Sliding Puzzle Solver

---

1. A* with the two heuristics

- Manhattan Distance:
    - This heuristic works with the sum of the distances between the current location of the tile and the goal position. It sums the vertical and horizontal distances for each tile and gives a better estimate of how many moves it will take to solve the puzzle.
    - I found the Manhattan Distance heuristic more effective than the Misplaced Tile heuristic because it provides faster results while guaranteeing an optimal solution, it was also more effective in reducing the depth of the search and the number of states explored.
    - Example: the solution path for the 4th test puzzle was as follows:
        - Solution Path: [Down, Down, Left, Up, Up, Up, Left, Down, Down, Down, Left, Up, Right, Down, Right, Up, Left, Up, Left, Up, Right, Down, Right, Up, Right, Down, Left, Down, Down, Right]
        - The optimal solution path is different than that of Misplaced Tiles Heuristics.
- Misplaced Tiles Heuristics:
    - This heuristic counts how many tiles are out of place compared to the goal state. This heuristic is relatively easy to implement, however since it does not consider how far each of the misplaced tiles is from the goal state, it expands to many unnecessary states.
    - The Misplaced Tiles heuristic works perfectly and provides an optimal solution as well, however, the depth of the search tends to be much deeper and the search takes more time. This heuristic will not be ideal for more complex puzzles where memory and performance are key requirements.
    - Example: the solution path for the 4th test puzzle was as follows:
        - Solution Path: [Down, Down, Left, Up, Up, Up, Left, Down, Left, Up, Right, Down, Down, Down, Left, Up, Right, Down, Right, Up, Left, Up, Right, Up, Right, Down, Left, Down, Down, Right]

        - The optimal solution path is different than that of the Manhattan Distance heuristics.

---

2. A* with a heuristic of your design/research

Linear Conflict Heuristic:

    - The third heuristic that I implemented builds upon the logic of Manhattan Distance. The logic behind Linear Conflict is that when two tiles are in the correct row or column but in the wrong order relative to their goal position.
    - The Linear Conflict heuristic performed similarly to the Manhattan distance heuristic, even though it did not show much improvement in these particular test puzzles, it will provide better solutions for puzzles with more linear conflict. The heuristic does add more complexity to calculate linear conflicts.

How I Found/Researched This Heuristic:

- The following are all the resources I used to research this particular heuristic:
    - https://science.slc.edu/~jmarshall/courses/2005/fall/cs151/lectures/heuristic-search/#:~:text=A%20good%20heuristic%20for%20the%208%2Dpuzzle%20is%20the%20number,direct%20adjacent%20tile%20reversals%20present.
    - https://cse.iitk.ac.in/users/cs365/2009/ppt/13jan_Aman.pdf
    - https://cdn.aaai.org/AAAI/1996/AAAI96-178.pdf
    - https://chatgpt.com/share/6716dd11-037c-8012-9cb7-ae8c6a161aac

---

3. A description of your heuristic implemented for your A* search.

The Linear Conflict starts with the basic Manhattan distance calculation and adds 2 moves for each conflicting pair of tiles in a row or column. As mentioned earlier, a linear conflict occurs when two tiles are in the correct row or column but are in the wrong relative order.

Logic and Code Implementation:

- The code first calculates the Manhattan Distance for the current puzzle state. This gives an initial estimate of how many moves each tile needs to reach its goal state.
- Detecting Linear Conflicts, once the Manhattan Distance is found, the code will check for any Linear Conflict that exists within the rows and columns. 2 moves are added for each conflict.

- The final heuristic value is calculated by adding the Manhattan Distance and Linear Conflict penalties together.

Linear Conflict = Manhattan Distance + (2 x number of Conflicting pairs)

This heuristic helps A* search by providing it with more informed guidance on path exploration, especially in cases where the tiles are in the correct row/column but not in the correct order.

