University of Toronto
CSC148—Introduction to Computer Science **St. George Campus, Fall 2007**

Assignment 2: Sudoku Puzzle

Due: 10:10am, Mon. Nov. 5 (before class).

Purpose. This assignment is mainly for fun, something you deserve for making it this far in CSC148. By solving this assignment you will gain some experience thinking about algorithms, and even more experience reading and writing Java code.

Marking. This assignment is worth 10% of your final mark. We will not mark all parts of this assignment.

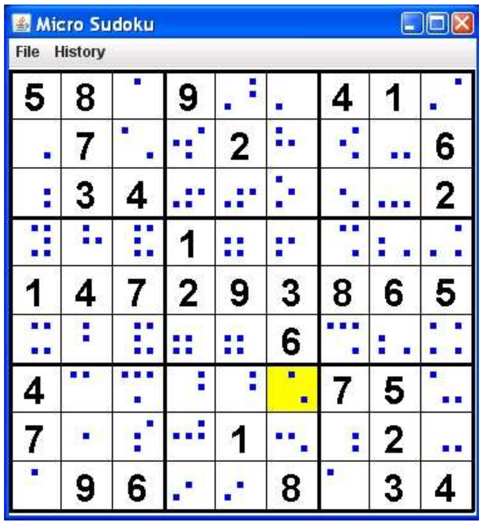

Figure 1: Sudoku game screenshot (view in colour).

Take Note of any Vital Rules. Throughout this course a "vital rule" means something that you must follow if you are to get any marks at all for the question. If you break a vital rule, you will automatically get a mark of zero on the corresponding part of the assignment. This might seem overly strict, but we want you to practice doing things a certain way. We use these vital rules to make sure you don't work around the main point of the question. We have listed the vital rules for this assignment further below.

Preparation. Make an A2 directory on your hard drive. Copy A2handout.zip from the course homepage to this directory. Unzipping this file will create several directories containing the handout "*.java" files. Finally, move A2handout.zip out of this directory, so you don't overwrite your work by mistake.

In the A2 directory, load GameApp.java into DrJava and click compile and then run. A window like the one in Fig. 1 will be popped up. You can move, minimize, or resize this window, just like any other window on your desktop. You can move the "focus" to any cell by clicking on it (it turns yellow). Cells with black numbers in them are frozen, the entries cannot be changed. Typing a number between 0 and 9 in a non-frozen cell changes the entry in that cell (0 clears the cell). If the entered number is red, that means it is violating one of the Sudoku rules. Valid numbers are shown in blue or black. (For the rules of Sudoku, see assignment 0, or the puzzle section of your favorite newspaper.) Finally, the "pencil marks" in blank cells indicate the valid moves, with a blue mark at the appropriate point in a 3×3 grid if and only if the corresponding number is a valid entry for this cell of the current board. Within one cell, the numbers 1-3 are indicated by pencil marks on the first row, 4-6 on the second, and 7-9 on the last row. With a little practice you will be able to understand the significance of these pencil marks.

The File menu works in the handout code. It can be used to load a new puzzle from an input file, and to exit the application. But the items on the History menu are not complete in the handout code. Your job will be to write the code supporting the History menu items.

Vital Rules.

1. In SudokuControl you can change the methods undoMove(), redoMove(), makeMove(SudokuMove), and solve(boolean). You also need to change their method comments. However you cannot change the parameters nor return type of these four methods. You cannot change the code anywhere else within SudokuControl, except to add private helper methods. You cannot use the keyword "new" in any of these methods, except to create instances of SudokuMove, Index2D, or any subclasses of Exception objects. You cannot create new arrays of any type.

2. In SudokuBoard you can only change the method nextOpenMinOptions(). Here you can add or delete any code within this method. You can add private helper methods to this class, if you like, but otherwise you cannot change anything outside of the nextOpenMinOptions() method. Also, you cannot add any method parameters, nor change the return type of nextOpenMinOptions().

3. All other Java files in the handout code must be left unchanged.

Definitions. Typing a number (0 through 9) in a selected non-frozen cell is considered a **move**. In addition, typing the up- or down-arrow key, the delete, or backspace key in a non-frozen cell (all of which are equivalent to typing 0 to clear the cell), are all considered to be moves. A move can violate the Sudoku rules (i.e., entering an invalid digit in a cell is still considered a move).

An **undo/redo request** occurs when the user selects the Undo or Redo menu items from the History menu, or when the left-arrow or right-arrow keys are pressed (of course, this only applies when the window is "listening" to the key presses, i.e., when the cursor is within the screen area of the board itself).

Your Job.

1. Undo requests are currently ignored. The goal of the Undo operation is to be able to Undo the previous move. In fact we want to be able to undo right back to the initial game board by repeatedly making undo requests. To do this we will maintain a stack of moves. One restriction on sequences of undo operations is that only a maximum number of them can be remembered at a time, due to the limited capacity of the stack used to store them.

   Each time an undo request is made, the undoMove() method in SudokuControl.java is called. Your first job is to implement this method (it takes no more than a dozen lines of Java). Your implementation should make use of two Stack instance variables named moves and undoneMoves. By the time the board is first displayed on the screen these two stacks have already initialized to be empty stacks by the method initBoard(String) in SudokuControl.

   To get this to work you need to modify methods makeMove(SudokuMove m) and undoMove(), both in SudokuControl. The idea is that makeMove(SudokuMove) is done anytime a move is made (see definition above). And undoMove() is called anytime a user enters an undo request. Working together, these two methods need to maintain the stack of SudokuMoves named "moves". Hint, when you undo a move, make sure the change in the board due to this undo is **not itself pushed on the moves stack**.

