**Machine Learning-Search Learning**

**Learning Objective:** Apply Informed and Uninformed Search Techniques and build the ability to theoretical and practical understanding of Blind and Informed machine search and machine learning techniques.

**Rubric A: Cognitive Domain**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CLO** | **0** | **1** | **2** | **3** | **4** | **5** |
| CLO1 | Student did not write the code. | Student write the code but the code has some errors | The pacman is navigating but not reaching at goal for any single maze | The pacman is navigating and reaching at the goal for atleast one maze | The pacman is navigating and reaching at the goal for atleast two maze | The pacman is navigating and reaching at the goal for all mazes |

**Assignment 03 Search Problems:**

**Objective**: Finding the Goal State for MR. PacMan using DFS Search. Pacman should reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.

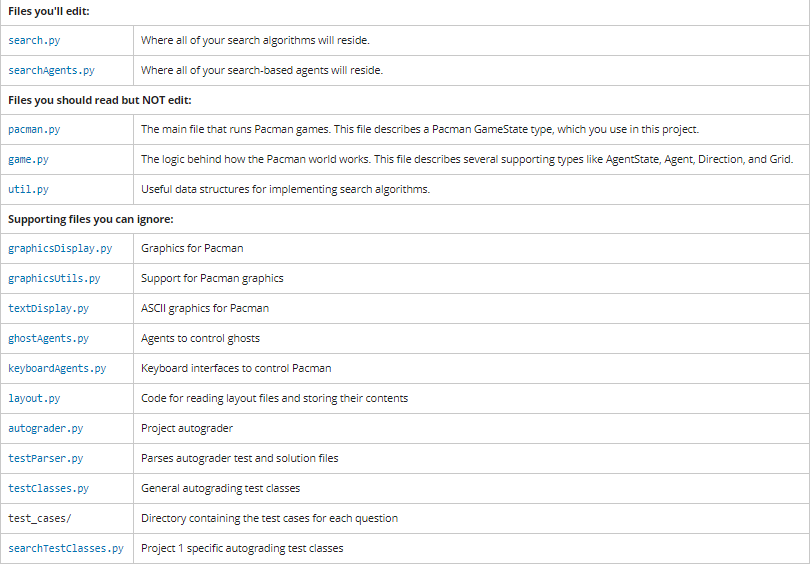
As in Python Refresher, this project includes an autograder for you to grade your answers on your machine. This can be run with the command:

python autograder.py

See the autograder tutorial in Python Refresher for more information about using the autograder.

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.

**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.



**Project Evaluation**

Note: All code should be written, until and unless not mentioned, in the depthFirstSearch (problem) function of Search.py

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! For your reference, the general search algorithm from lecture is shown in Figure 01:

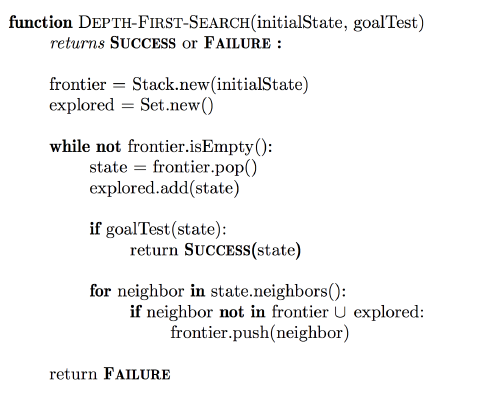


Figure 1: Depth First Search (DFS) Algorithm

**Important note:** 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.

**Challenge 01:** 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 –a fn=depthFirstSearch

python pacman.py -l mediumMaze -p SearchAgent –a fn=depthFirstSearch

python pacman.py -l bigMaze -z .5 -p SearchAgent –a fn=depthFirstSearch

**Challenge 02:**

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, write down what depth-first search is doing wrong.

**Challenge 03:**

Implement the breadth-first search (BFS) algorithm. 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

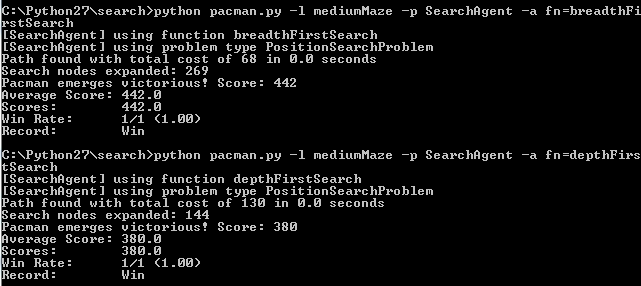
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

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.

**Comparison Between Breadth First Search and Depth First Search.**

1. Run both algorithms one by one and check the output on the command prompt. You shall see different parameters such as total Cost, search node expanded and average score.



