RV College of Engineering®

Go, change the world

Unit-2

Artificial Intelligence and Machine learning

IS353IA

Outline

- **Informed Search Methods**
  - → Heuristic Functions
  - → A* Search
- **Beyond Classical Search**
  - → Local Search & Optimization
  - → Hill-climbing Search
  - → Simulated Annealing
  - → Local-Beam Search
  - → Genetic Algorithms
- **Game Playing**
  - → Introduction
  - → Game as a Search problem
  - → Perfect Decisions in Two-Persons
  - → Games Imperfect Decisions
  - → Alpha-Beta Pruning

Informed Search Methods

Introduction

- **Informed Search Strategy**
  - One that uses problem-specific knowledge beyond the definition of the problem itself
  - This finds the solutions more efficiently than uninformed strategy
- **For example**
  - The search algorithm comprise of the knowledge like how far the goal is, path cost, how to reach to goal node, etc.
  - This knowledge help agents to explore less to the search space and find more efficiently the goal node

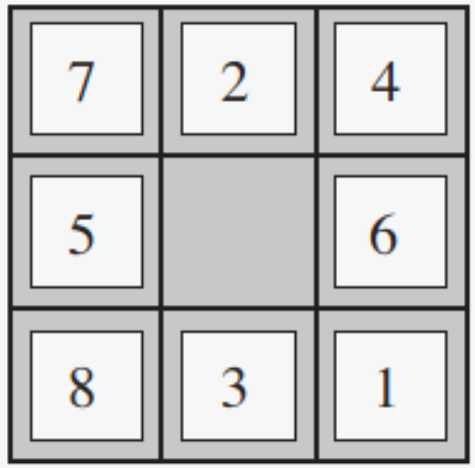
Heuristics Function

- Heuristic is a function which is used in Informed Search, and it finds the most promising path.
- It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal
- The heuristic method, might not always give the best solution, but it guaranteed to find a good solution in reasonable time
- Heuristic function estimates how close a state is to the goal
- It is represented by h(n), and it calculates the cost of an optimal path between the pair of states
- The value of the heuristic function is always positive

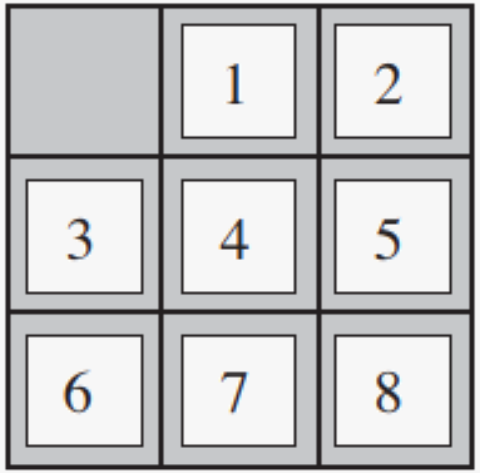
f(n) = h(n)

Here, f(n) is heuristic cost, and h(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

Heuristics Function (Contd.)



Start State



Goal State

- The 8-puzzle was one of the earliest heuristic search problems
- The objective of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration
- Here, there are two commonly used candidates:
  - h1 = the number of misplaced tiles, all of the eight tiles are out of position, so the start state would have h1 = 8
  - h2 = the sum of the distances of the tiles from their goal positions
    - The distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the city block distance or Manhattan distance
    - Tiles 1 to 8 in the start state give a Manhattan distance of h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18

Heuristics Function (Contd.)

⇒ Pure Heuristic Search

- Pure heuristic search is the simplest form of heuristic search algorithms
- It expands nodes based on their heuristic value h(n)
- It maintains two lists, **OPEN** and **CLOSED** list
- In the **CLOSED** list, it places those nodes which have already expanded
- **In OPEN** list, it places nodes which have yet not been expanded
- On each iteration, each node 'n' with the lowest heuristic value is expanded and generates all its successors and 'n' is placed to the closed list
- The algorithm continues until a goal state is found

Heuristics Function (Contd.)

⇒ A* Search Algorithm

- A* search is the most commonly known form of best-first search
- It uses heuristic function h(n), and cost to reach the node 'n' from the start state g(n)
- A* search algorithm finds the shortest path through the search space using the heuristic function
- This search algorithm expands less search tree and provides optimal result faster
- In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a fitness number

Heuristics Function (Contd.)

⇒ A* Search Algorithm

f(x) = g(x)+h(x)

g(x) - backward cost

h(x) - forward cost

Step1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not,

if the list is empty then return failure and stops.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise

Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for 'n' and place into Open list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.

Step 6: Return to **Step 2**

Heuristics Function (Contd.)

⇒ A* Search Algorithm

Figure 1

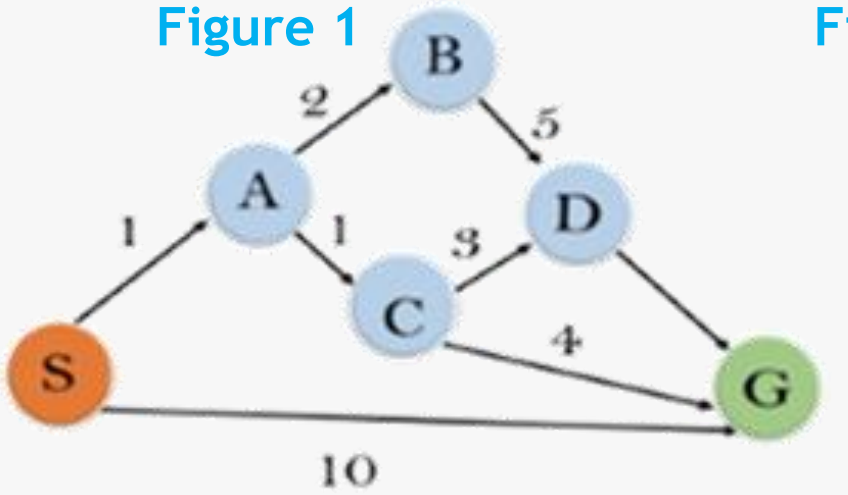
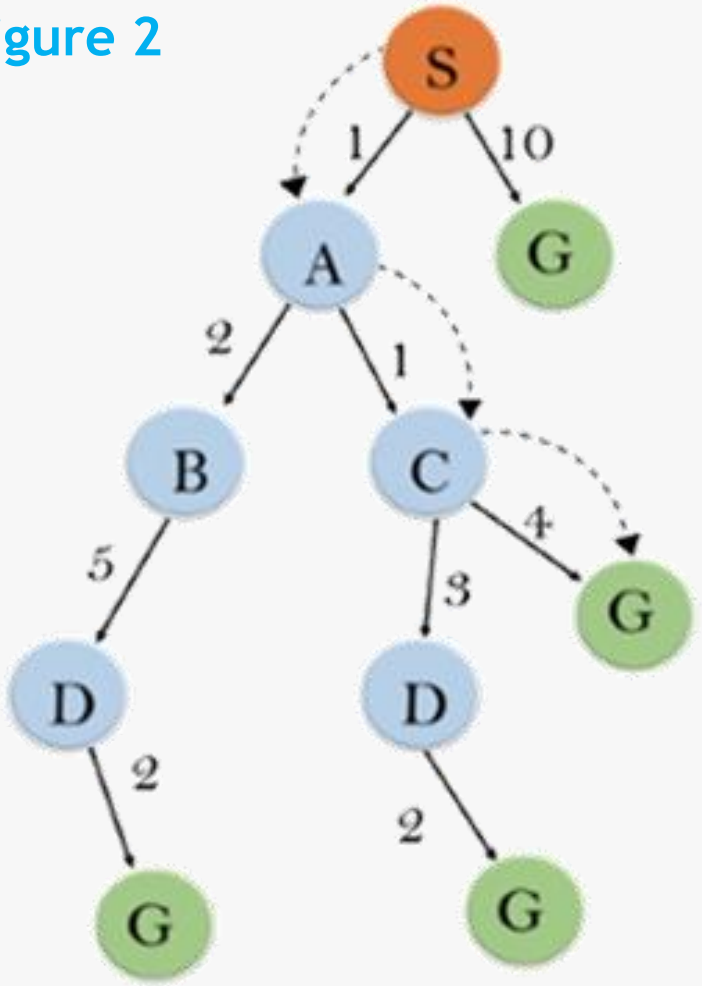


