MIT-WPU

।। विश्वशान्तिर्ध्रुवं ध्रुवा ।।

Dr. Vishwanath Karad

MIT WORLD PEACE UNIVERSITY | PUNE

TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

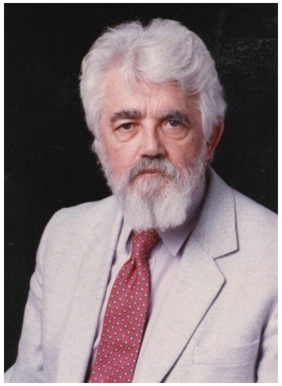
AI UNIT - I

UNIT-I

Overview of AI

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

Artificial Intelligence



John McCarthy

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

AI means –

- A system with intelligence is expected to behave as intelligently as a human

– A system with intelligence is expected to behave in the best possible manner

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.

?

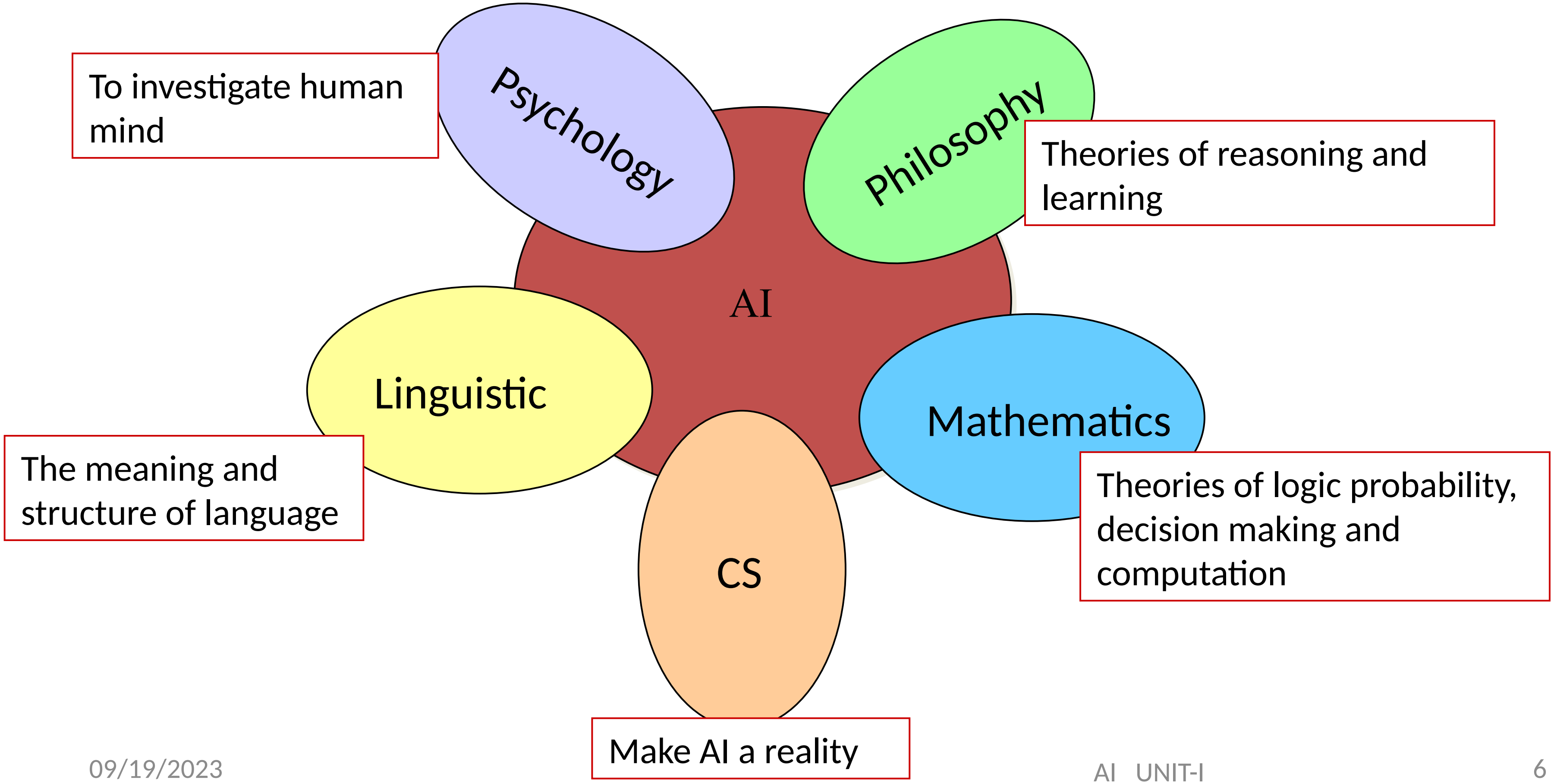
Goals of AI

To Create Expert Systems – The systems which exhibit intelligent behavior, learn, demonstrate, explain, and advice its users.

To Implement Human Intelligence in Machines – Creating systems that understand, think, learn, and behave like humans.

AI Foundations?

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



To investigate human mind

Psychology

Philosophy

Theories of reasoning and learning

AI

Linguistic

The meaning and structure of language

Mathematics

Theories of logic probability, decision making and computation

CS

Make AI a reality

AI FIELDS

AI Fields

- Speech Recognition
- Natural Language Processing
- Computer Vision
- Image Processing
- Robotics
- Pattern Recognition (Machine Learning)
- Neural Network (Deep Learning)

Define scope and view of Artificial Intelligence

Designing systems that are as intelligent as humans.

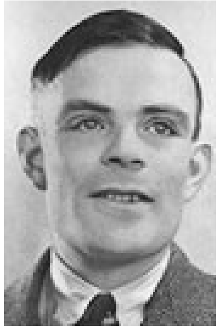
Embodied by the concept of the Turing Test – Turing test

Logic and laws of thought deals with studies of ideal or rational thought process and inference.

study of rational agents

The Turing Test

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



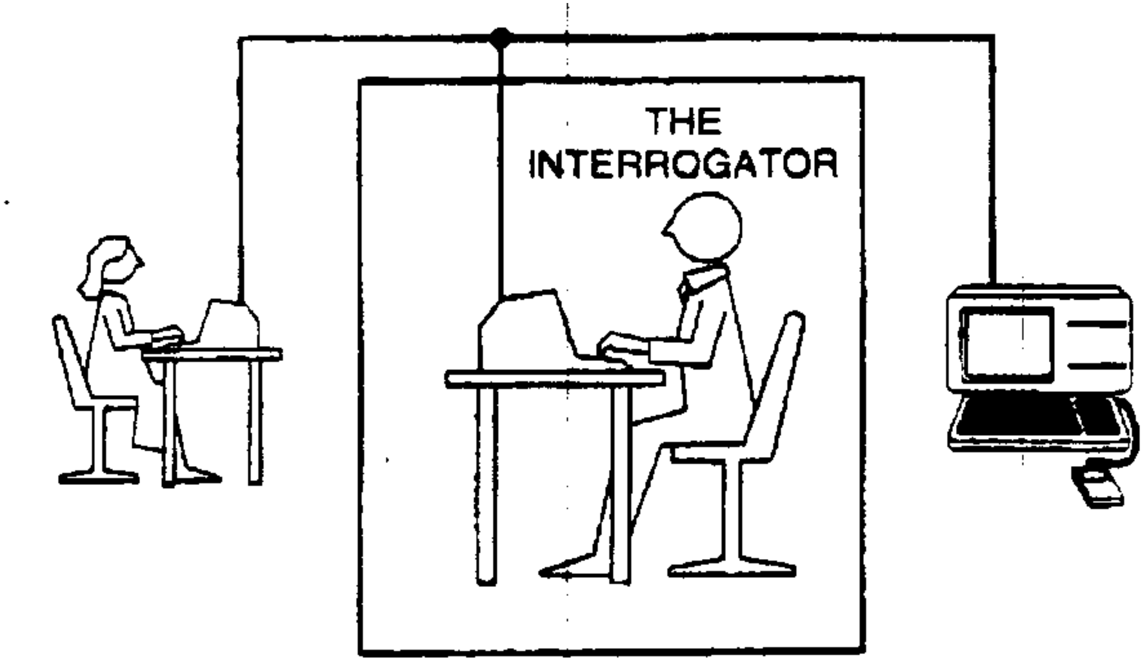


Figure 1.1 The Turing test.

History of AI

- **McCulloch and Pitts (1943)**
  - Developed a Boolean circuit model of brain
  - They wrote the paper explained how it is possible for neural networks to compute
- **Minsky and Edmonds (1951)**
  - Built a neural network computer (SNARC)
  - Used 3000 vacuum tubes and a network with 40 neurons.
- **Darmouth conference (1956):**
  - Conference brought together the founding fathers of artificial intelligence for the first time
  - In this meeting the term “Artificial Intelligence” was adopted.
- **1952-1969**
  - Newell and Simon - Logic Theorist was published (considered by many to be the first AI program)
  - Samuel - Developed several programs for playing checkers

History…. continued

- **1969-1979 Development of Knowledge-based systems**
  - Expert systems:
    - **Dendral**: Inferring molecular structures
    - **Mycin:** diagnosing blood infections
    - **Prospector:** recommending exploratory drilling.
- **In the 1980s, Lisp Machines developed and marketed.**
- **Around 1985, neural networks return to popularity**
- **In 1988, there was a resurgence of probabilistic and decision-theoretic methods**
- **The 1990's saw major advances in all areas**
    - machine learning, data mining
    - natural language understanding
    - vision, virtual reality, games etc

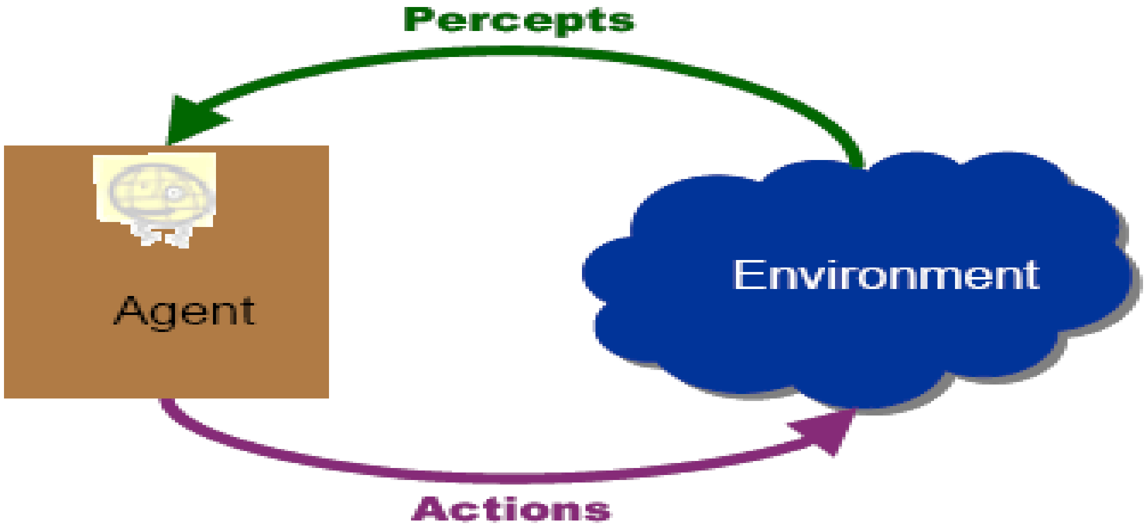
Applications of AI

- Gaming
- Natural Language Processing
- Expert Systems
- Vision Systems
- Speech Recognition
- Handwriting Recognition
- Intelligent Robots

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

Autonomous Agent

Agents	Environments
Robot	Room
Chatbot	Chatting
Vehicle	Road
Program	Data & Rules
Machine	Working Field

Examples Agents

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

- **Human agent:** eyes, ears, and other organs used as **sensors**;
- hands, legs, mouth, and other body parts used as **actuators/ Effector**

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

- **Software Agent(Soft Bot)**
  - Functions as sensors
  - Functions as actuators

- The term *bot* is derived from robot.
- software bots act only in digital spaces
- Is nothing more than a piece of code
- Example – **Chatbots**, the little messaging applications that pop up in the corner of your screen

Agent Terminology

- **Performance Measure of Agent** – It is the criteria, which determines how successful an agent is.
- **Behavior of Agent** – It is the action that agent performs after any given sequence of percepts.
- **Percept** – It is agent's perceptual inputs at a given instance.
- **Percept Sequence** – It is the history of all that an agent has perceived till date.
- **Agent Function** – It is a map from the precept sequence to an action.

What is an Intelligent Agent

- **An agent is anything that can**
  - *perceive* its *environment* through *sensors*, and
  - *act* upon that environment through *actuators* (or *effectors*)
- An Intelligent Agent must sense, must act, must be autonomous (to some extent),. It also must be rational.
- **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
- AI is about building rational agents.
- An agent should strive to "do the right thing"
- An agent is something that perceives and acts.
- A rational agent always does the right thing

  .

  - **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 it's abilities & constraints (Subject to resources)

What is an Intelligent Agent - Agent Function

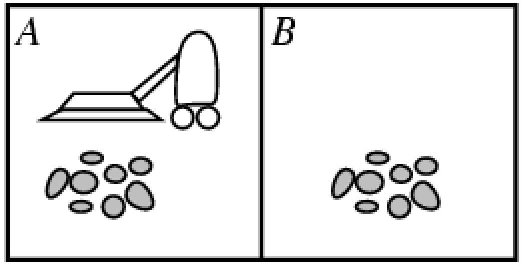
- Agent Function (percepts ==> actions)
  - Maps from percept histories to actions *f*: **P* ✉ A**
  - The agent program runs on the physical architecture to produce the function *f*
  - agent = architecture + program

```
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



- **Percepts**: location and contents, e.g., [*A*, *Dirty*]
- **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

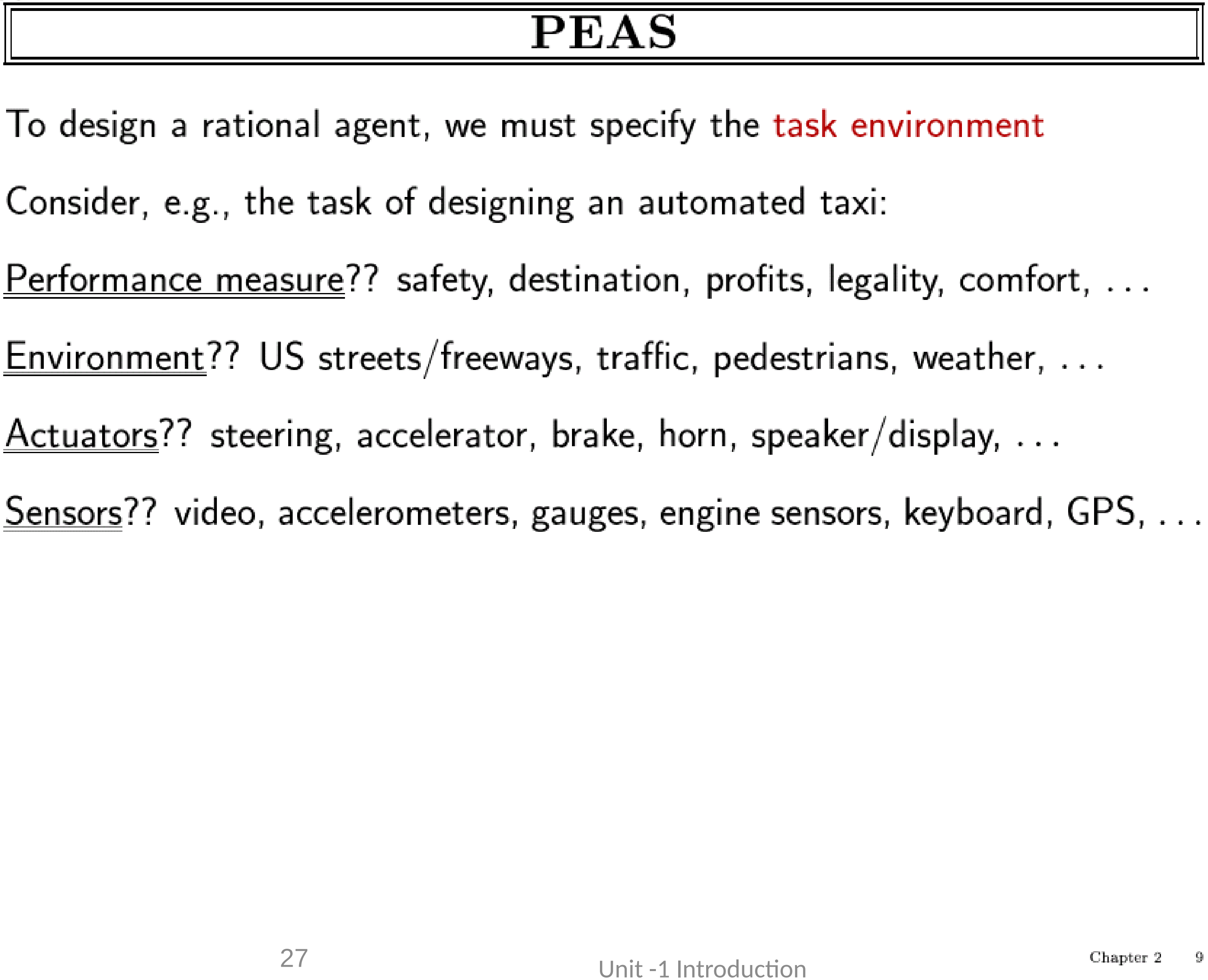
Consider, e.g., the task of designing an automated taxi:

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

- Environments in which agents operate can be defined in different ways
- Environment appears from the point of view of the agent itself.

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. Ex. Chess
- Most real world AI environments are not deterministic. Instead, they can be classified as stochastic.
- Ex: Self-driving vehicles are a classic example of stochastic AI processes.

Environment Types (cont.)

- Episodic (vs. sequential):
- In an episodic environment, there is a series of one-shot actions, and only the current percept is required for the action. Part Picking Robot
- However, in Sequential environment, an agent requires memory of past actions to determine the next best actions. Checker

- Static (vs. dynamic):
  - The environment is unchanged while an agent is deliberating
  - The environment is semi-dynamic if the environment itself does not change with the passage of time but the agent's performance score does.

- Discrete (vs. continuous):
  - Discrete AI environments are those on which a finite [although arbitrarily large] set of possibilities can drive the final outcome of the task. Chess is also classified as a discrete AI problem. Continuous AI environments rely on unknown and rapidly changing data sources. Vision systems in drones or self-driving cars operate on continuous AI environments.

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.

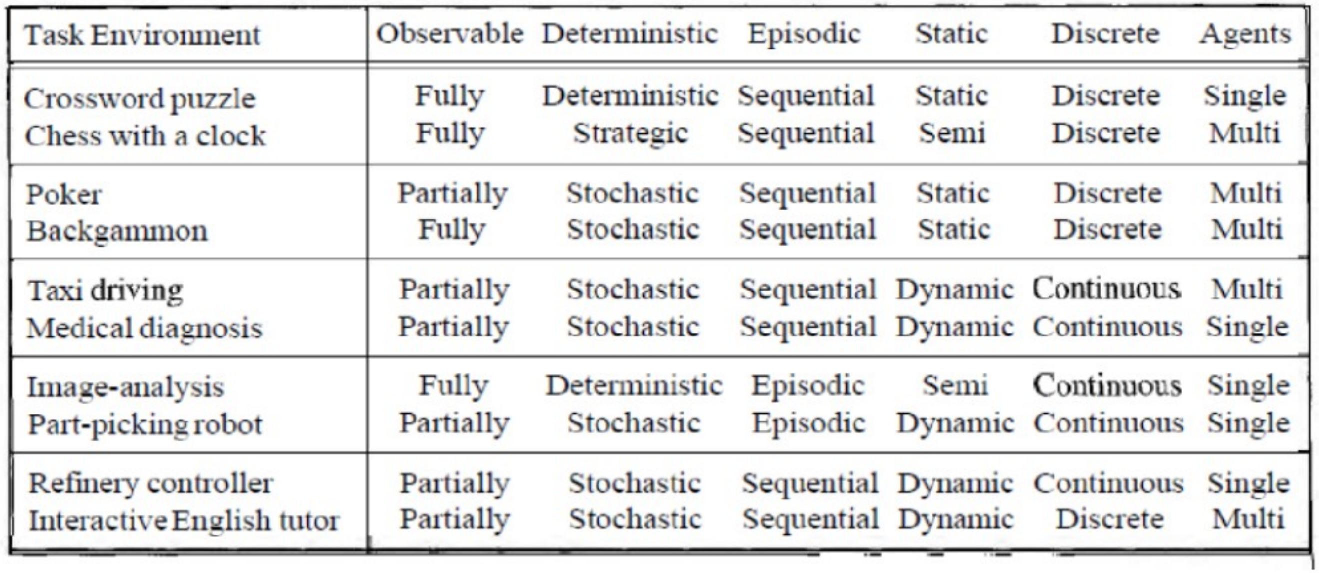
Ex: Poker, on the other hand, is an incomplete environments as AI strategies can't anticipate many moves in advance and, instead, they focus on finding a good 'equilibrium" at any given time.

- Single agent (vs. multi-agent):
  - An agent operating by itself in an environment.

Environment Types (cont.)

- Crossword Puzzle
- Chess
- Taxi driving
- Medical Diagnosis System

Environment Types (cont.)



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

The environment type largely determines the agent design.

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

Agent types

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

Agent Types

- Simple reflex agents
  - They choose actions only based on the current percept.
  - These are based on condition-action rules (It is a rule that maps a state (condition) to an action)
  - They are stateless devices which do not have memory of past world states.
- Model Based Reflex Agents (with memory)

  - Model : `knowledge about “how the things happen in the world”.`
  - have internal state which is used to keep track of past states of the world.
- Goal Based Agents
  - 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.

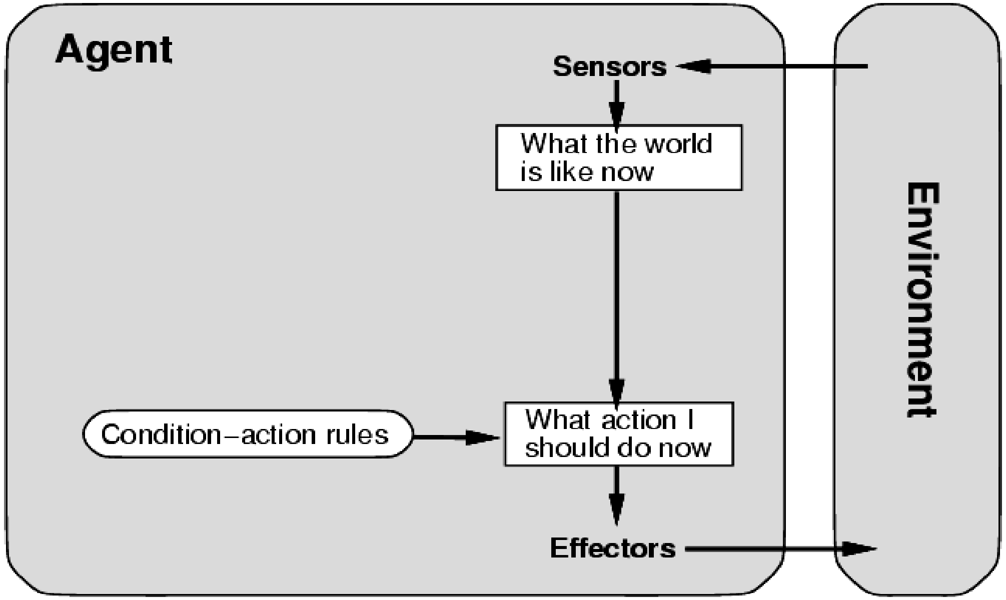
Note: All of these can be turned into “learning” agents

A Simple Reflex Agent

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

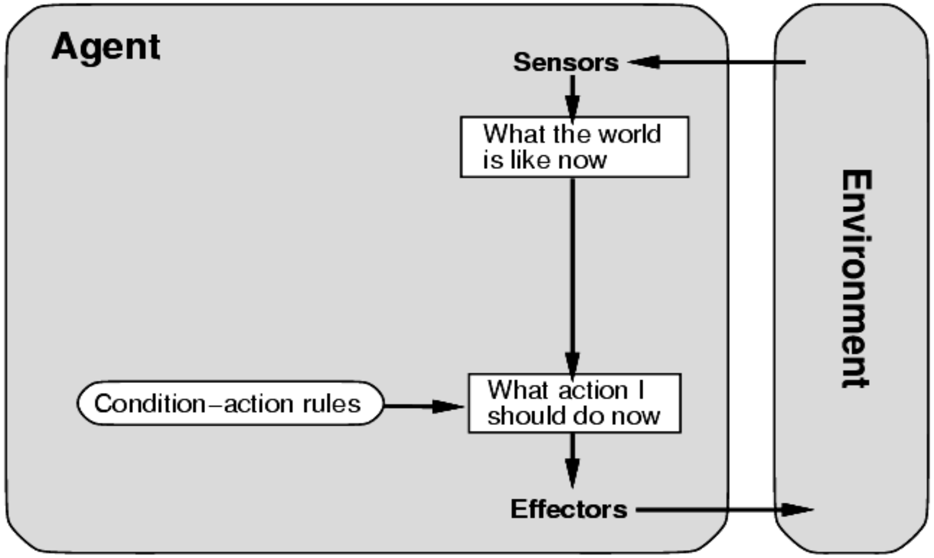
  if *car-in-front-brakes*

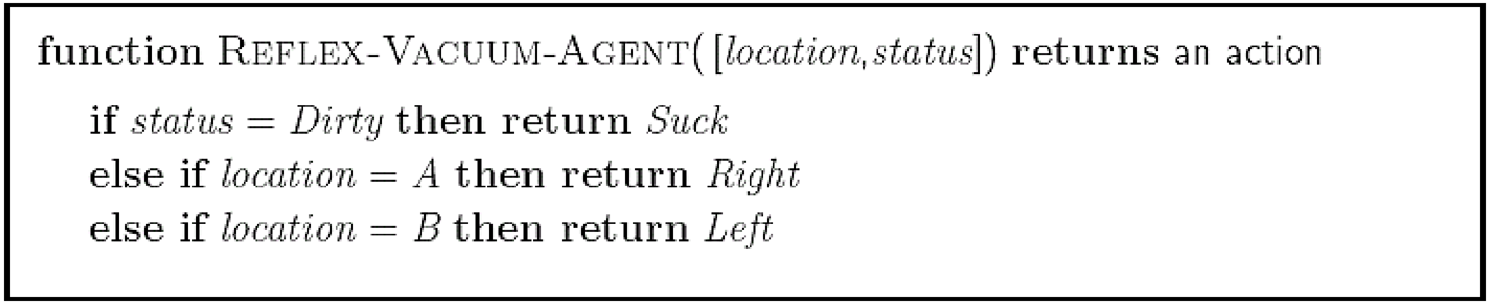
  then *initiate braking*
- Agent works by finding a rule whose condition matches the current situation
  - rule-based systems
- But, this only works if the current percept is sufficient for making the correct decision



```
function Simple-Reflex-Agent(percept) returns action
  static: rules, a set of condition-action rules

  state ← Interpret-Input(percept)
  rule ← Rule-Match(state, rules)
  action ← 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 $Left$
```

A Simple Reflex Agent

- **Problems with Simple reflex agents are :**

✓ Very limited intelligence.

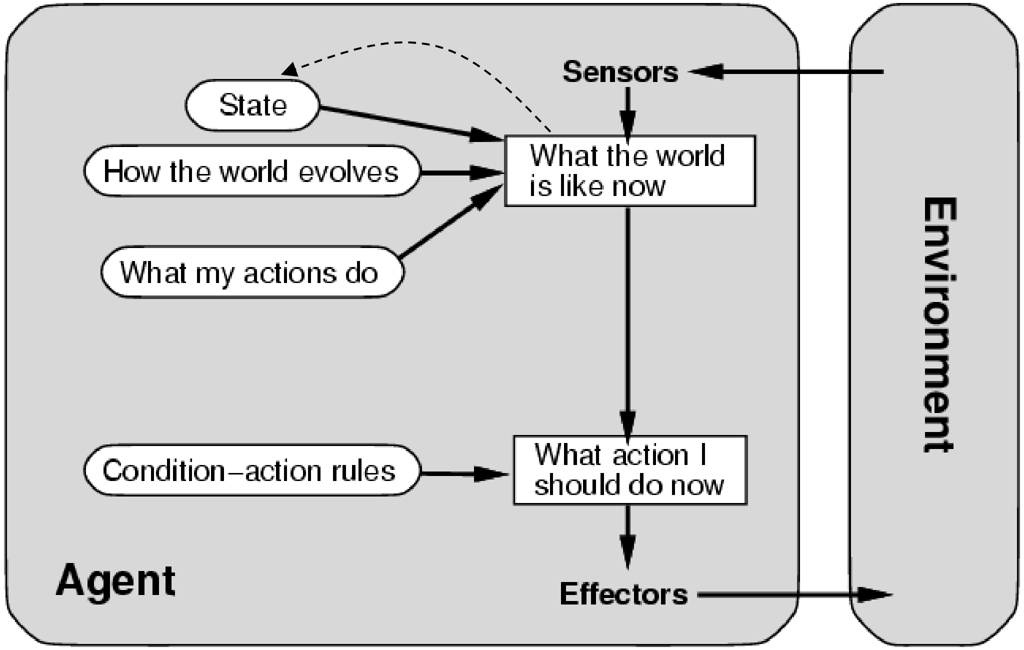
✓ No knowledge of non-perceptual parts of the state.

✓ Usually too big to generate and store.

✓ If there occurs any change in the environment, then the collection of rules needs to be updated.

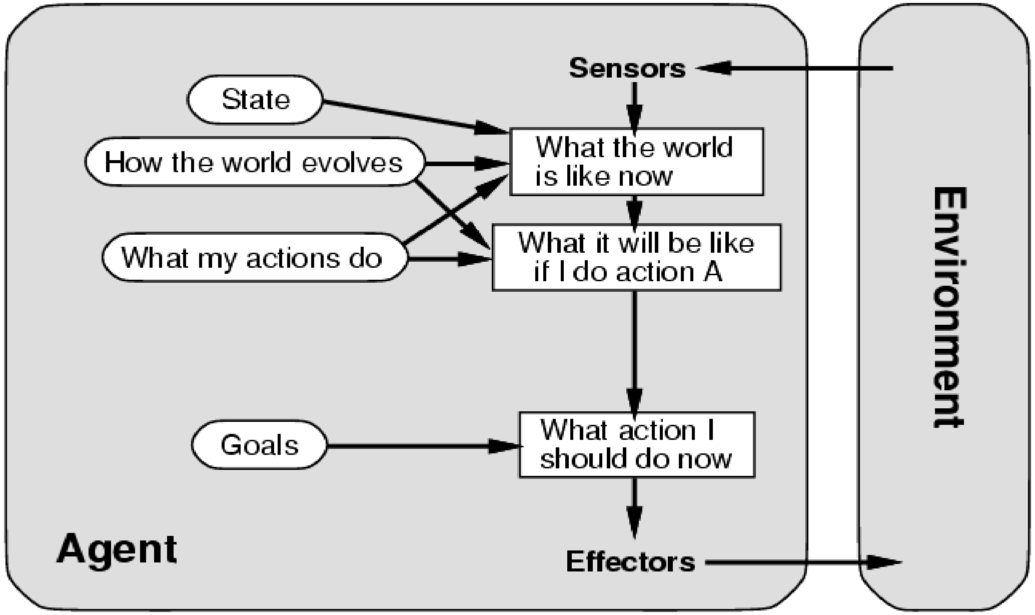
Agents that Keep Track of the World

- 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 ← Update-State(state, percept)
  rule ← Rule-Match(state, rules)
  action ← Rule-Action[rule]
  state ← Update-State(state, action)
  return action
```

Agents with Explicit Goals

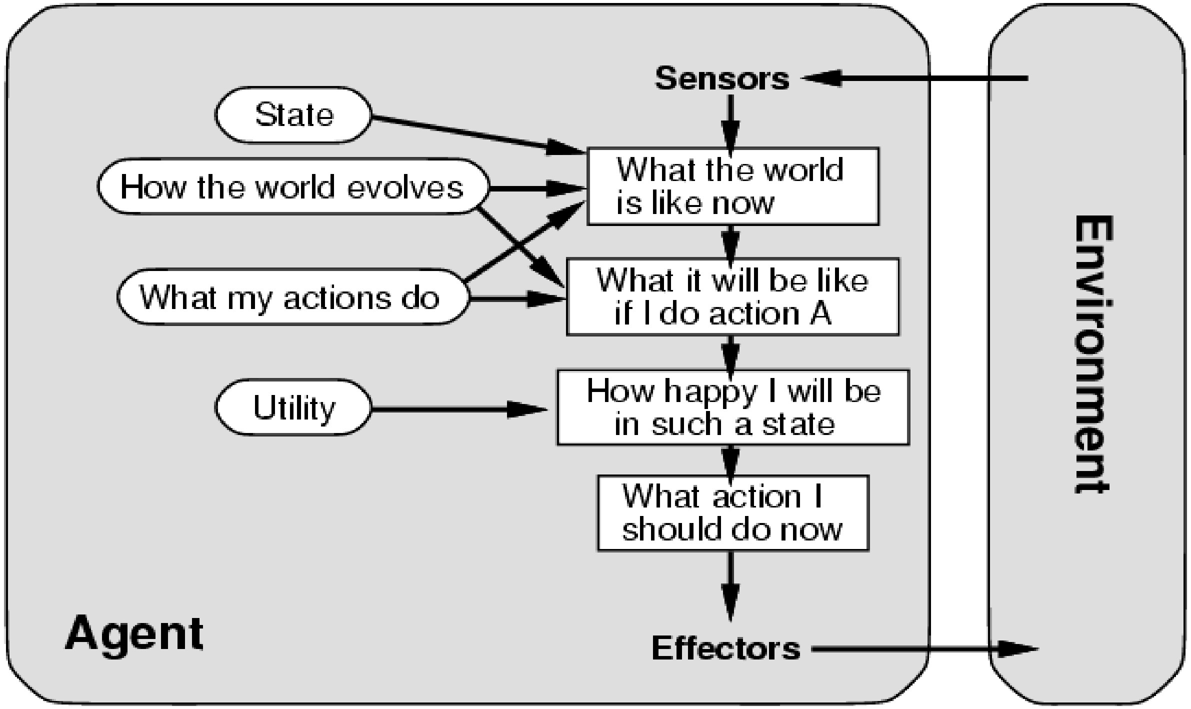

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Effectors
Environment

- **Reasoning about actions**
  - reflex agents only act based on pre-computed knowledge (rules)
  - goal-based (planning) act by reasoning about which actions achieve the goal
  - 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


Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Effectors
Agent
Environment

- 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

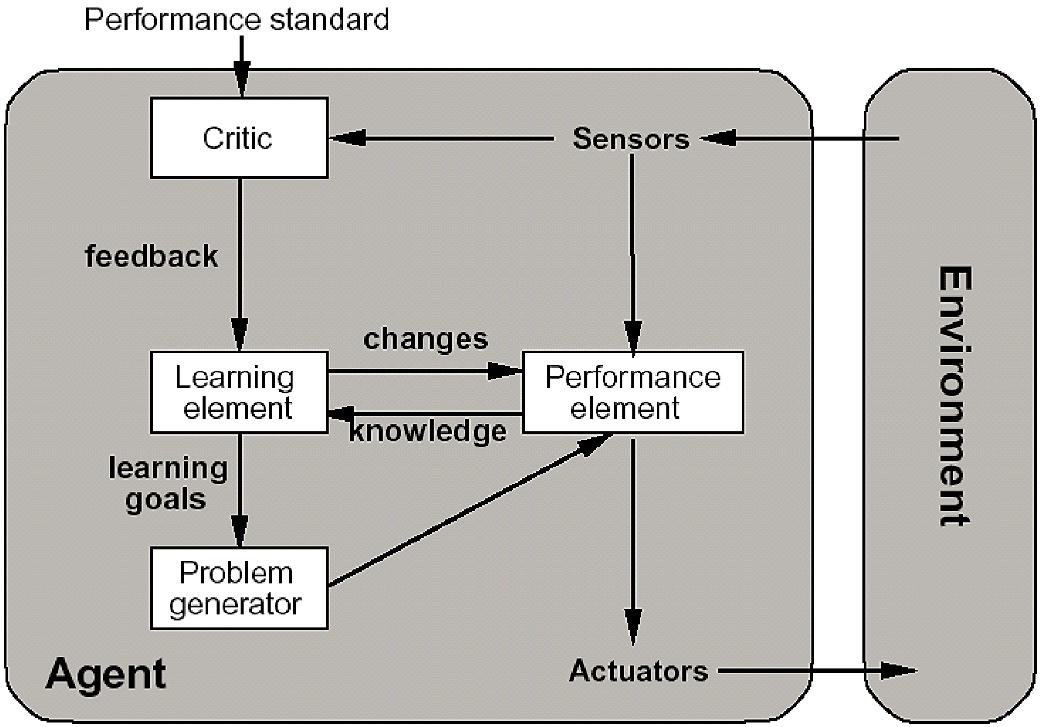
Utility-Based Agents (Cont.)

- 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(state) = measure of happiness

Learning Agents



- **Four main components:**
  - Performance element: the agent function
  - Learning element: responsible for making improvements by observing performance
  - Critic: gives feedback to learning element by measuring agent's performance
  - 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.

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

Search Strategies

- Problem solving and formulating a problem State Space Search
- Uninformed Search Techniques
- Informed Search Techniques
- Heuristic function
- A*
- AO* algorithms
- Hill climbing

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.

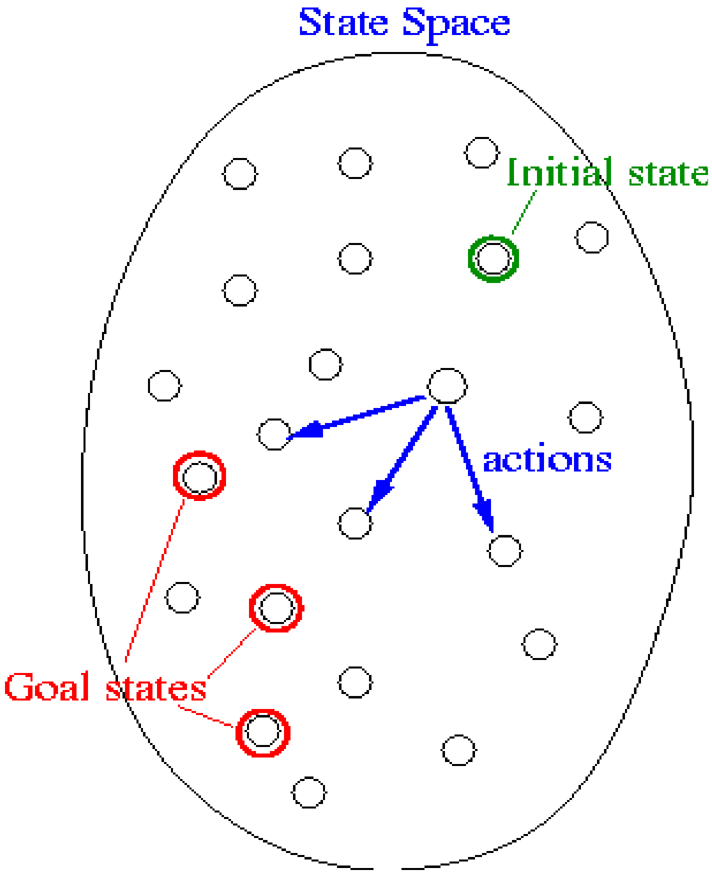
Search is the process of considering various possible sequences of operators applied to the initial state, and finding out a sequence which culminates in a goal state.

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→S is a set of operators
- G is the set of final states. Note that G ⊆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✉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.

