INTELLIGENT SYSTEMS

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA INFORMÀTICA

Lab assignment 1:

"Problem solving based on state-space search"

DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ
UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CONTENTS

1. Introduction

The aim of this lab assignment is to evaluate the efficiency, the time cost and the space cost of several search strategies when applied to the n-puzzle problem. To this purpose, a library "search.py" is provided implementing several algorithms that we have studied in our theory lessons, as well as a number of support methods for the execution and evaluation of the proposed solutions. Several Jupyter notebooks are also provided, in which a set of exercises is proposed.

2. General aspects of the puzzle game

This section briefly summarizes the main features of the puzzle game.

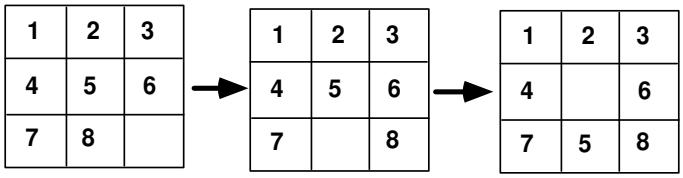
2.1. Representation

The puzzle game is represented on a board made up of n*n squares.

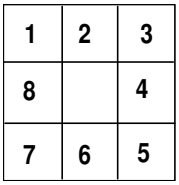
Focusing on a specific example, let us imagine an instance of the game where n=3, and so, there are 9 squares. 8 of the squares contain a tile which can be slid horizontally and vertically. Each tile is numbered from 1 to 8. There is an empty square in the board which allows the tiles to be moved.

The total number of states or configurations of the problem that can be generated in this puzzle game is 9! = 362 880 states.

Here is an example of a sequence of movements:



The **goal** of the problem is to obtain, as a solution, the sequence of "movements of the empty square" that brings us from the initial state to the goal state:



2.2. Operators

The operators in the puzzle game represent the "movements" of the empty square. These movements are: up, down, right, left. Note that any 4 of the movements is possible when the empty square is in the centre of the board; when the empty square is in a corner, then 2 of the movements are possible; in any other case, 3 of the movements are possible.

3. Search strategies

The search strategies implemented in the *puzzle* program are:

BFS. The Breadth-First Search strategy uses $f(n)=g(n)$ for node expansion, where $g(n)$ is the cost factor linked to the depth of node n in the search tree.

DFS (Backtracking). This strategy implements Depth-First Search with backtracking. To this end, it uses the function $f(n)=-g(n)$, where $g(n)$ is the cost factor linked to the depth of node n in the search tree. Unlike DFS (Graph Search), this strategy only keeps in memory the explored nodes that belong to the current path (PATH list); the rest of the explored nodes are not kept in the CLOSED list, but are deleted instead.

Greedy (Manhattan). This is a strategy that implements a *greedy algorithm* following the function $f(n)=D(n)$, where $D(n)$ is the Manhattan distance.

ID. This strategy implements an Iterative Deepening search. It involves performing successive depth-first searches with backtracking until the solution is reached. Each new search increases the depth of the exploration by 1.

A* Manhattan. This is an A-type search that uses the Manhattan distance as a heuristic; node expansion is performed according to the function $f(n)=g(n)+h(n)$, where $g(n)$ is the cost factor corresponding to the depth of node n in the search tree and $h(n)=D(n)$ is the Manhattan distance, i.e., the sum of the horizontal and vertical distances of each puzzle tile from its current position to the position it will take in the goal.

A* (MD + LC). This strategy implements an A-type search that uses an informed heuristic based on the combination of the Manhattan distance and linear conflicts (LC). Node expansion is performed using the function $f(n)=g(n)+h(n)$, where $g(n)$ is the cost associated with the depth of node n and $h(n)=D(n)+LC(n)$, where $D(n)$ is the Manhattan distance and $LC(n)$ is the number of conflicts between misplaced tiles in the same row or column that must reverse their position with each other. This heuristic improves accuracy over A* Manhattan, as it adds penalties for conflicts that involve additional necessary movements.

