# The N-Queens Problem

The number of queens, and the size of the board can vary.



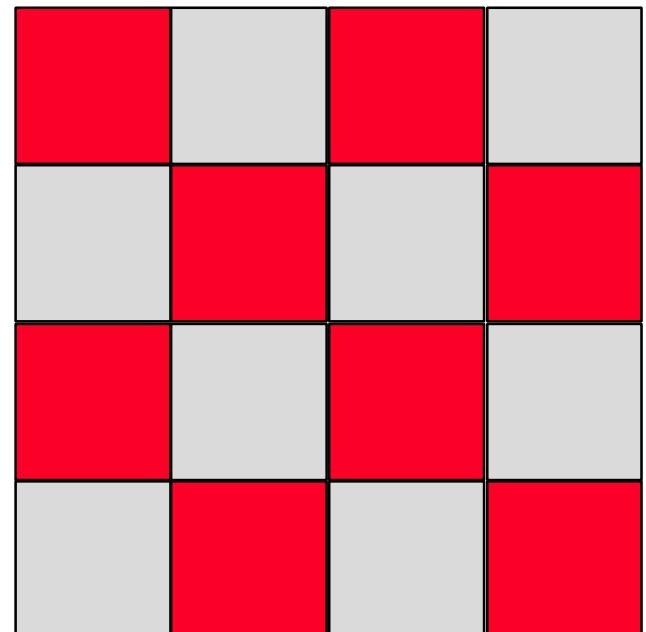
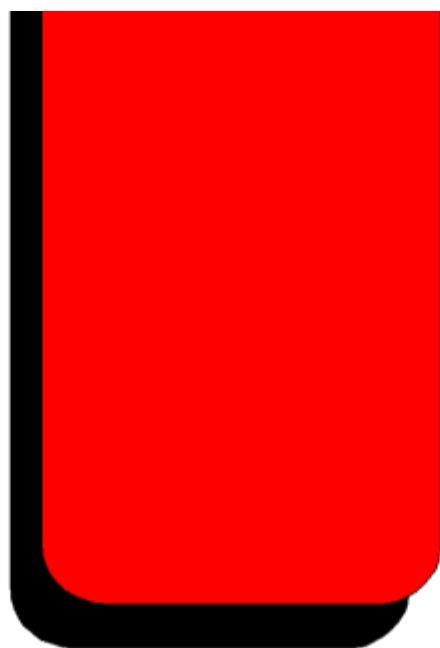
# The N-Queens Problem

We will write a program which tries to find a way to place N queens on an  $N \times N$  chess board.



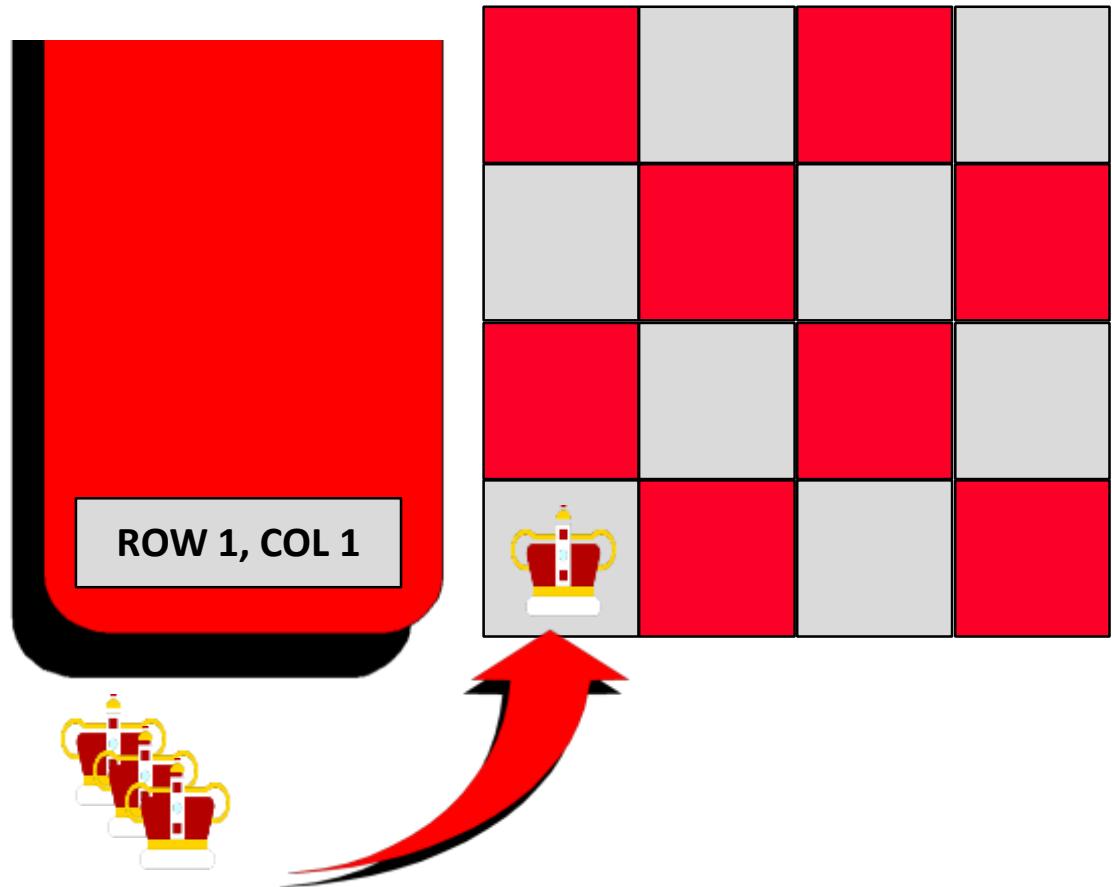
# How the program works

The program uses a stack  
to keep track of where  
each queen is placed.



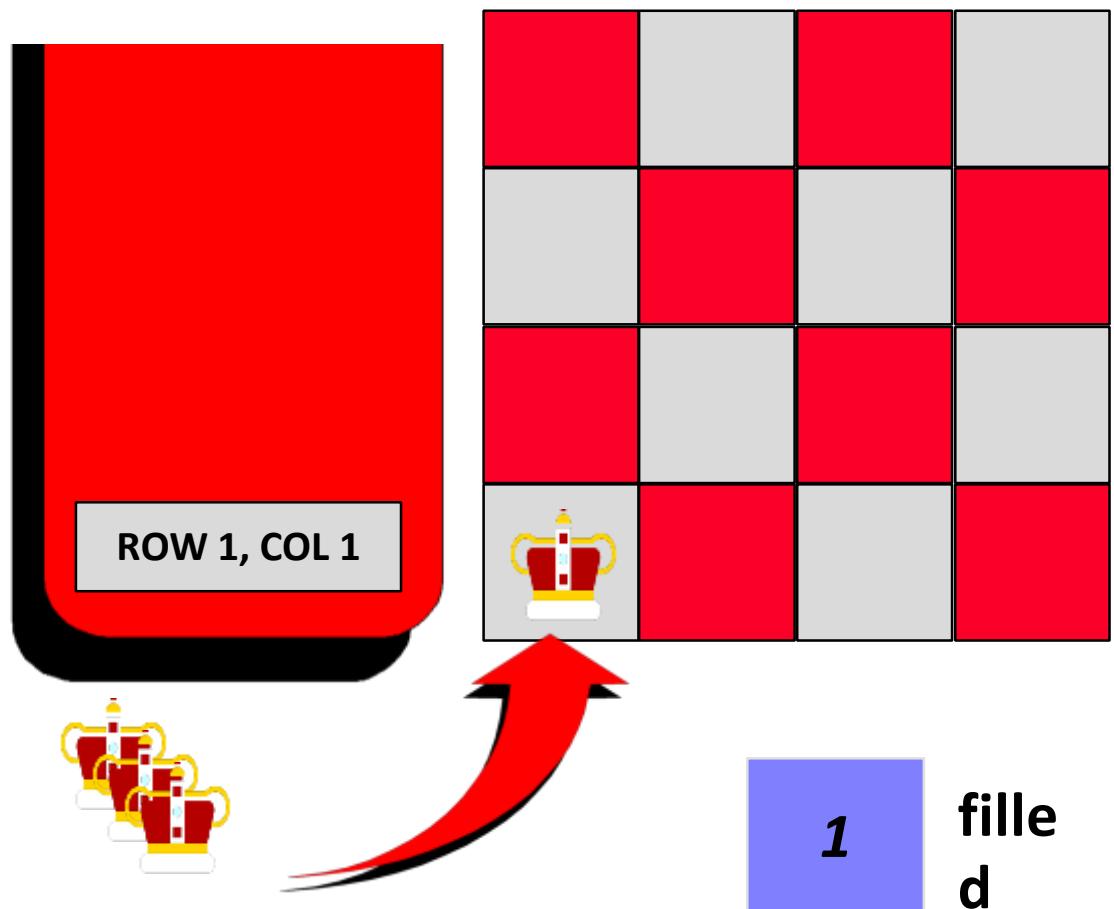
# How the program works

Each time the program decides to place a queen on the board, the position of the new queen is stored in a record which is placed in the stack.



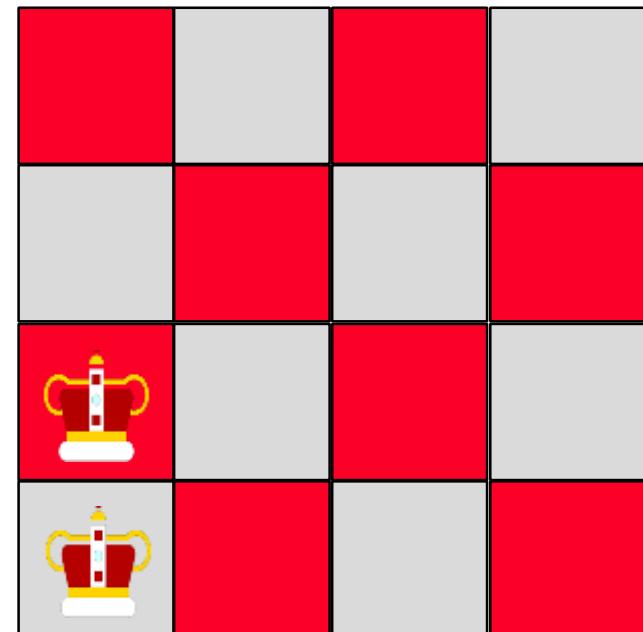
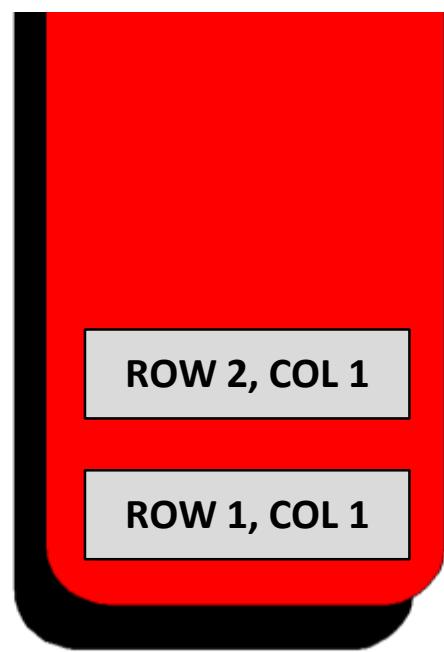
# How the program works

We also have an integer variable to keep track of how many rows have been filled so far.



# How the program works

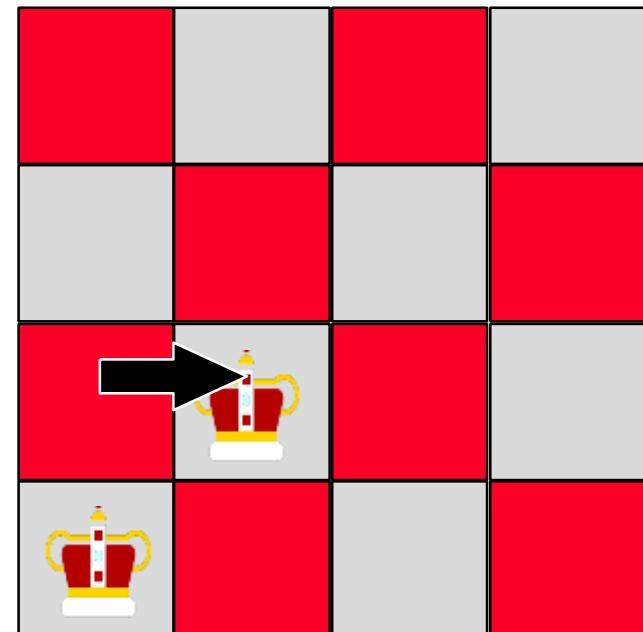
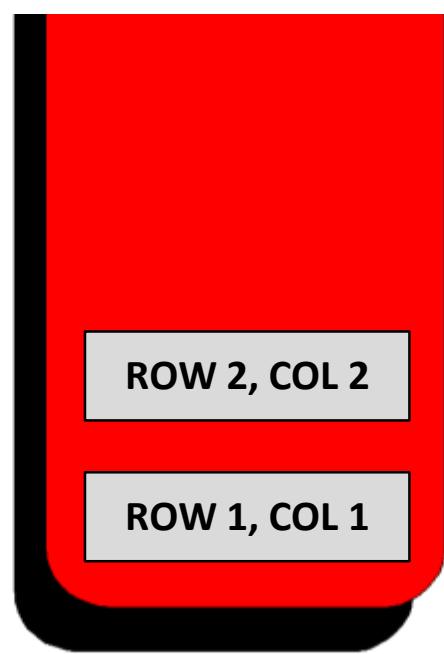
Each time we try to place a new queen in the next row, we start by placing the queen in the first column...



