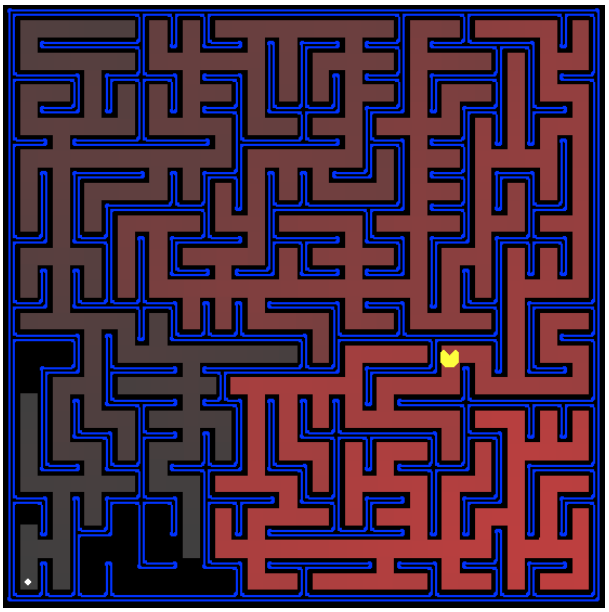
Künstliche Intelligenz und Data Science WS 2023/2024

Lab 1: State Space Search[1]

Due: November 12, 2023, 23:59 CET

Prof. Goran Glavas, Benedikt Ebing, Fabian David Schmidt
Chair for Natural Language Processing



Pacman visits Würzburg - Introduction

In this lab, we will help Pacman navigate through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms, experiment with heuristics, and apply all of it to Pacman´s world.

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. You can access and download the code from WueCampus. The code consists of several Python files, some of which you will need to read and understand, some of which you will edit, and some of which you can ignore. After downloading and unpacking the zip archive, you will see a list of files and folders shown in Table 1

Submission: You will edit the files `search.py` and `searchAgent.py`. Once you completed the assignment, you will submit a single zip archive containing all your work to WueCampus. Please do not change the other files.

Evaluation: Your our code will be autograded for technical correctness. Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation – not the autograder's judgements – will be the final judge of your grade. The lab assignment consists of eight subtasks, and the correct implementation of each one of them carries the same weight.

[1]The lab is based on "Intro to AI" from UC Berkley, which graciously permits the usage of their Pacman environment for other universities for educational purposes, and "Artificial Intelligence" from Univsersity of Zagreb.

Files you´ll edit:	
`search.py`	Where all of your search algorithms will reside.
`searchAgents.py`	Where all of your search-based agents will reside.
Files you might want to look at:	
`pacman.py`	The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this lab.
`game.py`	The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
`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
`layout.py`	Code for reading layout files and storing their contents
`autograder.py`	Lab autograder
`testParser.py`	Parses autograder test and solution files
`testClasses.py`	General autograding test classes
`test_cases/`	Directory containing the test cases for each question
`searchTestClasses.py`	Lab specific autograding test classes

Table 1: Files for the Pacman lab

Plagiarism: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. Submitting plagiarized work will result in failure of the lab, at the very least.

Getting Help: We encourage you to attempt to find solutions to your issues independently in the first instance. Nevertheless, if you get stuck, do not hesitate to contact Benedikt Ebing for assistance with your questions.

Technical Prerequisites: This lab can be solved using pure Python. However, upcoming labs might require additional dependencies. Hence, we recommend using Python 3.9 together with Pip or Conda via Miniconda. Installing Miniconda includes Python and Pip, so you only need to install one thing. Additionally, we recommend using an IDE like VS Code, IntelliJ PyCharm, or anything similar. For all of these components, you find detailed installation guidelines following the provided hyperlinks.

First Steps

Pacman resides in a vibrant, blue world filled with winding corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain. Once you have downloaded the code, unzipped it, and navigated to the subdirectory where it was extracted, you should be able to play a game of Pacman by typing the following at the command line:

```
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! The simplest agent in `searchAgents.py` is called the `GoWestAgent`, which always goes West (a trivial reflex agent):

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

But, things get ugly for this agent when turning is required:

```
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
```

Also, all of the commands that appear in this project also appear in `commands.txt`, for easy copying and pasting. In UNIX/Mac OS X, you can even run all these commands in order with `bash commands.txt`.

Q1: Finding Food by Depth First Search

Within `searchAgents.py`, you'll discover a complete implementation of the `SearchAgent`. This agent is capable of devising a path through Pacman's world and subsequently executing that path step-by-step. The search algorithms for formulating a path are not implemented – that's your job.

First, test that the `SearchAgent` is working correctly by running:

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

The command above tells the `SearchAgent` to use `tinyMazeSearch` as its search algorithm, which is implemented in `search.py`. Pacman should navigate the maze successfully.

Now it's time to write full-fledged generic search functions that assist Pacman in planning its routes! Keep in mind that a search node should include not only the current state but also all the information required to reconstruct the path leading to that state.

Important note: All of your search functions need to return a list of actions that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls).

Important note: Make sure to use the `Stack`, `Queue` and `PriorityQueue` data structures provided to you in `util.py`! These data structure implementations have particular properties which are required for compatibility with the autograder.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the selection of the next node to be expanded. Indeed, one possible implementation requires only

a single generic search method which is configured with an algorithm-specific queuing strategy. (Your implementation need not be of this form to receive full credit).

Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`. Write the complete version of DFS, which avoids expanding any already visited state. Your code should quickly find a solution for:

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

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q1
```

