ARTIFICIAL INTELLIGENCE (E016330)
ALEKSANDRA PIŽURICA
GHENT UNIVERSITY
2019/2020 AY

Assistants: Shaoguang Huang, Srđan Lazendić,
Xian Li, Laurens Meeus, Nina Žižakić
E-mail: ai@lists.ugent.be

Practical Session 0

1 Starting to use Python

We will use Python a lot throughout this course. Therefore, it is important to get familiar with its basics. It is useful to have Python installed on your own computer, although most of the exercises can be done online, via *Google Colab* – a free cloud service to run Python notebooks:
https://colab.research.google.com/notebooks/welcome.ipynb
Namely, you will only need a local installation of Python for this practical session, in sections 3 and 4. For those exercises, it is possible to use UGent computers in order to avoid installation whatsoever.

If you do, however, decide to install Python, you have many options. The easiest is to install it via Anaconda Distribution (https://www.anaconda.com/distribution/), which packs Python with other useful tools like PIP python package management system, Jupyter Notebook (a browser-based Matlab-like Python IDE), and the most commonly used Python libraries like NumPy, pandas, etc. (Hint: If you're a newbie, when installing Anaconda, choose the option to add Python to PATH.) If you think you will continue using Python for future projects, we recommend installing the PyCharm IDE.

Now it is time to dive into using Python. If this is the first time you program in Python, or if you need to refresh your memory, you should go through the Python_Basics.ipynb notebook. You will here learn how to use the basic operators, types, built-in structures such as lists, tuples, sets, dictionaries; how to write scripts and functions, and how to use object-oriented concepts such as classes and objects. If all of this sounds familiar and you consider yourself to be an expert in Python, you may skip this part.

2 Agents and environments in Pacman world

In lectures you have learned the theory about agents such as a simple reflex agent or simple reflex agent with a state. In order to obtain an even better understanding of them, we turn to Pacman. In the Pacman game, his arch enemies are ghosts. They can eat Pacman unless he has recently eaten a large flashing dot, in which case he can eat them.

Talking in technical terms, ghosts are agents in the game environment. Take a look at the file ghosts.py, under the search.zip file. There are two implementations of the ghost agent, RandomGhost, and DirectionalGhost. RandomGhost has been fully implemented – it just randomly selects a direction which to take when he reaches a junction on the map. In order to help the ghosts find Pacman more easily, we can tell each ghost to go in the direction in which he will minimise the distance to Pacman.

- Try to implement this behaviour in the getDistribution method of the DirectionalGhost. You can differ the ghost agents in the Pacman game by changing the -g variable, for example:

```
python2 pacman.py -g RandomGhost -l mediumScaryMaze
python2 pacman.py -g DirectionalGhost -l mediumScaryMaze
```

- What would be the considered as the environment in this Pacman setup?