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

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 :

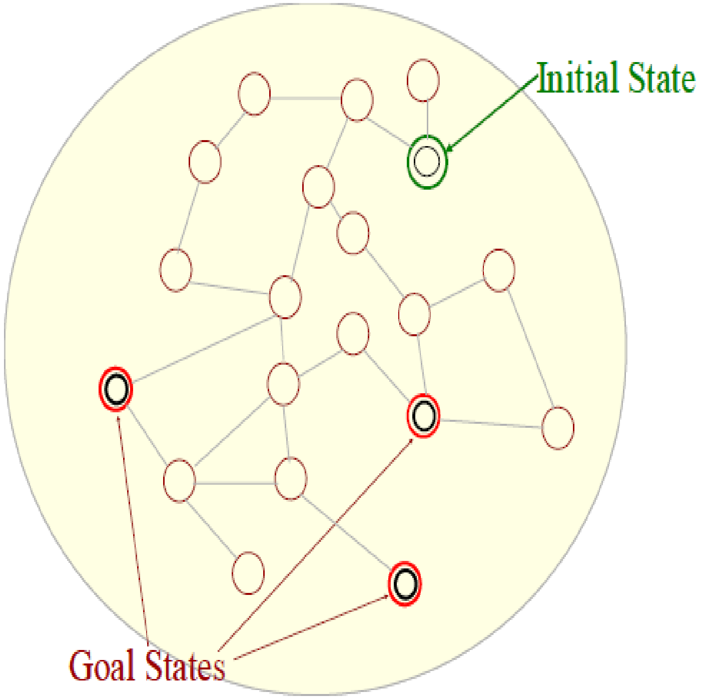
Do until a solution is found or the state space is exhausted.

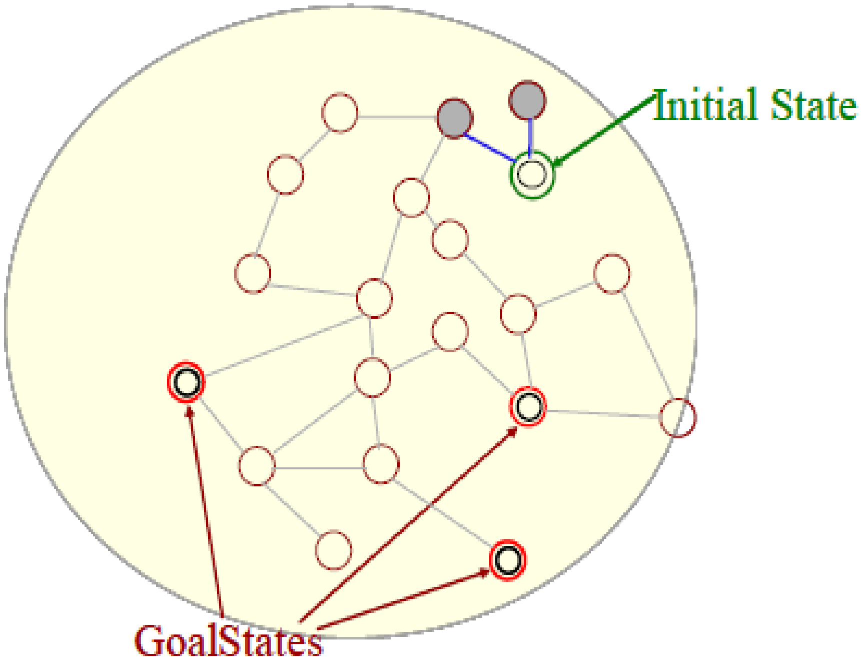
1. Check the current state
2. Execute allowable actions to find the successor states.
3. Pick one of the new states.
4. Check if the new state is a solution state

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

Examples

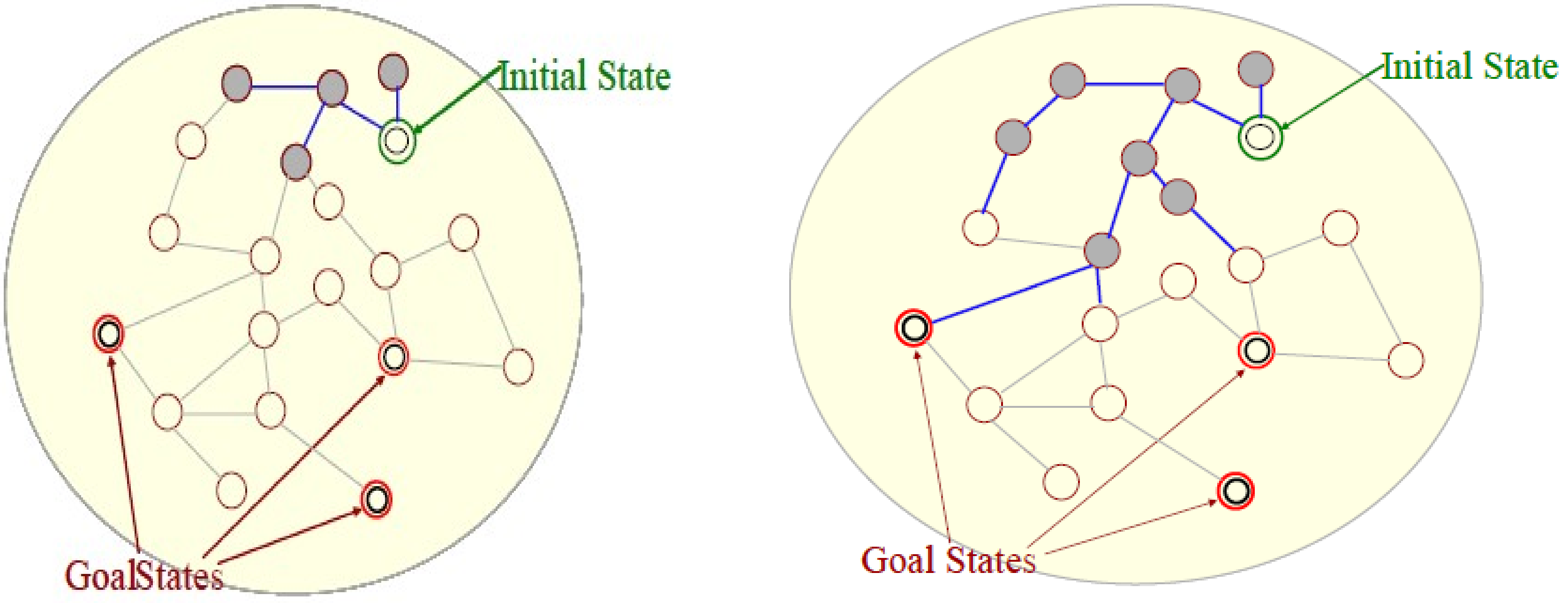
Illustration of a search process





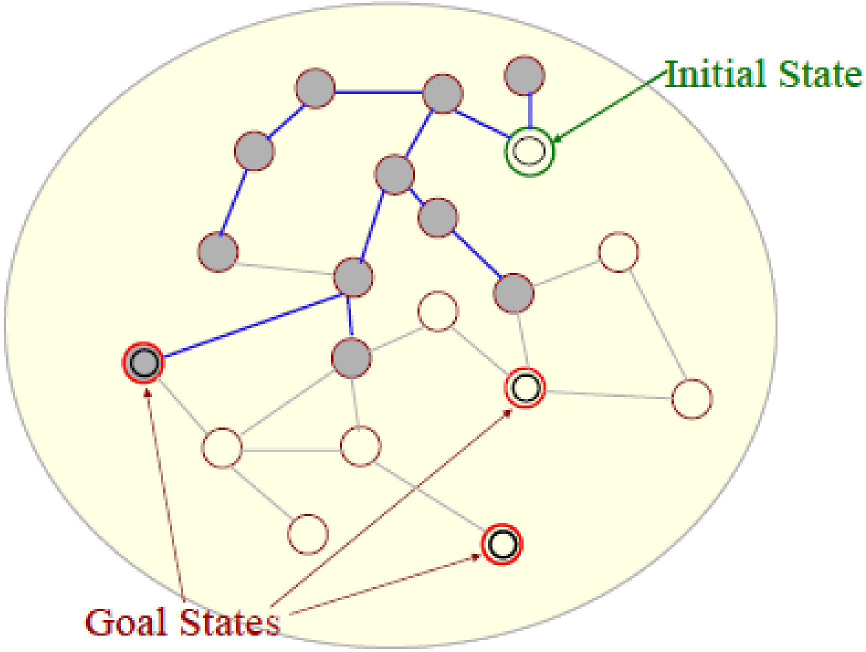
Examples

Illustration of a search process



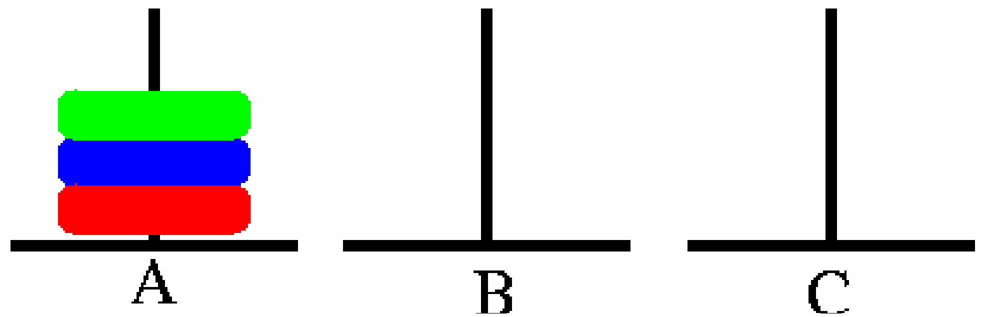
Examples

Illustration of a search process

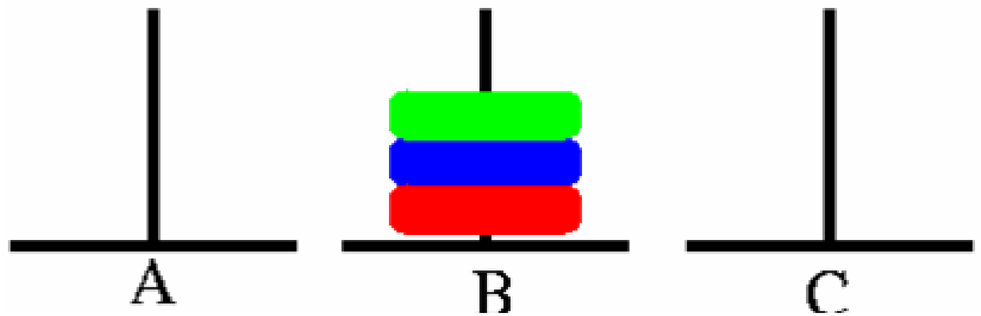


Example problem: Pegs and Disks problem

The initial state



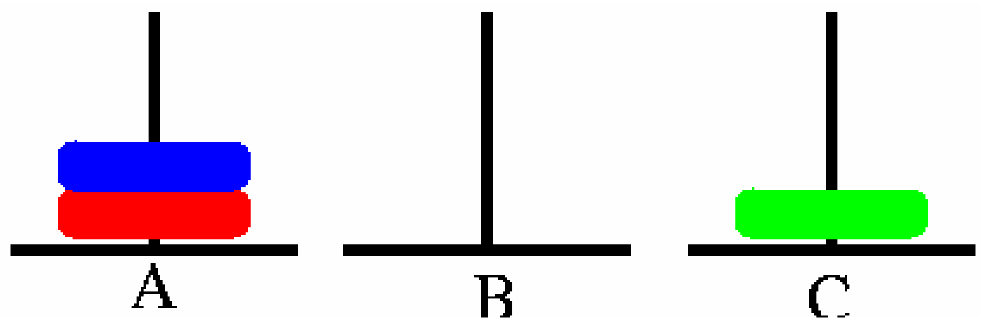
Goal State



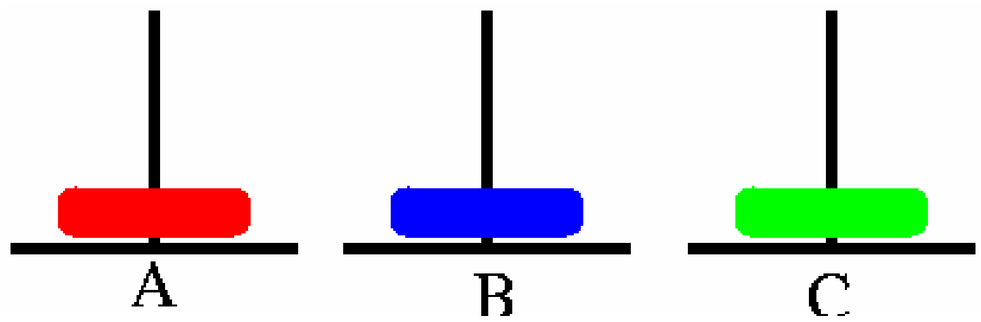
Example problem: Pegs and Disks problem

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

Step 1: Move A → C

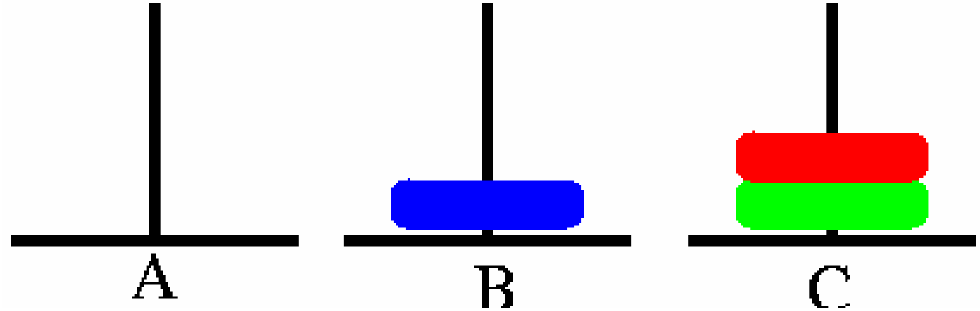


Step 2: Move A → B

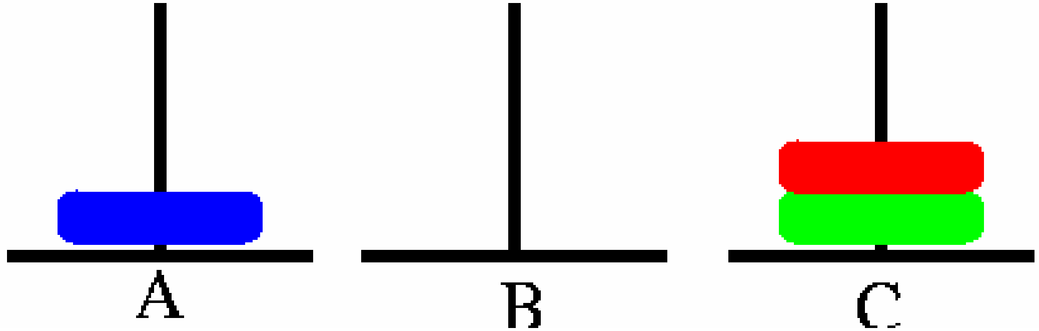


Example problem: Pegs and Disks problem

Step 3: Move A → C

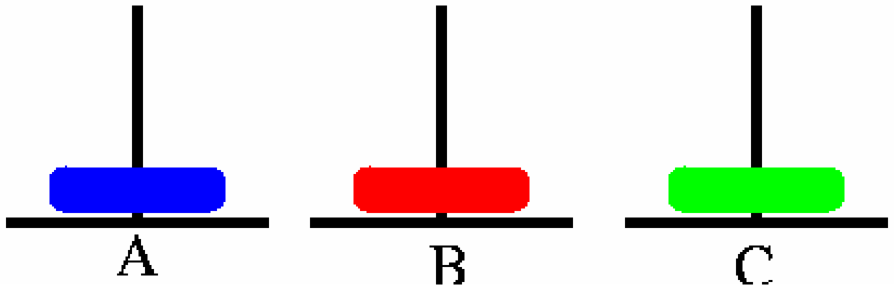


Step 4: Move B→ A

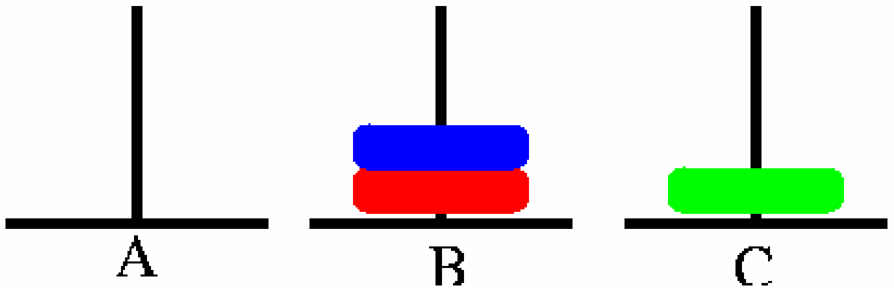


Example problem: Pegs and Disks problem

- Step 5: Move C → B

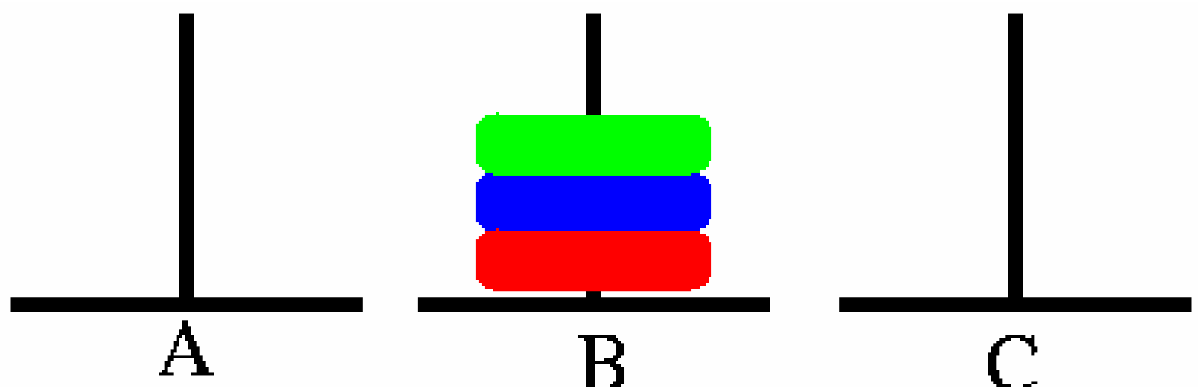


Step 6: Move A → B

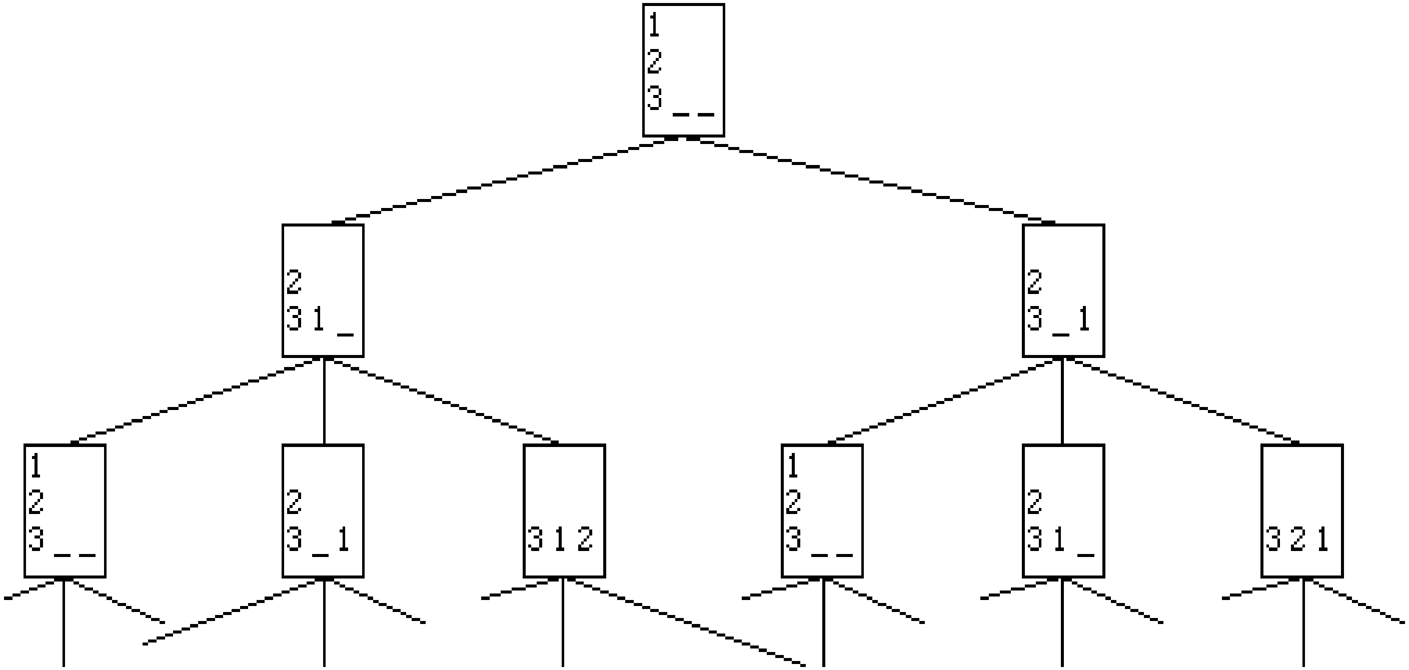


Example problem: Pegs and Disks problem

- 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 ▯ 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.

Search algorithm: Key issues

- How can we handle loops?
- Corresponding to a search algorithm, should we return a path or a node?
- Which node should we select?
- Alternatively, how would we place the newly generated nodes in the fringe?
- Which path to find?

The objective of a search problem is to find a path from the initial state to a goal state.

Our objective could be to find any path, or we may need to find the shortest path or least cost path.

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

The different search strategies that we will consider include the following:

1. Blind Search strategies or Uninformed search
    a. Breadth first search
    b. Depth first search
    c. Depth Limited Search
    d. Iterative deepening search

2. Informed Search
    a. A*

Blind Search

Blind Search

that does not use any extra information about the problem domain. The two common methods of blind search are:

- **BFS or Breadth First Search**
- **DFS or Depth First Search**

Search Tree – Terminology

- **Root Node:** The node from which the search starts.
- **Leaf Node:** A node in the search tree having no children.
- **Ancestor/Descendant:** X is an ancestor of Y is 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.

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.

The search process constructs a search tree, where

- **root** is the initial state and
- **leaf nodes** are nodes
  - not yet expanded (i.e., in fringe) or
  - having no successors (i.e., “dead-ends”)

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

Uninformed Search Strategies

- ***Uninformed*** strategies use only the information available in the problem definition
  - Also known as blind searching

- Breadth-first search
- Depth-first search
- Depth-limited search
- Iterative deepening search

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

- Recall from Data Structures the basic algorithm for a breadth-first search on a graph or tree

- Expand the **shallowest** unexpanded node

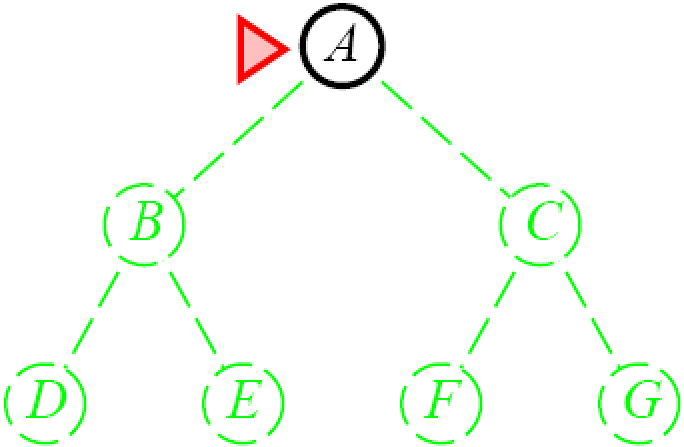
- Place all new successors at the end of a FIFO queue

Breadth First Search

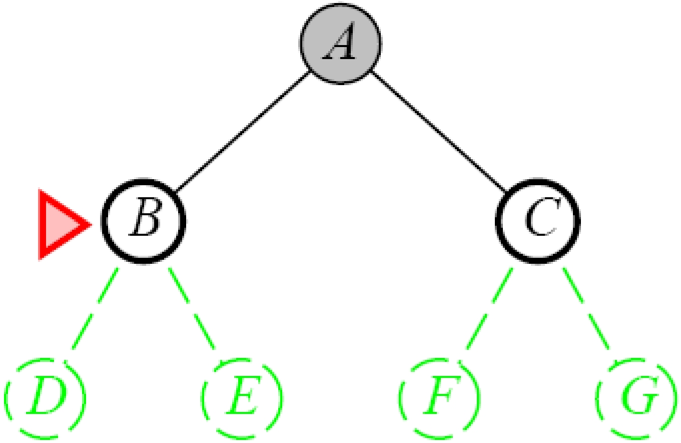
```
Algorithm  Breadth first search
Let fringe be a list containing the initial state
 Loop
          if fringe is empty return failure
                      Node  remove-first (fringe)
          if Node is a goal
                      then return the path from initial state to Node
                                  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.

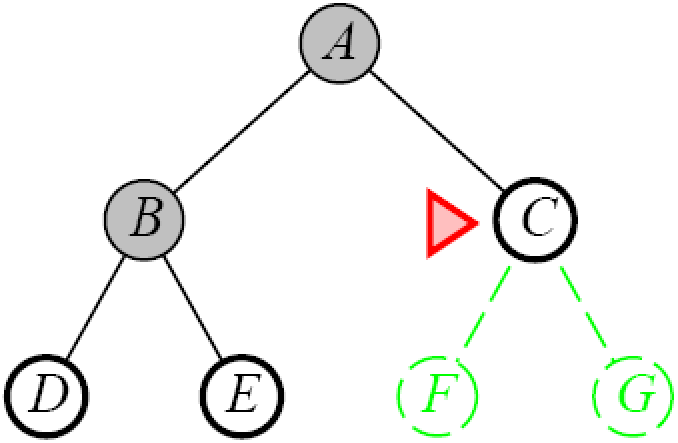
Breadth-First Search



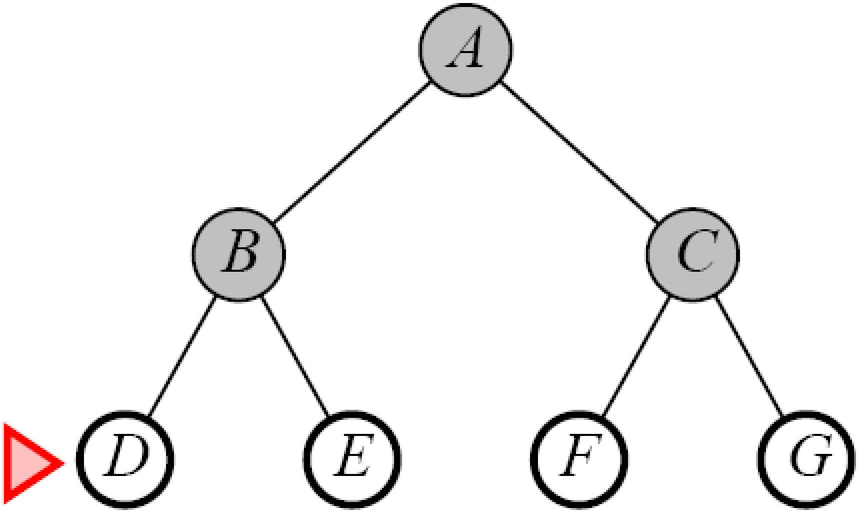
Breadth-First Search



Breadth-First Search

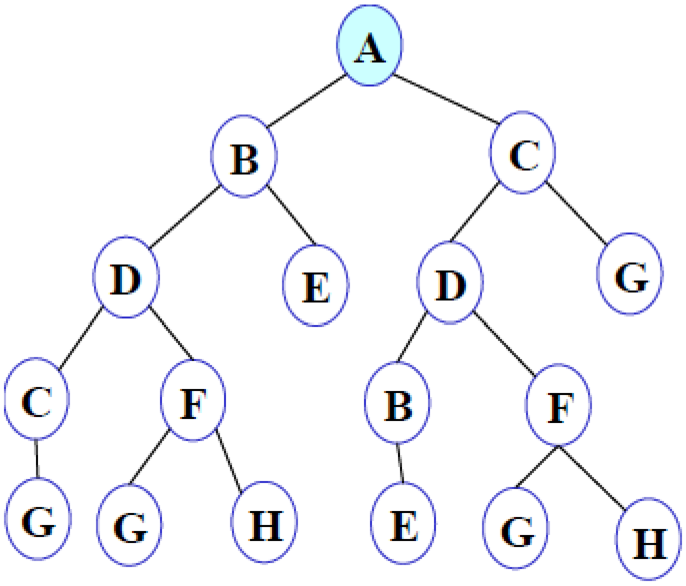


Breadth-First Search



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.

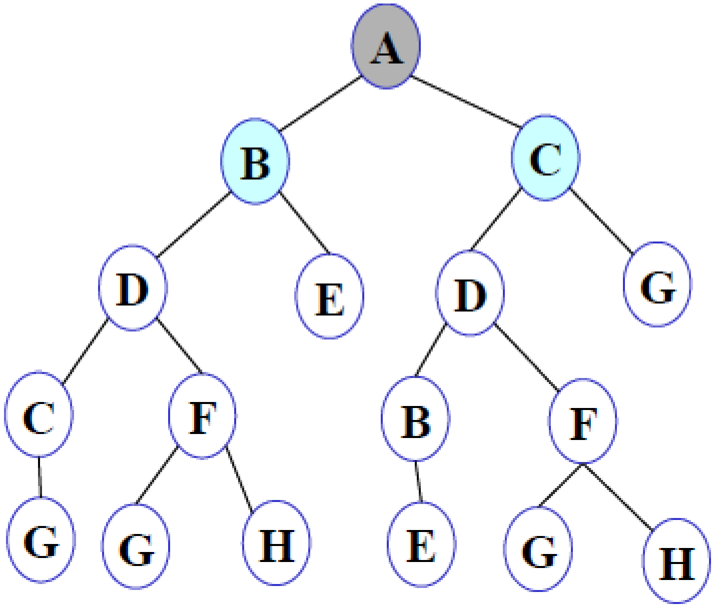


Figure 4

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

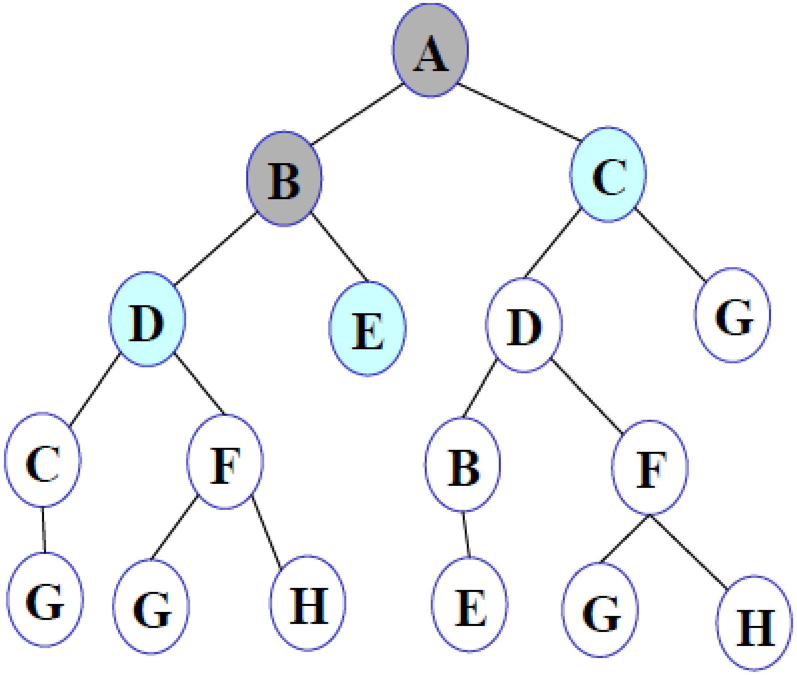


Figure 5

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.

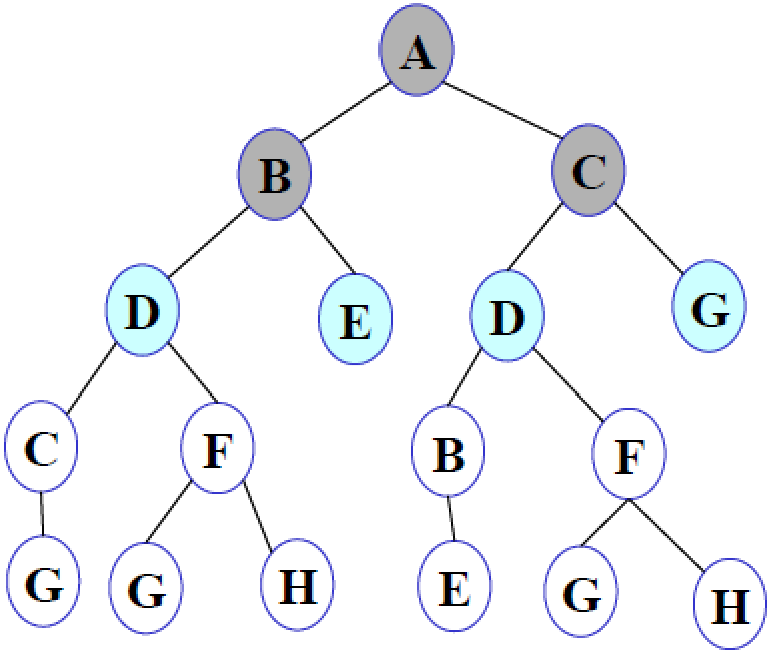
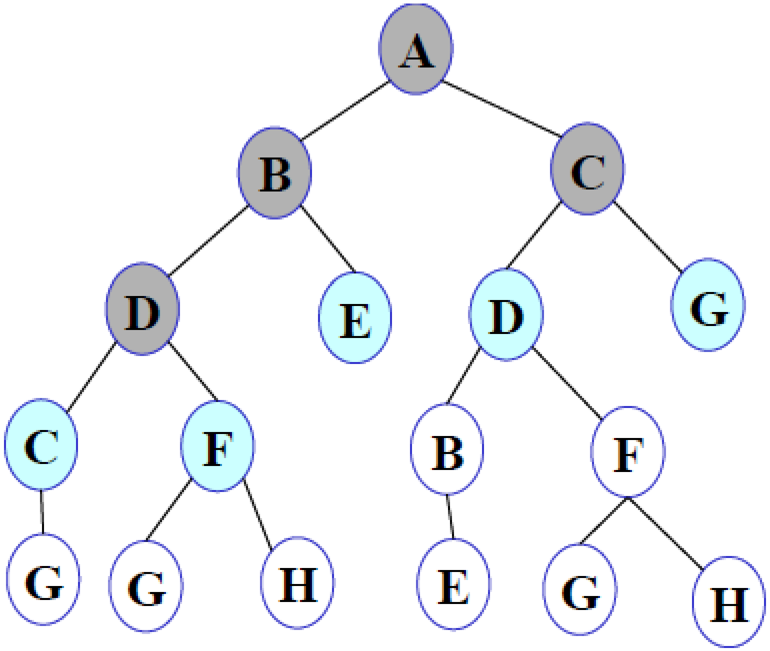


Figure 6

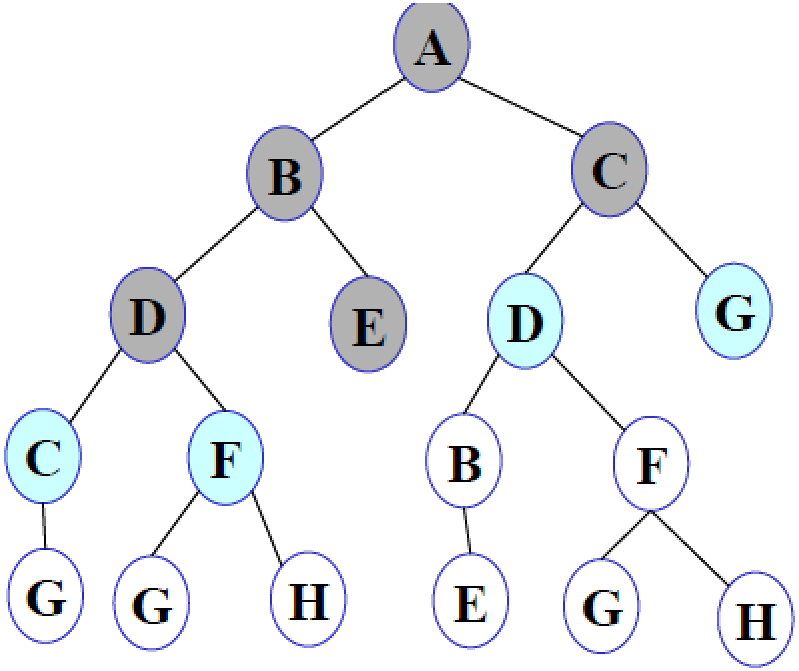
FRINGE: D E D G

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.

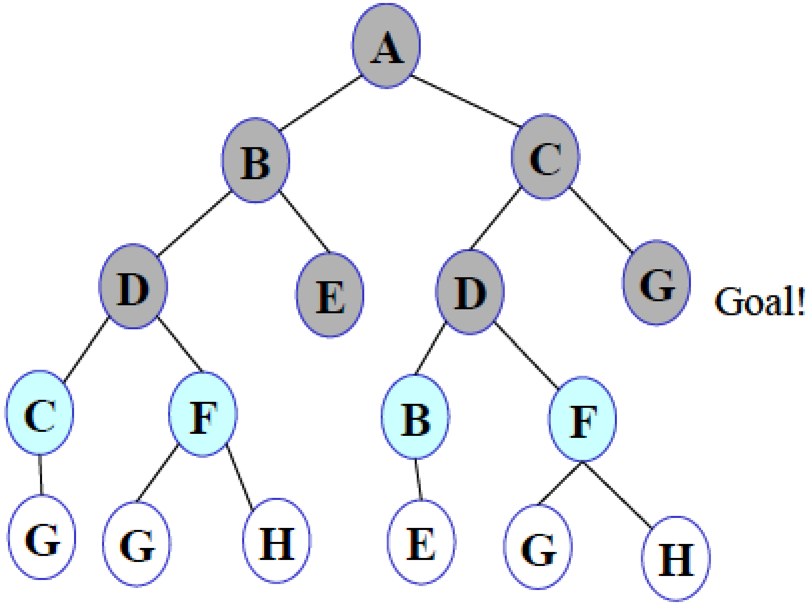
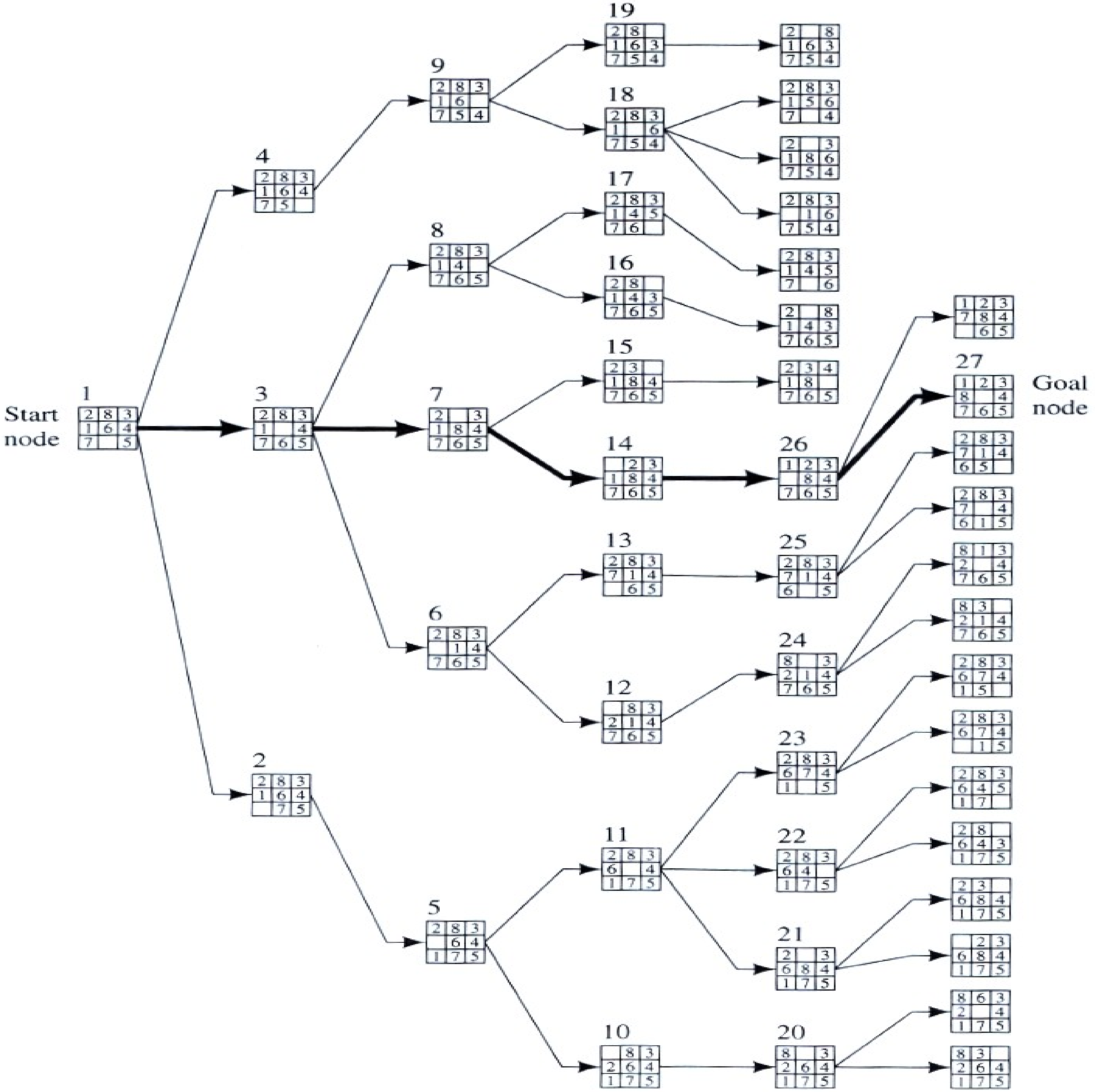


Figure 8

FRINGE: G C F B F

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

Example BFS

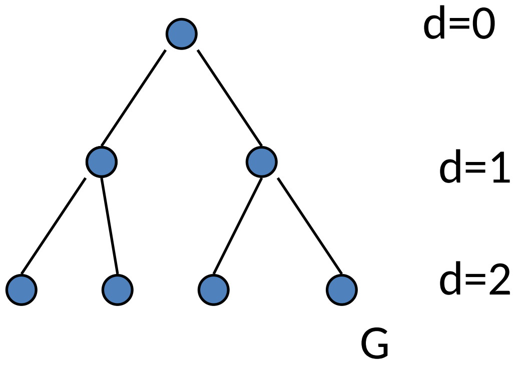

Start node
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
Goal node

What is the Complexity of Breadth-First Search?

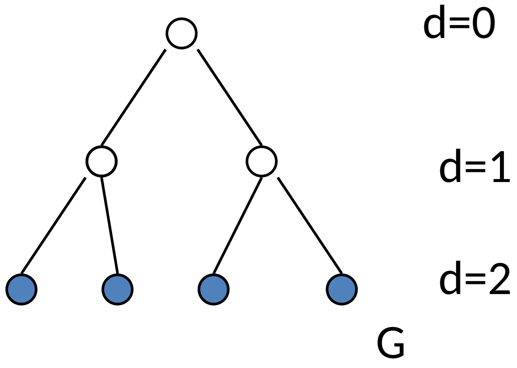
- **Time Complexity**
  - assume (worst case) that there is 1 goal leaf at the RHS
  - so BFS will expand all nodes

$= 1 + b + b^2 + \quad .... + b^d$

$= \mathbf{O\ (b^d)}$



- **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 = $\mathbf{O\ (b^d)}$



Advantages & Disadvantages of Breadth First Search

Advantages of Breadth First Search

Finds the path of minimal length to the goal.

Disadvantages of Breadth First Search

Requires the generation and storage of a tree whose size is exponential the the depth of the shallowest goal node

Properties of Breadth-First Search

- Complete
  - Yes if b (max branching factor) is finite
- Time
  - $1 + b + b^2 + \ldots + b^d = O(b^d)$
  - exponential in d
- Space
  - $O(b^d)$
  - Keeps every node in memory
  - This is the big problem; an agent that generates nodes at 10 MB/sec will produce 860 MB in 24 hours
- Optimal
  - Yes (if cost is 1 per step); not optimal in general

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 uniformed methods

Depth-First Search

- Recall from Data Structures the basic algorithm for a depth-first search on a graph or tree

- Expand the ***deepest*** unexpanded node

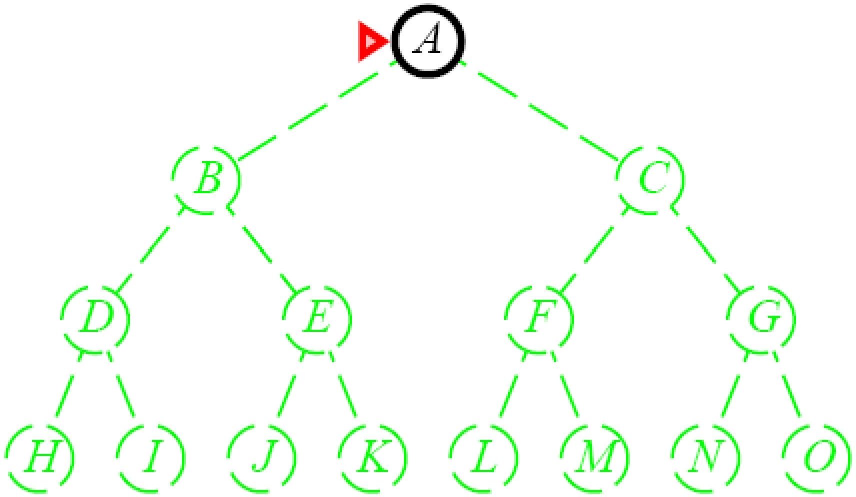
- Unexplored successors are placed **on a stack** until fully explored

Depth first Search

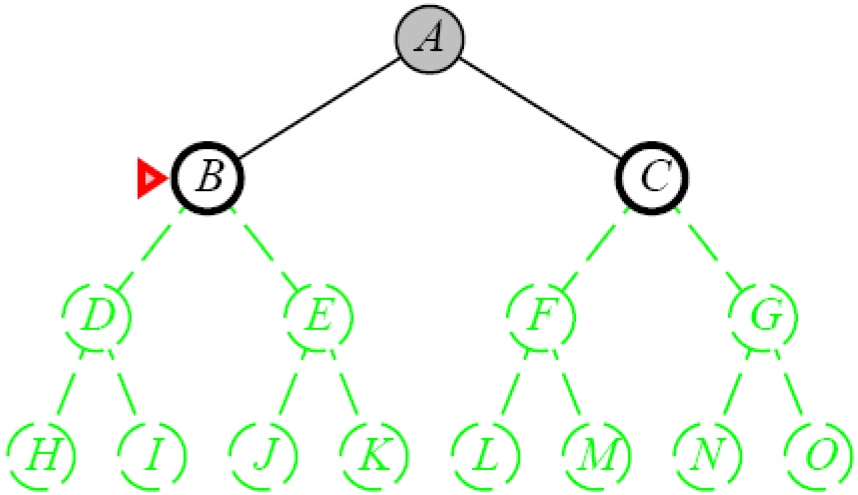
Algorithm

