Lab Report: N–Puzzle Problem Solving using Search Algorithms

Kanchan Joshi

October 2025

Objective

To implement and understand the working principle of the **N–Puzzle Problem** using different search algorithms such as **Breadth-First Search (BFS)**, **Depth-First Search (DFS)**, and **A* Search**. The goal is to find the sequence of moves that leads from an initial puzzle configuration to the goal configuration.

Theory

2.1 Introduction

The N–Puzzle problem is a classic problem in Artificial Intelligence (AI) and search-based problem solving. It consists of an $N \times N$ board containing $N^2 - 1$ numbered tiles and one blank space. The objective is to move the tiles around until they are arranged in a specific goal configuration.For example, the 8-puzzle is a 3×3 version of the problem with 8 numbered tiles and one blank space.

$$\text{Initial State} \Rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 4 & & 6 \\ 7 & 5 & 8 \end{bmatrix} \quad \text{Goal State} \Rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & \end{bmatrix}$$

The N-puzzle problem specification includes the following formal components:

- **Initial Configuration**: A grid arrangement containing numbered tiles and one vacant position
- **Goal Configuration**: Tiles arranged in ascending numerical order with the empty space in the final position
- **Available Actions**: Sliding adjacent tiles into the empty space (up, down, left, right movements)
- **Solution Cost**: Total number of moves required to achieve the goal configuration

2.2 State Space Representation

Each configuration of the board represents a **state**. Possible actions are movements of the blank tile:

- Move Up
- Move Down

- Move Left
- Move Right

2.3 Uninformed Search Algorithms

Uninformed search methodologies (alternatively termed blind search approaches) operate without utilizing domain-specific knowledge regarding the problem structure. These algorithms systematically examine the search space without directional guidance toward the goal state. The following uninformed search algorithms represent fundamental approaches to systematic state space exploration:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Depth-Limited Search (DLS)
- Iterative Deepening Search (IDS)
- Uniform Cost Search (UCS)
- Bidirectional Search

2.4 Search Algorithms Used

Search algorithms are fundamental in Artificial Intelligence for exploring problem spaces and finding solutions. They are used to traverse or search through trees or graphs to reach a goal state from an initial state. In the context of the N-Puzzle problem, these algorithms help in finding the sequence of moves that transforms the initial configuration into the goal configuration efficiently.

1. Breadth-First Search (BFS)

- Breadth-First Search explores all nodes at the present depth before moving to the next level.
- It uses a queue (FIFO) data structure.
- BFS guarantees the shortest path in terms of number of steps when all step costs are equal.

Characteristics:

- Completeness: Yes
- Optimality: Yes (for uniform step cost)
- Time Complexity: $O(b^d)$
- Space Complexity: $O(b^d)$

2. Depth-First Search (DFS)

- Depth-First Search explores as far as possible along each branch before backtracking.
- It uses a stack (LIFO) data structure, often implemented recursively.
- DFS is memory efficient but can get stuck in deep or infinite paths.

Characteristics:

- Completeness: No (for infinite-depth spaces)
- Optimality: No
- Time Complexity: $O(b^m)$, where m is the maximum depth of the search tree.
- Space Complexity: $O(b \times m)$

3. Depth-Limited Search (DLS)

- DLS is a variant of DFS that limits the depth of search to a predefined value l.
- It prevents infinite recursion and is useful when the depth of the goal is known approximately.

Characteristics:

- Completeness: Not complete if the goal is beyond the depth limit.
- Optimality: Not optimal.
- Time Complexity: $O(b^l)$
- Space Complexity: $O(b \times l)$

4. Iterative Deepening Search (IDS)

- IDS repeatedly applies DLS with increasing depth limits ($l = 0, 1, 2, \ldots$).
- It combines the space efficiency of DFS and the completeness of BFS.

Characteristics:

- Completeness: Yes (if step cost > 0)
- Optimality: Yes (for uniform step cost)
- Time Complexity: $O(b^d)$
- Space Complexity: $O(b \times d)$

5. Uniform Cost Search (UCS)

- UCS expands the node with the lowest cumulative path cost $g(n)$ first.
- It uses a priority queue ordered by path cost.
- UCS is equivalent to Dijkstra's algorithm and finds the least-cost path.

Characteristics:

- Completeness: Yes (if step costs are positive)
- Optimality: Yes
- Time Complexity: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$
- Space Complexity: $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

6. Bidirectional Search

- Bidirectional Search runs two searches simultaneously — one forward from the start node and one backward from the goal node.
- The search stops when both frontiers meet.
- This technique reduces the effective search depth to half, improving efficiency.

