Batch: B1	Roll No.: 16010121045
Experiment / assignment / tutorial No.	

Title: Implementation of uninformed search algorithm(BFS/DFS/DLS/IDS)

Objective: Comparison and analysis of uninformed search algorithms

Expected Outcome of Experiment:

Course Outcome	After successful completion of the course students should be able to
CO 2	Analyse and solve problems for goal based agent architecture (searching and planning algorithms).

Books/ Journals/ Websites referred:

1. **"Artificial Intelligence: a Modern Approach" by Russell and Norving, Pearson education Publications**
2. **"Artificial Intelligence" By Rich and knight, Tata Mcgraw Hill Publications**
3. **http://people.cs.pitt.edu/~milos/courses/cs2710/lectures/Class4.pdf**
4. **http://cs.williams.edu/~andrea/cs108/Lectures/InfSearch/infSearch.html**
5. **http://www.cs.mcgill.ca/~dprecup/courses/AI/Lectures/ai-lecture02.pdf**
   http://homepage.cs.uiowa.edu/~hzhang/c145/notes/04a-search.pdf
6. **http://wiki.answers.com/Q/Informed_search_techniques_and_uninformed_search_techniques**
7. **www.cs.swarthmore.edu/~eeaton/teaching/cs63/.../**UninformedSearch**.ppt**

Pre Lab/ Prior Concepts: Problem solving, state-space trees, problem formulation, goal based agent architecture

Historical Profile:

The AI researchers have come up many algorithms those operate on state space tree to give the result. Goal based agent architectures solve problems through searching or planning. Depending on availability of more information other than the problem statement decides if the solution can be obtained with uninformed search or informed search.

Its fact that not all search algorithms end up in giving the optimal solution. So, it states the need to have a better and methodological approach which guarantees optimal solution.

New Concepts to be learned: Uninformed (blind) search, iterative deepening, greedy best first search, A* search

---

Uninformed searching techniques:

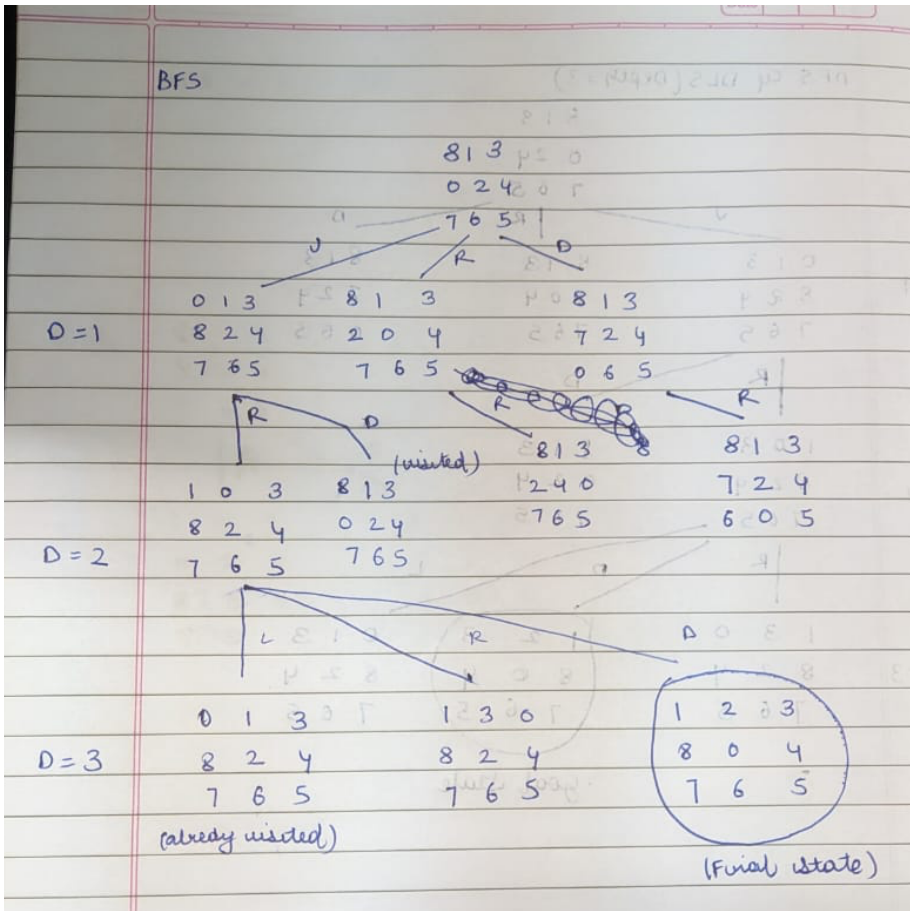
- Breadth first search
- Depth first search
- Iterative deepening search
- Depth limit search

