**Assignment 1**

**ii) Documentation of Code**

The C++ code implements the A\* algorithm to solve sliding puzzles (2x2, 3x3, 4x4, 5x5) using **Manhattan distance** as the heuristic. Here's a detailed breakdown:

**1. Puzzle Class**

* **Purpose:** Represents the puzzle, holds the current state, and provides functionality to calculate the Manhattan distance and generate neighboring states.
* **Attributes:**
* size\_of\_array: Size of the puzzle (e.g., 2, 3, 4, 5).
* starting\_state: The starting configuration of the puzzle.
* goal\_state: The target configuration to reach.
* goal: Maps each tile value to its index in the goal state.
* **Functions:**
* manhattan(): Calculates the Manhattan distance between the current state and the goal state.
* is\_goal(): Checks if the current state is the goal state.
* neighbours(): Generates possible moves by sliding adjacent tiles into the empty space (represented by 0).

**2. AStarNode Struct**

* **Purpose:** Represents a node in the A\* search, holding the current state, the cost to reach the node (node\_cost), the estimated total cost , and the parent node for backtracking.
* **Attributes:**
* state: The current configuration of the puzzle.
* node\_cost: Cost to reach this state.
* final\_cost: Total estimated cost to reach the goal (node\_cost + heuristic).
* parent: Pointer to the parent node (for reconstructing the path).
* direction: The direction of the move made to reach this node (e.g., up, down, left, right).

**3. Priorityqueue Struct**

* **Purpose:** Comparator for the priority queue to sort the nodes based on their f value, ensuring the lowest finalcost value is explored first.

**4. AStar Class**

* **Purpose:** Implements the A\* algorithm to solve the puzzle.
* **Functions:**
* solve(): Main function to solve the puzzle. It uses a priority queue to explore the puzzle states, guided by the Manhattan distance heuristic.
* helper(): Serializes the puzzle state for efficient lookup in hash maps.
* reconstruct\_path\_direction(): Reconstructs the solution path and movement directions by backtracking from the goal state.
* get\_move\_direction(): Converts the move direction (e.g., (-1, 0)) into a human-readable format (e.g., "Up", "Down").

**5. Input/Output Handling**

* **read:** Reads the input file, which contains the puzzle size, maximum moves allowed, start state, and goal state.
* **Main Program:**
* Solves the puzzle and prints the solution, including each move and the time taken to solve the puzzle.

**iii) Description of Implementation Details (Heuristic and Algorithm)**

***Algorithm: A Search*\***

The A\* algorithm is a graph traversal and pathfinding algorithm. It is widely used due to its performance and accuracy. A\* is an extension of Dijkstra’s algorithm, using a heuristic to guide its search more efficiently.

* **Cost to Reach Node (nodecost):** This represents the actual cost incurred to reach a particular state. In this puzzle-solving problem, each move (sliding a tile) adds 1 to the cost.
* **Heuristic (h): Manhattan Distance:** The **Manhattan distance** heuristic is used to estimate the distance between the current state and the goal state. The Manhattan distance works well in sliding puzzles because tiles can only be moved in four directions (up, down, left, right).
* **Total Cost (final\_cost):** A\* combines the cost nodecost with the heuristic h to form the total estimated cost final\_cost:  
   finalcost=nodecost+heuristics
* **Priority Queue:** The A\* algorithm uses a min-heap (priority queue) to always explore the node with the lowest final\_cost value, prioritizing the nodes that are closest to the goal in terms of total cost.

**Steps:**

* **Initialization:** The start state is pushed into a priority queue. The final\_cost value is initialized as node\_cost (0) + h (Manhattan distance).
* **Exploration:** The node with the smallest f value is expanded. Neighbors (possible moves) are generated and evaluated.
* **Heuristic Application:** For each neighbor, the Manhattan distance is calculated. The node is pushed into the priority queue with the updated cost (node\_cost + 1) and the heuristic estimate (h).
* **Goal Check:** If the goal state is reached, the solution path is reconstructed.
* **Backtracking:** The solution path is printed, showing the sequence of moves.

**Results and Stats :**  
 **Sample 1:**



**Output**

Solved in 18 moves:

1 2 0

5 4 7

6 3 8

Move 1: Left

1 0 2

5 4 7

6 3 8

Move 2: Down

1 4 2

5 0 7

6 3 8

Move 3: Right

1 4 2

5 7 0

6 3 8

Move 4: Down

1 4 2

5 7 8

6 3 0

Move 5: Left

1 4 2

5 7 8

6 0 3

Move 6: Up

1 4 2

5 0 8

6 7 3

Move 7: Left

1 4 2

0 5 8

6 7 3

Move 8: Down

1 4 2

6 5 8

0 7 3

Move 9: Right

1 4 2

6 5 8

7 0 3

Move 10: Right

1 4 2

6 5 8

7 3 0

Move 11: Up

1 4 2

6 5 0

7 3 8

Move 12: Left

1 4 2

6 0 5

7 3 8

Move 13: Down

1 4 2

6 3 5

7 0 8

Move 14: Left

1 4 2

6 3 5

0 7 8

Move 15: Up

1 4 2

0 3 5

6 7 8

Move 16: Right

1 4 2

3 0 5

6 7 8

Move 17: Up

1 0 2

3 4 5

6 7 8

Move 18: Left

0 1 2

3 4 5

6 7 8

Time taken: 0.0062434 seconds  
  
**Sample 2**  
  
  
**Sample 3**  
  
Solved in 44 moves:

1 2 3 4

5 6 7 8

0 10 11 12

13 15 9 14

Move 1: Right

1 2 3 4

5 6 7 8

10 0 11 12

13 15 9 14

Move 2: Down

1 2 3 4

5 6 7 8

10 15 11 12

13 0 9 14

Move 3: Right

1 2 3 4

5 6 7 8

10 15 11 12

13 9 0 14

Move 4: Up

1 2 3 4

5 6 7 8

10 15 0 12

13 9 11 14

Move 5: Up

1 2 3 4

5 6 0 8

10 15 7 12

13 9 11 14

Move 6: Right

1 2 3 4

5 6 8 0

10 15 7 12

13 9 11 14

Move 7: Down

1 2 3 4

5 6 8 12

10 15 7 0

13 9 11 14

Move 8: Left

1 2 3 4

5 6 8 12

10 15 0 7

13 9 11 14

Move 9: Left

1 2 3 4

5 6 8 12

10 0 15 7

13 9 11 14

Move 10: Up

1 2 3 4

5 0 8 12

10 6 15 7

13 9 11 14

Move 11: Right

1 2 3 4

5 8 0 12

10 6 15 7

13 9 11 14

Move 12: Right

1 2 3 4

5 8 12 0

10 6 15 7

13 9 11 14

Move 13: Up

1 2 3 0

5 8 12 4

10 6 15 7

13 9 11 14

Move 14: Left

1 2 0 3

5 8 12 4

10 6 15 7

13 9 11 14

Move 15: Left

1 0 2 3

5 8 12 4

10 6 15 7

13 9 11 14

Move 16: Down

1 8 2 3

5 0 12 4

10 6 15 7

13 9 11 14

Move 17: Right

1 8 2 3

5 12 0 4

10 6 15 7

13 9 11 14

Move 18: Right

1 8 2 3

5 12 4 0

10 6 15 7

13 9 11 14

Move 19: Down

1 8 2 3

5 12 4 7

10 6 15 0

13 9 11 14

Move 20: Left

1 8 2 3

5 12 4 7

10 6 0 15

13 9 11 14

Move 21: Down

1 8 2 3

5 12 4 7

10 6 11 15

13 9 0 14

Move 22: Right

1 8 2 3

5 12 4 7

10 6 11 15

13 9 14 0

Move 23: Up

1 8 2 3

5 12 4 7

10 6 11 0

13 9 14 15

Move 24: Left

1 8 2 3

5 12 4 7

10 6 0 11

13 9 14 15

Move 25: Left

1 8 2 3

5 12 4 7

10 0 6 11

13 9 14 15

Move 26: Left

1 8 2 3

5 12 4 7

0 10 6 11

13 9 14 15

Move 27: Up

1 8 2 3

0 12 4 7

5 10 6 11

13 9 14 15

Move 28: Right

1 8 2 3

12 0 4 7

5 10 6 11

13 9 14 15

Move 29: Right

1 8 2 3

12 4 0 7

5 10 6 11

13 9 14 15

Move 30: Down

1 8 2 3

12 4 6 7

5 10 0 11

13 9 14 15

Move 31: Left

1 8 2 3

12 4 6 7

5 0 10 11

13 9 14 15

Move 32: Left

1 8 2 3

12 4 6 7

0 5 10 11

13 9 14 15

Move 33: Up

1 8 2 3

0 4 6 7

12 5 10 11

13 9 14 15

Move 34: Right

1 8 2 3

4 0 6 7

12 5 10 11

13 9 14 15

Move 35: Up

1 0 2 3

4 8 6 7

12 5 10 11

13 9 14 15

Move 36: Left

0 1 2 3

4 8 6 7

12 5 10 11

13 9 14 15

Move 37: Down

4 1 2 3

0 8 6 7

12 5 10 11

13 9 14 15

Move 38: Right

4 1 2 3

8 0 6 7

12 5 10 11

13 9 14 15

Move 39: Down

4 1 2 3

8 5 6 7

12 0 10 11

13 9 14 15

Move 40: Down

4 1 2 3

8 5 6 7

12 9 10 11

13 0 14 15

Move 41: Left

4 1 2 3

8 5 6 7

12 9 10 11

0 13 14 15

Move 42: Up

4 1 2 3

8 5 6 7

0 9 10 11

12 13 14 15

Move 43: Up

4 1 2 3

0 5 6 7

8 9 10 11

12 13 14 15

Move 44: Up

0 1 2 3

4 5 6 7

8 9 10 11

12 13 14 15

Time taken: 24.8219 seconds

**Sample 4**  
   
**Output**  


—----------------------------------------------------------------------------------------------------------------------------