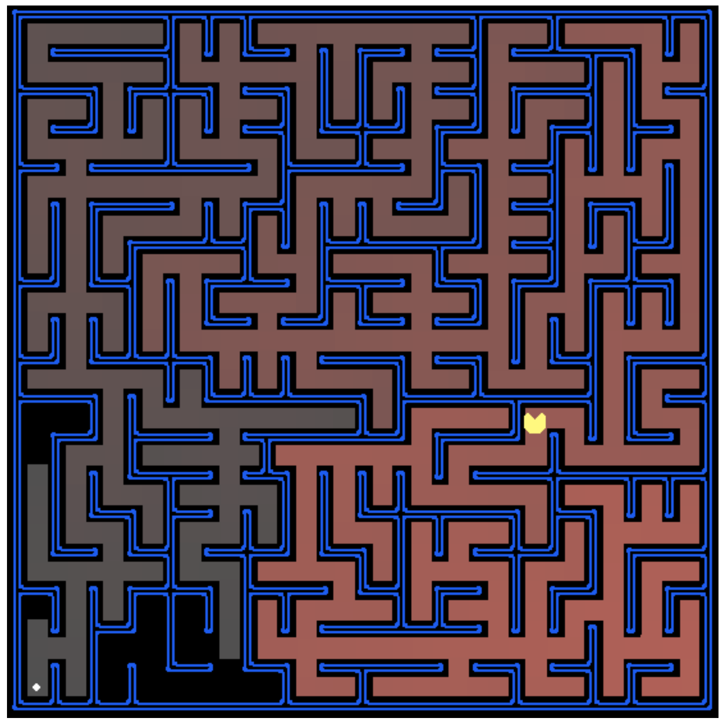
Programming Assignment One

Due Oct 9 by 11:59pm **Points** 35 **Available** after Sep 18 at 12am

Programming Assignment 1: Search (Individual assignment)

Modified version UC Berkeley CSC188 **Project 1 (https://inst.eecs.berkeley.edu/~cs188/fa20/project1/)**



All those colored walls,
Mazes give Pacman the blues,
So teach him to search.

Table of Contents.

Warning (Please read this)

We are aware that solutions to the original Berkeley project exist on the internet. **Do not use these solutions as this would be plagiarism.** To earn marks on this assignment you must develop your own solutions. Also please consider the following points.

- You are to implement the search algorithms presented in the course. These algorithms differ in subtle but important ways from other presentations of this material. If you implement your search based on other non-course material it might give the wrong answers. If you try to use solutions found on the internet the same problem might occur.

- Please do not implement your own "improvement" to the search algorithms: it will wreak havoc with the automarker.
- **Do not add any non-standard imports in the python files you submit** (all imports already in the starter code must remain). All imports that are available on teach.cs are considered to be standard.
- **Do not change any of the supplied files except for `search.py` and `searchAgents.py`**
- **Make certain that your code runs on teach.cs using python3**. You should all have an account on teach.cs and you can log in, download all of your code (including all of the supplied code) to a subdirectory of your home directory, and use the command `python3 autograder.py` and test it there before you submit. Your code will be graded by running it on teach.cs, so the fact that it runs on your own system but not on teach is not a legitimate reason for a regrade.
- No more test cases will be run on your code beyond the test cases used in the autograder. So the grade shown to you by the autograder will be a good predictor of your final grade on the assignment. **However**, we will look for certain things in the assignments (e.g., running them through code plagiarism checkers, looking at assignments that fail all tests, etc.). If we have **good reasons** we will change your grade from that given by the autograder either up or down.

Introduction

In this assignment, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.

This assignment includes an autograder for you to check your solutions as you develop them. The autograder can be run with the command:

```
python autograder.py
```

Note: **the autograder and pacman environment use python3. If you default python is not 3 then you can try executing. Note that on teach.cs the default is python2.7, so you must use the command below on teach.cs.**

```
python3 autograder.py
```

See the autograder tutorial in Assignment 0 for more information about using the autograder.

The code for this assignment consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore. You can download all the code and supporting files as a zip archive. In that zip you will find the following files.