1  
filled

# How the program works

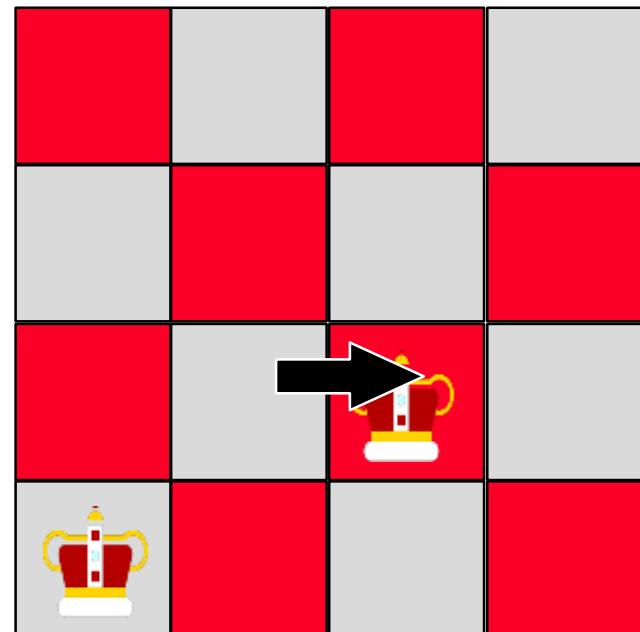
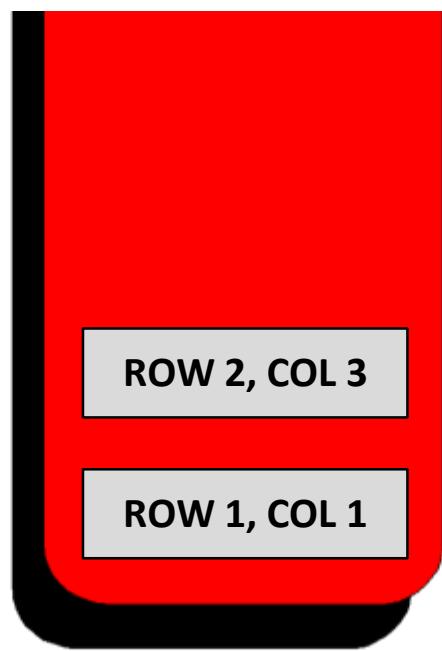
...if there is a conflict with another queen, then we shift the new queen to the next column.



1 filled

# How the program works

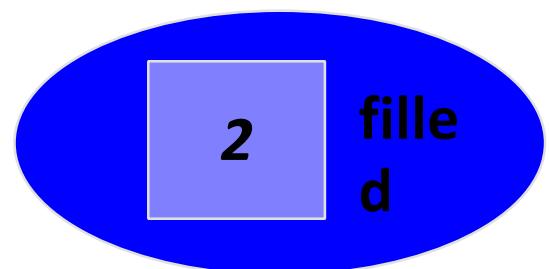
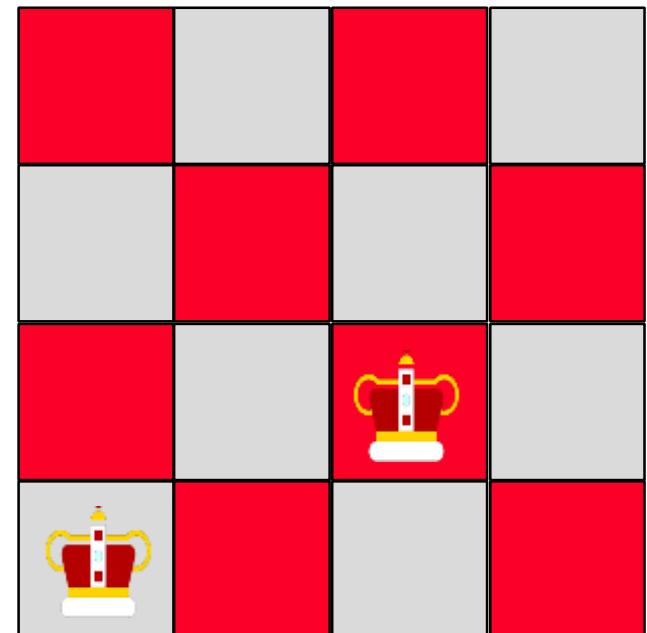
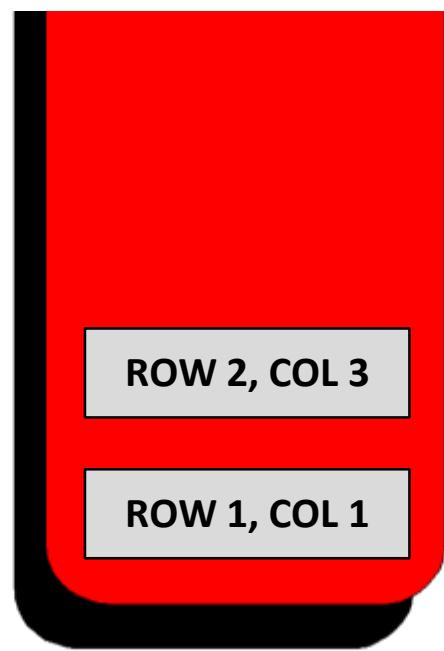
If another conflict occurs, the queen is shifted rightward again.



1 filled

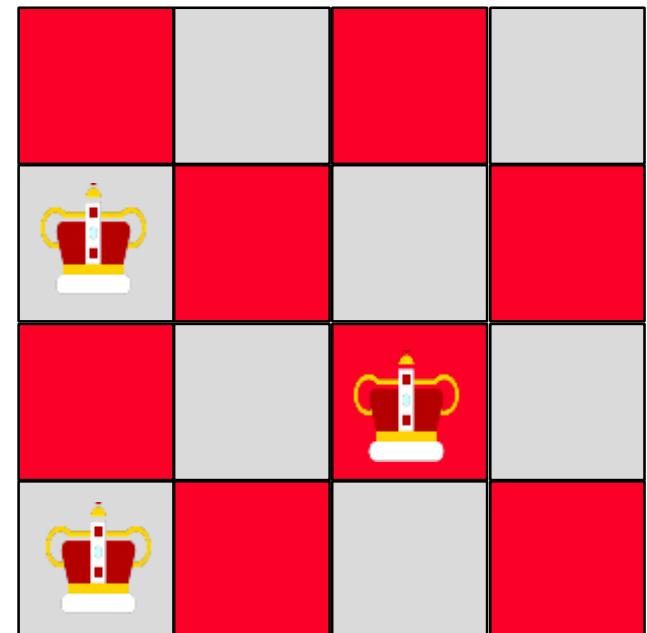
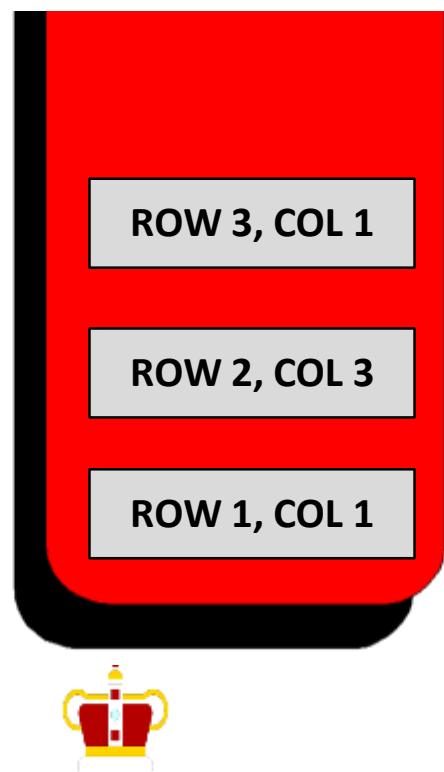
# How the program works

When there are no conflicts, we stop and add one to the value of filled.



# How the program works

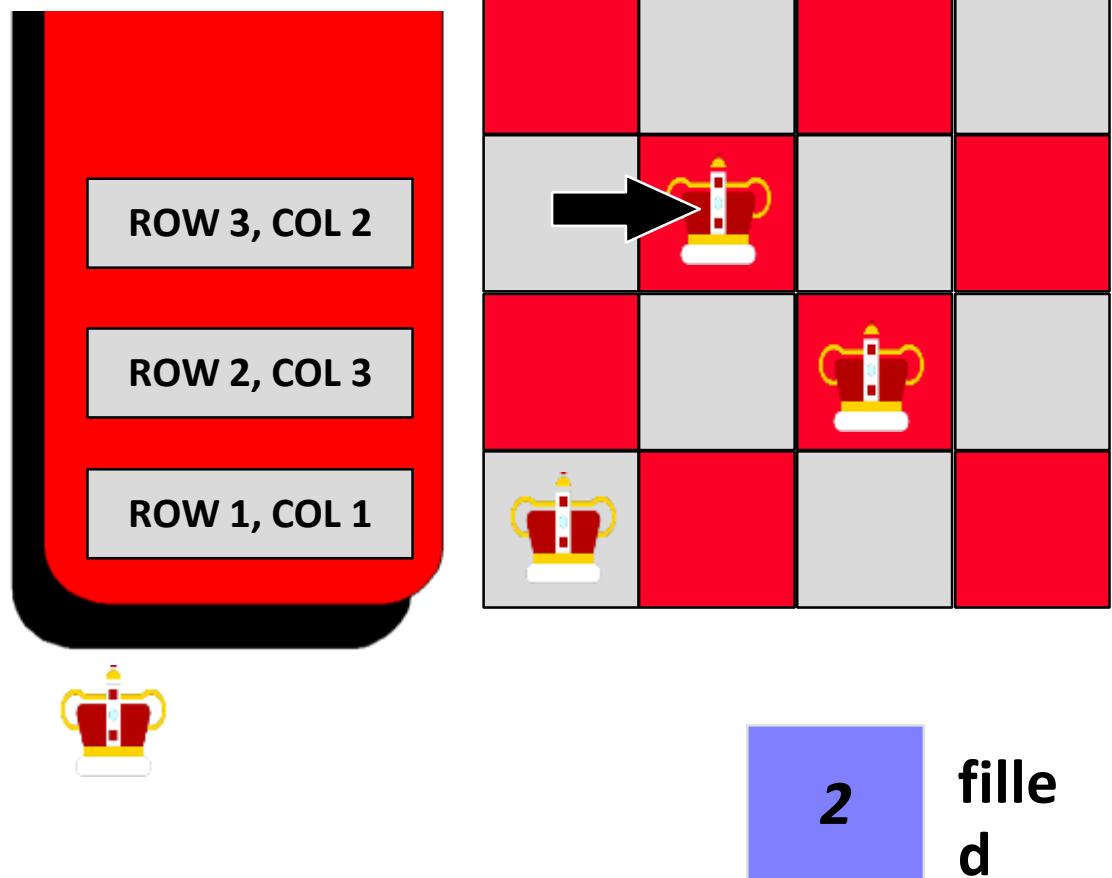
Let's look at the third row. The first position we try has a conflict...



