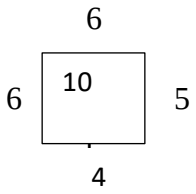
Assignment 4:
Name: Tejaswi Chaudhary
Banner ID: B00858613
Date: 09/11/2020

Overview:
Implement logic that solves edge-matching puzzle sometimes called tetravex puzzle. User provides n x m rectangular or square grid and set of n*m tiles. Each tile is identified by unique number and it has 4 edges (edge identifiers are also numbers).

Example of tile:



Here, in above example, 10 is name if piece and 6,6,4 and 5 are the edges of piece.

Piece can be placed in the puzzle grid at particular position if its edges match with neighbouring pieces edges, and each tile can be placed in the puzzle grid only once.

For example: piece with name 10 as given above can be store at grid[0][0] if its right edge(5) matches with the its neighbouring cell(grid[0][1]),and its bottom edge(4) matches with its neighbouring cell(grid[1][0]).

To solve puzzle, either user can put pieces in the puzzle grid by giving location of his/her choice and ask program to solve rest of the empty cells in the puzzle grid, or user can directly ask program to solve puzzle when puzzle grid is empty. It also keeps track of bad choices program has made that has led to dead end while solving puzzle. Basically, its state space emploration problem.

Files and external data:
This program contains following file:

- **mainUIPuzzle.java –** This file contains the main program that provides user interface. User can perform operations such as load puzzle, solve puzzle, print puzzle grid, place pieces in the puzzle and ask bad choices (just modified assignment-1 mainUI.java file according to requirements).
- **Tetravex.java –** class that load puzzle pieces, solve puzzle, set piece given by the user at given position if possible, print grid and also tells unused pieces and return number of bad choices had been made while solving puzzle.
- **Edges.java** – support class of Tetravex.java that stores edges of each store pieces.
- Other additional text file for testing the program.

Data Structures:

Following data structure is used in the program:

- Object of class Edges stores 4 edges of piece.
- LinkedHashMap is used to store pieces information. It stores piece name of type integer as key, and edges of type Edges as value to map. I have selected this data structure to store pieces as they are inserted (in order to maintain insertion order of input).
- ArrayLists are used. ArrayList_1 is used to store pieces that are already used or placed in the puzzle grid and ArrayList_2 is used to store keys (pieces name) from the LikedHashMap for future use in the program.
- 2-D Array is used to create puzzle grid of size n x m.

Algorithm used to solve problem:

Here, I have used brute force algorithm to solve puzzle. Basically, grid is filled recursively by enumerating the grid positions in row major order. Program always start solving puzzle by placing piece in puzzleGrid[0][0] instead of placing pieces to a random position. It will select pieces as in the same order as it is inserted instead of choosing randomly. It will try to fill every possible piece next to it recursively and follow the same steps until puzzle is solved or reach at dead end. If it will reach to a dead end then it will backtrack to previous state and place a different piece and continue the same steps.

Well this approach is very inefficient. For worst case, when the correct piece for puzzleGrid[0][0] was inserted last. But it will work little bit efficiently when it gets correct piece for puzzleGrid[0][0] as early as possible(for example, insertion order of correct piece is 1). Also, it place pieces in row major order instead of placing at random position also makes program little bit efficient.

Key algorithms and design elements:

public boolean loadPuzzle(BufferedReader stream) :

This method stores puzzle pieces information. It accepts the file that contains puzzle pieces data and height and width of the puzzle grid from the BufferedReader then it creates puzzle grid of given height and width and store pieces information in LinkedHashTable. It returns true if all the puzzle pieces are stored successfully, and return false if unexpected input data encounters.

- Firstly, it checks whether LikedHashMap has data stored into it in order to make sure that it is not storing same file data again, or one file data is already stored and user give another file to read (to read new file user needs to create new instance of the class).
- It will throw Exception if value in stream is null or empty
- In the following cases this method will return false and will clear stored valid pieces in the LikedHashMap before returning false:

    - If the piece is already stored as key in LikedHashTable (not accept two pieces with same number).
    - If the piece has more than 4 edges (line in file contains 6 space separated value).
    - If the 1st line of the file contains more than 2 space separated pieces.
    - If number of pieces are greater or less then the number of cells in the puzzle grid.
    - If the puzzle height and width are less than 2.
- If the above cases will not encounter, it will store each create puzzle grid of given height and width. Then after, it will store each piece name as key in LinkedHashMap ,and call Edges.java to store edges of the piece in object. Its will store object of Edges class as value to key(Piece) in LinkedHashMap.
- After storing set of pieces successfully, it will call getPuzzlePieces() to extract all the stores pieces as key in LinkedHashMap for future use in program and will return true.

public boolean solve()
This method solve puzzle by placing pieces in the puzzle grid.

