CSCI3230 / ESTR3108: Fundamentals of Artificial Intelligence 2023-2024 Term 1
The Chinese University of Hong Kong

Assignment 4

Due date: 10 Dec 2023 (Sun) 23:59 Full mark: 100
Expected time spent: 8-10 hours

Aims: 1. Understand knowledge about neural networks and hands-on programming of convolutional neural networks, including training and testing.
2. Enhance the understanding of uninformed and informed search, and hands-on programming of search algorithms such as DFS, BFS, UCS and A*.

Assignment 4 has two programming parts. The first part is about CNN, the second part is about search.

- For Part 1, please put all your plots and answers into a <ID>_part1.pdf file.
- For Part 2, please submit your code search.py, by renaming the file as <ID>_search.py
  Our TA will run your submitted code with his testing script for assessment.

Overall, please put <ID>_part1.pdf and <ID>_part2_search.py into the zipped folder <ID>_asmt4.zip and submit the overall .zip file through Blackboard.

Description Part 1:

In the first part, you will practice on essentials of convolutional neural networks, including network building, improving model training, and network evaluation, using PyTorch.

Questions:

1. In tutorial 7, we have introduced the dataset of MNIST and visualized some samples from it. We also set up a simple training framework to learn a MLP classifier. In this question, we will try to practice the training of networks on a more complex image dataset, i.e., CIFAR10. (50%)

   We provide a codebase (part1_codebase.py) for this question with which you can revise and complete the missing parts as required. To efficiently train the network, you are suggested to use GPU. If you do not have GPU card on your own PC, you can use the free GPUs from Google Colab. Here is a Google Colab CIFAR10 tutorial on how to use it.

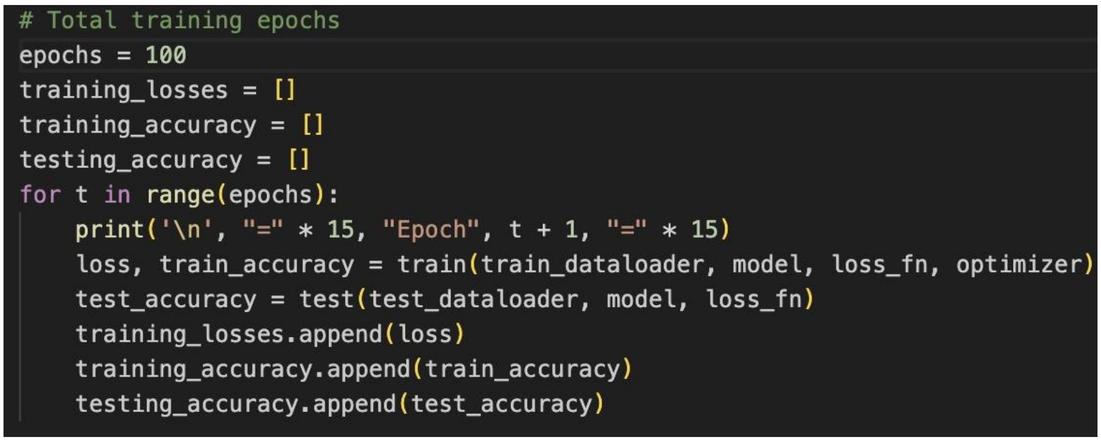
   To submit the answers for this question, you don't need to submit the actual code file. Please screenshoot your lines of key code that you revised/added to the provided codebase, and show the figure of curves that is output from your model.

   a) In tutorial 7, it is seen that if we only train a MLP for just a few epochs, the network can not converge well. So a solution for improving the performance of the classifier is to train for more epochs. Please train the fully-connected network provided in codebase for 100 epochs and present the followings:

      i) plot the curves of training loss;
      ii) plot the curves of training accuracy and test accuracy;
      iii) report the best test accuracy that you can achieve.

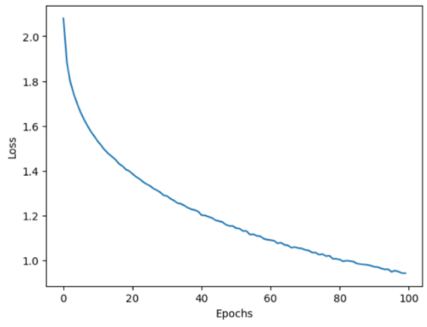
      (Hint: With proper implementation, the best test accuracy is expected to be larger than 50%).