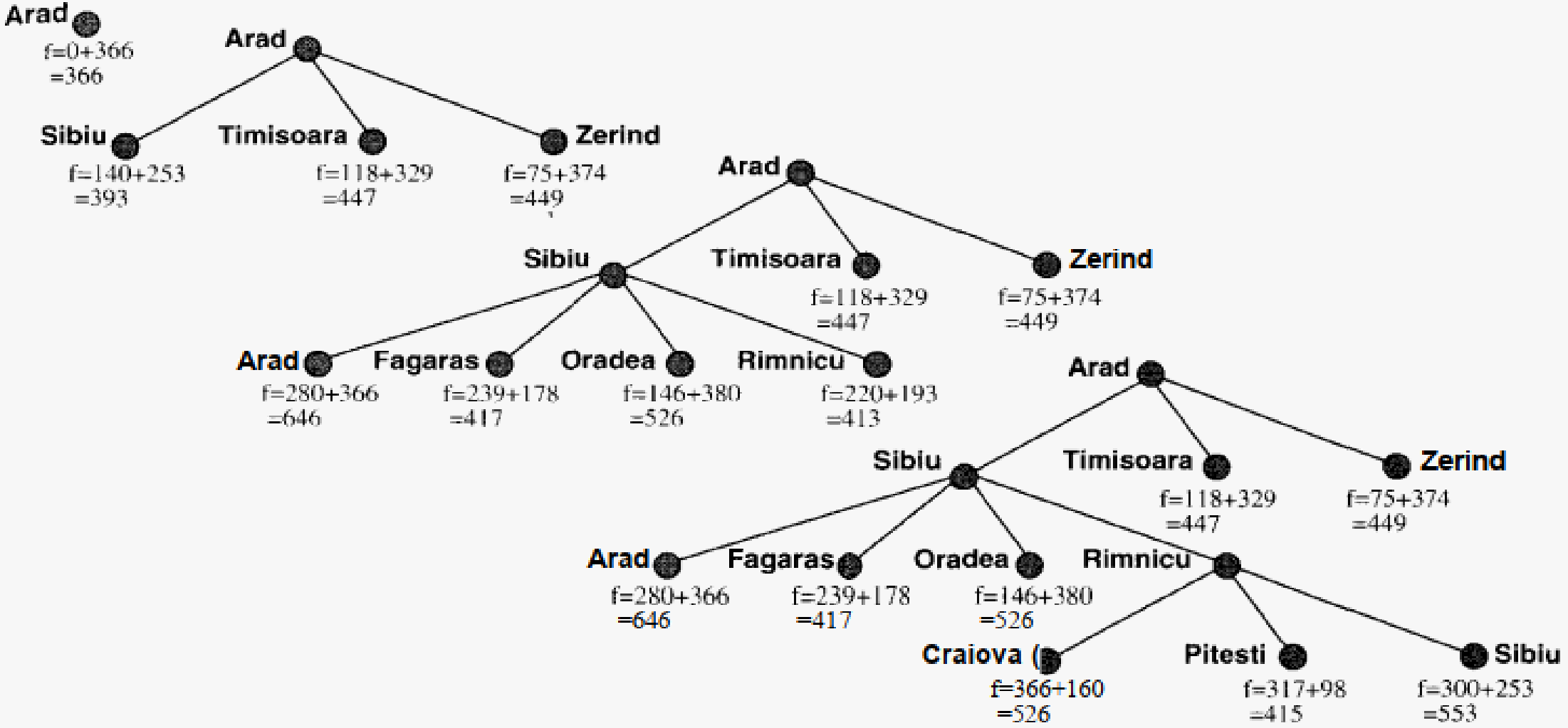
Figure 2



Node	H(n)
S	5
A	3
B	4
C	2
D	6
G	0

Heuristics Function (Contd.)

⇒ A* Search Algorithm



Memory Bounded Search

⇒ Iterative deepening A* search (IDA*)

- Iterative deepening A* (IDA*) is a graph traversal and path search algorithm
- The algorithm can find the shortest path between a designated start node and any member of a set of goal nodes in a weighted graph
- It is a variant of iterative deepening depth-first search that borrows the idea to use a heuristic function to evaluate the remaining cost to get to the goal from the A* search algorithm

```
function IDA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: f-limit, the current f-COST limit
          mot, a node

  root ← MAKE-NODE(INITIAL-STATE[problem])
  f-limit ← f-COST(root)
  loop do
     solution, f-limit ← DFS-CONTOUR(root, f-limit)
     if solution is non-null then return solution
     if f-limit = ∞ then return failure; end
```

```
function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f-COST limit
  inputs: node, a node
          f-limit, the current f-COST limit
  static: next-f, the f-COST limit for the next contour, initially ∞

  if f-COST[node] > f-limit then return null, f-COST[node]
  if GOAL-TEST[problem](STATE[node]) then return node, f-limit
  for each node s in SUCCESSORS(node) do
     solution, new-f ← DFS-CONTOUR(s, f-limit)
     if solution is non-null then return solution, f-limit
     next-f ← MIN(next-f, new-f); end
  return null, next-f
```

Iterative Improvement Algorithms (Contd.)

- Unlike A*, IDA* doesn't utilize dynamic programming and therefore often ends up exploring the same nodes many times
- It does everything that the A* does, it has the optimal characteristics of A* to find the shortest path but it uses less memory than A*.

⇒ Simplified Memory-Bounded A*

- SMA* is a shortest path algorithm that is based on the A* algorithm.
- The difference between SMA* and A* is that SMA* uses a bounded memory, while the A* algorithm might need exponential memory.
- SMA* has the following properties:
  - It will utilize whatever memory is made available to it.
  - It avoids repeated states as far as its memory allows.
  - It is complete if the available memory is sufficient to store the shallowest solution path.
  - It is optimal if enough memory is available to store the shallowest optimal solution path.
  - Otherwise, it returns the best solution that can be reached with the available memory.
  - When enough memory is available for the entire search tree, the search is optimally efficient

⇒ Simplified Memory-Bounded A*

- SMA*, just like A* evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from he node to the goal: **f(n) = g(n) + h(n)**
- Since g(n) → is the path cost from the start node to node n
  h(n) → is the estimated cost of the cheapest path from n to the goal
  f(n) → estimated cost of the cheapest solution through n
- The lower the f value is, the higher priority the node will have
- The difference from A* is that the f value of the parent node will be updated to reflect changes to this estimate when its children are expanded
- A fully expanded node will have an **'f'** value at least as high as that of its successors
- In addition, the node stores the f value of the best forgotten successor (or best forgotten child). This value is restored if the forgotten successor is revealed to be the most promising successor.

Iterative Improvement Algorithms

⇒ Simplified Memory-Bounded A*

- In many optimization problems, path is irrelevant
- In such cases, can use iterative improvement algorithms; keep a single "current" state, try to improve it
- Iterative improvement algorithms divide into two major classes
  - Hill-climbing (gradient descent)
  - Simulated annealing
- Hill-Climbing (Gradient Descent)
  - A hill-climbing algorithm is an Artificial Intelligence (AI) algorithm that increases in value continuously until it achieves a peak solution
- Simulated Annealing
  - Simulated Annealing (SA) mimics the Physical Annealing process but is used for optimizing parameters in a model

Beyond Classical Search

Local search and optimization

- **Local search**
  - Keep track of single current state
  - Move only to neighboring states
  - Ignore paths

- **Advantages:**
  - Use very little memory
  - Can often find reasonable solutions in large or infinite (continuous) state spaces.

- **"Pure optimization" problems**
  - All states have an objective function
  - Goal is to find state with max (or min) objective value
  - Does not quite fit into path-cost/goal-state formulation
  - Local search can do quite well on these problems.

Local Classical Algos. and Optimization Problems

Goal Satisfaction

reach the goal node
Constraint satisfaction

Optimization

optimize(objective fn)
Constraint Optimization

You can go back and forth between the two problems
Typically in the same complexity class

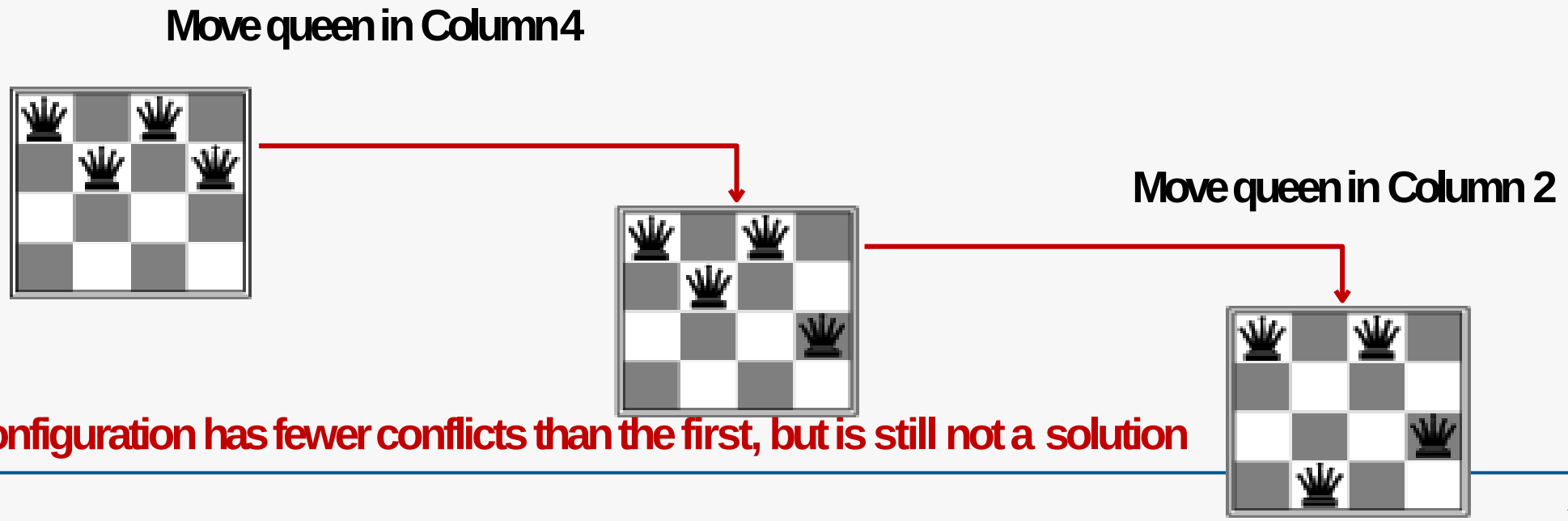
Local Classical Algos. and Optimization Problems

