MANJARA CHARITABLE TRUST
RAJIV GANDHI INSTITUTE OF TECHNOLOGY, MUMBAI

Artificial Intelligence

(AI)

Course Code: CSC604

TE Sem-VI-2021-2022

Prof.D.S.Kale

Department of Computer Engineering

Course Objectives (CO):

1. To conceptualize the basic ideas and techniques underlying the design of intelligent systems.
2. To make students understand and Explore the mechanism of mind that enables intelligent thought and action.
3. To make students understand advanced representation formalism and search techniques.
4. To make students understand how to deal with uncertain and incomplete information.

Course Outcomes: Students should be able to –

1. Ability to develop a basic understanding of AI building blocks presented in intelligent agents.
2. Ability to choose an appropriate problem solving method and knowledge representation technique.
3. Ability to analyze the strength and weaknesses of AI approaches to knowledge– intensive problem solving
4. Ability to design models for reasoning with uncertainty as well as the use of unreliable information.
5. Ability to design and develop AI applications in real world scenarios.

Chapter. 1. Introduction to Artificial Intelligence

Introduction and Definition of Artificial Intelligence:

Artificial Intelligence is a branch of *Science which deals with helping machines finds solutions to complex problems in a more human-like fashion.*

This generally involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer friendly way.

A more or less flexible or efficient approach can be taken depending on the requirements established, which influences how artificial the intelligent behavior appears.

Introduction and Definition of Artificial Intelligence:

- AI is a branch of computer science which is concerned with the study and creation of computer systems that exhibit
  - Some form of intelligence

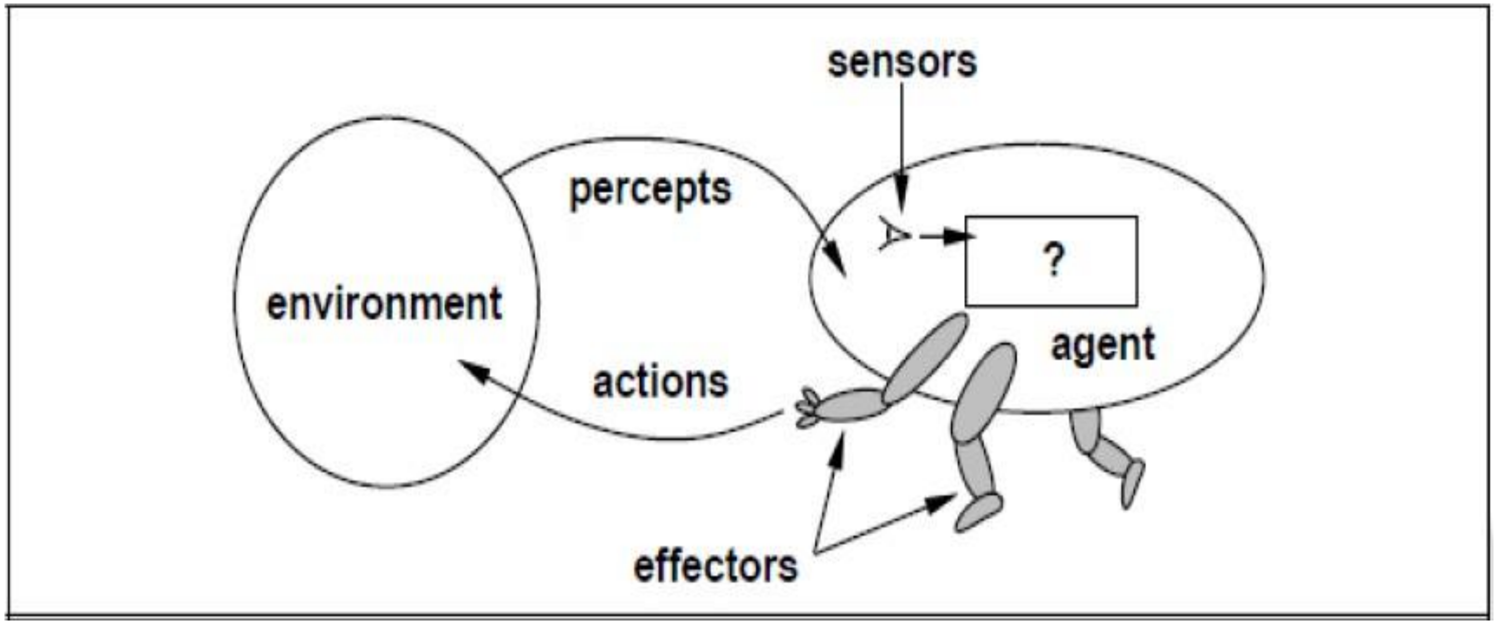
  OR

  - Those characteristics which we associate with intelligence in human behavior

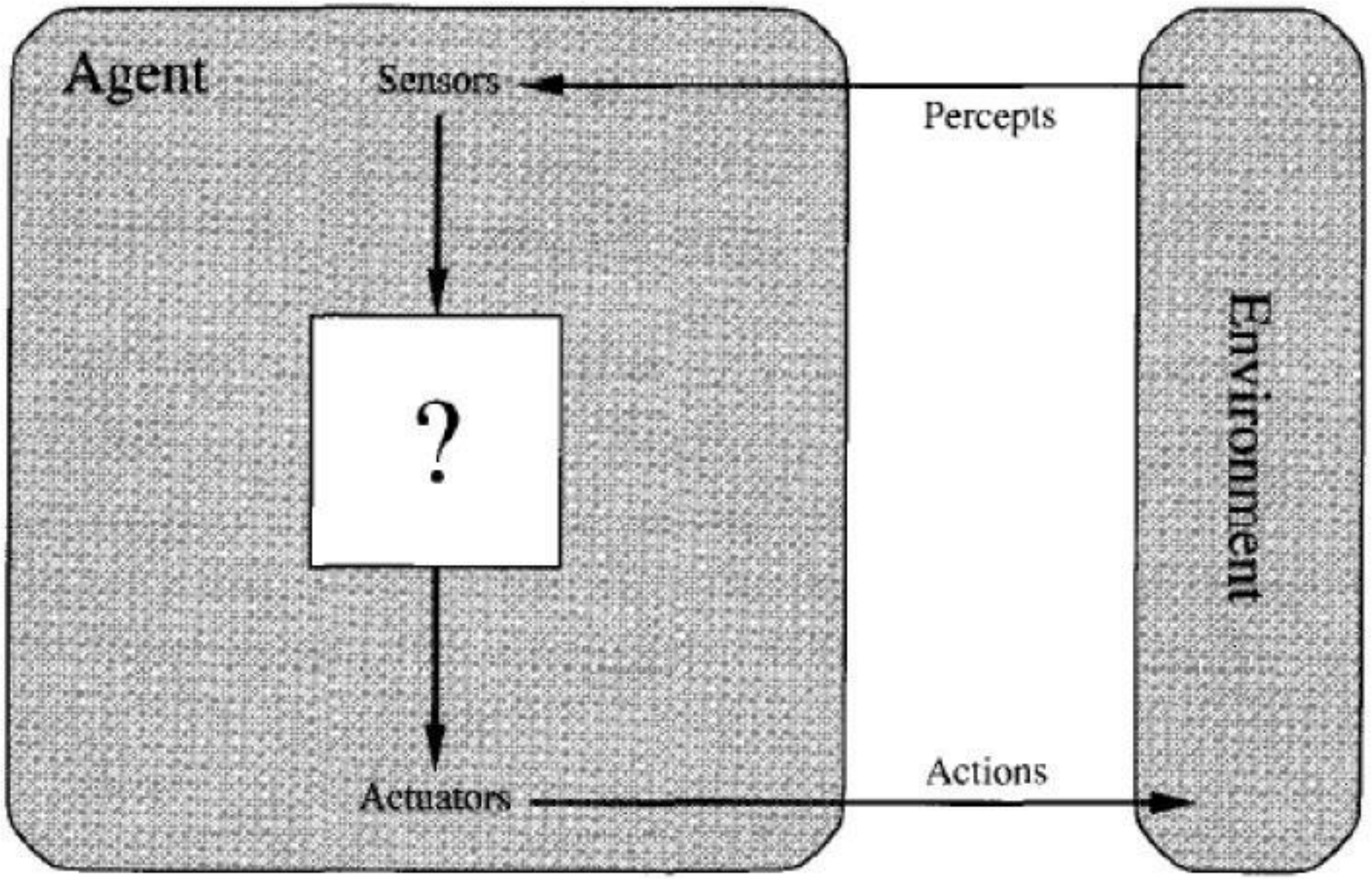
The branch of computer science concerned with making computers behave like humans.
"Artificial Intelligence is the study of human intelligence such that it can be replicated artificially."

Intelligent

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



Intelligent

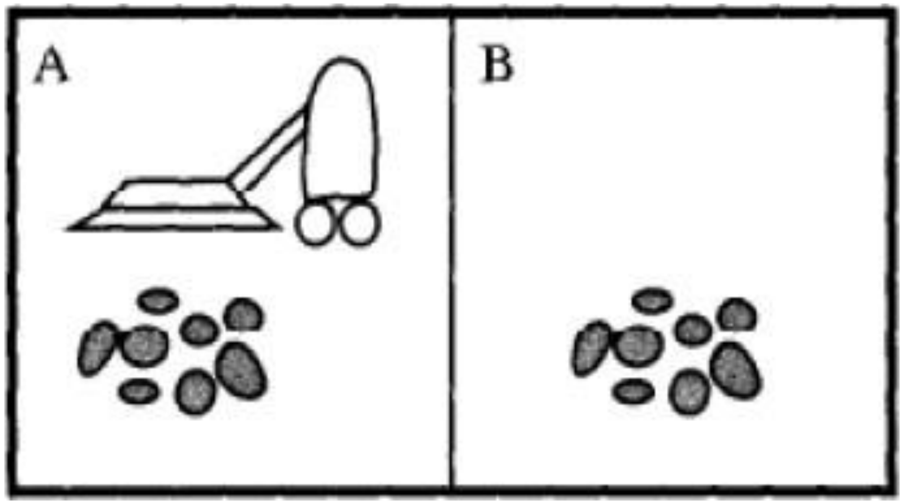


Agents interact with environments through sensors and actuators.

Intelligent

Example: Vacuum-cleaner world

- Perception: Clean or Dirty? Where it is in?
- Actions: Move left, Move right, suck(clean), do nothing(NoOp)



Program implements the **agent function** tabulated in above figure;

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

If *status = Dirty* ***then return***

Intelligent

Suck **else** if *location = A* ***then***
return *Right* **else** if *location =*
B ***then return*** *left*

Rationality

- A **rational agent** is one that does the right thing. As a first approximation, we will say that the right action is the one that will cause the agent to be most successful.
- That leaves us with the problem of deciding *how* and *when* to evaluate the agent's success.
- We use the term **performance measure** for the *how*—the criteria that determine how successful an agent is.

In summary, what is rational at any given time depends on four things:

- The **performance measure** that defines degree of success.
- Everything that the agent has perceived so far. We will call this complete perceptual history the **percept sequence.**
- What the agent knows about the **environment.**
- **The actions** that the agent can perform.

Rationality

This leads to a definition of an ***ideal rational agent***

- *For each possible percept sequence, an ideal rational agent should do whatever action is expected to maximize its performance measure, on the basis of the evidence provided by the percept sequence and whatever built-in knowledge the agent has.*

The Nature of Environment

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

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

- **Performance measure?**

  The performance given by taxi should make it most successful agent that is flawless performance.

  e.g. Safety, destination, profits, legality, comfort, . . .
- **Environment?**

  It is a first step in designing an agent. We should specify the environment which suitable for agent action. If swimming is the task for an agent then environment must be water not air.

  e.g. Streets/freeways, traffic, pedestrians, weather . . .
- **Actuators?**

  These are one of the important details of agent through which agent performs actions in related and specified environment.

  e.g. Steering, accelerator, brake, horn, speaker/display, . . .
- **Sensors?**

  It is the way to receive different attributes from environment.

  e.g. Cameras, accelerometers, gauges, engine sensors, keyboard, GPS . . .

(In designing an agent, the first step must always be to specify the task environment as fully as possible)

Nature of Environment

1. **Fully observable vs. partially observable**
2. **Deterministic vs. nondeterministic (stochastic)**
3. **Episodic vs. no episodic (Sequential).**
4. **Static vs. dynamic.**
5. **Discrete vs. continuous.**
6. **Single agent VS. Multi-agent**