- Firstly, it will check whether LikedHashMap has data stord into it. If not it will return false (this will execute when solve is called before loading puzzle pieces).
- This method is called when puzzle grid is empty or puzzle grid is partially filled.
- When solve is called for partially solved piece, it will call its support method supportSolve() to solve puzzle starting from column and rows as 0. If supportSolve() will return false to solve it will false to user, or if supportSolve() will return true to solve, it will return true to user.
- When solve is called for empty grid. it will take pieces one-by-one from ArrayList_2, and will place that piece at column 0 and row 0, and also add that piece in another ArrayList_1. After that, it will ask supportSolve() to solve puzzle starting from row 0 and column 1. If supportSolve() will return false, it will firstly remove that piece from the grid and will remove piece from the ArrayList_1 by passing the piece, column and row to the removeElement() method, and will take another piece from the ArrayList_2. It will increament bad choices variable, and then It will follow this steps until supportSolve() returns true, or all the pieces of puzzle are placed once in column 0 and row 0.
- When supportSolve() returns true, it will break the for loop by returning true to user. If all the pieces are placed once at column 0 and row 0 that mean puzzle can't be solved with given set of pieces.

private void getPuzzlePieces():
This method stores each key (Pieces name) of integer from the LikedHashTable to ArrayList_2 which is used by solve() and supportSolve().

private boolean supportSolve(int rows, int cols):
This method solves puzzle using recursion. It solves puzzle row wise from left to right. Firstly, it will solve 1st row then after goes to solve 2nd row so on. To solve puzzle method use following logic.
Called by solve() when grid is empty:

- Take single piece one-by-one from ArrayList_2.
- Check whether the current piece is already placed in the puzzle grid or not. If already placed in puzzle grid at any location, it will take another piece to store at current position. Otherwise, it will call placePieceSupport() to check the piece with given row and column can be stored or not. If placePieceSupport()return true, it will place that piece at given position in puzzle grid and will also add in ArrayList_1.
- After filling current position in the puzzle grid, it will move to next neighbour cell by calling itself recursively.
- After filling 1st row of the puzzle successfully, it will move to new row.
- When it reaches to dead end while solving puzzle, it will do backtrack to the state from where it is called.
- While backtracking, firstly it will remove that piece from the grid and from the ArrayList_1 by calling removeElement(), and will check whether another piece can store at the current position or not. If another piece in ArrayList_2 is possible to store, it will again follow above steps and call itself recursively to fill the next cell of the empty grid. In contrast, if another piece is not available to place at the current position, it will again perform backtracking for previous cell.
- It will return true to solve() when the size of ArrayList_1 is equals to the size of LikedListMap that mean all the pieces of puzzle are set in the puzzle grid and puzzle is solved.
- It will return false to solve() if it can not fill the grid even considering all the possibilities of pieces but cannot process further and reach to dead end.

Called by solve() when grid is partially filled(User has filled some pieces).

- For this case, it will also follow the all the above steps. But, when when it gets cell of the grid that is already placed, it will move to fill neighbour position, and leave current puzzle piece as it is.
- While it reaches to dead end, it will start backtracking, but will not remove pieces that are already placed by user. In that case, it will again backtrack to previous state and will remove if that previous place is not placed by user.

While performing backtracking, this method will count removed piece as one bad choice.

Private void removeElement(int piece, int cols, int row):
Basically this method is called by solve() and supportSolve() to remove piece from the puzzle grid place at given column and rows, and it will also remove the piece from ArrayList_1. To remove piece, it will simply put 0 to given column and row.

Public Boolean placePiece(int pieceName, int xPosition, int yPosition):
Firstly, it will check whether LikedHashMap has data stord into it. If not it will return false (this will execute when solve is called before loading puzzle pieces).

This method is called by user to place the pieceName at the given row(yPosition) and column(xPosition). It will call placePieceSupport(), placePieceSupport() will check whether the given piece edges match with its neighbours edges. If matches, placePieceSupport() will return true, this method with store pieceName at that postio and will return true to user. If

placePieceSupport() will return false, it will not add pieceName in the puzzle grid and will return false to user.

private placePieceSupport(int pieceName, int xPosition, int yPosition):
it will be called by solve(), supportSolve() and placePiece() to place the piece in the grid. Firstly, it will check whether the the pieceName is currently stored in the puzzle grid by looking in to ArrayList_2. If pieceName is already in puzzle grid, it will return false to the method which has called it. In contrast, it will find out its neighbour. It will perform following steps to find out neighbours.

- Left neighbour of the piece – will decrease column(xPosition) by 1.
- Right neighbour of the piece – will increasing column(xPosition) by 1
- Top neighbour of the piece – will decrease row(yPosition) by 1
- Bottom neighbour of the piece – will increase row(yPosition) by 1.

