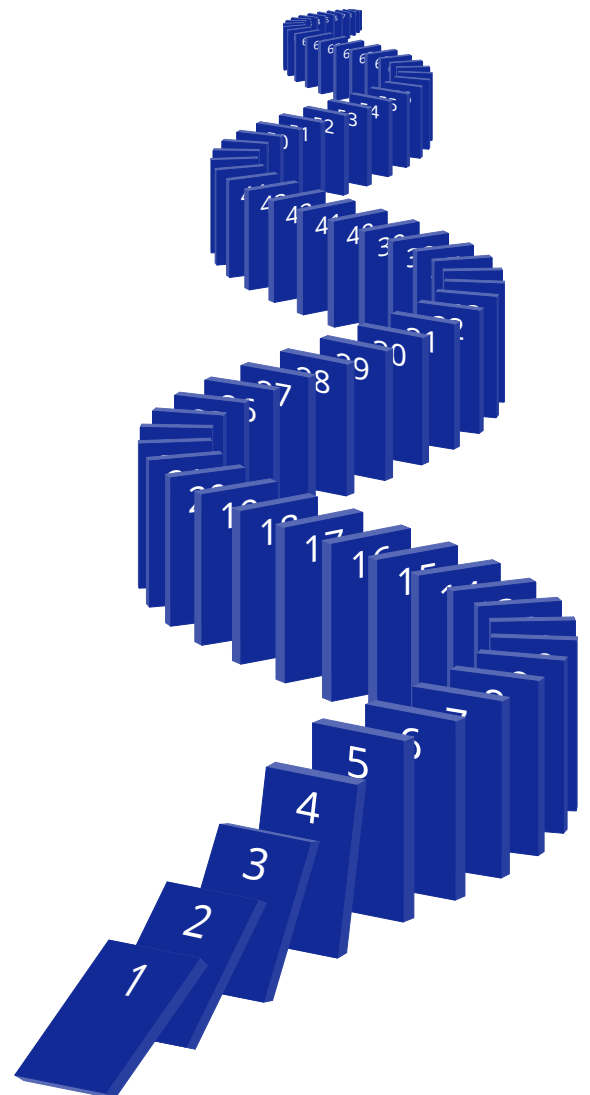


CSE 4712 Taskbook

Some people worry that artificial intelligence will make us feel inferior, but then, anybody in his right mind should have an inferiority complex every time he looks at a flower.

ALAN KAY



Contents

Lab 0	Python, Setup, & Autograder Tutorial	4
1	Setup	4
2	Autograding	5
3	Question 1 (1 point): Addition	8
4	Question 2 (1 point): These Prices are Bananas	9
5	Question 3 (1 point): You Can't Beet These Prices	9
6	Evaluation	10
7	Submission	10
Lab 1	Uninformed Search	11
1	Welcome to Pacman	12
2	Question 1 (3 points): Finding a Fixed Food Dot using Depth First Search	13
3	Question 2 (3 points): Breadth First Search	14
4	Question 3 (3 points): Varying the Cost Function	14
5	Evaluation	15
6	Submission	15
Lab 2	Informed Search	16
1	Welcome (back) to Pacman	17
2	Question 1 (3 points): A* search	18
3	Question 2 (3 points): Finding All the Corners	18
4	Question 3 (3 points): Corners Problem: Heuristic	19
5	Question 4 (4 points): Eating All The Dots	20
6	Question 5 (3 points): Suboptimal Search	21
7	Evaluation	21
8	Submission	21
Lab 3	Constraint Satisfaction Problem	23
1	Welcome to Consistency Based CSP Solver	23
2	Question 1 (4 points): Map Coloring	23
3	Question 2 (5 points): Eating Out	24
4	Question 3 (6 points): Finding Houses	24
5	Question 4 (7 points): Spots	25
6	Question 5 (10 points): Scheduling Tasks	25
7	Evaluation	26
8	Submission	26
Lab 4	Multi-Agent Search	27
1	Welcome to Multi-Agent Pacman	28
2	Question 1 (4 points): Reflex Agent	28
3	Question 2 (5 points): Minimax	29
4	Question 3 (5 points): Alpha-Beta Pruning	31

5	Question 4 (5 points): Expectimax	32
6	Question 5 (6 points): Evaluation Function	33
7	Evaluation	34
8	Submission	34
Lab 5	Markov Decision Process	35
1	MDPs	36
2	Question 1 (4 points): Value Iteration	37
3	Question 2 (1 point): Bridge Crossing Analysis	38
4	Question 3 (5 points): Policies	39
5	Question 4 (1 point): Asynchronous Value Iteration	40
6	Question 5 (1 point): Prioritized Sweeping Value Iteration	41
7	Evaluation	42
8	Submission	42
Lab 6	Reinforcement Learning	44
1	Question 6 (4 points): Q-Learning	45
2	Question 7 (2 points): Epsilon Greedy	46
3	Question 8 (1 point): Bridge Crossing Revisited	47
4	Question 9 (1 point): Q-Learning and Pacman	47
5	Question 10 (3 points): Approximate Q-Learning	49
6	Evaluation	50
7	Submission	50
Lab A	Linux Basics	51
1	File/Directory Manipulation	51
2	IDE Workflow	52
Lab B	Python Basics	54
1	Required Files	54
2	Invoking the Interpreter	54
3	Operators	54
4	Strings	55
5	Built-in Data Structures	57
6	Writing Scripts	62
7	Beware of Indentation!	63
8	Tabs Vs Spaces	64
9	Writing Functions	65
10	Object Basics	65
11	More Python Tips and Tricks	69
12	Troubleshooting	70

Lab 0 Python, Setup, & Autograder Tutorial

The lab tasks for this course assume you use a Linux Distro, e.g. Ubuntu. A brief introduction containing the basic useful commands are discussed in [Appendix A](#). If you use anything else, we believe you are cool enough to find a workaround, if needed.

Lab 0 will cover the following:

- Instructions on how to set up Python,
- Lab grading: Every lab's release includes its autograder that you can run locally to debug. When your tasks will be graded, the same autograder is ran.

Files to Edit and Show: You will fill in portions of `addition.py`, `buyLotsOfFruit.py`, and `shopSmart.py` in `tutorial.zip` during the lab. Please do not change the other files in this distribution. Once you are done, call your course teacher and show them 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. We will review and grade tasks individually to ensure that you receive due credit for your work.

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 show it with minor changes, we will know. We trust you all to show your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. If you can not make our office hours, let us know and we will schedule more. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.

Google Classroom: Please be careful not to post spoilers.

Report Due: 2 weeks after the lab. Please check the submission deadline in the classroom post for more details.

1 Setup

You need a Python 3.6 distribution for this lab. It should be installed by default in the lab PCs. Many of you may not have Python installed on your computers or you might have some other version. To check if you meet our requirements, you should open the terminal and run `python -V` and see that the version is high enough. On some systems that also have Python 2, you may have to use `python3` instead of the aforementioned.

In case you have other versions of Python installed on your computer, Conda is an easy way to manage many different environments, each with its own Python versions and dependencies. This allows us to avoid conflicts between your preferred version and that required for this course. You can install [Anaconda](#) following the instructions from the [link](#). We will walk through how to set up and use a conda environment. To get a refresher on Python, check out [Appendix B](#).

1.1 Creating a Conda Environment

The command for creating a conda environment with Python 3.6 is:

```
conda create --name <env-name> python=3.6
```

For us, we decide to name our environment ailaab, so we run the following command in Terminal and then follow the instructions to install any missing packages.

```
user@linux ~$ conda create --name ailaab python=3.6
```

1.2 Entering the Environment

To enter the conda environment that we just created, do the following. Note that the Python version within the environment is 3.6, just what we want.

```
user@linux ~$ conda activate ailaab
(ailab) [user@linux ~/python_basics]$ python -V
Python 3.6.13 :: Anaconda, Inc.
```

Note that, the tag (<env-name>) shows you the name of the conda environment that is active.

1.3 Leaving the Environment

Leaving the environment is just as easy.

```
(ailab) user@linux ~$ conda deactivate
```

If you check the Python version now, you will see that the version has now returned to whatever the system default is!

2 Autograding

To get you familiarized with the autograder, we will ask you to code, test, and submit solutions for three questions.

You can download all of the files associated with the autograder tutorial as a zip archive: `tutorial.zip` (note this is different from the zip file used in the UNIX and Python tutorials, `python_basics.zip`). If not done so already, download the file, unzip it, and examine its contents:

```
(ailab) user@linux ~$ unzip tutorial.zip
(ailab) user@linux ~$ cd tutorial
(ailab) user@linux ~/tutorial$ ls
addition.py      shop.py          textDisplay.py
autograder.py    shopSmart.py     town.py
buyLotsOfFruit.py submission_autograder.py tutorialTestClasses.py
grading.py        test_cases       util.py
projectParams.py testClasses.py
shopAroundTown.py testParser.py
```

This contains a number of files you will edit or run:

- `addition.py`: source file for question 1
- `buyLotsOfFruit.py`: source file for question 2
- `shopSmart.py`: source file for question 3
- `shop.py`: helper file for question 3
- `autograder.py`: autograding script (see below)

You can ignore the other files.

The command `python autograder.py` grades your solution to all three problems. If we run it before editing any files we get a page or two of output:

```
(ailab) user@linux ~\tutorial$ python autograder.py
Starting on 5-29 at 12:55:17

Question q1
=====

*** FAIL: test_cases/q1/addition1.test
***      add(a,b) must return the sum of a and b
***      student result: "0"
***      correct result: "2"
*** FAIL: test_cases/q1/addition2.test
***      add(a,b) must return the sum of a and b
***      student result: "0"
***      correct result: "5"
*** FAIL: test_cases/q1/addition3.test
***      add(a,b) must return the sum of a and b
***      student result: "0"
***      correct result: "7.9"
*** Tests failed.

### Question q1: 0/1 ###

Question q2
=====

*** FAIL: test_cases/q2/food_price1.test
***      buyLotsOfFruit must compute the correct cost of the order
***      student result: "0.0"
***      correct result: "None"
*** FAIL: test_cases/q2/food_price2.test
***      buyLotsOfFruit must compute the correct cost of the order
***      student result: "0.0"
```

```
***          correct result: "14.75"
*** FAIL: test_cases/q2/food_price3.test
***          buyLotsOfFruit must compute the correct cost of the order
***          student result: "0.0"
***          correct result: "6.4375"
*** Tests failed.
```

Question q2: 0/1

Question q3

=====

```
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
*** FAIL: test_cases/q3/select_shop1.test
***          shopSmart(order, shops) must select the cheapest shop
***          student result: "None"
***          correct result: "<FruitShop: shop1>"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
*** FAIL: test_cases/q3/select_shop2.test
***          shopSmart(order, shops) must select the cheapest shop
***          student result: "None"
***          correct result: "<FruitShop: shop2>"
Welcome to shop1 fruit shop
Welcome to shop2 fruit shop
Welcome to shop3 fruit shop
*** FAIL: test_cases/q3/select_shop3.test
***          shopSmart(order, shops) must select the cheapest shop
***          student result: "None"
***          correct result: "<FruitShop: shop3>"
*** Tests failed.
```

Question q3: 0/1

Finished at 12:20:03

Provisional grades

=====

Question q1: 0/1

Question q2: 0/1

Question q3: 0/1

Total: 0/3

Your grades are NOT yet registered. To register your grades, make sure to follow your instructor's guidelines to receive credit on this lab.

For each of the three questions, this shows the results of that question's tests, the questions grade, and a final summary at the end. Because you have not yet solved the questions, all the tests fail. As you solve each question you may find some tests pass while other fail. When all tests pass for a question, you get full marks.

Looking at the results for question 1, you can see that it has failed three tests with the error message "add(a,b) must return the sum of a and b". The answer your code gives is always 0, but the correct answer is different. We will fix that in the next section.

3 Question 1 (1 point): Addition

Open addition.py and look at the definition of add:

```
1 def add(a, b):
2     "Return the sum of a and b"
3     "*** YOUR CODE HERE ***"
4     return 0
```

The tests called this with a and b set to different values, but the code always returned zero. Modify this definition to read:

```
1 def add(a, b):
2     "Return the sum of a and b"
3     print("Passed a=%s and b=%s, returning a+b=%s" % (a, b, a+b))
4     return a+b
```

Now rerun the autograder. You will see something like this (omitting the results for questions 2 and 3):

```
(ailab) user@linux ~/tutorial$ python autograder.py -q q1
Starting on 1-21 at 23:52:05

Question q1
=====
Passed a=1 and b=1, returning a+b=2
*** PASS: test_cases/q1/addition1.test
***         add(a,b) returns the sum of a and b
Passed a=2 and b=3, returning a+b=5
*** PASS: test_cases/q1/addition2.test
***         add(a,b) returns the sum of a and b
Passed a=10 and b=-2.1, returning a+b=7.9
*** PASS: test_cases/q1/addition3.test
***         add(a,b) returns the sum of a and b
```



```
### Question q1: 1/1 ###
```

```
Finished at 23:41:01
```

```
Provisional grades
```

```
=====
```

```
Question q1: 1/1
```

```
Question q2: 0/1
```

```
Question q3: 0/1
```

```
-----
```

```
Total: 1/3
```

You now pass all tests, getting full marks for question 1. Notice the new lines “Passed a=...” which appear before “*** PASS: ...”. These are produced by the print statement in add. You can use print statements like that to output information useful for debugging.

4 Question 2 (1 point): These Prices are Bananas

Implement the `buyLotsOfFruit(orderList)` function in `buyLotsOfFruit.py` which takes a list of (fruit, pound) tuples and returns the cost of your list. If there is some fruit in the list which does not appear in `fruitPrices` it should print an error message and return `None`. Please do not change the `fruitPrices` variable.

Run `python autograder.py` until question 2 passes all tests and you get full marks. Each test will confirm that `buyLotsOfFruit(orderList)` returns the correct answer given various possible inputs. For example, `test_cases/q2/food_price1.test` tests whether:

```
Cost of [('apples', 2.0), ('pears', 3.0), ('limes', 4.0)] is 12.25
```

5 Question 3 (1 point): You Can't Beet These Prices

Fill in the function `shopSmart(orders, shops)` in `shopSmart.py`, which takes an `orderList` (like the kind passed in to `FruitShop.getPriceOfOrder`) and a list of `FruitShop` and returns the `FruitShop` where your order costs the least amount in total. Do not change the file name or variable names, please. Note that we will provide the `shop.py` implementation as a “support” file, so you do not need to submit yours.

