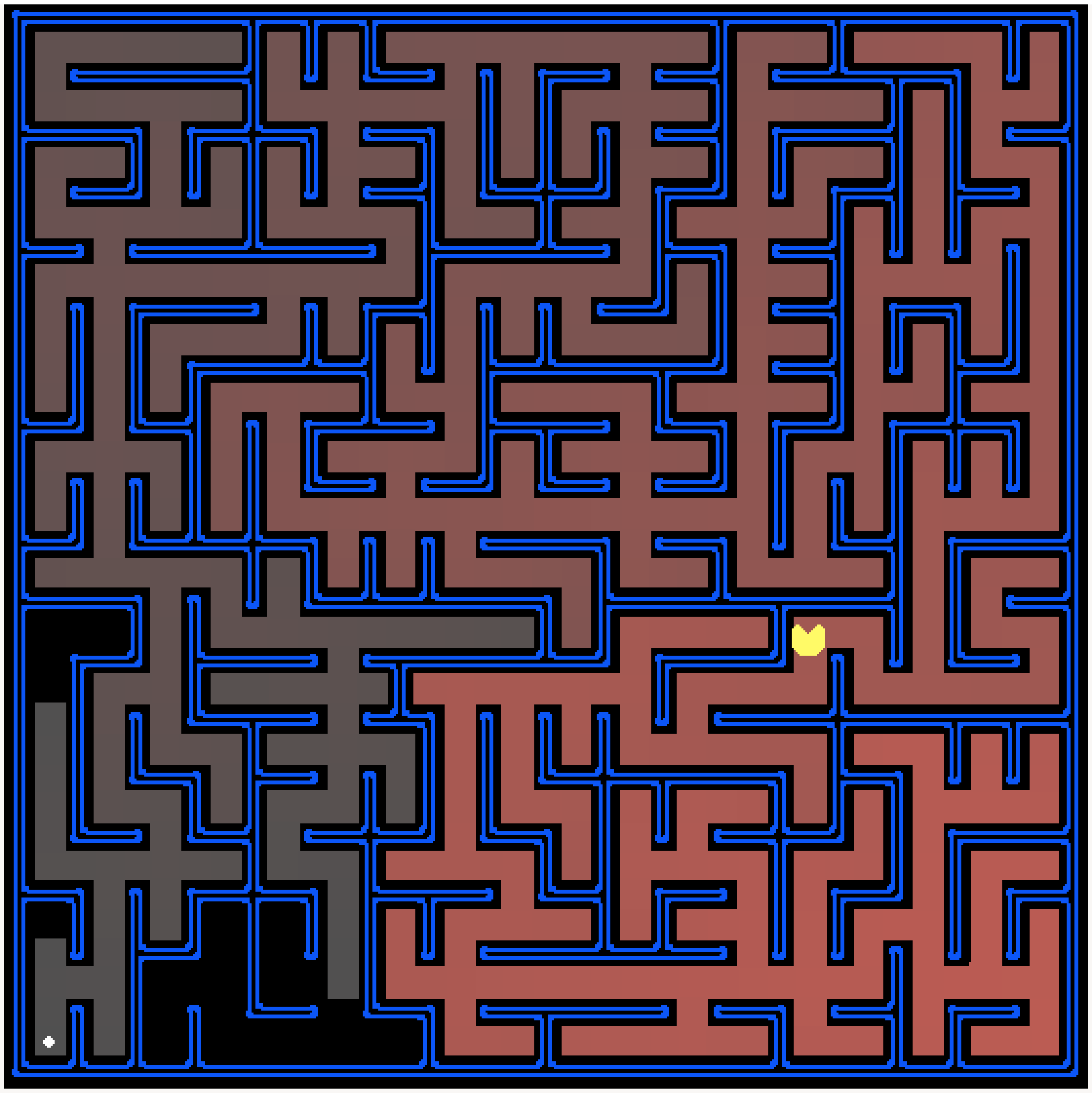
# Project 1: Search

Version 1.00. Last Updated: 12/13/2021

Due: **See Canvas**



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

## 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.

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

| | |
|---|---|
| **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. |
| **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 and Submit:** You will fill in portions of `search.py` and `searchAgents.py` during the assignment. Once you have completed the assignment, you will zip these files as `p1.zip` and submit them in Canvas. Zip only the files—do not zip the directory they are in. Please *do not* change the other files in this distribution or submit any of our original files other than these files.

