Project 1: Search

This project is adapted from: <https://inst.eecs.berkeley.edu/~cs188/fa20/projects/>

[Setup](#_j828co8z61ta)

[Introduction](#_kipar4krw966)

[Welcome to Pacman](#_cw88ytpxrmn7)

[Question 1: Finding a Fixed Food Dot using Depth First Search](#_m5c0ay9urfse)

[Question 2: Breadth First Search](#_s4fxsxfsr8rp)

[Question 3: Varying the Cost Function](#_ssg4nbuzk5oe)

[Turning in your work](#_5z7hytoqrvp2)

## **Setup**

1. Download p1.zip and unzip it.
2. If you are unfamiliar with running Python on your own machine, please walk through [this tutorial](https://inst.eecs.berkeley.edu/~cs188/fa20/project0/).
3. We’re using Python3, preferably version >= 3.6. On some installations, you’ll need to use the command python3 to run version 3 of Python. If so, all of the example commands below should use python3 instead of python.

## **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.

This project includes an autograder for you to grade your answers on your machine. This can be run with the command[[1]](#footnote-0):

python autograder.py

This will run a number of test cases against your code and report what passes/fails. To learn more about the autograder, see the autograder tutorial [here](https://inst.eecs.berkeley.edu/~cs188/fa20/project0/) (scroll down to the “Autograding” heading).

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.

| **The only files you'll edit are:** | |
| --- | --- |
| 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 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 |

**Files to Edit:** You will fill in portions of search.py and searchAgents.py during the assignment. Please *do not* change the other files in this distribution.

**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. 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, section, and the discussion forum 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.

**Discussion:** Please be careful not to post spoilers.

## **Welcome to Pacman**

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.

## **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 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:*** 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

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 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

***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

***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

## **Turning in your work**

1. Run the autograder and save the output in a file called grade.txt.

On a linux/mac you can do this with:

python autograder.py > grade.txt

Otherwise, you can just copy the autograder output to a **plain** text file and save it.

1. Zip the whole p1 directory into a file called p1.zip. On a linux/mac:

zip -r p1.zip p1 (the -r will recursively zip subdirectories).

1. Upload p1.zip to Canvas under P1.

1. Note that there seems to be some issues with TKinter and newer versions of Python (above 3.7) on the newest version of MacOS. If your computer randomly logs you out when you try to run PacMan then please try the tips here (<https://www.python.org/download/mac/tcltk/#built-in-8-6-8>) , or simply downgrade Tkinter using the command: conda install tk=8.6.7 [↑](#footnote-ref-0)