

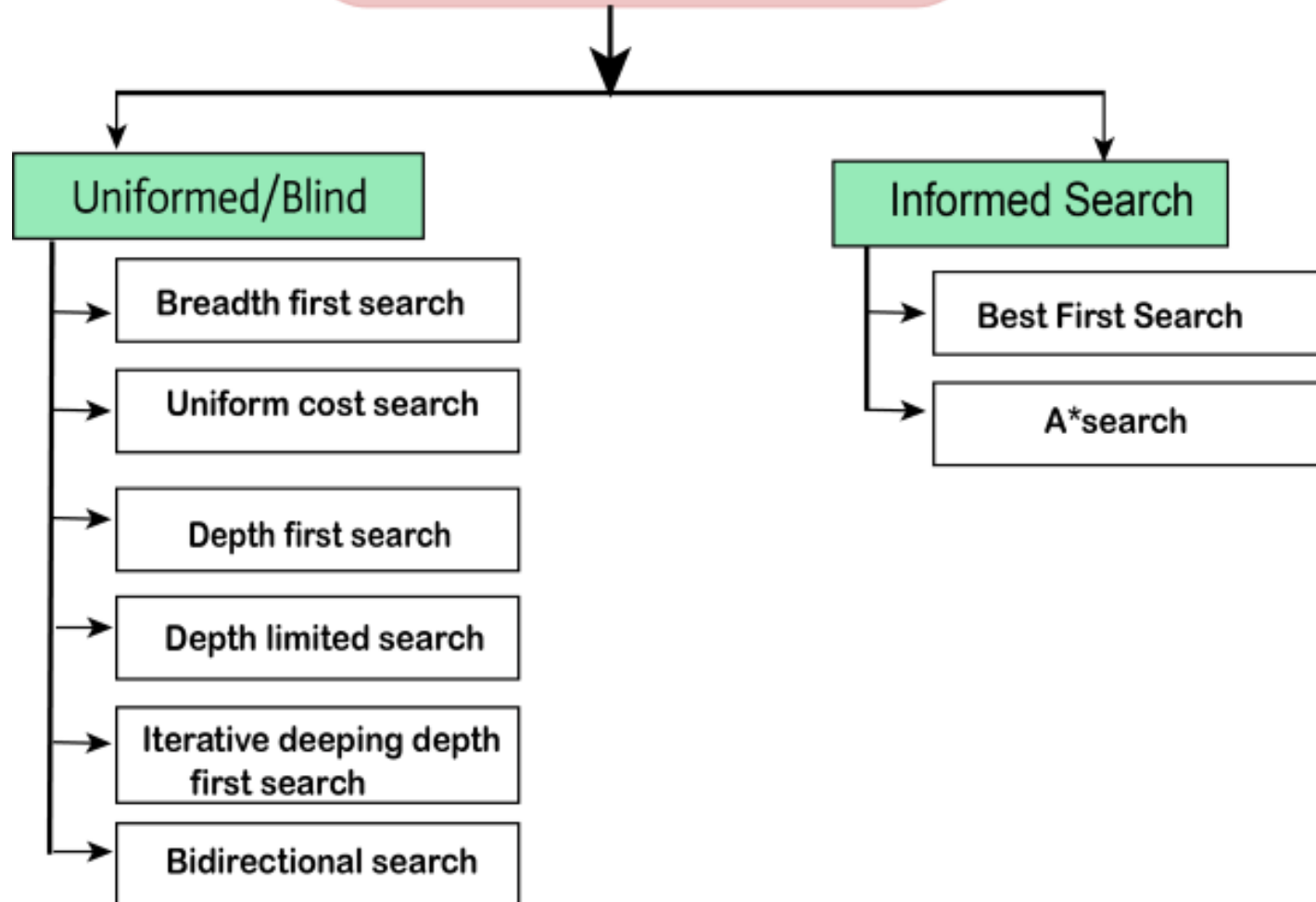
Unit 2: Representation and Search

Unit 3: Adversarial Search

Prepared by Prof. Kore S.S

(Assistant Professor(CSE) DYPCET, Kolhapur)

Search Algorithm



Uninformed/Blind Search:

- The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes. Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search. It examines each node of the tree until it achieves the goal node.

Informed Search /Heuristic search

- Informed search algorithms use domain knowledge. In an informed search, problem information is available which can guide the search. Informed search strategies can find a solution more efficiently than an uninformed search strategy. Informed search is also called a Heuristic search.

Blind Search

- This type of search takes all nodes of tree in specific order until it reaches to goal. The order can be in breath and the strategy will be called breadth–first search, or in depth and the strategy will be called depth first search.

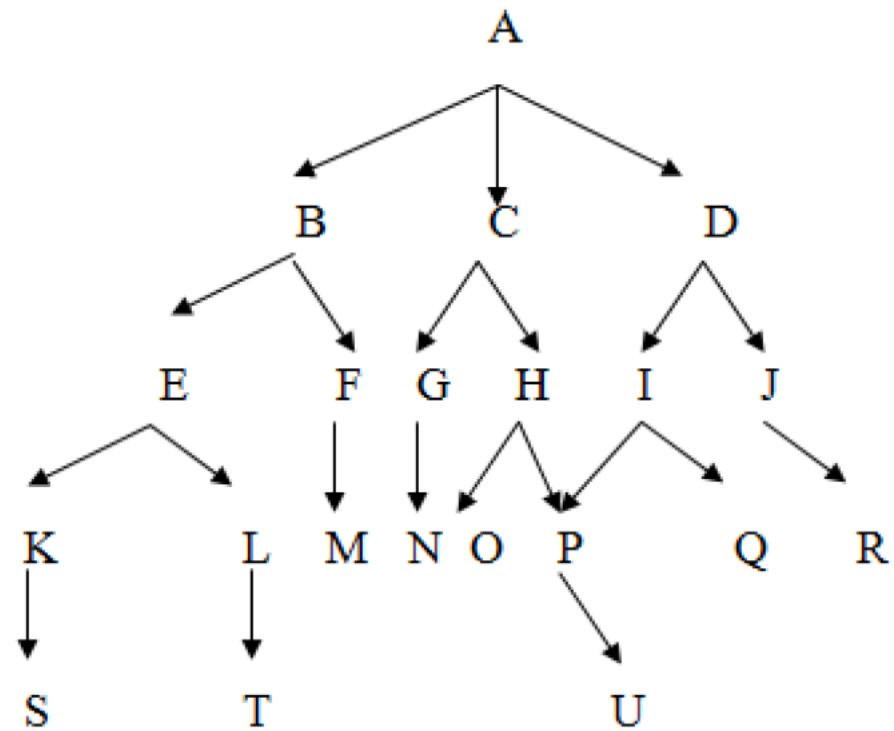
1- Breadth – First – Search

In breadth –first search, when a state is examined, all of its siblings are examined before any of its children. The space is searched level-by-level, proceeding all the way across one level before doing down to the next level.

Breadth – first – search Algorithm

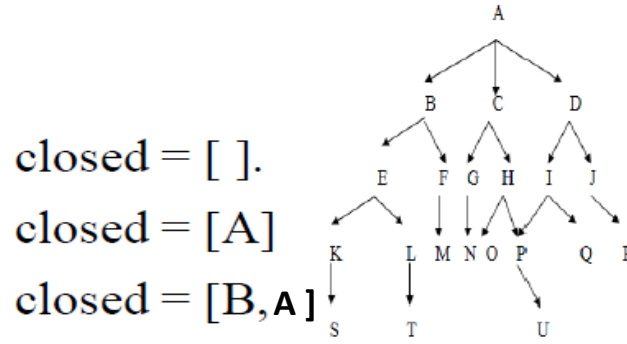
- *Begin*
- *Open: = [start];*
- *Closed: = [];*
- *While open \neq [] do*
- *Begin*
- *Remove left most state from open, call it x;*
- *If x is a goal the return (success)*
- *Else*
- *Begin*
- *Generate children of x;*
- *Put x on closed;*
- *Eliminate children of x on open or closed; (Removing the repeated child or node)*
- *Put remaining children on right end of open*
- *End*
- *End*
- *Return (failure)*
- *End.*

Breadth – First – Search



Breadth – First – Search

- 1 – Open= [A];
- 2 – Open= [B, C, D];
- 3 – Open= [C, D, E, F];
- 4 – Open= [D, E, F, G, H];
- 5 – Open= [E, F, G, H, I, J];
- 6 – Open= [F, G, H, I, J, K, L];
- 7 – Open= [G, H, I, J, K, L, M]; closed = [F, E, D, C, B, A].
- 8 – Open= [H, I, J, K, L, M, N,]; closed = [G, F, E, D, C, B, A].
- 9 – and so on until either U is found or open = [].



closed = [].

closed = [A]

closed = [B, A]

closed = [C, B, A].

closed = [D, C, B, A].

closed = [E, D, C, B, A].

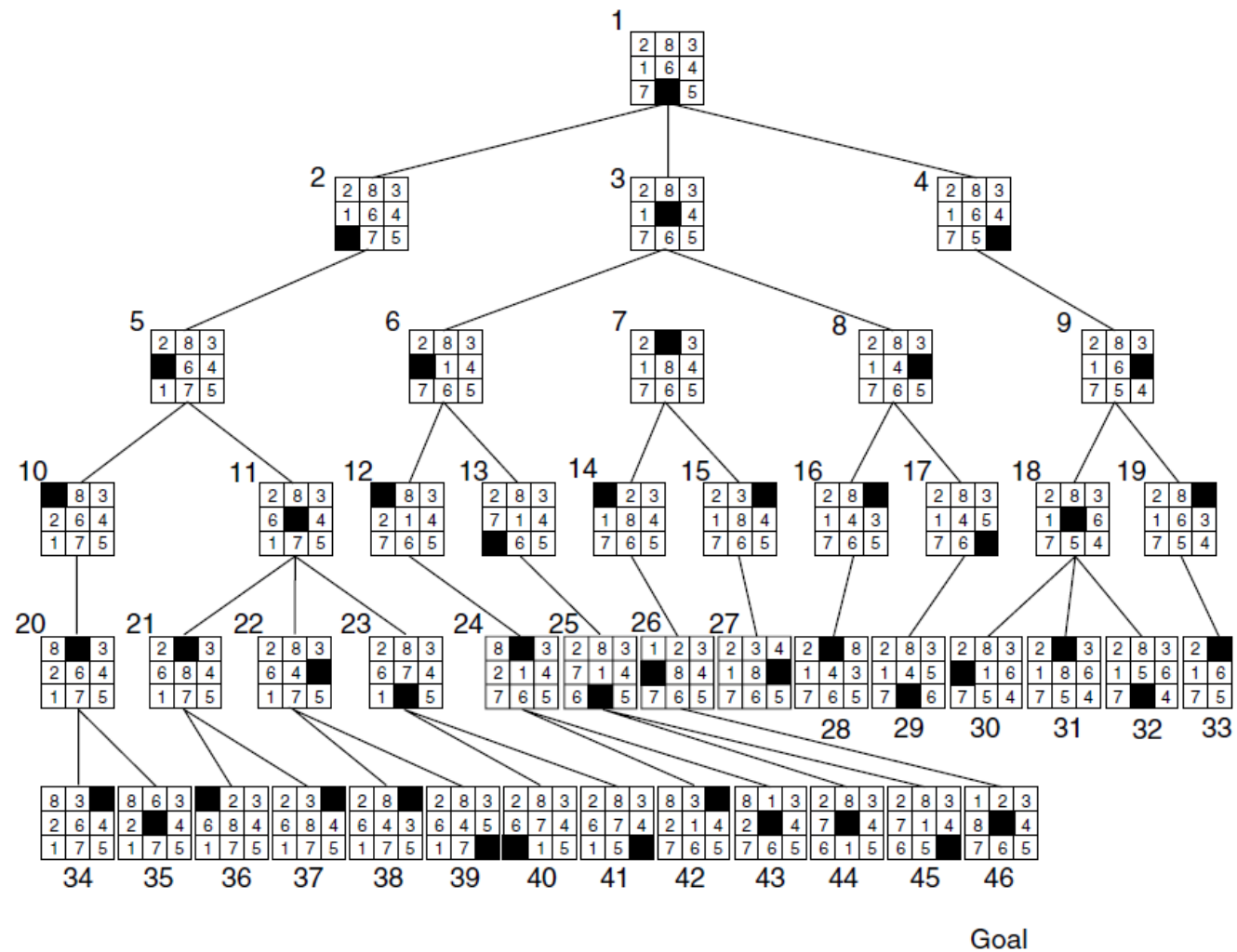


Figure 3.17 Breadth-first search of the 8-puzzle, showing order in which states were removed from open.

