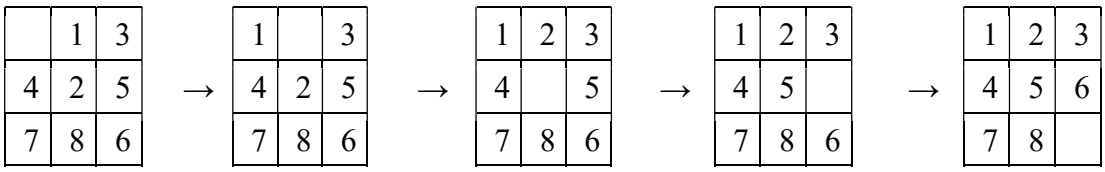
CSE 318: Offline Assignment

Solve n-Puzzle

Objective: Write a program to solve the n-puzzle problem using the A* search algorithm, where

$$n = k - 1, \qquad k = 3, 4, 5 \ldots$$

Problem description: The n-puzzle is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a $k \times k$ grid with $n = k - 1$ square blocks labeled 1 through n and a blank square. Your goal is to rearrange the blocks so that they are in order, using as few moves as possible. You are permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board (left) to the goal board (right).



Now, we describe a solution to the problem that illustrates the A* search algorithm. We define a search node of the game to be a board, the number of moves made to reach the board, and the previous search node. First, insert the initial search node (the initial board, 0 moves, and a null previous search node) into a priority queue (open list). Then delete from the priority queue the search node with the minimum priority (the most promising node), insert the node into the closed list, and insert onto the priority queue all neighboring search nodes (that can be reached by one move) which are not in the closed list. Repeat this procedure until the search node dequeued corresponds to the final goal board. The success of this approach hinges on the choice of priority function for a search node. In A* search algorithm, the priority function $f(n)$ represents an estimated cost of the path from the initial state to the goal state though node n. $f(n)$ is calculated as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach node n from the initial state and $h(n)$ is an estimated cost from node n to the goal state. The function $h(n)$ is often called a heuristic. Finding optimal path by A* search algorithm depends on the properties of the heuristic used.

In this assignment we will consider two priority functions:

1. Hamming distance: The number of blocks in the wrong position (not including the blank)
2. Manhattan distance: The sum of the Manhattan distances (sum of the vertical and horizontal distance) of the blocks to their goal positions.

Let us assume that our initial board is

7	2	4
6		5
8	3	1

Our goal board is

1	2	3
4	5	6
7	8	

From the following table we can calculate Hamming distance = 7

Digits	1	2	3	4	5	6	7	8
Is placed correctly (1 = no, 0 = yes)	1	0	1	1	1	1	1	1

From the following table we can calculate Manhattan distance = 16

Digits	1	2	3	4	5	6	7	8
Row distance	2	0	2	1	0	0	2	0
Column distance	2	0	1	2	1	2	0	1
Total distance	4	0	3	3	1	2	2	1

8	1	3
4		2
7	6	5

8	1	3
4	2	
7	6	5

8	1	3
4		2
7	6	5

8	1	3
4	2	5
7	6	

Previous node Search node Neighbor 1 Neighbor 2 Neighbor 3

We can see a search node, it's previous node and generated neighbors above. We must disallow neighbor 2 to avoid generating the previous node as neighbor.

Remember that, to calculate the priority function of the open list, you need to add the cost to reach the current node with the value of heuristic function. To solve the puzzle from a given search node on the priority queue, the total number of moves we need to make (including those already made) is at least its priority, using either the Hamming or Manhattan priority function. (For Hamming distance, this is true

because each block that is out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each block must move its Manhattan distance from its goal position. Note that we do not count the blank square when computing the Hamming or Manhattan priorities. Consequently, when the goal board is dequeued, we have discovered not only a sequence of moves from the initial board to the goal board, but one that makes the fewest number of moves (can you prove this fact?).

Detecting unsolvable puzzles: Not all initial boards can lead to the goal board by a sequence of legal moves, as the following board,

8	1	2
	4	3
7	6	5

To detect such situations, use the fact that boards are divided into two equivalence classes with respect to the reachability; those which lead to the goal board and those which cannot lead to the goal board. Moreover, we can identify in which equivalence class a board belongs without attempting to solve it if we use inversion. Given a board, an inversion is any pair of blocks 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).

If the grid size k is an odd integer, then each legal move changes the number of inversions by an even number. Thus, if a board has an odd number of inversions, then it cannot lead to the goal board by a sequence of legal moves because the goal board has an even number of inversions (zero). The converse is also true, if a board has an even number of inversions, then it can lead to the goal board by a sequence of legal moves.

If the grid size k is an even integer, puzzle instance is solvable if

- the blank is on an even row counting from the bottom (second-last, fourth-last, etc.) and number of inversions is odd.
- the blank is on an odd row counting from the bottom (last, third-last, fifth-last, etc.) and number of inversions is even.
- For all other cases, the puzzle instance is not solvable.

Courtesy:
How to check if an instance of 8 puzzle is solvable? - GeeksforGeeks
How to check if an instance of 15 puzzle is solvable? - GeeksforGeeks

.

Tasks:

1. Implement A* search algorithm using the two mentioned heuristics. Your first input will be the grid size k. Your second input will be the initial board position. Represent the blank using an asterisk '*'.
2. Print the solvability of the input initial board state. If the initial board state is solvable,
   a. Show the optimal cost to reach the goal state and the steps. You can show the board states at command prompt. Use your best judgement to decide your output layout.
   b. For two different heuristics, print number of explored nodes and expanded nodes.