Artificial Intelligence (AI) covers a range of techniques that appear as sentient behavior by the computer. For example, AI is used to recognize faces in photographs on your social media, beat the World's Champion in chess, and process your speech when you speak to Siri or Alexa on your phone.

Important Terms:

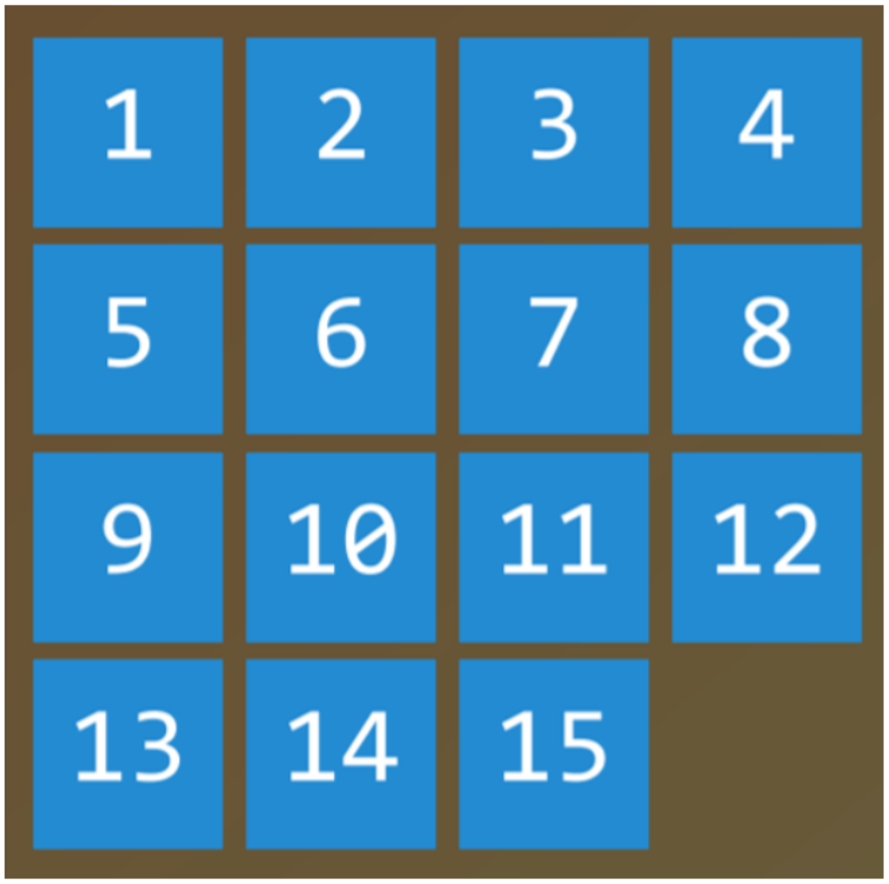
- **Search:** Finding a solution to a problem, like a navigator app that finds the best route from your origin to the destination, or like playing a game and figuring out the next move.
- **Knowledge:**Representing information and drawing inferences from it.
- **Uncertainty:** Dealing with uncertain events using probability.
- **Optimization:** Finding not only a correct way to solve a problem, but a better—or the best—way to solve it.
- **Learning:** Improving performance based on access to data and experience. For example, your email is able to distinguish spam from non-spam mail based on past experience.
- **Neural Networks:** A program structure inspired by the human brain that is able to perform tasks effectively.
- **Language:** Processing natural language, which is produced and understood by humans.

Search:

In which we see how an agent can find a sequence of actions that achieves its goals when no single action will do.

Search Problem:

Search problems involve an agent that is given an initial state and a goal state, and it returns a solution of how to get from the former to the latter. A navigator app uses a typical search process, where the agent (the thinking part of the program) receives as input your current location and your desired destination, and, based on a search algorithm, returns a suggested path. However, there are many other forms of search problems, like puzzles or mazes.



Finding a solution to a 15 puzzle would require the use of a search algorithm.

Agent:

An entity that perceives its environment and acts upon that environment. In a navigator app, for example, the agent would be a representation of a car that needs to decide on which actions to take to arrive at the destination.

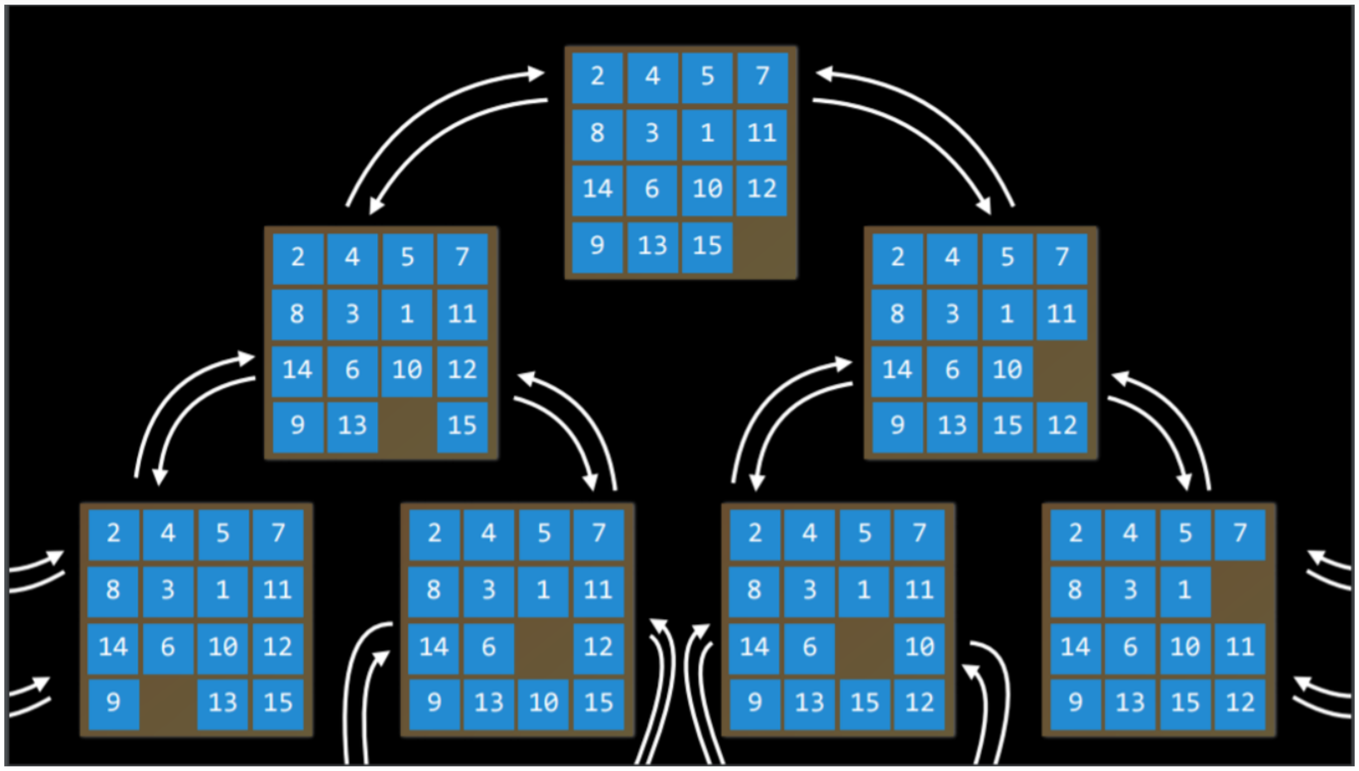
State:
A configuration of an agent in its environment. For example, in a 15 puzzle, a state is any one way that all the numbers are arranged on the board.

- **Initial State:**
  The state from which the search algorithm starts. In a navigator app, that would be the current location.

Actions:
Choices that can be made in a state. More precisely, actions can be defined as a function. Upon receiving state s as input, Actions(s) returns as output the set of actions that can be executed in state s. For example, in a 15 puzzle, the actions of a given state are the ways you can slide squares in the current configuration (4 if the empty square is in the middle, 3 if next to a side, 2 if in the corner).

Transition Model:
A description of what state results from performing any applicable action in any state. More precisely, the transition model can be defined as a function. Upon receiving state s and action a as input, Results(s, a) returns the state resulting from performing action a in state s. For example, given a certain configuration of a 15 puzzle (state s), moving a square in any direction (action a) will bring to a new configuration of the puzzle (the new state).

State Space
The set of all states reachable from the initial state by any sequence of actions. For example, in a 15 puzzle, the state space consists of all the 16!/2 configurations on the board that can be reached from any initial state. The state space can be visualized as a directed graph with states, represented as nodes, and actions, represented as arrows between nodes.



Goal Test
The condition that determines whether a given state is a goal state. For example, in a navigator app, the goal test would be whether the current location of the agent (the representation of the car) is at the destination. If it is — problem solved. If it's not — we continue searching.

