Unit 2

CLO-2 : Apply appropriate searching techniques to solve a real world problem

CLR-2 : Understand the search technique procedures applied to real world problems

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

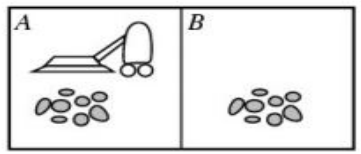
Examples of Search Algorithm

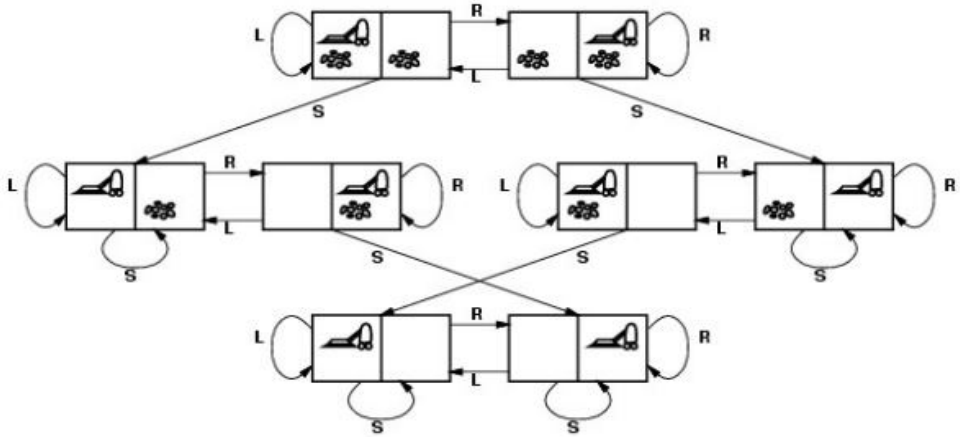
- TSP
- 8 Puzzle problem
- Tic Tac Toe
- N-queen problem
- Tower of Hanoi
- Water Jug Problem
- Blocks World
- Vacuum Cleaner

Vacuum Cleaner

Example: Vacuum-Cleaner

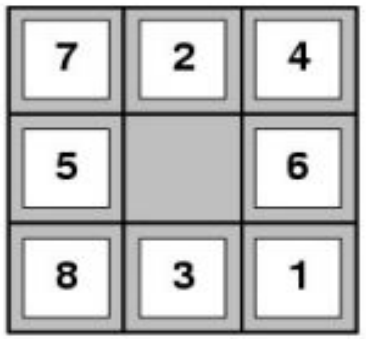
- States
  - 8 states
- Initial state
  - any state
- Actions
  - Left, Right, and Suck
- Transition model
  - complete state space, see next page
- Goal test
  - whether both squares are clean
- Path cost
  - each step costs 1



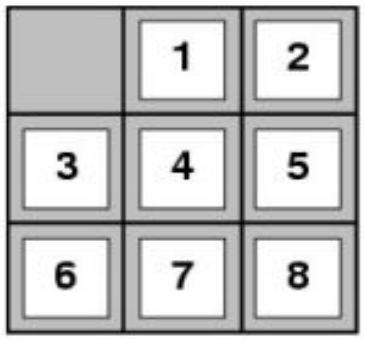


8 Puzzle Problem

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

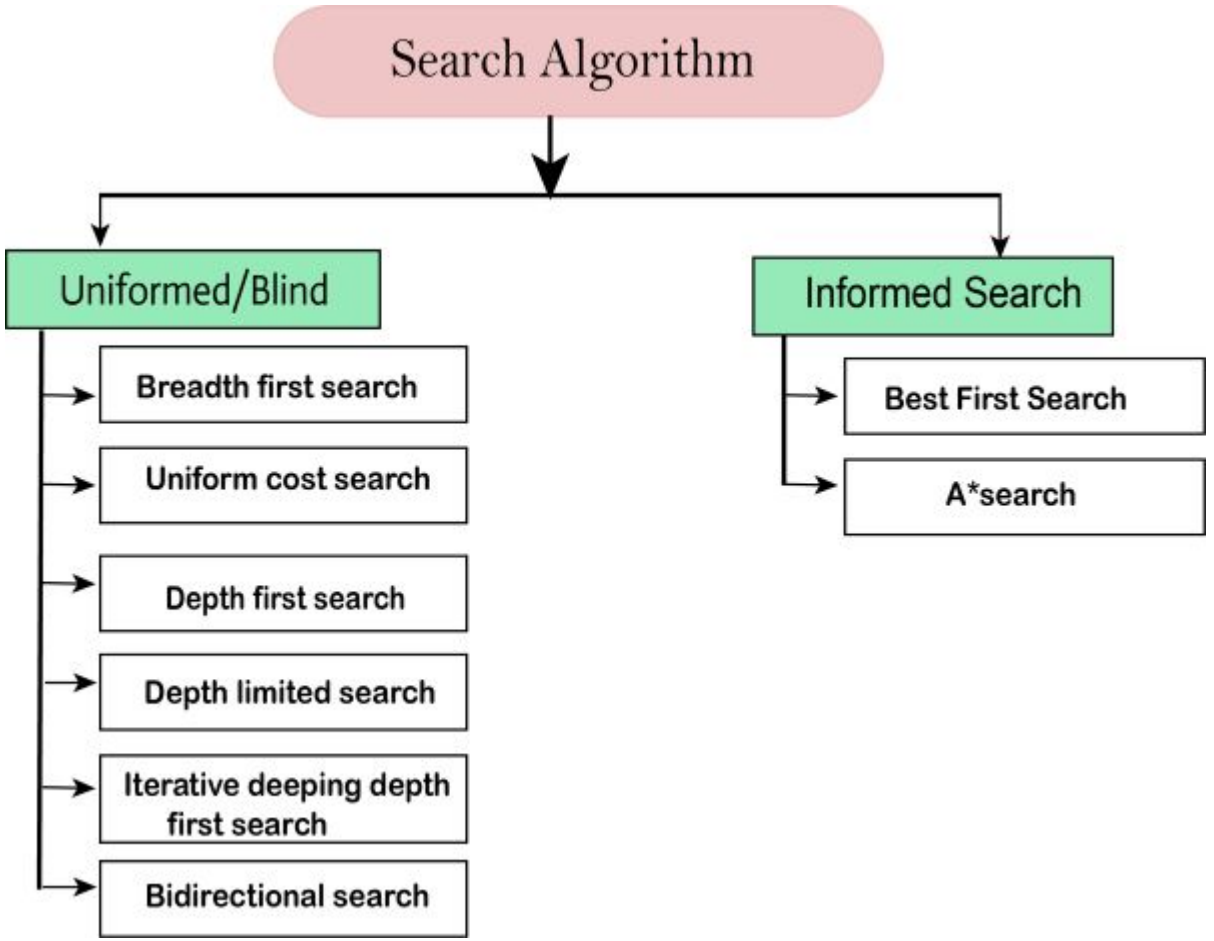
- **Completeness**: Is the algorithm guaranteed to find a solution if there exist one?
- **Optimality**: Does the algorithm find the optimal solution?
- **Time complexity**: How long does it take for the algorithm to find a solution?
- **Space complexity**: How much memory is consumed in finding the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


Search Algorithm
Uniformed/Blind
Breadth first search
Uniform cost search
Depth first search
Depth limited search
Iterative deeping depth first search
Bidirectional search
Informed Search
Best First Search
A*search

Uninformed Search methods

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

Breadth-first search

Uniform-cost search

Depth-first search

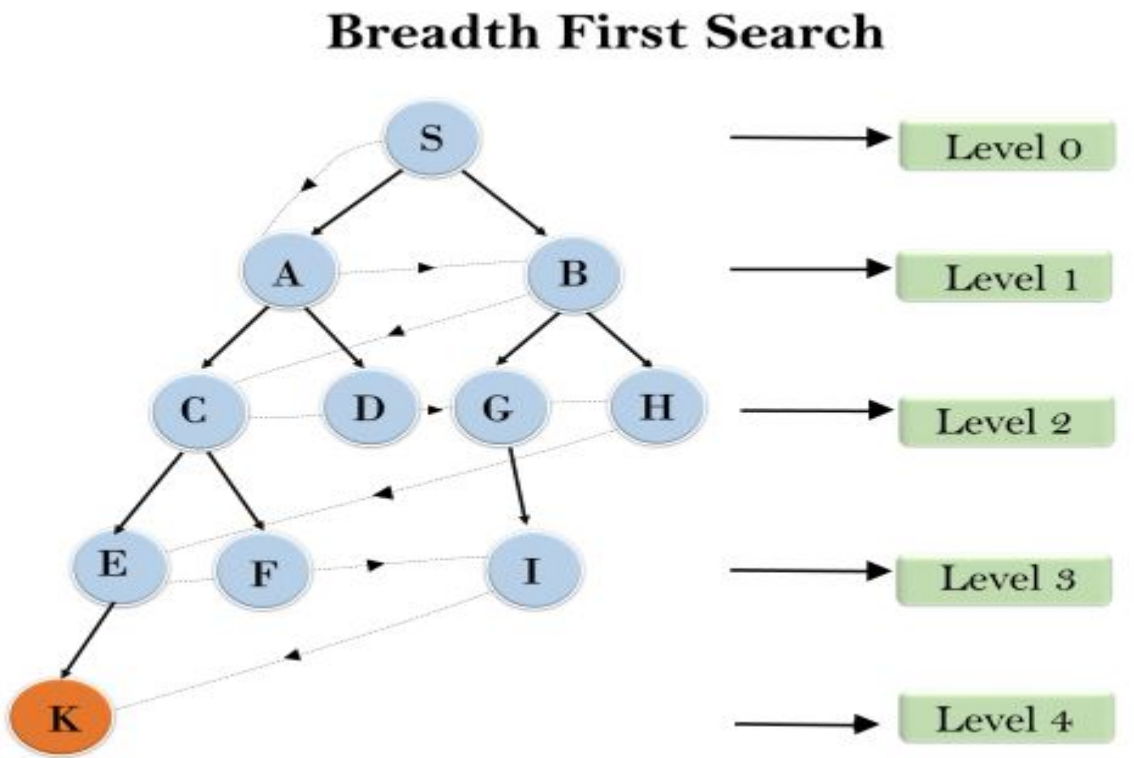
Depth-limited search

Iterative deepening search

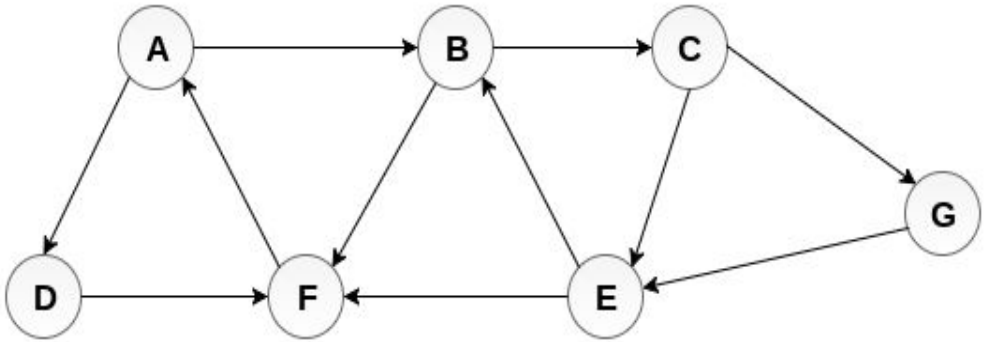
BFS

- 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 traversal of a graph produces a **spanning tree** as final result.

BFS - Example

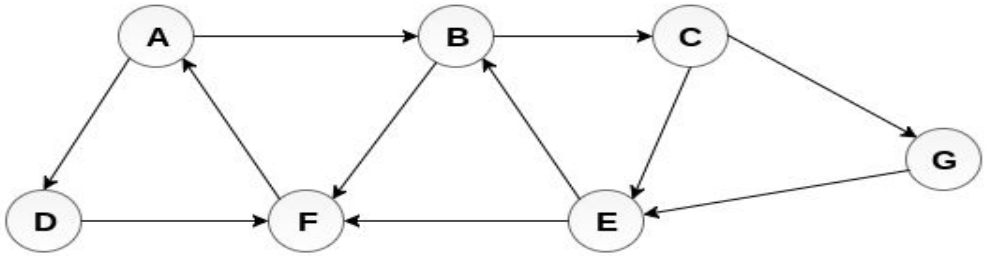

Breadth First Search
S
A
B
C
D
G
H
E
F
I
K
Level 0
Level 1
Level 2
Level 3
Level 4

BFS - Working



- Minimum Path P can be found by applying breadth first search algorithm that will begin at node A and will end at E.
- The algorithm uses two queues, namely **QUEUE1** and **QUEUE2**. **QUEUE1** holds all the nodes that are to be processed while **QUEUE2** holds all the nodes that are processed and deleted from **QUEUE1**.

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}
```

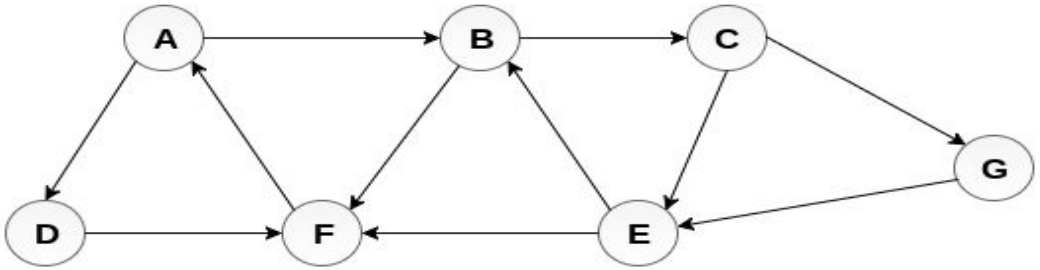
3. Delete the node B from QUEUE1 and insert all its neighbours. Insert node B into QUEUE2.

```
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.

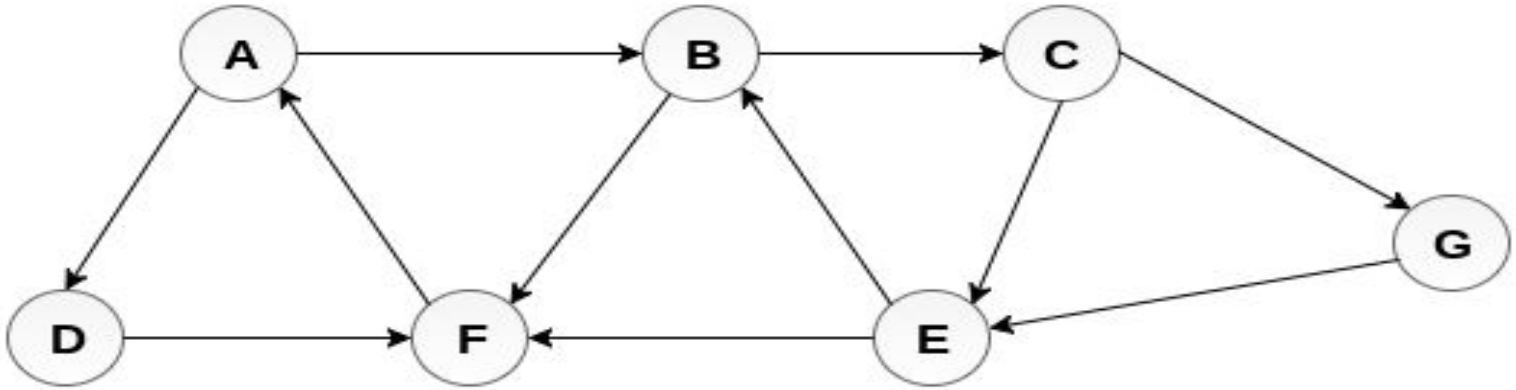
```
QUEUE1 = {E, G}
QUEUE2 = {A, B, D, C, F}
```

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.

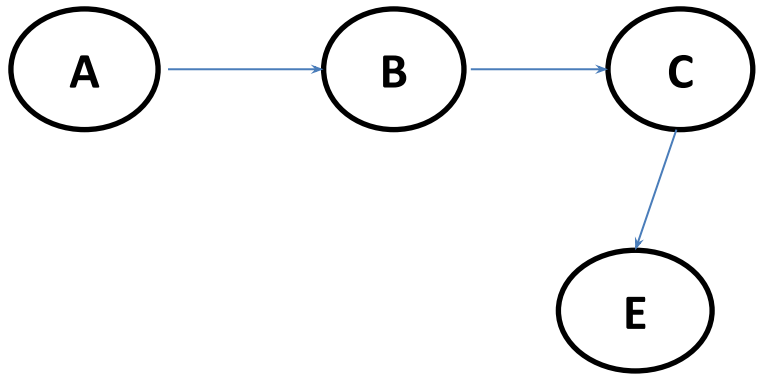
```
QUEUE1 = {G}
QUEUE2 = {A, B, D, C, F,  E}
```

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

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



SPANNING TREE



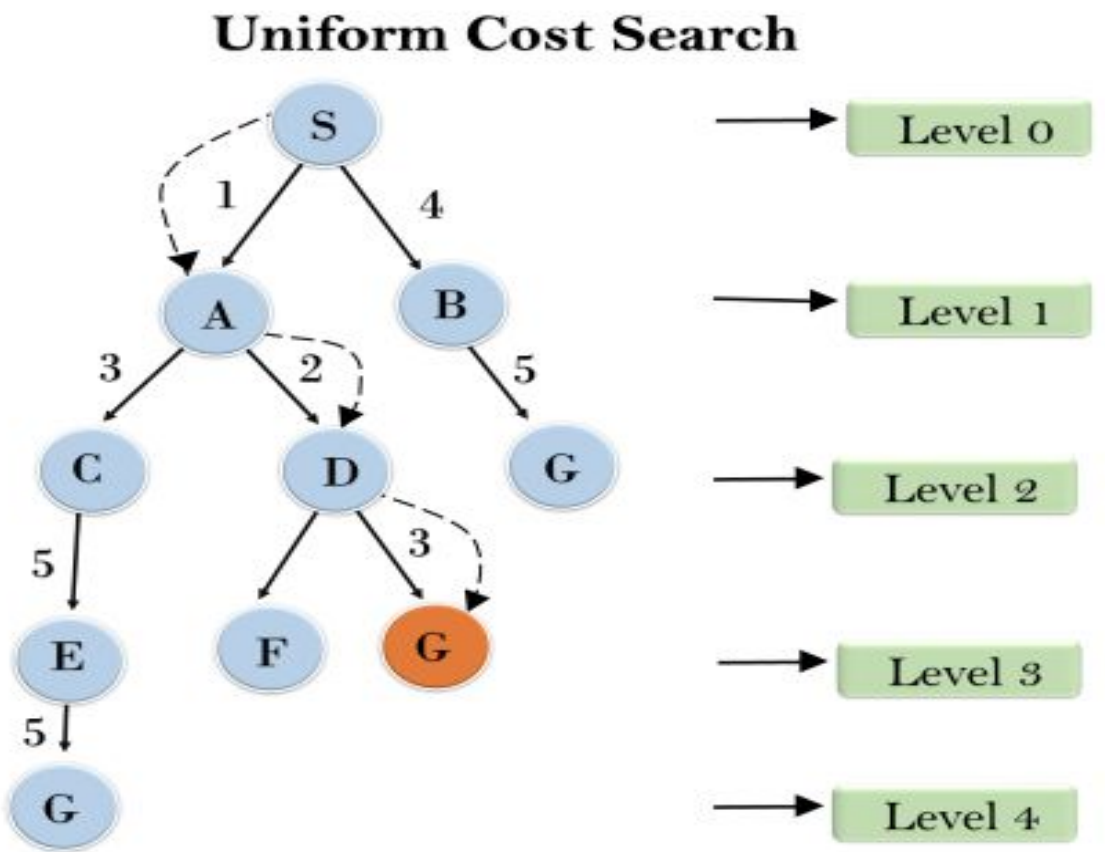
BFS – Time and Space

- **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.

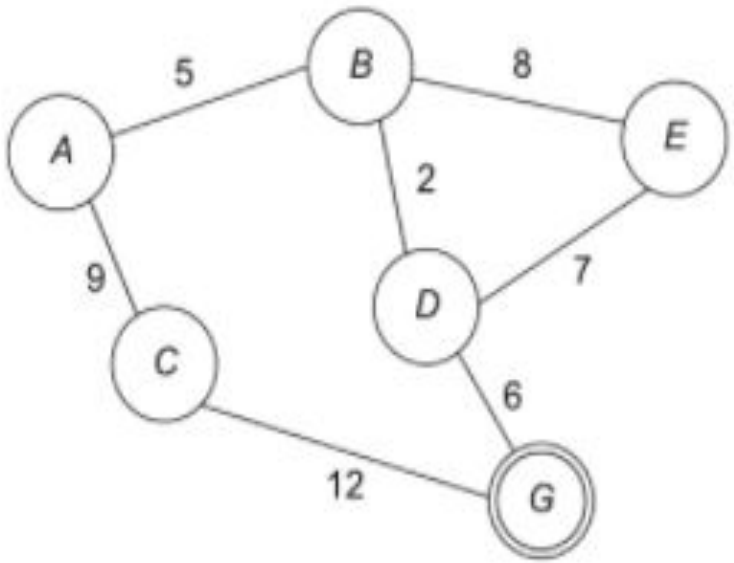
Uniform Cost search

- 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.

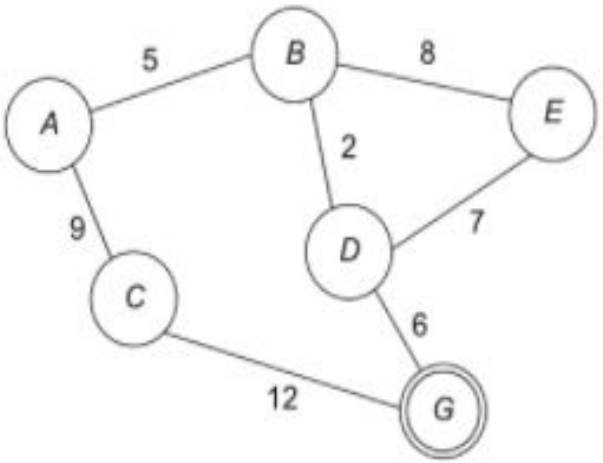
Uniform Cost search - Example



Uniform Cost search- Working



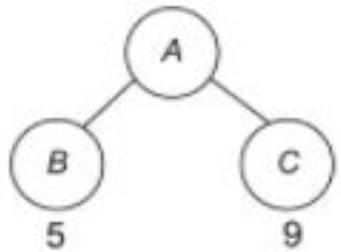
Uniform Cost search - Implementation



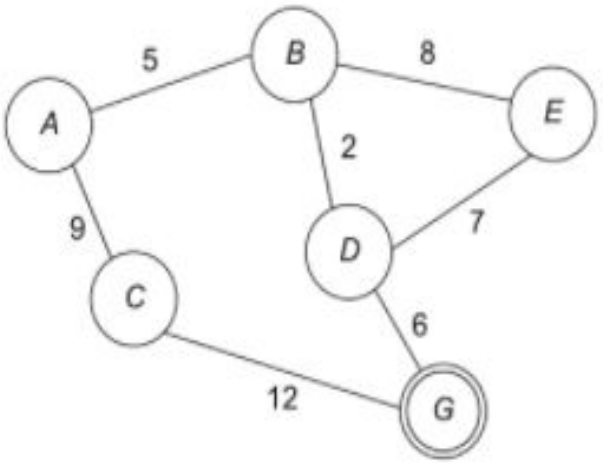
Step 1: Start from *A* — Expanded none

Step 2: Frontier — Expanded none

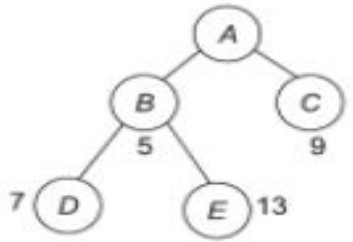
Step 3: Frontier: *B* and *C* — Expanded *A*



Uniform Cost search - Implementation

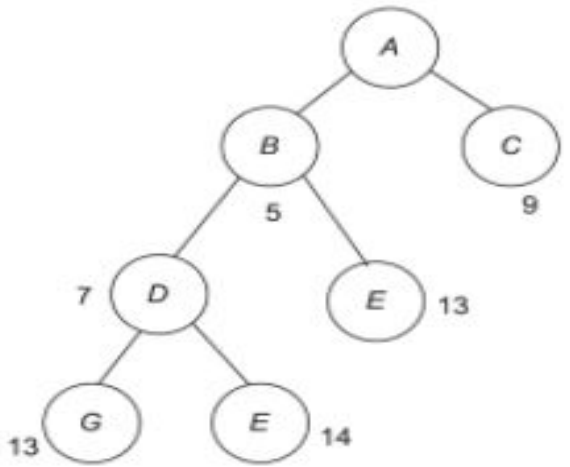


Step 4: Frontier: *D*, *C*, *E* (priority queue) Expanded *A*, *B*



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

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



Uniform Cost search- Time and space

Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is$O(b^{1 + [C^*/\varepsilon]})/$.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $O(b^{1 + [C^*/\varepsilon]})$.

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

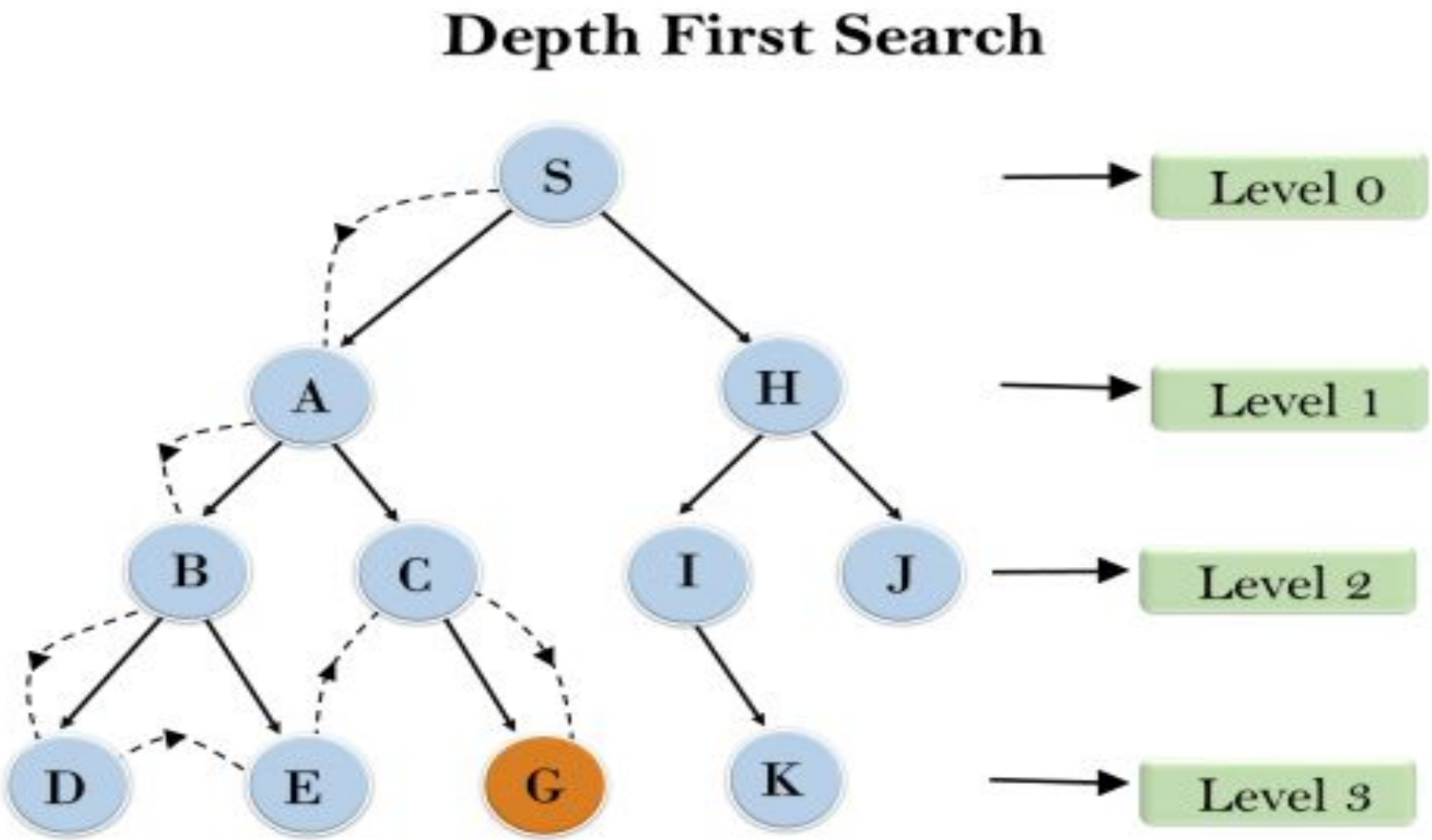
DFS

- 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.