2- Depth – first – search

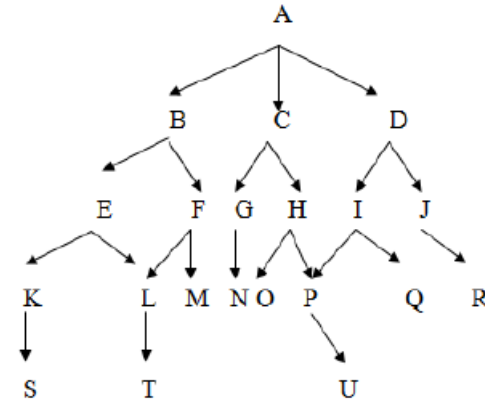
- In depth – first – search, when a state is examined, all of its children and their descendants are examined before any of its siblings.
- Depth – first search goes deeper in to the search space when ever this is possible only when no further descendants of a state can found.

Depth – first – search Algorithm

- *Begin*
- *Open: = [start];*
- *Closed: = [];*
- *While open \neq [] do*
- *Remove leftmost state from open, call it x;*
- *If x is a goal then return (success)*
- *Else begin*
- *Generate children of x;*
- *Put x on closed;*
- *Eliminate children of x on open or closed; (Removing the repeated child or node)*
- *put remaining children on left end of open end*
- *End;*
- *Return (failure)*
- *End.*

Depth – first – search

- 1 – Open= [A]; closed = [].
- 2 – Open= [B, C, D]; closed = [A].
- 3 – Open= [E, F, C, D]; closed = [B, A].
- 4 – Open= [K, L, F, , D]; closed = [E, B, A].
- 5 – Open= [S, L, F, C, D]; closed = [K, E, B, A].
- 6 – Open= [L, F, C, D]; closed = [S, K, E, B, A].
- 7 – Open= [T, F, C, D]; closed = [L, S, K, E, B, A].
- 8 – Open= [F, C, D,]; closed = [T, L, S, K, E, B, A].
- 9 – Open= [M, C, D] as L is already on; closed = [F, T, L, S, K, E, B, A].
- 10 – and so on until either U is found or open = [].



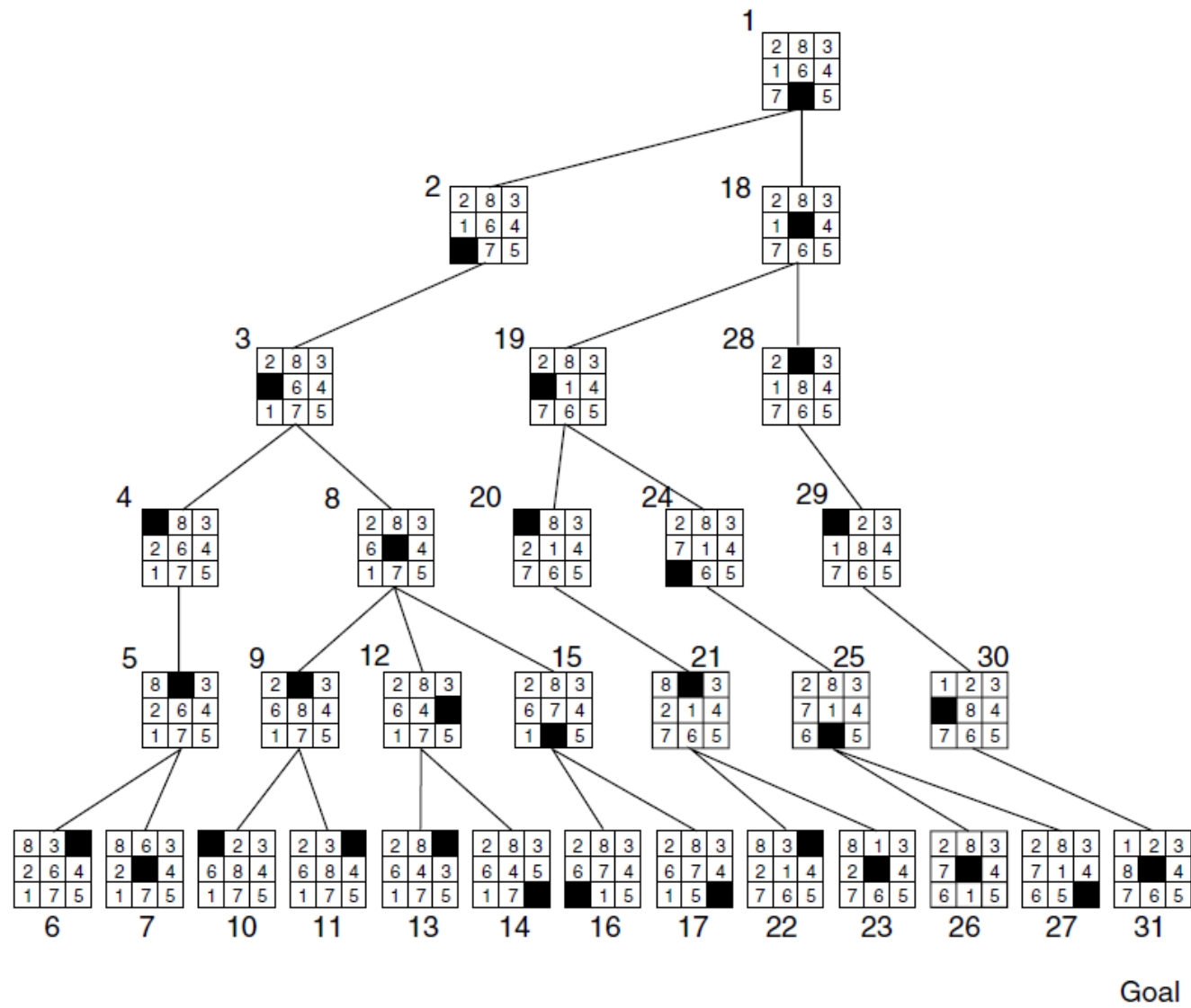


Figure 3.19 Depth-first search of the 8-puzzle with a depth bound of 5.

Uniform Cost Search Algorithm - Artificial Intelligence

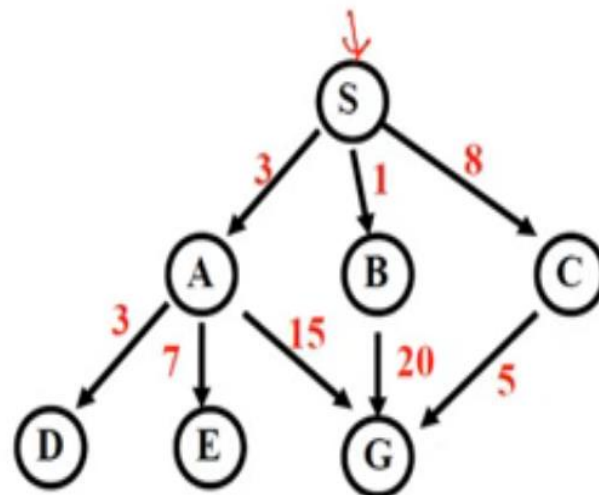
- Uniform-cost search (Branch & Bound) is an uninformed search algorithm in Artificial Intelligence
 - UCS algorithm uses the lowest cumulative cost to find a path from the source node to the goal node.
 - Nodes are expanded, starting from the root, according to the minimum cumulative cost.
 - The uniform-cost search is implemented using a Priority Queue.
-

Uniform Cost Search Algorithm - Artificial Intelligence

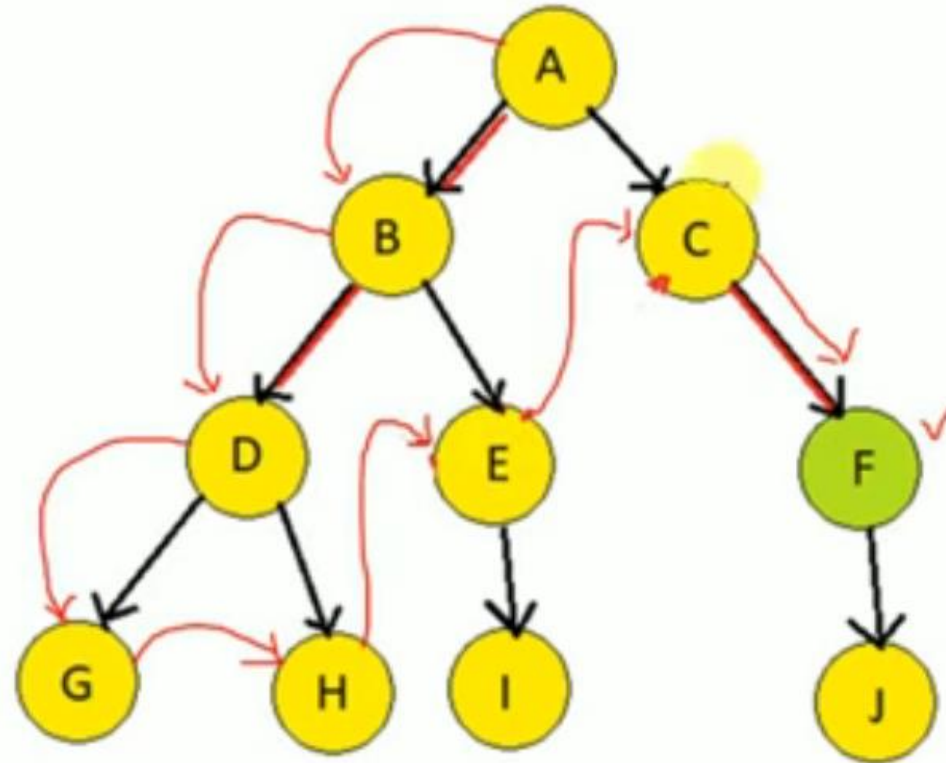
- Insert the root node into the priority queue.
- Remove the element with the highest priority.
- If the removed node is the goal node,
 - print total cost and stop the algorithm
- Else
 - Enqueue all the children of the current node to the priority queue, with their cumulative cost from the root as priority and the current node to the visited list.

Uniform-Cost Search...

- Expanded node Nodes list
 - { S0 }
 - S0 { B1, A3, C8 }
 - B1 { A3, C8, G21 }
 - A3 { D6, C8, E10, G18, G21 }
 - D6 { C8, E10, G18, G21 }
 - C8 { E10, G13, G18, G21 }
 - E10 { G13, G18, G21 }
 - G13 { G18, G21 }
- Solution path found is S C G, cost 13
- Number of nodes expanded (including goal node) = 7

