

CHAPTER 2

SEARCH TECHNIQUES

Introduction to Data Structures

Stacks

Queues

Linked lists

Trees

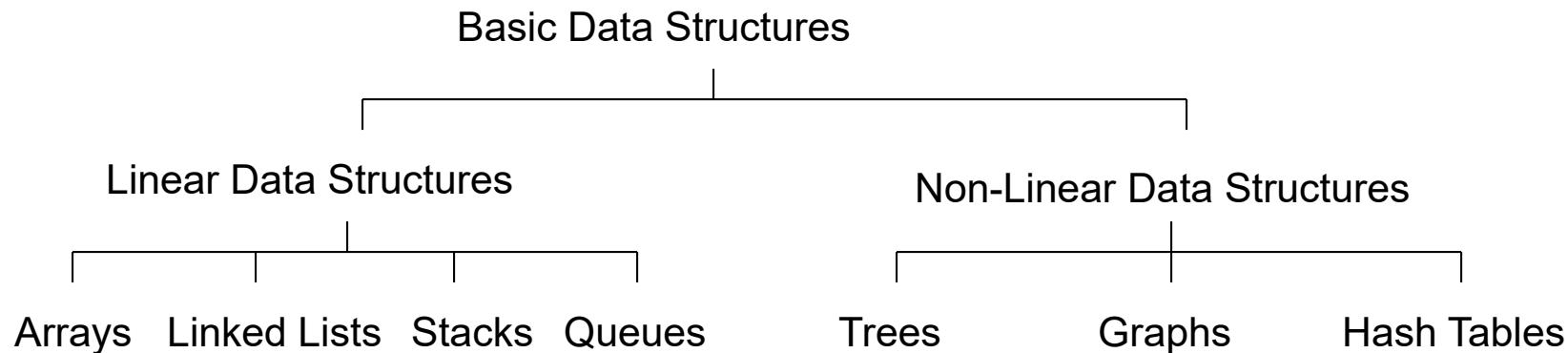
Graphs

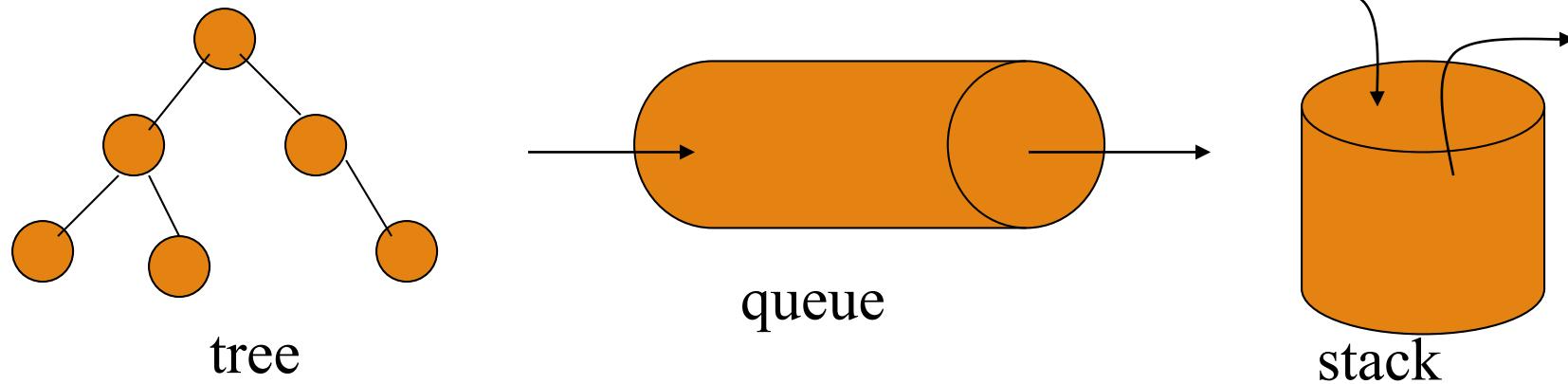
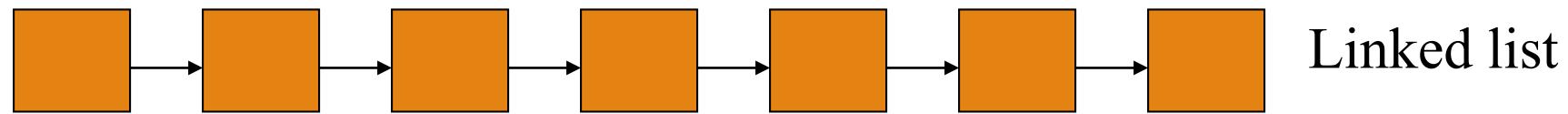
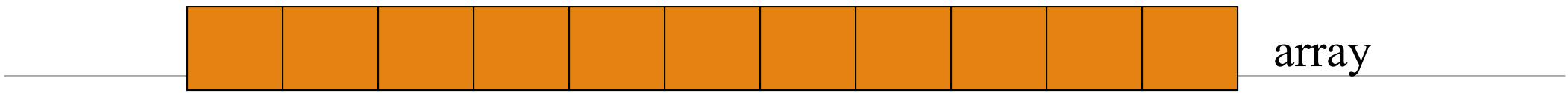
What is Data Structure?

Data structure is a representation of data and the operations allowed on that data.

- A data structure is a way to store and organize data in order to facilitate the access and modifications.
- Data Structure are the method of representing of logical relationships between individual data elements related to the solution of a given problem.

Basic Data Structure





Selection of Data Structure

The choice of particular data model depends on two consideration:

- It must be rich enough in structure to represent the relationship between data elements
- The structure should be simple enough that one can effectively process the data when necessary

Types of Data Structure

Linear: In Linear data structure, values are arranged in linear fashion.

- Array: Fixed-size
- Linked-list: Variable-size
- Stack: Add to top and remove from top
- Queue: Add to back and remove from front
- Priority queue: Add anywhere, remove the highest priority

Types of Data Structure

Non-Linear: The data values in this structure are not arranged in order.

- Hash tables: Unordered lists which use a ‘hash function’ to insert and search
- Tree: Data is organized in branches.
- Graph: A more general branching structure, with less strict connection conditions than for a tree

Type of Data Structures

Homogenous: In this type of data structures, values of the same types of data are stored.

- Array

Non-Homogenous: In this type of data structures, data values of different types are grouped and stored.

- Structures
- Classes

The Core Operations of ADT

Every Collection ADT should provide a way to:

- add an item
- remove an item
- find, retrieve, or access an item

Stacks

Collection with access only to the last element inserted

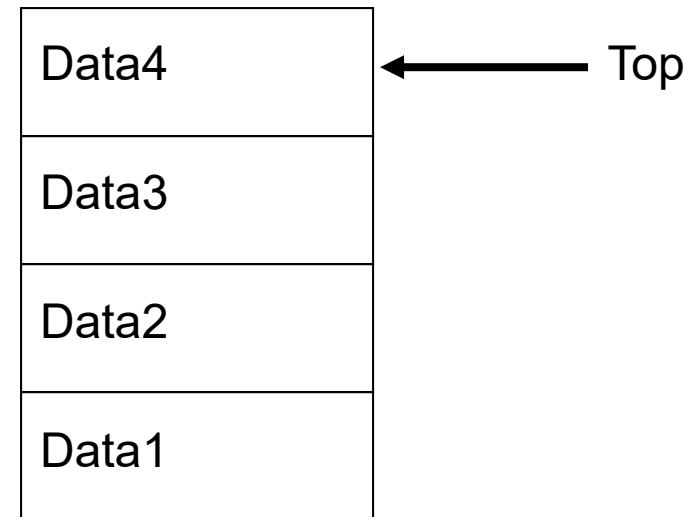
Last in first out

insert/push

remove/pop

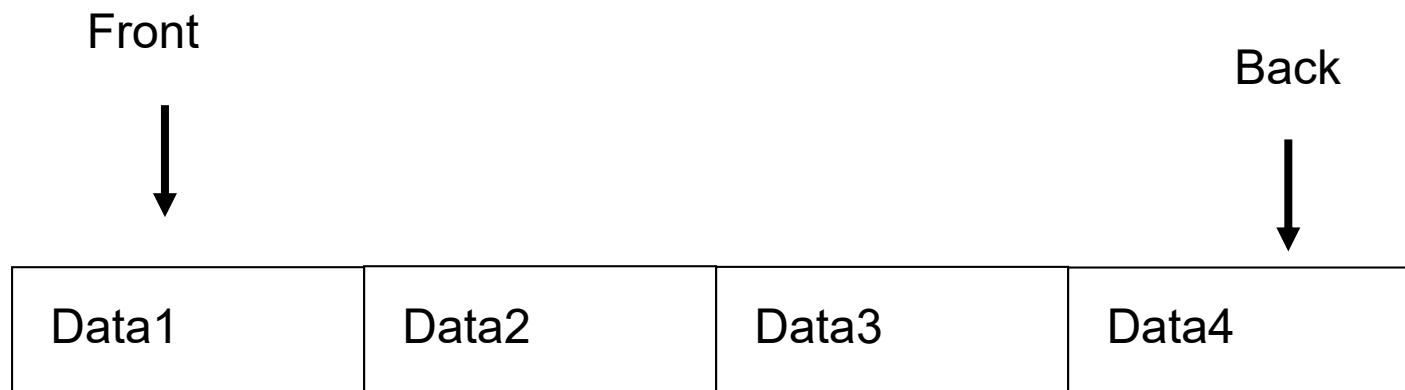
top

make empty



Queues

Collection with access only to the item that has been present the longest
first in first out
enqueue, dequeue



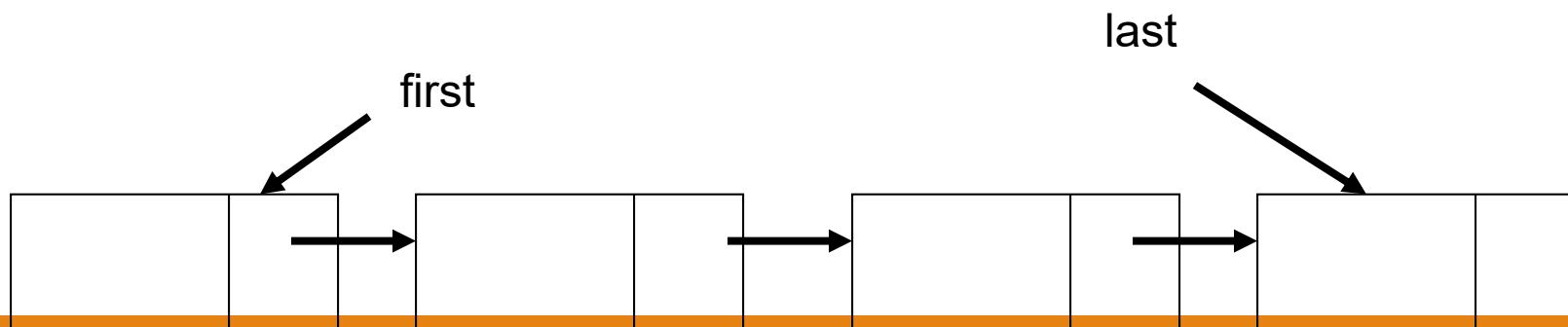
List

A **Flexible** structure, because can grow and shrink on demand.

Elements can be:

- Inserted
- Accessed
- Deleted

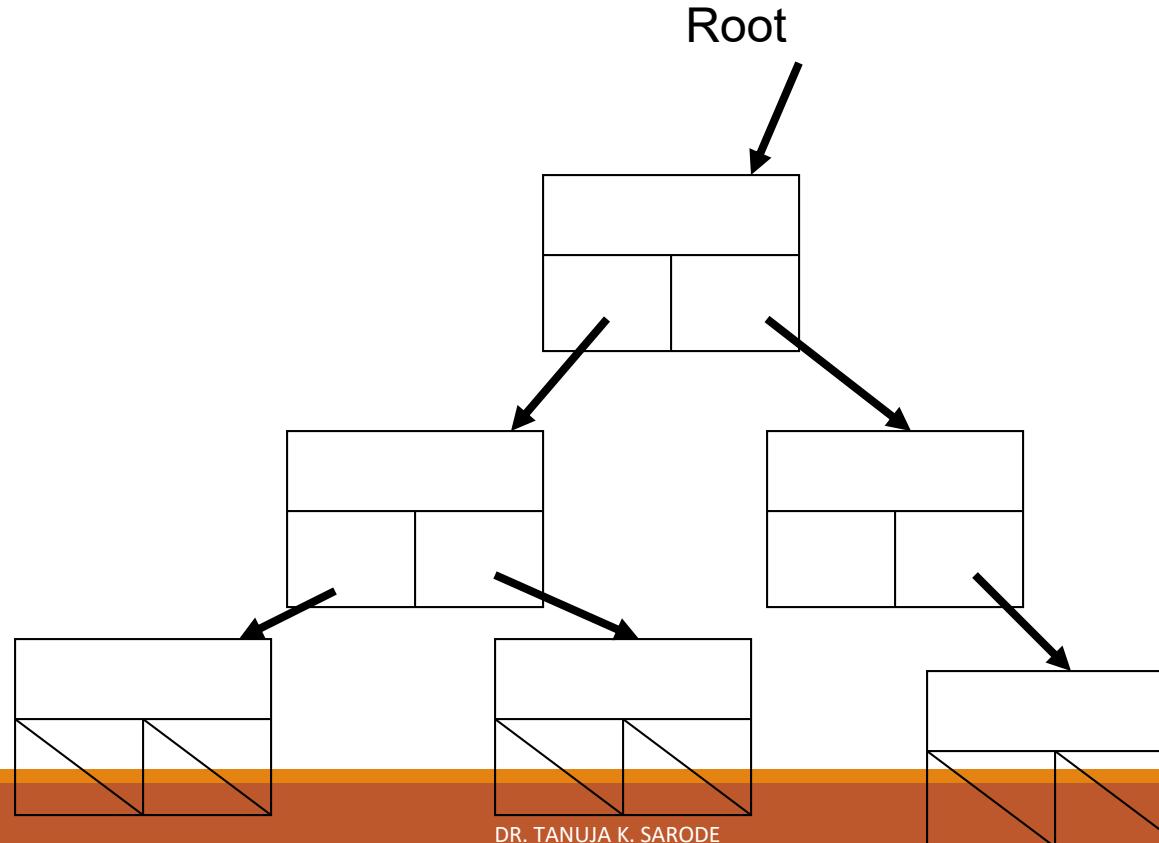
At **any** position



Tree

A **Tree** is a collection of elements called **nodes**.

One of the node is distinguished as a **root**, along with a relation (“parenthood”) that places a hierarchical structure on the nodes.



Graph

Graph is a mathematical non-linear data structure capable of representing many kind of physical structures.

Definition: A graph $G(V,E)$ is a set of vertices V and a set of edges E .

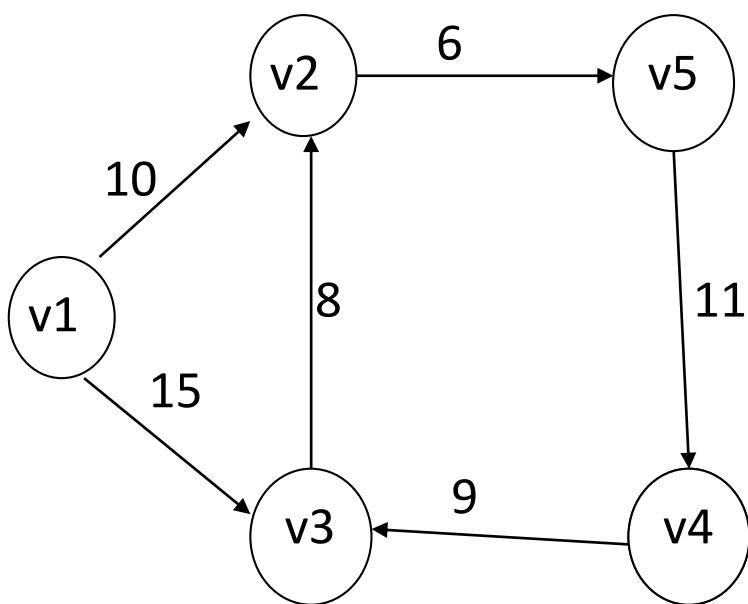
Graph

An edge connects a pair of vertices and many have weight such as length, cost and another measuring instrument for according the graph.

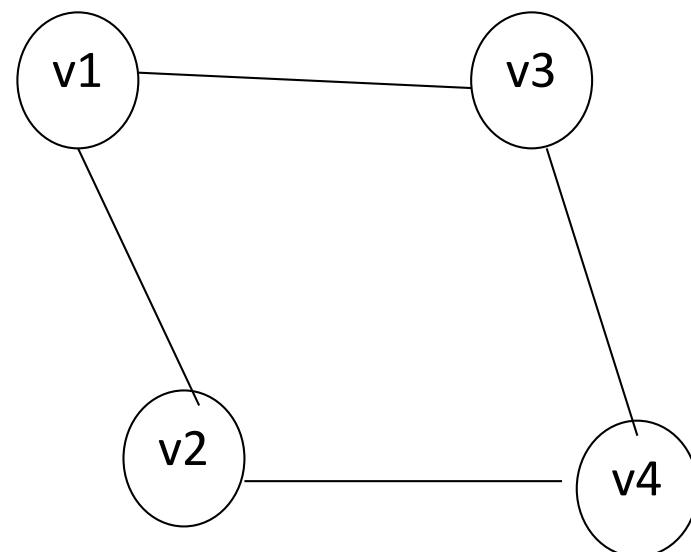
Vertices on the graph are shown as point or circles and edges are drawn as arcs or line segment.

Graph

Example of graph:



[a] Directed & Weighted Graph

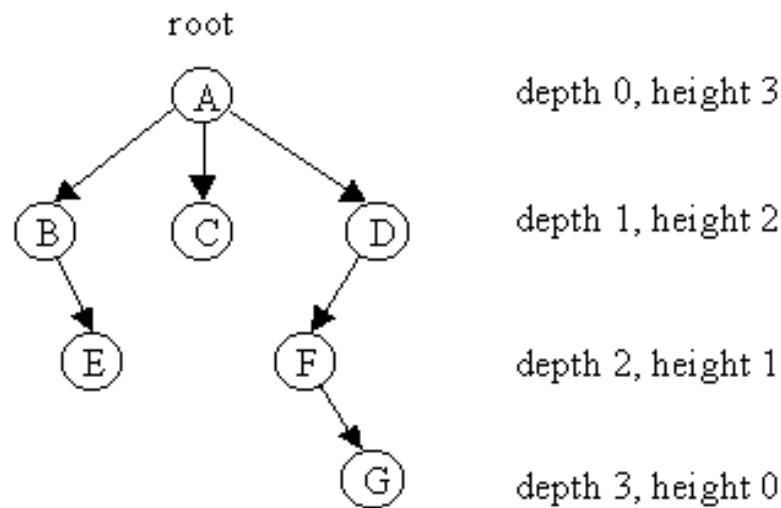


[b] Undirected Graph

Height and Depth of a tree

Depth or level : Number of edges in the path from root to that node

Height: Number of edges on the longest path from root to leaf



A tree of height 3

Graph

Types of Graphs:

- Directed graph
- Undirected graph
- Weighted graph
- Connected graph
- Non-connected graph

Syllabus

Uninformed Search: DFS, BFS, Uniform cost search, Depth Limited Search, Iterative Deepening.

Informed Search : Heuristic functions, Hill Climbing, Simulated Annealing, Best First Search, A*

Adversarial Search : Game Playing, Min-Max Search, Alpha Beta Pruning

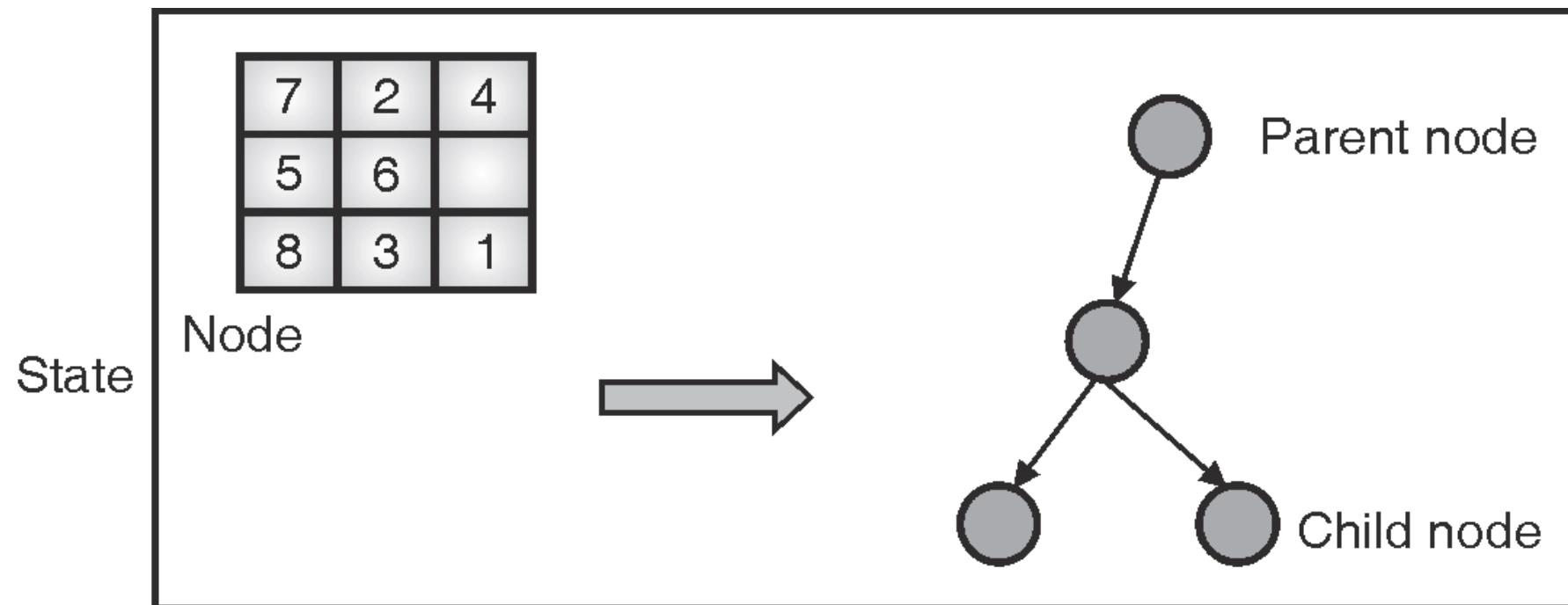
Introduction

- Search is an indivisible part of intelligence.
- An intelligent agent is the one who can search and select the most appropriate action in the given situation, among the available set of actions.
- When we play any game like chess, cards, tic-tac-toe, etc. We know that we have multiple options for next move, but the intelligent one who searches for the correct move will definitely win the game.

Measuring Performance of Problem Solving Algorithm

- 1. Completeness** : If the algorithm is able to produce the solution if one exists then it satisfies completeness criteria.
- 2. Optimality** : If the solution produced is the minimum cost solution, the algorithm is said to be optimal.
- 3. Time complexity** : It depends on the time taken to generate the solution. It is the number of nodes generated during the search.
- 4. Space complexity** : Memory required to store the generated nodes while performing the search.

Node Representation in Search Tree



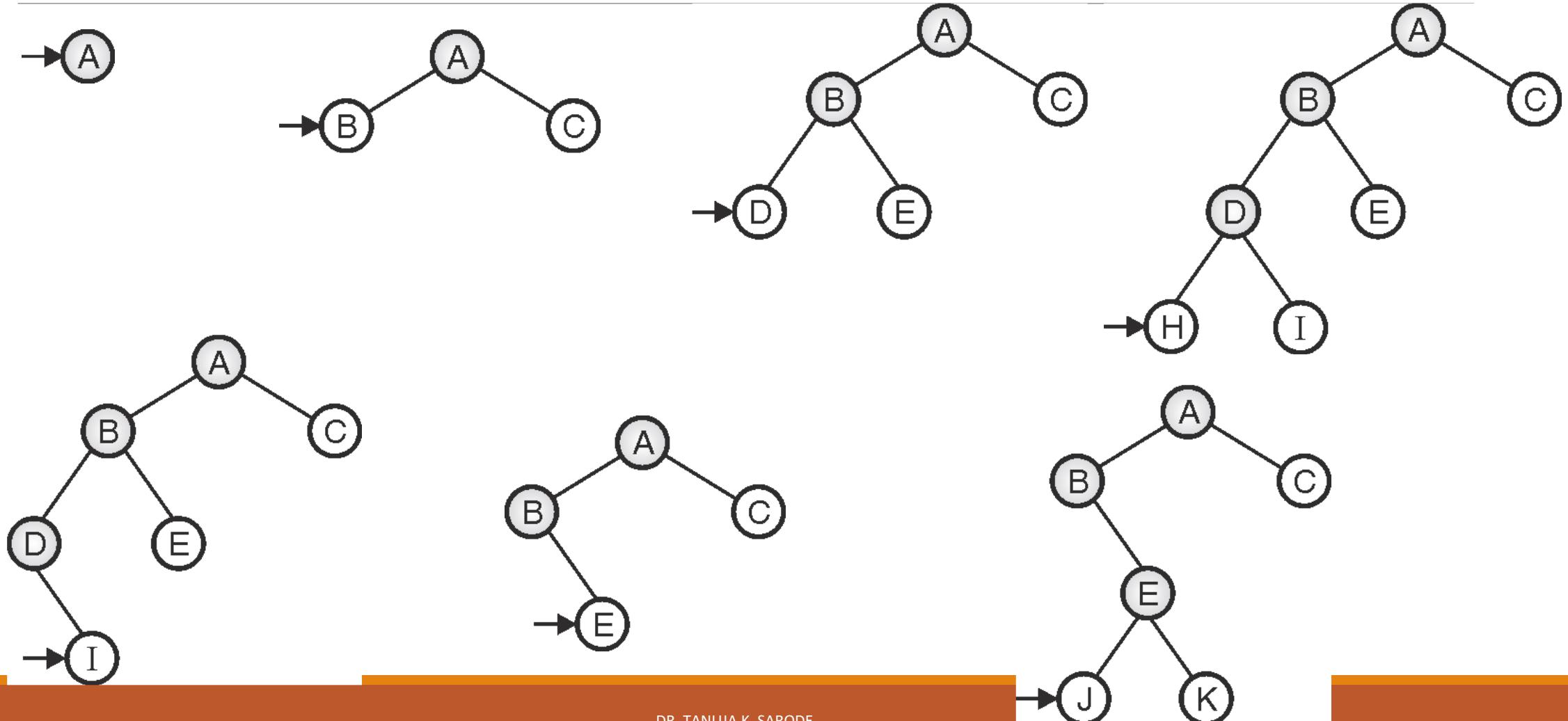
Uninformed Search

- The term “uninformed” means they have only information about what is the start state and the end state along with the problem definition.
- These techniques can generate successor states and can distinguish a goal state from a non-goal state.
- All these search techniques are distinguished by the order in which nodes are expanded.
- The uninformed search techniques also called as “blind search”.

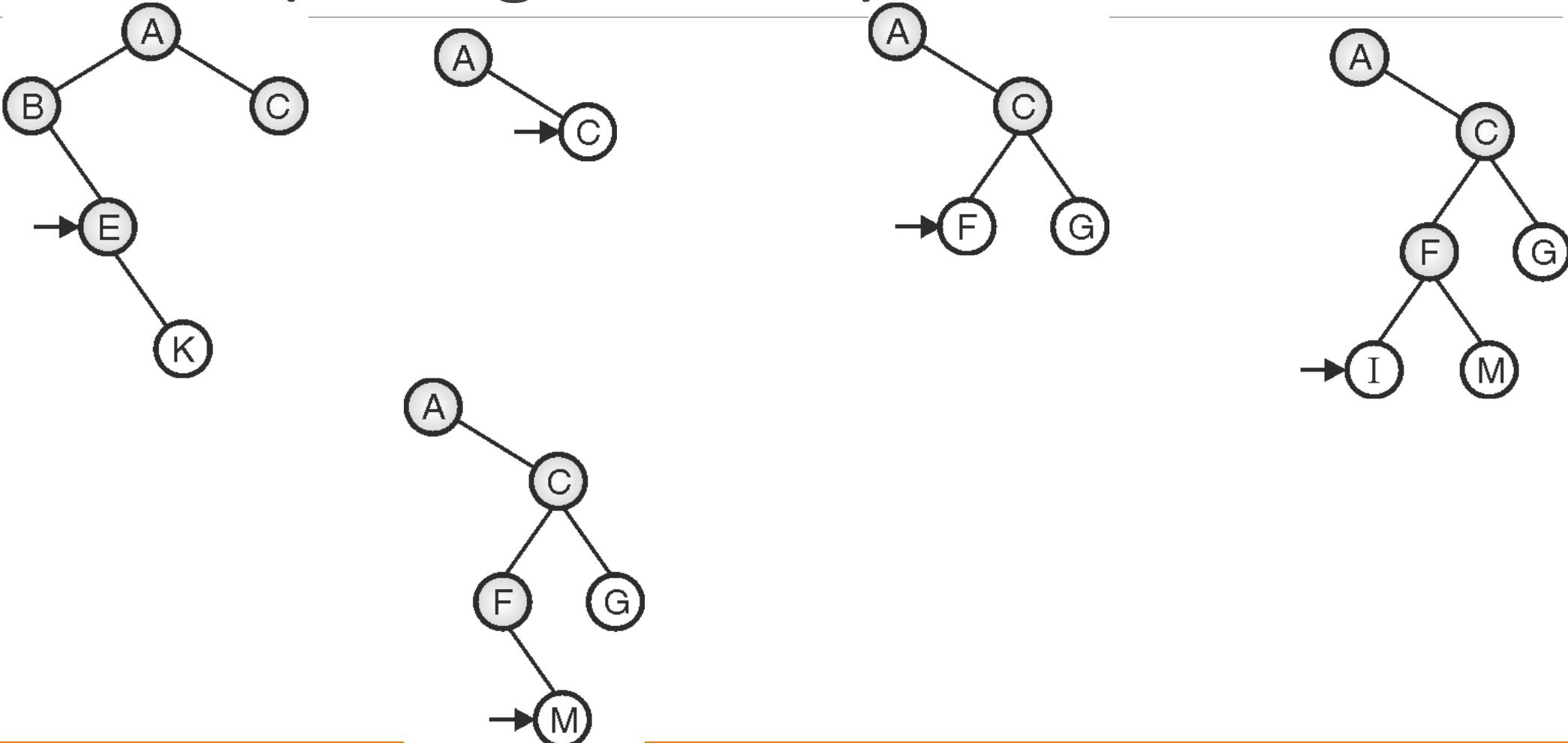
Depth First Search (DFS)

- In depth-first search, the search tree is expanded depth wise; i.e. the deepest node in the current branch of the search tree is expanded. As the leaf node is reached, the search backtracks to previous node.
- The explored nodes are shown in light gray. Explored nodes with no descendants in the fringe are removed from memory.

Process



Process(M is goal node)



Algorithm(uses a LIFO i.e. stack)

Non recursive implementation of DFS

1. Push the root node on a stack
2. while (stack is not empty)
 - a) pop a node from the stack;
 - (i) if node is a goal node then return success;
 - (ii) push all children of node onto the stack;
3. return failure

Recursive implementation of DFS

DFS(c) :

1. If node is a goal, return success;
2. for each child c of node
 - a) if $\text{DFS}(c)$ is successful,
 - (i) return success
3. return failure;

Performance Evaluation

- **Completeness** : Complete, if $m(\text{depth})$ is finite.
- **Optimality** : No, as it cannot guarantee the shallowest solution.
- **Time Complexity** : A depth first search, may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space.
- **Space Complexity** : For a search tree with branching factor b and maximum depth m , depth first search requires storage of only $O(b^m)$ nodes, as at a time only the branch, which is getting explored, will reside in memory.

n¹:

Define a stack of size total no: of vertices in graph.

p²:

Select any vertex as starting point for traversal.
Visit that vertex and push it on to the stack

p³:

Visit any one of the adjacent vertex of the vertex
which is at top of the stack, which is not visited,
& push it on to the stack.

-
- Step 4: Repeat Step 3, until there are no new vertex to be visit from the vertex on top of the stack.
 - Step 5: When there is no new vertex to be visit then use backtracking and pop one vertex from the stack.
 - Step 6: Repeat Step 3, 4, 5 until stack becomes empty.
 - Step 7: When stack becomes empty, then produce final spanning tree by removing unused edges from the graph.



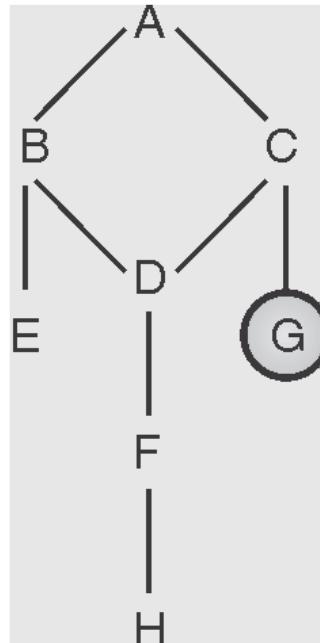
Detailed Solution Step wise

Refer PDF

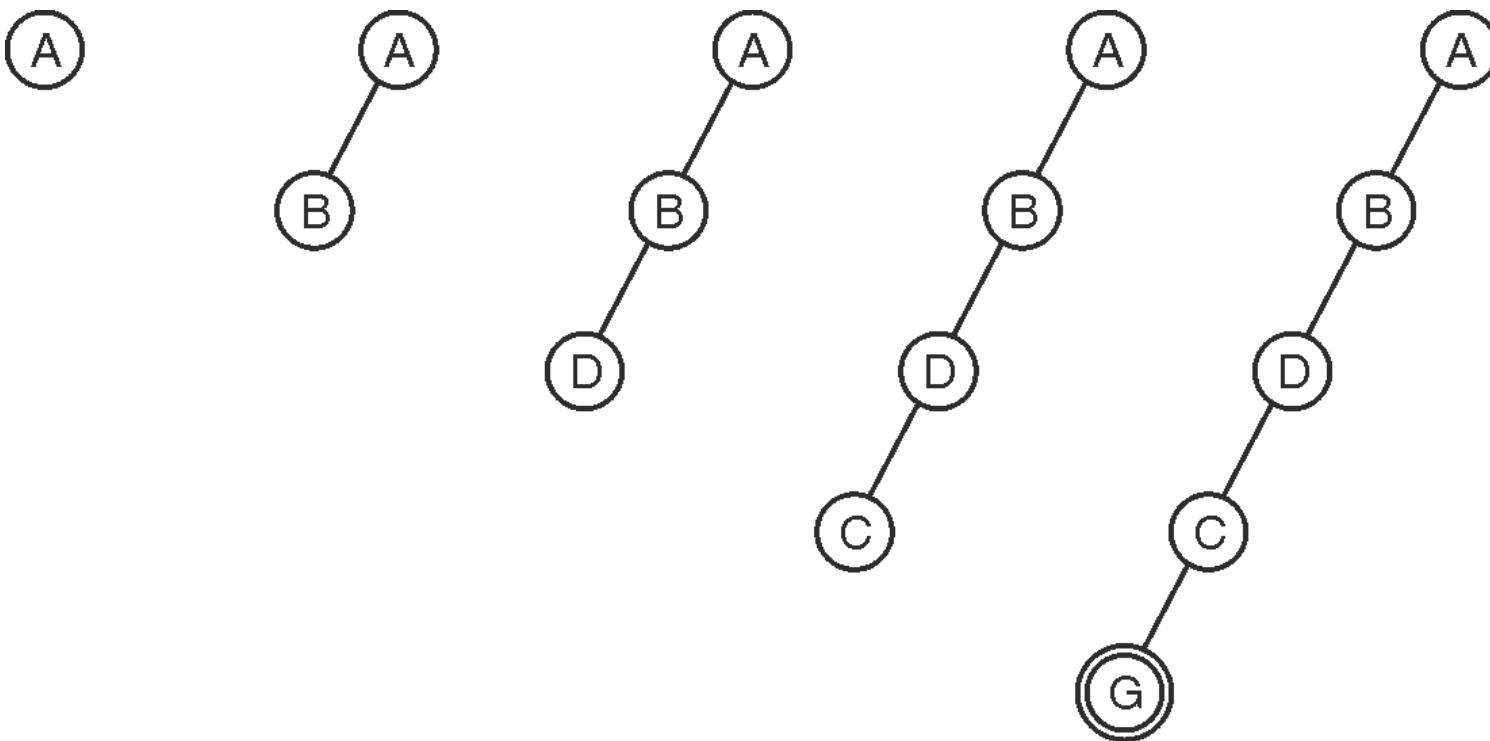
MU - May 16, 10 Marks

Consider following graph.

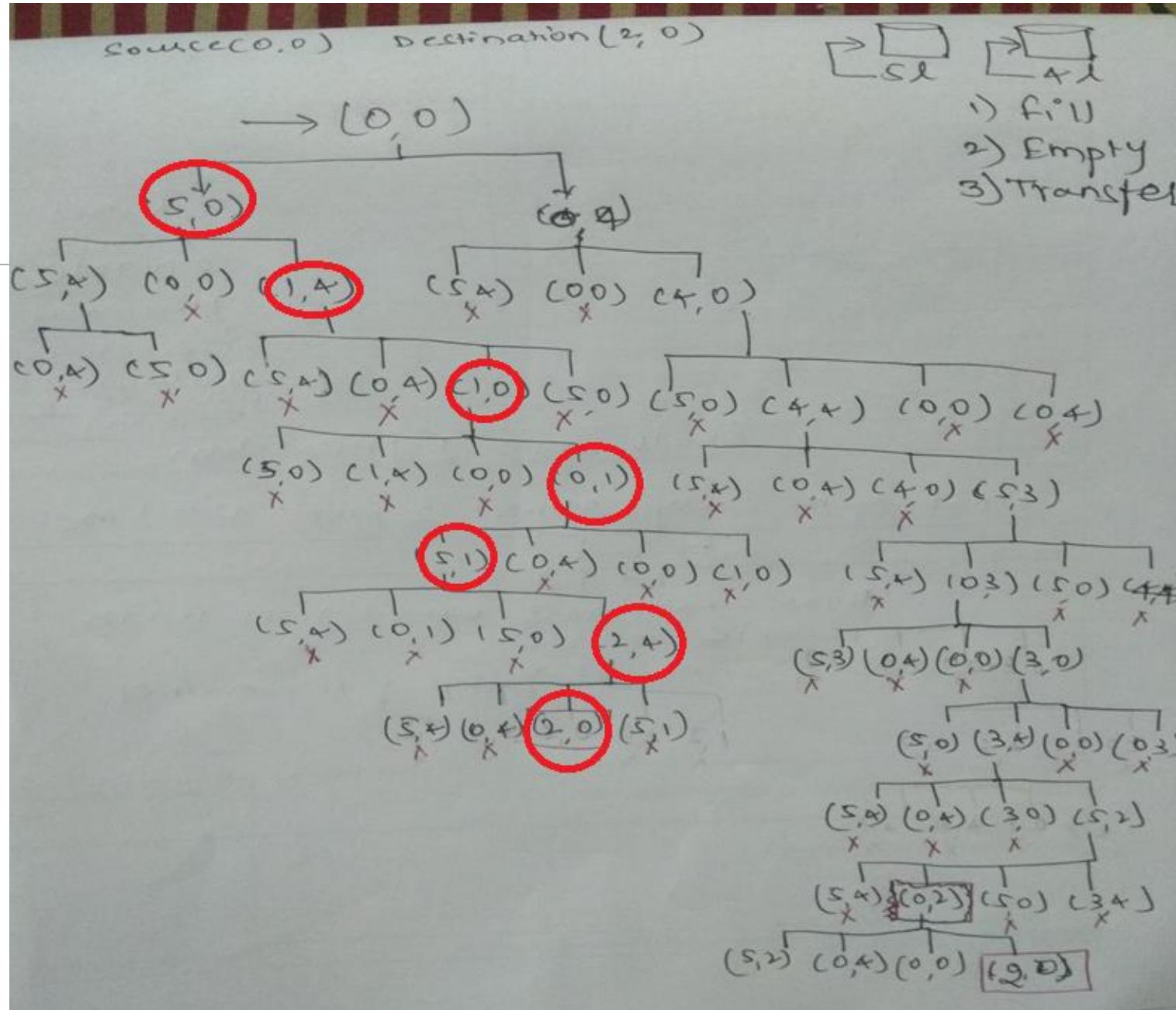
Starting from state A execute DFS. The goal node is G. Show the order in which the nodes are expanded. Assume that the alphabetically smaller node is expanded first to break ties.



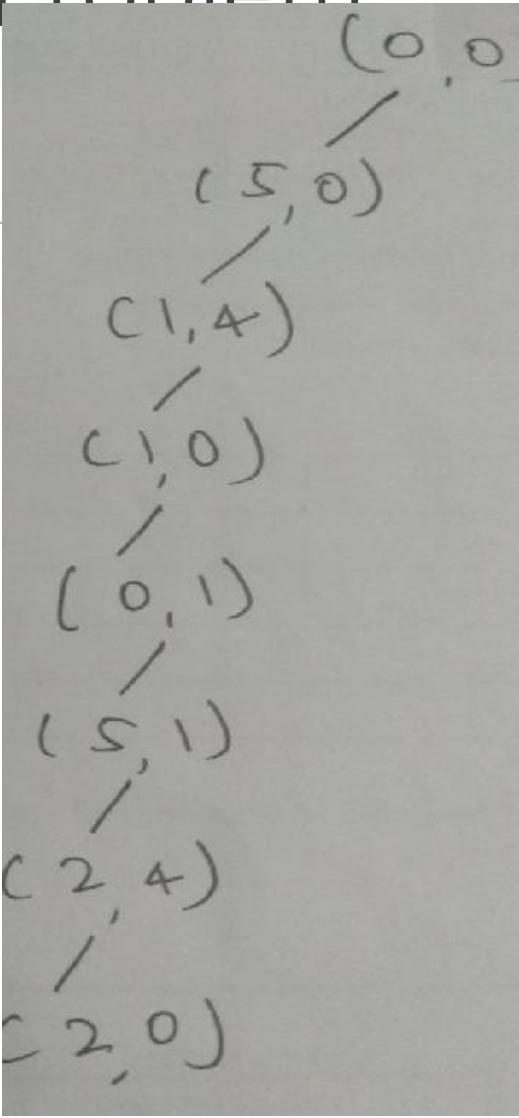
Solution(Write Detailed steps)

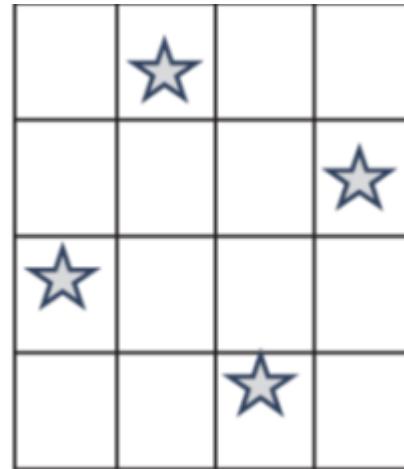
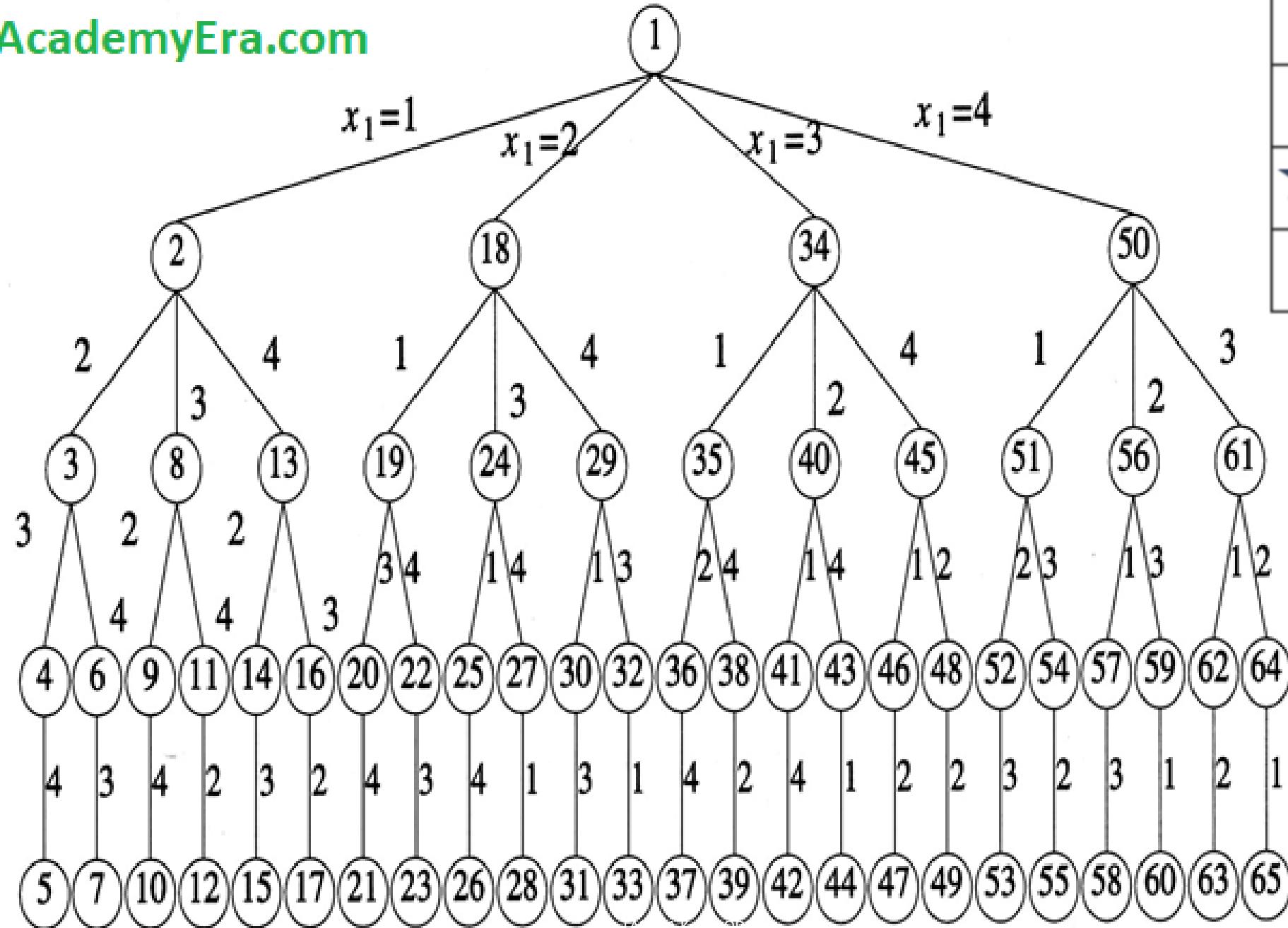


Apply DSF to solve Water Jug Problem

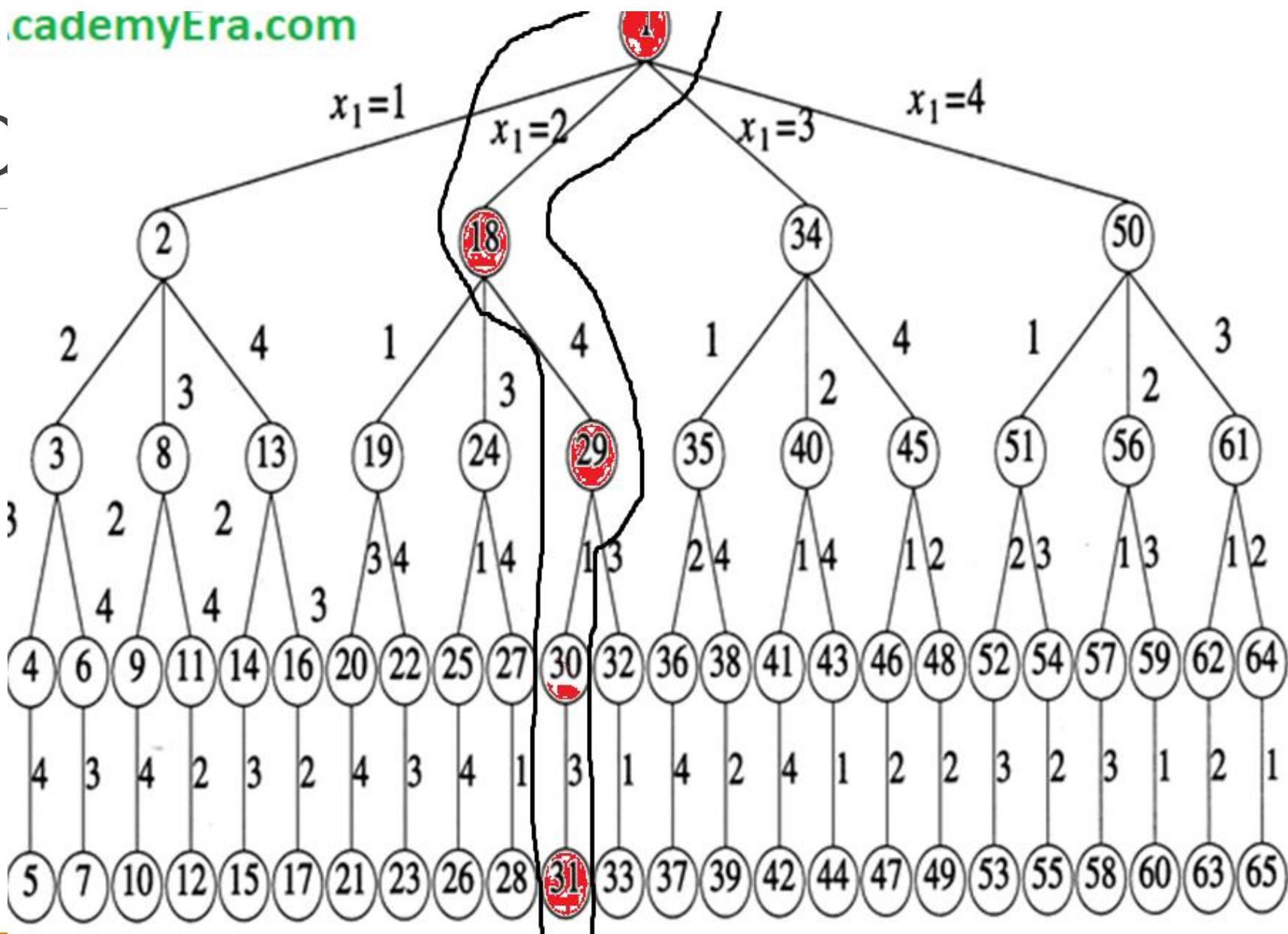
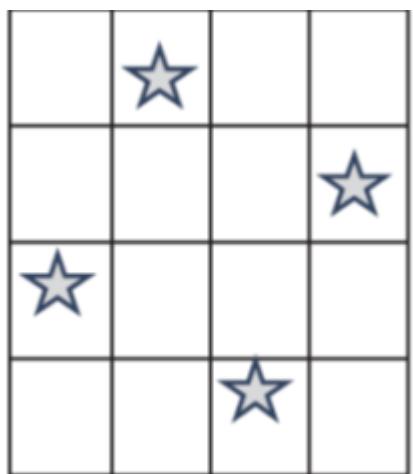


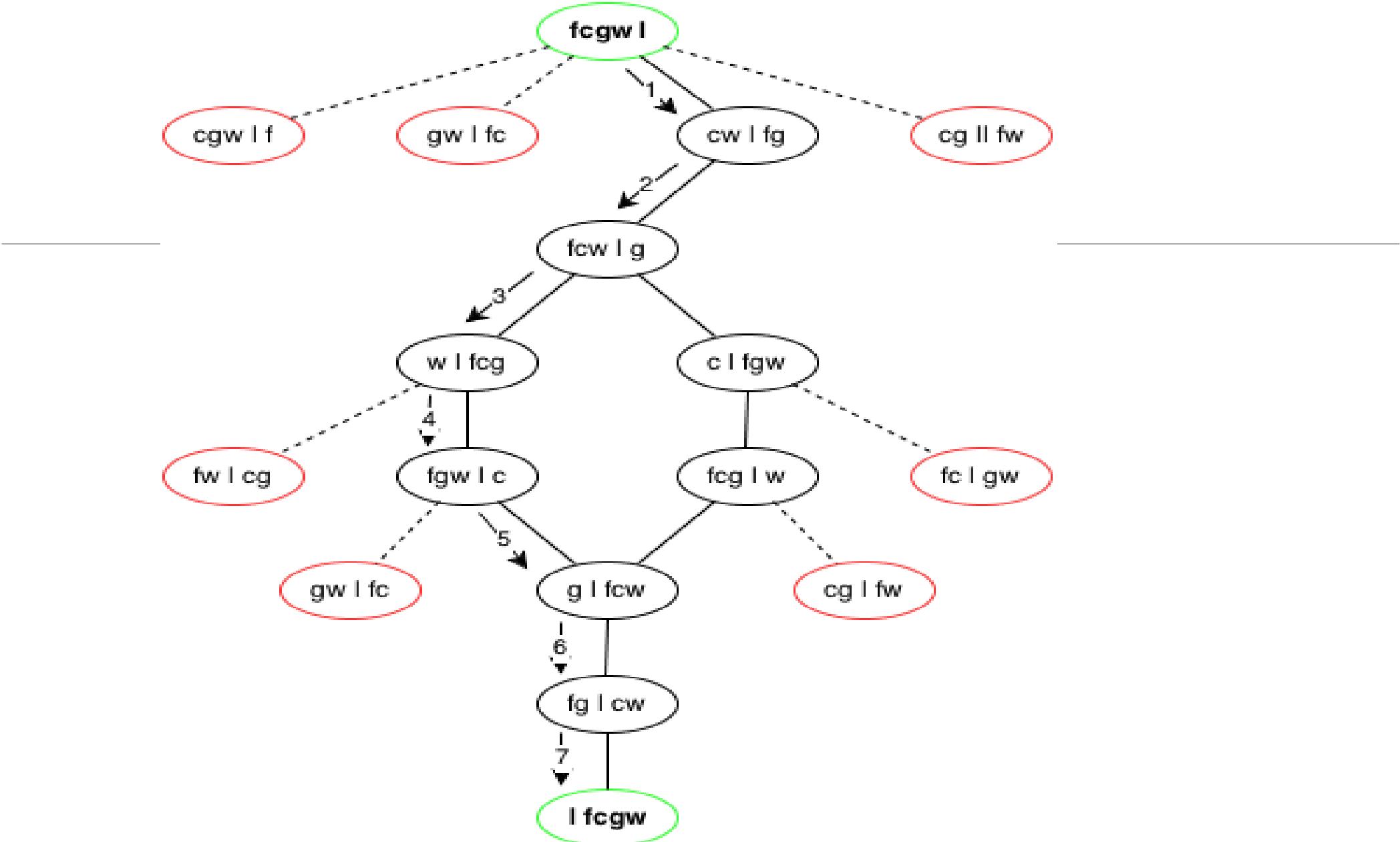
DFS on water Jug Problem





DFS on N C



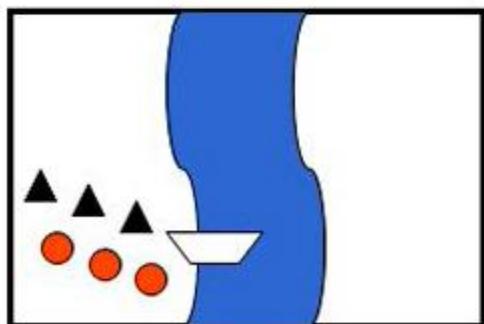


Simple Depth First Search on State Space Graph

Missionaries and Cannibals: Initial State and Actions

- initial state:

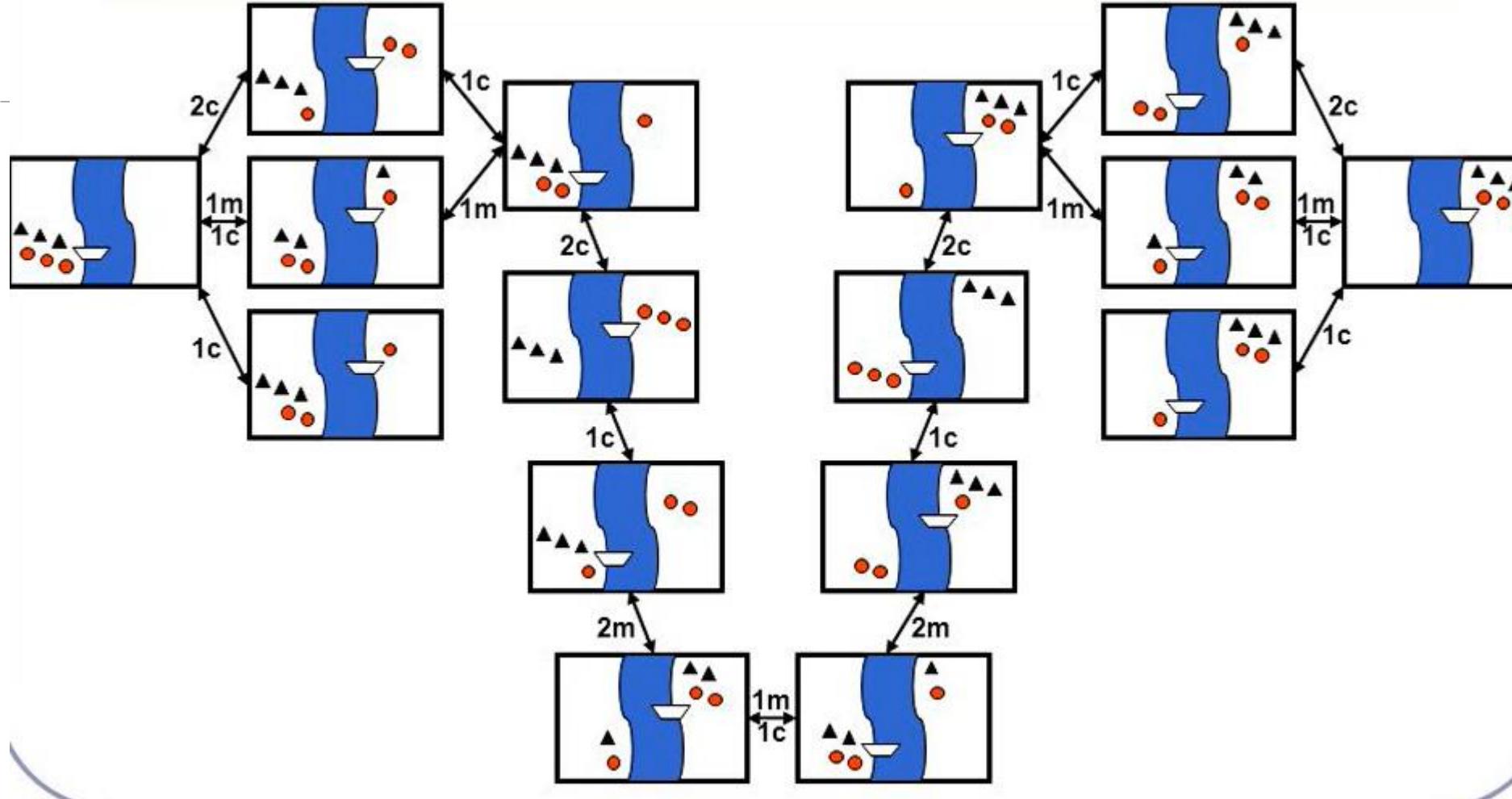
- all missionaries, all cannibals, and the boat are on the left bank



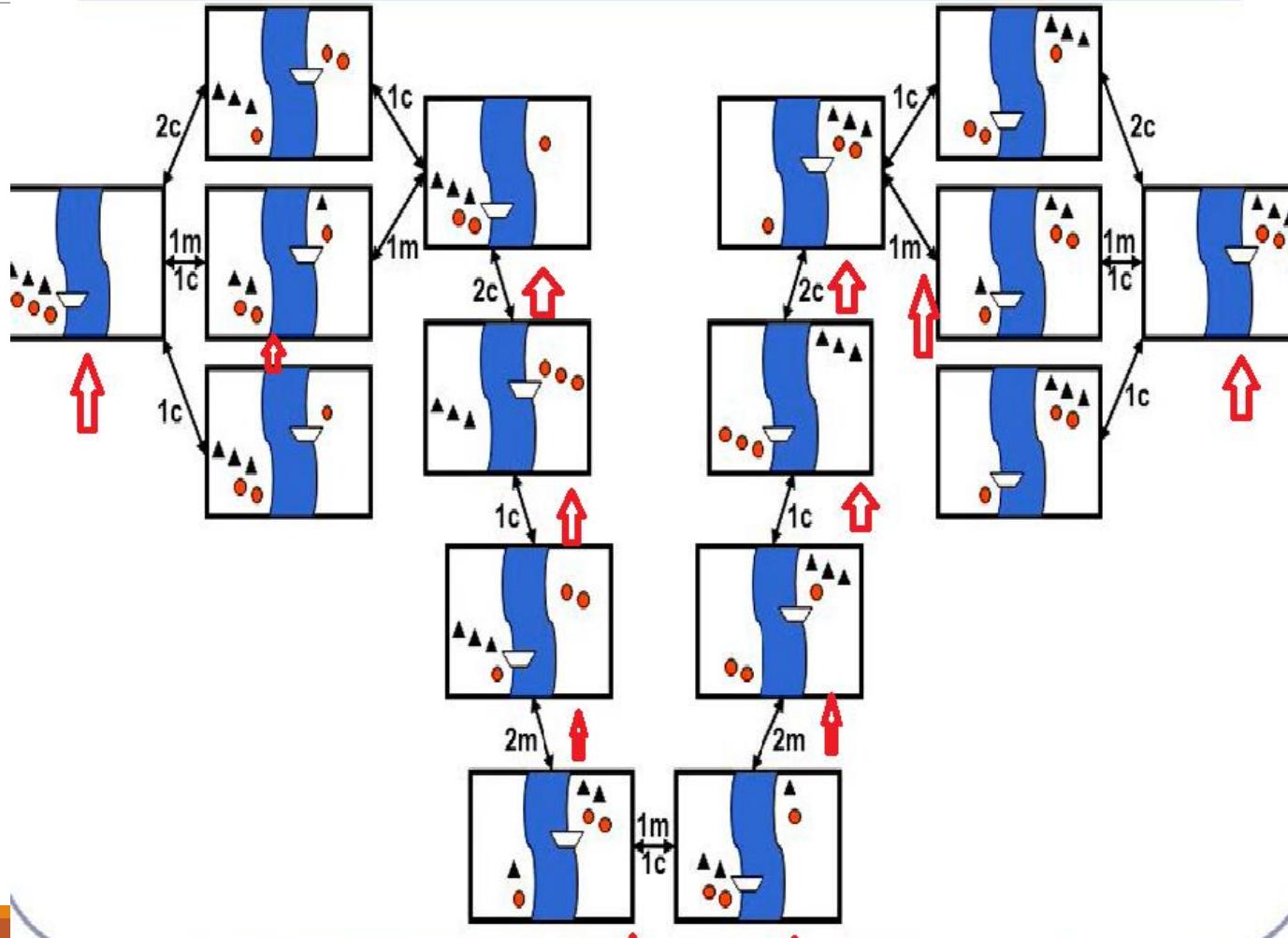
- 5 possible actions:

- one missionary crossing
- one cannibal crossing
- two missionaries crossing
- two cannibals crossing
- one missionary and one cannibal crossing

State Space Representation



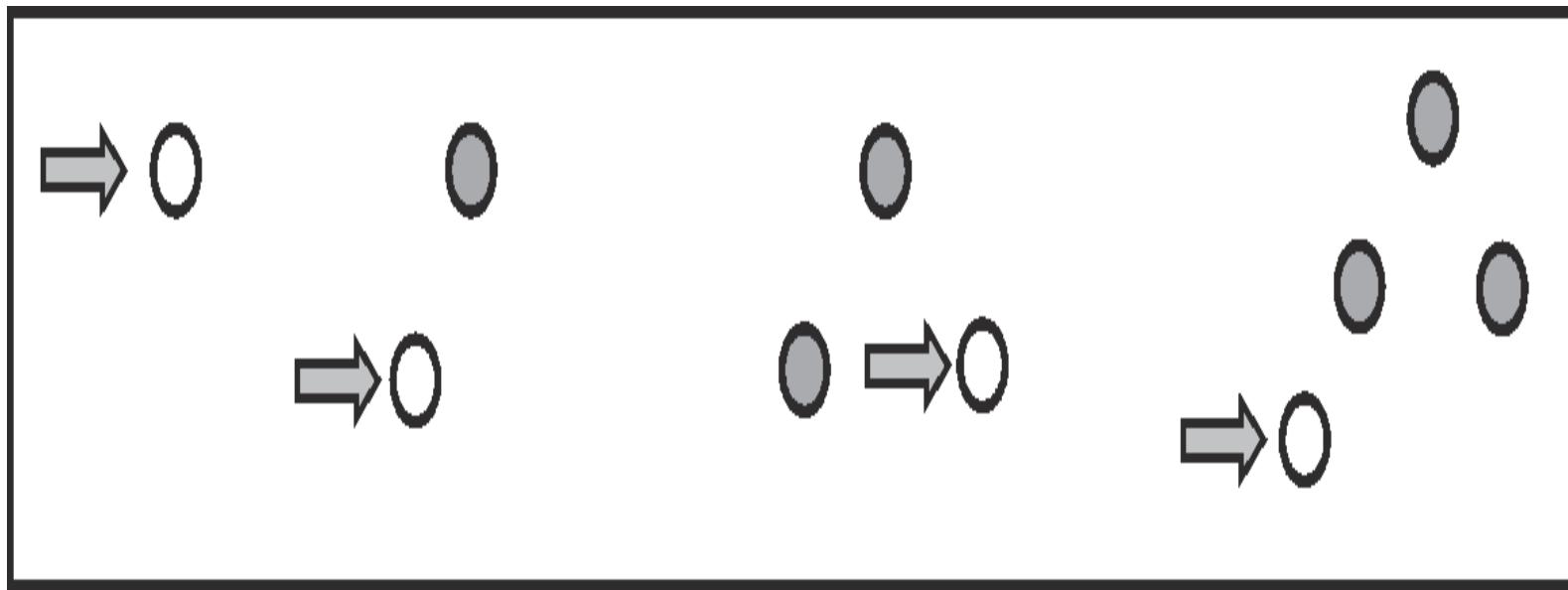
DFS on Missionary and Cannibals



Breadth First Search (BFS)

- As the name suggests, in breadth-first search technique, the tree is expanded breadth wise.
- The root node is expanded first, then all the successors of the root node are expanded, then their successors, and so on.
- In turn, all the nodes at a particular depth in the search tree are expanded first and then the search will proceed for the next level node expansion.

Process



Working of BFS on binary tree

Implementation

- In BFS we use a FIFO queue for the fringe. Because of which the newly inserted nodes in the fringe will automatically be placed after their parents.
- Thus, the children nodes, which are deeper than their parents, go to the back of the queue, and old nodes, which are shallower, get expanded first. Following is the algorithm for the same.

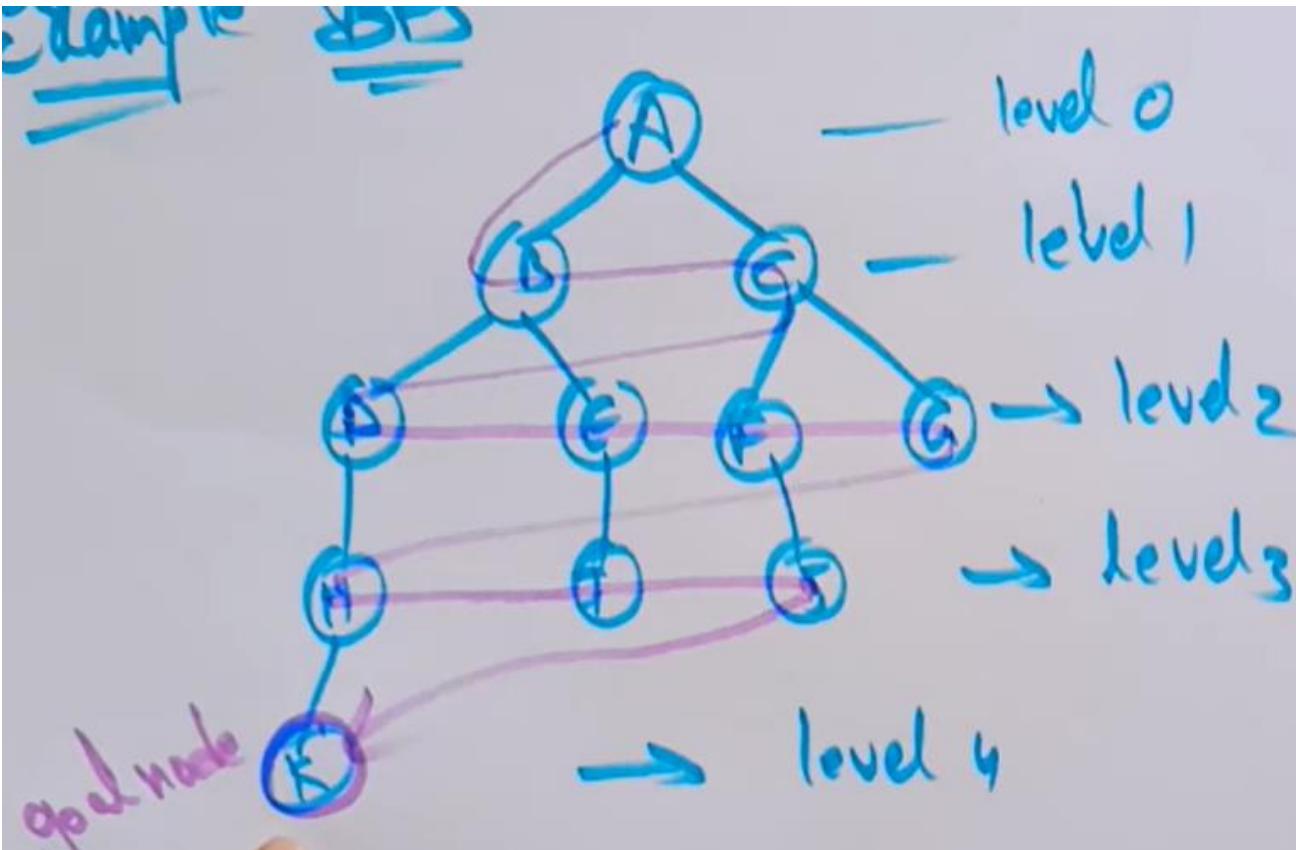
2.5.4 Algorithm

1. Put the root node on a queue
2. while (queue is not empty)
 - a) remove a node from the queue
 - (i) if (node is a goal node) return success;
 - (ii) put all children of node onto the queue;
 3. return failure;

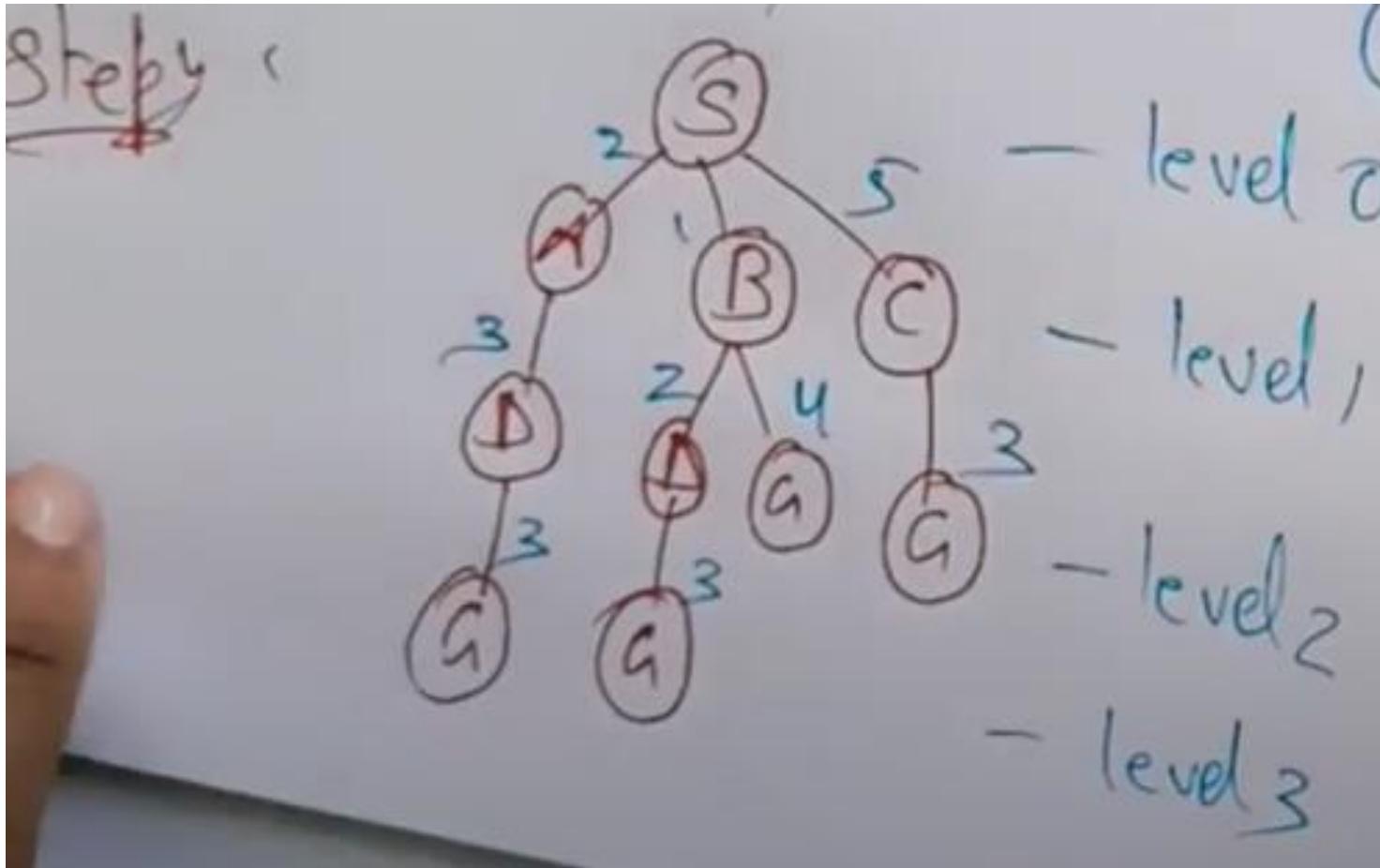
Performance Evaluation

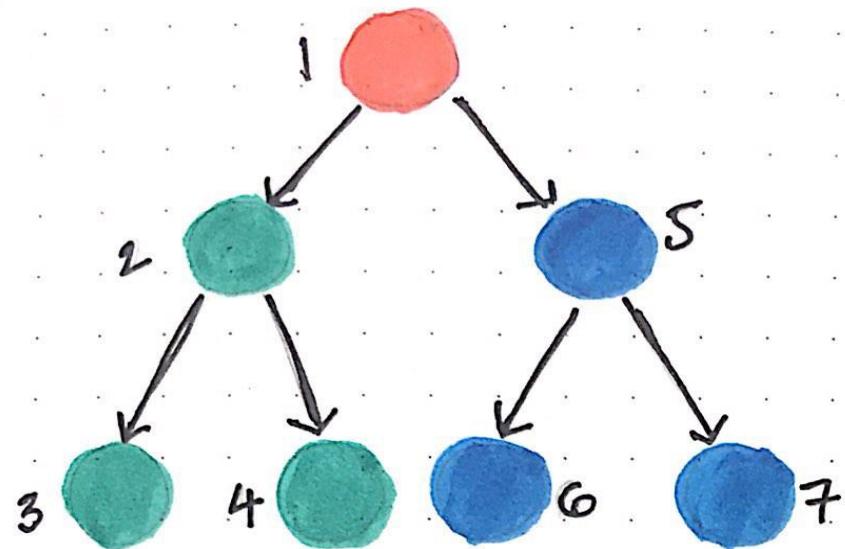
- **Completeness** : It is complete, provided the shallowest goal node is at some finite depth.
- **Optimality** : It is optimal, as it finds the shallowest solution.
- **Time complexity** : $O(b^d)$, number of nodes in the fringe.
- **Space complexity** : $O(b^d)$, total number of nodes explored.

BFS



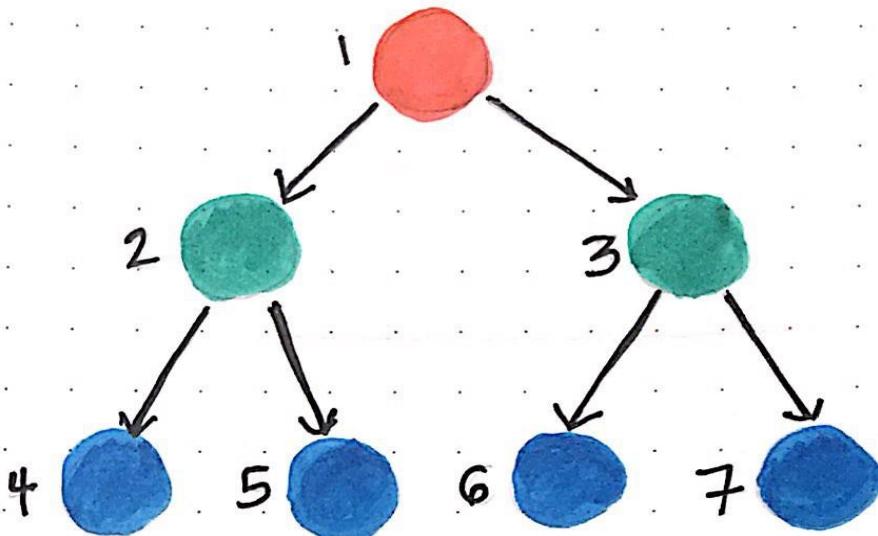
BFS(Path is SBG or SCG)





Depth-first search

- Traverse through left subtree(s) first, then traverse through the right subtree(s).



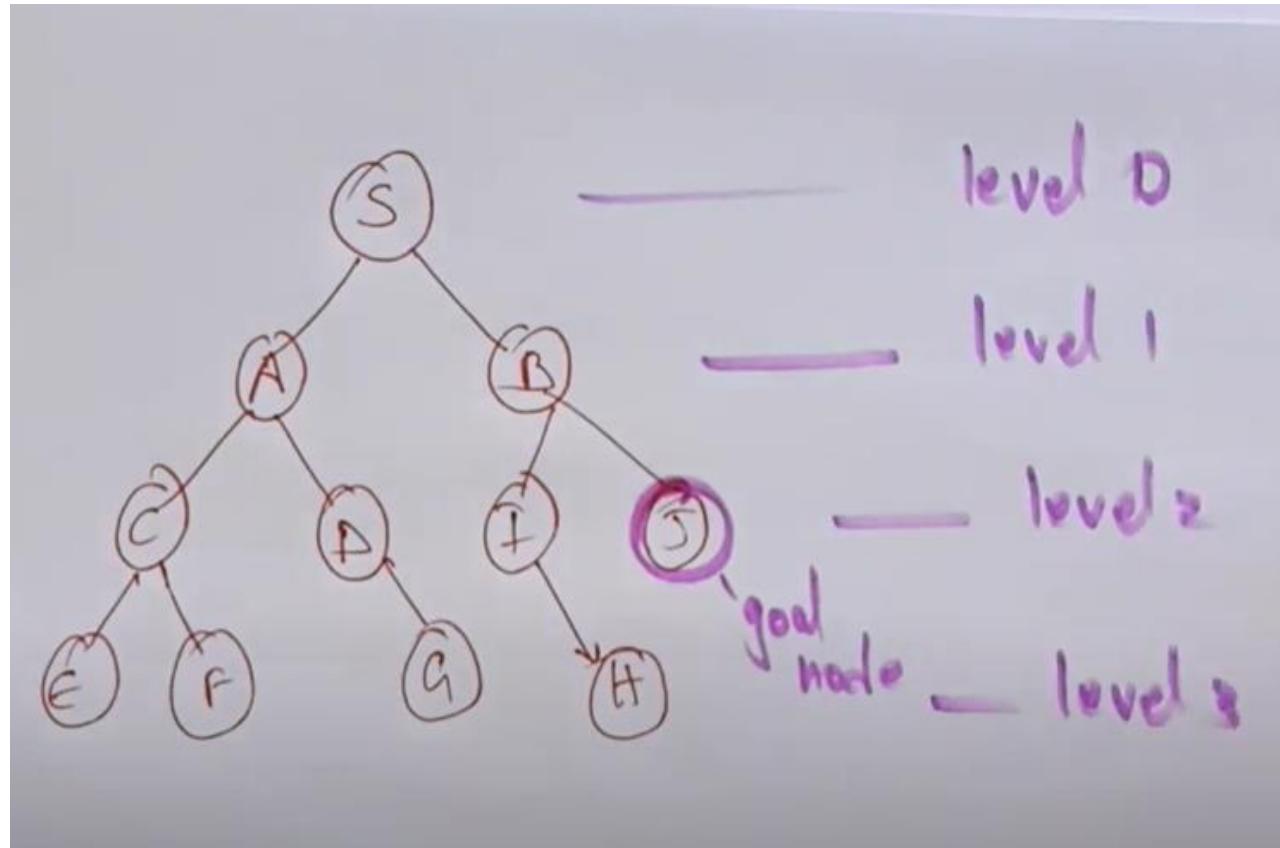
Breadth-first search

- Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

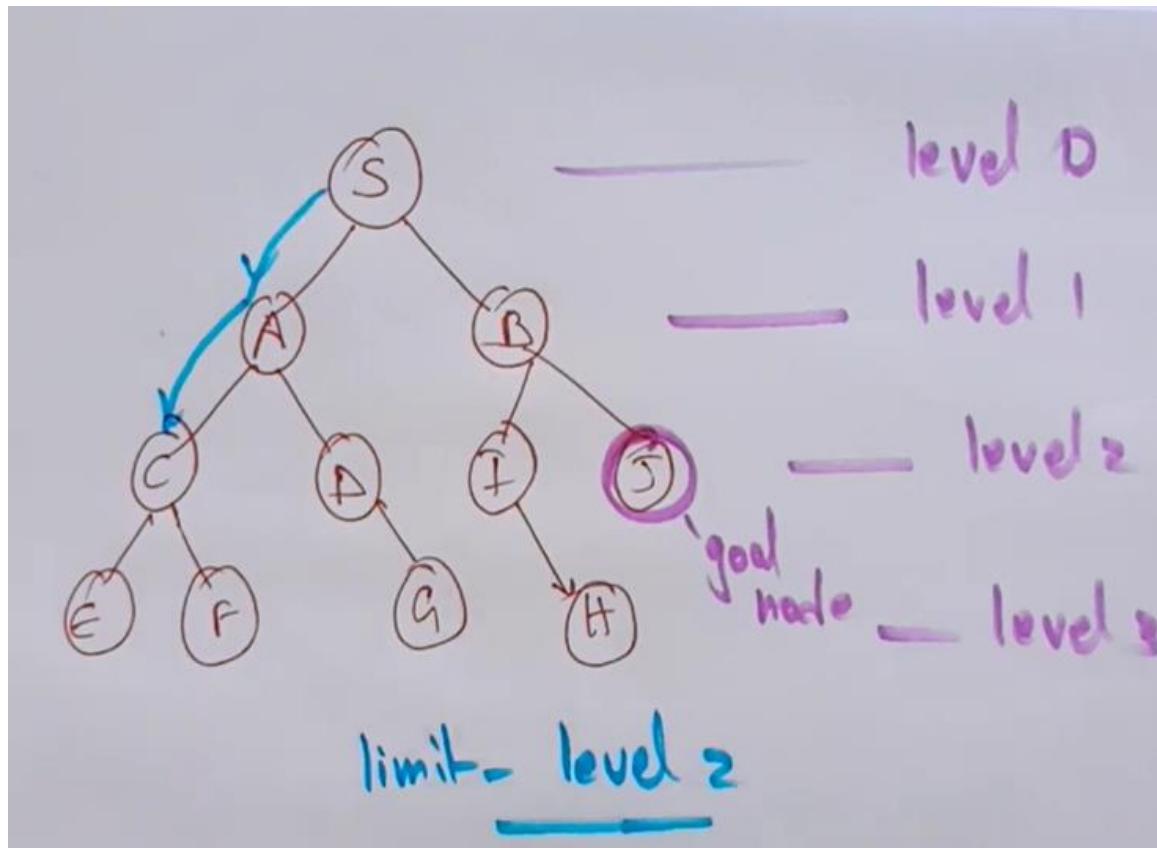
Depth Limited Search (DLS)

- In order to avoid the infinite loop condition arising in DFS, in depth limited search technique, depth-first search is carried out with a **predetermined depth limit**.
- The nodes with the specified depth limit are treated as if they don't have any successors. The depth limit solves the infinite-path problem.
- But as the search is carried out only till certain depth in the search tree, it **introduces problem of incompleteness**.
- Depth-first search can be viewed as a special case of depth-limited search with depth limit equal to the depth of the tree.

