National University of Computer & Emerging Sciences, Karachi
Computer Science Department
Spring 2022, Lab Manual – 05

Course Code: AI-2002	Course : Artificial Intelligence Lab
Instructor(s):	Kariz Kamal, Erum Shaheen, Mafaza, Danish Waseem, Ali Fatmi

Contents:

Objective

1. Introduction to Problem Solving by Searching
2. Implementing Uninformed/Blind Search Algorithms in Python

Problem-solving agent

In Artificial Intelligence, Search techniques are universal problem-solving methods. Rational agents or Problem-solving agents in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result.

Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

Steps performed by Problem-solving agent

1. **Goal Formulation:** It is the first and simplest step in problem-solving. It organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to achieve that goal. Goal formulation is based on the current situation and the agent's performance measure (discussed below).
2. **Problem Formulation:** It is the most important step of problem-solving which decides what actions should be taken to achieve the formulated goal. There are following five components involved in problem formulation:
    1. **Initial State:** It is the starting state or initial step of the agent towards its goal.
    2. **Actions:** It is the description of the possible actions available to the agent.
    3. **Transition Model:** It describes what each action does.
    4. **Goal Test:** It determines if the given state is a goal state.
    5. **Path cost:** It assigns a numeric cost to each path that follows the goal. The problem-solving agent selects a cost function, which reflects its performance measure. Remember, an optimal solution has the lowest path cost among all the solutions.

Note: Initial state, actions, and transition model together define the **state-space** of the problem implicitly.

State-space of a problem is a set of all states which can be reached from the initial state followed by any sequence of actions. The state-space forms a directed map or graph where nodes are the states, links between the nodes are actions, and the path is a sequence of states connected by the sequence of actions.

1. **Search:** It identifies all the best possible sequences of actions to reach the goal state from the current state. It takes a problem as an input and returns a solution as its output.
2. **Solution:** It finds the best algorithm out of various algorithms, which may be proven as the best optimal solution.
3. **Execution:** It executes the best optimal solution from the searching algorithms to reach the goal state from the current state.

Example Problems

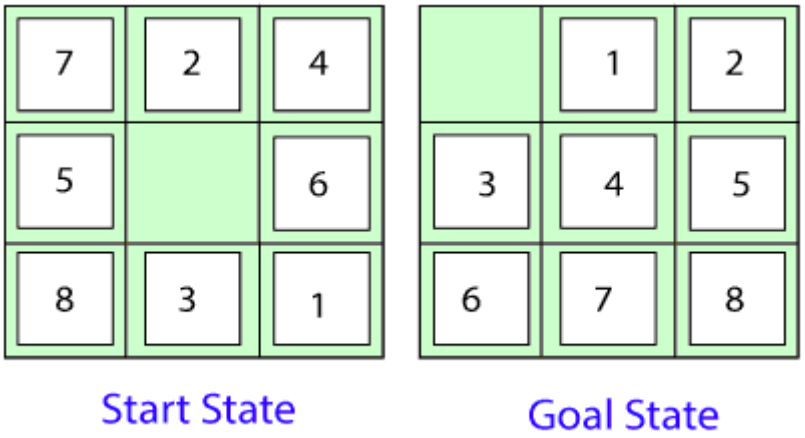
Basically, there are two types of problem approaches:

1. **Toy Problem:** It is a concise and exact description of the problem which is used by the researchers to compare the performance of algorithms.
2. **Real-world Problem:** It is real-world based problems which require solutions. Unlike a toy problem, it does not depend on descriptions, but we can have a general formulation of the problem.

Some Toy Problems:

1. 8 Puzzle Problem:

Here, we have a **3×3 matrix** with movable **tiles numbered from 1 to 8 with a blank space**. The tile adjacent to the blank space can slide into that space.The objective is to reach a specified goal state similar to the goal state, as shown in the below figure.In the figure, our task is to convert the current state into goal state by sliding digits into the blank space.