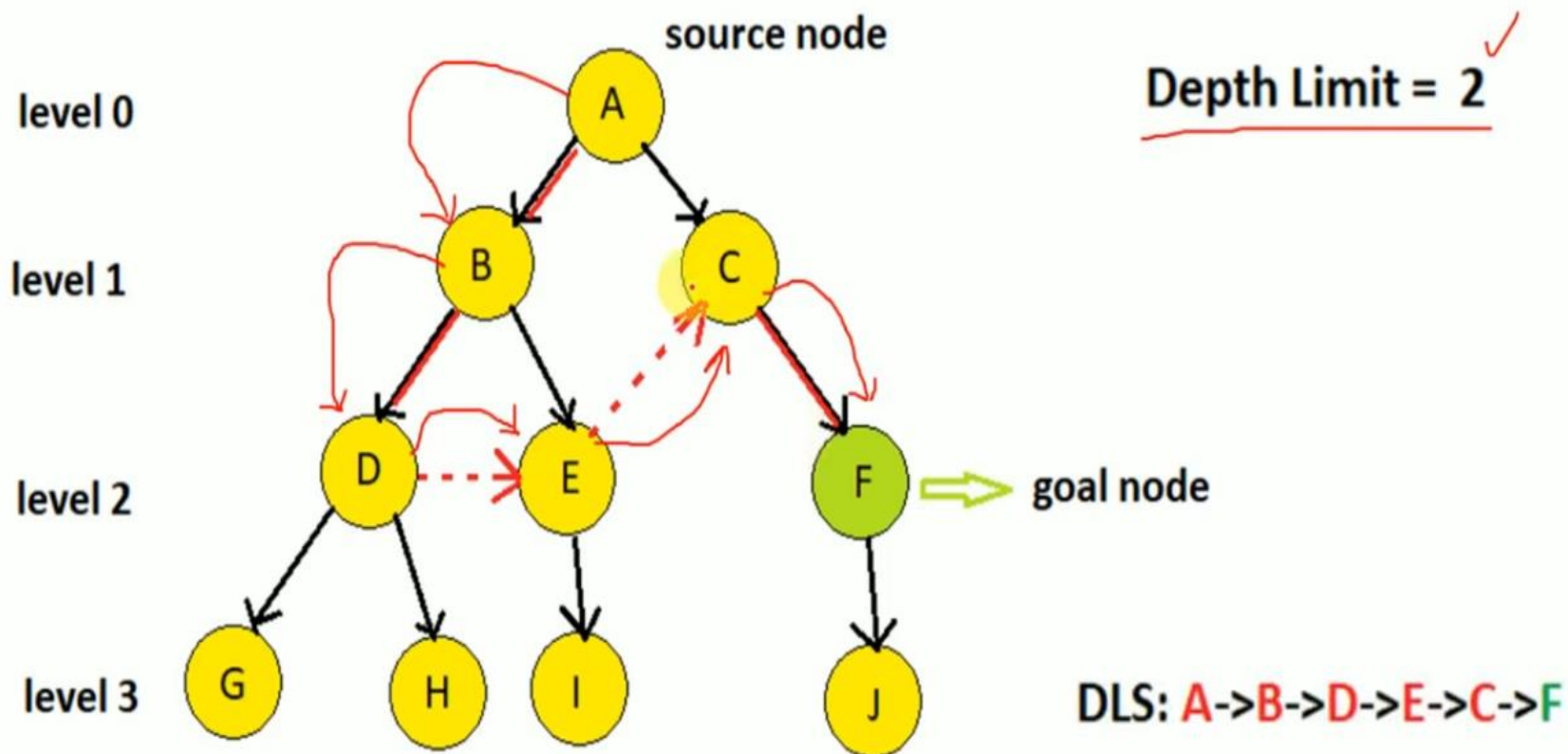


Depth First Search Algorithm in Artificial Intelligence

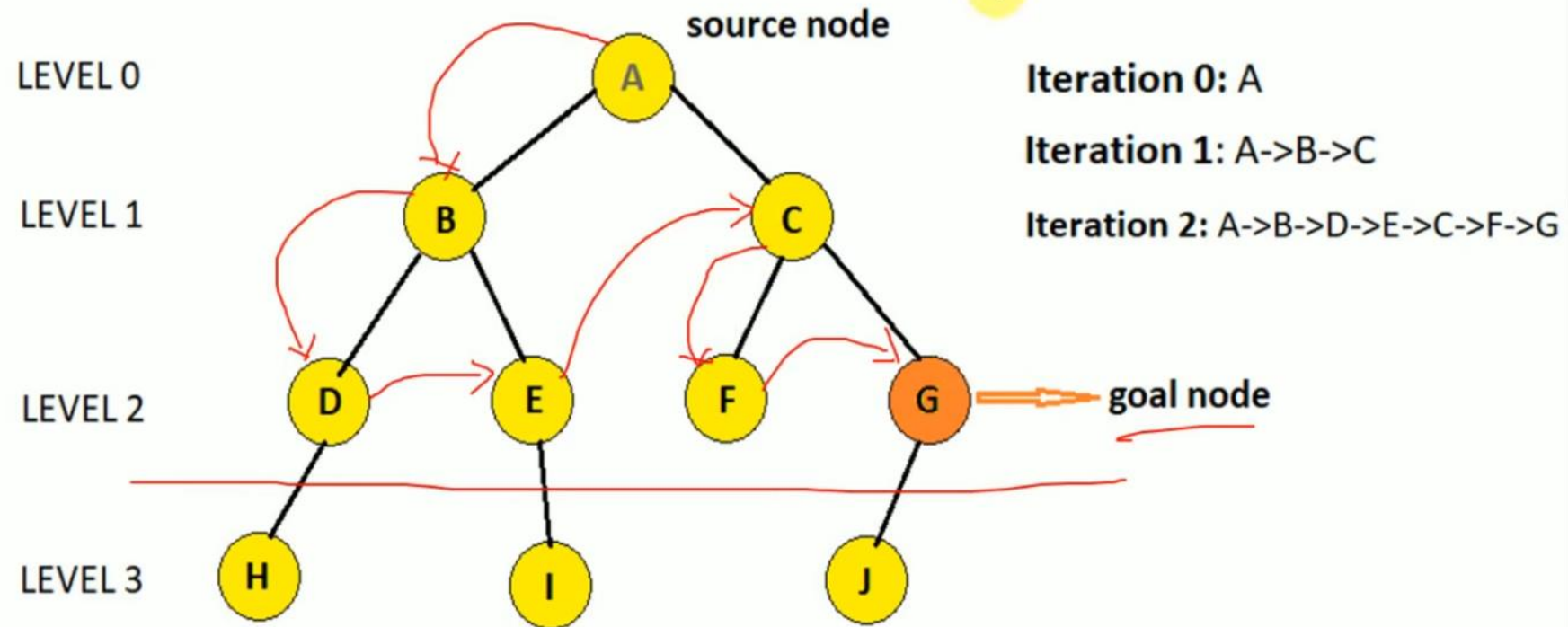


DFS: A->B->D->G->H->E->I->C->F

Depth Limited Search Algorithm in Artificial Intelligence



Iterative Deepening Depth First Search Algorithm in Artificial Intelligence



Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one from initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.

Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

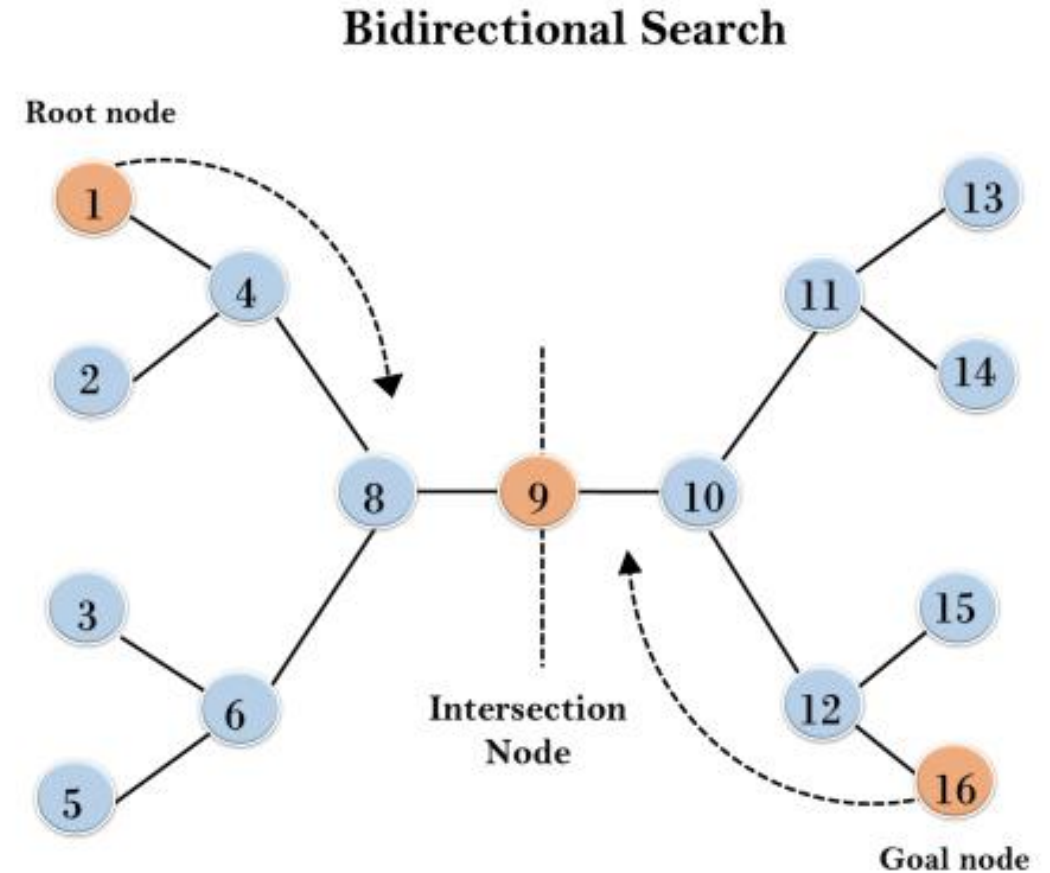
Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**

- In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.
- The algorithm terminates at node 9 where two searches meet. Bidirectional Search is complete if we use BFS in both searches.



Informed Search /Heuristic search

Best-first Search Algorithm (Greedy Search):

- Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

Evaluation function $f(n) = h(n)$.

$h(n)$ = estimated cost from node n to goal

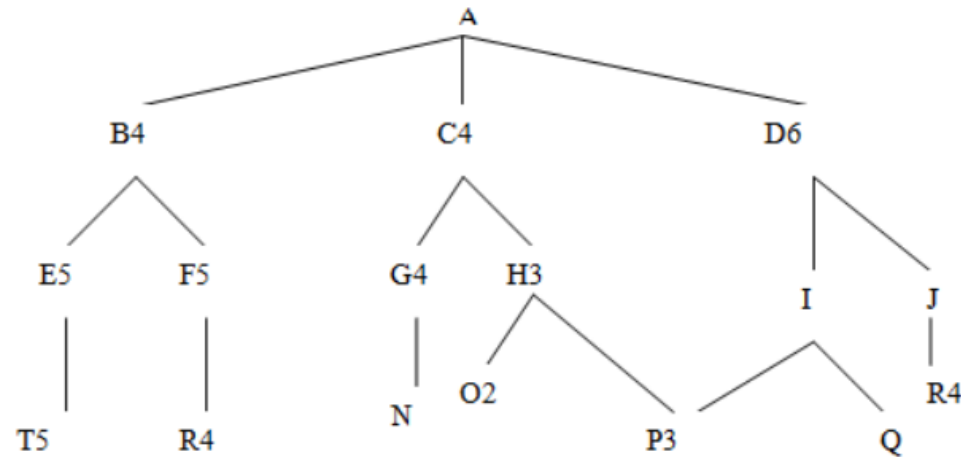
Greedy search ignores the cost of the path that has already been traversed to reach n

Therefore solution given is not optimal

2- Best-First-Search

- Best First search is away of combining the advantages of both depth-first and breadth-first search into a single method.
- In Best-First search, the search space is evaluated according to a heuristic function. Nodes yet to be evaluated are kept on an OPEN list and those that have already been evaluated are stored on a CLOSED list. The OPEN list is represented as a priority queue, such that unvisited nodes can be queued in order of their evaluation function. The evaluation function $f(n)$ is made from only the heuristic function ($h(n)$) as: $f(n) = h(n)$.

2- Best-First-Search



Open=[A5]

Open=[B4,C4,D6]

Open=[C4,E5,F5,D6]

Open=[H3,G4, E5,F5,D6]

Open=[O2,P3,G4,E5,F5,D6]

Open=[P3,G4,E5,F5,D6]

Open=[G4,E5,F5,D6]

Closed=[]

Closed=[A5]

Closed=[B4,A5]

Closed=[C4,B4,A5]

Closed=[H3, C4,B4,A5]

Closed=[O2,H3, C4,B4,A5]

Closed=[P3,O2,H3,C4,B4,A5]

The solution path is: A5 - B4 - C4 - H3 – O2- P3

Best first search algorithm:

- **Step 1:** Place the starting node into the OPEN list.
- **Step 2:** If the OPEN list is empty, Stop and return failure.
- **Step 3:** Remove the node n from the OPEN list, which has the lowest value of $h(n)$, and places it in the CLOSED list.
- **Step 4:** Expand the node n , and generate the successors of node n .
- **Step 5:** Check each successor of node n , and find whether any node is a goal node or not. If any successor node is the goal node, then return success and stop the search, else continue to next step.
- **Step 6:** For each successor node, the algorithm checks for evaluation function $f(n)$ and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both lists, then add it to the OPEN list.
- **Step 7:** Return to Step 2.

