AI 1: Lab Assignment 1

Agents and Search

Instructions

1. This is a ***team assignment*** to be completed in a team of two or three students and consists of a Python coding element and a report.
2. Answer all questions and ***formulate your answers concisely and clearly***.
3. Remember ***it is not allowed to use code not developed by your team***. Doing so constitutes plagiarism. If you use external sources (videos, websites, discussion fora, etc.) to develop your code make sure to ***clearly refer to them in your report and in your code***. Failing to disclose these sources constitutes again an instance of plagiarism.
4. ***Reports need to be written in LaTeX*** and submitted as a pdf ***separately from the code***. A template is available on Brightspace. The text should be ***authored by your team and your team only***.
5. Comply with the deadline provided on Brightspace. ***Deadlines are strict*** (penalties apply for late submissions, see below).
6. The code should be submitted in a ***separate zip file***.

Assessment The grade of this lab assignment will counts for ***10% of your final grade***. We ***subtract*** 2^{n-1} grade points for a submission that is between $n-1$ and n days late ($n \geq 1$).

Questions? Get in touch with us! You can do so during labs, tutorials, using the discussion boards on Brightspace or the helpdesk email `ai12023@rug.nl`. Make sure you read the instructions to contact the helpdesk that are provided on Brightspace (under 'Course overview'). Only emails sent according to the instructions will be replied to.

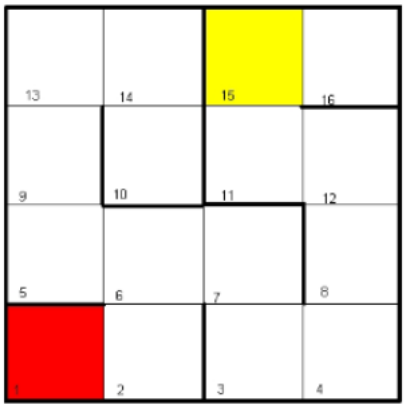

Figure 1: Maze

1 Theory assignment [30 pts]

1.1 PEAS descriptions [10 pts]

Study the material on PEAS descriptions from the book.

Questions:

a) Write a PEAS-description for the following agents:

- A reversi/othello computer (see `http://en.wikipedia.org/wiki/Reversi`)
- A robotic lawn mower (see `http://en.wikipedia.org/wiki/Robotic_lawn_mower`).

b) Characterize each domain as fully observable / partially observable, deterministic / stochastic, episodic / sequential, static / dynamic, discrete / continuous and single agent / multi-agent.

1.2 Maze [20 pts]

Given is a simple maze environment (see Figure 1 above). An agent in this environment can choose from 4 actions: move one step north, east, south, or west. Here is the problem definition:

States: The squares of the maze.
Actions: Step in direction N=north, E=east, S=south, or W=west.
Goal Test: Is the agent in the yellow colored square (square 15)?
Path cost: Each step has cost 1.
Initial state: The agent is in the red square (square 1).

Furthermore, the agent is provided with a collision detector that indicates whether it has just collided against a wall. If the agent hits a wall, then it knows that the corresponding direction is blocked and it will try the next direction from the set of possible actions. The agent always tries actions in the order [N, E, S, W].

First we try to solve the maze using the following *open list*[1] version of depth first search (DFS). The pseudo code for this implementation is given below.

[1] *Open list version* means we are using the tree-search variant of the algorithm, which does not keep track of visited states in an *explored set* (a.k.a. *closed list*). It is the version that does not check before expansion if the state is in the frontier or is already explored.

```
1  procedure mazeDFS(maze, start, goal):
2     stack = []
3     stack.push(start)
4     while stack is not empty:
5        loc = stack.pop()
6        if loc == goal:
7          print "Goal found"
8          return
9        for move in [W,S,E,N]:   # in this order!
10         if allowedMove(loc, move):
11            stack.push(neighbour(maze, loc, move))
12    print "Goal not found"
```

For example, the actions of the algorithm in the search for a DFS path from square 5 to square 10 (i.e. we call `MazeDFS(maze, 5, 10)`) is *N, N, E, S*.

