Session – 1

COURSE HAND OUT EXPLANATION

&

Introduction to Artificial Intelligence

Session Outcomes

- Student will learn the COs, syllabus and evaluation plan of the course.

- Student will understand the importance of AI.

What is AI?

•The following slide shows eight definitions of AI, laid out along two dimensions.

• The definitions on top are concerned with thought processes and reasoning, whereas the ones on the bottom address behavior.

• The definitions on the left measure success in terms of fidelity to human performance whereas the ones on the right measure against an ideal performance measure, called rationality.

Artificial Intelligence

Thinking Humanly	Thinking Rationally
“The exciting new effort to make computers think . . . machines with minds, in the full and literal sense.” (Haugeland, 1985) “[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . .” (Bellman, 1978)	“The study of mental faculties through the use of computational models.” (Charniak and McDermott, 1985) “The study of the computations that make it possible to perceive, reason, and act.” (Winston, 1992)
Acting Humanly	**Acting Rationally**
“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)	“Computational Intelligence is the study of the design of intelligent agents.” (Poole et al., 1998) “AI . . . is concerned with intelligent behavior in artifacts.” (Nilsson, 1998)

Applications of AI

Mundane Tasks

- Perception
  - Vision
  - Speech
- Natural language
  - Understanding
  - Generation
  - Translation
- Commonsense reasoning
  - Robot control

Formal Tasks

- Games
- Chess
- Backgammon
- Checkers – Go

Mathematics

- Geometry
- Logic
- Integral Calculus
- Proving properties of programs

Expert Systems

- Engineering
  - Design
  - Fault finding
  - Manufacturing planning
- Scientific analysis
- Medical diagnosis
- Financial analysis

Applications of AI (continued..):

Artificial Neural Networks: Neural Networks (NN), also known as artificial neural networks (ANN), are computational models, essentially algorithms. Neural networks have a unique ability to extract meaning from imprecise or complex data to find patterns and detect trends that are too convoluted for the human brain or for other computer techniques.

- Natural Language Processing :
  - Automated grammar correction, Translation from one language to another language
- Engineering Applications:
  - flight control, chemical engineering, power plants, automotive control, medical systems.
- Business applications:
  - fund analytics, marketing segmentation, and fraud detection, medical systems

Fuzzy Systems:

The term Fuzzy means something that is a bit vague. An algorithm based on Fuzzy Logic takes all available data while solving a problem. It then takes the best possible decision according to the given input.

- Medicine:
  - Controlling arterial pressure when providing anesthesia to patients
- Transportation systems:
  - Handling underground train operations
- Defense:
  - Locating and recognizing targets underwater
- Industry:
  - Controlling water purification plants
- Naval control:
  - Autonomous underwater vehicles are controlled using Fuzzy Logic
- Modern washing systems powered by Fuzzy Logic

Expert Systems:

Expert System is an interactive and reliable computer-based decision-making system which uses both facts and heuristics to solve complex decision-making problems.

MYCIN: It was based on backward chaining and could identify various bacteria that could cause acute infections. It could also recommend drugs based on the patient's weight.

DENDRAL: Expert system used for chemical analysis to predict molecular structure.

PXDES: Expert system used to predict the degree and type of lung cancer.

CaDet: Expert system that could identify cancer at early stages.

Self study:

- The birth of artificial intelligence
- Further Applications of AI
- Sources:
  - https://www.guru99.com/expert-systems-with-applications.html#2
  - https://www.upgrad.com/blog/fuzzy-login-in-artificial-intelligence/#:~:text=Although%20Fuzzy%20Logic%20in%20artificial,enhancing%20the%20execution%20of%20algorithms.
  - https://www.smartsheet.com/neural-network-applications
    - https://www.youtube.com/watch?v=fP5zFpsThqk
    - https://youtu.be/4jmsHaJ7xEA

End of Session

Session 2

Intelligent Agents

Session Outcomes

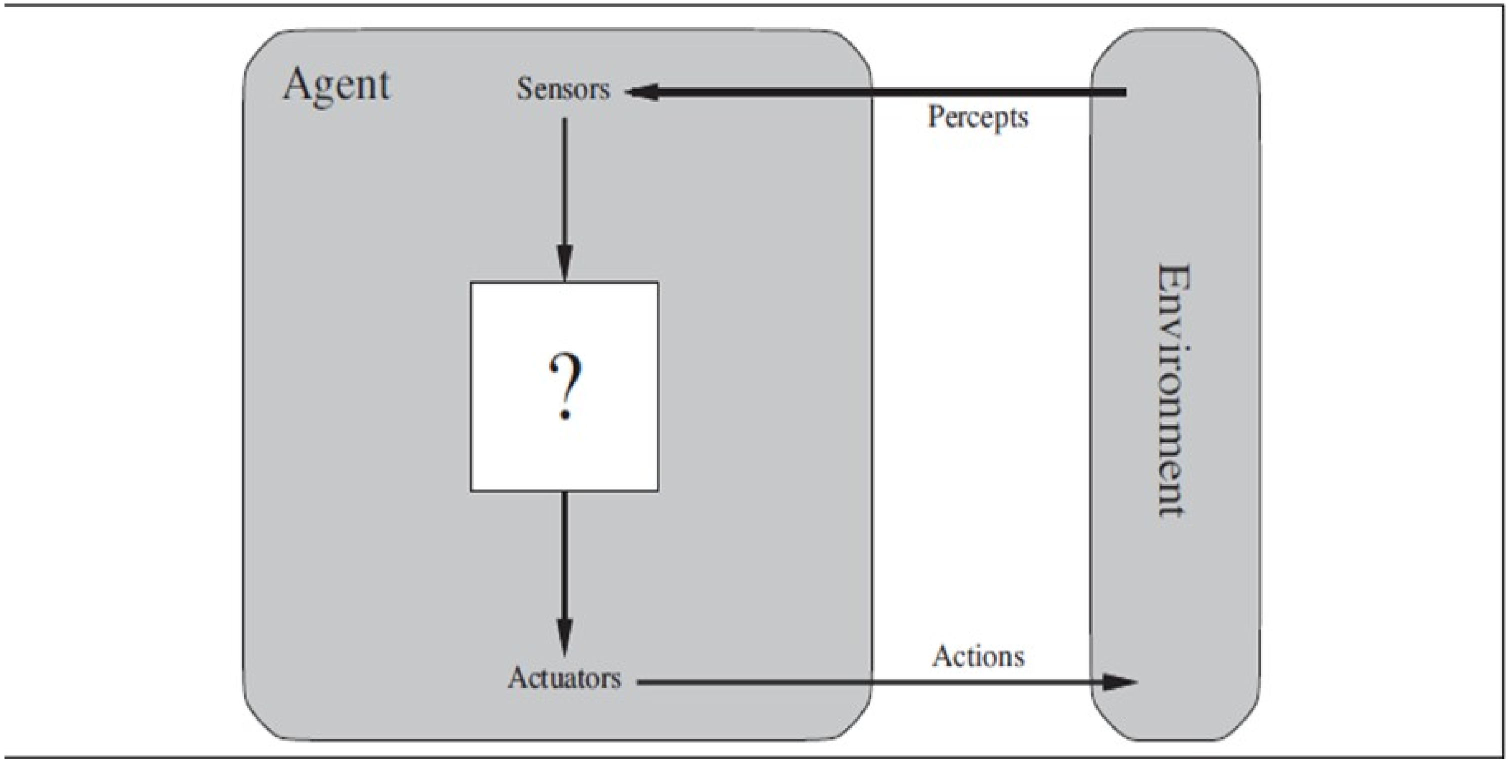
- **Student will understand Intelligent Agents & Environments AI.**

- **Student will learn the real world applications.**

Agents &Environments

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

- This simple idea is illustrated in the following figure:



Agents interact with environments through sensors and actuators.

Terminology:

- A human agent has eyes, ears, and other organs for **sensors** and hands, legs, vocal tract, and so on for **actuators**.

- A robotic agent might have cameras and infrared range finders for **sensors** and switches, motors etc., for **actuators.**

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

- **Percept Sequence:** An agent's percept sequence is the complete history of everything the agent has ever perceived.

- **Agent Function:** Agent's behavior is described by the agent function that maps any given percept sequence to an action.

- **Agent Mown:** Internally, the agent function for an artificial agent will be implemented by an AGMIT mown.

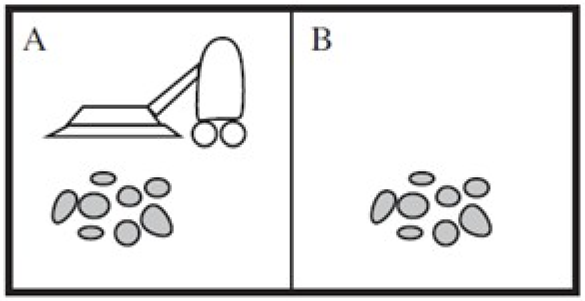
Example for Agent & Environment:

To illustrate the ideas, we use a very simple example—the **vacuum-cleaner world** shown in the following figure.

This particular world has just two locations: squares A and B.

The vacuum agent perceives which square it is in and whether there is dirt in the square.
It can choose to move left, move right, suck up the dirt, or do nothing.

One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square.



A vacuum-cleaner world with just two locations.

- We can imagine tabulating the agent function that describes any given agent

- Given an agent to experiment with, we can, in principle, construct this table by trying out all possible percept sequences and recording which actions the agent does in response)

- The table is of course, an external characterization of the agent.

- The agent function is an abstract mathematical description; the agent program is a concrete implementation, running within some physical system.

- A partial tabulation of this agent function is shown in the following figure.

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

Partial tabulation of a simple agent function for the vacuum-cleaner world

An agent program that implements is as follows:

```
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
```

GOOD BEHAVIOR: THE CONCEPT OF RATIONALITY

Rational Agent:

- A rational agent is one that does the right thing. Conceptually speaking, every entry in the table for the agent function is filled out correctly. Obviously, doing the right thing is better than doing the wrong thing, but what does it mean to do the right thing.

Performance Measure:

- When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives.

- This sequence of actions causes the environment to go through a sequence of states.

- If the sequence is desirable, then the agent has performed well. This notion of desirability is captured by a **performance measure** that evaluates any given sequence of environment states.

Rationality:

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

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

This leads to a definition of a rational agent:

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

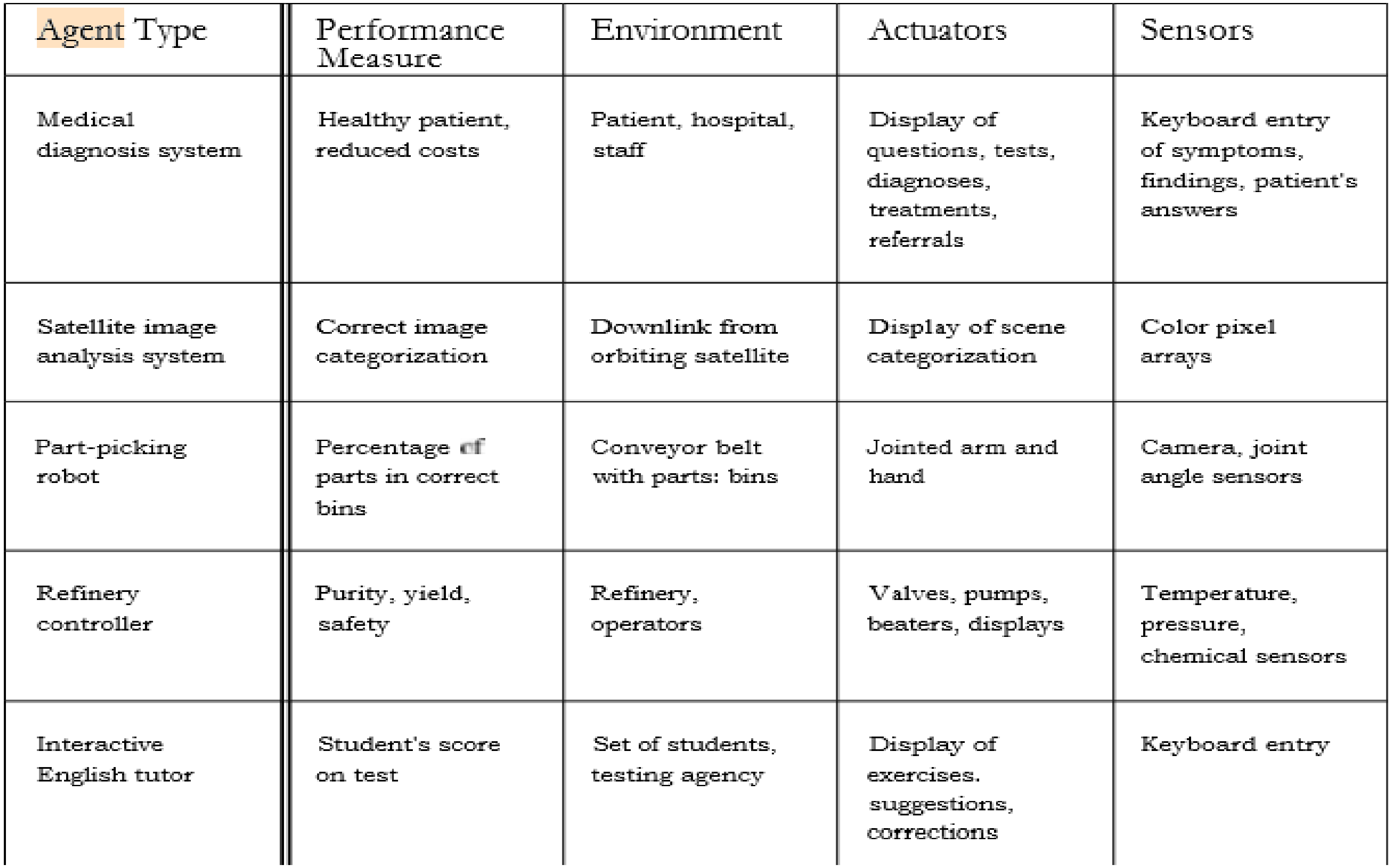
The Nature of Environments

- To develop any agent we had to specify the performance measure, the environment, the agent's actuators and sensors. We group all these under the heading of the **task environment**.
- We call this as PEAS (Performance, Environment, Actuators, Sensors) description.
- For example, the following table summarizes the PEAS description for the taxi's task environment.

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

PEAS description of the task environment for an automated taxi

we have sketched the basic PEAS elements for a number of additional agent types in the following table:



Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts: bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, beaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises. suggestions, corrections	Keyboard entry

Examples of agent types and their PEAS descriptions.

Different types of Environment

- Fully observable vs. partially Fully observable
- Single agent Vs. Multi agent
- Deterministic Vs. Stochastic
- Episodic Vs. Sequential
- Static vs. Dynamic
- Discrete vs. Continuous
- Known vs. Unknown

The following table lists the properties of a number of familiar environments:

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

Examples of task environments and their characteristics.