Best-First-Search Algorithm

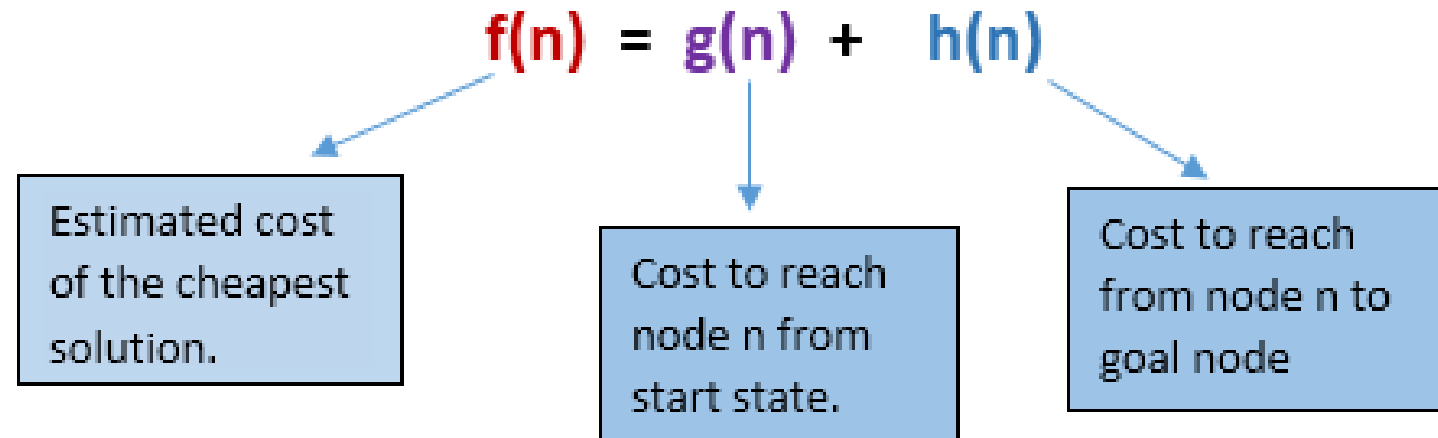
- {
- *Open:=[start];*
- *Closed:=[];*
- *While open $\neq []$ do*
- {
- *Remove the leftmost from open, call it x;*
- *If x= goal then*
- *Return the path from start to x*
- *Else*
- {
- *Generate children of x;*
- *For each child of x do*
- *Do case*
- *The child is not already on open or closed;*
- { *assign a heuristic value to the child state ;*
- *Add the child state to open;*
- }
- }

- *The child is already on open:*
- *If the child was reached along a shorter path than the state currently on open then give the state on open this shorter path value.*
- *The child is already on closed:*
- *If the child was reached along a shorter path than the state currently on open then*
- *{*
- *Give the state on closed this shorter path value*
- *Move this state from closed to open*
- *}*
- *}*
- *Put x on closed;*
- *Re-order state on open according to heuristic (best value first)*
- *}*
- *Return (failure);*
- *}*

A* search Algorithm

A* search is the most commonly known form of best-first search. It uses heuristic function $h(n)$, and cost to reach the node n from the start state $g(n)$. It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses $g(n)+h(n)$ instead of $g(n)$.

In A* search algorithm, we use search heuristic as well as the cost to reach the node. Hence we can combine both costs as following, and this sum is called as a **fitness number**.



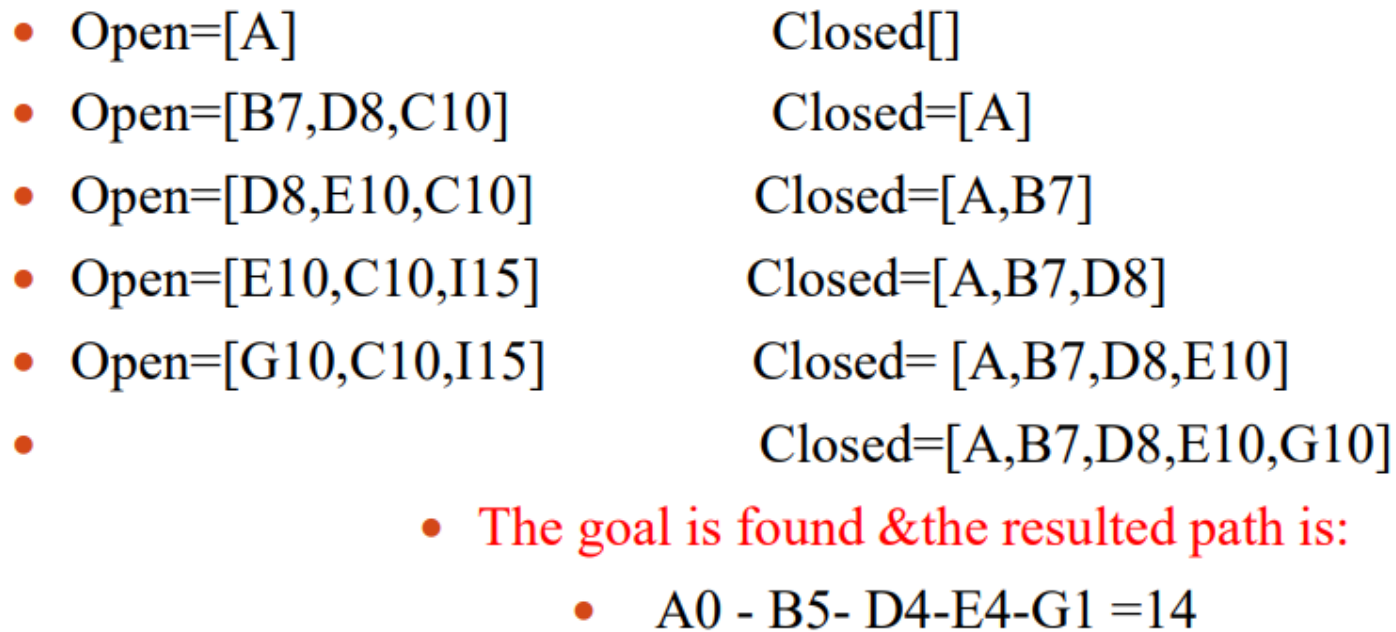
A*Algorithm

A* algorithm is simply define as a best first search plus specific function. This specific function represents the actual distance (levels) between the current state and the goal state and is denoted by $h(n)$. It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:

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

Since $g(n)$ gives the path cost from the start node to node n , and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have $f(n) =$ estimated cost of the cheapest solution through n .

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$. It turns out that this strategy is more than just reasonable: provided that the heuristic function $h(n)$ satisfies certain conditions, A* search is both complete and optimal.



Algorithm of A* search:

- **Step 1:** Place the starting node in the OPEN list.
- **Step 2:** Check if the OPEN list is empty or not. If the list is empty, then return failure and stops.
- **Step 3:** Select the node from the OPEN list which has the smallest value of the evaluation function ($g+h$). If node n is the goal node, then return success and stop, otherwise.
- **Step 4:** Expand node n and generate all of its successors, and put n into the closed list. For each successor n' , check whether n' is already in the OPEN or CLOSED list. If not, then compute the evaluation function for n' and place it into the Open list.
- **Step 5:** Else, if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest $g(n')$ value.
- **Step 6:** Return to Step 2.

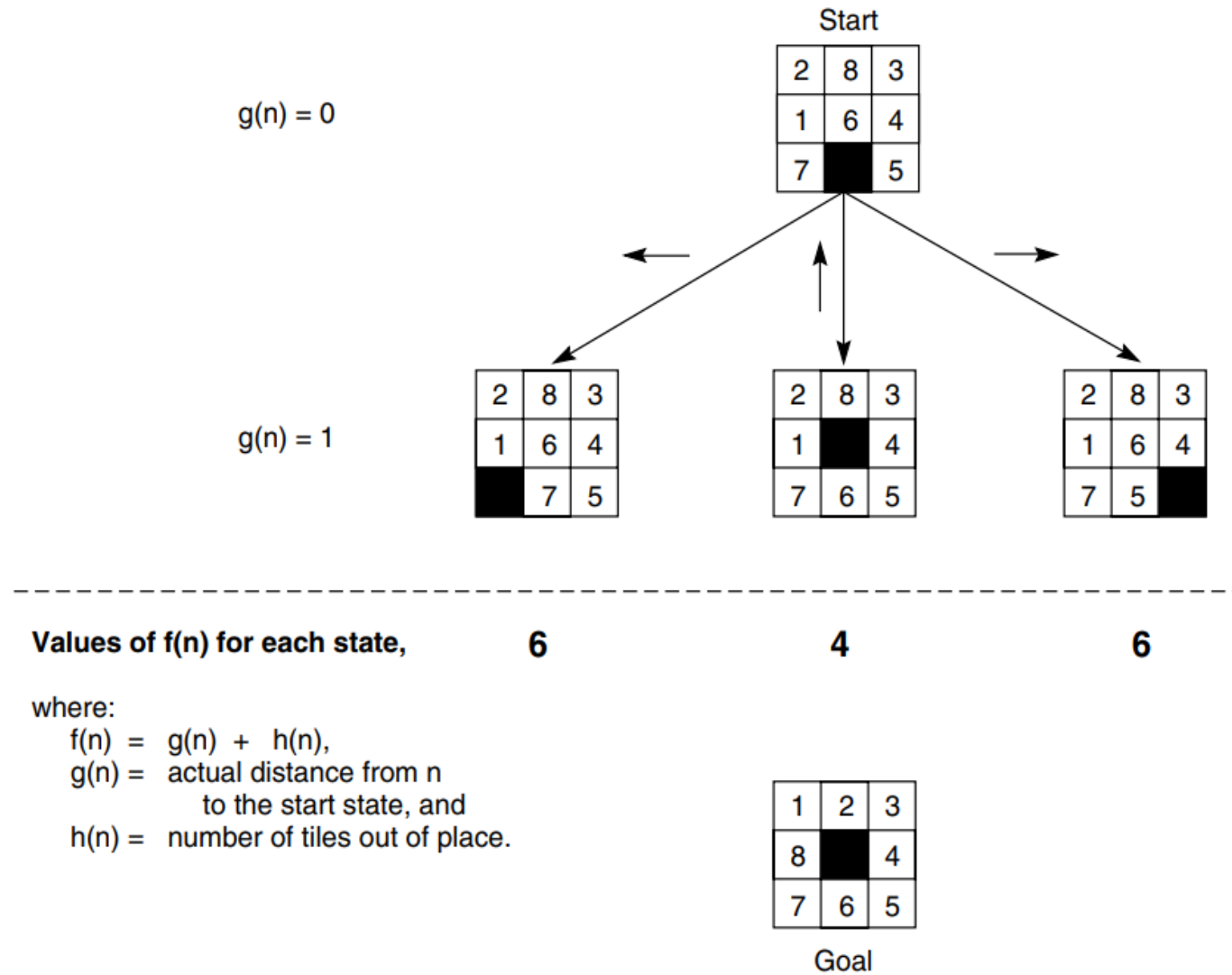


Figure 4.15 The heuristic f applied to states in the 8-puzzle.

