Assignment 1

(Last Update: 28 Feb)

Introduction

Download the code for this assignment here and then unzip the archive.

This assignment uses python 3. Do not use python 2.

You can work on the assignment using your favourite python editor. We recommend VSCode.

Post any questions or issues with this assignment to our discussion forum. Alternatively you may also contact your TA directly.

Search Assignment

Problem 1: DFS-GSA

In this part of the assignment you are going to implement

- a parser to read a search problem in the file `parse.py`, and
- Depth First Search (DFS) - Graph Search Algorithm (GSA) in the file `p1.py`

Both these python files have already been created for. Do not change anything that has already been implemented. Our autograder relies on the existing code. From the terminal you may run `python p1.py 1` to test if your code passes the first test case.

```
yiqun@Yiquns-MacBook-Pro a1 % python3 p1.py 1
Grading Problem 1 :
----------> Test case 1 PASSED <----------
```

As you can see we have passed test case 1 here. This is because we have hardcoded the solution for you in this function.

```
def dfs_search(problem):
    #Your p1 code here
    solution = 'Ar D C\nAr C G'
    return solution
```

Note that this is exactly what the test case expects. To find out what the test case expects, you can open the file `test_cases/p1/1.sol`.

```
Ar B C
Ar C G
```

As you can see, the content of the file `test_cases/p1/1.sol` is identical to the return value of the function `dfs_search(problem)`.

Note that `test_cases/p1/1.sol` and other solution files consist of two lines of text that represent the following.

- Exploration order (i.e., the order in which states are added to the explored set)
- solution path (i.e., the first solution found)

You should return these two lines of text in the dfs_search() function above.

The exploration order is `Ar`, `B`, `C` and the solution path is `Ar`, `C`, `G`. The search problem definition can be found in the file `test_cases/p1/1.prob`, which we will introduce shortly.

Let's try another test case by changing the argument to the python program to `2`.

```
yiqun@Yiquns-MacBook-Pro a1 % python3 p1.py 2
Grading Problem 1 :
----------> Test case 2 FAILED <----------
Your solution
Ar B C
Ar C G
Correct solution
A B D
A B D G
Delete "r" from position 1
Delete "C" from position 5
Add "D" to position 6
Delete "r" from position 9
Delete "C" from position 11
Add "B" to position 12
Add " " to position 13
Add "D" to position 14
```

We failed this test case. The correct solution can be found in `test_cases/p1/2.sol.`

```
A B D
A B D G
```

We will have to look at the `*.prob` files in the `test_case/p1/` folder to load the problem definition and then determine a corresponding solution. The `*.prob` files define weighted directed graphs with a single start state, a list of goal states, heuristics and arcs. For example, consider the problem definition of the first test case defined in the file `test_cases/p1/1.prob`.

```
start_state: Ar
goal_states: G
Ar 0
B 0
C 0
```

```
D 0
G 0
Ar B 1.0
Ar C 2.0
Ar D 4.0
C G 8.0
```

The search problem is specified as follows.
line 1: start state
line 2: list of goal states separated by a space
line 3 … (n+2): (n = number of states) heuristic for each state
`<state> <heuristic>`
line (n+3) … end: directional state transitions of the form
`<start state> <end state> <cost>`

You should implement the parsing for everything now so that you don't have to make modifications later.

You should decide on an appropriate data structure for the problem and return it from the following function in `parse.py`.

```
def read_graph_search_problem(file_path):
     #Your p1 code here
     problem = ''
     return problem
```

Once your implementation of both `read_graph_search_problem` and `dfs_search` is complete you can verify that you pass the first test case as follows.

```
yiqun@Yiquns-MacBook-Pro a1 % python3 p1.py 1
Grading Problem 1 :
----------> Test case 1 PASSED <----------
```

You may check if you pass all 5 test cases as follows.

```
yiqun@Yiquns-MacBook-Pro a1 % python3 p1.py -5
Grading Problem 1 :
----------> Test case 1 PASSED <----------
----------> Test case 2 PASSED <----------
----------> Test case 3 PASSED <----------
----------> Test case 4 PASSED <----------
----------> Test case 5 PASSED <----------
```

Note that you should visit the nodes in the order "always explore the child with the lowest cost edge first". This is in contrast to adding nodes alphabetically which is what we did in the lecture (Also different from the greedy search in the lecture).

