

LINGI2261: Artificial Intelligence

Assignment 2: Solving Problems with Informed Search

François Aubry, Michael Saint-Guillain, Thanh Khong Minh, Yves Deville
October 2017



Guidelines

- This assignment is due on **Wednesday 1st November 2017, 6:00 pm**.
- **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.
- 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.
- 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 following files are to be submitted on **INGInious** inside the *Assignment 2* task(s):
 - 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 `blockage.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^* 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 consistent. Also 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.

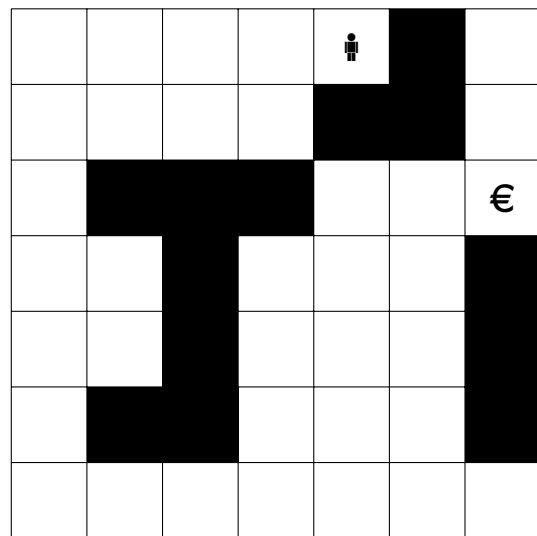
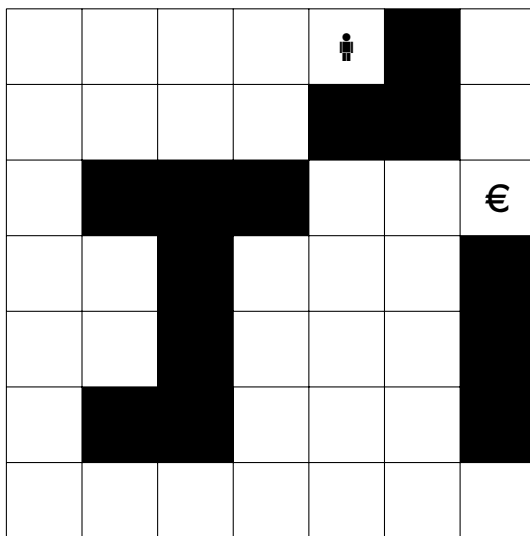


Figure 1

2 Blockage planning problem (17 pts)

The problem you will solve for this assignment is the Blockage planning problem. Again, the search procedures from `aima-python3` will help you implement the problem! The Blockage problem we consider is a simplified version of the online game version available at <http://www.kongregate.com/fr/games/guilovsh/blockage>. The game consists in a set of movable blocks of predefined colors, which must be moved to target places of the same color. Each colored block can be move of one square, either to the left or to the right. However, they are subject to the gravity and fall if they are not supported anymore ! The game is won as soon as all the target places hold a block of the same color. Figure 2 shows an instance that you can find on the online game.

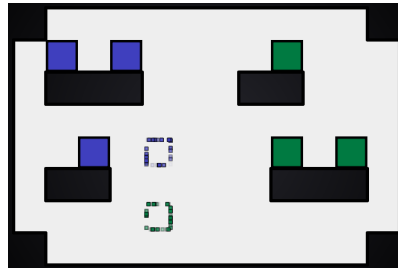


Figure 2: The Blockage game available online. The puzzle is solved once both a blue block and a green block are placed on the upper and lower empty squares, respectively.

Our version of the game will be a little bit simplified from the online game version. In our version, the only possible actions are either moving a block to the left or to the right. The colored movable blocks cannot be superposed that is, a colored block may hold another movable block on the top of it. Once a block stands on (one of) its target position(s), it cannot be moved anymore. There is thus no special action in order to solidify a moveable block, as we assume them to be solid from the beginning.

Input and output format

We distinguish in the instance files the *initial state* from the *goal state*. Your program must take two arguments: the initial state (e.g. `b01.init`) and the goal state (e.g. `b01.goal`). We use ASCII symbols in order to represent a state. Figure 3 shows an example of both initial and goal states, for the test instance *b01*.

