Artificial Intelligence
Lab 1: Exploration (2h)

In this lab, we are interested in problem solving by exploration in a state space. This implies:

- formalizing a problem as a state space;
- implementation of an exploration algorithm.

To start, we will assume a first problem already formalized (the sliding-blocks game) and focus on setting up the algorithm. In a second step, we will focus on the formalization of a new problem (path through a maze).

We consider the Python code provided in the TP_1.zip archive on Moodle. The class diagram for this code is shown in Figure 1. This code contains:

- An abstraction module which contains the abstract classes State, Action and Problem. The Problem class includes all the useful methods to formalize a state space, such as studied during the course. Most of these methods are static because they are common to all instances of the same problem. Only the goal method is not static because it can depend on attributes specific to an instance (for example, an explicitly defined final state, as in the sliding-blocks game).
- A module, called *taquin,* that implements each abstract class in the abstraction module. The internal implementation of these classes is not important. Only methods inherited from the abstract class should be used in the mining algorithm.
- A Node class that groups a state, a possible parent (None if none), a possible action having led to the state (None if none), a cumulative cost g, a resolution potential f whose nature varies according to the chosen exploration algorithm (Dijkstra, greedy, A*). To parameterize the calculation of f, each node also defines an attribute criterion which is a function associating a real to a node. Thus, the attribute f of a node is calculated as the value returned by criterion for this node.
- A skeleton of the Exploration class whose explorer() method will allow you to launch an exploration. The class is parameterized by an instance of a problem and a criterion function to measure the potential of a node. For the purposes of the exploration algorithm, it also contains 2 collections of nodes: one for known but not yet explored nodes (open), the other for already explored nodes (close). In addition, an attribute n_steps counts the number of iterations of the algorithm. This provides a useful statistic at the end of the exploration.
- Finally, the archive also contains a main.py file which instantiates a problem of the sliding-blocks game on a concrete case, creates and launches an exploration, then displays the result of the exploration. For now, only the cumulative cost of a node is taken into account as a criterion for choosing a new node to explore.

