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

Gaël Aglin, Alexander Gerniers, Yves Deville

February 2021

Guidelines

- This assignment is due on **Thursday 11 March, 18:00**.
- *No delay* will be tolerated.
- *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.
- Copying code or answers from other groups (or from the internet) is strictly forbidden. Each source of inspiration must be clearly indicated.
- Source code shall be submitted on the online *INGInious* system. Only programs submitted via this procedure will be graded. No 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*.
- The answer to questions must be given by filling in the latex template provided. The final file must be submitted on *gradescope*. No report sent by email will be accepted.
- Nothing must be modified in the template except your answer that you insert in the *answer* environments as well as your names and your group number. The names are provided through the command *students* while the command *group* is used for the group number. The dimensions of *answer* fields *must not* be modified either. For the tables, put your answer between the "&" symbols. Any other changes to the file will *invalidate* your submission.
- To submit on gradescope, go to `https://gradescope.com` and click on the "log in" button. Then choose the "school credentials" option and search for *UCLouvain Username*. Log in with your global username and password. Find the course LINGI2261 and the Assignment 2, then submit your report. Only one member should submit the report and add the second as group member.
- For those who have not been automatically added to the course, at the right bottom of gradescope homepage, ckick on "Enroll on course" button and type the code *86KJVB*.
- Check this link if you have any trouble with group submission `https://help.gradescope.com/article/m5qz2xsnjy-student-add-group-members`

Deliverables

- The following files are to be submitted:
  - `report_A2_group_XX.pdf`: Answers to all the questions in a single report, named. Remember, the more concise the answers, the better.
  - The file `blocks.py` containing your Python 3 implementation of the Blockage problem solver. Your program should take the paths to the instance files (initial and goal states) as only arguments. 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**.

Anti plagiat charter

As announced in the class, you'll have to electronically sign an anti plagiat charter. This should be done ***individually*** in the **INGInious** task entitled *Assignment 2: Anti plagiat charter.* Both students of a team must sign the charter.

1 Search Algorithms and their relations (3 pts)

1.1 $A^\star$ versus uniform-cost search

Consider the maze problems given on Figure 1. The goal is to find a path from 🚹 to € moving up, down, left or right. The black cells 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 consistent. Also prove that it is admissible.
2. Show on the left maze the states (board positions) that are visited when performing a uniform-cost graph search, by writing the order numbers in the relevant cells. 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 $A^\star$ graph search with a manhattan distance heuristic (ignoring walls), by writing the order numbers in the relevant cells. A state is visited when it is selected in the fringe and expanded. When several states have the smallest path cost, they are visited in the same lexicographical order as the one used for uniform-cost graph search.

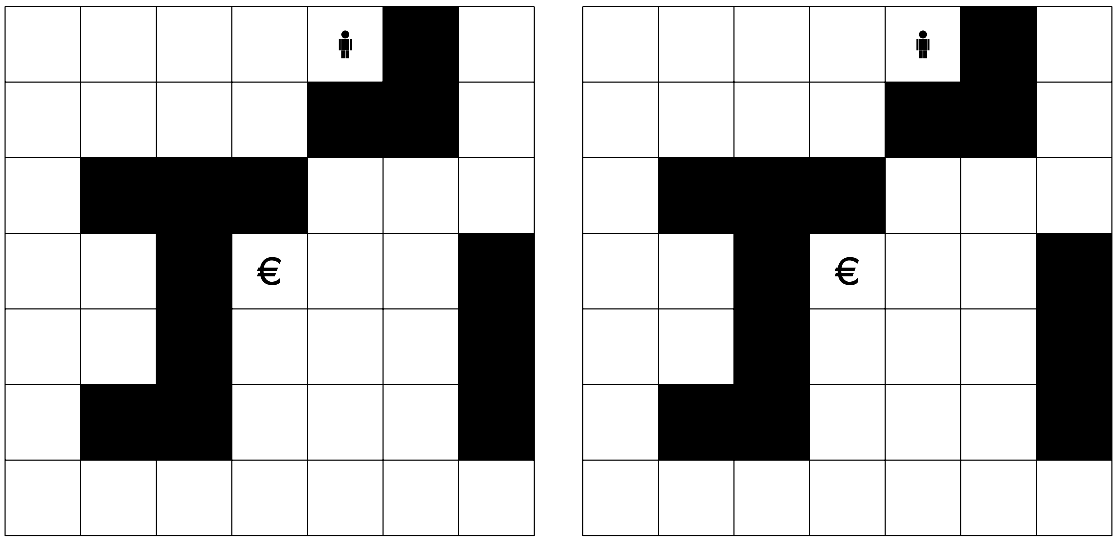
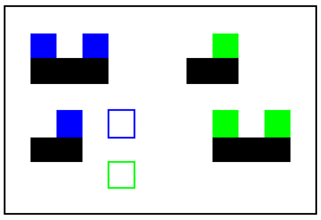


Figure 1

2 Blocks planning problem (17 pts)

The problem you will solve for this assignment, named ***Blocks planning problem*** is a variant of the Blocks world problem[1]. Again, the search procedures from aima-python3 will help you implement the problem. The Blocks planning problem consists in a set of movable blocks of predefined colors, possibly supported by a wall, which must be moved to corresponding target places of the same color. Each block can be moved of one position, either to the left or to the right. However, they are subject to the gravity and fall if they are not supported anymore. The blocks can be superposed that is, a block may hold another block on the top of it. There can be one or several target place(s) for a single color but only one block can be placed into a target position. Once a block reaches one of its target position, it becomes a fixed block and cannot be moved anymore. Note that during the fall of a block, it can only be stopped as soon as it reaches a block (movable or fixed), a wall or the end of the environment. The game is won as soon as each target place hold a block of its color. Figure 2 shows an instance of initial position for the game.


