A5: Toro-Tile-Straight Agents

CSE 415: Introduction to Artificial Intelligence
The University of Washington, Seattle, Autumn 2019

Toro-Tile-Straight Agent.

This continues the search theme of Assignments 2 and 3 and has the same associated reading: Chapter 5 (Search) of *Introduction to Artificial Intelligence Using Python*.

Due Friday, November 8 via Canvas at 11:59 PM.

Overview

In this assignment we explore two-person, zero-sum game playing using a family of games called "Toro-Tile Straight". Here we put our agents into competition, adding lookahead (with the Minimax technique) and pruning (with the alpha-beta method) to the search. The assignment has two parts: creating your agent and engaging your agent in a first round of competitive play.

Here is the starter code for this assignment.

PART I: Creating a Game-Playing Agent (80 points).

Create a program implementing an agent that can participate in a game of Toro-Tile Straight (defined below). Your program should consist of a single file, with a name of the form [UWNetID]_TTS_agent.py, where [UWNetID] is your own UWNetID. For example, my file would have tanimoto_TTS_agent.py for its name.

TTS Game Rules

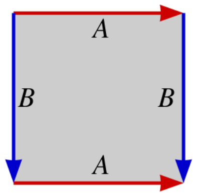
Although you create one Python file for this assignment, it will contain a collection of specific functions for playing games of "Toro-Tile Straight". We define a Toro-Tile Straight game as a kind of generalized K-in-a-Row with the following features:

- (a) Just as in a game like Othello, there are two players: one plays White and the other plays Black;
- (b) the board is rectangular, but is not necessarily 3 by 3; it is mRows by $m \times n$ nColumns, where these dimensions are chosen by the Game Master (referee) at

the beginning of the game;

- (c) a player wins by getting K in a row, where K is not necessarily 3; K can be any integer greater than 1;

- (d) the topology of the board is *toroidal*. This means that the left edge and right edge of the board can be considered equivalent, and the top and bottom edges can be considered equivalent. If a piece is placed in the left column of the array, then moving one unit in the West direction is defined to "wrap around" to the rightmost column (but same row). Similarly, moving South from the bottom row, wraps around to the top of the same column.



The toroidal topology is illustrated above, with line segments A unified, as well as line segments B unified. These are at the outer boundaries of the array.

In a discrete space, lines of tiles just "wrap around" when they go off the edges of the board. Here is an illustration of the ways to get 3 in a row on a 3 by 3 board. Note that the first 8 are the familiar Tic-Tac-Toe wins, but that there are four more, due to the toroidal topology.

```
W W W   - - -   - - -
- - -   W W W   - - -
- - -   - - -   W W W

W - -   - W -   - - W
W - -   - W -   - - W
W - -   - W -   - - W

W - -   - - W
- W -   - W -
- - W   W - -

- W -   - W -   W - -   - - W
W - -   - - W   - - W   W - -
- - W   W - -   - W -   - W -
```

- (e) there can be "forbidden squares" on the board; these are chosen at the beginning of the game by the Game Master; a square on the board that is available is represented by a blank, whereas a forbidden square is represented by a dash "-" ;

- (f) there can be "handicaps" in the initial state, meaning that some White and/or Black tiles can be set up on the board by the ~~Game Master~~ in order either to influence the succeeding play or to change the balance of advantage and disadvantage to the players.

You can see a full transcript of a sample game here. The two agents that played this game are very dumb, but they followed the rules. The code for each of these agents is part of the starter code. You can use this code to organize the required functions, adding to them to create a good player.

It will be essential that your program use the provided representation of a game state, so that it is compatible with all the other A5 TTS game agents developed in the class.

Turn Time Limit

Your program should be designed to anticipate time limits on moves. There are two aspects to this: (1) use iterative deepening search, and (2) poll a clock frequently in order to return a move before time runs out.

An agent that goes over time automatically forfeits the game, and if this happens during grading your agent will lose points.

Agent Utterances

In addition to being able to play the game, your program should make an utterance -- that is, a comment in each move, as if participating in a dialog. Ideally, your program would have a well-defined "personality". Some examples of possible personalities are these: friendly; harmless joker; blunt joker; paranoid; wisecracker; sage; geek; wimp; competitive freak; fortune-teller (based on the state of the game). The personality will be revealed during games via the "utterances" made by the program. (For more details, see the description of the take_turn function below.)

