# BICOL UNIVERSITY COLLEGE OF SCIENCE
## CS Elective – Artificial Intelligence
## Class Participation #2

**Search Algorithms using Python**

1. **# 8-Puzzle Solver using Breadth-First Search (BFS)**

```
from collections import deque
# Commonly used when implementing BFS for puzzles like the 8-puzzle
# Define the goal state
goal_state =    [[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 0]]

# Helper function to find the position of the blank (0)
def find_blank(state):
  for i in range(3):
    for j in range(3):
      if state[i][j] == 0:
        return i, j

# Check if two states are equal
def is_goal(state):
  return state == goal_state

# Generate new states from the current state
def get_neighbors(state):
  neighbors = []
  x, y = find_blank(state)
  moves = [(-1,0),(1,0),(0,-1),(0,1)]  # up, down, left, right

  for dx, dy in moves:
    nx, ny = x + dx, y + dy
    if 0 <= nx < 3 and 0 <= ny < 3:
      new_state = [row[:] for row in state]  # deep copy
      new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
      neighbors.append(new_state)
  return neighbors

# Convert state to a tuple for hashing
def state_to_tuple(state):
  return tuple(tuple(row) for row in state)

# BFS Implementation
def bfs(start_state):
```

```
    visited = set()
    queue = deque()
    queue.append((start_state, []))  # state and path
    visited.add(state_to_tuple(start_state))

    while queue:
        state, path = queue.popleft()

        if is_goal(state):
            return path + [state]

        for neighbor in get_neighbors(state):
            t_neighbor = state_to_tuple(neighbor)
            if t_neighbor not in visited:
                visited.add(t_neighbor)
                queue.append((neighbor, path + [state]))

    return None  # if no solution found

# Example start state
start_state =   [[5, 4, 0],
                 [6, 1, 8],
                 [7, 3, 2]]  # you can modify this

# Solve the puzzle
solution = bfs(start_state)

# Display the solution
if solution:
    print(f"Solution found in {len(solution)-1} moves:")
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

**2. # 8-Puzzle Solver using Depth-First Search (DFS)**

```
from copy import deepcopy
#deepcopy ensures that when you modify next_state in get_neighbors, not overwriting the original state
# Define the goal state
goal_state =     [[1, 2, 3],
                  [4, 5, 6],
                  [7, 8, 0]]

# Helper function to find the position of the blank (0)
def find_blank(state):
  for i in range(3):
    for j in range(3):
      if state[i][j] == 0:
        return i, j

# Check if two states are equal
def is_goal(state):
  return state == goal_state

# Generate new states from the current state
def get_neighbors(state):
  neighbors = []
  x, y = find_blank(state)
  moves = [(-1,0),(1,0),(0,-1),(0,1)]  # up, down, left, right

  for dx, dy in moves:
    nx, ny = x + dx, y + dy
    if 0 <= nx < 3 and 0 <= ny < 3:
      new_state = deepcopy(state)
      new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
      neighbors.append(new_state)
  return neighbors

# Convert state to a tuple for hashing
def state_to_tuple(state):
  return tuple(tuple(row) for row in state)

# DFS Implementation
def dfs(start_state, max_depth=50):
  visited = set()
  stack = [(start_state, [], 0)]  # state, path, depth

  while stack:
    state, path, depth = stack.pop()

    if is_goal(state):
```

```
            return path + [state]

        if depth >= max_depth:
            continue

        t_state = state_to_tuple(state)
        if t_state not in visited:
            visited.add(t_state)
            for neighbor in reversed(get_neighbors(state)):  # reverse to simulate left-to-right exploration
                stack.append((neighbor, path + [state], depth + 1))

    return None  # if no solution found

# Example start state
start_state =   [[5, 4, 0],
                 [6, 1, 8],
                 [7, 3, 2]]  # modify this as needed

# Solve the puzzle
solution = dfs(start_state, max_depth=30)  # adjust max_depth as needed

# Display the solution
if solution:
    print(f"Solution found in {len(solution)-1} moves:")
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print()
else:
    print("No solution found within max depth.")
```

### 3. # 8-Puzzle Solver using Uniform Cost Search (UCS)

```
from heapq import heappush, heappop
from copy import deepcopy
#deepcopy ensures that when you modify next_state in get_neighbors, not overwriting the original state
#heappush(heap, item) → pushes an item into the heap while maintaining order.
#heappop(heap) → pops the smallest-priority item.
#In UCS, the priority is the path cost so far (g(n)), so the node with the lowest cost gets expanded first.

