CSCI 3901 Assignment 4

Due date: 11:59pm Halifax time, Thursday, November 24, 2022 in git.cs.dal.ca at
https://git.cs.dal.ca/courses/2022-fall/csci-3901/assignment4/xxxx.git
where xxxx is your CSID. This repository is being created for you.

Goal

Work with exploring state space.

Background

Games are a domain in which the player searches through a set of possible local solutions to find an answer that satisfies all the puzzle's constraints. This assignment has you create a solver for a game.

Problem 1

Background

Tetris[1] is a video game that was developed in 1984. The game consists of a grid, usually 10 spaces wide and 20 rows high, in which you must place shaped pieces. The shaped pieces are tetrominoes…shapes that consist of four small blocks connected together (see Figure 2).

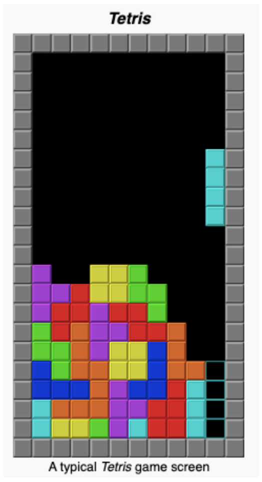

Figure 1 Sample Tetris grid from https://en.wikipedia.org/wiki/Tetris

A tetromino appears at the top of the grid and starts to move down the grid. The player can move the tetromino left or right or can rotate it. Once the tetromino is in the desired rotation and column, you let it fall into the grid. The tetromino settles in the grid where it rests on the highest supporting block already in the grid.

When a tetromino falls into the grid, we get points when we fill a full row of the grid with tetronimo pieces. Those full rows then disappear and the grid has fewer rows with leftover pieces of tetrominoes. The more full rows that disappear at once, the more points you get. Figure 1 shows a game configuation where dropping the vertical bar on the right will now see four rows become filled and so will disappear.

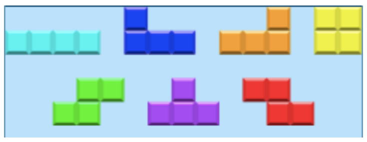

Figure 2 Tetris pieces, from https://tetris.fandom.com/wiki/Tetromino

As the game progresses, the tetrominoes fall more quickly, giving the player less time to decide on where the tetrominoes should go. The game ends when the grid overflows at the top.

[1] https://en.wikipedia.org/wiki/Tetris, retrieved November 8, 2022

Tetris players often focus on placing tetrominoes using a strategy that will give them an advantageous grid configuration. Common rules of thumb are

- Keep the number of rows small
- Avoid having spaces that have no tetromino piece surrounded by tetromino pieces, essentially making a hole among tetromino pieces
- Put pieces in the middle of the grid rather than the sides
- Plan for spaces where one particular tetromino could fit (if it appears at the top) to fill many rows at once.

Problem

Write a class called "TetrisSolver" that accepts a set of tetromino shapes and a puzzle grid. Given a puzzle grid, we can then provide the solver with one chosen tetromino shape and the solver will set the location of the shape into the puzzle, clearing filled rows, in a way that best reflects a solution strategy.

We capture the solution strategy in our evaluation of how undesirable a grid configuration is (a penalty) and in how valuable it is to clear a line from the grid (earned points). Given a grid configuration, consider one cell c in the grid. If cell c is part full (part of a tetromino piece) then its penalty has 2 components:

1. 1 * horizontal distance of c from the centre of the grid
2. 10 * number of rows below c in the grid

If cell c is empty, then its penalty has one component:

1. 7 * number of rows above c until we find the top of the filled cells

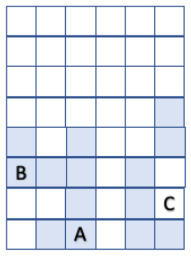

Figure 3 Sample Tetris grid

The penalty of a grid layout is then the sum of all the penalties of each individual cell in the grid. Consider the grid in Figure 3 where the shaded cells are parts of tetromino pieces. The penalty for cell A is 0 since it is on the middle of the grid and has no rows below it. The penalty for cell B is 2 for its horizontal position and 20 for having 2 rows below it, so a total penalty of 22. The penalty for cell C is 21 since C is empty and has 3 rows above it before we reach the top of all filled cells.

When we get to remove a full row, we get the following points: 50 if exactly one row is removed, 100 if exactly to rows are removed, 200 if exactly 3 rows are removed, and 400 if exactly 4 rows are removed.

When we are placing a piece, the value of the placement is then the points received less the penalty of the grid that remains. We want to place a piece in a position that produces the highest value.

The TetrisSolver class has at least 5 methods:

- Constructor that defines the grid width and height
- addPuzzleRow to define a row to add on top of the existing grid configuration. This method will generally be used to set up test cases.

