COMPUTER ENGINEERING DEPARTMENT

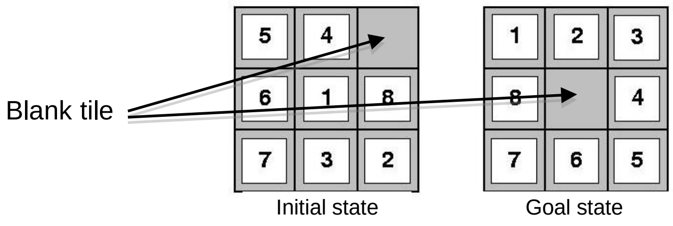
COMPUTER ENGINEERING

ARTIFICIAL INTELLIGENCE

2018/2019 – 2nd semester

SHEET 2 – SOLVING PROBLEMS BY SEARCHING

In this sheet 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 tile* that allow us to reach the goal state departing from the initial state. Legal actions consist in moving the *blank tile* up, down, to the right or to the left.



For your reference, the class diagram of the provided source code is included as an appendix to this sheet. 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`.

b) Please, implement the `Problem` class, which describes a generic problem. A problem has an initial state and a set of actions that can be applied to some state in order to generate new states. There should also exist methods that:

- Apply all the actions to some state and return the resulting states;
- Verify if some state is a goal state;
- Calculate the cost of a solution (which is basically a list of actions).

c) Please, implement the `EightPuzzleProblem` class, which represents the Eight Puzzle problem and which extends the `Problem` class. In this class's constructor, besides the initial state, you should also define the four actions Up, Down, Left and Right corresponding, respectively, to classes `ActionUp`, `ActionDown`, `ActionLeft` e `ActionRight`, already implemented (please note that all actions have cost 1 in this problem). You should also redefine methods `executeActions`, `isGoal` and `computePathCost`. Also note that, for this problem, the goal test (`isGoal` method) consists in comparing some state to the previously defined goal state. Therefore, you should also define a goal state as attribute (`goalState`). For simplicity, we consider that the goal state is always the same and that it corresponds to the `goalMatrix` attribute defined in the `EightPuzzleState` class.

d) Please, analyze the `EightPuzzleState` class, which represents a state of the Eight Puzzle problem and that extends the `State` class. Besides the puzzle's matrix, this class saves also the *blank tile* position. This class provides methods that allow to: verify if the blank tile can be moved according to each of the four directions (up, down, left and right); move the blank tile for each of these directions, return the position of each matrix tile, apply a set of actions to the state, return a textual representation of the state (`toString`), create a clone of the state (`clone`), compare two states (`equals`), return the number of lines, the number of columns and the value of a tile that is in some line and column. These three last methods are called, respectively, by methods `getRowCount`, `getColumnCount` and `getValueAt` from class `PuzzleTableModel`. This class connects the `EightPuzzleState` class and the `JTable` component that is used on the GUI to show the puzzle.

e) Please, analyze class `EightPuzzleAgent`, which extends the `Agent` class and which represents an agent able to solve the Eight Puzzle problem.

f) Please, implement the `graphSearch` method from class `GraphSearch`. 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.

g) Please, implement the breath first search method. To do so, you should redefine method `addSuccessorsToFrontier` from class `BreadthFirstSearch`. Remember that this method treats the frontier (list of nodes that have not been expanded yet) as a queue (FIFO). Use class `NodeLinkedList`, defined on package `utils`, in order to represent the frontier.

h) Please, implement method `addSuccessorsToFrontier` from class `UniformCostSearch`. This method orders the frontier by the value of *g* of the nodes, that is, for each node, *g* is the path cost from the initial state until the state corresponding to that node. Use class `NodePriorityQueue`, defined on package `utils`, in order to represent the frontier.

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

We can now implement the remaining non-informed search methods:

i) Depth first search (class `DepthFirstSearch`).

j) Limited depth first search (class `DepthLimitedSearch`), which extends class `DepthFirstSearch`.

k) Iterative deepening search (class `IterativeDeepeningSearch`).