# Define the goal state
goal_state =    [[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 0]]

# Helper function to find the blank (0) position
def find_blank(state):
  for i in range(3):
    for j in range(3):
      if state[i][j] == 0:
        return i, j

# Check if a state is the goal
def is_goal(state):
  return state == goal_state

# Generate neighbors (valid moves)
def get_neighbors(state):
  neighbors = []
  x, y = find_blank(state)
  moves = [(-1,0),(1,0),(0,-1),(0,1)]  # up, down, left, right

  for dx, dy in moves:
    nx, ny = x + dx, y + dy
    if 0 <= nx < 3 and 0 <= ny < 3:
      new_state = deepcopy(state)
      new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
      neighbors.append(new_state)
  return neighbors

# Convert state to tuple for hashing
def state_to_tuple(state):
  return tuple(tuple(row) for row in state)

# UCS Implementation
def ucs(start_state):
  visited = set()
  pq = []
```

```
    heappush(pq, (0, start_state, []))  # (cost, state, path)

    while pq:
        cost, state, path = heappop(pq)

        if is_goal(state):
            return path + [state], cost

        t_state = state_to_tuple(state)
        if t_state not in visited:
            visited.add(t_state)

            for neighbor in get_neighbors(state):
                heappush(pq, (cost + 1, neighbor, path + [state]))  # cost per move = 1

    return None, None  # no solution found

# Example start state
start_state =   [[5, 4, 0],
                 [6, 1, 8],
                 [7, 3, 2]]  # modify as needed

# Solve the puzzle
solution, total_cost = ucs(start_state)

# Display the solution
if solution:
    print(f"Solution found in {len(solution)-1} moves with total cost {total_cost}:")
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

**4. # 8-Puzzle Solver using Depth-Limited Search (DLS)**

```
from copy import deepcopy

# Define the goal state
goal_state =    [[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 0]]

# Helper function to find the blank (0) position
def find_blank(state):
  for i in range(3):
    for j in range(3):
      if state[i][j] == 0:
        return i, j

# Check if a state is the goal
def is_goal(state):
  return state == goal_state

# Generate neighbors (valid moves)
def get_neighbors(state):
  neighbors = []
  x, y = find_blank(state)
  moves = [(-1,0),(1,0),(0,-1),(0,1)]  # up, down, left, right

  for dx, dy in moves:
    nx, ny = x + dx, y + dy
    if 0 <= nx < 3 and 0 <= ny < 3:
      new_state = deepcopy(state)
      new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
      neighbors.append(new_state)
  return neighbors

# Convert state to tuple for hashing
def state_to_tuple(state):
  return tuple(tuple(row) for row in state)

# Depth-Limited Search (DLS) Implementation
def dls(start_state, limit):
  visited = set()
  stack = [(start_state, [], 0)]  # state, path, depth

  while stack:
    state, path, depth = stack.pop()

    if is_goal(state):
```

```
            return path + [state]

        if depth >= limit:
            continue

        t_state = state_to_tuple(state)
        if t_state not in visited:
            visited.add(t_state)

            for neighbor in reversed(get_neighbors(state)):
                stack.append((neighbor, path + [state], depth + 1))

    return None  # no solution within depth limit

# Example start state
start_state =    [[5, 4, 0],
                  [6, 1, 8],
                  [7, 3, 2]]  # modify as needed

# Set depth limit
depth_limit = 30  # adjust as needed

# Solve the puzzle
solution = dls(start_state, depth_limit)

# Display the solution
if solution:
    print(f"Solution found in {len(solution)-1} moves:")
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print()
else:
    print(f"No solution found within depth limit {depth_limit}.")
```

**5. # 8-Puzzle Solver using Iterative Deepening Search (IDS)**

```
from copy import deepcopy

# Define the goal state
goal_state =    [[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 0]]

# Helper function to find the blank (0) position
def find_blank(state):
  for i in range(3):
    for j in range(3):
      if state[i][j] == 0:
        return i, j

# Check if a state is the goal
def is_goal(state):
  return state == goal_state

# Generate neighbors (valid moves)
def get_neighbors(state):
  neighbors = []
  x, y = find_blank(state)
  moves = [(-1,0),(1,0),(0,-1),(0,1)]  # up, down, left, right

  for dx, dy in moves:
    nx, ny = x + dx, y + dy
    if 0 <= nx < 3 and 0 <= ny < 3:
      new_state = deepcopy(state)
      new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
      neighbors.append(new_state)
  return neighbors

# Convert state to tuple for hashing
def state_to_tuple(state):
  return tuple(tuple(row) for row in state)

# Depth-Limited Search used in IDS
def dls_ids(state, limit, visited=set()):
  stack = [(state, [], 0)]  # state, path, depth

  while stack:
    current_state, path, depth = stack.pop()

    if is_goal(current_state):
      return path + [current_state]
```

```
        if depth >= limit:
            continue

        t_state = state_to_tuple(current_state)
        if t_state not in visited:
            visited.add(t_state)

            for neighbor in reversed(get_neighbors(current_state)):
                stack.append((neighbor, path + [current_state], depth + 1))

    return None

# Iterative Deepening Search (IDS)
def ids(start_state, max_depth=50):
    for depth in range(max_depth):
        visited = set()
        result = dls_ids(start_state, depth, visited)
        if result is not None:
            return result
    return None

# Example start state
start_state =   [[5, 4, 0],
                 [6, 1, 8],
                 [7, 3, 2]]  # modify as needed

# Solve the puzzle
solution = ids(start_state, max_depth=100)  # adjust max_depth as needed

# Display the solution
if solution:
    print(f"Solution found in {len(solution)-1} moves:")
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print()
else:
    print("No solution found within max depth.")
```

