CS512– Artificial Intelligence

Instructor : Shashi Shekhar Jha (shashi@iitrpr.ac.in)

Lab Assignment - 2

Due on 10/03/2019 2400 Hrs

Submissions Instructions:

All implementation should be done in **python** (for fairness among all the submissions). Upload the submission to the **google classroom** in one zip file. In case you face any trouble with the submission, please contact the TAs:

- Sanatan Sukhija, sanatan@iitrpr.ac.in
- Prateek Munjal, 2017csm1009@iitrpr.ac.in
- Kapil Rana, 2018csz0007@iitrpr.ac.in

Your submission must be your original work. Do not indulge in any kind of plagiarism or copying. Abide by the honour and integrity code to do your assignment.

As mentioned in the class, late submissions will attract penalties.

You submission must include:

- A legible PDF document with all your answers to the assignment problems.
- A folder named as ‘code’ containing the scripts for the assignment along with the other necessary files to run your code.
- A README file explaining how to execute your code.

Naming Convention:

Name the ZIP file submission as follows: rollnumberhwnumber.zip

E.g. if your roll number is 2016csx1234 and submission is for lab2 Then you should name the zip file as: 2016csx1234lab2.zip

Q.1. The travelling salesman problem (TSP) is as follows: A salesman has to visit a set of cities. The problem is to find the shortest path for the salesman that visits each city exactly once and then returns to the starting city.

If the number of cities is large, then computing the absolute optimal path is extremely difficult; however, there are efficient techniques that compute very good paths. The algorithm below is not actually one of them. it can get stuck at very poor solutions that are local minima however it is a nice exercise in hill climbing.

Hill climbing **(10 points)**

For this assignment, you will use simple hill-climbing to compute an approximate solution to travelling salesman, as follows:

- The cities are points in the plane, and the distance between two cities is the geometric (Euclidean) distance.
- A state is ordering of the cities. The corresponding path goes from the first to the second to the last and back to the first.
- The heuristic function is just the total length of the path.
- The operator is to choose any two cities in the list and swap their position. For instance, suppose that the cities are labelled A to H and the current state S is the following sequence [B,F,C,A,G,H,D,E]. One possible swap would be to swap F with G. That would give you the new sequence [B,G,C,A,F,H,D,E]. If this is shorter, then it is an improvement.
- The start state is just the order in which the cities are read from input.
- At each stage of the hill-climbing, you have to consider swaps between all pairs of cities. Therefore, if there are N cities, you have to consider N(N-1)/2 swaps.

Simple algorithm

The basic algorithm is as follows

```
TravellingSaleman(Path: sequence of points) {
   Compute the initial length of Path
   loop {
      choose the pair of points U,V such that
         swapping U with V in Path has the shortest length;
      if there is no improvement, then return Path;
      swap U and V in the path and decrement the length by the change
    }
}
```

The problem with this is that it requires time O(n^3) on each iterations (all choices of U and V times the time to compute the length). You will get 9 points out of 10 for this.

Improved algorithm

The running time on the inner loop can be reduced to O(n^2) as follows:
There are two different kinds of swaps. If you are swapping two cities that are currently in sequence in the state, then you eliminate two edges and add two edges.

The second kind of swap involves two cities that are not in sequence. In that case, then you eliminate four edges and gain four new edges. For instance if you have the sequence [B,F,C,A,G,H,D,E] and you swap F and G, the path loses the edges B-F,F-C,A-G,G-H and gains the edges B-G,G-C,A-F,F-H. If the total length of the edges that have been gained is less than the total length of the edges that have been lost, then the path is now shorter, so this would be an improvement.

For large number of cities, this is an improvement over the simple algorithm because you don't have to recompute the entire length of the path for each swap, you just have to compute the change that the swap makes, which involves only 4 or 8 edges.

Input and output

The input is a file with the coordinates of points. There are two lines: first all the x-coordinates then all the y-coordinates per line. For example:

```
   0.0  1.0  2.0  0.0  2.0 1.0
   0.0  2.0  0.0  1.0  1.0 0.0
```
is the input for a problem with six cities.

Output: At each iteration of the hill climbing algorithm, show the current sequence of points and the total length of the path.

Extra points:
If you also render the path at each iteration of the hill climbing algorithm.