IDA* Manhattan. This search strategy implements an Iterative Deepening A* search where the search is bounded by the f-value, i.e., by the function $f(n)=g(n)+h(n)$, where $g(n)$ is the cost factor and $h(n)=D(n)$ is the Manhattan distance. The bound of an iteration i of the IDA* algorithm is the smallest f-value of any node that exceeds the bound in the previous iteration $i-1$.

IMPORTANT NOTE:

The DFS (Backtracking) strategy works with a maximum depth. If the current maximum depth is less than the depth of the optimal solution, then the strategy will not find a solution.

4. Description of the code provided

4 files are provided that make up the setup necessary to carry out the assignment:

1. **`search.py`**
   This Python source file contains the implementation of several classic search algorithms (such as BFS, DFS, A*, IDA*) and heuristics (such as Manhattan). It also includes auxiliary functions to visualize states, display solution paths, and compare results between algorithms.
2. **`puzzle-p1.ipynb`**
   Jupyter Notebook introducing the search algorithms applied to the puzzle problem. Its goal is to familiarize users with the use of the file search.py, the execution of search strategies, and the visualization of solutions.
3. **`puzzle-p2.ipynb`**
   Jupyter Notebook focused on how to implement and test additional heuristics, as well as on comparing heuristics. It is the main notebook for developing and validating the additional heuristic functions expected as student work in this lab assignment.
4. **`alg-compare-p3.ipynb`**
   More advanced Jupyter Notebook aimed at making detailed comparisons between heuristics. It includes graphical visualization and quantitative analysis to support the justification of results.

4.1 Detailed description of “search.py”

The file search.py contains the implementation of several classic search algorithms along with some auxiliary methods. **This file is not to be modified**.

The methods that can be used during this lab assignment are described below:

Provided heuristic functions

`getManhattanDistance(state, end_state)`

Computes the **total Manhattan distance** between the current state and the goal state. It is an admissible heuristic for n-puzzle-type problems.

- **Input parameters**:
    - `state`: list representing the current state of the board (e.g., `[1, 2, 3, 4, 5, 6, 7, 8, 0]`).
    - `end_state`: list with the goal state.
- **Returns**:
  An integer representing the sum of the Manhattan distances of all tiles (except 0/empty) between their current position and the goal position.

`getLinear_conflict(state, end_state)`

Computes the **number of linear conflicts** in rows and columns between state and end_state. A linear conflict occurs when two tiles are in the same row or column as their destination, but in the wrong order.

- **Input parameters**:
    - `state`: list representing the current state of the board.
    - `end_state`: list representing the goal state.
- **Returns**:
  An integer representing the total number of linear conflicts detected.

`get_md_plus_linear_conflict(state, end_state)`

Combines two heuristics: **Manhattan distance** and **linear conflicts**. It is a more informed heuristic than Manhattan alone, and it remains admissible.

- **Input parameters**:
    - `state`: list with the current state,
    - `end_state`: list with the goal state.
- **Returns**:
  An integer: `getManhattanDistance + 2 * getLinear_conflict`.

Functions for interpretation and visualization

`visualize_state(state)`

Displays on the command line the board state in matrix format for easy human reading.

- **Input parameters**:
  - `state`: list of integers representing the state of the puzzle.
- **Output**: Prints the board on the command line.

`show_path_for_algorithm(results, algorithm_name, size)`

Searches a dictionary of results for the information of a specific algorithm and displays its solution path.

- **Input parameters**:
  - `results`: a dictionary associating the name of each algorithm with its results.
  - `algorithm_name`: name of the algorithm the result of which we want to display.
  - `size`: board dimension.
- **Output**: Prints on the command line the steps of the corresponding solution.

`plot_algorithm_comparison(results)`

Plots a comparison of algorithms based on execution time, nodes generated, cost, etc.

- **Input parameters**:
  - `results`: dictionary with results for several algorithms (see the standard structure of the dictionary `results` in `run_algorithm`).
- **Output**:
  Plot generated using `matplotlib`.

