Artificial Intelligence and Neural Network

BE Software Engineering

[Fifth Semester]

Nepal College of Information Technology

POKHARA UNIVERSITY

4. Inference and Reasoning

4.1 Inference Theorems

4.2 Deduction and truth maintenance

4.3 Heuristic search state-space representation

4.4 Game Playing

4.5 Reasoning about uncertainty probability

4.6 Bayesian Networks

4.7 Case-based Reasoning

SEARCHING

- Searching is the process of finding the required states or nodes.
- Searching is to be performed through the state space.
- Search process is carried out by constructing a search tree.
- Search is a universal problem-solving technique.

State Vs Node

- A state is a (representation of) a physical configuration
- Nodes in the search tree are data structures maintained by a search procedure representing paths to a particular state
- The same state can appear in several nodes if there is more than one path to that state
- The path can be reconstructed by following edges back to the root of the tree

SEARCH TERMINOLOGY

- **Problem Space**: Environment in which the search takes place.
- **Problem Instance:** It is Initial state + Goal state
- **Problem Space Graph:** It represents problem state. States are shown by nodes and operators are shown by edges.
- **Depth of a problem:** Length of a shortest path or shortest sequence of operators from Initial State to goal state.

Well-defined problems

A problem can be defined by:

- Initial state (problem state)
- Actions (Using successor functions)
- Goal test (To determine the goal state)
- Path Cost

A problem when defined with these components is called a well defined problem.

The actions/rules should be defined in as general way as possible

SEARCH TERMINOLOGY

Space Complexity: The maximum number of nodes that are stored in memory.

Time Complexity: The maximum number of nodes that are created.

Admissibility: A property of an algorithm to always find an optimal solution.

Branching Factor: The average number of child nodes in the problem space graph.

Evaluation of Search Strategies

→ A search strategy is defined by picking the order of node expansion

→ Strategies are evaluated along the following dimensions:

- ***Completeness***: does it generate to find a solution if there is any?
- ***Optimality***: does it always find the highest quality (least-cost) solution?
- ***Time complexity***: How long does it take to find a solution?
- ***Space complexity***: How much memory does it need to perform the search?

→ Time and space complexity are measured in terms of

- *b:* maximum branching factor of the search tree
- *d:* depth of the least-cost solution
- *m*: maximum depth of the state space (may be ∞)

Classification of Search Techniques

Uninformed Search or Blind Search

- ✓ Depth First Search
- ✓ Breath First Search
- ✓ Depth Limit Search
- ✓ Iterative deepening
- ✓ Uniform Cost
- ✓ Bidirectional Search

Informed search or Heuristic search

- ✓ Best first Search
  - Greedy Best First Search
  - A* Search

Local search Algorithms

- ✓ Hill climbing Search
- ✓ Simulated Annealing

Adversarial Search

- ✓ The minimax algorithm
- ✓ Alpha-beta pruning

Uninformed Search (Blind Search/Brute force search)

- The search algorithms that do not use any extra information regarding the problem are called the blind searching or Brute force searching or uninformed searching.
- These can only expand current state to get a new set of states and distinguish a goal state from non-goal state.
- These searches typically visit all of the nodes of a tree in a certain order looking for a pre-specified goal.
- These type of searching are less effective than informed search.

DEPTH FIRST SEARCH (DFS)

- Proceeds down a single branch of the tree at a time
- Expands the root node, then the left most child of the root node
- Always expands a node at the deepest level of the tree
- Only when the search hits a dead end (a partial solution which can't be extended), the search backtrack and expand nodes at higher levels.

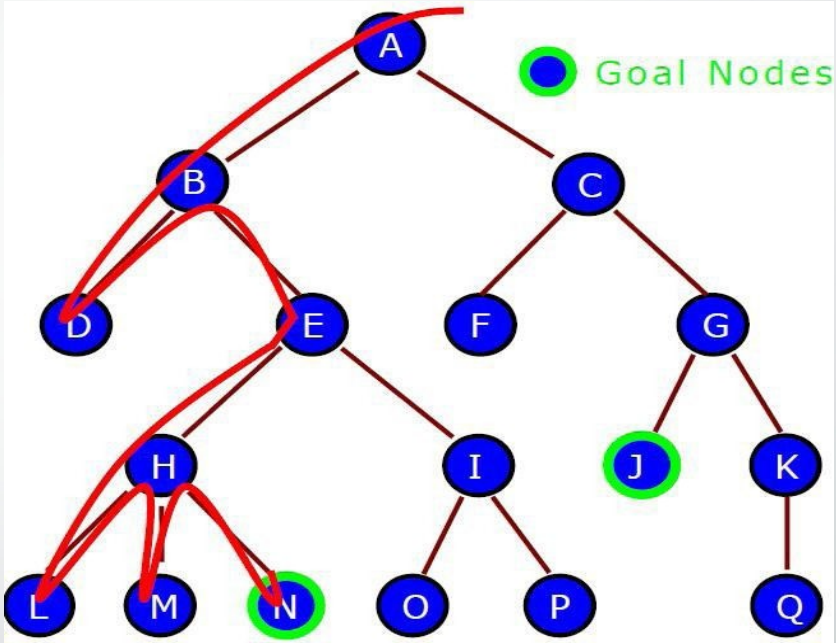


Fig. Depth-first search (DFS)

DEPTH FIRST SEARCH (DFS)

Algorithm

- Put the start node on the stack

- While stack is not empty

    a. Pop the stack

    b. If the top of stack is the goal, stop

    c. Other wiser push the nodes connected to the top of the stack on the stack (provided they are not already on the stack)

DEPTH FIRST SEARCH (DFS)

- ***Completeness***: Incomplete as it may get stuck going down an infinite branch that doesn't leads to solution.
- ***Optimality***: The first solution found by the DFS may not be shortest.
- ***Space complexity***: b as branching factor and d as tree depth level, Space complexity = $\mathbf{O(b^{d+1})}$
- ***Time Complexity***:

$$\mathbf{b + b^2 + b^3 + \ldots\ldots\ldots\ldots +}$$

$$\mathbf{b^d + b^{d+1} = O\,(b^{d+1})}$$

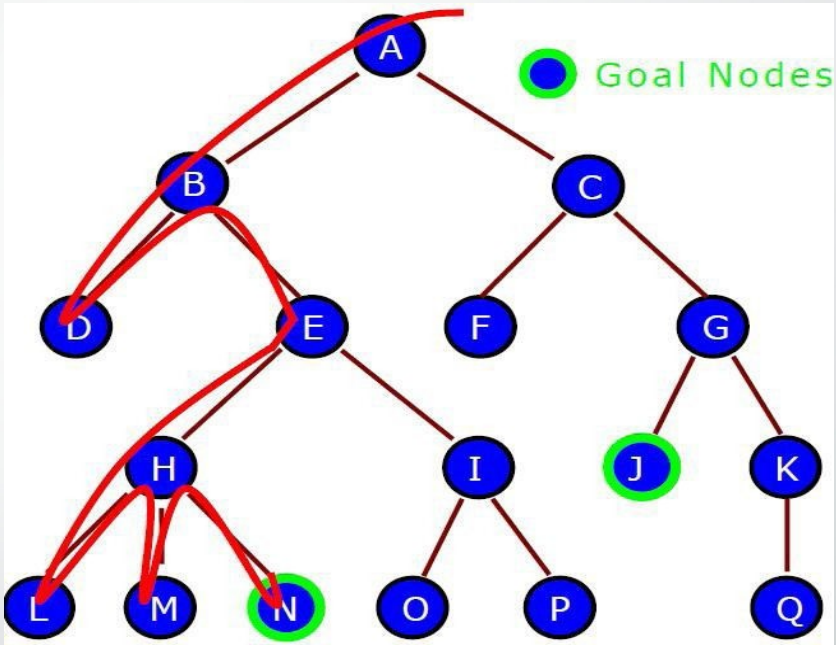
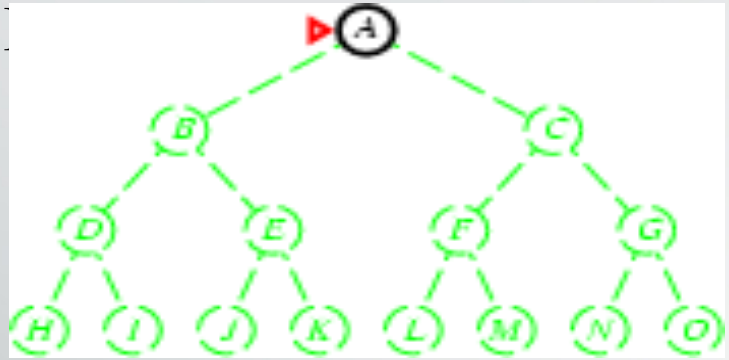
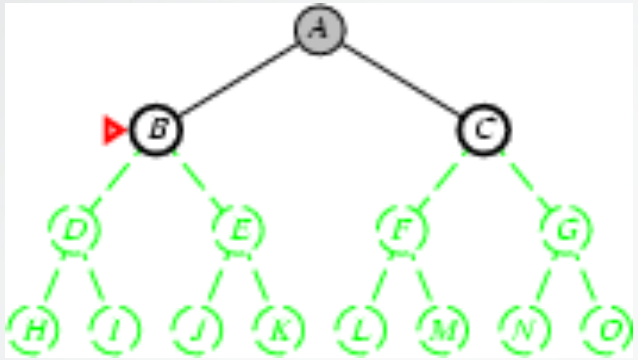
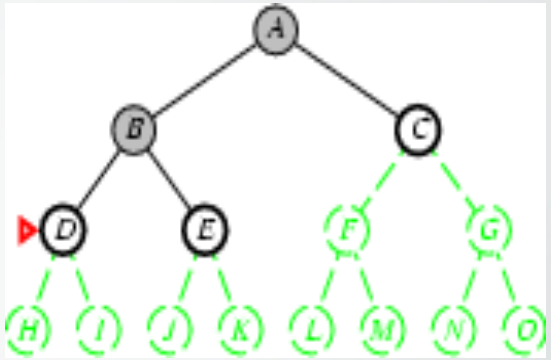


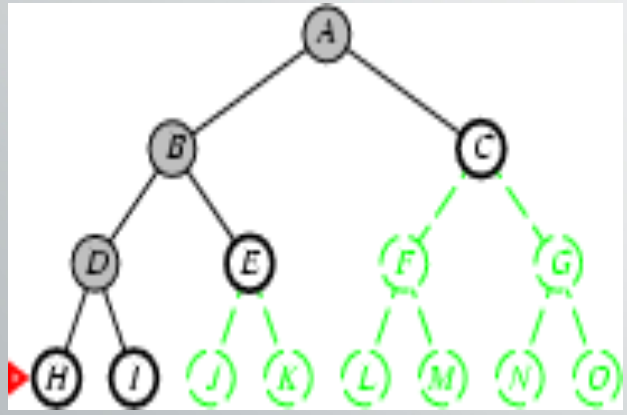
Fig. Depth-first search (DFS)

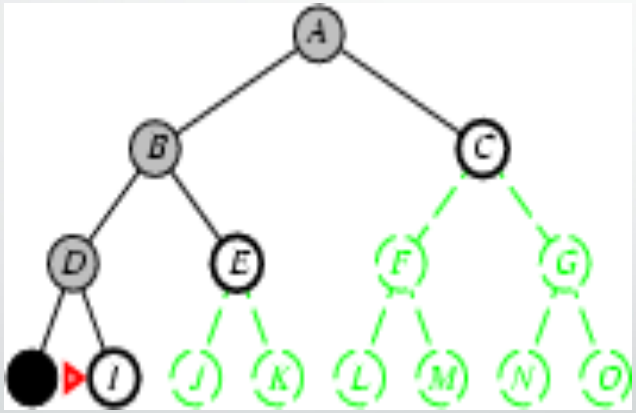
DFS example (find path from A to

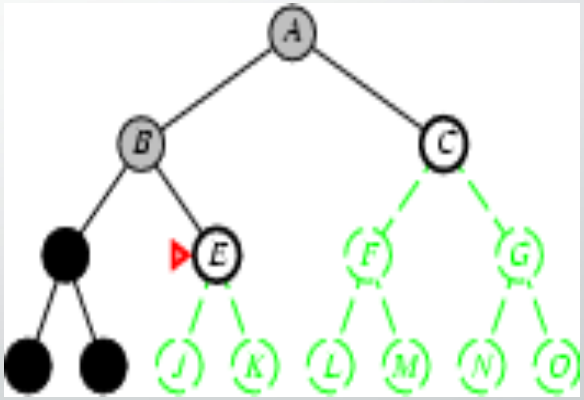












Depth First Search

→ **Advantages**

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

→ **Disadvantages**

- Can sometimes fail to find a solution;
- Can take a lot longer to find a solution.

Breadth-First Search (BFS)

- Proceeds level by level down the search tree
- Starting from the root node (initial state) explores all children of the root node, left to right
- If no solution is found, expands the first (leftmost) child of the root node, then expands the second node and so on

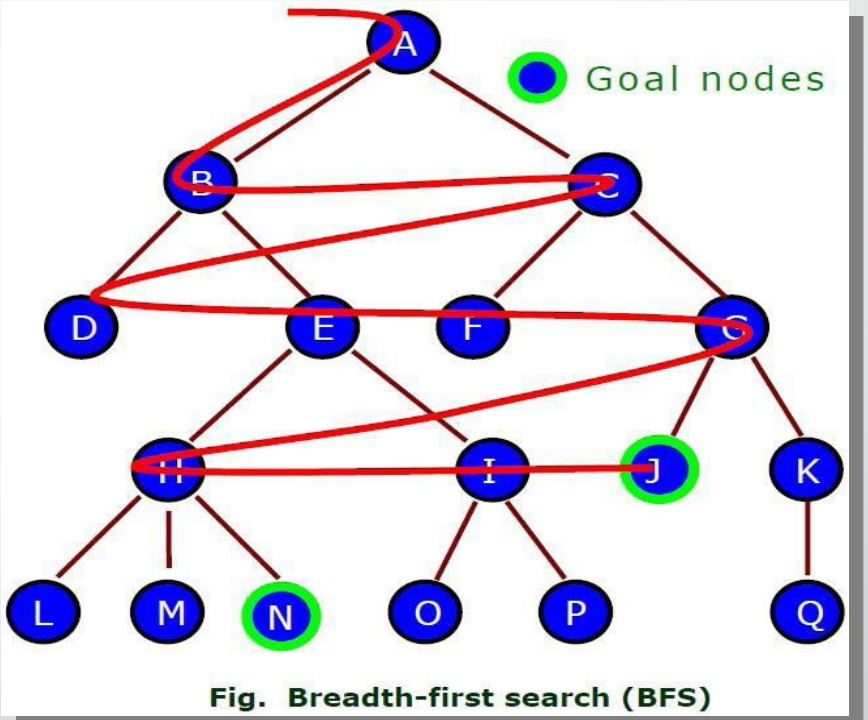


Fig. Breadth-first search (BFS)

Breadth-First Search (BFS)

Algorithm

- Place the start node in the queue

- Examine the node at the front of the queue

a. If the queue is empty, stop

b. If the node is the goal, stop

c. Otherwise, add the children of the node to the end of the queue

Breadth-First Search (BFS)

- Completeness: Complete if the goal node is at finite depth
- Optimality: It is guaranteed to find the shortest path
- Time Complexity:

  $b + b^2 + b^3 + \ldots\ldots\ldots\ldots\ldots = O(b^m)$
- Space Complexity:

  (1+ b + b + b + ...b) m times = O(b m)

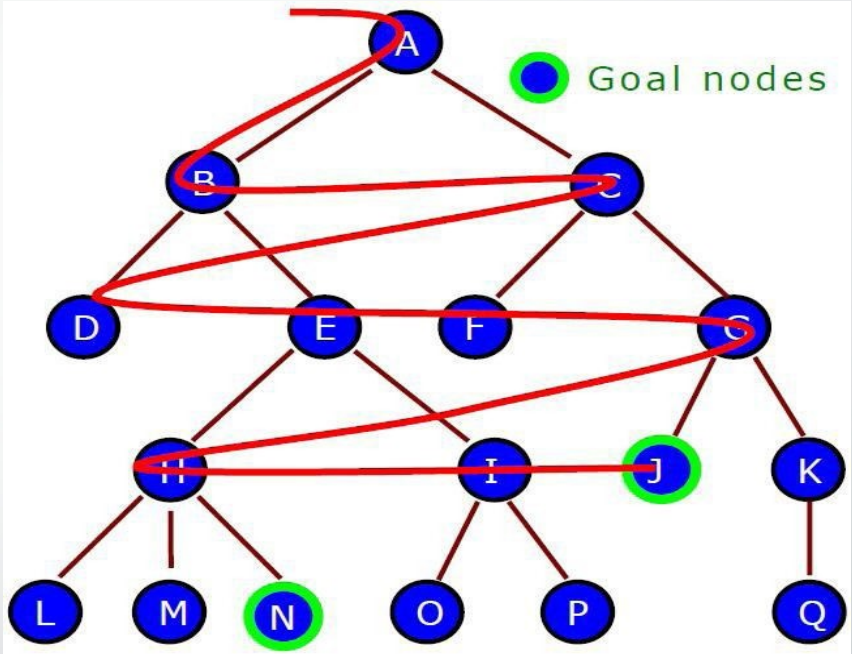
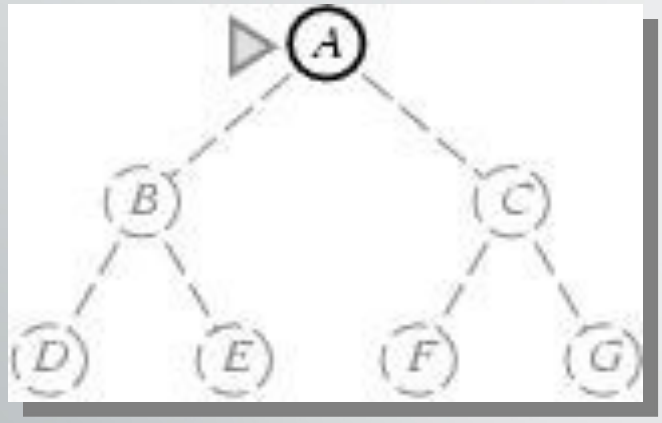
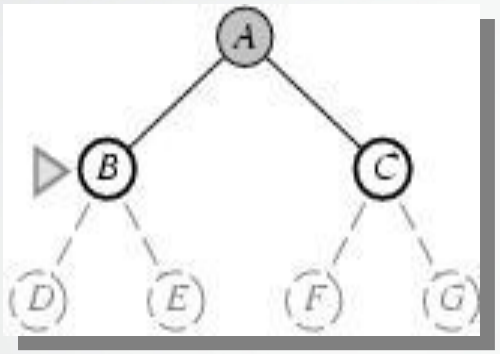
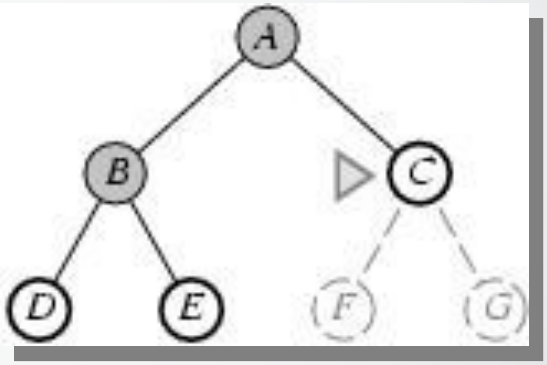


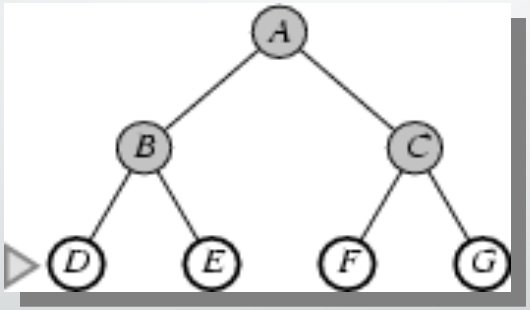
Fig. Breadth-first search (BFS)

BFS example (Find path from A to D)









BFS example (Find path from A to D)

→ **Advantages**

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

→ **Disadvantages**

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

Uniform-Cost Search

Uniform-cost is guided by path cost rather than path length like in BFS, the algorithms starts by expanding the root, then expanding the node with the lowest cost from the root, the search continues in this manner for all nodes.

- **Completeness:**

  Complete if the cost of each step exceeds some small positive integer, this to prevent infinite loops.

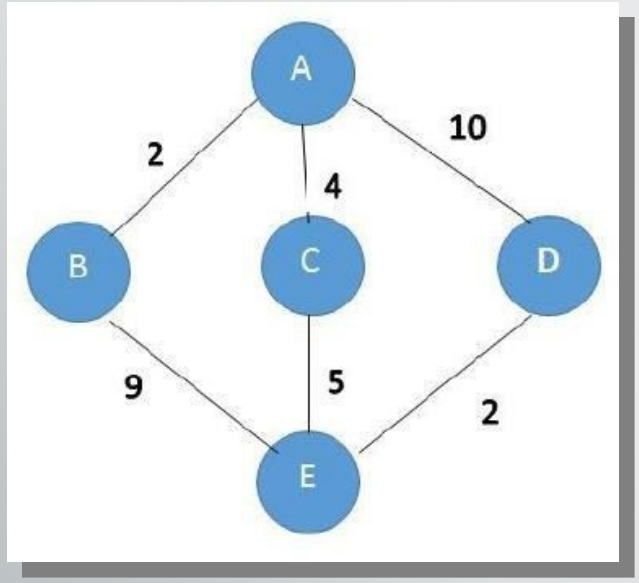
- **Optimality:**

- Optimal in the sense that the node that it always expands is the node
- with the least path cost.

  Time Complexity: $O(b^{C/e})$

  Space Complexity: $O(b^{C/e})$

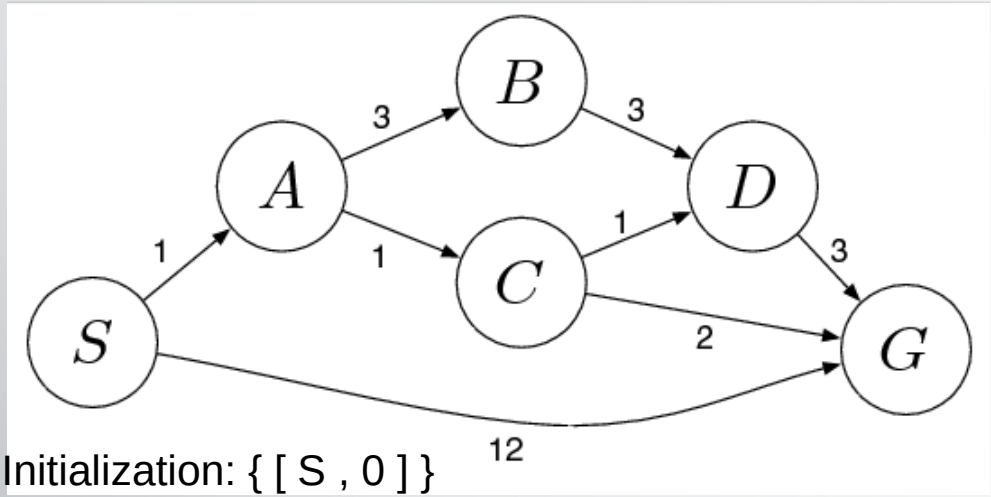
UCS example (Find path from A to E)

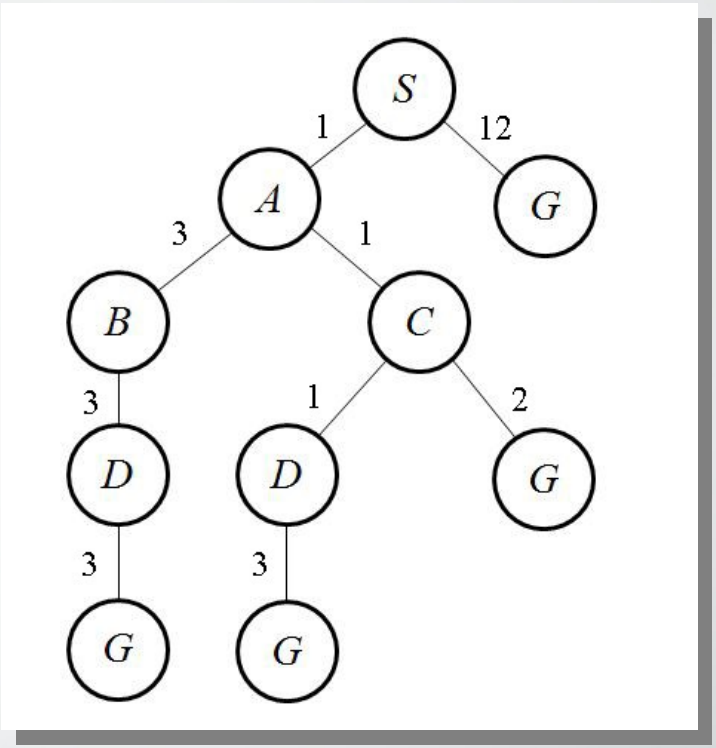


- Expand A to B, C, and D.
- The path to B is the cheapest one with path cost 2.
- Expand B to E
- Total path cost = 2+9 =11
- This might not be the optimal solution since the path AC as path cost 4 (less than 11)

- Expand C to E
- Total path cost = 4+5 =9
- Path cost from A to D is 10 (greater than path cost, 9)
- Hence optimal path is ACE.

UCS example (Find path from S to G)





Initialization: { [S , 0] }

Iteration1: { [S->A , 1] , [S->G , 12] }

Iteration2: { [S->A->C , 2] , [S->A->B , 4] , [S->G , 12] }

Iteration3: { [S->A->C->D , 3] , [S->A->B , 4] , [S->A->C->G , 4] , [S->G , 12] }

Iteration4: { [S->A->B , 4] , [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->G , 12] }

Iteration5: { [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->A->B->D , 7] , [S->G , 12] }

Iteration6: gives the final output as S->A->C->G.