End of session

SESSION 3

SOLVING PROBLEMS BY SEARCHING

Session Outcomes

- **Students will learn about problem solving agents.**
- **Students will learn precise definitions of various types of problems and their solutions .**

Reflex Agents:

- The simplest agents like Vacuum cleaner agent we discussed is an example of **Reflex Agents**.
- These agent base their actions on a direct mapping from states to actions.
- Such agents cannot operate well in environments for which this mapping would be too large to store and would take too long to learn.

Problem Solving Agents:

- Goal-based agents, on the other hand, consider future actions and the desirability of their outcomes.
- Some of these goal-based agent are called **problem-solving agents**.
- Problem-solving agents use atomic representations called states of the world which are considered as wholes, with no internal structure visible to the problem solving algorithms.

Planning Agents:

- Goal-based agents that use more advanced factored or structured representations are usually called **planning agent**.

Problem-Solving Agent:

Intelligent agents are supposed to maximize their performance measure in solving problems. Various steps in solving problem are:

•**Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.

•**Problem formulation** is the process of deciding what actions and states to consider, given a goal.

•The process of looking for a sequence of actions that reaches the goal is called **search**. A **search algorithm** takes a problem as input and returns a solution in the form of an action sequence.

•Once a solution is found, the actions it recommends can be carried out. This is called the **execution phase**.

We first describe the process of **problem formulation**, and then algorithms for the SEARCH function.

Well-defined problems and solutions : A problem can be defined formally by five components:

•The **initial state** that the agent starts in.

•A description of the possible actions available to the agent Given a particular state s, called **ACTIONS(s)**, returns the set of actions that can be executed in s.

•A description of what each action does; the formal name for this is the **transition Model**.

•The **goal test**, which determines whether a given state is a goal state.

- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.

  - The **step cost** of taking action a in state s to reach state s1 and denoted by C(s, a, s1).

EXAMPLE PROBLEMS :

- **Toy Problems**(intended to illustrate or exercise various problem-solving methods)
  - Vacuum Cleaner
  - 8-Puzzle Problem
  - 8-Queens Problem

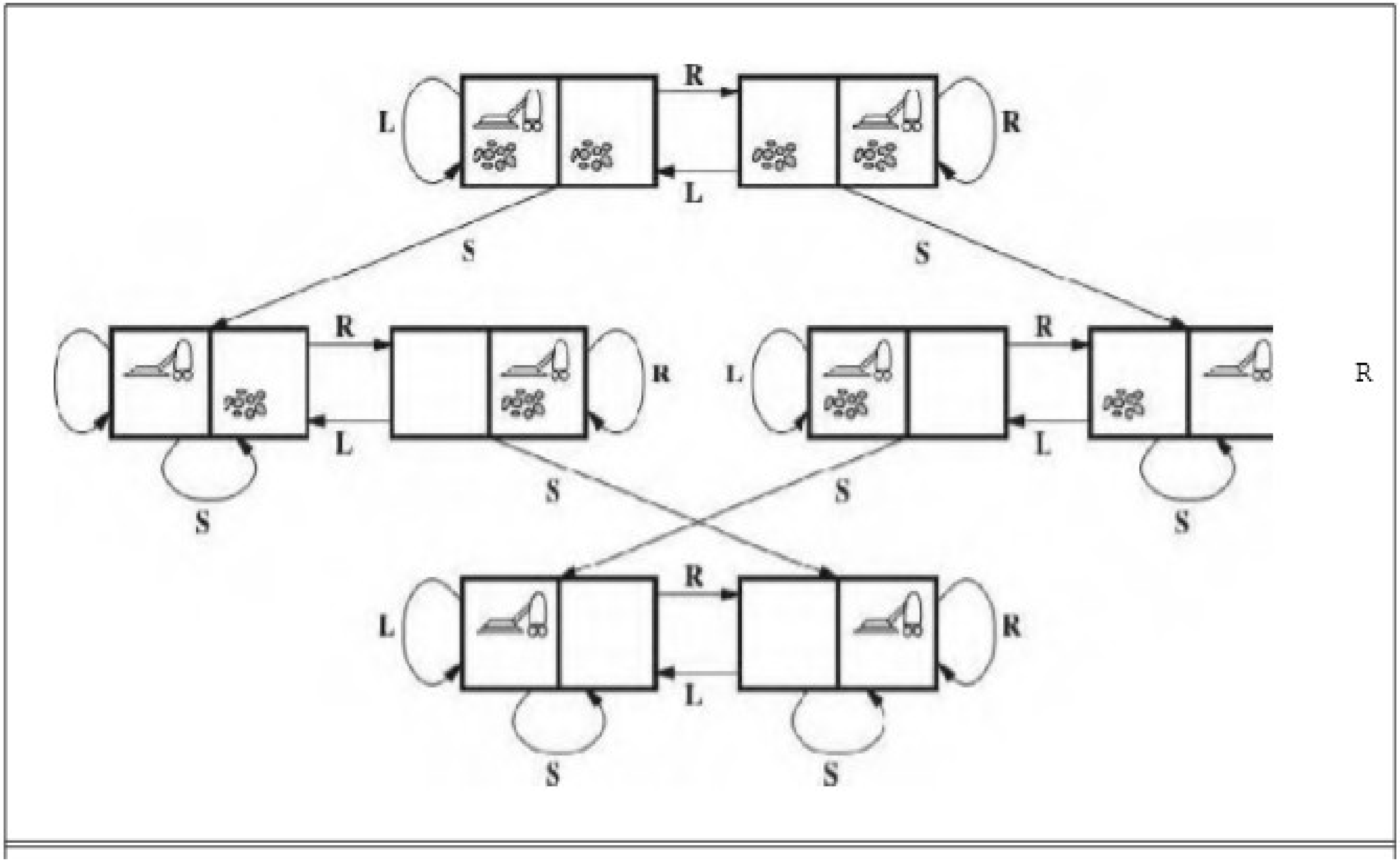
- **Real-World Problems(**whose solutions people actually care about)
  - Route Finding Problem
  - Travelling Salesperson Problem
  - Robot Navigation

1. Vacuum cleaner Example(Toy problem)

- **Initial state:** Any state can be designated as the initial state.
- **Actions:** *Left, Right, and Suck*
- **Transition model:** The complete state space is shown in the next slide.
- **Goal test:** This checks whether all the squares are clean.
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

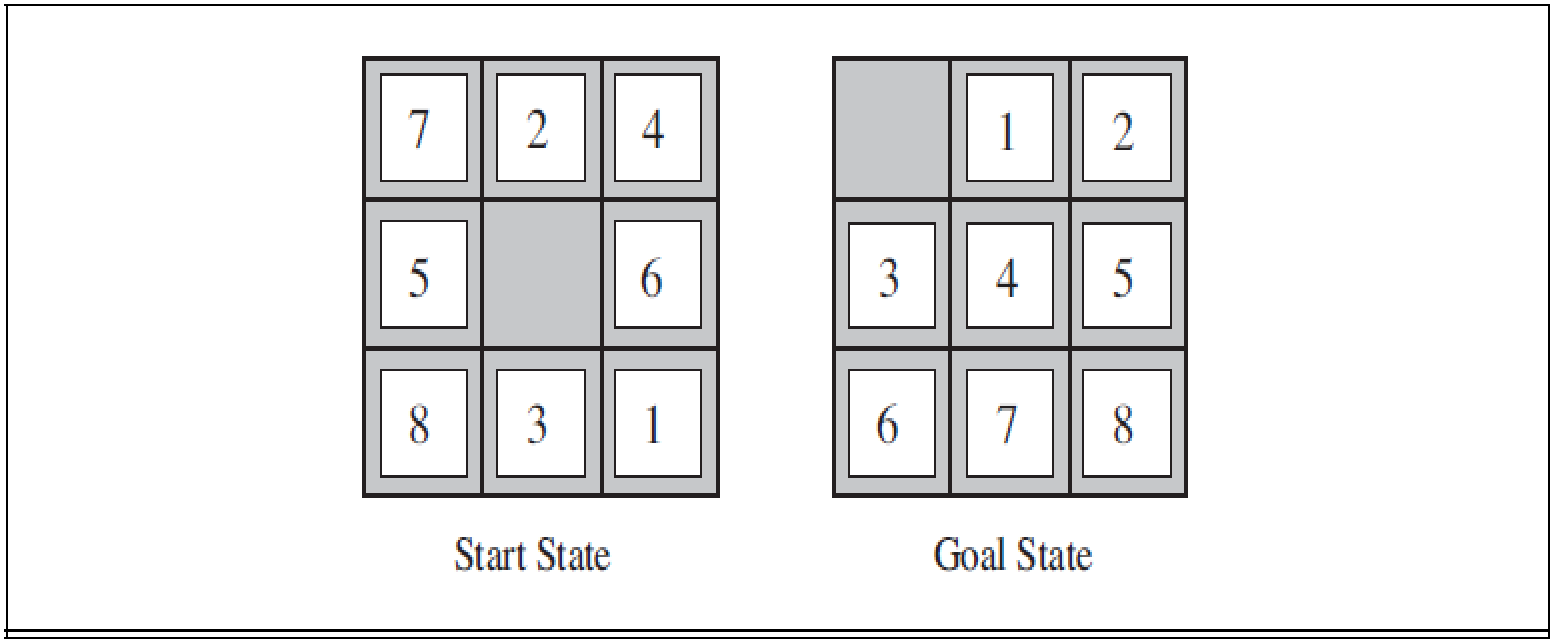
The state is determined by both the agent location and the dirt locations.

The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \times 2^n$ states.



The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, *S* = *Suck*

2. 8-puzzle problem (Toy problem)



A typical instance of the 8-puzzle.

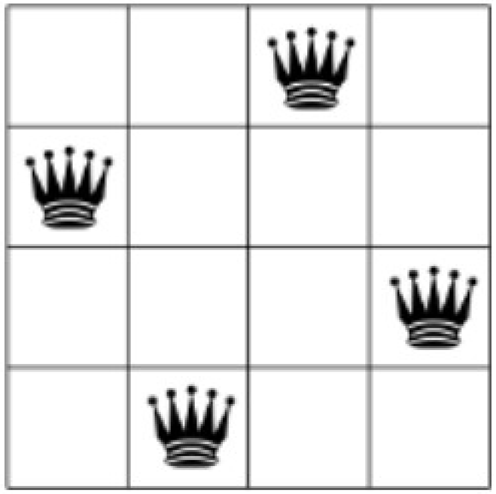
The 8-puzzle belongs to the family of **sliding-block puzzles,** which are often used as test problems for new search algorithms in AI.

- **Initial state:** Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
- **Actions:** The simplest formulation defines the actions as movements of the blank space. *Left, Right, Up, or Down. Different subsets of these are possible depending on where* the blank is.
- **Transition model:** Given a state and action, this returns the resulting state; for example, if we apply *Left to the start state in the above figure the resulting state has the 5 and the blank* switched.
- **Goal test:** This checks whether the state matches the goal configuration shown in Figure
- **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

- **States:** A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.

3. 4-queens problem (Toy Problem)

- The **4-Queens Problem** consists in placing four queens on a 4 x 4 chessboard so that no two queens can capture each other. That is, no two queens are allowed to be placed on the same row, the same column or the same diagonal (non-attacking).



- We can generalize it to nXn chessboards with n>=4.

- **Initial state:** No queens on the board.
- **Actions:** Add a queen to any empty square.
- **Transition model:** Returns the board with a queen added to the specified square.
- **Goal test:** 8 queens are on the board, none attacked**.**

- **States:** Any arrangement of 0 to 4 queens on the board is a state.

1. Route-Finding Problem (Real-world Problem)

Consider the airline travel problems that must be solved by a travel-planning Web site:

- **Initial state:** This is specified by the user’s query.
- **Actions:** Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- **Transition model:** The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time.
- **Goal test:** Are we at the final destination specified by the user?
- **Path cost:** This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on.

- **States:** Each state obviously includes a location (e.g., an airport) and the current time.

  Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these “historical” aspects.

Eg: Finding route in the following Romania map.

- Initial state: currently in Arad. In(Bucharest),Visit(Bucharest)
- Action: Drive between cities.
- Transition model: The state resulting from taking a flight will have the flight’s destination as the current location and the flight’s arrival time as the current time. Eg: in(Fagaras), Visited(Arad,Sibiu,Fagaras)
- Goal State: Be in Bucharest.
- Path cost: monetary cost, waiting time, flight time, customs and immigration procedures etc.
- States: various cities visited

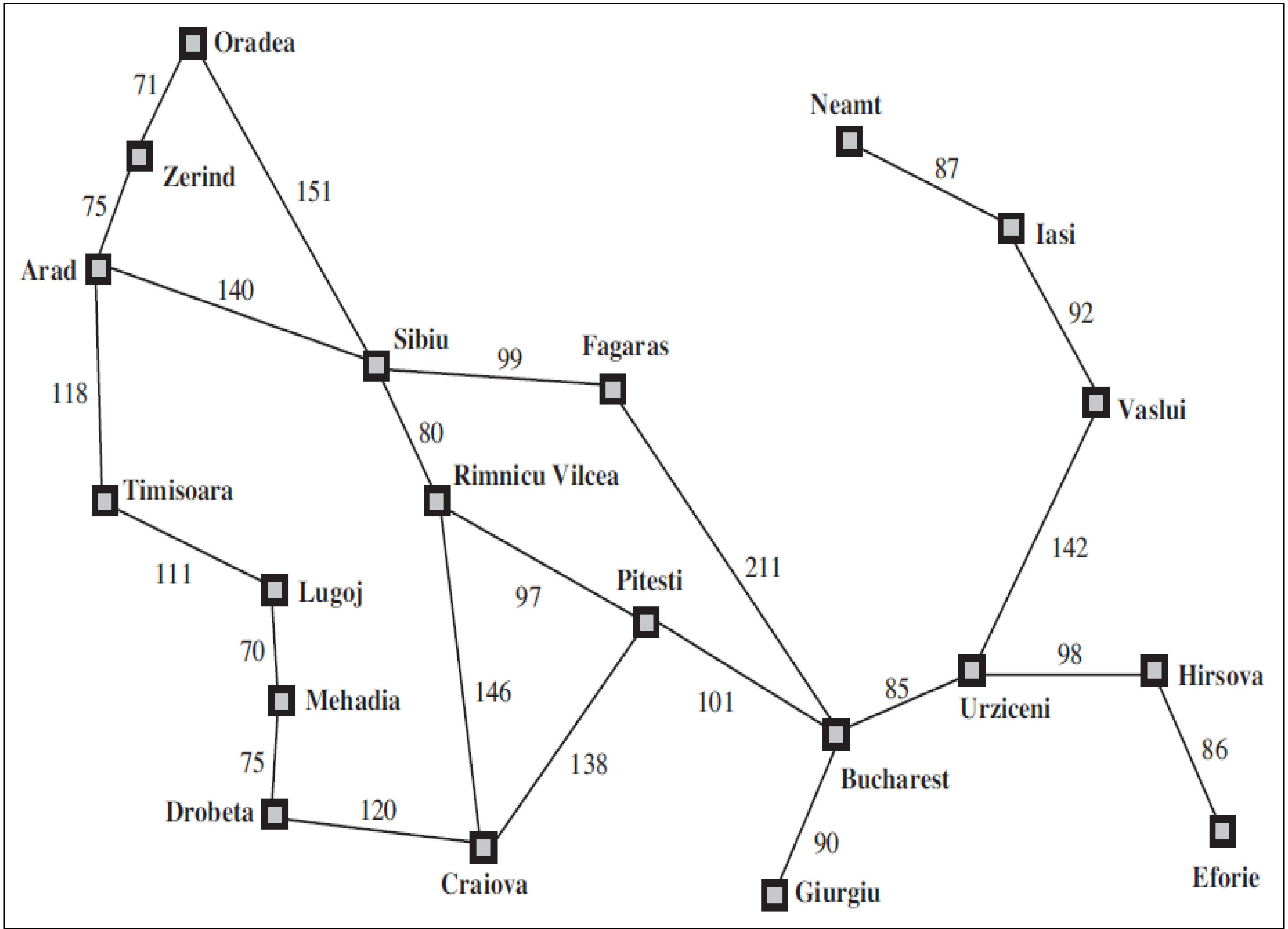


Figure 3.2 A simplified road map of part of Romania.