**Challenge 04:**

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

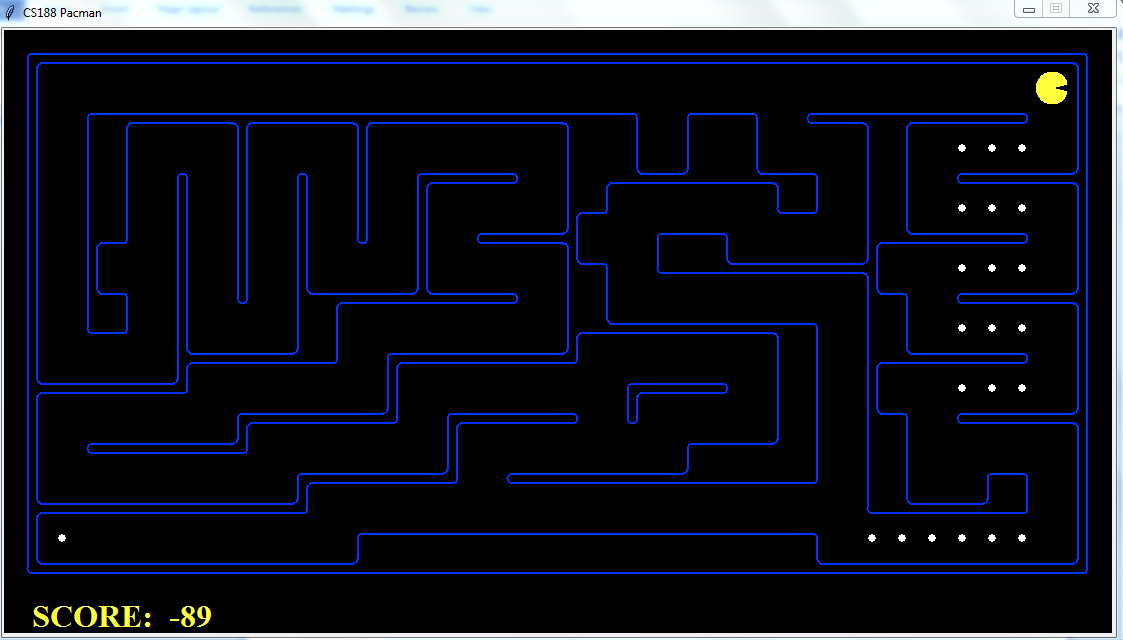
By changing the cost function, we can encourage Pacman to find different paths. For example, we can charge more for dangerous steps in ghost-ridden areas or less for steps in food-rich areas, and a rational Pacman agent should adjust its behavior in response.

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. You can get the cost of each step from the triplet of state.

Hint: Print the state information and you shall see the cost of each state.

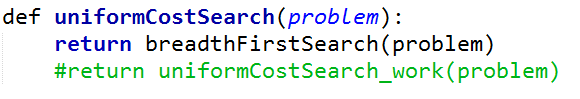
**Step 1:** To understand UCS, the program has built in mediumDottedMaze that maze return the utility according to the available pallet (food) on any state or not. You can view and paly around the maze with following command

**python pacman.py -l mediumDottedMaze**



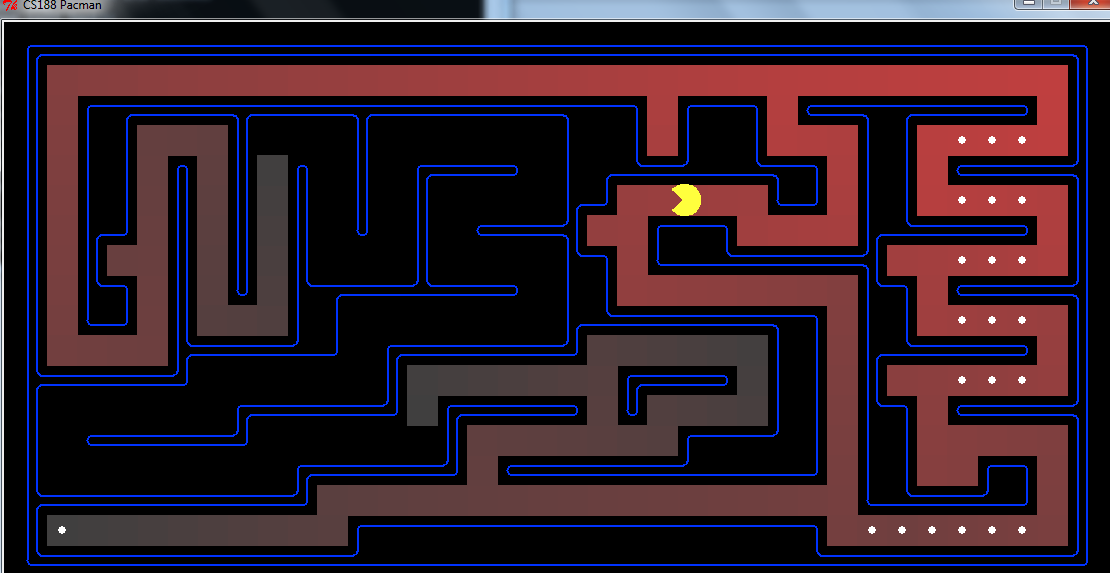
**Step 2:**  To understand how BFS work on the maze, you need to go Search.py file and edit the uniformCostFunction such that it should return the same value as the BFS is retrurning

In other word edit the function **def uniformCostSearch(problem):** and call breadthFirstFunction from this.



