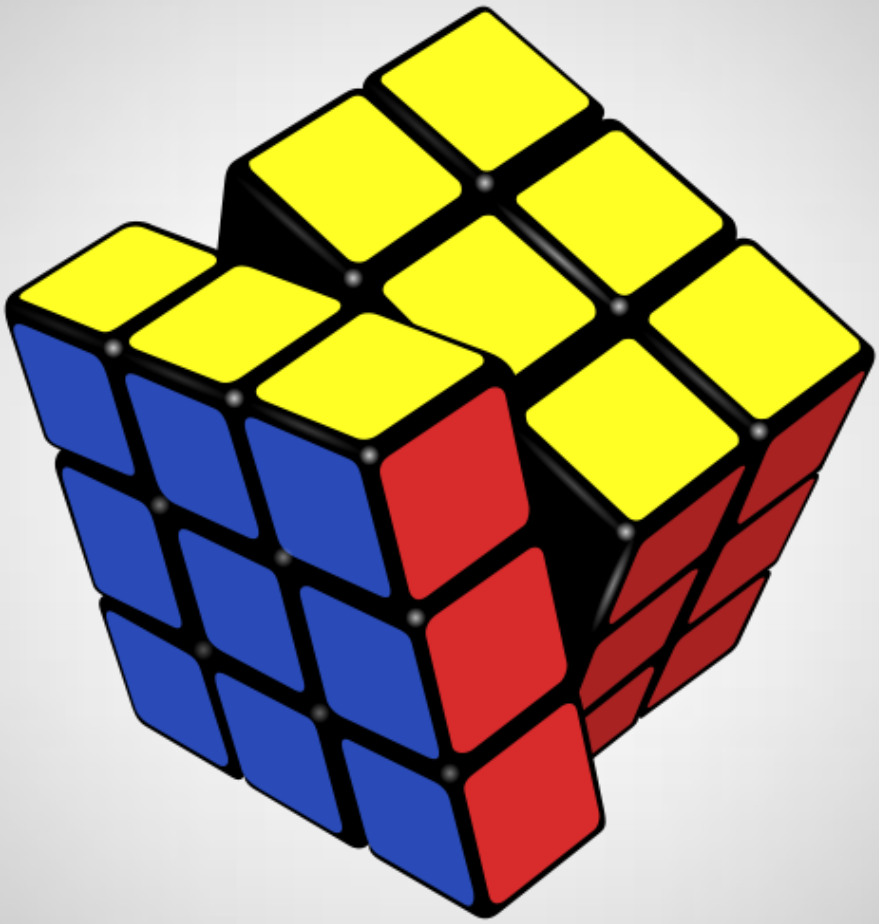
SECTION 4

SYSTEM MAINTENANCE



Contents

Environment Used

I used the IDE (Integrated Development Environment) 'Eclipse' (Juno Service Release 1) to develop the system. Eclipse is one of the two main IDEs used to develop Java applications, the other being Netbeans. I chose Eclipse over Netbeans because I am more familiar with Eclipse than Netbeans, and there are many more learning resources available for Eclipse.

There are important features that developers must be familiar with in order to unlock the full potential of the environment. These include:

Javadocs

A Javadoc comment can be added to any member by clicking on the identifier in the member declaration then pressing Ctrl + 1 (Quick Fix) and selecting 'Add Javadoc comment'. This will automatically generate a Javadoc comment for the specified member. For example, a method whose signature is

```
"public static double getBestAverageOf(int sizeOfAverage, int averageIndex,
LinkedList<Solve> times)"
```

will have the following comment generated:

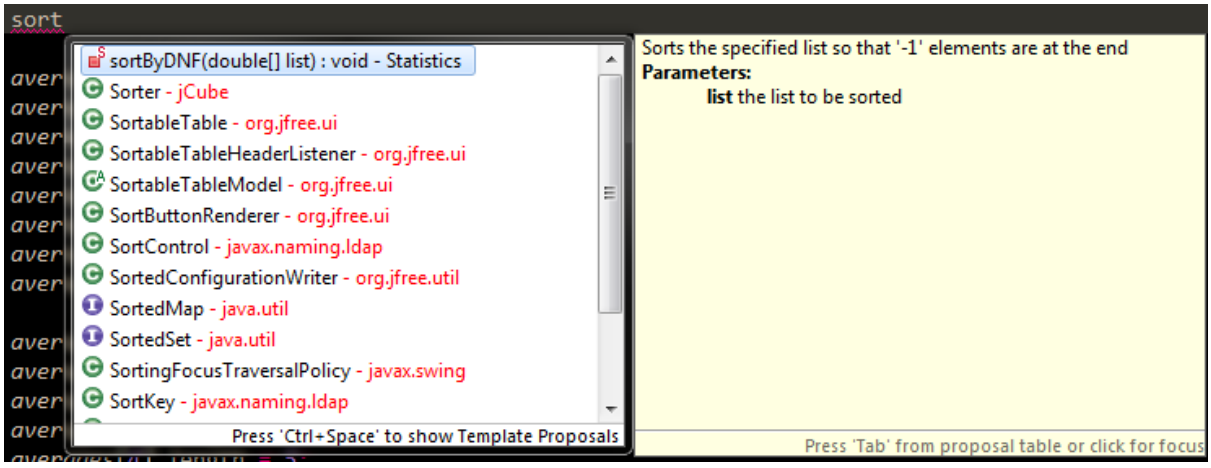
```
/**
* @param sizeOfAverage
* @param averageIndex
* @param times
* @return
*/
```

and the developer can then write a description for the method, descriptions for each of the parameters and an explanation of what is returned.

To view the Javadoc for a particular member, you can place the cursor over any reference to the member in the code, i.e. not necessarily the declaration, and the relevant Javadoc comment will be displayed. Linked Javadocs can be generated for the entire project by right-clicking on the project in the package explorer → Java → Javadoc, then choose the destination and the Javadocs will be generated and can be viewed in a web browser.

Content Assist and Quick Fix

When working with many variables and methods, it can become difficult to remember the identifiers, parameters, and return types used. To help this, you can use the content assist feature. By pressing Ctrl + Space, you will get suggestions for which variables/fields/methods you wish to use. For example, in the Statistics class (see page 410 of the appendix section), there are many methods with different return types. If I want to use the sortByDNF(…) method, but I can't quite remember its signature or return type, then I can type "sort" and then press Ctrl + Space to show the content assist pop-up. The method has been suggested at the top, the parameters, double[] list, are given, and the return type, void, is specified, and the corresponding Javadoc for this member is shown to the right.



If some of the code is underlined with yellow, then this indicates a warning; if some of the code is underlined with red, then this indicates an error. If you are unsure of how to fix the problem, or you want suggestions of how to fix the problem, click within the underlined area, then press Ctrl + 1. For example, in the image shown below, the string "Hello" is being assigned to the integer variable 'i'. Since "Hello" is not an integer (and cannot be implicitly cast), it is underlined in red.

```
public static void main(String[] args) {
    int i = "Hello";
    System.out.println(i);
}
```

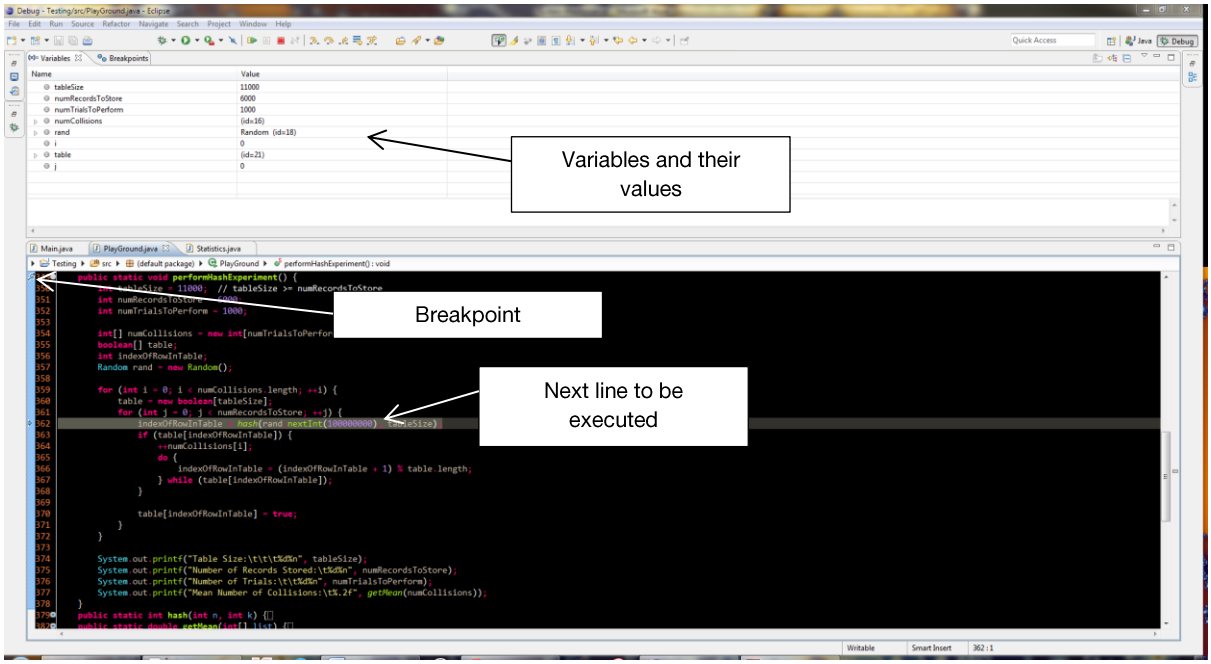
An quick solution to this would be to change the data type of 'i' so that it can store the string. Clicking somewhere within "Hello" (the underlined area) and pressing Ctrl + 1 shows suggestions and shows the resulting code if the suggestion is applied.

It is no longer flagged as an error after applying the suggestion.

```
public static void main(String[] args) {
    String i = "Hello";
    System.out.println(i);
}
```

Debug Mode

You can execute the program in debug mode so that variables' values can be tracked and execution of particular methods controlled. To do this, click on the signature of the method that you wish to be executed in real-time, then Run → Toggle Breakpoint. To execute the program in debug mode, Run → Debug. Once the execution of the program reaches the method you selected, you can control the execution line-by-line (like an interpreter) and see the variables' values etc. You can control the execution with the Run menu's menu items.



Tips and Tricks

- The Source menu has many useful features, such as 'Generate Constructor using Fields' which opens a wizard allowing you to generate a constructor which uses all of the fields in its parameters.
- To see the hierarchy of calls for a particular method, click on a reference to the method in the code then Navigate → Open Call Hierarchy.
- To automatically add missing imports, press Ctrl + Shift + O.
- To change a variable's identifier throughout the entire workspace, click on the identifier, press Ctrl + 1, then select 'Rename in workspace' and type the new name for the variable and it will be updated throughout the entire workspace.
- To automatically indent and format the code according to the accepted convention for Java, use Source → Format.

System Overview

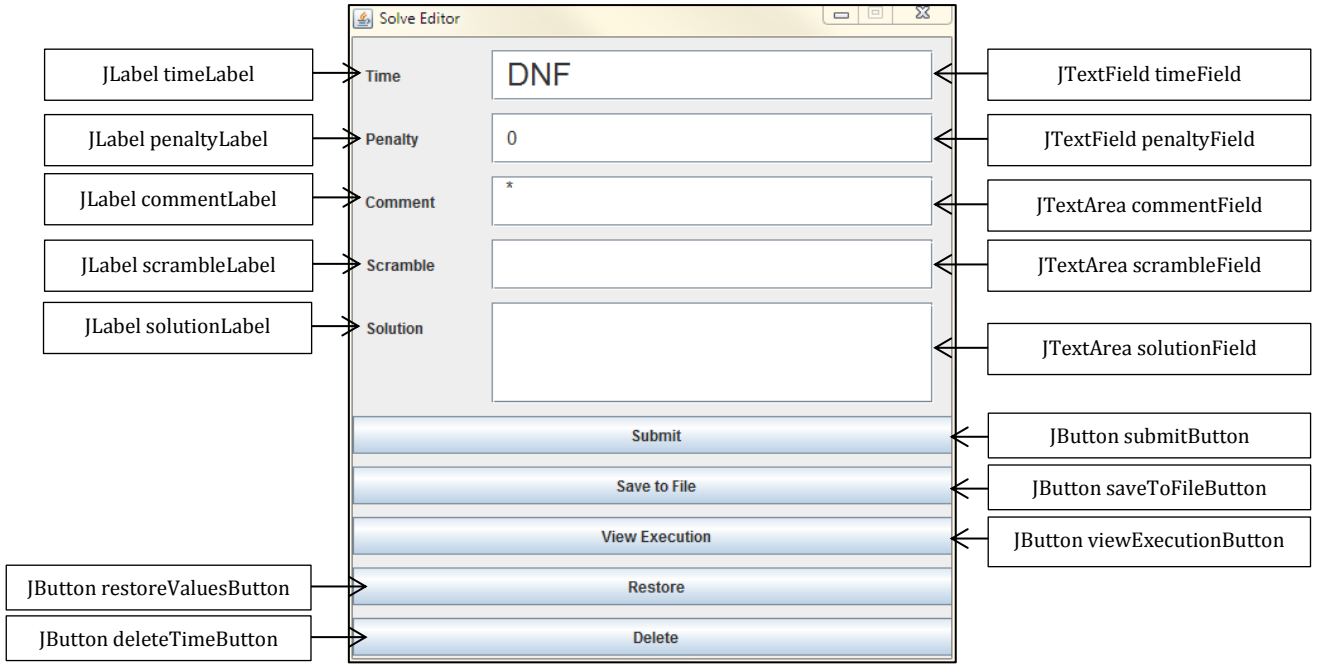
The system allows users to do the following:

- Learn how to solve the Rubik's cube using built-in and custom-made tutorials.
- Automatically generate solutions for a given cube-state.
- Paint custom states onto a virtual Rubik's cube.
- Practise solving the Rubik's cube using randomly generated scrambles.
- Save/load cube states, solve information, statistics and scrambles.
- View a graph of your times and save this graph as an image.
- Add, edit, and delete member details.
- Add, edit, and delete algorithms.
- Add, edit, and delete solve information.
- Add, edit, and delete scrambles.
- Navigate through the system with ease.
- Sort solves by time, date, or ID.
- Filter/search solves by time.

(For information on form/system navigation, see the user manuals)

Form Overview

Solve Editor Window



Button	JButton submitButton
Description	Clicking this button submits the data in the Solve Editor form for validation.

```
public void actionPerformed(ActionEvent arg0) {
   String time = timeField.getText().trim();

   if (!Solve.isValidTime(time)) {
      JOptionPane.showMessageDialog(null, helpMessage, "Error", JOptionPane.ERROR_MESSAGE, null);
   } else {
      if (time.equalsIgnoreCase("DNF")) {
         currentTime.setStringTime("DNF");
         currentTime.setPenalty("0");
      } else {
         currentTime.setStringTime(Solve.getPaddedTime(Solve.getSecondsToFormattedString(Solve
               .getFormattedStringToDouble(time))));
         currentTime.setPenalty(penaltyField.getText().trim());
      }

      if (penaltyField.getText().trim().equals(""))
         currentTime.setPenalty("0");

      currentTime.setComment(commentField.getText().trim());
      currentTime.setScramble(scrambleField.getText().trim());
      currentTime.setSolution(solutionField.getText().trim());
      setVisible(false);
      Main.refreshTimeGraph(true);
      Main.refreshStatistics();
   }
}
```

Button	JButton saveToFileButton
Description	Clicking this button opens a save dialog so that the information in the Solve Editor window can be saved to a chosen location.

```
public void actionPerformed(ActionEvent arg0) {
   String time = timeField.getText().trim();

   if (!Solve.isValidTime(time)) {
      JOptionPane.showMessageDialog(null, helpMessage, "Error", JOptionPane.ERROR_MESSAGE, null);
   } else {
      if (time.equalsIgnoreCase("DNF")) {
         currentTime.setStringTime("DNF");
         currentTime.setPenalty("0");
      } else {
         currentTime.setStringTime(Solve.getPaddedTime(time));
         currentTime.setPenalty(penaltyField.getText().trim());
      }

      currentTime.setComment(commentField.getText().trim());
      currentTime.setScramble(scrambleField.getText().trim());
      currentTime.setSolution(solutionField.getText().trim());
      setVisible(false);

      if (fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
         TextFile currentFile = new TextFile();

         try {
            currentFile.setFilePath(fileChooser.getSelectedFile().toString());
            currentFile.setIO(TextFile.WRITE);

            String comment = commentField.getText().trim();
            String scramble = scrambleField.getText().trim();
            String solution = solutionField.getText().trim();

            currentFile.writeLine(timeField.getText());
            currentFile.writeLine(penaltyField.getText());
            currentFile.writeLine(comment.equals("") ? "*" : comment);
            currentFile.writeLine(scramble.equals("") ? "*" : scramble);
            currentFile.write(solution.equals("") ? "*" : solutionField.getText());
         } catch (Exception e) {
            JOptionPane.showMessageDialog(null, "Could not save to file", "Error",
```

```
                    JOptionPane.ERROR_MESSAGE);
            } finally {
               currentFile.close();
            }
         }
      }
}
```

Button	JButton viewExecutionButton
Description	Clicking this button scrambles the cube in the main window and performs the solution in real time.

```
public void actionPerformed(ActionEvent arg0) {
   if (!(currentTime.getStringTime()).equals(timeField.getText())
         || !(currentTime.getPenalty()).equals(penaltyField.getText())
         || !(currentTime.getComment()).equals(commentField.getText())
         || !(currentTime.getScramble()).equals(scrambleField.getText())
         || !(currentTime.getSolution()).equals(solutionField.getText())) {
      JOptionPane.showMessageDialog(null, "You must submit the form first.", "Error",
            JOptionPane.ERROR_MESSAGE);
   } else {
      Main.performRealTimeSolving(currentTime.getScramble(), currentTime.getSolution());
   }
}
```

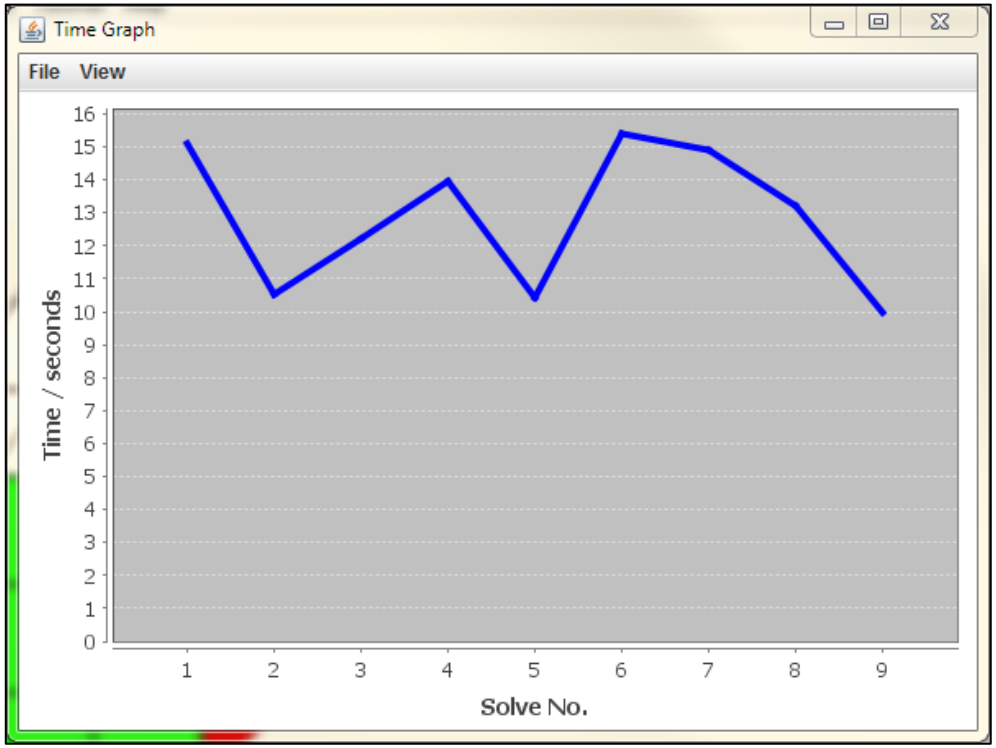
Button	JButton restoreValuesButton
Description	Clicking this button discards any changes and reloads the information in the window.

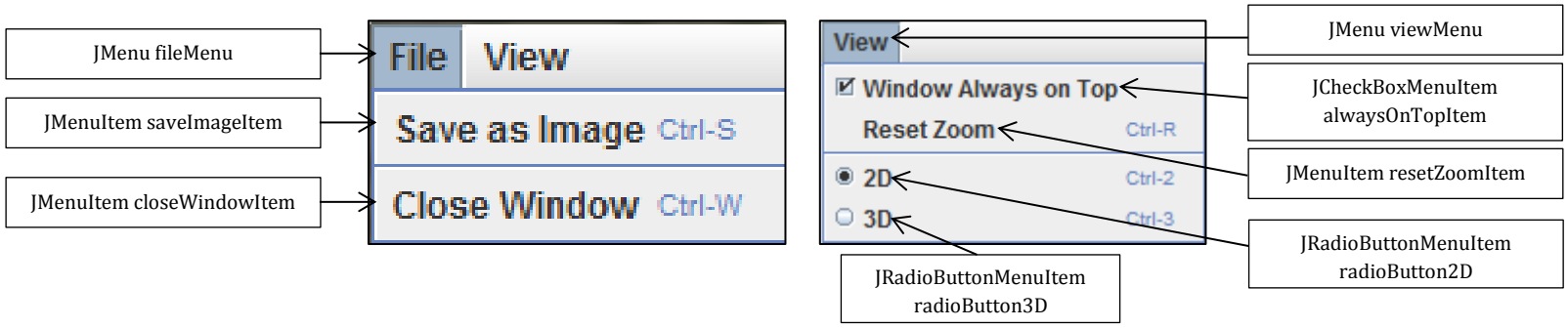
```
public void actionPerformed(ActionEvent e) {
   timeField.setText(currentTime.getStringTime());
   penaltyField.setText(currentTime.getPenalty());
   commentField.setText(currentTime.getComment());
   scrambleField.setText(currentTime.getScramble());
   solutionField.setText(currentTime.getSolution());
}
```

Button	JButton deleteTimeButton
Description	Clicking this button deletes the time/solve being viewed

```
public void actionPerformed(ActionEvent e) {
   currentTime.setStringTime("-1");
   Main.copyAllTimesToDisplay();
   Main.refreshTimeList();
   setVisible(false);
}
```

Time Graph Window





Menu Item	File Menu → JMenuItem saveImageItem
Description	Clicking this menu item opens a save dialog so that the graph can be saved as an image in the specified location.

```
public void actionPerformed(ActionEvent e) {
   boolean alwaysOnTopValue = alwaysOnTop;

   alwaysOnTop = false;
   setAlwaysOnTop(false);
   if (fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
      int width = 1024, height = 768;
      File lineChart = new File(fileChooser.getSelectedFile() + "");

      try {
         ChartUtilities.saveChartAsPNG(lineChart, chart, width, height);
      } catch (IOException exc) {
      }
   }
   alwaysOnTop = alwaysOnTopValue;
   setAlwaysOnTop(alwaysOnTop);
}
```

Menu Item	fileMenu → JMenuItem closeWindowItem
Description	Clicking this button closes the Time Graph window.

```
public void actionPerformed(ActionEvent arg0) {
   setVisible(false);
}
```

Menu Item	viewMenu → JCheckBoxMenuItem alwaysOnTopItem
Description	Clicking this button toggles whether the window is always on top.

```
public void actionPerformed(ActionEvent e) {
   alwaysOnTop = !alwaysOnTop;
   setAlwaysOnTop(alwaysOnTop);
}
```

Menu Item	viewMenu → JMenuItem resetZoomItem
Description	Clicking this button resets the graph to its default zoom.

```
public void actionPerformed(ActionEvent e) {
   resetGraphView();
   Main.refreshTimeGraph(false);
}
```