2. The Traveling Salesperson Problem (TSP) (Real-world Problem)

- TSP is a touring problem in which each city must be visited exactly once.

- The aim is to find the shortest tour.

- An enormous amount of effort has been expended to improve the capabilities of TSP algorithms.

- In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors.

3. Robot Navigation(Real-world problem)

•Robot navigation is a generalization of the route-finding problem described earlier.

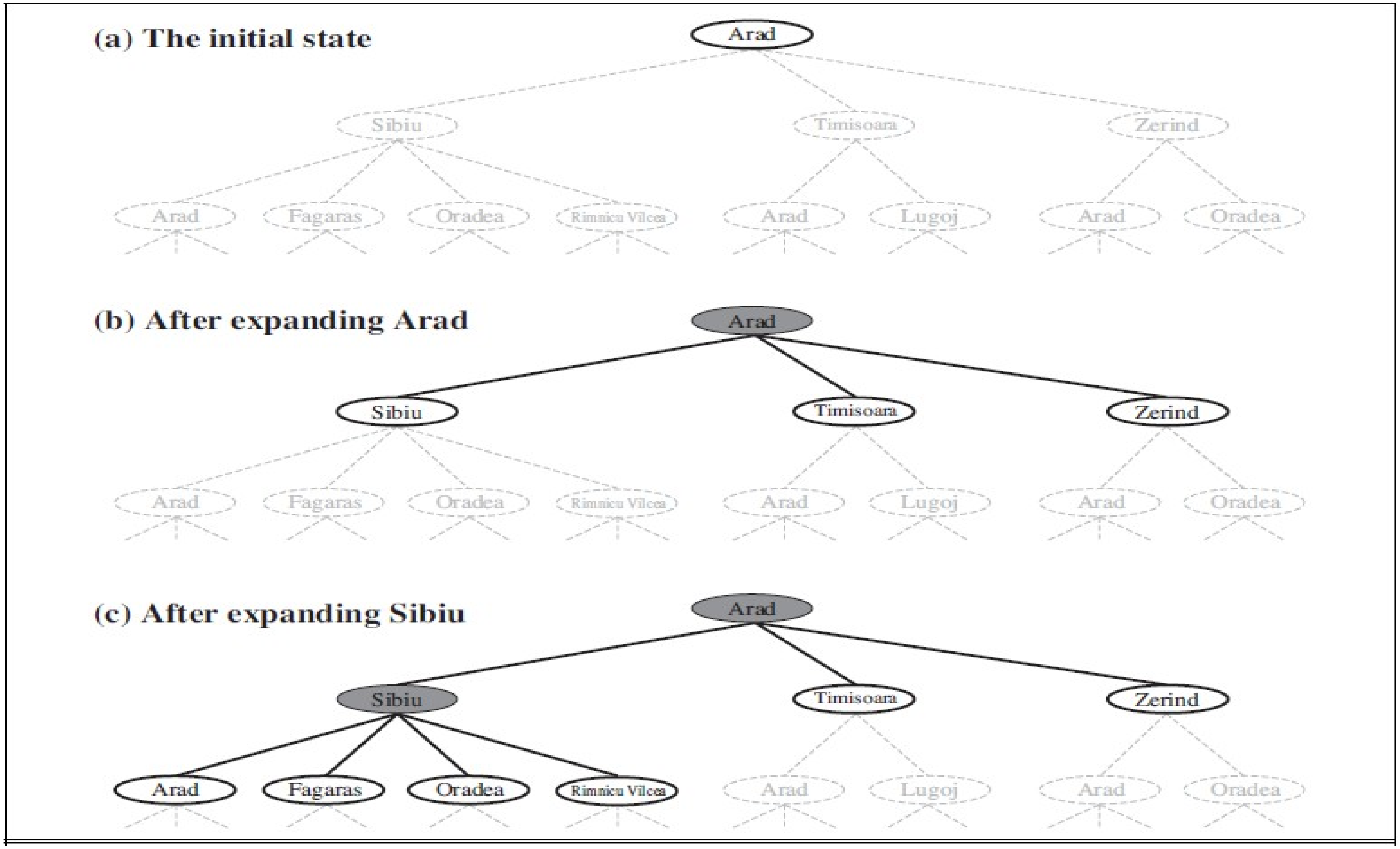
•Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states.

•For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional.

•Advanced techniques are required just to make the search space finite.

SEARCHING FOR SOLUTIONS

- Having formulated some problems, we now need to solve them.
- A solution is an action sequence, so search algorithms work by considering various action sequences. This can be implemented using search tree.
- **SEARCH TREE**: The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem.
- The following figure shows the few steps in constructing the search tree for finding a route from Arad to Bucharest in the Romania map.



Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

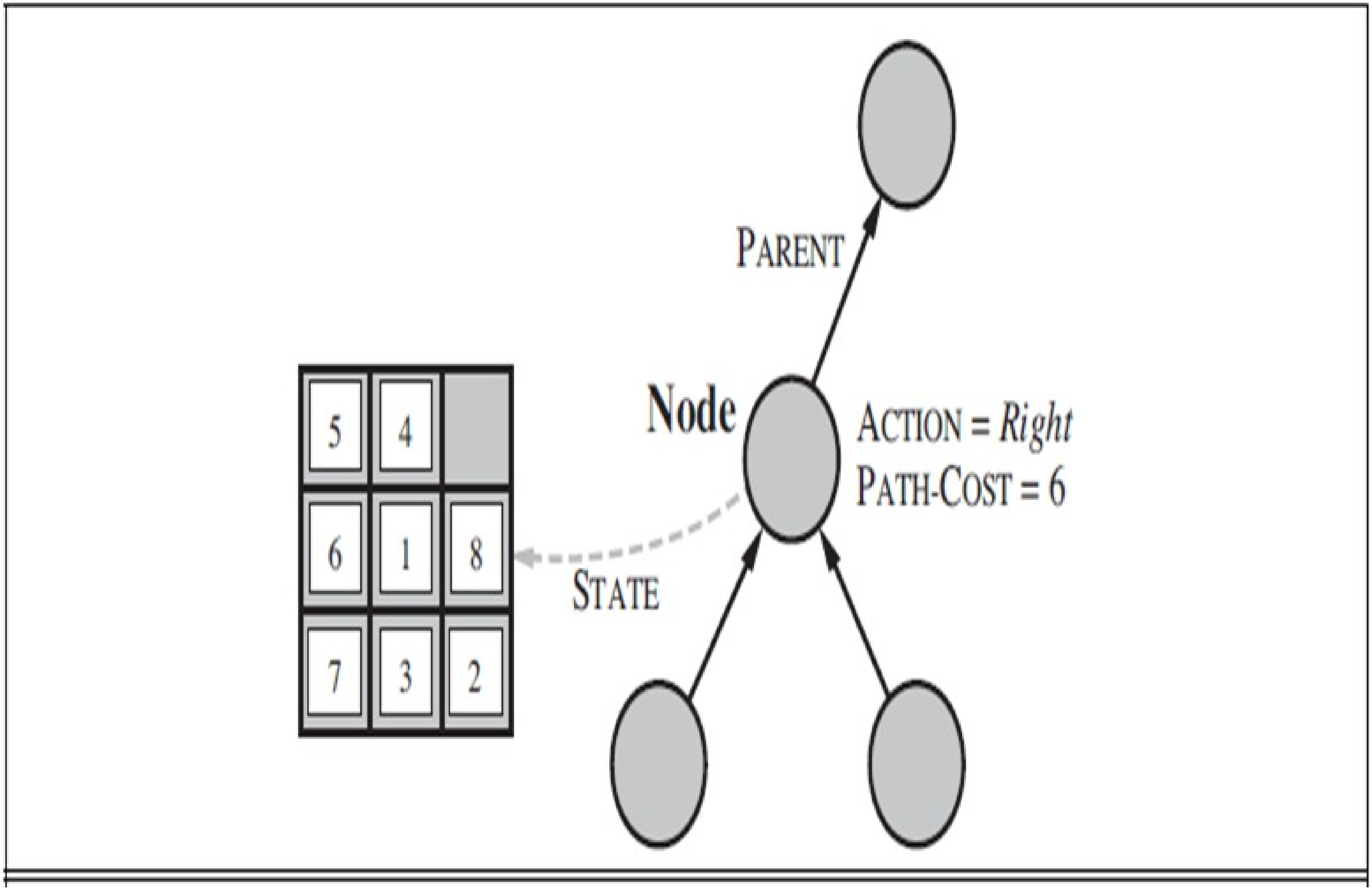
Terminology:

- **FRONTIER:** A set of all the leaf nodes available for expansion at any given point. Generally, it is considered as **open list.**
- **EXPLORED SET:** A set of nodes that are already explored. Generally the data structure is known as **Close List.**
- **GRAPH-SEARCH Algorithm:** An algorithm used to find the solution.
- **TREE-SEARCH Algorithm:** Uses a particular search strategy to choose which state to explore next.

Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four Components:

- **n.STATE:** The state in the state space to which the node corresponds.
- **n.PARENT:** The node in the search tree that generated this node.
- **n.Action:** The action that was applied to the parent to generate the node.
- **n.PATH-COST:** The cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.



Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields_ Arrows point from child to parent.

Measuring problem-solving performance

We can evaluate an algorithm's performance in four ways:

- **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
- **Optimality:** Does the strategy find the optimal solution
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

End of Session

SESSION 4

UNINFOMED SEARCH STRATEGIES

Session Outcomes

- Student will learn several uninformed search algorithms that can be used to solve search problems.

UNINFORMED SEARCH STRATEGIES

•In this session we discuss several search strategies that comes under the heading of uninformed search, also called blind search.

•i.e., the strategies have no additional information about states beyond what is provided in the problem definition.

•All they can do is generate successors and distinguish a goal state from a non-goal state.

•All search strategies are distinguished by the order in which nodes are expanded.

Various types of uninformed search algorithms are:

1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search
5. Bidirectional Search

1. Breadth-first Search

- Breadth-first search is the most common search strategy for traversing a tree or graph.

- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.

- Breadth-first search implemented using FIFO queue data structure for Frontier.

- Goal test is applied to each node when it is generated.

- The algorithm is shown in the following slide.

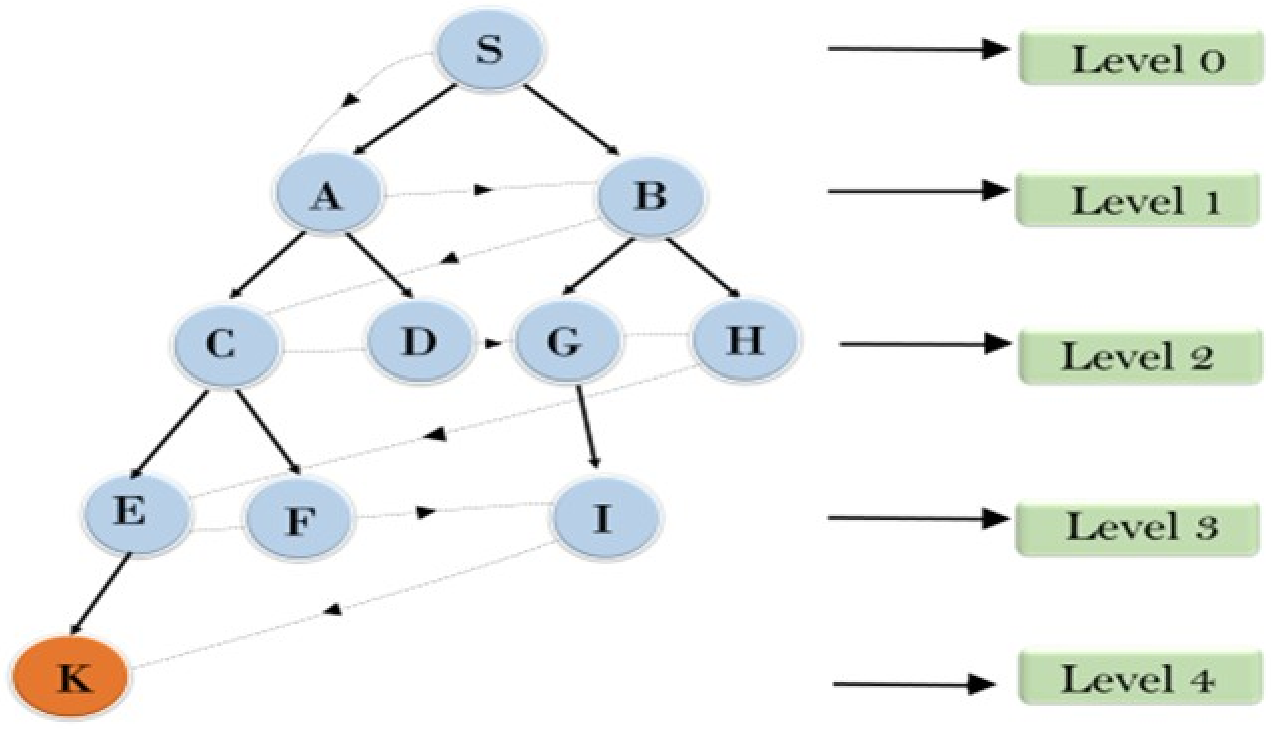
Breadth First Search (continued..)

