Artificial Intelligence – Programming Assignment 1

UNIZG FER, academic year 2018/19

Handed out: 18.3.2019. Due: 31.3.2019. at 23.59.

Uvod

In this lab assignment you will help Pacman find paths through his maze world in order to reach a particular location as well as to collect food efficiently. You will implement general search algorithms and model their heuristics, applying them to scenarios from the Pacman universe.

The code that describes the behavior of Pacman's universe is already functional, while you simply need to implement the algorithms which will control the way Pacman moves. The code you will be using can be downloaded as a zip archive at the web page of the university here or at the github repository of the class here.

The code for the project consists of Python files, a part of which you just need to read and understand, a part of which you need to modify yourselves and a part that you can ignore. After you download and unpack the zip archive, you will see a list of files and folders which we briefly describe below:

Files you'll edit:	
search.py	Where all of your search algorithms are
searchAgents.py	Where all of your search-based agents are
Files you should look at:	
pacman.py	The main file that runs Pacman games
game.py	The logic behind how the Pacman world works
util.py	Useful data structures for implementing search algorithms
Supporting files you can ignore:	
graphicsDisplay.py	Graphics for Pacman
graphicsUtils.py	Support for Pacman graphics
textDisplay.py	ASCII graphics for Pacman
ghostAgents.py	Agents to control ghosts
keyboardAgents.py	Keyboard interfaces to control Pacman
autograder.py	Project autograder (Berkeley)
testParser.py	Parses autograder test and solution files (Berkeley)
testClasses.py	General autograding test classes (Berkeley)
test_cases/	Directory containing the test cases for each question
searchTestClasses.py	Project 1 specific autograding test classes (Berkeley)

The code for the lab assignments was adapted from the course "Intro to AI" at Berkeley, which allowed the usage of their Pacman environment for other universities for educational purposes. The code is written in Python 2.7, which you require an interpreter for in order to run the lab assignment.

After you've downloaded and unpacked the code and positioned your console in the subdirectory where you have unpacked the archive, you can test the game by typing:

```
python pacman.py
```

However, artificial intelligence cannot rely on human control - Pacman has to be able to independently and efficiently fight all problems he might encounter on the way to finding food! A naive example of artificial intelligence is always moving in the same direction regardless of the map layout, which you can check by running the following:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

In this case, we passed the map *testMaze* and the logic that moves pacman west *GoWestAgent* as arguments to the game. As you can assume yourselves, this approach might hit a few roadbumps on the way when there is any turning required to reach the goal, as is evident from the following example:

```
python pacman.py --layout tinyMaze --pacman GoWestAgent
```

Through this lab assignment you will allow Pacman not only to navigate through the previous two mazes, but through any other he might face. The pacman game has a lot of possible arguments, and we recommend you study them yourselves by typing:

```
python pacman.py -h
```

All the commands you need to execute throughout the instructions for this lab assignment will be present in the text file *commands.txt*. In case you are running a UNIX-based system, you can run all the commands sequentially by typing *bash commands.txt* in your console.

Due to a known bug that has only been reproduced on laptops with the Ubuntu 14.04 operating system, in-game animations have been disabled. The bug manifests itself in a way that soon after running the game, the active user is logged out and all active windows are closed. In case you want to try if Pacman works on your computer with the full graphic options enabled, we advise you to save all of your current work in case of the possible system reboot. We do not assume any responsibility for possible unsaved progress in case the game crashes.

To activate the animations, it is required to uncomment the line 218. in the file *graphicsUtils.py*. The contents of the line are *edit(id, ('start', e[0]), ('extent', e[1] - e[0]))*. If you manage to run and play through the original version of pacman after that change (by running *python pacman.py*), the bug does not occur on your computer. In case the bug does occur, please contact us in order to save your PC configuration for reference in further classes.

The lab assignment consists of eight subtasks, and the correct implementation of each one of them carries the same weight. To ease the grading, each of the subtasks carries three points, with a total sum of 24. Your final points will be scaled and the actual maximum for the first lab assignment is 6.25.

Problem 1: Finding food by depth first search

In the file searchAgents.py you will find a fully functional search agent (class *SearchAgent*), which first plans out the route through Pacman's world, and then goes through the route step by step. Your job is to implement the search algorithms which formulate the plans. Firstly, check if the search agent is working as intended by running the following command:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

The command passes the search algorithm *tinyMazeSearch* to the search agent. The search algorithm is implemented in search.py, and by examining the code you will conclude that it works only in the case of the map that we provided - now it is the time to write the code which will find the solution for any map layout.

The pseudocode for the search algorithms can be found in the lecture slides - keep in mind that you don't just need to find the path to the goal state, but you also need to be able to reproduce the steps once you've found it. The search node needs to contain not only the information of the current state, but also the steps required to reach the state. You should also use a data structure to keep track of all the visited states in order to ensure completeness of the DFS algorithm.