Solution:

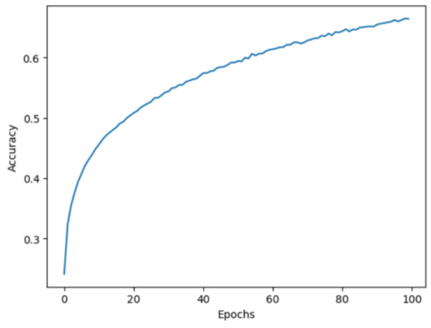


```
# Total training epochs
epochs = 100
training_losses = []
training_accuracy = []
testing_accuracy = []
for t in range(epochs):
    print('\n', "=" * 15, "Epoch", t + 1, "=" * 15)
    loss, train_accuracy = train(train_dataloader, model, loss_fn, optimizer)
    test_accuracy = test(test_dataloader, model, loss_fn)
    training_losses.append(loss)
    training_accuracy.append(train_accuracy)
    testing_accuracy.append(test_accuracy)
```

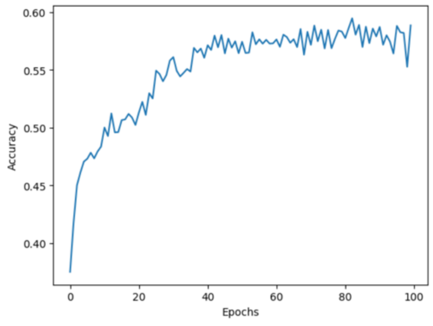
Training loss:



Training accuracy:



Test accuracy:



The best test accuracy can vary for different seeds. It is expected to be larger than 50%.

(10%)

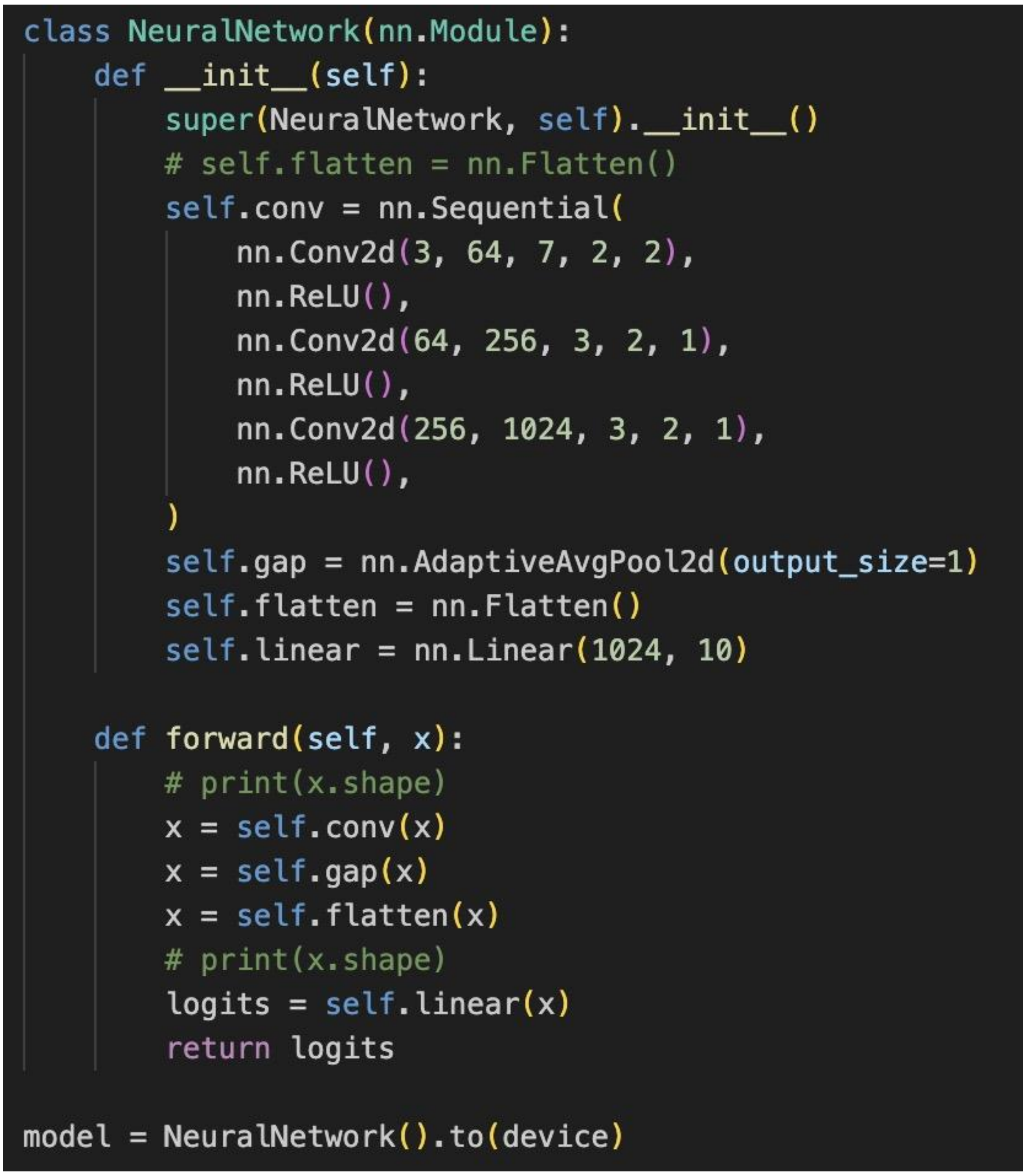
b) Further, we can make the neural network deeper for improving the performance. In tutorial 9, we just build a neural network composed with 3 linear layers, which is rather simple for image classification task. Here, you can build a CNN consisting of 3 convolutional layers following with ReLU activation function, 1 pooling layer, 1 flatten layer and 1 linear layer (*Please do not add/delete any other layers for this sub-question*). Please design your network, train your network for 100 epochs and present the followings:

i) plot the curves of training loss;
ii) plot the curves of training accuracy and test accuracy;
iii) report the best test accuracy that you can achieve.

(Hint: The best test accuracy is expected to be larger than 60%).

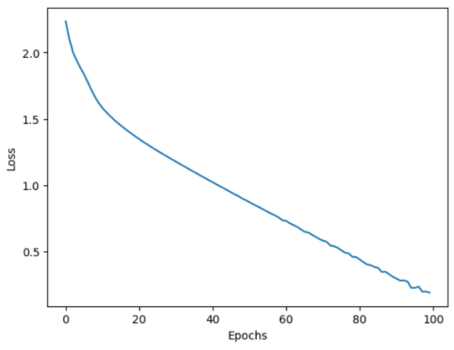
(15%)

Solution:

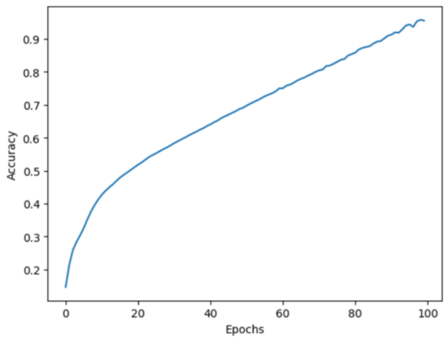


```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        # self.flatten = nn.Flatten()
        self.conv = nn.Sequential(
            nn.Conv2d(3, 64, 7, 2, 2),
            nn.ReLU(),
            nn.Conv2d(64, 256, 3, 2, 1),
            nn.ReLU(),
            nn.Conv2d(256, 1024, 3, 2, 1),
            nn.ReLU(),
        )
        self.gap = nn.AdaptiveAvgPool2d(output_size=1)
        self.flatten = nn.Flatten()
        self.linear = nn.Linear(1024, 10)

    def forward(self, x):
        # print(x.shape)
        x = self.conv(x)
        x = self.gap(x)
        x = self.flatten(x)
        # print(x.shape)
        logits = self.linear(x)
        return logits

model = NeuralNetwork().to(device)
```

