Assignment 2

CMPT310: Artificial Intelligence Survey

Due date: February 28, 2022 11:59PM

Note that you need to submit the complete assignment for both parts before the deadline, otherwise you get the late policy penalty.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; please don't let us down. If you do, we will pursue the strongest consequences available to us.

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

Discussion: Please be careful not to post spoilers.

Part I

Question 1 (4 points)

Explain about Recursive Best First Search (RBFS). (2 points)

What are its properties, i.e. optimality, completeness, time and memory complexity? (2 points)

Question 2 (2 points)

For each of the following assertions, say whether it is true or false and support your answer with examples or counterexamples where appropriate.

- For any local-search problem, hill-climbing will return a global optimum if the algorithm is run starting at any state that is a neighbor of a neighbor of a globally optimal state.

- Suppose the temperature schedule for simulated annealing is set to be constant up to time N and zero thereafter. For any finite problem, we can set N large enough so that the algorithm returns an optimal solution with probability 1.

Question 3 (3 points)

Solve the cryptarithmetic problem in Figure 1 by hand, using the strategy of backtracking with forward checking and the MRV and least-constraining-value heuristics. Show the trace of each variable choice and assignment. In this problem, each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed.

$$
\begin{array}{cccc}
 & T & W & O \\
+ & T & W & O \\
\hline
F & O & U & R \\
\end{array}
$$

Figure 1: Cryptarithmetic Problem.

Question 4 (3 points)

Use the AC-3 algorithm to show that arc consistency can detect the inconsistency of the partial assignment $\{WA = green, V = red\}$ for the problem shown in Figure 2.
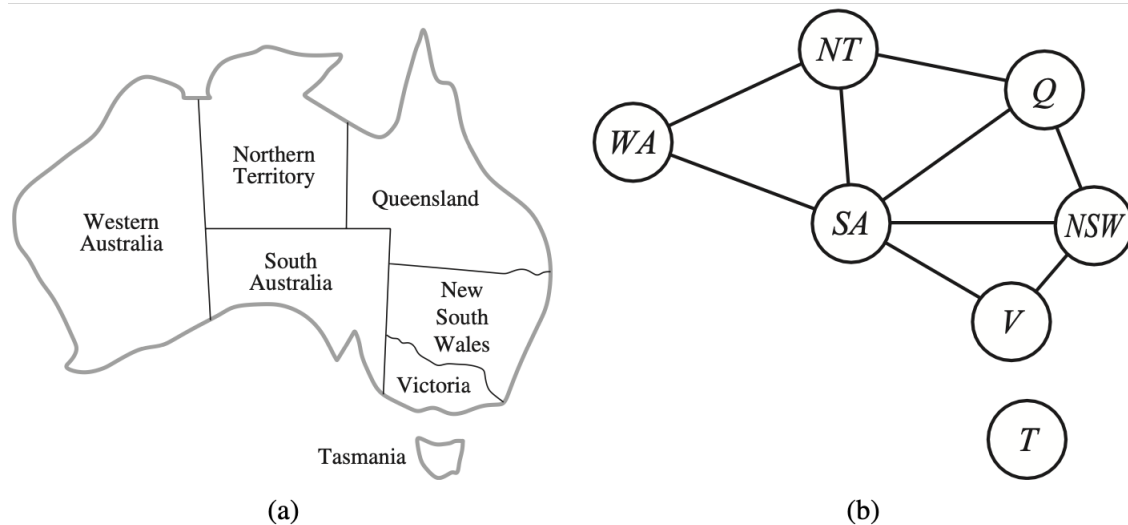

Figure 2: Map Coloring Problem.

Question 5 (3 points)

This question considers pruning in games with chance nodes. Figure 3 shows the complete game tree for a trivial game. Assume that the leaf nodes are to be evaluated in left- to-right order, and that before a leaf node is evaluated, we know nothing about its value–the range of possible values is $-\infty$ to $+\infty$.

(a) Copy the figure, mark the value of all the internal nodes, and indicate the best move at the root with an arrow. (0.5 point)

(b) Given the values of the first six leaves, do we need to evaluate the seventh and eighth leaves? Given the values of the first seven leaves, do we need to evaluate the eighth leaf? Explain your answers. (1 point)

(c) Suppose the leaf node values are known to lie between –2 and 2 inclusive. After the first two leaves are evaluated, what is the value range for the left-hand chance node? (1 point)

(d) Circle all the leaves that need not be evaluated under the assumption in (c). (0.5 point)
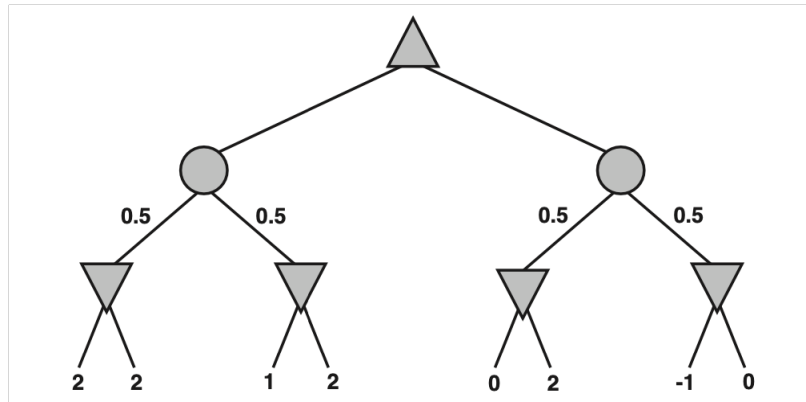
Figure 3: The complete game tree for a trivial game with chance nodes.

