# Assignment 2

## Introduction

- Download the code for this assignment and then unzip the archive.
- This assignment uses Python 3. Do not use python 2.
    - We have tested the assignment with Python 3.11.4.
- 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.

This assignment uses auto grading for Problems 1 and 3. For the other problems no auto grader is provided. The auto grader is provided for your own convenience and we won't use it to mark your submission. Instead, all submissions will be hand graded by our TA. Make sure that your code is readable and add appropriate comments, if necessary.

Important: Ensure that your code does not run too long. More than 5 min per test case (assuming at most 10 trials for each testcase) on our grading machine (M1 Mac) would be considered too long. For Problems 5 and 6 in particular it is required to search to reasonable depth which is only possible with a state representation that enables O(1) time queries. Problems 2, 4, 5 and 6 have a verbose option which is passed as a parameter into the search function. Only perform output computations when verbose is `True` to save valuable compute.

## Problem 1: Random Pacman vs. Single Random Ghost (Weight: 20%)

In this part of the assignment you are going to implement

- a parser to read a pacman layout file in the file `parse.py`, and
- a game of Pacman where Pacman and ghost select moves randomly in the file `p1.py`.

Both these python files have already been created for. Do not change anything that has already been implemented. Our auto grader relies on the existing code.

Start by implementing the `read_layout_problem()` in the file `parse.py`.

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

You can 'test' your code with the first layout as follows. This will simple output whatever you return from the `read_layout_problem()` function.

```
python parse.py 1 1
```

This will supply the `test_cases/p1/1.prob` file as an argument to the function. The file has the following Pacman layout.

```
seed: 8
```

```
%%%%
% W%
%  %
%  %
% .%
%P.%
%%%%
```

The first line is a random seed which will be used to initialize python's random number generator via the `random.seed()` function. This ensures that python generates a fixed sequence of random values. Important: A seed value of -1 will result in no seed being set. More on this later. The rest of the file is the Pacman layout that you will have to parse. You can expect the following characters in the file.

```
'%': Wall
'W': Ghost
'P': Pacman
'.': Food
' ': empty Square
```

You may assume the layout is always surrounded by walls and that there is at least one ghost. For problem 1 there will be a single ghost only. Later Pacman we will have to deal with more ghosts (`'X'`, `'Y'`, `'Z'`). There will always be at least one food and there will always be exactly one Pacman.

As in assignment 1, you can choose any data structure and return it from your `read_layout_problem` function. We recommend to design the data structure such that a location query of various entities in the world can be done in O(1) time, i.e., in constant time independent of world size.

Once you are done with the parsing you can move on to the second part of this problem and implement the `random_play_single_ghost()` function in the file `p1.py`.

A correct implementation will return the following string for the first test case.

```
seed: 8
0
%%%%
% W%
%  %
%  %
% .%
%P.%
%%%%
1: P moving E
%%%%
% W%
```

```
%  %
%  %
% .%
% P%
%%%%
score: 9
2: W moving W
%%%%
%W %
%  %
%  %
% .%
% P%
%%%%
score: 9
3: P moving W
%%%%
%W %
%  %
%  %
% .%
%P %
%%%%
score: 8
4: W moving E
%%%%
% W%
%  %
%  %
% .%
%P %
%%%%
score: 8
5: P moving E
%%%%
% W%
%  %
%  %
% .%
% P%
%%%%
score: 7
6: W moving S
%%%%
%  %
% W%
%  %
% .%
```

```
% P%
%%%%
score: 7
7: P moving N
%%%%
%  %
% W%
%  %
% P%
%  %
%%%%
score: 516
WIN: Pacman
```

As you can see, Pacman and the Ghost make moves alternatively. Pacman starts by making a move east. This is determined using the `random.choice()` function on the available moves to Pacman. In this case for the start state Pacman can move East (`E`) for the food or North(`N`) for the empty square. Pacman (`P`) moves east. Let's reproduce that decision to understand the sequence of moves generated here.

```
(base) scdirk@Dirks-Air a2 % python
>>> import random
>>> random.seed(8, version=1)
>>> random.choice(('E', 'N'))
'E'
```