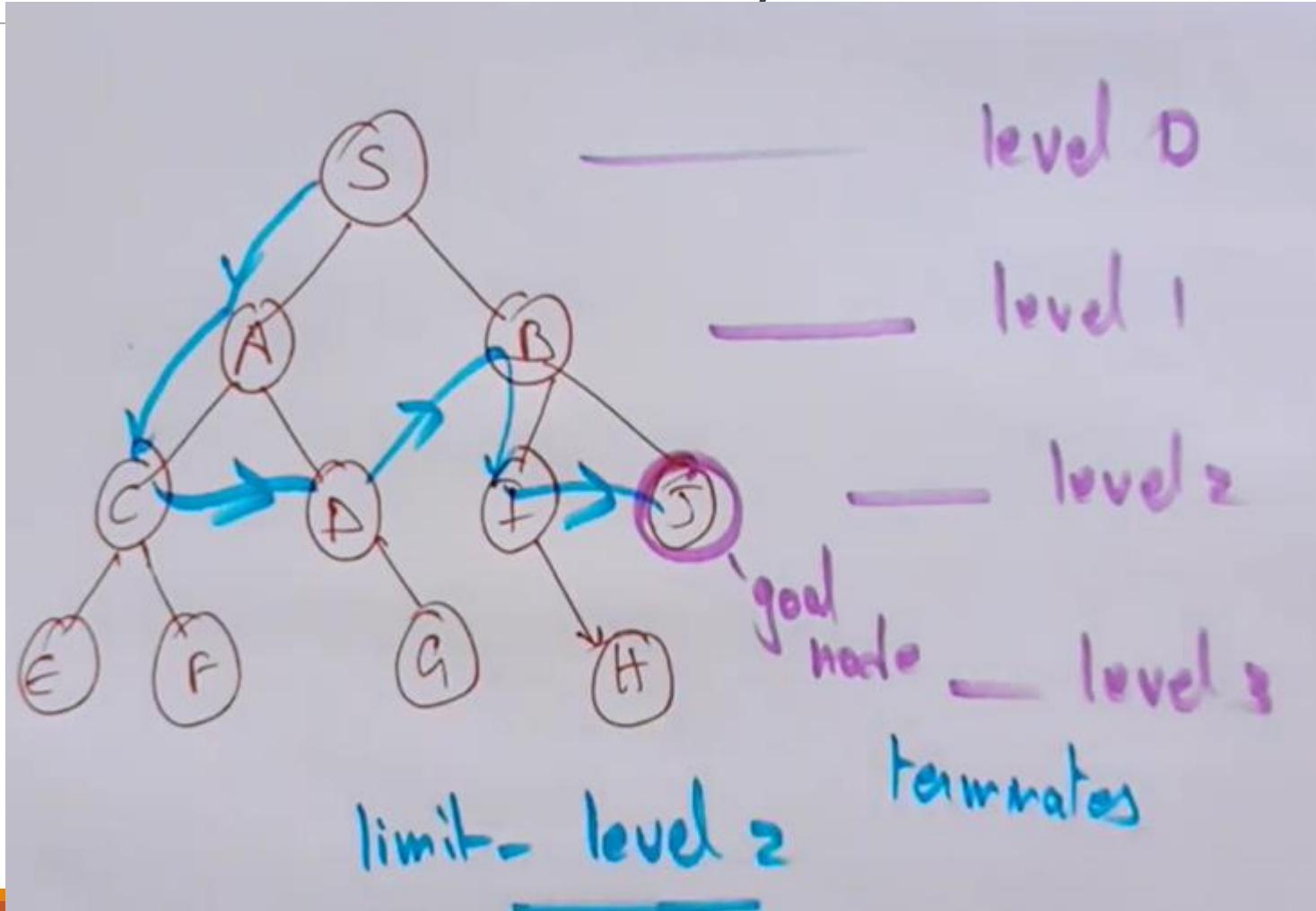
DLS(With limit level 2 and Goal is G)



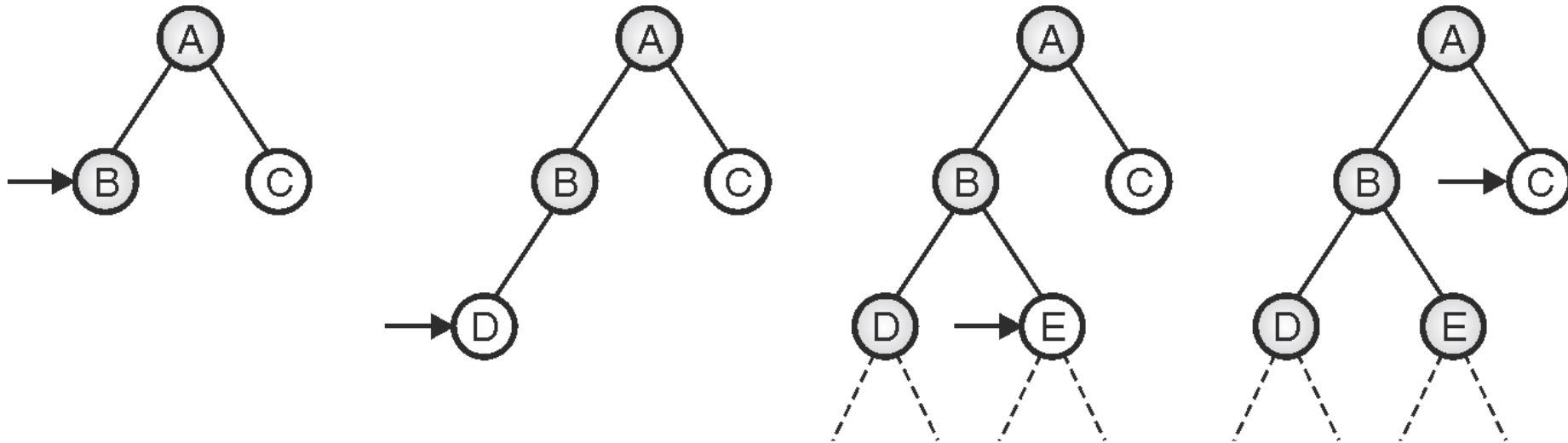
DLS(With limit level 2)



DLS(With limit level 2)



Process (DLS working with depth limit)



If depth limit is fixed to 2, DLS carries out depth first search till second level in the search tree.

Pseudo Code

```
booleanDLS (Node node, int limit, int depth)
{
    if (depth > limit) return failure;
    if (node is a goal node) return success;
    for each child of node
    {
        if (DLS(child, limit, depth + 1))
            return success;
    }
    return failure;
}
```

Performance Evaluation

Completeness : Its incomplete if shallowest goal is beyond the depth limit.

Optimality : Non optimal, as the depth chosen can be greater than d .

Time complexity : Same as DFS, $O(b^l)$, where l is the specified depth limit.

Space complexity : Same as DFS, $O(b^l)$, where l is the specified depth limit.

Iterative Deepening DFS (IDDFS)

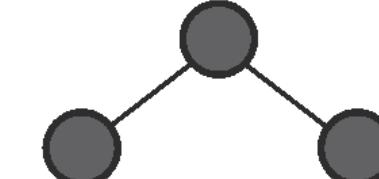
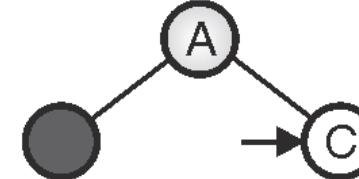
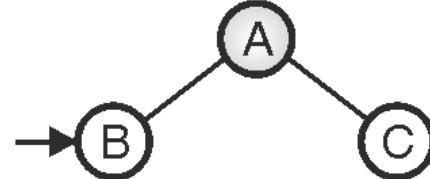
- Iterative deepening depth first search is a combination of BFS and DFS. In IDDFS search happens depth wise but, at a time the depth limit will be incremented by one. Hence iteratively it deepens down in the search tree.
- It eventually turns out to be the breadth-first search as it explores a complete layer of new nodes at each iteration before going on to the next layer.
- It does this by gradually increasing the depth limit-first 0, then 1, then 2, and so on-until a goal is found; and thus guarantees the optimal solution. Iterative deepening combines the benefits of depth-first and breadth-first search.

Flow of Algorithm

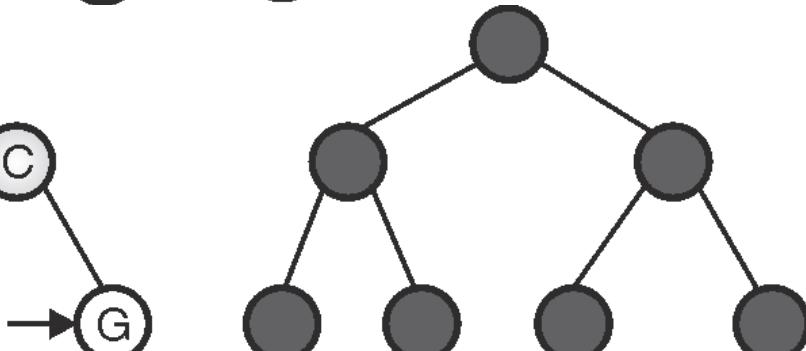
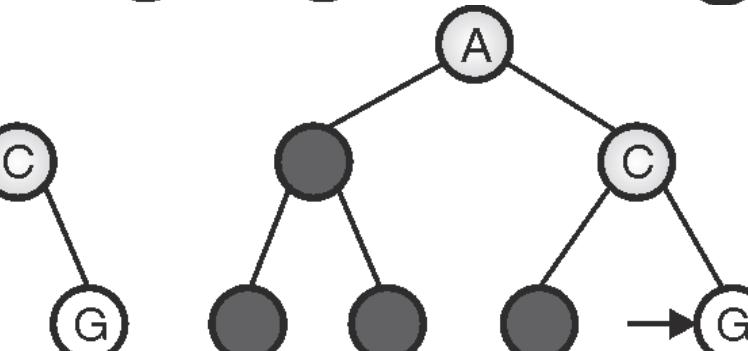
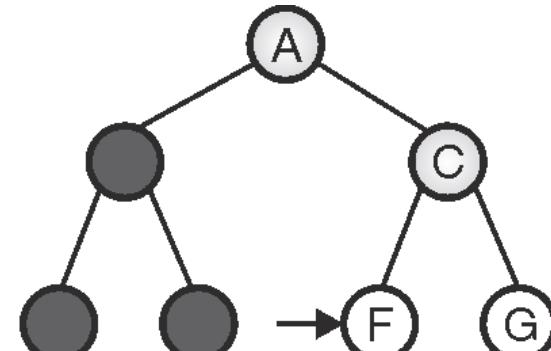
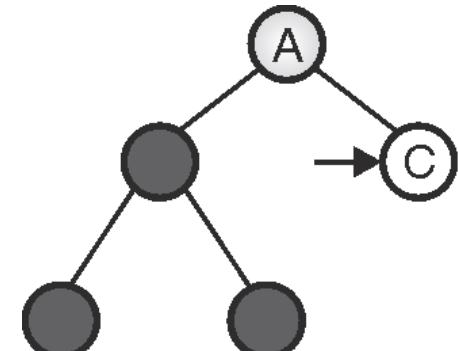
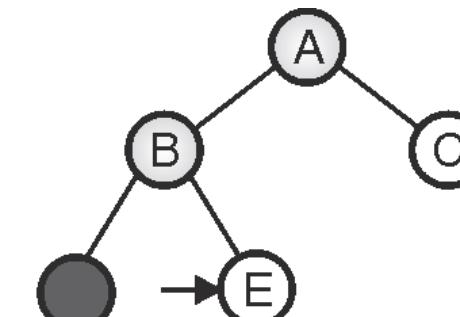
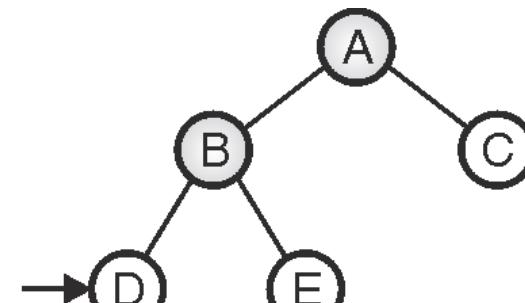
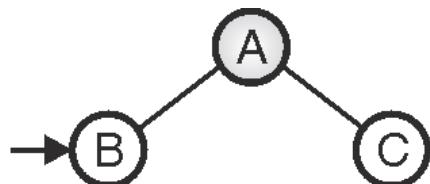
Limit = 0 → A



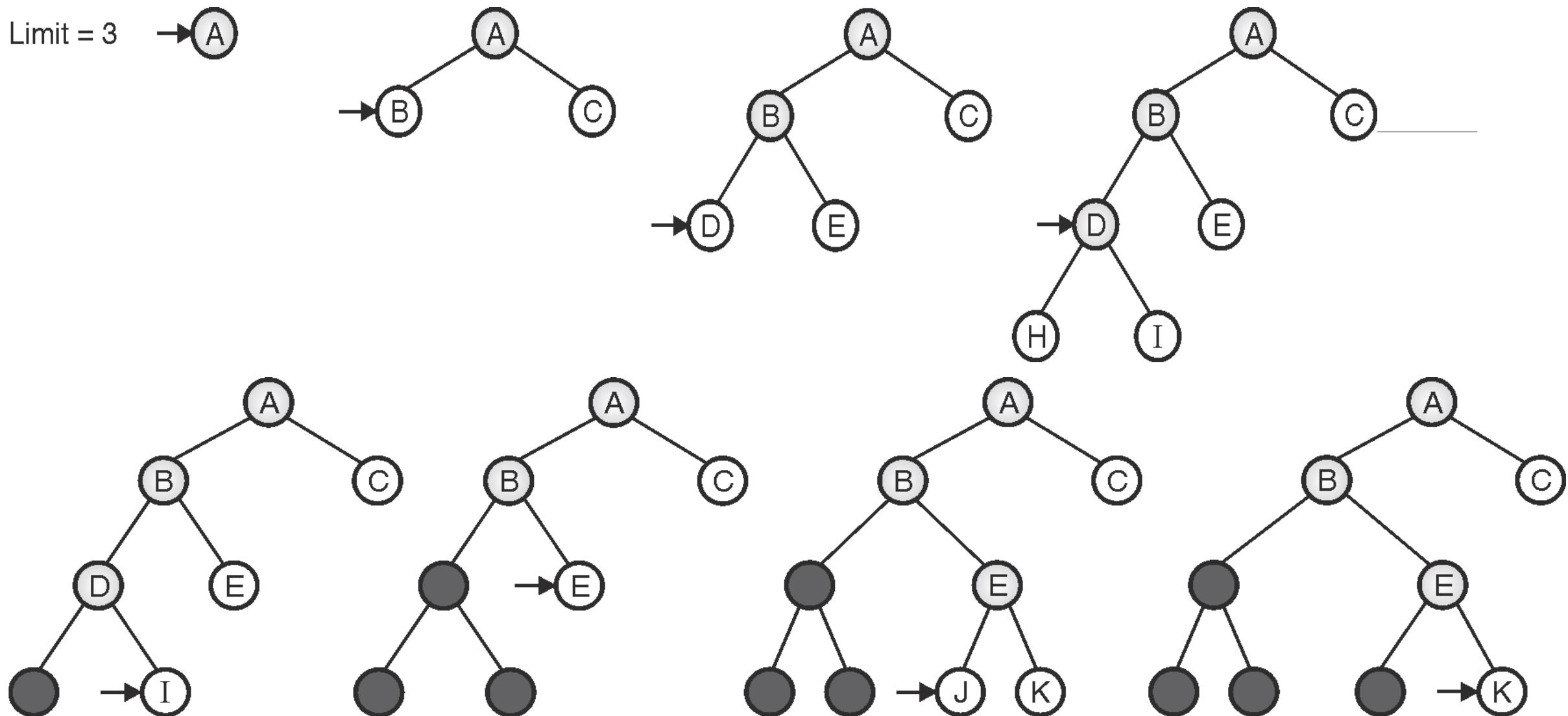
Limit = 1 → A

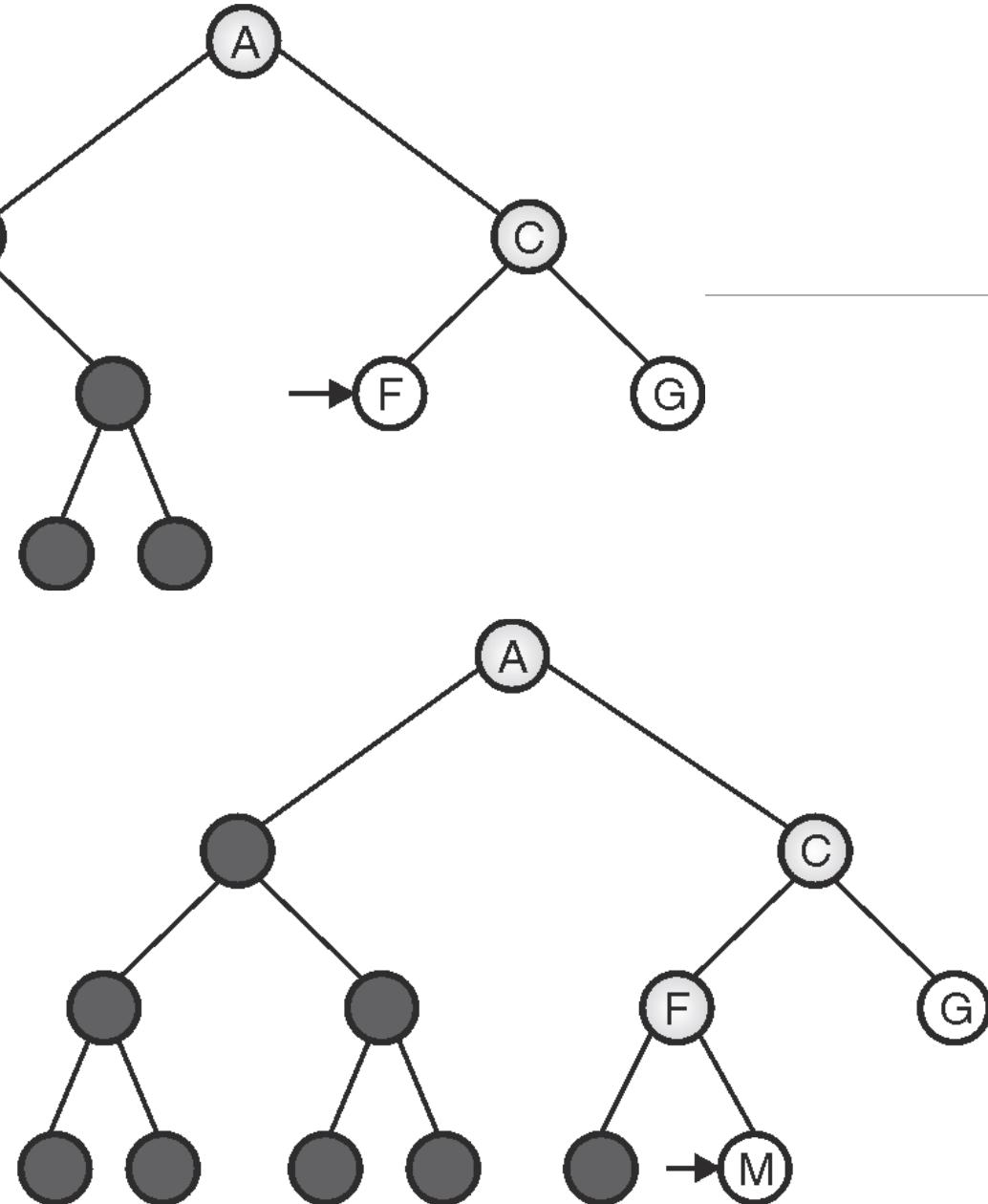
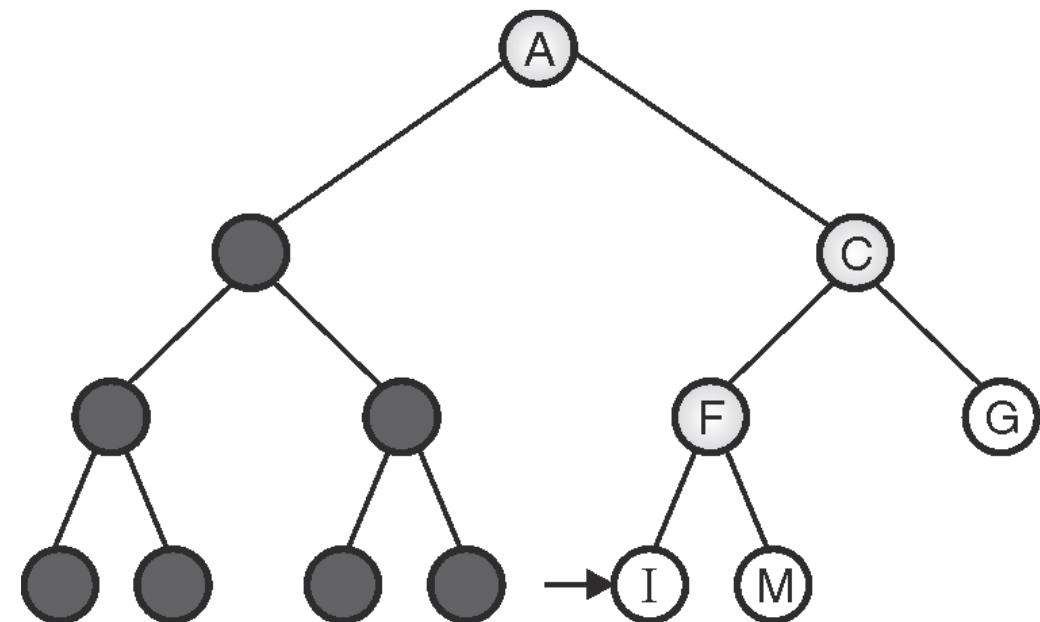
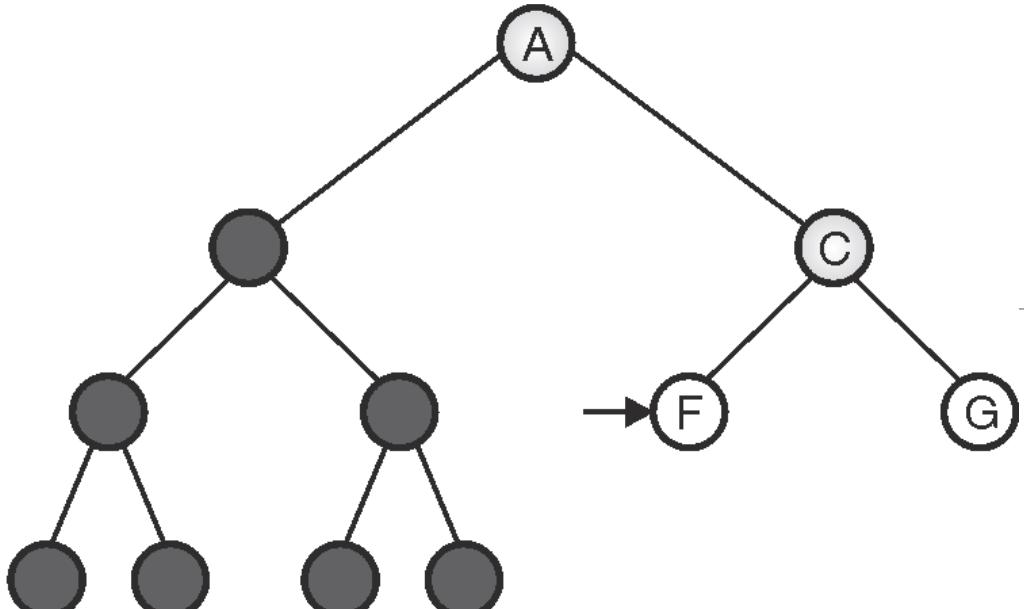
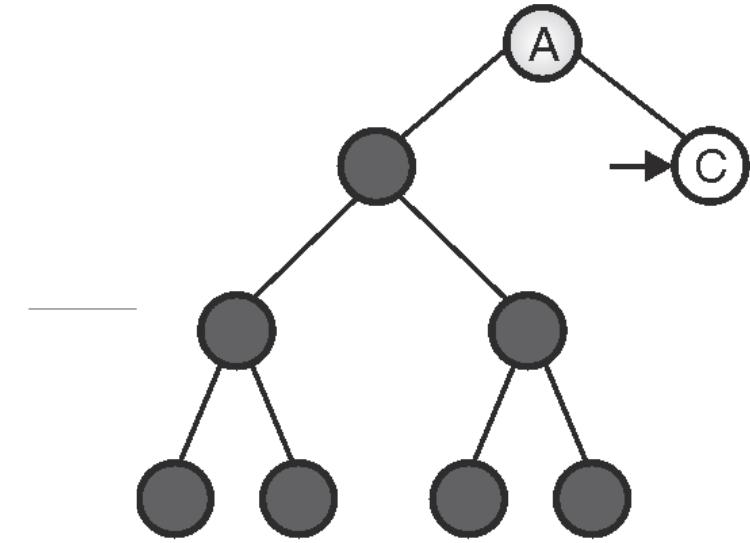


Limit = 2 → A



Limit = 3





IDDFS: DBDFS to Depth Bound 0

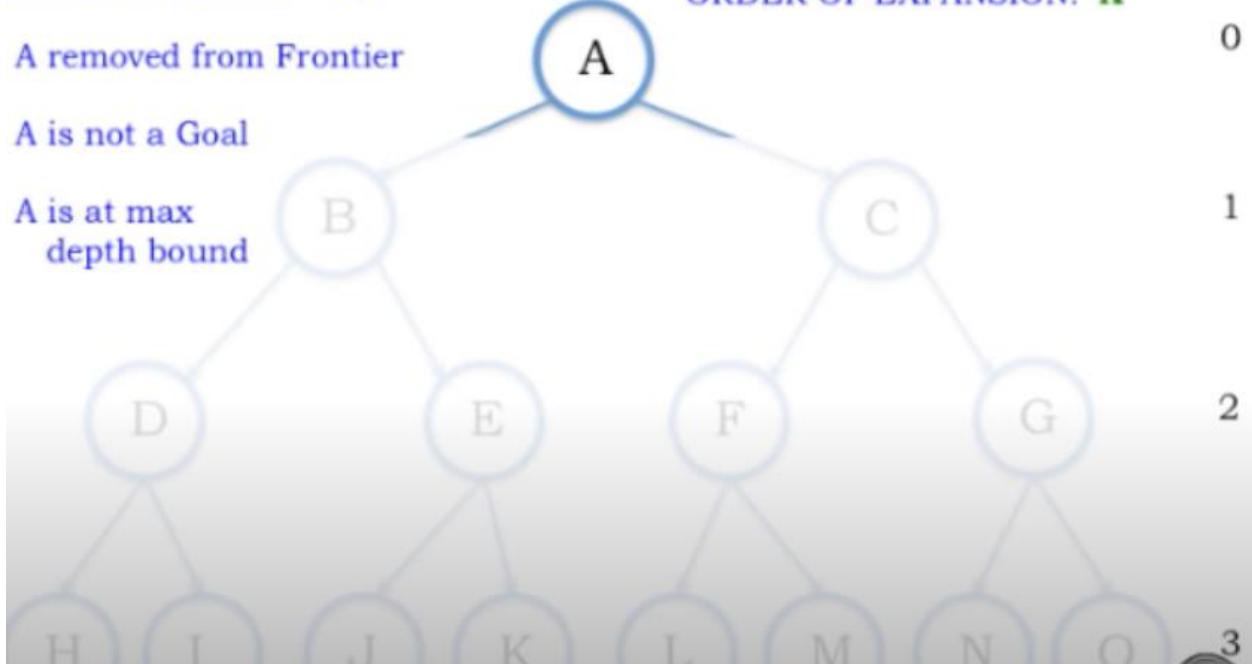
Initialize Frontier = {A}

A removed from Frontier

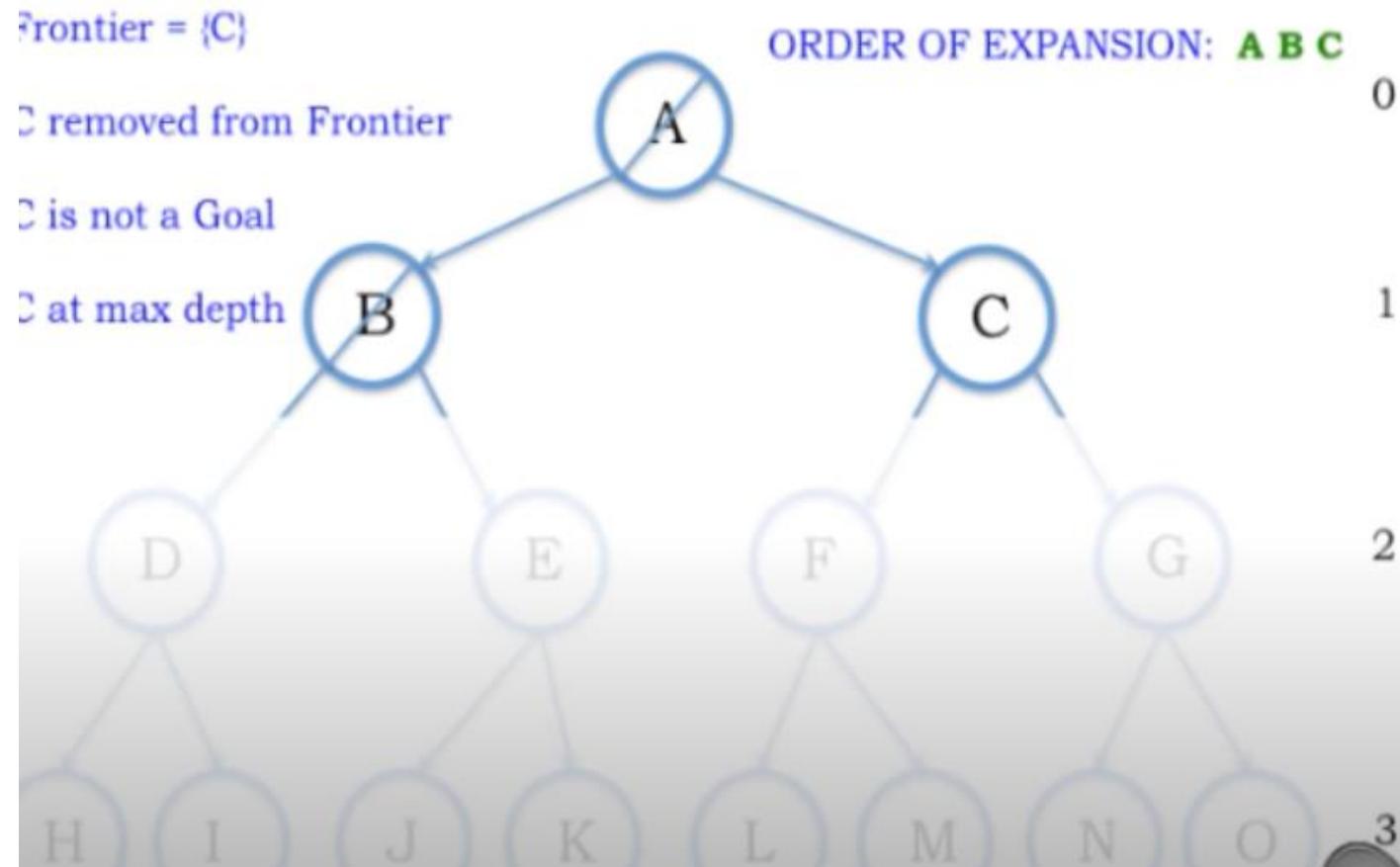
A is not a Goal

A is at max depth bound

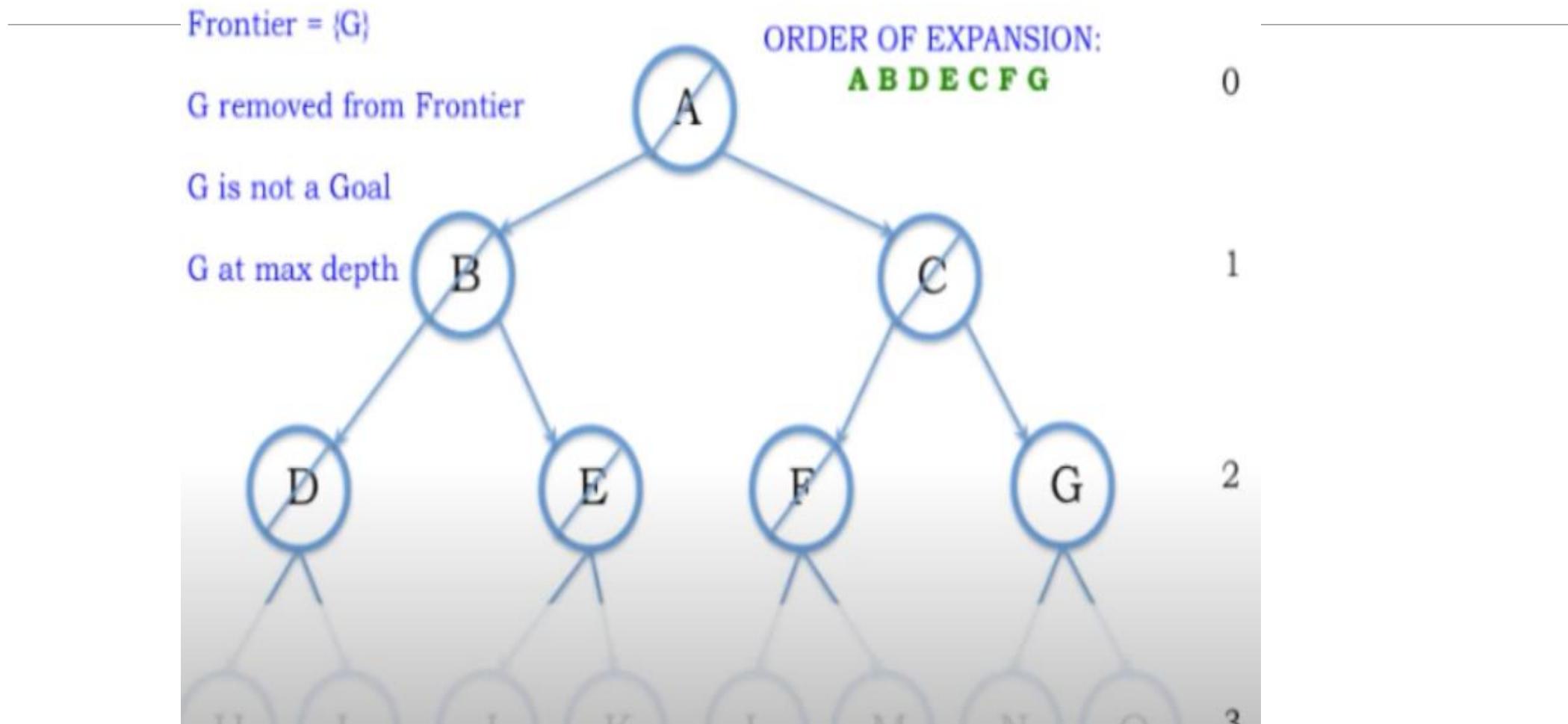
ORDER OF EXPANSION: A



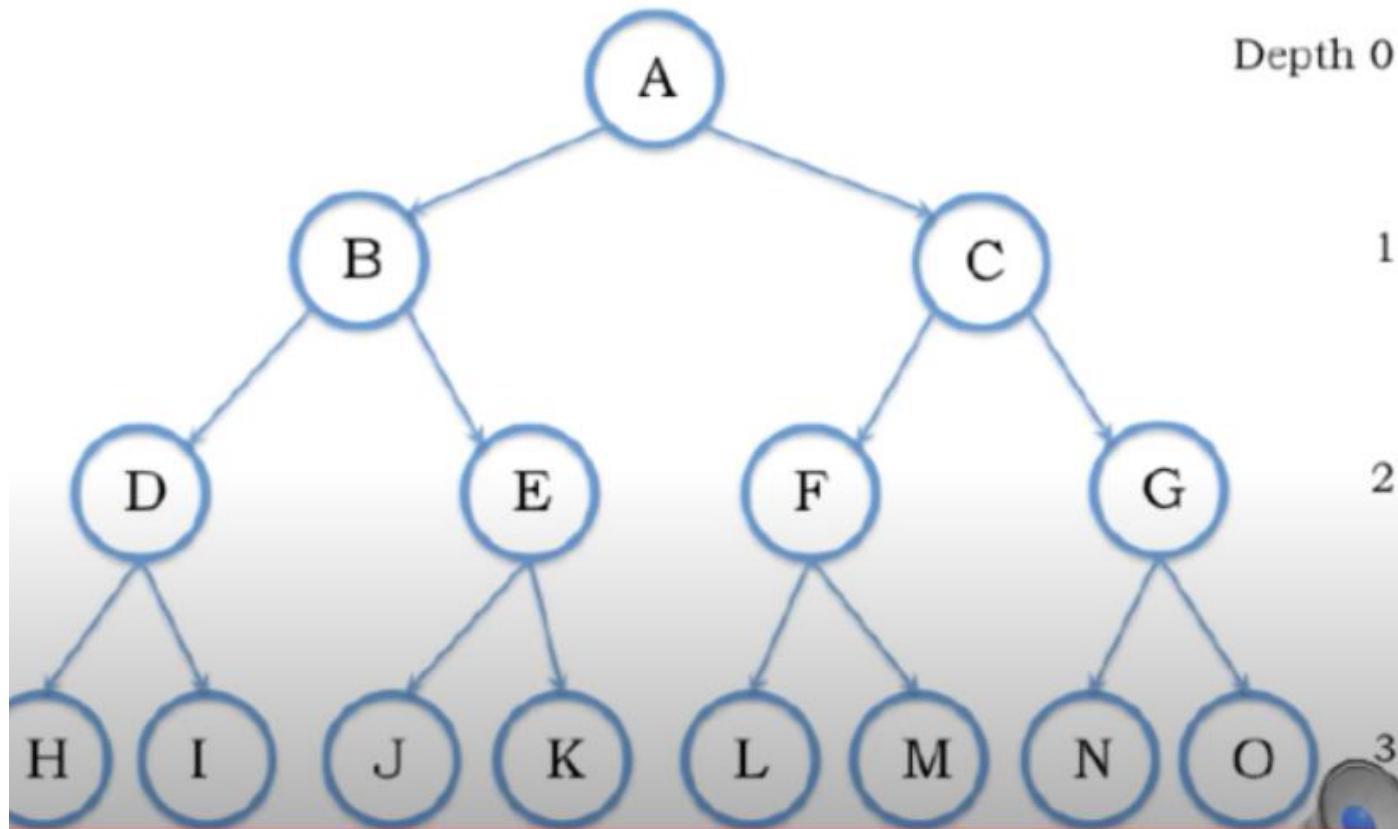
IDDFS: DBDFS to Depth Bound 1



IDDFS: DBDFS to Depth Bound 2

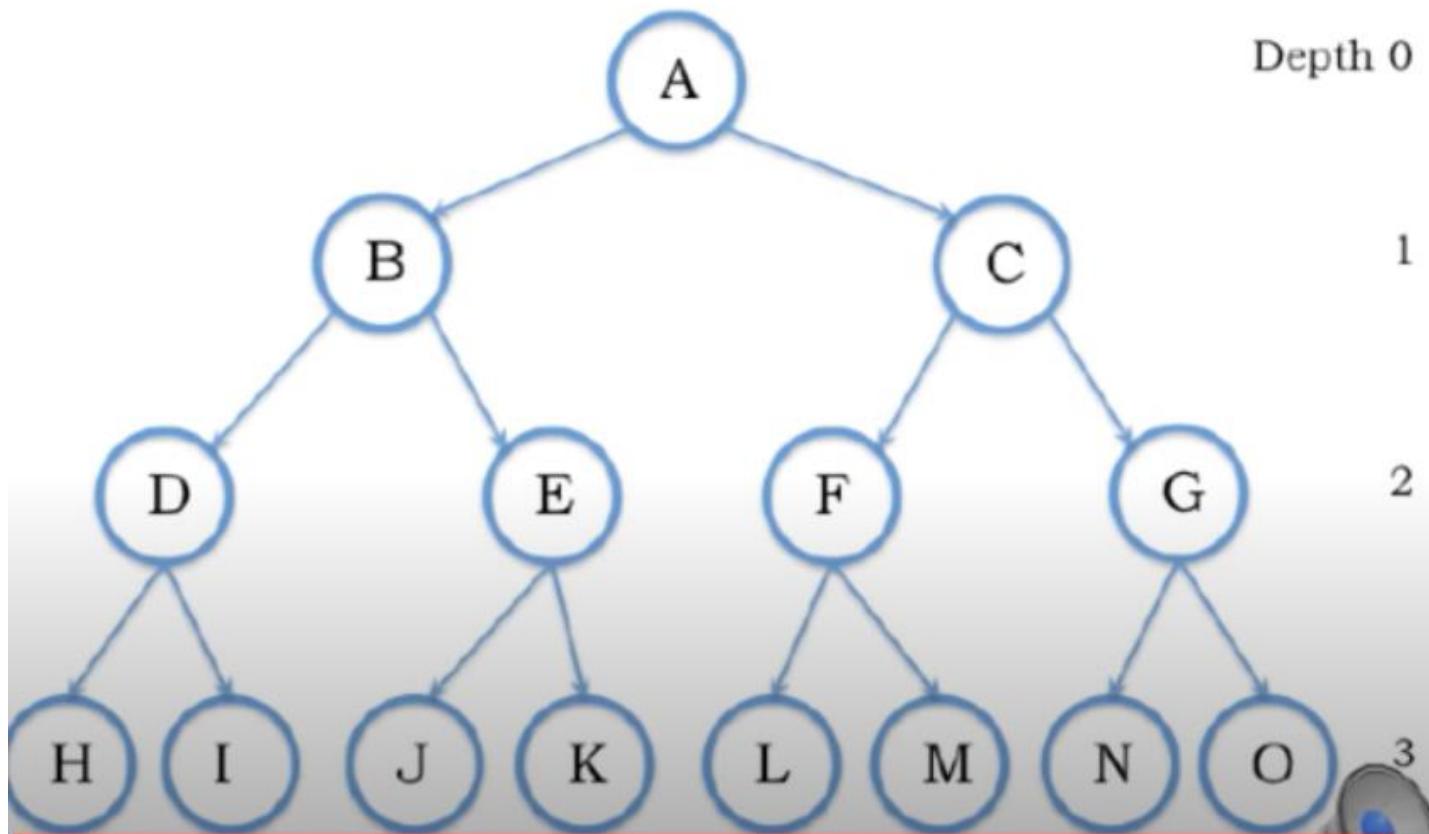


Find path for depth3



Over ALL the iterations, from depth bound 0 to 3, the order in which nodes removed from the frontier is:

A A B C A B D E C F G A B D H I E J K C F L M G N O



Algorithm

- Initialize depth limit to zero.
- Repeat Until the goal node is found.
 - (a) Call Depth limited search with new depth limit.
 - (b) Increment depth limit to next level.

Pseudo Code

```
IDDFS()  
{  
    limit = 0;  
    found = false;  
    while (not found)  
    {  
        found = DLS(root, limit, 0);  
        limit = limit + 1;  
    }  
}
```

Performance Evaluation

Completeness : IDDFS is complete when the branching factor b is finite.

Optimality : It is optimal when the path cost is a non-decreasing function of the depth of the node.

Time complexity :

- o The nodes on the bottom level that is level ' d ' are generated only once, those on the next to bottom level are generated twice, and so on, up to the children of the root, which are generated d times. Hence the time complexity is $O(b^d)$.

Space complexity : Memory requirements of IDDFS are modest, i.e. $O(b^d)$.

Uniform Cost Search (UCS)

- The algorithm shown below is almost same as BFS; except for the use of a **priority queue** and the addition of an extra check in case a shorter path to any node is discovered.
- The algorithm takes care of nodes which are inserted in the fringe for exploration, by using a data structure having priority queue and hash table.
- The priority queue used here contains total cost from root to the node. Uniform cost search gives the minimum path cost the maximum priority.

Algorithm

- Insert the root node into the queue.

While the queue is not empty :

(i) Dequeue the maximum priority node from the queue.

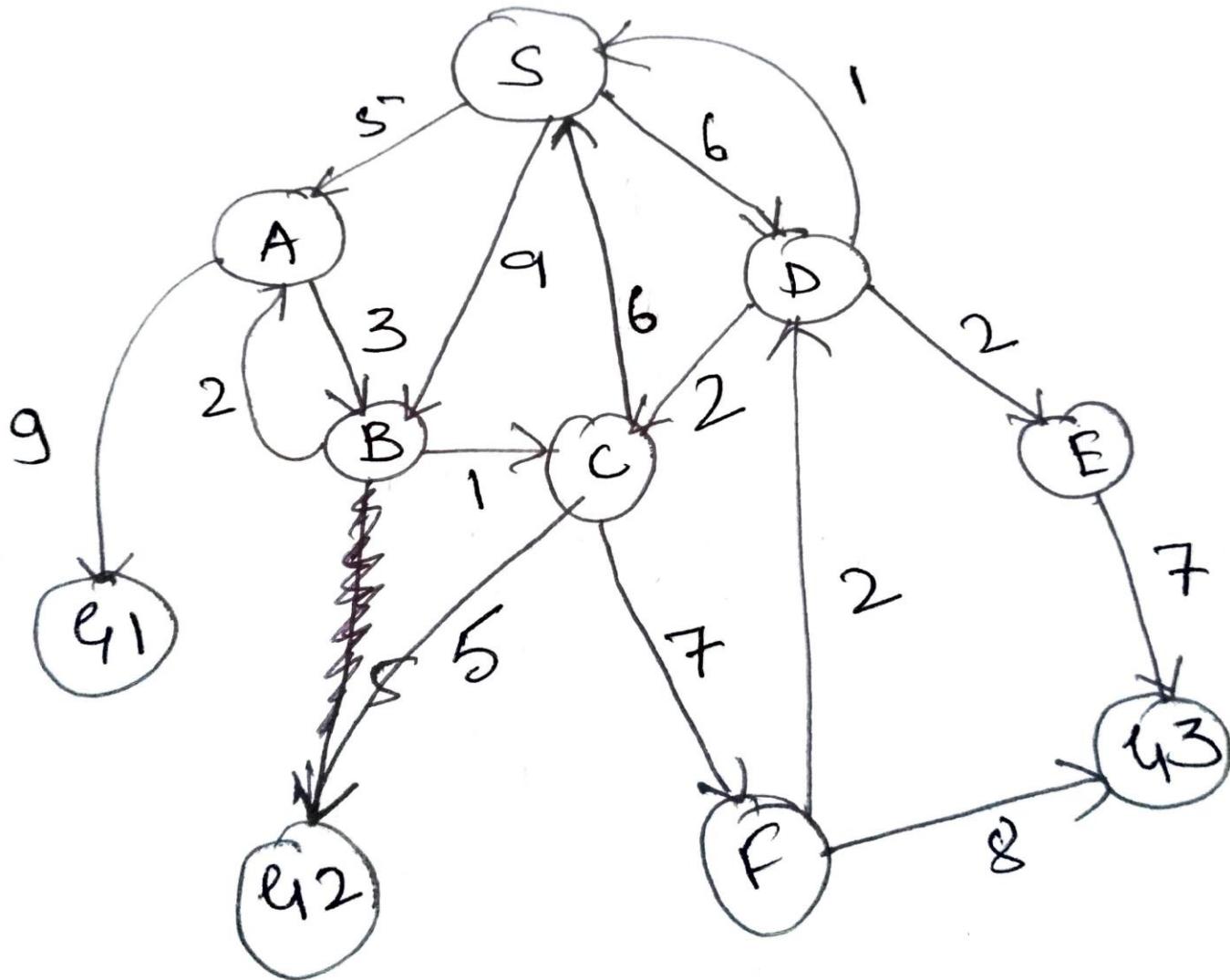
(If priorities are same, alphabetically smaller node is chosen)

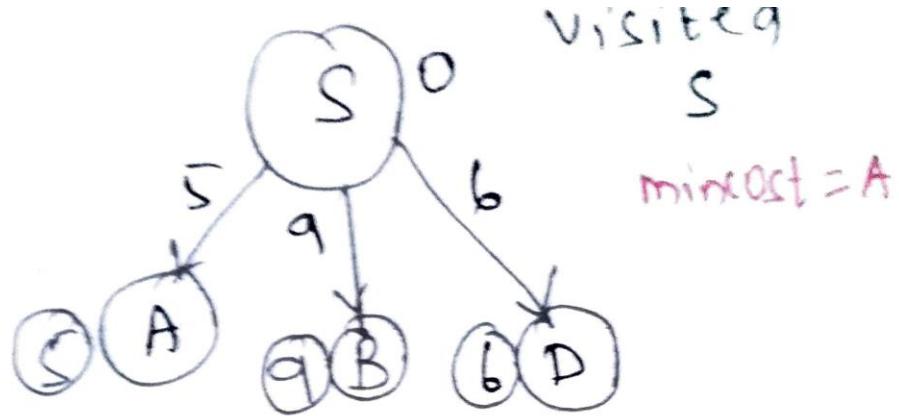
(ii) If the node is the goal node, print the path and exit.

Else

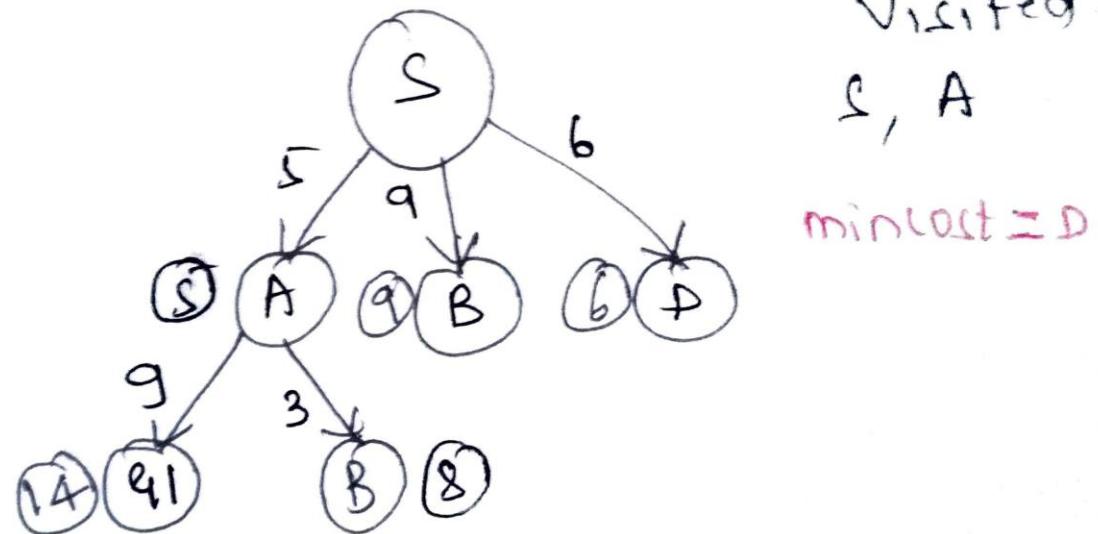
- Insert all the children of the dequeued node, with their total costs as priority.
- The algorithm returns the best cost path which is encountered first and will never go for other possible paths. The solution path is optimal in terms of cost.
- As the priority queue is maintained on the basis of the total path cost of node, the algorithm never expands a node which has a cost greater than the cost of the shortest path in the tree.
- The nodes in the priority queue have almost the same costs at a given time, and thus the name “Uniform Cost Search”.

Uniform cost search

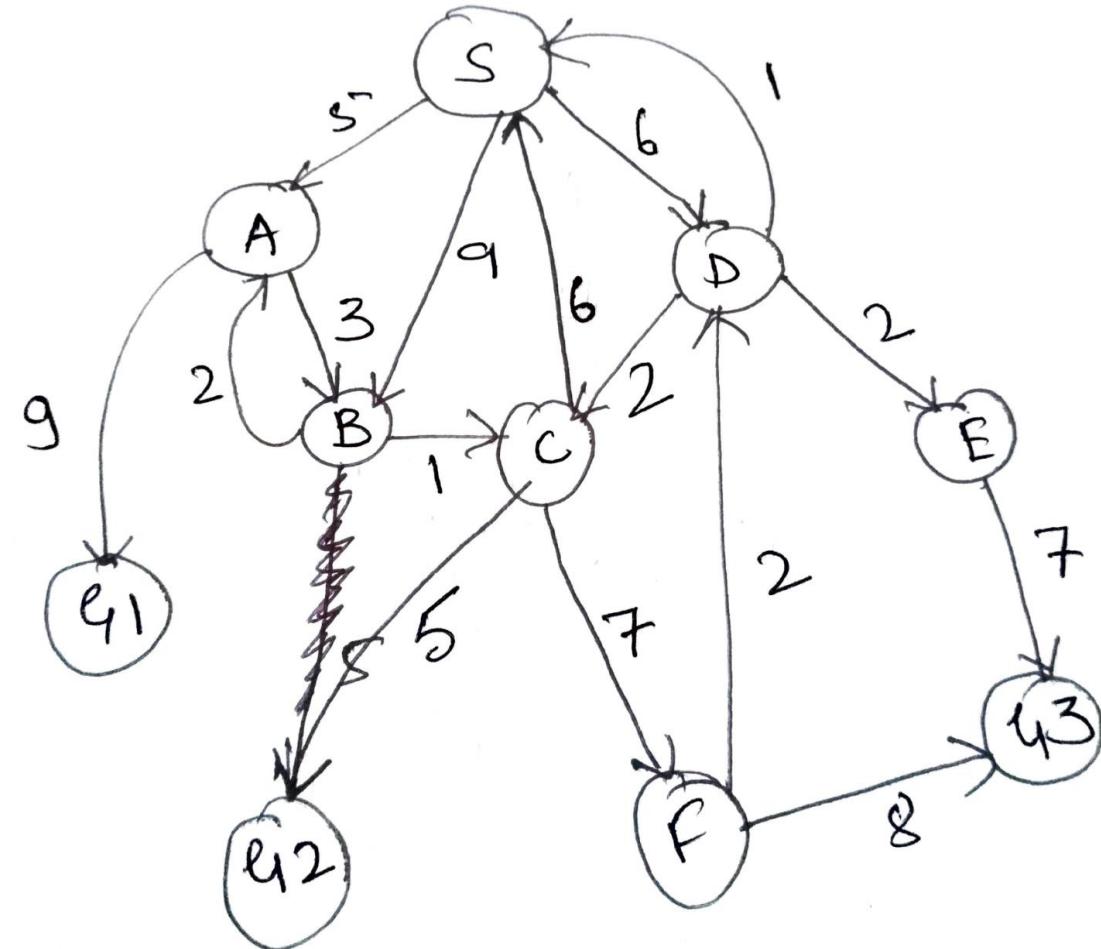




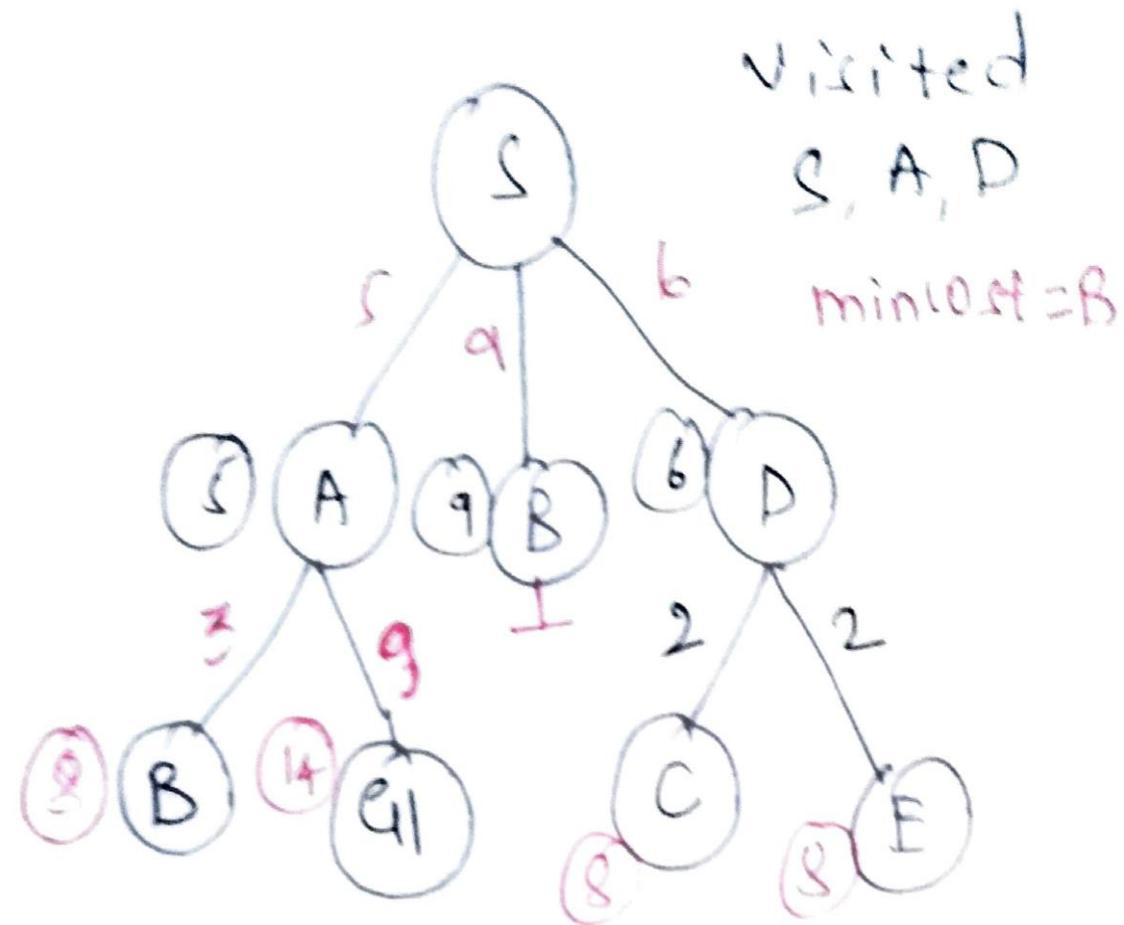
Step 2



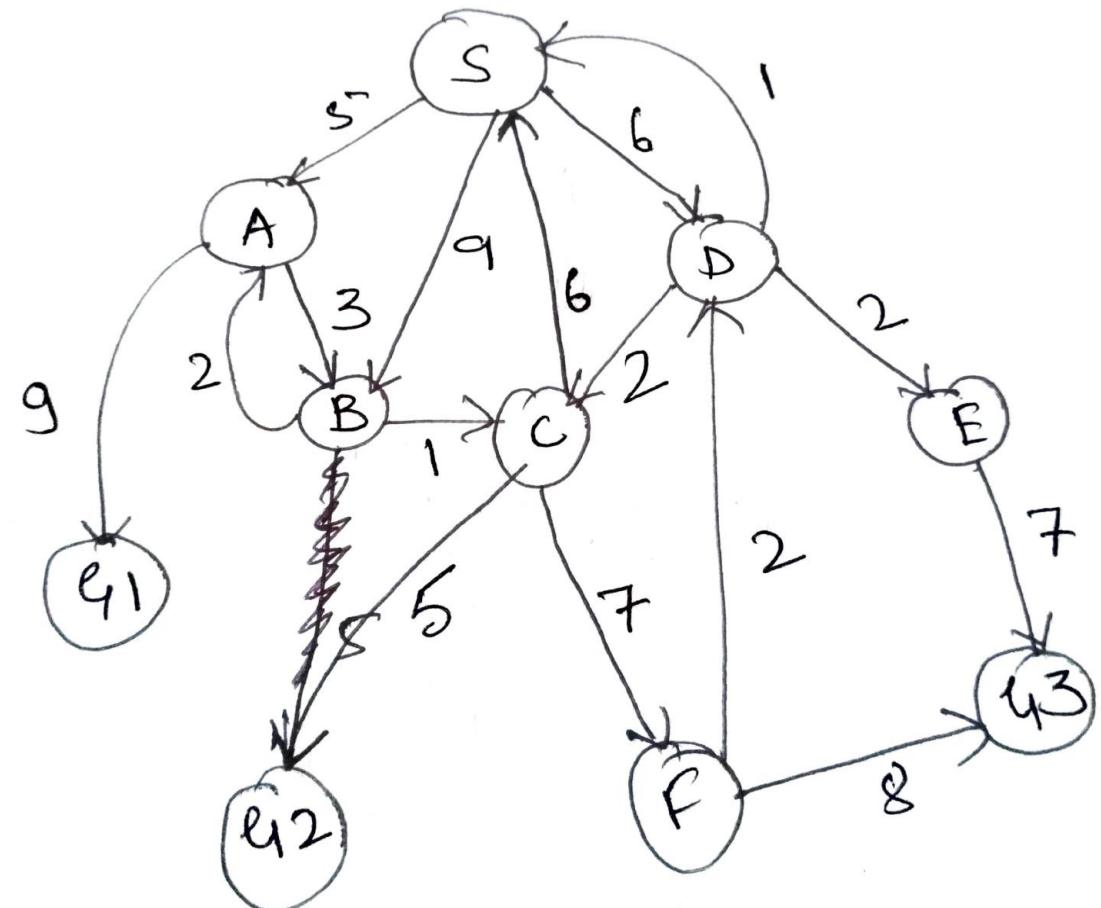
Uniform cost search



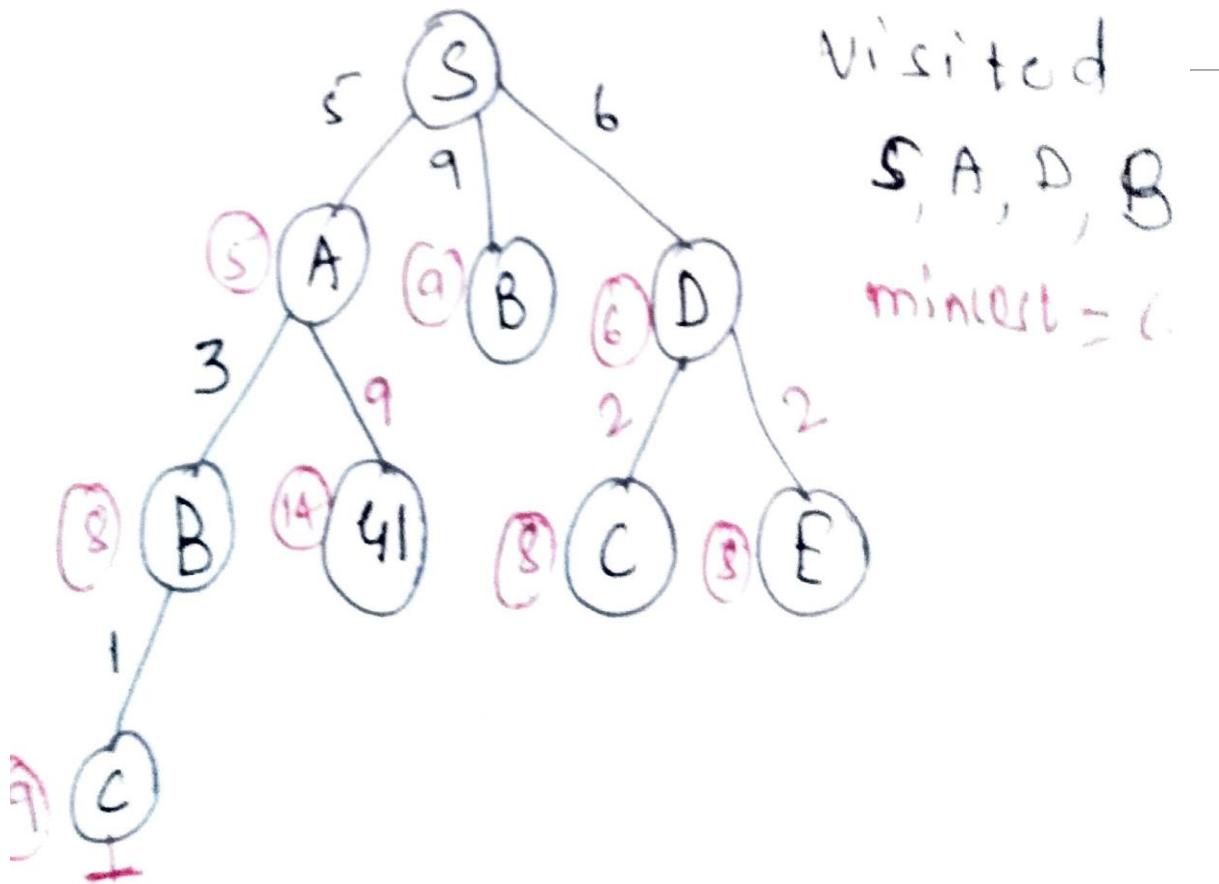
Step 3



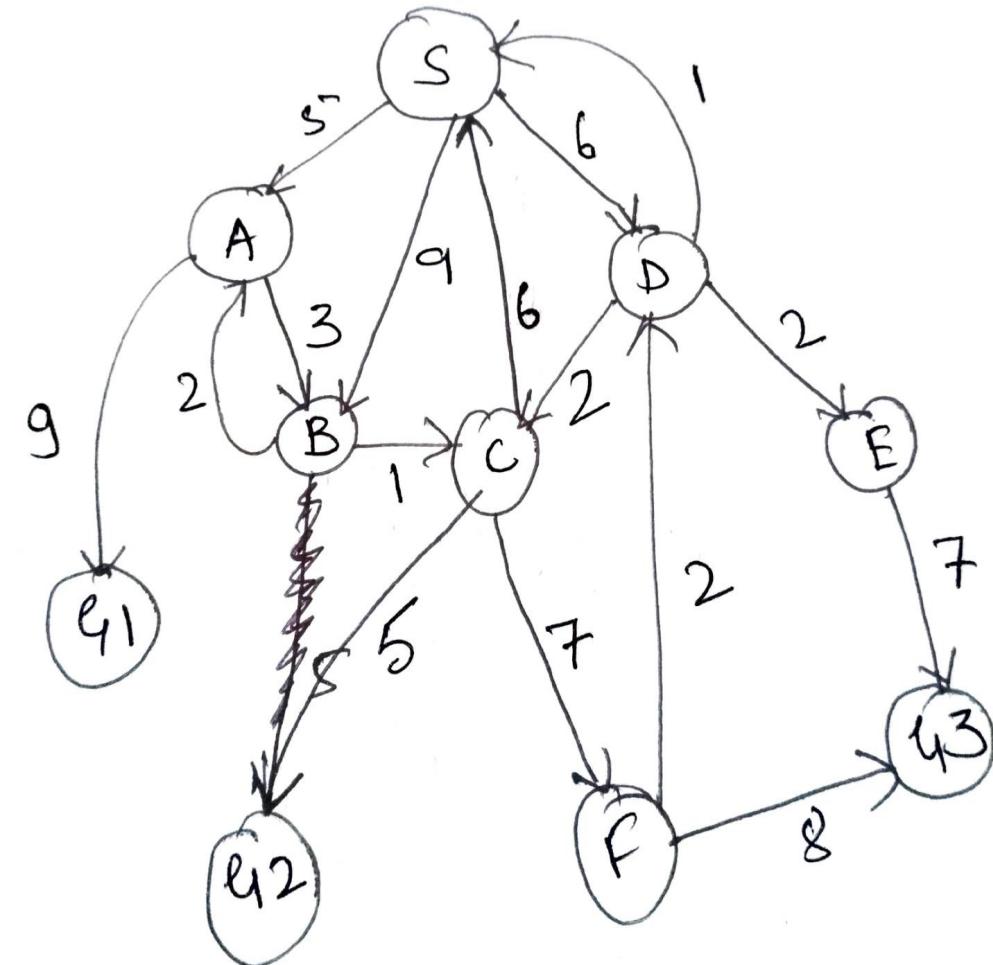
Uniform cost search



Step 4

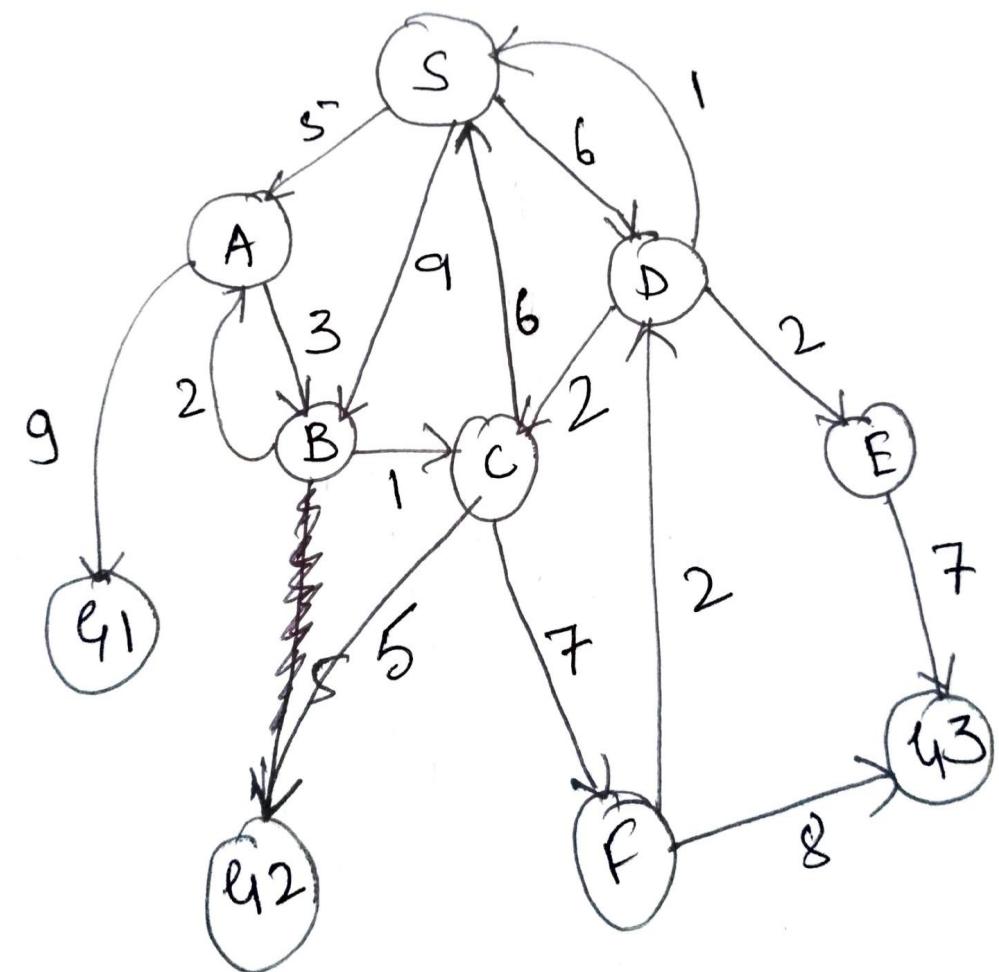
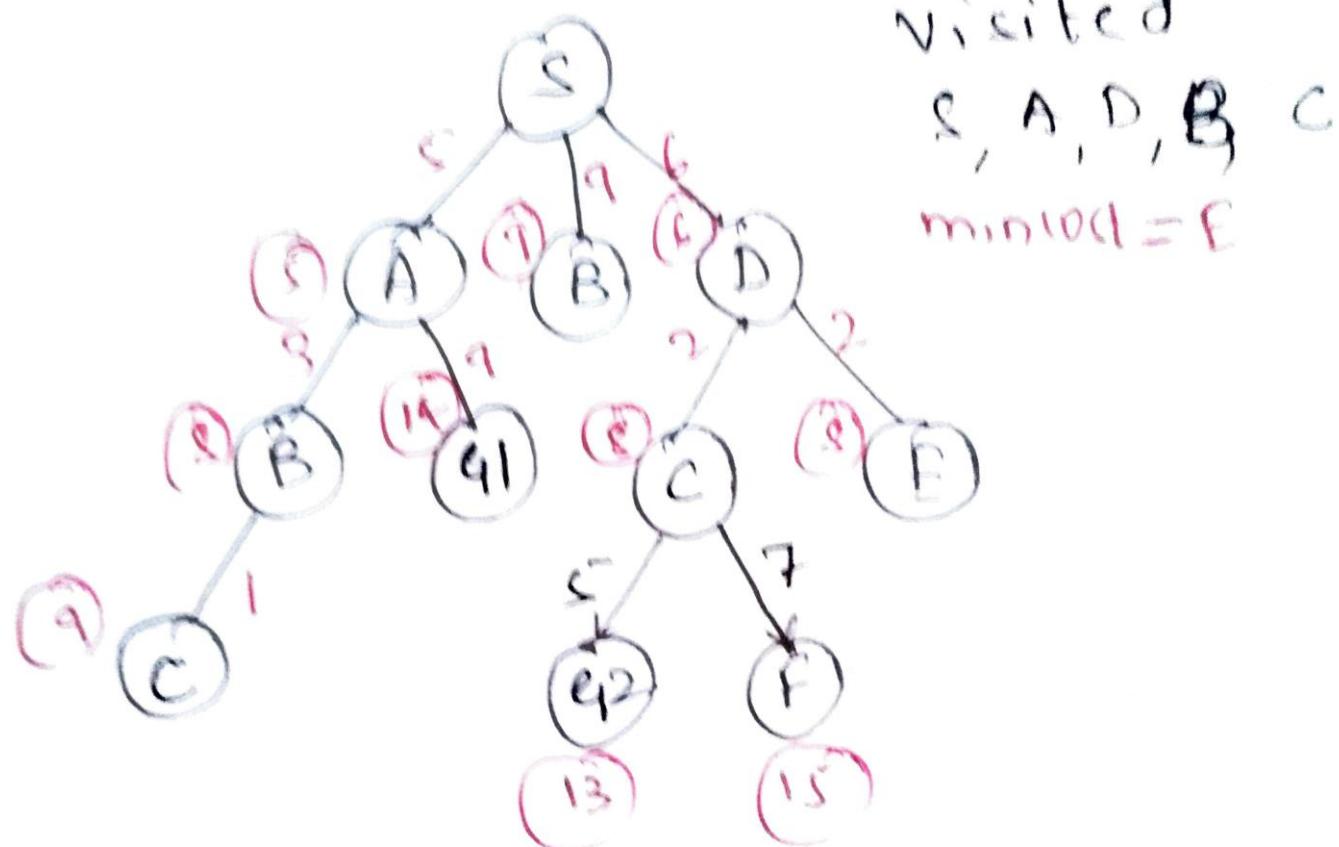


Uniform cost search

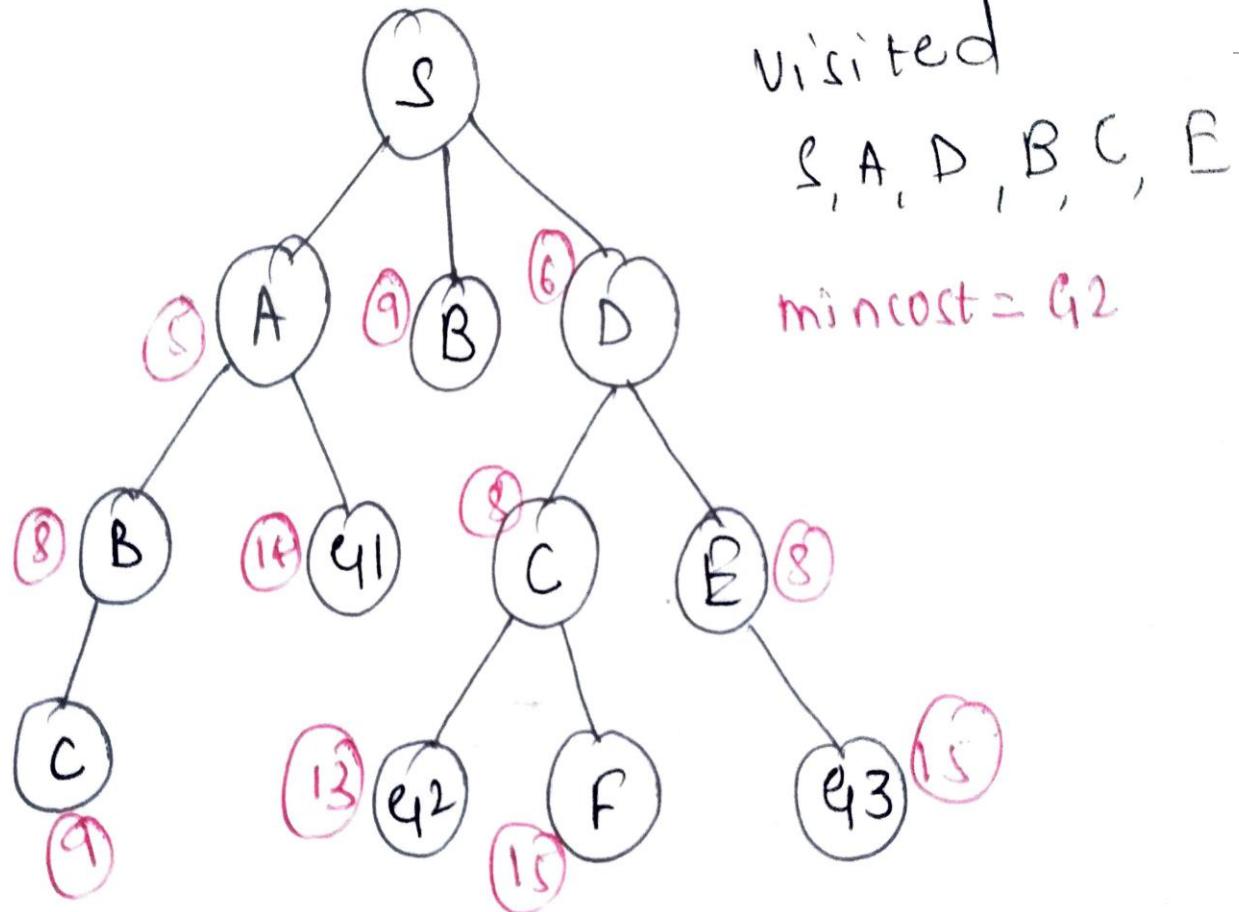


Uniform cost search

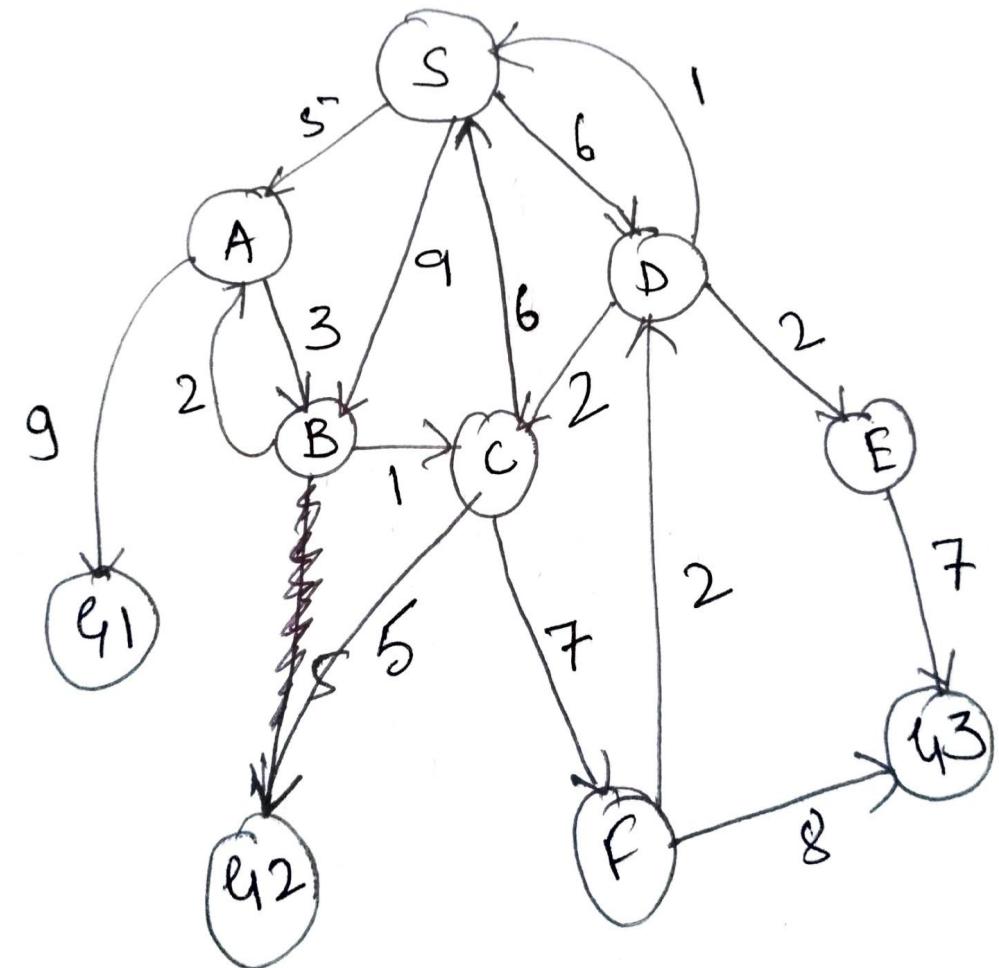
Step 5

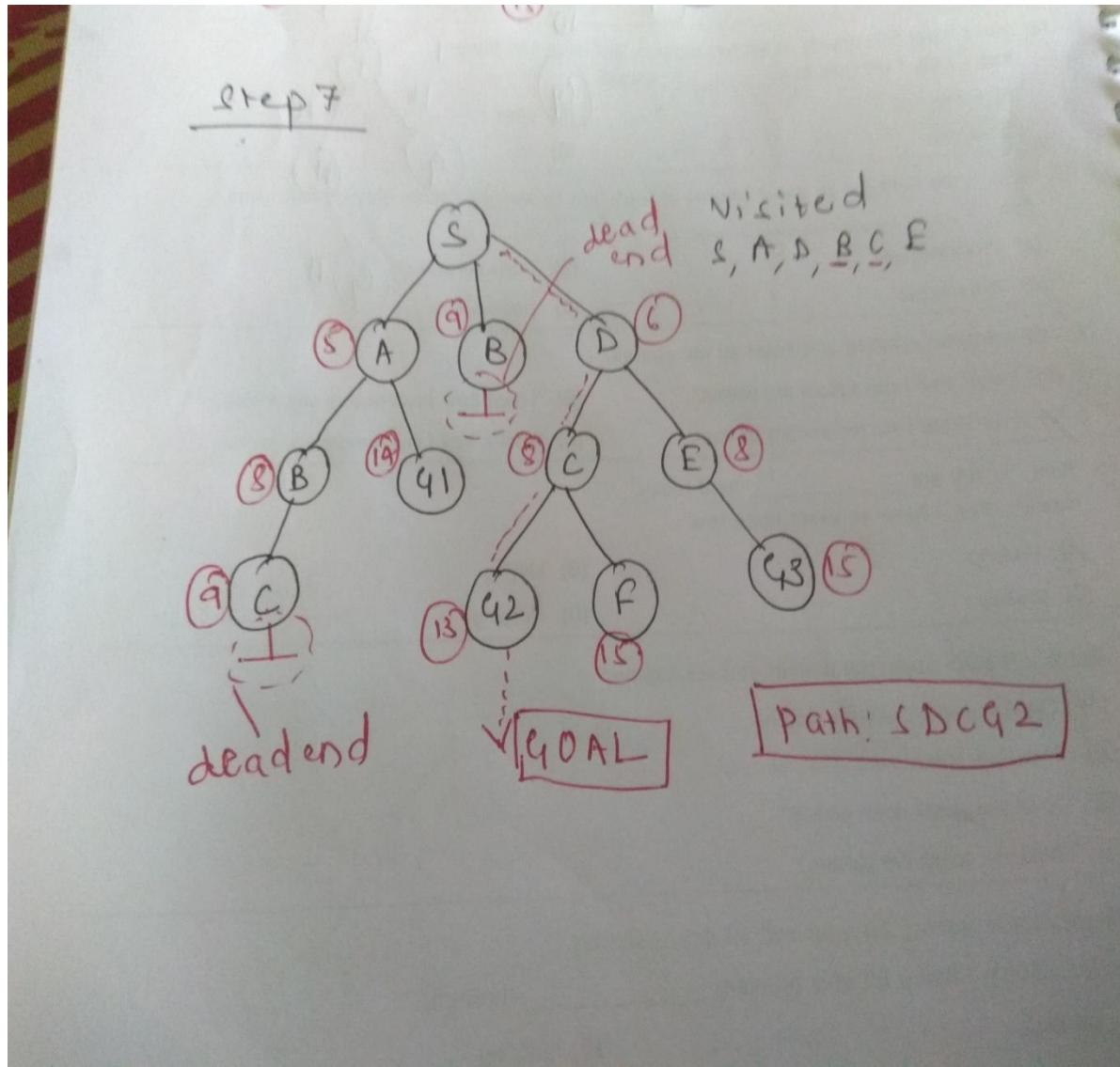


Step 6

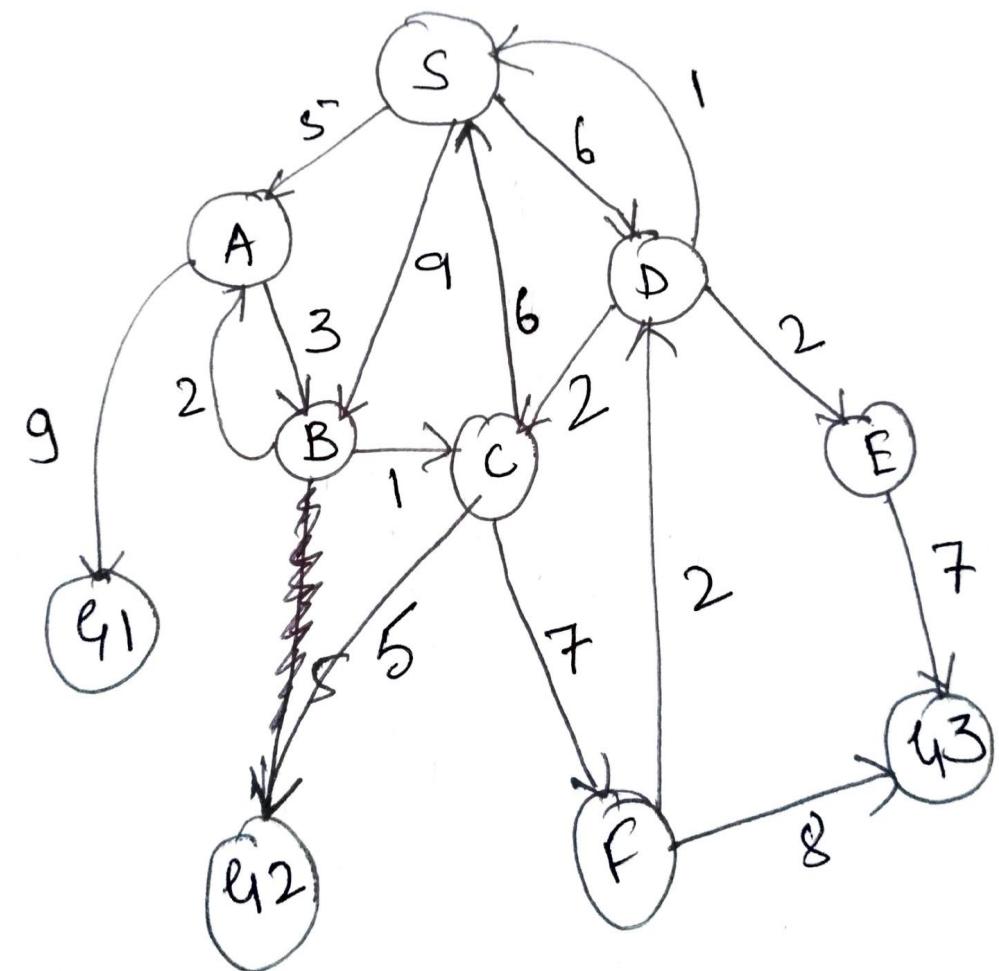


Uniform cost search

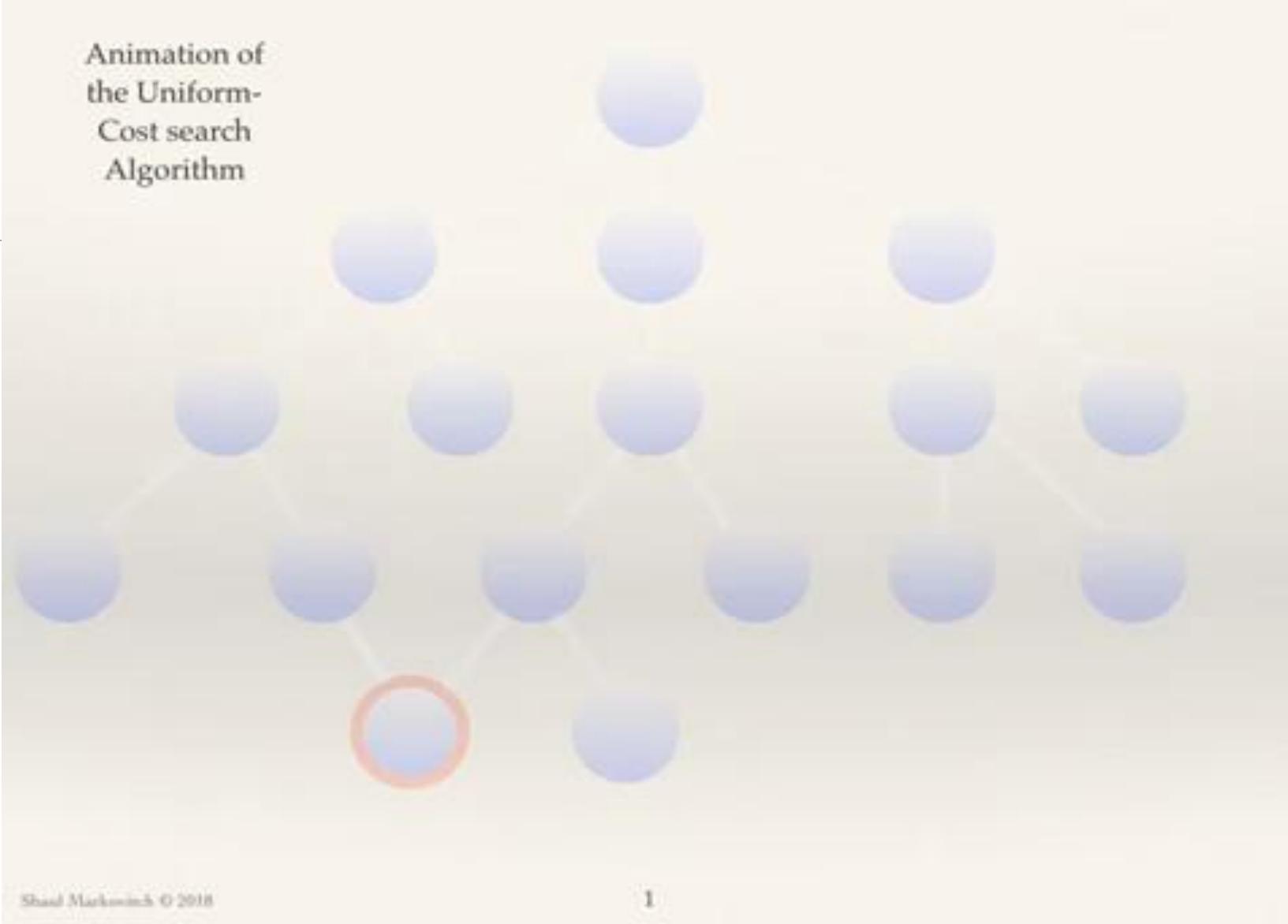




Uniform cost search



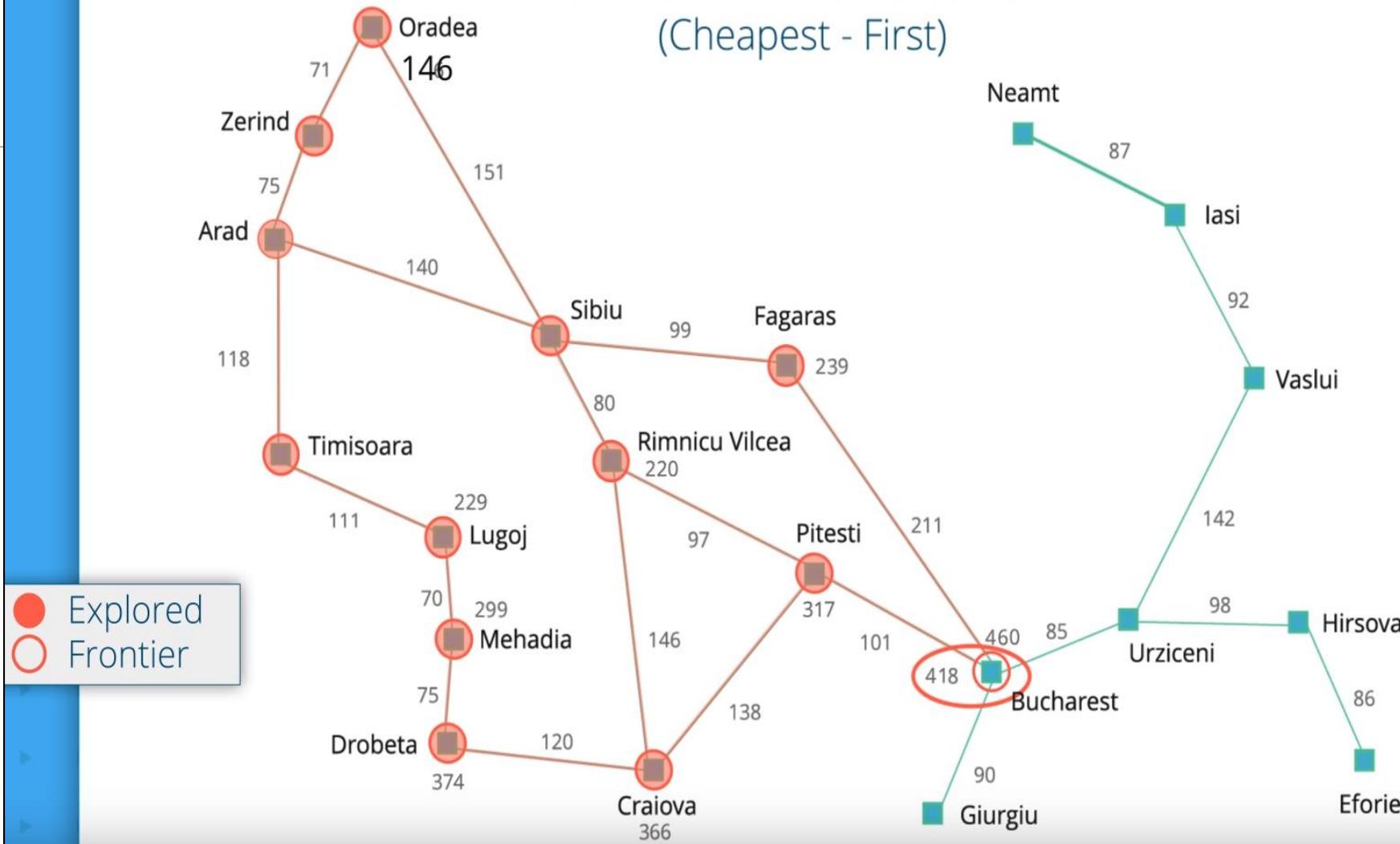
Animation of
the Uniform-
Cost search
Algorithm