Depth Limit Search

- Depth-first search will not find a goal if it searches down a path that has infinite length. So, in general, depth-first search is not guaranteed to find a solution, so it is not complete.
- This problem is eliminated by limiting the depth of the search to some value L. However, this introduces another way of preventing depth-first search from finding the goal: if the goal is deeper than L it will not be found.
- Perform depth first search but only to a pre-specified depth limit L.
- No node on a path that is more than L steps from the initial state

Depth Limit Search

- ***Completeness***: Incomplete as solution may be beyond specified depth level.
- ***Optimality***: not optimal

- ***Space complexity***: b as branching factor and L as tree depth level, O (b.L)

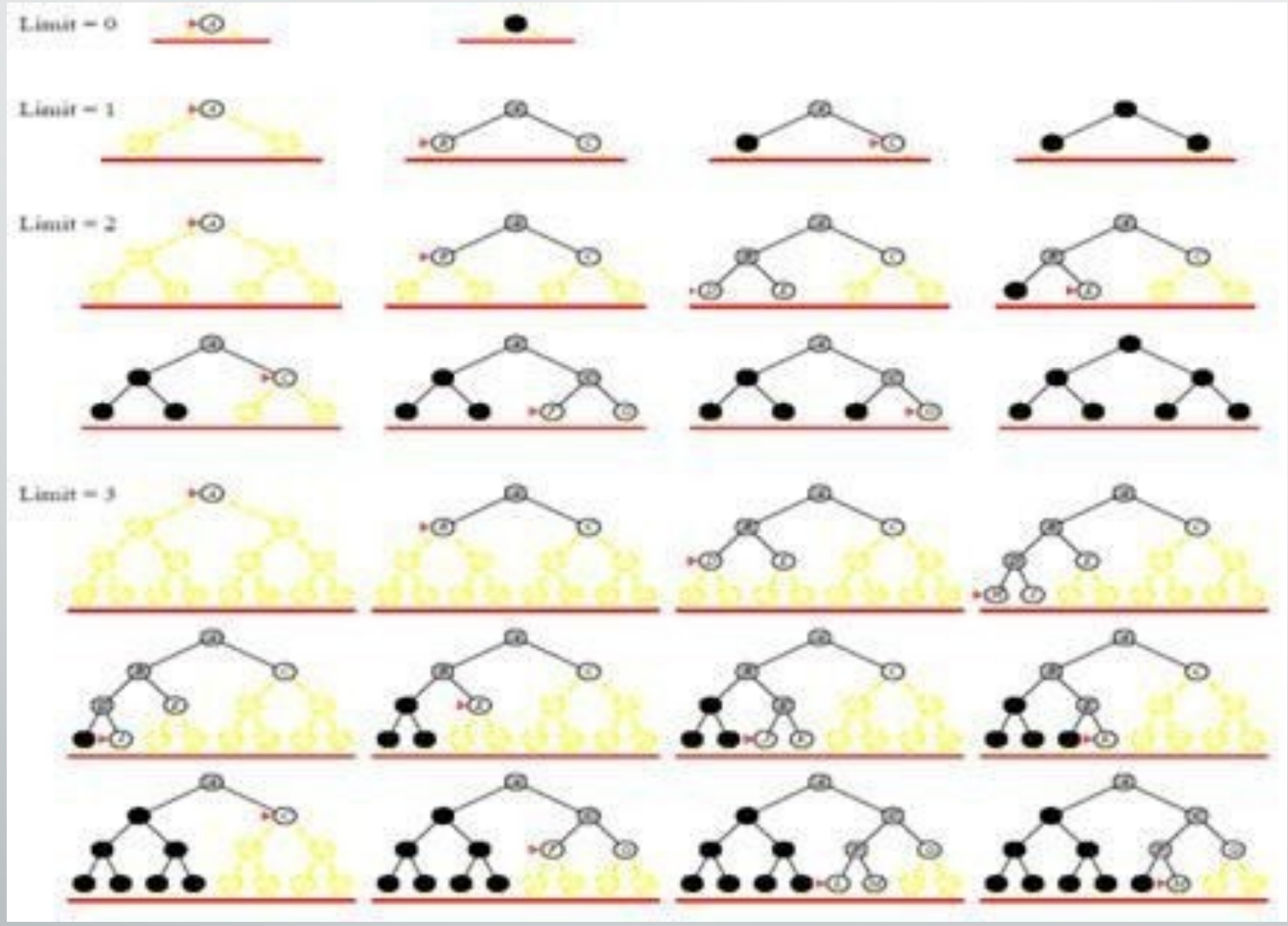
- ***Time Complexity***: O(bL)

Iterative Deepening Search (IDS)

- Iterative deepening search is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits: first depth 0, then depth 1, then depth 2, and so on.

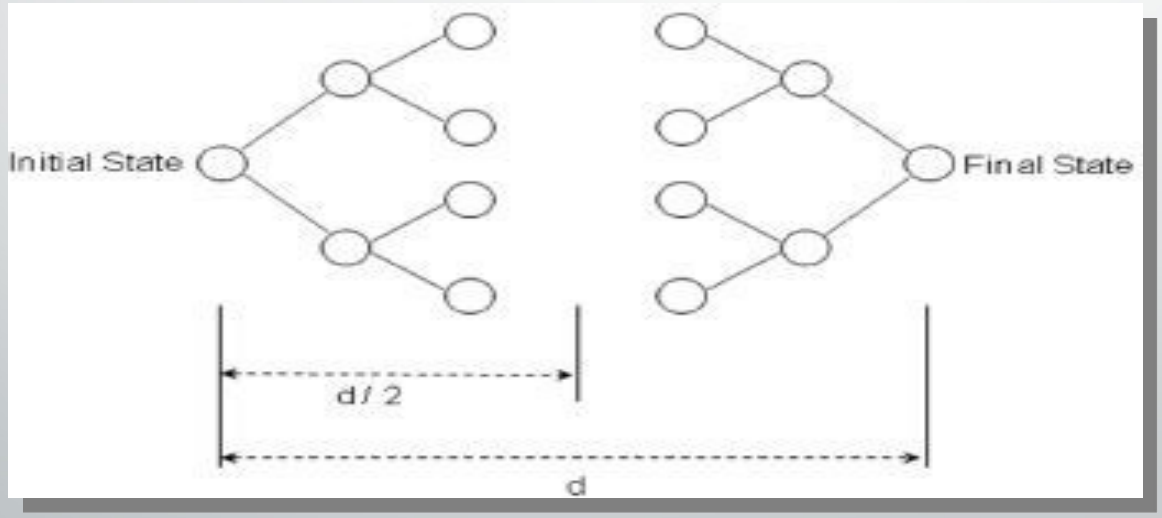
- In effect, iterative deepening combines the benefits of depth-first and breadth-first search.

- It is optimal and complete, like breadth-first search, but has only the modest memory requirements of depth-first search.


Limit = 0
Limit = 1
Limit = 2
Limit = 3

Bidirectional Search

- As the name suggests, bidirectional search suggests to run 2 simultaneous searches
- One from the initial state and the other from the Final state
- Those 2 searches stop when they meet each other at some point in the middle of the graph.



Bidirectional Search

- Completeness:

  Bidirectional search is complete when we use BFS in both searches

- Optimality:

  Like the completeness, bidirectional search is optimal when BFS is used.

- Time/Space Complexity : $O(b^{d/2})$

b: branching factor (assume finite) **d: goal depth** **m: graph depth**

	Complete	optimal	time	space
Breadth-first search	Y	Y, if [1]	$O(b^d)$	$O(b^d)$
Uniform-cost search[2]	Y	Y	$O(b^{C^*/\varepsilon})$	$O(b^{C^*/\varepsilon})$
Depth-first search	N	N	$O(b^m)$	$O(bm)$
Iterative deepening	Y	Y, if [1]	$O(b^d)$	$O(bd)$
Bidirectional search[3]	Y	Y, if [1]	$O(b^{d/2})$	$O(b^{d/2})$

1. edge cost constant, or positive non-decreasing in depth

Informed Search (Heuristic Search)

- Informed search have problem specific knowledge apart from problem definition.
- They use experimental algorithm which improves efficiency of search process.
- The idea is to develop a domain specific heuristic function h(n) where h(n) guesses the cost of getting to the goal from node n.

Heuristic Function

The heuristic function is a way to inform the search about the direction to a goal. It provides an informed way to guess which neighbor of a node will lead to a goal.

Best-First Search (BFS)

- Best-First search is a graph-based heuristic search algorithm
- The name “best-first” refers to the method of exploring the node with the best “score” first.
- An evaluation function is used to assign a score to each candidate node. The evaluation function must represent some estimate of the cost of the path from state to the closest goal state

Algorithm

1. Put the initial node on a list START
2. If START = GOAL or START = EMPTY, then terminate search
3. Assign the next node as START and call this node-A
4. If A = GOAL, terminate the search with success
5. Else-if, node has successor and generate all of them. Find out how far they are from the GOAL node.
6. Sort all the children generated so far by remaining distance from the goal. Name the list as START-1. Replace START = START-1
7. Go to step-2

Types of BFS

- Greedy Best First Search
- A* Search

Greedy Best First Search

- It tries to get as close as it can to the goal.
- It expands the node that appears to be closest to the goal
- It evaluates the node by using heuristic function only.
- Evaluation function f(n) = h(n)
  - heuristic= (estimate of cost from n to goal)
  - h(n) = 0 for goal state

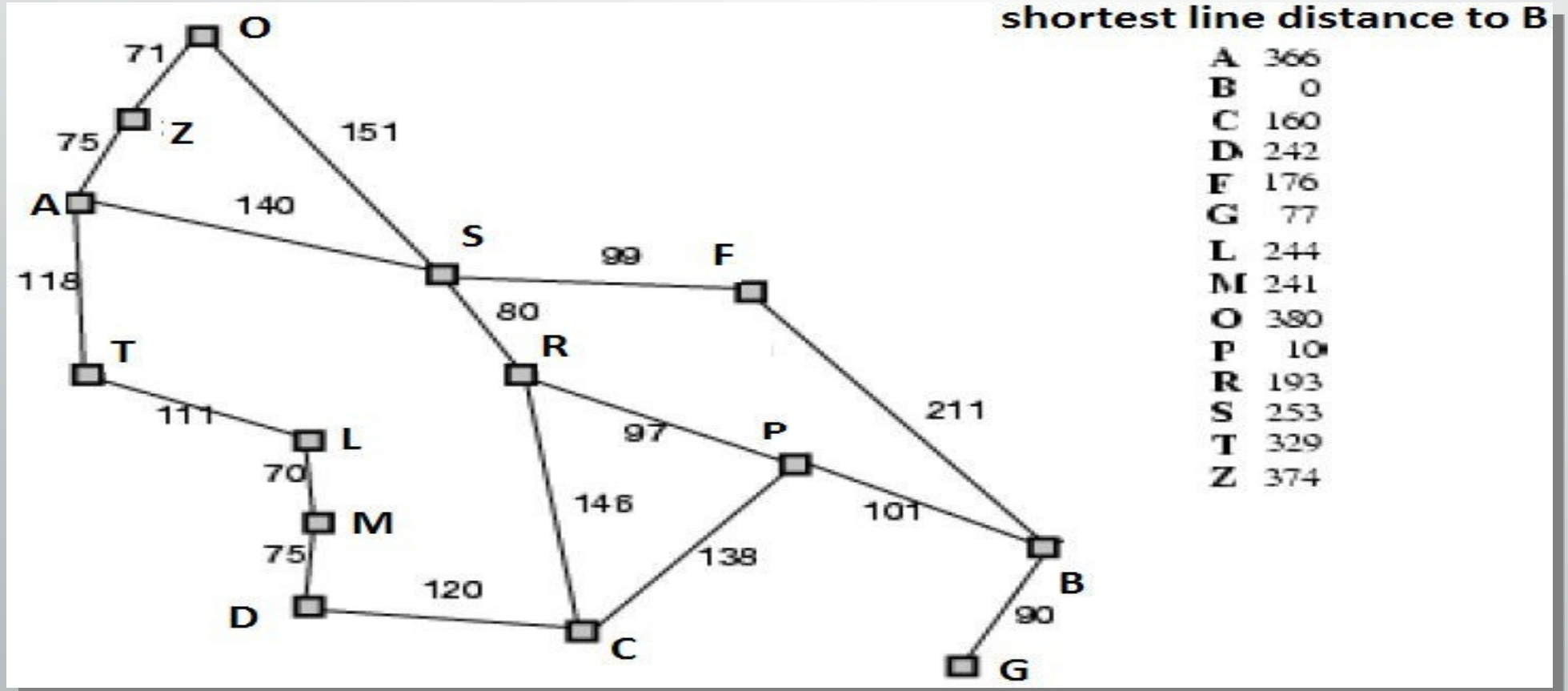
Properties

- ***Completeness:*** No – can get stuck in loops
- ***Time Complexity***: O (b^m), but a good heuristic can give dramatic improvement
- ***Space Complexity***: O (b^m), keeps all nodes in memory
- ***Optimality***: No

Applications:

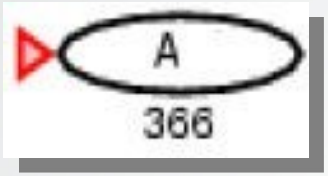
This algorithm is used in Huffman encoding, minimum spanning tree, Dijkstra's algorithm etc.

Given following graph of cities, starting At City "A", problem is to reach to the "B"

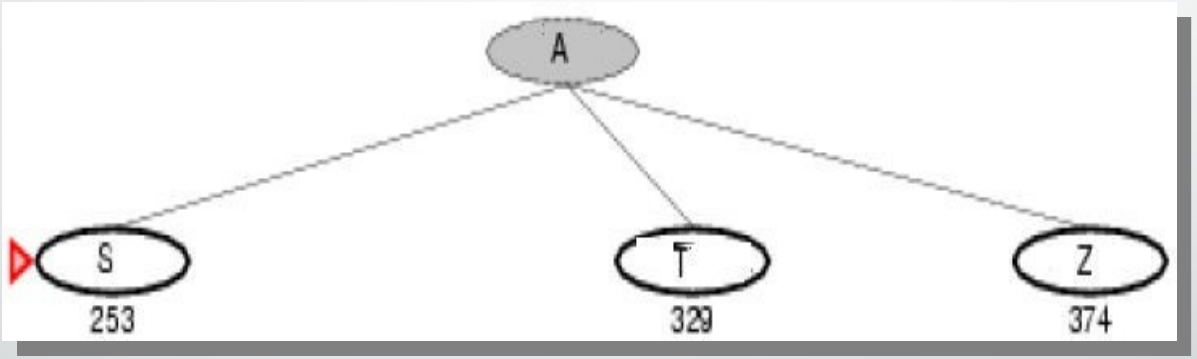


Solution using greedy best first can be as below:

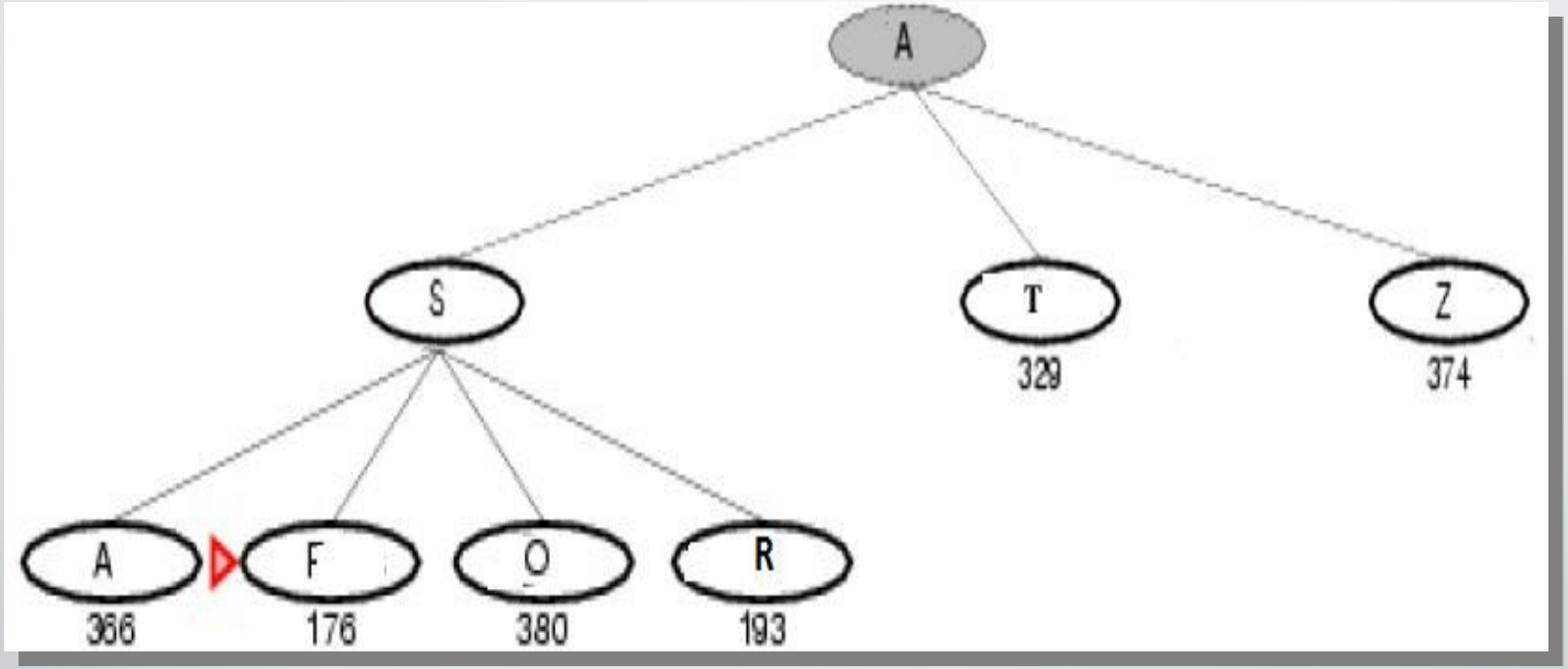
Step 1: Initial State



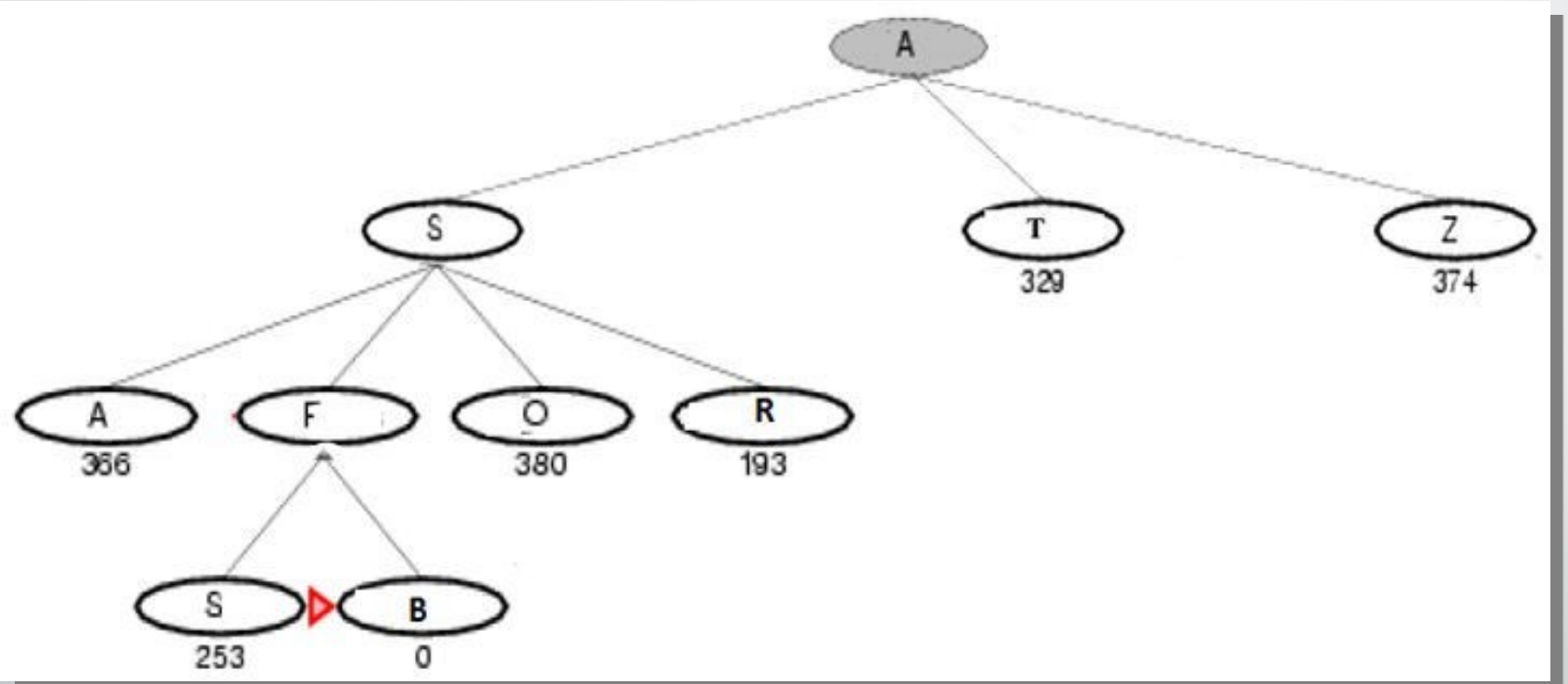
Step 2: After expanding A



Step 3: After expanding S



Step 3: After expanding F



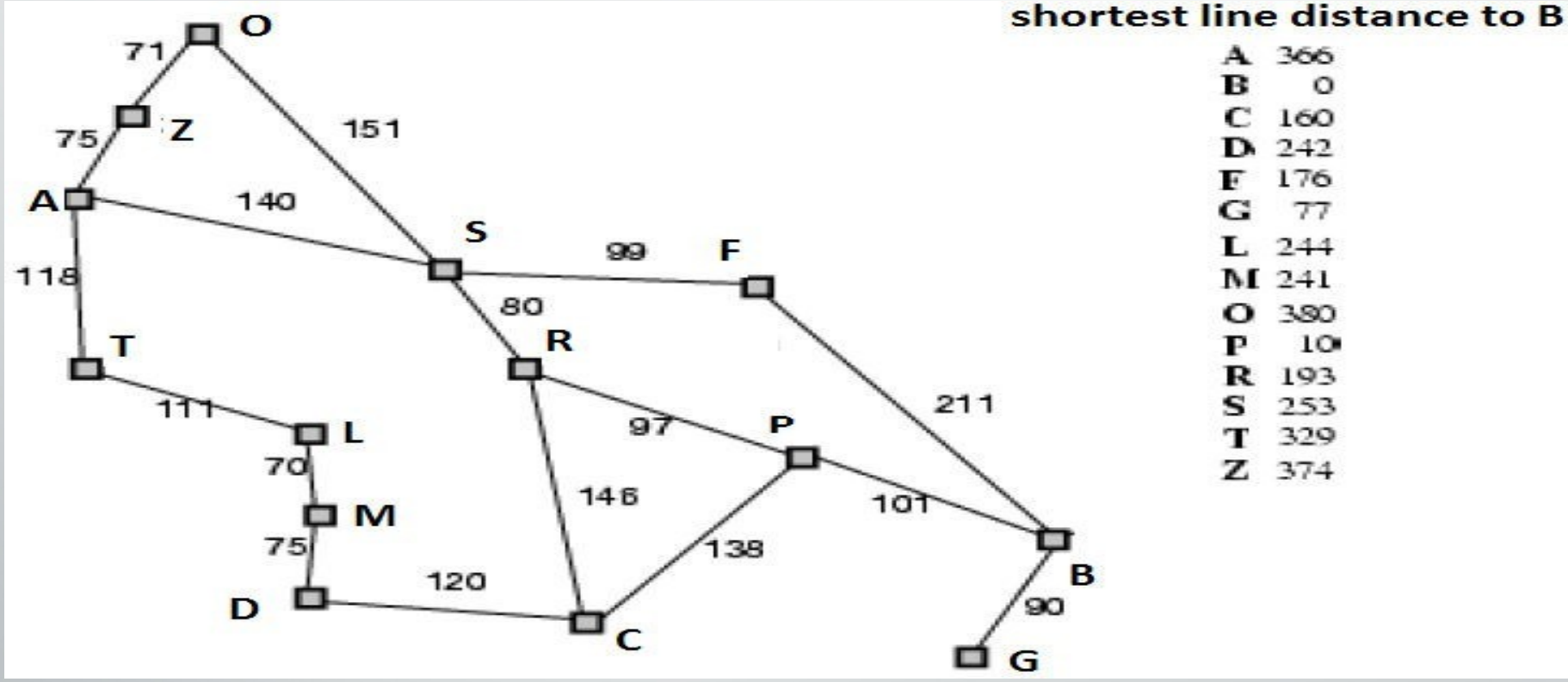
A* Search

- It finds a minimal cost-path joining the start node and a goal node for node n.
- Evaluation function: f(n) = g(n) + h(n)

  Where,
  - g(n) = cost so far to reach n from root
  - h(n) = estimated cost to goal from n

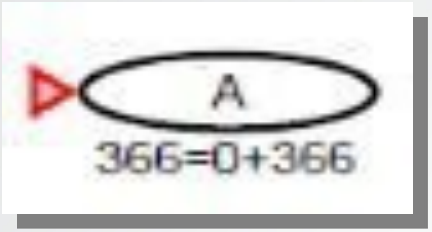
  Thus, f(n) estimates always the lowest total cost of any solution path going through node n.
- Avoid expanding paths that are already expensive
- The main drawback of A* algorithm and indeed of any best- first search is its memory requirement.

A* search example (Find path from A to B)



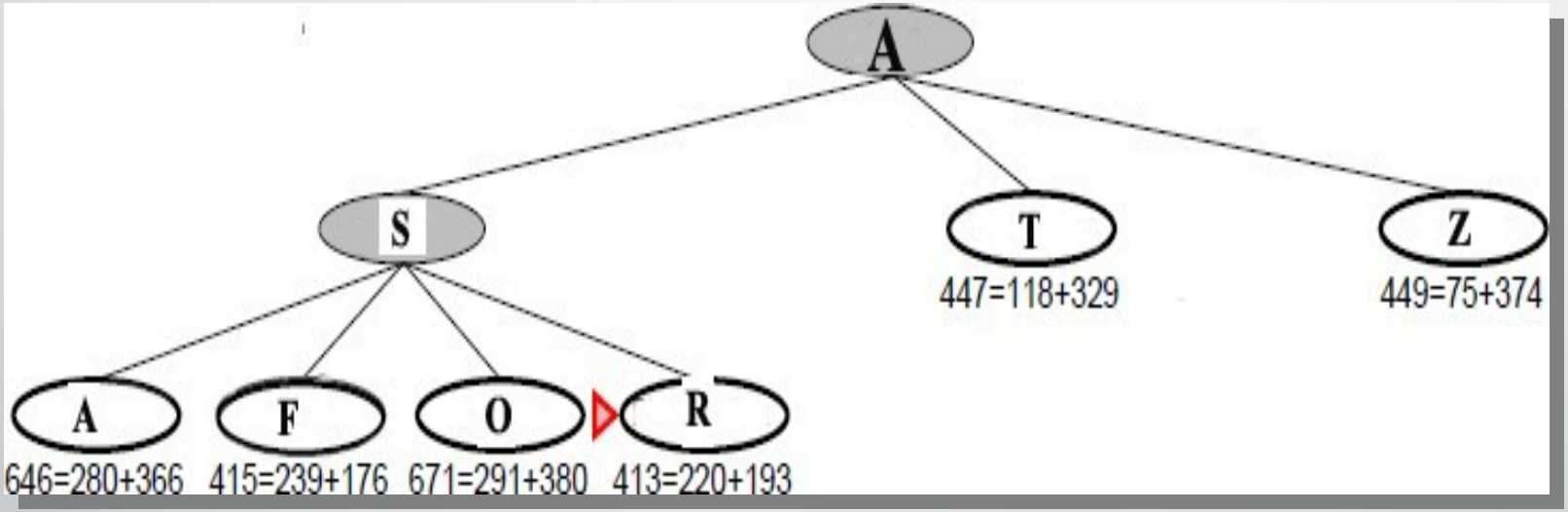
Here, evaluate nodes connected to source. Evaluate f(n)=g(n)+h(n) for each node. Select node with lowest f(n) value.

