# Foundations of Artificial Intelligence

## Unit 2

## Problem Solving & Search Techniques

### Problem Solving using State Space Representations

* **Problem Solving Components**

(i) Initial state - the first state the agent knows itself to be in.

(ii) Operator - description of an action

(iii) State space - all states reachable from the initial state by any sequence action

(iv) Goal Test - which the agent can apply to a single state description to determine if it is a goal state.

(v) Path cost function - assign a cost to a path, which is the sum of the costs of the individual actions along the path.

* **Percepts** - location and state of the environment

**Examples** - For each of the given problems, specify the appropriate state space formulation.

For each - write:
1. Initial state
2. Goal state
3. State space
4. Action / operator
5. Goal Test
6. Path cost

**Q1 A vacuum cleaner world**

1. Percepts - location and state of the environment. eg
   [A, Dirty], [B, Clean]
2. Actions - left, right, suck, No Op

3. States - $S_1, S_2, S_3 \ldots$ (each possible square the vacuum can move onto)

4. Operators - go left, go right, suck

5. Goal test - no dirt left

6. Path cost - each action costs 1

Q2 Traveling in Romania - go from Arad to Bucharest

1. Initial state - at Arad

2. Actions / Successor Function - S(x) = a set of action-state pairs

eg. S(Arad) = <Arad → nextcity, nextcity>

3. Goal Test - x = "at Bucharest"

4. Path cost - sum of distances travelled

5. Solution - sequence of actions from Arad to Bucharest

Q3 Navigation Problem

1. Initial State - start city

2. Set of States - individual cities that can be travelled to

3. Operators - freeway routes from one city to another

4. Goal State - final city to be in

5. Path cost - distance

6. Solution - sequence of operators to get to a specific city

## Q4 - 8 Queens Problem

1. states - an arrangement of $n \leq 8$ queens

2. initial state - no queens on the board

3. actions - add a queen to an empty square (or)

add a queen to a leftmost empty square such that it is not attacked by any other queens

4. goal test - 8 no. of attacking pairs = 0

5. path cost - none per move

6. solution - 8 queens on the board, none attacked.

## Q5 - Robot Assembly

1. States - any configuration of robot (angle/ position) & object parts.

2. Initial state - a configuration of robot parts & unassembled object

3. Actions - continuous motion of robot joints

4. goal test - object assembled

5. path cost - time taken / number of actions

6. solution - assembled object

## Q6 - Learning a spam email classifier

1. States - settings of model parameters

2. Initial state - random parameter settings

3. Actions - moving in parameter space

4. goal test - optimal accuracy?

5. path cost - time taken to find optimal parameters

6. solution - accurate parameters for correct classification

## Q7 - 8-puzzle

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | | 6 |
| 8 | 3 | 1 |

start state

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

goal state

1. states - tile location

2. initial state - starting arrangement of tiles

3. actions - move left, up, right, down

4. goal test - does configuration match the final given configuration?

5. Path cost - one per tile move

6. Solution - tiles match given configuration

Q8 - Crypt Arithmetic Puzzle - for a given word addition, assign unique digits to each alphabet such that they satisfy the given (0-9)

condition. (CROSS + ROADS = DANGER). No 2 letters should have the same value.

1. Initial state - unknown values for each letter

2. Goal state - a valid assignment of digits to letters such that the condition is satisfied

3. Path cost - no. of steps / time taken to find the optimal assignment

4. state space - all possible combination of digit assignments to letters that do not violate the constraints

5. goal test - check whether current assignment satisfies the addition equation

6. actions - assign a digit to a letter

## Q9 Water Jug Problem

1. Initial state : (0,0)

2. State Representation = (x,4)

3. Goal state = (2,n)

4. actions/operators :
   a. Fill 4 gallon $(x,4) \rightarrow (4,4)$ , $x < 4$
   b. Fill 3 gallon $(x,4) \rightarrow (x,3)$ , $4 < 3$
   c. Empty 4 gallon $(x,4) \rightarrow (0,4)$ , $x > 0$
   d. Empty 3 gallon $(x,4) \rightarrow (x,0)$ , $4 > 0$
   e. Pour from 3 gallon into 4 gallon, until full
      $(x,4) \rightarrow (4, 4-(4-x))$ $x+4 \geq 4$ , $4 > 0$
   f. Pour from 4 gallon into 3 gallon until full
      $(x,4) \rightarrow (x-(3-4), 4)$ , $x+4 \geq 3, x > 0$
   g. Pour all water from 3 → 4 $\Rightarrow (x,4) \rightarrow (x+4, 0)$ , $x+4 \leq 4$, $4 > 0$
   h. Pour all water from 4 → 3 $\Rightarrow (x,4) \rightarrow (0, x+4)$ , $x+4 \leq 3$, $x > 0$

5. path cost - one for each pour

6. goal test = is $(x,4) = (2,n)$ ?

**A possible solution to the water jug problem**

| | 4 gallon | 3 gallon | |
|---|---|---|---|
| 1. | 0 | 0 | $(x,4) \rightarrow (x,3)$ |
| 2. | 0 | 3 | $(x+4) \rightarrow (x+4, 0)$ |
| 3. | 3 | 0 | $(x,4) \rightarrow (x,3)$ |
| 4 | 3 | 3 | $(x+4) \rightarrow (4, 4-(4-x))$ |
| 5. | 4 | 2 | $(x,4) \rightarrow (0,4)$ |
| 6. | 0 | 2 | $(x,4) \rightarrow (x+4, 0)$ |
| 7. | 2 | 0 | = goal state. |

# * Uninformed Search Strategies

## * Tree Search Algorithms

→ exploration of state space by generating successors of already explored - states

General Implementation (with a tree & graph data structure)

function TREE SEARCH (problems) returns solution or a failure

assign using
frontier ← initial state

loop do

(i) if the frontier is empty return failure

(ii) choose a leaf node and remove it from the frontier.

(iii) if the node contains a goal state - return solution

(iv) expand the chosen node - add resulting nodes to frontier.

function GRAPH SEARCH (problem) returns solution or failure

frontier ← assign using the initial state

explored set ← initial explored set to empty

loop do

(i) if the frontier is empty return failure

(ii) choose a leaf node and remove it from the frontier.

(iii) if the node contains a goal state, ~~remove~~ return solution

add node to explored set

(iv) expand the chosen node - adding the resulting nodes to the frontier

only if not in frontier and explored set

* How are search strategies defined? → by picking the order of node expansion

* Evaluation of Search Strategies

→ Strategies are evaluated on the basis of the following criteria:

(i) completeness - does it always find a solution if one exists?

(ii) time complexity - number of nodes generated

(iii) space complexity - maximum number of nodes in memory

(iv) optimality - does it always find a least cost solution

→ Time and space complexity are measured in terms of:

(i) b - maximum branching factor of the search tree

(ii) d - depth of the least cost solution

(iii) m - maximum depth of the state space (may be ∞)

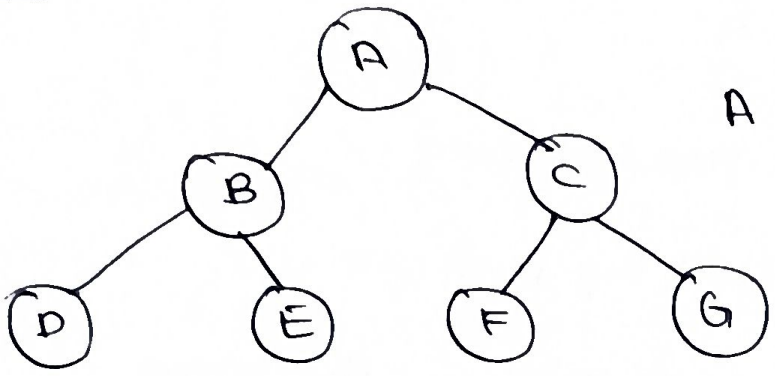
* What is an uninformed search strategy? → use only the information given in the problem definition.

## * Strategy 1: BFS

→ expand the shallowest unexpanded node

→ Implementation - using a FIFO queue - new successors go at the end.

