LINGI2261: Artificial Intelligence
Assignment 2: Solving Problems with Informed Search

François Aubry, Cyrille Dejemeppe, Yves Deville
October 2015

Guidelines

- This assignment is due on **Wednesday 21 october 2015, 6:00 pm**.
- *No delay* will be tolerated.
- Not making a *running implementation* in *Python 3* able to solve (some instances of) the problem is equivalent to fail. Writing some lines of code is easy but writing a correct program is much more difficult.
- *Document* your source code (at least the difficult or more technical parts of your programs). Python docstrings for important classes, methods and functions are also welcome.
- Indicate clearly in your report if you have *bugs* or problems in your program. The online submission system will discover them anyway.
- Copying code or answers from other groups (or from the internet) is strictly forbidden. Each source of inspiration must be clearly indicated. The consequences of *plagiarism* is *0/20 for all assignments*.
- Answers to all the questions must be delivered at the INGI *secretary* (paper version). Put your names *and your group number* on it. Remember, the more concise the answers, the better.
- Source code shall be submitted on the online *INGInious* system. Only programs submitted via this procedure will be graded. No report or program sent by email will be accepted.
- Respect carefully the *specifications* given for your program (arguments, input/output format, etc.) as the program testing system is *fully automated*.

Deliverables

- The answers to all the questions in paper format **at the secretary**. **Do not forget to put your group number on the front page**.
- The following files are to be submitted on *INGInious* inside the *Assignment 2* task(s):
  - The file `sokoban.py` containing your Python 3 implementation of the Sokoban problem solver. Your program should take the path to the instance file as only argument. The search strategy that should be enabled by default in your programs is **A* with your best heuristic**. Your program should print the solution to the standard output in the format described further. The file must be encoded in **utf-8**.

1 Search Algorithms and their relations (3 pts)

1.1 *A** versus uniform-cost search

Consider the maze problem given on Figure 1. The goal is to find a path from 🚹 to € moving up, down, left or right. The black positions represent walls. This question must be answered by hand and doesn't require any programming.

Questions

1. Give a consistent heuristic for this problem. Prove that it is admissible.
2. Show on the left maze the states (board positions) that are visited during an execution of a uniform-cost graph search. We assume that when different states in the fringe have the smallest value, the algorithm chooses the state with the smallest coordinate (i, j) ($(0, 0)$ being the bottom left position, i being the horizontal index and j the vertical one) using a lexicographical order.
3. Show on the right maze the board positions visited by A^* graph search with a manhattan distance heuristic (ignoring walls). A state is visited when it is selected in the fringe and expanded. When several states have the smallest path cost, this uniform-cost search visits them in the same lexicographical order as the one used for uniform-cost graph search.

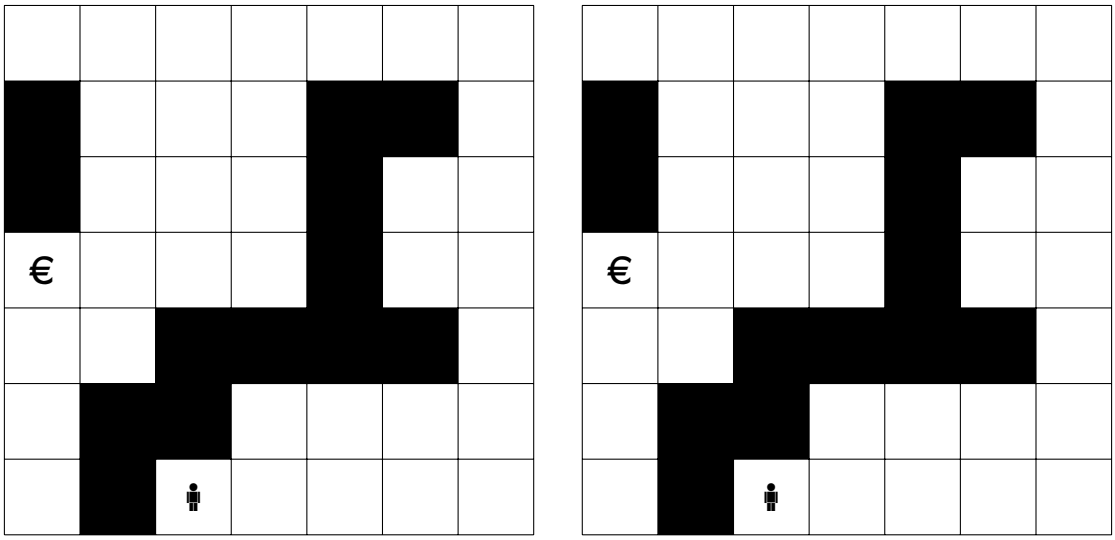

Figure 1

Important

As for the Assignment 1, a rapid tester of your program is provided on INGInious for you to quickly test the format of your outputs on some easy instances. It is named *"Sokoban: quick test"*. This is just a helper provided because the real submission can take some time due to the number of instances. **Submissions made in the *"Sokoban: quick test"* task will not be graded!** Only submissions made with your group login in *"Assignment 2: Sokoban"* will be used to grade your programs !

