CS 342 - Software Design

Programming Project 3, Spring 2014

Sliding Block Puzzles

Due: Tuesday, March 18, 2014 at 11:59 pm

A sliding block puzzle consists of a number of pieces that fit into a confined area. The goal is to move one of the pieces to a specific position. This piece will be called the "goal piece". The goal can only be achieved by moving all of the pieces is a certain specified order of moves. Each piece may be restricted in the direction that it can move.

Such games or puzzles can be found under the names of Traffic Jam, Rush Hour, Parking Lot, Blocked, Unblock Me, etc. Check out:

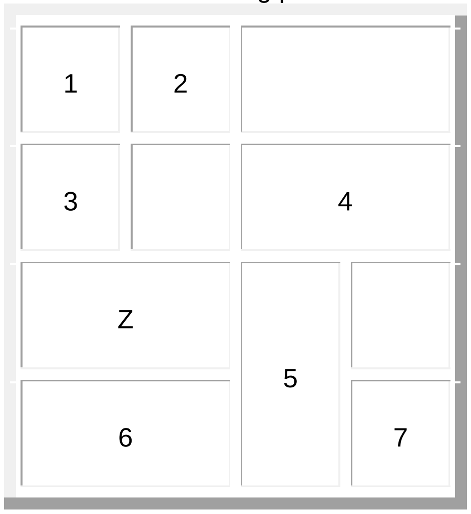
- http://www.johnrausch.com/slidingblockpuzzles/rushhour.htm
- http://www.mathsisfun.com/games/parking-lot-game.html

Your program is to provide a graphic user interface to play such a game. It is also to include a hint button, a solve button and a reset button which will allow the user to see the solution (or get a hint at the solution) and to try the puzzle again from the start. As the player moves the pieces around on the game, your program is to keep track of the "move count" or how many times a piece has been moved. Your program is to keep track of the minimum number of moves needed to solve a puzzle (i.e. fewest moves so far to solve the puzzle). Menu items for Exit, About and Help must exist.

Your program is to have at least 10 puzzles provided by you plus at least two puzzles provided by the class instructors. Your program should automatically load the next puzzle once the current one has been solved.

Example of a Sliding Block Puzzle

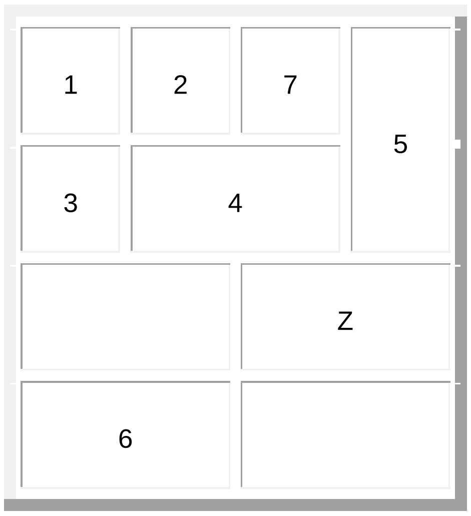
Consider the following puzzle:



The puzzle contains 8 pieces. The piece labeled "Z" is the "goal piece". For this puzzle to be solved, The "Z" piece must be moved to the right hand edge of the puzzle. In this puzzle all pieces can move any direction we wish (left/right or up/down). One solution is to:

- move piece 4 left one space,
- then move piece 7 up 3 spaces and left 1 space,
- then move piece 5 right one space then up 2 spaces,
- then finally move piece Z right 2 spaces.

The result looks like this:



When asked to solve the puzzle, your program MUST find the shortest solution for these types of puzzles. The solution will always have the goal piece move to the right hand side of the puzzle grid (i.e. once the goal piece moves into the last column). Once the solution is known, your interface should use animation to show the sequence of moves that gets the current state of the puzzle to the puzzle's solution. You are to be using a Breadth-First Search to find this sequence of moves. Note: a puzzle may have many solutions and may have multiple solutions of the same shortest length. You only need to find one of these shortest solutions. Also note that a puzzle might not be solvable.

A shortest solution for the above is to be given in the ordered list of moves shown below. Each move is to show the Piece moved, the direction (up, down, left or right) and the number of spaces moved in that direction. The solution for the above puzzle is as follows:

1.	Piece 4	left	1 space
2.	Piece 7	up	3 spaces
3.	Piece 7	left	1 space
4.	Piece 5	right	1 space
5.	Piece 5	up	2 space
6.	Piece Z	right	2 space

The shortest solution will have the fewest number of moves. Note that there could be multiple shortest solutions for a puzzle. The above solution could have reversed moves 3 & 4 and still have been the shortest solution.

Your program is to treat the following as 1 move:

2.	Piece 7	up	3 spaces

while the following is considered 3 moves:

2.	Piece 7	up	1 space
3.	Piece 7	up	1 space
4.	Piece 7	up	1 space

Note that the following two moves **CAN NOT** be combined into a single move since the direction that the piece is moving changes.

