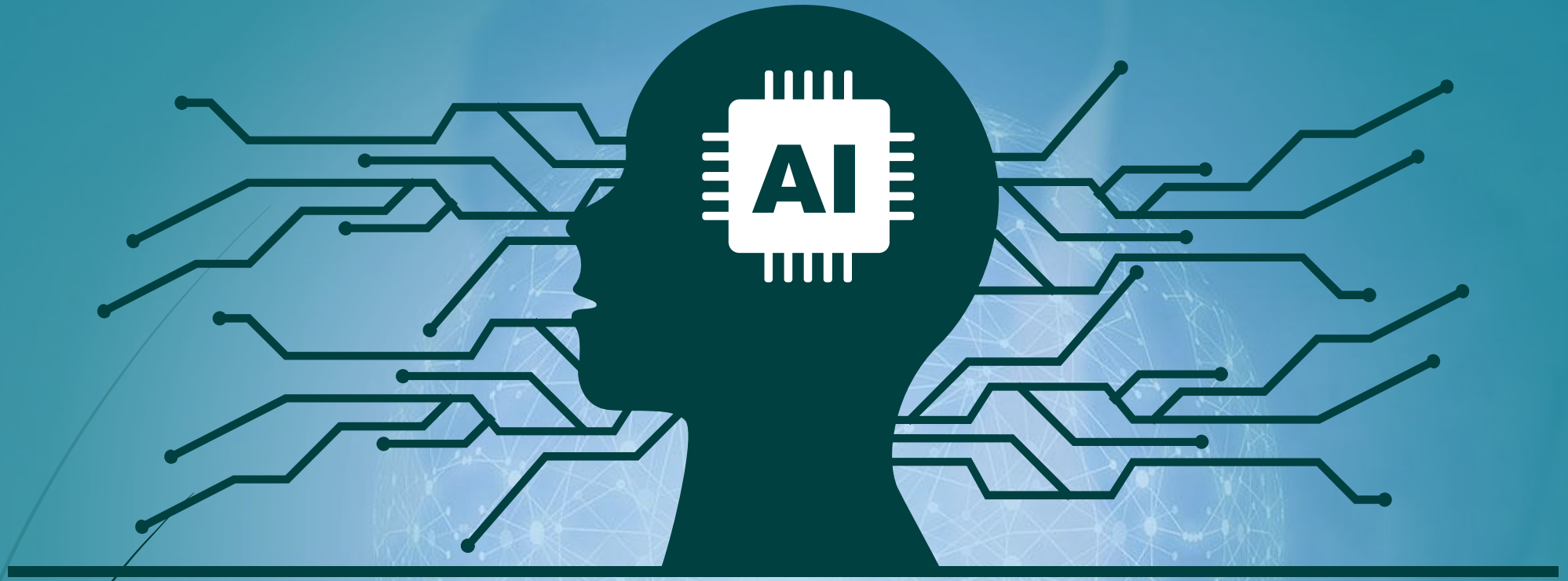


Advanced Artificial Intelligence

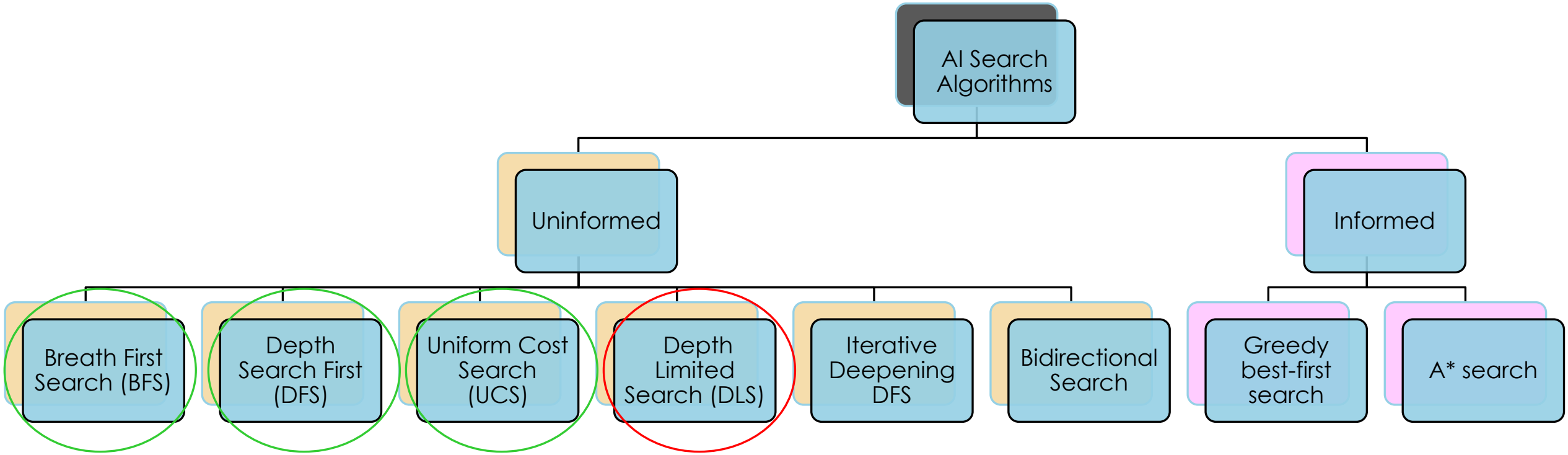
Week #3

Dr. Qurat Ul Ain
Assistant Professor
Dept. of AI & DS
FAST NUCES, Islamabad
Email:

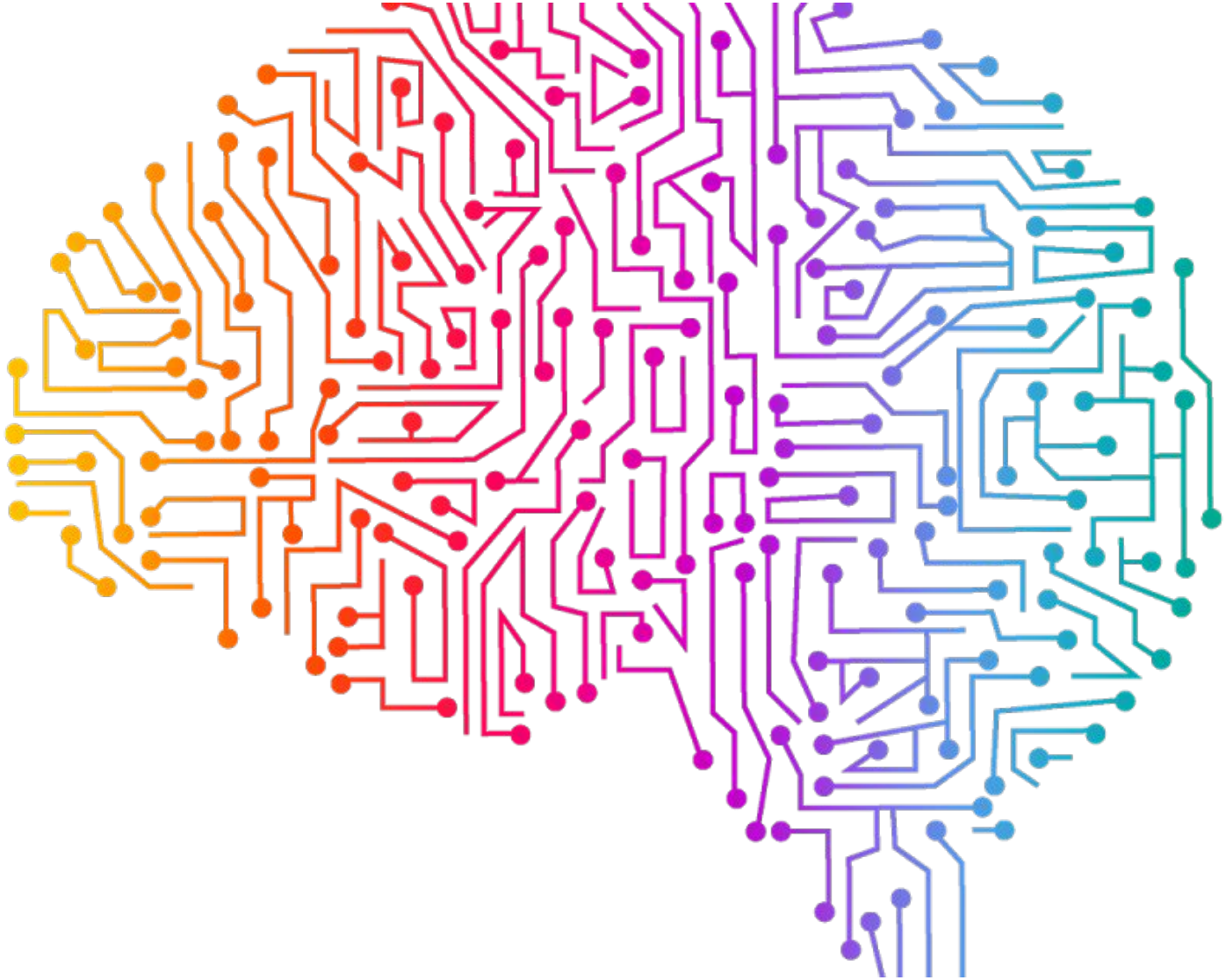
Uninformed Search Algorithms in Artificial Intelligence

