Search Strategies

Searching

- Searching is a process of finding a sequence of steps needed to solve any problem.

Uninformed vs Informed

- **Uninformed Search(blind search)** –
  - use only the information available in the problem definition
  - no additional information about states provided
  - generate successors and distinguish a goal state from a non-goal state.
  - A strategy is defined by picking the order of node expansion

- **Informed Search (heuristic search)-**
  - Strategies that know whether one non-goal state is "more promising" than another are called informed search or heuristic search strategies

Types of Uninformed Search

- Breadth first Search
- Depth first Search
- Uniform cost Search
- Depth Limited Search
- Iterative Deepening Search
- Bidirectional Search

Evaluating Search Strategies

- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time complexity**
  - How long (worst or average case) does it take to find a solution? Usually measured in terms of the number of nodes expanded
- **Space complexity**
  - How much space is used by the algorithm? Usually measured in terms of the maximum size of the "nodes" list during the search
- **Optimality/Admissibility**
  - If a solution is found, is it guaranteed to be an optimal one? That is, is it the one with minimum cost?

Evaluating Search Strategies(Contd..)

Time and space complexity are measured in terms of

- b—maximum branching factor of the search tree (maximum number of successors of any node)
- d—depth of the least-cost solution
- m—maximum depth of the state space (may be ∞)

Breadth First Search

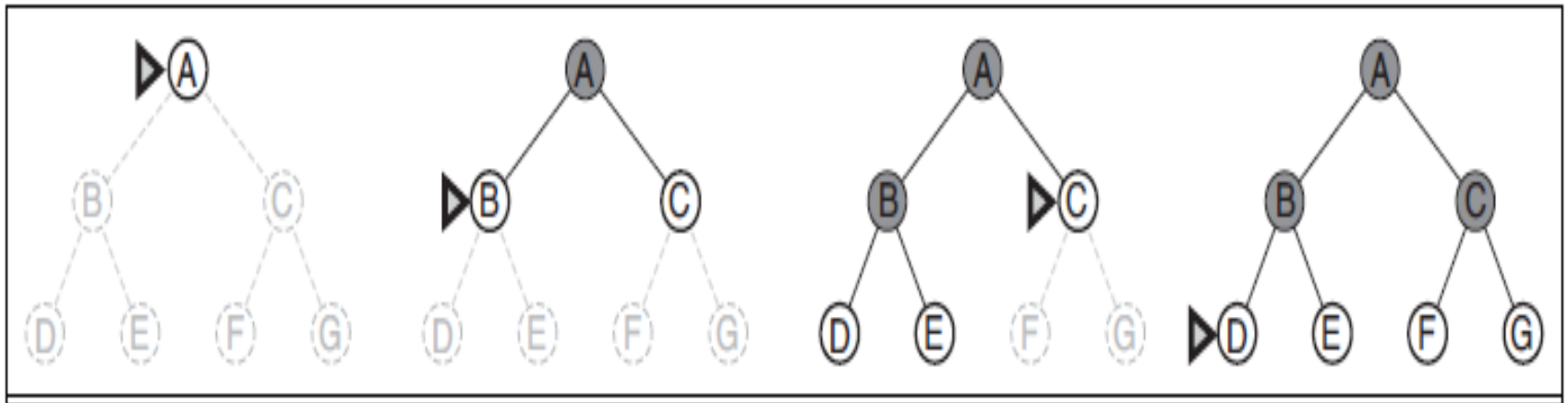


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

BFS-Properties

- Enqueue nodes on nodes in **FIFO** (first-in, first-out) order.
- **Complete**
- **Optimal** (i.e., admissible) if all operators have the same cost. Otherwise, not optimal but finds solution with shortest path length.
- **Exponential time and space complexity**, $O(b^d)$, where d is the depth of the solution and b is the branching factor (i.e., number of children) at each node
- Will take a **long time to find solutions** with a large number of steps because must look at all shorter length possibilities first . A complete search tree of depth d where each non-leaf node has b children, has a total of

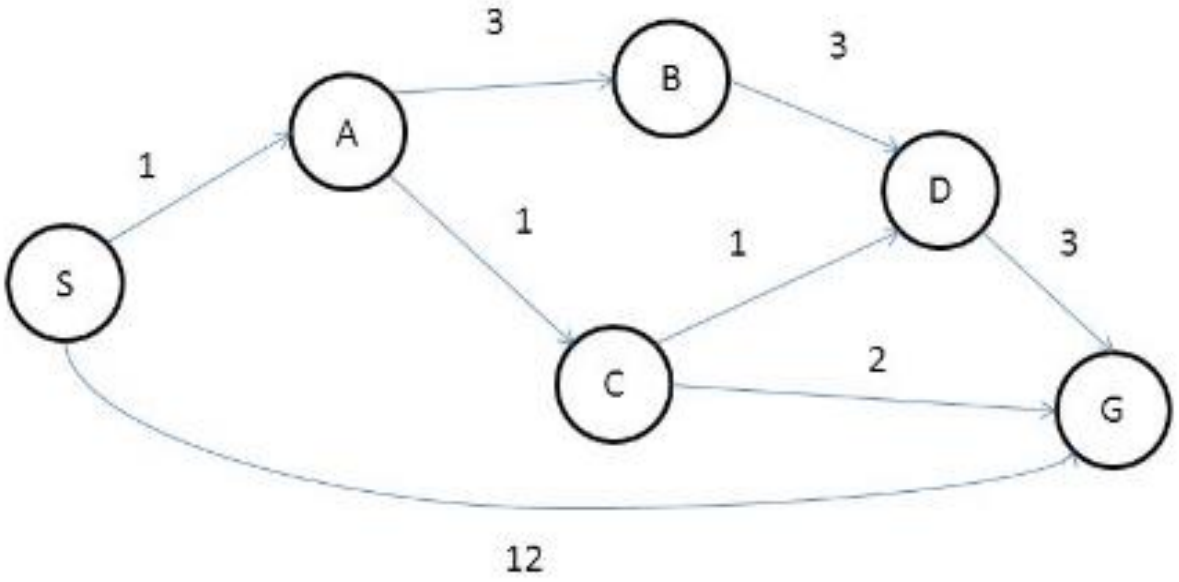
$$1 + b + b^2 + \ldots + b^d = (b^{(d+1)} - 1)/(b-1) \text{ nodes}$$

Time and memory requirements

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Uniform Cost Search



Initialization: { [S , 0] }
Iteration1: { [S->A , 1] , [S->G , 12] }
Iteration2: { [S->A->C , 2] , [S->A->B , 4] , [S->G , 12] }
Iteration3: { [S->A->C->D , 3] , [S->A->B , 4] , [S->A->C->G , 4] , [S->G , 12] }
Iteration4: { [S->A->B , 4] , [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->G , 12] }
Iteration5: { [S->A->C->G , 4] , [S->A->C->D->G , 6] , [S->A->B->D , 7] , [S->G , 12] }
Iteration6 gives the final output as S->A->C->G.

Uniform Cost Search

- Expand the cheapest unexpanded node
- Implementation:
  - frontier = priority queue ordered by path cost g(n)
- Equivalent to breadth-first search, if all step costs are equal

UCS-Properties

Complete?? Yes, if step cost $\geq \epsilon > 0$

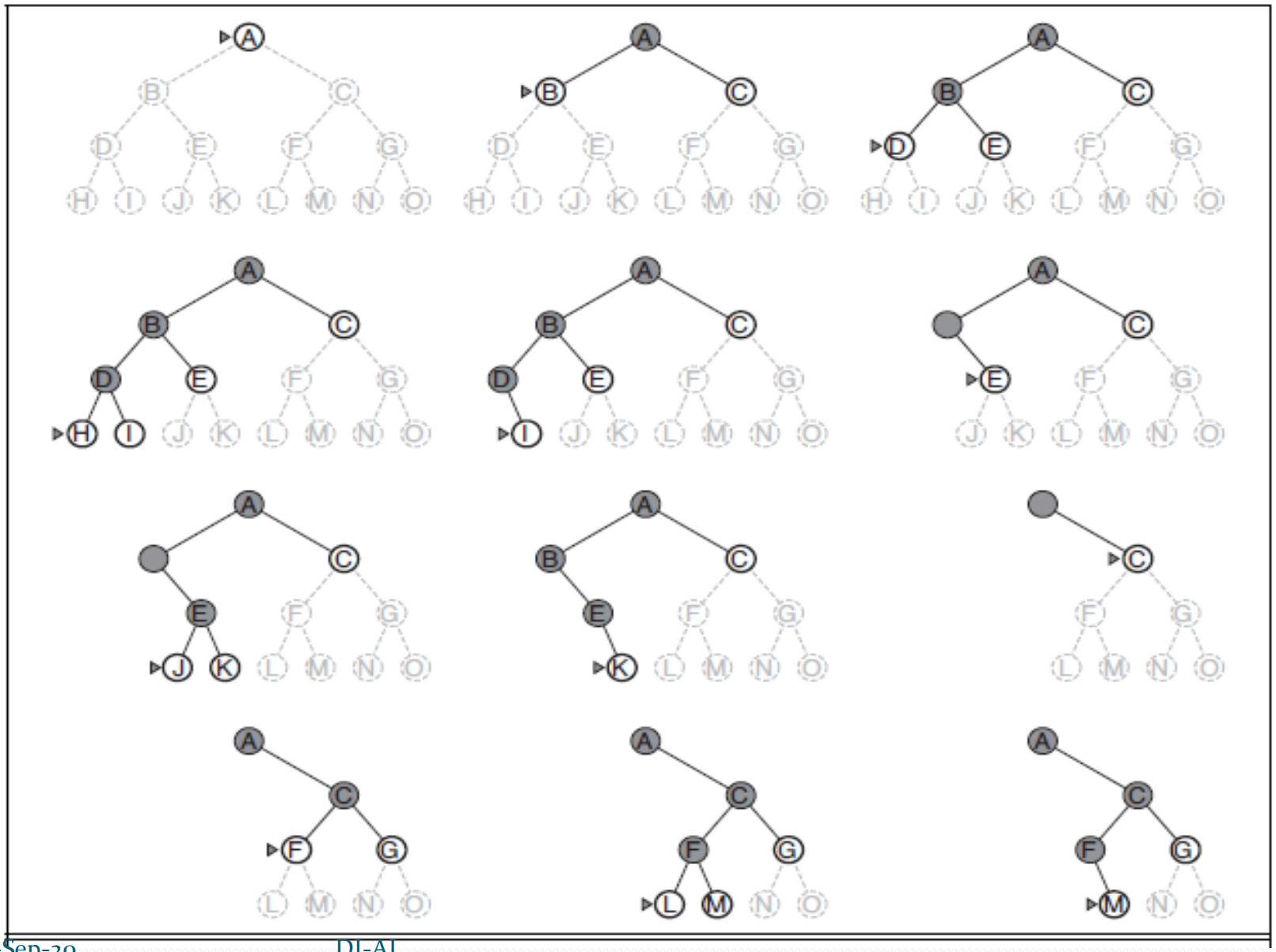
Time?? # of nodes with $g(n) \leq C^*$, i.e., $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution
and ϵ is the minimal step cost

Space?? Same as time

Optimal?? Yes—nodes are expanded in increasing order of $g(n)$

Depth-first search

- **Depth-first search** always expands the *deepest* node in the current frontier of the search tree.
- The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.
- DFS uses a LIFO queue- the most recently generated node is chosen for expansion


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

DFS-Properties

Complete?? No: it fails in infinite-depth spaces

it also fails in finite spaces with loops

but if we modify the search to avoid repeated states

$\Rightarrow$ complete in finite spaces (even with loops)

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, it may be much faster than breadth-first

Space?? $O(bm)$: i.e., linear space!

Optimal?? No

Depth-limited search

- Depth-first search with depth limit *l*, i.e., nodes at depth *l* have no successors
- The depth limit solves the infinite-path problem.

Iterative deepening search

- **Iterative deepening search** (or iterative deepening depth-first search) is a general strategy, used in combination with depth-first tree search, that finds the best depth limit.
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found.
- This will occur when the depth limit reaches d, the depth of the shallowest goal node.

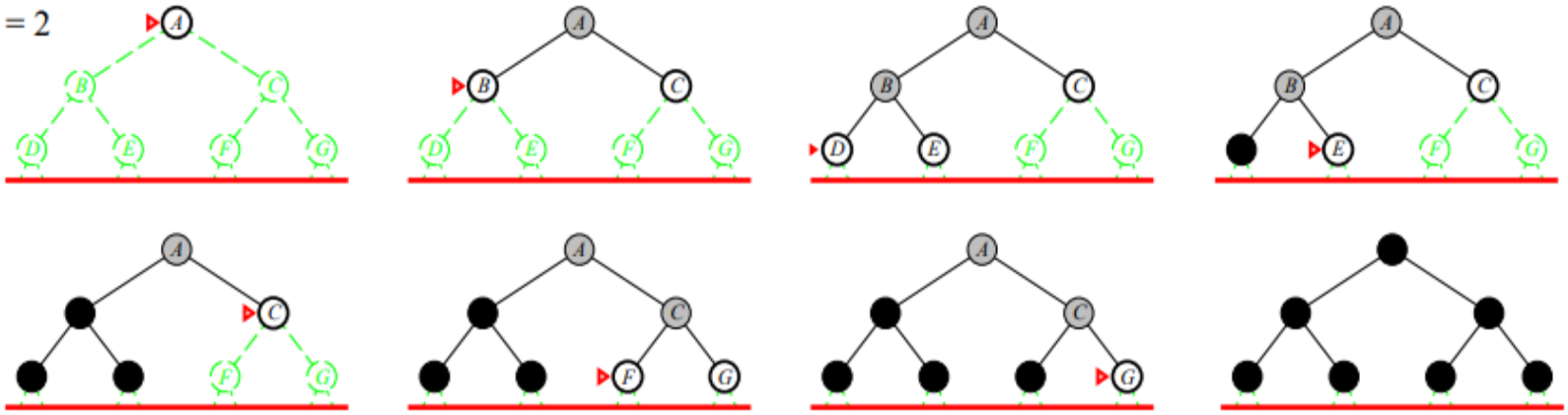
Iterative deepening search *l = 0*

Iterative deepening search $l = 1$

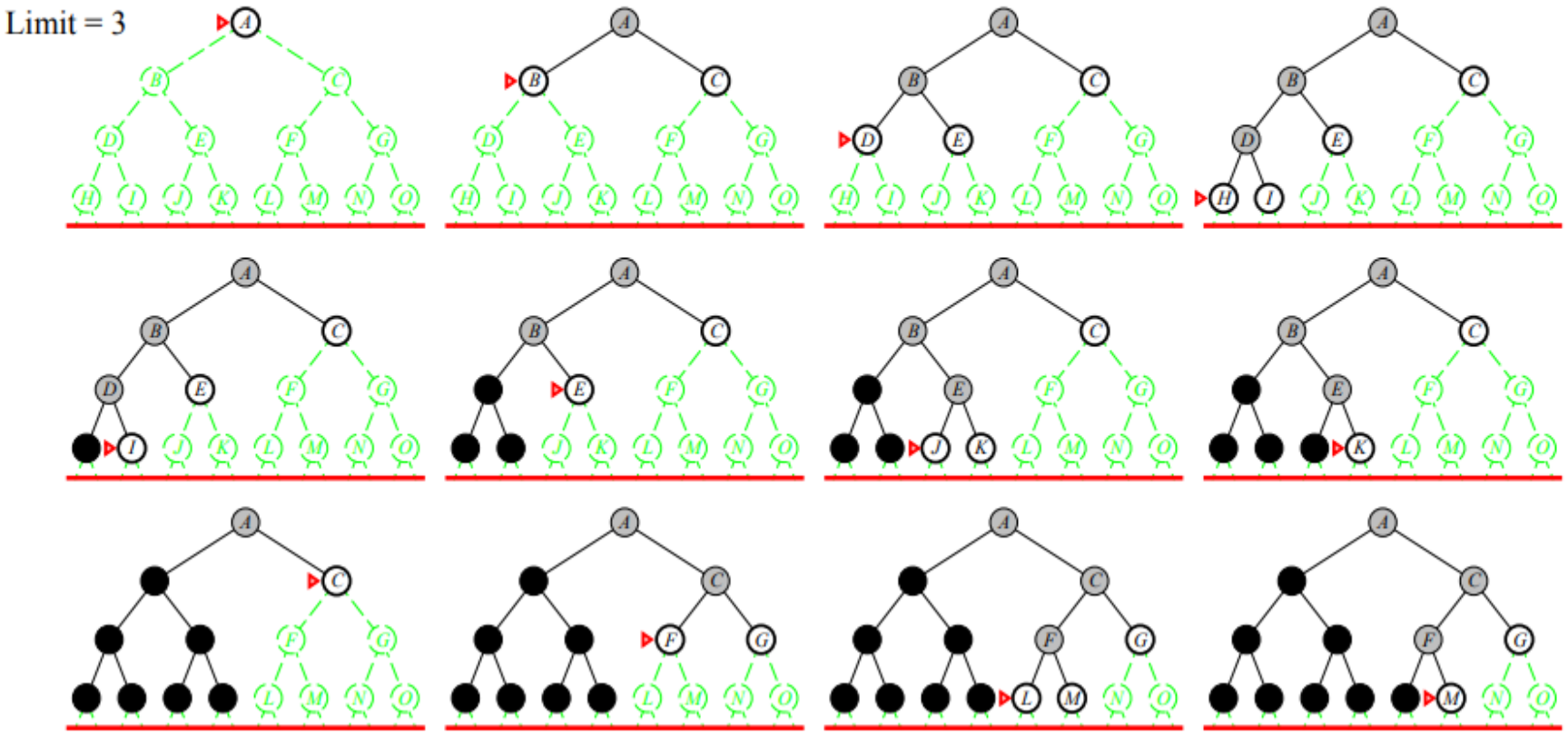
Limit = 1

Iterative deepening search $l = 2$

Limit = 2



Iterative deepening search $l = 3$



IDS- Properties

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$
it can be modified to explore a uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$:

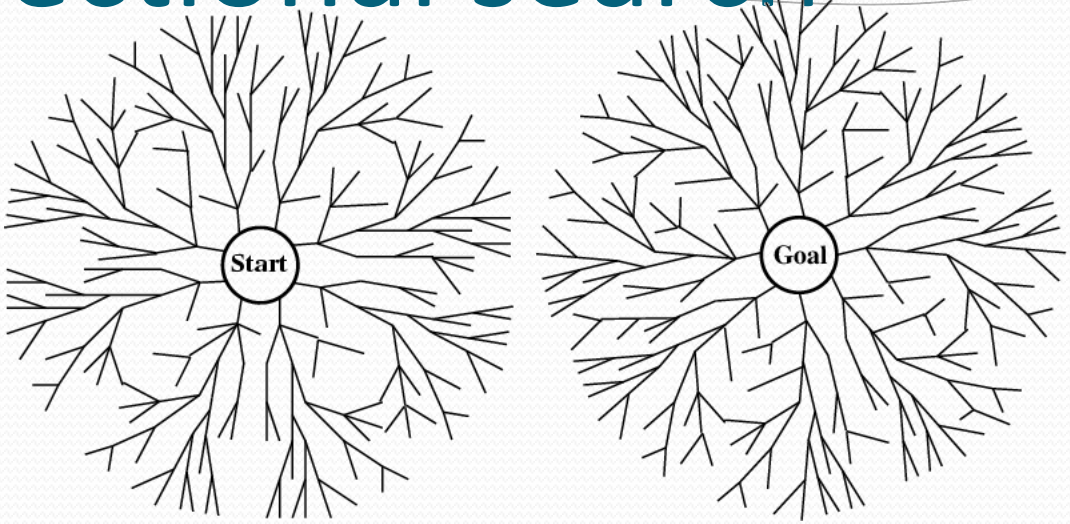
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$
$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

Note: IDS recalculates shallow nodes several times,
but this doesn't have a big effect compared to BFS!

IDS

- *In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known*

Bi-directional search



- Alternate searching from the start state toward the goal and from the goal state toward the start.
- Stop when the frontiers intersect.
- Works well only when there are unique start and goal states.
- Requires the ability to generate "predecessor" states.
- Can (sometimes) lead to finding a solution more quickly.
- Time complexity: $O(b^{d/2})$. Space complexity: $O(b^{d/2})$.

Summary of algorithms

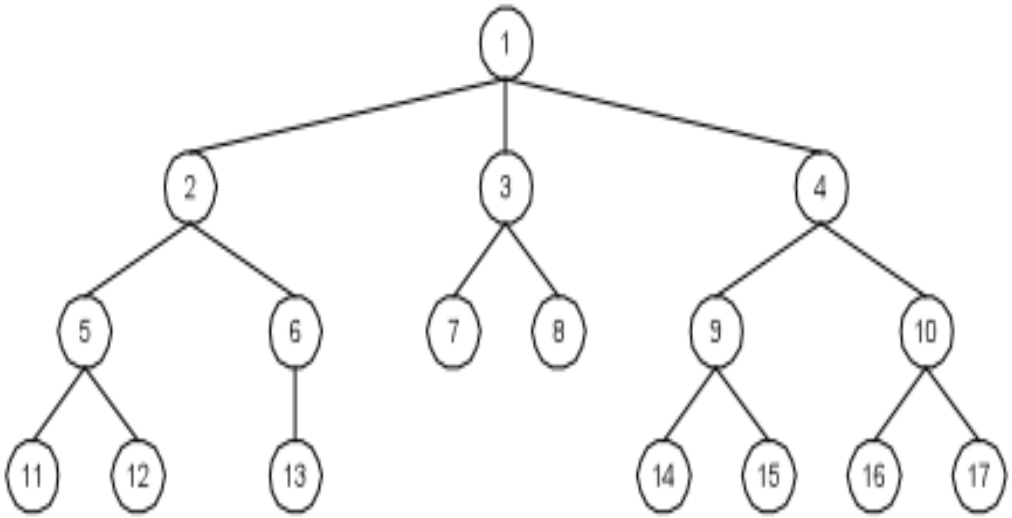
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

b = the branching factor
d = the depth of the shallowest solution
m = the maximum depth of the tree
l = the depth limit
ϵ = the smallest step cost
C^* = the cost of the optimal solution

Homework problem - 1

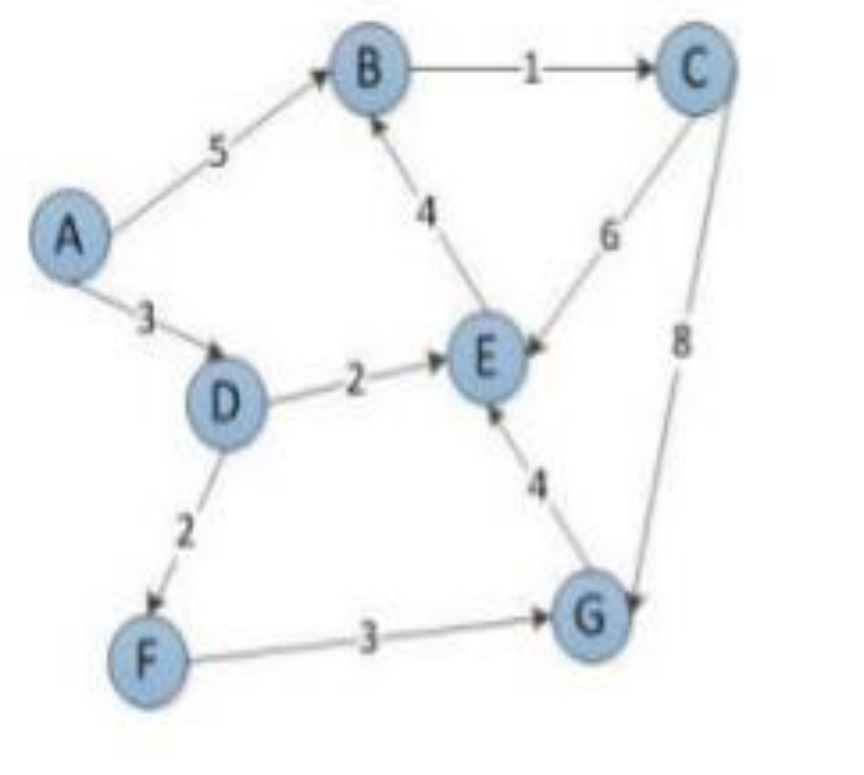
List the order in which the nodes are visited for the following three search strategies

a. Depth-First Search
b. Depth-First Iterative-Deepening Search
c. Breath-First Search



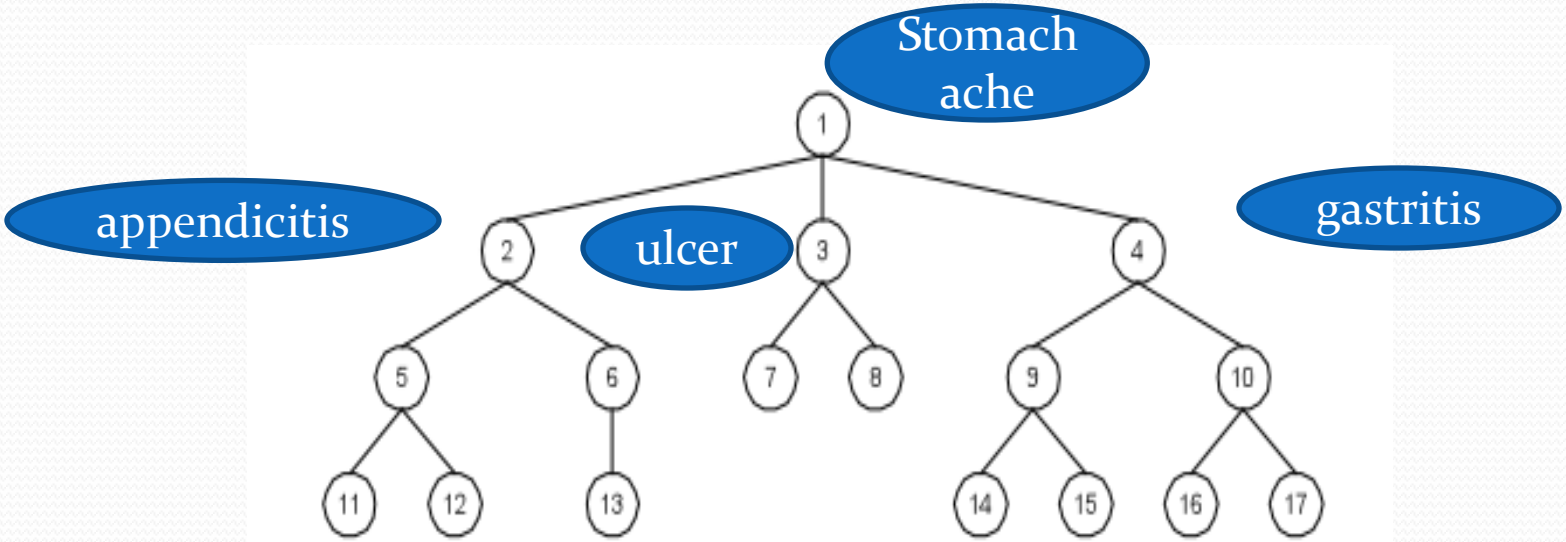
Homework problem - 2

List the order in which the nodes are visited based on uniform cost search



Informed Search

- **Informed search** strategy—one that uses problem-specific knowledge beyond the definition of the problem itself—can find solutions more efficiently than an uninformed strategy.