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
  - Local search: widely used for very big problems
  - Returns good but not optimal solutions in general
- The state space consists of "complete" configurations
  - For example, every permutation of the set of cities is a configuration for the traveling salesperson problem
- The goal is to find a “close to optimal” configuration satisfying constraints
  - Examples: n-Queens, VLSI layout, exam time table
- Local search algorithms
  - Keep a single "current" state, or small set of states
  - Iteratively try to improve it / them
  - Very memory efficient since only a few states are stored

Example: 4-queens

Goal: Put 4 queens on an 4 × 4 board with no two queens on the same row, column, or diagonal State space: All configurations with the queens in distinct columns

State transition: Move a queen from its present place to some other square in the same column Local Search: Start with a configuration and repeatedly use the moves to reach the goal



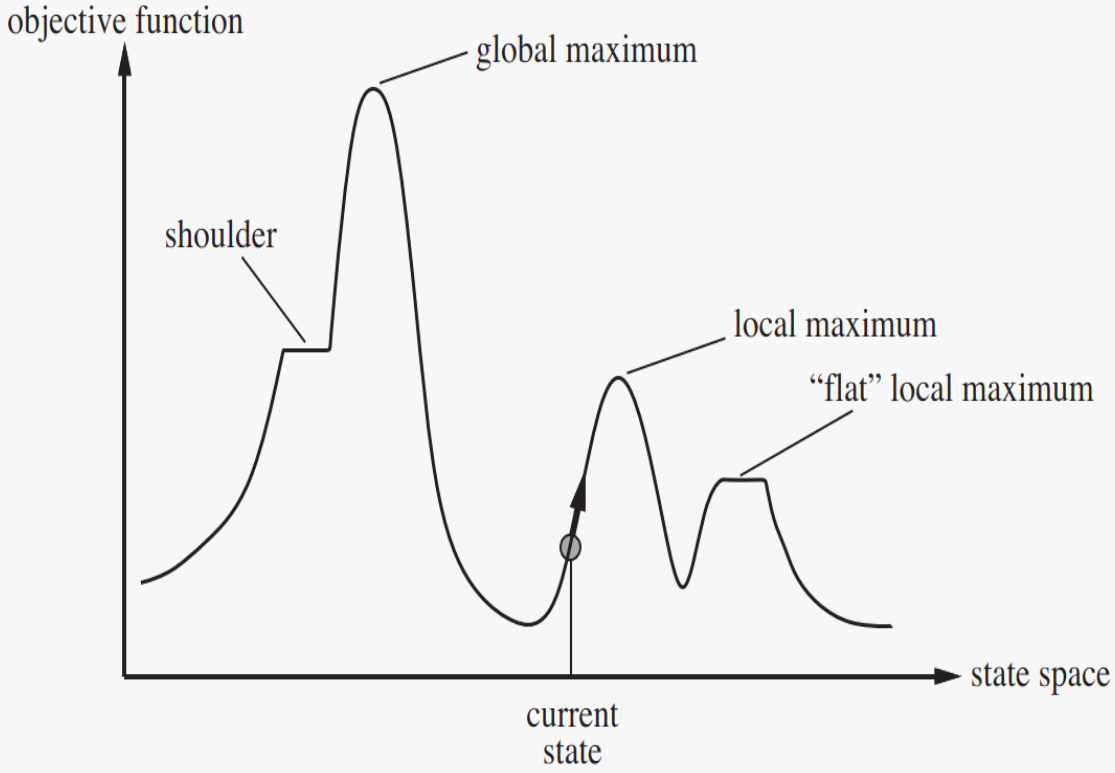
The last configuration has fewer conflicts than the first, but is still not a solution

Hill Climbing

- A hill-climbing algorithm is a local search algorithm that moves continuously upward (increasing) until the best solution is attained
- This algorithm comes to an end when the peak is reached
- This algorithm has a node that comprises two parts: state and value. It begins with a non-optimal state (the hill's base) and upgrades this state until a certain precondition is met
- The heuristic function is used as the basis for this precondition. The process of continuous improvement of the current state of iteration can be termed as climbing. This explains why the algorithm is termed as a hill-climbing algorithm
- A hill-climbing algorithm has four main features
  - It employs a **greedy approach**
  - **No Backtracking:** It does not look at the previous states
  - **Feedback mechanism:** The algorithm has a feedback mechanism that helps it decide on the direction of movement
  - **Incremental change:** The algorithm improves the current solution by incremental changes

Hill Climbing

- The following diagram shows a simple state-space diagram

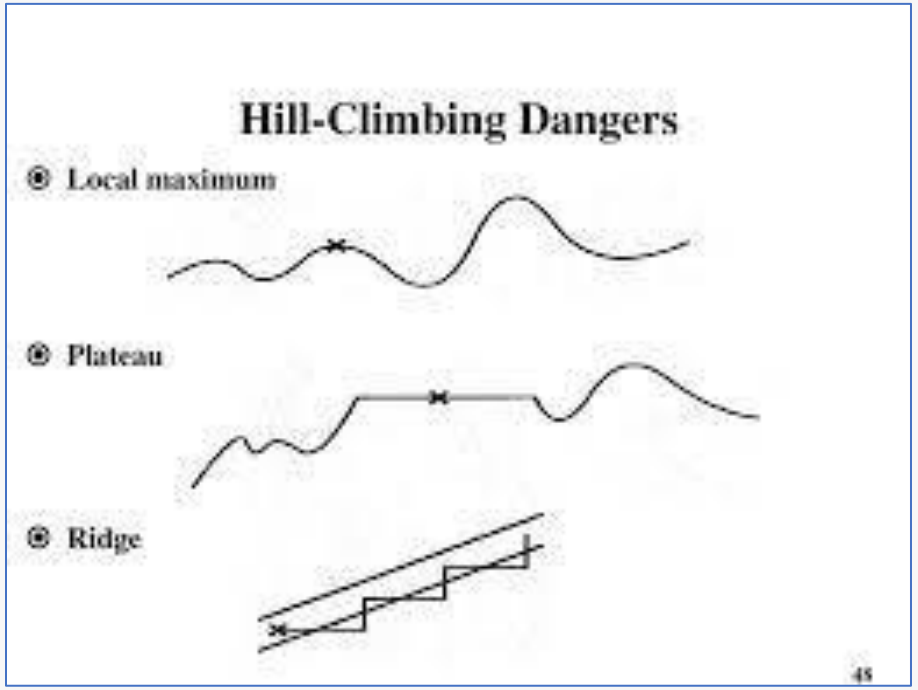


- **Local maximum:** A local maximum is a solution that surpasses other neighboring solutions or states but is not the best possible solution
- **Current state:** This is the existing or present state
- **Flat local maximum:** This is a flat region where the neighboring solutions attain the same value
- **Shoulder:** This is a plateau whose edge is stretching upwards
- **Global maximum:** This is the best possible solution achieved by the algorithm

Hill Climbing

- Problem with the Hill Climbing

  - **Local maximum:** At this point, the neighboring states have lower values than the current state. The greedy approach feature will not move the algorithm to a worse off state. This will lead to the hill-climbing process's termination, even though this is not the best possible solution
  - **Plateau:** In this region, the values attained by the neighboring states are the same. This makes it difficult for the algorithm to choose the best direction
  - **Ridge:** The hill-climbing algorithm may terminate itself when it reaches a ridge. This is because the peak of the ridge is followed by downward movement rather than upward movement



Hill Climbing

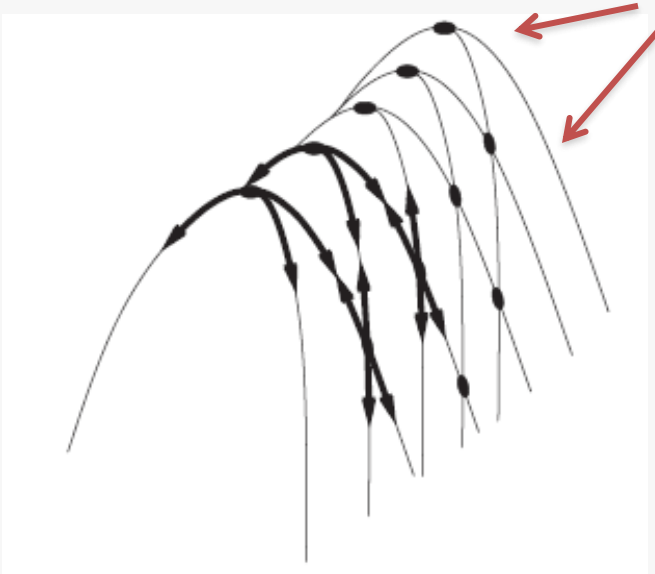

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

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

Hill Climbing - Difficulties

Note: these difficulties apply to all local search algorithms, and usually become much worse as the search space becomes higher dimensional

- **Ridge problem: every neighbor appears to be downhill**
  - **But, search space has an uphill (just not in neighbors)**
  -

Ridge:
Fold a piece of paper and hold it tilted up at an unfavorable angle to every possible search space step. Every step leads downhill; but the ridge leads uphill.