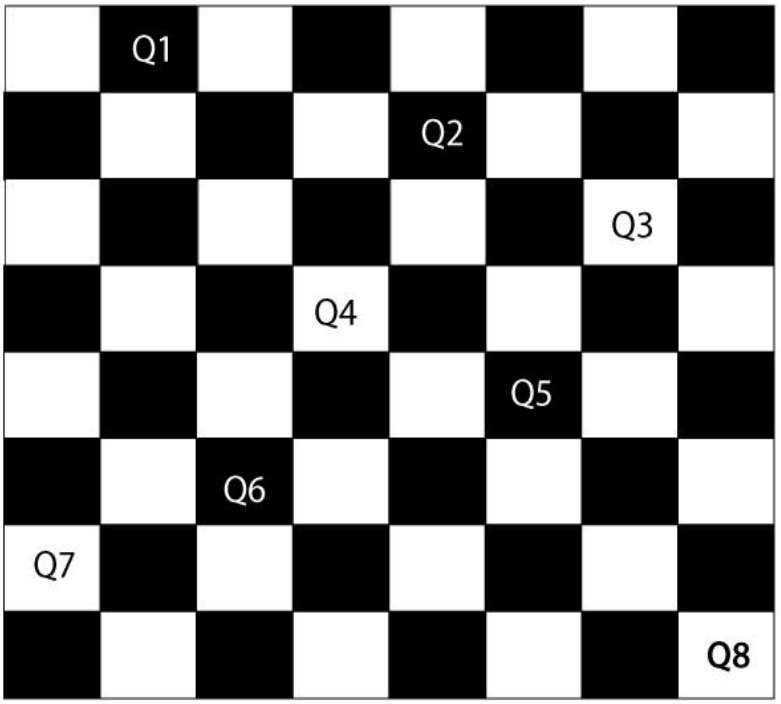


The problem formulation is as follows:

- **States:** It describes the location of each numbered tile and the blank tile.
- **Initial State:** We can start from any state as the initial state.
- **Actions:** Here, actions of the blank space is defined, i.e., either left, right, up or down
- **Transition Model:** It returns the resulting state as per the given state and actions.
- **Goal test:** It identifies whether we have reached the correct goal-state.
- **Path cost:** The path cost is the number of steps in the path where the cost of each step is 1.

2. 8-queens problem:

The aim of this problem is to place eight queens on a chessboard in an order where no queen may attack another. A queen can attack other queens either diagonally or in the same row and column. From the following figure, we can understand the problem as well as its correct solution.



For this problem, there are two main kinds of formulation:

1. Incremental formulation:

It starts from an empty state where the operator augments a queen at each step.

Following steps are involved in this formulation:

- **States:** Arrangement of any 0 to 8 queens on the chessboard.
- **Initial State:** An empty chessboard.
- **Actions:** Add a queen to any empty box.
- **Transition model:** Returns the chessboard with the queen added in a box.

- **Goal test:** Checks whether 8-queens are placed on the chessboard without any attack.
- **Path cost:** There is no need for path cost because only final states are counted. In this formulation, there are approximately 1.8 x 1014 possible sequences to investigate.

2. Complete-state formulation:

It starts with all the 8-queens on the chessboard and moves them around, saving from the attacks.

Following steps are involved in this formulation

- **States:** Arrangement of all the 8 queens one per column with no queen attacking the other queen.
- **Actions:** Move the queen at the location where it is safe from the attacks.

This formulation is better than the incremental formulation as it reduces the state space from 1.8 x 1014 to 2057, and it is easy to find the solutions.

Some Real-world problems

Traveling salesperson problem (TSP): It is a touring problem where the salesman can visit each city only once. The objective is to find the shortest tour and sell-out the stuff in each city.