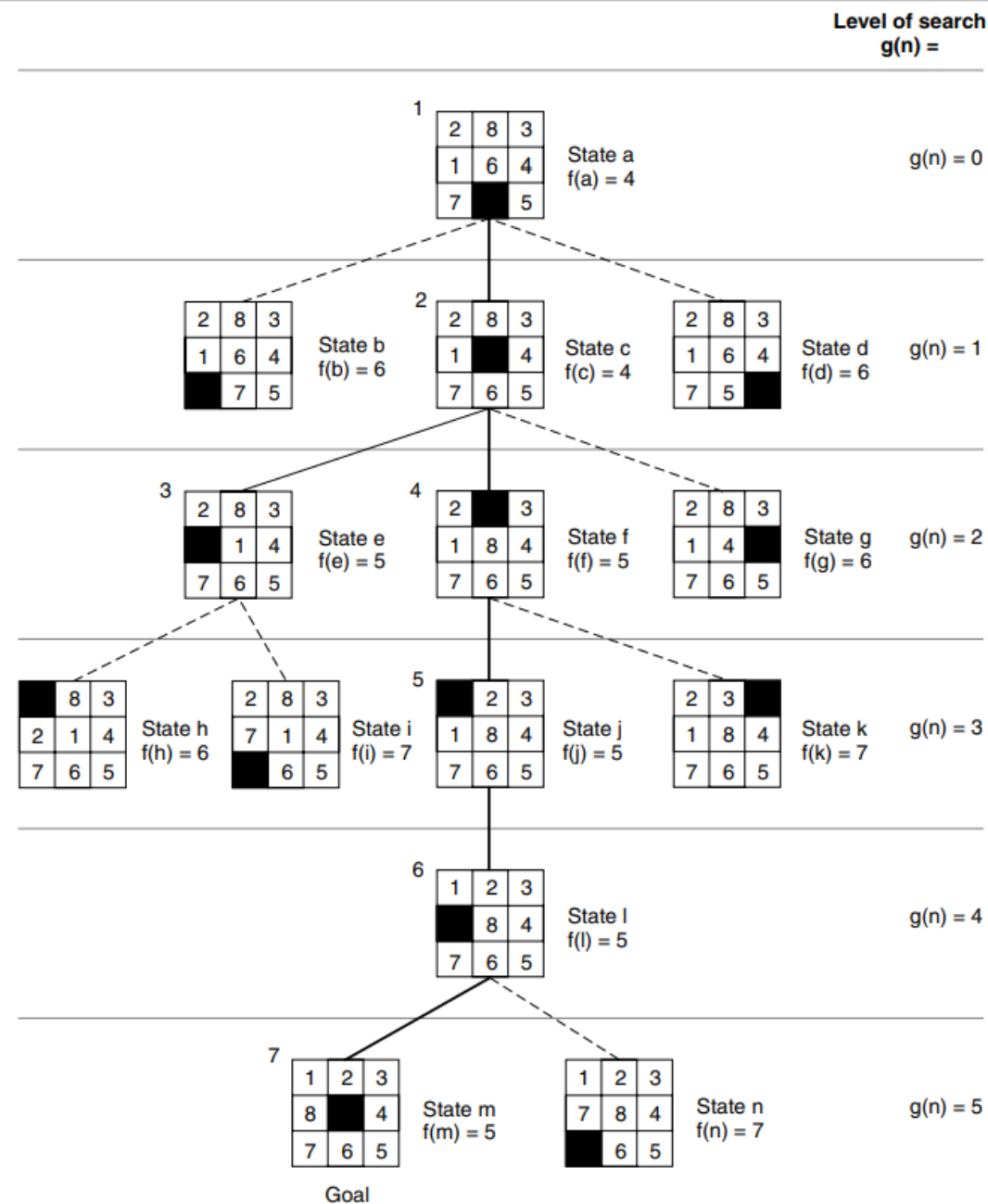
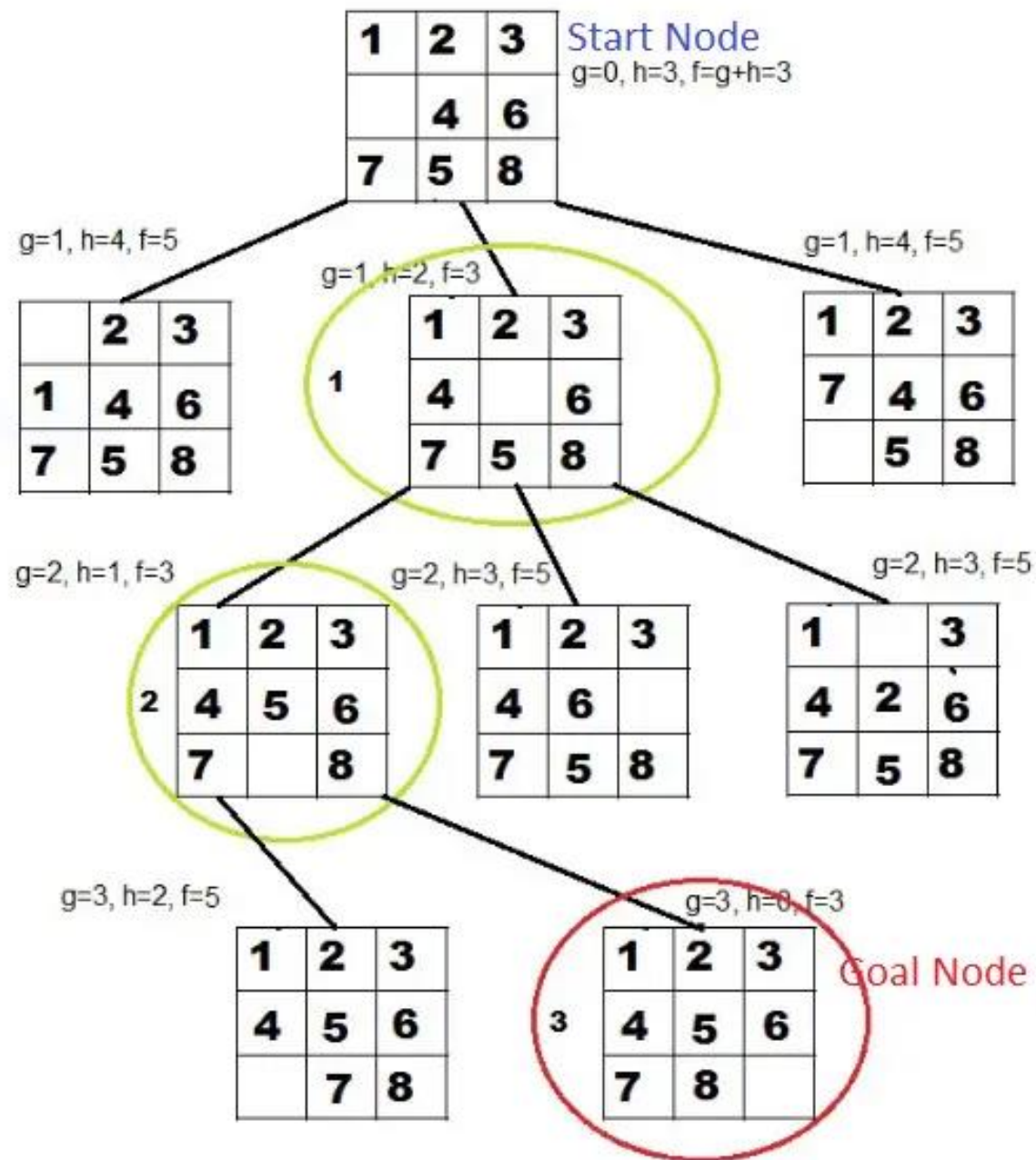


Figure 4.16 State space generated in heuristic search of the 8-puzzle graph.

The successive stages of open and closed that generate this graph are:

1. open = [a4];
closed = []
2. open = [c4, b6, d6];
closed = [a4]
3. open = [e5, f5, b6, d6, g6];
closed = [a4, c4]
4. open = [f5, h6, b6, d6, g6, i7];
closed = [a4, c4, e5]
5. open = [j5, h6, b6, d6, g6, k7, i7];
closed = [a4, c4, e5, f5]
6. open = [l5, h6, b6, d6, g6, k7, i7];
closed = [a4, c4, e5, f5, j5]
7. open = [m5, h6, b6, d6, g6, n7, k7, i7];
closed = [a4, c4, e5, f5, j5, l5]
8. success, m = goal!



Hill Climbing Algorithm

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.
- It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill Climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing:

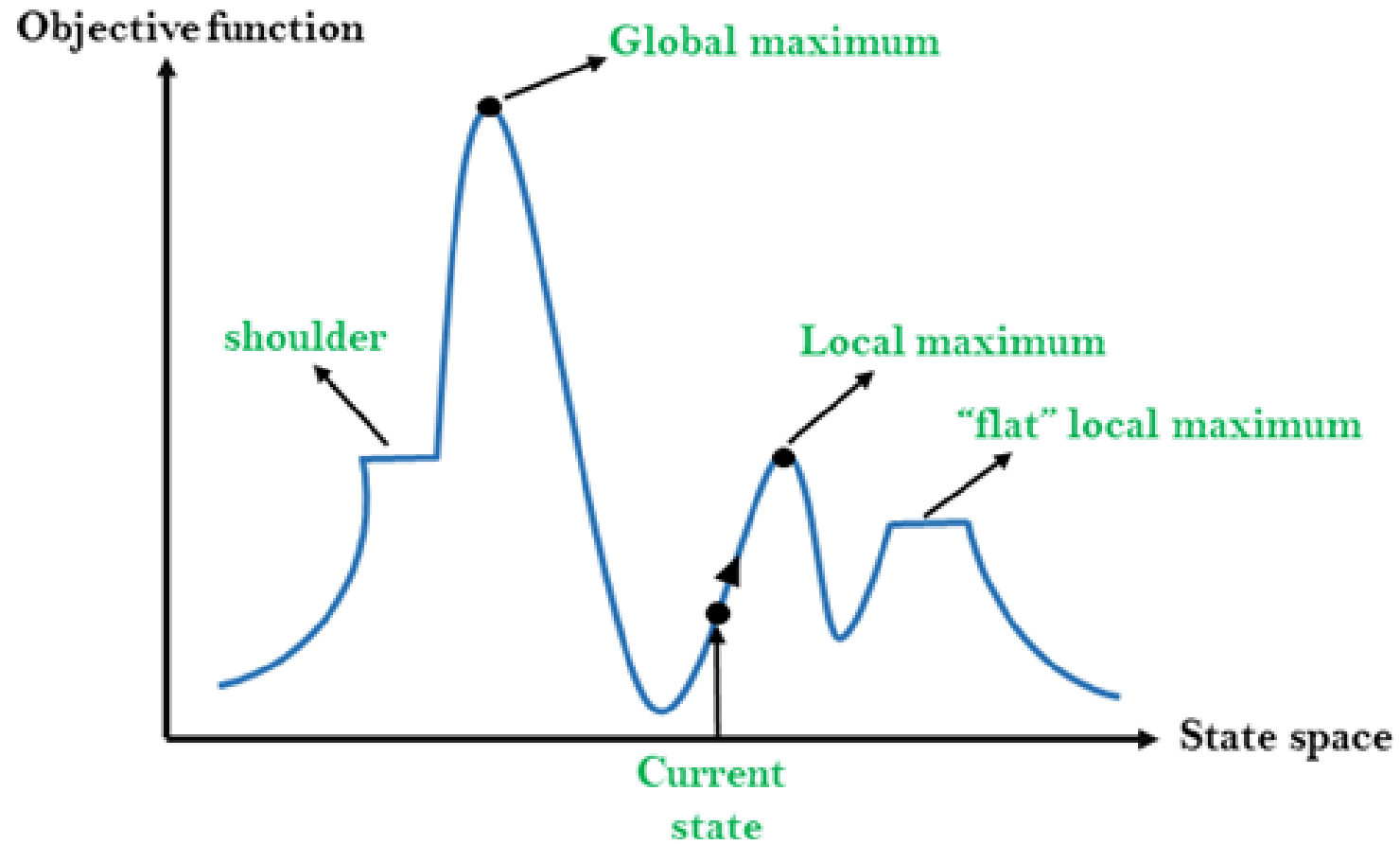
Following are some main features of Hill Climbing Algorithm:

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

Continue..

- The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.
- On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.

- State-space Diagram for Hill Climbing:



Different regions in the state space landscape

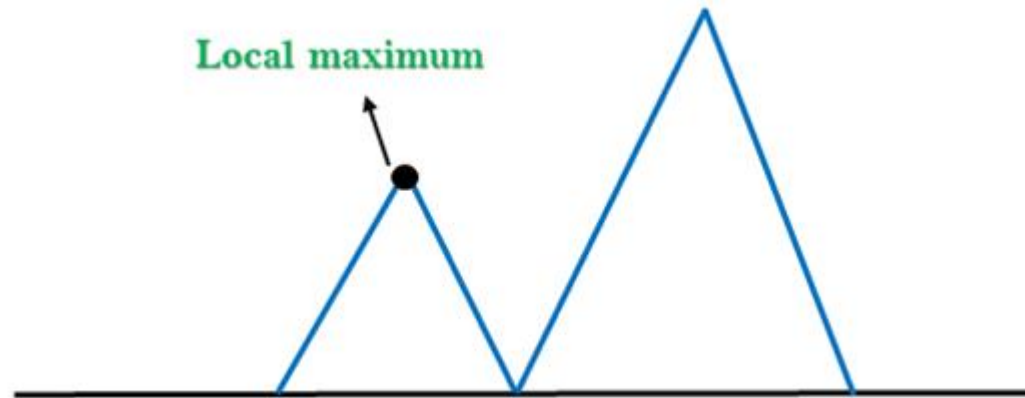
- **Local Maximum:** Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.
- **Global Maximum:** Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
- **Current state:** It is a state in a landscape diagram where an agent is currently present.
- **Plateau/Flat local maximum:** It is a flat space in the landscape where all the neighbor states of current states have the same value.
- **Shoulder:** It is a plateau region which has an uphill edge.

- Simple hill climbing is the simplest way to implement a hill climbing algorithm. **It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state.** It only checks its one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:
- Less time consuming
- Less optimal solution and the solution is not guaranteed
- **Algorithm for Simple Hill Climbing:**
- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
 - If it is goal state, then return success and quit.
 - Else if it is better than the current state then assign new state as a current state.
 - Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

Problems in Hill Climbing Algorithm:

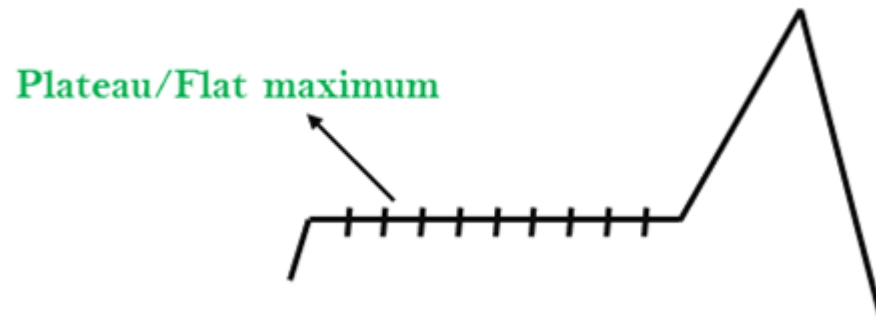
1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well



2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

- **Solution:** The solution for the plateau is to take big steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorithm could find non-plateau region.