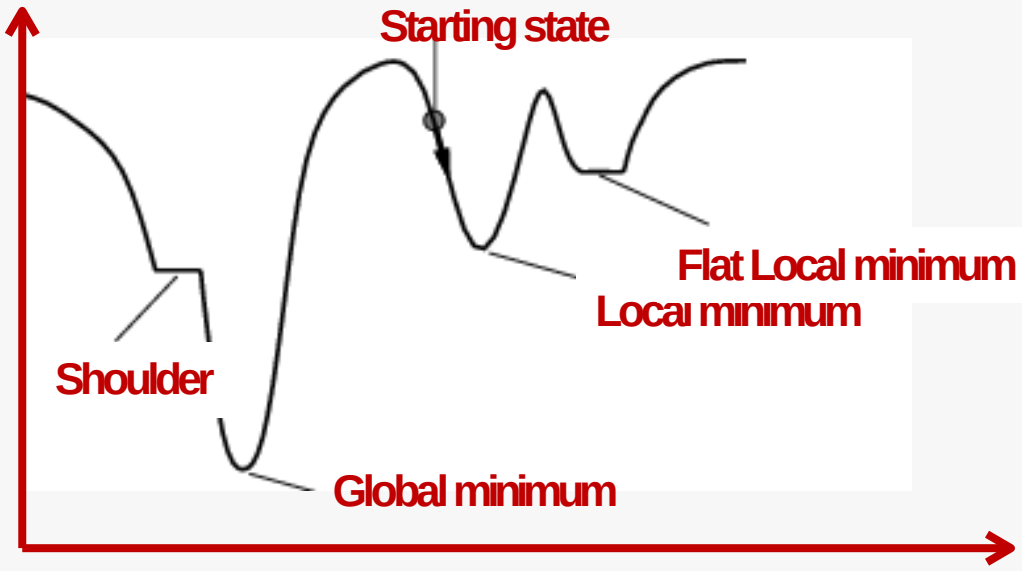


Simulated Annealing

IDEA: Escape local maxima by allowing some "bad" moves but gradually decrease their probability

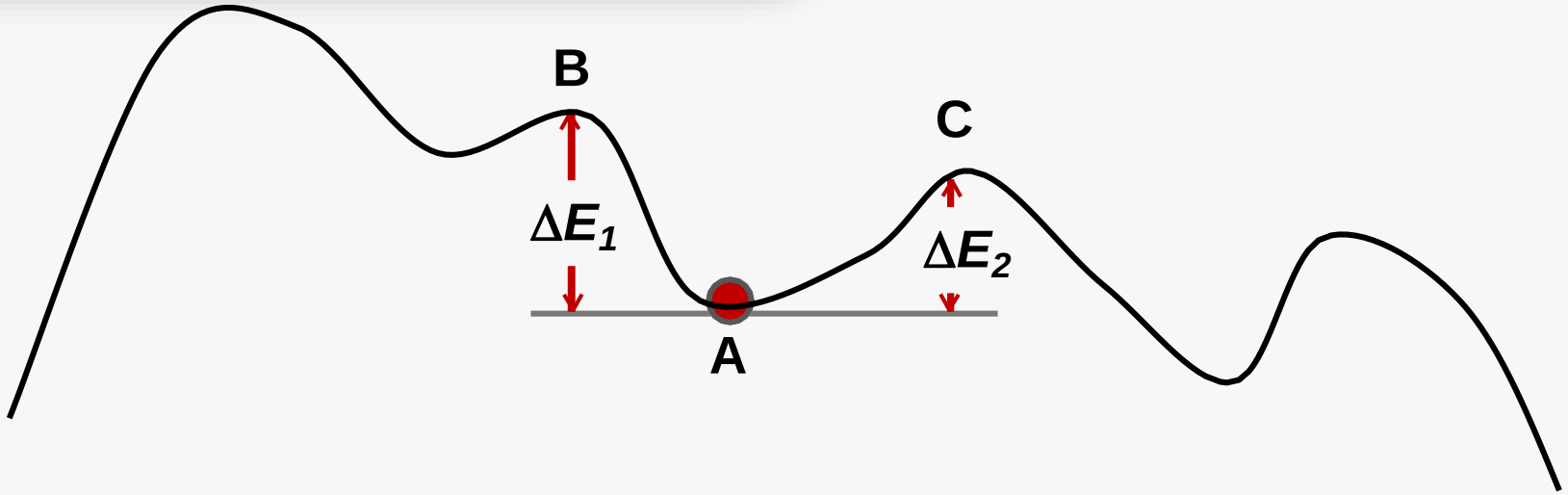
- The probability is controlled by a parameter called temperature
- Higher temperatures allow more bad moves than lower temperatures
- **Annealing:** Lowering the temperature gradually **Quenching:** Lowering the temperature rapidly

```
function SIMULATED-ANNEALING( problem, schedule )
    current ← INITIAL-STATE[ problem ]
    for t ← 1 to ∞ do
        T ← schedule[ t ]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[ next ] – VALUE[ current ]
        if ΔE < 0 then current ← next
        else current ← next with probability e^(−ΔE/T)
```



How Simulated Annealing Works ?

Probability of making a bad move = $e^{-\Delta E/T} = \frac{1}{e^{\Delta E/T}}$



Since $\Delta E_1 > \Delta E_2$ moving from A to C is exponentially more probable than moving from A to B

Properties-Simulated Annealing

- It can be proven that:
  - If *T* decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
  - Since this can take a long time, we typically use a temperature schedule which fits our time budget and settle for the sub-optimal solution
- Simulated annealing works very well in practice
- Widely used in VLSI layout, airline scheduling, etc.

Local-beam search

Instead of working on only one configuration at any time, we could work on multiple promising configurations concurrently

LOCAL BEAM SEARCH

Maintain *k* states rather than just one. Begin with *k* randomly generated states

In each iteration, generate all the successors of all *k* states

Stop if a goal state is found; otherwise Select the *k* best successors from the complete list and repeat

Genetic Algorithms

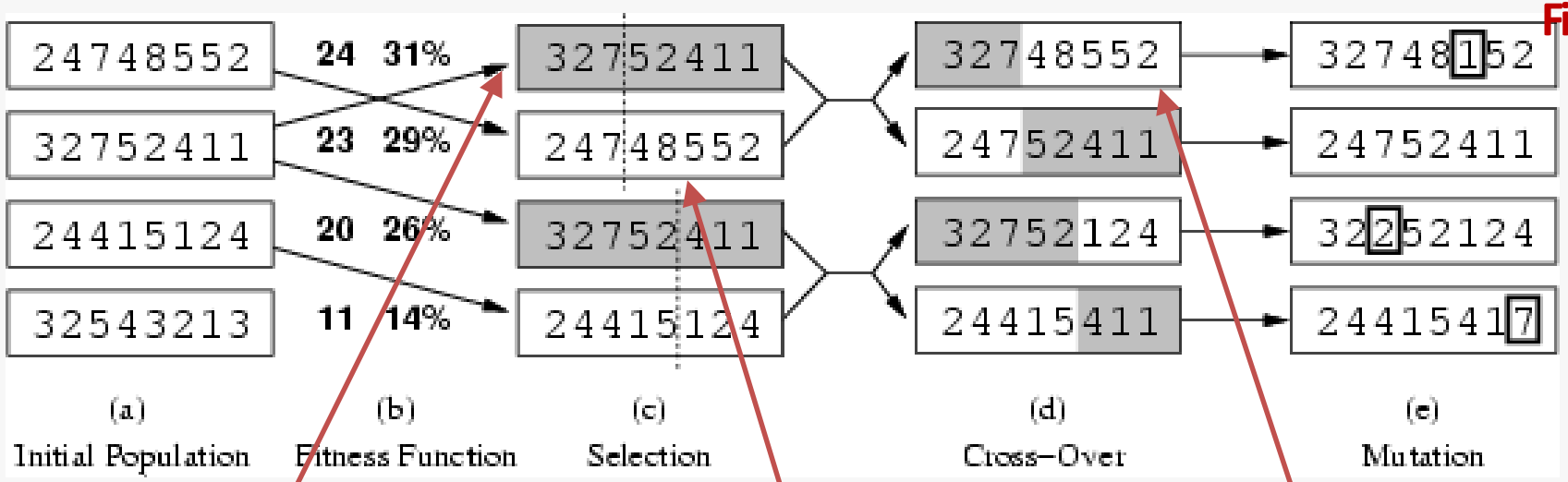
GENETIC ALGORITHMS

States are strings over a finite alphabet (genes). Begin with *k* randomly generated states (population). Select individuals for next generation based on a fitness function.