```
Let fringe be a list containing the initial state
Loop
          if fringe is empty return failure
          Node□ 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
```

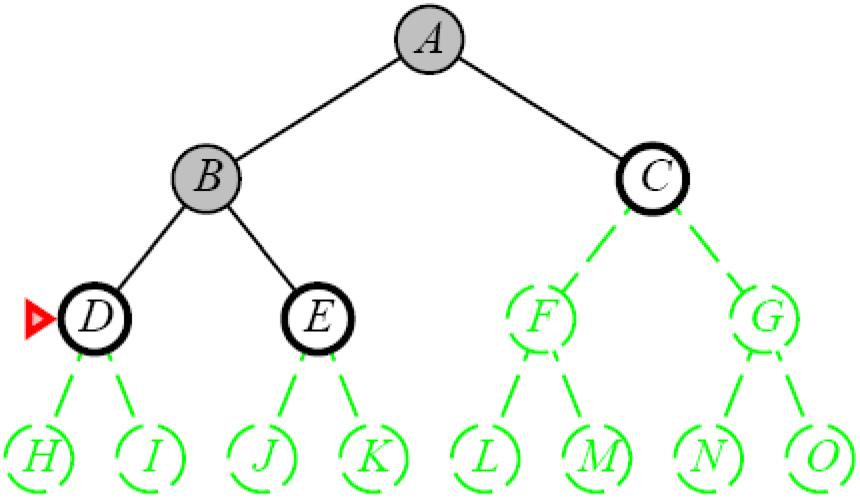
Depth-First Search



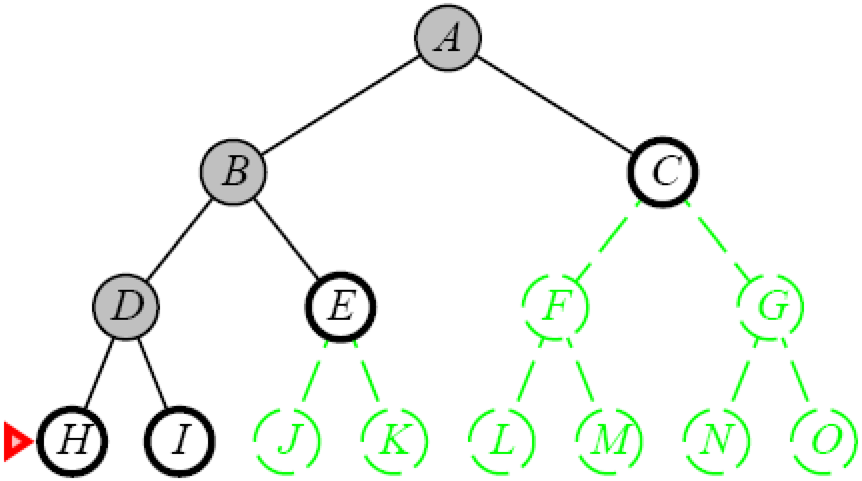
Depth-First Search



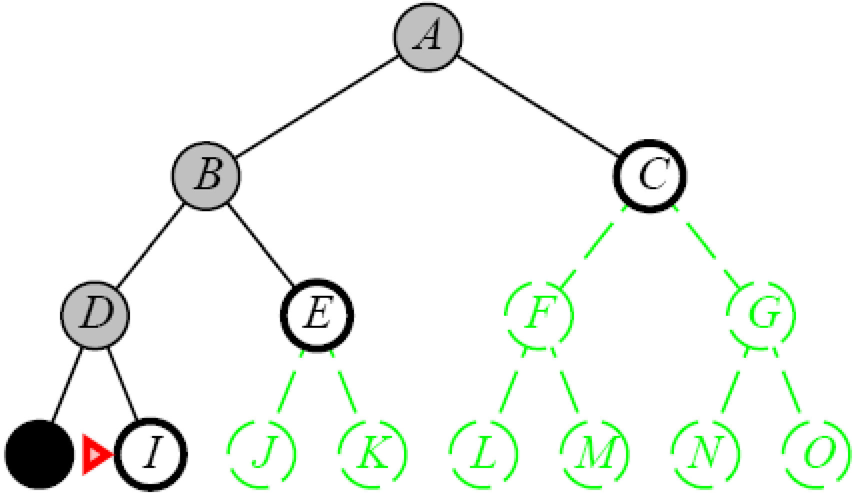
Depth-First Search



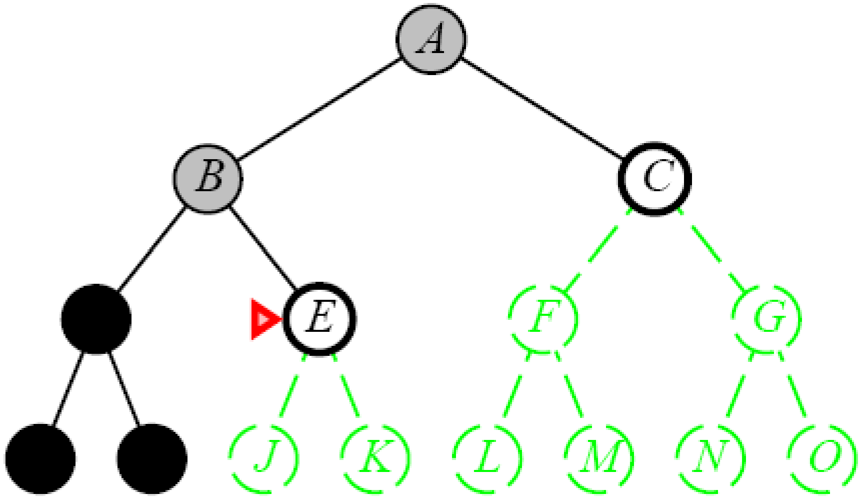
Depth-First Search



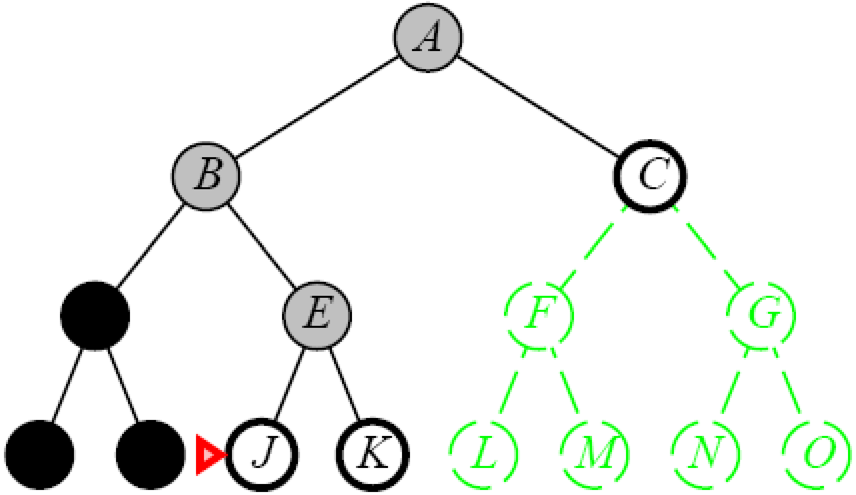
Depth-First Search



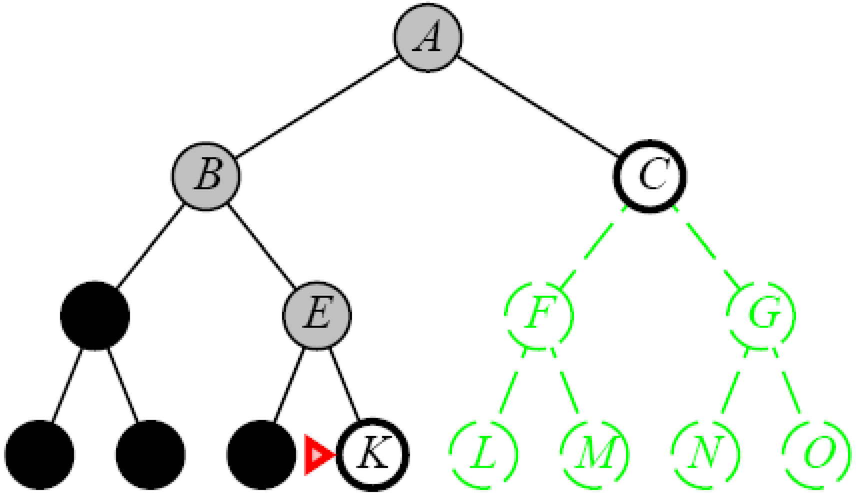
Depth-First Search



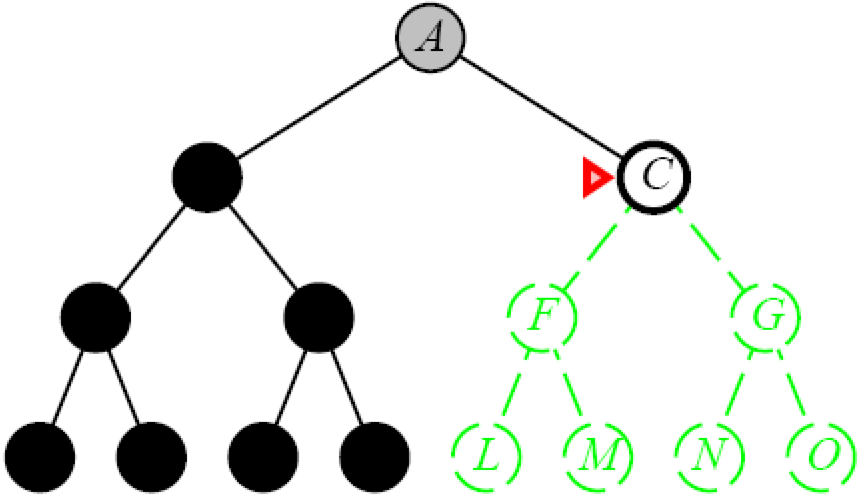
Depth-First Search



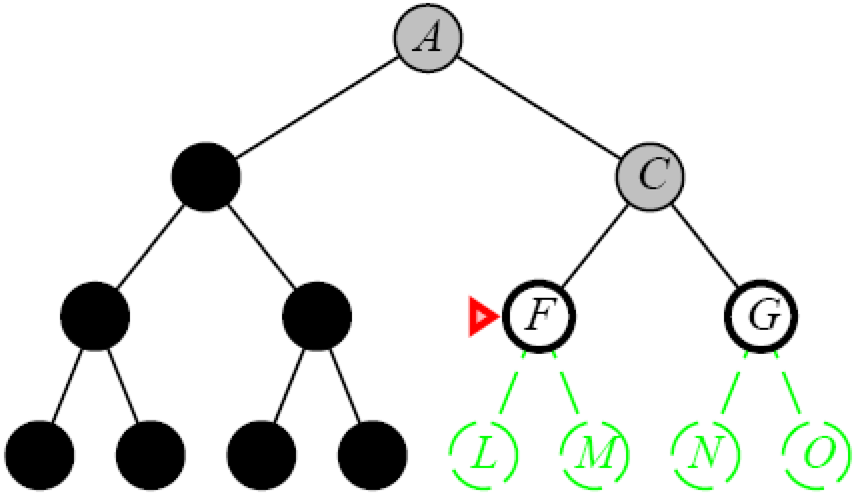
Depth-First Search



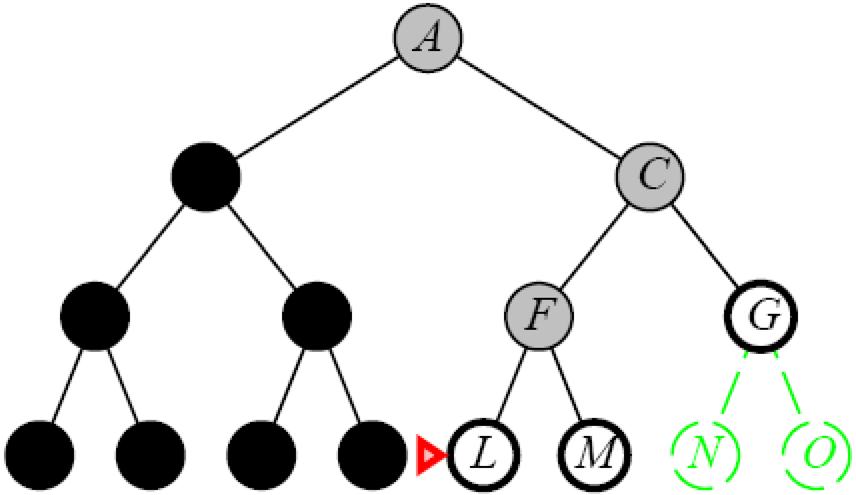
Depth-First Search



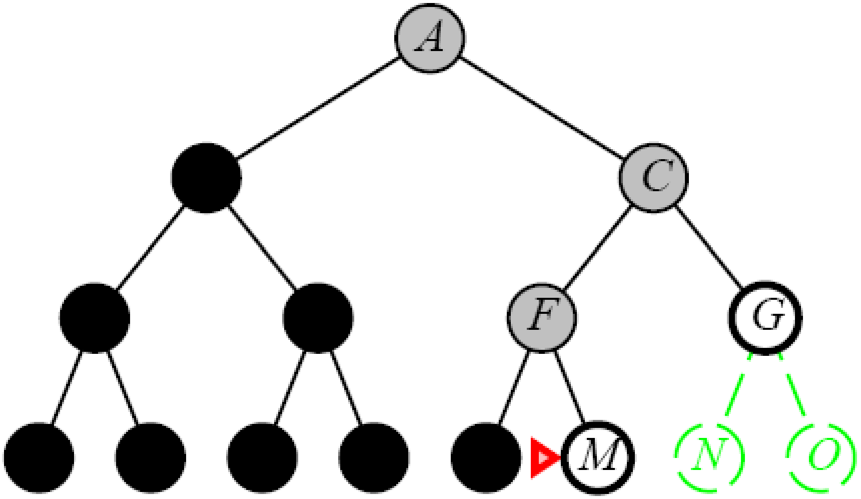
Depth-First Search



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.

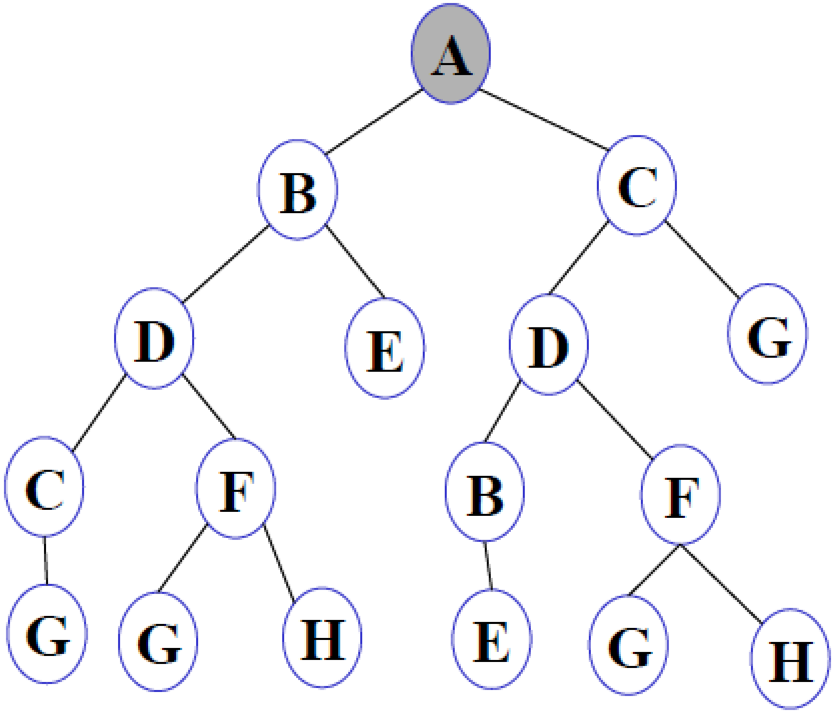
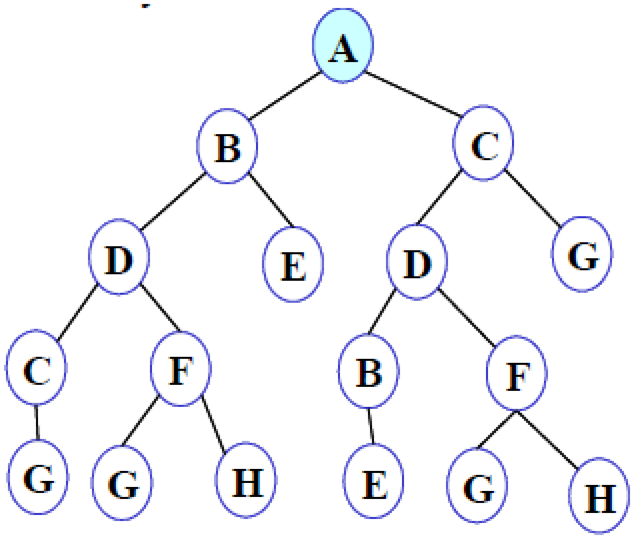


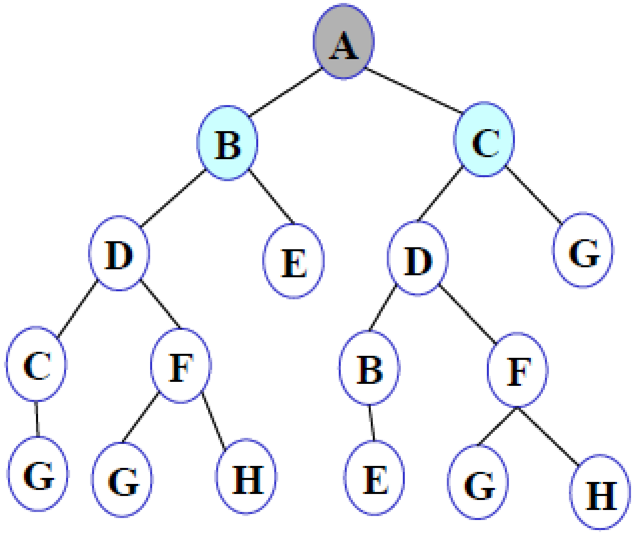
Figure 11: Search tree for the state space

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



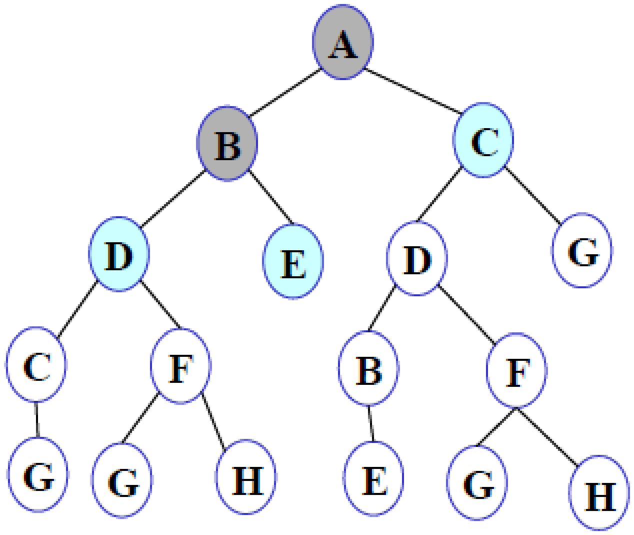
FRINGE: A

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



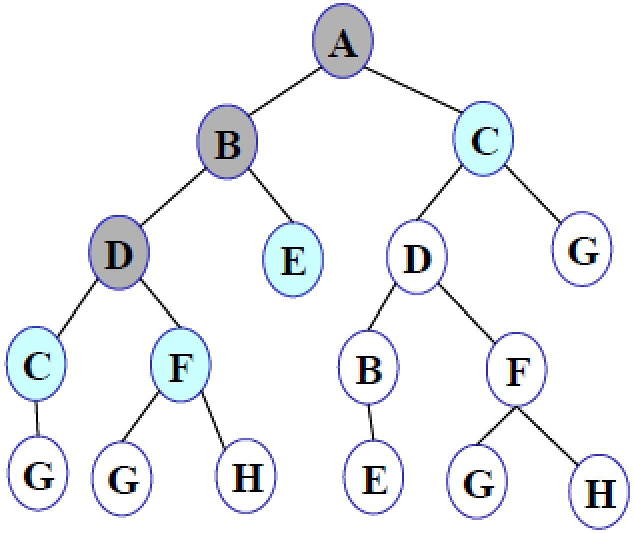
FRINGE: B C

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



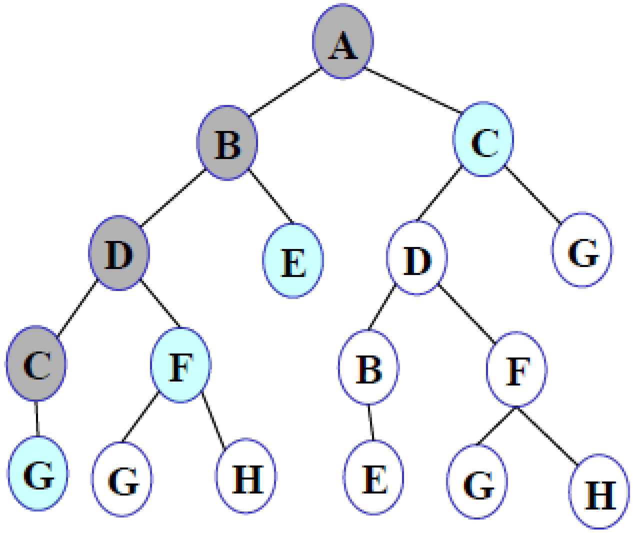
FRINGE: D E C

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



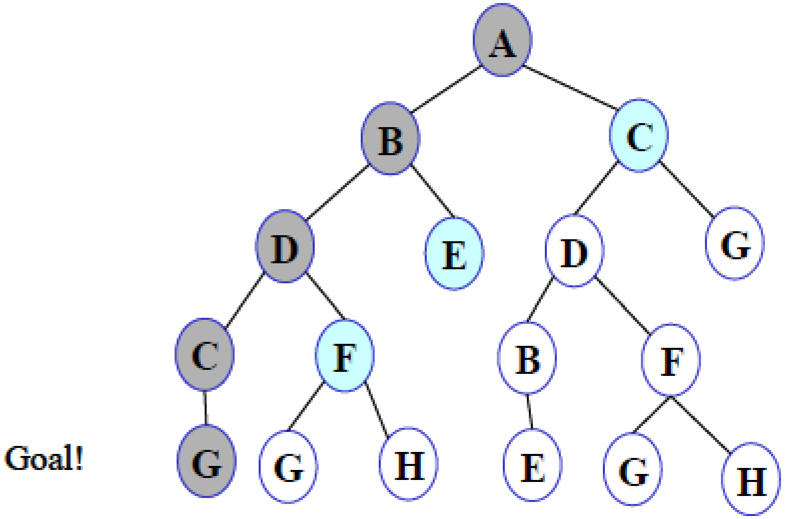
FRINGE: C F E C

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



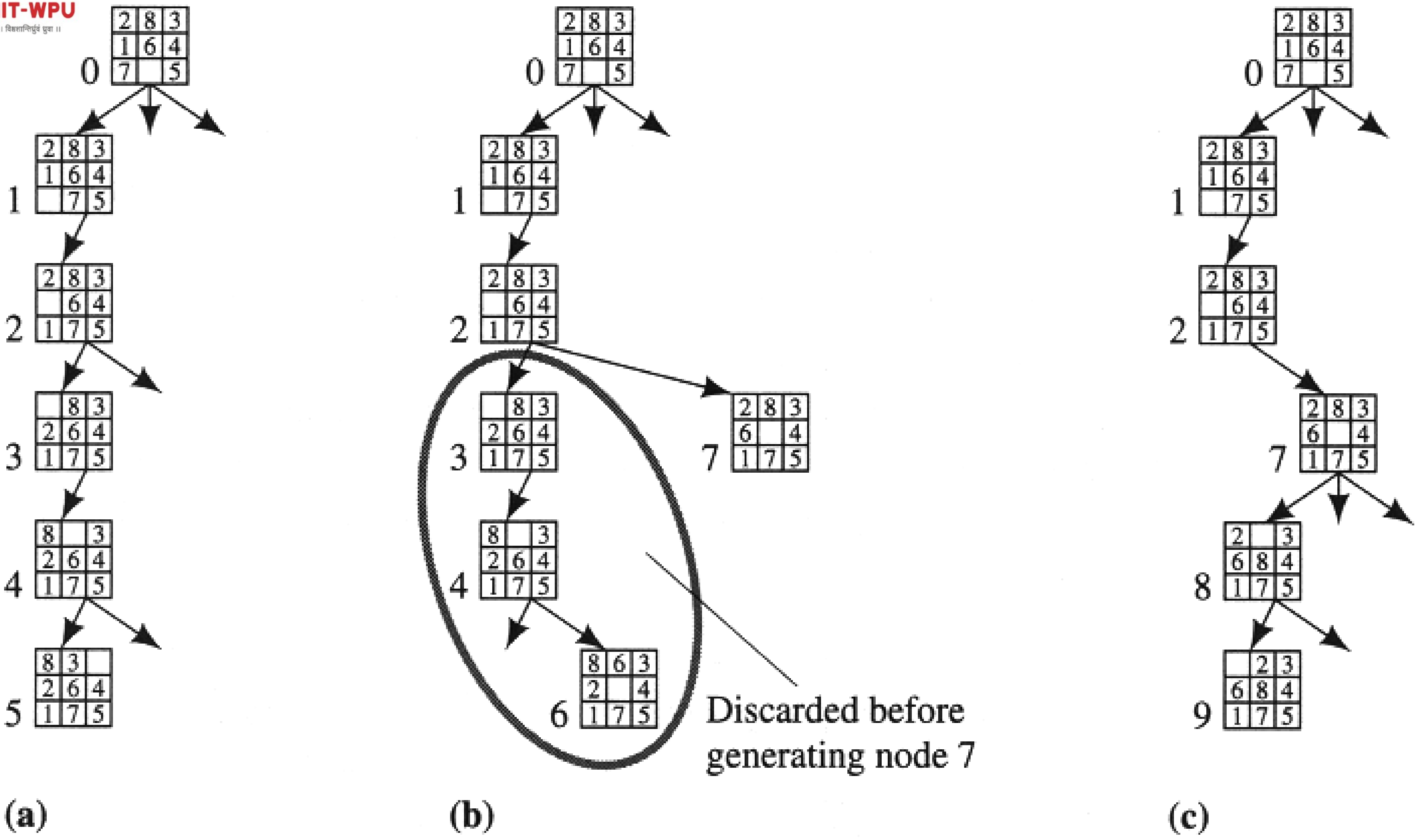
FRINGE: G F E C

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.

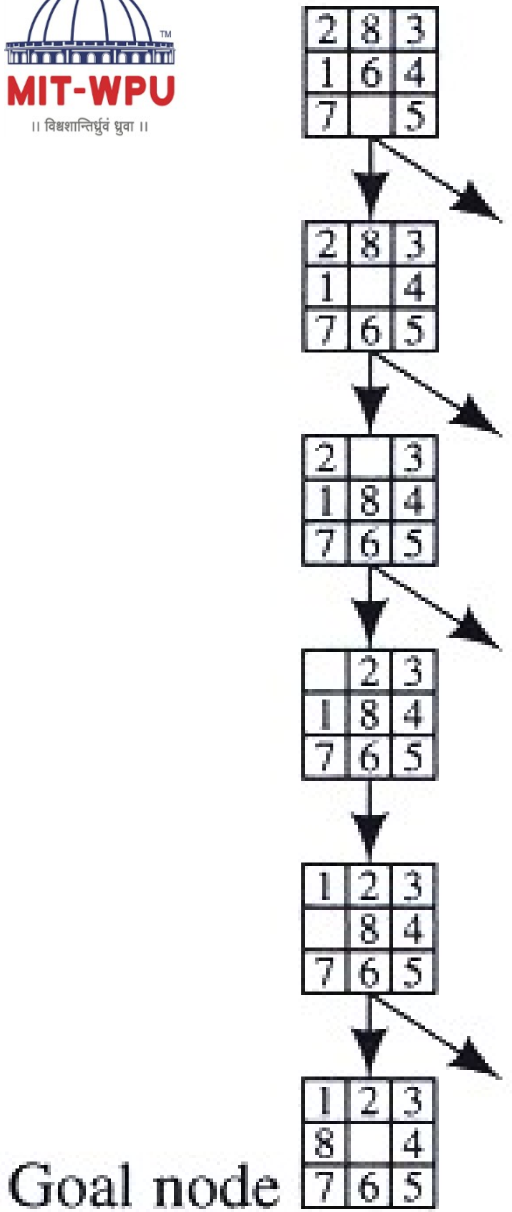


FRINGE: G F E C

Depth-First Search



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

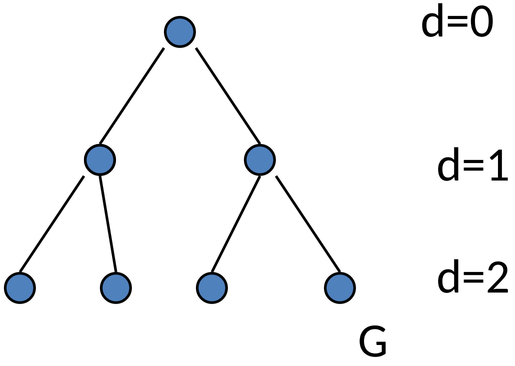

2 8 3
1 6 4
7 _ 5
2 8 3
1 _ 4
7 6 5
2 _ 3
1 8 4
7 6 5
_ 2 3
1 8 4
7 6 5
1 2 3
_ 8 4
7 6 5
1 2 3
8 _ 4
7 6 5
Goal node

What is the Complexity of Depth-First Search?

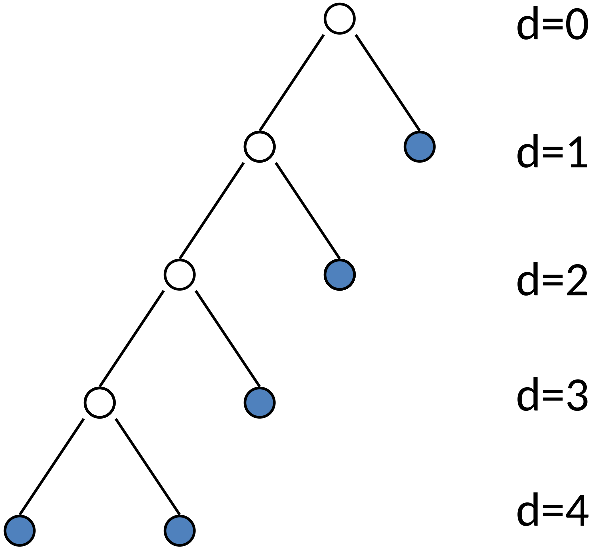
- Time Complexity
  - assume (worst case) that there is 1 goal leaf at the RHS
  - so DFS will expand all nodes

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

$= \mathbf{O\ (b^d)}$



- 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

- Complete
  - No: fails in infinite-depth spaces, spaces with loops
    - Modify to avoid repeated spaces along path
  - Yes: in finite spaces
- Time
  - O(b^d)
  - Not great if m is much larger than d
  - But if the solutions are dense, this may be faster than breadth-first search
- Space
  - O(bd)…linear space
- Optimal
  - No

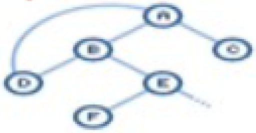
Depth-first vs. Breadth-first

Advantages of depth-first :

- **Simple** to implement;
- Needs relatively **small memory** for storing the state-space.

Disadvantages of depth-first :

- Sometimes fail to find a solution (may be get stuck in an infinite long branch) - **not *complete***;



- **Not** guaranteed to find an ***optimal*** solution (may not find the shortest path solution);
- Can take a lot **longer** to find a solution.

Advantages of breadth-first :

- Guaranteed to find a solution (if one exists) - ***complete***;
- Depending on the problem, can be guaranteed to find an ***optimal*** solution.

Disadvantages of breadth-first :

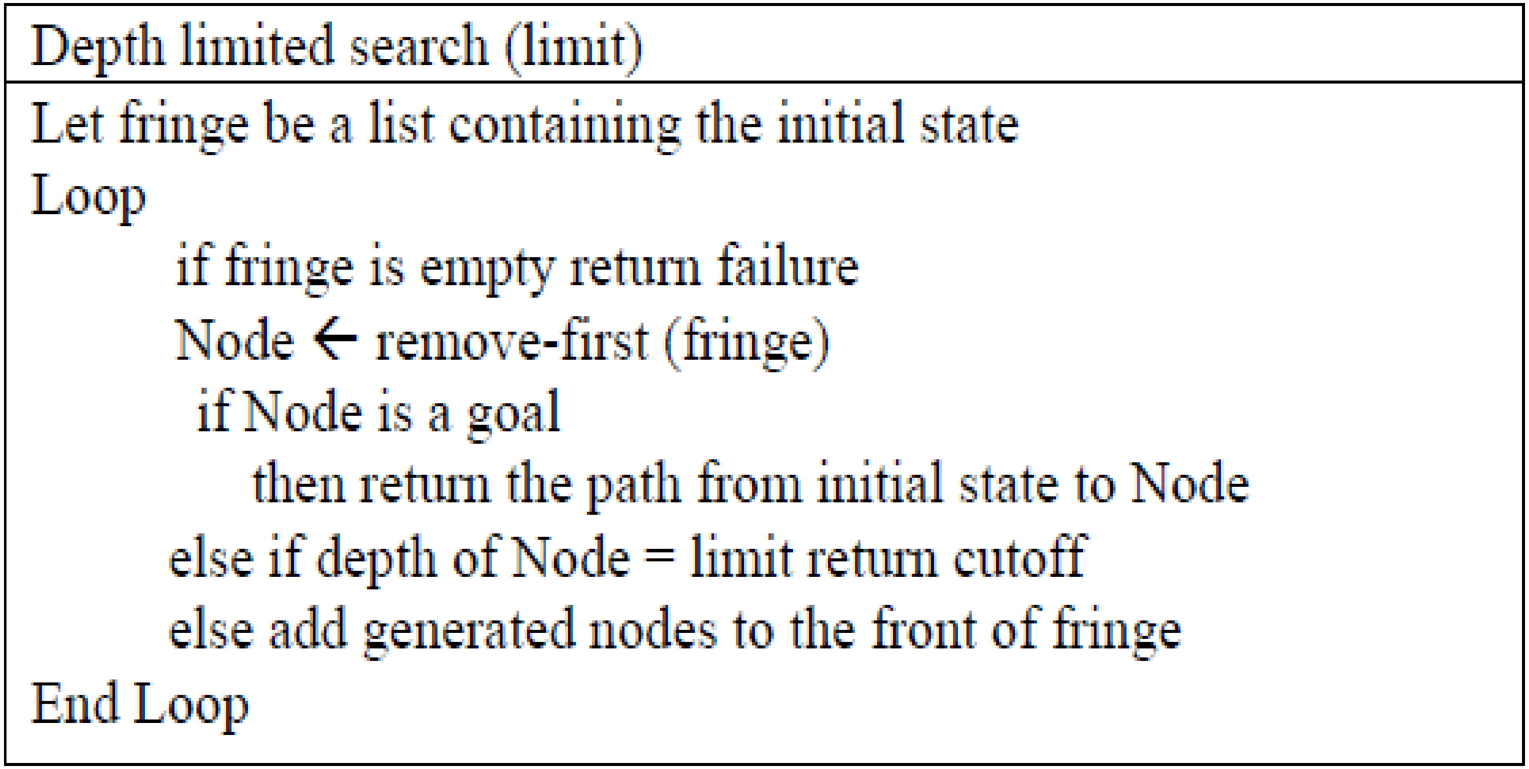
- More **complex** to implement;
- Needs a **lot of memory** for storing the state space if the search space has a high branching factor.

Depth-Limited Search

- A variation of depth-first search that uses a depth limit
  - Alleviates the problem of unbounded trees
  - Search to a predetermined depth **`l`** (“ell”)
  - Nodes at depth **`l`** have no successors

- Same as depth-first search if **`l`** = ∞
- Can terminate for failure and cutoff
- The time and space complexity of depth-limited search is similar to depth-first search.

Depth-Limited Search