Run `python autograder.py` until question 3 passes all tests and you get full marks. Each test will confirm that `shopSmart(orders, shops)` returns the correct answer given various possible inputs. For example, with the following variable definitions:

```
1 orders1 = [('apples', 1.0), ('oranges', 3.0)]
2 orders2 = [('apples', 3.0)]
3 dir1 = {'apples': 2.0, 'oranges': 1.0}
4 shop1 = shop.FruitShop('shop1', dir1)
5 dir2 = {'apples': 1.0, 'oranges': 5.0}
6 shop2 = shop.FruitShop('shop2', dir2)
7 shops = [shop1, shop2]
```

test_cases/q3/select_shop1.test tests whether:

```
shopSmart.shopSmart(orders1, shops) == shop1
```

and test_cases/q3/select_shop2.test tests whether:

```
shopSmart.shopSmart(orders2, shops) == shop2
```

6 Evaluation

Once you are done with the tasks, run autograder for all the tasks via:

```
(ailab) user@linux ~/tutorial$ python autograder.py
```

Then call your course teacher and show them the autograder results and codes.

7 Submission

Submit one file: StudentID_L0.pdf (StudentID will be replaced by your student ID) under **Lab 0** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce in the algorithms/code), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, hypothesize on why such phenomenon occurred, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.

You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 1 Uninformed Search

In this lab, 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 Lab 0, this lab includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
(ailab) user@linux ~/usearch$ python autograder.py
```

See the autograder tutorial in Lab 0 for more information about using the autograder.

The code for this lab 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 from Google Classroom as `usearch.zip`.

Files you will edit:	
search.py	Where all of your search algorithms will reside.
Files you might want to look at:	
searchAgents.py	Where all of your search-based agents reside.
pacman.py	The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this lab.
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	Lab 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	Lab 1 specific autograding test classes

Files to Edit and Show: You will fill in portions of `search.py` during the task. Please *do not* change the other files in this distribution. Once you are done, call your course teacher and show them this file.

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 judgments – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.

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 do not try. We trust you all to submit your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.

Google Classroom: Please be careful not to post spoilers.

Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 Welcome to Pacman

After downloading the code (`usearch.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:

```
(ailab) user@linux ~/usearch$ 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). The agent can occasionally win:

```
(ailab) user@linux ~/usearch$ python pacman.py --layout testMaze --pacman GoWestAgent
```

But, things get ugly for this agent when turning is required:

```
(ailab) user@linux ~/usearch$ python pacman.py --layout tinyMaze --pacman GoWestAgent
```

If Pacman gets stuck, you can exit the game by typing `CTRL-c` into your terminal. Soon, your agent will solve not only `tinyMaze`, but any maze you want.

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:

```
(ailab) user@linux ~/usearch$ python pacman.py -h
```

Also, all of the commands that appear in this lab also appear in `commands.txt`, for easy copying and pasting. In Linux, you can even run all these commands in order with `bash commands.txt`.

2 Question 1 (3 points): Finding a Fixed Food Dot using Depth First Search

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 — that is your job.

First, test that the `SearchAgent` is working correctly by running:

```
(ailab) user@linux ~/usearch$ python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMaz
```

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! The pseudocode for the search algorithms you will 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.

Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`. The function takes one parameter, `problem`, which provides you with a few important functions, such as `getStartState()` to initialize your algorithm, `isGoalState()` that takes the current state to check whether it is a goal state or not, and `getSuccessors()` that gives you a set of successor *elements* for a given state. Each *element* consists of 3 items: next state (the next position of Pacman), action (the action required to get to the next state from the current state), and cost (the cost of executing the action). For DFS and BFS, where our goal is to find a path to food, we can assume the cost as 1. 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:

```
(ailab) user@linux ~/usearch$ python pacman.py -l tinyMaze -p SearchAgent
(ailab) user@linux ~/usearch$ python pacman.py -l mediumMaze -p SearchAgent
(ailab) user@linux ~/usearch$ 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?

Important notes:

- ▷ 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).
- ▷ 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.
- ▷ 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).
- ▷ 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.

Grading: Run the following command to see if your implementation passes all the autograder test cases:

```
(ailab) user@linux ~/usearch$ python autograder.py -q q1
```

3 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 the depth-first search.

```
(ailab) user@linux ~/usearch$ python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
(ailab) user@linux ~/usearch$ python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z
```

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

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

If you have written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

```
(ailab) user@linux ~/usearch$ python eightpuzzle.py
```

Grading: Run the following command to see if your implementation passes all the autograder test cases:

```
(ailab) user@linux ~/usearch$ python autograder.py -q q2
```

4 Question 3 (3 points): 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. 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):

```
(ailab) user@linux ~/usearch$ python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
(ailab) user@linux ~/usearch$ python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
(ailab) user@linux ~/usearch$ python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

Important 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).

Grading: Run the following command to see if your implementation passes all the autograder test cases:

```
(ailab) user@linux ~/usearch$ python autograder.py -q q3
```

5 Evaluation

Once you are done with the tasks, run autograder for all the tasks via:

```
(ailab) user@linux ~/usearch$ python autograder.py
```

Then call your course teacher and show them the autograder results and codes.

6 Submission

Submit one file: StudentID_L1 . pdf (StudentID will be replaced by your student ID) under **Lab 1** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce in the algorithms/code), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, hypothesize on why such phenomenon occurred, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.

You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 2 Informed Search

In this lab, 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 Lab 0, this lab includes an autograder for you to grade your answers on your machine. This can be run with the command:

```
(ailab) user@linux ~/isearch$ python autograder.py
```

See the autograder tutorial in Lab 0 for more information about using the autograder.

The code for this lab contains the following files, available in Google Classroom as `isearch.zip`.

Files you will 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 lab.
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.
commands.txt	Contains the commands mentioned in the PDF but in a single line
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	Lab 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	Task 1 specific autograding test classes

Files to Edit and Show: You will fill in portions of `search.py` and `searchAgents.py` during the task. Please *do not* change the other files in this distribution. Once you are done, call your course teacher and show them 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 judgments – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.

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 do not try. We trust you all to submit your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.

Google Classroom: Please be careful not to post spoilers.

Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 Welcome (back) to Pacman

After downloading the code (`isearch.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:

```
(ailab) user@linux ~/isearch$ 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). The agent can occasionally win:

```
(ailab) user@linux ~/isearch$ python pacman.py --layout testMaze --pacman GoWestAgent
```

But, things get ugly for this agent when turning is required:

```
(ailab) user@linux ~/isearch$ python pacman.py --layout tinyMaze --pacman GoWestAgent
```

If Pacman gets stuck, you can exit the game by typing `CTRL-c` into your terminal. Soon, your agent will solve not only `tinyMaze`, but any maze you want.

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:

```
(ailab) user@linux ~/isearch$ python pacman.py -h
```

All of the commands that appear in this lab also appear in `commands.txt`, for easy copying and pasting. In Linux, you can even run all these commands in order with `bash commands.txt`.

2 Question 1 (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 (implemented already as `manhattanHeuristic` in `searchAgents.py`).

```
(ailab) user@linux ~/isearch$ 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). What happens on `openMaze` for the various search strategies?

Grading: Run the following command to see if your implementation passes all the autograder test cases:

```
(ailab) user@linux ~/isearch$ python autograder.py -q q1
```

3 Question 2 (3 points): Finding All the Corners

Note: Question 2 requires implementation of `breadthFirstSearch` function from `search.py`, which has been already done for you.

The real power of A* will only be apparent with a more challenging search problem. Now, it is 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). Note that for some mazes like `tinyCorners`, the shortest path does not always go to the closest food first!

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:

```
(ailab) user@linux ~/isearch$ python pacman.py -l tinyCorners -p SearchAgent \
-a fn=bfs,prob=CornersProblem
(ailab) user@linux ~/isearch$ 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, 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).

Important Notes:

- An instance of the `CornersProblem` class represents an entire search problem, not a particular state. Particular states are returned by the functions you write, and your functions return a data structure of your choosing (e.g. tuple, set, etc.) that represents a state.

- While a program is running, remember that many states simultaneously exist, all on the queue of the search algorithm, and they should be independent of each other. In other words, you should not have only one state for the entire `CornersProblem` object; your class should be able to generate many different states to provide to the search algorithm.
- The shortest path through `tinyCorners` takes 28 steps.
- 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.
- When coding up `getSuccessors`, make sure to add children to your successors list with a cost of 1.

Grading: Run the following command to see if your implementation passes all the autograder test cases:

```
(ailab) user@linux ~/isearch$ python autograder.py -q q2
```

4 Question 3 (3 points): Corners Problem: Heuristic

Note: Make sure to complete Question 1 before working on Question 3, because Question 3 builds upon your answer for Question 1.

Implement a non-trivial, consistent heuristic for the `CornersProblem` in `cornersHeuristic`.

```
(ailab) user@linux ~/isearch$ python pacman.py -l mediumCorners \
-p AStarCornersAgent -z 0.5
```

Here, `AStarCornersAgent` is a shortcut for

```
SearchAgent -a fn=aStarSearch, prob=CornersProblem, heuristic=cornersHeuristic
```

Admissibility vs. Consistency: Remember, heuristics are just functions that take search states and return numbers that estimate the cost to the 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 is not enough to guarantee correctness in graph search – you need a 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 that computes the true completion cost. The former won't save you any time, while the latter will timeout the autograder. You want a heuristic that reduces total compute time, though for this assignment the autograder will only check node counts (aside from enforcing a reasonable time limit).

Grading: Run the following command to see if your implementation passes all the autograder test cases:

```
(ailab) user@linux ~/isearch$ python autograder.py -q q3
```

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 will be graded:

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

Remember if your heuristic is inconsistent, you will receive *no* credit, so be careful!

5 Question 4 (4 points): Eating All The Dots

Note: Make sure to complete Question 1 before working on Question 4, because Question 4 builds upon your answer for Question 1.

Now we will solve a hard search problem: eating all the Pacman food in as few steps as possible. For this, we will need a new search problem definition that formalizes the food-clearing problem: `FoodSearchProblem` in `searchAgents.py` (implemented for you). A solution is defined to be a path that collects all of the food in the Pacman world. For this lab, solutions do not take into account 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! We will get to that in the next lab task in shaa Allah.) 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 of 7).

```
(ailab) user@linux ~/isearch$ python pacman.py -l testSearch -p AStarFoodSearchAgent
```

Note that, `AStarFoodSearchAgent` is a shortcut for

```
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, our implementation takes 2.1 seconds to find a path of length 27 after expanding 5057 search nodes. Fill in `foodHeuristic` in `searchAgents.py` with a *consistent* heuristic for the `FoodSearchProblem`. Try your agent on the `trickySearch` board:

```
(ailab) user@linux ~/isearch$ python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

Our UCS agent finds the optimal solution in about 16.9 seconds, exploring over 16,000 nodes.

Grading: Run the following command to see if your implementation passes all the autograder test cases:

```
(ailab) user@linux ~/isearch$ python autograder.py -q q4
```

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 will 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 that if your heuristic is inconsistent, you will receive no credit, so be careful! Can you solve `mediumSearch` in a short time? If so, we are either very, very impressed, or your heuristic is inconsistent.

6 Question 5 (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 would still like to find a reasonably good path, quickly. In this section, you will 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`. Notice that, the function calls `AnyFoodSearchProblem` class, which is also missing a goal test function. You need to implement that as well. Our agent solves this maze (suboptimally!) in under a second with a path cost of 350:

```
(ailab) user@linux ~/isearch$ python pacman.py -l bigSearch \
    -p ClosestDotSearchAgent -z .5
```

Your `ClosestDotSearchAgent` will not 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.

Grading: Run the following command to see if your implementation passes all the autograder test cases:

```
(ailab) user@linux ~/isearch$ python autograder.py -q q5
```

7 Evaluation

Once you are done with the tasks, run autograder for all the tasks via:

```
(ailab) user@linux ~/isearch$ python autograder.py
```

Then call your course teacher and show them the autograder results and codes.

8 Submission

Submit one file: `StudentID_L2.pdf` (StudentID will be replaced by your student ID) under **Lab 2** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce in the

algorithms/code), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, hypothesize on why such phenomenon occurred, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.

You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 3 Constraint Satisfaction Problem

In this lab, you will create simple CSPs and solve them using the amazing [applet](#) provided by [AIspace](#). The code for this lab requires running a Java applet. So you need to have Java installed on your computer. To check if Java is installed on your computer, run the following command in the terminal:

```
(base) user@linux ~/csp$ java -version
```

The output should display the version of the Java package installed on your system. If it does not, you need to install [Java Runtime Environment \(JRE\)](#).

Files to Edit and Show: You will save your formulations as XML files corresponding to the solutions of the tasks provided. Once you are done, call your course teacher and show them these files.

Evaluation: Your implementations will be inspected manually.

Academic Dishonesty: We will be checking your implementation against other submissions in the class for logical redundancy. If you copy someone else's files and submit it with minor changes, we will know. We trust you all to show your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Google Classroom: Please be careful not to post spoilers.

Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 Welcome to Consistency Based CSP Solver

After downloading the applet (`constraint.jar`) and navigating to the appropriate directory, you can open it using the following command:

```
(base) user@linux ~/csp$ java -jar constraint.jar
```

After opening it, you will be able to load sample CSPs by going to File -> Load Sample CSP and choosing the one that you want to load. There are two tabs in the applet, namely Create and Solve. In the Create tab, you can create new CSPs or edit an existing one. It allows you to create variables, create constraints, connect variables to constraints, select and move the placed objects, delete objects, and set properties. In the Solve tab, you can assign values for variables, apply arc consistency, apply backtracking to solve problems, etc.

Play around with the applet to get an idea about how it works. Then formulate and solve the following tasks using it:

2 Question 1 (4 points): Map Coloring

As shown in [Figure 3.1](#), Sulawesi, an Indonesian island, is divided into six provinces: Gorontalo (*G*), West Sulawesi (*WS*), South Sulawesi (*SS*), Central Sulawesi (*CS*), Southeast Sulawesi (*SES*), and North Sulawesi (*NS*).

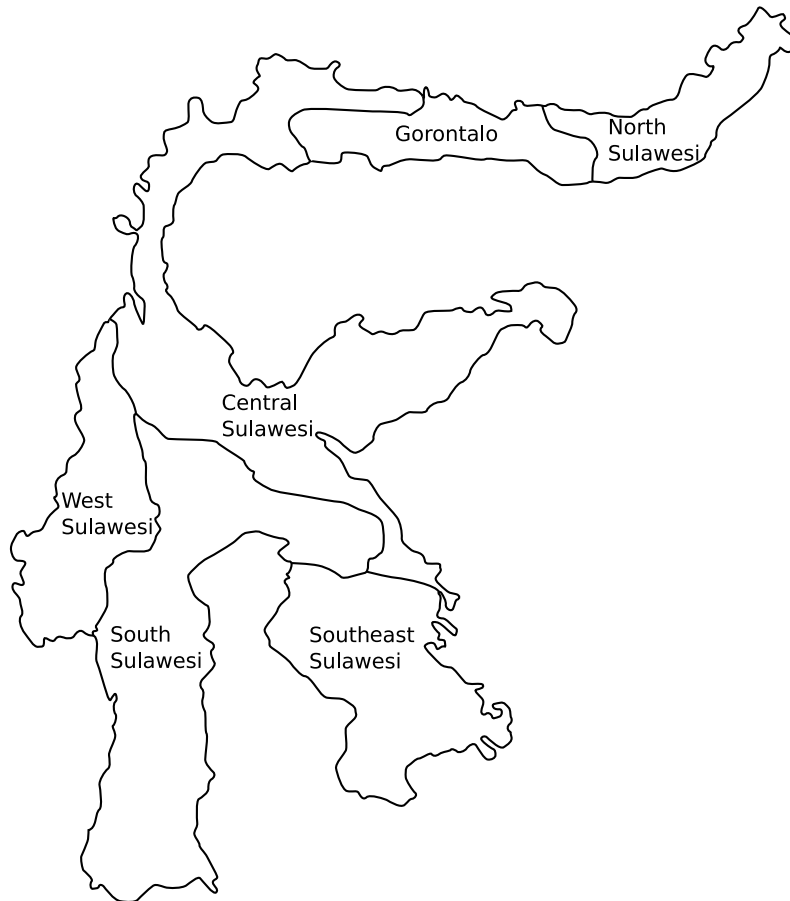


Figure 3.1. A Simplified Map of Sulawesi

We want to color each province either Red (R), Yellow (Y), or Blue (B) in such a way that no two neighboring regions (provinces that share their borders) have the same color. Formulate the problem as CSP and explore the possible solution(s).

3 Question 2 (5 points): Eating Out

Zahid (Z), Ishrak (I), Farabi (F), and Nafisa (N) came to Chini-Come, a restaurant near their university. The restaurant serves Special Rice (S), Biryani Rice (B), Kashmiri Naan (K), and Paratha (P). You overhear their conversations, and come up with the following preferences:

- Zahid does not like Paratha.
- Ishrak and Farabi want to grab a bite of each other's food. So they want to order different dishes.
- Farabi likes Rice items. So he will either take Special Rice or Biryani Rice.
- Zahid loves to copy Ishrak and will order the same dish as him. However, his dish needs to be different from the rest.
- Nafisa will not order Kashmiri Naan as she had them earlier.

We want to determine who will order which food. Formulate the problem as CSP and explore the possible solution(s).

4 Question 3 (6 points): Finding Houses

Four people, Ali (A), Sristy (S), Maliha (M), and Rafid (R) are looking to rent space in an apartment building. There are three floors in the building: 1, 2, and 3 (where 1 is the lowest and 3 is the highest). More than one

person can live on a single floor, but each person must be assigned to some floor. The following constraints must be satisfied on the assignment:

- Ali and Sristy must not live on the same floor.
- If Ali and Maliha live on the same floor, they must both be living on floor 2.
- If Ali and Maliha live on different floors, one of them must be living on floor 3.
- Rafid must not live on the same floor as anyone else.
- Rafid must live on a higher floor than Maliha.

We want to determine who will take which floor. Formulate the problem as CSP and explore the possible solution(s).

5 Question 4 (7 points): Spots

Assume that six friends Rifat (*R*), Atiq (*A*), Farhan (*F*), Ishmam (*I*), Tabassum (*T*), and Sabrina (*S*) are standing in a queue, each occupying a unique spot among the six possible spots labeled 1 to 6.

- Farhan is standing in between Atiq and Ishmam.
- Sabrina and Rifat are standing next to each other.
- Tabassum is either at the front of the line or the back of the line.
- Sabrina has one person behind her.

We want to determine who is standing in which position. Formulate the problem as CSP and explore the possible solution(s).

6 Question 5 (10 points): Scheduling Tasks

You need to prepare a schedule for two faculty members, *X* and *Y*. They need to carry out the following tasks:

- (G) Gather contents for Database Management Systems (DBMS) Lab, which takes 1 hour.
- (Q) Check quiz scripts, which takes 2 consecutive hours.
- (C) Take Artificial Intelligence (AI) class, which takes 1 hour.
- (D) Conduct DBMS Lab, which takes 1 hour.
- (L) Take AI lab, which takes 2 consecutive hours.

The schedule consists of one-hour slots: 8 am - 9 am, 9 am - 10 am, 10 am - 11 am, 11 am - 12 pm. The requirements are as follows:

- At any given time, each faculty member can do at most one task (G, Q, C, D, L).
- The AI class (C) must happen before AI lab (L).
- The contents (G) should be gathered before taking the DBMS Lab (D).
- The DBMS Lab (D) should be finished by 10 am.
- *X* is going to gather contents for DBMS (G) since s/he's good at browsing contents.
- The other faculty member not conducting DBMS lab (D) should attend the lab, and hence cannot do anything else at that time.

- The person taking DBMS Lab (D) does not take AI Lab (L)
- The person taking AI Lab (L) must also take the AI class (C)
- Checking quiz scripts (Q) takes 2 consecutive hours and hence should start at or before 10 am.
- Taking AI Lab (L) takes 2 consecutive hours and hence should start at or before 10 am.

We want to determine what task will be done by whom and when. Formulate the problem as CSP and explore the possible solution(s).

7 Evaluation

Once you are done with the tasks, call your course teacher and show them your implementations.

8 Submission

Submit one file: StudentID_L3.pdf (StudentID will be replaced by your student ID) under **Lab 3** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce), **solution explanation** (provide an overview of your approach, include figures depicting the formulation and/or solution for clarification, highlight key decisions you made during the implementation), **findings and insights** (share any interesting observations/insights you gained, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — simply copying and pasting what you showed in the class will not work. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.

You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 4 Multi-Agent Search

In this lab, 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. As in lab 0, this lab includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
(ailab) user@linux ~/multiagent$ python autograder.py
```

It can be run for one particular question, such as q2, by:

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q2
```

It can be run for one particular test by commands of the form:

```
(ailab) user@linux ~/multiagent$ python autograder.py -t test_cases/q2/0-small-tree
```

By default, the autograder displays graphics with the `-t` option but does not with the `-q` option. You can force graphics by using the `--graphics` flag, or force no graphics by using the `--no-graphics` flag. The code for this lab contains the following files, available in Google Classroom as `multiagent.zip`.

Files you will edit:	
multiAgents.py	Where all of your multi-agent search 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 lab.
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	Lab 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
multiagentTestClasses.py	Task 2 specific autograding test classes

Files to Edit and Show: You will fill in portions of `multiAgents.py` during the task. Please *do not* change the other files in this distribution. Once you are done, call your course teacher and show them this file.

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 judgments – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.

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 do not try. We trust you all to submit your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.

Google Classroom: Please be careful not to post spoilers.

Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 Welcome to Multi-Agent Pacman

First, play a game of classic Pacman by running the following command:

```
(ailab) user@linux ~/multiagent$ python pacman.py
```

and using the arrow keys to move. Now, run the provided `ReflexAgent` in `multiAgents.py`

```
(ailab) user@linux ~/multiagent$ python pacman.py -p ReflexAgent
```

Note that it plays quite poorly even on simple layouts:

```
(ailab) user@linux ~/multiagent$ python pacman.py -p ReflexAgent -l testClassic
```

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

2 Question 1 (4 points): Reflex Agent

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:

```
(ailab) user@linux ~/multiagent$ 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):

```
(ailab) user@linux ~/multiagent$ python pacman.py --frameTime 0 -p ReflexAgent -k 1
(ailab) user@linux ~/multiagent$ 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.

Important Notes:

- Remember that `newFood` has the function `asList()`
- As features, try the reciprocal of important values (such as distance to food) rather than just the values themselves.
- The evaluation function you are writing is evaluating state-action pairs; in later parts of the lab, you will be evaluating states.
- You may find it useful to view the internal contents of various objects for debugging. You can do this by printing the objects' string representations. For example, you can print `newGhostStates` with

```
1 print(str(newGhostStates))
```

- 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.

Grading: We will run your agent on the `openClassic` layout 10 times. You will receive 0 points if your agent times out, or never wins. You will receive 1 point if your agent wins at least 5 times, or 2 points if your agent wins all 10 games. You will receive an additional 1 point if your agent's average score is greater than 500, or 2 points if it is greater than 1000. You can try your agent out under these conditions with

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q1
```

To run it without graphics, use:

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q1 --no-graphics
```

Do not spend too much time on this question, though, as the meat of the lab lies ahead.

3 Question 2 (5 points): Minimax

Now you will write an adversarial search agent in the provided `MinimaxAgent` class stub in `multiAgents.py`. Recall the pseudo-code for the vanilla Minimax Algorithm is as follows:

```
def value(state)
```

```
    if the state is a terminal state: return the state's utility
```

```
    if the next agent is MAX: return max-value(state)
```

```
    if the next agent is MIN: return min-value(state)
```

```
def max-value(state)
```

```
    initialize  $v = -\infty$ 
```

```
    for each successor of state:
```

```
         $v = \max(v, \text{value}(\text{successor}))$ 
```

```
    return  $v$ 
```

```
def min-value(state)
```

```
    initialize  $v = +\infty$ 
```

```
    for each successor of state:
```

```
         $v = \min(v, \text{value}(\text{successor}))$ 
```

```
    return  $v$ 
```

Your minimax agent should work with any number of ghosts, so you will have to write an algorithm that is slightly more general than what you have previously seen in the lecture. 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 `MultiAgentSearchAgent`, 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 Notes

- Implement the algorithm recursively using helper function(s).
- Pacman is always agent 0, and the agents move in order of increasing agent index.
- A single search ply is considered to be one Pacman move and all the ghosts' responses, so a depth 2 search will involve Pacman and each ghost moving two times (see Figure 4.1).

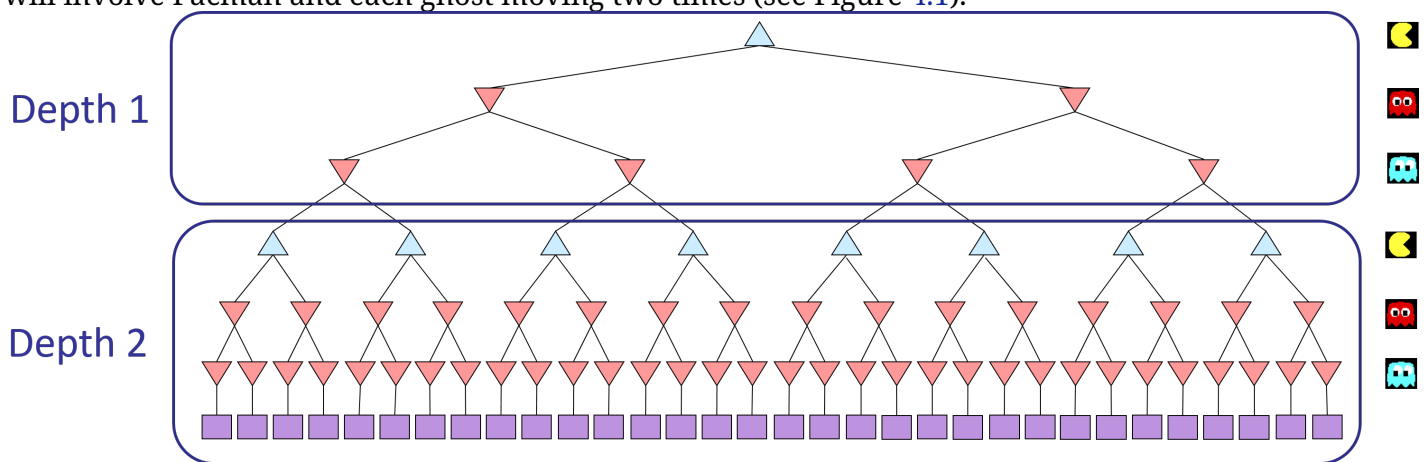


Figure 4.1. Explanation of ply for 2 ghosts

- The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not a problem: as it is correct behavior, it will pass the tests.
- The evaluation function for the Pacman test in this part is already written (`self.evaluationFunction`). You should not change this function but recognize that now we are 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 (665/1000 games for us) despite the dire prediction of depth 4 minimax.