---

4. A brief section about how each implemented search strategy performed and compared to the other strategies.

- Iterative Deepening Depth-First Search:
  - IDDFS search is able the find the optimal solution through a series of depth-first searches with increasing depth limits. The search performed well for all the test puzzles except the 4th puzzle, initially with a Max Depth limit of 25 it was unable to find any solution for the 4th puzzle. Once I increased the limit the search kept running for a very long time, and IDDFS expanded many unnecessary states before finding the solution.
  - Pros: Completeness and memory efficiency.
  - Cons: For complex puzzles, it performs far slower than A* search.
- A* Search:
  - A* Search uses three different heuristics to find an optimal path. Being an informed search, A* was able to find the optimal solution much faster, it was able to quickly solve the 4th puzzle as well which IDDFS took a long time for.
  - Pros: Informed Search, faster and more efficient for complicated puzzles.
  - Cons: Consumes more memory and can expand into significantly more states depending on the heuristic.

---

5. A brief section for each implemented heuristic function and how they compare to each other.

Misplaced Tiles:

- This Heuristic is the simplest and the least efficient out of the three. It finds the optimal solution, however, it explores many unnecessary states, which makes it less reliable for large/complex puzzles.

Manhattan Distance:

- Manhattan Distance does not expand nearly as many states compared to the Misplaced Tiles heuristic. It provides a better estimate of the actual moves required for each tile to reach its goal state.

Linear Conflict:

- This heuristic expands on the logic of Manhattan Distance, by adding moves when there is a conflict in the same row/column. With the given puzzles Linear Conflict performed the same as the Manhattan Distance heuristic, it was able to find the optimal solution in less time compared to Misplaced Tiles, and it expanded fewer states in the process. However, if we were to test puzzles with more linear conflicts then it may lead to better performance as it is expected to outperform Manhattan Distance by reducing the number of state expansions.

---

6. An explanation of how to run and use the program, how to interpret the output (if needed), and any additional information you feel is needed to inform the marker.

The program is implemented in Java using IntelliJ. The puzzle is read from a text file, and the user can choose how to solve a puzzle, using either IDDFS or A* Search (with 3 heuristics).

Project Breakdown:

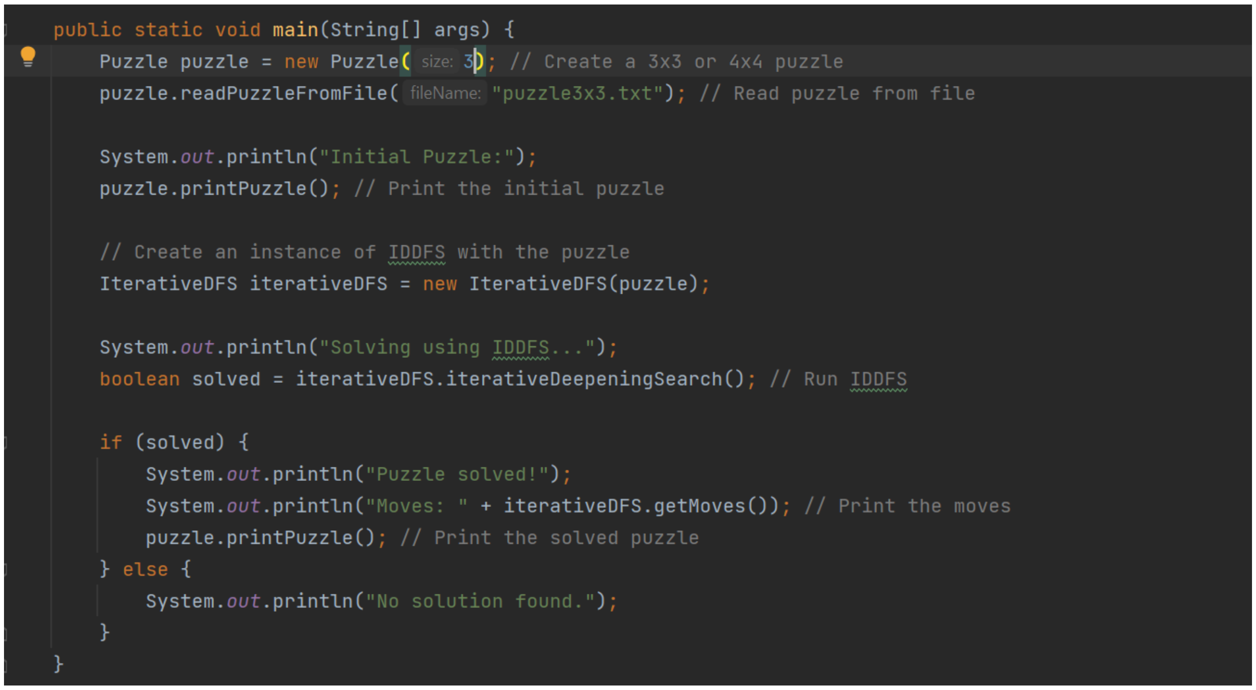
- The 'Puzzle.java' class handles reading the initial puzzle from a file and provides methods for calculating the optimal path using the heuristics or IDDFS using tile movements (right, left, up, and down).
- The 'IterativeDFS.java' class handles the main operations of the IDDFS search. This class has a main method inside which the user can use to run and test the IDDFS search.
- The 'AStar.java' class uses a priority queue to expand the state with the lowest total cost, $f(n) = g(n) + h(n)$. It also keeps track of all the visited states using a HashSet, this helps avoid going back to a state which has already been explored. This class also has a main