Informed Search

- Informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc.
- This knowledge help agents to explore less to the search space and find more efficiently the goal node.
- The informed search algorithm is more useful for large search space.
- Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function

- **Heuristics function:** A function used in Informed Search, and it finds the most promising path.
- The heuristic method, however, might not always give the best solution, but it is guaranteed to find a good solution in reasonable time.
- Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Heuristic search

- Expands nodes based on their heuristic value h(n).
- It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which are already expanded and in the OPEN list, it places nodes which are yet to be expanded.
- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list.
- The algorithm continues unit a goal state is found.

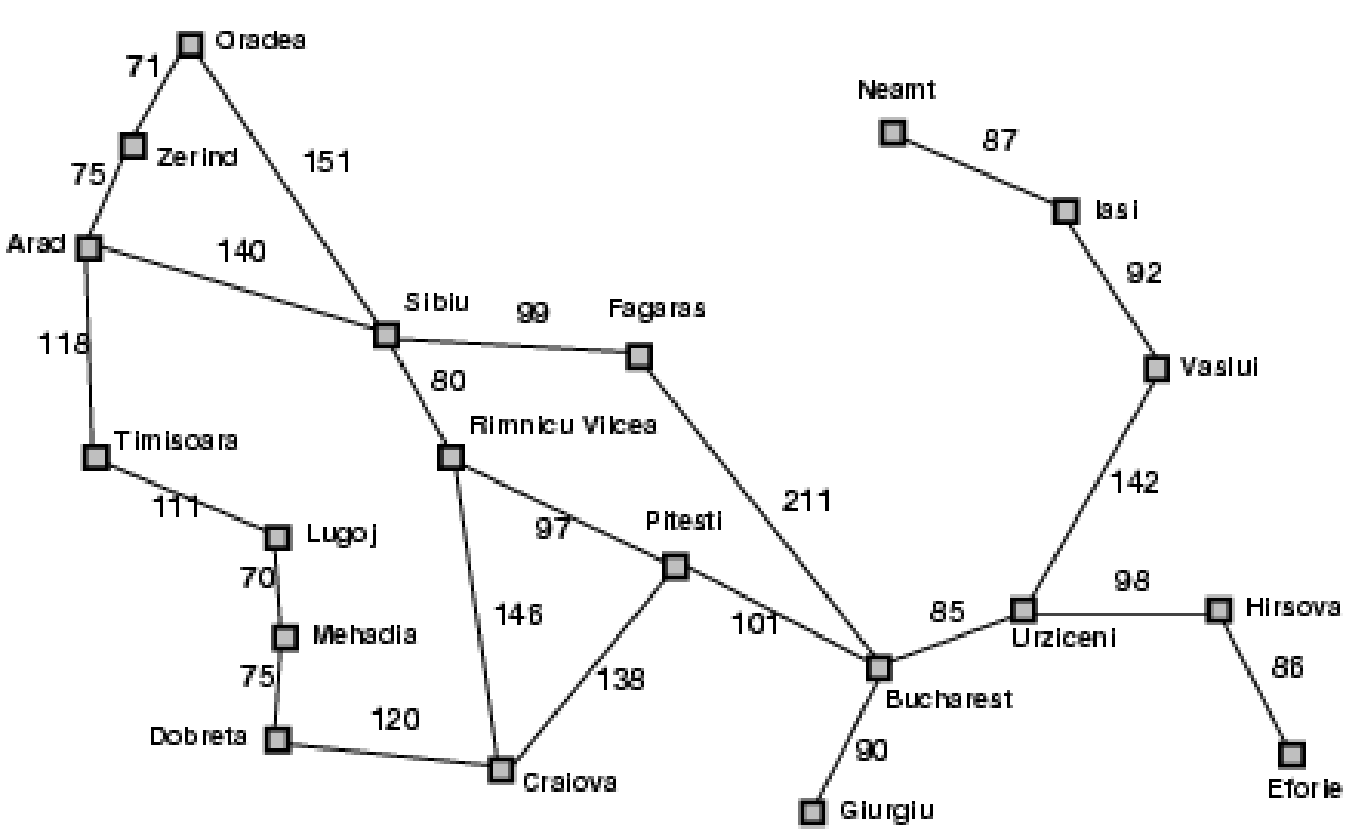
Best-first search

- Idea: use an evaluation function *f(n)* for each node
  - f(n) provides an estimate for the total cost.
  - → Expand the node n with smallest f(n).

- Implementation:
  Order the nodes in fringe increasing order of cost.

- Special cases:
  - greedy best-first search
  - A^* search

Romania with straight-line dist.

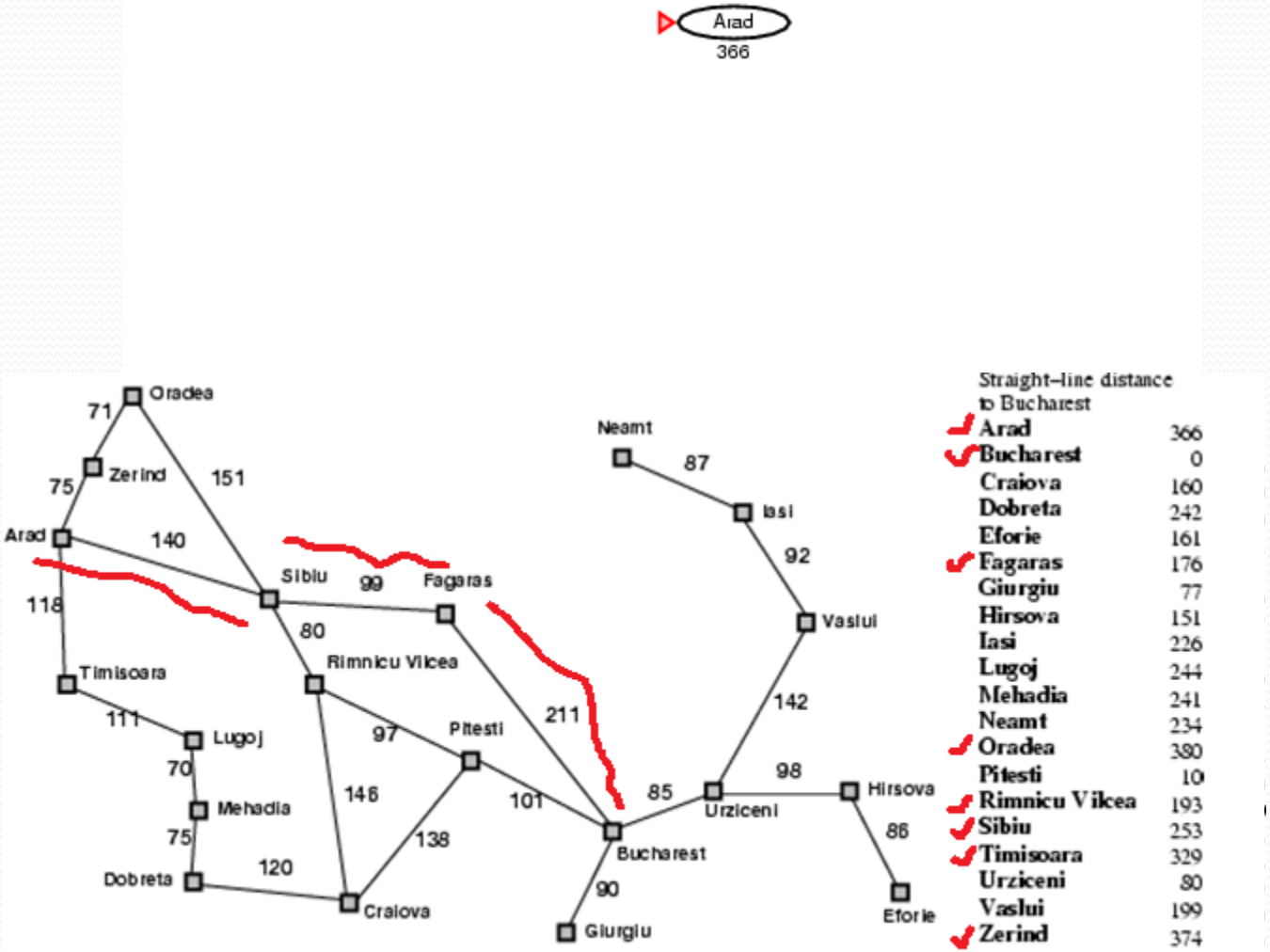


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

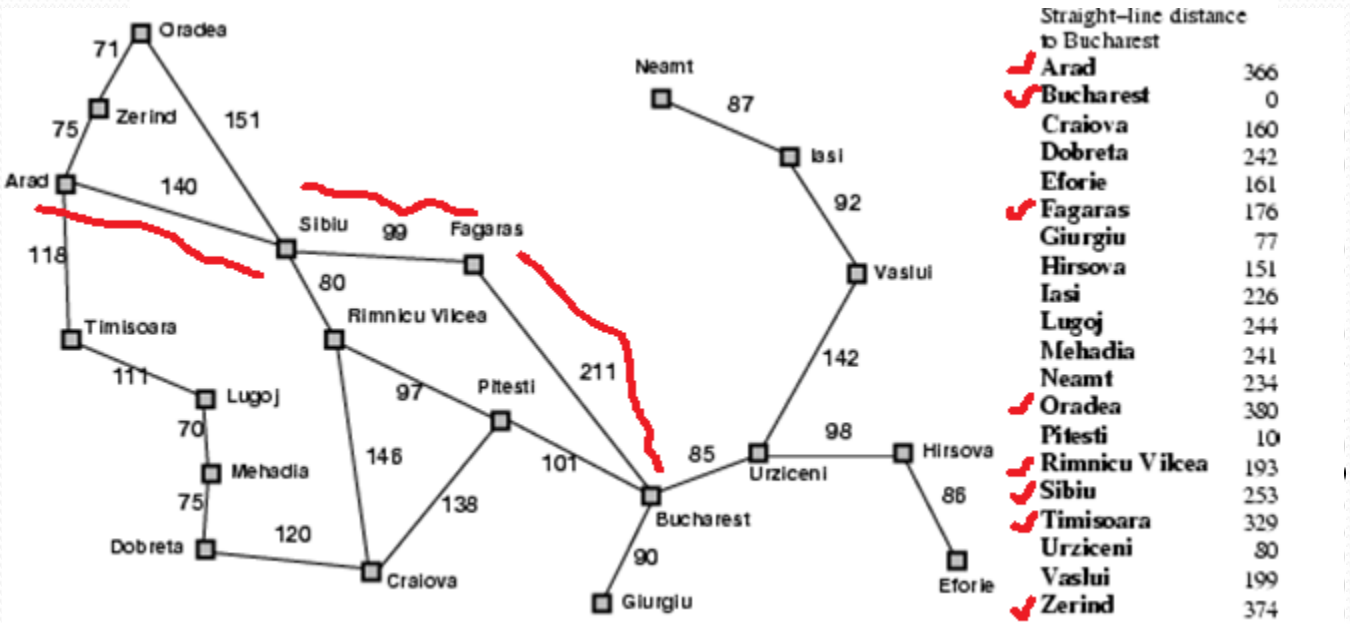
Greedy best-first search

- f(n) = estimate of cost from *n* to *goal*
- e.g., $f_{SLD}(n)$ = straight-line distance from *n* to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal.
- Best First search has $f(n)=h(n)$

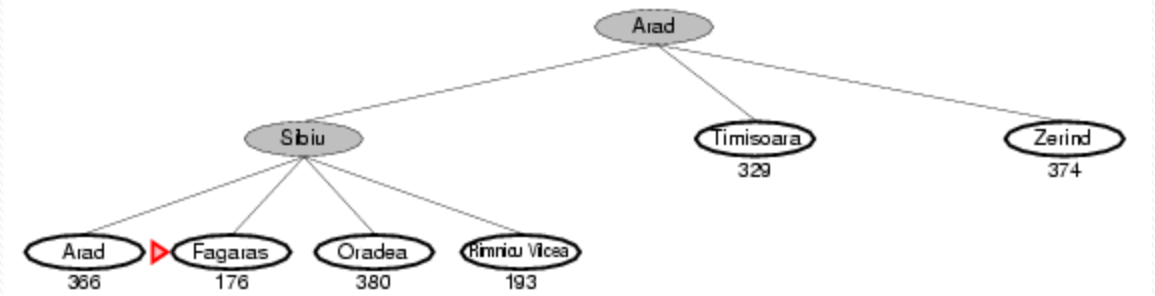
Greedy best-first search example

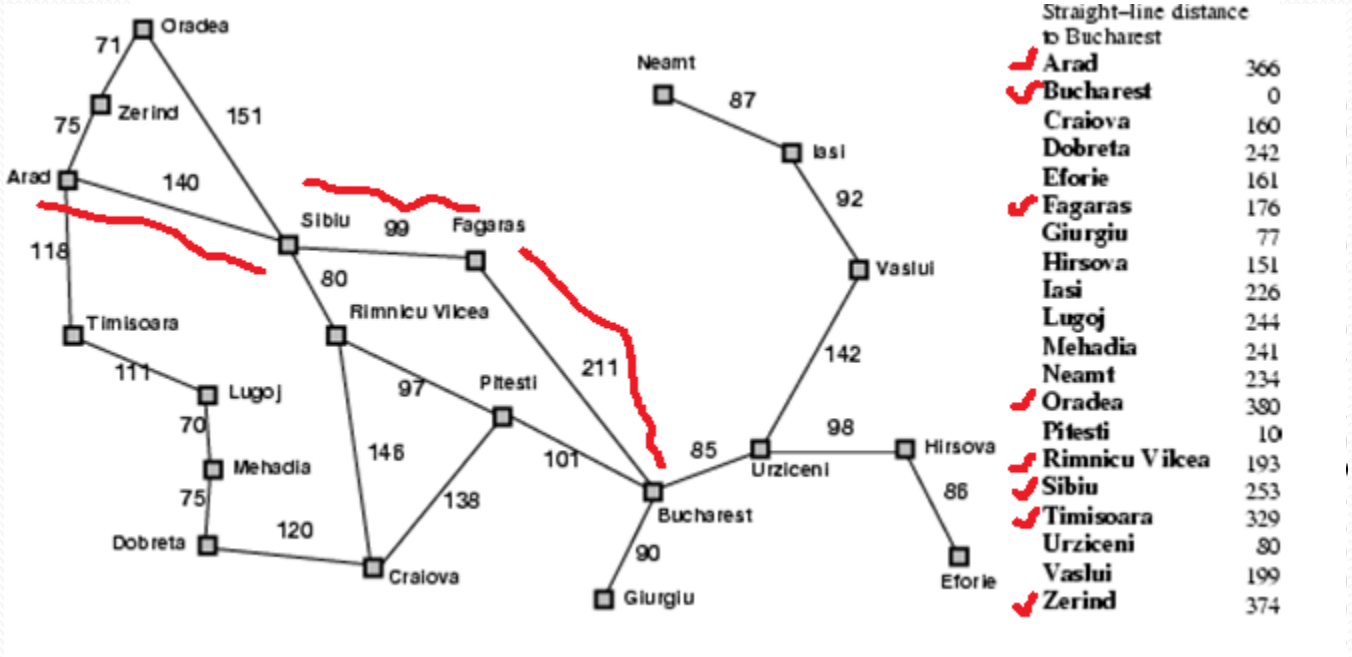


Greedy best-first search example

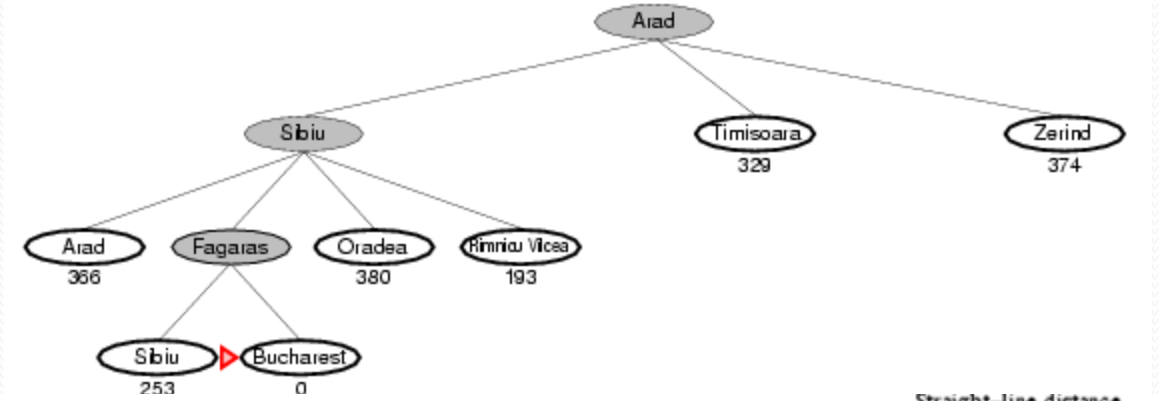


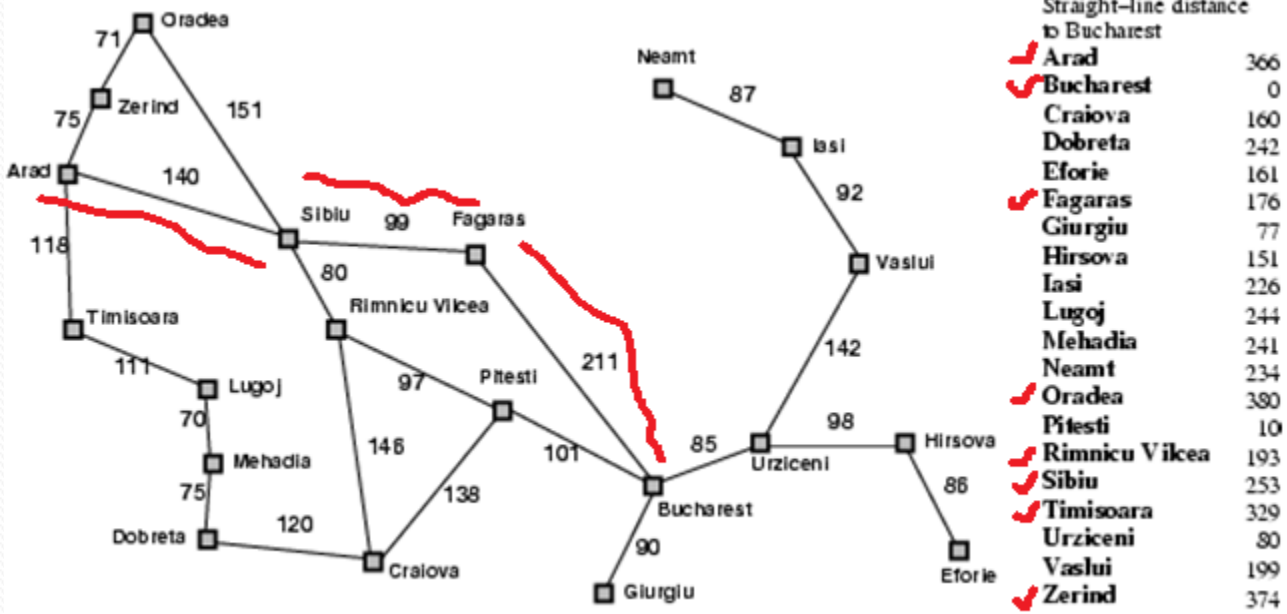
Greedy best-first search example





Greedy best-first search example

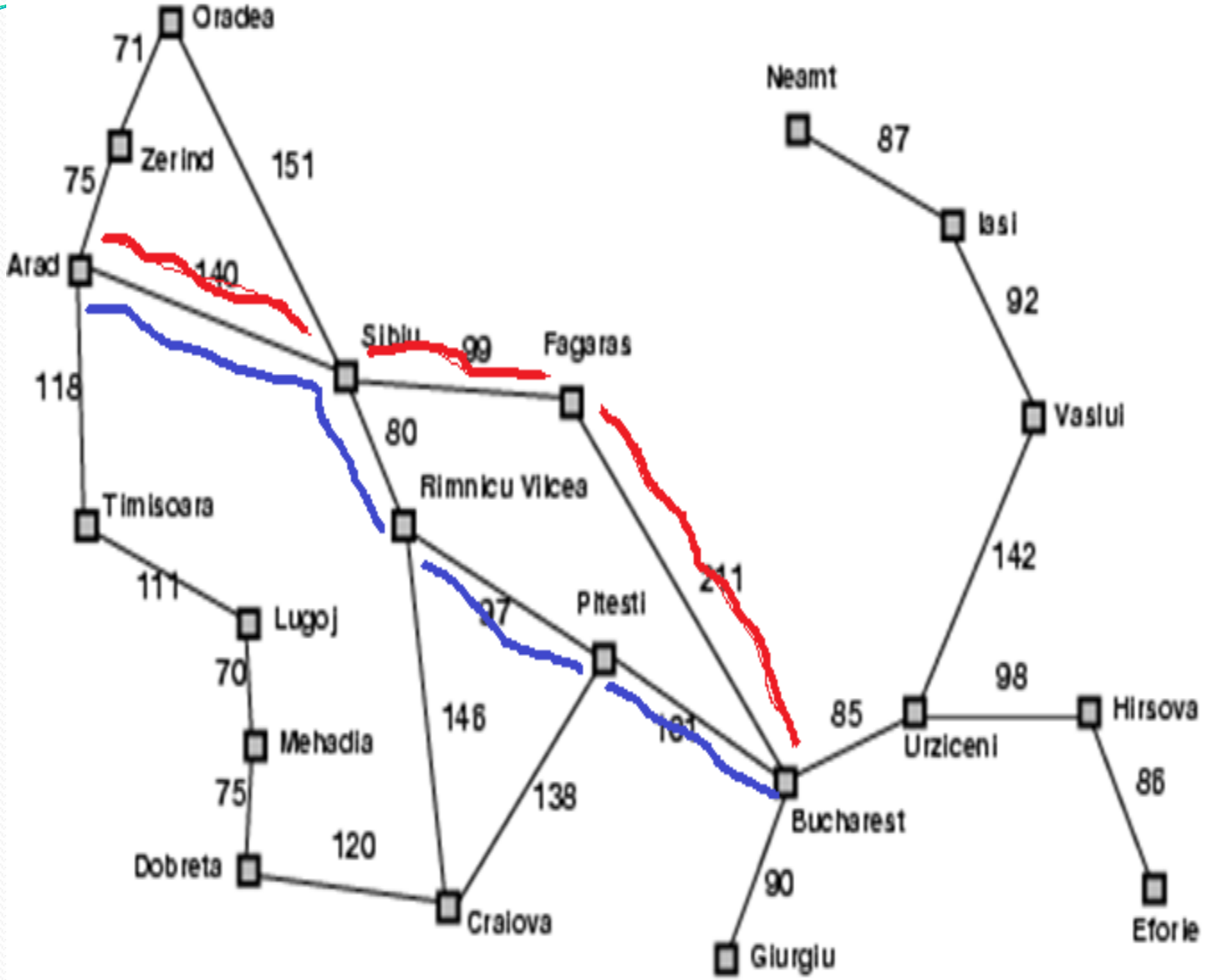




Properties of greedy best-first search

- Complete? No – can get stuck in loops.
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ - keeps all nodes in memory
- Optimal? No

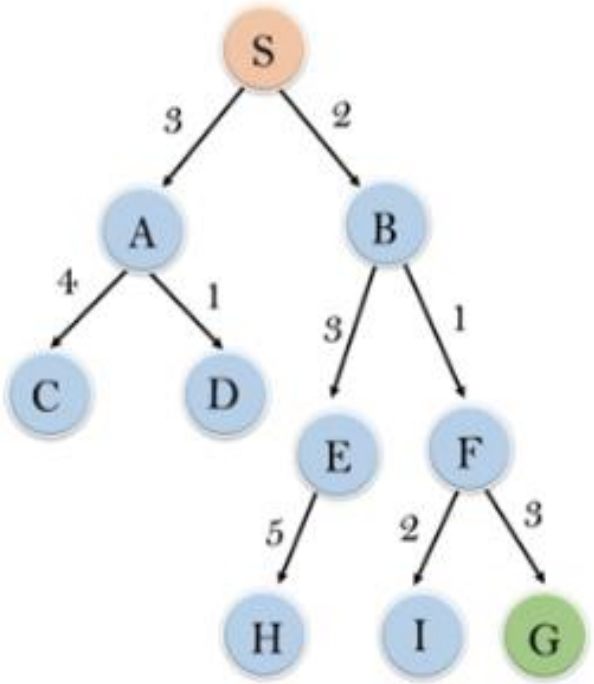
  e.g. Arad→Sibiu→Rimnicu Virea→Pitesti→Bucharest is shorter!