(a) Block planning problem with one target place per color

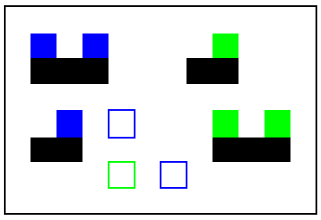

(b) Block planning problem with one target place for green blocks and two for blue blocks.

Figure 2: Examples of blocks planning problem.

Input and output format

Each instance is divided into two files. The first depicts the initial state while the second provides information about the goal state. They have respectively the extensions `.init` and `.goalinfo`. Your program must take the name of the instance (e.g. `a01.init`) as parameters and have to recover the corresponding initial state file as well as the goal information file. The syntax of the filename is `<instance_name + extension>`. For example, the initial state

[1] https://en.wikipedia.org/wiki/Blocks_world

file of instance `a01` is `a01.init` while its goal information file is `a01.goalinfo`. Note that the goal information state only describes the position and the color of the target places. No information are provided about infeasible blocks. Figure 3 shows and example of both initial and goal states, for the test instance `a01`. We use ASCII symbols in order to represent a state.

```
##############
#  c         #
#  b         #
#  a         #
####         #
####         #
##           #
##############
```

(a) Initial state

```
##############
#            #
#            #
#            #
####         #
####B        #
##A C        #
##############
```

(b) Goal state

Figure 3: The initial and goal instance files of instance `a01`. We always assume init and goal states to contain the same set of walls.

The frame structures over the environment is represented by `#`s, whereas the inner walls are indicated by `=`s. The moveable blocks are indicated by lowercase letters. The corresponding letter represents the color. There must be at least one color in the environment and there can be several blocks of the same color. The target places and their colors are indicated in the goal information file by uppercase letters. Again, there can be several target places of the same color. Once a block reaches a target place of its own color, the block becomes a fixed block represented by @ symbol, indicating that it cannot be moved anymore. The game is therefore won as soon as each target place holds a fixed block @. Figure 4 gives an example of a valid solution for instance `a01`. This solution, obtained by executing `python3 blockage.py path_to_a01`, is also optimal that is, involves a minimal number of actions (or moves).

```
##############
#  c         #
#  b         #
#  a         #
####         #
####         #
##           #
##############

##############
#            #
#  c         #
#  b         #
####         #
####         #
##  a        #
##############

##############
#            #
#  c         #
#  b         #
####         #
####         #
## a         #
##############

##############
#            #
#  c         #
#  b         #
####         #
####         #
##@          #
##############

##############
#            #
#            #
#  b         #
####         #
####         #
##@ @        #
##############

##############
#            #
#            #
#            #
####         #
####@        #
##@ @        #
##############
```

Figure 4: A possible optimal solution output for instance `a01`.

Be careful with the solution output format. Your solver must respect the exact same format as in Figure 4.

Questions

1. Model the Blocks planning problem as a search problem; describe:
   - States
   - Initial state
   - Actions / Transition model
   - Goal test
   - Path cost function
2. Consider the following state for the `a01` instance:

   ```
   ##############
   #            #
   #  c         #
   #  a         #
   ####         #
   ####         #
   ##  b        #
   ##############
   ```

   According to the goal state, such a situation cannot lead to a solution. Can you find other similar situations (in general, not only on that specific instance) that leads to a deadlock? If so, describe two.
3. Why is it important to identify dead states? How are you going to take it into account in your solver?
4. **Describe** a possible (non trivial) heuristic to reach a goal state. Is your heuristic admissible and/or consistent? Why ?
5. **Implement** this problem. Extend the *Problem* class and implement the necessary methods and other class(es) if necessary. **Experiment**, compare and analyze informed (*astar_graph_search*) and uninformed (*breadth_first_graph_search*) graph searches of aima-python3 on the 10 instances of Blocks planning provided. Report in a table the time, the number of explored nodes, the number of remaining nodes in the queue and the number of steps to reach each solution. When no solution can be found by a strategy in a reasonable time (say **1 min**), indicate the reason (time-out and/or swap of the memory).

   Are the number of explored nodes always smaller with *astar_graph_search*? What about the computation time? Why?
6. **Submit** your program on INGInious, using the A^* algorithm with your best heuristic(s). Your program must print to the standard output given a time limit of 1 minute, a solution to the Blocks planning instance passed as parameter to it, satisfying the described output format. Your program will be evaluated on 11 instances, one of which is hidden. We expect you to solve at least 8 out the 11.