Computer Science 384
St. George Campus

Sunday, March 19, 2017
University of Toronto

AI Fixed Project Assignment – Pacman
(Final Version)
Final Project Due: Wednesday, April 5, 2017 by 11:59 PM

Preface: Every student in CSC384 must either individually complete the "fixed project" or work in a group to complete a "custom project" designed by the group. This document describes the "fixed project" option. **** This document is the Final Version of the "Fixed Project" specification, and extends the Version 1 with a more elaborated explanation of Question 4.**

Late Policy: Project materials are due on the last day of class (April 5, 2017 by 11:59 PM). If you have remaining grace days, they may be applied to extend this deadline. Late assignments will not be accepted.

Total Marks: This assignment represents 15% of the course grade.

Teaming: No teaming. This project is to be performed **individually**.

What to submit electronically:

By Wednesday, April 5, 11:59 PM

- multiAgents.py, suitably modified to include code for your different game tree agents.

How to submit: Submit your project assignment using MarkUs. Your login to MarkUs is your teach.cs username and password. It is your responsibility to include all necessary files in your submission. You can submit a new version of any file at any time. Note that assignments are due on the last day of classes, April 5. This deadline can only be extended with grace days. More detailed instructions for using Markus are available at: http://www.cdf.toronto.edu/~csc384h/winter/markus.html.

Warning: marks will be deducted for incorrect submissions.

- *Make certain that your code runs on teach.cs using python2 (version 2.7) using only standard imports.* Your code will be tested using this version and you will receive zero marks if it does not run using this version. **Note that this version is different from the first 2 assignments. We are no longer using python3 for this assignment.**
- *Do not add any non-standard imports from within the python file you submit (the imports that are already in the template files must remain).* Once again, non-standard imports will cause your code to fail the testing and you will receive zero marks.
- *Do not change the supplied starter code.* Your code will be tested using the original starter code, and if it relies on changes you made to the starter code, you will receive zero marks.

MCTS Tutorial Slides: The MCTS tutorial slides are posted in the CSC384 Projects website: http://www.teach.cs.utoronto.ca/~csc384h/winter/Project/FProject/project.html.

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Fixed Project Clarification page, linked from the CSC384 web page. You are responsible for monitoring the Project Clarification page:

http://www.teach.cs.utoronto.ca/~csc384h/winter/Project/FProject/fixed-project_faq.html.

Questions: Questions about the assignment should be posted to Piazza:
https://piazza.com/utoronto.ca/winter2017/csc384/home.

Introduction

Acknowledgements: This project is based on Berkeley's CS188 EdX course project. It is a modified and augmented version of "Project 2: Multi-Agent Pacman" available at http://ai.berkeley.edu/multiagent.html.

For this project, you must implement **four different game tree search agents** for the game of Pacman.

Question 1: The implementation of a Minimax agent.

Question 2: The implementation of a Minimax agent with alpha-beta pruning.

Question 3: The implementation of an Expectimax agent.

Question 4: The implementation of a Monte-Carlo Tree Search (MCTS) agent with UCB.

What is Supplied

Important: `MonteCarloAgent.txt` is the only file that differs from the files released with the Version 1 of this document.
(File descriptions follow those of the UC Berkeley CS188 multi-agent project.)

- Files you will edit.
  - **multiAgents.py** - suitable augmented with your implementation of the search agents described in questions 1 to 4 in this handout.
  - **MonteCarloAgent.txt** - a python class that you will have to augment with your implementation of the UCT search agent, described in question 4. You must copy & paste the `MonteCarloAgent` class into the `multiAgents.py` python file.
- Files that are useful to your implementations.
  - **pacman.py** - Runs Pacman games. This file also describes a Pacman GameState, which you will use extensively in your implementations.
  - **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 that are unlikely to be of help to your implementations, but are required to run the project.
  - **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.
- **multiagentTestClasses.py** - Project specific autograding test classes.
- **grading.py** - File that specifies how much each question in autograder is worth.
- **projectParams.py** - Project parameters, facilitates nicer output during autograding.
- **pacmanAgents.py** - Classes for specifying ghosts' behaviour.

The Pacman Game

Pacman is a video game first developed in the 1980s. A basic description of the game can be found at https://en.wikipedia.org/wiki/Pac-Man. In order to run your agents in a game of Pacman, and to evaluate your agents with the supplied test code, you will be using the command line. To familiarize yourself with running this game from the command line, try playing a game of Pacman yourself by typing the following command:

```
python pacman.py
```

To run Pacman with a game agent use the `-p` command. Run Pacman as a `GreedyAgent`:

```
python pacman.py -p GreedyAgent
```

You can run Pacman on different maps using the `-l` command:

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

Important: If you use the command in an environment with no graphics interface (e.g. when you `ssh` to teach.cs, formerly CDF), you must use the flag `-q`.
The following commands are available to you when running Pacman. They are also accessible by running `python pacman.py --help`.

