Methods of Knowledge Based Software Development
ITI8600

Homework 1 :

Modeling of Klotski puzzle

Lecturer: **Mr. Juhan-Peep Ernits**

Noël SCHNEIDER
Guillaume RICARD

September 2017
Erasmus Students

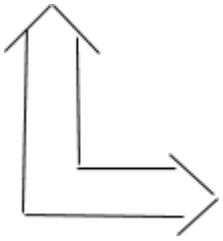
¤ Task :

In this first homework, our task was to model a Klotski game using all the notions learnt during the lecture namely search algorithms, heuristic functions and constraints.
To realise this program we chose to use Python language and an associated library search.py which give access to search algorithms presented in the Russell And Norvig's "Artificial Intelligence - A Modern Approach".

¤ Structure in memory

In order to use the search.py library, we modeled our Klotski board as a list. Each shape is characterised by a number in a way to consider each box bearing this number as part of the corresponding shape. This modelisation will be really useful to move the shapes in the same time :

```
28 state_task = (1,1,2,3,8,8,
29               1,1,4,5,9,9,
30               -1,-2,6,7,10,10,
31               11,11,12,13,13,14,
32               11,12,12,13,14,14)
```



1	1	2	3	8	8
1	1	4	5	9	9
-1	-2	6	7	10	10
11	11	12	13	13	14
11	12	12	13	14	14

¤ Enumerations to make the program clearer :

In order to make the program clearer, we chose these enumerations:

```
#We here define the relative position of the neighbours of a piece in the list

up    = -6
down  = 6
left  = -1
right = 1

#Limits of the puzzle
limit_line_up = 0
limit_line_down = 4
limit_column_left = 0
limit_column_right = 5

previous_actions=[]

blank = {-1, -2}
bsquare = 1
square = {2, 3, 4, 5, 6, 7}
rectangle = {8, 9, 10}
corner = {11, 12, 13, 14}
anything = {-1, -2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
```

For consistency reasons, we had to associate a different value for each instance of shapes: for instance, each small square will be represented by a value between 2 and 7, same thing with rectangles between 8 and 10, etc…

We chose this method for two reasons:

> Make tests clearer when displaying the Puzzle in an user-friendly way.
> Make moves easier to be determinated by our program.
> Increase the code lisibility.

If we had chosen the same character for each shape, we wouldn't be able to distinguish two same pieces side by side easily. Then, it's easier for someone to read our code with "bsquare" written rather that "1", and it allows us to use this kind of code:

```
if state[blank1+right] in square:
```

It means that we can check if a frame of our list is, or is not a certain shape. Because we are working with a list representing a 5x6 puzzle, talking about "the frame above" isn't very representative in terms of value. Basically, a list of 30 values is represented this way:

0	1	2	3	4	5
6	7	**8**	9	10	11
12	**13**	**14**	**15**	16	17
18	19	**20**	21	22	23
24	25	26	27	28	29

If we look at the 14th frame of our list, we have 8, 13, 15 and 20 as its neighbours. To make our code clear, we chose to define global values like this:

up = 6 down = -6 right = 1 left = -1

Now, if we want to access the value near any case, we can use:
state.index[value+up] for the frame above, or state.index[value+down] for the frame below. Everywhere in our code, we make sure that we never go out of our list by using divmod, that gives us the relative coordinates of our blanks. Thanks to it, we will make sure that the moves we are trying to do never go out of our list, and also that we do not consider that a frame is a neighbour of another if it's not the case. For example with the array above, the program would have considered that 17 is the left neighbour of 18, because its position is technically 18 + left.

```
blank1 = state.index(-1)
blank2 = state.index(-2)
line1, column1 = divmod(blank1,6)
line2, column2 = divmod(blank2,6)
```

```
limit_line_up = 0
limit_line_down = 4
limit_column_left = 0
limit_column_right = 5
```

For our goal_test, we also created a "anything" enumeration which lists all the shapes but the big square:

anything ⇒ {-1, -2, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}

¤ **Definition of the HW1Puzzle class :**

```
77
78 class HW1Puzzle(search.Problem):
79
```

The HW1Puzzle is divided in four essential parts :

A) Initial part :

The aim of this part is exclusively to initiate our class …

```
80     initial = ()
81
82     def __init__(self, initial):
83         self.initial = initial
84
```

B) Goal test part :