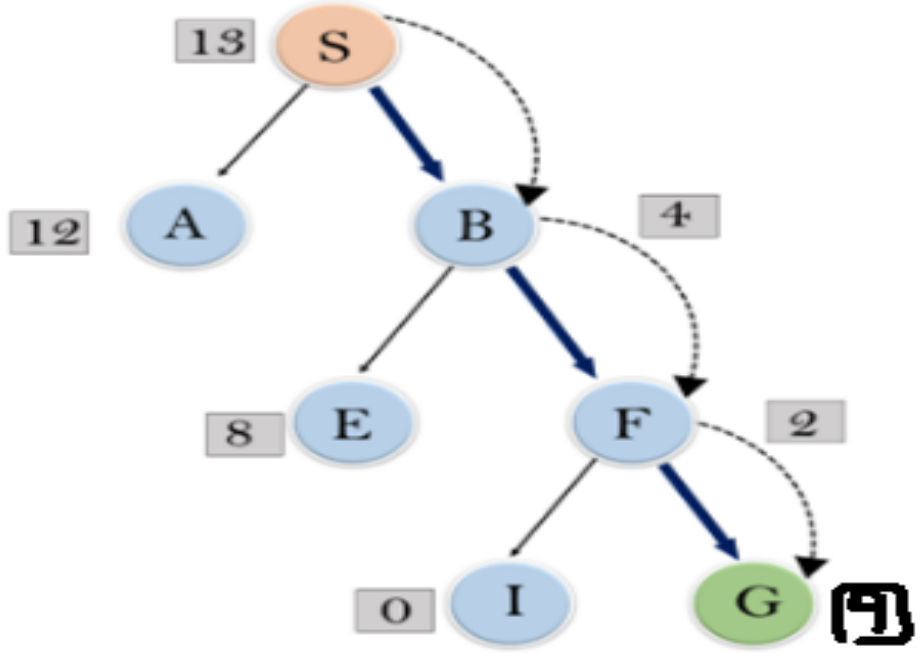
Straight-line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Home work problem -3



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]
: Open [E, A], Closed [S, B, F]

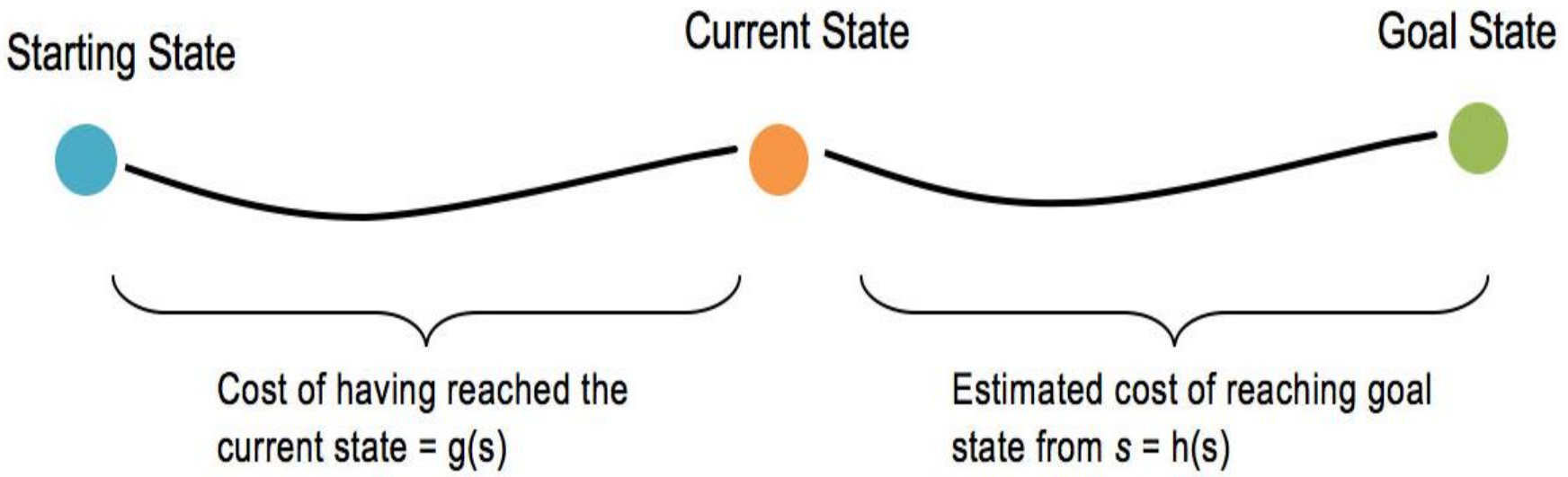
Iteration 3: Open [I, G, E, A], Closed [S, B, F]
: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: S----> B----->F----> G

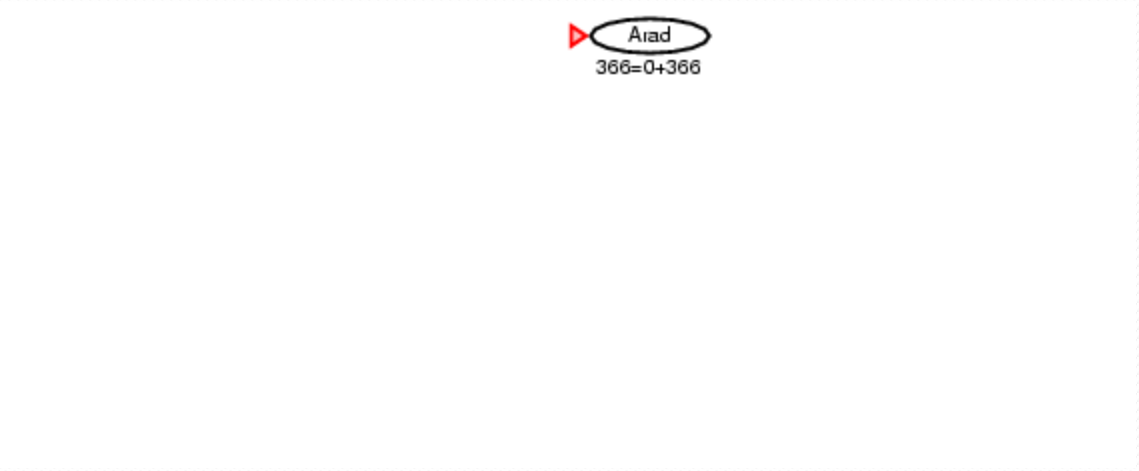
A* search

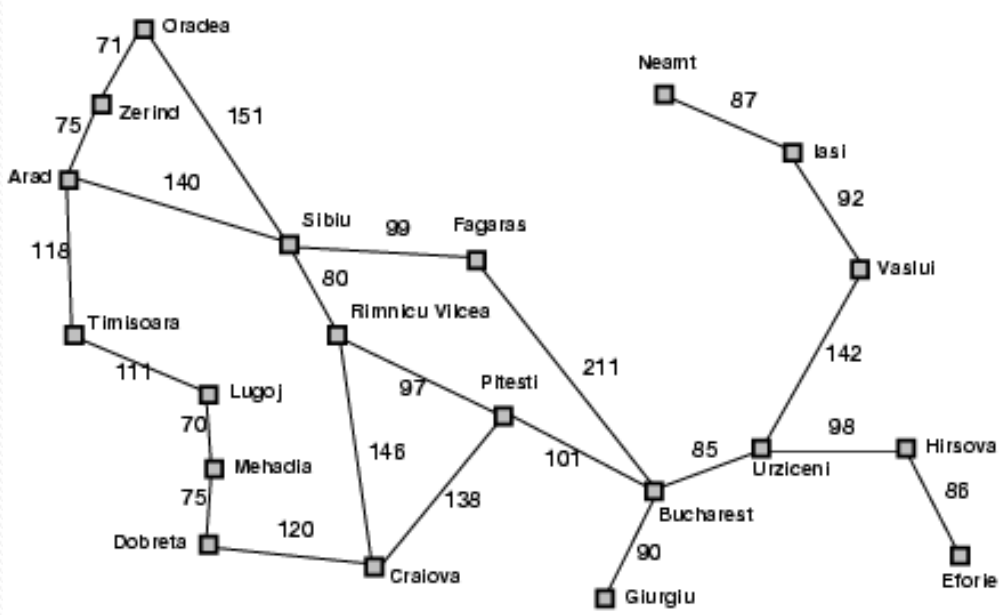
- Idea: avoid expanding paths that are already expensive
- Combines greedy Best FS (h(n)and Uniform cost search(g(n))
- A* Evaluation function $f(n) = g(n) + h(n)$
- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n)$ = estimated total cost of path through n to goal

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



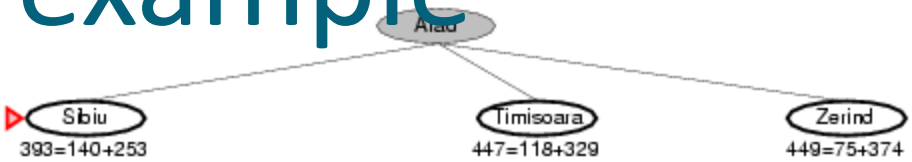
A* search example

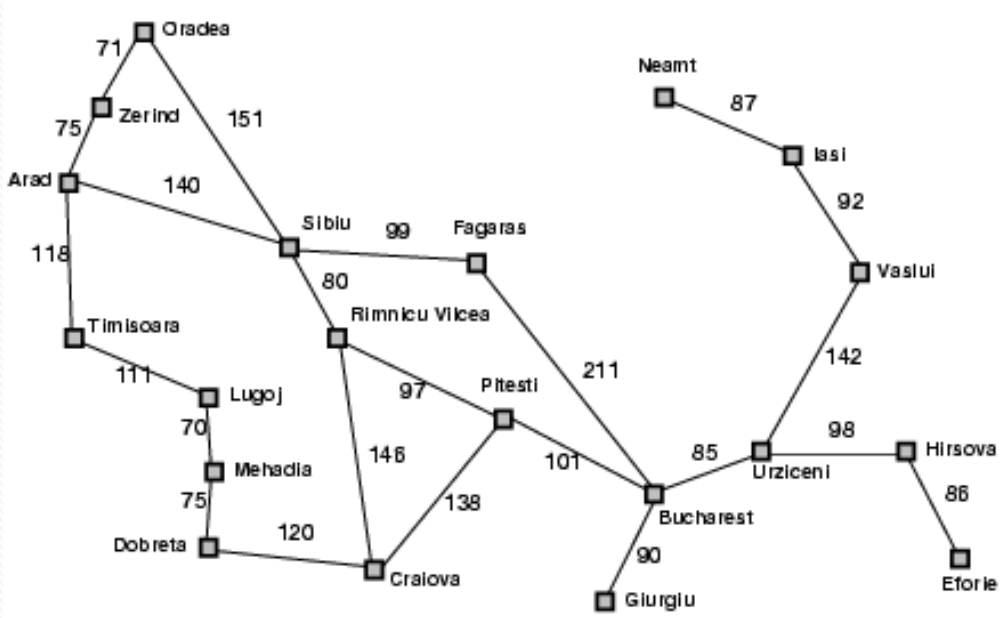




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

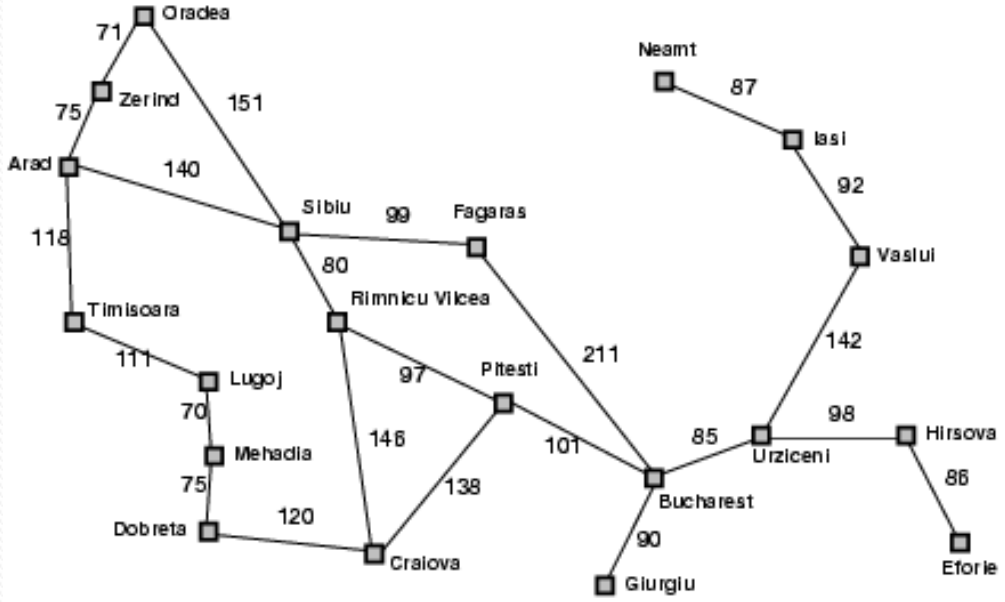
A* search example





Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

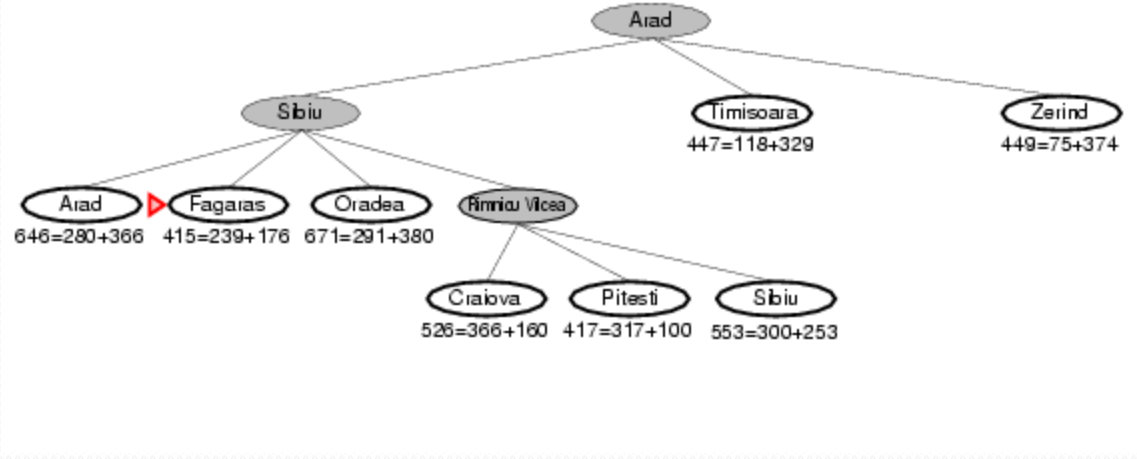
A* search example

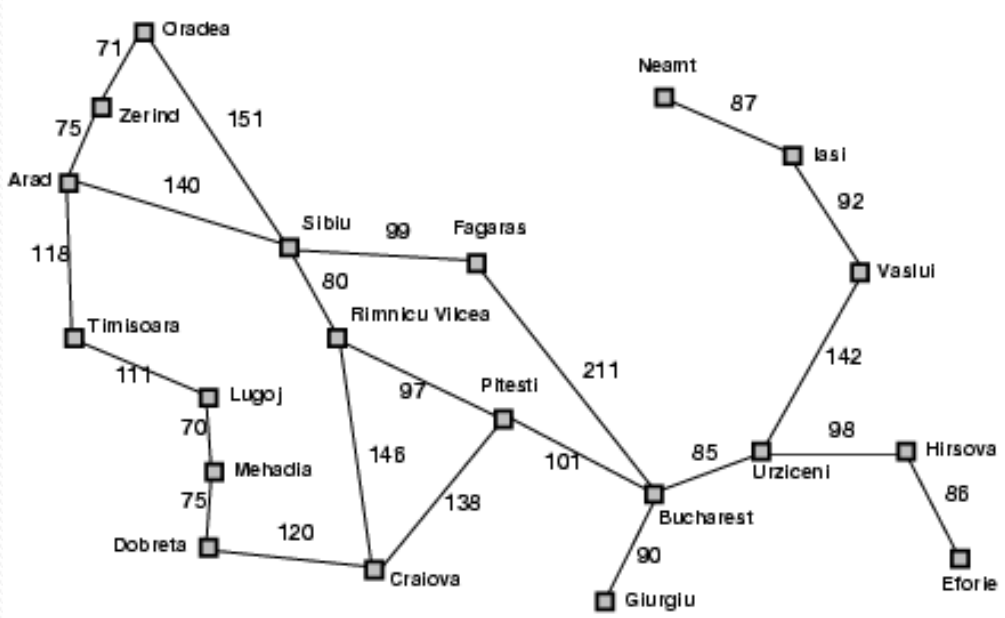


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example

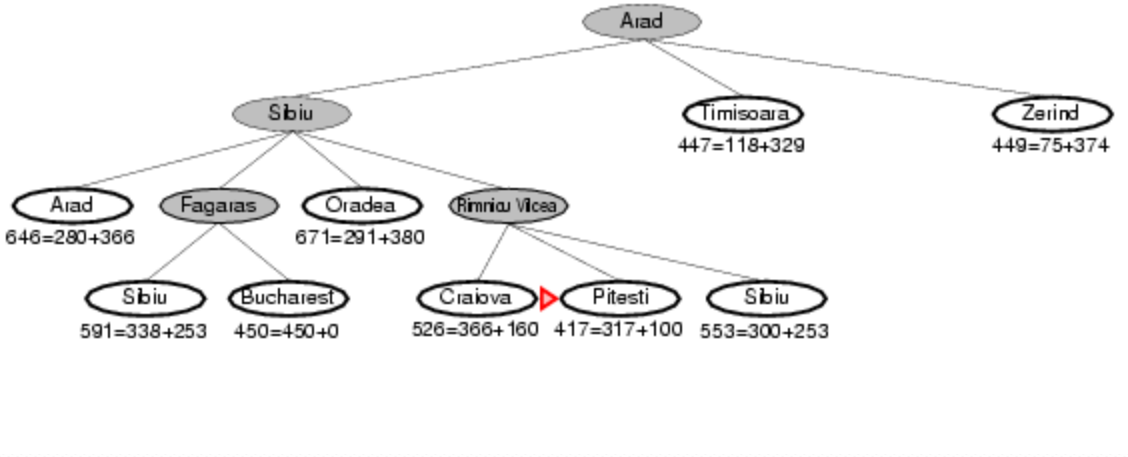


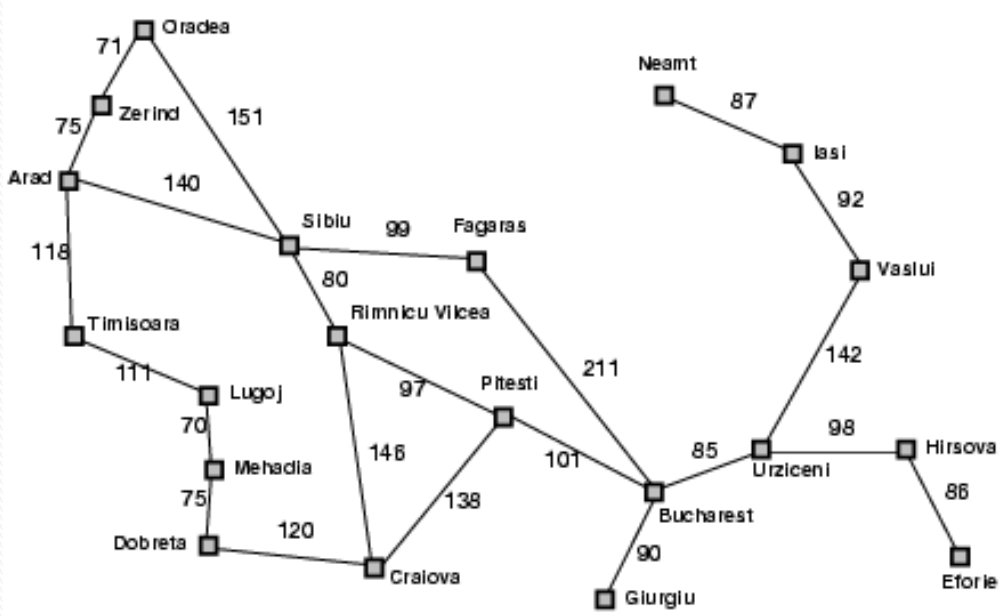


Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

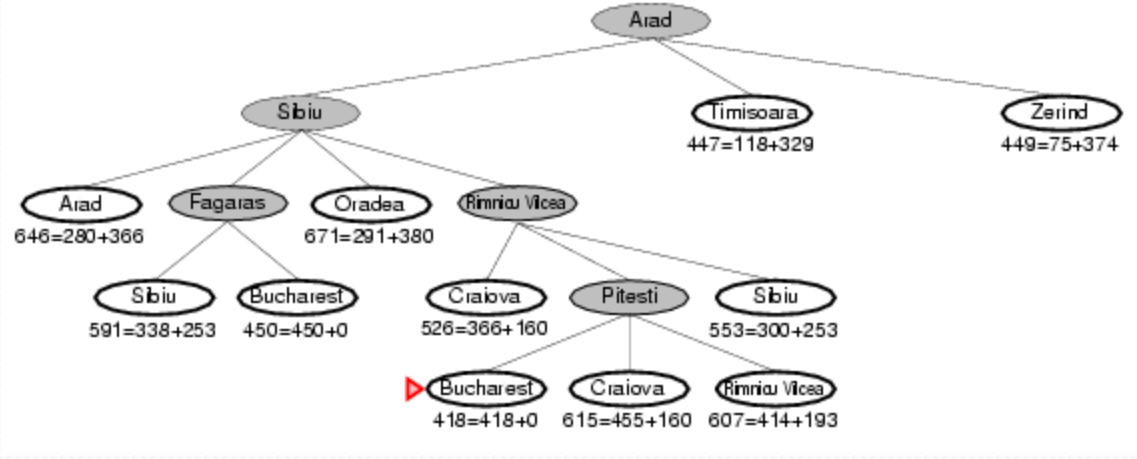
A* search example

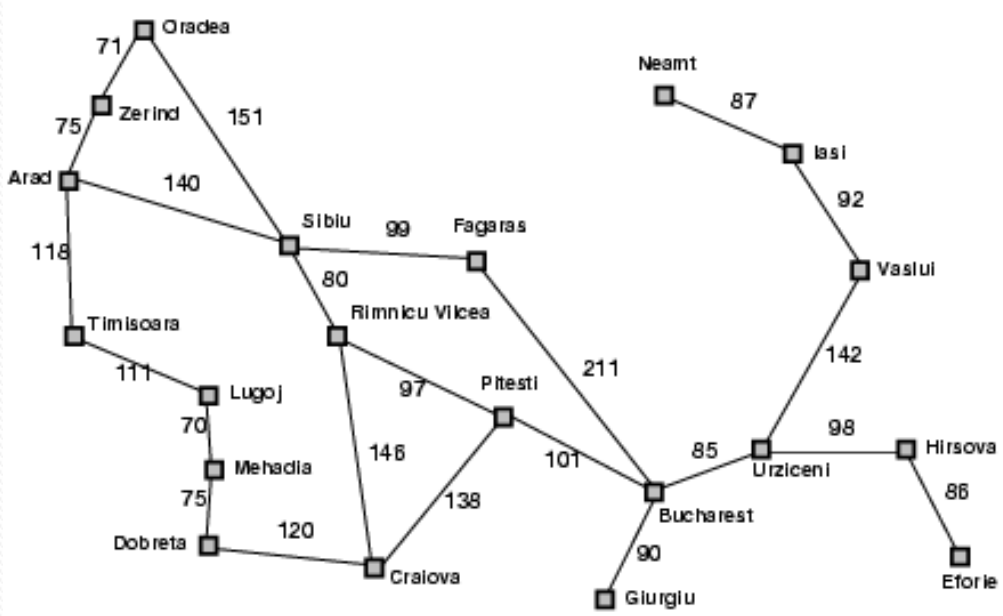




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search example





Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* Algorithm - Pseudocode

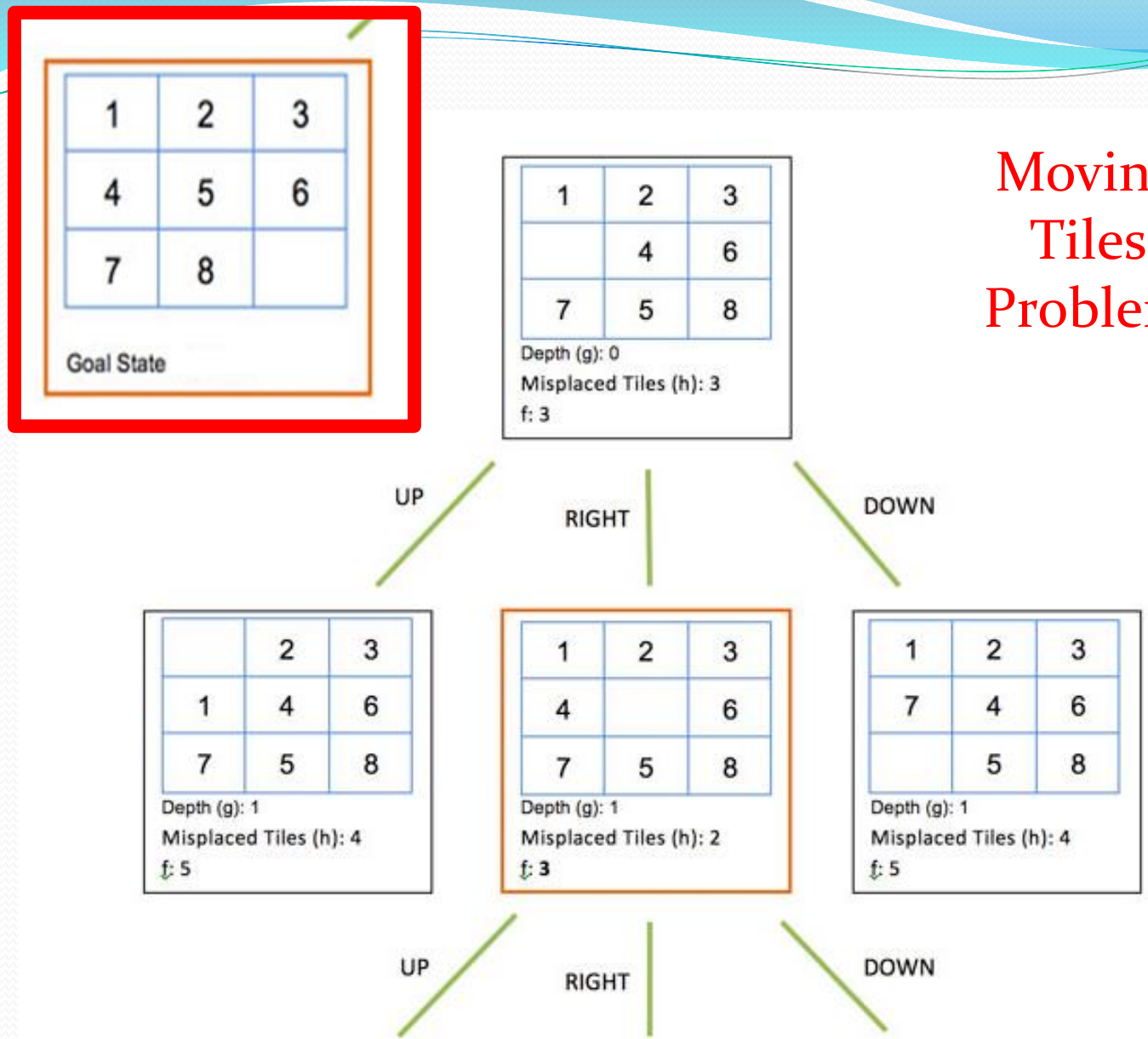
A-Star(Graph graph, Node start, Node goal, HeuristicFunction h)