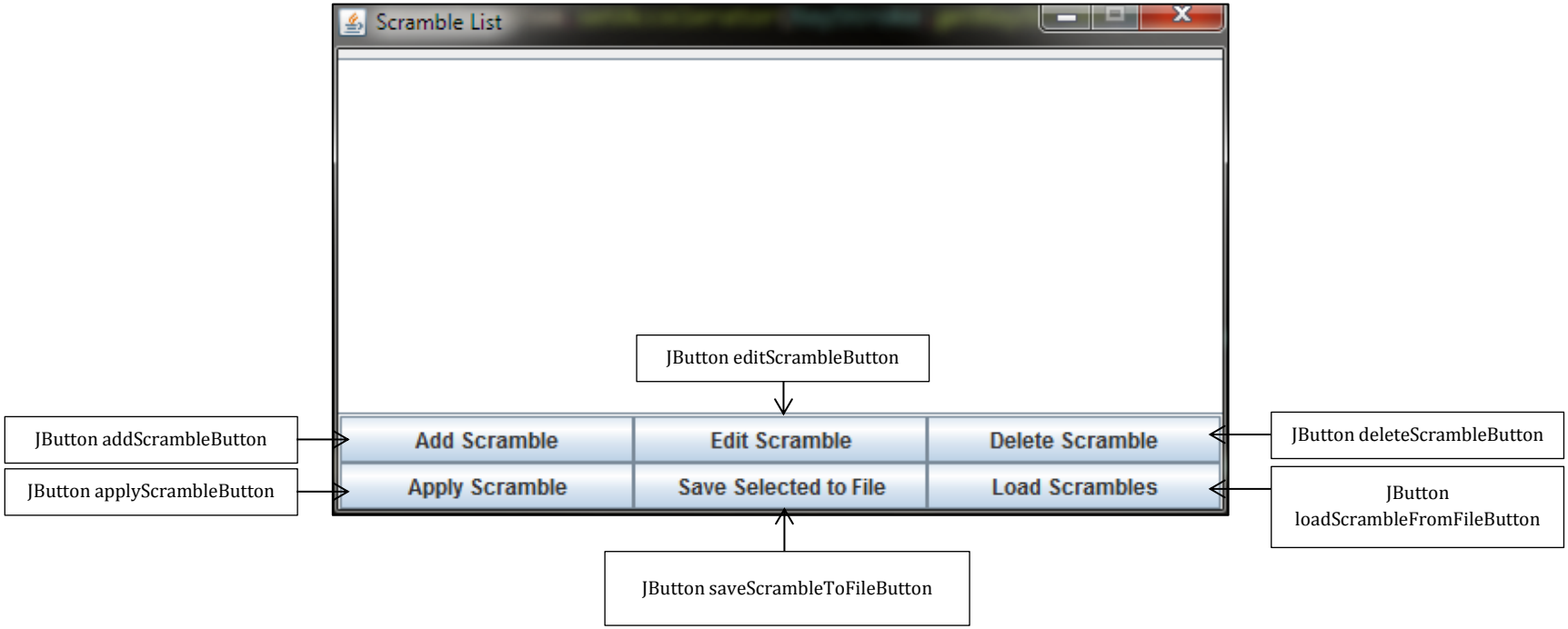
Menu Item	viewMenu → JRadioButtonMenuItem radioButton2D
Description	Selecting this radio button changes the graph so that it is rendered in 2D

```
public void actionPerformed(ActionEvent e) {
   dimension = 2;
   Main.refreshTimeGraph(false);
}
```

Menu Item	viewMenu → JRadioButtonMenuItem radioButton3D
Description	Selecting this radio button changes the graph so that it is rendered in 3D

```
public void actionPerformed(ActionEvent e) {
   dimension = 3;
   Main.refreshTimeGraph(false);
}
```

Scramble List



Button	JButton addScrambleButton
Description	Clicking this button opens an input dialog into which the user can enter a new scramble.

```
public void actionPerformed(ActionEvent e) {
   String scramble = JOptionPane.showInputDialog(null, "Enter Scramble", "R U R' U'");

   if (scramble != null && (!scramble.trim().equals(""))) {
      scrambleList.addElement(" " + scramble.trim());
      listHolder.ensureIndexIsVisible(scrambleList.getSize() - 1);
      listHolder.setSelectedIndex(scrambleList.getSize() - 1);
   }
}
```

Button	JButton editScrambleButton
Description	Clicking this button opens an input dialog displaying the existing scramble, which the user can edit.

```
public void actionPerformed(ActionEvent e) {
   int selectedIndex = listHolder.getSelectedIndex();

   if (selectedIndex == -1)
      return;
   else {
      String scramble = scrambleList.get(selectedIndex);
      scramble = JOptionPane.showInputDialog(null, "Enter New Scramble", scramble.trim());

      if (scramble != null && (!scramble.trim().equals(""))) {
         scrambleList.set(selectedIndex, " " + scramble.trim());
      }
   }
}
```

Button	JButton deleteScrambleButton
Description	Clicking this button deletes the selected scrambles from the list

```
public void actionPerformed(ActionEvent arg0) {
   int selectedIndex = listHolder.getSelectedIndex();

   if (selectedIndex == -1)
      return;
   else {
      int[] selectedIndices = listHolder.getSelectedIndices();
```

```
      for (int i = selectedIndices.length - 1; i >= 0; --i) {
         scrambleList.remove(selectedIndices[i]);
         listHolder.clearSelection();
      }

      if (scrambleList.size() > 0)
         listHolder.setSelectedIndex((selectedIndex < scrambleList.size()) ? selectedIndex
               : selectedIndex - 1);
   }
}
```

Button	JButton applyScrambleButton
Description	Clicking this button applies hte selected scramble to the cube in the main window and displays the scramble at the top of the main window

```
public void actionPerformed(ActionEvent e) {
   int selectedIndex = listHolder.getSelectedIndex();

   if (selectedIndex == -1)
      return;
   else
      Main.handleScramble(scrambleList.get(selectedIndex));
}
```

Button	JButton saveScrambleToFileButton
Description	Clicking this button opens a save dialog in which the user can choose a location to save the selected scrambles

```
public void actionPerformed(ActionEvent e) {
   int selectedIndex = listHolder.getSelectedIndex();

   if (selectedIndex == -1)
      return;
   else {
      int[] selectedIndices = listHolder.getSelectedIndices();

      if (fileChooser.showSaveDialog(null) == JFileChooser.APPROVE_OPTION) {
         TextFile currentFile = new TextFile();

         try {
```

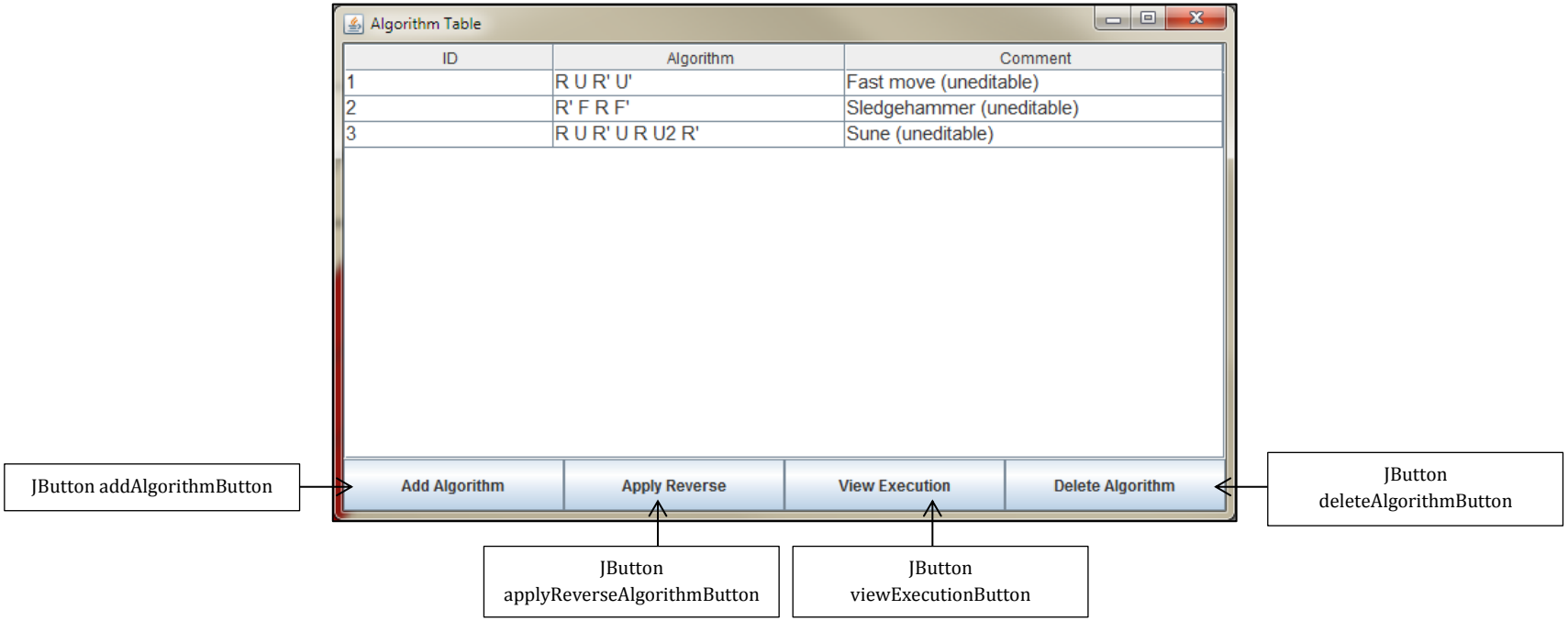
```
            currentFile.setFilePath(fileChooser.getSelectedFile().toString());
            currentFile.setIO(TextFile.WRITE);

            for (int i = 0; i < selectedIndices.length; ++i) {
               currentFile.writeLine(scrambleList.get(selectedIndices[i]).trim());
            }
         } catch (Exception exc) {
            JOptionPane.showMessageDialog(null, "Could not save to file", "Error",
                  JOptionPane.ERROR_MESSAGE);
         } finally {
            currentFile.close();
         }
      }
   }
}
```

Button	JButton loadScrambleFromFileButton
Description	Clicking this button opens a save dialog in which the user can choose a location to save the selected scramblesClicking this button opens a load dialog in which the user can select the text file which contains the scrambles they want to load

```
public void actionPerformed(ActionEvent e) {
   try {
      if (fileChooser.showOpenDialog(null) == JFileChooser.APPROVE_OPTION) {
         String current;
         int numLines;

         TextFile currentFile = new TextFile();
         currentFile.setFilePath(fileChooser.getSelectedFile().toString());
         currentFile.setIO(TextFile.READ);
         numLines = currentFile.getNumLines();

         for (int i = 0; i < numLines; ++i) {
            current = currentFile.readLine().trim();

            if (!current.equals("")) {
               scrambleList.addElement(" " + current);
            }
         }

         currentFile.close();
```

```
        }
    } catch (Exception exc) {
        JOptionPane.showMessageDialog(null, "Error Reading File", "Error", JOptionPane.ERROR_MESSAGE);
    }
}
```

Algorithm Table Window



Button	JButton addAlgorithmButton
Description	Clicking this button opens an input dialog so that a new algorithm can be entered

```
public void actionPerformed(ActionEvent e) {
   try {
      addRow();
   } catch (SQLException exc) {
      JOptionPane
            .showMessageDialog(null, "Could not access database", "Error", JOptionPane.ERROR_MESSAGE);
   } catch (ClassNotFoundException exc) {
      JOptionPane
            .showMessageDialog(null, "Could not access database", "Error", JOptionPane.ERROR_MESSAGE);
   }
}
```

Button	JButton applyReverseAlgorithmButton
Description	Clicking this button applies the reverse of the selected algorithm to the virtual cube in the main window

```
public void actionPerformed(ActionEvent e) {
   int selectedRow = algorithmTable.getSelectedRow();
   if (selectedRow != -1) {
      Main.handleScramble(SolveMaster.getReverseStringMoves("" + model.getValueAt(selectedRow, 1)));
   }
}
```

Button	JButton viewExecutionButton
Description	Clicking this button performs the algorithm in real-time on the virtual cube in the main window

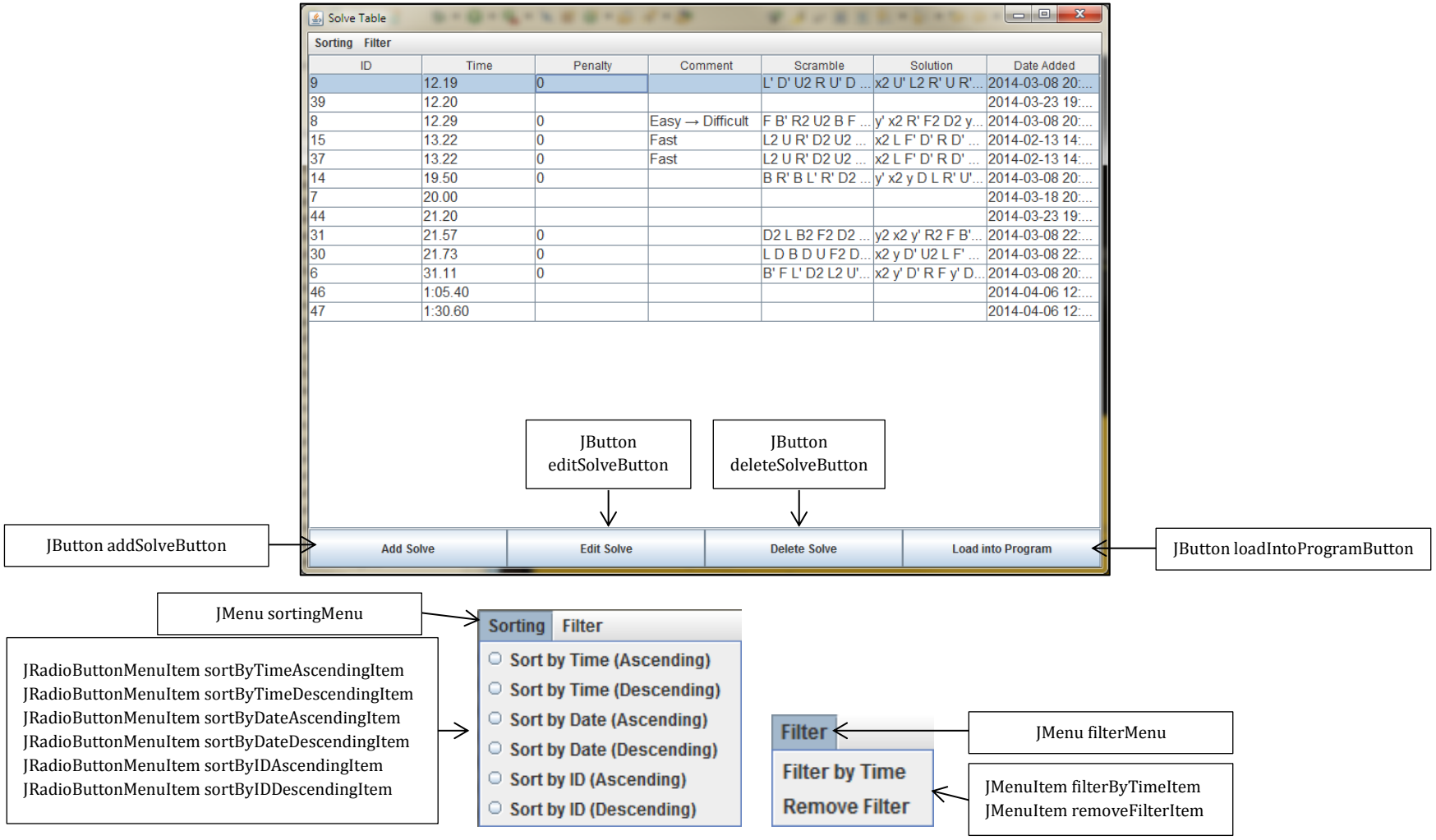
```
public void actionPerformed(ActionEvent e) {
   int selectedRow = algorithmTable.getSelectedRow();
   if (selectedRow != -1) {
      String algorithm = "" + model.getValueAt(selectedRow, 1);
      Main.performRealTimeSolving(SolveMaster.getReverseStringMoves(algorithm), algorithm);
   }
}
```

Button	JButton deleteAlgorithmButton
Description	Clicking this button deletes the selected rows from the table

```
public void actionPerformed(ActionEvent arg0) {
   try {
      Object[] options = { "Yes", "No" };
```

```
      int choice = -1;

      if (algorithmTable.getSelectedRows().length > 5) {
         choice = JOptionPane.showOptionDialog(null, "Are you sure you want to delete?", "Warning",
               JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[1]);
      } else
         choice = 0;

      if (choice == 0)
         deleteRow();
   } catch (ClassNotFoundException | SQLException e) {
      JOptionPane.showMessageDialog(null, "Unable to delete record from database", "Error",
            JOptionPane.ERROR_MESSAGE);
   }
}
```

Solve Table Window



Menu Item	sortingMenu → JRadioButtonMenuItem sortByTimeAscendingItem
Description	Selecting this radio button causes the rows in the table to be sorted by the time column in ascending order.

```
public void actionPerformed(ActionEvent arg0) {
   orderByAttribute = "solveTime";
   orderDirection = "ASC";
   populateCellsWithDatabaseData();
}
```

Menu Item	sortingMenu → JRadioButtonMenuItem sortByTimeDescendingItem
Description	Selecting this radio button causes the rows in the table to be sorted by the time column in descending order.

```
public void actionPerformed(ActionEvent arg0) {
   orderByAttribute = "solveTime";
   orderDirection = "DESC";
   populateCellsWithDatabaseData();
}
```

Menu Item	sortingMenu → JRadioButtonMenuItem sortByDateAscendingItem
Description	Selecting this radio button causes the rows in the table to be sorted by the date added column in ascending order.

```
public void actionPerformed(ActionEvent arg0) {
   orderByAttribute = "dateAdded";
   orderDirection = "ASC";
   populateCellsWithDatabaseData();
}
```

Menu Item	sortingMenu → JRadioButtonMenuItem sortByDateDescendingItem
Description	Selecting this radio button causes the rows in the table to be sorted by the date added column in descending order.

```
public void actionPerformed(ActionEvent arg0) {
   orderByAttribute = "dateAdded";
   orderDirection = "DESC";
   populateCellsWithDatabaseData();
}
```

Menu Item	sortingMenu → JRadioButtonMenuItem sortByIDAscendingItem
Description	Selecting this radio button causes the rows in the table to be sorted by the ID column in ascending order.

```
public void actionPerformed(ActionEvent arg0) {
   orderByAttribute = "ID";
   orderDirection = "ASC";
   populateCellsWithDatabaseData();
}
```

Menu Item	sortingMenu → JRadioButtonMenuItem sortByIDDescendingItem
Description	Selecting this radio button causes the rows in the table to be sorted by the ID column in descending order.

```
public void actionPerformed(ActionEvent arg0) {
   orderByAttribute = "ID";
   orderDirection = "DESC";
   populateCellsWithDatabaseData();
}
```

Menu Item	filterMenu → JMenuItem filterByTimeItem
Description	Clicking this menu item allows the user to filter the data in the table by certain criteria relating to the time column.

```
public void actionPerformed(ActionEvent arg0) {
   String originalLowerBoundary = lowerTimeBoundary;
   String originalUpperBoundary = upperTimeBoundary;
   lowerTimeBoundary = JOptionPane.showInputDialog(null, "Enter lower time boundary",
         "Lower time boundary");
   if (lowerTimeBoundary != null) {
      if (!Solve.isValidTime(lowerTimeBoundary)) {
         resetTimeBoundariesAndShowErrorMessage(originalLowerBoundary, originalUpperBoundary);
         return;
      }
      else
         lowerTimeBoundary = "" + Solve.getFormattedStringToDouble(lowerTimeBoundary);
   } else {
      lowerTimeBoundary = originalLowerBoundary;
      populateCellsWithDatabaseData();
      return;
   }

   upperTimeBoundary = JOptionPane.showInputDialog(null, "Enter upper time boundary",
         "Upper time boundary");
```

```
   if (upperTimeBoundary != null) {
      if (!Solve.isValidTime(upperTimeBoundary)) {
         resetTimeBoundariesAndShowErrorMessage(originalLowerBoundary, originalUpperBoundary);
         return;
      }
      else
         upperTimeBoundary = "" + Solve.getFormattedStringToDouble(upperTimeBoundary);
   } else {
      upperTimeBoundary = originalUpperBoundary;
      populateCellsWithDatabaseData();
      return;
   }

   if (Solve.getFormattedStringToDouble(lowerTimeBoundary) > Solve
         .getFormattedStringToDouble(upperTimeBoundary))
      resetTimeBoundariesAndShowErrorMessage(originalLowerBoundary, originalUpperBoundary);

   populateCellsWithDatabaseData();
}
```

Menu Item	filterMenu → JMenuItem removeFilterItem
Description	Clicking this menu item removes any filter present so that all data from the table is shown

```
public void actionPerformed(ActionEvent arg0) {
   lowerTimeBoundary = "-1";
   upperTimeBoundary = "10000000000";
   populateCellsWithDatabaseData();
}
```

Button	JButton addSolveButton
Description	Clicking this button opens the Solve Form window.

```
public void actionPerformed(ActionEvent e) {
   timeField.setText("");
   penaltyField.setText("");
   commentField.setText("");
   scrambleField.setText("");
   solutionField.setText("");
   dateAddedField.setText("");
   editing = false;
```

```
   solveInputForm.setVisible(true);
   menuBar.clearButtonGroupSelection();
   menuBar.setEnabledButtonGroup(false);
}
```

Button	JButton editSolveButton
Description	Clicking this button opens the Solve Form window displaying the existing data for the selected solve.

```
public void actionPerformed(ActionEvent arg0) {
   int selectedRow = solveTable.getSelectedRow();

   if (selectedRow != -1) {
      timeField.setText("" + solveTable.getValueAt(selectedRow, 1));
      penaltyField.setText("" + solveTable.getValueAt(selectedRow, 2));
      commentField.setText("" + solveTable.getValueAt(selectedRow, 3));
      scrambleField.setText("" + solveTable.getValueAt(selectedRow, 4));
      solutionField.setText("" + solveTable.getValueAt(selectedRow, 5));
      dateAddedField.setText("" + solveTable.getValueAt(selectedRow, 6));
      menuBar.clearButtonGroupSelection();
      menuBar.setEnabledButtonGroup(false);
      solveInputForm.setVisible(true);
      selectedRowIndex = selectedRow;
      editing = true;
   }
}
```

Button	JButton deleteSolveButton
Description	Clicking this button deletes the selected rows from the table.