```
(ailab) user@linux ~/multiagent$ python pacman.py -p MinimaxAgent -l minimaxClassic \
-a depth=4
```

- All states in minimax should be `GameStates`, either passed in to `getAction` or generated via `GameState.generateSuccessor`. In this lab, you will not be abstracting to simplified states.
- On larger boards such as `openClassic` and `mediumClassic` (the default), you will find Pacman to be good at not dying, but quite bad at winning. He will 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 would go after eating that dot. Do not worry if you see this behavior, question 5 will clean up all of these issues.

- 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:

```
(ailab) user@linux ~/multiagent$ python pacman.py -p MinimaxAgent -l trappedClassic \
-a depth=3
```

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

Grading: We will be checking your code to determine whether it explores the correct number of game states. This is the only reliable way to detect some very subtle bugs in implementations of minimax. As a result, the autograder will be very picky about how many times you call `GameState.generateSuccessor`. If you call it any more or less than necessary, the autograder will complain. To test and debug your code, run

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q2
```

This will show what your algorithm does on a number of small trees, as well as a Pacman game. To run it without graphics, use:

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q2 --no-graphics
```

4 Question 3 (5 points): Alpha-Beta Pruning

Make a new agent that uses alpha-beta pruning to efficiently explore the minimax tree, in `AlphaBetaAgent` from `multiagents.py`. Your algorithm will be slightly more general than the pseudocode from the lecture, so part of the challenge is to extend the alpha-beta pruning logic appropriately to multiple minimizer agents.

The pseudo-code below represents the algorithm you should implement for this question:

α : MAX's best option on path to root

β : MIN's best option on path to root

`def value(state)`

if the state is a terminal state: return the state's utility

if the next agent is MAX: return `max-value(state, α , β)`

if the next agent is MIN: return `min-value(state, α , β)`

`def max-value(state, α , β)`

initialize $v = -\infty$

for each successor of state:

$v = \max(v, \text{value}(\text{successor}, \alpha, \beta))$

if $v > \beta$ return v

$\alpha = \max(\alpha, v)$

return v

`def min-value(state, α , β)`

initialize $v = +\infty$

for each successor of state:

$v = \min(v, \text{value}(\text{successor}, \alpha, \beta))$

if $v < \alpha$ return v

$\beta = \min(\beta, v)$

return v

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.


```
(ailab) user@linux ~/multiagent$ python pacman.py -p AlphaBetaAgent \
-a depth=3 -l smallClassic
```

You must not prune on equality in order to match the set of states explored by our autograder. (Indeed, alternatively, but incompatible with our autograder, would be to also allow for pruning on equality and invoke alpha-beta once on each child of the root node, but this will not match the autograder.)

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.

Grading: Because we check your code to determine whether it explores the correct number of states, it is important that you perform alpha-beta pruning without reordering children. In other words, successor states should always be processed in the order returned by GameState.getLegalActions. Again, do not call GameState.generateSuccessor more than necessary.

To test and debug your code, run

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q3
```

This will show what your algorithm does on a number of small trees, as well as a Pacman game. To run it without graphics, use:

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q3 --no-graphics
```

The correct implementation of alpha-beta pruning will lead to Pacman losing some of the tests. This is not a problem: as it is correct behavior, it will pass the tests.

5 Question 4 (5 points): Expectimax

Minimax and alpha-beta are great, but they both assume that you are playing against an adversary who makes optimal decisions. As anyone who has ever won tic-tac-toe can tell you, this is not always the case. In this question, you will implement the ExpectimaxAgent, which is useful for modeling the probabilistic behavior of agents who may make suboptimal choices. The pseudo-code below represents the algorithm you should implement for this question:

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v+ = p × value(successor)
    return v
```

As with the search and constraint satisfaction problems covered so far in this class, the beauty of these algorithms is their general applicability. To expedite your own development, we have supplied some test cases based on generic trees. You can debug your implementation on small the game trees using the command:

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q4
```


Debugging on these small and manageable test cases is recommended and will help you to find bugs quickly.

Once your algorithm is working on small trees, you can observe its success in Pacman. Random ghosts are of course not optimal minimax agents, so modeling them with minimax search may not be appropriate. ExpectimaxAgent, will no longer take the min over all ghost actions, but the expectation according to your agent's model of how the ghosts act. To simplify your code, assume you will only be running against an adversary which chooses amongst their getLegalActions uniformly at random while you try to play optimally.

To see how the ExpectimaxAgent behaves in Pacman, run:

```
(ailab) user@linux ~/multiagent$ python pacman.py -p ExpectimaxAgent \
-l minimaxClassic -a depth=3
```

You should now observe a more cavalier approach in close quarters with ghosts. In particular, if Pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he will at least try. Investigate the results of these two scenarios:

```
(ailab) user@linux ~/multiagent$ python pacman.py -p AlphaBetaAgent \
-l trappedClassic -a depth=3 -q -n 10
(ailab) user@linux ~/multiagent$ python pacman.py -p ExpectimaxAgent \
-l trappedClassic -a depth=3 -q -n 10
```

You should find that your ExpectimaxAgent wins about half the time, while your AlphaBetaAgent always loses. Make sure you understand why the behavior here differs from the minimax case.

Grading: To test and debug your code, run

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q4
```

This will show what your algorithm does on a number of small trees, as well as a Pacman game. To run it without graphics, use:

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q4 --no-graphics
```

The correct implementation of expectimax will lead to Pacman losing some of the tests. This is not a problem: as it is correct behavior, it will pass the tests.

6 Question 5 (6 points): Evaluation Function

Write a better evaluation function for Pacman in the provided function betterEvaluationFunction from multiagents.py. The evaluation function should evaluate states, rather than actions like your reflex agent evaluation function did in Question 1. You may use any tools at your disposal for evaluation, including your search code from the last project. With depth 2 search, your evaluation function should clear the smallClassic layout with one random ghost more than half the time and still run at a reasonable rate (to get full credit, Pacman should be averaging around 1000 points when he's winning).

Grading: the autograder will run your agent on the smallClassic layout 10 times. We will assign points to your evaluation function in the following way:

- ▷ If you win at least once without timing out the autograder, you receive 1 point. Any agent not satisfying these criteria will receive 0 points.
- ▷ +1 for winning at least 5 times, +2 for winning all 10 times
- ▷ +1 for an average score of at least 500, +2 for an average score of at least 1000 (including scores on lost games)
- ▷ +1 if your games take on average less than 30 seconds on the autograder machine when run with `--no-graphics`. The autograder is run on a lab PC, so this machine will have a fair amount of resources, but your personal computer could be far less performant (netbooks) or far more performant (gaming rigs).
- ▷ The additional points for average score and computation time will only be awarded if you win at least 5 times.

You can try your agent out under these conditions with

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q5
```

To run it without graphics, use:

```
(ailab) user@linux ~/multiagent$ python autograder.py -q q5 --no-graphics
```

7 Evaluation

Once you are done with the tasks, run autograder for all the tasks via:

```
(ailab) user@linux ~/multiagent$ python autograder.py
```

Then call your course teacher and show them the autograder results and codes.

8 Submission

Submit one file: `StudentID_L4.pdf` (StudentID will be replaced by your student ID) under **Lab 4** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce in the algorithms/code), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, hypothesize on why such phenomenon occurred, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.

You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 5 Markov Decision Process

In this lab, you will implement value iteration. You will test your agent on Gridworld.

As in previous tasks, this lab includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
(ailab) user@linux ~/mdp$ python autograder.py
```

It can be run for one particular question, such as q2, by:

```
(ailab) user@linux ~/mdp$ python autograder.py -q q2
```

It can be run for one particular test by commands of the form:

```
(ailab) user@linux ~/mdp$ python autograder.py -t test_cases/q2/1-bridge-grid
```

The code for this lab contains the following files, available in Google Classroom as `mdp.zip`.

Files you will edit:	
valueIterationAgents.py	A value iteration agent for solving known MDPs.
analysis.py	A file to put your answers to questions given in the lab.
Files you might want to look at:	
mdp.py	Defines methods on general MDPs.
learningAgents.py	Defines the base classes ValueEstimationAgent, which your agents will extend.
util.py	Useful data structures for implementing MDPs.
gridworld.py	The Gridworld implementation.
Supporting files you can ignore:	
environment.py	Abstract class for general reinforcement learning environments. Used by gridworld.py.
graphicsGridworldDisplay.py	Gridworld graphical display.
graphicsUtils.py	Graphics utilities.
textGridworldDisplay.py	Plug-in for the Gridworld text interface.
crawler.py	The crawler code and test harness. You will run this but not edit it.
graphicsCrawlerDisplay.py	GUI for the crawler robot.
qlearningAgents.py	Q-learning agents for Gridworld, Crawler and Pacman.
featureExtractors.py	Classes for extracting features on (state, action) pairs. Used for the approximate Q-learning agent (in qlearningAgents.py).
autograder.py	Lab 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
reinforcementTestClasses.py	Lab specific autograding test classes

Files to Edit and show: You will fill in portions of `valueIterationAgents.py` and `analysis.py` during the task. Please *do not* change the other files in this distribution. Once you are done, call your course teacher and show them this file.

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 judgments – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.

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 do not try. We trust you all to submit your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.

Google Classroom: Please be careful not to post spoilers.

Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 MDPs

To get started, run Gridworld in manual control mode, which uses the arrow keys:

```
(ailab) user@linux ~/mdp$ python gridworld.py -m
```

You will see the two-exit layout from class. The blue dot is the agent. Note that when you press up, the agent only actually moves north 80% of the time. Such is the life of a Gridworld agent!

You can control many aspects of the simulation. A full list of options is available by running:

```
(ailab) user@linux ~/mdp$ python gridworld.py -h
```

The default agent moves randomly

```
(ailab) user@linux ~/mdp$ python gridworld.py -g MazeGrid
```

You should see the random agent bounce around the grid until it happens upon an exit. Not the finest hour for an AI agent.

Note that, the Gridworld MDP is such that you first must enter a pre-terminal state (the double boxes shown in the GUI) and then take the special ‘exit’ action before the episode ends (in the true terminal state called `TERMINAL_STATE`, which is not shown in the GUI). If you run an episode manually, your total return may be less than you expected, due to the discount rate (`-d` to change; 0.9 by default).

Look at the console output that accompanies the graphical output (or use `-t` for all text). You will be told about each transition the agent experiences (to turn this off, use `-q`).

As in Pacman, positions are represented by (x, y) Cartesian coordinates, and any arrays are indexed by $[x][y]$, with ‘north’ being the direction of increasing y , etc. By default, most transitions will receive a reward of zero, though you can change this with the living reward option (`-r`).

2 Question 1 (4 points): Value Iteration

Recall the value iteration state update equation:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

Write a value iteration agent in `ValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner that takes the number of iterations (option `-i`) as input in its initial planning phase. `ValueIterationAgent` takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.

Value iteration computes k -step estimates of the optimal values, V_k . In addition to running value iteration, implement the following methods for `ValueIterationAgent` using V_k .

- `computeActionFromValues(state)` computes the best action according to the value function given by `self.values`.
- `computeQValueFromValues(state, action)` returns the Q-value of the (state, action) pair given by the value function given by `self.values`.

These quantities are all displayed in the GUI: values are numbers in squares, Q-values are numbers in square quarters, and policies are arrows out from each square.

You should return the synthesized policy π_{k+1} .

Important Notes:

- Use the “batch” version of value iteration where each vector V_k is computed from a fixed vector V_{k-1} (as shown in the lecture), not the “online” version where one single weight vector is updated in place. This means that when a state’s value is updated in iteration k based on the values of its successor states, the successor state values used in the value update computation should be those from iteration $k - 1$ (even if some of the successor states had already been updated in iteration k). The difference is discussed in Sutton & Barto in Chapter 4.1 on page 91.
- A policy synthesized from values of depth k (which reflect the next k rewards) will actually reflect the next $k + 1$ rewards (i.e. you return π_{k+1}). Similarly, the Q-values will also reflect one more reward than the values (i.e. you return Q_{k+1}).
- On the default BookGrid, you can run value iteration for 5 iterations using the following command:

```
(ailab) user@linux ~/mdp$ python gridworld.py -a value -i 5
```

The result of running the code should look like Figure 5.1.

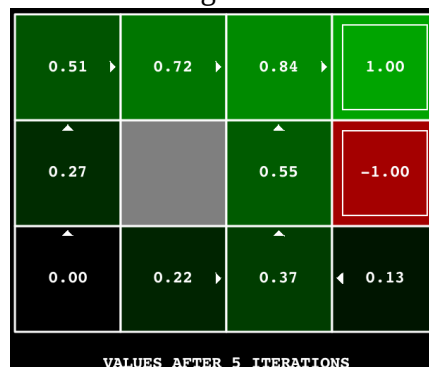


Figure 5.1. Result of running value iteration for 5 iterations in BookGrid

- You may optionally use the `util.Counter` class in `util.py`, which is a dictionary with a default value of zero. However, be careful with `argMax`: the actual `argmax` you want may be a key, not in the counter!
- Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

Grading: To test your implementation, run the autograder:

```
(ailab) user@linux ~/mdp$ python autograder.py -q q1
```

The following command loads your `ValueIterationAgent`, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state ($V(\text{start})$), which you can read off of the GUI and the empirical resulting average reward (printer after the 10 rounds of execution finish) are quite close.