Learning Objective of this Topic

- Uninformed Searching Algorithms
  - Uniform Cost Search (UCS)
  - Depth Limited Search (DLS)
  - Iterative Deepening Search (IDS)
  - Bidirectional Search (BS)


AI Search Algorithms
Uninformed
Informed
Breath First Search (BFS)
Depth Search First (DFS)
Uniform Cost Search (UCS)
Depth Limited Search (DLS)
Iterative Deepening DFS
Bidirectional Search
Greedy best-first search
A* search

Depth Limited Search (DLS)

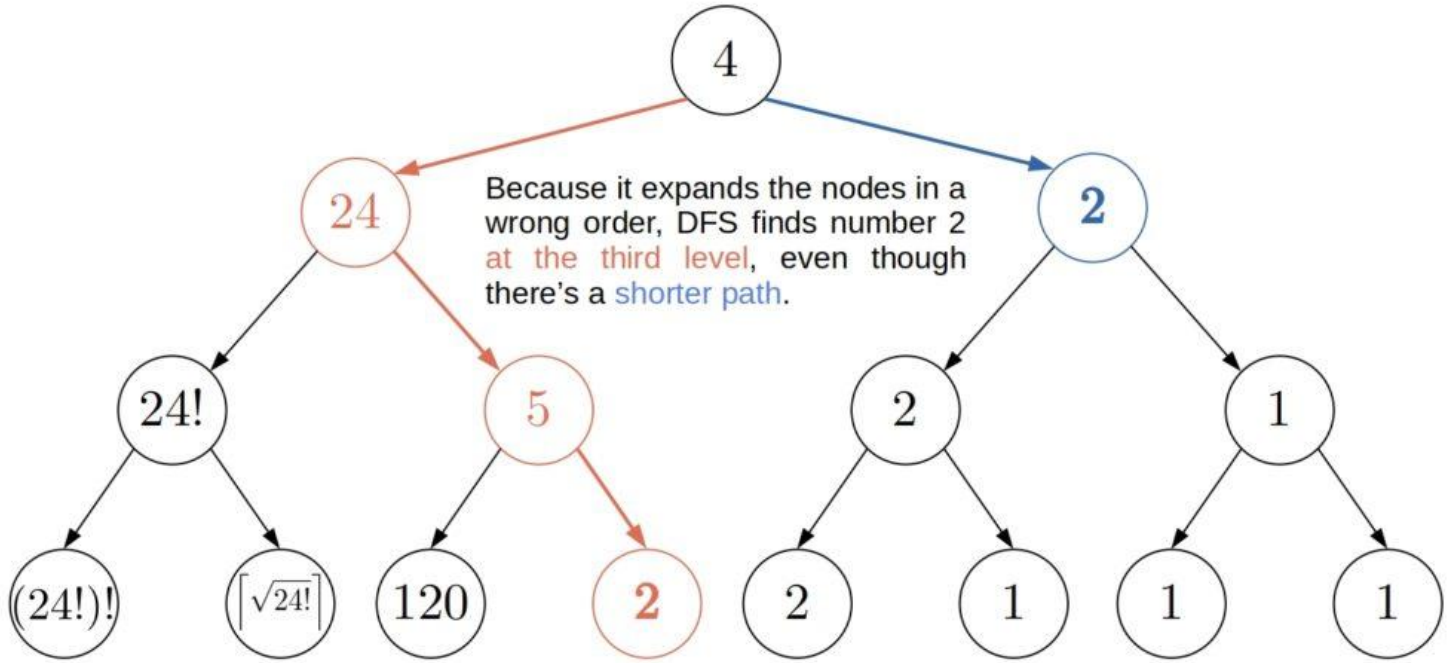


What is Depth-Limited Search (DLS)?

- The depth-limited search is a variation of a well-known depth-first search(DFS) traversing algorithm.
- It performs depth-first search (DFS) by implementing a depth limit.
- But why we need Depth Limited Search?

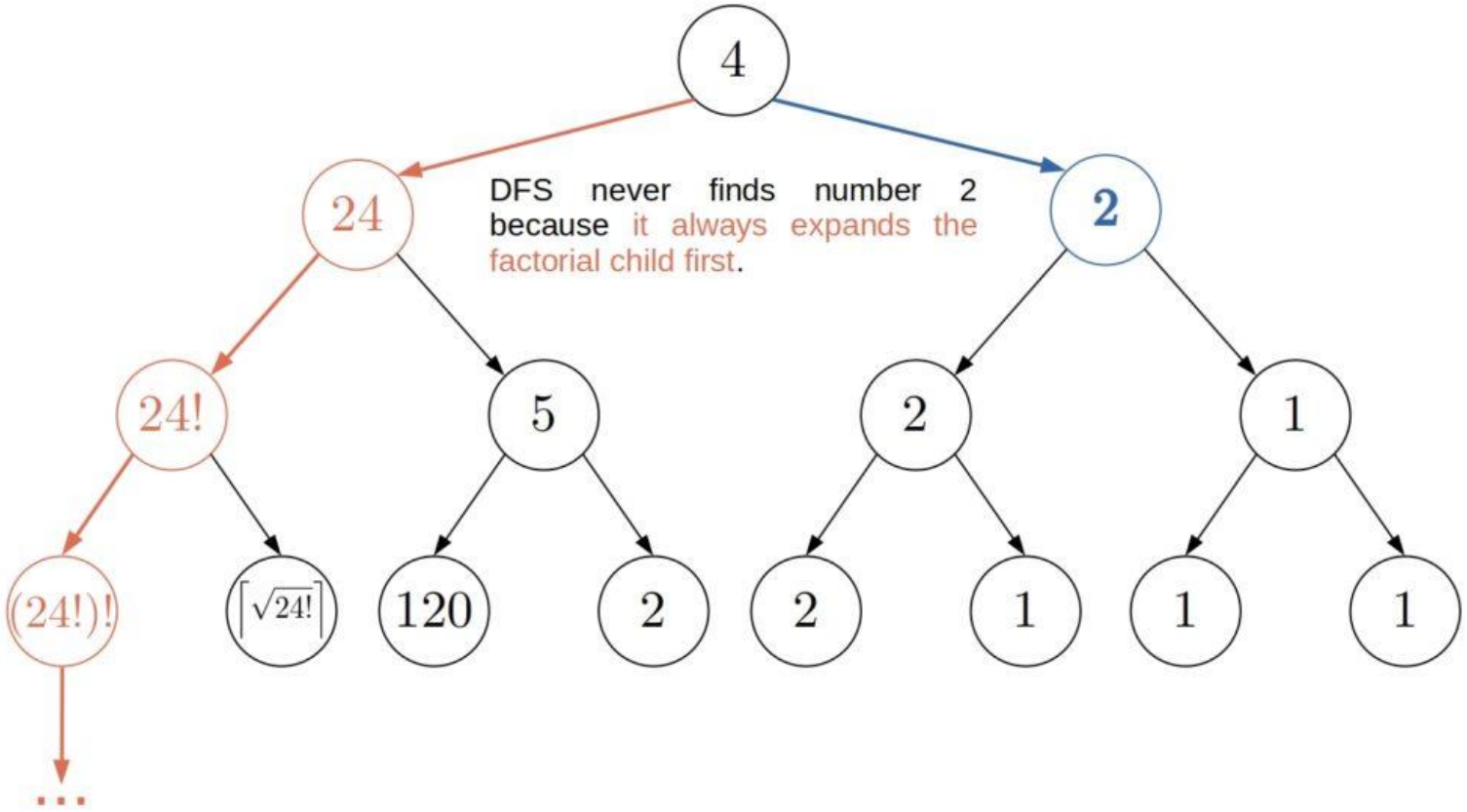
1- Problem with Depth First Search

- DFS may miss the optimal path. Depending on the order in which children returns the children of a node, DFS may expand more nodes than necessary:



2- Problem with Depth First Search

- Also, DFS may never end! It may get stuck at expanding the nodes that can't lead to a target node



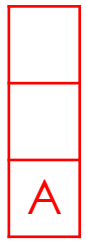
Depth Limited Search

- *limit:* the depth of the tree/graph until which we explore the nodes
- Expand Depth Node First (Depth Limit Search)
- *Stack*: nodes in the stack to be explored
- *explored:* Nodes that are already explored

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a node or failure
  Stack    ← a LIFO with source node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(Stack) then return failure
    node ← POP(Stack)  /* chooses the top node in Stack */
    add node to explored
    if node == goal then return SOLUTION
    for all adjacent node (n) of node
      if n is not in explored or Stack then
        Stack  ← INSERT(n)
```

Depth Limited Search

Stack =



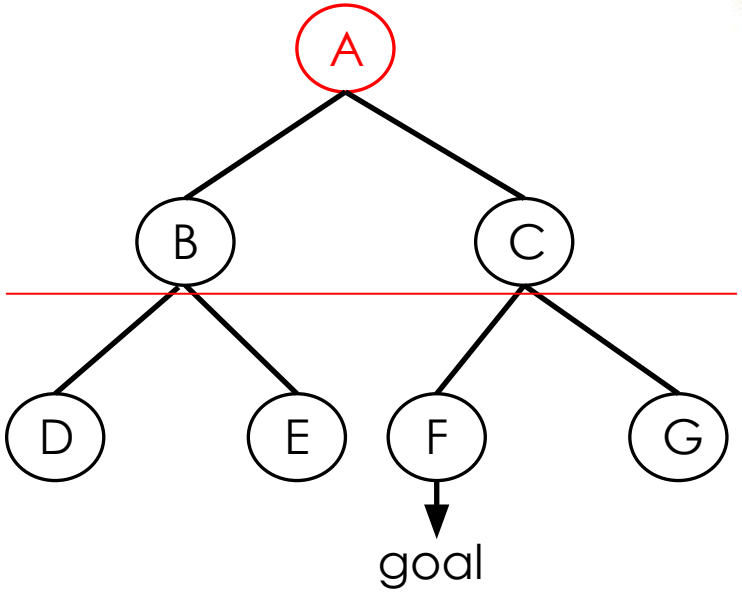
Explored = [ ]

Depth Limit = 1

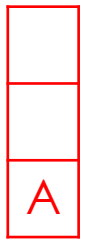
function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a node or *failure*

Stack ← a LIFO with *source node* as the only element
explored ← an empty set

loop do
  if EMPTY?(*Stack*) **then return** failure
  node ← POP(*Stack*) /* chooses the top node in *Stack* */
  add *node* to *explored*
  if *node* == *goal* **then return** SOLUTION
  for all *adjacent node* (*n*) of *node*
    if *n* is not in *explored* or *Stack* **then**
      Stack ← INSERT(*n*)

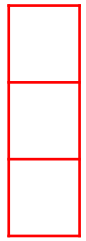


Stack =



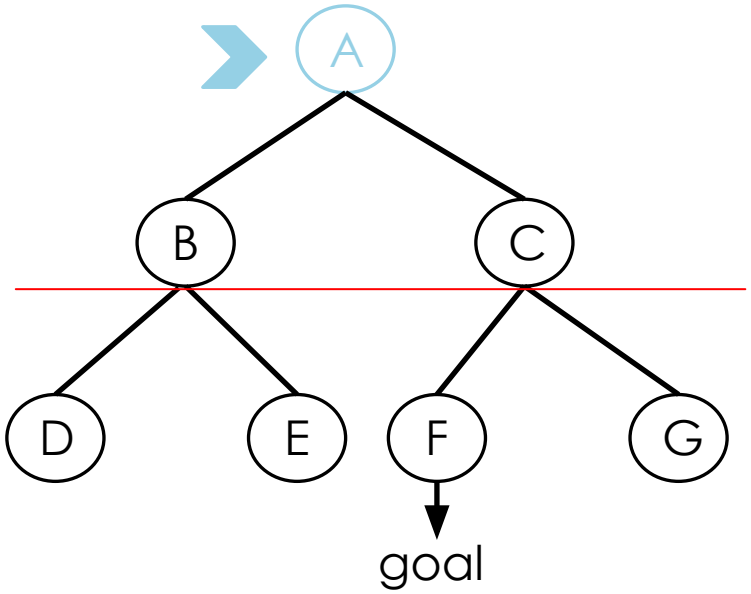
Explored = [ ]

Stack =



Is A a goal state?

Explored = [A]



Depth Limit = 1

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a node or *failure*
  Stack ← a LIFO with *source node* as the only element
  explored ← an empty set
  loop do
    if EMPTY?(*Stack*) **then return** failure
    node ← POP(*Stack*) /* chooses the top node in *Stack* */
    add *node* to *explored*
    if *node* == *goal* **then return** SOLUTION
    for all *adjacent node* (*n*) of *node*
      if *n* is not in *explored* or *Stack* **then**
        Stack ← INSERT(*n*)

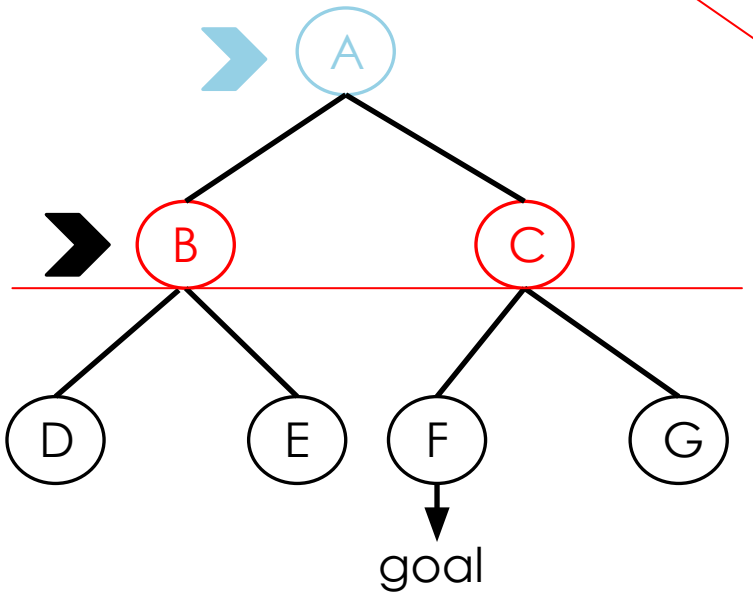
Stack =

Explored = [A]

Stack =

B

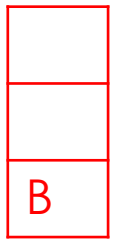
Explored = [A]



Depth Limit = 1

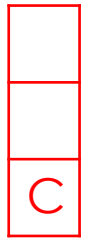
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a node or failure
   Stack     ← a LIFO with source node as the only element
   explored  ← an empty set
   loop do
      if EMPTY?(Stack ) then return failure
      node ← POP(Stack )   /* chooses the top node in Stack */
      add node to explored
      if node == goal then return SOLUTION
      for all adjacent node (n) of node
         if n is not in explored or Stack then
            Stack     ← INSERT( n )
```

Stack =



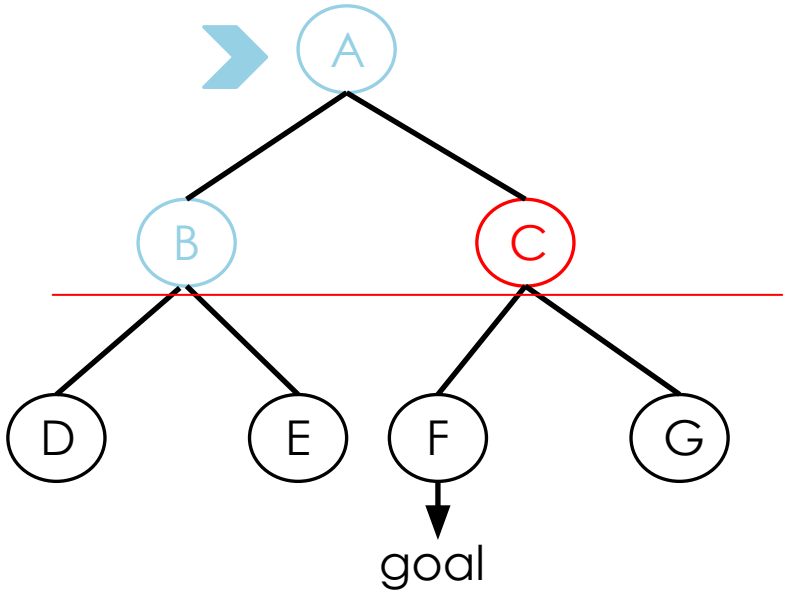
Explored = [A]

Stack =



Is B a goal state?

