Lab 2 & 3

1. To model a state, we should:
    - indicate the positions of the values and the empty cell, as well as the neighborhood relations, by using for example:
        - a vector and a structure to retain the neighbors
          OR
        - a matrix
    - a representation for the last moved cell (useful in: *"După mutarea unei celule, ea nu mai poate fi mutată din nou decât după ce unul din vecinii săi a fost mutat"*) for example:
        - the position of the last moved cell (or custom dummy values if the state is an initial state)
2. Obs: multiple final states!
3. The transition parameters should take into account:
    - the moved cell (or the moved empty cell)
    - the direction of movement (up, down, right, left)

   The validation(s) should consider that:
    - the moved cell can be moved in that direction
    - the moved cell cannot exit the grid
    - the moved cell was not previously moved (*"După mutarea unei celule, ea nu mai poate fi mutată din nou decât după ce unul din vecinii săi a fost mutat"*)
4. IDDFS (Iterative Deepening Depth First Search)
    - Combines the space efficiency of DFS with the fast search of states near the current state of BFS
    - DFS executed in a BFS manner

   Example of pseudocode:

```
def IDDFS(init_state, max_depth):
  for depth from 0 to max_depth:
      visited = []
      sol = depth_limited_DFS(init_state, depth, visited):
      if sol is not None:
          return sol
  return None

def depth_limited_DFS(state, depth, visited):
     if is_final(state):
        return state
     if depth == 0:
        return None
     visited.add(state)
     for each neighbor of state: #transition & validation(s)functions
         if is_valid(neighbor) and neighbor not in visited:
            res = depth_limited_DFS(neighbor, depth-1, visited)
            if res is not None:
               return res
      return None
```

5. Greedy (best-first):
    - Evaluate **all** unexplored states accessible from the current state
    - Select the unexplored state closer to the goal (the heuristic value indicates the closeness to the goal).

   Examples of other heuristics: http://www.ieee.ma/uaesb/pdf/distances-in-classification.pdf

   To measure the closeness between the current state and the goal, we can take into account either one final state (e.g., the one that has the empty cell on the same position) or all the final states. To measure closeness between a current state and all the final states we can add up/ average all heuristic scores measuring the closeness between that current state and each final state.

   Example of pseudocode Greedy best-first:

```
def greedy(init_state):
    pq = priorityQueue() #ordered by heuristic value
    pq.insert( (init_state, heur_val(init_state)) )
    visited = [init_state]

    while pq is not empty:
         state = pq.pop() #state with the best heuristic value
         pq.remove(state)

         if is_final(state):
             return state
         for each neighbor of state: #transition & validation(s)functions
             if is_valid(neighbor) and (neighbor not in visited):
                pq.insert( (neighbor, heur_val(neighbor)) )
                visited.add(neighbor)

    return None
```

6. The solution length = number of moves made to reach that solution from the initial state (e.g., you can use a structure to retain the "parent" of each state or just count the moves while passing through the states)

7. A* algorithm:
    - Combines Dijkstra + a heuristic
    - At each step we consider the state S, such that the length of the path from the initial state to the goal, which passes through S, is minimum.
        - f[S] = d[S] + h(S) (the length of the path)
        - d[S] = distance from the initial state to S (updated according to Dijkstra's algorithm)
        - h(S) = heuristic function approximating the distance from S to the goal
    - To find the shortest path from an initial state, A* needs an admissible heuristic
    - "An admissible heuristic never overestimates the distance between a state and the goal."
    - A consistent heuristic satisfies: h(A) <= dist(A, B) + h(B) if B is reachable from A,

where:

- h(X) = distance from state X to the goal
- dist(X,Y) = distance between X and Y (e.g., we can consider it being the number of moves to reach Y from X). A consistent heuristic is also admissible.

General pseudocode for A*:

```
def A_star(init_state):
    came_from = {}

    d = {}
    d[init_state] = 0

    f = {}
    f[init_state] = h(init_state)

    pq = priorityQueue() #ordered by f
    pq.insert((init_state, f[init_state]))

    while pq is not empty:
         state = pq.pop() #state with the minimum f value
         pq.remove(state)

         if is_final(state):
             return reconstruct_path(state, came_from)

         for each neighbor of state: #transition & validation(s)functions
             if is_valid(neighbor) and
               ( neighbor not in d or
                 d[neighbor] > d[state] + dist(neighbor, state) ):


                d[neighbor] = d[state] + dist(neighbor, state)
                f[neighbor] = d[neighbor] + h(neighbor)
                came_from[neighbor] = state
                pq.insert((neighbor, f[neighbor]))

    return None
```

Observation:

- The current problem contains multiple final states -> We can define the heuristic function to take into account all the final states

Other observations:

- Incomplete solutions for a subpoint: at most 0.1 points. (example of an incomplete solution: not taking into account the condition: *“După mutarea unei celule, ea nu mai poate fi mutată din nou decât după ce unul din vecinii săi (initiali) a fost mutat”*)
- Subpoints 4,5,6,7 use the state representations and the functions from 1,2,3.
- (NEW) It will take a considerably longer time for IDDFS to reach the final state(s) if these are far away from the initial state or if the solution does not exist.
- (NEW) Don’t forget this condition: “să se găsească, dacă există, o secvență de mutări ale celulelor astfel încât toate să fie plasate în ordine crescătoare în matrice”, for which you have an idea of implementation in the A* pseudocode (building a dictionary to retain the parent of each state).