Step 1:

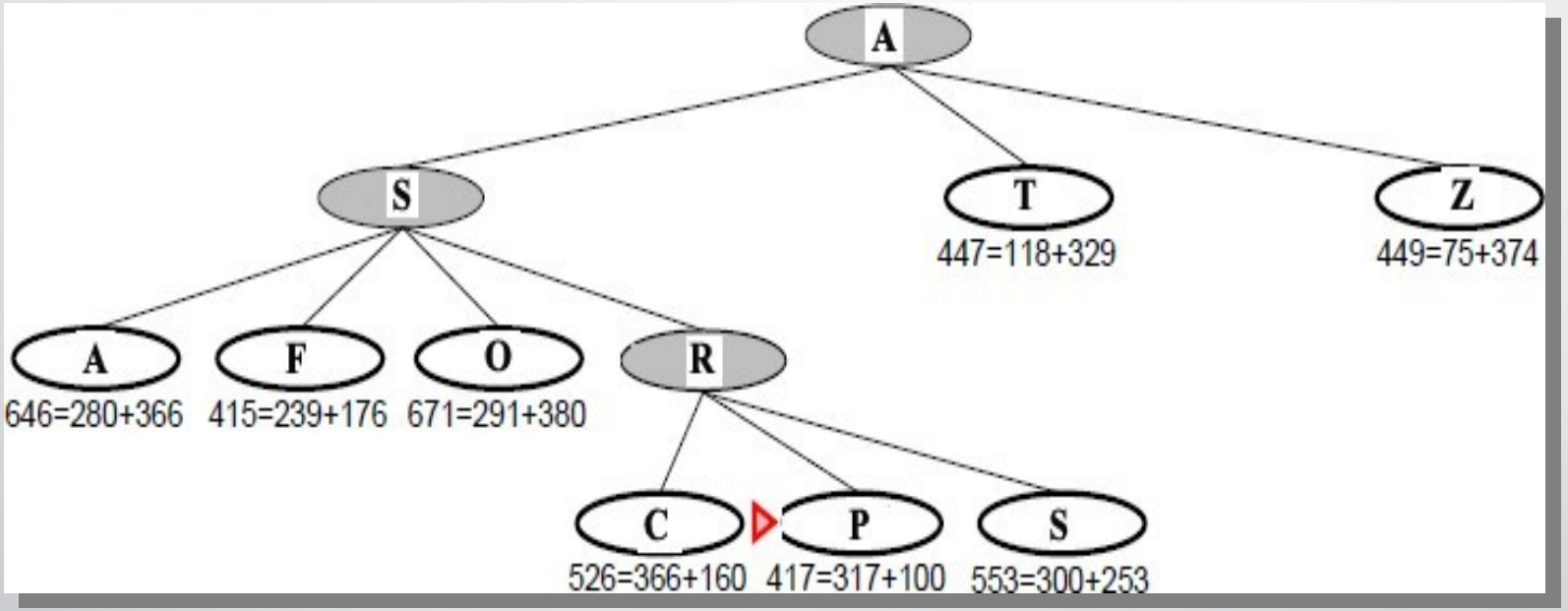


Step 2

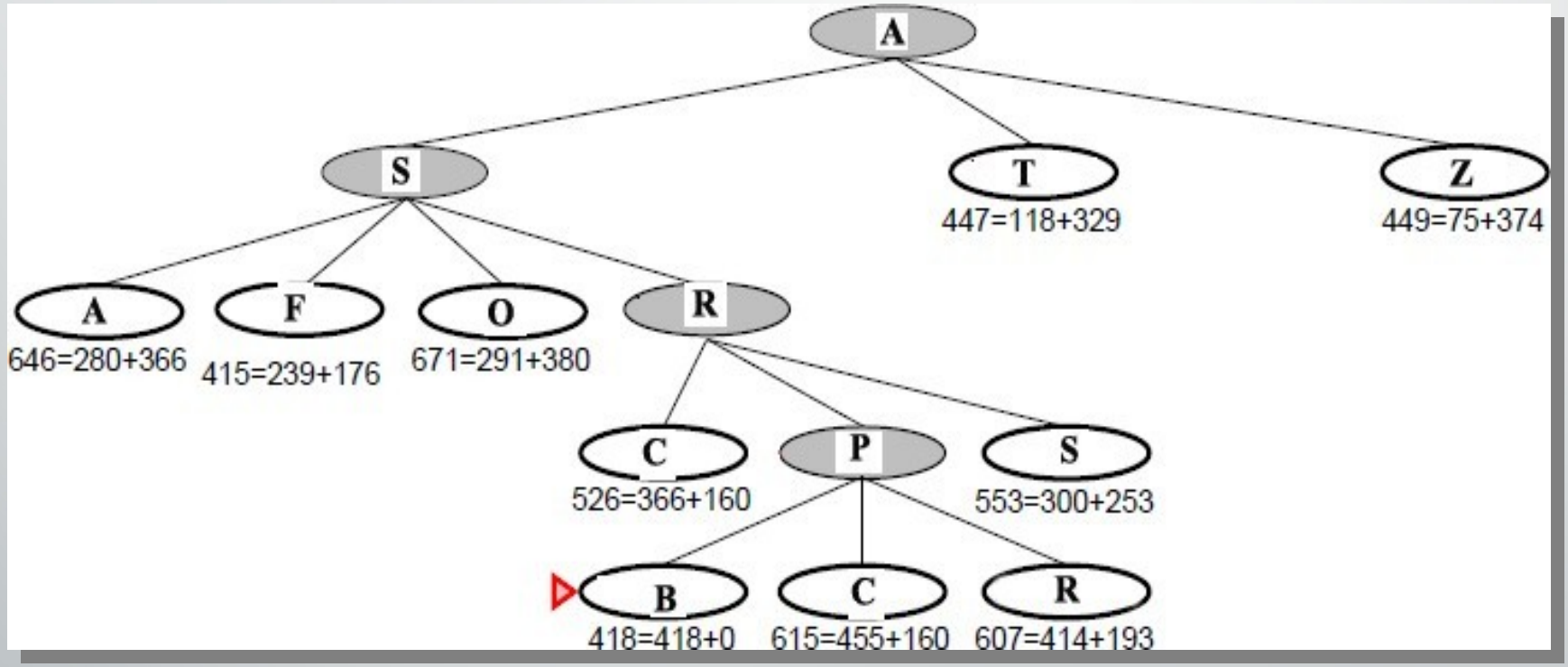
Step 3:


A
S
T
Z
447=118+329
449=75+374
A
F
O
R
646=280+366
415=239+176
671=291+380
413=220+193

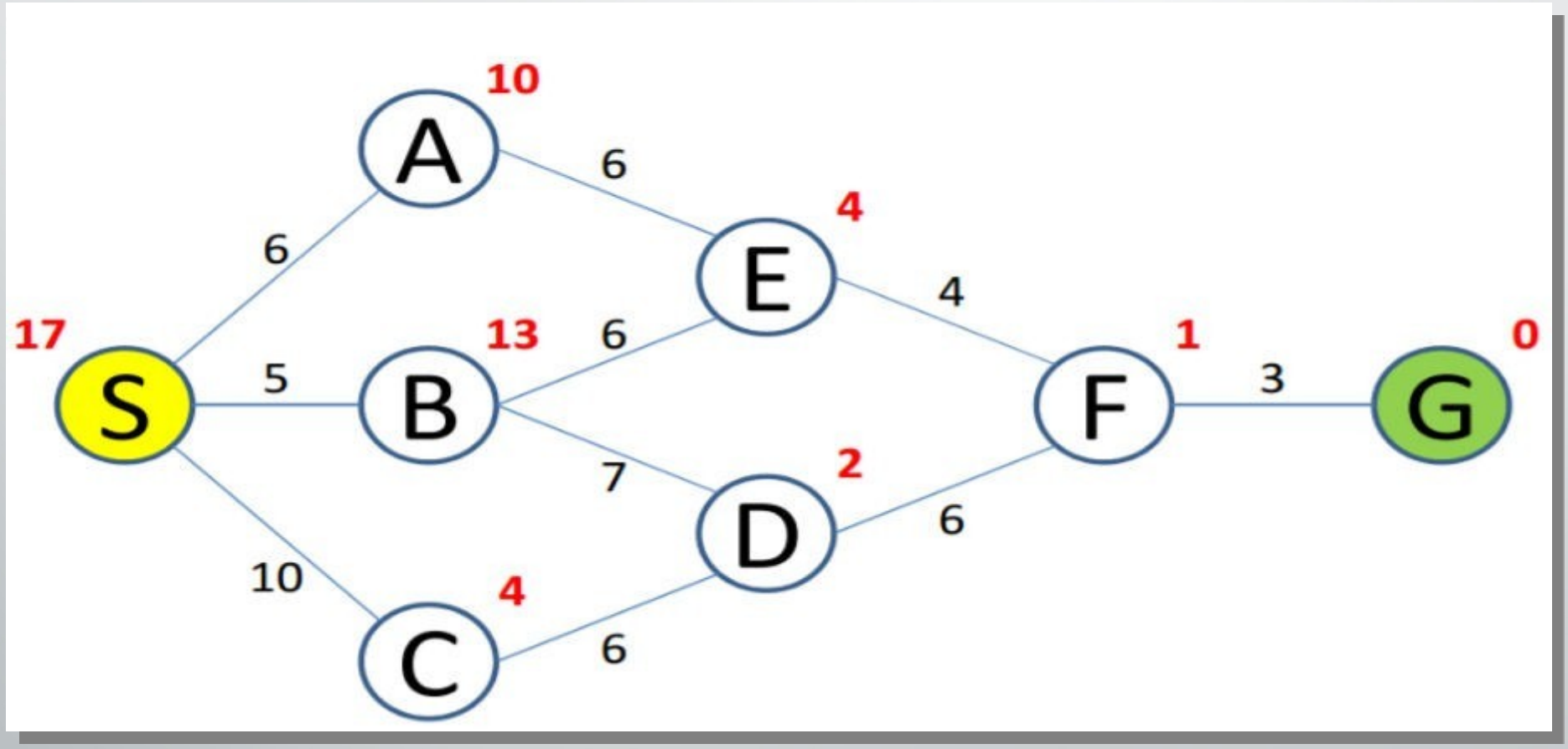
Step 4:


A
S
T
447=118+329
Z
449=75+374
A
646=280+366
F
415=239+176
O
671=291+380
R
C
526=366+160
P
417=317+100
S
553=300+253

Step 5:


A
S
T
447=118+329
Z
449=75+374
A
646=280+366
F
415=239+176
O
671=291+380
R
C
526=366+160
P
S
553=300+253
B
418=418+0
C
615=455+160
R
607=414+193

Class Work: Perform A* Search



AO* Searching

- AO* Search is a type of heuristic search algorithm .
- AO* Search is used when problems can be divided into sub parts and which can be combined
- AO* in artificial intelligence is represented using AND OR graph or AND OR tree
- AO* have one or more and arc in it

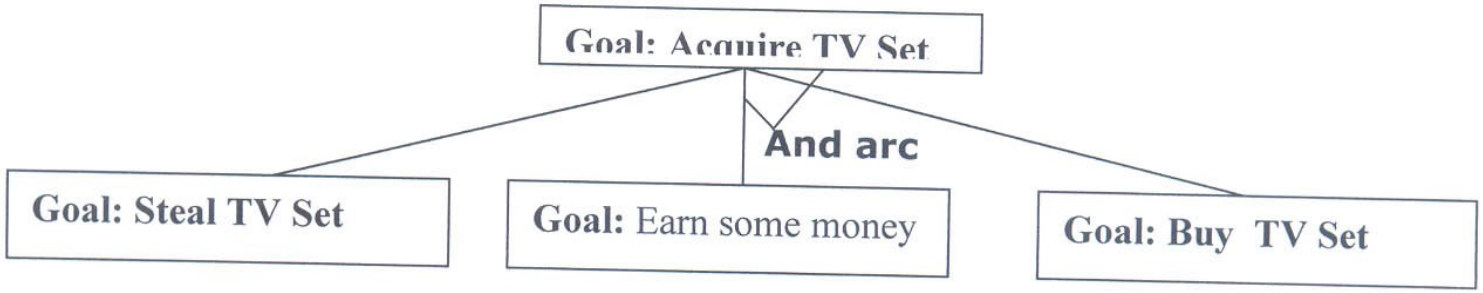
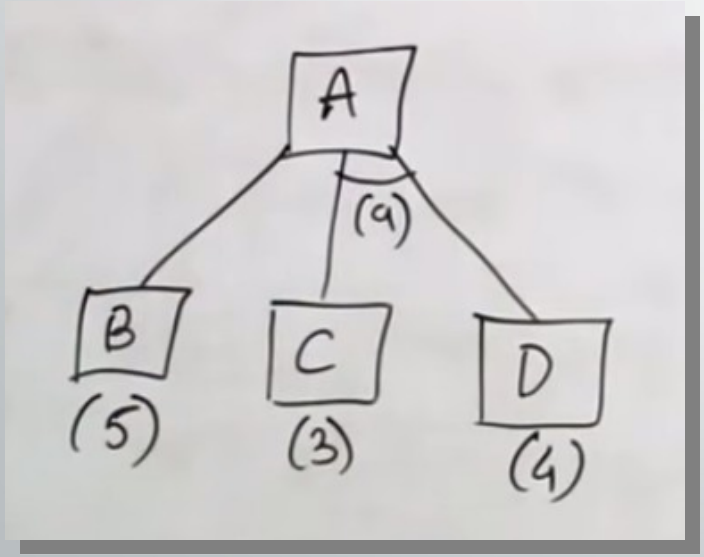
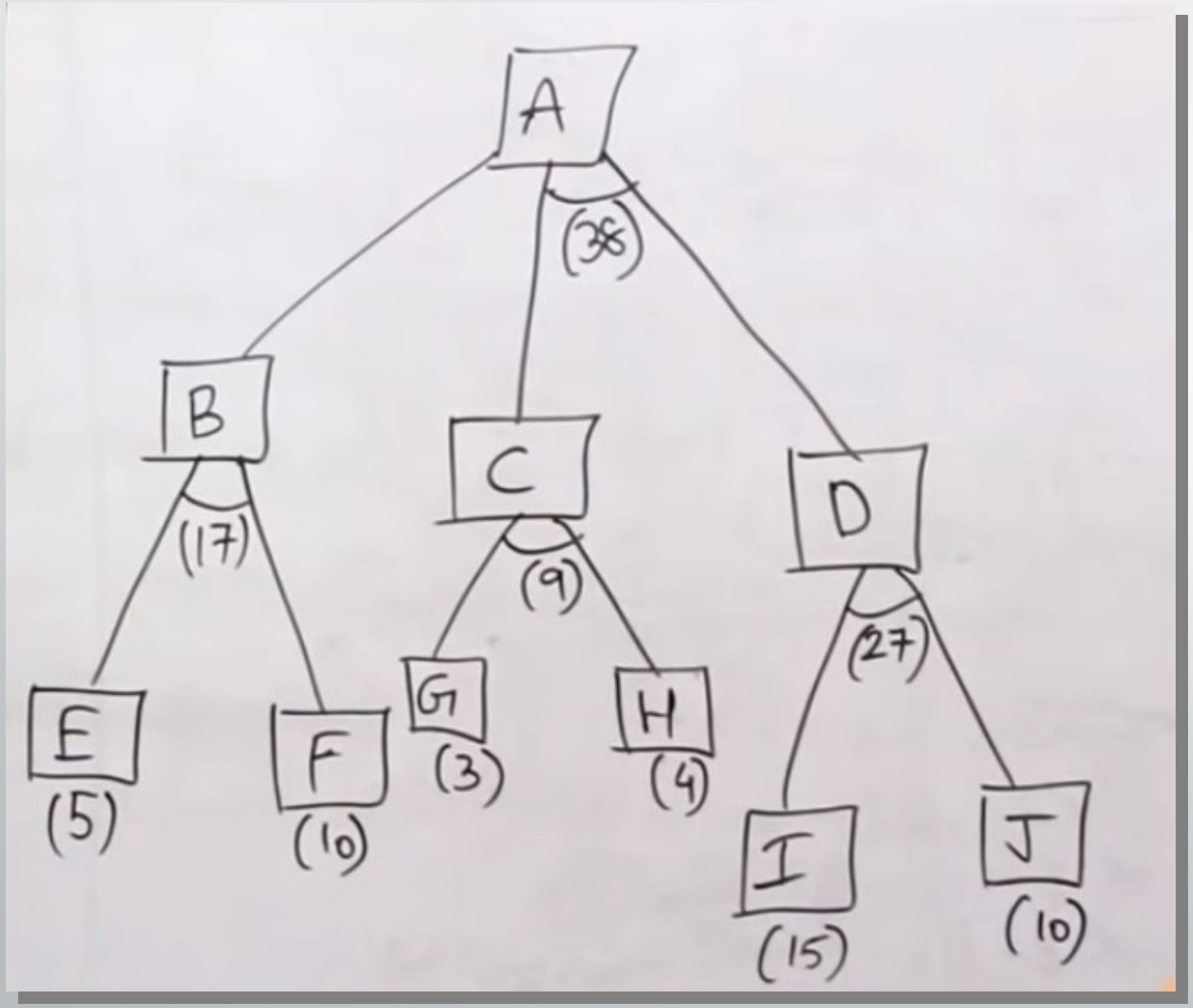


Figure shows AND - Or graph - an example.




A
(36)
B
C
D
(17)
(9)
(27)
E
F
G
H
I
J
(5)
(10)
(3)
(4)
(15)
(10)

Hill Climbing Search

Hill climbing is an extension of depth-first search which uses some knowledge such as estimates of the distance of each node from the goal to improve the search

It is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value.

Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next

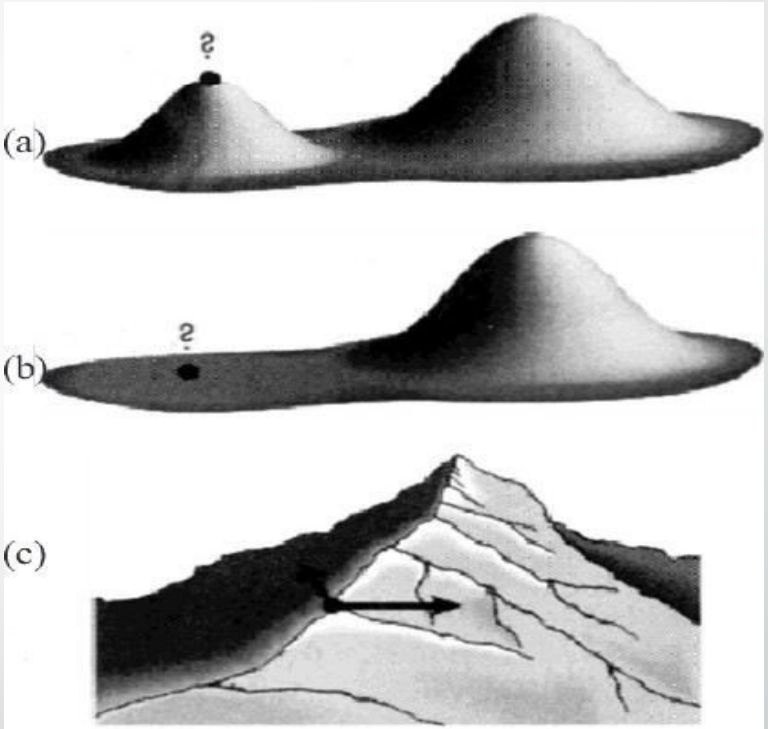


Figure 5.9 Local maxima, Plateaus and ridge situation for Hill Climbing

Hill Climbing Search

ALGORITHM

Push the root node on the stack

Pop the stack & examine it

- if the top of the stack is the goal, stop
- if the stack is empty, stop, there is no path
- Otherwise, sort the children of the current node and then push them on the stack

Hill climbing does not look ahead beyond the immediate neighbors of the current state.

Hill Climbing Search

Drawbacks

- **Local Maxima**

  A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.

- **Plateaus**: A plateau is an area of the search space where evaluation function is flat, thus requiring random walk

- **Ridges**: Where there are steep slopes and the search direction is not towards the top but towards the side

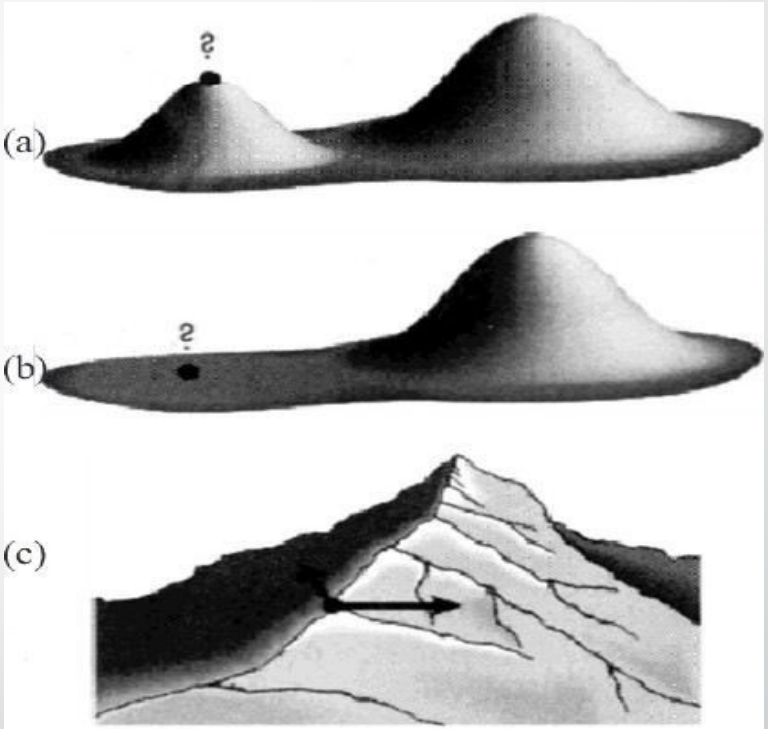


Figure 5.9 Local maxima, Plateaus and ridge situation for Hill Climbing

Hill Climbing Search

Remedies

- **Back tracking for local maximum:**

  The back tracking help in undoing what is been done so far and permit to try totally different part to attain the global peak.

- **Big Jump:**

  A big jump is the solution to escape from plateaus because all neighbors' points have same value using the greedy approach

- **Random restart:**

  Keep restarting the search from random locations until a goal is found

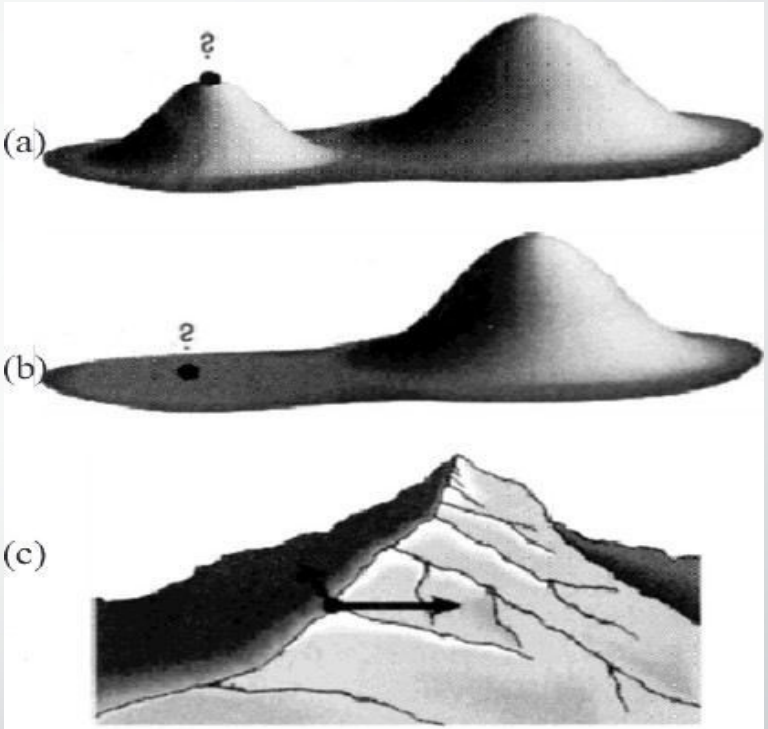
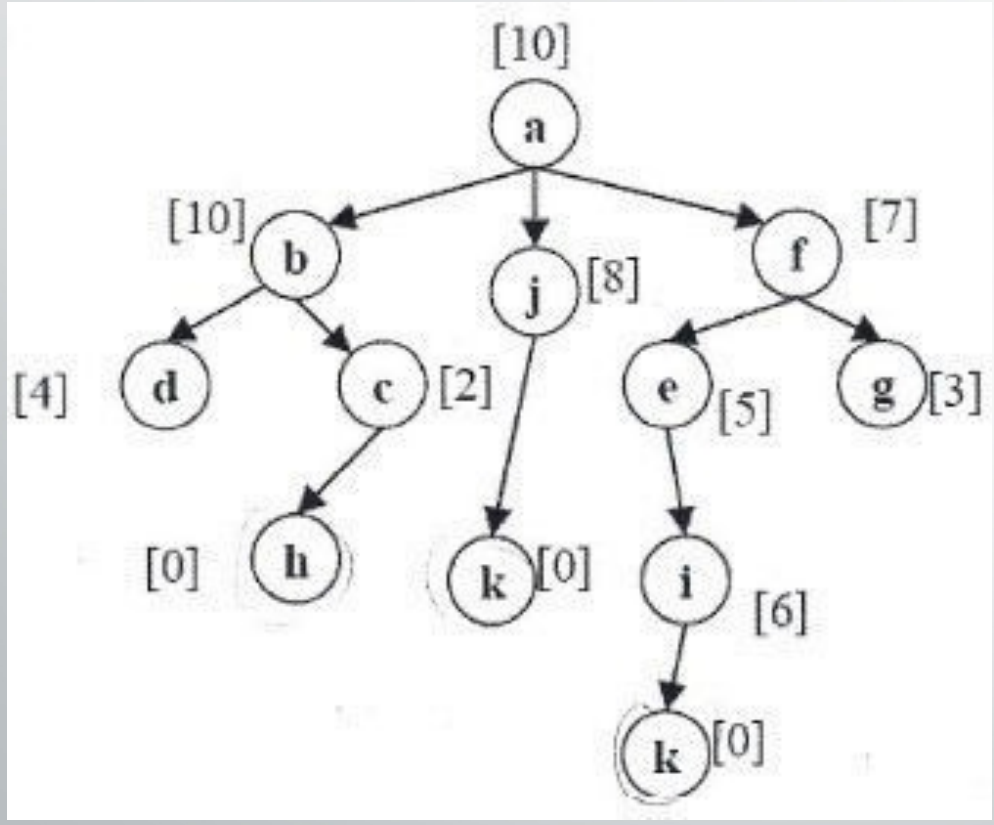


Figure 5.9 Local maxima, Plateaus and ridge situation for Hill Climbing

Why Hill Climbing is not Complete?

