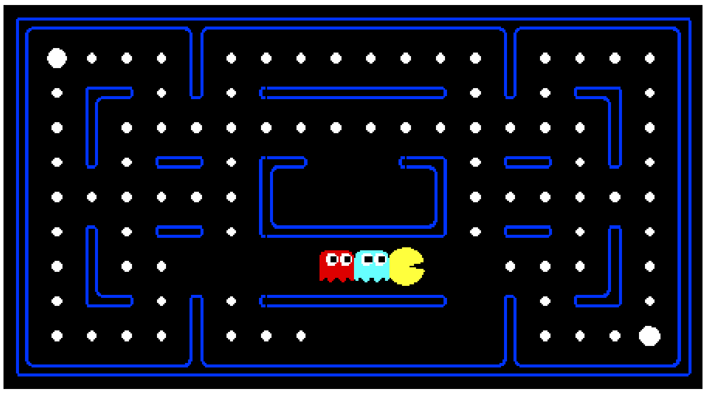
Project 2: Multi-Agent Pacman

This assignment is due Sunday, 8/02/09 at 11:59 pm

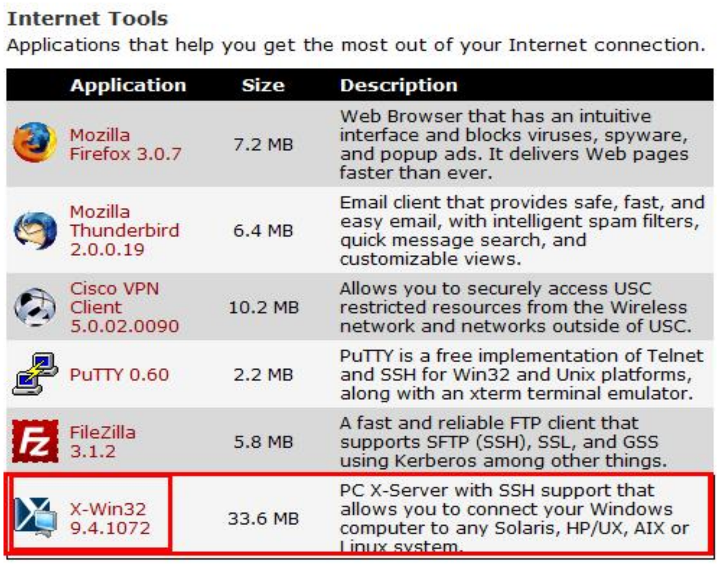
CS561 Pacman


Pacman, now with ghosts.

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.

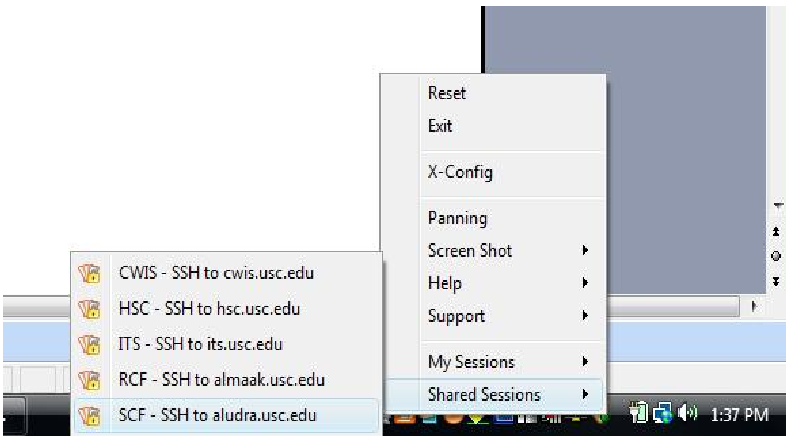


Internet Tools
Applications that help you get the most out of your Internet connection.

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.

Using Python in Windows

1. Download and install the Python Windows binary from the following link:
   http://www.python.org/ftp/python/2.6.2/python-2.6.2.msi

2. Update the Windows "Path" environment variable to include the directory of your Python installation:
   To update the "Path" system variable in Windows XP:

    1. Right-click My Computer, and then click Properties.
    2. Click the Advanced tab.
    3. Click Environment variables.
    4. Select the "Path" variable from the system variables and then click on Edit
    5. Attach a semi-colon followed by the directory of your Python installation to the existing string:

   For example:
   ;C:\Python26
   If your Python is installed in C:\Python26

   To update the "Path" system variable in Windows Vista:

    1. Right-click My Computer, and then click Properties.
    2. Click on Advanced system settings in the left panel.
    3. Click on the Environment variables button.
    4. Select the "Path" variable from the system variables and then click on Edit
    5. Attach a semi-colon followed by the directory of your Python installation to the existing string:

   For example:
   ;C:\Python26
   If your Python is installed in C:\Python26

3. Launch the DOS command line:

   Start >> Run >> cmd

4. Now you can invoke the Python interactive interpreter by typing the command "python" in any Windows directory:

Acknowledgement: Thanks Aqeel Al Sadah for sharing his experience of working Python in Windows with us.

Introduction

In this project, you will design agents for the classic version of Pacman, including ghosts. Along the way, you will implement both minimax and expectimax search and try your hand at evaluation function design.

The code base has not changed much from the previous project, but please start with a fresh installation, rather than intermingling files from project 1. You can, however, use your search.py in any way you want.