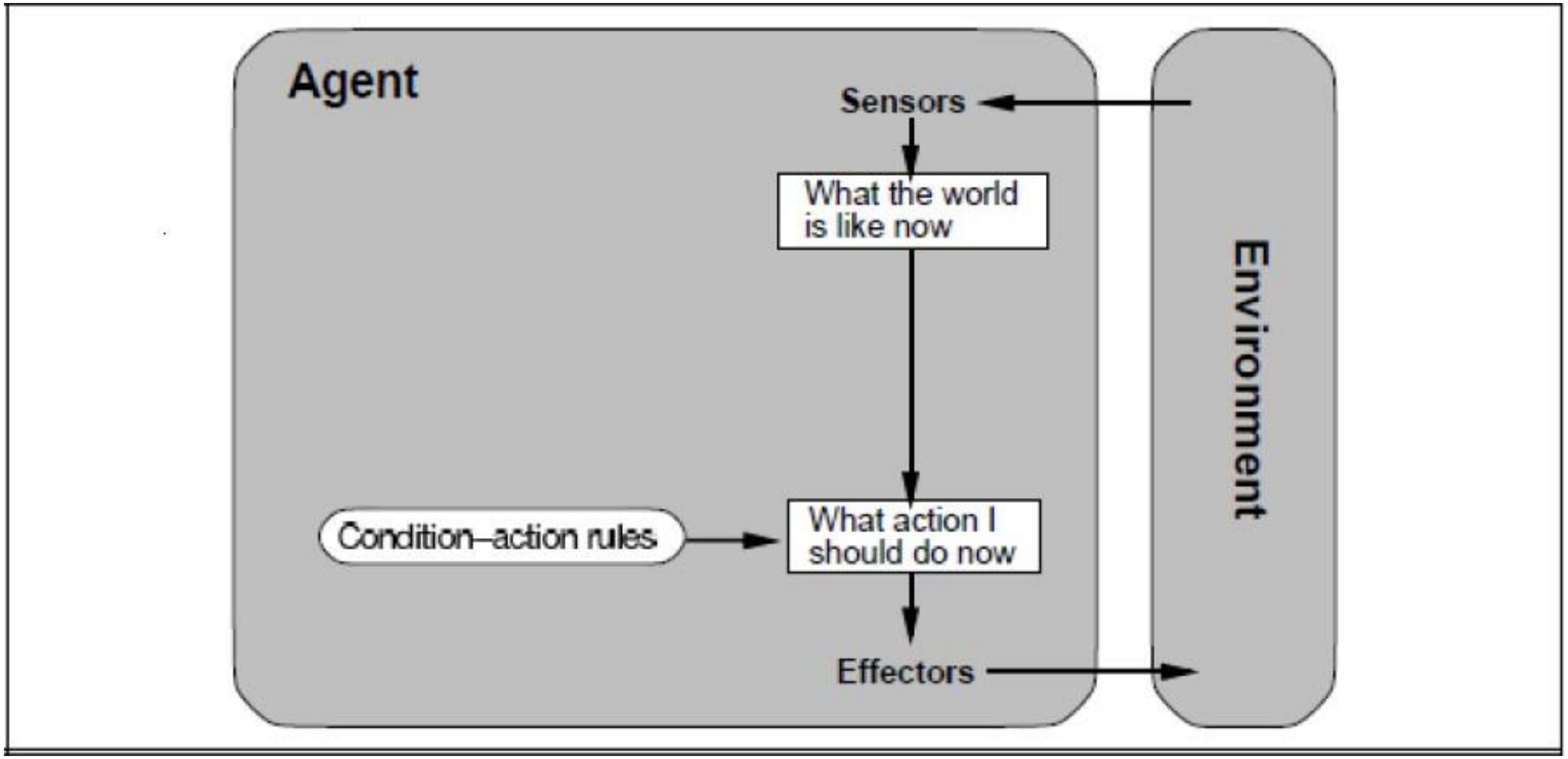
Structure of Agent

Agent = architecture + program

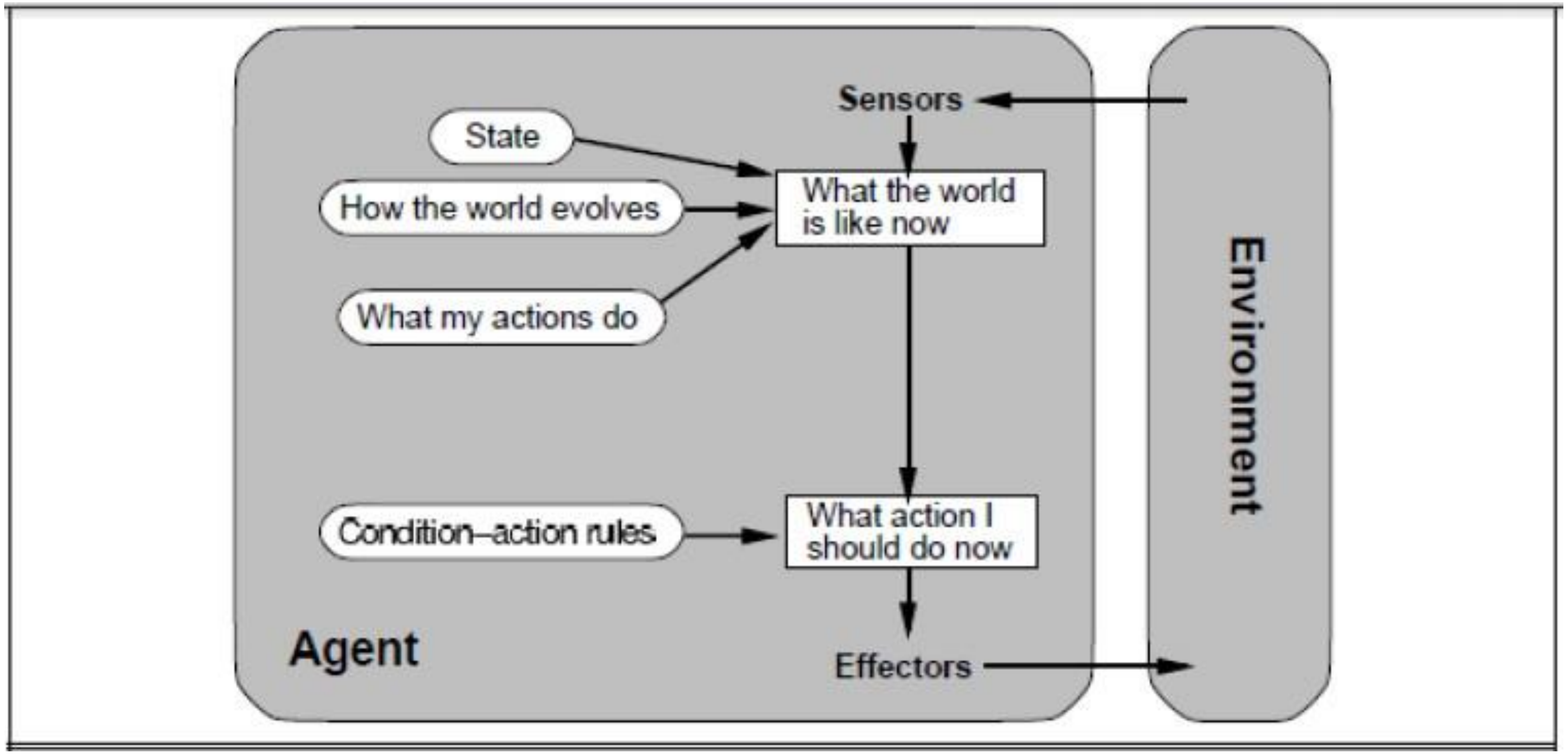
Types of Agent

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

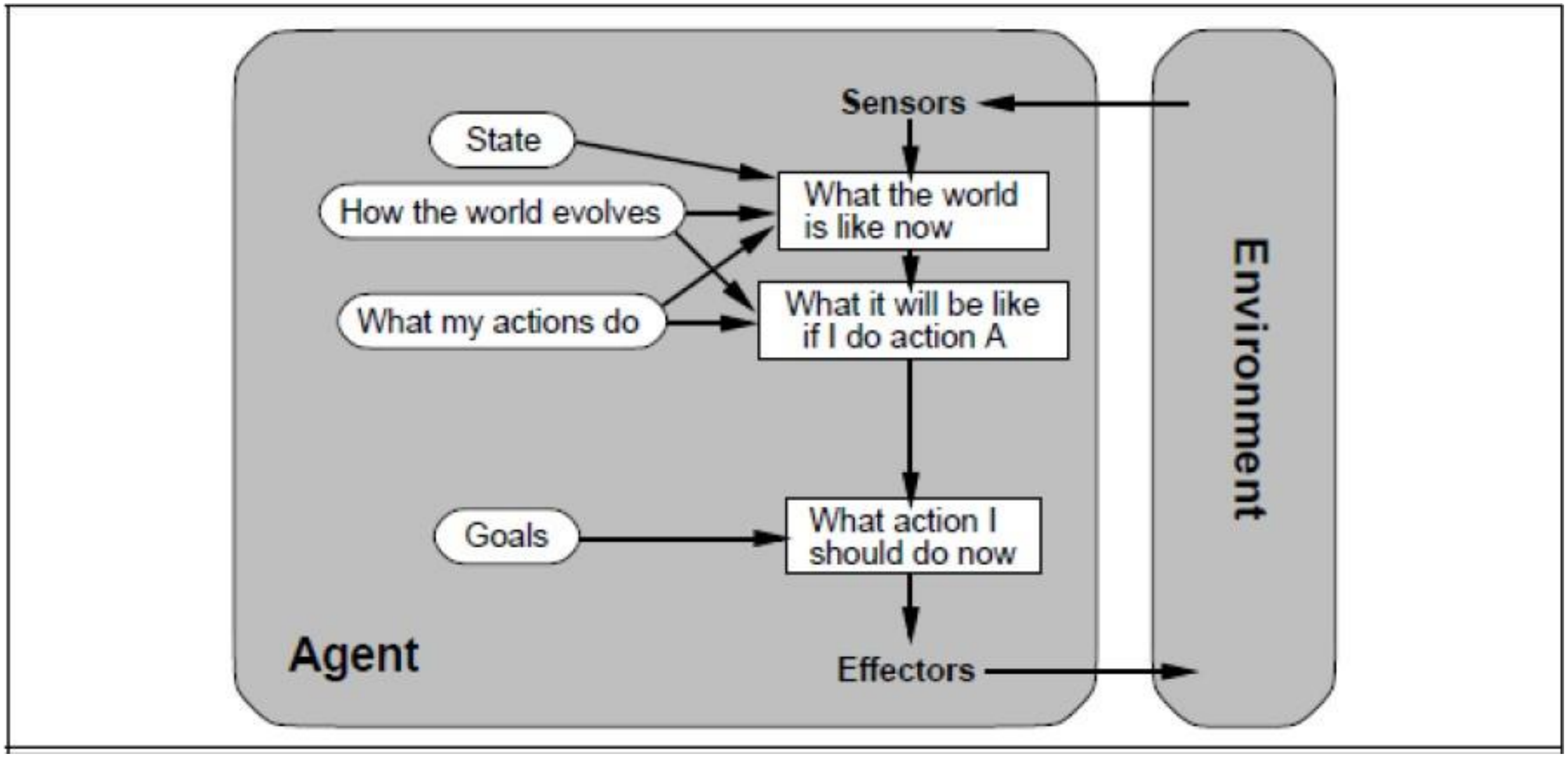
Simple reflex



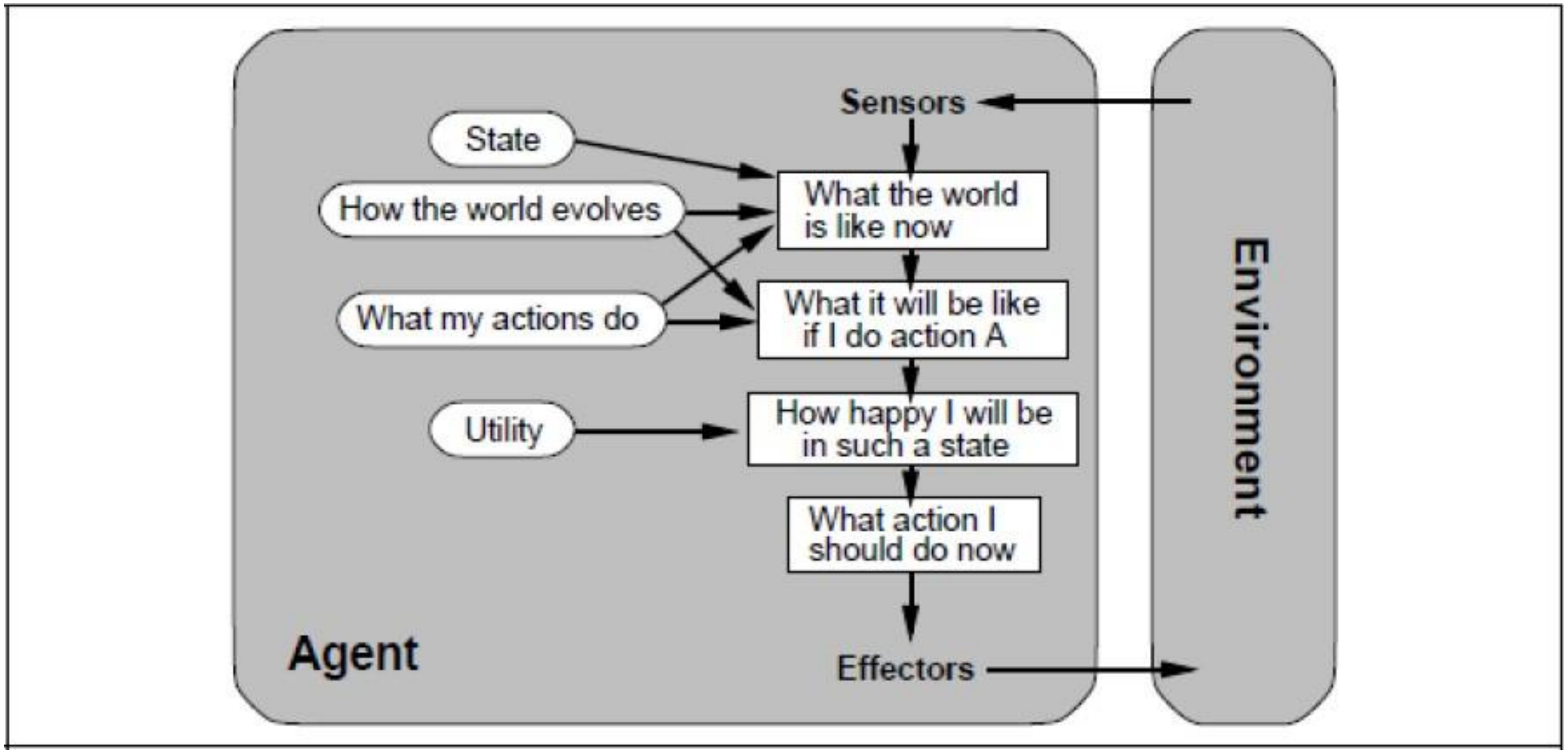
Model-Based reflex



Goal Based

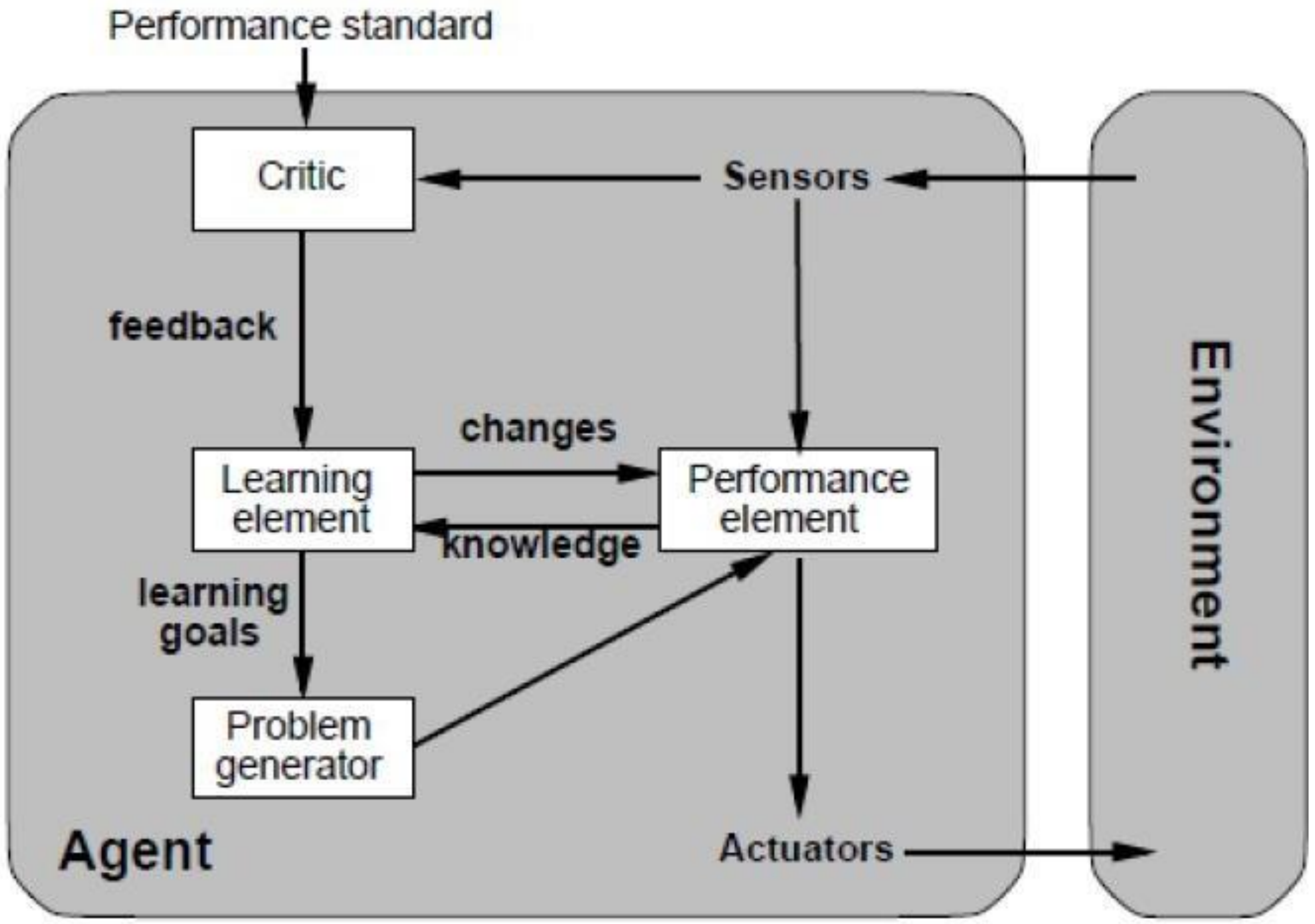

Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Effectors
Environment

Utility Based



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
Agent
Effectors
Environment

Learning


Performance standard
Critic
Sensors
feedback
changes
Learning element
Performance element
knowledge
learning goals
Problem generator
Agent
Actuators
Environment

Chapter.2. Problem Solving

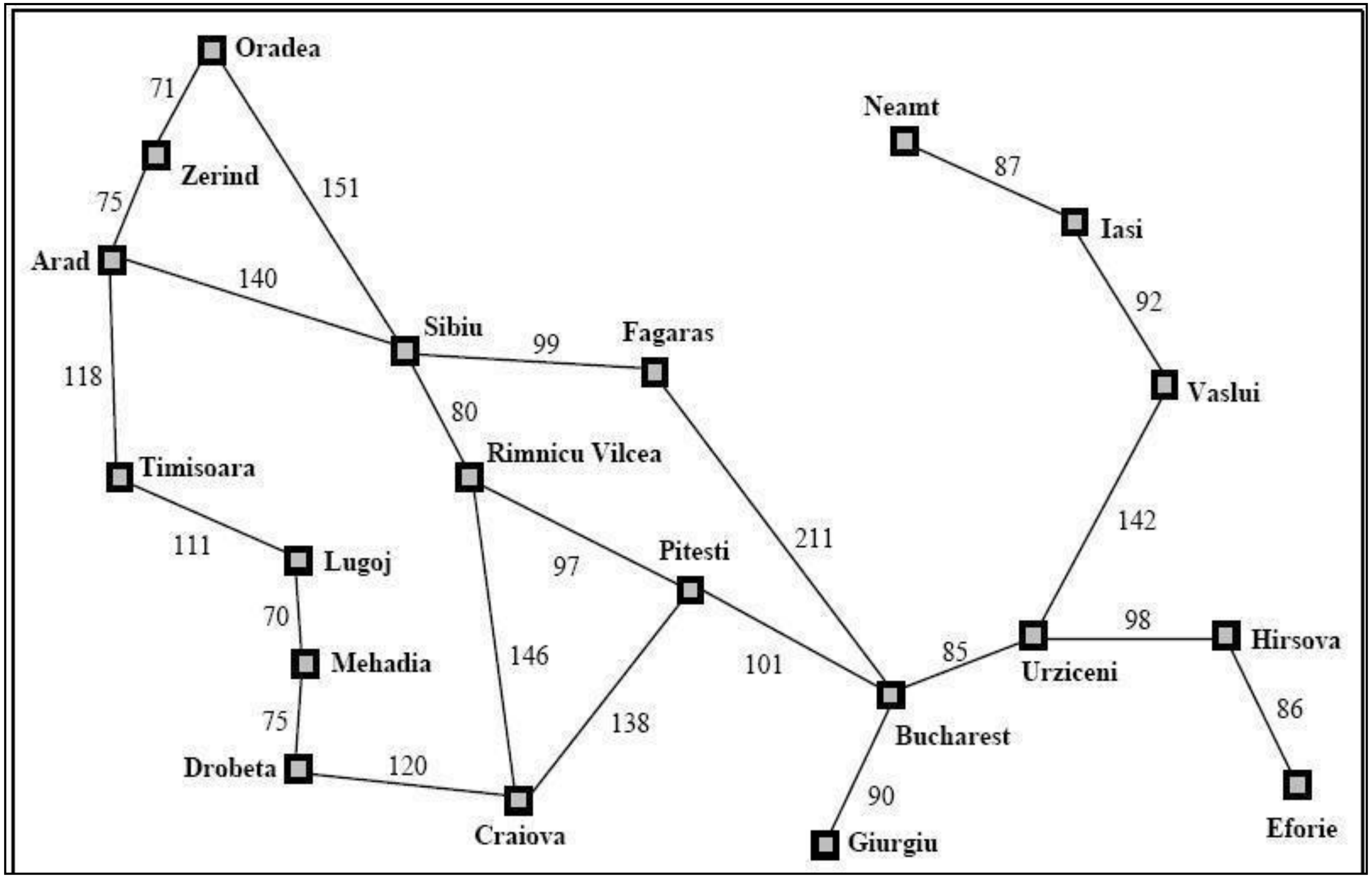
PROBLEM-SOLVING AGENTS

Intelligent agents are supposed to act in such a way that the environment goes through a sequence of states that maximizes the performance measure.

What is Problem solving agent?

- ➢ It is a kind of Goal-based Agents
- ➢ 4 general steps in problem-solving:
  - Goal Formulation
  - Problem Formulation
  - Search
  - Execute
- ➢ E.g. Driving from Arad to Bucharest...

Note: *In this chapter we will consider one example that "A map is given with different cities connected and their distance values are also mentioned. Agent starts from one city and reach to other."*


Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
138
Drobeta
120
Craiova
Neamt
87
Iasi
92
Vaslui
142
Urziceni
98
Hirsova
85
Bucharest
86
90
Giurgiu
Eforie

Subclass of goal-based agents

- Goal formulation
- Problem formulation
- Example problems
  - Toy problems
  - Real-world problems
- Search
  - Search strategies
  - Constraint satisfaction
- Solution

Goal Formulation

Goal formulation, based on the current situation, is the first step in problem solving. As well as formulating a goal, the agent may wish to decide on some other factors that affect the desirability of different ways of achieving the goal. For now, let us assume that the agent will consider actions at the level of driving from one major town to another. The states it will consider therefore correspond to being in a particular town.

- Declaring the Goal: Goal information given to agent *i.e. start from Arad and reach to Bucharest.*
- Ignoring the some actions: agent has to ignore some actions that will not lead agent to desire goal.
- Limits the objective that agent is trying to achieve: Agent will decide its action when he has some added knowledge about map. *i.e. map of Romania is given to agent.*
- Goal can be defined as set of world states.

Problem Formulation

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

- Process of looking for action sequence (number of action that agent carried out to reach to 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. Thus, we have a simple "formulate, search, execute" design for the agent.

Well-defined problem and solutions

A *problem* is defined by four items:

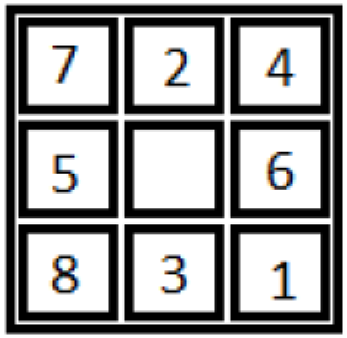
- **Initial State**: The initial state that the agent starts in. e.g., “In(Arad)”.
- **Successor Function S(x)** = <action, successor> ordered pair.
  e.g. ,from state In(Arad),the successor function for Romania problem would return

  {<Go(Zerind),In(Zerind)>,<Go(sibiu),In(sibiu)>,

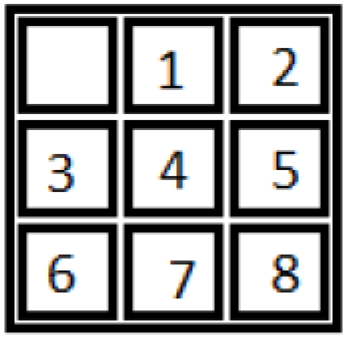
  <Go(Timisoara),In(Timisoara)>}.
- **Goal Test=**It determines whether a given state is a goal state.
- **Path Ccost** =Function that assigns a numeric cost to each path. e.g., sum of distances, number of actions executed, etc. Usually given as c(x, a, y), the step cost from x to y by action a, assumed to be ≥ 0.

Example Problems:

The 8-puzzle consists of a 3x3 board with 8 numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space.



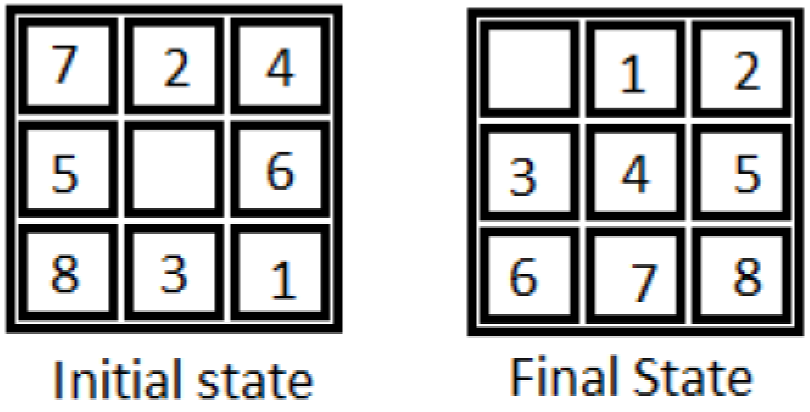
Initial state



Final State

Initial State/Function: ? Successor
function: ?
Goal State: ?
Path cost: ?

Example problems:



- **Initial State/Function**: Any state can be designated as the initial state.
- **Successor function**: This generates the legal states that result from trying the four actions (blank *moves Left, Right, Up or Down).*
- **Goal State**: This checks whether the state matches the goal configuration shown in figure.
- **Path cost**: Each step cost 1; so the path cost is the number of steps in path.

Example problems:

Problem: Missionaries and cannibals - 3 missionaries and 3 cannibals are stood on one side of a river with a boat. All 6 must cross the river, but the boat can only hold two people at once and the missionaries must never be left outnumbered at any point on either side.

- State: ?
- Operator: ?
- Goal test: ?
- Path cost: ?

There are two types of searching strategies,used in path finding,

1) Uninformed Search strategies.
2) Infirmed Search strategies.

Uninformed Search Methods

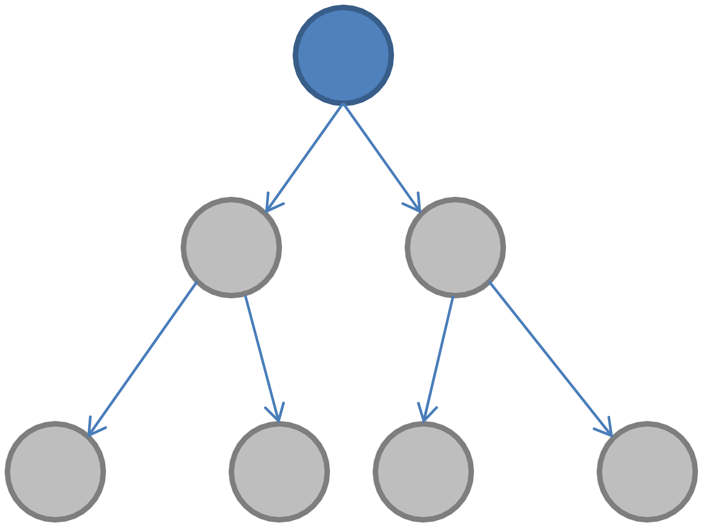
Uninformed search means that they have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from non-goal state.

***Uninformed* strategies use only the information available in the problem definition**

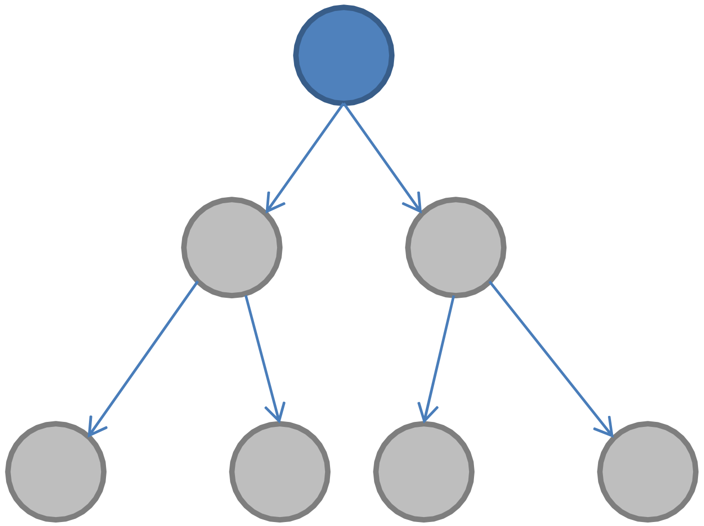
1) Breadth-first search
2) Depth-first search
3) Depth-limited search
4) Iterative deepening search

Uninformed Search strategy also called “Blind Search”.