→ Example: The order of expansion would be as follows:




A → B → C → D → E → F → G

→ Algorithm

function BFS (problem) returns solution/failure

node ← a node w/ STATE = problem. INITIAL-STATE

pathcost ← 0

frontier ← a FIFO queue with node as the only element

explored ← an empty set

Loop do

(i) if EMPTY (frontier) - return failure

(ii) node ← POP(frontier)

(iii) node.state = EXPLORED

(iv) for each action in problem.ACTIONS(node.STATE)

child ← CHILDNODE(problem, node, action)

If CHID. STATE is not in explored or frontier THEN

if problem. GOAL-TEST (child. STATE) - return

SOLUTION(child)

else :

frontier ← INSERT (child, frontier)

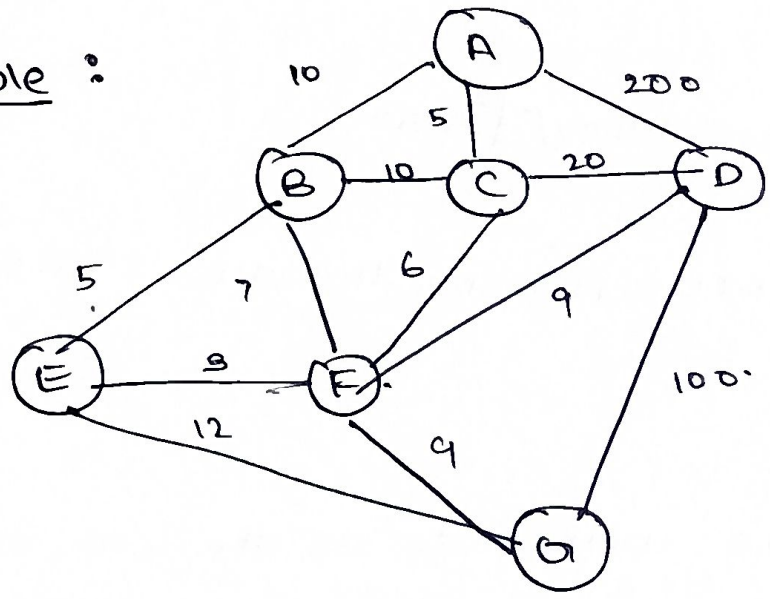
## * Strategy 2: Uniform Cost Search

→ expand the least cost unexpanded node

→ Implementation - queue ordered by path cost

→ equivalent to ~~BFS~~ BFS if path costs (step) are the same

→ Example :




From A-G

A → C → F → E → G

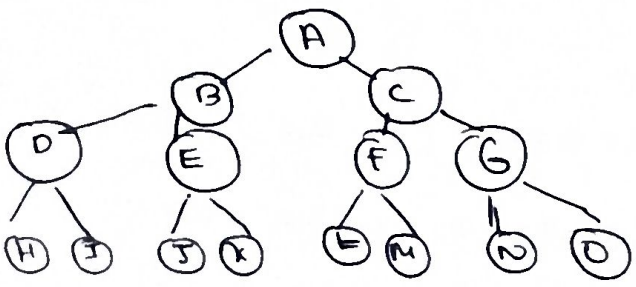
## * Strategy 3 : DFS

→ expand the deepest unexpanded node

→ Implementation : use a LIFO queue - put successors at front

aka STACK

→ Example : The order of traversal for the given tree would be:




A → B → D → H → I → E → J → K → C → F → L → M

→ G → N → O

## * Strategy 4: Depth- limited Search

→ DFS with a limit $l$, i.e nodes at level $l$ have no successors

Algorithm

function DLS(problem, limit) returns a solution, or failure/cutoff

return RECURSIVE DLS(makenode(problem.INITIAL STATE), problem, limit)

function RECURSIVEDLS(node, problem, limit) returns a solution or failure / cutoff

if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)

else if limit = 0 return cutoff

else

cutoff-occurred ← false.

for each action in problem.ACTIONS(node.STATE) do

child ← CHILDNODE(problem, node, action)

result ← RECURSIVE DLS(child, problem, limit-1)

if result = cutoff then cutoff-occurred ← true

else if result ≠ failure, return result

if cutoffoccurred? return cutoff, else return failure

## * Strategy 5: Iterative Deepening Search

It is like DLS, but after the limit, increase level by 1, and search again.

### Algorithm -

Function ITERATIVE_DEEPENING_SEARCH(problem) returns a solution, or

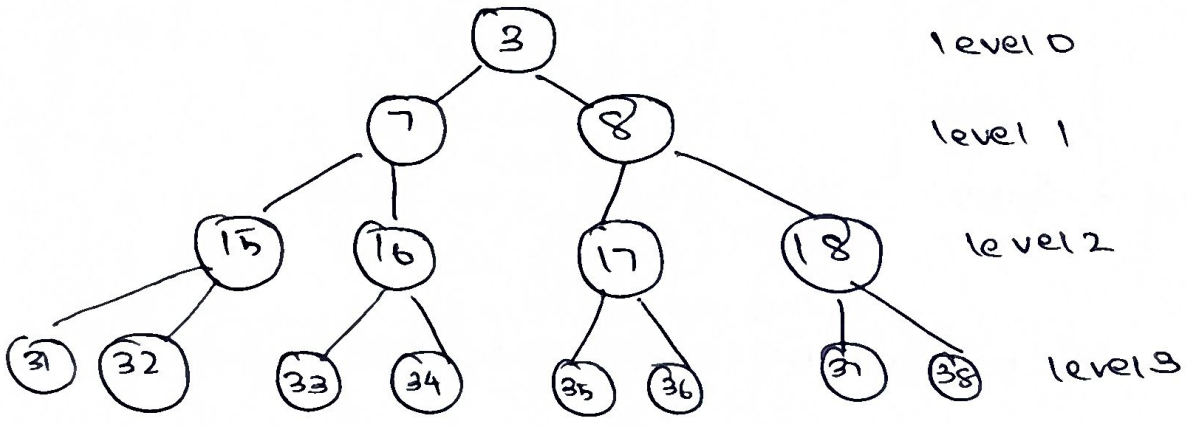
failure

for depth ← 0 to ∞ do

result ← DLS(problem, depth)

if result ≠ cutoff then return result

**Example** - For the given tree, the iterations would be:




level = 0 ⇒ 3

level = 1 ⇒ 3, 7, 8

level = 2 ⇒ 3, 7, 15, 16, 8, 17, 18

level = 3 ⇒ 3, 7, 15, 31, 32, 16, 33, 34, 8, 17, 35, 36

18, 37, 38

## * Strategy 6: Bidirectional Search

→ can move in either direction, top down / bottom up

## * Evaluation of Search Strategies

| Strategy / Criterion | BFS | Uniform cost | DFS | DLS | IDS | Bidirectional |
|---|---|---|---|---|---|---|
| Time | $O(b^d)$ | $O(b^{ceil(C^*/\varepsilon)})$ $C^*$ = cost of optimal solution. | $O(b^m)$ m = max depth (bad if $m \gg d$) | $O(b^l)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Space | $O(b^d)$ | $O(b^{ceil(C^*/\varepsilon)})$ | $O(bm)$ linear! | $O(bl)$ | $O(bd)$ | $O(b^{d/2})$ |
| Completeness | Yes (b is finite) | Yes | No (fails in $\infty$ depth) | No | Yes | Yes If b is finite both directions use BFS |
| Optimality | Yes* (cost = 1 per step) | Yes | No | No | Yes if step cost = 1 | Yes step costs are identical both directions use BFS |

## * Informed Search Strategies

→ uses problem-specific knowledge beyond the definition of the problem itself.

Best First Search - a variant of the general TREE/ GRAPH search, where a node is selected for expansion based on an evaluation function $f(n)$

→ The choice of $f$ determines the search strategy.

Heuristic Function - an estimated cost of the cheapest path from the state at node $n$ to a goal state.

→ It is a method by which additional knowledge of the problem is imparted to the search algorithm.

→ If n is a goal node, then $h(n) = 0$.

