Assignment 5: Game-Playing Agents

CSE 415: Introduction to Artificial Intelligence
The University of Washington, Seattle, Spring 2017

This continues the search theme of Assignments 3 and 4 and has the same associated reading: Chapter 5 (Search) of *Introduction to Artificial Intelligence Using Python*.

Due Monday, May 8 via Catalyst CollectIt at 11:59 PM.

Partnership Policy.

This assignment may be done in partnerships of two or individually. A partnership creates one Baroque-Chess player program. An individual creates one K-in-a-Row player program. Whether or not you are in a partnership, you should cooperate with another individual or team to produce a transcript of a game match between your player and another. (Although matches between your player and itself are allowed, they will only earn 5 of the 20 possible points for Part II.)

What to Do.

In this assignment we explore two-person, zero-sum game playing using your choice of (A) a game called Baroque Chess or (B) a family of games called "K-in-Row with Forbidden Squares." Here we put our agents into competition, adding lookahead (with the Minimax technique) and, optionally, 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 competition play.

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

Option A: Partnership and Baroque Chess

For option A, you will work with a partner to create a Python module (typically just a single .py file) that will be able to play a game of Baroque Chess and compete in a tournament. Although the tournament is not part of the assignment itself, your program must be technically able to participate in it. (The tournament will take place after the assignment is due; we may offer some extra credit for programs that place highly in the rankings.) Your player's file name should be of the form [free_name]_BC_Player.py, where you get to choose an original name, but that name is followed by "_BC_Player.py". An example might be Magnifico_BC_Player.py. If you need to split your agent to use any additional modules, please name them using the convention: [free_name]_BC_module_[module_name].py. Here an example could be Magnifico_BC_module_staticevaluation.py.

It will be essential that your program use the specified representation of states, so that it is compatible with all the other Baroque Chess programs developed in the class. Here is some code for representing states of Baroque Chess.

```
BLACK = 0
WHITE = 1

INIT_TO_CODE = {'p':2, 'P':3, 'c':4, 'C':5, 'l':6, 'L':7, 'i':8, 'I':9,
  'w':10, 'W':11, 'k':12, 'K':13, 'f':14, 'F':15, '-':0}

CODE_TO_INIT = {0:'-',2:'p',3:'P',4:'c',5:'C',6:'l',7:'L',8:'i',9:'I',
  10:'w',11:'W',12:'k',13:'K',14:'f',15:'F'}

def who(piece): return piece % 2
```

```
def parse(bs): # bs is board string
  '''Translate a board string into the list of lists representation.'''
  b = [[0,0,0,0,0,0,0,0] for r in range(8)]
  rs9 = bs.split("\n")
  rs8 = rs9[1:] # eliminate the empty first item.
  for iy in range(8):
    rss = rs8[iy].split(' ');
    for jx in range(8):
      b[iy][jx] = INIT_TO_CODE[rss[jx]]
  return b

INITIAL = parse('''
c l i w k i l f
p p p p p p p p
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
P P P P P P P P
F L I W K I L C
''')


class BC_state:
  def __init__(self, old_board=INITIAL, whose_move=WHITE):
    new_board = [r[:] for r in old_board]
    self.board = new_board
    self.whose_move = whose_move;

  def __repr__(self):
    s = ''
    for r in range(8):
      for c in range(8):
        s += CODE_TO_INIT[self.board[r][c]] + " "
      s += "\n"
    if self.whose_move==WHITE: s += "WHITE's move"
    else: s += "BLACK's move"
    s += "\n"
    return s

def test_starting_board():
  init_state = BC_state(INITIAL, WHITE)
  print(init_state)

test_starting_board()
```

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.

Starter code is available. This code will help you debug your move generators and handle the turn-taking between a pair of playing agents. Note that this code archive includes a subdirectory `__pycache__`. This contains two compiled versions of a file `new_succ.pyc`. One has a filename ending in 34 (for Python 3.4) and the other has a filename ending in 36 (for Python 3.6). Depending on which version of Python you are using, copy either the "34" version or the "36" version to the main folder (parent of the cache folder) as `new_succ.pyc`. Then it will be possible to import this file into your session.

