CSCI 3901: Assignment 4:

Waleed R. Alhindi (B00919848)

Overview:

The program submitted is designed to initialize a Tetris grid, set up its starting configuration, and then, based on a grid's current state, find the optimal placement of a given Tetris piece. An "optimal placement" differs when considering lookaheads, which specify how many blocks to consider after this placement to consider while calculating that placement. With a lookahead of 0, the program will find the optimal placement of that piece without considering any future placements. On the other hand, when the lookahead is greater than 0, the program will calculate the initial piece's placement based on the values of the proceeding <lookahead> number of pieces. Since each piece has a relative frequency that defines how often they will spawn, the value of potential proceeding pieces during the lookahead is a weighted value. Moreover, since "looking ahead" constitutes a state space exploration problem, as the <lookahead> value becomes larger, the time spent by the program in calculating the different possible combination of piece placements grows rapidly. This is exacerbated by the fact that each piece likely has multiple rotational orientations, which each form a separate combination branch. Thus, an important factor in designing and implementing this program is the efficiency of its data structures and algorithms. In other words, due to the essential complexity of this problem, our approach should minimize the additional accidental complexity it introduces.

Assumptions:

The assignment instructions grant the assumption that duplicate tetromino pieces will not be added. It does not, however, grant the assumption that an added piece will not have some rotation that duplicates an existing piece. Furthermore, it does not grant the assumption that pieces will strictly be tetrominos (i.e., they will always have 4 blocks). The approach taken to such is cases is outlined below in the design decisions section.

Design Decisions:

The program handles all error states by throwing exceptions, where each individual exception is supplied with an error message meant to indicate to the user what specifically went wrong. For example, if the user passes a puzzle piece that is not "tight" (i.e., it has an empty row or column) during addPuzzlePiece, that method will throw an IllegalArgumentException with the message "Piece is not tight". This type of error handling is consistently used throught the program.

The following are specific design decisions made during the development of this program:

- The TetrisSolver constructor throws an exception when a width or height of less than 1 is supplied. This is because negative dimensions are impossible, and dimensions of 0 would create a grid with 0 cells. However, TetrisSolver will accept a grid with a height and width of 1 (i.e., 1x1 grid), because it is still playable, albeit very trivial since addPuzzlrRow will always throw an exception because nextRow would be an empty or full row, and only one puzzle piece can be feasibly added ("*"). However, it is still technically a playable Tetris grid, and thus will accept 1x1 dimensions.

- The program will accept different characters that denote filled cells in addPuzzleRow and addPuzzlePiece, but will always format the filled cells in the grid as * chars, and empty cells as spaces. For example, the string " x! +" will be accepted in addPuzzleRow, but will be represented as " ** *" in the grid. This was done to allow flexibility in inputs, but also uniformity in outputs.

- addPuzzleRow will only accept nextRow strings that have lengths that exactly match the grid's width. This eliminates ambiguity regarding where the nextRow's columns should be placed and what omitted columns should be filled with. For example, if a grid with a width of 4 is supplied with a nextRow of "**", then it is unclear where in the row those blocks should be placed (can be placed starting from first, second, or third column), and it is unclear whether the remaining two columns omitted from nextRow should be empty or filled.

- addPuzzleRow will throw an exception if the user attempts to add a row when the grid's top row is already in use. In other words, if the method is called when at least one block in the grid's topmost row is filled, it will throw an exception, even when the nextRow added can fit in the top row (i.e., top row is " * " and nextRow added is "* *"). This decision is the outcome the interpretation of the assignment instructions, where it states "add a row of cells… to the top of whatever the top of the current grid configuration looks like."

- The assignment instructions state that addPuzzleRow "should not add a full row", thus the addPuzzleRow method throws an exception in such cases. However, no explicit instructions were made regarding blank rows. Thus, the decision was made to throw an exception in that case as well since the outcome of doing so would essentially be the same as adding a full row.

- addPuzzlePiece will throw an exception when none of the rotations of the added piece would fit in the grid, but not when at least one of the rotations would fit. For example, if the grid is 2x3 dimensions, then adding the piece "***\n***\n***\n" would throw an exception since it would not fit in the grid no matter how it is rotated. On the other hand, if the piece "***\n" is added, it will be accepted since it will fit when rotated.