Next, the Ghost (`W`) moves West (`W`). The available actions to the ghost are `W` and `S`.

```
(base) scdirk@Dirks-Air a2 % python
>>> import random
>>> random.seed(8, version=1)
>>> random.choice(('E', 'N'))
'E'
>>> random.choice(('S', 'W'))
'W'
```

Important: You must ensure that the parameter to the `random.choice()` function is sorted alphabetically. Otherwise you will not be able to reproduce the exact result and you won't be able to pass the auto grader.

In the test case 1 above, Pacman won because he cleared all the food. Pacman and Ghost may move to any square except a wall. A ghost may move on top of a food but it won't eat the food. Note that Pacman must always move. If Pacman is eaten, the game is over and Pacman loses.

Here is another run where that happens. This is test case 2.

```
seed: 42
0
%%%%
%.W%
%  %
%  %
% .%
%P.%
%%%%
1: P moving E
%%%%
%.W%
%  %
%  %
% .%
% P%
%%%%
score: 9
2: W moving S
%%%%
%. %
% W%
%  %
% .%
% P%
%%%%
score: 9
3: P moving W
%%%%
%. %
% W%
%  %
% .%
%P %
%%%%
score: 8
4: W moving N
%%%%
%.W%
%  %
%  %
% .%
%P %
%%%%
score: 8
5: P moving E
%%%%
%.W%
```

```
%  %
%  %
% .%
% P%
%%%%
score: 7
6: W moving S
%%%%
%. %
% W%
%  %
% .%
% P%
%%%%
score: 7
7: P moving N
%%%%
%. %
% W%
%  %
% P%
%  %
%%%%
score: 16
8: W moving W
%%%%
%. %
%W %
%  %
% P%
%  %
%%%%
score: 16
9: P moving W
%%%%
%. %
%W %
%  %
%P %
%  %
%%%%
score: 15
10: W moving S
%%%%
%. %
%  %
%W %
%P %
```

```
%  %
%%%%
score: 15
11: P moving E
%%%%
%. %
%  %
%W %
% P%
%  %
%%%%
score: 14
12: W moving S
%%%%
%. %
%  %
%  %
%WP%
%  %
%%%%
score: 14
13: P moving S
%%%%
%. %
%  %
%  %
%W %
% P%
%%%%
score: 13
14: W moving E
%%%%
%. %
%  %
%  %
% W%
% P%
%%%%
score: 13
15: P moving N
%%%%
%. %
%  %
%  %
% W%
%  %
%%%%
score: -488
```

```
WIN: Ghost
```

Scoring is done as follows.

```
EAT_FOOD_SCORE = 10
PACMAN_EATEN_SCORE = -500
PACMAN_WIN_SCORE = 500
PACMAN_MOVING_SCORE = -1
```

You may define any number of your own functions to solve the problem.
Once you are done you can check if you pass all the test cases for Problem 1.

```
(base) scdirk@Dirks-Air a2 % python p1.py
Grading Problem 1 :
----------> 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 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.

**Problem 2: Reflex Pacman vs. Single Random Ghost (Weight: 20%)**

Next, you will write a simple reflex agent that does not move randomly. The reflex agent should only consider the current state and have a simple logic. No need to make the agent too advanced (e.g., using expectimax) here. You should move to a better position based on a simple evaluation function, designed by you, that takes a state-action pair and returns the best action.

In our implementation we simply evaluating the distance to the closest ghost and combine with the distance to the closest food to rank the best move. *Do not run minimax/expectimax or any other more advanced algorithms here.* In this problem you are supposed to implement a simple reflex agent.

You may import (via `import` statement) some of the code that you wrote in Problem 1 and implement just the evaluation function and action selection by completing the code in the function `reflex_play_single_ghost()`. This function should return two values, the gameplay as usual and the winner which should be `'Pacman'` if Pacman wins. You can see how the `reflex_play_single_ghost()` is used in the main function of the script `p2.py`.

This time, we will have no auto grader and ghosts should move randomly without a seed. You may inspect the test cases and note that the seed is -1, which indicates that there will be no seed. This allows us to conduct multiple trials.