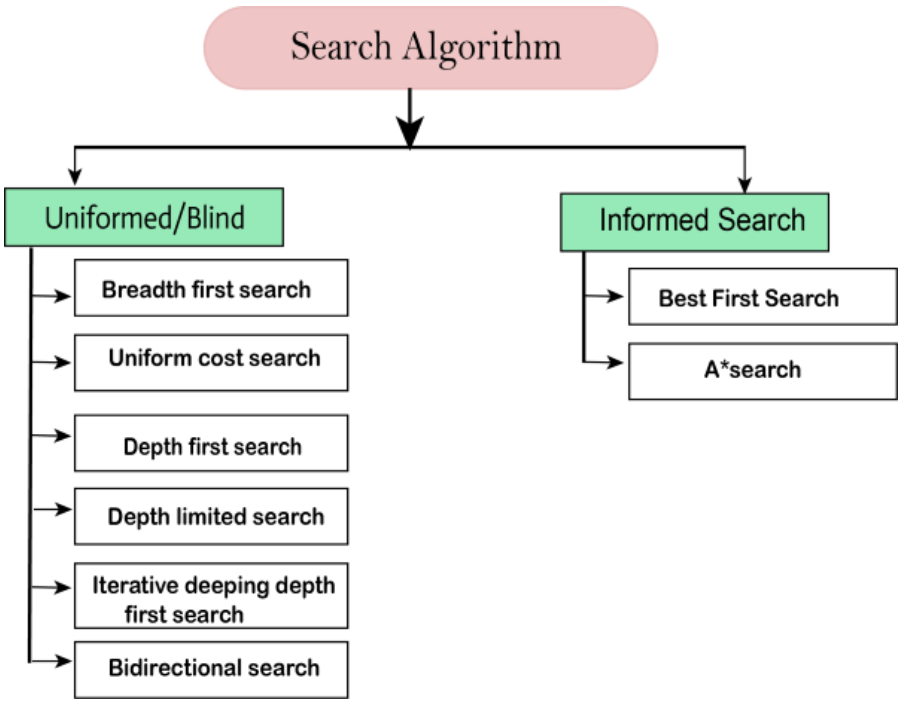
VLSI Layout problem: In this problem, millions of components and connections are positioned on a chip in order to minimize the area, circuit-delays, stray-capacitances, and maximizing the manufacturing yield.

Measuring problem-solving performance

1. **Completeness:** A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.
2. **Optimality:** If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution is said to be an optimal solution.
3. **Time Complexity:** Time complexity is a measure of time for an algorithm to complete its task.
4. **Space Complexity:** It is the maximum storage space required at any point during the search, as the complexity of the problem.

Types of search algorithms

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



Uninformed/Blind Search:

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

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadth wise in a tree or graph, so it is called breadth-first search.
- The BFS algorithm starts searching from the root node of the tree and expands all successor nodes at the current level before moving to nodes of the 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.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

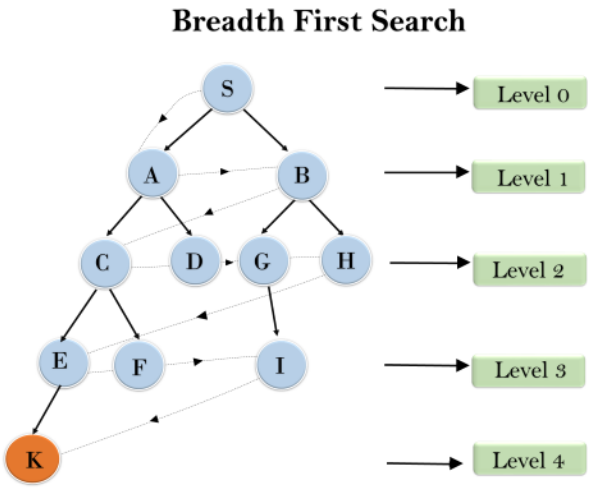
Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

S---> A--->B----->C--->D----->G--->H--->E----->F----->I----->K



1. **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.

$$T(b) = 1+b^2+b^3+.....+ b^d = O(b^d)$$

2. **Space Complexity:** Space complexity of the BFS algorithm is given by the Memory size of the frontier which is $O(b^d)$.
3. **Completeness:** BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.
4. **Optimality:** BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search:

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.