3. Ridges: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

- **Solution:** With the use of bidirectional search, or by moving in different directions, we can improve this problem.

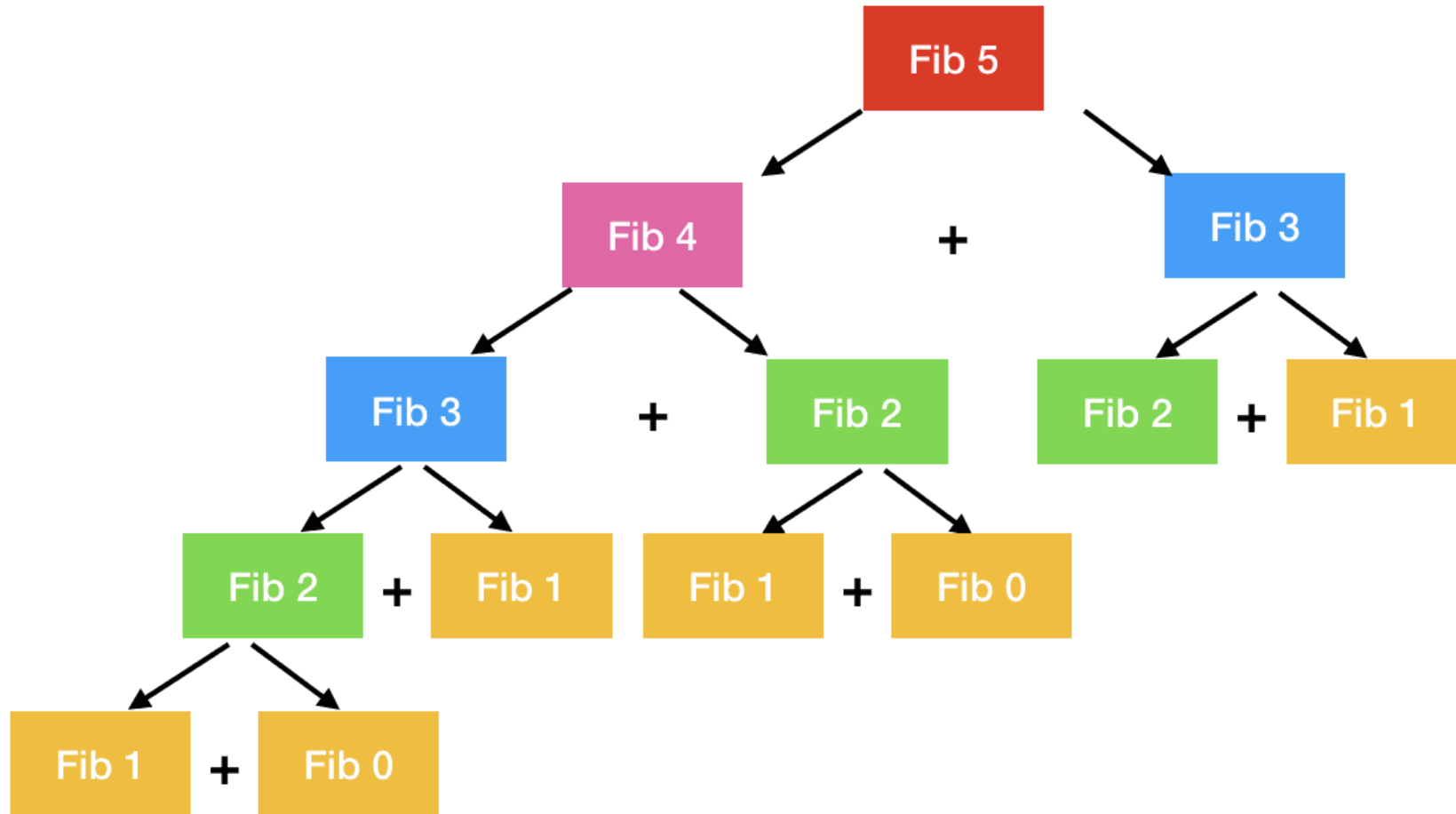
Dynamic Programming

- Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The result is an important algorithm often used for string matching, spell checking, and related areas in natural language processing

Memoization stores the result of expensive function calls (in arrays or objects) and returns the stored results whenever the same inputs occur again. In this way we can remember any values we have already calculated and access them instead of repeating the same calculation.

$$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$



	NULL	a	b	c	f	g
NULL	0 → 1 → 2 → 3 → 4 → 5					
a	↓ 1	0 → 1 → 2 → 3 → 4				
d	↓ 2	↓ 1	1 → 2 → 3 → 4			
c	↓ 3	↓ 2	↓ 2	1 → 2 → 3		
e	↓ 4	↓ 3	↓ 3	↓ 2	2 → 3	
g	↓ 5	↓ 4	↓ 4	↓ 3	↓ 3	2

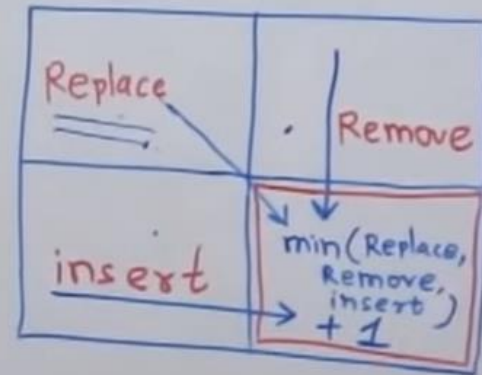
	NULL	a	b	c	f	g
NULL	0 → 1 → 2 → 3 → 4 → 5					
a	↓ 1	0 → 1 → 2 → 3 → 4				
d	↓ 2	↓ 1	1 → 2 → 3 → 4			
c	↓ 3	↓ 2	↓ 2	1 → 2 → 3		
e	↓ 4	↓ 3	↓ 3	↓ 2	2 → 3	
g	↓ 5	↓ 4	↓ 4	↓ 3	↓ 3	2

Hand pointing to a diagram showing a sequence of characters: ab c

adceg
a b c f g

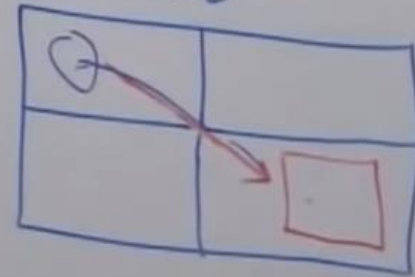
2
o/p

1. If $x \neq c$



2. If $x = c$

Just copy the diagonal element.



- Suppose we wanted to find the best possible alignment for the characters in the strings BAADDCABDDA and BBADCBA by using Minimum Edit distance

	—	B	A	A	D	D	C	A	B	D	D	A
—	0	1	2	3	4	5	6	7	8	9	10	11
B	1	0										
B	2											
A	3											
D	4											
C	5											
B	6											
A	7											

Figure 4.5 The initialization stage and first step in completing the array for character alignment using dynamic programming.

	—	B	A	A	D	D	C	A	B	D	D	A
—	0	1	2	3	4	5	6	7	8	9	10	11
B	1	0	1	2	3	4	5	6	7	8	9	10
B	2	1	2	3	4	5	6	7	6	7	8	9
A	3	2	1	2	3	4	5	6	7	8	9	8
D	4	3	2	3	2	3	4	5	6	7	8	9
C	5	4	3	4	3	4	3	4	5	6	7	8
B	6	5	6	5	4	5	4	5	4	5	6	7
A	7	6	5	4	5	6	5	4	5	6	7	6

Figure 4.6 The completed array reflecting the maximum alignment information for the strings.

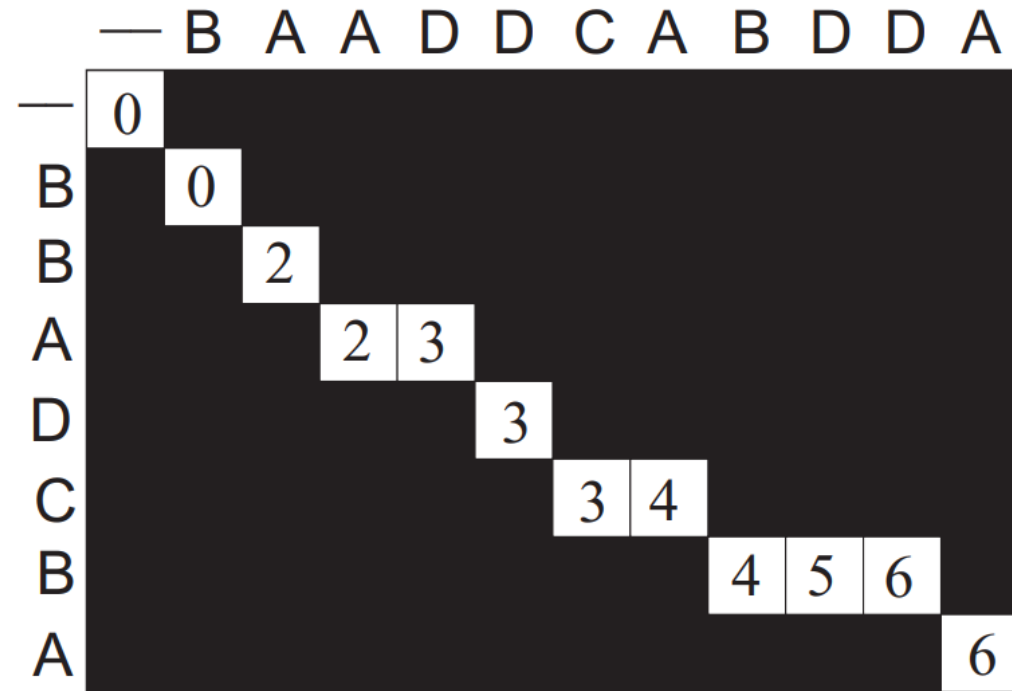


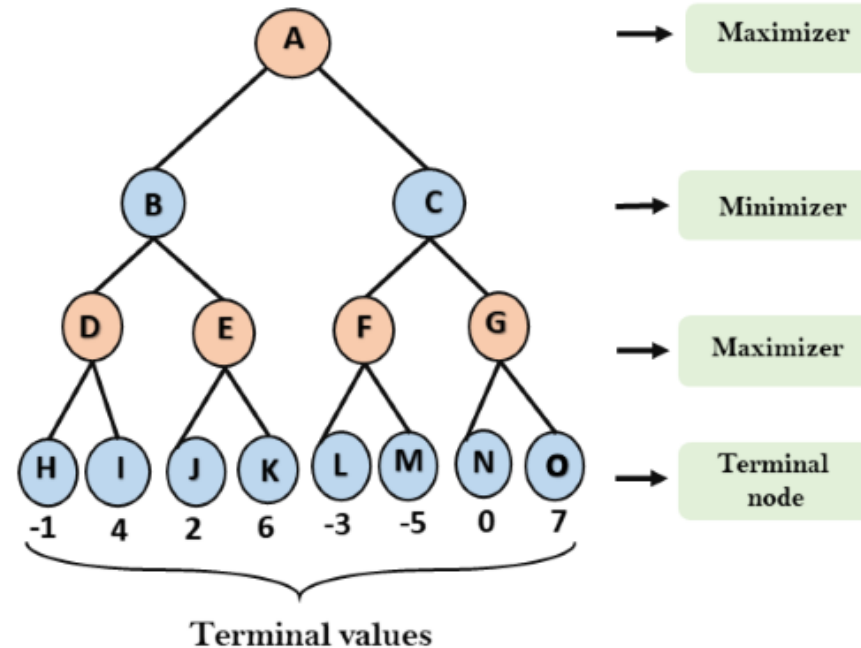
Figure 4.7 A completed backward component of the dynamic programming example giving one (of several possible) string alignments.