- `-p` Allows you to select a game agent for controlling pacman, e.g., `-p GreedyAgent`. By the end of this project, you will have implemented four more agents, listed.
  - `MinimaxAgent`
  - `AlphaBetaAgent`
  - `ExpectimaxAgent`
  - `MonteCarloAgent`
- `-l` Allows you to select a map for playing Pacman, e.g., `-l smallClassic`. There are 9 playable maps, listed.
  - `minimaxClassic`
  - `trappedClassic`
  - `testClassic`

  - `smallClassic`
  - `capsuleClassic`
  - `openClassic`
  - `contestClassic`
  - `mediumClassic`
  - `originalClassic`

- `-a` Allows you to specify agent specific arguments. For instance, for any agent that is a subclass of `MultiAgentSearchAgent`, you can specify the depth that you limit your search tree by typing `-a depth=3`
- `-n` Allows you to specify the amount of games that are played consecutively by Pacman, e.g., `-n 100` will cause Pacman to play 100 consecutive games.
- `-x` Allows you to specify an amount of training cases. The results of these cases are not displayed in the terminal. For example, `-x 50` will allow you to run Pacman over 50 training cases before starting to play games where the output is visible. This will be useful when testing you Monte-Carlo Agent.
- `-k` Allows you to specify how many ghosts will appear on the map. For instance, to have 3 ghosts chase Pacman on the `contestClassic map`, you can type `-l contestClassic -k 3`
  Note: There is a max number of ghosts that can be initialized on each map, if the number specified exceeds this number, you will only see the max amount of ghosts.
- `-g` Allows you to specify whether the ghost will be a `RandomGhost` (which is the default) or a `DirectionalGhost` that chases Pacman on the map. For instance, to have DirectionalGhost characters type `-g DirectionalGhost`
- `-q` Allows you to run Pacman with no graphics.
- `--frameTime` Specifies the frame time for each frame in the Pacman visualizer (e.g., `--frameTime 0`).

An example of a command you might want to run is:

```
python pacman.py -p GreedyAgent -l contestClassic -n 100 -k 2 -g DirectionalGhost -q
```

This will run a GreedyAgent over 100 cases on the `contestClassic` level with 2 `DirectionalGhost` characters, while supressing the visual output.

Evaluation

Your evaluation will be based on the correct implementation of 4 different agents: Minimax, Minimax with alpha-beta pruning, Expectimax, and UCT (Monte-Carlo Tree Search with UCB_1) search.

- **Correct Implementation (20 $\times$ 4 = 80 marks)**
  Each of your agents will be tested for correctness. For the first three, we will test this by observing whether the correct nodes in the game tree have been expanded. When you implement UCT (i.e.,

MCTS + UCB_1), keep in mind two things. (i) The states visited by MCTS rollouts are thrown away, and the win/loose information is propagated backwards **in the search tree**. (ii) The UCB score in a tree node is computed with the perspective of the player that associated with the node. That is, the average **wins** of a parent node following the path of a children node is computed from the number of **loses** of the children. The correct implementation of each agent is worth **20 marks**. A subset of the tests used in automarking your implementation will be released so you can test whether your implementations are correct.

- **MCTS Enhancements (20 marks)**
  You will experiment with the tuning of different parameters, as well as with an implementation of an improved evaluation function in order to enhance the performance of your `MonteCarloAgent`. We will ask you to achieve specific benchmarks on 4 different maps. The maximum achievable marks per map is **5 marks**.

We will auto-test your code rather than hand marking it. You will be supplied with a testing script that will run a **subset** of the tests. If your code fails all of the tests performed by the script (using Python version **2.7**), you will receive zero marks. It is your responsibility to make sure your code runs in Python version **2.7**, and that passes at least some of the tests in the provided testing script. It's up to you to figure out additional test cases to further test your code – that's part of the assignment.

When your code is submitted, we will run a more extensive set of tests which will include the tests run in the provided testing script and a number of other tests. You have to pass all of these tests to obtain full marks on the assignment.

Question 1: Minimax Agent **(worth 20/100 marks)**

Your first task is to implement a Minimax agent to play the game of Pacman. The function `getAction` in `MinimaxAgent` located in in `multiAgents.py` provides the complete **input/output specification** of the function you are to implement.

The correct implementation of the Minimax agent is worth 20/100 marks.

Brief implementation description: Your Minimax algorithm should follow the basic implementation discussed in lecture. However, your algorithm should work with any number of ghosts, so you'll have to modify the implementation you saw in class (which works only for two players). Your search tree will consist of multiple layers - a max layer for Pacman, and a min layer for each ghost. For instance, assume you are implementing Minimax for a game with 3 ghosts. Then, a search of depth 1 will consist of 4 levels in the search tree - one for Pacman, and then one for each of the ghosts.

