# CS 386: Lab Assignment 7

(TA in charge: Vaibhav Bhosale)

**Acknowledgement:** This lab assignment is based on Project 1: Search in Pacman , which is a part of a recent offering of CS188 at UC Berkeley. The code and resources provided here are almost entirely drawn from the Berkeley project. We thank the authors at Berkeley for making their project available to the public.

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. We use this game as a model to understand how different search algorithms work. 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.

## Code

The base code for this assignment is available in this zip file. The following files are the most relevant ones for you; you will only have to edit `search.py` and `searchAgents.sh`.

| File Name | Description |
|---|---|
| search.py | Where all of your search algorithms will reside. |
| searchAgents.py | Where all of your search-based agents will reside. |
| 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. |

After downloading the code, unzipping it, and changing to the directory, you should be able to play a game of Pacman by running the following command.

```
python pacman.py
```

`pacman.py` supports a number of options (e.g. `--layout` or `-l`). You can see the list of all options and their default values via `python pacman.py -h`.

All the commands you will need in this assignment can be found in the file `commands.txt` for easy copying and pasting.

## Task 0: Understanding the Code Base (Ungraded)

In `searchAgents.py`, you will 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: your task is precisely that of implementing them.

First, test that the `SearchAgent` is working correctly by running the following command.

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

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

Now you will implement different search algorithms to guide pacman to the goal. 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 from the start 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 that are required for compatibility with the autograder.

*Hint:* The algorithms are quite 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.)

### Task 1: Depth First Search (2 marks)

Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`.

Your code should be able to solve these tasks quickly.

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

**Evaluation:** Run the following command to test your solution: `python autograder.py -q q1`. The first 4 test cases are basic test cases. Together they account for 1 mark. If any one of them fails, the fifth test case will not be evaluated. The fifth test case accounts for 1 mark.

### Task 2: Breadth First Search (2 marks)

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

Your code should be able to solve these tasks quickly.

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

**Evaluation:** Run the following command to test your solution: `python autograder.py -q q2`. The first 4 test cases are basic test cases. Together they account for 1 mark. If any one of them fails, the fifth test case will not be evaluated. The fifth test case accounts for 1 mark.

### Task 3: Uniform Cost Search (2 marks)

BFS tries to minimise the number of actions taken, but not necessarily the least-cost path. By varying the cost function, the Pacman can be encouraged to explore different paths. For example, the ghost-ridden dangerous areas can be charged more whereas the food-rich areas be charged lesser.

Implement the uniform-cost search (UCS) algorithm in the `uniformCostSearch` function in `search.py` (the agents and the cost functions are provided to you).

You code should be able to solve these three situations successfully.

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

**Evaluation:** Run the following command to test your solution: `python autograder.py -q q3`. The first 5 test cases are basic test cases. Together they account for 1 mark. If any one of them fails, the next two test cases will not be evaluated. The fifth and sixth test cases will be evaluated for half a mark each.

### Task 4: A* Search (3 marks)

Implement the A* search algorithm in the `aStarSearch` function in `search.py`. A* takes a heuristic function as an argument.

You need to test your A* implementation on the original problem of finding a path through a maze to a fixed position using the Manhattan distance heuristic (already implemented).

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

**Evaluation:** Run the following command to test your solution: `python autograder.py -q q4`. The first 5 test cases are basic test cases. Together they account for 1.5 marks. If any one of them fails, the sixth test case will not be evaluated. The sixth test case will be evaluated for 1.5 marks.

### Task 5: Finding all the Corners (3 marks)

To understand the real power of A* algorithm, let's try it on a difficult problem. In corner mazes, each of the four corners has a dot. Our new search problem is to find the shortest path through the maze that touches all four corners.

Note: For some mazes the shortest path does not always go to the closest food first.

Implement the `CornersProblem` 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
```

**Evaluation:** Run the following command to test your solution: `python autograder.py -q q5`. The code will be tested on two layouts. The tiny layout accounts for 1 mark and the medium layout for 2 marks.

### Task 6: Corners Problem Heuristic (3 marks)

Implement a non-trivial, consistent heuristic for the `CornersProblem` in `cornersHeuristic`. Now your search agent should solve this task.

```
python pacman.py -l mediumCorners -p SearchAgent -a fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic -z 0.5
```

**Evaluation:** Your heuristic must be a non-trivial, consistent heuristic to receive any points. Make sure that your heuristic returns 0 at every goal state and never returns a negative value. You will be graded based on the number of nodes your heuristic expands.

Run the following command to test your solution: `python autograder.py -q q6`. The first 3 test cases are basic test cases. Together they account for 1 mark. If any one of them fails, the fourth test case will not be evaluated. The fourth test case will be evaluated for 2 marks. You will receive 2 marks if the number of nodes expanded does not exceed 1200; 1 mark if the number of nodes expanded is between 1201 and 1600; 0 marks otherwise.

### Submission

You are not done yet! Place all files in which you have written code in or modified in a directory named your roll number (say 12345678). Tar and Gzip the directory to produce a single compressed file (la7-12345678.tar.gz). It must contain the following files:

1. `search.py`
2. `searchAgents.py`

Submit this compressed file on Moodle, under Lab Assignment 7.