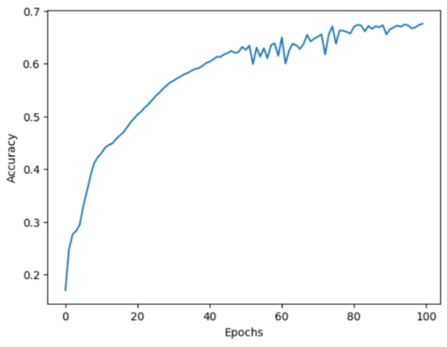
Training loss:



Training accuracy:



Test accuracy:



Best test accuracy: 71.65%

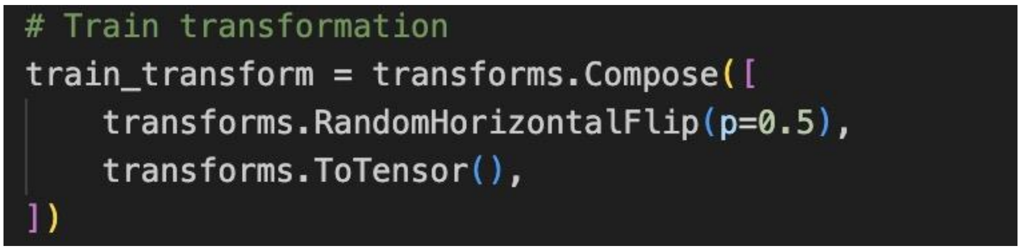
c) Based on CNN architecture, you can further use data augmention tricks to enrich training data, such as RandomCrop(32, padding=4). More operations for data augment in PyTorch are here. Based on the CNN architecture in (b), retrain your network (with data augmentations) for 100 epochs and present the followings:

i) plot the curves of training loss;
ii) plot the curves of training accuracy and test accuracy;
iii) report the best test accuracy that you can achieve.

(Hint: The best test accuracy is expected to be larger than 65%).

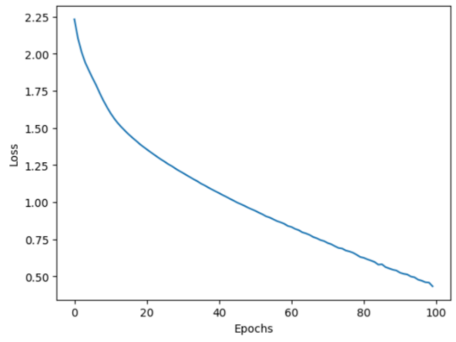
(10%)

Solution:

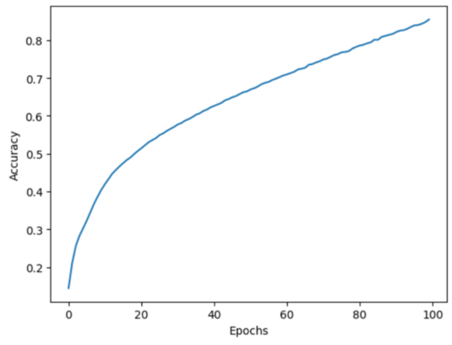


```
# Train transformation
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.ToTensor(),
])
```

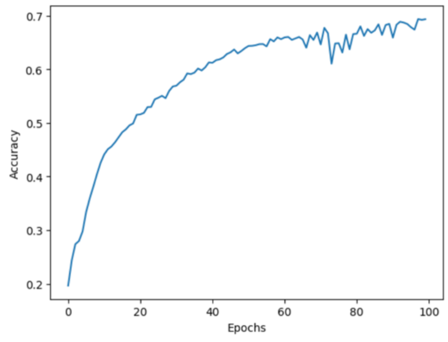
Training loss:



Training accuracy:



Test accuracy:



Best test accuracy: 74.23%

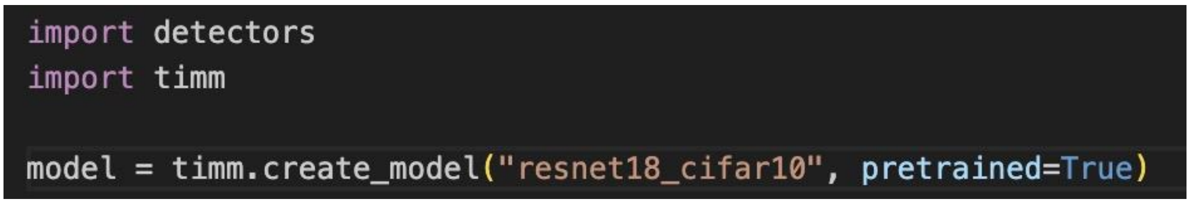
d) Congratulations, you have successfully built up a CNN model for CIFAR10 classification. Now, it is time for brainstorms! You can use any reasonable techniques to improve the model performance, such as using deeper neural networks, transferring pre-trained parameters, exploring other optimizers and so on. Try your best to improve the test accuracy and present the followings:
  i) State what specific tricks you are using, and show screenshot of your code corresponding to your added training tricks.
  ii) plot the curves of training loss and testing loss in a single figure;

iii) plot the curve of training accuracy and testing accuracy in a single figure;
iv) report the best test accuracy you can achieve.
(Hint: The expected best test accuracy should be larger than 80% to get full marks here. An extra bonus mark of 10% will be awarded if achieving an accuracy of larger than 90%).

(15%)

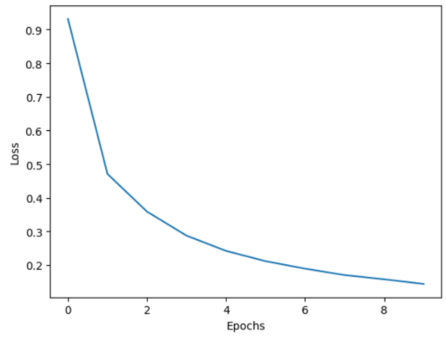
Solution:

ResNet18 with pretrained ImageNet weight

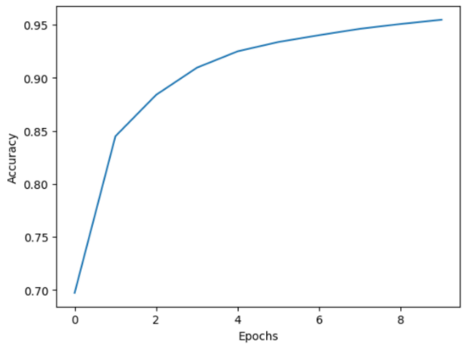


```python
import detectors
import timm

model = timm.create_model("resnet18_cifar10", pretrained=True)
```

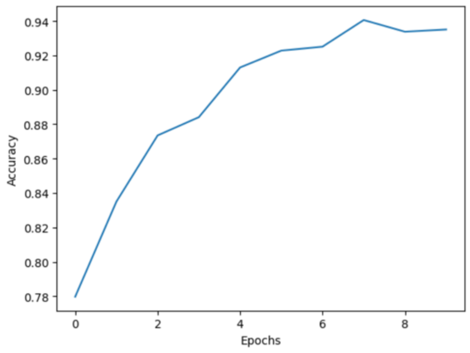
Training loss:



Training accuracy:



Test accuracy:

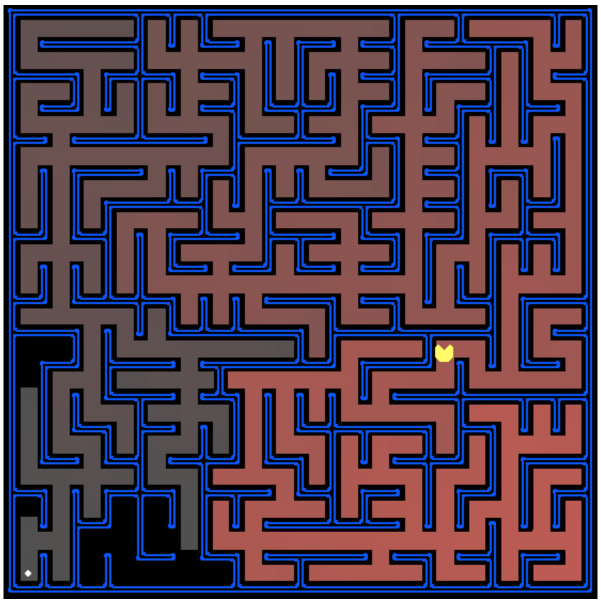


Best test accuracy: 94.51%

Description Part 2:

In the second part, you will conduct hands-on programming of search algorithms. We adopt the Pac-Man project from CS 188 (UC Berkeley), Project 1 - Search - CS 188: Introduction to Artificial Intelligence, Summer 2020 (berkeley.edu). In this project, your Pac-Man agent will find paths through the maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pac-Man scenarios. (50%)

To complete the assignment, you need to read the following description carefully. If you have any difficulty in writing searching algorithms, you can refer to the code in tutorial 10 and 11 slides.