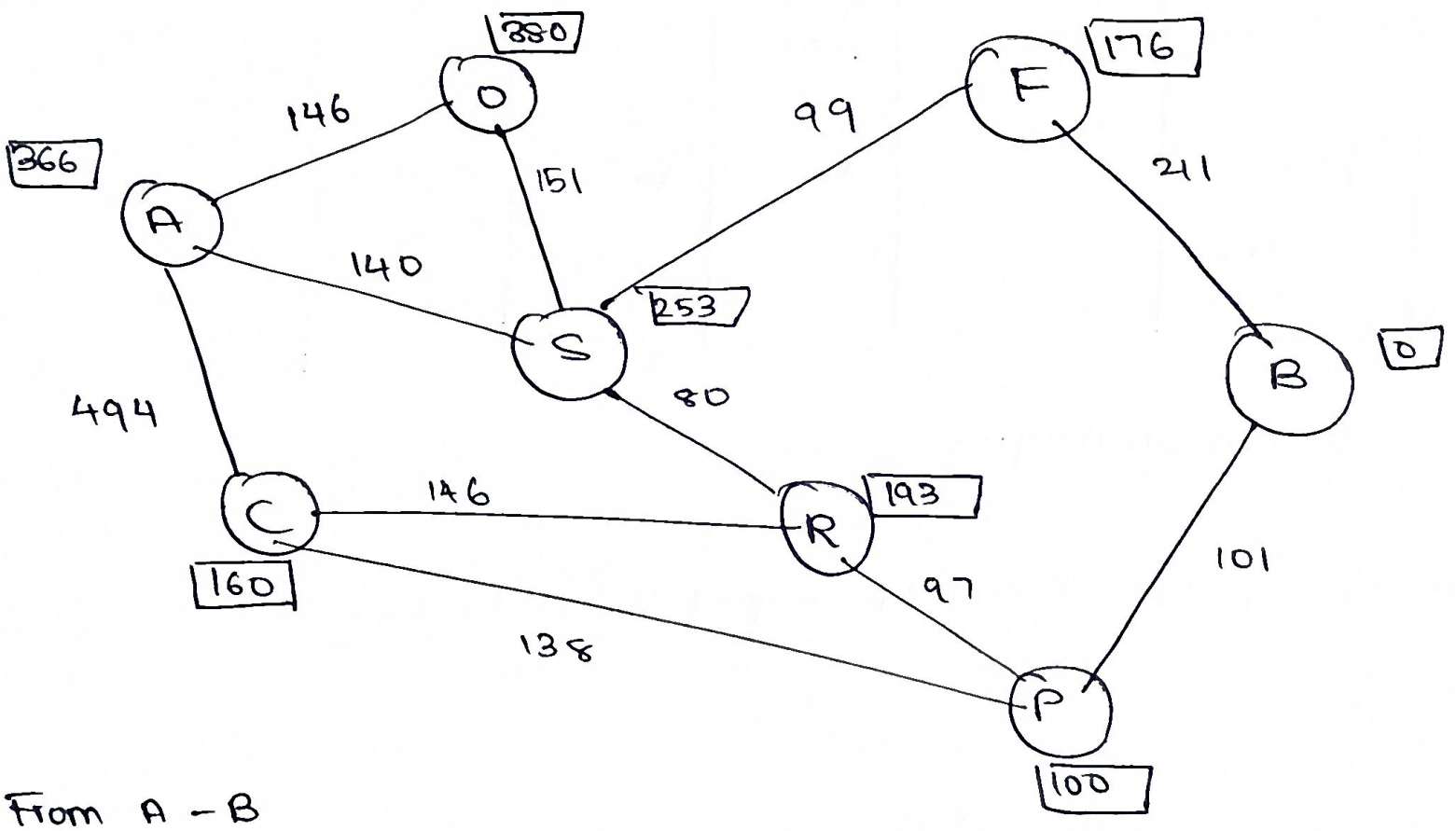
## * Informed Search Strategy 1 - Greedy BFS

→ Greedy BFS tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly.

→ Nodes are evaluated using just the heuristic function, i.e $f(n) = h(n)$

→ Straight line distance heuristic is used, represented as $h_{SLD}$.

Example: Consider the following graph.




From A - B

Initially - at A - min $f(n)$ value is that of C

$\Rightarrow$ A - C

at C, the min $f(n)$ is that of P

$\Rightarrow$ A - C - P

Thus, the path is | A-C-P-B |

## * Evaluation of Greedy BFS

Time Complexity : $O(b^m)$

Space Complexity : $O(b^m)$

m = maximum depth of the search space

Complete : No (after expanding a least cost path, may lead to a dead end)

Optimality : No (does not always find the shortest path)

## * Informed Search Strategy 2- A* Algorithm

→ Evaluates nodes by combining the cost to reach the node ($g(n)$) and the cost to get from the node to the goal ($h(n)$)

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

Example 1: Apply A* algorithm to move from the the source node (A) to destination node (J).

at node A: to node B : 14

to node F : 9

A → F

at node F:

to node G: (3+4) +5 = 9

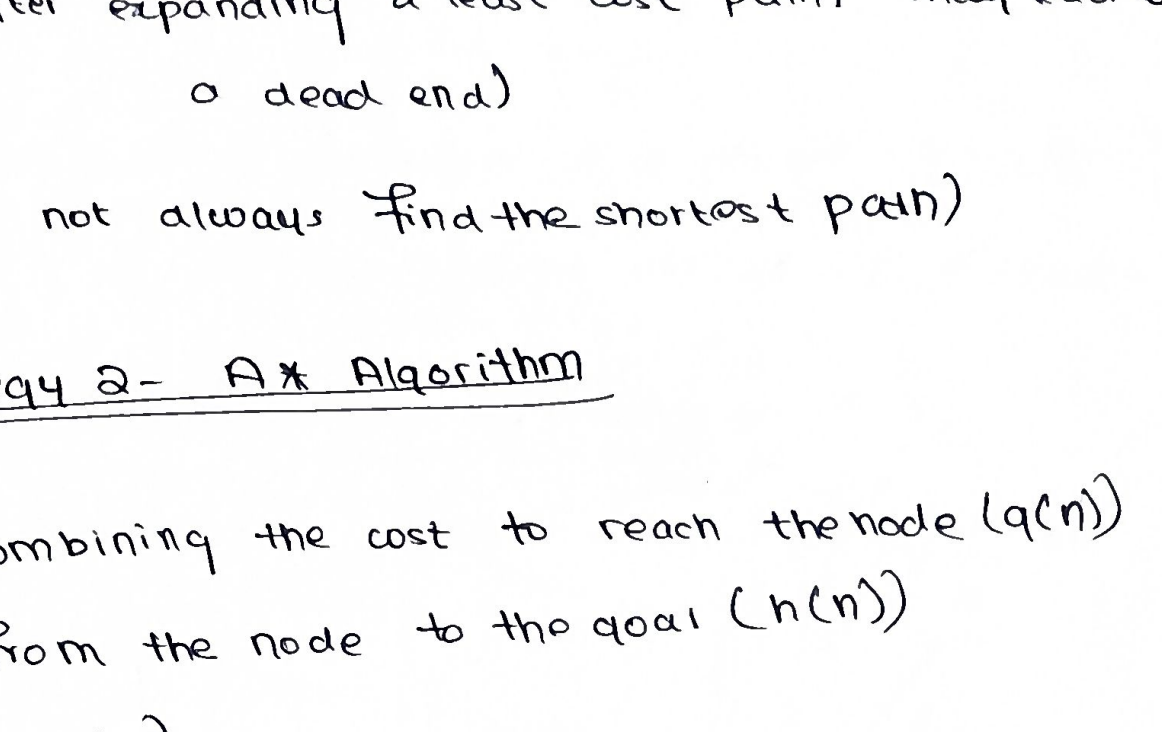
to node H : (3+7) + 3 = 13

A → F → G

at node G:

to node I: (3+1+3) + 1 = 8

A → F → G → I



at node I:

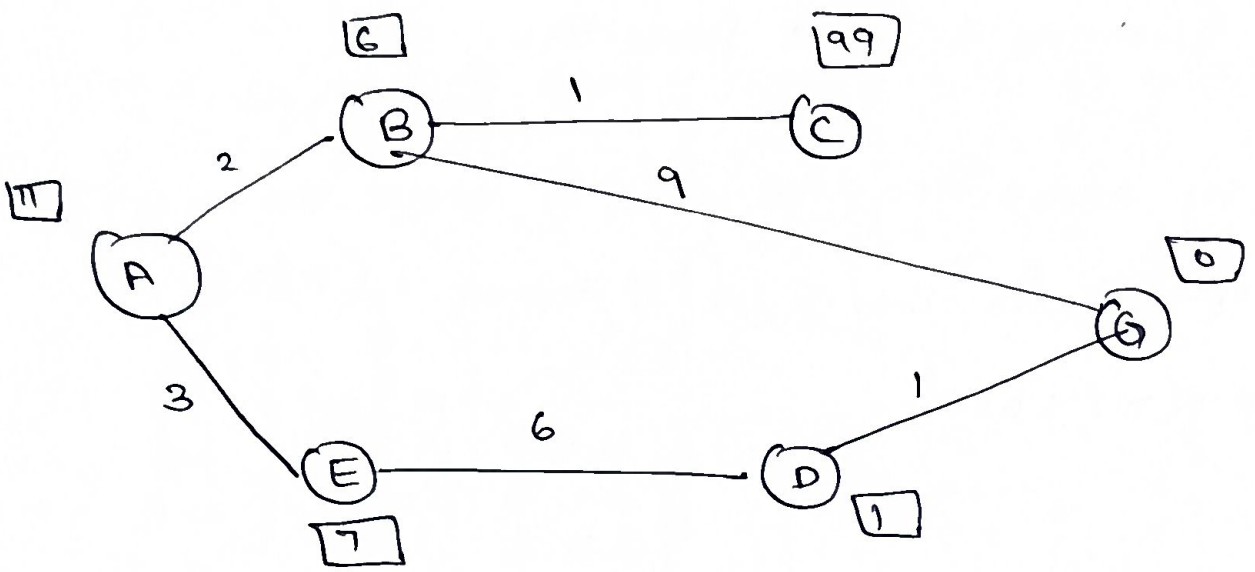
to H : (9) + 3 = 12

to E : (3+1+3+5) + 3 = 15

to J : (3+1+3+3) + 0 = 10

=> A → F → G → I → J

Example 2: Apply A * algorithm to find the path from the inital state(A) to the goal state (G).




at node A: to node B - 2+6 = 8

to node E - 3+7 = 10

go to B

A → B

at node B: to node C = 3+99 = 102

to node G = 11+0 = 11 → cost to go E go back & choose alt path

A → E

at node E: to node D = 3+6+1 = 10

A → E → D

at node D: to node G = 10+0 = 10

=> path

A → E → D → G

Exercise 3 : solving the 8 puzzle problem using the A* algorithm

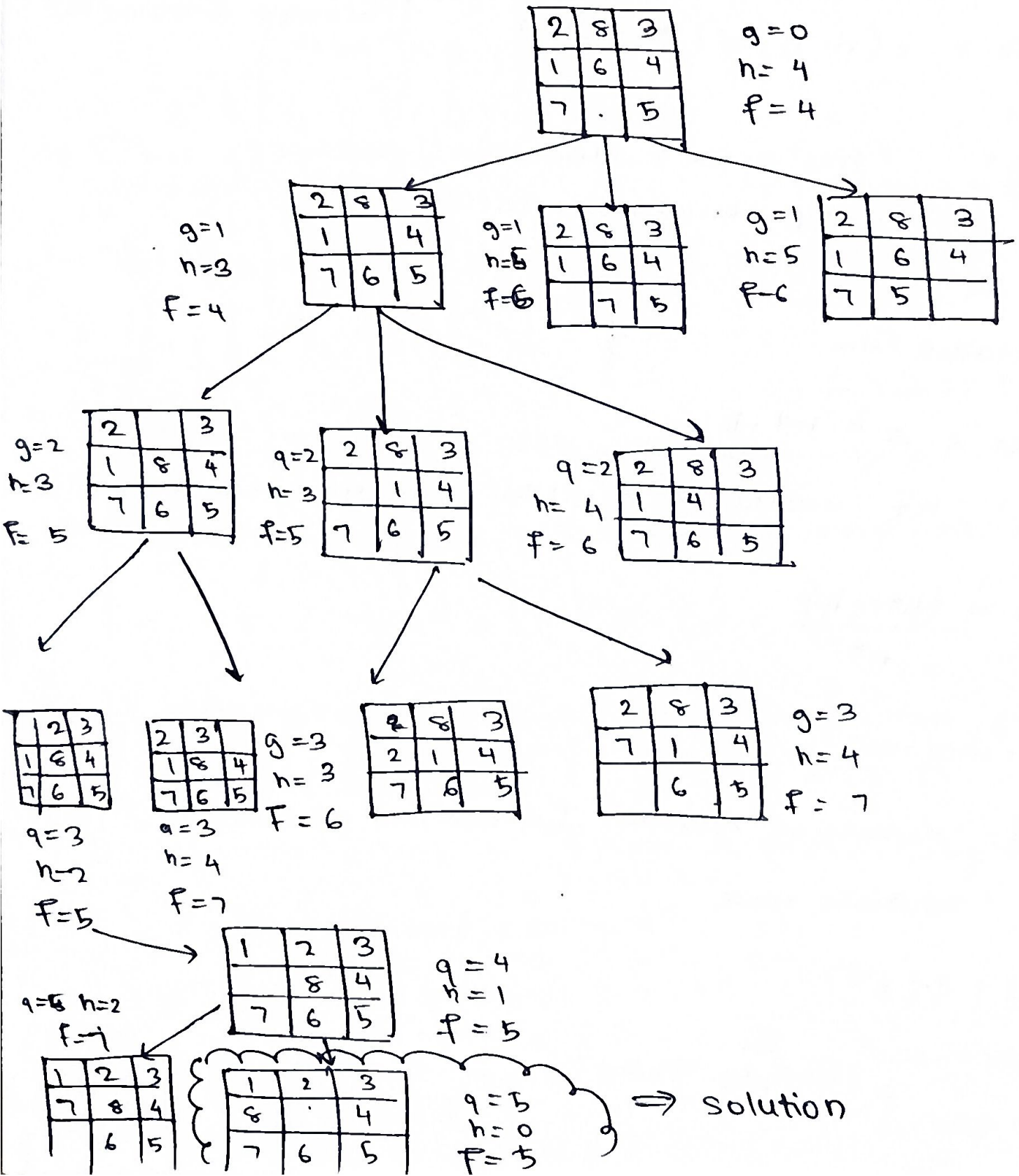
Initial State

| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| 7 | | 5 |

Final State

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

g(n) = depth of node h(n) = no. of misplaced tiles



## * Conditions for optimality : Admissibility and Consistency

A. **Admissible Heuristic** : An an admissible heuristic is one that never overestimates the cost to reach the goal.

B. **Consistent Heuristic** : A heuristic $h(n)$ is consistent if ~~for~~ every node $n$ and every successor $n'$ of $n$ generated by any action $a$, the estimated cost of reaching the goal from $n$ is no greater than the step cost of getting to $n'$ plus the estimated cost of reaching the goal ~~of~~ from $n'$ ie

$$h(n) \leq c(n, a, n') + h(n') \qquad \text{(Triangle Inequality)}$$

## * Optimality of A*

tree version = admissible

graph version = consistent

## * Absolute & Relative Error

Absolute error = $\Delta = h^* - h$

$h^*$ = actual cost of getting from the root to the goal

$$\text{Relative error} = \frac{(h^* - h)}{h^*}$$

## * A* Search Performance

Time Complexity - depends on heuristic function and admissible heuristic value

Space Complexity - $O(b^m)$

Optimality - yes

Completeness - yes

} for locally finite graphs

## * Generating and Evaluating Heuristics

→ A good heuristic function should never overestimate the number of steps to the goal.

→ In the case of the 8 puzzle problem - two commonly used heuristics are:

(i) number of misplaced tiles

(ii) Manhattan Distance - sum of distances of the tiles from their goal positions.

## * Effect of heuristic accuracy on performance

→ The quality of a heuristic can be the effective branching factor $b^*$.

→ For eg. using the A* algorithm for a particular problem, with a solution depth (d), with N+1 nodes would be:

$$N+1 = 1 + b^* + (b^*)^2 + \ldots (b^*)^d$$

→ A well-designed heuristic would have a value of $b^*$ close to 1.