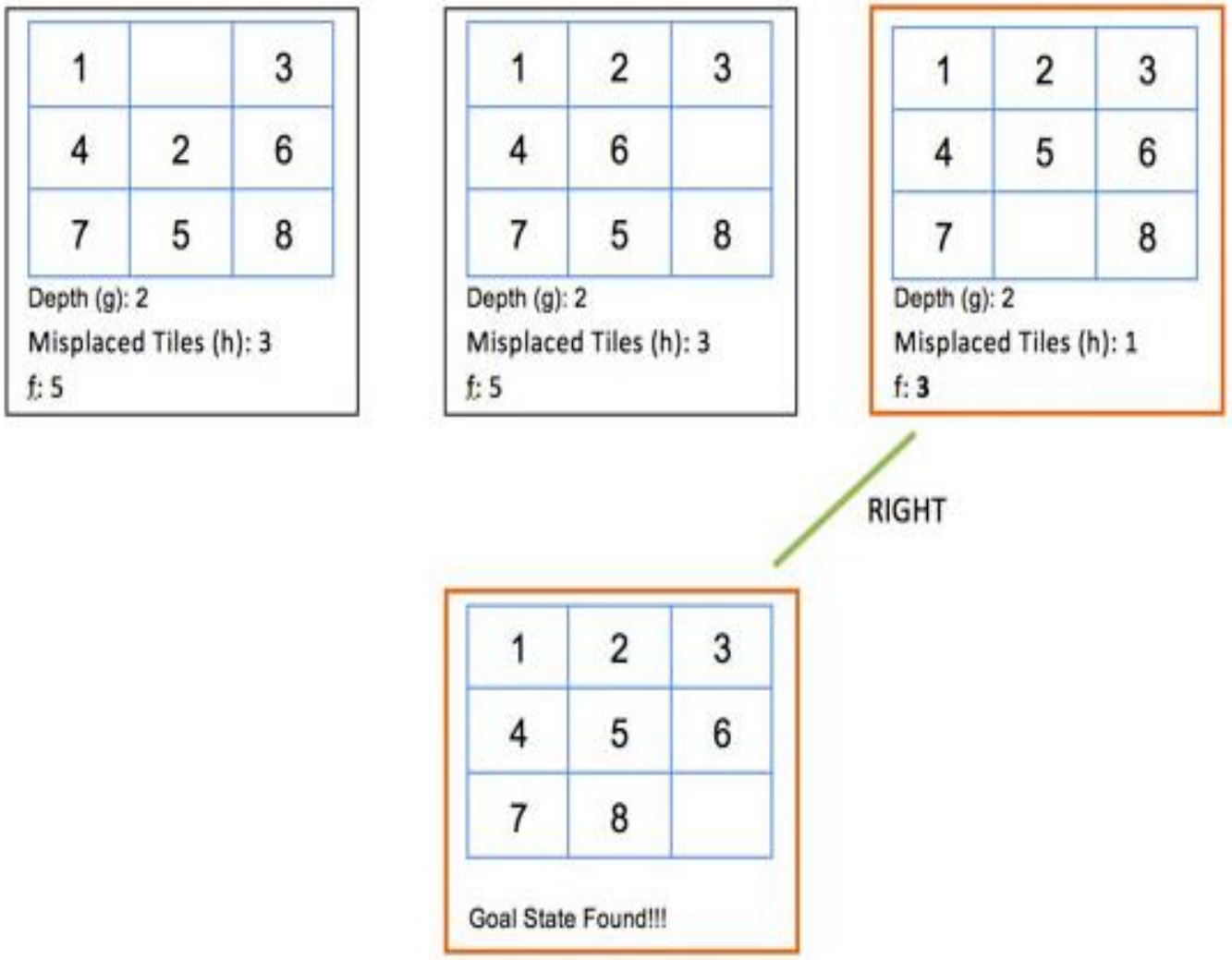
1. O← make_priority_queue(startNode) // open list
2. C←make_hash_table // closed list
3. While O not empty loop
    1. n ← O.remove_front() //O is sorted by $f(n)=g(n)+h(n)$ values
    2. If goal (n) return n
    3. If n is found on C → continue
    4. //otherwise
    5. S ← successors (n)
    6. For each node s in S
        1. Set s.g=n.g+w(n,s)
        2. Set s.parent=n //for path extraction
        3. Set s.h=h(s) //to calculate f
        4. O←s
    7. C←n
4. Return null //no goal found

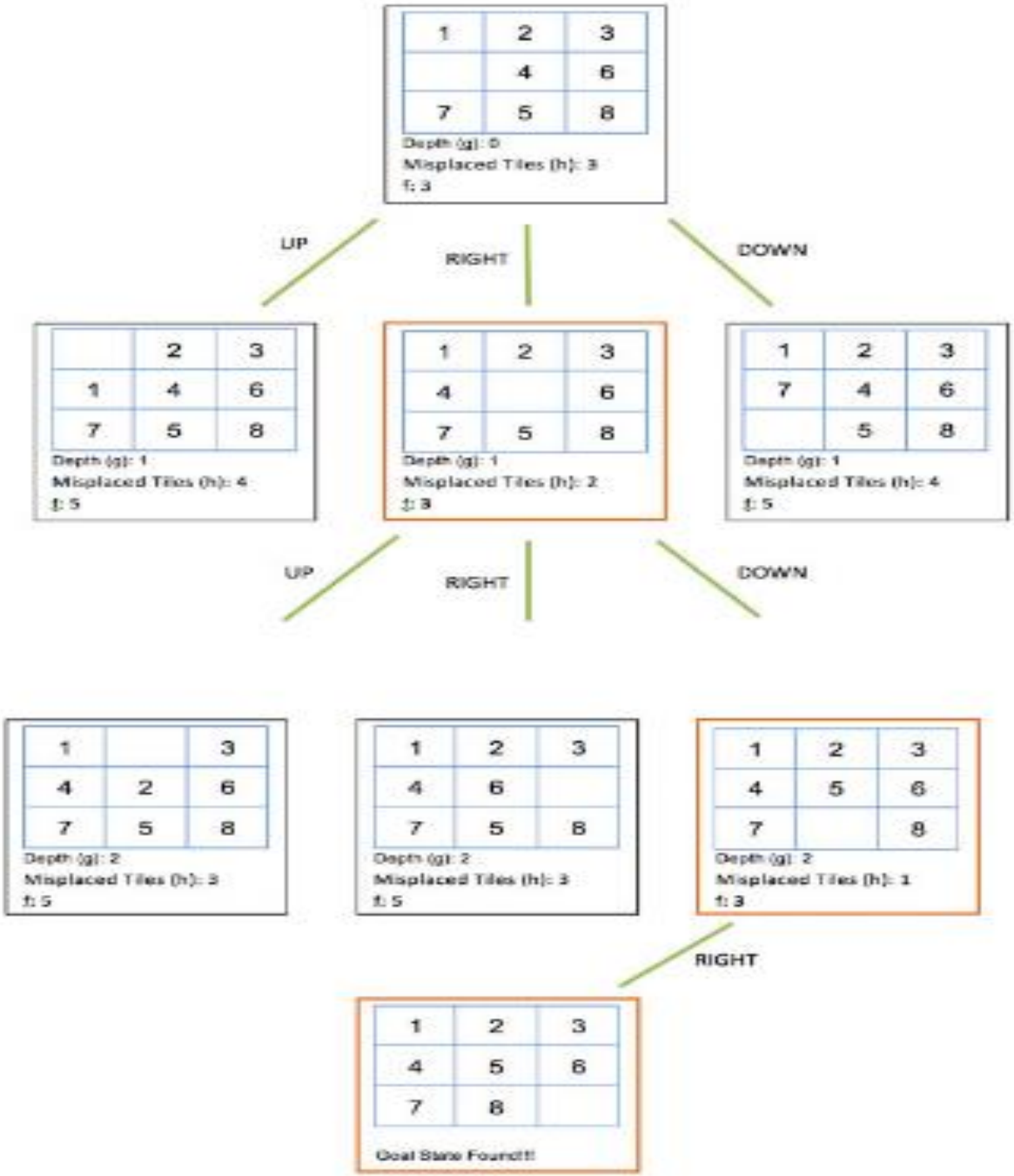
Admissible heuristics

- 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
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If $h(n)$ is admissible, A* using `TREE-SEARCH` is optimal

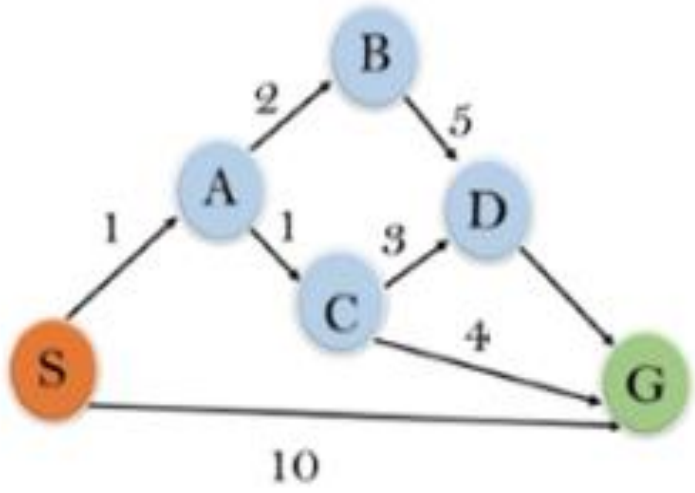
Moving Tiles Problem



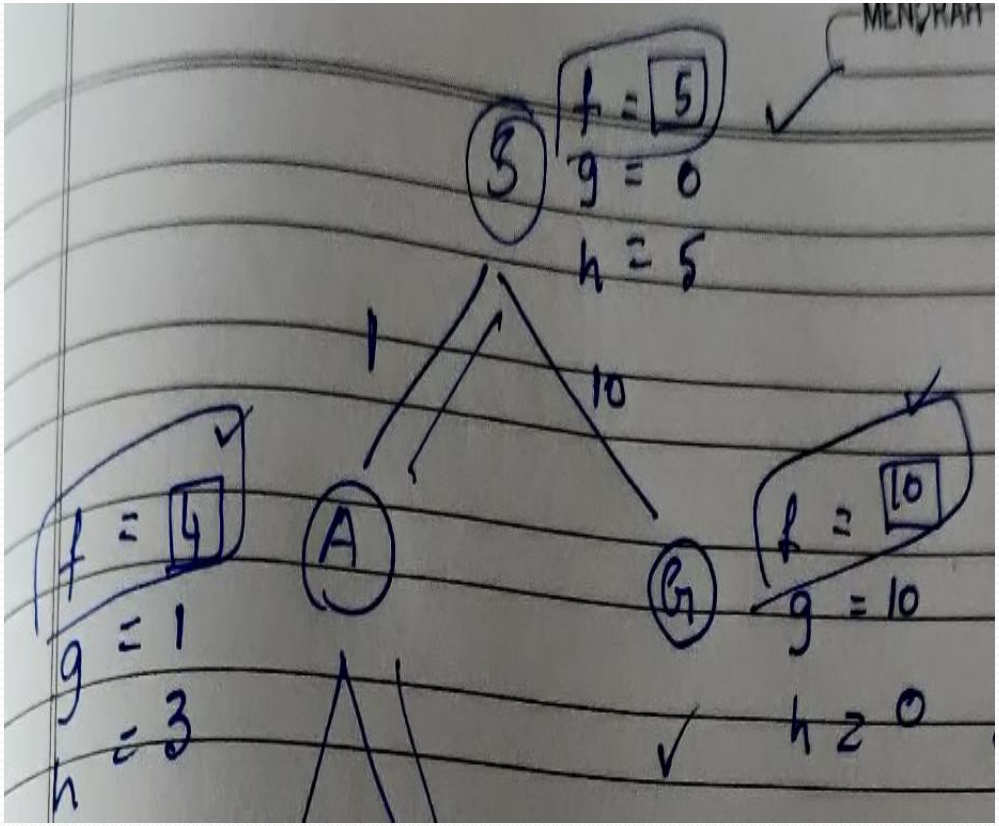

1 3
4 2 6
7 5 8
Depth (g): 2
Misplaced Tiles (h): 3
f: 5
1 2 3
4 6
7 5 8
Depth (g): 2
Misplaced Tiles (h): 3
f: 5
1 2 3
4 5 6
7 8
Depth (g): 2
Misplaced Tiles (h): 1
f: 3
RIGHT
1 2 3
4 5 6
7 8
Goal State Found!!!

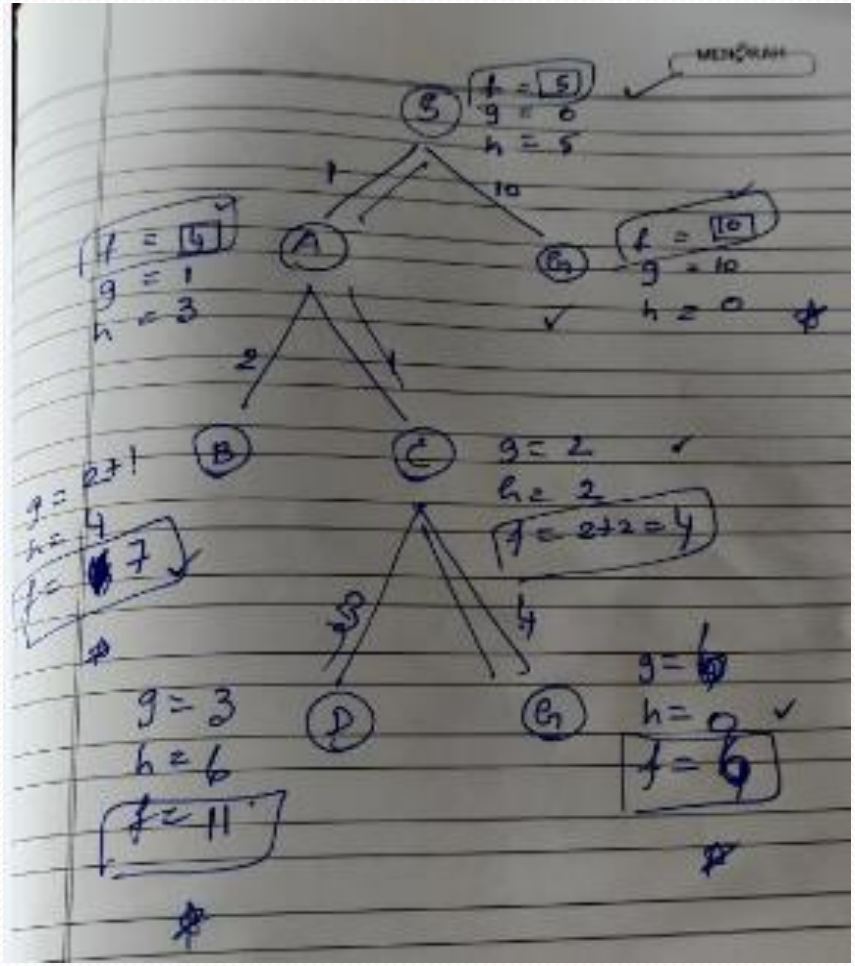

1 2 3
4 6
7 5 8
Depth (g): 0
Misplaced Tiles (h): 3
f: 3
UP
RIGHT
DOWN
2 3
1 4 6
7 5 8
Depth (g): 1
Misplaced Tiles (h): 4
f: 5
1 2 3
4 6
7 5 8
Depth (g): 1
Misplaced Tiles (h): 2
f: 3
1 2 3
7 4 6
5 8
Depth (g): 1
Misplaced Tiles (h): 4
f: 5
UP
RIGHT
DOWN
1 3
4 2 6
7 5 8
Depth (g): 2
Misplaced Tiles (h): 3
f: 5
1 2 3
4 6
7 5 8
Depth (g): 2
Misplaced Tiles (h): 3
f: 5
1 2 3
4 5 6
7 8
Depth (g): 2
Misplaced Tiles (h): 1
f: 3
RIGHT
1 2 3
4 5 6
7 8
Goal State Found!!

Home work problem -4



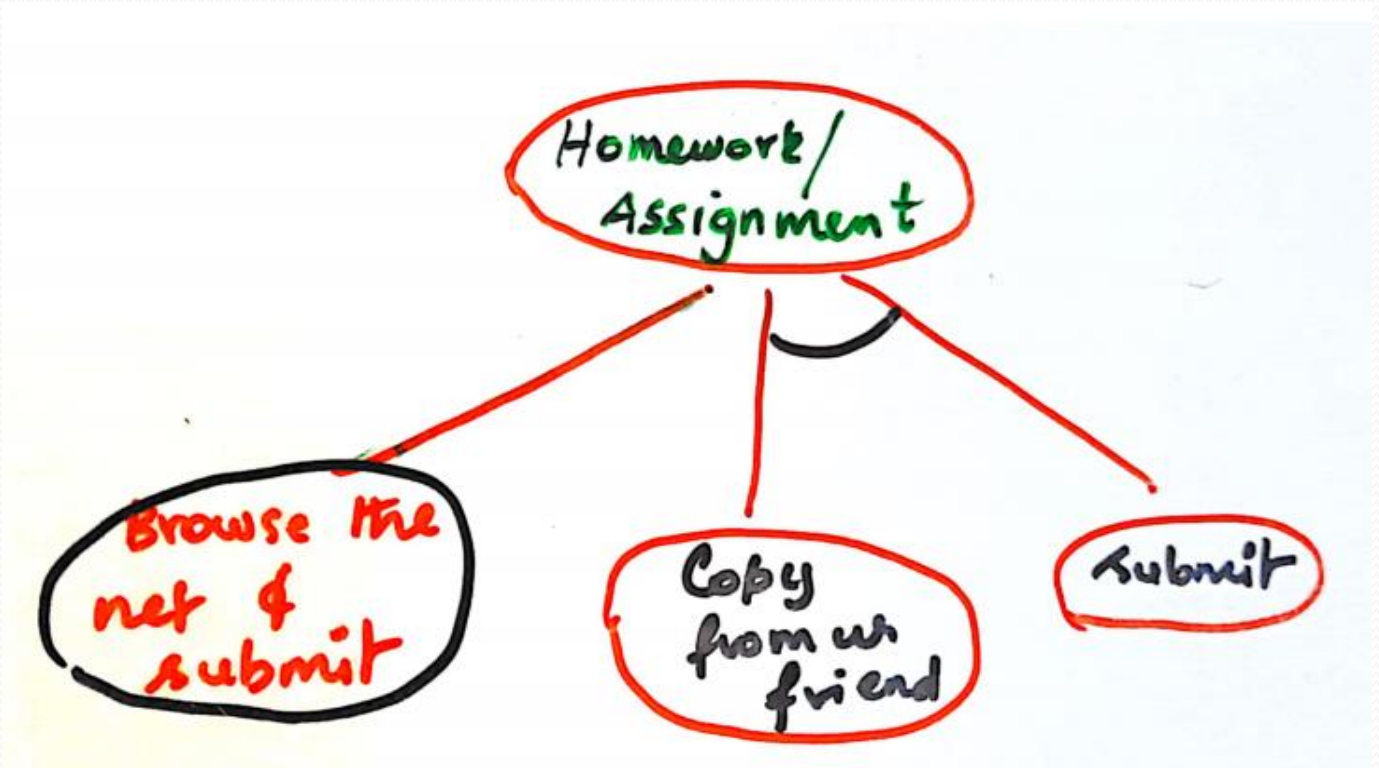
State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0


MENGHAH
f = 5
S
g = 0
h = 5
1
10
f = 4
g = 1
h = 3
A
G1
f = 10
g = 10
h = 0


S
f = 5
g = 0
h = 5
1
10
A
f = 4
g = 1
h = 3
G2
f = 10
g = 10
h = 0
2
1
B
C
g = 2
h = 2
f = 2+2 = 4
g = 2+1
h = 4
f = 7
3
4
D
G1
g = 3
h = 6
f = 11
g = 6
h = 0
f = 6

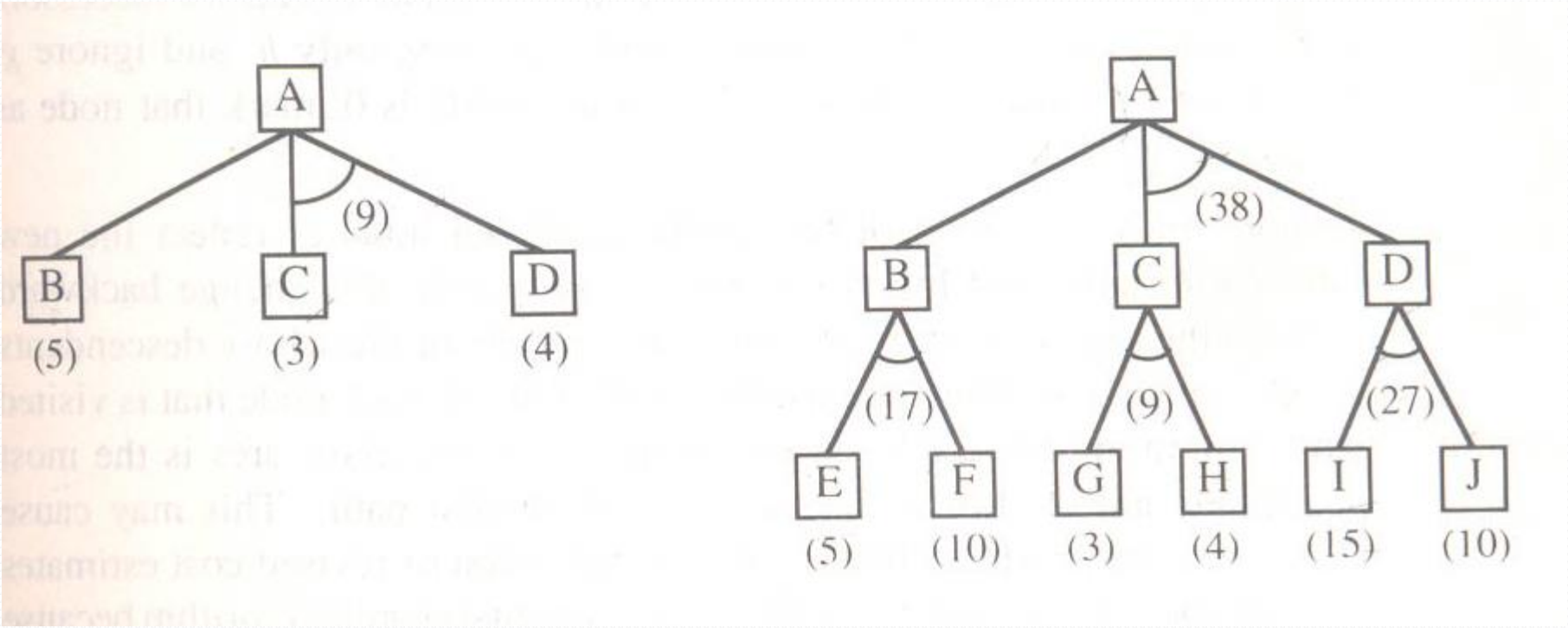
PROBLEM REDUCTION

- Decomposing into a set of smaller problems and then solving each problem
- Example



AND-OR GRAPH/TREE

- One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution.



AO* procedure

- Initialise the graph to start node
- Traverse the graph following the current path accumulating nodes that have not yet been expanded or solved
- Pick any of these nodes and expand it and if it has no successors call this value *FUTILITY* otherwise calculate only *f'* for each of the successors.
- If *f'* is 0 then mark the node as *SOLVED*
- Change the value of *f'* for the newly created node to reflect its successors by back propagation.
- Wherever possible use the most promising routes and if a node is marked as *SOLVED* then mark the parent node as *SOLVED*.
- If starting node is *SOLVED* or value greater than *FUTILITY*, stop, else repeat from 2.

The AND/OR graph search problem

- **Problem definition:**
  - **Given: [G, s, T] where**
    - G: implicitly specified AND/OR graph
    - S: start node of the AND/OR graph
    - T: set of terminal nodes
    - h(n) heuristic function estimating the cost of solving the sub-problem at n
  - **To find:**
    - A minimum cost solution tree

Algorithm AO*

1. Initialize: Set G* = {s}, f(s) = h(s)
   If s ∈ T, label s as SOLVED
2. Terminate: If s is SOLVED, then Terminate
3. Select: Select a non-terminal leaf node n from the marked sub-tree
4. Expand: Make explicit the successors of n
   For each new successor, m:
   Set f(m) = h(m)
   If m is terminal, label m SOLVED
5. Cost Revision: Call cost-revise(n)
6. Loop: Go To Step 2.

Cost Revision in AO*: cost-revise(n)

1. Create Z = {n}
2. If Z = { } return
3. Select a node m from Z such that m has no descendants in Z
4. If m is an AND node with successors

   $r_1, r_2, \ldots r_k$:

   Set $f(m) = \sum [f(r_i) + c(m, r_i)]$

   Mark the edge to each successor of m

   If each successor is labeled SOLVED, then label m as SOLVED

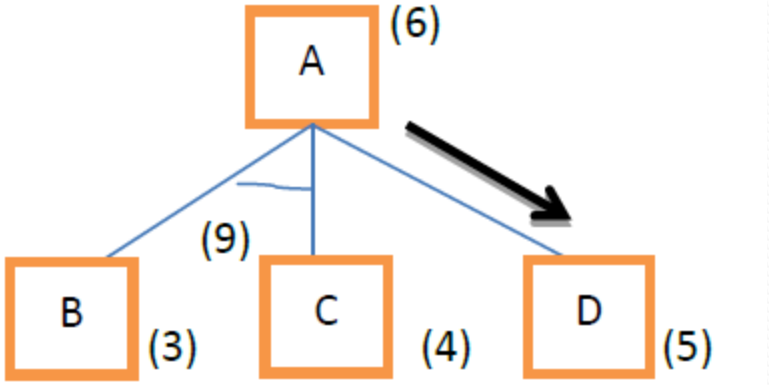
Cost Revision in AO*: cost-revise(n)

5. If m is an OR node with successors $r_1, r_2, \ldots r_k$:
   Set $f(m) = \min \{ f(r_i) + c(m, r_i) \}$
   Mark the edge to the best successor of m
   If the marked successor is labeled SOLVED, label m as SOLVED
6. If the cost or label of m has changed, then insert those parents of m into Z for which m is a marked successor
7. Go to Step 2.