- The process of the DFS algorithm is similar to the BFS algorithm.
  Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach the goal node than the BFS algorithm (if it traverses in the right path).

Disadvantage:

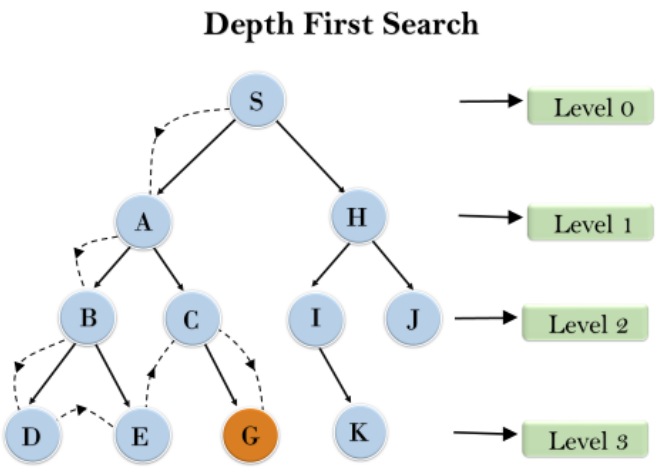
- There is the possibility that many states keep reoccurring, and there is no guarantee of finding the solution.
- The DFS algorithm goes for deep down searching and sometimes it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node -----> right node

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still the goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found the goal node.



1. **Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

2. **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)

3. **Space Complexity:** DFS algorithm needs to store only a single path from the root node. Hence space complexity of DFS is equivalent to the size of the fringe set, which is O(bm).
4. **Optimal:** DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach the goal node.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a pre-determined 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 be treated as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- **Standard failure value:** It indicates that the problem does not have any solution.
- **Cutoff failure value:** It defines no solution for the problem within a given depth limit.

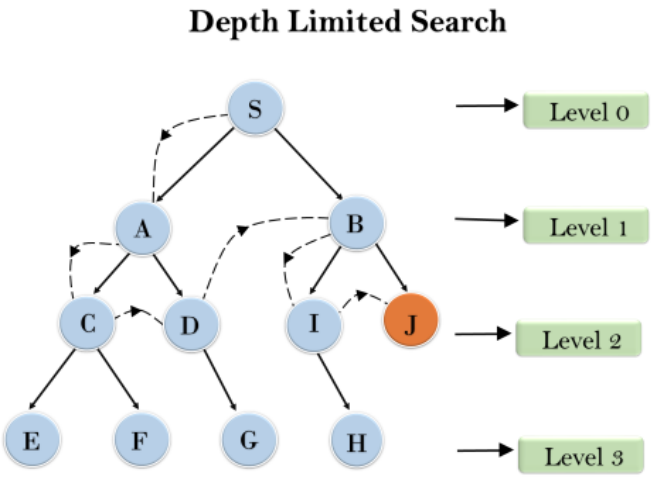
Advantages:

- 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.

Example:



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

4. Uniform-cost Search Algorithm

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 from 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 the BFS algorithm if the path cost of all edges is the same.