```
(base) scdirk@Dirks-Air a2 % python p2.py 1 10 0
test_case_id: 1
num_trials: 10
verbose: False
time: ?
win % 100.0
```

The three parameters control the test case id, number of trials and a verbose option that will output the actual gameplay if it is set to `1` instead of `0`.

```
base) scdirk@Dirks-Air a2 % python p2.py 1 1 1
test_case_id: 1
num_trials: 1
verbose: True
seed: -1
0
%%%%%
% . %
%.W.%
% . %
%. .%
%   %
%  .%
%   %
%P .%
%%%%%
…
123: P moving W
%%%%%
%   %
%   %
%   %
%P  %
%   %
%   %
%   %
%W  %
%%%%%
score: 518
WIN: Pacman
time: ?
win % 100.0
```

With a reasonable evaluation function you should be able to win some of the games. It is not necessary to achieve a high win rate for this question. You will implement expectimax soon and play better games.

For your reference, here is the performance (= win rate in % after playing 100 games) of a random agent, the min acceptable win rate of your agent and our reflex agent. You will have to check this manually because there is no auto grader for this question.

| Testcase | Random Agent | Min acceptable Reflex Agent | Our Reflex Agent |
|---|---|---|---|
| `1.prob` | 2 | > 50 | 100 |
| `2.prob` | 66 | > 80 | 100 |
| `3.prob` | 20 | > 60 | 97 |
| `4.prob` | 0 | > 50 | 100 |
| `5.prob` | 0 | > 50 | 100 |

Win rates are quite high because a reflex that is very careful and plays long games will be able to win most of the time. In our implementation we don't look at scores. To avoid long games and obtain a high score it would be desirable to win as fast as possible.

**Problem 3: Random Pacman vs. 4 Random Ghost (Weight: 10%)**

In this problem, Pacman is up against up to 4 ghosts ☹. This problem is similar to problem 1 except that you will implement a game against multiple ghosts.

Ghosts cannot move on top of each other. If a Ghost is stuck no move will be done. So a ghost may not make a move in this special case. Pacman will start followed by `W`, `X`, `Y`, `Z`. Note that the last three ghosts are optional. So the following combinations are possible:

- 1 Ghost: `W`
- 2 Ghosts: `W, X`
- 3 Ghosts: `W, X, Y`
- 4 Ghosts: `W, X, Y, Z`

Here is a game with 2 ghosts.

```
seed: 42
0
%%%%
%.X%
%W %
%P %
%%%%
1: P moving E
%%%%
%.X%
%W %
% P%
%%%%
```

```
score: -1
2: W moving E
%%%%
%.X%
% W%
% P%
%%%%
score: -1
3: X moving W
%%%%
%X %
% W%
% P%
%%%%
score: -1
4: P moving N
%%%%
%X %
% W%
%  %
%%%%
score: -502
WIN: Ghost
```

Note that ghosts move in alphabetical order, i.e., `W` first followed by `X` etc.

**Problem 4: Reflex Pacman vs. 4 Random Ghost (Weight: 20%)**

This problem is similar to P2 except that we have multiple ghosts as in P3. Again, no auto grading is provided and you may check the performance of your agent as follows.