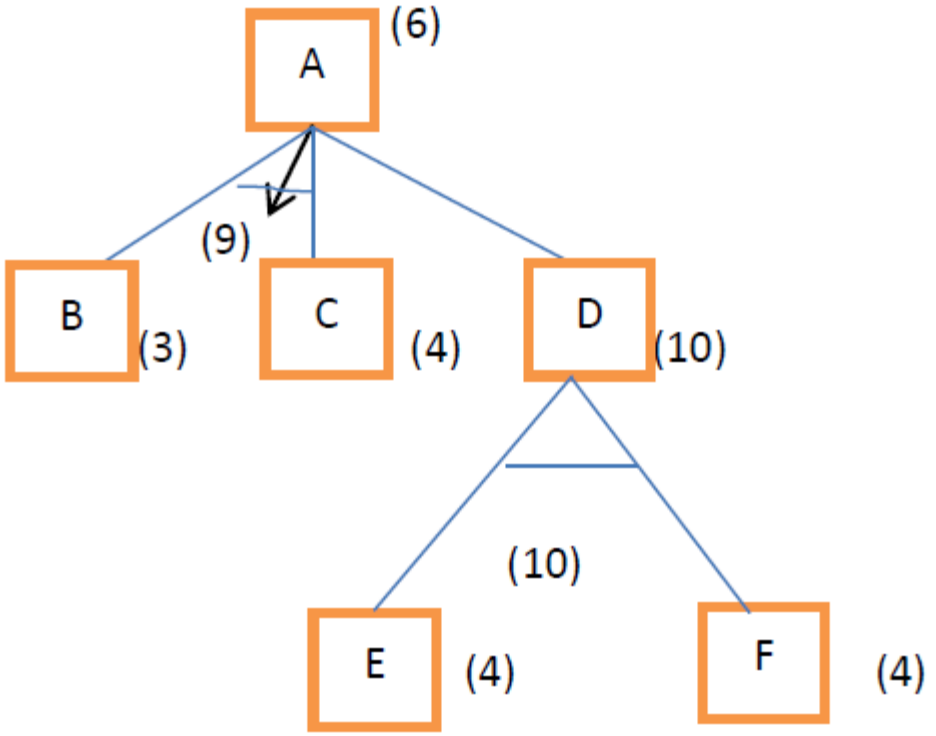
Example Problem -AO*

Example Problem -AO* (Contd)

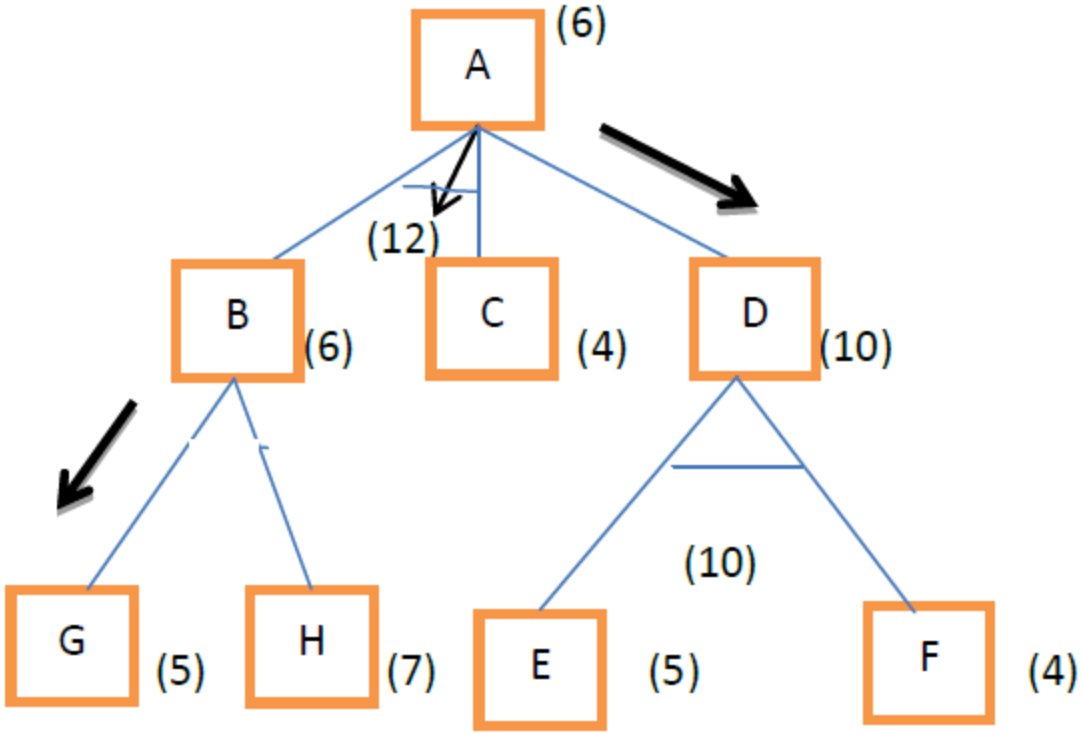
- A is the only node, it is at the end of the current best path. It is expanded, yielding nodes B, C, D. The arc to D is labeled as the most promising one emerging from A, since it costs 6 compared to B and C, Which costs 9
- Node D is chosen for expansion.



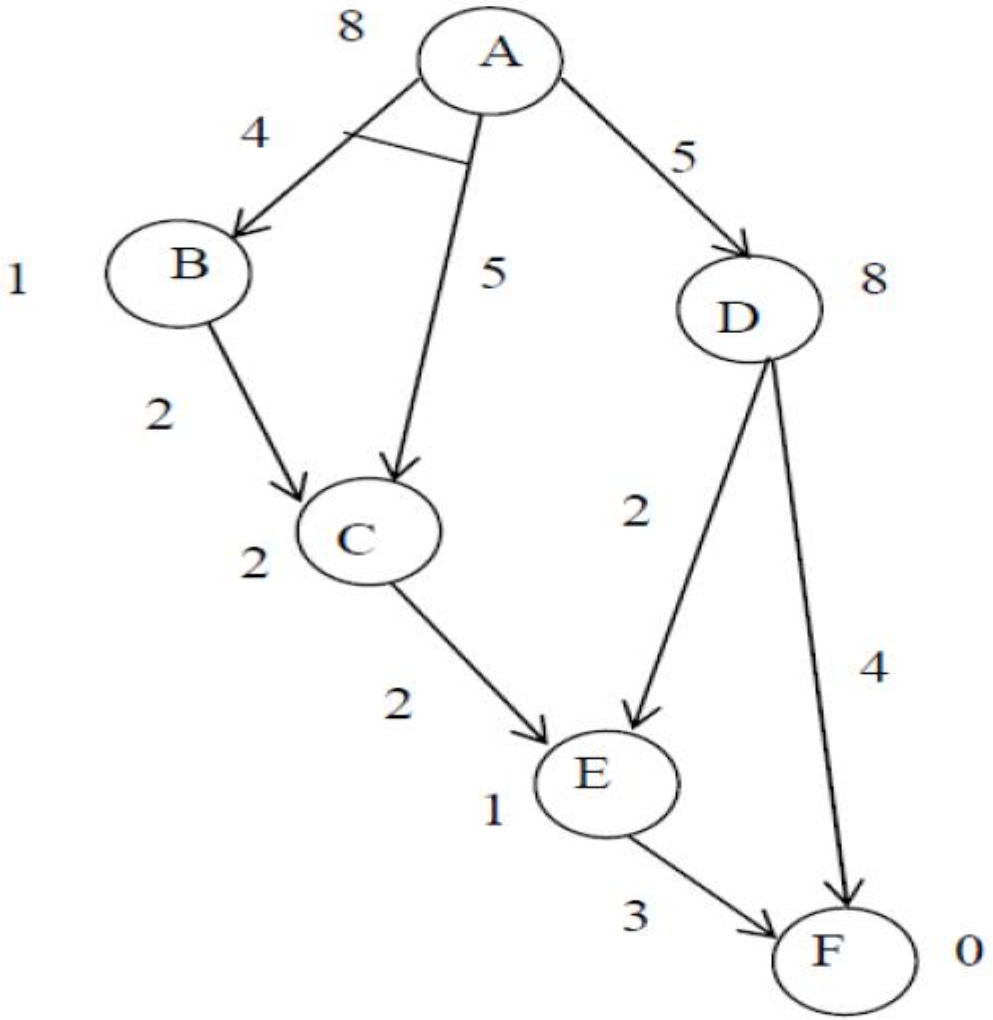
- D produces one new arc, the AND arc to E and F, with a combined cost estimate of 10.so update the f' value of D to 10.
- Going back one more level, makes the AND arc B-C better than the arc to D, so it is labeled as the current best path.

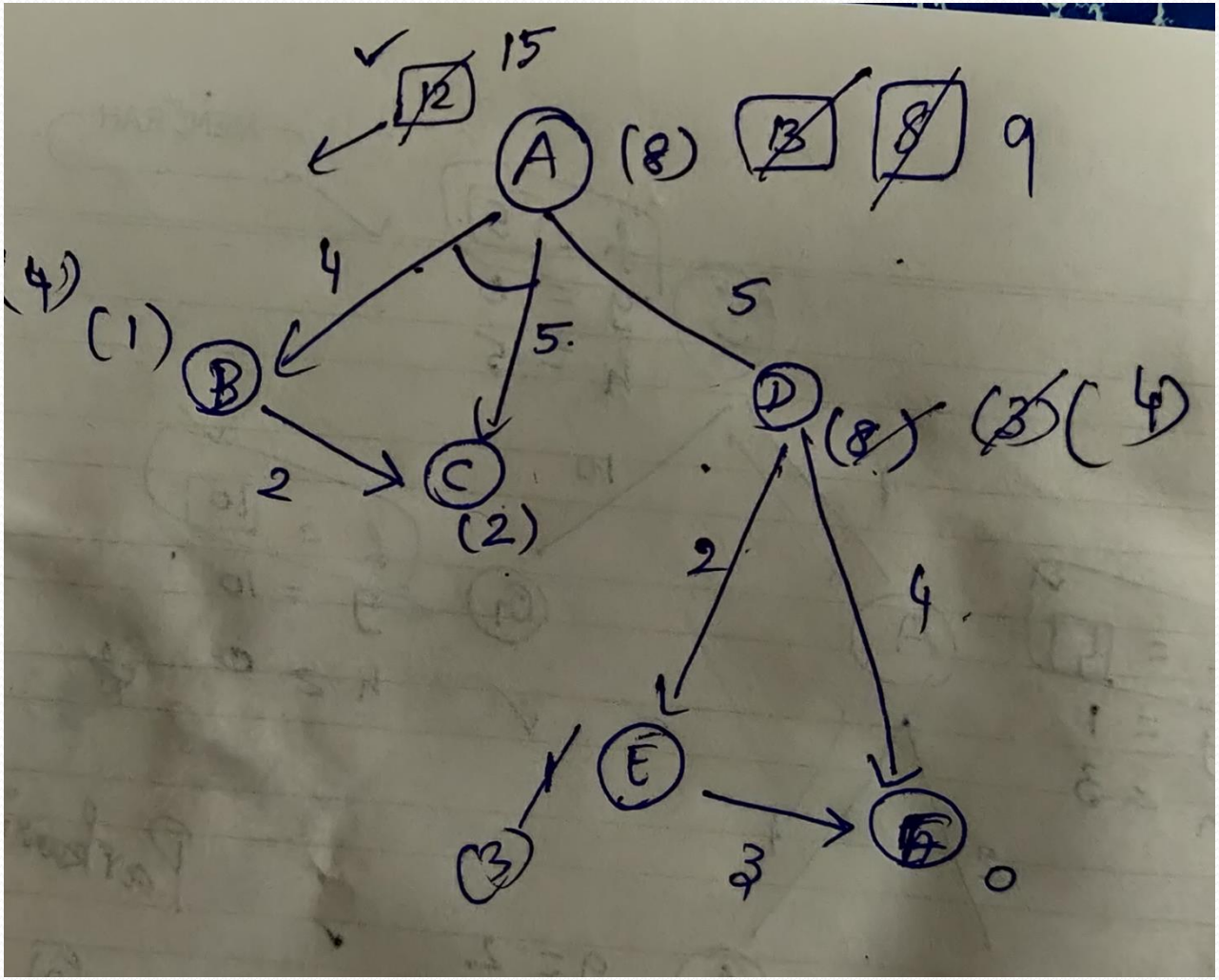


- B generates G and H. Propagating their f' values backward, update f' of B to 6. This requires updating the cost of the AND arc B-C to 12(6+4+2).
- After this, the arc to D is again the better path from A, so record that as the current best path and either node E or node F will chosen for expansion next



Home work problem -5




15
12
A
(8)
13
8
9
4
5
5
(4)
(1)
B
D
(8)
(3)
(4)
2
C
(2)
2
4
E
1
3
3
F
0

Adversarial Search

Mathematical Game Theory:

Branch of economics that views any multi-agent environment as a game, provided that the impact of each agent on the others is "significant", regardless of whether the agents are cooperative or competitive.

In AI, Games have special format

- Deterministic
- Turn taking
- 2-player
- Zero-sum game of perfect information (fully observable) "my win is your loss" and vice versa; utility of final states opposite for each player. My +10 is your -10.

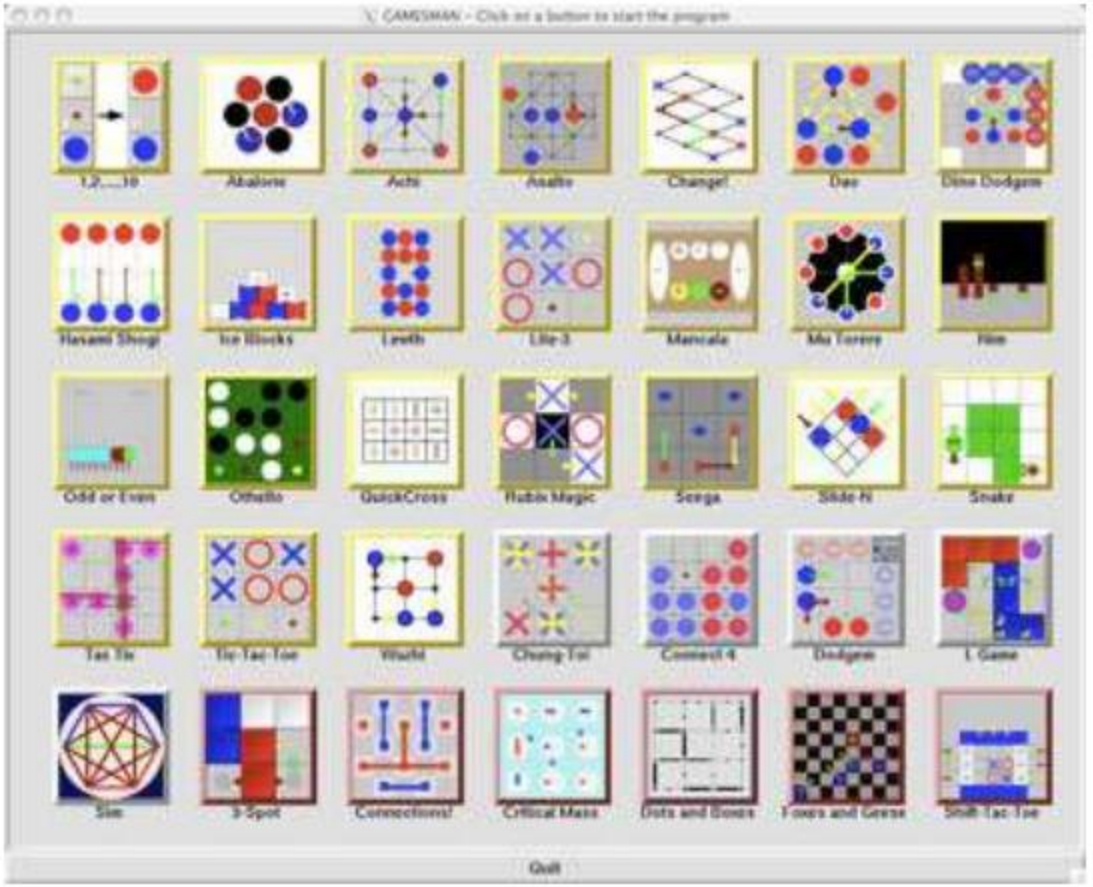
- It is this opposition between the agents' utility functions that makes the situation adversarial.

Adversarial Search

- Multi-agent environment:
  - any given agent needs to consider the actions of other agents and how they affect its own welfare
  - introduce possible contingencies into the agent's problem-solving process
  - cooperative vs. competitive
- Adversarial search problems: agents have conflicting goals -- games

Games Vs Search Problems

- "Unpredictable" opponent
  - specifying a move for every possible opponent reply
- Time limits
  - unlikely to find goal, must approximate



Games Vs Search Problems

- **Search – no adversary**
  - Solution is a path from start to goal, or a series of actions from start to goal
  - Heuristics and search techniques can find *optimal* solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Actions have cost
  - Examples: path planning, scheduling activities

- **Games – adversary**
  - Solution is strategy
    - strategy specifies move for every possible opponent reply.
  - Time limits force an *approximate* solution
  - Evaluation function: evaluate "goodness" of game position
  - Board configurations have utility
  - Examples: chess, checkers, Othello, backgammon

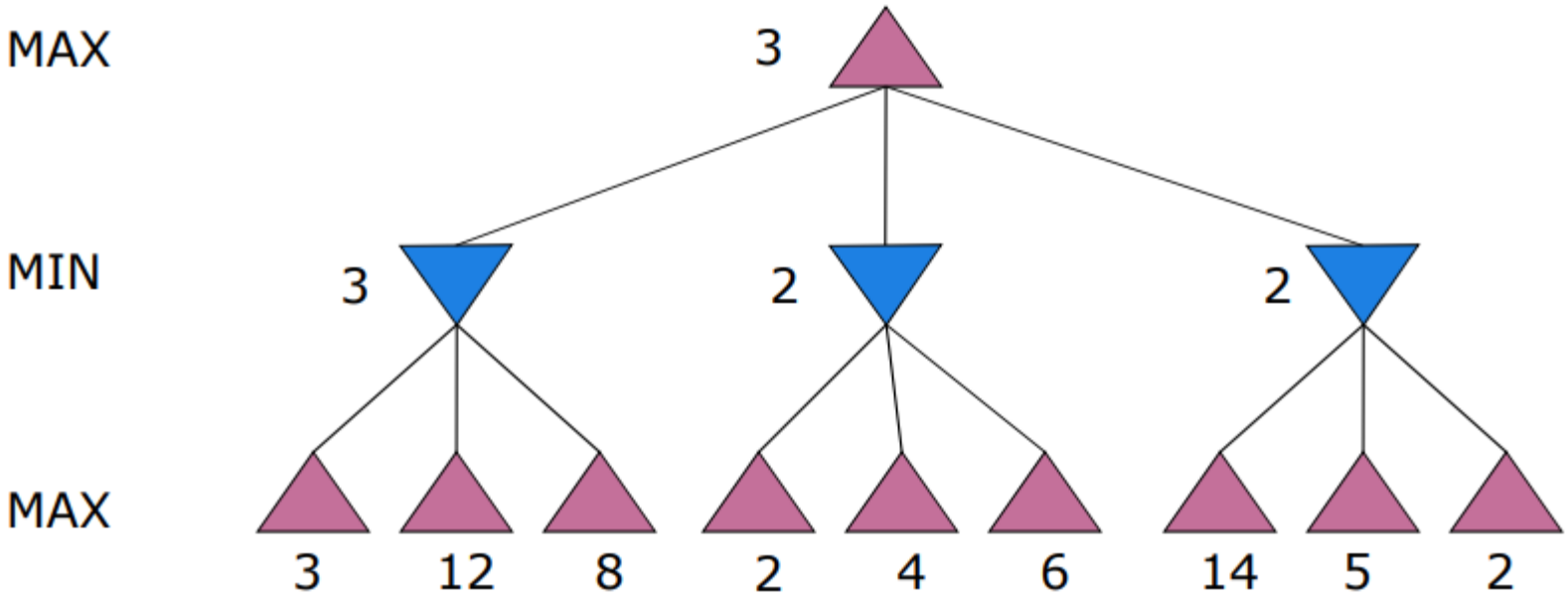
Game Problem Formulation

- A game with 2 players (MAX and MIN, MAX moves first, turn-taking) can be defined as a search problem with:
  - initial state: board position
  - player: player to move
  - successor function: a list of legal (move, state) pairs
  - goal test: whether the game is over – terminal states
  - utility function: gives a numeric value for the terminal states (win, loss, draw)

- Game tree = initial state + legal moves

Optimal Strategy

- MAX must find a contingent strategy, specifying MAX's move in:
  - the initial state
  - the states resulting from every possible response by MIN
- E.g., 2-ply game (the tree is one move deep, consisting of two half-moves, each of which is called a ply):



Minimax Value

- Perfect play for deterministic game, assume both players play optimally
- Idea: choose move to position with highest minimax value = best achievable payoff against best play

$$MINIMAX-VALUE(n) =$$

$$Utility(n) \quad \text{if n is a terminal state}$$

$$\max_{s \in Successors(n)} MINIMAX(s) \quad \text{if n is a MAX node}$$

$$\min_{s \in Successors(n)} MINIMAX(s) \quad \text{if n is a MIN node}$$

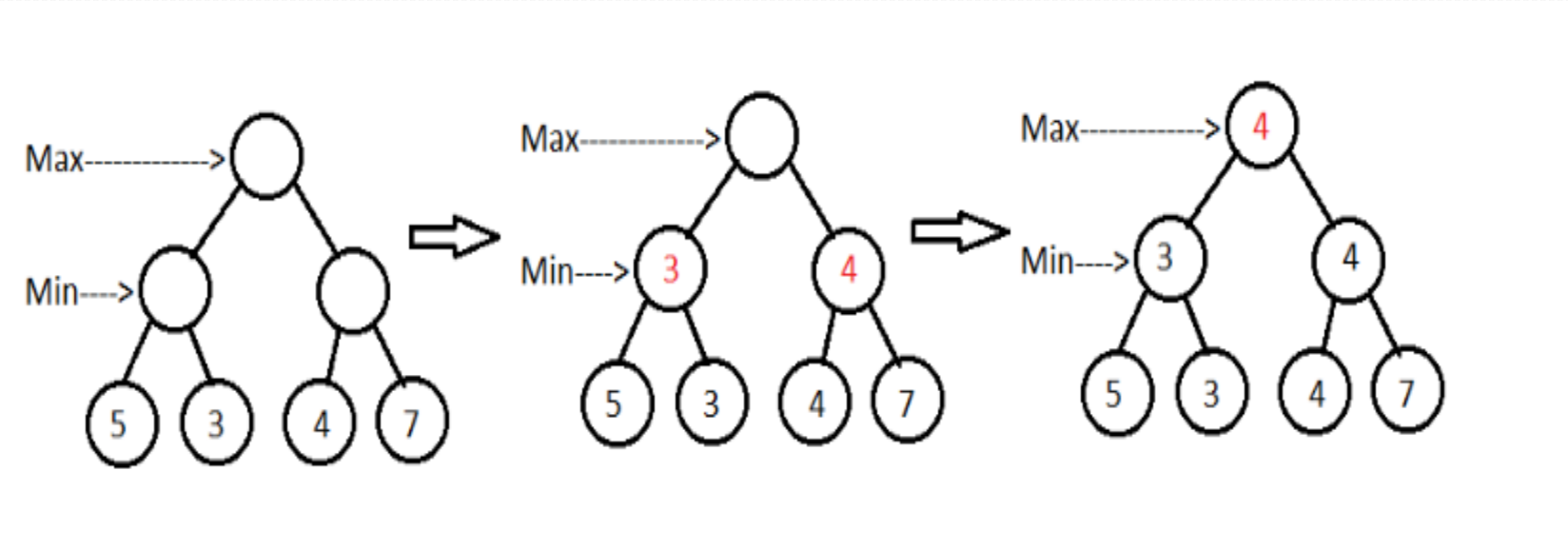
Minimax Search

- It is a depth first, depth limited search procedure
- Start at the current position and use the plausible move generator to generate the set of successor positions
- Apply the static evaluation function to those functions and choose the best one
- Push the value to the starting position to represent the new evaluation

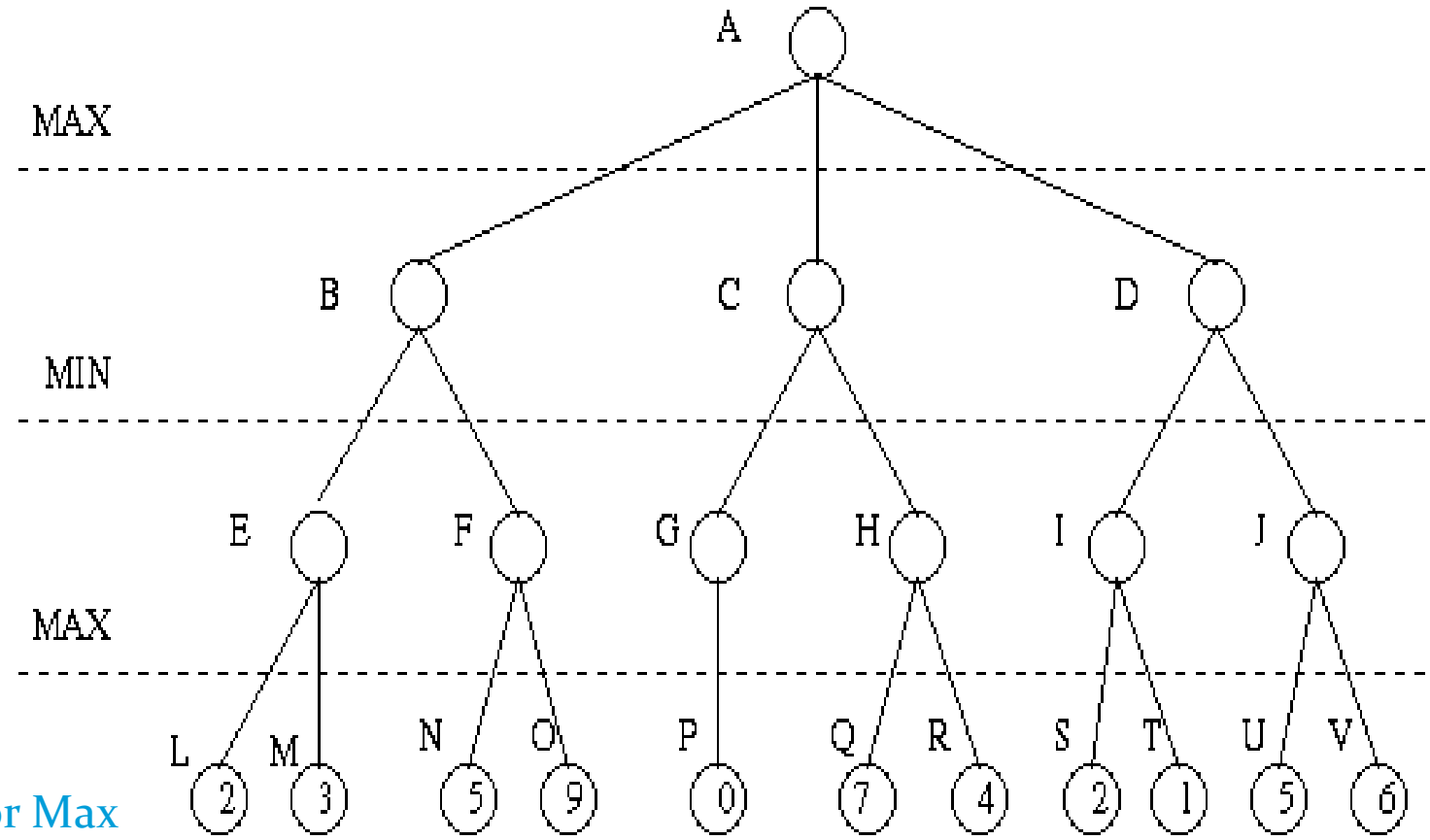
Minimax Search (Contd..)