Finally, we will explore informed search methods, which are based on heuristics. So, first, implement the two heuristics mentioned in the theoretical classes:

l) Number of tiles out of place (class `HeuristicTilesOutOfPlace`).

m) Sum of the distances of each tile to its goal position (class `HeuristicTileDistance`).

Now, implement the following informed search methods:

n) Greed search (class `GreedyBestFirstSearch`).

o) A* (class `AStarSearch`).

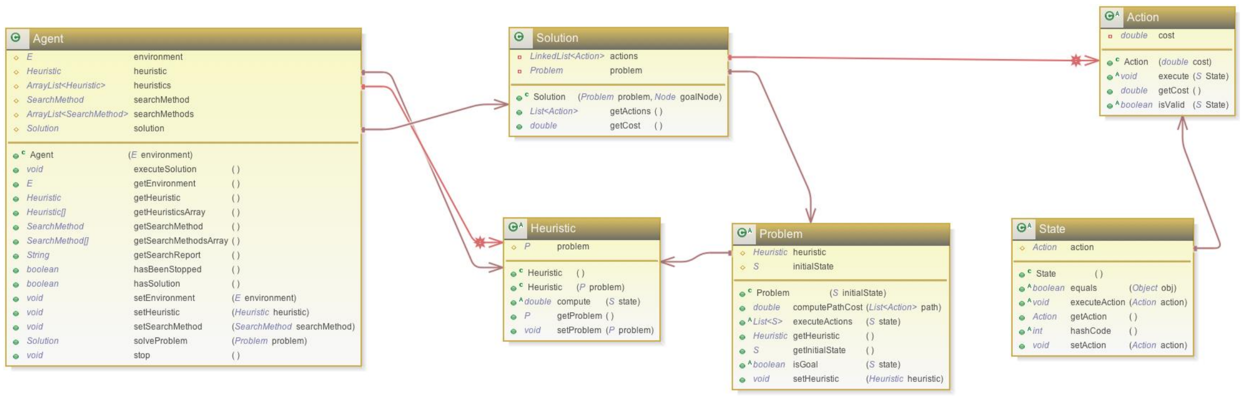
p) Beam search (class `BeamSearch`).

q) IDA* (class `IDAStarSearch`).

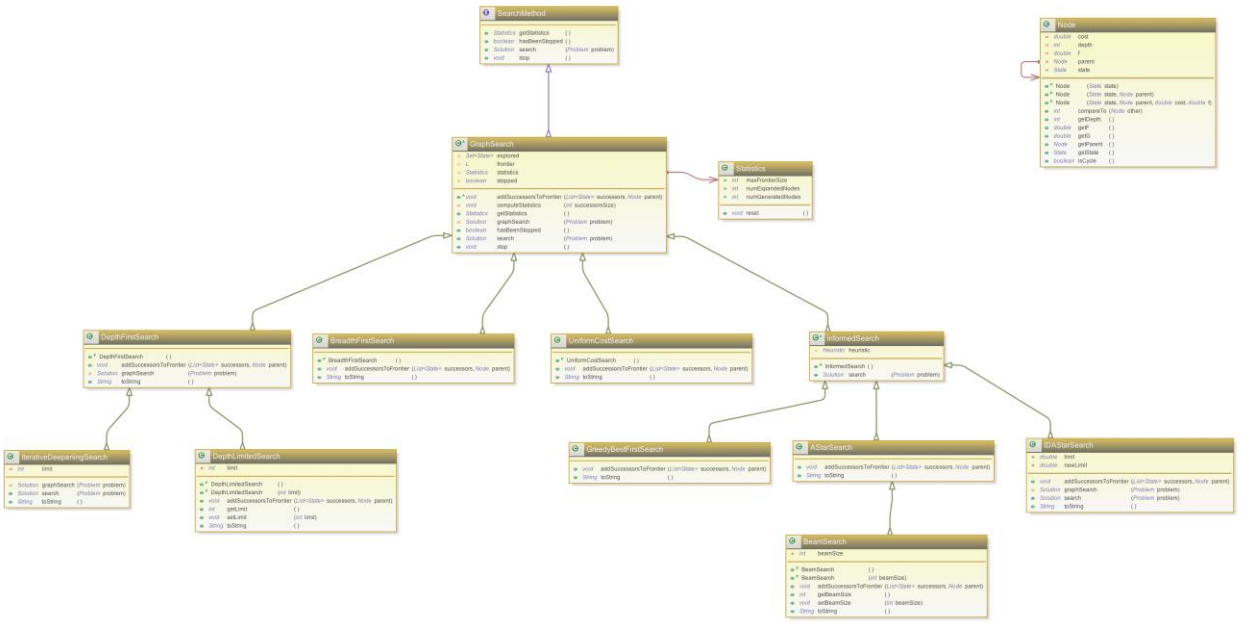
Test the two heuristics for each of the implemented algorithms.