Characteristics:

- Completeness: Yes (if both searches are complete)
- Optimality: Yes (if both use BFS)
- Time Complexity: $O(b^{d/2})$
- Space Complexity: $O(b^{d/2})$

2.5 Informed Search Algorithms

Informed search methodologies (also referred to as heuristic search approaches) leverage domain-specific knowledge or heuristics to guide the exploration of the search space. By estimating the cost or distance to the goal, these algorithms prioritize more promising paths, aiming to find solutions more efficiently compared to uninformed methods. Common informed search algorithms include:

1. A* Search Algorithm

- A* is an informed search algorithm that uses both path cost and heuristic information to guide the search.
- It selects the node with the minimum estimated total cost:

$$f(n) = g(n) + h(n)$$

  where:

  - $g(n)$ = cost from the start node to the current node
  - $h(n)$ = estimated cost from the current node to the goal (heuristic)
- When $h(n)$ is admissible (never overestimates), A* guarantees the optimal solution.

Characteristics:

- Completeness: Yes (if $h(n)$ is admissible)
- Optimality: Yes (for admissible and consistent $h(n)$)
- Time Complexity: Exponential in the worst case
- Space Complexity: Exponential (stores all generated nodes)

These algorithms generally achieve faster solution discovery and can reduce the number of explored states significantly, though their optimality depends on the admissibility and consistency of the heuristics employed.

2.6 Breadth-First Search (BFS) Algorithm for N–Puzzle :

1. Start with the initial state of the puzzle.
2. Initialize a queue (FIFO) called `frontier` with the initial state.
3. Initialize an empty set called `explored` to keep track of visited states.
4. While the `frontier` is not empty:
   (a) Remove the first state from the `frontier`.
   (b) If the state is the goal state, stop and return the solution path.
   (c) Otherwise, add the state to `explored`.
   (d) Generate all valid successor states (by moving the empty tile up, down, left, or right).
   (e) For each successor, if it is not in `explored` or `frontier`, add it to the end of `frontier`.
5. Repeat until the goal state is found.

2.7 Key Characteristics

- Explores states level by level (shallowest nodes first).
- Guarantees the shortest path in terms of number of moves for unweighted problems.
- Memory intensive for large puzzles due to storing all explored states.

Python Implementation

Listing 1: N–Puzzle Solver using BFS

```python
import matplotlib.pyplot as plt
import time
from collections import deque
import copy

# N-puzzle size
N = 3

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

class PuzzleState:
    def __init__(self, state, parent=None, depth=0):
        self.state = state
        self.parent = parent
        self.depth = depth

```

```python
def find_empty(state):
    for i in range(N):
        for j in range(N):
            if state[i][j] == 0:
                return i, j

def get_successors(node):
    successors = []
    row, col = find_empty(node.state)
    moves = [(-1,0), (1,0), (0,-1), (0,1)]  # up, down, left, right

    for dr, dc in moves:
        new_r, new_c = row+dr, col+dc
        if 0 <= new_r < N and 0 <= new_c < N:
            new_state = copy.deepcopy(node.state)
            new_state[row][col], new_state[new_r][new_c] = new_state[new_r][new_c], new_state[row][col]
            successors.append(PuzzleState(new_state, node, node.depth+1))
    return successors

def reconstruct_path(goal_node):
    path = []
    current = goal_node
    while current:
        path.append(current.state)
        current = current.parent
    return path[::-1]

def display_state(state):
    plt.imshow(state, cmap='Pastel1', interpolation='nearest')
    for i in range(N):
        for j in range(N):
            if state[i][j] != 0:
                plt.text(j, i, str(state[i][j]), ha='center', va='center', fontsize=20)
    plt.axis('off')
    plt.show(block=False)
    plt.pause(0.5)
    plt.clf()

def bfs(initial_state):
    frontier = deque([PuzzleState(initial_state)])
    explored = set()
```

```python

    while frontier:
        node = frontier.popleft()
        if node.state == goal_state:
            return reconstruct_path(node)
        explored.add(tuple(map(tuple, node.state)))
        for succ in get_successors(node):
            if tuple(map(tuple, succ.state)) not in explored and all(tuple(map(tuple, f.state)) != tuple(map(tuple, succ.state)) for f in frontier):
                frontier.append(succ)
    return None

# Example initial state
initial_state = [[1,2,3],
                 [4,0,6],
                 [7,5,8]]

solution_path = bfs(initial_state)

# Display the solution dynamically
for step, state in enumerate(solution_path):
    print(f"Step {step}:")
    display_state(state)

# Save final solution as PNG
plt.imshow(solution_path[-1], cmap='Pastel1', interpolation='nearest')
for i in range(N):
    for j in range(N):
        if solution_path[-1][i][j] != 0:
            plt.text(j, i, str(solution_path[-1][i][j]), ha='center', va='center', fontsize=20)
plt.axis('off')
plt.savefig("full_solution.png")
print("Final solution saved as full_solution.png")
```