Required Functions

Your program must include the following functions. You can have helper functions if you like. Please keep all the functions required by your player in just one Python file that follows the naming convention mentioned earlier. For example, my player would be in a file tanimoto_TTS_agent.py. This will facilitate your player's being part of the class tournament. To help you know if your agent function definitions are correct we have shared the agent_checker.py file in the starter code. Note that this is not an autograder and does not guarantee your agent will get full points.

1. `parameterized_minimax(current_state=None, max_ply=2, use_alpha_beta=False, use_basic_static_eval=True)`
   This method will let both you and the graders execute a search under a specific

set of conditions without having to set up a full TTS game. You can assume this function will only ever be called after get_ready(), though not during the course of a normal game. This method accepts a set of parameters that determine how the minimax search will run. Here are what the various arguments mean:

- **current_state**=None: This field will be assigned a valid TTS state that should be used as the root node for minimax search that will occur in this method function call.
- **max_ply**=2: This field determines the maximim allowed depth of your minimax search. This should limit the search regardless if iterative deepening is enabled or not.
- **use_alpha_beta**=False: This field determines if the search should use alpha-beta pruning or not.
- **use_basic_static_eval_function**=True: This field determines if you should use the basic eval (described below) or your own custom eval function during the search.

You should use a "standard" move generation method within `parameterized_minimax`
That method is to order available spots on the board from **left-to-right, top-to-bottom**. That is, if your board is an empty 2x2 grid, the move generation method should generate [upper-left, upper-right, lower-left, lower-right] in that order. This is necessary for us to consistently test your pruning algorithm.

Within `parameterized_minimax`, do not use iterative deepening.

When this method ends its search it should return a `dict` of attribute-value pairs giving info related to how it executed the search.

- 'CURRENT_STATE_STATIC_VAL': The static eval value of the current_state as determined by your minimax search
- 'N_STATES_EXPANDED': The number of states expanded as part of your minimax search
- 'N_STATIC_EVALS': The number of static evals performed as part of your minimax search
- 'N_CUTOFFS': The number of cutoffs that occurred during the minimax search (0 if alpha-beta was not enabled)

It is important that you implement this function in a manner consistent with the options you are handling, particularly the more advanced features such as alpha-beta pruning. Certain basic features of your code may also be tested using this function, and so you'll want to handle this well, both for testing your own code as you go and to allow any autograder system to award you as many points as possible.

2. **`get_ready(initial_state, k, what_side_i_play, opponent_moniker)`**
   This function takes four arguments and it should "remember" these values for the game that is about to be played. (However, if your agent is in a match with itself say for testing purposes, this get_ready method will be called twice. In this case, be careful not to let the agent assume it is playing 'B' on both turns.)

   - **initial_state**: Allows your agent to figure out any needed properties of the game board before the playing begins. This will be a legal game state that can be used by your player, for example, to determine the dimensions of the board, the locations of forbidden squares, and even the locations of any handicap items.
   - **k**: The number of pieces in a row (or column or diagonal) needed to win the game.
   - **what_side_i_play**: This will be 'W' if your agent will play as White; it is 'B' if your agent will play Black.
   - **opponent_moniker**: Allows your utterance-generation mechanism to refer to the opponent by name, from time to time, if desired.

   Note that your program does not really have to do much at all when its get_ready method is called. The main thing it should do is return "OK". However, the **get_ready** function offers your agent an opportunity to do any initialization of tables or other structures without the "clock running." This could be used to do any preprocessing related to, say, speeding up move generation, e.g., finding out which of the four directions a win can occur from a particular square -- a list of all the squares on the board where such a winning line could actually start. Having these lists can save time in your static evaluation function.

3. **`who_am_i()`**
   This function will return a multiline string that introduces your player, giving its full name (you get to make that up), the name and UWNetID of its creator (you), and some words to describe its character.

4. **`moniker()`**
   This function should return a short version of the playing agent's name (16 characters or fewer). This name will be used to identify the player's moves in game transcripts.