`show_results(results)`

Prints on the command line a summary of the results obtained by each algorithm (time, depth, nodes, etc.).

- **Input parameters**:
  - `results`: dictionary with the execution results for several algorithms (see the standard structure of the dictionary `results` in `run_algorithm`).
- **Output**:
  Text table with the comparative data.

Generating puzzle states

`generate_random_state(size, end_state)`

Generates a **valid (solvable) random state** of the puzzle, based on the goal end state. This function applies random moves on the goal state without repeating states, ensuring that the result is achievable.

- **Input parameters**:
  - `size`: board dimension (e.g., `3` for an 8-tile puzzle).
  - `end_state`: list representing the goal state (e.g., sorted from 1 to n^2-1 and 0 at the end).
- **Returns**:
  List of integers representing a new state of the puzzle, random but solvable.
- **Example of use**:

```
goal = generate_ordered_state(3)
initial = generate_random_state(3, goal)
```

`generate_ordered_state(n)`

Creates the **standard ordered state** of the puzzle, where the tiles are arranged from smallest to largest, with the empty square (`0`) in the last position.

- **Input parameters**:
  - `n`: board dimension (e.g., `3` for a 3x3 board).
- **Returns**:
  List of integers of size `n * n`, ordered (e.g., `[1, 2, 3, 4, 5, 6, 7, 8, 0]` for `n=3`).

`generate_spiral_state(n)`

Generates a goal state **with spiral ordering (snail mode)**, useful as an alternative to the classic order. The tiles are arranged in a spiral from the top left corner to the centre.

- **Input parameters**:
  - `n`: board dimension (`n x n`).
- **Returns**:
  List of integers of size `n * n`, arranged in spiral order, ending with `0` at the centre (if `n` is odd).
- **Return example for `n=3`**: (actually returns a list)

```
[1, 2, 3,
 8, 0, 4,
 7, 6, 5]
```

Additionally, the notebooks provided include two methods that facilitate the execution of algorithms and which can be modified to add new heuristics:

Auxiliary methods for the execution of algorithms

`run_algorithm(algorithm_name, initial_state, end_state, size, depth=50, heuristic_func=None)`

This method executes a single search algorithm specified by its name, on an initial state and a goal state of the puzzle. It returns a dictionary with the information of the solution found (or empty if there is no solution).

- **Input parameters**:
  - `algorithm_name`: text string with the name of the algorithm to be executed (see supported algorithm list below).
  - `initial_state`: list representing the initial state of the puzzle.
  - `end_state`: list representing the goal state.
  - `size`: board size (e.g., `3` for 3x3 puzzles).
  - `depth`: maximum depth (only used in some algorithms, such as bounded DFS-B).
  - `heuristic_func`: optional heuristic function (**not used in this implementation**; heuristics are integrated according to the algorithm name).
- **Returns**:
  Dictionary including a single key (the name of the algorithm), having as a value a dictionary with:
  - `'path'`: list of states from the initial state to the goal.
  - `'cost'`: total cost of the solution.
  - `'depth'`: depth reached.
  - `'nodes_generated'`, `'nodes_expanded'`, `'time'`, `'max_nodes'`: performance metrics.
- **Supported algorithms**:
  - `'BFS'`: Breadth-first search (uniform cost without heuristics).
  - `'DFS-B'`: Bounded depth-first search (specific version with recursive control).
  - `'Voraz (Manhattan)'`: Greedy search using the Manhattan distance as a heuristic.
  - `'ID'`: Iterative deepening depth-first search (IDDFS).
  - `'A* (Manhattan)'`: A* algorithm using Manhattan heuristics.
  - `'A (MD + LC)'`: A* with improved herutistics, combining Manhattan and linear conflicts.
  - `'IDA* (Manhattan)'`: Iterative deepening A* with Manhattan heuristics.