2. Here you will implement the History menu item Redo (right-arrow on the keyboard is the associated shortcut). The goal of the Redo operation is to let the user redo an undone move. Moreover, successive Redo operations should successively redo a stack of undone moves. For example, if we successfully complete a Sudoku (say, by only typing non-zero numbers in empty cells, so we can remember all the moves in the "moves" stack), then we should be able to repeatedly type the left-arrow key to reverse

the sequence of moves all the way back to the original board, and then refill the same solution with the original order of moves by repeatedly pressing the right arrow key.

However, if the user undoes a few moves and then types any number into a non-frozen cell or clears that cell (even if the typed entry is the same as the previous entry), then the redo stack is cleared. In other words, a redo request immediately after any move must fail (i.e., your method, redoMove() should return false).

This behaviour allows the user to explore the puzzle by making a series of moves, undoing the most recent moves, and then continuing on with a second series of moves, possibly again undoing the most recent moves, and so on.

In order to get this to work you can change any of the three SudokuControl methods (but note the vital rules above):

```
boolean undoMove()
boolean redoMove()
void makeMove(SudokuMove m)
```

When you are done, write appropriate method comments for the above three methods.

3. Play with the current version of the Sudoku game. It might be best to first load an "easy#.txt" case by using the File→New menu item. Try to discover the `Alzheimer's Sudoku Algorithm`. This algorithm is guaranteed to solve any Sudoku puzzle. Moreover, very little needs to be remembered by the user at any stage of the solution. Most of the memory required is in the state of the Sudoku board itself.

   In particular, the Alzheimer's Sudoku Algorithm is to be specified in terms of operations on our Java game board. To execute the Alzheimer's algorithm all the user needs to remember is the algorithm itself (which is a small set of rules) plus whether the user is currently moving forward (adding new numbers in cells), moving backwards (undoing moves), or turning from going forward to going backward, or vice versa. Remembering these four states, along with the algorithm itself, is enough to run the Alzheimer's Sudoku Algorithm to solve any puzzle (given enough persistence).

   On a separate piece of paper, in English, write out the Alzheimer's Sudoku Algorithm. You can separate it into six paragraphs, with each paragraph describing what to do in one of the six stages: starting, being in one of the four states described above, and having completed the puzzle.

4. Implement your Alzheimer's Sudoku Algorithm in the solve(boolean minOption) method in SudokuControl. This method is called with the argument set to false when "Solve 1" is selected from the History menu. The first three steps of the method should be to clear the board (remove all the blue numbers), and clear the moves and undoneMoves stacks. After that, you are on your own.

   The first constraint on this method is that you cannot use any long lists of items, other than what is already provided by the current state of the board along with the moves and undoneMoves stacks. Otherwise, you aren't minimizing the memory requirements, which is part of the Alzheimer's algorithm. The only exception to this is that you can store the ArrayList of move options at one cell, as returned by the getOptions method in Board.

   The second constraint on this method is that (given the minOption argument is false), when moving forward, the next blank cell to be selected is chosen in a top to bottom, left to right order. That is, the method board.nextOpen() is used to select the next cell to be filled.

5. A smarter variation on the Alzheimer's algorithm is, when moving forward, to select the next cell to be filled as one of the ones with the minimum number of options. (Again, if there is a tie here, we should use a top to bottom, left to right priority.) The method nextOpenMinOptions() in SudokuBoard.java finds such a cell. Complete this method according to its method comment.

   When the "Solve 2" item is selected on the History menu, the same solve(minOption) method as in part 3 above is called. Only now the argument is set to true. For this value of minOption, your solve method should call nextOpenMinOptions() method instead of nextOpen() whenever the algorithm is moving forward. This involves only a small modification of your previous solve(minOption) method.

Which of the two solve methods is faster? Note that nextOpenMinOptions() is looking through the whole board, while nextOpen() only has to search up to the next open cell. So we might expect nextOpenMinOptions() to typically take longer to run than nextOpen().

6. A direct big-Oh runtime analysis does not help to determine the relative speeds of these two solver algorithms. The reason is that no parameter is getting large, so the most we can say is that both algorithms run in time $O(1)$.

   However, another quantity that is of interest is the "risk" of the history of moves we have made. This history is taken from the intial state to the current state. After selecting a Solve 1 or Solve 2 operation, this history is actually stored on the moves stack in SudokuControl. Define the risk to be one over the probability that all the choices we have made so far are correct. For example, if on the first move we selected a cell with m_1 options, and filled in one of these options, then the probability that we chose correctly could be modelled as $1/m_1$. Similarly, if the second move had to select amoungst m_2 options, then the probability that both this move and the previous move was correct is $(1/m_1)(1/m_2)$. After K moves we find the probability we have chosen everything correctly so far is $1/[m_1 m_2 \ldots m_K]$. We defined the risk to be one over this probability, so the risk of our current sequence of moves is $m_1 m_2 \ldots m_K$. One complication here is determining the risk of clearing a cell (entering a zero). We can ignore this here since neither of the solve algorithms should ever push these moves on the "moves" stack.

   At the end of the method solve (in SudokuControl), compute the risk for the sequence of moves that ends up stored on the moves stack. Print out this risk (using a System.out.println call within the method solve). The result will show up on DrJava's console and interaction pane, not on the Sudoku puzzle itself.

   How do the risks of the two different solve methods compare when run on the same problem? Also, roughly speaking, how do the risks for the solve 2 algorithm compare across different grades of problems (easy, med, hard or evil)?