3 State-Space Search with Pacman

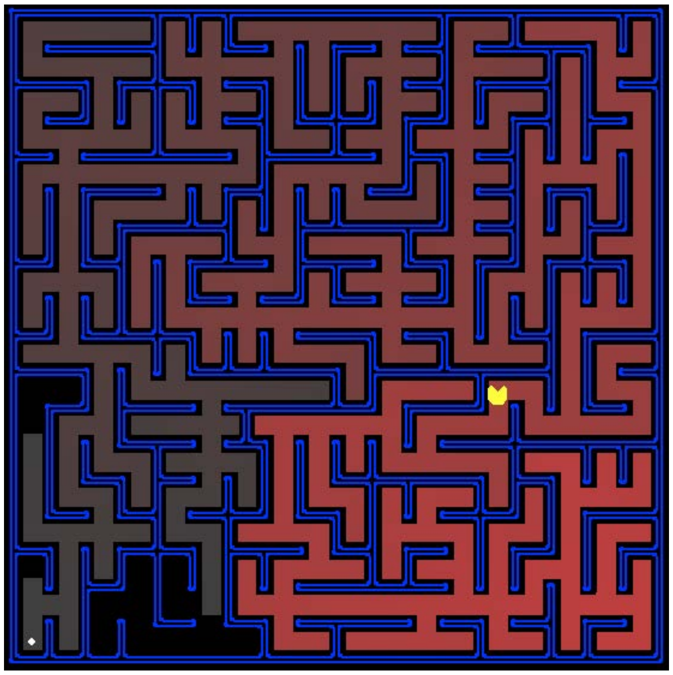

Figure 1: The states explored in breadth-first graph search are colored, with their shade indicating the order in which they were dequeued from the fringe.

For this part of the practical session, we will continue to use Python files under the search.zip. Different algorithms such as depth-first, breadth-first, uniform cost, and A^* search algorithms are implemented. These algorithms are used to solve navigation and traveling salesman problems in the Pacman world.

1. **Depth-first search algorithm** The depth-first search (DFS) algorithm is implemented in the depthFirstSearch method in the file search.py. Check that the implementation clears the following mazes without any issues by typing:

```
python2 pacman.py -l tinyMaze -p SearchAgent
python2 pacman.py -l mediumMaze -p SearchAgent
python2 pacman.py -l bigMaze -z .5 -p SearchAgent
```

Pacman's maze will color the explored states red, with brighter red meaning earlier exploration, i.e., their shade is indicating the order in which they were dequeued from the fringe.

- Is the order of exploration what you would have expected?

- Does pacman go through all the explored states on the way to the goal?
- Can the number of explored states in the solution vary, and if it can, what does the number of explored states depend on?

2. **Breadth-first search algorithm** Taking the DFS as the example, try to implement the Breadth-first search (BFS) in the search.py file under the breadthFirstSearch method. Note that, track of visited states should be kept so they wouldn't get explored again. Test your code in the same way you did for depth-first search:

```
    python2 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
    python2 pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

   If pacman moves too slow for your taste, Pacman's alter ego PacFlash is here:

```
    python2 pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 --frameTime 0
```

   - Does BFS find a minimum cost solution?

3. **Uniform-cost search** While BFS will find a path to the goal which requires the fewest actions, sometimes we want to find paths that are the best in some other sense - examples of this are mediumDottedMaze and mediumScaryMaze. The uniform-cost search algorithm is implemented in the method uniformCostSearch in search.py. The three following examples differ only in the pre-defined cost functions.

```
python2 pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
python2 pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python2 pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

   Note that we get very low and very high path costs for the last two examples respectively due to their exponential cost functions. If you want to know more, you can find the code in searchAgents.py.

4 Multi-Agent Search with Pacman

So far, we were considering only one agent at a time. In practice, we will often need to work with multiple agents, where each agents' behaviour depends on the behaviour of the other agents. For this part, you will be working with the code under the multiagents.zip file.

Take a look at the multiagents.py file. Try to understand how the MinimaxAgent and AlphaBetaAgent are implemented, and based on that, implement the methods of ExpectimaxAgent. You can test the two implemented agents as follows:

```
python2 pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

MinimaxAgent and AlphaBetaAgent both assume that you are playing against an adversary who makes optimal decisions. This is not always the case. ExpectimaxAgent is useful for such cases, because it models probabilistic behavior of agents who may make suboptimal choices. You may assume you will only be running against an adversary which chooses amongst their getLegalActions uniformly at random.

To see how the ExpectimaxAgent behaves in Pacman, run:

```
python2 pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
```

You should now observe a more cavalier approach in close quarters with ghosts. In particular, if Pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try. Investigate the results of these two scenarios:

```
python2 pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
python2 pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
```

You should find that your ExpectimaxAgent wins about half the time, while your AlphaBetaAgent always loses. Make sure you understand why the behavior here differs from the minimax case.

The correct implementation of expectimax will lead to Pacman losing some of the games.

All of the Pacman exercises were taken from the Berkeley course on Artificial Intelligence: https://ai.berkeley.edu.