Penn Engineering ONLINE LEARNING

CIS 521-Artificial Intelligence

Homework 4: Adversarial Games [50 points]

Instructions

In this assignment, you will explore a game from the perspective of adversarial search.

A skeleton file `homework4.py` containing empty definitions for each question has been provided. As before, since portions of this assignment will be graded automatically, none of the names or function signatures in this file should be modified. However, you are free to introduce additional variables or functions if needed.

You may import definitions from any standard Python library, and are encouraged to do so in case you find yourself reinventing the wheel. If you are unsure where to start, consider taking a look at the data structures and functions defined in the `collections`, `itertools`, `Queue`, and `random` modules.

You will find that in addition to a problem specification, most programming questions also include a pair of examples from the Python interpreter. These are meant to illustrate typical use cases, and should not be taken as comprehensive test suites.

Once you have completed the assignment, you should submit your file on Gradescope. You may submit as many times as you would like before the deadline, but only the last submission will be saved.

Style [5 points]

Your code should follow the proper Python style guidelines set forth in PEP 8, which was written in part by the creator of Python. Our autograders will automatically scan your submission for style errors using the `pycodestyle` library on default settings. If your submission contains **any** style errors, the autograder will show you some of them and you will not receive these 5 points. You can use `pycodestyle` or any other tool you like to make sure that your submission conforms to PEP 8 guidelines.

Dominoes Game

In this section, you will develop an AI for a game in which two players take turns placing 1×2 dominoes on a rectangular grid. One player must always place his dominoes vertically, and the other must always place his dominoes horizontally. The last player who successfully places a domino on the board wins.

As with the Tile Puzzle, an infrastructure that is compatible with the provided GUI has been suggested. However, only the search method will be tested, so you are free to choose a different approach if you find it more convenient to do so.

The representation used for this puzzle is a two-dimensional list of Boolean values, where `True` corresponds to a filled square and `False` corresponds to an empty square.

1. In the `DominoesGame` class, write an initialization method `__init__(self, board)` that stores an input board of the form described above for future use. You additionally may wish to store the dimensions of the board as separate internal variables, though this is not required.

2. Suggested infrastructure.

   In the `DominoesGame` class, write a method `get_board(self)` that returns the internal representation of the board stored during initialization.

   ```
   >>> b = [[False, False], [False, False]]
   >>> g = DominoesGame(b)
   >>> g.get_board()
   [[False, False], [False, False]]

   >>> b = [[True, False], [True, False]]
   >>> g = DominoesGame(b)
   >>> g.get_board()
   [[True, False], [True, False]]
   ```

   Write a top-level function `create_dominoes_game(rows, cols)` that returns a new `DominoesGame` of the specified dimensions with all squares initialized to the empty state.

   ```
   >>> g = create_dominoes_game(2, 2)
   >>> g.get_board()
   [[False, False], [False, False]]

   >>> g = create_dominoes_game(2, 3)
   >>> g.get_board()
   [[False, False, False],
    [False, False, False]]
   ```

   In the `DominoesGame` class, write a method `reset(self)` which resets all of the internal board's squares to the empty state.

   ```
   >>> b = [[False, False], [False, False]]
   >>> g = DominoesGame(b)
   >>> g.get_board()
   [[False, False], [False, False]]
   >>> g.reset()
   >>> g.get_board()
   [[False, False], [False, False]]

   >>> b = [[True, False], [True, False]]
   >>> g = DominoesGame(b)
   >>> g.get_board()
   [[True, False], [True, False]]
   ```

```
>>> g.reset()
>>> g.get_board()
[[False, False], [False, False]]
```

In the `DominoesGame` class, write a method `is_legal_move(self, row, col, vertical)` that returns a Boolean value indicating whether the given move can be played on the current board. A legal move must place a domino fully within bounds, and may not cover squares which have already been filled.

If the `vertical` parameter is `True`, then the current player intends to place a domino on squares `(row, col)` and `(row + 1, col)`. If the `vertical` parameter is `False`, then the current player intends to place a domino on squares `(row, col)` and `(row, col + 1)`. This convention will be followed throughout the rest of the section.

```
>>> b = [[False, False], [False, False]]
>>> g = DominoesGame(b)
>>> g.is_legal_move(0, 0, True)
True
>>> g.is_legal_move(0, 0, False)
True

>>> b = [[True, False], [True, False]]
>>> g = DominoesGame(b)
>>> g.is_legal_move(0, 0, False)
False
>>> g.is_legal_move(0, 1, True)
True
>>> g.is_legal_move(1, 1, True)
False
```

In the `DominoesGame` class, write a method `legal_moves(self, vertical)` which yields the legal moves available to the current player as (row, column) tuples. The moves should be generated in row-major order (i.e. iterating through the rows from top to bottom, and within rows from left to right), starting from the top-left corner of the board.

```
>>> g = create_dominoes_game(3, 3)
>>> list(g.legal_moves(True))
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1),
 (1, 2)]
>>> list(g.legal_moves(False))
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0),
 (2, 1)]

>>> b = [[True, False], [True, False]]
>>> g = DominoesGame(b)
>>> list(g.legal_moves(True))
```

```
[(0, 1)]
>>> list(g.legal_moves(False))
[]
```