2 filled

# How the program works

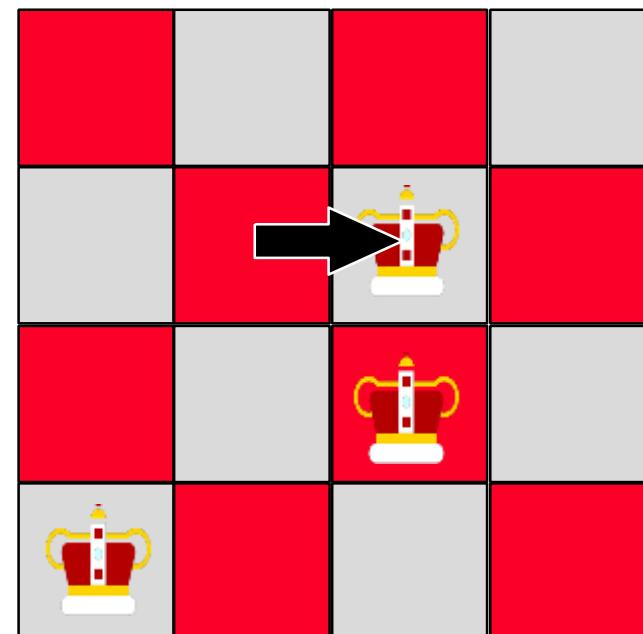
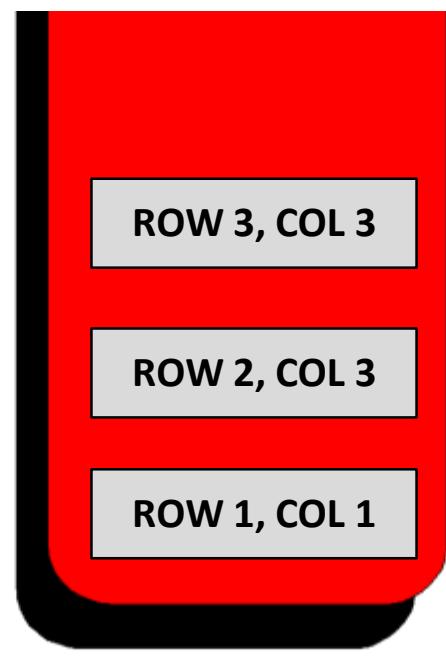
...so we shift to column  
2. But another conflict  
arises...



# How the program works

...and we shift to the third column.

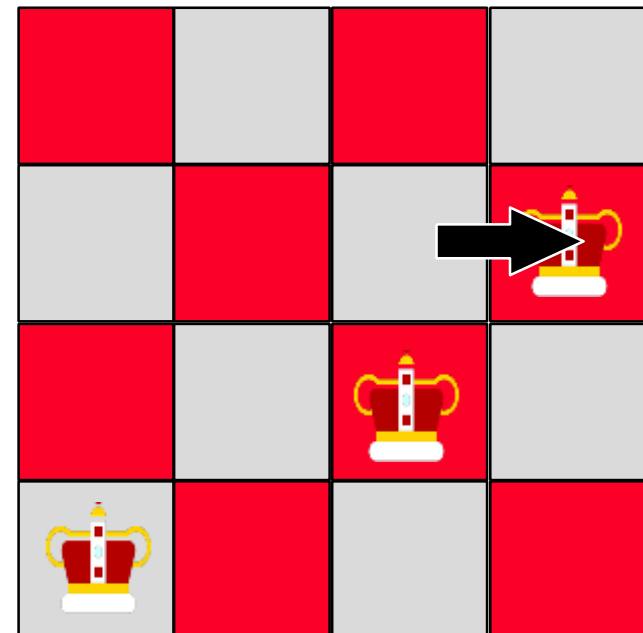
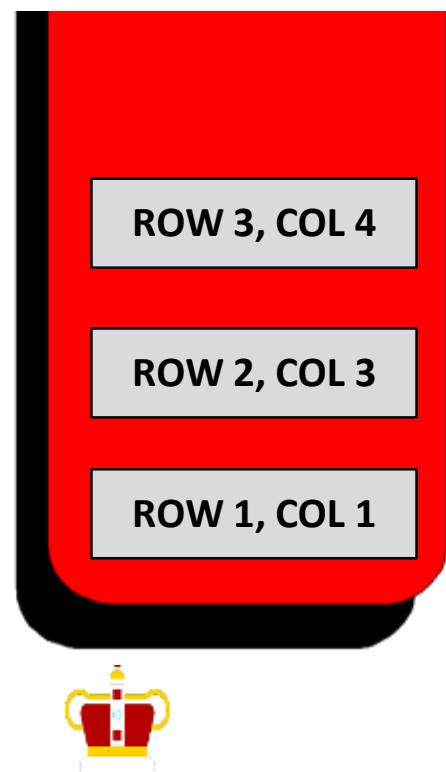
Yet another conflict arises...



2 filled

# How the program works

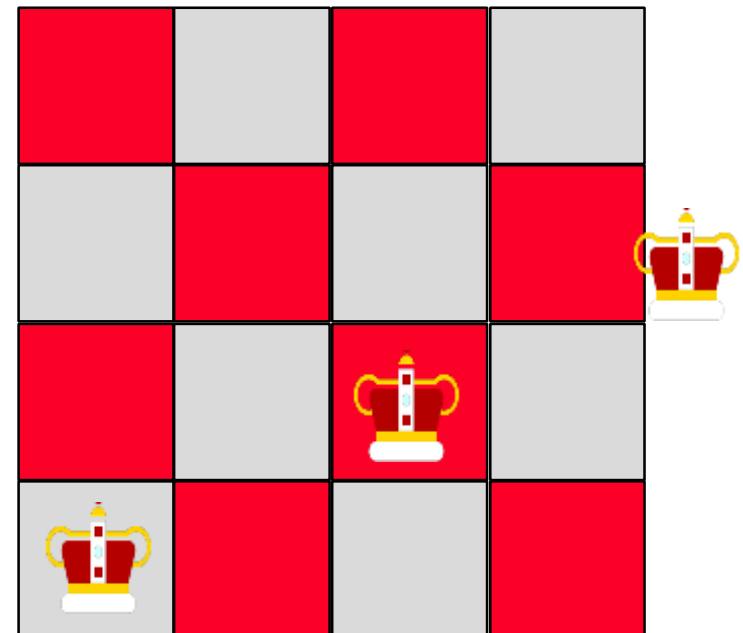
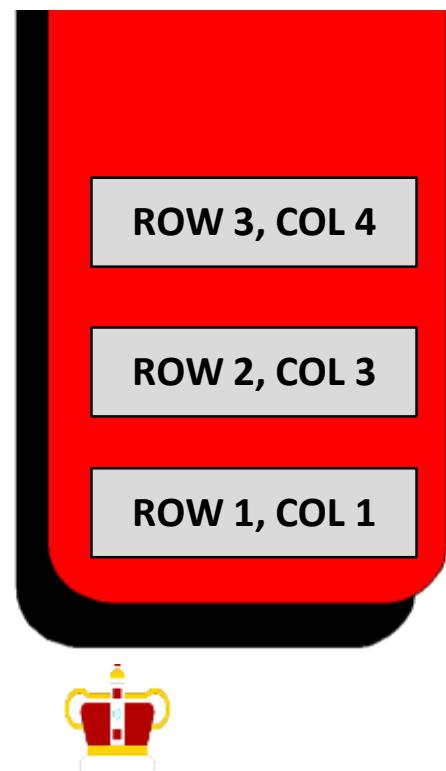
...and we shift to column 4. There's still a conflict in column 4, so we try to shift rightward again...



2 filled

# How the program works

...but there's nowhere  
else to go.



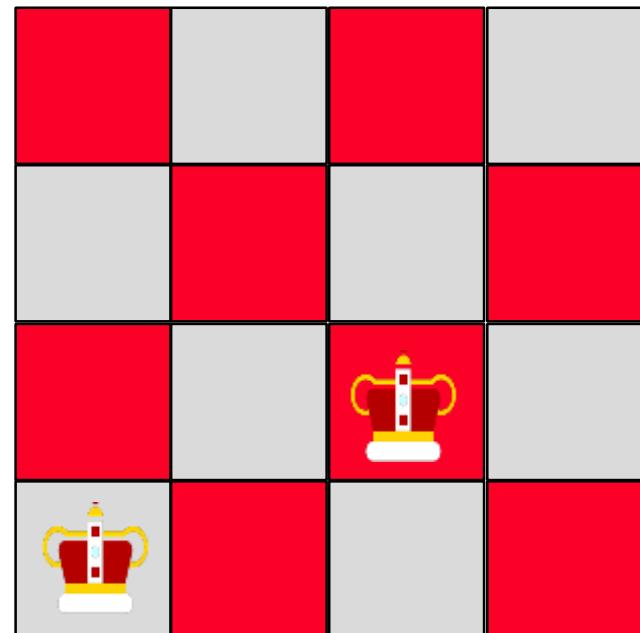
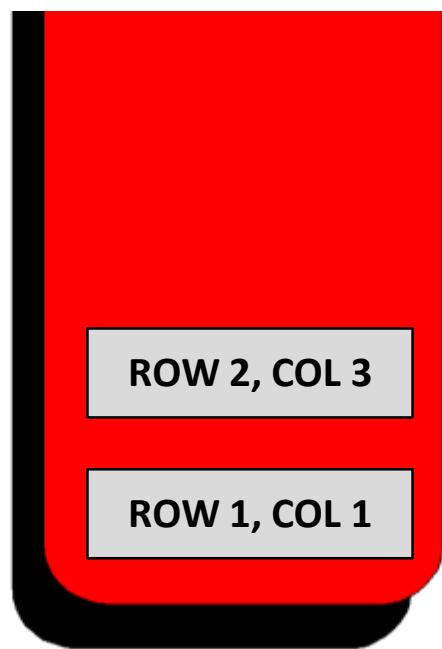
2 filled

# How the program works

When we run out of

room in a row:

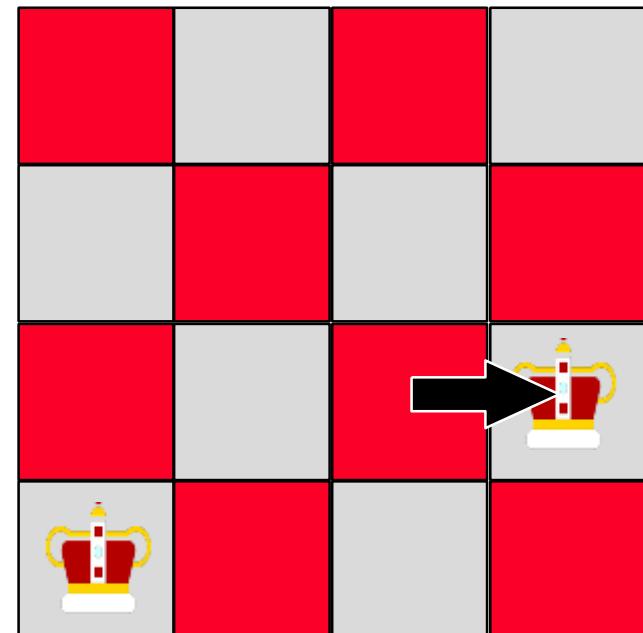
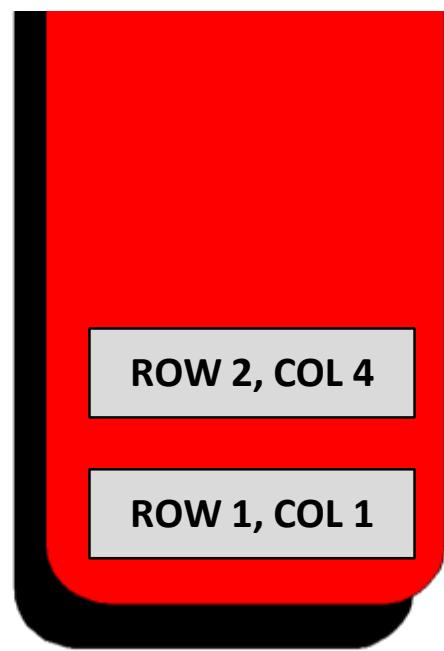
- pop the stack,
- reduce filled by 1
- and continue working on the previous row.



1  
filled

# How the program works

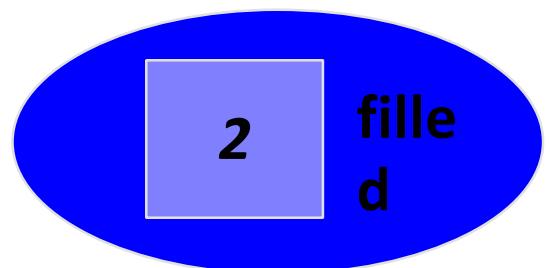
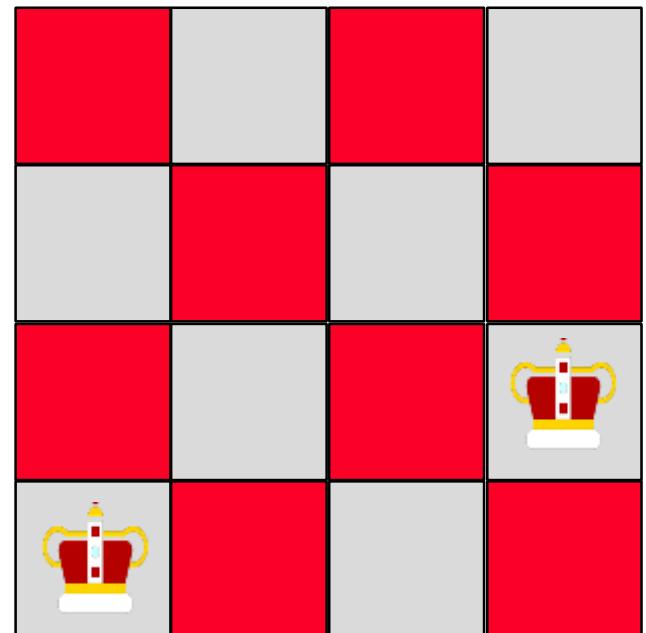
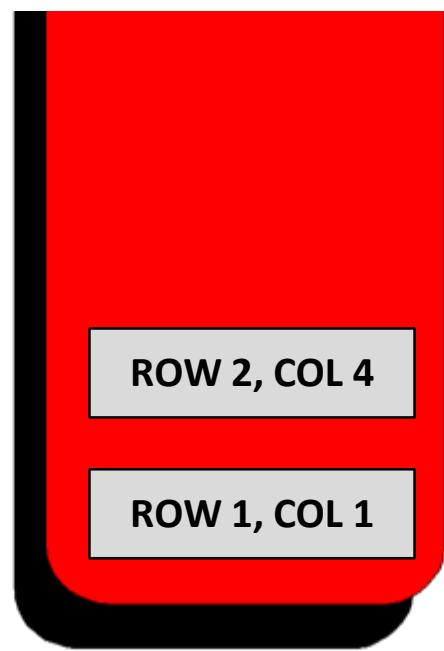
Now we continue working on row 2, shifting the queen to the right.



