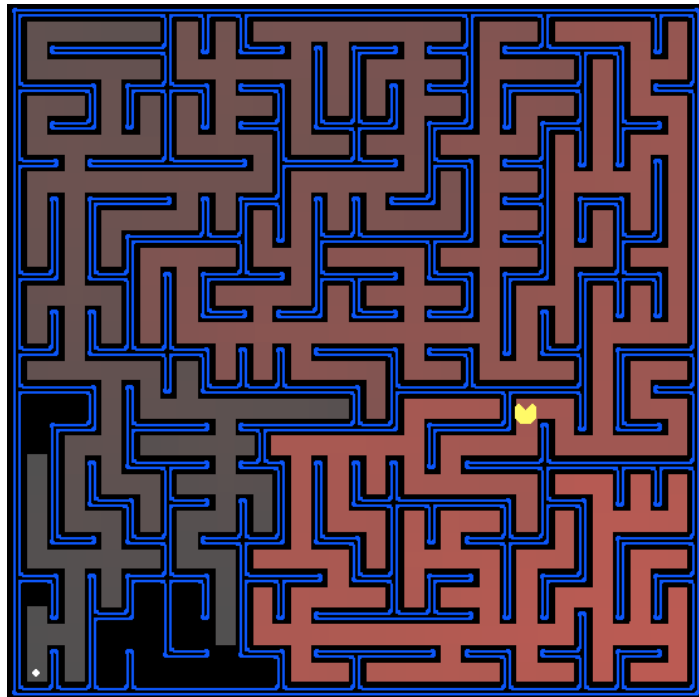


Project 1: Search

This assignment is due Friday, 7/17/09 at 11:59 pm

CS561 Pacman









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

Using X-Win32 to Access Aludra.usc.edu

If you are using Windows OS, you need to install X-Win32 (if you have not done so) to access aludra.usc.edu to complete this project since Python now has to use graphical interface tools provided by UNIX to demonstrate your Pacman.

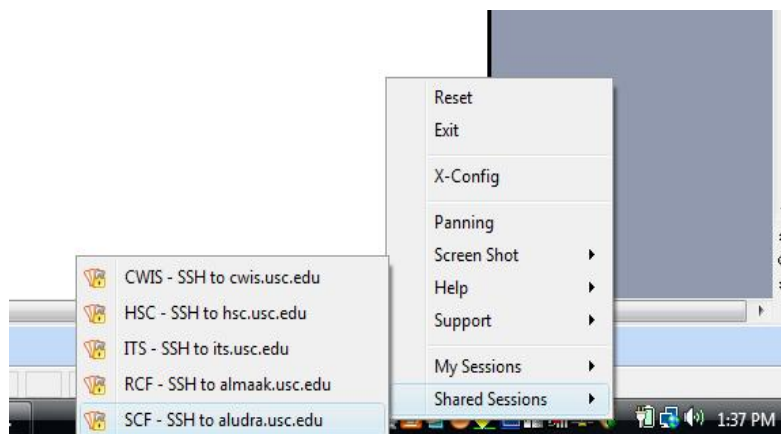
First, go to software.usc.edu to download the current version of X-Win32 as shown in the following figure.

Internet Tools
Applications that help you get the most out of your Internet connection.

Application	Size	Description
 Mozilla Firefox 3.0.7	7.2 MB	Web Browser that has an intuitive interface and blocks viruses, spyware, and popup ads. It delivers Web pages faster than ever.
 Mozilla Thunderbird 2.0.0.19	6.4 MB	Email client that provides safe, fast, and easy email, with intelligent spam filters, quick message search, and customizable views.
 Cisco VPN Client 5.0.02.0090	10.2 MB	Allows you to securely access USC restricted resources from the Wireless network and networks outside of USC.
 PuTTY 0.60	2.2 MB	PuTTY is a free implementation of Telnet and SSH for Win32 and Unix platforms, along with an xterm terminal emulator.
 FileZilla 3.1.2	5.8 MB	A fast and reliable FTP client that supports SFTP (SSH), SSL, and GSS using Kerberos among other things.
 X-Win32 9.4.1072	33.6 MB	PC X-Server with SSH support that allows you to connect your Windows computer to any Solaris, HP/UX, AIX or Linux system.

After you have installed the program successfully, go to Start-> All Programs -> Internet Tools -> X-Win32 to launch it.

Finally, right click the icon of X-Win32 in your Taskbar, choose "Shared Sessions" and invoke "SSH to aludra.usc.edu" as shown in the following figure.



Now, you can start to work on this project.

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.

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 (explore them if you have time, which may make your shine in your job interviews). You can download all the code and supporting files as a [zip archive](#) on den.usc.edu.

Files you'll edit:

[search.py](#) Where all of your search algorithms will reside.

Files you need to look at:

[searchAgents.py](#) Where all of your search-based agents will reside.

[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

What to submit: You will fill in portions of [search.py](#) during this assignment. You should submit this file (only) along with a [codes.txt](#) file to provide us your comments on your own codes so that we can better understand your implementations (you decide what to say in this document, just be clear and concise!).

Please use the following command to submit the answers as a zip archive:
`submit -user csci561b -tag project1 YourName_Project1.tar.gz`

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.

Welcome to Pacman

After downloading the code, unzipping it and changing to the *search* 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
```

[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
```

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

Finding a Fixed Food Dot using Search Algorithms

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 selecting 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 and textbook. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) 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).

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).

Hint: Make sure to check out the [Stack](#), [Queue](#) and [PriorityQueue](#) types provided to you in [util.py](#)!

Question 1 (30%) 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 (textbook section 3.5).

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 244 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 (20%) 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 (textbook section 3.2) without any changes.

```
python eightpuzzle.py
```

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.

Question 3 (20%) Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in [search.py](#). 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).

A* search

Question 4 (30%) Implement A* graph search in the empty function `aStarSearch` in [search.py](#). A* takes a heuristic function as an argument. Heuristics take two argument: 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 (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 (549 vs. 620 search nodes expanded in our implementation). What happens on `openMaze` for the various search strategies?