Q2: Breadth First Search

Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `search.py`. Again, write an algorithm that avoids expanding any already visited state. Test your code the same way you did for DFS.

```
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 least cost solution? If not, check your implementation.

Hint: If Pacman moves too slowly for you, try the option –frameTime 0.

Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

```
python eightpuzzle.py
```

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q2
```

Q3: Uniform Cost Search

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider `mediumDottedMaze` and `mediumScaryMaze`.

By modifying the cost function, we can influence Pacman's preferences in selecting different routes. For instance, we can assign higher costs to dangerous steps in areas infested with ghosts or lower costs to steps in food-rich areas. A logical Pacman agent should adapt its behavior accordingly in response to these cost changes.

Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in `search.py`. We encourage you to look through `util.py` for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

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

Note: You should get very low and very high path costs for the `StayEastSearchAgent` and `StayWestSearchAgent` respectively, due to their exponential cost functions (see `searchAgents.py` for details).

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q3
```

Q4: A* search

Implement A* graph search in the empty function `aStarSearch` in `search.py`. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The `nullHeuristic` heuristic function in `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 as `manhattanHeuristic` in searchAgents.py).

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

You should see that A* finds the optimal solution slightly faster than uniform cost search . What happens on `openMaze` for the various search strategies?

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q4
```

Q5: Finding All the Corners

Note: Make sure to complete Question 2 before working on Question 5, because Question 5 builds upon your answer for Question 2.

Pacman will only profit from the real power of A* with a more challenging search problem. Now, it's time to formulate it and design a heuristic to ensure Pacman's success.

In corner mazes, there are four dots, one in each corner. Our new search problem is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not). Note that for some mazes like `tinyCorners`, the shortest path does not always go to the closest food first! Hint: the shortest path through `tinyCorners` takes 28 steps.

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. Now, your search agent should solve:

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

To receive full credit, you need to define an abstract state representation that does not encode irrelevant information such as the positions of ghosts or the location of extra food. Avoid using a Pacman `GameState` as a search state because it will lead to slow and incorrect code.

An instance of the `CornersProblem` class represents an entire search problem, not a particular state. Particular states are returned by the functions you write, and your functions return a data structure of your choosing (e.g. tuple, set, etc.) that represents a state.

Furthermore, while a program is running, remember that many states simultaneously exist, all on the queue of the search algorithm, and they should be independent of each other. In other words, you should not have only one state for the entire `CornersProblem` object; your class should be able to generate many different states to provide to the search algorithm.

Hint 1: 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.

Hint 2: When coding up `getSuccessors`, make sure to add children to your successors list with a cost of 1.

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q5
```

Q6: A Heuristic for Finding All the Corners

Note: Make sure to complete Question 4 before working on Question 6, because Question 6 builds upon your answer for Question 4.

Implement a consistent heuristic for the `CornersProblem` in `cornersHeuristic`.

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

Note: `AStarCornersAgent` is a shortcut for

```
-p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
```

Grading: Your heuristic must be a non-negative consistent heuristic. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll be graded.

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q6
```

Q7: Lazy and ever hungry Pacman

Note: Make sure to complete Question 4 before working on Question 7, because Question 7 builds upon your answer for Question 4.

Now we'll solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we'll need a new search problem definition which formalizes the food-clearing problem: `FoodSearchProblem` in `searchAgents.py` (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. Solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. If you have written your general search methods correctly, A* with a null heuristic (equivalent to uniform-cost search) should quickly find an optimal solution to `testSearch` with no code change on your part (total cost of 7).

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

Note: `AStarFoodSearchAgent` is a shortcut for

```
-p SearchAgent -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic
```

You should find that UCS starts to slow down even for the seemingly simple `tinySearch`.

Fill in `foodHeuristic` in `searchAgents.py` with a consistent heuristic for the `FoodSearchProblem`. Try your agent on the `trickySearch` board:

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

Grading: Your heuristic must be a non-negative consistent heuristic. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. Depending on how few nodes your heuristic expands, you'll be graded.

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q7
```

Q8: Suboptimal Search

Sometimes, even with A* and a good heuristic, Pacman has a hard time finding the optimal path through all the dots. In these cases, we'd still like to help him out and find a reasonably

good path, quickly. In this section, you'll write an agent that always greedily eats the closest dot. `ClosestDotSearchAgent` is implemented for you in `searchAgents.py`, but it's missing a key function that finds a path to the closest dot.

Implement the function `findPathToClosestDot` in `searchAgents.py`.

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

Hint: The quickest way to complete `findPathToClosestDot` is to fill in the `AnyFoodSearchProblem`, which is missing its goal test. Then, solve that problem with an appropriate search function. The solution should be very short!

Please run the below command to see if your implementation passes all the autograder test cases.

```
python autograder.py -q q8
```