Algorithm:

```
Inaction BREADTH-FIRST-SEARCH (problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST =0
  if problem.GOAL-TEST(node. STATE) then return SOLUTION(node)
  frontier _ a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier) /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem .ACTIONS(n ode. STATE) do
          child ← CHILD-NODE(problem, node , action)
          if child . STATE is not in explored or frontier then
              if problem GOAL- TEST(child.STATE) then return SOLUTION( child)
              frontier    INSERT( child, frontier)
```

Breadth First Search (continued..)

Example:



S---> A--->B---->C--->D---->G--->H--->E---->F----> I---->K

Breadth First Search (continued..)

Time Complexity:

- Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node.

- Suppose d= depth of shallowest solution and b = branching factor

$T(b) = b+b^2+b^3+.......+ b^d= O(b^d)$

Space Complexity:

- Every node generated remains in memory. There will be $0(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier.

- So the space complexity is $O(b^d)$, i.e., it is dominated by the size of the frontier.

Breadth First Search (continued..)

- **Completeness:**
  - BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality:

- BFS is optimal if path cost is a non-decreasing function of the depth of the node.

Breadth First Search (continued..)

Advantages:

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

Disadvantages:

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

Water Jug Problem

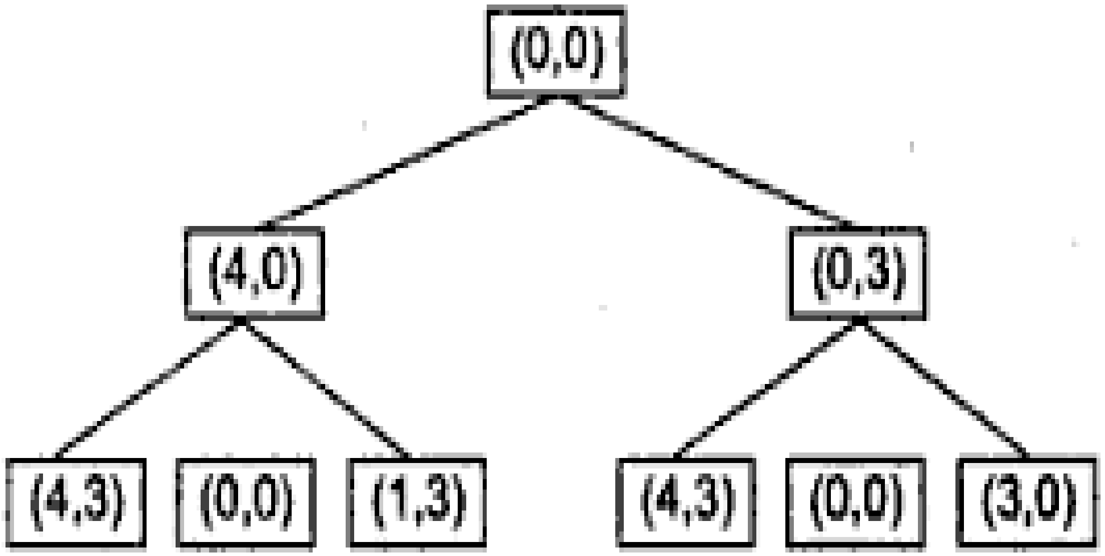
The state space for this problem can be described as the set of ordered pairs of integers (x,y) such that

x = 0, 1,2, 3 or 4

y = 0,1,2 or 3;

- x represents the number of gallons of water in the 4-gallon jug.
- y represents the quantity of water in 3-gallon jug.
- The start state is (0,0).
- The goal state is (2,y).

A Partial Search Tree for the Water jug Problem:



Production rules for Water Jug Problem

The operators to be used to solve the problem can be described as follows:

Sl No	Current state	Next State	Descritpion
1	(x, y) if x < 4	(4,y)	Fill the 4 gallon jug
2	(x,y) if y <3	(x,3)	Fill the 3 gallon jug
3	(x, y) if x > 0	(x-d, y)	Pour some water out of the 4 gallon jug
4	(x, y) if y > 0	(x, y-d)	Pour some water out of the 3-gallon jug
5	(x, y) if x>0	(0, y)	Empty the 4 gallon jug
6	(x, y) if y >0	(x,0)	Empty the 3 gallon jug on the ground

7	(x,y) if x+y >= 4 and y >0	(4, y-(4-x))	Pour water from the 3 –gallon jug into the 4 –gallon jug until the 4-gallon jug is full
8	(x, y) if x+y >= 3 and x>0	(x-(3-y), 3)	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if x+y <=4 and y>0	(x+y, 0)	Pour all the water from the 3-gallon jug into the 4-gallon jug
10	(x, y) if x+y <= 3 and x>0	(0, x+y)	Pour all the water from the 4-gallon jug into the 3-gallon jug

Solution to the water jug problem:

Gallons in the 4-gallon jug	Gallons in the 3-gallon jug	Rule applied
0	0	2
0	3	9
3	0	2
3	3	7
4	2	5
0	2	9
2	0	

Depth First Search

- Depth-first search is a Backtracking technique for finding all possible solutions using recursion algorithm for traversing a tree or graph data structure.

- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.

- DFS uses a LIFO queue for its implementation.

Depth First Search(continued….)

Algorithm:

1. If the initial state is a goal state, quit and return success.

2. Otherwise, do the following until success or failure is signaled:

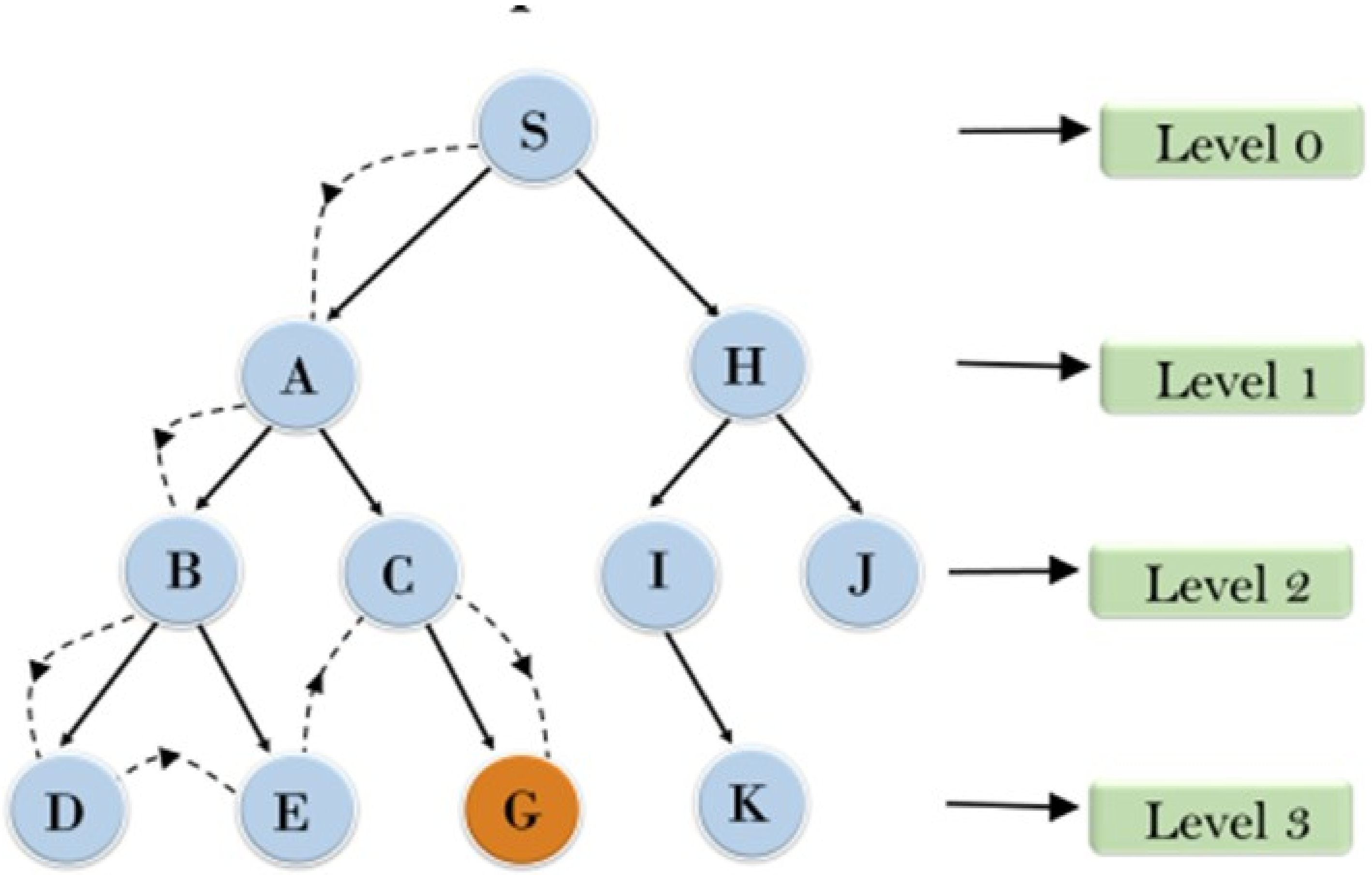
    a. Generate a successor, E, of initial state. If there are no more successors, signal failure.

    b. Call Depth-First Search, with E as the initial state

    c. If success is returned, signal success. Otherwise continue in this loop.

Depth First Search(continued….)

Example:



S---> A--->B---->D--->E---->C--->G

Depth First Search(continued….)

Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$$T(n)= 1+b^2+ b^3 +.........+ b^m=\mathbf{O(b^m)}$$

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

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.

Optimality: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Depth First Search(continued….)

Advantages:

•DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.

•It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantages:

•There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.

•DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Depth-Limited Search

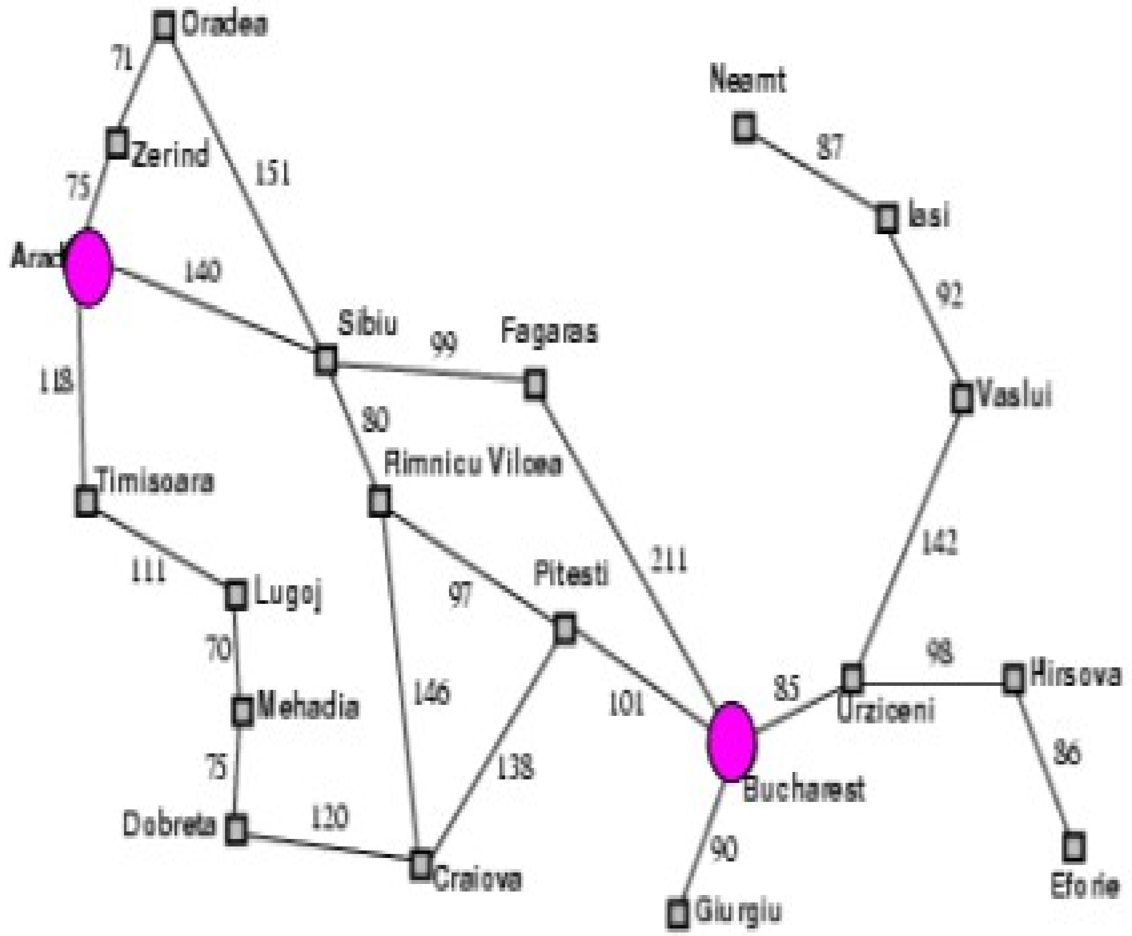
- A depth-limited search algorithm is similar to depth-first search with a predetermined limit.

- Depth-limited search can solve the drawback of the infinite path in the Depth-first search.

