Intelligent Systems

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Lab assignment 1:

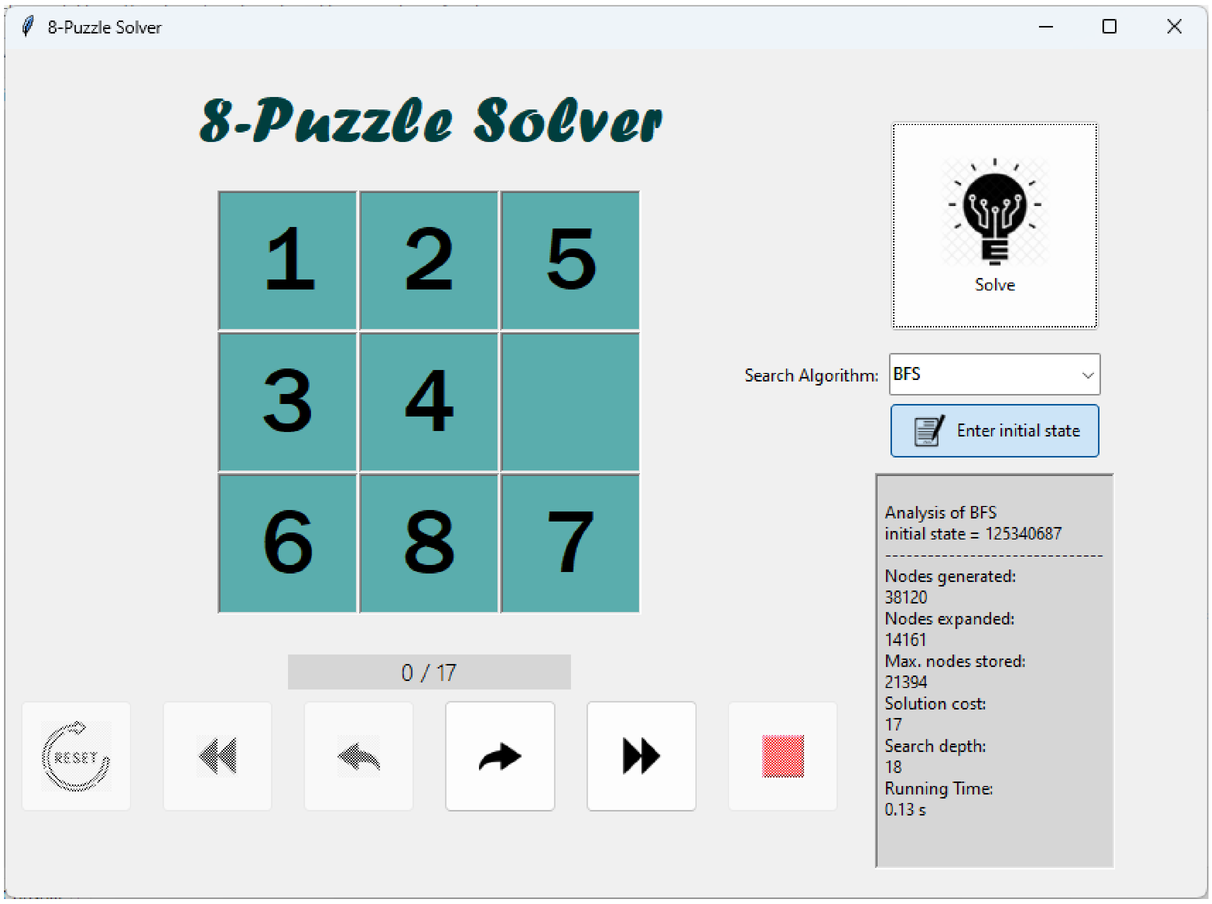
"Problem Solving Based on State Space Search"

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

1. Introduction

The aim of this lab assignment is to evaluate the efficiency and, temporal and spatial cost of different search strategies applied to the 8-puzzle problem. To this purpose, a solution of the 8-puzzle problem implemented in Python, named *puzzle*, is provided. To install and run *puzzle*, please follow the following steps:

1. Copy the **puzzle.zip** file that is available in PoliformaT, in the folder *Resources/2024-2025/ ENGLISH /LAB ASSIGNMENT 1* to your local machine.
2. When you unzip puzzle.zip, a puzzle directory will be created where the source files are located.
3. From a Python development environment (i.e., **Spyder/Visual Studio Code)**, available from the lab local installation, or from a terminal (if you have Python in your PATH), run the program **interface.py**.
4. Press the button "**Enter initial state**" and type the initial state as a sequence of numbers where the blank tile is represented with a zero. For example, the initial state in Figure 1 is the sequence 125340687.
5. Select the search strategy in the drop-down menu '**Search Algorithm'**.
6. If the search strategy requires a maximum depth, a pop-up window is shown to type the depth.
7. Press the button '**Solve'.**
8. The information related to the search process is shown, and the sequence of movements to reach the goal state are available by pressing the buttons below the puzzle.



