Ex 1#bfs

```
import networkx as nx

import matplotlib.pyplot as plt

from collections import deque


def bfs(graph, start):

    visited = set()

    queue = deque([start])


    while queue:

        vertex = queue.popleft()

        if vertex not in visited:

            print(vertex)

            visited.add(vertex)

            queue.extend(neighbor for neighbor in graph[vertex] if neighbor not in visited)


    return visited


graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F'],

    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E']

}


bfs(graph, 'A')


G = nx.Graph()


for node in graph:

    for neighbor in graph[node]:

        G.add_edge(node, neighbor)
```

```
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray', node_size=3000, font_size=15, font_weight='bold')

plt.title("Graph Visualization : BFS")
plt.show()
```

Ex 2

```
#dfs
import networkx as nx
import matplotlib.pyplot as plt

def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start)

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

    return visited

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

dfs(graph, 'A')
```

```
G = nx.Graph()

for node in graph:
    for neighbor in graph[node]:
        G.add_edge(node, neighbor)

pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='lightblue', edge_color='gray', node_size=3000, font_size=15, font_weight='bold')

plt.title("Graph Visualization:DFS")
plt.show()
```

ex 3

```
#tic tac toe

def print_board(board):
    print('-------------')
    for row in board:
        print('|', ' | '.join(row) + ' |')
        print('-------------')

def check_winner(board, player):
    # Check rows and columns
    for row in board:
        if all(cell == player for cell in row):
            return True
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
    # Check diagonals
    if all(board[i][i] == player for i in range(3)) or all(board[i][2-i] == player for i in range(3)):
        return True
    return False
```

```python
def is_full(board):
    # Return True if all cells are filled
    for row in board:
        if any(cell == ' ' for cell in row):
            return False
    return True


def tic_tac_toe():
    board = [[' ' for _ in range(3)] for _ in range(3)]  # Initialize board with spaces
    current_player = 'X'

    while True:
        print_board(board)
        print(f"Player {current_player}'s turn")

        while True:
            try:
                row = int(input("Enter row (1-3): ")) - 1
                col = int(input("Enter column (1-3): ")) - 1
                if row in range(3) and col in range(3) and board[row][col] == ' ':
                    break
                else:
                    print("Invalid move. Try again.")
            except ValueError:
                print("Invalid input. Please enter a number between 1 and 3.")

        board[row][col] = current_player

        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break

        if is_full(board):
```

```
        print_board(board)
        print("It's a tie!")
        break

    current_player = 'O' if current_player == 'X' else 'X'

if __name__ == "__main__":
    tic_tac_toe()
```

ex 4

```
#8 puzle game

import heapq

class PuzzleState:
    def __init__(self, board, goal, moves=0, previous=None):
        self.board = board
        self.goal = goal
        self.moves = moves
        self.previous = previous
        self.priority = self.moves + self.manhattan_distance()

    def manhattan_distance(self):
        distance = 0
        for i in range(3):
            for j in range(3):
                if self.board[i][j] != 0:
                    goal_flat = [num for row in self.goal for num in row]
                    goal_x, goal_y = divmod(goal_flat.index(self.board[i][j]), 3)
                    distance += abs(i - goal_x) + abs(j - goal_y)
        return distance

    def is_goal(self):
        return self.board == self.goal
```

```
    def get_neighbors(self):
        neighbors = []
        x, y = self.find_zero()

        def swap_and_create(new_x, new_y):
            new_board = [row[:] for row in self.board]
            new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]
            neighbors.append(PuzzleState(new_board, self.goal, self.moves + 1, self))

        if x > 0: swap_and_create(x - 1, y)
        if x < 2: swap_and_create(x + 1, y)
        if y > 0: swap_and_create(x, y - 1)
        if y < 2: swap_and_create(x, y + 1)

        return neighbors

    def find_zero(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j

    def __lt__(self, other):
        return self.priority < other.priority

def solve_puzzle(start, goal):
    start_state = PuzzleState(start, goal)
    priority_queue = []
    heapq.heappush(priority_queue, start_state)
    visited = set()

    while priority_queue:
        current_state = heapq.heappop(priority_queue)

        if current_state.is_goal():
```

```
        return reconstruct_path(current_state)

      visited.add(tuple(map(tuple, current_state.board)))

      for neighbor in current_state.get_neighbors():
        if tuple(map(tuple, neighbor.board)) not in visited:
          heapq.heappush(priority_queue, neighbor)

  return None

def reconstruct_path(state):
  path = []
  while state:
    path.append(state.board)
    state = state.previous
  return path[::-1]

# Example usage:
start_board = [
  [1, 2, 3],
  [4, 0, 5],
  [7, 8, 6]
]

goal_board = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 0]
]

solution_path = solve_puzzle(start_board, goal_board)
for step in solution_path:
  for row in step:
    print(row)
  print()
```