```
Depth limited search (limit)
Let fringe be a list containing the initial state
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
         if Node is a goal
            then return the path from initial state to Node
        else if depth of Node = limit return cutoff
        else add generated nodes to the front of fringe
End Loop
```

Depth-Limited Search

- Complete
  - Yes if l > d
- Time
  - $O(b^l)$
- Space
  - $O(bl)$
- Optimal
  - No if l < d

Uninformed : Iterative Deepening Search(IDS)

- Key idea: Iterative deepening search (IDS) applies DLS repeatedly with increasing depth. It terminates when a solution is found or no solutions exists.

- IDS combines the benefits of BFS and DFS: Like DFS the memory requirements are very modest (O(bd)). Like BFS, it is complete when the branching factor is finite.

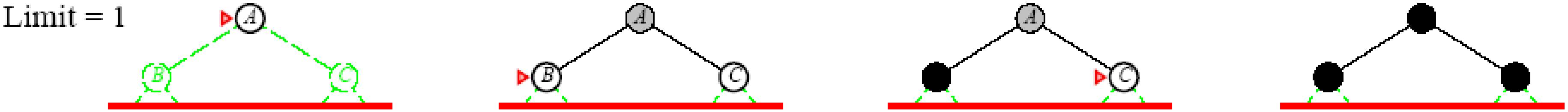
- The total number of generated nodes is :

  - $N(\text{IDS})=(d)b + (d\text{-}1)\, b^2 + \ldots+(1)b^d$

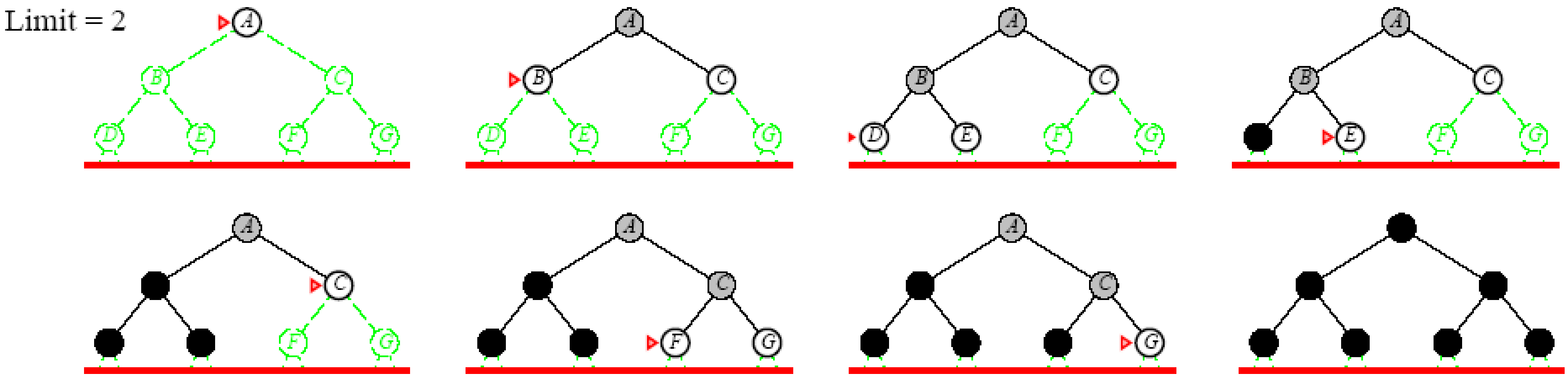
- In general, iterative deepening is the preferred Uninformed search method when there is a large search space and the depth of the solution is not known.

Iterative Deepening Search

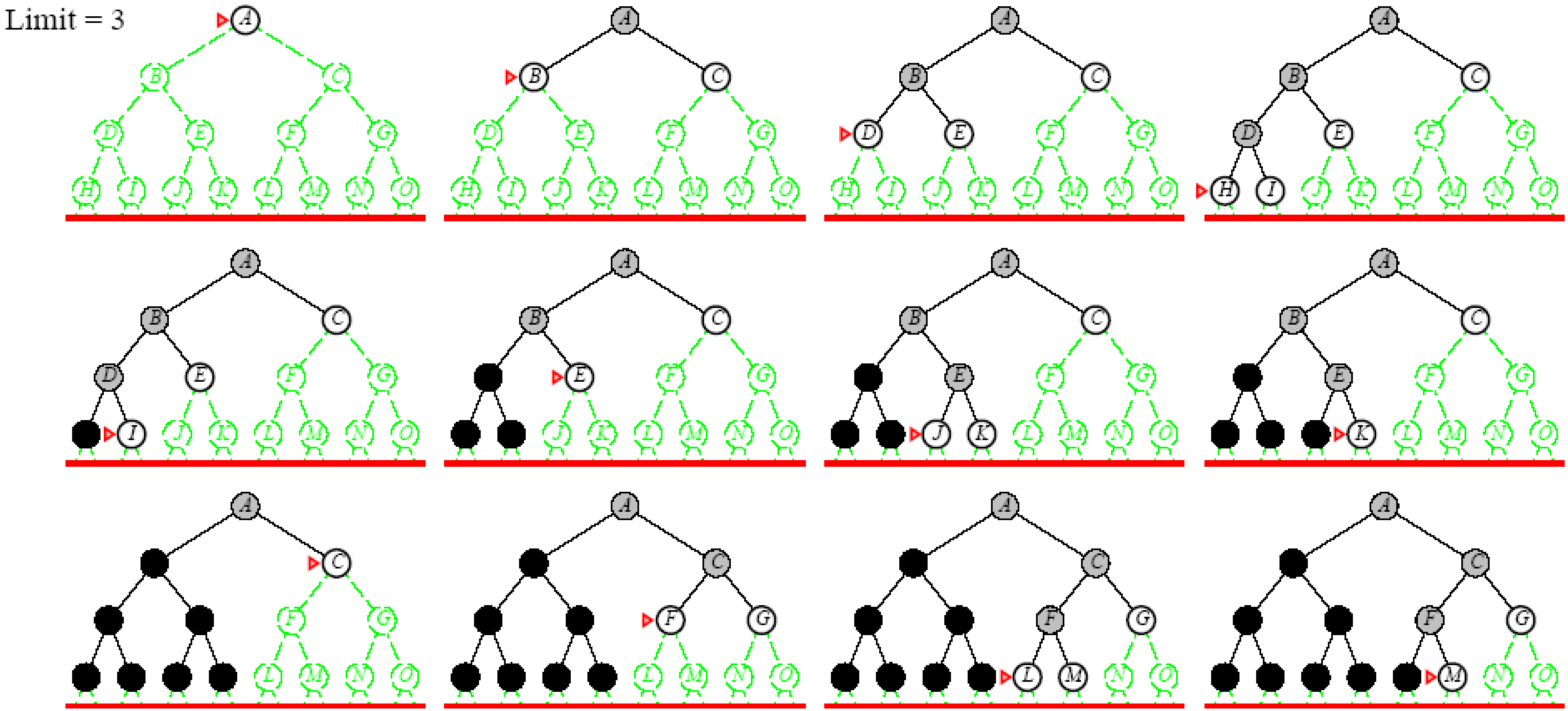
Iterative Deepening Search



Iterative Deepening Search



Iterative Deepening Search



Iterative Deepening Search

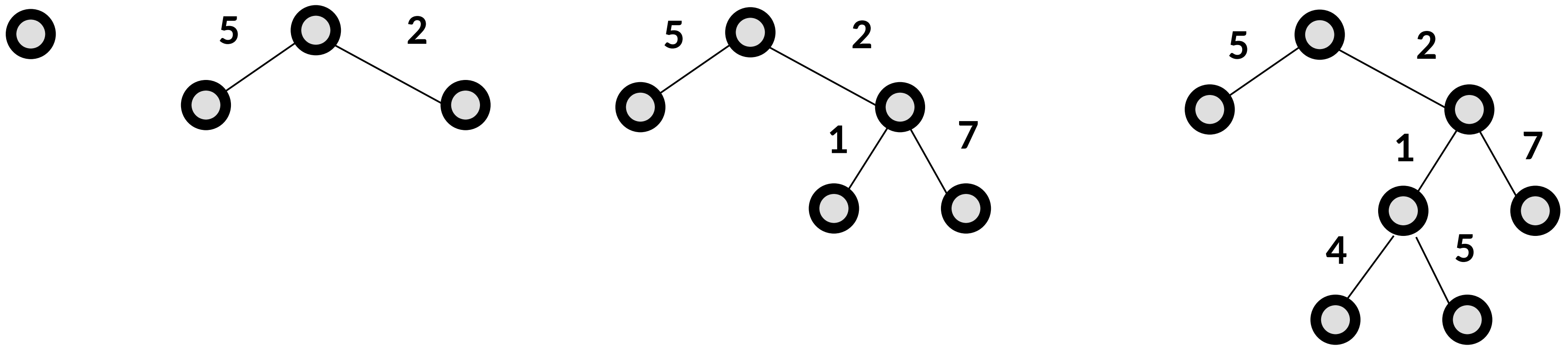
- Complete
  - Yes
- Time
  - $O(b^d)$
- Space
  - O(bd)
- Optimal
  - Yes if step cost = 1
  - Can be modified to explore uniform cost tree

5. Uniform-cost search

- This algorithm is by Dijkstra [1959]
- Used for weighted tree
- The Goal of UCS is to find the path to the goal node which is the lowest cumulative cost
- The algorithm expands nodes in the order of their cost from the source.
- The path cost is usually taken to be the sum of the step costs.
- In uniform cost search the newly generated nodes are put in OPEN according to their path costs.
- 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

Enqueue nodes in order of cost



Intuition: Expand the cheapest node. Where the cost is the path cost g(n)

- Complete? Yes.
- Optimal? Yes,
- Time Complexity: $O(b^d)$
- Space Complexity: $O(b^d)$

Note that Breadth First search can be seen as a special case of Uniform Cost Search, where the path cost is just the depth.

Uniform Cost Search in Tree

```
fringe □ MAKE-EMPTY-QUEUE()
fringe □ INSERT( root_node ) // with g=0
loop {
    if fringe is empty then return false // finished without goal
    node □ REMOVE-SMALLEST-COST(fringe)
    if node is a goal
        print node and g
        return true // that found a goal
    $L_g$ □ EXPAND(node) // $L_g$ is set of children with their g costs
                         // NOTE: do not check $L_g$ for goals here!!
    fringe □ INSERT-ALL($L_g$, fringe )
}
```

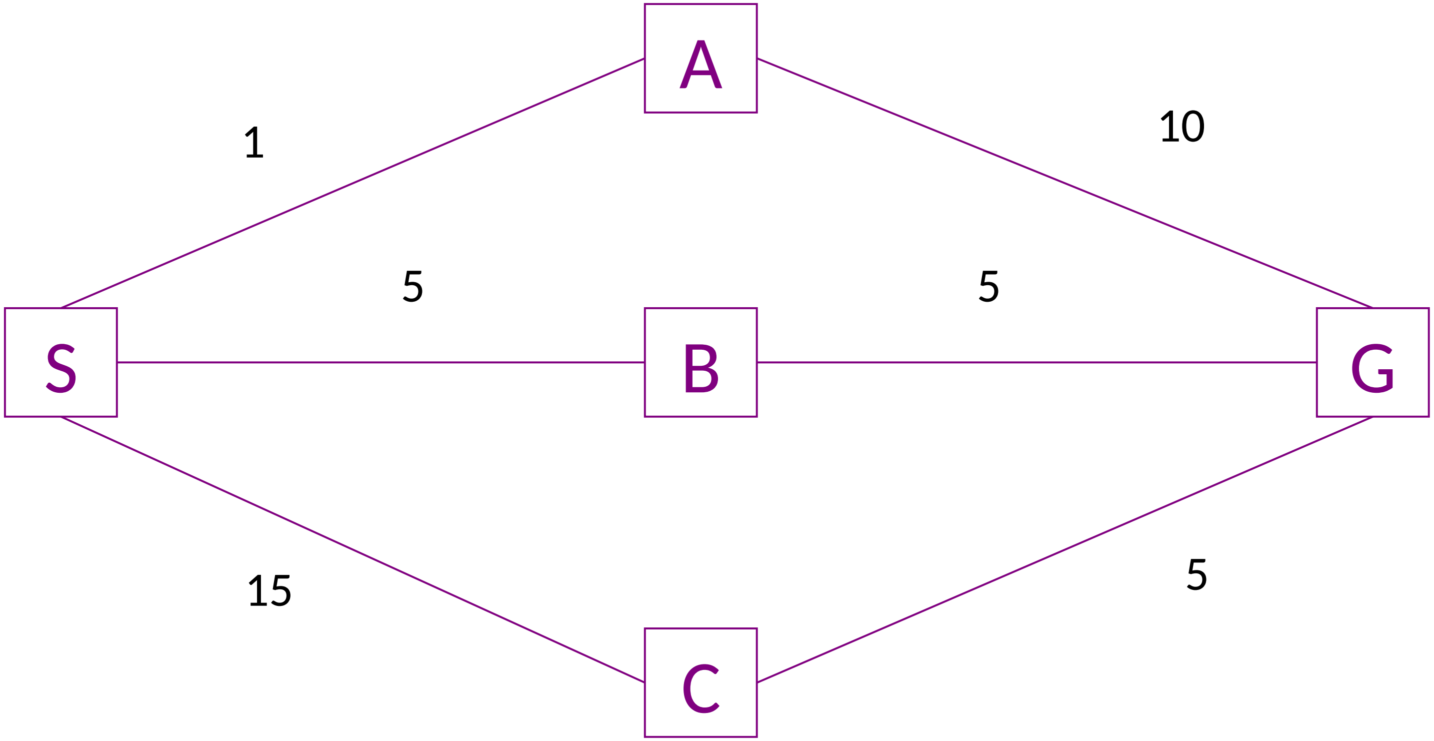
Uniform cost search

- A breadth-first search finds the shallowest goal state and will therefore be the cheapest solution provided the *path cost is a function of the depth of the solution*. But, if this is not the case, then breadth-first search is not guaranteed to find the best (i.e. cheapest solution).

- Uniform cost search remedies this by expanding the lowest cost node on the fringe, where cost is the path cost, *g(n)*.

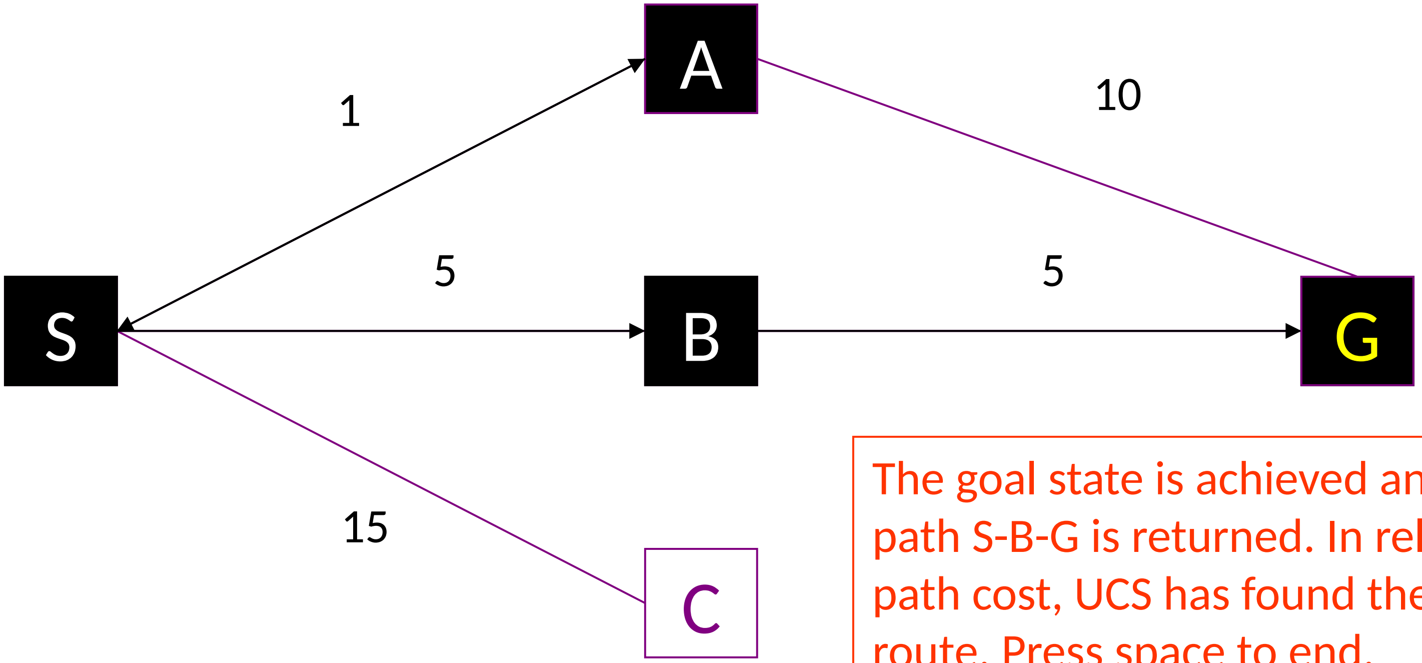
- In the following slides those values that are attached to paths are the cost of using that path.

Consider 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...

[illegible]. The
[illegible] case
[illegible]
[illegible] proceed to goal state. Press space.



The goal state is achieved and the path S-B-G is returned. In relation to path cost, UCS has found the optimal route. Press space to end.

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 (UCS)

- Let g(n) = cost of the path from the start node to an open node n

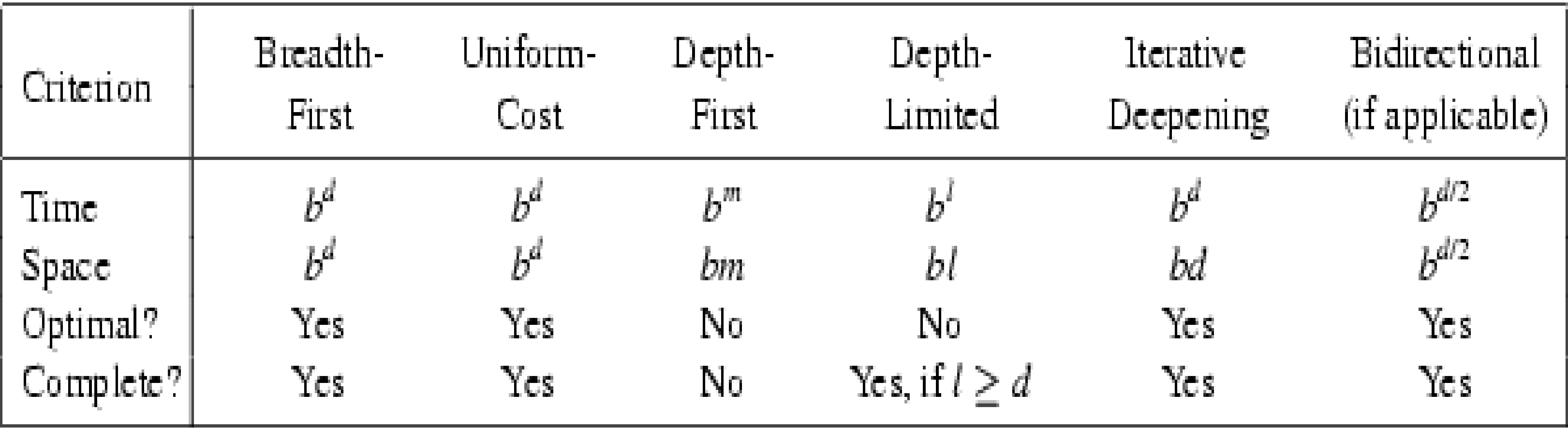
- Algorithm outline:
  - Always select from the OPEN the node with the least g(n) value for expansion, and put all newly generated nodes into OPEN
  - Nodes in OPEN are sorted by their g(n) values (in ascending order)
  - Terminate if a node selected for expansion is a goal

- Called " Dijkstra's Algorithm " in the algorithms literature and similar to " Branch and Bound Algorithm " in operations research literature

Uniform-Cost (UCS)

- **It is complete** (if cost of each action is not infinitesimal)
  - The total # of nodes n with g(n) <= g(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 it's 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



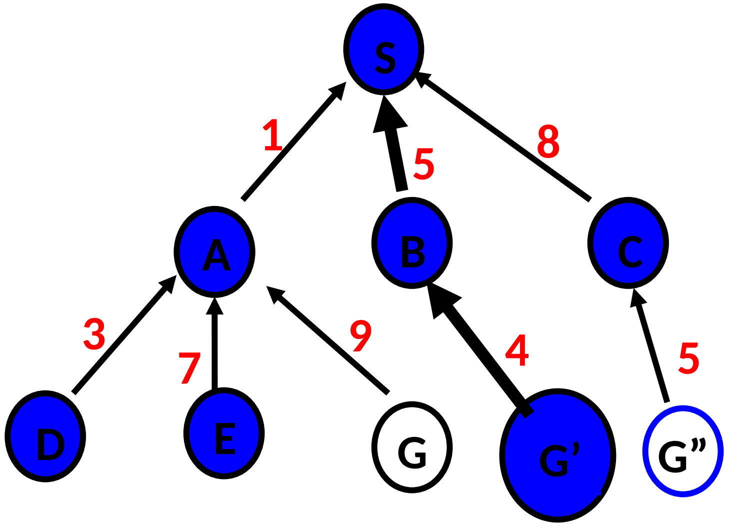
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

And how on our small example?

Uniform-Cost Search

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

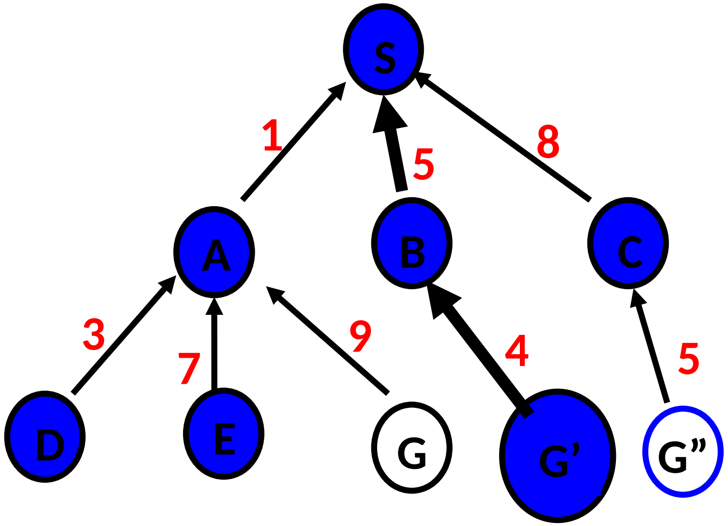
exp. node nodes list   CLOSED list



Uniform-Cost Search

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

exp. node	**nodes list**	**CLOSED 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) }	



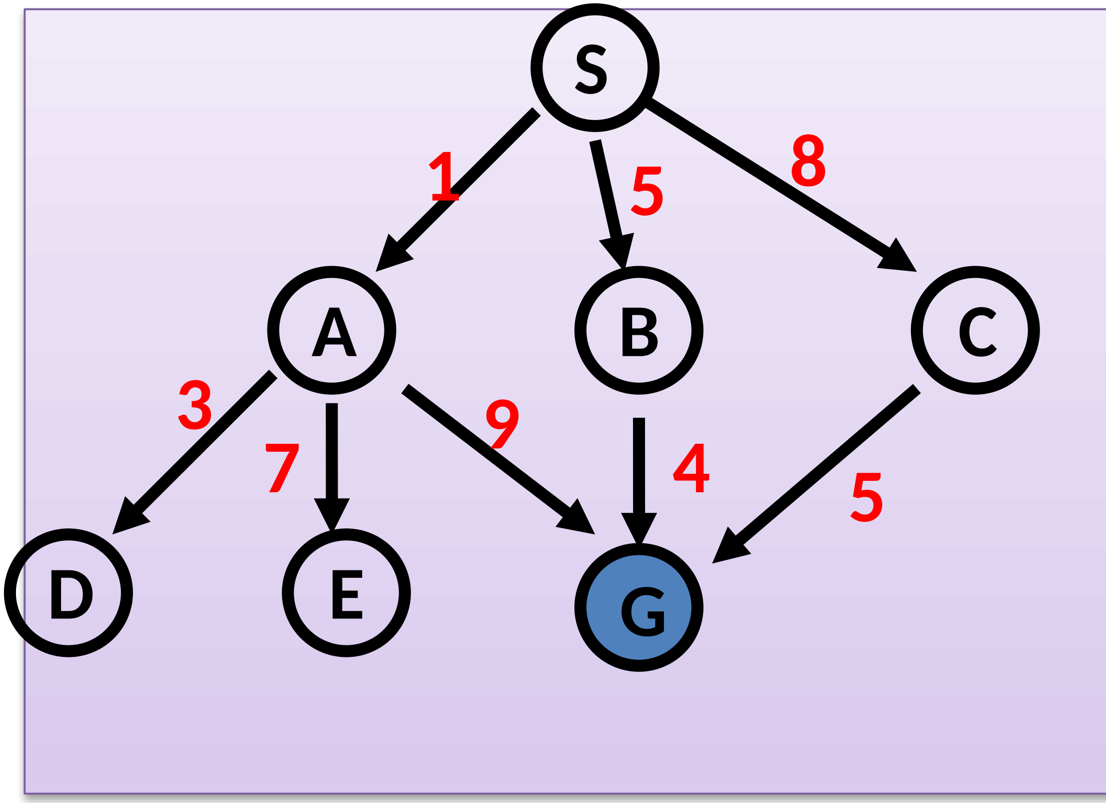
Solution path found is S B G <-- this G has cost 9, not 10

Number of nodes expanded (including goal node) = **7**

How they perform

- **Depth-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)



Depth-First Search:
Breadth-First Search:
Uniform-Cost Search:
Iterative-Deepening Search:

When to use what?

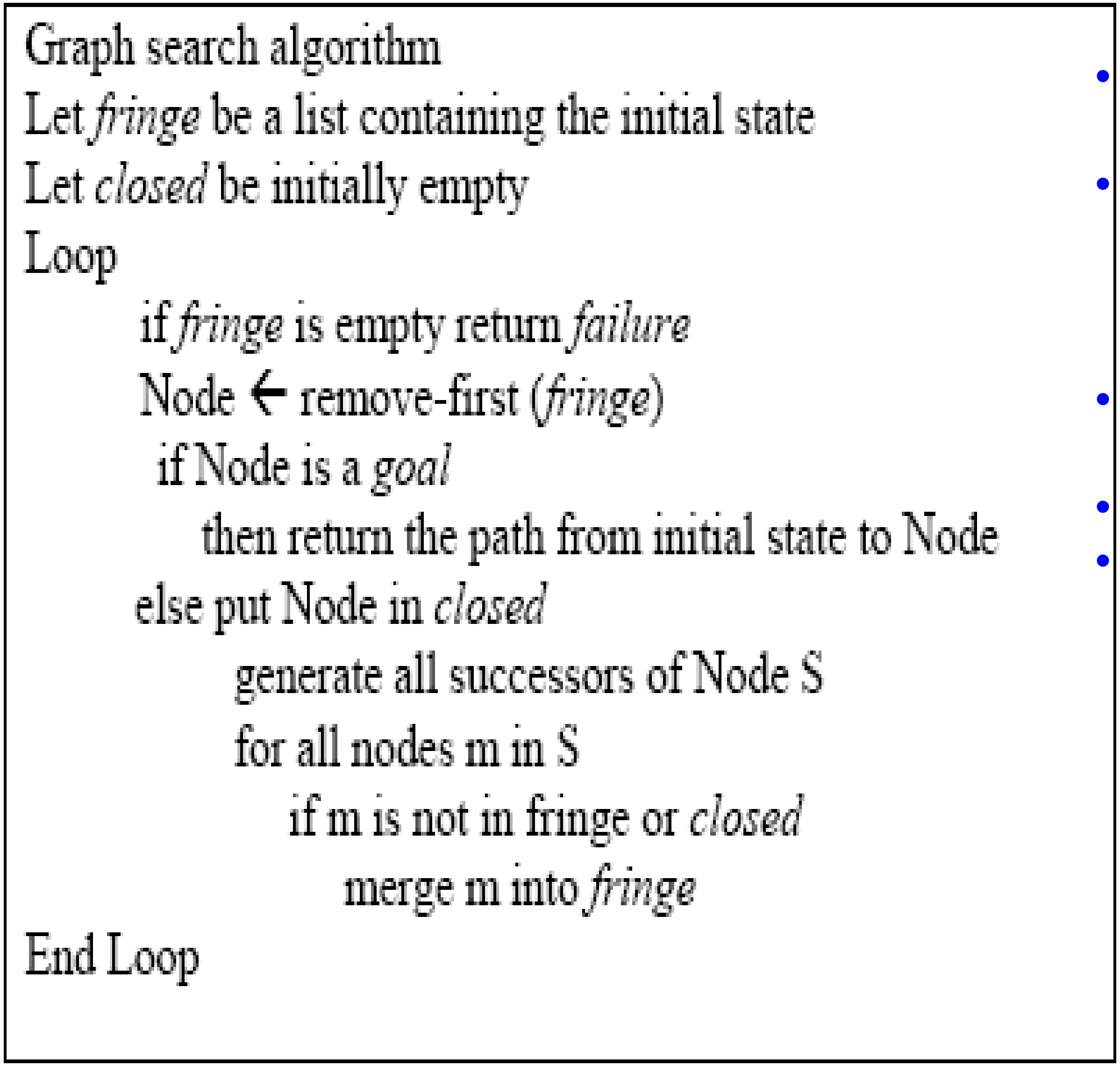
- **Depth-First Search:**
  - Many solutions exist
  - Know (or have a good estimate of) the depth of solution
- **Breadth-First Search:**
  - Some solutions are known to be shallow
- **Uniform-Cost Search:**
  - Actions have varying costs
  - Least cost solution is the required

  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 ← 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✉ Set of successor
- M✉node going to be generated

Informed Search

- We have seen that uninformed search methods that systematically explore the state space and find the goal.
- They are inefficient in most cases.
- Informed search methods use problem specific knowledge, and may be more efficient.
- At the heart of such algorithms there is the concept of a **heuristic function**.

Heuristics

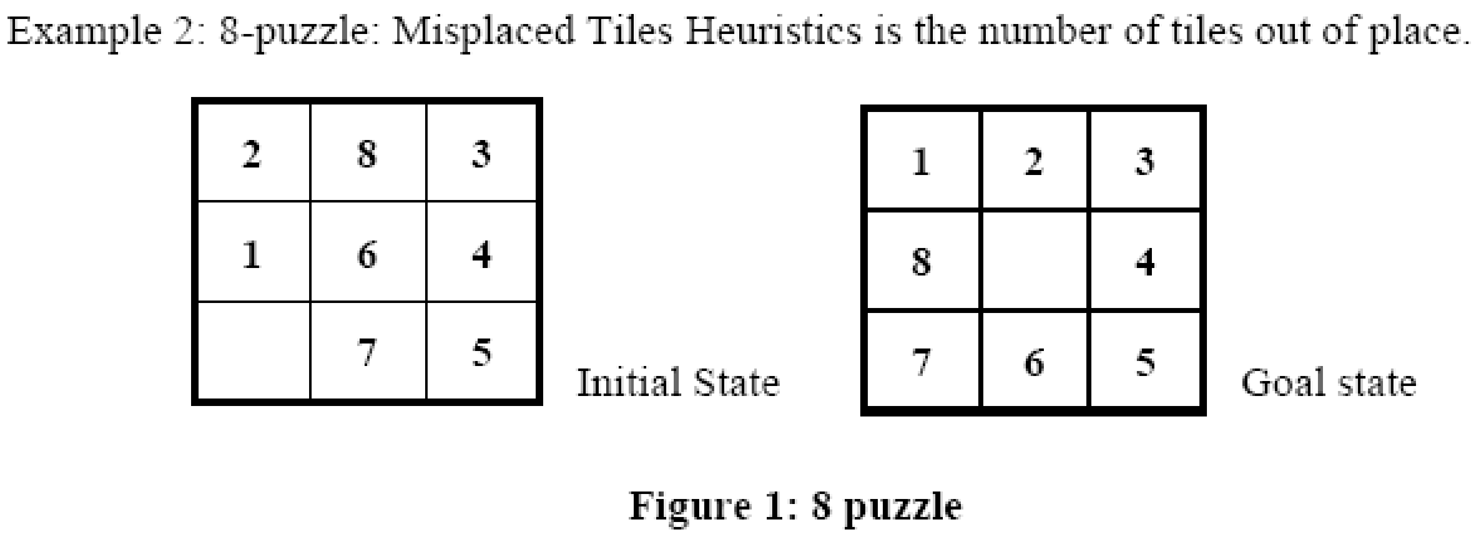
- Heuristic “Heuristics are criteria, methods or principles for deciding which among several alternative actions, promises to be the most effective in order to achieve some goal”.
- quote by Judea Pearl,

- In heuristic search or informed search, heuristics are used to identify the most promising search path.

Example of Heuristic Function

- A heuristic function 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)* = estimated cost of the cheapest path from node n 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(Kolkata) = euclideanDistance(Kolkata, Guwahati)*

Example 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.

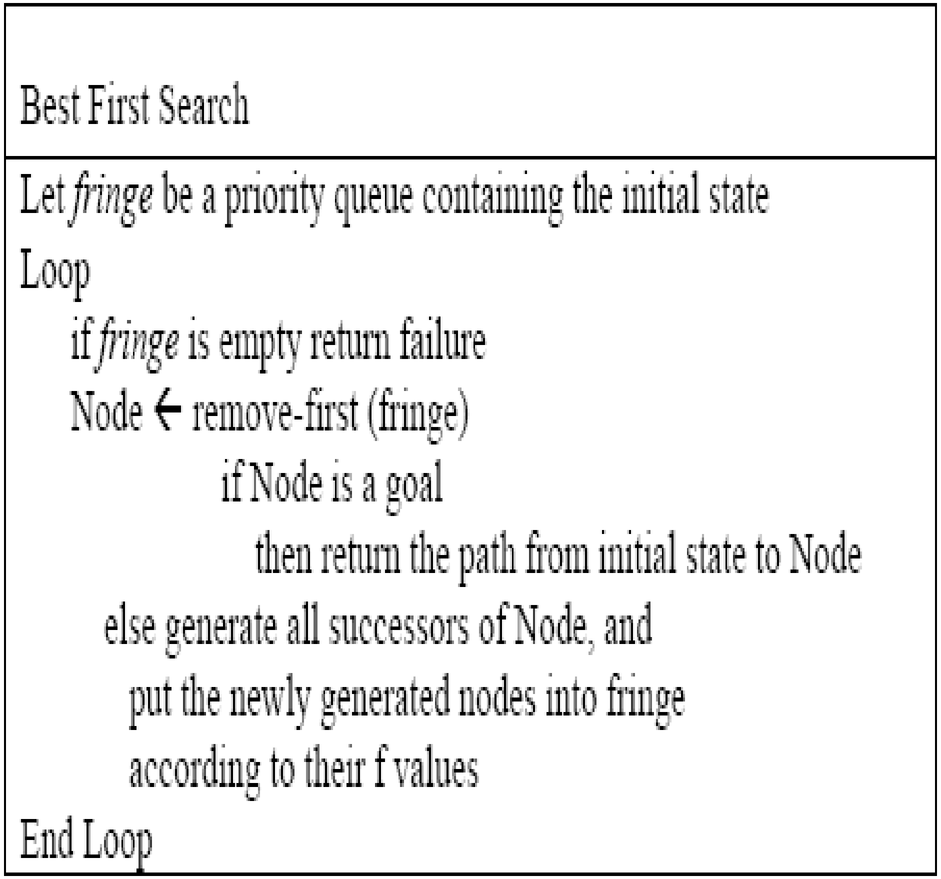
h(n) = 5 (because the tiles 2, 8, 1, 6 and 7 are out of place.)

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.



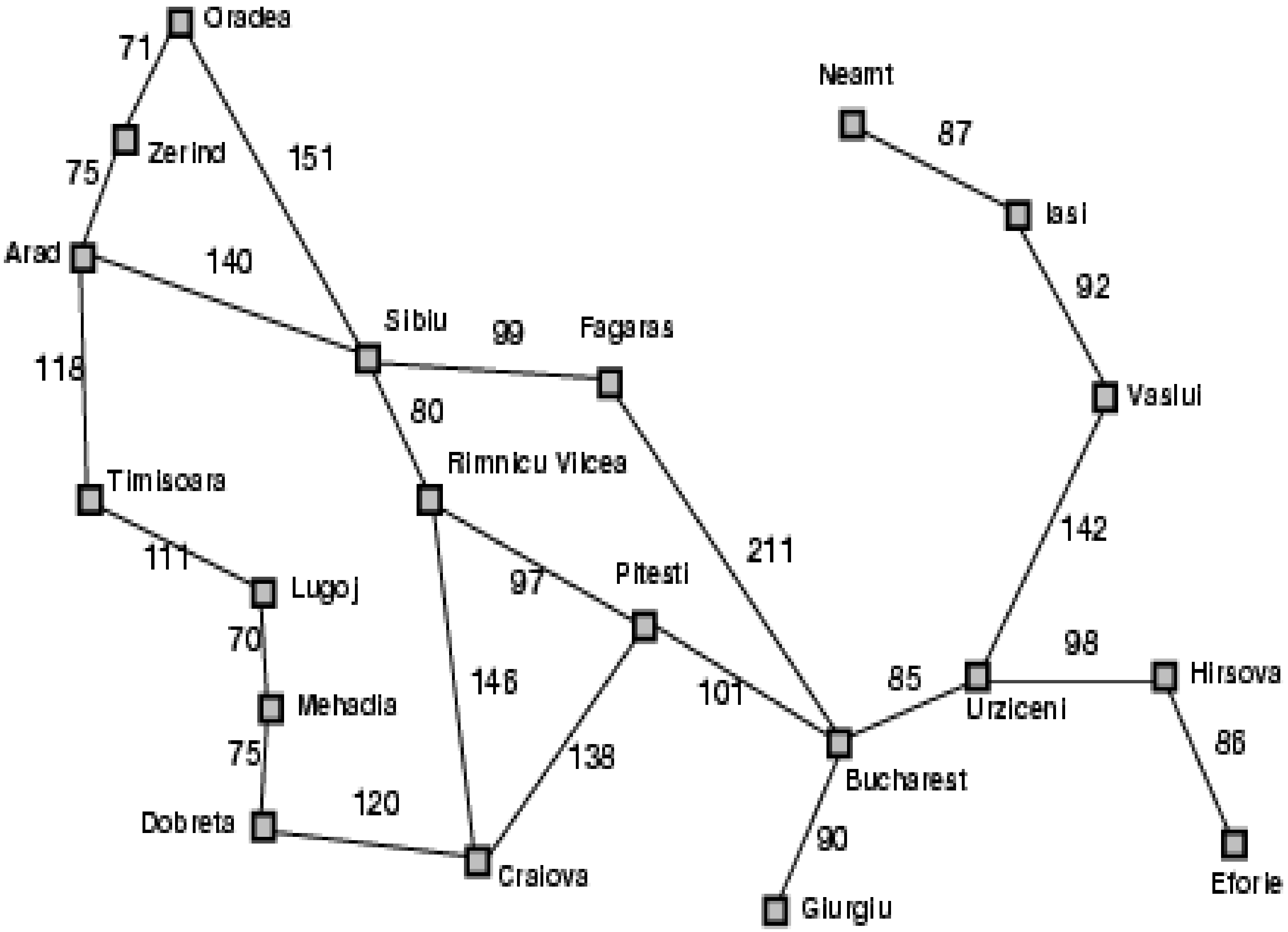
```
Best First Search