We are playing Baroque Chess with rules that: do not permit (a) any choice of center-counter symmetry (see the Wikipedia description of Baroque Chess), or (b) long-leaper double jumping (it's considered detrimental to having balanced and dynamic games), or (c) "suicide" moves. We will assume that the game ends when either player loses a king.

Your program should implement the following functions, so that they can be called by the game administration software:

- **introduce()**. This function will return a multiline string that introduces your player, giving its full name (you get to make that up), the names and UWNetIDs of its creators (you), and (optionally) some words to describe its character.

- **nickname()**. 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.

- **makeMove(currentState, currentRemark, timeLimit=10000)**. This is probably your most important function. It should return a list of the form [[move, newState], newRemark]. The move is a data item describing the chosen move, and you may choose to return the empty string for this "". (It is included for compatibility with possible future versions of this assignment.) The newState is the result of making the move from the given currentState. It must be a complete state and not just a board. The currentRemark argument is a string representing a remark from the opponent on its last move. (This may be ignored, and it's a placeholder for possible use in future versions of this assignment.) The timeLimit represents the number of milliseconds available for computing and returning the move.

The newRemark to be returned must be a string. During a game, the strings from your agent and its opponent comprise a dialog. (However, you may simply return a fixed string, such as "Your move!" For extra credit, you may make this more elaborate and context-sensitive, commenting on the current state or the direction in which the game seems to be heading.)

- **staticEval(state)**. This function will perform a static evaluation of the given state. The value returned should be high if the state is good for WHITE and low if the state is good for BLACK. Although you may wish to extend the BC_state class to make staticEval a member of that, it's not necessary, and we do need to be able to call your staticEval function directly, for example using

```
import The_Roman_BC_Player as player
staticResult = player.stateEval(some_state)
```

A portion (approximately 20 points) of your grade for Part I will depend on how well your staticEval function works.

In the code example above, the starting board is shown using ASCII text, and the encoding is as follows: (lower case for black, upper case for WHITE):

```
p: pincer
l: leaper
i: imitator
w: withdrawer
k: king
c: coordinator
f: freezer
-: empty square on the board
```

Option B: Individual Work and K-in-a-Row

Create a program implementing an agent that can participate in a game of K-in-a-Row with Forbidden Squares (defined below). Your program should consist of a single file, with a name of the form [UWNetID]KInARow.py, where [UWNetID] is your own UWNetID. For example, my file would have tanimotoKInARow.py for its name.

For this option, you will create a program -- consisting mainly of a collection of specific functions, for playing games of "K in a Row". We define a K in a Row game as a kind of generalized Tic-Tac-Toe with the following features:

- (a) Just as in Tic-Tac-Toe, there are two players: one plays X and the other plays O;
- (b) the board is rectangular, but is not necessarily 3 by 3; it is mRows by nColumns, where these

numbers 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 and less than or equal to the maximum of mRows and nColumns;
- (d) 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 "-" ;
- (e) there can be "handicaps" in the initial state, meaning that some X and/or O tokens 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.

In addition to being able to play the game, your program should make 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 makeMove function below.)

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 tanimotoKInARow.py. This will facilitate your player's being part of the class tournament.

1. **prepare(initial_state, k, what_side_I_play, opponent_nickname).** This function takes four arguments and it should "remember" these values for the game that is about to be played.

   The first parameter, initial_state, allows your agent to figure out any needed properties of the game board before the playing begins. It is 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. The second parameter, k, is the number of pieces in a row (or column or diagonal) needed to win the game.

   The parameter **what_side_I_play** is 'X' if your agent will play as X; it is 'O' if your agent will play O.

   The parameter **opponent_nickname** 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 prepare method is called. The main thing it should do is return "OK". However, the **prepare** function offers your agent an opportunity to do any initialization of tables or other structures without the "clock running." This is good for setting up for, say, Zobrist hashing, if you are using that. Another kind of preprocessing would be to make, for each of the four directions in which a win can occur, 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.

2. **introduce()**. 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.

3. **nickname()**. 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.