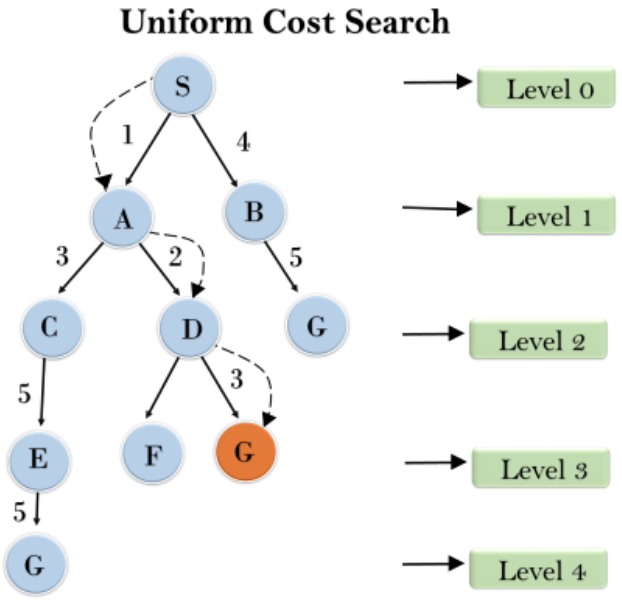
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- It does not care about the number of steps involved in searching and is only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



1. **Completeness:** Uniform-cost search is complete, such as if there is a solution.
2. **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(b1 + [C*/ε])/.
3. **Space Complexity:** The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $\mathbf{O(b^{1 + [C^*/\varepsilon]})}$
4. **Optimal:** Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. Iterative deepening depth-first Search

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful for uninformed search when the search space is large, and depth of the goal node is unknown.

Advantages:

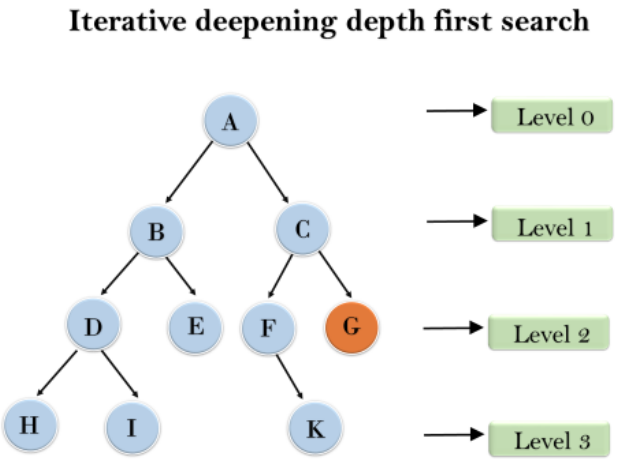
- It Combines the benefits of BFS and DFS search algorithms in terms of fast search and memory efficiency.

Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure is showing the iterative deepening depth-first search. The IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:



1'st Iteration-----> A

2'nd Iteration----> A, B, C

3'rd Iteration------>A, B, D, E, C, F, G

4'th Iteration------>A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.

1. **Completeness:** This algorithm is complete if the branching factor is finite.
2. **Time Complexity:** Let's suppose b is the branching factor and depth is d then the worst-case time complexity is $\mathbf{O(b^d)}$.
3. **Space Complexity:** The space complexity of IDDFS will be O(bd).
4. **Optimal:** IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

6. Bidirectional Search Algorithm:

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

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

Advantages:

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

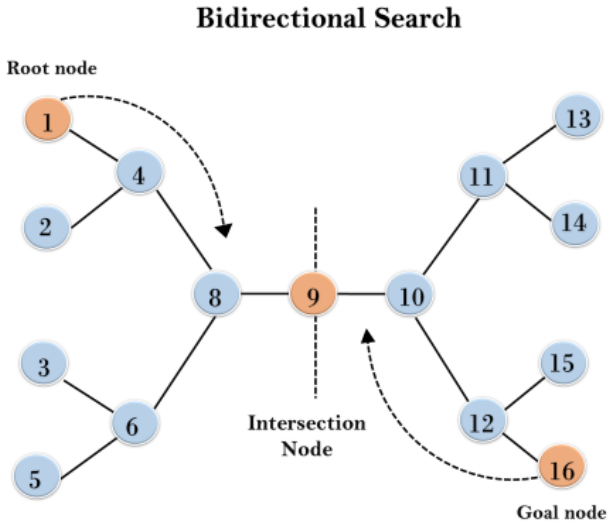
Disadvantages:

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

Example:

In the below search tree, a 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.