```
(ailab) user@linux ~/mdp$ python gridworld.py -a value -i 100 -k 10
```

Your value iteration agent will be graded in a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g. after 100 iterations).

3 Question 2 (1 point): Bridge Crossing Analysis

BridgeGrid is a grid world map with a low-reward terminal state and a high-reward terminal state separated by a narrow “bridge”, on either side of which is a chasm of high negative reward. Here, the expected value of a state depends on a number of factors, including how the future rewards are discounted, how noisy the actions are, and how much reward is received in the non-terminal states. The agent starts near the low-reward state. With the default discount of 0.9 and the default noise of 0.2, the optimal policy does not cross the bridge. Here, noise refers to how often an agent ends up in an unintended successor state when they perform an action. And discount determines how much the agent cares about rewards in the distant future relative to those in the immediate future. Change only ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge. Put your answer in `question2()` of `analysis.py`. The default can be seen using the following command:

```
(ailab) user@linux ~/mdp$ python gridworld.py -a value -i 100 -g BridgeGrid \
--discount 0.9 --noise 0.2
```

The resultant grid can be seen in Figure 5.2.



Figure 5.2. Default Values for Question 2

Grading: We will check that you only changed one of the given parameters and that with this change, a correct value iteration agent should cross the bridge. To check your answer, run the autograder:


```
(ailab) user@linux ~/mdp$ python autograder.py -q q2
```

4 Question 3 (5 points): Policies

Consider the DiscountGrid layout, shown in Figure 5.3. This grid has two terminal states with a positive payoff (in the middle row) – a close (near) exit with payoff +1, and a distant exit with payoff +10. The bottom row of the grid consists of terminal states with negative payoff (shown in red); each state in this “cliff” region has a payoff of -10. The starting state is the yellow square.

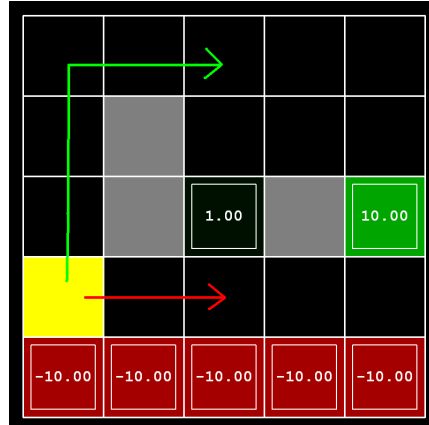


Figure 5.3. DiscountGrid Layout

We distinguish between two types of paths:

1. paths that “risk the cliff” and travel near the bottom row of the grid; these paths are shorter but risk earning a large negative payoff, and are represented by the red arrow in Figure 5.3.
2. paths that “avoid the cliff” and travel along the top edge of the grid. These paths are longer but are less likely to incur huge negative payoffs. These paths are represented by the green arrow in Figure 5.3.

In this question, you will choose values of the discount, noise, and living reward parameters for this MDP to produce optimal policies of several different types that are given below. **Your setting of the parameter values for each part should have the property that, if your agent followed its optimal policy without being subject to any noise, it would exhibit the given behavior.** If a particular behavior is not achieved for any setting of the parameters, assert that the policy is impossible by returning the string ‘NOT POSSIBLE’.

Here are the optimal policy types you should attempt to produce:

- a. Prefer the close exit (+1), risking the cliff (-10)
- b. Prefer the close exit (+1), avoiding the cliff (-1)
- c. Prefer the distant exit (+10), risking the cliff (-10)
- d. Prefer the distant exit (+10), avoiding the cliff (-10)
- e. Avoid both exits and the cliff (so an episode should never terminate)

To see what behavior a set of numbers ends up in, run the following command to see a GUI:

```
(ailab) user@linux ~/mdp$ python gridworld.py -g DiscountGrid -a value \
--discount [YOUR_DISCOUNT] --noise [YOUR_NOISE] --livingReward [YOUR_LIVING_REWARD]
```

Write your answers in `analysis.py`: `question3a()` through `question3e()` should each return a 3-item tuple of discount, noise, and living reward.

You can check your policies in the GUI with commands like this:

```
(ailab) user@linux ~/mdp$ python gridworld.py -a value -i 100 -g DiscountGrid \
-d 0.0 -n 0.0 -r 0.0
```

This will run the value iteration agent for 100 iterations on the DiscountGrid layout. The discount, noise, and living reward all are set to 0.

As shown in Figure 5.4, using a correct answer to 3(a), the arrow in (0, 1) should point east, the arrow in (1, 1) should also point east, and the arrow in (2, 1) should point north.

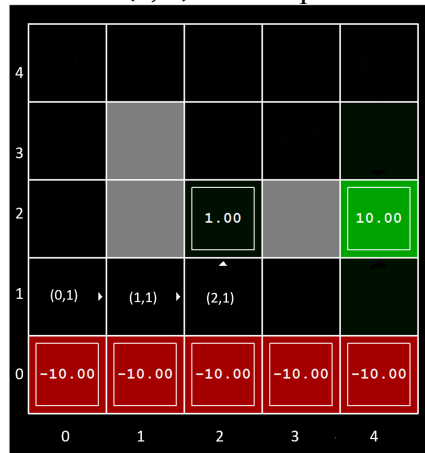


Figure 5.4. Sample policy with the index shown in Parentheses

On some machines you may not see an arrow. In this case, press a button on the keyboard to switch to the qValues display, and mentally calculate the policy by taking the argmax of the available qValues for each state.

Grading: We will check that the desired policy is returned in each case. To check your answers, run the autograder:

```
(ailab) user@linux ~/mdp$ python autograder.py -q q3
```

5 Question 4 (1 point): Asynchronous Value Iteration

Write a value iteration agent in `AsynchronousValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`. Your value iteration agent is an offline planner that takes the number of iterations (option `-i`) as input in its initial planning phase.

`AsynchronousValueIterationAgent` takes an MDP on construction and runs *cyclic* value iteration (described in the next paragraph) for the specified number of iterations before the constructor returns. Note that all this value iteration code should be placed inside the constructor (`__init__` method).

The reason this class is called `AsynchronousValueIterationAgent` is because we will update only **one** state in each iteration, as opposed to doing a batch-style update. Here is how cyclic value iteration works. In the first iteration, only update the value of the first state in the state's list. In the second iteration, only update the value of the second. Keep going until you have updated the value of each state once, then start back at the first state for the subsequent iteration. **If the state picked for updating is terminal, nothing happens in that iteration.** You can implement it as indexing into the states variable defined in the code skeleton.

As a reminder, here's the value iteration state update equation:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V_k(s')]$$

Value iteration iterates a fixed-point equation, as discussed in class. It is also possible to update the state values in different ways, such as in a random order (i.e., select a state randomly, update its value, and repeat) or in a batch style (as in Q1). In Q4, we will explore another technique. We want to iterate through the states in the order provided by the `getStates()` functions and update one state per iteration.

`AsynchronousValueIterationAgent` inherits from `ValueIterationAgent` from Q1, so the only method you need to implement is `runValueIteration`. Since the superclass constructor calls `runValueIteration`, overriding it is sufficient to change the agent's behavior as desired.

Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

The following command loads your `AsynchronousValueIterationAgent` in the Gridworld, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state ($V(\text{start})$), which you can read off of the GUI and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

```
(ailab) user@linux ~/mdp$ python gridworld.py -a asynchvalue -i 1000 -k 10
```

Grading: Your value iteration agent will be graded on a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g., after 1000 iterations).

To test your implementation, run the autograder. It should take less than a second to run.

```
(ailab) user@linux ~/mdp$ python autograder.py -q q4
```

6 Question 5 (1 point): Prioritized Sweeping Value Iteration

You will now implement `PrioritizedSweepingValueIterationAgent`, which has been partially specified for you in `valueIterationAgents.py`.

Note that this class derives from `AsynchronousValueIterationAgent`, so the only method that needs to change is `runValueIteration`, which actually runs the value iteration.

Prioritized sweeping attempts to focus updates of state values in ways that are likely to change the policy. For this lab, you will implement a simplified version of the standard prioritized sweeping algorithm, which is described in [this paper](#). We have adapted this algorithm for our setting. First, we define the **predecessors** of a state s as all states that have a **nonzero** probability of reaching s by taking some action a . Also, θ , which is passed in as a parameter, will represent our tolerance for error when deciding whether to update the value of a state. Here's the algorithm you should follow in your implementation:

- Compute predecessors of all states. To do that, get all states, iterated over them to check if it is a terminal state or not. If it is not a terminal state, find all successor in which can end up from this state using all the available actions.
- Initialize an empty priority queue.
- For each non-terminal state s , do: (note: to make the autograder work for this question, you must iterate over states in the order returned by `self.mdp.getStates()`).
 - ◊ Find the absolute value of the difference between the current value of s in `self.values` and the highest Q-value across all possible actions from s (this represents what the value should be); call this number `diff`. Do NOT update `self.values[s]` in this step.

- ◊ Push s into the priority queue with priority $-diff$ (note that this is **negative**). We use a negative because the priority queue is a min-heap, but we want to prioritize updating states that have a **higher** error.
- For iteration in $0, 1, 2, \dots, self.iterations - 1$, do:
 - ◊ If the priority queue is empty, then terminate.
 - ◊ Pop a state s off the priority queue.
 - ◊ Update s 's value (if it is not a terminal state) in `self.values`.
 - ◊ For each predecessor p of s , do:
 - * Find the absolute value of the difference between the current value of p in `self.values` and the highest Q-value across all possible actions from p (this represents what the value should be); call this number `diff`.
Do NOT update `self.values[p]` in this step.
 - * If `diff > theta`, push p into the priority queue with priority $-diff$ (note that this is **negative**), as long as it does not already exist in the priority queue with equal or lower priority. As before, we use a negative because the priority queue is a min-heap, but we want to prioritize updating states that have a **higher** error.

A couple of important notes on implementation:

- When you compute predecessors of a state, make sure to store them in a **set**, not a list, to avoid duplicates.
- Please use `util.PriorityQueue` in your implementation. The update method in this class will likely be useful; look at its documentation.

Grading: Your prioritized sweeping value iteration agent will be graded on a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g., after 1000 iterations).

To test your implementation, run the autograder. It should take about 1 second to run.

```
(ailab) user@linux ~/mdp$ python autograder.py -q q5
```

You can run the `PrioritizedSweepingValueIterationAgent` in the Gridworld using the following command:

```
(ailab) user@linux ~/mdp$ python gridworld.py -a priosweepvalue -i 1000
```

7 Evaluation

Once you are done with the tasks, run autograder for all the tasks via:

```
(ailab) user@linux ~/mdp$ python autograder.py
```

Then call your course teacher and show them the autograder results and codes.

8 Submission

Submit one file: `StudentID_L5.pdf` (StudentID will be replaced by your student ID) under **Lab 5** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you

tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce in the algorithms/code), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, hypothesize on why such phenomenon occurred, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.

You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Lab 6 Reinforcement Learning

In this lab, you will implement Q-learning. You will test your agent first on Gridworld (from class), then apply them to a simulated robot controller (Crawler) and Pacman.

As in previous tasks, this lab includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
(ailab) user@linux ~/reinforcement$ python autograder.py
```

The code for this lab contains the following files, available in Google Classroom as `reinforcement.zip`.

Files you will edit:	
qlearningAgents.py	Q-learning agents for Gridworld, Crawler and Pacman.
analysis.py	A file to put your answers to questions given in the lab.
Files you might want to look at:	
learningAgents.py	Defines the base classes ValueEstimationAgent and QLearningAgent, which your agents will extend.
util.py	Utilities, including util.Counter, which is particularly useful for Q-learners.
gridworld.py	The Gridworld implementation.
featureExtractors.py	Classes for extracting features on (state, action) pairs. Used for the approximate Q-learning agent (in qlearningAgents.py).
Supporting files you can ignore:	
mdp.py	Defines methods on general MDPs.
valueIterationAgents.py	A value iteration agent for solving known MDPs.
environment.py	Abstract class for general reinforcement learning environments. Used by gridworld.py.
graphicsGridworldDisplay.py	Gridworld graphical display.
graphicsUtils.py	Graphics utilities.
textGridworldDisplay.py	Plug-in for the Gridworld text interface.
crawler.py	The crawler code and test harness. You will run this but not edit it.
graphicsCrawlerDisplay.py	GUI for the crawler robot.
autograder.py	Lab 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
reinforcementTestClasses.py	Lab specific autograding test classes

Files to Edit and Show: You will fill in portions of `qlearningAgents.py` and `analysis.py` during the task. Please *do not* change the other files in this distribution. Once you are done, call your course teacher and show them 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 judgments – will be the final judge of your score. We will review and grade assignments individually to ensure that you receive due credit for your work.

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 do not try. We trust you all to submit your own work only; please do not let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact us for help. Office hours, Google Classroom, and Emails are there for your support; please use them. We want these labs to be rewarding and instructional, not frustrating and demoralizing. But, we do not know when or how to help unless you ask.

Google Classroom: Please be careful not to post spoilers.

Report Due: 2 weeks after the lab. Please check the submission deadline in the post for more details.

1 Question 6 (4 points): Q-Learning

Note that your value iteration agent does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a simulated environment like a Gridworld, but it is very important in the real world, where the real MDP is not available.

You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its `update(state, action, nextState, reward)` method. A stub of a Q-learner is specified in `QLearningAgent` in `qlearningAgents.py`, and you can select it with the option ‘-a q’. For this question, you must implement the `update`, `computeValueFromQValues`, `getQValue`, and `computeActionFromQValues` methods.

Important Notes:

- For `computeActionFromQValues`, you should break ties randomly for better behavior. The `random.choice()` function will help.
- When taking action, you should consider all the legal actions which also includes the actions your agent has not seen before. In a particular state, actions that your agent *has not* seen before still have a Q-value, specifically a Q-value of zero, and if all of the actions that your agent has seen before have a negative Q-value, an unseen action may be optimal.
- Make sure that in your `computeValueFromQValues` and `computeActionFromQValues` functions, you only access Q values by calling `getQValue`. This abstraction will be useful for question 10 when you override `getQValue` to use features of state-action pairs rather than state-action pairs directly.
- With the Q-learning update in place, you can watch your Q-learner learn under manual control, using the keyboard:

```
(ailab) user@linux ~/reinforcement$ python gridworld.py -a q -k 5 -m
```

Recall that `-k` will control the number of episodes your agent gets to learn. Watch how the agent learns about the state it was just in, not the one it moves to, and “leaves learning in its wake.”

- To help with debugging, you can turn off noise by using the `-noise 0.0` parameter (though this obviously makes Q-learning less interesting). If you manually steer Pacman north and then east along the optimal path for four episodes, you should see the Q-values shown in Figure 6.1.

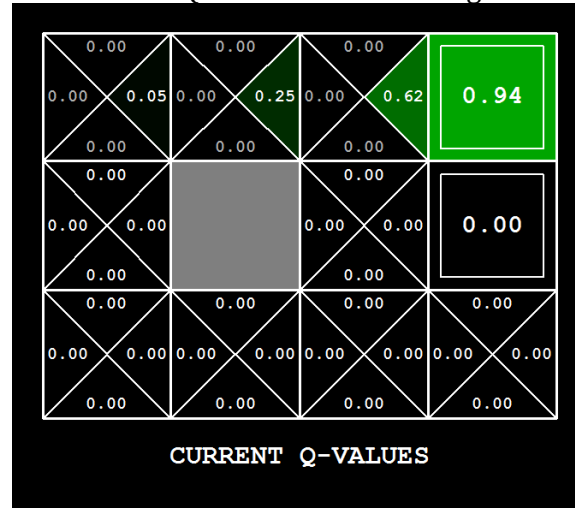


Figure 6.1. Optimal Path for Four Episodes (Manual)

Grading: We will run your Q-learning agent and check that it learns the same Q-values and policy as our reference implementation when each is presented with the same set of examples. To grade your implementation, run the autograder:

```
(ailab) user@linux ~/reinforcement$ python autograder.py -q q6
```

2 Question 7 (2 points): Epsilon Greedy

Remember that we need an Epsilon Greedy strategy to balance exploration and exploitation. The algorithm explores $\epsilon\%$ of the time and then exploits the best option k greedily. That means, initially it chooses random actions for a fraction of time and follows its current best Q-values otherwise. Now, complete your Q-learning agent by implementing epsilon-greedy action selection in `getAction`. Note that choosing a random action may result in choosing the best action - that is, you should not choose a random sub-optimal action, but rather any random legal action.