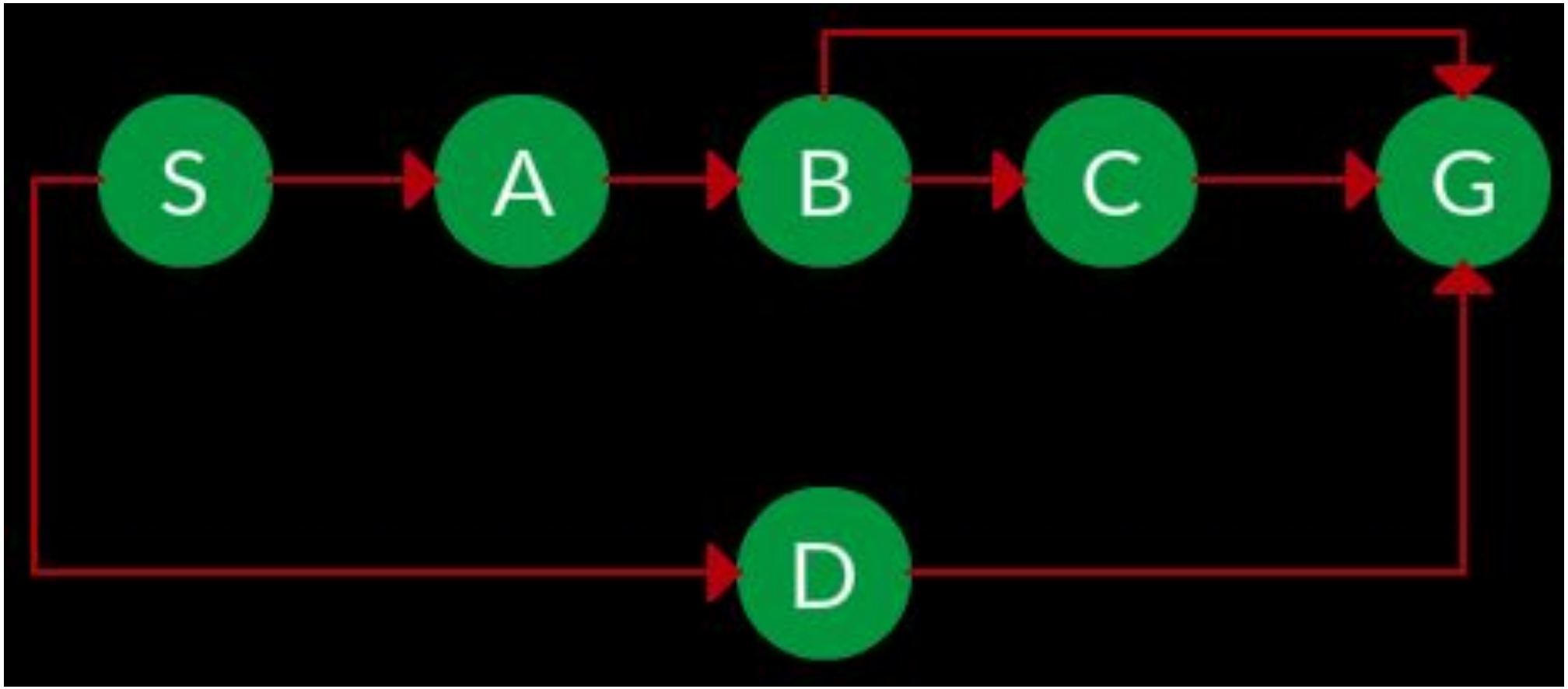
DFS

- 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**.

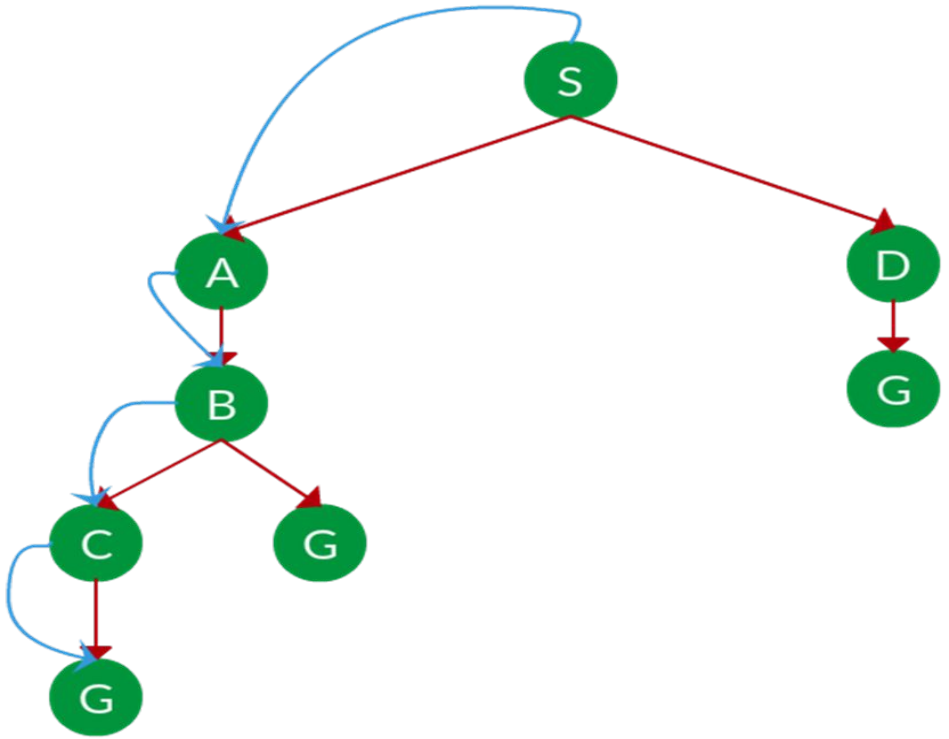
DFS - Example


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.

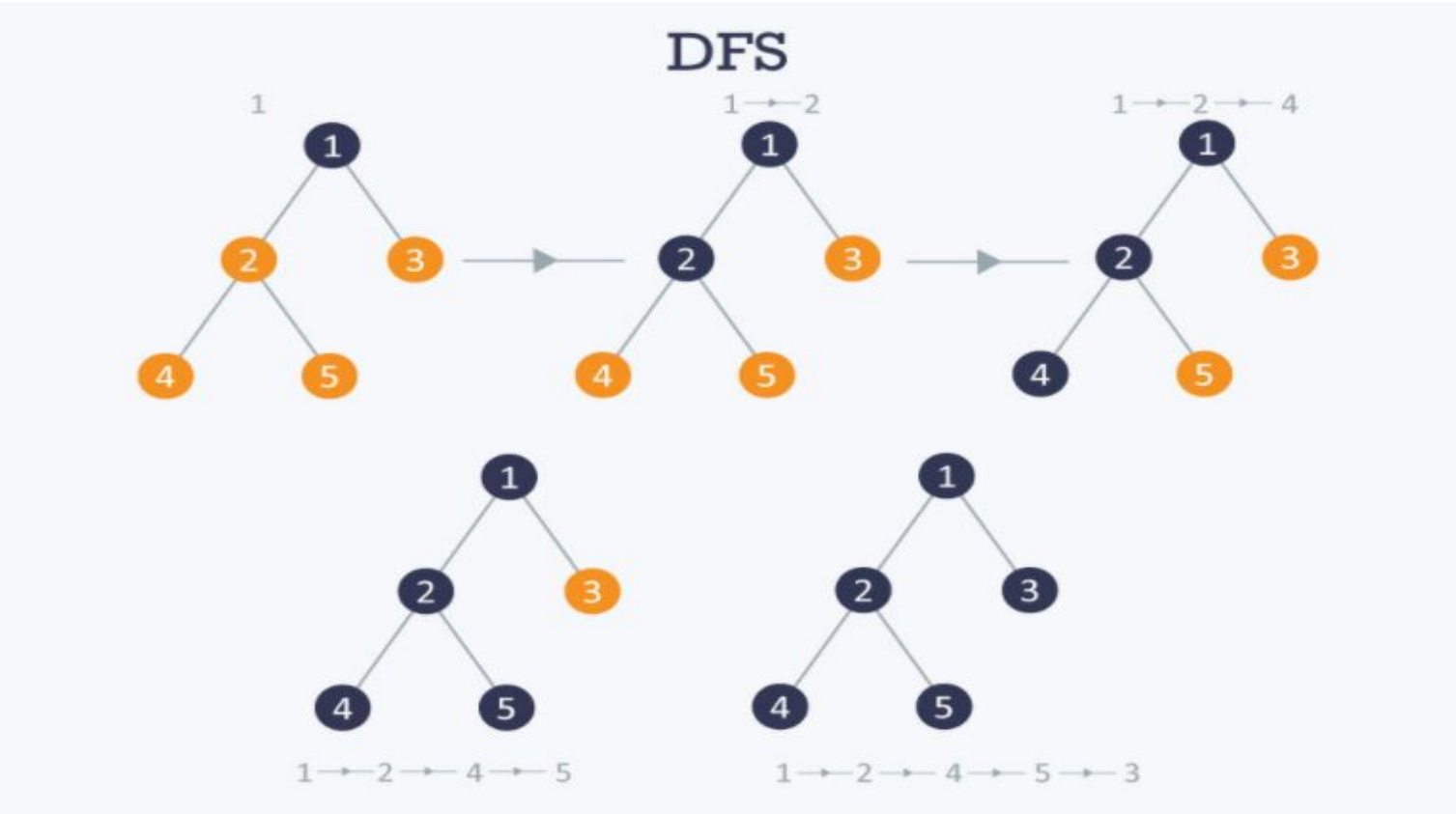
DFS - Iterative

```
    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
```