Search Algorithm: BFS

```
Analysis of BFS
initial state = 125340687
------------------------------
Nodes generated:
38120
Nodes expanded:
14161
Max. nodes stored:
21394
Solution cost:
17
Search depth:
18
Running Time:
0.13 s
```

Figure 1 Dashboard for puzzle

To compare the different search strategies , the same goal state will be used:

1	2	3
8		4
7	6	5

Remark 1: If the goal state cannot be reached from the initial state (the state space of the puzzle problem contains two connected components), the message “**Cannot solve**” will be shown. This means that the goal state cannot be reached by any sequence of movements (up, down, left , right).

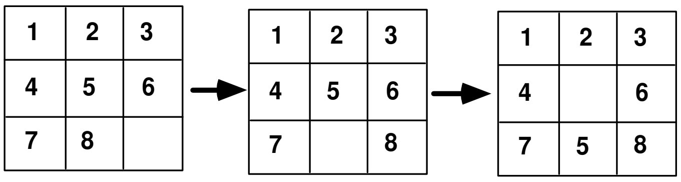
Remark 2: For those strategies that require to define a maximum depth (i.e. DFS), the message “**The state you entered is unsolvable**” is shown when the goal state is found at a deeper level than the maximum depth defined.

2. General aspects of the 8-puzzle problem

This section summarises the most important aspects of the 8-puzzle problem.

2.1. Representation

The 8-puzzle problem is represented on a 3x3 matrix with eight tiles from 1 to 8 that can be slided horizontally and vertically. There is a single blank (empty) tile that allows the movement of the other eight tiles. The total number of different states is 9! = 362.880 equally distributed in two connected state subspaces. The following is an example of sequence of movements:



The solution of the 8-puzzle problem provides the sequence of movements of the blank tile to go from the initial state to the goal state defined above.

2.2. Movements

Those of the blank tile (up, down, left and right). The number of possible movements from each state is four when the blank tile is in the center, two when the blank tile is in the corners and three, for the rest of states.

3. Search strategies

The search strategies implemented in *puzzle* are:

BFS. Breadth-First Search.

DFS (Graph Search). Depth-First Search with graph search (with closed set).

DFS (Bactracking). Depth-First Search with backtracking. In contrast to DFS with graph search, or DFS with tree search where the Open set stores all children for each level in the path being explored, DFS with backtracking only stores the nodes in the path.

Voraz (Manhattan). Greedy Search where `f(n) = h(n)` being `h(n)` the Manhattan distance. The Manhattan distance of a puzzle state is computed as the sum of horizontal and vertical distances in the 3x3 matrix of each tile (except for the blank tile) from its current position to its position in the goal state.

ID. Iterative Deepening Search, DFS with backtracking where the maximum depth is increased by 1 at each iteration.

A* Manhattan. A* Search where `f(n) = g(n) + h(n)` being `h(n)` the Manhattan distance.

A* Euclidean: A* Search where `f(n) = g(n) + h(n)` being `h(n)` the Euclidean distance. The Euclidean distance of a puzzle state is computed as the sum of Euclidean distances in the 3x3 matrix of each tile (except for the blank tile) from its current position to its position in the goal state.

IDA* Manhattan. Iterative Deepening A* Search where the search is bounded by a given value of the function `f` defined by `f(n) = g(n) + h(n)` being `h(n)` the Manhattan distance. At each iteration the current bound is updated with the minimum `f` value of those nodes exceeding the current bound.

IMPORTANT REMARK: DFS strategies require a maximum depth. If the depth at which the optimal solution is found is greater than the maximum depth explored by DFS, then DFS will not be able to find the solution.

4. Incorporating a new seach strategy

The incorporation of a new seach strategy involves modifying the following source files.