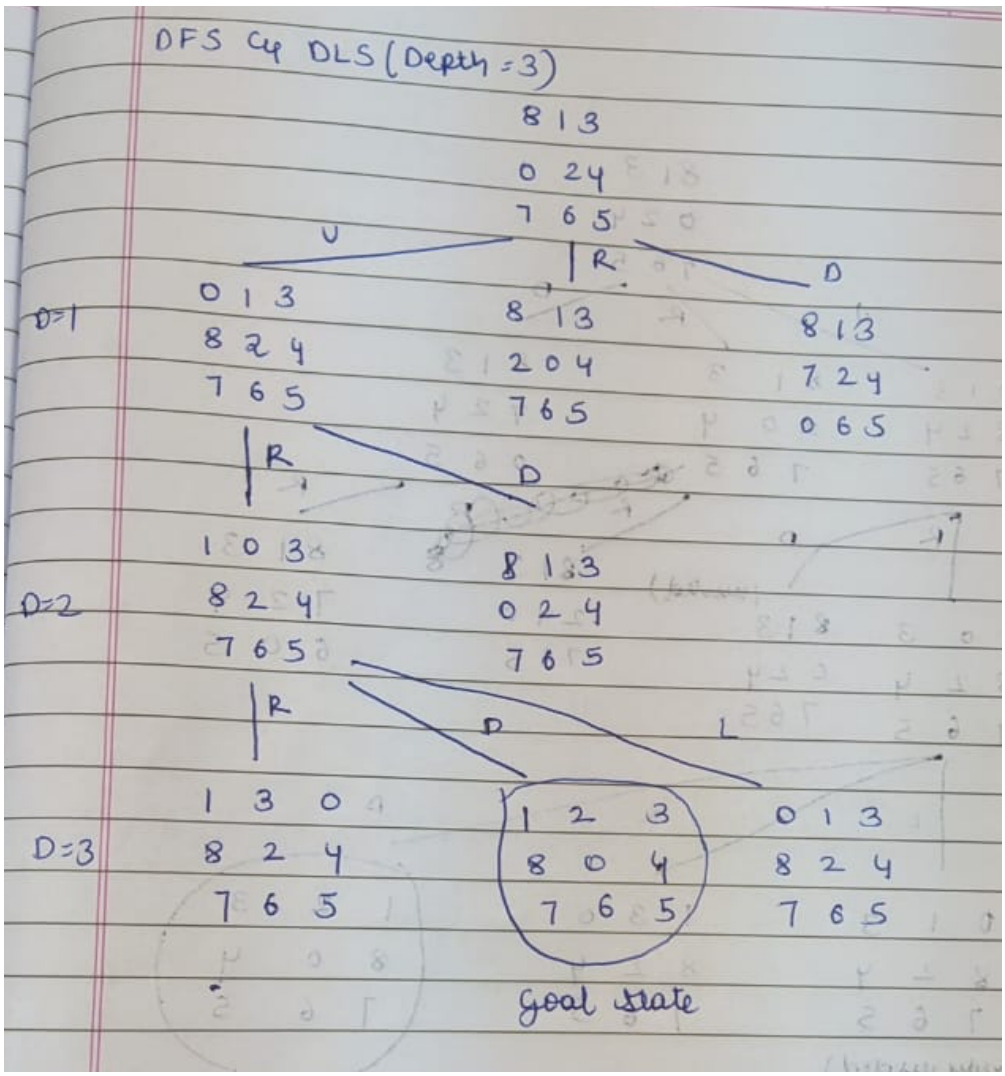
Chosen Problem statement: 8 Puzzle problem

The 8 Puzzle problem involves a 3x3 grid containing 8 numbered tiles and one blank tile. The objective is to rearrange the tiles from an initial state to a goal state using legal moves. The legal moves are swapping the blank tile with its adjacent (horizontal or vertical) numbered tile. The goal state is typically defined as the sorted arrangement of tiles, such as:

- **States**: Different configurations of the puzzle.
- **Initial State**: Any solvable configuration of the puzzle.
- **Transition Model**: Legal moves of swapping the blank tile with an adjacent numbered tile.
- **Actions**: Legal moves defined by the transition model.
- **Goal Test**: Check if the current state is the sorted arrangement of tiles.
- **Path Cost**: Each move has a uniform cost.