3.1 Depth-Limited Search (DLS) for N–Puzzle

DLS(initial_state, L)

1. Start with the initial state of the puzzle and define the **depth limit (**L**)**.
2. Initialize a **Stack (LIFO)** called `stack` with the initial state and a depth counter: $(\text{initial\_state}, \text{path}, 0)$.
3. Initialize an empty set called `explored` to keep track of visited states.
4. While the `stack` is not empty:

(a) Remove the **last** item from the `stack`: $(\text{state}, \text{path}, \text{depth}) \leftarrow$ `stack`.pop().
(b) If the state is the **goal state**, stop and return the path.
(c) Otherwise, add the state to `explored`.
(d) **Depth Limit Check:** If depth $< L$:
    i. Generate all valid successor states (by moving the empty tile).
    ii. For each successor (succ):
        A. If succ is **not** in `explored`:
        B. Add succ, the new path (path + [succ]), and the incremented depth (depth + 1) to the **top** of the `stack`.

5. If the loop terminates without finding the goal, return `Failure` (no solution found within the limit).

3.2 Output

3.3 Key Characteristics of Depth-Limited Search (DLS)

- **Search Strategy:** Explores states by going as **deep** as possible along a single path, but stops and backtracks when the current depth reaches the predefined **limit (L)**.
- **Completeness: Incomplete** if the shallowest goal is at a depth greater than L. DLS cannot guarantee finding a solution.
- **Optimality: Non-optimal.** If a solution is found, it is the first one encountered in the depth-first traversal and is not guaranteed to be the shortest path.
- **Space Complexity:** $O(b \cdot L)$, making it highly **memory efficient** (b is the branching factor).

Python Implementation (DLS)

Listing 2: N–Puzzle Solver using Depth-Limited Search (DLS)

```python
import copy

# N-puzzle size (e.g., N=3 for the 8-puzzle)
N = 3

# Goal state (defined globally for efficiency)
goal_state = [[1,2,3],
              [4,5,6],
              [7,8,0]]
GOAL_TUPLE = tuple(map(tuple, goal_state))

# Helper function to find the empty tile
def find_empty(state):
    """Finds the coordinates (row, column) of the empty tile (0)."""
    for i in range(N):
        for j in range(N):
```

```python
            if state[i][j] == 0:
                return i, j
    return -1, -1

# Helper function to generate successors
def get_successors(state):
    """Generates all possible next states (successors) from the current state."""
    successors = []
    row, col = find_empty(state)
    # Moves: up, down, left, right
    moves = [(-1,0), (1,0), (0,-1), (0,1)]

    for dr, dc in moves:
        new_r, new_c = row + dr, col + dc

        if 0 <= new_r < N and 0 <= new_c < N:
            new_state = copy.deepcopy(state)
            new_state[row][col], new_state[new_r][new_c] = new_state[new_r][new_c], new_state[row][col]
            successors.append(new_state)
    return successors

# Helper function for hashable state conversion
def list_to_tuple(state):
    """Converts a list-of-lists state to a hashable tuple-of-tuples."""
    return tuple(map(tuple, state))

# Depth-Limited Search implementation
def dls(initial_state, limit):
    """
    Solves the N-Puzzle using Depth-Limited Search.

    The search will not explore paths longer than 'limit'.
    """
    initial_tuple = list_to_tuple(initial_state)

    if initial_tuple == GOAL_TUPLE:
        return [initial_state]

    # Stack stores (state_as_list, path_to_state, current_depth)
    stack = [(initial_state, [initial_state], 0)]

```

```
    # Explored set is necessary for a graph search to prevent cycles
    # For DLS, we must re-explore states at different depths if the path to them is unique
    # Here, we keep the simpler cycle detection based on unique states
    explored = {initial_tuple}

    while stack:
        # LIFO operation for DFS
        state, path, depth = stack.pop()

        # KEY DLS CHECK: Stop if the limit has been reached
        if depth >= limit:
            continue

        # Explore successors
        for succ in get_successors(state):
            succ_tuple = list_to_tuple(succ)

            # Goal Check
            if succ_tuple == GOAL_TUPLE:
                return path + [succ]

            # Cycle Check
            if succ_tuple not in explored:
                explored.add(succ_tuple)

                # Add to stack with incremented depth
                stack.append((succ, path + [succ], depth + 1))

    return None # Return None if no solution found within the limit

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

# Define the maximum search depth
SEARCH_LIMIT = 5

print(f"--- Starting Depth-Limited Search with Limit = {SEARCH_LIMIT} ---")
solution_path = dls(initial_state, SEARCH_LIMIT)
```

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

4.1 Output

Output

```
--- Starting Depth-Limited Search with Limit = 5 ---
Solution Found in 2 moves (within limit):
Step 0:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

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

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


=== Code Execution Successful ===
```

4.2 Iterative Deepening Search (IDS) Algorithm for N–Puzzle :

1. Start with the initial state of the puzzle.
2. Set a depth limit `limit` starting from 0.
3. Perform Depth-Limited Search (DLS) with the current `limit`.