1  
filled

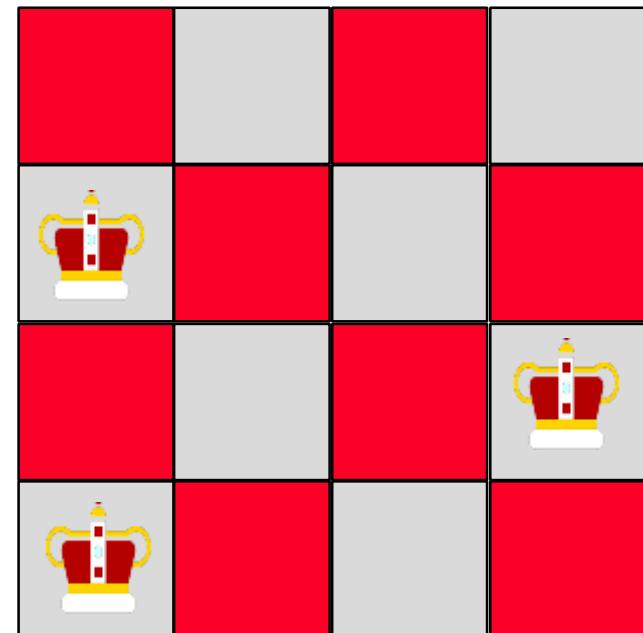
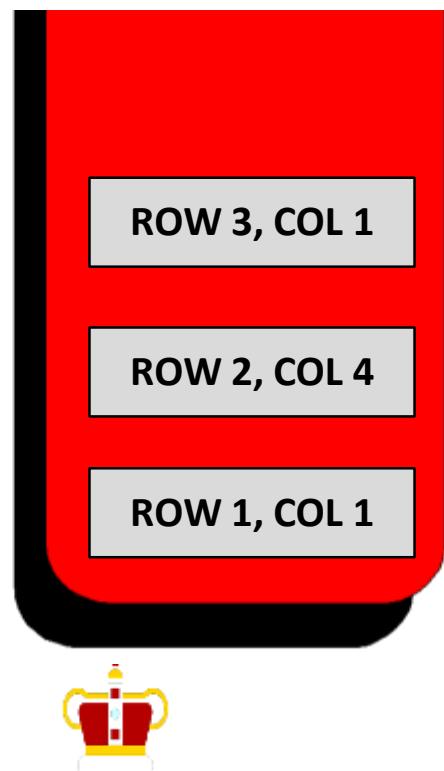
# How the program works

This position has no conflicts, so we can increase filled by 1, and move to row 3.



# How the program works

In row 3, we start again at the first column.

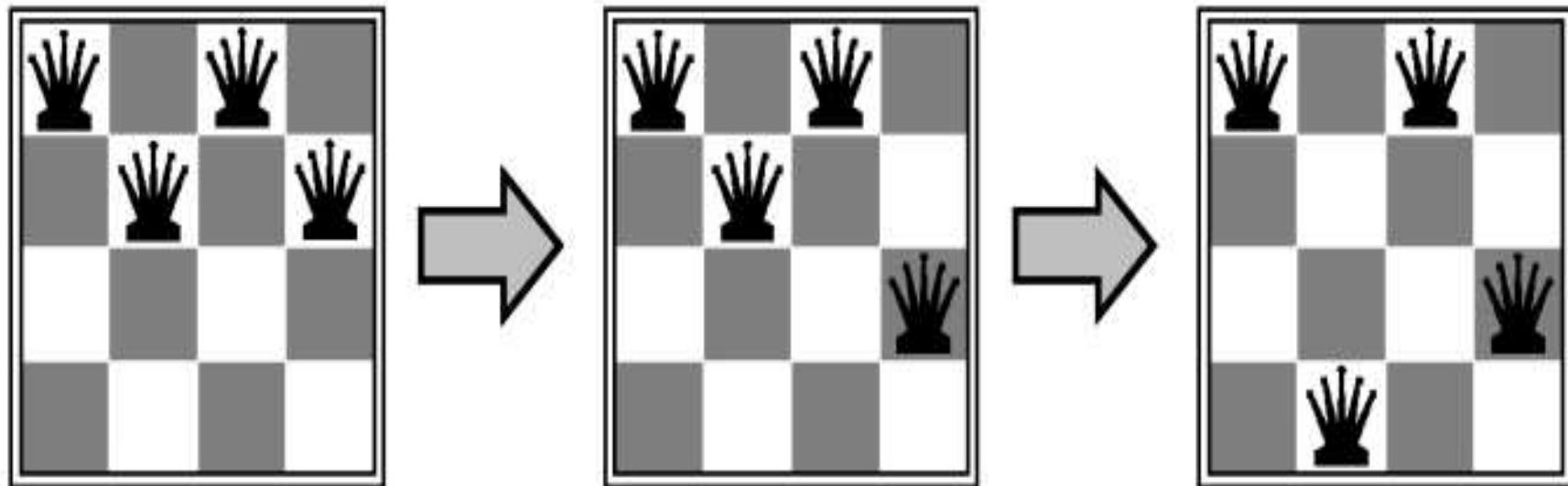


2  
filled

## Local search – example

Put  $n$  queens on an  $n \times n$  board with no two queens on the same row, column, or diagonal

Move a queen to reduce number of conflicts

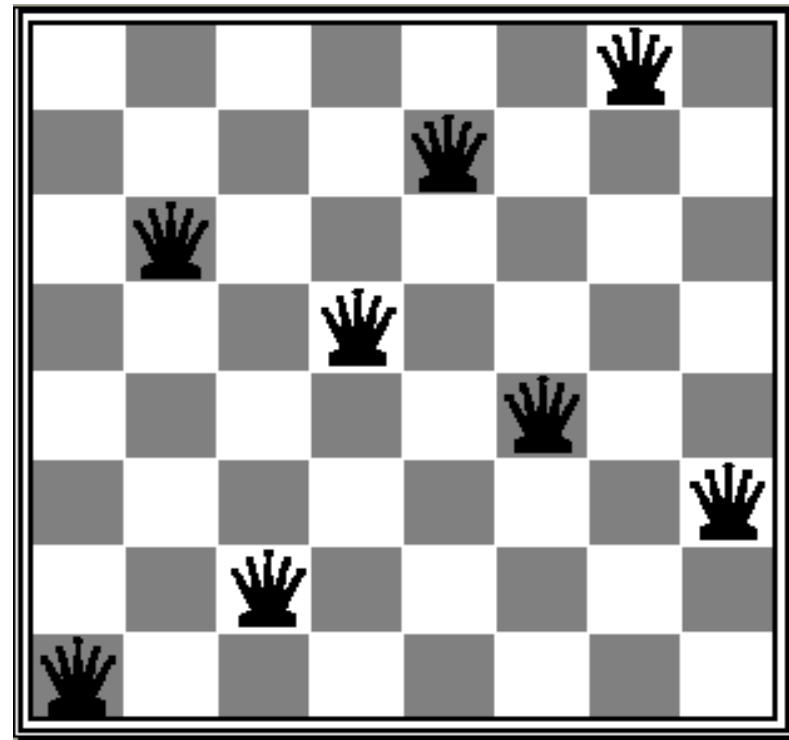


# Hill-climbing search - example

- complete-state formulation for 8-queens
  - successor function returns all possible states generated by moving a single queen to another square in the same column ( $8 \times 7 = 56$  successors for each state)
  - the heuristic cost function  $h$  is the number of pairs of queens that are attacking each other

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

best moves reduce  $h = 17$  to  $h = 12$



local minimum with  $h = 1$

# Artificial Intelligence

## Constraint satisfaction problems

# Constraint satisfaction problems (CSPs)

- Standard search problem: **state** is a "black box" – any data structure that supports successor function and goal test  
CSP:
  - CSP is defined by 3 components ( $X$ ,  $D$ ,  $C$ ):
  - **state** is defined by **variables**  $X_i$  with **values** from **domain**  $D_i$ ,
  - **goal test** is a set of **constraints** ( $C$ ) specifying allowable combinations of values for subsets of variables
- Allow  $X_1$  and  $X_2$  both have the domain  $\{A, B\}$  – with more power than standard search algor
  - Constraints:
    - $\langle(X_1, X_2), [(A, B), (B, A)]\rangle$ , or
    - $\langle(X_1, X_2), X_1 \neq X_2\rangle$

# 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.,  $WA \neq NT$ , or  $(WA, NT) \in \{(\text{red}, \text{green}), (\text{red}, \text{blue}), (\text{green}, \text{red}), (\text{green}, \text{blue}), (\text{blue}, \text{red}), (\text{blue}, \text{green})\}$

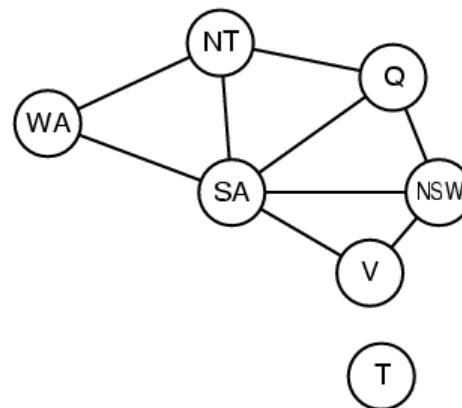
# Example: Map-Coloring



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

# Constraint graph

- **Binary CSP:** each constraint relates two variables
- **Constraint graph:** nodes are variables, arcs are constraints



# Varieties of CSPs

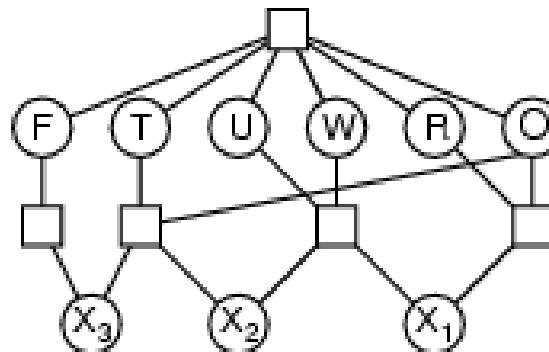
- **Discrete variables**
  - finite domains:
    - $n$  variables, domain size  $d \models O(d^n)$  complete assignments
    - e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
  - infinite domains:
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for 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

# Varieties of constraints

- **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., crypt arithmetic column constraints

# Example: Cryptarithmetic

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



- **Variables:**  $F, T, U, W, R, O, X_1, X_2, X_3$
- **Domains:**  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- **Constraints:**  $\text{Alldiff}(F, T, U, W, R, O)$

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

# Real-world CSPs Examples

- Assignment problems  
e.g., who teaches what class
- Timetabling problems  
e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling
- Notice that many real-world problems involve real-valued variables

## Backtracking search

- Variable assignments are **commutative**, e.g.  $a \times b = b \times a$ .  
eg: [ WA = red then NT = green ] same as [ NT = green then WA = red ]
- Only need to consider assignments to a single variable at each node
- Depth-first search for CSPs with single-variable assignments is called **backtracking** search
- Can solve  $n$ -queens for  $n \approx 25$

## Backtracking search algorithm

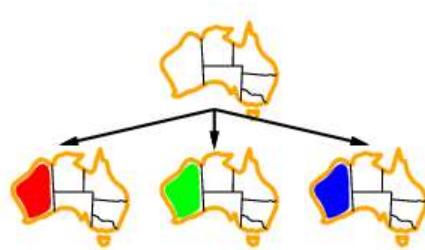
1. **Set** each variable as undefined, empty stack. All variables are future variables.
2. **Select** a future variable as current variable.  
If it exists, delete it from FUTURE and stack it (top = current variable),  
if not, the assignment is a *solution*.
3. **Select** an unused value for the current variable.  
If it exists, mark the value as used,  
if not, set current variable as undefined,  
    mark all its values as unused,  
    unstack the variable and add it to FUTURE,  
    if stack is empty, *there is no solution*,  
    if not, go to 3.
4. **Test** constraints between past variables and the current one.  
If they are satisfied, go to 2,  
if not, go to 3.

