Project 1 - Search

- Due Feb 4 by 11:59pm
- Points 100
- Submitting an external tool

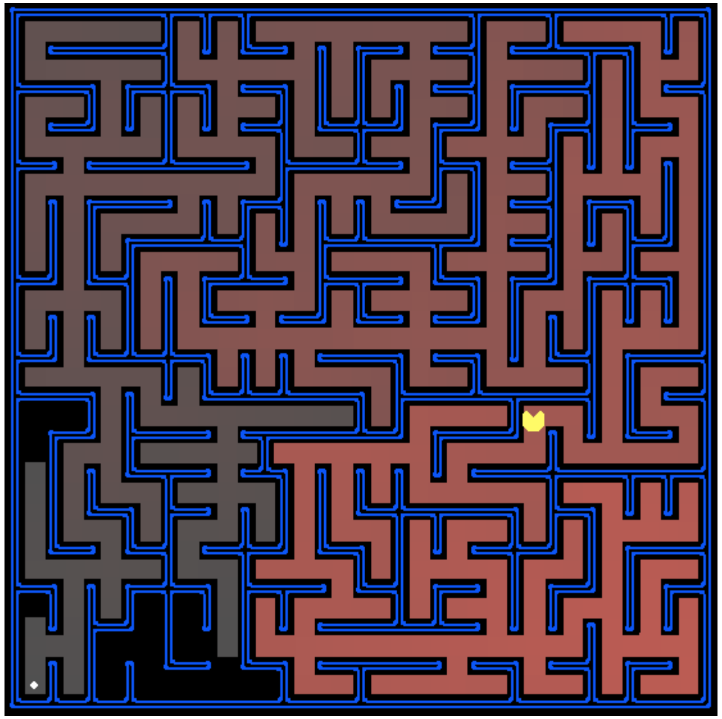

Table of Contents:

Introduction*

In this project, 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.

As in Project 0, this project includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
python autograder.py
```

See the autograder tutorial in Project 0 for more information about using the autograder.
Also, please note you **CANNOT** run this code on ***Windows Powershell*** itself. Please use ***WSL,*** or dual-boot with Ubuntu instead.

The code for this project 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. To complete this assignment download all the code and supporting files as **search.zip. (https://canvas.ou.edu/courses/384471/files/121538820)** **(https://canvas.ou.edu/courses/384471/files/121538820/download?download_frd=1)**

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.
`P1_supplemental.txt`	Answer QS in this text file, along with your team mamber and team number

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 project.
`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`	Project 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`	Project 1 specific autograding test classes

Note that you should answer questions marked as QS in file P1_supplemental.txt (15 points)

Files to Edit: You will fill in portions of search.py and searchAgents.py and **P1_supplemental.txt** during the assignment. Once you have completed the assignment, you will submit your question responses and your Python files (see Submission below). Also, write the answers to the **P1_supplemental.txt** and your team member names (including you) based on the google sheet we will provide you and submit all the files along with other codes mentioned above in Gradescope.

Important: the commenting is a part of grading (around 7-10 percent), make sure you comment your code clearly

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. However, the correctness of your implementation – not the autograder's judgements – will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Academic Dishonesty: 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. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours and discussion in Teams are there for your support; please use them. If you can't make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Welcome to Pacman!

Before running the GUI (below), ensure you have `tkinter` (https://docs.python.org/3/library/tkinter.html) , the Python GUI interface, installed on your machine. For Ubuntu/Debian (including WSL Ubuntu), just run

`sudo apt-get install python3-tk`.

After downloading the code, unzipping it, and changing to the directory, you should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

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
```

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`.

New Syntax

You may not have seen this syntax before:

```
def my_function(a: int, b: Tuple[int, int], c: List[List], d: Any, e: float=1.0):
```

This is annotating the type of the arguments that Python should expect for this function. In the example below, `a` should be an `int` – integer, `b` should be a `tuple` of 2 `int`s, `c` should be a `List` of `Lists` of anything – therefore a 2D array of anything, `d` is essentially the same as if it were not annotated and can be anything, and `e` should be a `float`. `e` is also set to 1.0 if nothing is passed in for it, i.e.:

```
my_function(1, (2, 3), [['a', 'b'], [None, my_class], [[]]], ('h', 1))
```

The above call fits the type annotations, and doesn't pass anything in for e. Type annotations are meant to be an adddition to the docstrings to help you know what the functions are working with. Python itself doesn't enforce these. When writing your own functions, it is up to you if you want to annotate your types; they may be helpful to keep organized or not something you want to spend time on.

Question 1: 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 is provided in the following figure. 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.