## * Methods of Generating Admissible Heuristics

1. From relaxed problems
2. From subproblems
3. From experience.

## ① From relaxed problems

→ A problem with fewer restrictions on the actions is called a relaxed problem.

→ The state space graph of the relaxed ~~graph~~ problem is a supergraph of the original state space because the removal of restrictions creates added edges to the graph.

→ Because the relaxed problem adds edges to the state space, any optimal solution in the original problem is also a solution in the relaxed problem, but the relaxed problem may have better solutions if the added edges provide short cuts.

→ Hence, the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.

Example : Relaxed subproblems for the 8-puzzle problem.

Original problem definition = A tile can move from square A to square B if A is horizontally or vertically adjacent to B and B is blank.

Three relaxed subproblems =

(i) A tile can move from square A to B, if A is adjacent to B (Manhattan)

(ii) A tile can move from square A to B, if B is blank — The heuristic is that - if the blank is where the tile should be in the goal configuration, move the mismatched tile into the blank.

(iii) A tile can move from square A to B (misplaced tile heuristic)

(2) From subproblems

→ Store solution costs for every possible subproblem instance. - in pattern databases.

→ For disjoint pattern databases, consider the cost of solving the subproblem, without the influence of the other nodes.

Example - in the 8 puzzle problem, a subproblem can be get tiles 1, 2, 3, 4 into their correct positions, and store the solution cost in pattern databases.

(3) From experience

→ Optimal solutions to problems can be examples from which the $h(n)$ can be learned.

→ Learning algorithms can be used to construct a function $h(n)$ that can predict solution costs for other states that arise during search

→ Inductive learning methods work best when supplied with features of a state that are relevant to predicting the state's value.

→ If the features are $x_1(n)$ & $x_2(n)$, the heuristic can be calculated as

$$h(n) = c_1 x_1(n) + c_2 x_2(n)$$

→ Adjust $c_1$ & $c_2$ to get the best fit.

Example - for the 8 puzzle, experience means solving lots of such puzzles.

→ Features that can be used include:

$x_1(n)$ = no. of misplaced tiles

$x_2(n)$ = no. of pairs of adjacent tiles that are not adjacent in the goal state.

## * Beyond Classical Search - Local Search Algorithms and Optimization Problems

### * Local Search Algorithms

→ operate using just a single node, rather than multiple paths, and movement is generally to the neighbor of that node.

→ the paths followed by the search are not retained.

#### Advantages of Local Search Algorithms:

(i) use very little memory

(ii) can find reasonable solutions in large / infinite state spaces where systematic algorithms are unsuitable

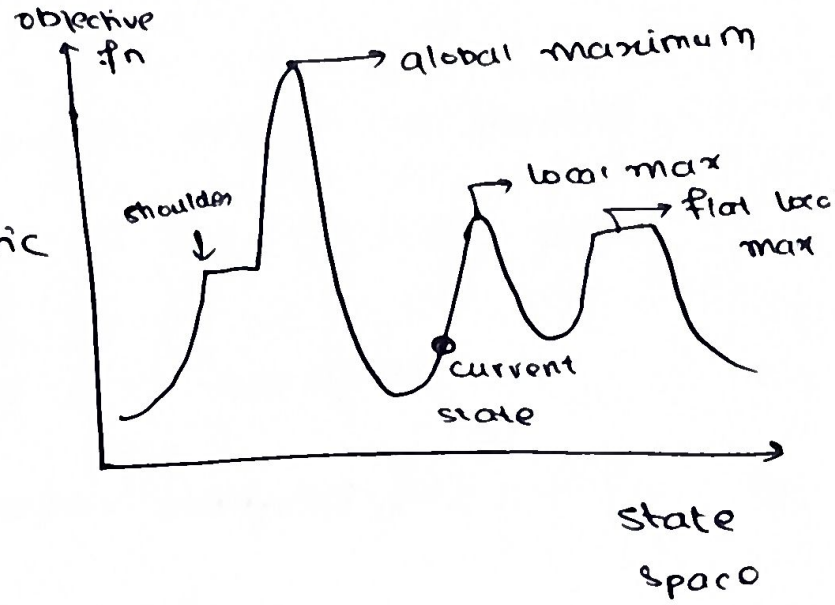
(iii) useful for solving pure optimization problems, where the aim is to find the best state according to an objective function.

## * State Space Landscape

→ A state space landscape has two major components:

(i) location - defined by the state

(ii) elevation - defined by the heuristic cost function or objective function.




→ If the elevation corresponds to cost -

find the lowest valley = global minimum

→ If the elevation corresponds to the objective fn

find the highest peak = global maximum.

## * Complete vs. Optimal Local Search Algorithm

complete local search algorithm - always finds a goal if one exists

optimal local search algorithm - always finds a global minimum/maximum.

## * Local Search Algorithm 1 - Hill Climbing

→ most basic local search technique

→ at each step, the current node is replaced by the best neighbor, which if:

(i) neighbor w/ the highest value

(ii) or lowest h, if a heuristic cost estimate is used.

→ a loop that moves in the direction of increasing value - uphill

## Algorithm

function HILL-CLIMBING (problem) returns a state that is a local maximum.

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

## Example of Hill Climbing - 8 queens problem

→ generate a complete-state formulation, where each state has 8 queens on the board, one per column.

→ generate successors by moving a single queen to another square in the same column.

→ The heuristic cost function is the number of attacking pair of queens.

→ Stop the hill climbing algorithm as soon as a local minimum is obtained.

## * When does the hill climbing algorithm get stuck?

→ The hill climbing algorithm gets stuck ~~fo~~ the following reasons:

(i) local maxima: a peak higher than the neighboring states but lower than the global maximum

(ii) Ridges: a sequence of local maxima that is very difficult to greedy algorithms to navigate.

(iii) Plateau - a flat area of the state space landscape. It may be a flat local maximum, from which no uphill exit exists, or a shoulder from which progress is possible. Sometimes, a hill-climbing search gets lost on the plateau.

## * How can the hill climbing algorithm navigate plateaus?

→ can allow sideways moves in the hope that the plateau is really a shoulder.

→ sideways moves are allowed when there are no uphill moves an infinite loop occurs whenever the algorithm reaches a flat local maximum that is not a shoulder.

→ Put a limit on the number of consecutive sideways moves allowed, so that an infinite loop does not occur.

## * Variants of Hill Climbing?

### ① Stochastic Hill Climbing?

→ choose at random from among the uphill moves

→ probability of selection varies with steepness.

### ② First choice hill-climbing?

→ implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.

→ A good strategy when a state has thousands of successors.

### ③ Random-restart hill climbing?

→ conduct a series of hill-climbing searches from randomly generated initial states, until a goal is found.

→ If each hill-climbing search has a probability p of success, then the number of expected restarts required is 1/p.

→ Works well when there are a few local maxima and plateaus

## * Local Search Algorithm -2 - Simulated Annealing?

→ A hill-climbing algorithm that never makes a downhill moves is incomplete, because it can get stuck on a local maximum

→ In contrary, purely random walks, would be complete, but very inefficient.

→ Combine hill climbing with a random walk, to yield both efficiency and completeness

→ Simulated Annealing - based off of the metallurgical process of annealing - used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the metal to reach a low energy crystalline state

→ The simulated annealing solution is to start by shaking hard at a high temperature, then gradually reducing the intensity of the shaking at lower temperatures.

Algorithm

→ basically a version of stochastic hill climbing, where some downhill moves are allowed.

→ Downhill moves are accepted readily early in the annealing schedule, and then less often as time goes on.

→ The schedule input determines the temperature T as a function of time.

function SIMULATED-ANNEALING (problem, schedule) returns soln. state
inputs: problem, a problem
schedule, a mapping from time to temperature.

current ← MAKE-NODE (problem. INITIAL-STATE)

for t = 1 to ∞ do

T ← schedule(t)

if T = 0, return current

next ← a randomly selected successor of current

ΔE ← next.value - current.value

if ΔE > 0, then current ← next

else current ← next, only with probability $e^{\Delta E/T}$

## * Local Search Algorithm - 3 - Local Beam Search

→ The local beam search keeps track of k states.

