Computer Science 384
St. George Campus

Monday, July 10, 2017
University of Toronto

AI Fixed Project Assignment – Pacman
Final Project Due: Sunday, August 14, 2017 by 11:59 PM

Preface: Some students have asked to work independently on the course project. While we'd prefer students to work on group projects, for those who wish to work individually, a "fixed project" described in this document can be completed. The protocol for group projects has already been posted to the course website.

Late Policy: Project materials are due on the last day of class (Sunday, August 14, 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: This project is an alternative to the group project and to be performed **individually**. Students who wish to work on other projects or in groups should complete the group project.

What to submit electronically:

By Wednesday, July 19, 11:59 PM

- `csc384-cover.pdf`: A pdf copy of your Project Cover Page, as described below.
- `csc384-proposal.pdf`: A pdf copy of your Project Proposal, as described below.

By Sunday, August 14, 2017, 11:59 PM

- `csc384-report.pdf`: A pdf copy of your Project Report, as described below.
- `csc384-source.zip`: A zip file containing all the source code for your project together. Your code must run on teach.cs unless otherwise discussed and approved in advance of submission.

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. This deadline can only be extended with grace days. More detailed instructions for using Markus are available at: `http://www.cdf.toronto.edu/~csc384h/summer/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 in that it does not use python3.**
- *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.

Clarification Page: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Project Clarification page (under the "Fixed Project" heading; this is linked from the CSC384 web page. You are responsible for monitoring the Project Clarification page:
http://www.teach.cs.utoronto.ca/~csc384h/summer/Project/project_faq.html.

Questions: Questions about the assignment should be posted to Piazza:
https://piazza.com/utoronto.ca/summer2017/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 **three different game tree search agents** for the game of Pacman, and a variant of the third agent:

Task 1: The implementation of a Minimax agent.

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

Task 3: The implementation of a Monte-Carlo Tree Search (MCTS) agent with UCB.

Task 4: The implementation of augmentations for your MCTS agent.

What is Supplied

(File descriptions follow those of the UC Berkeley CS188 multi-agent project.)

- Files you will edit:
  - **multiAgents.py** - Suitably augment this with your implementation of the search agents described in tasks 1 to 4 in this handout.
- 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 graphical interface (e.g. when you `ssh` to teach.cs, formerly CDF), you must use the flag `-q`, which suppresses graphical output.

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`
  - `MonteCarloAgent`
  - `AugmentedMonteCarloAgent`

- `-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.
- `-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.

Task 1: Minimax Agent

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.

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, for a game with 3 ghosts, 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. The `agentIndex` for Pacman is 0, and for ghosts, `agentIndex` ≥ 1. The function `gameState.generateSuccessor(agentIndex, action)` will return the successor of a state-action pair for a given agent. The function `gameState.getNumAgents()` returns the number of agents in the game. You will need to 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(state)`. For your first 3 agents, `self.evaluationFunction` is set to `scoreEvaluationFunction` (also located in `multiAgents.py`), and you will not need to change this. You will get to experiment with creating your own evaluation function to improve the performance of your `MonteCarloAgent`. 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 minimaxClassic -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?

Task 2: Alpha-Beta Pruning

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.

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` (accessed by calling `self.depth`), and evaluate nodes using `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?

Task 3: Monte-Carlo Tree Search (MCTS) Agent

You will now implement a UCT (UCT = Monte-Carlo Tree Search + UCB_1) agent to play the game of Pacman. The functions `getAction` and `runMonteCarlo` in `MonteCarloAgent` provide the complete **input/output specification** of the functions you are to implement.

Brief Implementation Description: You will implement Monte-Carlo Tree Search with UCB_1 as was shown in class. The algorithm described in class works with games of two agents. You will have to adapt this algorithm to include multiple adversarial agents, similarly to what you did for your previous agents. Read the `__init__` function of `MonteCarloAgent` and make sure that you understand it. As in your other agemts, the `depth` and `evaluationFunction` are specified. The `timeout` for simulating random trajectories during each call to `getAction` is also specified. The exploration parameter - `C`, used in UCB_1 is set. Two dictionaries - `self.plays` and `self.wins` are initialized. These dictionaries are used to keep statistics on states encountered during search. **It is important that the keys you use for these dictionaries are state-agent pairs. In our implementation, keys are tuples of the form `(GameState,int)`.**

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., number of wins divided by the number of plays.

