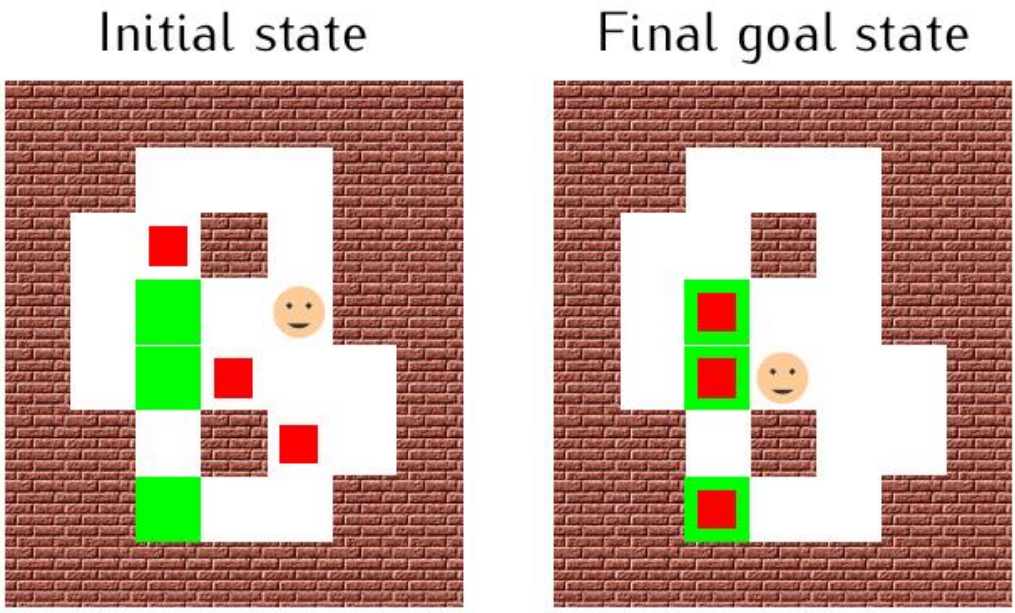
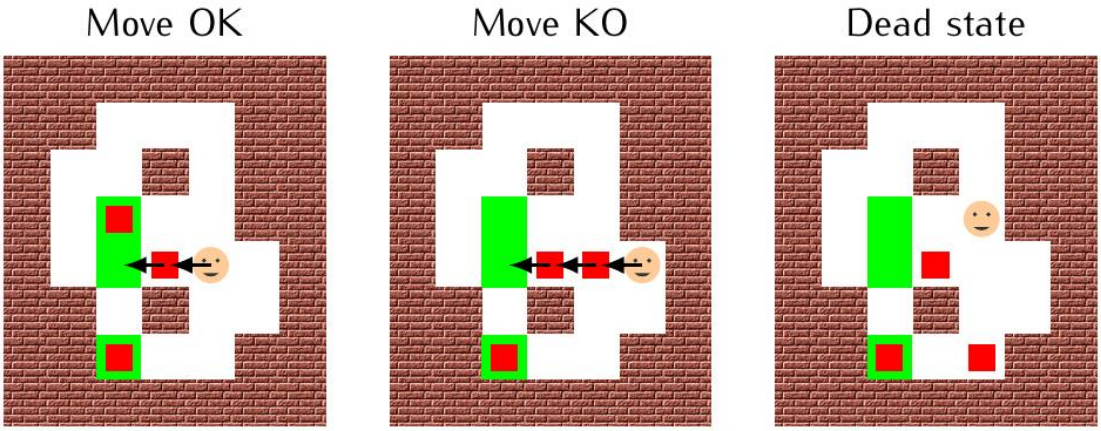
Sokoban planning problem

The problem you will solve for this assignment is the Sokoban planning problem. Again, the search procedures from search.py 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 https://www.mathsisfun.com/games/sokoban.html
If you find this easy, take a look at the benchmarks: the last instances are much more difficult!



Some examples of allowed and not allowed moves are illustrated on the next figure .
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).



Input and output format

Resources for this problem are available on teams. You 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).

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

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

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

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. Your code will be called in the manner:

python sokoban.py instance1

Here are some questions to help you start:

Why is it important to identify dead states in your successor function? How are you going to implement it?

Describe possible (non trivial) heuristic(s) to reach a goal state (with reference if any). Is(are) your heuristic(s) admissible and/or consistent?

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).

What are the performances of your program when you don't perform dead state detection?