Explored = [A, B]



Depth Limit = 1

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a node or *failure*
  Stack ← a LIFO with *source node* as the only element
  explored ← an empty set
  loop do
    if EMPTY?(*Stack*) **then return** failure
    node ← POP(*Stack*) /* chooses the top node in *Stack* */
    add *node* to *explored*
    if *node* == *goal* **then return** SOLUTION
    for all *adjacent node* (*n*) of *node*
      if *n* is not in *explored* or *Stack* **then**
        Stack ← INSERT(*n*)

Stack =

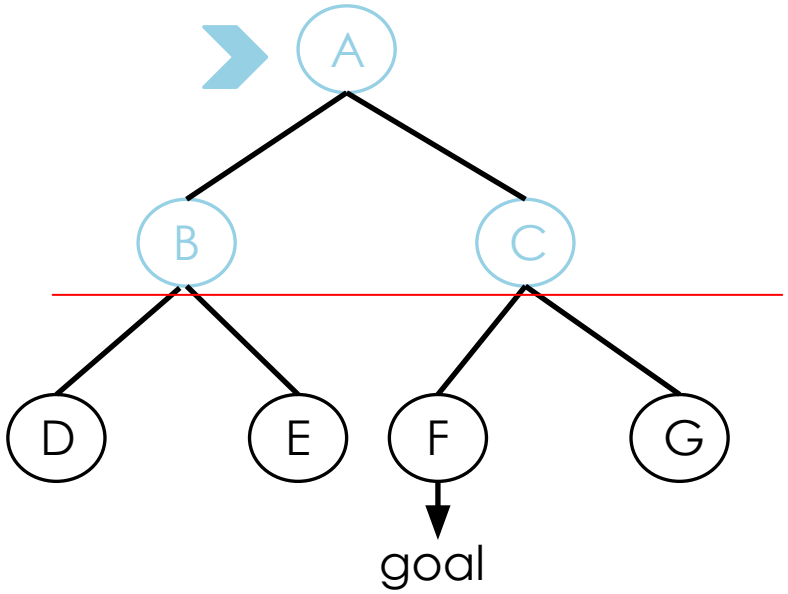
C

Explored = [ ]

Stack =

Is C a goal state?

Explored = [A, B, C]



Depth Limit = 1

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a node or failure
  Stack    ← a LIFO with source node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(Stack) then return failure
      node ← POP(Stack)  /* chooses the top node in Stack */
      add node to explored
      if node == goal then return SOLUTION
      for all adjacent node (n) of node
          if n is not in explored or Stack then
              Stack ← INSERT(n)
```

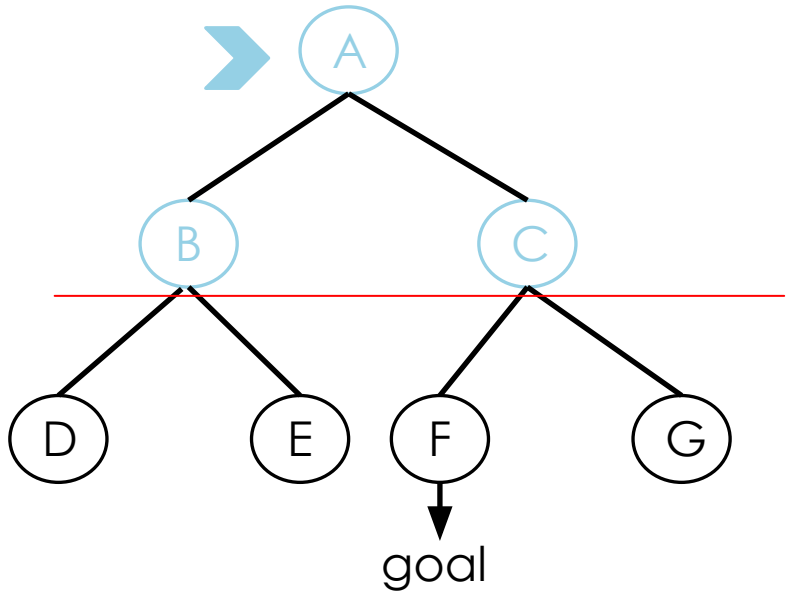
Stack =

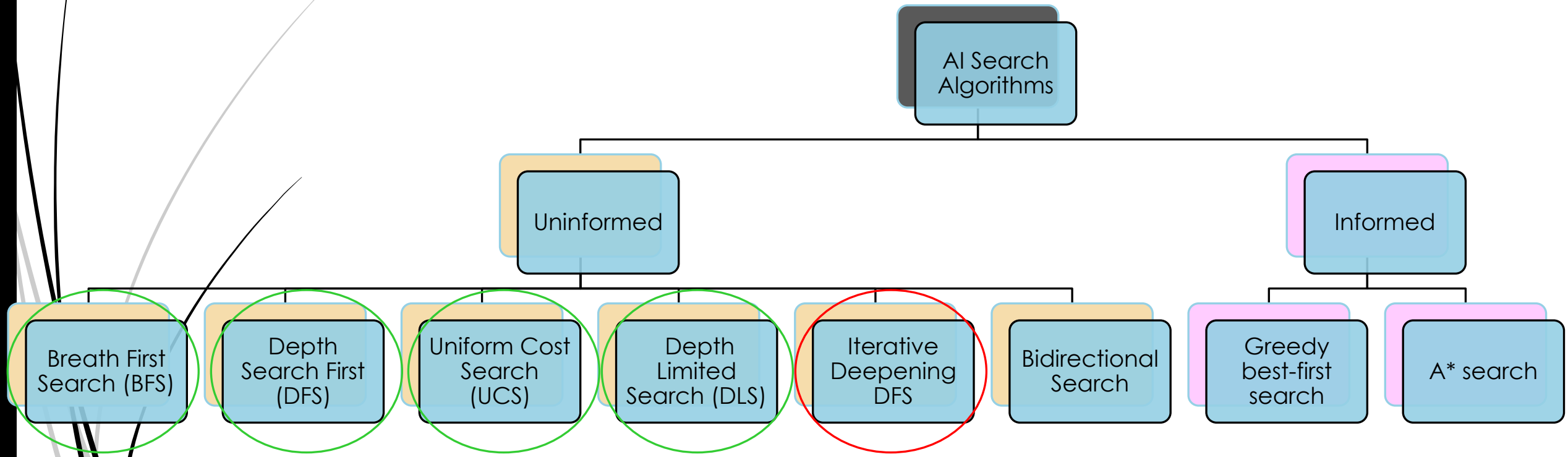
Explored = [A, B, C]

As Stack is empty, return failure !

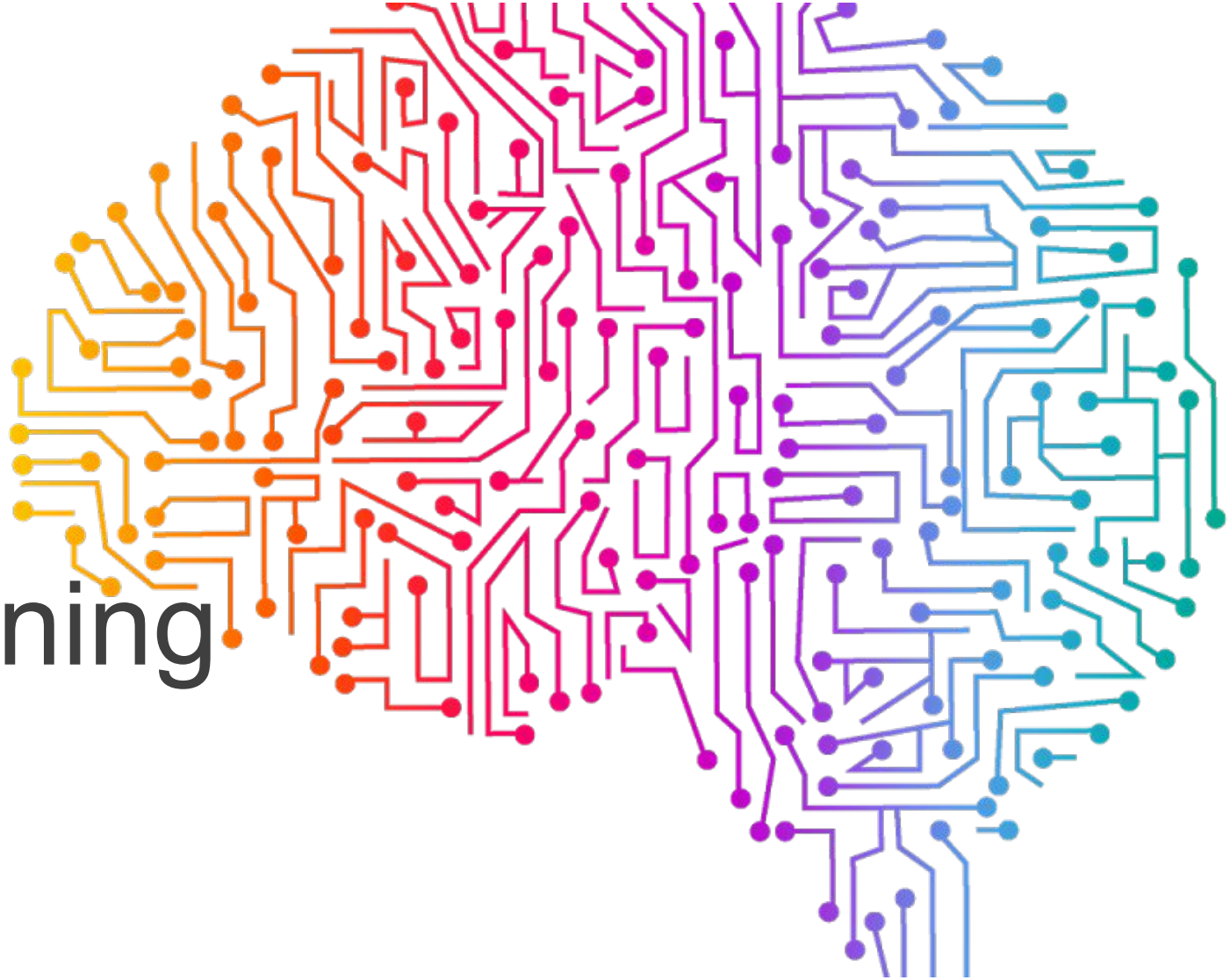
Depth Limit = 1

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a node or failure
  Stack ← a LIFO with source node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(Stack) then return failure
    node ← POP(Stack) /* chooses the top node in Stack */
    add node to explored
    if node == goal then return SOLUTION
    for all adjacent node (n) of node
      if n is not in explored or Stack then
        Stack ← INSERT(n)
```




