B551 Homework 2

Assigned: Sept. 15, 2011
Due: Sept. 29, 2011

1 Directions

The problems below will ask you to implement three strategies for a game-playing agent for the Gobblet Gobblers game demonstrated in class. You will need to:

- Rename the players/oware/username.py file to your IU user ID.
- Program your strategies in the indicated functions in that file.
  - A state evaluation function giving a minimax value for a game state
  - A minimax search procedure
  - An alpha-beta search procedure
  - A search procedure to be used in a game-playing tournament pitting all students' submissions against each other.
- Answer the written questions below.

See README.txt for a complete description of the game framework and the role of the file you are modifying.

1.1 Submission

Upload your [userID].py file and any auxiliary files (additional Python modules or data files you may have used to implement your strategies) to OnCourse. Do not upload modified versions of any of the other provided Python files (in fact, don't modify them at all; when we grade the homework we will use the canonical versions and your changes will not be reflected). Type or write your answers to the written questions and hand them in, hard-copy, by the end of class on the due date.

2 The Game

For this homework we are playing a two-player game called "Gobblet Gobblers". It is a variant of tic-tac-toe (or noughts and crosses) with the addition of sized pieces and the ability to move pieces that are already on the board. Larger pieces can "swallow" and conceal smaller pieces, either the player's own or the opposing player's. The smaller pieces remain on the board, under the larger ones, and are revealed if the larger pieces are moved. When computing the three-in-a-row objective, only the largest piece in a square is considered.

The game's complete rules are online and may be viewed at *http://www.blueorangegames.com/instructions.php*.

As the game is capable of cycling (repeating states), we introduce a new rule for our implementation: If a repeated state occurs, the game is declared a draw between the two players. We also adopt the convention that the blue player always moves first and the orange second.

3 Programming

3.1 Setup

1. Download the file hw2.zip from OnCourse. The file contains a directory structure, with base class definitions for a game-playing framework in the files *game.py*, *game_controller.py*, and *game_state.py*. Game definitions for tic-tac-toe and Gobblet Gobblers are in *tictactoe.py* and *gobblet.py*. Some sample game players are defined in the files in the *players* directory.

   For more information about these files and their roles, see README.txt and the files' documentation. You are principally interested in the *username.py* file in the *players/gobblet* subdirectory, which defines an incomplete Gobblet Gobblers minimax player. You will complete the definition of this player to allow it to play the game.

2. Rename *players/gobblet/username.py* so that the filename is your IU user ID (your email address). For instance, Mark's ID is mw54, so he would rename the file to *mw54.py* in the same directory. This is the file you will alter and upload to OnCourse.

3. You may run the program with this command from the Windows command line (assuming the Python executable is in your PATH):
   python game.py gobblet gobblet_simple gobblet_simple

Or, on any UNIX/Mac system's command line:
./game.py gobblet gobblet_simple gobblet_simple

The second and third arguments to the command specify the player modules (Python files) to use for the game; the first specifies the game definition we use. *gobblet_simple.py* resides in the same directory as your Gobblet agent and defines a not-very-smart Gobblet player: It simply selects the first move from the list of successors generated from the current game state. You can also use *gobblet_human* as one or both players to allow humans to play the game.

See README.txt and the Python files' documentation for more information on running the program, or execute:
./game.py -h

3.2 Programming Problems

1. **Evaluation function for the Gobblers game**

   The evaluation function, *evaluate()*, in your GobbletPlayer class is incomplete. Given a game state, the function should return a numeric value indicating how "favorable" a game state is to the MIN and MAX agents. (A value greater than 0 indicates a state favorable to MAX; less than 0, favorable to MIN; exactly 0, a neutral state with no advantage for either player.)

   Implement the evaluation function to return such a numeric value.

   Notes:

   - You may wish to consult the definition of the GobbletState class (the type of object you'll be evaluating) in *gobblet.py*.
   - A popular evaluation system for tic-tac-toe boards is to compare the number of three-in-a-row positions that have not been blocked off by the opposing player (and are therefore "available" to the player under consideration). This system is implemented in the *tictactoe_adv.py* player in the *players/tictactoe* directory, which you can feel free to use as a reference.
   - However, as pieces can be moved in this game and larger pieces may contain smaller pieces belonging to the other player, the Gobblet Gobblers game is more fluid than the above evaluation suggests. It is strongly suggested to incorporate other features, taking into account available and concealed pieces, in your evaluation.

2. **Minimax procedure**

   The minimax function, *minimax_move()*, in your GobbletPlayer class is incomplete. Given a game state, the procedure should execute a minimax search, use the *evaluate()* function as appropriate, and return a GobbletMove object representing the best move for the current player. (The current player is arbitrary, according to how the game is being played, and may be obtained with the provided state's *get_next_player()* function.

   You do not need to write a successor function for this assignment – one is already provided in the definition of the GobbletState class, and you can obtain successors for any state object by calling its *successors()* function. However, when executing the search, the player may only execute a set number of expansions – i.e., you may only call *GobbletState.successors()* a certain number of times per turn before it will stop returning moves. (We use this rather than time to limit players' search depth in the tournament, as time is unreliable and depends on the machine's load among other factors.) You may obtain the available number of expansions for your current move by calling the provided state's *expansions_count()* function. Obviously, you should stick to calling the provided successor function and not circumvent the limitation on expansions by writing a separate one of your own.

   In your minimax function, compute a horizon h and use it to limit the depth of your search so that you don't run out of expansions.

   Notes:

   - Again, feel free to consult the Tic-Tac-Toe example player. However, be careful about your math when computing the search horizon – the Tic-Tac-Toe example may not be correct!
   - Optionally, you may want to look into the *mirror()* and *rotations()* functions in *gobblet.py*.

3. **Alpha-beta procedure**

   The alpha-beta function, *alpha_beta_move()*, in your GobbletPlayer class is incomplete. Given a game state, the procedure should execute a minimax search with alpha-beta pruning and return a GobbletMove object representing the best move for the current player.

   This function should do all the same things as *minimax_move()*, but with alpha-beta pruning.

Notes:

- Again, you must compute a reasonable horizon h – but the horizon may be different for alpha-beta than for a simple minimax search.

4. **Tournament procedure**

   The search function used during the multi-agent tournament, *tournament_move()*, in your GobbletPlayer class is incomplete. Given a game state, the procedure should execute some search routine and return a GobbletMove object representing the best move for the current player.

   This function may simply call the minimax or alpha-beta search routines; or you may enhance it with advanced techniques such as singular extension, multiple extension, best-move databases, etc.

   Extra credit will be awarded for the agent that wins the tournament by winning the most games. We will also award extra credit for creative implementations of the tournament search.

3.3 Written Questions

Answer the following written questions, typed or written, and hand them in in hard-copy format.

1. Explain how you devised your state evaluation function, and why you think it is a good measure of a state's favorability to the players.

2. Given a default limit on the number of expansions, M, what is the largest horizon value h such that your minimax routine is *guaranteed* to expand a complete tree to that depth (i.e., all branches of the tree would unquestionably go that deep before you ran out of expansions)? Why?

3. Perform some experiments to see what is the largest horizon you can set without encountering the default limit on expansions with your alpha-beta search. (Hint: You may need to use some temporary print statements for this part.) Report on your findings.

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