- addPuzzlePiece will throw an exception if at least one of the added piece's rotations is identical to an existing piece. For example, if the piece "**\n" has already been added, then adding "*\n*\n" will throw an exception since at least one of its rotations is a duplicate of the existing piece. This also covers the case where identical puzzle piece strings are added (i.e., "**\n" and "**\n").

- addPuzzlePeice will throw an exception if the added piece is not tight or has hanging blocks. An example of a "untight" piece is "** \n** \n", where one of its columns is completely blank. An example of a piece with hanging blocks would be "** \n **" which can be illustrated as:

  **
    **

- addPuzzlePiece will accept piece strings even if they don't have a "\n" as long as it's valid. For example, "***" will be accepted and treated as if it was "***\n". Additionally, it will accept pieces even if they're rows do not match in length as long as it's valid, in which case it will fill out the rows as necessary. For example, "*\n**\n" will be accepted and treated as if it were "* \n**\n". This adds a layer of flexibility in how users represent their pieces in string form.

- addPuzzlePiece will throw an exception if the relativeFrequency supplied is less than 1. In the case of a negative frequency, that just doesn't make logical sense. However, in the case of a relative frequency of 0, this means that the piece will never appear during lookahead, but can be placed using placePiece. As such, I tried to allow for relative frequencies of 0 to be accepted and work around them during lookahead calculations, but I could not reliably do so since such cases would interfere with the way I am doing my calculations. So, I opted to reject relativeFrequencies of 0.

- placePiece will throw an exception if the pieceId supplied does not correspond to an existing piece. Additionally, it will throw an exception if lookahead is less than 0 since that would presumably be "looking backwards" which is outside the scope of this assignment.

- placePiece will throw an exception with the message "GAME OVER" when the piece being placed cannot fit in the grid due to its current configurations (i.e., the grid is almost full to the top and that piece will not fit no matter where it goes).

- In the case that lookahead is greater than the number of pieces that can be placed regardless of individual placement and order, then placePiece will stop "looking ahead" when it reaches the set of maximum possible placements and calculate the best value using the combinations that reach that highest look ahead. For example, if the current grid configuration can fit a maximum of 4 pieces, even when optimally placed, and a lookahead of 7 is passed, then it will only go up to a lookahead of 4 and calculate the best value between the placement combinations that reached that maximum possible lookahead of 4. The method will not, however, throw an exception. This approach is meant to increase the robustness of the program.

- The total value of a piece's placement when looking ahead is interpreted as:

  totalValue = [initialPlacementValue + sum(weighted next placement values)] – grid penalty after last placement in combination

  Where a placement's value is simply the points earned from clearing rows.
  Thus, the calculations during placePiece are built upon the above interpretations.

Program Components:

- TetrisSolver.java – this class constitutes the program's core functionality, providing the methods used to create a grid, place initial rows, add pieces, and place pieces.

- Tetromino.java – this class encapsulates the data of individual puzzle pieces into objects of this class. In other words, each puzzle piece added using the addPuzzlePiece method instantiates an object of this class, which will store the information of the puzzle piece being added. This class also verifies that a piece passed by the TetrisSolver class is valid before instantiating an object to encapsulate its data.

- PiecePlacement.java – this class encapsulates the data of individual piece placements, which include where that piece is being placed as represented by its starting row and starting column in the grid, the value of that placement, and the state of the Tetris grid after it has been placed. Objects of this class are used primarily when "looking ahead" during placePiece. They can be thought of as storing the state of the game at a particular point in time and, in turn, keep a reference to particular combinations of piece placements when looking ahead to future potential placements. Objects of this class can be thought of as a way to store past information while delving deeper into the state space exploration operations.

Data Structures:

The program uses a few different data structures to facilitate its operations. Each of the three classes that form the individual components of the program have their own dedicated data structures. Below is an outline of the data structures utilized by each class:

1. Tetromino.java:
   - A 2d int array that represents the piece, where 0s indicate an empty block, while 1s indicate a filled block. For example, the piece "* \n**\n" would be represented as [[1,0], [1,1]] in this 2d array.

   - A list of 2d int arrays that stores each of the piece's different rotational operations. For example, the piece "**" would create a list of { [[1, 1]], [[1], [1]]}. Notice how only unique rotational orientations are stored to save on memory. In this case, rotating the piece 90 degrees and 270 degrees results in the same piece, so we just store on of the two.