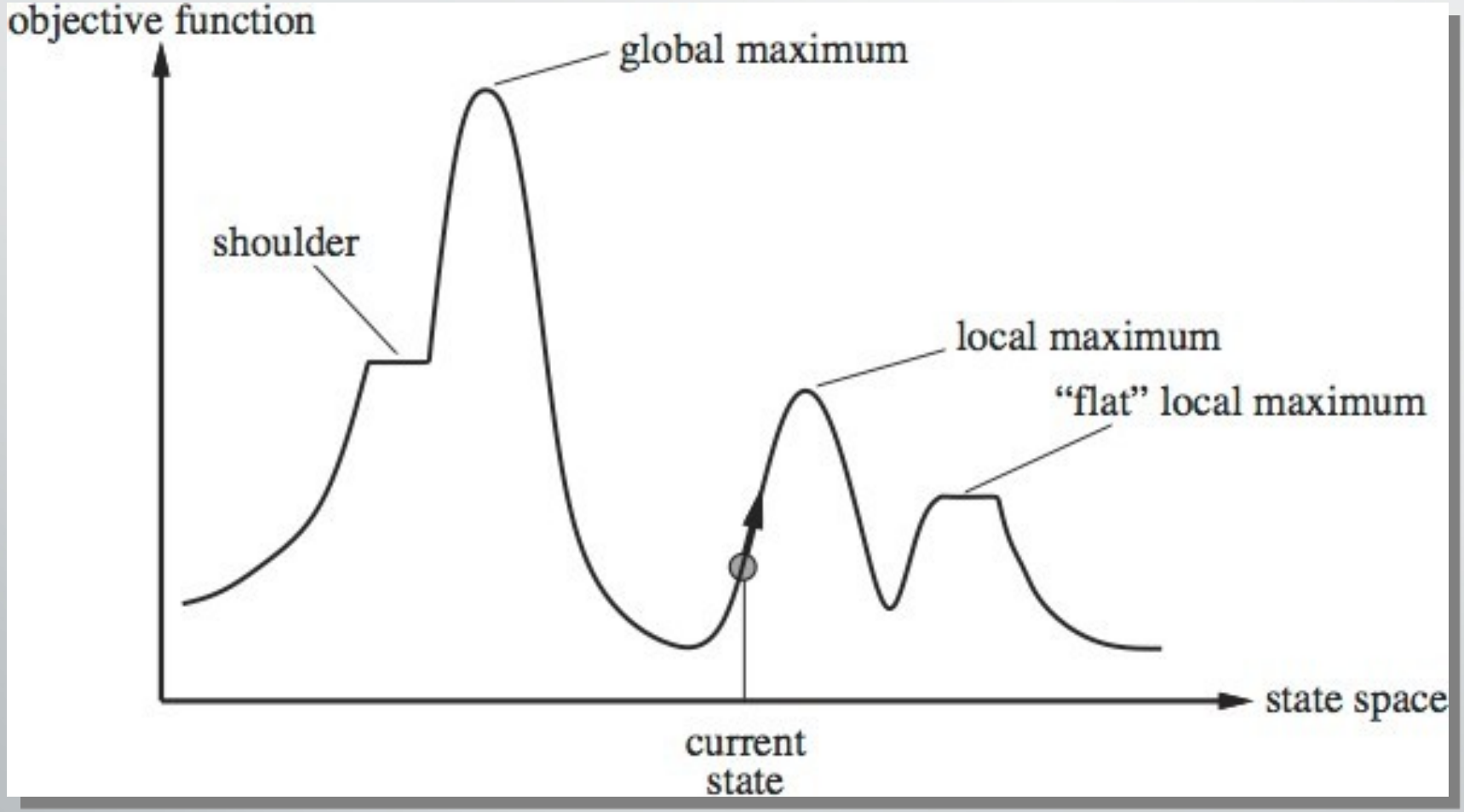
- Hill-climbing always attempts to improve the current state. make changes that
- The main problem that hill climbing can encounter is that of local maxima. This occurs when the algorithm stops making progress towards an optimal solution; mainly due to the lack of immediate improvement in adjacent states.

Problems in Hill Climbing



Here, "a" is initial and h and k are final states

- We start a-> f-> g and then what ??finish(without result)
- A common way to avoid getting stuck in local maxima with Hill Climbing is to use random restarts. In your example if G is a local maxima, the algorithm would stop there and then pick another random node to restart from. So if J or C were picked (or possibly A, B, or D), this would find the global maxima in H or K


objective function
global maximum
shoulder
local maximum
"flat" local maximum
state space
current
state

Simulated Annealing

- Simulated Annealing escapes local maxima by allowing some "bad" moves but gradually decrease their frequency.

- Instead of restarting from a random point, we allow the search to take some downhill steps to try to escape local maxima

Simulated Annealing

Algorithm:

- A random pick is made for the move
- If it improves the situation, it is accepted straight away
- If it worsens the situation, it is accepted with some probability less than 1. i.e. for bad moves the probability is low and for comparatively less bad moves, it is higher.

Applications Simulated Annealing

- VLSI layout problem
- Traveling salesman problem

Adversarial Search(Game Playing)

Competitive environments in which the agents goals are in conflict, give rise to adversarial (oppositional) search, often known as games.

In AI, games means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which utility values at the end of the game are always equal and opposite.

E.g. If first player wins, the other player necessarily loses.

Adversarial Search(Game Playing)

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

1. S0: The **initial state**

2. PLAYER(s)

3. ACTIONS(s)

4. RESULT(s, a

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

6. UTILITY(s, p): A **utility function** (objective function or payoff function), defines the final numeric value for a game that ends in terminal state 's‘ for a player p‘.

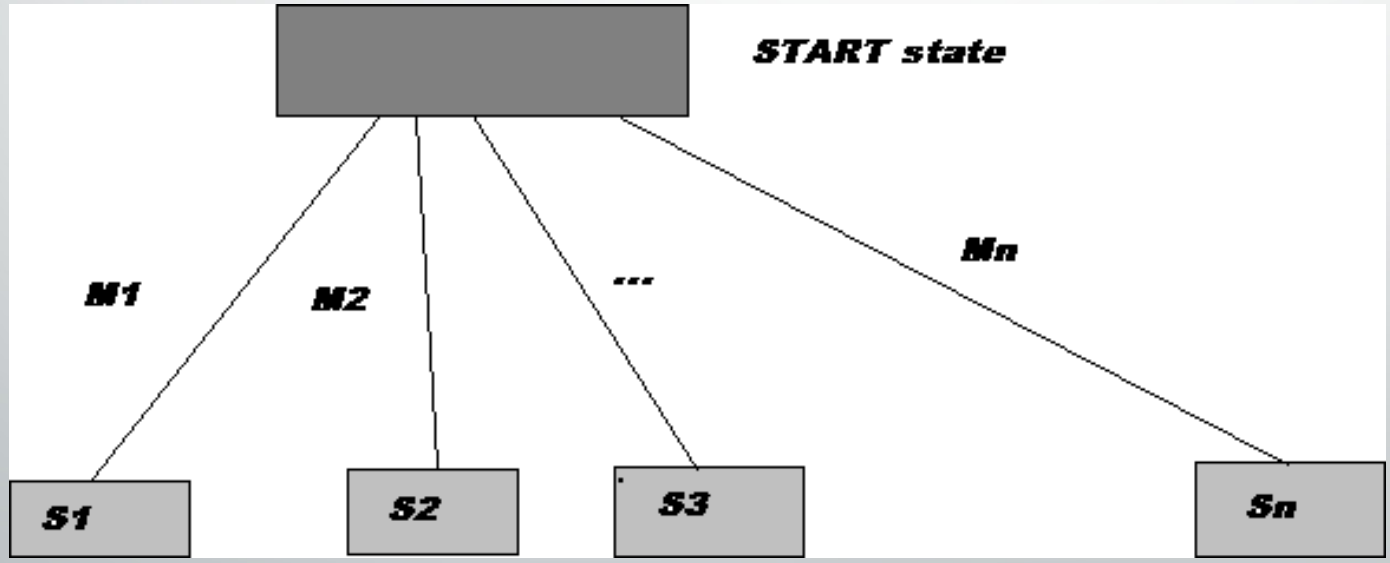
Adversarial Search(Game Playing)

A game can be formally defined as a kind of search problem as below:

- **Initial state**: It includes the board position and identifies the player's to move.
- **Successor function**: It gives a list of (move, state) pairs each indicating a legal move and resulting state.
- **Terminal test**: This determines when the game is over. States where the game is ended are called terminal states.
- **Utility function**: It gives numerical value of terminal states. E.g. win (+1), loose (-1) and draw (0).

Game Trees

- We can represent all possible games(of a given type) by a directed graph often called a game tree.
- The nodes of the graph represent the states of the game. The arcs of the graph represent possible moves by the players (+ and -)



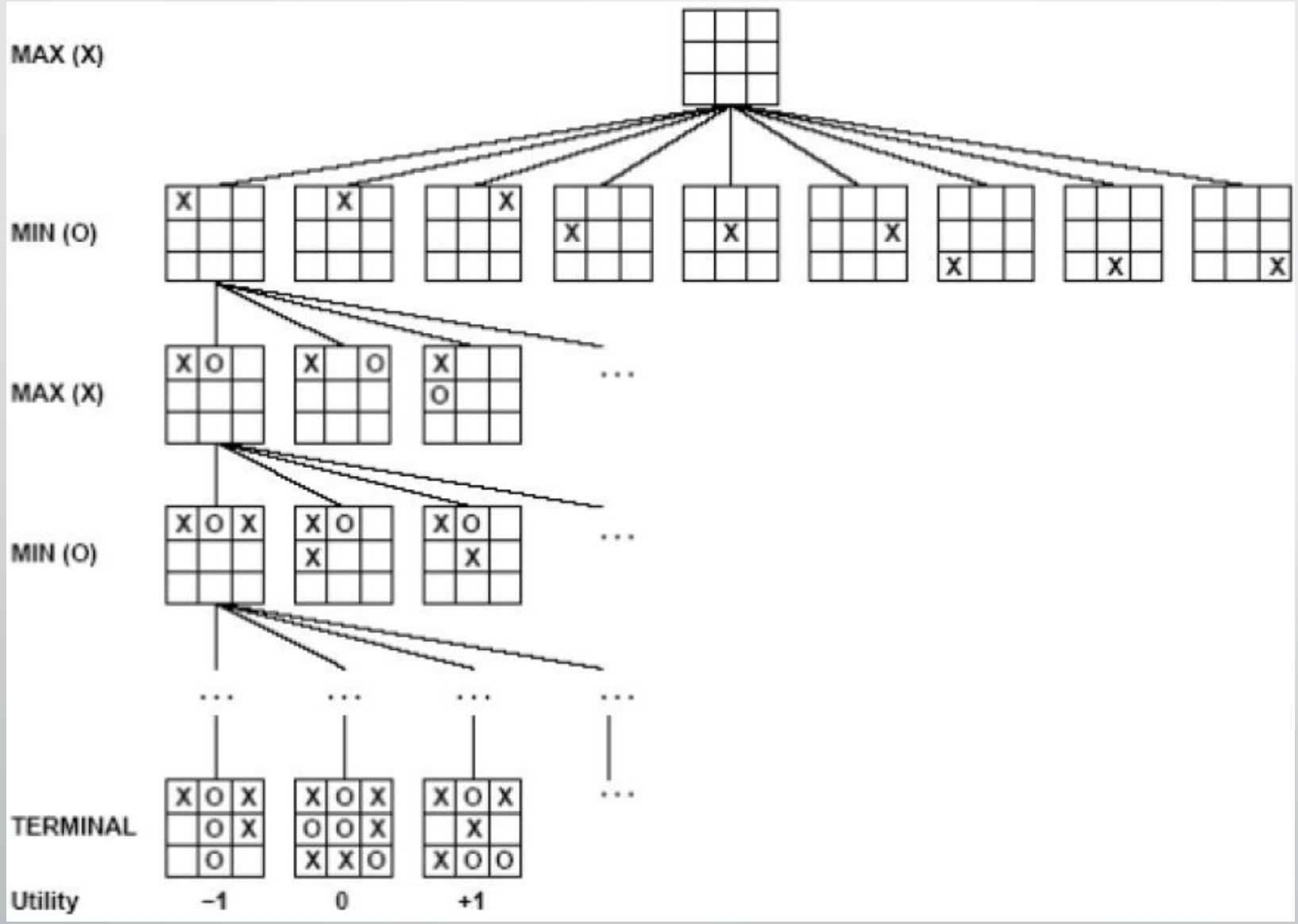
Example: Tic-tac-toe

There are two players denoted by X and O. They are alternatively writing their letter in one of the 9 cells of a 3 by 3 board. The winner is the one who succeeds in writing three letters in line.

The game begins with an empty board. It ends in a win for one player and a loss for the other, or possibly in a draw.

A complete tree is a representation of all the possible plays of the game. The root node is the initial state, in which it is the first player's turn to move (the player X).

The successors of the initial state are the states the player can reach in one move, their successors are the states resulting from the other player's possible replies, and so on.


MAX (X)
MIN (O)
MAX (X)
MIN (O)
TERMINAL
Utility
−1
0
+1

MiniMax Game Search

- It is a recursive algorithm for choosing the next move in a n-player game, usually a two player game
- A value is associated with each position or state of the game
- The value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach the position.
- The player then makes the move that maximizes the minimum value of the position from the opponents possible moves called maximizing player and other player minimize the maximum value of the position called minimizing player.

MiniMax Game Search

- It is a Depth-first search with limited depth.
- Assume the opponent will make the best move possible.

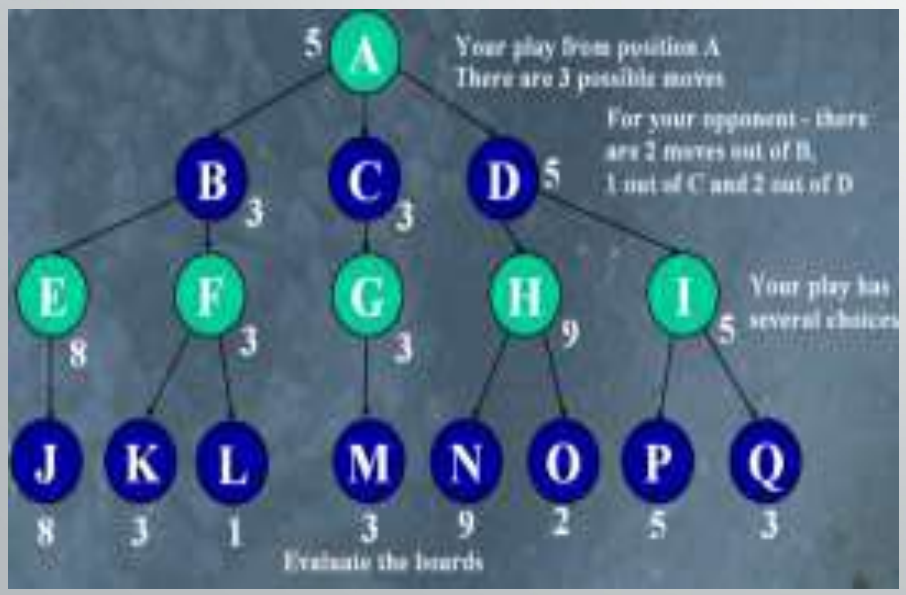
Algorithm

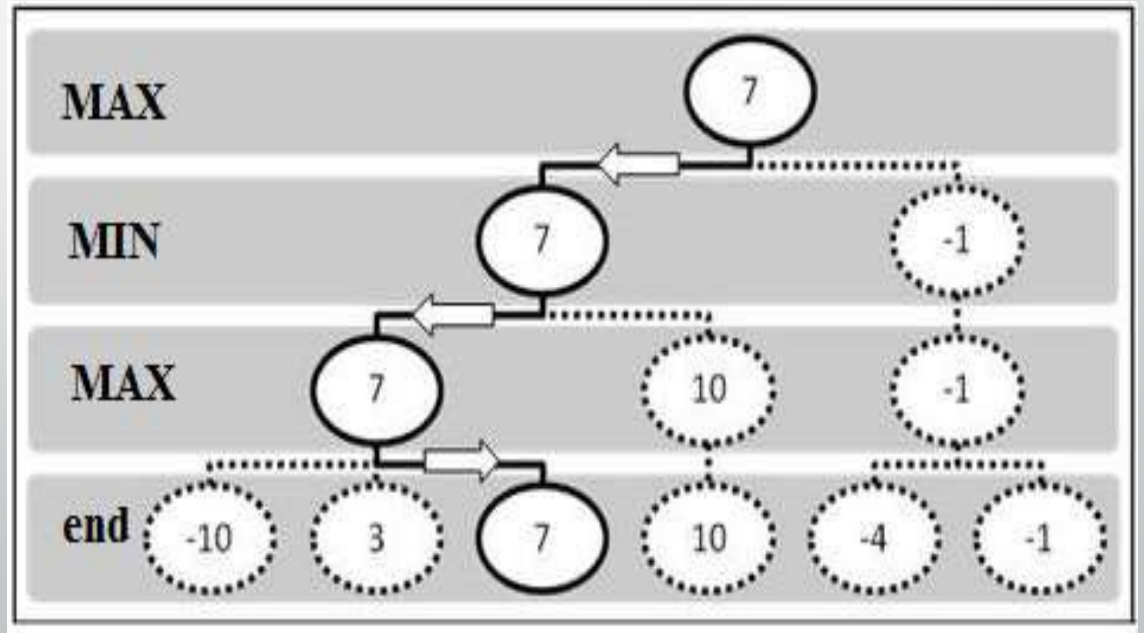
minimax (player,board)
if(game over in current board position)
return winner
if(max's turn)
return maximal score of calling minimax
else (min's turn)
return minimal score of calling minimax

MiniMax Game Search

Example

We first consider games with two players; MAX and MIN. MAX moves first, and then they take turns moving until the game is over. Each level of the tree alternates, MAX is trying to maximize her score, and MIN is trying to minimize MAX score in order to undermine her success. At the end of the game, points are awarded to the winning player and penalties are given to the loser.





Alpha–beta pruning

- Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the mini max algorithm in its search tree.
- It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.)
- It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further.

Alpha Beta Pruning(Clipping)

- Minimax Algorithm explores some parts of tree it doesn’t have to..
- The exact implementation of alpha-beta keeps track of the best move for each side as it moves throughout the tree.
- Alpha–beta pruning gets its name from the following two parameters that describe bounds on the backed up values that appear anywhere along the path:

Alpha Beta Pruning(Clipping)

- **α** = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.

- **β** = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

Alpha Beta Pruning(Clipping)

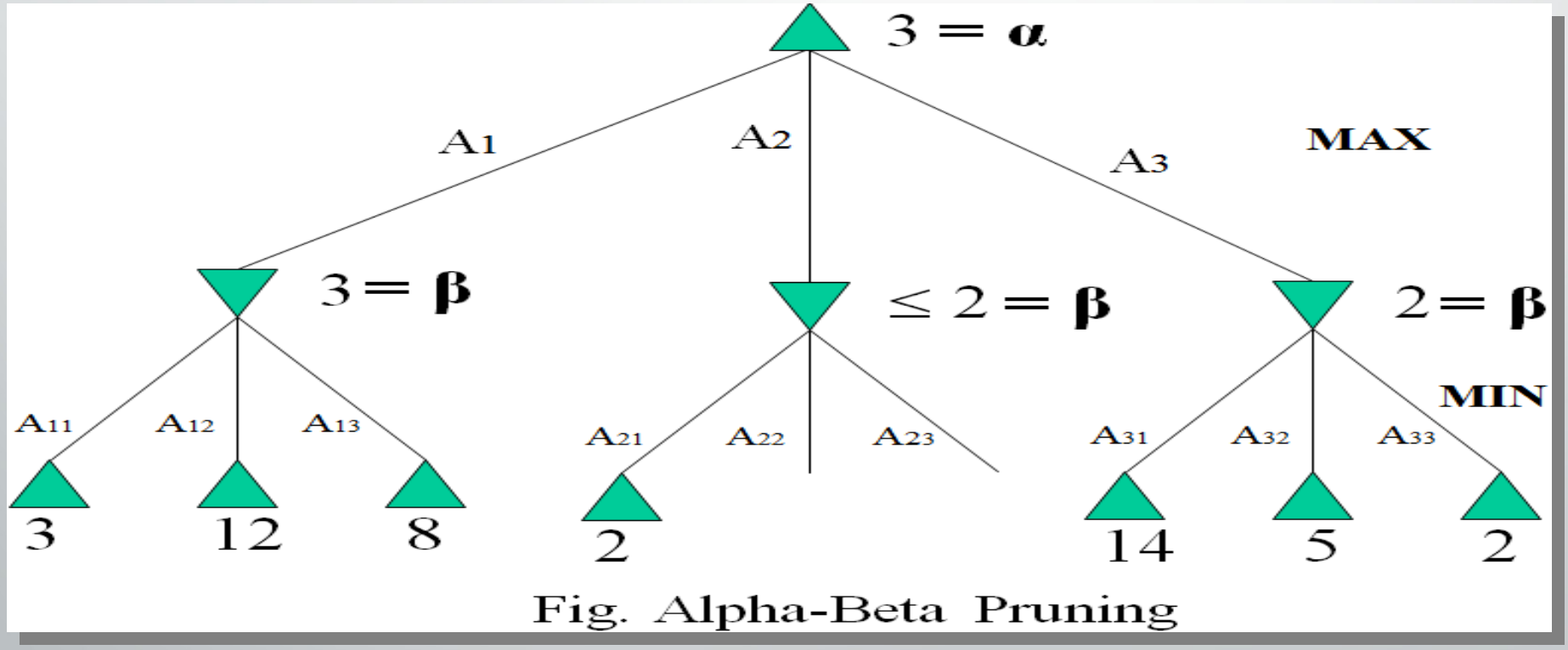
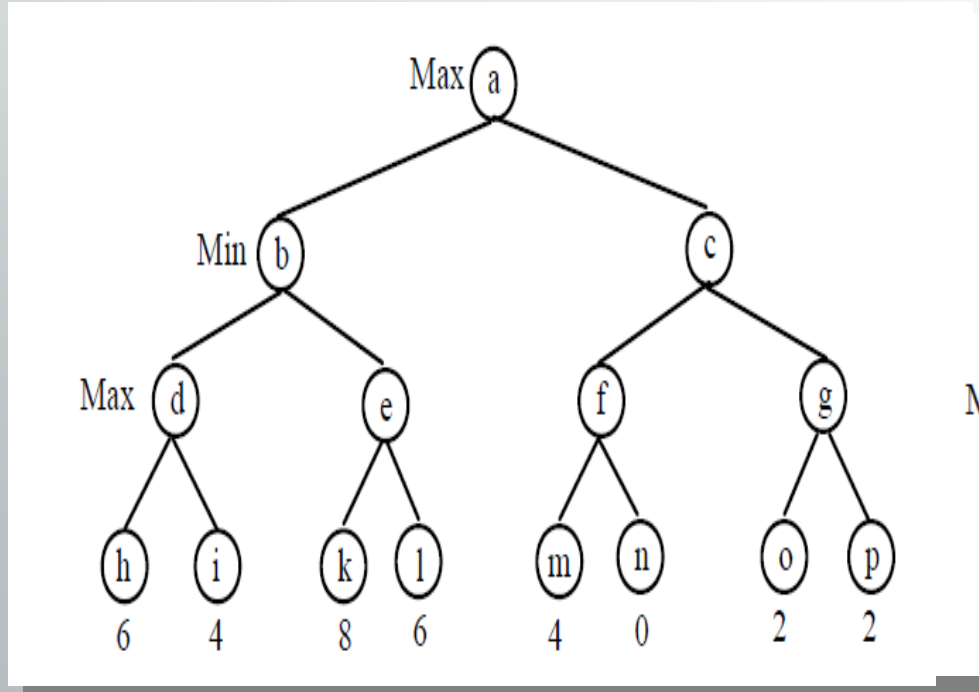
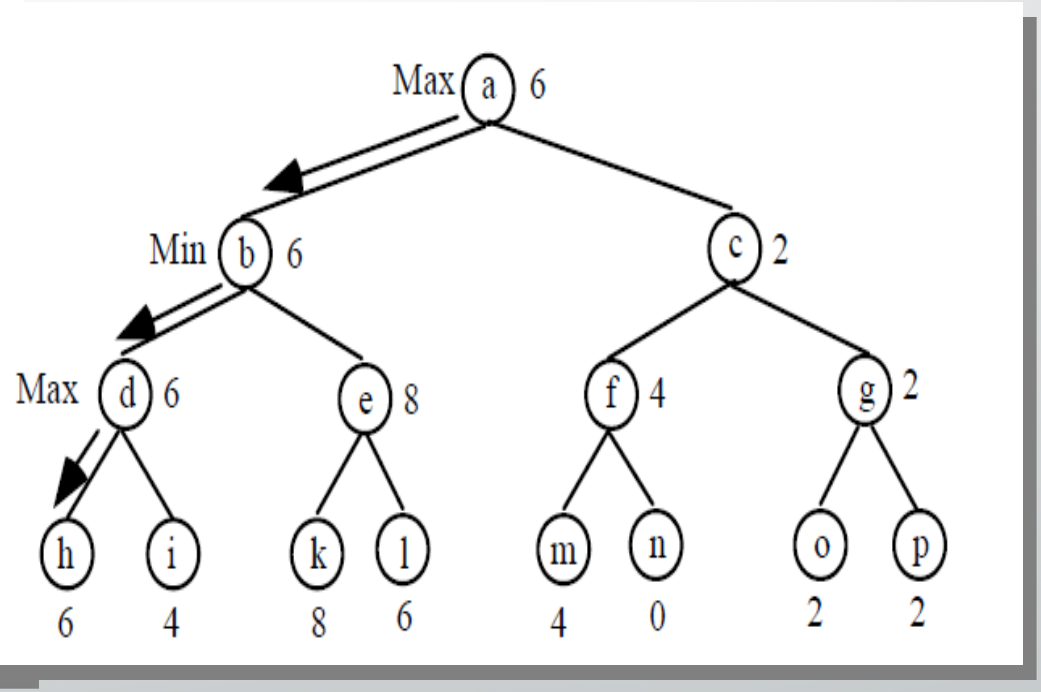