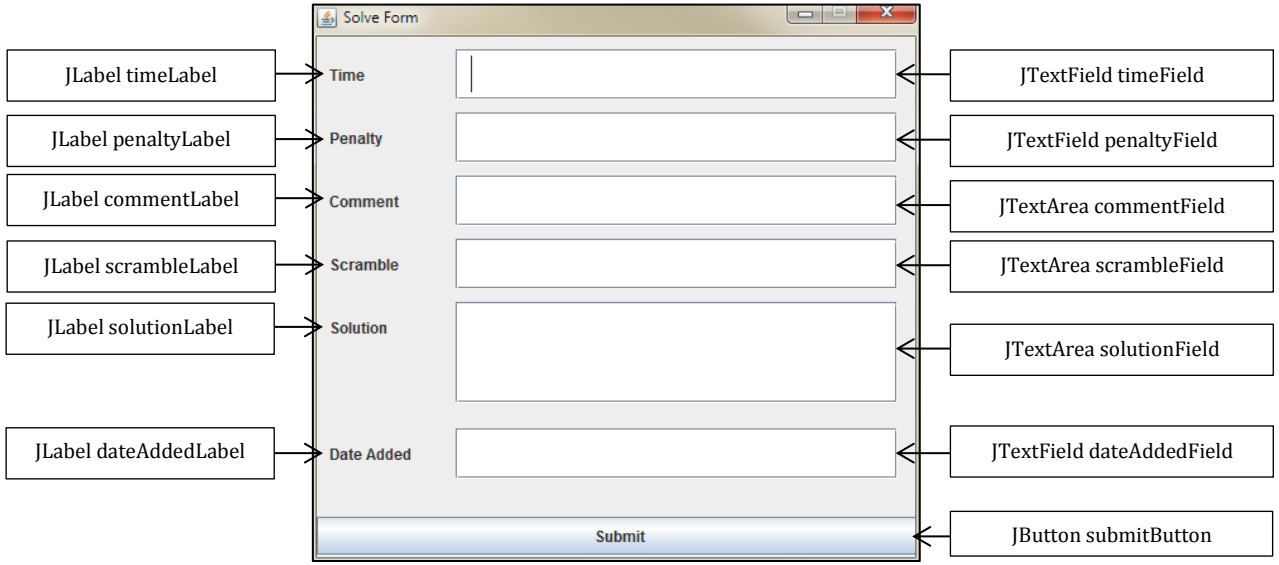
```
public void actionPerformed(ActionEvent arg0) {
   try {
      Object[] options = { "Yes", "No" };
      int choice = -1;

      if (solveTable.getSelectedRows().length >= 5) {
         choice = JOptionPane.showOptionDialog(null, "Are you sure you want to delete?", "Warning",
               JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[1]);
      } else
         choice = 0;
```

```
      if (choice == 0)
         deleteRow();
   } catch (ClassNotFoundException | SQLException e) {
      JOptionPane.showMessageDialog(null, "Unable to delete record from database", "Error",
            JOptionPane.ERROR_MESSAGE);
   } finally {
   }
}
```

Button	JButton loadIntoProgramButton
Description	Clicking this button loads the selected solves into the main window.

```
public void actionPerformed(ActionEvent e) {
   int[] selectedIndices = solveTable.getSelectedRows();
   String test;

   for (int i = 0; i < selectedIndices.length; ++i) {
      test = "" + model.getValueAt(selectedIndices[i], 2);
      Main.addSolveToList(new Solve("" + model.getValueAt(selectedIndices[i], 1),
            test.trim().equals("") ? "0" : test, "" + model.getValueAt(selectedIndices[i], 3), ""
                  + model.getValueAt(selectedIndices[i], 4), ""
                  + model.getValueAt(selectedIndices[i], 5)));
   }
}
```

Solve Form Window

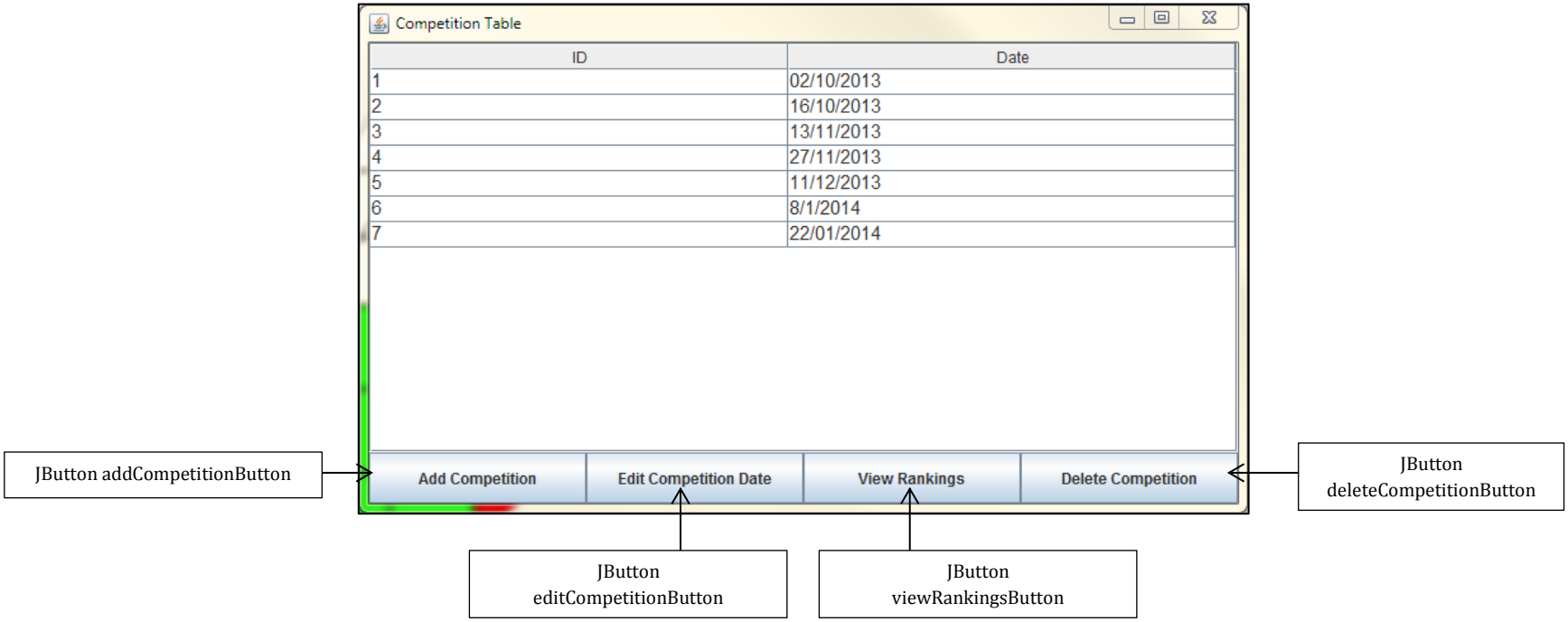


Button	JButton submitButton
Description	Clicking this button submits the data in the Solve Form window for validation.

```
public void actionPerformed(ActionEvent arg0) {
   String time = timeField.getText().trim();

   if (!Solve.isValidTime(time)) {
      JOptionPane.showMessageDialog(null, timeHelpMessage, "Error", JOptionPane.ERROR_MESSAGE, null);
      return;
   } else if ((dateAddedField.getText().trim().equals(""))
         || (!Solve.isValidDate(dateAddedField.getText().trim()))) {
      int choice = -1;
      Object[] options = { "Yes", "No" };
      choice = JOptionPane
            .showOptionDialog(
                  null,
                  "Date must be valid and have the form: yyyy-MM-dd HH:mm:ss\nWould you like to use the current
time?",
                  "Error", JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options,
                  options[0]);

      if (choice != 0)
         return;
      else {
         dateAddedField.setText(SolveDatabaseConnection.getCurrentTimeInSQLFormat());
      }
   }

   try {
      int row;
      if (!editing) {
         addRow();
         row = solveTable.getRowCount() - 1;
      } else {
         row = selectedRowIndex;
      }

      if (time.trim().equalsIgnoreCase("DNF")) {
         solveTable.setValueAt("DNF", row, 1);
         solveTable.setValueAt("0", row, 2);
```

```
      } else {
         solveTable.setValueAt(Solve.getPaddedTime(Solve.getSecondsToFormattedString(Solve
               .getFormattedStringToDouble(time))), row, 1);
         solveTable.setValueAt(penaltyField.getText().trim(), row, 2);
      }

      solveTable.setValueAt(commentField.getText().trim(), row, 3);
      solveTable.setValueAt(scrambleField.getText().trim(), row, 4);
      solveTable.setValueAt(solutionField.getText().trim(), row, 5);
      solveTable.setValueAt(dateAddedField.getText().trim(), row, 6);
      solveInputForm.setVisible(false);
      menuBar.setEnabledButtonGroup(true);
      editing = false;

      menuBar.clearButtonGroupSelection();

      SolveDatabaseConnection.executeUpdate(String.format("UPDATE solve " + "SET solveTime = \"%s\", "
            + "penalty = \"%s\", " + "comment = \"%s\", " + "scramble = \"%s\", "
            + "solution = \"%s\", " + "dateAdded = \"%s\" " + "WHERE solveID = %d",
            "" + model.getValueAt(row, 1), "" + model.getValueAt(row, 2),
            "" + model.getValueAt(row, 3), "" + model.getValueAt(row, 4),
            "" + model.getValueAt(row, 5), "" + model.getValueAt(row, 6),
            Integer.valueOf("" + model.getValueAt(row, 0))));
      resizeColumnWidths(solveTable);
   } catch (ClassNotFoundException e) {
      e.printStackTrace();
   } catch (SQLException e) {
      e.printStackTrace();
   }
}
```

Competition Table Window



Button	JButton addCompetitionButton
Description	Clicking this button adds a row to the table

```
public void actionPerformed(ActionEvent e) {
   try {
      addRow();
   } catch (SQLException exc) {
      exc.printStackTrace();
      JOptionPane
            .showMessageDialog(null, "Could not access database", "Error", JOptionPane.ERROR_MESSAGE);
   } catch (ClassNotFoundException exc) {
      JOptionPane
            .showMessageDialog(null, "Could not access database", "Error", JOptionPane.ERROR_MESSAGE);
   }
}
```

Button	JButton editCompetitionButton
Description	Clicking this button opens an input dialog so that the date of the competition can be edited

```
public void actionPerformed(ActionEvent e) {
   // Stores the string entered by the user.
   String date;
   // Stores the index of the row in the table selected by the user.
   int row = competitionTable.getSelectedRow();

   if (row != -1) {
      // Stores the date-string shown in the selected row so that if
      // editing, the existing date can be shown in the input window.
      String originalDate = "" + competitionTable.getValueAt(row, 1);
      date = JOptionPane.showInputDialog(null, "Enter Date of Competition",
            (originalDate.equals("")) ? "dd/MM/YYYY" : originalDate);

      if (date == null)
         return;

      date += "";
      date = date.trim();

      if ((date.equals("")) || (!Competition.isValidDate(date))) {
         JOptionPane.showMessageDialog(null, "Invalid date", "Error", JOptionPane.ERROR_MESSAGE);
      } else {
```

```
         try {
            CompetitionDatabaseConnection.executeUpdate(String.format("UPDATE competition "
                  + "SET competitionDate = \"%s\" " + "WHERE competitionID = %d", date,
                  Integer.valueOf("" + model.getValueAt(competitionTable.getSelectedRow(), 0))));
            competitionTable.setValueAt(date, row, 1);
         } catch (Exception exc) {
            exc.printStackTrace();
         }

      }
   }
}
```

Button	JButton viewRankingsButton
Description	Clicking this button opens the 'Member-Competition Table' window and displays the rankings for the selected competition

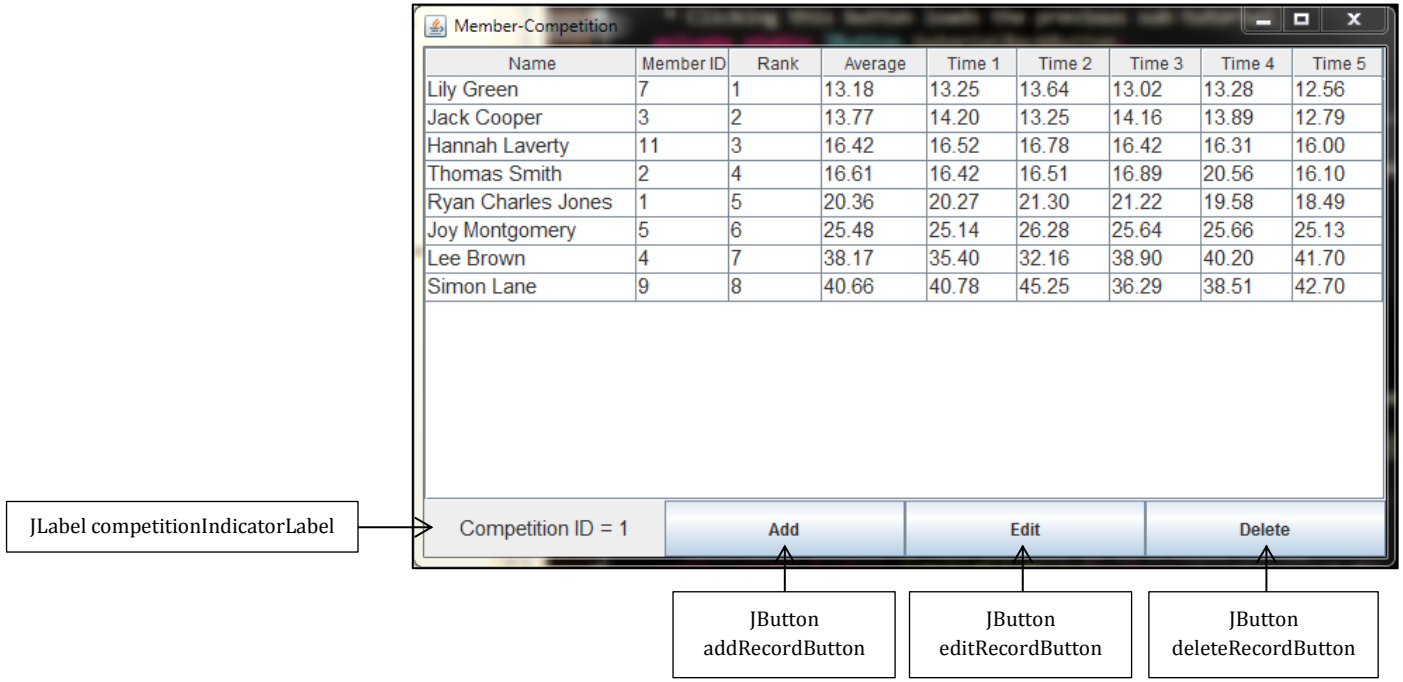
```
public void actionPerformed(ActionEvent arg0) {
   if ((competitionTable.getSelectedRow() > -1)) {
      memberCompetitionDatabasePopUp.setCurrentCompetitionID(Integer.valueOf(""
            + competitionTable.getValueAt(competitionTable.getSelectedRow(), 0)));
      memberCompetitionDatabasePopUp.populateCellsWithDatabaseData();
      memberCompetitionDatabasePopUp.setVisible(true);
   }
}
```

Button	JButton deleteCompetitionButton
Description	Clicking this button deletes the selected competitions from the table and from the database

```
public void actionPerformed(ActionEvent arg0) {
   try {
      Object[] options = { "Yes", "No" };
      int choice = -1;

      if (competitionTable.getSelectedRows().length == 0)
         return;

      choice = JOptionPane.showOptionDialog(null, "Are you sure you want to delete?", "Warning",
            JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[1]);

      if (choice == 0)
```

```
            deleteRow();
      } catch (ClassNotFoundException | SQLException e) {
         JOptionPane.showMessageDialog(null, "Unable to delete record from database", "Error",
               JOptionPane.ERROR_MESSAGE);
      }
}
```

Member-Competition Window



Button	JButton addRecordButton
Description	Clicking this button opens the Member-Competition Form window.

```
public void actionPerformed(ActionEvent e) {
   memberCompetitionTable.clearSelection();

   editing = false;
   resetMemberIDComboBoxItems();
   if (memberIDComboBox.getItemCount() == 0) {
      memberCompetitionInputForm.setVisible(false);

      JOptionPane.showMessageDialog(null, "No existing members remaining", "Error",
            JOptionPane.ERROR_MESSAGE);
   } else {
      time1Field.setText("");
      time2Field.setText("");
      time3Field.setText("");
      time4Field.setText("");
      time5Field.setText("");
      memberCompetitionInputForm.setVisible(true);
   }
}
```

Button	JButton editRecordButton
Description	Clicking this button opens the Member-Competition Form window if a row has been selected.

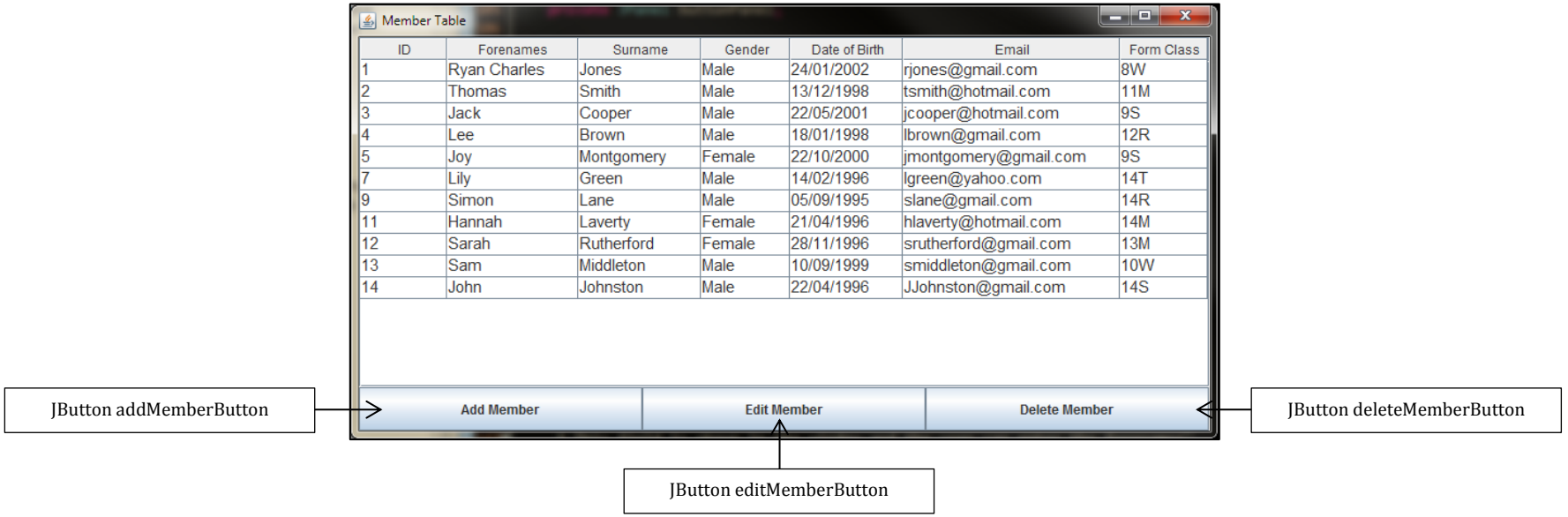
```
public void actionPerformed(ActionEvent e) {
   int selectedRow = memberCompetitionTable.getSelectedRow();

   if (selectedRow != -1) {
      editing = true;
      time1Field.setText("" + memberCompetitionTable.getValueAt(selectedRow, 4));
      time2Field.setText("" + memberCompetitionTable.getValueAt(selectedRow, 5));
      time3Field.setText("" + memberCompetitionTable.getValueAt(selectedRow, 6));
      time4Field.setText("" + memberCompetitionTable.getValueAt(selectedRow, 7));
      time5Field.setText("" + memberCompetitionTable.getValueAt(selectedRow, 8));
      resetMemberIDComboBoxItems();
      if (!memberIDComboBoxContains(Integer.valueOf("" + memberCompetitionTable.getValueAt(selectedRow, 1)))) {
         JOptionPane.showMessageDialog(null, "This member no longer exists", "Error", JOptionPane.ERROR_MESSAGE);
         editing = false;
         return;
```

```
      }

      memberIDComboBox.setSelectedItem(Integer.valueOf("" + memberCompetitionTable.getValueAt(selectedRow, 1)));
      selectedMCIndex = selectedRow;
      memberCompetitionInputForm.setVisible(true);
   }
}
```

Button	JButton deleteRecordButton
Description	Clicking this button deletes the selected rows from the table.

```
public void actionPerformed(ActionEvent arg0) {
   try {
      Object[] options = { "Yes", "No" };
      int choice = -1;

      int selectedIndex = memberCompetitionTable.getSelectedRow();

      if (selectedIndex == -1)
         return;

      choice = JOptionPane.showOptionDialog(null, "Are you sure you want to delete?", "Warning",
            JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[1]);

      if (choice == 0)
         deleteRow();
   } catch (ClassNotFoundException | SQLException e) {
      JOptionPane.showMessageDialog(null, "Unable to delete record from database", "Error",
            JOptionPane.ERROR_MESSAGE);
   }
}
```

Member Table Window



Button	JButton addMemberButton
Description	Clicking this button opens the 'Member Form' window.

```
public void actionPerformed(ActionEvent e) {
   memberTable.clearSelection();
   editing = false;
   forenamesField.setText("");
   surnameField.setText("");
   dateOfBirthField.setText("");
   emailField.setText("");
   formClassField.setText("");
   memberInputForm.setVisible(true);
}
```

Button	JButton editMemberButton
Description	Clicking this button opens the 'Member Form' window if a row has been selected.

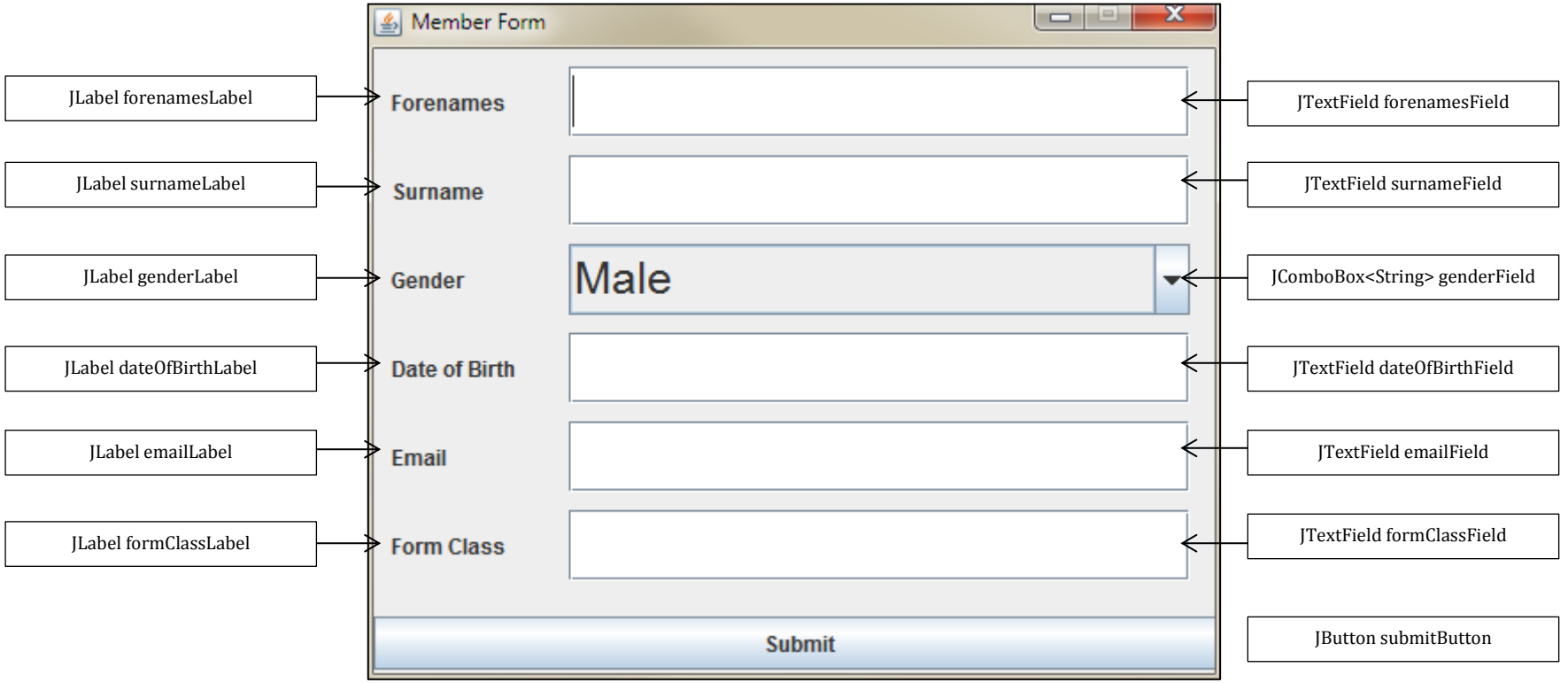
```
public void actionPerformed(ActionEvent e) {
   int selectedRow = memberTable.getSelectedRow();

   if (selectedRow != -1) {
      forenamesField.setText("" + memberTable.getValueAt(selectedRow, 1));
      surnameField.setText("" + memberTable.getValueAt(selectedRow, 2));
      genderField.setSelectedItem("" + memberTable.getValueAt(selectedRow, 3));
      dateOfBirthField.setText("" + memberTable.getValueAt(selectedRow, 4));
      emailField.setText("" + memberTable.getValueAt(selectedRow, 5));
      formClassField.setText("" + memberTable.getValueAt(selectedRow, 6));
      memberInputForm.setVisible(true);
      editing = true;
      selectedRowIndex = selectedRow;
   }
}
```

