Assignment 1 : . Implement depth first search algorithm and Breadth First Search algorithm. Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, vertex, edge):
        if vertex not in self.graph:
            self.graph[vertex] = []
        self.graph[vertex].append(edge)

    def dfs(self, start_vertex):
        visited = set()

        def dfs_recursive(vertex):
            visited.add(vertex)
            print(vertex, end=" ")

            for neighbor in self.graph.get(vertex, []):
                if neighbor not in visited:
                    dfs_recursive(neighbor)

        dfs_recursive(start_vertex)
        print()

    def bfs(self, start_vertex):
        visited = set()
        queue = []

        visited.add(start_vertex)
        queue.append(start_vertex)

        while queue:
            vertex = queue.pop(0)
            print(vertex, end=" ")

            for neighbor in self.graph.get(vertex, []):
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)

        print()

# Create an empty graph
```

```
g = Graph()

# Take user input for vertices and edges
while True:
    vertex = input("Enter a vertex (or 'done' to finish adding vertices): ")
    if vertex.lower() == 'done':
        break

    edge = input("Enter an edge for {}: ".format(vertex))
    g.add_edge(vertex, edge)

# Take user input for traversal type
while True:
    traversal_type = input("Enter 'DFS' or 'BFS' to perform traversal (or 'exit' to quit): ").upper()

    if traversal_type == 'EXIT':
        break
    elif traversal_type == 'DFS':
        start_vertex = input("Enter the starting vertex: ")
        print("DFS traversal:")
        g.dfs(start_vertex)
    elif traversal_type == 'BFS':
        start_vertex = input("Enter the starting vertex: ")
        print("BFS traversal:")
        g.bfs(start_vertex)
    else:
        print("Invalid input. Please enter 'DFS' or 'BFS' (or 'exit' to quit).")
```

```
Enter a vertex (or 'done' to finish adding vertices): 1
Enter an edge for 1: 2
Enter a vertex (or 'done' to finish adding vertices): 1
Enter an edge for 1: 3
Enter a vertex (or 'done' to finish adding vertices): 2
Enter an edge for 2: 6
Enter a vertex (or 'done' to finish adding vertices): 6
Enter an edge for 6: 4
Enter a vertex (or 'done' to finish adding vertices): 4
Enter an edge for 4: 3
Enter a vertex (or 'done' to finish adding vertices): done
Enter 'DFS' or 'BFS' to perform traversal (or 'exit' to quit): dfs
Enter the starting vertex: 1
DFS traversal:
1 2 6 4 3
Enter 'DFS' or 'BFS' to perform traversal (or 'exit' to quit):
```

Assignment 2 : Implement A star (A*) Algorithm for any game search problem.

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}              #store distance from starting node
    parents = {}        # parents contains an adjacency map of all nodes
    #distance of starting node from itself is zero
    g[start_node] = 0
    #start_node is root node i.e it has no parent nodes
    #so start_node is set to its own parent node
    parents[start_node] = start_node
    while len(open_set) > 0:
        n = None
        #node with lowest f() is found
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):
                #nodes 'm' not in first and last set are added to first
                #n is set its parent
                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                #for each node m,compare its distance from start i.e g(m) to the
                #from start through n node
                else:
                    if g[m] > g[n] + weight:
                        #update g(m)
                        g[m] = g[n] + weight
                        #change parent of m to n
                        parents[m] = n
                        #if m in closed set,remove and add to open
                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
```

```
            return None

        # if the current node is the stop_node
        # then we begin reconstructin the path from it to the start_node
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path
        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
```

```
/tmp/vyJCE99c1n.o
The destination cell is found
-> (0,0) -> (1,1) -> (2,2) -> (3,2) -> (4,2) -> (5,3) -> (5,4) -> (6,5) -> (7,5) -> (8,6) -> (7,7) -> (7,8)
    -> (8,9)
```

Assignment 3: Implement Alpha-Beta Tree search for any game search problem.