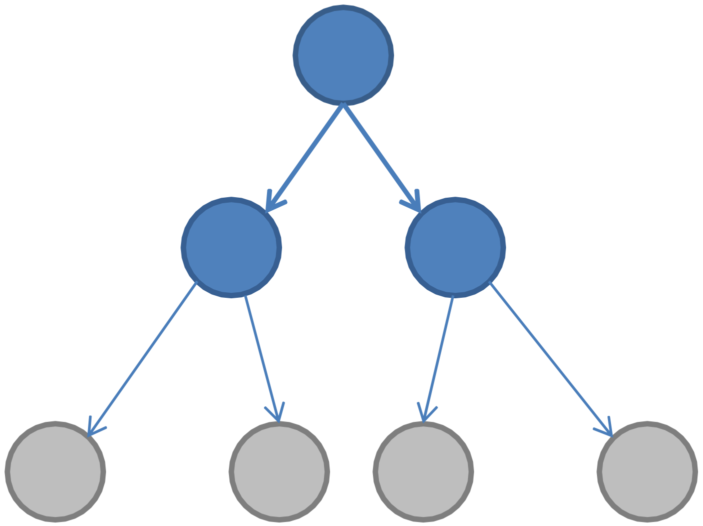
Breadth First



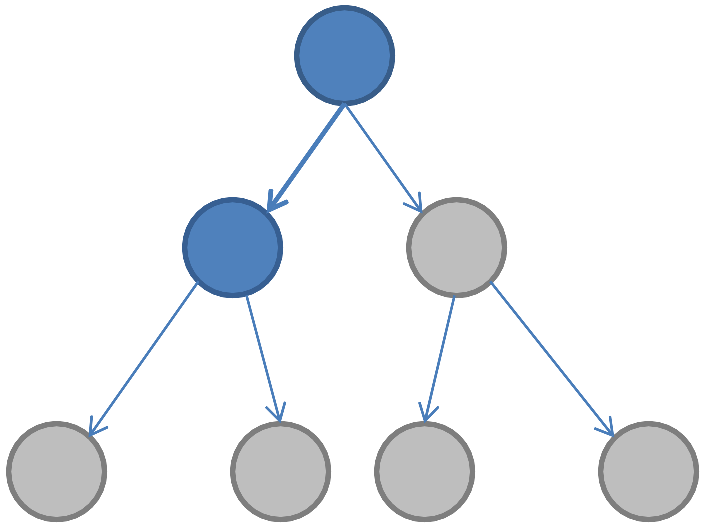
Depth First



Breadth First

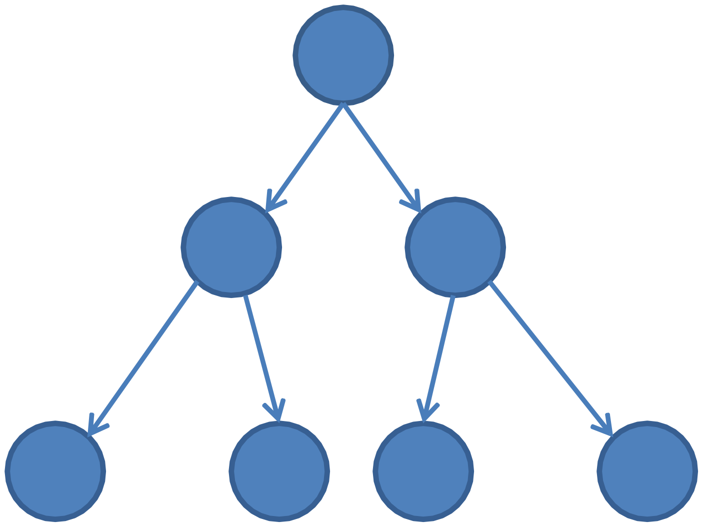


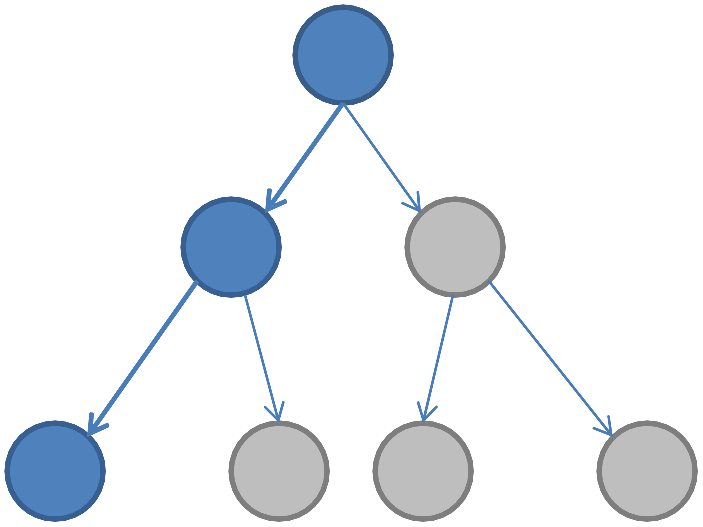
Depth First



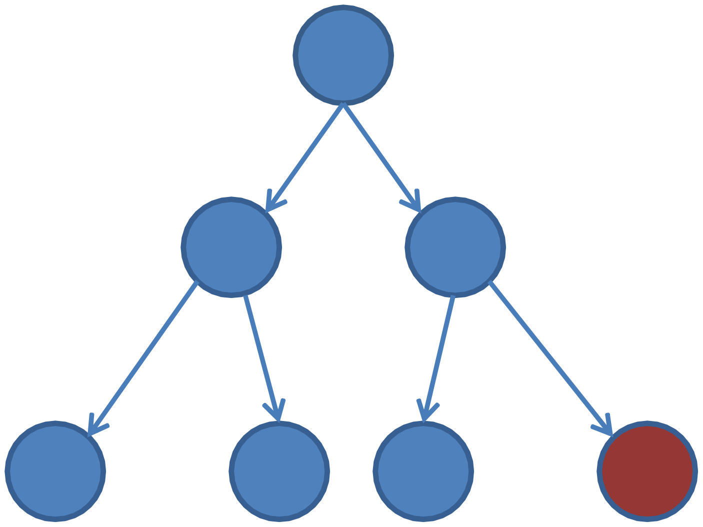
Breadth First

Depth First

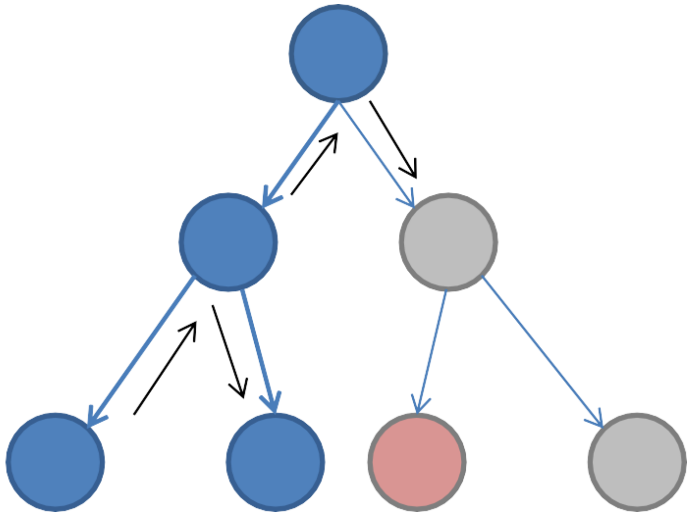




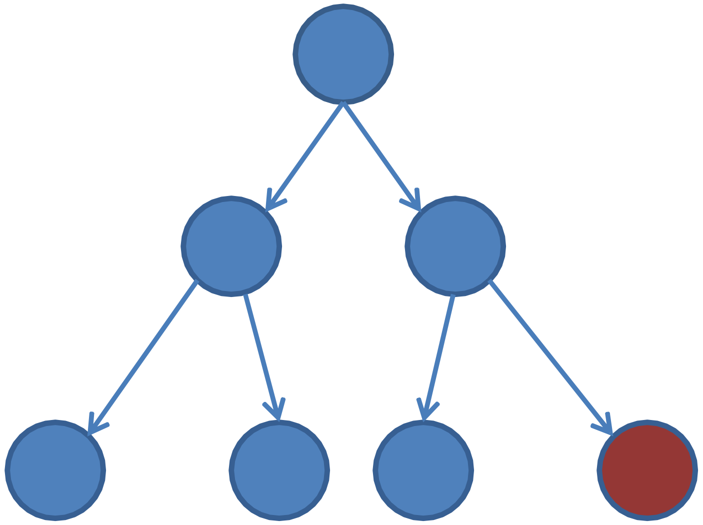
Breadth First



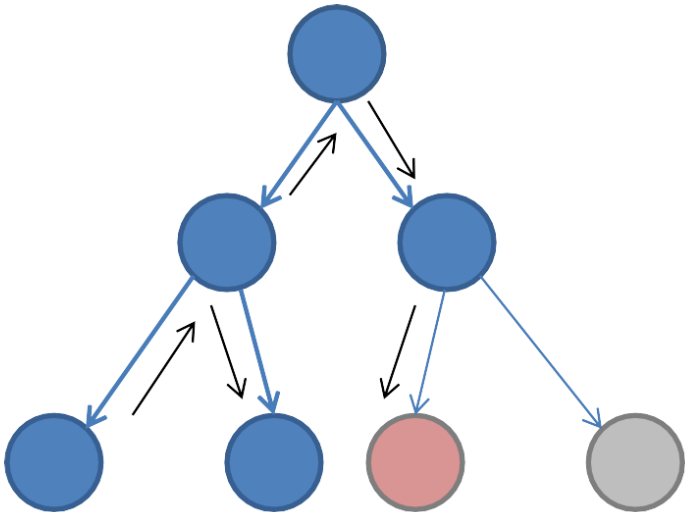
Depth First



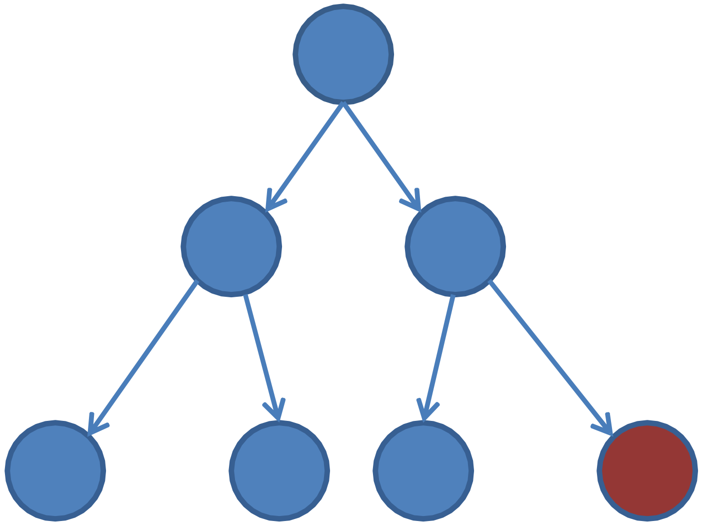
Breadth First



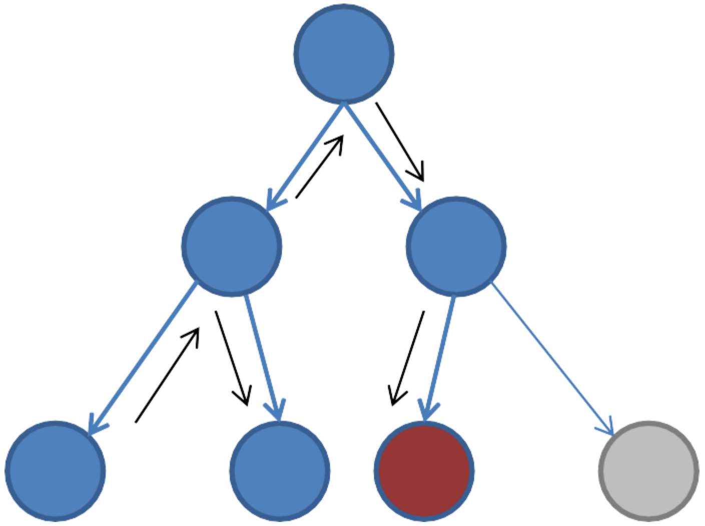
Depth First



Breadth First



Depth First



Breadth First

Depth First

Breadth First

Algorithm:

1. Place the starting node.
2. If the queue is empty return failure and stop.
3. If the first element on the queue is a goal node, return success and stop otherwise.
4. Remove and expand the first element from the queue and place all children at the end of the queue in any order.

Depth First

Algorithm:

1. Push the root node onto a stack.
2. Pop a node from the stack and examine it.
    1. If the element sought is found in this node, quit the search and return a result.
    2. Otherwise push all its successors (child nodes) that have not yet been discovered onto the stack.
3. If the stack is empty, every node in the tree has been examined – quit the search and return "not found".
4. If the stack is not empty, repeat from Step 2.

Depth Limited Search:

- Depth limited search (DLS) is a modification of depth-first search that minimizes the depth that the search algorithm may go.
- In addition to starting with a root and goal node, a depth is provided that the algorithm will not descend below.
- Any nodes below that depth are omitted from the search.
- This modification keeps the algorithm from indefinitely cycling by halting the search after the pre-imposed depth.
- The time and space complexity is similar to DFS from which the algorithm is derived.
- Space complexity: O(bd) and Time complexity : O(bd)

Why do Breadth First Search (BFS) and Depth First Search (DFS) fail in the case of an infinite search space?

- In DFS, you would recursively look at a node's adjacent vertex. DFS may not end in an infinite search space. Also, DFS may not find the shortest path to the goal. DFS needs O(d) space, where d is depth of search.
- BFS consumes too much memory. BFS needs to store all the elements in the same level. In the case of a tree, the last level has N / 2 leaf nodes, the second last level has N / 4. So, BFS needs O(N) space.

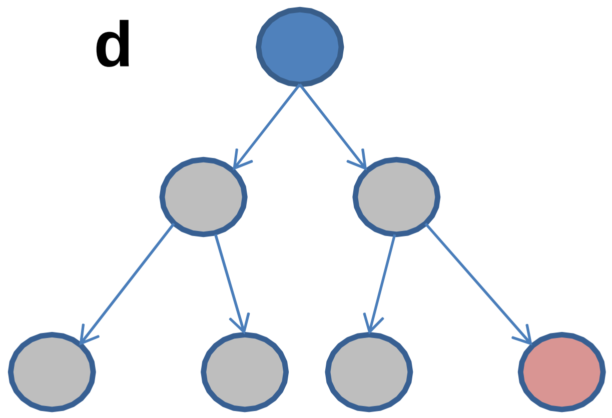
Iterative deepening depth first search (IDDFS) is a hybrid of BFS and DFS.

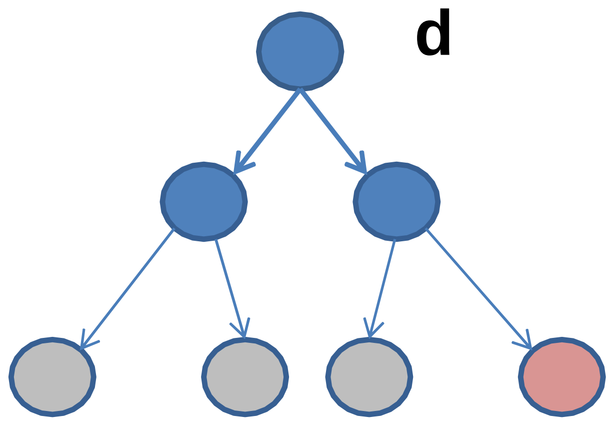
In IDDFS, we perform DFS up to a certain "limited depth," and keep increasing this "limited depth" after every iteration.

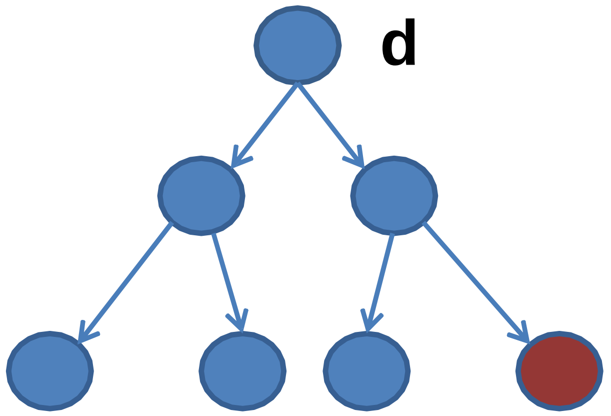
Depth First Iterative Deepening(DFID) Search or Iterative Deepening Search(DFS)

- Iterative Deepening Search (IDS) is a derivative of DLS and combines the feature of depth-first search with that of breadth-first search.
- IDS operate by performing DLS searches with increased depths until the goal is found.
- The depth begins at one, and increases until the goal is found, or no further nodes can be enumerated.
- By minimizing the depth of the search, we force the algorithm to also search the breadth of a graph.
- If the goal is not found, the depth that the algorithm is permitted to search is increased and the algorithm is started again.

Depth First Iterative Deepening(DFID) Search or Iterative Deepening Search(DFS)






d

Informed Search techniques

- A strategy that uses problem-specific knowledge beyond the definition of the problem itself.

- Also known as “heuristic search,” informed search strategies use information about the domain to (try to) (usually) head in the general direction of the goal node(s)

- Informed search methods: Hill climbing, best-first search(BFS), greedy best first search, beam search, A*.

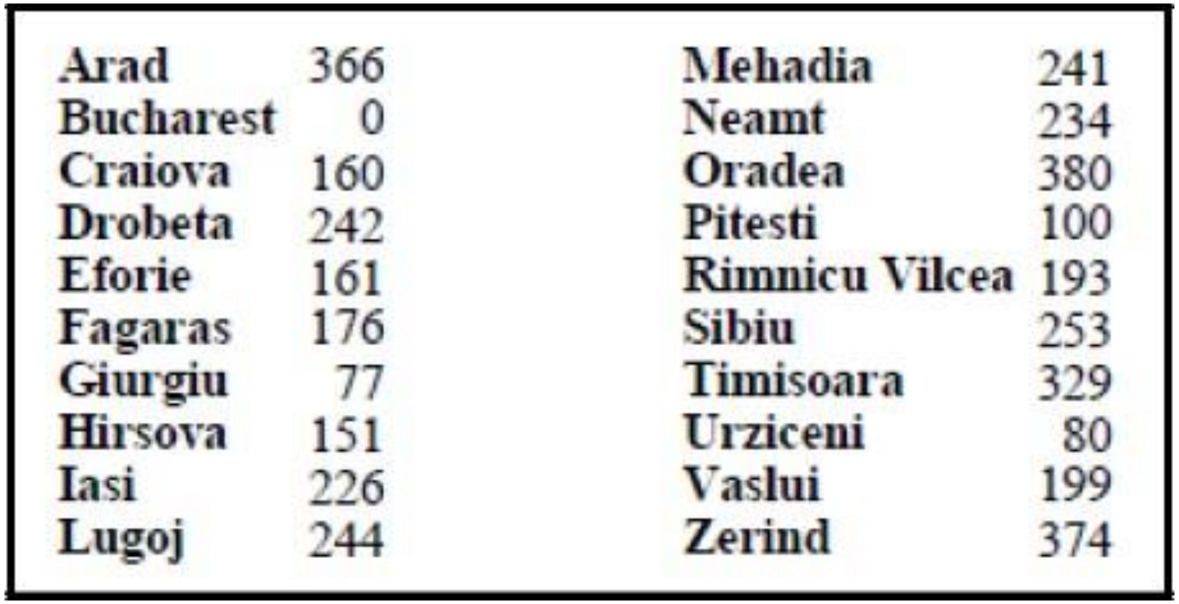
Best First Search(BFS)

- It is an algorithm in which a node is selected for expansion based on an evaluation function f(n).
- Choose the node that *appears* to be the best.
- There is a whole family of Best-First Search algorithms with different evaluation functions. Each has a heuristic function h(n).
- *h(n) = estimated cost of the cheapest path from node n to a goal node*
- Example: in route planning the estimate of the cost of the cheapest path might be the straight line distance between two cities.
- Best-First Search can be represented in two ways,
  - Greedy Best-First Search
  - A* search

Greedy Best-First Search:

- Greedy Best-First search tries to expand the node that is closest to the goal assuming it will lead to a solution quickly
  - $f(n) = h(n)$
  - aka “Greedy Search”
- Implementation
  - Expand the “most desirable” node into the fringe queue.
  - Sort the queue in decreasing order of desirability.

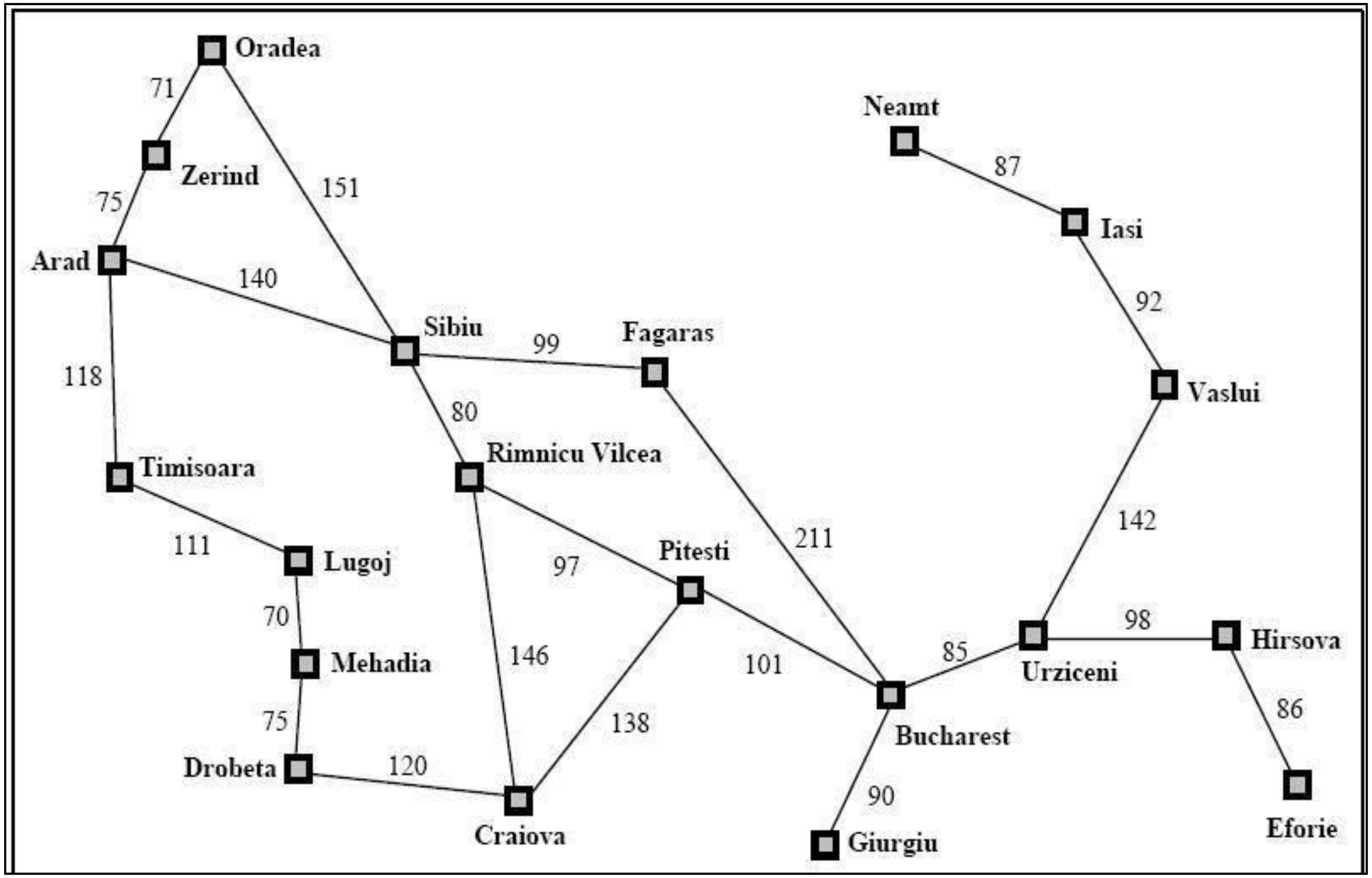
Example: Consider map of Romania, distances given below are straight line distance from city to goal city(Bucharest)



Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

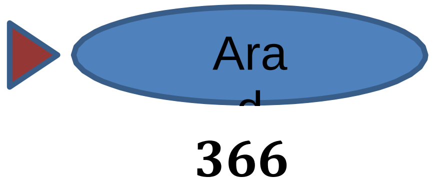
For Greedy BFS f(n) = h(n) , f(n) is a evaluation function.
h(n)SLD = straight line distance(value) from current city to goal city.
h(n) known as heuristic function.