Appendix

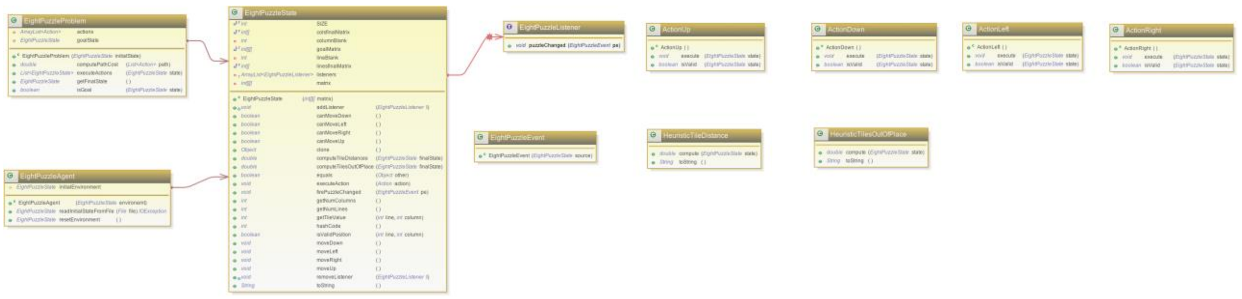
Package `agent`


Agent
E environment
Heuristic heuristic
ArrayList<Heuristic> heuristics
SearchMethod searchMethod
ArrayList<SearchMethod> searchMethods
Solution solution
Agent (E environment)
void executeSolution ()
E getEnvironment ()
Heuristic getHeuristic ()
Heuristic[] getHeuristicsArray ()
SearchMethod getSearchMethod ()
SearchMethod[] getSearchMethodsArray ()
String getSearchReport ()
boolean hasBeenStopped ()
boolean hasSolution ()
void setEnvironment (E environment)
void setHeuristic (Heuristic heuristic)
void setSearchMethod (SearchMethod searchMethod)
Solution solveProblem (Problem problem)
void stop ()
Solution
LinkedList<Action> actions
Problem problem
Solution (Problem problem, Node goalNode)
List<Action> getActions ()
double getCost ()
Action
double cost
Action (double cost)
void execute (S State)
double getCost ()
boolean isValid (S State)
Heuristic
P problem
Heuristic ()
Heuristic (P problem)
double compute (S state)
P getProblem ()
void setProblem (P problem)
Problem
Heuristic heuristic
S initialState
Problem (S initialState)
double computePathCost (List<Action> path)
List<S> executeActions (S state)
Heuristic getHeuristic ()
S getInitialState ()
boolean isGoal (S state)
void setHeuristic (Heuristic heuristic)
State
Action action
State ()
boolean equals (Object obj)
void executeAction (Action action)
Action getAction ()
int hashCode ()
void setAction (Action action)

Package `searchmethods`



Package `eightpuzzle`


EightPuzzleProblem
EightPuzzleState
EightPuzzleListener
ActionUp
ActionDown
ActionLeft
ActionRight
EightPuzzleEvent
HeuristicTileDistance
HeuristicTilesOutOfPlace
EightPuzzleAgent