Let fringe be a priority queue containing the initial state
Loop
    if fringe is empty return failure
    Node ← remove-first (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 fringe
        according to their f values
End Loop
```

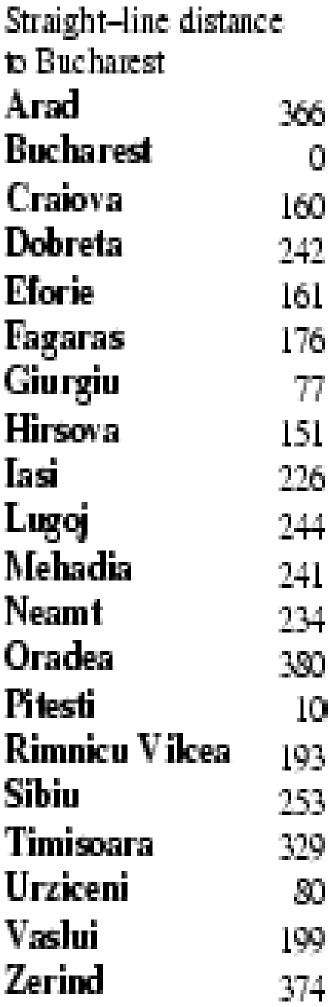
- The algorithm maintains a priority queue of nodes to be explored
- A cost function f(n) is applied to each node.
- The nodes are put in OPEN in the order of their f values.
- Nodes with smaller f(n) values are expanded earlier.

Greedy Search

- In greedy search, the idea is to expand the node with the smallest estimated cost to reach the goal.
- heuristic function --*f(n) = h(n)*
- *h(n)* estimates the distance remaining to a goal.
- Greedy algorithms often perform very well. They tend to find good solutions quickly, although not always optimal ones.
- The resulting algorithm is not optimal
- Incomplete -It may fail to find a solution even if one exists.
- This can be seen by running greedy search on the following example.
- A good heuristic for the **route-finding** problem would be straight-line distance to the goal.

Romania with step costs in km

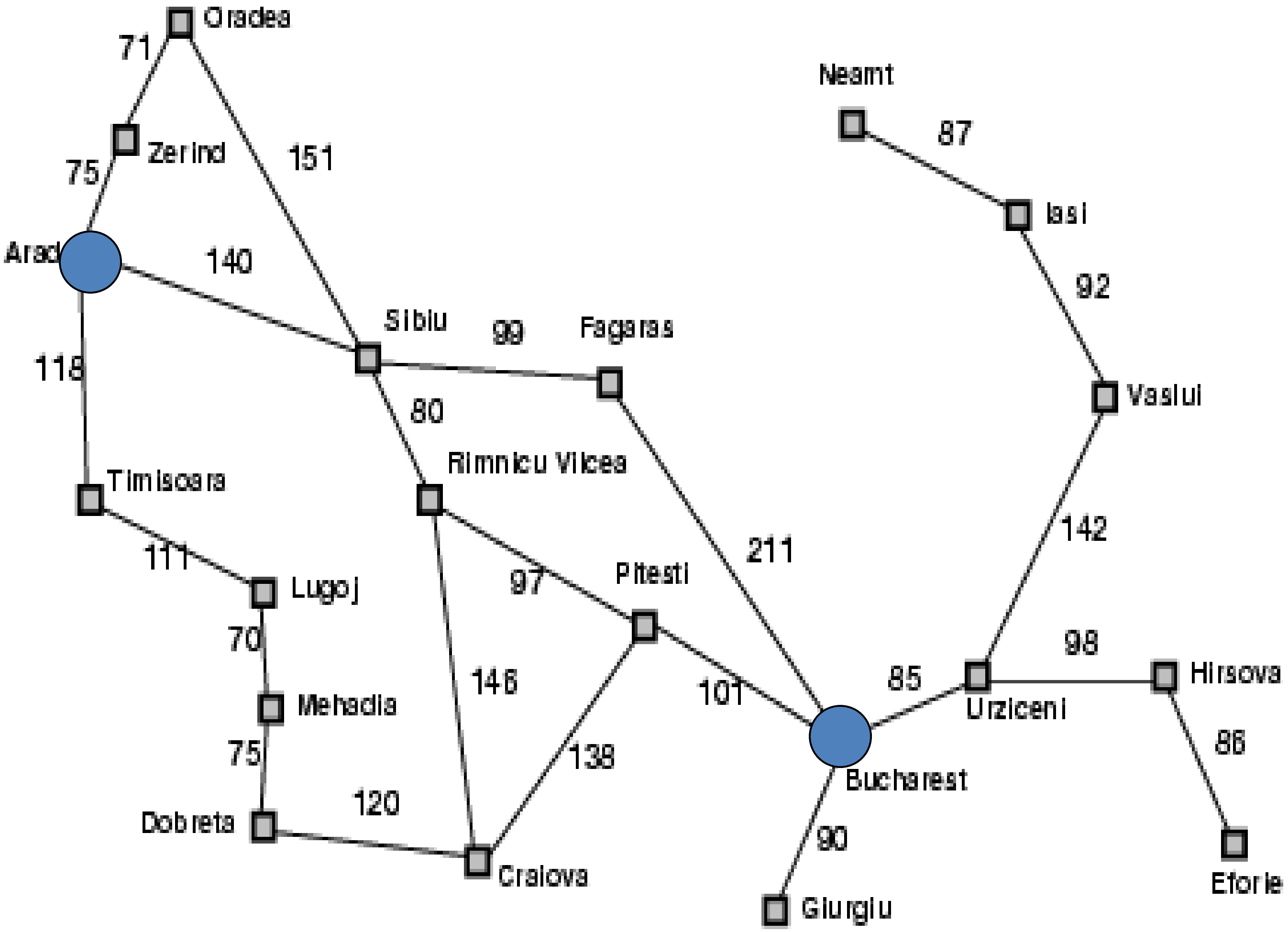




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search

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



Straight–line distance
to Bucharest

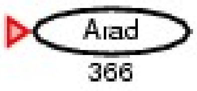
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

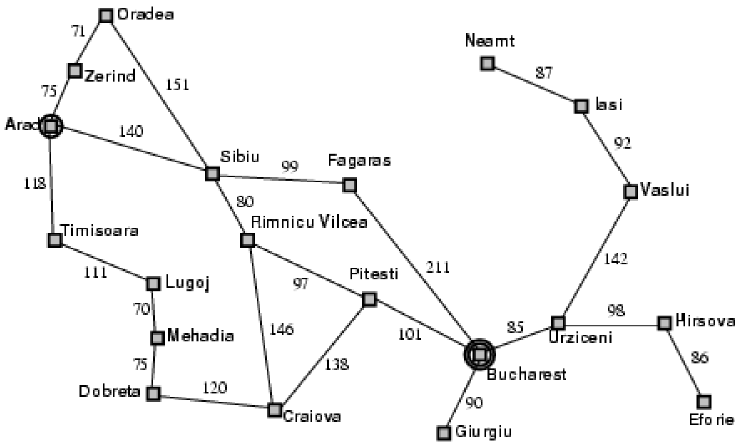
Greedy best-first search

- Evaluation function $f(n) = h(n)$ (heuristic)
- = estimate of cost from n to *goal*

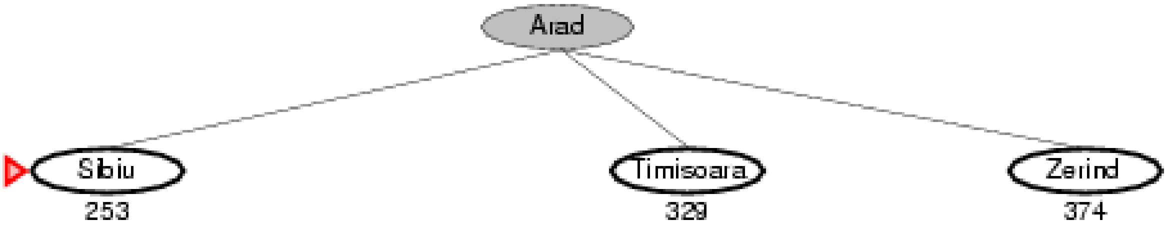
- e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal

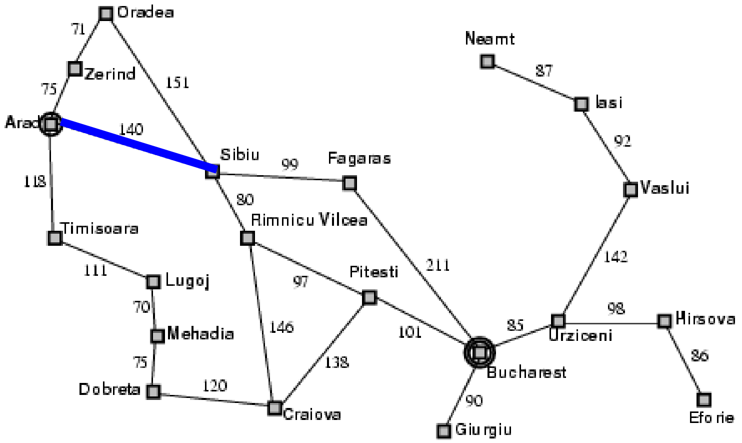
Greedy best-first search example



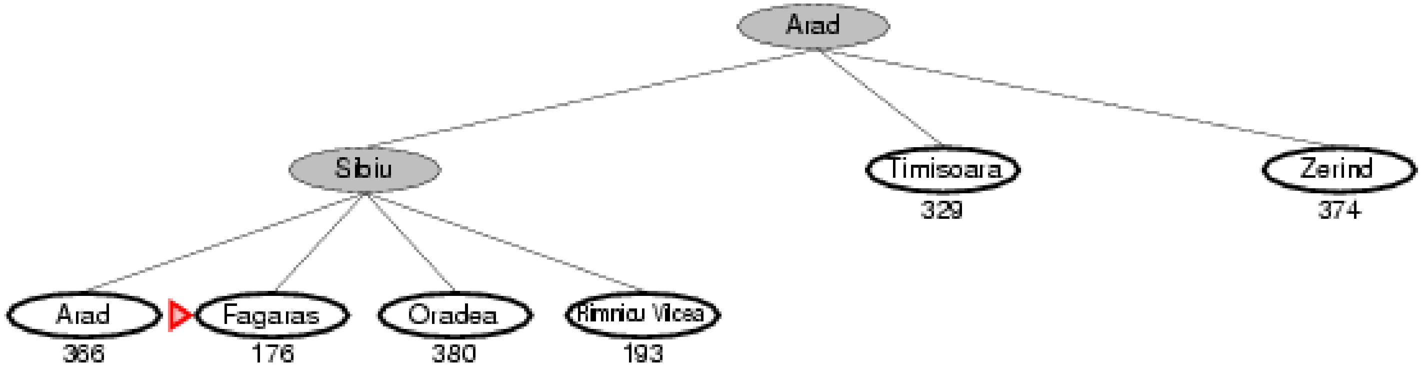


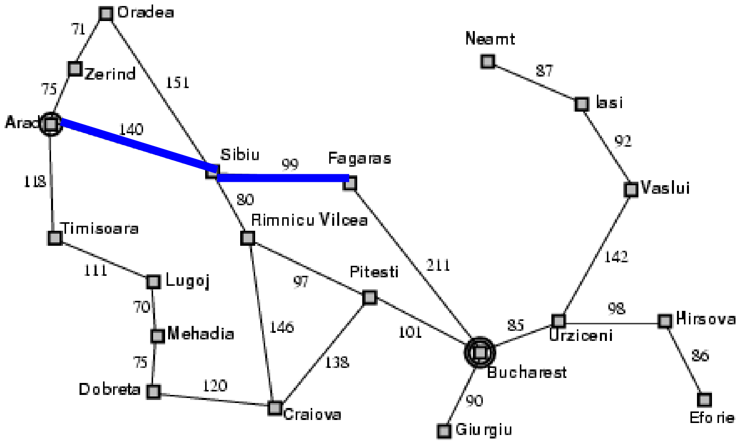
Greedy best-first search example



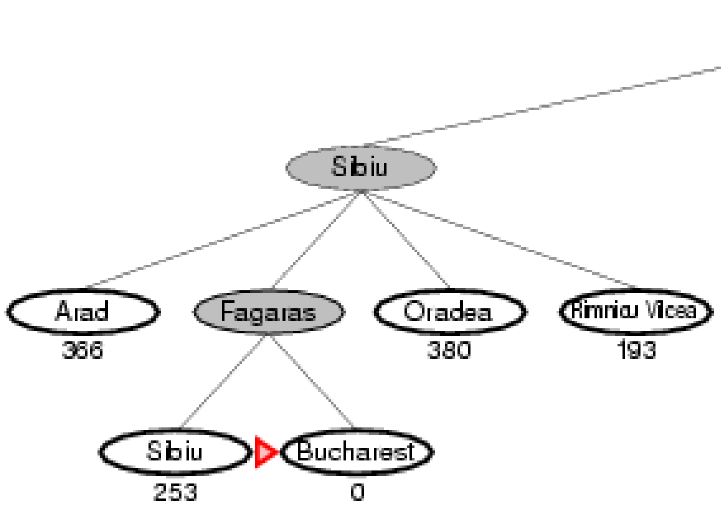


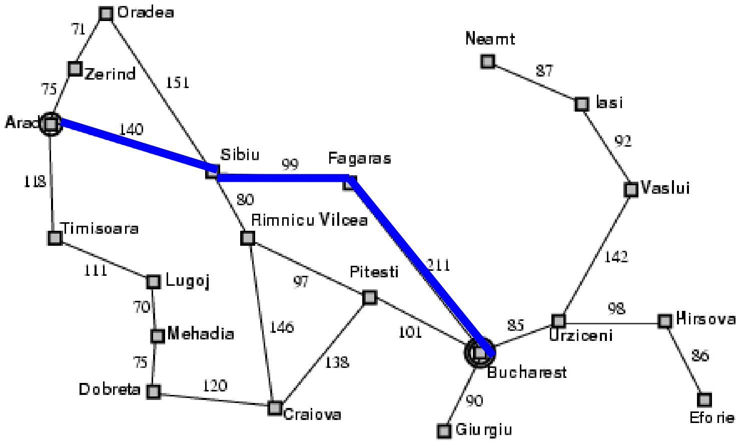
Greedy best-first search example



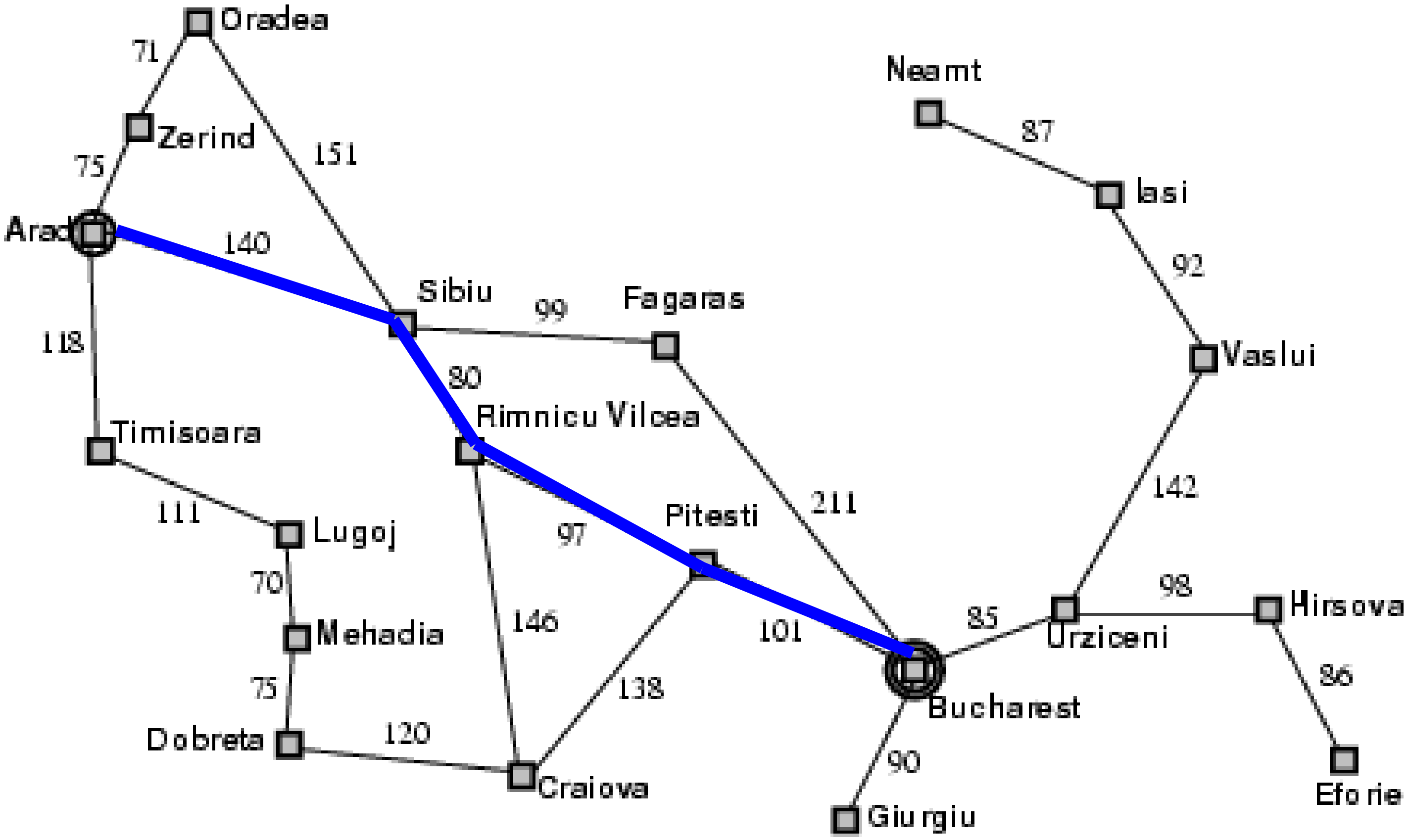


Greedy best-first search example

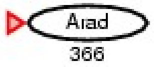


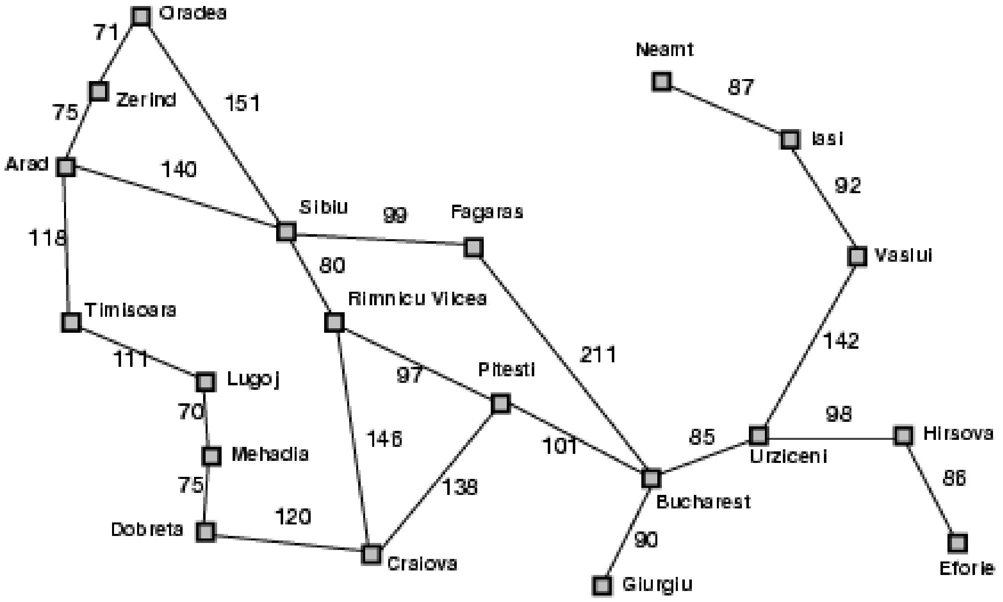


Optimal Path



Greedy best-first search example

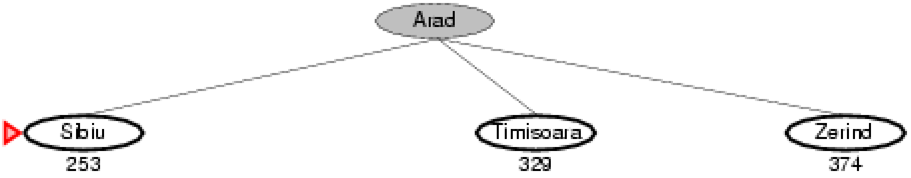


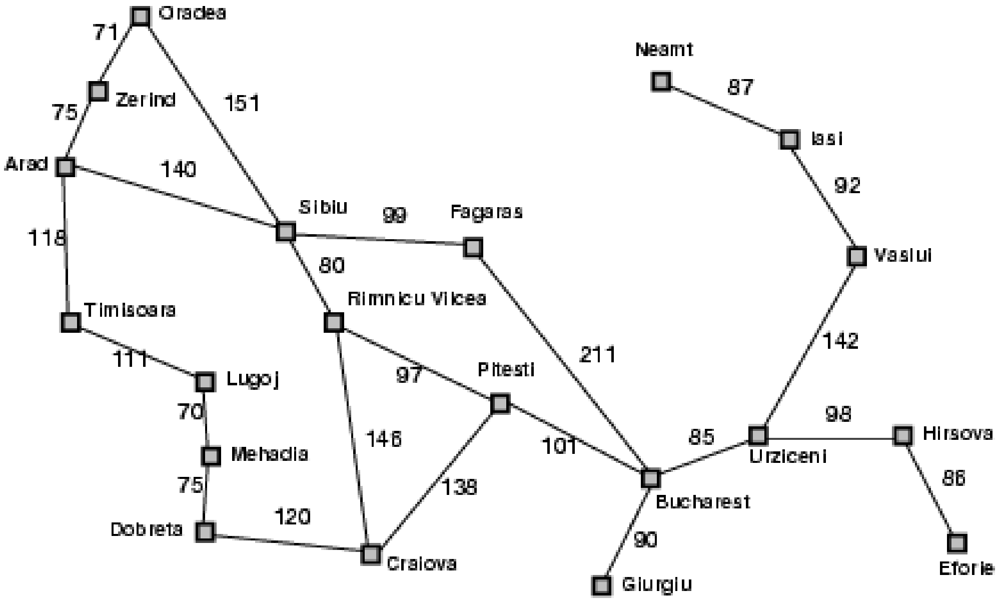


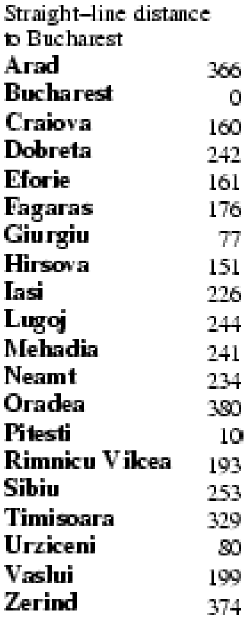
Straight–line distance
to Bucharest

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



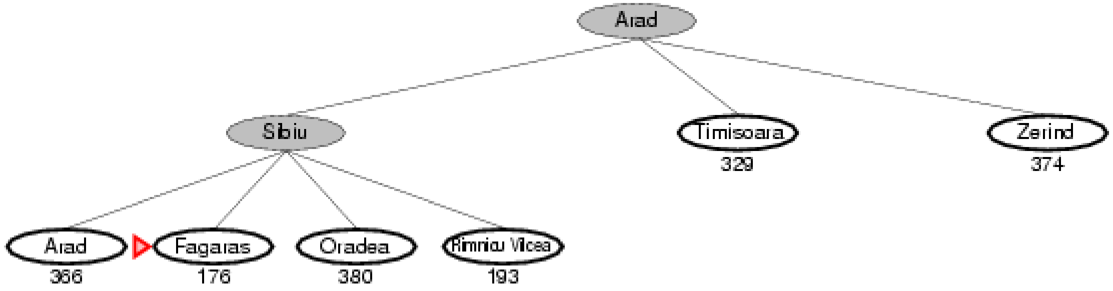


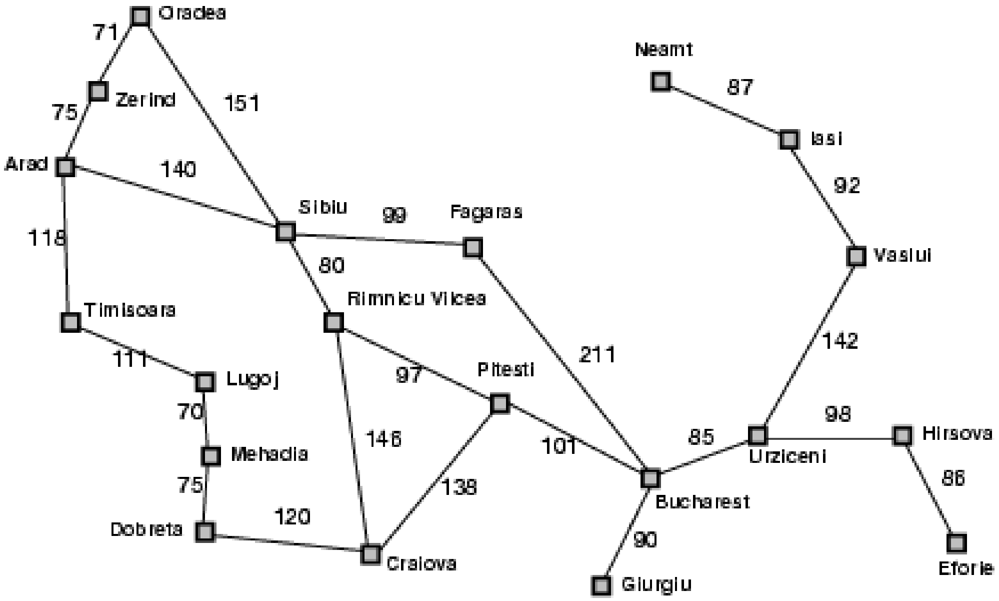


Straight–line distance
to Bucharest

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

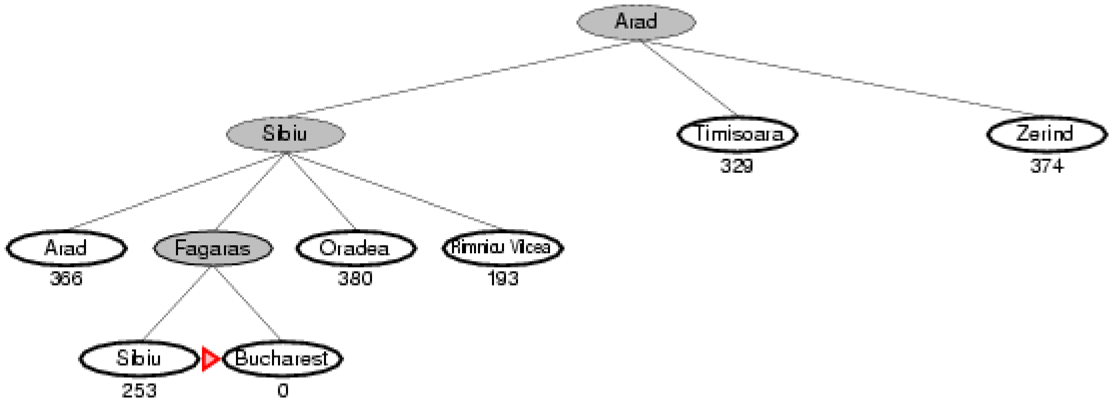


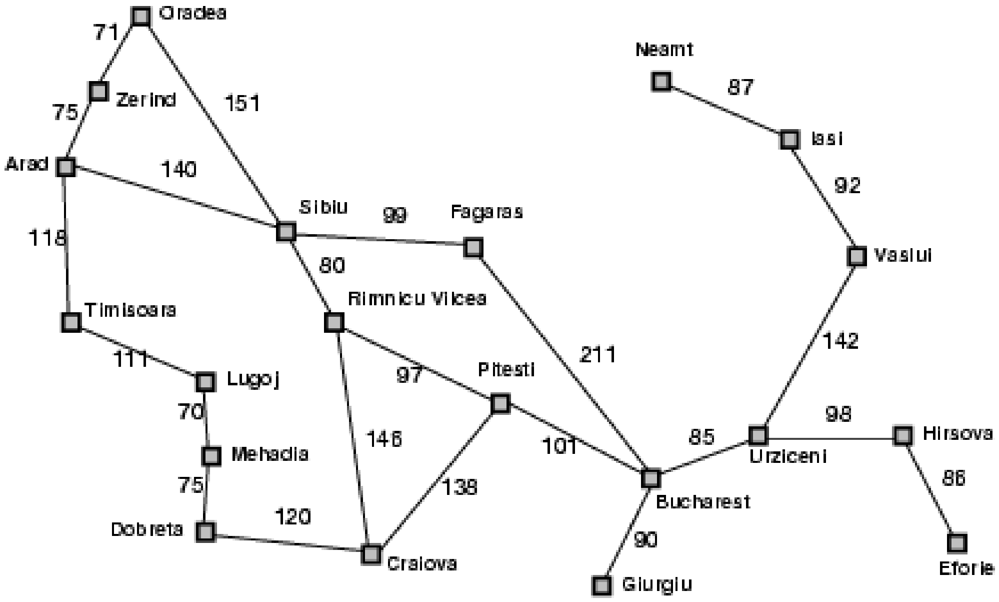


Straight–line distance
to Bucharest

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

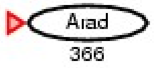




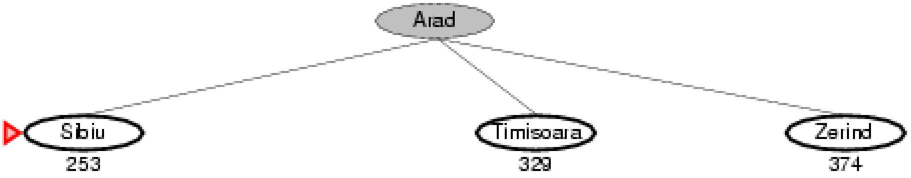
Straight–line distance
to Bucharest

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

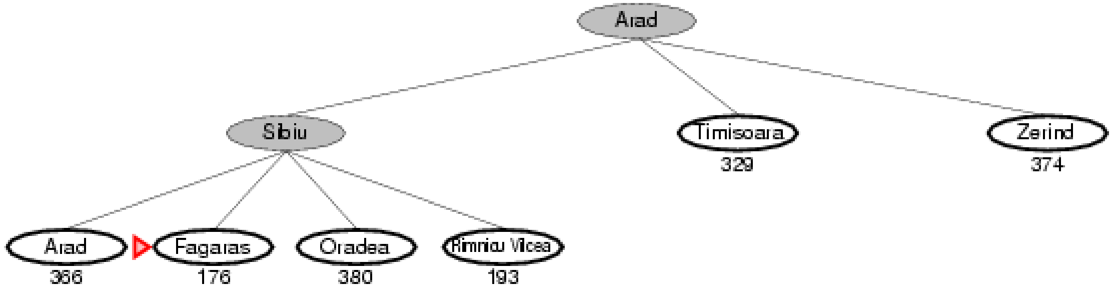
Greedy best-first search example



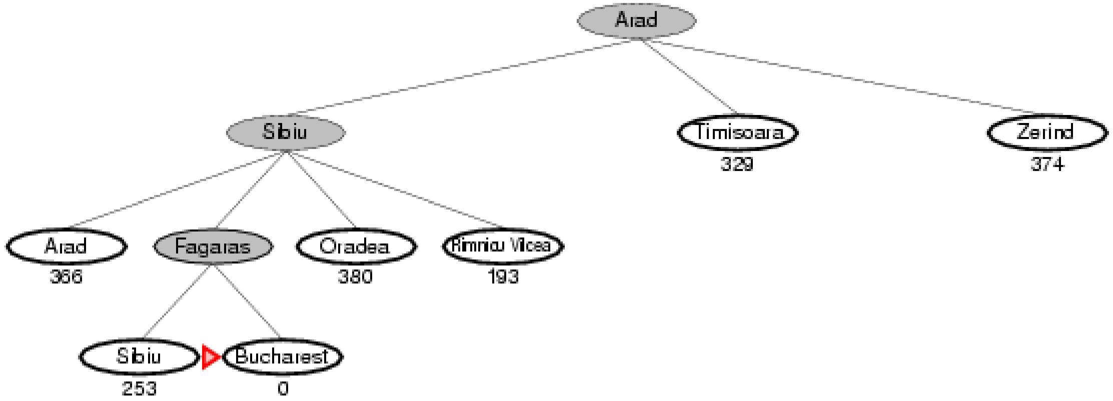
Greedy best-first search example



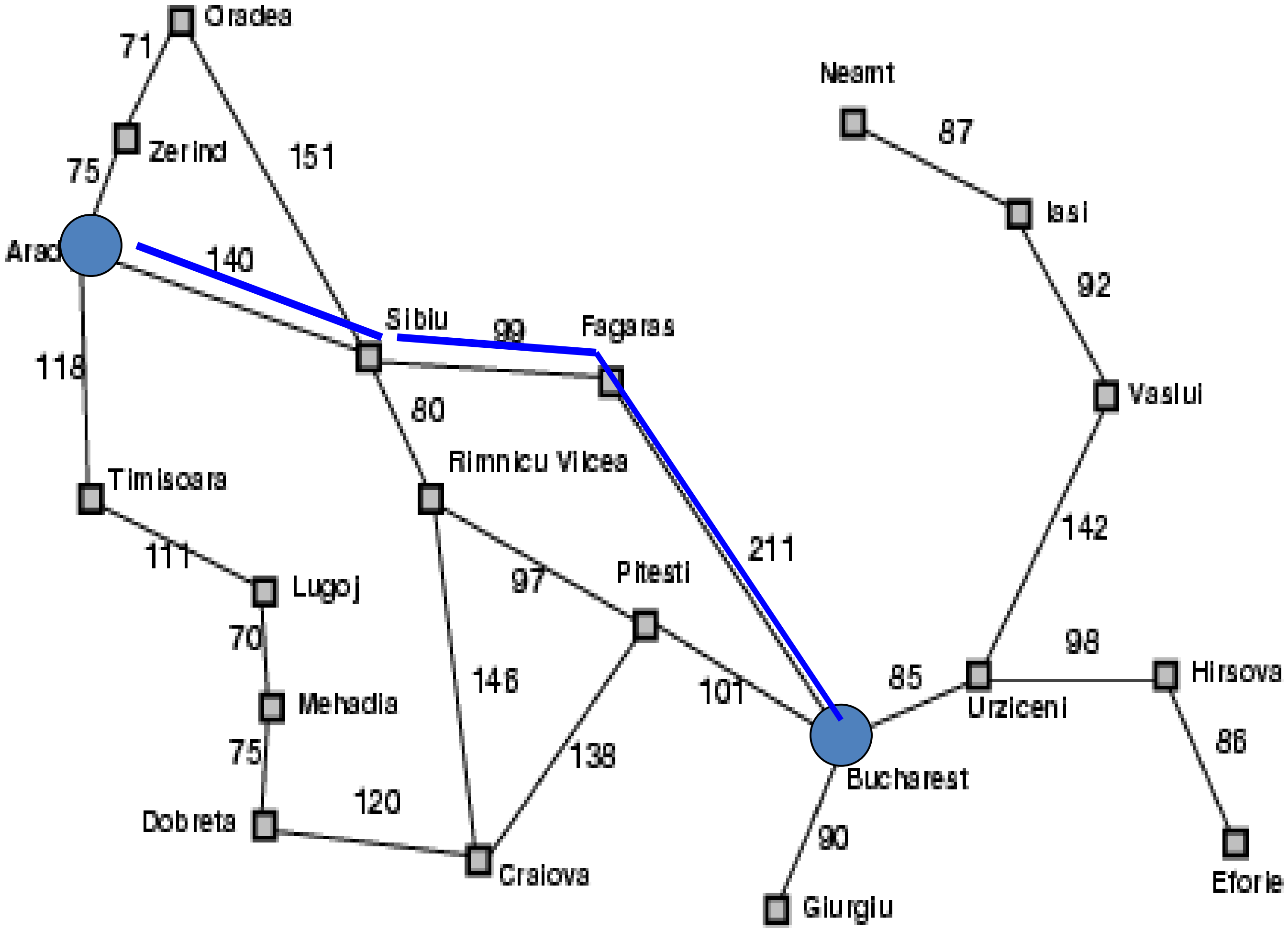
Greedy best-first search example



Greedy best-first search example



Romania with step costs in km

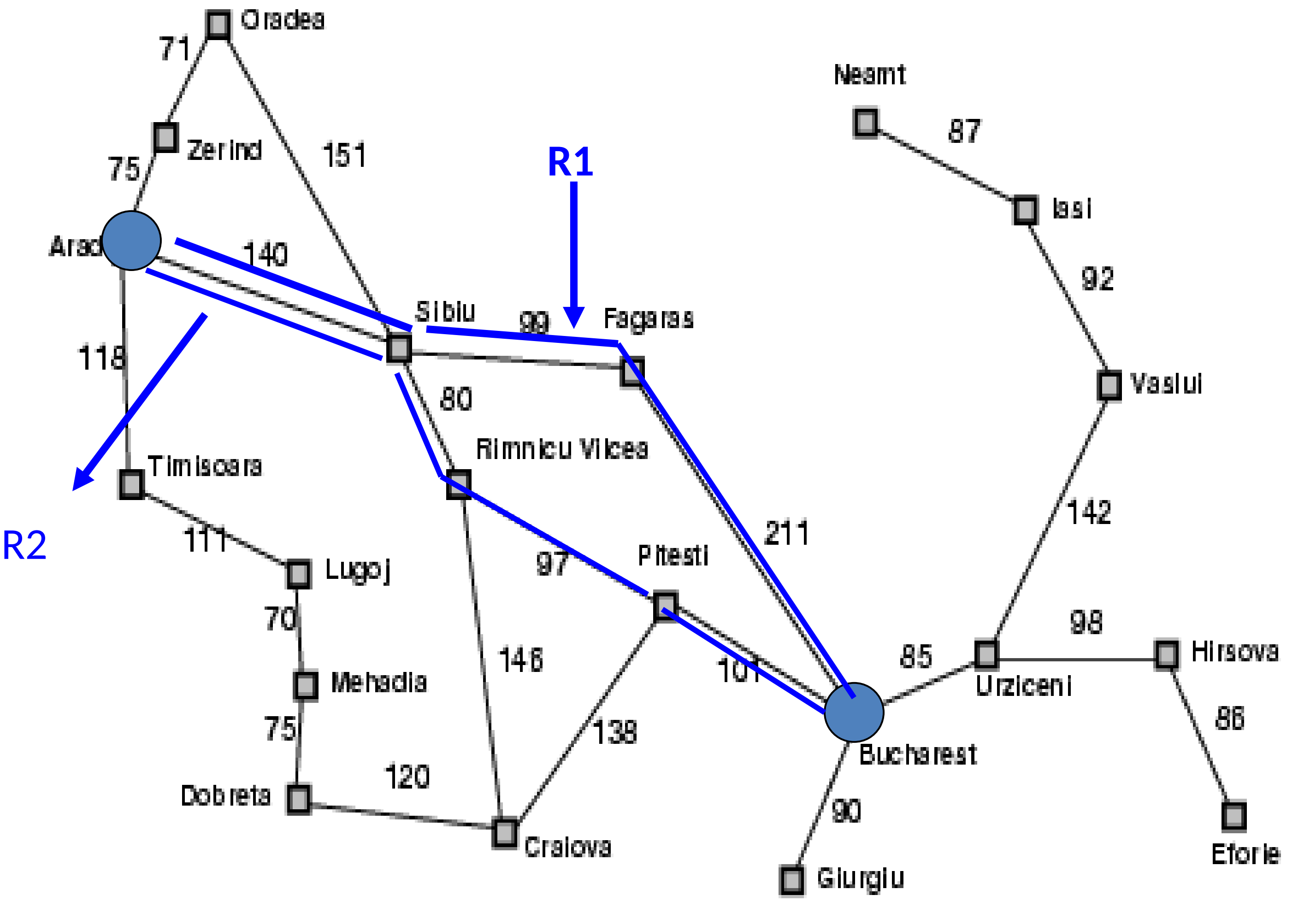


Straight–line distance
to Bucharest

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
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

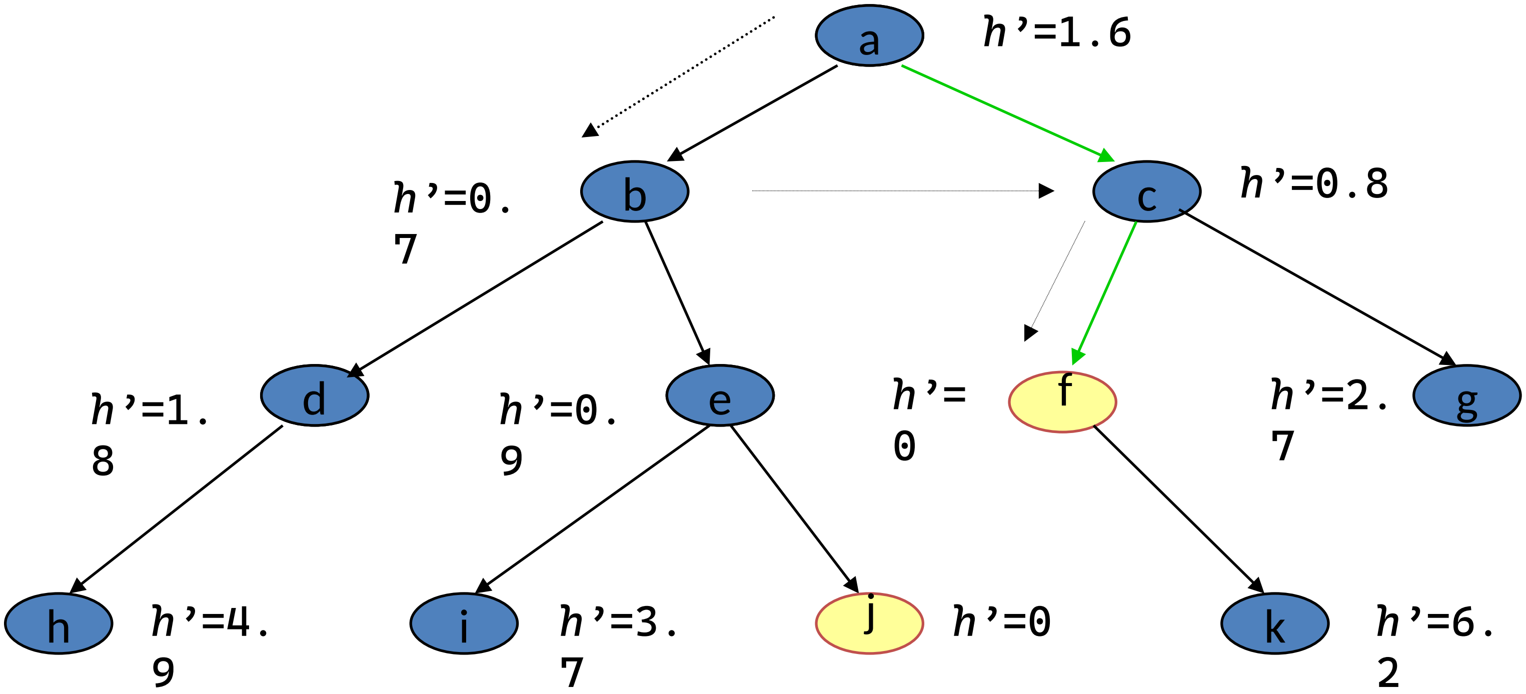
Properties of greedy best-first search

- Complete? No – can get stuck in loops, e.g., Iasi ✉ Neamt ✉ Iasi ✉ Neamt ✉
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

A* algorithm

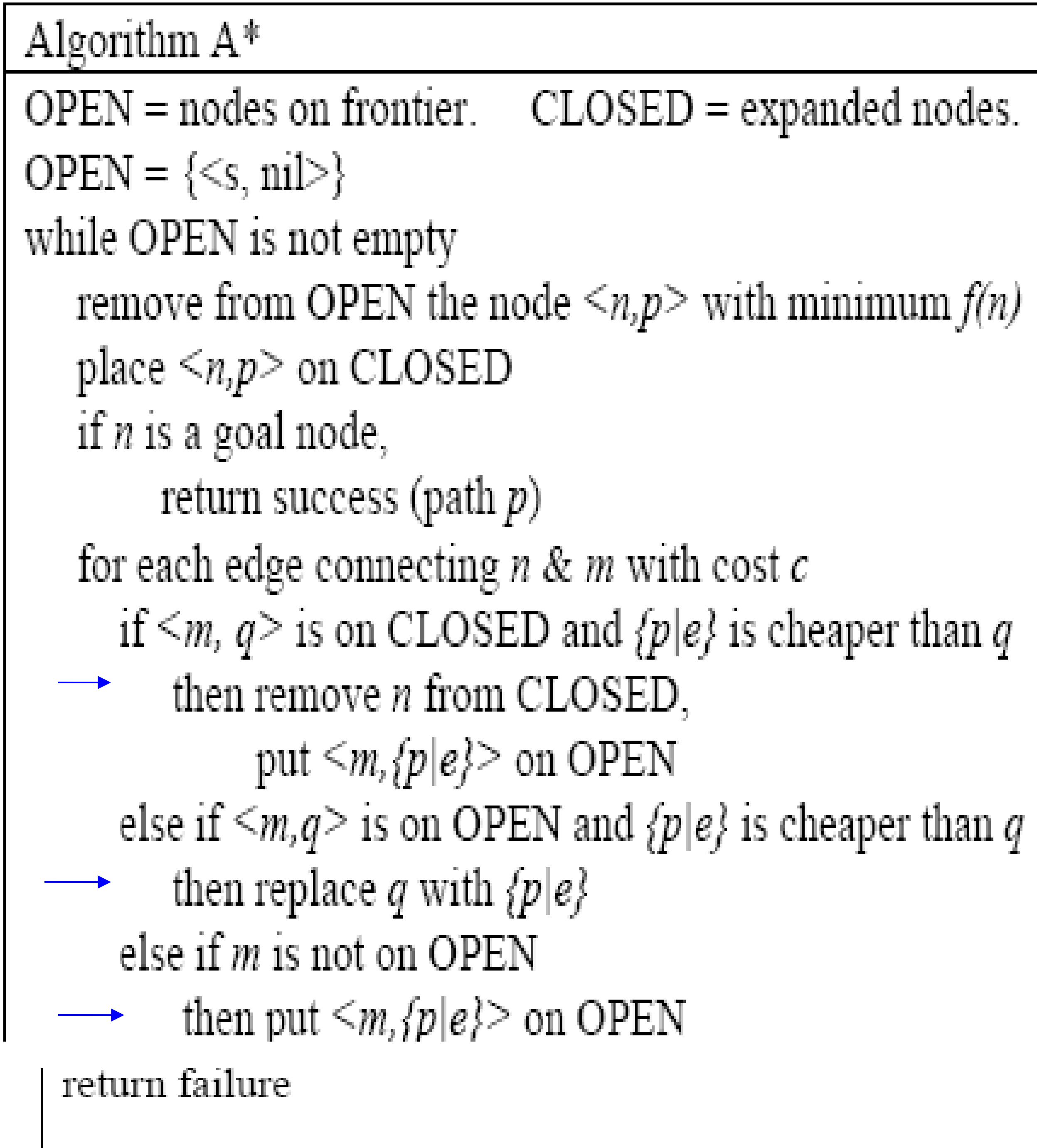
- We will next consider the famous A* algorithm. Nilsson & Rafael in 1968.
- A* is a best first search *f(n) = g(n) + h'(n)*
  - *g(n)* = sum of edge costs from start to n(start node → current node)
  - h'*(n)* = estimate of lowest cost path from n to goal
  - *f'(n)* = actual distance so far + estimated distance remaining (n → 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) <= C*(n).*
- we can prove that if *h'(n)* is admissible, then the search will find an optimal solution.

Example of A* search

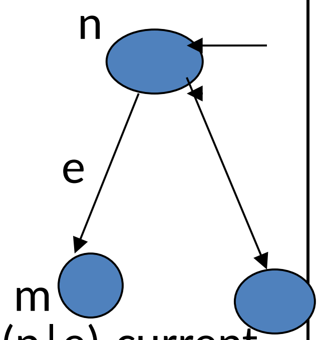


Cost per arc = 1

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