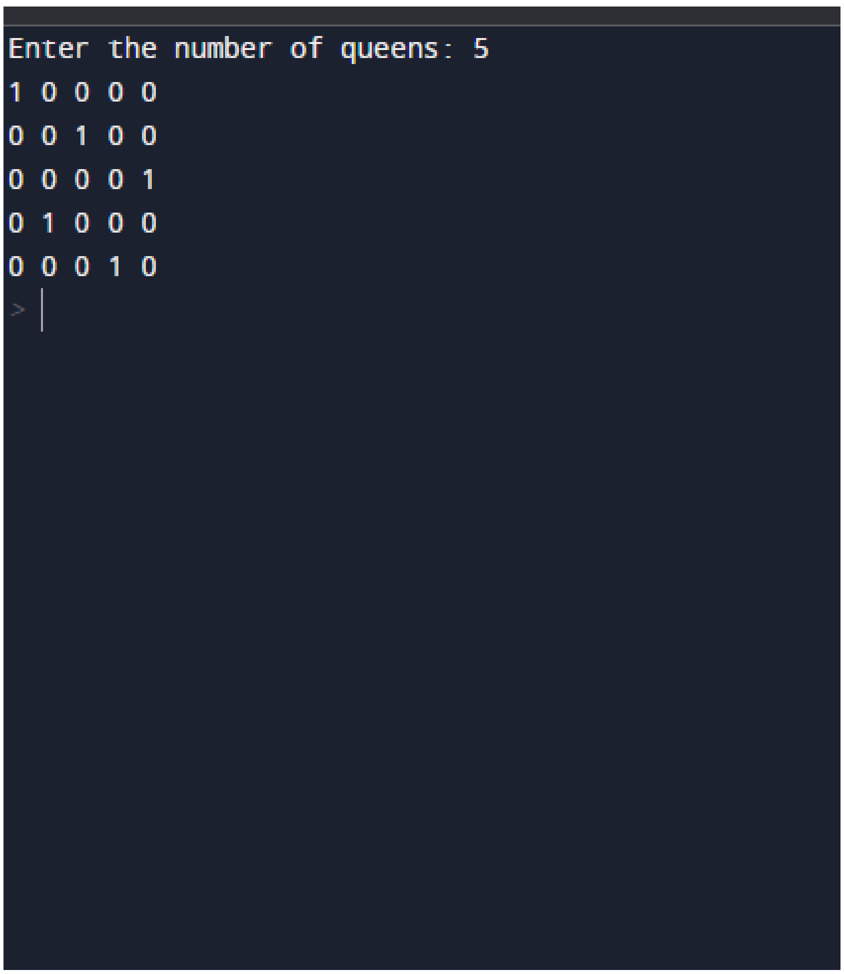
```
MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):

    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                          False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

    else:
        best = MAX

        # Recur for left and
        # right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                            True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

# Driver Code
if __name__ == "__main__":

    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

```
/tmp/vyJCE99c1n.o
The optimal value is : 12
```

Assignment 4: Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem

```
def is_safe(board, row, col, n):
    # Check if there is a queen in the same column
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check upper-left diagonal
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check upper-right diagonal
    for i, j in zip(range(row, -1, -1), range(col, n)):
        if board[i][j] == 1:
            return False

    return True

def solve_n_queens_util(board, row, n):
    if row == n:
        # All queens are placed, solution found
        print_board(board, n)
        return True

    for col in range(n):
        if is_safe(board, row, col, n):
            board[row][col] = 1
            if solve_n_queens_util(board, row + 1, n):
                return True
            board[row][col] = 0

    return False

def print_board(board, n):
    for i in range(n):
        for j in range(n):
            print(board[i][j], end=" ")
        print()

def solve_n_queens(n):
    board = [[0] * n for _ in range(n)]
```

```
    if not solve_n_queens_util(board, 0, n):
        print("No solution exists")

if __name__ == "__main__":
    try:
        n = int(input("Enter the number of queens: "))
        solve_n_queens(n)
    except ValueError:
        print("Invalid input. Please enter a valid number.")
```



Assignment 5 : Implement Greedy search algorithm for any of the following application:

```
def dijkstra(graph, start):
    vertices = len(graph)
    visited = [False] * vertices
    dist = [float('inf')] * vertices
    dist[start] = 0

    for _ in range(vertices):
        min_dist =  float('inf')
        for v in range(vertices):
            if not visited[v] and dist[v] < min_dist:
                min_dist = dist[v]
                u = v

        visited[u] = True

        for v in range(vertices):
            if not visited[v] and graph[u][v] > 0:
                if dist[u] + graph[u][v] < dist[v]:
                    dist[v] = dist[u] + graph[u][v]

    return dist

# Input for the graph
n = int(input("Enter the number of vertices: "))
graph = []

print("Enter the adjacency matrix:")
for _ in range(n):
    row = list(map(int, input().split()))
    graph.append(row)

start_vertex = int(input("Enter the starting vertex (0 to {}): ".format(n - 1)))

shortest_distances = dijkstra(graph, start_vertex)

print("Shortest distances from vertex {}:".format(start_vertex))
for i, distance in enumerate(shortest_distances):
    print("Vertex {}: {}".format(i, distance))
```

```
Enter the number of vertices: 3
Enter the adjacency matrix:
0 1 4
1 0 3
0 1 1
Enter the starting vertex (0 to 2): 0
Shortest distances from vertex 0:
Vertex 0: 0
Vertex 1: 1
Vertex 2: 4
>
```

Prim’s Algorithm

```
def prim(graph):
    vertices = len(graph)
    parent = [-1] * vertices
    key = [float('inf')] * vertices
    key[0] = 0
    mst_set = [False] * vertices

    for _ in range(vertices):
        min_key =  float('inf')
        for v in range(vertices):
            if not mst_set[v] and key[v] < min_key:
                min_key = key[v]
                u = v

        mst_set[u] = True


        for v in range(vertices):
            if graph[u][v] > 0 and not mst_set[v] and graph[u][v] < key[v]:
                parent[v] = u
                key[v] = graph[u][v]

    return parent
```

```
# Input for the graph
n = int(input("Enter the number of vertices: "))
graph = []

print("Enter the adjacency matrix:")
for _ in range(n):
    row = list(map(int, input().split()))
    graph.append(row)

minimum_spanning_tree = prim(graph)

print("Minimum Spanning Tree:")
for i in range(1, n):
    print("Edge: {} - {}".format(minimum_spanning_tree[i], i
```

```
Enter the number of vertices:  4
4
Enter the adjacency matrix:
1 0 1 3
0 1 0 4
1 2 3 4
0 5 0 1
Minimum Spanning Tree:
Edge: 2 - 1
Edge: 0 - 2
Edge: 0 - 3
>
```