4.1 File 'interface.py'

- Adding the name of the new search strategy to the drop-down menu of search strategies in function `__init__(self, master=None)`, lines 88-89 (aprox.):

```
self.algorithmbox.configure(cursor="hand2", state="readonly",
                            values=('BFS', 'DFS (Graph Search)', 'DFS
                            (Backtracking)', 'Voraz (Manhattan)', 'ID', 'A*
                            Manhattan', 'A* Euclidean', 'IDA* Manhattan'))
```

- If the search strategy requires a maximum depth, the name of the new search strategy must be added to the list of functions in function `selectAlgorithm`, line 248 (aprox):

```
  if algorithm in ['DFS (Graph Search)', 'DFS (Backtracking)']:
   cutDepth = int(simpledialog.askstring('Cut depth', 'Please, enter your max
             depth'))
```

- Invoke the search function. All search strategies **without backtracking** use the same call `main.graphSearch` but with different parameters. The modification should be made in the function `solveState` (line 381, aprox) adding a new case for the new search strategy invoking the function `main.graphSearch`. For example, A* Manhattan:

```
elif str(algorithm) == 'A* Manhattan':
     main.graphSearch(initialState,main.function_1,main.getManhattanDistance)
     path, cost, counter, depth, runtime, nodes, max_stored,memory_rep = \
         main.graphf_path, main.graphf_cost, main.graphf_counter,
                  main.graphf_depth,     main.time_graphf,     main.node_counter,
main.max_counter,
          main.max_rev_counter
```

The third line above receives the results and is the same for all search strategies without backtracking. The parameters of the function `graphSearch` are the following:

- `initialState:` Initial state of the puzzle problem
- `main.function_1:` Function computing g(n). There are three predefined functions:
  - `function_1:` It returns 1, this is default cost for a movement in the puzzle problem.
  - `function_0:` It returns 0, it is used in the greedy strategy where g(n) = 0
  - `function_N:` It return -1, It is used in DFS.
- `main.getManhattanDistance:` Function computing h(n). The function `getEuclideanDistance` is also defined, and new heuristic functions could be defined.
- `CutDepth:` Maximum depth in the search. It is an optional parameter not shown in the example above, that is used by those DFS strategies that need it.

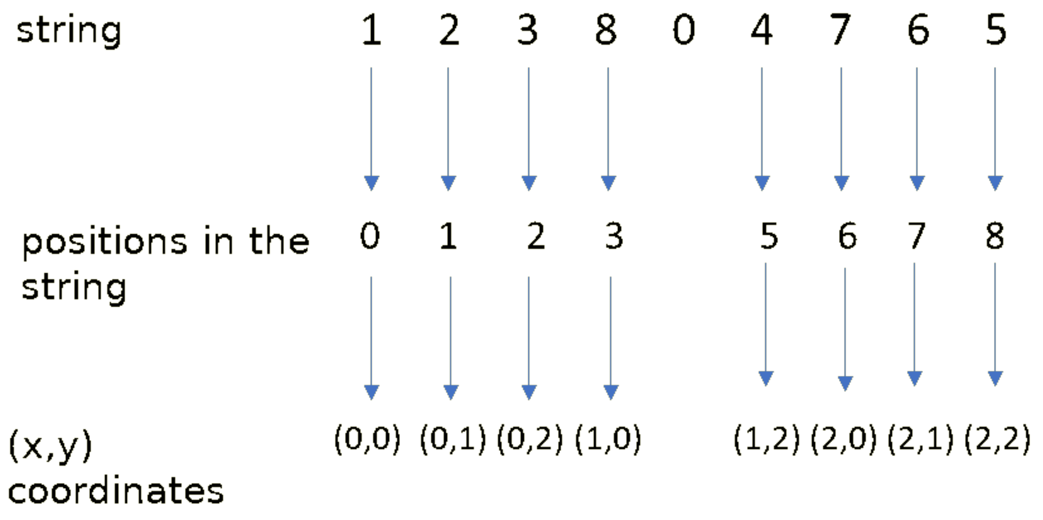
4.2 File 'main.py'

New heuristic functions must be included in this file. As an example, the function `getManhattanDistance` is already included in this file:

```
def getManhattanDistance(state):   # current state to be evaluated as a string
  tot = 0
  for i in range(1,9):                     # loop over the 8 non-blank tiles
    goal = end_state.index(str(i)) # position of the i-th tile in the goal state (end_state)
    goalX = int(goal/ 3)  # Computing x coordinate (row) in a 3x3 matrix of the goal state
    goalY = goal % 3       # Computing y coordinate (column) of the goal state
    idx = state.index(str(i))   # position of the i-th tile in the current state
   itemX = int(idx / 3)          # Computing x coordinate (row) of the current state
    itemY = idx % 3               # Computing y coordinate (column) of the current state
   tot += (abs(goalX - itemX) + abs(goalY - itemY)) # Manhattan distance
 return tot
```

The function has the parameter `state` that is the current state (in string format) to be evaluated. For each of the non-blank tiles, the distance is computed from the current state of this tile to its goal state (represented by the variable `end_state`). The rest of the operations compute the (x,y)[1] coordinates in a 3x3 matrix of each tile in the current and goal state from the lineal representation as a string.

