Puzzle PS1 – Exercise
Nguyen Dac Tin – M11323801

5.a. Search Strategies (solver 1-2)

Solver1: Uses Breadth-First Search (BFS)
- Use a Queue, which processes nodes in FIFO (First-In-First-Out) order
- BFS explores all nodes at the current depth before moving to nodes at the next depth level

Solver2: Uses Depth-First Search (DFS)
- Use a Stack, which processes nodes in LIFO (Last-In-First-Out) order
- DFS explores as far as possible along each branch before backtracking

5.b. Data structure

Solver1: Uses a Queue data structure
- Implemented with a list using `insert(0,item)` for adding (at front)
- Uses `pop()` for removing (from end)
- Creates FIFO behavior

Solver2: Uses a Stack data structure
- Implemented with a list using `append()` for pushing
- Uses `pop()` for popping
- Creates LIFO behavior

Solver3: Uses a Priority Queue
- Uses Python's built-in `PriorityQueue` from the `queue` module
- Implements a priority-based ordering of nodes
- The heuristic function determines the priority value

5.c. Operations of the data structure

Queue (Solver1)
[Front] -> [..., item3, item2, item1] -> [Back]

Add operation (at front):
Initial: [3, 2, 1]
Add(4): [4, 3, 2, 1]

Get operation (from back):
Initial: [4, 3, 2, 1]
Get(): [4, 3, 2] Returns: 1

- Apply FIFO (First-In-First-Out)
- `add()`: Inserts at position 0 (front)
- `get()`: Removes from end (back) using pop

Stack (Solver2)

| *item3* | ← *Top*
| *item2* |
| *item1* |

```
 └─────┘
```

Push operation:
Initial: [1, 2, 3]
Push(4):   [1, 2, 3, 4]

Pop operation:
Initial: [1, 2, 3, 4]
Pop():   [1, 2, 3]   Returns: 4

- Apply LIFO (Last-In-First-Out)
- `add()`: Adds to top using push
- `get()`: Removes from top using pop

Priority Queue (Solver3)
Min-Heap Structure:

```
        (1,A)
       /     \
    (3,B)   (4,C)
    /     \
  (5,D) (6,E)
```

Put operation (value=2):

```
        (1,A)
       /     \
    (2,F)   (4,C)
    /    \
  (5,D) (3,B)
```

Get operation:
Returns (1,A), then rebalances:

```
        (2,F)
       /     \
    (3,B)   (4,C)
    /    \
  (5,D) (6,E)
```

- Apply Priority-based ordering
- add():
+ Use `put((value, board))`:
+ Adds item with priority value
+ Maintains heap property (parent always smaller than children)
- `get()`:
+ Removes and returns item with lowest value
+ Restructures heap to maintain properties
- Value determined by heuristic (*count* of misplaced tiles)

Example:
Goal: Current:
1 2 3   1 2 3
4 5 6   4 6 5
7 8 0   7 8 0
Heuristic value = 1 (only one tile misplaced)

6. How Solver3 works and comparison

How Solver3 works

- Uses Best-First Search with a heuristic function
- Current heuristic: counts misplaced tiles

```
def heuristic(self,board):
    count = 0
    for x in range(self.w):
        for y in range(self.h):
            if self.goal[x][y] != board[x][y]:
                count += 1
    return count
```

- States with fewer misplaced tiles get higher priority
- Uses PriorityQueue to always explore most promising states first

Comparison with Solver1 and Solver2

Solver3 is generally better because:

	Solver1 (BFS)	Solver2 (DFS)	Solver3
Informative	Uninformed (blind)	Uninformed (blind)	Use tile position to guide
Efficiency	Go for all possibilities	May go deep into unproductive path	Prioritize promising states
Memory Usage	Store all stages at a level	Store path to current state	Store states but efficiently with promising ones

7. Solver4 and comparison with Solver3

Solver4's illustration:

```
class Solver4(Solver):
    """
    Concrete solver 4: Uses Priority Queue with Manhattan distance 
heuristic
    """
    def __init__(self, goal, width, height):
        self.pq = Q.PriorityQueue()  # Priority Queue for best-first 
search
        self.goal = goal             # Target configuration
        self.w = width               # Board width
        self.h = height              # Board height
```

```
    def get(self):
        if not self.pq.empty():
            (val, board) = self.pq.get()  # Get board state with
lowest heuristic value
            return board
        else:
            return None

    def add(self, board):
        value = self.manhattan_distance(board)  # Calculate priority
using Manhattan distance
        self.pq.put((value, board))            # Add to queue with
priority value

    def manhattan_distance(self, board):
        total_distance = 0
        # Create goal position lookup dictionary
        goal_positions = {}
        for x in range(self.w):
            for y in range(self.h):
                goal_positions[self.goal[x][y]] = (x, y)

        # Calculate Manhattan distance for each tile
```

- Data structure:
+ Orders states by Manhattan distance value
+ Lower values = higher priority

- Manhattan distance heuristic:
Example calculation:
Goal: Current:
1 2 3 4 8 7
4 5 6 1 0 6
7 8 0 2 3 5

For tile 4:
- Current: (0,0)
- Goal: (1,0)
- Distance = |0-1| + |0-0| = 1

Total = sum of distances for all tiles
- Search process:
+ States with smaller Manhattan distances explored first
+ More accurate estimation of moves needed

Why Solver4 is better

- More Informative Heuristic
+ Solver3 only counts if a tile is in wrong position but Solver4 (Manhattan distance) considers actual distance tiles need to move
+ Example:

Goal: Current:
1 2 3 4 8 7
4 5 6 1 0 6
7 8 0 2 3 5

Solver3: Counts 8 misplaced tiles
Solver4: Sums actual distances each tile must move

- Better Path Finding
+ Manhattan distance provides more granular information, choosing more efficient moves
+ Results in shorter solution paths for complex puzzles
- Efficiency Metrics
+ Time: Better performance on complex puzzles
+ Moves: Fewer moves needed for solution
+ States: Explores fewer states to find solution

Comparison metrics
1. For Simple Case `[[1,4,7],[0,5,8],[2,3,6]]`:
Solver3:
- Time taken: 5.951 seconds
- Moves made: 5
- States explored: 5
Solver4:
- Time taken: 5.939 seconds
- Moves made: 5
- States explored: 5
-> Both solvers perform equally effective

2. For Complex Case `[[4,8,7],[1,0,6],[2,3,5]]`:
Solver3:
- Time taken: 25.036 seconds
- Moves made: 25
- States explored: 25
Solver4:
- Time taken: 18.990 seconds
- Moves made: 19
- States explored: 19
-> Solver4 is more effective for complex configurations