TCSS 435/535
Programming Assignment 1

Note:
Be sure to adhere to the University's **Policy on Academic Integrity** as discussed in class. Programming assignments are to be written individually and submitted programs must be the result of your own efforts. Any suspicion of academic integrity violation will be dealt with accordingly.

Objective:
Expressing the problem as a search problem and identifying proper solving methods. Specifying, designing and implementing uninformed & informed search methods.

Assignment Details:
In this programming assignment you will implement a set of search algorithms (BFS, DFS, Greedy, A*) to find solution to the **sliding puzzle problem**:

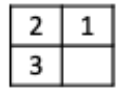
Theoretically solving the problems as a search process includes progressive construction of a solution:

- Establish the problem's components
  - Initial state
  - Final state
  - Operators (successor functions)
  - Solution
- Defining the search space
- Establish the strategy for search a solution into the search space.

Let's start by defining the **sliding puzzle problem***:*

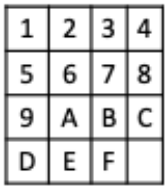
For a given puzzle of n x n squares with numbers from 1 to (n x n-1) (one square is empty) in an initial configuration, find a sequence of movements for the numbers in order to reach a final given configuration, knowing that a number can move (horizontally or vertically) on an adjacent empty square.

You will solve the puzzle for size n = 2 (2 x 2 squares), 3 (3 x 3 squares) and 4 (4 x 4 squares). The final configuration/goal state for each puzzle of size n is as follows:



Goal State
n = 2

Goal State
n = 3



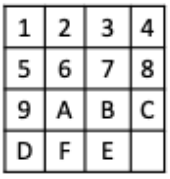
Goal State
n = 4

Important: Solvability of the sliding puzzle problem

Not all initial boards can lead to the goal board by a sequence of moves, including these two:

	2	1
3	4	5
6	7	8

unsolvable



1	2	3	4
5	6	7	8
9	A	B	C
D	F	E	

unsolvable

Remarkably, we can determine whether a board is solvable *without* solving it! To do so, we count *inversions*, as described next.

- *Inversions:* Given a board, an *inversion* is any pair of tiles *i* and *j* where $i < j$ but *i* appears after *j* when considering the board in row-major order (row 0, followed by row 1, and so forth).

1	2	3
	4	6
8	5	7

Row-major order: 1 2 3 4 6 8 5 7
3 inversions: 6-5, 8-5, 8-7

	1	3
4	2	5
7	8	6

Row-major order: 1 3 4 2 5 7 8 6
4 inversions: 3-2, 4-2, 7-6, 8-6

- *Odd-sized boards:* If a board has an odd number of inversions, it is *unsolvable* because the goal board has an even number (zero) of inversions. It turns out that the converse is also true: if a board has an even number of inversions, then it is *solvable*.

1	2	3
	4	6
8	5	7

Row-major order: 1 2 3 4 6 8 5 7
3 inversions: 6-5, 8-5, 8-7
Unsolvable

	1	3
4	2	5
7	8	6

Row-major order: 1 3 4 2 5 7 8 6
4 inversions: 3-2, 4-2, 7-6, 8-6
Solvable

In summary, when *n* is odd, an *n*-by-*n* board is solvable if and only if its number of inversions is even.

- *Even-sized boards:* Now, we'll consider the case when the board size *n* is an even integer. In this case, the parity of the number of inversions is not invariant. However, the parity of the number of inversions *plus* the row of the blank square (indexed starting at 0) is invariant: each move changes this sum by an even number.

1	2	3	4
5	6	7	8
9	A	B	C
D	F	E	

Row-major order: 1 2 3 4 5 6 7 8 9 A B C D F E
Inversion = 1 (F-E)
Row with Blank = 3
Sum = 4
Unsolvable

1	2	3	4
5	6	7	8
9	A		B
C	D	F	E

Row-major order: 1 2 3 4 5 6 7 8 9 A B C D F E
Inversion = 1 (F-E)
Row with Blank = 2
Sum = 3
Solvable

That is, when *n* is even, an *n*-by-*n* board is solvable if and only if the number of inversions plus the row of the blank square is odd.