```
Algorithm A*
OPEN = nodes on frontier.    CLOSED = expanded nodes.
OPEN = {<s, nil>}
while OPEN is not empty
    remove from OPEN the node <n,p> with minimum f(n)
    place <n,p> on CLOSED
    if n is a goal node,
        return success (path p)
    for each edge connecting n & m with cost c
        if <m, q> is on CLOSED and {p|e} is cheaper than q
            then remove n from CLOSED,
                put <m,{p|e}> on OPEN
        else if <m,q> 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 <m,{p|e}> on OPEN
return failure
```



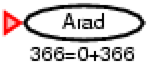
(m,p) (p|e)-current

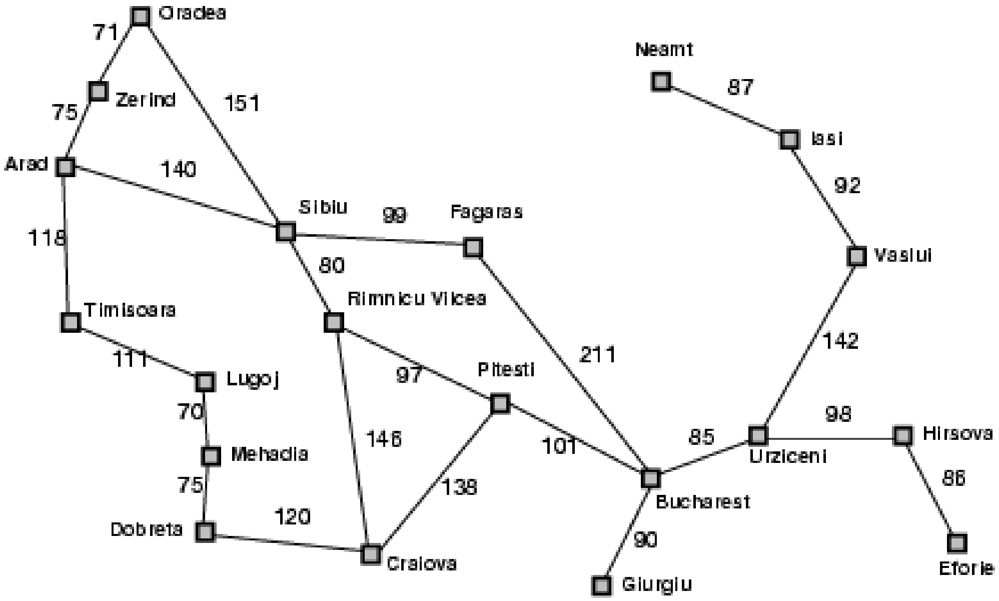
(m,q)-previously calculated distance

A* search: properties

- The algorithm A* is admissible.
- solution found by A* is an optimal solution.
- Complete
- No. of nodes searched still exponential in the worst case
- Otherwise, heuristic is logarithmically very accurate

A* search example

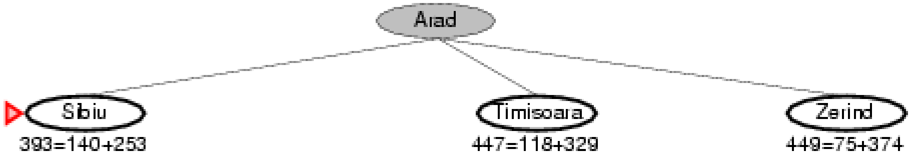


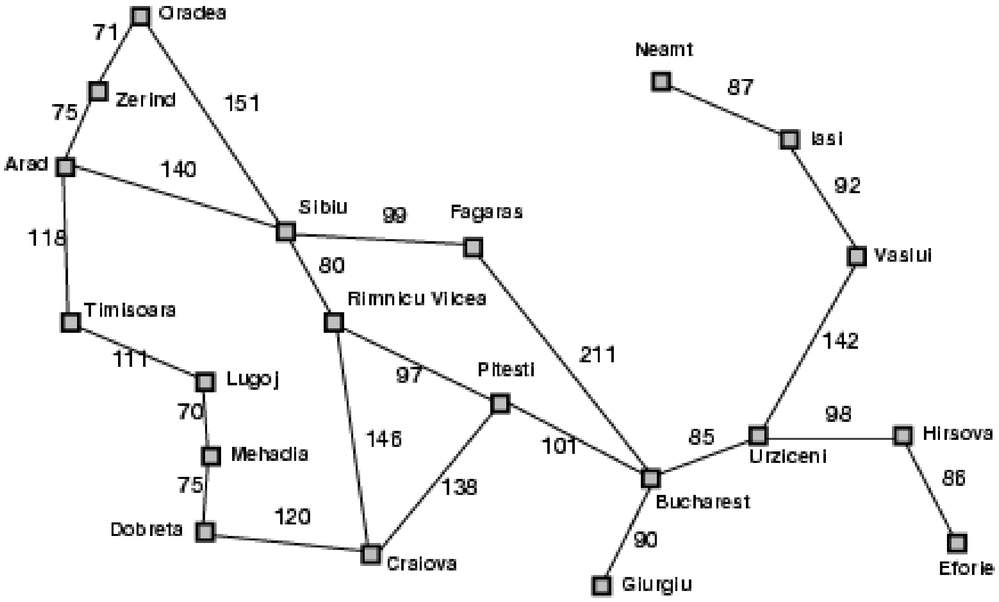


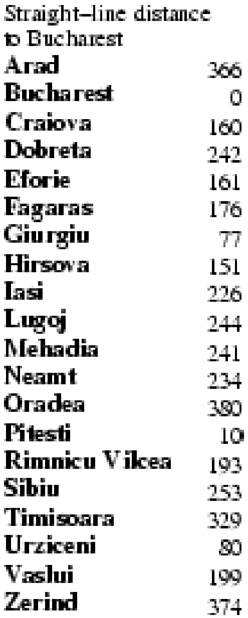
Straight–line distance
to Bucharest

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



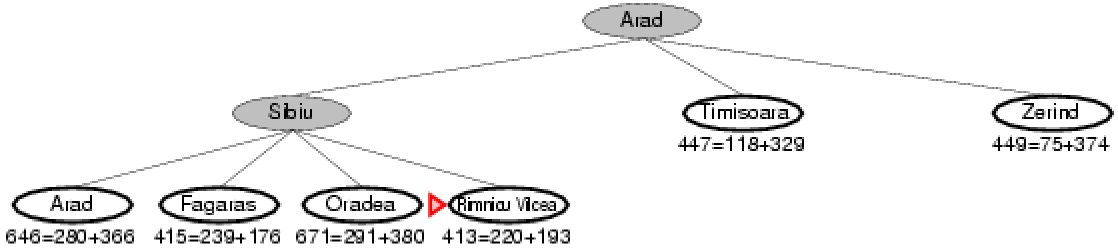


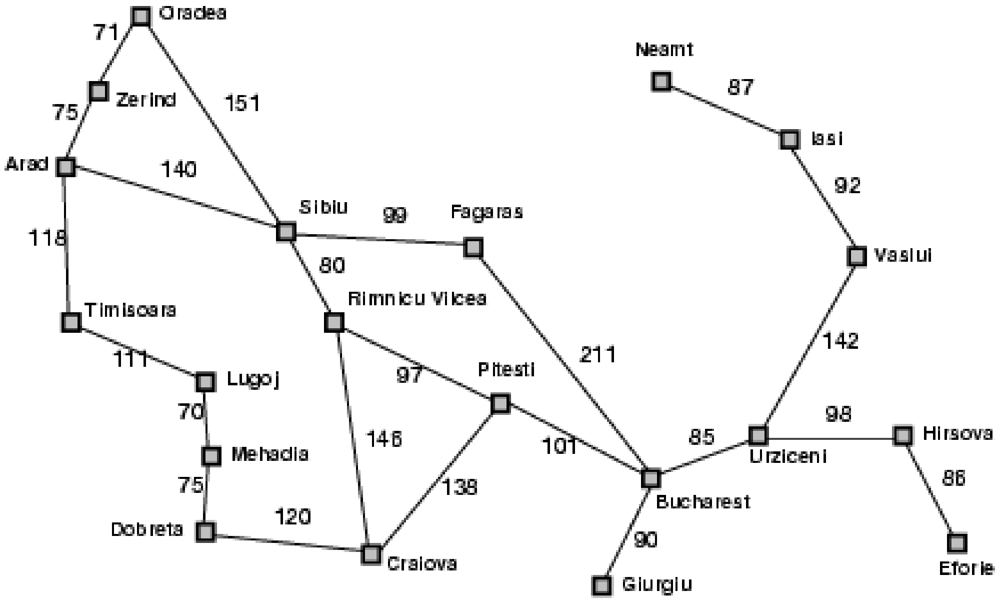


Straight–line distance
to Bucharest

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

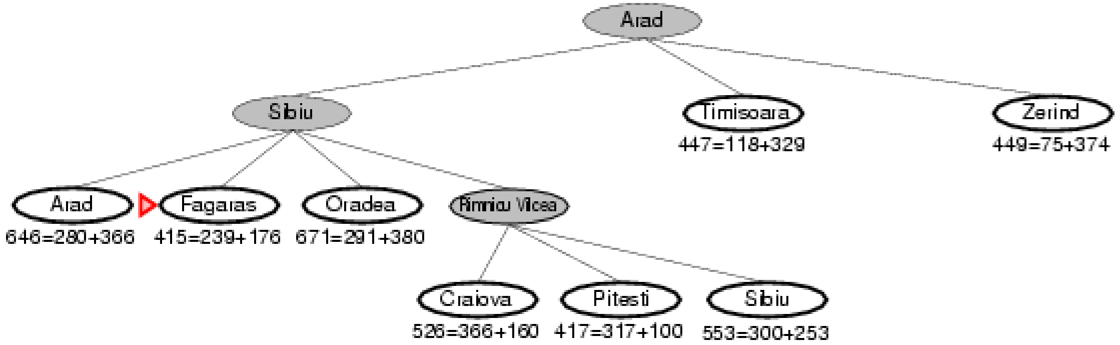


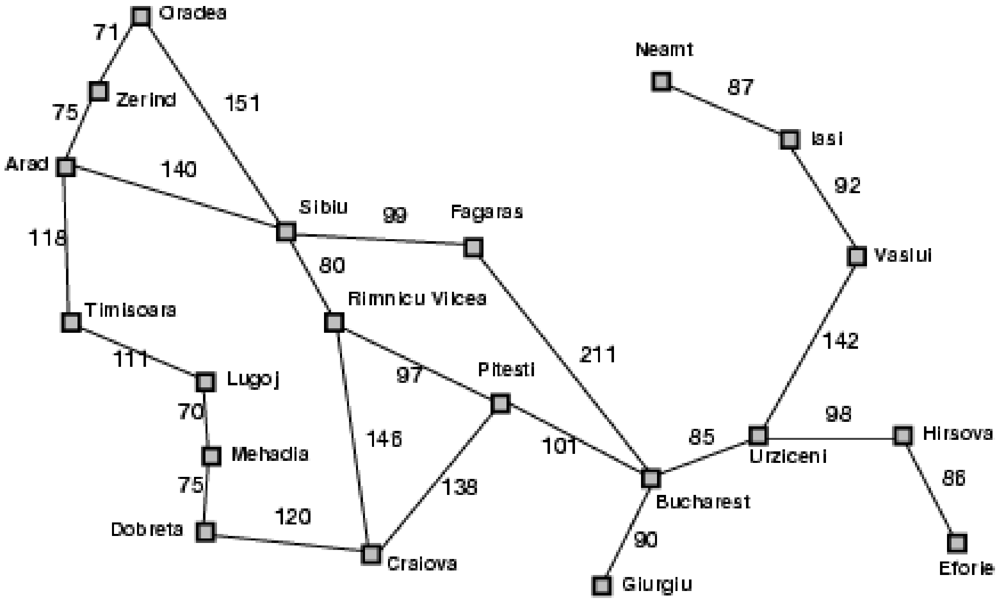


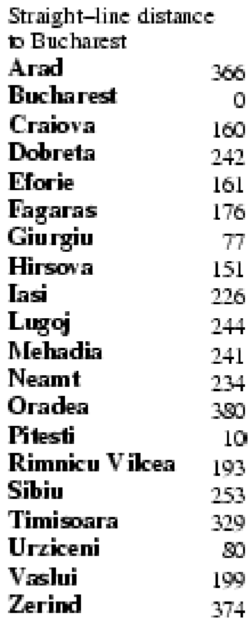
Straight–line distance
to Bucharest

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

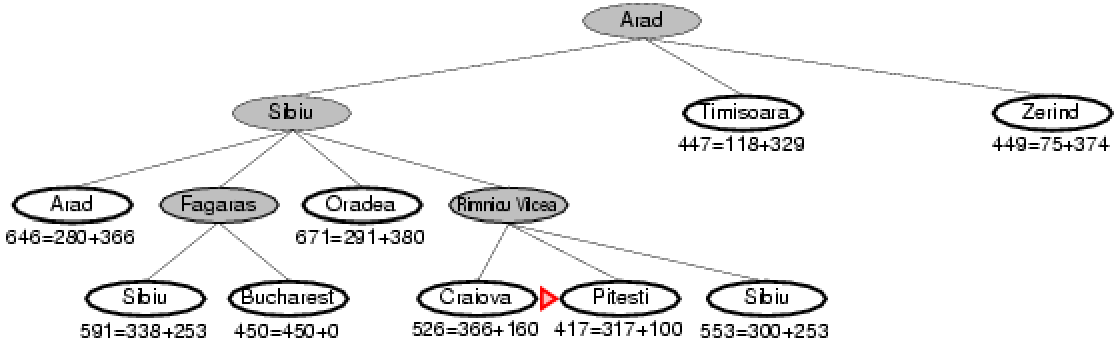


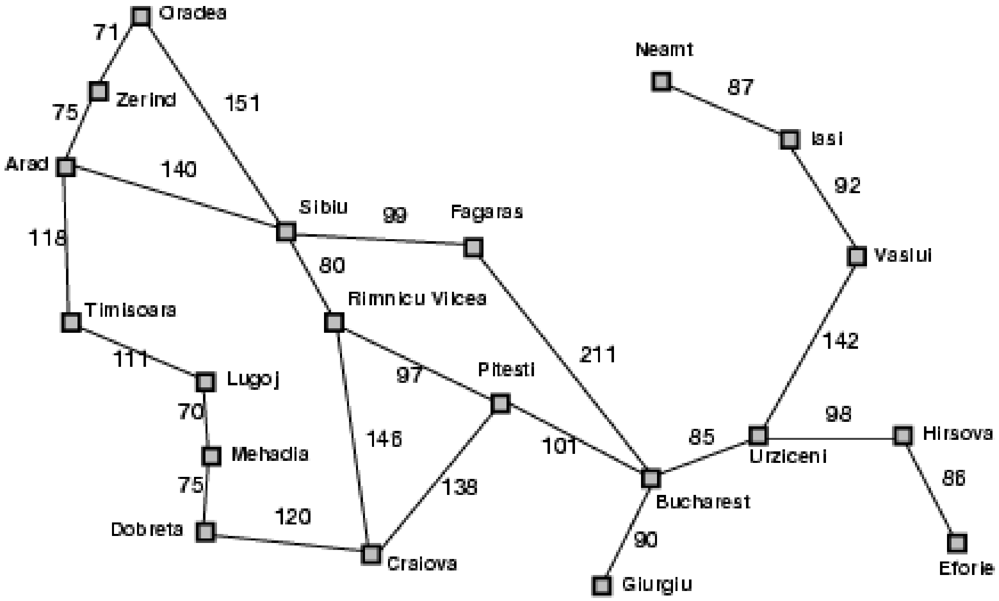




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

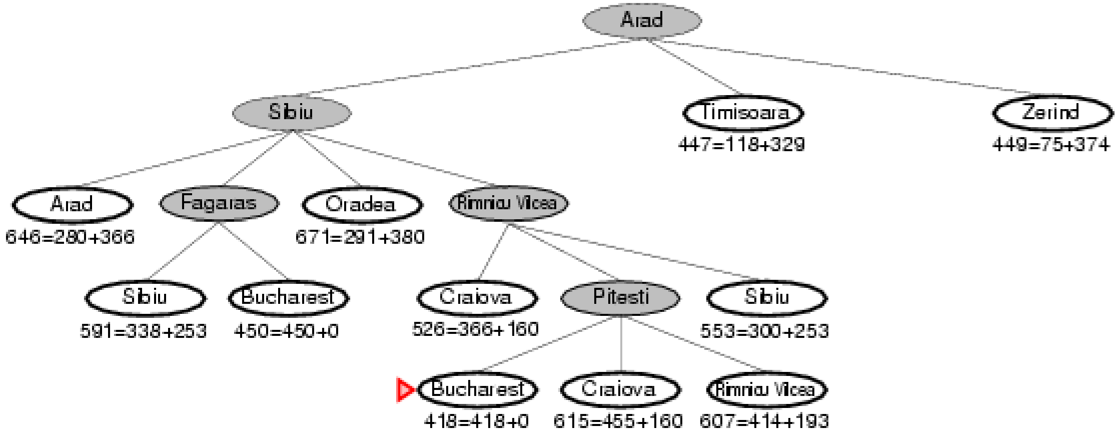
A* search example

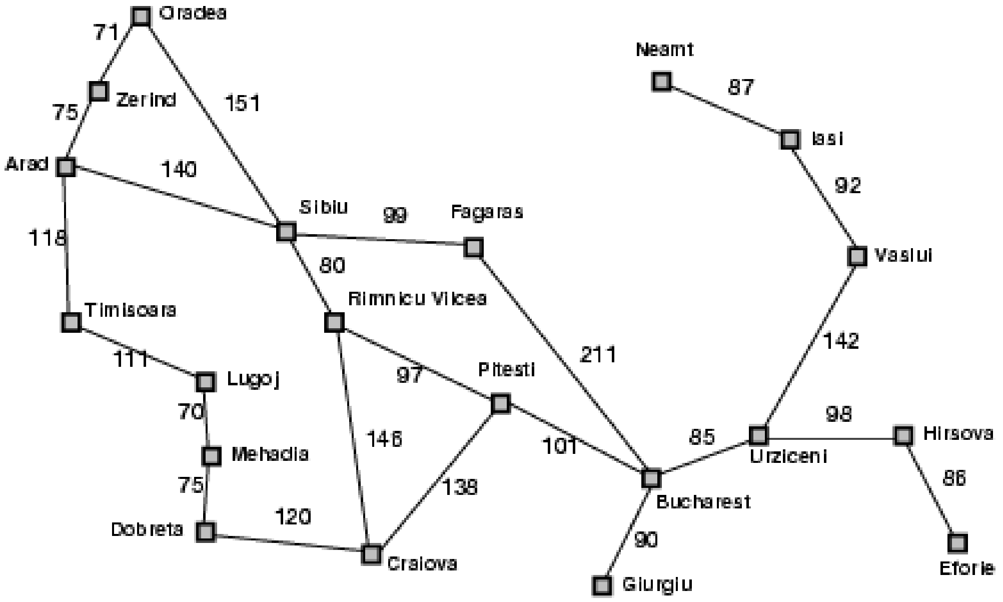




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

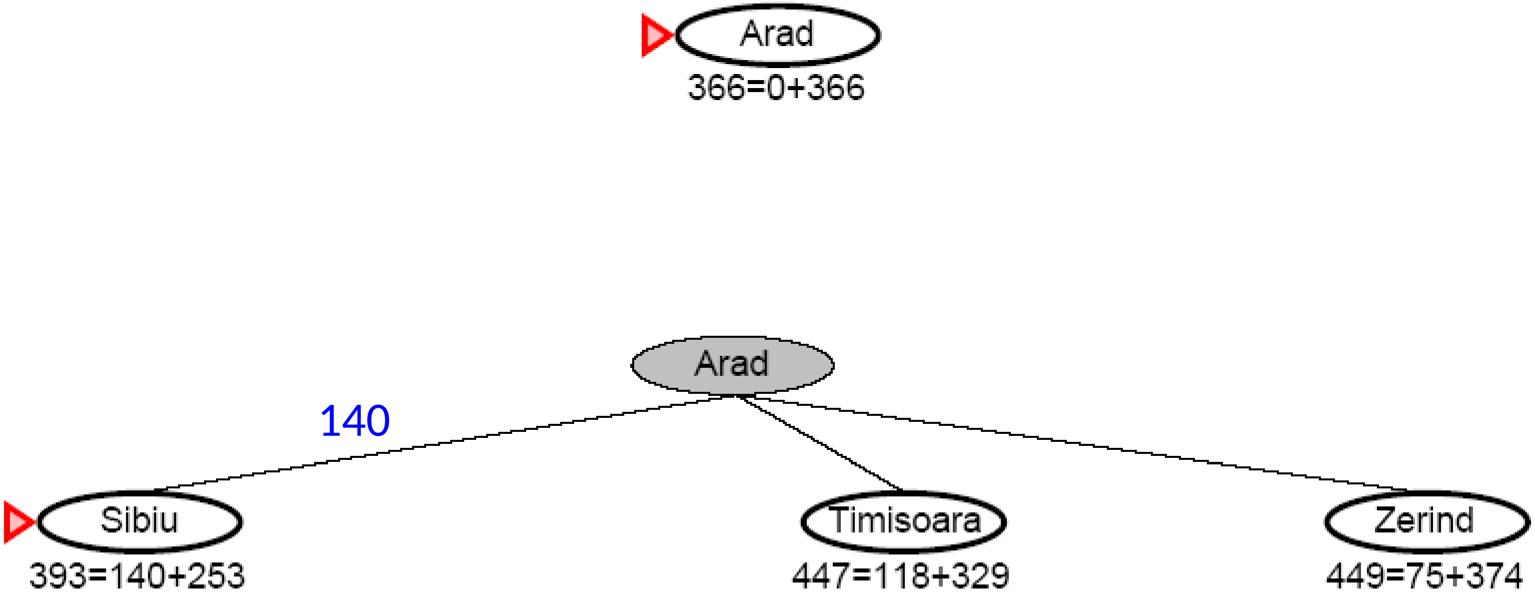




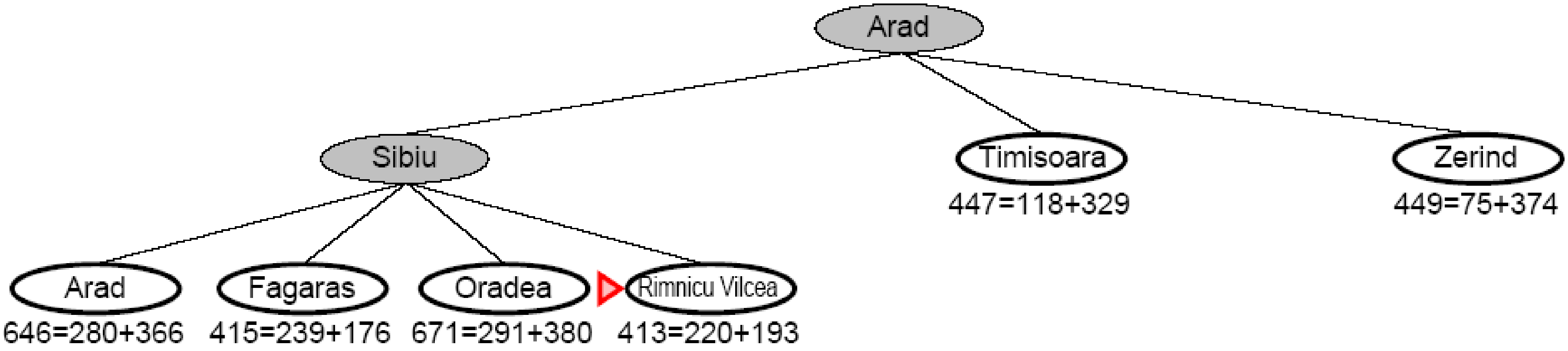
Straight–line distance
to Bucharest

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

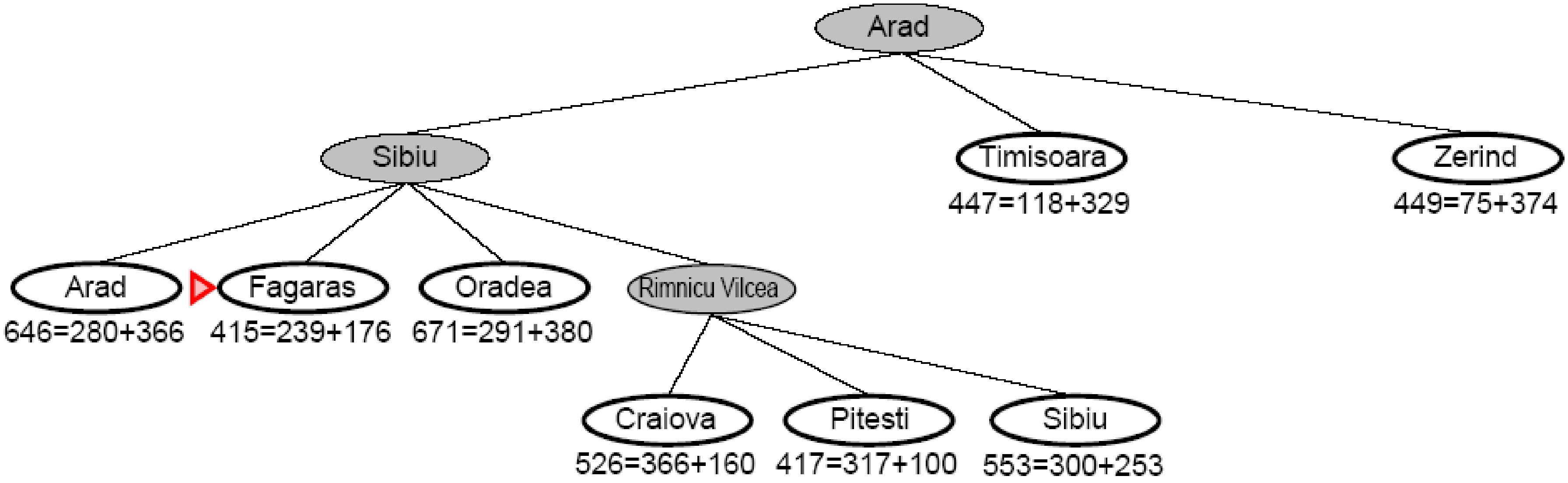
A* Search


Arad
366=0+366
Arad
140
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374

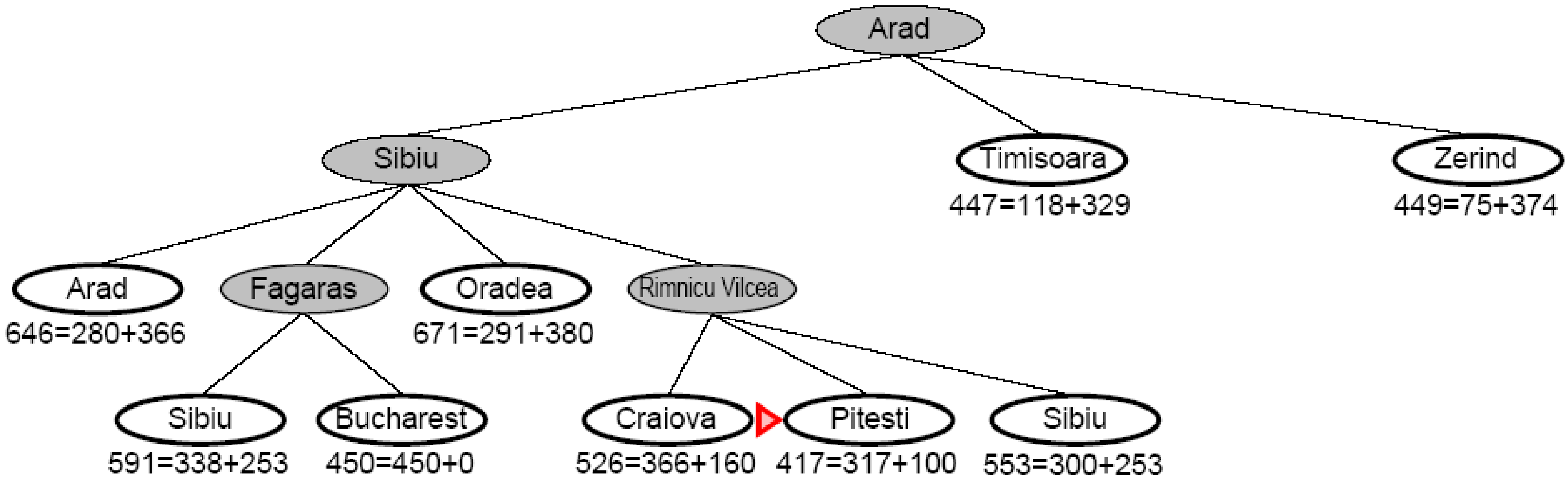
A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193

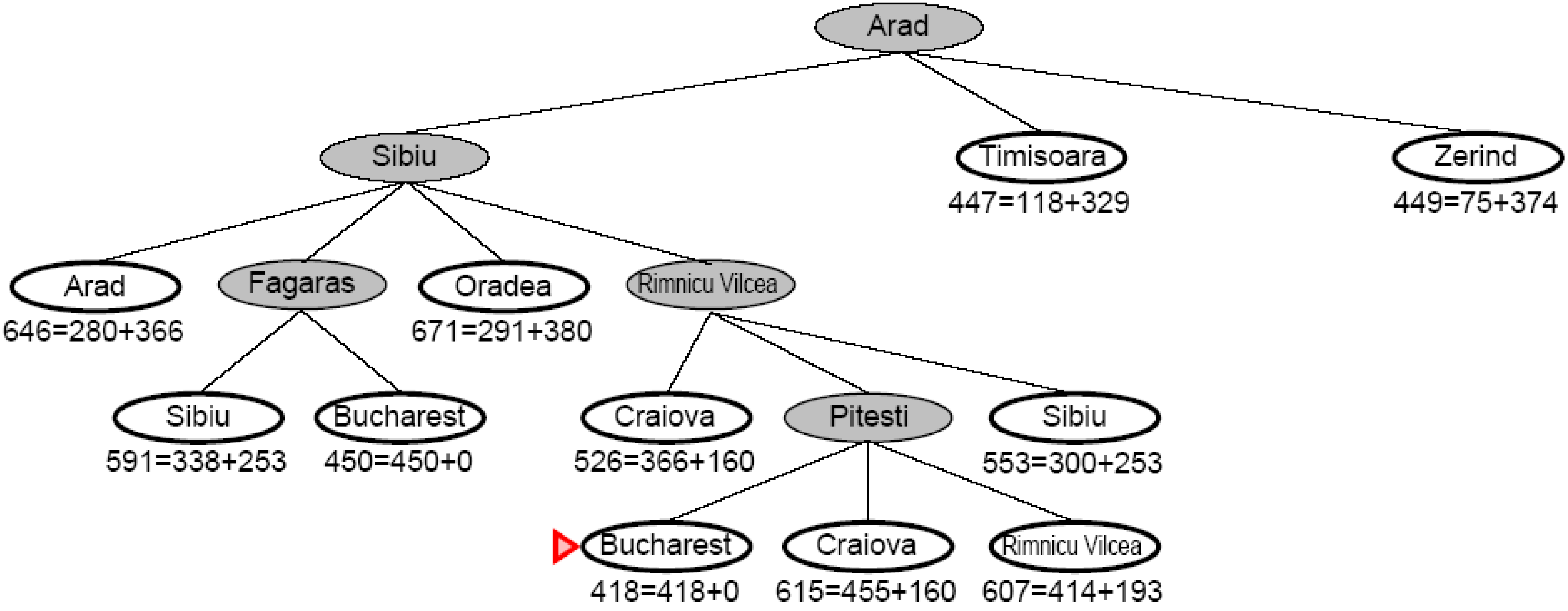
A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

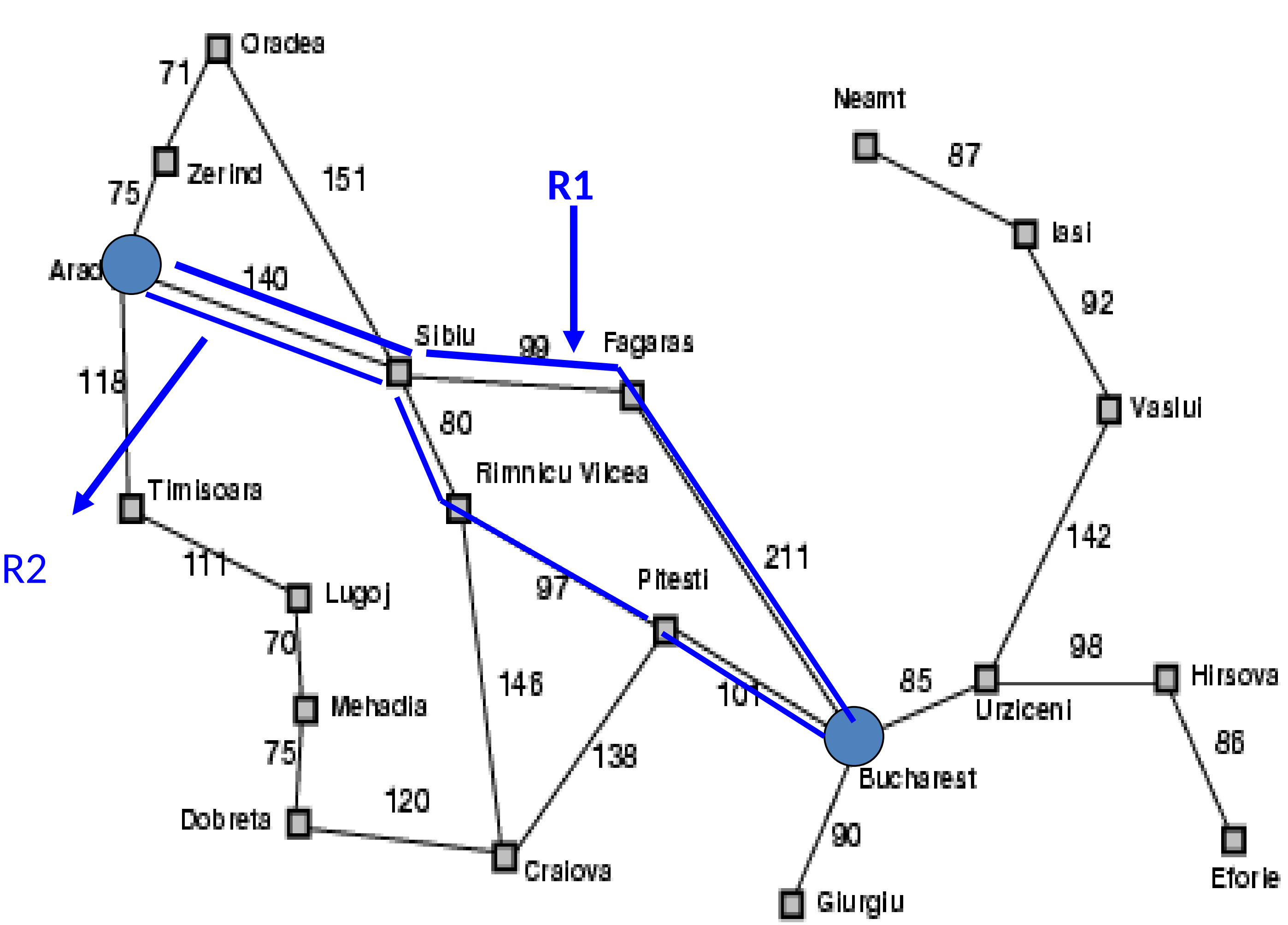
A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

A* Search


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vilcea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
Sibiu
553=300+253
Bucharest
418=418+0
Craiova
615=455+160
Rimnicu Vilcea
607=414+193

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*

- AO* algorithm is a best first search algorithm
- AO* algorithm uses the concept of AND-OR graphs to decompose any complex problem given into smaller set of problems which are further solved.
- AND-OR graphs are specialized graphs that are used in problems that can be broken down into

AO*

- AND side of the graph represent a set of task that need to be done to achieve the main goal
- or side of the graph represent the different ways of performing task to achieve the same main goal.

Problem decomposition into an and-or graph

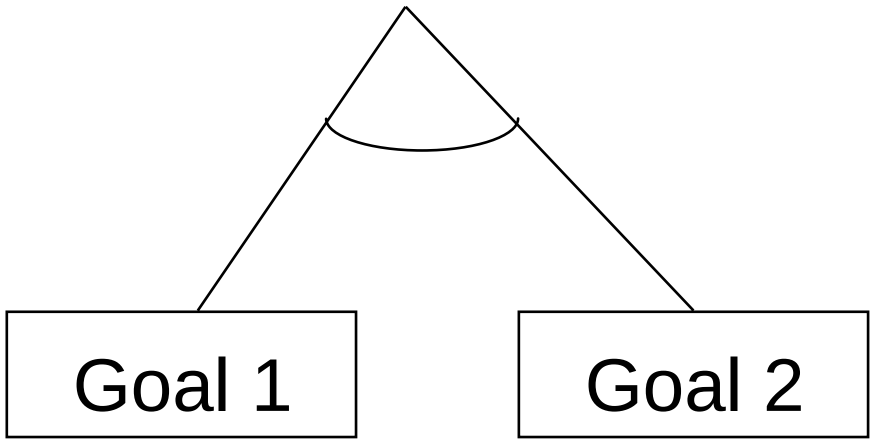
- A technique for reducing a problem to a production system, as follows:
  - The principle goal is identified; it is split into two or more sub-goals; these, too are split up.
  - A goal is something you want to achieve. A sub-goal is a goal that must be achieved in order for the main goal to be achieved.

Problem decomposition into an and-or graph

- A graph is drawn of the goal and sub-goals.
- Each goal is written in a box, called a node, with its subgoals underneath it, joined by links.

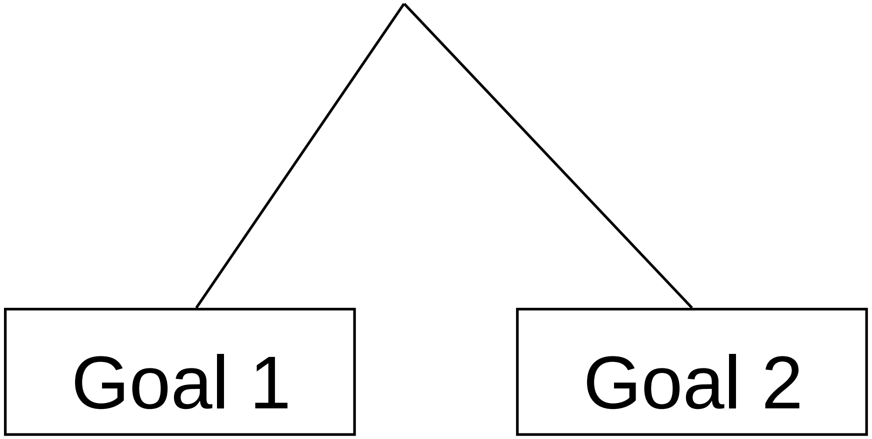
Problem decomposition into an and-or graph

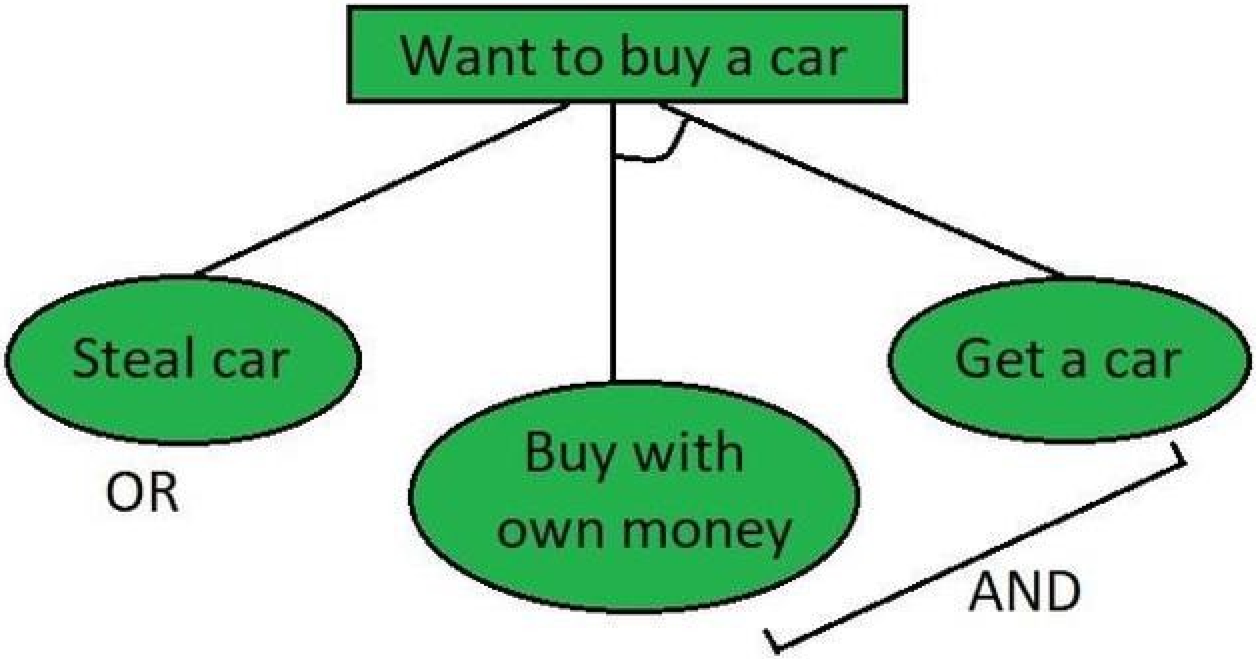
- A goal may be split into 2 (or more) sub-goals, BOTH of which must be satisfied if the goal is to succeed; the links joining the goals are marked with a curved line, like this:



Problem decomposition into an and-or graph

- Or a goal may be split into 2 (or more) sub-goals, EITHER of which must be satisfied if the goal is to succeed; the links joining the goals aren't marked with a curved line:




Want to buy a car
Steal car
OR
Buy with own money
Get a car
AND

Working of AO* algorithm:

The evaluation function in AO* looks like this:

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

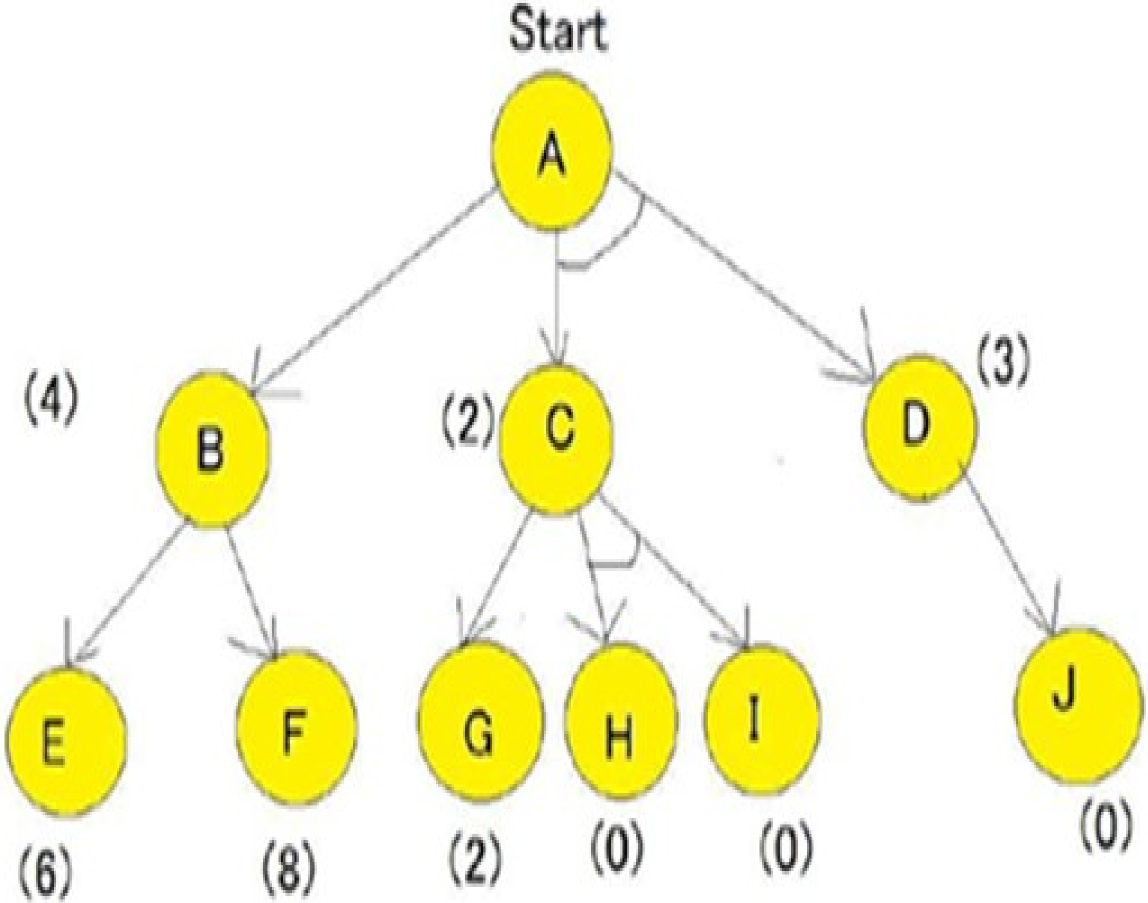
f(n) = Actual cost + Estimated cost

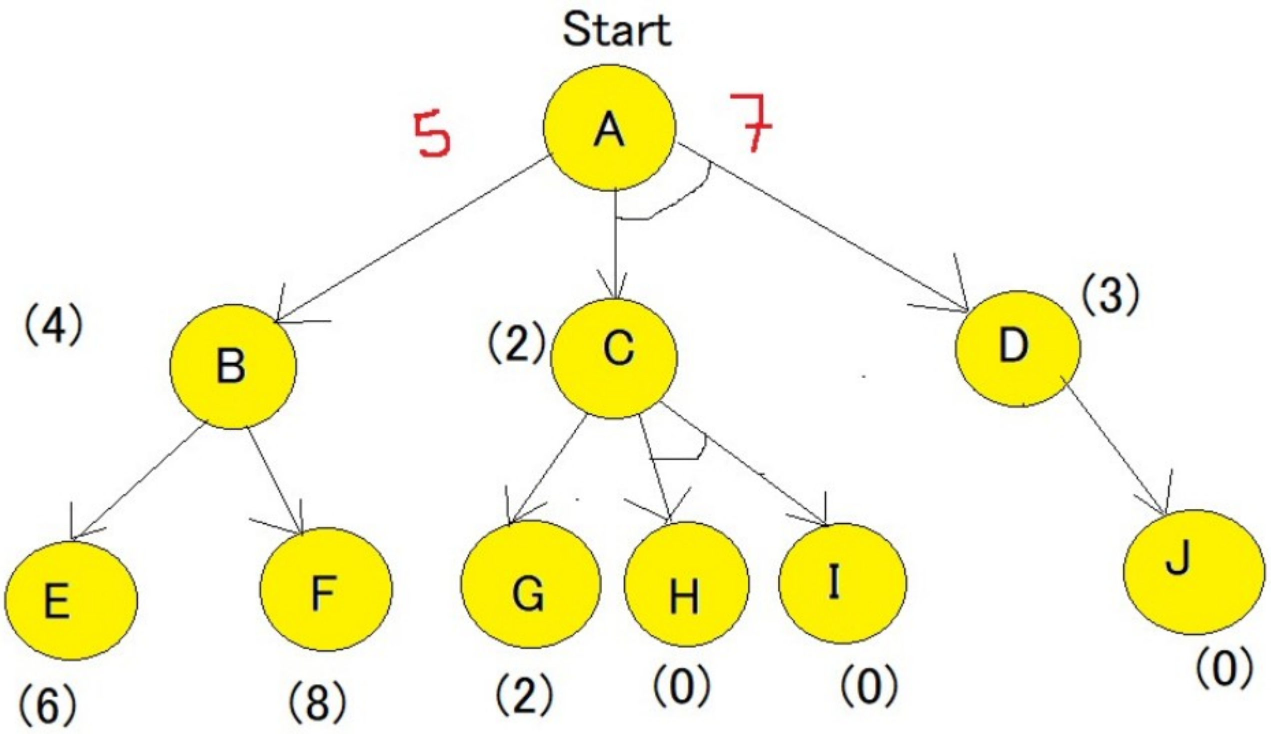
here,

f(n) = The actual cost of traversal.

g(n) = the cost from the initial node to the current node.

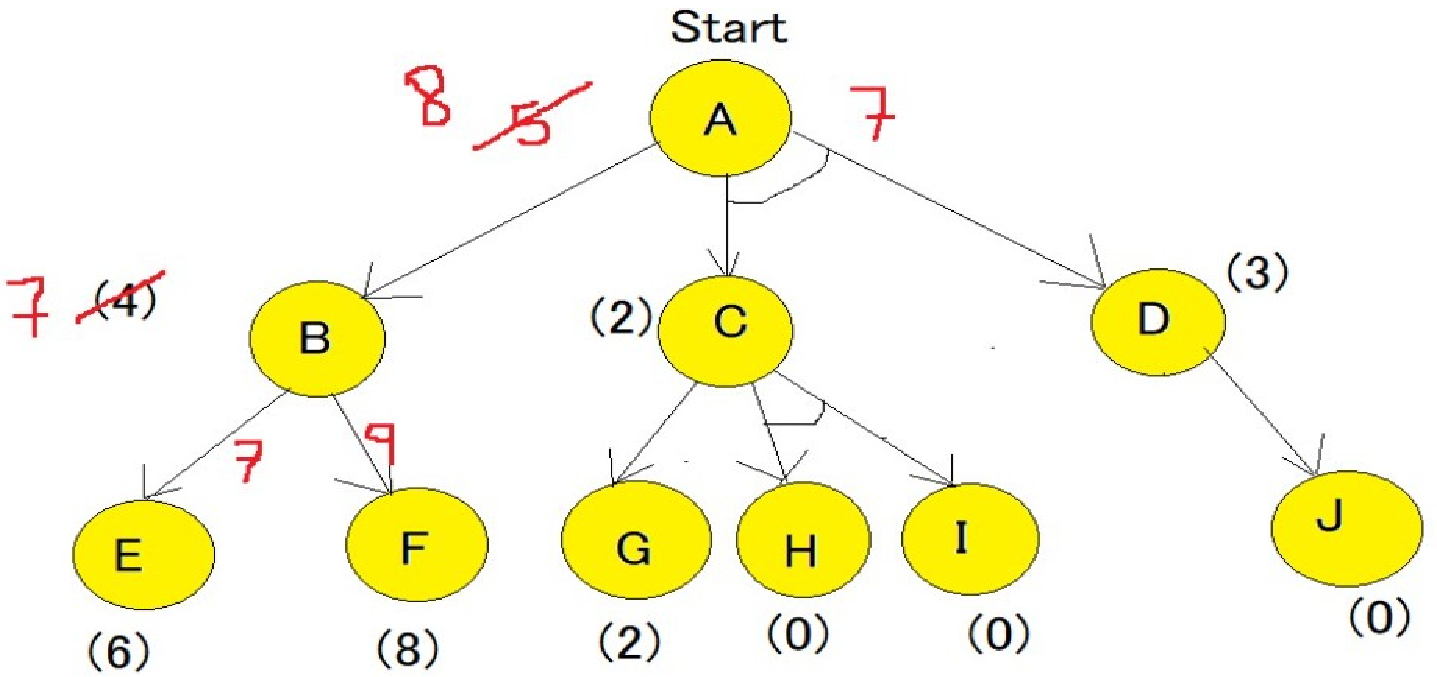
h(n) = estimated cost from the current node to the goal state.


Start
A
(4)
B
(2)
C
D
(3)
E
F
G
H
I
J
(6)
(8)
(2)
(0)
(0)
(0)



f(A-B) = g(B) + h(B) = 1+4= 5

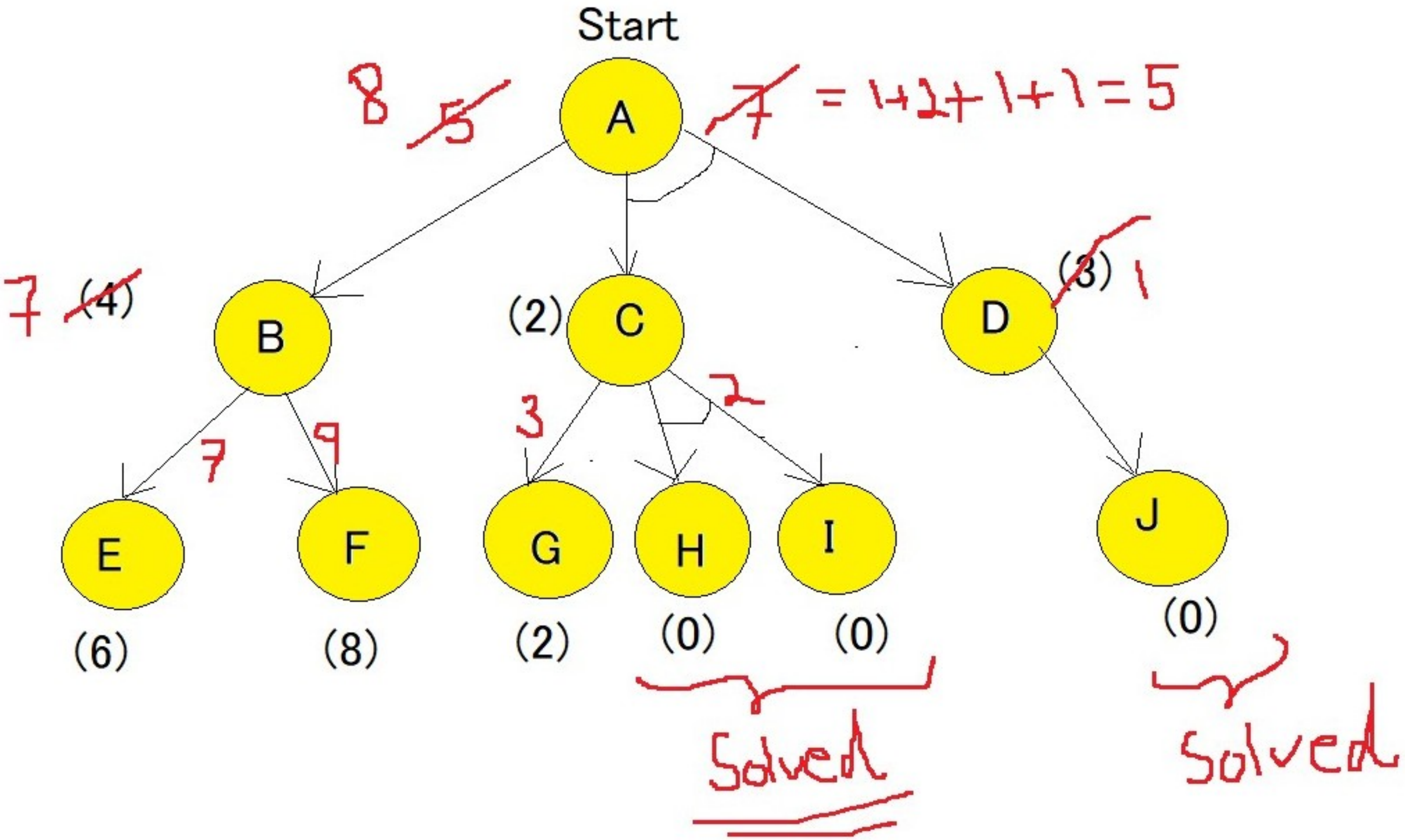
f(A-C-D) = g(C) + h(C) + g(D) + h(D) = 1+2+1+3 = 7



f(B-E) = 1 + 6 = 7.

f(B-F) = 1 + 8 = 9

f(A-B) = g(B) + updated((h(B)) = 1+7=8



f(C-G) = 1+2 = 3

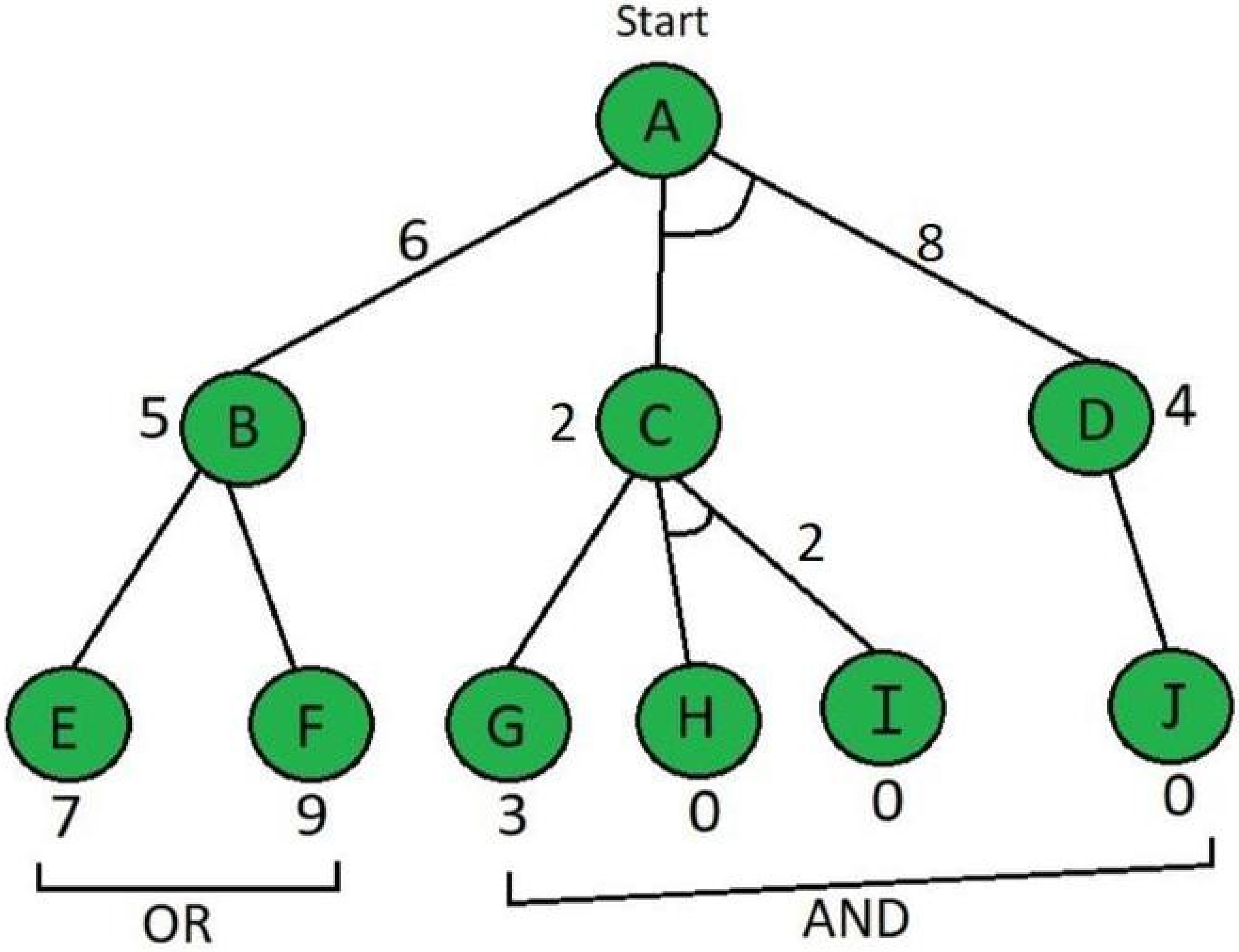
f(D-J) = 1+0 = 1

f(C-H-I) = 1+0+1+0 = 2

f(A-C-D) = g(C) + h(C) + g(D) + updated((h(D)) = 1+2+1+1 =5

Difference between the A* Algorithm and AO* algorithm

- A* algorithm and AO* algorithm both works on the **best first search**.
- They are both **informed search** and works on given heuristics values.
- A* always gives the optimal solution but AO* doesn’t guarantee to give the optimal solution.
- Once AO* got a solution **doesn’t explore** all possible paths but A* explores all paths.
- When compared to the A* algorithm, the AO* algorithm uses **less memory**.


Start
A
6
8
5
B
2
C
D
4
2
E
F
G
H
I
J
7
9
3
0
0
0
OR
AND

Local Search Algorithm

Local search algorithms and optimization

- Systematic search algorithms
  - to find the goal and to find the path to that 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

Local Search Algorithms

- Instead of considering the whole state space, consider only the current state

- Limits necessary memory; paths not retained

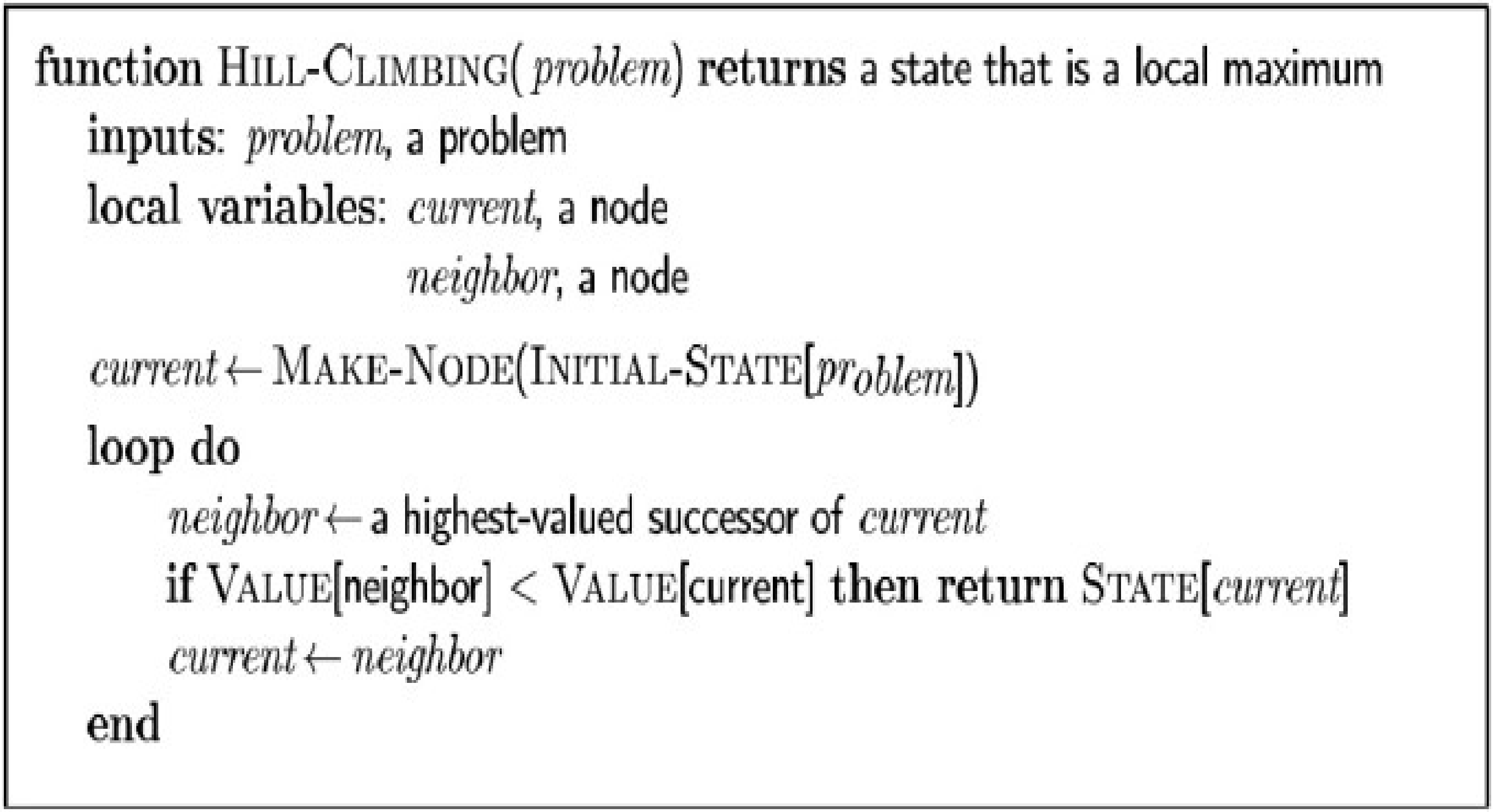
- Amenable to large or continuous (infinite) state spaces where exhaustive algorithms aren’t possible
- **Local search algorithms can’t backtrack!**

What we think hill-climbing looks like

What we learn hill-climbing is Usually like

Hill-climbing search

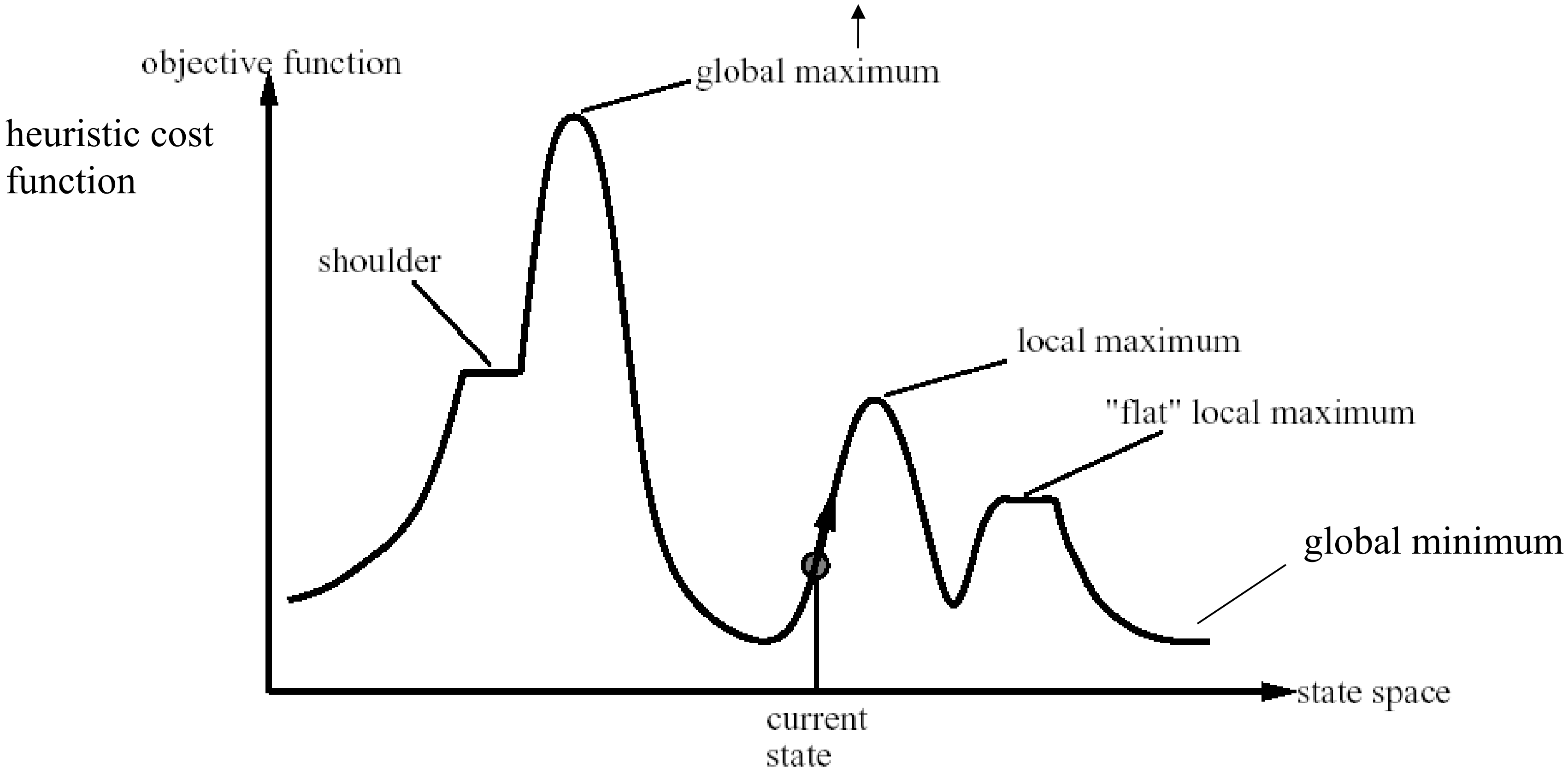
- moves in the direction of increasing value until a "peak"
  - current node data structure only records the state and its objective function
  - neither remember the history nor look beyond the immediate neighbors