Path Cost
A numerical cost associated with a given path. For example, a navigator app does not simply bring you to your goal; it does so while minimizing the path cost, finding the fastest way possible for you to get to your goal state.

Solving Search Problem:

Solution:A sequence of actions that leads from the initial state to the goal state.

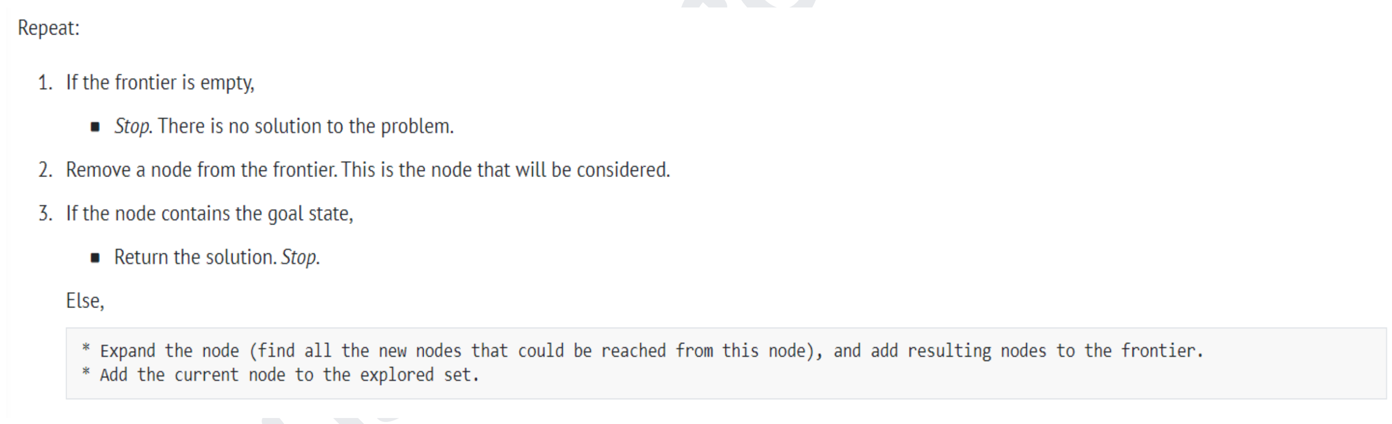
- Optimal Solution: A solution that has the lowest path cost among all solutions.

In a search process, data is often stored in a node, a data structure that contains the following data:

- A state
- Its parent node, through which the current node was generated
- The action that was applied to the state of the parent to get to the current node
- The path cost from the initial state to this node

Nodes contain information that makes them very useful for the purposes of search algorithms. They contain a state, which can be checked using the goal test to see if it is the final state. If it is, the node's path cost can be compared to other nodes' path costs, which allows choosing the optimal solution. Once the node is chosen, by virtue of storing the parent node and the action that led from the parent to the current node, it is possible to trace back every step of the way from the initial state to this node, and this sequence of actions is the solution.

However, nodes are simply a data structure — they don't search, they hold information. To actually search, we use the frontier, the mechanism that "manages" the nodes. The frontier starts by containing an initial state and an empty set of explored items, and then repeats the following actions until a solution is reached:

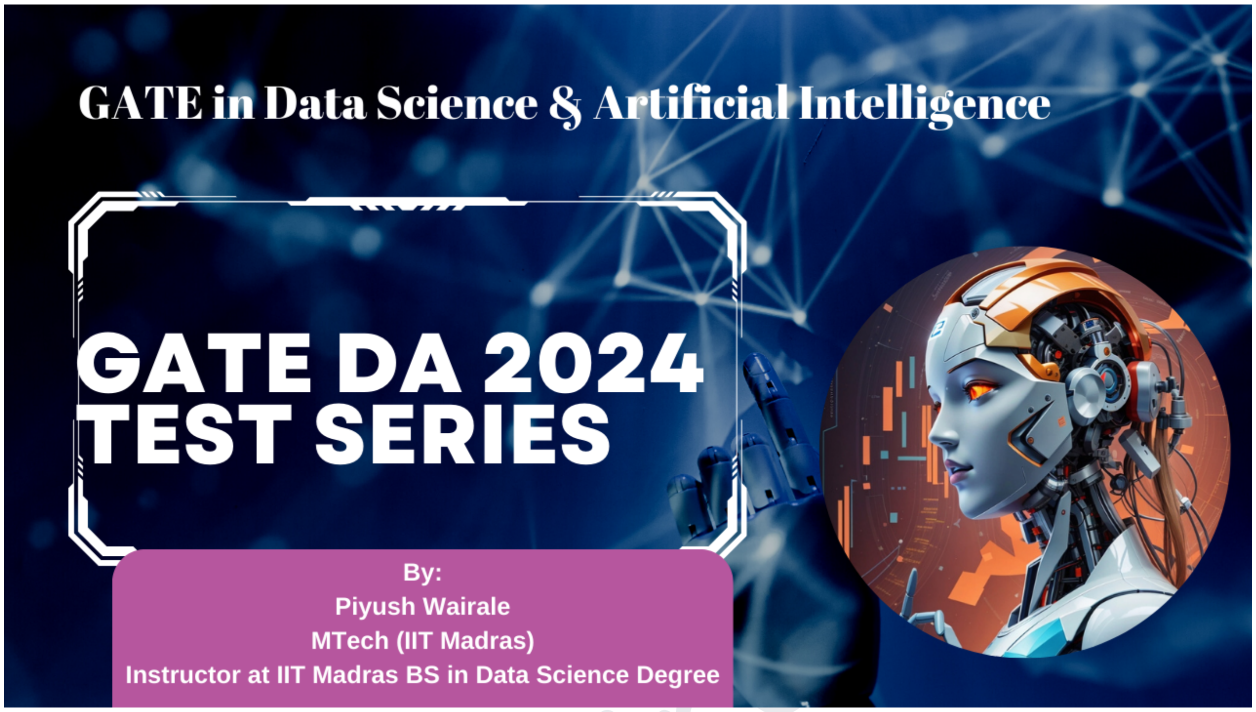


Repeat:

1. If the frontier is empty,
   - *Stop*. There is no solution to the problem.
2. Remove a node from the frontier. This is the node that will be considered.
3. If the node contains the goal state,
   - Return the solution. *Stop*.

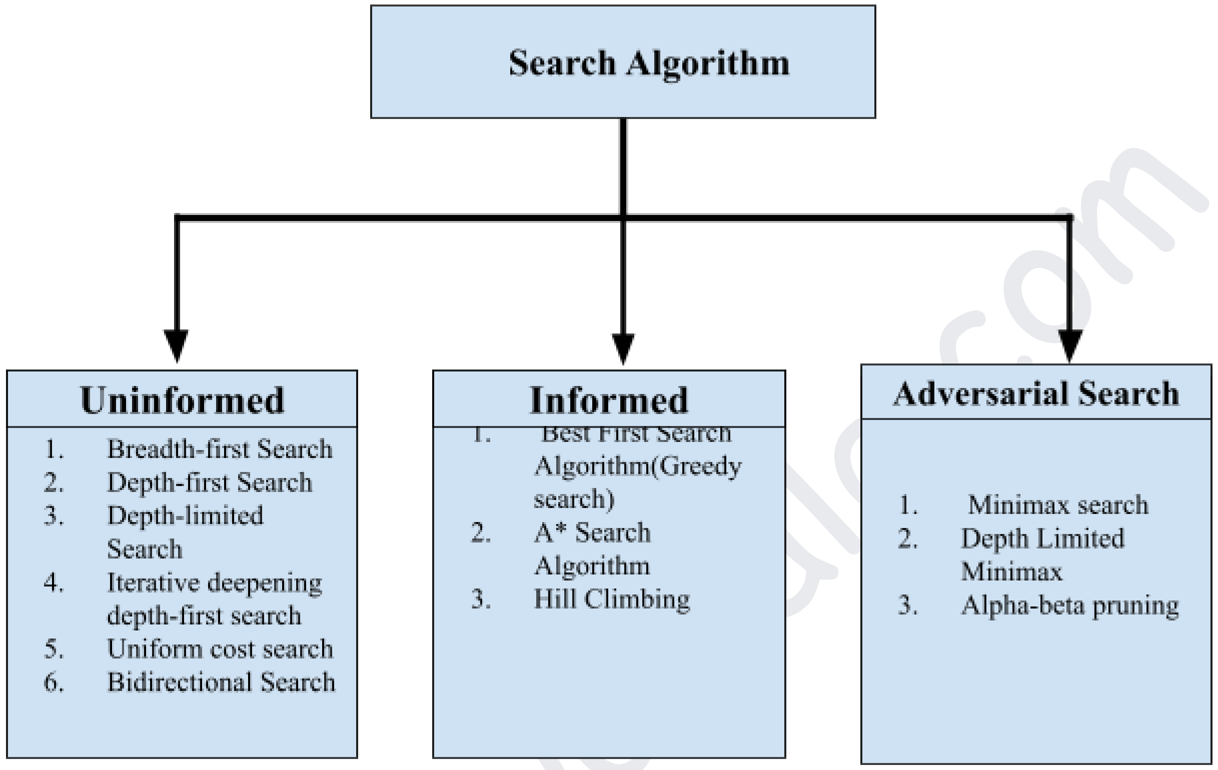
   Else,

   ```
   * Expand the node (find all the new nodes that could be reached from this node), and add resulting nodes to the frontier.
   * Add the current node to the explored set.
   ```


GATE in Data Science & Artificial Intelligence
GATE DA 2024
TEST SERIES
By:
Piyush Wairale
MTech (IIT Madras)
Instructor at IIT Madras BS in Data Science Degree

Connect with me:
Youtube: Piyush Wairale IITM
Linkedin: Piyush Wairale
Facebook: Piyush Wairale
Instagram: Piyush Wairale

In order to deal with various types of search problems, we have search algorithms, let us discuss them one by one.