Note: *In this chapter we will consider one example that "A map is given with different cities connected and their distance values are also mentioned. Agent starts from one city and reach to other."*


Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
138
Drobeta
120
Craiova
Neamt
87
Iasi
92
Vaslui
142
98
Hirsova
Urziceni
85
Bucharest
86
90
Giurgiu
Eforie

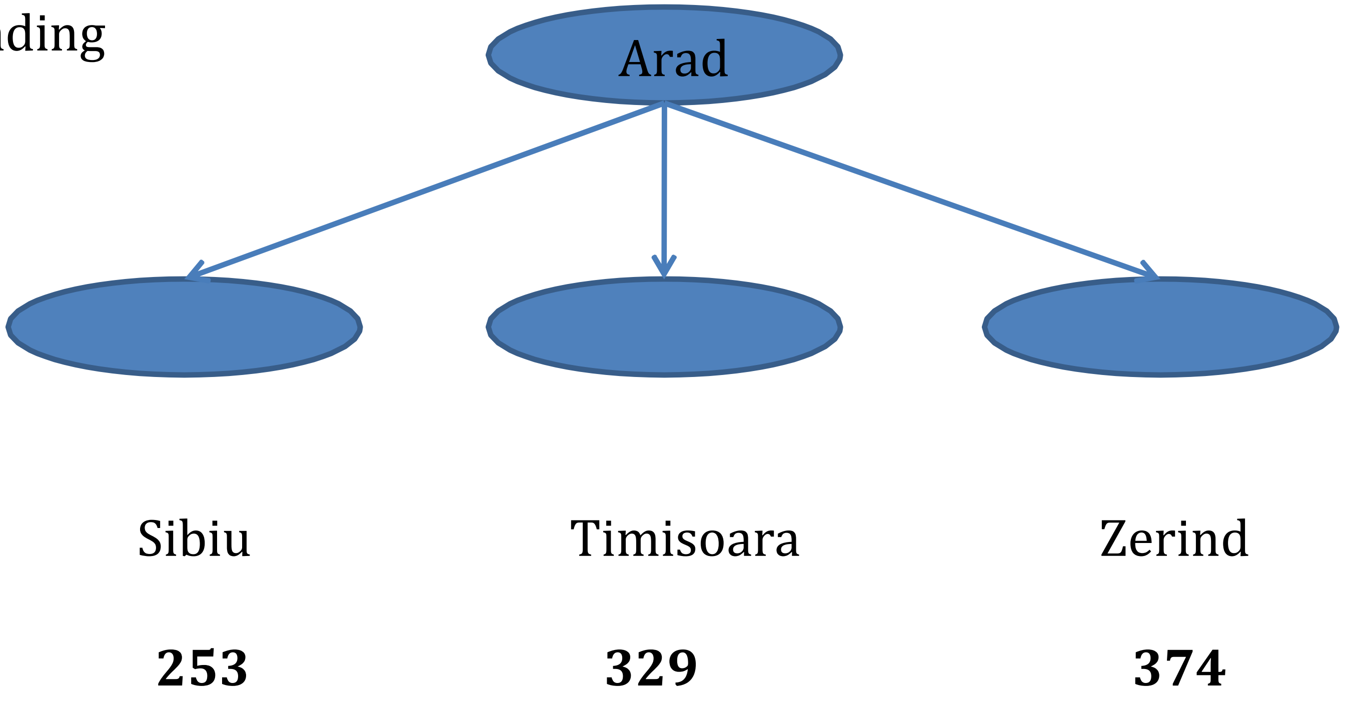
Map

(a) Initial State



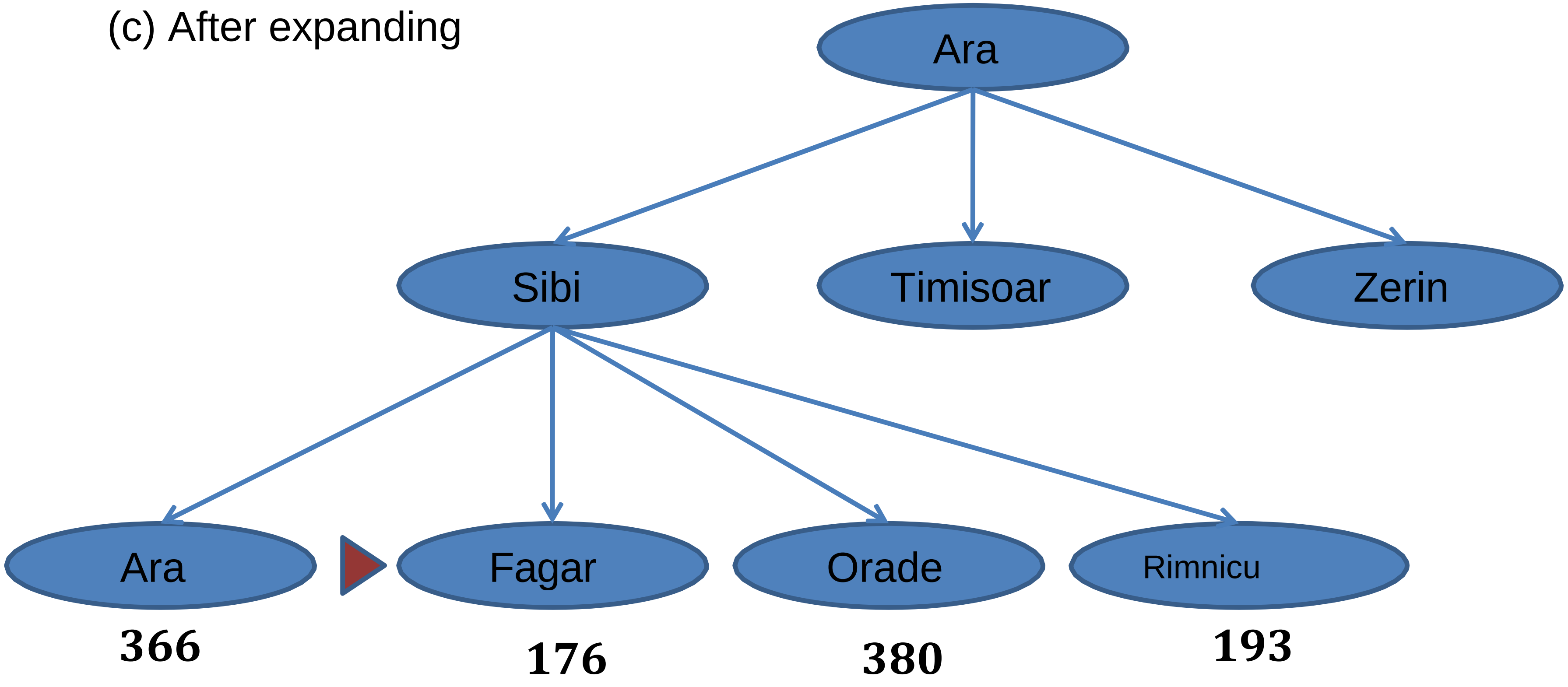
Map

(b) After expanding Arad



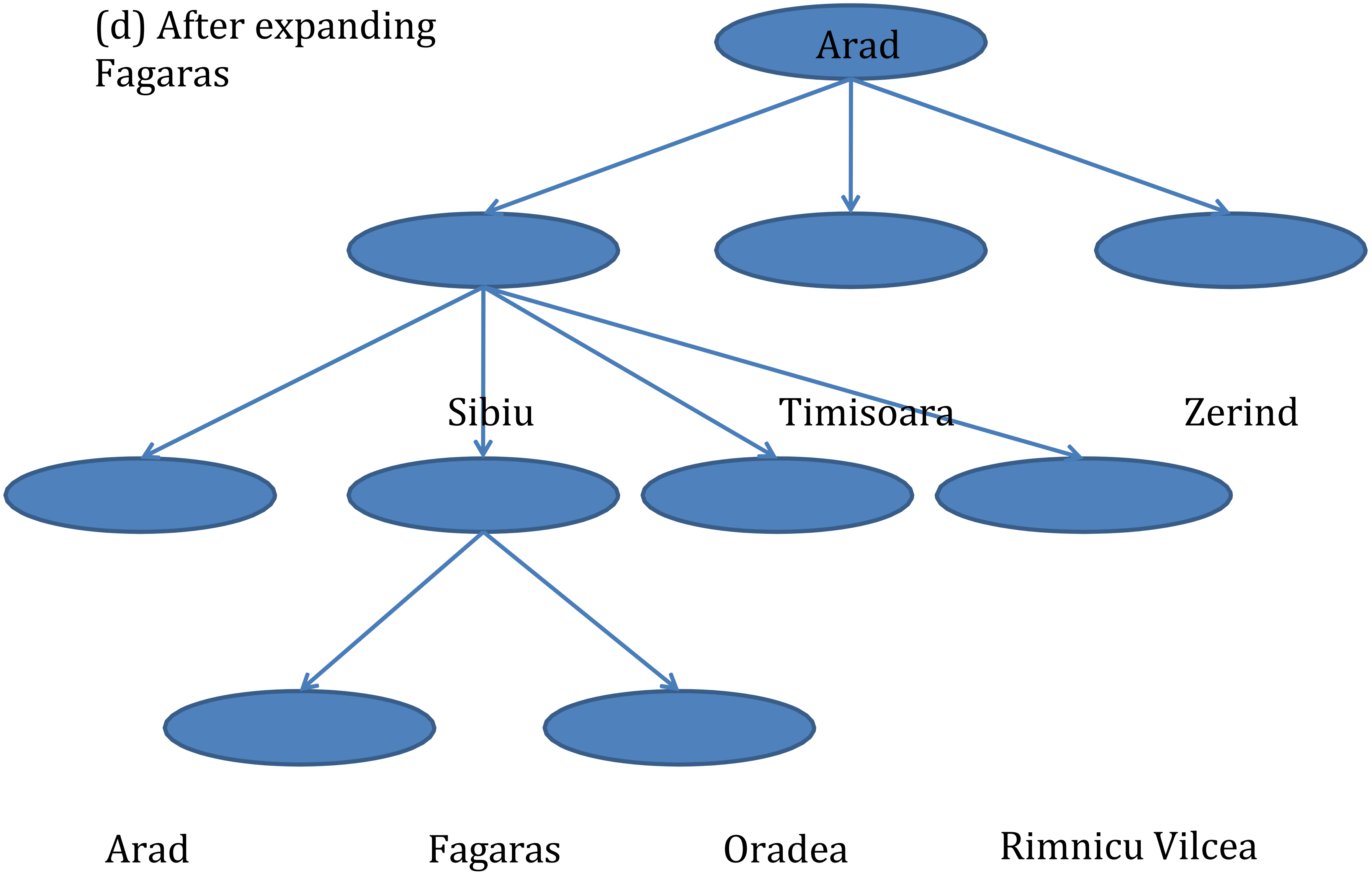
Map

(c) After expanding



Map

(d) After expanding Fagaras

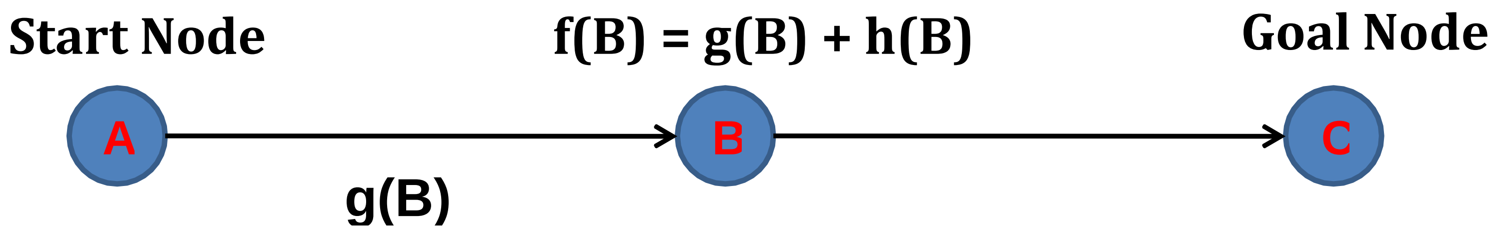


Map

Sibiu	▶	Bucharest	
253		**0**	**Goal State**

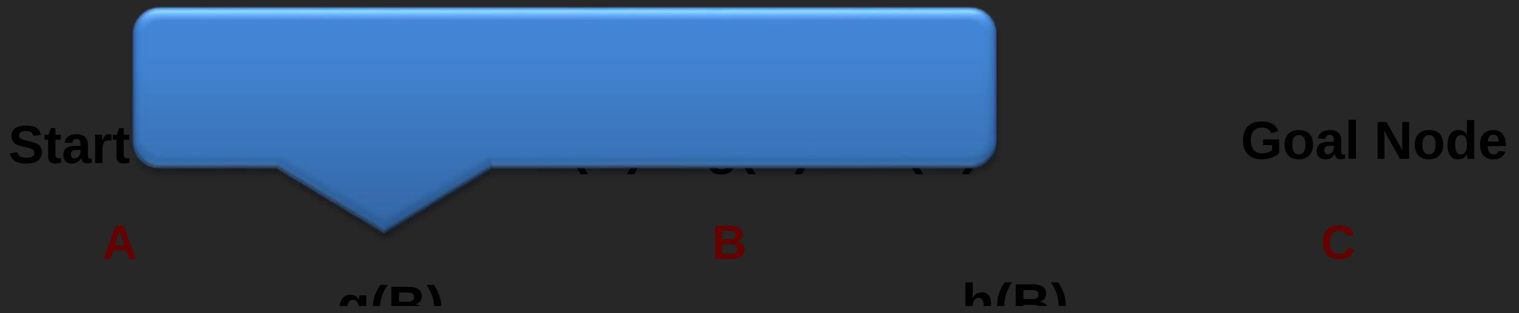
A* Search

- A* (A star) is the most widely known form of Best-Firstsearch
  - It evaluates nodes by combining g(n) and h(n).
  - f(n) = g(n) + h(n).
  - Where
    - g(n) = cost so far to reach n.
    - h(n) = estimated cost to goal from n.
    - f(n) = estimated total cost of path through n.

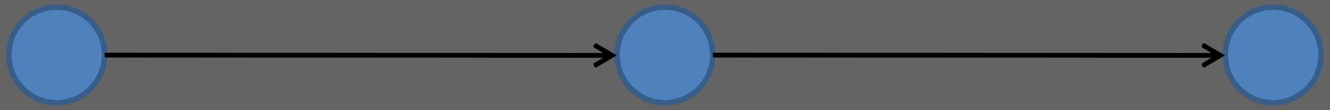


A* Search

- A* (A star) is the most widely known form of Best-First search
  - ➢ It evaluates nodes by combining g(n) and h(n).
  - ➢ f(n) = g(n) + h(n).
  - ➢ Where
    - g(n) = cost so far to reach n.
    - h(n) = estimated cost to goal from n.
    - f(n) = estimated total cost of path through n.



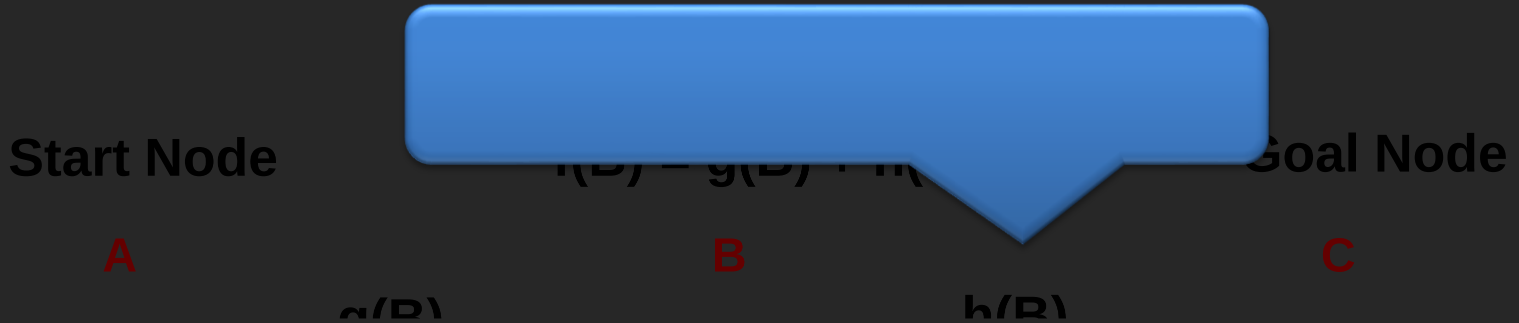
Distance from root node to current node



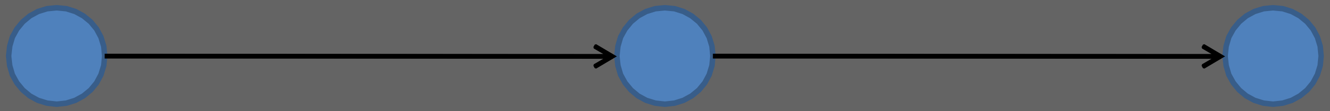
A* Search

• A* (A star) is the most widely known form of Best-First search

- ➢ It evaluates nodes by combining g(n) and h(n).
- ➢ f(n) = g(n) + h(n).
- ➢ Where
  - g(n) = cost so far to reach n.
  - h(n) = estimated cost to goal from n.
  - f(n) = estimated total cost of path through n.

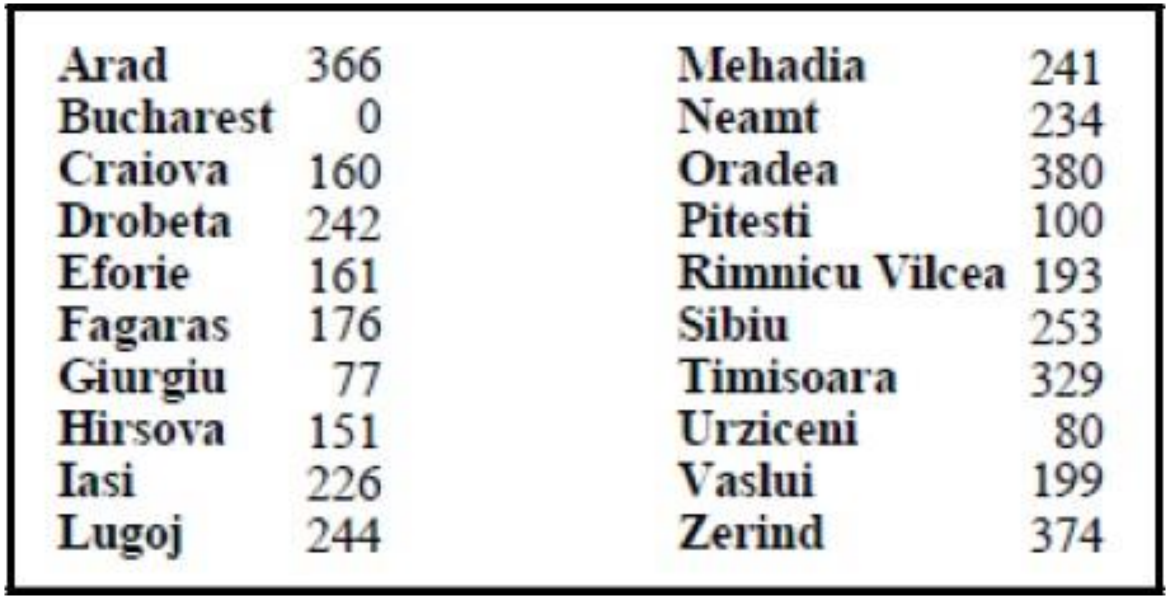


Distance from current node to goal node



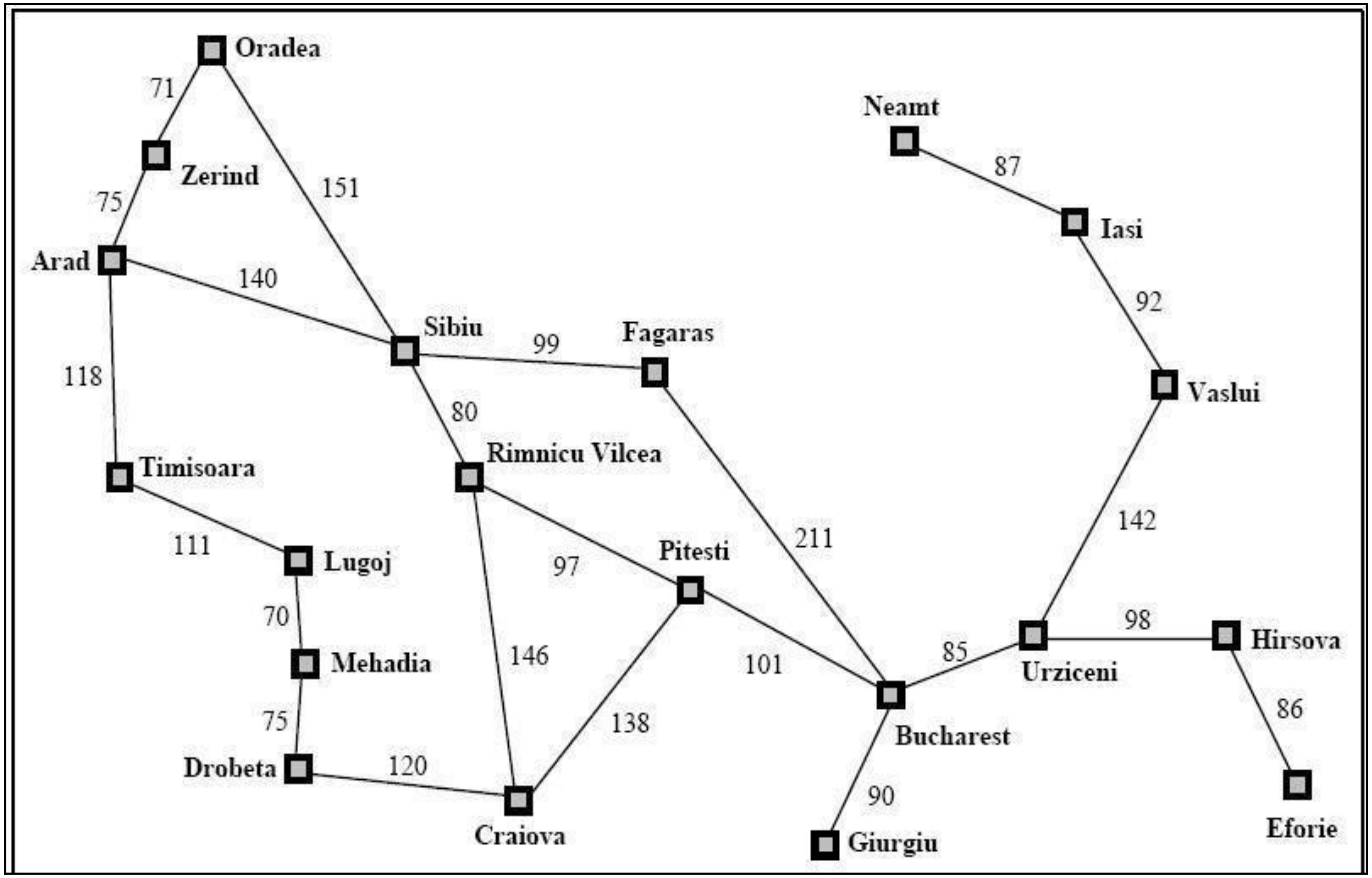
A* Example

- Evaluation function f(n)= g(n)+h(n)
- Now lets consider previous problem statement, in which map of Romania given, agent starts from Arad city and goal is to reach at Bucharest city.
- Let's apply A* and compare with Best First Search