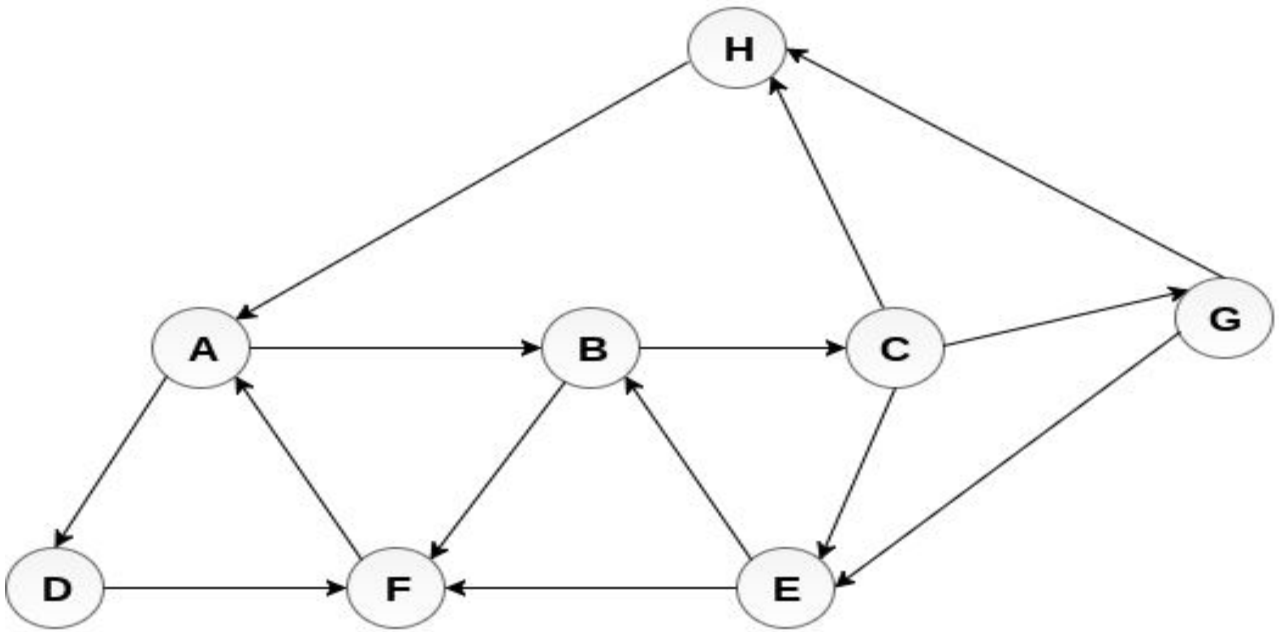
DFS - Recursive

```
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



Adjacency Lists

A : B, D

B : C, F

C : E, G, H

G : E, H

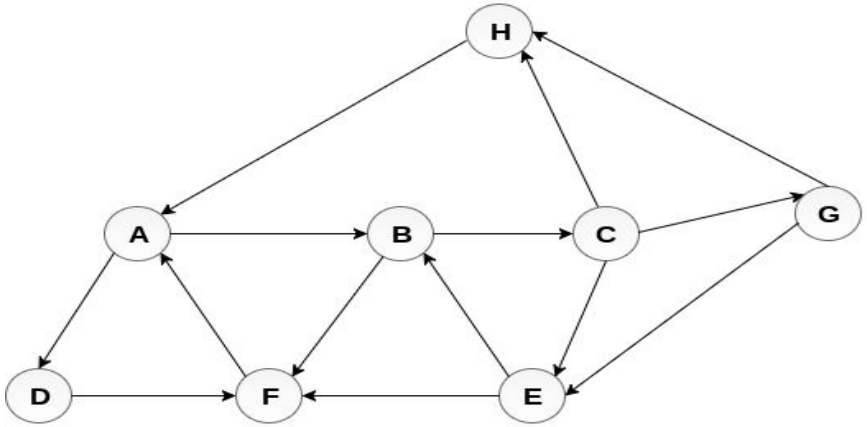
E : B, F

F : A

D : F

H : A

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.

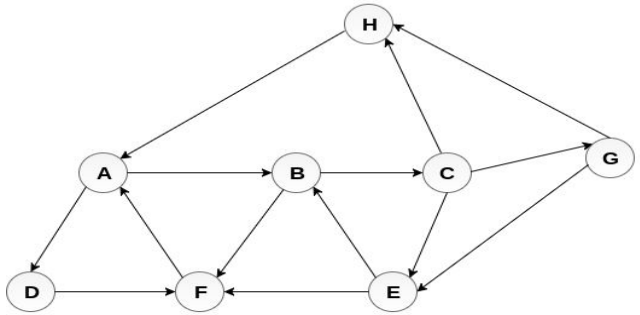
> Print H
> STACK : A

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-first vs. Breadth-first

Advantages of depth-first:

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

Advantages of breadth-first:

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

Disadvantages of depth-first:

- Can sometimes fail to find a solution;
- Not guaranteed to find an *optimal* solution;
- Can take a lot longer to find a solution.

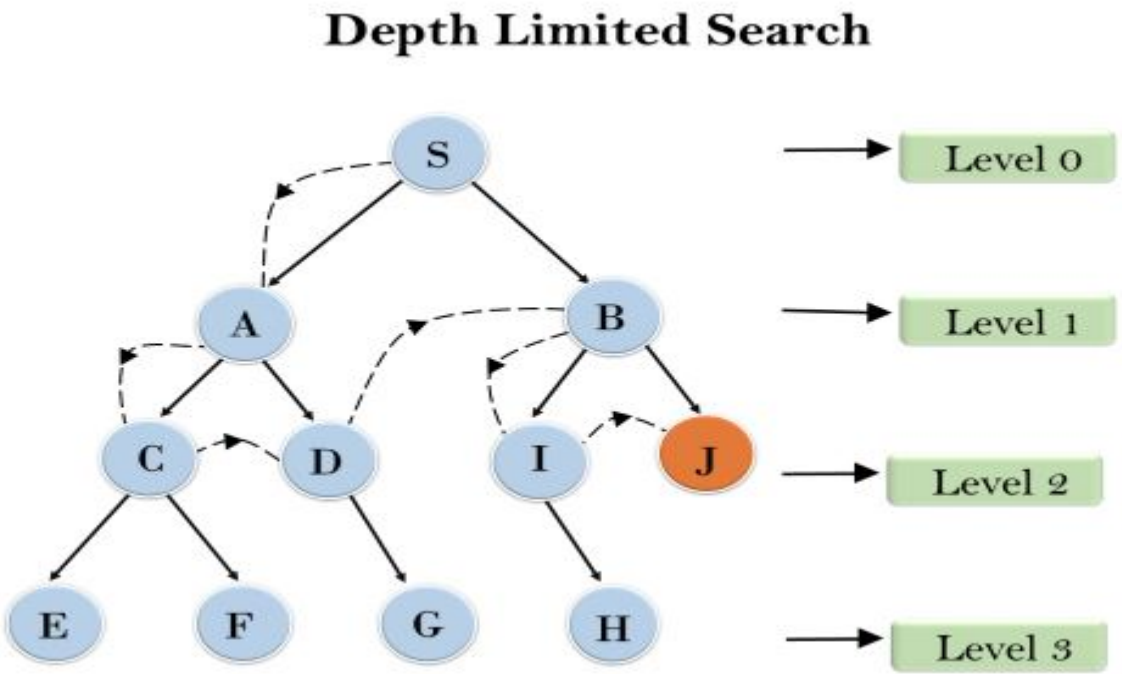
Disadvantages of breadth-first:

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

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.

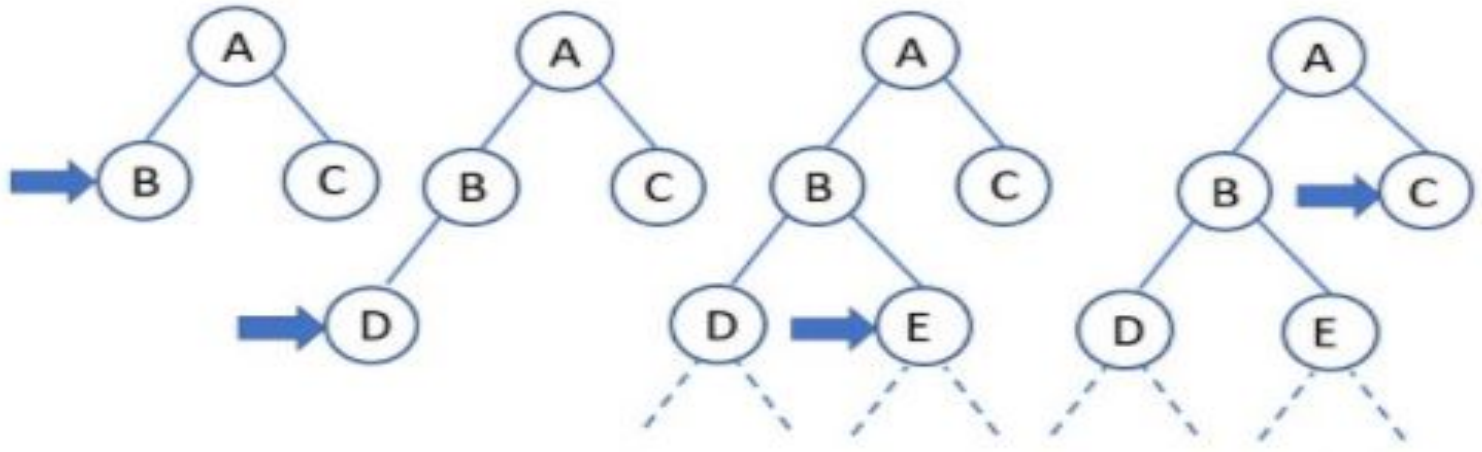
DLS


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 - Working



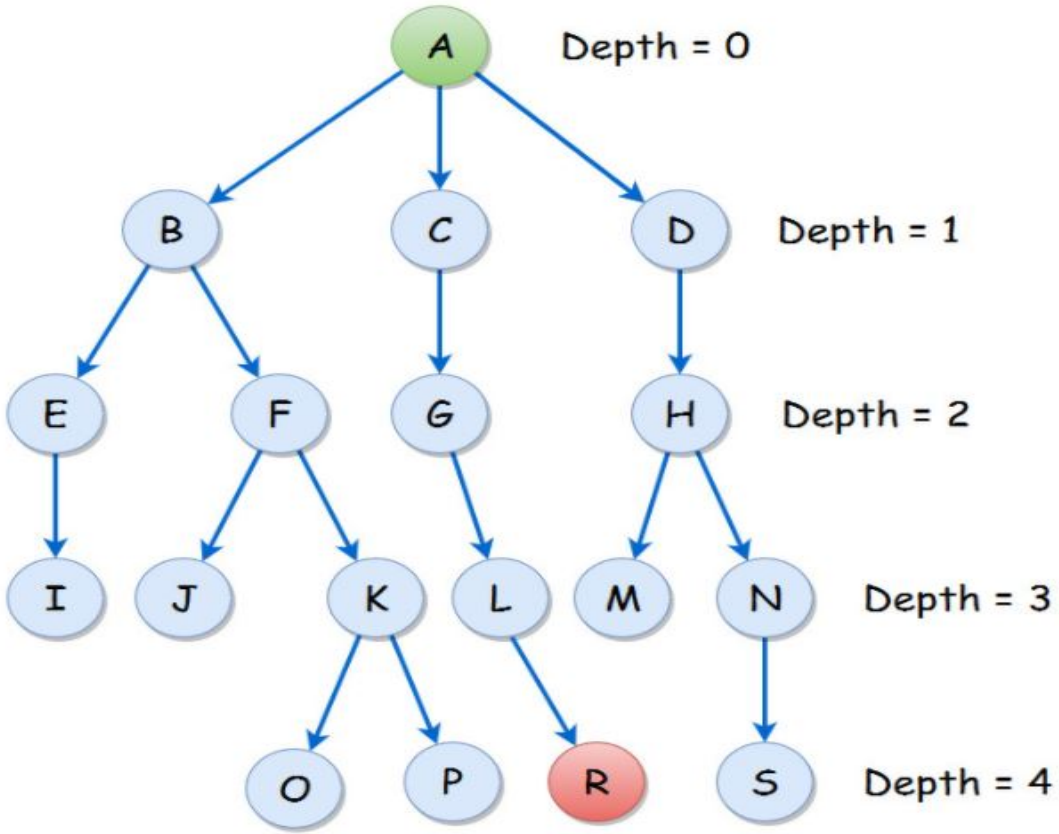
DLS – Time and Space

- **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)

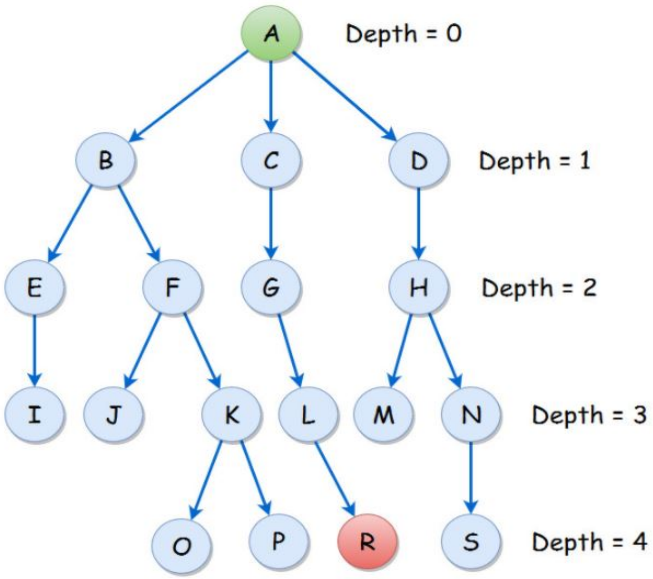
- A search algorithm which suffers neither the drawbacks of breadth-first nor depth-first search on trees is depth-first iterative-deepening
- **IDDFS** combines depth-first search's space-efficiency and breadth-first search's fast search (for nodes closer to root).
- Iterative deepening depth first 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

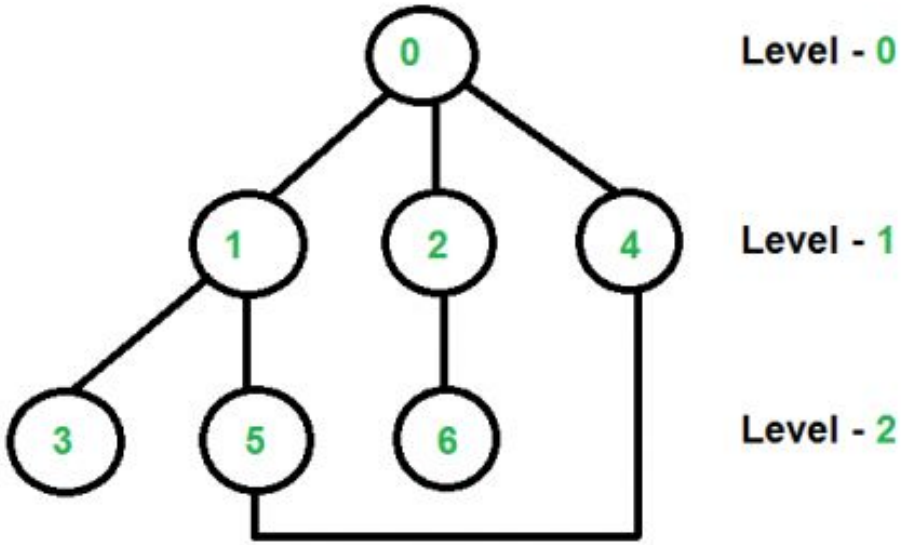
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ cutoff **then return** *result*



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

Completeness:

This algorithm is complete is ifthe branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is $\mathbf{O(b^d)}$.

Space Complexity:

The space complexity of IDDFS will be **O(bd)**.

Optimal:

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

A comparison table between DFS, BFS and IDDFS

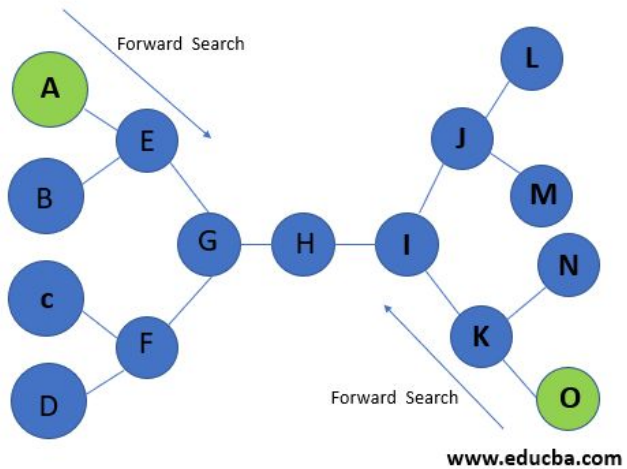
	Time Complexity	Space Complexity	When to Use ?
DFS	$O(b^d)$	O (d)	=> Don't care if the answer is closest to the starting vertex/root. => When graph/tree is not very big/infinite.
BFS	$O(b^d)$	$O(b^d)$	=> When space is not an issue => When we do care/want the closest answer to the root.
IDDFS	$O(b^d)$	O(bd)	=> You want a BFS, you don't have enough memory, and somewhat slower performance is accepted. In short, you want a BFS + DFS.

Bidirectional Search

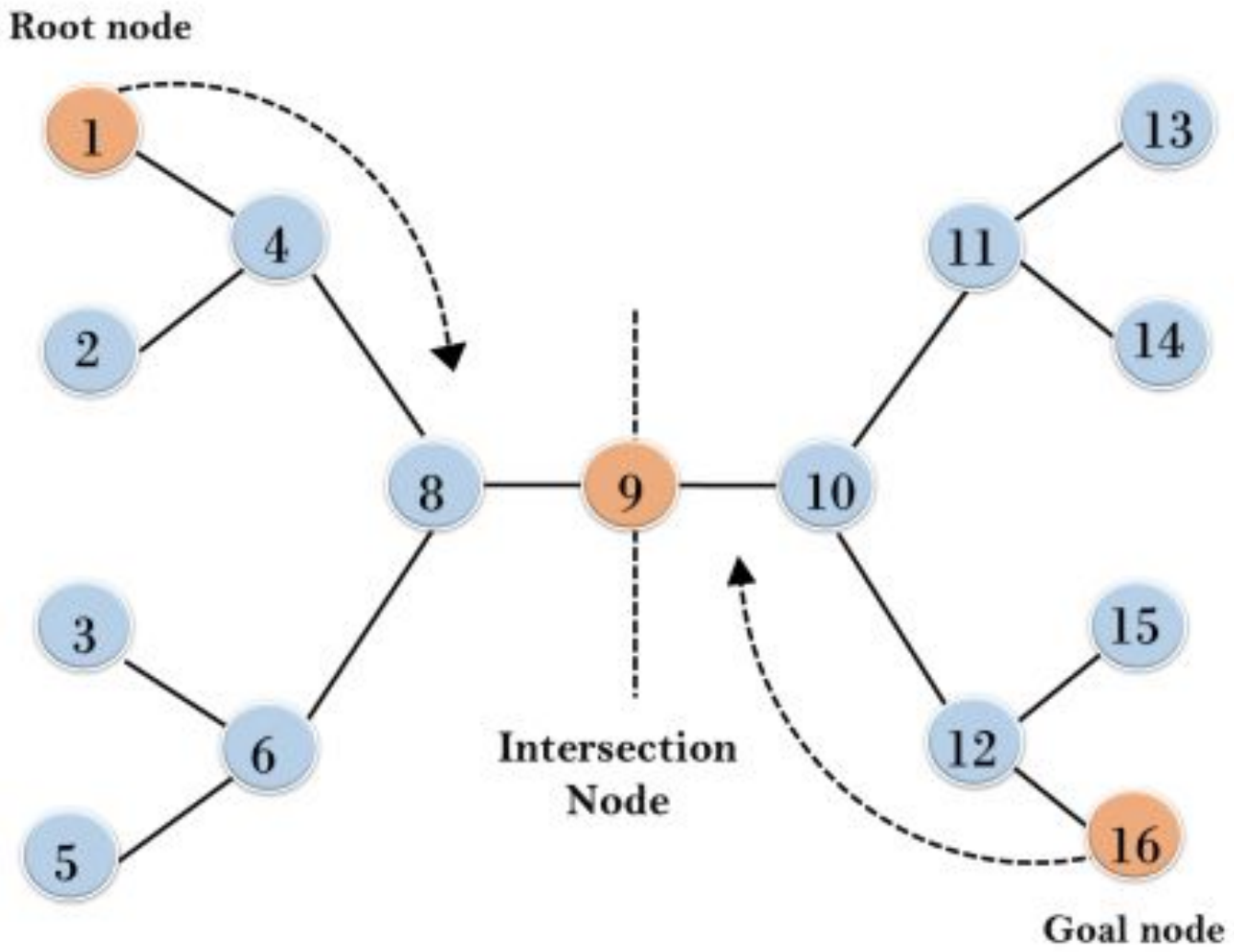
- Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node.
- Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from 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.

Bidirectional Search

- The bidirectional search algorithm works on a directed graph to find the shortest path between the source(initial node) to the goal node. The two searches will start from their respective places and the algorithm stops when the two searches meet at a node.



Bidirectional Search



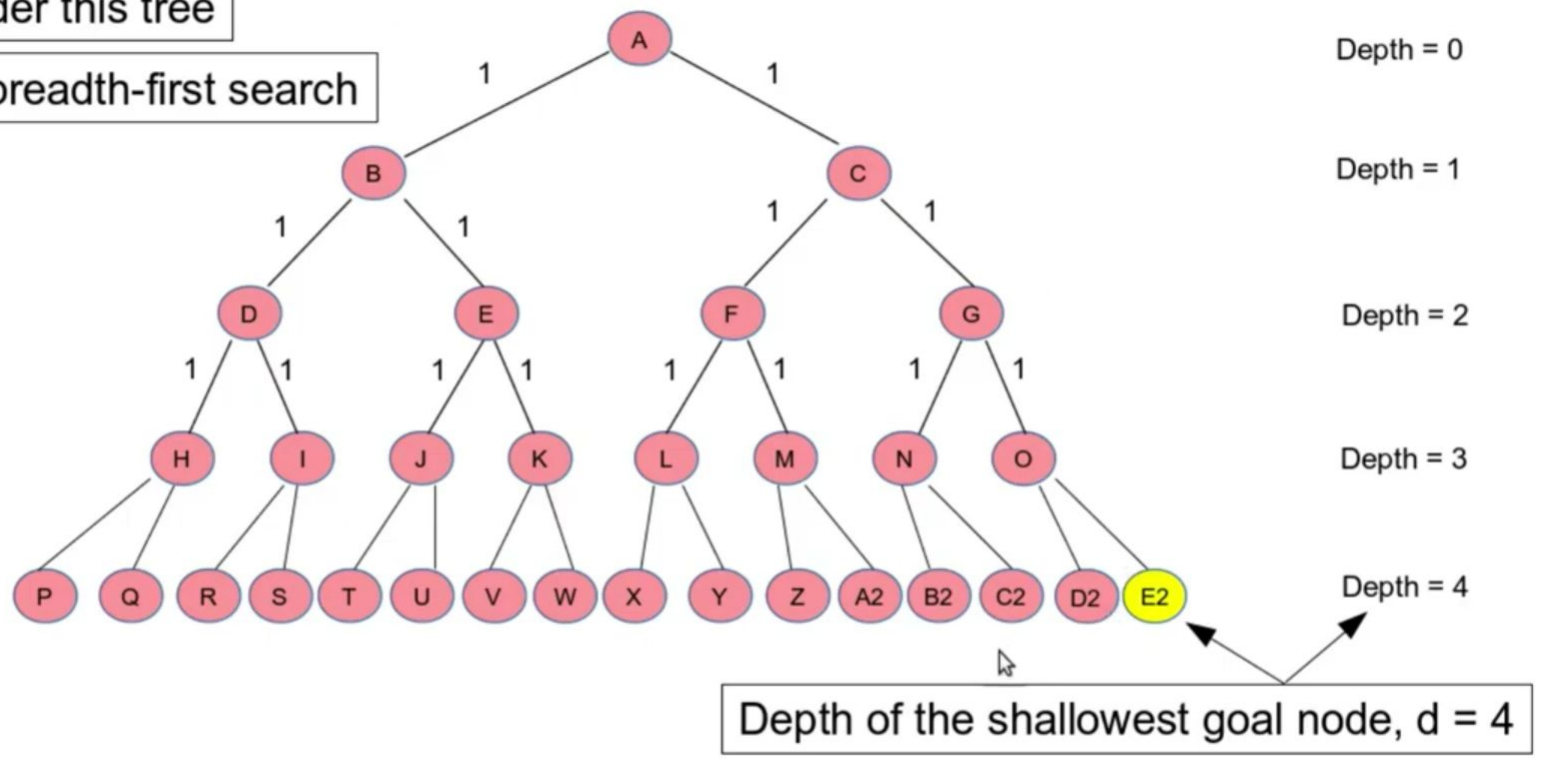
Example

- 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 - Working

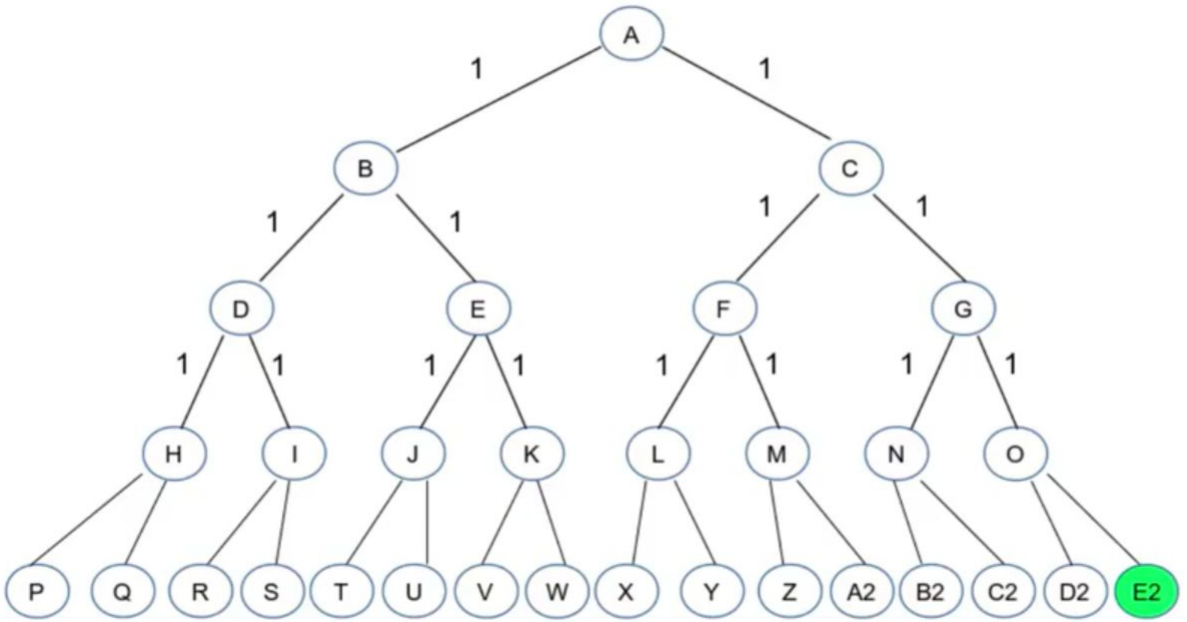
Let us consider this tree

Let us perform breadth-first search



By the time we reach the goal node E2, all these nodes will be in the explored set list

Now let us perform bidirectional search for this same tree



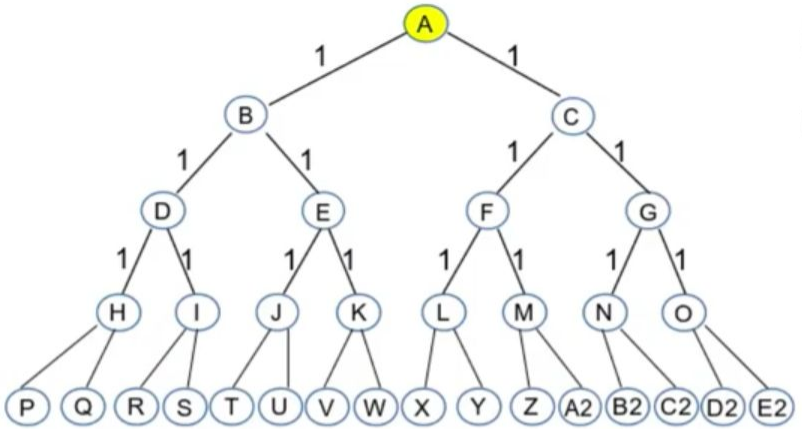
Depth = 0

Depth = 1

Depth = 2

Depth = 3

Depth = 4



Depth = 0

Depth = 1

Depth = 2

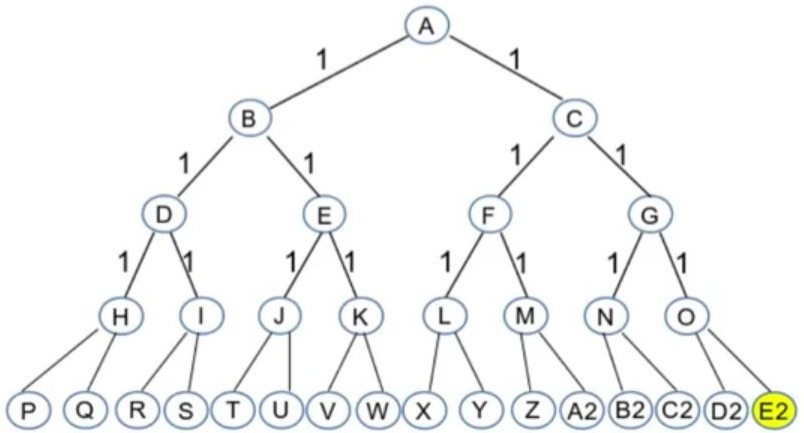
Depth = 3

Depth = 4

Forward search with A as the start node

Fringe list = [A]

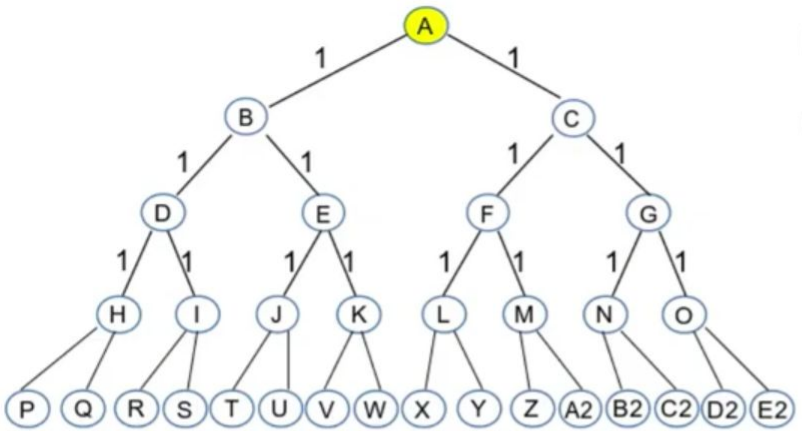
Explored set = [ ]



Backward search with E2 as the start node

Fringe list = [E2]

Explored set = [ ]



Depth = 0

Depth = 1

Depth = 2

Depth = 3

Depth = 4

Forward search with A as the start node

Fringe list = [A]

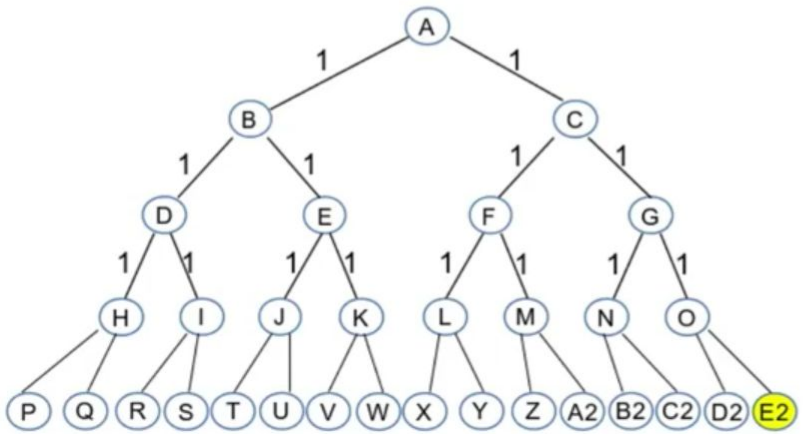
Explored set = [ ]

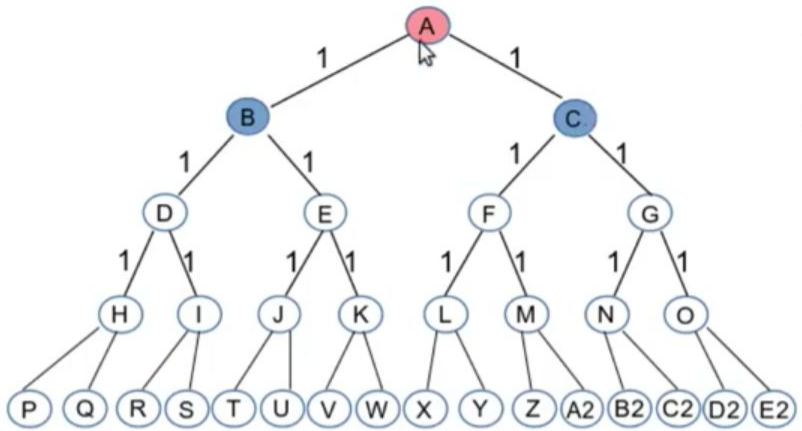
No common elements – goal not reached yet

Backward search with E2 as the start node

Fringe list = [E2]

Explored set = [ ]





Depth = 0

Depth = 1

Depth = 2

Depth = 3

Depth = 4

Forward search with A as the start node

Fringe list = [B, C]

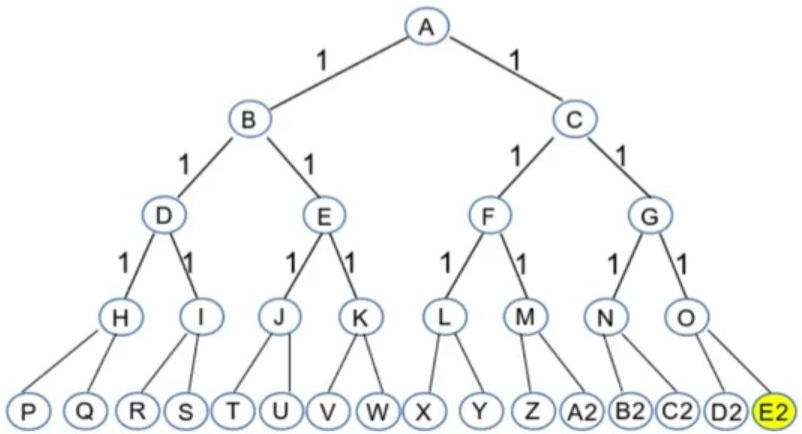
Explored set = [A]

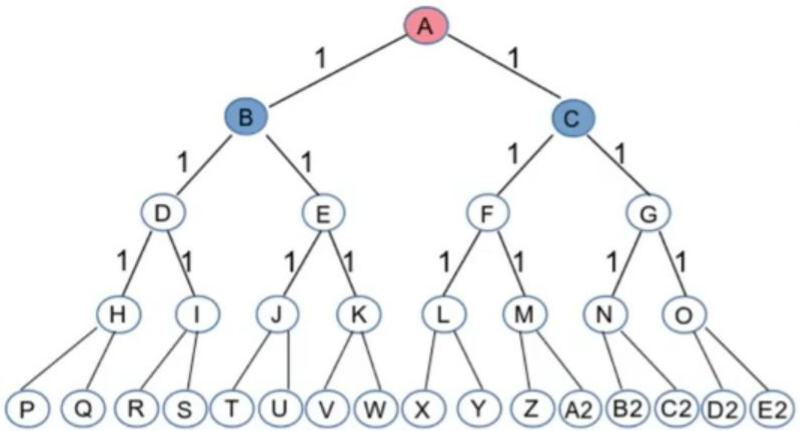
No common elements – goal not reached yet

Backward search with E2 as the start node

Fringe list = [E2]

Explored set = [ ]





Depth = 0

Depth = 1

Depth = 2

Depth = 3

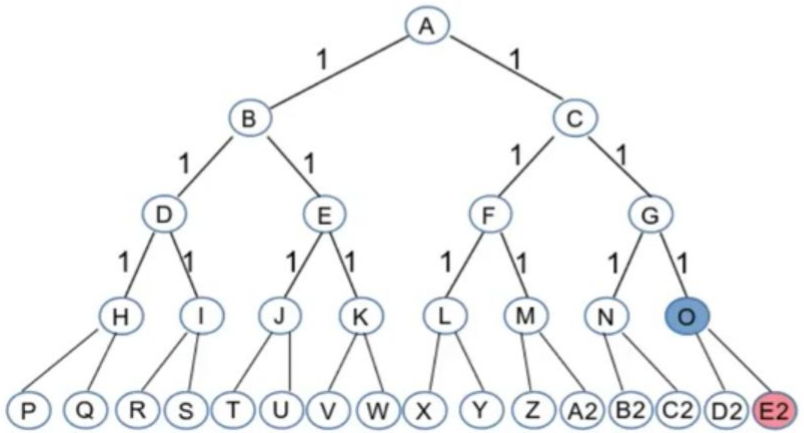