Shad Markowich © 2018

1

Uniform Cost Search (Cheapest - First)



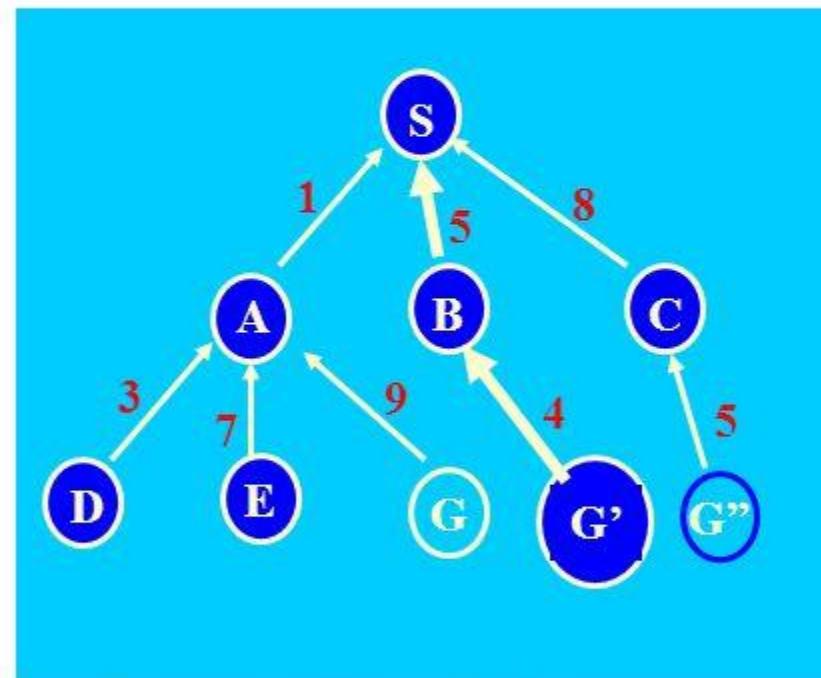
Uniform-Cost Search: example of operation

GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)

exp. node nodes list

	{S(0)}
S	{A(1) B(5) C(8)}
A	{D(4) B(5) C(8) E(8) G(10)}
D	{B(5) C(8) E(8) G(10) }
B	{C(8) E(8) G'(9) G(10) }
C	{E(8) G'(9) G(10) G''(13)}
E	{G'(9) G(10) G''(13) }
G'	{G(10) G''(13) }

CLOSED list

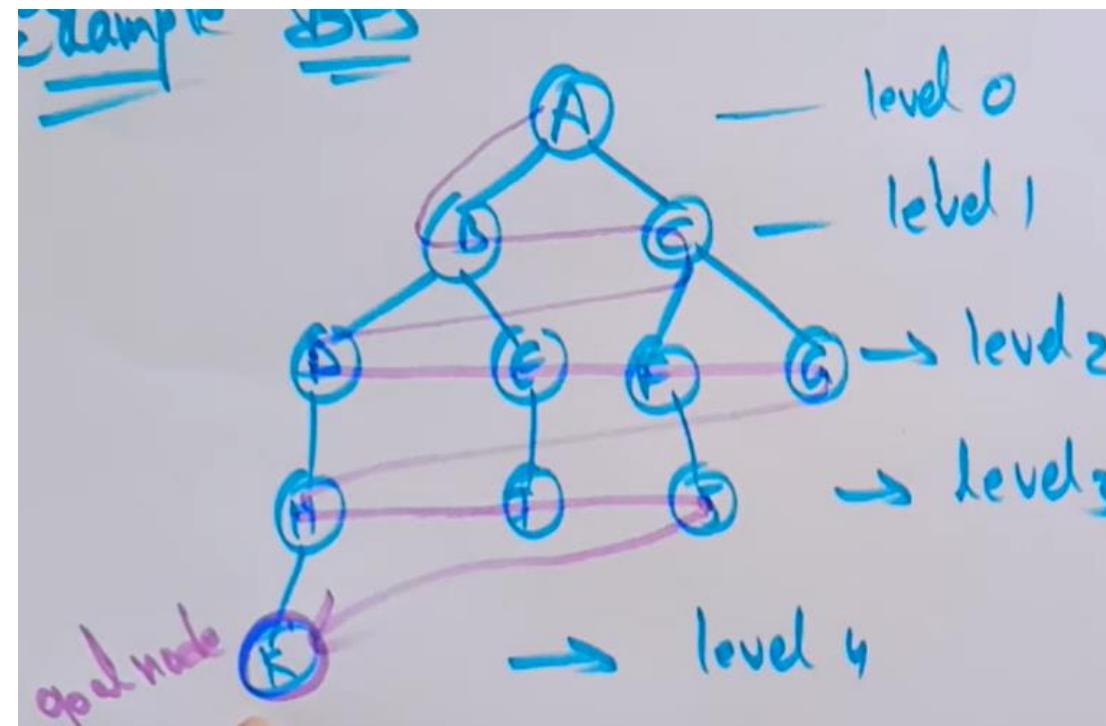
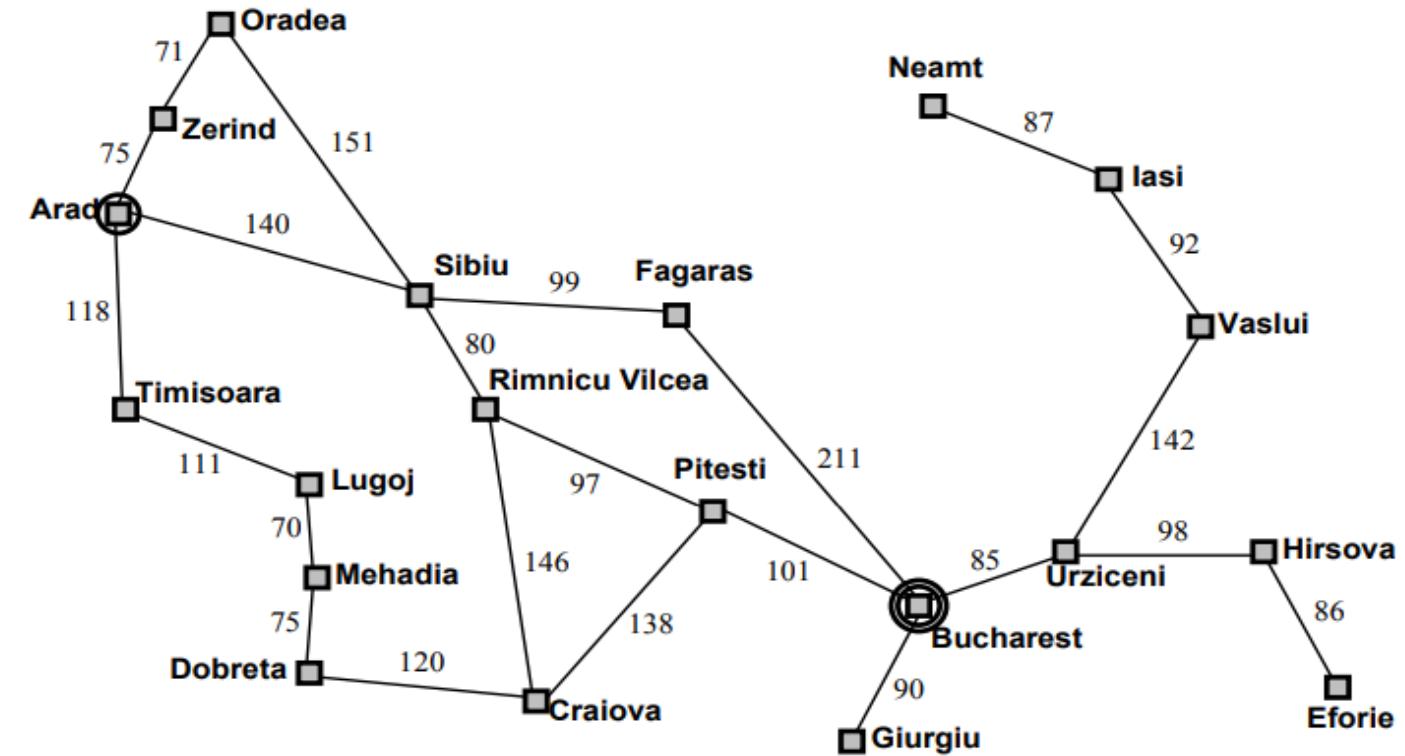


Solution path found is S B G <-- this G has cost 9, not 10

Number of nodes expanded (including goal node) = 7

• This example illustrated that Uniform-Cost can loose the best solution

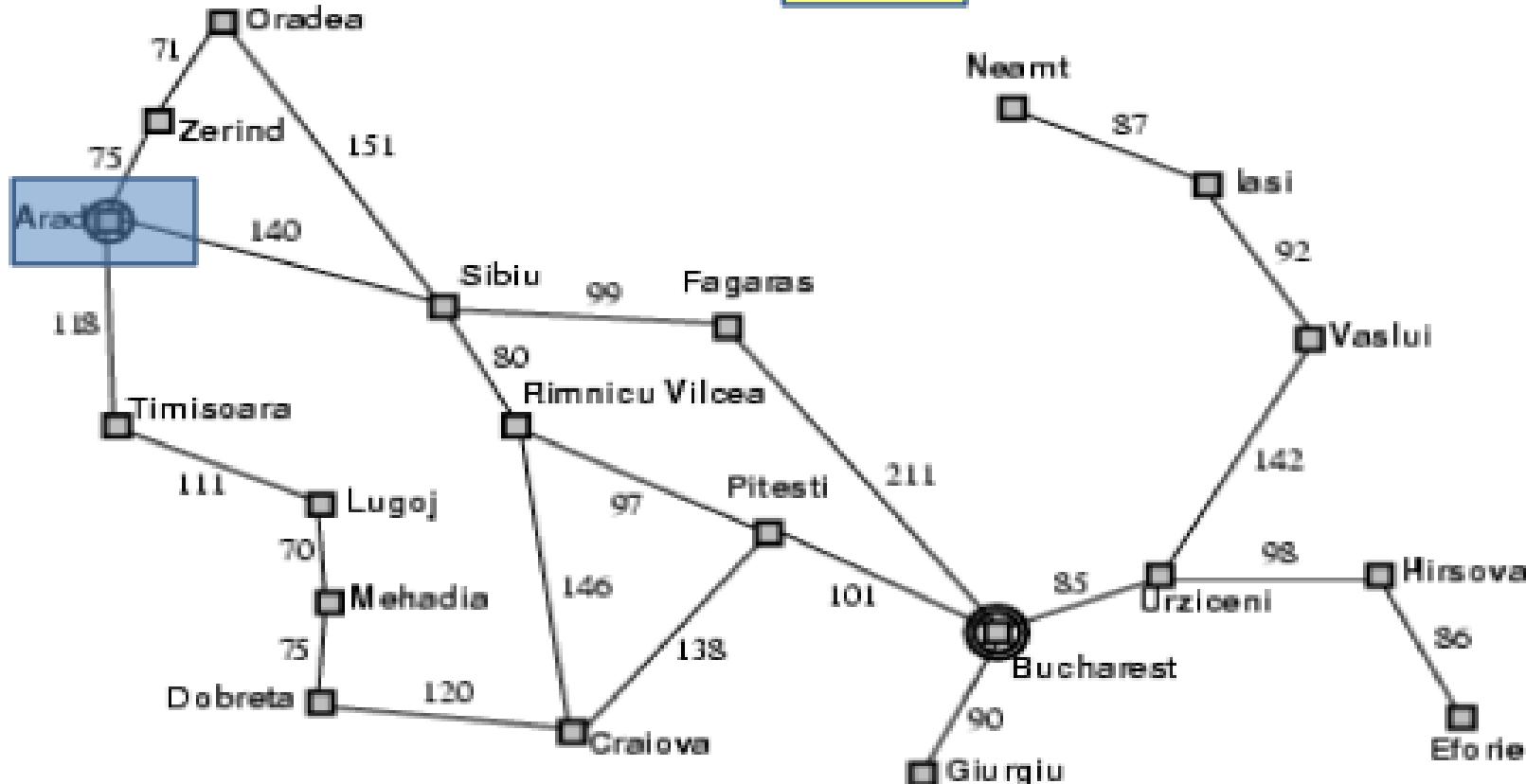
Problem: To find a route from Arad to Bucharest (Romania Problem)



Example: Romania BFS

Travel from Arad to Bucharest

D= 0

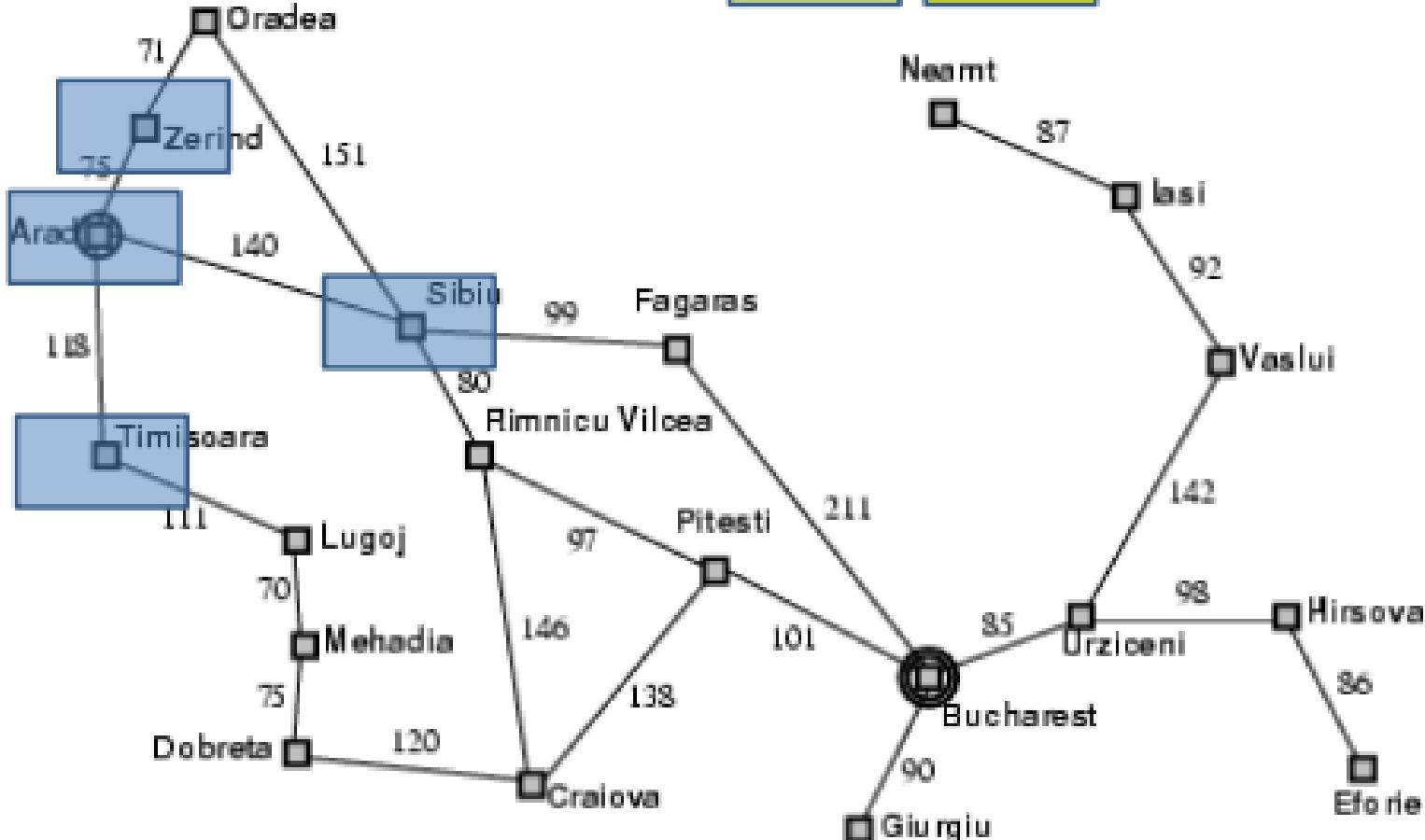


Example: Romania BFS

Travel from Arad to Bucharest

D= 0

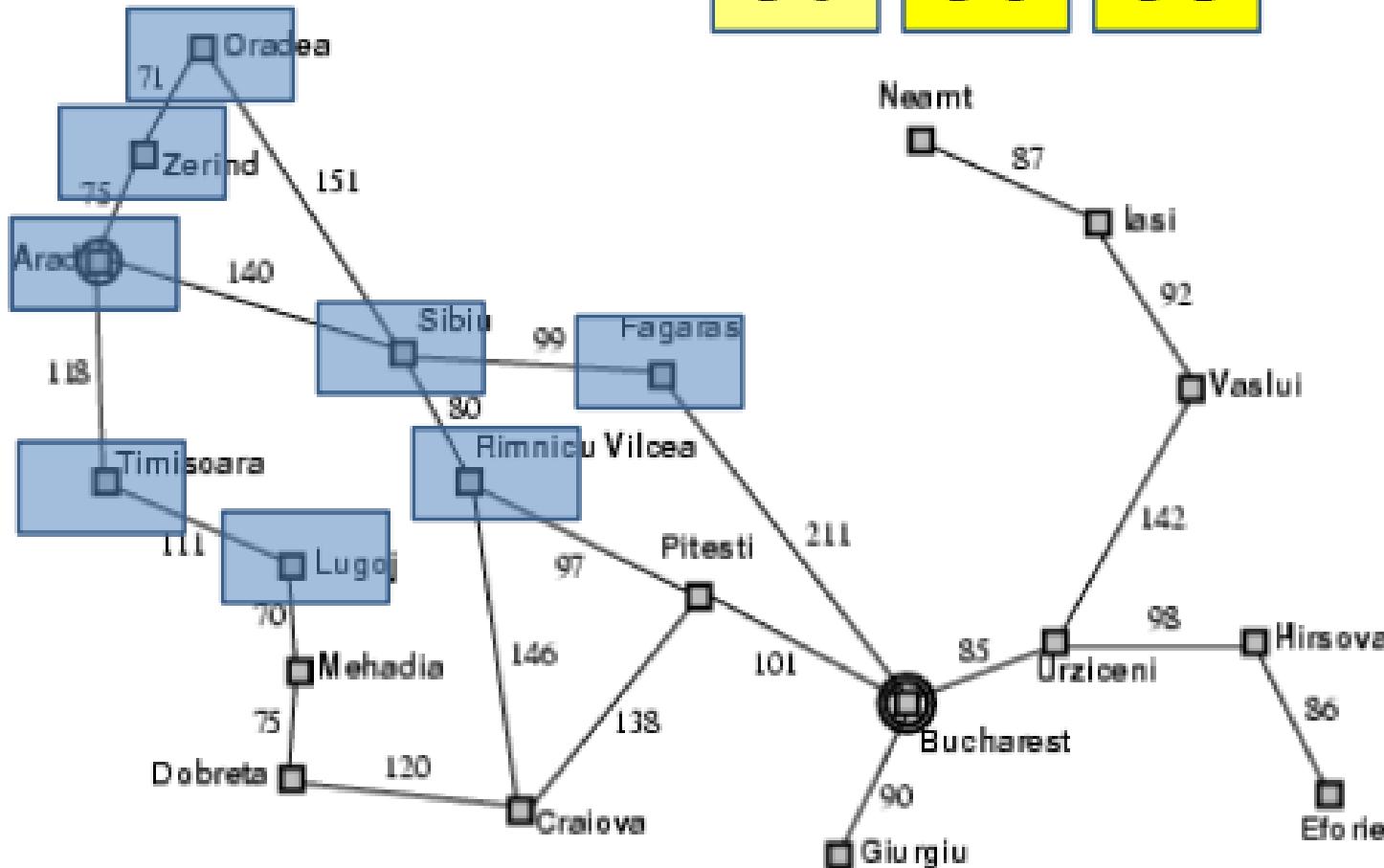
D= 1



Example: Romania BFS

Travel from Arad to Bucharest

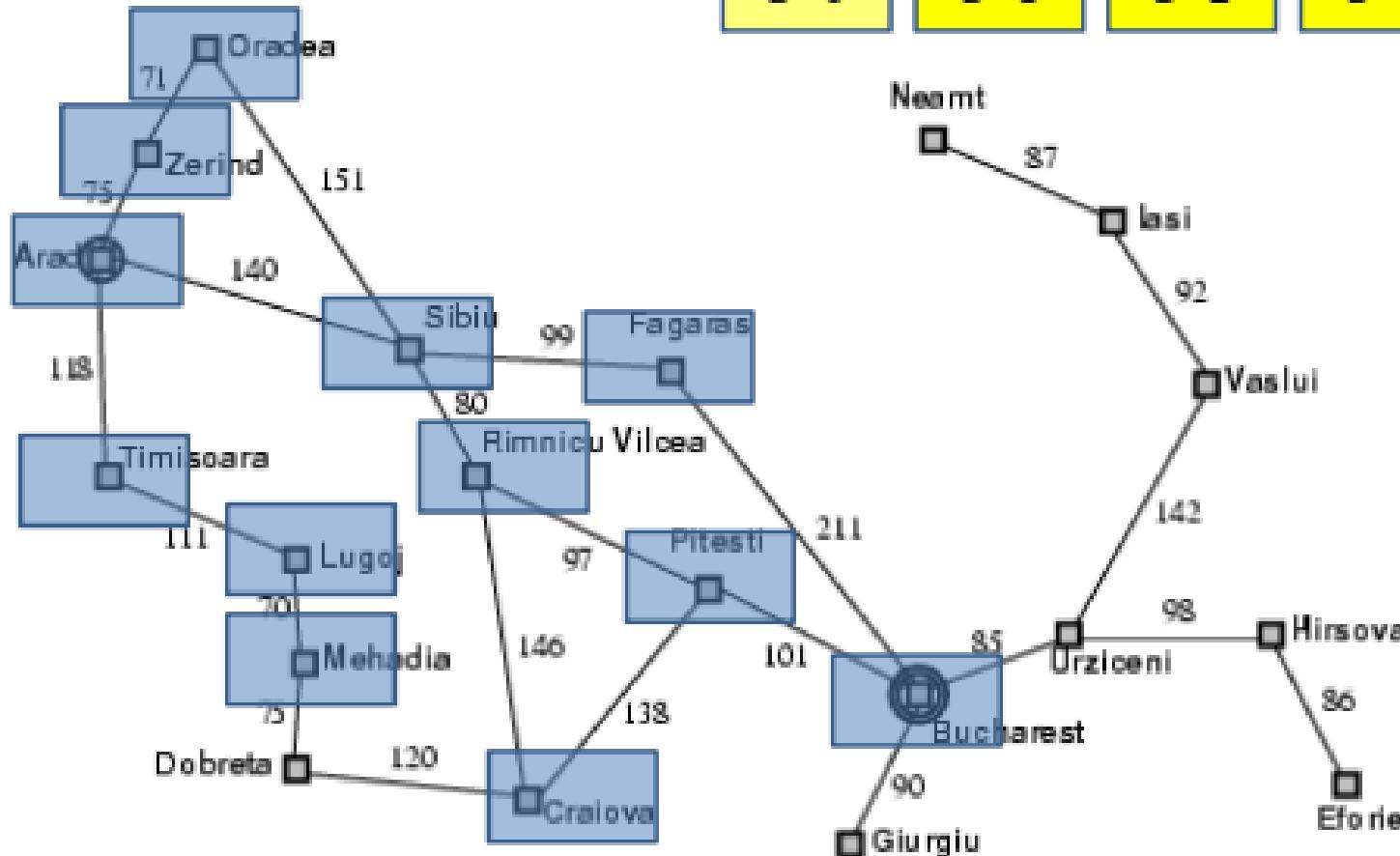
D = 0 D = 1 D = 2



Example: Romania BFS

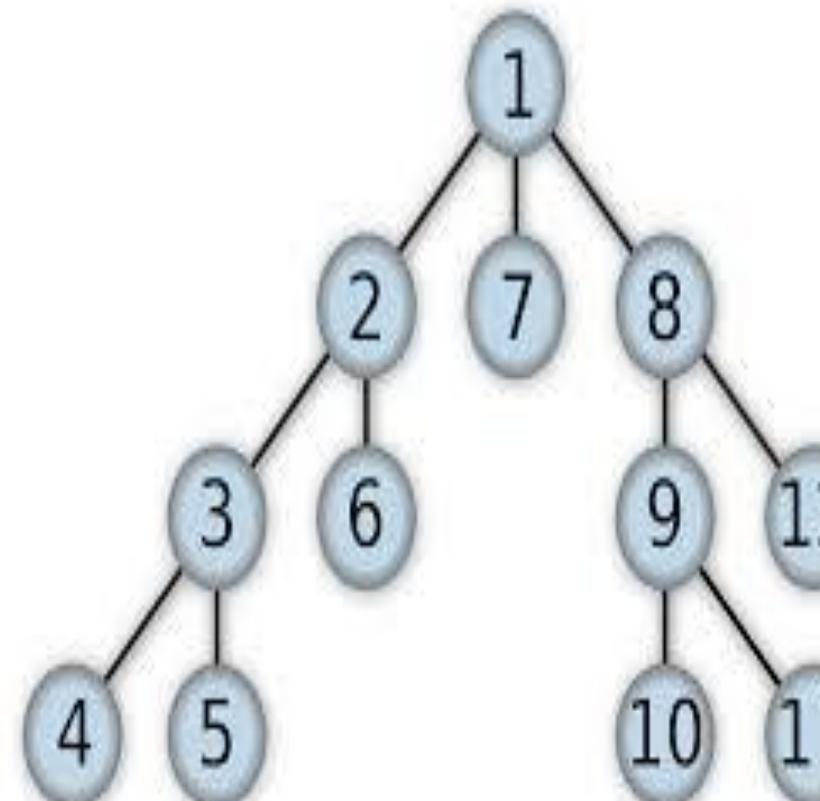
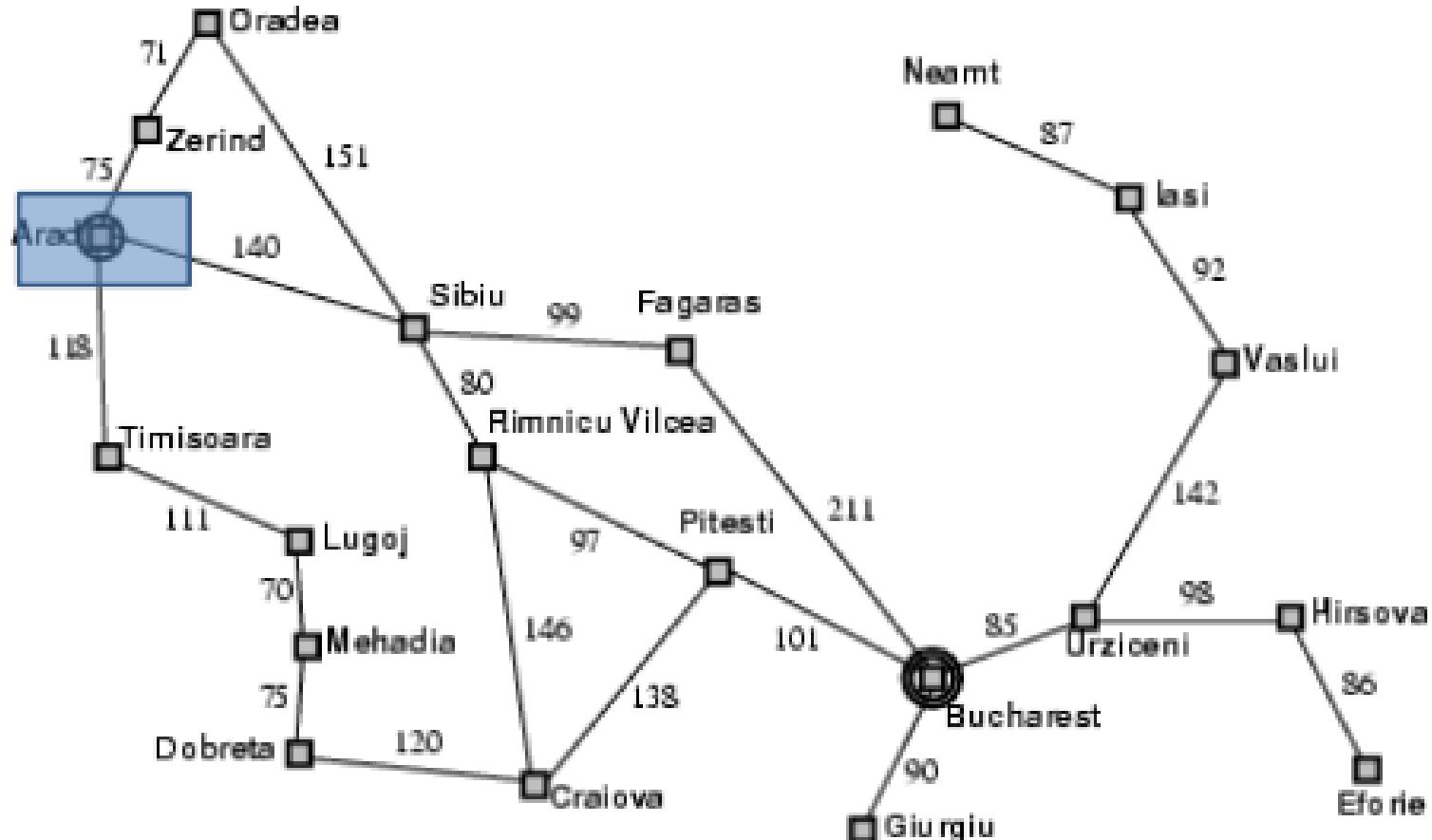
Travel from Arad to Bucharest

D= 0 D= 1 D= 2 D= 3



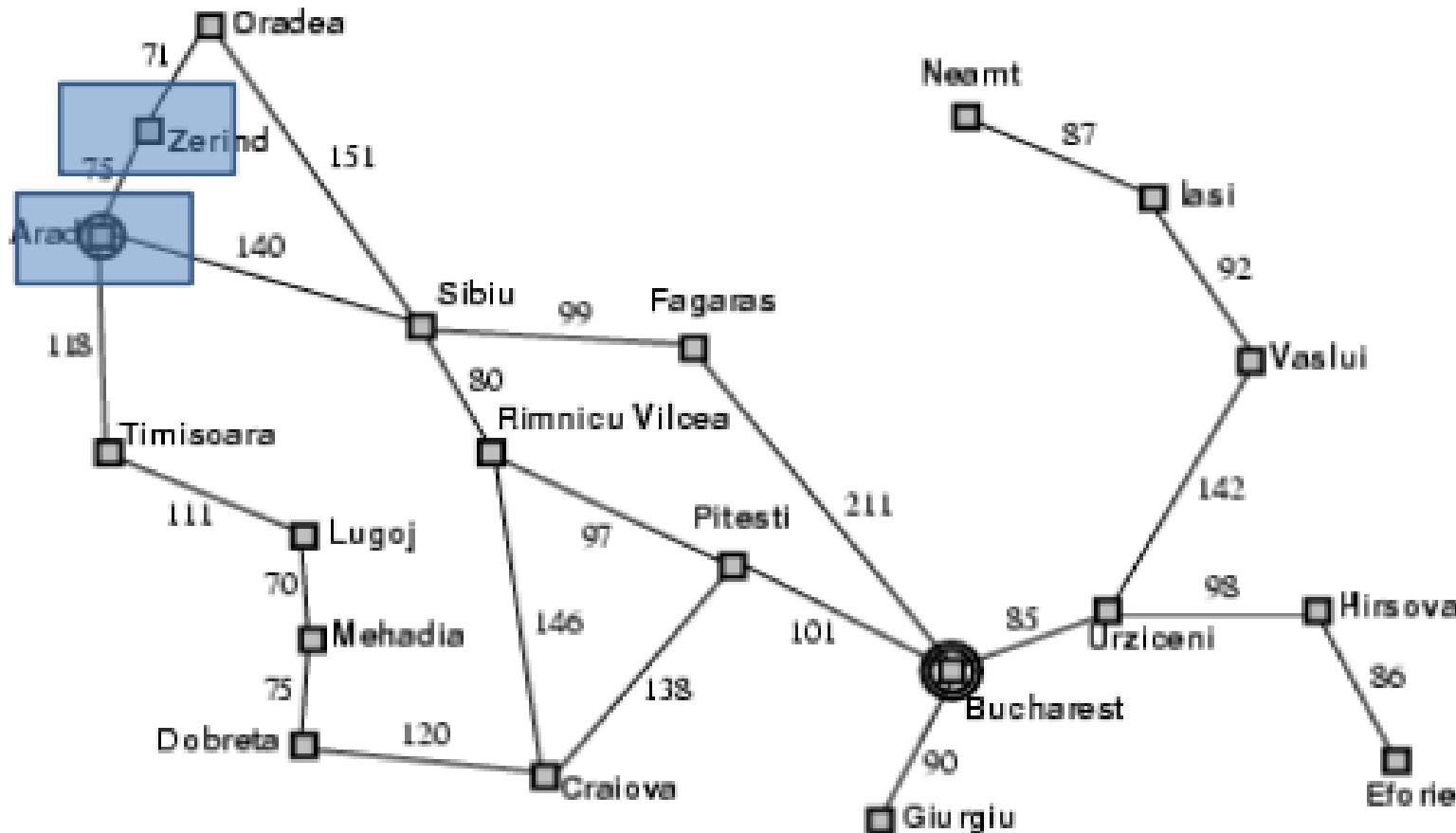
Example: Romania DFS

Travel from Arad to Bucharest



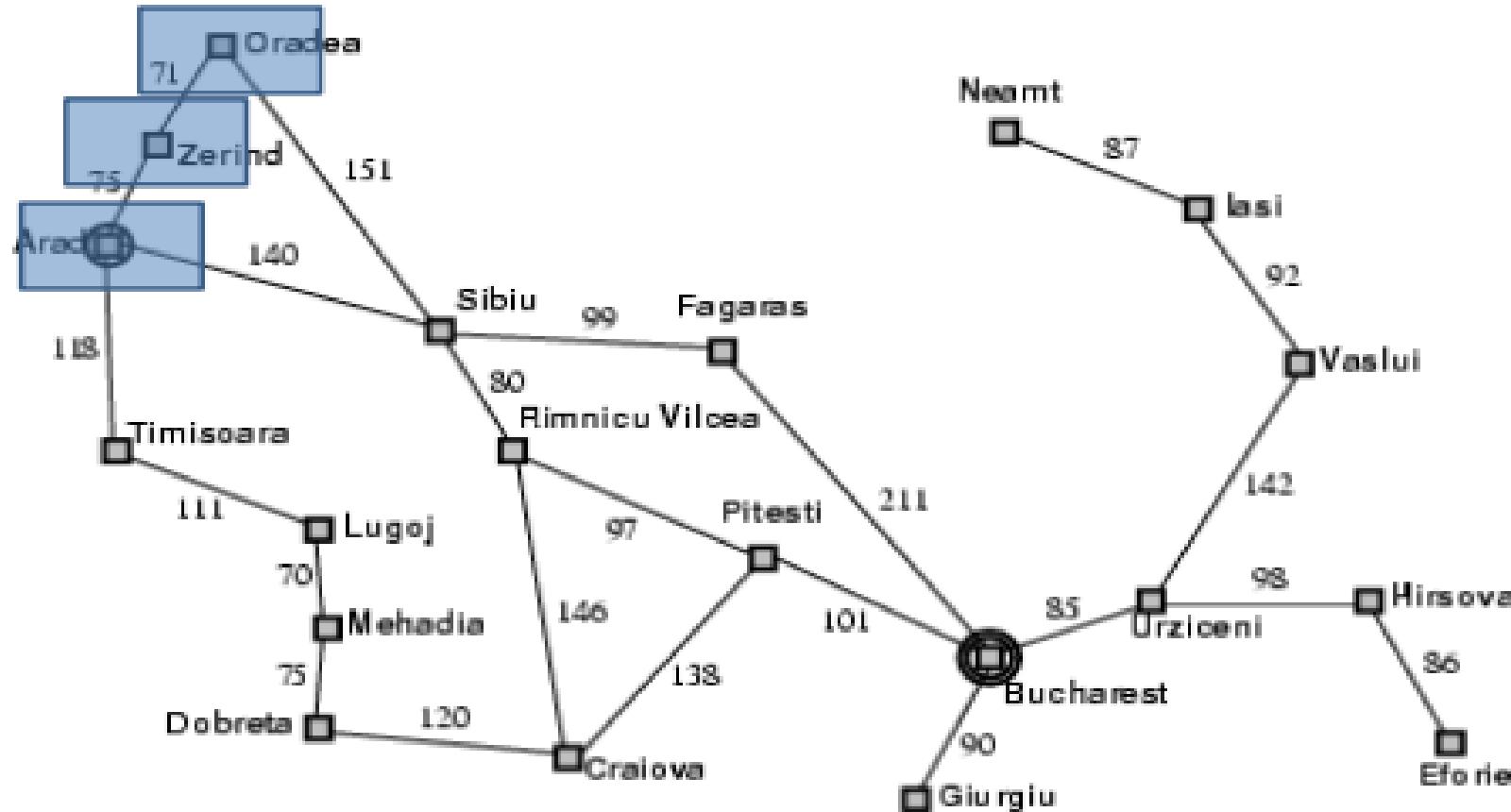
Example: Romania DFS

Travel from Arad to Bucharest



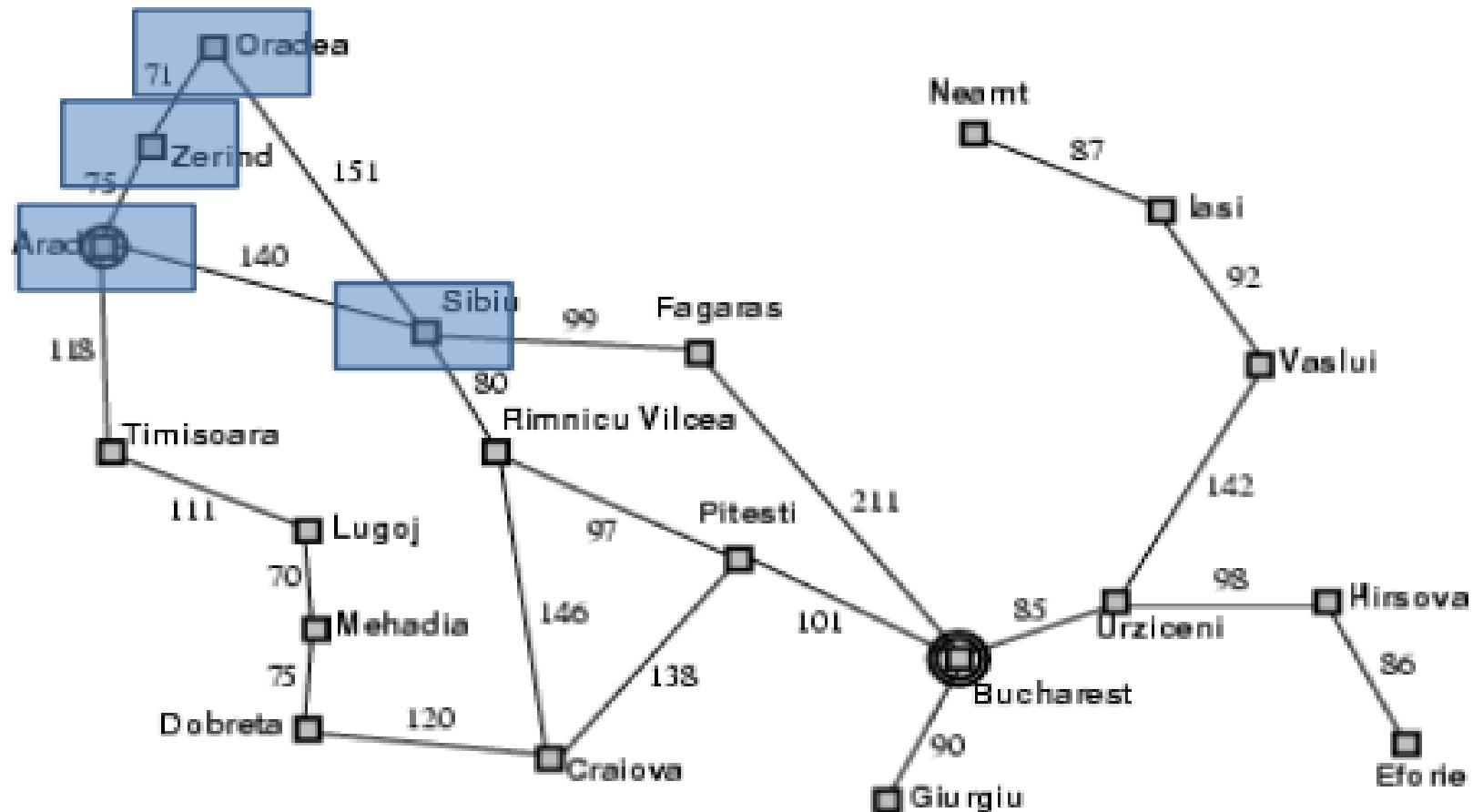
Example: Romania DFS

Travel from Arad to Bucharest



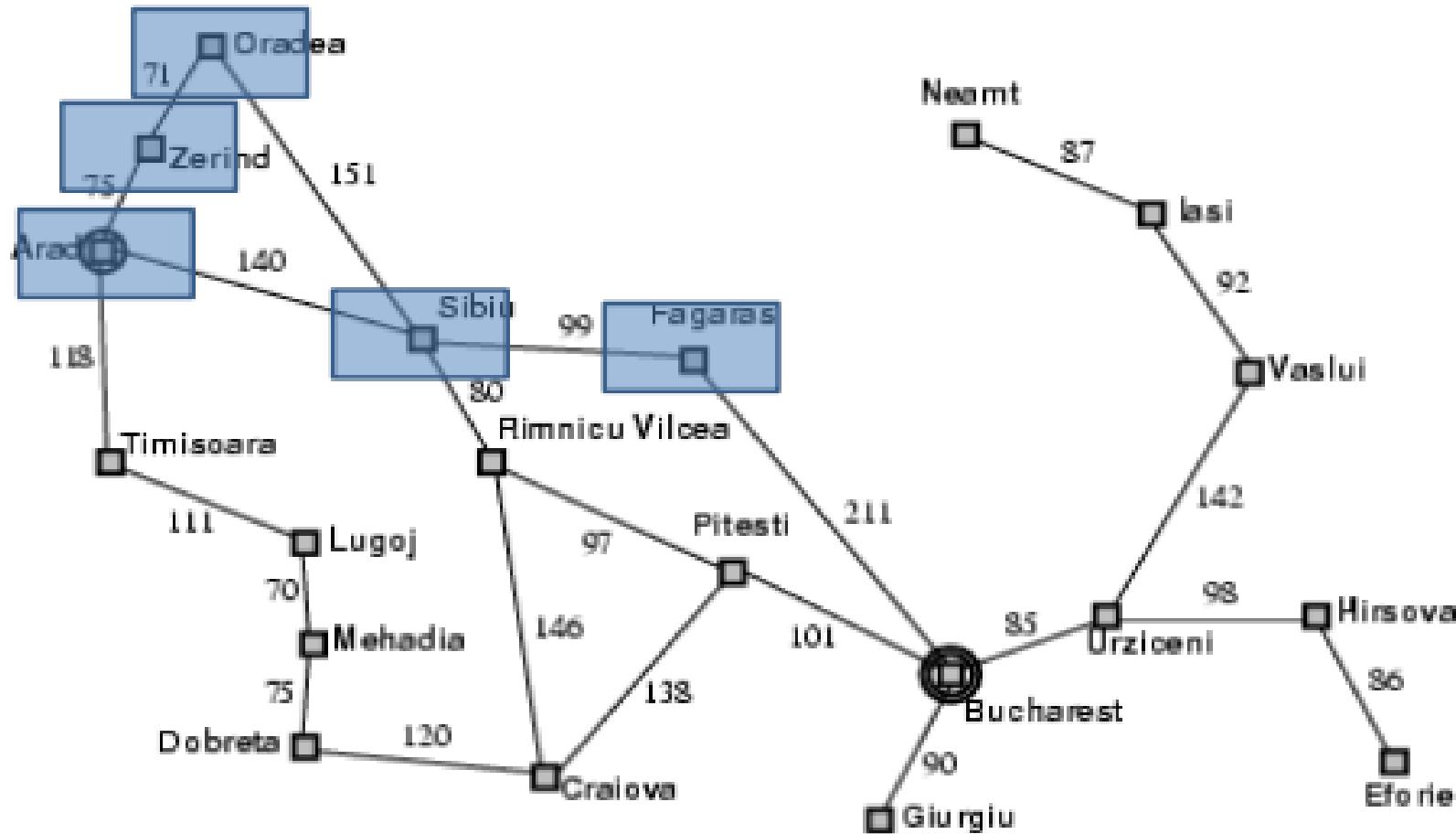
Example: Romania DFS

Travel from Arad to Bucharest



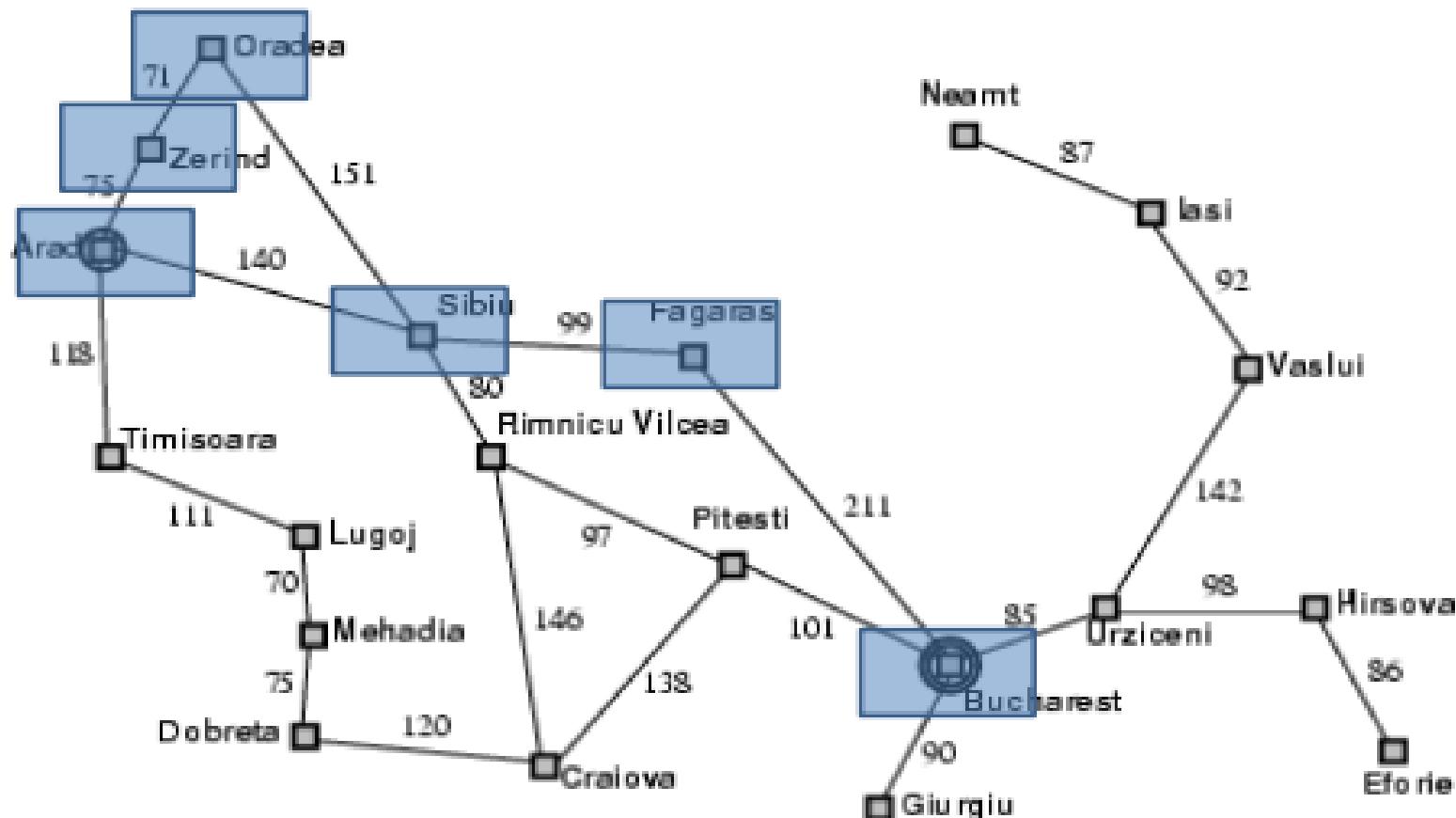
Example: Romania DFS

Travel from Arad to Bucharest



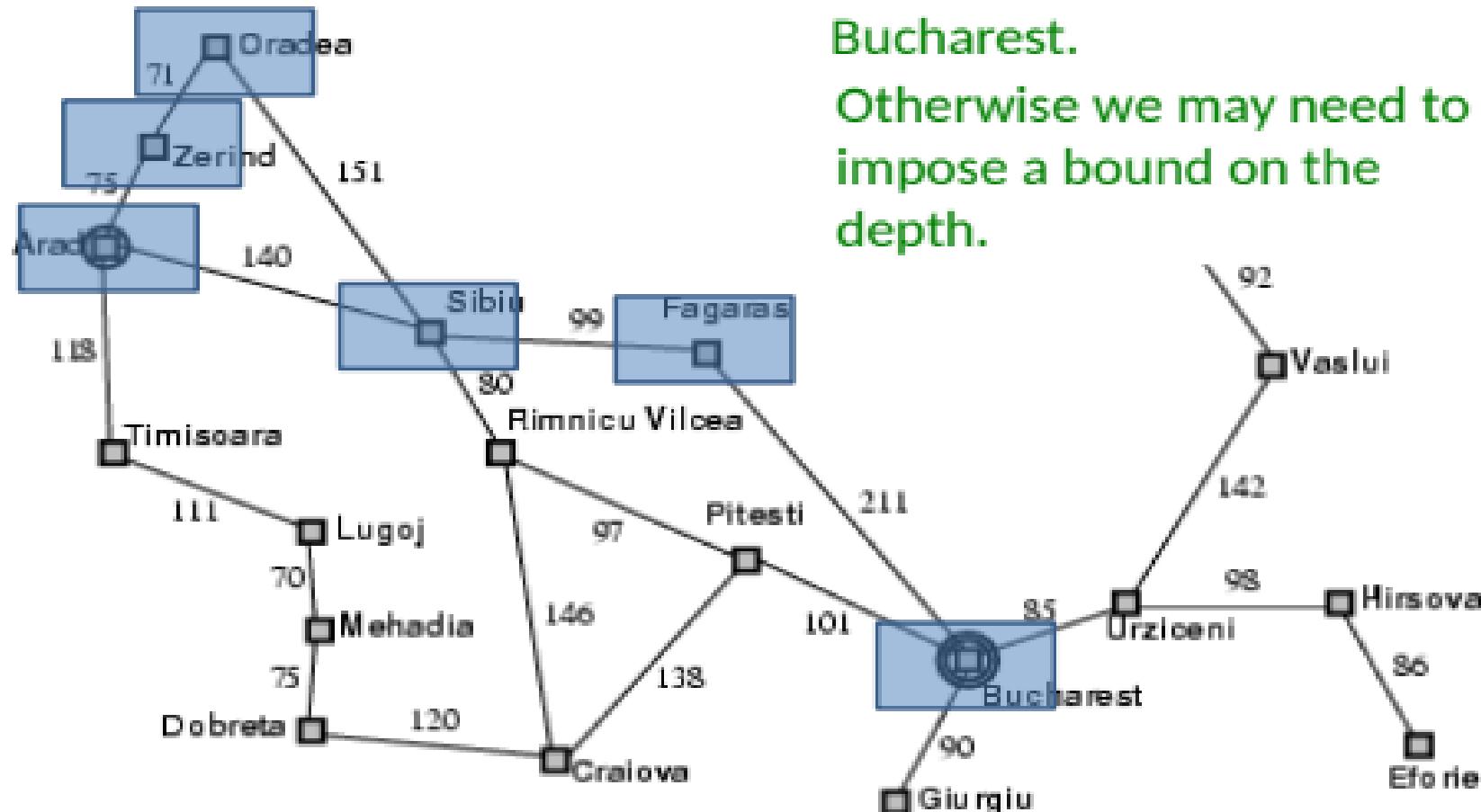
Example: Romania DFS

Travel from Arad to Bucharest



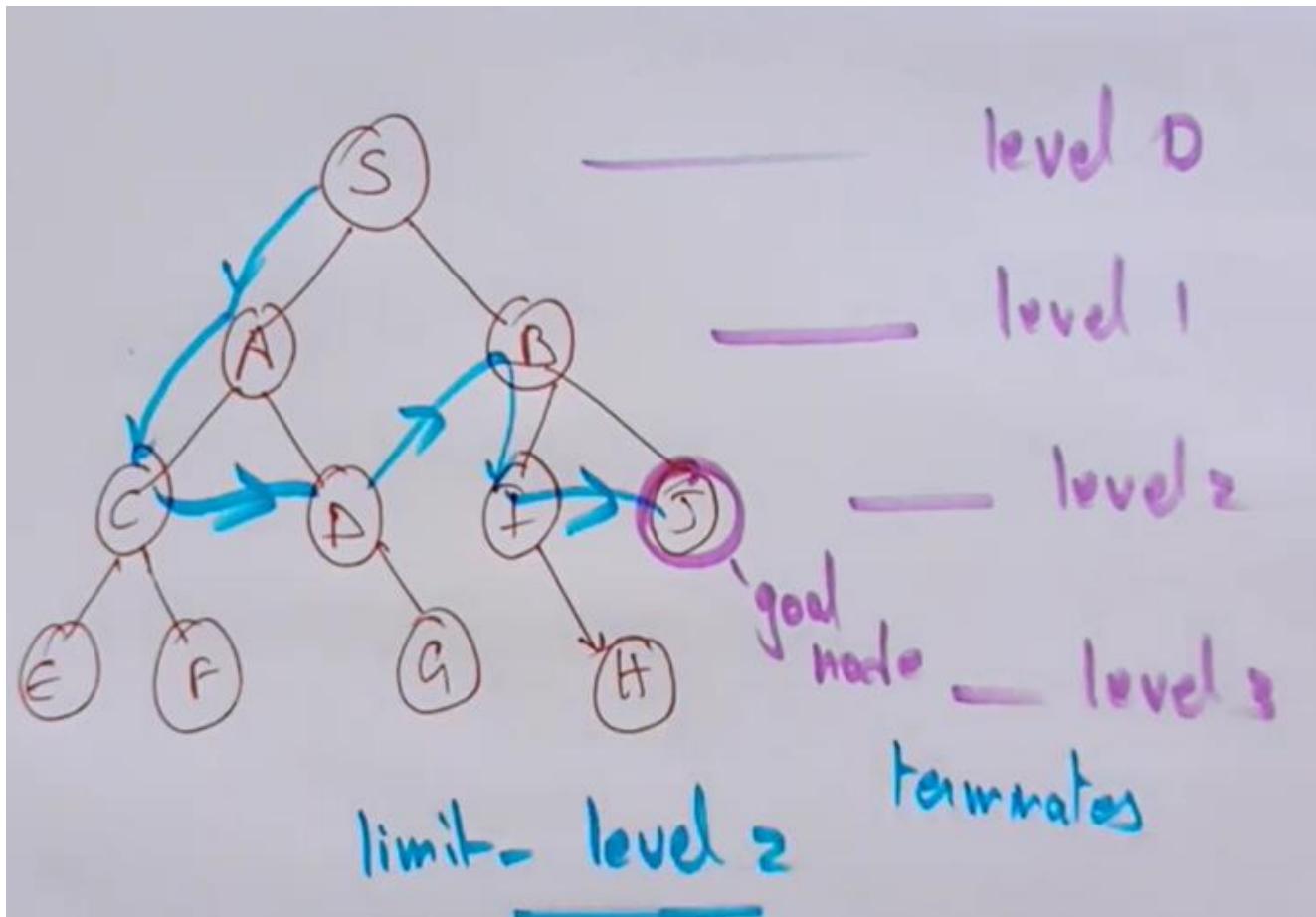
Example: Romania DFS

Travel from Arad to Bucharest



OK when all roads lead to Bucharest.
Otherwise we may need to impose a bound on the depth.

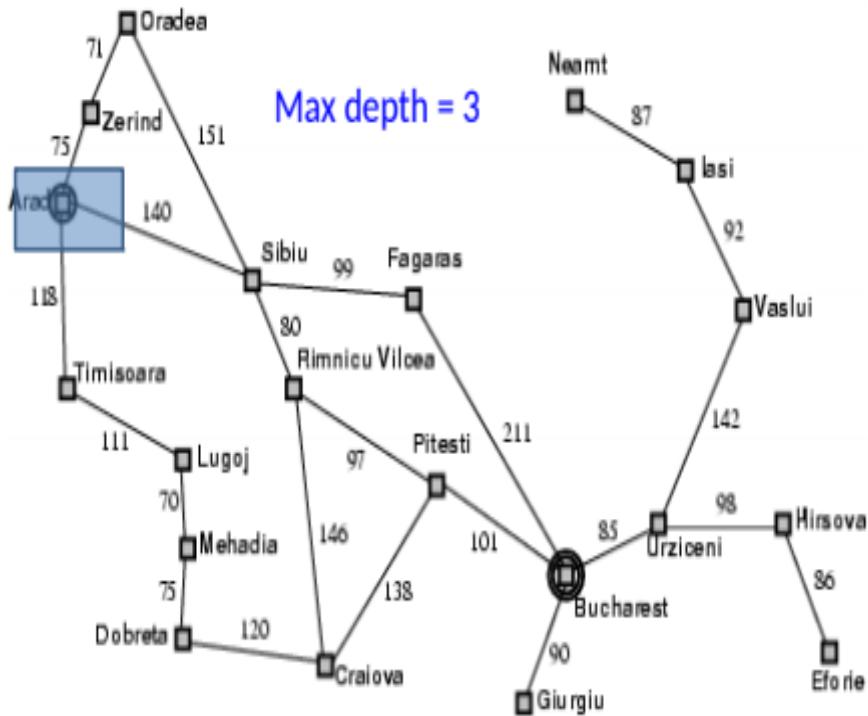
Depth Limited Search with limit=2



Depth Limited Search

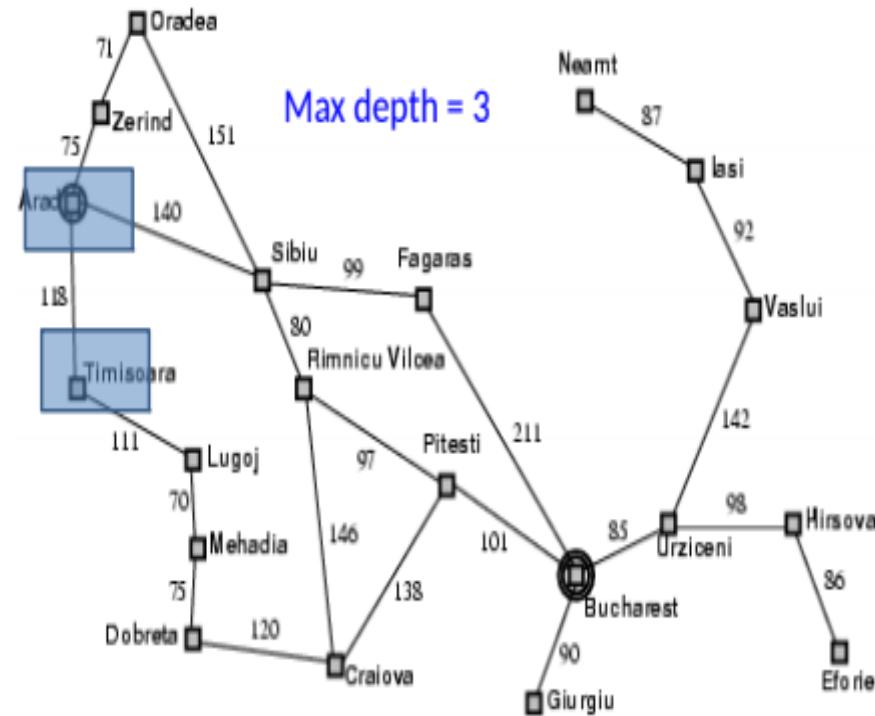
Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.
In fact, any city can reach any other in at most 9 steps.



Example: Romania Problem

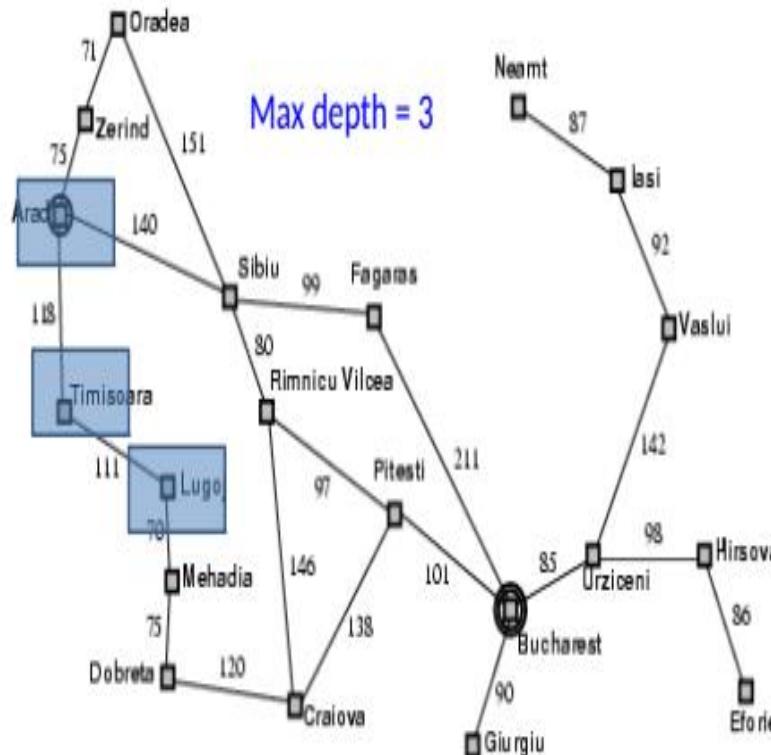
Only 20 cities on the map, so no path longer than 19.
In fact, any city can reach any other in at most 9 steps.



Depth Limited Search

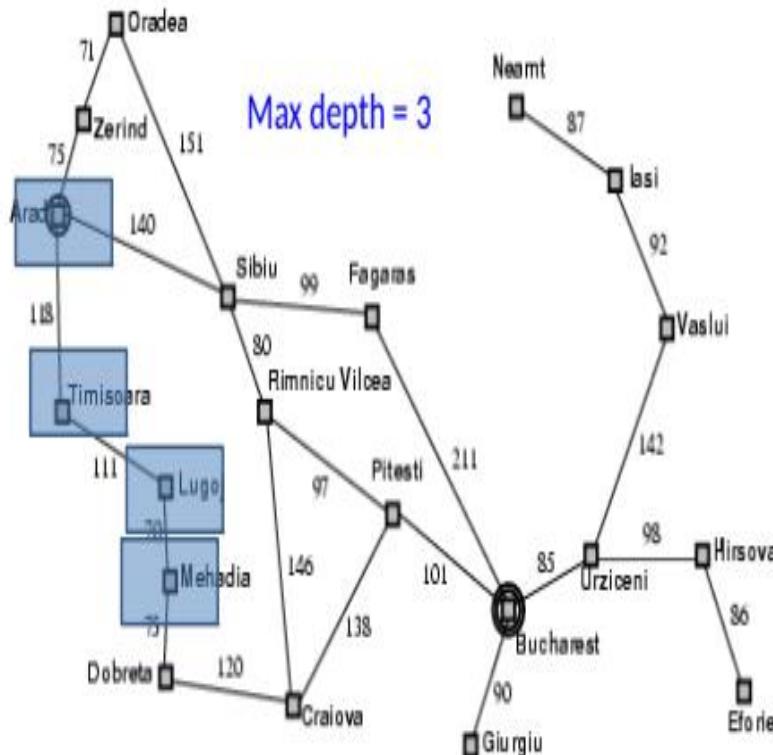
Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.
In fact, any city can reach any other in at most 9 steps.



Example: Romania Problem

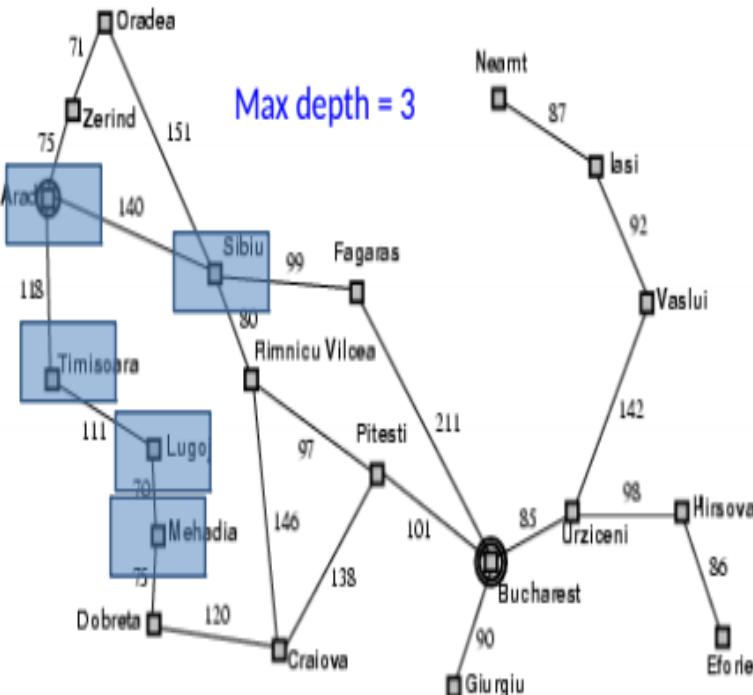
Only 20 cities on the map, so no path longer than 19.
In fact, any city can reach any other in at most 9 steps.



Depth Limited Search

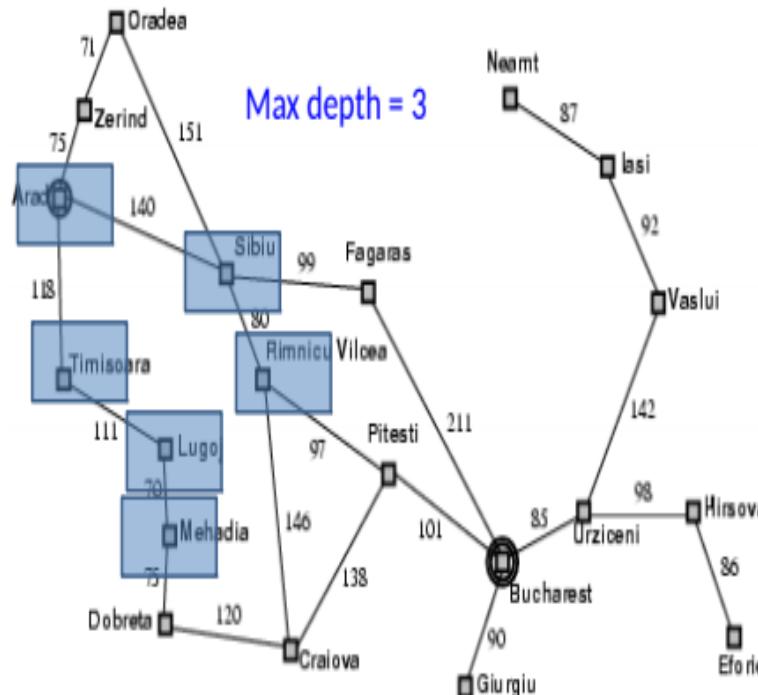
Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.
In fact, any city can reach any other in at most 9 steps.



Example: Romania Problem

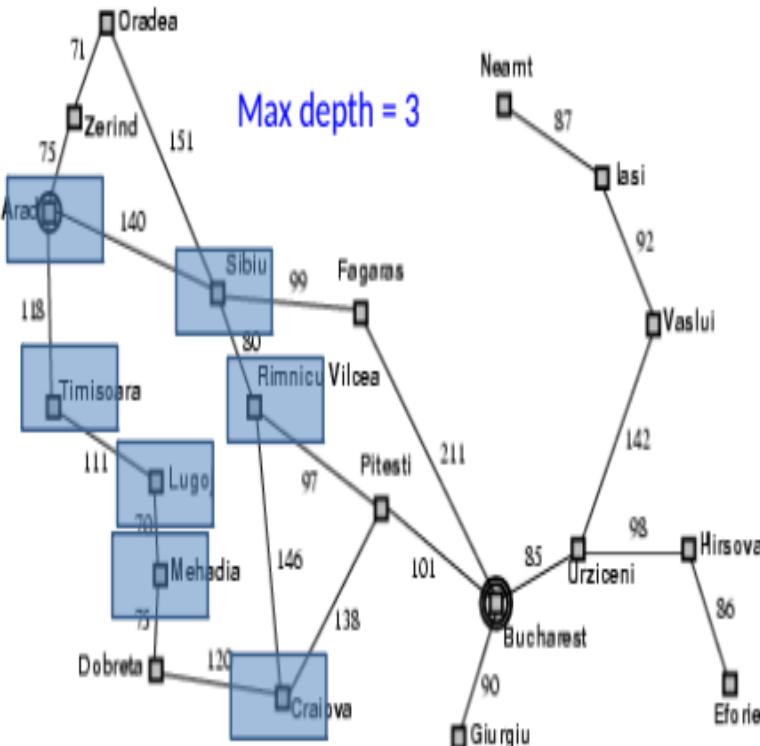
Only 20 cities on the map, so no path longer than 19.
In fact, any city can reach any other in at most 9 steps.



Depth Limited Search

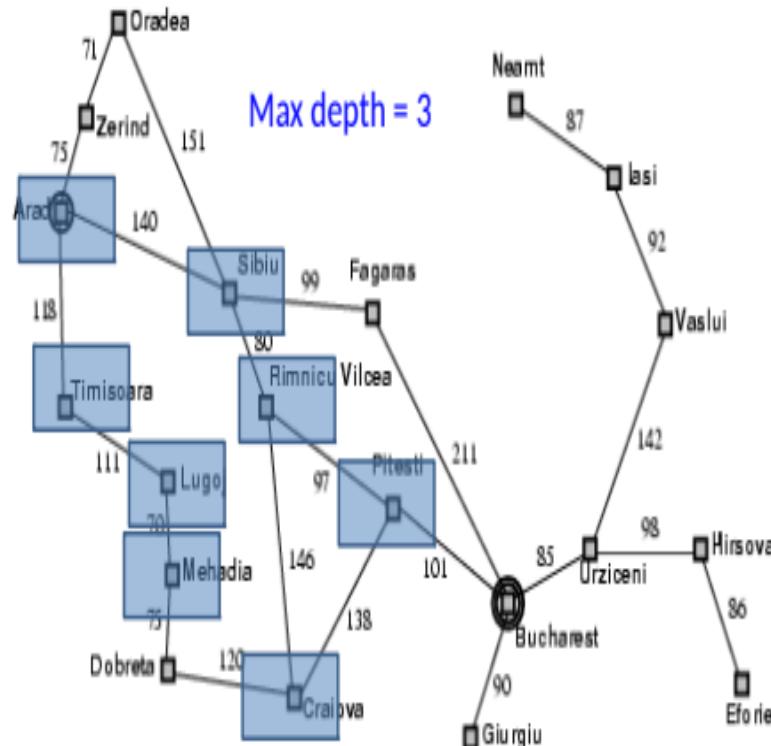
Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.
In fact, any city can reach any other in at most 9 steps.



Example: Romania Problem

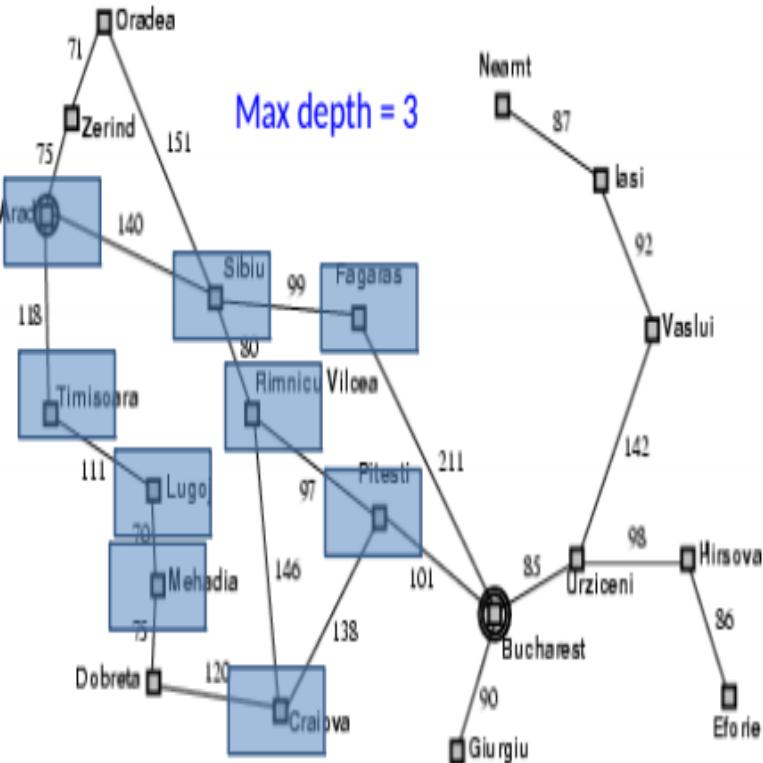
Only 20 cities on the map, so no path longer than 19.
In fact, any city can reach any other in at most 9 steps.



Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.

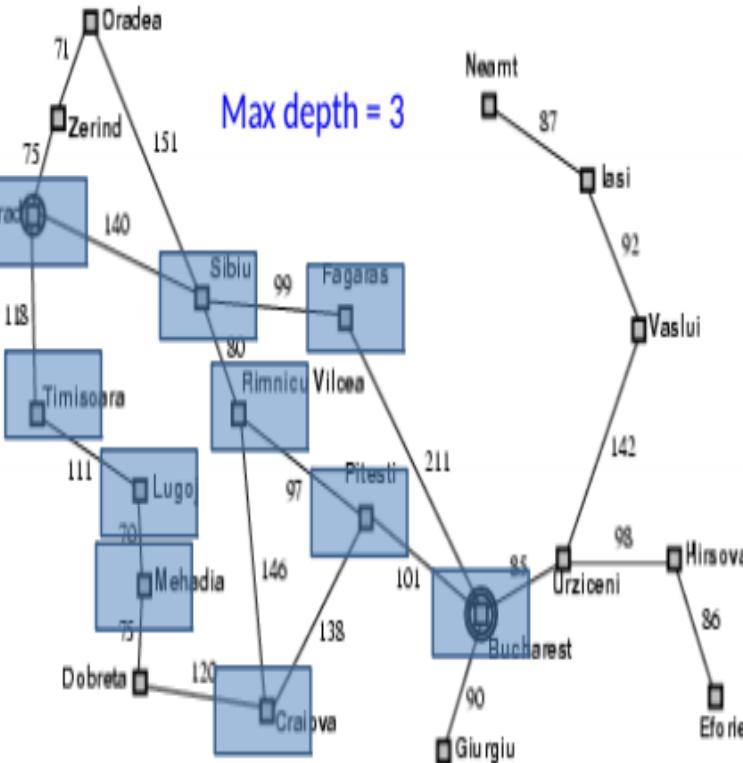
In fact, any city can reach any other in at most 9 steps.



Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.

In fact, any city can reach any other in at most 9 steps.

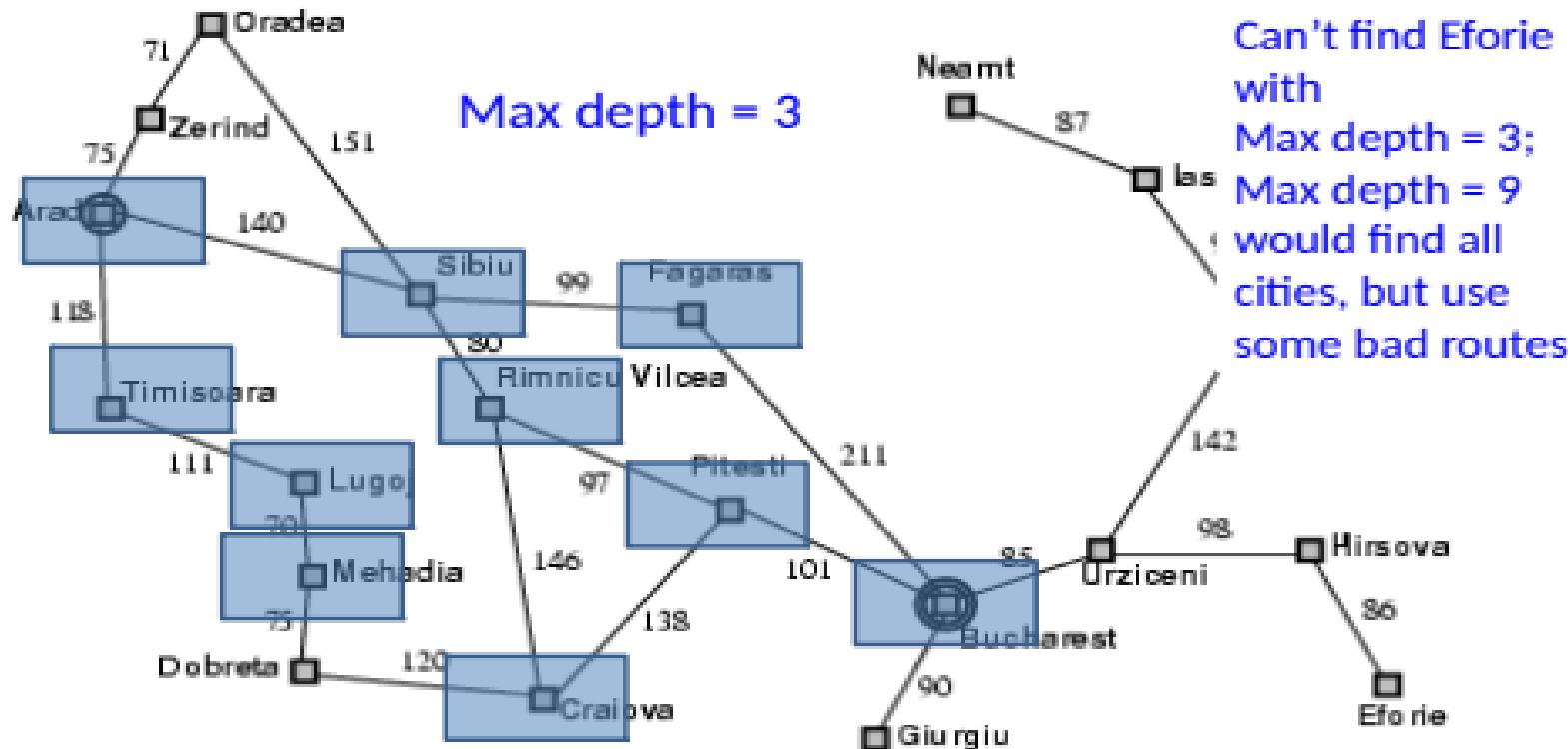


Depth Limited Search

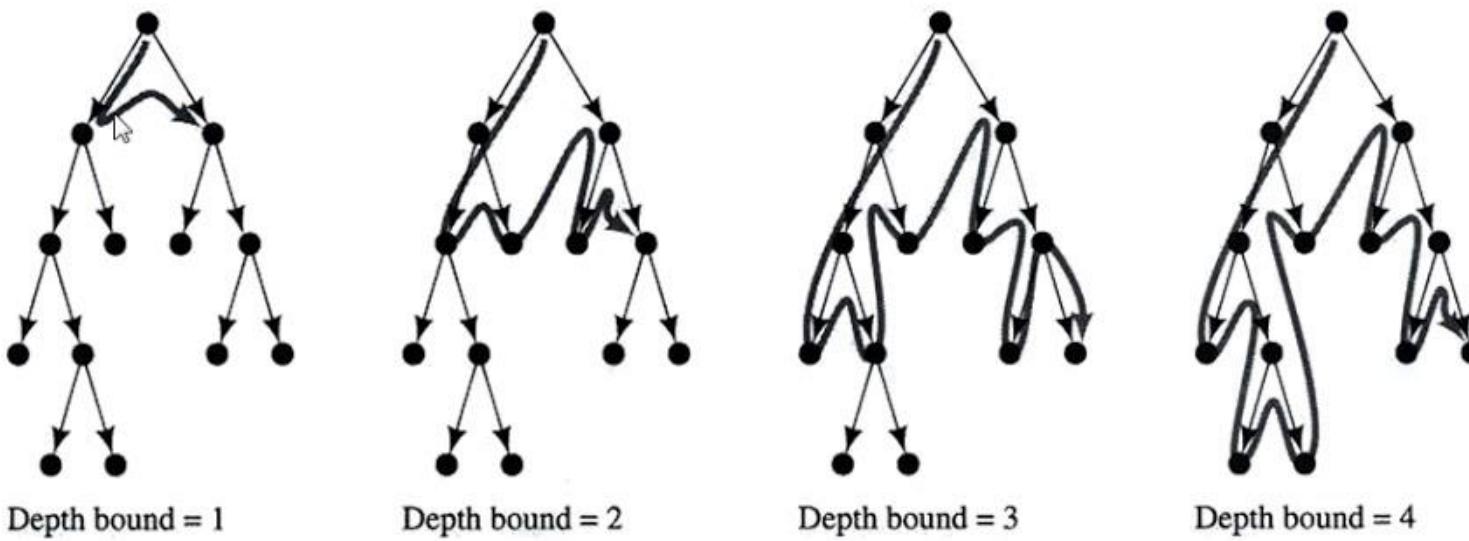
Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.

In fact, any city can reach any other in at most 9 steps.



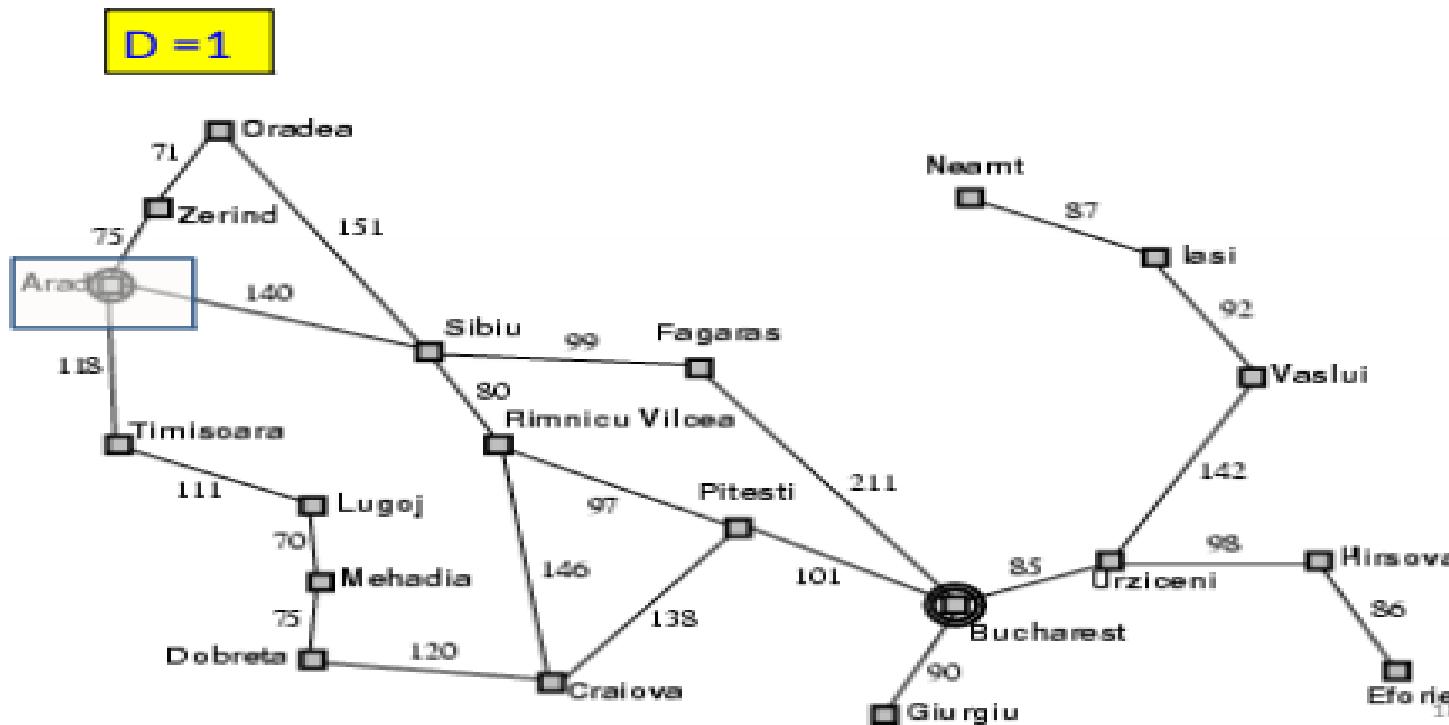
Example IDS



Stages in Iterative-Deepening Search

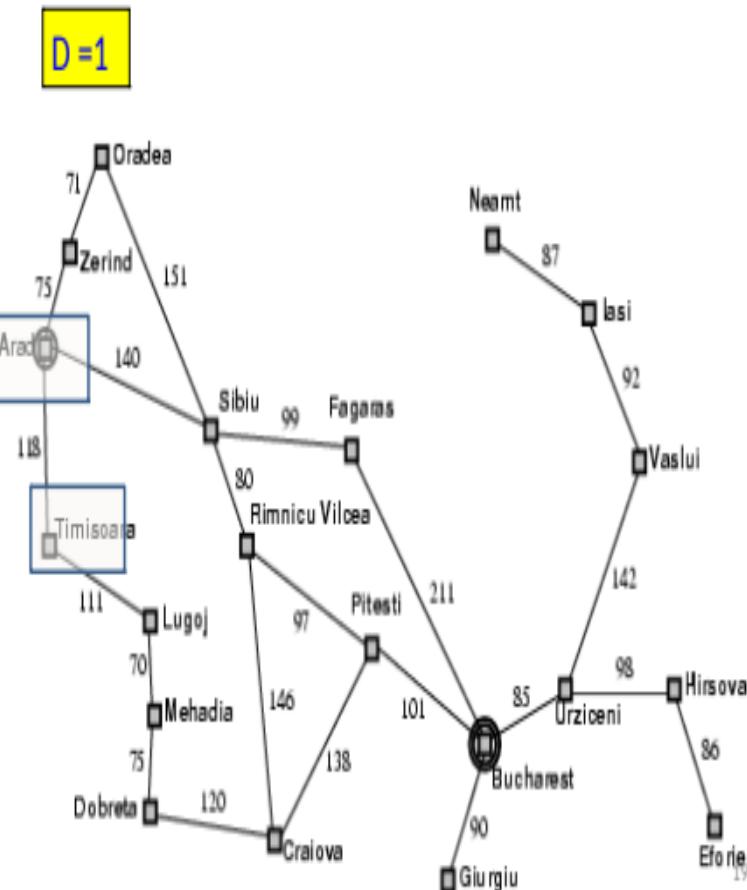
Iterative-deepening DFs

Example: Romania Problem

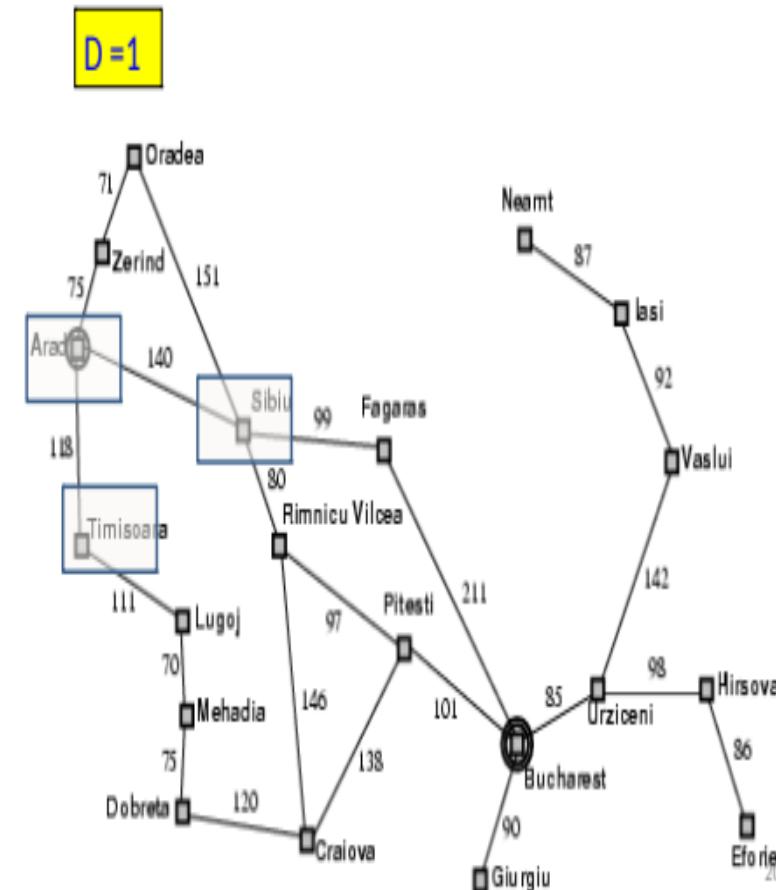


Iterative-deepening DFs

Example: Romania Problem



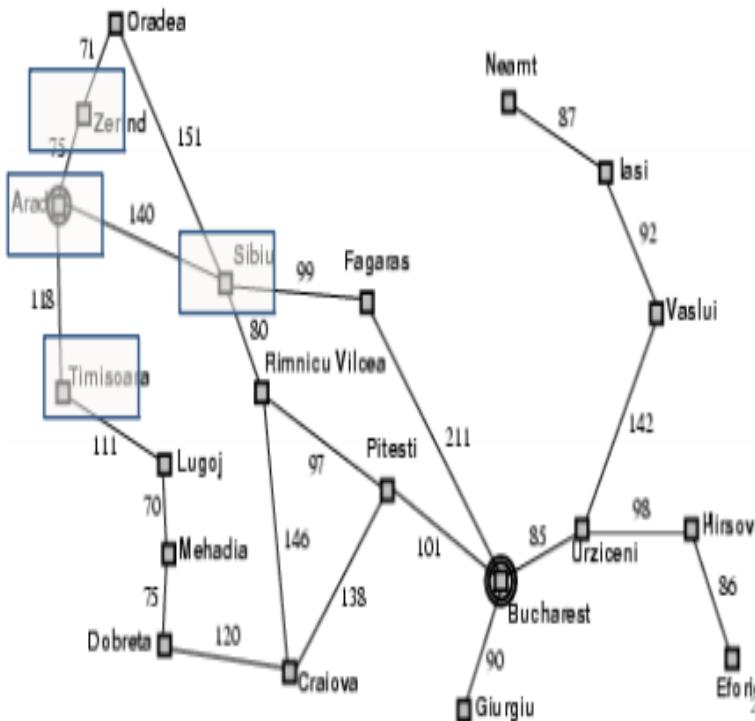
Example: Romania Problem



Iterative-deepening DFs

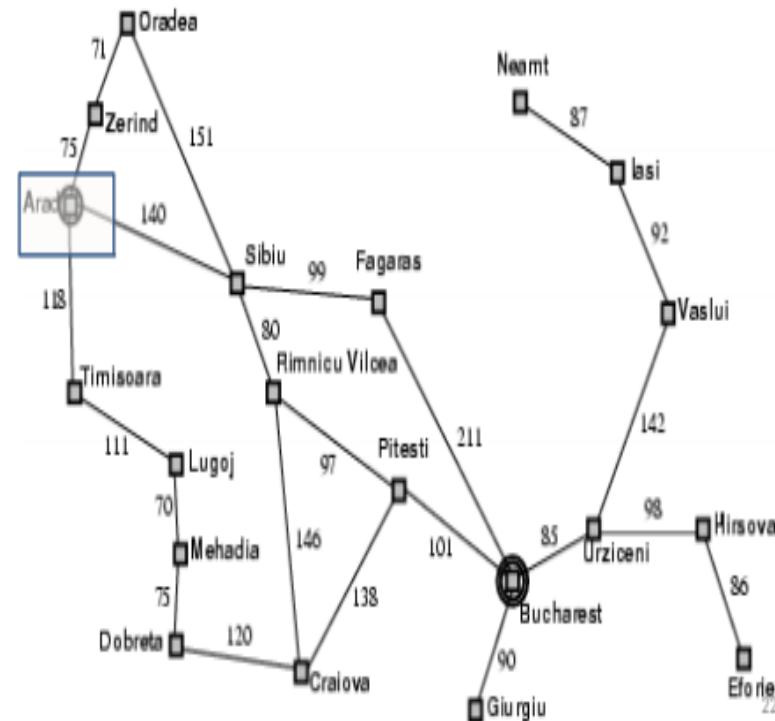
Example: Romania Problem

D=1



Example: Romania Problem

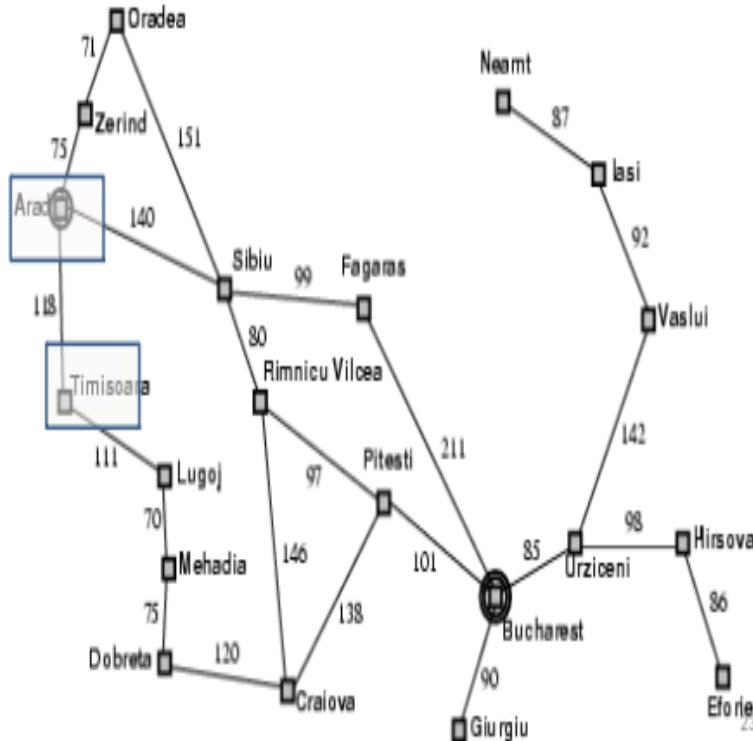
D=1 D=2



Iterative-deepening DFs

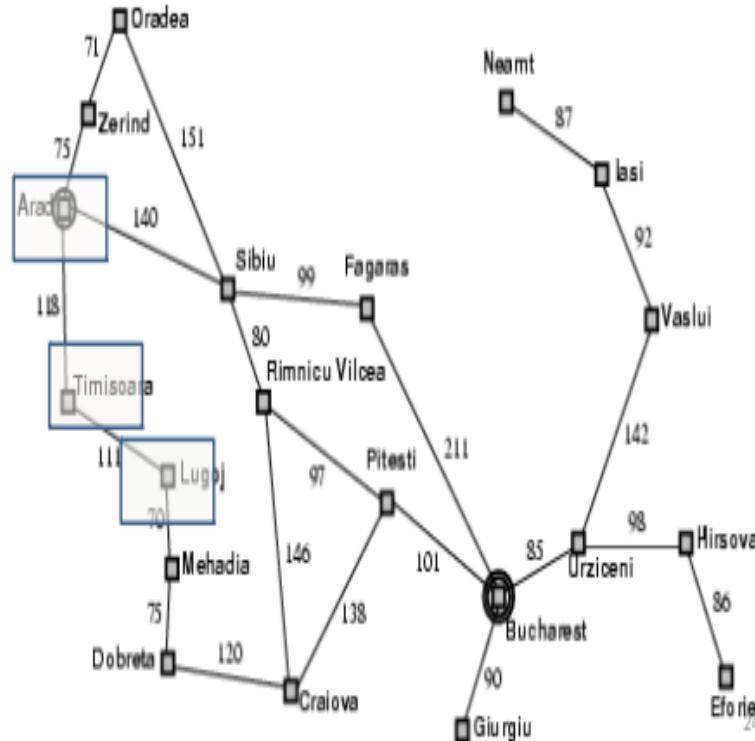
Example: Romania Problem

D=1 D=2



Example: Romania Problem

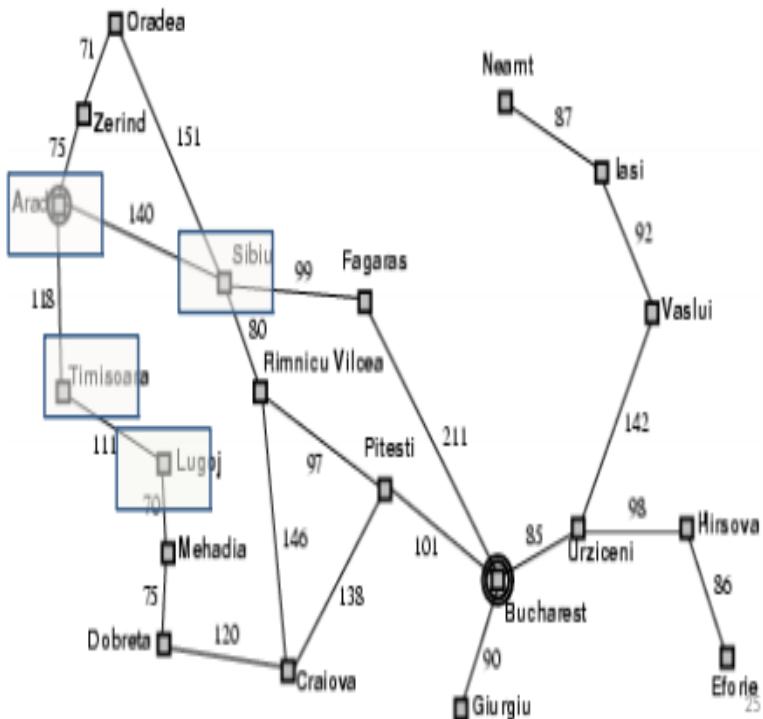
D=1 D=2



Iterative-deepening DFs

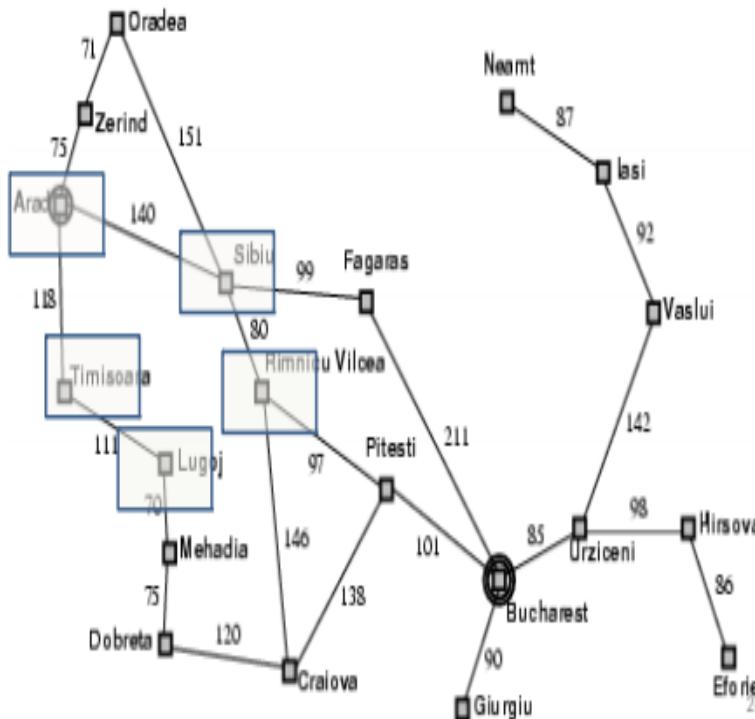
Example: Romania Problem

D=1 D=2



Example: Romania Problem

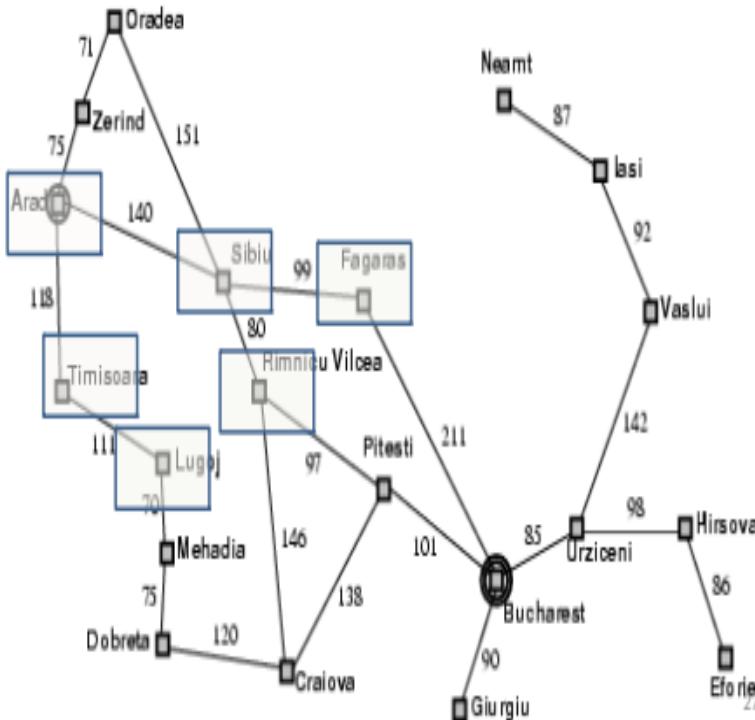
D=1 D=2



Iterative-deepening DFs

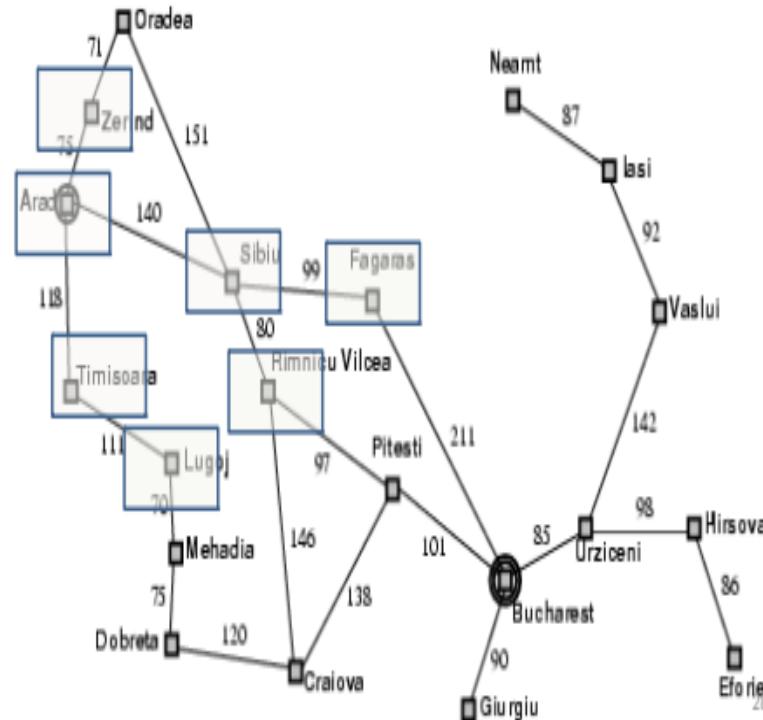
Example: Romania Problem

D=1 D=2



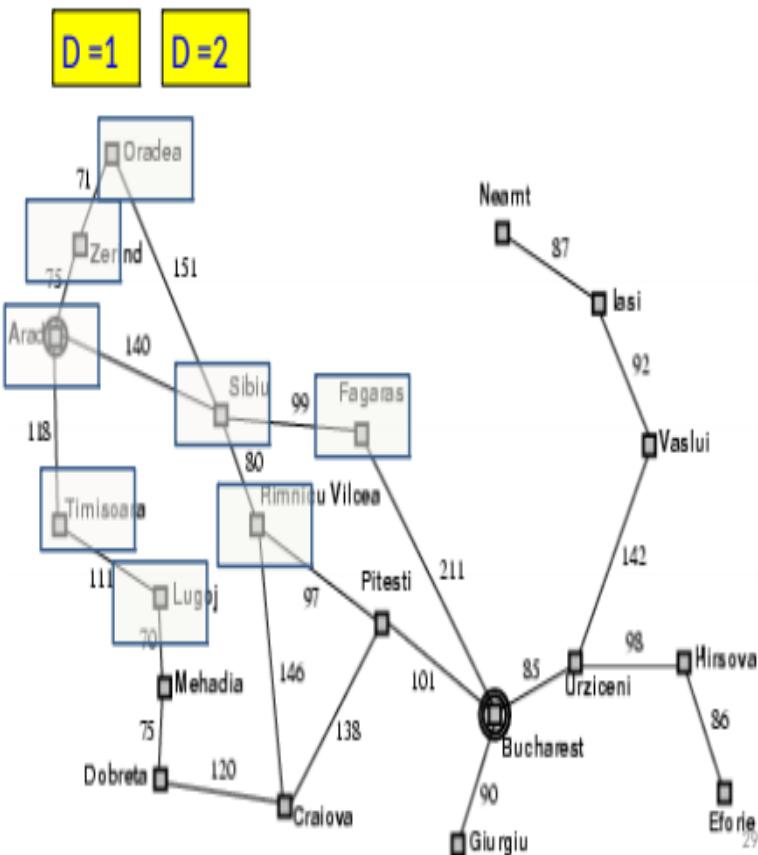
Example: Romania Problem

D=1 D=2

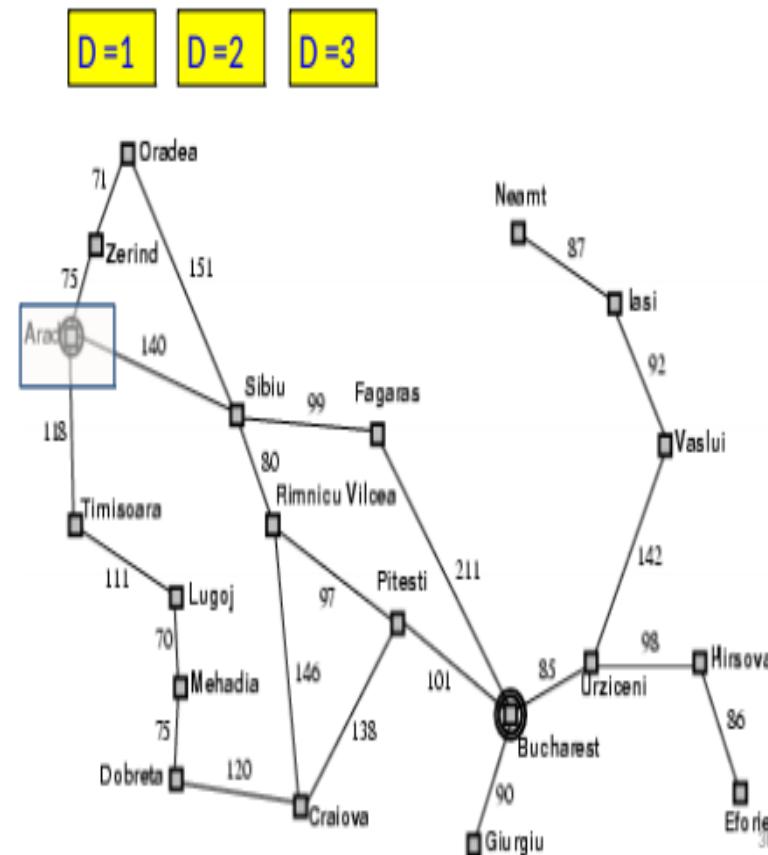


Iterative-deepening DFs

Example: Romania Problem

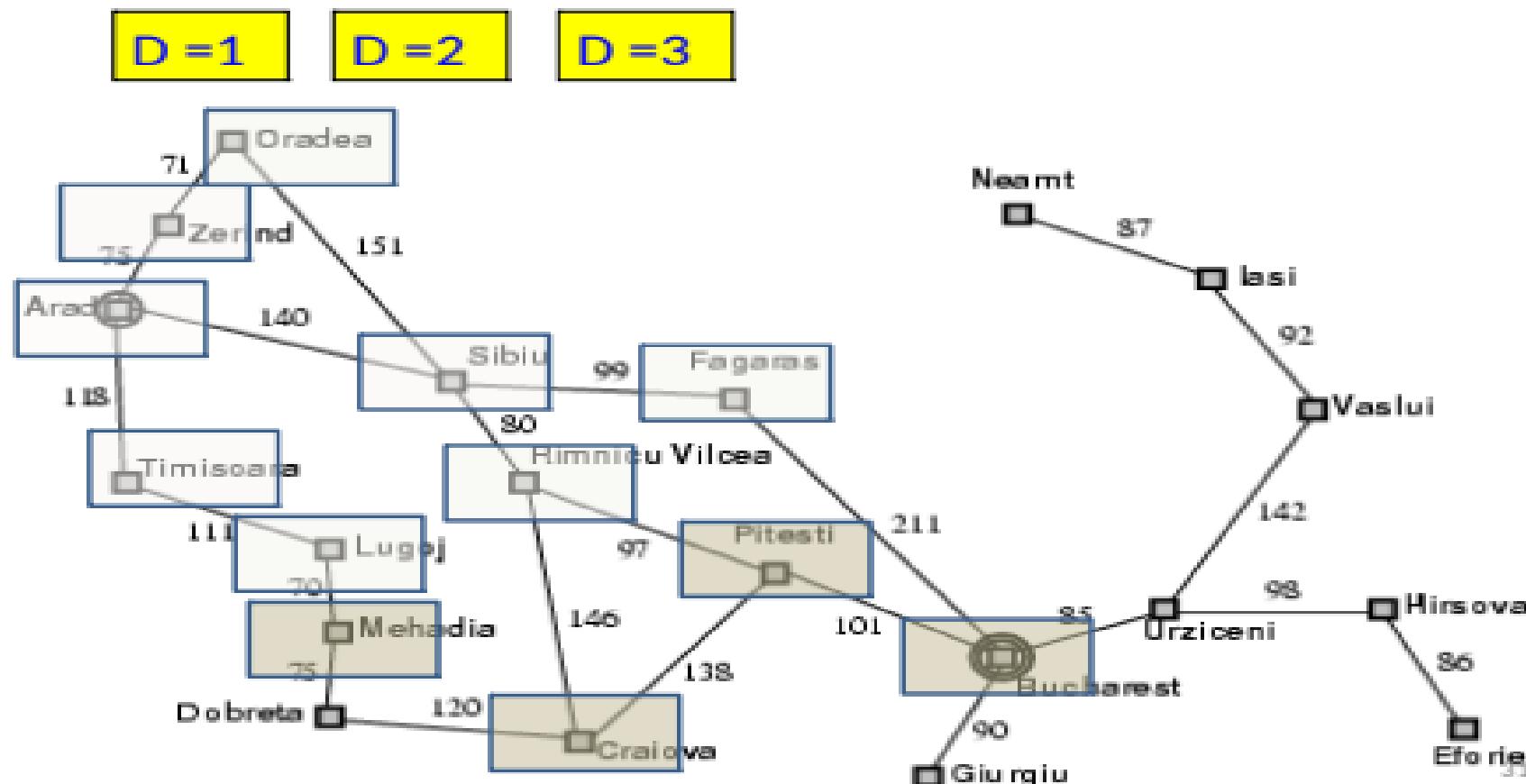


Example: Romania Problem

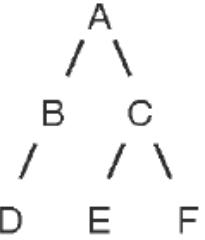
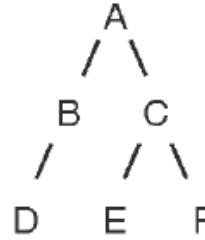


Iterative-Deepening DFS

Example: Romania Problem



Sr. No.	BFS	DFS
1.	BFS Stands for “Breadth First Search”.	DFS stands for “Depth First Search”.
2.	BFS traverses the tree level wise. i.e. each node near to root will be visited first. The nodes are explored left to right.	DFS traverses tree depth wise. i.e. nodes in particular branch are visited till the leaf node and then search continues branch by branch from left to right in the tree.
3.	Breadth First Search is implemented using queue which is FIFO list.	Depth First Search is implemented using Stack which is LIFO list.
4.	This is a single step algorithm, wherein the visited vertices are removed from the queue and then displayed at once.	This is two step algorithm. In first stage, the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped out.
5.	BFS requires more memory compare to DFS.	DFS require less memory compare to BFS.
6.	<p>Applications of BFS :</p> <ul style="list-style-type: none"> – To find Shortest path – Single Source and All pairs shortest paths – In Spanning tree – In Connectivity 	<p>Applications of DFS :</p> <ul style="list-style-type: none"> – Useful in Cycle detection – In Connectivity testing – Finding a path between V and W in the graph. – Useful in finding spanning trees and forest.
7.	BFS always provides the shallowest path solution.	DFS does not guarantee the shallowest path solution.
8.	No backtracking is required in BFS.	Backtracking is implemented in DFS.

Sr. No.	BFS	DFS
9.	BFS is optimal and complete if branching factor is finite.	DFS is neither complete nor optimal even in case of finite branching factor.
10.	BFS can never get trapped into infinite loops.	DFS generally gets trapped into infinite loops, as search trees are dense.
11.	Example :  <pre> graph TD A --- B A --- C B --- D B --- E C --- F </pre> <p>A, B, C, D, E, F</p>	Example :  <pre> graph TD A --- B A --- C B --- D B --- E C --- F </pre> <p>A, B, D, C, E, F</p>

Last Class Topics

Uninformed Search Techniques

DFS

BFS

DLS

Iterative Deepening DFS

Uniform Cost Search

Comparison of Uninformed Search

- DFS → The algorithm starts at the root node and explores depth wise as far as possible along each branch before backtracking.
- BFS → It starts at root, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.
- DLS->Depth-first with limit (depth below which nodes are not expanded)

Three possible outcomes:

- Solution
- Failure (no solution)
- **Cutoff (no solution within cutoff)**
- **Iterative deepening search** → It is the depth limited search in iteration. It does this by gradually increasing the limit from 0, then to 1 and then to 2 and so on. Iterative deepening DFS combines the benefits of DFS and BFS.
- **Uniform-cost search** → Expand first the nodes with lowest cost (instead of depth).

Informed Search Techniques

The main idea is to generate **additional information** about the search state space using the knowledge of problem domain, so that the search becomes more intelligent and efficient.

The evaluation function is developed for each state, which quantifies the desirability of expanding that state in order to reach the goal.

All the strategies use this evaluation function in order to select the next state under consideration, hence the name “Informed Search”. These techniques are very much efficient with respect to time and space requirements as compared to uninformed search techniques.

Heuristic Function

(MU - Dec. 12, Dec. 13, Dec. 14, May 15, Dec. 15)

- A heuristic function is **an evaluation function**, to which the search state is given as input and it generates the tangible representation of the state as output.
- It maps the problem state description to measures of desirability, usually represented as number weights. The value of a heuristic function at a given node in the search process gives a good estimate of that node being on the desired path to solution.
- It evaluates individual problem state and determines how much promising the state is.
- Heuristic functions are the most common way of imparting additional knowledge of the problem states to the search algorithm.

Heuristic Function

- The representation may be the approximate cost of the path from the goal node or number of hops required to reach to the goal node, etc.
- The heuristic function that we are considering , for a node n is, $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.
- **Example :** For the Travelling Salesman Problem, the sum of the distances traveled so far can be a simple heuristic function.
- Heuristic function can be of two types depending on the problem domain. It can be a **Maximization Function** or **Minimization function** of the path cost.
- In **maximization types of heuristic**, **greater the cost of the node, better is the node** while; in case of **minimization heuristic**, **lower is the cost, better** is the node. There are heuristics of every general applicability as well as domain specific. The search strategies are general purpose heuristics.

Heuristic Function

- It is believed that in general, a **heuristic will always lead to faster and better solution**, even though there is **no guarantee** that it will **never lead in the wrong direction** in the search tree.
- Design of heuristic plays a **vital role in performance of search**.
- As the purpose of a heuristic function is to guide the search process in the most profitable path among all that are available; a well designed heuristic functions can provides a fairly good estimate of whether a path is good or bad.
- However in many problems, **the cost of computing the value of a heuristic function** would be more than the effort saved in the search process. Hence generally there is a **trade-off between the cost of evaluating a heuristic function and the savings in search that the function provides**.

Properties of Good Heuristic Function

1. It should generate a unique value for each unique state in search space.
2. The values should be a logical indicator of the profitability of the state in order to reach the goal state.
3. It may not guarantee to find the best solution, but almost always should find a very good solution.
4. It should reduce the search time; specifically for hard problems like travelling salesman problem where the time required is exponential.

Example of 8-Puzzle Problem

Initial State	Goal State																		
<table border="1"><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>8</td><td></td><td>4</td></tr><tr><td>7</td><td>6</td><td>5</td></tr></table>	1	2	3	8		4	7	6	5	<table border="1"><tr><td>2</td><td>8</td><td>1</td></tr><tr><td></td><td>4</td><td>3</td></tr><tr><td>7</td><td>6</td><td>5</td></tr></table>	2	8	1		4	3	7	6	5
1	2	3																	
8		4																	
7	6	5																	
2	8	1																	
	4	3																	
7	6	5																	

Two simple heuristic functions are :

h_1 = the number of misplaced tiles. This is also known as the **Hamming Distance**. The start state has $h_1 = 5$.

h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot be moved diagonally, the distance counted is the sum of horizontal and vertical distances. This is also known as the **Manhattan Distance**.

The start state has $h_2 = 2+1+1+2+1+0+0+0=7$

Initial

1	2	3
	4	6
7	5	8

Goal

1	2	3
4	5	6
7	8	

$$h = 3$$

misplaced tiles.

1	2	3
7	4	6
7	5	8

$$h = 4$$

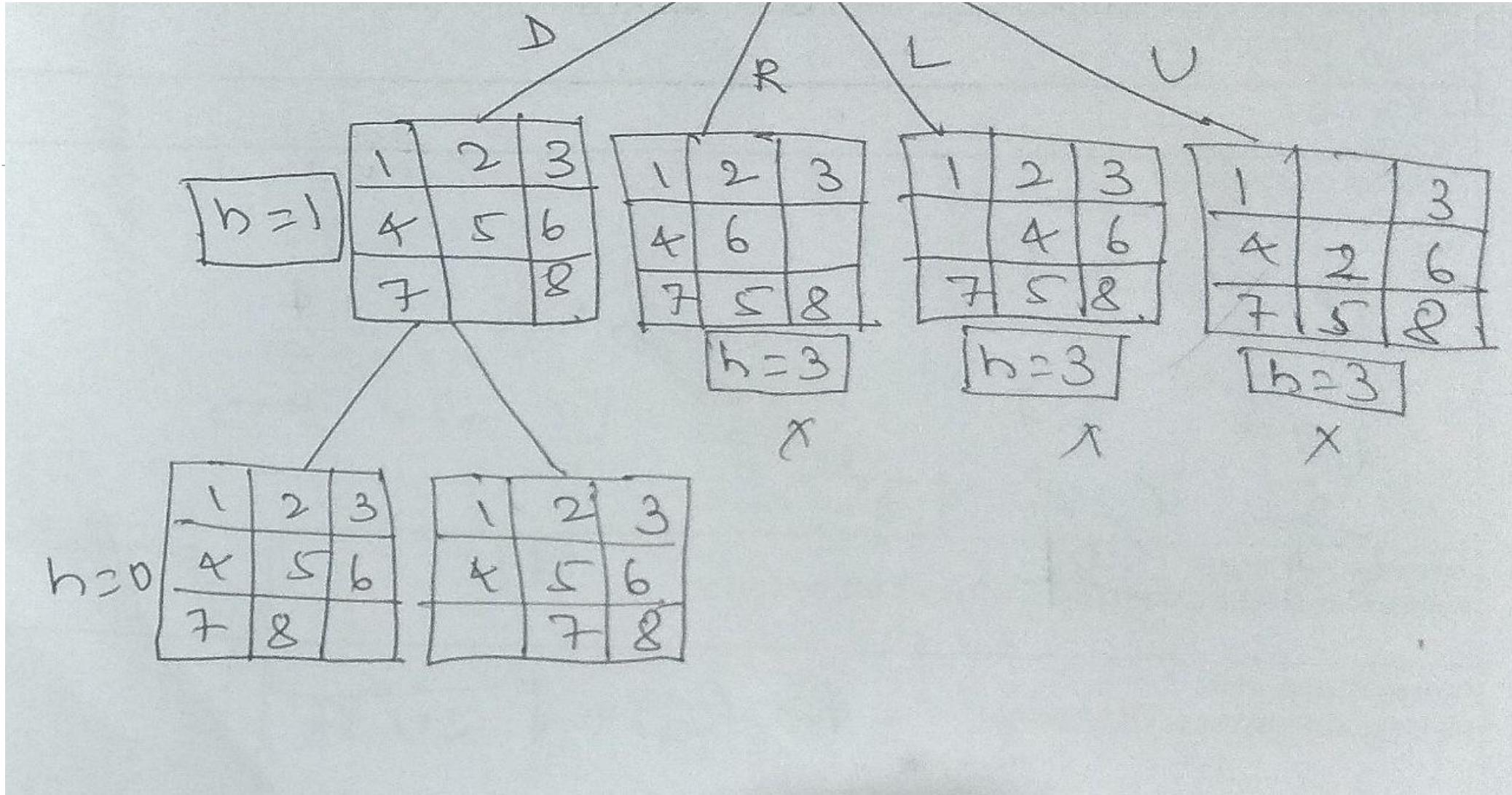
$$h = 4$$

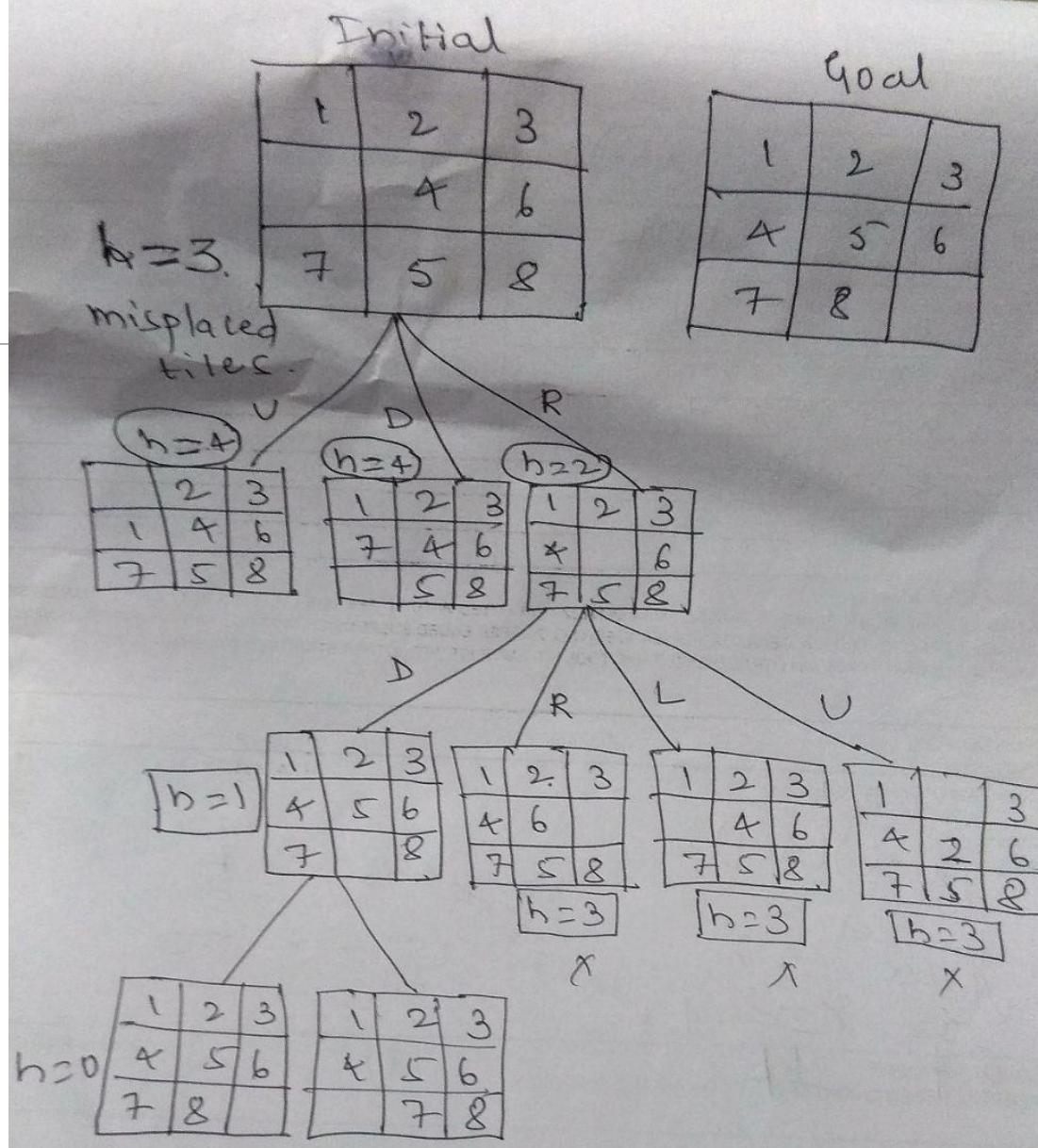
D

1	2	3
7	4	6
5	8	

$$h = 2$$

1	2	3
*		6
7	5	8





Last class Topic

Informed Search

Heuristic Function

Maximization Heuristic Function

Minimization Heuristic Function

Example of heuristic Functions

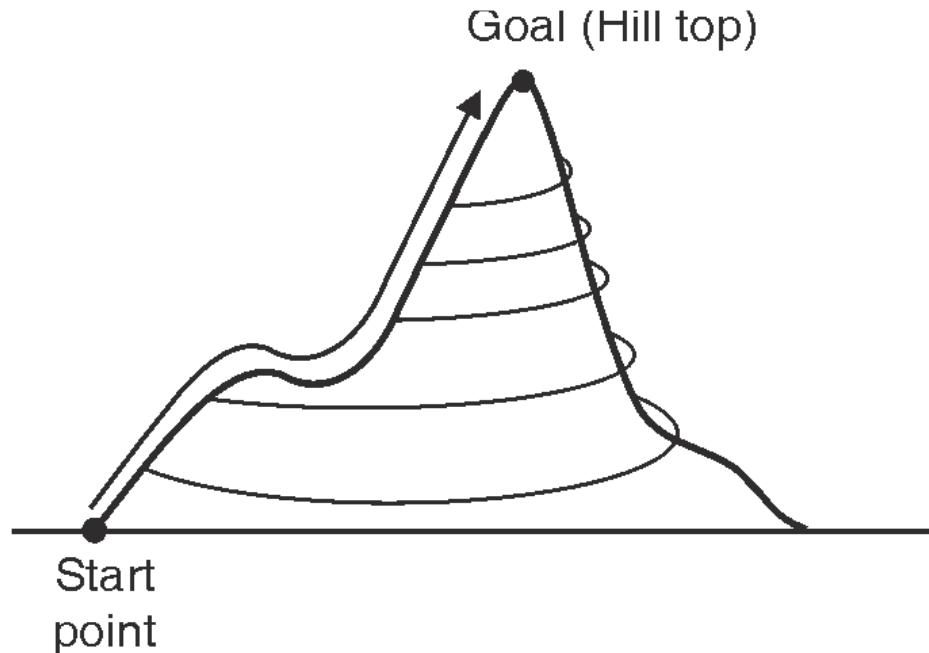
Hamming Distance

Manhattan Distance.

Hill- Climbing

In Hill climbing, each state is provided with the additional information needed to find the solution, i.e. **the heuristic value**. The algorithm is memory efficient since it does not maintain the complete search tree. Rather, it looks only at the current state and immediate level states.

Hill climbing attempts to iteratively improve the current state by means of an evaluation function. “Consider all the possible states laid out on the surface of a landscape. The height of any point on the landscape corresponds to the evaluation function of the state at that point”



Hill- Climbing

Hill climbing always attempts to make changes that improve the current state. In other words, hill climbing can only advance if there is a higher point in the adjacent landscape.

Hill climbing is a type of local search technique. It is relatively simple to implement. In many cases where state space is of moderate size, hill climbing works even better than many advanced techniques.

There are two variations of hill climbing as discussed follow.

(A) Simple Hill Climbing

(B) Steepest Ascent Hill Climbing

Simple Hill Climbing

Algorithm

1. Evaluate the initial state. If it is a goal state, then return and quit; otherwise make it a current state and go to Step 2.
2. Loop until a solution is found or there are no new operators left to be applied (i.e. no new children nodes left to be explored).
 - a. Select and apply a new operator (i.e. generate new child node)
 - b. Evaluate the new state :
 - (i) If it is a goal state, then return and quit.
 - (ii) If it is better than current state then make it a new current state.
 - (iii) If it is not better than the current state then continue the loop, go to Step 2.

Simple Hill Climbing

As we study the algorithm, we observe that in every pass the first node / state that is better than the current state is considered for further exploration. This strategy may not guarantee that most optimal solution to the problem, but may save upon the execution time.

Steepest Ascent Hill Climbing

As the name suggests, steepest hill climbing always finds the steepest path to hill top. It does so by selecting the best node among all children of the current node / state. All the states are evaluated using heuristic function. Obviously, the time requirement of this strategy is more as compared to the previous one.

Algorithm

1. Evaluate the initial state, if it is a goal state, return and quit; otherwise make it as a current state.
2. Loop until a solution is found or a complete iteration produces no change to current state :
 - a. SUCC = a state such that any possible successor of the current state will be better than SUCC.
 - b. For each operator that applies to the current state, evaluate the new state:
 - (i) If it is goal; then return and quit
 - (ii) If it is better than SUCC then set SUCC to this state.
 - c. SUCC is better than the current state \Rightarrow set the current state to SUCC.

Comparison...

As we compare simple hill climbing with steepest ascent, we find that there is a trade-off for the time requirement and the accuracy or optimality of the solution.

In case of **simple hill climbing technique** as we go for **first better successor**, the time is saved as **all the successors are not evaluated** but it may lead to more number of nodes and branches getting explored, in turn the solution found may **not be the optimal one**.

While in case of steepest ascent hill climbing technique, as **every time the best among all the successors is selected for further expansion**, it involves more time in evaluating all the successors at earlier stages, but the solution found will be always the optimal solution, as only the states leading to hill top are explored.

This also makes it clear that the evaluation function i.e. the **heuristic function definition plays a vital role** in deciding the performance of the algorithm.

Limitations of Hill Climbing

MU - May 13, May 14, Dec. 14, May 15

Local maximum : A “local maximum” is a location in hill which is at height from other parts of the hill but is not the actual hill top. In the search tree, it is a state better than all its neighbours, but there is not next better state which can be chosen for further expansion. Local maximum sometimes occur within sight of a solution.

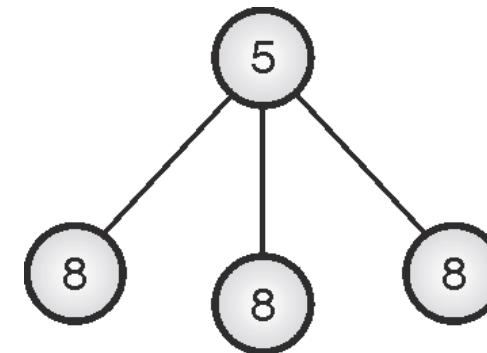
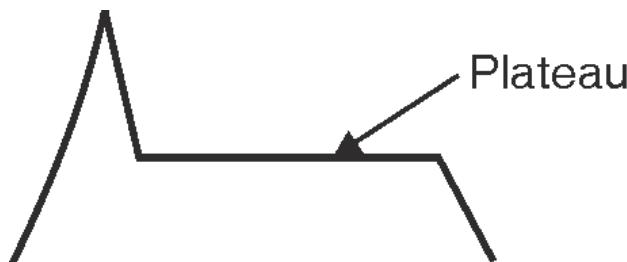


Local Maxima

Plateau :

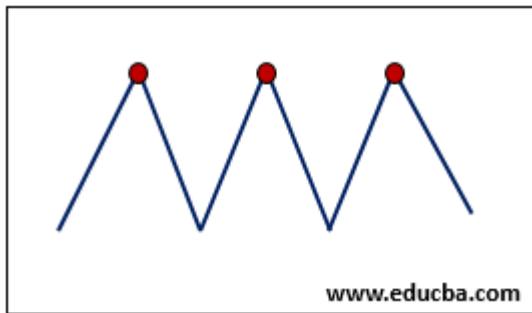
A “plateau” is a flat area at some height in hilly region. There is a large area of same height in plateau.

In the search space, plateau situation occurs when all the neighboring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

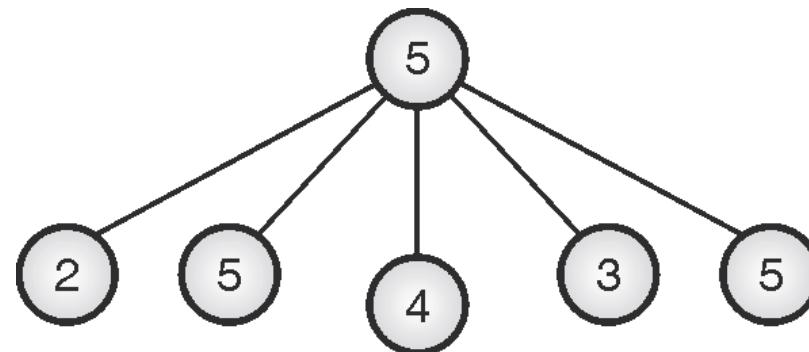


In the search tree ridge can be identified as follows

Ridge

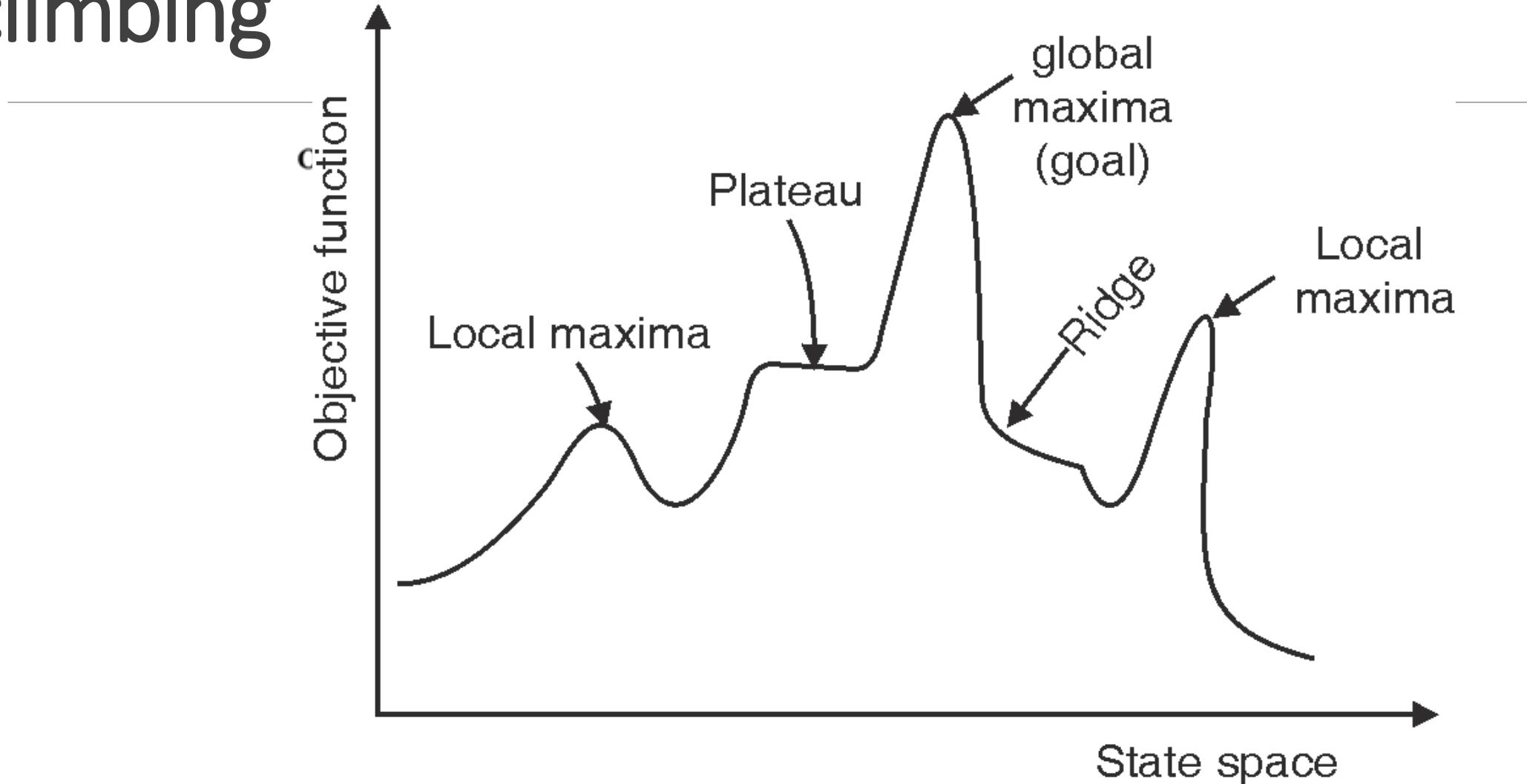


Ridges result in a sequence of local maxima that is very difficult for algorithms to navigate.

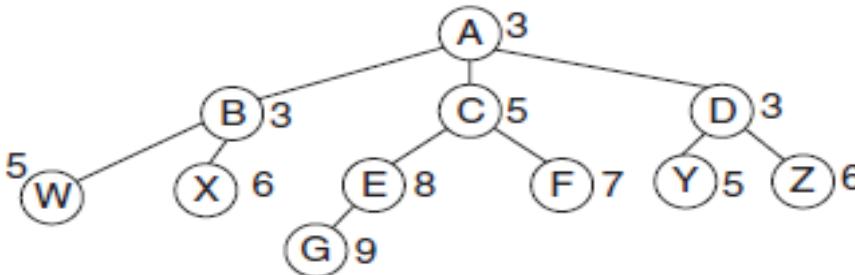


In the search tree ridge can be identified as follows :

Different Limitations together in hill climbing



Apply hill climbing to the following tree considering G is the Goal State and Node A as the initial state.



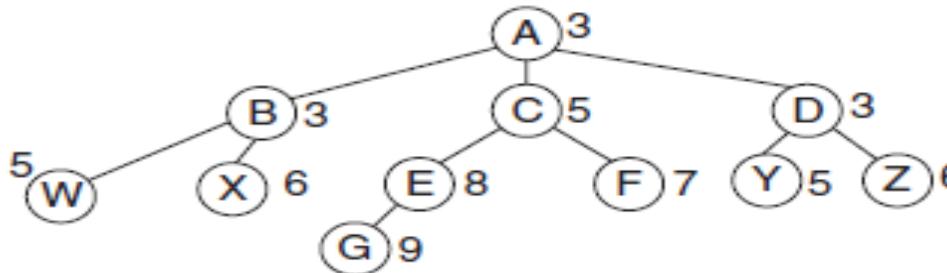
Step 1: The hill climbing algorithm starts with the initial state.



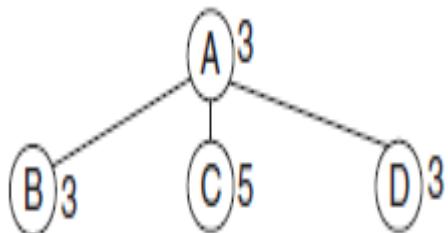
OPEN = [A3]
CLOSE = []

Note that leftmost node in the OPEN list is Node A, and hence, we expand Node A in the next step.

Apply hill climbing to the following tree considering G is the Goal State and Node A as the initial state.



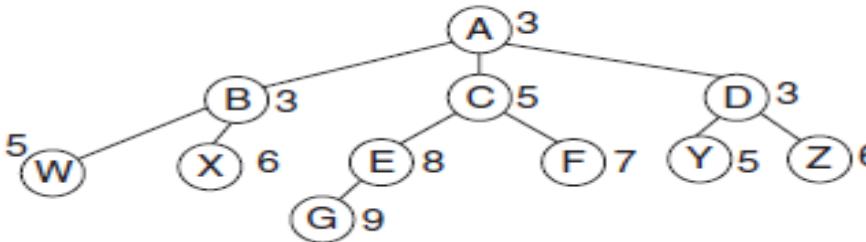
Step 2: All the children of Node A are now generated.



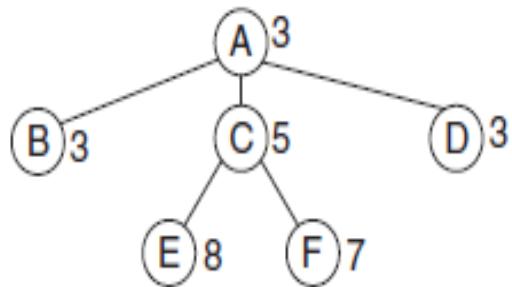
OPEN = [C5, B3, D3]
CLOSE = [A]

Note that the leftmost node in the OPEN list is Node C. Hence, we expand Node C in Step 3.

Apply hill climbing to the following tree considering G is the Goal State and Node A as the initial state.



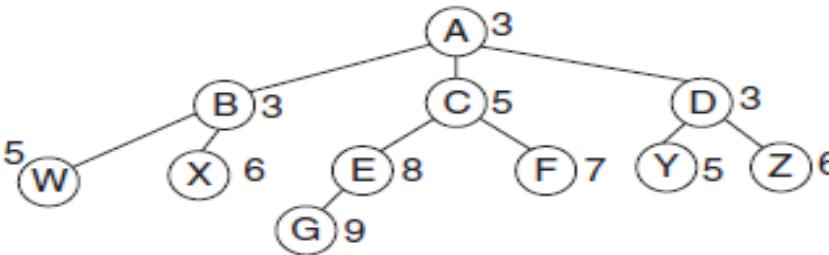
Step 3: Amongst all the children of Node root Node A Node C has the highest heuristic. Hence, all the children of Node C are now generated.



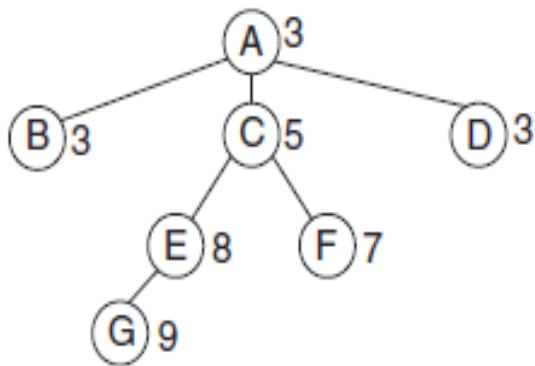
OPEN = [E8, F7, B3, D3]
CLOSE = [A, C]

Note that leftmost node in OPEN list is Node E. Hence, we expand Node E in Step 4.

Apply hill climbing to the following tree considering G is the Goal State and Node A as the initial state.



Step 4: Now the algorithm proceeds towards Node E and among all the children of Node E (only G is the child of Node C). Node G has a highest Heuristic merit; and hence, Node G is generated.



OPEN = [G9, F7, B3, D3]
CLOSE = [A, C, E]

Note that the leftmost node in OPEN list is Node G and Node G is a GOAL and hence we STOP

Simulated Annealing

Simulated annealing is a variation of hill climbing. Simulated annealing technique can be explained by an analogy to annealing in solids. In the annealing process in case of solids, a solid is heated past melting point and then cooled.

With the changing rate of cooling, the solid changes its properties. If the liquid is cooled slowly, it gets transformed in steady frozen state and forms crystals. While, if it is cooled quickly, the crystal formation will not get enough time and it produces imperfect crystals.

The aim of physical annealing process is to produce a minimal energy final state after raising the substance to high energy level. Hence in simulated annealing we are actually going downhill and the heuristic function is a minimal heuristic. The final state is the one with minimum value, and rather than climbing up in this case we are descending the valley.

Simulated Annealing

- The idea is to use simulated annealing to search for feasible solutions and converge to an optimal solution.
- In order to achieve that, at the beginning of the process, some downhill moves may be made. These downhill moves are made purposely, to do enough exploration of the whole space early on, so that the final solution is relatively insensitive to the starting state. It reduces the chances of getting caught at a local maximum, or plateau, or a ridge.

Algorithm

1. Evaluate the initial state.

2. Loop until a solution is found or there are no new operators left to be applied:

Set T according to an annealing schedule

Select and apply a new operator

Evaluate the new state :

If Goal then quit

$$E = \text{Val}(\text{current state}) - \text{Val}(\text{new state})$$

$$E < 0 = \text{new current state}$$

else new current state with probability e^{-E} / kT .

Comparing Simulated Annealing with Hill Climbing

A hill climbing algorithm never makes “downhill” moves toward states with lower value and it can be incomplete, because it can get stuck on a local maximum.

As we know that, hill climbing can get stuck at local maxima, thereby halting the algorithm abruptly, it may not guarantee optimal solution.

Hill climbing procedure chooses the best state from those available or at least better than the current state for further expansion. Unlike hill climbing, simulated annealing chooses a random move from the neighbourhood. If the successor state turned out to be better than its current state then simulated annealing will accept it for further expansion. If the successor state is worse, then it will be accepted based on some probability.

Local Beam Search

In all the variations of hill climbing till now, we have considered only one node getting selected at a time for further search process. These algorithms are memory efficient in that sense. But when an unfruitful branch gets explored even for some amount of time it is a complete waste of time and memory. Also the solution produced may not be the optimal one.

The local beam search algorithm keeps track of k best states by performing parallel k searches. At each step it generates successor nodes and selects k best nodes for next level of search. Thus rather than focusing on only one branch it concentrates on k paths which seems to be promising. If any of the successors found to be the goal, search process stops.

In parallel local beam search, the parallel threads communicate to each other, hence useful information is passed among the parallel search threads.

Local Beam Search

In turn, the states that generate the best successors say to the others, “Come over here, the grass is greener!” The algorithm quickly terminates unfruitful branches exploration and moves its resources to where the path seems most promising..

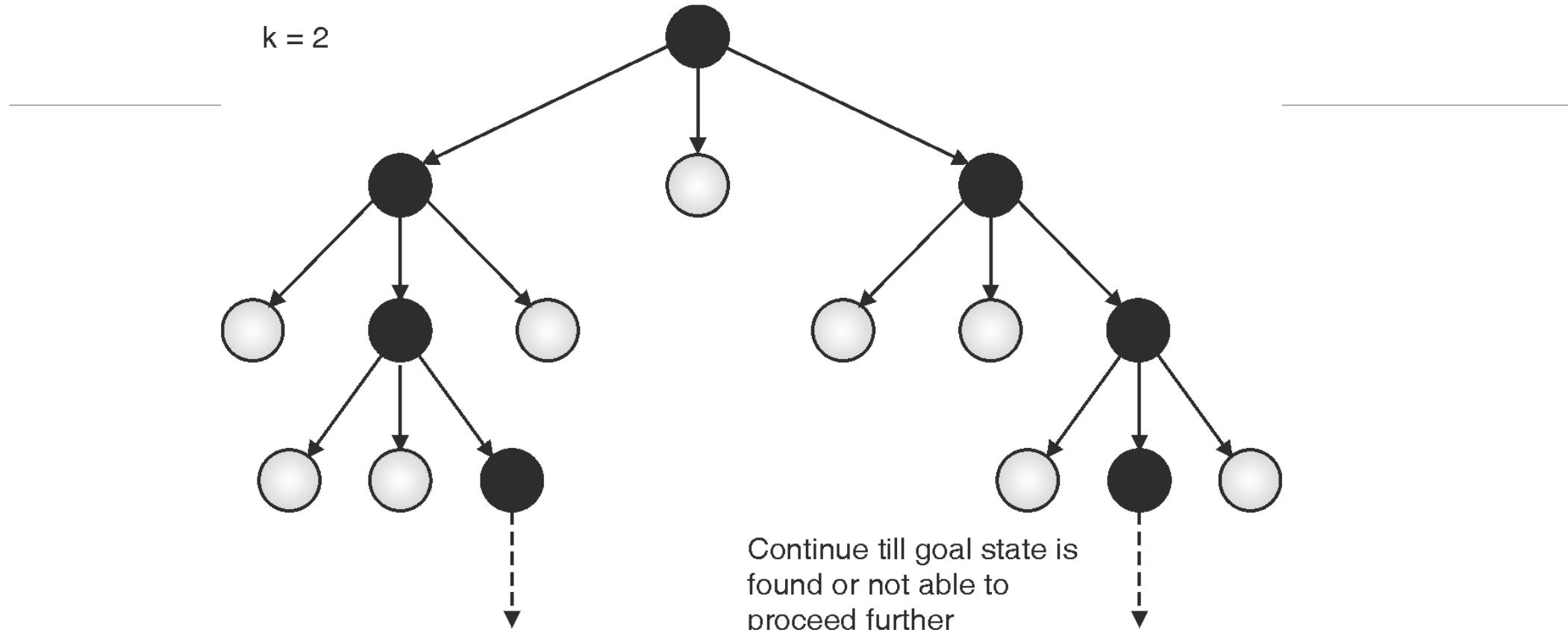
Local Beam Search

www.Bandicam.com

Lecture 14

- ▶ What is Beam search
- ▶ in artificial intelligence

Local Beam Search



here $k = 2$, hence two better successors are selected for expansion at first level of search and at each next level, two better successors will be selected by both searches. They do exchange their information with each other throughout the search process. The search will continue till goal state is found or no further search is possible.

Topics covered in the last class

Informed Search

Heuristic Function

Hill Climbing

Simple Hill Climbing

Steepest Ascent Hill Climbing

Limitations of Hill Climbing

Simulated Annealing

Informed Search

- Uninformed search methods, whether breadth-first or depth-first, are exhaustive methods for finding paths to a goal node.
 - These methods provide a solution; but often are infeasible to use because search expands too many nodes before a path is found.
- Informed search methods use task-dependent information to help reduce the search.
 - Task-dependent information is called heuristic information; and search procedures using it are called heuristic search methods.

Heuristic Function

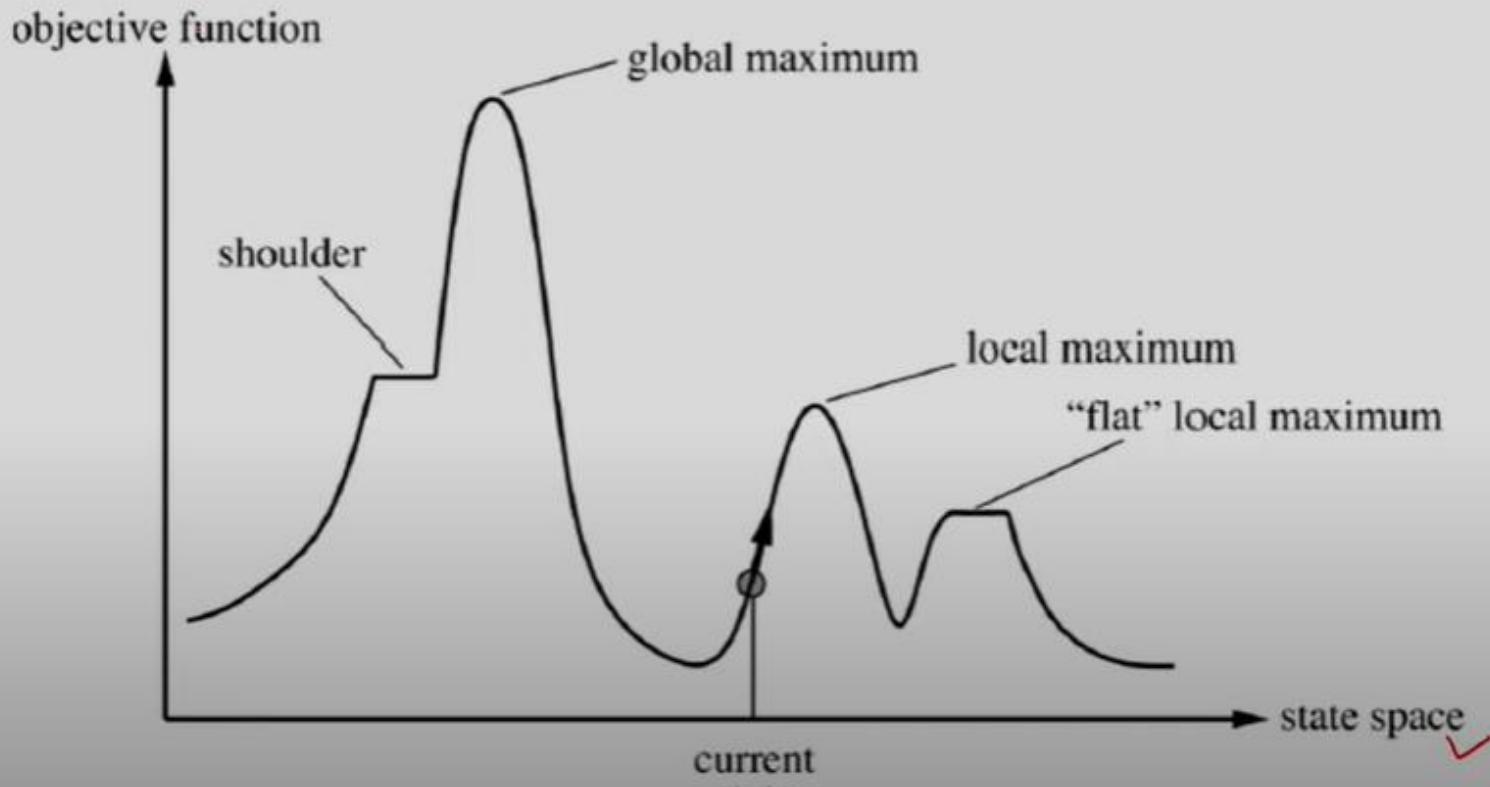
- Heuristic function, $h(n)$: estimates the cost to a goal; most promising states are selected.
- For ordering nodes for expansion, we need a method to compute the promise of a node. This is done using a real-valued evaluation function.
 - Evaluation functions have been based on a variety of ideas.

Hill Climbing search

□ Properties:

- Terminates when a peak is reached.
- **Does not look ahead** of the immediate neighbors of the current state.
- Chooses randomly among the set of best successors, if there is more than one.
- **Does not backtrack**, since it doesn't remember where it's been

Search Space Features

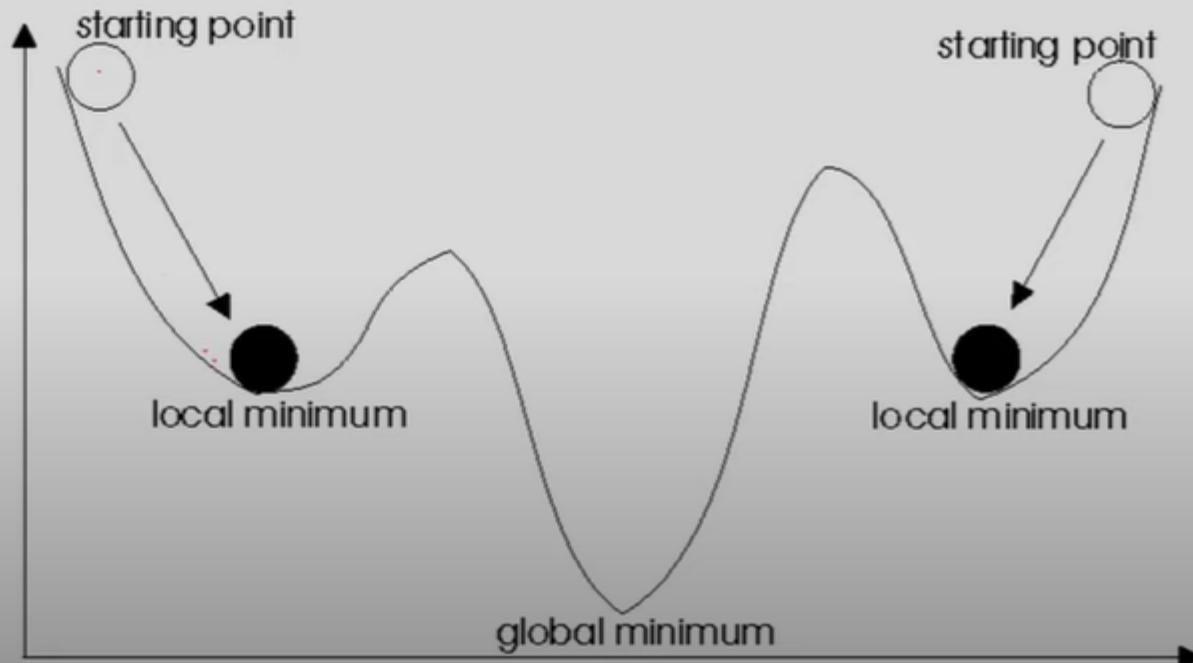


Simulated Annealing

- It is a **stochastic hill-climbing** algorithm:
 - A successor is selected among all possible successors according to a probability distribution.
 - The successor can be worse than the current state.
- **Random steps** are taken in the state space.
- It is inspired by the **physical process of controlled cooling** (crystallization, metal annealing):
 - A metal is heated up to a high temperature and then is progressively cooled in a controlled way.
 - If the cooling is adequate, the minimum-energy structure (a global minimum) is obtained.

Simulated Annealing

Aim: to avoid local optima, which represent a problem in hill climbing.



Simulated Annealing

Solution: to take, occasionally, steps in a different direction from the one in which the increase (or decrease) of energy is maximum.



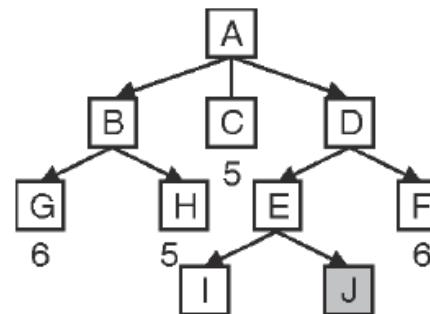
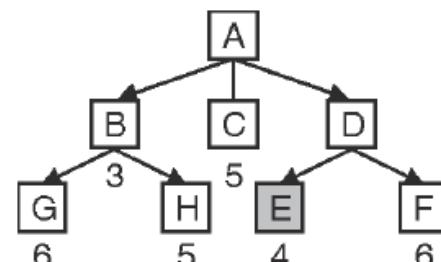
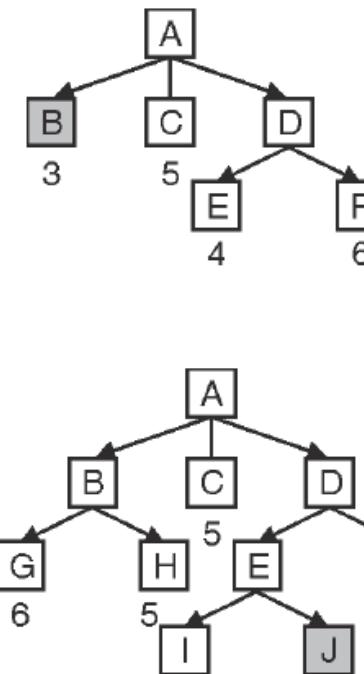
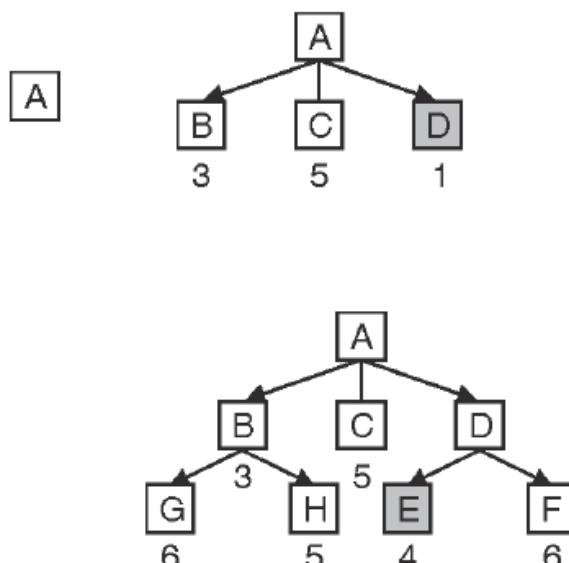
Simulated Annealing

- It is suitable for problems in which the global optimum is surrounded by many local optima.
- It is suitable for problems in which it is difficult to find a good heuristic function.
- Determining the values of the parameters can be a quite difficult and often requires experimentation.