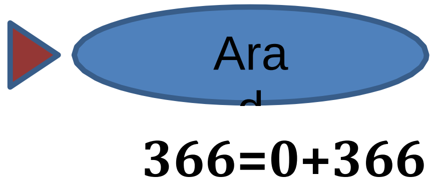


Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

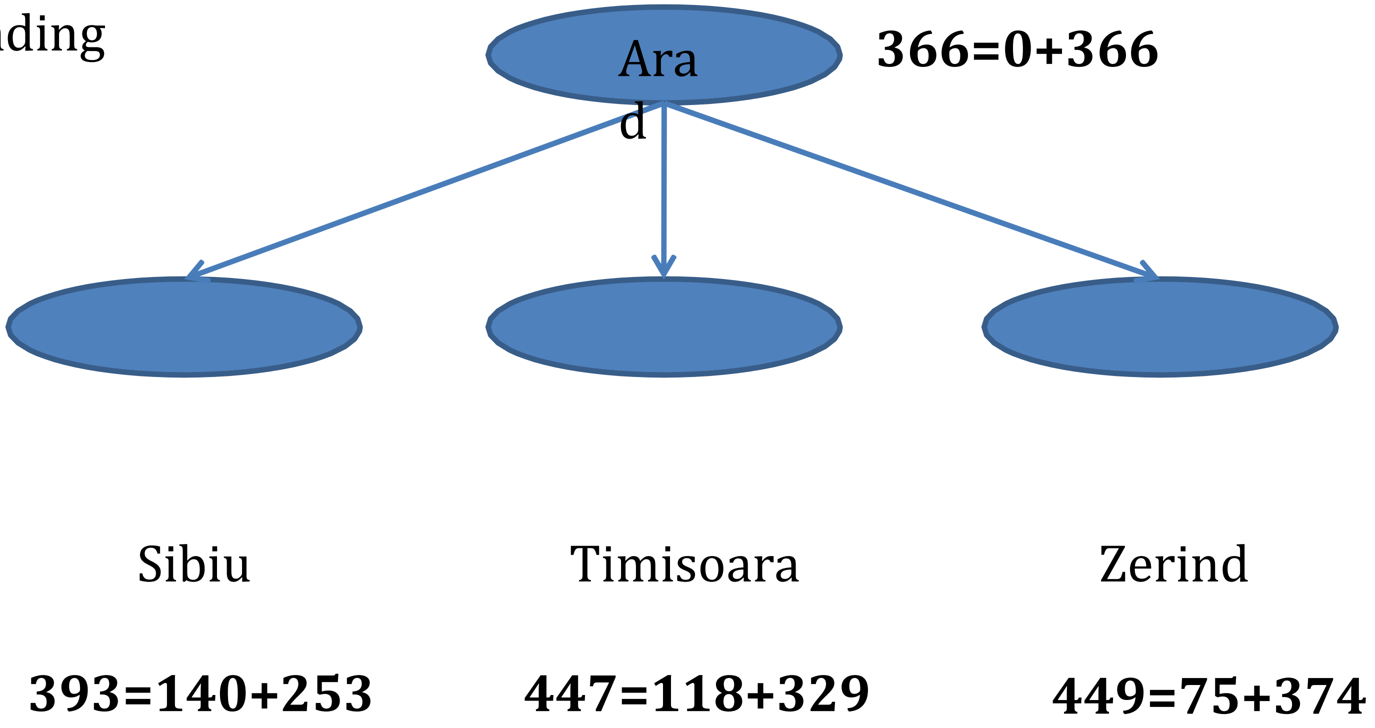
Note: *In this chapter we will consider one example that "A map is given with different cities connected and their distance values are also mentioned. Agent starts from one city and reach to other."*


Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
Drobeta
120
138
Craiova
Neamt
87
Iasi
92
Vaslui
142
98
Hirsova
Urziceni
85
Bucharest
86
90
Giurgiu
Eforie

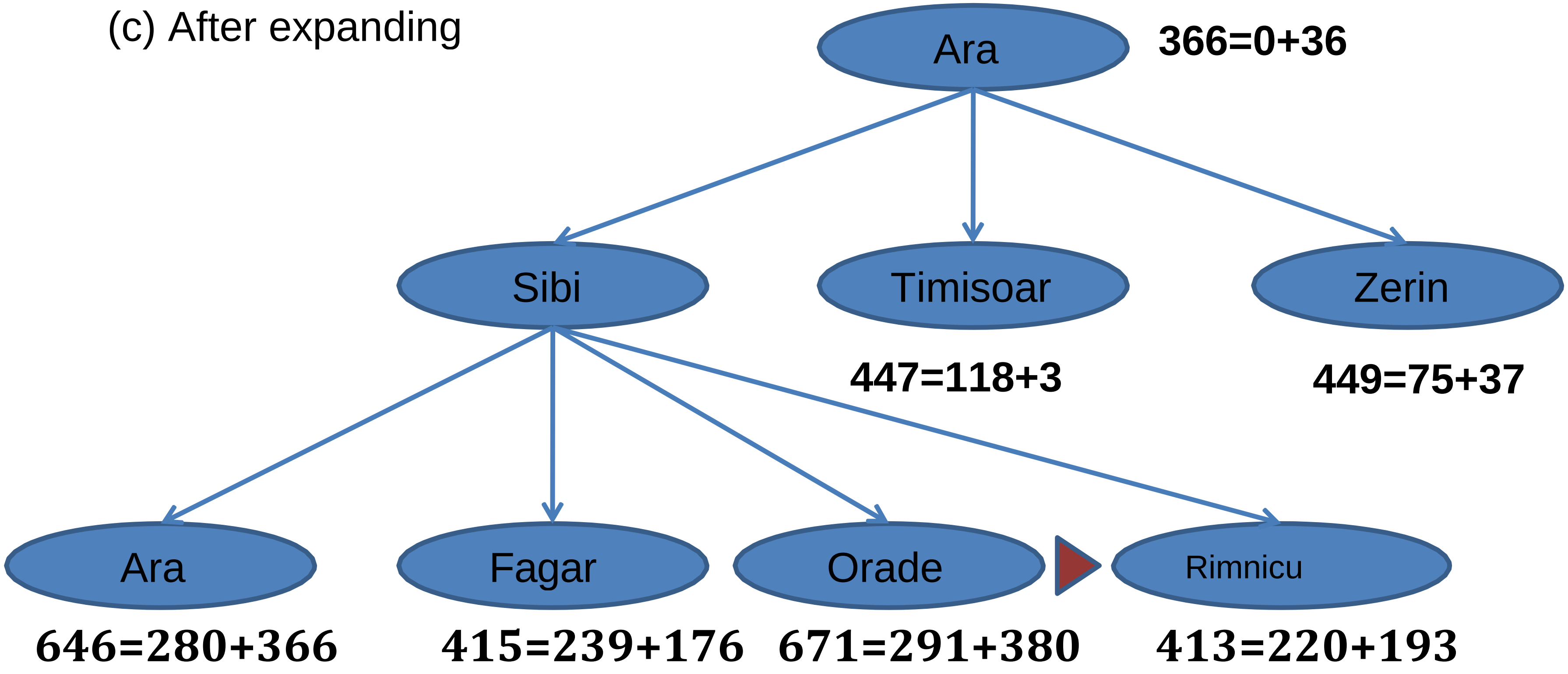
(a) Initial State



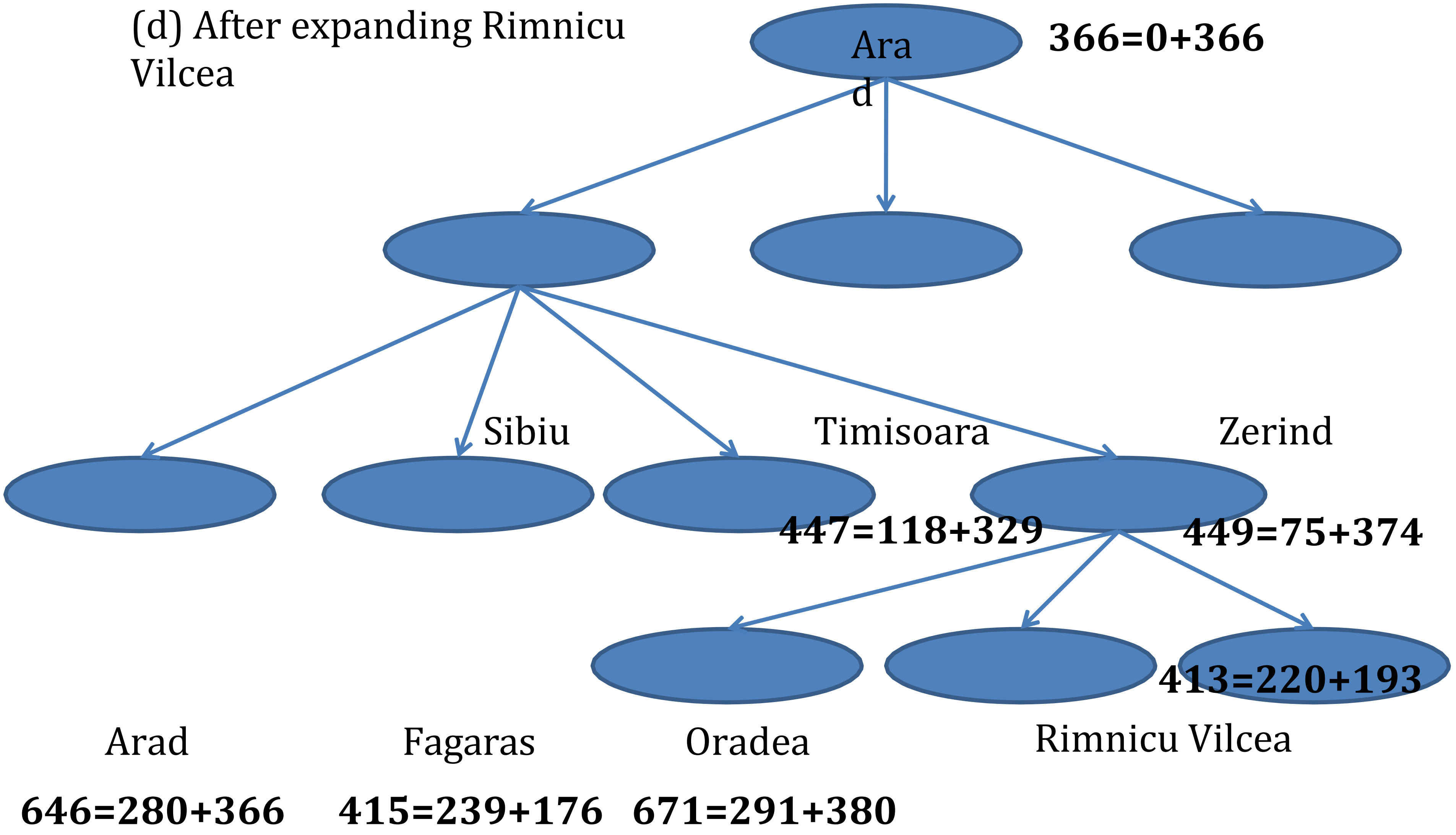
(b) After expanding Arad



(c) After expanding



(d) After expanding Rimnicu Vilcea

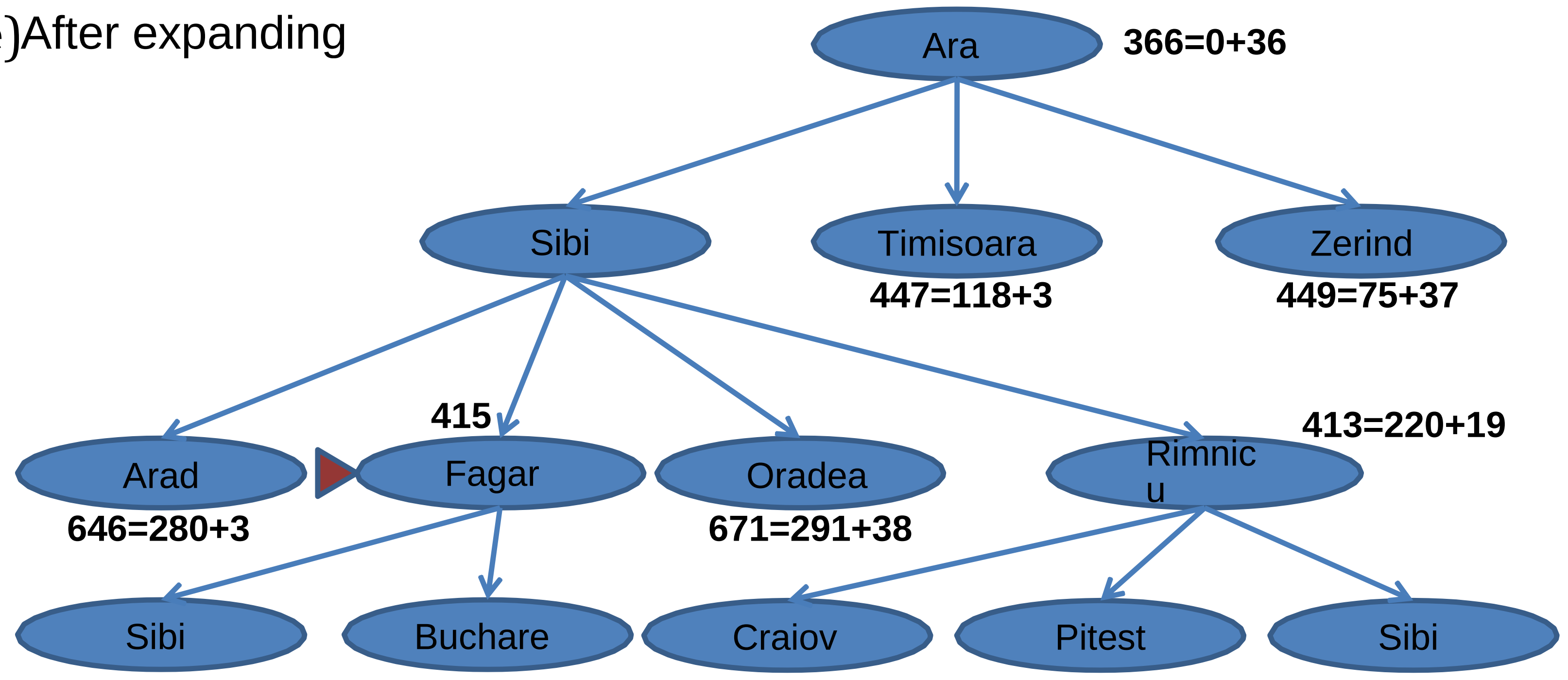


Craiova Pitesti Sibiu

526=366+160 **417=317+100**

553=300+253

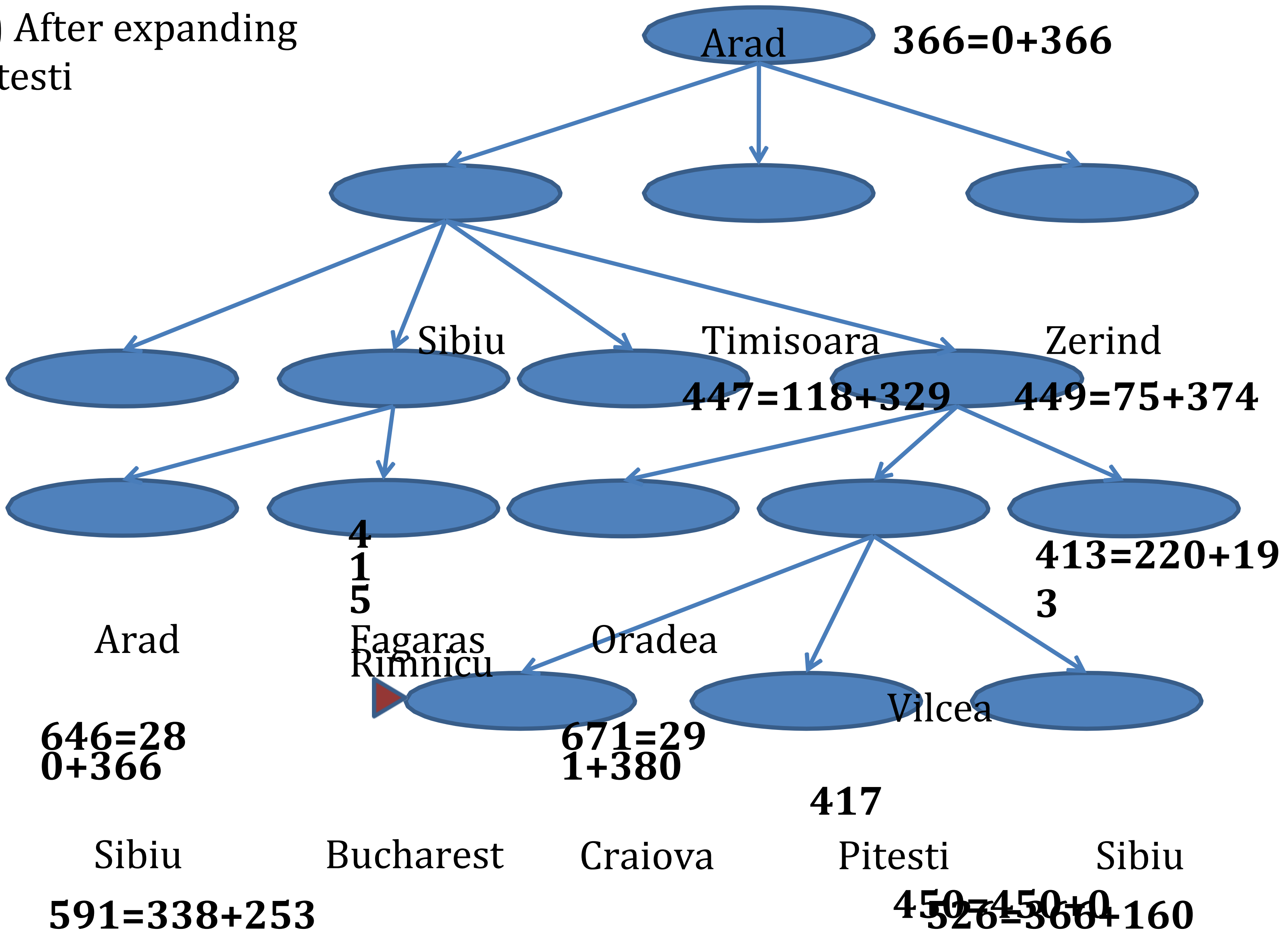
(e)After expanding



591=338+253 **450=450+0** **526=366+160**

417=317+100 **553=300+253**

(f) After expanding Pitesti



553=300+253

Bucharest

Craiova

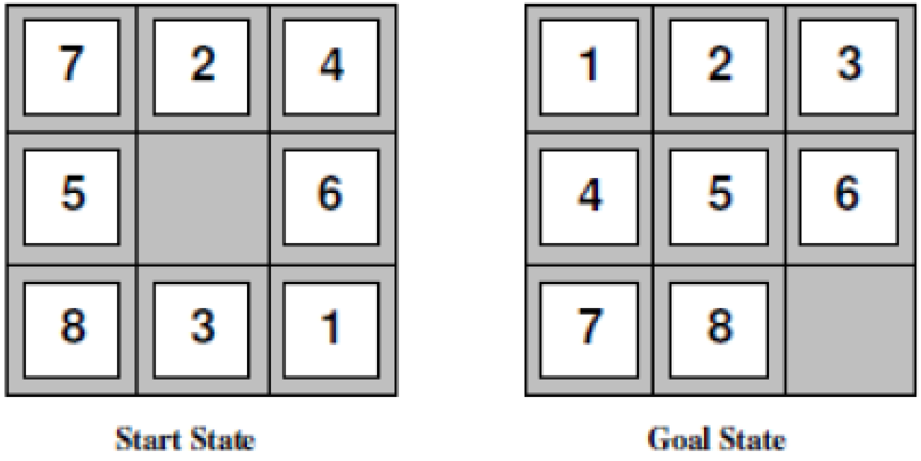
R
imni
cu
Vilc
ea

418=418+0 **615=455+160** **607=414+193**

Heuristic Function:

- *“A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood.”*
  - ➢ h(n) = estimated cost of the cheapest path from node n to goal node.
  - ➢ If n is goal then h(n)=0

Heuristic Function:



Admissible heur

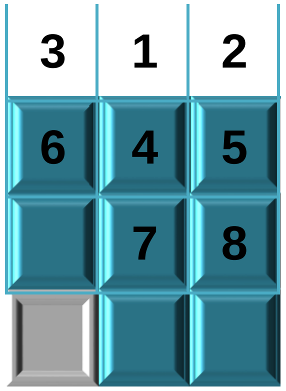
h1(n) = number

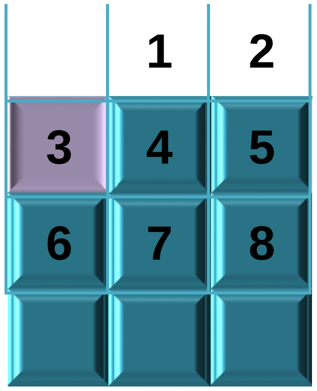
h2(n) = total Ma desired location of each tile)

In the example

- h1(S) = 6
- h2(S) = 2 + 0 + 3 + 1 + 0 + 1 + 3 + 4 = 14
- If h2 dominates h1, then h2 is better for search than h1.