**python pacman.py -l mediumDottedMaze -p StayEastSearchAgent**

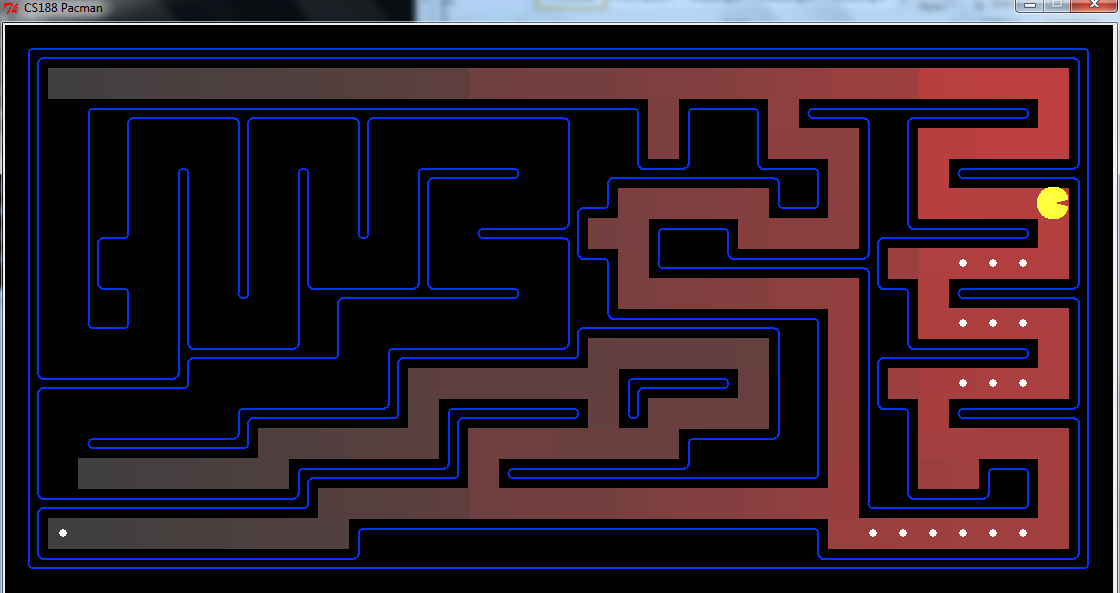
Output of the command:



**Note:** Actually, in this step, we are using the breadth function as uniform cost search to understand that the optimality BFS is to find the nearest path but one agent have some other optimality or utility. For example for this specific Maze the utility of the search Agent is to eat as many as dots and not the shortest path. However, BFS is not considering that utility. Your job is to write down code such that the agent understand it utility function and eat all the food. So the path of your solution should be like of following output.

Note: Now write down the program for UniformCostSearch function and run following command to view the output.

**python pacman.py -l mediumDottedMaze -p StayEastSearchAgent**



**Challenge 05**

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.

**Step1:**

Open the search.py file and copy your UCS code into the aStartSearch algorithm

**Step2:**

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. Write to Manhattan Heuristic function and update the cost calculation formula. If there are two points A (x1,y1) and B(x2,y2) in xy coordinate space the manhattan distance can be calculated as.

**Distance= abs(x2-x1) + abs(y2-y1)**

Hint (you can get goal state information using **problem.goal** )

Define the function in search.py and write the code of manHattanHeuristic in it.

**def manHattanHeuristic(state, problem=None):**

**Step 3:**

Modify aStarSearch function such that it shall consider the cost of heuristic as well cost from the parent node. For heuristic call the method that you have created in step 1

Before running the code you should confirm following abbreviation at the end of the search.py is added. If not then add it

astar = aStarSearch

Run the aStar algorithm using following command

**python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar**

You should see that A\* finds the optimal solution slightly faster than uniform cost search (about less search nodes expanded in our implementation, but ties in priority may make your numbers differ slightly).

**Challenge 06:**

1. Fill following table with the information.

|  |  |  |
| --- | --- | --- |
|  | **A\* Heuristic** | **UCS** |
| **Total Cost** |  |  |
| **Nodes Expanded** |  |  |
| **Score** |  |  |

1. **Why Node Expanded is Greater in UCS and Less in A\* Heuristic?**

1. **Compare other parameters and give reason why they greater/less/equal.**

**Challenge 07: Write a function that calculates heuristic based on the** Euclidean distance. Now run the code on mediumClass maze for UCS, Manhattan and Euclidean distance one by one and report your results. Also discuss the results why these results are coming.

Python pacman.py –l mediumClassicMaze –p SearchAgent –a fn=astar

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Euclidean** | **Manhattan** | **UCS** |
| **Total Cost** |  |  |  |
| **Nodes Expanded** |  |  |  |
| **Score** |  |  |  |