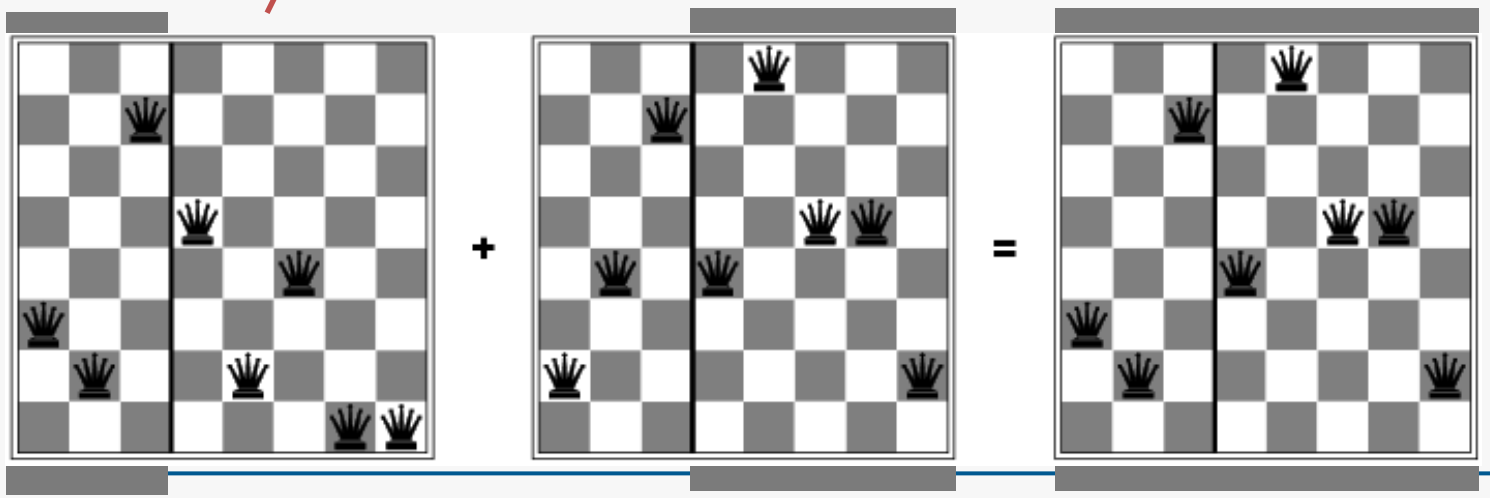
Two types of operators for creating the next states:

- Crossover: Fit parents to yield next generation (offspring)
- Mutation: Mutate a parent to create an offspring randomly with some low probability

Genetic Algorithms



Fitness function: # non-attacking pairs (min = 0, max = 8×7 / 2 = 28)