1. **Completeness:** Bidirectional Search is complete if we use BFS in both searches.
2. **Time Complexity:** Time complexity of bidirectional search using BFS is $O(b^d)$.
3. **Space Complexity:** Space complexity of bidirectional search is $O(b^d)$.
4. **Optimal:** Bidirectional search is Optimal.

TASKS

Task# 01: Implement a binary tree as follows and traverse it with *(a.) Breadth First Search* and *(b.) Depth First Search.* Print the nodes in the order in which they are visited.

```
class Node:
   def __init__(self ,key):
       self.data = key
       self.left = None
       self.right = None
```

```
root = Node('S')
root.left = Node('A')
root.right = Node('B')
root.left.left = Node('C')
root.left.right = Node('D')
root.right.left = Node('Z')
root.right.right = Node('K')
root.left.left.left = Node('M')
root.left.left.right = Node('O')
root.right.left.right = Node('G')
root.right.left.right.left = Node('H')
print ("Level Order Traversal of binary tree is -")

printOrder(root) # ←  Implement this function
```

Task# 02: Create the following graph and find the Minimum cost from node 0 to node 6 with *Uniform-cost Search* algorithm

```
# create the graph
graph,cost = [[] for i in range(8)],{}
# add edge
graph[0].append(1)
graph[0].append(3)
graph[3].append(1)
graph[3].append(6)
graph[3].append(4)
graph[1].append(6)
graph[4].append(2)
graph[4].append(5)
graph[2].append(1)
graph[5].append(2)
graph[5].append(6)
graph[6].append(4)

# add the cost
cost[(0, 1)] = 2
cost[(0, 3)] = 5
cost[(1, 6)] = 1
cost[(3, 1)] = 5
cost[(3, 6)] = 6
```

```
cost[(3, 4)] = 2
cost[(2, 1)] = 4
cost[(4, 2)] = 4
cost[(4, 5)] = 3
cost[(5, 2)] = 6
cost[(5, 6)] = 3
cost[(6, 4)] = 7

# goal state
goal = []
# set the goal
# there can be multiple goal states
goal.append(6)
# get the answer

answer = uniform_cost_search(goal, 0) # ←  Implement this function


# print answer
print("Minimum cost from 0 to 6 is = ",answer)
```

Task# 04: In the following class "Graph", implement *Depth Limited Search* and *Iterative Deepening Depth Search* methods. Find if the target node = 6, is reachable from source node = 0, given max depth = 3.

```
from collections import defaultdict
# This class represents a directed graph using adjacency list representation
class Graph:
    def __init__(self,vertices):
        # No. of vertices
        self.V = vertices
        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function to perform a Depth-Limited search from given source 'src'
    def DLS(self,src,target,maxDepth):

        # ←  Implement this function
```

```
    # IDDFS to search if the target is reachable from v.
    # It uses recursive DLS()
    def IDDFS(self,src, target, maxDepth):
        # Repeatedly depth-limit search till the
        # maximum depth

        # ←  Implement this function


# Create a graph given in the above diagram
g = Graph (7);
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)

target = 6; maxDepth = 3; src = 0

if g.IDDFS(src, target, maxDepth) == True:
    print ("Target is reachable from source within max depth")
else :
    print ("Target is NOT reachable from source within max depth")
```