Measuring problem-solving performance:

Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them. We can evaluate an algorithm's performance in four ways:
- **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
- **Optimality:** Does the strategy find the optimal solution?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

1.Uninformed Search:

Uninformed search algorithms are fundamental tools in the field of Artificial Intelligence (AI) used to solve problems where the solution is not known in advance. These algorithms are also known as **blind search algorithms** because they operate without any specific knowledge about the problem domain.
The term means that the strategies have no additional information about states beyond that provided in the problem definition.
All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the order in which nodes are expanded.

Following are the various types of uninformed search algorithms:

1. **Breadth-first Search**
2. **Depth-first Search**
3. **Depth-limited Search**
4. **Iterative deepening depth-first search**
5. **Uniform cost search**
6. **Bidirectional Search**

1.1 Breadth-first search:

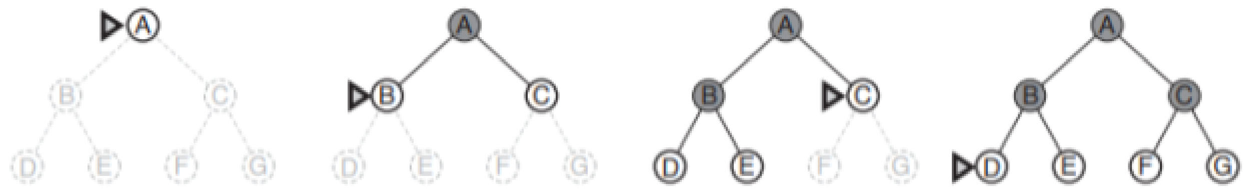
- Breadth-first search is a complete search algorithm, guaranteeing that it will find a solution if one exists by exploring all nodes at each level of the search tree.
- It ensures optimality in terms of finding the shallowest goal node, but it may not always find the optimal solution in terms of the least-cost path if edge costs are not uniform.
- The algorithm generates all shallower nodes before deeper ones, making it inefficient in terms of time and memory when the search tree is deep.
- The use of a **FIFO queue** for the frontier ensures that new nodes are always deeper than their parents, maintaining the breadth-first exploration strategy.
- Breadth-first search applies the goal test when a node is generated, ensuring that any found goal node is indeed the shallowest one, eliminating the need to compare multiple shallow goal nodes.

Algorithm of Breadth-first search:

1. Start with an initial node containing the starting state of the problem and a path cost of 0.
2. Check if the initial state is a goal state. If it is, you're done, and you have a solution.
3. Create a queue (think of it as a line) called the "frontier" and add the initial node to it.
4. Create an empty set called "explored" to keep track of the states we've already seen.
5. Now, repeat the following steps:
   a. If the frontier is empty (there's no one left in the line), and you haven't found a solution, then you've failed to find a solution. So, return failure.
   b. Take the node at the front of the line (the shallowest one).
   c. Add the state of this node to the explored set, so we remember we've seen it.
   d. For each possible action you can take from the current state:
   i. Create a new node that represents the result of taking that action.
6. Keep repeating these steps until you either find a solution or determine that there is no solution (if the frontier becomes empty).

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

Breadth-first search on a graph
For example:-



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

One more example - here one maze ,

S 0 0 0

0 1 1 0

0 1 0 G

0 0 0 0

In this maze:

- S: Starting point
- G: Goal
- 0: An open path
- 1: An obstacle (you can't pass through it)

Now, let's apply BFS step by step:

1. Start with the initial state at S and initialize the frontier with S.
   -Frontier: [S]
   -Explored: []
2. Take the first node from the frontier (S), mark it as explored, and check its neighboring nodes.
   -Frontier: []
   -Explored: [S]
   Neighbors of S:
   a. Move right (to 0,1)
   b. Move down (to 1,0)
3. Add the neighboring nodes to the frontier.
   Frontier: [0,1, 1,0]
   Explored: [S]
4. Take the next node from the frontier (0,1) and check its neighbors.
   Frontier: [1,0]

Explored: [S, 0,1]
Neighbors of (0,1):
  a. Move up (to S)
  b. Move right (to 0,2)

5. Add the neighboring nodes to the frontier.
   Frontier: [1,0, 0,2]
   Explored: [S, 0,1]
6. Continue this process, always selecting the shallowest node from the frontier, until you reach the goal node (G).
   a. Move down (to 1,1)
   b. Move right (to 0,3)
   c. Move down (to 2,2)
   d. Move right (to 1,3)
   e. Move down (to G)

   Frontier: [1,0, 0,2, 1,1, 0,3, 2,2, 1,3, G]
   Explored: [S, 0,1, 1,0, 0,2, 1,1, 0,3, 2,2]
7. Once you've reached the goal node (G), you have found a solution.
   The path from S to G is: S -> (1,0) -> (0,2) -> (1,1) -> (0,3) -> (2,2) -> (1,3) -> G.

BFS guarantees that you find the shortest path in terms of the number of steps, and it systematically explores all possible paths before returning a solution.

Performance:

Completeness: A search algorithm is complete if it guarantees finding a solution if one exists.

Optimality: An optimal search algorithm finds the shortest path to the goal.

Time Complexity:

- In the worst case, BFS also has a time complexity of O(b^d)(Exponential).
- However, BFS systematically explores all nodes at each depth level before moving to the next level.
- It is particularly efficient in finding the shortest path in an unweighted graph because it guarantees that the shallowest goal node is reached first.

Space Complexity:

- The space complexity of BFS is typically O(b^d) (Exponential) because it stores all nodes at each depth level in a queue.
- BFS can consume significant memory, especially when the branching factor is high or the tree is deep.
- It guarantees that it explores the shallowest nodes first, but this comes at the cost of increased memory usage.

Where b = branching factor (require finite b) d = depth of shallowest solution

When to Use Uninformed Search:

- Uninformed search algorithms are suitable for small to moderately sized search spaces where domain-specific knowledge is limited or unavailable.
- They are a good choice when you need a simple, general-purpose approach to problem-solving.

Limitations:

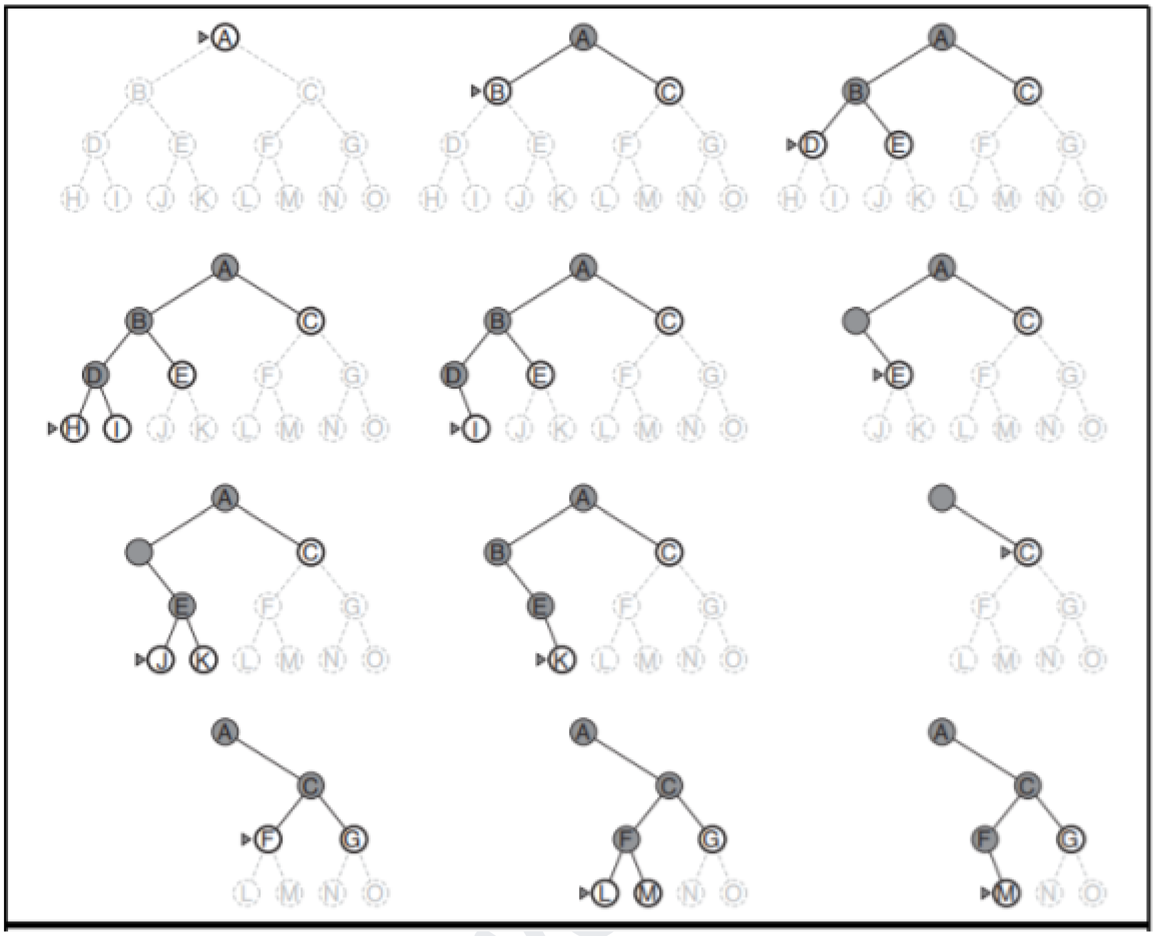
- Uninformed search algorithms may be inefficient for large search spaces.
- They do not take advantage of domain-specific information, which can lead to unnecessary exploration.