ex 5 water jug

```
from collections import deque

# Define the jug capacities
JUG1_CAPACITY = 3
JUG2_CAPACITY = 5

# Define the goal state
GOAL_STATE = (0, 4)  # 0 liters in jug1, 4 liters in jug2

# Define the possible actions
ACTIONS = [
    ("fill jug1", lambda x, y: (JUG1_CAPACITY, y)),
    ("fill jug2", lambda x, y: (x, JUG2_CAPACITY)),
    ("empty jug1", lambda x, y: (0, y)),
    ("empty jug2", lambda x, y: (x, 0)),
    ("pour jug1 to jug2", lambda x, y: (max(0, x - (JUG2_CAPACITY - y)), min(y + x, JUG2_CAPACITY))),
    ("pour jug2 to jug1", lambda x, y: (min(x + y, JUG1_CAPACITY), max(0, y - (JUG1_CAPACITY - x))))
]

def bfs_search():
    # Initialize the queue with the initial state and the action path
    queue = deque([((0, 0), [])])  # state (0 liters in both jugs), empty path

    # Initialize the set to keep track of visited states
    visited = set([(0, 0)])

    while queue:
        # Dequeue the next state and the current path
        (state, path) = queue.popleft()

        # If the goal state is reached, return the path
        if state == GOAL_STATE:
            return path

        # Apply each possible action to the current state
```

```
        for action, next_state_func in ACTIONS:
            next_state = next_state_func(*state)

            # If the next state is not visited, mark it as visited and enqueue it
            if next_state not in visited:
                visited.add(next_state)
                queue.append((next_state, path + [(action, state, next_state)]))

    # If no solution is found, return None
    return None

# Run the BFS search
solution = bfs_search()

# Print the solution
if solution:
    print("Solution found:")
    for action, state, next_state in solution:
        print(f"{action}: {state} -> {next_state}")
else:
    print("No solution found.")
```

Ex:6 TSP

```
import itertools
import matplotlib.pyplot as plt
import networkx as nx
def tsp(graph):
    num_cities = len(graph)
    all_routes = itertools.permutations(range(num_cities))
    min_cost = float('inf')
    best_route = None
    for route in all_routes:
        cost = sum(graph[route[i]][route[i + 1]] for i in range(num_cities - 1))
        cost += graph[route[-1]][route[0]]
```

```
        if cost < min_cost:
            min_cost = cost
            best_route = route
    return min_cost, best_route
def draw_tsp_graph():
    G = nx.Graph()
    cities = [0, 1, 2, 3]
    G.add_nodes_from(cities)
    G.add_weighted_edges_from([(0, 1, 10), (0, 2, 15), (1, 2, 35), (1, 3, 25), (2, 3, 30)])
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=500)
    edge_labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
    plt.title("Traveling Salesman Problem (TSP)")
    plt.show()
graph = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
min_cost, best_route = tsp(graph)
print("Minimum cost:", min_cost)
print("Best route:", best_route)
draw_tsp_graph()
```

EX 7 TOWER OF HANOI

```
import matplotlib.pyplot as plt
import networkx as nx
def tower_of_hanoi(n, source, target, auxiliary):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n - 1, source, auxiliary, target)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n - 1, auxiliary, target, source)
def draw_tower_of_hanoi():
    G = nx.DiGraph()  # Use directed graph to show the flow
```

```
    pegs = ['Peg A', 'Peg B', 'Peg C']
    G.add_nodes_from(pegs)
    G.add_edges_from([('Peg A', 'Peg B'), ('Peg A', 'Peg C'), ('Peg B', 'Peg C')])
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color='lightgreen', node_size=2000, font_size=10)
    plt.title("Tower of Hanoi")
    plt.show()
n = 3
tower_of_hanoi(n, 'A', 'C', 'B')
draw_tower_of_hanoi()
```