4. If the goal state is found, stop and return the solution path.
5. Otherwise, increment the `limit` by 1 and repeat DLS.

4.3 Key Characteristics

- Combines benefits of BFS (completeness) and DFS (low memory usage).
- Explores nodes depth by depth, gradually increasing the depth limit.
- Memory efficient compared to BFS for large puzzles.

Python Implementation

Listing 3: N–Puzzle Solver using Iterative Deepening Search (IDS)

```python
import copy

print("=== N-Puzzle Solver using Iterative Deepening Search (IDS) ===")

# N-puzzle size
N = 3

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

def find_empty(state):
    for i in range(N):
        for j in range(N):
            if state[i][j] == 0:
                return i, j

def get_successors(state):
    successors = []
    row, col = find_empty(state)
    moves = [(-1,0), (1,0), (0,-1), (0,1)]  # up, down, left, right
    for dr, dc in moves:
        new_r, new_c = row + dr, col + dc
        if 0 <= new_r < N and 0 <= new_c < N:
            new_state = copy.deepcopy(state)
            new_state[row][col], new_state[new_r][new_c] = new_state[new_r][new_c], new_state[row][col]
            successors.append(new_state)
```

```python
    return successors

def dls(state, limit, path, explored):
    if state == goal_state:
        return path
    if limit <= 0:
        return None
    explored.add(tuple(map(tuple, state)))
    for succ in get_successors(state):
        if tuple(map(tuple, succ)) not in explored:
            result = dls(succ, limit-1, path + [succ], explored)
            if result is not None:
                return result
    explored.remove(tuple(map(tuple, state)))
    return None

def ids(initial_state, max_depth=10):
    for depth in range(max_depth):
        print(f"Trying depth limit: {depth}")
        explored = set()
        result = dls(initial_state, depth, [initial_state], explored)
        if result is not None:
            print(f"Solution found at depth {depth}!")
            return result
    return None

# Example initial state (solvable in 2 moves)
initial_state = [[1,2,3],
                 [4,5,6],
                 [0,7,8]]

solution_path = ids(initial_state)

# Print the solution path
for step, state in enumerate(solution_path):
    print(f"\nStep {step}:")
    for row in state:
        print(row)
```

5.1 Output

```
        print(row)
```

```
=== N-Puzzle Solver using Iterative Deepening Search (IDS) ===
Trying depth limit: 0
Trying depth limit: 1
Trying depth limit: 2
Solution found at depth 2!

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

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

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

Bidirectional Search (BDS) Algorithm for N–Puzzle:

1. Start with the initial state and the goal state of the puzzle.
2. Initialize two frontiers (queues): one from the start state and one from the goal state.
3. Initialize two sets of explored states corresponding to each frontier.
4. While both frontiers are not empty:
   (a) Expand one node from the start frontier:
      - Generate all valid successors.
      - If any successor is in the goal frontier's explored set, a meeting point is found. Stop and reconstruct the path.
      - Otherwise, add unexplored successors to the start frontier.

   (b) Expand one node from the goal frontier:
      - Generate all valid successors.
      - If any successor is in the start frontier's explored set, a meeting point is found. Stop and reconstruct the path.
      - Otherwise, add unexplored successors to the goal frontier.
5. Reconstruct the full path from the start to goal through the meeting point.

5.2 Key Characteristics

- Explores the search space from both start and goal simultaneously.
- Reduces the search depth compared to unidirectional BFS.

- Memory usage can be high due to storing explored nodes from both directions.
- Can significantly reduce time for large puzzles with short solution paths.

Python Implementation

Listing 4: N–Puzzle Solver using Bidirectional Search

```
import copy
from collections import deque

print("=== N-Puzzle Solver using Bi-directional Search ===")