The code for this project contains the following files, available as a zip archive.

Key files to read

multiAgents.py	Where all of your multi-agent search agents will reside.
pacman.py	The main file that runs Pacman games. This file also describes a Pacman `GameState` type, which you will use extensively 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.

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 multiAgents.py during the assignment. You should submit this file along with a `codes.txt (or codes.pdf)` 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!). You may also submit supporting files (like search.py, etc.) that you use in your code. Please *do not* change the other files in this distribution or submit any of our original files other than multiAgents.py.

Please use the following command to submit the answers as a zip archive:
submit -user csci561b -tag project2 YourName_Project2.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.

Multi-Agent Pacman

First, play a game of classic Pacman:

```
  python pacman.py
```

Now, run the provided `ReflexAgent` in multiAgents.py:

```
  python pacman.py -p ReflexAgent
```

Note that it plays quite poorly even on simple layouts:

```
  python pacman.py -p ReflexAgent -l testClassic
```

Inspect its code (in multiAgents.py) and make sure you understand what it's doing.

Question 1 (10%) Improve the `ReflexAgent` in multiAgents.py to play respectably. The provided reflex agent code provides some helpful examples of methods that query the `GameState` for information. A capable reflex agent will have to consider both food locations and ghost locations to perform well. Your agent should easily and reliably clear the`testClassic` layout:

```
  python pacman.py -p ReflexAgent -l testClassic
```

Try out your reflex agent on the default `mediumClassic` layout with one ghost or two (and animation off to speed up the display):

```
  python pacman.py --frameTime 0 -p ReflexAgent -k 1
  python pacman.py --frameTime 0 -p ReflexAgent -k 2
```

How does your agent fare? It will likely often die with 2 ghosts on the default board, unless your evaluation function is quite good.

Note: you can never have more ghosts than the layout permits.

Note: As features, try the reciprocal of important values (such as distance to food) rather than the values themselves.

Options: Default ghosts are random; you can also play for fun with slightly smarter directional ghosts using `-g DirectionalGhost`. If the randomness is preventing you from telling whether your agent is improving, you can use `-f`to run with a fixed random seed (same random choices every game). You can also play multiple games in a row with `-n`. Turn off graphics with `-q` to run lots of games quickly.

```
python pacman.py -p ReflexAgent -l openClassic -n 10 -q
```

Don't spend too much time on this question, though, as the meat of the project lies ahead.

Question 2 (40%) Now you will write an adversarial search agent in the provided `MinimaxAgent` class stub in multiAgents.py. Your minimax agent should work with any number of ghosts, so you'll have to write an algorithm that is slightly more general than what appears in the textbook. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

Your code should also expand the game tree to an arbitrary depth. Score the leaves of your minimax tree with the supplied `self.evaluationFunction`, which defaults to `scoreEvaluationFunction`. `MinimaxAgent` extends`MultiAgentAgent`, which gives access to `self.depth` and `self.evaluationFunction`. Make sure your minimax code makes reference to these two variables where appropriate as these variables are populated in response to command line options.

Important: A single search ply is considered to be one pacman move and all the ghosts' responses, so depth 2 search will involve pacman and each ghost moving two times.

Hints and Observations

- The evaluation function in this part is already written (`self.evaluationFunction`). You shouldn't change this function, but recognize that now we're evaluating *states* rather than actions, as we were for the reflex agent. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.
- The minimax values of the initial state in the `minimaxClassic` layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. Note that your minimax agent will often win (15/20 games for us) despite the dire prediction of depth 4 minimax.
  ```
  python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
  ```
- To increase the search depth achievable by your agent, remove the `Directions.STOP` action from Pacman's list of possible actions. Depth 2 should be pretty quick, but depth 3 or 4 will be slow. Don't worry, the next question will speed up the search somewhat.
- Pacman is always agent 0, and the agents move in order of increasing agent index.

- All states in minimax should be `GameStates`, either passed in to `getAction` or generated via`GameState.generateSuccessor`. In this project, you will not be abstracting to simplified states.
- On larger boards such as `openClassic` and `mediumClassic` (the default), you'll find Pacman to be good at not dying, but quite bad at winning. He'll often thrash around without making progress. He might even thrash around right next to a dot without eating it because he doesn't know where he'd go after eating that dot. Don't worry if you see this behavior.
- When Pacman believes that his death is unavoidable, he will try to end the game as soon as possible because of the constant penalty for living. Sometimes, this is the wrong thing to do with random ghosts, but minimax agents always assume the worst:

  ```
  python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
  ```

  Make sure you understand why Pacman rushes the closest ghost in this case.

Question 3 (50%) Make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in `AlphaBetaAgent`. Again, your algorithm will be slightly more general than the pseudo-code in the textbook, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.

You should see a speed-up (perhaps depth 3 alpha-beta will run as fast as depth 2 minimax). Ideally, depth 3 on `smallClassic` should run in just a few seconds per move or faster.

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

The `AlphaBetaAgent` minimax values should be identical to the `MinimaxAgent` minimax values, although the actions it selects can vary because of different tie-breaking behavior. Again, the minimax values of the initial state in the`minimaxClassic` layout are 9, 8, 7 and -492 for depths 1, 2, 3 and 4 respectively.