EX 8 MONKEY BANANA

```
import matplotlib.pyplot as plt
import networkx as nx
class Monkey:
    def __init__(self, has_banana=False):
        self.has_banana = has_banana
    def get_banana(self):
        self.has_banana = True
        print("Monkey got the banana!")
def draw_monkey_banana():
    G = nx.Graph()
    nodes = ['Monkey', 'Banana', 'Box']

    G.add_nodes_from(nodes)
    G.add_edges_from([('Monkey', 'Banana'), ('Monkey', 'Box')])
    pos = {'Monkey': (0, 0), 'Box': (1, -1), 'Banana': (1, 1)}
    nx.draw(G, pos, with_labels=True, node_color='lightyellow', node_size=2000, font_size=10)
    plt.title("Monkey Banana Problem")
    plt.show()
monkey = Monkey()
monkey.get_banana()
draw_monkey_banana()
```

EX 9 ALPHA BETA

```
import math
```

```
import matplotlib.pyplot as plt

import networkx as nx

def alpha_beta_pruning(depth, node_index, is_maximizing_player, values, alpha, beta, graph, parent=None):

    if depth == 3:  # At the leaf node level

        graph.add_node(node_index, label=str(values[node_index]))  # Leaf node

        if parent is not None:

            graph.add_edge(parent, node_index)

        return values[node_index]

    if is_maximizing_player:

        best = -math.inf

        graph.add_node(node_index, label=f"α={alpha}, β={beta}")

        if parent is not None:

            graph.add_edge(parent, node_index)

        for i in range(2):  # Each node has 2 children

            val = alpha_beta_pruning(depth + 1, node_index * 2 + i, False, values, alpha, beta, graph, node_index)

            best = max(best, val)

            alpha = max(alpha, best)

            if beta <= alpha:

                break  # Beta cutoff (prune remaining children)

        return best

    else:

        best = math.inf

        graph.add_node(node_index, label=f"α={alpha}, β={beta}")

        if parent is not None:

            graph.add_edge(parent, node_index)


        for i in range(2):

            val = alpha_beta_pruning(depth + 1, node_index * 2 + i, True, values, alpha, beta, graph, node_index)

            best = min(best, val)

            beta = min(beta, best)

            if beta <= alpha:

                break  # Alpha cutoff (prune remaining children)

        return best

def draw_tree(graph):

    pos = nx.spring_layout(graph)  # Layout for visualizing the tree

    labels = nx.get_node_attributes(graph, 'label')

    plt.figure(figsize=(10, 8))
```

```
    nx.draw(graph, pos, with_labels=True, labels=labels, node_size=3000, node_color='lightblue', font_size=10)
    plt.title("Alpha-Beta Pruning Decision Tree")
    plt.show()
values = [3, 5, 6, 9, 1, 2, 0, -1]
alpha = -math.inf
beta = math.inf
graph = nx.DiGraph()
optimal_value = alpha_beta_pruning(0, 0, True, values, alpha, beta, graph)
print("Optimal value:", optimal_value)
draw_tree(graph)
```

EX 10: 8 QUEEN

```
def is_safe(board, row, col):
    for i in range(col):
        if board[row][i] == 1 or \
           (row - i >= 0 and board[row - i][col - i] == 1) or \
           (row + i < len(board) and board[row + i][col - i] == 1):
            return False
    return True
def solve_n_queens(board, col):
    if col >= len(board):
        display_board(board)
        return True
    for row in range(len(board)):
        if is_safe(board, row, col):
            board[row][col] = 1
            if solve_n_queens(board, col + 1):
                return True
            board[row][col] = 0  # Backtrack
    return False
def display_board(board):
    for row in board:
        print(" ".join("Q" if x else "." for x in row))
    print()
def n_queens(n):
    board = [[0] * n for _ in range(n)]
    solve_n_queens(board, 0)
n_queens(8)
```