It will find valid (for example, value of Left and Top is greater than or equals to zero, and value of Right is less than column and Bottom is less than row) then it will find out whether that neighbour cells are empty or have pieces stored. For empty cells, it will not be going to match that neighbour edges with the piece. For cells that has pieces stored, it will match that pieces edges with piece. If edge-matching conditions satisfying it will return flag value 0, else it will return 1.

public String print():
Firstly, it will check whether LikedHashMap has data stord into it. If not it will return false (this will execute when solve is called before loading puzzle pieces).

This method return string that contains value of each cell in the puzzle grid and pieces with it's edges that are not stored (Unused pieces). It will separate used and unused pieces by 20 (-) signs.
It will print values of puzzle grid as all rows are separated by \n and and pieces stored at row are separated by \t. and unused pieces will be print as list of pieces each are separated by \n. it will print xxx for empty cell of the puzzle grid.
Example: placed Pieces: String= grid[0][0] +\t+grid[0][1]+\t+grid[0][2]+\n+grid[1][0]+ xxx..
Unused pieces: 10 1 2 3 4+\n+15 1 2 3 6
It will concatenate the values in the cell of puzzle grid and unused pieces shown in above example, and return that string to main.

public int Choices():
This method will return number of bad choices that program has made while solving and performed undo at later stage. Bad choices will be counted by solve() and supportSolve().

Assumption:

- The puzzle piece names and edges are positive integers.
- The height and width of the puzzle grid is positive integers.
- Height or width of the puzzle grid is grater or equals to 2.

Limitation:

- Once user place piece into puzzle grid, he/she cannot undo that choice.

- Puzzle piece cannot be rotated to solve the puzzle.

Test cases:

Input Validation:
public boolean loadPuzzle(BufferedReader stream):

- Null value passed for stream – throw Exception.
- Empty string passed for stream – throw Exception.
- Valid value passed to stream – return true.

public boolean placePiece(int pieceName, int xPosition, int yPosition):

- For xPosition and yPosition:
  - Negative integer passed – return false.
  - Give valid integer value – return true.
- Provide integer value for pieceName that is not loaded as puzzle piece- return false.

Boundary Cases:
public boolean loadPuzzle(BufferedReader stream):

- File does not exist.
- Store set of nm puzzle pieces.
- Create grid for 2 x 2.

public boolean solve() :

- Solve puzzle when the puzzle grid is empty.
- Solve puzzle when one of the cells in the grid is filled by user.
- Solve puzzle when the puzzle grid is filled (no empty cells).

public boolean placePiece(int pieceName, int xPosition, int yPosition):

- place pieces at any four edges or corner of the puzzle grid.
- place pieces when puzzle grid is empty.

public String print():

- Call when grid is filled (no empty cell) and there is not unused pieces left.
- Call when one of cells in the grid is filled.
- Call when grid is empty.

public int Choices():

- Call when we have 1 bad choices.

Control Flow:

public boolean loadPuzzle(BufferedReader stream):

- Load the set of puzzle pieces again(eg. read same file again).
- Pieces information is separated by blank line.
- Provide square or rectangular grid.
- Provide set of puzzle pieces greater than or less than nm.
- Piece information is separated by multiple space or tab.
- Provide piece which has more than or less than 4 edges (eg: piece information contains 6 values separated by space)
- Provide more than or less than 2 parameters to grid (eg: line contains 3 values separated by space).
- Provide grid where heigh or width is less than 2.
- Provide grid of any size where height and width are greater than 2.

public Boolean solve():

- Solve puzzle when user has placed pieces at wrong position in the puzzle grid.
- Solve puzzle when user has placed pieces at currect position in the puzzle grid.
- Solve puzzle with puzzle Pieces which can be never solved(one of the cell remain empty in the grid).

public boolean placePiece(int pieceName, int xPosition, int yPosition):

- Place pieces when puzzle grid is filled (with no empty cells).
- Place pieces when puzzle grid is partially filled.
- Provide integer value for pieceName that is currently stored in puzzle grid at any position.
- Ask to place pieceName at position where its edges are not matching with its neighbouring pieces's edges.
- Ask to place pieceName at position in grid which is already filled.
- Ask to store by giving valid value piceName and valid position in the grid where its edges are matching with its neighbouring pieces's edges.
- For xPosition and yPosition:
    - Value passed is exceed either height or width of the grid.
    - Value passed is greater or equals to 0 but not exceed height and width of the grid.

public String print():

- Call print when puzzle grid is partially filled.

public int Choices():

- Call method when program has made 0 bad choices.
- Call method when program has made more than 1 bad choices.

Data Flow:

- Ask program to solve puzzle without loading puzzle pieces.
- Ask program to place piece at given position before loading puzzle pieces.
- Ask program to return bad choices before solving the puzzle or loading the puzzle pieces.
- Ask program to print before loading puzzle pieces.