COMPUTER ENGINEERING DEPARTMENT

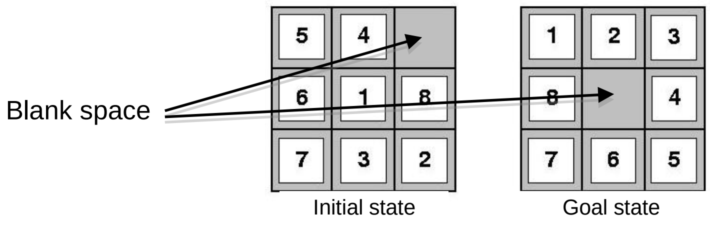
COMPUTER ENGINEERING

ARTIFICIAL INTELLIGENCE

2nd semester

WORKSHEET - SOLVING PROBLEMS BY SEARCHING

In this worksheet we will deal with problem solving using classical search methods, both non-informed (or blind) and informed methods. The goal consists in implementing several search methods and apply them to a specific problem: the Eight Puzzle. The goal in this problem is to find the sequence of actions of the blank space that allow us to reach the goal state departing from the initial state. Legal actions consist in moving the blank space up, down, to the right or to the left.



In the course's Moodle page, you will find a scaffold project (Search_Class.zip) containing the base code to be used in this worksheet. An **agent** solves some **problem**, using some **search method** – from the ones available (see the `Agent` class). Informed search methods use a **heuristic** to improve the search. During the search process, the environment is represented by a **state**, which can be modified through the application of **actions**. The output of the search process is a **solution** to the problem.

a) Please, analyze classes `Agent`, `Solution`, `State` and `Action` in package `agents`.

b) Please, analyze the `Problem` class in package `agents`, which describes a generic problem. A problem has an initial state as attribute. For convenience reasons, we also include a heuristic as attribute (class Heuristic). We will later deal with heuristics. The methods are the following:

- `get_actions()`: returns the list of actions that can be applied to some state (abstract method);
- `get_successor()`: applies an action to some state and return the resulting state (abstract method);
- `is_goal()`: verifies if some state is a goal state (abstract method);
- `compute_path_cost()`: computes the cost of a solution, which is basically a list of actions.

c) Please, analyze the `EightPuzzleProblem` class in package `eightpuzzle`, which represents the Eight Puzzle problem and which extends the `Problem` class. This class's constructor receives the initial and goal states as arguments. It also defines the four actions Up, Down, Left and Right corresponding, respectively, to classes `ActionUp`, `ActionDown`, `ActionLeft` and `ActionRight`, already implemented (please note that all actions have cost 1 in this problem). It also has methods `get_actions()`, `get_successor()`, `is_goal()` and `compute_path_cost()`. For this problem, the goal test (`is_goal()` method) consists in comparing some state to the previously defined goal state. The goal state is read in the `Window` class from file `states/goal_state_1.txt`.

d) Please, analyze the `EightPuzzleState` class in package `eightpuzzle`, which represents a state of the Eight Puzzle problem and that extends the `State` class. Besides the puzzle's matrix, this class also saves the blank space position. This class provides methods that allow to: verify if the blank space can be moved according to each of the four directions (up, down, left and right); move the blank space for each of these directions, return the value of a tile that is in some line and column, `get_tile_value()`, return a textual representation of the state, `__str__()`, compare two states, `__eq__()`, and return a hash value of the state, `__hash__()`.

e) Please, implement the `graph_search()` method from class `GraphSearch` in package `search_methods`. This class represents the generic graph search algorithm that serves as the basis to the most part of the search algorithms that you are going to implement. The algorithm is described in the code comments.

f) Please, implement the breath-first search method. To do so, you should redefine method `add_successors_to_frontier()` from class `BreadthFirstSearch`. Remember that this method treats the frontier (list of nodes that have not been expanded yet) as a queue (FIFO). It uses class `NodeQueue`, defined on package `utils`, to represent the frontier.

g) Please, implement method `add_successors_to_frontier()` from class `UniformCostSearch`. This method orders the frontier by the value of *g* of the nodes. Remember that, for each node, *g* is the path cost from the initial state until the state corresponding to that node. It uses class `NodePriorityQueue`, defined on package `utils`, to represent the frontier.

After these steps, run the application to test the code implemented so far (test with different initial configurations).

The following non-informed search methods are already implemented:

- Depth first search (class `DepthFirstSearch`). In this case, the `graphSearch()` method has been reimplement so that the algorithm does not use the explored set. Remember that the only advantage of depth first search is that it has a smaller space complexity but this is the case only if the explored set is not used.

- Limited depth first search (class `DepthLimitedSearch`), which extends class `DepthFirstSearch`.
- Iterative deepening search (class `IterativeDeepeningSearch`).

Finally, we will explore informed search methods, which are based on heuristics.

a) First, implement the “number of tiles out of place” heuristic (class `HeuristicTilesOutOfPlace`). The “sum of the distances of each tile to its goal position” heuristic is already implemented (class `HeuristicTileDistance`).

Now, implement the following informed search methods:

b) Greedy search (class `GreedyBestFirstSearch`).

c) A* (class `AStarSearch`).

The Beam (class `BeamSearch`) and IDA* (class `IDAStarSearch`) search algorithms are already implemented.

Test the two heuristics for each of the implemented algorithms.