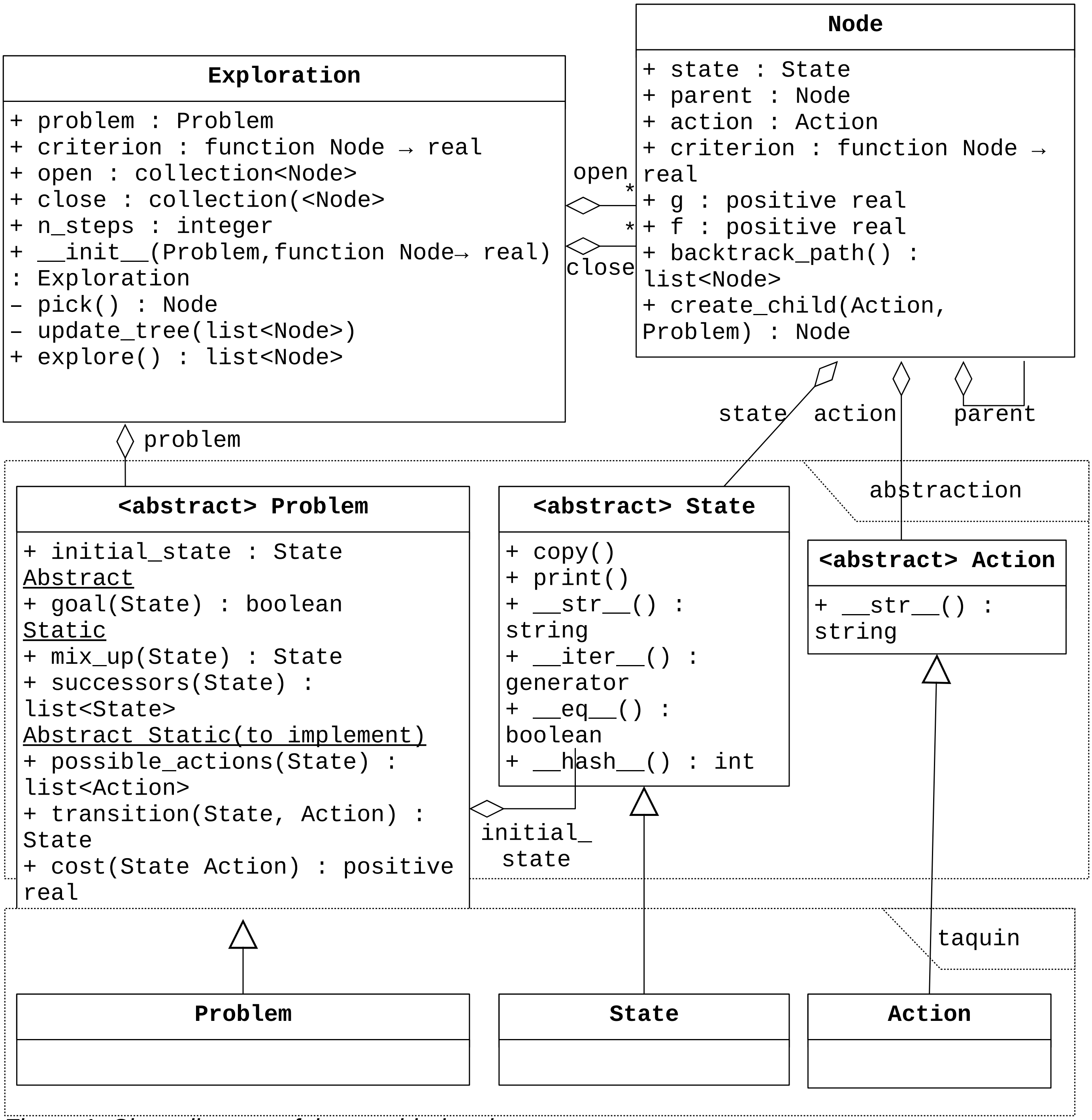

Figure 1: Class diagram of the provided code.

Question 1

Implement the explore(), pick() and update_tree() methods of the Exploration class.

The exploration pseudo-code is recalled by Algorithm 1.

Note: It is proposed to implement the open and close node collections in Python dictionaries (i.e. key-value tables). As the algorithm forbids storing 2 nodes with the same state, the keys are states and the values are nodes. This choice is made to speed up the search steps in the collections.

Question 2

```
function explore() :
    open : collection of nodes
    close : collection of nodes

    initialize open and close

    while open is not empty
    do
        current_node ← pick_in_open()
        add_in_close(current_node)
        if is_goal(current_node)
        then return solution
        else
            new_nodes ← successors(current_node)
            update_tree(new_nodes, open, close)
        end if
    end while

    return FAIL
end function

function update_tree(new_nodes, open, close):
    for each new_node in new_nodes
    do
        if new_node already in close
        then
            if new_node better than old one
            then
                remove old node from close
                add new_node in open
            end if
        else
            if new_node already in open
            then
                if new_node better than old one
                then
                    remove old node from open
                    add new_node in open
                end if
            else
                add new_node in open
            end if
        end if
    end for
end function
```

Algorithm 1 : Exploration pseudo-code.

In main.py, modify the evaluation function of a node to obtain the behavior of the greedy algorithm with the Manhattan distance heuristic (provided in the taquin.problem.Problem class).

Test and compare with Dijkstra.

Question 3

In main.py, modify the evaluation function of a node to obtain the behavior of the A* algorithm, still with the Manhattan distance heuristic.

Test and compare with Dijkstra and the Greedy Algorithm.

Question 4

In taquin.problem.Problem, implement the heuristic of the number of misplaced pieces (see course).

Compare its performance with the Manhattan distance heuristic on different cases, for use in A* and/or in the greedy algorithm.

We now consider a problem of movement in a maze. For this we represent a maze as a grid in which the boxes can be empty or contain a wall. One of the empty squares corresponds to the exit from the maze. A character moves through the maze from one empty space to another (it's up to you to allow diagonal movement if you wish).

Question 5

Create the State, Action and Problem classes corresponding to a problem of movement in a maze.

Test your implementation in the exploration algorithm.