(It is possible to use heuristics to select variables (2.) and values (3.).

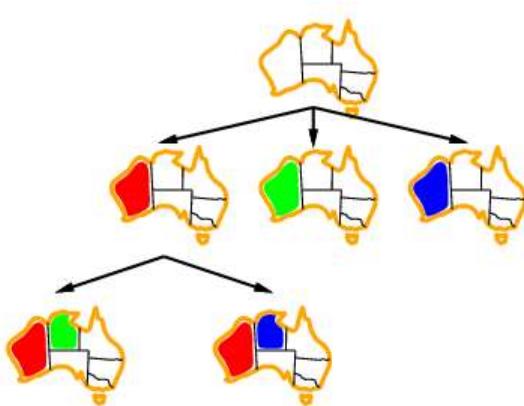
# Backtracking example



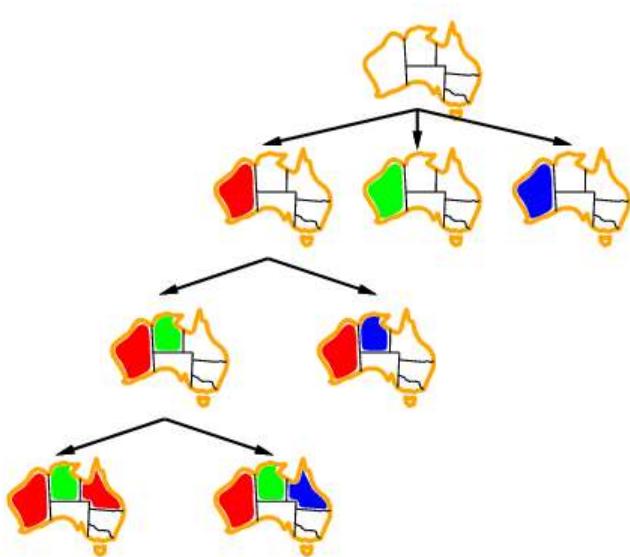
# Backtracking example



# Backtracking example



# Backtracking example

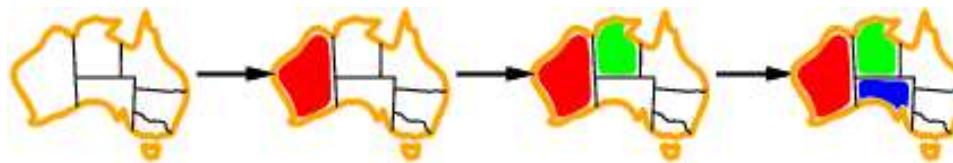


## Improving backtracking efficiency

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

# Most constrained variable

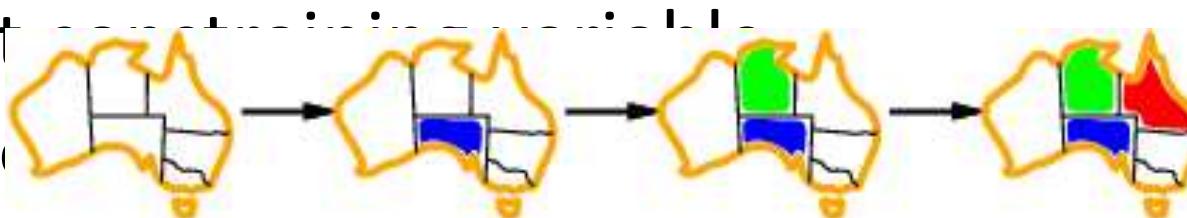
- Most constrained variable:  
choose the variable with the fewest legal values



- a.k.a. **minimum remaining values (MRV)** heuristic

## Most constraining variable

- A good idea is to use it as a tie-breaker among most constrained variables
- Most constrained variable:
  - choose variable with fewest possible values
  - choose variable with fewest possible values

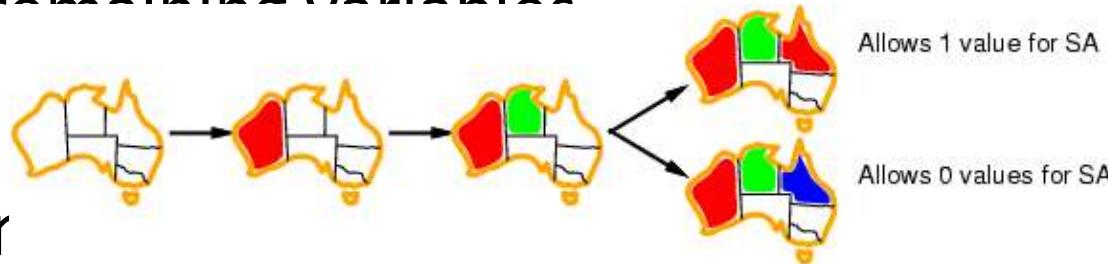


ints on

remaining variables

## Least constraining value

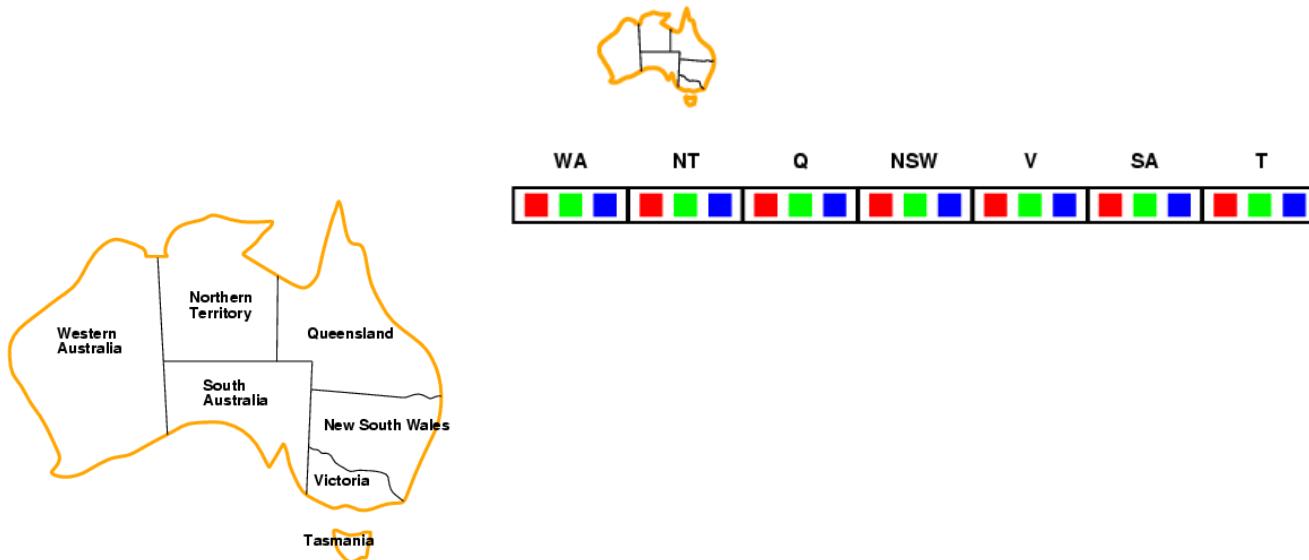
- Given a variable to assign, choose the least constraining value:
  - The one that rules out the fewest values in the remaining variables



- Cor  
queens feasible .000

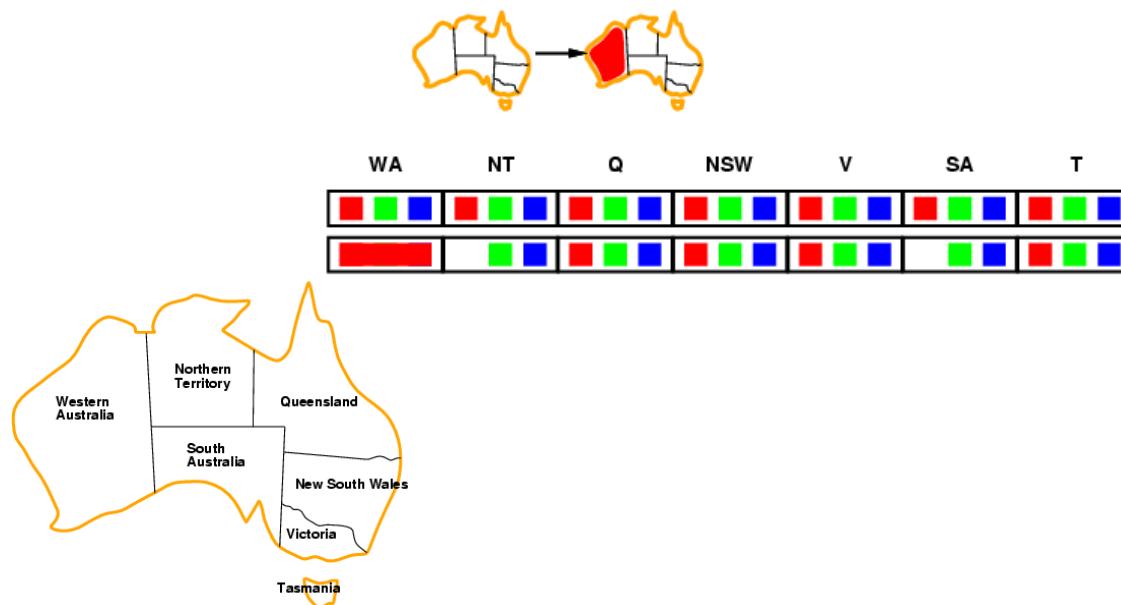
# Forward checking: example

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



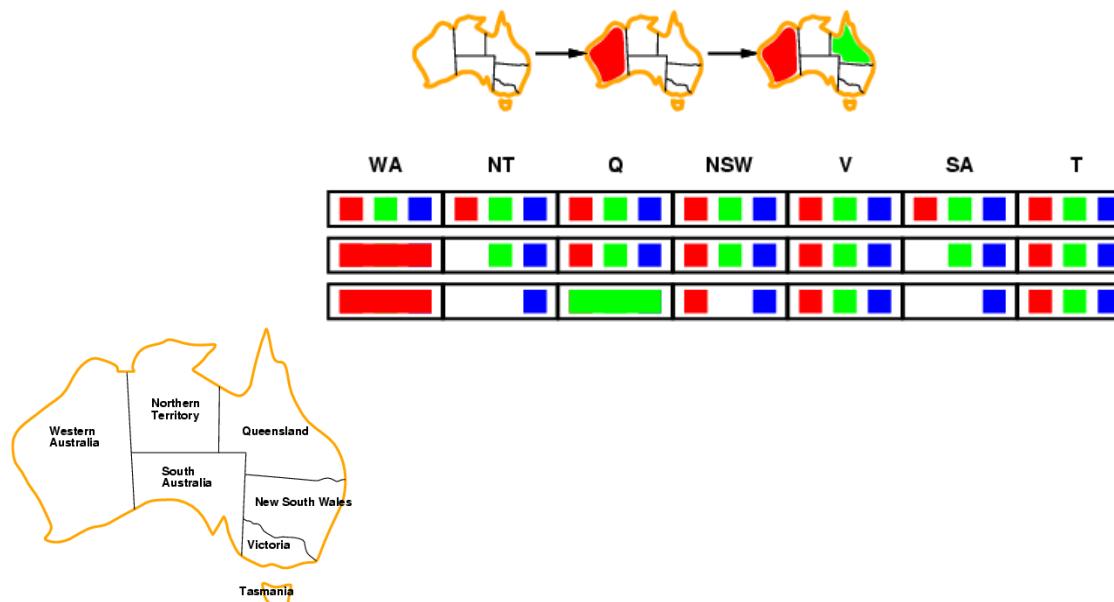
# Forward checking: example

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



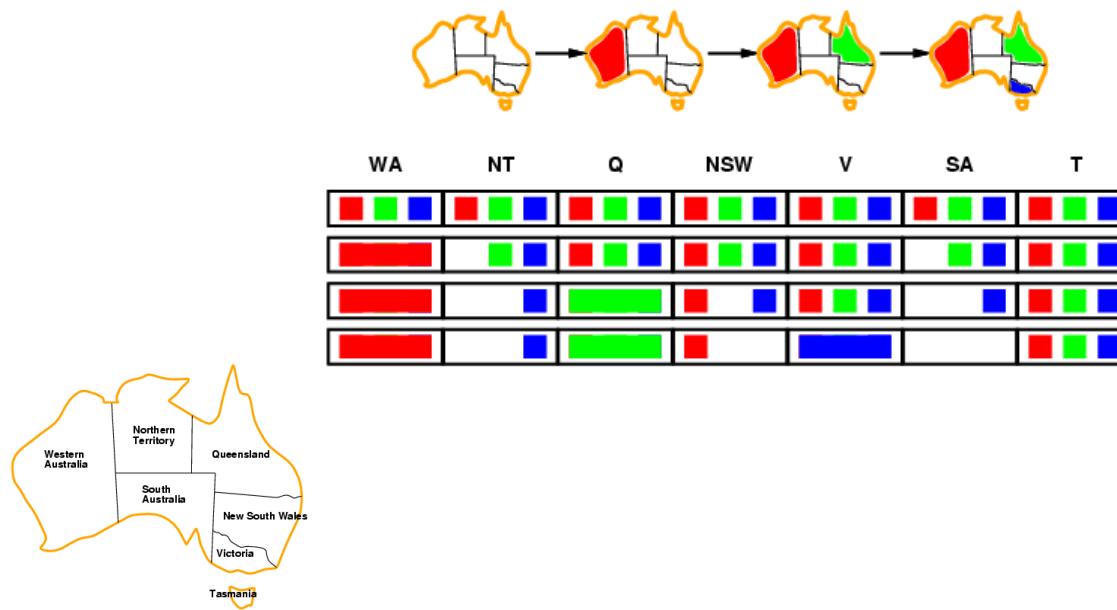
# Forward checking: example

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



# Forward checking: example

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



# Constraint propagation

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



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

- NT and SA cannot both be red
- Constraint propagation repeatedly enforces constraints locally

# Constraint propagation

- **Forward checking** does not detect the “blue” inconsistency, because it does not look far enough ahead.
- **Constraint propagation** is the general term for propagating the implications of a constraint on one variable onto other variables.
- The idea of **arc consistency** provides a fast method of constraint propagation that is substantially stronger than forward checking.