`run_all_algorithms(initial_state, end_state, size, depth=50)`

This method **automatically** runs **all supported algorithms** on the same initial and goal states. It is used to compare their results.

- **Input parameters**:
  - `initial_state`: list representing the initial state of the puzzle.
  - `end_state`: list representing the goal state
  - `size`: board dimension.
  - `depth`: maximum depth (used in bounded algorithms such as DFS-B).
- **Returns**:
  Dictionary that associates the name of each algorithm with its corresponding result (same structure as that returned by `run_algorithm`).
- **Algorithms executed:**
  - `'BFS'`
  - `'DFS-B'`
  - `'Voraz (Manhattan)'`
  - `'ID'`
  - `'A* (Manhattan)'`
  - `'A (MD + LC)'`
  - `'IDA* (Manhattan)'`

Representation of the states and execution of the algorithms

The states of the puzzle are represented as n x n matrices, which are converted into one-dimensional lists to be processed by the algorithms. Below is an example of code for a 3x3 puzzle:

```
size = 3  # 3x3 puzzle

# Initial and goal states as matrices
initial_state_matrix = [
    [7, 8, 1],
    [4, 0, 6],
    [2, 3, 5]
]

end_state_matrix = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]
]

# Conversion into one-dimensional lists
initial_state = [num for row in initial_state_matrix for num in row]
end_state = [num for row in end_state_matrix for num in row]
```

This code defines the size of the puzzle as 3x3 (`size = 3`) and represents both the initial state and the goal state in the form of matrices. It then converts these matrices into one-dimensional lists (`initial_state` and `end_state`), which is the format required by the search algorithms to process the puzzle states.

Next, the code checks whether the puzzle is solvable and executes all the algorithms:

```
results = {}
if isSolvable(initial_state, end_state):
    results = run_all_algorithms(initial_state, end_state, size, depth=40)
    show_results(results)
    plot_algorithm_comparison(results)
else:
    print("The puzzle is not solvable.")
```

This code fragment checks whether the initial state of the puzzle can be transformed into the goal state, using the function `isSolvable`. If so, it runs all search algorithms using `run_all_algorithms` and saves the results in the dictionary `results`. It then displays a summary of the metrics using `show_results` and a graphical comparison using `plot_algorithm_comparison`.