AI Search Algorithms
Uninformed
Informed
Breath First Search (BFS)
Depth Search First (DFS)
Uniform Cost Search (UCS)
Depth Limited Search (DLS)
Iterative Deepening DFS
Bidirectional Search
Greedy best-first search
A* search

Iterative Deepening Search (IDS)



Iterative Deepening Search

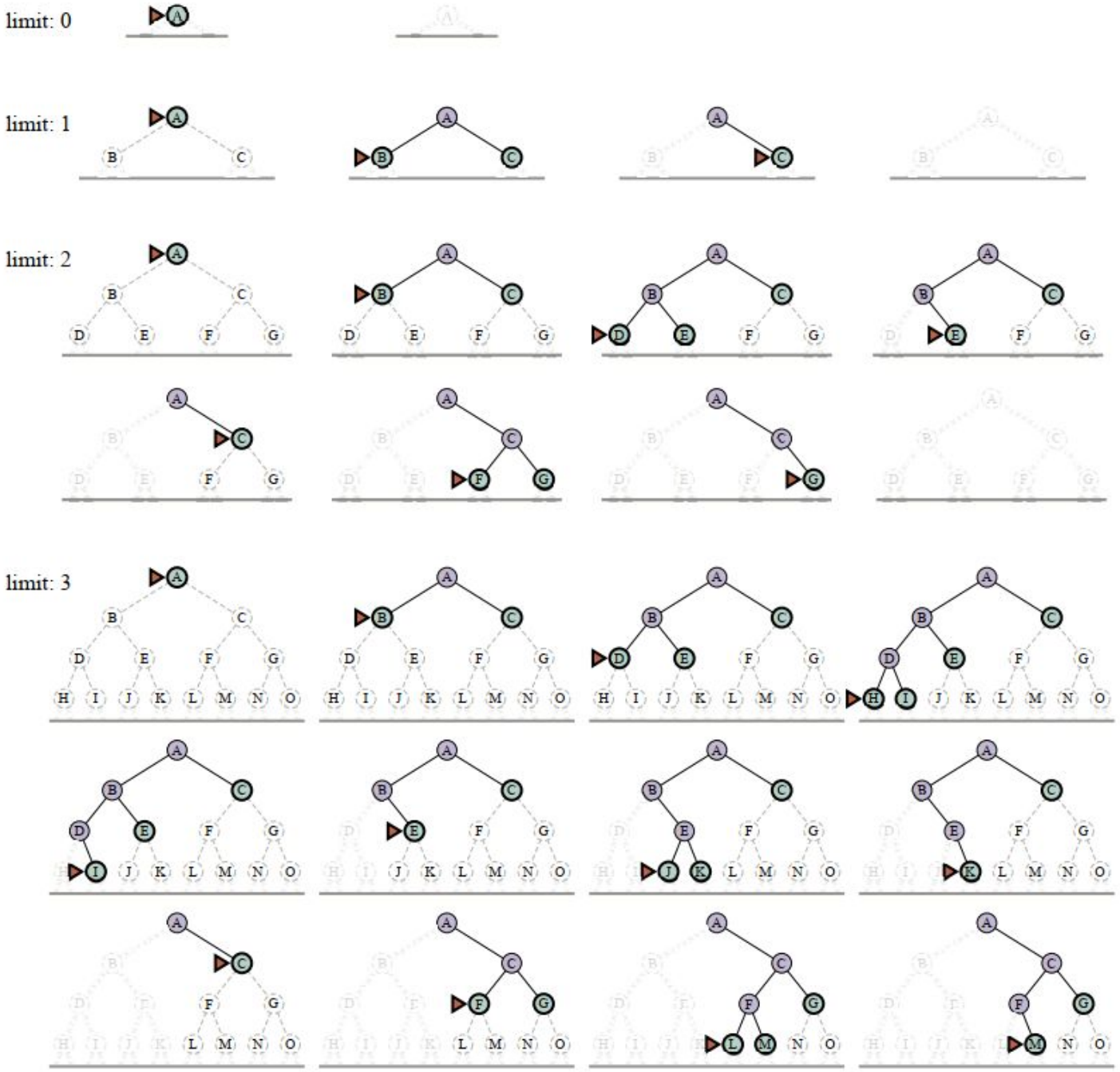
- What if the solution is deeper than Limit?
  - Increase depth iteratively
  - Iterative Deepening Search
- Iterative deepening search solves the problem of picking a good value for depth limit by trying all values: first 0, then 1, then 2, and so on—until either a solution is found, or the depth-limited search returns the failure.
- Also known as Iterative Deepening Depth First Search or Iterative Deepening Depth Limited Search.