Assignment Summary:

The search algorithms you are expected to implement are:

- Breadth-first search (BFS)
- Depth-first search (DFS) – depth-first search needs to check for cycles, or a solution will most likely never be found.
- Greedy best-first search (GBFS), using Manhattan Distance as the heuristic.
- A* (AStar) using Manhattan Distance (discussed during lecture session) as the heuristic.

Input:

Your program will accept instructions from the command line. The program should accept the following inputs in the following format:

- Sliding puzzle problem: `[size] "[initialstate]" [searchmethod]`
  - `[size]` of sliding puzzle problem (2,3 or 4)
  - `[initialstate]` must characters, namely the digits 1-9, letters A-F (only for size 4) and a space, in any order. Make sure the initial state is solvable from the given goal state.
  - `[searchmethod]` can be: BFS, DFS, GBFS, AStar.
  - Examples:
    - `2 "32 1" DFS`
    - `3 "47315862 " GBFS`
    - `4 "123456789AB DEFC" BFS`

Output:

Your program will generate 2 different outputs – one to the console & and another to a readme file.

- Console Output:
  Show solution path to the sliding puzzle. Remember planning is part of the simulation process and on console (to user) we simply want to show the final solution path from initial state to the goal state for each search method.
- `Readme.txt:`
  The Readme should include –
  - You need to include `size`, `initial` and `goal` state of the problem.

- And for each `state` and `size` of the problem, include `searchmethod` and report a comma-separated list of integers listed in the following format. This format should represent the state of your search tree at the moment the solution was discovered:
  `[depth], [numCreated], [numExpanded], [maxFringe]`
- `[depth]` represents the depth in the search tree where the solution is found. The integer will be zero if the solution is at the root and it will be "-1" if a solution was not found.
- `[numCreated]` is the counter that is incremented every time a node of the search tree is created (output 0 if depth == -1).
- `[numExpanded]` is the counter that will be incremented every time the search algorithm acquires the successor states to the current state, i.e., every time a node is pulled off the fringe and found not to be the solution (output 0 if depth == -1).
- `[maxFringe]` is the maximum size of the fringe at any point during the search (output 0 if depth == -1).

Additional Component for TCSS 535:

Additionally, you will be solving this problem for increasing size of n (sliding puzzle size): 5, 6, 7, 8, … What is the highest n you were able to solve the problem for? In your output file make sure to report the data for all n = 2, 3, 4, 5, 6, 7, 8, …highest n you were able to solve for.

Hints:

- The successors of a state in this problem depend greatly on the position of the blank spot. Rather than think about which tiles can move into the blank spot, try considering where the blank spot can move. Certain numerical qualities about this position will determine whether or not the blank can move left, right, up, or down. If a blank spot moves up, then its location is being swapped with the location above it.
- The heuristics can be generated by comparing the state being evaluated to the goal state. The number of misplaced tiles is easily calculated in time linear to the number of tiles in the puzzle, but the simple solution to the Manhattan distance requires time quadratic to the number of tiles in the puzzle.
- If you plan to start from scratch, spend a lot of time thinking about what data structures you will use for your fringe. Much of the variation between the algorithms comes in how to determine which elements to pull off the fringe next.
- Many of these algorithms are more similar than different. Think about how you can use polymorphism & fringe (one queue/list for all) to make your life easier.

Submission Guidelines:

Zip and upload the following files on Canvas using the Programming Assignment 1 submission link:

- `Board.java/Board.py`: Source code that models an n-by-n board with sliding tiles. Your heuristic function resides in this file.
- `Solver.java/Solver.py`: Source code that implements BFS, DFS, GBFS, AStar to solve n-by-n sliding puzzle.
- `Tester.java/Tester.py`: A tester file that connects your Board and Solver code.

- `Readme.txt`: Output for each of the algorithm and problem size in the format explained in the output section.

Please refer to the grading rubric for breakdown of the points.