For example, for the goal state (`end_state`), the translation from lineal representation (as a string) to matrix would be:



[1]Please bear in mind that (x,y) coordinates correspond to the (row, column) cell in the 3x3 matrix.

LAB ASSIGNMENT

In this section, the lab assignment on the 8-puzzle problem to be carried out (in pairs or individually) is described.

This assignment consists in the implementation of three heuristic functions and to answer a series of questions. In the lab exam, the students will have to upload to the exam assignment the following files:

- The Python code implementing each of the three heuristic functions
- A text file with the answer to the questions stated below

In case of working in pairs, both members of the team will upload the same files to their corresponding exam assignment. **IMPORTANT: The text file with the answer to the questions must include the name of both members of the team.**

1. Implement the heuristic function **MISPLACED TILES**. The cost of a state n is the number of misplaced (non-blank) tiles with respect to the goal state.

2. Implement the heuristic function **ROWS_COLUMNS** (h(n)=RowCol(n)) for an A search strategy. This heuristic function will work as follows: for each (non-blank) tile on the board, if the tile is not located in the correct row (with respect to the goal state), add 1 to the heuristic function; if the tile is not located in the correct column, add 1. Therefore, the minimum value for a tile is zero (when the tile is located at the same position as in the goal state), and the maximum value for a tile is two (when neither the row nor the column are those of this tile in the goal state).

3. Implement the heuristic function **SEQUENCES** that is described as follows:

   SEQUENCES: h(n) = 3*S(n) where S(n) is the sum of the cost for each tile. To compute the cost of each tile bear in mind that in the goal state

1	**2**	**3**
8		**4**
7	**6**	**5**

   the tile i is clockwisely followed by the tile i+1 (except for the tile 8 that is followed by 1) when moving over the outer tiles in the 8-puzzle. Formally, the next tile in the sequence is i % 8 + 1 . Taking this into account, the cost of each tile is defined as follows:

   - If the tile i is not followed by tile i % 8 + 1 , the cost is 2, zero, otherwise.
   - If there is a non-blank tile in the center, the cost is 1.
   - If the blank tile is not located in the center, the cost is 2, zero, otherwise.

Let us consider the following state n

2	8	3
1	6	4
7		5

Function S(n) is computed as follows:

Tile 2: Add 2 because it is not followed by 3
Tile 8: Add 2 because it is not followed by 1
Tile 3: Add 0 because it is followed by 4
Tile 4: Add 0 because it is followed by 5
Tile 5: Add 2 because it is not followed by 6
Tile 0 (blank): Add 2 because it is not in the center
Tile 7: Add 2 because it is not followed by 8
Tile 1: Add 0 because it is followed by 2
Tile 6: Add 1 because it is a non-blank tile in the center.

Then, S(n) = 11, and h(n) = 3*S(n) = 33.

4. Answer the questions that you will find at the end. To this purpose, it is recommended to test a set of initial states for each of the search strategies available in the program, including those three new heuristic functions implemented by you (Misplaced, RowCol, Sequences). For example, for the following two initial states

	2	3
8	4	5
7	1	6

7	8	1
4		6
2	3	5

fill the following table

	BFS	DFS (GS)	DFS (Back)	Greedy	ID	A* Manh	A* Euclid	IDA* Manh	A* Mispl	A* RowCol	A* Sequen
Nodes generated											
Nodes expanded											
Max nodes stored											
Solution cost											
Search depth											
Running time											

and analyse the results of each search strategy to answer the following questions:

- Is the heuristic function Sequences admissible? Why?
- Is the heuristic function RowCol admissible? Why?
- Compare the search strategies using the heuristic function Manhattan with Sequences explaining which of the two provides better solutions in terms of number of movements and nodes involved to reach the goal state.
- Do the same as in the previous question with the heuristic functions Misplaced and RowCol.