Files you'll edit and submit on Markus:

`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 (look but don't modify):

`pacman.py`	The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this assignment.
`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`	Assignment autograder
`testParser.py`	Parses autograder test and solution files
`testClasses.py`	General autograding test classes
`test_cases/`	Directory containing some test cases for each question
`searchTestClasses.py`	Assignment 1 specific autograding test classes

Files to Edit and Submit: You will fill in portions of `search.py` and `searchAgents.py` during the assignment. You may also add other functions and code to these files so as to create a modular implementation. You will submit these files with your modifications. **Do not** put your modifications in other files as those other files will not be uploadable to Markus. **Do not** change the other files in this distribution. Note that all of the code that your implementation depends on must reside inside of one of the two submitted files. You may use standard python imports (**test these on teach.cs!) .**

Evaluation: Your 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. The grade given by the autograder will be a good indication of your actual grade. But we if we find some other issues with your program you actual grade can differ from the autograder grade.

Getting Help: There will be scheduled help sessions (to be announced), the piazza discussion forum will be monitored and questions answered, and you can also ask questions about the assignment during office hours or tutorials. These things are for your support; please take advantage of them. If you can't make our office hours, let us know and we will arrange a different appointment. We want the assignment to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Piazza Discussion: Do not post spoilers! Students posting information on piazza that hinders the learning process of other students will be in violation of class policy.

What to Submit

You will be using MarkUs to submit your assignment. MarkUs accounts for the course will **be set up on Sept 24th**. You will submit two files:

1. Your modified `search.py`
2. Your modified `searchAgents.py`

Note: In the various parts below we ask a number of questions. You do not have to hand in answers to these questions, rather these questions are designed to help you understand what is going on with search.

Welcome to Pacman

After downloading the code (`search.zip`), unzipping it, and changing to the directory created, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

Note: **if python3 is not the default python on the machine you are using you will have to change `python` to the command that invokes python3 (typically by using `python3` rather than plain `python`).**

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.

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

```
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
```

If Pacman gets stuck, you can exit the game by typing CTRL-c into your terminal.

Soon, your agent will solve not only `tinyMaze`, but any maze you want.

Note that `pacman.py` supports a number of options that can each be expressed in a long way (e.g., `--layout`) or a short way (e.g., `-l`). You can see the list of all options and their default values via:

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

Note: if you have a non-graphics connection you can run the `pacman.py` code with the `-t` argument. If you install `python` on your personal machine it should come with the correct graphics support, and the graphics should also work if you are connected to the teaching labs with an X-windows connection.

Question 1 (4 points): Finding a Fixed Food Dot using Depth First Search

In `searchAgents.py`, you'll find a fully implemented `SearchAgent`, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan 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 to help Pacman plan routes! Pseudocode for the search algorithms you'll write can be found in the lecture slides. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets 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: You might find the `Stack`, `Queue` and `PriorityQueue` data structures provided in `util.py` useful. If you create your own data structures for OPEN and have autograder errors check your data structures against these predefined data structures to ensure that you are implementing the same functionality.

Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how OPEN is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. 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`. To ensure that DFS does not run around in circles, implement **path checking** to prune cyclic paths during search. **Do not** use full cycle checking for DFS (we will use full cycle checking for the other searches)

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). Check that the exploration order is what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Hint: If you use a `Stack` as your data structure, the solution found by your DFS algorithm for `mediumMaze` should have a length of 130 (provided you push successors onto the fringe in the order provided by getSuccessors; you might get 246 if you push them in the reverse order). Is this a least cost solution? If not, think about what depth-first search is doing wrong.

Question 2 (3 points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `search.py`.