You can choose an element from a list uniformly at random by calling the `random.choice` function. You can simulate a binary variable with probability p of success by using `util.flipCoin(p)`, which returns `True` with probability p and `False` with probability $1 - p$.

After implementing the `getAction` method, observe the following behavior of the agent in `gridworld` (with $\epsilon = 0.3$).

```
(ailab) user@linux ~/reinforcement$ python gridworld.py -a q -k 100
```

Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths. However, your average returns will be lower than the Q-values predict because of the random actions and the initial learning phase.

You can also observe the following simulations for different epsilon values. Does the behavior of the agent match what you expect?


```
(ailab) user@linux ~/reinforcement$ python gridworld.py -a q -k 100 --noise 0.0 -e 0.1
(ailab) user@linux ~/reinforcement$ python gridworld.py -a q -k 100 --noise 0.0 -e 0.9
```

With no additional code, you should now be able to run a Q-learning crawler robot:

```
(ailab) user@linux ~/reinforcement$ python crawler.py
```

If this does not work, you have probably written some code too specific to the GridWorld problem and you should make it more general to all MDPs.

This will invoke the crawling robot from the class using your Q-learner. Play around with the various learning parameters to see how they affect the agent's policies and actions. Note that the step delay is a parameter of the simulation, whereas the learning rate and epsilon are parameters of your learning algorithm, and the discount factor is a property of the environment.

Grading: To test your implementation, run the autograder:

```
(ailab) user@linux ~/reinforcement$ python autograder.py -q q7
```

3 Question 8 (1 point): Bridge Crossing Revisited

First, train a completely random Q-learner with the default learning rate on the noiseless BridgeGrid for 50 episodes and observe whether it finds the optimal policy.

```
(ailab) user@linux ~/reinforcement$ python gridworld.py -a q -k 50 -n 0 \
-g BridgeGrid -e 1
```

Now try the same experiment with an epsilon of 0. Is there an epsilon and a learning rate for which it is highly likely (greater than 99%) that the optimal policy will be learned after 50 iterations? `question8()` in `analysis.py` should return EITHER a 2-item tuple of (epsilon, learning rate) OR the string 'NOT POSSIBLE' if there is none. Epsilon is controlled by `-e`, learning rate by `-l`.

Note that, your response should not depend on the exact tie-breaking mechanism used to choose actions. This means your answer should be correct even if for instance we rotated the entire bridge grid world 90 degrees.

Grading: To grade your answer, run the autograder:

```
(ailab) user@linux ~/reinforcement$ python autograder.py -q q8
```

4 Question 9 (1 point): Q-Learning and Pacman

Time to play some Pacman! Pacman will play games in two phases. In the first phase, training, Pacman will begin to learn about the values of positions and actions. Because it takes a very long time to learn accurate Q-values even for tiny grids, Pacman's training games run in quiet mode by default, with no GUI (or console) display. Once Pacman's training is complete, he will enter *testing* mode. When testing, Pacman's `self.epsilon` and `self.alpha` will be set to 0.0, effectively stopping Q-learning and disabling exploration, in order to allow Pacman to exploit his learned policy. Test games are shown in the GUI by default. Without any code changes you should be able to run Q-learning Pacman for very tiny grids as follows:


```
(ailab) user@linux ~/reinforcement$ python pacman.py -p PacmanQAgent -x 2000 \
-n 2010 -l smallGrid
```

Note that PacmanQAgent is already defined for you in terms of the QLearningAgent you have already written. PacmanQAgent is only different in that it has default learning parameters that are more effective for the Pacman problem ($\epsilon=0.05$, $\alpha=0.2$, $\gamma=0.8$). You will receive full credit for this question if the command above works without exceptions and your agent wins at least 80% of the time. If you have considered the unseen actions while implementing `getAction` and/or `computeActionFromQValues` methods, then you should be getting full credit for this question. The autograder will run 100 test games after the 2000 training games.

If your QLearningAgent works for `gridworld.py` and `crawler.py` but does not seem to be learning a good policy for Pacman on `smallGrid`, it may be because your `getAction` and/or `computeActionFromQValues` methods do not in some cases properly consider unseen actions. In particular, because unseen actions have by definition a Q-value of zero if all of the actions that have been seen have negative Q-values, an unseen action may be optimal. Beware of the `argmax` function from `util.Counter`!

Grading: To grade your answer, run:

```
(ailab) user@linux ~/reinforcement$ python autograder.py -q q9
```

Important Notes:

- If you want to experiment with learning parameters, you can use the option `-a`, for example, `-a epsilon=0.1,alpha=0.3,gamma=0.7`. These values will then be accessible as `self.epsilon`, `self.gamma` and `self.alpha` inside the agent.
- While a total of 2010 games will be played, the first 2000 games will not be displayed because of the option `-x 2000`, which designates the first 2000 games for training (no output). Thus, you will only see Pacman play the last 10 of these games. The number of training games is also passed to your agent as the option `numTraining`.
- If you want to watch 10 training games to see what's going on, use the command:

```
(ailab) user@linux ~/reinforcement$ python pacman.py -p PacmanQAgent -n 10 \
-l smallGrid -a numTraining=10
```

During training, you will see output every 100 games with statistics about how Pacman is faring. Epsilon is positive during training, so Pacman will play poorly even after having learned a good policy: this is because he occasionally makes a random exploratory move into a ghost. As a benchmark, it should take between 1,000 and 1400 games before Pacman's rewards for a 100-episode segment becomes positive, reflecting that he's started winning more than losing. By the end of the training, it should remain positive and be fairly high (between 100 and 350).

Make sure you understand what is happening here: the MDP state is the exact board configuration facing Pacman, with the now complex transitions describing an entire ply of change to that state. The intermediate game configurations in which Pacman has moved but the ghosts have not replied are not MDP states, but are bundled into the transitions.

Once Pacman is done training, he should win very reliably in test games (at least 90% of the time), since now he is exploiting his learned policy.

However, you will find that training the same agent on the seemingly simple `mediumGrid` does not work well. In our implementation, Pacman's average training rewards remain negative throughout training. At test time, he plays badly, probably losing all of his test games. Training will also take a long time, despite its ineffectiveness.

Pacman fails to win on larger layouts because each board configuration is a separate state with separate Q-values. He has no way to generalize that running into a ghost is bad for all positions. Obviously, this approach will not scale.

5 Question 10 (3 points): Approximate Q-Learning

Implement an approximate Q-learning agent that learns weights for features of states, where many states might share the same features. Write your implementation in `ApproximateQAgent` class in `qlearningAgents.py`, which is a subclass of `PacmanQAgent`.

Note that approximate Q-learning assumes the existence of a feature function $f(s, a)$ over state and action pairs, which yields a vector $f_1(s, a) \dots f_i(s, a) \dots f_n(s, a)$ of feature values. We provide feature functions for you in `featureExtractors.py`. Feature vectors are `util.Counter` (like a dictionary) objects containing the non-zero pairs of features and values; all omitted features have a value of zero.

The approximate Q-function takes the following form

$$Q(s, a) = \sum_{i=1}^n f_i(s, a) w_i$$

where each weight w_i is associated with a particular feature $f_i(s, a)$. In your code, you should implement the weight vector as a dictionary mapping features (which the feature extractors will return) to weight values. You will update your weight vectors similarly to how you updated Q-values:

$$w_i \leftarrow w_i + \alpha \cdot \text{difference} \cdot f_i(s, a)$$

$$\text{difference} = (r + \gamma \max_{a'} Q(s', a')) - Q(s, a)$$

Here, the *difference* term is the same as in normal Q-learning, and r is the experience reward.

By default, `ApproximateQAgent` uses the `IdentityExtractor`, which assigns a single feature to every (state, action) pair. With this feature extractor, your approximate Q-learning agent should work identically to `PacmanQAgent`. You can test this with the following command:

```
(ailab) user@linux ~/reinforcement$ python pacman.py -p ApproximateQAgent -x 2000 \
-n 2010 -l smallGrid
```

`ApproximateQAgent` is a subclass of `QLearningAgent`, and it therefore shares several methods like `getAction`. Make sure that your methods in `QLearningAgent` call `getQValue` instead of accessing Q-values directly, so that when you override `getQValue` in your approximate agent, the new approximate q-values are used to compute actions.

Once you are confident that your approximate learner works correctly with the identity features, run your approximate Q-learning agent with our custom feature extractor, which can learn to win with ease:

```
(ailab) user@linux ~/reinforcement$ python pacman.py -p ApproximateQAgent \
-a extractor=SimpleExtractor -x 50 -n 60 -l mediumGrid
```

Even much larger layouts should be no problem for your ApproximateQAgent. (*warning*: this may take a few minutes to train)

```
(ailab) user@linux ~/reinforcement$ python pacman.py -p ApproximateQAgent \
-a extractor=SimpleExtractor -x 50 -n 60 -l mediumClassic
```

If you have no errors, your approximate Q-learning agent should win almost every time with these simple features, even with only 50 training games.

Grading: We will run your approximate Q-learning agent and check that it learns the same Q-values and feature weights as our reference implementation when each is presented with the same set of examples. To grade your implementation, run the autograder:

```
(ailab) user@linux ~/reinforcement$ python autograder.py -q q10
```

Congratulations! You have a learning Pacman agent!

6 Evaluation

Once you are done with the tasks, run autograder for all the tasks via:

```
(ailab) user@linux ~/reinforcement$ python autograder.py
```

Then call your course teacher and show them the autograder results and codes.

7 Submission

Submit one file: StudentID_L6.pdf (StudentID will be replaced by your student ID) under **Lab 6** on **Google Classroom**. The report can contain (but is not limited to) **introduction** (briefly introduce the problem you tackled, highlight the specific algorithm/technique you implemented), **problem analysis** (discuss the nuances of the problem, explain what modification or unique aspects you were asked to introduce in the algorithms/code), **solution explanation** (provide an overview of your approach, include portions of the code only where necessary for clarification, highlight key decisions you made during the implementation, analyze the complexity of your code), **findings and insights** (share any interesting observations/insights you gained, hypothesize on why such phenomenon occurred, discuss how your solution performed in various scenarios), **challenges faced** (describe any challenges you encountered, explain how you overcame these challenges), **hyperparameter exploration** (discuss the impact of different hyperparameter values on your solution's behavior, analyze how changes influenced the performance), **additional information** (feel free to include any extra details, such as extensions, additional features, or creative elements you added to your solution). The report is about insights, analysis, and reflection — no need to duplicate your entire code. Be clear and concise in your explanations. This is your chance to showcase your understanding and creativity, so make the most of it.

You will have **2 weeks** to submit the file. Plagiarism is strictly prohibited and will result in significant penalties.

Appendix A Linux Basics

Linux is a family of open-source Unix-like operating systems based on the Linux kernel, an operating system kernel first released on September 17, 1991, by Linus Torvalds. Linux is typically packaged as a Linux distribution (distro), which includes the kernel and supporting system software and libraries, many of which are provided by the GNU Project. Many Linux distributions use the word “Linux” in their name, but the Free Software Foundation uses and recommends the name “GNU/Linux” to emphasize the use and importance of GNU software in many distributions, causing some controversy.

Popular Linux distributions include Debian, Fedora Linux, Arch Linux, and Ubuntu. Commercial distributions include Red Hat Enterprise Linux and SUSE Linux Enterprise. Desktop Linux distributions include a windowing system such as X11 or Wayland and a desktop environment such as GNOME or KDE Plasma. Distributions intended for servers may not have a graphical user interface at all, or include a solution stack such as LAMP. Because Linux is freely redistributable, anyone may create a distribution for any purpose.

1 File/Directory Manipulation

When you open a terminal window, you are placed at a command prompt:

```
[user@linux: ~]$
```

The prompt shows your username, the host you are logged onto, and your current location in the directory structure (your path). The tilde character is shorthand for your home directory. Note your prompt may look slightly different. To make a directory, use the `mkdir` command. Use `cd` to change to that directory:

```
[user@linux ~]$ mkdir foo
[user@linux ~]$ cd foo
[user@linux ~/foo]$
```

Use `ls` to see a listing of the contents of a directory and `touch` to create an empty file:

```
[user@linux ~/foo]$ ls
[user@linux ~/foo]$ touch hello_world
[user@linux ~/foo]$ ls
hello_world
[user@linux ~/foo]$ cd ..
[user@linux ~]$
```

Download `python_basics.zip` into your home directory. Use `unzip` to extract the contents of the zip file:

```
[user@linux ~]$ ls *.zip
python_basics.zip
[user@linux ~]$ unzip python_basics.zip
[user@linux ~]$ cd python_basics
[user@linux ~/python_basics]$ ls
foreach.py
helloWorld.py
```

```
listcomp.py
listcomp2.py
quickSort.py
shop.py
shopTest.py
```

Some other useful Linux commands:

- `cp` copies a file or files
- `rm` removes (deletes) a file
- `mv` moves a file (i.e., cut/paste instead of copy/paste)
- `man` displays documentation for a command
- `pwd` prints your current path
- `xterm` opens a new terminal window
- `rmdir` allows you to delete specific folder(s) from your system
- `tar` for archiving files and extracting them
- `clear` to clear your current terminal
- `locate` for finding the location of a specific file
- `history` shows previously used commands
- Press “Ctrl-c” to kill a running process
- Append `&` to a command to run it in the background
- `fg` brings a program running in the background to the foreground

When typing a command, you can press the *Tab* button on your keyboard to autofill what you are typing. For example, let’s assume your current working directory contains a folder named “Documents” and you want to navigate to that folder. You can type `cd Documents` and press *Enter* on your keyboard like a peasant! Or you can type a portion of the name of the directory, for example, `cd Docu`, then hit the *Tab* key. The terminal will fill up the rest showing you `cd Documents`. Then you can press *Enter*.