Adversarial Search

Working of Min-Max Algorithm:

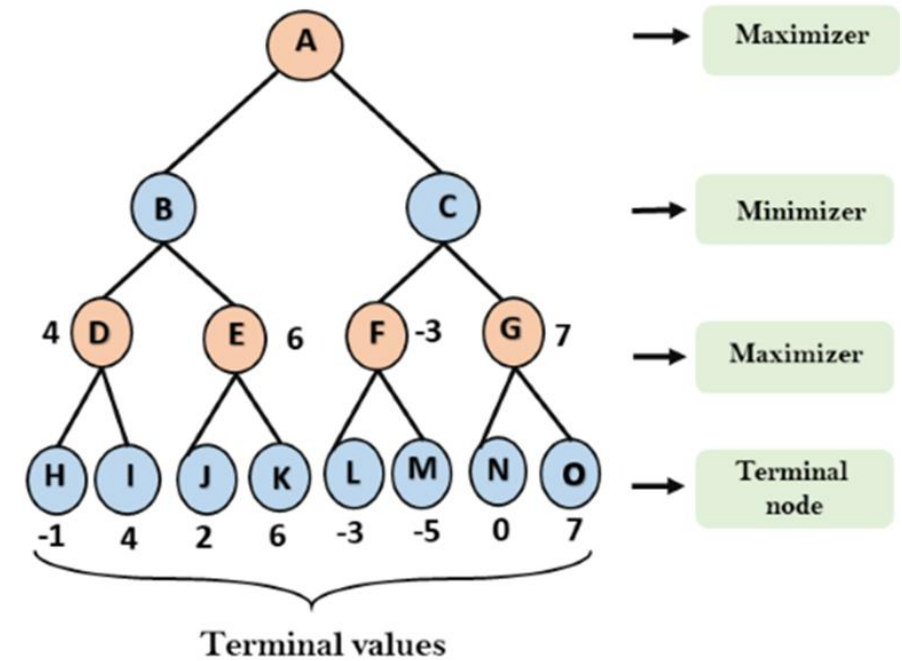
- The working of the minimax algorithm can be easily described using example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

- **Step-1:** In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value $= -\infty$, and minimizer will take next turn which has worst-case initial value $= +\infty$.

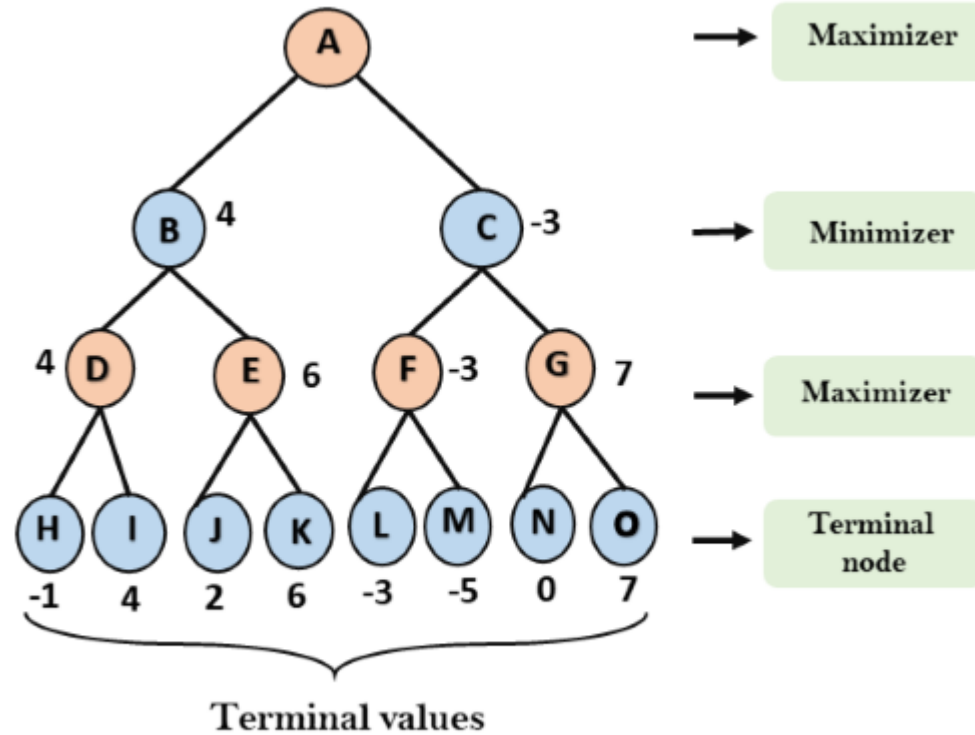


- **Step 2:** Now, first we find the utilities value for the Maximizer, its initial value is $-\infty$, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- For node D $\max(-1, -\infty) \Rightarrow \max(-1, 4) = 4$
- For Node E $\max(2, -\infty) \Rightarrow \max(2, 6) = 6$
- For Node F $\max(-3, -\infty) \Rightarrow \max(-3, -5) = -3$
- For node G $\max(0, -\infty) = \max(0, 7) = 7$

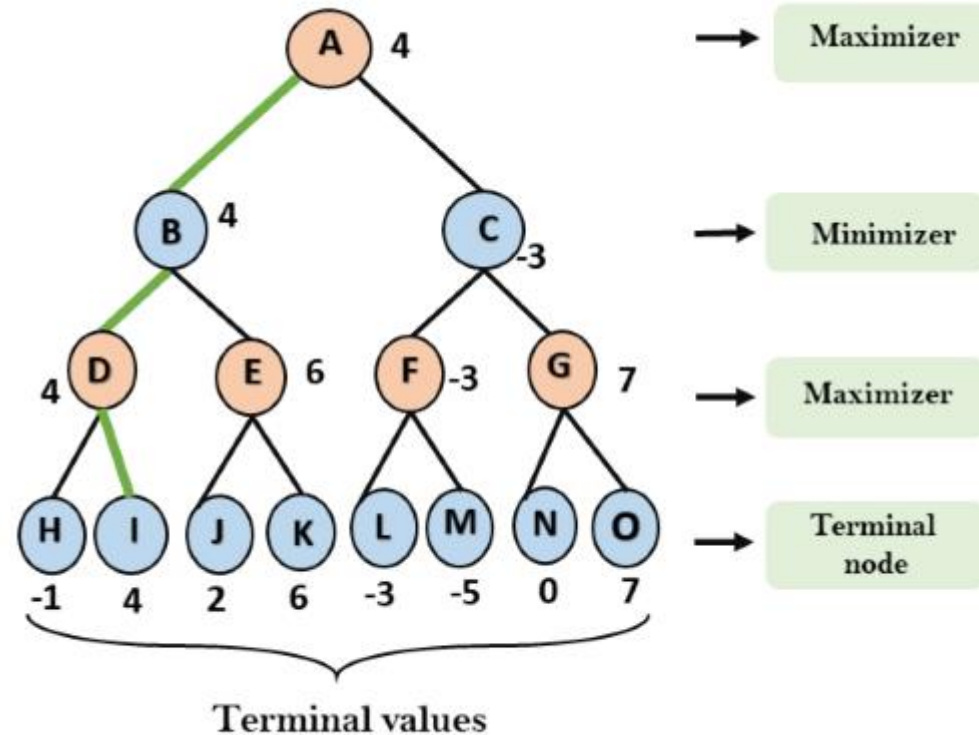


- Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with $+\infty$, and will find the 3rd layer node values.
- For node B = $\min(4, 6) = 4$
- For node C = $\min(-3, 7) = -3$



- Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

- For node A $\max(4, -3) = 4$



Alpha Beta Pruning

- <https://www.youtube.com/watch?v=l11QnZrnUjU>
- https://www.youtube.com/watch?v=z9_XYfYMI9Q
- Video is shared on WhatsApp group also

Propositional logic in Artificial intelligence

- Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and **logical connectives**.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of the propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called **tautology**, and it is also called a valid sentence.
- A proposition formula which is always false is called **Contradiction**.
- Statements which are questions, commands, or opinions are not propositions such as "**Where is Rohini**", "**How are you**", "**What is your name**", are not propositions.

- The syntax of propositional logic defines the allowable sentences for the **knowledge representation**. There are two types of Propositions:

1.Atomic Propositions

2.Compound propositions

- **Atomic Proposition:** Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false.
 - a) $2+2$ is 4, it is an atomic proposition as it is a **true** fact.
 - b) "The Sun is cold" is also a proposition as it is a **false** fact.
- **Compound proposition:** Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.
 - a) "It is raining today, and street is wet."
 - b) "Ankit is a doctor, and his clinic is in Mumbai."

- **Logical Connectives:**
- **Negation:** A sentence such as $\neg P$ is called negation of P. A literal can be either Positive literal or negative literal.
- **Conjunction:** A sentence which has \wedge connective such as, $P \wedge Q$ is called a conjunction.
 - Example: Rohan is intelligent and hardworking. It can be written as,
 - $P =$ Rohan is intelligent,
 - $Q =$ Rohan is hardworking. $\rightarrow P \wedge Q$.
- **Disjunction:** A sentence which has \vee connective, such as $P \vee Q$. is called disjunction, where P and Q are the propositions.
 - Example: "Ritika is a doctor or Engineer",
 - Here $P =$ Ritika is Doctor. $Q =$ Ritika is Doctor, so we can write it as $P \vee Q$.
- **Implication:** A sentence such as $P \rightarrow Q$, is called an implication. Implications are also known as if-then rules. It can be represented as
 - If it is raining, then the street is wet.
 - Let $P =$ It is raining, and $Q =$ Street is wet, so it is represented as $P \rightarrow Q$
- **Biconditional:** A sentence such as $P \Leftrightarrow Q$ is a Biconditional sentence, example If I am breathing, then I am alive
 - $P =$ I am breathing, $Q =$ I am alive, it can be represented as $P \Leftrightarrow Q$.

- Following is the summarized table for Propositional Logic Connectives:

Connective symbols	Word	Technical term	Example
\wedge	AND	Conjunction	$A \wedge B$
\vee	OR	Disjunction	$A \vee B$
\rightarrow	Implies	Implication	$A \rightarrow B$
\Leftrightarrow	If and only if	Biconditional	$A \Leftrightarrow B$
\neg or \sim	Not	Negation	$\neg A$ or $\neg B$

- Truth Table:

For Negation:

P	$\neg P$
True	False
False	True

For Conjunction:

P	Q	$P \wedge Q$
True	True	True
True	False	False
False	True	False
False	False	False

For disjunction:

P	Q	$P \vee Q$
True	True	True
False	True	True
True	False	True
False	False	False

For Implication:

P	Q	$P \rightarrow Q$
True	True	True
True	False	False
False	True	True
False	False	True

Logical equivalence:

- Logical equivalence is one of the features of propositional logic. Two propositions are said to be logically equivalent if and only if the columns in the truth table are identical to each other.
- Let's take two propositions A and B, so for logical equivalence, we can write it as $A \Leftrightarrow B$. In below truth table we can see that column for $\neg A \vee B$ and $A \rightarrow B$, are identical hence A is Equivalent to B

A	B	$\neg A$	$\neg A \vee B$	$A \rightarrow B$
T	T	F	T	T
T	F	F	F	F
F	T	T	T	T
F	F	T	T	T

Properties of Operators:

- **Commutativity:**

- $P \wedge Q = Q \wedge P$, or
- $P \vee Q = Q \vee P$.

- **Associativity:**

- $(P \wedge Q) \wedge R = P \wedge (Q \wedge R)$,
- $(P \vee Q) \vee R = P \vee (Q \vee R)$

- **Identity element:**

- $P \wedge \text{True} = P$,
- $P \vee \text{True} = \text{True}$.

- **Distributive:**

- $P \wedge (Q \vee R) = (P \wedge Q) \vee (P \wedge R)$.
- $P \vee (Q \wedge R) = (P \vee Q) \wedge (P \vee R)$.

- **DE Morgan's Law:**

- $\neg (P \wedge Q) = (\neg P) \vee (\neg Q)$
- $\neg (P \vee Q) = (\neg P) \wedge (\neg Q)$.

- **Double-negation elimination:**

- $\neg (\neg P) = P$.

- **Limitations of Propositional logic:**
- We cannot represent relations like ALL, some, or none with propositional logic. Example:
 - **All the girls are intelligent.**
 - **Some apples are sweet.**
- Propositional logic has limited expressive power.
- In propositional logic, we cannot describe statements in terms of their properties or logical relationships.

- **Inference:**

- In artificial intelligence, we need intelligent computers which can create new logic from old logic or by evidence, **so generating the conclusions from evidence and facts is termed as Inference.**

- **Inference rules:**

- Inference rules are the templates for generating valid arguments. Inference rules are applied to derive proofs in artificial intelligence, and the proof is a sequence of the conclusion that leads to the desired goal.
- In inference rules, the implication among all the connectives plays an important role. Following are some terminologies related to inference rules:
- **Implication:** It is one of the logical connectives which can be represented as $P \rightarrow Q$. It is a Boolean expression.
- **Converse:** The converse of implication, which means the right-hand side proposition goes to the left-hand side and vice-versa. It can be written as $Q \rightarrow P$.
- **Contrapositive:** The negation of converse is termed as contrapositive, and it can be represented as $\neg Q \rightarrow \neg P$.
- **Inverse:** The negation of implication is called inverse. It can be represented as $\neg P \rightarrow \neg Q$.