This time you must implement full cycle checking in your search algorithm to avoid the overhead of cyclic paths. Test your code 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 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
```

Question 3 (3 points): Varying the Cost Function

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 using `-l mediumDottedMaze` and `-l mediumScarymaze` respectively)

By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

Implement the uniform-cost search algorithm with full cycle checking 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).

Question 4 (3 points): A* search

Implement A* search with full cycle checking 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 (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly).

What happens on `openMaze` for the various search strategies? Try the commands

DFS:
`python pacman.py -l openMaze -p SearchAgent`

BFS:
`python pacman.py -l openMaze -p SearchAgent -a fn=bfs`

UCS:
`python pacman.py -l openMaze -p SearchAgent -a fn=ucs`

A*:
`python pacman.py -l openMaze -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic`

You will find that DFS will take too long on `openMaze` maze since it only does path-checking. If you use full cycle checking with DFS it will find a solution (a very unusual one!). You will also find that A* finds the same solution as BFS and UCS but it explores far less of the grid.

Question 5 (3 points): Finding All the Corners

The real power of A* will only be apparent with a more challenging search problem. Now, it's time to formulate a new problem and design a heuristic for it.

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.

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

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 (like the position of ghosts, where extra food is, etc.). In particular, do not use a Pacman `GameState` as a search state. Your code will be very, very slow if you do (and also wrong).

Hint: 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 `breadthFirstSearch` expands just under 2000 search nodes on `mediumCorners`. However, heuristics (used with A* search) can reduce the amount of searching required.

Question 6 (3 points): Corners Problem: Heuristic

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

Implement a non-trivial, admissible 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`.

Non-Trivial Heuristics: The trivial heuristics are the ones that return zero everywhere (UCS) and the heuristic which computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic which reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Grading: Your heuristic must be a non-trivial non-negative admissible heuristic to receive any points. 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:

Number of nodes expanded	Grade
more than 2000	0/3
at most 2000	1/3
at most 1500	2/3
at most 900	3/3

Remember: If your heuristic is inadmissible, you will receive *no* credit, so be careful!

Question 7 (13 points): Eating All The Dots

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. For the present assignment, solutions do not take into account any ghosts or power pellets; solutions only depend on the placement of walls, regular food and Pacman. (Of course ghosts can ruin the execution of a solution! We'll get to that in the next assignment.) 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 (`python pacman.py -l tinySearch -p SearchAgent -a fn=ucs,prob=FoodSearchProblem`) starts to slow down even for the seemingly simple `tinySearch`. As a reference, our implementation finds a path of length 27 after expanding 5168 search nodes.

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

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

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

Our UCS agent searches over 16,000 nodes before it can find an optimal solution.

Any non-trivial non-negative admissible heuristic will receive 1 point. 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 get additional points:

Number of nodes expanded trickySearch	Grade
more than 15000 (and non-trivial)	1/5
at most 15000	2/5
at most 12000	3/5
at most 9000	4/5 (medium)
at most 7000	5/5 (hard)

Also try your agent on the `oneDotFocus` board:

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

On this problem your heuristic must expand no more than 200 nodes to get a mark, and for full marks you must expand no more than 70 nodes.

Number of nodes expanded oneDotFocus	Grade
at most 200 (and non-trivial)	1/4
at most 180	2/4
at most 130	3/4
at most 70	4/4

Finally try your agent on the largeGrid board (note the use of the -z 0.25 zoom factor as this grid is to large to display without shrinking):

```
python pacman.py -l largeGrid -z 0.25 -p AStarFoodSearchAgent
```

On this problem your heuristic must expand no more than 1000 nodes to get a mark, and for full marks you must expand no more than 360 nodes.
Your implementation must also not take more than 5 mins of CPU time when run on teach.cs to solve

largeGrid, if it does you will not get any marks for the largeGrid test.

Number of nodes expanded largeGrid	Grade
Takes > 5mins CPU on teach.cs	0/4
at most 1000 (and non-trivial)	1/4
at most 750	2/4
at most 500	3/4
at most 360	4/4

Remember: If your heuristic is inadmissible, you will receive *no* credit, so be careful! Can you solve `mediumSearch` in a short time? If so, we're either very, very impressed, or your heuristic is inadmissible.

Question 8 (3 points): Suboptimal Search

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard (even the layout `mediumSearch` is probably too hard for A*). In these cases, we'd still like to 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`. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350:

```
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 (its goal should get to any food location). Then, solve that problem with an appropriate search function. The solution should be very short!

Your `ClosestDotSearchAgent` won't always find the shortest possible path through the maze. Make sure you understand why and try to come up with a small example where repeatedly going to the closest dot does not result in finding the shortest path for eating all the dots.

Submission

You're not done yet! You will also need to submit your code, answers to the written questions and your signed acknowledgment form to MarkUs (which will be available on Sept 24th).