State-space tree :




DFS & DLS (Depth = 3)
D=1
D=2
D=3
U
R
D
L
Goal state

Code:
BFS

```
from collections import deque

def print_board(board):
    for row in board:
        print(row)

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

def is_valid_move(i, j):
    return 0 <= i < 3 and 0 <= j < 3

def swap_tiles(board, blank_pos, new_pos):
    i1, j1 = blank_pos
    i2, j2 = new_pos
    board[i1][j1], board[i2][j2] = board[i2][j2],
board[i1][j1]

def is_goal_state(board, goal_state):
    return board == goal_state

def get_neighbors(board):
    blank_pos = get_blank_position(board)
    neighbors = []

    moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    for move in moves:
        new_pos = (blank_pos[0] + move[0], blank_pos[1] +
move[1])
        if is_valid_move(*new_pos):
            new_board = [row.copy() for row in board]
```

```
            swap_tiles(new_board, blank_pos, new_pos)
            neighbors.append(new_board)

    return neighbors

def bfs_8_puzzle(initial_state, goal_state):
    visited = set()
    queue = deque([(initial_state, [])])

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

        if is_goal_state(current_state, goal_state):
            return path + [current_state]

        if tuple(map(tuple, current_state)) not in visited:
            visited.add(tuple(map(tuple, current_state)))
            neighbors = get_neighbors(current_state)

            for neighbor in neighbors:
                queue.append((neighbor, path +
[current_state]))

    return None

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

goal_state = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]
]
solution = bfs_8_puzzle(initial_state, goal_state)
```

```
if solution:
    print("Solution found:")
    for step in solution:
        print_board(step)
        print()
else:
    print("No solution found.")
```

```
Solution found:
[8, 1, 3]
[0, 2, 4]
[7, 6, 5]

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

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

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

DFS:

```
from collections import deque

def print_board(board):
    for row in board:
        print(row)

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

def is_valid_move(i, j):
    return 0 <= i < 3 and 0 <= j < 3

def swap_tiles(board, blank_pos, new_pos):
    i1, j1 = blank_pos
    i2, j2 = new_pos
    board[i1][j1], board[i2][j2] = board[i2][j2],
board[i1][j1]

def is_goal_state(board, goal_state):
    return board == goal_state

def get_neighbors(board):
    blank_pos = get_blank_position(board)
    neighbors = []

    moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    for move in moves:
        new_pos = (blank_pos[0] + move[0], blank_pos[1] +
move[1])
        if is_valid_move(*new_pos):
            new_board = [row.copy() for row in board]
            swap_tiles(new_board, blank_pos, new_pos)
            neighbors.append(new_board)
```

```
    return neighbors

def dfs_8_puzzle(initial_state, goal_state):
    visited = set()
    stack = [(initial_state, [])]

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

        if is_goal_state(current_state, goal_state):
            return path + [current_state]

        if tuple(map(tuple, current_state)) not in visited:
            visited.add(tuple(map(tuple, current_state)))
            neighbors = get_neighbors(current_state)

            for neighbor in neighbors:
                stack.append((neighbor, path +
[current_state]))

    return None

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

goal_state = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]
]
solution = dfs_8_puzzle(initial_state, goal_state)

if solution:
    print("Solution found:")
```

```
    for step in solution:
        print_board(step)
        print()
else:
    print("No solution found.")
```

```
Solution found:
[8, 1, 3]
[0, 2, 4]
[7, 6, 5]

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

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

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

DLS:

```
from collections import deque

def print_board(board):
    for row in board:
        print(row)

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

def is_valid_move(i, j):
    return 0 <= i < 3 and 0 <= j < 3

def swap_tiles(board, blank_pos, new_pos):
    i1, j1 = blank_pos
    i2, j2 = new_pos
    board[i1][j1], board[i2][j2] = board[i2][j2],
board[i1][j1]

def is_goal_state(board, goal_state):
    return board == goal_state

def get_neighbors(board):
    blank_pos = get_blank_position(board)
    neighbors = []

    moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    for move in moves:
        new_pos = (blank_pos[0] + move[0], blank_pos[1] +
move[1])
        if is_valid_move(*new_pos):
            new_board = [row.copy() for row in board]
            swap_tiles(new_board, blank_pos, new_pos)
            neighbors.append(new_board)
```

```
    return neighbors

def dls_8_puzzle(initial_state, goal_state, max_depth):
    visited = set()
    stack = [(initial_state, [], 0)]

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

        if depth > max_depth:
            continue

        if is_goal_state(current_state, goal_state):
            return path + [current_state]

        if tuple(map(tuple, current_state)) not in visited:
            visited.add(tuple(map(tuple, current_state)))
            neighbors = get_neighbors(current_state)

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

    return None

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

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