Fig. Alpha-Beta Pruning

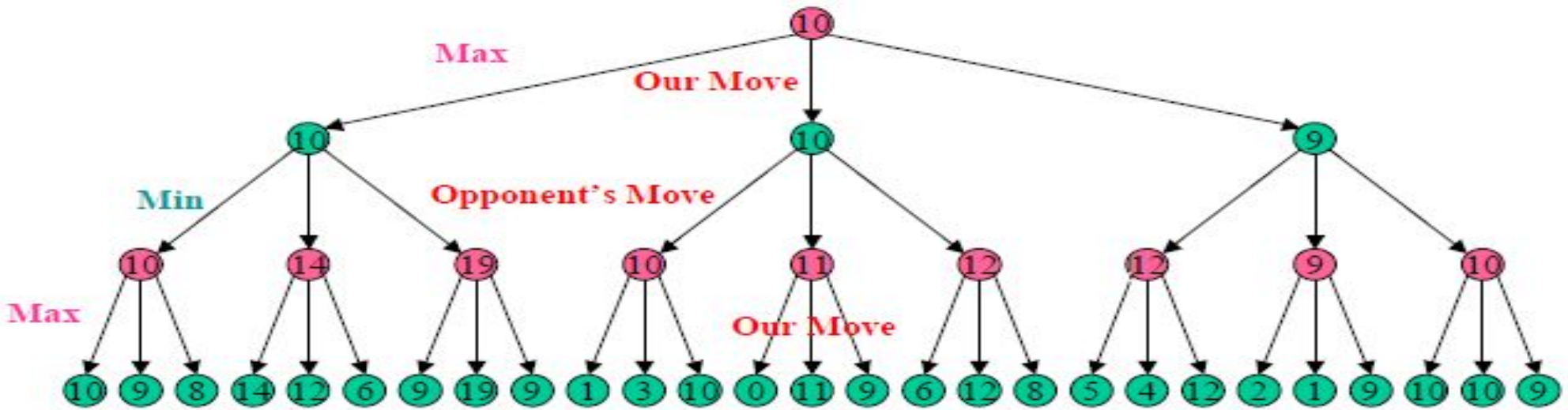
Q.1Consider the following game tree (drawn from the point of view of the Maximizing player)

1. Use the mini-max procedure and show what moves should be chosen by the two players.





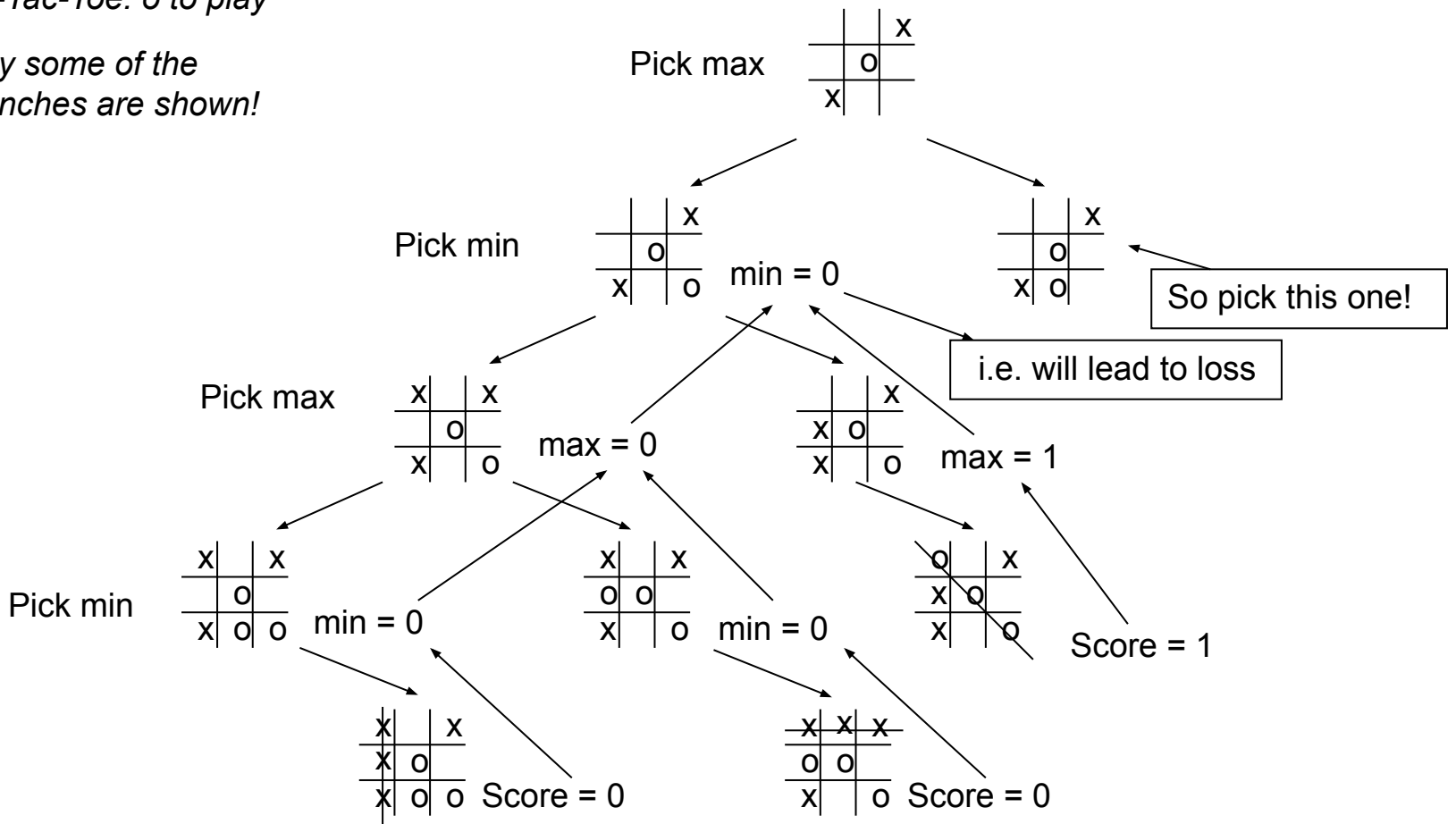
We first consider games with two players; MAX and MIN. MAX moves first, and then they take turns moving until the game is over. Each level of the tree alternates, MAX is trying to maximize score, and MIN is trying to minimize MAX score in order to undermine success



Example Max-Min Strategy

Tic-Tac-Toe: o to play

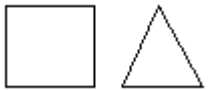
Only some of the Branches are shown!



The mini max algorithm returns the best move for MAX under the assumption that MIN play optimally. What happens when MIN plays sub optimally ?

Optimality is still well defined, even if the opponent isn't playing well. Moreover, if the game tree is small enough that the agent can fully explore it, then the optimal player really doesn't care what the other one does.

Let's say Max goes first. What will Max do? He will look at every possible game sequence. He will then take the action which guarantees that he will get a score of X. No matter what Min does in subsequent moves, Min can never get a score less than X.

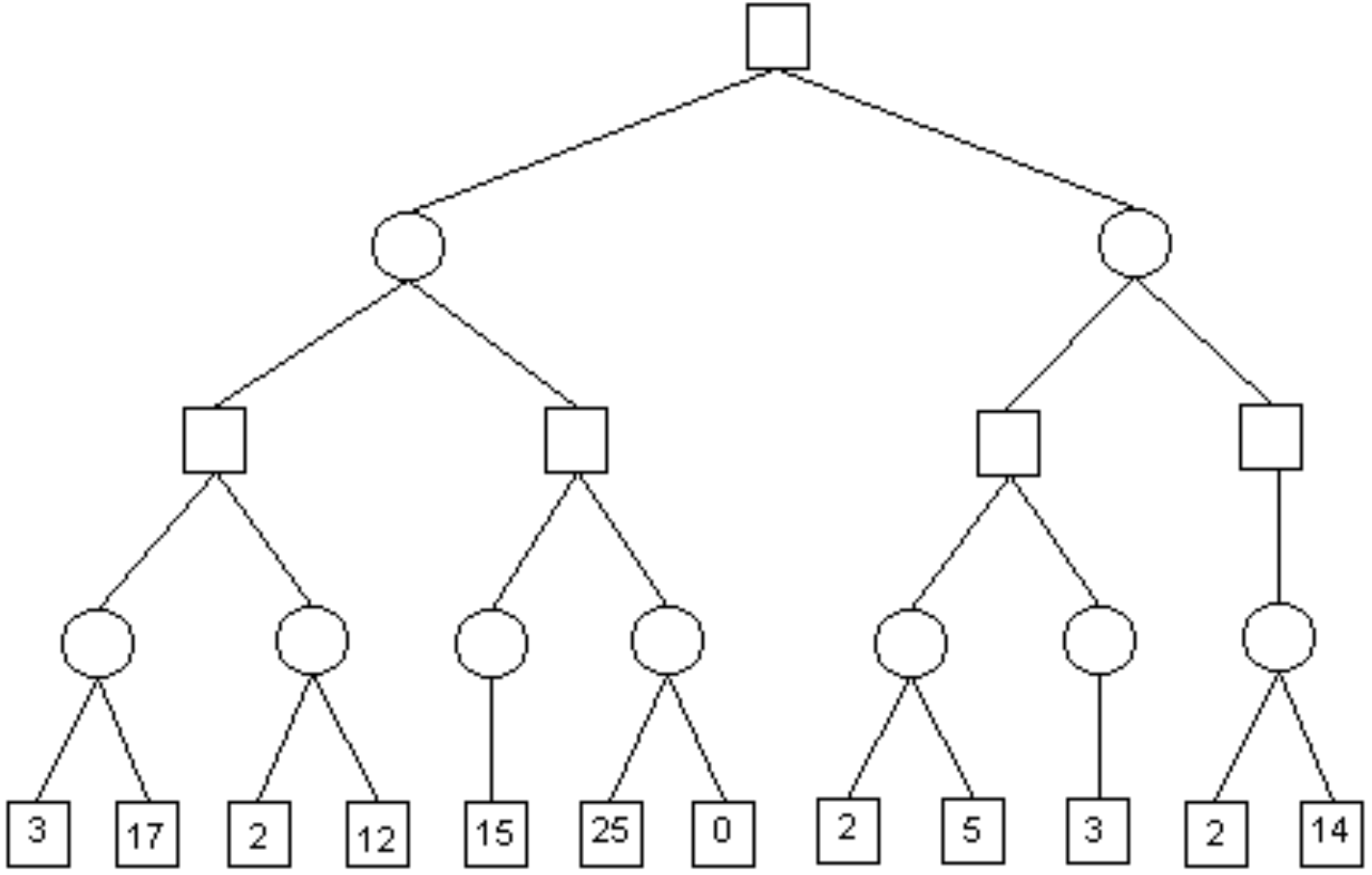


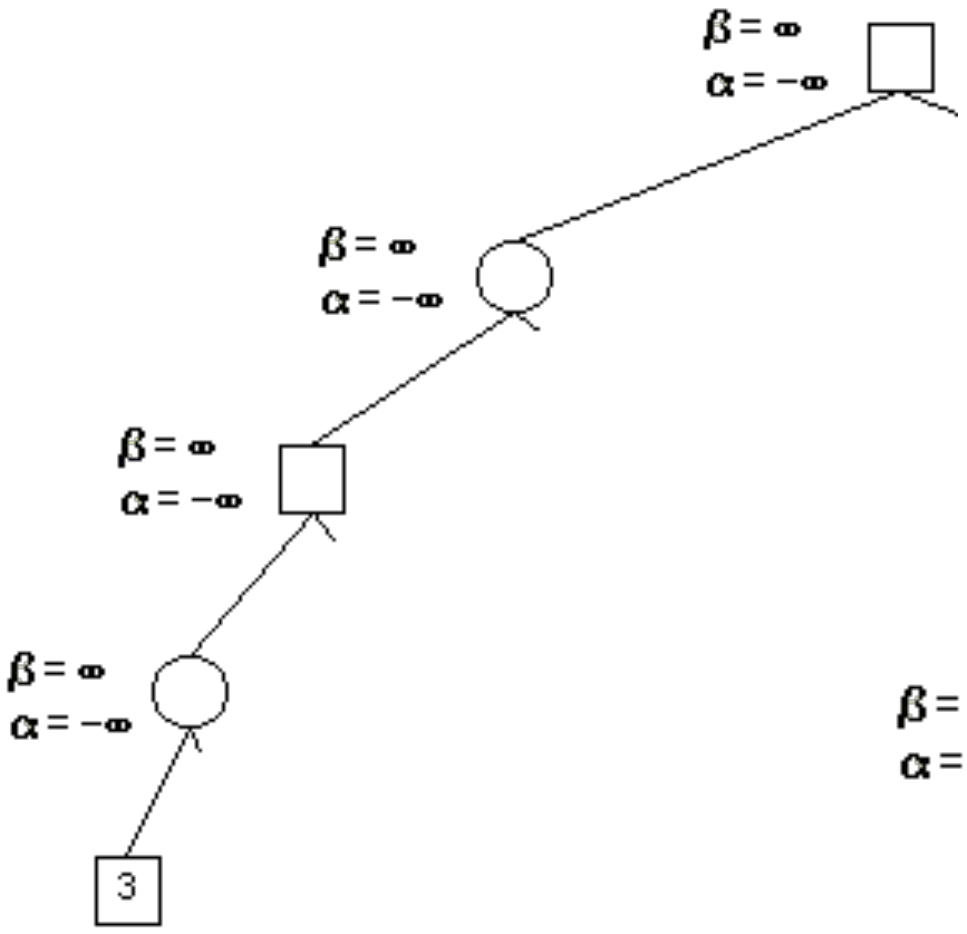
These are also called MAX nodes. The goal at a MAX node is to maximize the value of the subtree rooted at that node. To do this, a MAX node chooses the child with the greatest value, and that becomes the value of the MAX nod

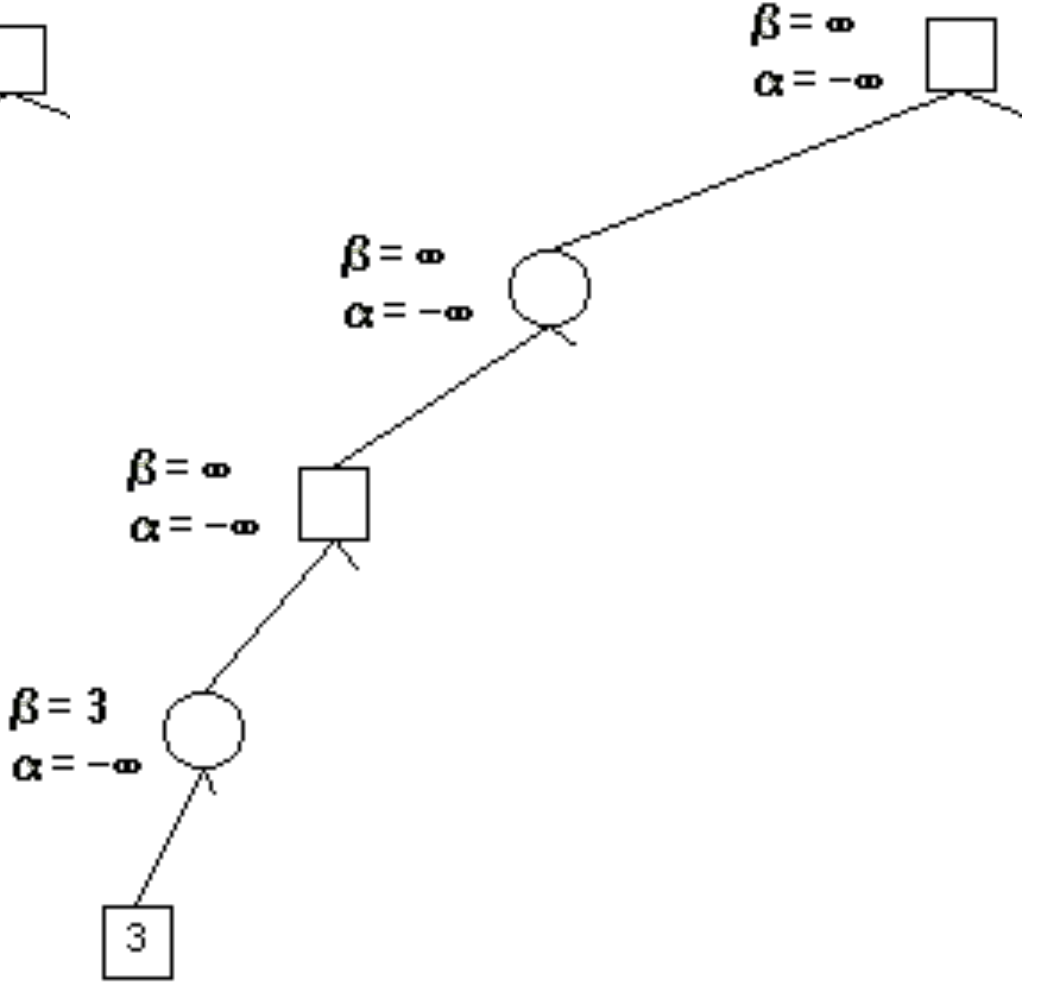
These are also called MIN nodes. The goal at a MIN node is to minimize the value of the subtree rooted at that node. To do this, a MIN node chooses the child with the least (smallest) value, and that becomes the value of the MIN node

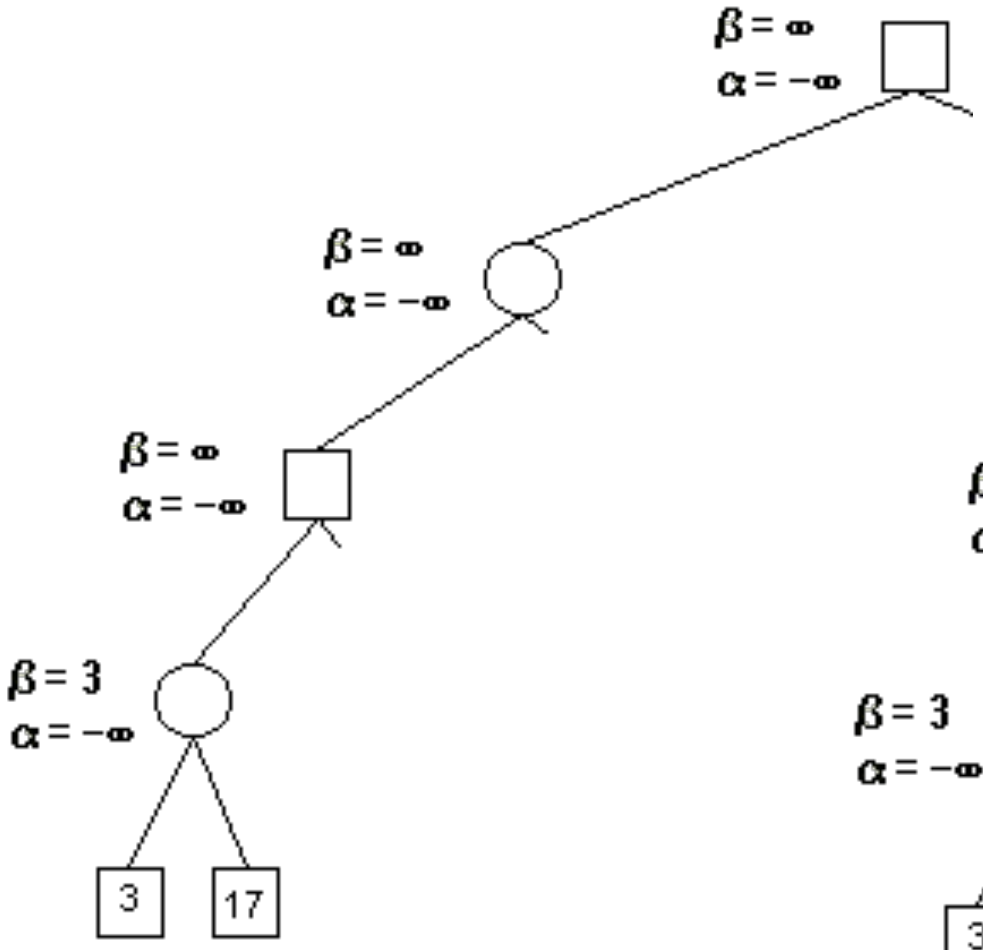
β = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
α = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.