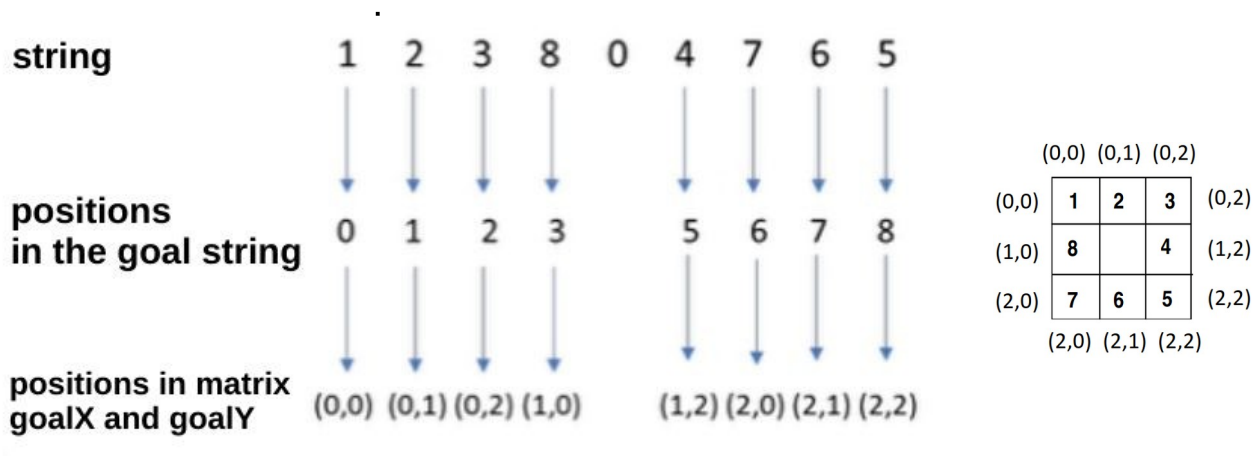
5. How to add a new heuristic

Let's suppose you want to add a function that implements an additional heuristic function. As an example, we explain here the `getManhattanDistance` function already included in the code:

```
def getManhattanDistance(state, end_state):  # state is the current state to be
                                             # evaluated, end_state is the goal state
  tot = 0
  size=int(math.sqrt(len(state)))
  for i in range(1, len(state)):      # goes through the n tiles
    goal = end_state.index(i) # position of tile i in the goal state (end_state)
    goalX = goal // size        # we turn the end_state from a string into a nxn matrix
    goalY = goal % size             # row (goalX) and column (goalY) of the tile
    idx = state.index(i)   # same operation but for the current row (idx) and column (idy)
    itemX = idx // size          # occupied by that tile in the current state
    itemY = idx % size
    tot += (abs(goalX - itemX) + abs(goalY - itemY)) # computing the distances
  return tot
```

The function receives in `state` the current state to be evaluated. For each one of the 8 tiles (the empty square is represented as 0, but it is not a tile), the distance from the current position of that tile to its position in the goal state (represented in the variable `end_state`) is computed. The intermediate operations are used to compute the positions of each tile in an nxn matrix (as in the puzzle) from the linear representation as a string.

For example, for the goal state (`end_state`), the transformation from the linear representation (string) to a matrix in a 3x3 puzzle would be:



Once the new heuristic function has been implemented, it would need to be added as a new algorithm in the `run_algorithm` method:

```
def run_algorithm(algorithm_name, initial_state, end_state, size, depth=50,
heuristic_func=None):

…


#############################################################################
#################

    # we add the new heuristic

#############################################################################
#################

    elif algorithm_name == 'New heuristic':
        solution_data = graphSearch(initial_state, end_state, lambda x: 1,
                        lambda s, es: new_heuristic(s, end_state), size)
        print("New heuristic executed")


#############################################################################
#################

…
```

Where `new_heuristic` is the name of the method in which the new heuristic has been implemented.

The `run_all_algorithms` method must also be modified to include the new algorithm, adding the label by which the new heuristic will be identified:

```
def run_all_algorithms(initial_state, end_state, size, depth = 50):

…

    algorithms = [
        'BFS',
        'DFS-B',
        'Voraz (Manhattan)',
        'ID',
        'A* (Manhattan)',
        'A (MD + LC)',
        'IDA* (Manhattan)',
        'New heuristic', ### WE ADD THE LABEL FOR THE NEW HEURISTIC TO THE LIST
    ]

…
```

To run the new heuristic by itself, you can execute this:

```
# assuming that initial_state, end_state and size have been initialized

result = run_algorithm('New heuristic', initial_state, end_state, size)

show_results(result)
```

We can also run the new heuristic together with the rest of the algorithms:

```
# assuming that initial_state, end_state, size and max_depth have been
# initialized

results = run_all_algorithms(initial_state, end_state, size, max_depth)
show_results(results)
```

WORK TO BE DONE BY THE STUDENT

This section outlines the task about the n-puzzle to be completed by each lab group (either one person or a group of two people maximum) must complete on the n-puzzle.

Students are asked to implement several heuristic functions and justify their behaviour. On the day of the lab exam, all students must upload their assignment:

- The Python code for each one of the heuristics (notebook files), including the answers and justifications for the questions asked.

In the case of a two-people lab group, both of them will upload the same files to their corresponding tasks. **NOTE: Do state clearly the names of both people in the text file containing the answers to the questions asked.**

1. Implement the **MISPLACED TILES** heuristic function seen in our theory lessons ($h(n)=W(n)$) for an A-type search.

2. Implement the **A* Euclidean** heuristic function: This is an A-type search that uses the Euclidean distance as a heuristic function; node expansion is performed according to the function $f(n)=g(n)+D(n)$, where $g(n)$ is the cost factor corresponding to the depth of node n in the search tree and $h(n)=D(n)$ is the Euclidean distance, i.e. the sum of the straight-line distances of each puzzle tile from its current position to the position it will occupy in the goal state (square root of the sum of the squares of the horizontal and vertical distances).