Let’s take the following graph as an example:
A B 1.0
A G 2.0
A D 4.0
When you are processing the edges starting from A, the orders to visit these nodes should be B, G, D

You can use the ‘sorted’ function in python on ‘tuple’ as follows:

```
nodes = [(1, "B"), (2, "G"), (4, "D")]
sorted_nodes = sorted(nodes, key=lambda x: x[0])
```

Then you can add the nodes one by one following the order in sorted_nodes.

You may import anything from the Python Standard Library.

Do not import packages that are not included in Python such as numpy.

Make sure that you pass all provided test cases before moving on to the next question.

Note that we may use novel test cases for marking. You can also design your own new test cases for testing. For our novel test cases you may assume that there is always a single start state and at least one reachable goal state. For multi-goal cases, your program is supposed to return the first one that is reached along the path.

You are allowed to copy any code from the lecture slides as a starting point to solve this assignment.

Marking of this and all other questions will be done manually. We won’t solely rely on the autograder.

Problem 2: BFS-GSA

In this part of the assignment you are going to implement Breadth First Search (BFS) - Graph Search Algorithm (GSA) in the file `p2.py`.

This should just involve copying over your DFS implementation from p1 and changing the visiting strategy accordingly.

Once you have done this, check if you pass all test cases as follows.

```
yiqun@Yiquns-MacBook-Pro a1 % python3 p2.py -5
Grading Problem 2 :
----------> Test case 1 PASSED <----------
----------> Test case 2 PASSED <----------
----------> Test case 3 PASSED <----------
----------> Test case 4 PASSED <----------
----------> Test case 5 PASSED <----------
```

Problem 3: Greedy

In this part of the assignment you are going to implement Greedy Search - Graph Search Algorithm (GSA) in the file `p3.py`.
"**For paths in the frontier that have the same heuristic value, you should prioritize and pop the one with the lowest alphabetical order first**."
Once you have done this, check if you pass test cases as follows.

```
yiqun@Yiquns-MacBook-Pro a1 % python3 p3.py -6
Grading Problem 3 :
----------> Test case 1 PASSED <----------
----------> Test case 2 PASSED <----------
----------> Test case 3 PASSED <----------
----------> Test case 4 PASSED <----------
----------> Test case 5 PASSED <----------
----------> Test case 6 PASSED <----------
```

Problem 4: A*

In this part of the assignment you are going to implement A* - Graph Search Algorithm (GSA) in the file `p4.py`.

This should involve copying over your Greedy implementation from p4 and adding the backward cost.

Once you have done this, check if you pass test cases as follows.

```
yiqun@Yiquns-MacBook-Pro a1 % python p4.py -6
Grading Problem 4 :
----------> Test case 1 PASSED <----------
----------> Test case 2 PASSED <----------
----------> Test case 3 PASSED <----------
----------> Test case 4 PASSED <----------
----------> Test case 5 PASSED <----------
----------> Test case 6 PASSED <----------
```

You may also use the following command to make sure you pass all test cases for this assignment.

```
yiqun@Yiquns-MacBook-Pro a1 % python grader.py
```

Q-Learning Pacman Assignment

In this assignment, you will explore reinforcement learning by developing a Pacman agent that evolves from random behavior to intelligent decision-making using Q-Learning. Part A introduces you to the game environment with a random player, while Part B challenges you to implement Q-Learning to help Pacman navigate mazes and avoid ghosts. Let's dive into this exciting journey of coding and learning!

Part A: Random Pacman Player

Objective: Implement a random Pacman player that competes against a single random ghost in a maze. This part lays the groundwork for understanding the game dynamics before introducing Q-Learning.

Tasks

1. Explore the Parser and Data Structures

Familiarize yourself with the provided parser and the data structures representing the game state.

Understand how Pacman's position, ghost positions, and maze layout are stored and updated.

Goal: Gain confidence in manipulating game states for action selection.

2. Implement Random Actions (get_action)

Modify the get_action method in the Pacman agent class to return a random valid action (e.g., up, down, left, right) based on the current game state. Ensure the selected action is legal by checking the maze constraints (e.g., no movement through walls).

Hint: Use the provided list of valid actions to simplify your implementation.