**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:** Copying someone else's code and submitting it as your own is asking for a grade you did not earn and claiming mastery of skills of which you have not demonstrated mastery. We may or may not use a plagiarism tool on your code in this class.

You are not alone, though we do expect you to know and practice basic problem-solving skills. If you find yourself stuck on something, contact the instructor or a classmate for help. Class time, Office Hours, Discord and Teams are there for your support; please use them. If you need to, set up an appointment for help. These projects should be rewarding and instructional, not frustrating and demoralizing—but I don't know when or how to help unless you ask.

**Discord and Teams:** Please be careful not to post spoilers nor executable code.

---

## Welcome to Pacman

After downloading the code (search.zip), 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. Use the arrow keys to move.

The simplest agent in `searchAgents.py` is called the `GoWestAgent`, which always goes West. A trivial reflex agent:

```
python pacman.py --layout testMaze --pacman GoWestAgent
```

Things get ugly for GoWest 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, you will create an agent that solves `tinyMaze`, and any maze you want.

`pacman.py` supports several options that can be given longhand or shorthand: `--layout` or `-l`. To list all options and their default values:

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

Also, all the commands that appear in this project also appear in `commands.txt`, for easy copying and pasting. In UNIX/Mac OS X, you can run all these commands in order with `bash commands.txt`.

---

## Question 1 (3 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 is 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 your search functions need to return a list of *actions* that will lead the agent from the start to the goal. These actions all must be legal moves (valid directions, no moving through walls).

*Important note:* Make sure to **use** the `Stack`, `Queue` and `PriorityQueue` data structures implementations provided to you in `util.py`! **These data structure implementations have 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. For example, one possible design requires only a single generic search method that is configured with an algorithm-specific queuing strategy. You don't have to implement it that way 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 go to all the explored squares on his way to the goal as expected?

*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`. 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
```

---

## Question 3 (3 points): Varying the Cost Function

BFS will find a fewest-actions path to the goal. We might want to find paths that are "best" in other ways. 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, because they use exponential cost functions. See `searchAgents.py` for details.

---

## Question 4 (3 points): 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 given to you 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. Our implementation yields about 549 vs. 620 search nodes expanded, but ties in priority may make your numbers differ slightly. What happens on `openMaze` for the various search strategies?

---

## 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 has food there or not. For some mazes like `tinyCorners`, the shortest path does not always go to the closest food first. 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 or where extra food is. **Do not use a Pacman `GameState` as a search state. Your code will be very, very slow and wrong if you do.**

*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`. 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, **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:*** 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 non-negative lower bounds on the actual shortest path cost to the nearest goal. 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*.

Admissibility is not 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 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. 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 and enforce a reasonable time limit.

***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:

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

If your heuristic is inconsistent, you will receive 0 credit, so be careful!

---

## Question 7 (4 points): Eating All The Dots

Now we will solve a hard search problem: eating all the Pacman food in as few steps as possible. `FoodSearchProblem` in `searchAgents.py` formally defines the food-clearing problem and is implemented for you. A *solution* is defined to be a path that collects all the food in the Pacman world. For the present project, solutions do not consider 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, but we will 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 should be 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, a recent canonical 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
```

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 | 1/4 |
| at most 15000 | 2/4 |
| at most 12000 | 3/4 |
| at most 9000 | 4/4 (full credit; medium) |
| at most 7000 | 5/4 (optional extra credit; hard) |

*Remember:* If your heuristic is inconsistent, you will receive 0 credit, so be careful! Can you solve `mediumSearch` in a short time? If so, either we're very impressed or your heuristic is inconsistent.

---

## Question 8 (3 points): 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 fast. 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 is 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 1 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!

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

To submit your project, run `python autograder.py` on your solution, then zip the files as instructed above and submit the zip file to the `Project 1` assignment in Canvas.