Note 1.: All of your methods need to return a list of valid actions that will lead the agent from the start to the goal. Validity in this case means that in no case should Pacman walk through walls.

Note 2.: Make sure to use classes *Stack*, *Queue* and *PriorityQueue* classes in your implementation, since the automatic testing relies on them, and that way you can validate your solution easier.

Hint 1.: The lab assignment can be solved by simply filling out all the marked methods in the code - howevery by solving in that fashion you will encounter a lot of code duplication. Since DFS, BFS, UCS and A* differ only in the selection of the next node to be expanded, try to implement a single generic search method which can be configured with an algorithm-specific queueing strategy. Your implementation **does not** need to be in this form to get full points in the assignment.

Implement the depth-first search (DFS) algorithm in the *depthFirstSearch* method in the file *search.py*. Your implementation should clear the following mazes without any issues:

```
python pacman.py -l tinyMaze -p SearchAgent

python pacman.py -l mediumMaze -p SearchAgent

python pacman.py -l bigMaze -z .5 -p SearchAgent
```

Pacman's maze will color the explored states red, with brighter red meaning earlier exploration. 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 your solution vary, and if it can, what does the number of explored states depend on?

A sample of solving the *bigMaze* with DFS can be found at Figure 1.

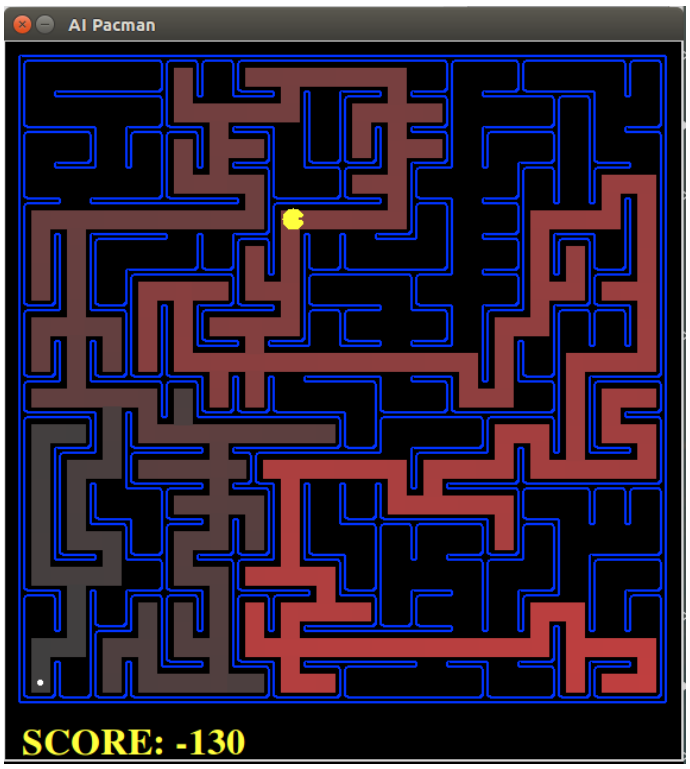

Figure 1: An example of depth first search

Problem 2: Breadth First Search

Implement the breadth-first search (BFS) algorithm in the *breadthFirstSearch* method in the file *search.py*. Again, keep track of visited states so you wouldn't explore them again. Test your code in the same way you did for depth-first search:

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

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

Does BFS find a minimum cost solution? If not, you might have a bug in your code.

If pacman moves too slow for your taste, Pacman's alter ego PacFlash is here - to make Pacman show his true face, enable the option *–frameTime 0* like in the following example:

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

In case you've written the code generically, your search code should work equally well when applied to the eight-puzzle:

```
python eightpuzzle.py
```

Problem 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 toll of the fear from going through ghost infested areas is surely greater than calm and peaceful areas, as well as areas rich in food are definitely easier on the spirit than the cold, dark and damp walls of the maze - and as such, every reasonable Pacman adapts his route accordingly.

Implement the uniform-cost search algorithm in the method *uniformCostSearch* in *search.py*. You should easily clear each of the three following examples, which differ only by the pre-defined cost functions.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

You should 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*.

Problem 4: A* search

Implement the A* search algorithm in the *aStarSearch* method in the file *search.py*. A* takes a heuristic function as an argument, while a heuristic function accepts two arguments - the state in the search problem (the main argument) and the problem itself (for reference information). The *nullHeuristic* in the file *search.py* is a trivial example.

You can test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic - implemented already for you as *manhattanheuristic* in *searchAgents.py*.

```
python pacman.py -l bigMaze -z .5 -p SearchAgent
                                    -a fn=astar,heuristic=manhattanHeuristic
```

(Write the command in a single line)

The solution using A* search should be marginally faster than uniform-cost search - our implementation of A* solves the maze with 549 nodes expanded, while UCS expands 620. The numbers may vary depending on ties in priority. Test your implementation on the *openMaze* problem - what differs compared to other search strategies?