Depth = 4

Forward search with A as the start node

Fringe list = [B, C]

Explored set = [A]

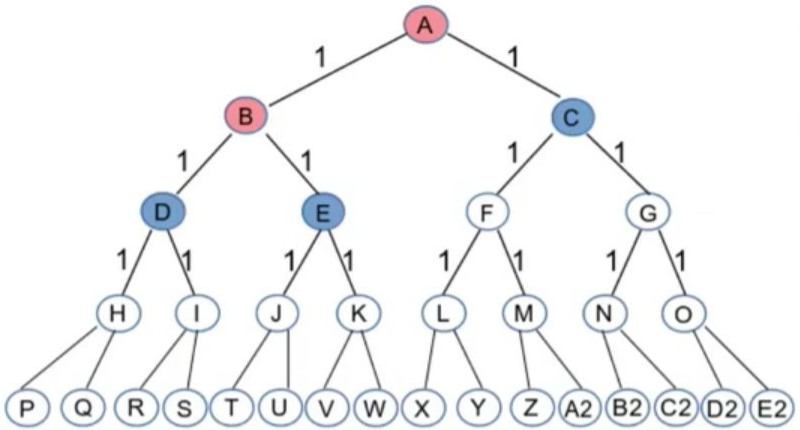
No common elements – goal not reached yet



Backward search with E2 as the start node

Fringe list = [O]

Explored set = [E2]



Depth = 0

Depth = 1

Depth = 2

Depth = 3

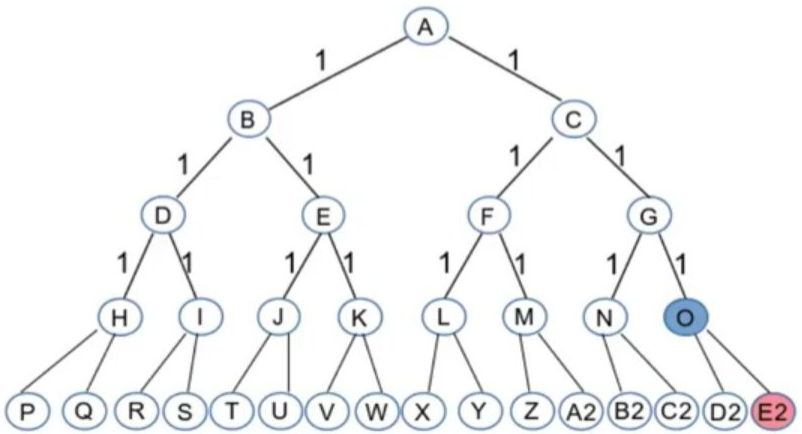
Depth = 4

Forward search with A as the start node

Fringe list = [C, D, E]

Explored set = [A, B]

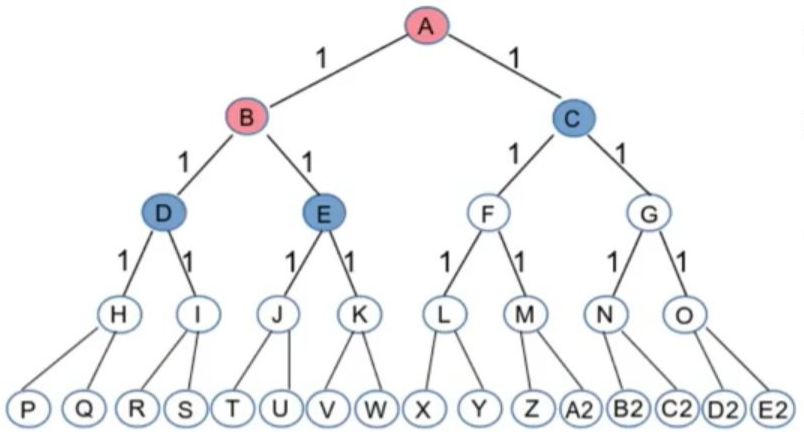
No common elements – goal not reached yet



Backward search with E2 as the start node

Fringe list = [O]

Explored set = [E2]



Depth = 0

Depth = 1

Depth = 2

Depth = 3

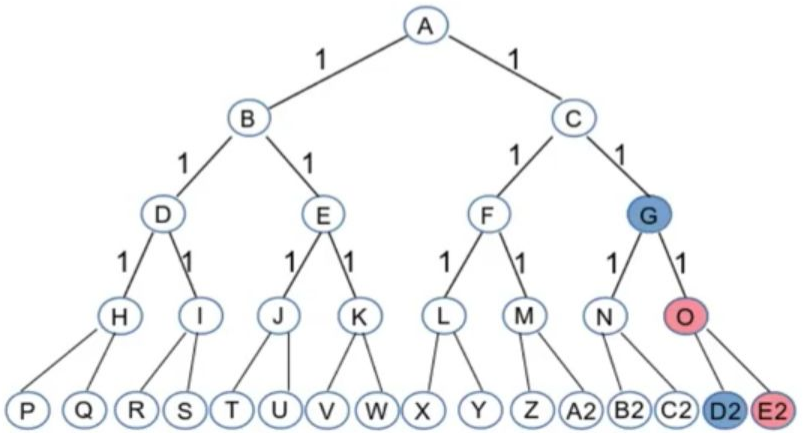
Depth = 4

Forward search with A as the start node

Fringe list = [C, D, E]

Explored set = [A, B]

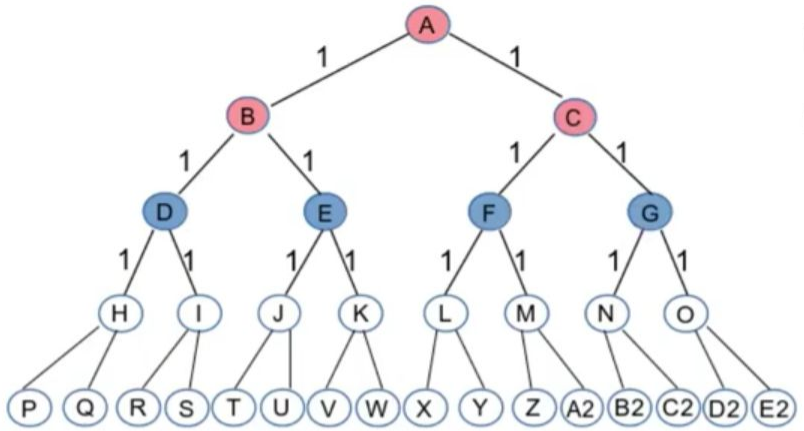
No common elements – goal not reached yet



Backward search with E2 as the start node

Fringe list = [G, D2]

Explored set = [E2, O]



Depth = 0

Depth = 1

Depth = 2

Depth = 3

Depth = 4

Forward search with A as the start node

Fringe list = [D, E, F, G]

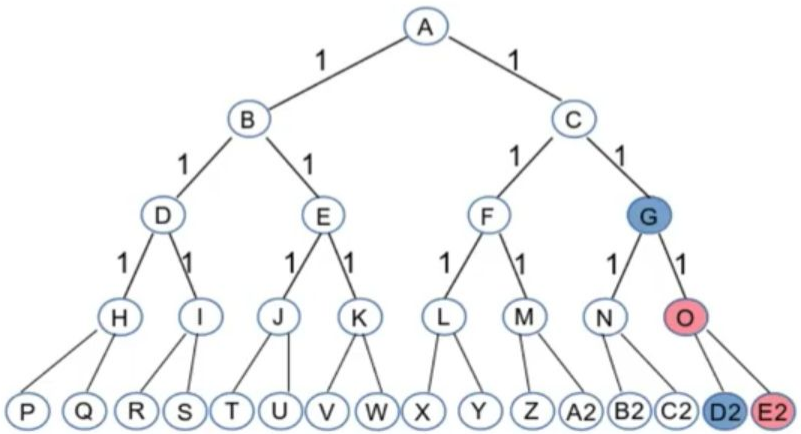
Explored set = [A, B, C]

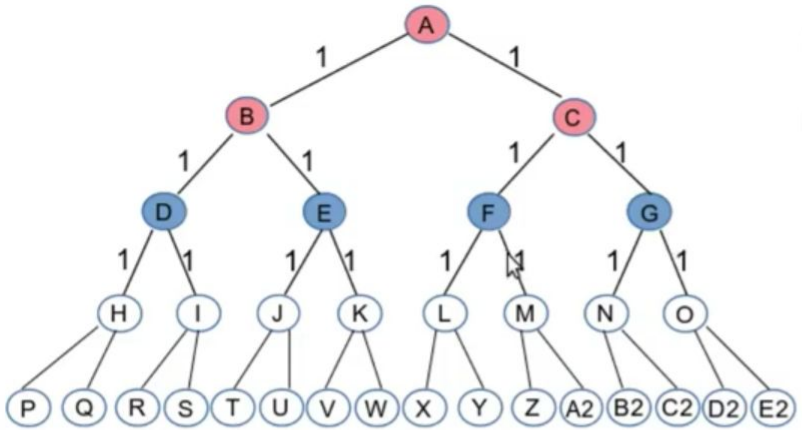
Common elements – goal found

Backward search with E2 as the start node

Fringe list = [G, D2]

Explored set = [E2, O]





Depth = 0

Depth = 1

Depth = 2

Depth = 3

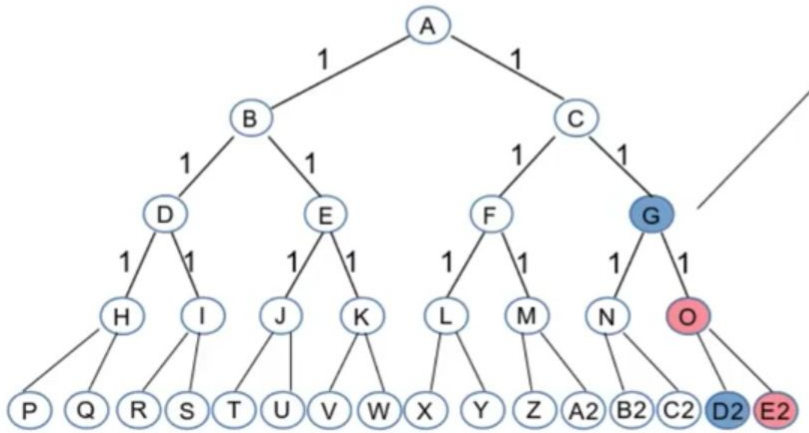
Depth = 4

Forward search with A as the start node

Fringe list = [D, E, F, G]

Explored set = [A, B, C]

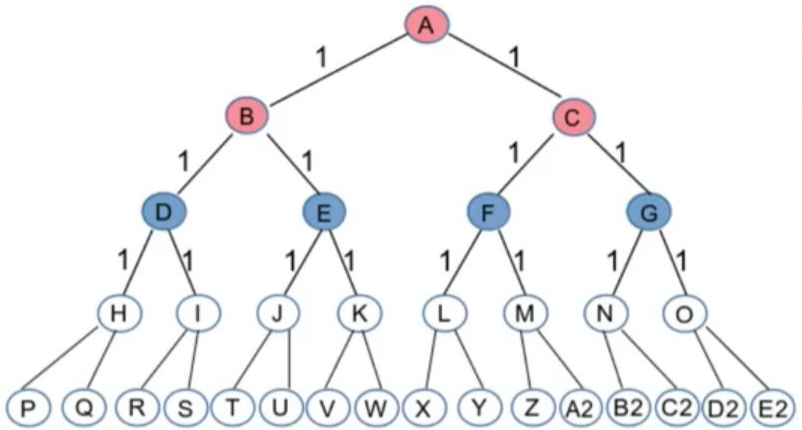
Search proceeded only upto depth 2



Backward search with E2 as the start node

Fringe list = [G, D2]

Explored set = [E2, O]

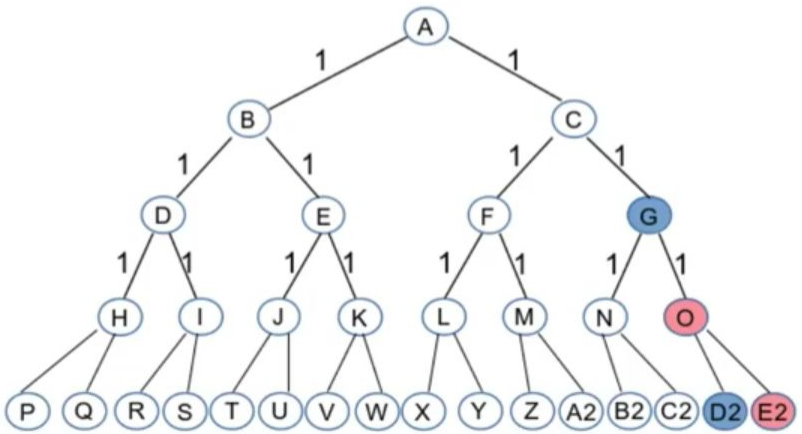


In general, if ***d*** is the depth of the shallowest goal node

Both forward and backward search will meet after each search expands **d/2** levels

Which means each search in the worst case generates $O(b^{d/2})$ nodes

Forward search generates $O(b^{d/2})$ nodes



Backward search generates $O(b^{d/2})$ nodes

Total nodes generated thus is = $O(b^{d/2}) + O(b^{d/2})$ = $O(b^{d/2})$ nodes

Space and time complexity of bidirectional search = $O(b^{d/2})$

The amount of space required can be reduced by half by using iterative deepening for one of the two search. But the frontier from at least one the search has to be in the memory

Let b = 10 and d = 6

Both forward and backward search will expand nodes upto depth d/2 = 3

Let breadth first search be used for both forward and backward search

Forward search will then generate = 10 + 100 + 1000 = 1110 nodes

Backward search will then generate = 10 + 100 + 1000 = 1110 nodes

Total nodes generated by bidirectional search = 1110 + 1110 = 2220 nodes

If breadth first search is used instead of bidirectional search, search will proceed till depth 6

Total nodes generated by BFS = 10 + 100 + 1000 + 10000 + 100000 + 1000000 = 1111110

This comparison shows the advantage of bidirectional search over breadth-first search

Performance measures

- **Completeness** – Bidirectional search is complete if BFS is used in both searches.
- **Optimality** – It is optimal if BFS is used for search and paths have uniform cost.
- **Time and Space Complexity** – Time and space complexity is **O(b^{d/2})**

Advantages

- One of the main advantages of bidirectional searches is the speed at which we get the desired results.
- It drastically reduces the time taken by the search by having simultaneous searches.
- It also saves resources for users as it requires less memory capacity to store all the searches.

Disadvantages

- user should be aware of the goal state to use bidirectional search and thereby to decrease its use cases drastically.
- The implementation is another challenge as additional code and instructions are needed to implement this algorithm, and also care has to be taken as each node and step to implement such searches.
- The algorithm must be robust enough to understand the intersection when the search should come to an end or else there's a possibility of an infinite loop.
- It is also not possible to search backwards through all states.

Summary of Blind Search Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Algorithm	Space	Time	Complete	Optimal
BFS	Theta (b^d)	Theta(b^d)	Yes	Yes
DFS	Theta(d)	Theta(b^m)	No	No
UCS	Theta(b^(ceil(C*/e))	Theta(b^d)	Yes	Yes
DLS	Theta(l)	Theta(b^l)	No	No
IDS	Theta(d)	Theta(b^d)	Yes	Yes

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

Informed Search Algorithms

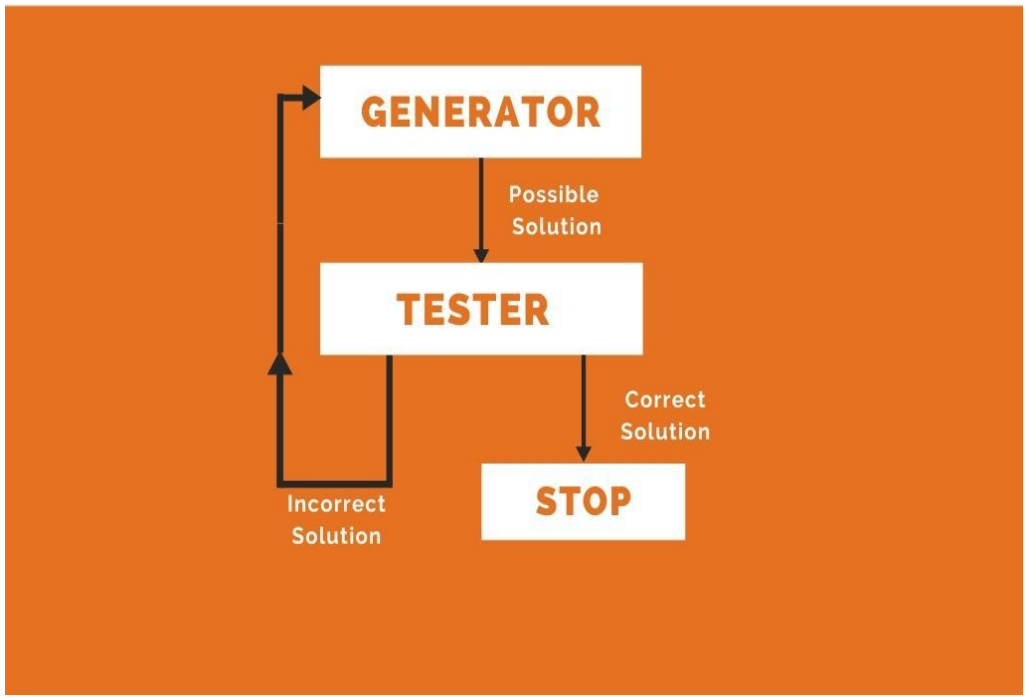
- Generate and Test
- Best-first search
- Greedy best-first search
- A^* search
- Heuristics