Best First Search

Best-first search is a search algorithm which explores the search tree by expanding the most promising node chosen according to the heuristic value of nodes.

Efficient selection of the current best candidate for extension is typically implemented using a priority queue.



A* Search

It's a variation of Best First search where the evaluation of a state or a node not only depends on the heuristic value of the node but also considers its distance from the start state. It's the most widely known form of best-first search. A* algorithm is also called as OR graph / tree search algorithm.

In A* search, the value of a node n , represented as $f(n)$ is a combination of $g(n)$, which is the cost of heuristic estimation to reach to the node from the root node, and $h(n)$, which is the cost of cheapest path to reach from the node to the goal node. Hence $f(n) = g(n) + h(n)$.

As we observe the difference between the A* and Best first search is that; in Best first search only the heuristic estimation of $h(n)$ is considered while A* counts for both, the distance travelled till a particular node and the estimation of distance need to travel more to reach to the goal node, it always finds the cheapest solution.

A* search is both complete and optimal

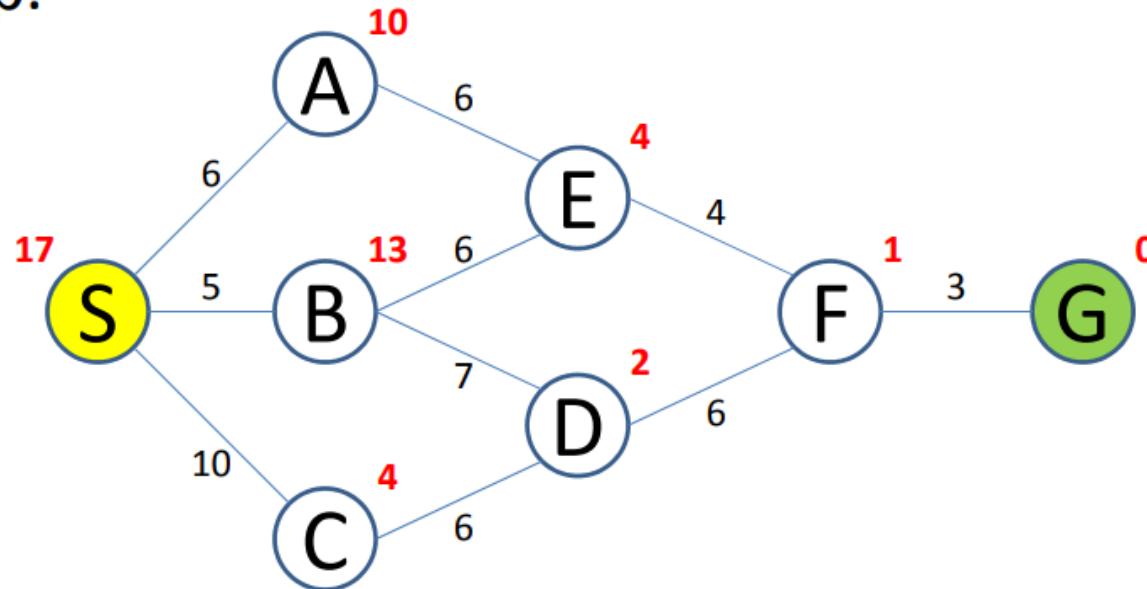
Properties of A*

- 1. Completeness** : It is complete, as it will always find solution if one exist.
- 2. Optimality** : Yes, it is Optimal.
- 3. Time Complexity** : $O(b^m)$, as the number of nodes grows exponentially with solution cost.
- 4. Space Complexity** : $O(b^m)$,as it keeps all nodes in memory.

A* Search

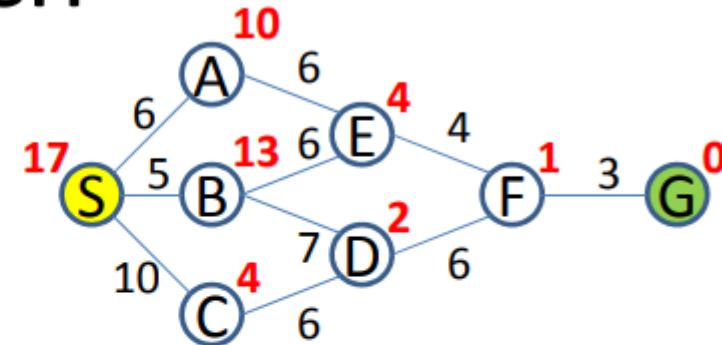
(Red figures indicate $h(n)$ and black figures $g(n)$ in the diagram)

Perform the A* Algorithm on the following figure. Explicitly write down the queue at each step.



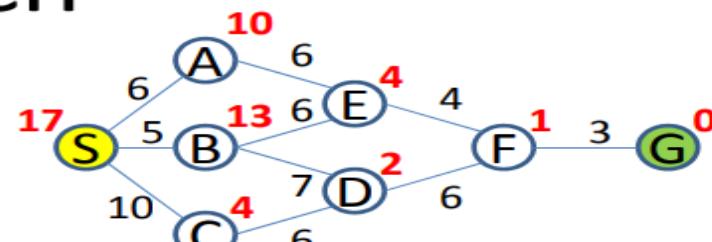
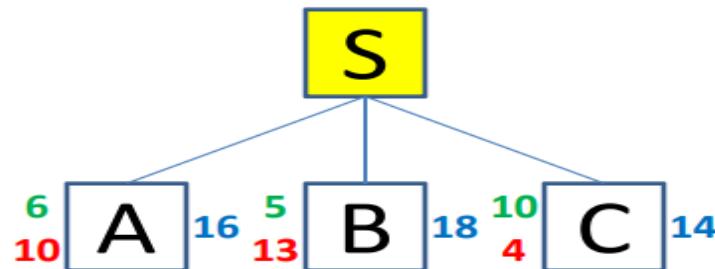
A* Search

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



QUEUE:
S

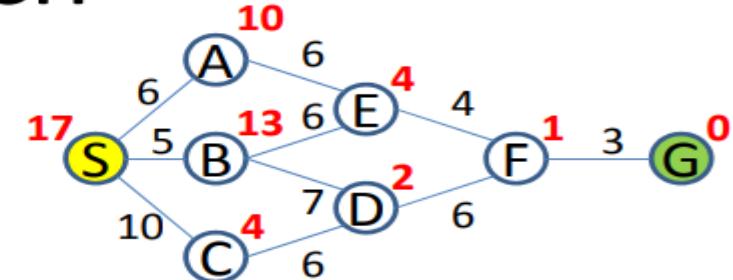
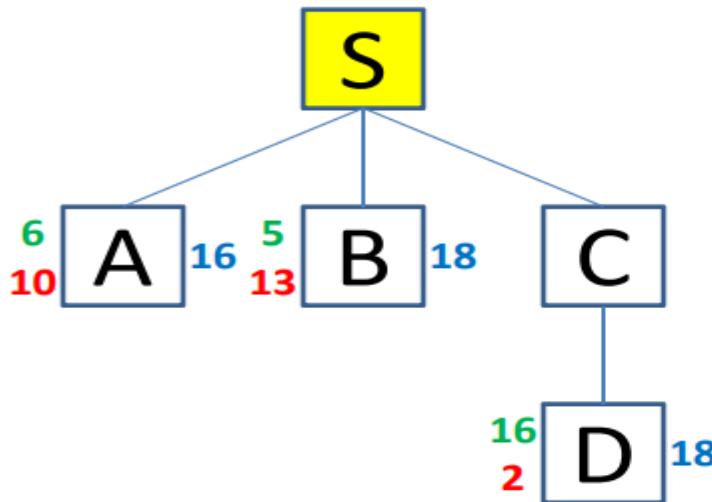
A* Search



QUEUE:
SC
SA
SB

A* Search

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



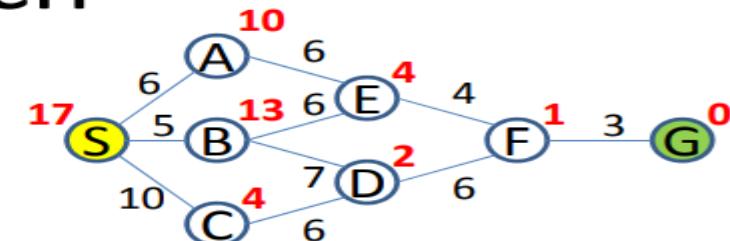
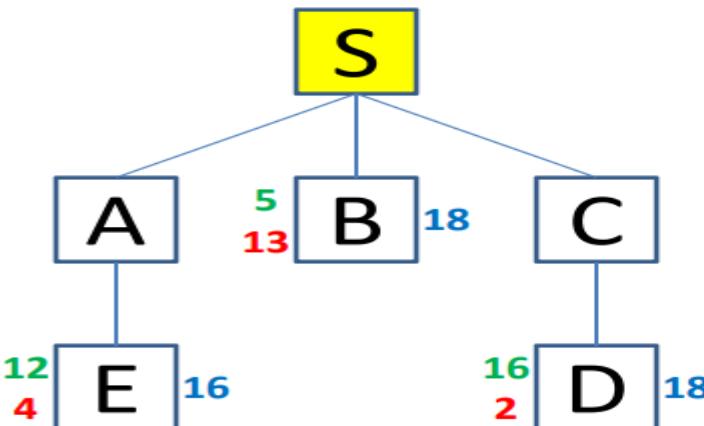
QUEUE:

SA

SCD

SB

A* Search



QUEUE:

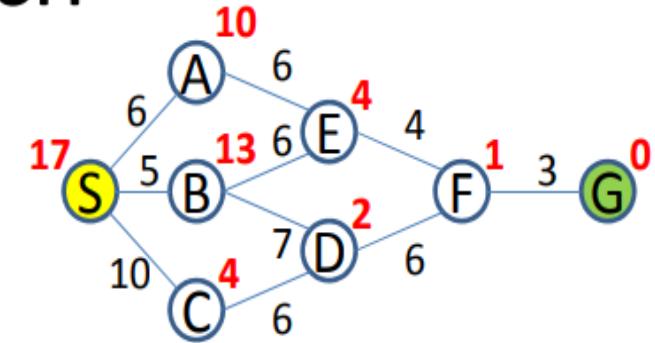
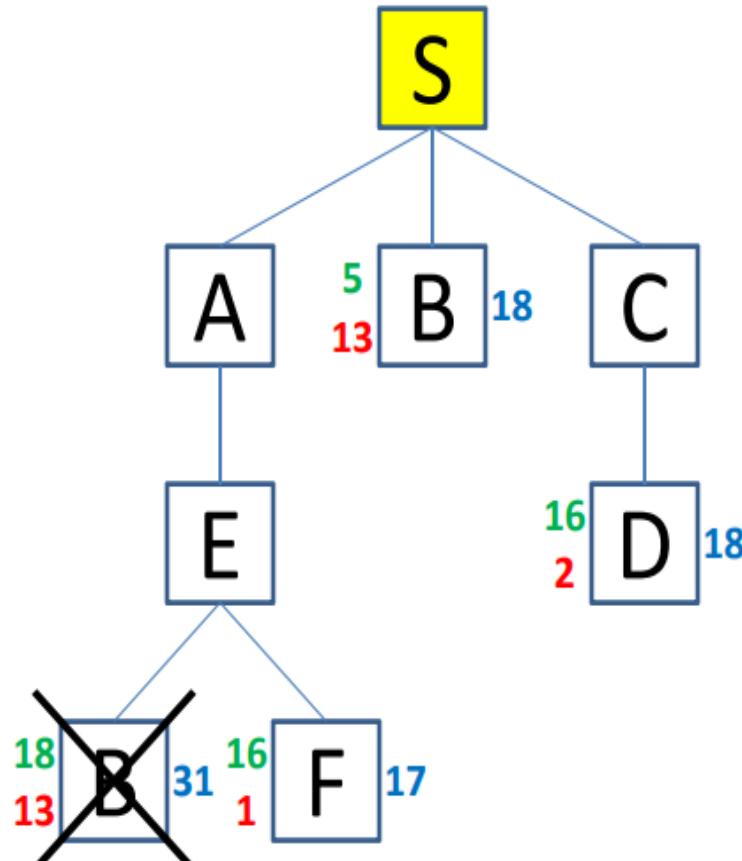
SAE

SCD

SB

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

A* Search



QUEUE:

SAEF

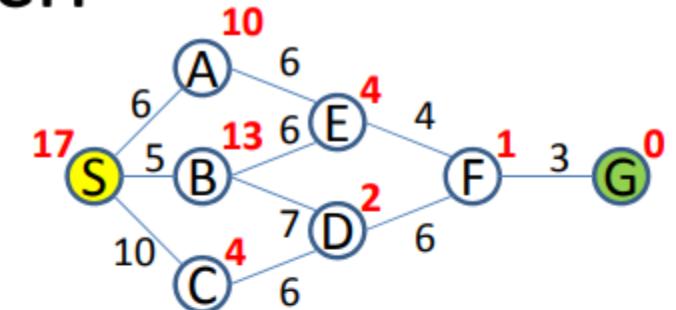
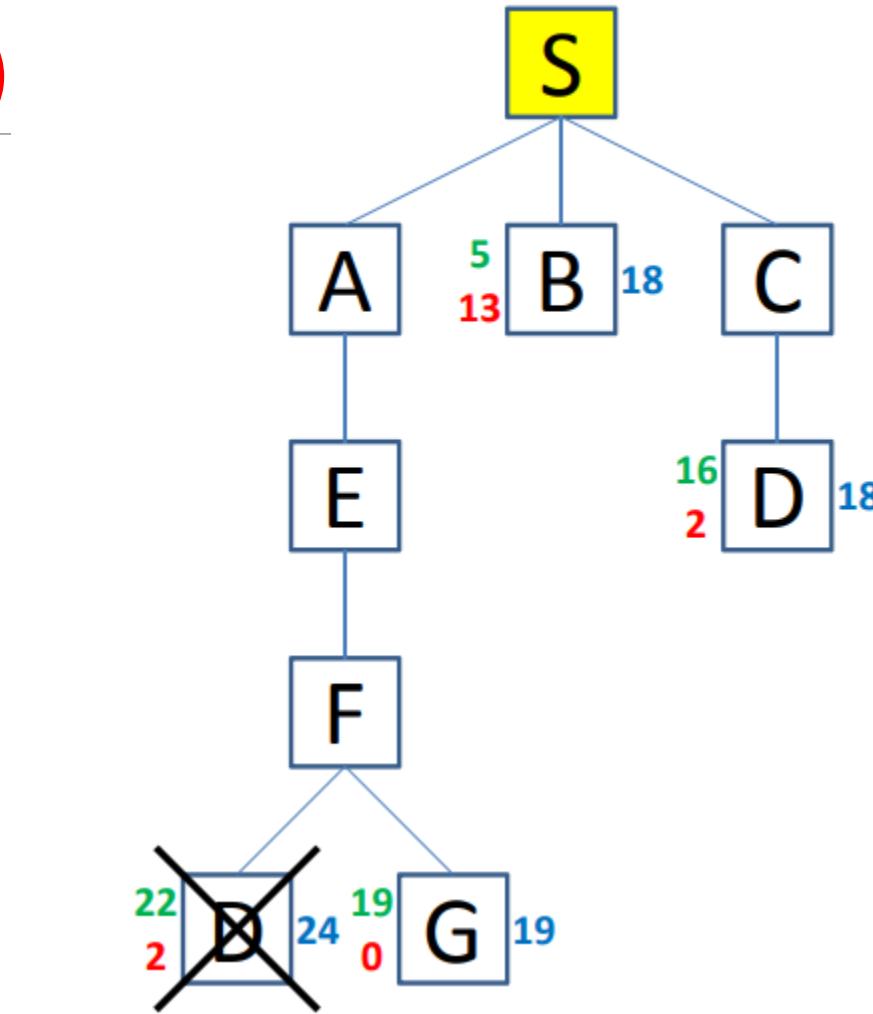
SCD

SB

SAEB

A* Search

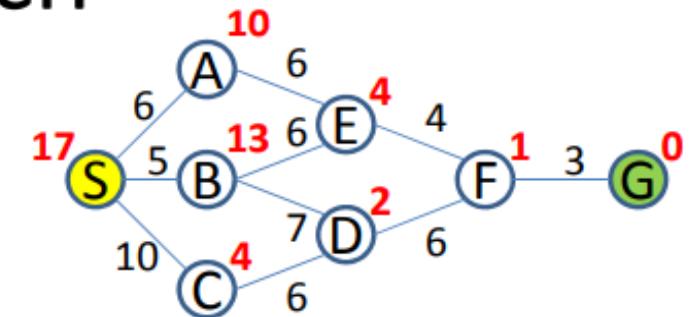
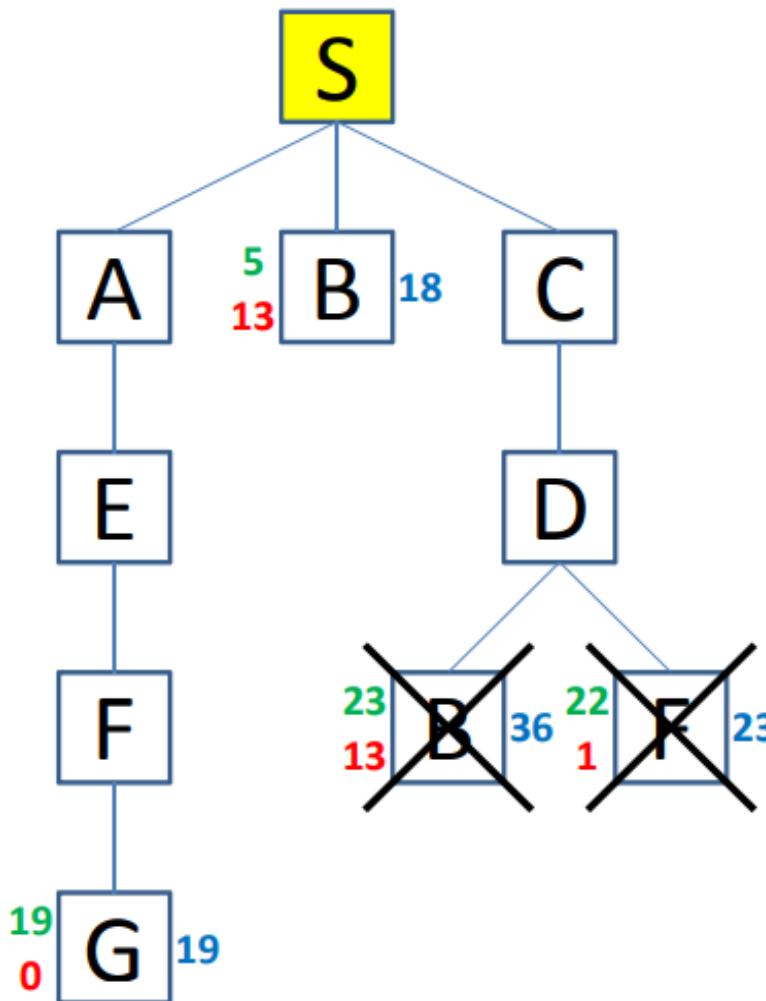
$$f(n) = g(n) + h(n)$$



QUEUE:
SCD
SB
SAEFG
SAEFD

A* Search

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



QUEUE:

SB

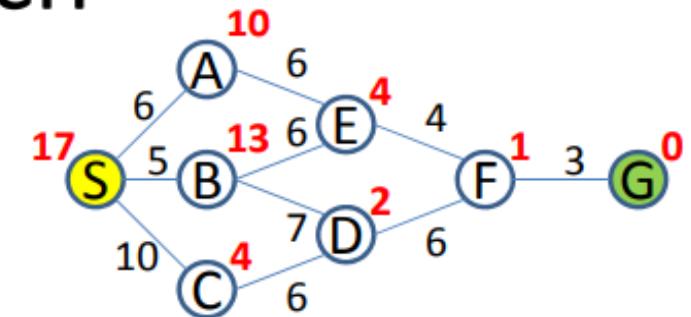
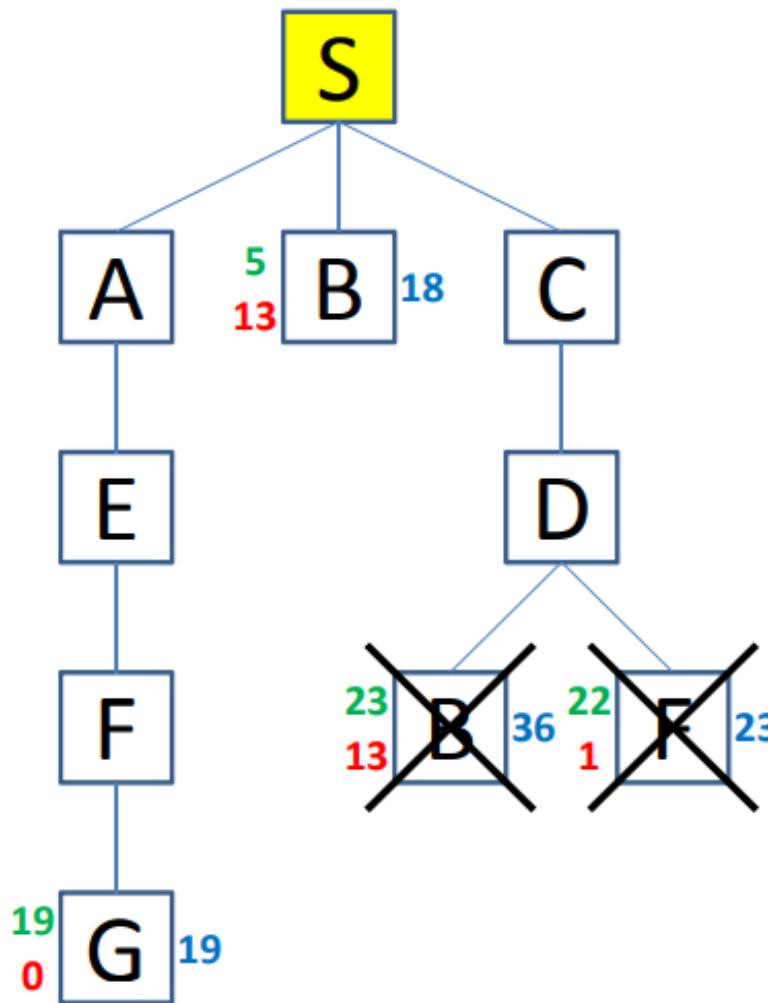
SAEFG

SCDF

SCDB

A* Search

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



QUEUE:

SB

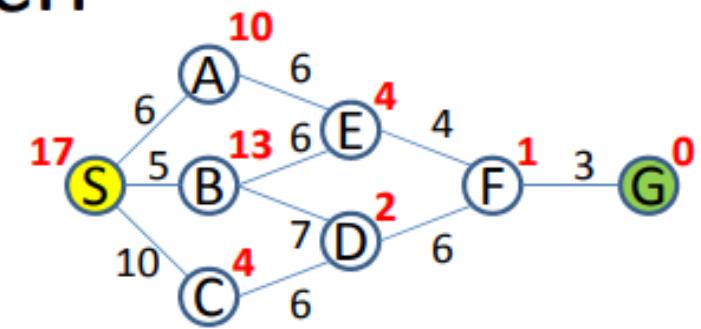
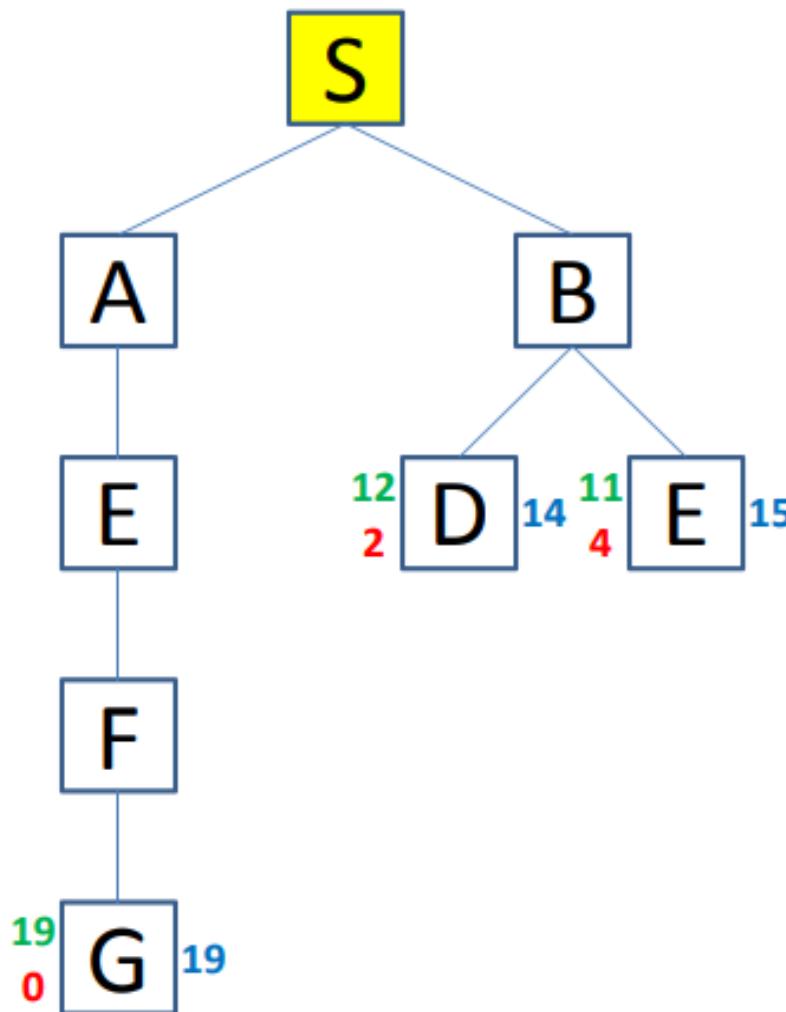
SAEFG

SCDF

SCDB

A* Search

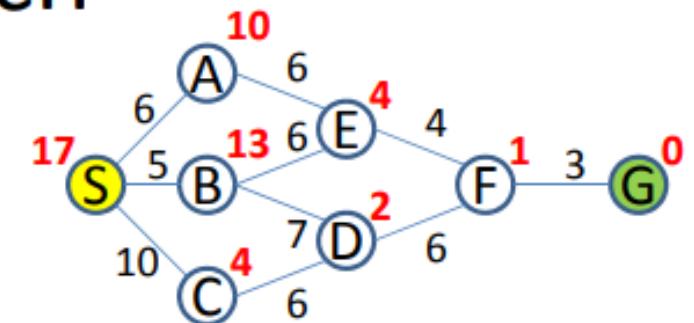
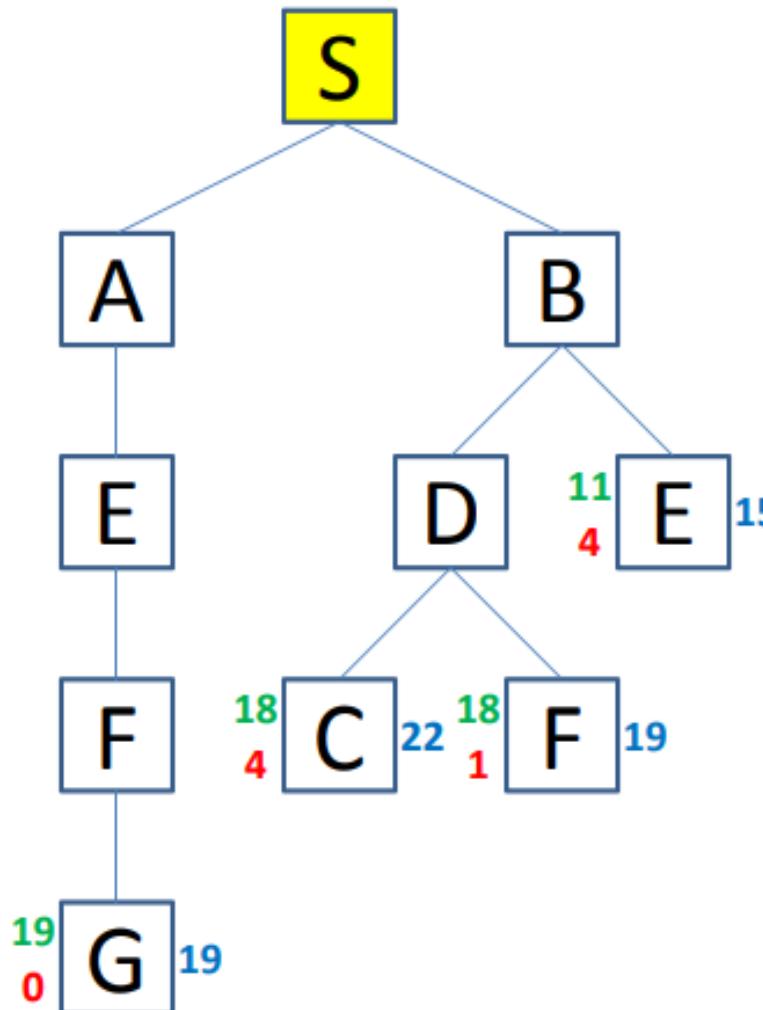
$$f(n) = g(n) + h(n).$$



QUEUE:
SBD
SBE
SAEFG

A* Search

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



QUEUE:

SB

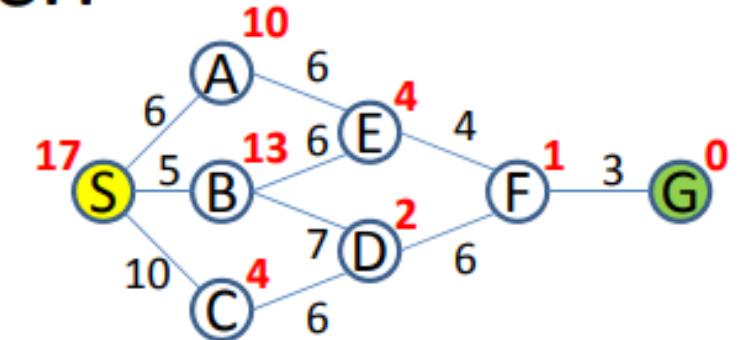
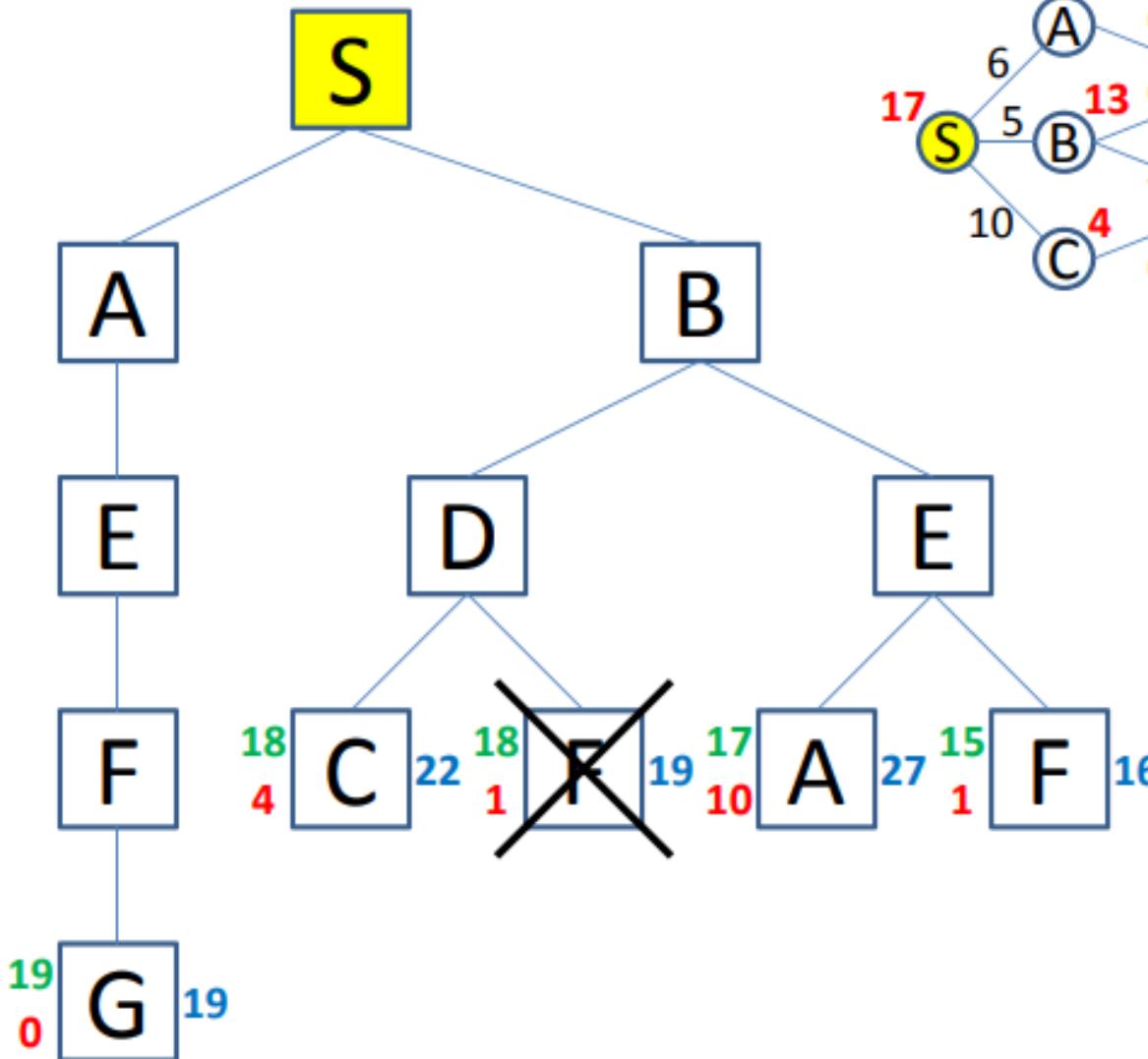
SBDF

SAEFG

SBDC

A* Search

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



QUEUE:

SBEF

SAEFG

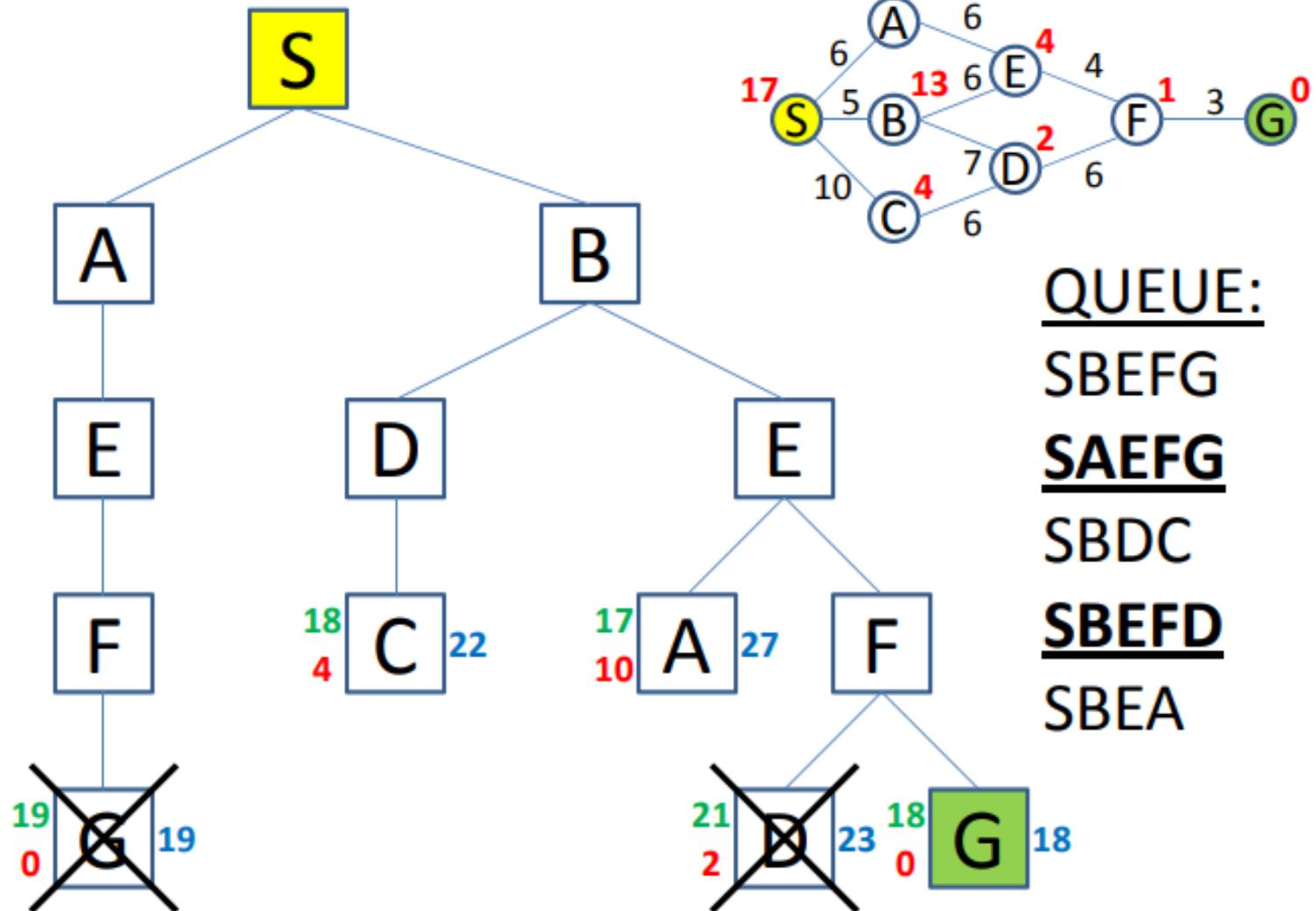
SBDF

SBDC

SBEA

A* Search

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



Admissible heuristics

A heuristic $h(n)$ is **admissible** if for every node n ,

$h(n) \leq h^*(n)$, where $h^*(n)$ is the **true** cost to reach the goal state from n .

An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is **optimistic**

A* Search

Start state

3	7	6
5	1	2
4	□	8

Goal state

5	3	6
7	□	2
4	1	8

Search tree

Start state

$$f = 0 + 4$$

3	7	6
5	1	2
4	□	8

up
(1+3)

3	7	6
5	□	2
4	1	8

left
(1+5)

3	7	6
5	1	2
□	4	8

right
(1+5)

3	7	6
5	1	2
4	8	□

up
(2+3)

3	□	6
5	7	2
4	1	8

left
(2+3)

3	7	6
□	5	2
4	1	8

right
(2+4)

3	7	6
5	2	□
4	1	8

left
(3+2)

□	3	6
5	7	2
4	1	8

right
(3+4)

3	6	□
5	7	2
4	1	8

down
(4+1)

5	3	6
□	7	2
4	1	8

5	3	6
7	□	2
4	1	8

Goal
state

A* Search

The choice of evaluation function critically determines search results.

- Consider Evaluation function

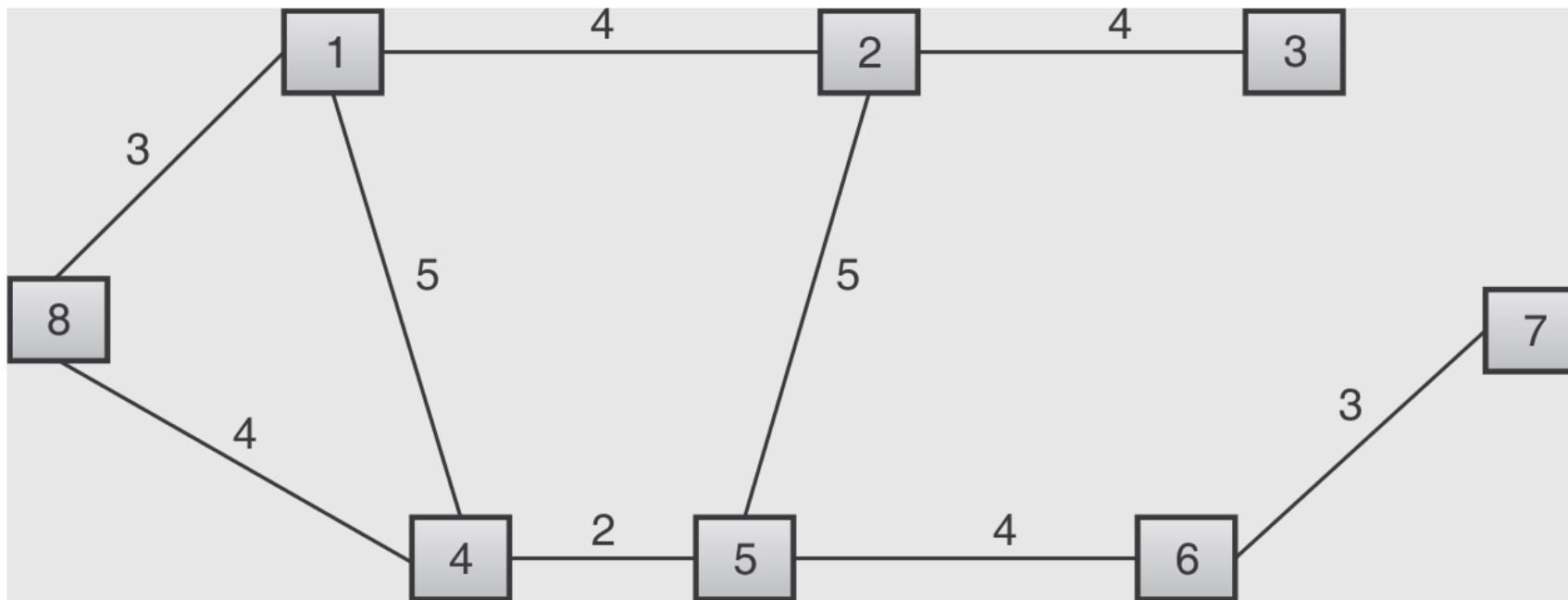
$$f(X) = g(X) + h(X)$$

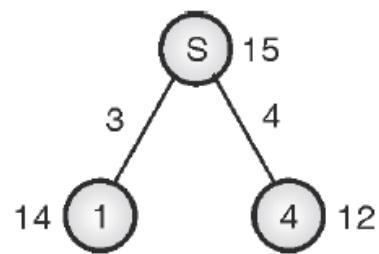
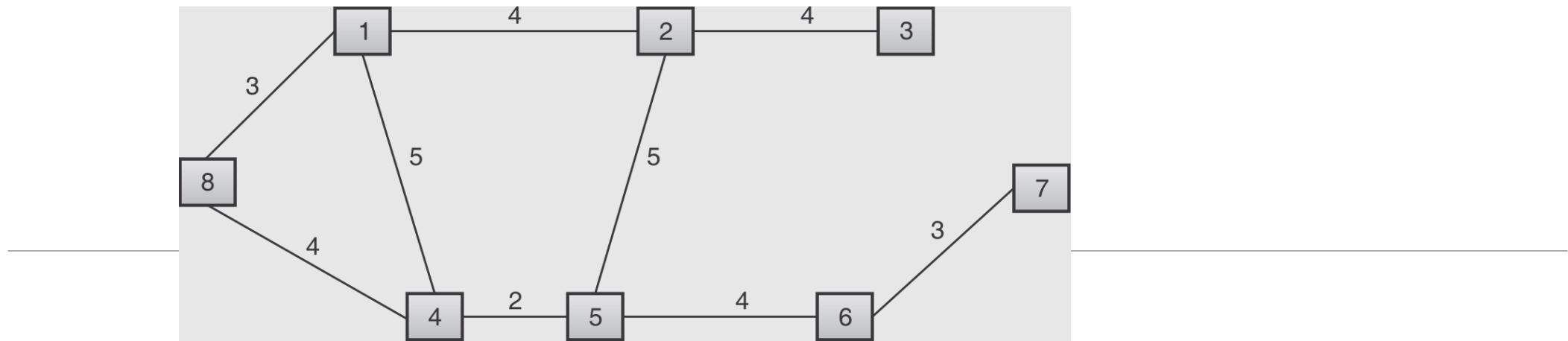
$h(X)$ = the number of tiles not in their goal position in a given state X

$g(X)$ = depth of node X in the search tree

- For Initial node $f(\text{initial_node}) = 4$

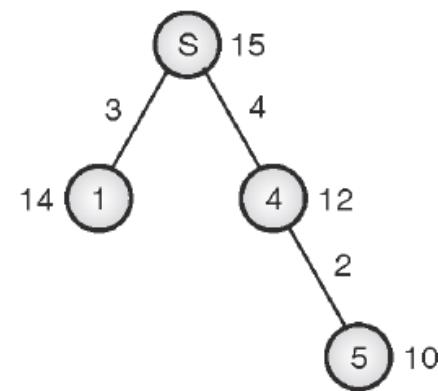
Consider the graph given below. Assume that the initial state is S and the goal state is 7. Find a path from the initial state to the goal state using A* Search. Also report the solution cost. The straight line distance heuristic estimates for the nodes are as follows: $h(1) = 14$, $h(2) = 10$, $h(3) = 8$, $h(4) = 12$, $h(5) = 10$, $h(6) = 10$, $h(S) = 15$.





Open : 4(12 + 4), 1(14 + 3)

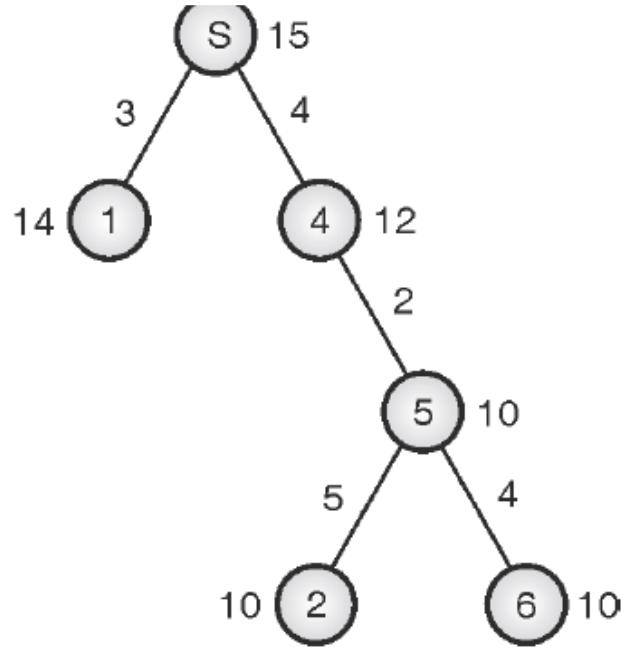
Closed : S(15)



Open : 5(10 + 6), 1(14 + 3)

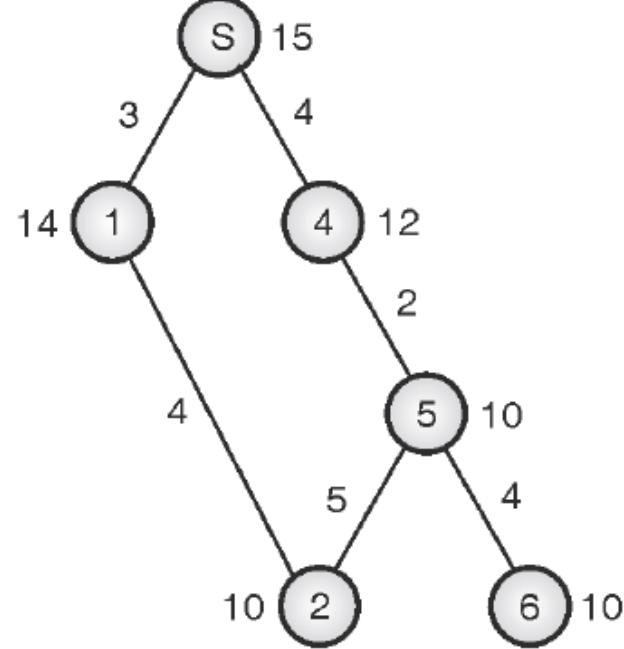
Closed : S(15), 4 (12 + 4)

$$h(1) = 14, h(2) = 10, h(3) = 8, h(4) = 12, h(5) = 10, h(6) = 10, h(S) = 15.$$



Open : 1($14 + 3$) 6($10 + 10$) 2($10 + 11$)

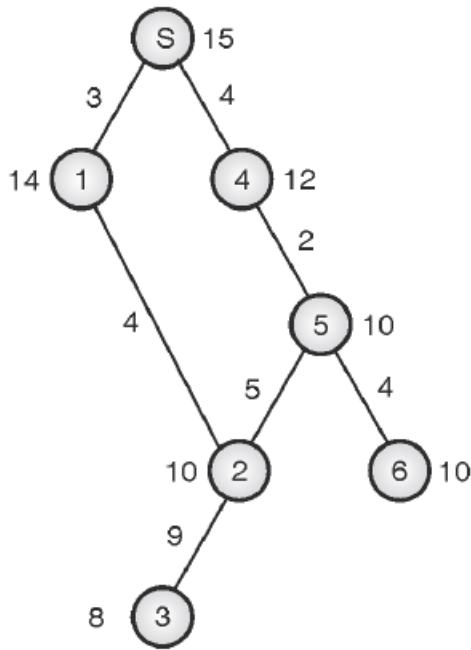
Closed : S(15) 4($12 + 4$) 5($10 + 6$)



Open : 2($10 + 7$) 6($10+10$)

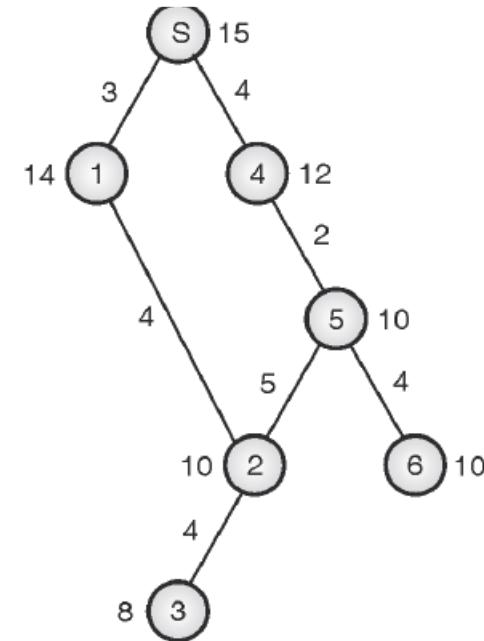
Closed : S(5) 4($12 + 4$) 5($10 + 6$) 1($14 +3$)

$$h(1) = 14, h(2) = 10, h(3) = 8, h(4) = 12, h(5) = 10, h(6) = 10, h(S) = 15.$$



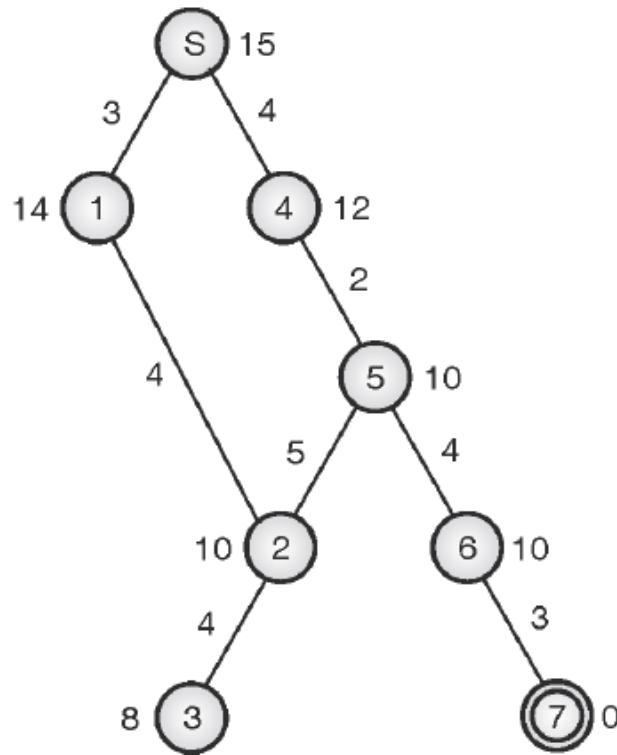
Open : $3(8 + 11), 6(10 + 10)$

Closed : $S(15), 4(12+4), 5(10+6),$
 $1(14+3), 2(10+7)$



Open : $6(10 + 10)$

Closed : $s(15), 4(12 + 4), 5(10 + 6),$
 $1(14 + 3) 2(10 + 7) 3(8 + 11)$



Open : 7(0 + 13)

Closed : S(15), 4(12 + 4), 5(10 + 6), 1(14 + 3), 2(10 + 7), 5(8 + 4), 6(10 + 10)

Topics covered in the last class

Informed Search

Heuristic Function

Hill Climbing

Simple Hill Climbing

Steepest Ascent Hill Climbing

Limitations of Hill Climbing

Simulated Annealing

Local Beam Search

Best First Search

A* Search

Constraint Satisfaction Problem

Constraint Satisfaction Problem

Constraint :

Constraint is a logical relation among variables. They are a natural medium for people to express problems in many fields.

Examples of constraints :

- The sum of three angles of a triangle is 180 degrees,
- The sum of the currents flowing into a node must equal zero.

Constraint satisfaction :

The Constraint satisfaction is a process of finding a solution to a set of constraints. Constraints articulate allowed values for variables, and finding solution is evaluation of these variables that satisfies all constraints.

Constraint Satisfaction Problem

In standard search problems, we are given with state space, heuristic function and goal state. In this case the state is represented in the form of any data structure that supports **successor function, heuristic function, and goal test**.

– Constraint Satisfaction Problems are **special type of search in which, a state is defined by variables X_i with values from domain D_i and goal test is a set of constraints specifying allowable combinations of values for subsets of variables. It is given by three tuple $\langle X, D, C \rangle$**

Many real problems in AI can be modeled as Constraint Satisfaction Problems. CSP allows useful general purpose algorithms with more power than standard search algorithms. CSPs are solved through search.

Examples of CSPs

Map colouring problem :

In this example, Variables are WA, NT, Q, NSW, V, SA, T

Domains $D_i = \{\text{red, green, blue}\}$

Constraints : adjacent regions must have different colours.

Ex. : $WA \neq NT$, or $(WA, NT) \in \{(\text{red, green}), (\text{red, blue}), (\text{green, red}), (\text{green, blue}), (\text{blue, red}), (\text{blue, green})\}$

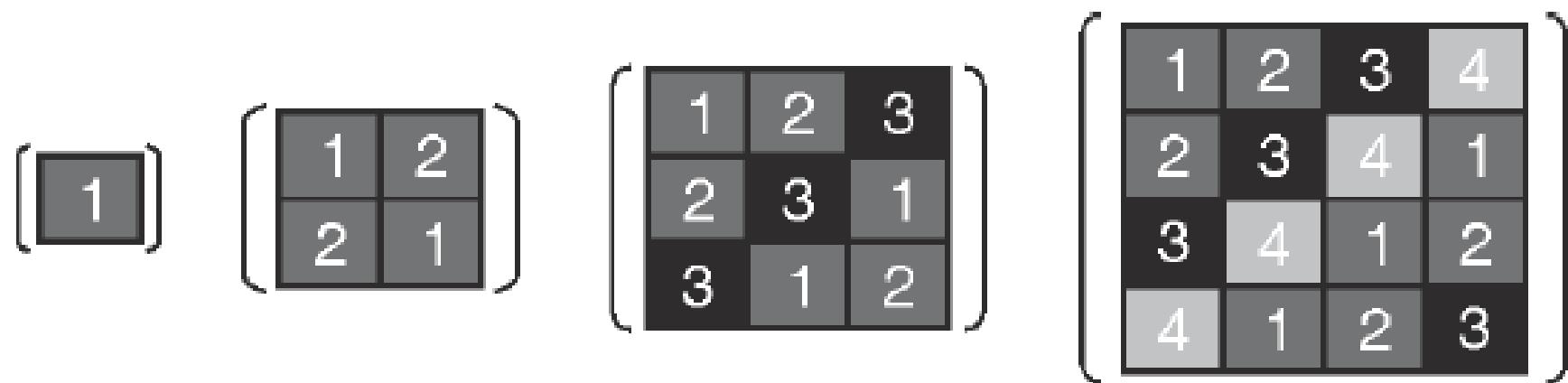


Examples of CSPs

Latin square problem :

How can one fill an $n \times n$ table with n different symbols such that each symbol occurs exactly once in each row and each column ?

Solutions : The Latin squares for $n = 1, 2, 3$ and 4 are :



The image shows a sequence of four nested curly braces, each enclosing a Latin square of a specific size. The first brace encloses a single cell containing the number 1. The second brace encloses a 2x2 square with cells (1,1)=1 and (2,2)=2. The third brace encloses a 3x3 square with cells (1,1)=1, (1,2)=2, (1,3)=3, (2,1)=2, (2,2)=3, (2,3)=1, (3,1)=3, (3,2)=1, and (3,3)=2. The fourth brace encloses a 4x4 square with cells (1,1)=1, (1,2)=2, (1,3)=3, (1,4)=4, (2,1)=2, (2,2)=3, (2,3)=4, (2,4)=1, (3,1)=3, (3,2)=4, (3,3)=1, (3,4)=2, (4,1)=4, (4,2)=1, (4,3)=2, and (4,4)=3.

1			
1 2	2 1		
1 2 3	2 3 1	3 1 2	
1 2 3 4	2 3 4 1	3 4 1 2	4 1 2 3

Examples of CSPs

Eight queens puzzle problem :

How can one put 8 queens on a (8×8) chess board such that no queen can attack any other queen ?

Solutions : The puzzle has 92 distinct solutions. If rotations and reflections of the board are counted as one, the puzzle has 12 unique solutions.

Examples of CSPs

Real-world CSPs :

1. Assignment problems : E.g., who teaches what class.
2. Timetabling problems : E.g., which class is offered when and where?
3. Transportation scheduling : Factory scheduling

Constraint Satisfaction Problem

Set of variables $\{X_1, X_2, \dots, X_n\}$

Each variable X_i has a domain D_i of possible values

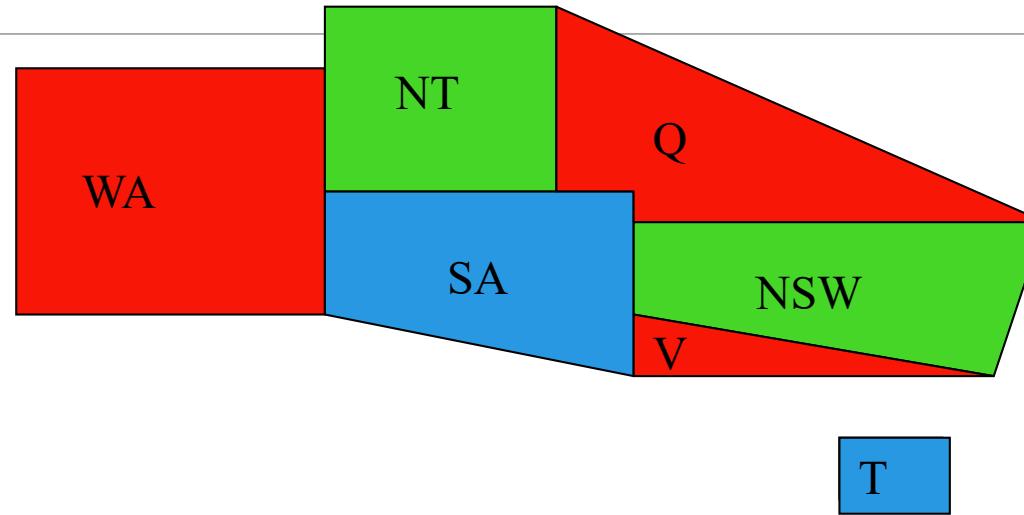
Usually D_i is discrete and finite

Set of constraints $\{C_1, C_2, \dots, C_p\}$

Each constraint C_k involves a subset of variables and specifies the allowable combinations of values of these variables

Assign a value to every variable such that all constraints are satisfied

Example: Map Coloring



- 7 variables {WA,NT,SA,Q,NSW,V,T}
- Each variable has the same domain {red, green, blue}
- No two adjacent variables have the same value:
WA \neq NT, WA \neq SA, NT \neq SA, NT \neq Q, SA \neq Q, SA \neq NSW, SA \neq V, Q \neq NSW, NSW \neq V

Example: 8-Queens Problem

64 variables X_{ij} , $i = 1$ to 8, $j = 1$ to 8

Domain for each variable {yes,no}

Constraints are of the forms:

- $X_{ij} = \text{yes} \rightarrow X_{ik} = \text{no}$ for all $k = 1$ to 8, $k \neq j$
- $X_{ij} = \text{yes} \rightarrow X_{kj} = \text{no}$ for all $k = 1$ to 8, $k \neq i$
- Similar constraints for diagonals

Varieties of CSPs

Basis on various combinations of different types of variables and domains, we have varieties of CSPs.

They are as follows :

1. **Discrete variables and finite domains** : n variables, domain size $d \rightarrow O(d^n)$ complete assignments.

e.g., Boolean CSP

2. **Discrete variables and infinite domains** : integers, strings, etc. range of values

e.g., job scheduling, variables are start/end for each job need a constraint language, e.g., $StartJob1 + 5 \leq StartJob3$

3. **Continuous variables** : E.g., start/end times for Hubble Space Telescope observations

Varieties of Constraints

1. Unary constraints involve a single variable.

e.g. SA \neq green

2. Binary constraints involve pairs of variables.

e.g. SA \neq WA

3. Higher-order constraints involve 3 or more variables.

e.g. crypt arithmetic column constraints

Standard Search Formulation

- Let's start with a basic, naive approach and then improve it
- States are defined by the values assigned so far
 - Initial state: the empty assignment, {}✓
 - Successor function: assign a value to an unassigned variable that does not conflict with current assignment; fail if no legal assignments.
 - Goal test: the current assignment is complete

Note:

1. This is the same for all CSPs!
2. Every solution appears at depth n with n variables✓
3. Path is irrelevant, so can also use complete-state formulation
4. Domain of size d , branching factor $b = (n - \ell) d$ at depth ℓ ; $n!d^n$ leaves!

Backtracking Search

- CSPs can be solved by a specialized version of depth first search.

□ Key intuitions:

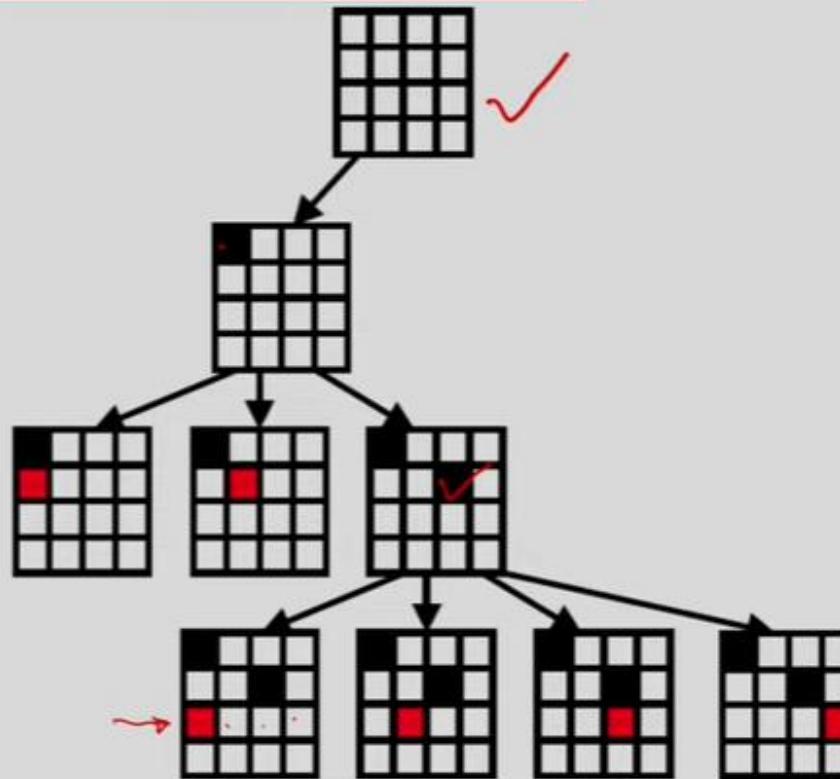
- We can build up to a solution by searching through the space of partial assignments.
- Order in which we assign the variables does not matter
 - eventually they all have to be assigned.
- If during the process of building up a solution we falsify a constraint, we can immediately reject all possible ways of extending the current partial assignment.

Backtracking Search

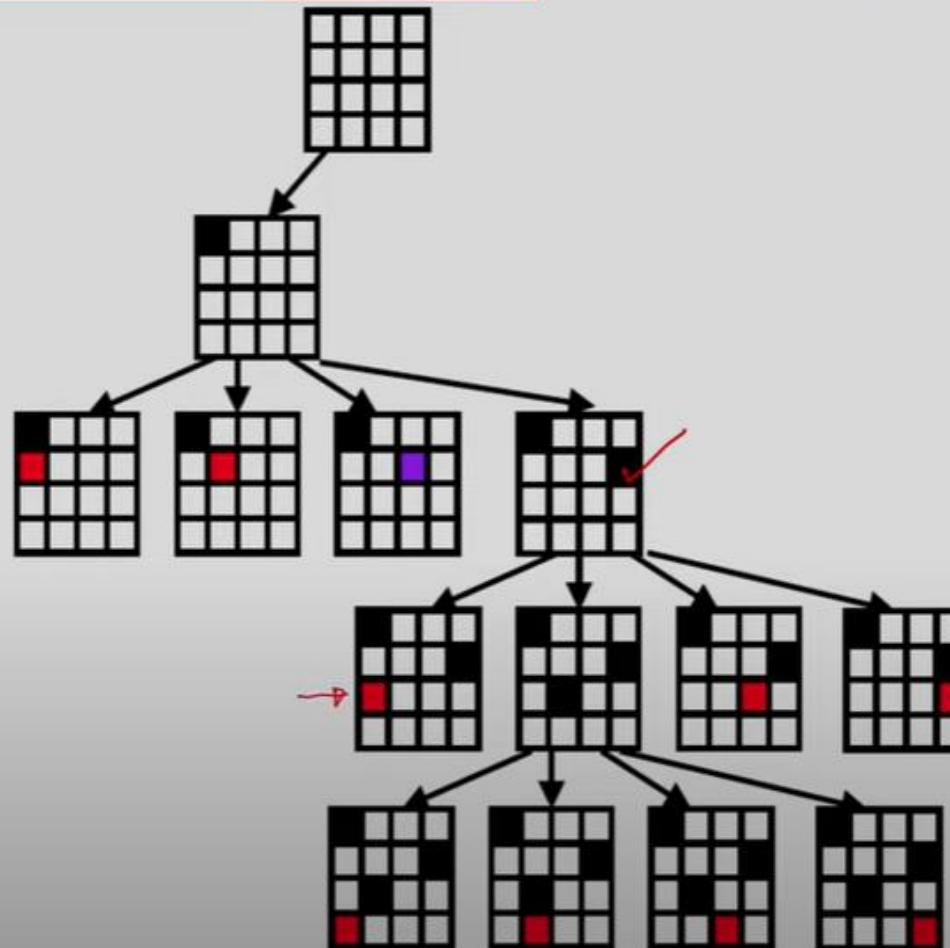
The term **backtracking search** is used for a **depth-first search** that chooses **values** for one variable at a time and **backtracks** when **no legal values left to assign**.