The goal_test function is essential to know if the puzzle is solved. To test it we have to compare all the boxes of the list. In our program, the goal is filled when the big square is in the right-bottom corner :

```
86
87     goal = (anything,anything,anything,anything,anything,anything,
88             anything,anything,anything,anything,anything,anything,
89             anything,anything,anything,anything,anything,anything,
90             anything,anything,anything,anything,bsquare,bsquare,
91             anything,anything,anything,anything,bsquare,bsquare)
92
93     def goal_test(self, state):
94         if (state[22] == self.goal[22] and
95         state[23] == self.goal[23] and
96         state[28] == self.goal[28] and
97         state[29] == self.goal[29]):
98             return True
99
```

C) The “actions” part :

```
101
102     def actions(self, state):
103         blank1 = state.index(-1)
104         blank2= state.index(-2)
105         line1, column1 = divmod(blank1,6)  #peut-être inversés
106         line2, column2 = divmod(blank2,6)
107         action=[]
108         global previous_actions
109         #print("Actions précédentes:", previous_actions)
110
111         ''' Définition des collisions entre blocs'''
112
113         ''' STEP 0 : One-blank situation '''
114
115         if column1 < limit_column_right:
116             if state[blank1+right] in square:
117                 if "left_square1" not in previous_actions: action.append("right_square1")
118         if column2 < limit_column_right:
119             if state[blank2+right] in square:
120                 if "left_square2" not in previous_actions: action.append("right_square2")
121         if column1 < limit_column_right-1:
122             if state[blank1+right] in rectangle and state[blank1+2*right] == state[blank1+right]:
123                 if "left_hrectangle1" not in previous_actions: action.append("right_hrectangle1")
124         if column2 < limit_column_right-1:
125             if state[blank2+right] in rectangle and state[blank2+2*right] == state[blank2+right]:
126                 if "left_hrectangle2" not in previous_actions: action.append("right_hrectangle2")
```

The purpose of this function is to analyze the current configuration of the board and return all the authorized moves in the *action* array.

1	1	2	3	8	8
1	1	4	5	9	9
-1	-2	6	7	10	10
11	11	12	13	13	14
11	12	12	13	14	14

In this example, there are three possible movements : right_square2, up_bsquare, down_corner.
The syntax is made as followed:
> “*right”* means that the shape is on the right of the blank(s)
> “*square”* means that the shape to move is a square
> “*2”* is added to specify that we will move the shape to the 2nd blank, represented by -2. We use “1” if we want to move it to the 1st blank.

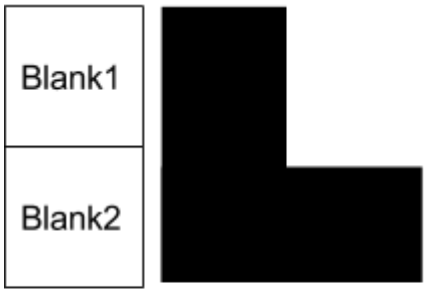
```
    ...: astar_with_h1(puzzle_task)
['right_square2', 'up_bsquare', 'down_corner']
```

In our code, we can check all the possible movements by adding a “print(action)” at the end of the *actions* class.

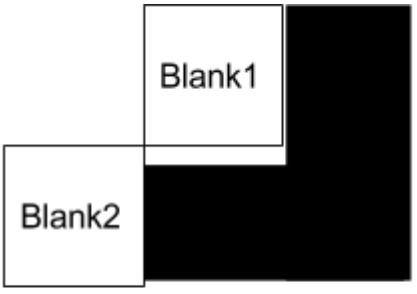
We have defined a list of all the possible movements depending on the shape, the allowed direction and the number of blanks to move during the operation (blue = one blank to move ; pink = two blanks to move). For each blue action, we have to specify which blank we have to move.

Shape \ Movements name	UP	DOWN	LEFT	RIGHT
Big square (2*2)	up_bsquare	down_bsquare	left_bsquare	right_bsquare
Corners (or “L” blocks)	up_corner	down_corner	left_corner	right_corner
small squares	up_square	down_square	left_square	right_square
horizontal rectangles (2*1)	up_hrectangle	down_hrectangle	left_hrectangle	right_hrectangle
vertical rectangles (1*2)	up_vrectangle	down_vrectangle	left_vrectangle	right_vrectangle

As we said, we need to specify which blank we are talking of for a “one blank” move. We then add “1” or “2” at the end. Finally, we also distinguish the case when we want to move a corner while the blanks are diagonally side by side.



right_corner



right_corner_diag

The reason is because when the blanks are diagonally, we require more precision regarding their position in the puzzle, so we don’t try to get out of the list’s range. In fact, we chose not to distinguish the different shapes of a corner in the *actions*, but in the *result* part (mostly because it doesn’t mathematically matter to know how exactly is the corner).