Problem 5: Finding All the Corners

This problem builds on the solution of problem 2

The true power of A* may not be apparent from simple problems - this is why we will formulate a new problem and then design an adequate heuristic for it. In *cornery* (not an actual word) mazes there are four food dots, one in each corner of the maze. Our new problem is to find the shortest path through the maze that takes us through all the four corners. Keep in mind that for some mazes like the *tinyCorners* the shortest path does not always go through the closest food first! For reference, the shortest path through *tinyCorners* takes 28 steps.

To start, we need to formulate the cornery maze problem. Implement the *CornersProblem* search problem in *searchAgents.py*. You will need to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. Your search agent should easily solve the following problems:

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem

python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
```

Your implementation should in no case include irrelevant information such as ghost locations, additional food etc. In no case should you use the *GameState* class as a search state - your code will be extremely slow (and also wrong).

Hint 2.: the only parts of the game state you need to reference in your implementation are the starting Pacman position and the location of the four corners.

Our implementation of the BFS solves the problem by expanding under 2000 nodes on the *mediumCorners* problem. However, heuristics can further reduce the number of expanded nodes.

Problem 6: A Heuristic for Finding All the Corners

This problem builds on the solution of problem 4

Implement a non-trivial, consistent heuristic for the *CornersProblem* in the method *cornersHeuristic*. Test your implementation by running:

```
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

AstarCornersAgent is short for -p SearchAgent -a fn=aStarSearch, prob=CornersProblem, heuristic=cornersHeuristic.

Remember the requirements for a heuristic to be good. Pay attention to the fact that your heuristic has to be zero in each goal state and should never return values less than zero. You can judge the efficiency of your heuristic based on the number of expanded nodes.

In this case, your heuristic should be better than breadth-first search. Your points on this assignment will depend on the number of expanded nodes, and the maximum you will be eligible for is given in the following table:

Nodes expanded	Max. points
> 2000	0/3
$\leqslant 2000$	1/3
$\leqslant 1600$	2/3
$\leqslant 1200$	3/3

Problem 7: lazy, insatiable Pacman

This problem builds on the solution of problem 4

Now we'll make things a bit more difficult and solve a complicated search problem - Pacman needs to eat all the food in as little steps as possible. To do this, we will need to define a new search problem which formalizes the problem of collection all the food - already written in the class *FoodSearchProblem* in the file *searchAgents.py*.

In the future, ghosts and power pellets will create additional complications, but currently we focus just on the walls, food and Pacman. If your implementation of search algorithms is generic enough, it should find the minimum cost path of cost 7 by using *nullHeuristic* on the following problem:

```
python pacman.py -l testSearch -p AStarFoodSearchAgent
```

AStarFoodSearchAgent is short for -p SearchAgent -a fn=astar, prob=FoodSearchProblem, heuristic=foodHeuristic.

Even though we solved the previous problem quickly, things get more complicated even with small mazes such as *tinySearch*. As a reference, to find the optimal cost path of 27 uniform-cost search takes 2.5 seconds!

```
python pacman.py -l tinySearch -p SearchAgent -a fn=ucs,prob=FoodSearchProblem
```

Your task is to fill out the *foodHeuristic* in the file *searchAgents.py* with a consistent heuristic for the *FoodSearchProblem*. Test your implementation on the *trickySearch* maze.

```
python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

The uniform-cost search algorithm finds the optimal path in 13 seconds with expanding just over 16000 nodes.

Any non-trivial non-negative heuristic will receive 1 point. Depending on how few nodes you expand, the maximum points you can achieve are as in the following table:

Nodes expanded	Max. points
> 15000	1/3
$\leqslant 15000$	2/3
$\leqslant 12000$	3/3
$\leqslant 7000$	4/3

Pay attention to that your heuristic has to be consistent, otherwise you will not receive any points for this part of the assignment!

Problem 8: Suboptimal Search

Sometimes, even with A* and a good heuristic, finding the optimal path through all food is not simple. In some cases, we are satisfied with a reasonably good path we find quickly. In this problem, you will implement the logic which will always greedily eat the closest food dot. The class *ClosestDotSearchAgent* is already implemented in *searchAgents.py*, however it lacks the key method which finds the path to the closest food dot.

Implement the *findPathToClosestDot* method in the file *searchAgents.py*. Your solution should solve the *bigSearch* in less than a second with the total cost of 350.

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
```

Hint 3.: The fastest way to complete *findPathToClosestDot* is to fill in the code in *AnyFoodSearchProblem*, which lacks a way to check if the state is a goal. Then, you need to use an appropriate search algorithm. Your solution should require very few code changes.