2 IDE Workflow

To edit Python files, there are lots of [IDE and editors](#). We use [Visual Studio Code](#) for the demonstrations. You can use whichever you prefer. For VS Code, install the suggested extensions such as Python, PyLance, etc.

2.1 Opening the Workspace

Open your IDE and open the lab related zip folder through the GUI. This is how to do it:

- Download zip from the given link, drag and drop it into your folder for this lab (if any) and unzip the folder.
- Open your IDE in the folder.
 - ◊ Open terminal in the current folder and close the current folder. You can right click in your file manager and select `Open in Terminal`.

- ◇ Type the following in your terminal `your_editor .`, i.e., `code .` (run program aliased as `code` in the folder “.”, which is the current directory).

Note that, this is only for editors that can open folders, such as VS Code (`code .`) and IntelliJ (`pycharm .`) editors.

- ▷ If your editor has an integrated terminal like VS Code or PyCharm, you can close the terminal. Otherwise, you can `Alt-Tab` or `Cmd-Tab` to switch between windows.

2.2 Using the IDE

VS Code Linux shortcuts are noted, but other OS's and most IDE's should have the same/similar features, so you can look up how to do it in yours. You should try these out in the python basics or the tutorial; this can save you a lot of time in the future labs.

These are in rough order of usefulness:

- ▷ See descriptions of variables: `mouse hover` to see what a variable is, `Ctrl + mouse hover` to see string-doc or definition, `Ctrl + mouse click` to navigate to definition. Especially helpful for functions, although the IDE may not always figure out the definitions, for instance when PyLance in VS Code cannot figure out what class's method is getting called.
- ▷ Search in all files in lab: `Ctrl + Shift + F`. If you want to find what function `foo` does, search for `def foo`.
- ▷ Shortcut for hiding and showing the integrated terminal to free up the screen: `Ctrl + `` (the symbol to the left of 1).
- ▷ Debugger (needs a little bit of setup): Set breakpoints where you want them by clicking next to line number on the left, and run the debugger on the `autograder.py` file. You should open the `autograder` and press `F5`, select Python, make sure the correct Python Interpreter is in use, choose “current python file”.
- ▷ Auto format code: `Ctrl + Shift + I`.
- ▷ Search all filenames in the opened folder: `Ctrl + P`.
- ▷ Search things you can do in the IDE: `Ctrl + Shift + P`.

Appendix B Python Basics

This appendix provides a foundational introduction to Python, which is essential for completing the programming tasks in this course. Python is a versatile, interpreted, object-oriented language that shares features with Java and Scheme, making it both powerful and easy to learn. Whether you are new to Python or need a refresher, this tutorial will guide you through the primary syntactic constructions using concise examples. Engaging with this material actively will enhance your understanding and prepare you for the lab tasks ahead.

1 Required Files

To get started, download all the files associated with the Python mini-tutorial as a zip archive: `python_basics.zip`. If you have already completed the Linux Basics tutorial from Appendix A, you should have this file downloaded and unzipped.

For those who need a refresher or are new to Python, we strongly recommend completing this exercise. Even if you are confident in your Python skills, reviewing this appendix can be beneficial. You can also use the Table of Contents to navigate to specific topics of interest.

This tutorial includes step-by-step examples that illustrate Python's syntax and fundamental concepts. We encourage you to type the provided Python code on your own machine to ensure it functions as expected. This hands-on practice is crucial for solidifying your understanding.

If you encounter any issues, refer to the Troubleshooting section, which addresses common problems previous CSE 4712 students have faced while following this tutorial. This section is designed to help you overcome any obstacles and continue your learning smoothly.

By completing this Python basics tutorial, you will be well-equipped to tackle the lab tasks in this course.

2 Invoking the Interpreter

Python can be run in two primary modes: interactively via an interpreter or by executing a script from the command line. We will start by using the Python interpreter interactively, which is particularly useful for experimenting with code snippets and testing small pieces of code.

To invoke the Python interpreter, use the command `python` in the terminal. Here is an example of what you might see when you start the interpreter:

```
user@linux:~/python_basics$ python
Python 3.12.3 (main, May 28 2024, 05:37:50) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Once the interpreter is running, you can type Python commands directly at the “>>>” prompt and see immediate results. This interactive mode is a great way to explore Python's features and test simple code snippets quickly.

3 Operators

The Python interpreter can evaluate expressions, including simple arithmetic and Boolean expressions. When you enter such expressions at the prompt (>>>), they will be evaluated, and the result will be dis-

played on the next line.

You can perform basic arithmetic operations such as addition, multiplication, and more. Here are a few examples:

```
>>> 1 + 1
2
>>> 2 * 3
6
>>> 5 - 2
3
>>> 8 / 4
2.0
>>> 7 % 3
1
>>> 2 ** 3
8
```

Python also supports Boolean operators for manipulating the primitive True and False values. These operators include == (equal to), not, and, and or. Here are some examples:

```
>>> 1 == 0
False
>>> not (1 == 0)
True
>>> (2 == 2) and (2 == 3)
False
>>> (2 == 2) or (2 == 3)
True
```

Using these operators, you can construct more complex logical expressions, which are fundamental in controlling the flow of programs through conditional statements and loops.

4 Strings

Python has a built-in string type, similar to Java, with the '+' operator overloaded to concatenate string values.

```
>>> 'artificial' + "intelligence"
'artificialintelligence'
```

Python provides numerous built-in methods to manipulate strings:

```
>>> 'artificial'.upper()
'ARTIFICIAL'
>>> 'HELP'.lower()
'help'
>>> len('Help')
4
```

You can use either single quotes ‘’ or double quotes “” to define strings, allowing for easy nesting:

```
>>> "I accidentally swallowed some 'food coloring'"
"I accidentally swallowed some 'food coloring'"
>>> 'I have "dyed" a little inside.'
'I have "dyed" a little inside.'
```

You can also store expressions into variables:

```
>>> s = 'hello world'
>>> print(s)
hello world
>>> s.upper()
'HELLO WORLD'
>>> len(s.upper())
11
```

Variables do not need to be declared before assigning a value to them.

Exercise Learn about the methods Python provides for strings. To discover available methods, use the `dir` and `help` commands:

```
>>> s = 'abc'
>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattr__', '__getitem__',
'__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

Use `help` to learn more about a specific method:

```
>>> help(s.find)
"""
Help on built-in function find:

find(...) method of builtins.str instance
    S.find(sub[, start[, end]]) -> int

    Return the lowest index in S where substring sub is found,
    such that sub is contained within
```

```

        S[start:end]. Optional
        arguments start and end are interpreted as in slice notation.

        Return -1 on failure.
"""

>>> s.find('b')
1

```

You need to press q to exit a help screen.

Now, try out some of the string functions listed in `dir`. Ignore those with underscores `_` around the method name. Functions with underscores around their names are private helper methods.

5 Built-in Data Structures

Python comes equipped with several useful built-in data structures that are broadly similar to Java's collections package. These data structures are essential for organizing and manipulating data efficiently.

5.1 Lists

Lists in Python store a sequence of mutable items, allowing for various operations such as indexing, slicing, and concatenation.

You can create a list and access its elements using indices:

```

>>> fruits = ['apple', 'orange', 'pear', 'banana']
>>> fruits[0]
'apple'

```

Python allows negative indexing to access elements from the end of the list:

```

>>> fruits[-2]
'pear'
>>> fruits.pop()
'banana'
>>> fruits
['apple', 'orange', 'pear']
>>> fruits.append('grapefruit')
>>> fruits
['apple', 'orange', 'pear', 'grapefruit']
>>> fruits[-1] = 'pineapple'
>>> fruits
['apple', 'orange', 'pear', 'pineapple']

```

Use the `+` operator to concatenate lists:

```

>>> otherFruits = ['kiwi', 'strawberry']
>>> fruits + otherFruits
['apple', 'orange', 'pear', 'banana', 'kiwi', 'strawberry']

```

You can slice lists to access multiple adjacent elements. For instance, `fruits[1:3]`, returns a list containing the elements at position 1 and 2. In general `fruits[start:stop]` will get the elements in `start`, `start+1`, ..., `stop-1`. We can also do `fruits[start:]` which returns all elements starting from the `start` index. Also `fruits[:end]` will return all elements before the element at position `end`:

```
>>> fruits[0:2]
['apple', 'orange']
>>> fruits[:3]
['apple', 'orange', 'pear']
>>> fruits[2:]
['pear', 'pineapple']
>>> len(fruits)
4
```

Lists can contain any Python data type, including other lists:

```
>>> lstOfLsts = [['a', 'b', 'c'], [1, 2, 3], ['one', 'two', 'three']]
>>> lstOfLsts[1][2]
3
>>> lstOfLsts[0].pop()
'c'
>>> lstOfLsts
[['a', 'b'], [1, 2, 3], ['one', 'two', 'three']]
```

Play with some of the list functions. Use the `dir` command to find methods available for lists and the `help` command to get information about them:

```
>>> dir(list)
['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
 '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__getitem__', '__getstate__', '__gt__', '__hash__', '__iadd__',
 '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
 '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
 '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert',
 'pop', 'remove', 'reverse', 'sort']
```

For example, you can reverse a list in place:

```
>>> help(list.reverse)
Help on method_descriptor:

reverse(self, /) unbound builtins.list method
Reverse *IN PLACE*.
```

Here's how it works

```
>>> lst = ['a', 'b', 'c']
>>> lst.reverse()
>>> lst
['c', 'b', 'a']
```

5.2 Tuples

A data structure similar to the list is the tuple, which is immutable once created (i.e., you cannot change its content after creation). Tuples are surrounded by parentheses, while lists use square brackets.

You can create a tuple and access its elements using indices:

```
>>> pair = (3, 5)
>>> pair[0]
3
```

Tuples support unpacking, allowing you to assign their elements to multiple variables simultaneously:

```
>>> x, y = pair
>>> x
3
>>> y
5
```

Attempting to modify an element in a tuple will raise an exception because tuples are immutable:

```
>>> pair[1] = 6
TypeError: 'tuple' object does not support item assignment
```

This attempt to modify an immutable structure raised an exception. Exceptions indicate errors, such as index out of bounds errors, type errors, and so on. Python reports these errors using exceptions.

Tuples are useful for storing a collection of items that should not change throughout the program, providing a reliable way to group related data together.

5.3 Sets

A set is another data structure that serves as an unordered collection with no duplicate items. Sets are useful for membership testing, removing duplicates from a sequence, and performing mathematical set operations.

Note that the objects in the set are unordered; you cannot assume that their traversal or print order will be the same across different runs or machines.

You can create a set from a list, which will automatically remove any duplicate items:

```
>>> shapes = ['circle', 'square', 'triangle', 'circle']
>>> setOfShapes = set(shapes)
>>> setOfShapes
{'circle', 'square', 'triangle'}
```

Alternatively, you can create a set using curly braces:

```
>>> setOfShapes = {'circle', 'square', 'triangle', 'circle'}
>>> setOfShapes
{'circle', 'square', 'triangle'}
```

Add elements to a set using the add method:

```
>>> setOfShapes.add('polygon')
>>> setOfShapes
{'circle', 'square', 'triangle', 'polygon'}
```

Check if an item is in the set using the in keyword:

```
>>> 'circle' in setOfShapes
True
>>> 'rhombus' in setOfShapes
False
```

Perform common set operations such as difference, intersection, and union:

```
>>> favoriteShapes = ['circle', 'triangle', 'hexagon']
>>> setOfFavoriteShapes = set(favoriteShapes)

# Difference
>>> setOfShapes - setOfFavoriteShapes
{'square', 'polygon'}

# Intersection
>>> setOfShapes & setOfFavoriteShapes
{'circle', 'triangle'}

# Union
>>> setOfShapes | setOfFavoriteShapes
{'circle', 'square', 'triangle', 'polygon', 'hexagon'}
```

5.4 Dictionaries

The last built-in data structure we will cover is the dictionary. A dictionary stores a mapping from one type of object (the key) to another (the value). The key must be of an immutable type (e.g., string, number, or tuple), while the value can be any Python data type.

Here's how you can create a dictionary and access its elements:

```
>>> studentIds = {'knuth': 42.0, 'turing': 56.0, 'nash': 92.0}
>>> studentIds['turing']
56.0
```

You can modify the values in a dictionary or add new key-value pairs:

```
>>> studentIds['nash'] = 'ninety-two'
>>> studentIds
{'knuth': 42.0, 'turing': 56.0, 'nash': 'ninety-two'}
```

You can delete a key-value pair using the `del` keyword:

```
>>> del studentIds['knuth']
>>> studentIds
{'turing': 56.0, 'nash': 'ninety-two'}
```

Dictionaries can store complex data structures, including lists or other dictionaries:

```
>>> studentIds['knuth'] = [42.0, 'forty-two']
>>> studentIds
{'knuth': [42.0, 'forty-two'], 'turing': 56.0, 'nash': 'ninety-two'}
```

Dictionaries have several useful methods to access keys, values, and items:

```
>>> studentIds.keys()
dict_keys(['knuth', 'turing', 'nash'])

>>> studentIds.values()
dict_values([42.0, 'forty-two', 56.0, 'ninety-two'])

>>> studentIds.items()
dict_items([('knuth', [42.0, 'forty-two']), ('turing', 56.0), ('nash', 'ninety-two')])

>>> len(studentIds)
3
```

Note: The printed order of the keys in a dictionary may differ from what is shown above. This is because dictionaries are implemented as hash tables, which do not maintain any specific order for the keys. The order may appear arbitrary and can change based on the underlying hashing algorithm. Therefore, your code should not rely on any specific ordering of dictionary keys.

