AI Lab 3 Documentation Report

*This report provides an overview of the implementation and performance of the 8 puzzle problem using search strategies such as **A Star** and **Uniform Cost Search** to solve the problem optimally.*

Class overview

Class	Purpose
`GameState`	Represents a 3x3, 8 puzzle board configuration of the initial and goal states. **Key Features:** Track the position of the empty cell, generates all valid and possible moves, checks if the current game state is the goal state, implements `equals()` and `hashCode()` for efficient state comparisons and storage in collections like a hash set. **Usage:** Used by the `Node` class to represent the state of the puzzle at each step. Called by `A Star` and `UCS` to generate successors during the search.
`Node`	Represents a node structure in the search tree, stores the *current state*, *parent node*, *cost* and *heuristic*. **Key Features:** Tracks the cost of `g(n)` and the heuristic value of `f(n)`. Implements a comparable to define priority and ordering of node in a priority queue. Overrides `equals()` and `hashCode` to compare nodes based on the `gameState`. **Usage:** Used by `A Star` and `UCS` to build the search tree. Ensures that nodes are prioritized in the correct way with the implementation of the `compareTo()` method, `f(n)` for `A star` and `g(n)` for `UCS`.
`Solver`	Abstract class that serves as the base for `A Star` and `UCS`. **Key Features:** Provides common features such as tracking expanded and unexpanded nodes. Provides a method to report solution which prints the solution path to a file. **Usage:** Both `A Star` and `UCS` extend the *Solver* class and implement specific search logic. Handles commons tasks and reporting.
`A Star`	Solve the 8 puzzle problem using the `A Star` algorithm. **Key Features:** Uses a priority queue to prioritize the nodes with the lowest

Class	Purpose
	`f(n)` score. Calculates the heuristic `calculateHeuristic()` using the Manhattan distance.
`UCS`	Solve the 8 puzzle problem using the `Uniform Cost Search` algorithm. **Key Features:** Uses a queue to explore nodes in order of increasing cost `g(n)`.
`RunSolver`	Entry point of the program, allows the user to choose between `A Star` or `UCS`

Implementation of `possibleMoves()`

*This method is responsible for generating all **valid** and **possible** moves for a given game state and returning a list of those valid moves.*

The empty space is represented by a `0`. In 3x3 board setup which uses a `2D array` this can be represented as an `emptyRow` and `emptyCol` which gives the row and column position of the empty space in the `2D array`. The method checks for all possible directions that can be taken for a empty position.

The possible directions include `UP, DOWN, LEFT, RIGHT`. This is represented by a `2D array` which has directional coordinates for each move, specifically,

```
static final int[][] DIRECTIONS = {
    {-1, 0},  // UP (row decreases by 1, column remains the same)
    {1, 0},   // DOWN (row increases by 1, column remains the same)
    {0, -1},  // LEFT (column decreases by 1, row remains the same)
    {0, 1}    // RIGHT (column increases by 1, row remains the same)
};
```

Validation: For each direction, the method `possibleMoves()` checks if the new position is within the bounds of a `3x3` board using a helper function called `checkIfValidPosition()`. If the move is valid we generate a new `gameState` and add it to the list of possible moves.

Below is the **pseudocode** for generating all possible moves:

```
FUNCTION possibleMoves():
    moves = []
    FOR EACH direction IN DIRECTIONS:

        Calculate the new position of the empty space
        Check if the new position is within the bounds of the board
        Create a copy of the current board
        Swap the empty space with the tile in the new position
```

```
        Update the empty space position
        Add the new state to the list of valid moves

    RETURN moves
```

Implementation of `calculateHeuristic()`

This method is responsible for implementing an admissible heuristic for the 8 puzzle problem used by the `A Star` algorithm.

The heuristic implemented was the **Manhattan Distance**.

For each tile in a given game state, the method calculates the distance of how far each tile is from its goal position. The cumulative sum of each tiles distance to their goal is than calculated to represent the **total** of how far the board is from completion (*goal state*) for the given game state.

Below is the **pseudocode** for the **Manhattan Distance.**

```
FUNCTION calculateHeuristic(gameState):
    distance = 0
    Get Board from GameState

    FOR i FROM 0 TO 2:
        FOR j FROM 0 TO 2:
            Get current tile value

            IF current tile value is NOT empty:
                CALCULATE goal position of current tile
                CALCULATE manhattan distance of current tile
                UPDATE distance

    RETURN distance
```

Performance for UCS and A Star

*The performance of **UCS** and **A Star** are tracked based on the following initial board configuration.*

8	**7**	**6**
5	4	3
2	1	

Uniform Cost Search	A Star
30 Moves required to solve	30 Moves required to solve
Time Taken: ~0.30 seconds	Time Taken: ~0.035 seconds
Nodes expanded: 181393 Nodes unexpanded: 47	Nodes expanded: 8105 Nodes unexpanded: 4238

Conclusion: A Star is **~88% faster** than UCS for the 8 puzzle problem