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.
- It can be implemented on a search graph rather than a tree

Generate and Test

DIAGRAMMATIC REPRESENTATION



1. Generate a possible solution. For example, generating a particular point in the problem space or generating a path for a start state.

2. Test to see if this is a actual solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.

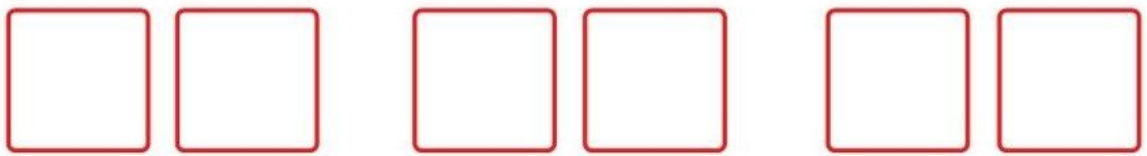
3. If a solution is found, quit. Otherwise go to Step 1

Properties of Good Generators

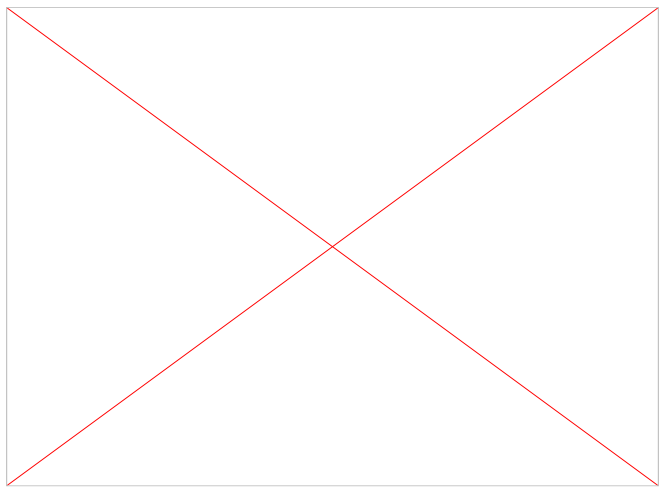
- ***Complete***
- ***Non Redundant***
- ***Informed***

Generate and Test - Example

- Let us take a simple example to understand the importance of a good generator. Consider a pin made up of three 2 digit numbers i.e. the numbers are of the form,



- In this case, one way to find the required pin is to generate all the solutions in a brute force manner for example,



Generate and Test - Example

- The total number of solutions in this case is $(100)^3$ which is approximately 1M.
- Now let's say if we generate 5 solutions every minute. Then the total numbers generated in 1 hour are 5*60=300 and the total number of solutions to be generated are 1M.
- using heuristic function where we have domain knowledge that every number is a prime number between 0-99 then the possible number of solutions are $(25)^3$ which is approximately 15,000.
- We can conclude for here that if we can find a good heuristic then time complexity can be reduced gradually. But in the worst-case time and space complexity will be exponential. It all depends on the generator i.e. better the generator lesser is the time complexity.

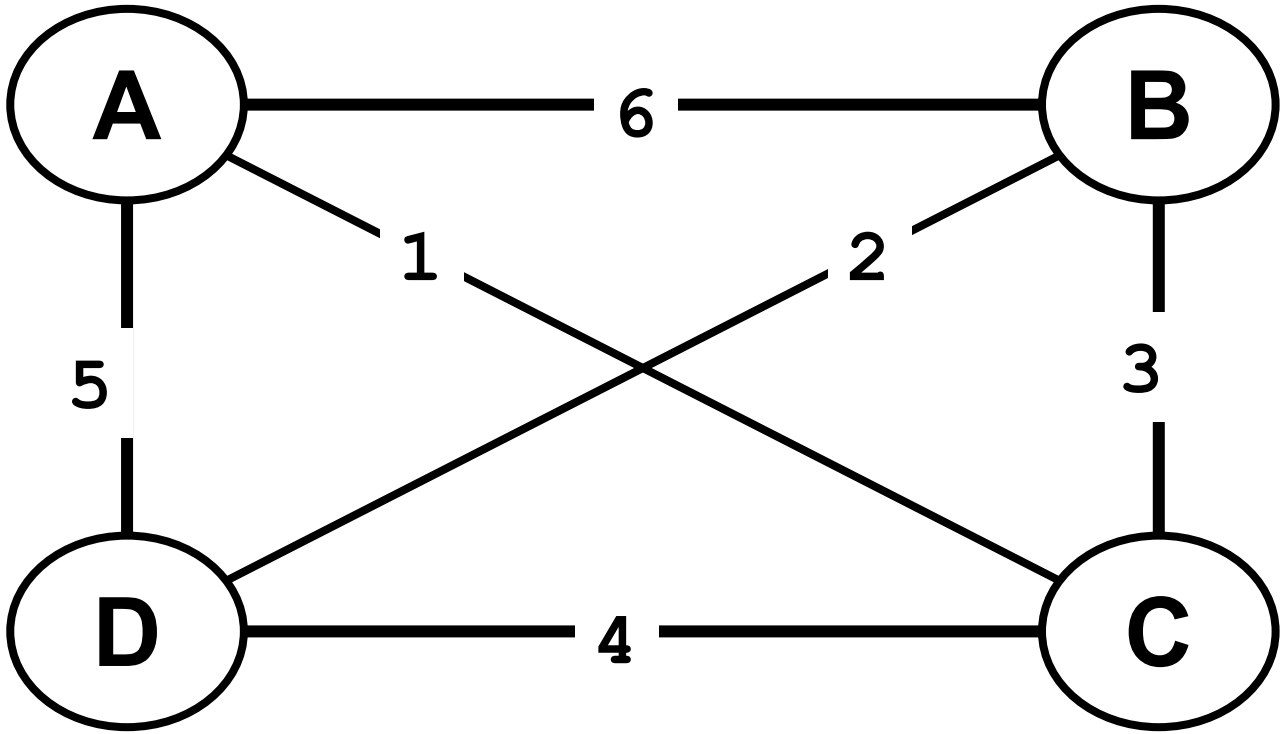
Generate-and-test

Example - Traveling Salesman Problem (TSP)

- Traveler needs to visit *n* cities.
- Know the distance between each pair of cities.
- Want to know the shortest route that visits all the cities once.
- *n=80* will take millions of years to solve exhaustively!

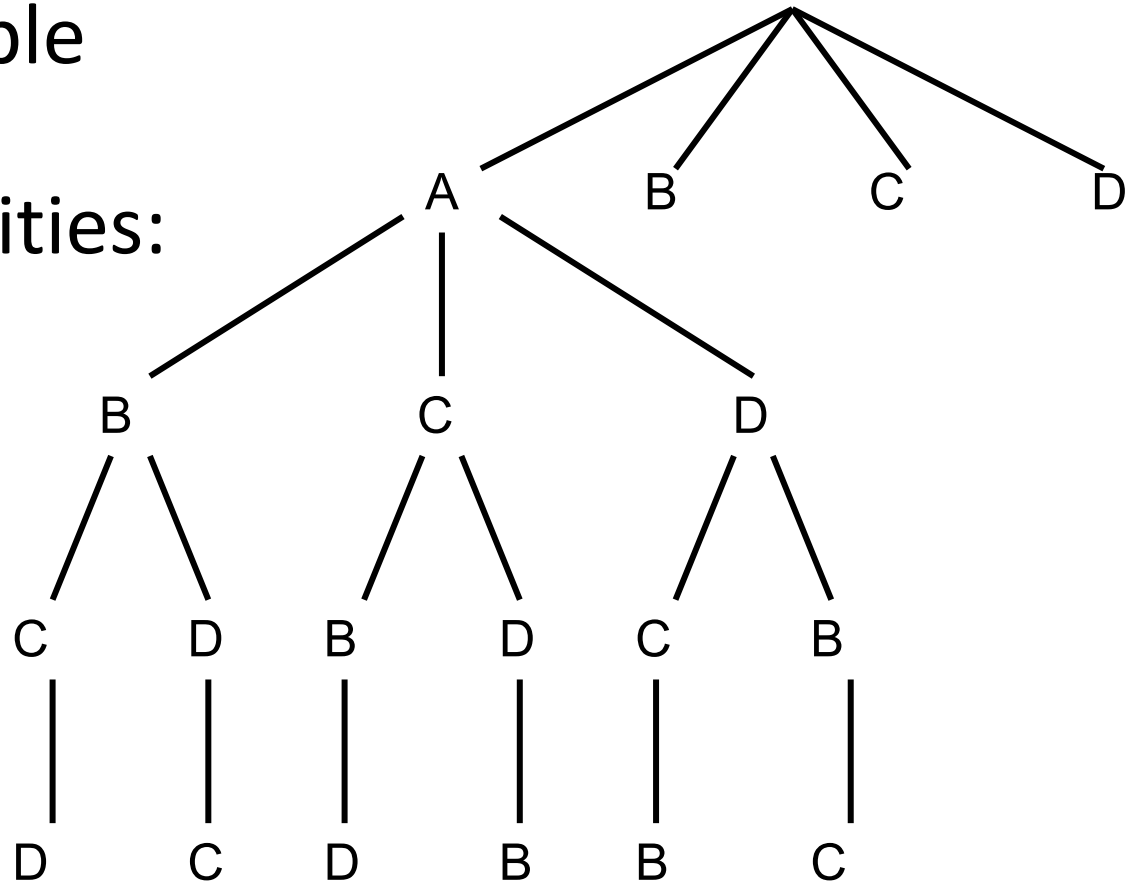
Generate-and-test

TSP Example



Generate-and-test Example

- TSP - generation of possible solutions is done in lexicographical order of cities:
  1. A - B - C - D
  2. A - B - D - C
  3. A - C - B - D
  4. A - C - D - B
  ...



Best First Search

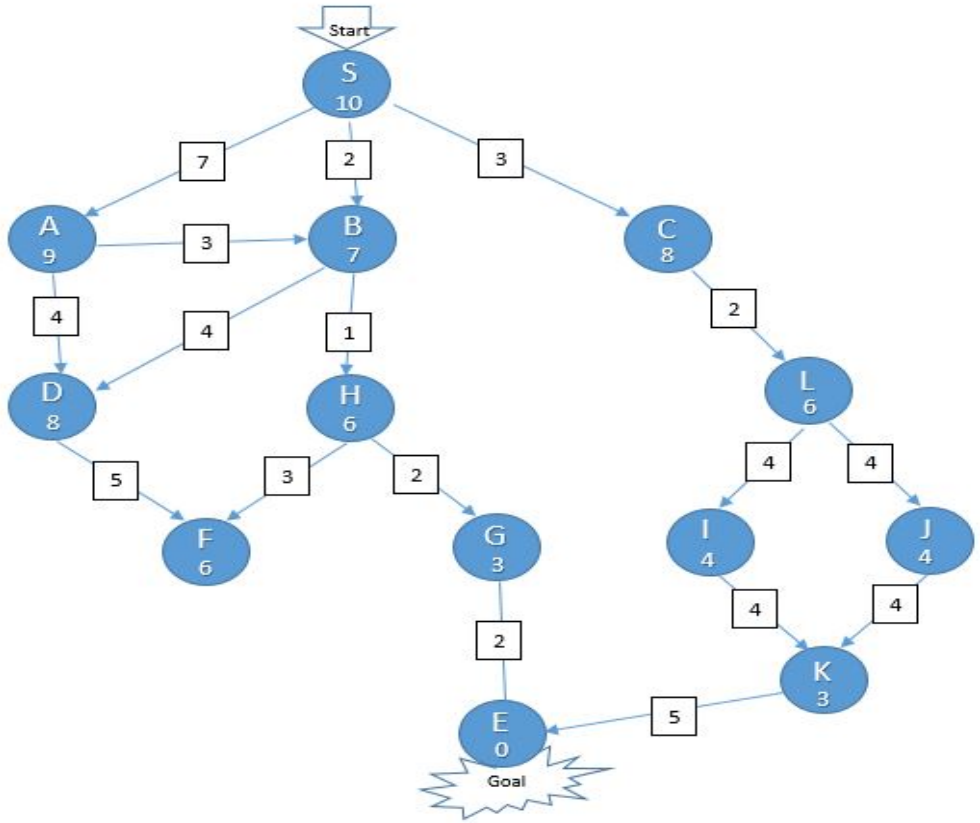
- Best first search is a traversal technique that decides which node is to be visited next by checking which node is the most promising one and then check it.
- Best first search is an instance of graph search algorithm in which a node is selected for expansion based on evaluation function f (n)
- Best 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

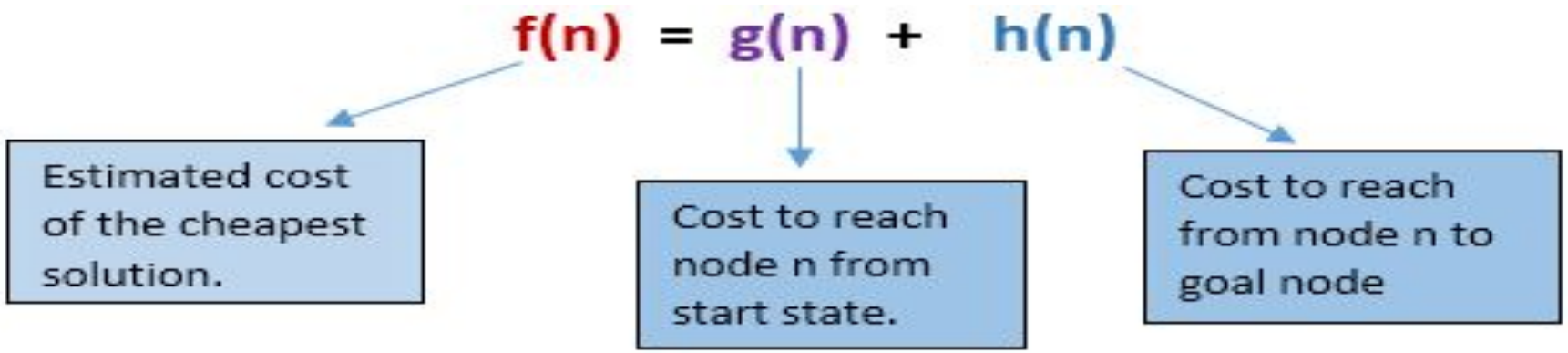
- It is an informed search algorithm, as it uses information about path cost and also uses heuristics to find the solution.
- To find the shortest path between nodes or graphs
- **A * algorithm** is a searching algorithm that searches for the shortest path between the *initial and the final state*
- 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.

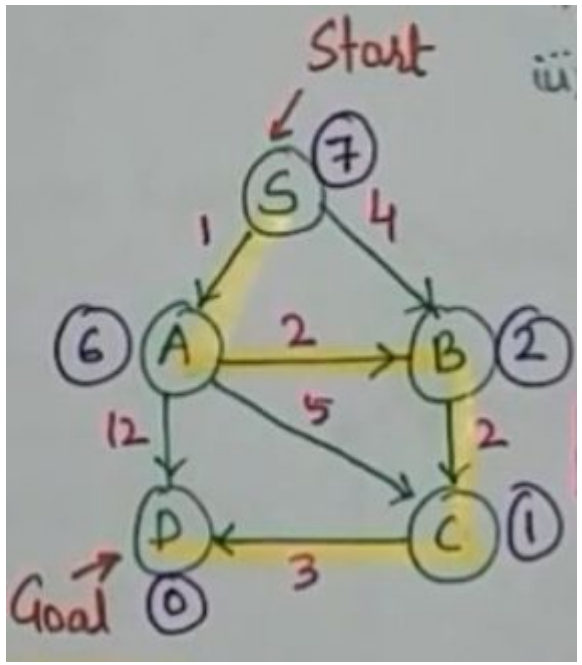
b. Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR

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 (1)



S → A = 1+6 = 7 ✓

S → B = 4+2 = 6 ✓

S → B → C = 4+2+1 = 7 ✓

S → B → C → D = 4+2+3+0 = 9 r1

S → A → B = 1+2+2 = 5 ✓

S → A → C = 1+5+1 = 7 ✓

S → A → D = 1+12 = 13 r2

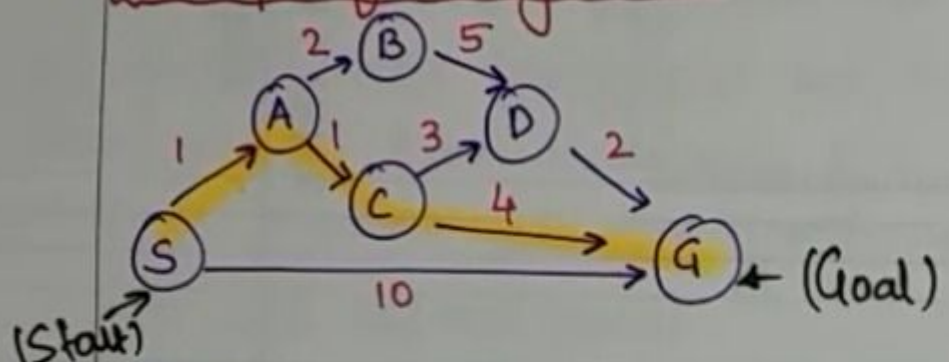
S → A → B → C = 1+2+2+1 = 6 ✓

S → A → B → C → D = 1+2+2+3 = 8 r3

S → A → C → D = 1+5+3 = 9 r4

A* search example (2)

Example of A* Algorithm:-



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

$S \to A = 1+3 = 4$

$S \to G = 10+0 = 10$

$S \to A \to B = 1+2+4 = 7$ ✓

$S \to A \to C = 1+1+2 = 4$ ✓

$S \to A \to C \to D = 1+1+3+6 = 11$ ✓

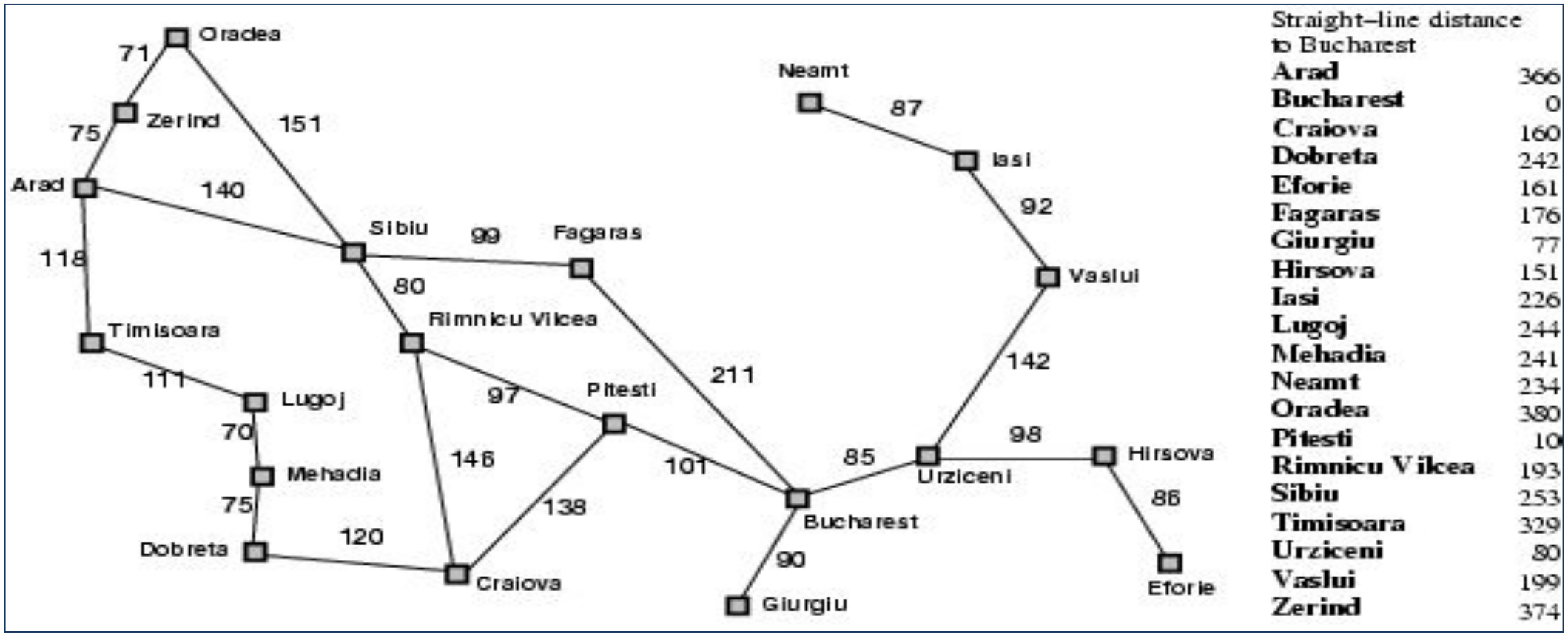
$S \to A \to C \to G = 1+1+4 = 6$.

$S \to A \to B \to D = 1+2+5+6 = 14$

$S \to A \to C \to D \to G = 1+1+3+2 = 7$

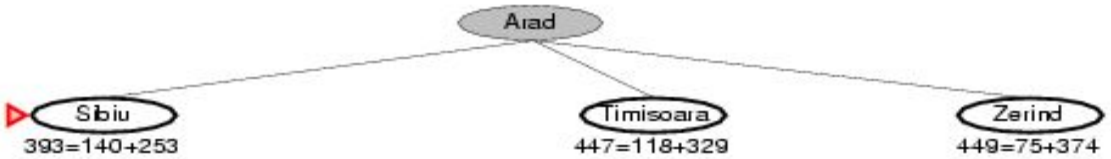
$S \to A \to B \to D \to G = 1+2+5+2 = 10$

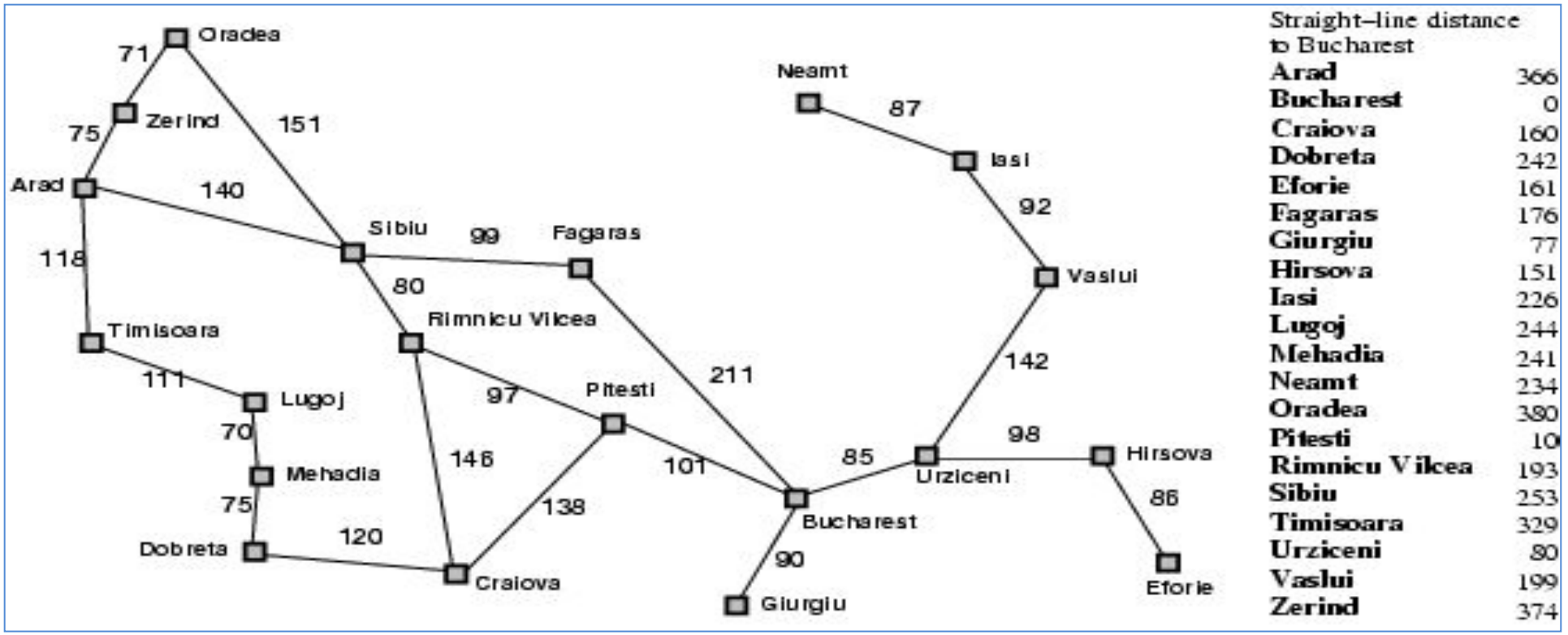
A* search example (3)



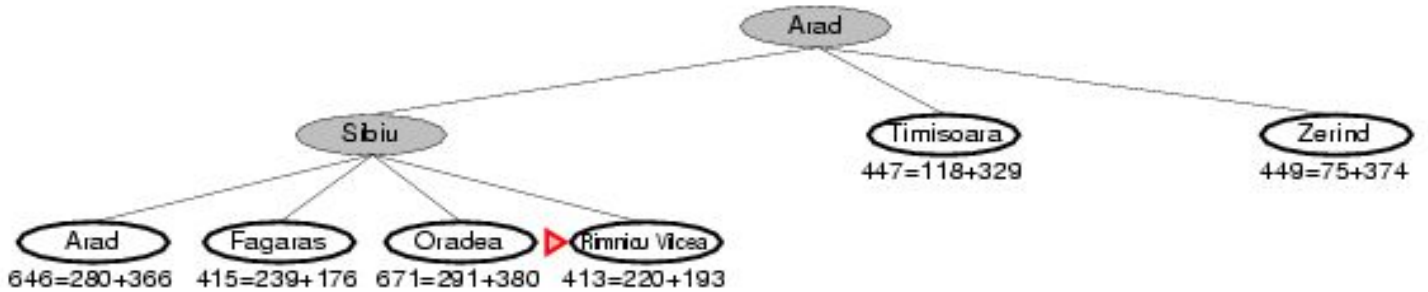
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

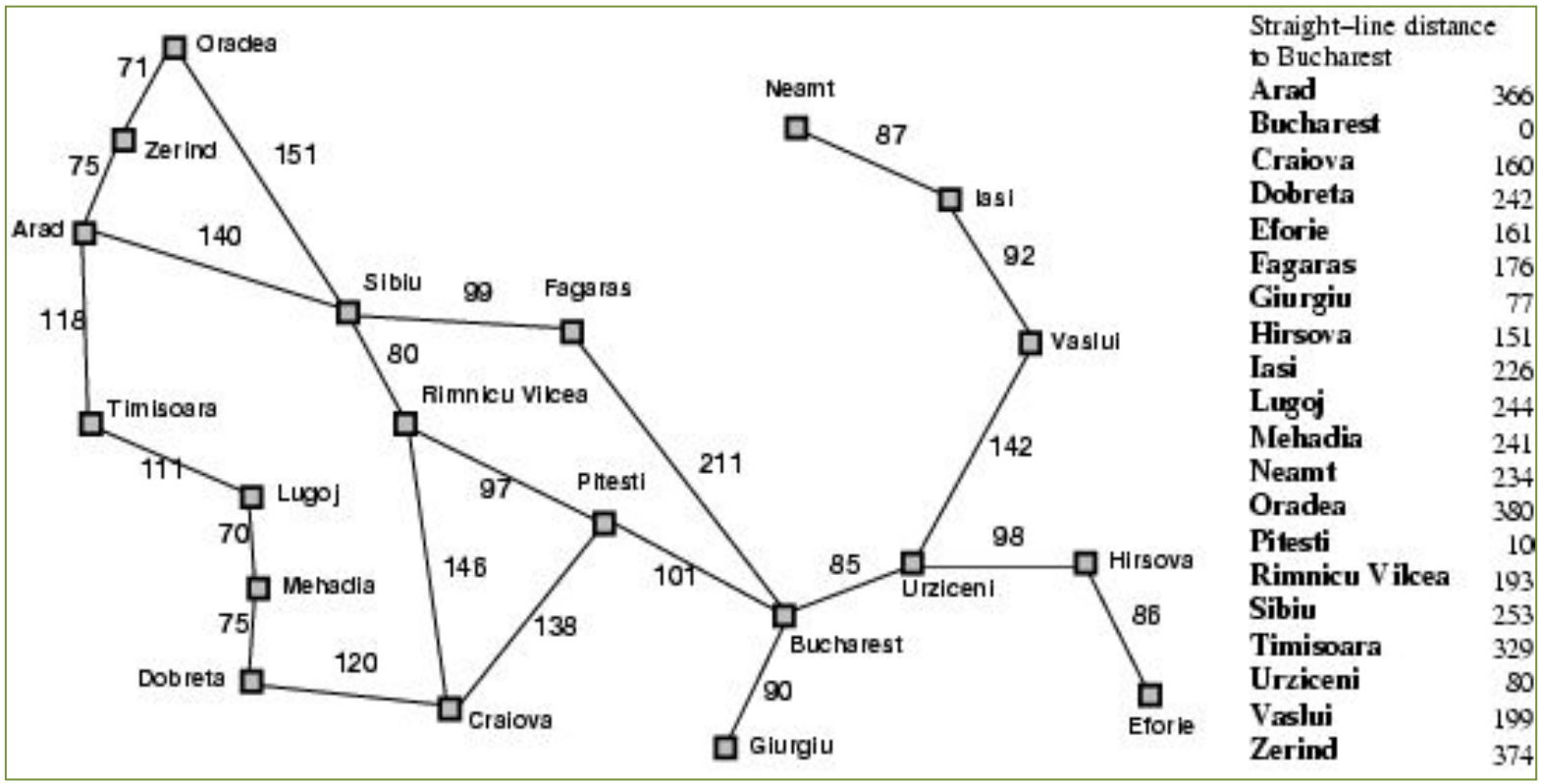
A* search example



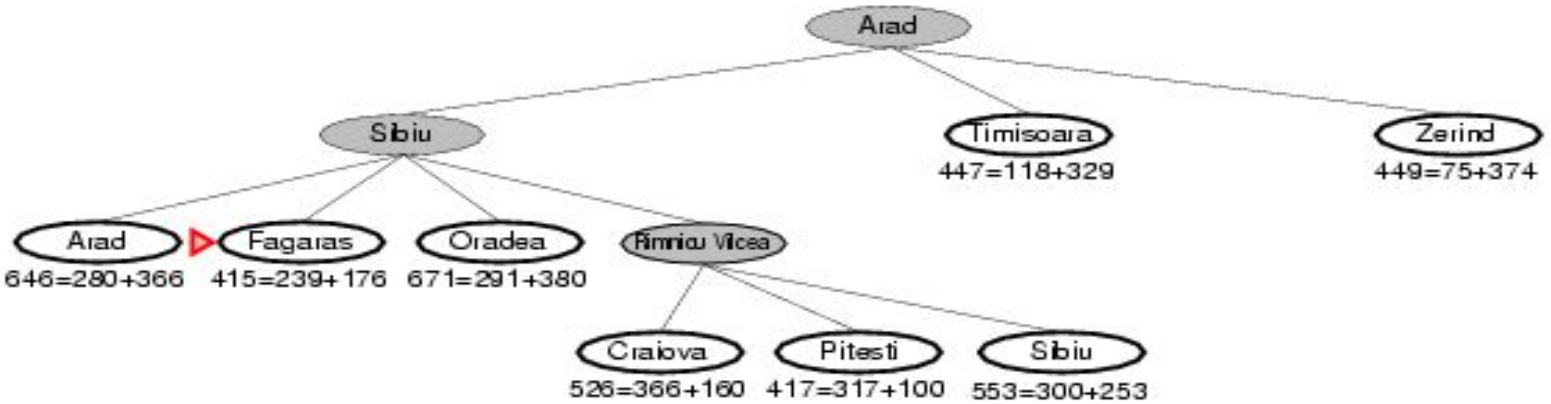


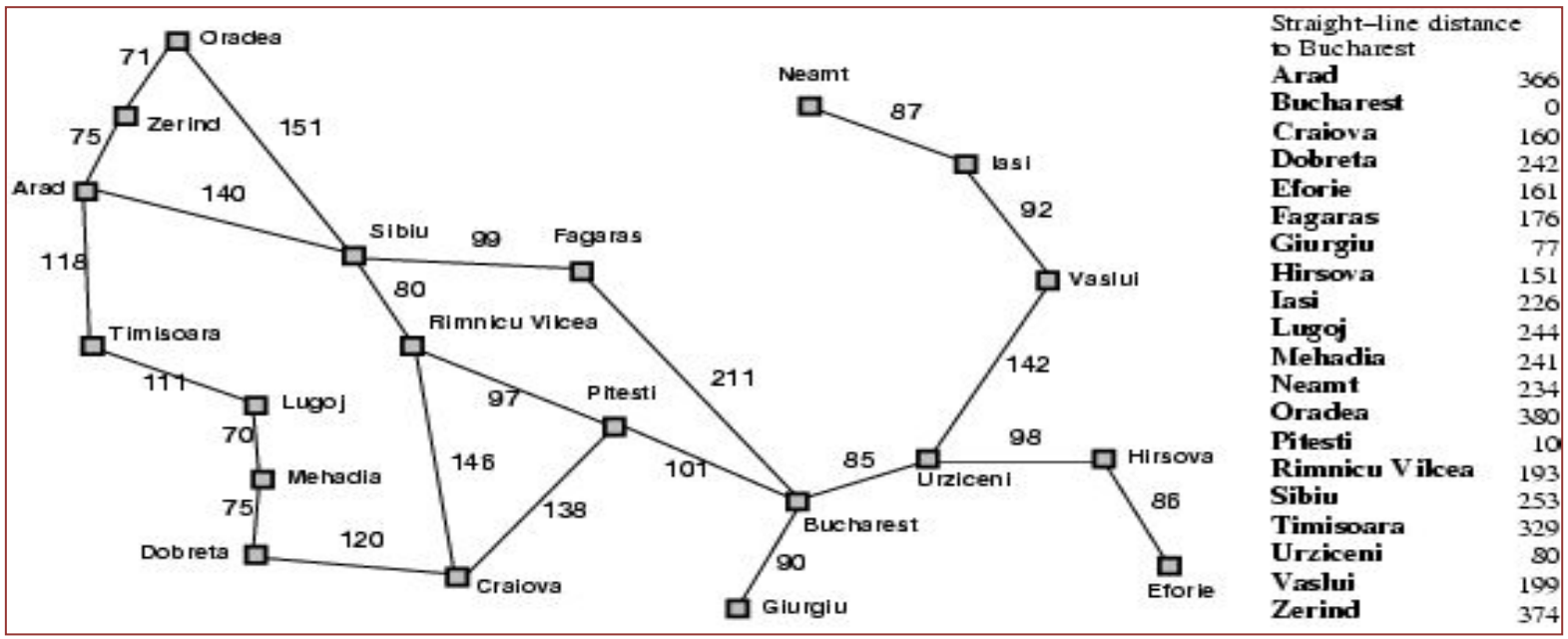
A* search example



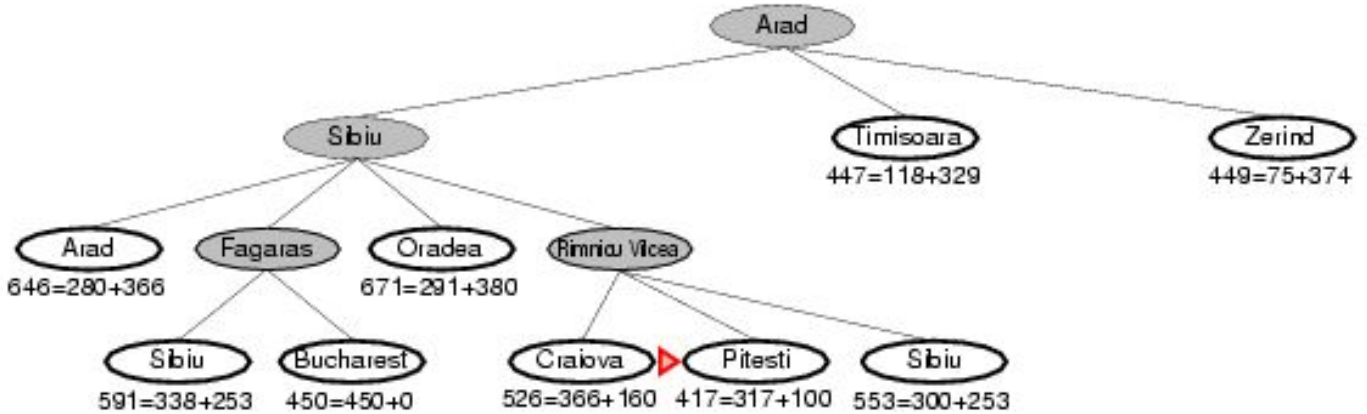


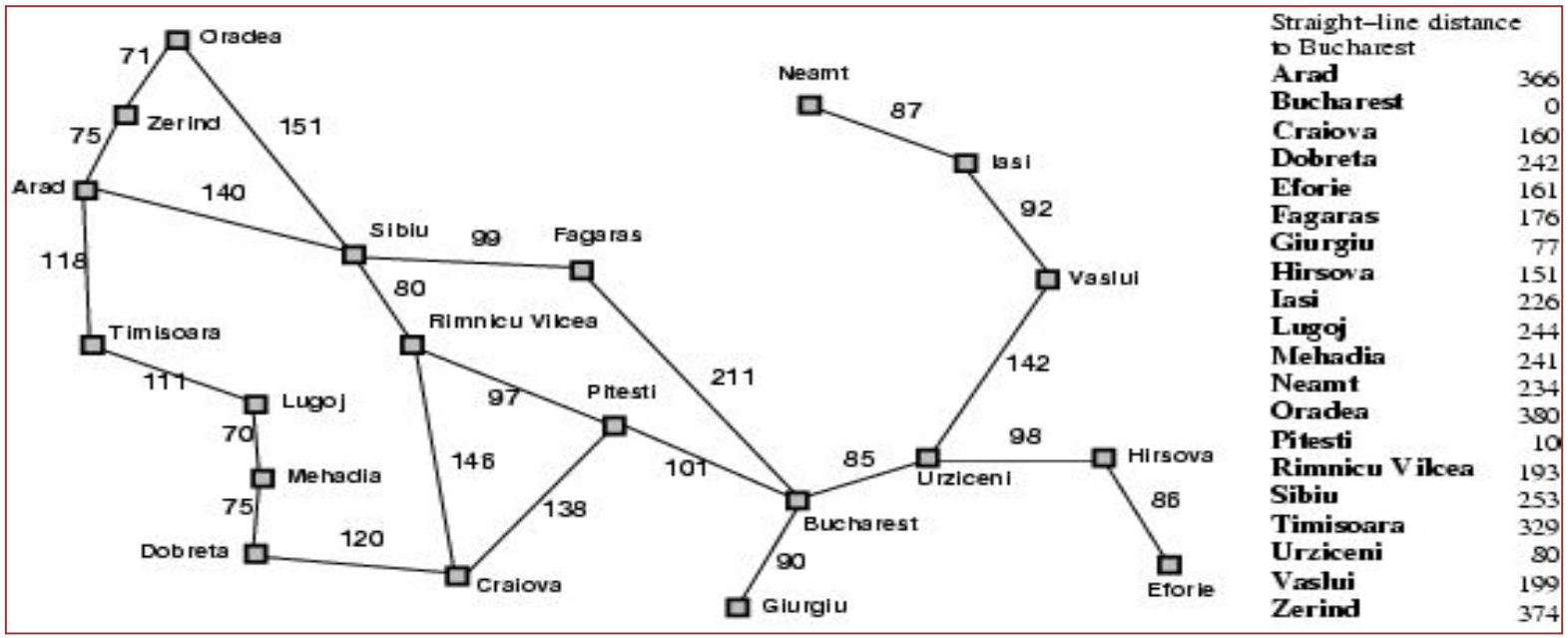
A* search example



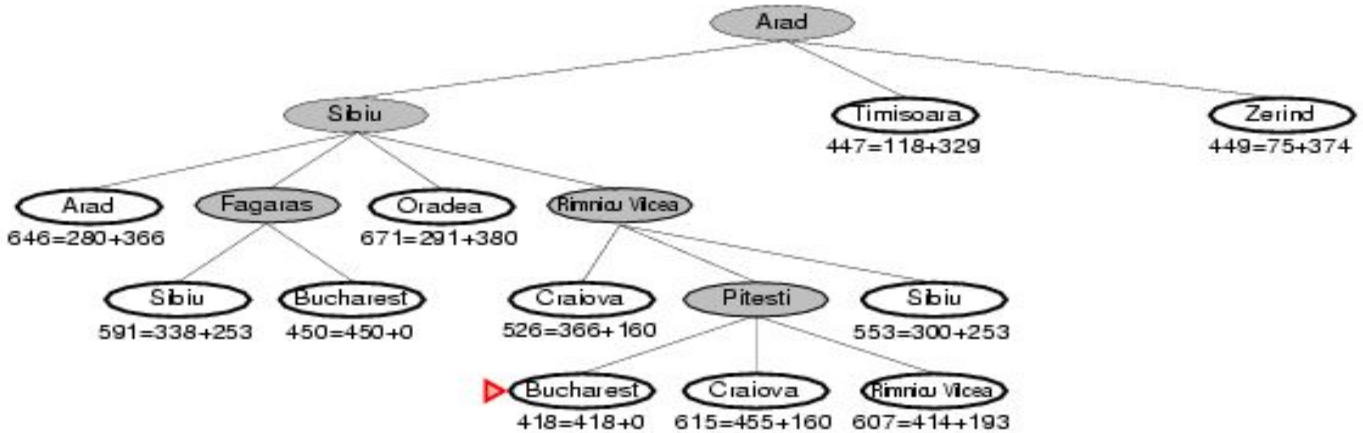


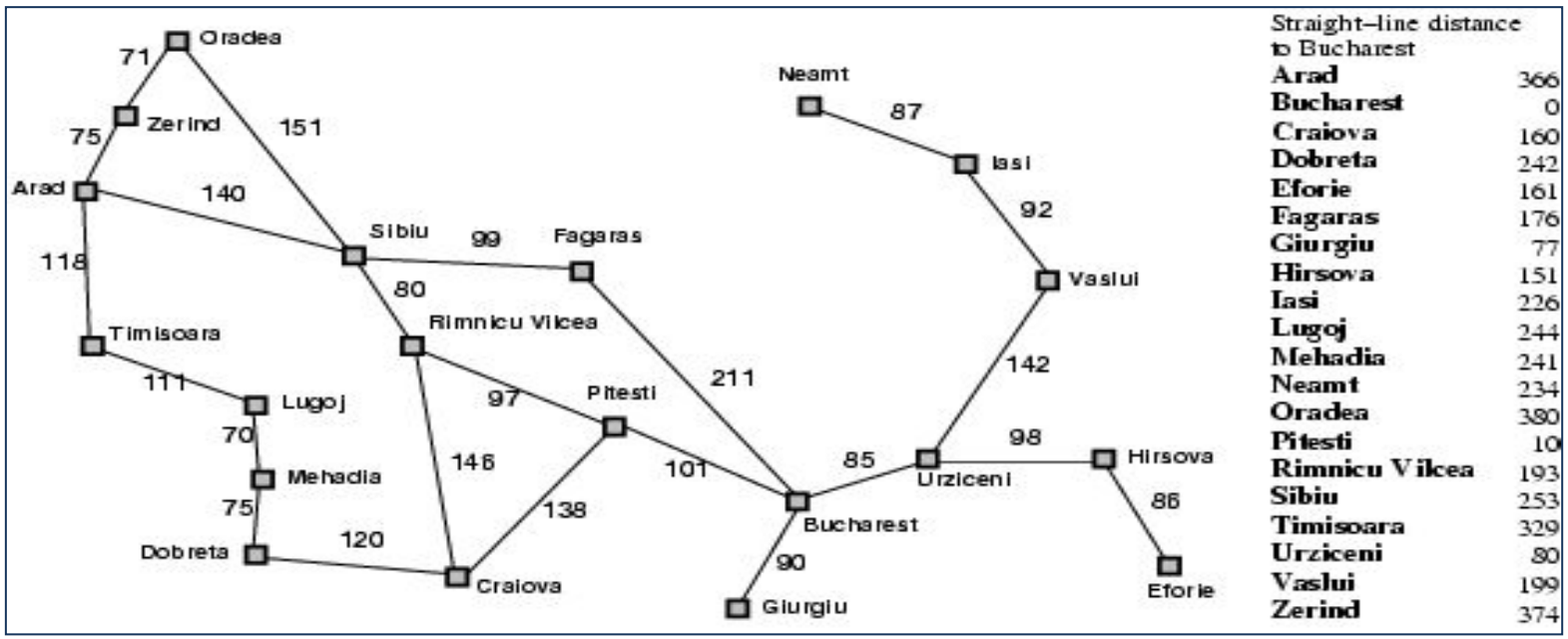
A* search example



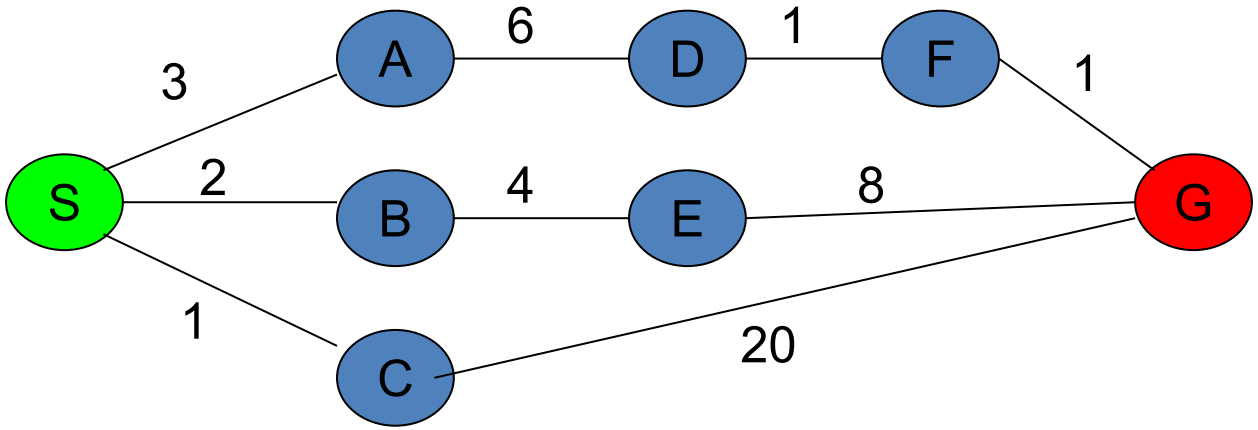


A* search example





try yourself !



straight-line distances