1.2 Depth-First Search:

- Depth-First Search (DFS) is a complete search algorithm that explores as far as possible along a branch of the search tree before backtracking.
- It may not guarantee optimality in terms of finding the shortest path, especially when edge costs are not uniform.
- DFS tends to generate a deeper search tree before exploring shallower nodes, which can be more memory-efficient than BFS for deep search trees.
- It uses a **Last-In-First-Out (LIFO) stack** for managing the frontier, which means it explores one branch entirely before moving on to the next.
- The goal test is applied when a node is selected for expansion, which may lead to multiple solutions.

Algorithm of Depth-First Search:

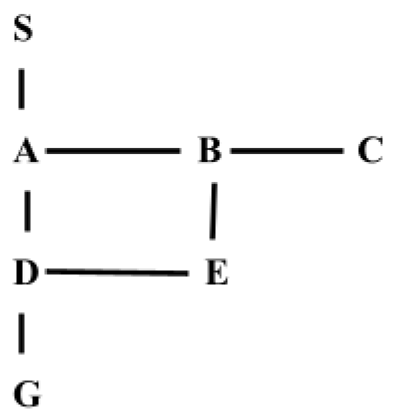
1. Start with an initial node containing the starting state of the problem and a path cost of 0.
2. Check if the initial state is a goal state. If it is, you've found a solution.
3. Create a stack (think of it as a stack of plates) called the "frontier" and add the initial node to it.
4. Create an empty set called "explored" to keep track of the states we've already seen.
5. Now, repeat the following steps:
    a. If the frontier is empty, return failure.
    b. Take the node at the top of the stack (the deepest one).
    c. Add the state of this node to the explored set.
    d. For each possible action you can take from the current state:
        i. Create a new node that represents the result of taking that action.
    e. Add the neighboring nodes to the stack.
6. Keep repeating these steps until you either find a solution or determine that there is no solution (if the frontier becomes empty).



Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only

Depth-First Search on a Graph:

For example-



Starting from S, DFS explores as far down a branch as possible before backtracking:

- S -> A -> D -> G
- S -> A -> E
- S -> B -> C

Performance:

Completeness: Yes (if the search tree is finite).
Optimality: No (it may not find the shortest path).
Time Complexity

- In the worst case, DFS can have a time complexity of O(b^d) (Exponential)
- If the search tree is deep and has a high branching factor, DFS can take a long time to find a solution.
- DFS can perform relatively well when the solution is shallow or located early in the search space (small d).

Space Complexity:

- The space complexity of DFS is typically O(bd) (Linear), which accounts for the maximum depth of the search tree.
- It uses memory proportional to the depth of the tree because it needs to store all nodes along the current branch in the stack.
- DFS can be memory-efficient when the depth is limited, but it may run into memory issues with deep search trees.

Advantages of Depth-First Search:

1. **Memory Efficiency**: DFS can be more memory-efficient than BFS for deep search trees because it explores deeply before moving to the next branch.
2. **Simplicity:** Like BFS, DFS is conceptually simple and easy to implement.
3. **Solves Uninformed Problems**: It's suitable for solving uninformed search problems where the goal is to find any valid solution.

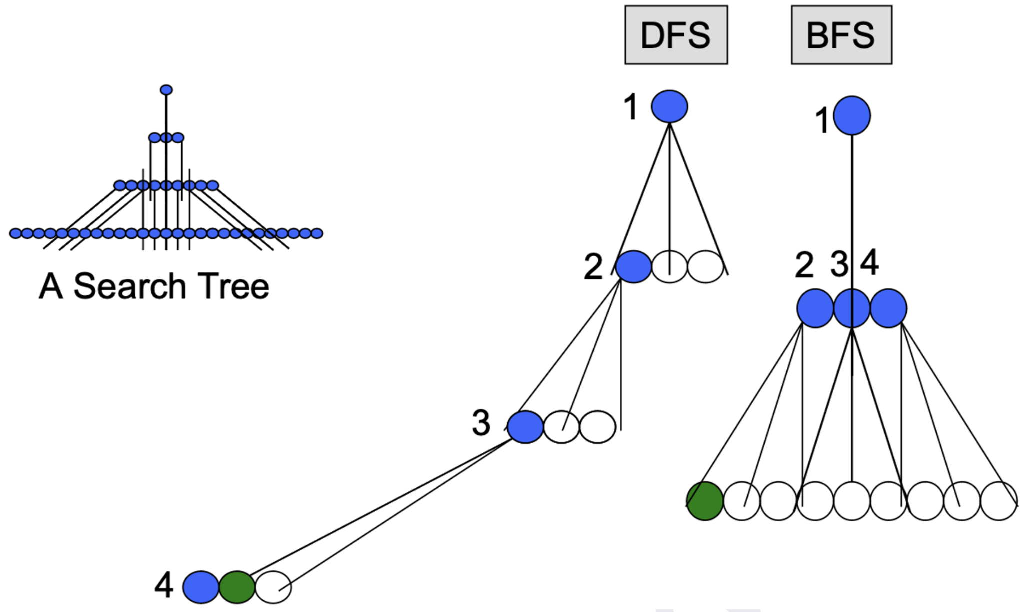
Disadvantages of Depth-First Search:

1. **Completeness and Optimality:** It may not find a solution in infinite or cyclical graphs and doesn't guarantee optimality.
2. **Lack of Guidance:** DFS doesn't use heuristic information, which can make it less efficient than informed search algorithms in some cases.
3. **Potential Infinite Loops**: In infinite graphs or graphs with cycles, DFS may get stuck in infinite loops without reaching a solution.
4. **Not Suitable for Shortest Path**: It doesn't prioritize finding the shortest path, making it unsuitable for certain problems.

.

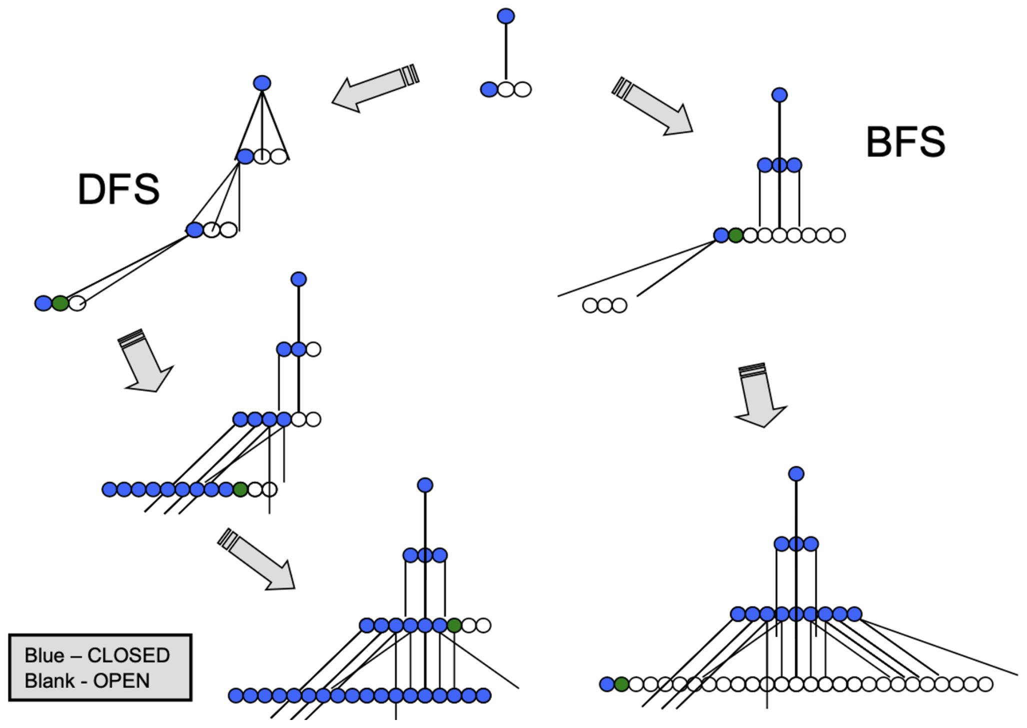
Analysis of BFS and DFS

Behaviour:



When the two searches pick the fifth node, coloured green, they have explored different parts of the search tree, expanding the four nodes before in the order shown. (Discussed in Lectures)

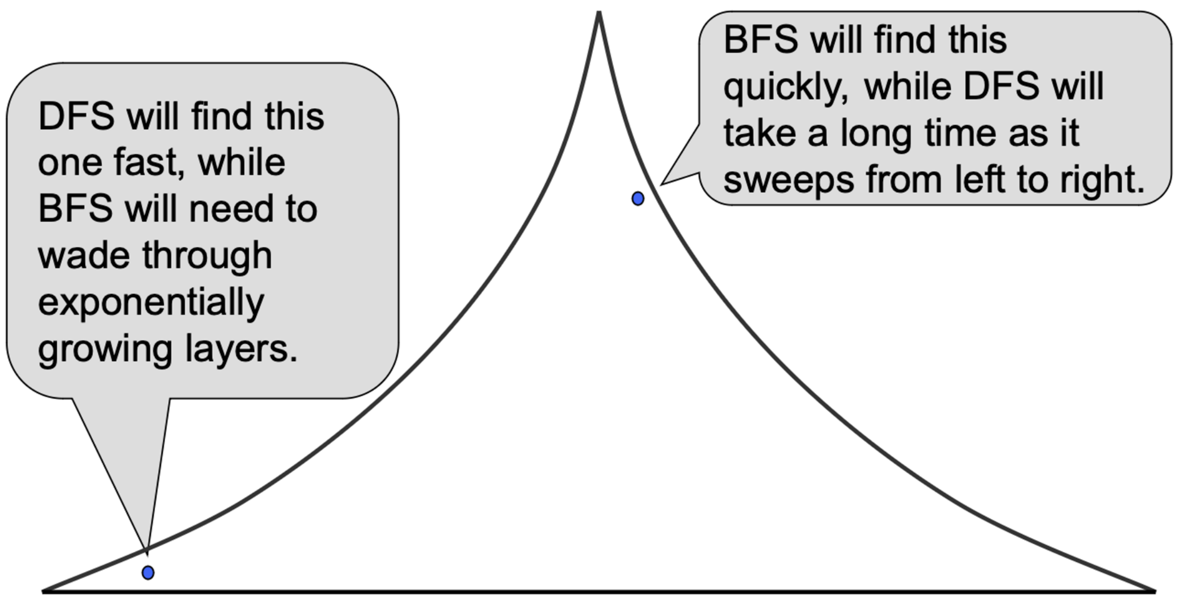
Space:



Completeness:

1. **When Search space is finite:**
    - Both DFS and BFS are complete for finite state spaces.
    - Both are systematic
    - They will explore the entire search space before reporting the failure. This is because the termination criteria for both is same.
2. **When Search space is infinite:**
    - If the state space is infinite, but with finite branching then DFS may go down to an infinite path and not terminate.
    - BFS will find a solution if there exists one.
    - If there is no solution , both algorithm will not terminate for infinite spaces

Running Time


DFS will find this one fast, while BFS will need to wade through exponentially growing layers.
BFS will find this quickly, while DFS will take a long time as it sweeps from left to right.

Time & Space Complexity:

- Assume constant branching fact *b*
- Assume that goal occurs somewhere at depth *d*
- Let N_{DFS} be the number of nodes inspected by DFS
- Let N_{BFS} be the number of nodes inspected by BFS

If the goal is on the extreme left then *DFS* finds it after examining *d* nodes. These are the ancestors of the goal node. If the goal is on the extreme right, it has to examine the entire search tree, which is b^d nodes. Thus, on the average, it will examine N_{DFS} nodes given by,

$$N_{DFS} = \frac{(d+1) + (b^{d+1}-1)/(b-1)}{2} \approx \frac{b^d}{2}$$

The search arrives at level *d* after examining the entire subtree above it. If the goal is on the left, it picks only one node, which is the goal. If the goal is on the right, it picks it up in the end (like the *DFS),* that is, it picks b^d nodes at the level *d*. On an average, the number of nodes N_{BFS} examined by *BFS* is

$$N_{BFS} = \frac{(b^d-1)/(b-1) + 1 + (b^{d+1}-1)/(b-1)}{2} \approx \frac{b^d(b+1)}{2(b-1)}$$

By comparing, we can say:

$$N_{BFS} \approx N_{DFS} \frac{(b+1)}{(b-1)}$$

From above, we can say that the problems with large branching factor (b) both will tend to do the same work.

Summary:

- BFS is effective when search tree has low branching factor
- BFS can work even in trees that are infinitely deep.
- BFS requires a lot of memory as number of nodes in level of the tree increases exponentially
- BFS is superior when the GOAL exists in the upper right portion of a search tree.
- BFS gives the optimal solution.
- DFS is effective when there are few sub trees in the search tree that have only one connection to the rest of the states.
- DFS is best when the GOAL exists in the lower left portion of the search tree.
- DFS can be dangerous when the path closer to the START and farther from the GOAL has been chosen.

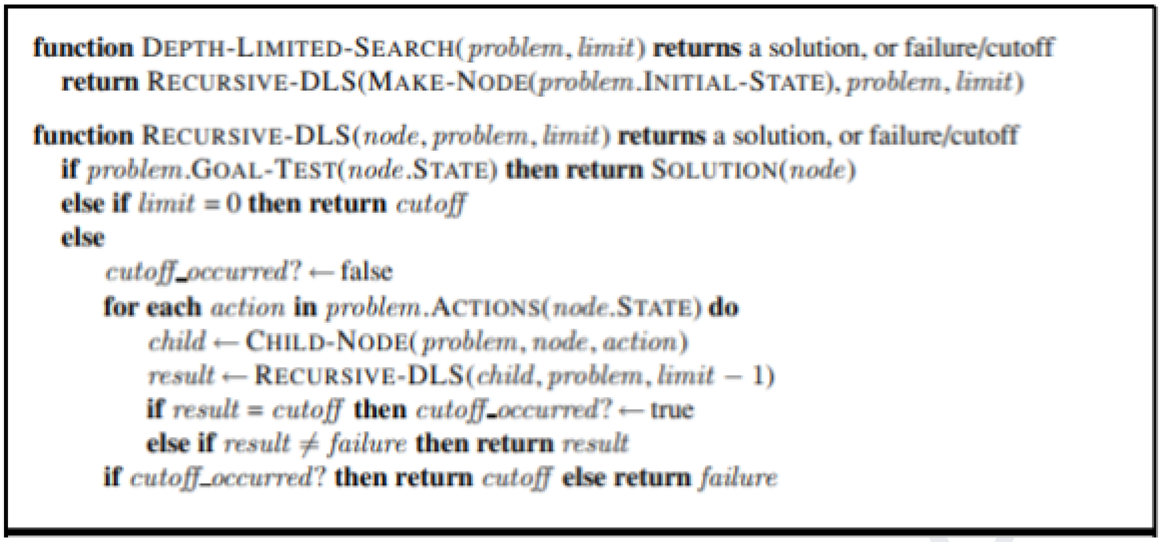
- DFS is memory efficient as the path from start to current node is stored. Each node should contain a state and its parent.
- DFS may not give an optimal solution.

1.3.Depth-limited Search or Depth Bounded DFS:

1. DLS is a modification of DFS that restricts the depth of exploration.
2. It uses a depth limit parameter to determine how deep into the tree or graph it can explore
3. Depth-limited search alleviates the infinite-state space problem by setting a predetermined depth limit.
4. It solves the infinite-path problem but can be incomplete if the shallowest goal is beyond the limit.
5. Depth-limited search becomes nonoptimal if the depth limit is greater than the depth to the goal state.
6. Its time complexity is $O(b^d)$, and its space complexity is O(bd), where b is the branching factor and d is the depth limit.
7. Depth-first search is a special case of depth-limited search with an infinite depth limit (= ∞).