# Constraint Satisfaction Problems

- So far
  - All solutions are equally good
- In some real world applications, we
  - Not only want **feasible** solutions, but also **good** solutions
  - We have different **preferences** on constraints
  - Problems are too constrained that there is no solution satisfying all constraints

## **SEND MORE MONEY - Problem**

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline = \text{M O N E Y} \end{array}$$

Cryptarithmetic problem: mathematical puzzles where digits are replaced by symbols

Find unique digits the letters represent, satisfying the above constraints

## SEND MORE MONEY - Model

- Variables
  - S, E, N, D, M, O, R, Y
- Domain
  - $\{0, \dots, 9\}$
- Constraint
  - AllDiff{S, E, N, D, M, O, R, Y }

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline = \text{M O N E Y} \end{array}$$

# SEND MORE MONEY

- Model

- Constraints

- Distinct variables,  $S \neq E, M \neq S, \dots$
- $S*1000 + E*100 + N*10 + D$   
+  
 $M*1000 + O*100 + R*10 + E$   
=  $M*10000 + O*1000 + N*100 + E*10 + Y$

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline = \text{M O N E Y} \end{array}$$

## **SEND MORE MONEY – How?**

- How would you solve the problem using CP techniques?
  - Search tree with backtracking
  - Constraint propagation
  - Forward & backward checking
  - Combination of above?
- Different problems may find different techniques more appropriate

## SEND MORE MONEY - Solution

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R} \\ \hline \text{E} \\ = \text{M O N E Y} \end{array}$$

$$\begin{array}{r} 9 5 6 7 \\ + 1 0 8 5 \\ \hline = 1 0 6 5 2 \end{array}$$

$$\begin{array}{r} \text{S E N D} \\ + \text{M O S T} \\ \hline = \text{M O N E Y} \end{array}$$

- Is this the only solution?

# 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.
- In previous topics, we have studied the search strategies which are only associated with a single agent that aims to find the solution which often expressed in the form of a sequence of actions.
- But, there might be some situations where more than one agent is searching for the solution in the same search space, and this situation usually occurs in game playing.
- The environment with more than one agent is termed as multi-agent environment, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance.
- So, 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.
- Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

# Game Playing

Why do AI researchers study game playing?

1. It's a good reasoning problem, formal and nontrivial.
2. Direct comparison with humans and other computer programs is easy.

# Types of Games in AI:

	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

- **Perfect information:** A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.
- **Imperfect information:** If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.
- **Deterministic games:** Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
- **Non-deterministic games:** Non-deterministic are those games which have various unpredictable events and has a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games.

## Zero-Sum Game

- Zero-sum games are adversarial search which involves pure competition.
- In Zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent.
- One player of the game try to maximize one single value, while other player tries to minimize it.
- Each move by one player in the game is called as ply.
- Chess and tic-tac-toe are examples of a Zero-sum game.
- The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:
  1. What to do.
  2. How to decide the move
  3. Needs to think about his opponent as well
  4. The opponent also thinks what to do
- Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI

## **Formalization of the problem:**

- A game can be defined as a type of search in AI which can be formalized of the following elements:
- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0,  $\frac{1}{2}$ . And for tic-tac-toe, utility values are +1, -1, and 0.

## 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.
- **Example: Tic-Tac-Toe game tree:**
- The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:
- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.

# Game Tree (2-player, Deterministic, Turns)

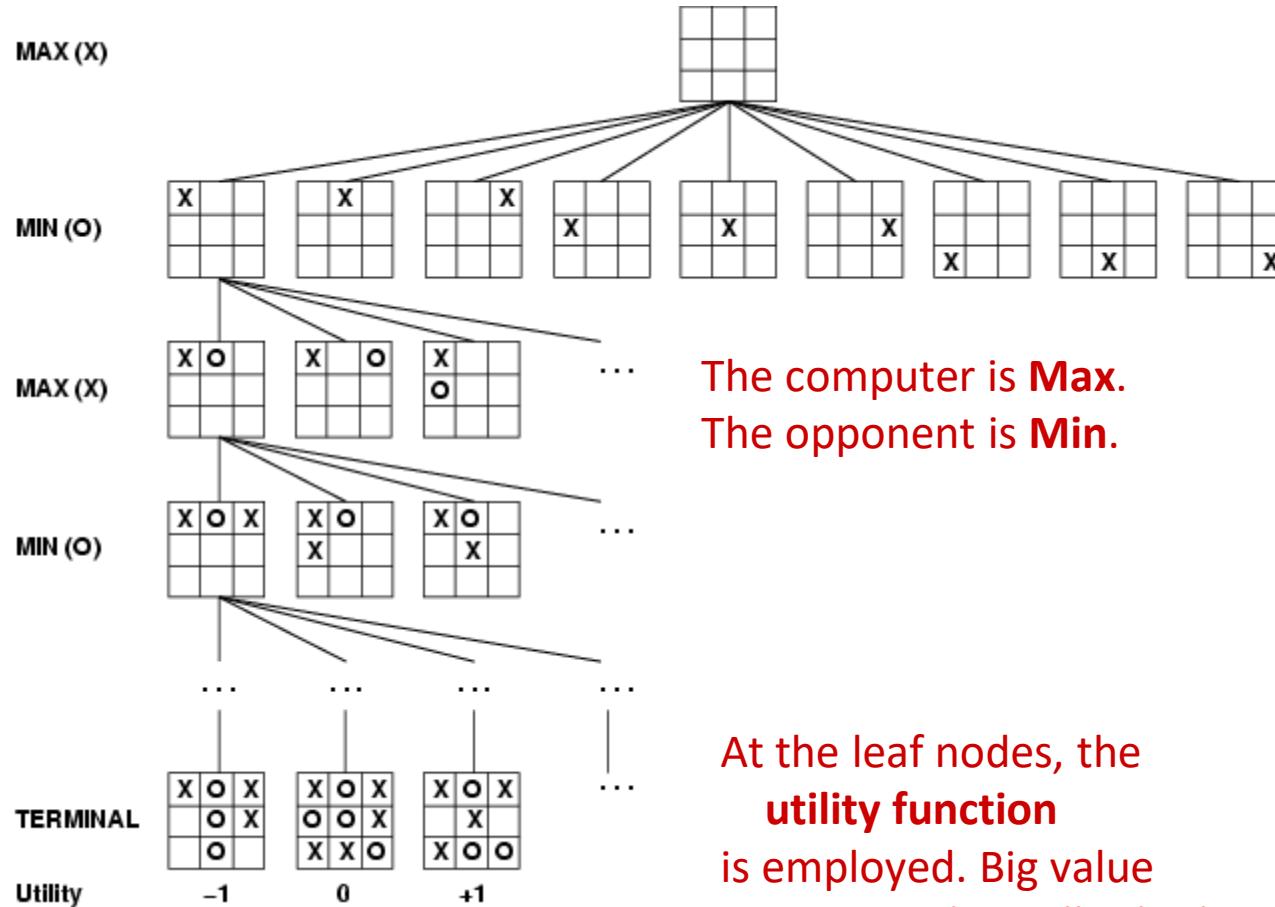
computer's  
turn

opponent's  
turn

computer's  
turn

opponent's  
turn

leaf nodes  
are evaluated



## Example Explanation:

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

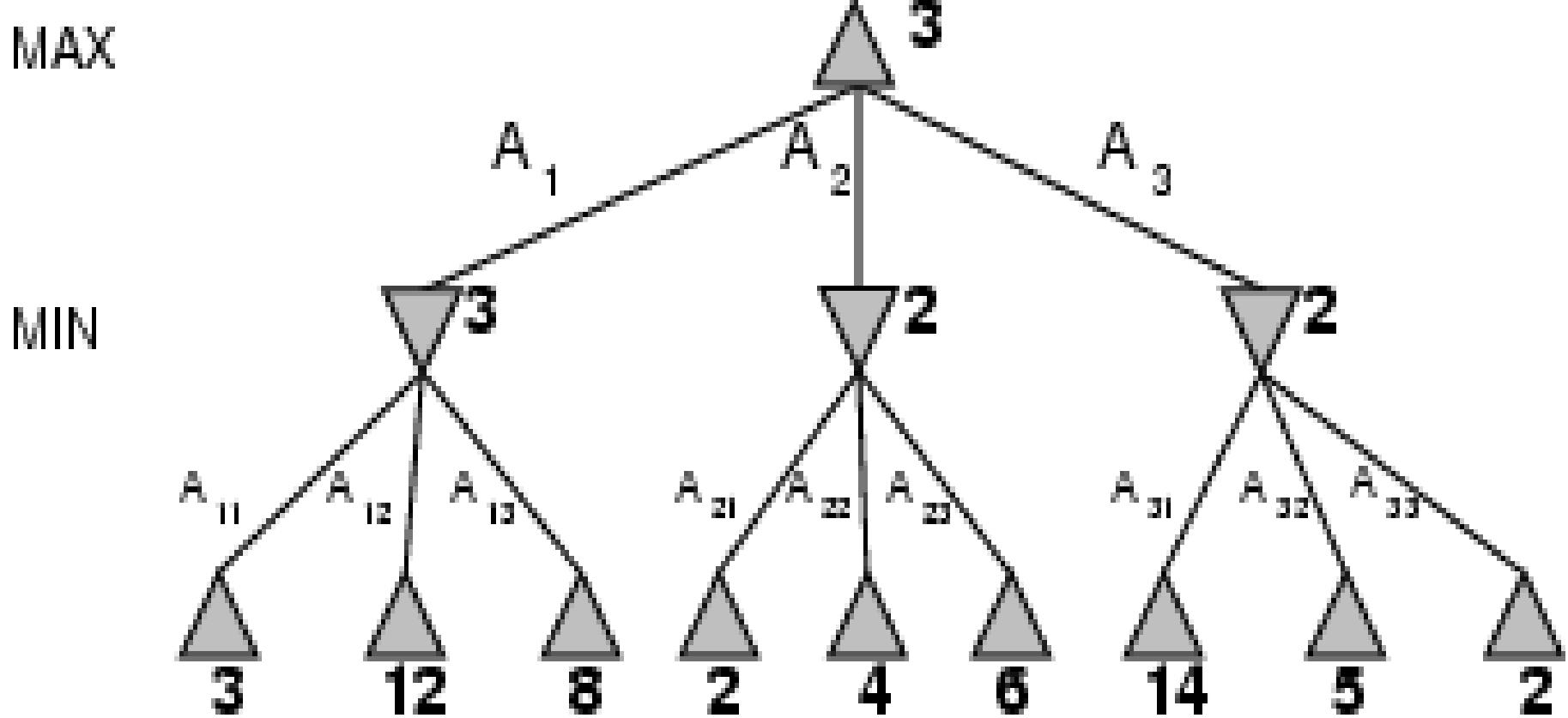
- Hence adversarial Search for the minimax procedure works as follows:
- It aims to find the optimal strategy for MAX to win the game.
- It follows the approach of Depth-first search.
- In the game tree, optimal leaf node could appear at any depth of the tree.
- Propagate the minimax values up to the tree until the terminal node discovered.
- In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as  $\text{MINIMAX}(n)$ . MAX prefer to move to a state of maximum value and MIN prefer to move to a state of minimum value then:

For a state  $s$   $\text{MINIMAX}(s) =$

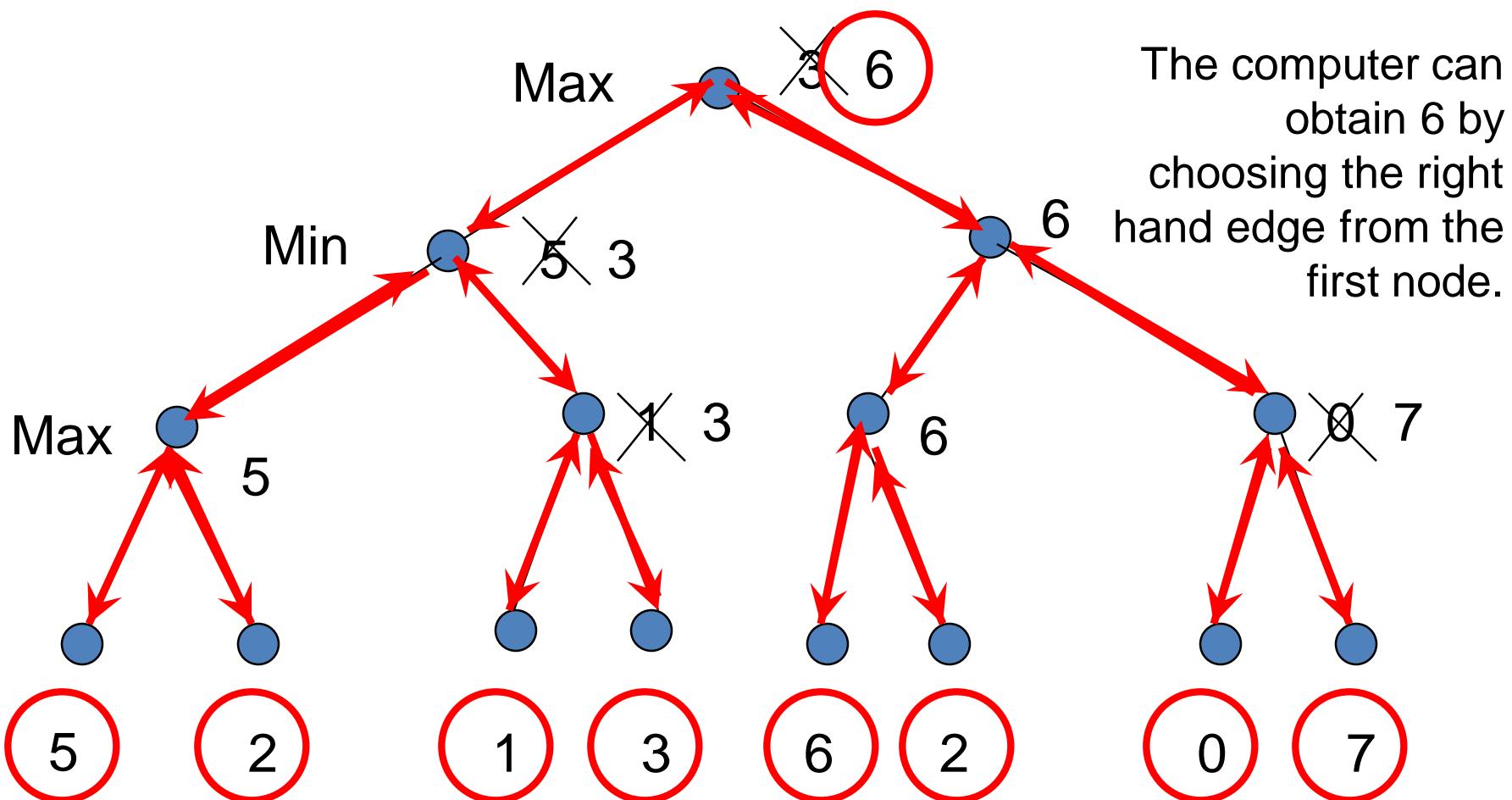
$$\left\{ \begin{array}{ll} \text{UTILITY}(s) & \text{If } \text{TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If } \text{PLAYER}(s) = \text{MIN}. \end{array} \right.$$

# Minimax

- Perfect play for **deterministic** games
- Idea: choose move to position with highest **minimax value**  
= best achievable payoff against best play
- E.g., 2-ply game:



# Minimax – Animated Example



# Minimax algorithm

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

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

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

---

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

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

```
  v  $\leftarrow -\infty$ 
```

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

```
    v  $\leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$ 
```

```
  return v
```

---

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

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

```
  v  $\leftarrow \infty$ 
```

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

```
    v  $\leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$ 
```

```
  return v
```

# Properties of Minimax

- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is  $O(b^m)$ , where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is  $O(bm)$ .

## Limitation of the minimax Algorithm:

- The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games 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** which we have discussed in the next topic.

## Alpha-Beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree.
- Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**.
- This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.

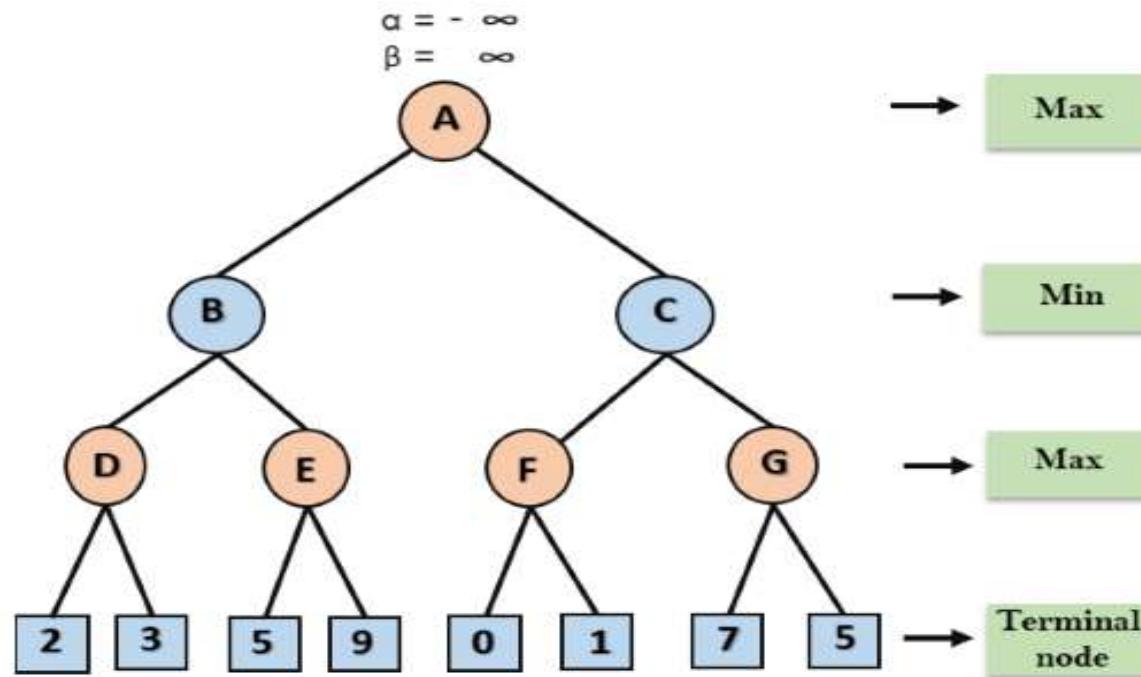
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
  - **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is  $-\infty$ .
  - **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is  $+\infty$ .
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

## Condition for Alpha-beta pruning:

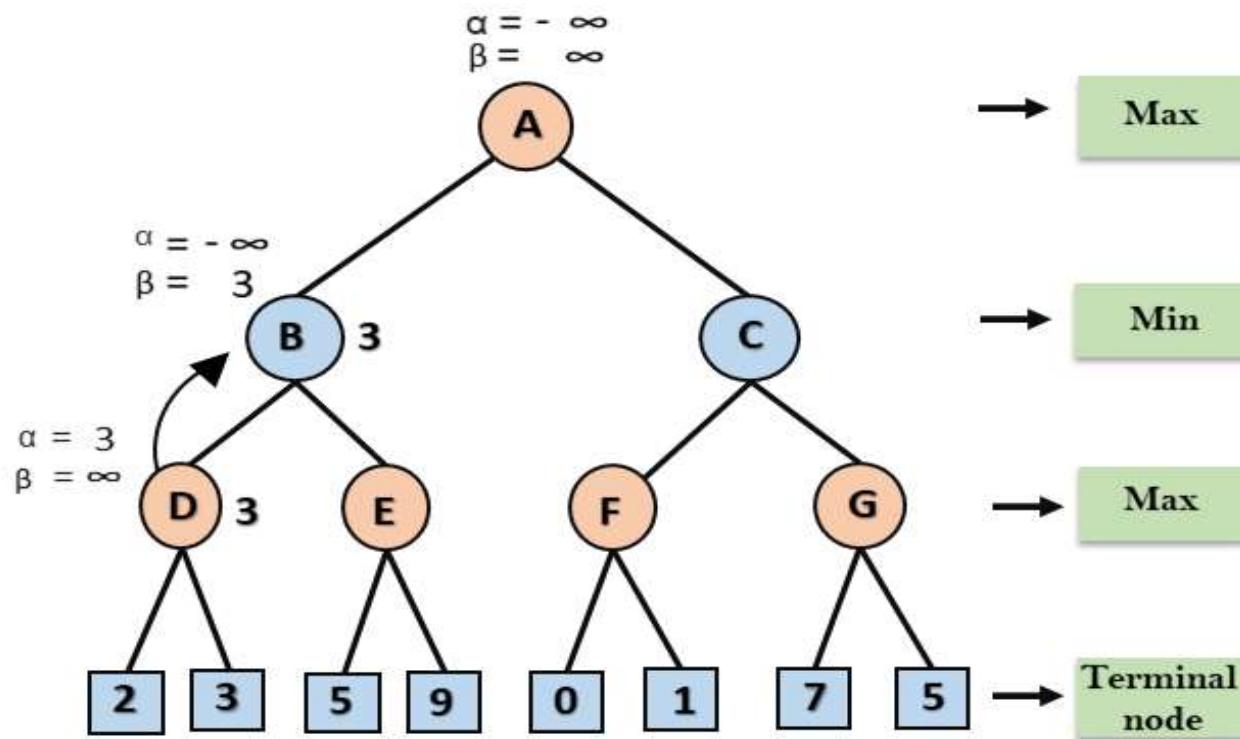
- The main condition which required for alpha-beta pruning is:  
 $\alpha >= \beta$
- Key points about alpha-beta pruning:
  - The Max player will only update the value of alpha.
  - The Min player will only update the value of beta.
  - While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
  - We will only pass the alpha, beta values to the child nodes.

## Working of Alpha-Beta Pruning:

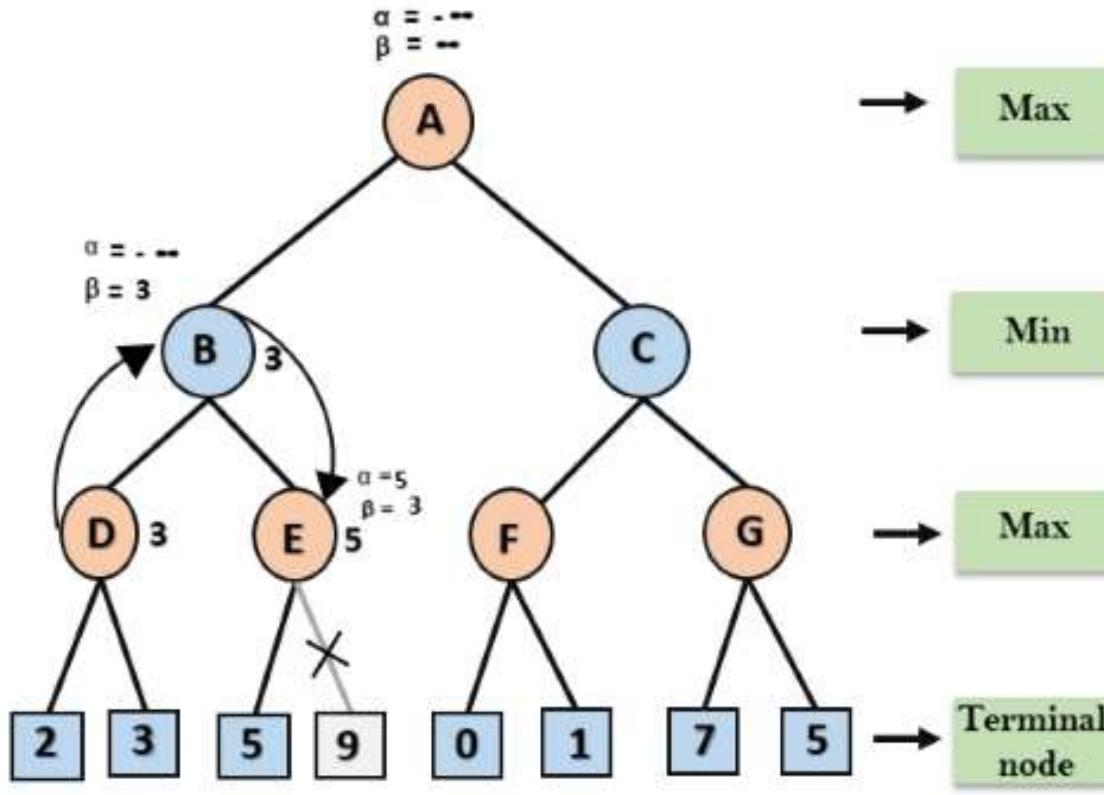
- Let's take an example of two-player search tree to understand the working of Alpha-beta pruning
- Step 1:** At the first step the, Max player will start first move from node A where  $\alpha = -\infty$  and  $\beta = +\infty$ , these value of alpha and beta passed down to node B where again  $\alpha = -\infty$  and  $\beta = +\infty$ , and Node B passes the same value to its child D.