- In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

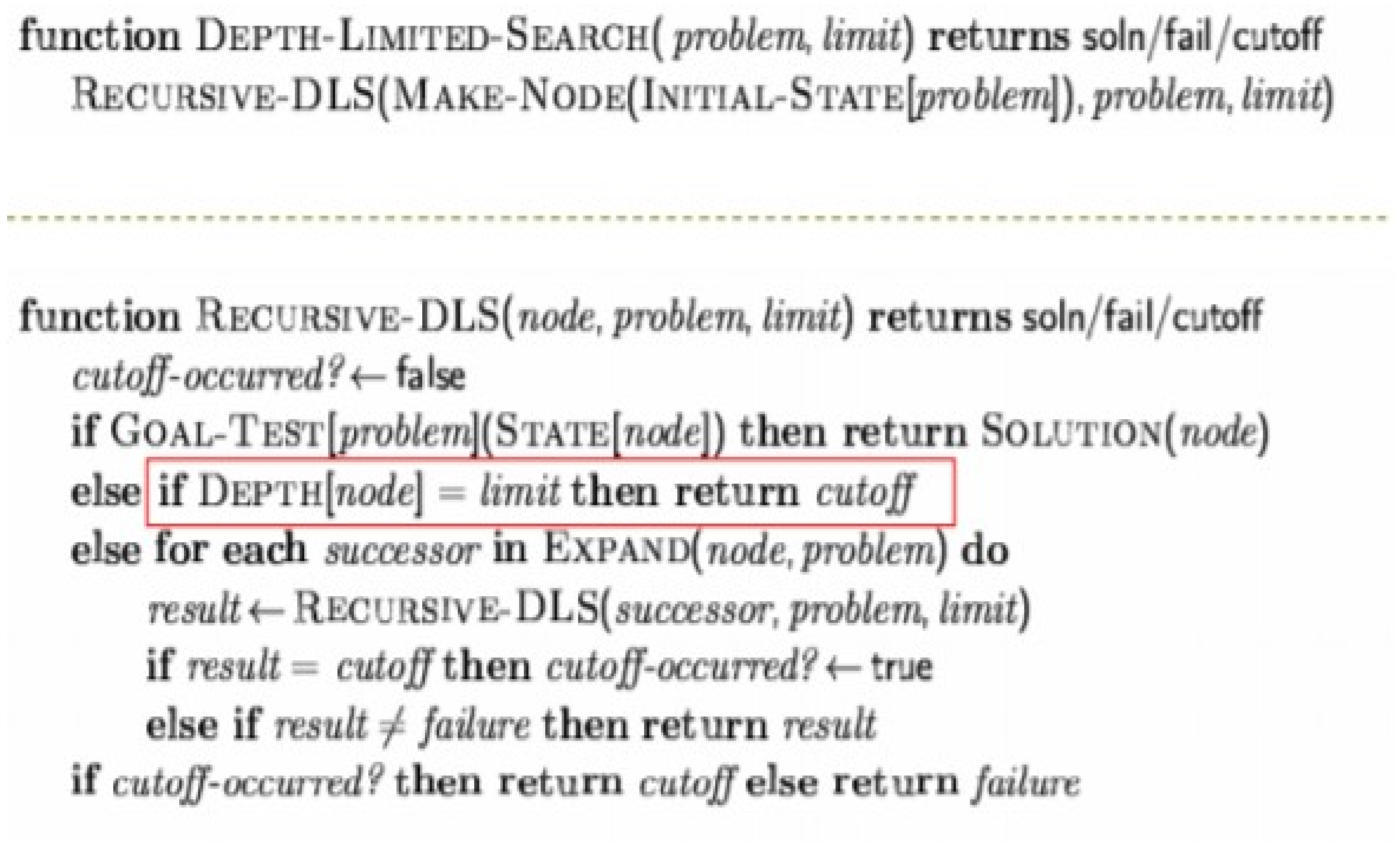
• Some times Depth limit can be set based on knowledge of the problem

What value will you set for ℓ ?



Depth-Limited Search(continued….)

Algorithm:

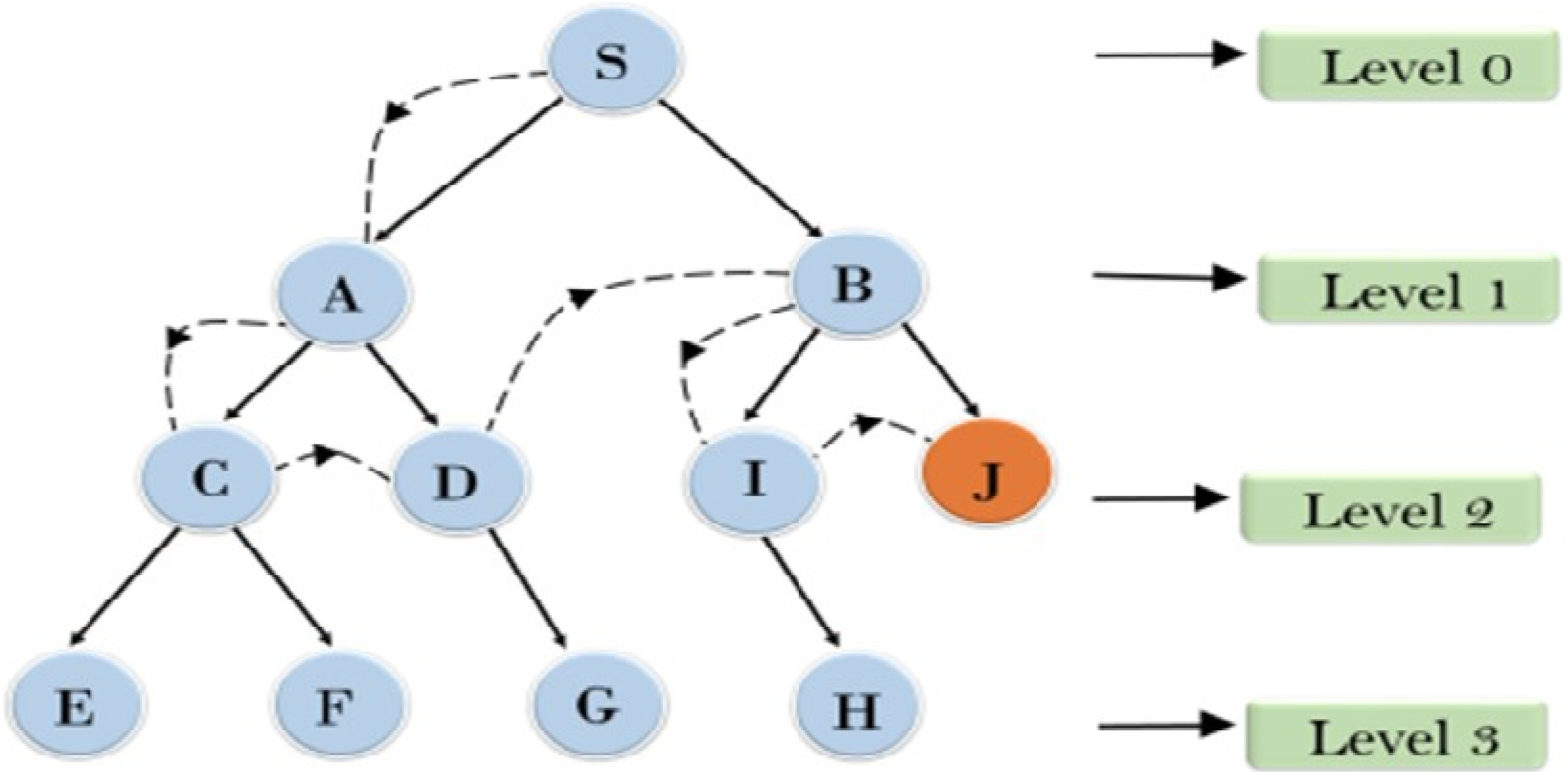


```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
```

```
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Depth Limited Search(continued…)

Example:



Depth Limited Search(continued…)

•**Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.

•**Time Complexity:** Time complexity of DLS algorithm is **$O(b^{\ell})$**. (where ℓ is the depth limit of the search).

•**Space Complexity:** Space complexity of DLS algorithm is **$O(b \times \ell)$.**

•**Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal, if $\ell > d$.

Depth Limited Search(continued…)

Advantages:

- Depth-limited search is Memory efficient.

Disadvantages:

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

SESSION 5

Uninformed Search Strategies(Continued...)

&

Informed Search Strategies
(Heuristic Search Techniques)

Session Outcomes

- Student will learn several general-purpose search algorithms that can be used to solve these problems.

Iterative deepening depth-first Search (IDDFS)

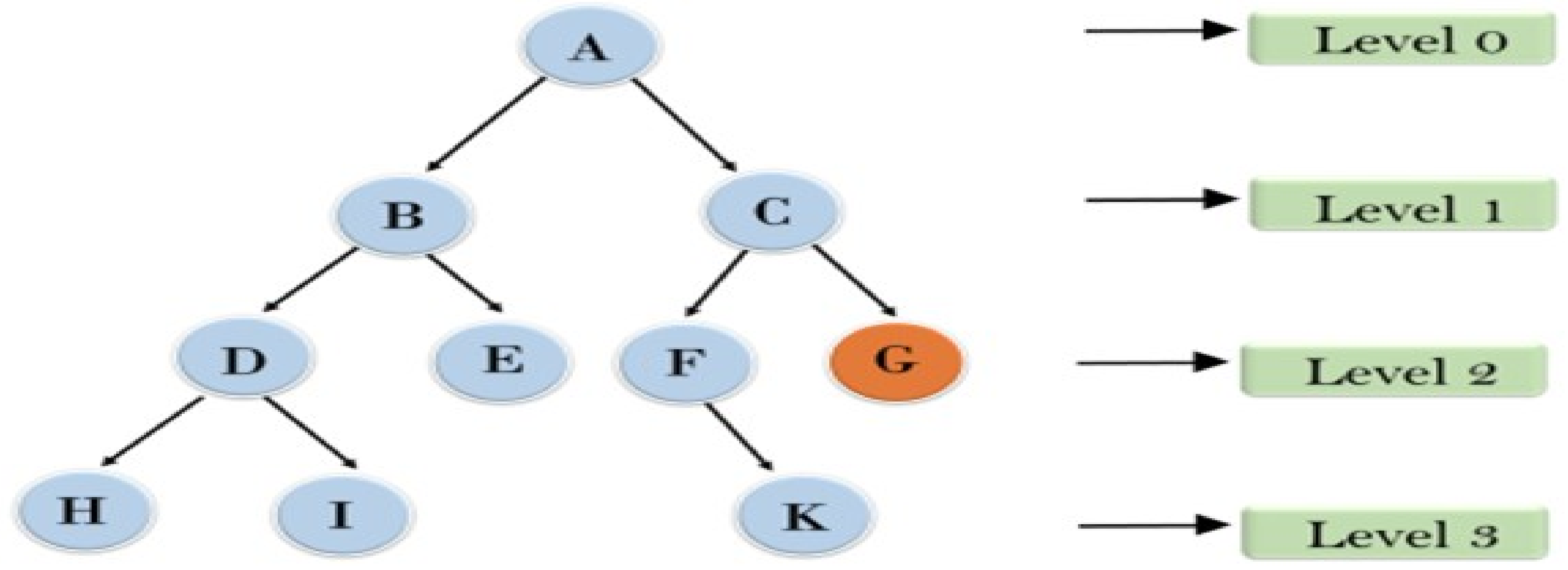
- Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit.

- This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

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

Iterative deepening depth-first Search(continued….)

Example:



- 1'st Iteration-----> A
- 2'nd Iteration----> A, B, C
- 3'rd Iteration------>A, B, D, E, C, F, G
- In thrid iteration the algorithm will find the gold node i.e., G.

Iterative deepening depth-first Search(continued….)

•**Completeness:** This algorithm is complete is if the branching factor is finite.

•**Time Complexity:** Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **$O(b^d)$**.

•**Space Complexity:** The space complexity of IDDFS will be **O(bd)**.

•**Optimality:** IDDFS algorithm is optimal if path cost is a non-decreasing function of the depth of the node.

Iterative deepening depth-first Search(continued….)

Advantages:

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

•Like DFS, Its memory requirement is modest.

Disadvantages:

•Some times, Iterative deepening search may seem wasteful because states are generated multiple times. It turns out this as too costly.

Bidirectional Search

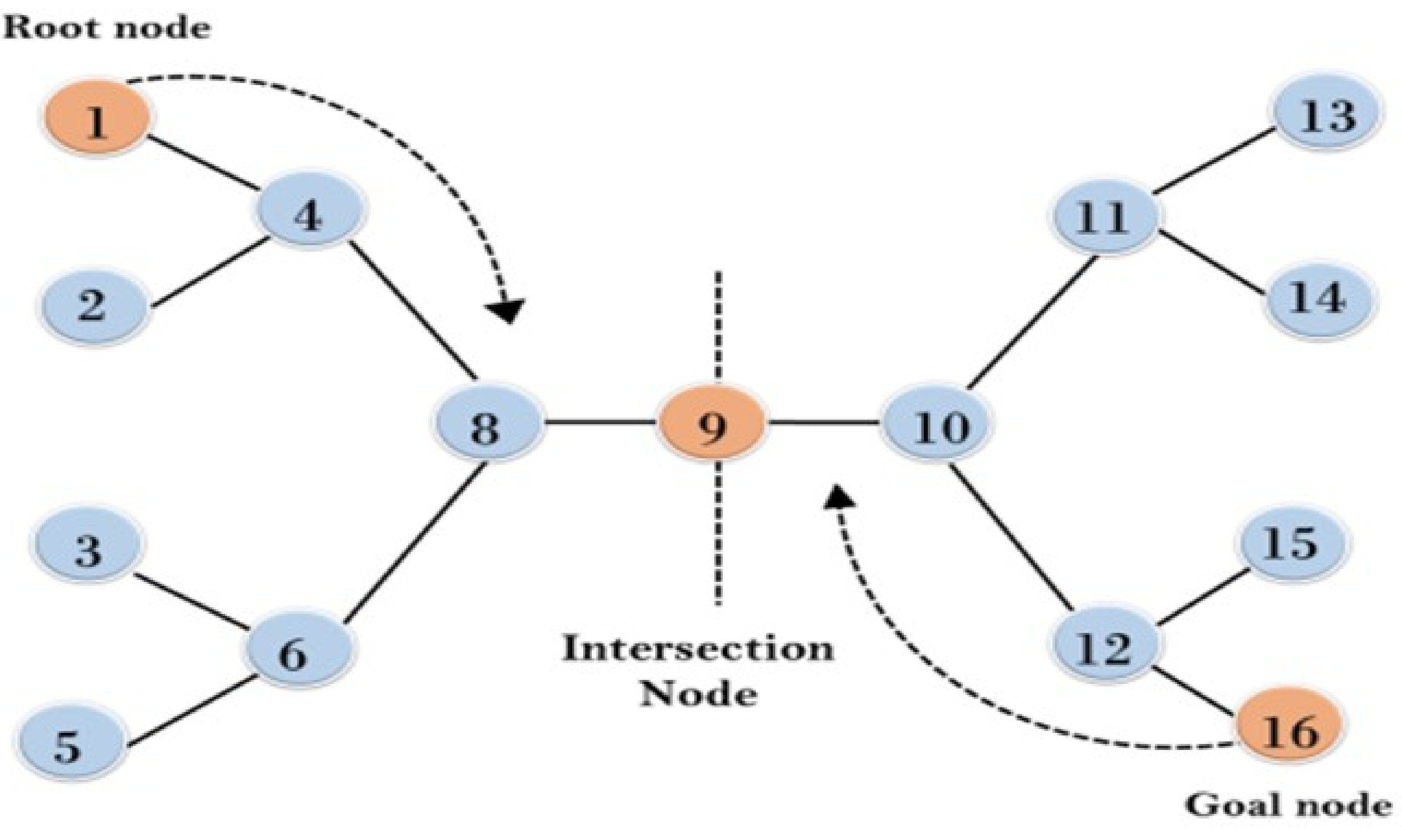
- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.

- Bidirectional search replaces one single search graph with two small sub-graphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.

- Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Bidirectional Search(Continued……)

Example:



Bidirectional Search(Continued……)

- **Completeness:** Bidirectional Search is complete if we use **BFS** in both searches.

- Time Complexity: **Time complexity of bidirectional search** using BFS is $O(b^{d/2})$.

- **Space Complexity:** Space complexity of bidirectional search is $O(b^{d/2})$.

- **Optimal:** Bidirectional search is Optimal

Bidirectional Search(Continued……)

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- In bidirectional search, one should know the goal state in advance.

INFORMED SEARCH STRATEGIES
(Heuristics Search Techniques)

•Informed search strategy is one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than can an uninformed strategy.

•We learn the following two strategies as part of Informed Search Strategies. We discuss GBFS in this session and A* algorithm in the following session:

- Greedy Best First Search
- A* search

Heuristic Search

•All of the search methods in the preceding section are uninformed in that they did not take into account the cost incurred to reach the goal.

•They do not use any information about where they are trying to get to unless they happen to stumble on a goal.

•One form of heuristic information about which nodes seem the most promising is a heuristic function h(n), which takes a node n and returns a non-negative real number that is an estimate of the path cost from node n to a goal node.

•The function h(n) is an underestimate if h(n) is less than or equal to the actual cost of a lowest-cost path from node n to a goal.

Greedy best-first search

•Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$.

•The algorithm is called "greedy“ because at each step it tries to get as close to the goal as it can.