Algorithm of Depth-Limited Search:

- Initialize the search with the root node.
- Perform a depth-first traversal with the depth limited to a predefined level.
- When the depth limit is reached or a goal state is found, stop the search.
- If the goal state is not found, backtrack to the previous node and continue the search from there.
- Repeat steps 2-4 until the goal is reached or the entire search space is explored.

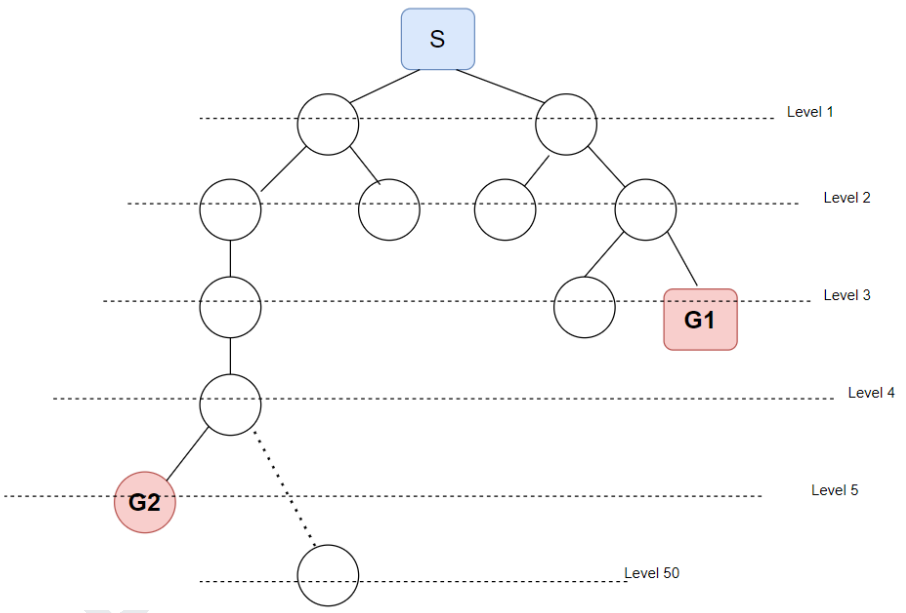


```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

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

A recursive implementation of depth-limited tree search

Consider the below infinite search space, S is start state and G is the Goal State.



Case 1: When l > d

In this case, the depth of the goal node G1 is 3 (d = 3) and say we put a predefined search limit of level 5 (l = 5). Then in this case our algorithm will reach the required goal state, so completeness can be achieved. But the solution may or may not be optimal.

Case 2: When d > l

In this case, the depth of the goal node G2 is 3 (d = 5) and say we put a predefined search limit of level 4 (l = 4). Then in this case our algorithm will never reach the goal state, thus the algorithm will be incomplete.

For example:

Suppose we have a simplified navigation map with cities and roads. We want to find a path from the "Start" city to the "Goal" city, but we don't want to explore paths that are too long. We will use DLS to limit the depth of exploration.

```
        A
       / \
      B   C
     / \   \
    D   E   F
   / \    / \
  G   H   I   J
 / \       / \
K   L     M   N
           \
            Goal
```

- Start: A
- Goal: Goal
- Depth Limit: 3 (We want to limit the depth of exploration to 3 levels)

We start at city A and use DLS to explore the map while limiting the depth to 3 levels.

- Start at A (Depth 0).
- Explore neighbors of A: B and C (Depth 1).
- Explore neighbors of B: D and E (Depth 2).
- Explore neighbors of C: F (Depth 2).
- Explore neighbors of D: G and H (Depth 3).
- Explore neighbors of E: (No neighbors at Depth 3).

- Explore neighbors of F: I and J (Depth 3).

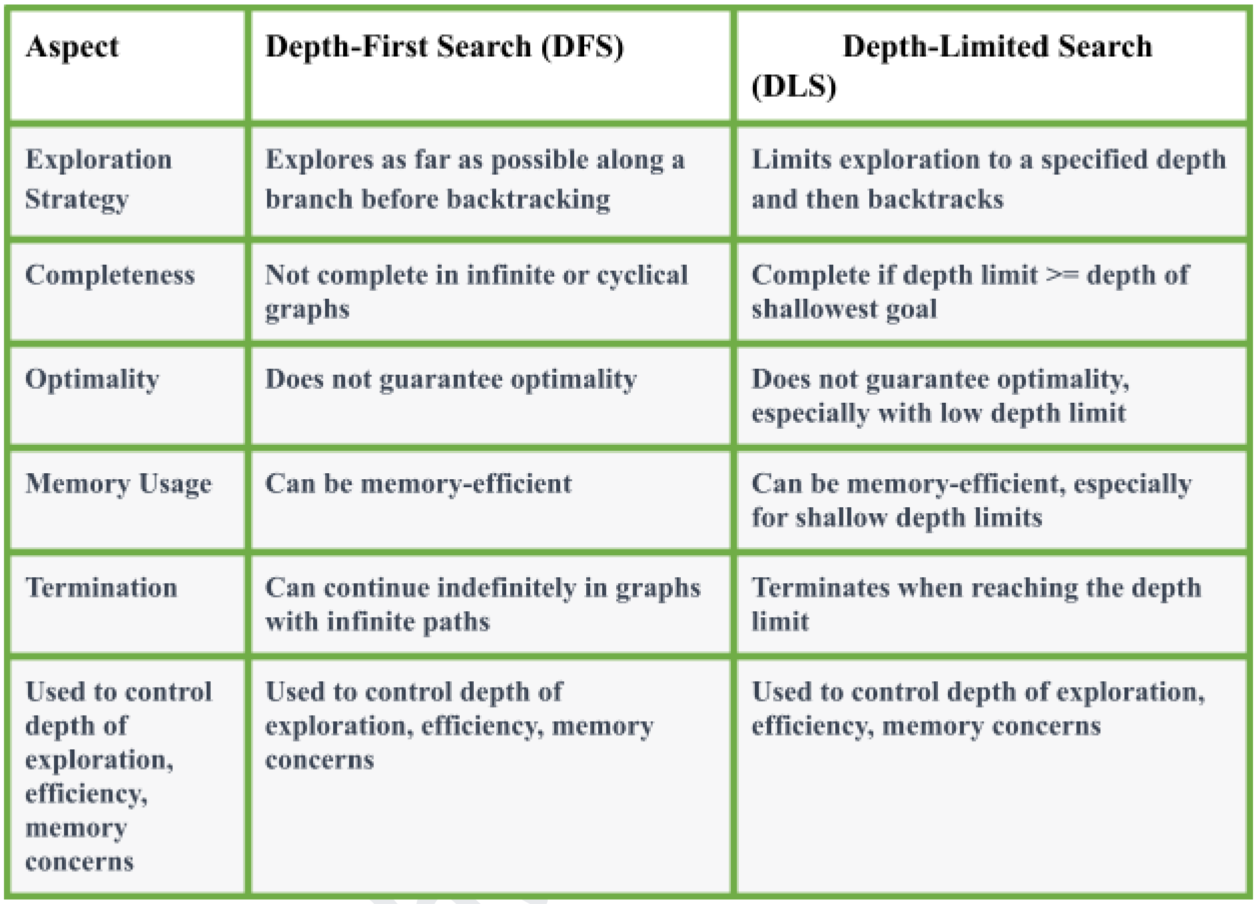
Now, we have reached the depth limit of 3. Since the "Goal" city is not within this limit, we stop the exploration.
The path from "Start" to "Goal" within the depth limit is: A -> C -> F -> J -> Goal.

In this example, Depth-Limited Search (DLS) limits the exploration to a certain depth level (in this case, 3 levels) to efficiently find a solution within the specified depth constraint. If the goal is not found within the depth limit, DLS stops the search.

Performance of DFS and DLS:

- Completeness: DFS is not complete in infinite or cyclical graphs, while DLS is not complete if the depth limit is insufficient.
- Optimality: Neither DFS nor DLS guarantee optimality.
- Time complexity: Exponential: O(b^m) in the worst case (DFS), and linear in terms of depth limit for DLS.
- Space complexity: Linear in the maximum depth for both DFS and DLS.

Difference between DFS and DLS:



Aspect	Depth-First Search (DFS)	Depth-Limited Search (DLS)
Exploration Strategy	Explores as far as possible along a branch before backtracking	Limits exploration to a specified depth and then backtracks
Completeness	Not complete in infinite or cyclical graphs	Complete if depth limit >= depth of shallowest goal
Optimality	Does not guarantee optimality	Does not guarantee optimality, especially with low depth limit
Memory Usage	Can be memory-efficient	Can be memory-efficient, especially for shallow depth limits
Termination	Can continue indefinitely in graphs with infinite paths	Terminates when reaching the depth limit
Used to control depth of exploration, efficiency, memory concerns	Used to control depth of exploration, efficiency, memory concerns	Used to control depth of exploration, efficiency, memory concerns

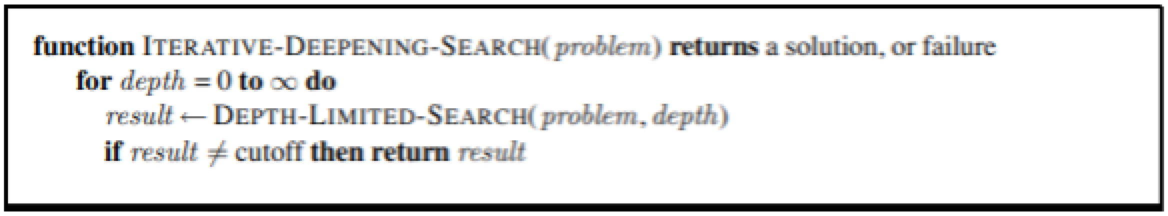
1.4 Iterative deepening depth-first search:

1. IDS gradually increases the depth limit to find the best depth for searching until a goal is found.
2. It combines benefits of depth-first and breadth-first search, offering modest memory requirements and completeness.
3. IDS generates nodes at different levels multiple times but remains reasonably efficient.
4. In terms of node generation, it is asymptotically similar to breadth-first search (BFS).
5. IDS is preferred for large search spaces when the depth of the solution is unknown.

Iterative Deepening Depth-First Search, often abbreviated as IDDFS, is a search strategy that combines the benefits of depth-first search and iterative deepening. It is particularly useful when the depth of the solution is unknown.

Algorithm of Iterative Deepening Depth-First Search:

1. Start with a depth limit of 0.
2. Apply Depth-Limited Search (DLS) with the current depth limit.
3. If DLS finds a solution, return it.
4. If DLS returns "cutoff," increment the depth limit and repeat from step 2.
5. If DLS returns failure and all depth limits have been exhausted, return failure.