Button	JButton deleteMemberButton
Description	Clicking this button deletes the selected rows from the table.

```
public void actionPerformed(ActionEvent arg0) {
   try {
      Object[] options = { "Yes", "No" };
      int choice = -1;
```

```
      int selectedIndex = memberTable.getSelectedRow();

      if (selectedIndex == -1)
         return;

      choice = JOptionPane.showOptionDialog(null, "Are you sure you want to delete?", "Warning",
            JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE, null, options, options[1]);

      if (choice == 0)
         deleteRow();
   } catch (ClassNotFoundException | SQLException e) {
      JOptionPane.showMessageDialog(null, "Unable to delete record from database", "Error",
            JOptionPane.ERROR_MESSAGE);
   }
}
```

Member Form Window

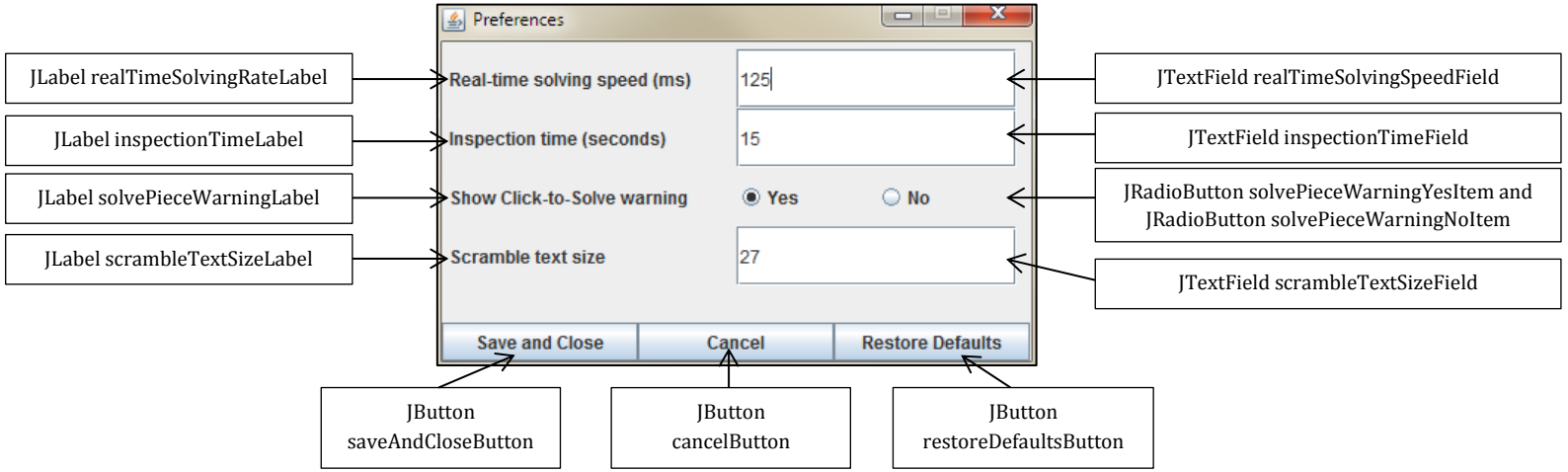


Button	JButton submitButton
Description	Clicking this button submits the data in the 'Member Form' window for validation.

```
public void actionPerformed(ActionEvent arg0) {
   int row, memberID;
   String dateOfBirth = dateOfBirthField.getText().trim();
   String email = emailField.getText().trim();
   String formClass = formClassField.getText().trim();
   String forenames = forenamesField.getText().trim();
   String surname = surnameField.getText().trim();
   String gender = (genderField.getSelectedItem() + "").trim();

   if ((dateOfBirth.equals("")) || (!isValidDate(dateOfBirth))) {
      JOptionPane.showMessageDialog(null, "Invalid date of birth", "Error", JOptionPane.ERROR_MESSAGE);
      return;
   } else if ((email.equals("")) || (!Member.isValidEmail(email))) {
      JOptionPane.showMessageDialog(null, "Invalid email", "Error", JOptionPane.ERROR_MESSAGE);
      return;
   } else if ((formClass.equals("")) || (!Member.isValidFormClass(formClass))) {
      JOptionPane.showMessageDialog(null, "Invalid form class", "Error", JOptionPane.ERROR_MESSAGE);
      return;
   } else if ((forenames.equals(""))) {
      JOptionPane.showMessageDialog(null, "Empty forenames", "Error", JOptionPane.ERROR_MESSAGE);
      return;
   } else if ((surname.equals(""))) {
      JOptionPane.showMessageDialog(null, "Empty surname", "Error", JOptionPane.ERROR_MESSAGE);
      return;
   }

   try {
      if (!editing) {
         addRow();
         row = memberTable.getRowCount() - 1;
      } else {
         // row = memberTable.getSelectedRow();
         row = selectedRowIndex;
      }
      memberID = Integer.valueOf("" + memberTable.getValueAt(row, 0));

      model.setValueAt(forenames + "", row, 1);
```

```
      model.setValueAt(surname + "", row, 2);
      model.setValueAt(gender + "", row, 3);
      model.setValueAt(dateOfBirth + "", row, 4);
      model.setValueAt(email + "", row, 5);
      model.setValueAt(formClass + "", row, 6);

      memberInputForm.setVisible(false);
      forenamesField.requestFocus();
      editing = false;

      MemberDatabaseConnection.executeUpdate(String.format("UPDATE member " + "SET forenames = \"%s\", "
            + "surname = \"%s\", " + "gender = \"%s\", " + "dateOfBirth = \"%s\", "
            + "email = \"%s\", " + "formClass = \"%s\" " + "WHERE memberID = %d", "" + forenames, ""
            + surname, "" + gender, "" + dateOfBirth, "" + email, "" + formClass, memberID));
      Main.resizeColumnWidths(memberTable);
   } catch (ClassNotFoundException | SQLException e) {
      e.printStackTrace();
   }
}
```

Preferences Window



Button	JButton saveAndCloseButton
Description	Clicking this button submits the data in the window for validation. If the data is valid, then the window Preferences window will close.

```
public void actionPerformed(ActionEvent e) {
   boolean valid = true;

   try {
      realTimeSolvingSpeed = (int) Double.parseDouble(realTimeSolvingSpeedField.getText().trim());

      if ((realTimeSolvingSpeed <= 0) || (realTimeSolvingSpeed >= 10000)) {
         throw new Exception();
      }

      realTimeSolvingSpeed = Math.abs(realTimeSolvingSpeed);
      realTimeSolvingSpeedField.setBackground(Color.white);
   } catch (Exception exc) {
      realTimeSolvingSpeedField.setBackground(new Color(255, 150, 150));
      valid = false;
   }

   try {
      inspectionTime = (int) Double.parseDouble(inspectionTimeField.getText().trim());

      if ((inspectionTime <= 0) || (inspectionTime >= 100)) {
         throw new Exception();
      }

      inspectionTimeField.setBackground(Color.white);
   } catch (Exception exc) {
      inspectionTimeField.setBackground(new Color(255, 150, 150));
      valid = false;
   }

   try {
      scrambleTextSize = (int) Double.parseDouble(scrambleTextSizeField.getText().trim());

      if ((scrambleTextSize <= 0) || (scrambleTextSize >= 100))
         throw new Exception();

      scrambleTextSizeField.setBackground(Color.white);
```

```
    } catch (Exception exc) {
       scrambleTextSizeField.setBackground(new Color(255, 150, 150));
       valid = false;
    }

    if (valid) {
       setVisible(false);
       savePreferences();
       populateFieldsWithValues();
       Main.setScrambleTextSize(scrambleTextSize);
    }
}
```

Button	JButton cancelButton
Description	Clicking this button discards any unsaved changes and closes the window.

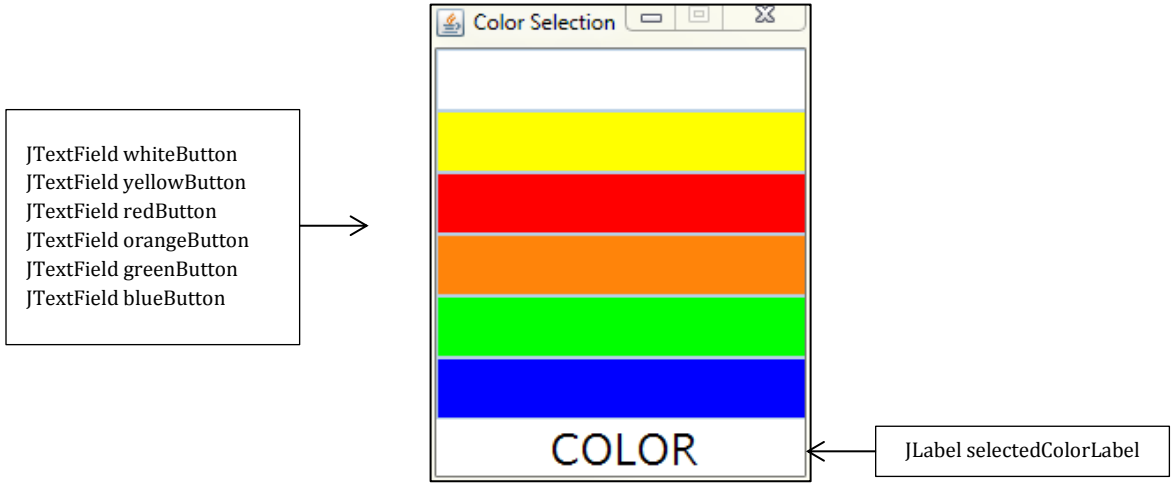
```
public void actionPerformed(ActionEvent arg0) {
   setVisible(false);
   populateFieldsWithValues();
}
```

Button	JButton restoreDefaultsButton
Description	Clicking this button restores the default preferences and updates the fields accordingly.

```
   solvePieceWarningYesItem.setSelected(true);
   solvePieceWarningNoItem.setSelected(false);

   realTimeSolvingSpeedField.setText(String.format("%d", REAL_TIME_SOLVING_SPEED));
   realTimeSolvingSpeed = REAL_TIME_SOLVING_SPEED;
   realTimeSolvingSpeedField.setBackground(Color.white);

   inspectionTimeField.setText(String.format("%d", INSPECTION_TIME));
   inspectionTime = INSPECTION_TIME;
   inspectionTimeField.setBackground(Color.white);

   scrambleTextSize = SCRAMBLE_TEXT_SIZE;
   scrambleTextSizeField.setText("" + SCRAMBLE_TEXT_SIZE);
   scrambleTextSizeField.setBackground(Color.white);

   solvePieceWarningEnabled();
```

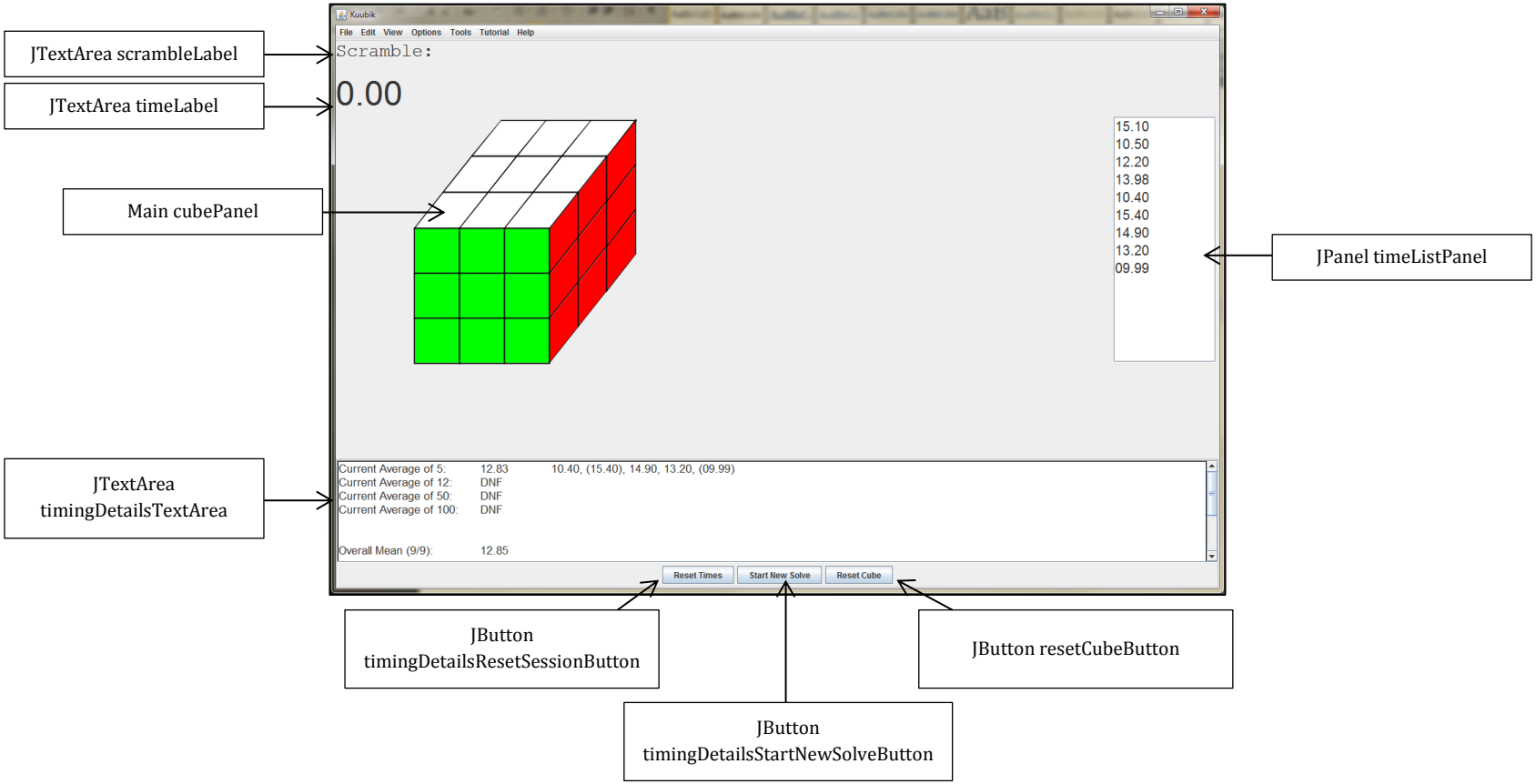
```
    Main.setScrambleTextSize(scrambleTextSize);
    savePreferences();
```

Color Selection Window



```
public void mouseClicked(MouseEvent arg0) {
   Object source = arg0.getSource();

   selectedColorLabel.setForeground(Color.BLACK);

   if (source == whiteButton)
      selectedColor = (Color.white);
   else if (source == yellowButton)
      selectedColor = (Color.yellow);
   else if (source == redButton)
      selectedColor = (Color.red);
   else if (source == orangeButton)
      selectedColor = (Cubie.orange);
   else if (source == greenButton)
      selectedColor = (Color.green);
   else if (source == blueButton) {
      selectedColorLabel.setForeground(Color.white);
      selectedColor = (Color.blue);
   }

   selectedColorLabel.setBackground(selectedColor.brighter());
   Main.requestCubePanelFocus();
}
```

Main Window (Timing Mode)



Button	JButton timingDetailsResetSessionButton
Description	Clicking this button clears all times in the list at the right-hand side of the main window

```
public void actionPerformed(ActionEvent e) {
   solves.clear();
   copyAllTimesToDisplay();
   refreshTimeGraph(true);
   refreshStatistics();
}
```

Button	JButton timingDetailsStartNewSolveButton
Description	Clicking this button starts a new solve

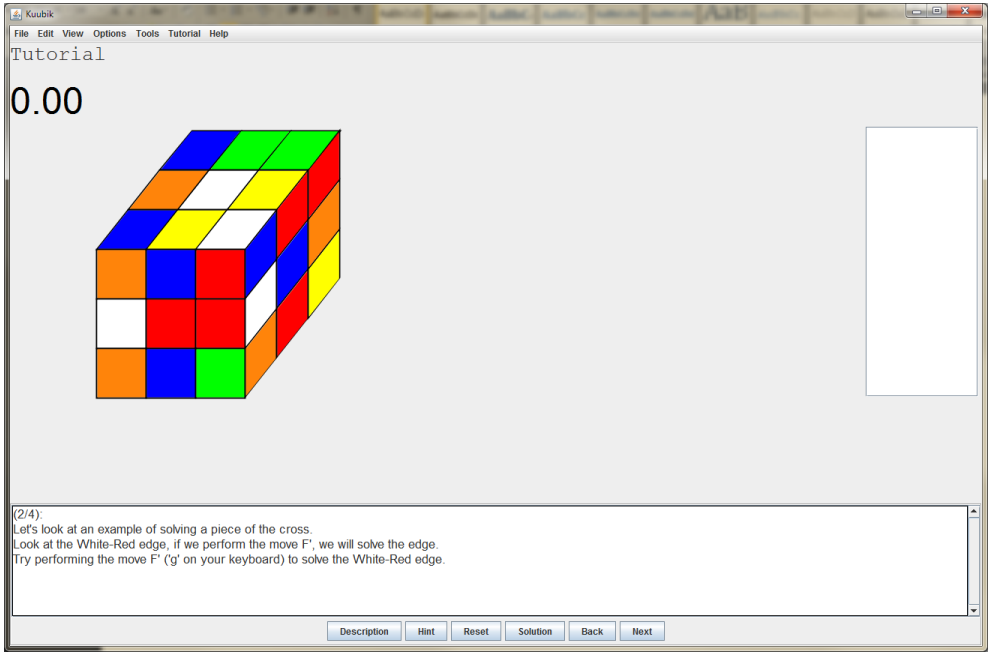
```
public void actionPerformed(ActionEvent e) {
   if (((customPaintingInProgress) || (tutorialIsRunning))
         || ((inspectionTimer != null) && (inspectionTimer.isRunning()))
         || ((incTimer != null) && (incTimer.isRunning()))
         || ((realTimeSolutionTimer != null) && (realTimeSolutionTimer.isRunning()))) {
      cubePanel.requestFocus();
      return;
   }

   if (!isValidCubeState(false)) {
      int choice = MouseSelectionSolver
            .getQuestionDialogResponse("The cube is not in a valid state, do you want to reset it?");

      if (choice == 0)
         resetCube();
      else
         return;
   }

   if (MenuBar.isUsingScramblesInList()) {
      try {
         currentScramble = scramblePopUp.getCurrentScramble();
      } catch (IndexOutOfBoundsException scrambleListIsEmpty) {
         JOptionPane.showMessageDialog(null, "No scrambles in scramble list", "Error", JOptionPane.ERROR_MESSAGE);
         return;
      }
   } else {
      currentScramble = Scramble.generateScramble();
```

```
    }

    movesAllowed = false;
    timerHasPermissionToStart = true;
    movesToBeRecorded = true;
    cubeSolved = false;

    inspectionTimeRemaining = preferencesPopUp.getInspectionTime();
    inspectionTimer = new Timer(1000, new InspectionTimerListener());
    MenuBar.setSolvePieceSelected(false);
    inspectionTimer.start();
    currentPenalty = "0";

    timeLabel.setForeground(Color.red);
    timeLabel.setText("" + inspectionTimeRemaining);

    scrambleLabel.setText("Scramble: " + currentScramble);
    resetCube();
    solveMaster.rotateToTopFront(Color.white, Color.green);
    solveMaster.clearMoves();
    trackingMoves.clear();

    cube.performAbsoluteMoves(currentScramble);
    cubePanel.repaint();
    cubePanel.requestFocus();
}
```

Button	JButton resetCubeButton
Description	Clicking this button resets the cube to the solve state white on top and green on front

```
public void actionPerformed(ActionEvent e) {
    cancelSolve();
    resetCube();
    refreshStatistics();
    cubePanel.repaint();
    cubePanel.requestFocus();
}
```

Main Window (Tutorial Mode)



JButton tutorialShowDescriptionButton
JButton tutorialHintButton
JButton tutorialResetStateButton
JButton tutorialShowSolutionButton
JButton tutorialBackButton
JButton tutorialNextButton

Button	JButton tutorialShowDescriptionButton
Description	Clicking this button shows the description for the current sub-tutorial

```
public void actionPerformed(ActionEvent arg0) {
   if (tutorial.isLoaded()) {
      tutorialTextArea.setText(tutorial.getDescription());
      tutorialTextArea.setCaretPosition(0);
      cubePanel.requestFocus();
   }
}
```

Button	JButton tutorialHintButton
Description	Clicking this button loads the next hint in the sub-tutorial

```
public void actionPerformed(ActionEvent arg0) {
   if (tutorial.isLoaded() && (tutorial.requiresUserAction())) {
      tutorial.loadNextHint();
      tutorialTextArea.setText(tutorial.getHint());
      tutorialTextArea.setCaretPosition(0);
      cubePanel.requestFocus();
   }
}
```

Button	JButton tutorialResetStateButton
Description	Clicking this button resets the cube to the solved state with white on top and green on front

```
public void actionPerformed(ActionEvent arg0) {
   if (tutorial.isLoaded() && (tutorial.requiresUserAction())) {
      resetCube();
      solveMaster.rotateToTopFront(Color.white, Color.green);
      solveMaster.clearMoves();

      cube.performAbsoluteMoves(tutorial.getScramble());
      cubePanel.requestFocus();
      cubePanel.repaint();
   }
}
```

Button	JButton tutorialShowSolutionButton
Description	Clicking this button shows the solution for the current sub-tutorial

```
public void actionPerformed(ActionEvent arg0) {
```

```
   if (tutorial.isLoaded() && (tutorial.requiresUserAction())) {
      resetCube();
      solveMaster.rotateToTopFront(Color.white, Color.green);
      solveMaster.clearMoves();

      movesAllowed = false;

      cube.performAbsoluteMoves(tutorial.getScramble());
      cube.performAbsoluteMoves(tutorial.getOptimalSolutions()[0]);

      String[] optimalSolutions = tutorial.getOptimalSolutions();
      tutorialTextArea.setText("Solution: " + tutorial.getExplanation()
            + "\n\nOptimal solutions include:");

      for (int i = 0; i < optimalSolutions.length; ++i)
         tutorialTextArea.setText(tutorialTextArea.getText() + "\n" + optimalSolutions[i]);

      tutorialTextArea.setCaretPosition(0);

      cubePanel.repaint();
      cubePanel.requestFocus();
   }
}
```

Button	JButton tutorialBackButton
Description	Clicking this button loads the previous sub-tutorial

```
public void actionPerformed(ActionEvent arg0) {
   if (tutorial.isLoaded()) {
      resetCube();
      solveMaster.rotateToTopFront(Color.white, Color.green);
      solveMaster.clearMoves();
      trackingMoves.clear();

      tutorial.loadPreviousSubTutorial();
      tutorialTextArea.setText(tutorial.getDescription());
      tutorialTextArea.setCaretPosition(0);
      cube.performAbsoluteMoves(tutorial.getScramble());
      cubePanel.repaint();
      cubePanel.requestFocus();
```

```
      movesAllowed = tutorial.requiresUserAction();
      tutorialBackButton.setEnabled(!tutorial.isFirstSubTutorial());
      tutorialNextButton.setEnabled(!tutorial.isLastSubTutorial());
      tutorialHintButton.setEnabled(tutorial.requiresUserAction());
      tutorialShowSolutionButton.setEnabled(tutorial.requiresUserAction());
   }
}
```

Button	JButton tutorialNextButton
Description	Clicking this button loads the next sub-tutorial

```
public void actionPerformed(ActionEvent arg0) {
   if (tutorial.isLoaded()) {
      resetCube();
      solveMaster.rotateToTopFront(Color.white, Color.green);
      solveMaster.clearMoves();

      trackingMoves.clear();

      tutorial.loadNextSubTutorial();
      tutorialTextArea.setText(tutorial.getDescription());
      tutorialTextArea.setCaretPosition(0);
      cube.performAbsoluteMoves(tutorial.getScramble());
      cubePanel.repaint();
      cubePanel.requestFocus();
      movesAllowed = tutorial.requiresUserAction();
      movesToBeRecorded = movesAllowed;
      tutorialBackButton.setEnabled(!tutorial.isFirstSubTutorial());
      tutorialNextButton.setEnabled(!tutorial.isLastSubTutorial());
      tutorialHintButton.setEnabled(tutorial.requiresUserAction());
      tutorialShowSolutionButton.setEnabled(tutorial.requiresUserAction());
   }
}
```

Acknowledgement of Non-User Code

I obtained the following algorithm from stackoverflow.com:

Resizes the widths of the columns in the specified table so that there is maximum visibility in each column

Parameters:

`table` - the table whose columns need resized

```
private void resizeColumnWidths(JTable table) {
   final TableColumnModel columnModel = table.getColumnModel();
   for (int column = 0; column < table.getColumnCount(); column++) {
      int width = 50; // Min width
      for (int row = 0; row < table.getRowCount(); row++) {
         TableCellRenderer renderer = table.getCellRenderer(row, column);
         Component comp = table.prepareRenderer(renderer, row, column);
         width = Math.max(comp.getPreferredSize().width, width);
      }
      columnModel.getColumn(column).setPreferredWidth(width);
   }
}
```

I obtained the scramble-generator (see class *Scramble* on page 311 of the appendix section) from speedsolving.com.