→ Begin with k randomly generated states

→ At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts

→ Otherwise, it selects the k-best successors from the complete list and repeats.

→ Not like random-restart search, where each search process runs independently.

→ In local beam search, useful info is passed among the parallel search threads.

## * Local Search-Algorithm - 4 - Stochastic Beam Search

→ Local beam search lacks diversity among the k states - becomes concentrated in a small region of space.

→ Stochastic beam search alleviates this problem

→ Instead of choosing the k best from the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing successor being an increasing function of its value.

## * Genetic Algorithms

→ A variant of stochastic beam search in which successor states are generated by combining two parent states rather than modifying a single state.

→ It iteratively enhances a population of potential solutions by simulating the natural evolution process - including mutation, crossover and selection

→ The process continues until a satisfactory solution or maximum number of generations is reached.

## * Steps Involved in Genetic Algorithm

1. Initialization: Create a randomized population of potential solutions
2. Fitness Evaluation: Calculate each potential solution's fitness
3. Selection: Select parents from the current population to make the next generation
4. Crossover: Perform crossover between pairs of parents to create new offspring.

5. Mutation - Bing minor changes or mutations to the existing offspring solutions to maintain diversity and explore the ~~state~~ search space.

6. Replacement - Replace the current population with the new generation of offspring solutions

7. Termination - Repeat steps 2-6 until a termination condition is met.

Example : Genetic Algorithm on the N- Queens Problem

1. Represent the position of each queen in a binary string (3 bit binary 0-7)

2. Rate each state by the objective function (the fitness fn).

3. The fitness function should return higher values for better states

⇒ non attacking pairs is used as the fitness. It has a value of 28 for a solution

4. Probability of being chosen for reproduction of fitness score.

5. Select 2 pairs - and a random position in them as the crossover point.

6. The crossover operation can produce a state that is quite different from both parents.

=> Crossover frequently takes large steps in the state space early and smaller steps later on.

Algorithm

function GENETIC ALGORITHM ( population, fitness fn ) RETURNS an individual

input : population

FITNESS-FN

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 the population, according to the FITNESS FN

function REPRODUCE (x,y) returns an individual

input : x,y, parents

n ← length

c ← random no. from 1 to n

return APPEND( SUBSTRING(x,1,c), SUBSTRING(y,c+1,n))

## *Adversarial Search

*Deterministic - Turn-Taking Games - Two agents act alternately in which the utility values at the end of the game are always equal and opposite.

→ For ea. if one player wins a game of chess, the other player necessarily loses.

→ It is this opposition between the agents' utility functions that makes the situation adversarial.

## * Formulation of a Game-Tree

→ A game can be formally defined as a kind of search problem with the following elements:

(i) $S_0$ - initial state - specifies how the game is set up at the start

(ii) PLAYER(s) - defines which player has the move in a state

(iii) ACTIONS(s) - returns the set of legal moves in a state

(iv) RESULT(s,a) : the transition model, which defines the result of a move

(v) TERMINAL-TEST(s) - A terminal test is true when the game is over, and false otherwise. States where the game has ended are called terminal states.

(vi) UTILITY (s, p) - A utility function - also called the objective / payoff function defines the final numeric value for a game that ends in terminal state s for a player p.

eg. in chess : W = 1
L = 0
Draw = 1/2

*Zero Sum Game - A game in which the total payoff to all players is the same for every instance of the game.

→ Chess is zero-sum because every game has a payoff of either 0+1, 1+0 or 1/2 + 1/2

*Minimax (n)

→ The minimax value of a node is the utility (for MAX) of being in the corresponding state, assuming that both players play optimally from there to the end of the game.

→ The minimax value of a terminal state is just its utility.

→ Given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

## * Minimax Algorithm

→ The minimax algorithm performs a complete depth-first exploration of the game tree.

→ The whole game tree is generated to the leaves

→ Apply the utility function to the leaves.

→ Use DFS for expanding the tree

→ Back up values from leaves towards the root

(i) A MAX node computes the max value from its child values

(ii) A MIN node computes the min value from its child values.

function MINIMAX-DECISION(state) returns an action

return $\underset{a \in ACTIONS(s)}{\text{argmax}}$ MIN-VALUE (RESULT(state, a))

function MAX-VALUE(state) returns a utility value

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

$v \leftarrow -\infty$

for each a in ACTIONS (state) do

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

return v

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

If TERMINAL-TEST (state) then return UTILITY (state)

v ← ∞

for each a in ACTIONS (state) do

    v ← min( v, MAX-VALUE (RESULT (s,a)))

return v
```

## * Performance of Minimax Algorithm

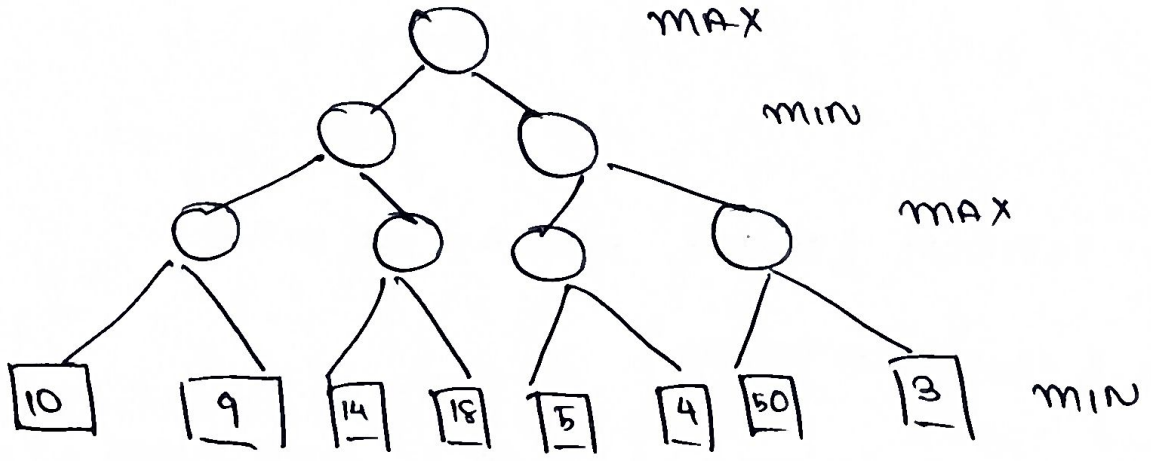
Time Complexity = $O(b^m)$ — maximum depth = m; b = no. of legal moves at each point

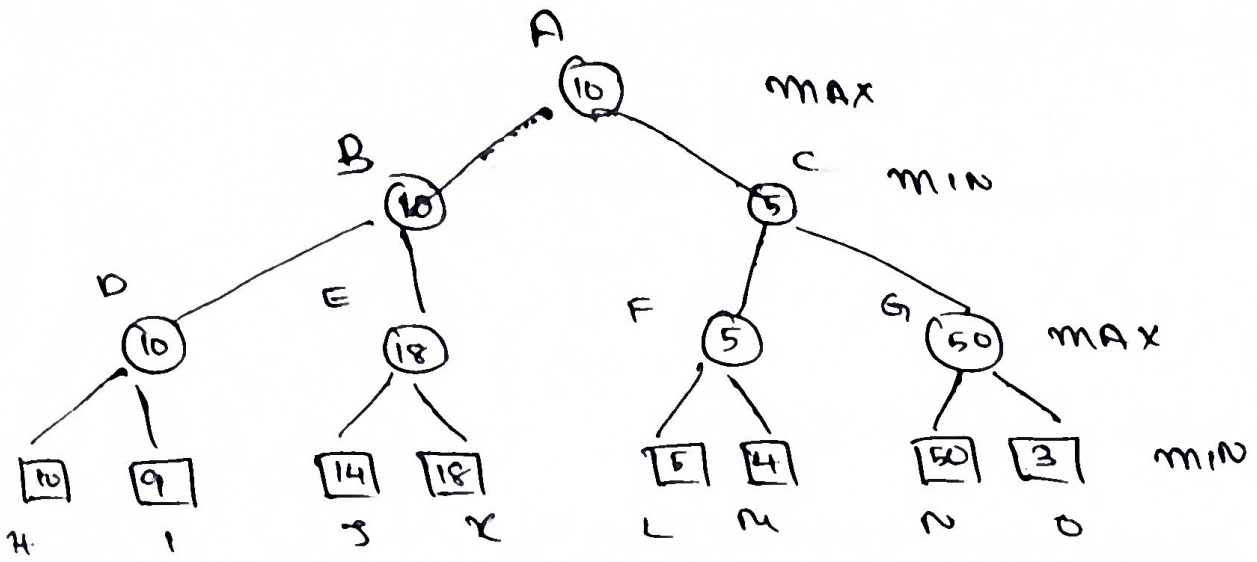
Space Complexity = $O(bm)$ → If all actions are generated at once

(or)

$= O(m)$ → actions are generated one at a time.

Example - Find the value of the root node and the optimal path using the Minimax Algorithm.






To find optimal path - follow path of 10

A → B → D → 11

## * Alpha - Beta Pruning :

Drawback of minimax: The no. of game states to explore is exponential in the depth of the tree.

→ It is possible to compute the correct minimax decision without looking at every node in the game tree.

→ $\alpha$-$\beta$ pruning returns the same move as minimax would, but prunes away branches that cannot possibly affect the final decision.

Algorithm

Function ALPHA-BETA-SEARCH (state) returns an action

v ← MAXVALUE (state, $-\infty$, $\infty$)

return the action in ACTIONS (state) with value v

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

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

v ← -∞

for each a in ACTIONS (state) do

        v ← MAX ( v, MIN VALUE (RESULT (s,a), α, β))

        if v ≥ β return v

        α ← MAX (α, v)

return v
```