- After one move , the situation appear to be very good but after 2nd move, it may appear worse, so look ahead to see what will happen at the next move, which will be made by the opponent
- The alternations of maximising and minimizing at alternate ply corresponds to the opposing strategies of the 2 players and hence the name minimax
- It follows a recursive procedure

Example

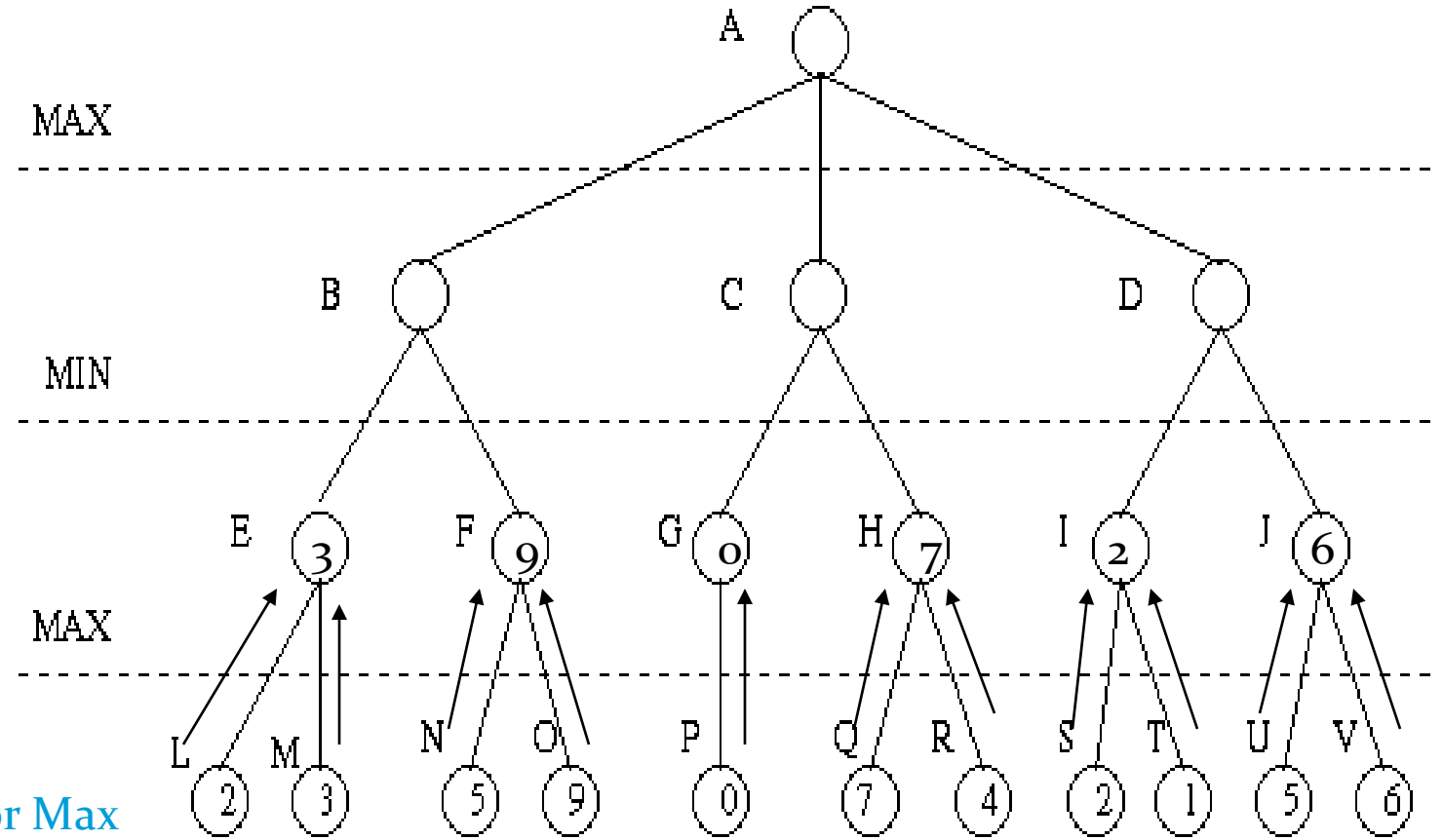


Minimax Algorithm



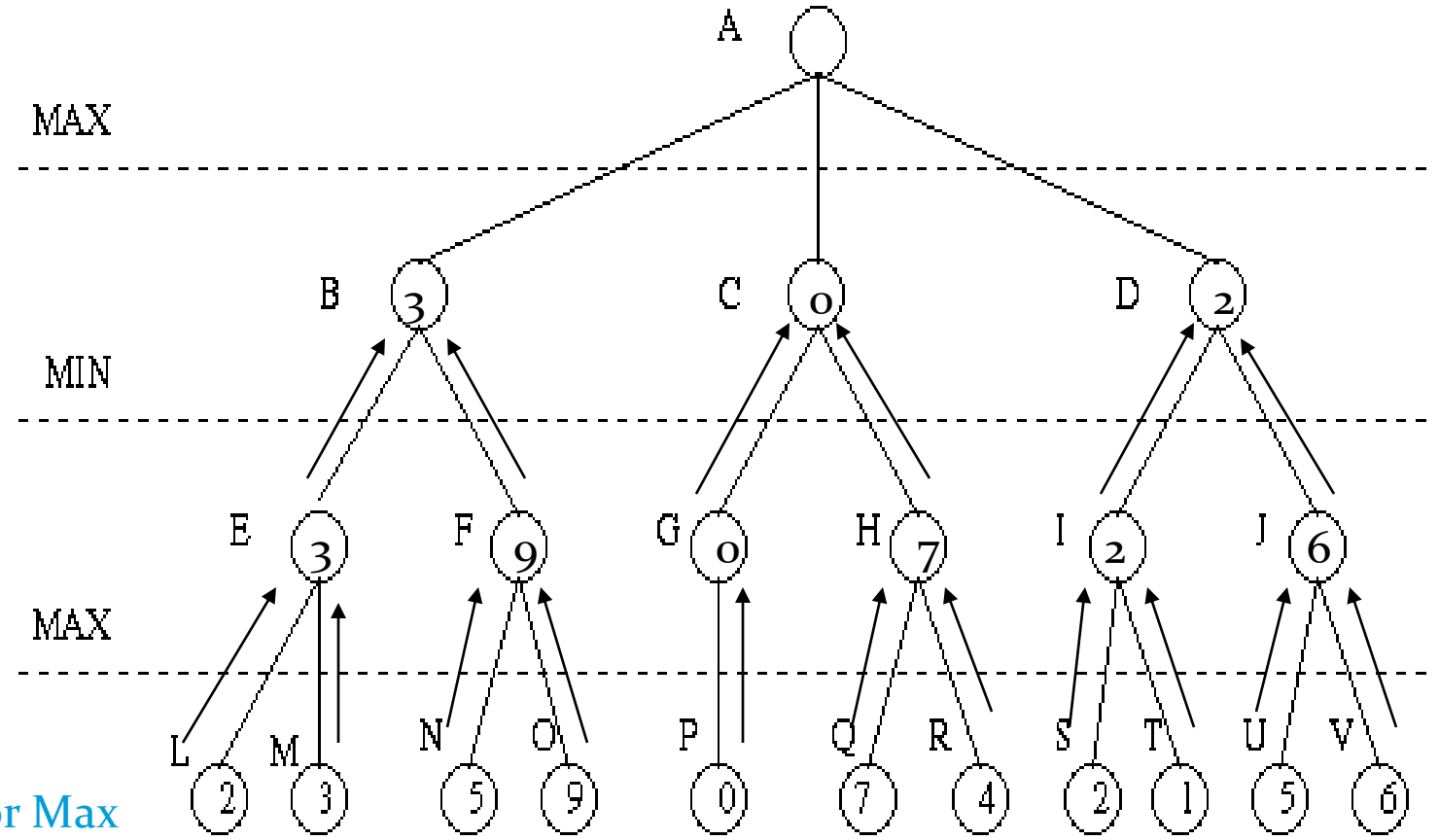
Payoff for Max

Minimax Algorithm (cont'd)



Payoff for Max

Minimax Algorithm (cont'd)



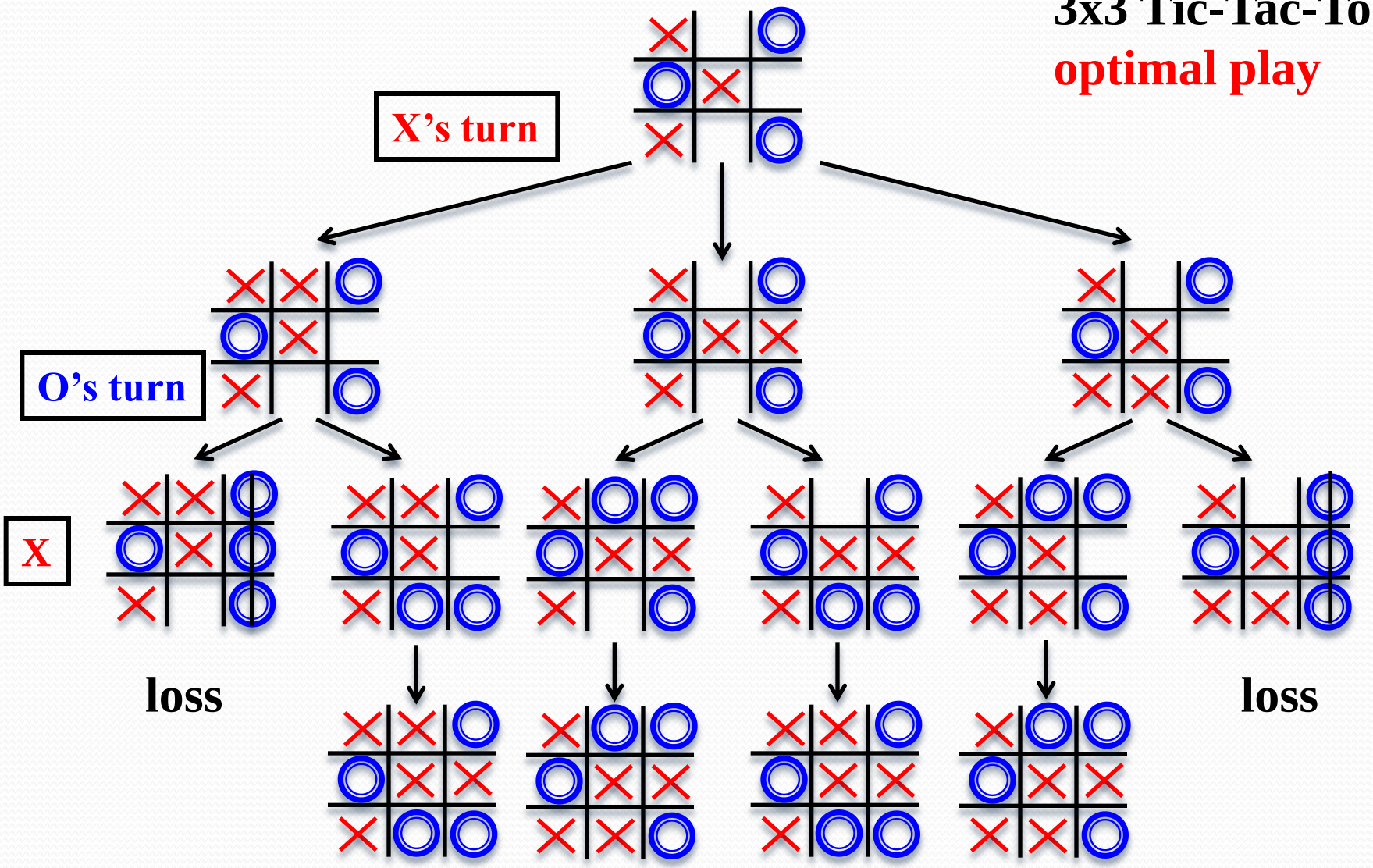
Payoff for Max

Tic-tac-toe (or Noughts and crosses, Xs and Os)

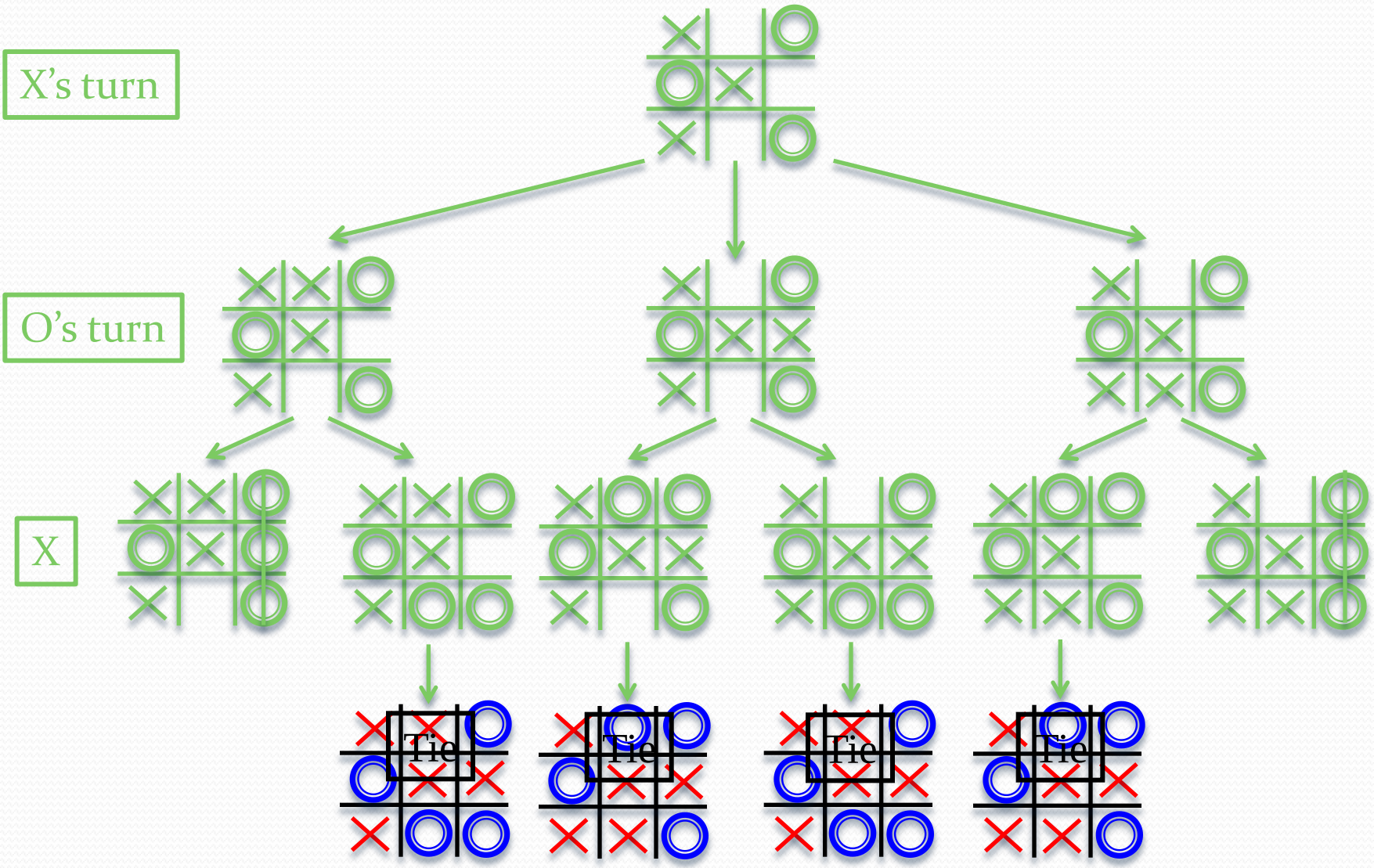
Key Idea: Look Ahead

We start 3 moves per player in:

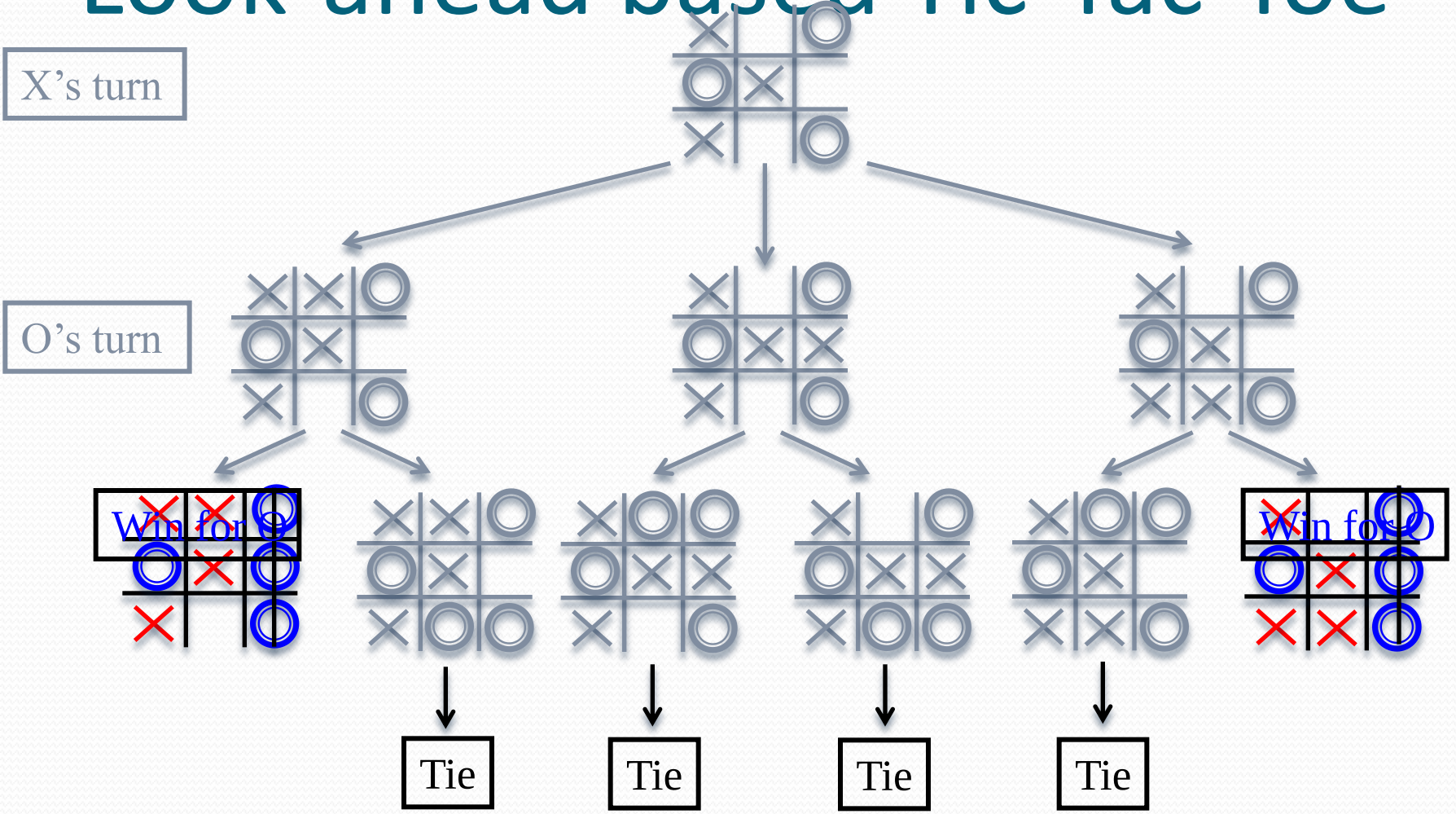
3x3 Tic-Tac-Toe
optimal play



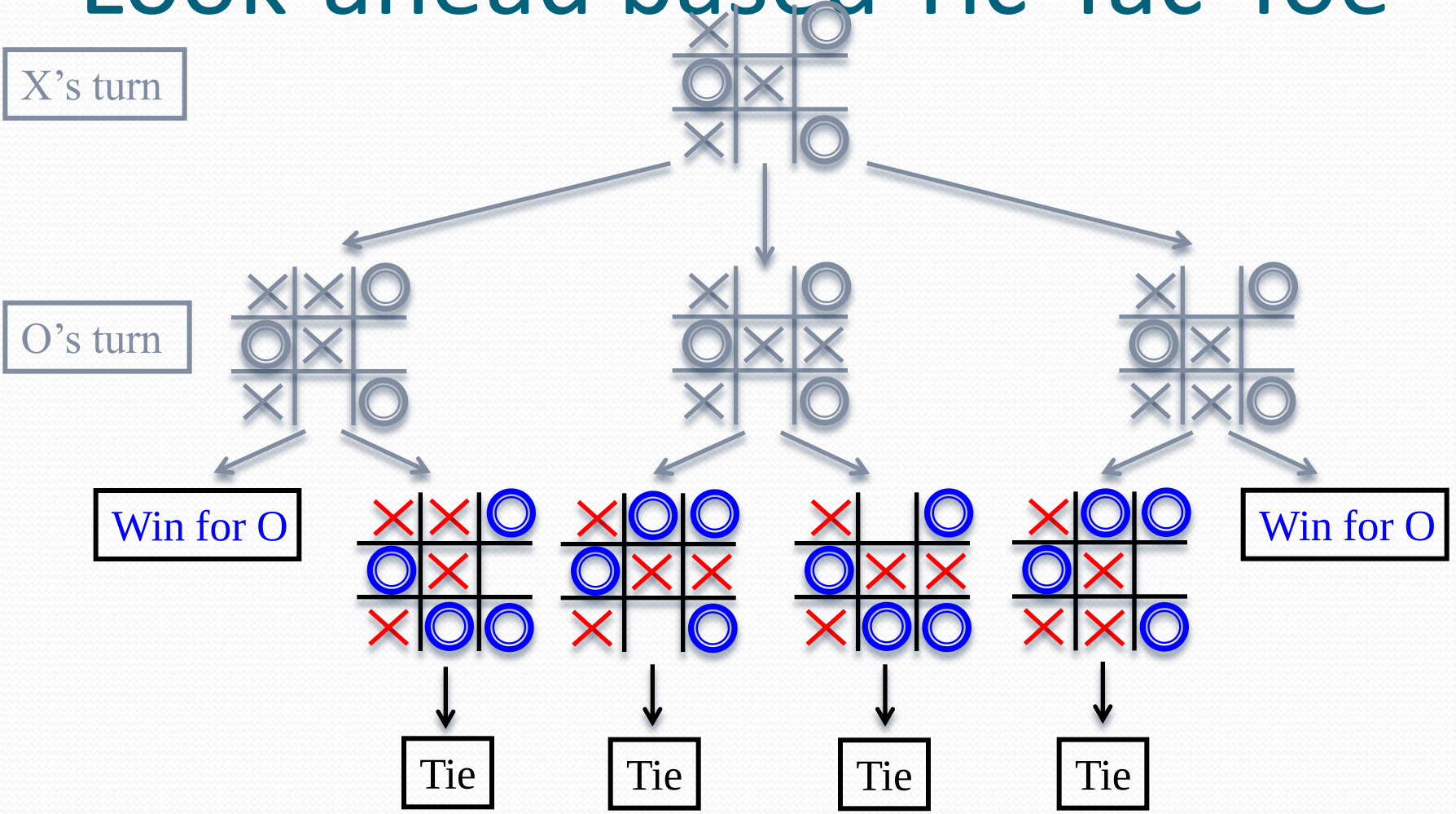
Look-ahead based Tic-Tac-Toe



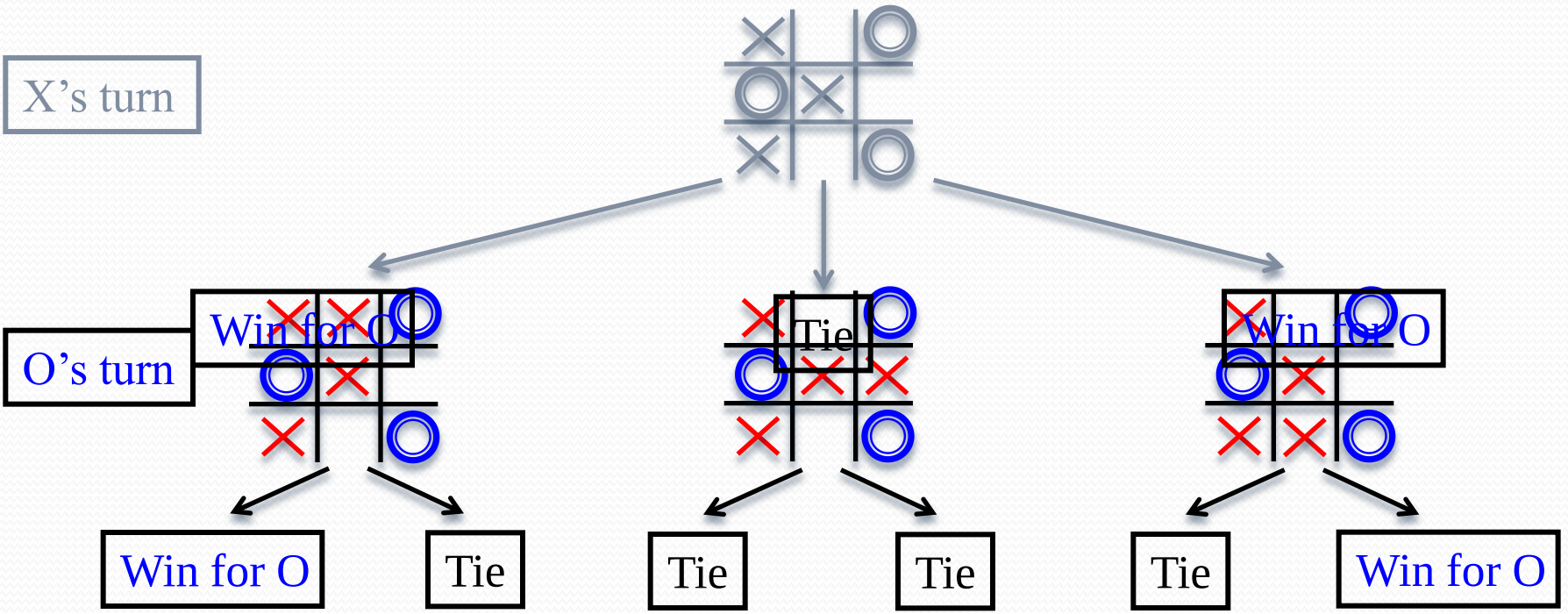
Look-ahead based Tic-Tac-Toe



Look-ahead based Tic-Tac-Toe

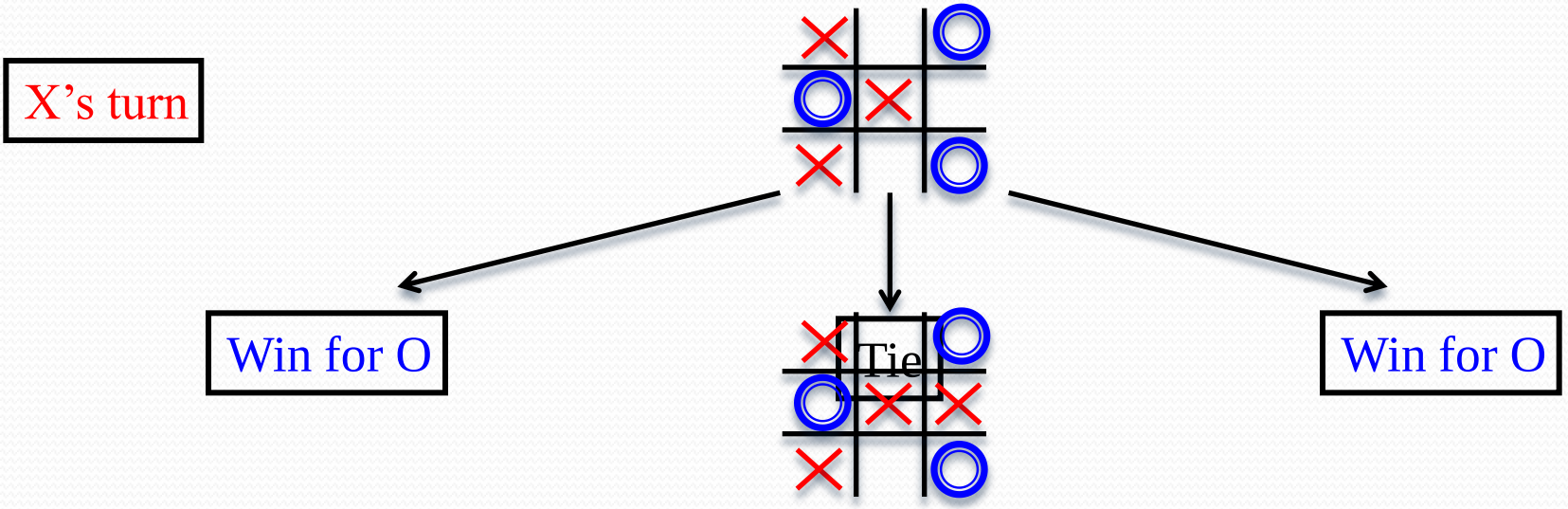


Look-ahead based Tic-Tac-Toe



Each board in game tree gets *unique* game tree value (utility; -1/0/+1) under optimal rational play.

E.g. 0 for top board.



- Approach: Look first at bottom tree. Label bottom-most boards.
- Then label boards one level up, according result of **best possible move.**
- … and so on. Moving up layer by layer.
- Termed the **Minimax Algorithm**
  - Implemented as a depth-first search

Alpha-Beta Procedure

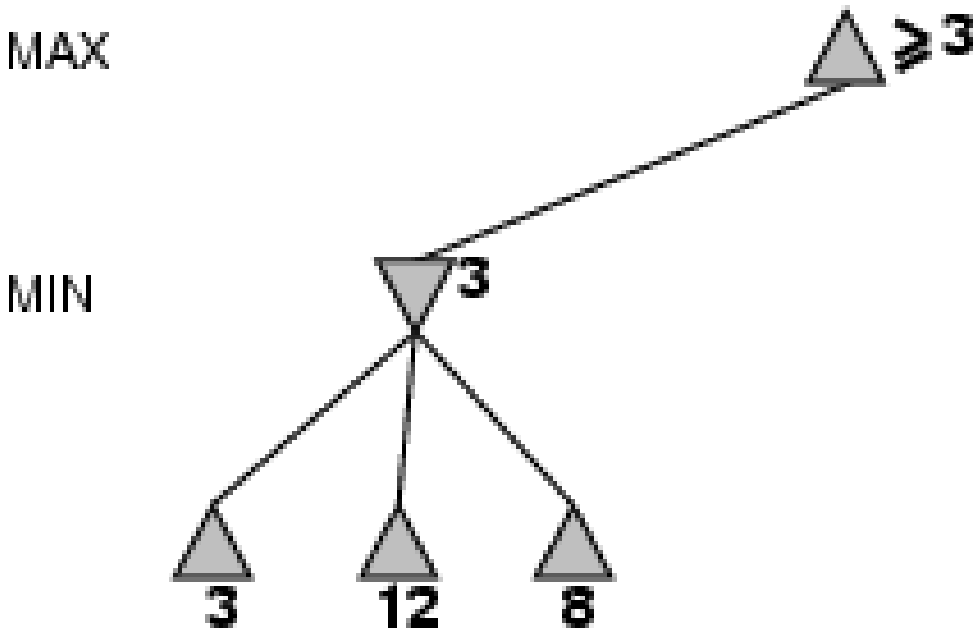
- The alpha-beta procedure can speed up a depth-first minimax search.
- Alpha: a lower bound on the value that a max node may ultimately be assigned

$$v \geq \alpha$$

- Beta: an upper bound on the value that a minimizing node may ultimately be assigned

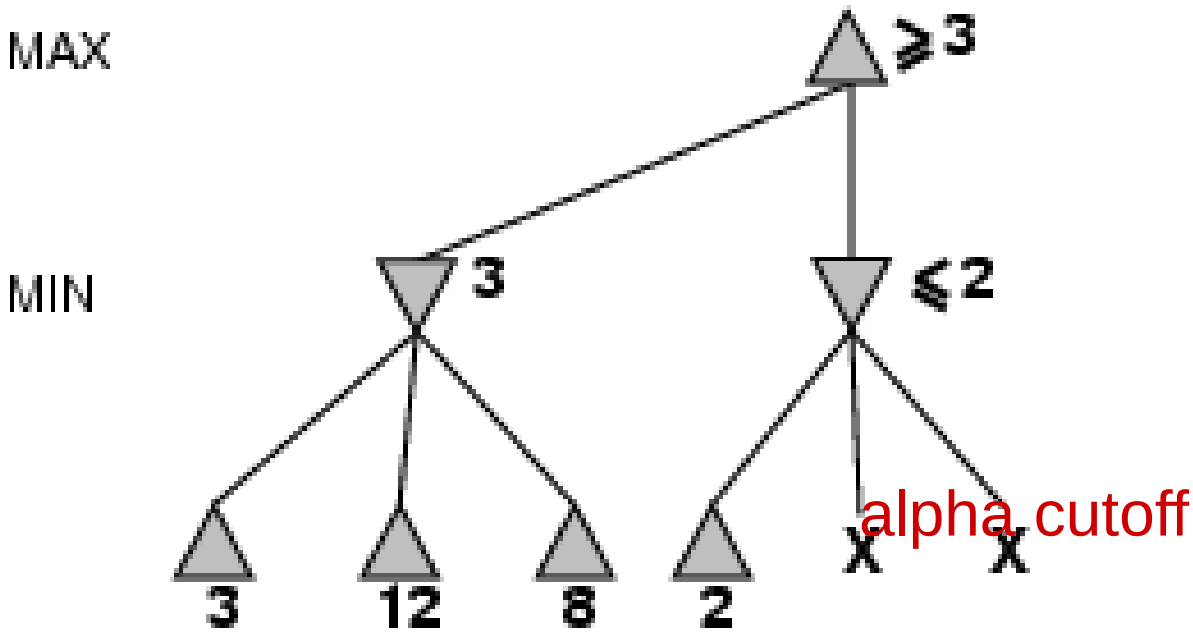
$$v \leq \beta$$

α-β pruning example

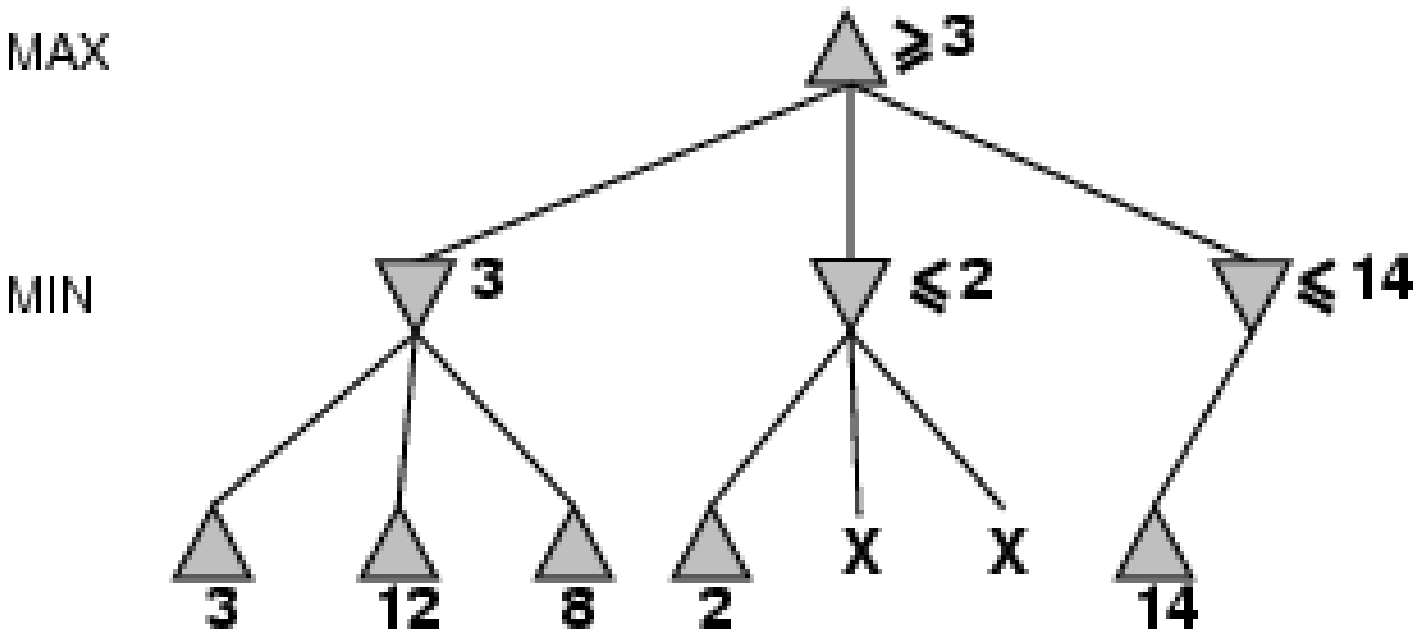


α-β pruning example

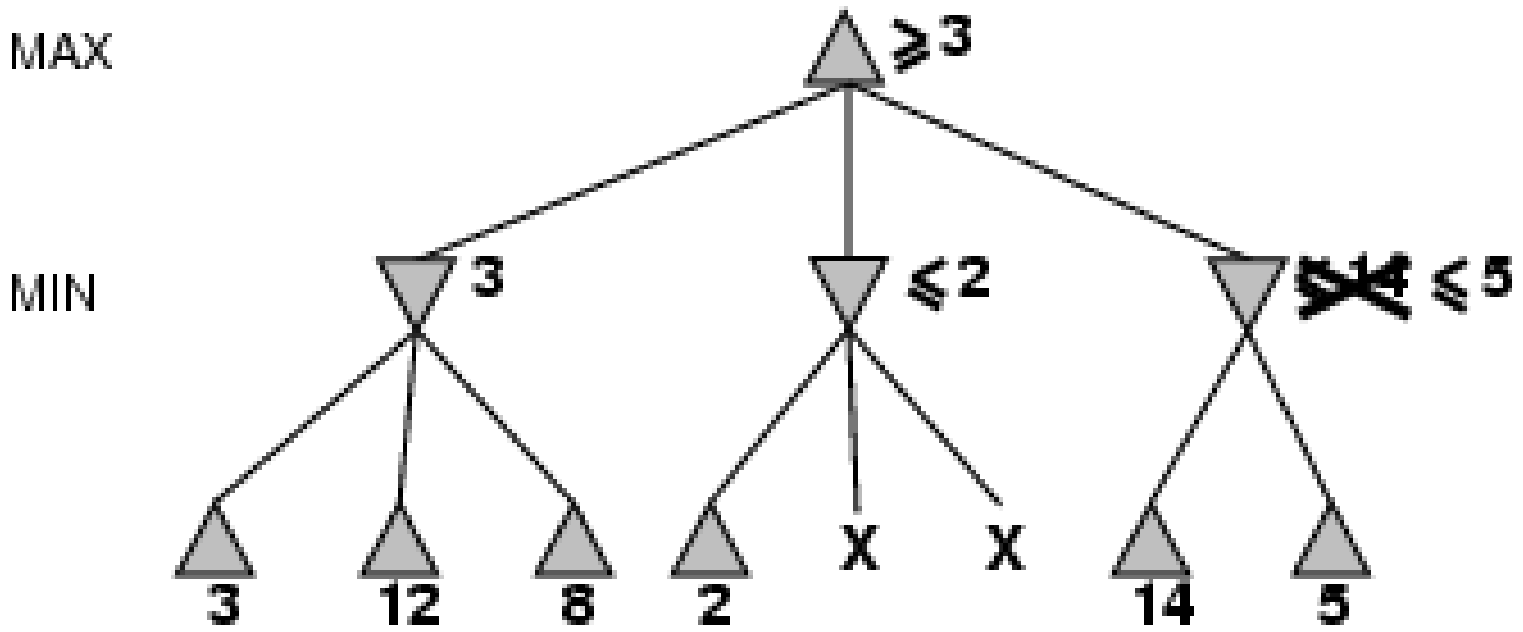
$\alpha = 3$



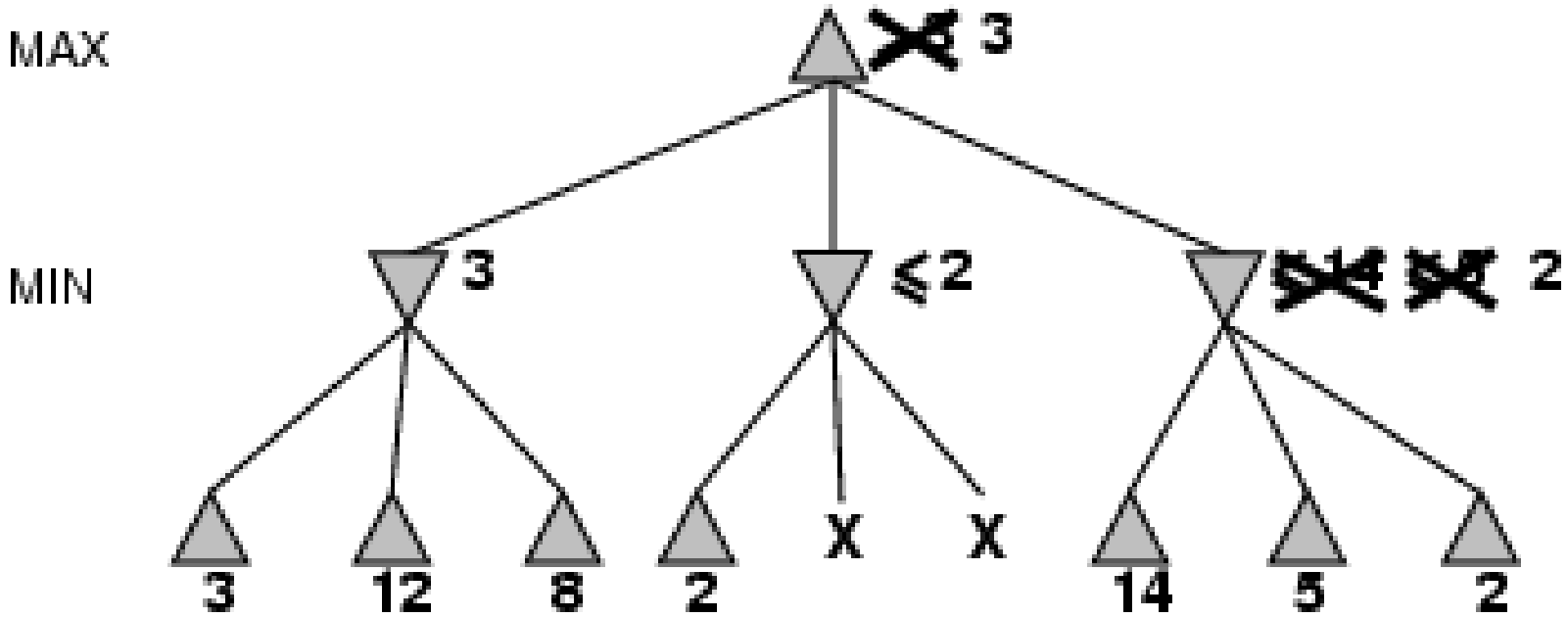
α-β pruning example



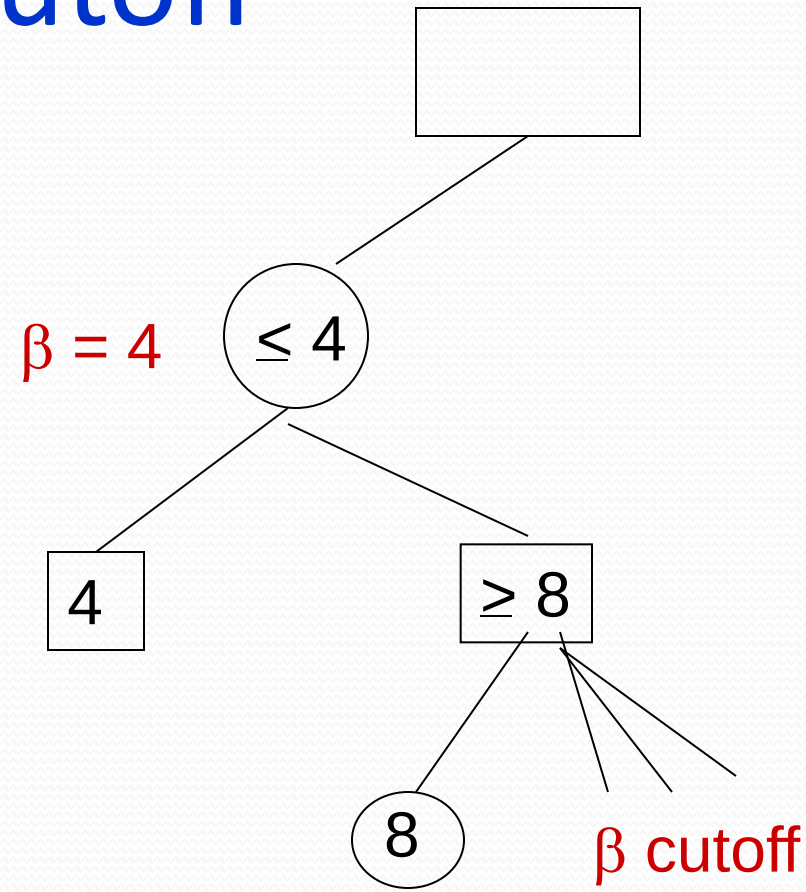
α-β pruning example



α-β pruning example



Beta Cutoff



The α-β algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** *an action*
  inputs: *state*, current state in game

  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in SUCCESSORS(*state*) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  inputs: *state*, current state in game
    α, the value of the best alternative for MAX along the path to *state*
    β, the value of the best alternative for MIN along the path to *state*

  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for a, s in SUCCESSORS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s, α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