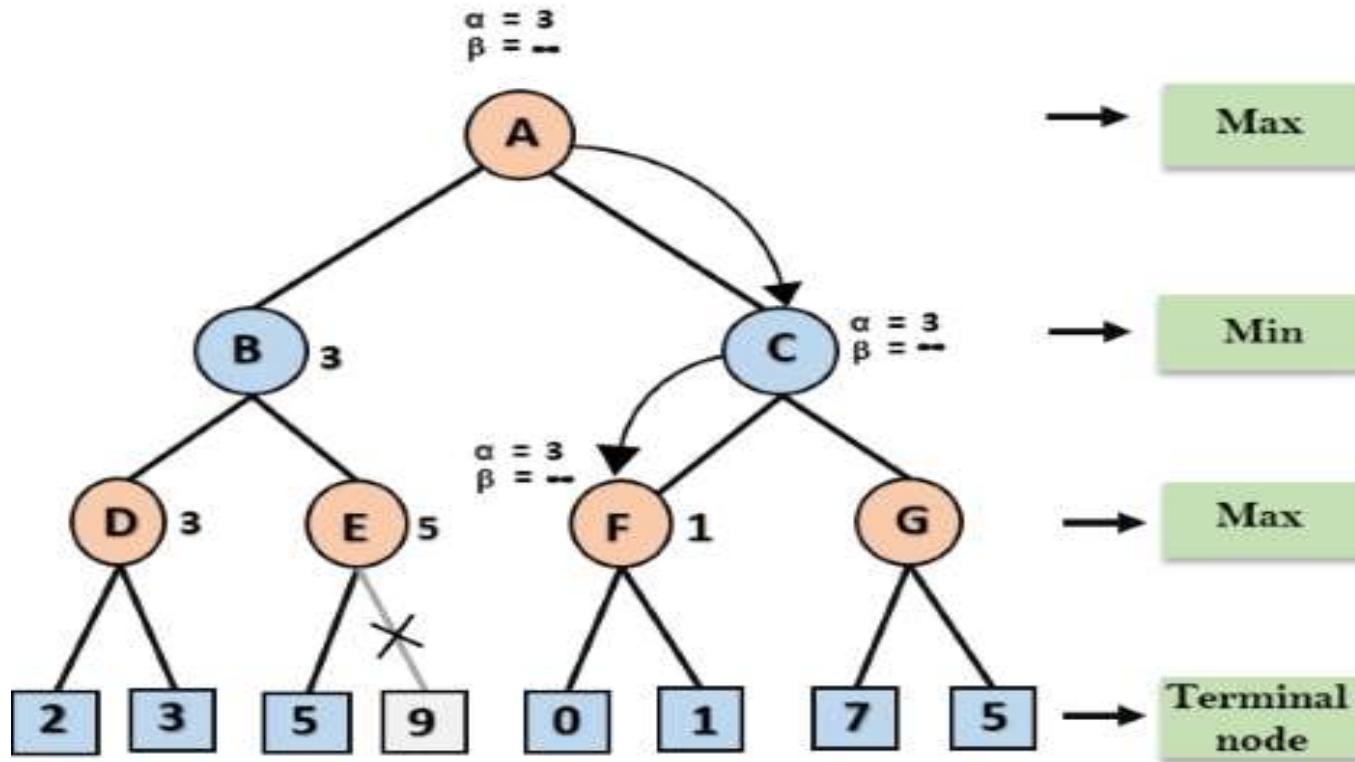
- **Step 2:** At Node D, the value of  $\alpha$  will be calculated as its turn for Max. The value of  $\alpha$  is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of  $\alpha$  at node D and node value will also 3.
- **Step 3:** Now algorithm backtrack to node B, where the value of  $\beta$  will change as this is a turn of Min, Now  $\beta = +\infty$ , will compare with the available subsequent nodes value, i.e. min ( $\infty$ , 3) = 3, hence at node B now  $\alpha = -\infty$ , and  $\beta = 3$ .



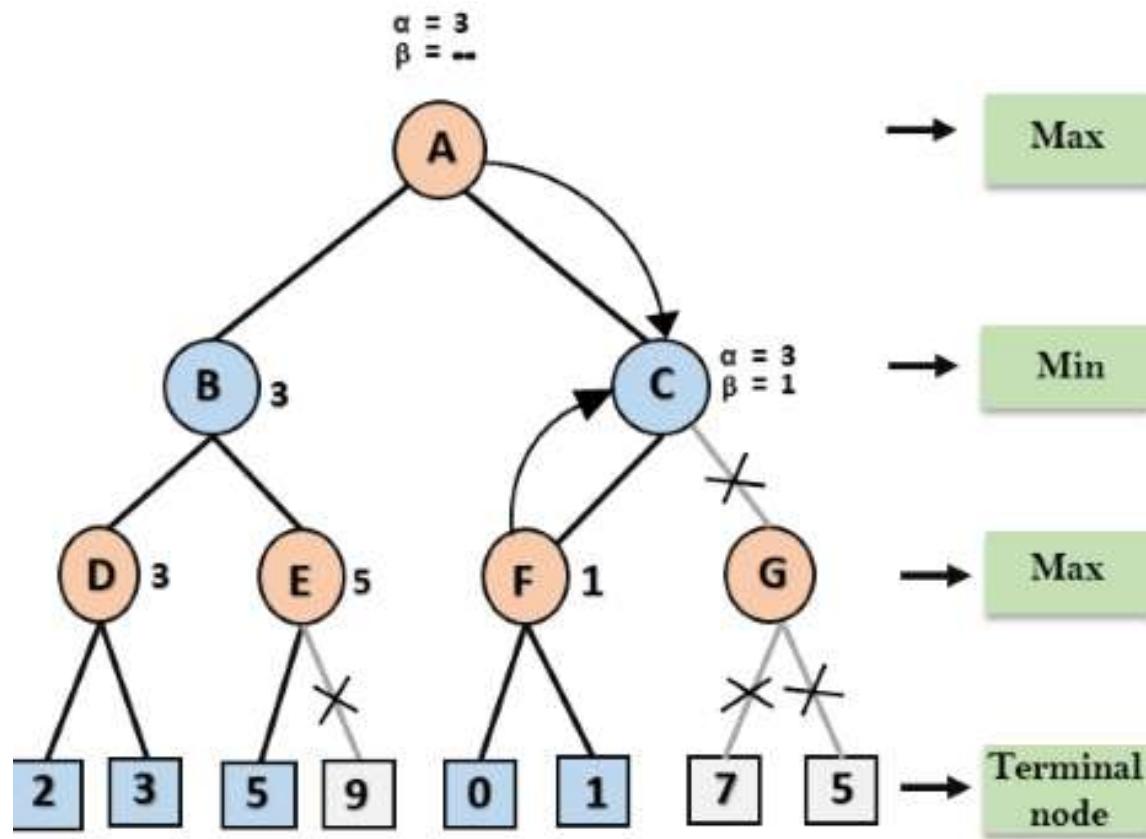
- In the next step, algorithm traverse the next successor of Node B which is node E, and the values of  $\alpha = -\infty$ , and  $\beta = 3$  will also be passed.
- **Step 4:** At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so  $\max(-\infty, 5) = 5$ , hence at node E  $\alpha = 5$  and  $\beta = 3$ , where  $\alpha >= \beta$ , so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



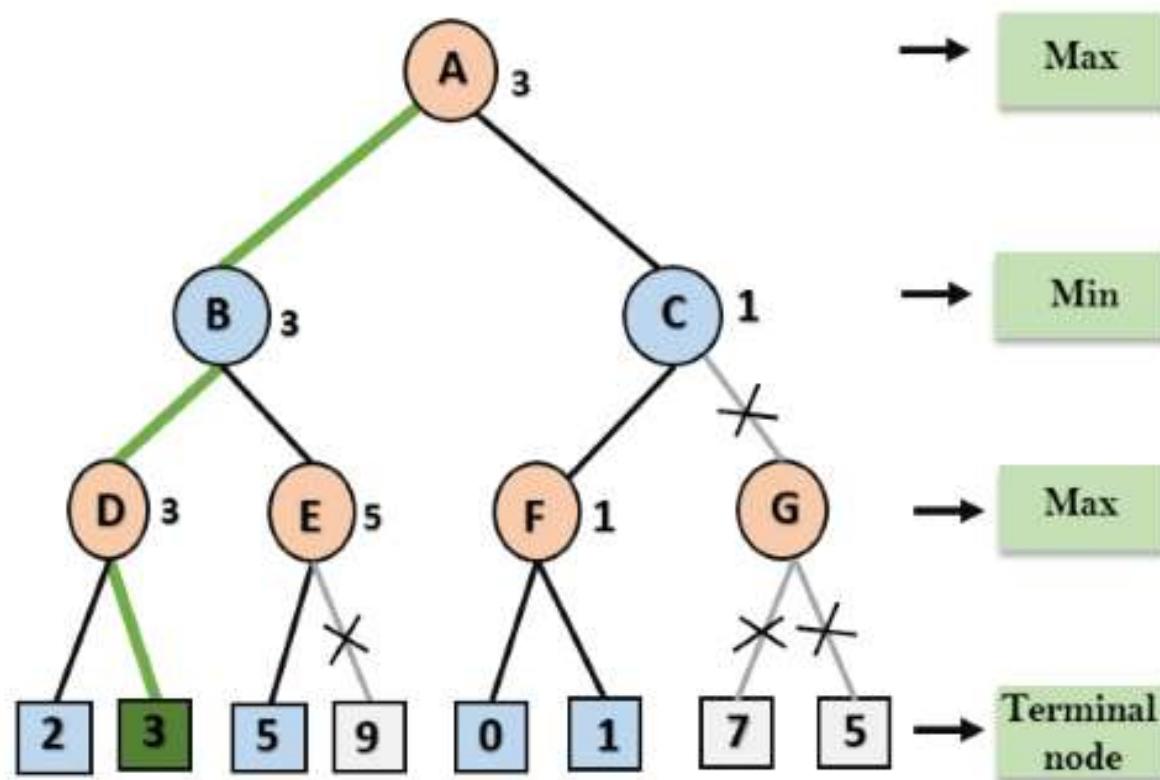
- **Step 5:** At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as  $\max(-\infty, 3) = 3$ , and  $\beta = +\infty$ , these two values now passes to right successor of A which is Node C.
- At node C,  $\alpha = 3$  and  $\beta = +\infty$ , and the same values will be passed on to node F.
- **Step 6:** At node F, again the value of  $\alpha$  will be compared with left child which is 0, and  $\max(3, 0) = 3$ , and then compared with right child which is 1, and  $\max(3, 1) = 3$  still  $\alpha$  remains 3, but the node value of F will become 1.



**Step 7:** Node F returns the node value 1 to node C, at C  $\alpha = 3$  and  $\beta = +\infty$ , here the value of beta will be changed, it will compare with 1 so  $\min(\infty, 1) = 1$ . Now at C,  $\alpha = 3$  and  $\beta = 1$ , and again it satisfies the condition  $\alpha >= \beta$ , so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



**Step 8:** C now returns the value of 1 to A here the best value for A is  $\max(3, 1) = 3$ . Following is the final game tree which is showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



## Move Ordering in Alpha-Beta pruning:

- The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.
- It can be of two types:
- **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is  $O(b^m)$ .
- **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is  $O(b^{m/2})$ .

## Rules to find good ordering:

- Following are some rules to find good ordering in alpha-beta pruning:
- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- We can bookkeep the states, as there is a possibility that states may repeat.

# Properties of $\alpha$ - $\beta$

- Pruning **does not** affect final result. This means that it **gets the exact same result as does full minimax.**
- Good move ordering improves effectiveness of pruning
- With "perfect ordering," time complexity =  $O(b^{m/2})$ 
  - Reduced in **doubles** depth of search

# Thank you!