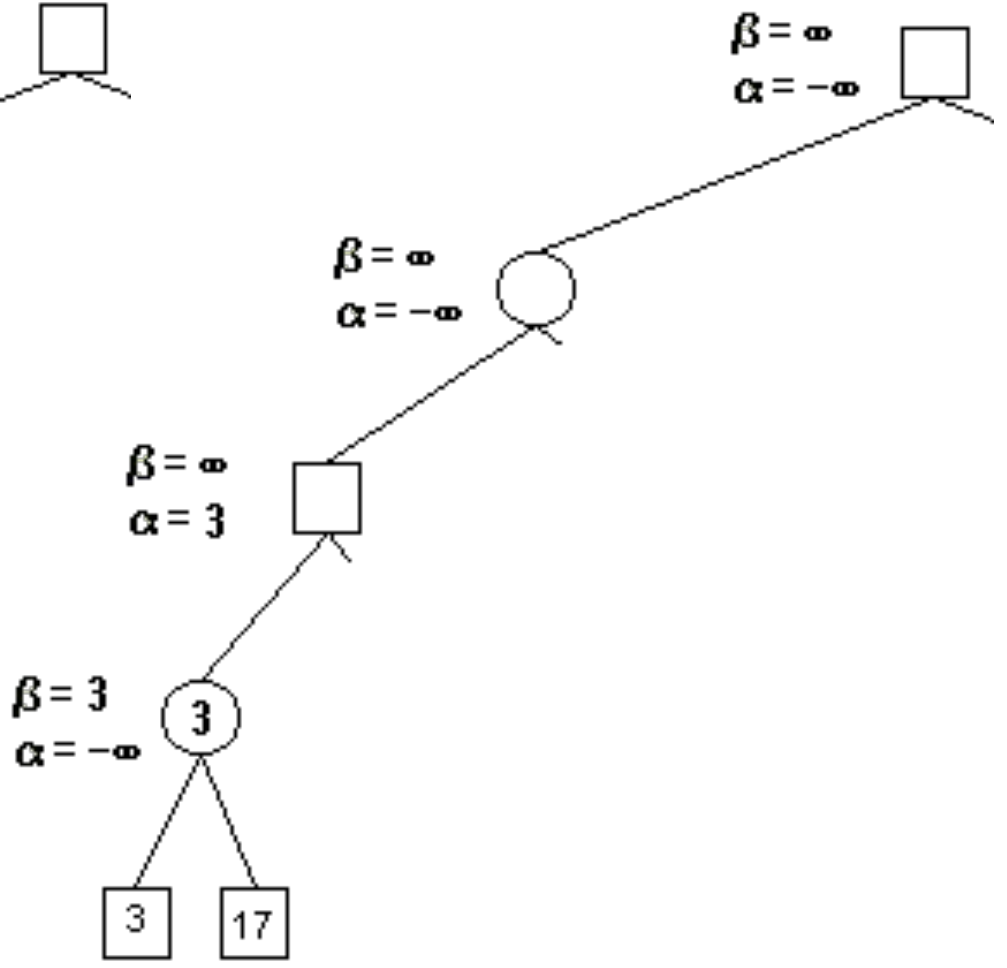
Demonstrate alpha-beta pruning

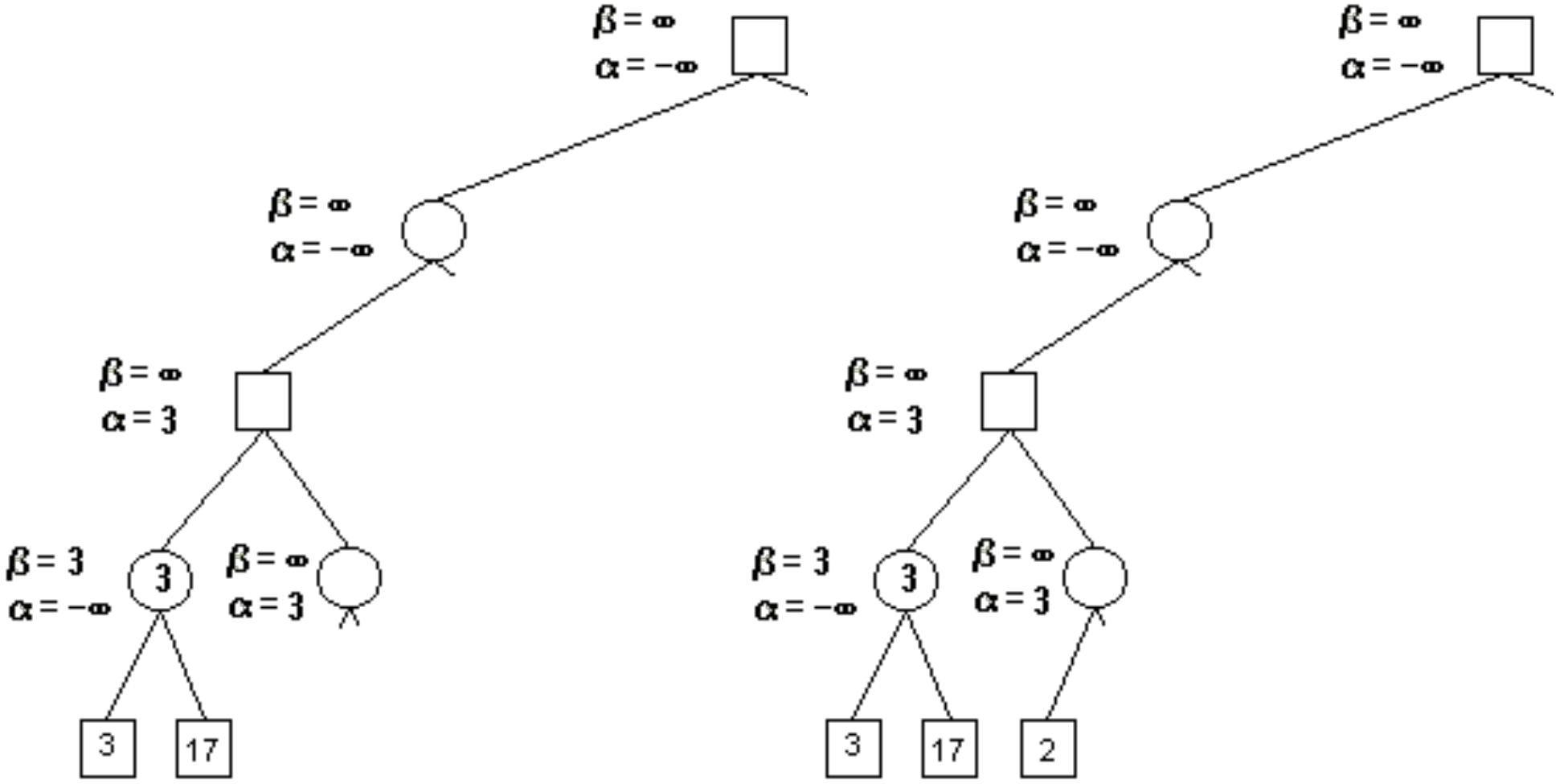


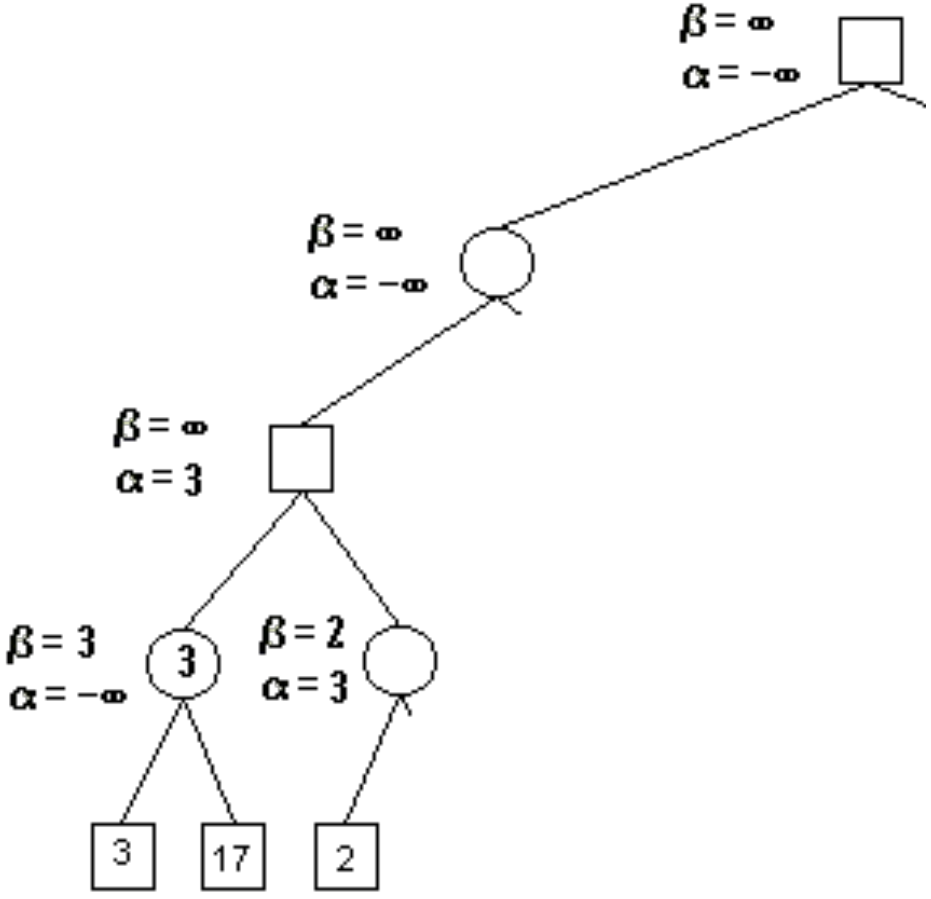

β = ∞
α = −∞
β = ∞
α = −∞
β = ∞
α = −∞
β = ∞
α = −∞
3

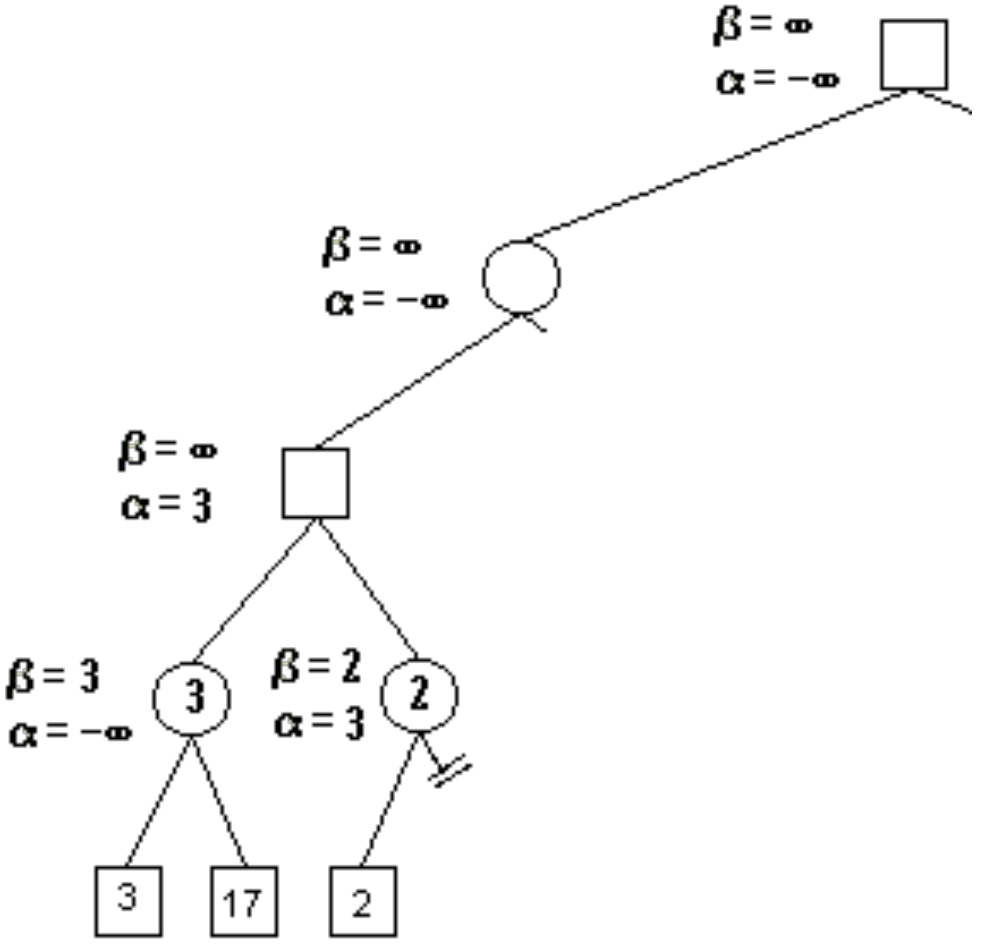

β = ∞
α = −∞
β = ∞
α = −∞
β = ∞
α = −∞
β = 3
α = −∞
3

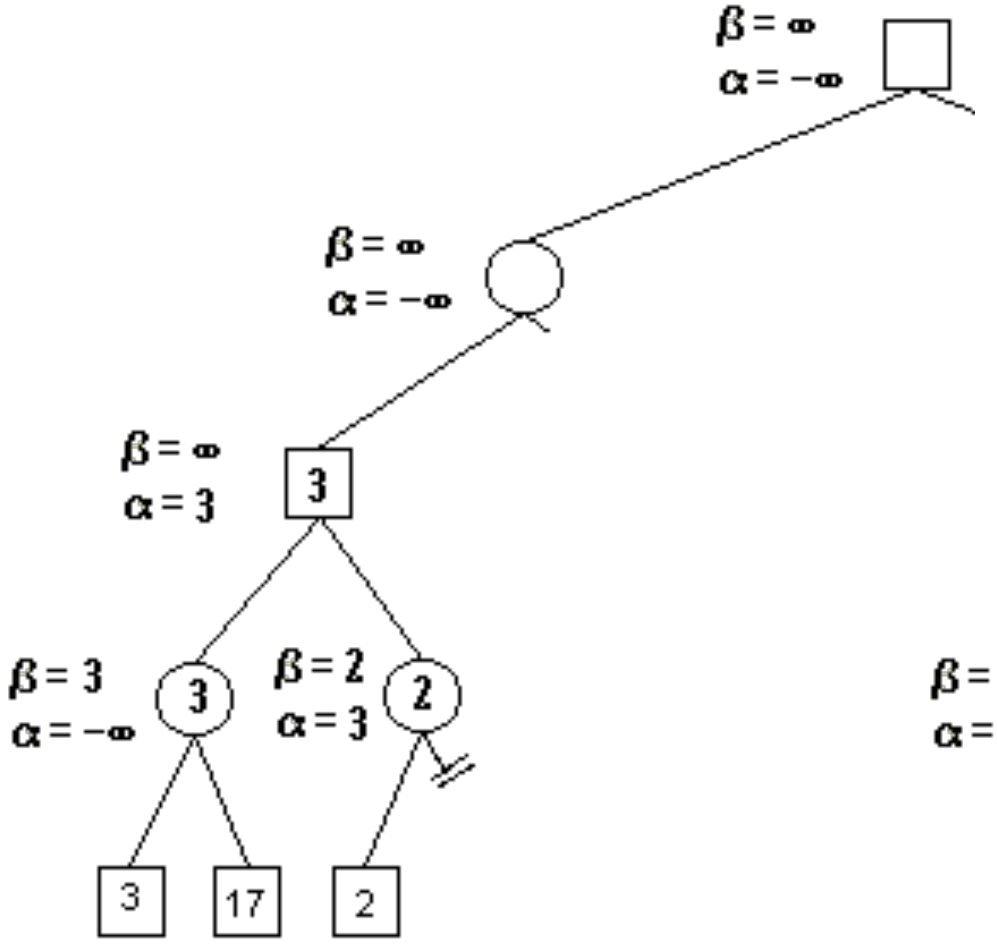

β = ∞
α = −∞
β = ∞
α = −∞
β = ∞
α = −∞
β = 3
α = −∞
3
17

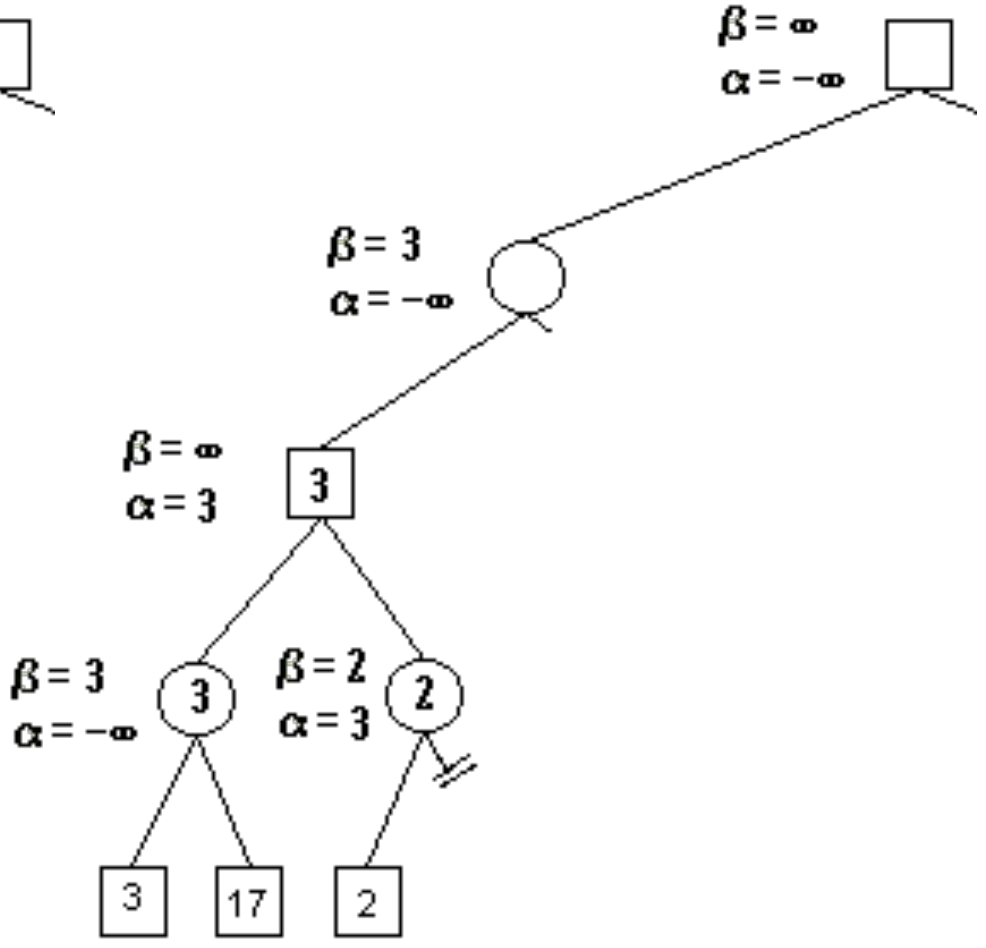

β = ∞
α = −∞
β = ∞
α = −∞
β = ∞
α = 3
β = 3
α = −∞
3
3
17

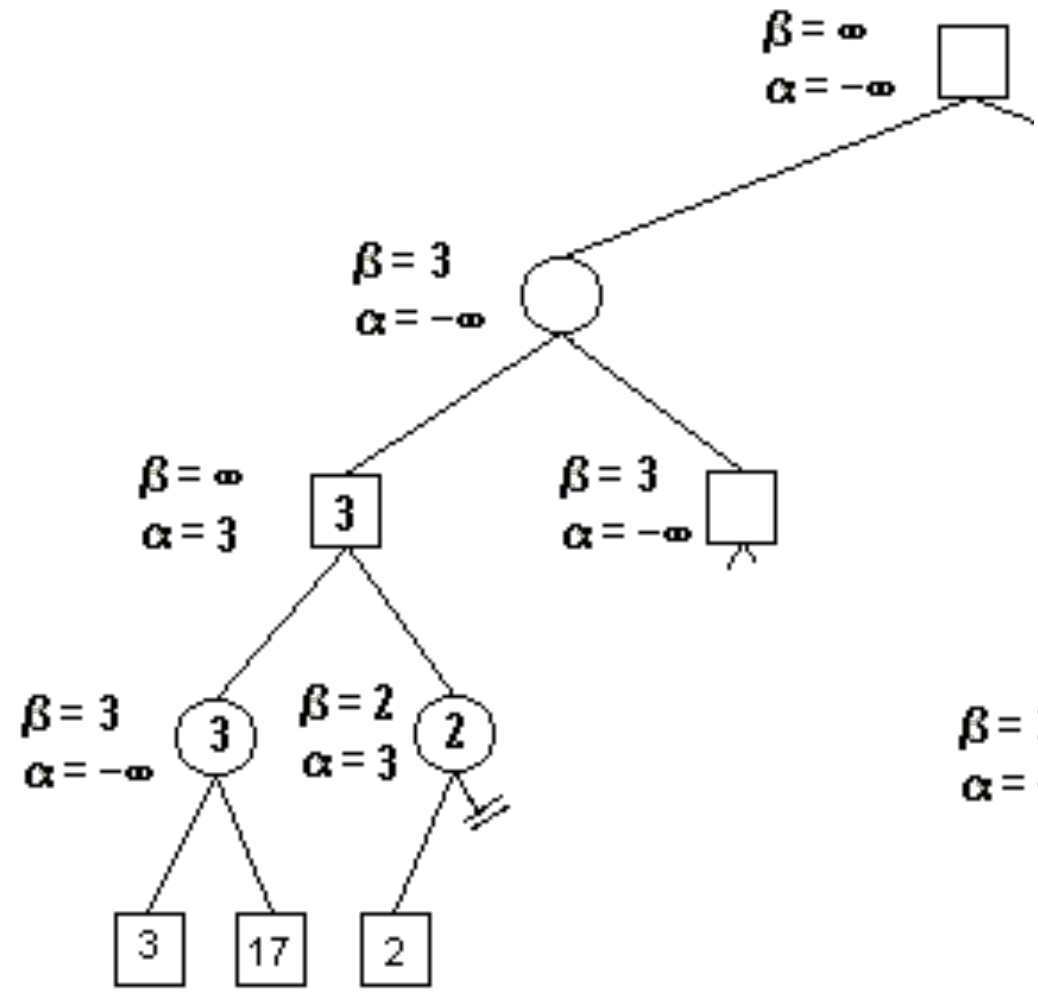

β = ∞
α = −∞
β = ∞
α = −∞
β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
3
17
β = ∞
α = −∞
β = ∞
α = −∞
β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
3
17
2

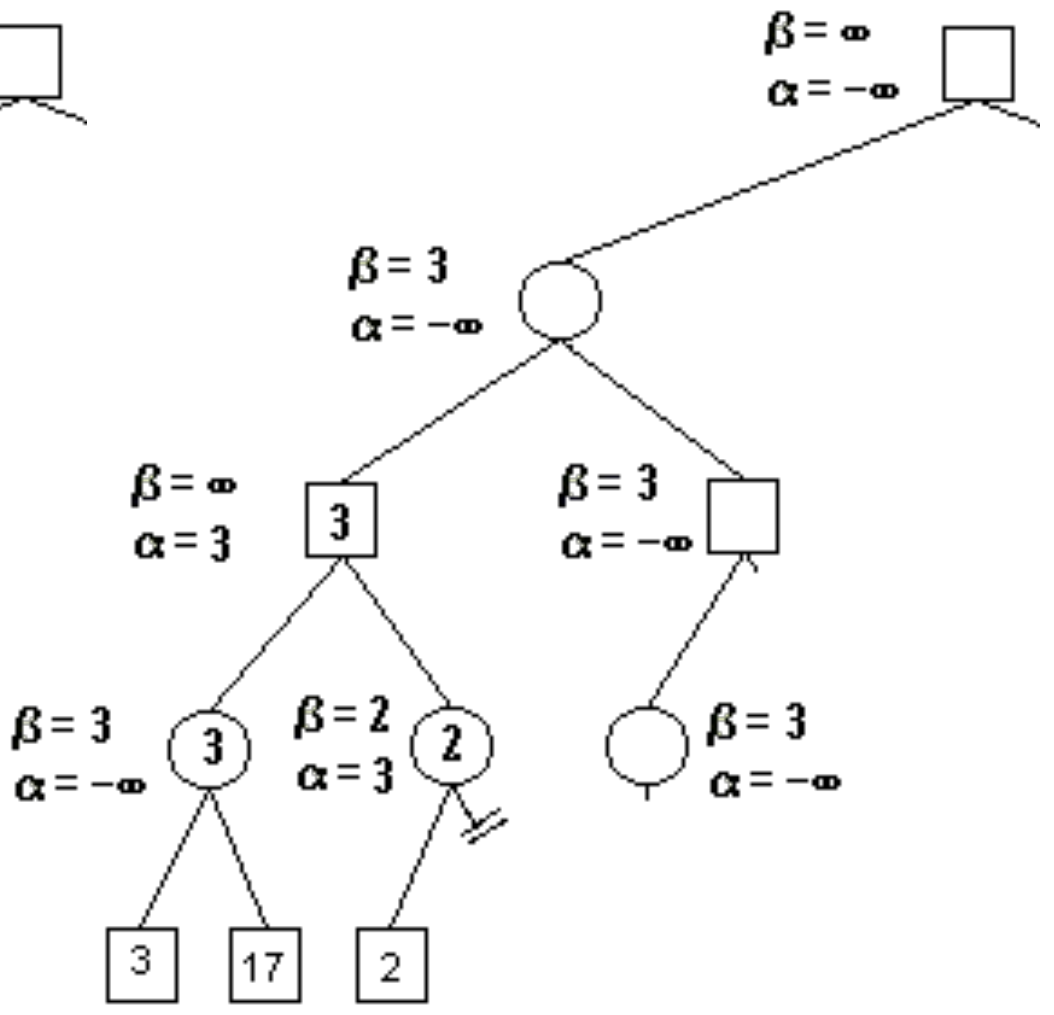

β = ∞
α = −∞
β = ∞
α = −∞
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
3
17
2

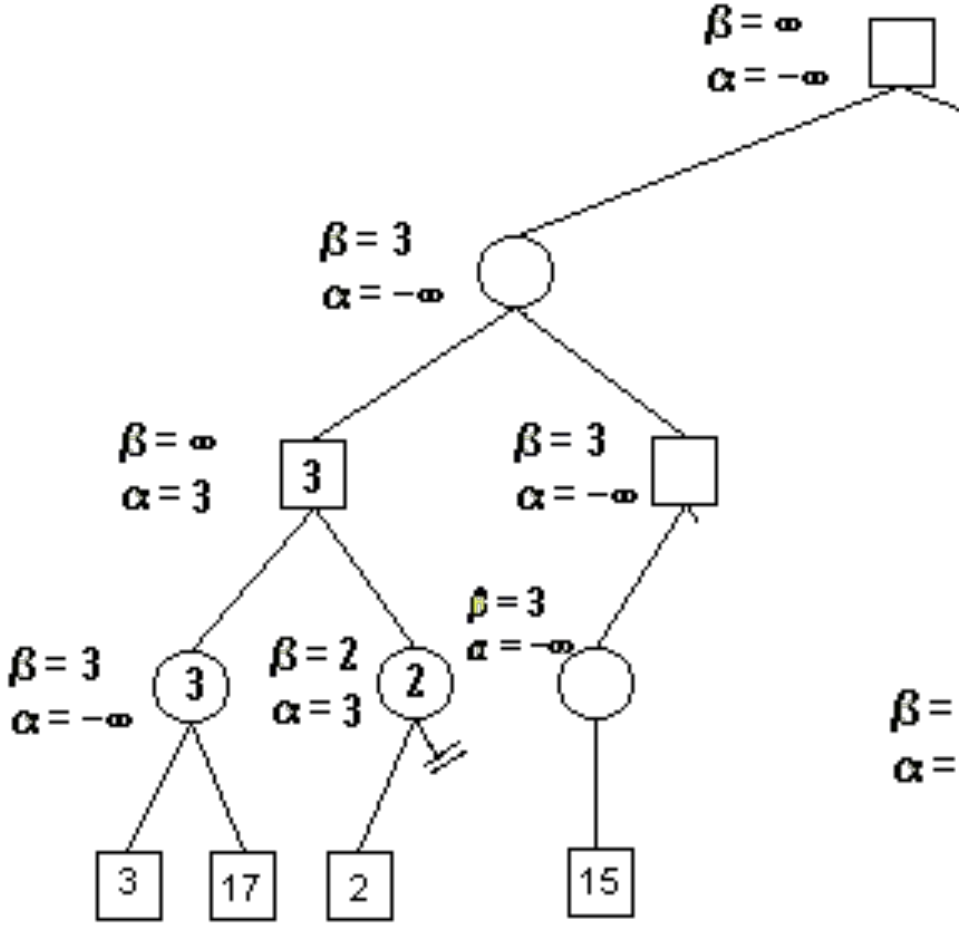

β = ∞
α = −∞
β = ∞
α = −∞
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
2
3
17
2