File Structure:

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment. You only need to fill in the search algorithm in search.py. In search.py, there are four different search algorithms you need to fill in. We list some APIs that might be helpful for you in the notes of every search algorithm.

Files you will edit:	
search.py	Where all of your search algorithms will reside.

As this assignment only involves part of the questions from original Pac-Man projects. Files except search.py, should keep unchanged. You can find some useful APIs and search agent structure or game structure in the files below.

Files you might want to look at:	
searchAgents.py	Where all of your search-based agents will reside.
pacman.py	The main file that runs Pac-Man games. This file describes a Pac-Man GameState type, which you use in this project.
Game.py	The logic behind how the Pac-Man world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.

Util.py	Useful data structures for implementing search algorithms.

The followings are supporting files.

Supporting files you can ignore in this assignment:	
graphicsDisplay.py	Graphics for Pacman
graphicsUtils.py	Support for Pacman graphics
textDisplay.py	ASCII graphics for Pacman
ghostAgents.py	Agents to control ghosts
keyboardAgents.py	Keyboard interfaces to control Pacman
layout.py	Code for reading layout files and storing their contents
autograder.py	Project autograder
testParser.py	Parses autograder test and solution files
testClasses.py	General autograding test classes
test_cases/	Directory containing the test cases for each question
searchTestClasses.py	Project 1 specific autograding test classes

Introduction:

After downloading the code (search.zip), unzipping it, and changing to the directory, you should be able to play the game of Pac-Man by typing the following at the command line:

```
python pacman.py
```

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain. The simplest agent in searchAgents.py is called the GoWestAgent, which always goes West (a trivial reflex agent). This agent can occasionally win:

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

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

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

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:

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

Also, all of the commands that appear in this project also appear in commands.txt, for easy copying and pasting. The commands for autograder is also in commands.txt.

Questions:

1. Finding a Fixed Food Dot using Depth First Search:

   In searchAgents.py, you'll find a fully implemented SearchAgent, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan in search.py are not implemented – that's your job. First, test that the SearchAgent is working correctly by running:

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

   The command above tells the SearchAgent to use tinyMazeSearch as its search algorithm, which is implemented in search.py. Pacman should navigate the maze successfully.

   Now it is time to write full-fledged generic search functions to help Pacman plan routes! Pseudocode for the search algorithms you'll write could be found in the tutorial slides. You can refer to them. Remember that a search node can contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

   Important note: All of your search functions need to return a list of actions that will lead the agent from the start to the goal. These actions all have to be legal moves (valid directions, no moving through walls). For this assignment, you will use the PositionSearchProblem in searchAgents.py, where actions already avoid illegal movies.

   Important note: Make sure to use the Stack, Queue and PriorityQueue data structures provided to you in util.py! These data structure implementations have particular properties which are required for compatibility with the autograder.

   Hint: Each algorithm is very similar. Algorithms for DFS, BFS, UCS, and A* differ only in the details of how the fringe is managed. So, concentrate on getting DFS right and the rest should be relatively straightforward. Indeed, one possible implementation requires only a single generic search method which is configured with an algorithm-specific queuing strategy.
   (Note: your implementation need not be of this form to receive full credit).

   Implement the depth-first search (DFS) algorithm in the depthFirstSearch function in search.py. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

   In this assignment, you need to provide **two** depth first search algorithm (recursive and iterative). You can write them in the same function with one of them commented out. For recursive algorithm, you can create a new function inside depthFirstSearch. Your code should pass the autograder and quickly find a solution for:

   ```
   python pacman.py -l tinyMaze -p SearchAgent

   python pacman.py -l mediumMaze -p SearchAgent

   python pacman.py -l bigMaze -z .5 -p SearchAgent
   ```

The Pacman board will show an overlay of the states explored, and the order in which they were explored (brighter red means earlier exploration). Is the exploration order what you would have expected? Does Pacman actually go to all the explored squares on his way to the goal?