- addPuzzlePiece to define a tetromino shape that canbe used in the game along with a relative frequency of appearance of the shape during the game.
- placePiece to put one tetromino into the grid (dropping from the top) in a position that produces the most value.
- showPuzzle to return a string representation of the puzzle and it’s state of being solved.

You get to choose how you will represent the puzzle in your program and how you will proceed to place the tetromino. Solution strategies are likely to need some trial-and-error with the puzzle as you try out a piece to see if it fits or not with other pieces.

You can also have any of these methods return an exception of your choice (that you document) in error conditions.

Method definitions

void TetrisSolver(int width, int height)

Create the TetrisSolver object for a grid that is “width” cells wide and has the capacity for “height” rows in the grid. Any tetromino piece that would fit outside those ranges represents a game that is ending.

void addPuzzleRow(String nextRow)

Given a Tetris puzzle that you have created, add a row of grid cells (filled or empty) to the top of whatever the top of the current game configuration looks like. You shouldn’t end up with a blank row between this new row and the rest of the puzzle. You also shouldn’t add a full row since the game would remove that full row.

Adding a row should keep the puzzle as a rectangle, so the width of the new row should match the current width of the puzzle (unless it’s the first row, which is defining the width). The added row will be a sequence of characters in the string, where each character represents one space of the grid row. That character is a space character if the grid cell is empty and is a non-space character if the grid cell is occupied by some tetromino piece.

int addPuzzlePiece (String piece, int relativeFrequency)

Add a tetromino as a puzzle piece to the puzzle. The “piece” parameter describes the shape of the tetromino while the “relativeFrequency” parameter identifies how frequently the tetromino would appear when the game generates random pieces to send to the player. If this method is called with the same piece shape a second time then the second invocation should fail.

The ”piece” parameter, is a character encoding of a rectangle in which the puzzle piece is placed. The string has a number of rows for the rectangle, each separated by a carriage return (\n), including at the end of the string. Each cell of a row is represented by a character; that character is a space if the cell isn’t part of the puzzle piece and is a non-space if the cell is part of the puzzle pieces.

For example, the bottom right red tetromino in Figure 2 would be the string “** \n **\n”.

Essentially, if you printed the string to the screen in a mono-width font then you would see the picture of the piece on the screen. This example would produce

```
**
 **
```

The rectangle containing the puzzle piece must be tight on all sides, meaning that there is no blank top or bottom row and no blank left or right column in the rectangle.

The “relativeFrequency” parameter is an indication of how frequently this piece would be sent to a player. For example, if a piece has a relativeFrequency of 15 and the sum of all relative frequencies of all pieces is 100 then the piece would be sent 15% of the time. In a perfectly fair game, you can supply a relative frequency of 1 to each piece to say that each piece is equally likely. By changing the frequencies, you can make the game harder by making valueable pieces (like 4 cells in a row) happen less often.

The method returns an integer that represents the pieces. We then use this integer in the placePiece method to identify which piece to add.

int placePiece (int pieceId, int lookahead)

The placePiece() method drops a tetromino with shape identified by “pieceId”, allowing rotations, into the game in a place that produces the most value. The method returns the value returned (i.e. points received for adding the piece less the penalty of the grid that remains). The returned value can be negative if the piece added does not fill any rows.

You may put a piece in one place in tetris in anticipation of what the upcoming pieces may be. This strategy is where the “lookahead” parameter comes in. The lookahead parameter tells you how many future pieces to consider when you place the piece.

If lookahead is 0 then you are placing the tetromino in the puzzle that produces the most value with that single tetromino piece. For example, if we have a 2x2 tetromino to add in Figure 4 with lookahead of 0 then we would put the pieces in the red position of Figure 4a since it is in the centre and stays lower.

If lookahead is 1 then we want to consider what the next tetromino might be in our calculations. In this case, we are looking at the expected value of the game after two pieces are added: the one piece being added now and then what the possibilities of the second piece might be. For example, consider the situation in Figure 4 where we have a 2x2 square to add. If we know that a tetromino with 4 cells in a row is likely to come next then we would choose the configuration in Figure 4b with the 2x2 square added in the green space and then the anticipated bar in the orange column, which would collapse two rows and would give us more points.

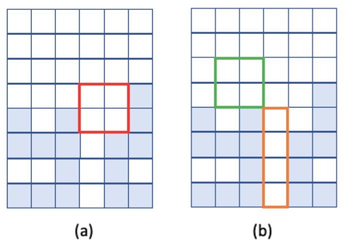

Figure 4 Two positions to consider for adding a square.

Similarly, if lookahead is 2 then we consider what the next two tetrominos might be in our calculation of value and positioning. As lookahead increases, expect your calculation time to grow exponentially.