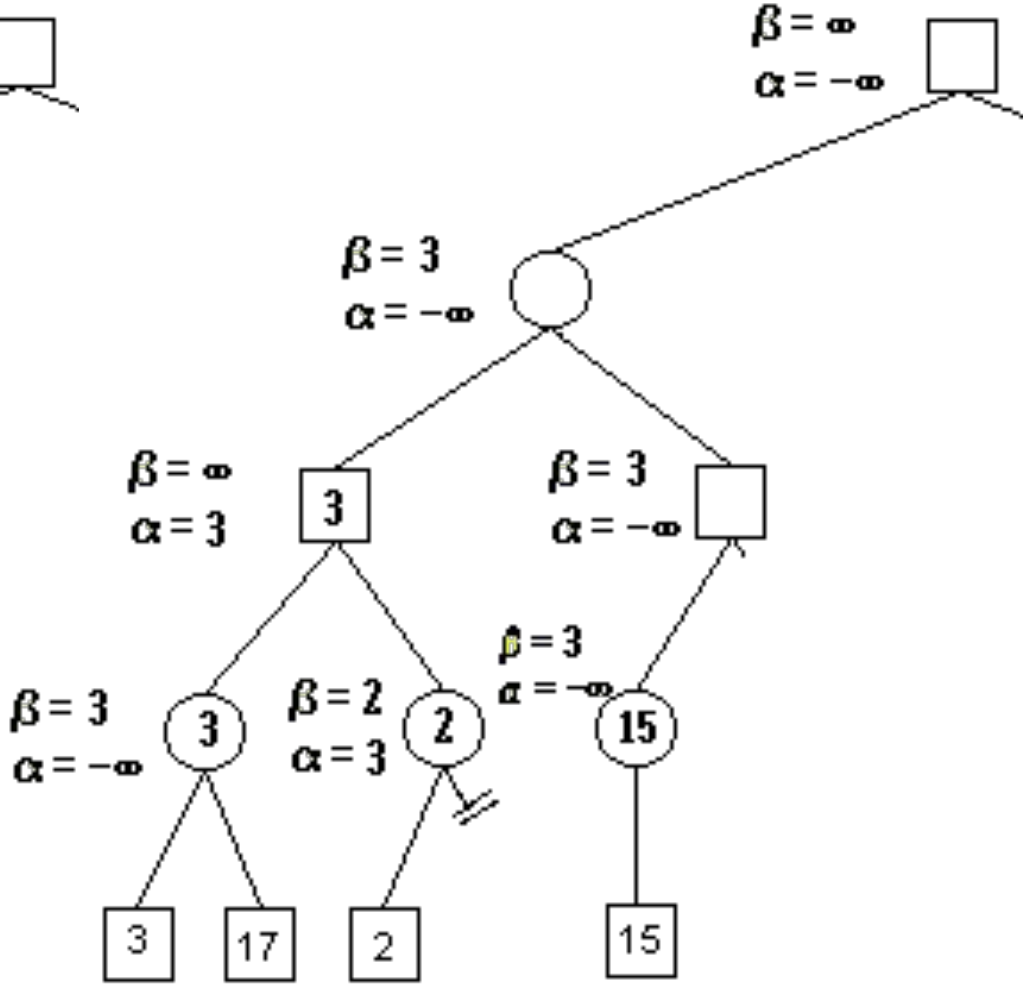

β = ∞
α = −∞
β = ∞
α = −∞
β = ∞
α = 3
3
β = 3
α = −∞
3
β = 2
α = 3
2
3
17
2

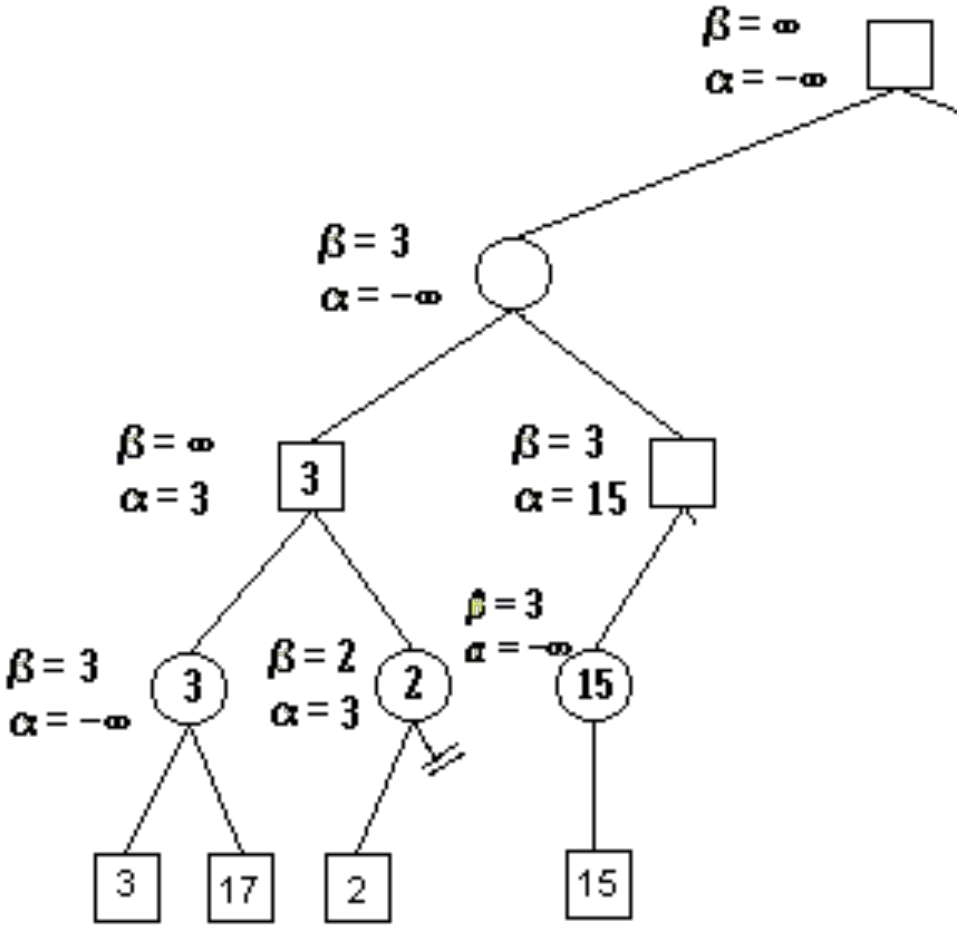

β = ∞
α = −∞
β = 3
α = −∞
β = ∞
α = 3
3
β = 3
α = −∞
3
β = 2
α = 3
2
3
17
2

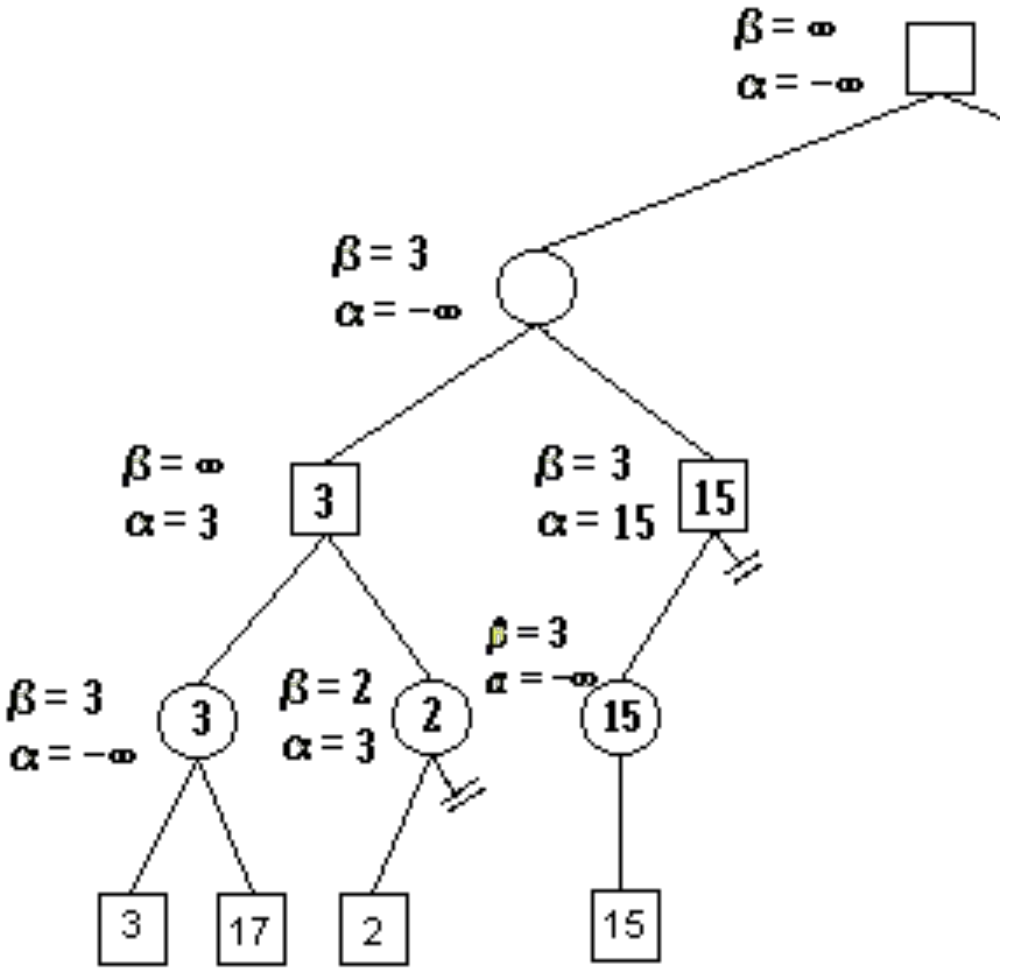

β = ∞
α = −∞
β = 3
α = −∞
β = ∞
α = 3
3
β = 3
α = −∞
β = 3
α = −∞
3
β = 2
α = 3
2
3
17
2

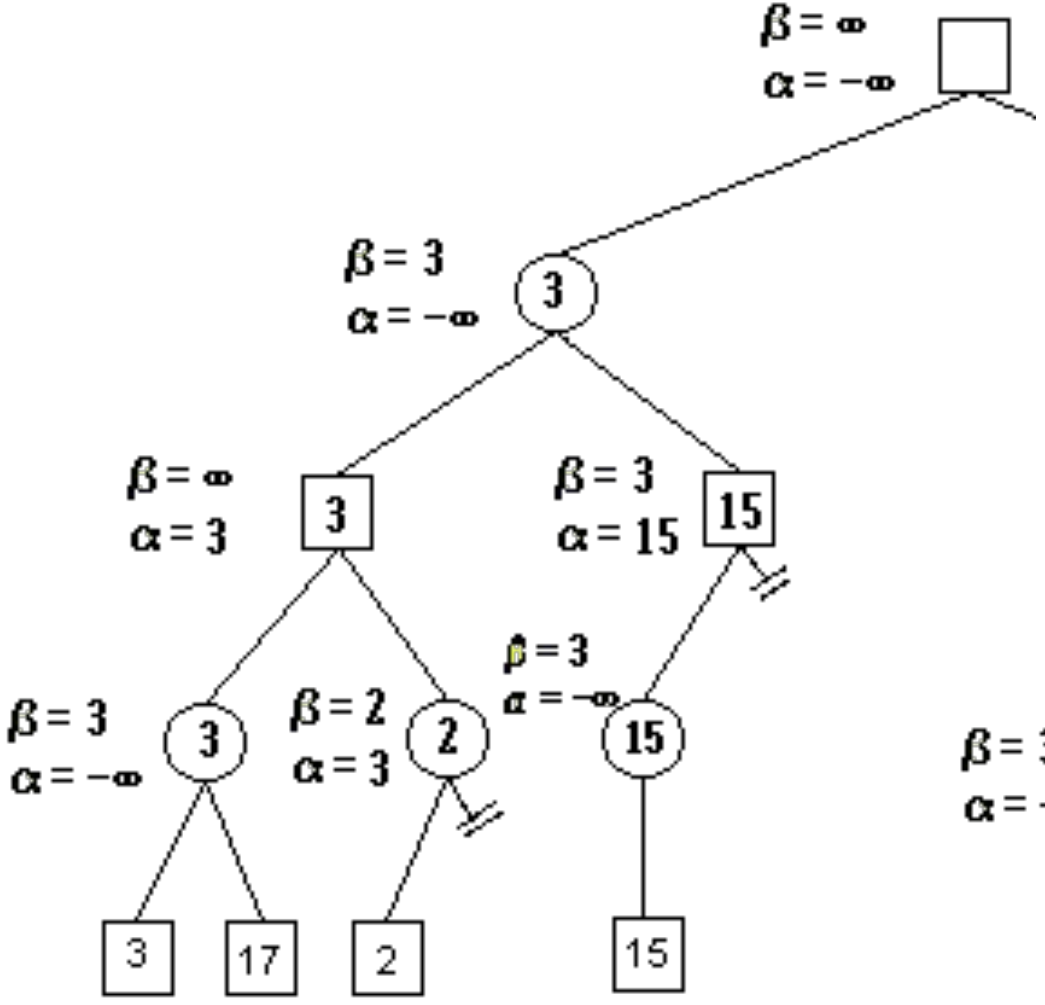

β = ∞
α = −∞
β = 3
α = −∞
β = ∞
α = 3
3
β = 3
α = −∞
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
3
17
2

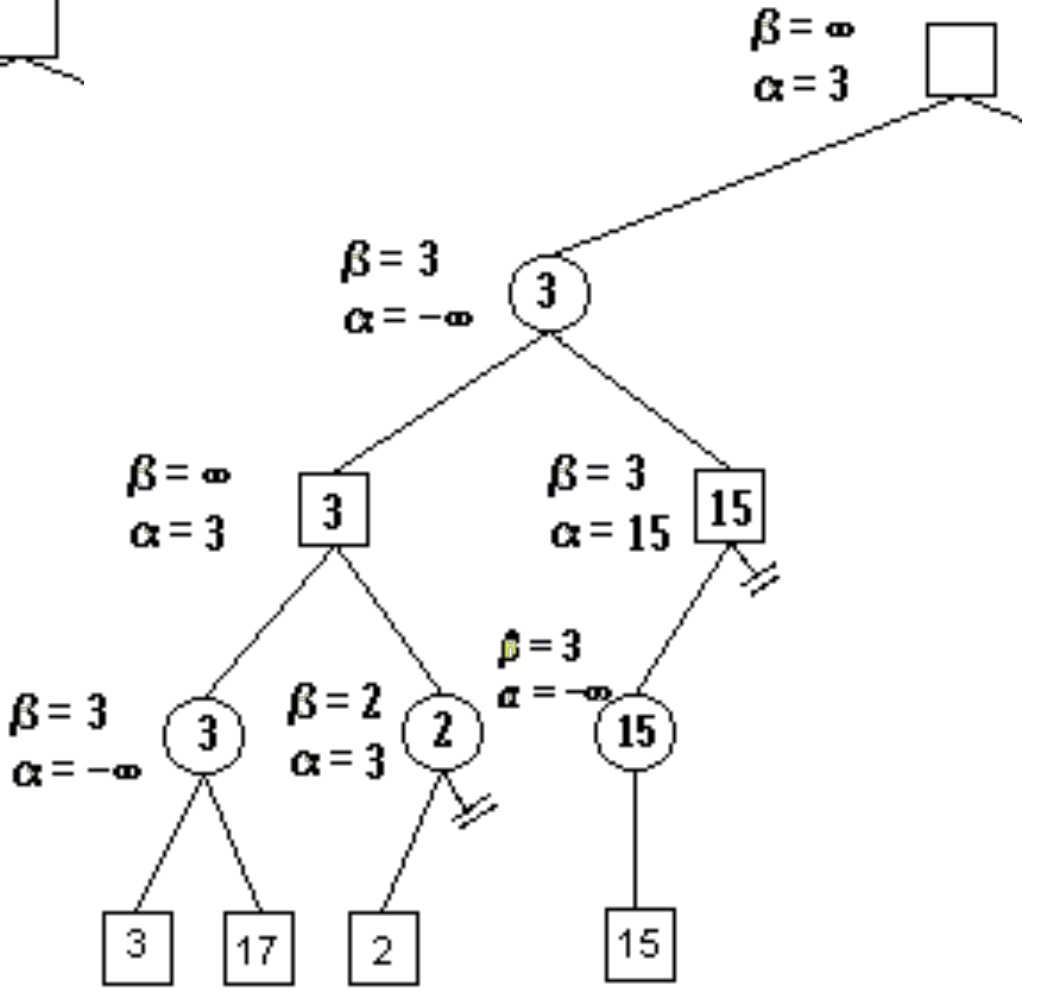

β = ∞
α = −∞
β = 3
α = −∞
β = ∞
α = 3
3
β = 3
α = −∞
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
3
17
2
15

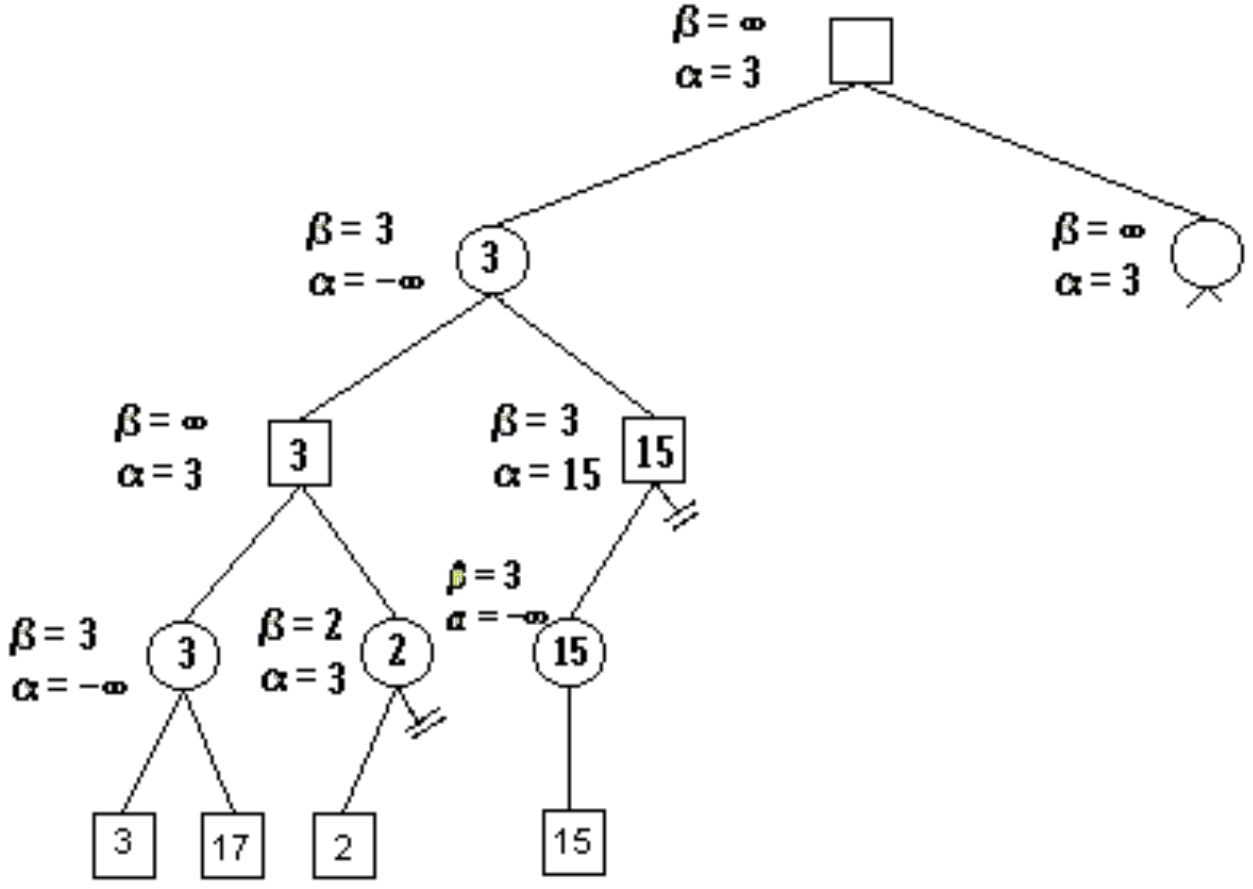

β = ∞
α = −∞
β = 3
α = −∞
β = ∞
α = 3
3
β = 3
α = −∞
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
3
17
2
15

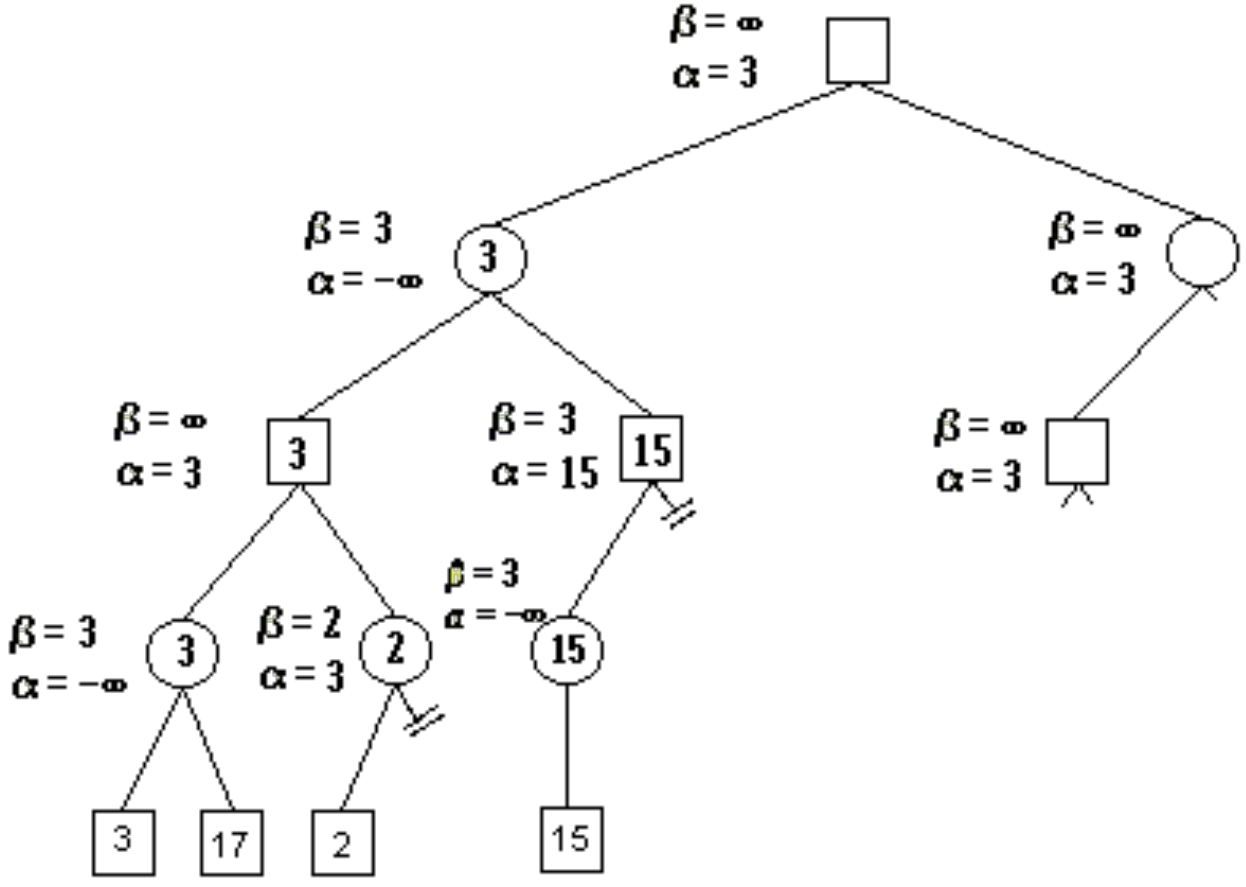

β = ∞
α = −∞
β = 3
α = −∞
β = ∞
α = 3
3
β = 3
α = 15
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
3
17
2
15

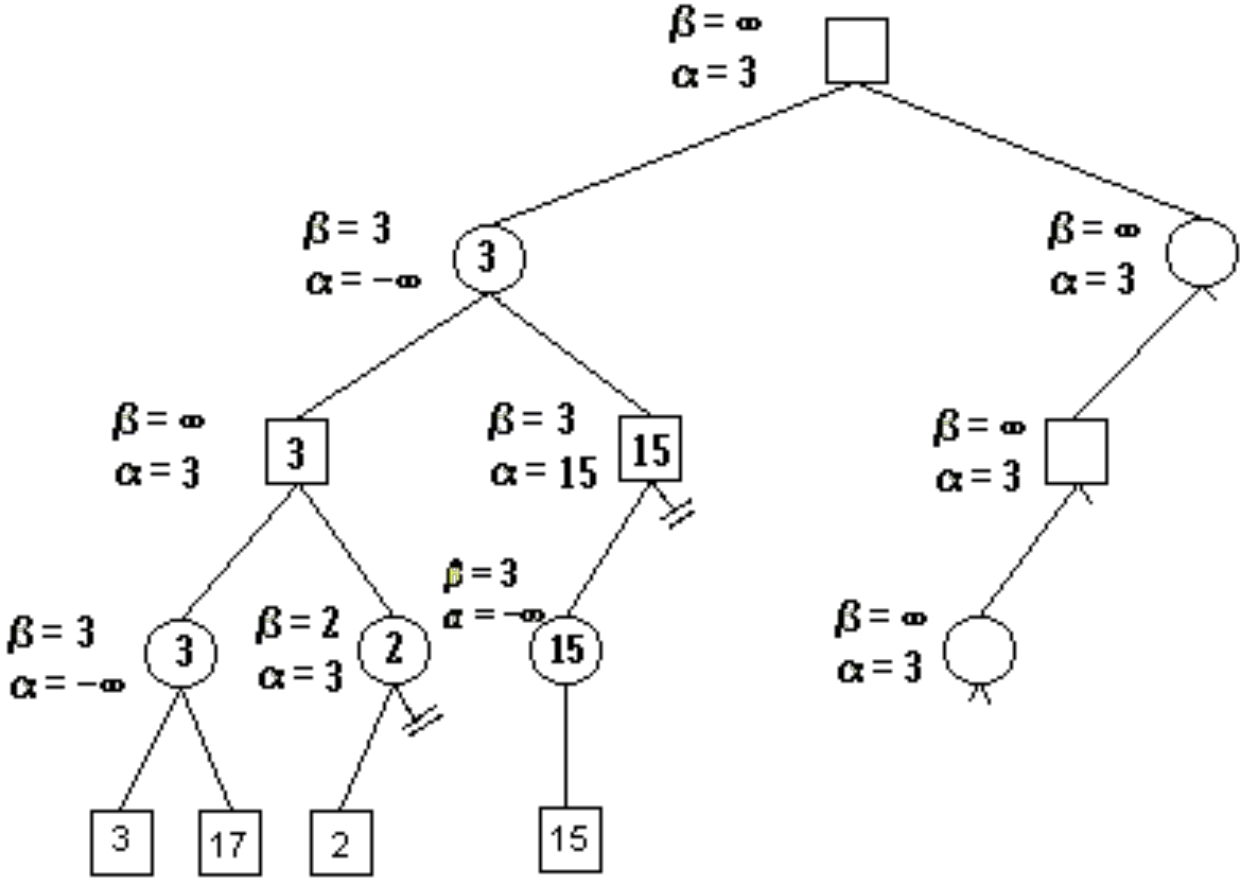

β = ∞
α = −∞
β = 3
α = −∞
β = ∞
α = 3
3
β = 3
α = 15
15
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
3
17
2
15

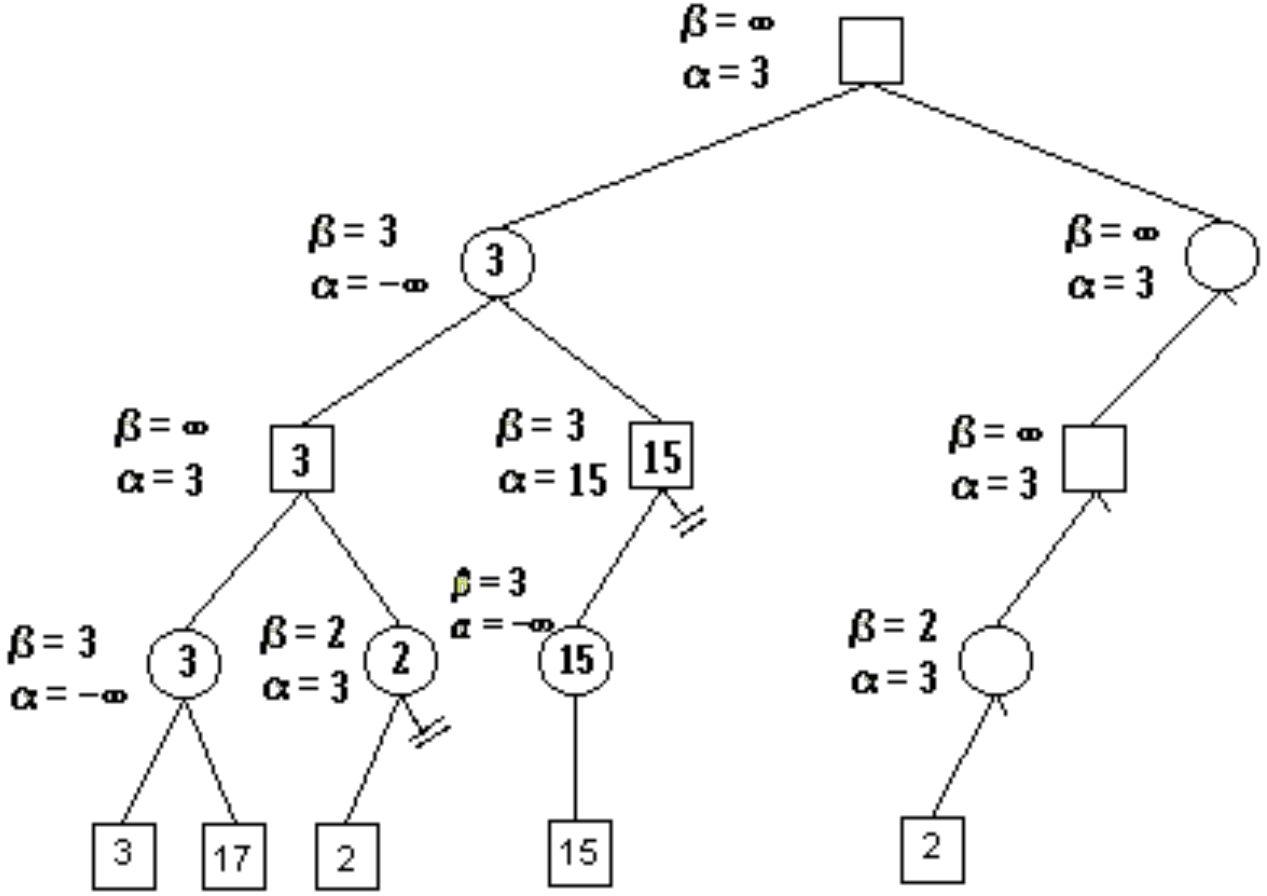

β = ∞
α = −∞
β = 3
α = −∞
3
β = ∞
α = 3
3
β = 3
α = 15
15
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
3
17
2
15