Heuristic Function Example(8-Puzzle)

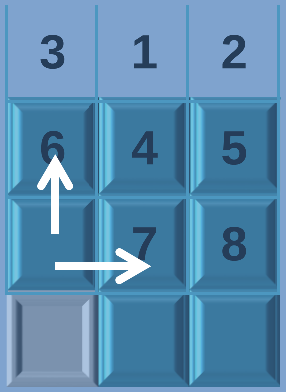




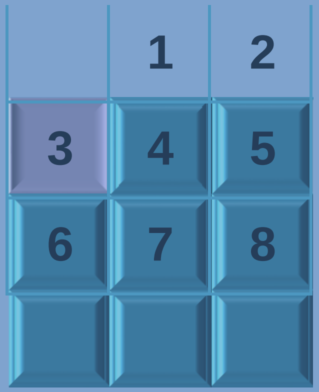
Initial State h(n) =2

Goal State

Heuristic Function Example(8-Puzzle)

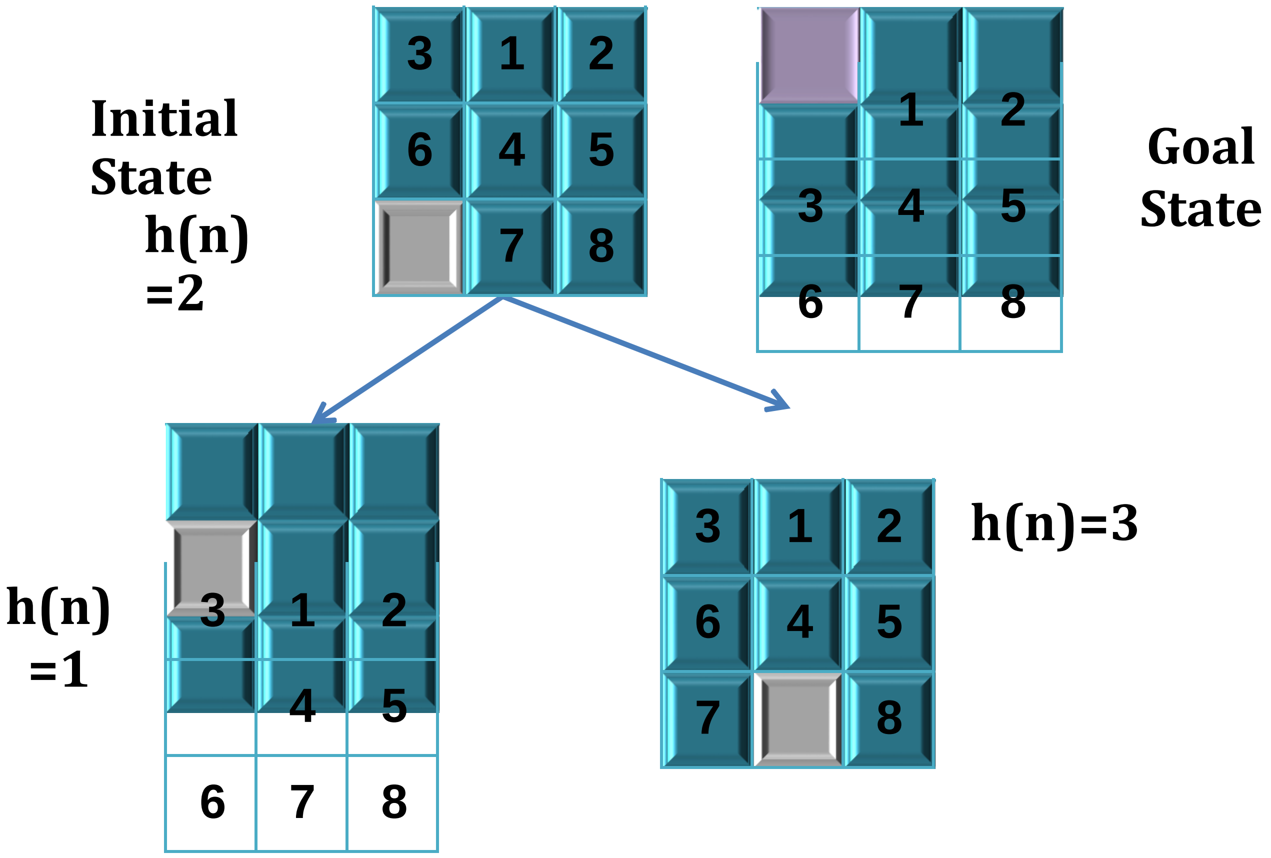


Initial State h(n) =2

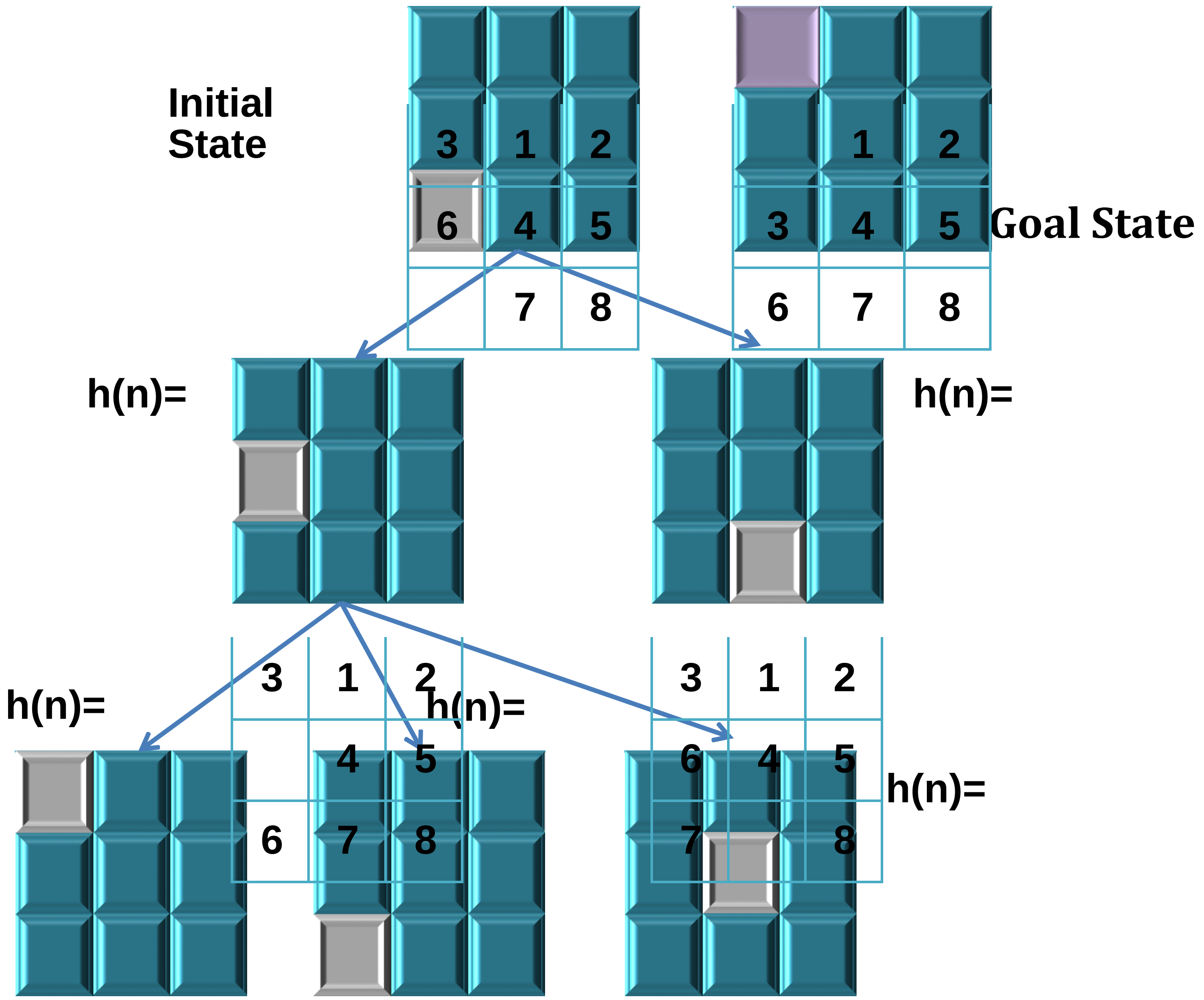


Goal State

Heuristic Function Example(8-



Heuristic Function Example(8-



Heuristic Function Example(8-

	1	2
3	4	5
6	7	8

3	1	2
6	4	5
	7	8

3	1	2
4		5
6	7	8

Optimization Problems

Optimization Problems

Optimization problem

Classical Method

- The search algorithms that we have studied so far are based on classical search techniques, which is a systematic way.
- In this search methods one or more paths are kept in memory, follow and record alternatives to explore points along the path.
- Solution not only include goal but also path that constitutes a solution to given problem.

Non-Classical/Local search

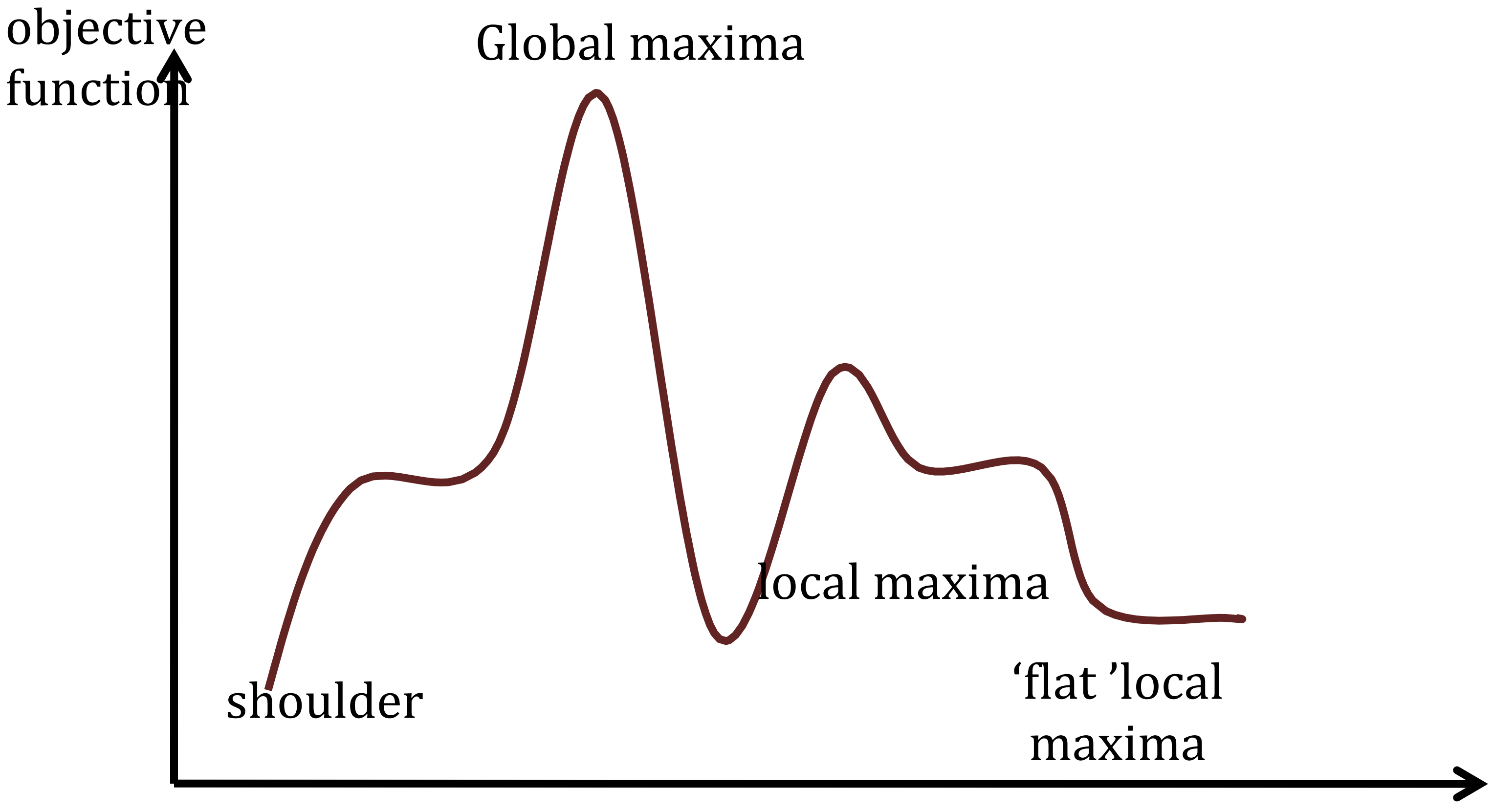
- What if the path to the goal is irrelevant?
- For example, in the 8-queens problem, most important part is the final configuration of queens, not the sequence or order in which they are added. Many applications follow such method like integrated-circuit design, Recommendation systems.
- If path to goal does not matter, we might need to consider a different class if algorithm, which do not worry about path.

Optimization Problems

Hill Climbing

- **Hill climbing** is a mathematical optimization technique which belongs to the family of local search.
- It is an iterative algorithm that starts with an arbitrary node or point to a problem, then attempts to find a better solution by incrementally changing a single element of the solution.
- If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.

Hill


objective function
Global maxima
local maxima
shoulder
'flat 'local maxima

Hill

State space

Hill climbing

```
function hillClimbing(problem)
      current =
      problem.INITIAL_STATE
      loop
          neighbor = highest-value successor of
          current If(neighbor.value<=current.value)
          then
            return current.state
          current = neighbor
```

Problems in Hill climbing (Drawbacks)

Local maxima:

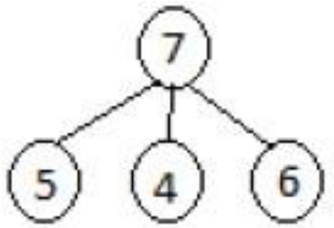
- Local maxima = no uphill step
- Allow “random restart” which is complete, but might take a very long time.

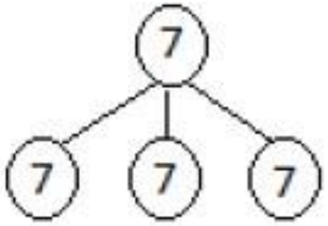
Ridges:

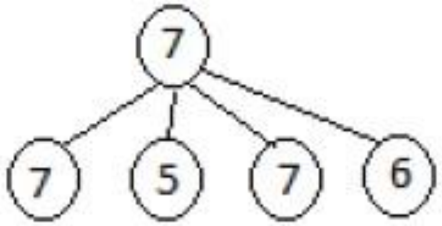
- A "ridge" which is an area in the search that is higher than the surrounding areas, but cannot be searched in a simple move.

Plateau:

- All steps equal (flat or shoulder)
- A plateau is encountered when the search space is flat, or sufficiently flat that the value returned by the target function is indistinguishable from the value returned for nearby regions due to the precision used by the machine to represent its value.


(a) Local maxima


(b) Platue


(c) Ridge

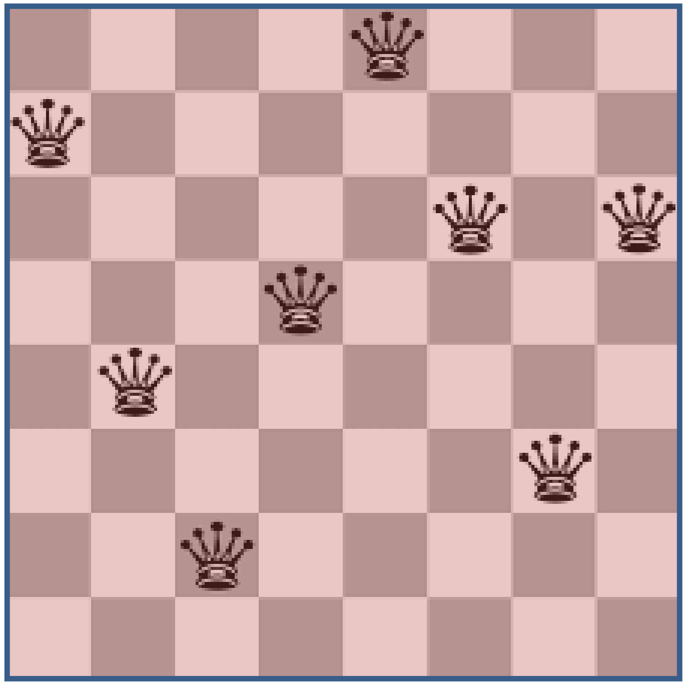
a). Successor of current state are heuristically worst than the current state.
b).Successor of current state are heuristically equivalent to the current state.
c).Successor of current state are heuristically worst or heuristically equivalent than the current state.

Prof. D.S.Kale

Examples: Hill Climbing

8-Queen

h = number of pairs of queens that are attacking each other



A local minimum of h in the 8-queens

A state with h=17 and the h-value for

Prof. Suresh Mestry

Simulated Annealing

- **Annealing** is a process in metallurgy where metals are slowly cooled to make them reach a state of low energy where they are very strong.
- Simulated annealing is an analogous method for optimization.
- Simulated annealing is a process where the temperature is reduced slowly, starting from a random search at high temperature eventually becoming pure greedy descent as it approaches zero temperature.
- **Simulated annealing** maintains a current assignment of values to variables.
- At each step, it picks a variable at random, then picks a value

  at random.

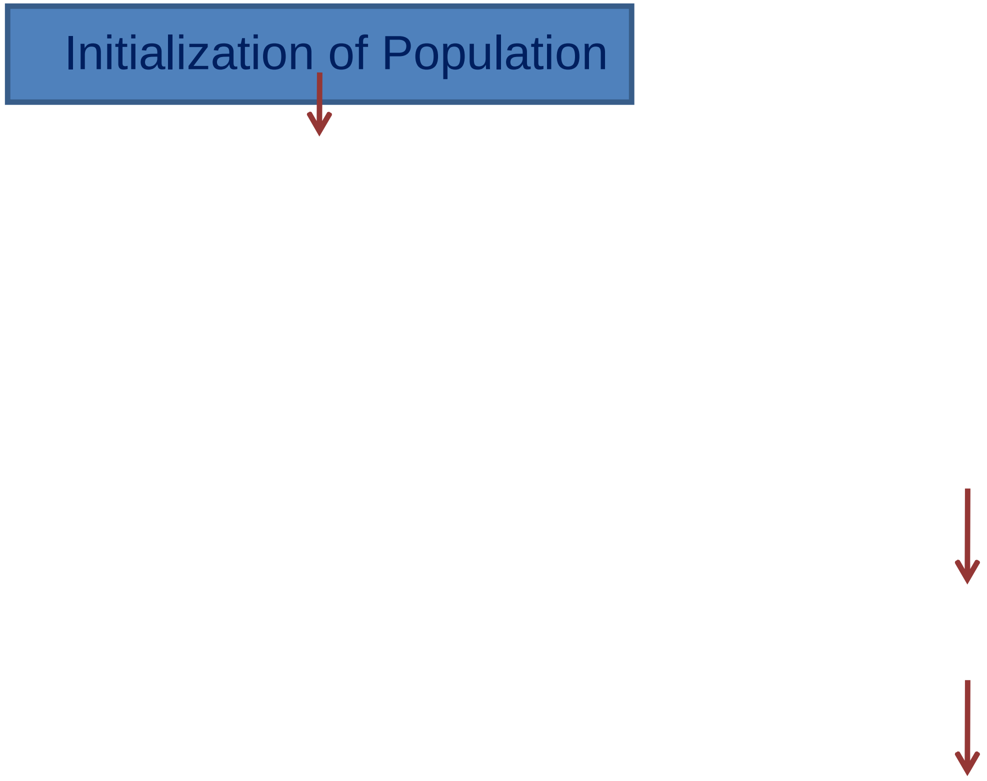
Genetic Algorithm

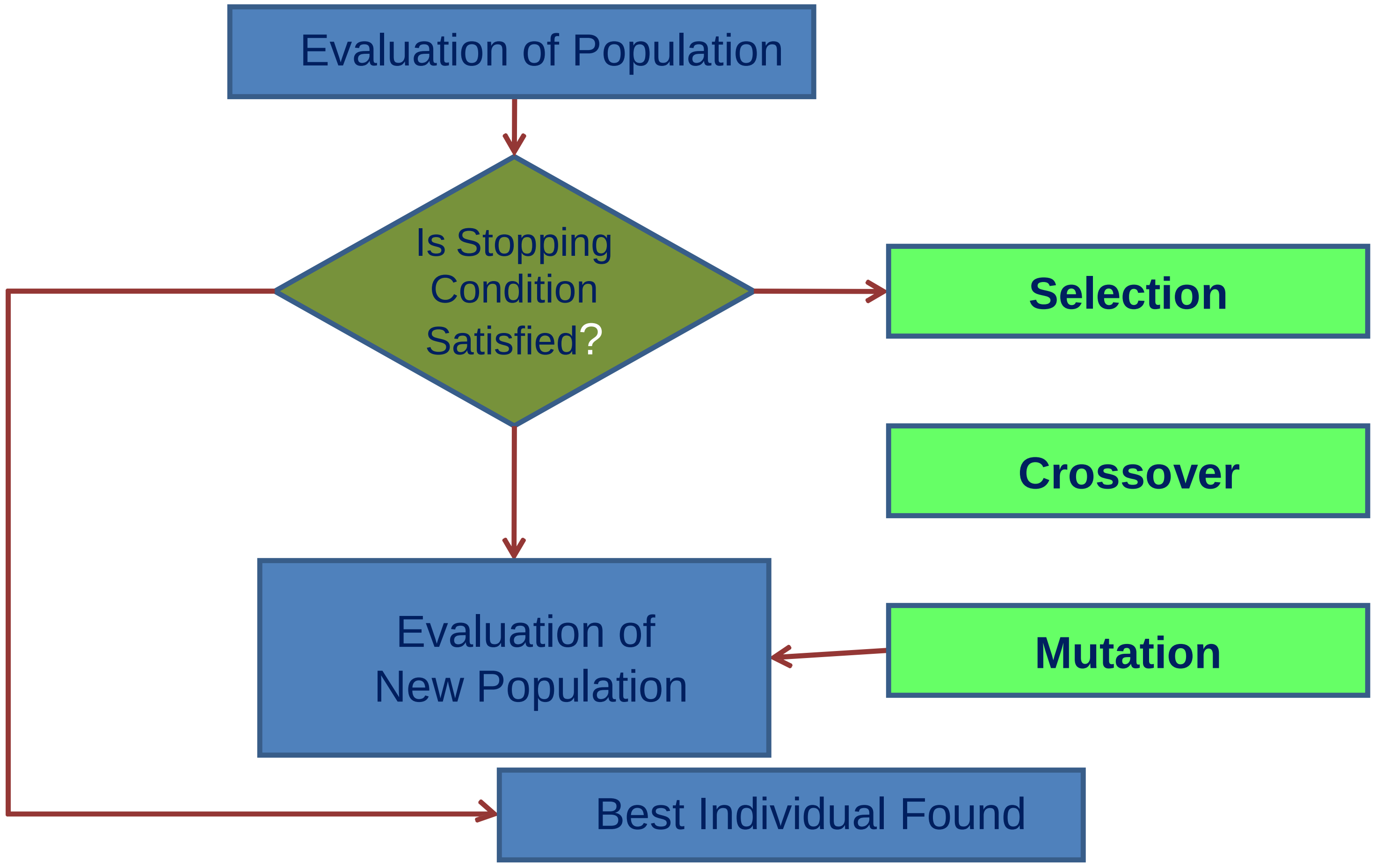
- A genetic algorithm conceptually follows steps inspired by the biological processes of evolution.
- Genetic Algorithms follow the idea of SURVIVAL OF THE FITTEST- Better and better solutions evolve from previous generations until a near optimal solution is obtained.
- Also known as evolutionary algorithms.
- Genetic algorithms demonstrate self organization and adaptation similar to the way that the fittest biological organism survive and reproduce.

Genetic Algorithm

- A genetic algorithm is an iterative procedure that represents its candidate solutions as strings of genes called chromosomes.
- Generally applied to spaces which are too large
- Genetic Algorithms are often used to improve the performance of other AI methods such as expert systems or neural networks.
- The method learns by producing offspring that are better and better as measured by a fitness function, which is a measure of the objective to be obtained (maximum or minimum).