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```

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 details of how the fringe 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 make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

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
```

QS1.1: What data structure do you use for implementing DFS? Why?
QS1.2: 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?

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.

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

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

Question 2: Breadth First Search

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

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

Does BFS find a least cost solution (fewest action)? If not, check your implementation.
QS2.1: What data structure do you use for implementing BFS? Why?

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
```

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

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

Question 3: 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 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 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
```

QS3.1: What cost function did you implement? Please briefly describe how your cost function is designed (what factors does it consider and how heavy of a weight has each factor considered been given?)

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).

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

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

Question 4: 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 (about 549 vs. 620 search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly).
QS4.1: What is the difference between the `nullHeuristic` and the Manhattan distance heuristic? Does `nullHeuristic` give the optimal solution? Why is the Manhattan distance heuristic better?
Question 4.2: What happens on `openMaze` for the various search strategies?

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

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

Question 5: 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.

QS5.1: What states representation do you choose in this problem?

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 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.

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.

QS5.2: Briefly explain your implementation

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

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

Question 6: 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, 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
```

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to a nearest goal. More effective heuristics will return values closer to the actual goal costs. To be *admissible*, the heuristic values must be lower bounds on the actual shortest path cost to the nearest goal (and non-negative). To be *consistent*, it must additionally hold that if an action has cost c, then taking that action can only cause a drop in heuristic of at most c.

Remember that admissibility isn't enough to guarantee correctness in graph search – you need the stronger condition of consistency. However, admissible heuristics are usually also consistent, especially if they are derived from problem relaxations. Therefore it is usually easiest to start out by brainstorming admissible heuristics. Once you have an admissible heuristic that works well, you can check whether it is indeed consistent, too. The only way to guarantee consistency is with a proof. However, inconsistency can often be detected by verifying that for each node you expand, its successor nodes are equal or higher in in f-value. Moreover, if UCS and A* ever return paths of different lengths, your heuristic is inconsistent. This stuff is tricky!

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).

QS6.1: What heuristic did you use for this corner problem? Explain the potential strong points and weak points of your chosen heuristic. Is your heuristic consistent? Why?

Grading: Your heuristic must be a non-trivial non-negative consistent 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 according to the following table:

Number of nodes expanded	Grade
more than 2000	0/3
at most 2000	1/3
at most 1600	2/3
at most 1200	3/3

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

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

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

Question 7: 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 project, 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 project.) 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`. As a reference, our implementation takes 2.5 seconds to find a path of length 27 after expanding 5057 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 a *consistent* heuristic for the `FoodSearchProblem`. Try your agent on the `trickySearch` board:

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

QS7.1: What heuristic did you use for this problem? Explain the potential strong points and weak points of your chosen heuristic. Is your heuristic consistent? Why?
Our UCS agent finds the optimal solution in about 13 seconds, exploring over 16,000 nodes.

Any non-trivial non-negative consistent 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	Grade
more than 15000	3/12
at most 15000	6/12
at most 12000	9/12
at most 9000	12/12 (full credit; medium)
at most 7000	15/12 (optional extra credit; hard)

Remember: If your heuristic is inconsistent, 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 inconsistent.

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

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

Question 8: Suboptimal Search

Sometimes, even with A* and a good heuristic, finding the optimal path through all the dots is hard. 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. Then, solve that problem with an appropriate search function. The solution should be very short!

QS8.1: Describe your solution and explain the potential strong points and weak points of your solution.

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.

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

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

Commented Code

We expect that students will put adequate comments and documentation in their code for each problem, as well as following good coding practices in general (no unnecessarily hardcoded values, descriptive variable names, mostly consistant styling, and so on.) Lack of doing so will result in a deduction of points.

Submission

If you are submitting as a group, please reflect this in Gradescope by adding your group members to your submission.

To submit your work, upload your `search.py` and `searchAgent.py` files in Gradescope, along with your answers to the supplemental questions in the provided text file, `P1_answers_supplemental.txt`.

---

*Reference: UC Berkeley AI course

Project 1 Requirements (Details in Gradescope)

Criteria	Ratings		Pts
Autograder Results Submission should pass all tests on Gradescope.	**75 pts** **Full Marks**	**0 pts** **No Marks**	75 pts
Code Quality Submission should be clear, consistent, readable, and well-documented.	**10 pts** **Full Marks**	**0 pts** **No Marks**	10 pts
Supplemental Questions Responses should correctly reflect the course material and the code submissions, and answer the questions fully.	**15 pts** **Full Marks**	**0 pts** **No Marks**	15 pts
			Total Points: 100