Libraries Used:

- SQLite-JDBC (sqlite.org): this is used to access the database.
- iTextPDF (itextpdf.com): this is used to generate PDF files.
- JFreeChart (jfree.org): this is used to display graphs.
- JCommon (jfree.org): this is used to display graphs.

Visual Design

All visual aspects of the system were written manually in code, not generated. Quite a few components use relative positioning so that when other elements are added, the code does not need to be changed for the others. For example:

```
…
buttonPanel.setLayout(new GridLayout(2, 3));
buttonPanel.add(addScrambleButton);
buttonPanel.add(editScrambleButton);
buttonPanel.add(deleteScrambleButton);
buttonPanel.add(applyScrambleButton);
buttonPanel.add(saveScrambleToFileButton);
buttonPanel.add(loadScrambleFromFileButton);
…
```

means that buttonPanel holds elements in two rows of three columns. The first three elements added will be sized and positioned automatically in the first row, and the last three elements in the last row.

The visual side of the system could be improved by changing certain groups of components so that they are displayed using a relative layout rather than absolute.

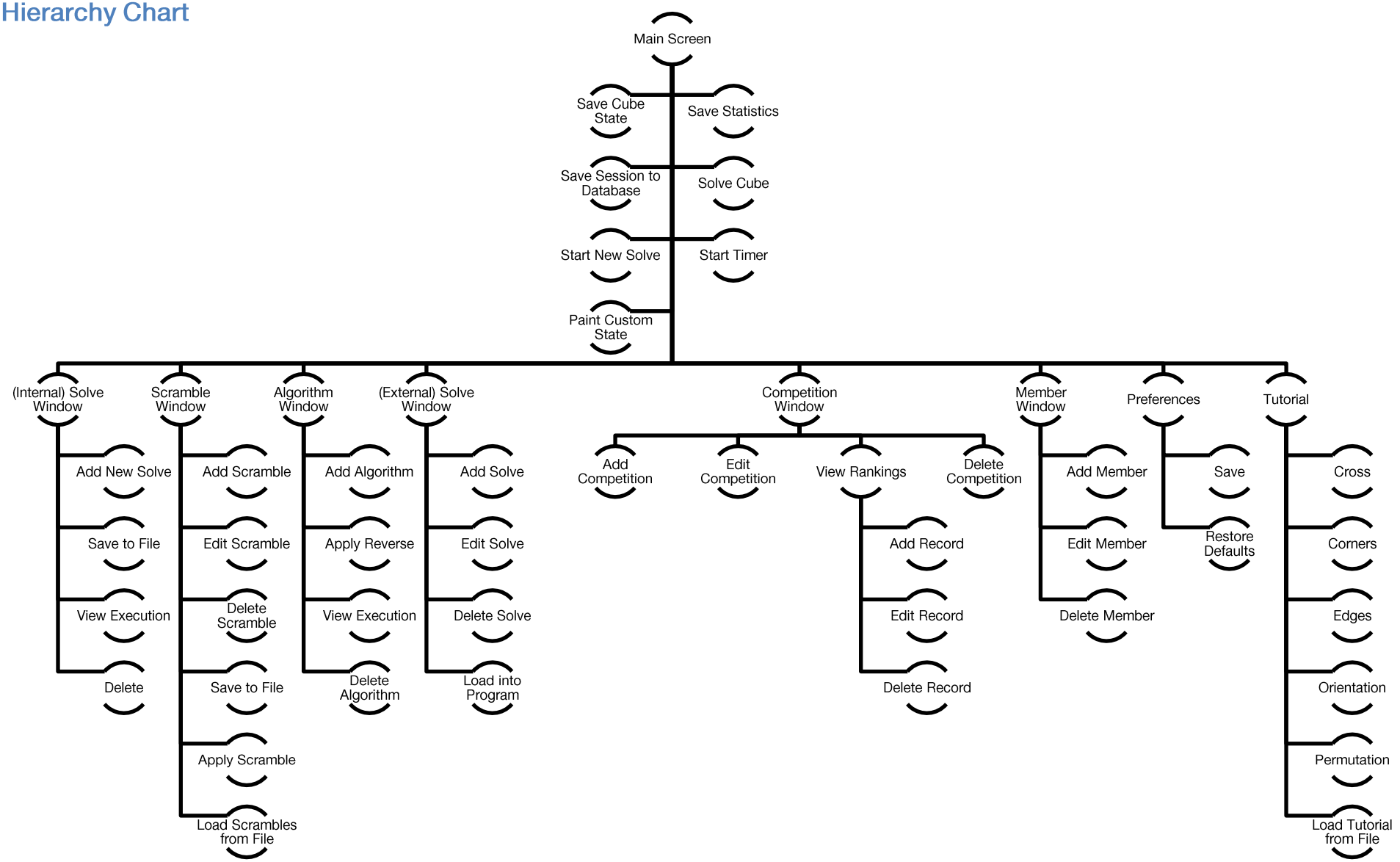
Modular Structure of Code

Class Hierarchy for Package jCube

- java.lang.Object
  - jCube.**Algorithm**
  - jCube.**AlgorithmDatabaseConnection**
  - jCube.**Competition**
  - jCube.**CompetitionDatabaseConnection**
  - java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
    - java.awt.Container
      - javax.swing.JComponent (implements java.io.Serializable)
        - javax.swing.JPanel (implements javax.accessibility.Accessible)
          - jCube.**Main** (implements java.awt.event.KeyListener)
      - java.awt.Window (implements javax.accessibility.Accessible)
        - java.awt.Frame (implements java.awt.MenuContainer)
          - javax.swing.JFrame (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)
            - jCube.**AlgorithmDatabasePopUp**
            - jCube.**ColorSelection** (implements java.awt.event.MouseListener)
            - jCube.**CompetitionDatabasePopUp**
            - jCube.**MemberCompetitionDatabasePopUp** (implements java.awt.event.KeyListener)
            - jCube.**MemberDatabasePopUp** (implements java.awt.event.KeyListener)
            - jCube.**Preferences** (implements java.awt.event.ActionListener)
            - jCube.**ScramblePopUp**
            - jCube.**SolveDatabasePopUp** (implements java.awt.event.KeyListener)
            - jCube.**TimeGraph**
            - jCube.**TimeListPopUp** (implements java.awt.event.KeyListener)
  - jCube.**CornerSolver.SolveCandidate**
  - jCube.**Cube**
  - jCube.**Cubie** (implements java.lang.Comparable<T>)
    - jCube.**Corner**
    - jCube.**Edge**
  - jCube.**CustomPdfWriter**
  - jCube.**EdgeSolver.SolveCandidate**
  - javax.swing.filechooser.FileFilter
    - jCube.**TimeGraph.MenuBar.ImageFilter**
  - jCube.**LinearSearch**
  - jCube.**Main.InspectionTimerListener** (implements java.awt.event.ActionListener)
  - jCube.**Main.MenuBar**
  - jCube.**Main.PopUpWindowListener** (implements java.awt.event.WindowListener)
  - jCube.**Main.RealTimeTimerListener** (implements java.awt.event.ActionListener)

- jCube.**Main.TimerListener** (implements java.awt.event.ActionListener)
- jCube.**Member**
- jCube.**MemberCompetition**
- jCube.**MemberCompetitionDatabaseConnection**
- jCube.**MemberDatabaseConnection**
- jCube.**Scramble**
- jCube.**Slice**
- jCube.**Solve**
  - jCube.**SolveDBType**
- jCube.**SolveDatabaseConnection**
- jCube.**SolveDatabasePopUp.MenuBar**
- jCube.**SolveMaster**
  - jCube.**CornerSolver**
  - jCube.**CrossSolver**
  - jCube.**EdgeSolver**
  - jCube.**MouseSelectionSolver**
  - jCube.**OrientationSolver**
  - jCube.**PermutationSolver**
- jCube.**Sorter**
- jCube.**Statistics**
- jCube.**Statistics.Average**
- jCube.**TextFile**
- jCube.**TimeGraph.MenuBar**
- jCube.**Tutorial**

Hierarchy Chart



Field List

(See 'Procedure List' in the appendix section for the list of public/protected/private methods)

Algorithm	
`int algorithmID`	The ID of the algorithm in the database.
`String comment`	The comment associated with the algorithm.
`String moveSequence`	The moves of the algorithm.
AlgorithmDatabaseConnection	
`String[] PRESET_ALGORITHM_UPDATES`	This stores the SQL update statements required to insert the preset algorithms into the algorithm table.
AlgorithmDatabasePopUp	
`JButton addAlgorithmButton`	Clicking this button opens an input dialog so that a new algorithm can be entered.
`JPanel algorithmListPanel`	The list of algorithms is stored inside this pane.
`JTable algorithmTable`	This stores the contents of the table.
`JButton applyReverseAlgorithmButton`	Clicking this button applies the reverse of the selected algorithm to the virtual cube in the main window.
`JPanel buttonPanel`	The buttons are placed in this panel so that they are grouped together.
`JButton deleteAlgorithmButton`	Clicking this button deletes the selected rows from the table.
`DefaultTableModel model`	By setting the model of the algorithmTable to 'model', certain cells of the table can be made uneditable, and the columns can be given names.
`long serialVersionUID`	Default serialVersionUID.
`JScrollPane tableContainer`	algorithmTable is placed inside this so that when the size of the table exceeds the size of the window, the rest of the table can be viewed by scrolling.
`JButton viewExecutionbutton`	Clicking this button performs the algorithm in real-time on the virtual cube in the main window.
`int WIDTH`	This indicates the inital width of the 'Algorithm Table' window.
ColorSelection	
`JTextField blueButton`	This is displayed as a blue rectangle on the screen.
`int BUTTON_HEIGHT`	This indicates the height of each of the rectangles in the window.
`JPanel contentPane`	The contents of the window are placed in this JPanel.
`JTextField greenButton`	This is displayed as a green rectangle on the screen.
`int HEIGHT`	This indicates the height of the window.
`JTextField orangeButton`	This is displayed as an orange rectangle on the screen.
`JTextField redButton`	This is displayed as a red rectangle on the screen.
`Color selectedColor`	This stores the selected color.
`JLabel selectedColorLabel`	This label is displayed at the bottom of the window with a black background and shows the text 'COLOR'.
`long serialVersionUID`	Default serialVersionUID.
`JTextField whiteButton`	This is displayed as a white rectangle on the screen.
`int WIDTH`	This indicates the width of the window.

`JTextField yellowButton`	This is displayed as a yellow rectangle on the screen.	
Competition		
`int competitionID`	The ID of the competition. See page 70 of the design section.	
`String date`	The date of the competition. See page 70 of the design section.	
CompetitionDatabasePopUp		
`JButton addCompetitionButton`	Clicking this button adds a row to the table.	
`JPanel buttonPanel`	The buttons are placed in this panel so that they are grouped together.	
`JPanel competitionListPanel`	The list of competitions is stored inside this panel.	
`JTable competitionTable`	Stores the contents of the table.	
`JButton deleteCompetitionButton`	Clicking this button deletes the selected competitions from the table and from the database.	
`JButton editCompetitionButton`	Clicking this button opens an input dialog so that the date of the competition can be edited.	
`MemberCompetitionDatabasePopUp memberCompetitionDatabasePopUp`	This is the window that opens when the viewRankingsButton is clicked.	
`DefaultTableModel model`	By setting the model of the algorithmTable to 'model', certain cells of the table can be made uneditable, and the columns can be given names.	
`long serialVersionUID`	Default serialVersionUID.	
`JScrollPane tableContainer`	competitionTable is placed inside this so that when the size of the table exceeds the size of the window, the rest of the table can be viewed by scrolling.	
`JButton viewRankingsButton`	Clicking this button opens the 'Member-Competition Table' window and displays the rankings for the selected competition.	
`int WIDTH`	The initial width of the window.	
Corner		
`Color[][] INITIAL_CORNERS`	This stores the initial permutation of the corners on a solved cube with white and green centres on top and front respectively.	
CornerSolver		
`Corner[] fLCorners`	This stores *Corner* objects representing the white-red-green, white-green-orange, white-orange-blue, and white-blue-red corners.	
`SolveCandidate[] solveCandidates`	This stores the remaining first layer *Corner* objects that need to be solved. See page 74 of the design section.	
`class SolveCandidate`	int index	This stores the index of the corner on the cube.
	int score	This stores the number of moves required to solve the corner.
CrossSolver		
`Edge[] crossEdges`	This stores the *Edge* objects representing the white-green, white-red, white-blue, and white-orange edges.	
`Color originalCentre`	This is used to store the centre that was originally at the front when a solution for the cross was being generated.	
Cube		

`int NUM_SLICES`	This field is used for readability in the code; instead of using the number 6 throughout, the identifier 'NUM_SLICES' can be used.	
`int NUM_EDGES`	This field is used for readability in the code; instead of using the number 6 throughout, the identifier 'NUM_SLICES' can be used.	
`int NUM_CORNERS`	This field is used for readability in the code; instead of using the number 6 throughout, the identifier 'NUM_SLICES' can be used.	
`int[][] rotationSlices`	This two-dimensional array stores the indices of the slices involved in a corresponding rotation. The first element stores the slices involved in 'x' rotations, the second stores those involved in 'y' rotations, and the third stores those involve in 'z' rotations.	
`int[][] cubieIndices`	The i^{th} element of this array stores the cubie indices for the i^{th} slice. The cubie indices indicate the index of a slice's cubies in relation to the rest of the cube.	
`Edge[] edges`	This stores the edges of the cube.	
`Corner[] corners`	This stores the corners of the cube.	
`Slice[] slices`	This stores the slices of the cube.	
Cubie		
`Color orange`	This stores a custom definition for the colour 'orange' used to paint on the cube.	
`int orientation`	This indicates the orientation of the cubie. • 0 → Oriented • 1 → Flipped/Clockwise • -1 → Anticlockwise	
`Color[] stickers`	This stores the stickers of the cubie.	
`int cubieIndex`	This stores the index of the cubie on the cube in relation to the other pieces.	
CustomPdfWriter		
`Font tableHeaderFont`	This stores a font for the headings of the table stored when saving statistics.	
Edge		
`Color[][] INITIAL_EDGES`	This stores the initial permutation of the edges on a solved cube with white and green centres on top and front respectively.	
EdgeSolver		
`Edge[] mLEdges`	This stores the *Edge* objects representing the red-green, green-orange, orange-blue, and blue-red edges.	
`SolveCandidate[] solveCandidates`	This stores the remaining middle-layer *Edge* objects that need to be solved. See page 74 of the design section.	
`class SolveCandidate`	int index	This stores the index of the edge on the cube.
	int score	This stores the number of moves required to solve the edge.
Member		
`int memberID`	This stores the ID of the member. See page 70 of the design section.	
`String forenames`	This stores the forenames of the member. See page 70 of the design section.	
`String surname`	This stores the surname of the member. See page 70 of the design section.	

String gender	This stores the gender of the member. See page 70 of the design section.
String dateOfBirth	This stores the date of birth of the member. See page 70 of the design section.
String email	This stores an email address belonging to the member. See page 70 of the design section.
String formClass	This stores the form class to which the member belongs. See page 70 of the design section.
MemberCompetition	
int competitionID	This stores the competition ID of the record. See page 71 of the design section.
int memberID	This stores the member ID of the record. See page 71 of the design section.
String[] times	This stores the 5 times of the average of the record. See page 71 of the design section.
MemberCompetitionDatabasePopUp	
int WIDTH	This stores the initial width of the Member-Competition window.
int pad	This stores the padding of the elements in the window. The greater the padding, the further towards the centre of the window the elements will be.
int fieldYSpacing	This indicates the vertical spacing between the text boxes etc. in the window.
int buttonYSpacing	This indicates the vertical spacing between the buttons etc. in the window.
int y	This variable keeps track of the current y position of the last element placed in the window.
Font fieldFont	Stores the font to be used in the text fields
boolean editing	If this variable is true, it indicates that a record is being edited, otherwise a record is being added.
int currentCompetitionID	Stores the competition ID of the selected competition.
int selectedMCIndex	After choosing to edit a record in the table, the index of this row in the table is stored in this variable.
JPanel memberCompetitionListPanel	This panel is used to store the table.
JTable memberCompetitionTable	This is the table that is displayed in the window; it stores the contents of the table and the rendering features required to display the data.
JScrollPane tableContainer	*memberCompetitionTable* is placed 'inside' this variable so that when the size of the table exceeds the size of the window, the user can scroll in order to view the rest of the table.
DefaultTableModel model	This variable can be customised so that certain cells of the table are uneditable and the columns of the table can be given text. This variable is then set as the model of *memberCompetitionTable*.
JPanel buttonPanel	The buttons in the window are placed in this panel.
JButton addRecordButton	Clicking this button opens the *Member-Competition Form* window.
JButton editRecordButton	Clicking this button opens the *Member-Competition Form* window if a row has been selected.
JButton deleteRecordButton	Clicking this button deletes a row from the table.
JLabel competitionIndicatorLabel	This label is shown at the bottom left of the

	window an indicates the competition ID of the selected competition.
`JLabel memberIDLabel`	This label is shown in the *Member-Competition Form* window with the text 'Member ID'.
`JLabel time1Label`	This label is shown in the *Member-Competition Form* window with the text 'Time 1'.
`JLabel errorMessageLabel`	This label is shown in the *Member-Competition Form* window when the user enters invalid data.
`JComboBox<Integer> memberIDComboBox`	This drop-down list stores the member IDs in the *Member-Competition Form* window.
`JTextField time1Field`	This field is shown in the *Member-Competition Form* window and the user can enter a time into this field.
`JFrame memberCompetitionInputForm`	This represents the *Member-Competition Form* window.
`JButton submitButton`	Clicking this button submits the data in the *Member-Competition Form* window for validation.
`JPanel contentPane`	This panel stores the elements of the *Member-Competition Form* window.
MemberDatabasePopUp	
`int pad`	This stores the padding of the elements in the window. The greater the padding, the further towards the centre of the window the elements will be.
`int fieldYSpacing`	This indicates the vertical spacing between the text boxes etc. in the window.
`int buttonYSpacing`	This indicates the vertical spacing between the buttons in the window.
`int y`	This variable keeps track of the current y position of the last element placed in the window.
`Font fieldFont`	Stores the font to be used in the text fields.
`boolean editing`	If this variable is true, it indicates that a record is being edited, otherwise a record is being added.
`int selectedRowIndex`	After choosing to edit a record in the table, the index of this row in the table is stored in this variable.
`JPanel memberListPanel`	This panel is used to store the table.
`JTable memberTable`	This is the table that is displayed in the window; it stores the contents of the table and the rendering features required to display the data.
`JScrollPane tableContainer`	*memberTable* is placed 'inside' this variable so that when the size of the table exceeds the size of the window, the user can scroll in order to view the rest of the table.
`DefaultTableModel model`	This variable can be customised so that certain cells of the table are uneditable and the columns of the table can be given text. This variable is then set as the model of *memberTable*.
`JPanel buttonPanel`	The buttons in the window are placed in this panel.
`JLabel forenamesLabel`	This label is shown in the *Member Form* window with the text 'Forenames'.
`JTextField forenamesField`	This field is shown in the *Member Form* window and the user can enter the forenames into this field.
`JComboBox<String> genderField`	This drop-down list stores the strings 'Male' and 'Female' in the *Member-Competition Form*

	window.
`JFrame memberInputform`	This represents the *Member Form* window.
`JButton submitButton`	Clicking this button submits the data in the *Member Form* window for validation.
`JPanel contentPane`	This panel stores the elements of the *Member Form* window.
MouseSelectionSolver (SolveMaster)	
`SolveMaster solveMaster`	This variable allows certain methods to be accessed to perform general operations on the cube.
`CrossSolver crossSolver`	This variable allows a solution for the cross to be generated.
`CornerSolver cornerSolver`	This variable allows a solution for the first-layer corners to be generated.
`EdgeSolver edgeSolver`	This variable allows a solution for the middle-layer edges to be generated.
`OrientationSolver orientationSolver`	This variable allows a solution for the orientation of the last layer to be generated.
`PermutationSolver permutationSolver`	This variable allows a solution for the permutation of the last layer to be generated.
`double xFactor`	This variable is used in calculations for determining the indices of pieces on the screen.
`double uAngle`	This variable is used in calculations for determining the indices of pieces on the screen.
`double rAngle`	This variable is used in calculations for determining the indices of pieces on the screen.
`String solution`	This accumulates the solutions generated by *crossSolver*, *edgeSolver* etc.
PermutationSolver (SolveMaster)	
`Edge[] topEdges`	This stores *Edge* objects representing the Yellow-Green, Yellow-Orange, Yellow-Blue, Yellow-Red edges.
Preferences	
`int ySpacing`	This indicates the vertical spacing beween the elements in the window.
`int pad`	This stores the padding of the elements in the window. The greater the padding, the further towards the centre of the window the elements will be.
`int FRAME_WIDTH`	This stores the width of the window.
`int MIDDLE_X`	This stores (FRAME_WIDTH/2) and is used to identify the middle x-coordinate of the window.
`int COMPONENT_WIDTH`	This stores the width of each label and text field added to the window.
`int COMPONENT_HEIGHT`	This stores the height of each label and text field added to the window.
`int REAL_TIME_SOLVING_SPEED`	This stores the default real-time solving speed. See page 71 of the design section.
`int INSPECTION_TIME`	This stores the default inspection time. See page 71 of the design section.
`int SCRAMBLE_TEXT_SIZE`	This stores the default scramble text size. See page 71 of the design section.
`JPanel fieldPanel`	This stores all components of the window other than buttons.
`JPanel buttonPanel`	This stores the buttons in the window.
`JLabel realTimeSolvingRateLabel`	This label is shown in the window with the text ‘Real-time solving speed (ms)’.
`JTextField realTimeSolvingSpeedField`	This field is shown in the window and the user

	can enter the speed into this field.
`ButtonGroup solvePieceWarningButtonGroup`	This button group stores the *solvePieceWarningYesItem* and the *solvePieceWarningNoItem* radio buttons. Using the button group, only one of the two radio buttons can be selected, not both.
`JRadioButtonMenuItem solvePieceWarningYesItem`	By selecting this radio button, a warning/information message will be shown when 'Click to Solve' mode is enabled. If the *solvePieceWarningNoItem* is selected, then no message will be shown. See page 71 of the design section.
`JButton saveAndCloseButton`	Clicking this button submits the data in the window for validation. If the data is valid, then the window *Preferences* window will close.
`JButton cancelButton`	Clicking this button discards any unsaved changes and closes the window.
`JButton restoreDefaultsButton`	Clicking this button restores the default preferences and updates the fields accordingly.
`int realTimeSolvingSpeed`	This stores the current real-time solving speed saved in the preferences file. See page 71 of the design section.
`int inspectionTime`	This stores the current inspection time saved in the preferences file. See page 71 of the design section.
`int scrambleTextSize`	This stores the current scramble text size saved in the preferences file. See page 71 of the design section.
Scramble	
`String[] moves`	This array stores all possible types of moves and their corresponding rotation directions.
`String[] directions`	This stores the three different directions that can be attributed to a move.
ScramblePopUp	
`JPanel scrambleListPanel`	The list of scrambles is stored in this panel.
`DefaultListModel<String> scrambleList`	This stores the contents (scrambles) in the list.
`JList<String> listHolder`	This is the list that is added to *scrambleListPanel* so that the contents of the list can be displayed.
`JScrollPane listScrollPane`	*listHolder* is placed 'inside' this variable so that once the size of the list exceeds the size of the window, the user can scroll to view the rest of the list.
`JPanel buttonPanel`	The buttons of the window are stored on this panel.
`JButton addScrambleButton`	Clicking this button opens an input dialog into which the user can enter a new scramble.
`int scrambleIndex`	This stores the index of the next scramble to be used if the 'Use Scrambles in List' option is selected in the main window.
`JFileChooser fileChooser`	This variable is used to select a location to save or load scrambles.
Slice	
`int NUM_EDGES`	This variable stores the number '4' and is used for readability in code.
`int NUM_CORNERS`	This variable stores the number '4' and is used for readability in code.
`Edge[] edges`	This array stores the *Edge*s in the slice.

