Artificial Intelligence

P2 (Week 4+5)

Sokoban tasks are with an asterisk (that is, they do not count towards the maximum and can be treated as optional). To make the list easier to read, the contents of all the tasks are given first, and details of the formats and communication with the checker are given later.

Obrazki logiczne

Task 1. (4p) In this task we return to logic pictures, this time in the full version. You are to write a program that solves full logic pictures (that is, it reads in a description and prints out the corresponding picture).

Remark: data for this task will be the size of (at most) 15x15, in a reasonable time (up to a minute) it should be handled by a program that implements the algorithm from the P1 list (with a new function that checks the row/column match to the description).

Sokoban

Task 2. (4p)*

We are going to consider the game Sokoban[1], in which the warehouseman is tasked with placing boxes in preset positions. The warehouseman moves in four directions (up, down, right and left). If there is a crate in front of the warehouseman, he can try to push it, but he cannot pull it.

Implement a BFS search in which we consider the movements of the warehouseman. Add any detection of dead states (i.e., states that are known to be impossible to get a solution from, despite the fact that it is still possible to make various warehouseman moves and box transfers).

Task 3 (3p)* This task contains more difficult tests for Sokoban. It is quite possible that the code for the previous task will pass it, especially if the detection of dead states is a bit more sophisticated than the quite simplest. The score depends linearly on the number of tests passed.

Commando in the maze

Task 4. (4p) We consider a commando who moves through a maze (a maze consists of square fields, forming a rectangle). We have the following types of fields:

1. walls, denoted by #, on which one cannot move,

[1]eg., `https://www.sokobanonline.com/`

2. target points (marked G), which must be reached to detonate a the charge over the enemy's underground warehouses,

3. starting points (marked S), these are where the commando can find himself in the first turn,

4. launch and target points (marked B),

5. the remaining points, marked with a space.

The commando can move in 4 directions (UDLR, marked as in Sokoban). Movement toward a wall does not trigger a change of state. The commando is dropped at night and does not know where exactly the drop took place (at which starting point), but knows the map of the maze. We are interested in finding a sequence of moves that will definitely lead to some end state (i.e. our soldier executes his plan, which is a sequence of moves, after which he can set off the bomb, because regardless of where he was at the beginning of the trek, at the end he will be at one of the target points. We will call such a plan a winning one

Write a program that solves the commando task, that is, for each test case it writes out the winning plan. In this task, the plan does not have to be optimal, it is only required to be shorter than 150 moves (for each case). The solution should have two phases implemented:

1. to perform random/voluntary moves to reduce uncertainty,

2. to perform a BFS search (A* must not be used).

Your task, therefore, is, among other things, to see how much uncertainty the BFS is able to accept and to propose an action scheme for part one that the uncertainty is able to reduce to the required level. The evaluation depends linearly on the number of test cases the scheme will handle in the time limit.

Task 5 (4p). We solve the same task as above, but using A*. In addition, we require that this time the winning plan be optimal (no longer than any other winning plan). The tests for this task will be structured so that step 1, which reduces uncertainty, will not be necessary.

The score depends linearly on the number of test cases the program will handle within the time limit.

Task 6 (2p). Suggest a sensible modification of the heuristic from the previous task Task to make it unacceptable. The heuristic should have a parameter (let's call it: degree of unacceptability), check what time gain you manage to achieves for different degrees of unacceptability, and what cost you pay for it in the total length of the plans found.

We test the task on the same tests as the previous ones. Hint: there is a solution that requires extremely little additional code.

Task 7 (1p). This task is a combination of the previous three tasks. You should propose some combination of methods from these tasks that:

a) Is able to solve the tests for task 4 in the time limit,

b) Achieves better results (in the sense of sum of plan lengths) than those from the task with the BFS.

Checker

A checker will be provided for tasks 1, 2, 3, 4 and 5. Task 6 is checked with the tester for task 5. Task 7 is checked with the tester for tasks 4 and 5. Examples of using the checker:

1. launching all tests for a given task:
   `python validator.py zad1 python rozwiazanie.py`
2. launching selected tests:
   `python validator.py --cases 1,3-5 zad1 a.out`
3. Writing out an example input/output:
   `python validator.py --show_example zad1`
4. Writing out the boards for Sokoban and Komandos
   `python validator.py --verbose zad1 python rozwiazanie.py`

Please prepare solutions in a format compatible with the checker.

Communication with the checker in Logical Images Input data. are transmitted in the file zad_input.txt. The data format is as follows:

```
<number-of-rows> <number-of-columns>.
<description-row1>.
<description-row2>.
...
<row-descriptionK>.
<description-column1>.
...
<description-columnM>.
```

(row and column descriptions are a string of space-separated numbers) Your program is supposed to generate a zad_output.txt file containing a decoded image in which the zeros are . and the ones are #.

Example of an input:

```
5 5
5
1 1 1
3
2 2
5
2 2
1 3
3 1
1 3
2 2
```

Example of an output:

```
#####
#.#.#
.###.
##.##
#####
```

Communication with the checker in Sokoban The description of the board is transmitted is in the file zad_input.txt. The file represents a map of the warehouse. For a board of size N × M, the file contains N lines, and each line consists of M characters describing the fields of the board: . stands for an empty field, W for a wall, K for a warehouseman, B for a box, G the target field, * the box on the target field and + the warehouseman standing on the target field. There are the same number of crates and target fields on the board and exactly one warehouseman.

The solution found is to be written by the program in the file zad_output.txt in the form of a single line containing a string of characters specifying the movements of the warehouseman: U up, D down, L left and R right.

The solution will be considered correct if one of the shortest sequences of moves correctly positioning the boxes is returned.

Example of input:

```
WWWWWWW
W.GWWWW
W..WWW
W*K..W
W..B.W
W..WWW
WWWWWWW
```

Example of output:
`DLURRRDLULLDDRULURUULDRDDRRULDLUU`

The checker allows you to play a sequence of moves if you run it with the verbose option, e.g. python validator.py –verbose zad2 python zad.py.

Communicating with the validator in Commando

The input provided in the zad_input.txt file is a description of the maze:

```
######################
# G   G              #
#    #       #S      #
# S  #       #       #
######### ############
#    G#           G  #
##   ##    ###########
#    #       S       #
# ## ###########     #
#    #            S  #
#S         ####      #
######################
```

If you want, you can assume that the maze is always surrounded by walls. The program is to write to the file zad_output.txt one line containing the solution, for example, for the above maze `LLUULLULLLLURRULUUULLLLLLLLLRRRRR`. After specifying flag `--verbose`, the checker shows the board after each step of the solution.