2.	Piece 7	up	3 spaces
3.	Piece 7	left	1 space

This is the same for the following two moves:

4.	Piece 5	right	1 space
5.	Piece 5	up	2 space

Input Format

The input for a puzzle will always come from a file. The exact form of the input file is left up to you, but a suggested form is as follows (which is used in the sample input files).

- The first line of the file will contain 2 integers, the number of rows and the number of columns of the puzzle "grid". If either of these values is zero or less, print an appropriate error message and end the program. These values will be separated by one or more white space characters. The puzzle will always use a rectangular grid.
- The second line of the file will contain the starting position of the goal piece.
- The remaining lines of the file will contain the starting position of the other pieces in the puzzle.
- Once the end of the file is read, then all of the pieces have been read in.

Each piece will always have a rectangular shape (while such puzzles with non-rectangular shape do exist, it adds a complexity we don't need to deal with here). Each piece's starting position is given by 4 integer values and one character value. These values will be separated by one or more white space characters.

- The first integer will be the starting row position.
- The second integer will be the starting column position.
- The third integer will be the width in columns.
- The fourth integer will be the height in rows.
- The character value will specify the direction of movement the piece can have. This character can be either an "h" for horizontal movement (left or right), a "v" for vertical movement (up or down), a "b" for both horizontal and vertical movement, or a "n" for no movement (the piece cannot move, it must stay in that space).

If a piece would fall outside of the puzzle grid, have an invalid direction of movement, or overlap with another piece, an appropriate error message should be printed and the piece should be discarded from the puzzle (i.e. don't quit the program). If the goal piece is listed incorrectly, the first correctly listed piece becomes the goal piece. The upper left corner of the grid has row = 1 and column = 1. The input for the above puzzle would be as follows:

```
4   4

3   1   2   1   b

1   1   1   1   b

1   2   1   1   b

2   1   1   1   b

2   3   2   1   b

3   3   1   2   b

4   1   2   1   b

4   4   1   1   b
```

Note that the names of the pieces are not specied in the input file. The goal piece will always have the name of "Z". The next nine pieces will have the names from "1" to "9". The next 26 pieces will be given names using the lower case letters from "a" to "z". The next 25 pieces will be given names using the upper case letters from "A" to "Y" (since "Z" is already in use). If the file has more than 61 pieces, come up with some additional naming scheme. You can assume a puzzle will have less than 128 pieces.

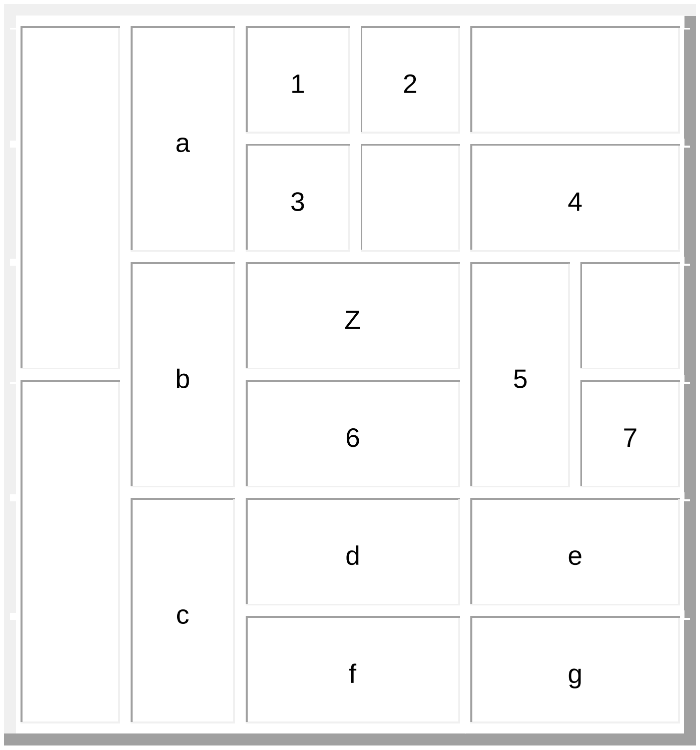
Sample Input

The traditional “rush hour” game normally is played on a grid of 6x6. Since you are creating your own data for the assignment, you can keep all of the grid sizes to be the same.

Some sample input data files are:

- proj3a.data: The one listed above
- proj3b.data
- proj3c.data: This one has no solution
- proj3d.data
- proj3e.data
- proj3f.data
- proj3g.data: Error in input
- proj3h.data: Error in input
- proj3i.data: Error in input
- proj3j.data: Error in input
- proj3k.data: Three piece puzzle
- proj3l.data: Error in input
- proj3m.data
- proj3n.data

Smaller puzzles can be made into a larger one by putting a columns and rows of unmovable piece around the existing puzzle.



