In this assignment, you will explore the alpha-beta search algorithm from the perspective of a domino-based board game.

A skeleton file homework4.py containing empty definitions for each question has been provided. 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, 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.

You are strongly encouraged to follow the Python style guidelines set forth in PEP 8, which was written in part by the creator of Python. However, your code will not be graded for style.

Once you have completed the assignment, you should submit your file on Eniac using the following turnin command, where the flags -c and -p stand for “course” and “project”, respectively.

```
turnin -c cis391 -p hw4 homework4.py
```

You may submit as many times as you would like before the deadline, but only the last submission will be saved. To view a detailed listing of the contents of your most recent submission, you can use the following command, where the flag -v stands for “verbose”.

```
turnin -c cis391 -p hw4 -v
```

## 1 Dominoes Game [95 Points]

In this section, you will develop an AI for a game in which two players take turns placing 1 \times 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.

1. [2 Points] A natural representation 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. In the DominoesGame class, write an initialization method __init__(self, board) that stores an input board of this form for future use. Also write a method get_board(self) that returns this internal representation. You additionally may wish to store the dimensions of the board as separate internal variables, though this is not required.
2. [3 Points] 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.

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

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

3. [3 Points] In the `DominoesGame` class, write a method `reset(self)` which resets all of the internal board’s squares to the empty state.

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

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

4. [5 Points] 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.

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

```python
>>> 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
```

5. [5 Points] 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.
6. **[5 Points]** 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.

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

7. **[5 Points]** In the `DominoesGame` class, write a method `game_over(self, vertical)` that returns whether the current player is unable to place any dominoes.

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

8. **[2 Points]** 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.

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

9. **[5 Points]** 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.

```python
>>> 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]]
```

10. **[5 Points]** 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. *Hint:* The `random` module contains a function `random.choice(seq)` which returns a random element from its input sequence.

11. **[20 Points]** 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. 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.

```python
>>> g = create_dominoes_game(3, 3)
>>> g.get_best_move(True, 1)
((0, 1), 2, 6)
>>> g.get_best_move(True, 2)
((0, 1), 3, 10)
```

```python
>>> g = create_dominoes_game(3, 3)
>>> 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)
```

12. **[20 Points]** In the `DominoesGame` class, write a method `get_best_move_custom(self, vertical)` which returns the best move according to your own criteria. At minimum, you should incorporate new elements into the evaluation function used in the previous exercise. For example, you might consider the fact that certain squares may be strategically more valuable than others, or that partway through the game, some regions of the board may be accessible by only one player. On top of this, you may also choose to implement a more sophisticated pruning technique, or incorporate some form of nondeterminism into your search. To encourage you to experiment, anything which differs in a significant way from the previous exercise but is still strong enough to beat a random player on a consistent basis will be awarded full credit.

13. **[15 Points]** Describe your custom criteria for move selection, and give a qualitative comparison of its performance relative to the baseline alpha-beta search. Your response should be at least two paragraphs in length.

Once you have implemented the functions and methods 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 homework4_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, press a number between 1 and 9 to perform the best move found according to an alpha-beta search with that limit, or press ‘c’ to perform the best move found according to your custom criteria. The GUI is merely a wrapper around your implementations of the relevant functions, and may therefore serve as a useful visual tool for debugging.

2 **Feedback**  [5 Points]

1. **[1 Point]** Approximately how long did you spend on this assignment?

2. **[2 Points]** Which aspects of this assignment did you find most challenging? Were there any significant stumbling blocks?

3. **[2 Points]** Which aspects of this assignment did you like? Is there anything you would have changed?