3. Implement the heuristic function **ROWS_COLUMNS** (h(n)=RowCol(n)) for an A-type search. This heuristic works as follows: for each tile on the board, if the tile is not in its correct goal row, 1 point is added; if the tile is not in its correct goal column, 1 point is added. Therefore, the minimum value of this heuristic for a tile is 0 (when the tile is correctly placed in its goal position), while the maximum value is 2 (when neither the row nor the column of the tile's position are equal to the value of the row and column in the goal position).

4. Implement the **A* Euclidean + LC** heuristic function

5. Implement a heuristic proposed by you as a combination of the above. It does not have to be A*. The idea is to try to propose a heuristic that you think works well.

6. For each heuristic, a comparative study with the other algorithms is required, including:

    - Determine which heuristics are A* and which ones are not. Justify your answers.
    - Propose examples to support the justifications for your answers.
    - Display results in tables such as the one below, where you can justify your answers using specific data.

Algorithm	Cost	Depth	Time (s)	Nodes Generated	Nodes Expanded	Max Nodes Stored
BFS	22	22	0.373731	233687	86871	110729
DFS-B	38	40	1.54931	2028383	1301341	41
Voraz (Manhattan)	22	22	0.000297546	82	30	54
ID	22	22	1.71706	2198820	1390965	23
A* (Manhattan)	22	22	0.00196576	704	266	429
A (MD + LC)	22	22	0.00277257	415	158	251
IDA* (Manhattan)	22	22	0.00259209	1409	882	23
Misplaced Tiles	22	22	0.0270152	17460	6452	10068
A* Euclidean	22	22	0.00872922	2770	1028	1648
Rows Columns	22	22	0.0149882	6371	2215	3914
A* Euclidean LC	22	22	0.0119114	1696	630	1019
Custom	22	22	0.454935	234938	87288	111261

7. A detailed analysis study comparing the two best heuristics you have identified using the notebook "alg-compare-p3.ipynb". Justify your answers.

Appendix. How to use Jupyter Notebook or Google Colab

Option 1: Using Jupyter Notebook (recommended if working locally)

1. Make sure that you have Python and Jupyter Notebook installed. You can install it with:

```
pip install notebook
```

2. Go to the folder where the lab assignment files are located (search.py, puzzle-p1.ipynb, etc.).

3. Run the following command in the command line:

```
jupyter notebook
```

4. The browser will open. Click on the .ipynb file you want to use (for example, puzzle-p1.ipynb).

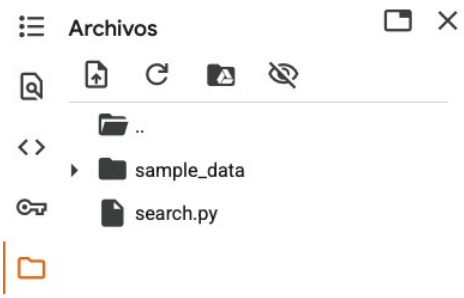
5. Run each cell with Shift + Enter.

Note: Make sure that the search.py file is in the same directory so that the functions are imported correctly.

Option 2: Using Google Colab

1. Go to https://colab.research.google.com

2. Upload the file "search.py" file through the “folder” icon > tab "Files" > "Upload". It should look like this image.



3. Open the intended notebook (e.g., puzzle-p2.ipynb) through “File” > “Upload notebook”.

4. Add, if it is not already there, a cell at the beginning of the notebook to import the file search.py:

```
from search import *
```

5. Run each cell with Shift + Enter.

Note: If you need to re-upload search.py after modifying it, restart the runtime environment (Runtime > Restart) and run it again.