•Greedy Best First search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal.

Greedy Best First Search(Continued….)

Algorithm:

1. Start with OPEN containing just the initial state

2. Until a goal is found or there are no nodes left on OPEN do:
   a. Pick the best node on OPEN
   b. Generate its successors
   c. For each successor do:
      i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent.
      ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

Greedy Best First Search(Continued….)

Explanation:

- It proceeds in steps, expanding one node at each step, until it generates a node that corresponds to a goal state.
- At each step, it picks the most promising of the nodes that have so far been generated but not expanded.
- It generates the successors of the chosen node, applies the heuristic function to them, and adds them to the list of open nodes, after checking to see if any of them have been generated before.
- By doing this check, we can guarantee that each node only appears once in the graph, although many nodes may point to it as a successor.

Greedy Best First Search(Continued….)

- Heuristic function h(n)= estimated as cost of the cheapest path from the state at node n to a goal state, i.e., h(n)=straight line distance

- h_{sld} cannot be computed from the problem description itself

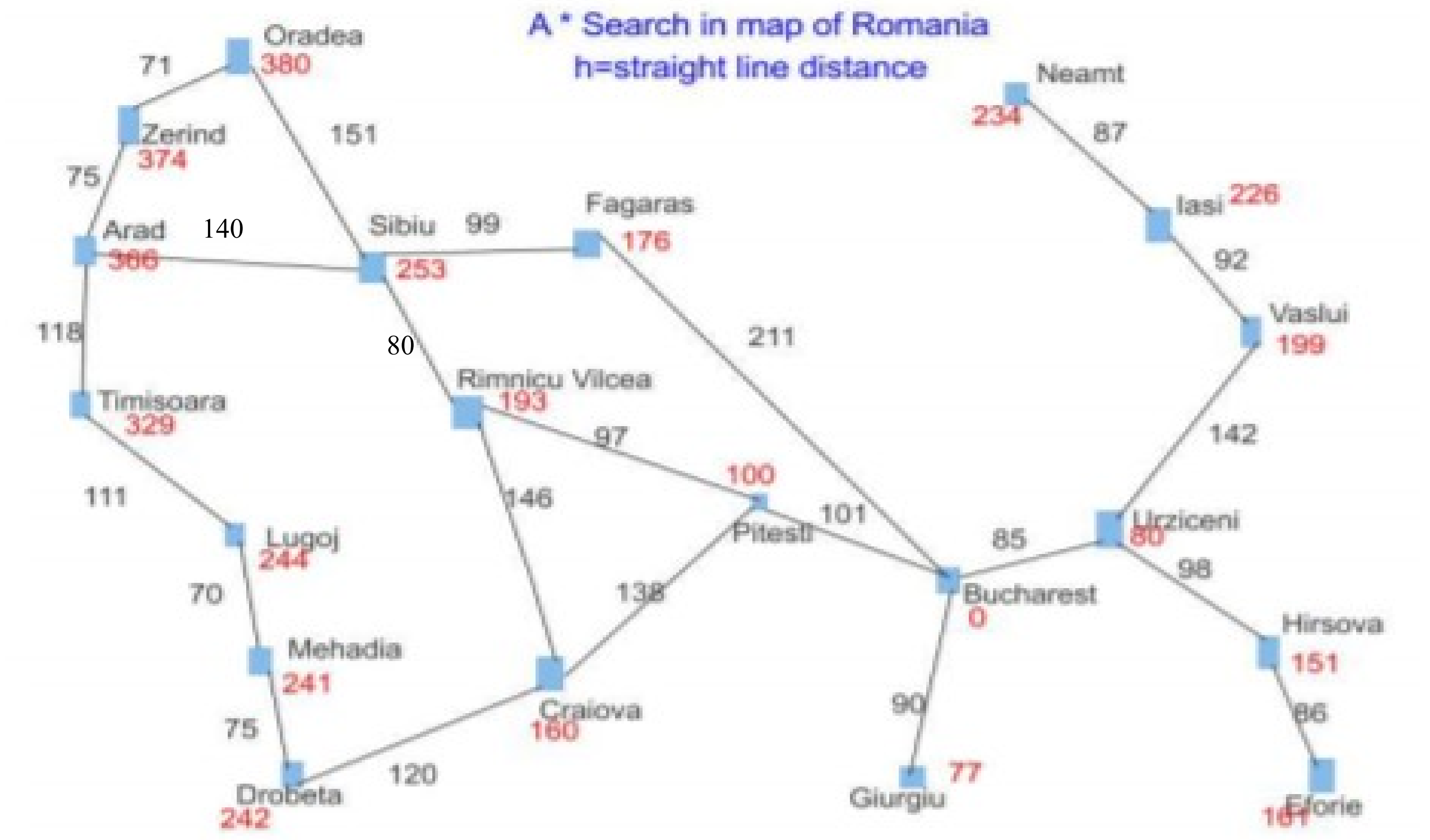
- Its search cost is minimal(Search cost is very low but not lowest). However, it is not optimal.

- The worst-case time and space complexity for the tree version is $O(d^{m})$, where m is the maximum depth of the search space.

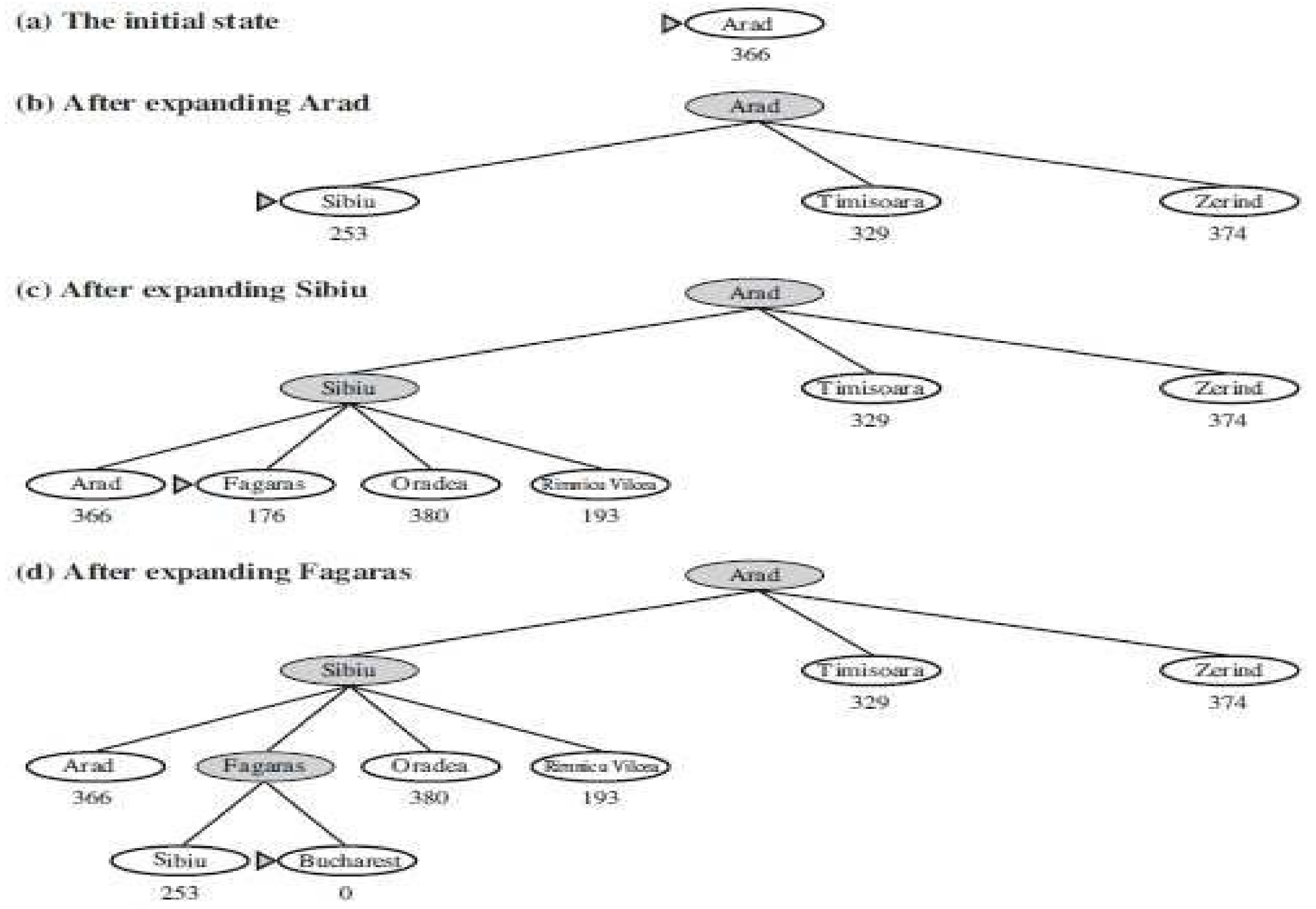
Greedy Best First Search(Continued….)

Example:

- Consider the following Romania map:
- Goal is to find a path from Arad to Bucharest.



Greedy Best First Search(Continued….)



Greedy Best First Search(Continued….)

Advantages:

GBFS mostly finds a solution without ever expanding a node that is not on the solution path; hence, its search cost is minimal.

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly.

Disadvantages:

Greedy best-first tree search is incomplete even in a finite state space. For Eg., Consider the problem of getting from Iasi to Fagaras. The heuristic sug- gests that Neamt be expanded first because it is closest to Fagaras, but it is a dead end.

End of session

Session 6

A* search
(Minimizing the total estimated solution cost)

Session Outcomes

- Student will learn the general-purpose informed (heuristic) search algorithm, A* that can be used to solve search problems.

A* search (Best-first search)

•For Minimizing the total estimated solution cost

•It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal: f(n) = g(n) + h(n)
.

•Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f (n) = estimated cost of the cheapest solution through n .

• If we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of f(n)=g(n) + h(n).

•The algorithm is identical to GBFS except that A* uses g + h instead of h.

A* search (Continued…)

Algorithm:

1. Start with OPEN containing only initial node. Set that node's g(n) value to 0, its h(n) value to whatever it is, and its f(n) value to h+0 or h(n). Set CLOSED to empty list.

2. Until a goal node is found, repeat the following procedure:
   I. If there are no nodes on OPEN, report failure.
   II. Otherwise pick the node on OPEN with the lowest f(n) value. Call it BESTNODE.
   III. Remove it from OPEN. Place it in CLOSED.
   IV. See if the BESTNODE is a goal state. If so exit and report a solution.
   V. Otherwise, generate the successors of BESTNODE but do not set the BESTNODE to point to them yet.

A* search (Continued…)

3. For each of the SUCCESSOR, do the following:
   a. Set SUCCESSOR to point back to BESTNODE.

      (These backwards links will make it possible to recover the path once a solution is found.)

   b. Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR

   c. See if SUCCESSOR is the same as any node on OPEN. If so call the node OLD. See whether it is cheaper to get to OLD via its current parent or SUCCESSOR via BESTNODE by comparing their g values. If SUCCESSOR is cheaper, then reset OLD's parent to point to BESTNODE, record the new cheaper path in g(OLD) and update f'(OLD).

A* search (Continued…)