```
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 ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
       neighbor ← a highest-valued successor of current
       if VALUE[neighbor] < VALUE[current] then return STATE[current]
       current ← neighbor
   end
```

Local search – state space landscape

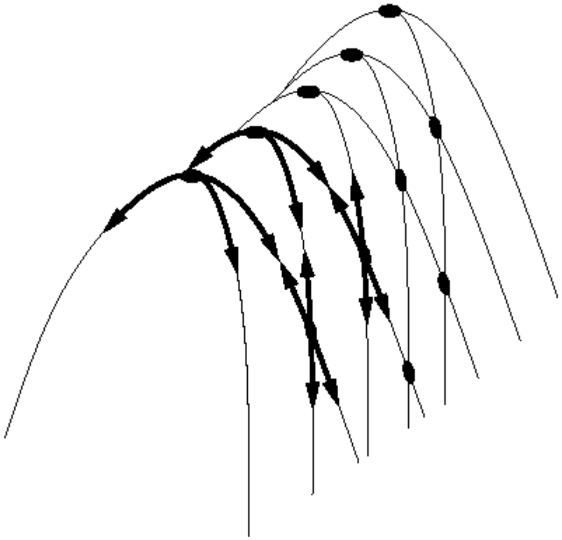
- Elevation = the value of the objective function or heuristic cost function

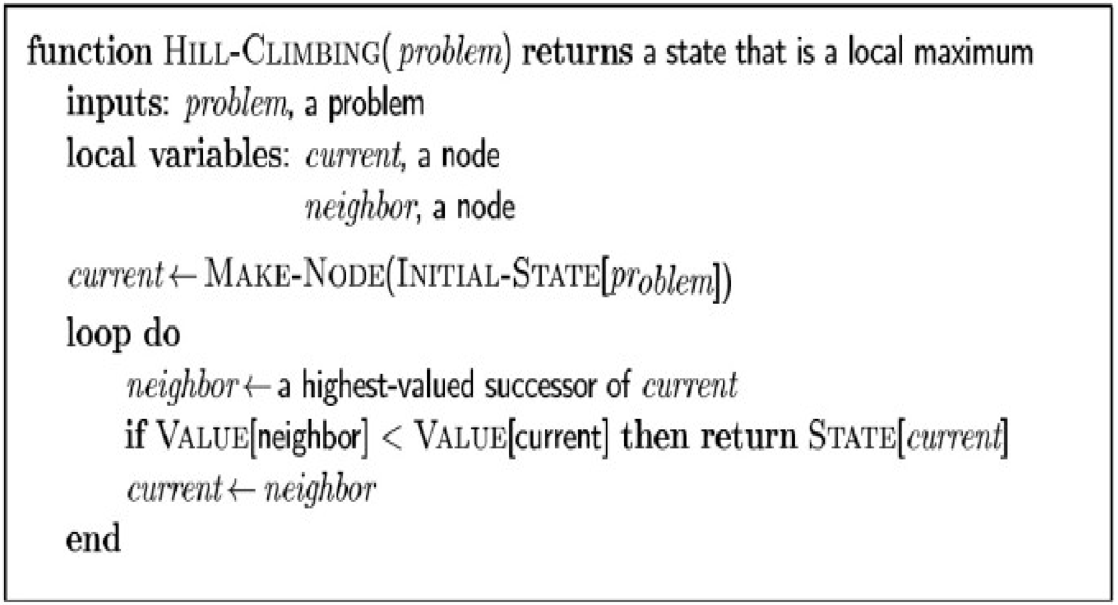


- A complete local search algorithm finds a solution if one exists
- A optimal algorithm finds a global minimum or maximum

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

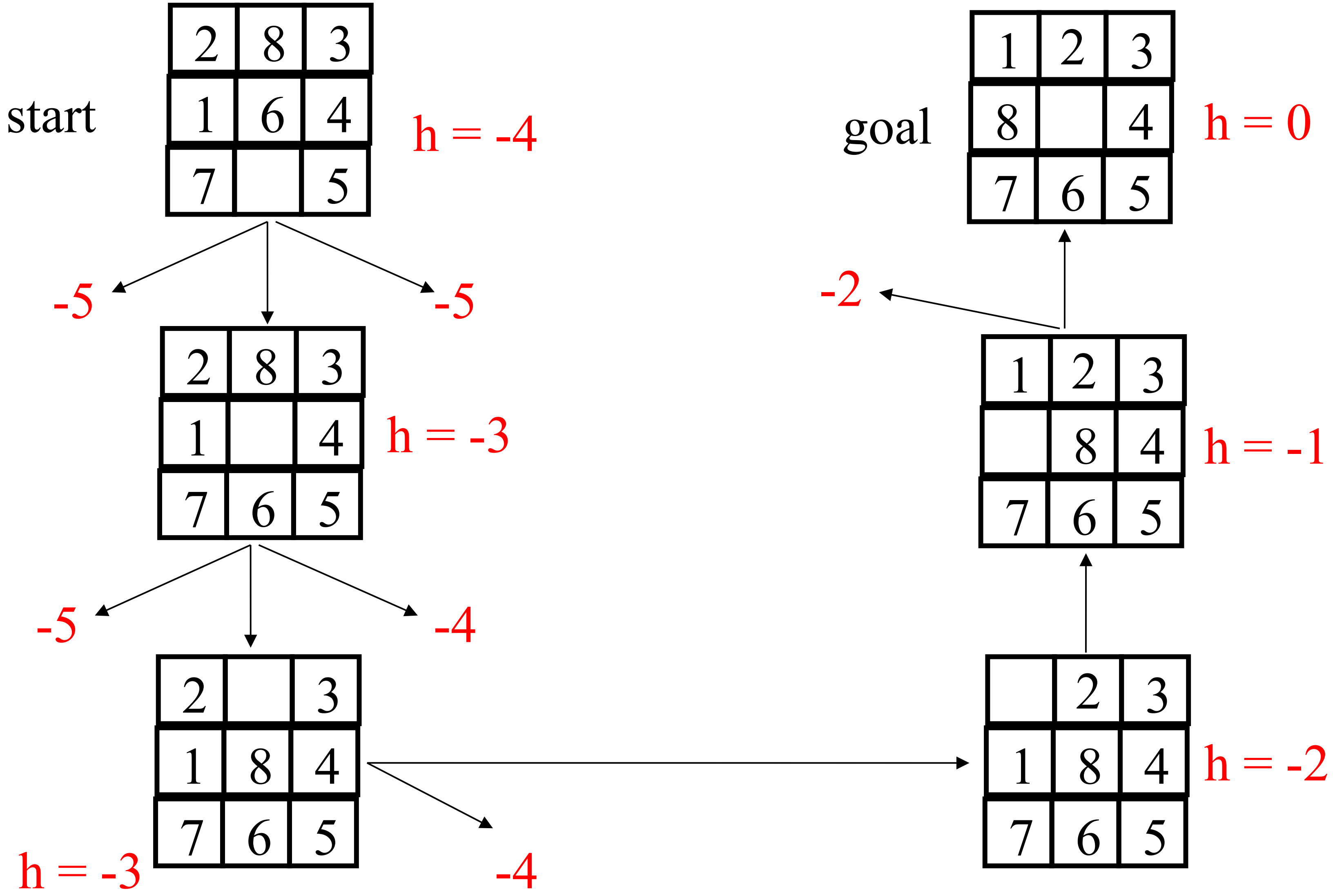