Hint: If you use a Stack as your data structure, the solution found by your non-recursive 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, think about what we learned in tutorial!).

(20%)

Solution:

Iterative function: Pass all commands (5%) Pass autograder(5%)
Recursive function: Pass all commands (5%) Pass autograder(5%)

Note:
Failing 1 maze command deducts 2 marks, 5 marks at most.
Failing 1 test case deducts 1 mark, 5 marks at most.
No mark will be deducted if your iterative DFS can't get 130 in the medium maze.

2. Breadth First Search:

Implement the breadth-first search (BFS) algorithm in the breadthFirstSearch function in search.py. Again, write a search algorithm that avoids expanding any already visited states. Your code should pass the autograder and quickly find a solution for:

python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

Does BFS find a least cost solution? If not, check your implementation.
Hint: If Pacman moves too slowly for you, try the option --frameTime 0.
Note: If you've written your search code generically, your code should work equally well for the eight-puzzle search problem without any changes.

python eightpuzzle.py

(10%)

Solution:

BFS: Pass all commands (5%) Pass autograder(5%)

Note:
Failing 1 maze command deducts 2 marks, 5 marks at most.
Failing 1 test case deducts 1 mark, 5 marks at most.

3. 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. Your code should pass the autograder. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use (the agents and cost functions are written for you):

   python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

   python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

   python pacman.py -l mediumScaryMaze -p StayWestSearchAgent

   Note: You should get very low and very high path costs for the StayEastSearchAgent and StayWestSearchAgent respectively, due to their exponential cost functions (see searchAgents.py for details)

   (10%)

   Solution:
   UCS: Pass all commands (5%) Pass autograder(5%)

   Note:
   Failing 1 maze command deducts 2 marks, 5 marks at most.
   Failing 1 test case deducts 1 mark, 5 marks at most.

4. A* search:
   Implement A* graph search in the empty function aStarSearch in search.py. A* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The nullHeuristic heuristic function in search.py is a trivial example.

   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). Your code should pass the autograder and quickly find a solution for:

   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).

(10%)

Solution:

A*: Pass all commands (5%) Pass autograder(5%)

Note:
Failing 1 maze command deducts 5 marks.
Failing 1 test case deducts 1 mark, 5 marks at most.

Submission:

Submit the zipped folder <ID>_asmt4.zip which includes <ID>_part1.pdf and <ID>_part2_search.py where <ID> is your student ID. Your file should contain the following header (put it on the head of file and comment out). Contact Professor Dou before submitting the assignment if you have anything unclear about the guidelines on academic honesty.

```
CSCI3230 / ESTR3108 2023-24 First Term Assignment 4

I declare that the assignment here submitted is original except for source
material explicitly acknowledged, and that the same or closely related material
has not been previously submitted for another course. I also acknowledge that I
am aware of University policy and regulations on honesty in academic work, and
of the disciplinary guidelines and procedures applicable to breaches of such
policy and regulations, as contained in the following websites.

University Guideline on Academic Honesty:
http://www.cuhk.edu.hk/policy/academichonesty/
Faculty of Engineering Guidelines to Academic Honesty:
http://www.erg.cuhk.edu.hk/erg-intra/upload/documents/ENGG_Discipline.pdf

Student Name: <fill in your name>
Student ID  : <fill in your ID>
```

Submit your files using the Blackboard online system.

Notes:

1. Remember to submit your assignment by 23:59pm of the due date. We may not accept late submissions.
2. If you submit multiple times, **ONLY** the content and time-stamp of the **latest** one would be considered.

University Guideline for Plagiarism

Please pay attention to the university policy and regulations on honesty in academic work, and the disciplinary guidelines and procedures applicable to breaches of such policy and regulations. Details can be found at http://www.cuhk.edu.hk/policy/academichonesty/. With each assignment, students will be required to submit a statement that they are aware of these policies, regulations, guidelines and procedures.