Questions:

a) Why is it not possible to use `mazeDFS()` to find a path from the red square to the yellow square? Where does it go wrong?

b) How should the (pseudo-)code be modified such that it is guaranteed to always find a solution (if one exists)?

c) In which order are the states visited by the modified algorithm for the call `mazeDFS(maze, 1, 15)`?

d) In which order are states visited by the modified algorithm for the call `mazeDFS(maze, 1, 15)` if we change the order of the list at line 9 into `[E,W,S,N]`?

e) If we replace the stack in the original algorithm (meaning `[W,S,E,N]` at line 9) by a FIFO queue, then the algorithm is turned into (the open list version of) Breadth First Search (BFS). Will this algorithm always find a solution (if one exists)? Explain your answer.

f) In which order are states visited by the open list BFS algorithm for the call `mazeDFS(maze, 1, 15)` (again, using the unaltered pseudo-code)? Only give the first ten steps!

g) Can we reduce the number of visited states of the BFS algorithm? If yes, how should the program be modified?

h) Would you use (a variation of) DFS or (a variation of) BFS for path searching in large mazes? Explain your answer.

2 Programming assignment - 3D maze [60 pts]

In this first programming assignment we consider a three-dimensional maze problem. On Brightspace, you can find the file `maze.zip`. After extracting it, you will find a directory `maze` with files in it. Go into this directory. You can find the following files in the directory:

***.maze:** These files are simple textfiles with information about the maze :

- Coordinates (x, y, z) denote cells in the maze, with X, Y and Z in the directions as pictured in figure 2. The top left cell on floor 0 has coordinate (0,0,0).

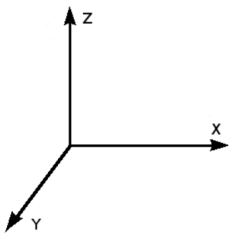

Figure 2: Coordinate system in the maze

- The current position is indicated by the capital letter X. The letter G in cell (1,1,0) in default.maze denotes the goal cell.
- If you run the code, the maze will also be printed with the coordinates at the bottom of each cell.
- Connections between rooms are indicated by spaces. Thus, it is possible to go from (0,3,1) to (1,3,1), but it is not possible to go from (0,2,1) to (1,2,1).
- If a room has a staircase going up, then this is indicated by the letter U (Up). The letter D (Down) denotes a staircase going down.
- All connections between rooms are symmetric: if (x, y, z) has a staircase going up, then (x, y, z +1) has a staircase going down.
- So, in the maze it is possible to go up from (1,3,0) to (1,3,1), and vice versa.
- Note that oblique moves (moves along a diagonal) are not allowed. The cost for going up is 3, for going down is 2 and sideways is 1.

maze.py: This file defines the Room and Maze class. It shouldn't be necessary to change these files. Look over the available methods for the Room class, as they may be useful - `Room.get_connections()` in particular. Use that function to provide you with the order in which to visit neighboring states.

state.py: State is a class to keep track of the current path and cost. Feel free to add things if needed.

maze_solver.py: File where the functions for the search algorithms are. The open list version of DFS and BFS is already implemented.

main.py: File to read arguments from command line or start solver with default options

fringe.py: Fringe is a wrapper around the queue library of python in order to track some statistic. This can be used to see how many states were added to the queue and how maney states where visited.

You can run it with: `python3 main.py [BFS|DFS] [mazefile]`. When the program finds a solution it will print the fringe statistics and the path. It will also print the maze, the path displayed in it and the cost in the upper left corner of a cell.

Questions:

a) Try to run default.maze with the depth first search algorithm (**python3 main.py DFS**). Will it find a solution? If not, why not?

b) Try to run BFS.maze with the breadth first search algorithm. Will it find a solution? If not, why not?

c) Fix (if needed) the BFS and DFS algorithms so that they will find a solution in the mentioned cases.

d) Implement the uniform cost search algorithm (make sure that you can call it with: **python3 main.py UCS greedy_astar.maze**).

e) Run two_paths.maze with the BFS and uniform cost search algorithm. Why are the paths that the algorithms have found different?

f) Implement the greedy search algorithm (make sure that you can call it with: **python3 main.py GREEDY greedy_astar.maze**).

g) Implement the A* search algorithm (make sure that you can call it with: **python3 main.py ASTAR greedy_astar.maze**).

h) Why are the paths that the A* and the greedy search algorithm have found different?

i) Implement the iterative deepening search algorithm (make sure that you can call it with: **python3 main.py IDS**). Note: recursion can be useful here, but is not needed. If you decide to use recursion, make sure that you print the path and cost.