d. If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call the node on CLOSED OLD and add OLD to the list of BESTNODE's successors.

Check to see if the new path is better. If so, set the parent link and g and f' values appropriately.

We must propagate the improvements to OLD's successors. OLD points to successors. Each successor, in turn, points to its successors, and so forth until each branch terminates with a node that either is still on OPEN or has no successors.

So to propagate the new cost downward, do a depth-first traversal of the tree starting at OLD, changing each node's g value (and thus also its f' value), terminating each branch when you reach either a node with no successor or a node to which an equivalent or better path has already been found.

e. If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN and add it to the list of BESTNODE's successors.

Compute

$$f'(\text{SUCCESSOR}) = g(\text{SUCCESSOR}) + h'(\text{SUCCESSOR})$$

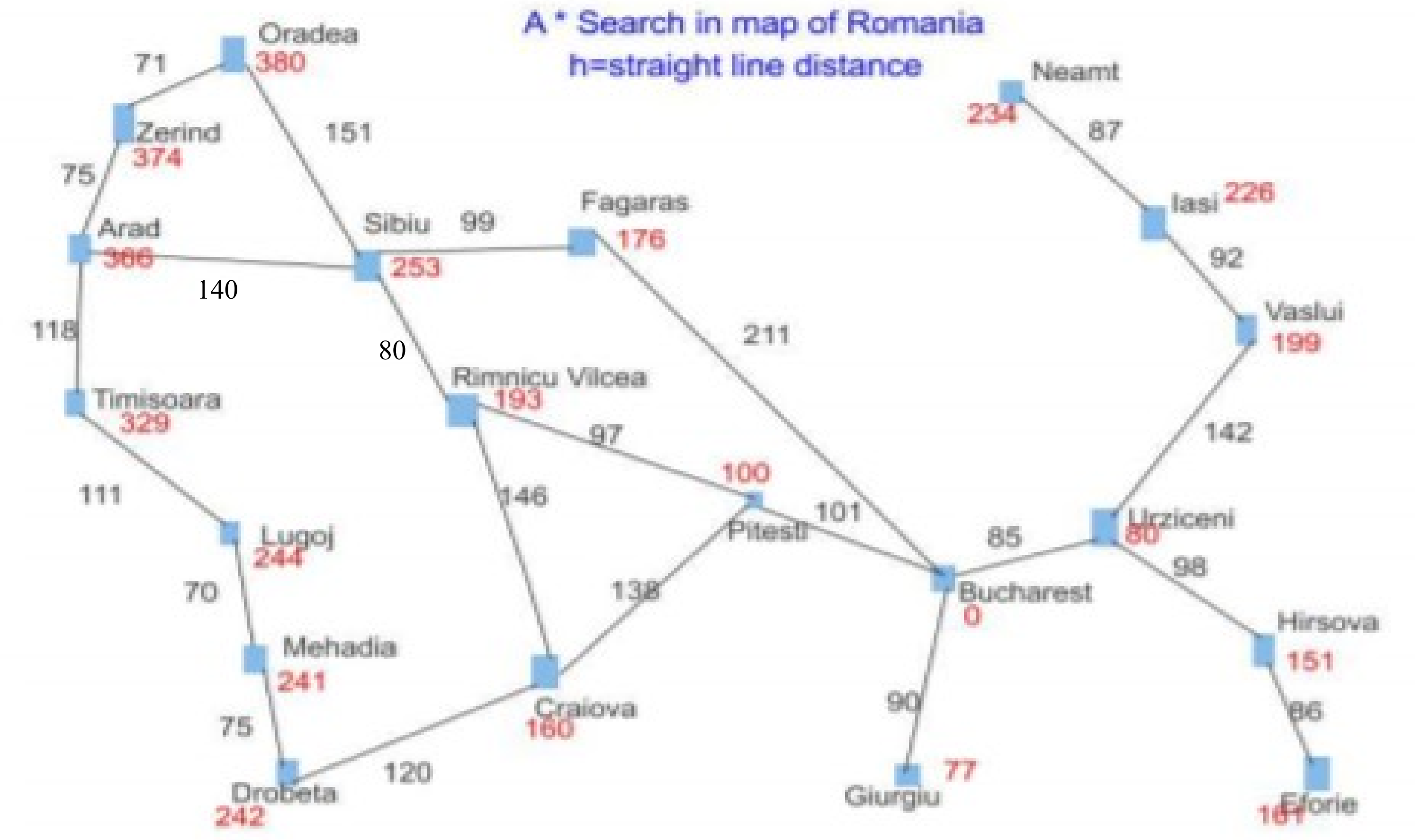
A* search (Continued…)

Example:

A∗ search applied to map of Romania:

- This figure in the next slide represents the intial map of Romania.
- The values representing in red colour are heuristic values(i.e h(n)).
- The values representing in silver colour are path cost values i.e g(n).
- The values representing in blue colour are f(n) values i.e., f(n) = g(n) + h(n).

A* search (Continued…)



Initial map of Romania

A* search (Continued…)

After expanding Arad :

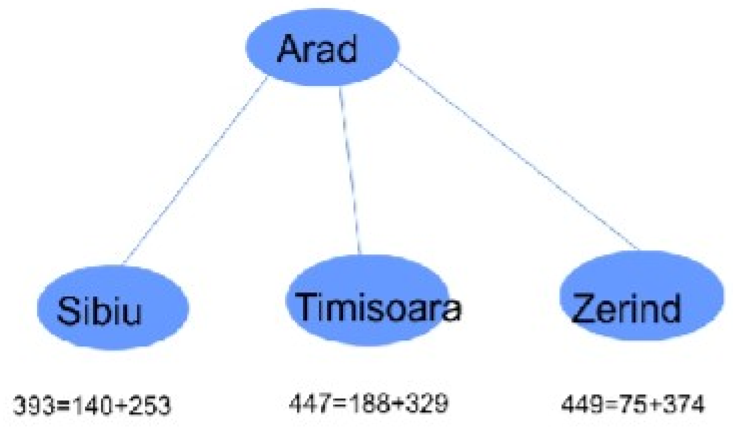
- We have three nodes i.e Zerind, Sibiu and Timisoara.
- As we know f(n)=g(n)+h(n).
- f(Sibiu)=f(n)=140+253=393 (g(n)=140 and h(n)=253).

- f(Zerind)=f(n)=75+374=449 (g(n)=75 and h(n)=374).

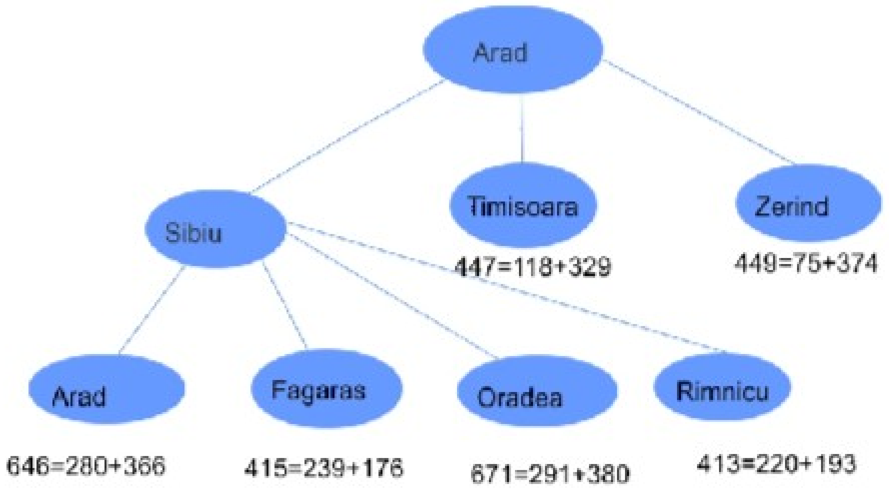
- f(Timisoara)=f(n)=118+329=447 (g(n)=118 and h(n)=329).

- From these nodes we have to choose the least f(n) value, so f(Sibiu) is least among these nodes.

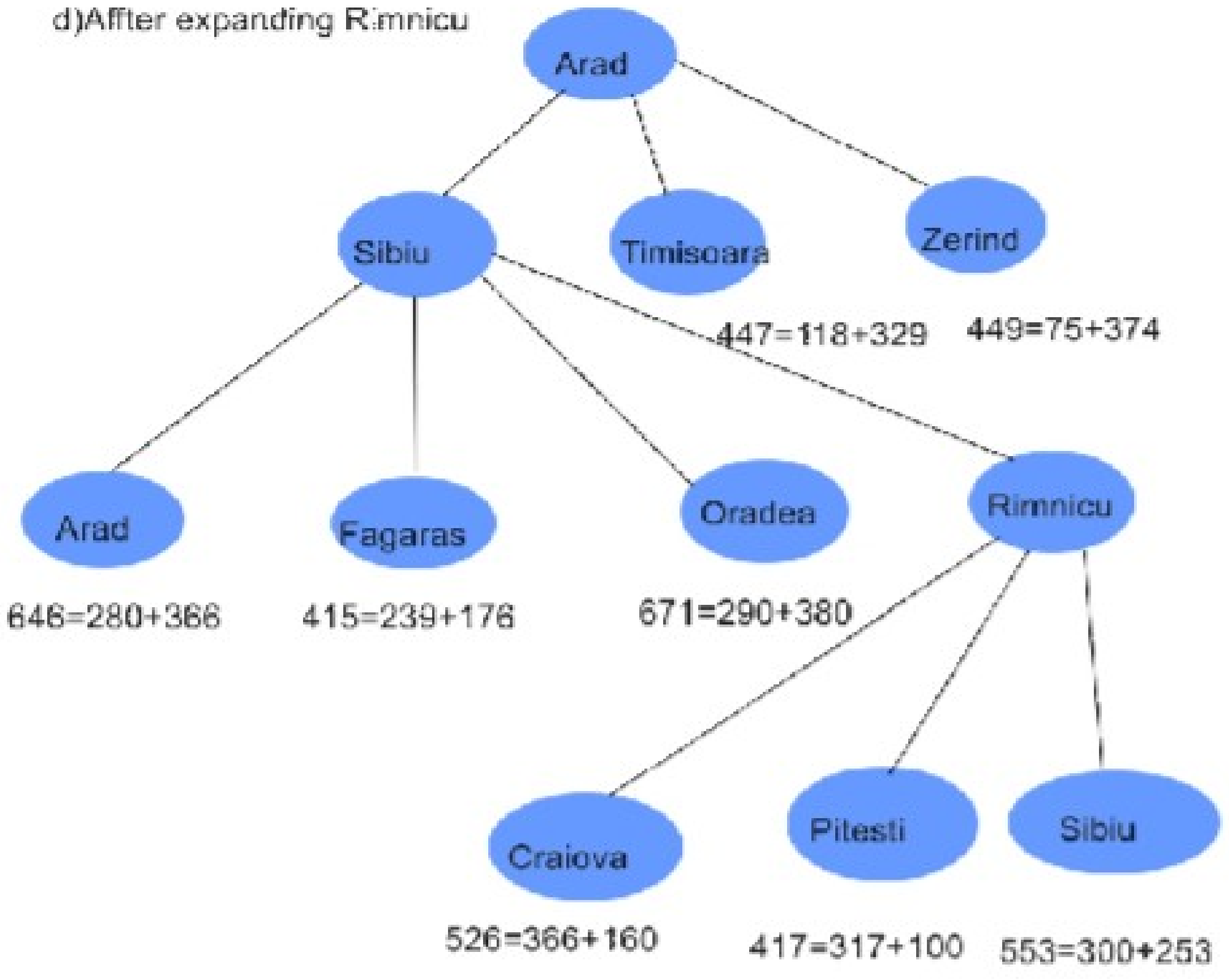
a) Aftter expanding Arad



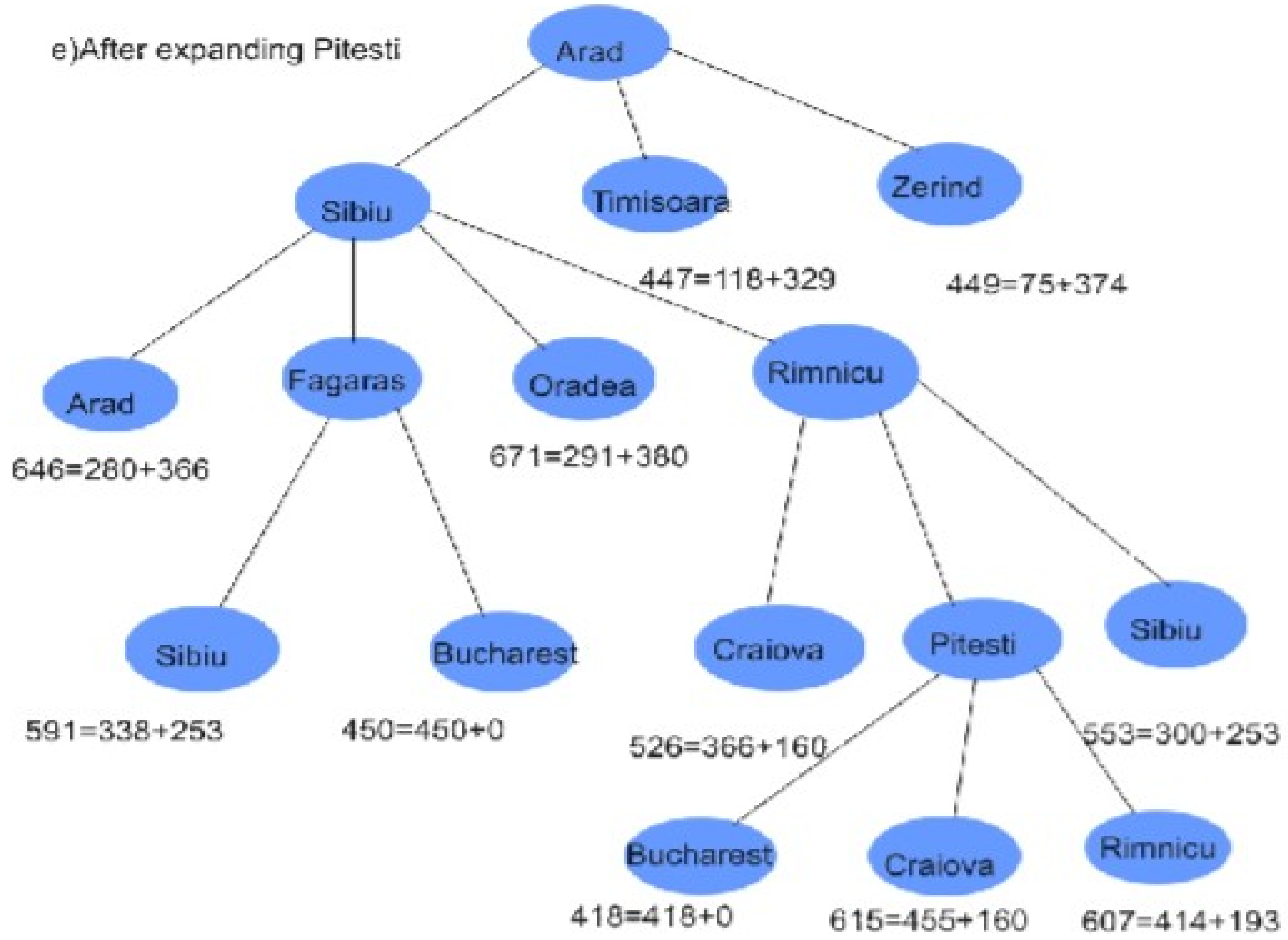
b)After expanding Sibiu



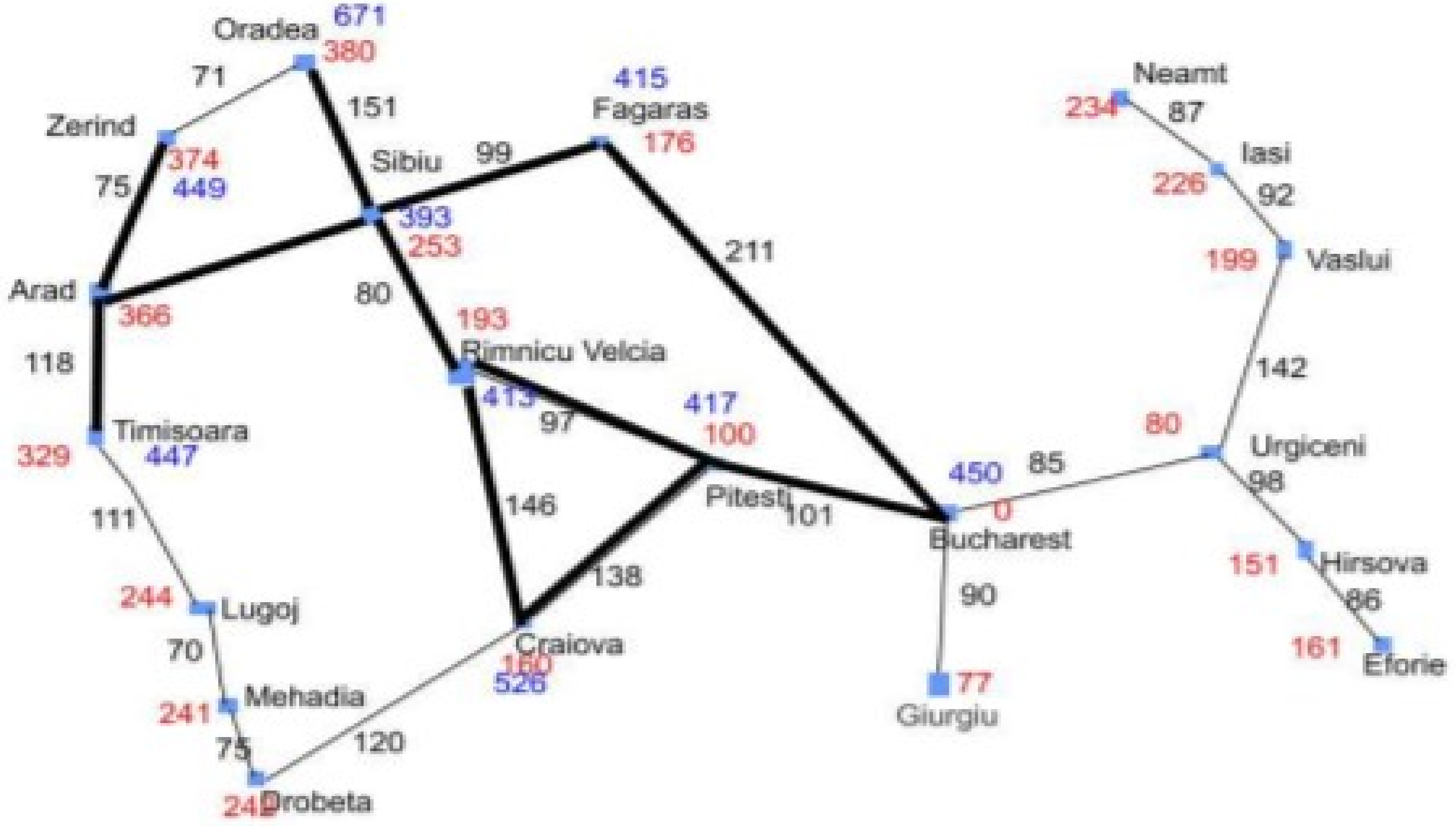
d)Affter expanding Rimnicu



e)After expanding Pitesti

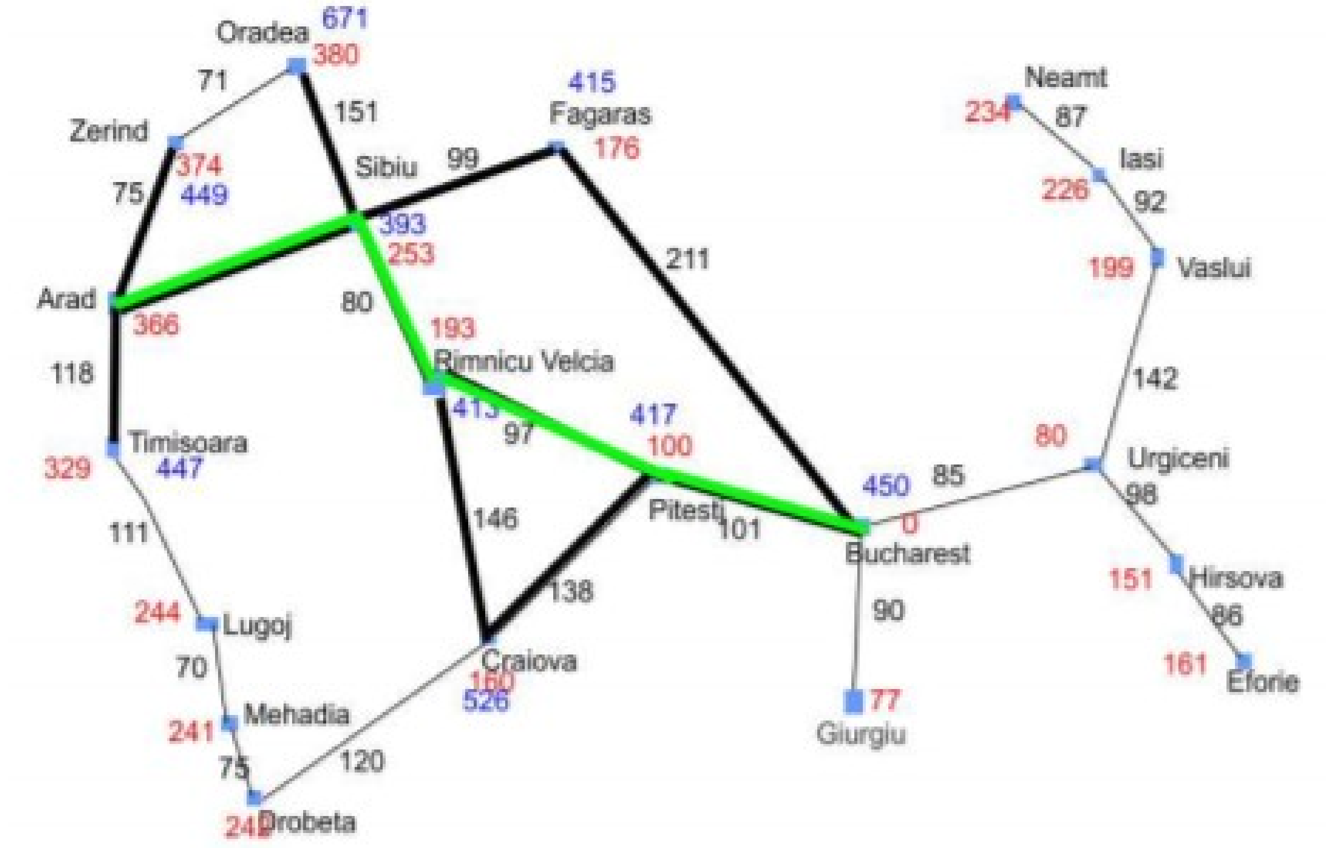


A* search (Continued…)



Nodes visited/Expanded

A* search (Continued…)



Shortest path from Arad to Bucharest

A* search (Continued…)

Advantages:

- That A* search is complete, optimal, and optimally efficient among all such algorithms.

Disadvantages:

- For most problems, the number of states within the goal contour search space is still exponential in the length of the solution.

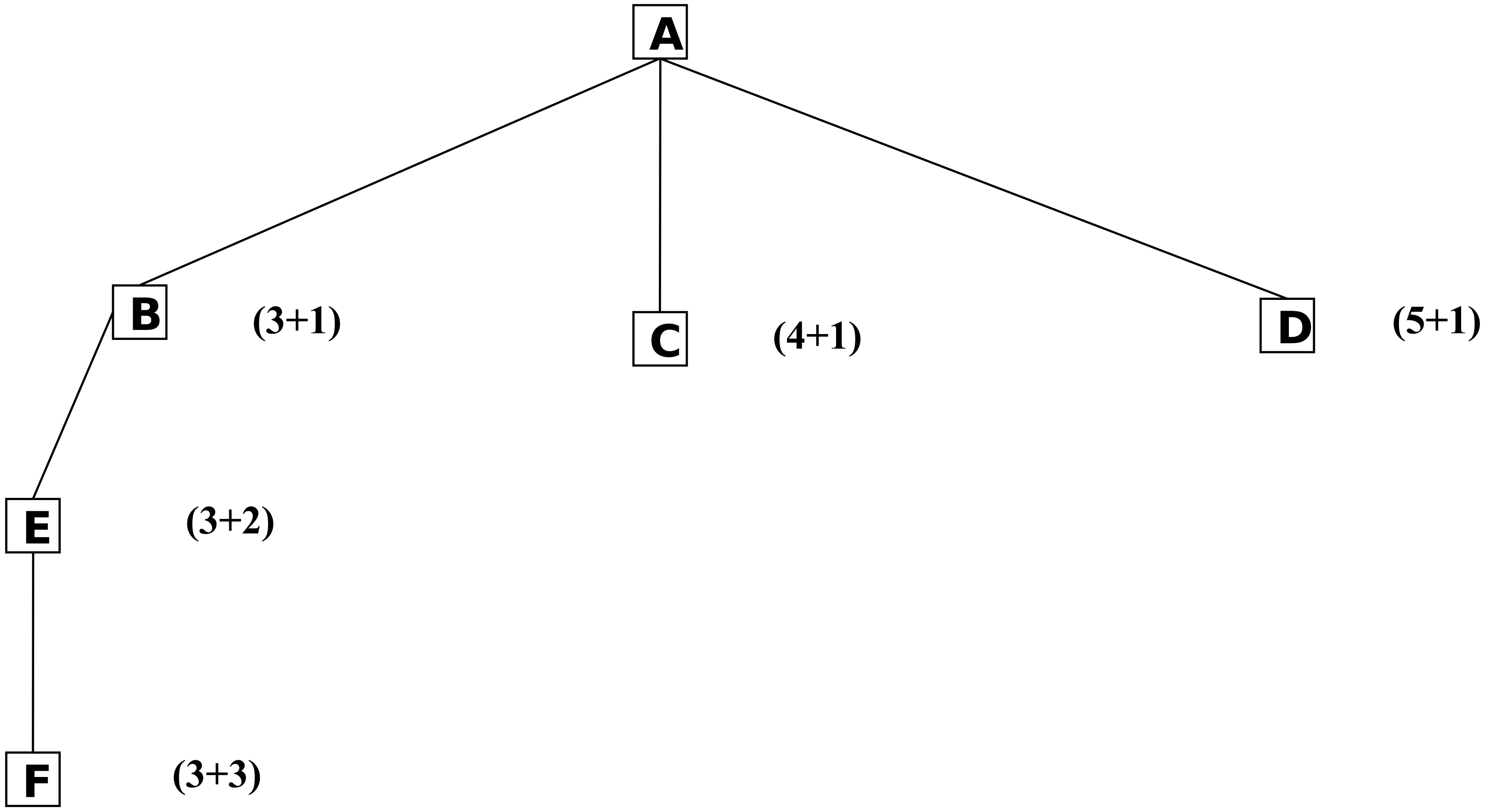
A* search (Continued…)

•That A* search is complete, optimal, and optimally efficient among all such algorithms.

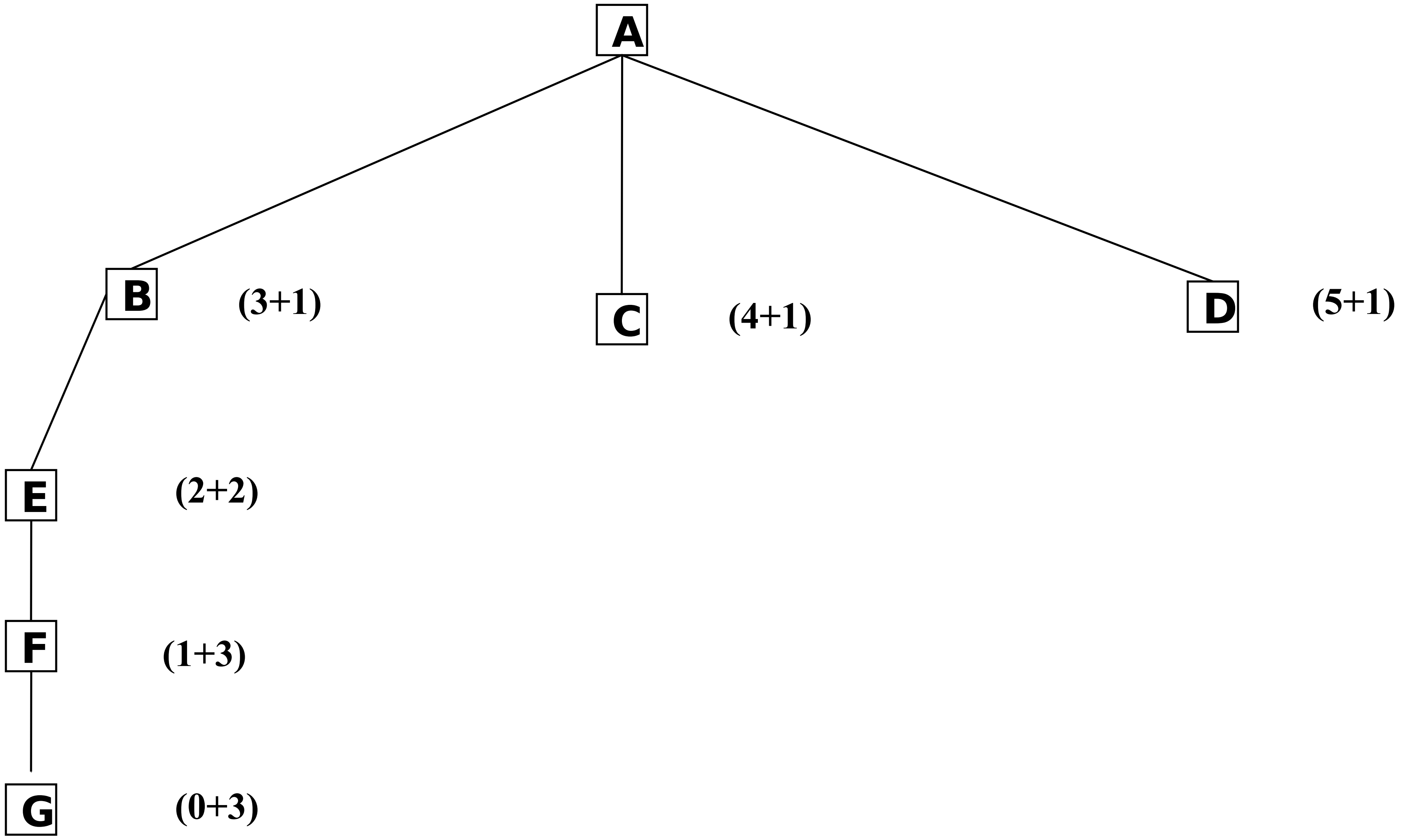
Conditions for optimality:

- Admissibility: The first condition we require for optimality is that h(n) be an admissible heuristic. An admissible heuristic is one that never overestimates the cost to reach the goal.

- Consistency: A second, slightly stronger condition called consistency (or sometimes monotonicity) MCNOTONICITY is required only for applications of A* to graph search.



h’ underestimates h at node B



h’ overestimates h at node D

Observations about A*

1. F(n)=g(n)+h(n)

2. If g=0 then F(n)=h(n), then the node that seems closest to the goal will be chosen (and it becomes GBFS).

3. If g=1 and h=0 then the path involving fewest number of steps will be chosen.

4. If h=0 , then F(n)=g(n) then the search will be controlled by g (and it becomes uniform cost search).

5. If h=0 and g= 0 then the search strategy will be random.

MEMORY-BOUNDED SEARCH (HEURISTIC SEARCH Technique) (Self Learning)

- The simplest way to reduce memory requirements for A* is to adapt the idea of ITERATIVE DEPENING deepening to the heuristic search context, resulting in the iterative-deepening A* (IDA*) algorithm.

- The main difference between IDA* and standard iterative deepening is that the cutoff used is the *f-cost (g + h) rather than the depth; at each iteration, the cutoff value is the small*est f-cost of any node that exceeded the cutoff on the previous iteration.

- IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes.

- Two memory bound algorithms are RBFS and MA*

End of Session