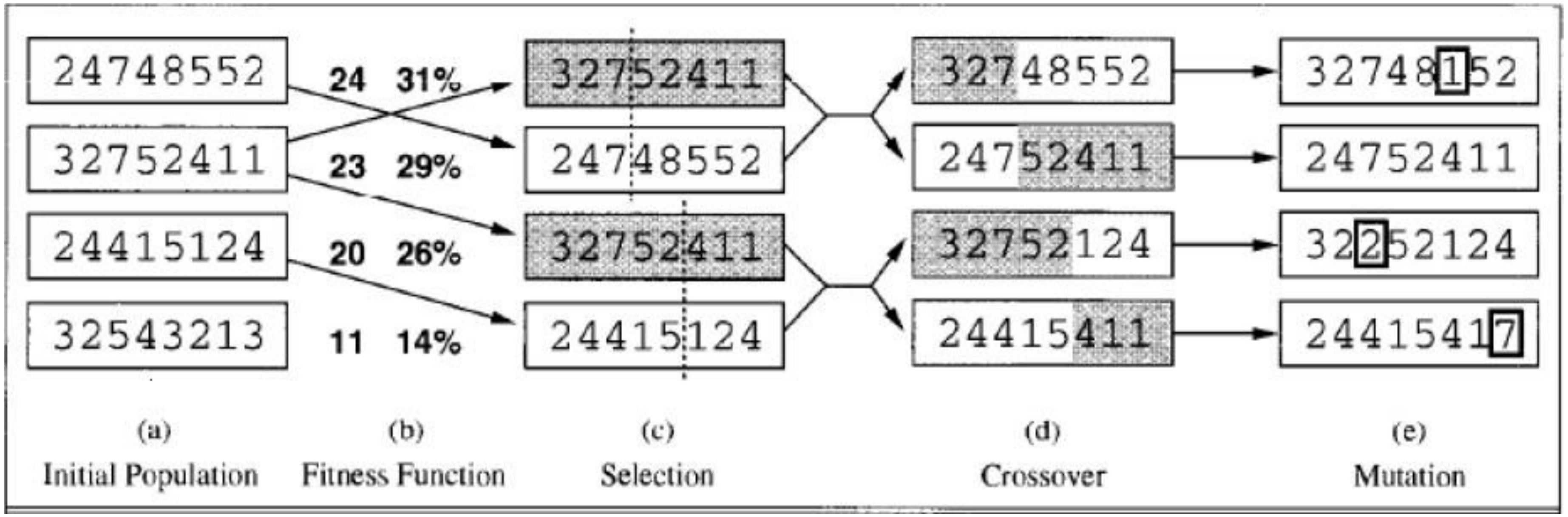
Steps in Genetic Algorithm





Prof. Suresh Mestry

Example


24748552
32752411
24415124
32543213
24 31%
23 29%
20 26%
11 14%
32752411
24748552
32752411
24415124
32748552
24752411
32752124
24415411
32748152
24752411
32252124
24415417
(a) Initial Population
(b) Fitness Function
(c) Selection
(d) Crossover
(e) Mutation

Concepts

- Population: set of individuals each representing a possible solution to a given problem.
- Gene: a solution to problem represented as a set of parameters ,these parameters known as genes.
- Chromosome:genes joined together to form a string of values called chromosome.
- Fitness score(value):every chromosome has fitnessscore can be inferred from the chromosome itself by using fitness function.

Stochastic operators

- Selection replicates the most successful solutions found ina population at a rate proportional to their relative quality
- Recombination (Crossover) decomposes two distinct solutions and then randomly mixes their parts to form novel solutions
- Mutation randomly perturbs a candidate solution

Suppose a Genetic Algorithm uses chromosomes of the form x=abcdefgh with a fixed length of eight genes. Each gene can be any digit between 0 and 9. Let the fitness of individual x be calculated as :

$$f(x) = (a+b)-(c+d)+(e+f)-(g+h)$$

Solution: Let the initial population consist of four individuals x1,... ,x4 with the following chromosomes :

X1 = 6 5 4 1 3 5 3 2

F(x1) = (6+5)-(4+1)+(3+5)-(3+2) = 9

X2 = 8 7 1 2 6 6 0 1

F(x2) = (8+7)-(1+2)+(6+6)-(0+1) = 23

X3 = 2 3 9 2 1 2 8 5

F(x3) = (2+3)-(9+2)+(1+2)-(8+5) = -16

X4= 4 1 8 5 2 0 9 4

F(x4) = (4+1)-(8+5)+(2+0)-(9+4) = -19

The arrangement is (assume maximization) x2 x1
x3 x4

(the fittest individual) (least fit individual)

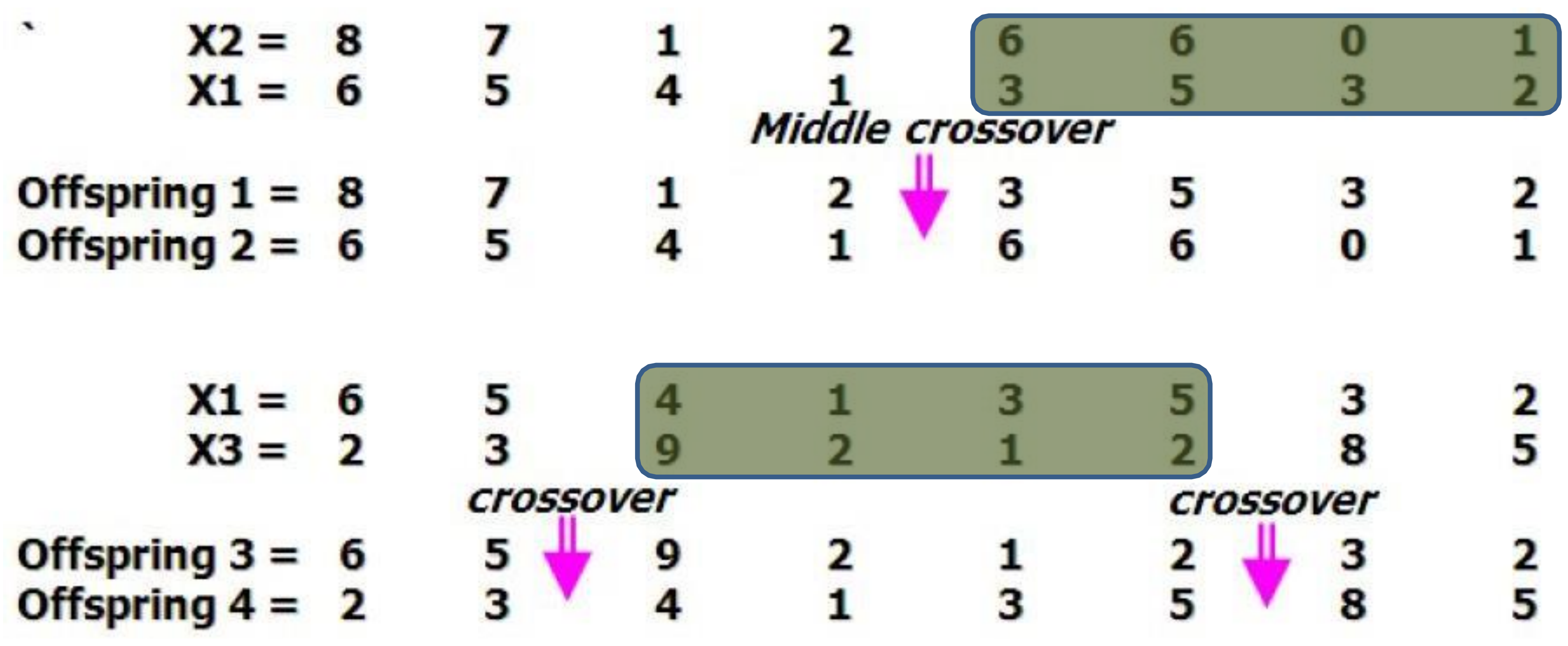
Put the calculations in table for simplicity

Individuals	**String Representation**	**Fitness**	**Arrangement Assume maximization**
X1	65413532	9	X2(fittest individual)
X2	87126601	23	X1(second fittest individual)
X3	23921285	-16	X3 (third fittest individual)
X4	41852094	-19	X4 (least fit individual)

Average fitness = (9+23+ -16 + -19)/ 4 =-0.75

So Average fitness: -0.75 (Best: 23 Worst: -19)

Next step is to apply crossover operation



X2 =	8	7	1	2	6	6	0	1
X1 =	6	5	4	1	3	5	3	2
Offspring 1 =	8	7	1	2	3	5	3	2
Offspring 2 =	6	5	4	1	6	6	0	1
X1 =	6	5	4	1	3	5	3	2
X3 =	2	3	9	2	1	2	8	5
Offspring 3 =	6	5	9	2	1	2	3	2
Offspring 4 =	2	3	4	1	3	5	8	5

Calculating fitness function of offspring.

Offspring 1 = 8 7 1 2 3 5 3 2

F (Offspring 1) =(8+7)-(1+2)+(3+5)-(3+2) = **15**

Offspring 2 = 6 5 4 1 6 6 0 1

F (Offspring 2) =(6+5)-(4+1)+(6+6)-(0+1) = **17**

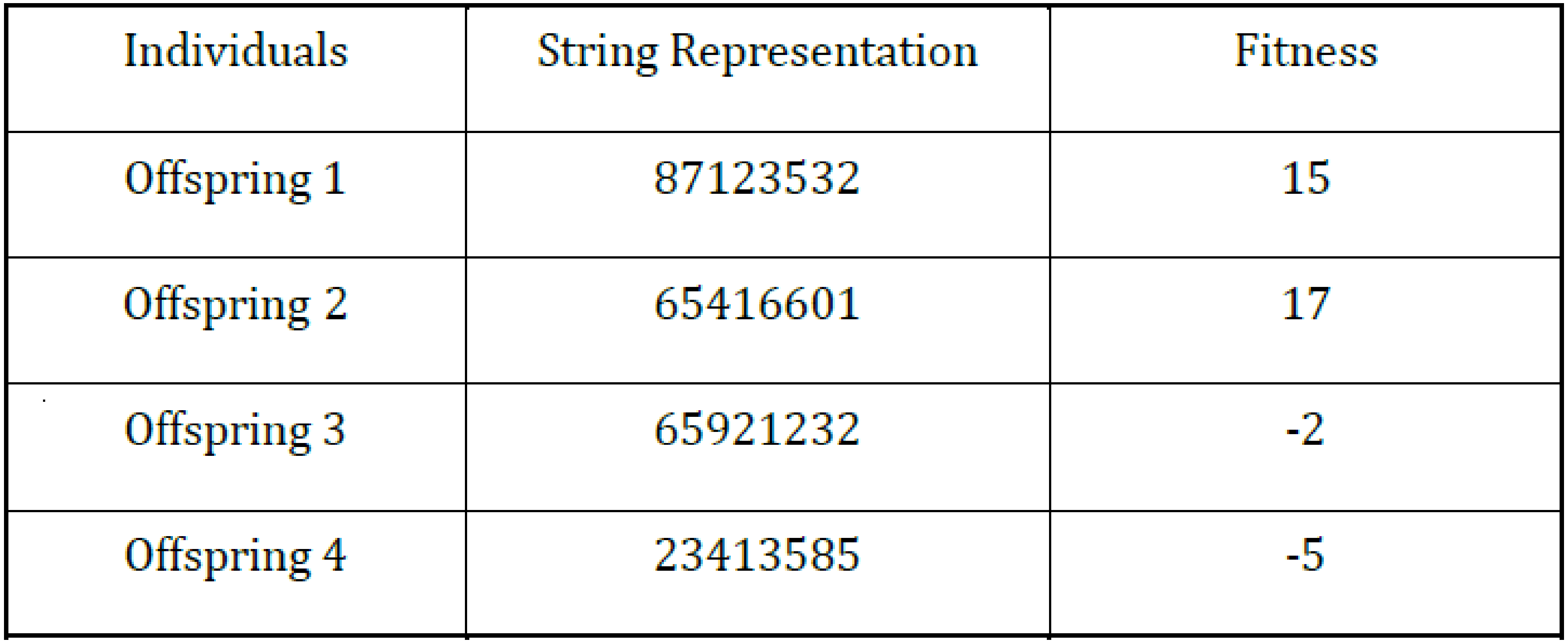
Offspring 3 = 6 5 9 2 1 2 3 2

F (Offspring 3) =(6+5)-(9+2)+(1+2)-(3+2) = **-2**

Offspring 4 = 2 3 4 1 3 5 8 5

F (Offspring 4) =(2+3)-(4+1)+(3+5)-(8+5) = **-5**

Put the calculation in table for simplicity



Individuals	String Representation	Fitness
Offspring 1	87123532	15
Offspring 2	65416601	17
Offspring 3	65921232	-2
Offspring 4	23413585	-5

- Average fitness: 6.25 (Best: 17 Worst: -5)
- So that, the overall fitness is improved, since the average is better and worst is improved.
  Average fitness = (15+17+ -5 + -2)/ 4 = 6.25

Prof. D.S.Kale

3.3 Adversarial search in Artifical Intelligence

AI Adversarial search: Adversarial search is a **game-playing** technique where the agents are surrounded by a competitive environment. A conflicting goal is given to the agents (multiagent). These agents compete with one another and try to defeat one another in order to win the game. Such conflicting goals give rise to the adversarial search. Here, game-playing means discussing those games where **human intelligence** and **logic factor** is used, excluding other factors such as **luck factor**. **Tic-tac-toe, chess, checkers**, etc., are such type of games where no luck factor works, only mind works.
Mathematically, this search is based on the concept of **'Game Theory.'** ***According to game theory, a game is played between two players. To complete the game, one has to win the game and the other looses automatically.'***

We are opponents- I win, you loose.

Techniques required to get the best optimal solution

There is always a need to choose those algorithms which provide the best optimal solution in a limited time. So, we use the following techniques which could fulfill our requirements:

- **Pruning:** A technique which allows ignoring the unwanted portions of a search tree which make no difference in its final result.
- **Heuristic Evaluation Function:** It allows to approximate the cost value
- at each level of the search tree, before reaching the goal node.

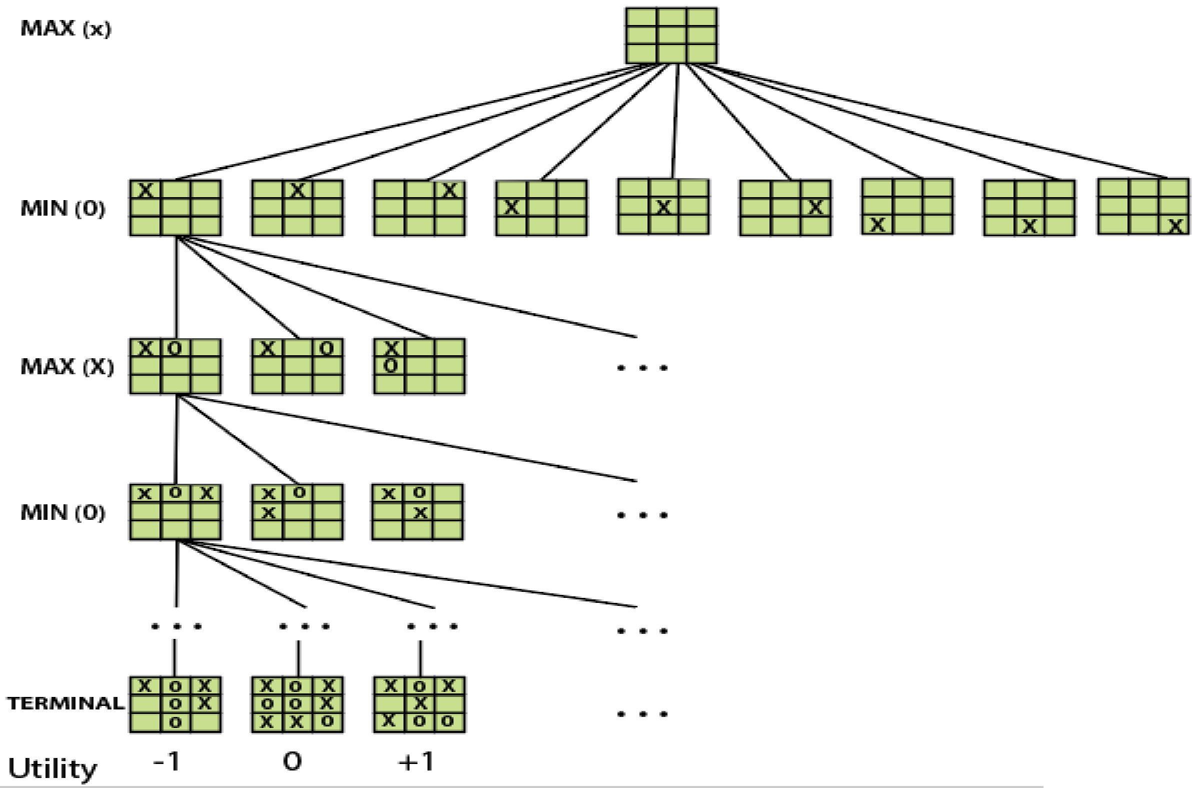
Elements of Game Playing search

To play a game, we use a game tree to know all the possible choices and to pick the best one out. There are following elements of a game-playing:

- **S_0:** It is the initial state from where a game begins.

- **PLAYER (s):** It defines which player is having the current turn to make a move in the state.
- **ACTIONS (s):** It defines the set of legal moves to be used in a state.
- **RESULT (s, a):** It is a transition model which defines the result of a move.
- **TERMINAL-TEST (s)**: It defines that the game has ended and returns true.
- **UTILITY (s,p):** It defines the final value with which the game has ended. This function is also known as **Objective function** or **Payoff function**. The price which the winner will get i.e.
- **(-1):** If the PLAYER loses.
- **(+1):** If the PLAYER wins.
- **(0):** If there is a draw between the PLAYERS.

For example**, in **chess, tic-tac-toe,** we have two or three possible outcomes. Either to win, to lose, or to draw the match with values **+1,-1 or 0.
Let's understand the working of the elements with the help of a game tree designed for **tic-tac-toe**. Here, the *node represents the game state and edges represent the moves taken by the players.*



A game-tree for tic-tac-toe

- **INITIAL STATE (S_0):** The top node in the game-tree represents the initial state in the tree and shows all the possible choice to pick out one.
- **PLAYER (s):** There are two players, **MAX and MIN**. **MAX** begins the game by picking one best move and place **X** in the empty square box.
- **ACTIONS (s):** Both the players can make moves in the empty boxes chance by chance.

- **RESULT (s, a):** The moves made by **MIN** and **MAX** will decide the outcome of the game.
- **TERMINAL-TEST(s):** When all the empty boxes will be filled, it will be the terminating state of the game.
- **UTILITY:** At the end, we will get to know who wins: **MAX** or **MIN,** and accordingly, the price will be given to them**.**

Types of algorithms in Adversarial search

In a **normal search**, we follow a sequence of actions to reach the goal or to finish the game optimally. But in an **adversarial search**, the result depends on the players which will decide the result of the game. It is also obvious that the solution for the goal state will be an optimal solution because the player will try to win the game with the shortest path and under limited time.
There are following types of adversarial search:

- **Minmax Algorithm**
- **Alpha-beta Pruning**

MIN MAX STRATEGY

In artificial intelligence, minimax is a **decision-making** strategy under **game theory,** which is used to minimize the losing chances in a game and to maximize the winning chances. This strategy is also known as ‘**Minmax,**’ ’**MM,**’ **or** ‘**Saddle point.**’ Basically, it is a two-player game strategy where *if one wins, the other loose the game*. This strategy simulates those games that we play in our day-to-day life. Like, if two persons are playing chess, the result will be in favor of one player and will unfavor the other one. The person who will make his bes*t try,efforts as well as cleverness, will surely win.*

We can easily understand this strategy via **game tree**- where the *nodes represent the states of the game and edges represent the moves made by the players in the game*. Players will be two namely:

- **MIN:** Decrease the chances of **MAX** to win the game.
- **MAX:** Increases his chances of winning the game.

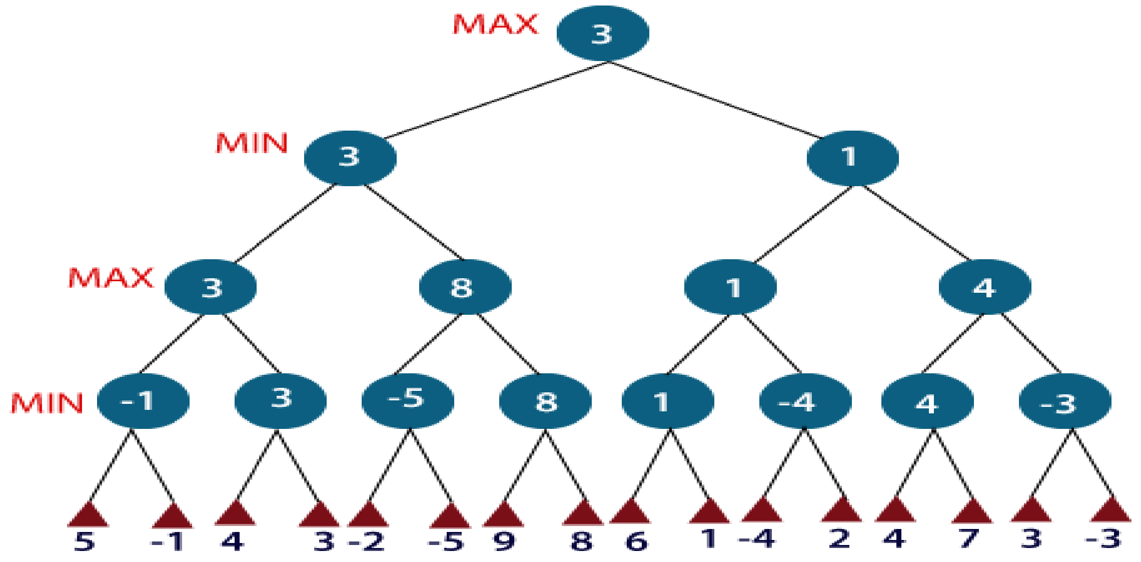
They both play the game alternatively, i.e., turn by turn and following the above strategy, i.e., if one wins, the other will definitely lose it. Both players look at one another as competitors and will try to defeat one-another, giving their best.

In minimax strategy, the result of the game or the utility value is generated by a **heuristic function** by propagating from the initial node to the root node. It follows the **backtracking technique** and backtracks to find the best choice. MAX will choose that path which will increase its utility value and MIN will choose the opposite path which could help it to minimize MAX’s utility value.

MINIMAX Algorithm

MINIMAX algorithm is a backtracking algorithm where it backtracks to pick the best move out of several choices. MINIMAX strategy follows the **DFS (Depth-first search)** concept. Here, we have two players **MIN and MAX,** and the game is played alternatively between them, i.e., when **MAX** made a move, then the next turn is of **MIN.** It means the move made by MAX is fixed and, he cannot change it. The same concept is followed in DFS strategy, i.e., we follow the same path and cannot change in the middle. That's why in MINIMAX algorithm, instead of BFS, we follow DFS.

- Keep on generating the game tree/ search tree till a limit **d.**
- Compute the move using a heuristic function.
- Propagate the values from the leaf node till the current position following the minimax strategy.
- Make the best move from the choices.



For example, in the above figure, the two players **MAX** and **MIN** are there. **MAX** starts the game by choosing one path and propagating all the nodes of that path. Now, **MAX** will backtrack to the initial node and choose the best path where his utility value will be the maximum. After this, its **MIN** chance. **MIN** will also propagate through a path and again will backtrack, but **MIN** will choose the path which could minimize **MAX** winning chances or the utility value.

So, if the level is minimizing, the node will accept the minimum value from the successor nodes. If the level is maximizing, the node will accept the maximum value from the successor.

Note: The time complexity of MINIMAX algorithm is **$O(b^d)$** where b is the branching factor and d is the depth of the search tree.