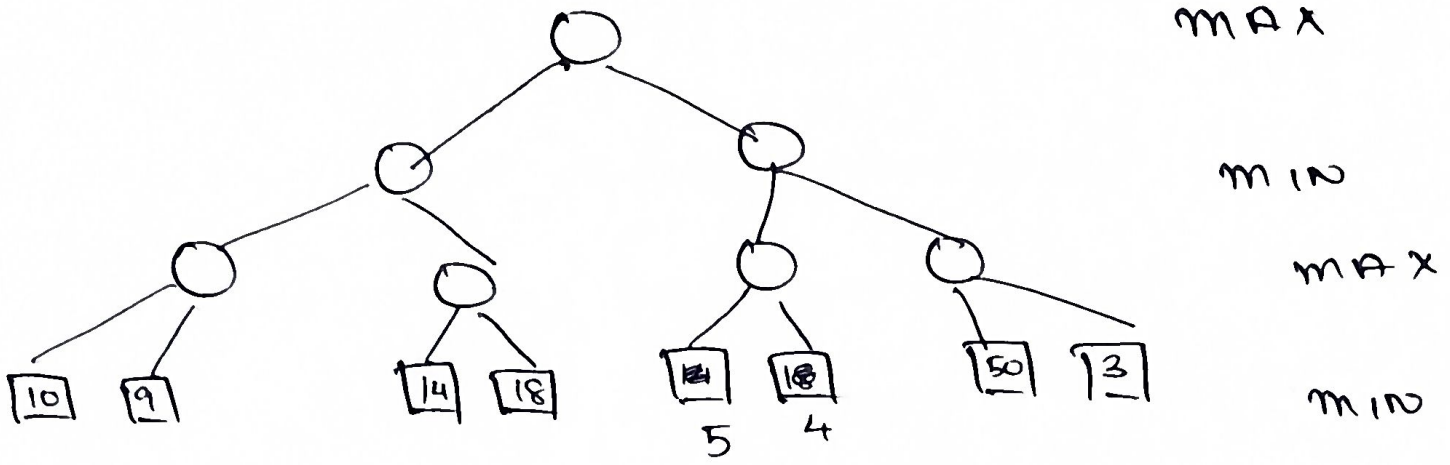
```
function MIN-VALUE (state, α, β) returns a utility valu

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

v ← +∞

for each a in ACTIONS (state) do

        v ← MIN (v, MAX-VALUE( RESULT (s,a) , α, β))

        if v ≤ α return v

        β ← MIN (β, v)

return v
```

* Performance

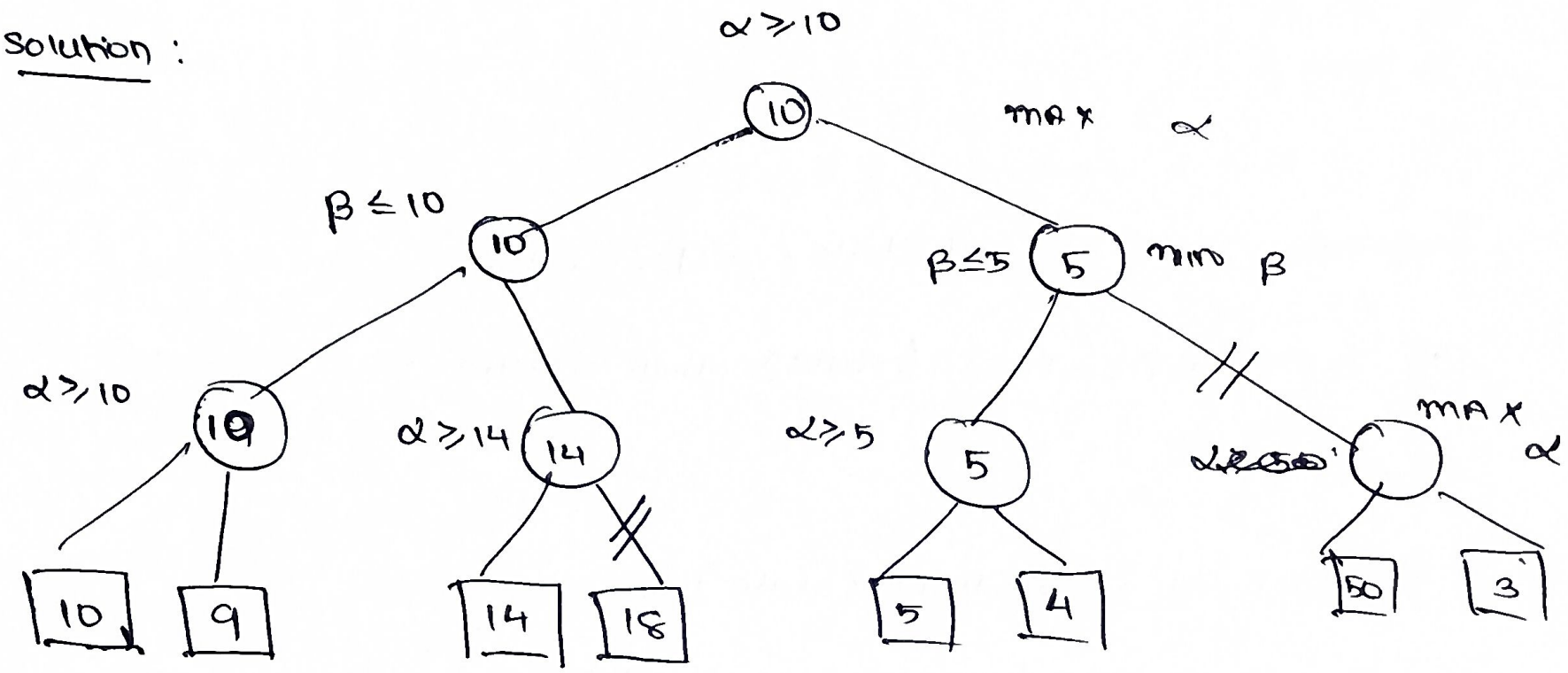
Time Complexity- $O(b^{m/2})$ if the best move is picked (highly dependent on the order in which states are examined)