Testing

To test your implementation, run the following command in your terminal:

```
(base) ➜  final git:(master) ✗ python p1.py 1
Grading Problem 1 :
----------> Test case 1 PASSED <----------
(base) ➜  final git:(master) ✗ python p1.py 2
Grading Problem 1 :
----------> Test case 2 PASSED <----------
(base) ➜  final git:(master) ✗ python p1.py 3
Grading Problem 1 :
----------> Test case 3 PASSED <----------
(base) ➜  final git:(master) ✗ python p1.py 4
Grading Problem 1 :
----------> Test case 4 PASSED <----------
```

 # Part B: Q-Learning for Pacman Player

In this part, you will implement an advanced Pacman agent using Reinforcement Learning, specifically Q-Learning, to navigate mazes and avoid ghosts. The goal is to train Pacman to make intelligent decisions that maximize its chances of winning by balancing food collection and ghost avoidance.

Objective

Your task is to implement the core components of Q-Learning to enable Pacman to navigate mazes effectively while avoiding ghosts. Your implementation should achieve at least a 70% win rate across

test cases. You are encouraged to enhance the algorithm to achieve a higher win rate, as this will be a key grading metric.

Overview of Q-Learning in Pacman
Q-Learning is a model-free reinforcement learning algorithm that learns an optimal policy by estimating the expected future reward for each state-action pair, denoted \(Q(s, a) \). For Pacman, the algorithm follows these steps:

1. Initialize Q(s, a) for all state-action pairs (handled in the provided code).
2. Take an action based on the current strategy (random or Q-value-based).
3. Update Q(s, a) using the observed reward and the next state's Q-values.
4. Choose the next action using an ε-greedy strategy.
5. Repeat steps 3 and 4 until the game ends.
6. Perform a final update for the last state-action pair before the game concludes.

Most of the foundational code is provided. Your task is to complete the following components:
- Feature Engineering (`get_features`)
- Q-Value Calculation (`get_q_value`)
- Q-Value Update (`update`)
- Exploration vs. Exploitation Strategy (`get_action`)

```
"""
Q-Learning Pacman Assignment Part B

Objective: Implement core Q-learning components to help Pacman navigate mazes while avoiding ghosts.

Part 1: Feature Engineering (get_features)
- Implement state feature extraction using Manhattan distance
- Suggested features:
  1. closest_food: (distance to nearest food)
  2. ghost_distance: distance to ghost
  3. ghost_nearby: 1 if ghost within 2 units else 0

Part 2: Q-Value Calculation (get_q_value)
- Compute Q(s,a) = Σ(feature * weight)

Part 3: Q-Value Update (update)
- Implement Q-learning update rule using TD error

Part 4: Exploration vs Exploitation (get_action)
- Implement ε-greedy strategy

Part 5: Hyperparameter Tuning (Optional)
- Adjust feature weights
- Experiment with ε, α, γ values

Testing:
- Run: python p2.py 1 40 0 (problem_id, num_runs, if verbose information)
- Aim for >70% win rate on p1, p2, p3, p4
```

```
"""
```

Expected Outcome
Your implementation should achieve around a 70% win rate for each test case. To exceed this baseline:
- Experiment with additional features to better capture the game dynamics.
- Tune hyperparameters such as the learning rate (\(\alpha \)), discount factor \gamma, and exploration rate \varepsilon.
- The win rate will be a primary metric for evaluating your solution, so aim to maximize it.

Tips for Success
- Feature Design: Prioritize features that address immediate threats (e.g., ghost proximity) and long-term goals (e.g., food collection).
- Testing: Validate your implementation on small mazes before scaling to larger ones.
- Debugging: Monitor Q-values and weight updates during training to ensure they evolve as expected.

```
(base) ➜  final git:(master) X python p2.py 1 40 0
test_case_id: 1
num_trials: 40
verbose: False
[Training Complete]
Episodes: 500
Win Rate: 100.0%
Avg Steps per Episode: 74.0
time:  0.20802092552185059
win % 65.0
```

```
test_case_id: 4
num_trials: 40
verbose: False
[Training Complete]
Episodes: 500
Win Rate: 100.0%
Avg Steps per Episode: 314.5
time:  1.3849127292633057
win % 97.5
```

To submit your assignment to Moodle, `*.zip` the following files ONLY:

A1_Search
- p1.py
- p2.py
- p3.py
- p4.py
- parse.py

A1_RL:
- p1.py
- p2.py
- parse.py

- `grader.py`

Do not zip any other files. Use the `*.zip` file format.
Make sure that you have submitted the correct files.