SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Unit 2

Dr. Kayalvizhi Jayavel
&
Dr. N. Arunachalam

Search

- Searching is the universal technique of problem solving in Artificial Intelligence
- A search problem consists of:
  - **A State Space.** Set of all possible states where you can be.
  - **A Start State.** The state from where the search begins.
  - **A Goal Test.** A function that looks at the current state returns whether or not it is the goal state.
  - The **Solution** to a search problem is a sequence of actions, called the **plan** that transforms the start state to the goal state.

Search Problem Components

- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the available actions to the agent.
- **Transition model:** A description of what each action do, can be represented as a transition model.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

General Search Algorithm

Generic Search Algorithm

1. Initialize the search tree using the initial state of the problem
2. Repeat
   (a) If no candidate nodes can be expanded, return failure
   (b) Choose a leaf node for expansion, according to some search strategy
   (c) If the node contains a goal state, return the corresponding path
   (d) Otherwise expand the node by:
   - Applying each operator
   - Generating the successor state
   - Adding the resulting nodes to the tree

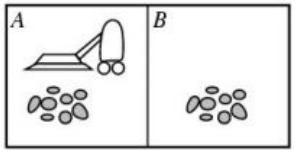
Examples of Search Algorithm

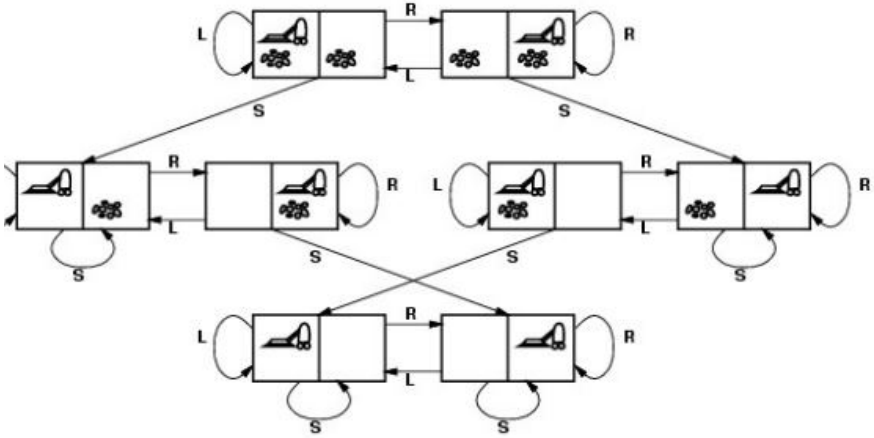
- TSP
- 8 Puzzle problem
- Tic Tac Toe
- N-queen problem
- Tower of Hanoi
- Water Jug Problem
- Blocks World
- Vacuum Cleaner

Vacuum Cleaner

Example: Vacuum-Cleaner

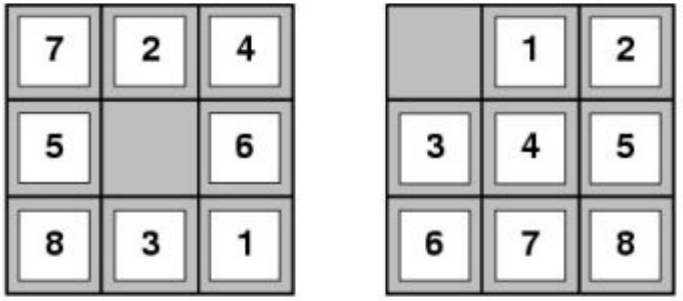
- States
  - 8 states
- Initial state
  - any state
- Actions
  - Left, Right, and Suck
- Transition model
  - complete state space, see next page
- Goal test
  - whether both squares are clean
- Path cost
  - each step costs 1





8 Puzzle Problem

Example: 8-puzzle



Start State

Goal State

NP-Complete

- States:
  - location of each tile and the blank
- Initial state: any, 9!/2
- Actions:
  - blank moves Left, Right, Up or Down
- Transition model:
  - Given a state and action, returns the resulting state
- Goal test: Goal configuration
- Path cost: Each step costs 1

Parameters for Search Evaluation

- **Completeness**: Is the algorithm guaranteed to find a solution if there exist one?
- **Optimality**: Does the algorithm find the optimal solution?
- **Time complexity**: How long does it take for the algorithm to find a solution?
- **Space complexity**: How much memory is consumed in finding the solution?

Uninformed and Heuristic Search Strategies

- Based on the information about the problem available for the search strategies, the search algorithms are classified into uninformed (Blind) and informed (or heuristic) search algorithms.
- For the **uninformed search algorithms**, the strategies have no additional information about the states beyond that provided by the problem definition. Generate successors and differentiate between goal and non-goal states.
- Strategies that know whether one non-goal state is more promising than the other is called **informed search strategies** or **heuristic search strategies**.

Uninformed 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.

Uninformed Search methods

Uninformed search strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

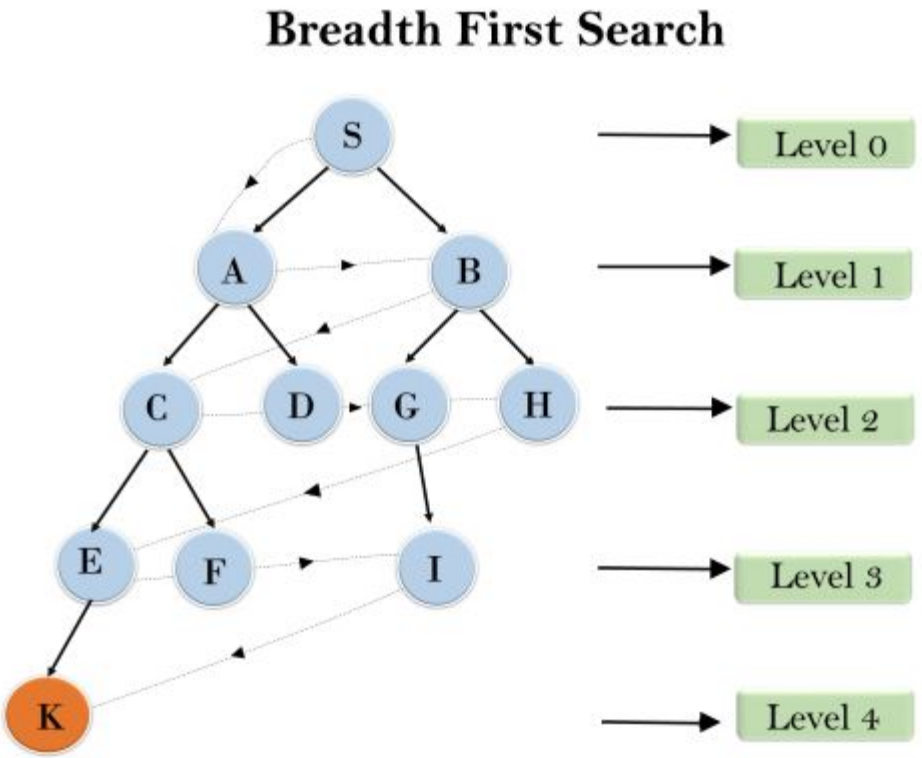
Depth-limited search

Iterative deepening search

BFS

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

BFS - Example


Breadth First Search
S
A
B
C
D
G
H
E
F
I
K
Level 0
Level 1
Level 2
Level 3
Level 4

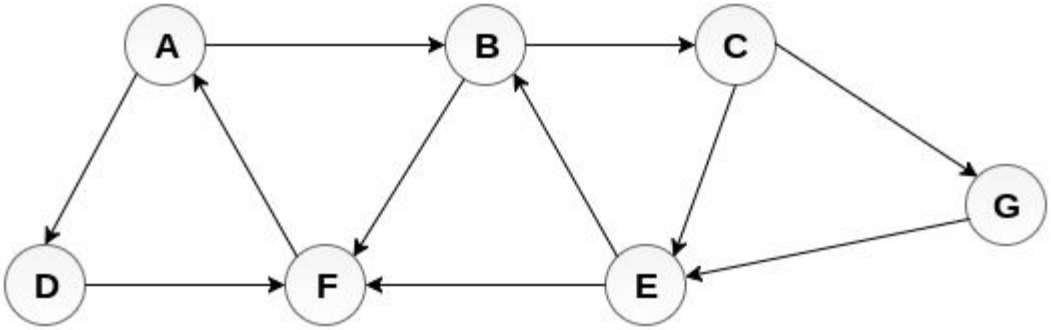
BFS Algorithm

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Child Node

```
function CHILD-NODE(problem, parent, action) returns a node
    return a node with
        STATE = problem.RESULT(parent.STATE, action)
        PARENT = parent
        ACTION = action
        PATH-COST = parent.PATH-COST +
                    problem.STEP-COST(parent.STATE, action)
```

BFS - Working



BFS - Implementation

1. Add A to QUEUE1 and NULL to QUEUE2.

```
QUEUE1 = {A}
QUEUE2 = {NULL}
```

2. Delete the Node A from QUEUE1 and insert all its neighbours. Insert Node A into QUEUE2

```
QUEUE1 = {B, D}
QUEUE2 = {A}
```

3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

```
QUEUE1 = {D, C, F}
QUEUE2 = {A, B}
```

4. Delete the node D from QUEUE1 and insert all its neighbours. Since F is the only neighbour of it which has been inserted, we will not insert it again. Insert node D into QUEUE2.

```
QUEUE1 = {C, F}
QUEUE2 = { A, B, D}
```

BFS - Implementation

5. Delete the node C from QUEUE1 and insert all its neighbours. Add node C to QUEUE2.

```
QUEUE1 = {F, E, G}
QUEUE2 = {A, B, D, C}
```

6. Remove F from QUEUE1 and add all its neighbours. Since all of its neighbours has already been added, we will not add them again. Add node F to QUEUE2.

```
QUEUE1 = {E, G}
QUEUE2 = {A, B, D, C, F}
```

7. Remove E from QUEUE1, all of E's neighbours has already been added to QUEUE1 therefore we will not add them again. All the nodes are visited and the target node i.e. E is encountered into QUEUE2.

```
QUEUE1 = {G}
QUEUE2 = {A, B, D, C, F,  E}
```

Now, backtrack from E to A, using the nodes available in QUEUE2.

The minimum path will be **A → B → C → E.**

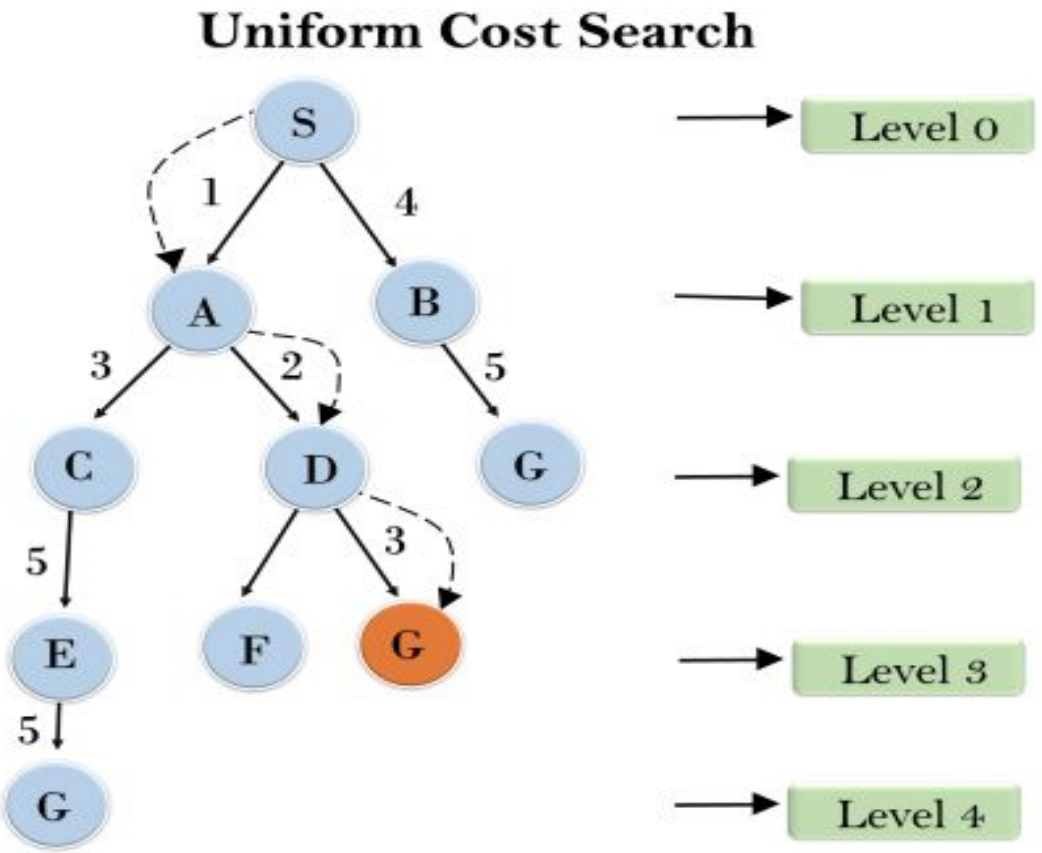
BFS – Time and Space

- **Time Complexity:** Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.
- **$T(b) = 1+b^2+b^3+.....+ b^d= O(b^d)$**
- **Space Complexity:** Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.
- **Completeness:** BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.
- **Optimality:** BFS is optimal if path cost is a non-decreasing function of the depth of the node.