```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] < VALUE[current] then return STATE[current]
        current ← neighbor
    end
```

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

Hill climbing example



f(n) = -(number of tiles out of place)

Hill Climbing Search

- Variants of Hill climbing
  - Stochastic 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

Game Playing

- Minimax algorithm
- alpha beta cut offs

Game Playing

Why do AI researchers study game playing?

1. It’s a good reasoning problem, formal and nontrivial.

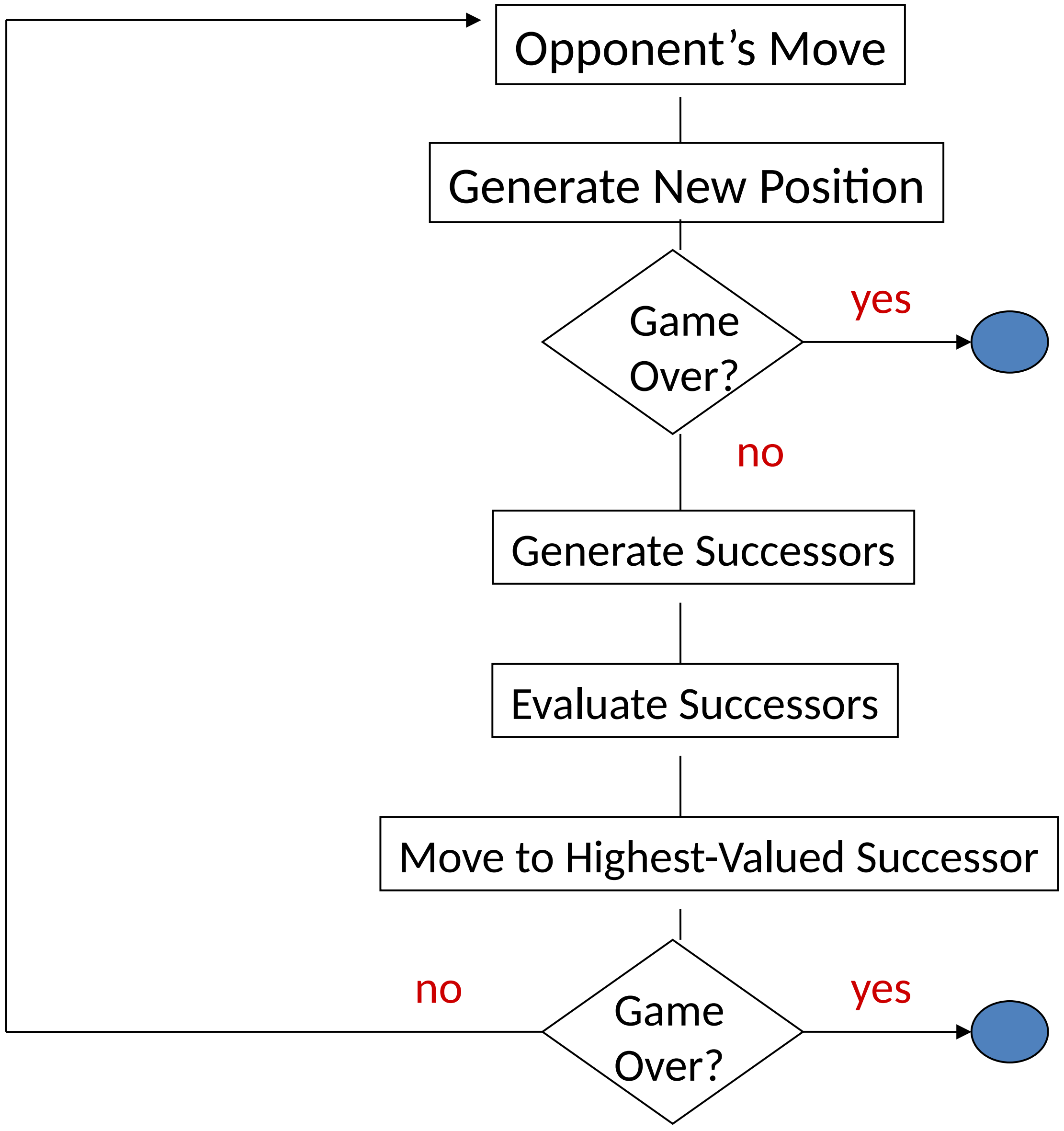
2. Direct comparison with humans and other computer programs is easy.

What Kinds of Games?

Mainly games of strategy with the following characteristics:

1. Sequence of moves to play
2. Rules that specify possible moves
3. Rules that specify a payment for each move
4. Objective is to maximize your payment

Two-Player Game



Opponent's Move

Generate New Position

Game Over?

yes

no

Generate Successors

Evaluate Successors

Move to Highest-Valued Successor

no

Game Over?

yes

Mini-Max Terminology

- utility function: the function applied to leaf nodes

- backed-up value
  - of a max-position: the value of its largest successor
  - of a min-position: the value of its smallest successor

- minimax procedure: search down several levels; at the bottom level apply the utility function, back-up values all the way up to the root node, and that node selects the move.

Game Tree (2-player, Deterministic, Turns)

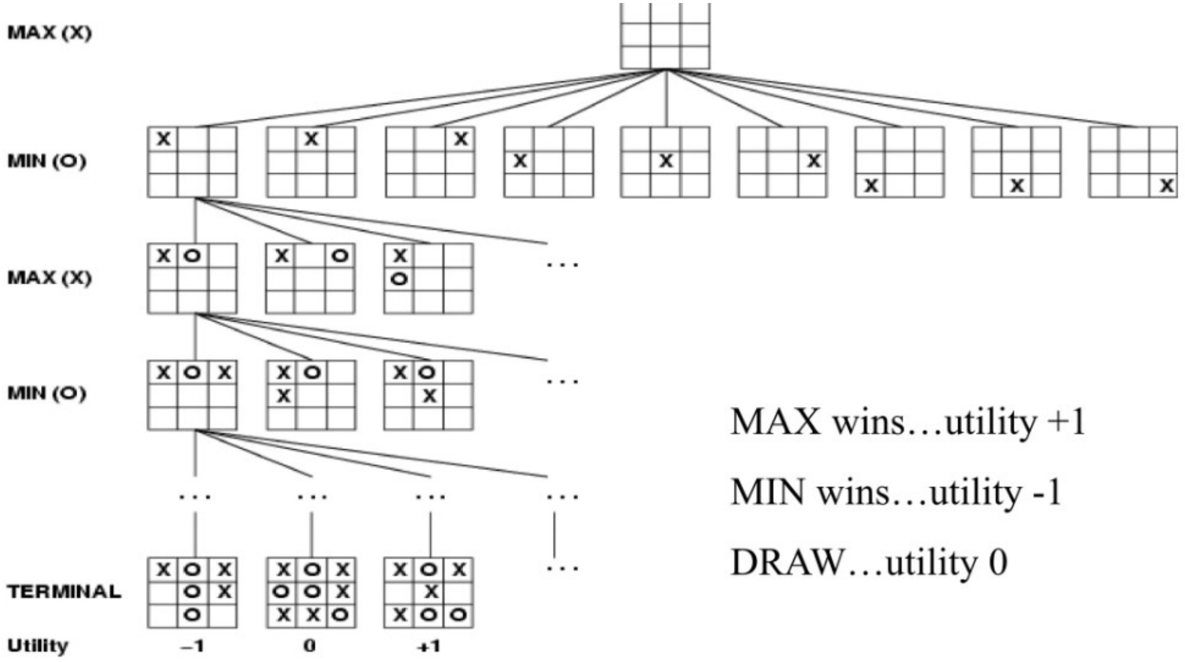
computer's turn

opponent's turn

computer's turn

opponent's turn

leaf nodes are evaluated



MAX wins…utility +1

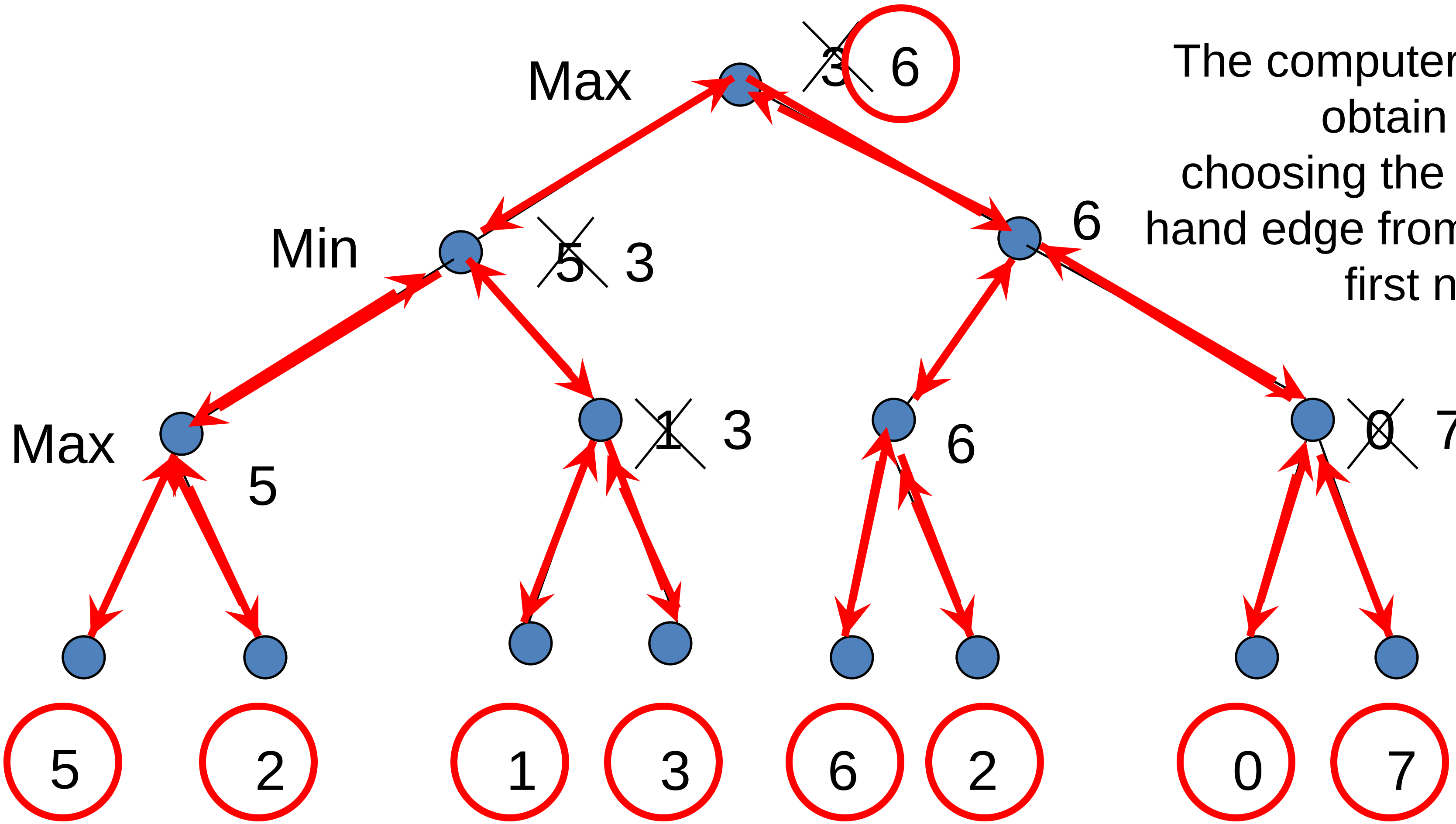
MIN wins…utility -1

DRAW…utility 0

The computer is **Max**.
The opponent is **Min**.

At the leaf nodes, the **utility function** is employed. Big value means good, small is bad.

Minimax – Animated Example



The computer can obtain 6 by choosing the right hand edge from the first node.

Minimax Strategy

- Why do we take the min value every other level of the tree?
- These nodes represent the opponent's choice of move.
- The computer assumes that the human will choose that move that is of least value to the computer.

Minimax Function

- MINIMAX-VALUE(*n*) = UTILITY(*n*)

  if *n* is a terminal state

- max$_{s \in \mathit{Successors}(n)}$ MINIMAX-VALUE(s)

  if *n* is a MAX node

  mins ∈ Successors(n) MINIMAX-VALUE(s)
  if n is a MIN node

Properties of Minimax

- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? $O(b^m)$
- Space complexity? O(bm) (depth-first exploration)

- For chess, b ≈ 35, m ≈100 for "reasonable" games
  ✉ exact solution completely infeasible

Need to speed it up.

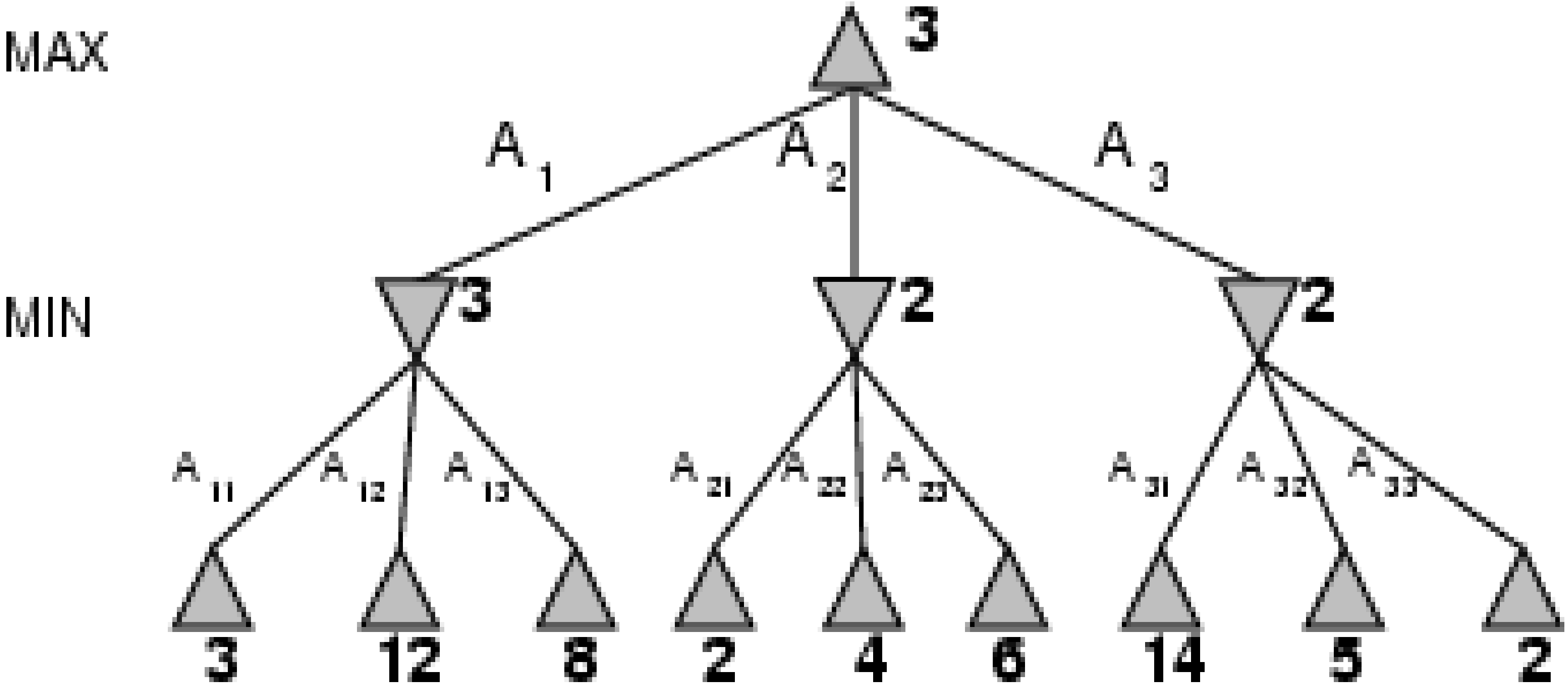
Searching Game Trees

- Exhaustively searching a game tree is not usually a good idea.
- Even for a simple tic-tac-toe game there are over 350,000 nodes in the complete game tree.
- An additional problem is that the computer only gets to choose every other path through the tree – the opponent chooses the others.

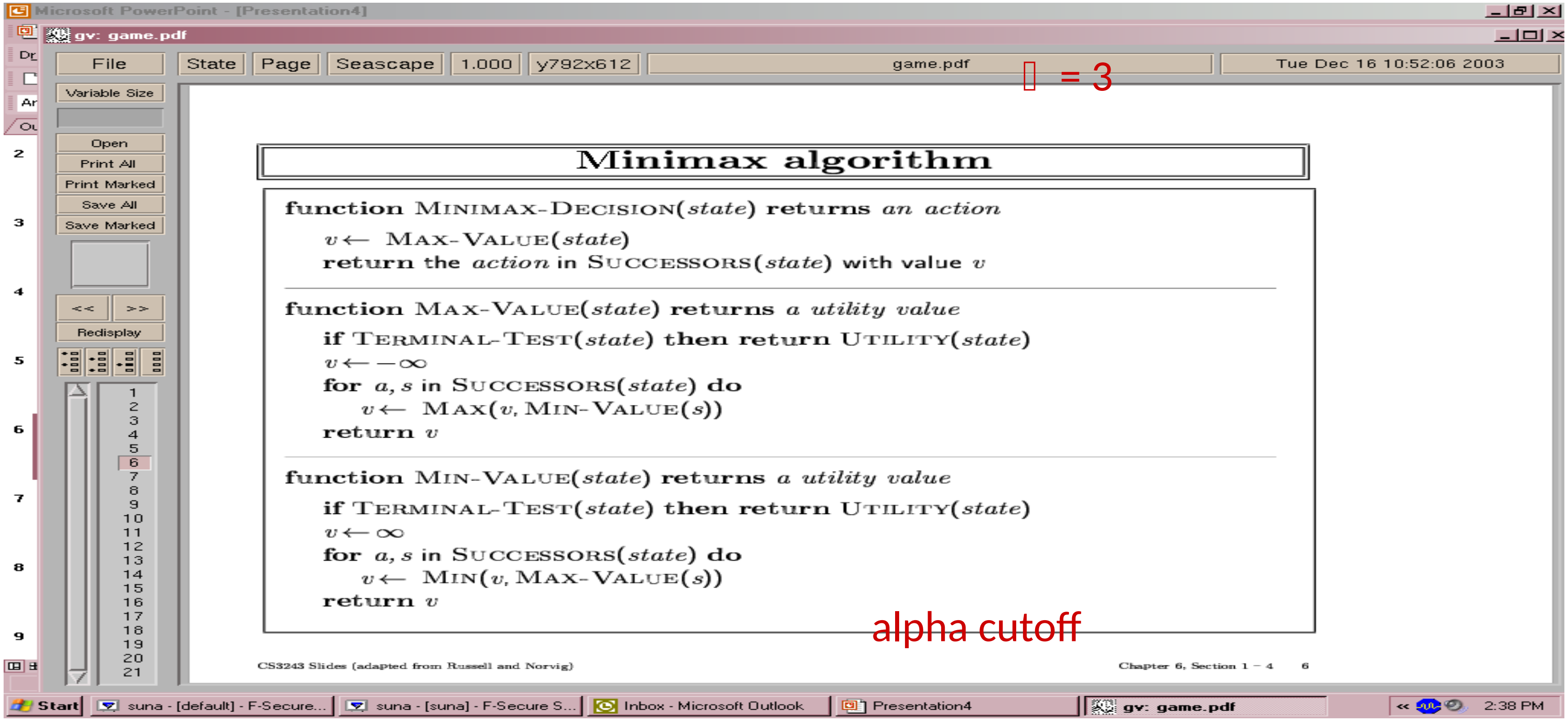
Alpha-beta Pruning

- A method that can often cut off a half the game tree.
- Based on the idea that if a move is clearly bad, there is no need to follow the consequences of it.
- alpha – highest value we have found so far
- beta – lowest value we have found so far

α-β pruning example



α-β pruning example



□ = 3

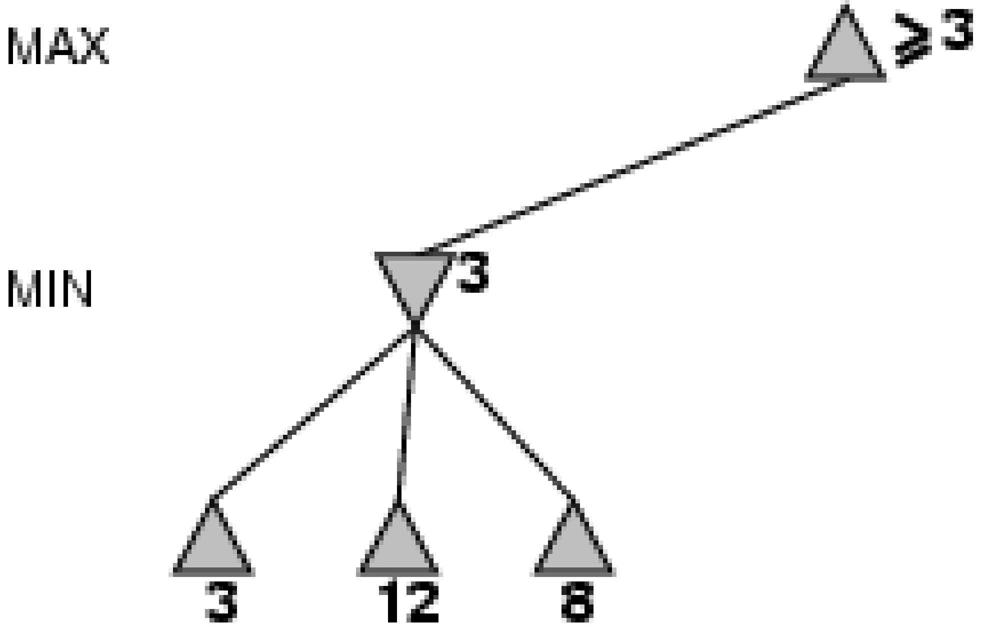
Minimax algorithm

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

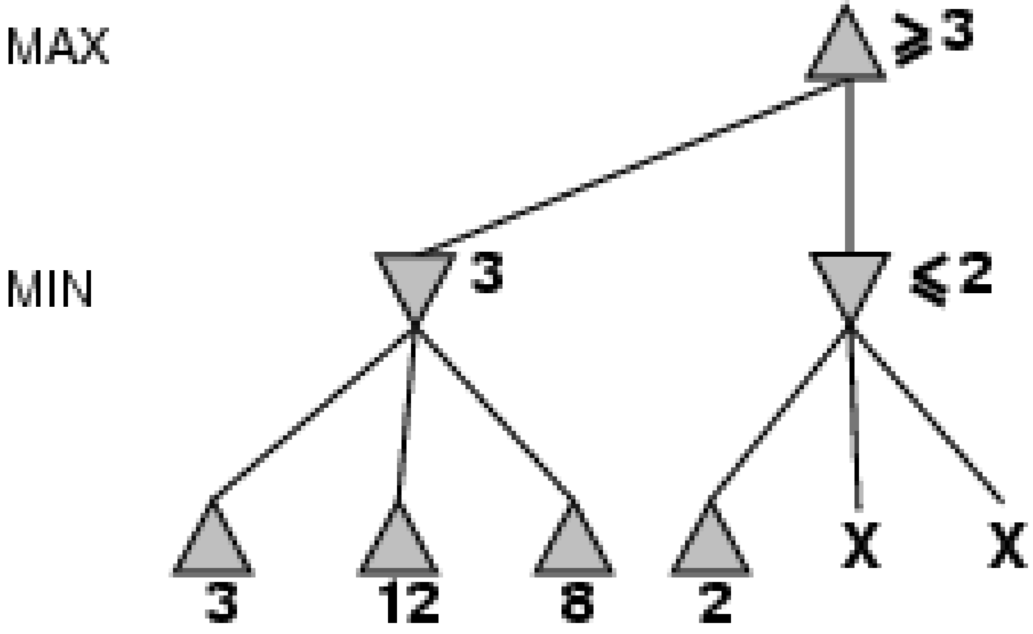
alpha cutoff

CS3243 Slides (adapted from Russell and Norvig)   Chapter 6, Section 1 − 4   6

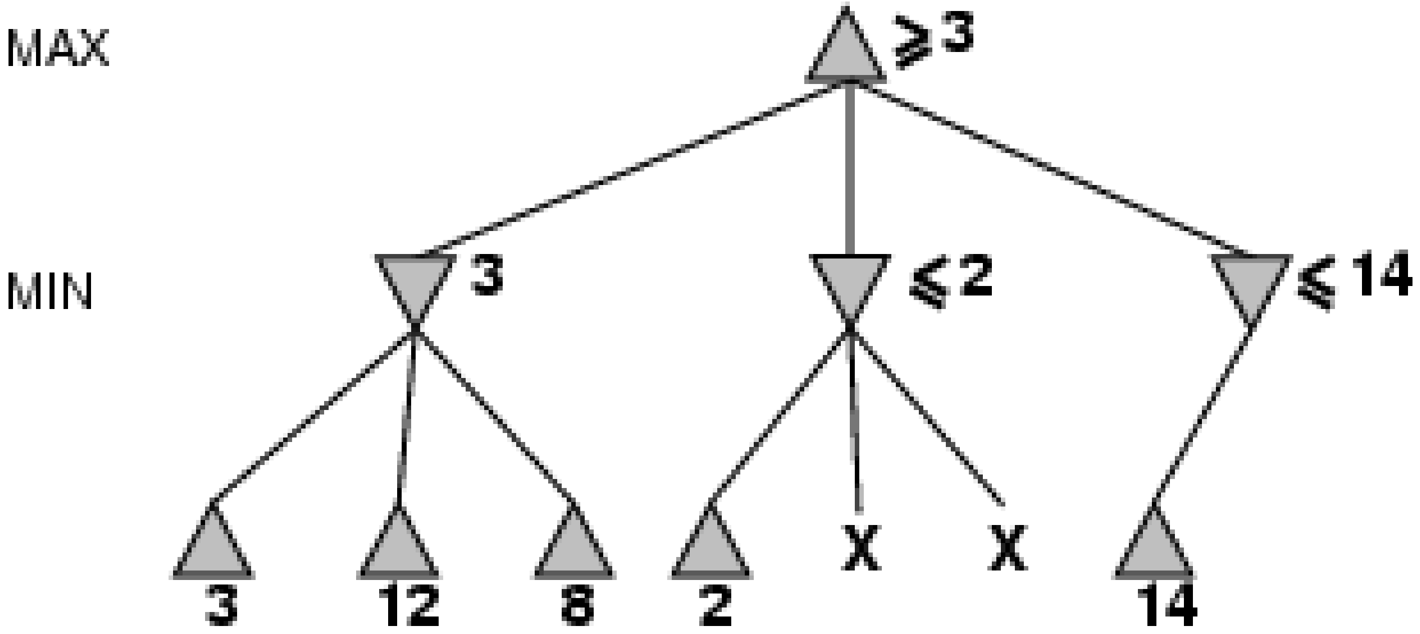
α-β pruning example



α-β pruning example



α-β pruning example

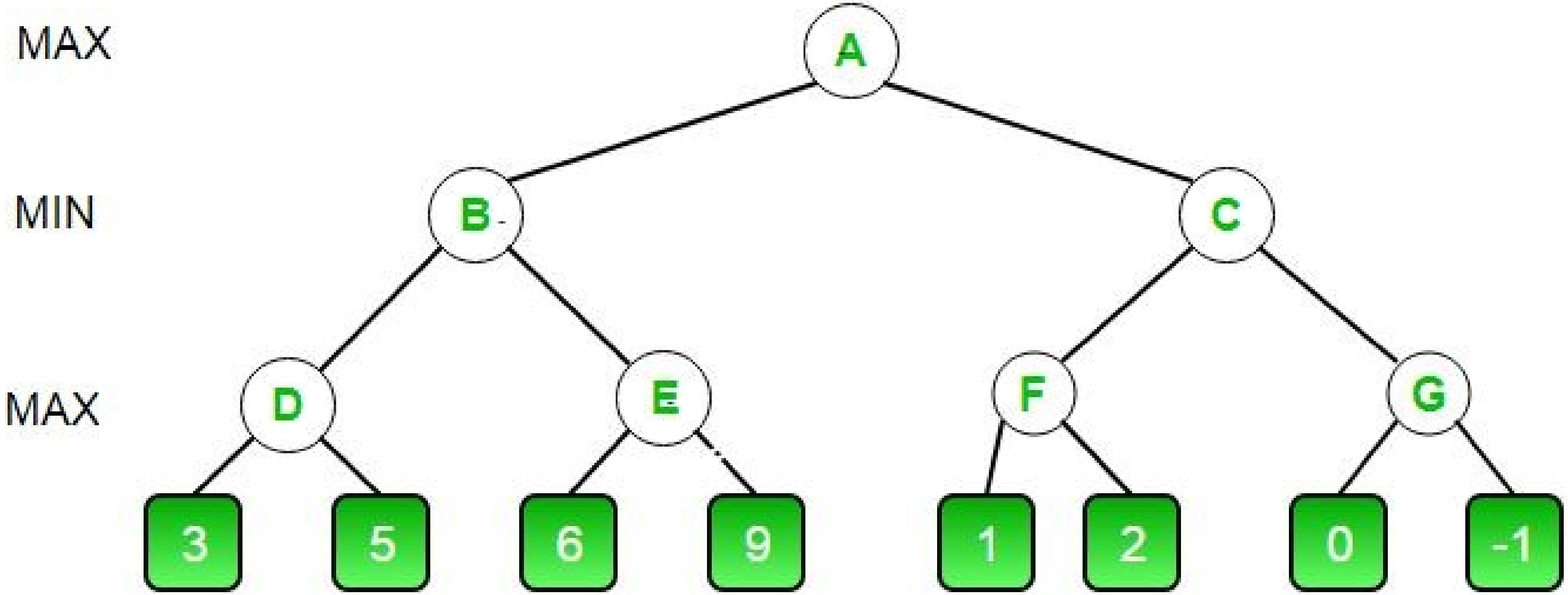


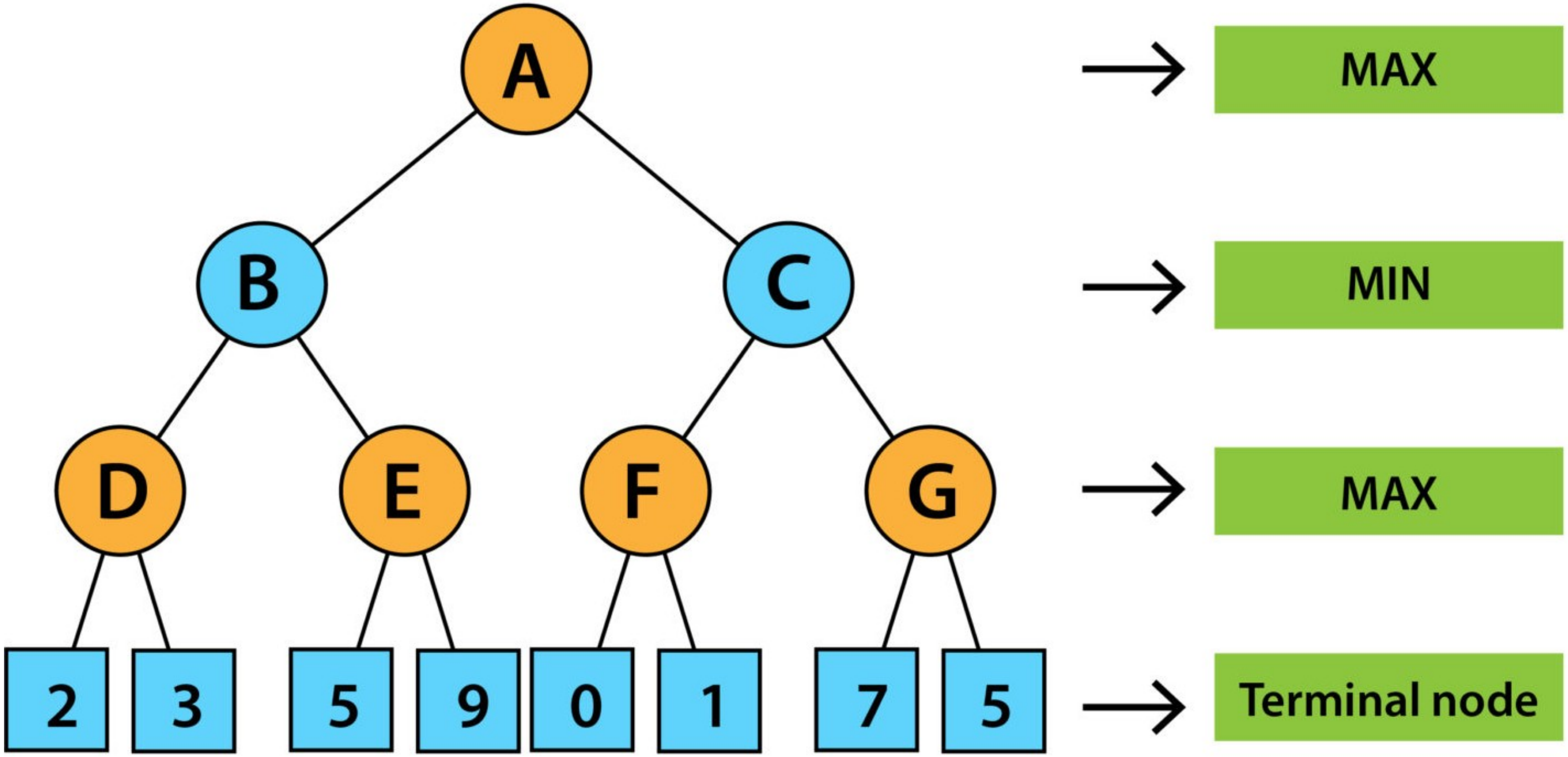
Properties of α-β

- 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


MAX
MIN
MAX
A
B
C
D
E
F
G
3
5
6
9
1
2
0
-1


A
B
C
D
E
F
G
2
3
5
9
0
1
7
5
MAX
MIN
MAX
Terminal node

Thank you!