### 6. # 8-Puzzle Solver using Greedy Best-First Search (GBFS)

```
from heapq import heappush, heappop
from copy import deepcopy

# Start and goal states
start_state =     [[5, 4, 0],
                   [6, 1, 8],
                   [7, 3, 2]]

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

# Helper functions
def find_blank(state):
  for i in range(3):
    for j in range(3):
      if state[i][j] == 0:
        return i, j

def is_goal(state):
  return state == goal_state

def get_neighbors(state):
  neighbors = []
  x, y = find_blank(state)
  moves = [(-1,0),(1,0),(0,-1),(0,1)]  # up, down, left, right

  for dx, dy in moves:
    nx, ny = x + dx, y + dy
    if 0 <= nx < 3 and 0 <= ny < 3:
      new_state = deepcopy(state)
      new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
      neighbors.append(new_state)
  return neighbors

def state_to_tuple(state):
  return tuple(tuple(row) for row in state)

# Heuristic: Manhattan Distance
def manhattan_distance(state):
  distance = 0
  for i in range(3):
    for j in range(3):
      val = state[i][j]
      if val != 0:
```

```
            goal_x, goal_y = divmod(val-1, 3)
            distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

# Greedy Best-First Search
def gbfs(start_state):
    visited = set()
    pq = []
    heappush(pq, (manhattan_distance(start_state), start_state, []))  # (heuristic, state, path)

    while pq:
        h, state, path = heappop(pq)

        # Show heuristic value of expanded node
        print(f"Expanding node with heuristic h(n) = {h}")

        if is_goal(state):
            return path + [state]

        t_state = state_to_tuple(state)
        if t_state not in visited:
            visited.add(t_state)
            for neighbor in get_neighbors(state):
                h_val = manhattan_distance(neighbor)
                heappush(pq, (h_val, neighbor, path + [state]))

    return None

# Solve the puzzle
solution = gbfs(start_state)

# Display the solution
if solution:
    print(f"\nSolution found in {len(solution)-1} moves:")
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

**7. # 8-Puzzle Solver using A* Search**

```
from heapq import heappush, heappop
from copy import deepcopy

# Start and goal states
start_state = [[5, 4, 0],
               [6, 1, 8],
               [7, 3, 2]]

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

# Helper functions
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def is_goal(state):
    return state == goal_state

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    moves = [(-1,0),(1,0),(0,-1),(0,1)]  # up, down, left, right

    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = deepcopy(state)
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def state_to_tuple(state):
    return tuple(tuple(row) for row in state)

# Heuristic: Manhattan Distance
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            val = state[i][j]
            if val != 0:
                goal_x, goal_y = divmod(val-1, 3)
```

```
            distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

# A* Search Implementation with heuristic display
def a_star(start_state):
    visited = set()
    pq = []
    g_cost = 0
    h_cost = manhattan_distance(start_state)
    f_cost = g_cost + h_cost
    heappush(pq, (f_cost, g_cost, start_state, []))  # (f, g, state, path)

    while pq:
        f, g, state, path = heappop(pq)

        # Calculate h for current state
        h = f - g

        # Display expansion details
        print(f"Expanding node with g={g}, h={h}, f={f}")
        if is_goal(state):
            return path + [state], g  # solution and number of moves

        t_state = state_to_tuple(state)
        if t_state not in visited:
            visited.add(t_state)
            for neighbor in get_neighbors(state):
                g_new = g + 1
                h_new = manhattan_distance(neighbor)
                f_new = g_new + h_new
                heappush(pq, (f_new, g_new, neighbor, path + [state]))

    return None, None

# Solve the puzzle
solution, total_moves = a_star(start_state)

# Display the solution
if solution:
    print(f"\nSolution found in {total_moves} moves:")
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

**Experiment with the provided start and goal states, and answer the following:**

1. 10 points

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

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

1. How many total steps for BFS? DFS? IDS? 3 points
2. For UCS where the cost per move is equal to 1, what is the total cost? 1 point
3. For DLS where the depth limit is set to 30, what is the total steps? 1 point
4. For GBFS, what are the expanding node values for the 20$^{th}$ to 25$^{th}$ move h(n)=? 2 points
5. For A*, what are the expanding node values for the 10$^{th}$ move g=?, h=?, f=? 3 points

2. 10 points

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

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

6. How many total steps for BFS? DFS? IDS? 3 points
7. For UCS where the cost per move is equal to 1, what is the total cost? 1 point
8. For DLS where the depth limit is set to 30, what is the total steps? 1 point
9. For GBFS, what are the expanding node values for the 20$^{th}$ to 25$^{th}$ move h(n)=? 2 points
10. For A*, what are the expanding node values for the 10$^{th}$ move g=?, h=?, f=? 3 points