Iterative Deepening Search

- Also known as Iterative Deepening Depth First Search or Iterative Deepening Depth Limited Search.

- IDS —GENERALLY THE PREFERRED UNINFORMED SEARCH
  - Inherits the memory advantage of depth-first search
  - Has the completeness property of breadth-first search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for limit = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, limit)
```

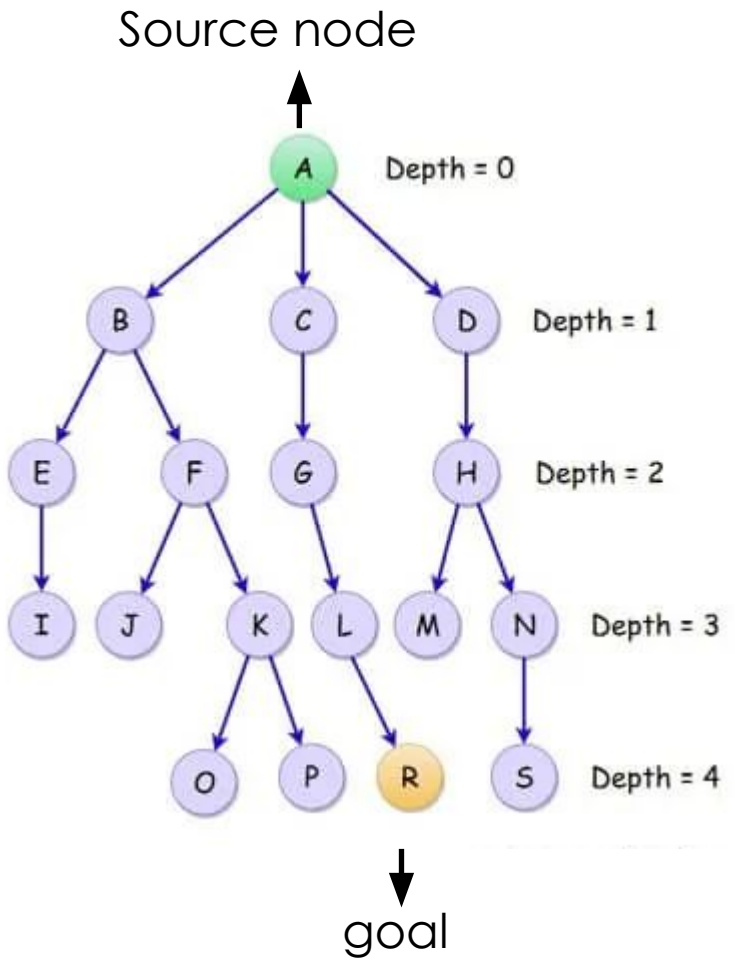


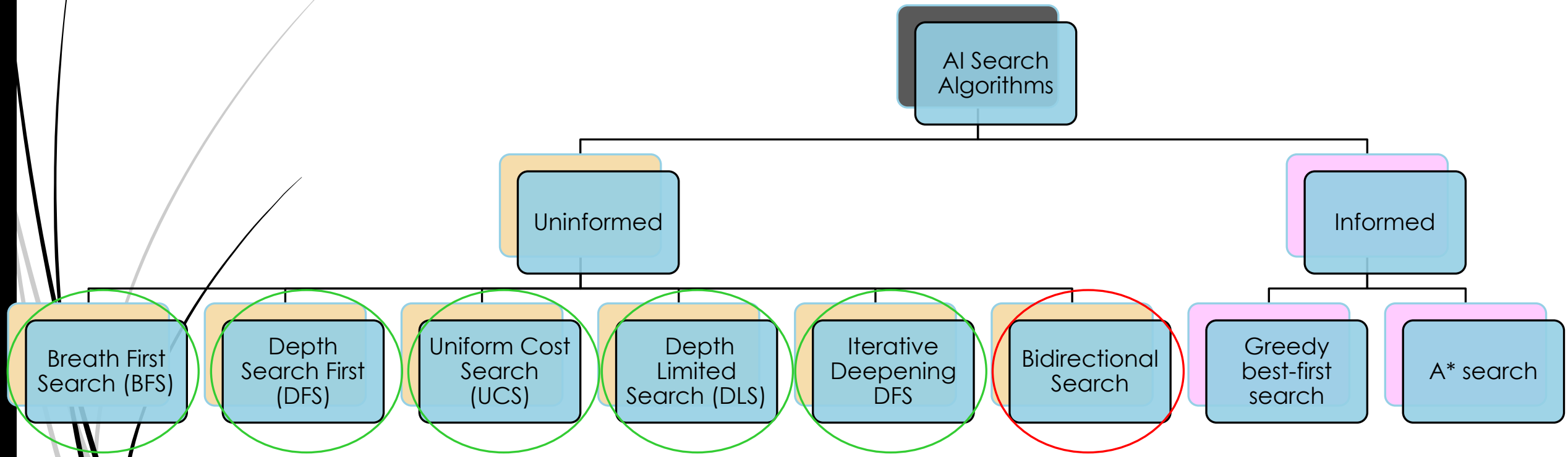
M is the goal node

Example IDS

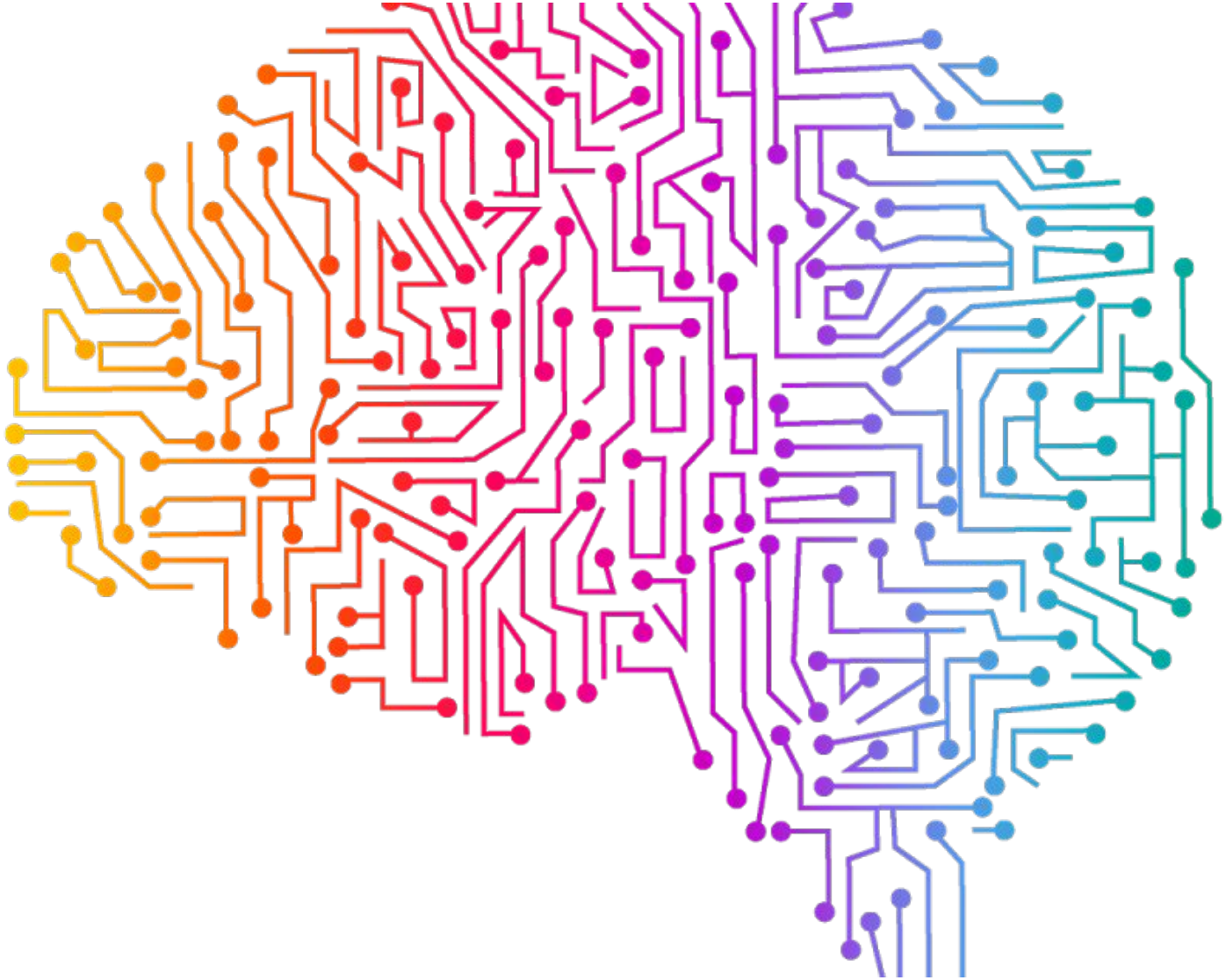
$DEPTH = \{0, 1, 2, 3, 4\}$

DEPTH LIMITS	IDDFS
0	A
1	A B C D
2	A B E F C G D H
3	A B E I F J K C G L D H M N
4	A B E I F J K O P C G L R




AI Search Algorithms
Uninformed
Informed
Breath First Search (BFS)
Depth Search First (DFS)
Uniform Cost Search (UCS)
Depth Limited Search (DLS)
Iterative Deepening DFS
Bidirectional Search
Greedy best-first search
A* search

Bidirectional Search (BS)



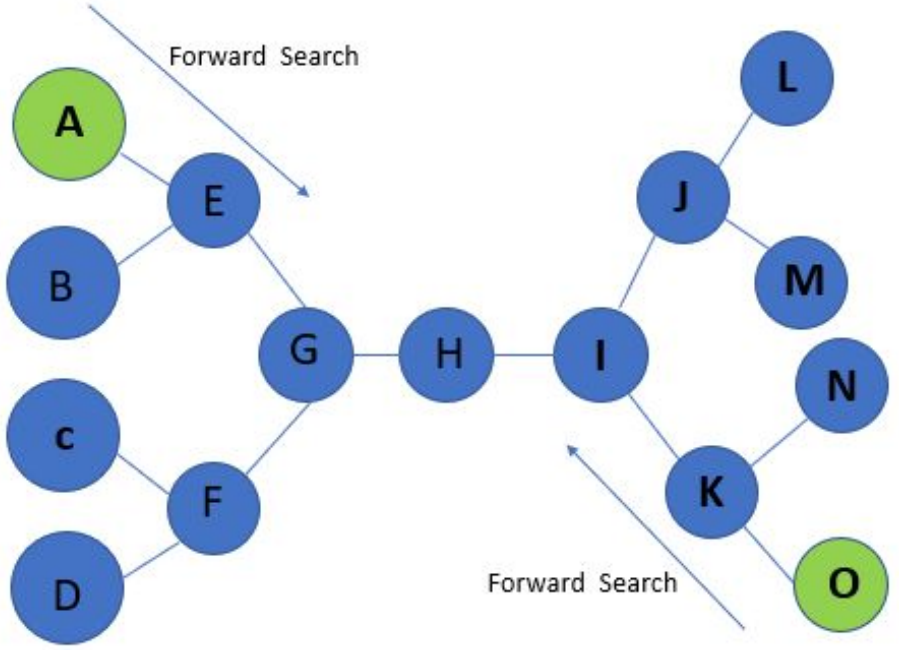