method that can be used to run/test all the heuristics (Manhattan Distance, Linear Conflict, and Misplaced Tiles)

- The ‘State.java’ class stores the cost to reach the current state g(n), the heuristic cost h(n), and the total cost f(n). It also keeps track of the previous state and implements comparisons for PQ ensuring that the state with the lowest f(n) is expanded first.

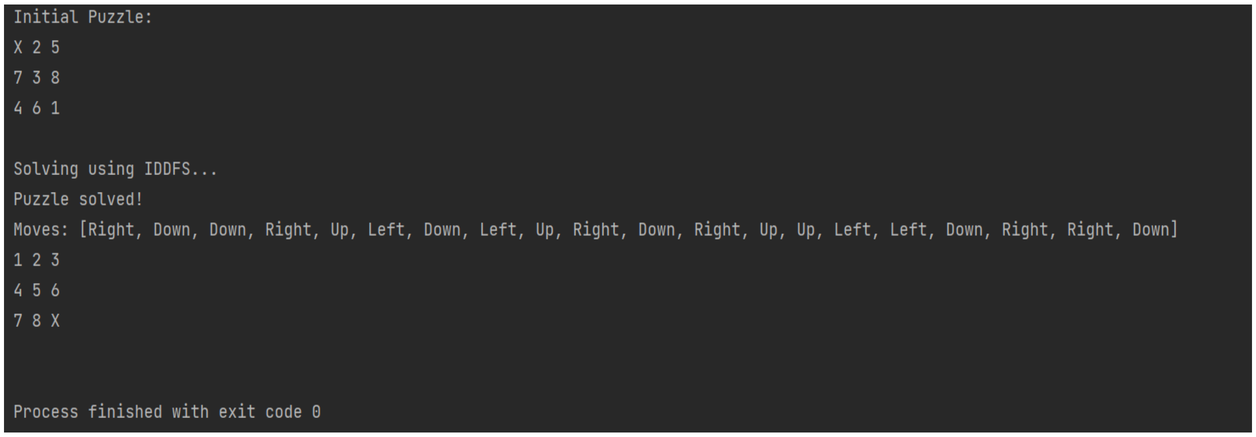
How to run the code:

- Iterative DFS:
  - You pick whether to create a 3x3 or a 4x4 puzzle
  - I have created two files ‘puzzle3x3’ and ‘puzzle4x4’ which contain a 3x3 and a 4x4 puzzle respectively. You can either replace those puzzles in the text files with the puzzle of your preference or add a new text file and provide its path.
  - Example: This creates a 3x3 puzzle and reads off ‘puzzle3x3.txt’



```java
public static void main(String[] args) {
    Puzzle puzzle = new Puzzle( size: 3); // Create a 3x3 or 4x4 puzzle
    puzzle.readPuzzleFromFile( fileName: "puzzle3x3.txt"); // Read puzzle from file

    System.out.println("Initial Puzzle:");
    puzzle.printPuzzle(); // Print the initial puzzle

    // Create an instance of IDDFS with the puzzle
    IterativeDFS iterativeDFS = new IterativeDFS(puzzle);

    System.out.println("Solving using IDDFS...");
    boolean solved = iterativeDFS.iterativeDeepeningSearch(); // Run IDDFS

    if (solved) {
        System.out.println("Puzzle solved!");
        System.out.println("Moves: " + iterativeDFS.getMoves()); // Print the moves
        puzzle.printPuzzle(); // Print the solved puzzle
    } else {
        System.out.println("No solution found.");
    }
}
```

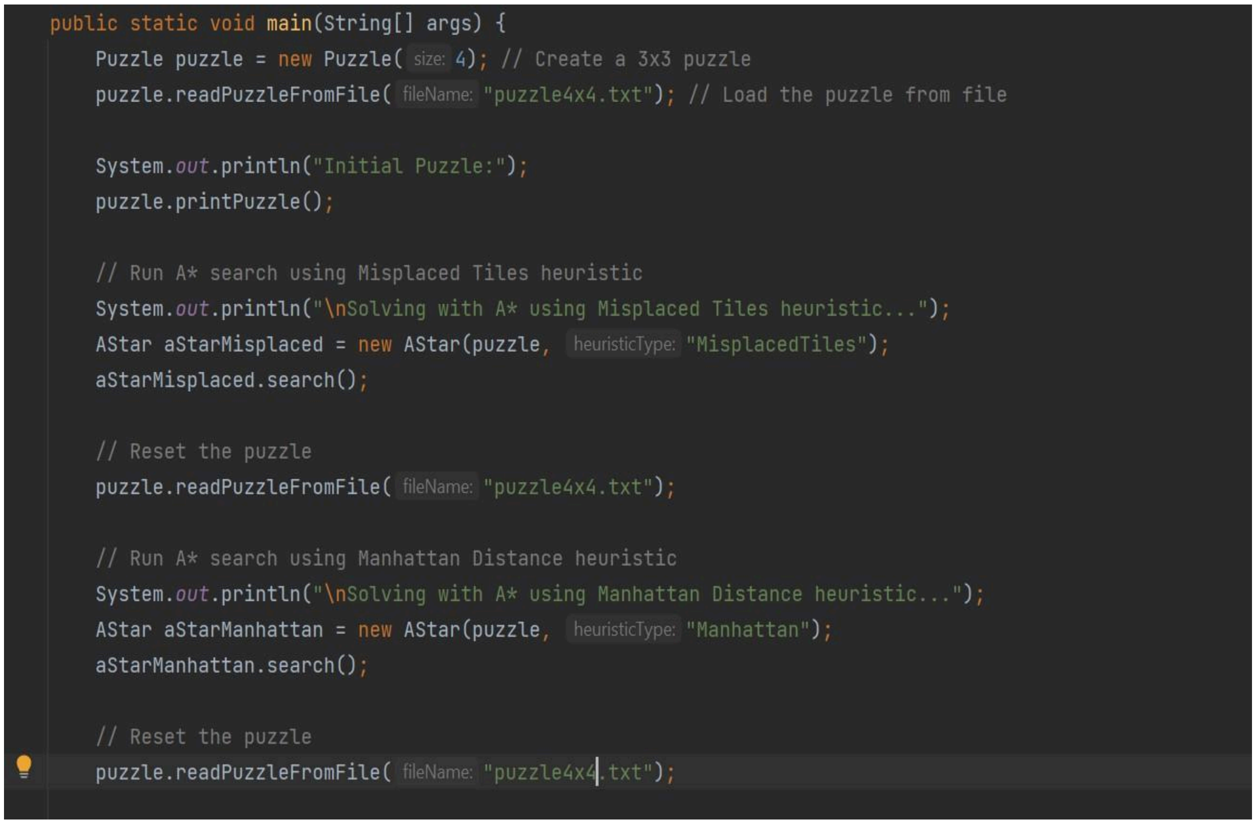
In this instance, the code outputs the following:



```
Initial Puzzle:
X 2 5
7 3 8
4 6 1

Solving using IDDFS...
Puzzle solved!
Moves: [Right, Down, Down, Right, Up, Left, Down, Left, Up, Right, Down, Right, Up, Up, Left, Left, Down, Right, Right, Down]
1 2 3
4 5 6
7 8 X


Process finished with exit code 0
```