```
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
```

The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns failure, meaning that no solution exists.

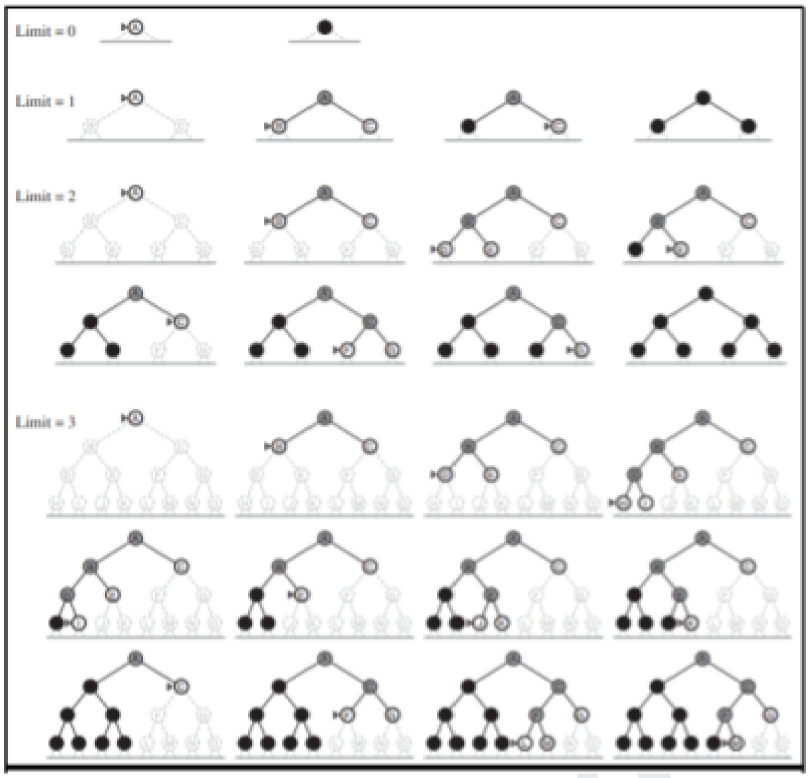
For Example-

```
     A
    / \
   B   C
  / \   \
 D   E   F
/ \     / \
G  H    I  J
```

Algorithm Execution:

1. Start with a depth limit of 0.
2. Apply Depth-Limited Search (DLS) with the current depth limit:
    a. At depth limit 0, we explore only node A.
    b. Since node A is not the goal, we increment the depth limit to 1.
3. Apply DLS with depth limit 1:
    a. At depth limit 1, we explore nodes A, B, and C.
    b. Node C is not the goal, so we continue.
    c. We increment the depth limit to 2.
4. Apply DLS with depth limit 2:
    a. At depth limit 2, we explore nodes A, B, C, D, E, and F.
    b. Node F is not the goal, so we continue.
    c. We increment the depth limit to 3.
5. Apply DLS with depth limit 3:
    a. At depth limit 3, we explore nodes A, B, C, D, E, F, G, H, I, and J.
    b. Node I is found at this depth, so we return it as the solution.

IDDFS successfully finds node I in this example. It starts with a shallow depth limit and gradually increases it until it reaches the depth where the goal node is located. This strategy allows IDDFS to control memory usage while systematically searching deeper levels of the tree.



Four iterations of iterative deepening search on a binary tree.

Performance:

- **Completeness:** IDDFS is complete as long as the depth limit reaches or exceeds the depth of the shallowest goal node.
- **Optimality:** Like DFS, IDDFS does not guarantee optimality; it may find a solution, but it might not be the optimal one.
- **Time Complexity:** The time complexity is O(b^d), where b is the branching factor and d is the depth of the shallowest goal.
- **Space Complexity:** The space complexity is O(b*d), where b is the branching factor and d is the depth of the shallowest goal.

Advantages of IDDFS:

1. **Completeness:** IDDFS guarantees finding a solution if one exists.
2. **Memory Efficiency:** It has relatively modest memory requirements compared to some other search algorithms.
3. **Simplicity:** Conceptually easy to understand and implement.
4. **Versatility:** Suitable for problems with unknown solution depths.

Disadvantages of IDDFS:

1. **Lack of Optimality:** IDDFS does not guarantee finding the optimal solution.
2. **Repetitive Node Generation:** Nod
3. es at different levels may be generated multiple times.
4. **Inefficiency for Large Branching:** Becomes inefficient with high branching factors.
5. **Limited Heuristic Use:** Like DFS, it doesn't incorporate heuristic information for efficient search.

1.5 Uniform cost search:

Uniform Cost Search, often abbreviated as UCS, is a search algorithm that aims to find the optimal solution by exploring the lowest-cost paths in a weighted graph.

Algorithm of Uniform Cost Search:

1. Start with an initial node containing the starting state of the problem and a path cost of 0.
2. Create a priority queue called the "frontier" and add the initial node to it with a priority of 0.
3. Create an empty set called "explored" to keep track of the states we've already seen.
4. Repeat the following steps:
   a. If the frontier is empty (there's no one left in the queue), and you haven't found a solution, then you've failed to find a solution. Return failure.
   b. Take the node with the lowest path cost from the frontier.
   c. If the state of this node is a goal state, you've found a solution. Return it.
   d. Add the state of this node to the explored set, so we remember we've seen it.
   e. For each possible action you can take from the current state:
   i. Create a new node that represents the result of taking that action.
   ii. If the new state is not in the explored set, calculate its path cost and add it to the frontier with the calculated cost as its priority.

Example:

Suppose you want to find the cheapest route from City A to City G on a road network with different road segments having varying costs. The road network can be represented as a weighted graph, where the cities are nodes, and the roads between them are edges with associated costs.

Here's a simplified representation of the road network:

```
       A
      / \
     1   5
    /     \
   B----C
  / \    / \
 2   3 1   4
/     \/    \
D----E-----F
|      |      |
3      2      6
|      |      |
G----H-----I
```

- A, B, C, D, E, F, G, H, and I are cities.
- The numbers on the edges represent the cost of traveling between the cities.

UCS Execution:

1. Start at City A with a path cost of 0.
2. Create a priority queue (frontier) and add the initial node (A) with a priority of 0 to the frontier.
3. Initialize an empty set (explored) to keep track of visited cities.
4. Perform the following steps: a. Take the node with the lowest path cost from the frontier (initially, A with cost 0). b. Check if the current city is the goal (G). If yes, you've found a solution. c. If not, mark the current city as explored and expand it:
    a. From A, you can go to B (cost 1) or C (cost 5). Add B and C to the frontier.
    b. Explored: [A], Frontier: [B(1), C(5)] d. Select the node with the lowest cost from the frontier (B with cost 1).

c. Expand B to D (cost 2) and E (cost 3). Add them to the frontier.
d. Explored: [A, B], Frontier: [C(5), D(3), E(4)] e. Continue this process, always selecting the lowest-cost node from the frontier:
e. Expand D to G (cost 3). Add G to the frontier.
f. Expand E to H (cost 6). Add H to the frontier.
g. Expand C to F (cost 4). Add F to the frontier. f. Select the lowest-cost node (D with cost 3) from the frontier:
h. Expand G to H (cost 5) and I (cost 8). Add H to the frontier. g. Continue exploring:
i. Expand H to I (cost 7).
j. Expand F to I (cost 10). h. Select the lowest-cost node (G with cost 3) from the frontier:
k. You've reached the goal (City G) with a cost of 3.

Solution:
The cheapest route from City A to City G is through the following cities with the associated costs: A -> B -> D -> G, with a total cost of 3.

Uniform Cost Search (UCS) systematically explores paths with increasing costs and finds the optimal solution in terms of the lowest cost. In this example, it successfully determined the most cost-effective route.

Performance:

- **Completeness:** UCS is complete if the edge costs are non-negative. It will find the optimal solution.
- **Optimality:** UCS guarantees finding the optimal solution by exploring the lowest-cost paths first.
- **Time Complexity:** The time complexity depends on the graph structure but is generally O(b^d), where b is the branching factor and d is the depth of the optimal solution.
- **Space Complexity:** The space complexity depends on the number of nodes in the frontier, which can be high in some cases.

Advantages of UCS:

1. **Optimality:** UCS is guaranteed to find the optimal solution with non-negative edge costs.
2. **Precision:** It considers the exact cost of each path, making it suitable for cost-sensitive problems.
3. **Versatility**: UCS can handle a wide range of problem domains with weighted graphs.
4. **Minimal Memory Usage:** Uses memory efficiently compared to some other search algorithms.

Disadvantages of UCS:

1. **Inefficient for Large Costs:** If edge costs are very high, UCS can be slow and memory-intensive.
2. **Not Suitable for Negative Costs**: UCS cannot handle graphs with negative edge costs.
3. **Complexity:** The time complexity can be high in graphs with many branches.

1.6 Bidirectional search:

1. Bidirectional search runs two simultaneous searches from the initial state and the goal state, aiming to meet in the middle, which can significantly reduce the search space.
2. The goal test in bidirectional search checks whether the frontiers of the two searches intersect, and if they do, a potential solution is found.
3. The first solution encountered in bidirectional search may not always be optimal, requiring additional search to confirm.
4. In the worst case, bidirectional search generates a total of $b^{d/2}$ nodes when both directions use breadth-first search, reducing time complexity compared to unidirectional breadth-first search.
5. The space complexity of bidirectional search is O($b^{d/2}$), with the need to keep at least one of the frontiers in memory for intersection checking, which is a notable limitation.

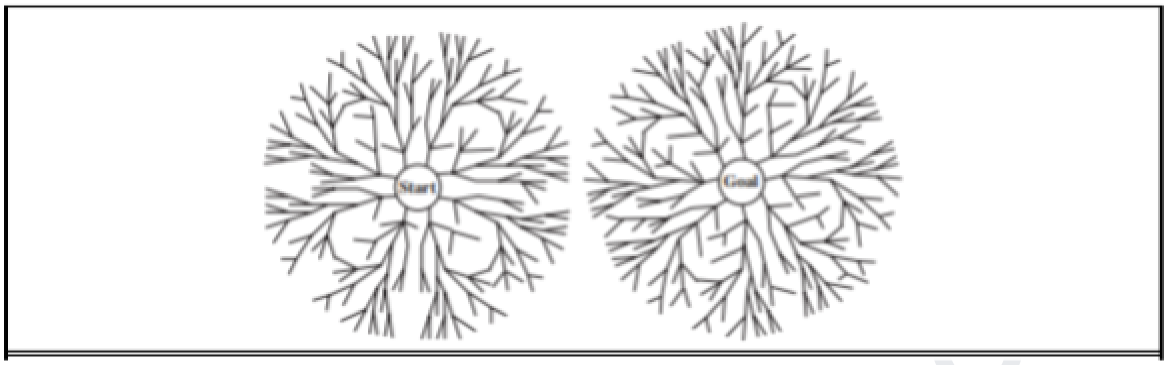
Bidirectional Search is an advanced graph search algorithm that operates by simultaneously exploring the search space from both the start and goal states. It aims to meet in the middle, where the two search frontiers intersect. This approach is particularly useful when you have information about both the start and goal states and want to reduce the search space.

Algorithm:

1. Start with two search frontiers: one from the start state and one from the goal state.
2. Initialize two sets: one for explored nodes in the forward direction (from start) and one for explored nodes in the backward direction (from the goal).
3. While neither of the frontiers is empty:
   a. Expand the frontier in the forward direction by selecting a node to explore.
   b. Check if this node is in the set of explored nodes from the backward direction. If yes, you've found a meeting point, and the solution can be constructed.
   c. If not, add the node to the set of explored nodes from the forward direction and generate its child nodes.
   d. Expand the frontier in the backward direction by selecting a node to explore.
   e. Check if this node is in the set of explored nodes from the forward direction. If yes, you've found a meeting point, and the solution can be constructed.

f. If not, add the node to the set of explored nodes from the backward direction and generate its parent nodes.

4. If the two frontiers do not meet, there is no solution.



A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

Example:

Consider a simple bidirectional search example on a graph with cities:

```
 A -- B
 / \   |
C D  E
 \ | /
   F
```

- Start state: A
- Goal state: F

Execution:

1. Initialize two frontiers: one from A (start) and one from F (goal).
2. Initialize explored sets for both directions.
3. Expand from both directions simultaneously:
    a. Forward: A -> B -> E
    b. Backward: F -> E
4. Meeting point: E is explored from both directions.
5. Construct the solution: A -> B -> E -> F.

Bidirectional Search efficiently found the path from A to F, exploring from both directions and meeting at E.

Bidirectional Search is a powerful technique for reducing search space in scenarios where you have information about both the start and goal states. It can significantly improve search efficiency, especially in large graphs.

Performance of Bidirectional Search:

1. **Reduced Time Complexity:** Bidirectional search significantly reduces the time complexity compared to traditional search algorithms like breadth-first search. It generates only $b^{d/2}$ nodes in the worst case, leading to faster search times when the frontiers meet.
2. **Space Efficiency:** The space complexity of bidirectional search is O($b^{d/2}$), which is considerably lower than the O(bd) space complexity of standard breadth-first search. This makes it more memory-efficient, especially for large search spaces.
3. **Optimality Trade-off:** While bidirectional search can find a potential solution faster, the first solution encountered may not be optimal. Additional exploration is often required to confirm optimality.
4. **Effective for Unknown Depths:** Bidirectional search is well-suited for situations where the depth of the solution is unknown. It combines the benefits of breadth-first search with reduced memory requirements.
5. **Limited Memory:** Despite its efficiency, bidirectional search still requires maintaining at least one frontier in memory for intersection checking. In cases with severe memory constraints, this can be a limitation.

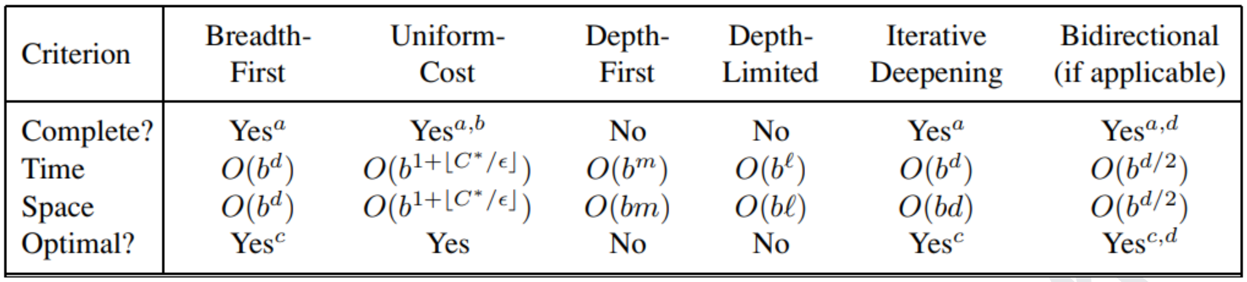
Advantages:

- Can be much faster than traditional search algorithms, especially in large search spaces.
- Reduces the search space by simultaneously exploring from both ends.
- Guarantees optimality if the underlying search algorithm is optimal (e.g., breadth-first search).

Disadvantages:

- Requires bidirectional information (knowledge of both the start and goal states).
- Complexity and overhead of managing two frontiers and explored sets.
- May not always be more efficient, depending on the specific problem and graph structure.

❖ **Comparing uninformed search strategies:**



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

Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

References:

1. FIRST **COURSE** IN **ARTIFICIAL INTELLIGENCE**. **Deepak** Khemani.
2. Artificial Intelligence: A Modern Approach Textbook by Peter Norvig and Stuart J. Russell
3. Artificial Intelligence Book by Saroj Kaushik
4. IIT Madras BS in Data Science Degree Lectures