Assuming that pieces a,b and c can only move vertically and pieces d, e, f and g can only move horizontally. The original 4x4 puzzle is transformed into a 6x6 puzzle.

You are more than welcome to restrict the types of pieces the puzzle can have but it needs to have the following 4 types of pieces as a minimum: a 2x1 piece that can move vertically, a 3x1 piece that can move vertically, a 1x2 piece that can move horizontally, and a 1x3 piece that can move horizontally. These are the pieces that you traditionally find in this type of puzzle.

Program Output

Your output is to inform the user when the puzzle has been solved. Also, if the puzzle is not-solvable, you should inform the user of this as well.

Use of the Java Foundation Classes Library's Data Structures

Your program is to take full advantage of the Java Foundation Classes Library. This library contains classes for a number of data structures that can be used in solving this problem. The idea is "why write your own code if someone else already has done it". This is not to be taken to the extreme of "borrowing code" from fellow class members.

To store your data, you will need to create a class to hold each piece. The puzzle will be a collection of these pieces. Since the number of pieces is unknown, this should be dynamic. The use of the classes of **List**, **ArrayList** or **Vector** would be a good choice here. When trying to find the shortest number of moves, the breadth-first search algorithm will work. Since the breadth-first search uses a queue, the use of the class of **Queue** is good choice when implementing a breadth-first search.

Algorithm Ideas and Hints

A good algorithm to solve this problem is to create a class that will hold a "snapshot" of the puzzle. A snapshot is to contain all needed information about the "current" state of the puzzle. The current state is the current position of all pieces in the puzzle and what moves it took to reach the current state from the initial state. So the snapshot needs two main sets of data:

1. the pieces with their positions
2. a list of moves

The initial state/snapshot is the position of the pieces as given in the input file and no moves (an empty list of moves). Then all of the snapshots that could be created from moving a single piece from the initial snapshot are added onto a queue (be sure to add the move information to the list of moves).

If moving the piece causes piece Z to move to the right-most column of the puzzle, the puzzle is solved and the list of moves which contains the solution is printed.

Otherwise the first snapshot is removed from the queue and all of the snapshots that could be created from moving a single piece from this snapshot are added onto the queue. If we attempt to remove a snapshot from the queue and the queue is empty, the puzzle has no solution.

Of course, we have to make sure that each specific arrangement of pieces is only added onto the queue one time; otherwise, we may get into an infinite loop. This is step from the Breadth-First Search algorithm were a node is marked as visited. One way to do this is to create a simplified version of the current layout of the pieces and to store and compare this simplified version using a JFC **Set**. One way to create a simplified version is to create a string from the puzzle. The string would have (# of rows)x(# of column) characters and the first (row-length)-th character would have the piece names from the first row (using a space character for an empty position in the puzzle. , the second (row-length)-th characters would have the piece names from the second row... For example the initial puzzle from above would have the following 16 character string:

```
    "12  3 44ZZ5 6657"
```

and the solution of the puzzle would have the following 16 character string:

```
    "12753445  ZZ66  "
```

This idea is nice, since comparisons are already defined for strings, we don't have to make up some complicated algorithm to determine if two snapshots have all of their pieces in the same positions.

Identifying Piece Moves

What are the possible first moves from the initial puzzle shown above? The first piece I would check would be the Z piece, since moving that piece determines if the puzzle is solved and if a solution is found, the remaining pieces don't have to be moved. The following is a piece by piece listing of the first moves from the inital puzzle:

- Piece Z
  - Can not be moved
- Piece 1
  - Can not be moved
- Piece 2
  - Move down 1 space
  - Move right 1 space
  - Move right 2 spaces
- Piece 3

    - Move right 1 space
- Piece 4
    - Move up 1 space
    - Move left 1 space
- Piece 5
    - Can not be moved
- Piece 6
    - Can not be moved
- Piece 7
    - Move up 1 space

Therefore, after the first initial snapshot is taken care of, the queue should have 7 snapshots on it. One for each of the moves mentioned above.

Multiple Source Code Files

This program will require the use of multiple source code files and separate compilation. The division of the subroutines between the multiple source code files must be logical. One suggestion would be to have each class in its own file (or one file with all classes) and the command interface code in another source code file. Such classes might include a **piece** class and a **grid** class. While the above does not state that multiple classes are required, it is strongly implied!!

Programming Style

Your program must be written in good programming style. This includes (but is not limited to)

- Multiple Source Code Files
- Meaningful Variable Names
- Proper Indentation of Code
- Blank Lines between Code Sections
- Use of Methods and Functions
- Use of Multiple Classes
- In-Line Commenting
- Header Comment for the File
- Header Comments for each Method, Function and Class.

The work you turn in must be 100% your own. You are not allowed to share code with any other group (inside this class or not). You may discuss the project with other persons; however, you may not show any code you write to another group nor may you look at any other group's written code.

Project Submission

Follow the directions on the class web page for the creation of a web page to house your final version of your program.