CS 340 - Software Design

Machine Problem 2, Spring 2013

Sliding Block Puzzles

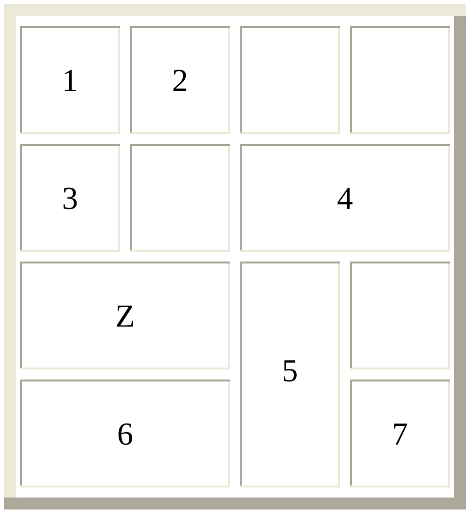
Due: Tuesday, February 19, 2013 at 11:59 pm

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

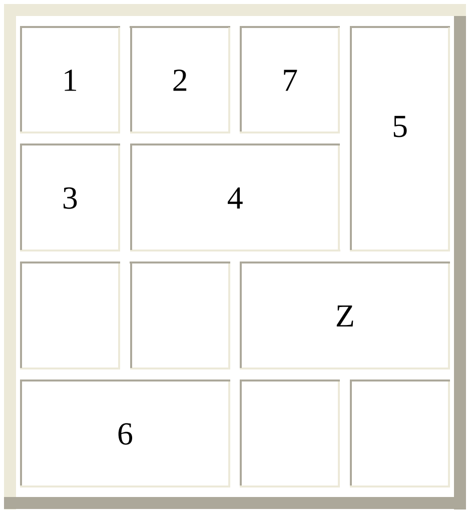
Consider the following puzzle:



The puzzle contains 8 pieces. The unmarked areas of the puzzle are meant to be empty (but there was some trouble getting the graphic to look just right). The piece labeled "Z" is the "goal piece". This 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:



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). Your program is to prompt the user for the name of a data file containing the needed information about the puzzle. The output of the puzzle will be a sequence of moves that give the shortest solution (or one of the shortest solutions if two equal length shortest solutions exist).

The solution is to be given as an ordered list of moves. 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. Finding any one of these shortest solutions is good enough for this programming assignment. 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 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

Some sample input data files are:

- proj2a.data (The one listed above)
- proj2b.data
- proj2c.data This one has no solution.
- proj2d.data
- proj2e.data
- proj2f.data
- proj2g.data Error in input
- proj2h.data Error in input
- proj2i.data Error in input
- proj2j.data Error in input
- proj2k.data A 3 piece puzzle
- proj2l.data Error in input
- proj2m.data
- proj2n.data

Program Output

Your output from the program should be written to standard output and should consist of four parts:

1. Any error messages generated by invalid input.
2. 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).

3. Then list out the moves that solve the puzzle or a message stating the puzzle is not solvable. It is possible that a puzzle may have no solution.

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

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

Extra Credit

For 5 points extra credit, after the above output is given ask the user if they wish to also write the solution to an html file. The html filename will be the same as the input data filename with the file extension of ".html". The output in the html file should contain the same information as the standard output, but it should be formatted better. The grid(s) in this html file should use html tag of <table> as was shown in the write-up of this assignment. The list of moves should show the information is some neat arrangement. How you do this is left completely up to you (do not expect hints from either the instructor or the TA).

Use of the Supporting Classes

You must write your own classes to solve the problem. You may not use the C++ Standard Library or the Qt Libraries for "data structure" classes.

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. A class that acts similar to a **vector** or **list** 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 creation of a **queue** class is needed.

Algorithm Ideas and Hints (and other Classes)

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: a list of the pieces with current positions and a list of moves. The initial state/snapshot is the position of the pieces as given in the input file and zero moves (an empty move list). 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 move list which contains the solution is printed. Then 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 arrangement of pieces only is added onto the queue one time; otherwise, we may get into an infinite loop. 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 the operations associated with a **set class**. 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. Whether you create an actual set class or create another class the implements the needed operations is left up to you.

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.

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.

Note: a "source code file" is not the same as a "header file". Source code files will have a file extension of .cpp (or .c, .C, .CC, etc.), while a header file will have a file extension of .h. To perform separate compilation, your makefile must separately compile the source code files into ".o" files and then link the ".o" files together. Using a #include statement with a source code file (a .cpp file) will NOT satisfy a requirement of separate compilation.

In addition to using and submitting a makefile, this program will also require a 1-2 page write up of the data structures used in the program and the logical division of your program into multiple source code files (i.e. which routines are where). Remember that this write-up is to be written in ASCII format and is to be electronically turned in with your program. The name of this file should be "readme.txt". Also recall that your program will be given to another student to write a critique. Therefore, it is suggested that you do not include your UIN in your program. Instead use your name and your NetID to identify yourself.

Programming Style

Your program must be written in good programming style. This includes (but is not limited to) meaningful identifier names, a file header at the beginning of each source code file, a function header at the beginning of the function, proper use of blank lines and indentation to aide in the reading of your code, explanatory "value-added" in-line comments, etc.

The work you turn in must be 100% your own. 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 person nor may you look at any other person's written code.

Project Submission

You are to submit this project using the assignment link in Blackboard for project 2.