The α-β algorithm

```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(s, α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Alpha-Beta Pruning (αβ prune)

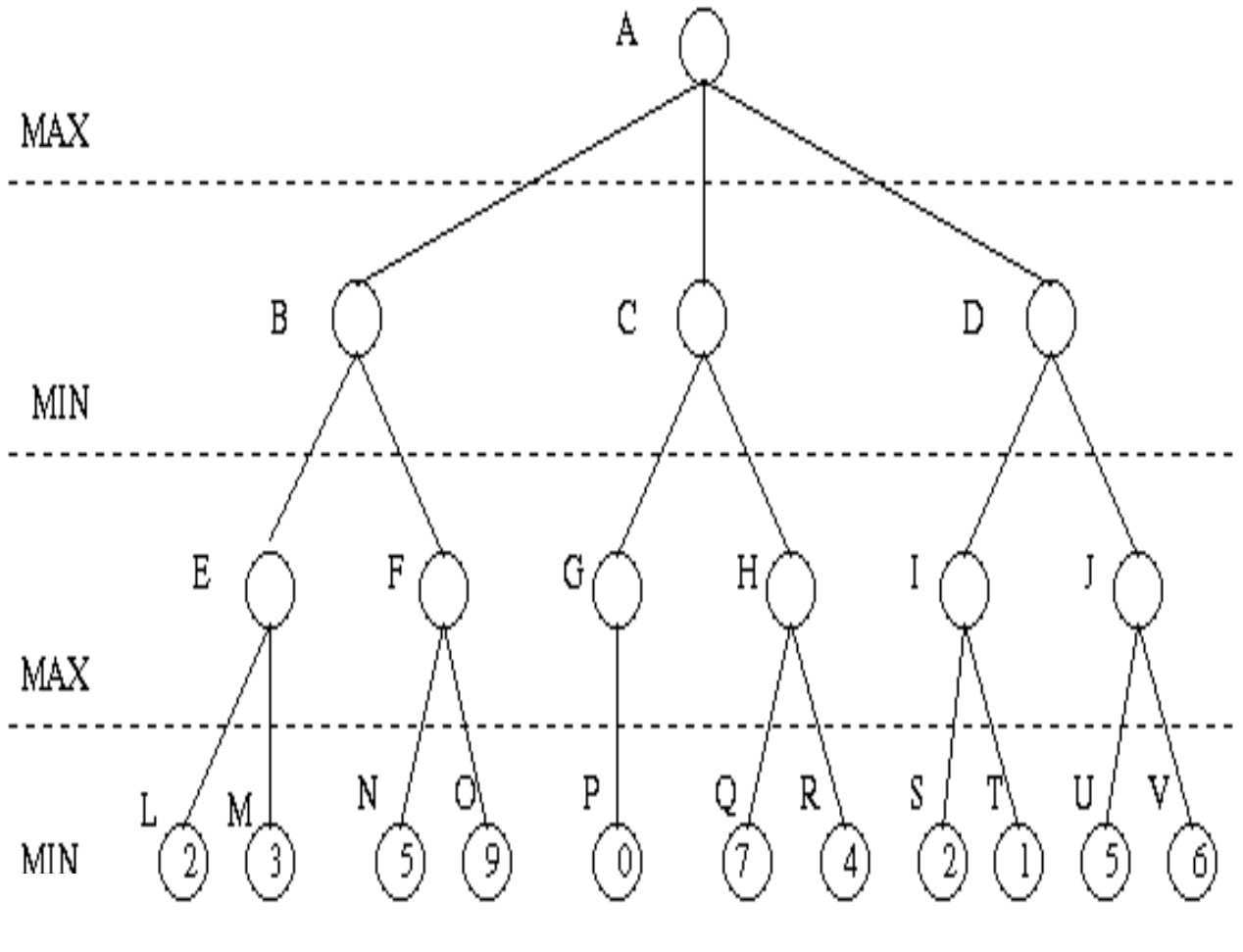
- Rules of Thumb

  - α is the best (highest) found so far along the path for Max
  - β is the best (lowest) found so far along the path for Min
  - Search below a MIN node may be alpha-pruned if the its $\beta \leq \alpha$ of some MAX ancestor
  - Search below a MAX node may be **beta-pruned** if the its $\alpha \geq \beta$ of some MIN ancestor.

Alpha-Beta Pruning Example

1. Search below a MIN node may be alpha-pruned if the beta value is <= to the alpha value of some MAX ancestor.

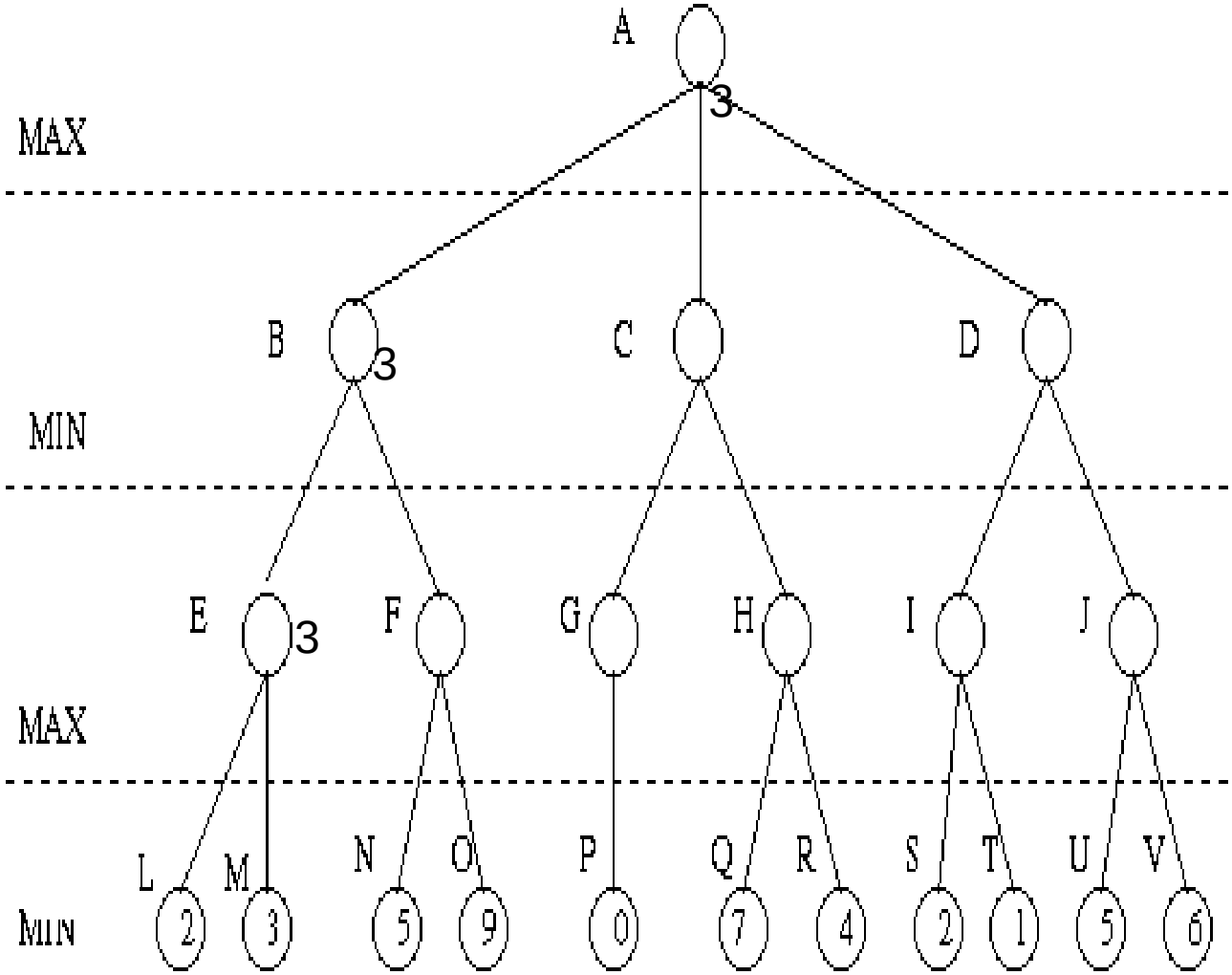
2. Search below a MAX node may be beta-pruned if the alpha value is >= to the beta value of some MIN ancestor.



Alpha-Beta Pruning Example

1. Search below a MIN node may be alpha-pruned if the beta value is <= to the alpha value of some MAX ancestor.

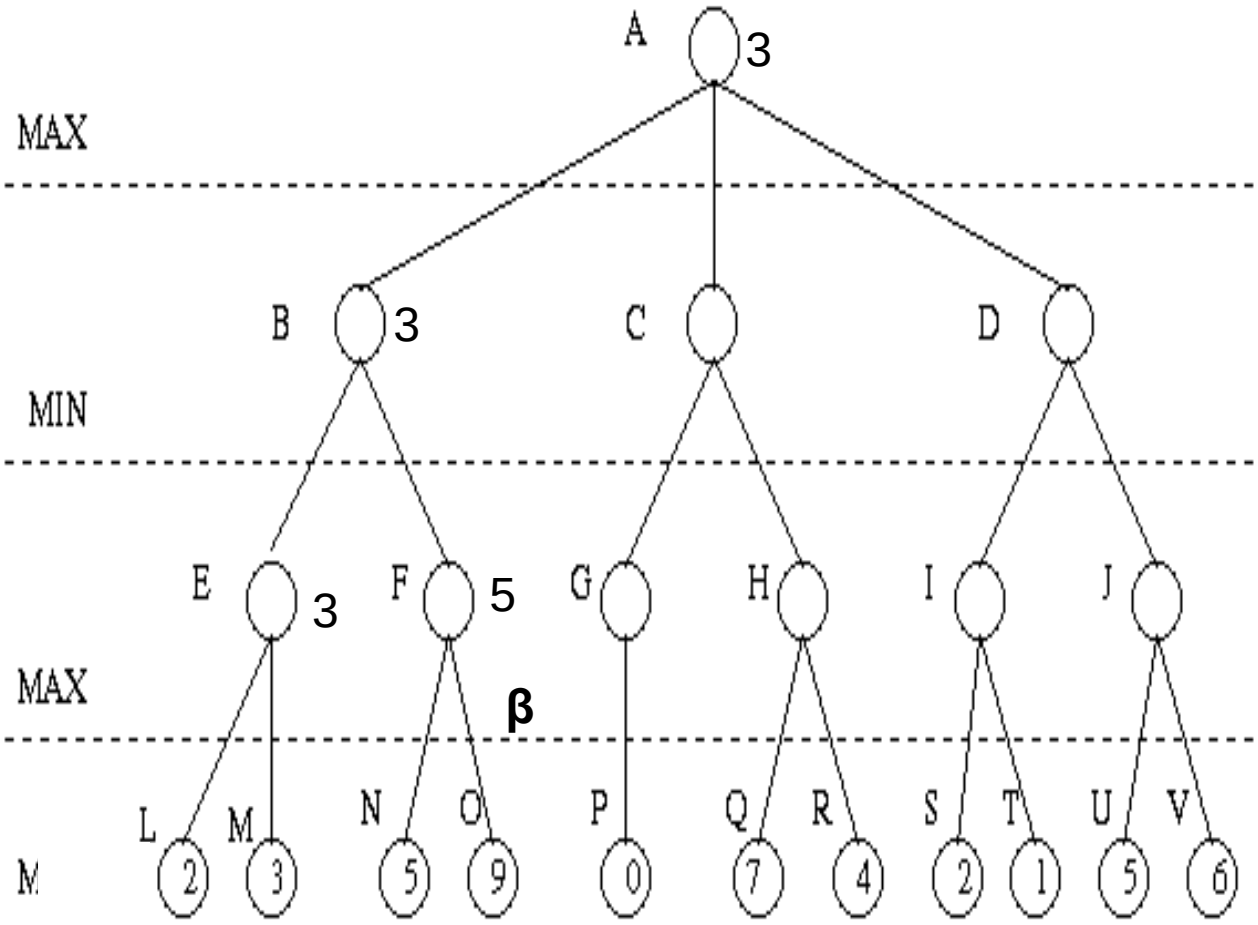
2. Search below a MAX node may be beta-pruned if the alpha value is >= to the beta value of some MIN ancestor.



Alpha-Beta Pruning Example

1. Search below a MIN node may be alpha-pruned if the beta value is <= to the alpha value of some MAX ancestor.

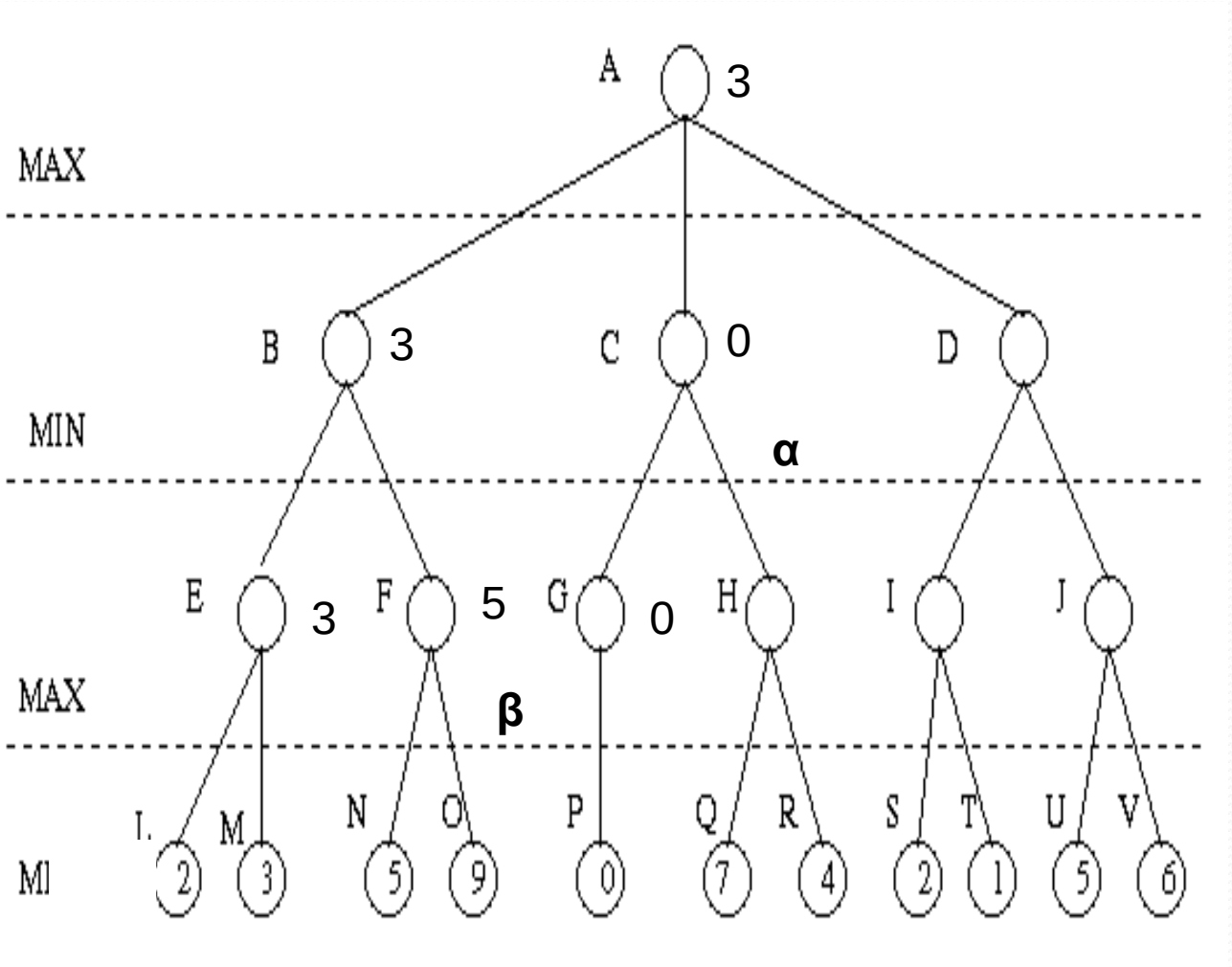
2. Search below a MAX node may be beta-pruned if the alpha value is >= to the beta value of some MIN ancestor.



Alpha-Beta Pruning Example

1. Search below a MIN node may be alpha-pruned if the beta value is <= to the alpha value of some MAX ancestor.

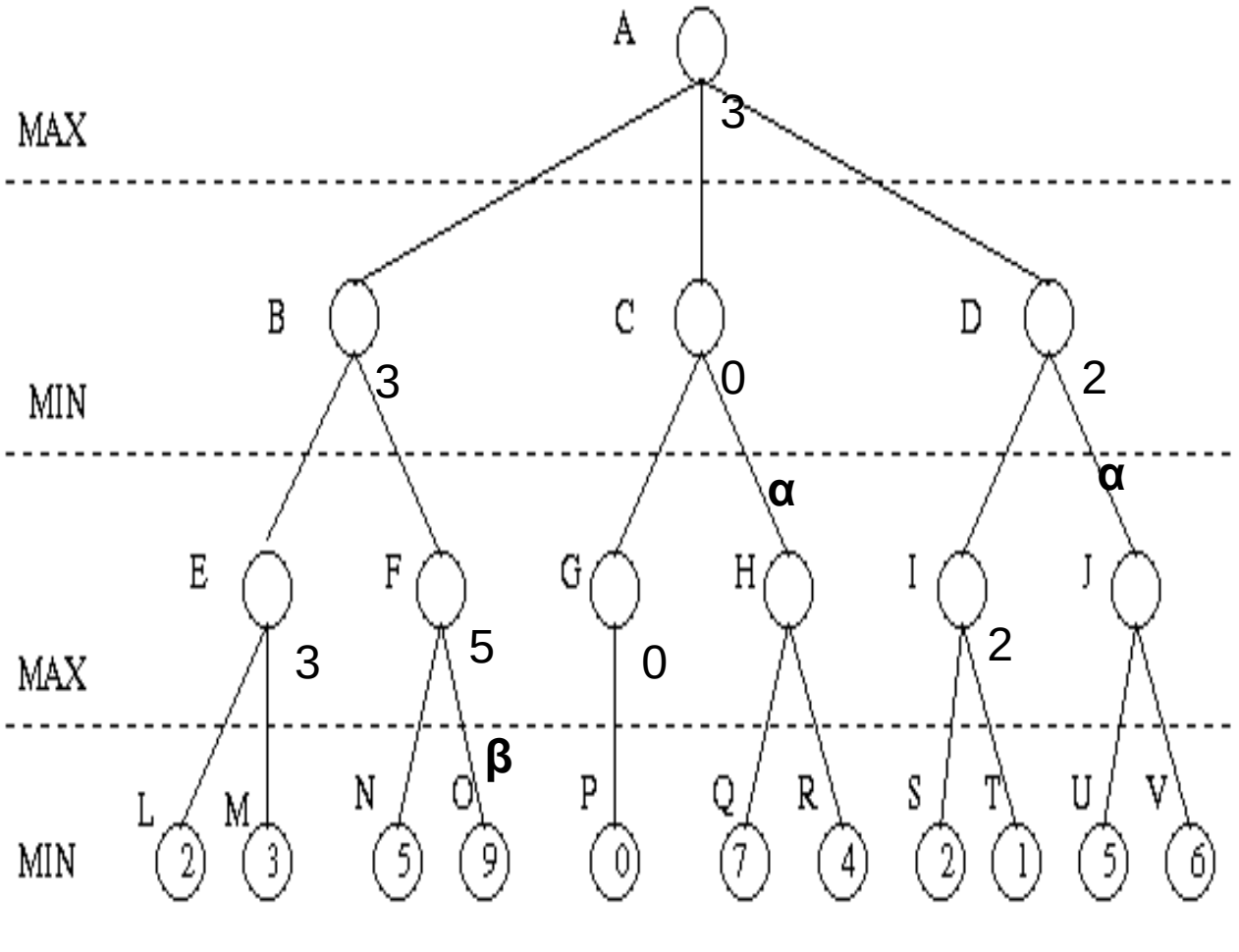
2. Search below a MAX node may be beta-pruned if the alpha value is >= to the beta value of some MIN ancestor.

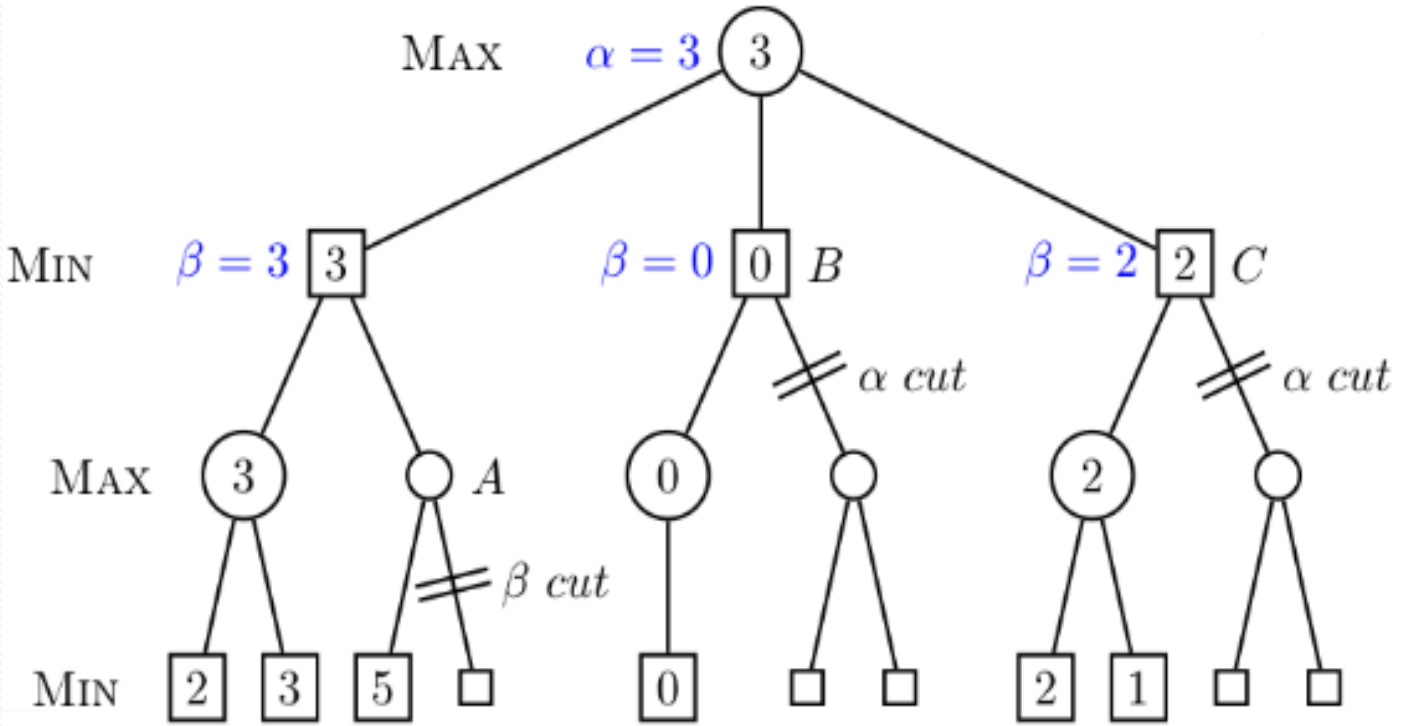


Alpha-Beta Pruning Example

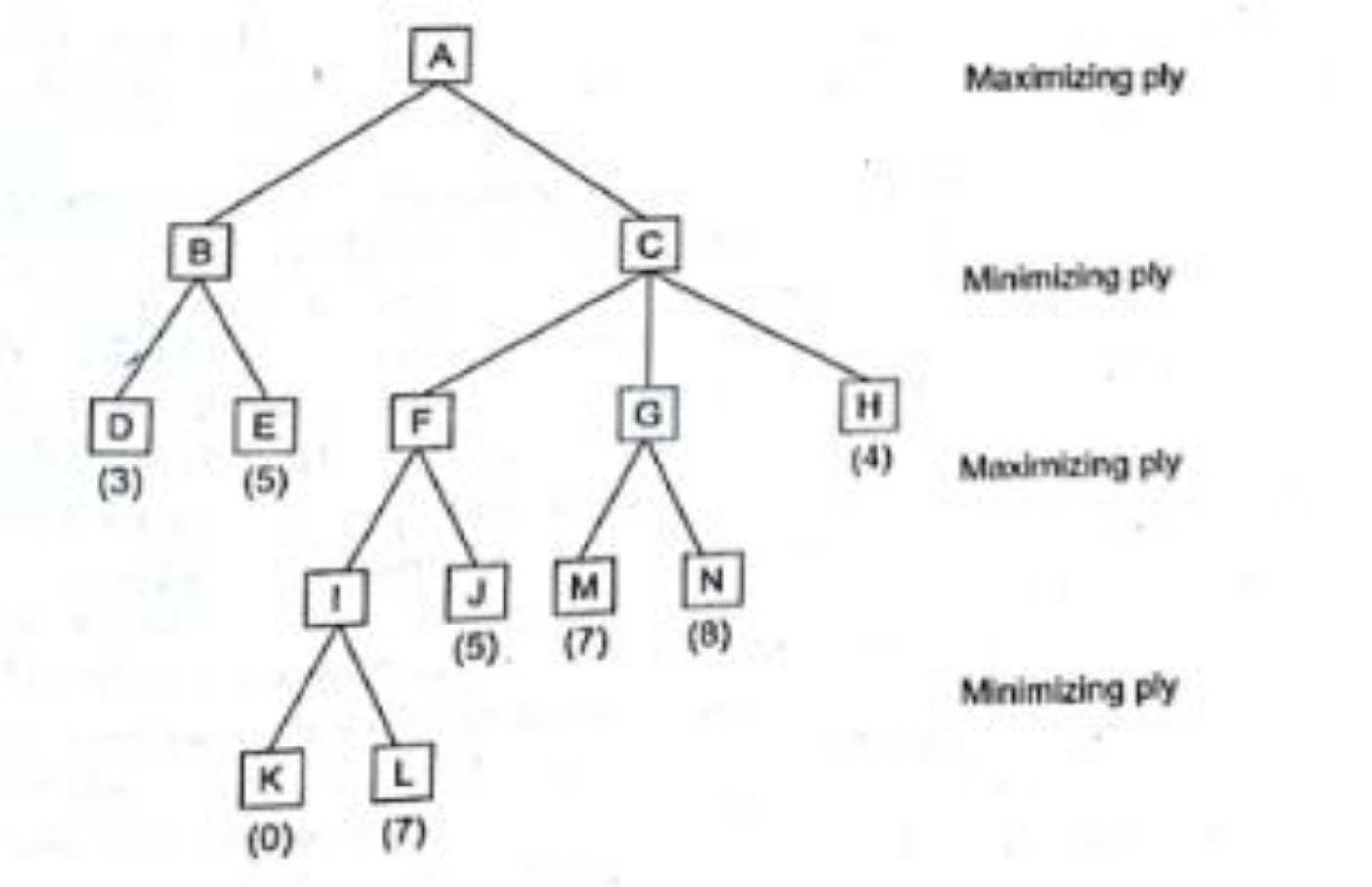
1. Search below a MIN node may be alpha-pruned if the beta value is <= to the alpha value of some MAX ancestor.

2. Search below a MAX node may be beta-pruned if the alpha value is >= to the beta value of some MIN ancestor.



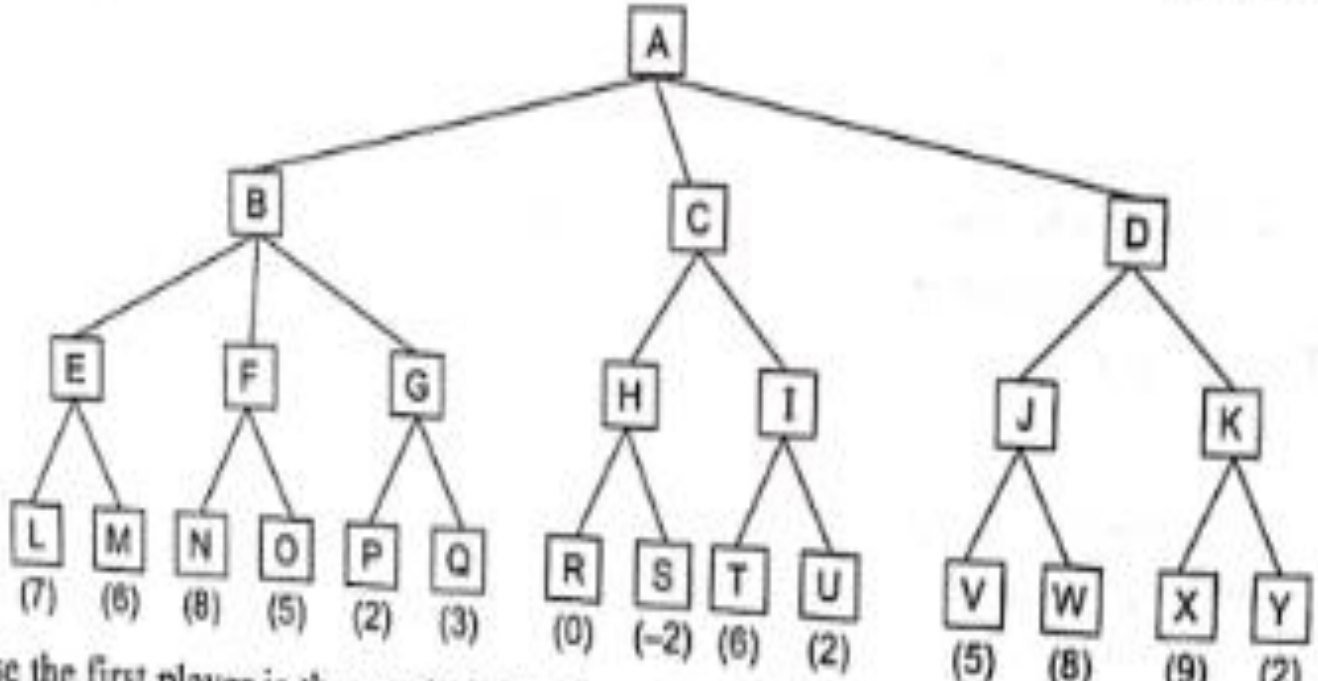

MAX
$\alpha = 3$
3
MIN
$\beta = 3$
3
$\beta = 0$
0
B
$\beta = 2$
2
C
α cut
α cut
MAX
3
A
0
2
β cut
MIN
2
3
5
0
2
1

Home work problem -6



Home work problem -7

Consider the following game tree in which static scores are all from the first player's point of view.



Suppose the first player is the maximizing player. What move should be chosen?