Initial state	Goal state
#####	#####
# c #	# # #
# b #	# # #
# a #	# # #
#### #	#### #
#### #	####B #
## #	##A C #
#####	#####
b01.init	b01.goal

Figure 3: The initial and goal instance files of instance *b01*. We always assume init and goal states to contain the same set of walls.

The solid wall structures are represented by `#`'s, whereas moveable blocks are indicated

by lowercase letters. The corresponding letter represents the color. We consider at most 3 different colors: **a**'s, **b**'s and **c**'s, but there can be several blocks of the same color. The target places and their colors are indicated in the goal state 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 **@** symbol, indicating that the block is fixed and cannot be moved anymore. The game is therefore won as soon as all the target places hold a fixed block **@**. Figure 4 gives an example of a valid solution for instance *b01*. This solution, obtained by executing `python3 blockage.py instances/b01.init instances/b01.goal`, is also optimal that is, involves a minimal number of actions (or moves).

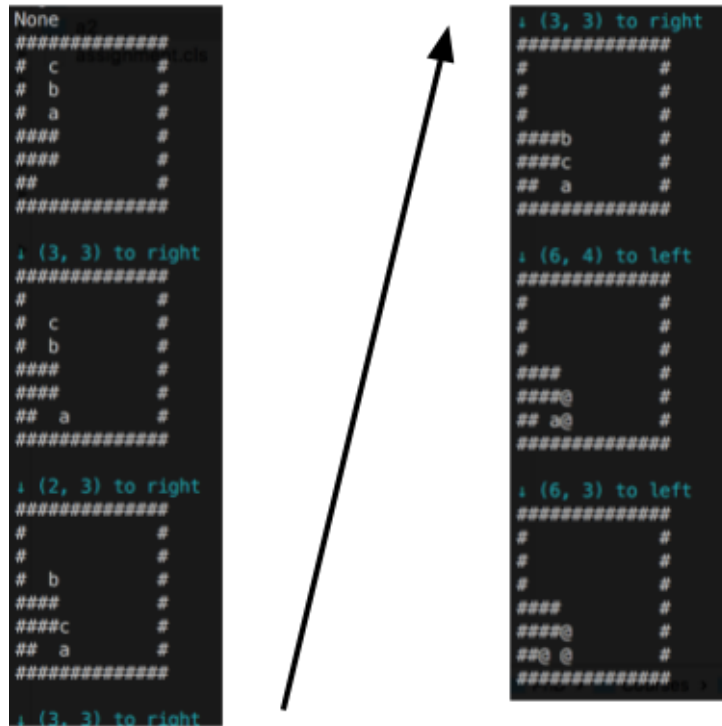


Figure 4: A possible optimal solution output for instance *b01*.

Be careful with the solution output format ! Your solver must respect the exact same format as in Figure 4. In particular, it must start with one line of comments (the content of this line doesn't matter) followed by the initial state and finally an empty line. Thereafter, each intermediate state is similarly preceded by a comment line (its content doesn't matter) and followed by an empty one, until we reach the final state.



Questions

1. Model the Blockage problem as a search problem; describe:

- States
- Initial state
- Actions / Transition model
- Goal test
- Path cost function

2. Consider the following state for the *b01* instance:

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

Obviously, such a situation cannot lead to a solution.

Are there other similar situations (in general, not only on that specific instance)? If so, describe them.

3. Why is it important to identify dead states? How are you going to take it into account in your solver?
4. **Describe** possible (non trivial) heuristic(s) to reach a goal state Is(are) your heuristic(s) admissible and/or consistent?
5. **Implement** this problem. Extend the *Problem* class and implement the necessary methods and other class(es) if necessary.
6. **Experiment**, compare and analyze informed (*astar_graph_search*) and uninformed (*breadth_first_graph_search*) graph search of aima-python3 on the 10 instances of Blockage 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*? What about the computation time? Why?
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).
7. **Submit** your program on INGIInious, using the A^* algorithm with your best heuristic(s). Your file must be named *blockage.py*. Your program must print to the standard output a solution to the Blockage instance given in arguments, satisfying the described output format.