h(S-G)=10
h(A-G)=7
h(D-G)=1
h(F-G)=1
h(B-G)=10
h(E-G)=8
h(C-G)=20
h(G-G)=0

The graph above shows the step-costs for different paths going from the start (S) to the goal (G). On the right you find the straight-line distances.

1. Draw the search tree for this problem. *Avoid repeated states*.

2. Give the order in which the tree is searched (e.g. S-C-B...-G) for A* search.
   Use the straight-line dist. as a heuristic function, i.e. h=SLD,
   and indicate for each node visited what the value for the evaluation function, f, is.

AO* SEARCH ALGORITHM

AO* search Algorithm

AO* Algorithm

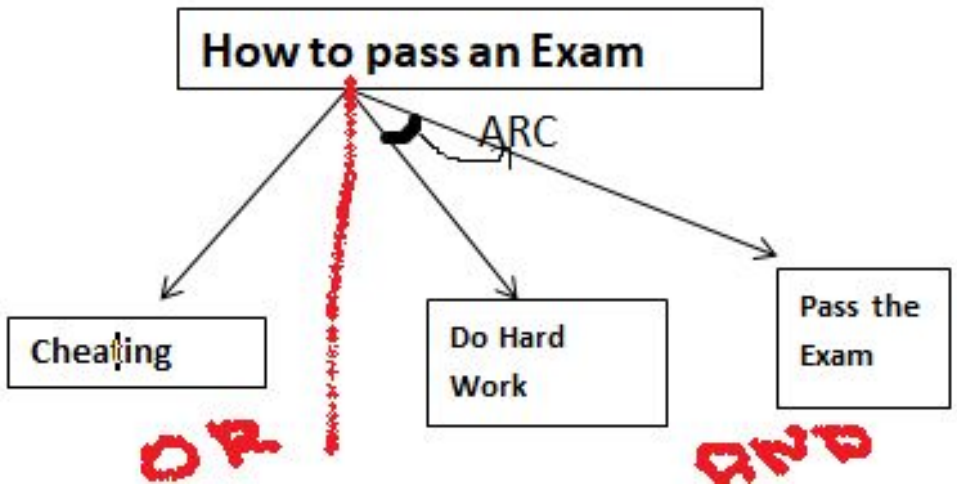
- It represents an AND-OR Graph algorithm that is used to find more than one solution.
- AO* Algorithm basically based on problem decompositon (Breakdown problem into small pieces).
- When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, **AND-OR graphs** or **AND - OR trees** are used for representing the solution.
- Its an efficient method to explore a solution path.
- The decomposition of the problem or problem reduction generates AND arcs.

AND – OR Graph

- AND-OR graph is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved.

AO* search Algorithm

- AND – OR Graph



The figure shows an AND-OR graph

1) To pass any exam, we have two options, either cheating or hard work.

2) In this graph we are given two choices, first do cheating **or (The red line)** work hard and **(The arc)** pass.

3) When we have more than one choice and we have to pick one, we apply **OR condition** to choose one.(That's what we did here).

4) Basically the **ARC** here denote **AND condition**.

Here we have replicated the arc between the work hard and the pass because by doing the hard work possibility of passing an exam is more than cheating.

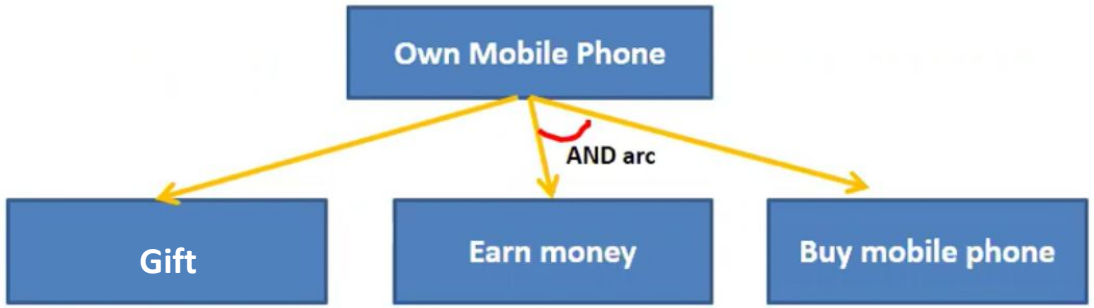
AO* search Algorithm

- AND – OR Graph

The figure shows an AND-OR graph

AO* search Algorithm

- AND – OR Graph



AO* search Algorithm

- Node in the graph will point both down to its successors and up to its parent nodes.
- Each Node in the graph will also have a heuristic value associated with it.

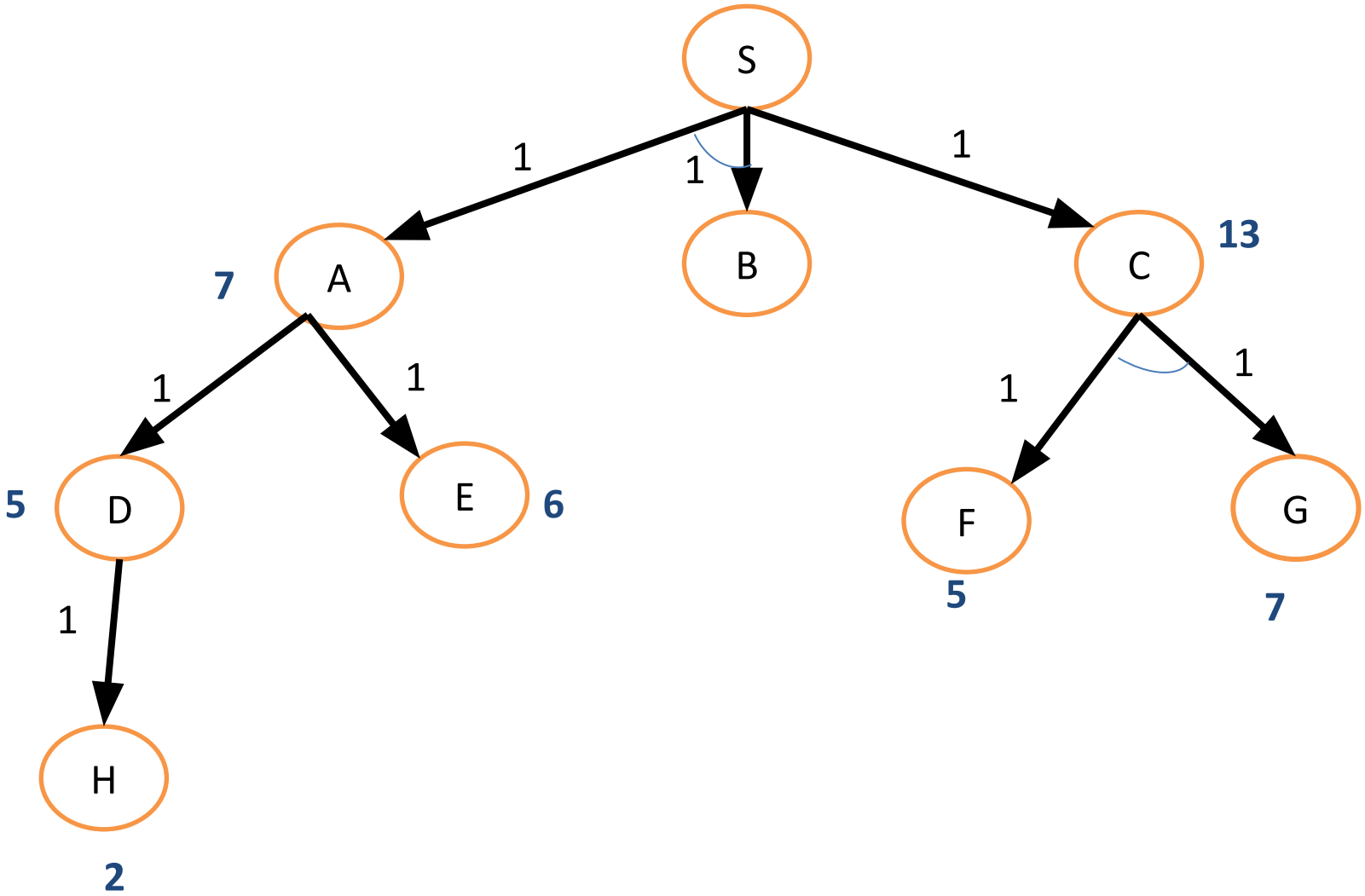
$$f(n)=g(n)+h(n)$$

- f(n) – Cost function.
- g(n) — actual cost or edge value
- h(n) — heuristic/ estimated value of the nodes

*AO** Algorithm

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

ESTIMATED HEURISTIC VALUE



AO*: Example #1

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

Path-1: f(S-C)= 1+13=14

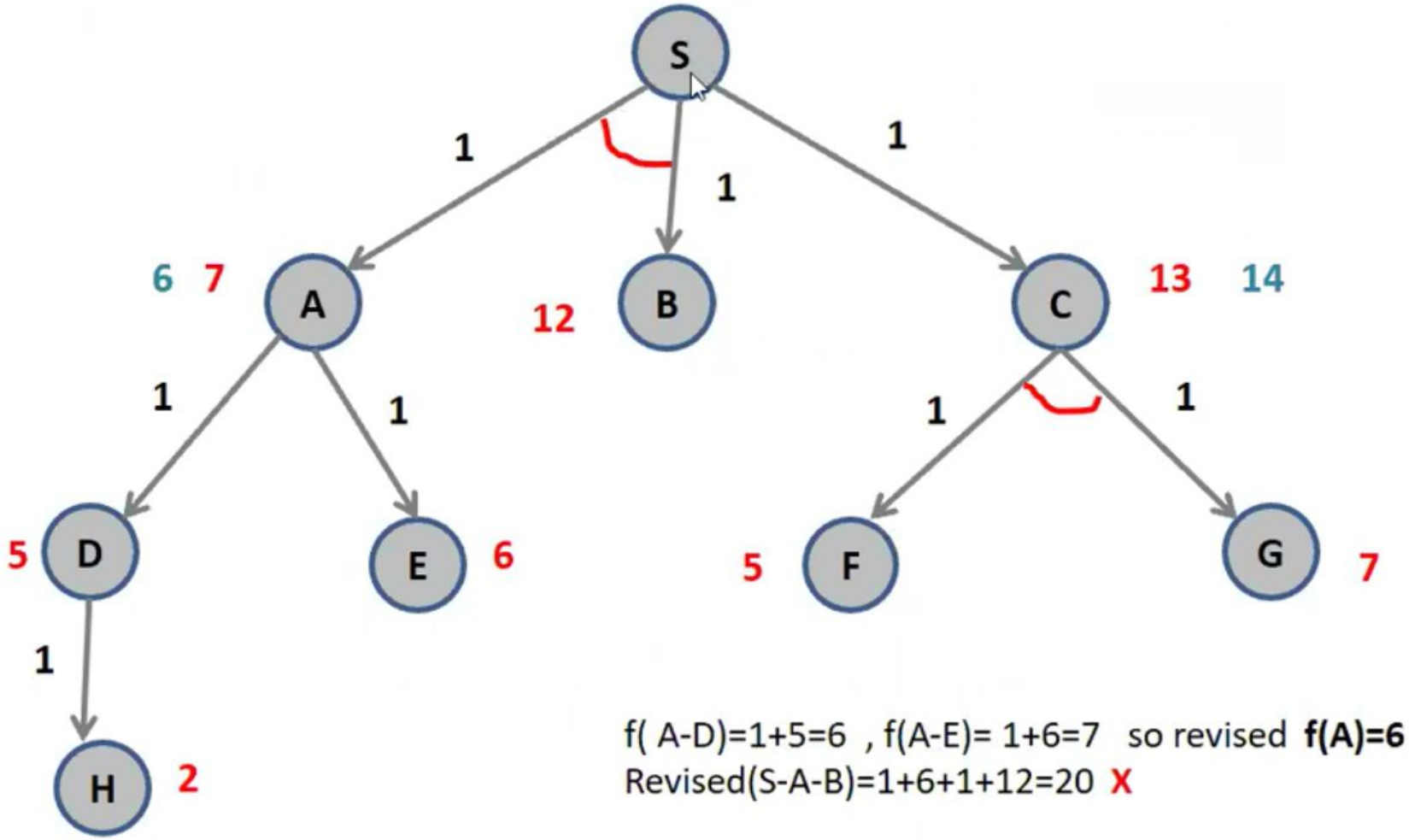
Path-2: f(S-A-B)=1+1+7+12=21

f(C-F-G)=1+1+5+7=14

f(S-C)=1+14=15 (revised)

Revised cost: 15

Min(14,21)=14



f(A-D)=1+5=6 , f(A-E)= 1+6=7 so revised **f(A)=6**

Revised(S-A-B)=1+6+1+12=20 X

A*	AO*
• It represents an OR graph algorithm that is used to find a single solution (either this or that). • It is a computer algorithm which is used in path-finding and graph traversal. It is used in the process of plotting an efficiently directed path between a number of points called nodes. • In this algorithm you traverse the tree in depth and keep moving and adding up the total cost of reaching the cost from the current state to the goal state and add it to the cost of reaching the current state.	• It represents an AND-OR graph algorithm that is used to find more than one solution by ANDing more than one branch. • In this algorithm you follow a similar procedure but there are constraints traversing specific paths. • When you traverse those paths, cost of all the paths which originate from the preceding node are added till that level, where you find the goal state regardless of the fact whether they take you to the goal state or not.

Local search algorithms

Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution

- State space = set of "complete" configurations
- Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- keep a single "current" state, try to improve it.
- Very memory efficient (only remember current state)

Types of Local Search

- **Hill-climbing Search**
- **Simulation Annealing Search**

Hill Climbing

- Searching for a goal state = Climbing to the top of a hill.
- Generate-and-test + direction to move.
- Heuristic function to estimate how close a given state is to a goal state.

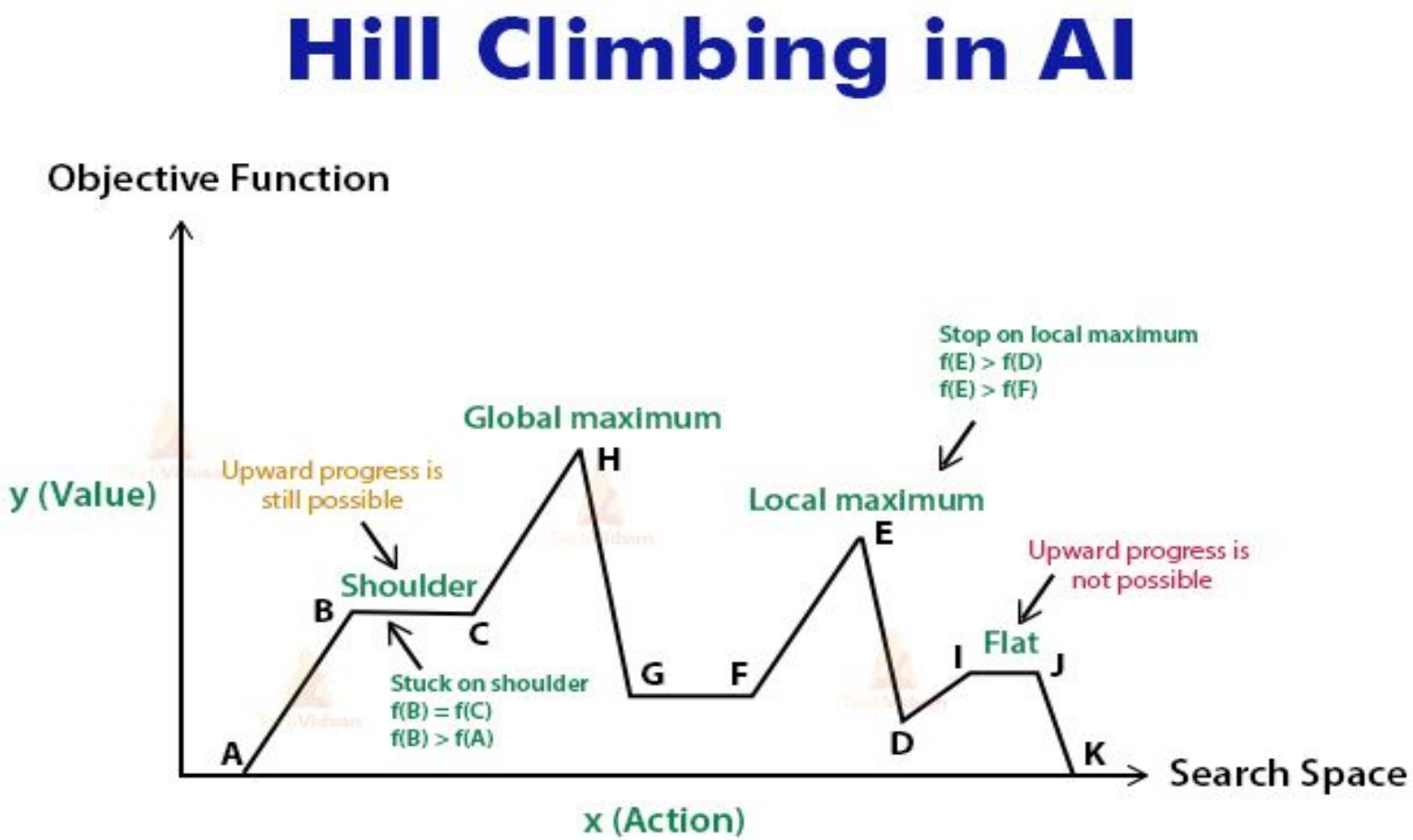
Hill Climbing

Algorithm

1. Evaluate the initial state.

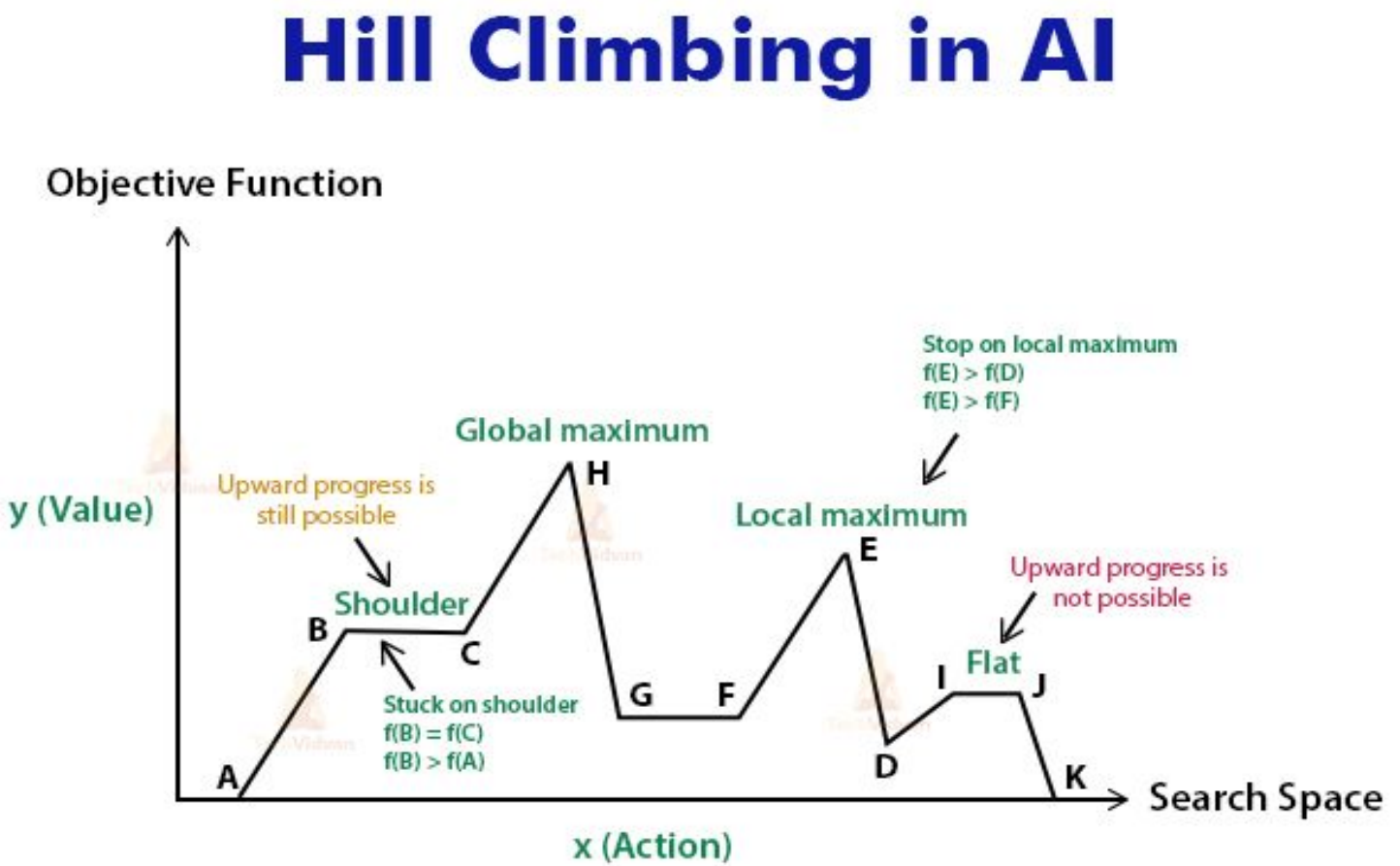
2. Loop until a solution is found or there are no new operators left to be applied:
   - Select and apply a new operator
   - Evaluate the new state:
     goal → quit
     better than current state → new current state

State Space diagram for Hill Climbing



Hill-climbing search