2 Sokoban planning problem (17 pts)

The problem you will solve for this assignment is the Sokoban planning problem. Again, the search procedures from aima-python3 will help you implement the problem! The Sokoban problem is the following. A person is in a 2d grid maze environment with walls on some positions. This person can potentially move in the 4 directions (up,right,down,left). In the initial state, some positions contain blocks and as many other positions are labeled as target positions for blocks. Blocks can be pushed one at a time in the moving direction of the person. The objective is to reach a final state where every block lies in a target position as illustrated on the Figure 2. Train yourself at `http://www.everyflashgame.com/game/39/Sokoban.html`. If you find this easy, take a look at the benchmarks: the last instances are much more difficult!

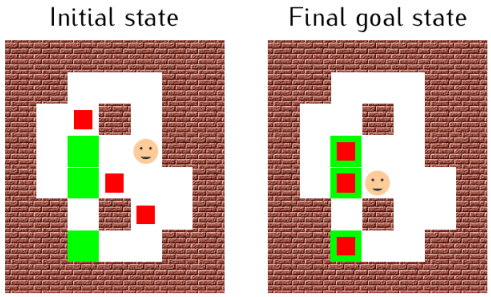

Figure 2

Some examples of allowed and not allowed moves are illustrated on Figure 3.

1. Move OK: The person pushes the block since the position behind the block is empty.
2. Move KO: The position behind the block is not empty, the block cannot be pushed in this direction.
3. This state cannot lead to a solution since one block is stuck in a corner that is not a target position (it will not be possible to move it).

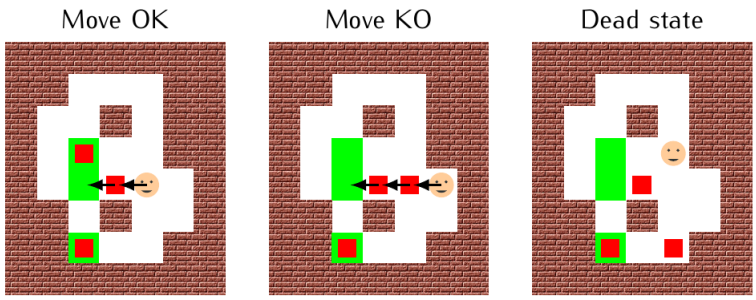

Figure 3

2.1 Input and output format

Resources for this problem are available on the Moodle web page of the course. Your are given a set of 5 sokoban instances. Each instance is composed of an init file, where you can find the initial position of the avatar and the boxes and a goal find where only the walls and target positions are depicted. The format of those files is the following:

- # are the wall positions
- @ is the position of the avatar
- $ are the positions of the boxes
- an empty position is represented with a white space
- · are the target positions for the boxes

The file sokoInst01.init and sokoInst01.goal are given below.

```
##########          ##########
#        #          #    .   #
#   $  $ #          #        #
#@       #          #     .  #
##########          ##########
```

A solution to this problem is composed of the successive states of the game that end up with each box on a target position. The three last states of a solution file for the instance 01 are represented below. Each state is represented using the same format as the input and separated from the others with an empty line. The target positions should not be represented in the printed states. The avatar can move only from one position to an adjacent one between two successive states in the solution file (of course it can be different in your model but not in the solution file).

```
##########
#    $ @ #
#     $  #
#        #
##########

##########
#    $@  #
#     $  #
#        #
##########

##########
#    $   #
#     @  #
#     $  #
##########
```

2.2 Your program

Questions

1. As illustrated on Figure 3 some situations cannot lead to a solution. Are there other similar situations? If yes, describe them.
2. Why is it important to identify dead states in your successor function? How are you going to implement it?
3. Describe possible (non trivial) heuristic(s) to reach a goal state (with reference if any). Is(are) your heuristic(s) admissible and/or consistent?
4. Implement this problem. Extend the *Problem* class and implement the necessary methods and other class(es) if necessary. Your file must be named *sokoban.py*. You program must print to the standard output a solution to the sokoban instance given in argument satisfying the described format. You will receive the name of the instance and you have to read from the two files .init and .goal, for instance, given instance `instance1` you will read from files `instance1.init` and `instance1.goal`.
5. Experiment, compare and analyze informed (*astar_graph_search*) and uninformed (*breadth_first_graph_search*) graph search of aima-python3 on the 15 instances of sokoban provided. Report in a table the time, the number of explored nodes and the number of steps to reach the solution. Are the number of explored nodes always smaller with *astar_graph_search*, why?
   When no solution can be found by a strategy in a reasonable time (say 5 min), explain the reason (time-out and/or swap of the memory).
6. What are the performances of your program when you don't perform dead state detection?