# N-puzzle size
N = 3

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

# Moves with directions
moves = [(-1,0,"Up"), (1,0,"Down"), (0,-1,"Left"), (0,1,"Right")]

def find_empty(state):
    for i in range(N):
        for j in range(N):
            if state[i][j] == 0:
                return i, j

def get_successors(state):
    successors = []
    row, col = find_empty(state)
    for dr, dc, move in moves:
        new_r, new_c = row + dr, col + dc
        if 0 <= new_r < N and 0 <= new_c < N:
            new_state = copy.deepcopy(state)
            new_state[row][col], new_state[new_r][new_c] = new_state[new_r][new_c], new_state[row][col]
            successors.append((new_state, move))
    return successors

def reconstruct_path(meet_state, parents_start, parents_goal):
    path = []
    moves_path = []
```

```

    # From start to meeting point
    state = tuple(map(tuple, meet_state))
    while state:
        node, parent, move = parents_start[state]
        path.append(node)
        if move:
            moves_path.append(move)
        state = parent
    path = path[::-1]
    moves_path = moves_path[::-1]

    # From meeting point to goal
    state = tuple(map(tuple, meet_state))
    goal_moves = []
    goal_path = []
    while state:
        node, parent, move = parents_goal[state]
        goal_path.append(node)
        if move:
            rev_move = {"Up":"Down","Down":"Up","Left":"Right",
                "Right":"Left"}[move]
            goal_moves.append(rev_move)
        state = parent
    goal_path = goal_path[1:][::-1]  # skip meeting point
    goal_moves = goal_moves[::-1]

    full_path = path + goal_path
    full_moves = moves_path + goal_moves
    return full_path, full_moves

def bidirectional_search(initial_state):
    frontier_start = deque([initial_state])
    frontier_goal = deque([goal_state])

    parents_start = {tuple(map(tuple, initial_state)): (
       initial_state, None, None)}
    parents_goal = {tuple(map(tuple, goal_state)): (goal_state,
        None, None)}

    explored_start = set()
    explored_goal = set()

    while frontier_start and frontier_goal:
        # Expand from start
```

```
        current_start = frontier_start.popleft()
        explored_start.add(tuple(map(tuple, current_start)))
        for succ, move in get_successors(current_start):
            t_succ = tuple(map(tuple, succ))
            if t_succ not in parents_start:
                parents_start[t_succ] = (succ, tuple(map(tuple, current_start)), move)
                frontier_start.append(succ)
            if t_succ in explored_goal:
                print("Solution found by Bidirectional Search!")
                return reconstruct_path(succ, parents_start, parents_goal)

        # Expand from goal
        current_goal = frontier_goal.popleft()
        explored_goal.add(tuple(map(tuple, current_goal)))
        for succ, move in get_successors(current_goal):
            t_succ = tuple(map(tuple, succ))
            if t_succ not in parents_goal:
                parents_goal[t_succ] = (succ, tuple(map(tuple, current_goal)), move)
                frontier_goal.append(succ)
            if t_succ in explored_start:
                print("Solution found by Bidirectional Search!")
                return reconstruct_path(succ, parents_start, parents_goal)
    return None, None

# Example initial state (solvable in 2-5 steps)
initial_state = [[1,2,3],
                 [4,0,6],
                 [7,5,8]]

solution_path, moves_taken = bidirectional_search(initial_state)

# Print solution path with steps and moves
if solution_path:
    for step, (state, move) in enumerate(zip(solution_path, ["Start"] + moves_taken)):
        print(f"Step{step}: Move -> {move}")
        for row in state:
            print(row)
```

```python
        print()
    print("Puzzle matched the goal state. Solution Found!")
else:
    print("No solution found.")
```

6.1 Output

```
=== N-Puzzle Solver using Bi-directional Search ===
Step 0: Move -> Start
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Step 1: Move -> Down
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Step 2: Move -> Right
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Puzzle matched the goal state. ✅ Solution Found!
```

6.2 Uniform Cost Search (UCS) Algorithm for N–Puzzle :

1. Start with the initial state of the puzzle.
2. Initialize a priority queue (min-heap) called `frontier` with the initial state, where priority is the path cost.
3. Initialize an empty set called `explored` to keep track of visited states.
4. While the `frontier` is not empty:
   (a) Remove the state with the lowest path cost from the `frontier`.
   (b) If the state is the goal state, stop and return the solution path.
   (c) Otherwise, add the state to `explored`.
   (d) Generate all valid successor states (by moving the empty tile up, down, left, or right).
   (e) For each successor, calculate its path cost. If it is not in `explored` or has a lower cost than previously recorded, add it to `frontier`.
5. Repeat until the goal state is found.

6.3 Key Characteristics

- Expands nodes based on the lowest cumulative cost from the start.
- Guarantees an optimal solution for uniform step costs.
- Similar to BFS when all moves have the same cost.
- Can be memory intensive for large state spaces.

Python Implementation

Listing 5: N–Puzzle Solver using Uniform Cost Search

