Practice Activity: Solitaire Tantrix

Tantrix is puzzle game involving a grid of colored hexagonal tiles. In particular, each tile displays three curves of different colors that connect pairs of edges on the tile. During a game of Tantrix, players place their tiles on the grid subject to the restriction that curves incident on a common edge shared by two adjacent tiles must be of the same color. These placements generate *legal* tile configurations.

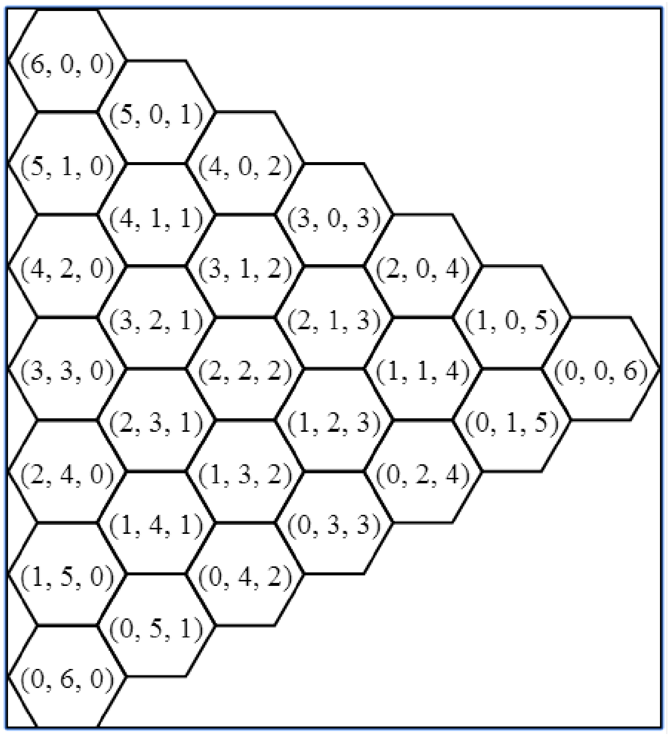
During the course of a two-player game of Tantrix, player alternate placing tiles with the objective of creating the longest curve (or loop) of a specified color. In the solitaire version of Tantrix, the player is given ten tiles with the objective of placing the tiles on the grid in a legal configuration that contains a loop of length ten. For this activity, you will implement a version of Solitaire Tantrix that works with a provided GUI. Your task will be to implement a `Tantrix` class that models and manipulates a collection of tiles lying on a grid of hexagons.

Before proceeding, we recommend that you run and experiment with our solution to this activity to gain an overall feel for how Solitaire Tantrix works. (Of course, please resist the urge to examine the code closely if you wish to implement things yourself.) The GUI displays a collection of hexagonal cells with the ten Tantrix tiles placed in the right portion of the grid. You may move a tile to another cell by clicking and dragging the tile. You may rotate that tile in its current cell by clicking on it. Note that the GUI allows placement of tiles in illegal configurations. The button labeled "Is legal?" tests whether the current tile configuration is legal. The buttons labeled "... loop of length 10?" checks whether the tiles have been placed in legal configuration that contains a single closed loop using all ten tiles.

Modeling a hexagonal grid

For grids of rectangular cells, we modeled the grid as a 2D list whose elements are indexed by the row and column of the cells. For grids of hexagonal cells, choosing a model for the grid is not as simple. For this activity, we will model the grid as a dictionary whose entries are indexed by tuples of the form (i, j, k) where $i + j + k = n$. We refer to this number n as the*size*of the grid.

The image below show a triangular portion of a grid of size 6 where $i, j, k \geq 0$. Each hexagonal tile is labeled by its corresponding index.



While this indexing scheme may seem very strange to you, we have chosen to use it for several reasons. From a mathematical viewpoint, the scheme has the advantage that each of the six directions in the grid are treated in the manner. From a pedagogical viewpoint, this model provides extra practice working with the class invariant $i + j + k = n$ that our scheme will maintain.

Phase one - implement basic `Tantrix` methods

In phase one, your task is to take the initial program template for Tantrix and implement the following methods: `__init__`, `__str__`, `get_tiling_size`, `tile_exists`, `place_tile`, `remove_tile`, and `rotate_tile`. Most of the methods except for `__init__` have very short implementations that manipulate the dictionary used to relate a tile's index to its encoding.

The `__init__` method should initialize the grid's dictionary and store the ten tiles in the list `SOLITAIRE_CODES` in the grid. Each tile in this list is modeled by a `code` that is string of six letters. These letters represent the colors of the curves incident on each of the tile's edges, with the edges taken in clockwise order. For example, the first tile has the code `"BBRRYY"` which signifies that the tile's edges have curves of the colors blue, blue, red, red, yellow, and yellow, respectively, incident on them.

The `__init__` method should places these tiles in a grid of the specified size. In our implementation, the ten tile are initially placed in the rightmost portion of a grid of size 6. For example, the tile with code `"BBRRYY"` is stored at the cell indexed by the tuple $(0, 0, 6)$. In Python, the grid's dictionary would include the key `(0, 0, 6)` with associated value `"BBRRYY"`. In this dictionary representation, the indices for cells that are empty do not appear in the dictionary.

Once you implemented these methods you should be able to interact with the GUI and manipulate the tiles using a combination of clicking and dragging. Note the clicking on a tile calls `rotate_tile`. If the required behavior of `rotate_tile` is unclear, you may wish to experiment with the GUI using our implementation of phase one.

Phase two - detect whether a configuration is legal

For phase two, your task is to implement two more methods in the `Tantrix` class: `get_neighbor` and `is_legal`. The method `get_neighbor` takes the index of a tile and a direction. The direction is one of the six tuples in the dictionary `DIRECTIONS`. Note that each tuple in this dictionary corresponds to the difference between the indices of two tiles that are share a common edge. Given an index and a direction, we simply add the three components of these tuples and return the resulting tuple as the index of the neighboring cell.

The method `is_legal` should iterate over the tiles in the grid and check whether a neighboring tile exists in any of the six directions. For each neighboring tile, the method should verify that the edge shared by these two tiles has the same color for both tiles. We recommend that you use the method `get_neighbor` in your implementation of `is_legal`. Once you have implemented `is_legal`, clicking the button labeled "Is legal?" should correctly return whether the current configuration of the tiles is legal. You are welcome to compare your implementation of phase two to our implementation.

Phase three - check for a win

For the final phase, your task is to implement the method `has_loop`. This method takes a color and checks whether the current configuration of the puzzle is legal and contains a single closed loop of the specified color that uses all ten tiles. Implementing `has_loop` will require some type of simple search that follows the sequences of edges in the grid connected by curves of the same color. If you need further hints, you are welcome to examine our implementation of phase three.

Mark as completed