Artifficial Intelligence assignment: 8-puzzle solving

Javier del Olmo Sánchez

In the code, in line 92 we can manage the start state for the game. It is possible to write any valid board you want (it should include every number between 0 and 8, both included, and not more, as it is a 3x3 board). It can occur that the board is not solvable, and in that case the console will give you a message telling you that. I have checked whether the puzzle is solvable or not by checking in an individual function if the number of inversions is even or odd. I searched in the Internet about the game and saw that if the number of inversions is even, the puzzle is solvable, so if it is odd it won´t be solvable.

I have defined some small functions like *misplaced_tiles_heuristic*, which calculates the number of misplaced tiles in the board (the heuristic); or the function *is_solvable*, which works as I mentioned before.

Then, I have my main function *a_star_8_puzzle*, which does the fully function of the application. First of all, this function checks if the start state is solvable or not. Then, I created a mapping of each tile value in the goal state to its position (row, column). The *goal_positions* dictionary is used to quickly look up the position of any tile in the goal state without having to iterate over the goal board each time. I flatten the board and find the position of the empty space for knowing which is the tile I am going to move. I create after that a set() for the visited states, ensuring that none repeated state is stored. Then, it is necessary to use *heapq* to always expand the node with the lowest f(x). This is how the tree structure is implemented, as we use the heap to store the states as a priority queue. In the following line I have defined the different possible movements for the empty space in the board and after it I start the count for the states generated to 0.

In this moment starts the main loop of the program, where we extract the current state with the lowest f(x) from the priority queue. I add the current state of the board to the set of visited ones. After that, I check if the current board is the same as the goal board we have to achieve. If the goal state is reached, we reconstruct the solution path by using the parent pointers of the tree. It is important to make the inversion of the path after reconstructing it, because when we travel through the parent pointers we are going in the opposite direction.

Next, I implemented a for loop which is going to create a new board configuration for each valid movement. Firstly, it obtains the new position of the empty space when performing the movement, and then check if the new position is inside the board, for only continuing with the valid ones. In line 75, the new position of the empty space is swapped with the neighbour tile. The next step is to find if the new

configuration of the board has been visited or not, because we don ´t want to repeat any state. If it hasn ´t been visited yet, we continue with the code and add it to the visited set. The new h(x) and g(x) are obtained and stored by creating a new puzzle state. As the new state wasn ´t visited before, it is added to the priority queue and the count for the generated states is incremented in one unit.

Starting in line 105 I wrote everything needed to show the solution to the assignment in the console. First are the lines for the UCS solution and after that the lines for writing the solution of the A* algorithm. If the puzzle has no solution, a message indicating that is shown in the console. If it has a solution, the output will be the start state and the goal state, followed by the sequence of actions from start state to goal state. Then I print the number of stated generated during the algorithm run and a string of the capital letters of the sequence of moves done to reach the solution.

In summary, the fringe is implemented using a priority queue (with the heapq). Each value of the heap is a tuple (cost (f(x)), state). In each iteration of the while loop, the state with the lowest f(x) is popped from the heap using the heappop. This is the current state to be expanded. The new states are added to the heap if the performed movement is valid and the state wasn ´t visited before, using heappush.

The tree structure for the search process is implicitly represented using the PuzzleState class and its parent attribute. This structure allows the program to trace the sequence of moves and reconstruct the path from the initial state to the goal state once the solution is found.

Results of the 2 settings with the start and goal states in the slides:

```
Start state
3 1 2
4 7 5
6 8 0

Goal state
0 1 2
3 4 5
6 7 8

UCS Solution:
3 1 2
4 7 5
6 8 0

3 1 2
4 7 5
6 0 8

3 1 2
4 0 5
6 7 8

3 1 2
0 4 5
6 7 8

0 1 2
3 4 5
6 7 8

States generated: 39
```

```
Sequence of moves: LULU
Number of moves: 4

A* with Misplaced Tiles Heuristic Solution:
3 1 2
4 7 5
6 8 0

3 1 2
4 7 5
6 0 8

3 1 2
4 0 5
6 7 8

3 1 2
0 4 5
6 7 8

0 1 2
3 4 5
6 7 8

States generated: 9
Sequence of moves: LULU
Number of moves: 4
```