Uniform Cost search

- Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph.
- This algorithm comes into play when a different cost is available for each edge.
- The primary goal of the uniform-cost search is to find a path to the goal node which has the **lowest cumulative cost**. Uniform-cost search expands nodes according to their path costs form the root node.
- It can be used to solve any graph/tree where the optimal cost is in demand.
- A uniform-cost search algorithm is implemented by the **priority queue.**
- It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

Uniform Cost search - Example

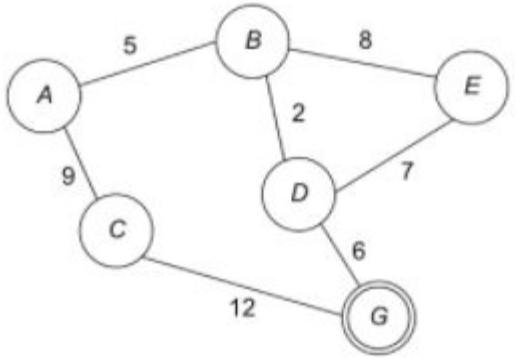


Uniform Cost search - Algorithm

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure

  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Uniform Cost search- Working

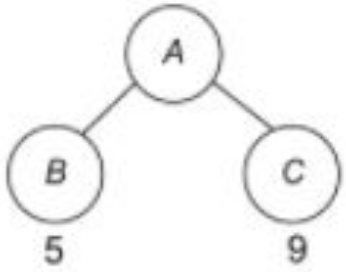


Uniform Cost search - Implementatio

Step 1: Start from *A* — Expanded none

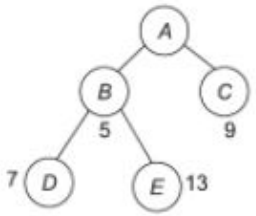
Step 2: Frontier — Expanded none

Step 3: Frontier: *B* and *C* — Expanded *A*



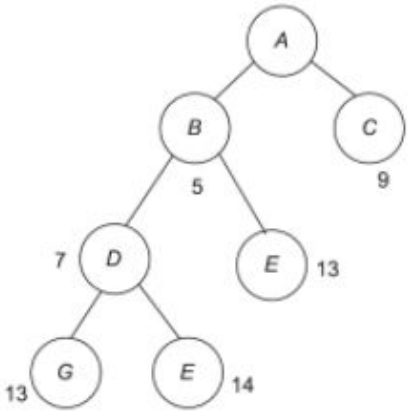
Uniform Cost search - Implementation

Step 4: Frontier: *D*, *C*, *E* (priority queue) Expanded *A*, *B*



Note: Remember the costs are added from the root till the node in consideration. Select the lowest, i.e., *D*.

Step 5: Frontier (*C*, *E*, *E*, *G*) Expanded *A*, *B*, *D*



Uniform Cost search- Time and space

Completeness:

- YES, if step-cost > ε (smal positive constant)

Time complexity:

- Assume C* the cost of the optimal solution.
- Assume that every action costs at least ε
- Worst-case: $O(b^{C^*/\varepsilon})$

Space complexity:

- Idem to time complexity

Optimality:

- nodes expanded in order of increasing path cost.
- YES, if complete.

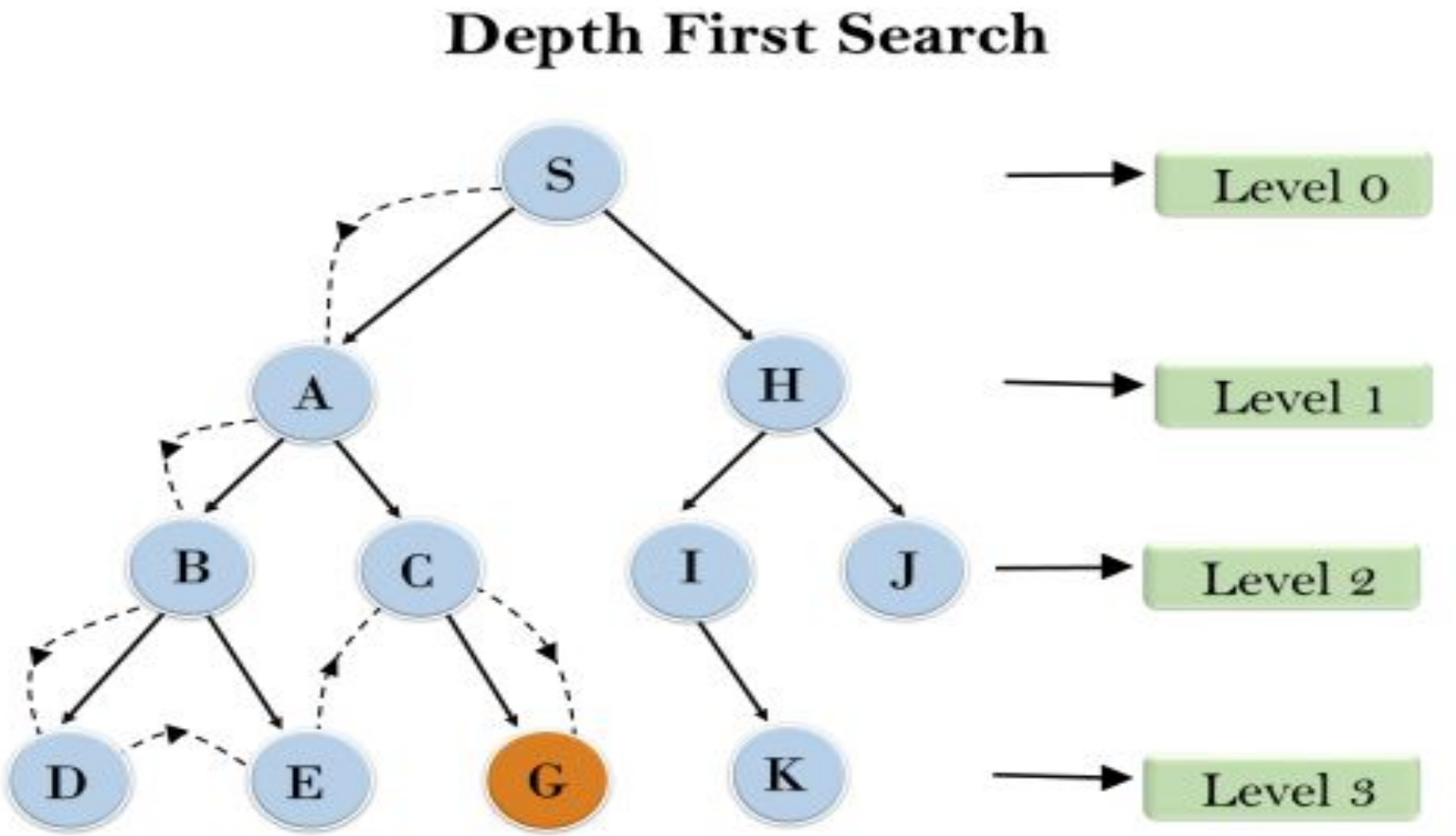
DFS

- Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures.
- The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

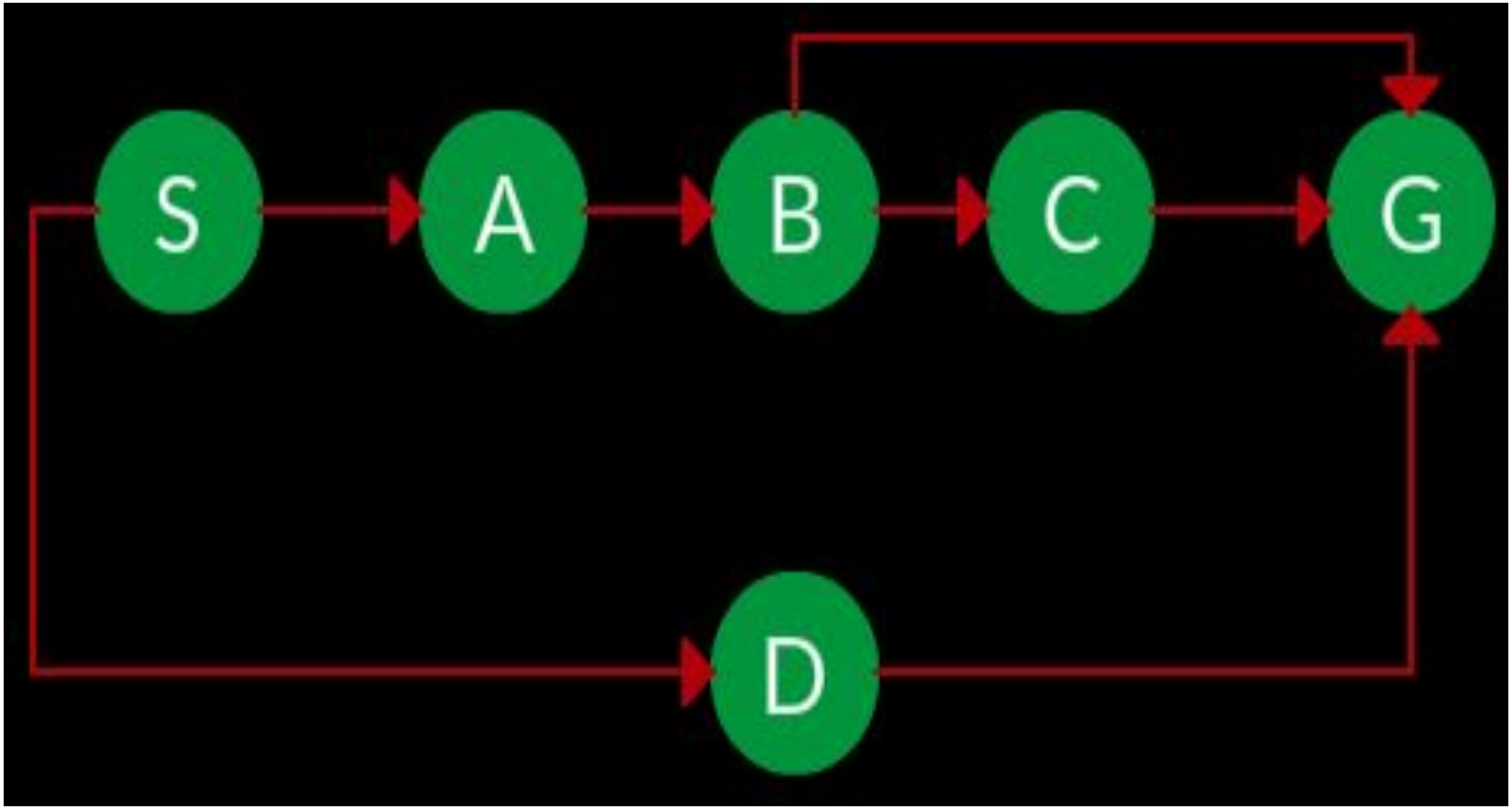
DFS

- Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until the **goal node** or the node which has no children.
- The algorithm, then **backtracks** from the dead end towards the most recent node that is yet to be completely unexplored.
- The data structure which is being used in DFS is stack. In DFS, the edges that leads to an unvisited node are called **discovery edges** while the edges that leads to an already visited node are called **block edges**.

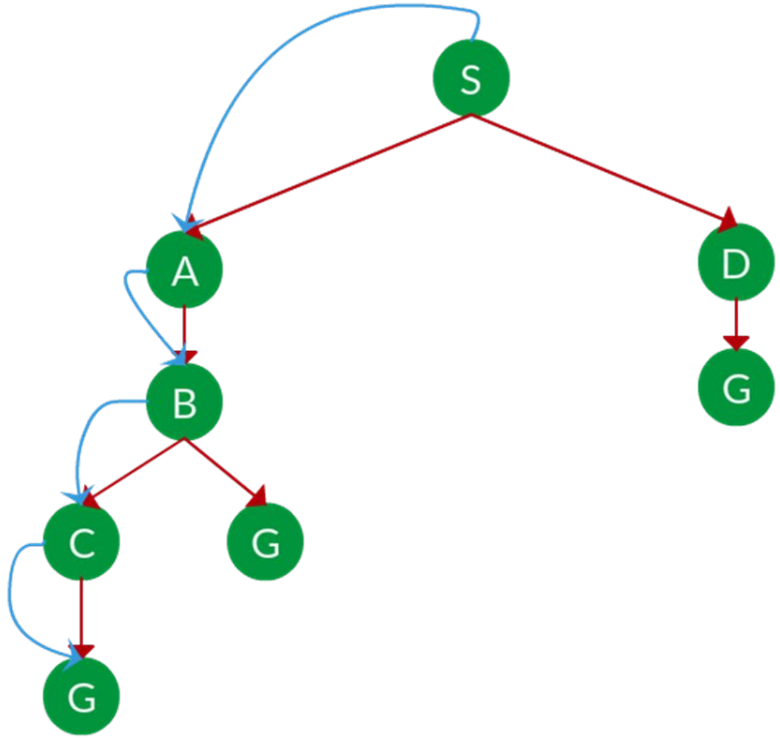
DFS - Example


Depth First Search
S
A
H
B
C
I
J
D
E
G
K
Level 0
Level 1
Level 2
Level 3

DFS - Graph



- ***DFS SEARCH***



Path: S -> A -> B -> C -> G

DFS – Working Principle

- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty.

- Ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once.
- If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

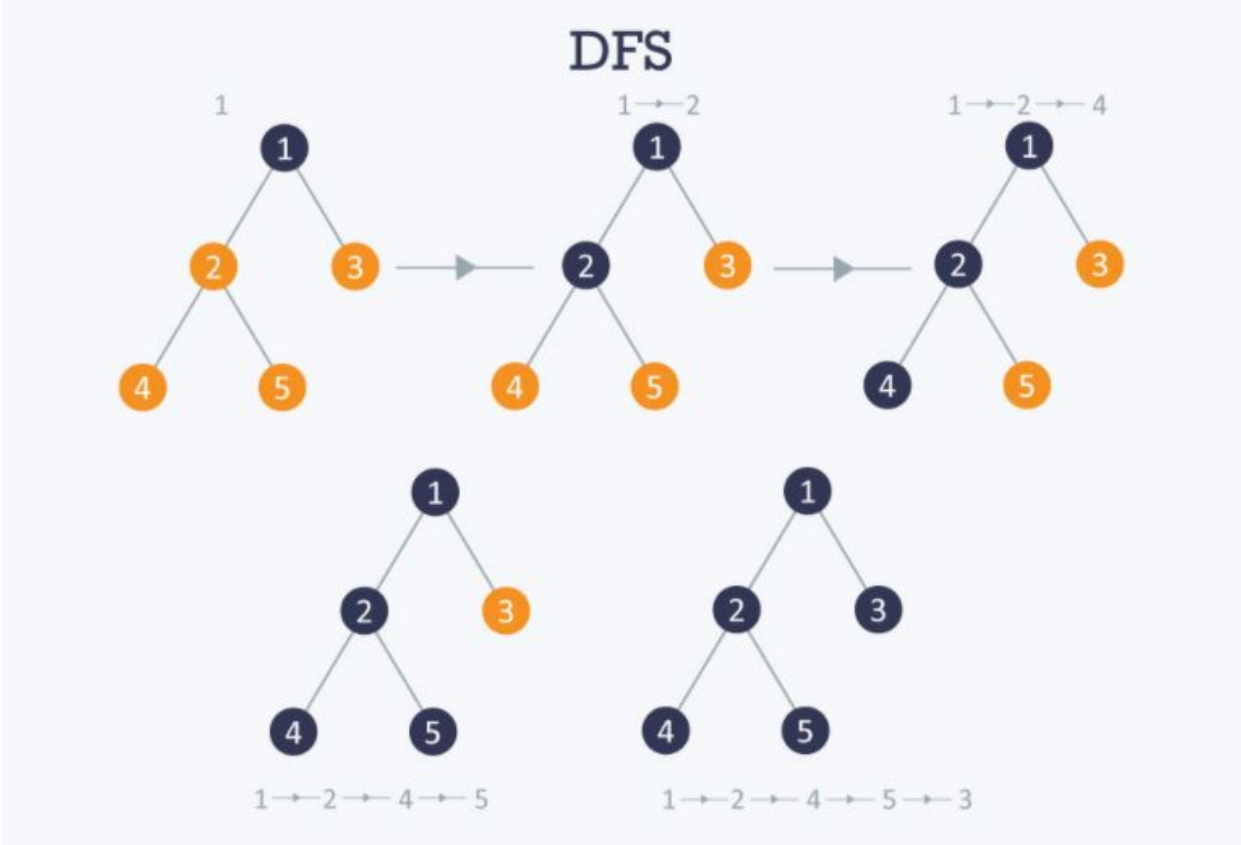
DFS - Iterative

```
    DFS-iterative (G, s):                                          //Where G is graph
source vertex
       let S be stack
       S.push( s )              //Inserting s in stack
       mark s as visited.
       while ( S is not empty):
           //Pop a vertex from stack to visit next
           v  =  S.top( )
          S.pop( )
          //Push all the neighbours of v in stack that are not visited
         for all neighbours w of v in Graph G:
             if w is not visited :
                       S.push( w )
                      mark w as visited
```