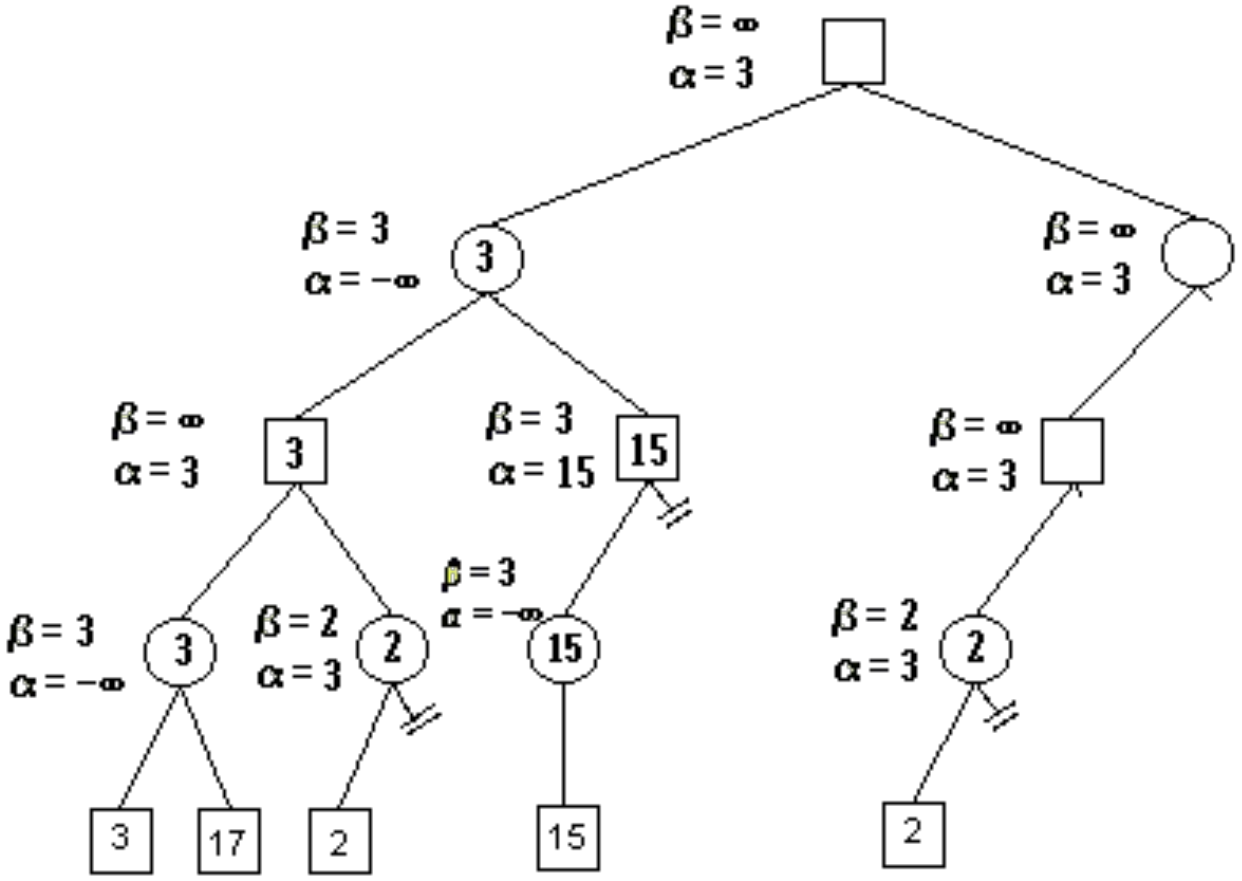

β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
3
β = 3
α = 15
15
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
3
17
2
15

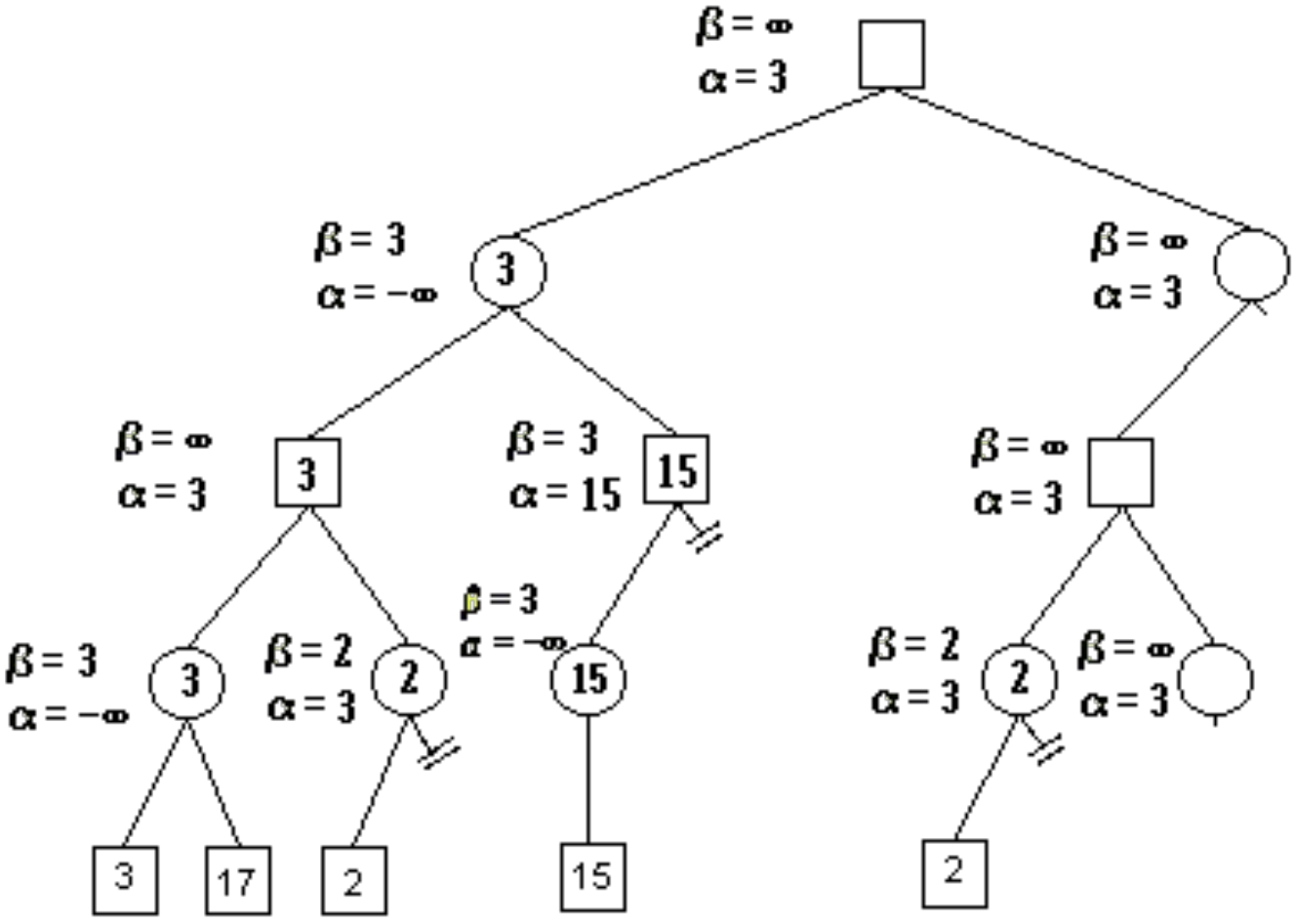

β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = 3
α = −∞
β = 3
α = −∞
3
β = 2
α = 3
2
15
3
17
2
15

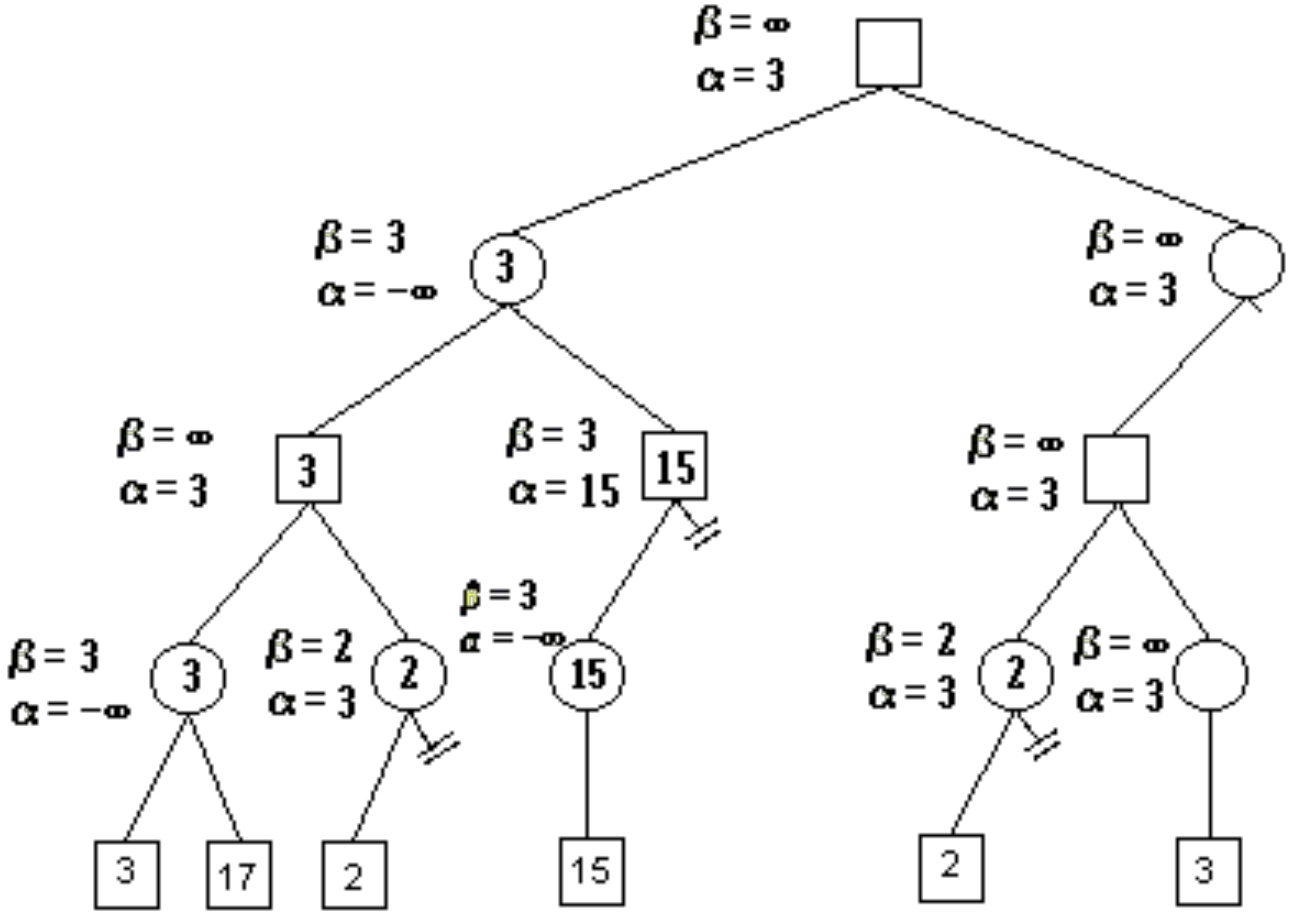

β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
3
17
2
15

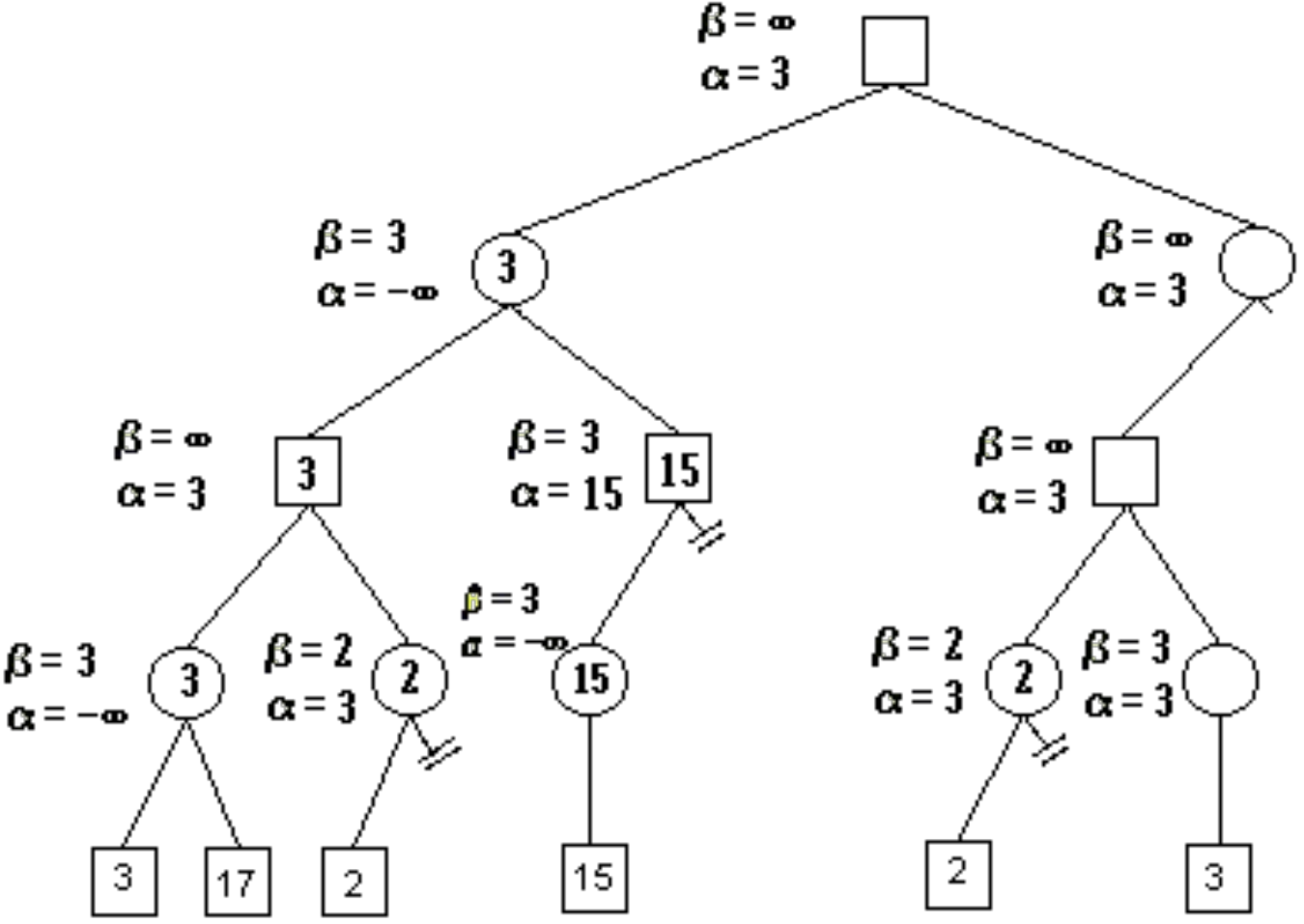

β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
β = ∞
α = 3
3
17
2
15

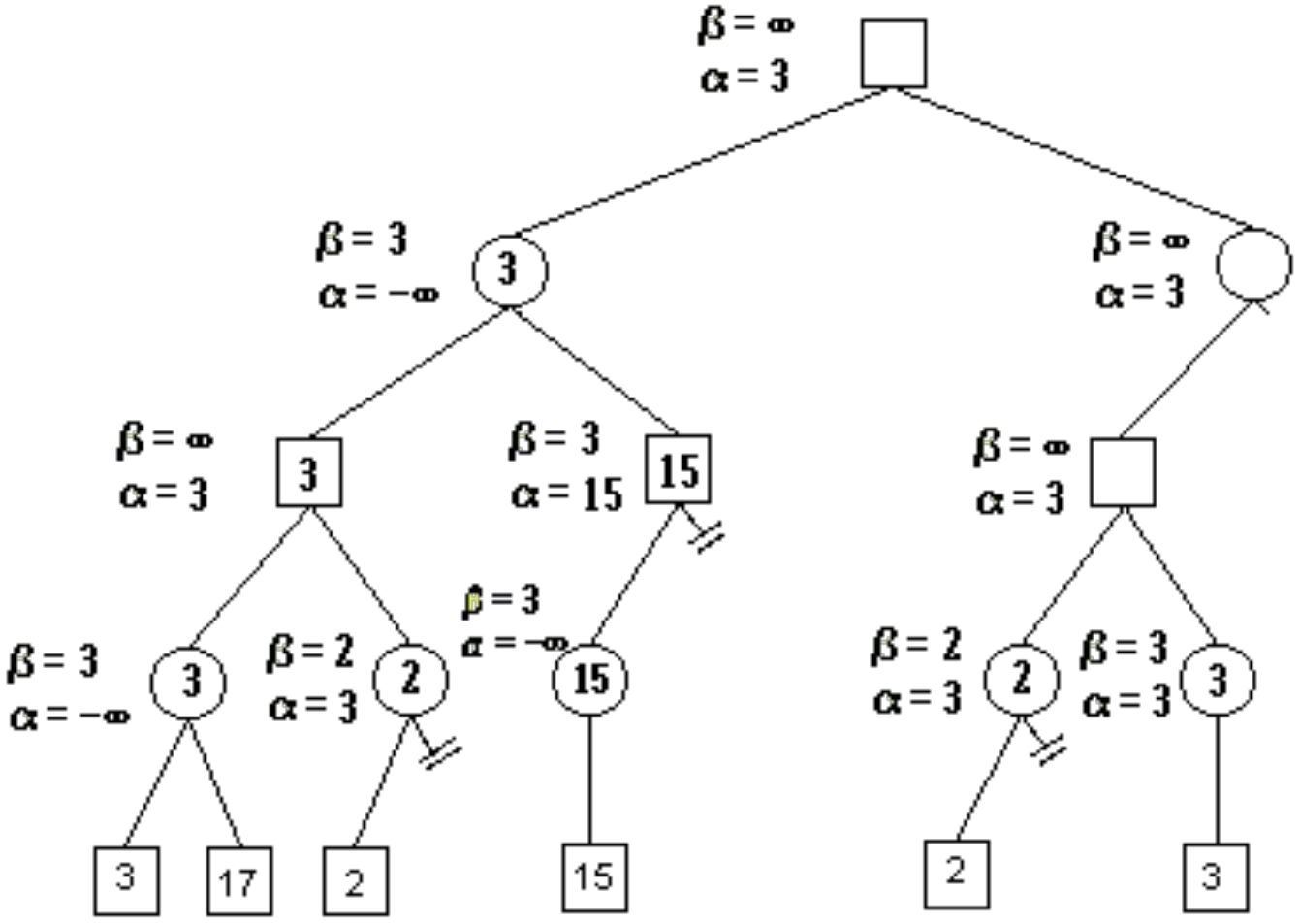

β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
β = 2
α = 3
3
17
2
15
2

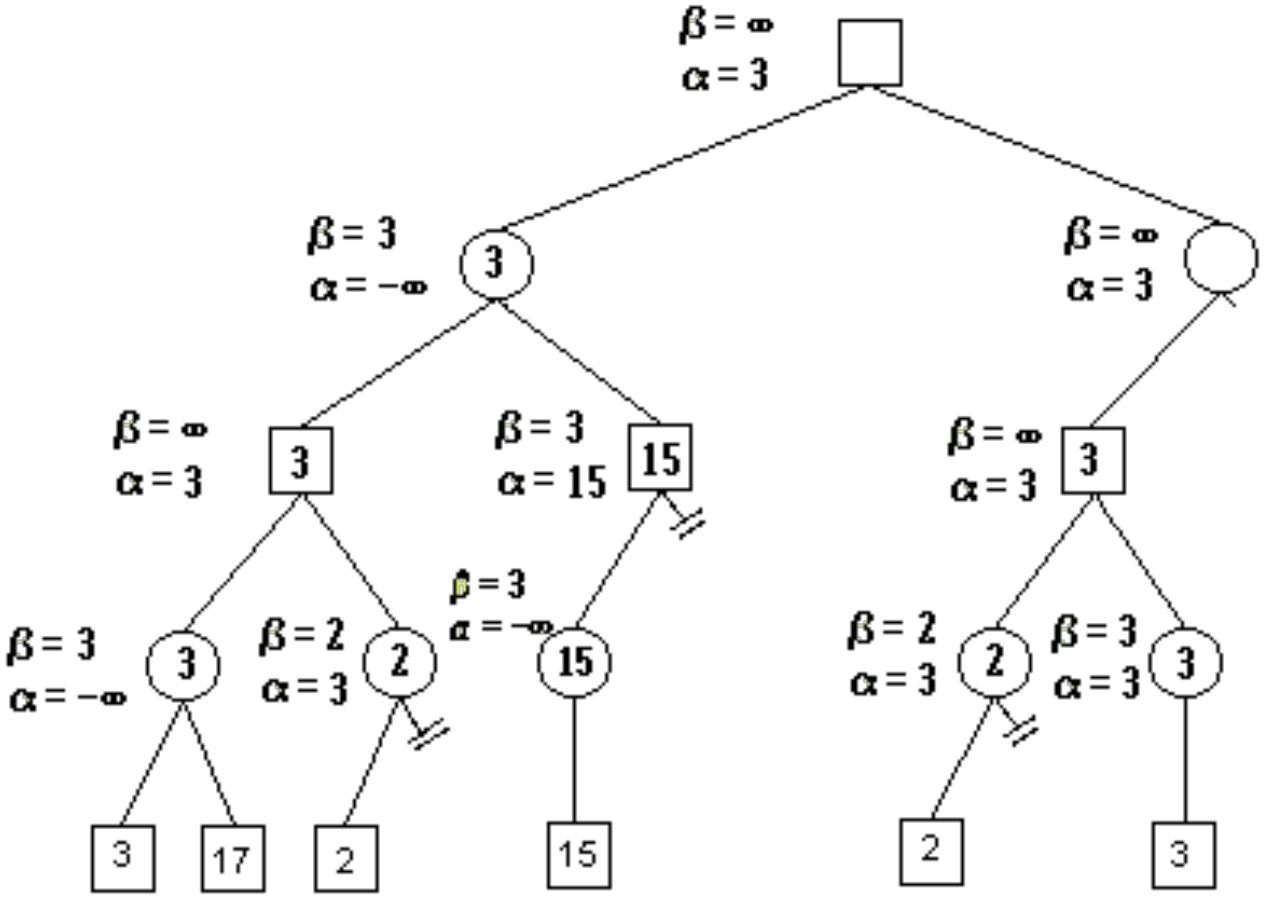

β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
β = 2
α = 3
2
3
17
2
15
2

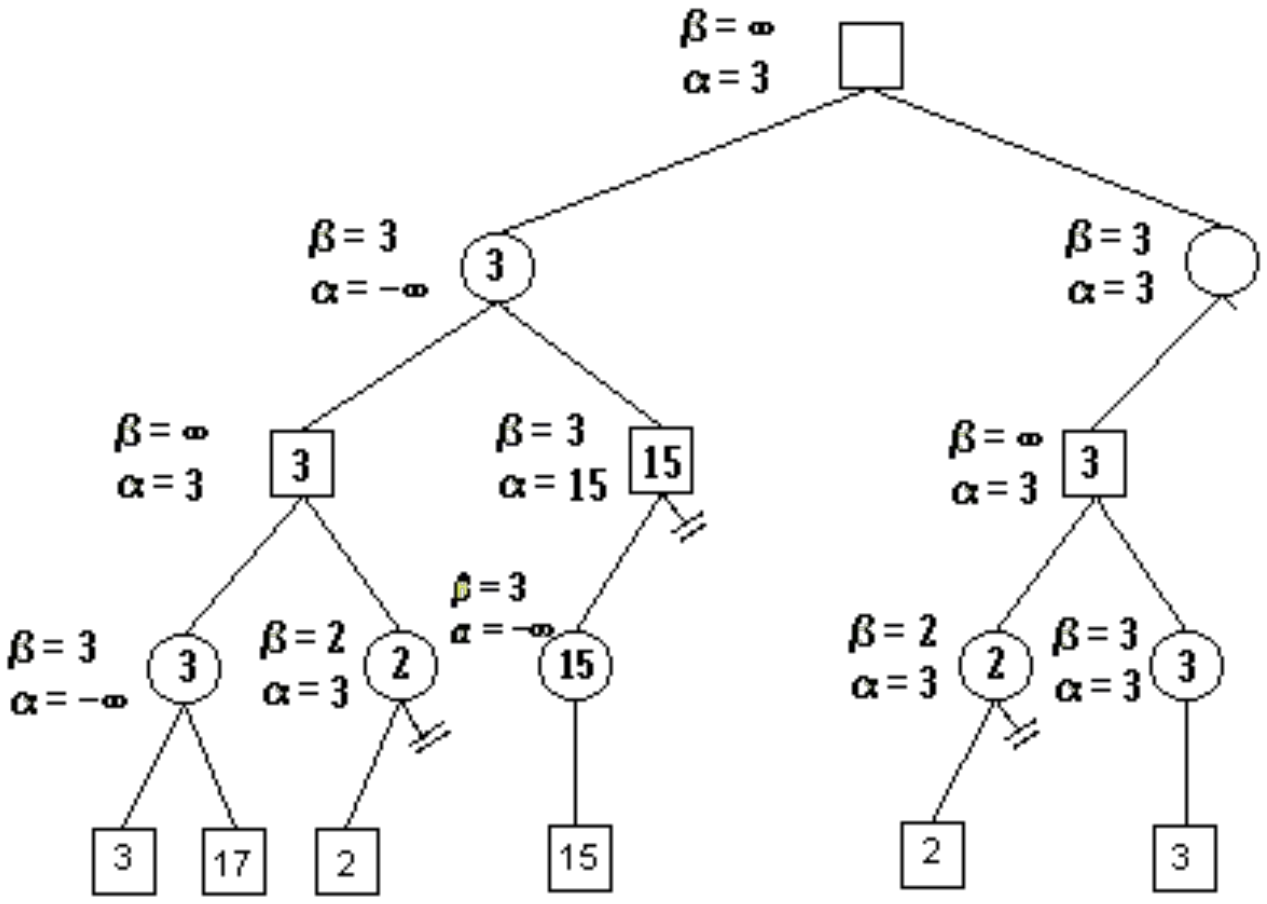

β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
β = 2
α = 3
2
β = ∞
α = 3
3
17
2
15
2


β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
β = 2
α = 3
2
β = ∞
α = 3
3
17
2
15
2
3


β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
β = 2
α = 3
2
β = 3
α = 3
3
17
2
15
2
3


β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
β = 2
α = 3
2
β = 3
α = 3
3
3
17
2
15
2
3


β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
β = 2
α = 3
2
β = 3
α = 3
3
3
17
2
15
2
3


β = ∞
α = 3
β = 3
α = −∞
3
β = 3
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
β = 2
α = 3
2
β = 3
α = 3
3
3
17
2
15
2
3

4. Inference and Reasoning

4.1 Inference Theorems

4.2 Deduction and truth maintenance

4.3 Heuristic search state-space representation

4.4 Game Playing

4.5 Reasoning about uncertainty probability

4.6 Bayesian Networks

4.7 Case-based Reasoning