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add {var = value} to assignment
            result  $\leftarrow$  RECURSIVE-BACKTRACKING(assignment, csp)
            if result  $\neq$  failure then return result
            remove {var = value} from assignment
    return failure
```

Backtracking Search



Backtracking Search



Backtracking Search

- Plain backtracking is an uninformed algorithm
 - Do not expect it to be very effective for large problems.
- We remedied the poor performance of uninformed search algorithms by supplying them with domain-specific heuristic functions derived from our knowledge of the problem.

Constraint Satisfaction Problem

In standard search problems, we are given with state space, heuristic function and goal state. In this case the state is represented in the form of any data structure that supports **successor function, heuristic function, and goal test**.

– Constraint Satisfaction Problems are **special type of search in which, a state is defined by variables X_i with values from domain D_i and goal test is a set of constraints specifying allowable combinations of values for subsets of variables. It is given by three tuple $\langle X, D, C \rangle$**

Many real problems in AI can be modeled as Constraint Satisfaction Problems. CSP allows useful general purpose algorithms with more power than standard search algorithms. CSPs are solved through search.

Improving Backtracking Search

- We can solve CSPs efficiently without such domain-specific knowledge; general-purpose methods that address the following questions:
 1. Which **variable** should be assigned next, and in what **order** should its values be tried?
 2. What are the **implications** of the current **variable assignments** for the other unassigned variables?
 3. When a path fails i.e., a state is reached in which a variable has no legal values; **can the search avoid repeating this failure** in subsequent paths?

Variable and Value Ordering



- Minimum remaining values (MRV):
 - choose variable with the fewest legal values.



The backtracking algorithm contains the line

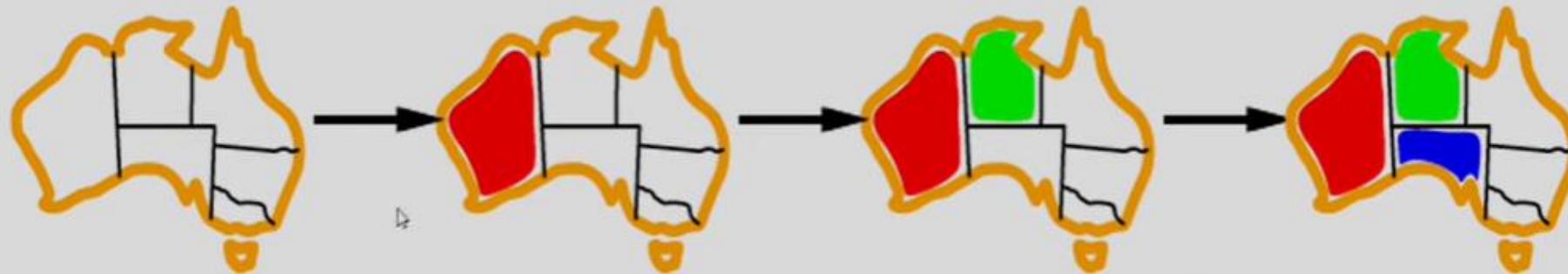
```
var SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp).
```

By default, SELECT-UNASSIGNED-VARIABLE simply selects the next unassigned variable in the order given by the list VARIABLES[csp]. This static variable ordering seldom results in the most efficient search.



Variable and Value Ordering

- Minimum remaining values (MRV):
 - choose variable with the fewest legal values.



After the assignments for WA=red and NT =green, there is only one possible value for SA, so it makes sense to assign SA=blue next rather than assigning Q.

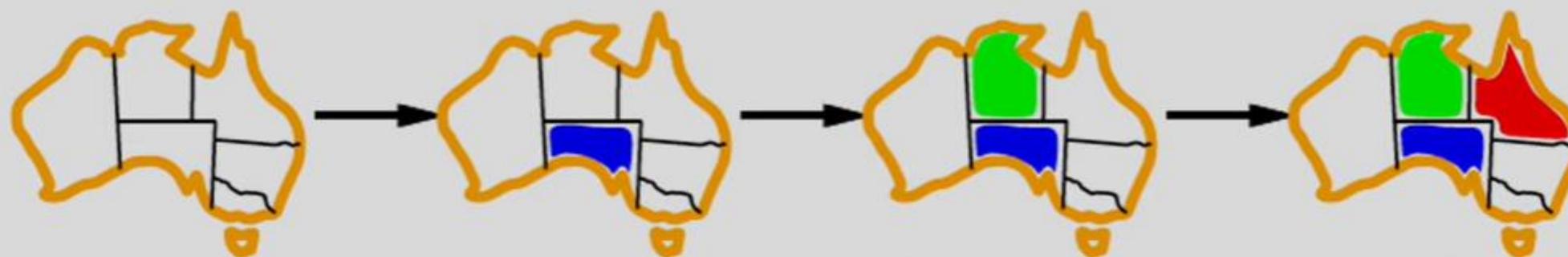
It also has been called the “most constrained variable” or “fail-first” heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree.

Variable and Value Ordering



□ Degree Heuristics:

- choose the variable with the most constraints on remaining vars.



SA is the variable with highest degree, 5; the other variables have degree 2 or 3, except for T, which has 0.

Applying the degree heuristic solves the problem without any false steps —choose any consistent colour at each choice point and still arrive at a solution with no backtracking.

Variable and Value Ordering

□ Least Constraining value:

- choose the one that rules out the fewest values in the remaining variables.



Once a variable has been selected, the algorithm must decide on the order in which to examine its values. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.

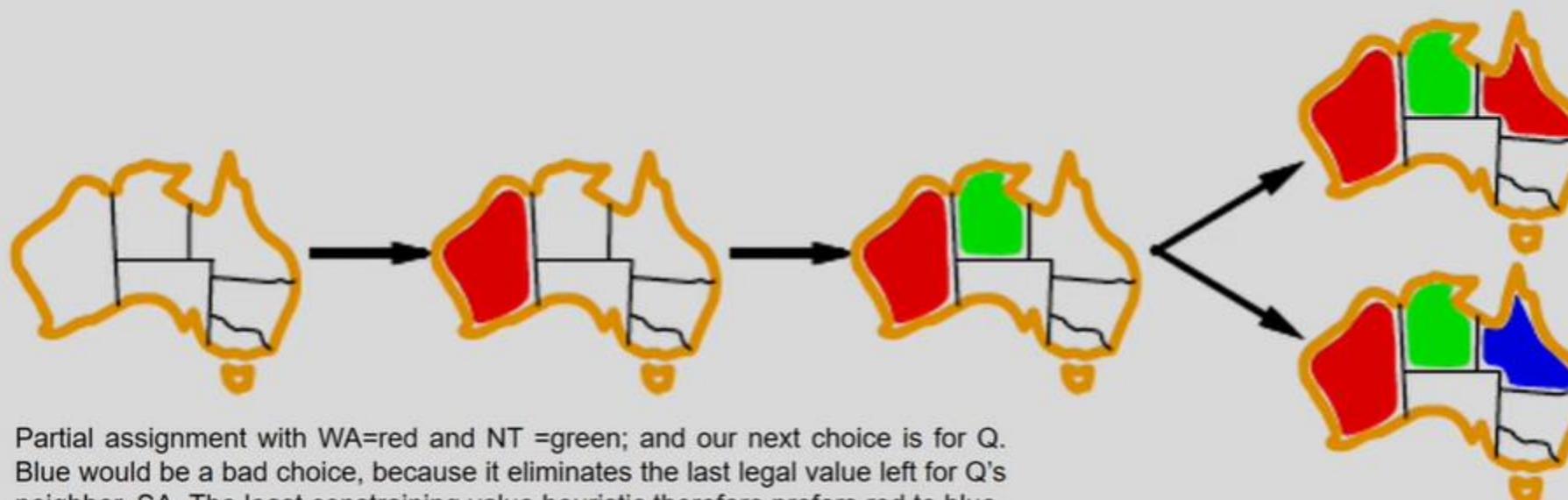
The least-constraining-value heuristic can be effective!

In general, the heuristic is trying to leave the maximum flexibility for subsequent variable assignments.

Variable and Value Ordering

□ Least Constraining value:

- choose the one that rules out the fewest values in the remaining variables.



Propagating Constraints

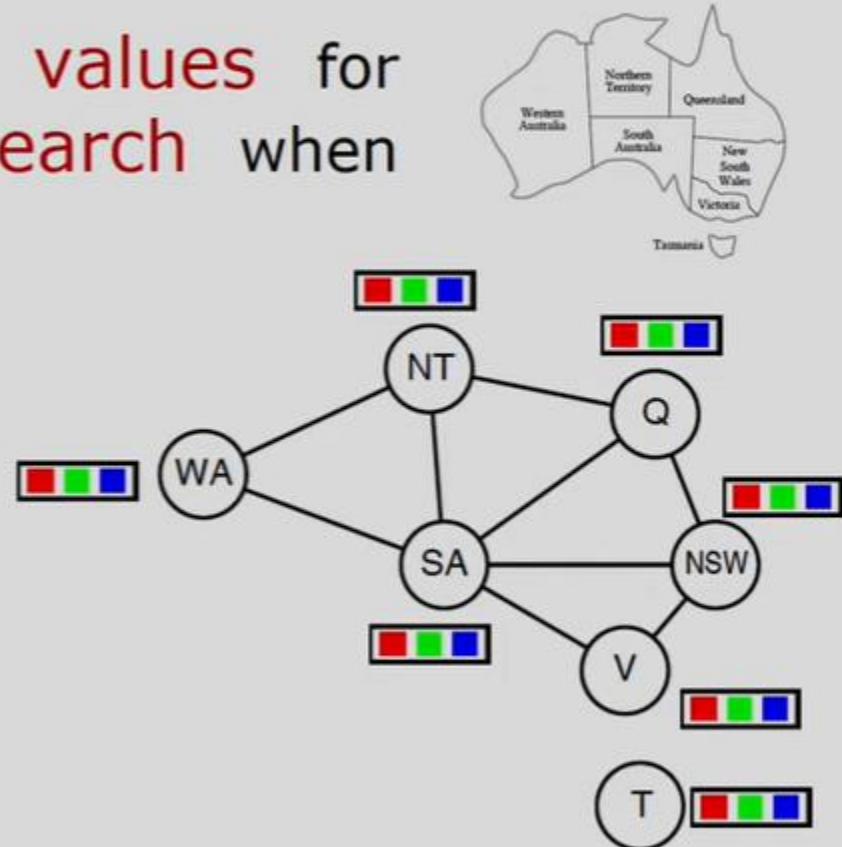
So far our search algorithm considered the constraints on a variable only at the time that the variable is chosen

var SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*], *assignment*, *csp*).

Looking at some of the constraints earlier in the search, or even before the search has started, can drastically reduce the search space.

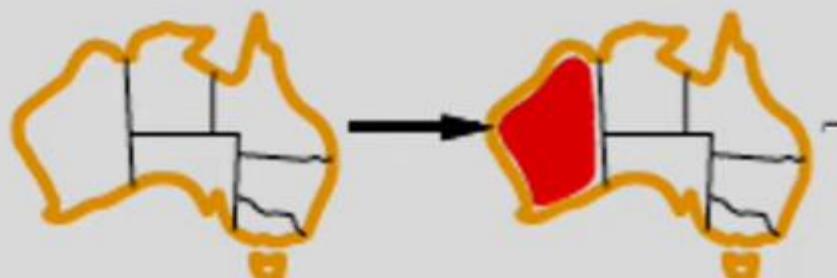
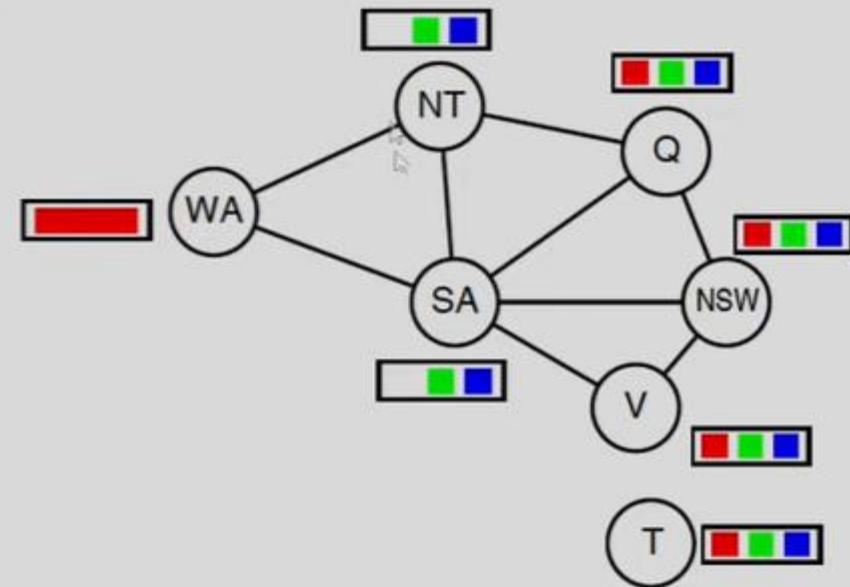
Forward Checking

- Keep track of remaining legal values for unassigned variables. Terminate search when any variable has no legal values.



Forward Checking

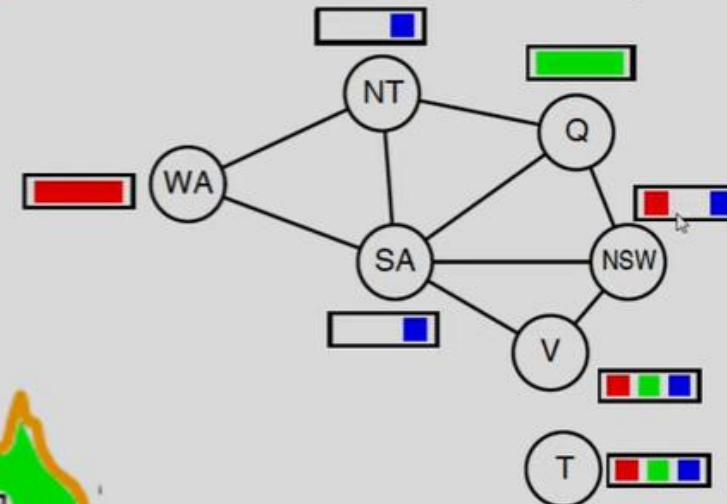
- Keep track of remaining legal values for unassigned variables. Terminate search when any variable has no legal values.



Forward Checking



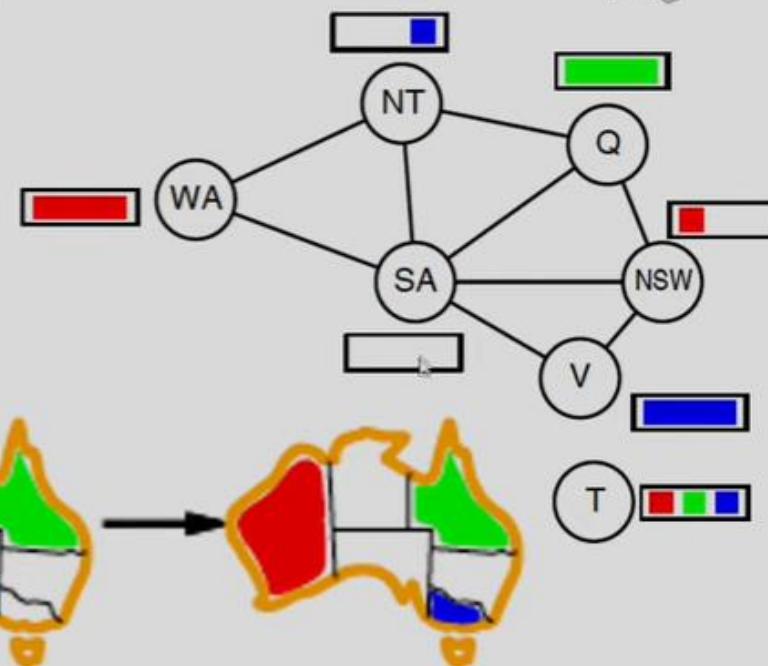
- Keep track of remaining legal values for unassigned variables. Terminate search when any variable has no legal values.



Forward Checking

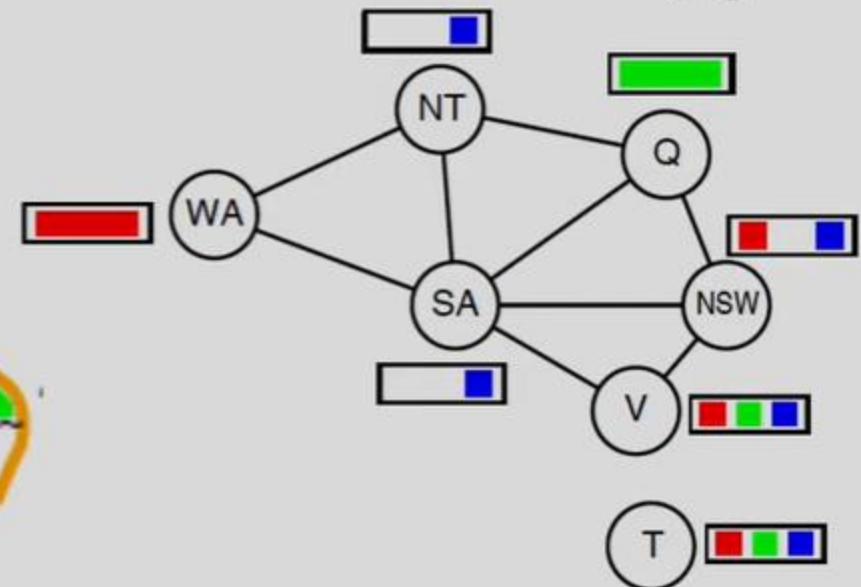


- Keep track of remaining legal values for unassigned variables. Terminate search when any variable has no legal values.



Constraint Propagation

Propagate the implications of a constraint on one variable onto other variables to detect inconsistency.



NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally.

FC propagates information, but doesn't provide early detection for all failures:

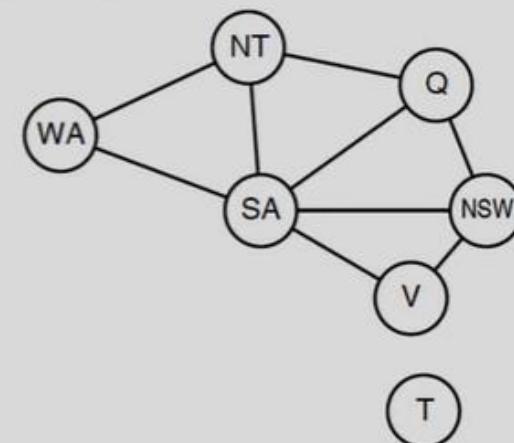
Constraint Propagation

- Constraint propagation is the general term for propagating the implications of a constraint on one variable onto other variables.
- And we want to do this fast: it is no good reducing the amount of search if we spend more time propagating constraints than we would have spent doing a simple search.
- Arc Consistency - a fast method of constraint propagation that is substantially stronger than forward checking.

Arc Consistency

- The idea of arc consistency provides a fast method of constraint propagation that is substantially stronger than forward checking.
 - Here, *arc* refers to a directed arc in the constraint graph.
 - For example the arc from SA to NSW.

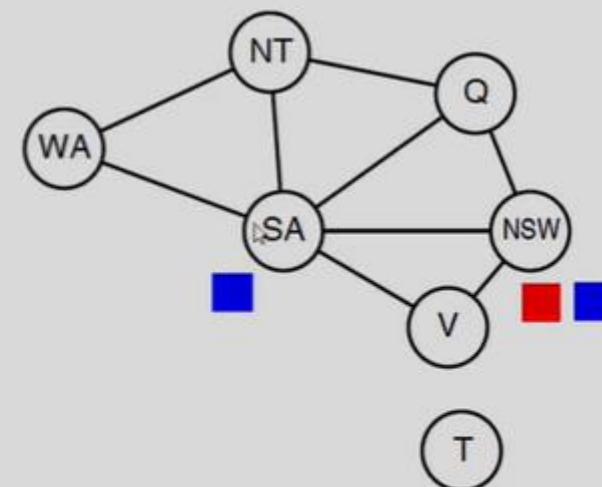
Arc $X \rightarrow Y$ is consistent iff for every value v_x of X there is some allowed value v_y for Y



Arc Consistency

- The idea of arc consistency provides a fast method of constraint propagation that is substantially stronger than forward checking.
 - Here, *arc* refers to a **directed arc** in the constraint graph.
 - For example the arc from SA to NSW.

Given the **current domains** of SA and NSW, the **arc** is **consistent** if, for **every** value v_x of SA, there is some value v_y of NSW that is **consistent** with v_x .

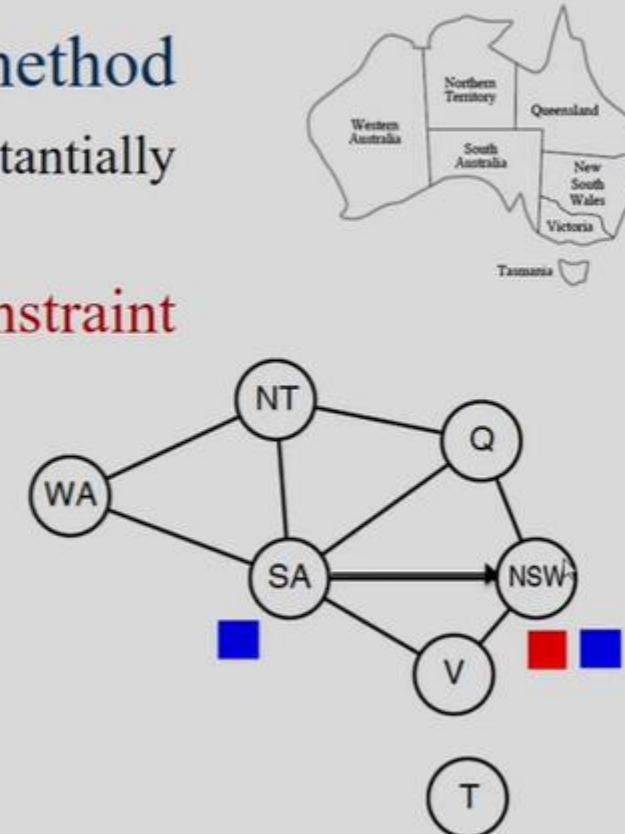


Current domains of SA and NSW are {blue} and {red; blue} respectively. For SA=blue, there is a consistent assignment for NSW, namely, NSW =red; therefore, the arc from SA to NSW is consistent.

Arc Consistency

- The idea of arc consistency provides a fast method of constraint propagation that is substantially stronger than forward checking.
 - Here, *arc* refers to a **directed arc** in the constraint graph.
 - For example the arc from SA to NSW.

Given the **current domains** of SA and NSW, the **arc** is **consistent** if, for **every** value v_x of SA, there is some value v_y of NSW that is **consistent** with v_x .



Current domains of SA and NSW are {blue} and {red; blue} respectively. For SA=blue, there is a consistent assignment for NSW, namely, NSW =red; therefore, the arc from SA to NSW is consistent.

Arc Consistency



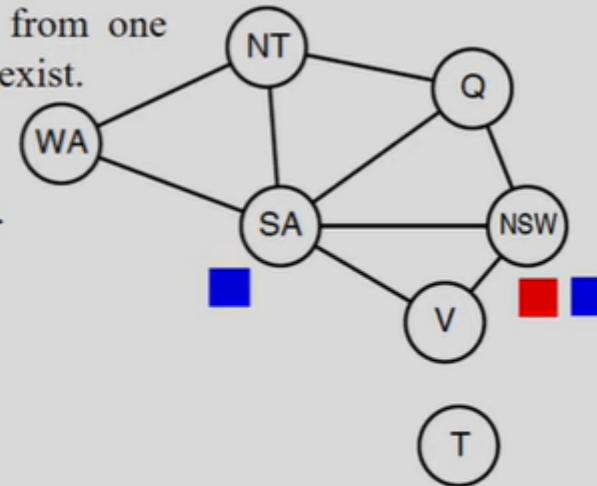
- The idea of arc consistency provides a fast method of constraint propagation that is substantially stronger than forward checking.
- Full Arc Consistency Lookahead

■ Considers all pairs of future variables and removes values from one domain for which a consistent value in the other domain does not exist.

- Directional Arc Consistency Lookahead
- Considers only directional arc consistency for the future variables.

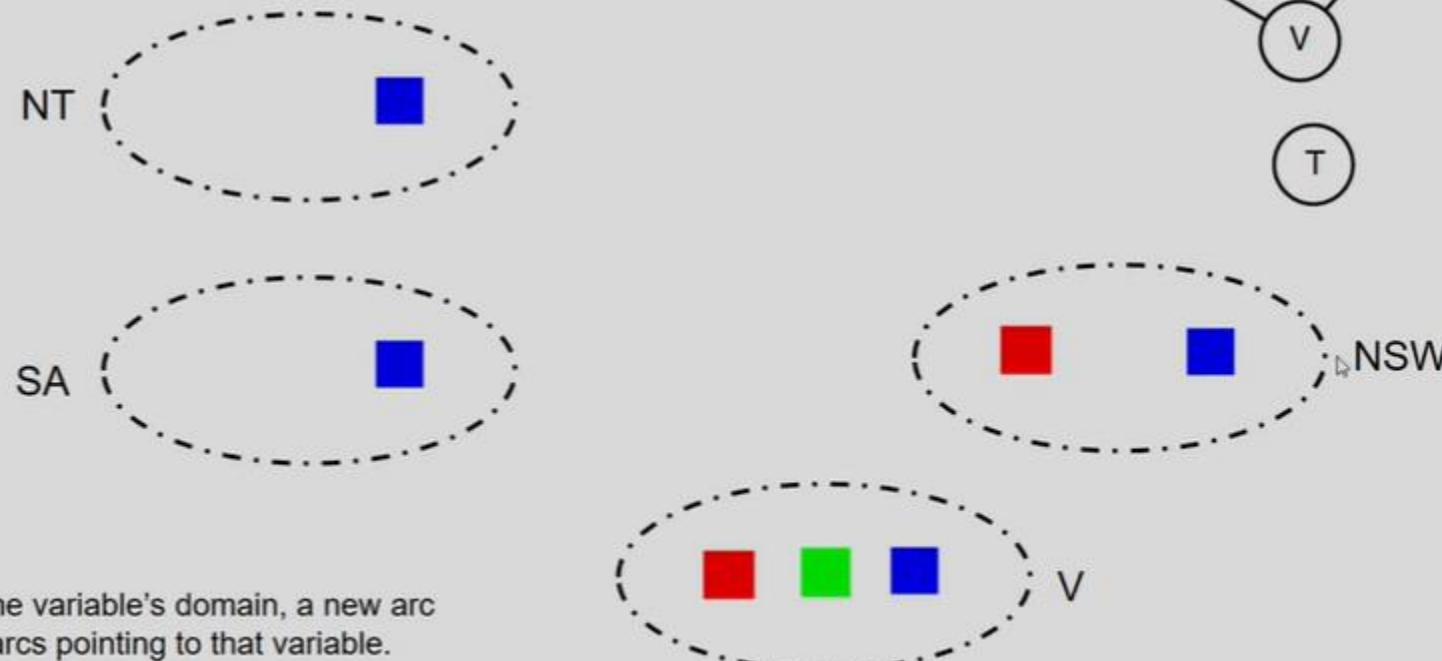
Given the current domains of SA and NSW, the arc is consistent if, for every value v_x of SA, there is some value v_y of NSW that is consistent with v_x .

Arc consistency Lookahead for search space reduction



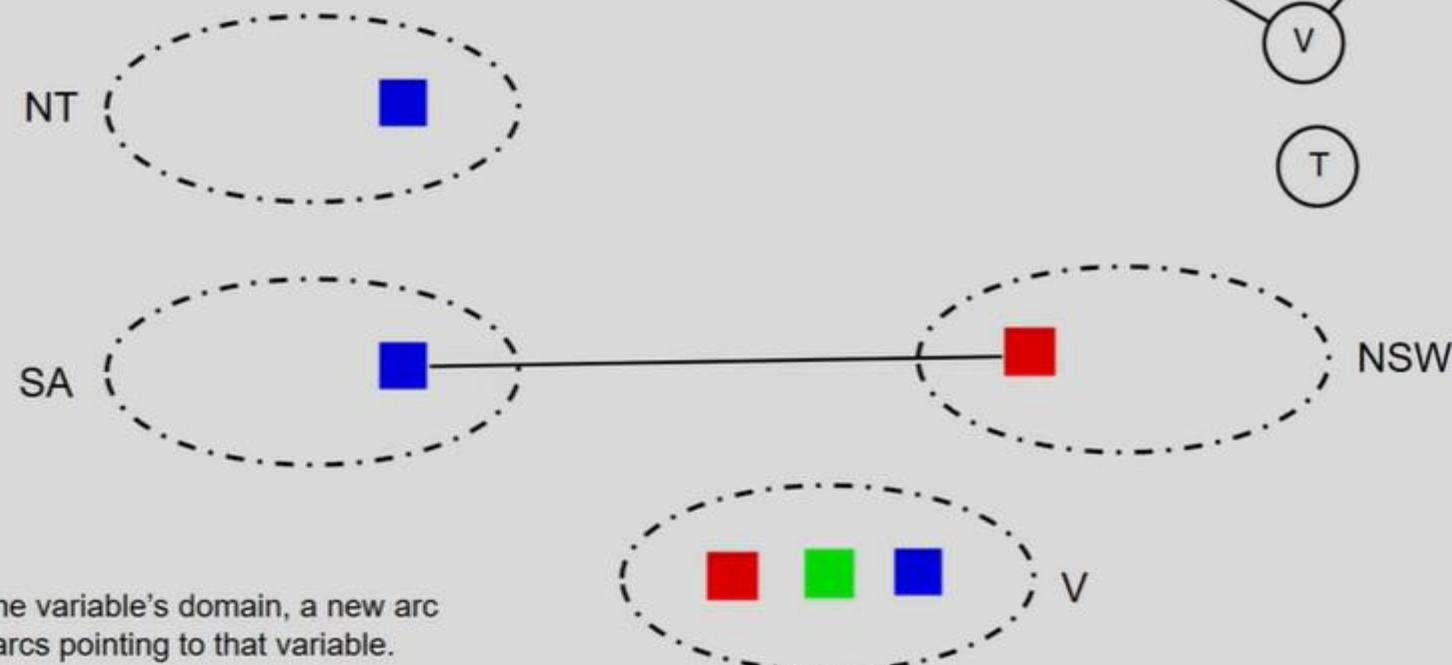
Arc Consistency

Arc consistency **must be applied repeatedly** until no more inconsistencies remain.



Arc Consistency

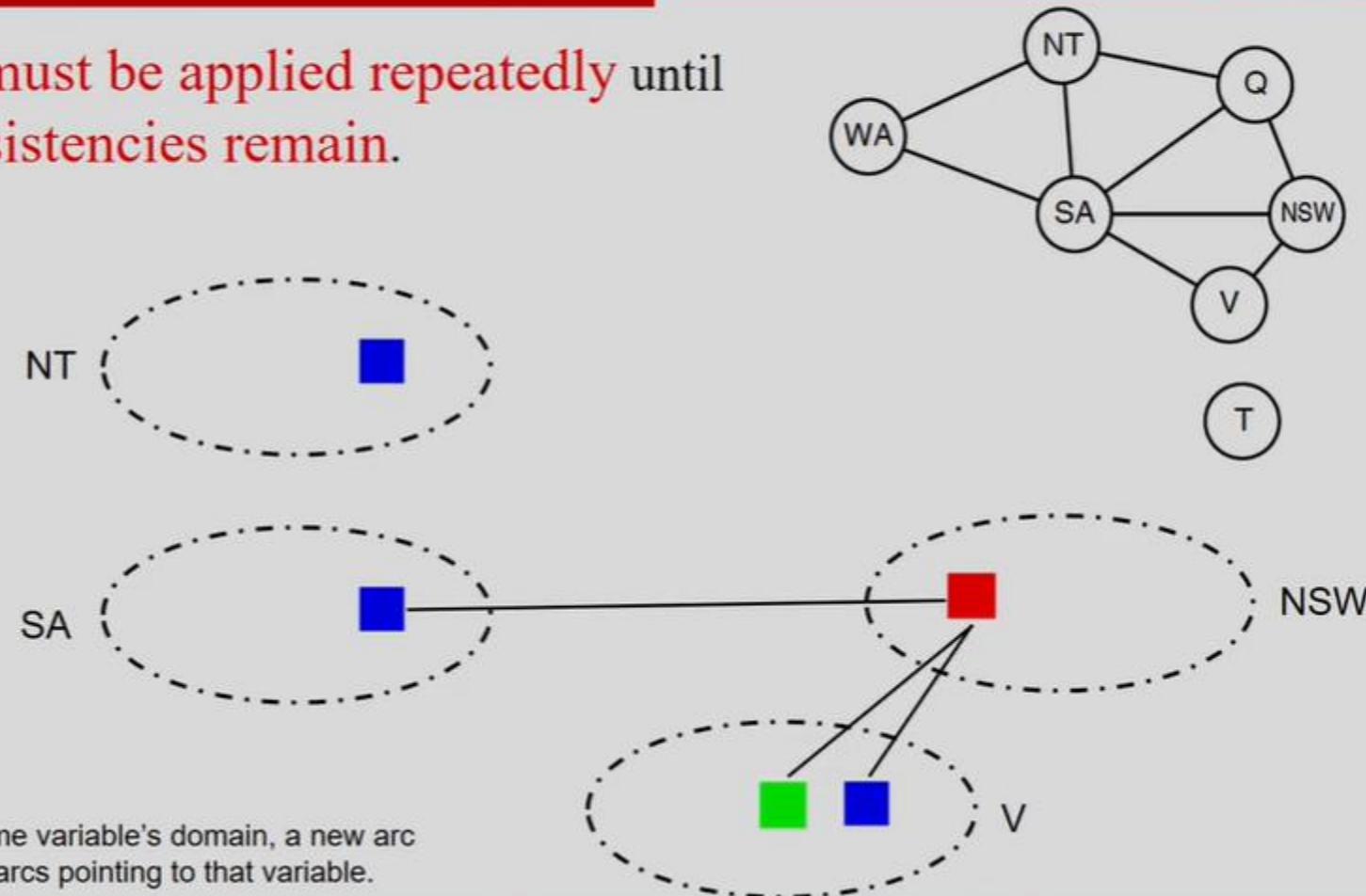
Arc consistency **must be applied repeatedly** until no more inconsistencies remain.



For a value deleted from some variable's domain, a new arc inconsistency could arise in arcs pointing to that variable.

Arc Consistency

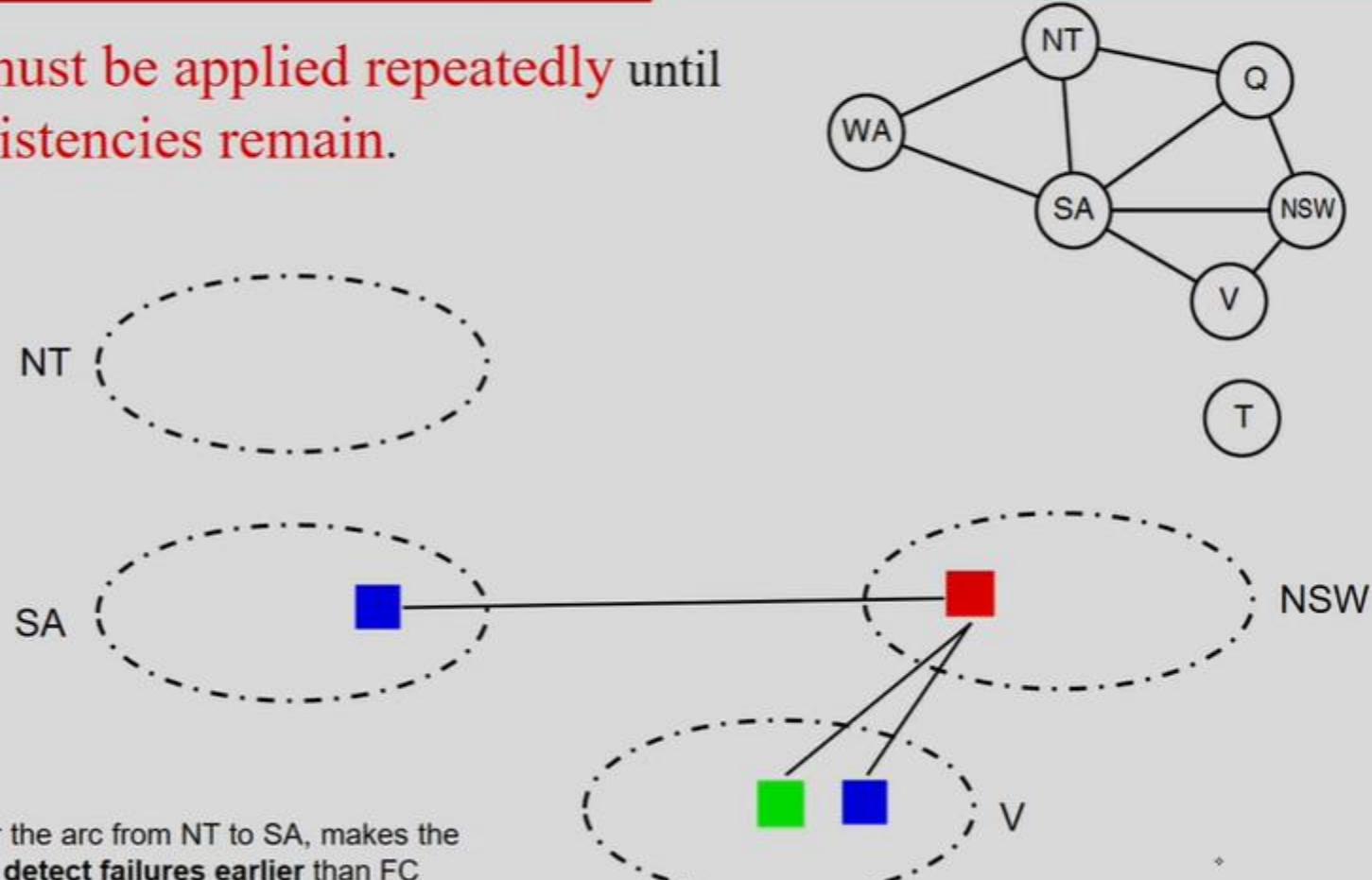
Arc consistency must be applied repeatedly until no more inconsistencies remain.



For a value deleted from some variable's domain, a new arc inconsistency could arise in arcs pointing to that variable.

Arc Consistency

Arc consistency must be applied repeatedly until no more inconsistencies remain.



Crypto-Arithmetic Problem

It is a mathematical puzzle, where the numbers are represented by letters or symbols. Each letter represents a unique digit.

The goal is to find the digits such that a given mathematical equation is verified. In general, there are few variables in form of letters which is to be assigned numeric values in the range of 0 to 9, such that the given equation hold true.

To solve crypto-arithmetic problem, we have to generate the constraints by observing the given equation. Then we may be able to assign a single final value to one of the variable or can reduce the range of possibly allowed values for a particular variable. That helps in leading towards the solution.

From the above discussion one thing is clear that there is no specific algorithm to solve crypto-arithmetic, but it's a trial and error method based on backtracking strategy. Following example illustrates the procedure for solving crypto-arithmetic problems.

Crypto-Arithmetic Problem

$$\begin{array}{r} \text{T W O} \\ + \text{T W O} \\ \hline \text{FOUR} \end{array}$$

F	1
O	4
U	6
R	8
T	7
W	3

Can I take T=5?

Can I take T=6?

Can I take w=2?

Can I take W=4?

Crypto-Arithmetic Problem

Solve crypt arithmetic problem

$$\begin{array}{r} \text{EAT} \\ + \text{ THAT} \\ \hline \text{APPLE} \end{array}$$

A	1
T	9
P	0
E	8
H	2
L	3

Adversarial Search

Adversarial Search Problem is having competitive activity which involves 'n' players and it is played according to certain set of protocols.

- Game is called adversarial because there are agents with conflicting goals and the surrounding environment in a game is competitive as there are 'n' players or agents participating.
- We say that goals are conflicting and environment is competitive because every participant wants to win the game.
- From above explanation it is understandable that we are dealing with a competitive multi agent environment.
- As the actions of every agent are unpredictable there are many possible moves/actions.
- In order to play a game successfully every agent in environment has to first analyze the action of other agents and how other agents are contributing in its own winning or loosing. After performing this analysis agent executes.

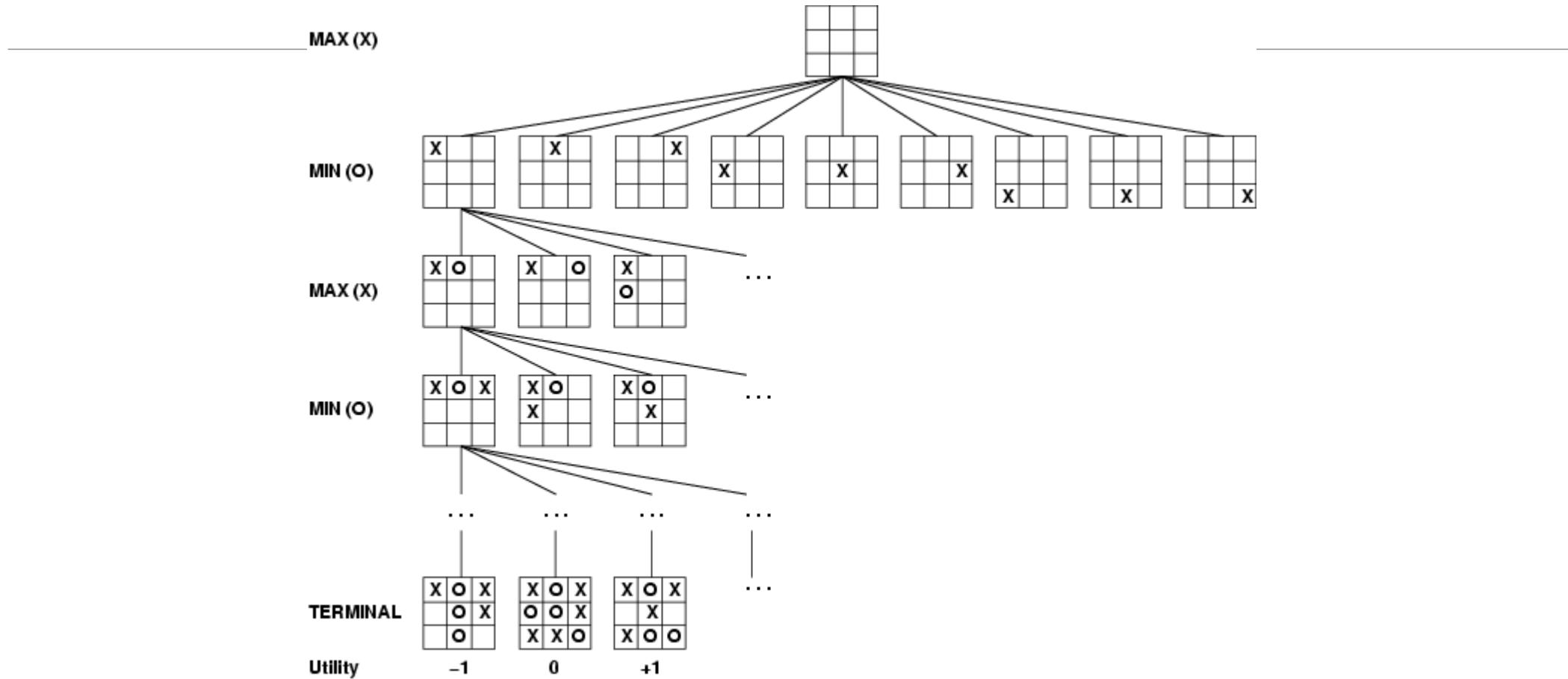
What is Game Tree? Dec 15 MU

Game tree is defined as a directed graph with nodes and edge. Here nodes indicate positions in a game and edges indicate next actions.

A game with two players (MAX and MIN, MAX moves first, taking turn) can be defined as a search problem with the following:

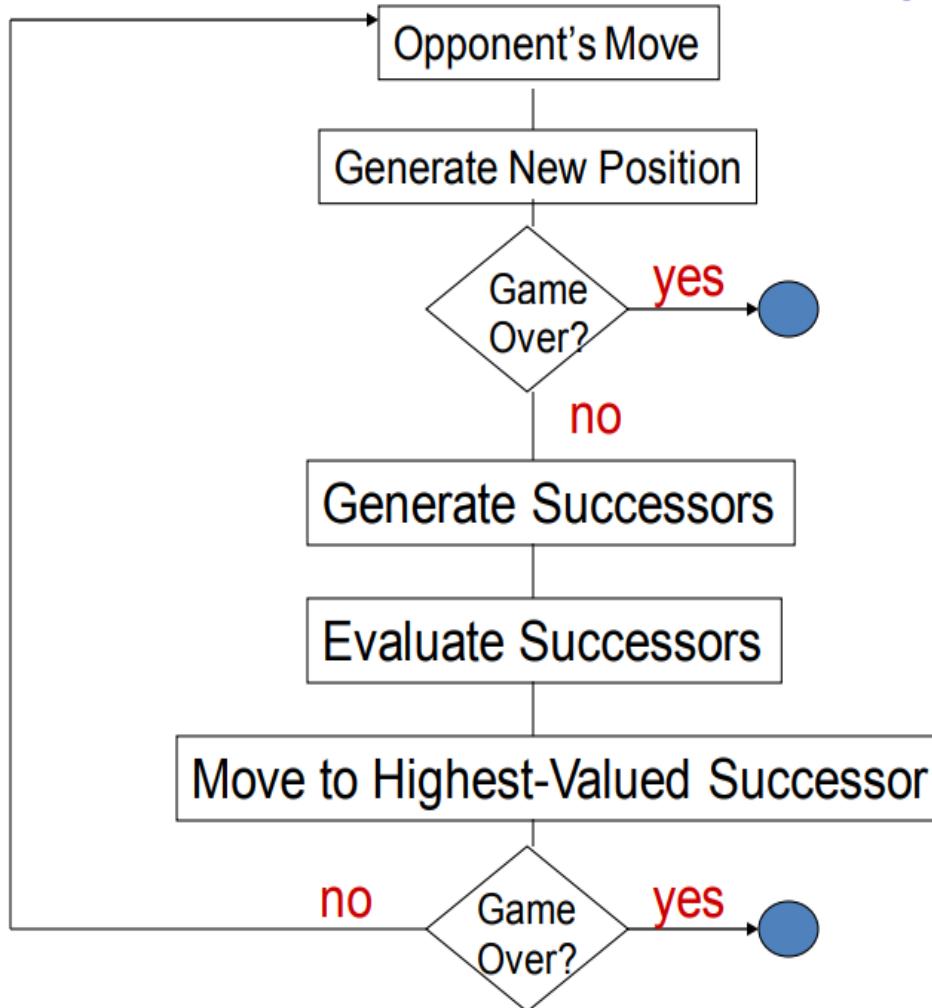
1. Initial state: Initial state consists of the position of the board and shows whose move it is.
2. Successor function: Successor function defines of the legal moves a player can make.
3. Terminal state: Terminal state is the position of the board when the game is over.
4. Goal test: Whether the game is over – Terminal states.
5. Utility function: Utility function is the function that assigns a numeric value for the outcome of a game.
6. Game tree = Initial state + Legal moves.

Game tree Tic-Tac-Toe



Optimal strategies

Two-Player Game



The minmax algorithm

Minimax is a recursive algorithm that is used to choose an optimal move for a player assuming that the other player is also playing optimally.

The players in the game are referred to as MAX and MIN. MAX represents the person who is trying to win the game and hence maximise his or her score. MIN represents the opponent who is trying to minimise the score of MAX.

This algorithm is used to look ahead and decide which move to make first. If the game space is small enough the entire space can be generated and the leaf nodes can be allocated a win (1) or loss (0) value.

The minmax algorithm

These values might be propagated back up the tree to decide which node to use. In propagating the values back up the tree, a MAX node is appointed the maximum value of all its children and MIN node is appointed the minimum values of all its children.