Population fitness = 24+23+20+11 = 78

P(Gene-1 is chosen)
= Fitness of Gene-1 / Population fitness
= 24 / 78 = 31%

P(Gene-2 is chosen)
= Fitness of Gene-2 / Population fitness
= 23 / 78 = 29%

Genetic Algorithms


```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Game Playing

Game Playing

⇒ Introduction

- A Game playing is one of the oldest areas of endeavor in artificial intelligence
- In 1950, almost as soon as computers became programmable, the first chess programs were written by Claude Shannon (the inventor of information theory) and by Alan Turing.
- The presence of an opponent makes the decision problem somewhat more complicated than the search problems discussed.
- The opponent introduces uncertainty, because one never knows what he or she is going to do. In essence, all game-playing programs must deal with the **Contingency Problem**

Game Playing

⇒ Game as a Search Problem

- A game can be defined as a type of search in AI, which can be formalized of the following elements:
  - **Initial state:** It specifies how the game is set up at the start
  - **Player(s):** It specifies which player has moved in the state space
  - **Action(s):** It returns the set of legal moves in state space
  - **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
  - **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
  - **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0

Game Playing (Contd.)

⇒ Perfect Decisions in Two-Persons Game

- Consider the general case of a game with two players, whom we will call MAX and MIN
- MAX moves first, and then they take turns moving until the game is over
- At the end of the game, points are awarded to the winning player
- A game can be formally defined as a kind of search problem with the following components
  - The initial state, which includes the board position and an indication of whose move it is.
  - A set of operators, which define the legal moves that a player can make
  - A terminal test, which determines when the game is over. States where the game has ended are called terminal states
  - A utility function (also called a payoff function), which gives a numeric value for the outcome of a game. In chess, the outcome is a win, loss, or draw, which we can represent by the values +1, —1, or 0.

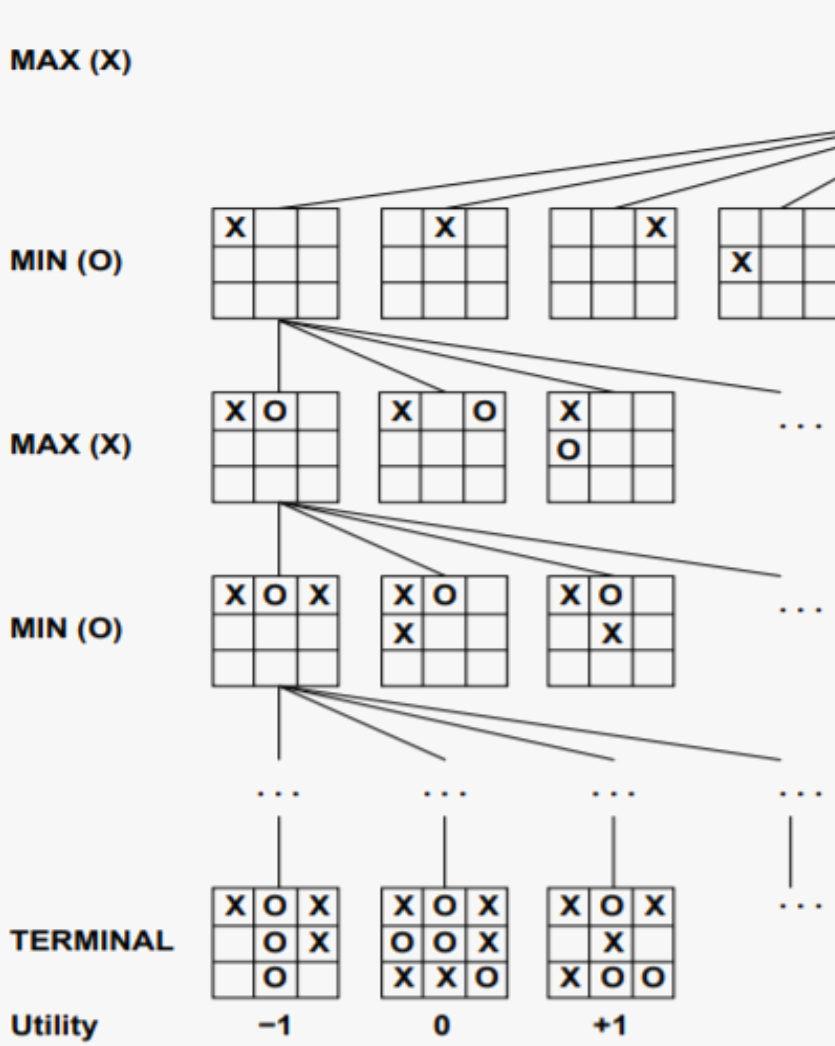
Game Playing (Contd.)

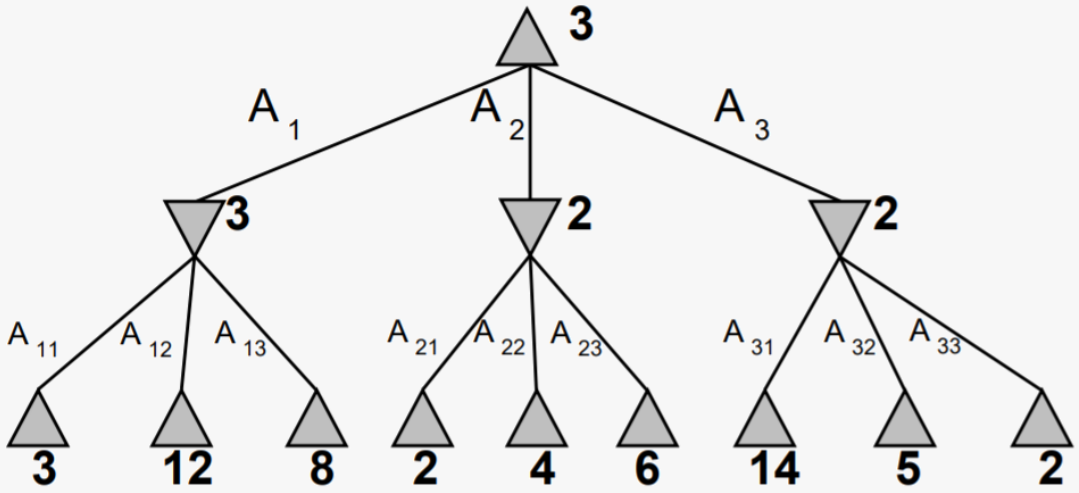
⇒ Perfect Decisions in Two-Persons Game

- If this were a normal search problem, then all MAX would have to do is search for a sequence of moves that leads to a terminal state that is a winner
- From the initial state, MAX has a choice of nine possible moves
- Play alternates between MAX placing x's and MIN placing o's until we reach leaf nodes corresponding to terminal states: states where one player has three in a row or all the squares are filled.
- The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names).
- It is MAX'S job to use the search tree (particularly the utility of terminal states) to determine the best move
- Even a simple game like Tic-Tac-Toe is too complex to show the whole search tree

Game Playing (Contd.)

⇒ Perfect Decisions in Two-Persons Game





Game Playing (Contd.)

⇒ Minimax Algorithm

- The minimax algorithm is designed to determine the optimal strategy for MAX, and thus to decide what the best first move is
- The algorithm consists of five steps:
  - Generate the whole game tree, all the way down to the terminal states
  - Apply the utility function to each terminal state to get its value
  - Use the utility of the terminal states to determine the utility of the nodes one level higher up in the search tree
  - Continue backing up the values from the leaf nodes toward the root, one layer at a time
  - Eventually, the backed-up values reach the top of the tree; at that point, MAX chooses the move that leads to the highest value

Game Playing (Contd.)

⇒ Minimax Algorithm

- 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
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game
- In this algorithm two players play the game, one is called MAX and other is called MIN
- 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

Game Playing (Contd.)

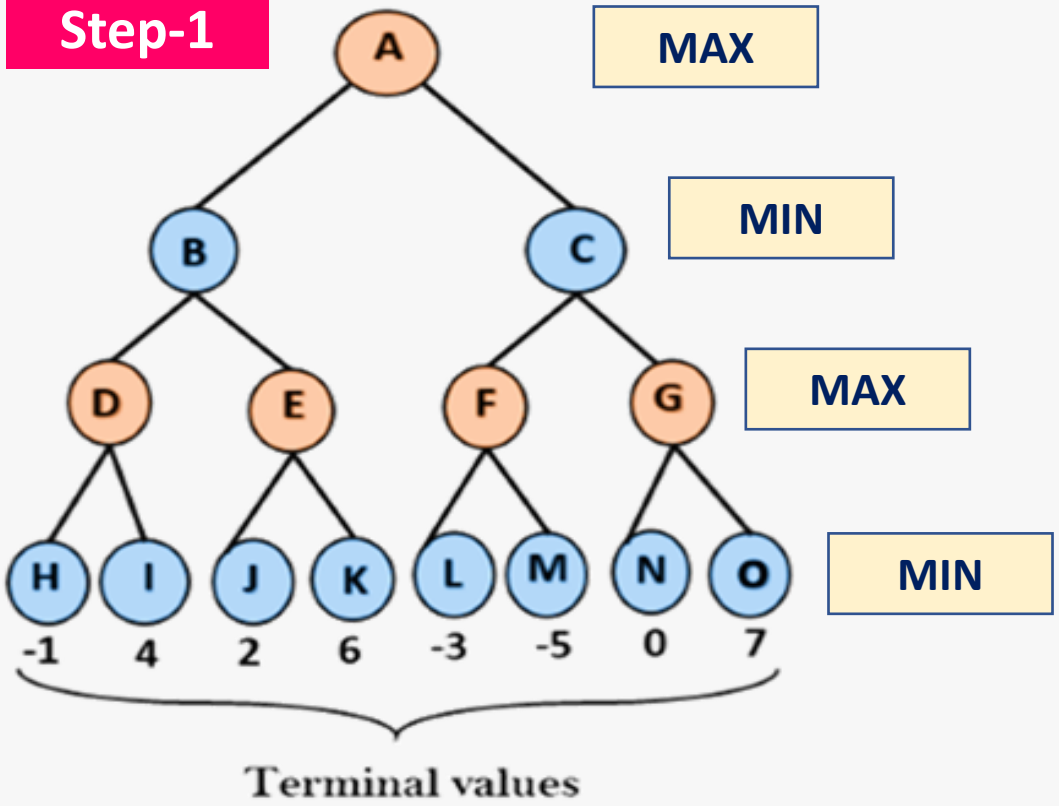
⇒ Minimax Algorithm

- The minimax algorithm is designed to determine the optimal strategy for MAX, and thus to decide what the best first move is
- The algorithm consists of five steps:
  - Generate the whole game tree, all the way down to the terminal states
  - Apply the utility function to each terminal state to get its value
  - Use the utility of the terminal states to determine the utility of the nodes one level higher up in the search tree
  - Continue backing up the values from the leaf nodes toward the root, one layer at a time
  - Eventually, the backed-up values reach the top of the tree; at that point, MAX chooses the move that leads to the highest value

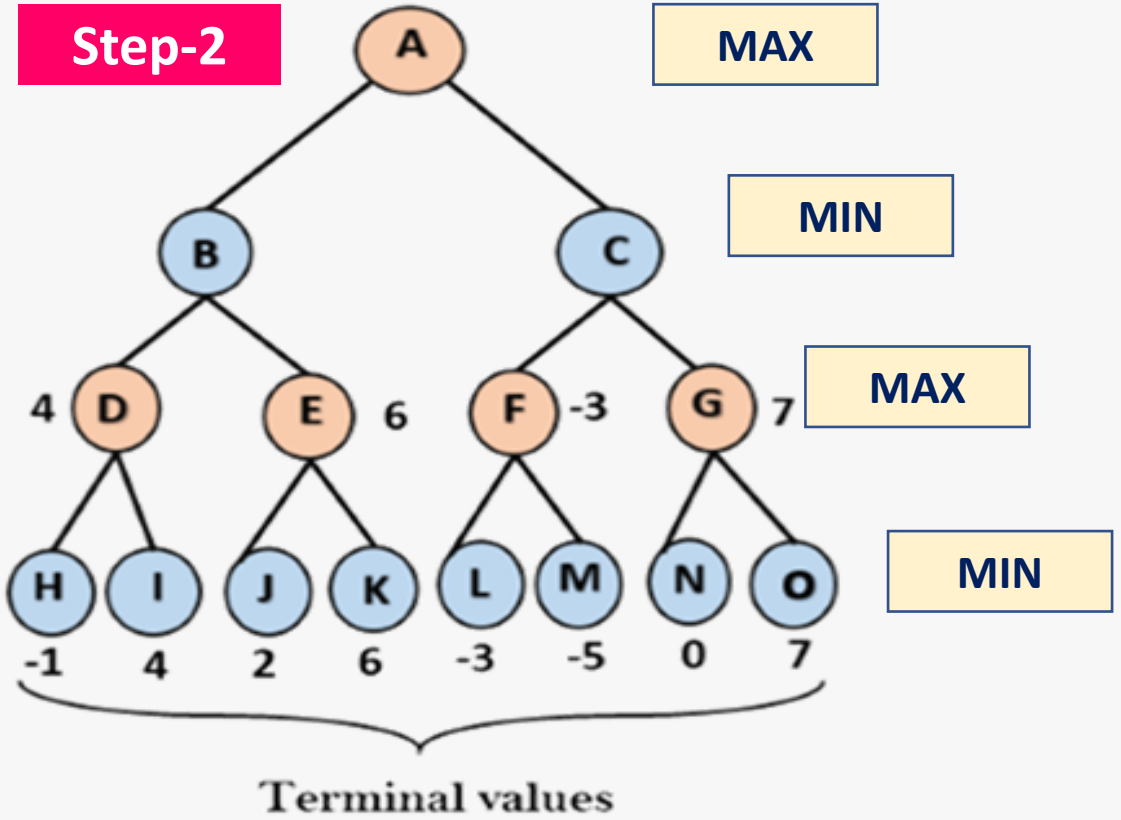
Game Playing (Contd.)

⇒ Minimax Algorithm

Step-1

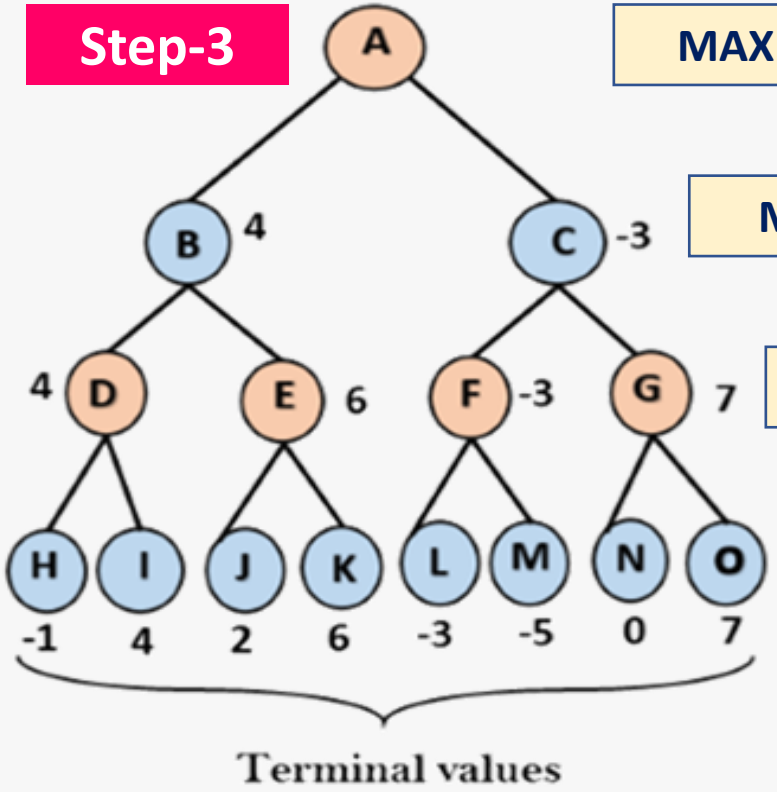


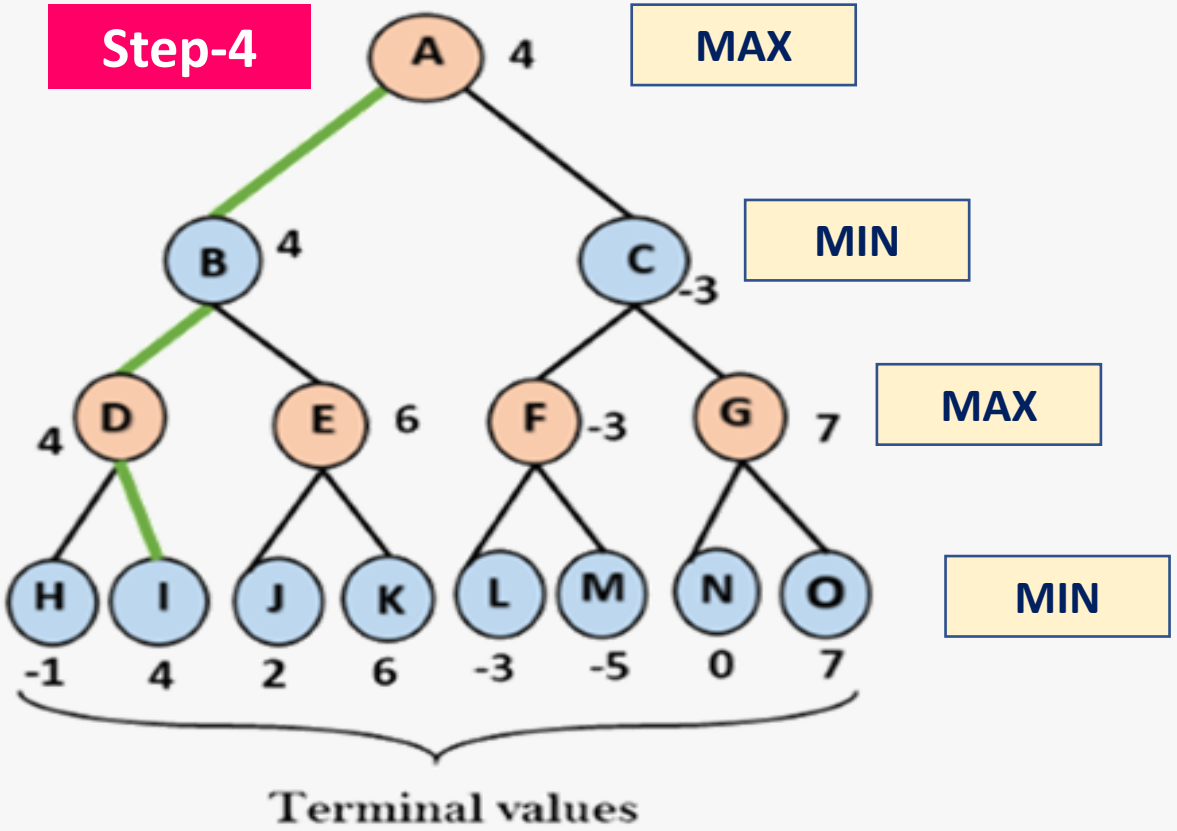
Step-2



Game Playing (Contd.)

⇒ Minimax Algorithm





Game Playing (Contd.)

⇒ Minimax Algorithm