5. **`take_turn(current_state, opponents_utterance, time_limit=3)`**
   This is probably your most important function. Unlike the parameterized_minimax function described ~~below~~ above, this method should always use your most advanced search techniques (while still respecting the time limit) to make your agent as competitive as possible. Do feel free to invoke parameterized_minimax in this function or use some of the same code, but be careful of the time limit that is not in parameterized_minimax. take_turn should return a list of the form `[[move, new_state], new_utterance]`.

   - **move** is a data item describing the chosen move.
   - **new_state** is the result of making the move from the given current_state. It must be a complete state and not just a board.
   - new **~~opponents~~_utterance** is a string representing a remark from the opponent on its last move.

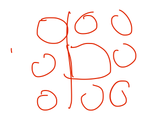
   The time_limit represents the number of **seconds** available for computing and returning the move.

   The new_utterance to be returned must be a string. During a game, the strings from your agent and its opponent comprise a dialog. Your agent must make "interesting" utterances, described in more detail in the feature section.

6. **Static Eval**
   This will be represented by two functions that will perform a static evaluation of the given state. The value returned should be a number that is high if the state is good for White and low if the state is good for Black. See the MY_TTS_State class provided in the starter code in PlayerSkeleton.py for where to invoke the two required versions of this. Here is a description of these two versions:

   ✓ a. `basic_static_eval(self)`
      This is probably the very first function you should implement.

      It must compute the following value accurately (whereas with your custom function, you'll get to design the function to be computed). Depending on the game type, there will be a particular value of K, the number of tiles a player must place in a straight line in order to win.



      We define a "piece" on the board to be either a W tile or a B tile. We define a "freedom" of a tile t on the board to be any unoccupied square that is one of t's (up to eight) neighbors. A neighbor of t touches t either along an edge or at a corner of the tile. Each piece has some number of freedoms, in the range 0 to 8. Let TWF be the total of the numbers of freedoms of all the W tiles on the board, and let TBF be the total of the numbers of freedoms of all the B tiles. Then our basic static value is just TWF - TBF. If White has more total freedoms than Black does, then the value will be positive, etc.

An autograder will be checking to make sure your function comes up with the right numbers for various given states.

b. `custom_static_eval(self)`
You get to design this function. You'll want it to perform "better" than the basic function above. Here better could mean any of the following: (a) more accurate assessment of the value of the board, to better inform the choice of the best move, (b) faster to compute, or (c) achieving a better combination of accuracy and efficiency than the basic function. (It might be less efficient, but a lot more accurate, for example.)

Two motivations for putting some thought into this function are: (i) your agent's ability to play well depends on this function, and (ii) an autograder will likely try comparing your function's values on various states, looking for reasonable changes in value as a function of how good the states are, in terms of likelihood of a win.

Potentially Useful Python Technique

When one of the game master programs calls your take_turn function or a testing program calls your `parameterized_minimax` function, you may convert the current_state object into your own derived subclass object as follows, assuming you are using the My_TTS_State class:

```
current_state.__class__ = My_TTS_State
```

Then you will be able to invoke your own methods on it, such as static_eval.

Required Features

To reiterate the above requirements in terms of features we expect to see in your agent, the following features must be implemented and demonstrated somewhere in your agent code.

1. Minimax search: The number of ply to use should either be a parameter (if part of parameterized_minmax) or determined by you according to preference or time limit requirements.

2. Iterative deepening: You should start your time-limited algorithm by finding a legal move quickly, and then as time permits, finding better and better moves. When there is no longer enough time to safely compute a better move, then the best move found so far should be returned. You will do this by running your minimax search to deeper and deeper levels. However, note that the parameterized_minimax function has no time limit and so doesn't need to use iterative deepening. Also note that the autograder may test for times lower than the default 10 seconds, though you can assume the time limit will never be below 1 second.

3. Utterances: program your agent in such a way that it reveals a personality through its utterances. The utterances should not repeat often. Let us say 10 is the minimum number of turns, before one of these utterances can be repeated. Here is an example of this type of utterance: "Although I hate waiting in lines, I'm going to make a nice long one!" (The personality here is one of a joker.)

4. Alpha-beta pruning: Alpha-beta pruning should be implemented as part of your minimax search that happens during take_turn and also as a search option in parameterized_minimax according the algorithm described in lecture.

   This feature will be autograded, in which case the move-generation order will need to be consistent. Your pruning should work with minimax search using the basic eval function and the simple move generation order described above.