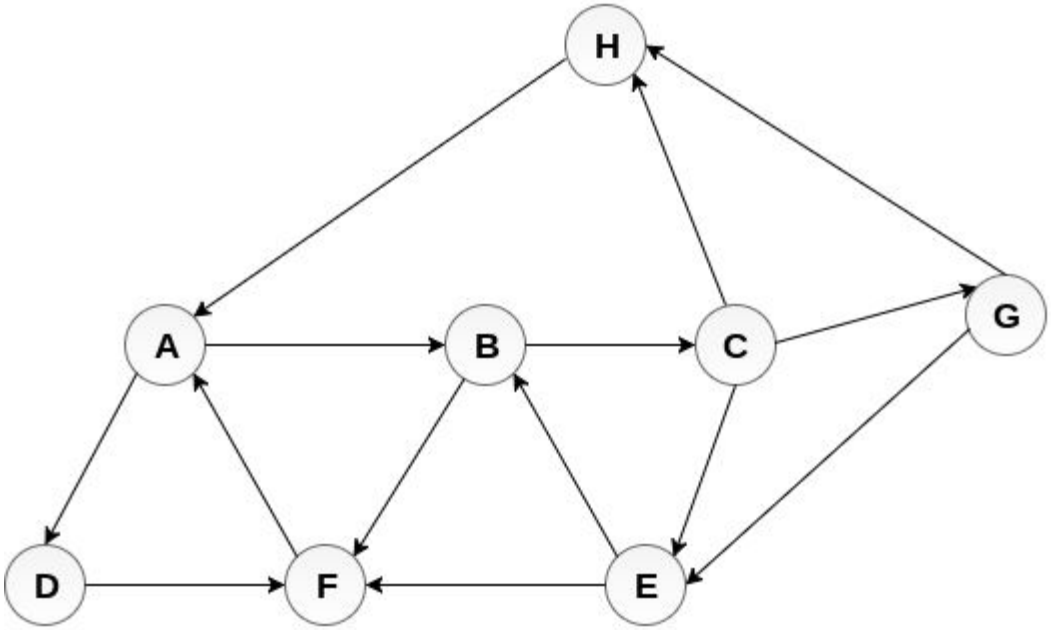
DFS - Recursive

```
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```

DFS



DFS - Graph



DFS

Push H onto the stack

> STACK : H

POP the top element of the stack i.e. H, print it and push all the neighbours of H onto the stack that are is ready state.

> Print H
> STACK : A

Pop the top element of the stack i.e. A, print it and push all the neighbours of A onto the stack that are in ready state.

> Print A
> Stack : B, D

Pop the top element of the stack i.e. D, print it and push all the neighbours of D onto the stack that are in ready state.

> Print D
> Stack : B, F

Pop the top element of the stack i.e. F, print it and push all the neighbours of F onto the stack that are in ready state.

> Print F
> Stack : B

Pop the top of the stack i.e. B and push all the neighbours

> Print B
> Stack : C

Pop the top of the stack i.e. C and push all the neighbours.

> Print C
> Stack : E, G

Pop the top of the stack i.e. G and push all its neighbours.

> Print G
> Stack : E

Pop the top of the stack i.e. E and push all its neighbours.

> Print E
> Stack :

H → A → D → F → B → C → G → E

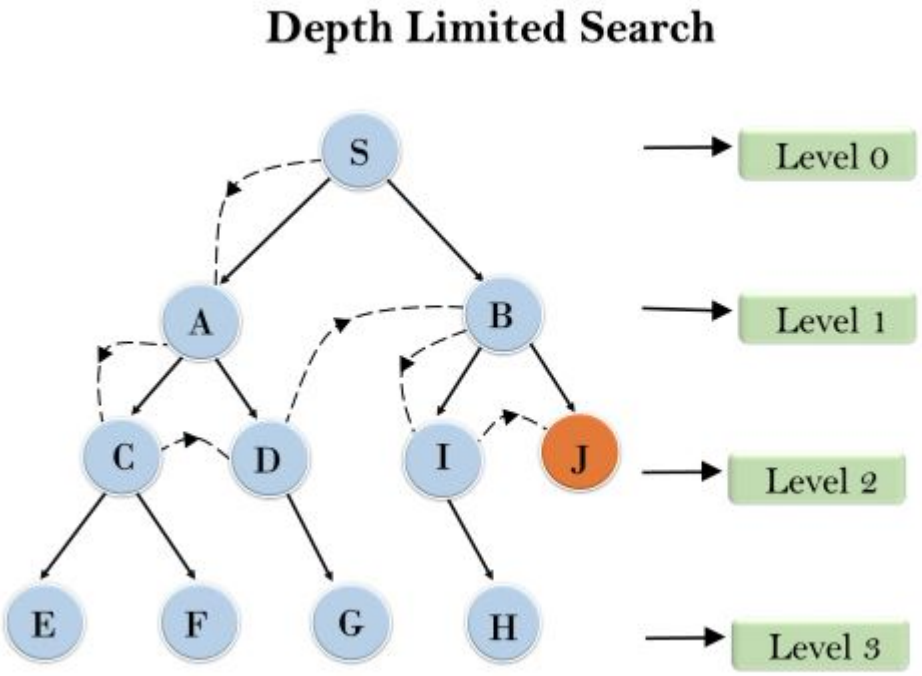
DFS – Time and Space

- **Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.
- **Time Complexity:** Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:
- **$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$**
- **Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)**
- **Space Complexity:** DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.
- **Optimal:** DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Depth Limited Search

- A depth-limited search algorithm is similar to depth-first search with a predetermined limit.
- Depth-limited search can solve the drawback of the infinite path in the Depth-first search.
- In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

DLS


Depth Limited Search
S
A
B
C
D
I
J
E
F
G
H
Level 0
Level 1
Level 2
Level 3

DLS - Limitation

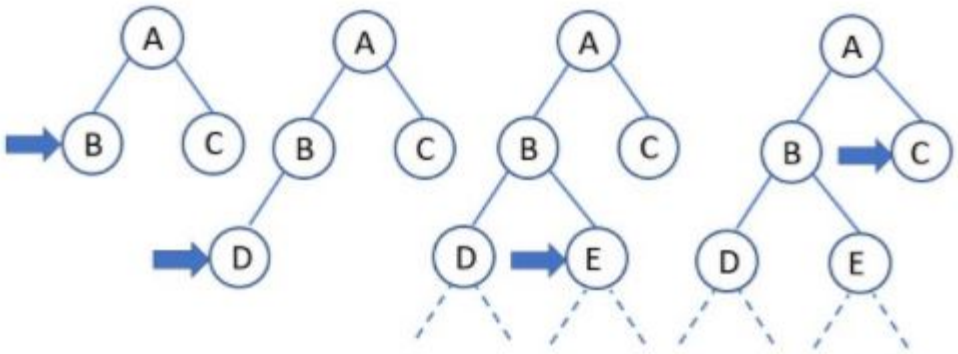
- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.
- **Advantages:**
  - Depth-limited search is Memory efficient.
- **Disadvantages:**
  - Depth-limited search also has a disadvantage of incompleteness.
  - It may not be optimal if the problem has more than one solution

DLS - Algorithm

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

DLS - Working



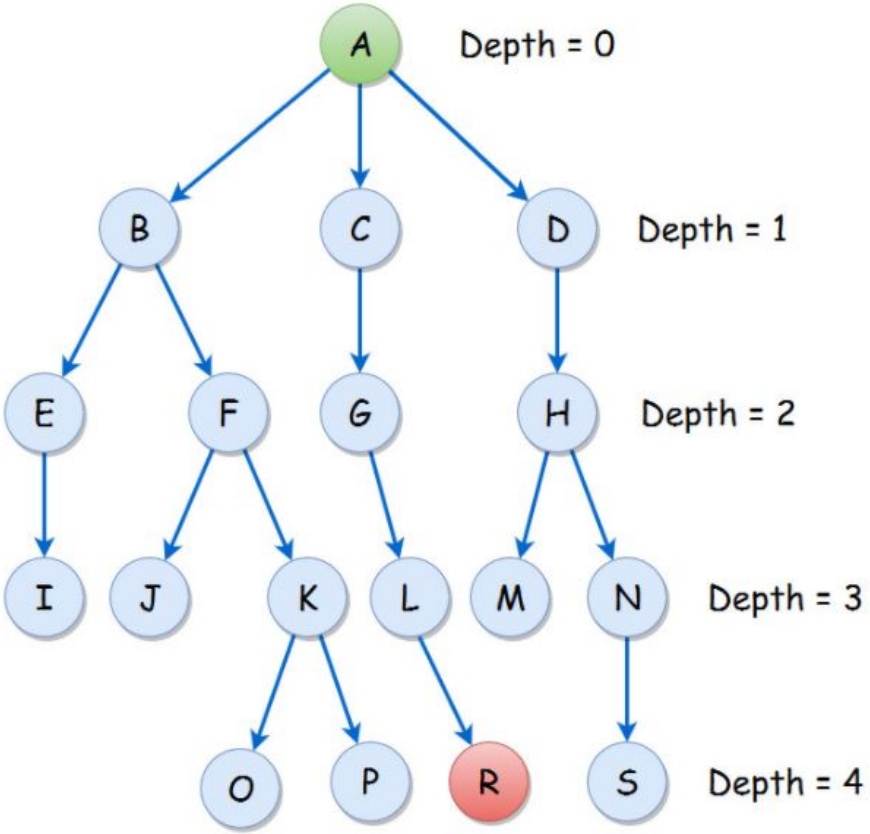
DLS – Time and Space

- **Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.
- **Time Complexity:** Time complexity of DLS algorithm is **$O(b^{\ell})$**.
- **Space Complexity:** Space complexity of DLS algorithm is **$O(b \times \ell)$**.
- **Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.

Iterative Deepening Depth First Search(IDDFS)

- A search algorithm which suffers neither the drawbacks of breadth-first nor depth-first search on trees is depth-first iterative-deepening
- **IDDFS** combines depth-first search's space-efficiency and breadth-first search's fast search (for nodes closer to root).
- Iterative deepening depth first search (IDDFS) is a hybrid of BFS and DFS. In IDDFS, we perform DFS up to a certain "limited depth," and keep increasing this "limited depth" after every iteration
- The idea is to perform depth-limited DFS repeatedly, with an increasing depth limit, until a solution is found

IDDFS



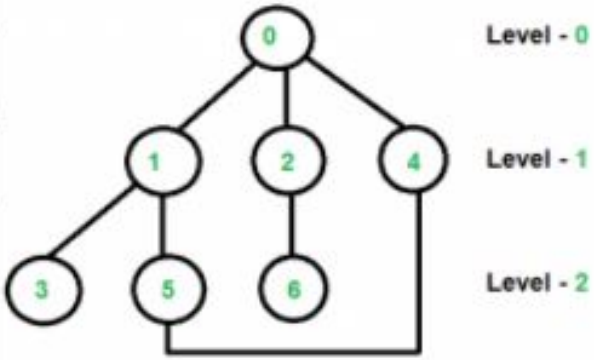
IDDFS

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*

DEPTH	DLS traversal
0	A
1	ABCD
2	ABEFCGDH
3	ABEIFJKCGLDHMN
4	ABEIFJKOPCGLRDHMNS

IDDFS

Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



The explanation of the above pattern is left to the readers.

IDDFS – Time and Space

- Space complexity = O(bd)
  - (since its like depth first search run different times, with maximum depth limit d)
- Time Complexity
  - $b + (b+b^2) + ...(b+....b^d) = O(b^d)$
    (i.e., asymptotically the same as BFS or DFS to limited depth d in the worst case)
- Complete?
  - Yes
- Optimal
  - Only if path cost is a non-decreasing function of depth
- IDS combines the small memory footprint of DFS, and has the completeness guarantee of BFS

Informed Search

- Informed Search algorithms have information on the goal state which helps in more efficient searching.
- This information is obtained by a function that estimates how close a state is to the goal state.
- Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristic Search

- Add domain-specific information to select what is the best path to continue searching along
- Define a **heuristic function**, h(n), that estimates the "goodness" of a node n. Specifically, h(n) = estimated cost (or distance) of minimal cost path from n to a goal state.
- The term heuristic means "serving to aid discovery" and is an estimate, based on domain-specific information that is computable from the current state description, of how close we are to a goal

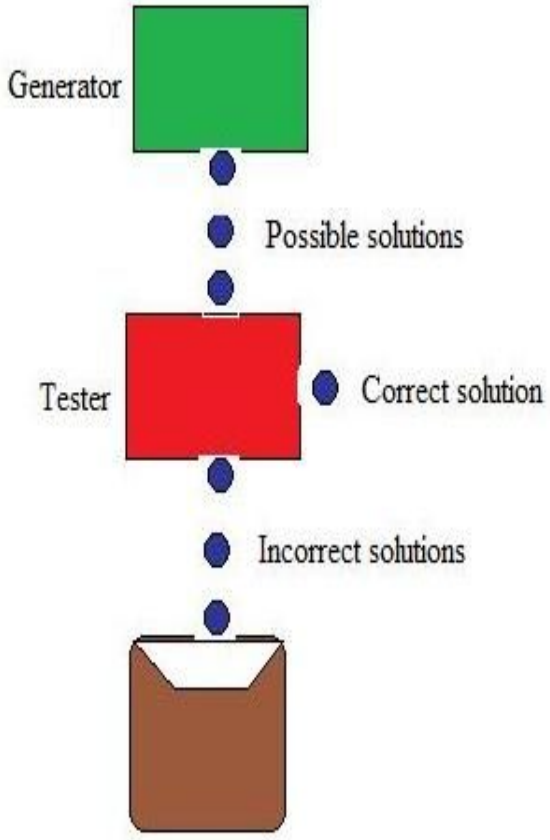
Heuristic function

- Heuristic is a function which is used in Informed Search, and it finds the most promising path.
- It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.
- The heuristic method, however, might not always give the best solution, but it 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

Generate and Test

- Generate-and-test search algorithm is a very simple algorithm that guarantees to find a solution if done systematically and there exists a solution.
- The generate and Test algorithm is a depth first search procedure because complete possible solutions are generated before test.
- This can be implemented states are likely to appear often in a tree
- It can be implemented on a search graph rather than a tree

Generate and Test



1. Generate a possible solution.
2. Test the solution.
3. If solution found THEN done ELSE return to step 1.

Generate and Test - Example

- The delivery robot must carry out a number of delivery activities, A, B, C, D, and E. Suppose that each activity happens at any of times 1, 2, 3, or 4
- {B≠3), (C≠2), (A≠B), (B≠C), (C<D), (A=D),

(E<A), (E<B), (E<C), (E<D), (B≠D)}

A – 1 4 4

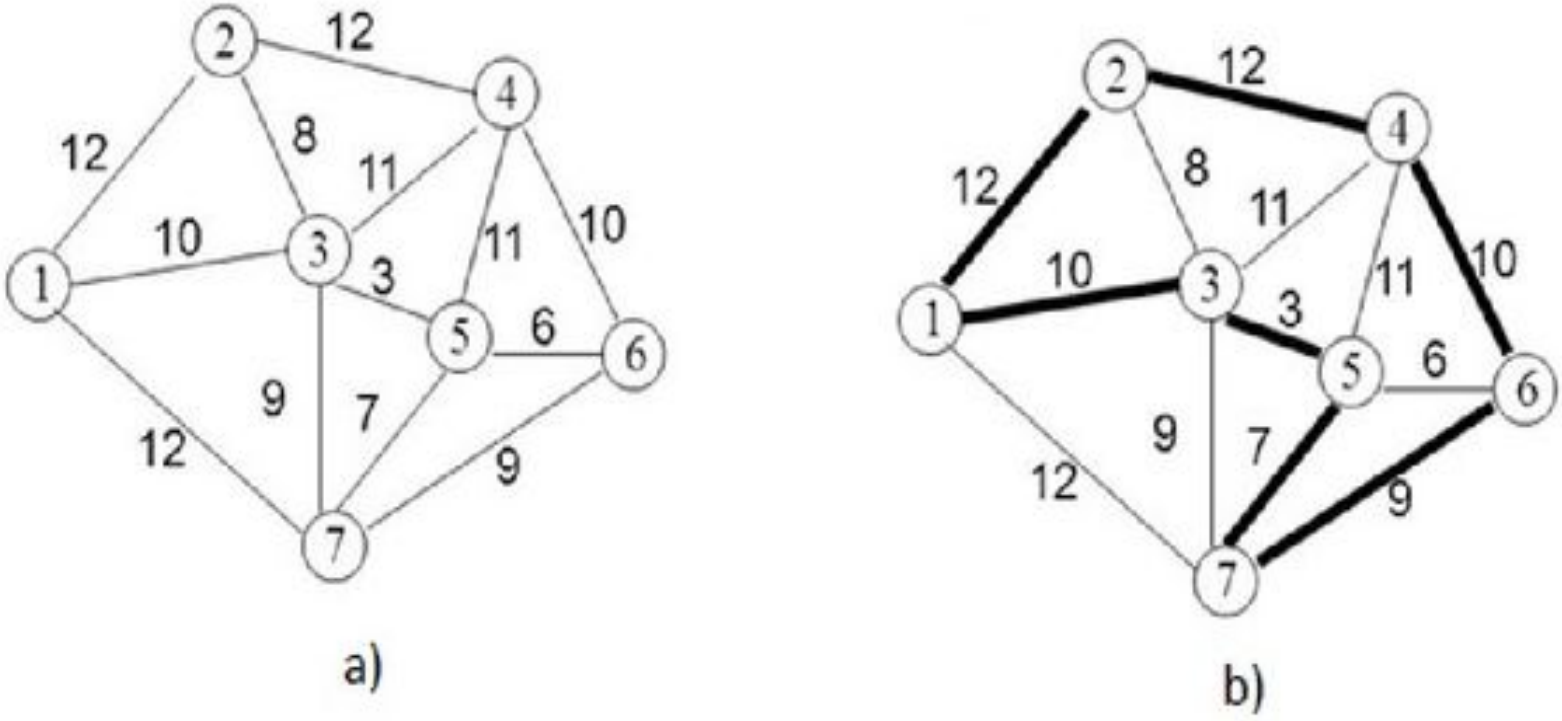
B – 1 2 4 2

C - 1 3 4 3

D - 4 4

E 1

TSP - Example



1 – 3 – 5 – 7 - 6 – 4 – 2

Best First Search

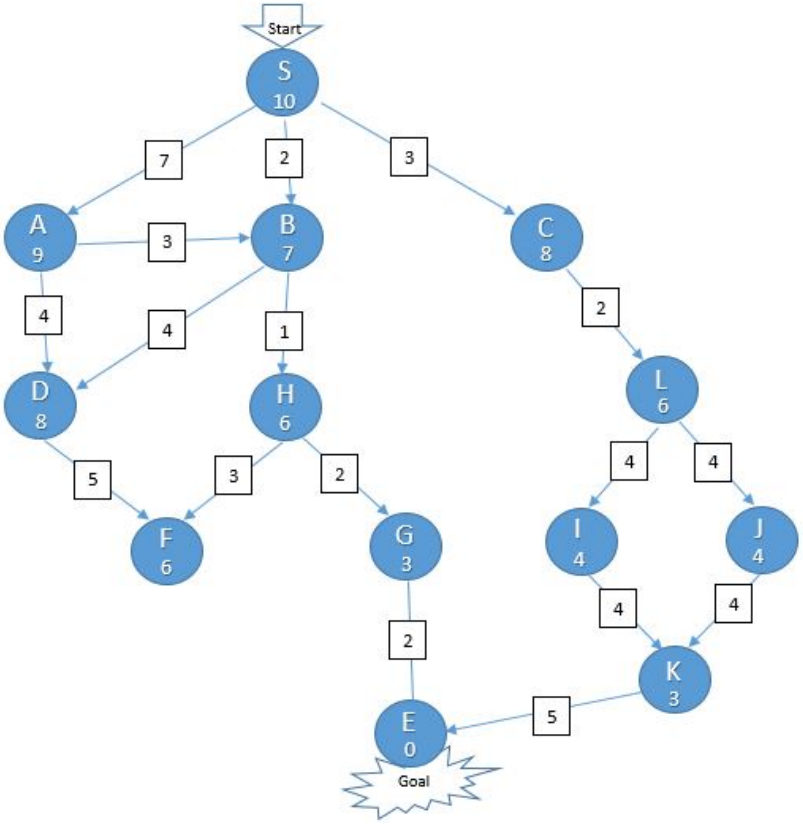
- Best first search is a traversal technique that decides which node is to be visited next by checking which node is the most promising one and then check it.
- Best first search is an instance of graph search algorithm in which a node is selected for expansion based on evaluation function f (n)
- Best first search can be implemented a priority queue, a data structure that will maintain the fringe in ascending order of f values.

Best First Search - Algorithm

Algorithm: BFS

1. Start with OPEN containing just the initial state
2. Until a goal is found or there are no nodes left on OPEN do:
   a. Pick the best node on OPEN
   b. Generate its successors
   c. For each successor do:
      i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent.
      ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

BFS - Example


Start
S
10
7
2
3
A
9
3
B
7
C
8
4
4
1
2
D
8
H
6
L
6
5
3
2
4
4
F
6
G
3
I
4
J
4
4
4
2
K
3
5
E
0
Goal

BFS - Working

Step 1 - Start by adding the start node (S) to the open list with the path distance as 0

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
S	10		

Repeat the next steps until the OPEN List is empty or the Goal node is moved to the CLOSED list

Step 2 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
A	9	S	
B	7		
C	8		

BFS - Working

Step 2 (b) - Re-order the list in ascending order of the combined hueristic value

OPEN			CLOSED				
Node	h(n)		Node	Parent Node			
B	7		S				
C	8						
A	9						

Step 3 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN			CLOSED				
Node	h(n)		Node	Parent Node			
C	8		S				
A	9		B	S			
D	8						
H	6						

BFS - Working

Step 3 (b) - Re-order the list in ascending order of the combined hueristic value

OPEN			CLOSED	
Node	h(n)		Node	Parent Node
H	6		S	
C	8		B	S
D	8			
A	9			