Example:
Input (to be read from a file)
```
   0.0  1.0  2.0  0.0  2.0  1.0
   0.0  2.0  0.0  1.0  1.0  0.0
```

Output

```
   Path:
   0.0  1.0  2.0  0.0  2.0  1.0
   0.0  2.0  0.0  1.0  1.0  0.0

   Length = 11.1224

   Swap nodes at position 1 and 5

   2.0  1.0  2.0  0.0  0.0  1.0
   1.0  2.0  0.0  1.0  0.0  0.0

   Length = 9.3006

   Swap nodes at position 2 and 3

   2.0  2.0  1.0  0.0  0.0  1.0
   1.0  0.0  2.0  1.0  0.0  0.0

   Length = 8.0645

   Swap nodes at position 1 and 2

   2.0  2.0  1.0  0.0  0.0  1.0
   0.0  1.0  2.0  1.0  0.0  0.0

   Length = 6.8284

   End of hill climbing
```

Q.2. Implement simulated annealing to the problem discussed in Q.1. You can use the 2-opt neighbourhood function (https://en.wikipedia.org/wiki/2-opt) for local search. A linear cooling regime can be used with a large initial temperature which is decreased with each iteration. You can look into the TSPLIB (http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/) for various benchmark problems to test your implementation. **[10 points]**

Q.3. A further generalisation of TSP is the vehicle routing problem (VRP) where the goal is to find a set of optimal routes for a fleet of vehicles to visit a set of customers locations. All the vehicles start from a Depot location and return to the Depot. The set of customer locations are given as x, y coordinates. Implement a genetic algorithm (GA) based solution to find the best set of routes to visit all the customers. The states is represented as a string of customer nodes such as the following:

D	A	B	C	**D**	**D**	E	F	**D**	**D**	G	H	I	**D**

Here, the representation is for three routes demarcated by the depots **D.**
Note, a sequence of 3 or more depots (D) should be eliminated since such a sequence doesn't contribute to any route. You are free to choose/design the cross-over and mutation operator along with the population size and selection procedure. **[15 points]**

The inputs to the program will be from a file with the following information:
MaxGen [number of generations to run GA]
Random seed [A random seed to be used to generate all random values in your implementation]

CUST No.	XCOORD.	YCOORD.
1	35	35
2	41	49
3	35	17
4	55	45
5	55	20
6	15	30
7	25	30

Output:
Number of routes found: 3
Route 1
D A B C D

Route 2
D E F D

Route 3
D G H I D

Multi-Agent Pacman (http://ai.berkeley.edu/multiagent.html**)**

This part of the lab has been adopted from Berkeley Pacman projects. 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 lab assignment, but please start with a fresh installation, rather than intermingling files from lab 1. As in lab 1, this lab 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
```

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.

The code for this project contains the following files, available as a zip archive.

Files you'll edit:	
multiAgents.py	Where all of your multi-agent search agents will reside.
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.
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
multiagentTestClasses.py	Project 2 specific autograding test classes

You will fill in portions of `multiAgents.py` during the assignment. You should submit this file with your code and comments. Please do not change the other files in this distribution or submit any of our original files other than this file. 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. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Multi-Agent Pacman
First, play a game of classic Pacman:
```
python pacman.py
```

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.

Q.4. **Reflex Agent** (4 points)
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 must 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.

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
```

Q.5. **Minimax** (5 points)
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. 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 one Pacman move and all the ghosts' responses, so depth 2 search will involve Pacman and each ghost moving two times.
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.evaluationFunction). 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 lab, 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, last question 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.

Q.6. **Alpha-Beta Pruning** (5 points)

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.

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.)

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.7. **Expectimax** (6 points)

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 modelling probabilistic behaviour 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. Make sure when you compute your averages that you use floats. Integer division in Python truncates, so that 1/2 = 0, unlike the case with floats where 1.0/2.0 = 0.5.

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.

Q.8. **Evaluation Function** (5 points)
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).

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

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.

The additional points for average score and computation time will only be awarded if you win at least 5 times.

Hints and Observations
As for your reflex agent evaluation function, you may want to use the reciprocal of important values (such as distance to food) rather than the values themselves. One way you might want to write your evaluation function is to use a linear combination of features. That is, compute values for features about the state that you think are important, and then combine those features by multiplying them by different values and adding the results together. You might decide what to multiply each feature by based on how important you think it is