COMPSCI 270 — Homework 2

Instructor: Kris Hauser

Due: 02/20/2017

1 Instructions

Read the assignment carefully, complete the programming tasks, and answer all of the written questions. Place all of your game-playing code as indicated in *agents.py*. Add your written answers as indicated in *hw2_answers.py* based on the question number listed for each question. Submit both files on Sakai.

1.1 Homework Policies

Homework and lab assignments **must be completed individually**. Students are permitted and even encouraged to discuss assignments. However, any attempt to duplicate work that is not your own – for example, in the form of detailed written notes, copied code, or seeking answers from online sources – is strictly prohibited and will be considered cheating.

Homework assignments are due by the end of class on the due date unless otherwise specified. **No extensions will be granted** except for extenuating circumstances. Extensions are granted at the instructor's discretion, and valid extenuating circumstances include, for example, a debilitating illness (with STINF), death in the family, or travel for varsity athletics. Extensions will not be granted for personal or conference travel, job interviews, or a heavy course load.

2 Minecraft Pickup

In this homework, you will implement strategies for a game-playing agent for an item pickup game built in Minecraft. The game is represented as a $3x5$ board where each square is a location that can be occupied by an agent, a cookie, or nothing. Agents collect cookies by moving onto a square which contains a cookie and their goal is to collect more than their opponent before time runs out. An agent cannot move to a square that is currently occupied by the other agent.

For technical reasons, the game is implemented as a larger platform with the cookies spaced further apart. However, this is abstracted away by representing the state as a $3x5$ grid and increasing the distance traveled when taking actions.

3 Programming

The *hw2.zip* archive contains this file, four Python files: *hw2.py*, *agents.py*, *game.py*, and *hw2_answers.py*, and an xml file: *hw2.xml*.

3.1 Running the program

In order to run the game-playing program, you must extract the provided Python files into a working directory of your choosing and copy the *MalmoPython.lib* and *MalmoPython.pyd* (Windows) or *MalmoPython.so* (OS X, Linux) files from the Project Malmo *Python_Examples* directory into your working directory.

Additionally, you must ensure that two separate Project Malmo Minecraft clients are already running. This is done much the same as you did in HW0 and HW1, but you must start each one from a separate terminal. Once the clients are running, you can start the program with the following command (in a third terminal):

- `python hw2.py <agent0type> <agent1type> <agent0alg> <agent1alg>`

where **¡agent0type¿** and **¡agent1type¿** are placeholders where you specify which class of agent will be used for each player - either *basic* (for the provided basic player) or *student* (for your student player) and where **¡agent0alg¿** and **¡agent1alg¿** are placeholders where you specify which algorithm each agent should use - either *minimax* or *alphabeta*.

4 Problems

4.1 Implement the evaluation function

Programming For this question, you will implement the evaluation function, *evaluate* in your *StudentPlayer* class in *agents.py*. Given a game state (see the *PickupState* class in *game.py*), this function must return a numeric value which indicates how "favorable" the game state current player. A value greater than 0 indicates that the state is favorable to the current player, less than 0 is favorable to the opposing player, and 0 indicates a neutral state with no advantage for either player. The game state consists of a 3x5 grid implemented as a list of lists of strings where the elements of the grid indicate what, if any, entity is present at those coordinates. The possible values and the entities they represent are:

- " (empty string) - no entity
- '0' - agent 0
- '1' - agent 1
- 'c' - cookie

The agents start in specific squares and the remainder initially contain cookies.

Written Questions

4.1.1 How did you come up with your evaluation function? Why is it a good measure how favorable the state is to the players?

4.1.2 What is the size of the search space? In other words, how many distinct possible states exist?

4.2 Implement Minimax Search

Programming In this problem, you will finish implementing minimax search in the *minimax_move* method of your *StudentPlayer* class by completing the *minimax_dfs* function. This function should return a (value, action) tuple for minimax search up to the given depth.

Given a current game state, *minimax_move* should search the game tree up to some depth and use the minimax backup procedure to back up the *evaluate* function from the leaf nodes up to the root. The return value is an action string indicating which action the current player should take.

The valid actions in this environment are:

- "n" - move one "block" north
- "s" - move one "block" south
- "e" - move one "block" east
- "w" - move one "block" west
- "z" - stay put

Notes:

- Your search should work whether the move is being computed for MIN or MAX.
- A successor function is already implemented in the *PickupGame.get_successors* function. An instance of this class is provided to your *BasicPlayer* and stored in *self.game*.

Written Questions

4.2.1 If you could expand the entire game tree for the pickup game, what could you conclude from the backed up value at the root node?

4.3 Implement Alpha-Beta Pruning

Programming For this problem, you will implement the alpha-beta pruning algorithm in the *alpha_beta_move* method of your *StudentPlayer* class. Given a game state, this method should perform a minimax search with alpha-beta pruning and return an action string indicating what action the current player should take. In other words, this method should work like the *minimax_move* method, but with alpha-beta pruning. You may create a helper function (like *minimax_dfs*), if needed.

Written Questions

4.3.1 Play your alpha-beta player against your minimax player. What do you observe?

4.3.2 Compare standard minimax and alpha-beta pruning in terms of how many search nodes they expand while playing the game. You might need to change your code to track this.