CISC 3665, Spring 2014

Assignment II: Introduction to Game AI and Agents

Instructions

This is the assignment for unit II, Introduction to Game AI and Agents. It is worth 5 points (i.e., 5% of your term grade).

The assignment is due on **March 6**.

This assignment is based on the reading distributed in class (ch 2 of *An Introduction to Multiagent Systems*, by Michael Wooldridge).

The assignment has two parts:

- Part 1 is a programming part. This must be completed using the Processing environment (see lab1-1). This part must be submitted as a **zip file** containing the Processing folder in which you created your sketch.
- Part 2 is a written component. This part must be submitted on paper as a **printout**.

Preparation

- Read the chapter entitled Intelligent Agents (from An Introduction to Multiagent Systems by Michael Wooldridge), which was handed out in class.
- Note the section about **Tileworld** on pages 37-38 of the book chapter. Download the sample Processing code from the class web page called **tileworld0.zip**. This code simulates a simplified version of Tileworld, with a simple agent (blue circle), moving around a 40 × 60 grid of squares. There is one obstacle (grey rectangle), one hole (black rectangle) and one tile (magenta rectangle) in the world. The agent moves randomly, in either of the 4 compass directions (north, south, east, west); it moves one grid square per iteration through the Processing draw() loop. This version of Tileworld is simpler than the one described on pages 37-38 of the book chapter because the holes do not change location while the game runs (we'll get to the more complex version later in the semester).
  Run this code to see how it works. Notice that the agent does not sense any of the elements in its environment, and so it may randomly move over the obstacle, hole or tile. Your job with this assignment is to fix this and to make your agent behave intelligently.

1 Programming component: Simple Agent (*4 points*)

1.1 Modify the **tileworld0.pde** code so that your Tileworld contains multiple obstacles, holes and tiles. The sample code only has one of each. You need to figure out how you are going to keep track of multiple obstacles, holes and tiles in such a way that you can (see below) easily detect if the agent has run into any of them. You will probably want to create an array that keeps track of where each of the elements in the Tileworld are — the agent, the obstacles, the holes and the tiles. Initialize all your obstacles, holes and tiles to randomly placed locations around the Tileworld. Make sure that any single grid square contains either an obstacle, a hole, a tile, or none of the above; i.e., you cannot place an obstacle, hole and/or tile in the same grid square.
(*1 point this part*)

1.2 Modify the code to give the agent a **perceive**() function. This function should let the agent look for obstacles, holes and tiles in each of the four compass directions. Assume that the agent has 3 types of "sensors": one for detecting obstacles, one for detecting holes and one for detecting tiles; and that the agent has 4 sensors of each type, one for north, south, east and west. Decide what the agent's

sensor range is, i.e., what is the maximum number of grid squares it can see in each direction. You may want to define different ranges for each type of sensor.
Each of the agent’s 12 sensors should be set as follows:
= 0 if there is nothing ahead between the agent and its maximum sensor range
> 0 if there is an obstacle/hole/tile ahead, and the value is the number of squares ahead
(*1 point this part*)

1.3 Modify your code so that it fixes the problem of the agent ignoring obstacles. If the agent tries to move into a square that has an obstacle in it, then the agent should not move.
(*0.3 points this part*)

1.4 Modify your code so that it fixes the problem of the agent ignoring holes. If the agent moves into a square with a hole in it, then the agent falls down the hole and dies and the game is over.
(*0.3 points this part*)

1.5 Modify your code so that the agent, instead of moving completely randomly, moves a bit more *rationally*; i.e., your agent should try to survive. If the agent senses a hole ahead, it changes its direction so that it does not fall down a hole and die. You could use a flag to include both behaviors (i.e. allowing gameover from 1.4 via a toggle)
(*0.4 points this part*)

1.6 The object of the game is for the agent to find all the tiles in the world and push the tiles into holes. If the agent is next to a tile and moves toward that tile, then both the agent and the tile move ahead one grid square in that direction. See the example below (where A is the agent and T is a tile):

starting position:

	T		
	A		

agent moves north, pushing tile:

	T		
	A		

Modify your code so that if the agent is next to a tile, it will push it as illustrated above and according to the following rules:

- If there is an obstacle in the square ahead of the tile, the agent (and the tile) cannot move in that direction.
- If there is another tile in the square ahead of the tile, the agent (and the tile) cannot move in that direction; i.e., we treat a second tile in the agent’s path like an obstacle (our agent is weak and can only move one tile at a time).
- If there is a hole in the square ahead of the tile, the tile falls down the hole and the agent earns a dollar.

(*1 point this part*)

2 Written component: Simple Agent (*1 point*)

The agent from the simplified Tileworld that you coded in the first part of this assignment has two states: STOPPED and RUNNING. Later (not part of this assignment), you will write code for a more intelligent agent. A more intelligent agent existing in Tileworld, even the simple one, will need to have more than two states in order for the agent to make choices about what to do next.

For this part of the assignment, you will define a set of states for your more intelligent agent and analyze the utility of these states. (See the lecture slides from lec2-2)

You already know what the agent's set of actions are, e.g.: $Ac = \{\alpha_0, \alpha_1, \alpha_2, \alpha_3\}$, where α_0 means move north, α_1 means move west, α_2 means move south, and α_3 means move east.

You also know the constraints on the agent's actions:

- the agent cannot move into a square where there is an obstacle
- the agent will die if it falls down a hole
- the agent cannot push a tile into an obstacle or another tile
- the agent will earn a dollar if it pushes a tile into a hole

So now, define some states and utilities.

Here is an example to get you started (where A is the agent and H is a hole):

	H	
	A	

e_0

	H	
		A

e_1

	H	
	A	

e_2

	H	
A		
✓		

e_3

	H	

e_4

If the agent is in state e_0 and executes action α_0 (move north), then it will end up in state e_4 and fall down the hole and die. If the agent is in state e_0 and executes action α_1 (move west), then it will end up in state e_3. If the agent is in state e_0 and executes action α_2 (move south), then it will end up in state e_2. If the agent is in state e_0 and executes action α_3 (move east), then it will end up in state e_1.

You should also assign a (comparative) utility to each of these states. Since the agent dies in state e_4, then there should be negative utility associated with that state and with any action that takes the agent into that state. The remaining states (e_0, e_1 and e_2) are all equally better than e_4, since the agent does not die in any of these states. So any action taking the agent into any of these states should be given higher utility.

So, we can say:

$u(e_0 \xrightarrow{\alpha_0} e_4) = -1$
$u(e_0 \xrightarrow{\alpha_1} e_3) = 1$
$u(e_0 \xrightarrow{\alpha_2} e_2) = 1$
$u(e_0 \xrightarrow{\alpha_3} e_1) = 1$

Complete a similar analysis for obstacles and tiles, starting with states *e₅*, *e₆*, *e₇*, *e₈* and *e₉*, shown below. (A is the agent, H is a hole, T is a tile, and X is an obstacle.)

	X	
	A	

e_5

	T	
	A	

e_6

	H	
	T	
	A	

e_7

	X	
	T	
	A	

e_8

	T	
	T	
	A	

e_9

Note that I expect that your answer will include: drawings of additional states, and comparative utilities of the transitions between states (as in my example above). I don't care what the actual utility numbers are—we only really care here about the relative values (i.e., which states are better than others, which states are harmful, etc.).