2. TetrisSolver.java:
   - A 2d char array that represents the grid.

   - A <Integer, Tertromino> map that stores the different tetrominos added using addPuzzlePiece. The integer key is the piece's id, and the value is the corresponding Tetromino object that stores information about that piece.

Note: during execution, many temporary data structures are used to facilitate operations during different methods. These include primarily maps and lists. However, these temporary data structures are local to the methods they appear in and are not lasting.

3. PiecePlacement.java:
   - A 2d int array that stores the orientation of the piece being placed. For example, if the piece "**\n" was being placed at a 90 degree rotation, then this 2d int array would store [[1], [1]].

   - A 2d char array that stores the configuration of the grid after that piece is placed. This can be seen as a copy of the grid that is used to propagate the lookahead functionality of TetrisSolver.

Key Algorithms:

Adding a Puzzle Row

Check that the row being added is valid (i.e., it is not null, not a full or empty row, and has a width equal to the grid's width)

    If it is not valid, throw an exception

Find the next usable row in the grid configuration

Check if the top usable row exceeds the grid's height (i.e., the top playable row is already in use)

    If it exceeds the grid's height, throw an exception

Iterate through the columns of the next usable row

    If the character in the nextRow string that corresponds to this column is not a space

        Put "*" in this column

Adding a Puzzle Piece

Check if the piece string is a valid input (i.e., not null or empty string)

    If it is not valid, throw an exception

Check if the relativeFrequency is less than 1

    If it is, throw an exception

Split the piece string into an array using "\n" as the delimiter

Try

    Feed that array into the Tetromino constructor

        Tetromino constructor checks if piece has any blank rows or columns

            If it does, it throws back an exception

        Tetromino constructor check if there are any hanging blocks

            If it does, it throws back an exception

Tetromino creates object of that piece

Catch

If the Tetromino constructor throws an exception

Throw it back out to the user with an appropriate error message depending on the Tetromino constructor's thrown exception

Iterate through all the existing puzzle pieces

Iterate through all that piece's rotations

If that piece's rotation equals the new piece (i.e., their 2d int arrays are equal)

Throw an exception (since this piece is a duplicate)

Add the new piece's Tetromino object to the list of puzzle pieces

Placing a Piece

Check if the ID supplied correspond to an existing piece

If it doesn't, throw an exception

Check that the lookahead is not negative

If it is, throw an exception

Create a list to store the initial piece's placements

Get the puzzle piece that corresponds to this ID

Iterate through each of the piece's rotations

Add all the possible placements of that rotation (i.e., the resultant PiecePlacement objects) into the initial placements list

Check if the initial placements list is empty

If it is empty, throw an exception (since the initial piece couldn't be placed given the current grid configuration)

Check if the lookahead is 0

If it is 0

Find the best value in the list of initial placements

Set the grid to correspond to the placement that yields the best value

Return the best value

If it is greater than 0

Create a lookahead combinations list of lists

Iterate through the initial placements list

Create a combination list

Add that initial placement to the combination list

Add the combination list to the lookahead combinations list of lists

Set a counter to 1

While the counter is less than lookahead

Iterate through the lookahead combinations list of lists

Using this combination list

Iterate through all puzzle pieces

Iterate through each of this piece's rotations

Find all possible placements using the grid of the last element in the combination list (i.e., the last grid prior to adding this new piece)

Create a list that expands upon this combination list for each of the possible next placements

Add that expanded combination list to he lookahead combinations list of lists

Increment the counter

Remove any lists from the lookahead combinations list of lists that has a length less than the incremented counter

Check if the resultant list of lists is empty

If it is (no more lookaheads can be done)

Use the subset of lists that have a length of the counter - 1

break the while loop

Using the filter lookahead combinations list of lists

Find the best total value list (i.e., the ordered set of piece placements that will generate the best total value based on the weighted averages described in the assignment instructions)

Get the first element in that list (which corresponds to the initial piece being placed)

Set the grid to reflect that best initial piece placement

Return the value of the best initial piece's placement

Efficiency:

With regards to data structures, the efficiency of this program is two-fold:

- Redundant data that does not add any value is omitted. For example, only unique piece rotations are stored (i.e., if the 90 degree and 270 degree rotations of a piece are identical, the program only stores on of them)

- Data structures that grow exponentially during certain method executions (particularly placePiece) are temporary and local only to that method. This means that these rapidly growing data structures are not kept in memory for longer than needed. Rather, the only "permanent" data structures throughout the entire program is one map to store the grid's configurations, one 2d array and list of 2d array (that has up to 4 arrays) per Tetromino object, and two 2d arrays per PiecePlacement object.