Step 4 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN			CLOSED	
Node	h(n)		Node	Parent Node
C	8		S	
D	8		B	S
A	9		H	B
F	6			
G	3			

BFS - Working

Step 4 (b) - Re-order the list in ascending order of the combined hueristic value

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
G	3	S	
F	6	B	S
C	8	H	B
D	8		
A	9		

Step 5 (a) - Move the first node in the OPEN list to the CLOSED list and expand its immediate successors by adding them to the OPEN list

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
F	6	S	
C	8	B	S
D	8	H	B
A	9	G	H
E	0		

BFS - Working

Step 6 (a) - Move the first node in the OPEN list to the CLOSED list and expand it's immediate successors by adding them to the OPEN list

OPEN		CLOSED	
Node	h(n)	Node	Parent Node
F	6	S	
C	8	B	S
D	8	H	B
A	9	G	H
		E	G

EXIT returning 'True' as the Goal node (E) is moved to the CLOSED list. Backtrack the closed list to get the optimal path (E --> G --> H --> B --> S)

S --> B --> H --> G --> E

A* Search

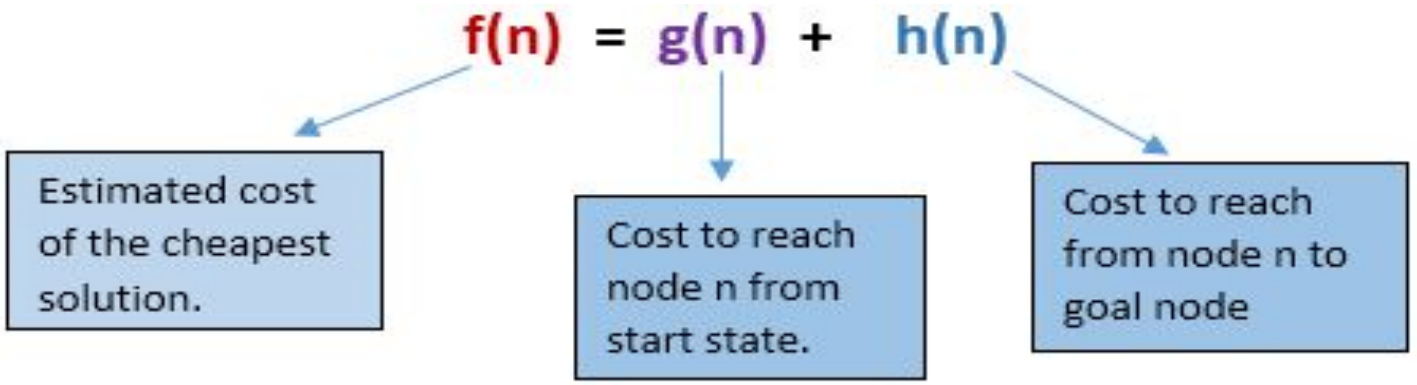
- It is an informed search algorithm, as it uses information about path cost and also uses heuristics to find the solution.
- To find the shortest path between nodes or graphs
- **A * algorithm** is a searching algorithm that searches for the shortest path between the *initial and the final state*
- It is an **advanced BFS** that *searches for shorter paths first rather than the longer paths*.
- A* is **optimal** as well as a **complete** algorithm

A* Search

- It calculates the cost, f(n) (n being the neighboring node), to travel to all of the neighboring nodes, and then enters the node with the lowest value of f(n).
- These values are calculated with the following formula:

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

- g(n) being the value of the shortest path from the start node to node *n*, and h(n) being a heuristic approximation of the node's value.



Heuristic value h(n)

- A given heuristic function h(n) is *admissible* if it never overestimates the real distance between *n* and the goal node.
- Therefore, for every node *n* , **h(n)≤ h*(n)**
- h*(n) being the real distance between *n* and the goal node.
- A given heuristic function h(n) is *consistent* if the estimate is always less than or equal to the estimated distance between the goal *n* and any given neighbor, plus the estimated cost of reaching that neighbor: c(n,m)+h(m) ≥h(n)
- c(n,m) being the distance between nodes n and m. Additionally, if h(n) is consistent, then we know the optimal path to any node that has been already inspected. This means that this function is *optimal*.

A* Search Algorithm

A* Algorithm

1. Start with OPEN containing only initial node. Set that node's g value to 0, its h' value to whatever it is, and its f' value to h'+0 or h'. Set CLOSED to empty list.
2. Until a goal node is found, repeat the following procedure: If there are no nodes on OPEN, report failure. Otherwise picj the node on OPEN with the lowest f' value. Call it BESTNODE. Remove it from OPEN. Place it in CLOSED. See if the BESTNODE is a goal state. If so exit and report a solution. Otherwise, generate the successors of BESTNODE but do not set the BESTNODE to point to them yet.

A* Search Algorithm

A* Algorithm (contd)

- For each of the SUCCESSOR, do the following:

a. Set SUCCESSOR to point back to BESTNODE. These backwards links will make it possible to recover the path once a solution is found.

b. Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR

c. See if SUCCESSOR is the same as any node on OPEN. If so call the node OLD.

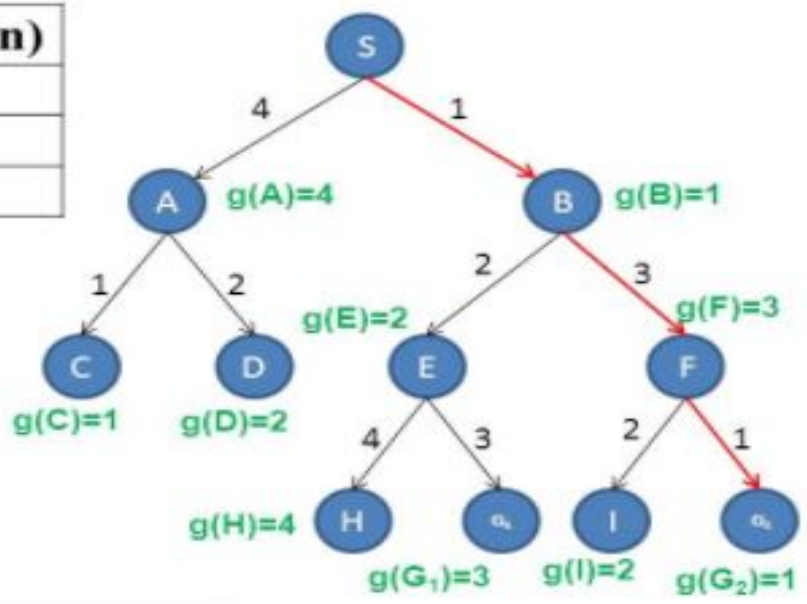
d. If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call the node on CLOSED OLD and add OLD to the list of BESTNODE's successors.

e. If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN and add it to the list of BESTNODE's successors. Compute f'(SUCCESSOR) = g(SUCCESSOR) + h'(SUCCESSOR)

A* Search - Example

EXAMPLE: Use A* search algorithm to find the solution.
Initial state: S, Goal state: G_1 or G_2

node	h(n)	node	h(n)	node	h(n)
A	11	D	8	H	7
B	5	E	4	I	3
C	9	F	2		

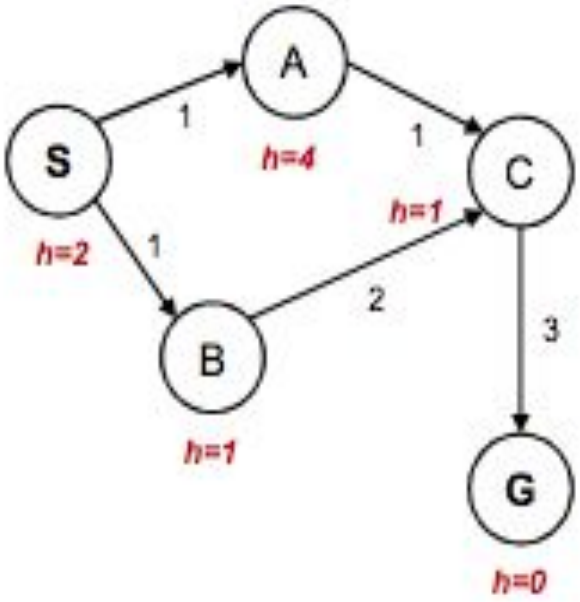


Solution:

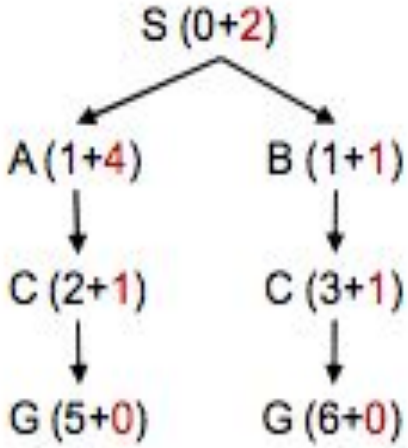
f(A)=h(A)+g(A)=11+4=15
f(B)=h(B)+g(B)=5+1=6
f(E)=h(E)+g(E)=4+2=6
f(F)=h(F)+g(F)=2+3=5
f(I)=h(I)+g(I)=3+2=5
$f(G_2)=h(G_2)+g(G_2)=0+1=1$

A* Search - Example

State space graph



Search tree



Memory Bounded Heuristic Search

- **Iterative Deepening A* (IDA*)**
  - Similar to Iterative Deepening Search, but cut off at (g(n)+h(n)) > max instead of depth > max
  - At each iteration, cutoff is the first f-cost that exceeds the cost of the node at the previous iteration
- **RBFS –**
  - It attempts to mimic the operation of BFS.
  - It replaces the f-value of each node along the path with the best f-value of its children.
  - Suffers from using too little memory.
- **Simple Memory Bounded A* (SMA*)**
  - Set max to some memory bound
  - If the memory is full, to add a node drop the worst (g+h) node that is already stored
  - Expands newest best leaf, deletes oldest worst leaf

IDA*

- IDA* (memory-bounded) algorithm does not keep any previously explored path (the same as A*).
- It needs to re-expand path if it is necessary and this will be a costly operation

Algorithm IDA*(n):
$limit \leftarrow f(n)$;
do until *success* **or** *limit* **unchanged**
  $limit \leftarrow DFS(n, limit)$;

Function DFS(n,limit):
if $f(n) > limit$ **return** $f(n)$;
if $goal(n)$ **then return with success**
else return lowest value of $DFS(s, limit)$ **for** $s \in S(n)$.

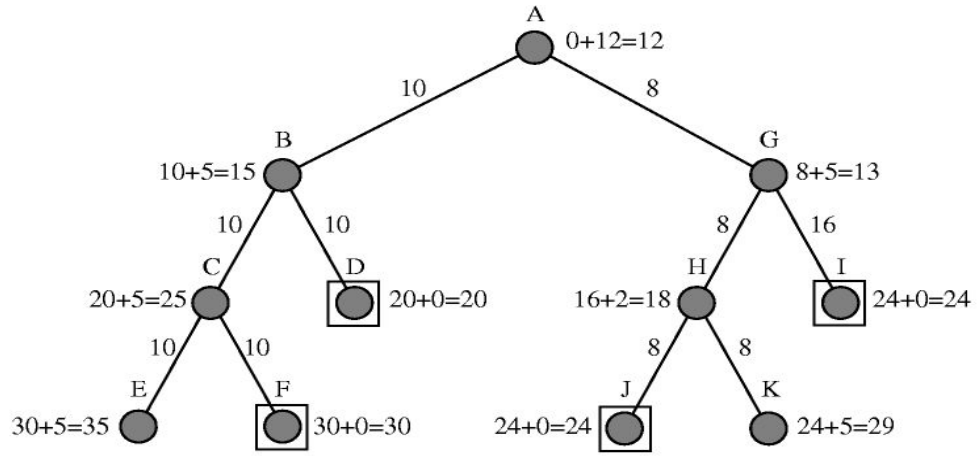
SMA*

- Proceeds life A*,expands best leaf until memory is full.
- Cannot add new node without dropping an old one. (always drops worst one)
- **Expands** the **best leaf** and **deletes** the **worst leaf**.
- If all have same f-value-selects same node for expansion and deletion.
- When the number of nodes in OPEN and CLOSED reaches some preset limit, MA* prunes the OPEN list by removing the leaf-node with highest f-cost.
- For each new successor the f-cost is propagated back up the tree. This keeps the tree very “informed” allowing the search to make better decisions, at the cost of some overhead

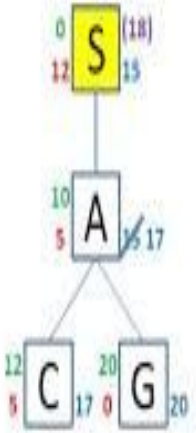
SMA* Algorithm

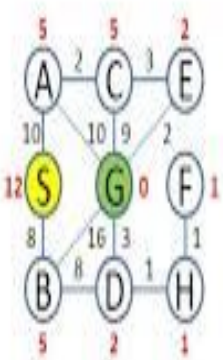
Algorithm SMA*(start):
put *start* on OPEN; USED ← 1;
loop
  if *empty*(OPEN) return with failure;
  best ← deepest least-f-cost leaf in OPEN;
  if *goal*(*best*) then return with success;
  succ ← *next-successor*(*best*);
  $f(succ) \leftarrow max(f(best), g(succ) + h(succ))$;
  if *completed*(*best*), $BACKUP(best)$;
  if $S(best)$ all in memory, remove *best* from OPEN.
  USED ← USED+1;
  if USED > MAX then
    delete shallowest, highest-f-cost node in OPEN;
    remove it from its parent's successor list;
    insert its parent on OPEN if necessary;
    USED ← USED-1;
  insert *succ* on OPEN.

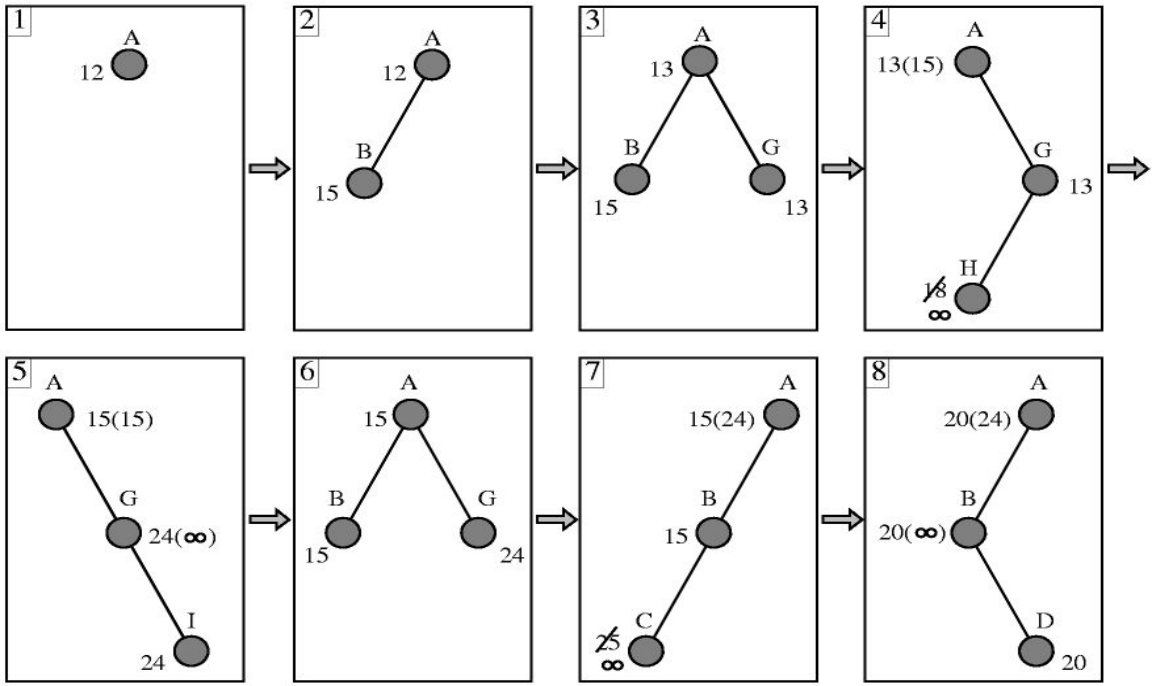
Procedure BACKUP(n):
if n is completed and has a parent then
  $f(n)$ ← least f-cost of all successors;
  if $f(n)$ changed, $BACKUP(parent(n))$.


A
0+12=12
10
8
B
10+5=15
G
8+5=13
10
10
8
16
C
D
H
I
20+5=25
20+0=20
16+2=18
24+0=24
10
10
8
8
E
F
J
K
30+5=35
30+0=30
24+0=24
24+5=29

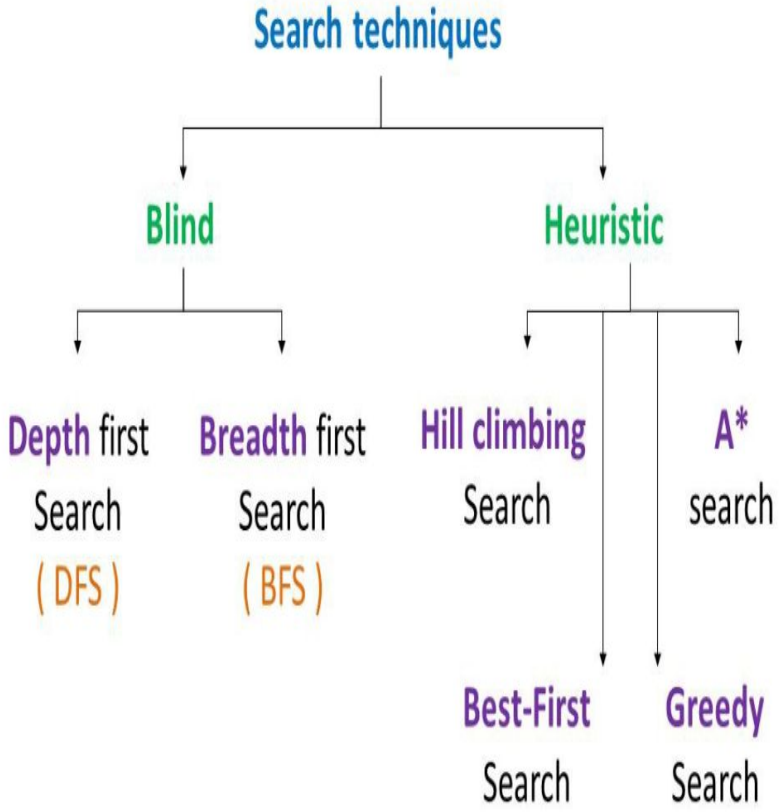
Problem


S
A
C
G


A
C
E
S
G
F
B
D
H


1
A
12
2
A
12
B
15
3
A
13
B
G
15
13
4
A
13(15)
G
13
H
5
A
15(15)
G
24(∞)
I
24
6
A
15
B
G
15
24
7
A
15(24)
B
15
C
8
A
20(24)
B
20(∞)
D
20

Search Techniques- A visual Quick Recall



S7-Local search Algorithms

- **Local search** is a heuristic method for solving computationally hard optimization problems.
- Local search can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions.
- Local search algorithms move from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed.
- Advantages: Very little memory — usually constant amount- Can often find reasonable solutions in infinite (continuous) state spaces
- Nutshell:
  - All that matters is the solution state
  - Don't care about solution path
- Examples
  - Hill Climbing
  - Simulated Annealing (suited for either local or global search)