Alpha-beta pruning in Artificial Intelligence

Alpha-beta pruning is an advance version of MINIMAX algorithm. The drawback of minimax strategy is that it explores each node in the tree deeply to provide the best path among all the paths. This increases its time complexity. But as we know, the performance measure is the first consideration for any optimal algorithm. Therefore, alpha-beta pruning reduces this drawback of minimax strategy by less exploring the nodes of the search tree.
The method used in alpha-beta pruning is that it **cutoff the search** by exploring less number of nodes. It makes the same moves as a minimax algorithm does, but it prunes the unwanted branches using the pruning technique (discussed in adversarial search). Alpha-beta pruning works on two threshold values, i.e., **? (alpha)** and **? (beta).**

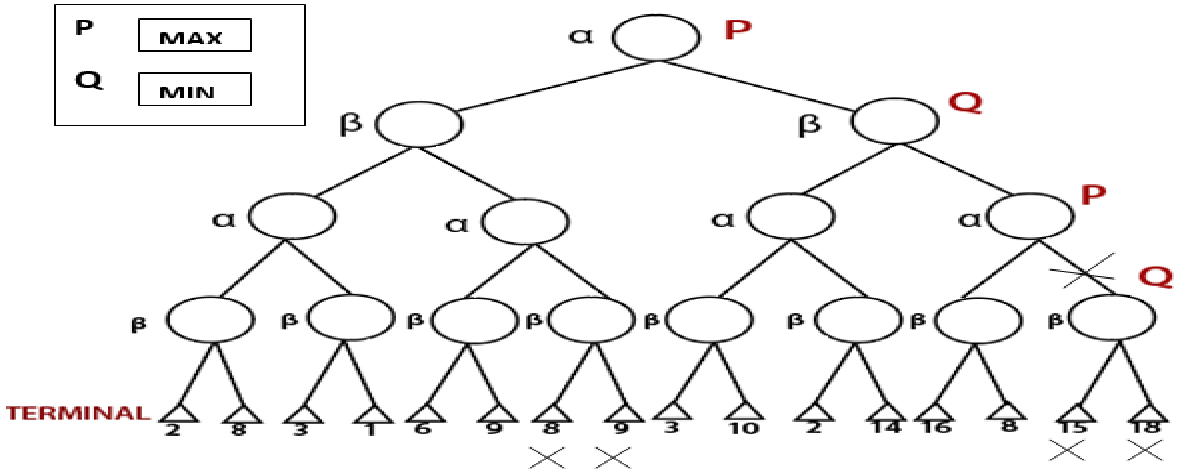
- **?:** It is the best highest value, a **MAX** player can have. It is the lower bound, which represents negative infinity value.
-
- **?:** It is the best lowest value, a **MIN** player can have. It is the upper bound which represents positive infinity.

So, each MAX node has ?-value, which never decreases, and each MIN node has ?-value, which never increases.

Note: Alpha-beta pruning technique can be applied to trees of any depth, and it is possible to prune the entire subtrees easily.

Working of Alpha-beta Pruning

Consider the below example of a game tree where **P** and **Q** are two players. The game will be played alternatively, i.e., chance by chance. Let, **P** be the player who will try to win the game by maximizing its winning chances. **Q** is the player who will try to minimize **P's** winning chances. Here, **?** will represent the maximum value of the nodes, which will be the value for **P** as well. **?** will represent the minimum value of the nodes, which will be the value of **Q**.



Alpha-beta pruning

- Any one player will start the game. Following the DFS order, the player will choose one path and will reach to its depth, i.e., where he will find the **TERMINAL** value.
- If the game is started by player P, he will choose the maximum value in order to increase its winning chances with maximum utility value.
- If the game is started by player Q, he will choose the minimum value in order to decrease the winning chances of A with the best possible minimum utility value.
- Both will play the game alternatively.
- The game will be started from the last level of the game tree, and the value will be chosen accordingly.
- Like in the below figure, the game is started by player Q. He will pick the leftmost value of the TERMINAL and fix it for beta (?). Now, the next TERMINAL value will be compared with the ?-value. If the value will be smaller than or equal to the ?-value, replace it with the current ?-value otherwise no need to replace the value.
- After completing one part, move the achieved ?-value to its upper node and fix it for the other threshold value, i.e., ?.
- Now, its P turn, he will pick the best maximum value. P will move to explore the next part only after comparing the values with the current ?-value. If the

value is equal or greater than the current ?-value, then only it will be replaced otherwise we will prune the values.

- The steps will be repeated unless the result is not obtained.
- So, number of pruned nodes in the above example are **four** and MAX wins the game with the maximum **UTILITY** value, i.e.,**3**

The rule which will be followed is: **"Explore nodes if necessary otherwise prune the unnecessary nodes."**

Note: It is obvious that the result will have the same **UTILITY** value that we may get from the MINIMAX strategy.

Mini-Max Algorithm in Artificial Intelligence

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.

- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Pseudo-code for MinMax Algorithm:

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
maxEva= -infinity
 for each child of node do
 eva= minimax(child, depth-1, false)
maxEva= max(maxEva,eva)       //gives Maximum of the values
    return maxEva

```

```
2.     else                    // for Minimizer player
3.      minEva= +infinity
4.      for each child of node do
5.      eva= minimax(child, depth-1, true)
6.      minEva= min(minEva, eva)        //gives minimum of the values
7.      return minEva
```

Initial call:

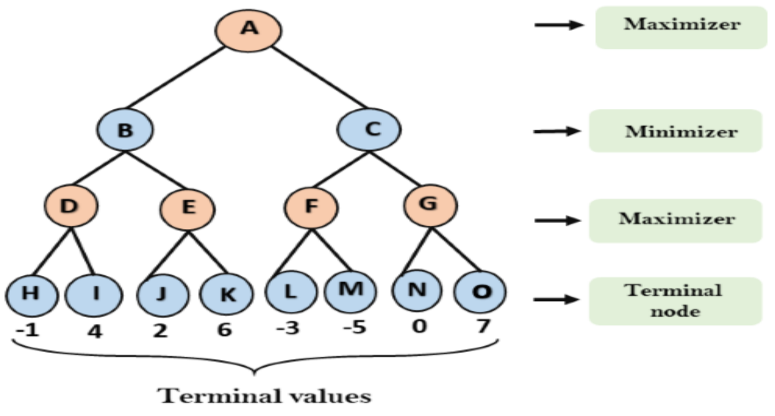
Minimax(node, 3, true)

Working of Min-Max Algorithm:

- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.

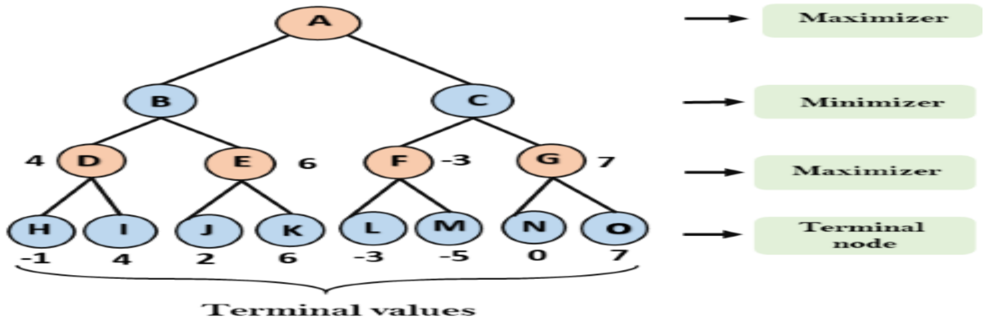
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



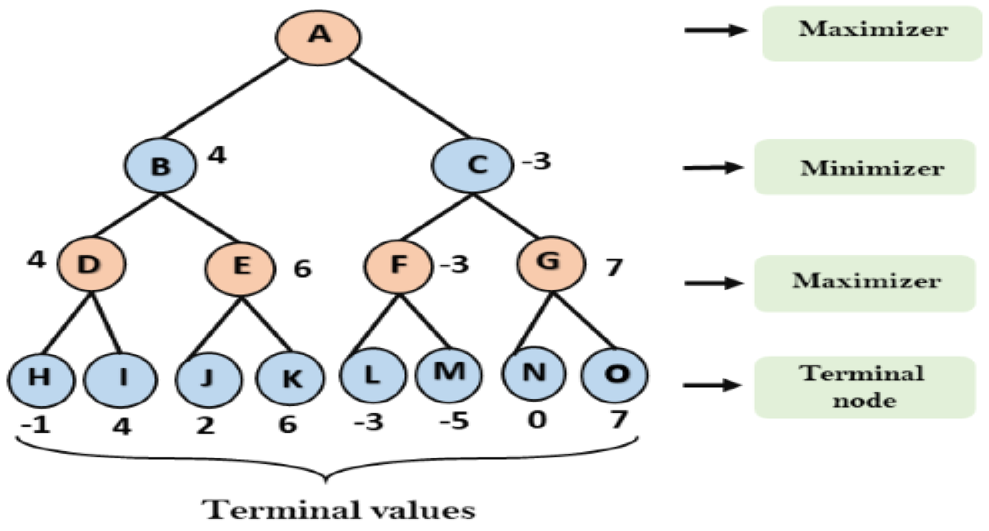
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7



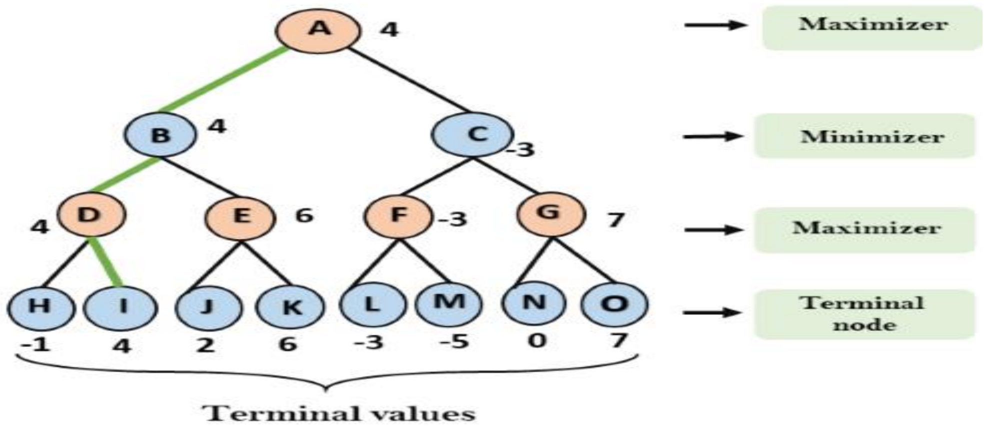
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3



Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

- For node A max(4, -3)= 4



That was the complete workflow of the minimax two player game.

Properties of Mini-Max algorithm:

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

Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from **alpha-beta pruning**

Example on Alpha Beta Pruning and Pseudo-code Alpha-Beta Pruning

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

a. **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.

b. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.

- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Condition for Alpha-beta pruning:

The main condition which required for alpha-beta pruning is:

. $\alpha >= \beta$

Key points about alpha-beta pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

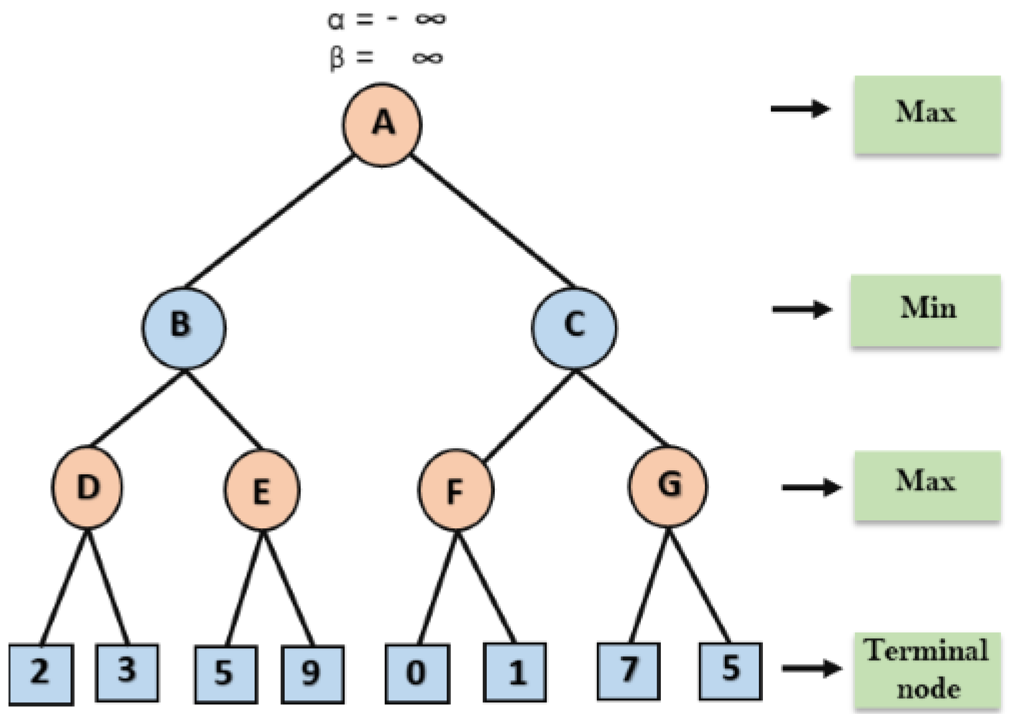
Pseudo-code for Alpha-beta Pruning:

```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
   maxEva= -infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, False)
  maxEva= max(maxEva, eva)
        alpha= max(alpha, maxEva)
         if beta<=alpha
       break
       return maxEva

     else                           // for Minimizer player
         minEva= +infinity
         for each child of node do
         eva= minimax(child, depth-1, alpha, beta, true)
         minEva= min(minEva, eva)
         beta= min(beta, eva)
        if beta<=alpha
       break
     return minEva
```

Working of Alpha-Beta Pruning:

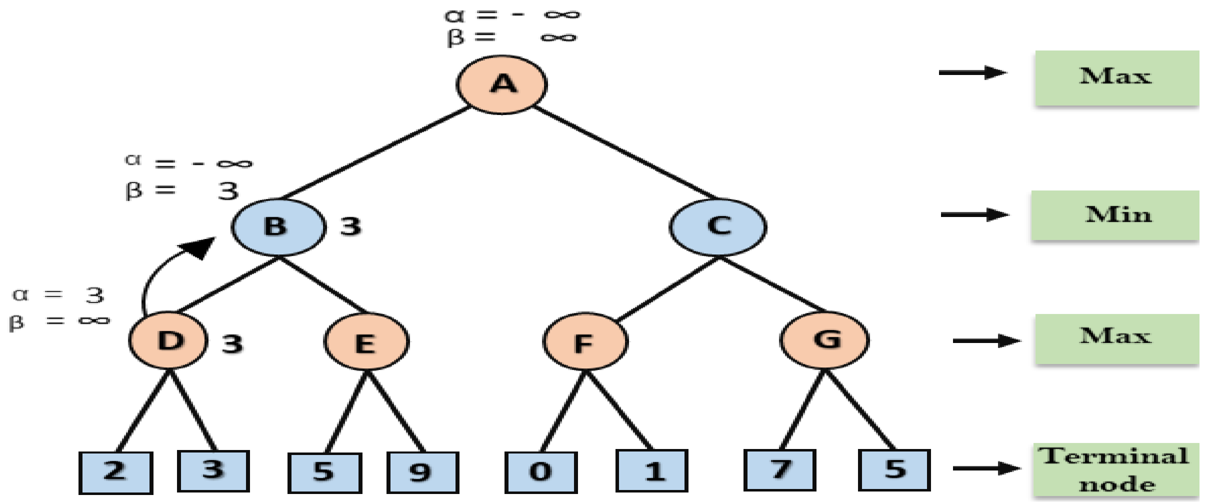
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.



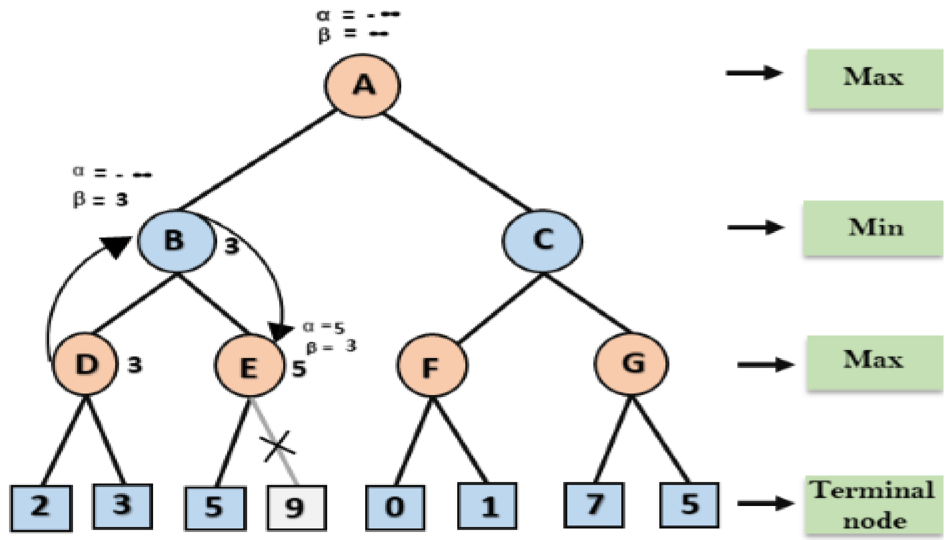
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.



In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

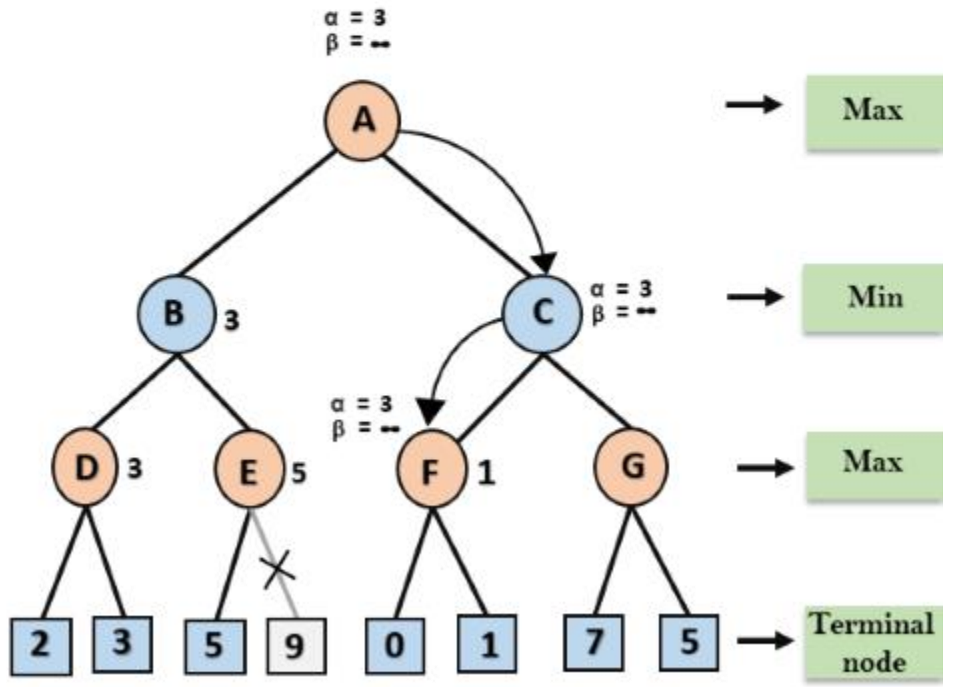
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



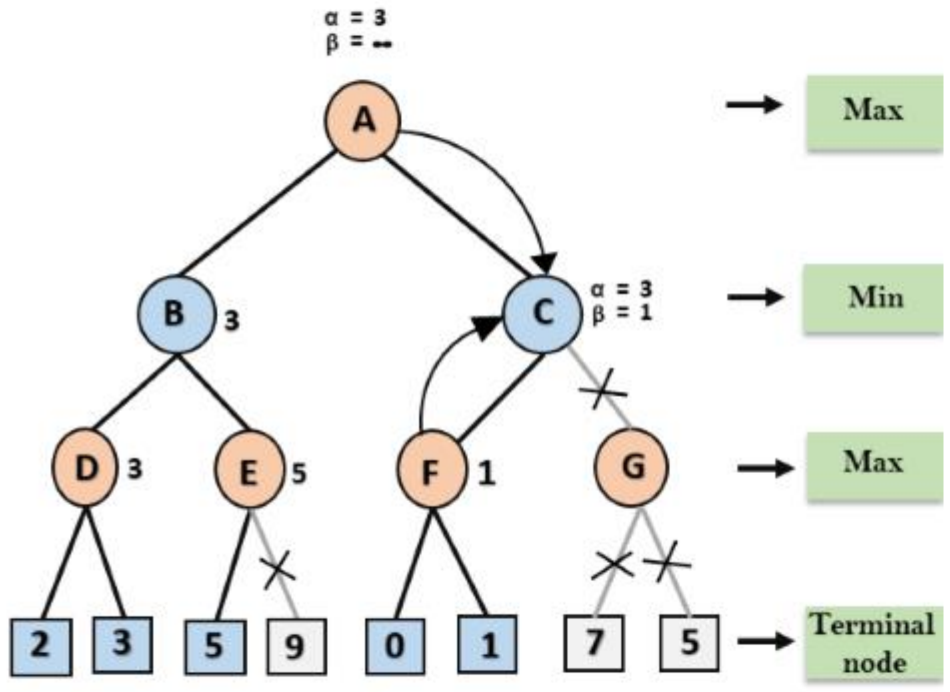
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

At node C, α=3 and β= +∞, and the same values will be passed on to node F.

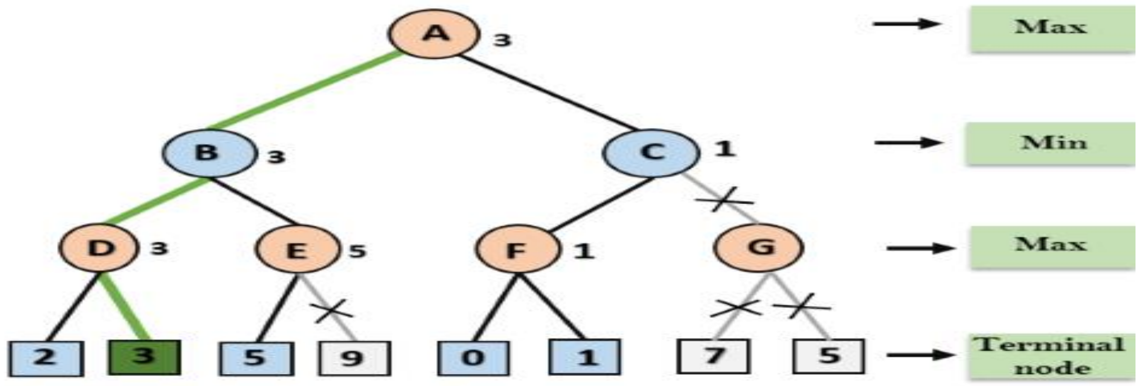
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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



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



Move Ordering in Alpha-Beta pruning:

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

- **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$.

- **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b^{m/2})$.

Rules to find good ordering:

Following are some rules to find good ordering in alpha-beta pruning:

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