Submission

In order to submit your written assignment, please upload it as a PDF file to Canvas.

Part II

Pacman Problem

This part is adapted from UC Berkeley and UW.

Introduction

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

The code base has not changed much from the previous project, but please start with a fresh installation, rather than intermingling files from project 1.

As in project 1, this project includes an autograder for you to grade your answers on your machine. This can be run on all questions with the command:

```
python autograder.py
```

Note: If your python refers to Python 2.7, you may need to invoke python3 autograder.py (and similarly for all subsequent Python invocations) or create a conda environment as described in Project 0.

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

```
python autograder.py -q q2
```

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

```
python autograder.py -t test_cases/q2/0-small-tree
```

By default, the autograder displays graphics with the -t option, but doesn't with the -q option. You can force graphics by using the –graphics flag, or force no graphics by using the –no-graphics flag.

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

You can download all the code and supporting files as a zip archive from `Files/Assignment 2/multiagent.zip`.

The code for this project contains the following files:

Files you'll 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 also describes a Pacman GameState type, which you will use extensively in this project.

- `game.py` → The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.

- `util.py` → Useful data structures for implementing search algorithms. You don't need to use these for this project, but may find other functions defined here to be useful.

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` → 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 2 specific autograding test classes.

Welcome to Multi-Agent Pacman

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

`python pacman.py`

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

`python pacman.py -p ReflexAgent`

Note that it plays quite poorly even on simple layouts:

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

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

Question 1 (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:

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

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

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

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

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

Note: The evaluation function you're writing is evaluating state-action pairs; in later parts of the project, you'll be evaluating states.

Note: 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 `print(str(newGhostStates))`.

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

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 addition 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

```
python autograder.py -q q1
```

To run it without graphics, use:

```
python autograder.py -q q1 --no-graphics
```

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

Question 2 (5 points): Minimax

Now you will write an adversarial search agent in the provided `MinimaxAgent` class stub in `multiAgents.py`. Your minimax agent should work with any number of ghosts, so you'll have to write an algorithm that is slightly more general than what you've previously seen in 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: A single search ply is considered to be one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving two times.

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

```
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:

```
python autograder.py -q q2 --no-graphics
```

Hints and Observations

- The correct implementation of minimax will lead to Pacman losing the game in some tests. This is not a problem: as it is correct behaviour, it will pass the tests.

- The evaluation function for the Pacman test in this part is already written (`self.evaluationFuncti`
  You shouldn't change this function, but recognize that now we're evaluating states rather than actions, as we were for the reflex agent. Look-ahead agents evaluate future states whereas reflex agents evaluate actions from the current state.

- The minimax values of the initial state in the `minimaxClassic` layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. Note that your minimax agent will often win (665/1000 games for us) despite the dire prediction of depth 4 minimax.

  ```
  python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
  ```

- Pacman is always agent 0, and the agents move in order of increasing agent index.

- All states in minimax should be `GameStates`, either passed in to `getAction` or generated via `GameState.generateSuccessor`. In this project, you will not be abstracting to simplified states.

- On larger boards such as `openClassic` and `mediumClassic` (the default), you'll find Pacman to be good at not dying, but quite bad at winning. He'll often thrash around without making progress. He might even thrash around right next to a dot without eating it because he doesn't know where he'd go after eating that dot. Don't worry if you see this behavior, 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:

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

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

Question 3 (5 points): Alpha-Beta Pruning

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

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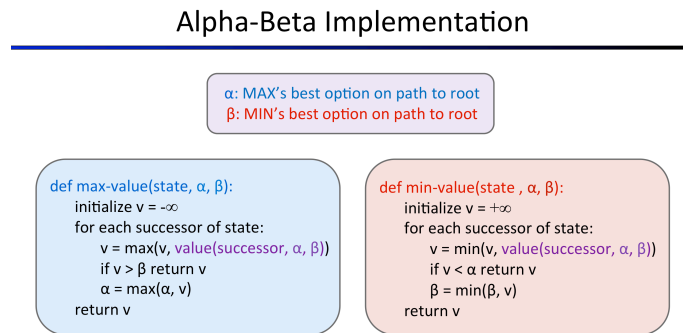
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic

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

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.

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

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v > β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v < α return v
        β = min(β, v)
    return v
```

To test and debug your code, run

```
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:

```
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 behaviour, it will pass the tests.

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 probabilistic behavior of agents who may make suboptimal choices.

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've supplied some test cases based on generic trees. You can debug your implementation on small the game trees using the command:

```
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, and 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.

To see how the `ExpectimaxAgent` behaves in Pacman, run:

```
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'll at least try. Investigate the results of these two scenarios:

```
python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
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.

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

Question 5 (6 points): Evaluation Function

Write a better evaluation function for pacman in the provided function `betterEvaluationFunction`. The evaluation function should evaluate states, rather than actions like your reflex agent evaluation function did. 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 points. 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 attu, 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

```
python autograder.py -q q5
```

To run it without graphics, use:

```
python autograder.py -q q5 --no-graphics
```

Submission

In order to submit your project, please upload `multiAgents.py` file to Canvas. Please do not upload the files in a zip file or a directory.