    - Non Traditional Search and Optimization technique
  - Genetic Algorithm
    - Non Traditional Search and Optimization technique
    - Conceived by Prof. Holland of University of Michigan

https://medium.com/hengky-sanjaya-blog/informed-search-local-search-664888a32cf1

S7-Hill Climbing

- Heuristic search -used for mathematical optimization problems
- Variant of generate and test algorithm
  - *1. Generate possible solutions.*
    2. Test to see if this is the expected solution.
    3. If the solution has been found quit else go to step 1.
- Hill Climbing
  - It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as next node.

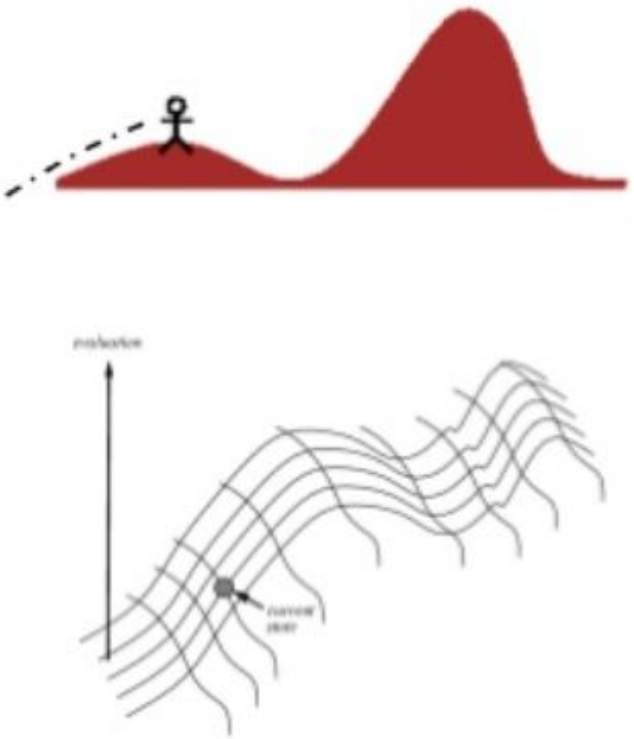
Extra Reference:
https://www.tutorialspoint.com/artificial_intelligence/artificial_intelligence_popular_search_algorithms.htm#:~:text=Local%20Beam%20Search,function%2C%20then%20the%20algorithm%20stops.

S7-Hill Climbing

- **"Like climbing Everest in thick fog with amnesia"**

- Hill climbing search algorithm (also known as greedy local search) uses a loop that continually moves in the direction of increasing values (that is uphill).

- It terminates when it reaches a peak where no neighbor has a higher value.



S7-Hill Climbing

- ***Step 1 :*** *Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.*
- ***Step 2 :*** *Loop until the solution state is found or there are no new operators present which can be applied to the current state.*
- *a) Select a state that has not been yet applied to the current state and apply it to produce a new state.*
- *b) Perform these to evaluate new state*
  i. If the current state is a goal state, then stop and return success.
  ii. If it is better than the current state, then make it current state and proceed further.
  iii. If it is not better than the current state, then continue in the loop until a solution is found.
- ***Step 3 :*** *Exit.*

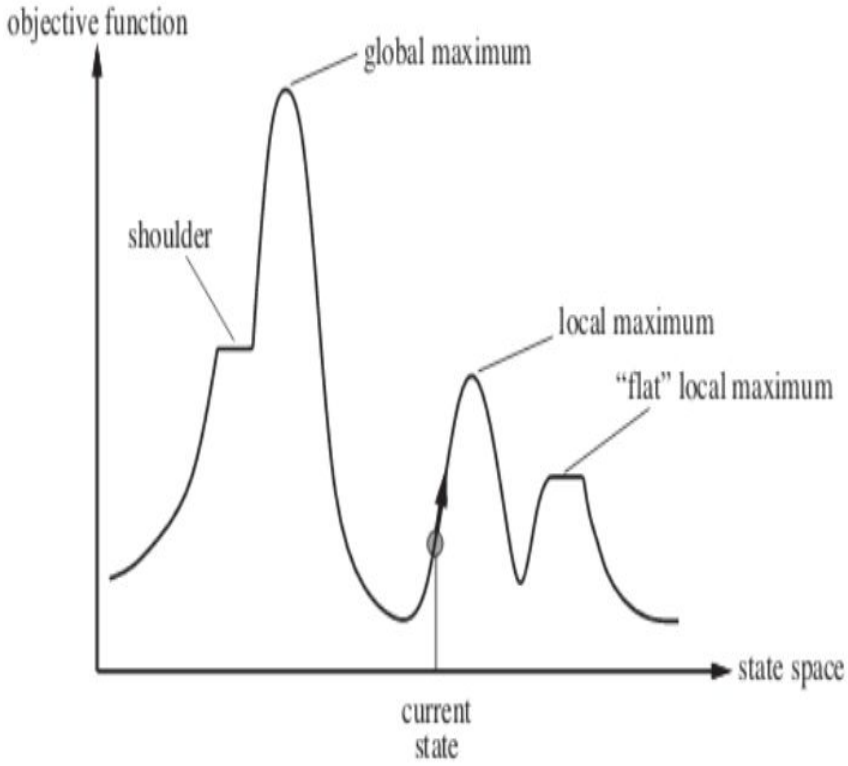
State Space diagram for Hill Climbing

- State space diagram is a graphical representation of the set of states our search algorithm can reach vs the value of our objective function(the function which we wish to maximize).

  X-axis : denotes the state space ie states or configuration our algorithm may reach.

  Y-axis : denotes the values of objective function corresponding to a particular state.

  The best solution will be that state space where objective function has maximum value(global maximum).



Different regions in the State Space Diagram

- **Local maximum:** It is a state which is better than its neighboring state however there exists a state which is better than it(global maximum). This state is better because here the value of the objective function is higher than its neighbors.
- **Global maximum :** It is the best possible state in the state space diagram. This because at this state, objective function has highest value.
- **Plateua/flat local maximum :** It is a flat region of state space where neighboring states have the same value.
- **Ridge :** It is region which is higher than its neighbours but itself has a slope. It is a special kind of local maximum.
- **Current state :** The region of state space diagram where we are currently present during the search.
- **Shoulder :** It is a plateau that has an uphill edge.

Problems in different regions in Hill climbing

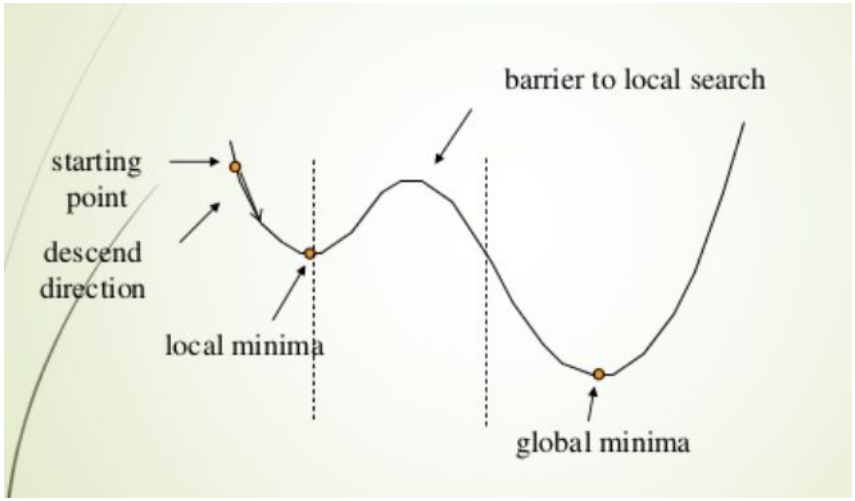
- Hill climbing cannot reach the optimal/best state(global maximum) if it enters any of the following regions :
- **Local maximum :** At a local maximum all neighboring states have a values which is worse than the current state. Since hill-climbing uses a greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.
  To overcome local maximum problem : Utilize backtracking technique. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.
- **Plateau :** On plateau all neighbors have same value . Hence, it is not possible to select the best direction.

  To overcome plateaus : Make a big jump. Randomly select a state far away from the current state. Chances are that we will land at a non-plateau region
- **Ridge :** Any point on a ridge can look like peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.
  To overcome Ridge : In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

S7-Simulated Annealing

- *Simulated Annealing came from the concept of annealing in physics. This technique is used to increase the size of crystals and to reduce the defects in crystals. This was done by heating and then suddenly cooling of crystals.*
- *Advantages*
  - can deal with arbitrary systems and cost functions
  - is relatively easy to code, even for complex problems
  - generally gives a "good" solution

Why SA?:
Difficulty in searching Global Optima



- SA is a global optimization technique.
- SA distinguishes between different local optima.
- SA is a memory less algorithm, the algorithm does not use any information gathered during the search
- SA is motivated by an analogy to annealing in solids.
- Simulated Annealing – an iterative improvement algorithm.

S7-Simulated Annealing

Algorithm

- First generate a random solution
- Calculate it's cost using some cost function
- Generate a random neighbor solution and calculate it's cost
- Compare the cost of old and new random solution
- If C old > C new then go for old solution otherwise go for new solution
- Repeat steps 3 to 5 until you reach an acceptable optimized solution of given problem

S8-Local Beam Search

- The search begins with k randomly generated states
- At each step, all the successors of all k states are generated
- If any one of the successors is a goal, the algorithm halts
- Otherwise, it selects the k best successors from the complete list and repeats
- The parallel search of beam search leads quickly to abandoning unfruitful searches and moves its resources to where the most progress is being made
- In stochastic beam search the maintained successor states are chosen with a probability based on their goodness

function BEAM-SEARCH(*problem*, *k*) returns a solution state

start with *k* randomly generated states

loop

generate all successors of all *k* states

if any of them is a solution then return it

else select the *k* best successors

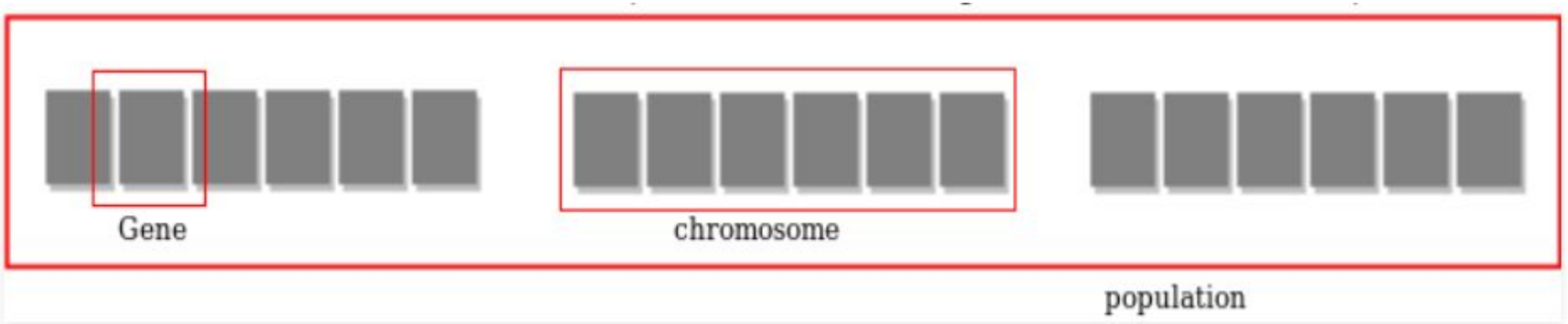
- **Local Beam Search Algorithm**
- Keep track of *k* states instead of one
  - Initially: *k* random states
  - Next: determine all successors of *k* states
    - Extend **all paths** one step
    - Reject all paths with loops
    - Sort all paths in queue by **estimated** distance to goal
  - If any of successors is goal → finished
  - Else select *k* best from successors and repeat.

S8-Genetic Algorithms

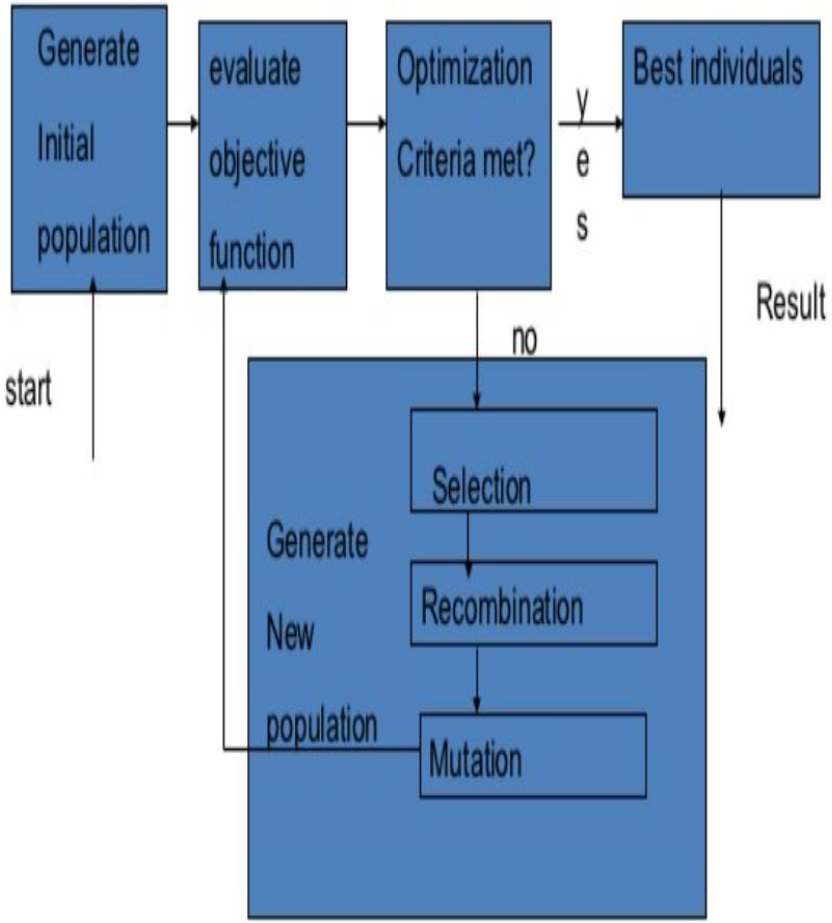
- Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics.
- **They are commonly used to generate high-quality solutions for optimization problems and search problems.**
- n simple words, they simulate “survival of the fittest” among individual of consecutive generation for solving a problem. **Each generation consist of a population of individuals** and each individual represents a point in search space and possible solution. Each individual is represented as a string of character/integer/float/bits. This string is analogous to the Chromosome.

S8-Genetic Algorithms

- Search Space
  - The population of individuals are maintained within search space. Each individual represent a solution in search space for given problem. Each individual is coded as a finite length vector (analogous to chromosome) of components. These variable components are analogous to Genes. Thus a chromosome (individual) is composed of several genes (variable components).
  - A Fitness Score is given to each individual which **shows the ability of an individual to "compete"**. The individual having optimal fitness score (or near optimal) are sought.



S8-Genetic Algorithms



- Genetic algorithms are based on the theory of selection

  1.A set of random solutions are generated
- Only those solutions survive that satisfy a fitness function