Bidirectional Search

- Bidirectional search is a graph search algorithm that finds the smallest path from the source to the goal vertex. It runs two simultaneous searches –

- Forward search from source/initial vertex toward goal vertex
- Backward search from goal/target vertex toward source vertex

Steps in Bidirectional Search

- **Step 1**: Say, A is the initial node and O is the goal node, and H is the intersection node.
- **Step 2**: We will start searching simultaneously from start to goal node and backward from goal to start node.
- **Step 3**: Whenever the forward search and backward search intersect at one node, then the searching stops.



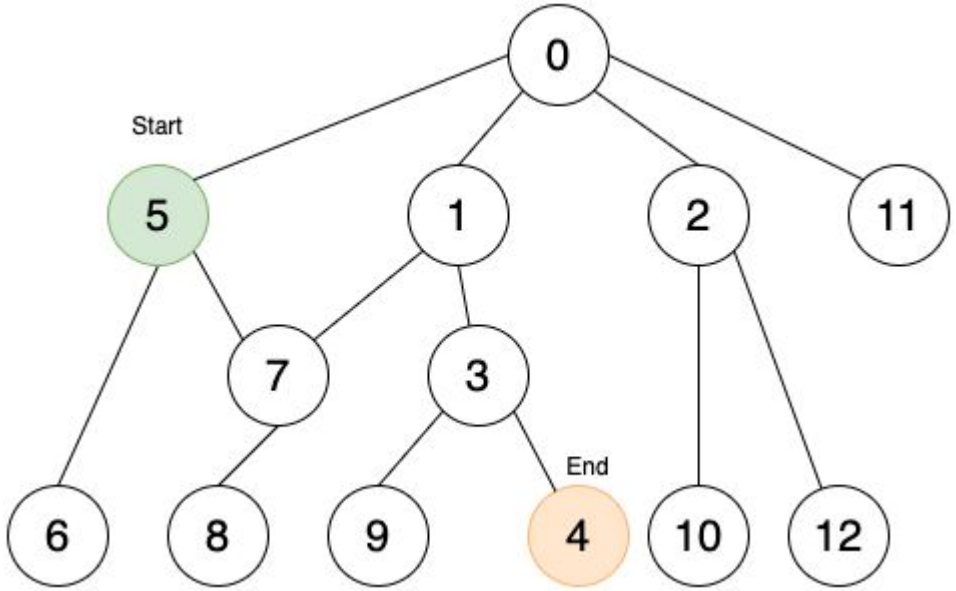
- **Completeness** : Bidirectional search is complete if BFS is used in both searches.

- **Optimality** : It is optimal if BFS is used for search and paths have uniform cost.

Example

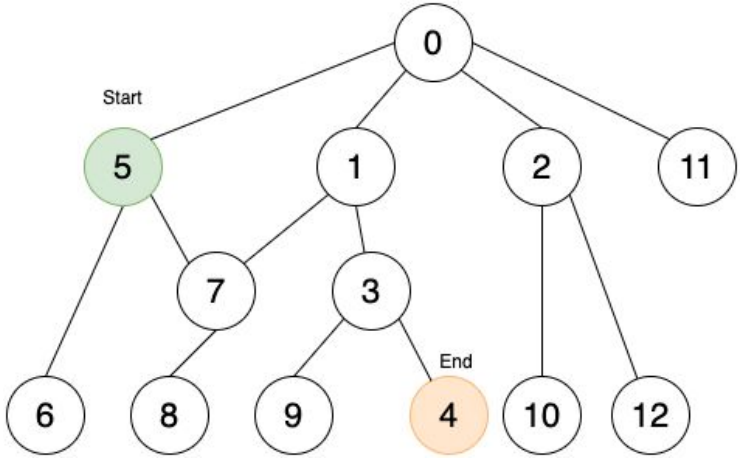
- The start node is 5 and the end node is 4.