`getAction` The function receives a `gameState`, and must return the best action according to Minimax search. `gameState.getLegalActions(agentIndex)` will get all the legal actions for a given agent. `agentIndex` for Pacman is 0, and ghosts are ≥ 1. `gameState.generateSuccessor(agentIndex, action)` will return the successor of a game-action pair for a given agent. `gameState.getNumAgents()` returns the number of agents in the game. `depth` is one of the parameters passed to your `MinimaxAgent` upon initialization. You will need to store this variable, and make sure that your agent only searches to the specified depth. Once you reach the specified depth, you should evaluate the nodes of the tree using `self.evaluationFunction` which will default to `scoreEvaluationFunction` (also located in `multiAgents.py`). Notice that a search of depth 2 will result in two moves for Pacman, and two moves for each ghost.

You can run your `MinimaxAgent` on a game of Pacman by running the following command:

```
python Pacman.py -p MinimaxAgent -l minmaxClassic -a depth=3
```

*Explorative question (you do **not** have to hand these in)*: Note that Pacman will have suicidal tendencies when playing in situations where death is imminent. Why do you think this is the case?

Question 2: Alpha-Beta Pruning **(worth 20/100 marks)**

You will now implement a Minimax agent with alpha-beta pruning to play the game of Pacman. The function `getAction` in `AlphaBetaAgent` located in in `multiAgents.py` provides the complete **input/output specification** of the function you are to implement.

The correct implementation of the AlphaBeta agent is worth 20/100 marks.

Brief implementation description: Your alpha-beta pruning algorithm should follow the implementation discussed in lecture. Your algorithm should work with any number of ghosts, so you'll have to modify the implementation you saw in class to accommodate for multiple adversarial agents, similarly to what you did for your `MinimaxAgent`.

`getAction` This function receives a gameState as input, and should return the best action determined by alpha-beta pruning upon return. Once again, your function should only search to the specified depth passed to the agent upon initialization, and evaluate nodes based on self.evaluationFunction.

Watch your agent play by running the following command:

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l minimaxClassic
```

*Explorative questions (you do **not** have to hand these in)*: You should notice a significant speed-up compared to your Minimax agent. Should the performance for the AlphaBetaAgent be expected to be the same as Minimax agent given a certain depth?

Question 3: Expectimax Agent **(worth 20/100 marks)**

You will now implement an Expectimax agent to play the game of Pacman. The function `getAction` in `ExpectimaxAgent` located in in `multiAgents.py` provides the complete **input/output specification** of the function you are to implement. An Expectimax agent no longer makes the assumption that it is playing against an optimal adversary. As this is the case in Pacman, this will lead to better modelling of the opposing ghosts. The only difference in implementation of Expectimax search and Minimax search, is that at a min node, Expectimax search will return the average value over its children as opposed to the minimum value.

The correct implementation of the Expectimax agent is worth 20/100 marks.

Brief implementation description: Your Expectimax algorithm should follow the implementation of Minimax discussed in lecture, with the average taken over the children of a min node, rather than the minimum. Your algorithm should work with any number of ghosts, similarly to your `MinimaxAgent` and `AlphaBetaAgent` classes.

`getAction` This function receives a gameState as input, and should return the best action determined by Expectimax search upon return. Once again, your function should only search to the specified depth passed to the agent upon initialization, and evaluate nodes based on self.evaluationFunction.

Watch your agent play by typing in the following command:

```
python pacman.py -p ExpectimaxAgent -a depth=3 -l minimaxClassic
```

*Explorative questions (you do **not** have to hand these in)*: Try running Pacman on the `trappedclassic` map. Compare the results with those of your Minimax agent. Which agent has better performance? Why do you think this is the case?

Question 4: Monte-Carlo Tree Search (MCTS) Agent **(worth 40/100 marks)**

You will now implement a UCT (UCT = Monte-Carlo Tree Search + UCB_1) agent to play the game of Pacman. The functions `getAction, runSimulation`, and `final` in `MonteCarloAgent` (you have to copy & paste this class, from `MonteCarloAgent.txt`, into `multiAgents.py`) provide the complete **input/output**

specification of the functions you are to implement.

The correct implementation of the Monte-Carlo Tree Search agent is worth 20/100 marks.

Brief Implementation Description: The algorithm described in class works in games with two adversarial agents. In the Pacman game, one agent plays the role of Pacman (A1), and the other agent plays the role of N ghosts (A2). Notice that the game has perfect information, and A2 controls the movement of all N ghosts each turn.

Read the `__init__` function of `MonteCarloAgent` and make sure that you understand it. For instance, `depth` sets the depth to which your agent will search to *after the expansion phase*. You will traverse the tree using UCB_1 until you reach a leaf node, and from this point search to the specified depth. Setting `depth` $= -1$ will cause your agent to search until it reaches a win or a loss (i.e., infinite depth). Furthermore, if a depth is set, then your agent will use `scoreEvaluationFunction` rather than `mctsEvalFunction`.

Your implementation will be marked for whether it searches to the correct depth, whether after a certain number of simulations your agent has expanded the correct number of nodes, and whether your agent adheres to the correct timeout when running a simulation.

`getAction` This function should return the best action from `gameState` after running Monte-Carlo Tree Search for the set timeout (50 ms). Begin by returning the action with the highest v_i value, i.e., the number of wins divided by the number of plays.

`runSimulation` This function should update values in your search tree dictionaries (`self.plays` and `self.wins`). **Keys of these dictionaries *must* be `GameState` instances, and values must be `int`.** Iterate through your search tree, choosing the state that should be moved into according to the UCB_1 criterion. Once you reach a leaf node, expand that node, randomly selecting states to move into until a terminal is reached, or the maximum depth has been surpassed. Remember, states encountered after the expansion stage don't have their values updated. Update all of the values in your search tree based on the result of the simulation (post-expansion) phase. Your implementation should work with multiple adversarial ghosts. Depth should be as was done for the other agents - searching to a depth of 2 should result in 2 moves for Pacman, and 2 moves for each of the ghosts.

`final` This function should take the states that were encountered by Pacman during the game, and update their values according to the result of the game. This function is called every time a game of Pacman ends, and receives the final `state` of the game. States are updated in the same way that they are updated after a simulation.

Enhancements for MCTS Agent **(worth 20/100 marks)**

In this section, you will experiment with tuning different parameters, and with an **optional** implementation of an alternative evaluation function to improve the performance of your `MonteCarloAgent` over 4 specified maps. You are allowed to specify different parameters for each of the 4 different maps. For example, if you want to use your `MonteCarloAgent` without a depth cutoff for the `minimaxClassic` level, but want to use a depth-limited UCT for the `smallClassic` level, then that is acceptable. Remember, that your enhanced implementation must run on cdf, and thus non-standard imports cannot be used. This section will not be tested for correctness, only for performance.

Ways to make an improvement for your `MonteCarloAgent`:

- **Adjusting C (exploration parameter in UCB_1):** Adjusting the exploration parameter C will yield different performance. A higher C will result in Pacman putting a higher emphasis on exploring new states for which it has less information.
- **Search to a specified depth:** On some of the bigger maps, forcing Pacman to expand nodes until a win or loss is encountered can lead to not evaluating enough states to make good decisions. Setting the `depth` parameter will allow Pacman to evaluate more states. Depth-limited UCT should default to using a heuristic evaluation function (by default `scoreEvaluationFunction`) to evaluate the final state in the expansion phase.
- **Implementing an improved evaluation function:** There is space for you to implement an improved evaluation function - `betterEvaluationfunction` located in `MutliAgents.py`. This function can be passed to your agent in the terminal by setting `evalFn=better`. This function replaces the default `scoreEvaluationFunction` when using depth-limited UCT.
- **Altering the selection criterion:** When selecting which state to move into after simulation (in `getAction`), you were asked to choose the state with the highest v_i (`self.wins` / `self.plays`). You might want to implement an alternative selection criteria (as discussed in the tutorial) for choosing an action in `getAction`. **Hint:** We recommend you not to invest time exploring different action selection criteria because, in principle, there should not be dramatic changes in terms of performance. Remember the convergence guarantees of UCT.

The following table shows the marking scheme for a set of different maps. You will have to achieve **both the winning percentage and the average score** in order to achieve full marks:

Map	3 marks		4 marks		5 marks	
	avg score	**win %**	**avg score**	**win %**	**avg score**	**win %**
`minimaxClassic`	250	0.65	300	0.75	350	0.85
`testClassic`	-150	0.30	-100	0.40	0	0.50
`smallClassic`	-200	0.00	-150	0.05	-100	0.10
`contestClassic`	-200	0.00	200	0.00	500	0.01

The performance of your agents will be evaluated over 100 games to see if they achieve the designated benchmarks. This is process is stochastic, and we will allow for a margin of error. If your agent receives an average score 100 below the specified score, and wins 20% times less than what the benchmark, you will still receive full marks. Place the setting of C, the `depth`, and the evaluation function you want to use for each of the four maps (See the `__init__` function). For instance, if you would like to use a setting of `C=2, depth=10, evalFn=better` for the `testClassic` map, then you would place those parameters under "`if Q='minimaxClassic':`" in the `__init__` function.

HAVE FUN and GOOD LUCK!