```
function MINIMAX-DECISION(game) returns an operator

    for each op in OPERATORS[game] do
        VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
    end
    return the op with the highest VALUE[op]

---

function MINIMAX-VALUE(state, game) returns a utility value

    if TERMINAL-TEST[game](state) then
        return UTILITY[game](state)
    else if MAX is to move in state then
        return the highest MINIMAX-VALUE of SUCCESSORS(state)
    else
        return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

Game Playing (Contd.)

⇒ Imperfect Decisions

- The minimax algorithm assumes that the program has time to search all the way to terminal states, which is usually not practical
- The program should cut off the search earlier and apply a heuristic evaluation function to the leaves of the tree
- In other words, the suggestion is to alter minimax in two ways
  - The utility function is replaced by an evaluation function EVAL, and the terminal test is replaced by a cutoff test CUTOFF-TEST

Game Playing (Contd.)

⇒ Alpha-Beta Pruning

- In 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

Game Playing (Contd.)

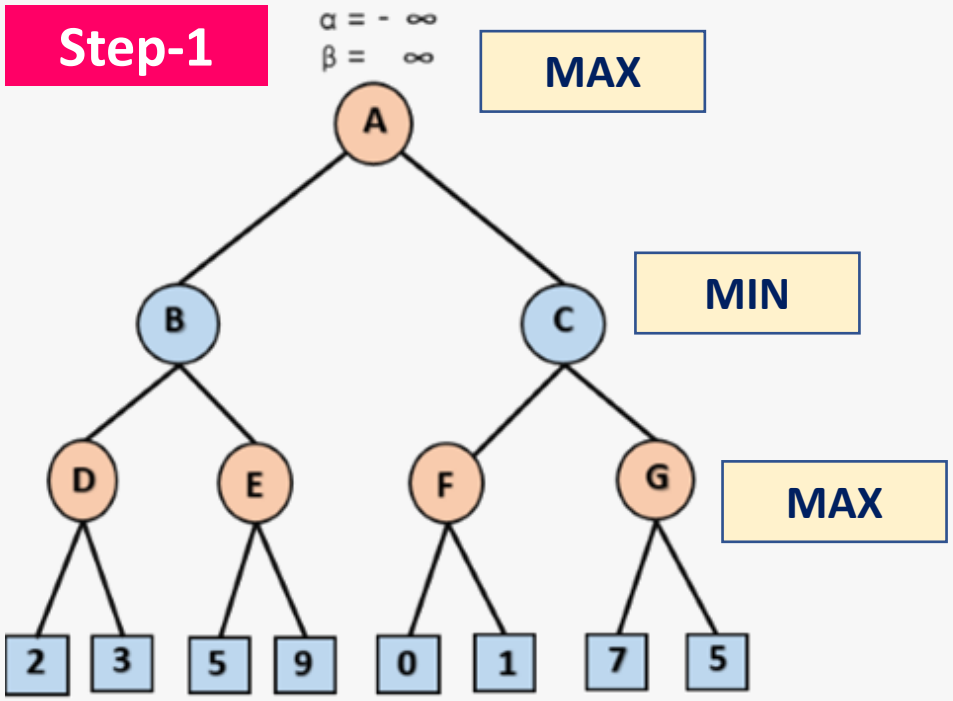
⇒ Alpha-Beta Pruning

- The two-parameter can be defined as:
  - Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.
  - 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
- Points to remember
  - 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

Game Playing (Contd.)

