Ethan Yi Sheng, Chan cs61bl-ag
Kevin, Chian cs61bl-ae

CS61BL Project 1 Readme

Board Representation

- We decided to represent 15 Connectors instead of 20 Triangles on our data structure
- We decided to represent board representation as a 2-Dimensional ArrayList containing Connectors.
    - An ArrayList containing 6 ArrayList<Connector>, indexed 0-5
    - Within each ArrayList<Connector>, it contains 7 cells, indexed 0-6.
    - We initialized position (x,y) on the 2D ArrayList to correspond exactly to where each Connector belongs. For example: A connector (1,3) would go in the first column, third row. In our data structure, it would reside in the first ArrayList<Connector> in the third index. The max possible connector was (5,6).
    - To prevent accidental access of other positions, we initialized the rest of the cells to null.
    - We created a 2D Color array to store the colors of the connectors corresponding to the same (x,y) position. For example, a connector with endpoints 1 and 3 would have its color stored in cell [1][3].
    - We initialized all possible connectors to be originally white and the rest to be null to represent a blank board.

Following are picture representations of our 2 data structures:

ArrayList<ArrayList<Connector>>

		ArrayList<ArrayList<Connector>>					
		0	1	2	3	4	5
ArrayList<Connector>	0	Null	Null	Null	Null	Null	Null
	1	Null	Null	Null	Null	Null	Null
	2	Null	Cnctr	Null	Null	Null	Null
	3	Null	Cnctr	Cnctr	Null	Null	Null
	4	Null	Cnctr	Cnctr	Cnctr	Null	Null
	5	Null	Cnctr	Cnctr	Cnctr	Cnctr	Null
	6	Null	Cnctr	Cnctr	Cnctr	Cnctr	Cnctr

Ethan Yi Sheng, Chan cs61bl-ag
Kevin, Chian cs61bl-ae

Color [ ][ ] connectorColors

		Color[x]					
		0	1	2	3	4	5
Color[x][y]	0	Null	Null	Null	Null	Null	Null
	1	Null	Null	Null	Null	Null	Null
	2	Null	Color	Null	Null	Null	Null
	3	Null	Color	Color	Null	Null	Null
	4	Null	Color	Color	Color	Null	Null
	5	Null	Color	Color	Color	Color	Null
	6	Null	Color	Color	Color	Color	Color

Justification of our 2D Structure

- We chose a two-dimensional structure that is first indexed by one endpoint of a connector and then indexed by the other rather than a combination of one or more linear collections because:
    - We wanted to view it in a very systematic way, like storing our Connectors on a grid map. This 2 Dimensional structure allows us to retrieve our desired connector very fast and easily just by calling their index. If we had used combinations of linear structures, we would have to remember to make 2 calls to retrieve our desired information. Only one call is needed here to retrieve our desired connector.
    - It fits very well because a Connector line has 2 end points, hence a 2D structure is the most efficient and friendly way to implement the data structure
    - It is easy to understand and avoid errors while coding. This way we never had issues with certain cells corresponding to certain connectors.
- The same concept applies for our Color 2 Dimensional matrix that allows quick retrieval of information just by the index of the two points of the connector.
    - We decided to take advantage of the built-in Color 2 dimensional matrix in java and hence saved a lot of implementation time.
    - This allows easy and fast access to Color values stored in the matrix with the index corresponding to the actual connector line index points.

Ethan Yi Sheng, Chan cs61bl-ag
Kevin, Chian cs61bl-ae

Our AI Choice( ) Algorithm

- Our algorithm has multiple layers of filtering the best move.

1. We first sorted all the possible moves into non-losing moves for the computer and losing moves into a goodMovesList and badMovesList.

    a. This ensures the computer will never pick a losing move if it has a non-losing move to choose.

    b. If it has run out of good moves, it will pick a move from the bad moves list and will definitely lose.

2. From the goodMovesList, we iterate through the good moves to ensure that the computer will not pick a losing move for the opponent. Meaning: If the computers move helps to complete a triangle for the opponent's color, our choice method will do our best to avoid it.

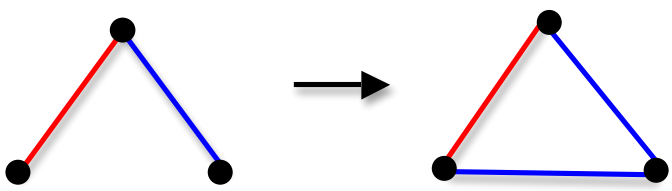
    a. This ensures that the opponent will always have one more losing move to choose from, thus increasing the odds of the computer winning.

3. Our most clever "mixed Triangle" tactic is that we iterate through the possible good moves to see if our move will form a mixed triangle. This means that the computer will choose a move to fill up a triangle that consists of 1 blue line and 1 red Line.

    a. These types of triangles are guaranteed non-losing moves for both players. It takes one good move from the opponent and keeps itself from losing.

    b. This will ensure the move the computer just made does not create a new losing move situation for itself when it's the computer's turn again.

    c. By reducing the losing moves pile for the computer, it will increase its chances of winning in the game at the end, since the player has to go one more turn.



4. To ensure that the player does not use that same tactic against the computer, we came up with an adjacency method that iterates through the list of good

moves to ensure that if the computer can help it, it will not choose a move that touches any end of a red line.

a. This is done so that the player will not apply this “mixed Triangle” tactic against the computer, especially in the earlier stages of the game.