```
import copy
from heapq import heappush, heappop

print("=== N-Puzzle Solver using Uniform Cost Search (UCS) ===")

# N-puzzle size
N = 3

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

# Moves with directions
moves = [(-1,0,"Up"), (1,0,"Down"), (0,-1,"Left"), (0,1,"Right")]

def find_empty(state):
    for i in range(N):
        for j in range(N):
            if state[i][j] == 0:
                return i, j

def get_successors(state):
    successors = []
    row, col = find_empty(state)
    for dr, dc, move in moves:
        new_r, new_c = row + dr, col + dc
        if 0 <= new_r < N and 0 <= new_c < N:
            new_state = copy.deepcopy(state)
            new_state[row][col], new_state[new_r][new_c] = new_state[new_r][new_c], new_state[row][col]
            successors.append((new_state, move))
    return successors

def reconstruct_path(goal_node, parents):
    path = []
    moves_path = []
    state = tuple(map(tuple, goal_node))
    while state:
```

```python
        node, parent, move = parents[state]
        path.append(node)
        if move:
            moves_path.append(move)
        state = parent
    return path[::-1], moves_path[::-1]

def ucs(initial_state):
    frontier = []
    heappush(frontier, (0, initial_state))  # (cost, state)
    parents = {tuple(map(tuple, initial_state)): (initial_state, None, None)}
    explored = set()

    while frontier:
        cost, state = heappop(frontier)
        t_state = tuple(map(tuple, state))

        if state == goal_state:
            print("Solution found by UCS!")
            return reconstruct_path(state, parents)

        explored.add(t_state)

        for succ, move in get_successors(state):
            t_succ = tuple(map(tuple, succ))
            new_cost = cost + 1  # each move has cost 1
            if t_succ not in explored or t_succ not in [tuple(map(tuple, s[1])) for s in frontier]:
                parents[t_succ] = (succ, t_state, move)
                heappush(frontier, (new_cost, succ))
    return None, None

# Example initial state (solvable in 2-5 steps)
initial_state = [[1,2,3],
                 [4,0,6],
                 [7,5,8]]

solution_path, moves_taken = ucs(initial_state)

# Print solution path with steps and moves
if solution_path:
    for step, (state, move) in enumerate(zip(solution_path, ["Start"] + moves_taken)):
        print(f"Step {step}: Move -> {move}")
```

```python
        for row in state:
            print(row)
        print()
    print("Puzzle matched the goal state. Solution Found!")
else:
    print("No solution found.")
```

```
TERMINAL
PS C:\Users\ASUS> python -u "c:\Users\ASUS\Downloads\Untitled-1.py"
=== N-Puzzle Solver using Uniform Cost Search (UCS) ===
Solution found by UCS!
Step 0: Move -> Start
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Step 1: Move -> Down
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Step 2: Move -> Right
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Puzzle matched the goal state. ✅ Solution Found!
PS C:\Users\ASUS>
```

7.1 A* Search Algorithm for N–Puzzle:

1. Start with the initial state of the puzzle.
2. Initialize a priority queue called `frontier` with the initial state. Each state is prioritized based on $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach the state and $h(n)$ is the heuristic estimate to the goal.
3. Initialize an empty set called `explored` to keep track of visited states.
4. While the `frontier` is not empty:
    (a) Remove the state with the lowest $f(n)$ from the `frontier`.
    (b) If the state is the goal state, stop and return the solution path.
    (c) Otherwise, add the state to `explored`.
    (d) Generate all valid successor states (by moving the empty tile up, down, left, or right).
    (e) For each successor, if it is not in `explored`, compute $f(n) = g(n) + h(n)$ and add it to the `frontier`.
5. Repeat until the goal state is found.

7.2 Heuristic Function (Manhattan Distance)

$$h(n) = \sum_{i=1}^{N^2-1} \left(|x_i - x_i^*| + |y_i - y_i^*|\right)$$

where (x_i, y_i) is the current position of tile i, and (x_i^*, y_i^*) is its goal position.

7.3 Key Characteristics

- Uses a heuristic to guide the search towards the goal efficiently.
- Guarantees the shortest path if the heuristic is admissible (never overestimates).
- More memory-efficient than BFS for large search spaces if a good heuristic is used.
- Typically faster than uninformed search algorithms like BFS or DFS.

Python Implementation

Listing 6: N–Puzzle Solver using A* Search

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

# N-puzzle size
N = 3

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

class PuzzleState:
    def __init__(self, state, parent=None, depth=0, cost=0):
        self.state = state
        self.parent = parent
        self.depth = depth
        self.cost = cost  # f(n) = g(n) + h(n)

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

def manhattan_distance(state):
    distance = 0
    for i in range(N):
        for j in range(N):
            val = state[i][j]
            if val != 0:
                goal_row = (val-1) // N
                goal_col = (val-1) % N
                distance += abs(i - goal_row) + abs(j -
                    goal_col)
    return distance