- Each solution in the set is a chromosome
- A set of such solutions forms a population

2. The algorithm uses three basic genetic operators namely

(i)Reproduction

(ii)crossover and

(iii) mutation along with a fitness function to evolve a new population or the next generation

- Thus the algorithm uses these operators and the fitness function to guide its search for the optimal solution
- It is a guided random search mechanism

S8-Genetic Algorithms

Significance of the genetic operators

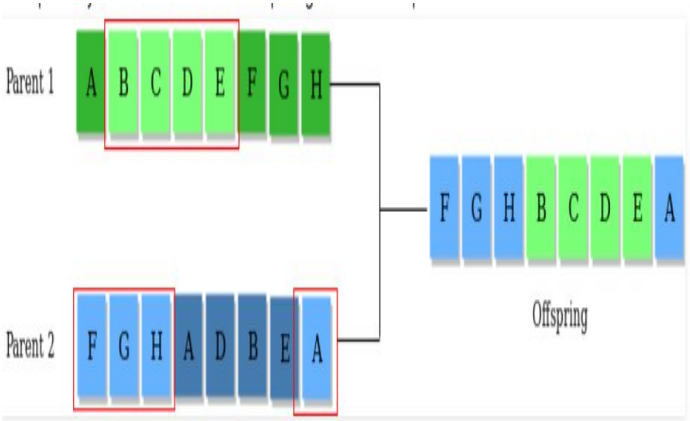
- Reproduction or selection by two parent chromosomes is done based on their fitness
- Reproduction ensures that only the fittest of the solutions made to form offsprings
- Reproduction will force the GA to search that area which has highest fitness values

Crossover or recombination: Crossover ensures that the search progresses in the right direction by making new chromosomes that possess characteristics similar to both the parents

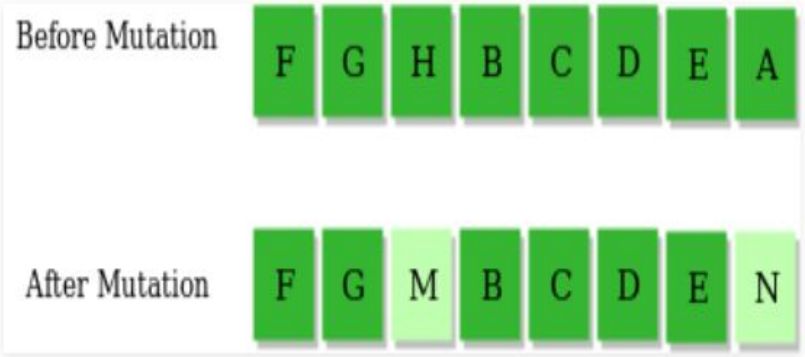
Mutation: To avoid local optimum, mutation is used

It facilitates a sudden change in a gene within a chromosome allowing the algorithm to see for the solution far way from the current ones

- Cross Over



- Mutation



GA can be summarized as:

- 1) Randomly initialize populations p
- 2) Determine fitness of population
- 3) Untill convergence repeat:
  - a) Select parents from population
  - b) Crossover and generate new population
  - c) Perform mutation on new population
  - d) Calculate fitness for new population

https://www.geeksforgeeks.org/genetic-algorithms/

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

Lab 5: Developing Best first search and A* Algorithm for real world problems

S9-S10 LAB

S11-Adversarial search Methods-Game playing-Important concepts

- Adversarial search: Search based on Game theory- Agents- Competitive environment
- ***According to game theory, a game is played between two players. To complete the game, one has to win the game and the other looses automatically.'***
- Such Conflicting goal- adversarial search
- Game playing technique- Those games- Human Intelligence and Logic factor- Excluding other factors like Luck factor
  - Tic-Tac-Toe, Checkers, Chess – Only mind works, no luck works

S11-Adversarial search Methods-Game playing-Important concepts

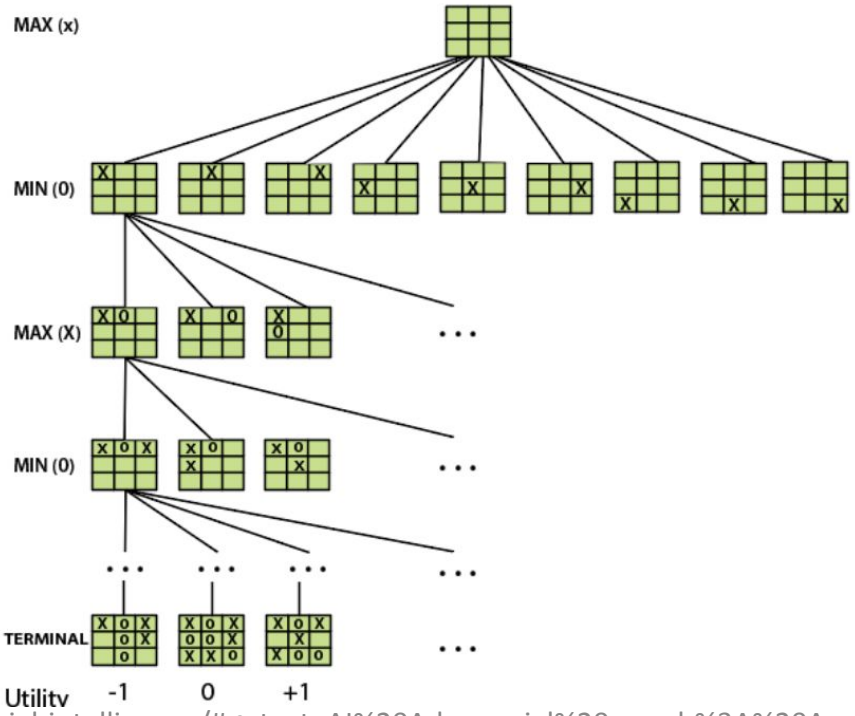
- **Techniques required to get the best optimal solution (Choose Algorithms for best optimal solution within limited time)**
  - **Pruning:** A technique which allows ignoring the unwanted portions of a search tree which make no difference in its final result.
  - **Heuristic Evaluation Function:** It allows to approximate the cost value at each level of the search tree, before reaching the goal node.

https://www.tutorialandexample.com/adversarial-search-in-artificial-intelligence/#:~:text=AI%20Adversarial%20search%3A%20Adversarial%20search,order%20to%20win%20the%20game.

S11- Game playing and knowledge structure- **Elements of Game Playing search**

- To play a game, we use a game tree to know all the possible choices and to pick the best one out. There are following elements of a game-playing:
- **S_0:** It is the initial state from where a game begins.
- **PLAYER (s):** It defines which player is having the current turn to make a move in the state.
- **ACTIONS (s):** It defines the set of legal moves to be used in a state.
- **RESULT (s, a):** It is a transition model which defines the result of a move.
- **TERMINAL-TEST (s)**: It defines that the game has ended and returns true.
- **UTILITY (s,p):** It defines the final value with which the game has ended. This function is also known as **Objective function** or **Payoff function**. The price which the winner will get i.e.
- **(-1):** If the PLAYER loses.
- **(+1):** If the PLAYER wins.
- **(0):** If there is a draw between the PLAYERS.

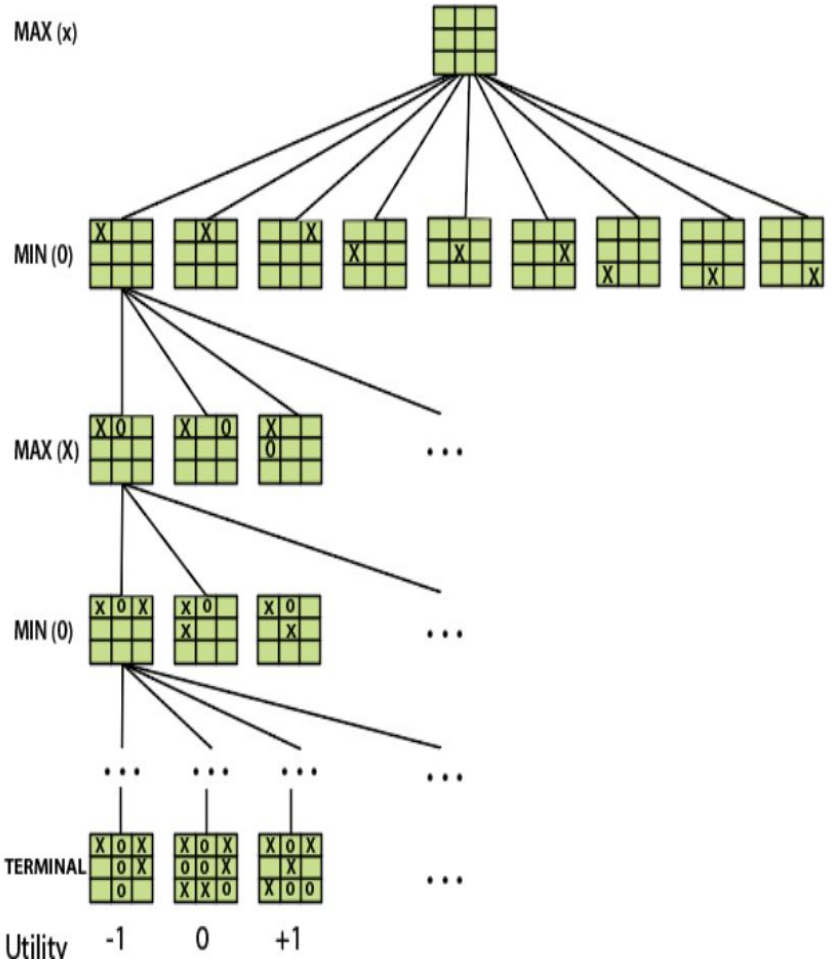
- ***For example**, in **chess, tic-tac-toe,** we have two or three possible outcomes. Either to win, to lose, or to draw the match with values **+1,-1 or 0.***
- ***Game Tree for Tic-Tac-Toe***
  - Node: Game states, Edges: Moves taken by players



https://www.tutorialandexample.com/adversarial-search-in-artificial-intelligence/#:~:text=AI%20Adversarial%20search%3A%20Adversarial%20search,order%20to%20win%20the%20game.

S11- Game playing and knowledge structure- **Elements of Game Playing search**

- **INITIAL STATE (S_0):** The top node in the game-tree represents the initial state in the tree and shows all the possible choice to pick out one.
- **PLAYER (s):** There are two players, **MAX and MIN**. **MAX** begins the game by picking one best move and place **X** in the empty square box.
- **ACTIONS (s):** Both the players can make moves in the empty boxes chance by chance.
- **RESULT (s, a):** The moves made by **MIN** and **MAX** will decide the outcome of the game.
- **TERMINAL-TEST(s):** When all the empty boxes will be filled, it will be the terminating state of the game.
- **UTILITY:** At the end, we will get to know who wins: **MAX** or **MIN,** and accordingly, the price will be given to them**.**



S12-Game as a search problem

- **Types of algorithms in Adversarial search**
  - In a **normal search**, we follow a sequence of actions to reach the goal or to finish the game optimally. But in an **adversarial search**, the result depends on the players which will decide the result of the game. It is also obvious that the solution for the goal state will be an optimal solution because the player will try to win the game with the shortest path and under limited time.
    - **Minmax Algorithm**
    - **Alpha-beta Pruning**

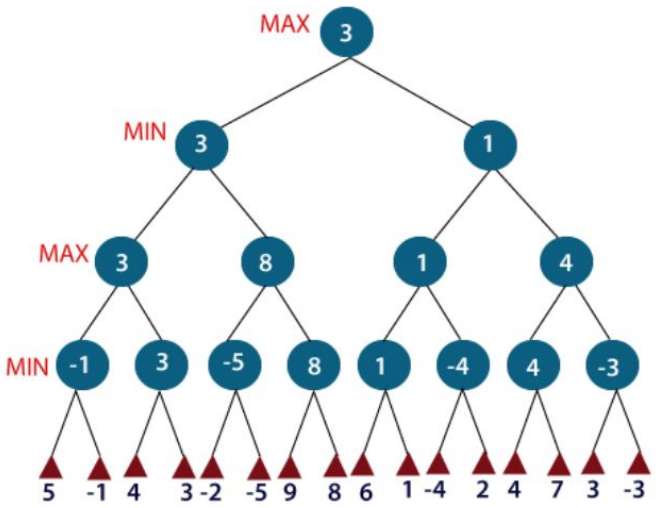
S12-Game as a search problem-Minimax approach

- Minimax/Minmax/MM/Saddle Point:
  - Decision strategy-Game Theory
    - Minimize loosing chances- Maximize winning chances
    - Two-player game strategy
    - Players will be two namely:
    - **MIN:** Decrease the chances of **MAX** to win the game.
    - **MAX:** Increases his chances of winning the game.
    - Result of the game/Utility value
      - Heuristic function propagating from initial node to root node
      - Backtracking technique-Best choice

S12-Minimax Algorithm

- Follows DFS
  - Follows same path cannot change in middle- i.e., Move once made cannot be altered- That is this is DFS and not BFS
- Algorithm
  - Keep on generating the game tree/ search tree till a limit **d.**
  - Compute the move using a heuristic function.
  - Propagate the values from the leaf node till the current position following the minimax strategy.
  - Make the best move from the choices.

S12-Minimax Algorithm



- For example, in the figure, the two players **MAX** and **MIN** are there. **MAX** starts the game by choosing one path and propagating all the nodes of that path. Now, **MAX** will backtrack to the initial node and choose the best path where his utility value will be the maximum. After this, its **MIN** chance. **MIN** will also propagate through a path and again will backtrack, but **MIN** will choose the path which could minimize **MAX** winning chances or the utility value.
- ***So, if the level is minimizing, the node will accept the minimum value from the successor nodes. If the level is maximizing, the node will accept the maximum value from the successor.***
- **Note**: The time complexity of MINIMAX algorithm is **$O(b^d)$** where b is the branching factor and d is the depth of the search tree.

S13- Alpha beta pruning

- Advanced version of MINIMAX algorithm
- Any optimization algorithm- performance measure is the first consideration
- Drawback of Minimax:
  - Explores each node in the tree deeply to provide the best path among all the paths
  - Increases time complexity
- Alpha beta pruning: Overcomes drawback by less exploring nodes of search tree

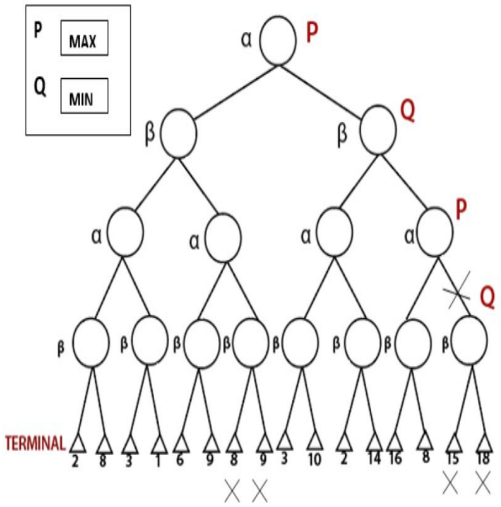
S13- Alpha beta pruning

- Cutoff the search by exploring less number of nodes
- It makes same moves as Minimax algorithm does- but prunes unwanted branches using the pruning techniques
