CS 474 – Object Orient Languages and Environments

Programming Project 1, Spring 2016

Sliding Block Puzzles

Due: Thursday, February 11, 2016 at 11:59 pm

This project is to be written using the Smalltalk Language.

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.

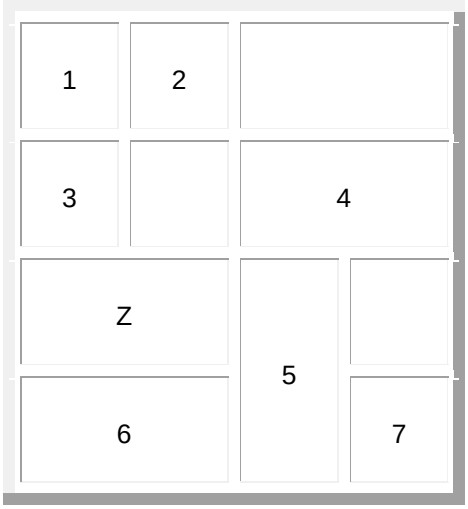
Such games or puzzles can be found under the names of Traffic Jam, Rush Hour, Parking Lot, Blocked, Unblock Me, etc. Check out:

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

Your program is to 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).

Example of a Sliding Block Puzzle

Consider the following puzzle:

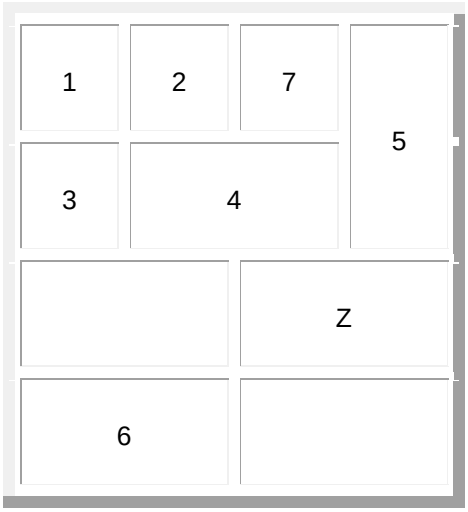


The puzzle contains 8 pieces. The piece labeled "Z" is the "goal piece". For this puzzle to be solved, The “Z” piece must be moved to the right hand edge of the puzzle. In this puzzle all pieces can move any direction we wish (left/right or up/down). One solution is to:

- 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 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
```

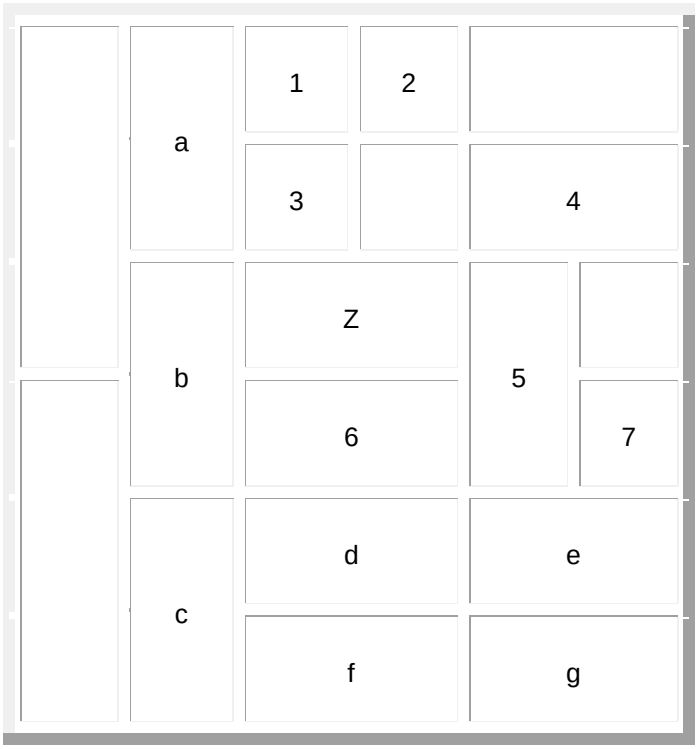
Note that the names of the pieces are not specified in the input file. The goal piece will always have the name of "Z". The next nine pieces will have the names from "1" to "9". The next 26 pieces will be given names using the lower case letters from "a" to "z". The next 25 pieces will be given names 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

Some sample input data files are given below. Note the names of the files reflect a name used for a previous semester (but the links are all still valid):

- 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

The rush hour game is played with a 6x6 grid, but this is not a requirement here. However, 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.

Program Output

It is possible that a puzzle may have no solution. Your output from the program should be written to standard output and should consist of four parts:

Any error messages generated by invalid input.

A listing of the grid as the start of the puzzle. This can be a simple ASCII graphic as shown below:

```
******
*12  *
*3 44*
*ZZ5 *
*6657*
******
```

Perhaps using periods instead of spaces makes this a bit more readable:

```
******
*12..*
*3.44*
```

```
*ZZ5.*
*6657*
******
```

Of course, this grid is less readable if the grid so large that it causes line wrap (but this is not the programmer's problem).

Then list out the moves that solve the puzzle or a message stating the puzzle is not solvable.

A list of the grid showing the solution, if one exists. For example:

```
******
*1275*
*3445*
*..ZZ*
*66..*
******
```

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.

Algorithm Ideas and Hints

The solution to this problem is really doing a breadth-first search on a graph. The nodes in the graph show all of the possible arrangements of the pieces in the graph. The edges in the graph connect any two nodes that differ by only performing a single move on one piece in a valid manner. The breadth-first search would start from the initial arrangement of pieces as specified in the input and finish when an arrangement of pieces is encountered that has the Z piece in the right most column.

The main problem to the above idea is that such a graph for most sliding block puzzles is huge. Also we only know the starting arrangement of the pieces. So following a traditional approach would first require us to build the entire graph before we run the breadth-first search algorithm.

So the better solution that we will use is be “build the graph” as we work our way through the breadth-first search. Once any arrangement of pieces is known, we can algorithmically determine which arrangements of pieces are neighbors in the graph. So we start the search with just the initial arrangement of pieces as specified in the input file as the only node in the graph. As the search progresses, additional nodes are determined As it turned out, this solution relies so heavily on the breadth-first search that the actual graph data can be replaced with a queue that performs the bread-first search..

A good approach 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 **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 initial 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.

Programming Style and Restrictions

Your program must be written in good programming style. This includes (but is not limited to) the follow. Note that we understand that some items on this list may not directly have counterparts in Smalltalk (i.e. Multiple Source Code Files).

- 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. This project wants you to create your own list class, queue class, map class, hash class, etc (whichever "standard" classes you may be using) instead of using built-in classes in Smalltalk. Use of the Smalltalk Array class is permitted. If you are unsure if you may use a class or not, assume you may not, but you can ask for permission from the instructor. This restriction is meant to apply toward "data structure typical" classes.

You are not allowed to share code with any other person (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.

Program Execution

Your program is to contain a class named **SBSolver**.

This class is to have a **new:** message that takes a string as an argument. This string is to be the filename of a datafile that contains the sliding block puzzle data in the format described above. Sending this message should cause the entire program to run.

```
SBSolver new: 'sbdata1.txt'
```

You may wish to add other ways of starting the program to help with your own development.

Project Submission

Directions will be post on the class web page for submission of the project.