How do you do this lookahead? Consider the situation where we have two tetromino pieces for the puzzle: A and B with relative frequencies of 5 and 15 respectively. We call placePiece to put in tetromino B and lookahead = 1. We have several possible places to put tetromino B. For each place of putting piece B, we then consider where each of the next pieces might go. If tetromino A were the next one to be placed suppose that it would have a value of X. If tegromino B were the next one to be placed then suppose that it would have a value of Y. The total value would then be (5X + 15Y) / (5 + 15) as an expected value. The best placement for the original tetromino B is the best of these expected values.

String showPuzzle()

The showPuzzle() method returns a string that, if printed, will show the puzzle state. The puzzle state is the rectangular puzzle with one character per cell in a row and with each row separated by a carriage return (\n). The character is either

- # if the cell is filled with some tetromino piece
- A space character if the cell is not covered by a tetromino piece

The string ends with a carriage return. Omit blank rows at the top of the puzzle.

For example, the grid of Figure 3 would return the following string

```
“     #\n# #  #\n### # \n  # # \n ## ##\n”
```

When printed to the screen in a mono-width font, you then get

```
     #
# #  #
### #
  # #
 ## ##
```

Sample problem

Consider a grid of width 5 and height 6:

```
addPuzzleRow( “ *** ” )
addPuzzleRow( “  *  “ )
```

Configuration is now as in Figure 5a.

```
addPuzzlePiece( “**\n**\n”, 1)  → returns id 1
addPuzzlePiece( “****\n”, 1)  → returns id 2
addPuzzlePiece( “** \n **\n”, 1)  → returns id 3
addPuzzlePiece( “ **\n** \n”, 1)  → returns id 4
```

```
placePiece( 3, 0 )
```

Configuration is now as in Figure 5b with the added piece in the orange space of the figure.

```
placePiece( 4, 0 )
```

Configuration goes to the case in Figure 5c when the piece is added in the green space and then we remove full rows to end in the configuration of Figure 5d.

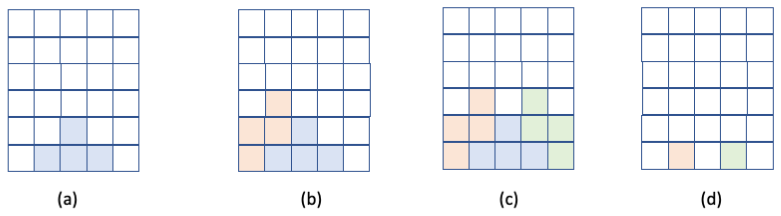

Figure 5 Puzzle progression (a) start, (b) add tetromino 3, (c) intermediate state as tetromino 4 is added, (d) final state after tetromino 4 is added

Assumptions

You may assume that

- We won’t try to add the same tetromino pattern twice to your TetrisSolver. The interface is defined where it could return an error condition in that case, but having you identify duplicate patterns is tangential to the purpose of the assignment.

Constraints

- You may use any data structures from the Java Collection Framework.
- You may not use an existing library that already solves this puzzle

- If in doubt for testing, I will be running your program on tiberlea.cs.dal.ca. Correct operation of your program shouldn't rely on any packages that aren't available on that system.

Notes

- Develop a strategy on how you will solve the puzzle before you finalize and start coding your data structure(s) for the puzzle.
- Work incrementally. The showPuzzle() method is likely to be your friend when debugging. I recommend breaking down the solution part into a few steps:
    - Develop code where you can manually put a tetromino into one position and return the grid value.
    - Develop code to decide on where to put a tetromino into the grid, assuming that you don't rotate the tetromino at all and with lookahead only set to 0.
    - Update the previous solution to now handle rotations of a tetromino.
    - Implement code to handle a lookahead value of 1.
    - Implement code to handle larger lookahead values.
- Recall that you should first seek _a_ solution to solving the puzzle. That alone can be tricky in some instances. Even consider a brute-force version that tries all numbers in each cell.

Marking scheme

The nature of this problem is one that you either have a solution or you don't. It's harder to get part-marks for a nearly-solved puzzle since you don't know if the partial solution will lead to an answer or not. So, focus on getting the main puzzle working well before optimizing.

- Documentation (internal and external), program organization, clarity, modularity, style – 4 marks
- List of test cases for the problem – 3 marks
- Clear explanation of how you are doing your solution, why your strategy works, and what steps you have taken to provide some degree of efficiency (include in your external documentation) – 3 marks
- Providing a solution that allows you to manually place some tetromino into the puzzle – 4 marks
- Providing a solution that works with a lookahead value of zero – 4 marks
- Providing a solution that handles a lookahead value of more than zero – 5 marks
- The effectiveness of your strategy to be efficient and to keep the running time of your program small – 2 marks