```
(base) scdirk@Dirks-Air a2 % python p4.py 1 100 0
test_case_id: 1
num_trials: 100
verbose: False
time: ?
win % ?
```

*Don't worry too much if the performance of your agent is worse compared to P2.* This is to be expected considering the problem difficulty has increased with multiple ghosts.

For your reference, here is the performance (= win rate in % after playing 100 games) of a random agent, the min acceptable win rate of your agent and our reflex agent. You will have to check this manually because there is no auto grader for this question.

| Testcase | Random Agent | Min acceptable Reflex Agent | Our Reflex Agent |
|---|---|---|---|
| `1.prob` | 0 | > 30 | 49 |

| | | | |
|---|---|---|---|
| 2.prob | 22 | > 50 | 69 |
| 3.prob | 1 | > 10 | 27 |
| 4.prob | 0 | > 70 | 98 |
| 5.prob | 2 | > 10 | 24 |
| 6.prob | 1 | > 20 | 42 |
| 7.prob | 5 | > 10 | 28 |
| 8.prob | 0 | > 60 | 82 |
| 9.prob | 0 | > 50 | 74 |

As expected, the win rates are much lower compared to P2.

**Problem 5: Expectimax Pacman vs. Single Random Ghost (Weight: 15%)**

In this problem, you will implement an Expectimax agent. Hopefully, Pacman will be able to outsmart a single random Ghost on any map. To make this happen you will have to carefully design your model with an evaluation function and an appropriate depth limit. So your Expectimax search will search until a ply depth k (i.e., k moves by everyone) and then use an evaluation function to determine the value of the state at that depth.

Find an appropriate value for k such that with your evaluation function and state representation your code does not run too long. *More than 5 min per problem (assuming at most 10 trials for each testcase) on our grading machine would be considered too long.* How deep you will be able to go will to a large extend depend on how efficiently you have represented the model. For Pacman vs. Single ghost we may not have to go extremely deep to play well. However, for the next problem you will have to go deeper or develop a more advanced evaluation function.

For this assignment, we only care about win rate and not the score. Therefore, you can change how games are scored internally or modify your evaluation function to max the win rate.

The parameter k is provided as an argument to the function. You can test the performance of your game playing agent as follows.

```
(base) scdirk@Dirks-Air a2 % python p5.py 1 3 10 0
test_case_id: 1
k: 3
num_trials: 10
verbose: False
time: ?
win % ?
```

**Problem 6: Expectimax Pacman vs. 4 Random Ghost (Weight: 15%)**

In this problem, you will implement an Expectimax agent again. This time against up to 4 random ghosts. Test cases are the same as in P4. Are you able to perform better?

A good performance will strongly depend on your evaluation function and the efficiency (time and space) of your state representation. You should describe your design choices and the performance of your agent in detail in the short written report. More details below.

**Congratulations you have completed this assignment.**

**Short Written Report**

Write a short (at most four A4 pages) written report in PDF format explaining which problem were completed and where you have struggled. The report should focus on the problems that do not come with an auto grader (i.e., Problems 2, 4, 5 and 6). For each of these problems write down the performance of your agent for each test case and provide the parameters (e.g., num trails, k) that you have used to measure the performance. We will verify this when marking your work. We will be looking at your code when marking. Make sure everything is well documented and the code is well organized and easy to read. A portion of the marks will be allocated to well-maintained documented and easy to read code.

In the report, you should explain the various design choices that you have made regarding state representation and evaluation functions etc. What was the impact of your design choices?

Finally, write down the approximate number of hours you have spent per questions for this assignment.

**Submission**

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

- `p1.py`
- `p2.py`
- `p3.py`
- `p4.py`
- `p5.py`
- `p6.py`
- `parse.py`
- `report.pdf`

Do not zip any other files. Use the `*.zip` file format. Name the file `UID.zip`, where UID is your 10 digit university number. Make sure that you have submitted the correct files and named all files correctly. We will deduct up to 5% for files with incorrectly file names. We will not allow late submissions. Submission of incorrect files may result in 0 marks.

Check that you have submitted the correct files before the deadline.