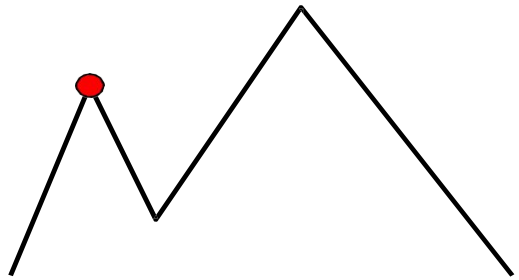
- Problem: depending on initial state, can get stuck in local maxima



Hill Climbing: Disadvantages

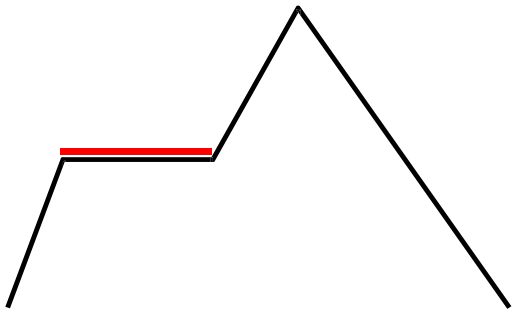
Local maximum

A state that is better than all of its neighbours, but not better than some other states far away.



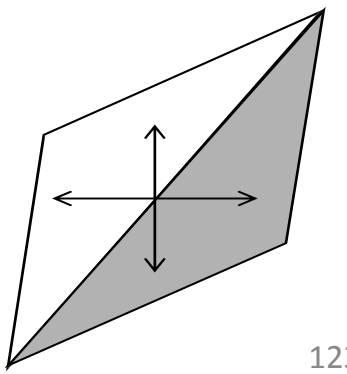
Plateau

A flat area of the search space in which all neighbouring states have the same value.



Ridge

The orientation of the high region, compared to the set of available moves, makes it impossible to climb up. However, two moves executed serially may increase the height.



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

Simulated annealing search

Physical Annealing

- Physical substances are melted and then gradually cooled until some solid state is reached.
- The goal is to produce a minimal-energy state.
- Annealing schedule: if the temperature is lowered sufficiently slowly, then the goal will be attained.
- Nevertheless, there is some probability for a transition to a higher energy state: $e^{-\Delta E/kT}$.

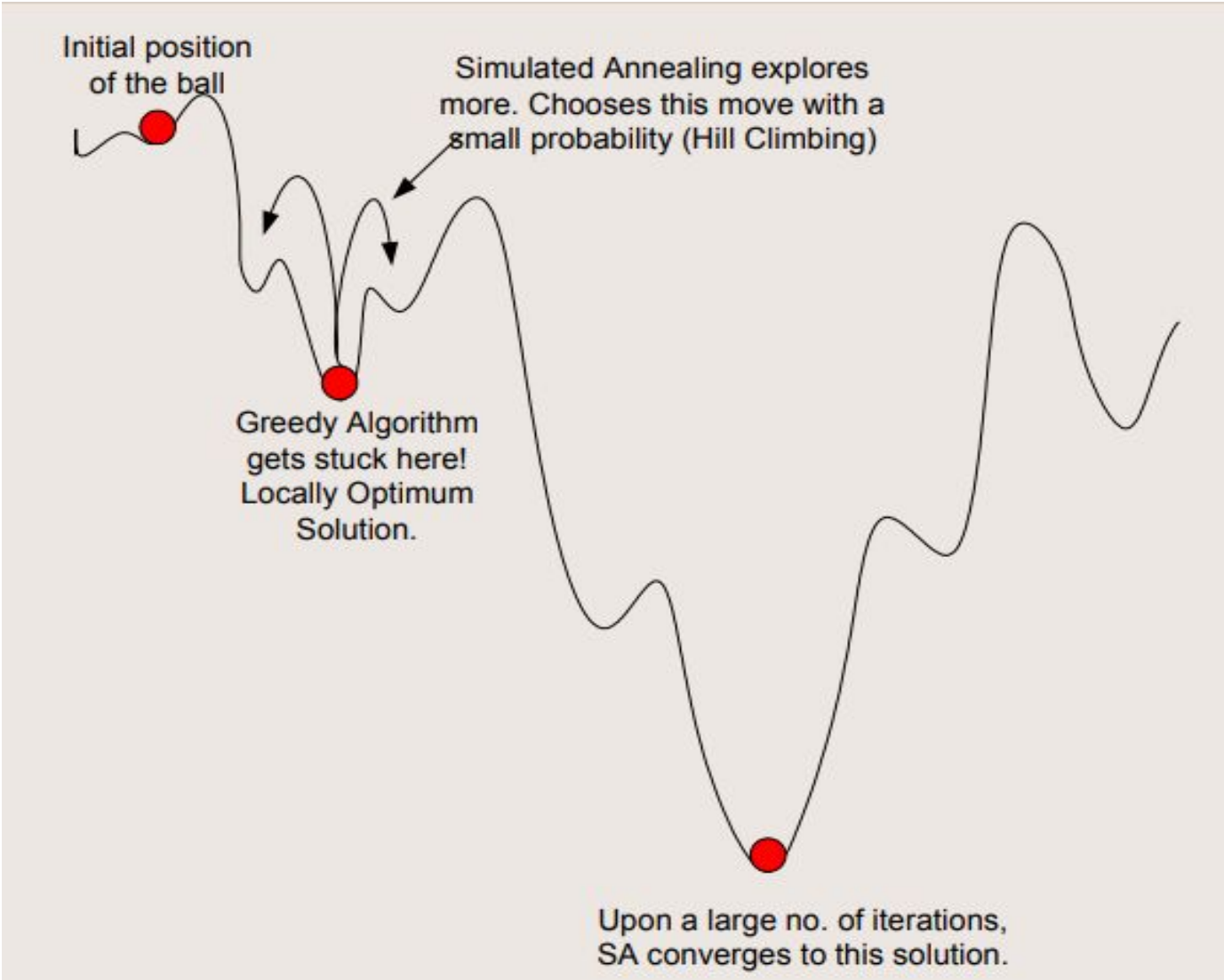
Simulated annealing search

- A variation of hill climbing in which, at the beginning of the process, some downhill moves may be made.
- To do enough exploration of the whole space early on, so that the final solution is relatively insensitive to the starting state.
- Lowering the chances of getting caught at a local maximum, or plateau, or a ridge.

Simulated Annealing

Algorithm

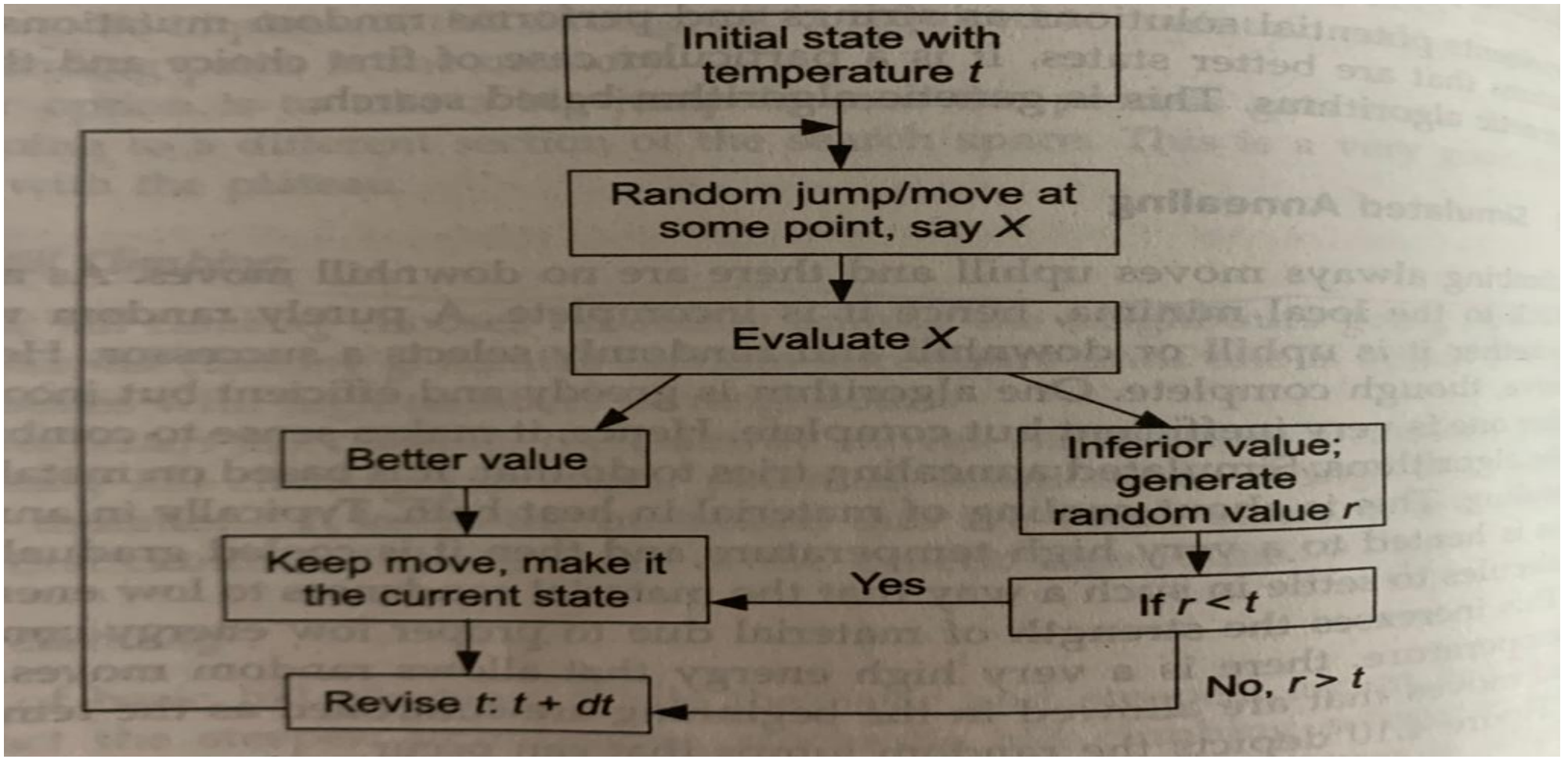
- First generate a random solution
- Calculate it's cost using some cost function
- Generate a random neighbor solution and calculate it's cost
- Compare the cost of old and new random solution
- If C old > C new then go for old solution otherwise go for new solution
- Repeat steps 3 to 5 until you reach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


Initial position of the ball
Simulated Annealing explores more. Chooses this move with a small probability (Hill Climbing)
Greedy Algorithm gets stuck here! Locally Optimum Solution.
Upon a large no. of iterations, SA converges to this 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.

Simulated Annealing – Basic Steps



Local Beam Search

- **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.

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.**
- In 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.

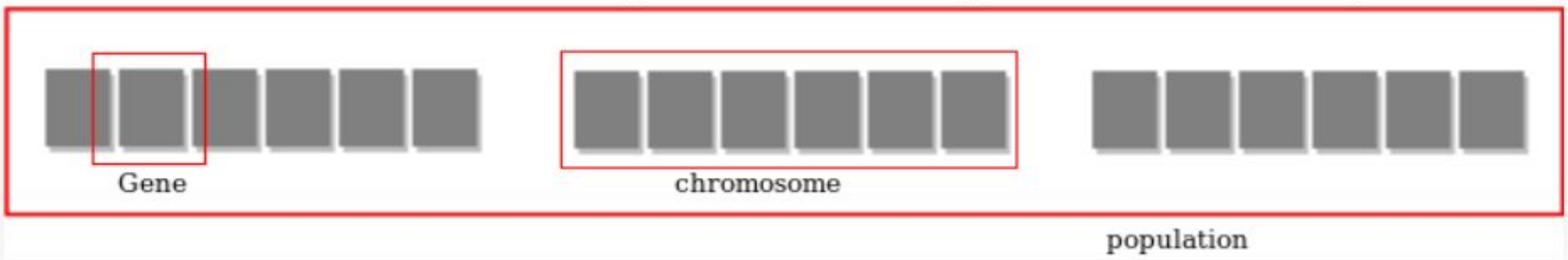
Optimization

Optimization is the process of **making something better**

Finding the values of inputs in such a way that we get the “best” output values

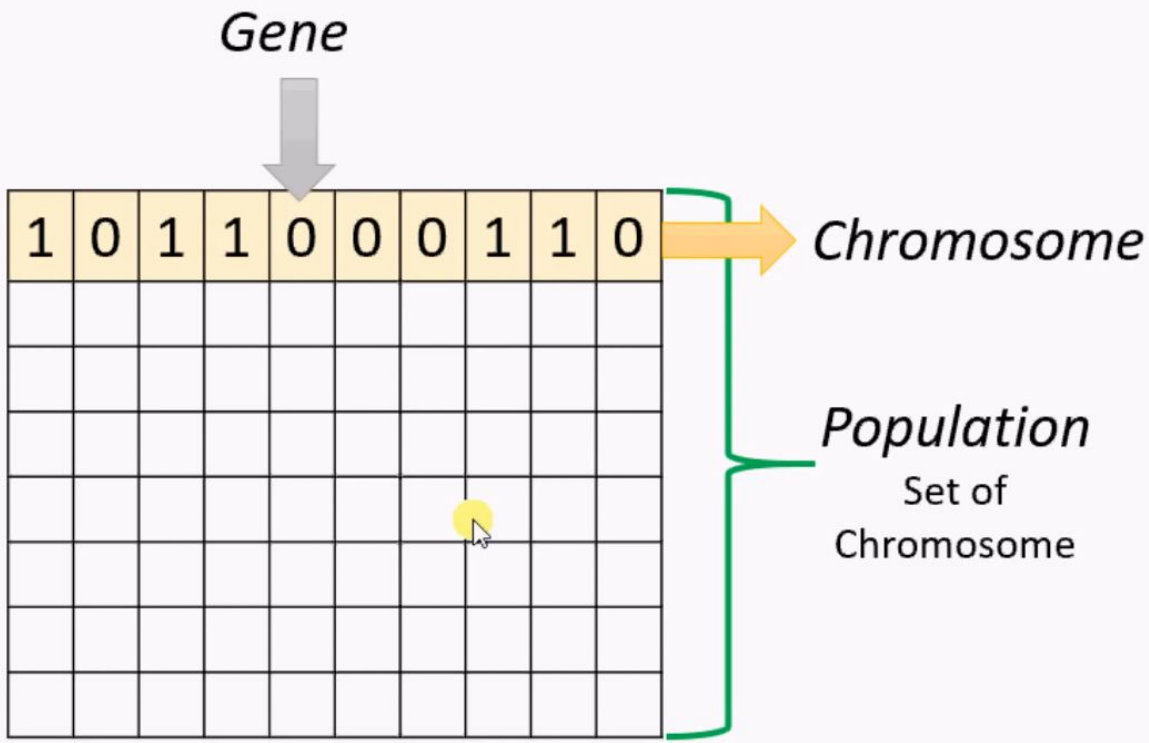
Genetic Algorithms

- 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

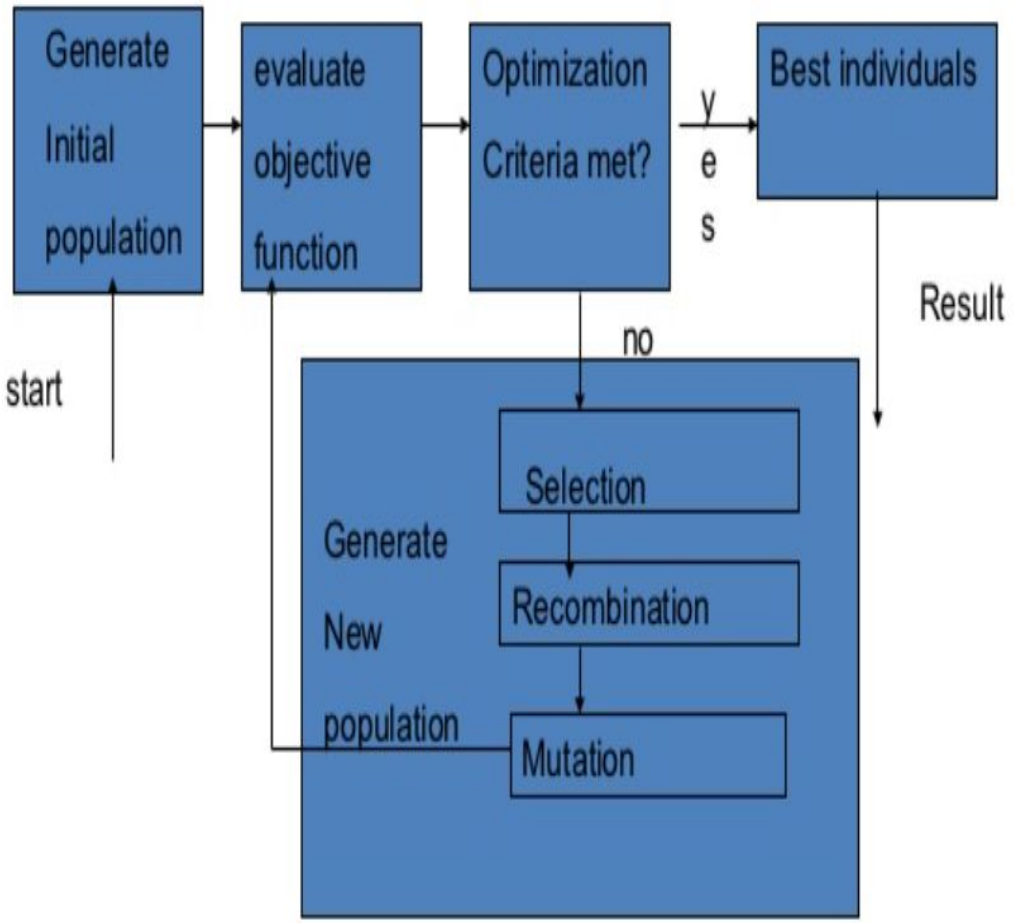


Terminology

- Population
- Chromosomes
- Gene



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

Fitness Function

The *fitness function* is the function you want to optimize. Function which takes the solution as input and produces the suitability of the solution as the output.

Genetic Algorithms

Significance of the genetic operators

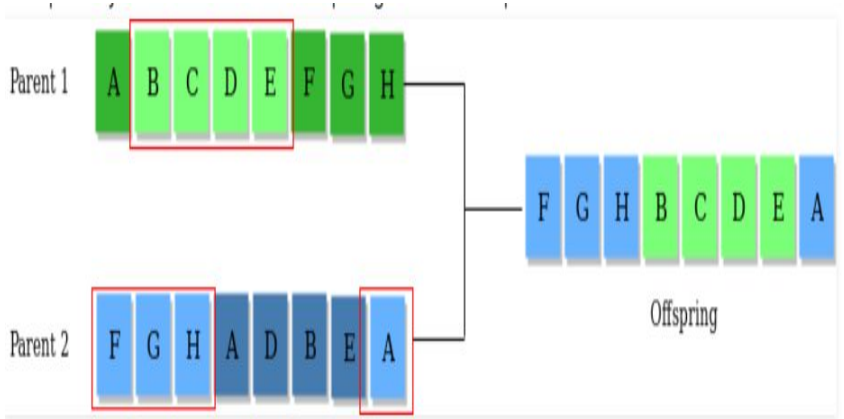
- Reproduction or selection by two parent chromosomes is done based on their fitness
- Reproduction ensures that only the fittest of the solutions made to form offsprings
- Reproduction will force the GA to search that area which has highest fitness values

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

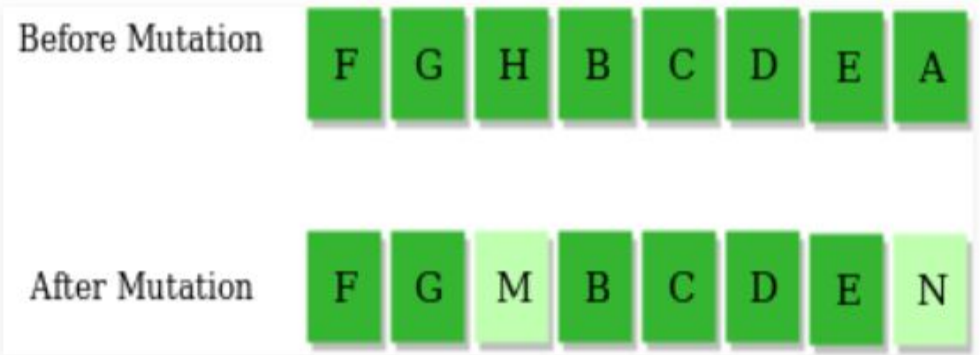
Mutation: To avoid local optimum, mutation is used

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

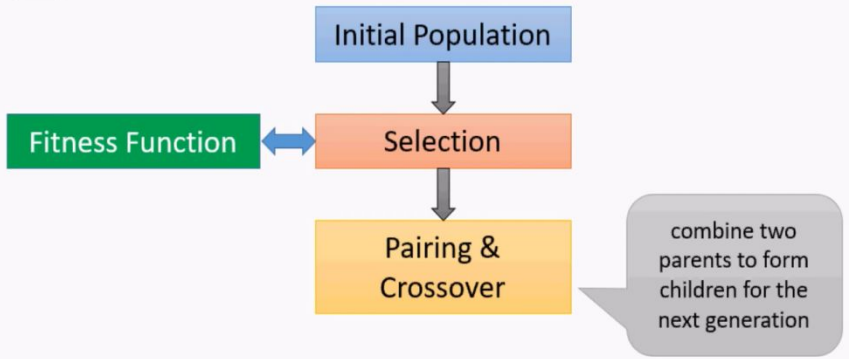


- Mutation



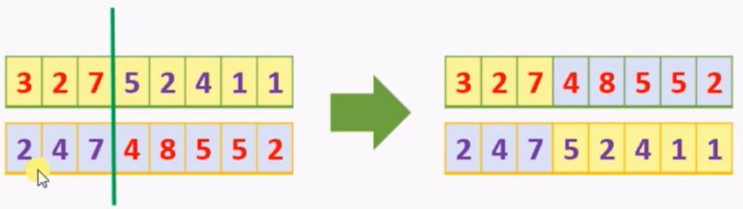
Cross over

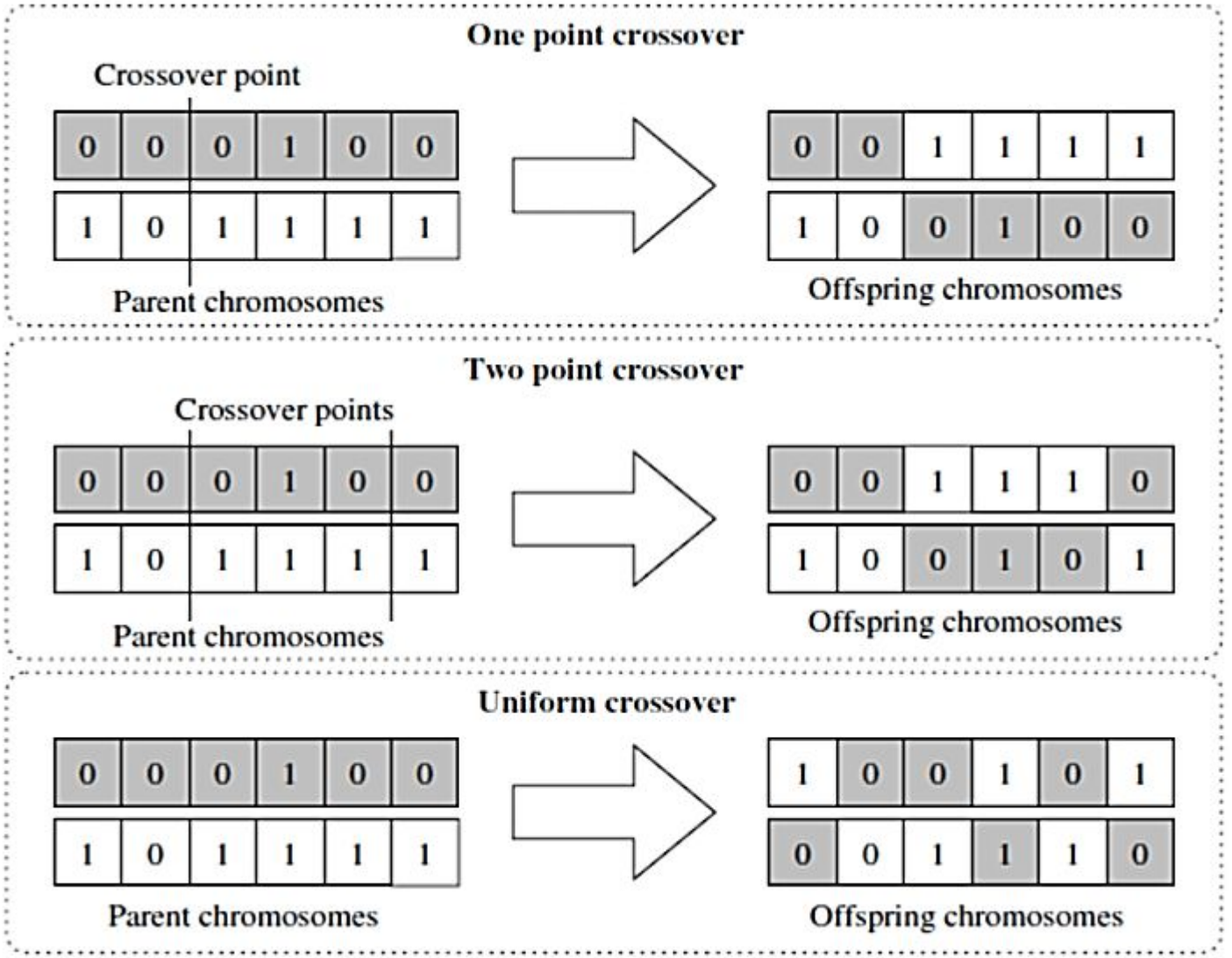
Concept



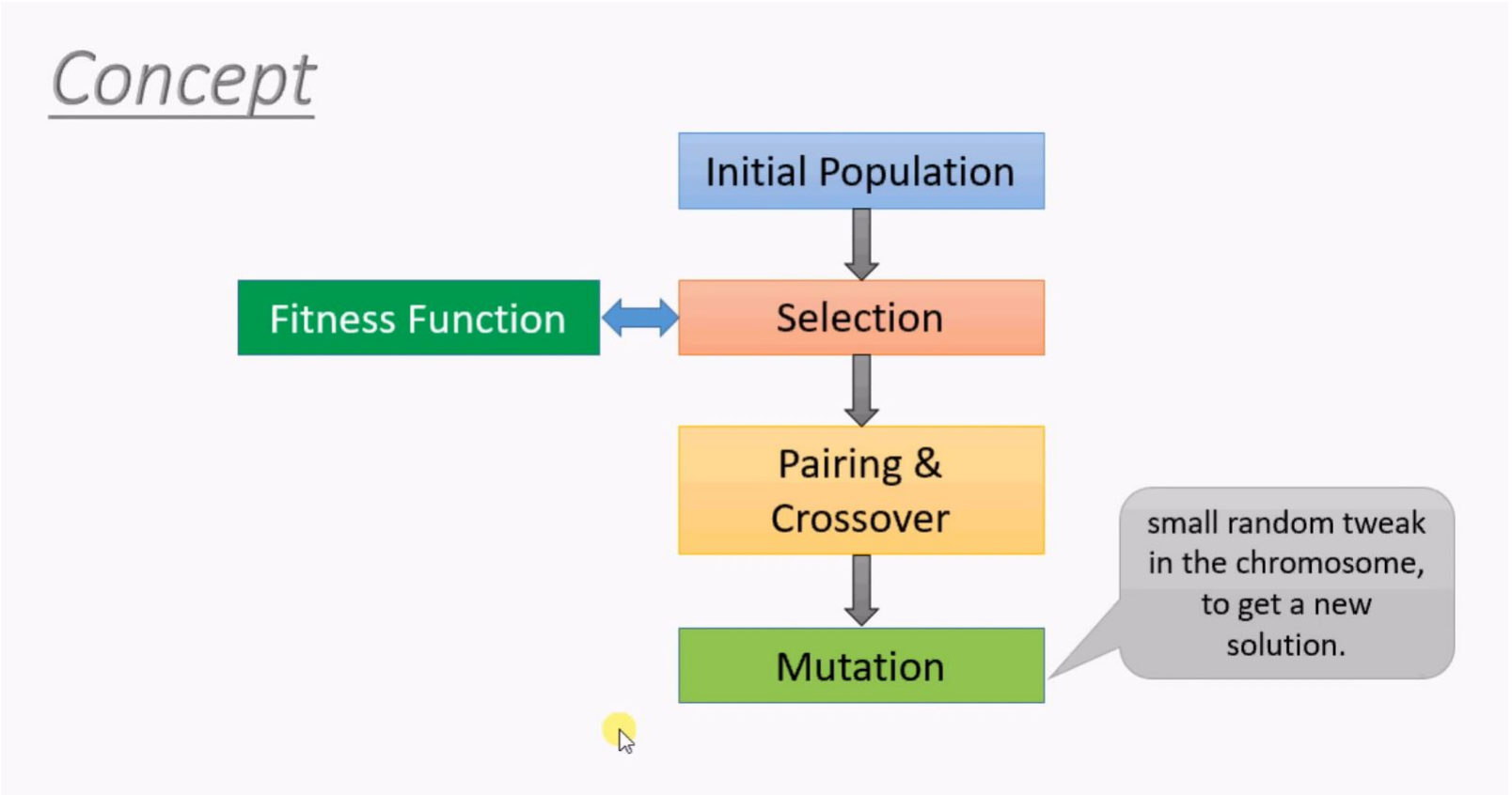
Pairing and Crossover

- *One Point Crossover* : A random crossover point is selected and the tails of its two parents are swapped to get new off-springs.





Mutation


Concept
Initial Population
Fitness Function
Selection
Pairing & Crossover
Mutation
small random tweak in the chromosome, to get a new solution.

Mutation

- *Bit Flip Mutation* : Select one or more random bits and flip them.

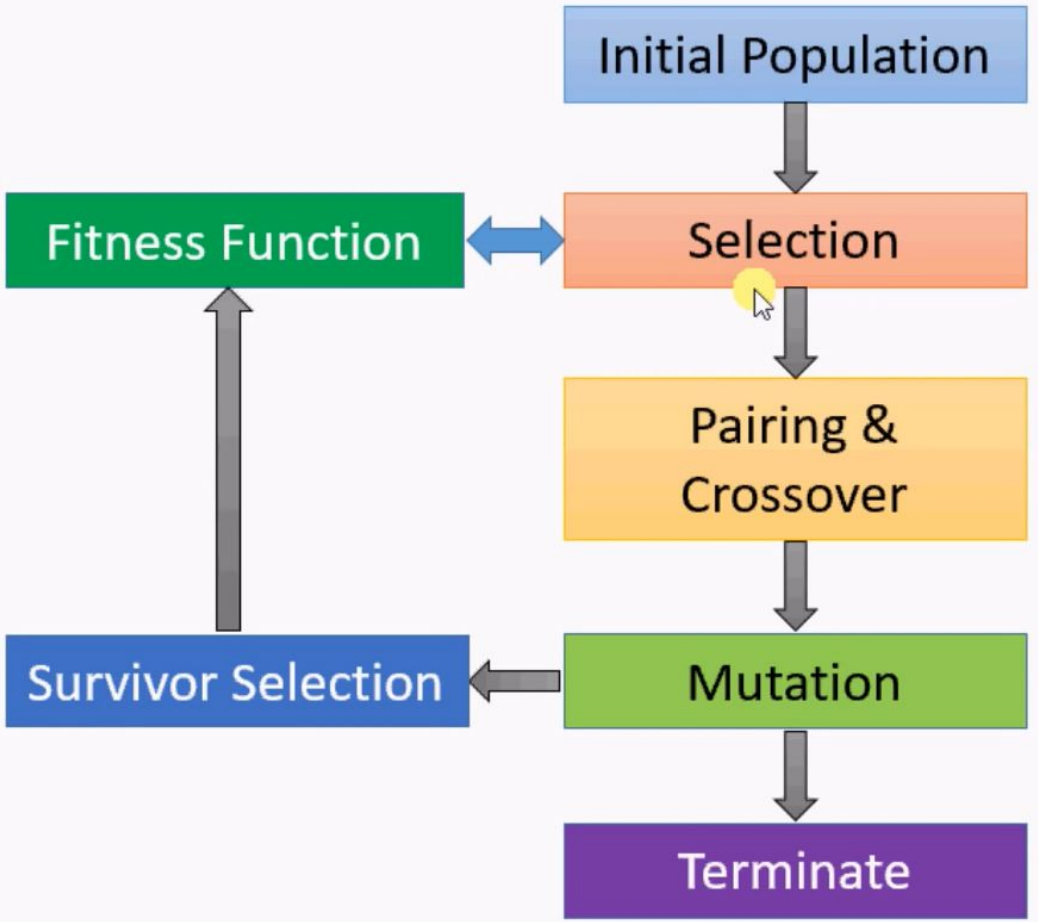
- *Swap Mutation*

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/

Concept


Initial Population
Fitness Function
Selection
Pairing & Crossover
Survivor Selection
Mutation
Terminate

Advantages of Genetic Algorithm

- Does not require any derivative information.
- faster and more efficient as compared to the traditional methods.
- Optimizes both continuous and discrete functions and also multi-objective problems.