5. Basic Competency: After implementing minimax and the basic_static_eval you may notice that your agent still doesn't win most games against even the "dumb" agents. This is to be expected as the basic eval is not very good. Your agent, using your custom static eval, should be able to win on most boards against the dumb agents however. This also means your agent should not time out on the larger boards supplied in the starter code.

PART II: Game Transcript (20 points).

Follow the directions below to produce a transcript of a match between your agent and another student's agent.

Using the timed_tts_game_master.py program to run a Gold Rush match, create a transcript of a game between your agent and the agent of another student in the class. Set up the match so that your agent plays White and your opponent plays Black. Use the following game instance for your match. For this run, you should set a time limit of 1.00 seconds per move. This can be specified on the command line when you run timed_tts_game_master.py.

The following is the representation of the initial board in a TTS game we are calling "Gold Rush" in which the objective is to get a (connected) line segment of five tiles. The line may go horizontally, vertically, or diagonally.

```
[[['-',' ',' ','-',' ',' ','-'],
  [' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' '],
  ['-',' ',' ','-',' ',' ','-'],
  [' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' '],
  ['-',' ',' ','-',' ',' ','-']], "W"]
```

Note that in the following state, White has just won, because there is a diagonal line of Ws that starts near the lower left corner of the space, and continues up and to the right, wrapping around twice in the toroidal space (once from right to left, and once from top to bottom). With the upper-left square (which happens to be forbidden in this game) at row 0, column 0, we can describe White's winning line as this sequence of locations: [(3,5), (2,6), (1,0), (0, 1), (6, 2)]

```
      0   1   W2  3   4   5   6          W
0 [[['-','W',' ','-',' ',' ','-'],     W
1   ['W',' ',' ',' ',' ',' ','B'],W
2   [' ',' ',' ',' ','B','B','W'],
3   ['-',' ',' ','-',' ','W','-'],
4   [' ',' ',' ',' ',' ',' ',' '],
5   [' ',' ','B',' ',' ',' ',' '],
6   ['-',' ','W','-',' ',' ','-']], "B"]
```

It is not required that the line of tiles involve the toroidal wraparound. Five in a line somewhere near the middle of the board could also win, for example. However, the wraparound feature can often help to open up many more possibilities for winning.

The timed_tts_game_master program will automatically generate an HTML file containing a formatted game transcript of your game. Turn in that HTML file. Note that your agent doesn't have to win this game, we are just asking for a demonstration that your agent works.

Competition Option

You can earn extra credit by having your agent participate in a competition. To enter the competition, first your agent must qualify, and then you submit your evidence of qualification. For your agent to qualify, you must submit game transcripts for two games that meet the following criteria.

1. Your agent must be the winner of each of these two games.
2. Each game must be between your agent and a different opponent. These should be "final" or almost-final versions of the opponent agents, without any intentional handicapping to permit an easy win. If there were a rematch between your agent and one of these opponents, we would expect your agent to win again, or at least not lose.

3. The games must involve a version of Toro-Tile Straight in which (a) K is at least 5 and the board is at least 7 by 7, but it can be bigger if you wish. (I.e., you may use the Whites_Challenge_Game_Type or the Gold_Rush_Game_Type).
4. The games must be run with a time limit of 2.0 seconds per move for each player.
5. If you submit two game transcript files, then we will assume that you are saying that your agent has won both games, and it therefore qualifies for the competition.

If you enter your agent and it does qualify, you will automatically get 5 more points of extra credit, even if your agent doesn't win games against any more agents. If your agent is a finalist (one of the top 20 percent of the qualifying contestants), you'll get 5 more points of extra credit. Finally, if your agent is the overall winner, you'll receive five additional points for a total of 15 points for participating in and winning the competition.

What to Turn In

1^{o}

Turn in at least one Python file (the one for your agent). If you have created one or two new Python files that your agent needs to import, include those as well (named properly). Do not turn in any of the starter code files. Turn in either one or two game files (Two if your agent has qualified and you are entering it in the competition.) The game files are the HTML files that are automatically created when you match up your player with the opponent. 2^{o}

Updates and Corrections:

If necessary, updates and corrections will be posted here and/or mentioned in class, or our ED forum.