Task# 05: Following code implements a class for *Bi-directional Search* with BFS. It checks for intersecting vertices and prints the path from source to target. Implement the bidirectional_search method in this class.

```
# Python program for Bidirectional BFS Search to check path between two
vertices

class AdjacentNode:
    def __init__(self, vertex):
        self.vertex = vertex
        self.next = None

# BidirectionalSearch implementation
class BidirectionalSearch:
    def __init__(self, vertices):
        # Initialize vertices and
```

```
        # graph with vertices
        self.vertices = vertices
        self.graph = [None] * self.vertices
        # Initializing queue for forward and backward search
        self.src_queue = list()
        self.dest_queue = list()

        # Initializing source and destination visited nodes as False
        self.src_visited = [False] * self.vertices
        self.dest_visited = [False] * self.vertices
        # Initializing source and destination parent nodes
        self.src_parent = [None] * self.vertices
        self.dest_parent = [None] * self.vertices

    # Function for adding undirected edge
    def add_edge(self, src, dest):
        # Add edges to graph
        # Add source to destination
        node = AdjacentNode(dest)
        node.next = self.graph[src]
        self.graph[src] = node
        # Since graph is undirected add
        # destination to source
        node = AdjacentNode(src)
        node.next = self.graph[dest]
        self.graph[dest] = node

    # Function for Breadth First Search
    def bfs(self, direction = 'forward'):
        if direction == 'forward':
            # BFS in forward direction
            current = self.src_queue.pop(0)
            connected_node = self.graph[current]
            while connected_node:
                vertex = connected_node.vertex
                if not self.src_visited[vertex]:
                    self.src_queue.append(vertex)
                    self.src_visited[vertex] = True
                    self.src_parent[vertex] = current

                connected_node = connected_node.next
```

```
        else:
            # BFS in backward direction
            current = self.dest_queue.pop(0)
            connected_node = self.graph[current]
            while connected_node:
                vertex = connected_node.vertex
                if not self.dest_visited[vertex]:
                    self.dest_queue.append(vertex)
                    self.dest_visited[vertex] = True
                    self.dest_parent[vertex] = current

                connected_node = connected_node.next

    # Check for intersecting vertex
    def is_intersecting(self):
        # Returns intersecting node
        # if present else -1
        for i in range(self.vertices):
            if (self.src_visited[i] and self.dest_visited[i]):
                return i

        return -1

    # Print the path from source to target
    def print_path(self, intersecting_node, src, dest):
        # Print final path from source to destination
        path = list()
        path.append(intersecting_node)
        i = intersecting_node
        while i != src:
            path.append(self.src_parent[i])
            i = self.src_parent[i]
        path = path[::-1]
        i = intersecting_node
        while i != dest:
            path.append(self.dest_parent[i])
            i = self.dest_parent[i]

        print("*****Path*****")
        path = list(map(str, path))
        print(' '.join(path))
```

```
    # Function for bidirectional searching
    def bidirectional_search(self, src, dest):

        # ←  Implement this function


# Number of Vertices in graph
n = 15
# Source Vertex
src = 0
# Destination Vertex
dest = 14
# Create a graph
graph = BidirectionalSearch(n)
graph.add_edge(0, 4)
graph.add_edge(1, 4)
graph.add_edge(2, 5)
graph.add_edge(3, 5)
graph.add_edge(4, 6)
graph.add_edge(5, 6)
graph.add_edge(6, 7)
graph.add_edge(7, 8)
graph.add_edge(8, 9)
graph.add_edge(8, 10)
graph.add_edge(9, 11)
graph.add_edge(9, 12)
graph.add_edge(10, 13)
graph.add_edge(10, 14)
out = graph.bidirectional_search(src, dest)
if out == -1:
    print(f"Path does not exist between {src} and {dest}")
```

Task# 06:

Traveling Salesman Problem:

Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Like any problem, which can be optimized, there must be a cost function. In the context of TSP, total distance traveled must be reduced as much as possible.

Consider the below matrix representing the distances (Cost) between the cities. Find the shortest possible route that visits every city exactly once and returns to the starting point.

```
# matrix representation of graph
graph = [[0, 10, 15, 20],
         [10, 0, 35, 25],
         [15, 35, 0, 30],
         [20, 25, 30, 0]]
s = 0

print(travellingSalesmanProblem(graph, s)) # ←  Implement this function
```