Types of Inference rules:

1. Modus Ponens:

- The Modus Ponens rule is one of the most important rules of inference, and it states that if P and $P \rightarrow Q$ is true, then we can infer that Q will be true. It can be represented as:

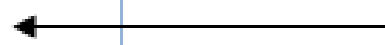
Notation for Modus ponens:
$$\frac{P \rightarrow Q, P}{\therefore Q}$$

Example:

- Statement-1: "If I am sleepy then I go to bed" $\Rightarrow P \rightarrow Q$
- Statement-2: "I am sleepy" $\Rightarrow P$
- Conclusion: "I go to bed." $\Rightarrow Q$.
- Hence, we can say that, if $P \rightarrow Q$ is true and P is true then Q will be true.

- Proof by Truth-Table:

P	Q	$P \rightarrow Q$
0	0	0
0	1	1
1	0	0
1	1	1



2. Modus Tollens:

- The Modus Tollens rule states that if $P \rightarrow Q$ is true and $\neg Q$ is true, then $\neg P$ will also be true. It can be represented as:

$$\text{Notation for Modus Tollens: } \frac{P \rightarrow Q, \neg Q}{\neg P}$$

- Statement-1: "If I am sleepy then I go to bed" $\Rightarrow P \rightarrow Q$
- Statement-2: "I do not go to the bed." $\Rightarrow \neg Q$
- Statement-3: Which infers that "I am not sleepy" $\Rightarrow \neg P$
- Proof by Truth-Table:

P	Q	$\neg P$	$\neg Q$	$P \rightarrow Q$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	0	0	1

3. Hypothetical Syllogism:

- The Hypothetical Syllogism rule states that if $P \rightarrow R$ is true whenever $P \rightarrow Q$ is true, and $Q \rightarrow R$ is true. It can be represented as the following notation:
- **Example:**
- **Statement-1:** If you have my home key then you can unlock my home. $P \rightarrow Q$
- **Statement-2:** If you can unlock my home then you can take my money. $Q \rightarrow R$
- **Conclusion:** If you have my home key then you can take my money. $P \rightarrow R$
- Proof by Truth-Table:

P	Q	R	$P \rightarrow Q$	$Q \rightarrow R$	$P \rightarrow R$	
0	0	0	1	1	1	←
0	0	1	1	1	1	←
0	1	0	1	0	1	
0	1	1	1	1	1	←
1	0	0	0	1	1	
1	0	1	0	1	1	
1	1	0	1	0	0	
1	1	1	1	1	1	←

4. Disjunctive Syllogism:

- The Disjunctive syllogism rule state that if $P \vee Q$ is true, and $\neg P$ is true, then Q will be true. It can be represented as:

$$\text{Notation of Disjunctive syllogism: } \frac{P \vee Q, \neg P}{Q}$$


Example:

Statement-1: Today is Sunday or Monday. $\implies P \vee Q$

Statement-2: Today is not Sunday. $\implies \neg P$

Conclusion: Today is Monday. $\implies Q$

P	Q	$\neg P$	$P \vee Q$
0	0	1	0
0	1	1	1
1	0	0	1
1	1	0	1



5. Addition:

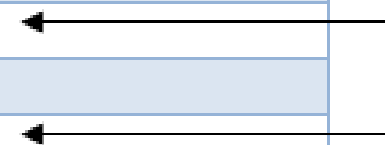
- The Addition rule is one the common inference rule, and it states that If P is true, then $P \vee Q$ will be true.

$$\text{Notation of Addition: } \frac{P}{P \vee Q}$$

Example:

- Statement: I have a vanilla ice-cream. $\Rightarrow P$
- Statement-2: I have Chocolate ice-cream.
- Conclusion: I have vanilla or chocolate ice-cream. $\Rightarrow (P \vee Q)$

P	Q	$P \vee Q$
0	0	0
1	0	1
0	1	1
1	1	1




6. Simplification:

- The simplification rule state that if $P \wedge Q$ is true, then Q or P will also be true. It can be represented as:

Notation of Simplification rule: $\frac{P \wedge Q}{Q}$ Or $\frac{P \wedge Q}{P}$

- Proof by Truth-Table:

P	Q	$P \wedge Q$
0	0	0
1	0	0
0	1	0
1	1	1

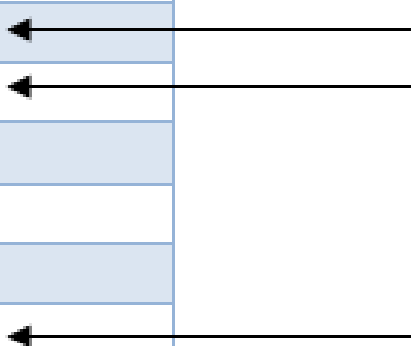


7. Resolution:

- The Resolution rule states that if $P \vee Q$ and $\neg P \wedge R$ is true, then $Q \vee R$ will also be true. It can be represented as

$$\text{Notation of Resolution} \frac{P \vee Q, \neg P \wedge R}{Q \vee R}$$

P	$\neg P$	Q	R	$P \vee Q$	$\neg P \wedge R$	$Q \vee R$
0	1	0	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	1	1	1
0	1	1	1	1	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	1
1	0	1	0	1	0	1
1	0	1	1	1	0	1



Predicate logic or First-order predicate logic

- In the topic of Propositional logic, we have seen that how to represent statements using propositional logic. But unfortunately, in propositional logic, we can only represent the facts, which are either true or false. PL is not sufficient to represent the complex sentences or natural language statements. The propositional logic has very limited expressive power. Consider the following sentence, which we cannot represent using PL logic.
- **"Some humans are intelligent", or**
- **"Sachin likes cricket."**
- To represent the above statements, PL logic is not sufficient, so we required some more powerful logic, such as first-order logic.

- First-order logic is another way of **knowledge representation** in artificial intelligence. It is an extension to propositional logic
- First-order logic is also known as **Predicate logic or First-order predicate logic**. First-order logic is a powerful language that develops information about the objects in a more easy way and can also express the relationship between those objects.

First-order logic also has two main parts:

1.Syntax

2.Semantics

Syntax of First-Order logic:

- The syntax of FOL determines which collection of symbols is a logical expression in first-order logic. The basic syntactic elements of first-order logic are symbols. We write statements in short-hand notation in FOL.
- Basic Elements of First-order logic:

Constant	1, 2, A, John, Mumbai, cat,....
Variables	x, y, z, a, b,....
Predicates	Brother, Father, >,....
Function	sqrt, LeftLegOf,
Connectives	\wedge , \vee , \neg , \Rightarrow , \Leftrightarrow
Equality	$=$
Quantifier	\forall , \exists

Atomic sentences:

- Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as **Predicate (term1, term2,, term n)**.

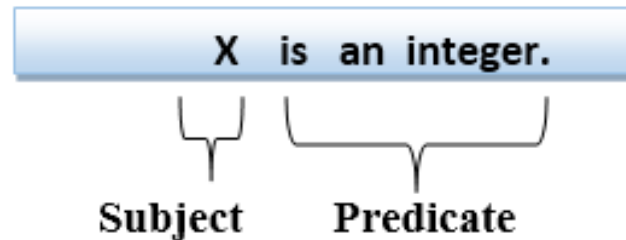
Example: Ravi and Ajay are brothers: => Brothers(Ravi, Ajay).
 Chinky is a cat: => cat (Chinky).

Complex Sentences:

- Complex sentences are made by combining atomic sentences using connectives.

First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.
- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.
- **Consider the statement: "x is an integer."**, it consists of two parts, the first part x is the subject of the statement and second part "is an integer," is known as a predicate.



- **Quantifiers in First-order logic:**
- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
- These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. There are two types of quantifier:
 - **Universal Quantifier, (for all, everyone, everything)**
 - **Existential quantifier, (for some, at least one).**

- **Universal Quantifier:**

- Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.
- The Universal quantifier is represented by a symbol \forall , which resembles an inverted A.
- In universal quantifier we use implication " \rightarrow ".
- If x is a variable, then $\forall x$ is read as:
 - For all x
 - For each x
 - For every x .

Example :All man drink coffee.

- $\forall x \text{ man}(x) \rightarrow \text{drink}(x, \text{coffee})$.
- It will be read as: There are all x where x is a man who drink coffee.

- **Existential Quantifier:**

- Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.
- It is denoted by the logical operator \exists , which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.
- In Existential quantifier we always use AND or Conjunction symbol (\wedge).
- If x is a variable, then existential quantifier will be $\exists x$ or $\exists(x)$. And it will be read as:
 - **There exists a 'x.'**
 - **For some 'x.'**
 - **For at least one 'x.'**

Example: Some boys are intelligent.

- $\exists x: \text{boys}(x) \wedge \text{intelligent}(x)$
- It will be read as: There are some x where x is a boy who is intelligent.

Some Examples of FOL using quantifier:

1. All birds fly.

In this question the predicate is "fly(bird)."

And since there are all birds who fly so it will be represented as follows.

$$\forall x \text{ bird}(x) \rightarrow \text{fly}(x).$$

Read as: There are all x where x is bird who fly

2. Every man respects his parent.

In this question, the predicate is "respect(x, y)," where x=man, and y= parent.

Since there is every man so will use \forall , and it will be represented as follows:

$$\forall x \text{ man}(x) \rightarrow \text{respects}(x, \text{parent}).$$

Read as: There are all x where x is man who respect his parent

3. Some boys play cricket.

In this question, the predicate is "play(x, y)," where x= boys, and y= game. Since there are some boys so we will use \exists , and it will be represented as:

$$\exists x \text{ boys}(x) \rightarrow \text{play}(x, \text{cricket}).$$

4. Not all students like both Mathematics and Science.

In this question, the predicate is "like(x, y)," where x= student, and y= subject. Since there are not all students, so we will use \forall with negation, so following representation for this:

$$\neg \forall (x) [\text{student}(x) \rightarrow \text{like}(x, \text{Mathematics}) \wedge \text{like}(x, \text{Science})].$$

There are all x where x is student who don't like Mathematics and Science

5. Only one student failed in Mathematics.

In this question, the predicate is "failed(x, y)," where x= student, and y= subject.

Since there is only one student who failed in Mathematics, so we will use following representation for this:

$$\exists (x) [\text{student}(x) \rightarrow \text{failed}(x, \text{Mathematics}) \wedge \forall (y) [\neg (x=y) \wedge \text{student}(y) \rightarrow \neg \text{failed}(y, \text{Mathematics})].$$



Some Examples of FOL

1. Marcus was a man.
2. Marcus was a Pompeian.
3. All Pompeians were Romans.
4. Caesar was a ruler.
5. All Pompeians were either loyal to Caesar or hated him.
6. Every one is loyal to someone.
7. People only try to assassinate rulers they are not loyal to.
8. Marcus tried to assassinate Caesar.





Some Examples of FOL

- Marcus was a man.

$\text{man}(\text{Marcus})$

- Marcus was a Pompeian.

$\text{Pompeian}(\text{Marcus})$

- All Pompeians were Romans.

$\forall x: \text{Pompeian}(x) \rightarrow \text{Roman}(x)$

- Caesar was a ruler.

$\text{ruler}(\text{Caesar})$





Some Examples of FOL

- All Pompeians were either loyal to Caesar or hated him.

inclusive-or

$$\forall x: \text{Roman}(x) \rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar})$$

exclusive-or

$$\begin{aligned} \forall x: \text{Roman}(x) \rightarrow & (\text{loyalto}(x, \text{Caesar}) \wedge \neg \text{hate}(x, \text{Caesar})) \vee \\ & (\neg \text{loyalto}(x, \text{Caesar}) \wedge \text{hate}(x, \text{Caesar})) \end{aligned}$$





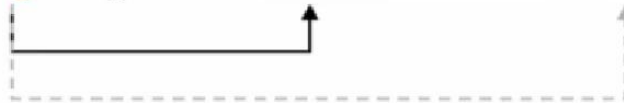
Some Examples of FOL

Every one is loyal to someone.

$$\forall x: \exists y: \text{loyalto}(x, y)$$

$$\exists y: \forall x: \text{loyalto}(x, y)$$

- People **only** try to assassinate rulers they are not loyal to.



$$\forall x: \forall y: \text{person}(x) \wedge \text{ruler}(y) \wedge \text{tryassassinate}(x, y) \\ \rightarrow \neg \text{loyalto}(x, y)$$



Some Examples of FOL

- Marcus tried to assassinate Caesar.
`tryassassinate(Marcus, Caesar)`
- Was Marcus loyal to Caesar?
`man(Marcus)`
`ruler(Caesar)`
`tryassassinate(Marcus, Caesar)`
 $\Downarrow \quad \forall x: \text{man}(x) \rightarrow \text{person}(x)$
`¬loyalto(Marcus, Caesar)`