⇒ Alpha-Beta Pruning

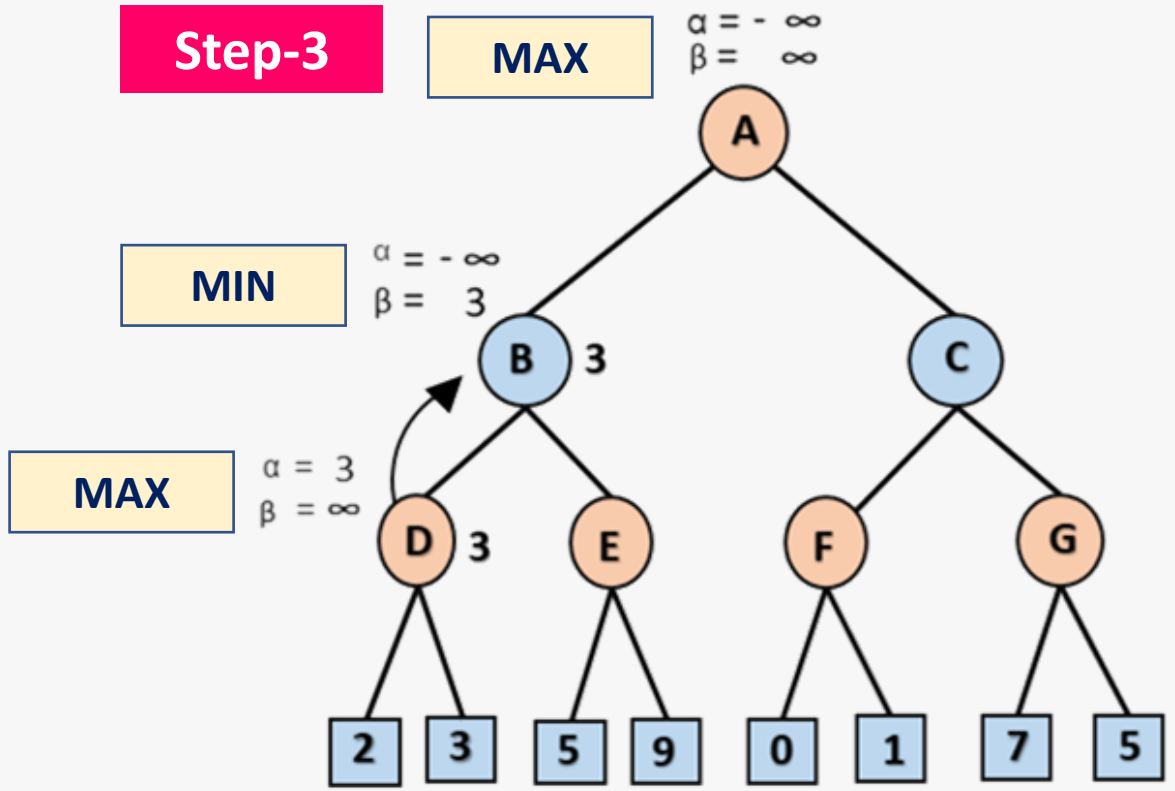
Step-1



Step-2

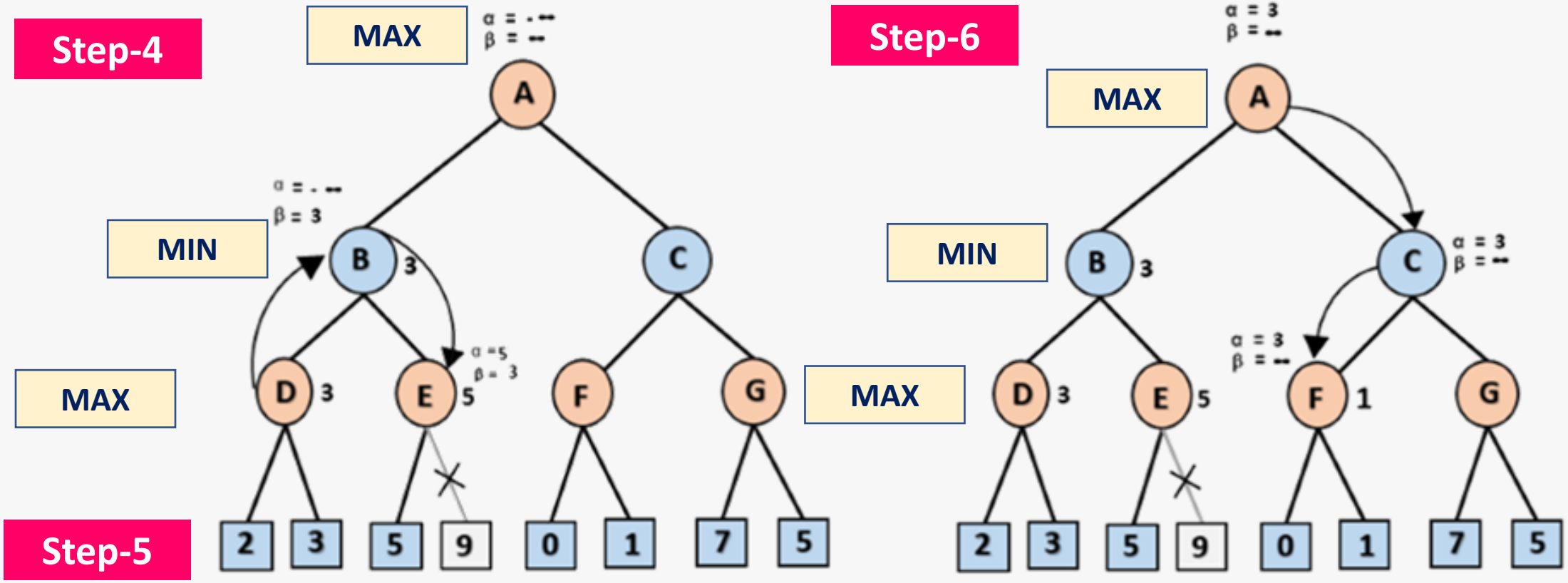
At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3

Step-3



Game Playing (Contd.)

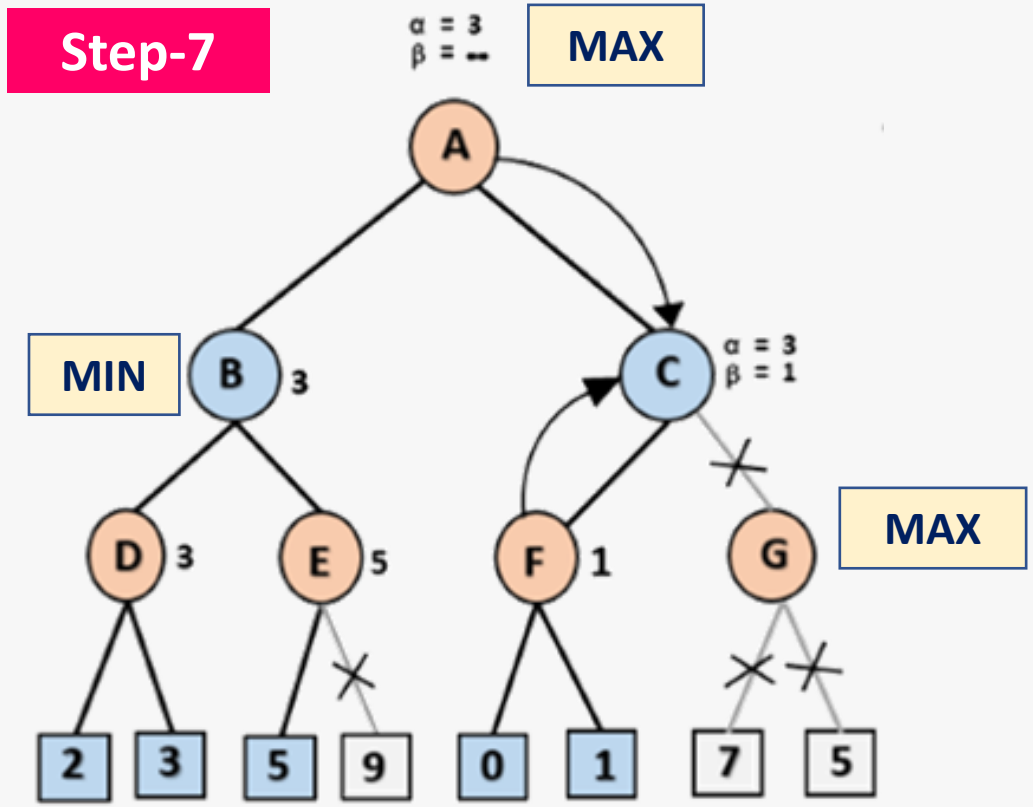
⇒ Alpha-Beta Pruning

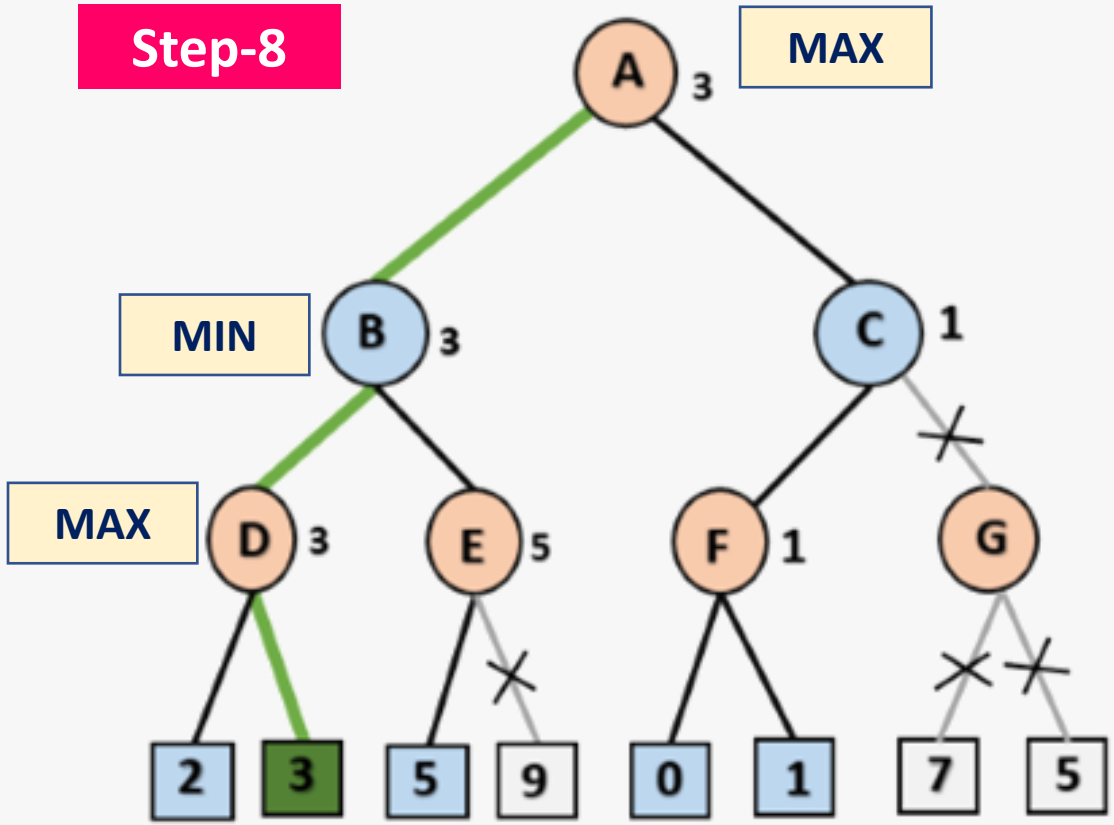


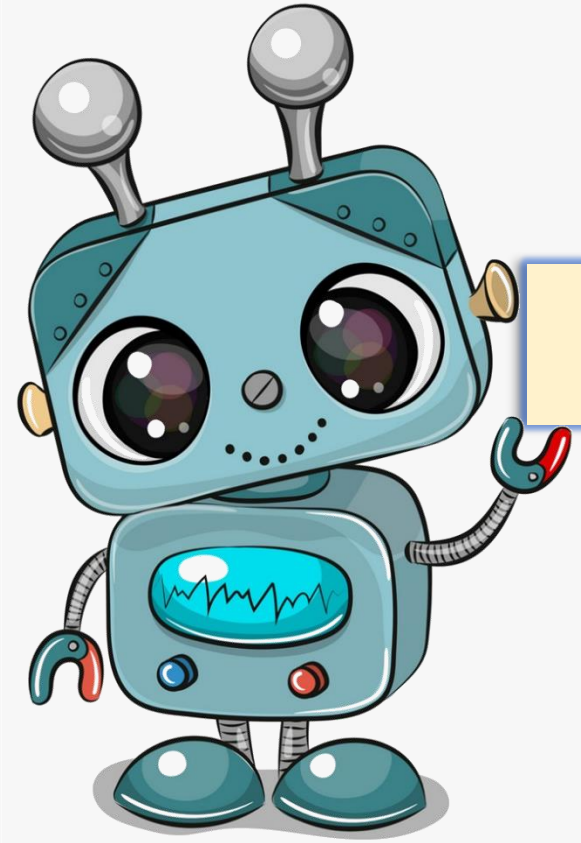
At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C

Game Playing (Contd.)

⇒ Alpha-Beta Pruning







Thank You