Project 1 - Search Algorithms

Version: 0.1

Acknowledgements

This project's materials are adapted from the UC Berkeley CS188 Intro to AI course. We thank the course authors for making the materials available to the public.

Outline

This project assumes you use **Python 2.7**.

Project 1 will cover the following:

- Introduction
- Game of Pacman
- Question 1: Depth First Search
- Question 2: Breadth First Search
- Question 3: Uniform Cost Search
- Question 4: A* Search
- Question 5: Corners Problem: Finding All the Corners
- Question 6: Corners Problem: Heuristic
- Python commands for different searches

Files to Edit and Submit:

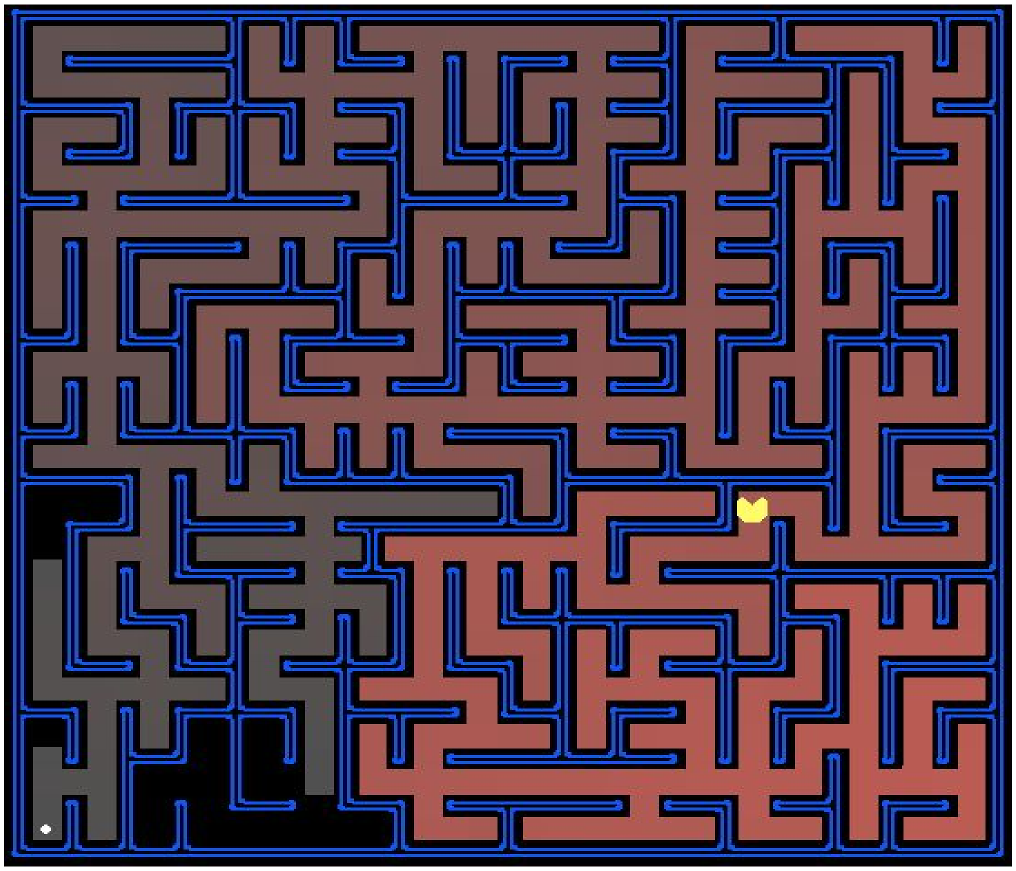
`search.py`	Where all of your search algorithms will reside.
`searchAgents.py`	Where all of your search-based agents will reside.

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

Please check the course website for details.



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.

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:

`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

---

Game of Pacman

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.

Download the code (`search.zip`), unzip it, and change to the directory.

The simplest agent in `searchAgents.py` is called the `GoWestAgent`, which always goes West (a trivial reflex agent).

Soon, your agent will solve not only `tinyMaze`, but any maze you want.

Question 1: Depth First Search - 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. As you work through the following questions, you might find it useful to refer to the object glossary (the second to last tab in the navigation bar above).

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.

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.

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.

Does BFS find a least cost solution? If not, check your implementation.

Question 3: Uniform Cost Search - 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.

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).

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.

Question 5: Corners Problem - 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.

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: 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.

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.

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 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).

Python commands for different searches

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. You can download all the code and supporting files as a zip archive.

You should be able to play a game of Pacman by typing the following at the command line:

```
python pacman.py
```

The goWestAgent can occasionally win as by typing the following command:

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

When turning is required by the agent then the following command has to be tested:

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

If Pacman gets stuck, you can exit the game by typing `CTRL-c` into your terminal.

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.

Note: if you get error messages regarding Tkinter, see this page.

Before testing Depth First search, first test that the `SearchAgent` is working correctly by running:

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

For Depth First Search, 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
```

For Breadth First Search (BFS), 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
```

Hint: If Pacman moves too slowly for you, try the option `--frameTime 0`.

You should now observe successful behavior in all three of the following layouts, where the agents below are all Uniform Cost Search (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
```

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).

Your search agent for Corner’s problem representation 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
```

The implementation of `breadthFirstSearch` expands just under 2000 search nodes on `mediumCorners in Corner’s representation`. However, heuristics (used with A* search) can reduce the amount of search required in Corner’s problem representation.

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.
```

Autograding
As in Project 0, this project also 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.

Submission

You're not done yet! Follow your instructor's guidelines to receive credit on your project!