`Corner[] corners`	This arrays stores *Corner*s in the slice.
`Color centre`	This array stores the colour of the centre of the slice.
`int[] cubieIndices`	This array stores the indices of the cubies in relation to the other pieces of the cubie. The first four elements are corner indices; the last four are edge indices.
Solve	
`String time`	This stores the time of the solve. See page 70 of the design section.
`String comment`	This stores the comment for the solve. See page 70 of the design section.
`String penalty`	This stores the penalty for the solve. See page 70 of the design section.
`String scramble`	This stores the scramble for the solve. See page 70 of the design section.
`String solution`	This stores the solution for the solve. See page 70 of the design section.
`int moveCount`	This stores the move count for the solve. See page 70 of the design section.
SolveDatabasePopUp	
`int WIDTH`	This stores the initial width of the window.
`int pad`	This stores the padding of the elements in the window. The greater the padding, the further towards the centre of the window the elements will be.
`int fieldYSpacing`	This indicates the vertical spacing between the text boxes etc. in the window.
`int buttonYSpacing`	This indicates the vertical spacing between the buttons etc. in the window.
`int y`	This variable keeps track of the current y position of the last element placed in the window.
`Font fieldFont`	Stores the font to be used in the text fields
`String timeHelpMessage`	This variable stores the text that is shown in the error message when the user enters an invalid time.
`int selectedRowIndex`	After choosing to edit a record in the table, the index of this row in the table is stored in this variable.
`boolean editing`	If this variable is true, it indicates that a record is being edited, otherwise a record is being added.
`JPanel solveTablePanel`	*tableContainer* is placed 'inside' this variable so that is can be positioned and displayed in the window.
`JScrollPane tableContainer`	*solveTable* is placed inside this variable so that when the size of the table exceeds the size of the window, the user can scroll to view the rest of the table.
`JTable solveTable`	This stores the contents of the table displayed in the window.
`DefaultTableModel model`	This variable can be customised so that certain cells of the table are uneditable and the columns of the table can be given text. This variable is then set as the model of *solveTable*.
`JPanel buttonPanel`	The buttons are placed in this panel.
`JButton addSolveButton`	Clicking this button opens the *Solve Form* window.
`String orderByAttribute`	This indicates the attribute by which the records

	in the table should be sorted.	
`String orderDirection`	This indicates whether the records should be sorted in ascending or descending order.	
`JLabel timeLabel`	This label is shown in the *Solve Form* window with the text 'Time'.	
`JTextField timeField`	This field is shown in the *Solve Form* and the user can enter the time into this field.	
`JTextArea commentField`	This is shown in the *Solve Form* and the user can enter the comment into this field.	
`JScrollPane commentScrollPane`	*commentField* is placed 'inside' this variable so that when the length of the comment exceeds the length of the *commentField*, the user can scroll to view the rest of the comment.	
`JScrollPane scrambleScrollPane`	*scrambleField* is placed 'inside' this variable so that when the length of the comment exceeds the length of the *scrambleField*, the user can scroll to view the rest of the scramble.	
`JScrollPane solutionScrollPane`	*solutionField* is placed 'inside' this variable so that when the length of the comment exceeds the length of the *solutionField*, the user can scroll to view the rest of the solution.	
`JFrame solveInputForm`	This represents *Solve Form* window.	
`JButton submitButton`	Clicking this button submits the data in the *Solve Form* window for validation.	
`JPanel contentPane`	This stores the contents of the *Solve Form*.	
`MenuBar menuBar`	This stores the contents of the menu bar in the *Solve Table* window.	
SolveDBType		
`int id`	This indicates the ID of the current solve. See page 70 of the design section.	
`String dateAdded`	This indicates the date when the solve was added to the database. See page 70 of the design section.	
SolveMaster		
`int CANCELLATIONS`	This is used for readability in the code.	
`int CORNER_EDGE`	This is used for readability in the code.	
`int CROSS`	This is used for readability in the code.	
`Cube cube`	This represents the cube displayed on screen. It can be used to generate a solution for the current state of the cube.	
`int[][] sliceEdgeSharing`	This array indicates which slices are adjacent to each other such that they share edges.	
`int[][] sliceCornerSharing`	This array indicates which slices are adjacent to each other such that they share corners.	
`LinkedList<String> solveMoves`	This accumulates the moves required to solve the cube.	
`String solutionExplanation`	This accumulates the explanation of how to solve the cube.	
Statistics		
`LinkedList<Solve> times`	This variable shares the same memory location as the list in the main window. This means that statistics of recent times can be found.	
`Average[] averages`	Stores the calculated averages.	
`class Average`	`String name`	This stores the name of the average, e.g. 'Average of 5'
	`double average`	This stores the calculated numerical

		interpretation of the average.
	`int length`	This stores the number of times in the average, e.g. average of 5 will have a length of 5.
	`String formattedTimes`	This stores the times of the average in a formatted fashion; the fastest and slowest times are surrounded by brackets.
TextFile		
`int READ`	This stores '0' for readability in the code.	
`int WRITE`	This stores '1' for readability in the code.	
`int WRITE_APPEND`	This stores '2' for readability in the code.	
`String filePath`	This stores the file path of the text file.	
`FileReader frO`	This is used with *brO* to read the data in a file.	
`BufferedReader brO`	This is used with *frO* to read the data in a file.	
`FileWriter fwO`	This is used with *pwO* to write data to a file.	
`PrintWriter pwO`	This is used with *fwO* to writer data to a file.	
TimeGraph		
`JFreeChart chart`	This is the chart displayed on the screen.	
`ChartPanel chartPanel`	The chart is placed 'inside' this variable so that it can be displayed.	
`DefaultCategoryDataset dataset`	This stores the dataset that is rendered ont the graph.	
`String chartTitle`	This stores the main title of the chart.	
`String xLabel`	This stores the label that is shown on the x-axis.	
`String yLabel`	This stores the label that is shown on the y-axis.	
`int dimension`	This variables stores either '2' or '3', indicating that the graph should be displayed in 2D or 3D respectively.	
`BasicStroke stroke`	This stores the properties of the stroke used to paint the graph.	
`boolean alwaysOnTop`	If *true*, then the window will be on top of all other windows, otherwise it can be hidden.	
TimeListPopUp		
`int pad`	This stores the padding of the elements in the window. The greater the padding, the further towards the centre of the window the elements will be.	
`int fieldYSpacing`	This indicates the vertical spacing between the text boxes etc. in the window.	
`int buttonYSpacing`	This indicates the vertical spacing between the buttons in the window.	
`int y`	This variable keeps track of the current y position of the last element placed in the window.	
`JPanel contentPane`	This panel stores the elements of the *Solve Editor* window.	
`JLabel timeLabel`	This label is shown in the *Solve Editor* window with the text 'Time'.	
`JTextField timeField`	This field is shown in the *Solve Editor* window and the user can enter the time into this field.	
`JTextArea commentField`	This is shown in the *Solve Editor* window and the user can enter the comment into this field.	
`JScrollPane commentScrollPane`	*commentField* is placed 'inside' this variable so that when the length of the comment exceeds	

	the size of *commentField*, the user can scroll to view the rest of the comment.
`JButton submitButton`	Clicking this button submits the data in the *Solve Editor* form for validation.
`Solve currentTime`	This stores the *Solve* currently being edited.
`Font fieldFont`	Stores the font to be used in the text fields.
`String helpMessage`	This variable stores the text that is shown in the error message when the user enters an invalid time.
`JFileChooser fileChooser`	This variable is used to select a location to save or load scrambles.
Tutorial	
`Cube cube`	This represents the cube displayed on screen. It can be used to generate a solution for the current state of the cube.
`SolveMaster solveMaster`	This variable allows certain methods to be accessed to perform general operations on the cube.
`CrossSolver crossSolver`	This variable allows a solution for the cross to be generated.
`CornerSolver cornerSolver`	This variable allows a solution for the first-layer corners to be generated.
`EdgeSolver edgeSolver`	This variable allows a solution for the middle-layer edges to be generated.
`OrientationSolver orientationSolver`	This variable allows a solution for the orientation of the last layer to be generated.
`PermutationSolver permutationSolver`	This variable allows a solution for the permutation of the last layer to be generated.
`String[] scrambles`	The i^{th} element of this array stores the scramble for the i^{th} sub-tutorial.
`String[] descriptions`	The i^{th} element of this array stores the description for the i^{th} sub-tutorial.
`String[][] expectedSolvedPieces`	The i^{th} element of this array stores the pieces that are expected to be solved for the i^{th} sub-tutorial.
`String[][] hints`	The i^{th} element of this array stores the hints for the i^{th} sub-tutorial.
`String[][] optimalSolutions`	The i^{th} element of this array stores the optimal solutions for the i^{th} sub-tutorial.
`String[] explanations`	The i^{th} element of this array stores the solution for the i^{th} sub-tutorial.
`TextFile currentFile`	This variable is used to open the file containing the tutorial.
`int hintIndex`	This stores the index of the next hint to be shown.
`int subTutorialIndex`	This stores the index of the current sub-tutorial.
`String[] fileData`	Each element of this array stores a single line in the text file being read.
`int numSubTutorials`	This stores the number of sub-tutorials.
`boolean tutorialLoaded`	If *true*, then a tutorial has been loaded.
`boolean[] tutorialsRequiringUserAction`	If the i^{th} element of the array is *true*, then during the i^{th} tutorial the user is granted permission to perform moves.
`boolean[] tutorialsRequiringUserSolution`	If the i^{th} element of the array is *true*, then during the i^{th} tutorial the users actions will be checked to see if they fulfil certain criteria.
`String CROSS_SOLVED`	This stores the string “cross”. Used for readability.
`String CORNERS_SOLVED`	This stores the string “corners”. Used for

	readability.
String EDGES_SOLVED	This stores the string “edges”. Used for readability.
String EDGE_ORIENTATION_SOLVED	This stores the string “edge orientation”. Used for readability.
String CORNER_ORIENTATION_SOLVED	This stores the string “corner orientation”. Used for readability.
String ORIENTATION_SOLVED	This stores the string “oll”. Used for readability.
String CORNER_PERMUTATION_SOLVED	This stores the string “corner permutation”. Used for readability.
String EDGE_PERMUTATION_SOLVED	This stores the string “edge permutation”. Used for readability.
String PERMUTATION_SOLVED	This stores the string “pll”. Used for readability.
Main	
class TimerListener	This is used to run the visible incrementing timer in the main window.
class InspectionTimerListener	This is used to run the inspection timer in the main window
class RealTimeTimerListener	This is used to perform real-time animations
class PopUpWindowListener	This WindowListener is used for most windows so that the all windows are updated in the appropriate manner.
class MenuBar	Represents the menu bar in the main window.
long serialVersionUID	Default serialVersionUID
float strokeThickness	This stores the thickness of the strokes used to paint the visual cube
BasicStroke stroke	This stroke is used to paint the lines on the visual cube
Cube cube	The cube shown in the main window
SolveMaster solveMaster	This variable allows useful methods to be accessed to operate on the cube
CrossSolver crossSolver	This allows a solution for the cross to be generated
CornerSolver cornerSolver	This allows a solution for the corners to be generated
EdgeSolver edgeSolver	This allows a solution for the edges to be generated
OrientationSolver orientationSolver	This allows a solution for the orientation to be generated
PermutationSolver permutationSolver	This allows a solution for the permutation to be generated
Tutorial tutorial	This is used to load tutorials and access their features
MouseSelectionSolver selectionSolver	This is used to solve a piece that is clicked when in Click-to-Solve mode
ColorSelection colorSelection	This represents the Color Selection window
Preferences preferencesPopUp	This represents the Preferences window
ScramblePopUp scramblePopUp	This represents the Scramble List window
AlgorithmDatabasePopUp algorithmDatabasePopUp	This represents the Algorithm Table window
SolveDatabasePopUp solveDatabasePopUp	This represents the Solve Table window
CompetitionDatabasePopUp competitionDatabasePopUp	This represents the Competition Table window
MemberDatabasePopUp memberDatabasePopUp	This represents the Member Table window
Color[][] faceletColors	This array stores the facelet/sticker colours for each sticker on the cube. The first dimension

	stores the stickers for the top face,
`int[] edgePaintOrder`	This specifies the order in which the edges should be painted when using the for loops later
`int[] cornerPaintOrder`	This specifies the order in which the corners should be painted when using the for loops later
`int numTimesToGraph`	This stores the number of times that should be graphed
`Timer incTimer`	This is used as the display-timer used to record times for solves
`Timer inspectionTimer`	This is used as the count-down timer for inspection
`Timer realTimeSolutionTimer`	This is used to regulate the speed of animations (real-time solving etc.)
`float elapsedTimingSeconds`	Stores the number of elapsed seconds while timing
`int elapsedMinutes`	Stores the number of elapsed minutes while timing
`int inspectionTimeRemaining`	Stores the number of seconds remaining during inspection time
`boolean tutorialIsRunning`	This indicates whether the system is in tutorial mode or not
`boolean timeToBeRecorded`	This indicates whether the time/solve should be recorded when the timer stops
`boolean movesAllowed`	Indicates whether or not the user is allowed to perform moves.
`boolean cubeSolved`	Indicates whether or not the cube is solved
`boolean timerHasPermissionToStart`	Indicates whether or not incTimer has permission to start
`boolean movesToBeRecorded`	Indicates whether or not the moves being performed by the user should be recorded
`boolean clickToSolve`	Indicates whether or not the user can click to solve a piece
`boolean customPaintingInProgress`	Indicates whether or not the user can paint a custom state on the cube
`boolean customTimerRunning`	Indicates whether or not incTimer is running for a time that is being performed by the user manually (i.e. the 'Start New Solve' button was not clicked; spacebar was pressed to start)
`float timingDisplayInterval`	This specifies how many seconds to wait before rendering the next time on the display.
`LinkedList<String> trackingMoves`	This can store moves performed by the user in different situations. See page 74 of the design section.
`LinkedList<Solve> solves`	This stores the data for the solves listed at the right-hand side of the main window
`String currentScramble`	Stores the last scramble used to store the cube
`String currentPenalty`	Stores the penalty to apply to the current solve
`JButton resetCubeButton`	Clicking this button resets the cube to the solve state white on top and green on front
`Statistics statistics`	This can be used to calculate statistics for the current session
`TimeListPopUp timeListPopUp`	This represents the Solve Editor window
`TimeGraph timeGraph`	This represents the Time Graph window
`JLabel timeLabel`	This label shows the current time of incTimer or inspectionTimer at the top-left of the window
`JTextArea scrambleLabel`	This label shows the current scramble being used
`JPanel topComponentsPanel`	This panel stores the components at the top of

	the main window
`JList<String> listHolder`	This list can be placed in the panel at the right-hand side of the screen so that the list can be displayed
`DefaultListModel<String> timeDisplayList`	This list stores the contents of the displayed elements in the time list at the right-hand side of the screen
`JPanel timeListPanel`	This stores the listed times
`JPanel totalPaintingPanel`	This panel holds the painted components of the main window (such as cubePanel)
`JPanel timingDetailsComponentsPanel`	Stores the components unique to timing mode
`KPanel timingDetailsButtonPanel`	Stores the buttons unique to timing mode
`JPanel bottomPanel`	Stores the components at the bottom of the main window
`JFrame totalFrame`	The total frame/window to be shown
`JPanel tutorialButtonsPanel`	Stores the buttons unique to tutorial mode
`JPanel tutorialComponentsPanel`	Stores the components unique to tutorial mode
`JTextArea tutorialTextArea`	The text for sub-tutorials are shown in this text area
`JTextArea timingDetailsTextArea`	The text used in timing mode, such as statistics or solutions to cube-states, is shown in this text area.
`JScrollPane timelistScrollPane`	This scroll pane allows all contents of the time-list to be viewed when its size exceeds the size of the panel in which it is placed
`JScrollPane tutorialScrollPane`	This scroll pane allows all text in the tutorial text area to be viewed when the length of the text exceeds the size of the text area
`JScrollPane timingDetailsScrollPane`	This scroll pane allows all text in the timing-details text area to be viewed when the length of the text exceeds the size of the text area
`JButton timingDetailsResetSessionButton`	Clicking this button clears all times in the list at the right-hand side of the main window
`JButton timingDetailsStartNewSolveButton`	Clicking this button starts a new solve
`JButton tutorialResetStateButton`	Clicking this button resets the cube to the solved state with white on top and green on front
`JButton tutorialNextButton`	Clicking this button loads the next sub-tutorial
`JButton tutorialBackButton`	Clicking this button loads the previous sub-tutorial
`JButton tutorialHintButton`	Clicking this button loads the next hint in the sub-tutorial
`JButton tutorialShowDescriptionButton`	Clicking this button shows the description for the current sub-tutorial
`JButton tutorialShowSolutionButton`	Clicking this button shows the solution for the current sub-tutorial
`Main cubePanel`	The visual cube is stored in this panel

List of System Settings/Configuration

The default settings/preferences for the program are:

- Real-time solving speed (ms): 250
- Inspection time (seconds): 15
- Click to Solve Warning Enabled: Enabled
- Scramble text size: 27

For the system to run correctly, Java and a PDF reader must be installed on the machine. These can be downloaded from www.java.com/en/ and *get.adobe.com/uk/reader* respectively.

For development, all external libraries/JARs must be linked to the build path of the project.
To do this in Eclipse: (*Project → Properties → Java Build Path → Libraries Tab → Add External JARs*) then choose the JARs to add.

The JARs needed for the system are:

- SQLite-JDBC (sqlite.org): this is used to access the database.
- iTextPDF (itextpdf.com): this is used to generate PDF files.
- JFreeChart (jfree.org): this is used to display graphs.
- JCommon (jfree.org): this is used to display graphs.

Ensure the directory in which the program files are placed is not a read-only directory, otherwise the system will probably not function correctly.

Samples of Annotated Algorithms

Sorter – Sorting an array of numbers

– Go to page 402 of the appendix section to see Sorter Class.

Sorts the specified list into ascending order. The argument is checked to see if it is null or has fewer than 2 elements.

Parameters:

`list` - the list to sort

```
public static void quickSort(double[] list) {
   /*
    * If the list is null, then it cannot be sorted, so return; If the
    * list's length is less than 2, then it is already sorted, so return.
    */
   if ((list == null) || (list.length <= 1))
      return;

   qSort(list, 0, list.length - 1);
}
```

Sorts the specified list in to ascending order.

Parameters:

`list` - the list to be sorted

`start` - the index of the first element to be sorted

`end` - the index of the last element to be sorted

```
private static void qSort(double[] list, int start, int end) {
   double pivot = list[(start + end) / 2];
   // This points to the element on the left of the pivot
```

```
   int i = start;
   // This points to the element on the right of the pivot
   int j = end;

   while (i <= j) {
      while (list[i] < pivot)
         ++i;

      while (list[j] > pivot)
         --j;

      if (i <= j) {
         swap(list, i, j);
         ++i;
         --j;
      }
   }

   if (i < end)
      qSort(list, i, end);

   if (start < j)
      qSort(list, start, j);
}
```

Sorter - Sort an array of SolveDBType by the time field

- See section 3.01 of the testing section. This algorithm correctly sorts the records and rearranges their corresponding data, i.e. not just the time fields are sorted.
- Go to page 402 of the appendix section to see Sorter class.

Checks that the list is not null or empty then sorts the list with the fastest time first and the slowest time last.

Parameters:

`list` - the list to be sorted

`start` - the index of the first element to be sorted

`end` - the index of the last element to be sorted

```
public static void sortByTime(SolveDBType[] list, int start, int end) {
   /*
    * If the list is null, then it cannot be sorted, so return; If the
    * list's length is less than 2, then it is already sorted, so return.
    */
   if ((list == null) || (list.length <= 1))
      return;

   sBT(list, start, end);
}
```

Sorts a list with the fastest time first and the slowest time last.

Parameters:

`list` - the list to be sorted

`start` - the index of the first element to be sorted

`end` - the index of the last element to be sorted

```
public static void sBT(SolveDBType[] list, int start, int end) {
   double pivot = Solve.getFormattedStringToDouble(list[((start + end) / 2)].getStringTime());

   // This points to the element on the left of the pivot
   int i = start;
   // This points to the element on the right of the pivot
   int j = end;

   // Stores the numeric representation of the time currently being
   // examined
   double current = 0;

   if (pivot == -1)
      pivot = 1e10;

   while (i <= j) {
      /*
       * This converts the ith element to its numerical value, e.g.
       * 1:10.50 becomes 70.5. If the element is -1, then set it to
       * infinity (1e10), otherwise take its numerical value. This results
       * in the list being sorted with -1s at the end rather than at the
       * start because -1 represents DNF which represents infinity.
       */
      while (((current = Solve.getFormattedStringToDouble(list[i].getStringTime())) == -1 ? 1e10 : current) < pivot)
         ++i;

      while (((current = Solve.getFormattedStringToDouble(list[j].getStringTime())) == -1 ? 1e10 : current) > pivot)
         --j;

      if (i <= j) {
         swap(list, i, j);
         ++i;
         --j;
      }
   }

   if (i < end)
      sBT(list, i, end);
```

```
   if (start < j)
      sBT(list, start, j);
}
```

AlgorithmDatabaseConnection - Reset IDs index the algorithm table in the database

- See section 4.14 of the testing section. This algorithm correctly resets all IDs so that they are continuous. It does not crash if the table size is zero.
- Go to page 31 of the appendix section to see AlgorithmDatabaseConnection class.