```
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)
```

The minmax algorithm

ALGORITHM

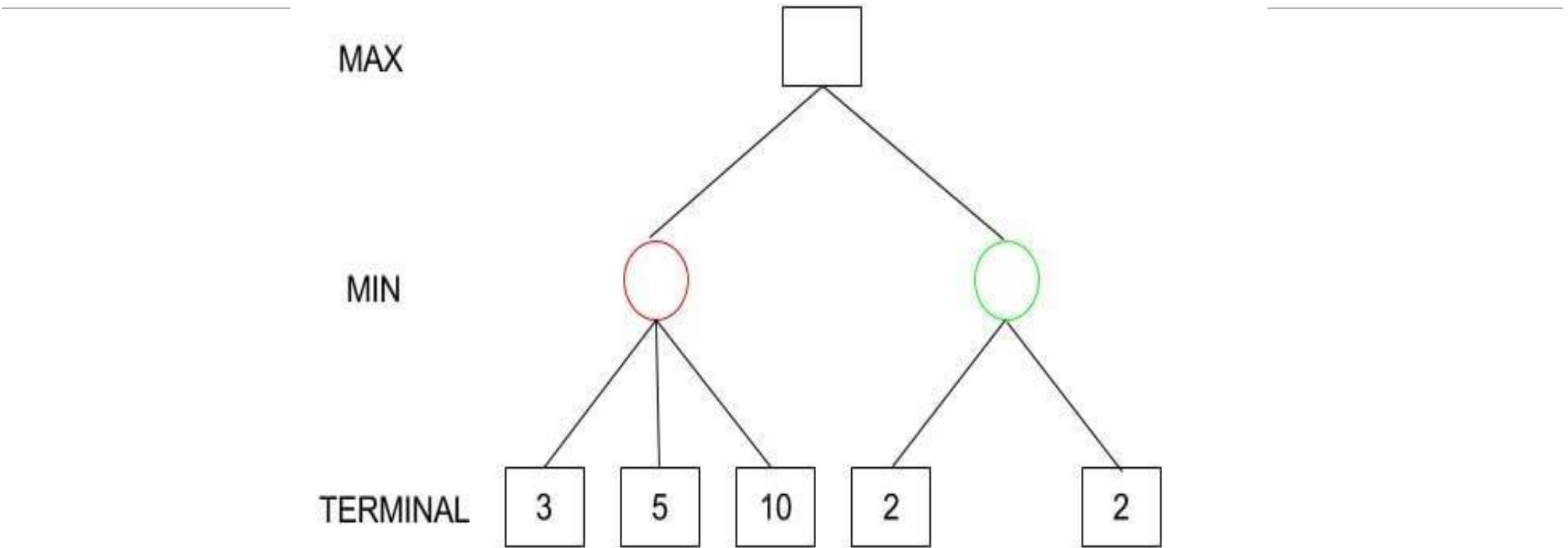
Step 1: In case of search reaching its limit, the static value of the current position relative to the appropriate player is calculated. The result is reported.

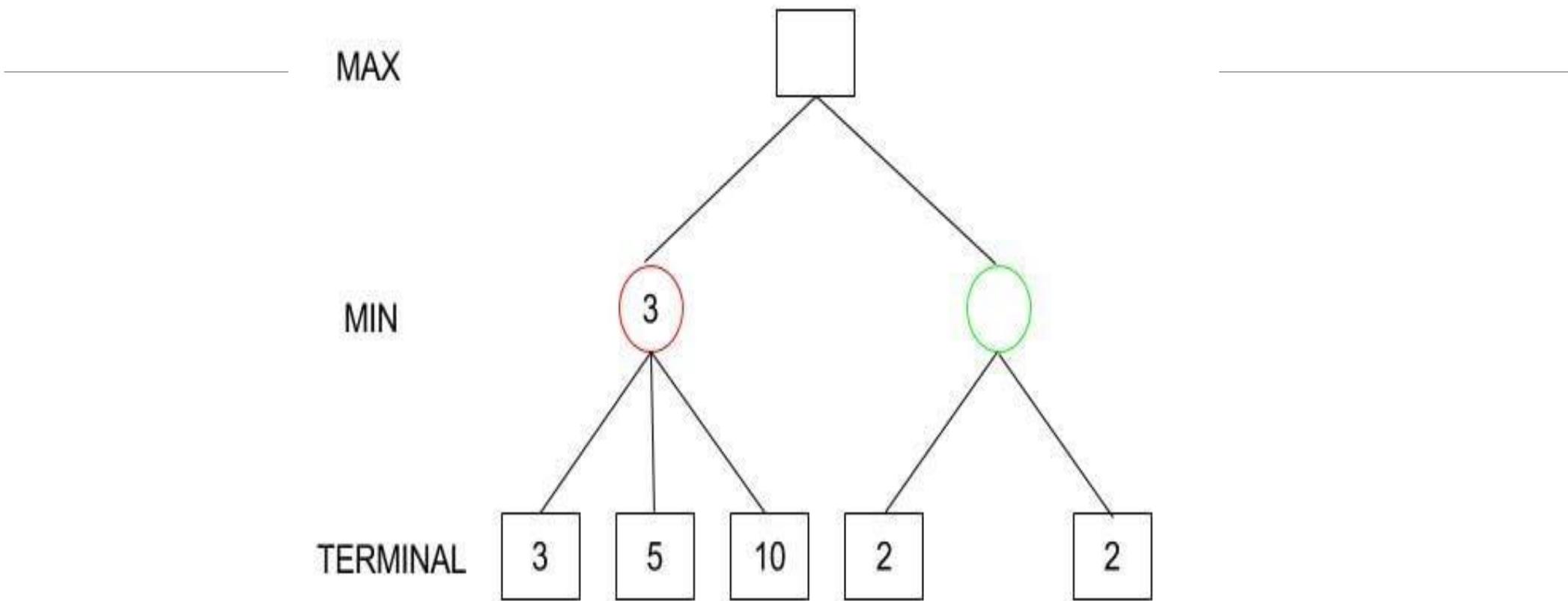
Step 2: Otherwise, if the level is a minimising level, use the minimax on the children of the current position. The minimum value of the results is reported here

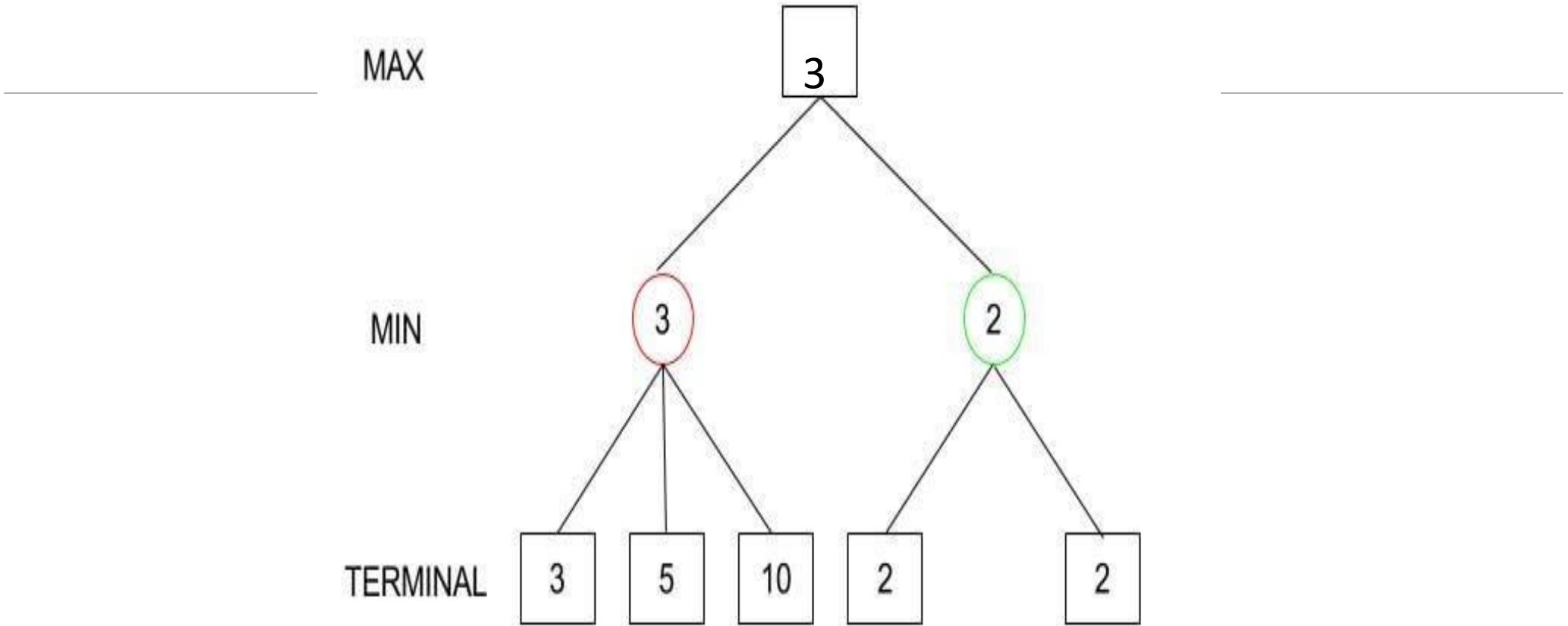
Step 3: Also, if the level is a maximising level, use the minimax on the children of the current position. The maximum of the results is reported here.

Step 4: The utility values is calculated with the help of leaves considering one layer at a time until the root of the tree.

Step 5: Eventually, all the backed-up values reach to the root of the tree, that is, the topmost point. At that point, MAX is responsible for choosing the highest value.



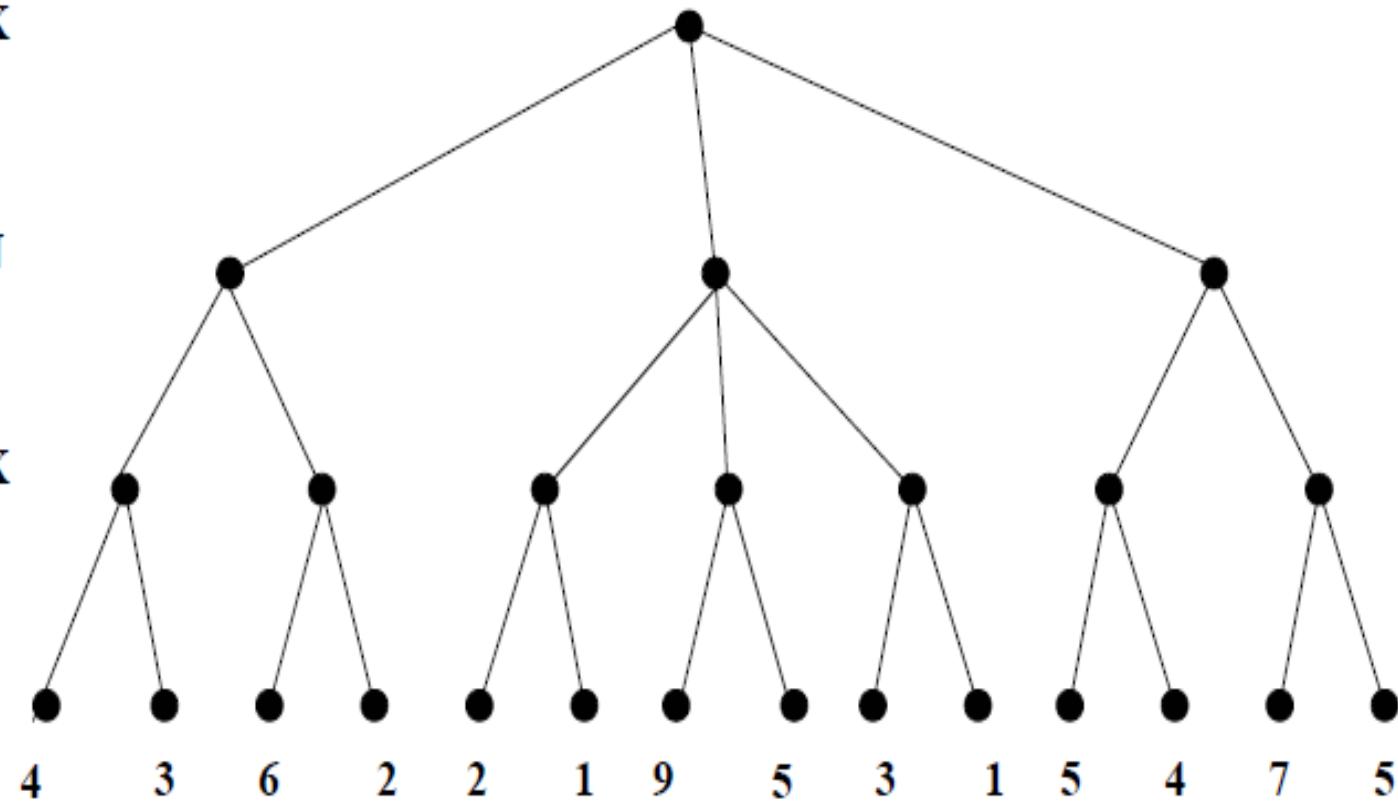




MAX

MIN

MAX



MAX

MIN

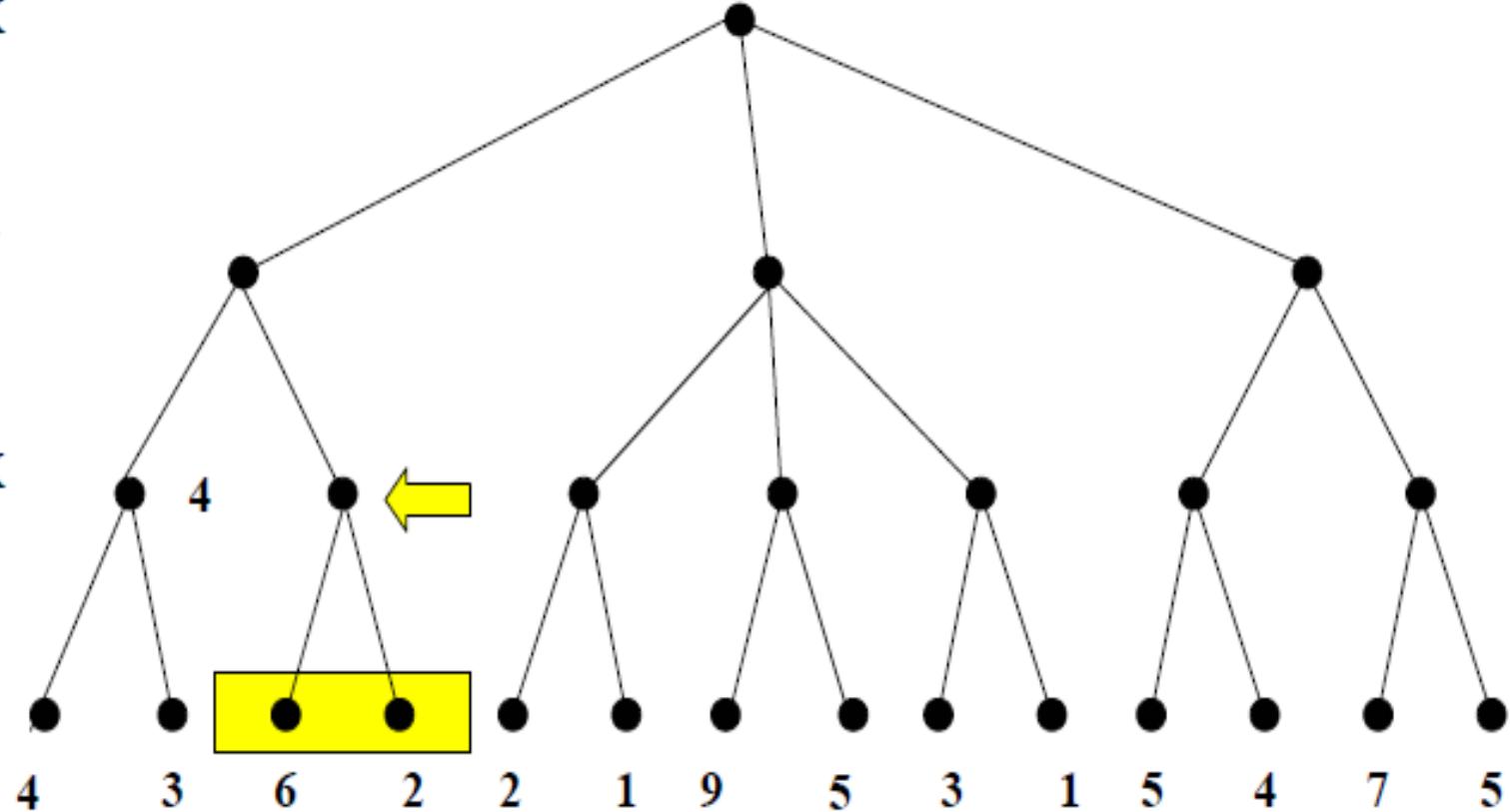
MAX



MAX

MIN

MAX

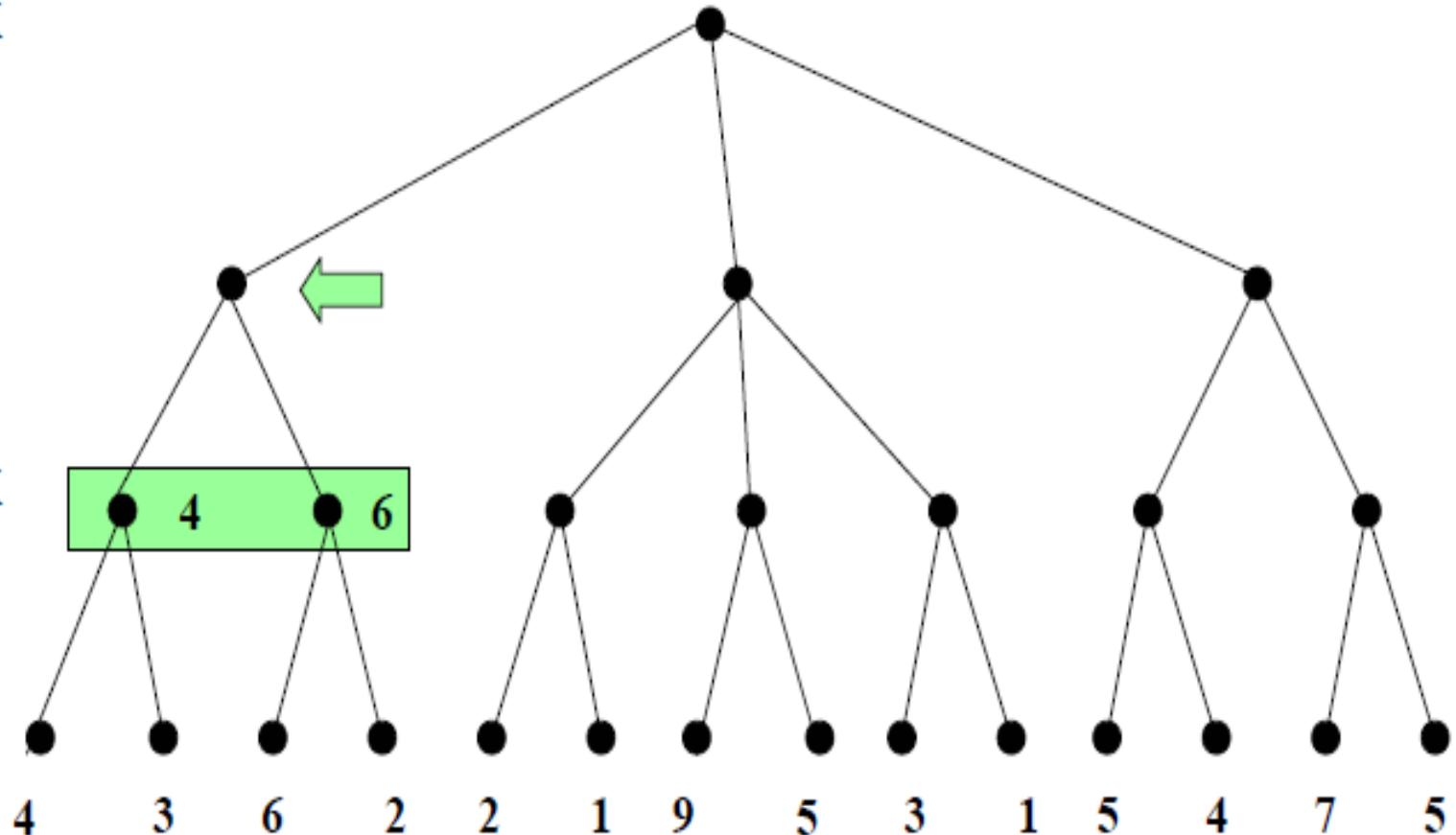


MAX

MIN

MAX

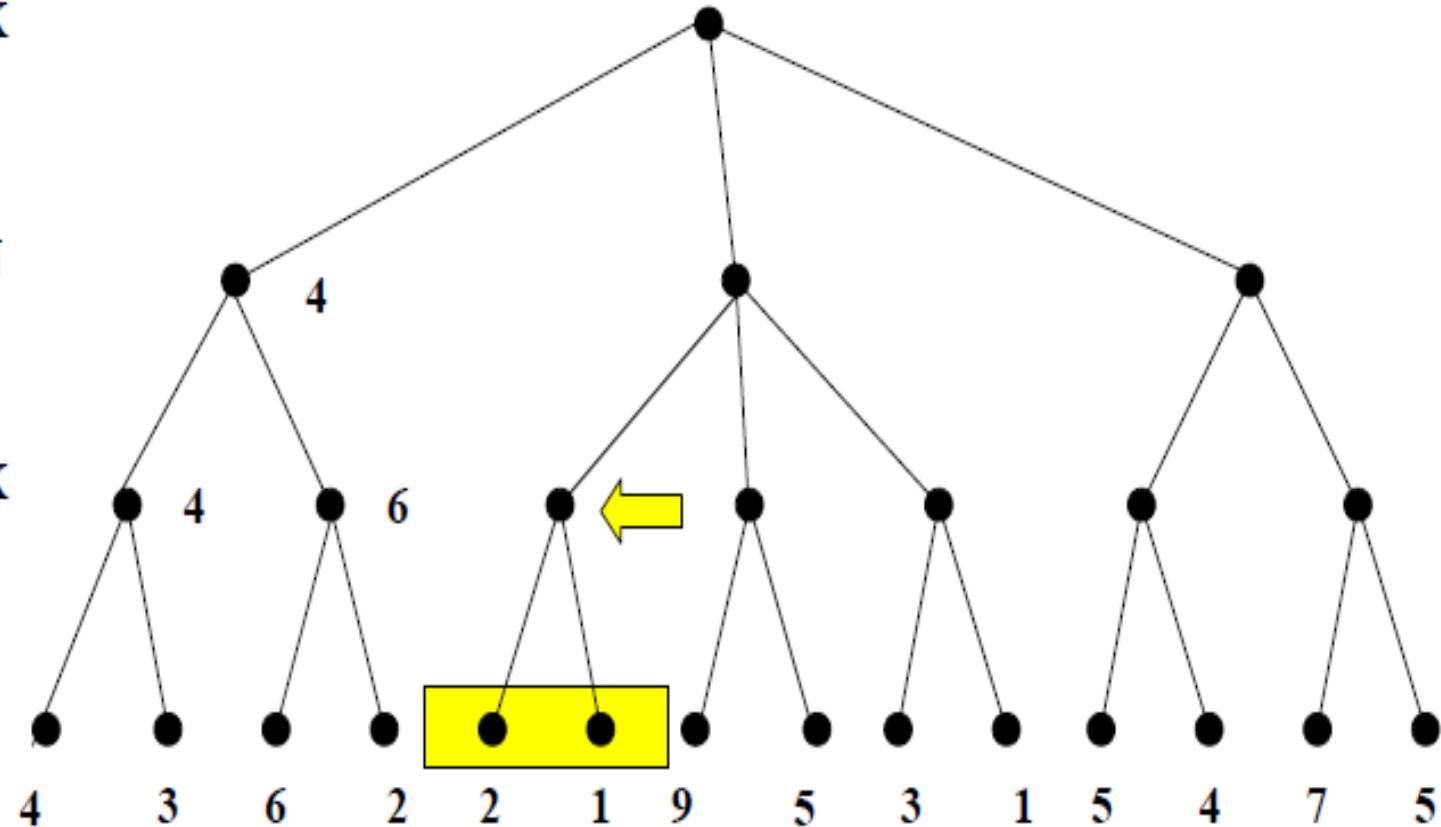
4 6



MAX

MIN

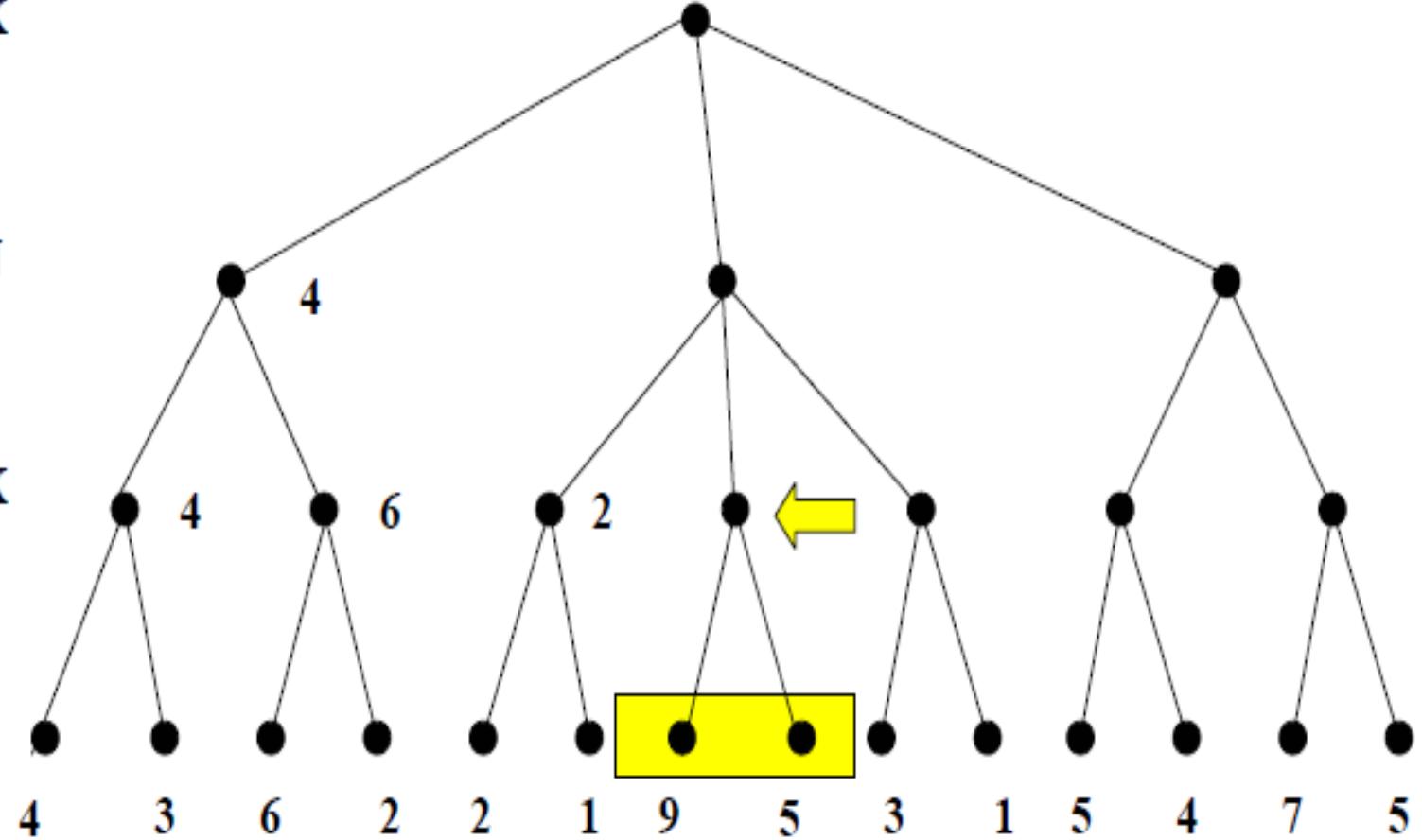
MAX



MAX

MIN

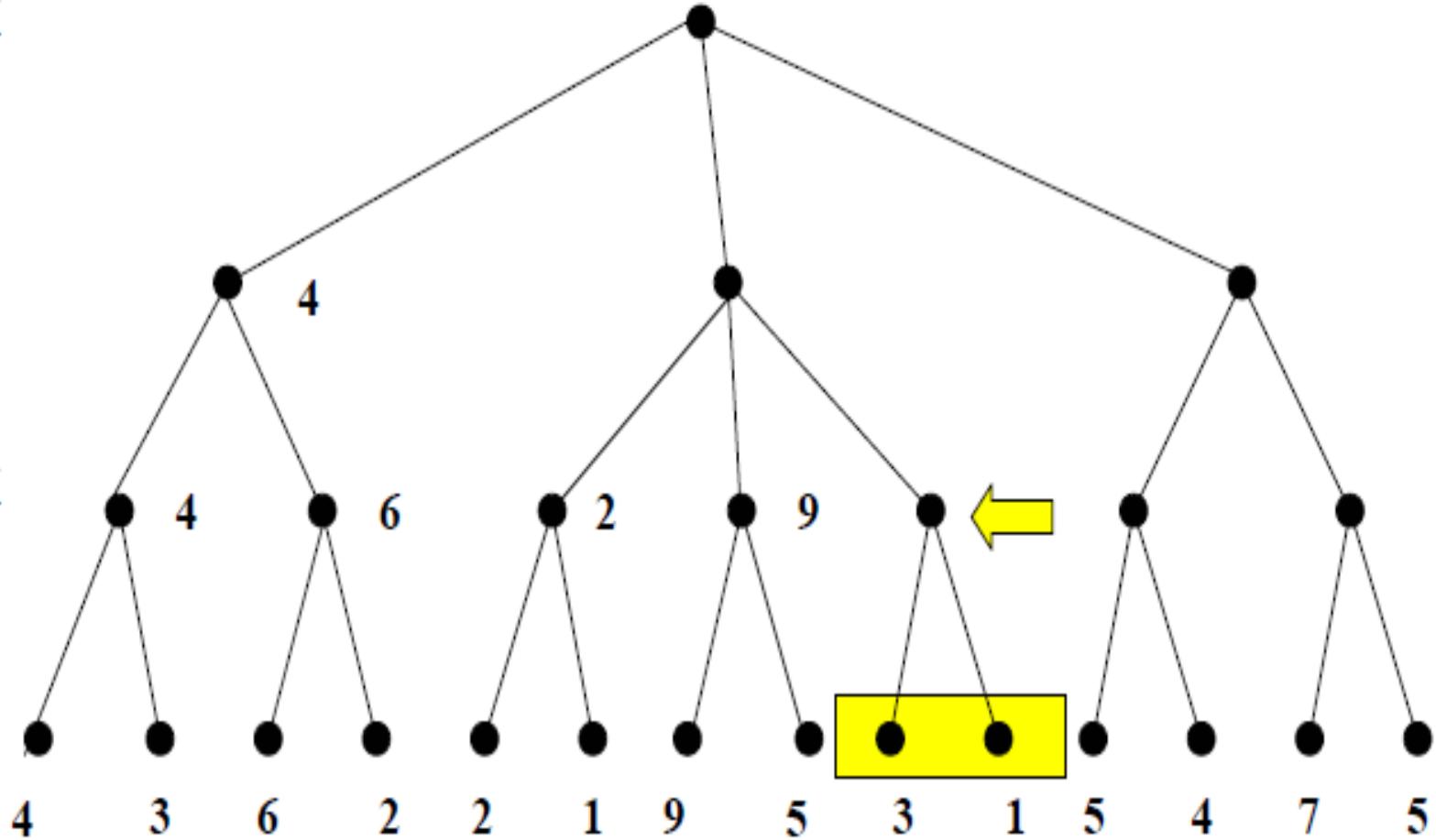
MAX



MAX

MIN

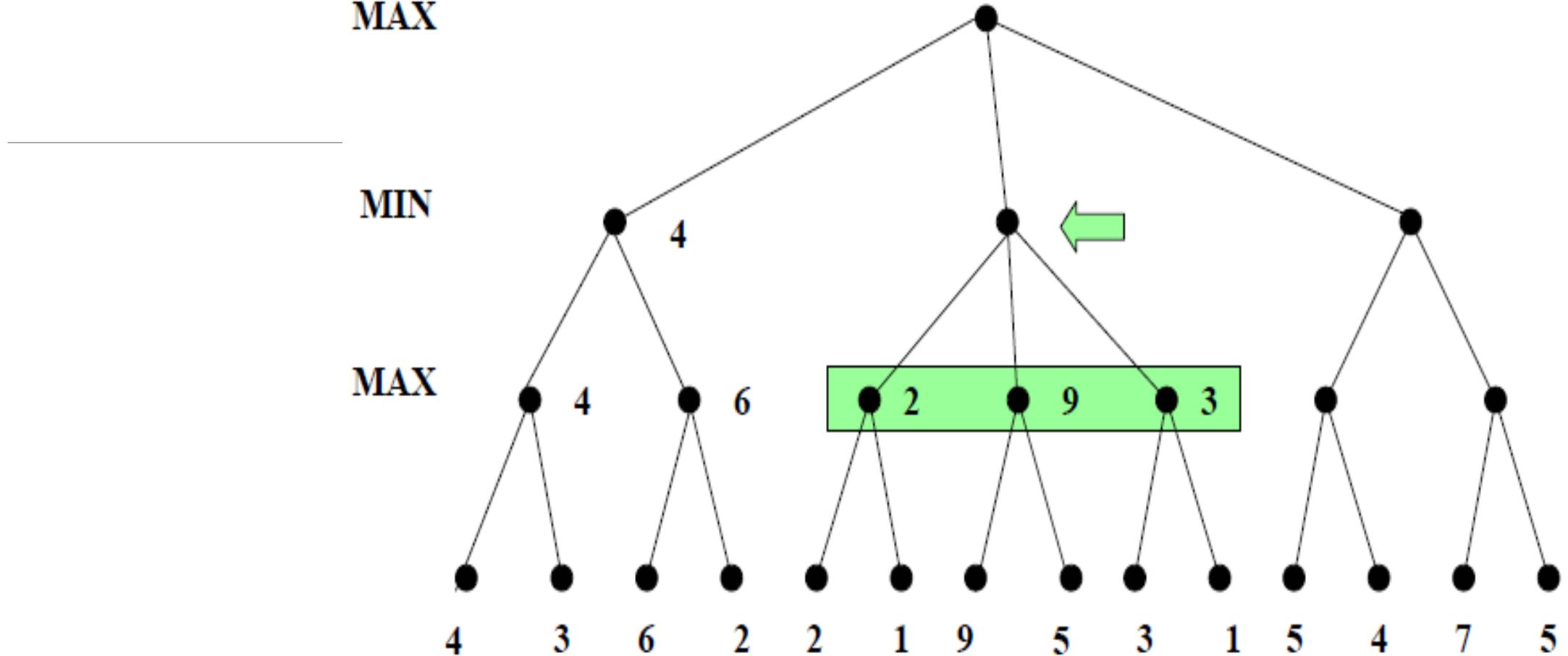
MAX

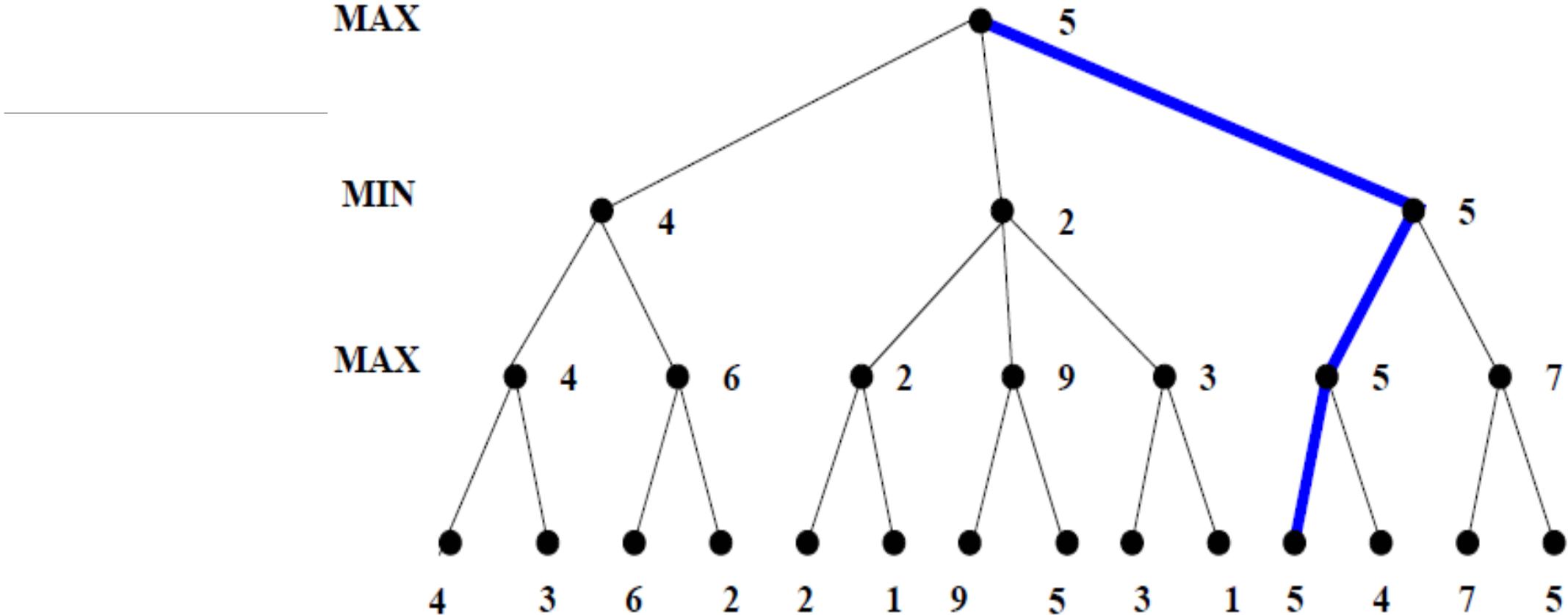


MAX

MIN

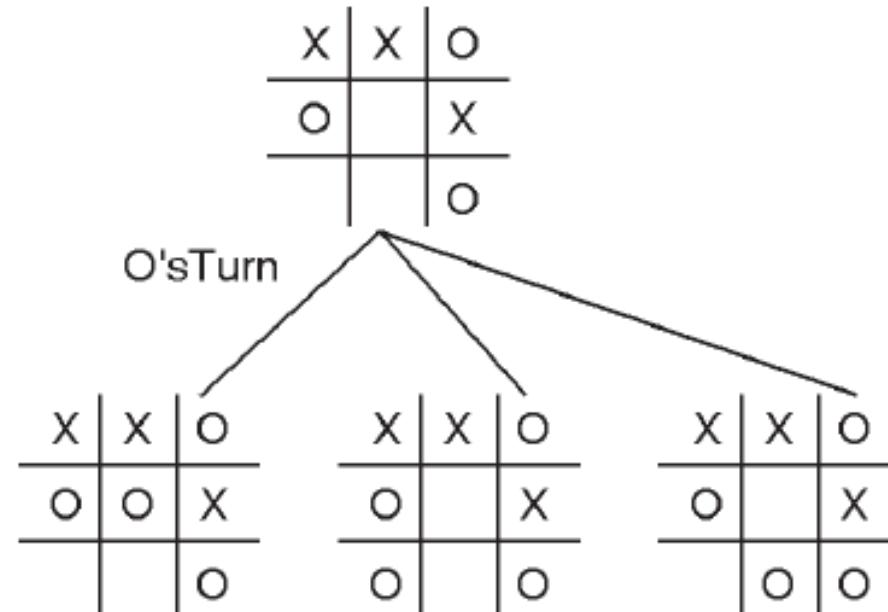
MAX



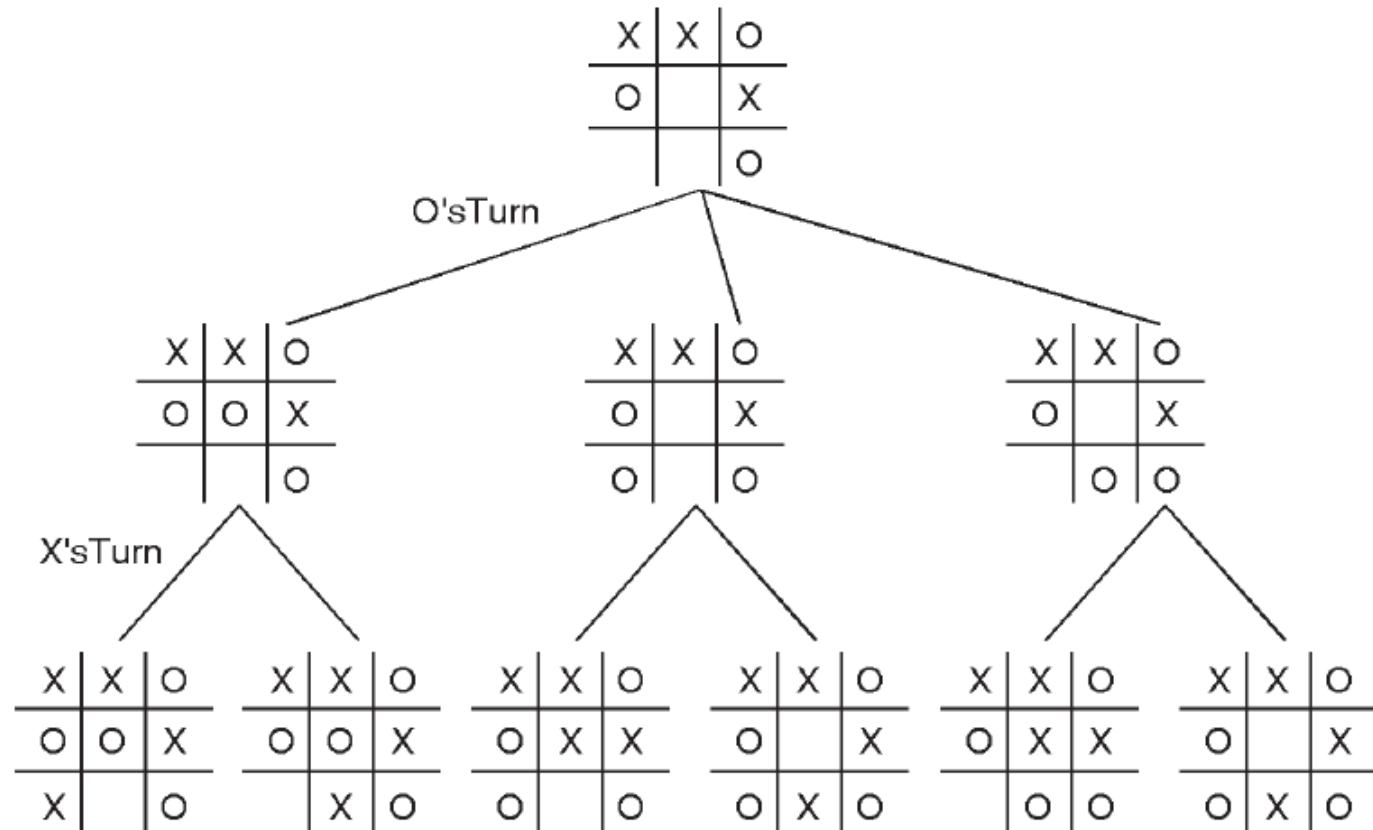


tic-tac toe game with minimax algorithm

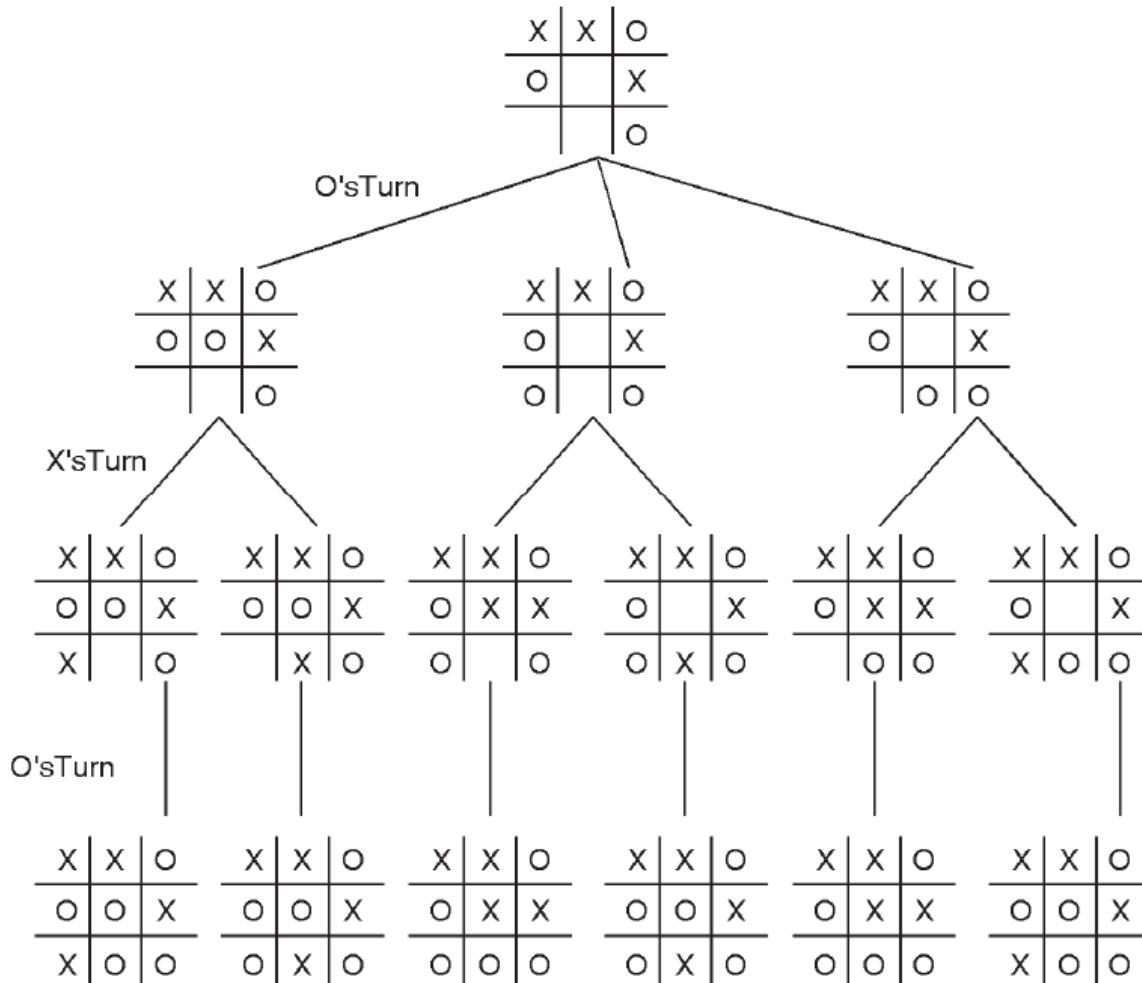
Start action : 'O'

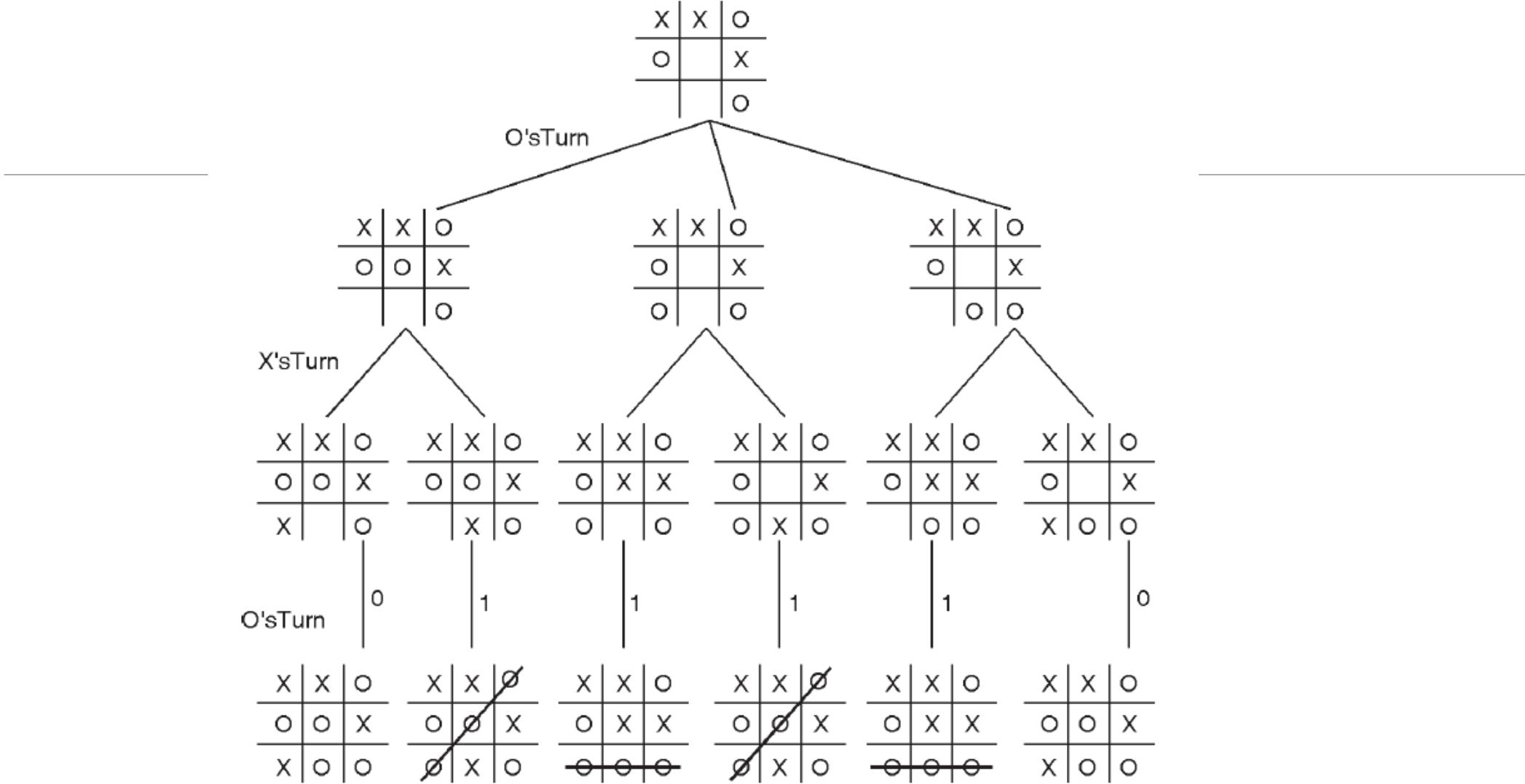


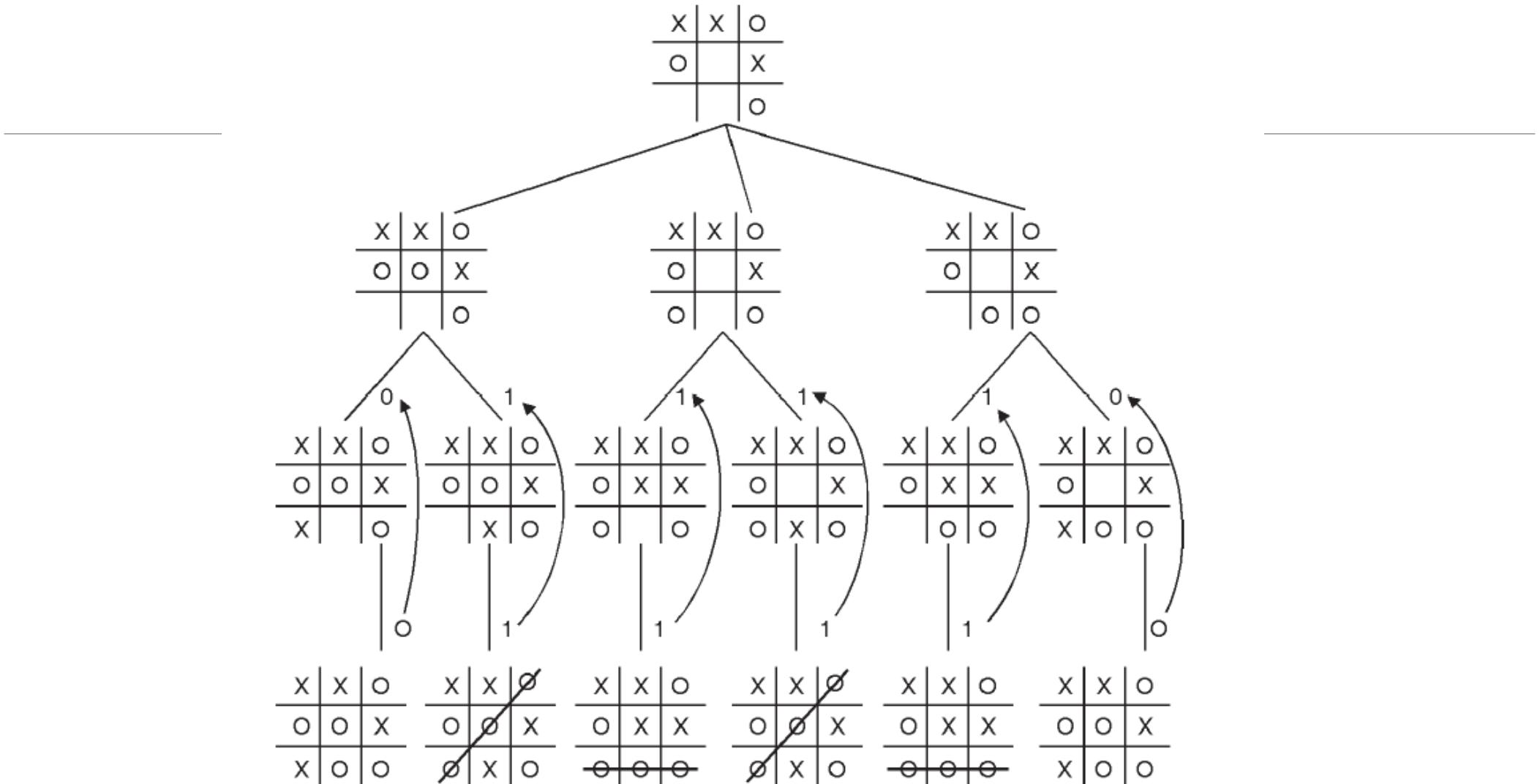
Next action : 'X'

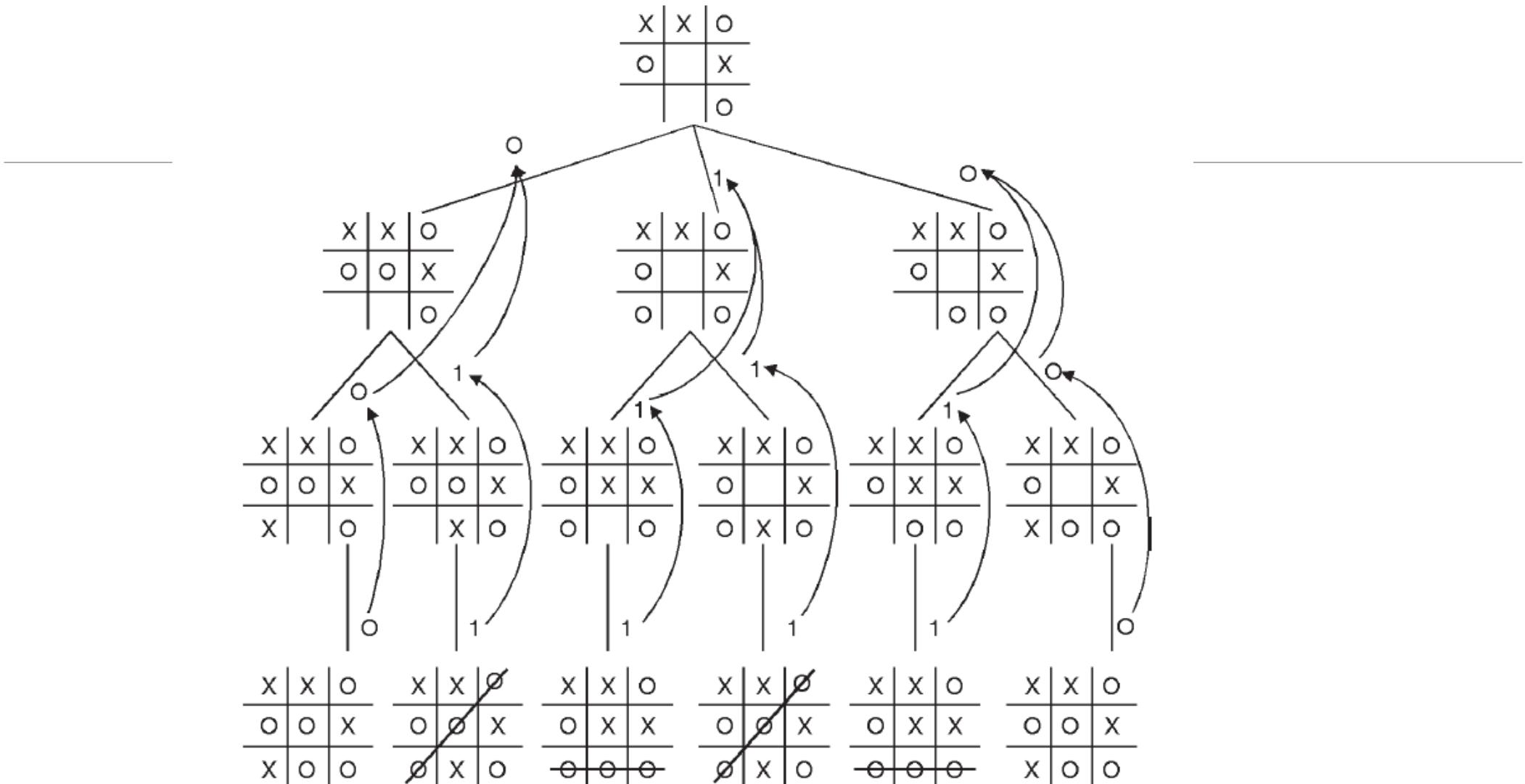


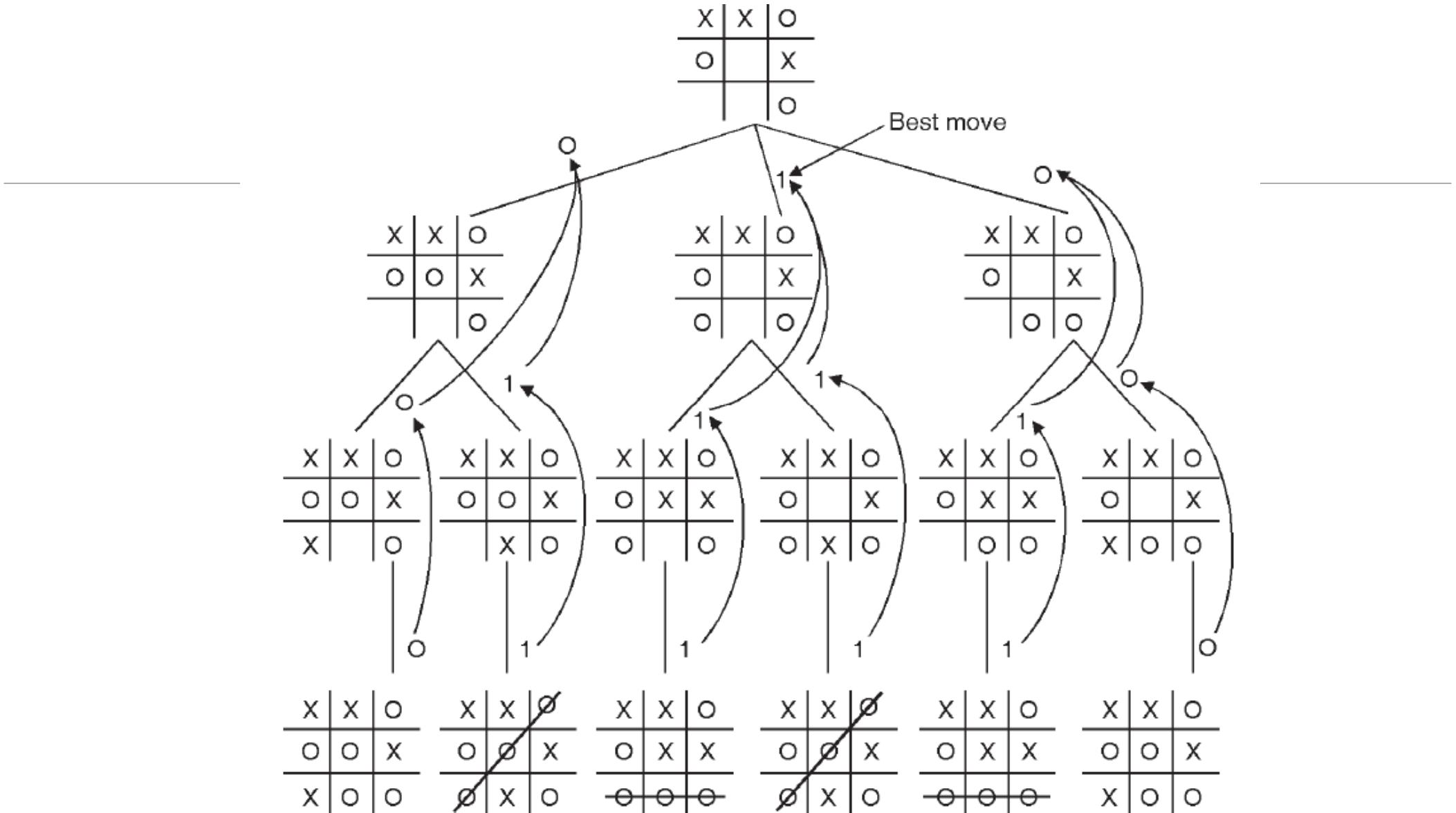
Next action : 'O'











Alpha Beta Pruning

Pruning means cutting off. In game search it resembles to clipping a branch in the search tree, probably which is not so fruitful.

- α - β pruning is an extension to minimax algorithm where, decision making process need not consider each and every node of the game tree.
- Only the important nodes for quality output are considered in decision making. Pruning helps in making the search more efficient.
- **Pruning** keeps only those parts of the tree which contribute in improving the quality of the result remaining parts of the tree are removed.

Alpha-beta pruning

Alpha is the best choice so far for player MAX. The highest possible value for this is demanded here.

Beta proves to be the best choice so far for MIN, and it has to be the lowest possible value.

ALGORITHM

Alpha is set to -infinity and beta to infinity.

If the node is a leaf node, the value is returned.

If the node is a min node, then for each children the minimax algorithm is applied with the alpha–beta pruning.

If the value returned by a child is less, the beta sets beta to this value.

If at any stage, beta is less than or equal to alpha do not examine any more children.

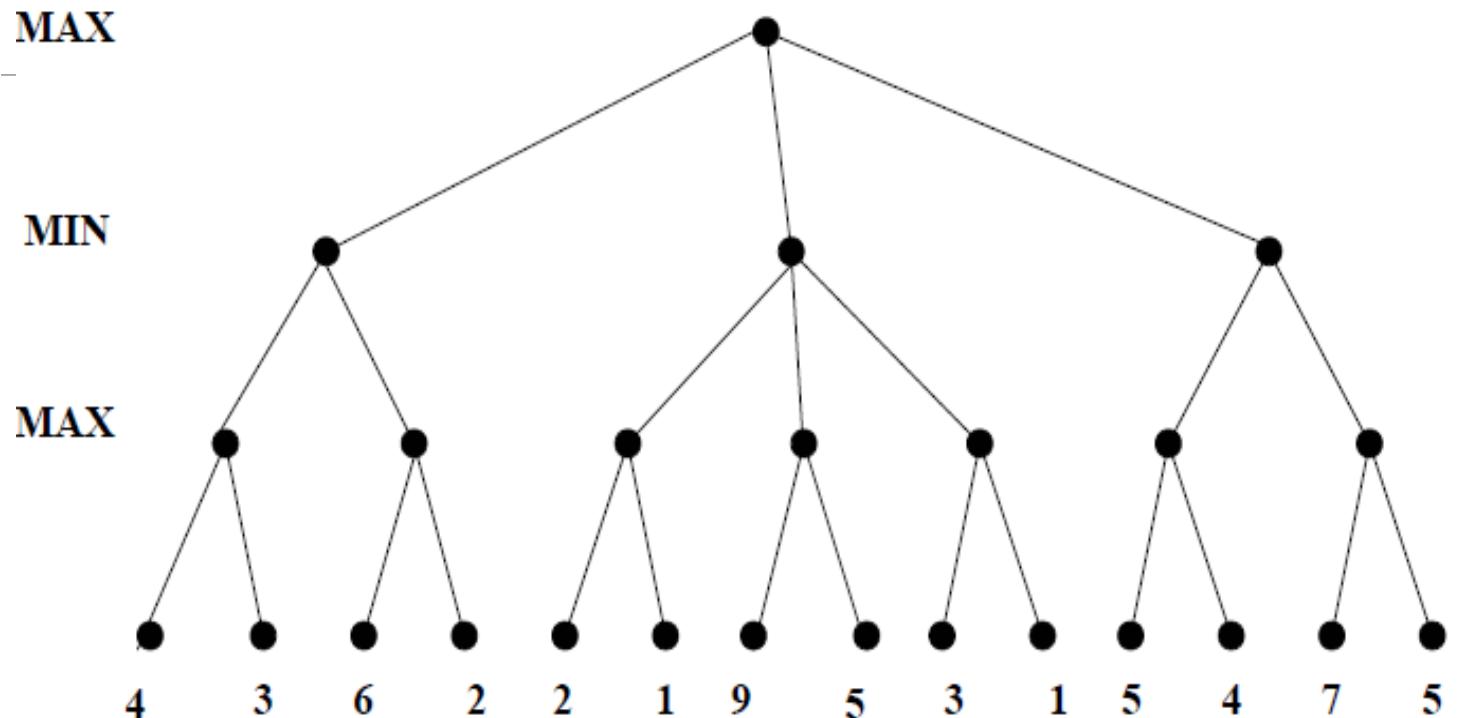
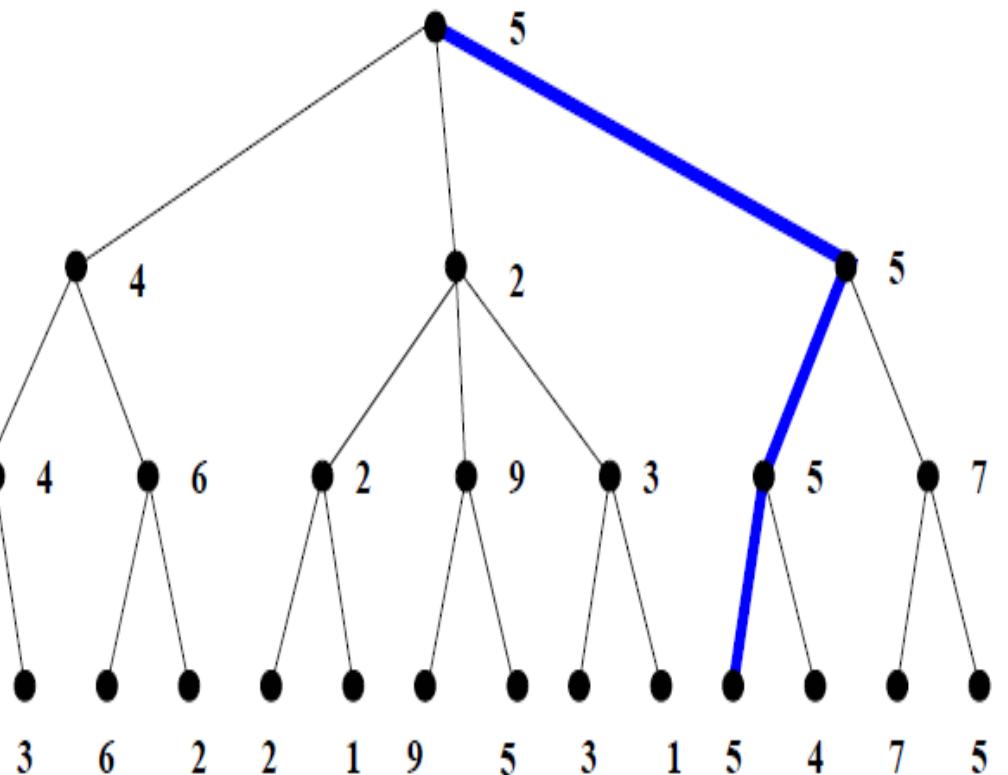
Alpha-beta pruning

If the node is a max node, the value of beta is returned. For each of the children apply the minimax algorithm with the alpha– beta pruning.