On the other hand, the algorithms were harder to make efficient due to the nature of the problem at hand. Initially, I wanted to approach the lookahead functionality in a recursive manner but had difficulties implementing that approach. Rather, I settled on an iterative approach where I had to keep track of previous data (i.e., the data between lookaheads). This meant that I had to create data structures, which usually consisted of lists or lists of lists, that grew exponentially the "deeper down" we go into lookaheads. If I was able to implement the recursive approach, the recursive call stack would have stored most of the information, which would have saved on memory – so, that is one optimization I would implement if I had the chance. However, to reduce the inefficiency caused by the iterative approach, data structures used when lookahead were filtered to keep only the information needed for the next lookahead between iterations. But, in the grand scheme of things, due to the exponential nature of looking ahead, I believe this type of optimization is negligible.

Furthermore, all iterations used throughout the program contain some condition to break their loops prematurely as to reduce the total iterations to the minimum number sufficient to accomplish a task. For example, when looking ahead, between each iteration, the method checks whether the last look ahead yielded any new combinations. If it didn't, then the method breaks the loop prematurely. In such a case, because of the grid's configuration, no new pieces can be added regardless of the piece and placement combinations preceding it. Instead of continuing the lookahead process, which increases execution time and data stored exponentially, that will yield empty results, the method ends the loop and uses the last set of lookahead placement combinations to compute the best placement for the initial piece. By doing so, meaningless iterations are reduced which significantly optimizes the method. However, this does not affect the inherent inefficiency of the iterative approach, and does has no effect when all the lookaheads yield some combination of placements.

Limitations:

- One limitation encountered while testing is that a large enough lookahead causes the compiler to throw an OutOfMemory exception with the message "Java heap space". For example, if the grid has a width of 3 and a height of 1 and is empty and only a "*\n" piece is added. Then lookahead can be infinite since a next piece can always be placed. So, for example, given the mentioned setup, if placePiece is called with a lookahead of 100, the compiler will throw an OutOfMemory exception.

- The program has no way of manually deleting rows. Although a user can manually place a piece that will clear some rows by calling placePiece with a lookahead of 0, they have no way of explicitly removing a specific row in the grid's current configuration.

- The nature of the problem inherently makes execution times significantly longer as lookahead becomes larger. Although, efficient implementation can reduce this growth in execution time, it would be negligible given the rate at which execution time grows. In other words, the problem inherently has an exponential time complexity.

Test Cases:

Input Validation:

TetrisSolver(int width, int height):

- Width = zero → throws IllegalArgumentException
- Height = zero → throws IllegalArgumentException
- Width < zero → throws IllegalArgumentException
- Height < zero → throws IllegalArgumentException

addPuzzleRow(String nextRow):

- nextRow = null → throws IllegalArgumentException
- nextRow is an empty string → throws IllegalArgumentException
- nextRow's length does not equal the grid's width → throws IllegalArgumentException
- nextRow is a string that constitutes a full row (i.e., a string of <width> length containing only characters [ex. "***"] → throws IllegalArgumentException
- nextRow is a string that constitutes an empty row (i.e., a string of <width> length containing only spaces [ex. " "] → throws IllegalArgumentException

int addPuzzlePiece(String piece, int relativeFrequency):

- piece = null → throws IllegalArgumentException
- one or more rows in piece are empty substrings [ex. "\n**\n"] → throws IllegalArgumentException
- one or more rows in piece contain only spaces [ex. " \n**\n"] → throws IllegalArgumentException
- one or more columns in piece are empty substrings [ex. "* \n*"] → throws IllegalArgumentException
- one or more columns in piece contain only spaces [ex. "* \n* "] → throws IllegalArgumentException
- piece is not tight (i.e., hanging blocks) [ex. "** \n **"] → throws IllegalArgumentException

- If all the piece's rotations have a height greater than the grid's height AND a width greater than the grid's width [ex. "**\n" when grid's height is 1 and grid's width is 1] → throws IllegalArgumentException
- relativeFrequency < 1 → throws IllegalArgumentException

int placePiece(int pieceId, int lookahead):

- lookahead < 0 → throws IllegalArgumentException

Boundary Cases:

TetrisSolver(int width, int height):