Similar to nested lists, you can create dictionaries of dictionaries to represent more complex data structures:

```
>>> courses = {
...     'CSE4307': {'instructor': 'ARMK', 'credits': 3},
...     'CSE4703': {'instructor': 'AZM', 'credits': 4}
... }
>>> courses['CSE4307']['instructor']
'ARMK'
```

Exercise Use the `dir` and `help` commands to explore the functions you can call on dictionaries:

6 Writing Scripts

Now that you have got a handle on using Python interactively, let's write a simple Python script that demonstrates Python's for loop. Open the file called `foreach.py`, which should contain the following code:

```
1 # This is what a comment looks like
2 fruits = ['apples', 'oranges', 'pears', 'bananas']
3 for fruit in fruits:
4     print(fruit + ' for sale')
5
6 fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}
7 for fruit, price in fruitPrices.items():
8     if price < 2.00:
9         print('%s cost %f a pound' % (fruit, price))
10    else:
11        print(fruit + ' are too expensive!')
```

At the command line, use the following command in the directory containing `foreach.py`:

```
user@linux:~/python_basics$ python foreach.py
```

You should see output similar to this:

```
apples for sale
oranges for sale
pears for sale
bananas for sale
apples are too expensive!
oranges cost 1.500000 a pound
pears cost 1.750000 a pound
```

Remember that the print statements listing the costs may be in a different order on your screen than in this tutorial. This is because we are looping over dictionary keys, which are unordered. To learn more about control structures (e.g., if and else) in Python, check out [the official Python tutorial section](#) on this topic. If you like functional programming, you might also like `map` and `filter`:

```
>>> list(map(lambda x: x * x, [1, 2, 3]))
[1, 4, 9]

>>> list(filter(lambda x: x > 3, [1, 2, 3, 4, 5, 4, 3, 2, 1]))
[4, 5, 4]
```

The next snippet of code demonstrates Python's list comprehension construction:

```
1 nums = [1, 2, 3, 4, 5, 6]
2
3 # Increment each number by 1
4 plusOneNums = [x + 1 for x in nums]
5
```

```

6 # Filter out odd numbers
7 oddNums = [x for x in nums if x % 2 == 1]
8 print(oddNums) # Output: [1, 3, 5]
9
10 # Increment odd numbers by 1
11 oddNumsPlusOne = [x + 1 for x in nums if x % 2 == 1]
12 print(oddNumsPlusOne) # Output: [2, 4, 6]

```

This code is in a file called `listcomp.py`, which you can run:

```
user@linux:~/python_basics$ python listcomp.py
```

You should see the following output:

```
[1, 3, 5]
[2, 4, 6]
```

Exercise Write a list comprehension which, from a list, generates a lowercased version of each string that has a length greater than five. You can find the solution in `listcomp2.py`.

7 Beware of Indentation!

Unlike many other languages, Python uses indentation in the source code for interpretation. This means that the level of indentation directly affects the program's logic and flow control. Consider the following script:

```

1 if 0 == 1:
2     print('We are in a world of arithmetic pain')
3 print('Thank you for playing')

```

This script will output:

```
Thank you for playing
```

But if we had written the script as:

```

1 if 0 == 1:
2     print('We are in a world of arithmetic pain')
3     print('Thank you for playing')

```

There would be no output. The first script prints "Thank you for playing" regardless of the condition because the second print statement is not indented and is outside the `if` block. In the second script, both print statements are inside the `if` block, and since the condition `0 == 1` is `False`, neither statement is executed.

Here is another example to highlight the importance of indentation:

```

1 for i in range(3):
2     print(i)
3     if i % 2 == 0:
4         print('Even number')
5     print('Loop iteration complete')
6 print('All iterations done')

```

The script will output:

```
0
Even number
Loop iteration complete
1
Loop iteration complete
```

```
2
Even number
Loop iteration complete
All iterations done
```

Notice how the indentation determines which block of code belongs to the `for` loop and the `if` statement. If you mistakenly alter the indentation, it can lead to logical errors in your program. For example:

```
1 for i in range(3):
2     print(i)
3 if i % 2 == 0:
4     print('Even number')
5 print('Loop iteration complete')
6 print('All iterations done')
```

This altered script will output:

```
0
1
2
Even number
Loop iteration complete
All iterations done
```

Here, the `if` statement and the subsequent `print` statements are outside the `for` loop, leading to different behavior.

8 Tabs Vs Spaces

Note that, because Python uses indentation for code evaluation, it needs to keep track of the level of indentation across code blocks. This means that if your Python file switches from using tabs as indentation to spaces as indentation, the Python interpreter will not be able to resolve the ambiguity of the indentation level and will throw an exception. Even though the code can be lined up visually in your text editor, Python “sees” a change in indentation and most likely will throw an exception (or rarely, produce unexpected behavior).

This most commonly happens when opening up a Python file that uses an indentation scheme that is opposite from what your text editor uses (e.g., your text editor uses spaces and the file uses tabs). When you write new lines in a code block, there will be a mix of tabs and spaces, even though the whitespace is aligned. For a longer discussion on tabs vs spaces, see [this discussion on StackOverflow](#).

Always be mindful of your indentation in Python to avoid subtle and hard-to-find bugs. Four spaces per indentation level is a widely accepted standard that helps maintain readability and consistency in your code, which is what we use.

9 Writing Functions

As in Java, in Python you can define your own functions to encapsulate reusable blocks of code. Here's an example:

```
1 fruitPrices = {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}
2
3 def buyFruit(fruit, numPounds):
4     if fruit not in fruitPrices:
5         print("Sorry we don't have %s" % (fruit))
6     else:
7         cost = fruitPrices[fruit] * numPounds
8         print("That'll be %f please" % (cost))
9
10 # Main Function
11 if __name__ == '__main__':
12     buyFruit('apples', 2.4)
13     buyFruit('coconuts', 2)
```

In this script, we define a dictionary `fruitPrices` that maps fruit names to their prices per pound. The function `buyFruit` takes two parameters: the name of the fruit and the number of pounds to buy. It checks if the fruit is available in the `fruitPrices` dictionary and, if so, calculates the cost and prints it. If the fruit is not available, it prints an appropriate message.

Notice that, rather than having a main function as in Java, the `__name__ == '__main__'` check is used to delimit expressions which are executed when the file is called as a script from the command line. The code after the main check is thus the same sort of code you would put in a main function in Java.

Save this script as `fruit.py` and run it:

```
user@linux: ~/python_basics$ python fruit.py
That'll be 4.800000 please
Sorry we don't have coconuts
```

The output demonstrates that the script calculates the cost for apples correctly and informs us that coconuts are not available.

Advanced Exercise For those looking for a challenge, try writing a `QuickSort` function in Python using list comprehensions. Use the first element as the pivot. `QuickSort` is a popular sorting algorithm that works by partitioning an array into two halves, sorting the halves independently, and then combining the results. You can find the solution in `quickSort.py` file.

10 Object Basics

Although this is not a class in object-oriented programming, you will have to use some objects in the labs, and so it is worth covering the basics of objects in Python. An object encapsulates data and provides func-

tions for interacting with that data.

10.1 Defining Classes

Here is an example of defining a class named `FruitShop`:

```

1 class FruitShop:
2
3     def __init__(self, name, fruitPrices):
4         """
5         Initializes a FruitShop instance.
6
7         Parameters:
8         name (str): The name of the fruit shop.
9         fruitPrices (dict): Dictionary with keys as fruit names (str)
10        and prices per pound (float) as values.
11        e.g. {'apples': 2.00, 'oranges': 1.50, 'pears': 1.75}
12        """
13        self.fruitPrices = fruitPrices
14        self.name = name
15        print(f'Welcome to {name} fruit shop')
16
17    def getCostPerPound(self, fruit):
18        """
19        Returns the cost per pound of a given fruit.
20
21        Parameters:
22        fruit (str): The name of the fruit.
23
24        Returns:
25        float: The price per pound of the fruit, or None if the fruit is
26              not available.
27        """
28        return self.fruitPrices.get(fruit, None)
29
30    def getPriceOfOrder(self, orderList):
31        """
32        Calculates the total cost of an order.
33
34        Parameters:
35        orderList (list of tuples): List of (fruit, numPounds) tuples.
36
37        Returns:
38        float: The total cost of the order.
39        """

```

```

39         totalCost = 0.0
40         for fruit, numPounds in orderList:
41             costPerPound = self.getCostPerPound(fruit)
42             if costPerPound is not None:
43                 totalCost += numPounds * costPerPound
44         return totalCost
45
46     def getName(self):
47         """
48         Returns the name of the fruit shop.
49
50         Returns:
51         str: The name of the fruit shop.
52         """
53         return self.name

```

The FruitShop class has some data, such as the name of the shop and the prices per pound of some fruits, and it provides methods to interact with this data. What advantage is there to wrapping this data in a class?

- Encapsulating the data prevents it from being altered or used inappropriately.
- The abstraction that objects provide makes it easier to write general-purpose code.

10.2 Using Objects

So how do we create and use an object? Make sure you have the FruitShop implementation in `shop.py`. We then import the code from this file, making it accessible to other scripts using `import shop`, since `shop.py` is the name of the file. Then, we can create FruitShop objects as follows:

```

1 import shop
2
3 shopName = 'Raju Bazar'
4 fruitPrices = {'apples': 1.00, 'oranges': 1.50, 'pears': 1.75}
5 berkeleyShop = shop.FruitShop(shopName, fruitPrices)
6 applePrice = berkeleyShop.getCostPerPound('apples')
7 print(applePrice)
8 print('Apples cost $%.2f at %s.' % (applePrice, shopName))
9
10 otherName = 'Khoab'
11 otherFruitPrices = {'kiwis': 6.00, 'apples': 4.50, 'peaches': 8.75}
12 otherFruitShop = shop.FruitShop(otherName, otherFruitPrices)
13 otherPrice = otherFruitShop.getCostPerPound('apples')
14 print(otherPrice)
15 print('Apples cost $%.2f at %s.' % (otherPrice, otherName))
16 print("My, that's expensive!")

```

This code is in `shopTest.py`; you can run it like this:

```

user@linux: ~/python_basics$ python shopTest.py
Welcome to Raju Bazar fruit shop
1.0
Apples cost $1.00 at Raju Bazar.
Welcome to Khoab fruit shop
4.5

```

```

Apples cost $4.50 at Khoab.
My, that's expensive!

```

So what just happened? The `import shop` statement told Python to load all of the functions and classes in `shop.py`. The line `rajuShop = shop.FruitShop(shopName, fruitPrices)` constructs an instance of the `FruitShop` class defined in `shop.py`, by calling the `__init__` function in that class. Note that we only passed two arguments in, while `__init__` seems to take three arguments: (`self`, `name`, `fruitPrices`). The reason for this is that all methods in a class have `self` as the first argument. The `self` variable's value is automatically set to the object itself; when calling a method, you only supply the remaining arguments. The `self` variable contains all the data (`name` and `fruitPrices`) for the current specific instance (similar to this in Java). The print statements use the substitution operator (described in the [Python docs](#) if you are curious).

10.3 Static vs Instance Variables

The following example illustrates how to use static and instance variables in Python. Create the `person_class.py` containing the following code:

```

1 class Person:
2     population = 0
3
4     def __init__(self, myAge):
5         self.age = myAge
6         Person.population += 1
7
8     def get_population(self):
9         return Person.population
10
11    def get_age(self):
12        return self.age

```

We first compile the script:

```

user@linux ~/python_basics$ python person_class.py

```

Now, use the class as follows:


```
>>> import person_class
>>> p1 = person_class.Person(12)
>>> p1.get_population()
1
>>> p2 = person_class.Person(63)
>>> p1.get_population()
2
>>> p2.get_population()
2
>>> p1.get_age()
12
```

```
>>> p2.get_age()
63
```

In the code above, age is an instance variable and population is a static variable. population is shared by all instances of the Person class whereas each instance has its own age variable.

11 More Python Tips and Tricks

This tutorial has briefly touched on some major aspects of Python that will be relevant to the course. Here are some more useful tidbits:

- Use range to generate a sequence of integers, useful for generating traditional indexed for loops:

```
1 for index in range(3):
2     print(index)
```

- After importing a file, if you edit a source file, the changes will not be immediately propagated in the interpreter. For this, use the reload command:

```
>>> reload(shop)
```

- Swap two variables with one line of code.

```
>>> a = 7
>>> b = 5
>>> b, a = a, b
>>> a
5
>>> b
7
```

- Assign multiple values at the same time:

```
>>> a = [1, 2, 3]
>>> x, y, z = a
>>> x
1
>>> y
2
>>> z
3
```

- The place to go for more Python information: www.python.org
- A good reference book: [Learning Python](#)

12 Troubleshooting

These are some problems (and their solutions) that students commonly encounter:

- **Problem:** ImportError: No module named py

Solution:

When using import, do not include the “.py” from the filename.

For example, you should say: `import shop`

NOT: `import shop.py`

- **Problem:** NameError: name ‘MY VARIABLE’ is not defined. Even after importing you may see this.

Solution:

To access a member of a module, you have to type `MODULE NAME.MEMBER NAME`, where `MODULE NAME` is the name of the .py file, and `MEMBER NAME` is the name of the variable (or function) you are trying to access. Another reason for this problem would be using due to scope issues. It is best to stick with local variables within the code block. If you want to utilize certain variables throughout the program, use the global variable format.

- **Problem:** TypeError: ‘dict’ object is not callable

Solution:

Dictionary look ups are done using square brackets: [and]. NOT parenthesis: (and).

- **Problem:** ValueError: too many values to unpack

Solution:

Make sure the number of variables you are assigning in a for loop matches the number of elements in each item of the list. Similarly for working with tuples.

For example, if `pair` is a tuple of two elements (e.g `pair = ('apple', 2.0)`) then the following code would cause the “too many values to unpack error”:

```
1 (a, b, c) = pair
```

Here is a problematic scenario involving a for loop:

```
1 pairList = [('apples', 2.00), ('oranges', 1.50), ('pears', 1.75)]
2 for fruit, price, color in pairList:
3     print('%s fruit costs %f and is the color %s' % (fruit, price,
        color))
```

- **Problem:** `AttributeError: 'list' object has no attribute 'length'` (or something similar)

Solution:

Finding length of lists is done using `len(NAME OF LIST)`.

- **Problem:** Changes to a file are not taking effect.

Solution:

Make sure you are saving all your files after any changes. If you are editing a file in a window different from the one you are using to execute python, make sure you `reload(YOUR_MODULE)` to guarantee your changes are being reflected. `reload` works similarly to `import`.

- **Problem:** `AttributeError: module 'cgi' has no attribute 'escape'`

Solution:

You are probably using the wrong version of Python. Make sure that you are in the correct conda environment.