- Alpha beta pruning works on 2 threshold values α and β
  - **α:** It is the best highest value, a **MAX** player can have. It is the lower bound, which represents negative infinity value.
  - **β:** It is the best lowest value, a **MIN** player can have. It is the upper bound which represents positive infinity.
- So, each MAX node has α-value, which never decreases, and each MIN node has β-value, which never increases.
- **Note:** Alpha-beta pruning technique can be applied to trees of any depth, and it is possible to prune the entire subtrees easily.

S13- Alpha beta pruning

- Consider the below example of a game tree where **P** and **Q** are two players.
- The game will be played alternatively, i.e., chance by chance.
- Let, **P** be the player who will try to win the game by maximizing its winning chances. **Q** is the player who will try to minimize **P'**s winning chances.
- Here, **α** will represent the maximum value of the nodes, which will be the value for **P** as well.
- **β** will represent the minimum value of the nodes, which will be the value of **Q**.

S13- Alpha beta pruning



- Any one player will start the game. Following the DFS order, the player will choose one path and will reach to its depth, i.e., where he will find the **TERMINAL** value.
- If the game is started by player P, he will choose the maximum value in order to increase its winning chances with maximum utility value.
- If the game is started by player Q, he will choose the minimum value in order to decrease the winning chances of A with the best possible minimum utility value.
- Both will play the game alternatively.
- The game will be started from the last level of the game tree, and the value will be chosen accordingly.
- Like in the below figure, the game is started by player Q. He will pick the leftmost value of the TERMINAL and fix it for beta (β). Now, the next TERMINAL value will be compared with the β-value. If the value will be smaller than or equal to the β-value, replace it with the current β-value otherwise no need to replace the value.
- After completing one part, move the achieved β-value to its upper node and fix it for the other threshold value, i.e., α.
- Now, its P turn, he will pick the best maximum value. P will move to explore the next part only after comparing the values with the current α-value. If the value is equal or greater than the current α-value, then only it will be replaced otherwise we will prune the values.
- The steps will be repeated unless the result is not obtained.
- So, number of pruned nodes in the above example are **four** and MAX wins the game with the maximum **UTILITY** value, i.e.,**3**
- The rule which will be followed is: **"Explore nodes if necessary otherwise prune the unnecessary nodes."**
- **Note:** It is obvious that the result will have the same **UTILITY** value that we may get from the MINIMAX strategy.

S13- Game theory problems

- **Game theory** is basically a branch of mathematics that is used to typical strategic interaction between different players (agents), all of which are equally rational, in a context with predefined rules (of playing or maneuvering) and outcomes.
- GAME can be defined as a set of players, actions, strategies, and a final playoff for which all the players are competing.
- Game Theory has now become a describing factor for both Machine Learning algorithms and many daily life situations.

https://www.geeksforgeeks.org/game-theory-in-ai/

S13- Game theory problems

- Types of Games
  - **Zero-Sum and Non-Zero Sum Games:** In non-zero-sum games, there are multiple players and all of them have the option to gain a benefit due to any move by another player. In zero-sum games, however, if one player earns something, the other players are bound to lose a key playoff.
  - **Simultaneous and Sequential Games:** Sequential games are the more popular games where every player is aware of the movement of another player. Simultaneous games are more difficult as in them, the players are involved in a concurrent game. BOARD GAMES are the perfect example of sequential games and are also referred to as turn-based or extensive-form games.
  - **Imperfect Information and Perfect Information Games:** In a perfect information game, every player is aware of the movement of the other player and is also aware of the various strategies that the other player might be applying to win the ultimate playoff. In imperfect information games, however, no player is aware of what the other is up to. CARDS are an amazing example of Imperfect information games while CHESS is the perfect example of a Perfect Information game.
  - **Asymmetric and Symmetric Games:** Asymmetric games are those win in which each player has a different and usually conflicting final goal. Symmetric games are those in which all players have the same ultimate goal but the strategy being used by each is completely different.
  - **Co-operative and Non-Co-operative Games:** In non-co-operative games, every player plays for himself while in co-operative games, players form alliances in order to achieve the final goal.

https://www.geeksforgeeks.org/game-theory-in-ai/

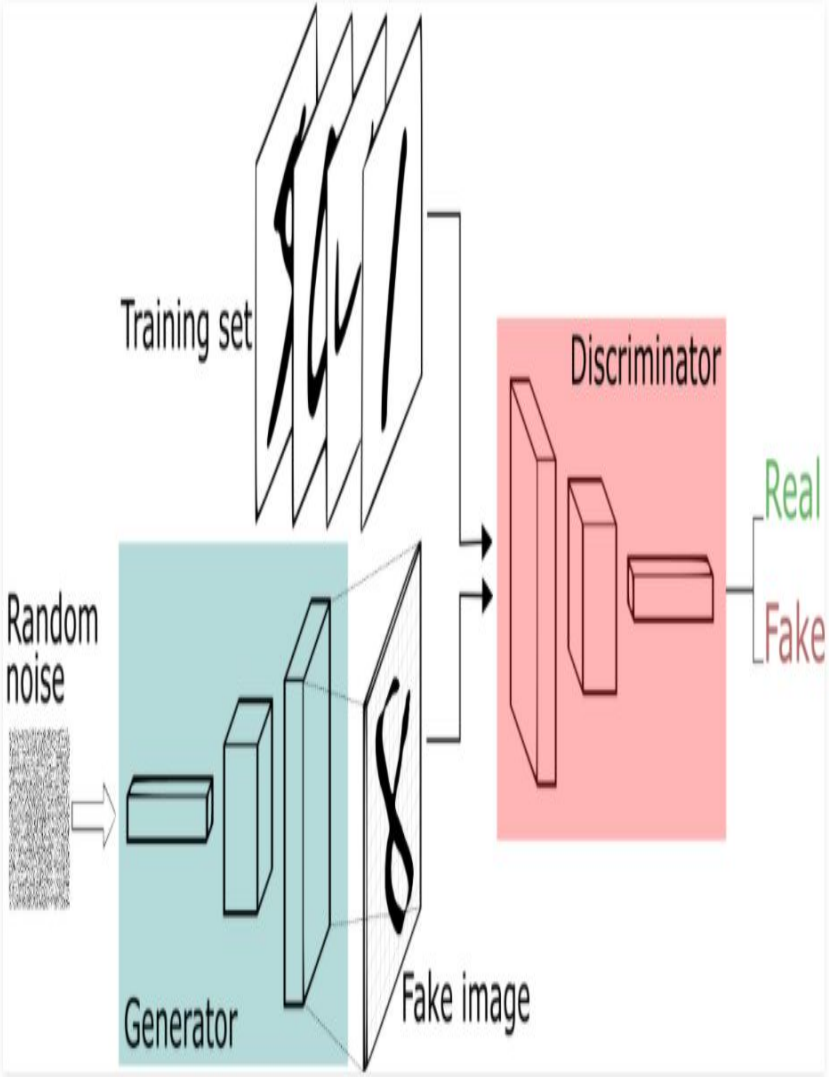
S13- Game theory problems

- **Nash equilibrium:**
  Nash equilibrium can be considered the essence of Game Theory. It is basically a state, a point of equilibrium of collaboration of multiple players in a game. Nash Equilibrium guarantees maximum profit to each player. Let us try to understand this with the help of Generative Adversarial Networks (GANs).
- What is GAN?
  - It is a combination of two neural networks: the Discriminator and the Generator. The Generator Neural Network is fed input images which it analyzes and then produces new sample images, which are made to represent the actual input images as close as possible

GAN

- Once the images have been produced, they are sent to the Discriminator Neural Network. This neural network judges the images sent to it and classifies them as generated images and actual input images. If the image is classified as the original image, the DNN changes its parameters of judging.
- If the image is classified as a generated image, the image is rejected and returned to the GNN. The GNN then alters its parameters in order to improve the quality of the image produced.
- This is a competitive process which goes on until both neural networks do not require to make any changes in their parameters and there can be no further improvement in both neural networks. This state of no further improvement is known as NASH EQUILIBRIUM.
- In other words, GAN is a 2-player competitive game where both players are continuously optimizing themselves to find a Nash Equilibrium.

GAN



Where is GAME THEORY now?

- Game Theory is increasingly becoming a part of the real-world in its various applications in areas like public health services, public safety, and wildlife.
- Currently, game theory is being used in adversary training in GANs, multi-agent systems, and imitation and reinforcement learning. In the case of perfect information and symmetric games, many Machine Learning and Deep Learning techniques are applicable.
- The real challenge lies in the development of techniques to handle incomplete information games, such as Poker.
- The complexity of the game lies in the fact that there are too many combinations of cards and the uncertainty of the cards being held by the various players.

S13- Game theory problems

- Prisoner's Dilemma.
- Closed-bag exchange Game,
- The Friend or Foe Game, and
- The iterated Snowdrift Game.

https://www.geeksforgeeks.org/game-theory-in-ai/

References

- 1. Parag Kulkarni, Prachi Joshi, “Artificial Intelligence –Building Intelligent Systems ”PHI learning private Ltd, 2015
- 2.Kevin Night and Elaine Rich, Nair B., “Artificial Intelligence (SIE)”, Mc Graw Hill- 2008.
- 3.Stuart Russel and Peter Norvig “AI – A Modern Approach”, 2nd Edition, Pearson Education 2007
- 4. www.javatpoint.com
- 5. www.geeksforgeeks.org
- 6.www.mygreatlearning.com
- 7. www.tutorialspoint.com