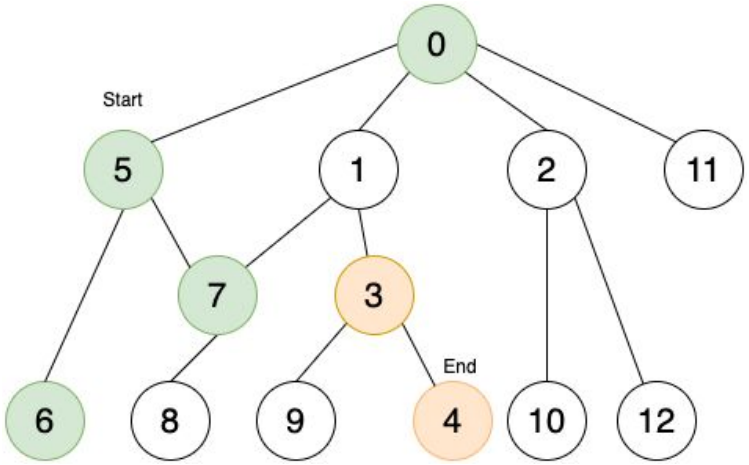
Aim: To find the shortest path from 5 to 4 using bidirectional search.



Do BFS from both directions.

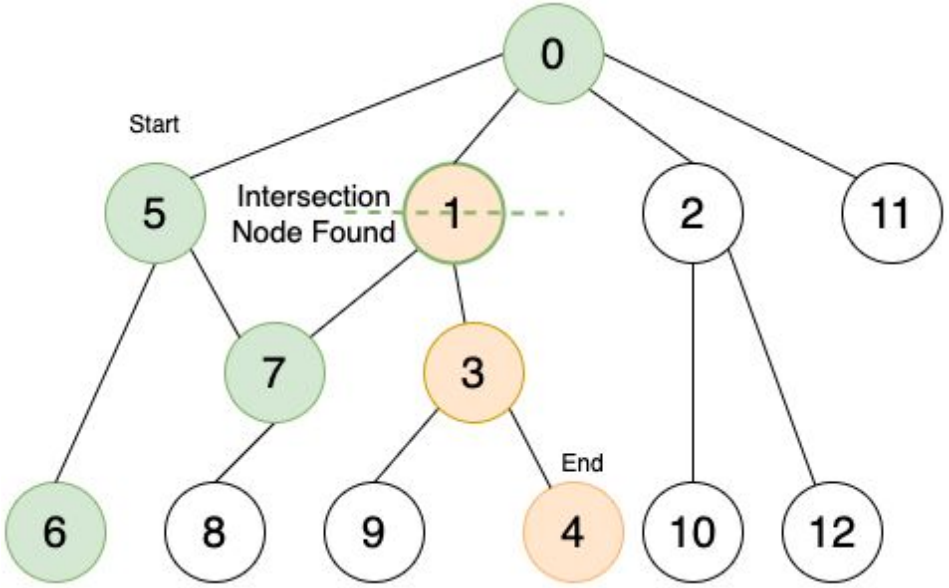
1. Start moving forward from start node (Green) and backwards from end node (Orange).

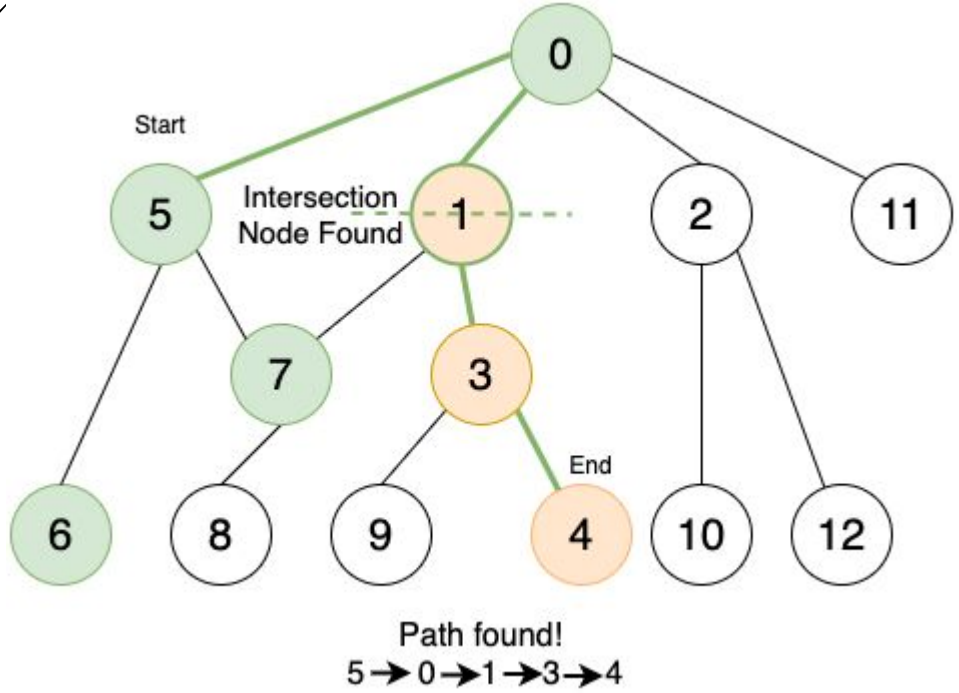




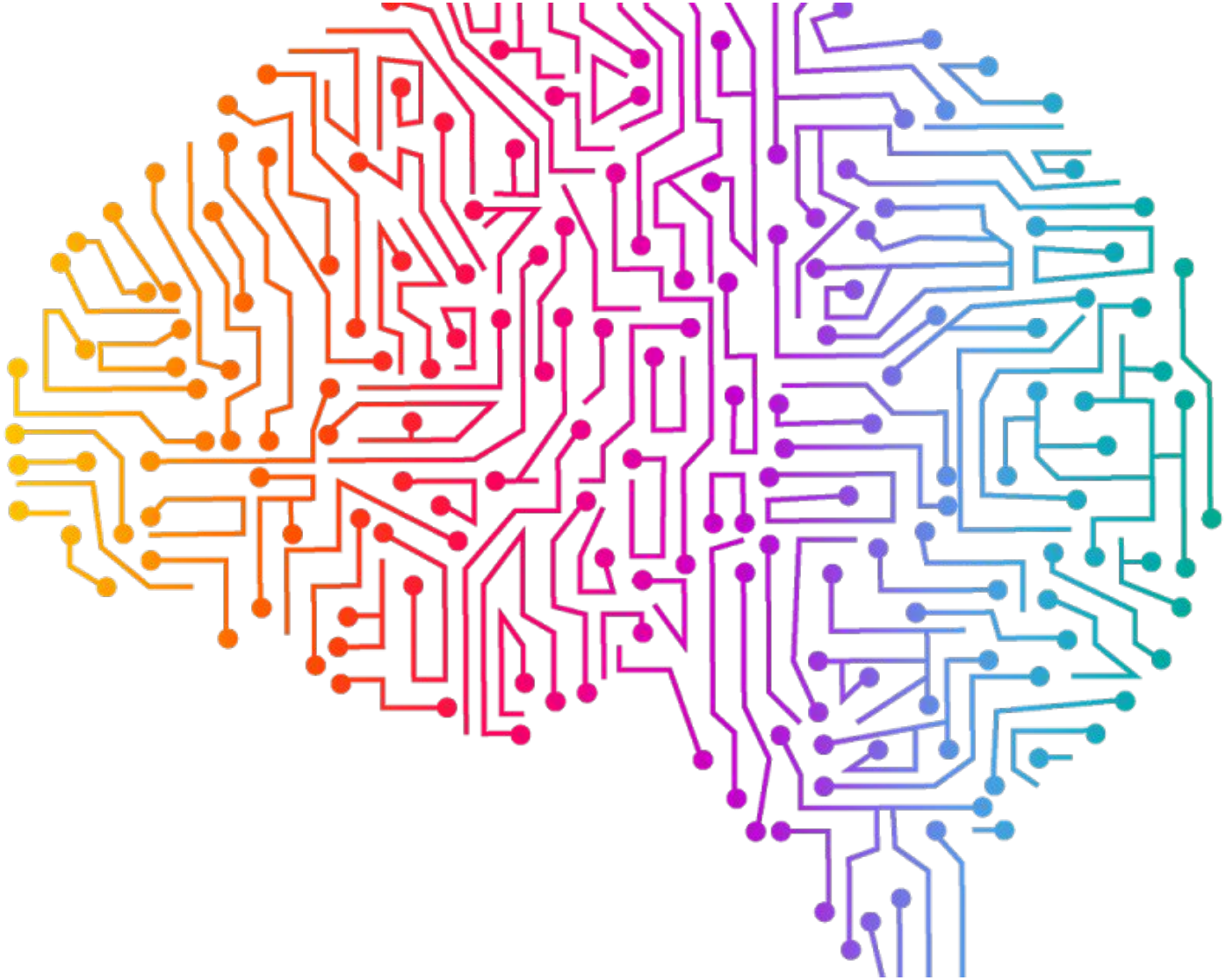
2. Similar to BFS, at every point explore the next level of nodes till you find an intersecting node.

3. Stop on finding the intersecting node.





4. Trace back to find the path.



Comparing Uninformed Searching Algorithms

Comparing Uninformed Searching Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, ℓ is the depth limit.

[1] complete if b is finite, and the state space either has a solution

[2] complete if all action costs are $\geq \epsilon > 0$;

[3] cost-optimal if action costs are all identical;

[4] if both directions are breadth-first or uniform-cost.

ϵ *is an arbitrarily small positive quantity*


Questions
?
Answers
?
?