In total, there are 32 possible movements.

¤ First issue : edge management and “Out of range” problems

In our model, we have to take into account the edges of the board. To do this, we created four useful variables corresponding to the coordinates in a 2-dimensions table :

```
104     def actions(self, state):
105
106
107         blank1 = state.index(-1)
108         blank2= state.index(-2)
109         line1, column1 = divmod(blank1,6)
110         line2, column2 = divmod(blank2,6)
111
```

“Out of range” errors are caused by the reading of a non-existing box in the list :

```
  File "<ipython-input-36-cdc695a63472>", line 1, in <module>
    state_task[-48]

IndexError: tuple index out of range
```

To avoid these errors, we will use our variables “line1”, “line2”, “column1” and “column2” to control if blank1 and blank2 (respectively designated by the numbers -1 and -2) are in the frontiers of the board :

```
121         if column1 < limit_column_right:
122             if state[blank1+right] in square:
123                 if "left_square1" not in previous_actions: action.append("right_square1")
```

For instance, in this case the “actions” function won’t authorize a “right movement” of blank1 if it is situated on the right edge.
Moreover, we always keep on memory the previous actions done with the global variable *previous_actions*. It allows us to check before trying a move that the opposite move hasn’t been done. It significantly reduced the number of nodes in the search algorithms (earlier in the project, we spared more than 2000 moves while trying to solve the “hard” puzzle in the assingment’s task, Figure 1).

D) Result function

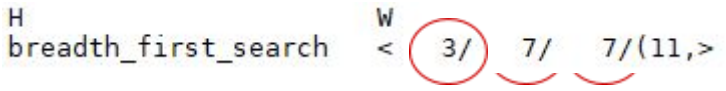
This final essential part of our HW1Puzzle class represents someway the hands of our puzzle. It will listen to the “actions” part and will only execute the authorized movements. This part will translate the name of the movement into a modification of elements in the list :

```
389
390         if action == "up_vrectangle2":
391             newState[blank2] = newState[blank2+up]
392             newState[blank2+2*up] = -2
393
```

¤ Search algorithms

The final part of the task consists in using three different search algorithms to solve the Klotski Puzzle.

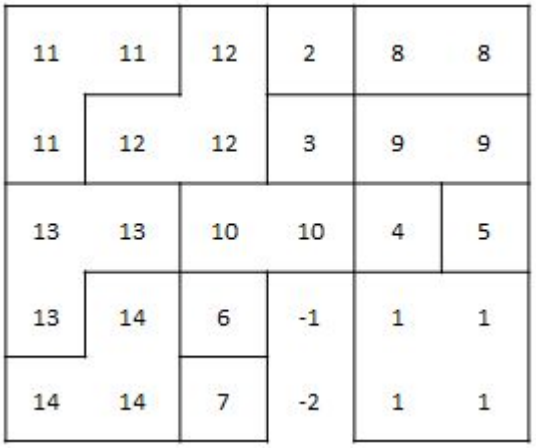
Problem : Our program displays too big values to be taken in consideration (20 minutes computing to display thousands of necessary moves). We didn't manage to find the mistakes which can cause these too huge results. That is why we will compare the different search algorithms with simpler board configuration. To compare it we will use an interesting function present in the search.py library : search.compare_searchers :



- The first number indicates the number of movements that the program needs to do to solve the puzzle (in this case 3 movements are necessary)
- The second number indicates the number of times that the program compares the current configuration to the final goal (in this case, 7 times)
- The third number indicates the number of the several configuration during the solving process

To compare the search algorithms, we chose to compare different initial configurations :

In a first step, we would like to solve the puzzle in a winning configuration :

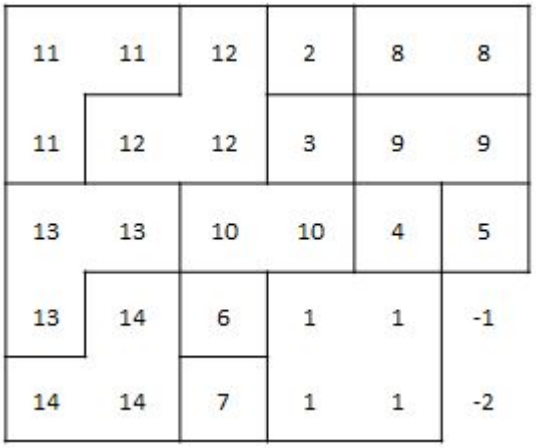


In this case, we expect the program to give us zero movement to do.