`runMonteCarlo` This function will be the biggest component of your MonteCarloAgent, running Monte-Carlo Tree Search until the specified `timeout`. This function updates values in your search tree dictionaries (`self.plays` and `self.wins`). Monte-Carlo Tree Search consists of four phases - selection, expansion, simulation, and backpropagation. For selection, you will traverse the search tree, choosing each successor state that should be moved into according to the UCB_1 criterion. Once you reach a leaf node, i.e, a state that has not yet been encountered, expand that node: **If its plays is 0, start a simulation from that node - if it's plays is not 0, add each of its successors to the search tree by setting its plays and wins to 0, and beginning a random simulation from one of these successors**. After expansion, simulate a trajectory in the game by randomly selecting states to move into until a terminal is reached, or the maximum depth has been surpassed. Rememeber, `depth` sets the amount of states you *simulate* not the amount of states that you traverse during selection. Furthermore, depth should be calculated as it was for your other agents - a depth of 1 will result in one move for each game agent. Backpropagate through all of the states encountered during selection your based on the result of the simulation phase. **Make sure that you backpropagate through the root node so you can use the UCB_1 formula as it was shown in class.** Also, backpropagation should always be done from the perspective of the agent to which the state belongs to.

Task 4: Enhancements for MCTS Agent

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`. Try these out on different maps, and describe your results. For instance, you might want to use your `MonteCarloAgent` without a depth cutoff for the `minimaxClassic` level, but use a depth-limited UCT for the `smallClassic` level. 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 given 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. Setting the `depth` parameter will allow Pacman to evaluate more states. Depth-limited UCT defaults to using `scoreEvaluationFunction` to evaluate the final state in the expansion phase.
- **Implementing an improved evaluation function:** There is space for you to implement an improved heuristic evaluation function for the depth-limited UCT - `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.

Project Proposal

If you intend to work individually, we would like you to indicate this in your project proposal. Proposals will not be graded but will be used to provide feedback and guide support. Each project proposal must contain:

1. Project Cover Page (submitted as csc384-cover.pdf)
   (a) The title "Fixed PacMan Project Proposal"
   (b) Your name and teach.cs-login ID
2. Project Proposal (submitted as csc384-proposal.pdf)
   (a) The title "Fixed PacMan Project Proposal"

(b) Brief evaluation plan (200 words or less). The expectations are that you will provide comparisons between the game tree agents you implemented for the Pacman game. Your evaluation plan, then, should clearly explain how you will you evaluate your four solutions. How do they perform when compared to each other? How will you measure performance and how will you indicate that one solution is "better" than another? Clearly illustrating the inefficiencies in a given implementation can be a valuable outcome of your work. If you will be developing test cases, please note the nature and number of test cases you anticipate developing.

Project Report

1. The report you submit should explain your MCTS enhancements and provide an evaluation of your solutions. This evaluation should compare your MCTS implementation with alternative solutions. This report must have a **title page** containing the following information:

   - Title for your report;
   - Your name and teach.cs-login ID

2. Following the title page, include the details of your MCTS solution and your evaluation in the **report body**. This can be a maximum of 4 single-spaced pages, formatted in 12pt font. Sections for the report body should be as follows:

   - **Methods**. Explain which MCTS enhancements you chose to implement and why you thought these enhancements would improve the MCTS solution. For example, if you chose to improve the heuristic evaluation function of your Monte Carlo Agent, explain why you thought this would make MCTS better and how, precisely, the heuristic function was encoded.

   - **Evaluation and Results.** Here, describe your evaluation objectives and strategy, and your results. In particular, describe how you decided to measure the effect of your enhancements, i.e. how you chose to illustrate that the enhanced solution was "better" or "worse" than alpha-beta pruning, Minimax, or the standard MCTS implementation. Evaluation metrics may include the number of nodes expanded by each algorithm, the depth of the search horizon at each move, the number of prunes to the search trees, or the amount of time or memory that each algorithm used. Experiment with the tuning of different parameters to see if these make a difference to your performance measures. Note that we encourage the use of diagrams, graphs, and/or tables to summarize experimental results and to convey important points. Also note that its o.k. if your system proves to be inefficient in some way; that's still a result and we want to know.
   - **Limitations/Obstacles.** Here, document any obstacles you encountered during your project implementation or shortcomings you discovered in your solution approach.
   - **Conclusions.** Finally, explain what you learned and how you might improve or modify your program were you to try again in the future. Other reflections are welcome.

3. You may include up to 1 additional page after the report body for citations and references or any other attributions or acknowledgements.

Project Source Code

Your project code must run on teach.cs. All source code to run your completed project on teach.cs must be submitted, together with a README file explaining how to run the code, in a single zip file.

HAVE FUN and GOOD LUCK!