```

```python
def find_empty(state):
    for i in range(N):
        for j in range(N):
            if state[i][j] == 0:
                return i, j

def get_successors(node):
    successors = []
    row, col = find_empty(node.state)
    moves = [(-1,0), (1,0), (0,-1), (0,1)]  # up, down, left, right

    for dr, dc in moves:
        new_r, new_c = row + dr, col + dc
        if 0 <= new_r < N and 0 <= new_c < N:
            new_state = copy.deepcopy(node.state)
            new_state[row][col], new_state[new_r][new_c] = new_state[new_r][new_c], new_state[row][col]
            g = node.depth + 1
            h = manhattan_distance(new_state)
            successors.append(PuzzleState(new_state, node, g, g+h))
    return successors

def reconstruct_path(goal_node):
    path = []
    current = goal_node
    while current:
        path.append(current.state)
        current = current.parent
    return path[::-1]

def a_star(initial_state):
    start_node = PuzzleState(initial_state, depth=0, cost=manhattan_distance(initial_state))
    frontier = []
    heappush(frontier, start_node)
    explored = set()

    while frontier:
        node = heappop(frontier)
        if node.state == goal_state:
            return reconstruct_path(node)
        explored.add(tuple(map(tuple, node.state)))
        for succ in get_successors(node):
```

```python
            if tuple(map(tuple, succ.state)) not in explored:
                heappush(frontier, succ)
    return None

# Example initial state
initial_state = [[1,2,3],
                 [4,0,6],
                 [7,5,8]]

solution_path = a_star(initial_state)

# Print the solution path
for step, state in enumerate(solution_path):
    print(f"Step {step}:")
    for row in state:
        print(row)
    print()
```

8.1 Output

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

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

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

- The A* algorithm successfully found the optimal sequence of moves to solve the 8-puzzle problem.
- Using Manhattan distance as the heuristic ensured faster convergence and fewer node expansions compared to uninformed searches.

Result and Discussion

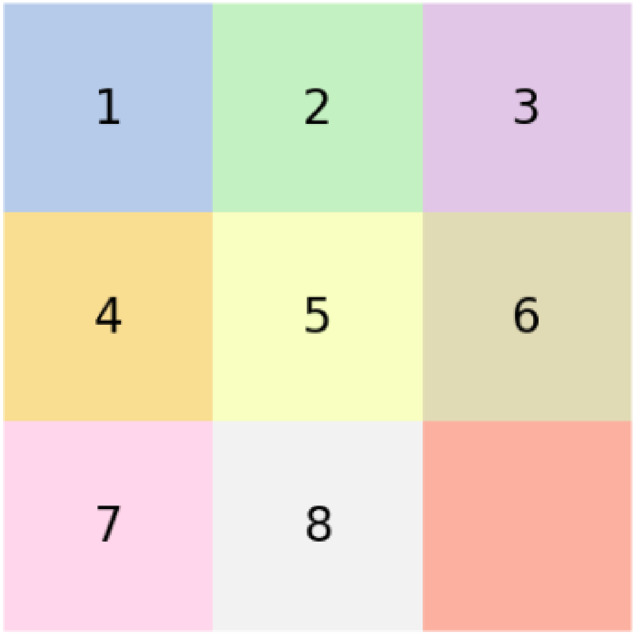
Step 0:

1	2	3
4		6
7	5	8

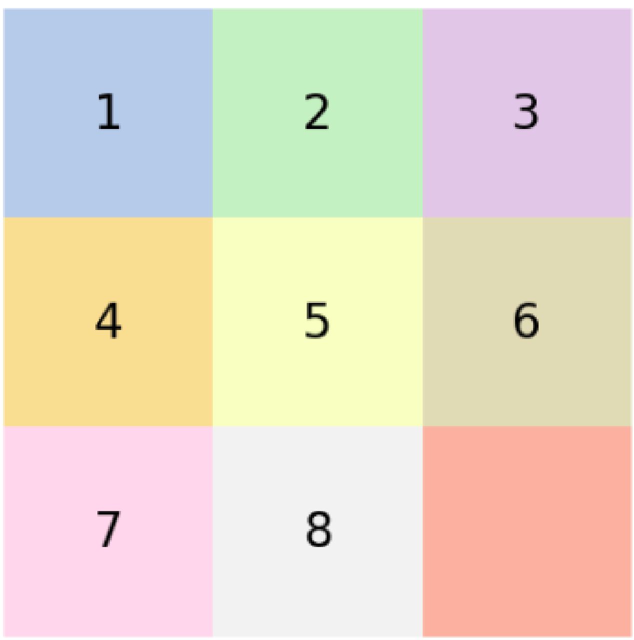
Step 1:

1	2	3
4	5	6
7		8

```
Step 2:
```




```
Final solution saved as full_solution.png
```



N-Puzzle Solving Using Different Algorithms

Performance Analysis and Conclusions

10.1 Comparative Performance of Search Algorithms

Various search strategies exhibit unique trade-offs in terms of time, memory, completeness, and optimality. The following table summarizes these characteristics:

Algorithm Type	Time Complexity	Space Complexity	Optimality
$O(b^d)$	$O(b^d)$	Yes	Yes Depth-First Search
$O(bm)$	No	No* Iterative Deepening Search (IDS)	$O(b^d)$
Yes	Yes A* Search	$O(b^d)$	$O(b^d)$
Yes** Uniform-Cost Search (UCS)	$O(b^d)$	$O(b^d)$	Yes

Table 1: Performance Comparison of Common Search Algorithms

This table highlights the inherent trade-offs between time and memory efficiency versus optimality and completeness. For example, BFS guarantees the shortest path but consumes large memory, whereas DFS is memory-efficient but may not find the optimal solution.

Educational Insights and Learning Outcomes

Through the implementation and analysis of these algorithms, several key learning outcomes can be achieved:

- **Algorithm Design and Strategy:** Developing systematic approaches to problem-solving and understanding the differences between informed and uninformed search methods.
- **Data Structure Applications:** Utilizing appropriate data structures such as queues, stacks, priority queues, and sets for efficient state management.
- **Complexity Awareness:** Evaluating algorithms based on their time and space requirements to choose suitable strategies for different problem sizes.
- **Performance Measurement:** Observing algorithm behavior through experimental execution and comparing theoretical expectations with practical outcomes.
- **Trade-off Analysis:** Learning to balance the demands of optimality, completeness, and efficiency depending on the scenario.

Summary and Conclusions

This laboratory exercise offered hands-on experience in implementing and evaluating AI search algorithms. Specifically, solving the N–Puzzle using BFS, IDS, A*, UCS, and other strategies allowed students to:

- Apply a structured methodology for algorithm development and testing.
- Observe the impact of algorithm choice on memory usage, time performance, and solution quality.
- Understand the benefits and limitations of different search techniques in practice.
- Gain insights into optimization opportunities and real-world constraints in algorithmic problem-solving.

While BFS ensures optimality for unit-cost problems, its exponential memory consumption limits practical use to smaller instances. IDS provides a memory-efficient alternative while preserving completeness. Heuristic approaches such as A* combine optimality with efficiency by guiding the search intelligently. UCS guarantees optimal paths for varying step costs but may expand many nodes unnecessarily in large state spaces.

Future exercises should extend this framework by implementing additional informed and hybrid search strategies, performing comparative evaluations, and exploring real-world problem domains beyond puzzles. This laboratory framework establishes a foundation for methodical algorithm analysis and practical AI problem-solving skills.

Applications

- Robotics pathfinding
- Game AI (puzzle and board games)
- Route optimization problems
- AI search-based problem solving

Conclusion

The N–Puzzle problem highlights the strengths and limitations of various search strategies:

- **BFS (Breadth-First Search):** Guarantees the shortest path but suffers from high memory usage, making it impractical for larger puzzles.
- **DFS (Depth-First Search):** Uses minimal memory and can reach deep solutions quickly, but does not guarantee optimality and may get trapped in long or infinite paths.
- **DLS (Depth-Limited Search):** Controls DFS's depth problem, but selecting an appropriate depth limit is challenging, and it may fail if the solution lies beyond the limit.
- **UCS (Uniform-Cost Search):** Guarantees optimal solutions for cost-based problems and expands fewer nodes than BFS when edge costs vary, though it still consumes significant memory for large state spaces.
- **BDS (Bidirectional Search):** Reduces the search space by meeting in the middle, but requires additional memory and careful frontier management.
- **A* Search:** Combines the benefits of UCS with heuristic guidance, efficiently finding optimal solutions while exploring fewer nodes. With an admissible and consistent heuristic, it outperforms uninformed algorithms in both time and space for most N–Puzzle instances.

Overall: Among the evaluated algorithms, **A*** demonstrates the best balance of efficiency, optimality, and resource usage. Uninformed strategies like BFS, DFS, and UCS illustrate fundamental search principles, while heuristic-based approaches such as A* and bidirectional search showcase how intelligent exploration can greatly enhance performance.