Resets the IDs in the 'algorithm' table so that they are continuous, e.g. if the IDs were 1, 2, 5, 6, 7, 12, 13 then they would be reset to 1, 2, 3, 4, 5, 6, 7

Throws:

`java.lang.ClassNotFoundException` - if SQLite classes are missing

`java.sql.SQLException` - if the table does not exist etc.

```
public static void resetIDs() throws ClassNotFoundException, SQLException {
   Class.forName("org.sqlite.JDBC");
   // This is used to establish a connection with the database.
   Connection connection = null;
   // This is used to perform updates and queries on the database
   Statement statement = null;
   // This stores the records returned from a query.
   ResultSet rs;

   try {
      connection = DriverManager.getConnection("jdbc:sqlite:res/cube.db");
      statement = connection.createStatement();
      statement.setQueryTimeout(30); // set timeout to 30 sec.

      rs = statement.executeQuery("SELECT * FROM algorithm;");

      statement = connection.createStatement();
      statement.executeUpdate("DROP TABLE IF EXISTS algorithmCopy");
      statement.executeUpdate("CREATE TABLE algorithmCopy(" + "algorithmID INTEGER PRIMARY KEY AUTOINCREMENT,"
            + "moveSequence TEXT," + "comment TEXT" + ");");

      while (rs.next()) {
         String moveSequence = rs.getString("moveSequence"), comment = rs.getString("comment");
         statement.executeUpdate(String.format(
```

```
                "INSERT INTO algorithmCopy(moveSequence, comment) VALUES (\"%s\", \"%s\");", moveSequence,
                comment));
      }

      statement.executeUpdate("DROP TABLE algorithm");
      statement.executeUpdate("ALTER TABLE algorithmCopy RENAME TO algorithm");
   } catch (SQLException e) {
      if (connection != null)
         connection.close();

      throw new SQLException();

   }
}
```

Competition – Date checking

– Go to page 50 of the appendix section to see Competition class.

```
Returns a value indicating whether the argument is a valid date for a competition

Parameters:
    dateString - the date to be analysed
  Returns:
    true if the argument is valid and is in the format dd/MM/yyyy; false otherwise
public static boolean isValidDate(String dateString) {
   try {
      /*
       * This will throw a ParseException if dateString is not in the
       * correct format. No assignment statement is needed; this statement
       * just confirms that dateString is in the correct format to
       * proceed.
       */
      new SimpleDateFormat("dd/MM/yyyy").parse(dateString);

      // Stores an integer representation of the day in the month.
      int day;
      // Stores an integer representation of the month in the year.
      int month;
      // Stores an integer representation of the year.
      int year;

      day = Integer.valueOf(dateString.substring(0, dateString.indexOf("/")));
      dateString = dateString.substring(dateString.indexOf("/") + 1);

      month = Integer.valueOf(dateString.substring(0, dateString.indexOf("/")));
      year = Integer.valueOf(dateString.substring(dateString.indexOf("/") + 1));

      /*
```

```
       * This means the method will return false if a month greater than
       * '12' is indicated. It will also return false if the day indicated
       * is greater than the number of days in the specified month.
       */
      if ((month > 12) || (getNumDaysInMonth(month, year) < day))
         return false;
      else
         return true;
   } catch (Exception e) {
      return false;
   }
}
```

Parameters:

`month` - the month of the date in question

`year` - the year of the date in question

Returns:

the number of days in the month

```
private static int getNumDaysInMonth(int month, int year) {
   switch (month) {
   case 2:
      /*
       * This is a leap year checker. A year is a leap year if it is
       * divisible by four and not divisible by 100 unless it is also
       * divisible by 400.
       */
      if ((year % 4 == 0) && ((year % 100 == 0) ? (year % 400 == 0) : true)) {
         return 29;
      } else {
         return 28;
      }
   case 4:
   case 6:
   case 7:
   case 11:
      return 30;
   default:
      return 31;
   }
}
```

CompetitionDatabasePopUp – Editing a competition

- See section 1.34 of the testing section.
- Go to page 58 of the appendix section to see CompetitionDatabasePopUp class.

Performs the operations required to allow the selected row to be edited and validation to be performed on the data entered

```
private void editButtonFunction() {
   // Stores the string entered by the user.
   String date;
   // Stores the index of the row in the table selected by the user.
   int row = competitionTable.getSelectedRow();

   if (row != -1) {
      // Stores the date-string shown in the selected row so that if
      // editing, the existing date can be shown in the input window.
      String originalDate = "" + competitionTable.getValueAt(row, 1);
      date = JOptionPane.showInputDialog(null, "Enter Date of Competition",
            (originalDate.equals("")) ? "dd/MM/YYYY" : originalDate);

      if (date == null)
         return;

      date += "";
      date = date.trim();

      if ((date.equals("")) || (!Competition.isValidDate(date))) {
         JOptionPane.showMessageDialog(null, "Invalid date", "Error", JOptionPane.ERROR_MESSAGE);
      } else {
         try {
            CompetitionDatabaseConnection.executeUpdate(String.format("UPDATE competition "
                  + "SET competitionDate = \"%s\" " + "WHERE competitionID = %d", date,
                  Integer.valueOf("" + model.getValueAt(competitionTable.getSelectedRow(), 0))));
            competitionTable.setValueAt(date, row, 1);
         } catch (Exception exc) {
            exc.printStackTrace();
         }
```

```
        }
    }
}
```

CornerSolver - Solve first-layer corners

- See sections 3.05 - 3.10 of the testing section. This algorithm produces a valid solution for the first-layer corners and stores the moves and explanation generated.
- Go to page 70 of the appendix section to see CornerSolver class.

Solves the corners in the first layer of the cube and records the solution and explanation at the same time

```
public void solveFirstLayerCorners() {
   // Stores the properties of the corner to be solved.
   Corner corner;
   rotateToTop(Color.yellow);
   // Stores the index of the unsolved corner being examined.
   int solveCandidatesIndex;
   // Stores the index of the corner to solve.
   int indexOfCornerToSolve;
   // Stores the stickers of the corner so that a better explanation can be
   // generated.
   Color[] stickers;

   solveCandidates = new SolveCandidate[4];
   for (int i = 0; i < 4; ++i)
      // Initialise solveCandidates
      solveCandidates[i] = new SolveCandidate();

   while (!firstLayerCornersSolved()) {
      solveCandidatesIndex = 0;

      for (int i = 0; i < 8; ++i) {
         corner = cube.getCorner(i);
         if (isFLCorner(corner) && (!isPieceSolved(corner))) {
            solveCandidates[solveCandidatesIndex].index = i;
            solveCandidates[solveCandidatesIndex].score = getScore(i);
            ++solveCandidatesIndex;
         }
      }

      indexOfCornerToSolve = solveCandidates[getIndexOfMinScore(solveCandidatesIndex)].index;
```

```
      stickers = cube.getCorner(indexOfCornerToSolve).getStickers();
      solutionExplanation += String.format("Corners - %s-%s-%s corner:\n", Cubie.getColorToWord(stickers[0]),
            Cubie.getColorToWord(stickers[1]), Cubie.getColorToWord(stickers[2]));

      solveCorner(indexOfCornerToSolve);
   }

   try {
      // Removes the last newline character
      solutionExplanation = solutionExplanation.substring(0, solutionExplanation.lastIndexOf("\n"));
   } catch (IndexOutOfBoundsException e) {
   }
}
```

Solves the specified corner, and records the solution and explanation at the same time

Parameters:

`currentIndex` - the index of the corner to be solved

```
public void solveCorner(int currentIndex) {
   // Stores a copy of the properties of the corner to be solved.
   Corner corner = cube.getCorner(currentIndex);
   // Stores the orientation (-1, 0, 1) of the corner.
   int orientation = corner.getOrientation();
   // Stores the index of the setup location for the corner.
   int overCornerIndex = getIndexOfDestination(corner);
   overCornerIndex += (overCornerIndex % 2 == 0) ? -3 : -5;

   if (isPieceSolved(corner))
      return;

   /*
    * The cube will perform y rotatations until the corner is at URF (cubie
    * index 2) or URD (cubie index 7) (
    */
   while (cube.getCorner((currentIndex >= 4) ? 7 : 2).compareTo(corner) == -1) {
```

```
      cube.rotate("y");
      catalogMoves("y");
   }

   // i.e. if the corner is in the bottom layer
   if (currentIndex >= 4) {
      if (orientation > 0) {
         cube.performAbsoluteMoves("R U' R'");
         catalogMoves("R U' R'");
         solutionExplanation += "The corner is trapped in the bottom layer, so remove it using  R U' R'";
      } else {
         cube.performAbsoluteMoves("R U R' U'");
         catalogMoves("R U R' U'");
         solutionExplanation += "The corner is trapped in the bottom layer, so remove it using  R U R'";
      }
      solutionExplanation += "\n";
      // The corner is now at cubie index 2, so recursively call the
      // method to solve the corner from this index
      solveCorner(2);
   }
   /*
    * overCornerIndex is basically the setup position for the corner. If
    * the corner is at this index, then only one more sequence of moves
    * needs to be performed in order to solve the corner
    */
   else if (currentIndex == overCornerIndex) {
      if (lastMoveWasU(getCatalogMoves()))
         solutionExplanation += "Bring the corner over its destination. ";

      if (orientation == 0) {
         cube.performAbsoluteMoves("R U2 R' U' R U R'");
         catalogMoves("R U2 R' U' R U R'");
         solutionExplanation += "White is facing top, so perform  R U2 R' U' R U R'";
      } else if (orientation == 1) {
         cube.performAbsoluteMoves("R U R'");
         catalogMoves("R U R'");
         solutionExplanation += "White is facing right, so perform  R U R'";
      } else {
         cube.performAbsoluteMoves("F' U' F");
```

```
         catalogMoves("F' U' F");
         solutionExplanation += "White is facing front, so perform  F' U' F";
      }
      solutionExplanation += "\n\n";
   }
   /*
    * If this block is reached, then it means the corner is in the top
    * layer, is at cubie index 2, and needs to be moved (using U) to its
    * setup/overCornerIndex position. This method performs U until the
    * corner is at its setup position then rotates the cube so that the
    * corner is still at cubie index 2.
    */
   else {
      for (int i = 0; i < ((overCornerIndex - currentIndex) + 4) % 4; ++i) {
         cube.performAbsoluteMoves("U");
         catalogMoves("U");
      }

      for (int i = 0; i < ((overCornerIndex - currentIndex) + 4) % 4; ++i) {
         cube.rotate("y'");
         catalogMoves("y'");
      }
      solveCorner(2);
   }
}
```

CornerSolver – Determining whether the last move was 'U'

- See sections 3.05 - 3.10 of the testing section.
- Go to page 70 of the appendix section to see CornerSolver class.

Parameters:

`originalMoves` - the moves to be examined

Returns:

true if the last move (after simplification) was U;
false otherwise

```
private boolean lastMoveWasU(LinkedList<String> originalMoves) {
   // Stores a copy of originalMoves because 'moves' is later altered to
   // check for cancellations.
   LinkedList<String> moves = new LinkedList<>();
   int size = originalMoves.size();

   for (int i = 0; i < size; ++i)
      moves.add(originalMoves.get(i));

   simplifyMoves(moves, SolveMaster.CANCELLATIONS);

   if (moves.size() == 0)
      return false;

   else {
      for (int i = moves.size() - 1; i >= 0; --i) {
         if ("xyz".contains(moves.get(i).substring(0, 1)))
            continue;
         else if (!moves.get(i).contains("U"))
            return false;
         else
            return true;
      }
   }
```

```
    return false;
}
```

CrossSolver – Solving the cross

- See sections 3.05 - 3.10 of the testing section. This algorithm generates a correct solution for the cross and stores the moves required, but it does not save an explanation. The original algorithm saved the explanation, but the problem with this is that there are many simplifications made after the solution is generated; this means that the explanation doesn't match the moves. To fix this, another algorithm would have to be devised which would be able to generate an explanation for a given solution. Currently, this is the most complex algorithm in the program, both in terms of method and code.
- Go to page 78 of the appendix section to see CrossSolver class.

Solves the cross of the cube in the main window, and records the solution and explanation at the same time

```
public void solveCross() {
   // Stores the index of the current cross edge in the current slice.
   int indexOfCrossEdgeInSlice;
   // Stores the index of the target for the current cross edge in the E
   // slice.
   int workingEdgeTarget;
   // Counts the number of rotations made so that no more than four are
   // performed.
   int count;
   // Stores the expected offset between the current cross edge and a
   // solved cross edge in the top slice.
   int expectedOffset;
   // Stores true if a cross edge is found in the E slice; false otherwise.
   boolean workingEdgeFound;

   rotateToTop(Color.white);
   clearMoves();
   catalogMoves("Holding " + Cubie.getColorToWord(cube.getSlice(0).getCentre()) + " top and "
         + Cubie.getColorToWord(cube.getSlice(4).getCentre()) + " front:");

   while (!isCrossSolved()) {
      count = 0;
      workingEdgeFound = false;

      /*
       * This searches the middle-layer/E-slice for a cross edge, which is
       * then designated as the 'working' edge. If a cross edge is found,
```

```
       * it will be at the FR position because the cube is rotated between
       * each comparison.
       */
      for (count = 0; count < 4; ++count) {
         if (LinearSearch.linearSearch(cube.getSlice(4).getEdge(1).getStickers(), Color.white) != -1) {
            workingEdgeFound = true;
            break;
         } else {
            cube.performAbsoluteMoves("y");
            catalogMoves("y");
         }
      }

      if (workingEdgeFound) {
         /*
          * If orientation = 0, then the target will be UR, otherwise it
          * will be UF. This is because if the edge's orientation is 0,
          * then you would place it in the top face by performing R, i.e.
          * by placing it at UR, and if the orientation was 1, then you
          * would place it in the top face by performing F, i.e. by
          * placing it at UF
          */
         workingEdgeTarget = cube.getSlice(4).getEdge(1).getOrientation() + 1;
         // Stores a copy of workingEdgeTarget for later manipulation
         int helperTarget = workingEdgeTarget;

         if /* there is a cross edge in U-slice */(((indexOfCrossEdgeInSlice = getIndexOfCrossEdgeInSlice(0, 0)) !=
-2)
               || (((indexOfCrossEdgeInSlice = getIndexOfCrossEdgeInSlice(0, 1)) != -2))) {
            if /* the orientation of the found cross edge is 0 */(cube.getSlice(0)
                  .getEdge(indexOfCrossEdgeInSlice).getOrientation() == 0) {
               /*
                * This calculates the expected offset (relative
                * distance) between the working edge and the existing
                * cross edge. See getExpectedOffset(...) for more
                * information.
                */
               expectedOffset = getExpectedOffset(
                     cube.getSlice(0).getEdge(indexOfCrossEdgeInSlice).getStickers()[1],
```

```
                    cube.getSlice(4).getEdge(1).getStickers()[(cube.getSlice(4).getEdge(1).getOrientation() + 1) %
2]);

               indexOfCrossEdgeInSlice = (indexOfCrossEdgeInSlice == 3) ? -1 : indexOfCrossEdgeInSlice;

               /*
                * This moves the existing cross edge to the correct
                * position.
                */
               while ((workingEdgeTarget - indexOfCrossEdgeInSlice != expectedOffset)
                     && (workingEdgeTarget - indexOfCrossEdgeInSlice != (expectedOffset + 4) % 4)) {
                  cube.performAbsoluteMoves("U");
                  catalogMoves("U");
                  indexOfCrossEdgeInSlice = ((indexOfCrossEdgeInSlice + 2) % 4) - 1;
               }
            }
            /*
             * Reaching this block means there is an unoriented edge in
             * the top slice, so you can remove this from the top slice
             * and solve the working edge at the same time.
             */
            else {
               indexOfCrossEdgeInSlice = (indexOfCrossEdgeInSlice == 3) ? -1 : indexOfCrossEdgeInSlice;
               for (int i = 0; i < (workingEdgeTarget - indexOfCrossEdgeInSlice + 4) % 4; ++i) {
                  cube.performAbsoluteMoves("U");
                  catalogMoves("U");
               }
            }
         } else if /* there is an edge in bottom slice */(((indexOfCrossEdgeInSlice = getIndexOfCrossEdgeInSlice(
               1, 0)) != -2) || ((indexOfCrossEdgeInSlice = getIndexOfCrossEdgeInSlice(1, 1)) != -2)) {
            indexOfCrossEdgeInSlice = (indexOfCrossEdgeInSlice == 3) ? -1 : indexOfCrossEdgeInSlice;

            /*
             * If the bottom slice contains an unoriented edge, then you
             * want to place this edge in the plane of motion of the
             * working edge so that you can solve the working edge and
             * fix the bad edge in the bottom layer at the same time.
             */
            if (sliceContainsCrossEdgeOriented(1, 1)) {
```

```
            helperTarget = (helperTarget + 2) % 4;

            while (cube.getSlice(1).getEdge(helperTarget).getStickers()[1] != Color.white) {
               cube.performAbsoluteMoves("D");
               catalogMoves("D");
            }
         } else {
            while (isCrossEdge(cube.getSlice(1).getEdge((helperTarget)))) {
               cube.performAbsoluteMoves("D");
               catalogMoves("D");
            }
         }
      } else { // All edges are in the E-Slice

         if /* an oriented edge will become misoriented */(cube.getSlice(helperTarget + 1).getEdge(1)
               .getOrientation() == (helperTarget % 2)) {
            cube.performAbsoluteMoves((helperTarget == 1) ? "B'" : "L");
            catalogMoves((workingEdgeTarget == 1) ? "B'" : "L");
         }
      }

      cube.performAbsoluteMoves((workingEdgeTarget == 1) ? "R" : "F'");
      catalogMoves((workingEdgeTarget == 1) ? "R" : "F'");
   } else { // There are no edges in the top or bottom layers
      // Stores the index of the cross edge's destination in the top
      // slice.
      int uSliceIndex = 0;
      // Stores the location of the cross edge in the bottom slice.
      int dSliceIndex;
      count = 0;

      if /* there is an oriented cross edge in the bottom slice */((dSliceIndex = getIndexOfCrossEdgeInSlice(
            1, 0)) != -2) {
         if (dSliceIndex == 1)
            uSliceIndex = 3;
         else if (dSliceIndex == 3)
            uSliceIndex = 1;
         else
            uSliceIndex = dSliceIndex;
```

```
            /*
             * Move any unoriented cross edges in the top slice to the
             * target for the edge found in the bottom slice so that two
             * operations can be performed at once.
             */
            if (sliceContainsCrossEdgeOriented(0, 1))
               while (cube.getSlice(0).getEdge(uSliceIndex).getStickers()[1] != Color.white) {
                  cube.performAbsoluteMoves("u");
                  catalogMoves("U");
               }
            else {
               /*
                * Move any existing (and oriented) cross edges out of
                * the way
                */
               while (isCrossEdge(cube.getSlice(0).getEdge(uSliceIndex))) {
                  cube.performAbsoluteMoves("u");
                  catalogMoves("U");
               }
            }
         }
         /*
          * If there is an unoriented cross edge in the top slice and no
          * oriented edges in the bottom slice
          */
         else if ((uSliceIndex = getIndexOfCrossEdgeInSlice(0, 1)) != -2) {
            if (uSliceIndex == 1)
               dSliceIndex = 3;
            else if (uSliceIndex == 3)
               dSliceIndex = 1;
            else
               dSliceIndex = uSliceIndex;

            /*
             * If there is an unoriented cross edge in the bottom
             * slice...
             */
            if (sliceContainsCrossEdgeOriented(1, 1)) {
```

```
            while (cube.getSlice(1).getEdge(dSliceIndex).getStickers()[1] != Color.white) {
               cube.performAbsoluteMoves("d");
               catalogMoves("D");
            }
         } else {
            while (isCrossEdge(cube.getSlice(1).getEdge(dSliceIndex))) {
               cube.performAbsoluteMoves("d");
               catalogMoves("D");
            }
         }
      }
      /*
       * If there is an unoriented cross edge in the bottom slice and
       * no unoriented cross edges in the top slice
       */
      else {
         dSliceIndex = getIndexOfCrossEdgeInSlice(1, 1);
         count = 0;

         if (dSliceIndex == 1)
            uSliceIndex = 3;
         else if (dSliceIndex == 3)
            uSliceIndex = 1;
         else
            uSliceIndex = dSliceIndex;

         try {
            /*
             * Move oriented cross edges out of the way
             */
            while ((count < 4) && (isCrossEdge(cube.getSlice(0).getEdge(uSliceIndex)))) {
               ++count;
               dSliceIndex = (dSliceIndex + 1) % 4;

               if (dSliceIndex == 1)
                  uSliceIndex = 3;
               else if (dSliceIndex == 3)
                  uSliceIndex = 1;
               else
```

```
                    uSliceIndex = dSliceIndex;

                cube.performAbsoluteMoves("d");
                catalogMoves("D");
             }
          }
          /*
           * This exception is thrown if all cross edges are in the
           * top layer, hence uSliceIndex will be negative
           */
          catch (ArrayIndexOutOfBoundsException e) {
             if (edgesAreOpposite(cube.getEdge(0), cube.getEdge(2))) {
                cube.performAbsoluteMoves("R U2 R' U2 R");
                catalogMoves("R U2 R' U2 R");
             } else if (edgesAreOpposite(cube.getEdge(0), cube.getEdge(1))) {
                cube.performAbsoluteMoves("R' U' R U'");
                catalogMoves("R' U' R U'");
             } else {
                cube.performAbsoluteMoves("L U L' U");
                catalogMoves("L U L' U");
             }
          }
       }

       /*
        * Perform the move required to get the edge in the top slice.
        */
       switch (uSliceIndex) {
       case 0:
          cube.performAbsoluteMoves("B");
          catalogMoves("B");
          break;
       case 1:
          cube.performAbsoluteMoves("R");
          catalogMoves("R");
          break;
       case 2:
          cube.performAbsoluteMoves("F");
          catalogMoves("F");
```

```
            break;
         case 3:
            cube.performAbsoluteMoves("L");
            catalogMoves("L");
            break;
         default:
            break;
         }
      }
   }

   // Stores the y moves in the solution.
   LinkedList<String> temp = new LinkedList<>();
   LinkedList<String> catalogMoves = getCatalogMoves();

   for (int i = 0; i < catalogMoves.size(); ++i) {
      if (catalogMoves.get(i).contains("y"))
         temp.add(catalogMoves.get(i));
   }

   simplifyMoves(temp, SolveMaster.CANCELLATIONS);

   if (temp.size() > 0) {
      if (temp.get(0).contains("'")) {
         // The cross doesn't require any
         // rotations, so undo any
         // rotations performed.
         cube.rotate("y");
      } else if (temp.get(0).contains("2")) {
         cube.rotate("y2");
      } else {
         cube.rotate("y'");
      }
   }

   cube.rotate("z2");
   catalogMoves("z2");
}
```

Cube – Rotating

- See sections 3.05 - 3.10 of the testing section.
- Go to page 91 of the appendix section to see Cube class.

Performs the specified rotation on the cube

Parameters:

`rotation` - the rotation to be performed - this should one of the following:
{x, x', y, y', z, z'}

```
public void rotate(String rotation) {
   if ((rotation.trim().length() == 0) || !"xyz".contains(rotation.substring(0, 1)))
      return;

   if (rotation.equals("z")) {
      rot("y");
      rot("x'");
      rot("y'");
   } else if (rotation.equals("z'")) {
      rot("y'");
      rot("x'");
      rot("y");
   } else {
      if (rotation.contains("2")) {
         for (int i = 0; i < 2; ++i)
            rotate(rotation.substring(0, 1));
      } else {
         rot(rotation);
      }
   }
}
```

Performs the specified rotation on the cube

Parameters:

`rotation` - the rotation to be performed - must be "x" or "y" without "2"s

```
private void rot(String rotation) {
   // Stores the indices of the slices affected by the rotation.
   int[] sliceIndices = rotationSlices[((int) rotation.toCharArray()[0] - (int) 'x')];
   // Stores a Slice temporarily so that the cycle of Slices can be
   // completed.
   Slice tempSlice = slices[sliceIndices[0]];
   // Stores the index of the next slice to be changed.
   int nextIndex;
   // Stores the direction in which the slices should be cycled.
   int direction = (rotation.contains("'") ? -1 : 1);
   // Stores the index of the element in slicesIndices that stores the
   // index of the last slice that should be changed.
   int end = (4 + direction) % 4;

   /**
    * This cycles around the slices with the indices specified in
    * sliceIndices, e.g. <br>
    * 0 -> 5 <br>
    * 5 -> 1 <br>
    * 1 -> 4 <br>
    * 4 -> 0 <br>
    * would be the cycle order for an "x" rotation. <br>
    */
   for (int i = 0; i != end; i = (i - direction + 4) % 4) {
      nextIndex = (i - direction + 4) % 4;
      slices[sliceIndices[i]] = slices[sliceIndices[nextIndex]];
      performRotationMaintenance(rotation, sliceIndices[i]);
      updateCubies(sliceIndices[i]);
   }

   /**
    * This is the final step of the cycle. This is similar to declaring a
    * 'temp' variable when performing a swap, e.g. <br>
```

```
     * temp <- a <br>
     * a <- b <br>
     * b <- temp
     */
    slices[sliceIndices[end]] = tempSlice;
    performRotationMaintenance(rotation, sliceIndices[end]);
    updateCubies(sliceIndices[end]);

    /*
     * This performs the necessary operations to ensure that the
     * corners/edges are twisted/flipped correctly after a particular
     * orientation.
     */
    switch (rotation) {
    case "x'":
    case "x":
       slices[0].twistAllCorners(1);
       updateCubies(0);
       slices[1].twistAllCorners(-1);
       updateCubies(1);
       break;
    case "y":
    case "y'":
       for (int i = 0; i < 4; ++i) {
          slices[sliceIndices[i]].flipEdge(1);
          updateCubies(sliceIndices[i]);
       }

       break;
    default:
       break;
    }

    if (rotation.equals("x'")) {
       slices[0].performMove(2);
       updateCubies(0);
       slices[1].performMove(2);
       updateCubies(1);
    }
```

```
    /*
     * Update all cubies in each slice to the cubies in this instance of
     * Cube.
     */
    updateAll();
}
```

Cubie – Comparing two cubies

- See sections 3.05 - 3.10 and 5.05 of the testing section.
- Go to page 108 of the appendix section to see Cubie class.