- Provides a list of "good" solutions and not just a single solution.
- Useful when the search space is very large and there are a large number of parameters involved.

Limitations of Genetic Algorithm

- Computationally expensive as Fitness value is calculated repeatedly.
- Not suited for all problems, especially problems which are simple and for which derivative information is available.
- GA may not converge to the optimal solution, if not implemented properly.

Genetic Algorithms

Suppose a genetic algorithm uses chromosomes of the form x = abcdefgh with a fixed length of eight genes. Each gene can be any digit between 0 and 9. Let the fitness of individual x be calculated as: f(x) = (a + b) − (c + d) + (e + f) − (g + h) , and let the initial population consist of four individuals with the following chromosomes:

x1 = 6 5 4 1 3 5 3 2

x2 = 8 7 1 2 6 6 0 1

x3 = 2 3 9 2 1 2 8 5

x4 = 4 1 8 5 2 0 9 4

X1, f(x1) = (6+5) – (4+1) + (3+5) – (3+2) = 11-5+8-5 = 9

X2 , f(x2) = 23

X3 , f(x3) = -16

x4, f(x4) = -19 x2, x1, x3, x4

a) Evaluate the fitness of each individual, showing all your workings, and arrange them in order with the fittest first and the least fit last.

Genetic Algorithms

b. i) Cross the fittest two individuals using one–point crossover at the middle point.

x1 = 6 5 4 1 | 3 5 3 2
x2 = 8 7 1 2 | 6 6 0 1

X1 = 6 5 4 1 6 6 0 1
X2= 8 7 1 2 3 5 3 2

ii) Cross the second and third fittest individuals using a two–point crossover (points b and f).

x1 = 6 5 | 4 1 3 5 | 3 2
x3 = 2 3 | 9 2 1 2 | 8 5

X1 = 6 5 9 2 1 2 3 2
X3 = 2 3 4 1 3 5 8 5

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

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

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.

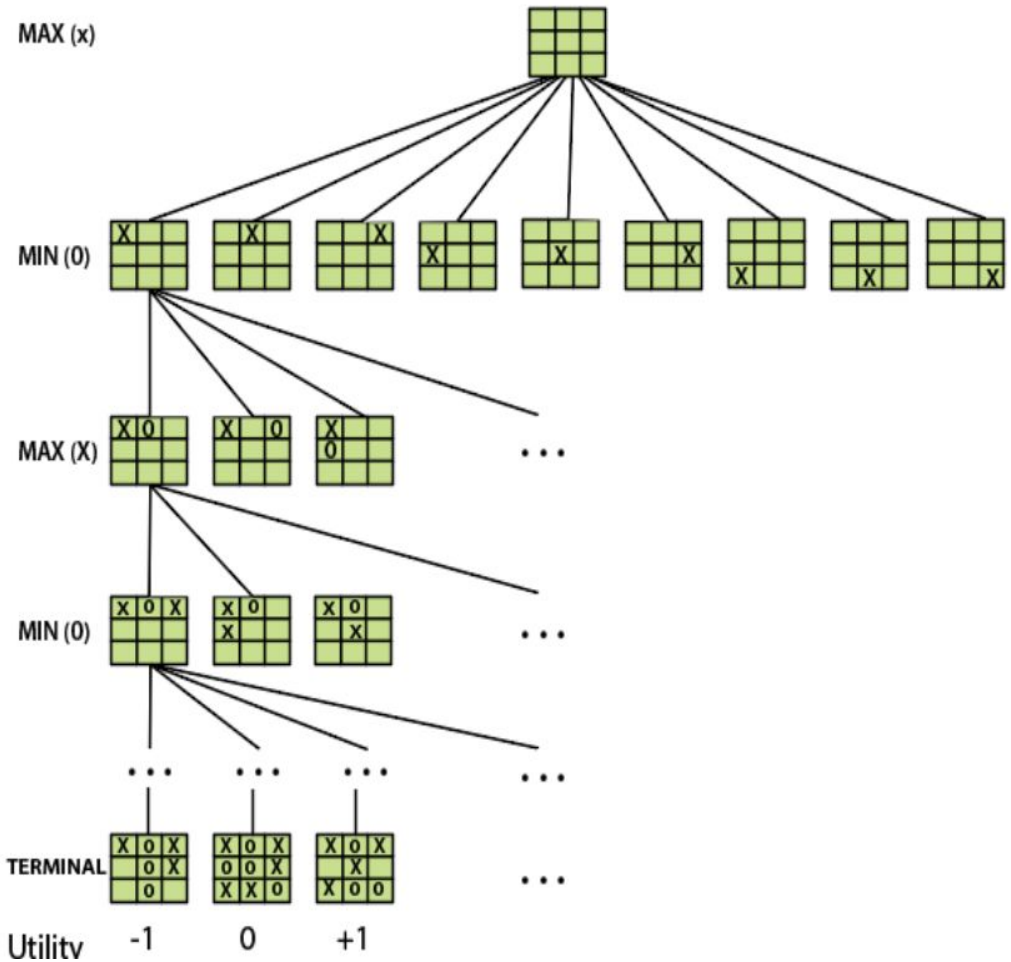
Game playing and knowledge structure-
Elements of Game Playing search

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

Game playing and knowledge structure-

Elements of Game Playing search for tic-tac-toe

- **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**.**



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**

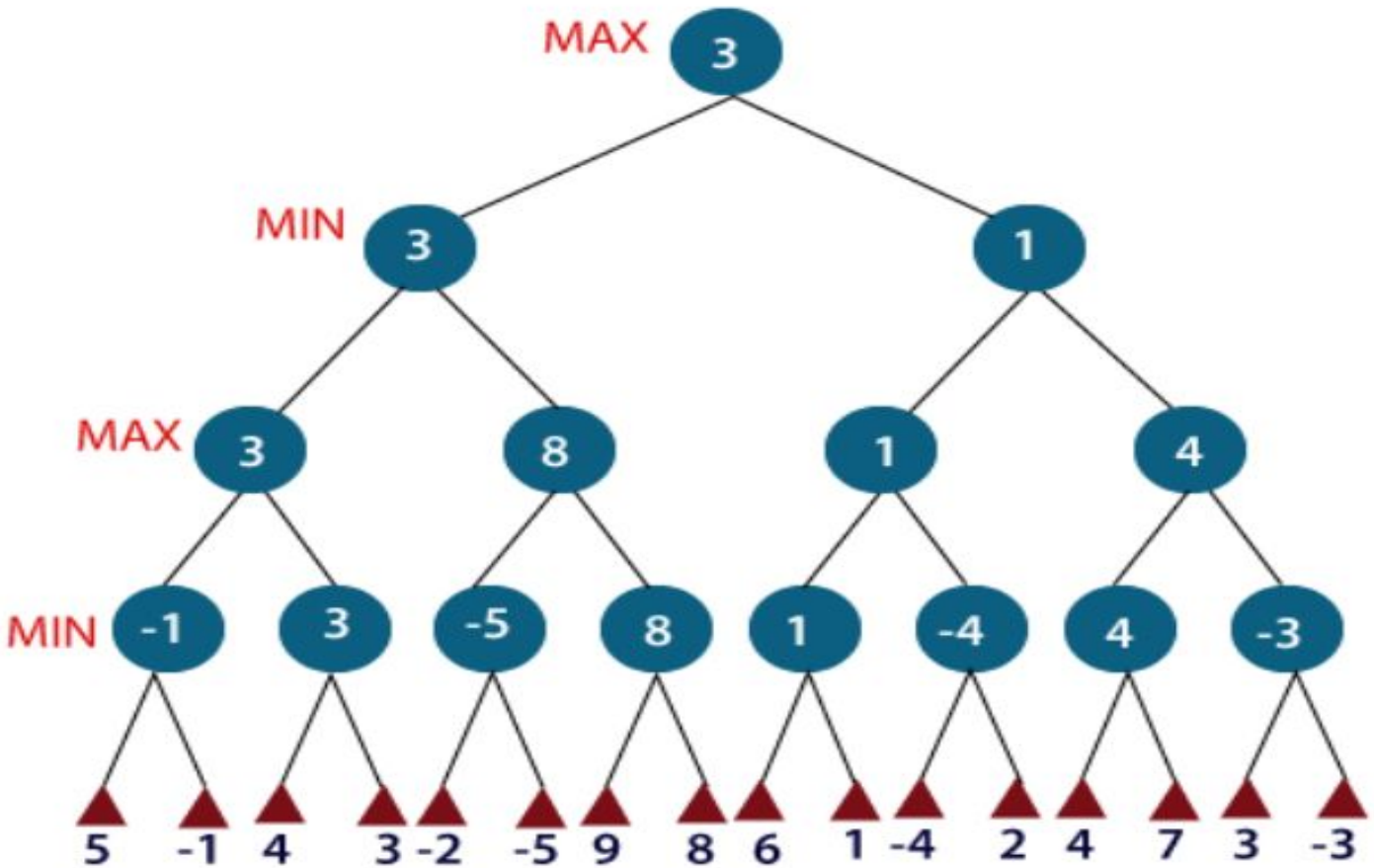
Game as a search problem

- 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

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.

Minimax Algorithm



Minimax Algorithm

- Properties of minimax algorithm:
- Complete? Yes (if tree is finite)
  Optimal? Yes (against an optimal opponent)
  Time complexity? $O(b^m)$

  m – maximum depth of tree; b branching factor
- Space complexity? O(bm) (depth-first exploration, if it generates all successors at once)

m – maximum depth of the tree; b – legal moves;

Minimax Algorithm

- Limitations
  - Not always feasible to traverse entire tree
  - Time limitations
- Key Improvement
  - Use evaluation function instead of utility
    - Evaluation function provides estimate of utility at given position

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

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.

Alpha beta pruning

α - Maximum value of the nodes.

β - Minimum value of the nodes.

Condition for Alpha-beta pruning:

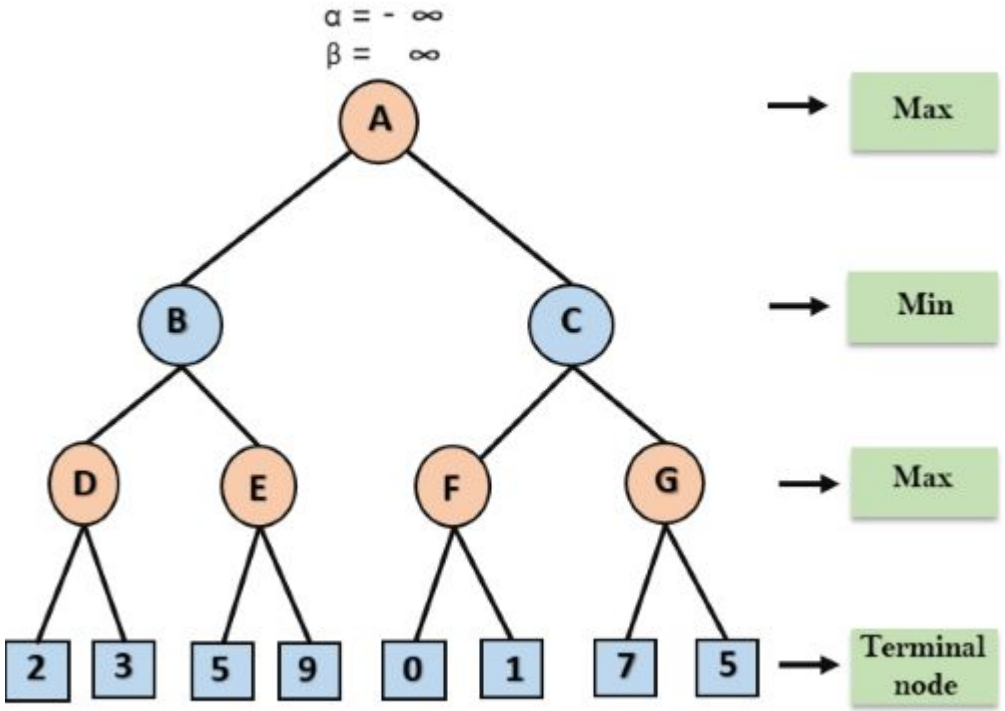
- The main condition which required for alpha-beta pruning is:

α >= β

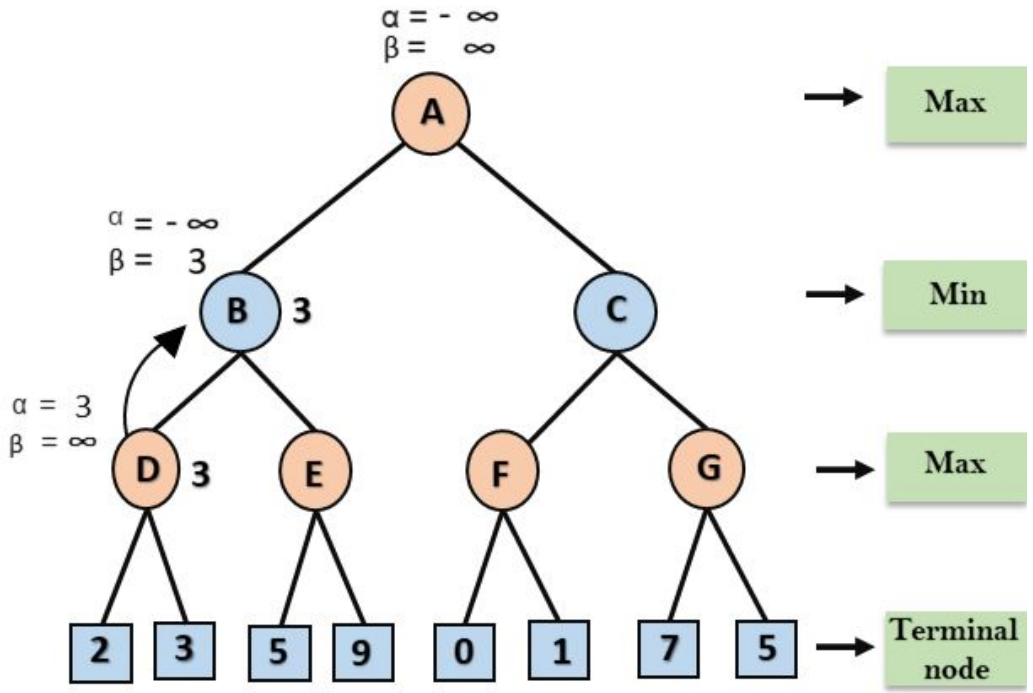
- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Alpha beta pruning

STEP 1:

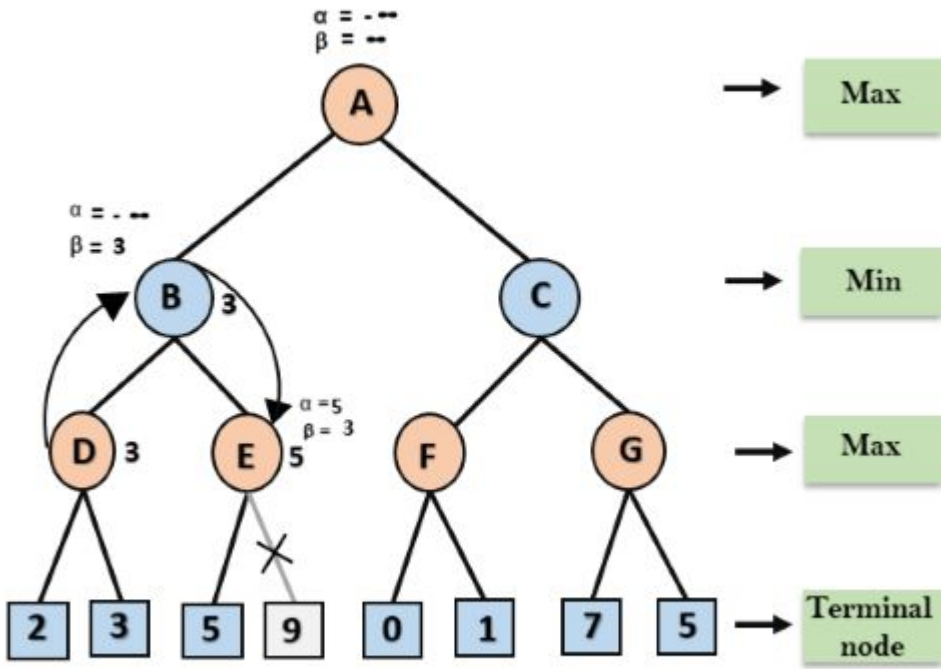


STEP 2:

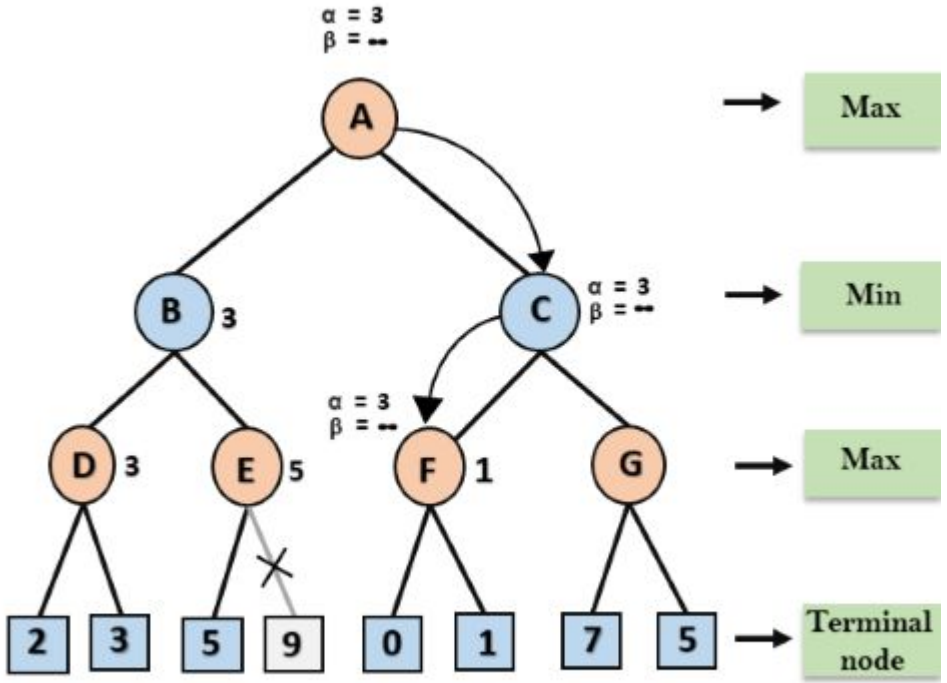


Alpha beta pruning

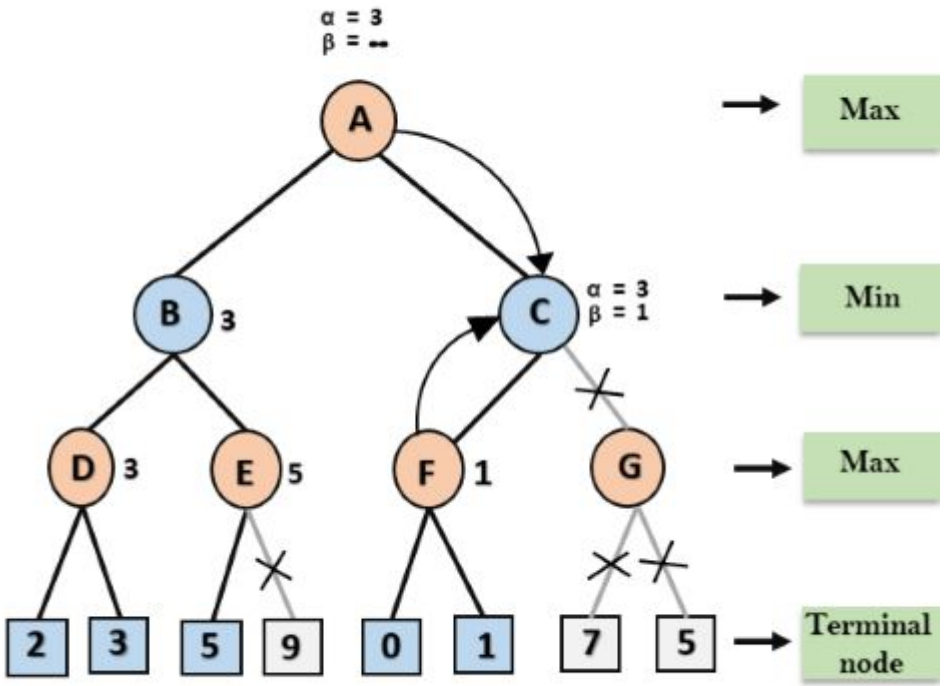
STEP 3:

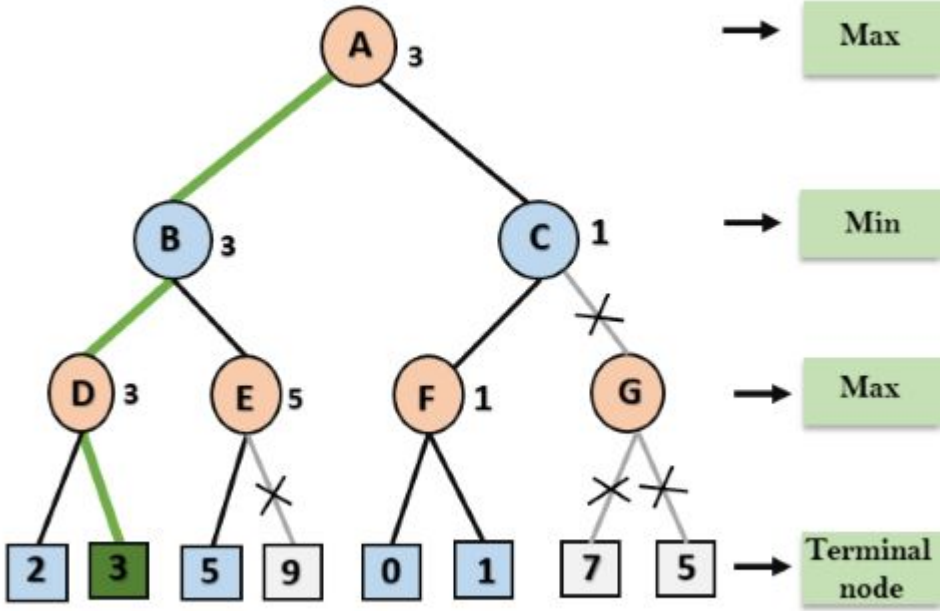


STEP 4:



Alpha beta pruning





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/

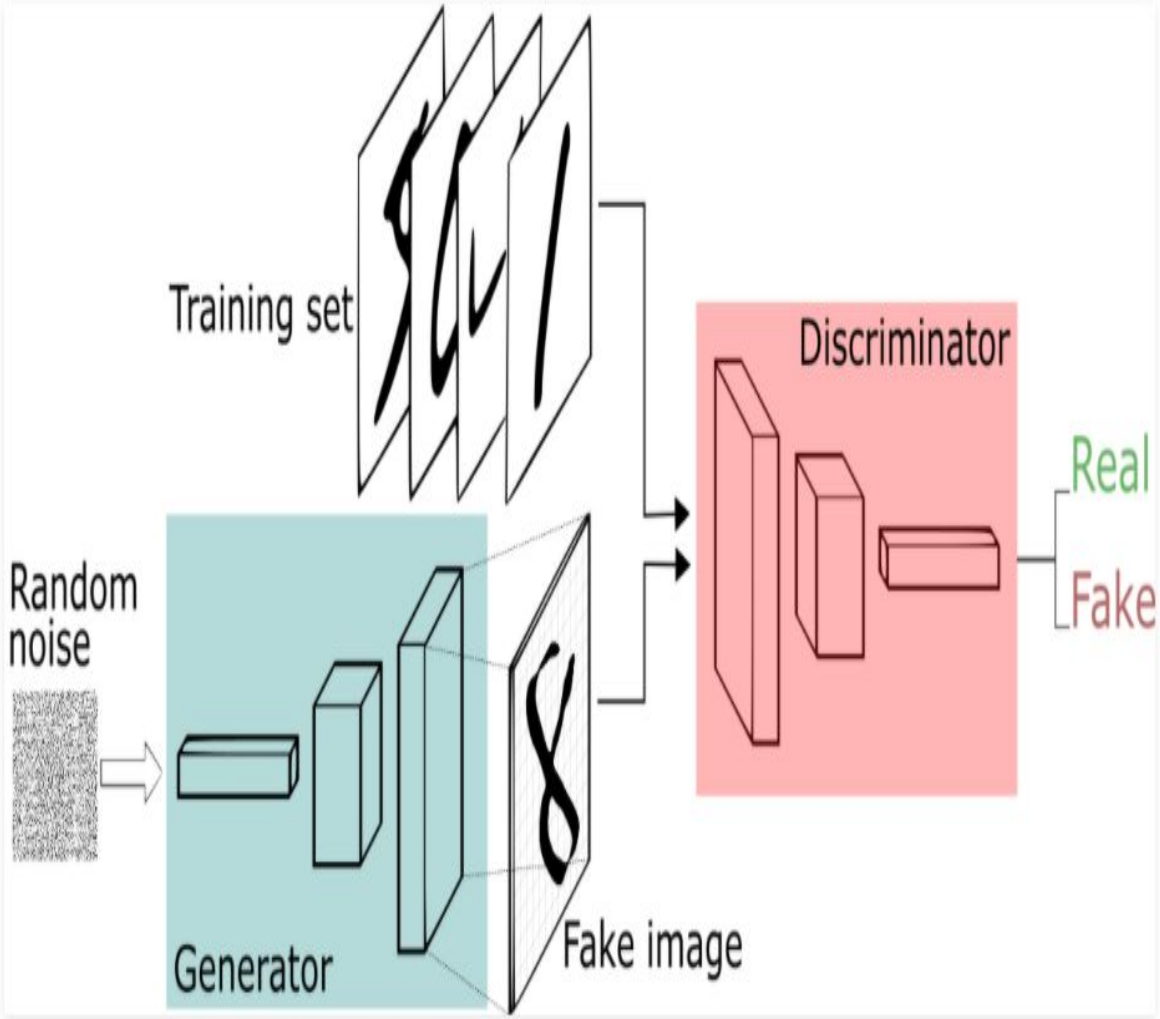
Game theory problems

- 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

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



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.

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/

Prisoner's Dilemma.

Players: The two suspects.
Actions: Each player's set of actions is {Quiet, Defect}.
Preferences Suspect 1's ordering of the action profiles, from best to worst, is (Defect, Quiet) (he defects and suspect 2 remains quiet, so he is freed), (Quiet, Quiet) (he gets one year in prison), (Defect, Defect) (he gets three years in prison), (Quiet, Defect) (he gets four years in prison). Suspect 2's ordering is (Quiet, Defect), (Quiet, Quiet), (Defect, Defect), (Defect, Quiet).
All of the above data can be represented in tabular form.

Suspect1/ Suspect 2	**Quiet**	**Defect**
Quiet	**2,2**	**0,3**
Defect	**3,0**	**1,1**

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