Indeed, regardless of the algorithm used, the comparison function displays us the same path :

```
Puzzle 0:
H                       W
breadth_first_search    <   0/   1/   0/(11,>
astar_with_h1           <   0/   1/   0/(11,>
astar_with_h2           <   0/   1/   0/(11,>
```

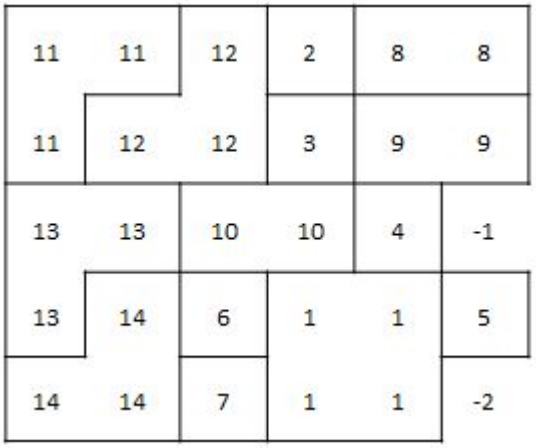
In a second step, we gave to the search algorithm a new configuration that requires only one movement to be solved :



In this second case, we can also observe that the “perfect” path is followed by the three algorithms :

```
H                       W
breadth_first_search    <   1/   3/   2/(11,>
astar_with_h1           <   1/   3/   2/(11,>
astar_with_h2           <   1/   3/   2/(11,>
```

In a third step, we gave to the search a configuration requiring two movements to be solved :

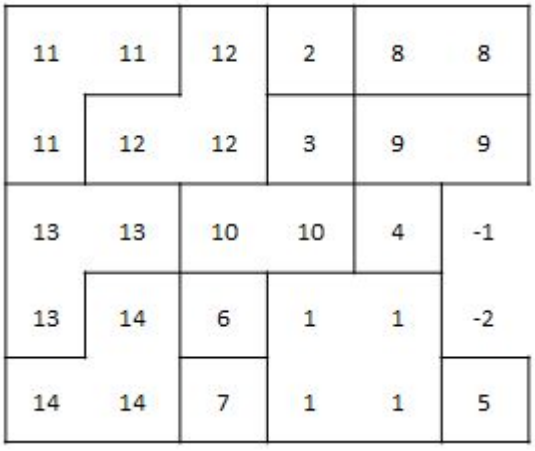


From this problem, the several search algorithms give us different results :

```
Puzzle 2:
H                          W
breadth_first_search     < 252/ 300/ 602/(11,>
astar_with_h1            <   4/   6/   8/(11,>
astar_with_h2            <  75/  77/ 176/(11,>
```

A really strange phenomenon appears : while the puzzle needs only two moves to be solved, the algorithms give us really different results with an huge gap between those.

It was interesting to analyse the results for puzzles needing more moves as the following one :



Oddly, for this configuration, the results seems to be really close to the perfect path and the difference between the three algorithms is not so huge :

```
Puzzle 3:
H                          W
breadth_first_search     <   6/  10/  13/(11,>
astar_with_h1            <   6/   8/  13/(11,>
astar_with_h2            <   5/   7/   9/(11,>
```

Others configurations could be tested in our program. Because of the too huge time to be executed we won't display the results of the "task configuration" and the "hard configuration" given by the lecturer. The results can reach 100 000 moves.
This is certainly due to mistakes in our program, like some movements not authorized whereas they should be. To solve it, we put some "print" at every call of an action, and checked if some were never happening.

We hope to improve our program before the next week presentation.

Conclusion :

Even if our results do not live up to our expectations, this homework helped us to understand the different notions addressed in class : search algorithms, heuristic functions, and constraints. We think that our puzzle modelization if technically correct, even if we could certainly improve it by factorising some functions, especially inside the *actions* method. We also lacked of rigor in the beginning of our work, by not defining clearly enough the structure of our program. In a first part, we splitted the work between *actions* and *results*, and figured few days later that we sometimes didn't use the same syntaxes between our two parts. In this case, it happened that some actions given didn't figure in the *results* class with the same nomenclature.

We learned from these mistakes, mostly due to a lack of experience in group programming projects, so we can be more efficient for the next homework.