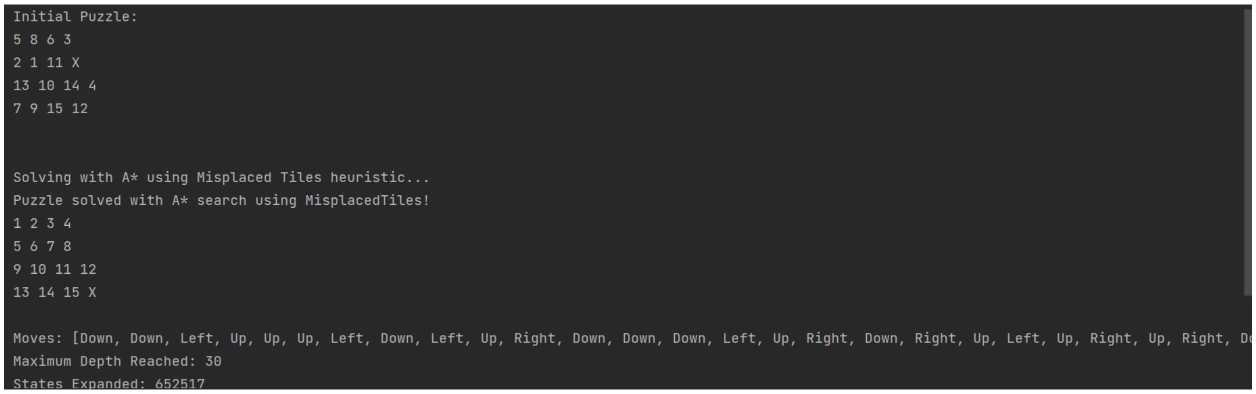
The output gives you the initial puzzle, the solved puzzle, and the moves it took to achieve the goal state.

- A*:
  - It works similar to IDDFS, you create either 3x3 or 4x4 puzzle and it will read off either ‘puzzle3x3’ or ‘puzzle4x4’
  - Example:

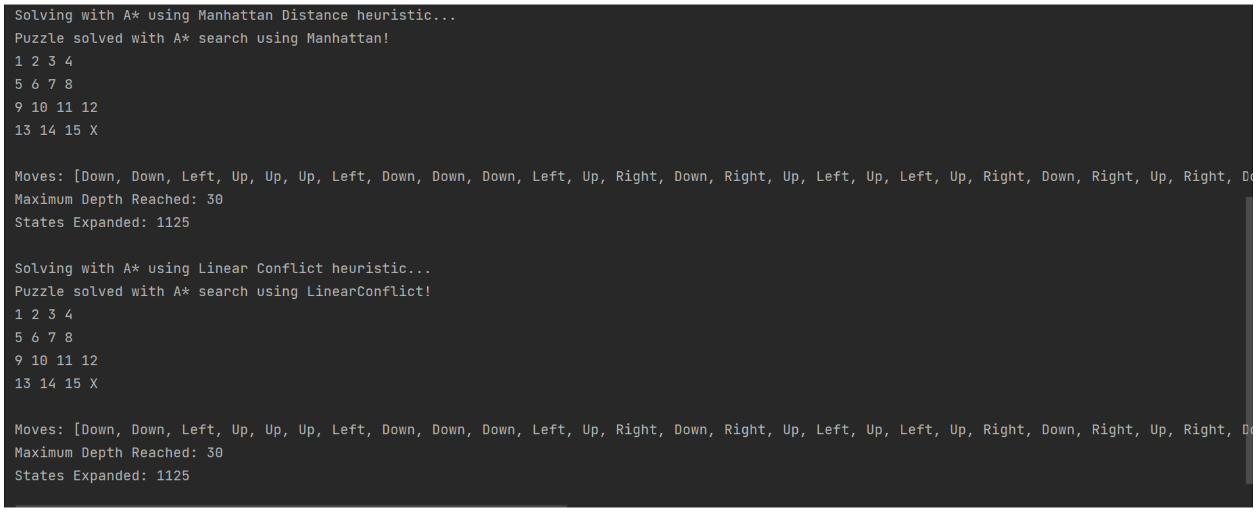


```
public static void main(String[] args) {
    Puzzle puzzle = new Puzzle( size: 4); // Create a 3x3 puzzle
    puzzle.readPuzzleFromFile( fileName: "puzzle4x4.txt"); // Load the puzzle from file

    System.out.println("Initial Puzzle:");
    puzzle.printPuzzle();

    // Run A* search using Misplaced Tiles heuristic
    System.out.println("\nSolving with A* using Misplaced Tiles heuristic...");
    AStar aStarMisplaced = new AStar(puzzle,  heuristicType: "MisplacedTiles");
    aStarMisplaced.search();

    // Reset the puzzle
    puzzle.readPuzzleFromFile( fileName: "puzzle4x4.txt");

    // Run A* search using Manhattan Distance heuristic
    System.out.println("\nSolving with A* using Manhattan Distance heuristic...");
    AStar aStarManhattan = new AStar(puzzle,  heuristicType: "Manhattan");
    aStarManhattan.search();

    // Reset the puzzle
    puzzle.readPuzzleFromFile( fileName: "puzzle4x4.txt");
```

In this instance, the program is reading a 4x4 puzzle and the output is as follows:



```
Initial Puzzle:
5 8 6 3
2 1 11 X
13 10 14 4
7 9 15 12


Solving with A* using Misplaced Tiles heuristic...
Puzzle solved with A* search using MisplacedTiles!
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 X

Moves: [Down, Down, Left, Up, Up, Up, Left, Down, Left, Up, Right, Down, Down, Down, Left, Up, Right, Down, Right, Up, Left, Up, Right, Up, Right, Do
Maximum Depth Reached: 30
States Expanded: 652517
```



```
Solving with A* using Manhattan Distance heuristic...
Puzzle solved with A* search using Manhattan!
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 X

Moves: [Down, Down, Left, Up, Up, Up, Left, Down, Down, Down, Left, Up, Right, Down, Right, Up, Left, Up, Left, Up, Right, Down, Right, Up, Right, Do
Maximum Depth Reached: 30
States Expanded: 1125

Solving with A* using Linear Conflict heuristic...
Puzzle solved with A* search using LinearConflict!
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 X

Moves: [Down, Down, Left, Up, Up, Up, Left, Down, Down, Down, Left, Up, Right, Down, Right, Up, Left, Up, Left, Up, Right, Down, Right, Up, Right, Do
Maximum Depth Reached: 30
States Expanded: 1125
```

The output displays the initial puzzle, solved puzzle, and moves as per the requirements for each of the heuristics, and to help better understand and compare the heuristics I have added the maximum depth reached and states expanded methods, however, you can completely ignore those two things since the assignment does not explicitly ask for those.