In the `DominoesGame` class, write a method `perform_move(self, row, col, vertical)` which fills the squares covered by a domino placed at the given location in the specified orientation.

```
>>> g = create_dominoes_game(3, 3)
>>> g.perform_move(0, 1, True)
>>> g.get_board()
[[False, True,  False],
 [False, True,  False],
 [False, False, False]]

>>> g = create_dominoes_game(3, 3)
>>> g.perform_move(1, 0, False)
>>> g.get_board()
[[False, False, False],
 [True,  True,  False],
 [False, False, False]]
```

In the `DominoesGame` class, write a method `game_over(self, vertical)` that returns whether the current player is unable to place any dominoes.

```
>>> b = [[False, False], [False, False]]
>>> g = DominoesGame(b)
>>> g.game_over(True)
False
>>> g.game_over(False)
False

>>> b = [[True, False], [True, False]]
>>> g = DominoesGame(b)
>>> g.game_over(True)
False
>>> g.game_over(False)
True
```

In the `DominoesGame` class, write a method `copy(self)` that returns a new `DominoesGame` object initialized with a deep copy of the current board. Changes made to the original puzzle should not be reflected in the copy, and vice versa.

```
>>> g = create_dominoes_game(4, 4)
>>> g2 = g.copy()
>>> g.get_board() == g2.get_board()
```

```
True

>>> g = create_dominoes_game(4, 4)
>>> g2 = g.copy()
>>> g.perform_move(0, 0, True)
>>> g.get_board() == g2.get_board()
False
```

In the `DominoesGame` class, write a method `successors(self, vertical)` that yields all successors of the puzzle for the current player as (move, new-game) tuples, where moves themselves are (row, column) tuples. The second element of each successor should be a new `DominoesGame` object whose board is the result of applying the corresponding move to the current board. The successors should be generated in the same order in which moves are produced by the `legal_moves(self, vertical)` method.

```
>>> b = [[False, False], [False, False]]
>>> g = DominoesGame(b)
>>> for m, new_g in g.successors(True):
...     print m, new_g.get_board()
...
(0, 0) [[True, False], [True, False]]
(0, 1) [[False, True], [False, True]]

>>> b = [[True, False], [True, False]]
>>> g = DominoesGame(b)
>>> for m, new_g in g.successors(True):
...     print m, new_g.get_board()
...
(0, 1) [[True, True], [True, True]]
```

Optional.

In the `DominoesGame` class, write a method `get_random_move(self, vertical)` which returns a random legal move for the current player as a (row, column) tuple. The `random` module contains a function `random.choice(seq)` which returns a random element from its input sequence.

3. In the `DominoesGame` class, write a method `get_best_move(self, vertical, limit)` which returns a 3-element tuple containing the best move for the current player as a (row, column) tuple, its associated value, and the number of leaf nodes visited during the search. Recall that if the `vertical` parameter is `True`, then the current player intends to place a domino on squares `(row, col)` and `(row + 1, col)`, and if the `vertical` parameter is `False`, then the current player intends to place a domino on squares `(row, col)` and `(row, col + 1)`. Moves should be explored row-major order, described in further detail above, to ensure consistency.

   Your search should be a faithful implementation of the alpha-beta search given on page 170 of the course textbook, with the restriction that you should look no further than `limit` moves into the future.

To evaluate a board, you should compute the number of moves available to the current player, then subtract the number of moves available to the opponent.

```
>>> b = [[False] * 3 for i in range(3)]
>>> g = DominoesGame(b)
>>> g.get_best_move(True, 1)
((0, 1), 2, 6)
>>> g.get_best_move(True, 2)
((0, 1), 3, 10)

>>> b = [[False] * 3 for i in range(3)]
>>> g = DominoesGame(b)
>>> g.perform_move(0, 1, True)
>>> g.get_best_move(False, 1)
((2, 0), -3, 2)
>>> g.get_best_move(False, 2)
((2, 0), -2, 5)
```

If you implemented the suggested infrastructure described in this section, you can play with an interactive version of the dominoes board game using the provided GUI by running the following command:

```
python homework3_dominoes_game_gui.py rows cols
```

The arguments `rows` and `cols` are positive integers designating the size of the board.

In the GUI, you can click on a square to make a move, press 'r' to perform a random move, or press a number between 1 and 9 to perform the best move found according to an alpha-beta search with that limit. The GUI is merely a wrapper around your implementations of the relevant functions, and may therefore serve as a useful visual tool for debugging.

Feedback

1. Approximately how long did you spend on this assignment?
2. Which aspects of this assignment did you find most challenging? Were there any significant stumbling blocks?
3. Which aspects of this assignment did you like? Is there anything you would have changed?