- width = 1 AND height = 1 → creates a grid of 1x1 dimensions

addPuzzleRow(string nextRow):

- nextRow is a string of <grid's width> length containing only one character [ex. " * "] → adds row to top of grid's configuration
- nextRow is a string of <grid's width> length containing only one space [ex. "** **"] → adds row to top of grid's configuration
- *Some covered in control flow*

int addPuzzlePiece(String piece, int relativeFrequency):

- only one of piece's rotations has a height <= grid's height and a width <= grid's width [ex. "*\n*\n*\n" when grid's height is 2 and grid's width is 3] → adds puzzle piece and returns its id
- relativeFrequency = 1 → adds puzzle piece and returns its id

int placePiece(int pieceId, int lookahead):

- lookahead = 0 → calculate best piece placement without calculating values of next potential pieces, places it in the grid, then returns the placement's value
- *Some covered in control flow*

Control Flow:

addPuzzleRow(string nextRow):

- Call when grid's top row has at least one filled cell → throws IllegalArgumentException (nextRow would go out of bounds of the grid since its top row is already in use)
- Call when grid is empty → fills in the grid's bottom row according to nextRow string
- Call when grid is partially filled (top filled cell is in a row < grid's height) → fills the next available row in the grid according to nextRow string

Int addPuzzlePiece(String piece, int relativeFrequency):

- *Covered in data flow*

int placePiece(int pieceId, int lookahead):

- lookahead > 0 → calculate best initial piece placement while "looking ahead" <lookahead> potential pieces, places initial piece in the grid, then returns that placement's value
- lookahead is greater than the amount of next potential pieces that any combination of placements could fit on the grid → looks ahead to the greatest number of potential pieces possible, calculates the initial piece's placement based on that, then return the initial placement's value
    - Ex. TetrisSolver(3, 5) → addRow("** ") → addPuzzlePiece("**\n**\n", 5) [pieceId = 1] → placePiece(1, 4)
        - Here the max number of lookaheads that can be performed are 1 after the initial piece regardless of the combination of placements due to the grid's initial configuration. So, the method will use the combinations with the highest lookahead possible (1 in this case) to calculate the initial piece's best placement
- *Some covered in data flow*

String showPuzzle():

- Call when grid is partially full → returns string of grid up to last row in use (i.e., excluding the upper rows that are empty)
- Call when grid's top row has at least one cell filled → returns string of entire grid (i.e., does not exclude any upper rows)
- *Some covered in data flow*

Data flow:

addPuzzleRow(String nextRow):

- Call when grid initialized with width = 1 and height = 1→ always throws IllegalArgumentException
    - Ex. TetrisSolver(1, 1) → addPuzzleRow("*") → throws IllegalArgumentException (full row)
    - Ex. TetrisSolver(1, 1) → addPuzzleRow(" ") → throws IllegalArgumentException (empty row)
- *Some covered in control flow*

int addPuzzlePiece(String piece, int relativeFrequency):

- one or more of the piece's rotations is a duplicate of an existing piece → throws IllegalArgumentException
    - Ex. addPuzzlePiece("*\n*\n", 1) → addPuzzlePiece("**\n", 3) → throws IllegalArgumentExcpetion
    - Ex. addPuzzlePiece("**\n**\n", 5) → addPuzzlePiece("**\n**\n, 15) → throws IllegalArgumentException

int placePuzzlePiece(int pieceId, int lookahead):

- pieceId does not reference a piece added using addPuzzlePiece (i.e., pieceId does not exist) → throws IllegalArgumentException
    - Before adding any pieces:

      Ex. TetrisSolver() → placePuzzlePiece() → always throws IllegalArgumentException
    - When the pieceId does not correspond with any of the pieces added:

      Ex. TetrisSolver() → addPuzzlePiece [pieceId = 1] → placePuzzlePiece(2, 4) → throws IllegalArgumentException
- pieceId *does* reference a piece added using addPuzzlePiece → calculates best piece placement, places it in the grid, then returns the placement's value

String showPuzzle():

- Call when grid is empty → returns empty string
    - Before adding any rows or placing any pieces:

Ex. TetrisSolver() → showPuzzle() → returns empty string

- After clearing all the remaining rows:

  Ex. TetrisSolver() → addPuzzleRow() → addPuzzlePiece() → placePuzzlePiece() [clears all remaining rows] → showPuzzle() → returns empty string