Ethan Yi Sheng, Chan cs61bl-ag
Kevin, Chian cs61bl-ae

Development Sequence

1. **Data Structure to store Connectors -** We first met together in order to brainstorm the concepts and organization of this project. After a few ideas involving ArrayLists and arrays, we decided to settle on using a 2D ArrayList to store our connectors. Afterwards, we decided to create a few invariants this 2D ArrayList could never violate:

    a. It must only store possible value connectors. For example, a (4,1) connector could never be stored because (1,4) was where we designated that specific connector. This wasn't an issue, because the connector class automatically assigned p1 to be the smaller of the endpoints.

    b. It could only have indexes up to (5,6). We limited the storage capacity on these ArrayLists to ensure that anything added past (5,6) would not be possible.

2. **Data Structure to store Colors** - Then, we decided to create a Color 2-dimensional array to correspond to the points/cells on the ArrayList. We originally wanted to add the color to be just another part of the connector information stored in the 2D ArrayList, but the rules of the project prohibited us from modifying the Connector class to store colors. By using an array, we could more easily access the cells in cleaner code, rather than having to type ArrayList.get(endPt1).get(endPt2) we could just write array[endPt1][endPt2]. The array has to follow the same invariants as the ArrayList. In a beginning board, we could initialize all possible "connectors" to be white.

3. **colorOf Method -** Using these two data structures, coding the constructor, add, and colorOf was all easy to do. We used loops to initially fill the two data structures, then accessed the data types using built-in methods to change or return something. One issue we came across here was an OutOfBoundsException for the ArrayList because we didn't know ArrayLists didn't initialize the size, but instead the capacity. This was easily changed later by modifying the constructor to avoid the OutofBoundsException.

4. **testAdd –** We also created tests which added connectors of both red and blue colors into the data structure. We then tested the adding method by using the colorOf method to test it by testing for the right color (red,white or blue) at the right points on our data structure. After which, it helped us develop and fine tune our testAdd method very well.

5. **toConnector Method and Testing** - We then moved to the toConnector method in the Connector class. The toConnector method had a string input and two digit output. We brainstormed together to think of all possible incorrect inputs, then used the try and catch statements to catch IllegalFormatException

to check each one. If our test caught the IllegalFormatException, means our toConnector works. If our catch statement did not catch it, the code would have gone to the next assertFalse statement and our toConnector method would fail. Then we converted the string to individual characters without spaces and made them into integers using a built in ability of Java. We then again checked for other illegal inputs and otherwise returned a new Connector.

6. **toConnector and colorOf Method further testing -** At this stage, we completed our ConnectorTest class and some BoardTest methods to make them even more comprehensive. We entered in all possible toConnector illegal and legal inputs, checking multiple different inputs for each "edge case" that will cause an IllegalFormatException. The JUnit tests we created helped to further refine our method by covering even more cases. We also added the very basic add, colorOf tests checked just the basic purpose of each method. Afterwards, we added some code in isOK to check our methods. After some trials and testing, everything compiled and we advanced to the next step: iterators.

7. **formsTriangle Method –** We first created a test for forms triangle method by adding colored red and blue connectors and fed input to our method to ensure that the connector input would create a triangle for both red and blue. After which, we went on to write out the actual formsTriangle method which went smoothly. We tested the formsTriangle method for both red and blue colors to ensure its effectiveness.

8. **Implementing Iterator and its tests -** Due to the 2-dimensional nature of our ArrayList structure, Java does not have an iterator for a 2-dimensional ArrayList iterator, we created a new private class ArrayListIterator implementing the Iterator interface to help us iterate though our connectors. We created a new ArrayListIterator class with its own hasNext(), next() and increment() methods to help us iterate through our connectors. We then created a JUnit test to test its implementation, by creating a new Board of connectors and asserting that our iterator works for exactly 15 white connectors we had initialized. It worked smoothly and which confirmed our implementation of the iterator.

9. **Implementing the Color Iterator with its tests –** Due to the fact that our color data structure was independent from our actual iterator, we had the intention of using the color iterator as the way to access the colored iterators quickly and efficiently. We created another private class ColorArrayListIterator extending the ArrayListIterator, and added the Color variable to it allowing us to cycle through the various colored iterators. We created test cases for each, red, blue, and white to ensure that it worked correctly as it supposed to, iterating to the exact number of connectors of the desired color that had been initialized.

10. **isOk Method –** For our isOk method, we made it check that at every time it is called, the invariants of the board will always be checked. We knew our first condition was that the total number of reds, blue and white always equal 15. We always checked that the number of red and blues are always equal or the number of red moves is exactly one more than the blue moves. We also tested for null connectors or connectors that have invalid end points on the board, to make ensure the state of the board is always consistent with the game. We also ensured that there was no blue triangles formed on the board, to ensure the computer choice will actually lose.

11. **Debugger –** For most of our development sequence, we used the debugger when the methods we wrote failed the tests. We could create break points at the faulty lines of code and step though what every array cell and variable's value at every step to see what errors we made. This was extremely helpful and saved us considerable time. We managed to solve many null pointer and index out of bound exceptions with the use of the debugger. We also used a lot of System.out.print statements that aided us to keep track of where our code reaches on top of the debugger.

Team Contributions

- We both brainstormed the ideas together (for hours) and worked on whiteboards in soda hall to draw out our basic organization and coding plans, this is when we decided on the type of data structure to use and how we were going to implement it.

- Afterwards, we coded the testing methods together, and then developed the actual method together according to how we designed each method to be. Through pair programming, along the way one of us would point our syntax or potential logic errors in our code.

- The entire project was done together in person; the only parts of the project done separately were the test classes, since we thought we could come up with a more comprehensive test class that way.

- Overall, this project was done in a very cooperative effort. Each method has equal contribution by both partners' work in it.