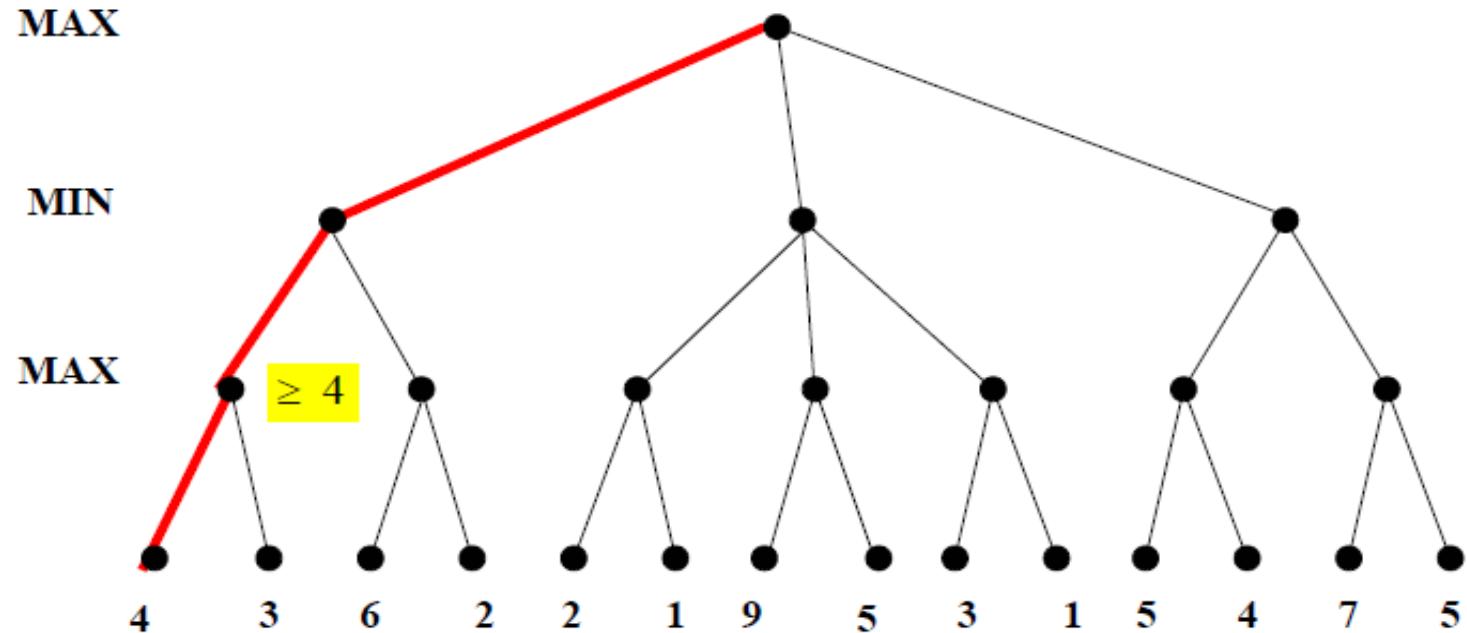
If the value returned by a child is greater, the alpha set alpha to this value.

If at any stage alpha is greater than or equal to beta, more children are not examined, and the value of alpha is returned.

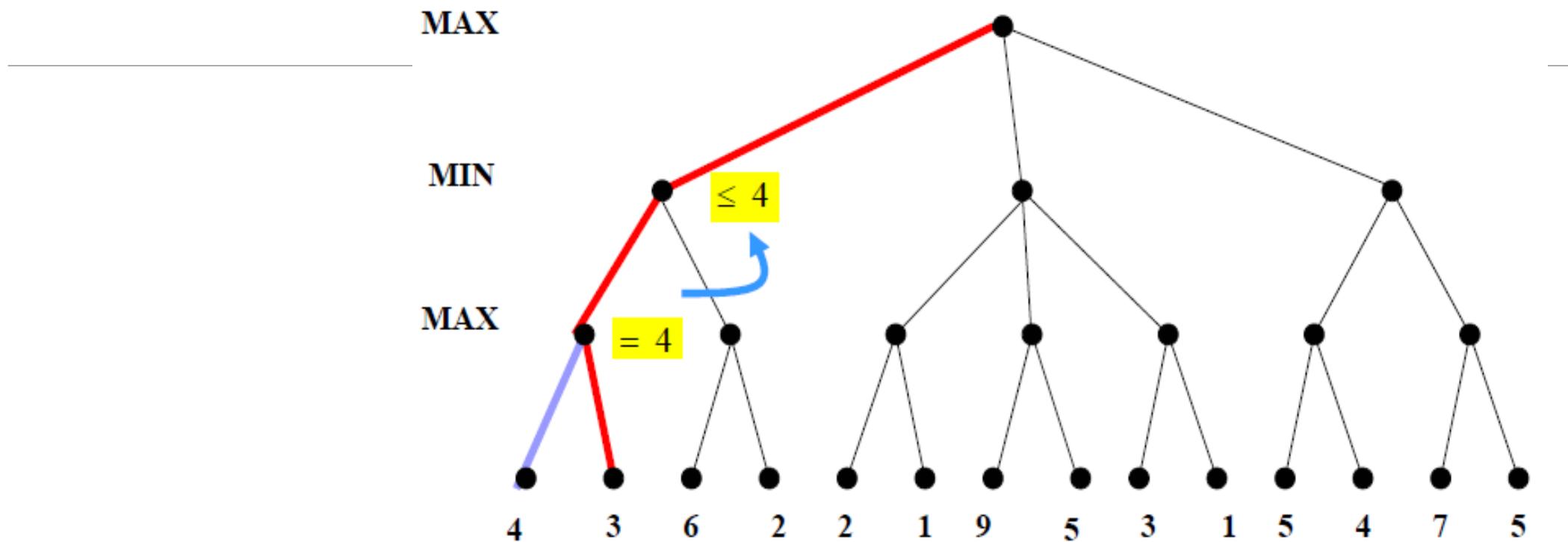
Alpha-beta pruning Example 2



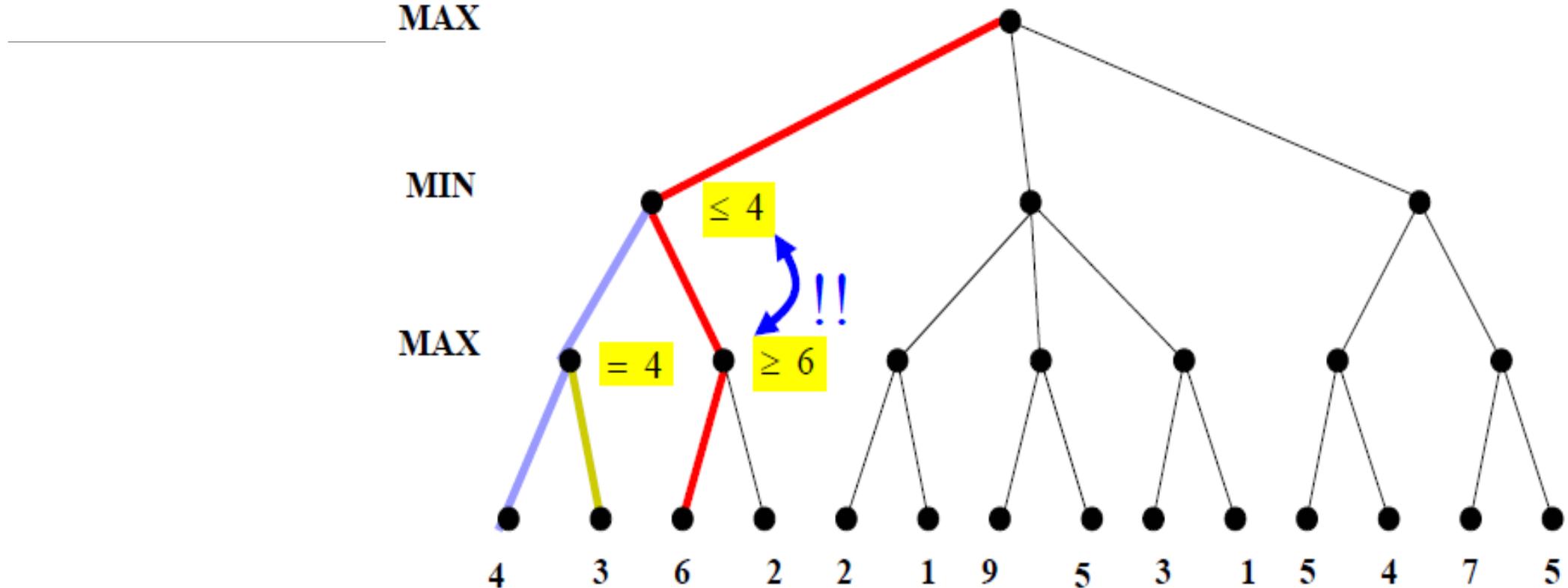
Alpha-beta pruning



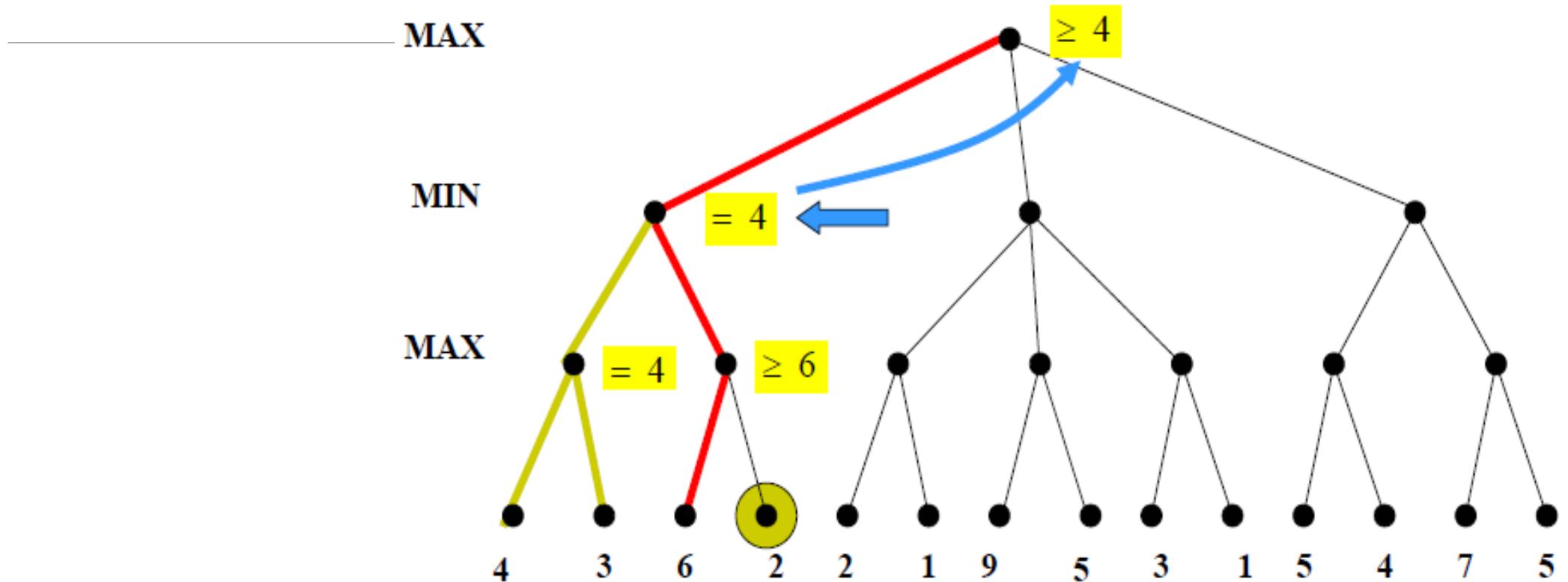
Alpha-beta pruning



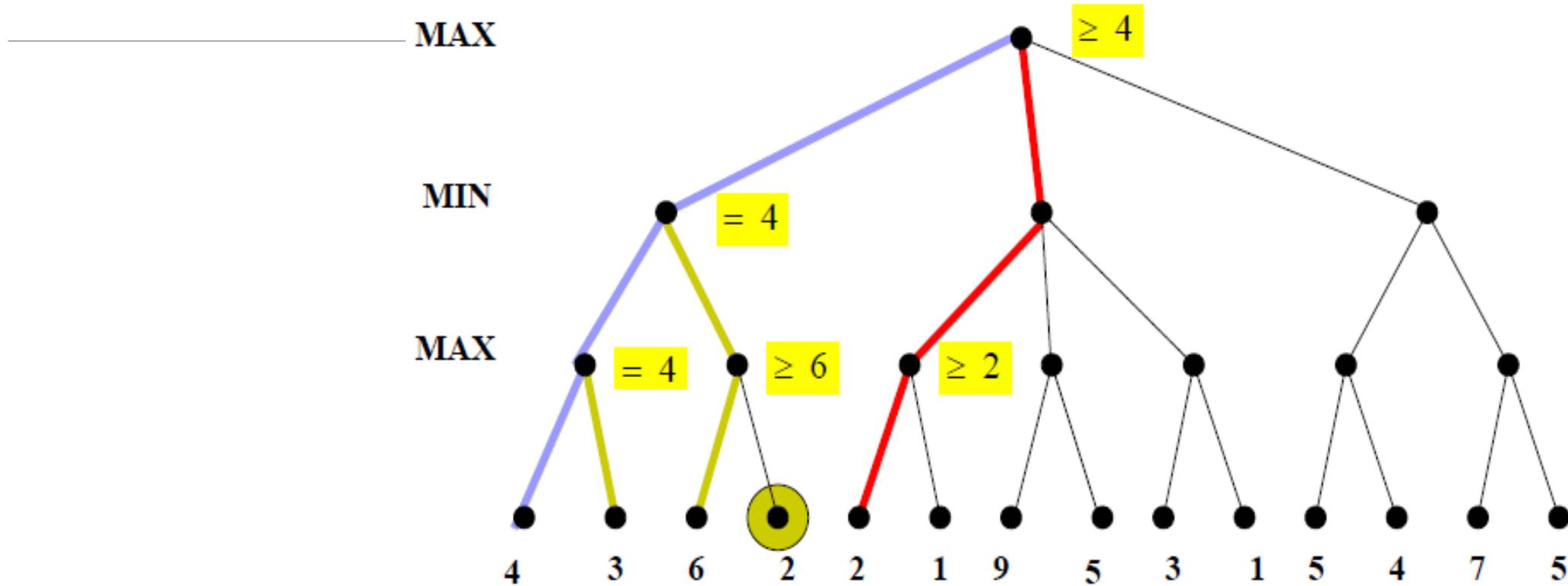
Alpha-beta pruning



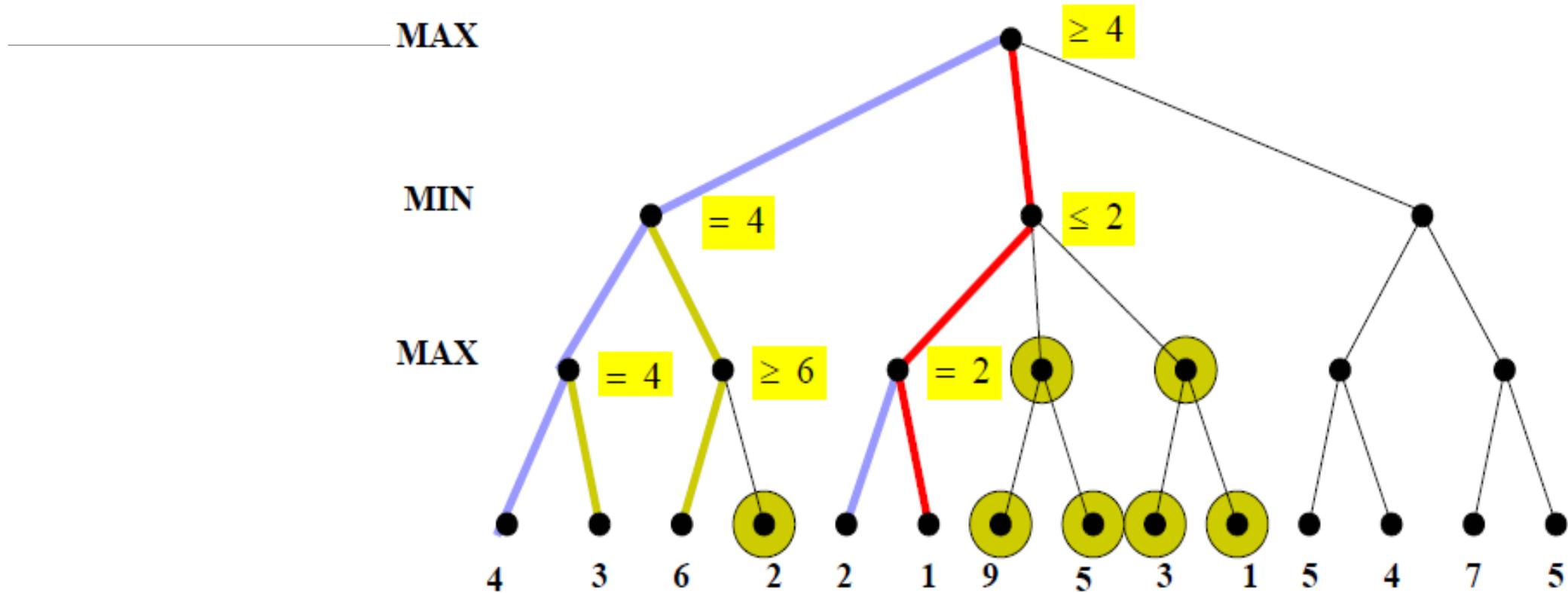
Alpha-beta pruning



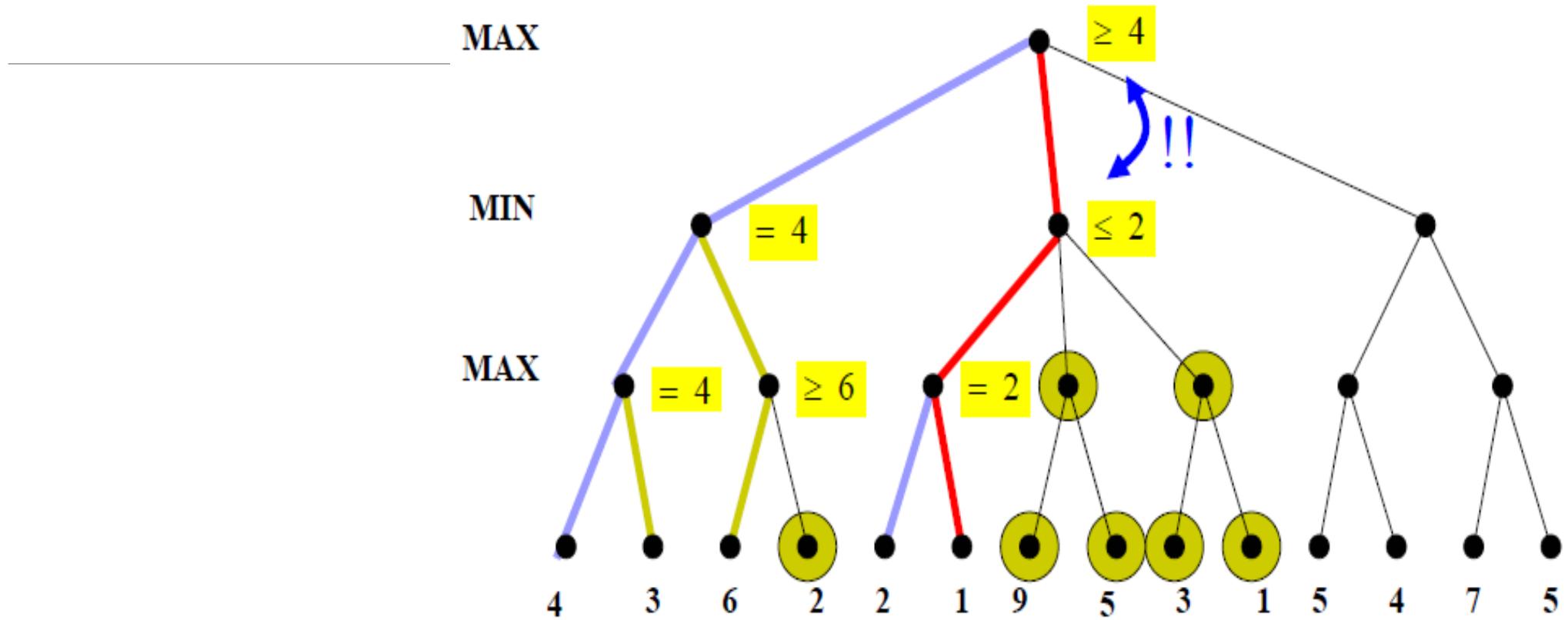
Alpha-beta pruning



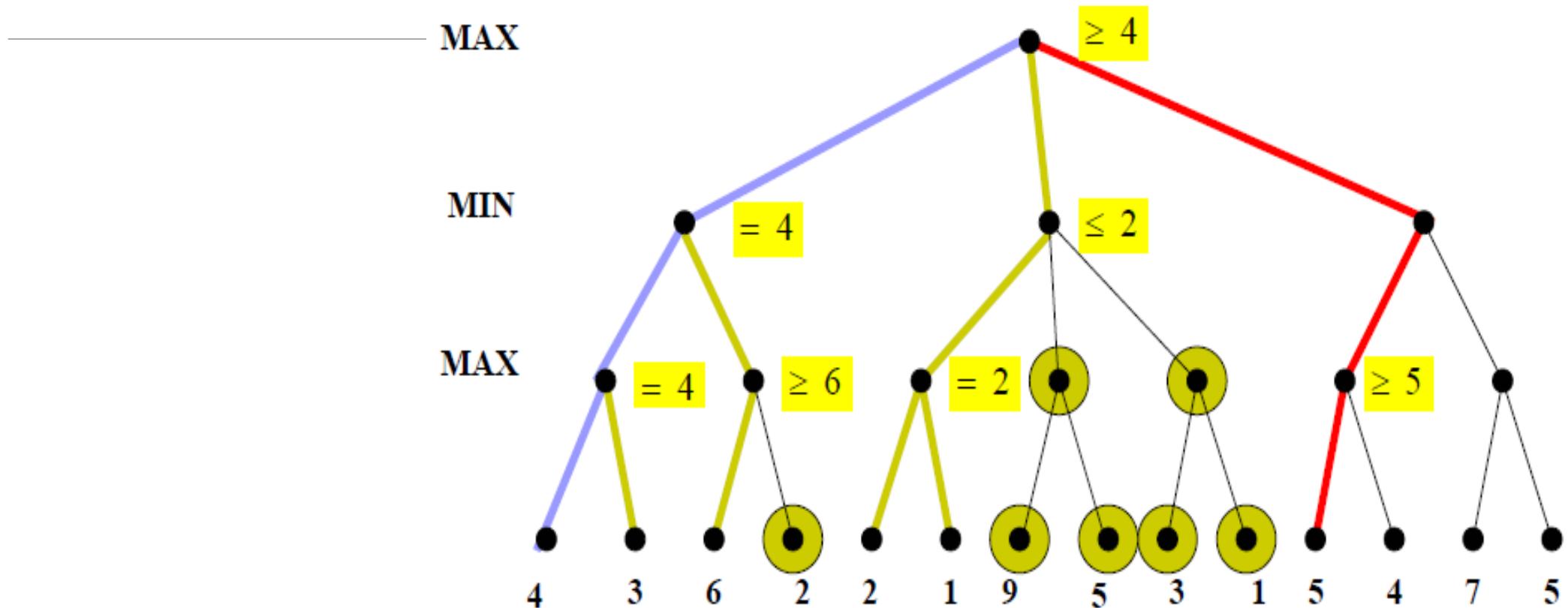
Alpha-beta pruning



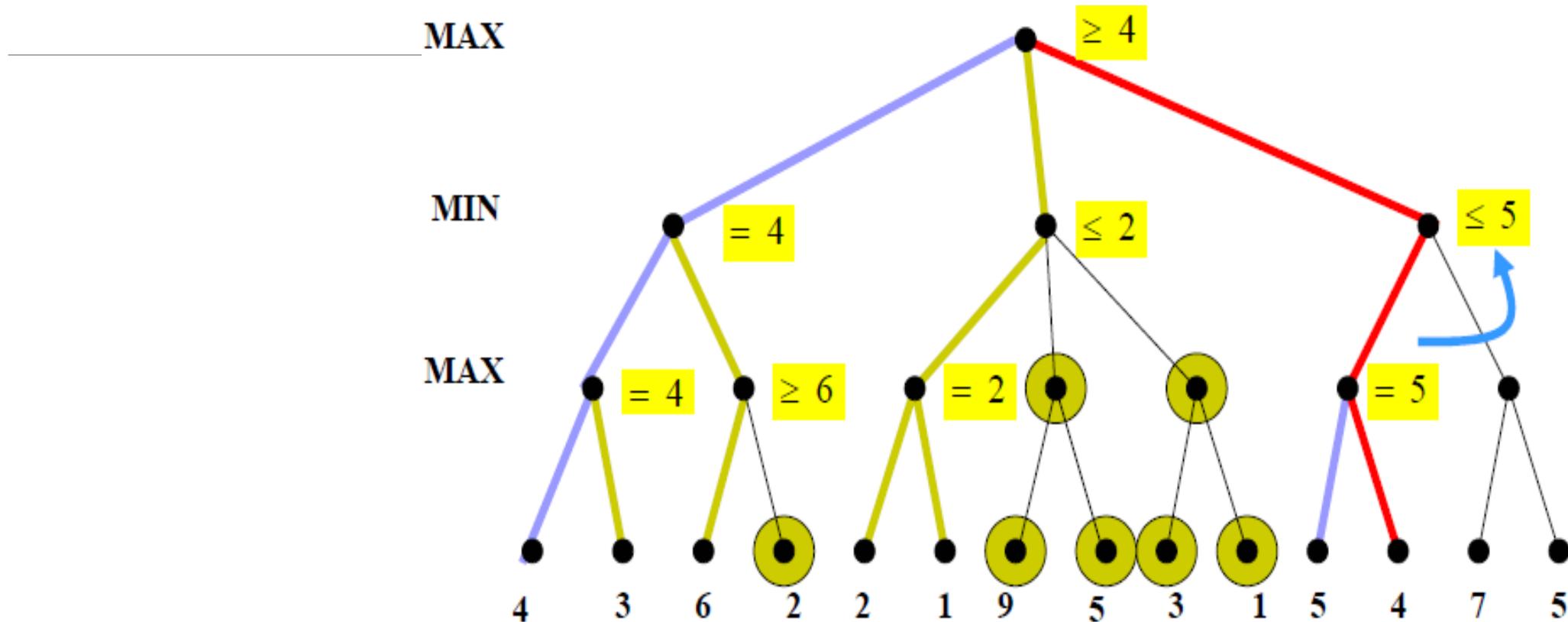
Alpha-beta pruning



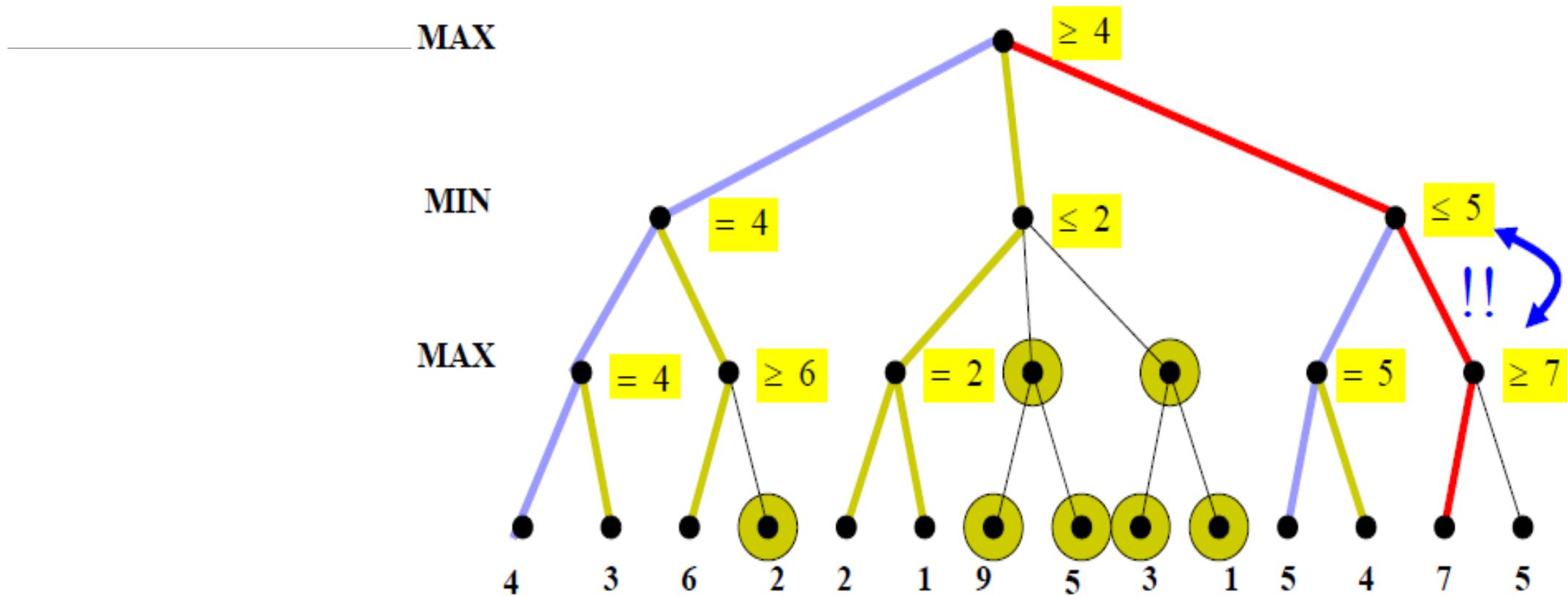
Alpha-beta pruning



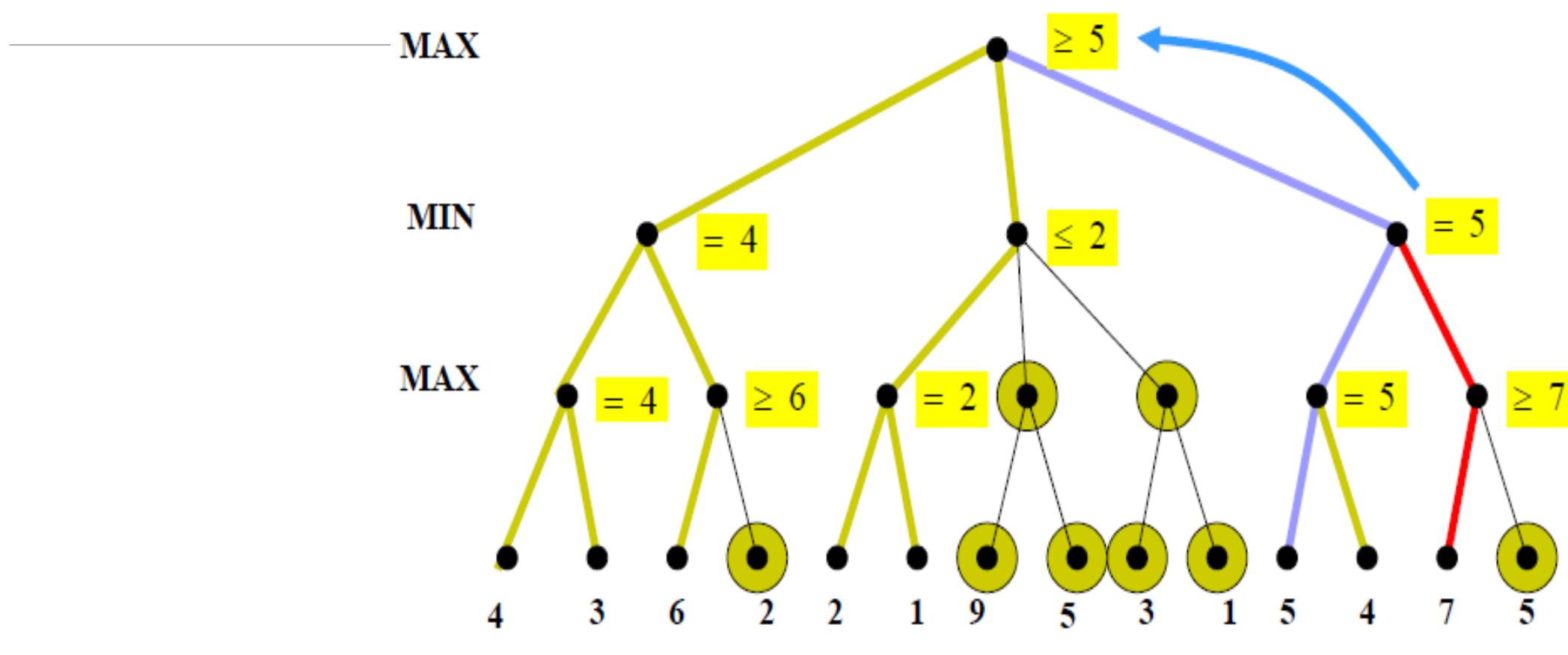
Alpha-beta pruning



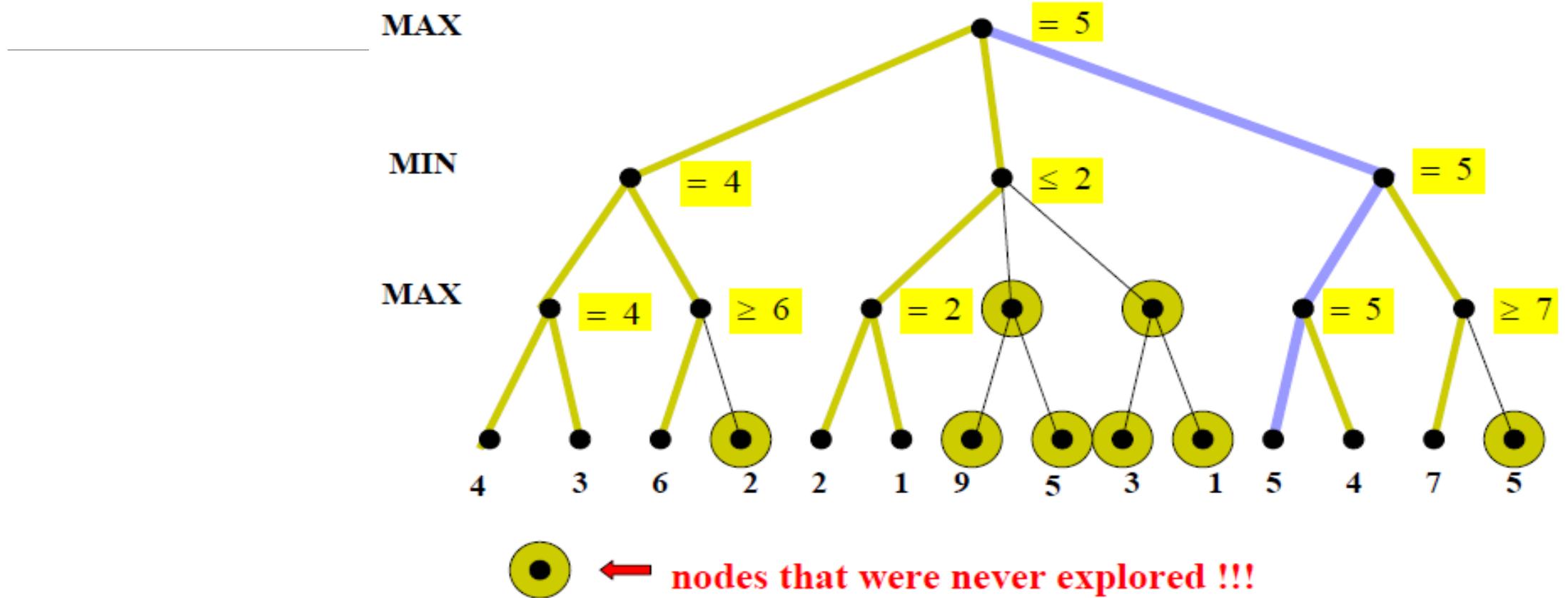
Alpha-beta pruning



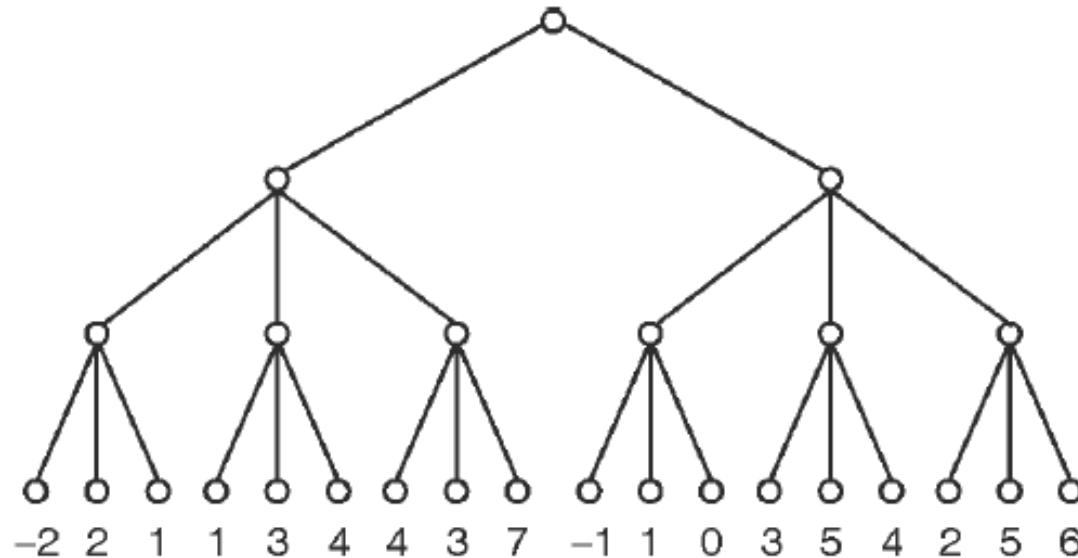
Alpha-beta pruning



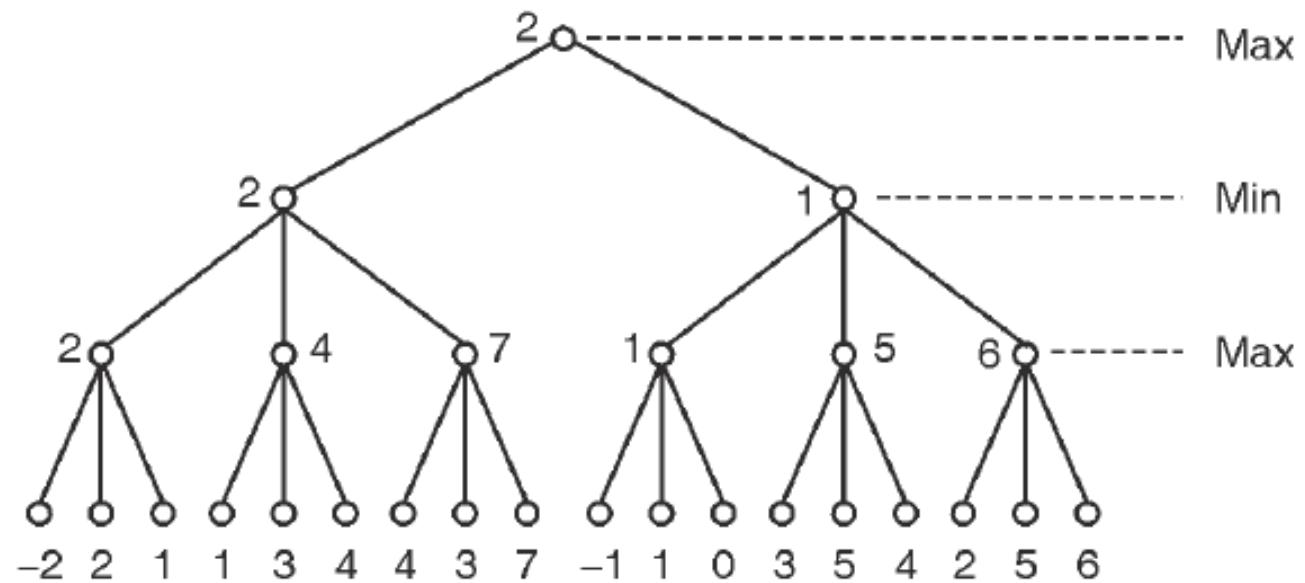
Alpha-beta pruning



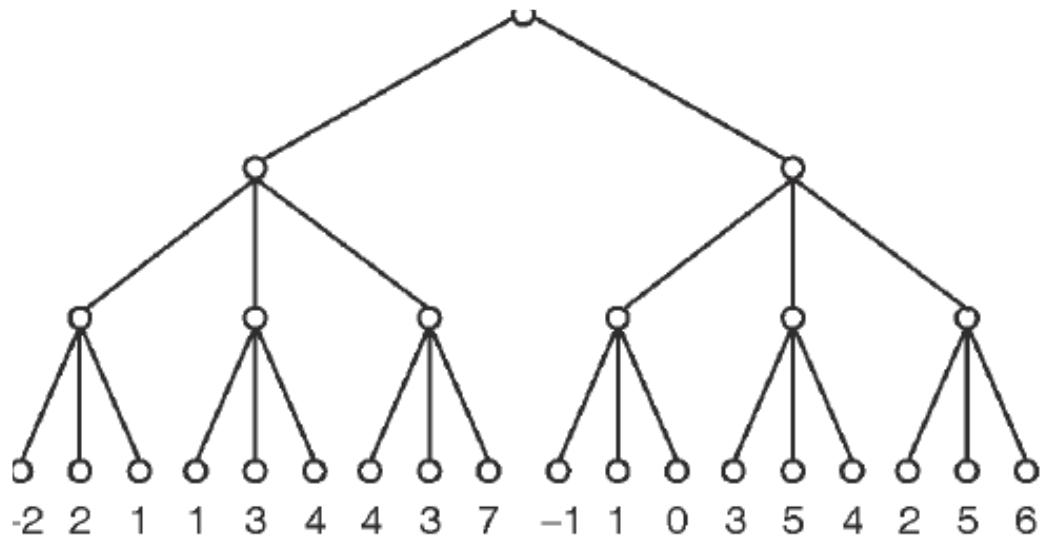
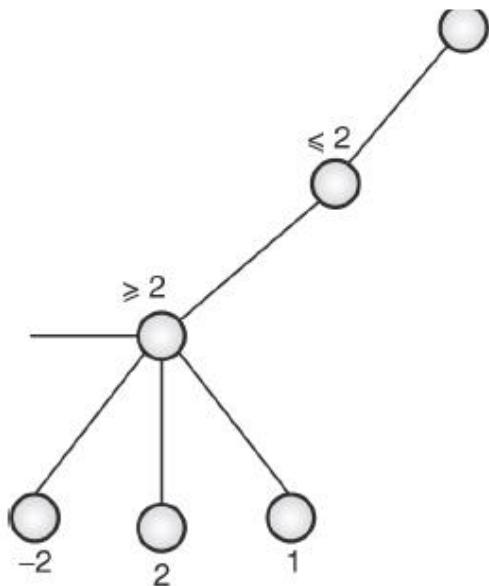
Explain Min-Max and Alpha Beta pruning algorithm with following example.

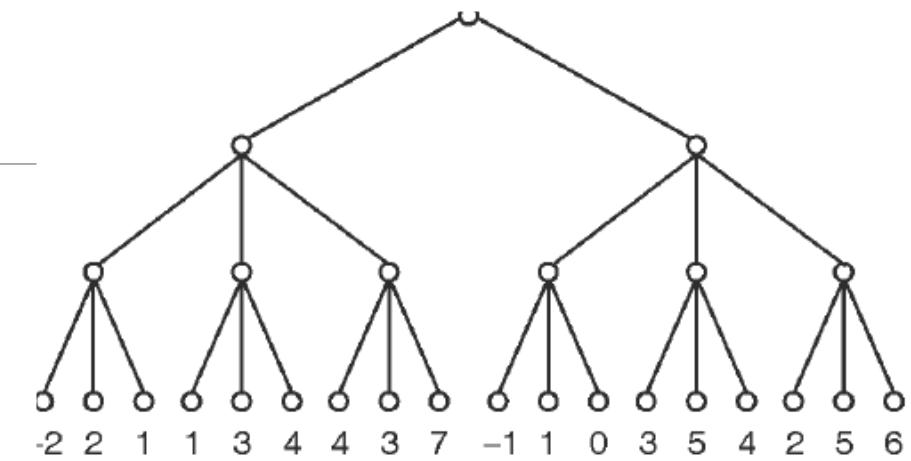
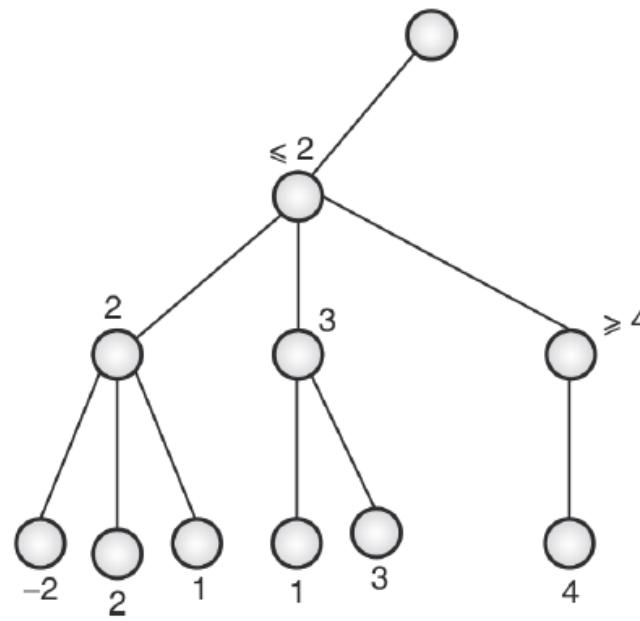
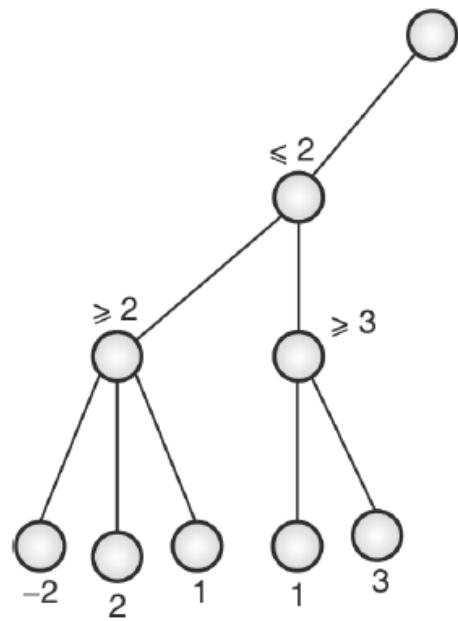


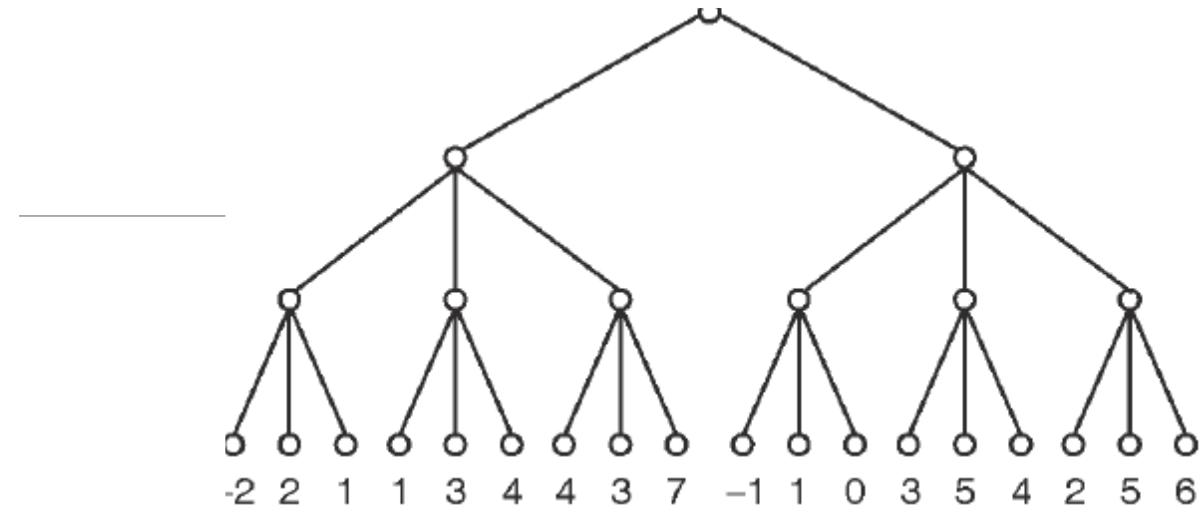
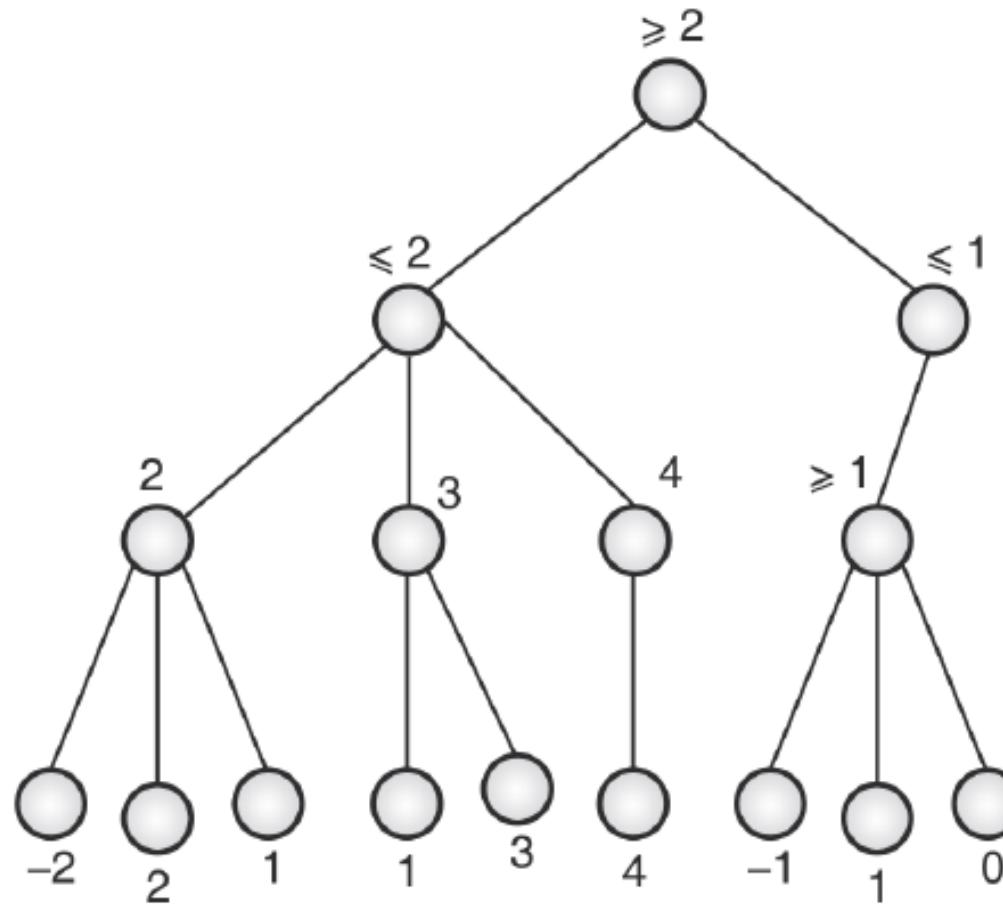
Using min-max algorithm

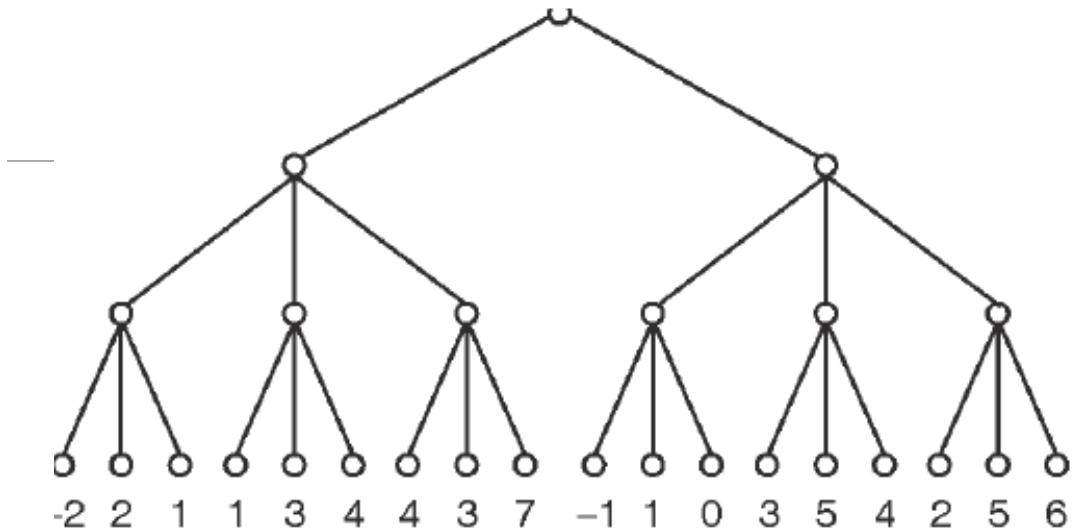
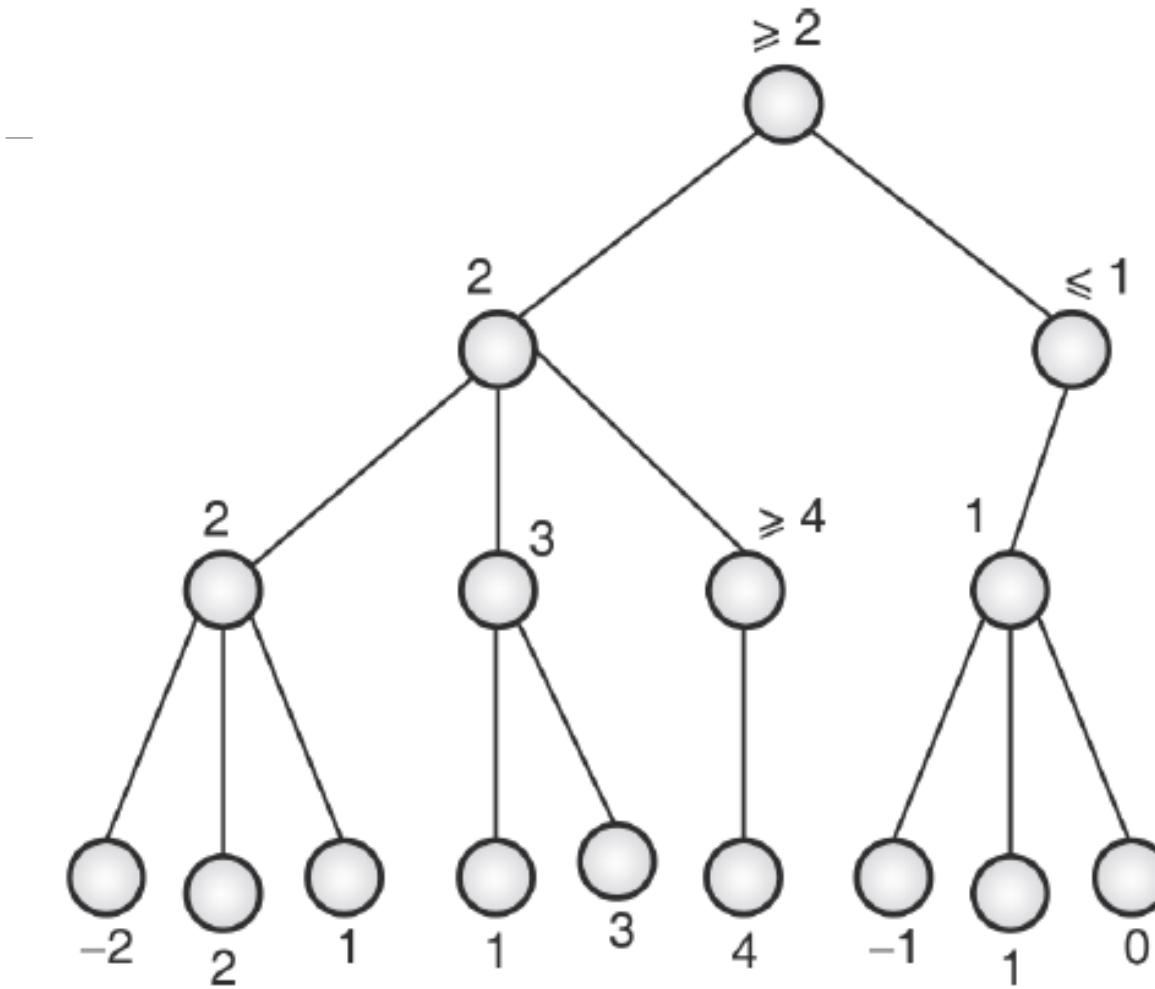


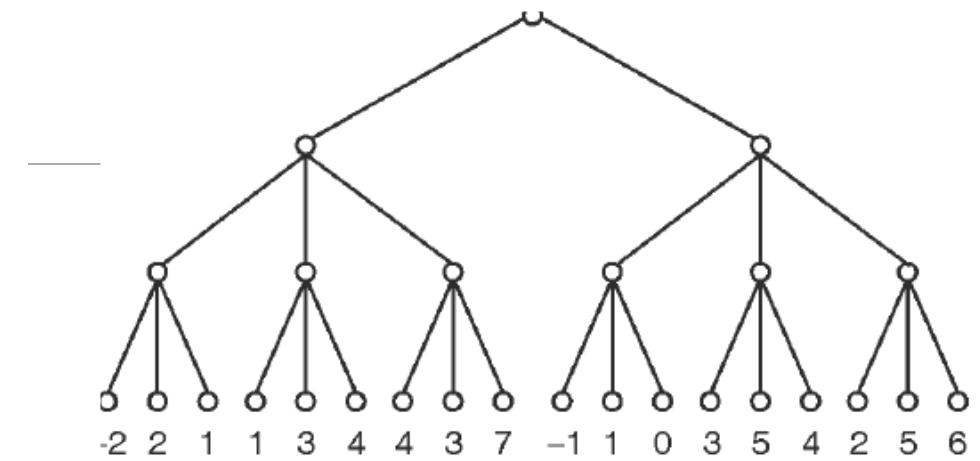
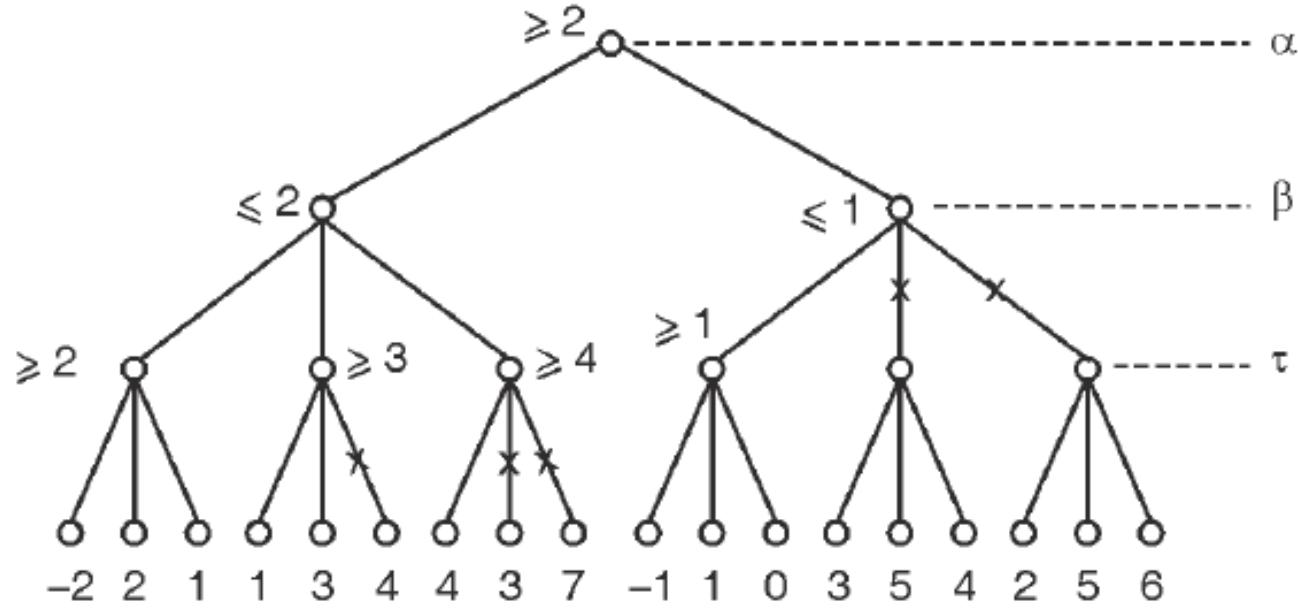
Alpha Beta Pruning



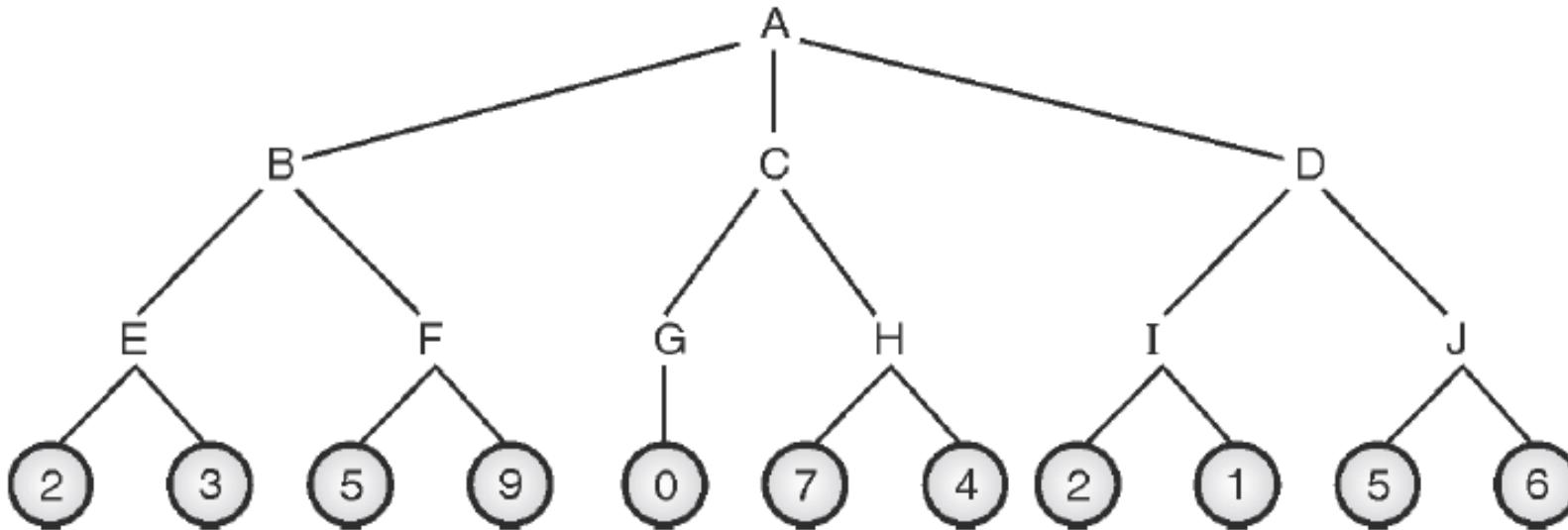






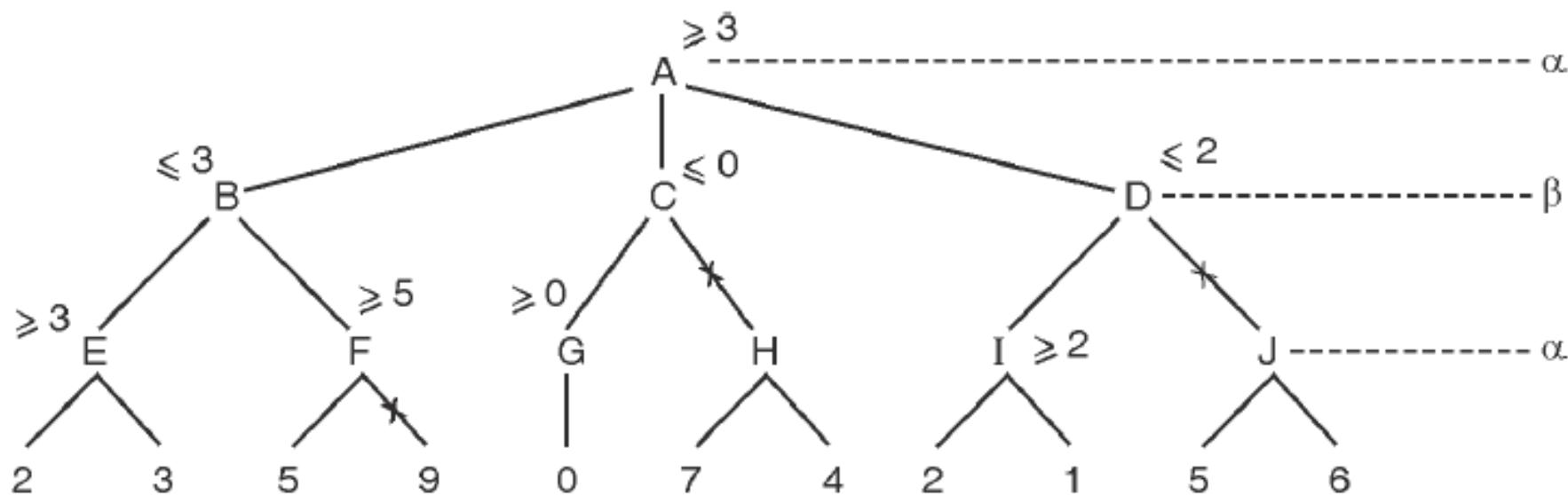


Perform $\alpha - \beta$ cutoff on the following.



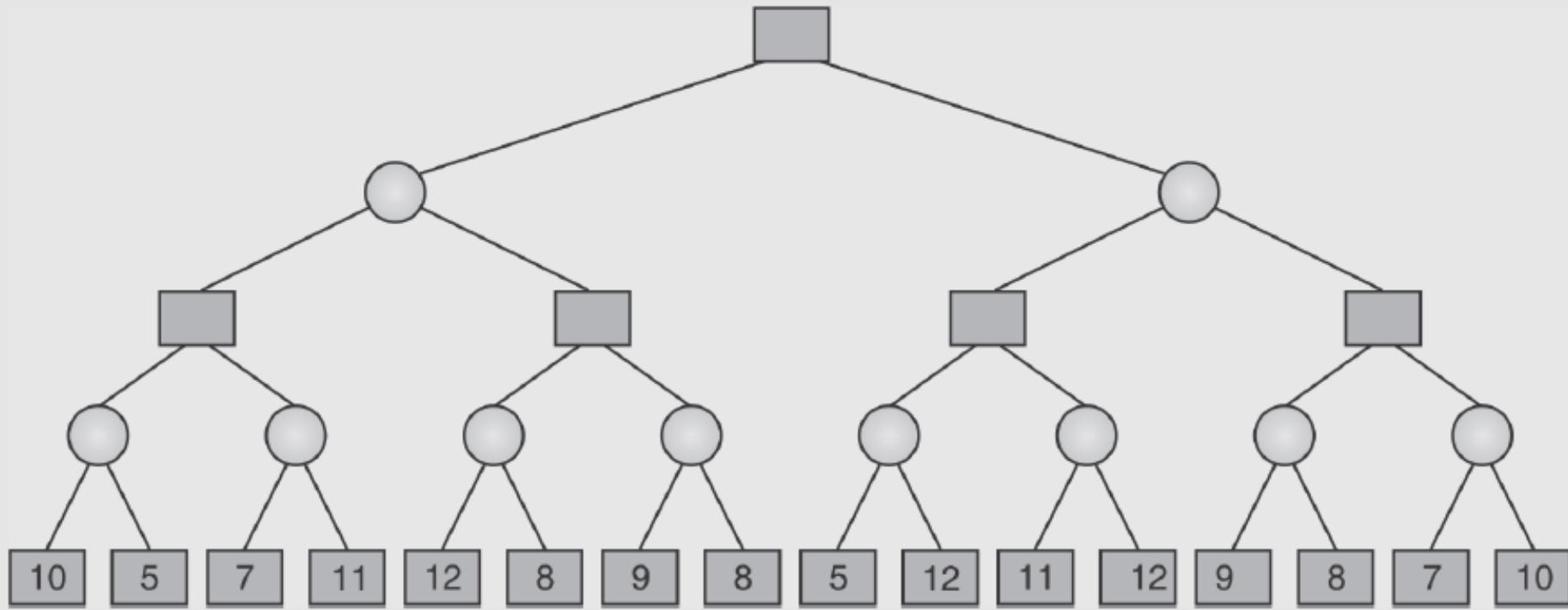
No. of α - cuts = 1

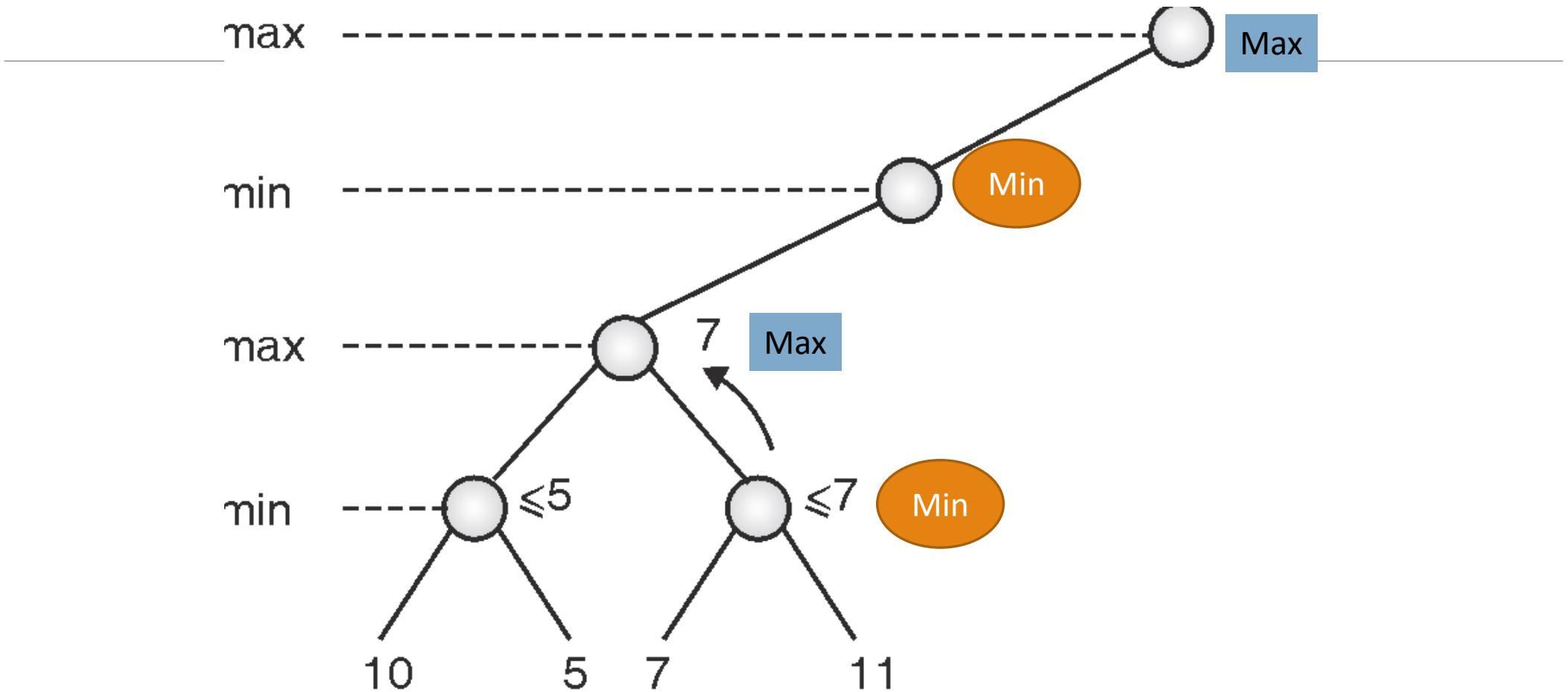
No. of β - cuts = 2

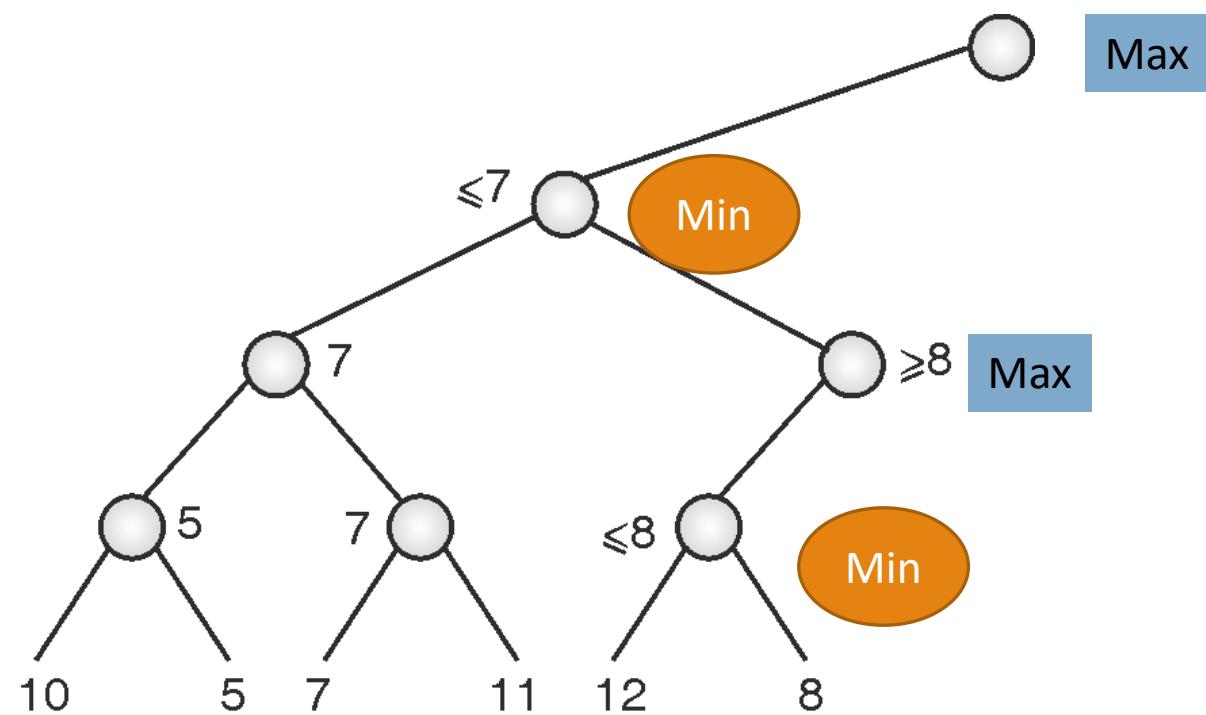


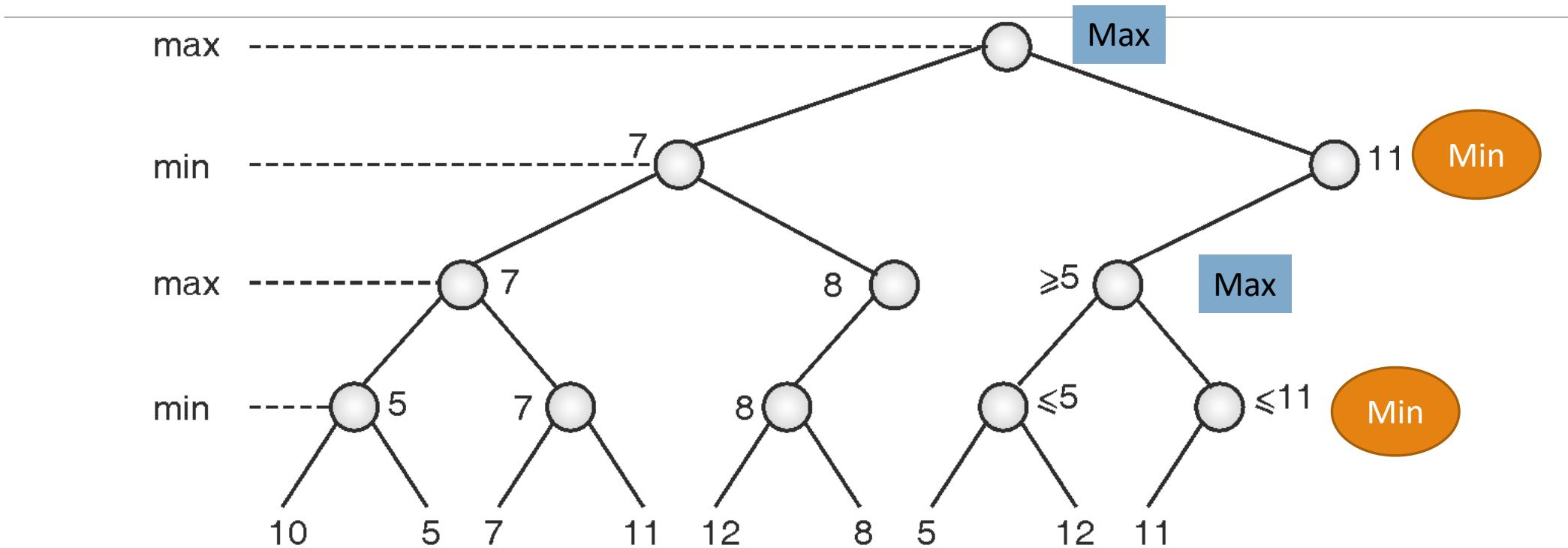
Apply alpha-beta pruning on example given in Fig. P. 2.22.3 considering first node as max.

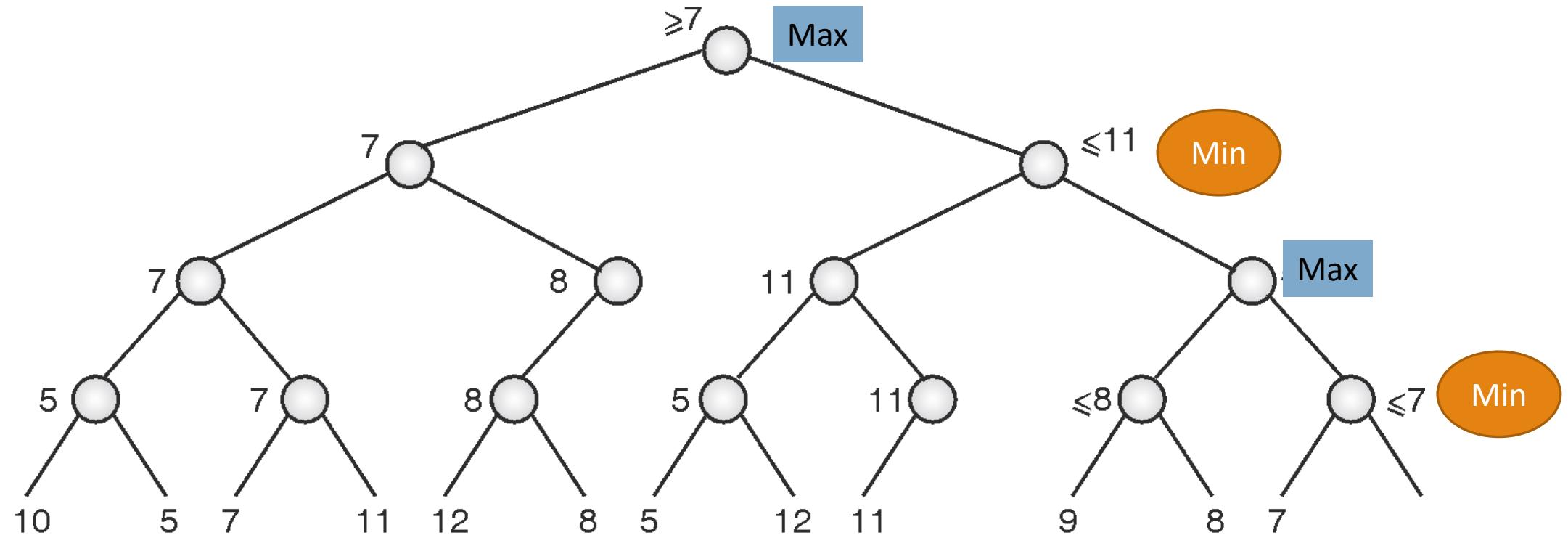
MU - May 16, 10 Marks

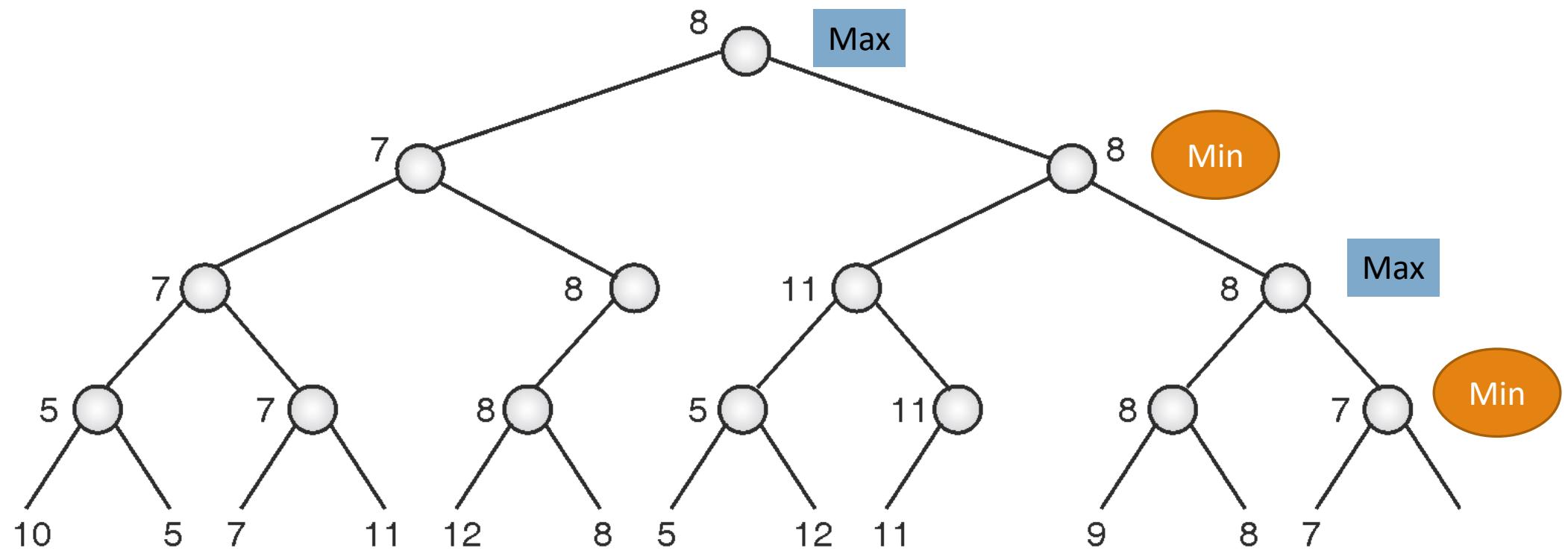










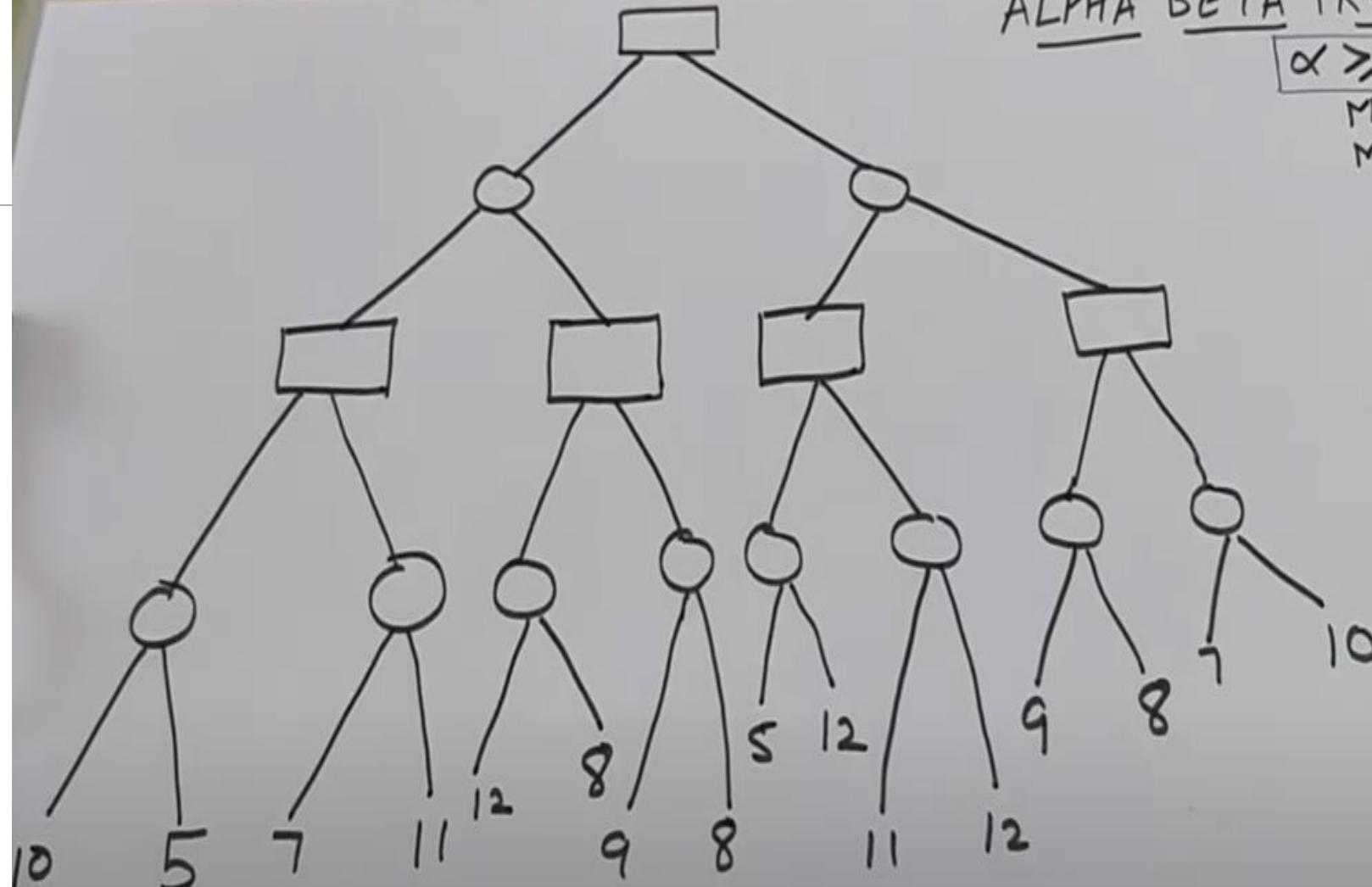


ALPHA BETA PRUNING

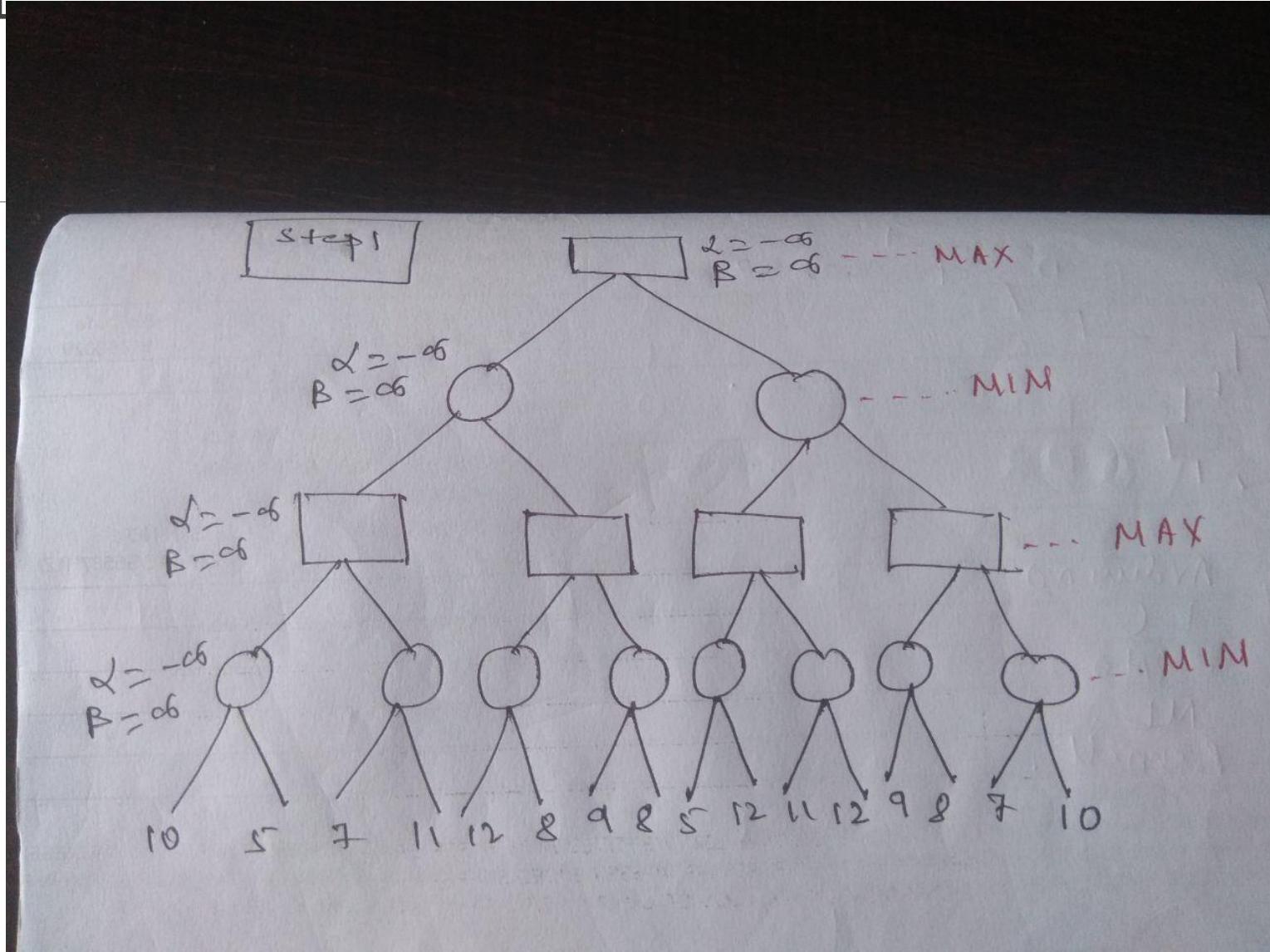
$$\alpha \geq \beta$$

MAX $\rightarrow \alpha$

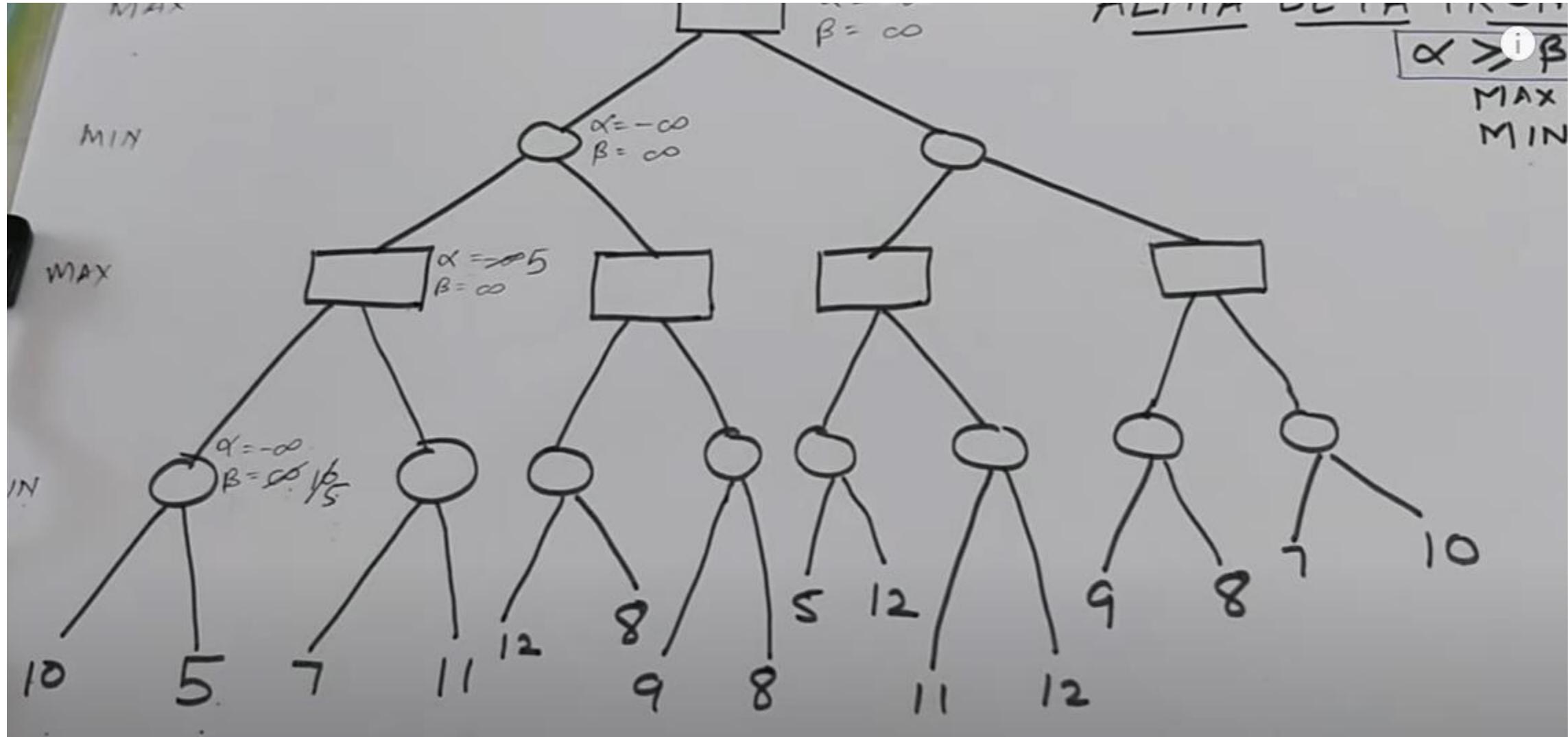
MIN $\rightarrow \beta$



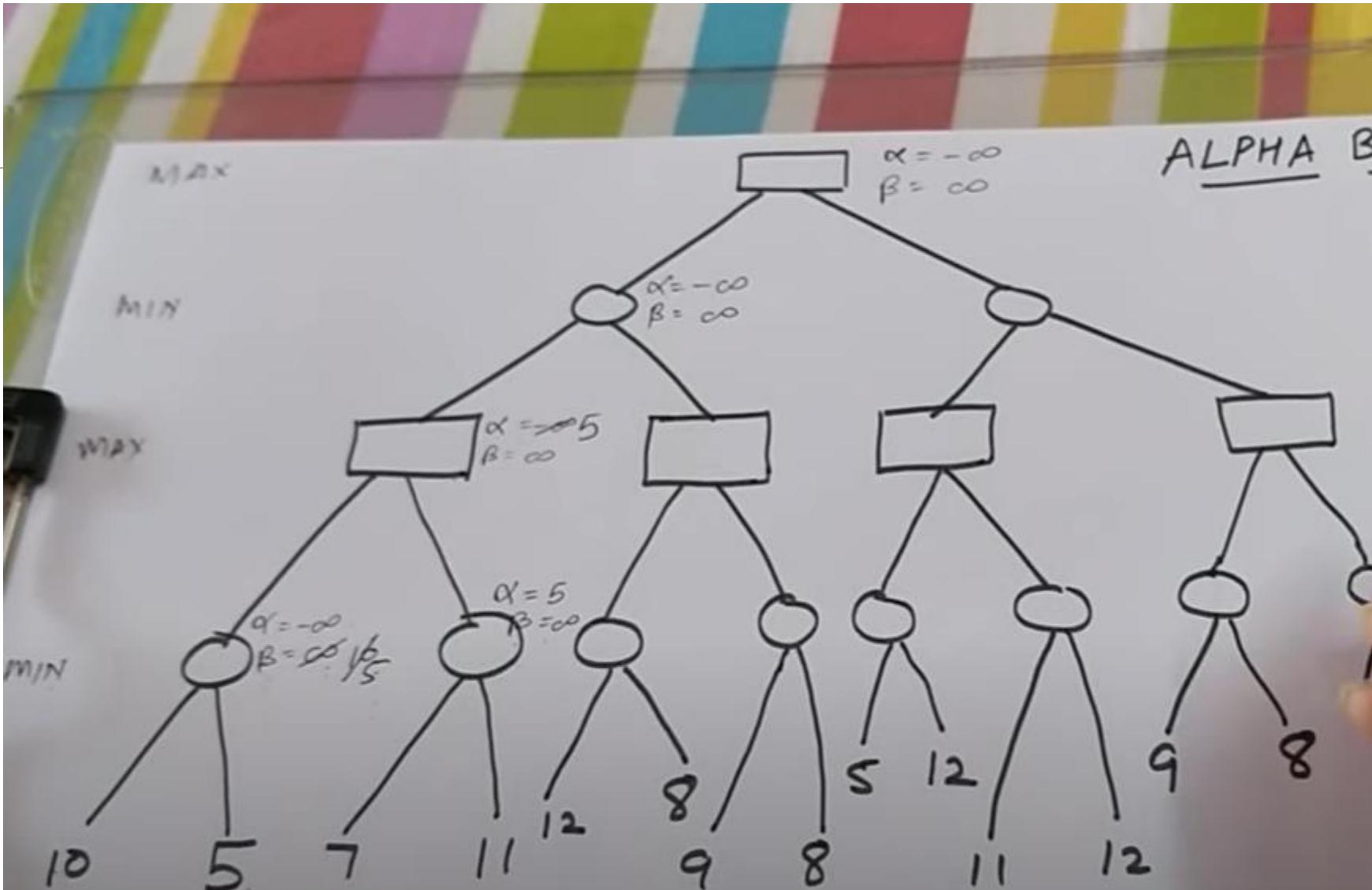
Step 1



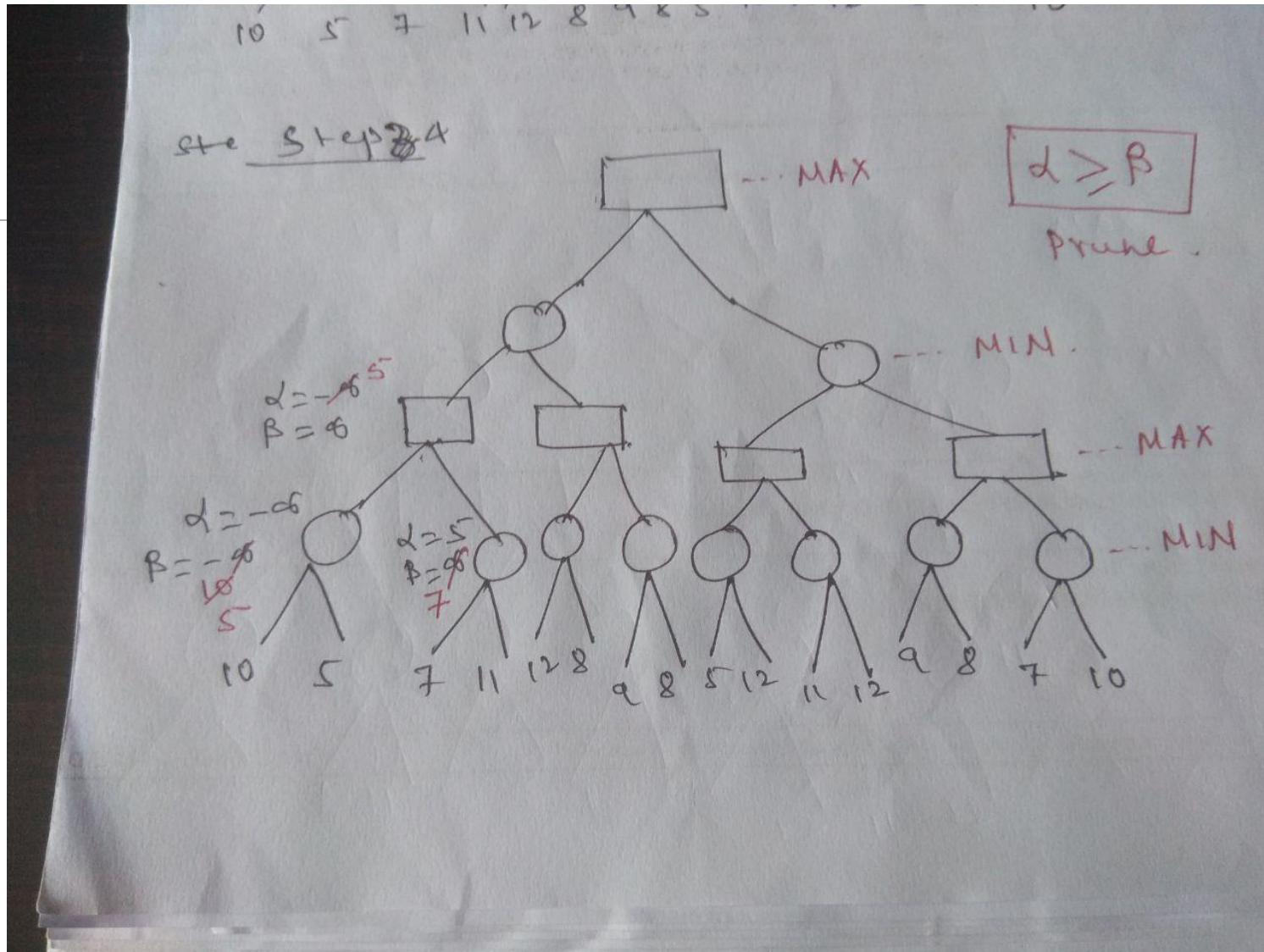
Step2



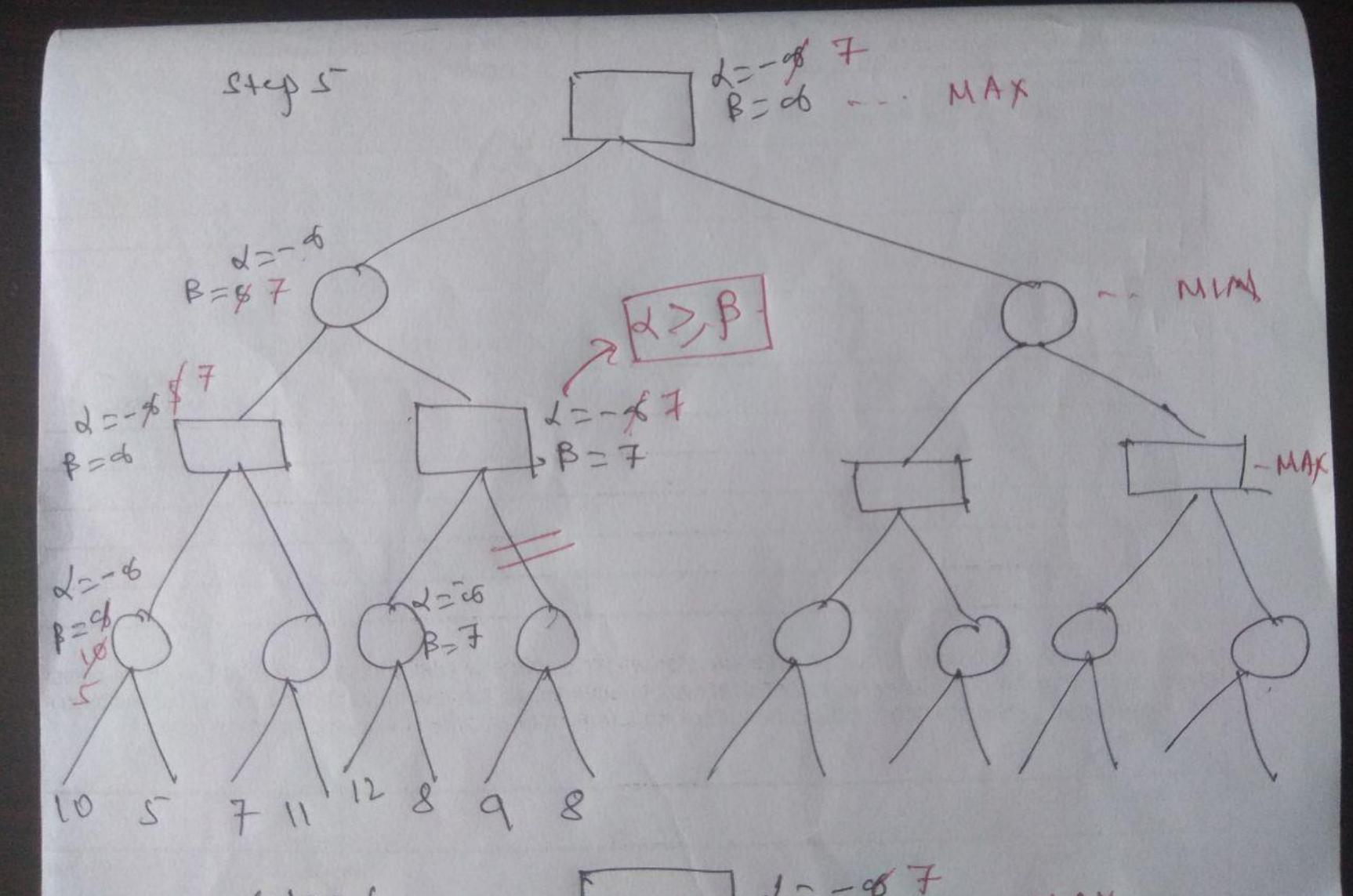
Step3



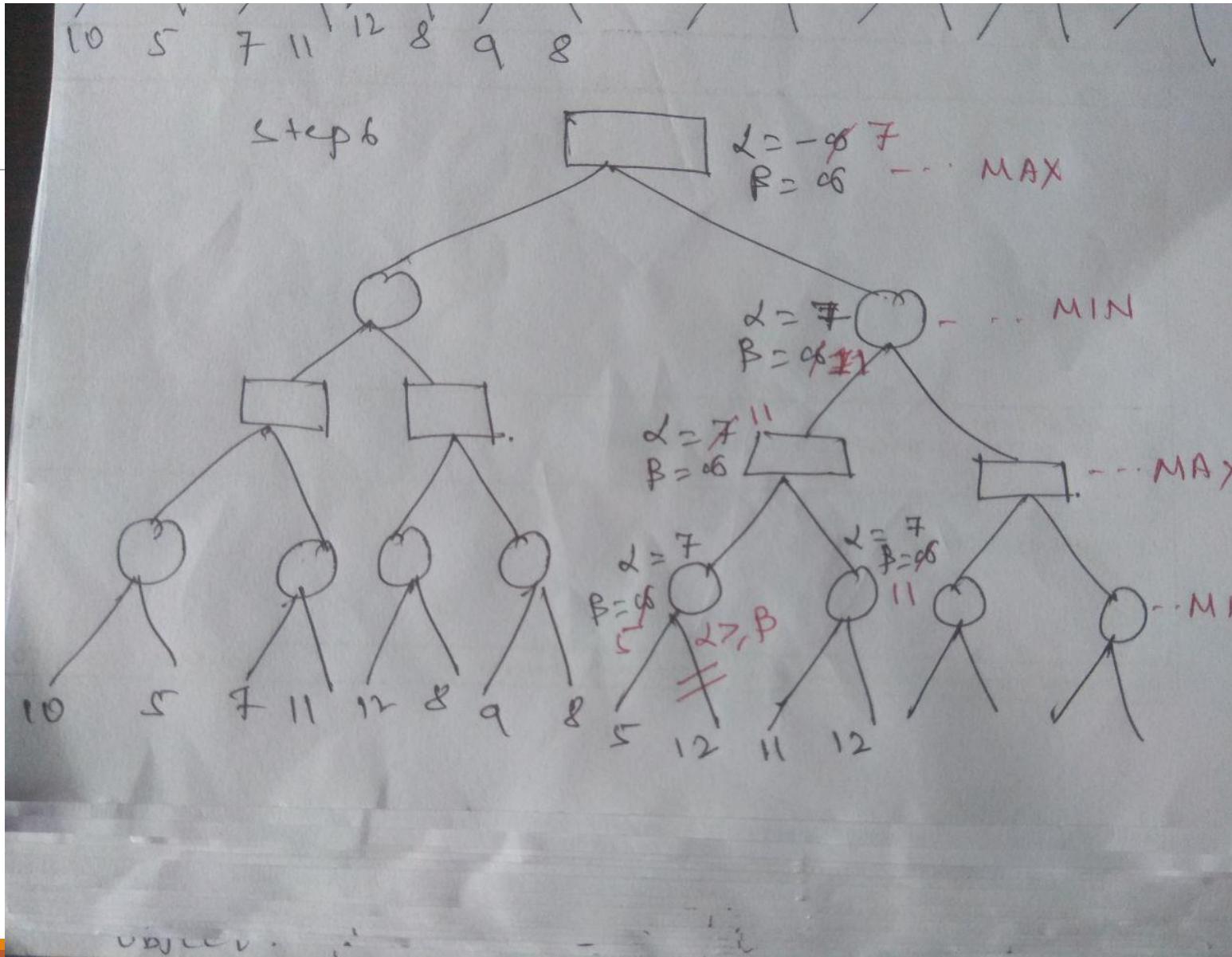
Step4



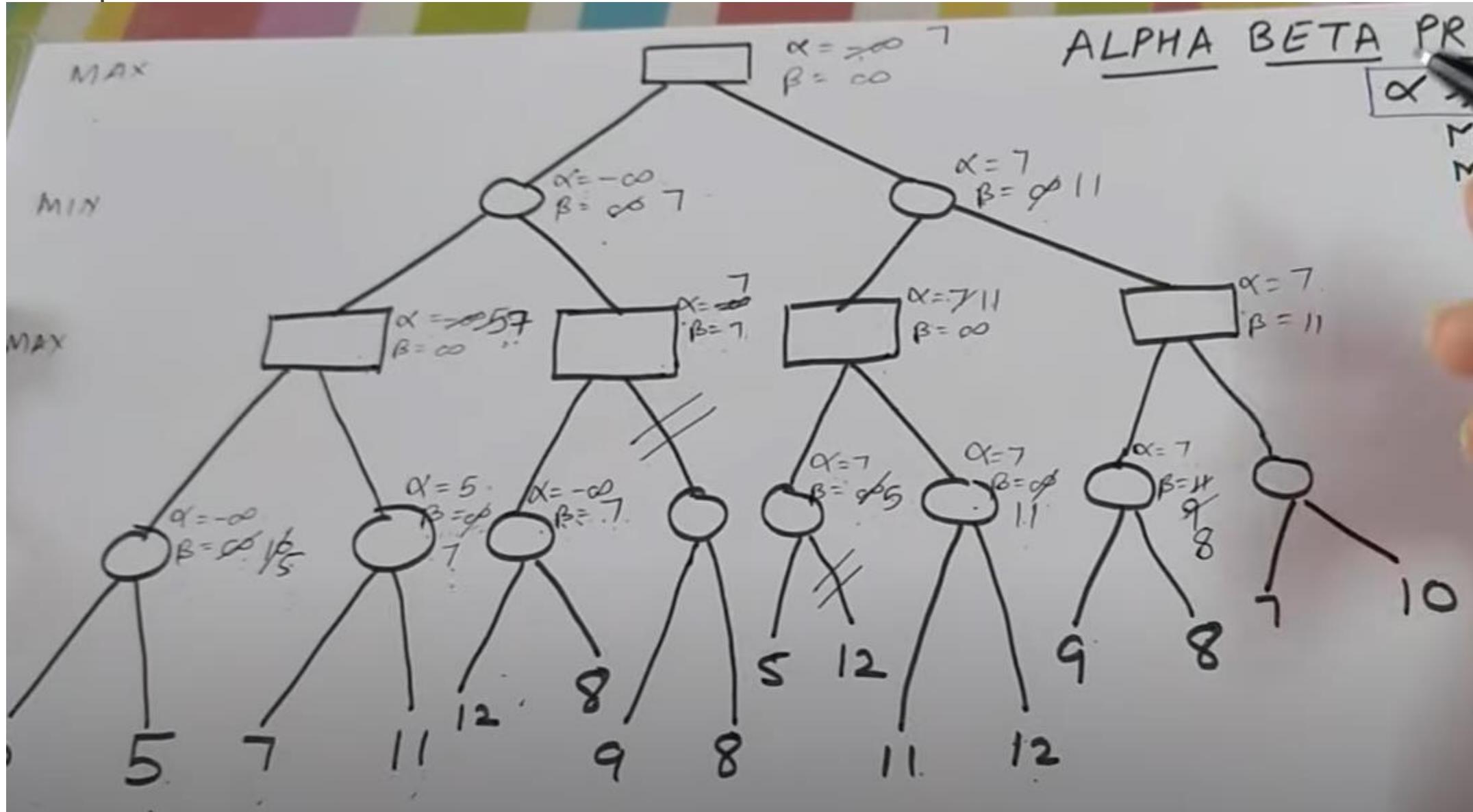
Ste



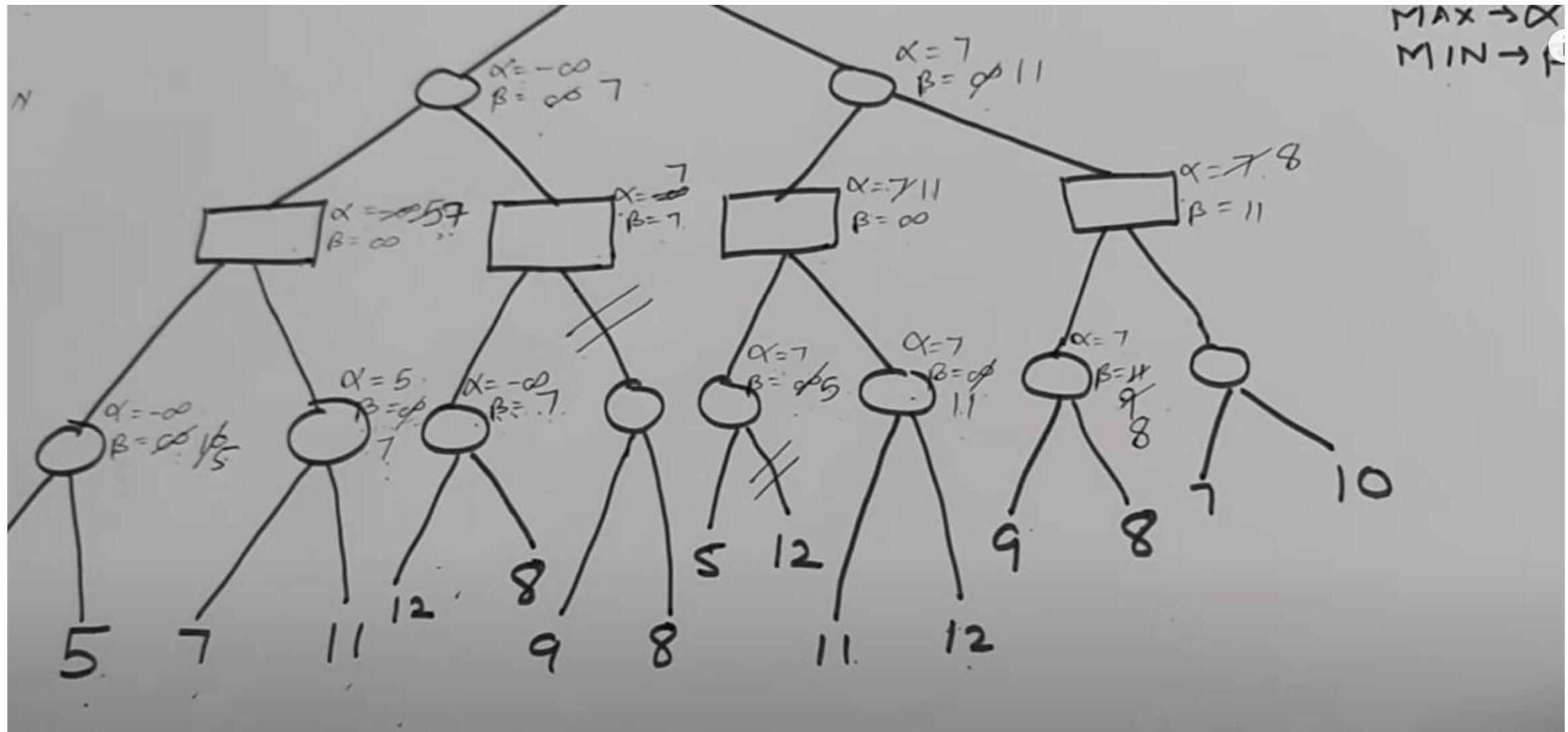
Step 6



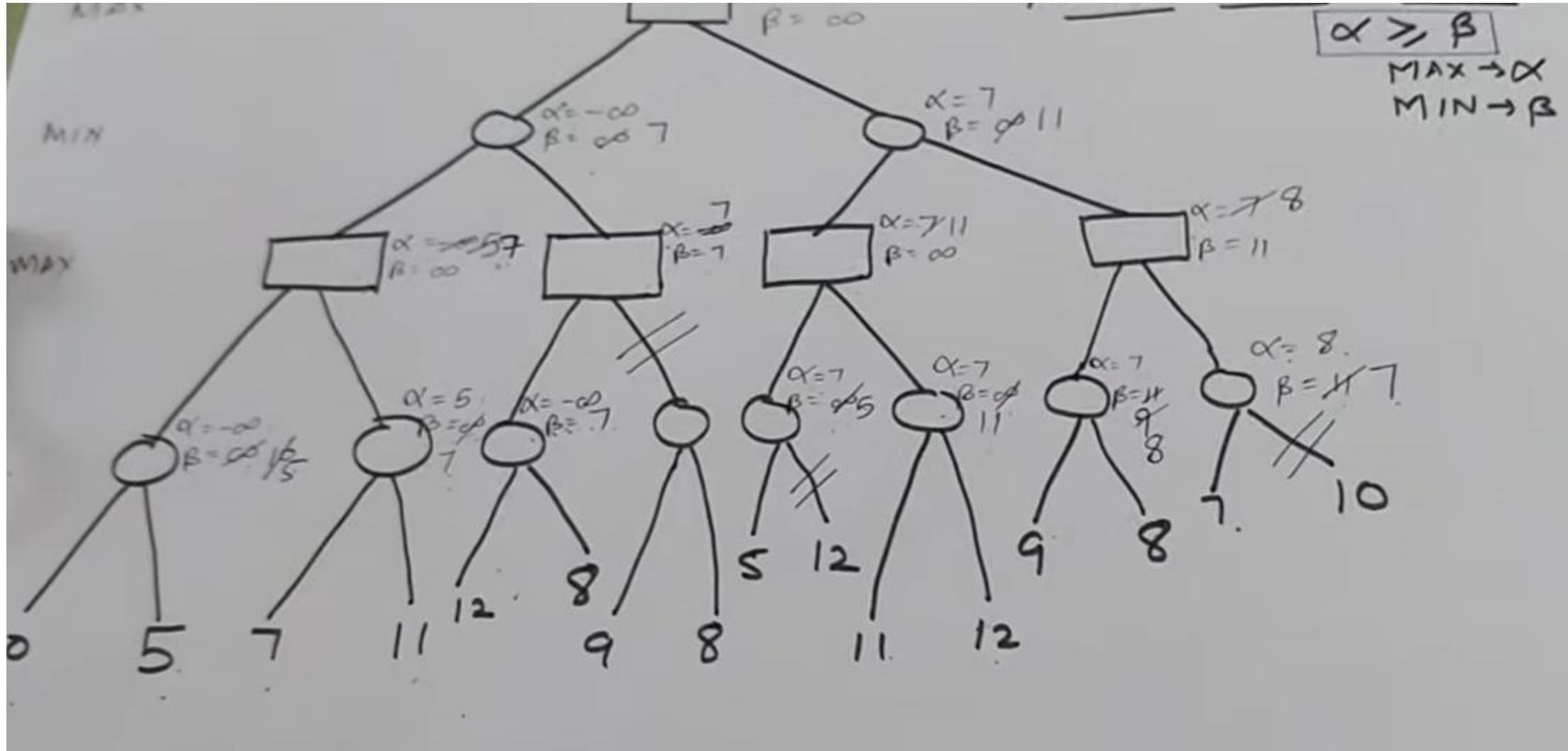
Step 7

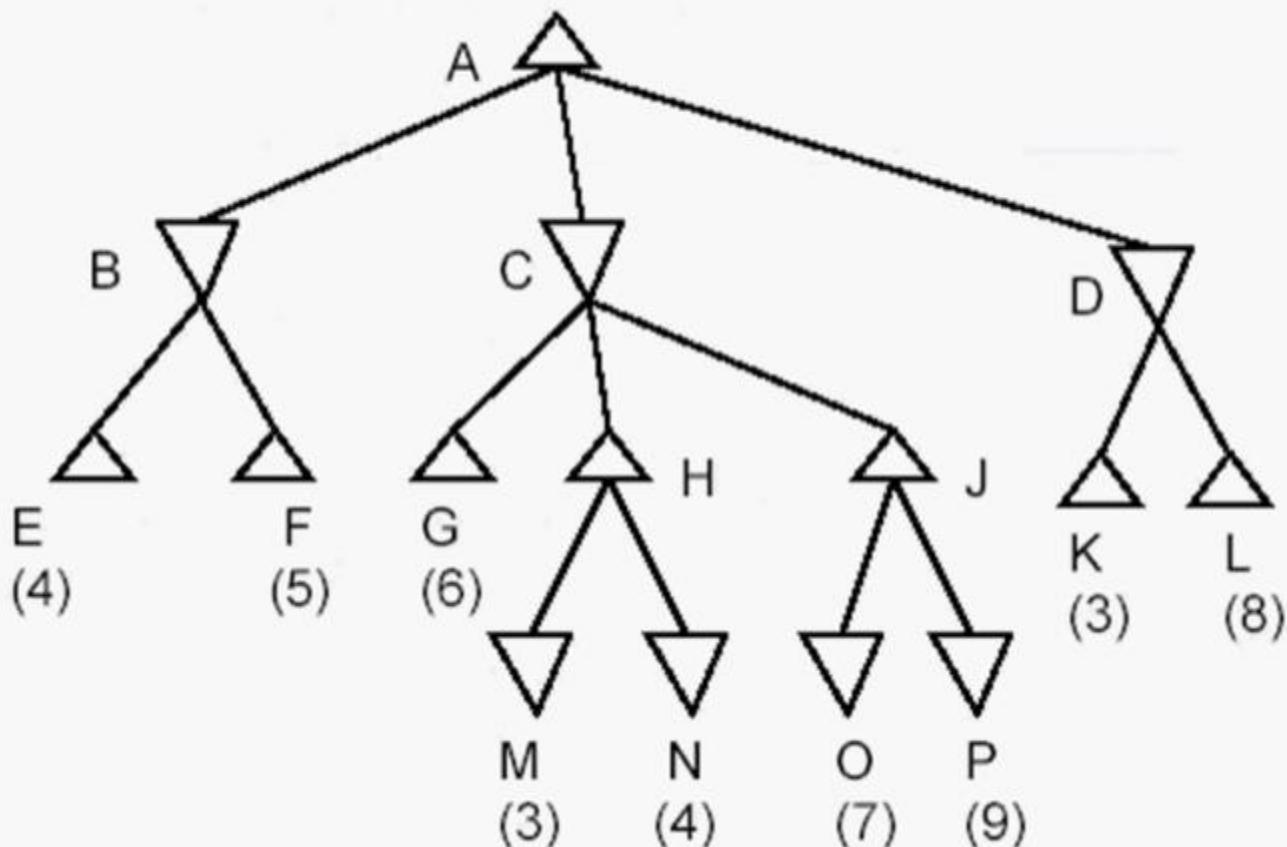


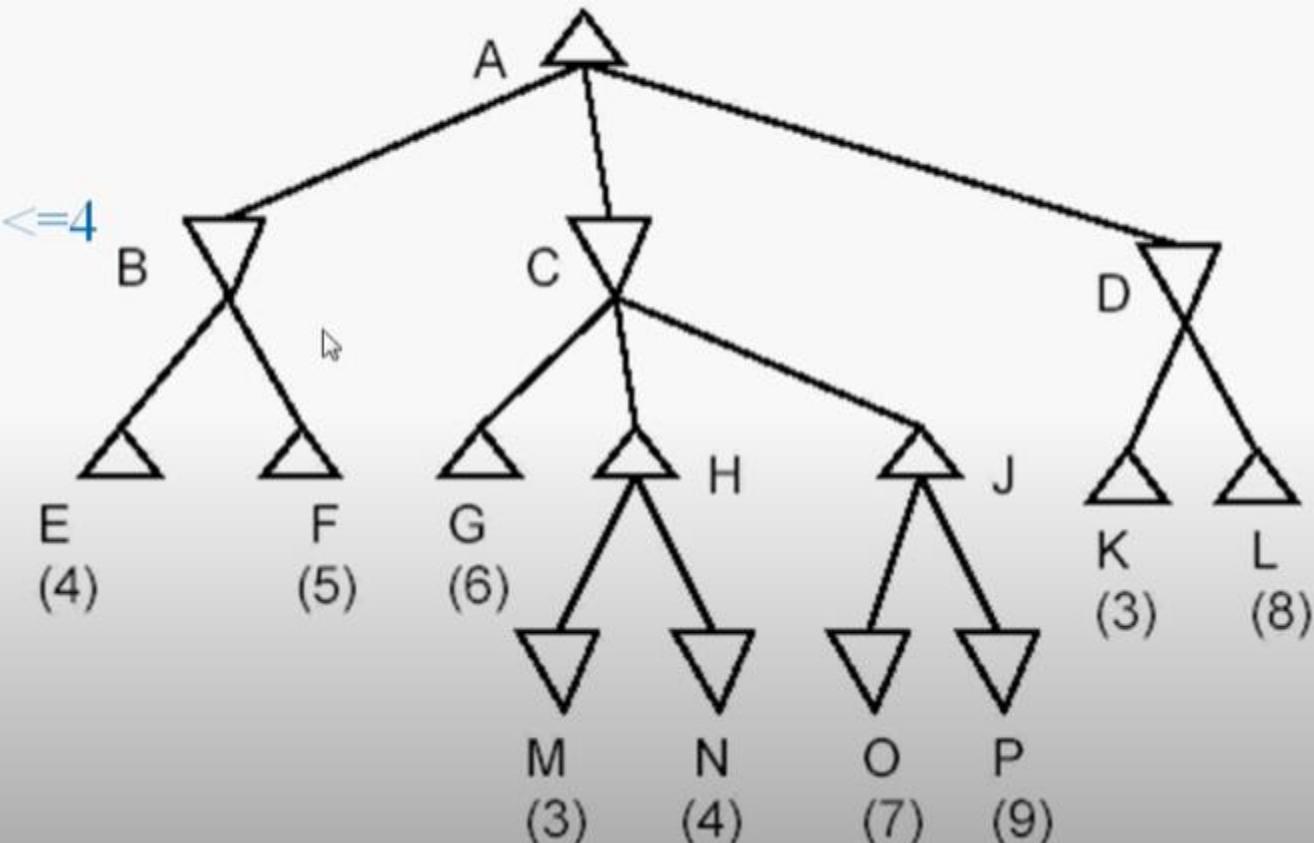
Step8

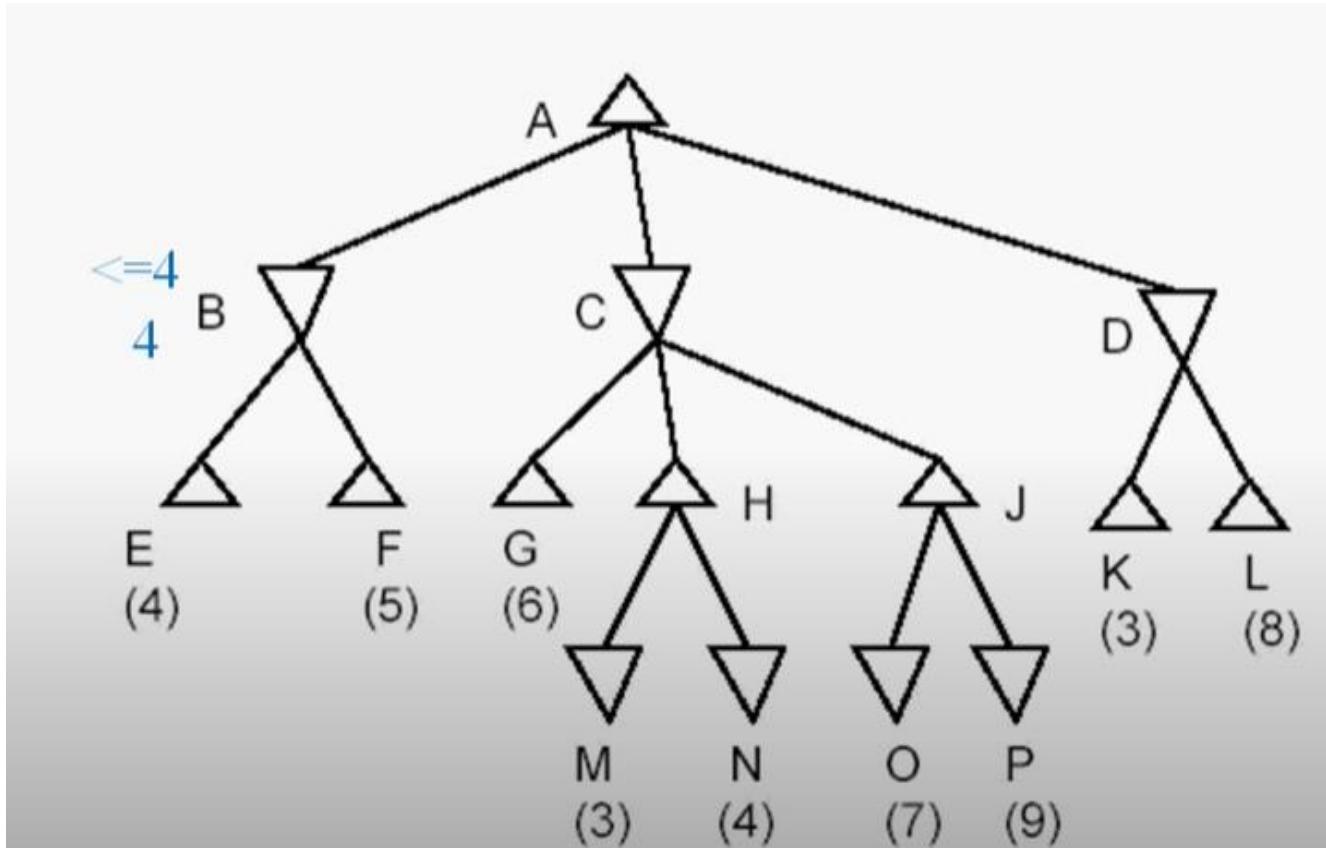


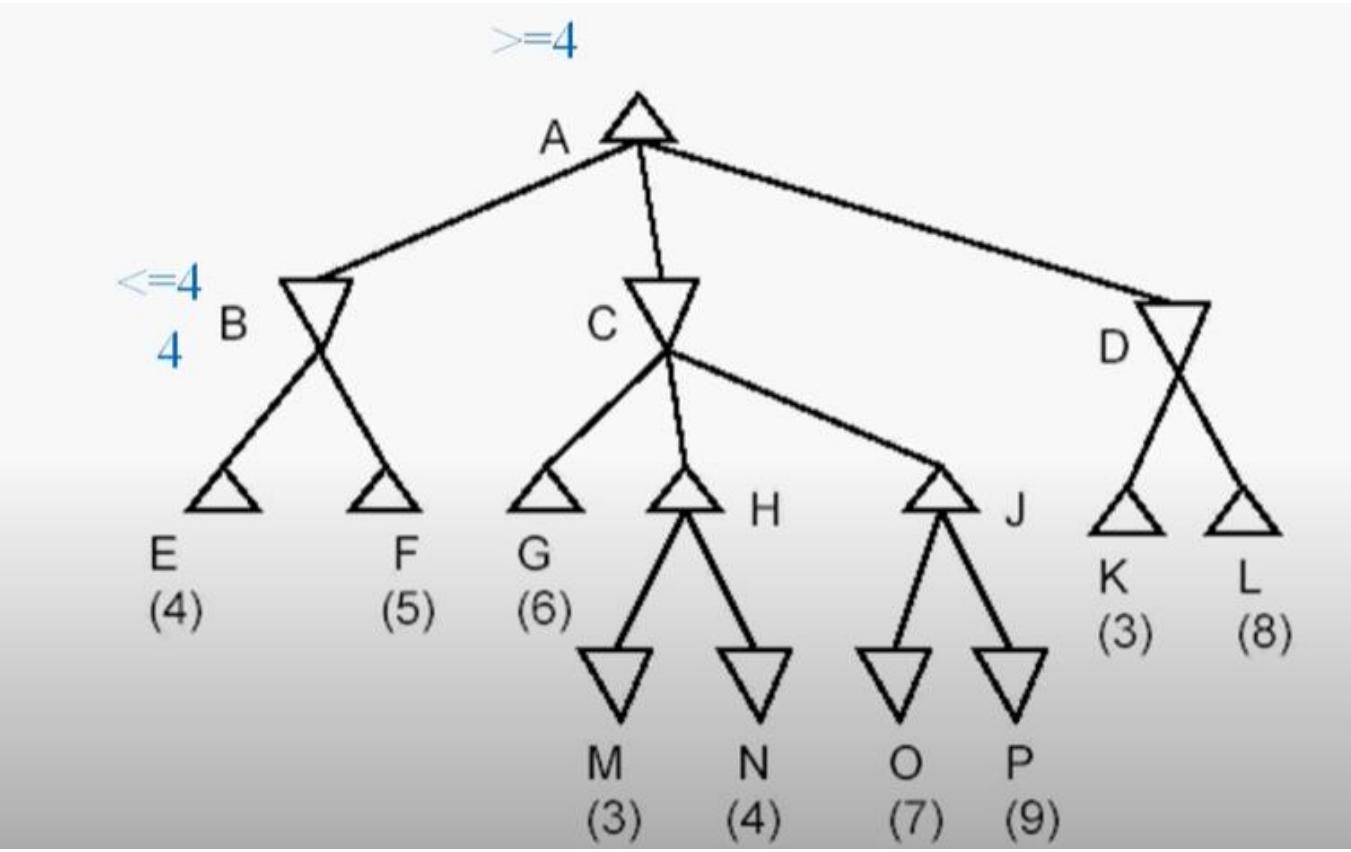
Step9

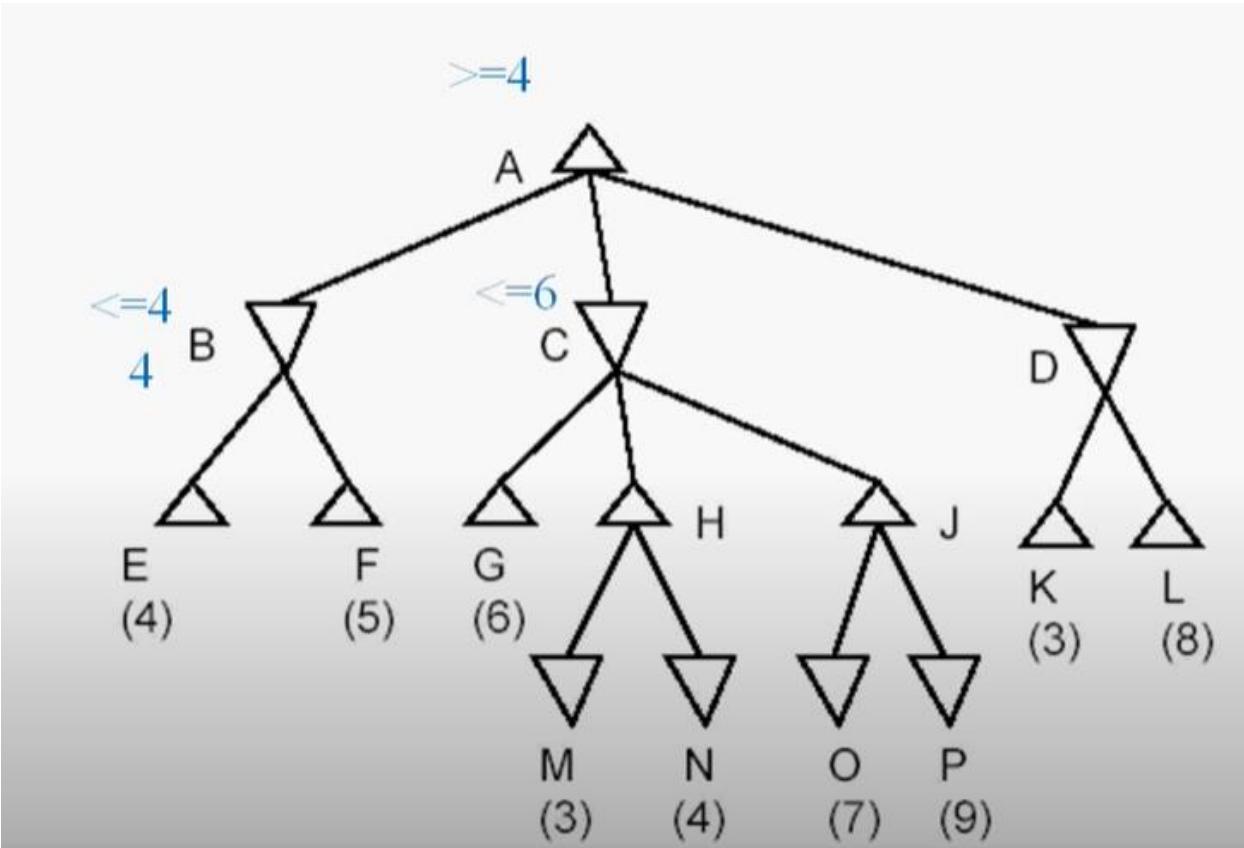


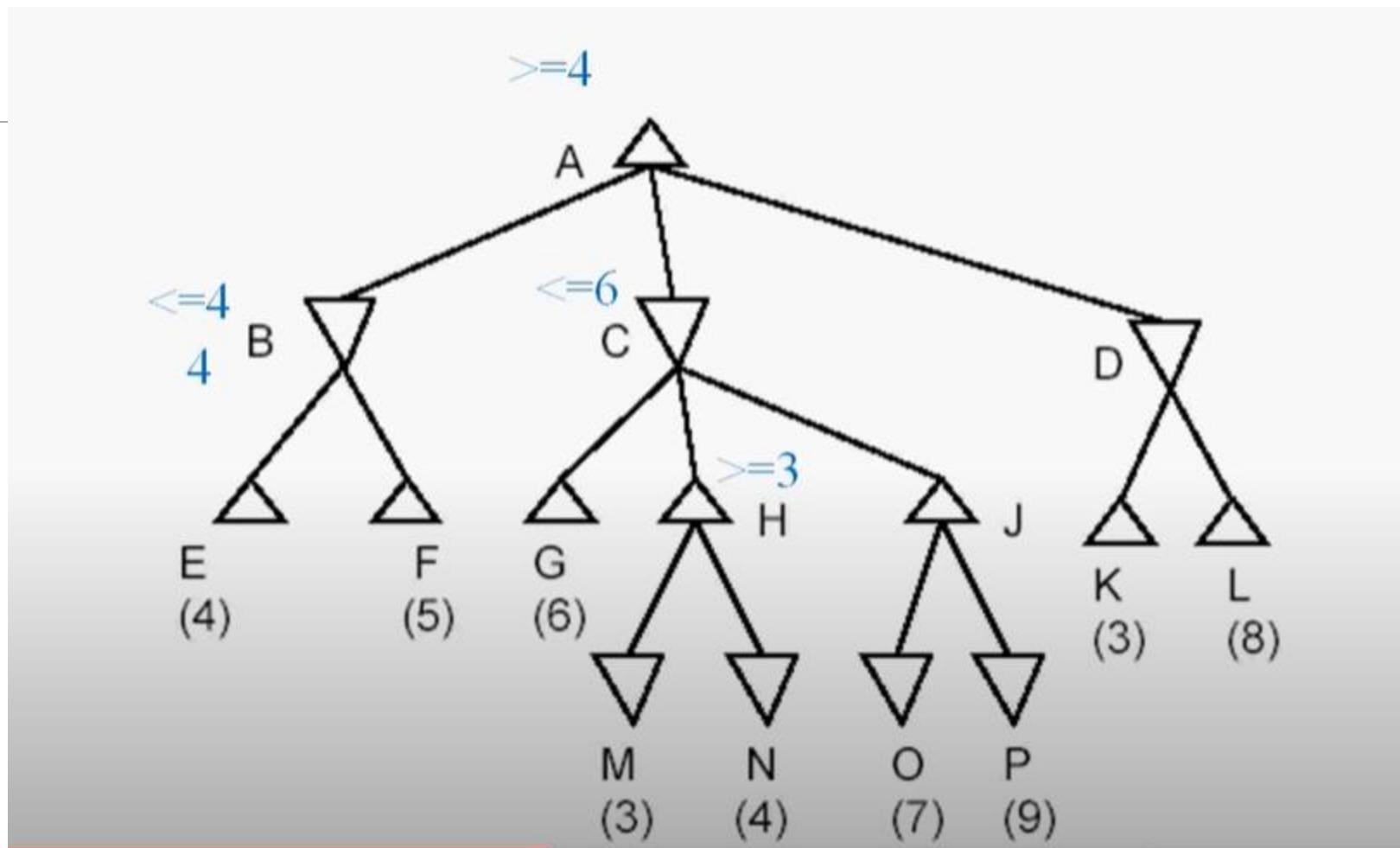


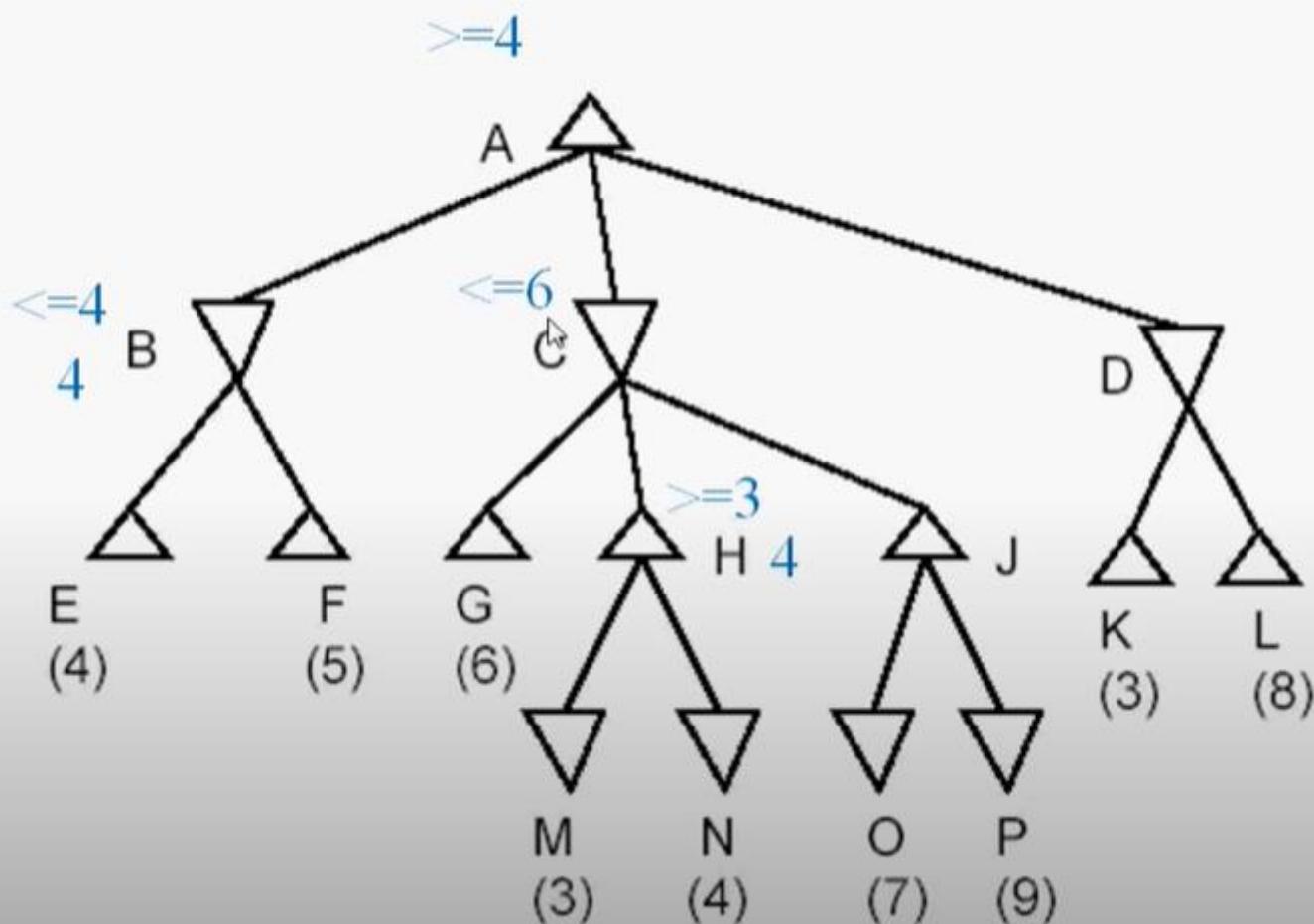


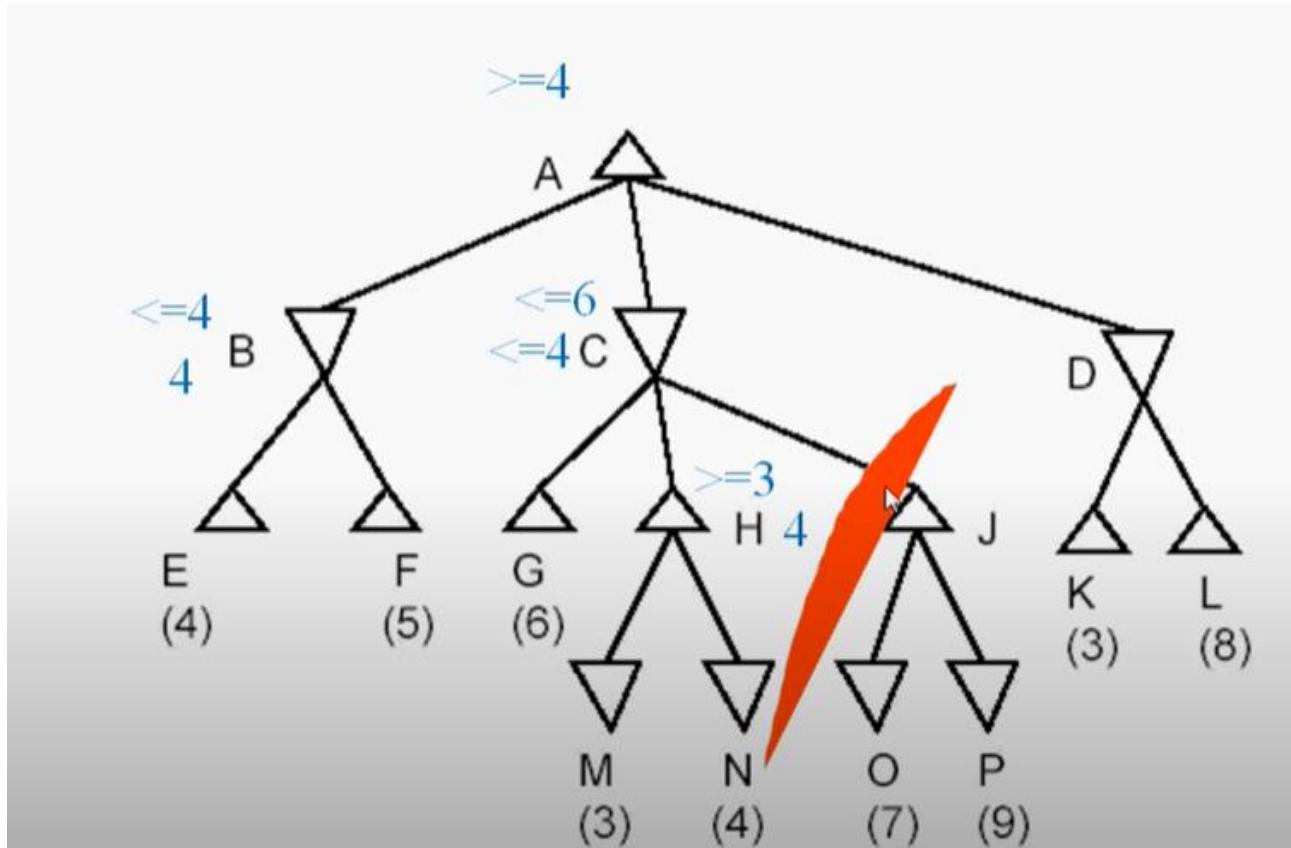


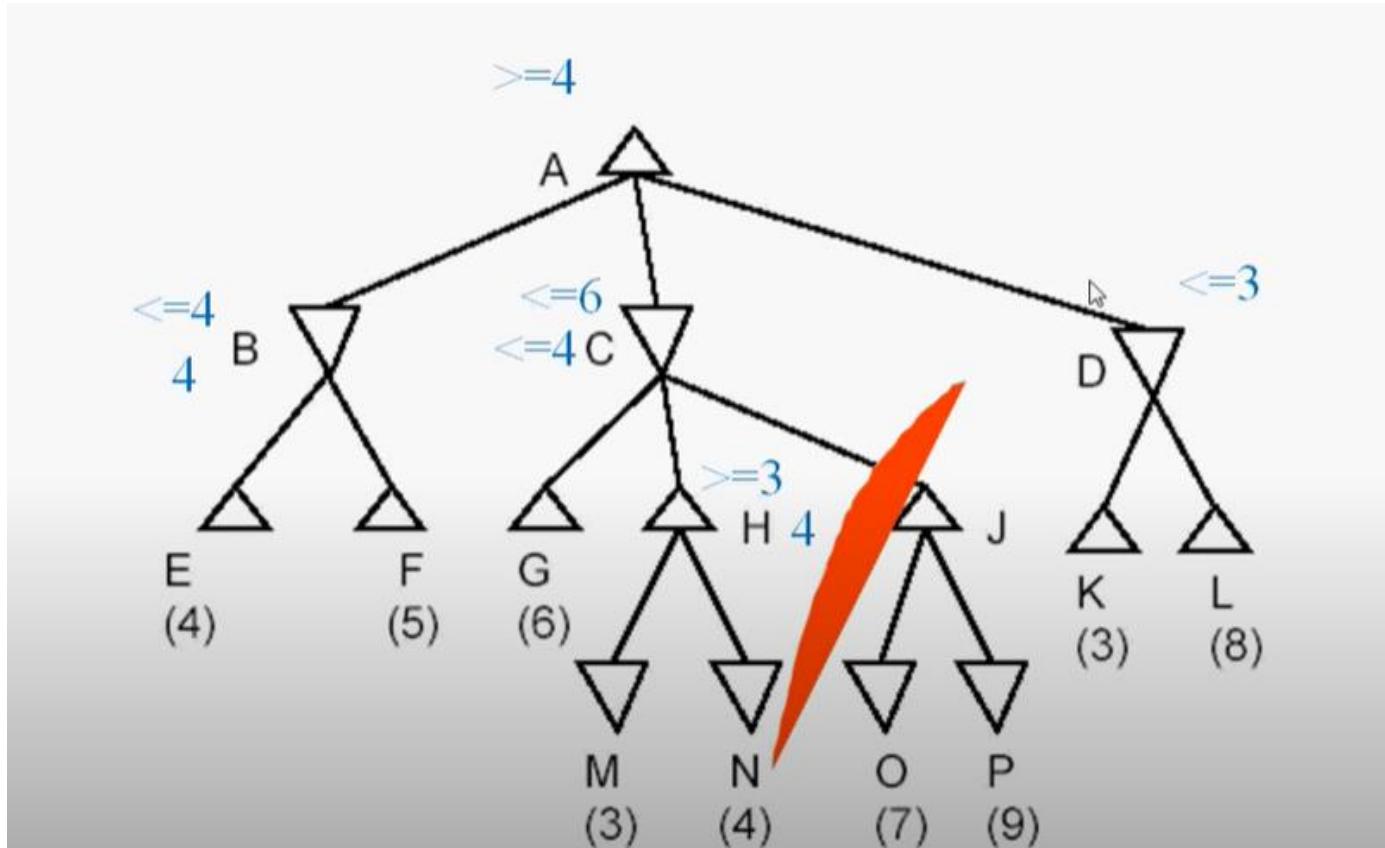


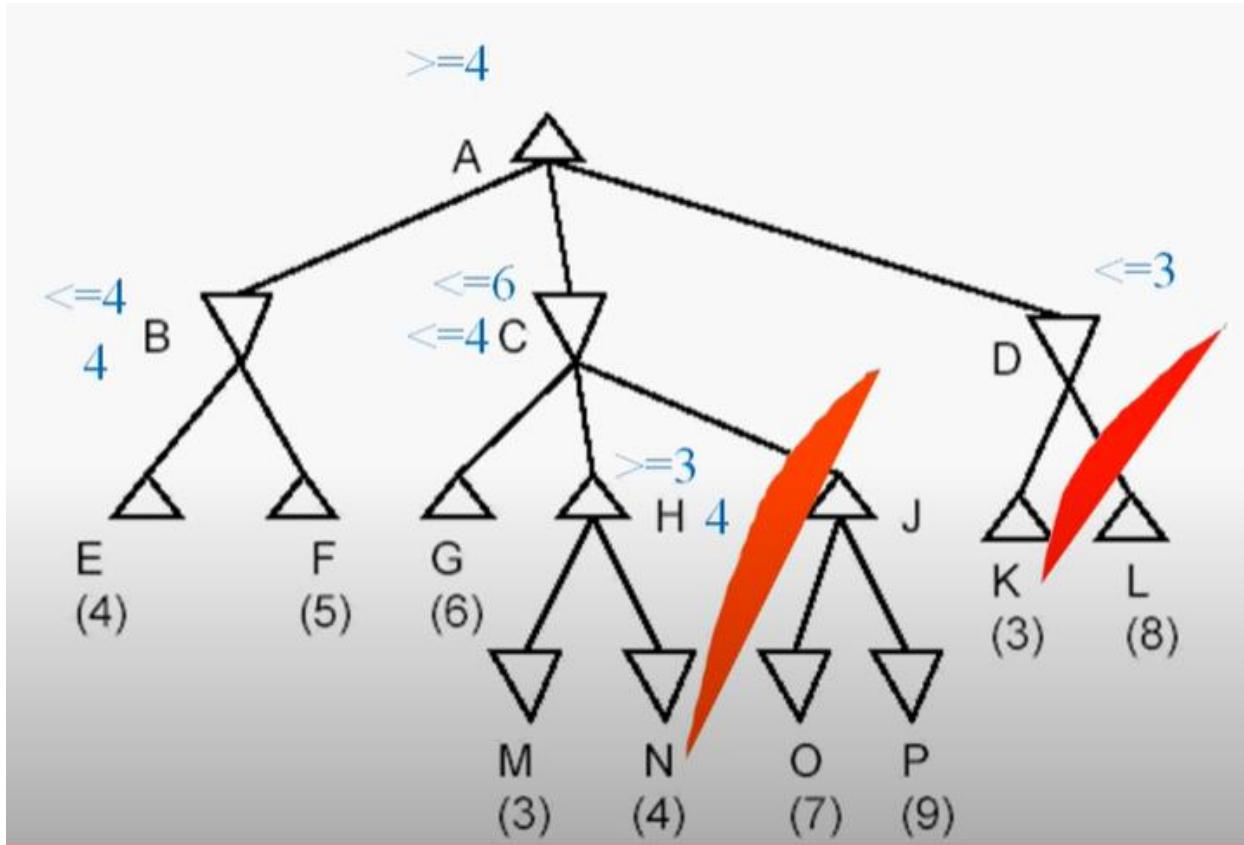












END