```
solution = dls_8_puzzle(initial_state, goal_state,3)

if solution:
    print("Solution found:")
    for step in solution:
        print_board(step)
        print()
else:
    print("No solution found.")
```

```
Solution found:
[8, 1, 3]
[0, 2, 4]
[7, 6, 5]

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

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

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

Comparison of performance of uninformed Algorithm:

Given the characteristics of the 8 Puzzle problem, which has a relatively small state space, BFS is generally the preferred choice because:

- The state space of the 8 Puzzle problem is not excessively large, so the memory-intensive nature of BFS is less of a concern.
- BFS guarantees the optimal solution, which is desirable for solving puzzles.

- BFS will explore the shallowest nodes first, which can be advantageous in situations where the solution is closer to the initial state.

Therefore, in this specific case, BFS is considered the best algorithm. However, if memory constraints were a significant concern or if the state space were much larger, DFS or DLS might be more practical despite their lack of optimality.

Post Lab Objective questions

1. **Which search algorithm imposes a fixed depth limit on nodes?**
   a. Depth-limited search
   b. Depth-first search
   c. Iterative Deepening search
   d. Only (a) and (b)
   e. Only (a), (b) and (c).

Answer: a

2. **Optimality of BFS is**
   a. When all step costs are equal
   b. When all step costs are unequal
   c. When there is less number of nodes
   d. Both a & c

Answer: a

Post Lab Subjective Questions:

1. **Mention the criteria for the evaluation of search Algorithm.**

Criteria for the evaluation of search algorithms:

- Completeness: Does the algorithm guarantee finding a solution if one exists?
- Optimality: Does the algorithm find the optimal solution, i.e., the one with the lowest cost?
- Time complexity: How long does the algorithm take to find a solution?
- Space complexity: How much memory does the algorithm require?
- Admissibility: Does the algorithm always underestimate the cost to reach the goal?
- Heuristic accuracy: How accurate are the heuristic estimates used in informed search algorithms?
- Any domain-specific considerations: Certain problems may have additional criteria specific to their characteristics.

2. **State the properties of BFS,DFS, DLS and IDS**

Properties of BFS, DFS, DLS, and IDS:

- **Breadth-First Search (BFS)**:
  - Completeness: Guaranteed to find a solution if one exists.
  - Optimality: Finds the optimal solution when step costs are equal.
  - Time complexity: □(□□)$O(bd)$, where □b is the branching factor and □d is the depth of the shallowest goal node.
  - Space complexity: □(□□)$O(bd)$ due to storing all nodes at each level.
- **Depth-First Search (DFS)**:
  - Completeness: Not guaranteed to find a solution; can get stuck in infinite loops.
  - Optimality: Not guaranteed to find the optimal solution.
  - Time complexity: □(□□)$O(bm)$, where □b is the branching factor and □m is the maximum depth of the search tree.
  - Space complexity: □(□□)$O(bm)$ due to storing a single path from the root to a leaf node.
- **Depth-Limited Search (DLS)**:
  - Similar to DFS but with a depth limit imposed.
  - Completeness: Depends on the depth limit; may not find a solution if the limit is too shallow.
  - Optimality: Not guaranteed to find the optimal solution.
- **Iterative Deepening Search (IDS)**:
  - A variant of DFS with incrementally increasing depth limits.
  - Combines the benefits of BFS and DFS.
  - Completeness: Guaranteed to find a solution if one exists.
  - Optimality: Optimal when step costs are equal.
  - Time complexity: □(□□)$O(bd)$, similar to BFS, but with less memory overhead.

3. **Explain why BFS is worst approach when the branching factor and solution depth in state-space tree is large (value =10 or more)**

Explanation of why BFS is the worst approach when the branching factor and solution depth in the state-space tree are large:

- BFS explores all nodes at each depth level before moving to the next level.
- With a large branching factor (□b) and solution depth (□d), the number of nodes at each level can grow exponentially.
- BFS needs to store all these nodes in memory, leading to a significant memory overhead.

- The memory consumption of BFS becomes prohibitively large as the branching factor and solution depth increase.
- Therefore, BFS becomes impractical and inefficient for large state-space trees with high branching factors and depths.