4. **makeMove(currentState, currentRemark, timeLimit=10000)**. This is probably your most important function. It should return a list of the form [[move, newState], newRemark]. The move is a data item describing the chosen move, and its format is the same as that used in Assignment 3. The newState is the result of making the move from the given currentState. It must be a complete state and not just a board. The currentRemark argument is a string representing a remark from the opponent on its last move. The timeLimit represents the number of milliseconds available for computing and

returning the move.

The newRemark to be returned must be a string. During a game, the strings from your agent and its opponent comprise a dialog. Your agent might contribute to this dialog in three ways:

1. by convincingly representing the character that you have chosen or designed for your agent,
2. by showing awareness of the game state and game dynamics (changes in the game state), and
3. by responding in a convincing way to the opponent's remarks.

Extra credit will be based on how well your agent's remarks meet these criteria. To get the extra credit, implement (1) and one, the other, or both, of (2) and (3) and then describe, in a separate file called ExtraCredit.txt what the features are and how they work. Something working that adds in a noticeable way to the dialog is worth 5. A more thorough implementation that shows considertion for multiple features related to one of these two latter criteria is worth up to 10 additional points. The maximum extra credit here is therefore 15 points.

5. **staticEval(state)**. This function will perform a static evaluation of the given state. The value returned should be high if the state is good for X and low if the state is good for O.

   A portion (approximately 20 points) of your grade for Part I will depend on how well your staticEval function works.

PART II: Game Transcript (20 points).

If you are doing option A, use the BaroqueGameMaster.py program to create your Baroque Chess demonstration game. If you need to adjust the time limit for each move, you are free to do that to create your demonstration game.

If you are doing option B, following the directions below to run your K-in-a-Row demonstration game.

Using the TimedGameMaster.py program to run a K-in-a-Row 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 'X' and your opponent plays 'O'. Use the following game instance for your match. Your player should play X and the opponent should play O. For this run, you should set a time limit of 1.00 seconds per move (e.g., by editing the appropriate line of code near the top of the TimedGameMaster.py program).

Here is the starter code for this assignment (Option B)

The following is the representation of the initial board in a KinARow game called "5 in a Row on 7-by-7 board with Corners Forbidden".

```
[[['-',' ',' ',' ',' ',' ','-'],
  [' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' '],
  [' ',' ',' ',' ',' ',' ',' '],
  ['-',' ',' ',' ',' ',' ','-']], "X"]
```

The TimedGameMaster program will automatically generate an HTML file containing a formatted game transcript of your game. Turn in that HTML file.

Competition Option.

For either Option A or Option B, you may also choose to take part 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. Each game must be between your agent and a different opponent.
2. Your agent must be the winner of each of these two games.
3. If you are doing Option B, the games must involve a version of K-in-a-Row 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.

If you enter your agent and supply this qualifying information, you will automatically get 5 more points of extra credit, even if your agent doesn't win any more games. If your agent is a finalist (one of the top 3 agents in its option category), you'll get 5 more points of extra credit. Finally, if your agent is the overall winner in its category, you'll receive five additional points or a total of 15 points for participating in and winning the competition.

What to Turn In.

For Option A, turn in your Baroque Chess agent files, named according to the guidelines, via Catalyst CollectIt. Also, turn in an ASCII file with a name such as Magnifico_vs_Goliath.txt, that represents the printed transcript of a game between your agent and another team's. We assume here that Magnifico was playing WHITE, because it is first in the name.

For Option B, turn in two (or three) files: your player program, which should be a single python file, and your game file, which is the HTML file that is automatically created when you match up your player with the opponent. For your player file use the first partner's UWNetID (or your own if you are working alone). The file name should follow this pattern: **[YourUWNetID]KInARow.py** If you are doing the extra credit items in Part I, then also turn in your file ExtraCredit.txt that explains those special features of your agent.
If you wish to enter the competition, explain that in the ExtraCredit.txt file, and submit the additional qualifying game transcript.

Updates and Corrections:

Updated on May 2 with information about how to enter a competition and receive extra credit for doing so.

If necessary, further updates and corrections will be posted here and/or mentioned in class, in GoPost, or via the mailing list.

Feedback Survey

After submitting your solution, please answer this survey