Space Complexity - $O(bm)$

**Example1** Apply α-β pruning to find the root node value and the optimal path

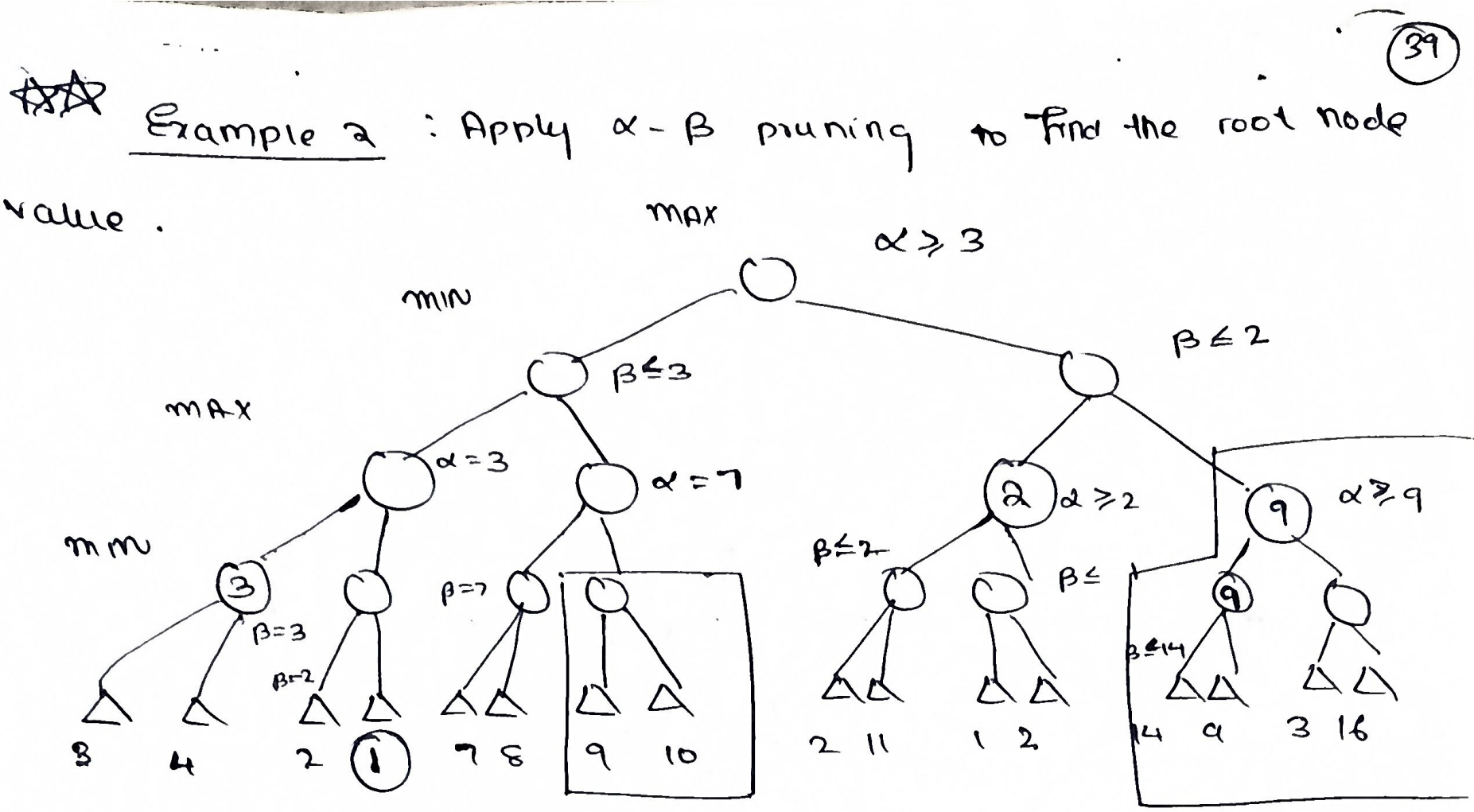



**Solution:**




~~check 10 ≥ 9 , 10 is max - move to parent~~

~~check α > β if so prune (for MAX)~~



## Example 2 : Apply α - β pruning to find the root node value.



Level 3 - out of (3, 4) $\beta = 3$, parent $\alpha = 3$

out of (2, 1) $\beta = 2$, check ~~right~~ box with parent

$\alpha > \beta$ prune RHS

out of (7, 8) $\beta = 7$, propagate up

check with parent $\alpha > \beta$ prune RHS

out of (2, 11) make $\beta = 2$

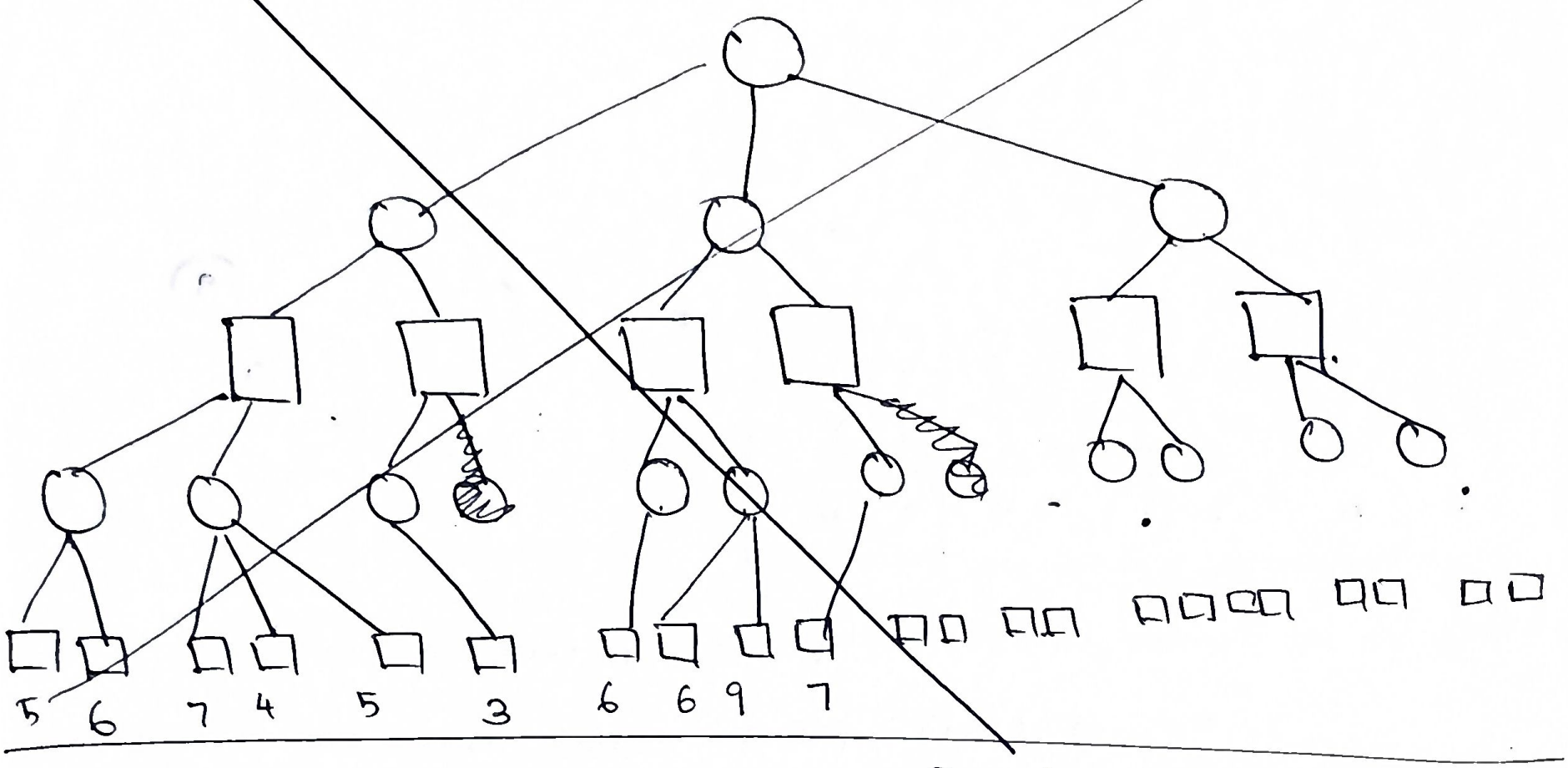
check RHS

check top 2 nodes $\beta \leq 2$ & $\beta \leq 3$

choose LHS

⇒ **root node = 2**

Example 3 - Find the value of the root node using α-β pruning




Example3 - Apply α-β pruning to find the value of the root node