Specified by:

`compareTo` in interface `java.lang.Comparable<Cubie>`

Parameters:

`otherCubie` - the cubie to which this cubie is compared

Returns:

0 if this cubie represents the same cubie as **otherCubie**;
-1 otherwise

```
public int compareTo(Cubie otherCubie) {
   // Stores true if the current sticker of the current cubie is found on
   // otherCubie; false otherwise.
   boolean found;

   /*
    * i.e. if this instance is a corner, and the other instance is an edge,
    * then they are obviously not the same.
    */
   if (otherCubie.getStickers().length != this.stickers.length)
      return -1;

   for (int i = 0; i < stickers.length; ++i) {
      found = false;
      for (int j = 0; j < stickers.length; ++j) {
         /*
          * Each element of this object's stickers should be in the other
          * object's stickers. If the element is not found in the other
          * array, then the method will return false.
          */
         if (this.getStickers()[i].equals(otherCubie.getStickers()[j])) {
```

```
            found = true;
            break;
         }
      }
      /*
       * This means the one of the elements of this.stickers has not been
       * found in otherCubie.stickers, so their stickers are not the same
       */
      if (!found)
         return -1;
   }
   return 0;
}
```

Cubie – Comparing two cubies in a stricter manner

- See sections 3.05 - 3.10 and 5.05 of the testing section.
- Go to page 108 of the appendix section to see Cubie class.

Parameters:

`otherCubie` - the cubie to which this cubie is compared

Returns:

0 if the cubies are the same and the stickers are in the same order (but not necessarily positions, e.g. the red-green-white corner will be shown to be the same as the green-white-red corner;
-1 otherwise

```
public int strictCompareTo(Cubie otherCubie) {
   int stickersLength = this.stickers.length;

   if (stickersLength != otherCubie.getStickers().length)
      return -1;

   int i = 0;

   /*
    * Find the index of a matching sticker
    */
   while ((i < stickersLength) && (!otherCubie.getStickers()[0].equals(this.getStickers()[i]))) {
      ++i;
   }

   /*
    * If no matching sticker is found then i == stickersLength, so return -1
    */
   if (i == stickersLength) {
      return -1;
   }

   /*
    * Compare each element until will the expected offset. If any elements differ, then return -1.
```

```
     */
    for (int j = i + 1; j < i + stickersLength; ++j) {
       if (!stickers[j % stickersLength].equals(otherCubie.getStickers()[(j-i+stickersLength) % stickersLength])) {
          return -1;
       }
    }

    /*
     * No contradictions have been found, so return 0
     */
    return 0;
}
```

EdgeSolver – Solve middle-layer edges

- See sections 3.05 - 3.10 of the testing section.
- Go to page 122 of the appendix section to see EdgeSolver class.

Solves the edges in the middle layer, and records the solution and explanation at the same time.

```
public void solveMiddleLayerEdges() {
   // Stores the properties of the edge to be solved.
   Edge edge;
   rotateToTop(Color.yellow);
   // Stores the index of the unsolved corner being examined.
   int solveCandidatesIndex;
   // Stores the index of the corner to solve.
   int edgeIndex;
   // Stores the stickers of the edge so that a better explanation can be
   // generated.
   Color[] stickers;

   solveCandidates = new SolveCandidate[4];
   for (int i = 0; i < 4; ++i)
      // initialise solveCandidates
      solveCandidates[i] = new SolveCandidate();

   while (!middleLayerEdgesSolved()) {
      solveCandidatesIndex = 0;

      for (int i = 0; i < 8; ++i) {
         edge = cube.getEdge(i);
         if (isMLEdge(edge) && (!isPieceSolved(edge))) {
            solveCandidates[solveCandidatesIndex].index = i;
            solveCandidates[solveCandidatesIndex].score = getScore(i);
            ++solveCandidatesIndex;
         }
      }

      edgeIndex = solveCandidates[getIndexOfMinScore(solveCandidatesIndex)].index;
```

```
      stickers = cube.getEdge(edgeIndex).getStickers();
      solutionExplanation += String.format("Edges - %s-%s edge:\n", Cubie.getColorToWord(stickers[0]),
            Cubie.getColorToWord(stickers[1]));

      solveEdge(edgeIndex);
   }

   try {
      // Remove last newline character
      solutionExplanation = solutionExplanation.substring(0, solutionExplanation.lastIndexOf("\n"));
   } catch (IndexOutOfBoundsException e) {
   }
}
```

Solves the Edge at the specified index, and record the solution and explanation at the same time.

Parameters:

`currentIndex` - the index of the Edge to be solved.

```
public void solveEdge(int currentIndex) {
   rotateToTop(Color.yellow);
   // Stores a copy of the edge to be solved.
   Edge edge = new Edge(cube.getEdge(currentIndex).getStickers());

   if (isPieceSolved(edge))
      return;

   if /* edge is in top layer */(currentIndex < 4) {
      while (!edgeInSetupPosition(edge)) {
         cube.performAbsoluteMoves("U");
         catalogMoves("U");
      }

      while (cube.getEdge(2).compareTo(edge) == -1) {
         cube.rotate("y");
         catalogMoves("y");
      }
```

```
      if (edge.getStickers()[0].equals(cube.getSlice(2).getCentre())) {
         cube.performAbsoluteMoves("U R U' R' U' F' U F");
         catalogMoves("U R U' R' U' F' U F");
         solutionExplanation += "The edge needs to go to the right, so set up the edge then perform  R U' R' U' F' U
F\n\n";
      } else {
         cube.performAbsoluteMoves("U' L' U L U F U' F'");
         catalogMoves("U' L' U L U F U' F'");
         solutionExplanation += "The edge needs to go to the left, so set up the edge then perform  L' U L U F U'
F'\n\n";
      }
   } else /* edge is in E slice/middle layer */{
      // Rotate the cube until the edge is at FR
      while (cube.getEdge(6).compareTo(edge) == -1) {
         cube.rotate("y");
         catalogMoves("y");
      }

      cube.performAbsoluteMoves("R U' R' U' F' U F");
      catalogMoves("R U' R' U' F' U F");
      solutionExplanation += "The edge is trapped in the middle layer, so remove it using  R U' R' U' F' U F.\n";

      // The edge is now at UB, so solve the edge at this index
      solveEdge(0);
   }
}
```

EdgeSolver – Determine how many moves to solve a piece

- See sections 3.05 - 3.10 of the testing section.
- Go to page 122 of the appendix section to see EdgeSolver class.

This determines how many moves are required to solve the edge at the specified index.

Parameters:

`index` - the index of Edge to be analysed

Returns:

the number of moves required to solve the Edge at the specified **index**

```
private int getScore(int index) {
   /*
    * This determines the number of moves to solve the edge at the
    * specified index by solving the piece, counting the moves, then
    * reverses the moves so that the cube is in its initial state before
    * the method was invoked.
    */

   // Stores the number of moves to solve the edge.
   int score;

   EdgeSolver es = new EdgeSolver(cube);
   es.solveEdge(index);
   simplifyMoves(es.getCatalogMoves(), SolveMaster.CORNER_EDGE);
   score = es.getCatalogMoves().size();
   cube.performAbsoluteMoves(getReverseStringMoves(es.getCatalogMoves()));
   es.clearMoves();

   return score;
}
```

Main – Generate dataset for time graph

- See section 4.13 of the testing section. The correct times are shown in the graph, and DNF (-1) times are not added.
- Go to page 132 of the appendix section to see Main class.

Returns the dataset to be used by `timeGraph`

Returns:

the dataset of the past `numTimesToGraph` times

```
private static DefaultCategoryDataset getDatasetOfSelectedTimes() {
   // Stores the data contents of the graph
   DefaultCategoryDataset dataset = new DefaultCategoryDataset();
   // Stores the index of the first element in 'solves' to be graphed.
   int start = solves.size() - numTimesToGraph;
   // Stores the number of DNF times in the past 'numTimesToGraph' times.
   int numDNF = Statistics.getNumDNF(numTimesToGraph, solves);
   int end = solves.size();
   // Stores the numeric representation of the current time be examined.
   double current;
   // Accumulates the number of times graphed.
   int numDataValues = 0;
   // Stores the index of a valid time in case there is only one valid time
   // to be graphed.
   int indexOfValidTime = 0;

   /*
    * DNF times are not graphed, so this gets a start index that will allow
    * the maximum numbers of times (<= numTimesToGraph) to be graphed.
    */
   if (start > 0) {
      /*
       * The start index must be greater than or equal to zero, so exit
       * when 'start' will become -1. When numDNF = 0, then this block of
       * code has completed its purpose.
       */
```

```
      while ((start > 0) && (numDNF > 0)) {
         --start;
         if (solves.get(start).getNumericTime() != -1)
            --numDNF;
      }
   }
   /*
    * This will execute when the number of solves is less than the number
    * of times to graph, but if this is the case, then start must be reset
    * to 0. There will be fewer than numTimesToGraph times graphed.
    */
   else
      start = 0;

   for (int i = start; i < end; ++i) {
      current = solves.get(i).getNumericTime();

      if (current != -1) {
         dataset.addValue(solves.get(i).getNumericTime(), "times", "" + Integer.toString(i + 1));
         ++numDataValues;
         indexOfValidTime = i;
      }
   }

   /*
    * If the number of dataset values added is 1, then the graph will
    * consist of an invisible point. This if statement adds an identical
    * value to the graph so that instead of a point, it shows a straight
    * line.
    */
   if (numDataValues == 1)
      dataset.addValue(solves.get(indexOfValidTime).getNumericTime(), "times", "");

   return dataset;
}
```

Main – Determine whether the cube is in a valid state

- See section 4.04 and 5.05 of the testing section. Only permutations of the pieces that can be achieved on a standard Rubik's cube are accepted, as expected.
- Go to page 132 of the appendix section to see Main class.

Determines whether or not the cube is in a valid state

Parameters:

`clearStickers` - if **true**, then when an invalid piece is found, its stickers will be cleared, i.e. turn grey;
if **false** then the stickers will remain as they are

Returns:

true if the cube is in a valid state;
false otherwise

```
public static boolean isValidCubeState(boolean clearStickers) {
   // Stores all valid corner sticker options.
   Color[][] validCornerStickers = Corner.getAllInitialStickers();
   // Stores all valid edge sticker options.
   Color[][] validEdgeStickers = Edge.getAllInitialStickers();
   // Stores all valid corners on a standard Rubik's cube.
   Corner[] validCorners = new Corner[8];
   // Stores all valid edges on a standard Rubik's cube.
   Edge[] validEdges = new Edge[12];
   boolean isValidCubeState = true;
   // Indicates whether the ith corner has already been found. This can be
   // used to check for duplicate pieces.
   boolean[] cornersFound = new boolean[8];
   // Indicates whether the ith edge has already been found. This can be
   // used to check for duplicate pieces.
   boolean[] edgesFound = new boolean[12];
   int validCubieIndex;
   String originalState = solveMaster.getStateString();

   for (int i = 0; i < 8; ++i) {
```

```
      validCorners[i] = new Corner(validCornerStickers[i]);
   }

   for (int i = 0; i < 12; ++i) {
      validEdges[i] = new Edge(validEdgeStickers[i]);
   }

   for (int i = 0; i < 8; ++i) { // These two for loops check if every
                           // cubie has valid stickers.
      validCubieIndex = getValidCubieIndex(cube.getCorner(i), validCorners);

      /*
       * If valueCubieIndex = -1, then the current corner does not exist
       * on a standard cube, so set isValidCubeState to false. If
       * cornersFound[validCubieIndex] is already true then this corner is
       * a duplicate, so set isValidCubeState to false.
       */
      if ((validCubieIndex == -1) || (cornersFound[validCubieIndex])) {
         isValidCubeState = false;
         if (clearStickers)
            cube.getCorner(i).setStickers(Color.LIGHT_GRAY, Color.LIGHT_GRAY, Color.LIGHT_GRAY);
      } else
         cornersFound[validCubieIndex] = true;
   }
   for (int i = 0; i < 12; ++i) {
      validCubieIndex = getValidCubieIndex(cube.getEdge(i), validEdges);

      if ((validCubieIndex == -1) || (edgesFound[validCubieIndex])) {
         isValidCubeState = false;

         if (clearStickers)
            cube.getEdge(i).setStickers(Color.LIGHT_GRAY, Color.LIGHT_GRAY);
      } else
         edgesFound[validCubieIndex] = true;
   }

   if (!isValidCubeState)
      return false;
```

```
   // Start solving cube
   solveMaster.rotateToTop(Color.white);
   assignOrientationsToCubies();

   crossSolver.solveCross();
   cornerSolver.solveFirstLayerCorners();
   edgeSolver.solveMiddleLayerEdges();
   orientationSolver.solveOrientation();
   permutationSolver.solvePermutation();
   // Stop solving cube

   /*
    * This block checks if any pieces are unsolved after the solve masters
    * have generated a solution.
    */
   for (int i = 0; i < 4; ++i) {
      if (!crossSolver.isPieceSolved(cube.getCorner(i))) {
         isValidCubeState = false;

         if (clearStickers)
            cube.getCorner(i).setStickers(Color.LIGHT_GRAY, Color.LIGHT_GRAY, Color.LIGHT_GRAY);
      }
      if (!crossSolver.isPieceSolved(cube.getEdge(i))) {
         isValidCubeState = false;

         if (clearStickers)
            cube.getEdge(i).setStickers(Color.LIGHT_GRAY, Color.LIGHT_GRAY);
      }
   }

   solveMaster.clearMoves();
   SolveMaster.simplifyMoves(crossSolver.getCatalogMoves(), SolveMaster.CROSS);
   SolveMaster.simplifyMoves(cornerSolver.getCatalogMoves(), SolveMaster.CORNER_EDGE);
   SolveMaster.simplifyMoves(edgeSolver.getCatalogMoves(), SolveMaster.CORNER_EDGE);
   SolveMaster.simplifyMoves(orientationSolver.getCatalogMoves(), SolveMaster.CANCELLATIONS);
   SolveMaster.simplifyMoves(permutationSolver.getCatalogMoves(), SolveMaster.CANCELLATIONS);

   solveMaster.applyStateString(originalState);
```

```
    return isValidCubeState;
}
```

Main – Assign orientation values to cubies

- See section 4.04 of the testing section. The state entered by the user is extracted correctly so that the normal operations of the program, e.g. generating solutions, performing solves etc., can be performed.
- Go to page 132 of the appendix section to see Main class.

Assigns the corresponding orientation to each cubie on the cube. This method is used to find the orientation of each piece after, e.g. painting a custom state or loading a state from file

```
public static void assignOrientationsToCubies() {
   // Stores the original colours of the centres on top and front.
   Color[] originalTopFront = { cube.getSlice(0).getCentre(), cube.getSlice(4).getCentre() };
   solveMaster.rotateToTopFront(Color.white, Color.green);

   // Stores the colour of the current centre at the top
   Color topCentre = cube.getSlice(0).getCentre();
   // Stores the colour of the current centre at the bottom
   Color bottomCentre = cube.getSlice(1).getCentre();
   // Stores the colour of the current centre at the right
   Color rightCentre = cube.getSlice(2).getCentre();
   // Stores the colour of the current centre at the left
   Color leftCentre = cube.getSlice(3).getCentre();

   // Stores the orientation of the current cubie.
   int orientation;
   // Stores the properties of the current corner being examined.
   Corner currentCorner;
   // Stores the properties of the current edge being examined.
   Edge currentEdge;
   // Stores the stickers on the current edge being examined.
   Color[] edgeStickers;

   for (int i = 0; i < 8; ++i) {
      currentCorner = cube.getCorner(i);

      /*
       * Finds the index of the top or bottom centre on the corner so that
       * the orientation can be determined.
```

```
        */
      if ((orientation = LinearSearch.linearSearchCornerOrientation(currentCorner.getStickers(), topCentre)) == -2)
         currentCorner.setOrientation(LinearSearch.linearSearchCornerOrientation(currentCorner.getStickers(),
               bottomCentre));
      else
         currentCorner.setOrientation(orientation);
   }

   for (int i = 0; i < 12; ++i) {
      currentEdge = cube.getEdge(i);
      edgeStickers = currentEdge.getStickers();

      if ((edgeStickers[0].equals(rightCentre)) || (edgeStickers[0].equals(leftCentre))
            || (edgeStickers[1].equals(topCentre)) || (edgeStickers[1].equals(bottomCentre)))
         currentEdge.setOrientation(1);
      else
         currentEdge.setOrientation(0);
   }

   solveMaster.rotateToTopFront(originalTopFront[0], originalTopFront[1]);
}
```

Main – After releasing spacebar

- See section 5.09 of the testing section. If the situation permits it, the visual timer's value starts to increment and the user can time the current task.
- Go to page 132 of the appendix section to see Main class.

```
public void keyReleased(KeyEvent e) {
   /*
    * This method stops timers and records times, so if a tutorial is
    * running or custom painting is in progress, then nothing should
    * happen, so return.
    */
   if (tutorialIsRunning || customPaintingInProgress)
      return;

   if (e.getKeyChar() == ' ') {
      /*
       * This starts the timer when the inspection timer is running
       */
      if (timerHasPermissionToStart && (inspectionTimer != null)) {
         inspectionTimer.stop();
         inspectionTimer = null;
         timerHasPermissionToStart = false;
         timeToBeRecorded = true;
         movesAllowed = true;
         timeLabel.setForeground(Color.black);
         elapsedTimingSeconds = 0.00F;
         elapsedMinutes = 0;
         incTimer = new Timer((int) (timingDisplayInterval * 1000.0), new TimerListener());
         incTimer.start();

         int size = solves.size();
         if (size > 0) {
            listHolder.setSelectedIndex(size - 1);
            listHolder.ensureIndexIsVisible(size - 1);
         }
      } else if ((inspectionTimer == null) && (realTimeSolutionTimer == null) && (incTimer == null)
            && (movesAllowed) && (!tutorialIsRunning)) {
         // Start timer with spacebar (no inspection)
```

```
         customTimerRunning = true;
         timerHasPermissionToStart = false;
         timeToBeRecorded = false;
         movesToBeRecorded = false;
         movesAllowed = true;
         timeLabel.setForeground(Color.black);
         elapsedTimingSeconds = 0.00F;
         elapsedMinutes = 0;
         incTimer = new Timer((int) (timingDisplayInterval * 1000.0), new TimerListener());
         incTimer.start();
      } else if ((!timeToBeRecorded) && (incTimer != null) && (incTimer.isRunning())) {
         // Stop timer with spacebar
         customTimerRunning = false;
         incTimer.stop();
         incTimer = null;
         timerHasPermissionToStart = true;

         int previousSelectedIndex = listHolder.getSelectedIndex();
         SolveMaster.simplifyMoves(trackingMoves, SolveMaster.CANCELLATIONS);

         Solve currentTime = new Solve(timeLabel.getText(), "0", "");
         solves.add(currentTime);
         timeDisplayList.addElement(timeLabel.getText());

         // Makes sure the appropriate element in the list is selected.
         if (previousSelectedIndex == solves.size() - 2)
            listHolder.setSelectedIndex(previousSelectedIndex + 1);

         listHolder.ensureIndexIsVisible(listHolder.getSelectedIndex());

         refreshTimeGraph(true);
         refreshStatistics();
      }
   } else if (cubeSolved && timeToBeRecorded) {
      endSolve();
   }
}
```

Main – After typing a key

- See section 5.01 of the testing section.
- Go to page 132 of the appendix section to see Main class.

This method handles the release of spacebar or the solving of the cube after a key is released.

```
public void keyTyped(KeyEvent e) {
   if (e.isAltDown() || (e.getKeyChar() == ' '))
      return;

   // Stores the move in the correct notation, e.g. instead of "j", it
   // stores "U"
   String move = SolveMaster.getKeyToMove("" + e.getKeyChar());

   if (SolveMaster.isValidMove(move)) {
      if (movesToBeRecorded) {
         if (inspectionTimer != null) {
            if ("xyz".contains(move.substring(0, 1))) {
               trackingMoves.add(move);
            }
         } else {
            trackingMoves.add(move);
         }
      }

      if (!movesAllowed) {
         if (realTimeSolutionTimer == null) {
            cube.rotate(move);
            repaint();
            return;
         }
      } else if (!customPaintingInProgress) {
         cube.performAbsoluteMoves(move);
      }
   }

   if ((!tutorialIsRunning) && (!customPaintingInProgress) && (!customTimerRunning) && (incTimer != null)
```

```
         && (incTimer.isRunning()) && (isCubeSolved())) {
      // Stops timer when cube is solved
      incTimer.stop();
      incTimer = null;
      movesToBeRecorded = false;
      timerHasPermissionToStart = false;
      cubeSolved = true;
   }

   repaint();

   if (tutorialIsRunning) { // Check if criteria is filled
      if (tutorial.criteriaFilled()) {
         // Stores the text to be shown in the dialog box
         String dialogText = "";

         SolveMaster.simplifyMoves(trackingMoves, SolveMaster.CANCELLATIONS);

         if (Tutorial.getNumMovesWithoutRotations(trackingMoves) <= tutorial.getOptimalSolutionLength()) {
            dialogText = "Well done!" + "\nWould you like to play again?";
         } else {
            dialogText = "You solved the problem, but you could have done it in fewer moves."
                  + "\nWould you like to try again?";
         }

         trackingMoves.clear();

         Object[] options = { "Yes", "No" };
         // Stores 0 if the user selects 'Yes'.
         int choice = JOptionPane.showOptionDialog(null, dialogText, "Congratulations",
               JOptionPane.YES_NO_OPTION, JOptionPane.QUESTION_MESSAGE, null, options, options[1]);

         if (choice == 0) // User wants to play again
            tutorialResetStateButton.doClick();
         else { // User wants to move on OR user closes dialog
            tutorialShowSolutionButton.doClick();
            movesAllowed = false;
            movesToBeRecorded = false;
         }
```

```
        }
    }
}
```

Main – Assign painting colours to stickers/facelets

– Go to page 132 of the appendix section to see Main class.

Assigns the sticker colours to the appropriate elements of `faceColors`

```
private void assignFaceletPaintingColors() {
   /*
    * This specifies that the top face should be painted first, then the
    * front face, then the right face.
    */
   int[] sliceIndices = { 0, 4, 2 };
   // Stores the properties of the current slice be examined.
   Slice currentSlice;
   // Stores the index of the current corner being examined on the current
   // slice.
   int cornerIndex;
   // Stores the index of the current corner being examined on the current
   // slice.
   int edgeIndex;

   currentSlice = cube.getSlice(sliceIndices[0]);
   cornerIndex = 0;
   edgeIndex = 0;
   // Iterates over each row of the current face
   for (int i = 0; i < 3; ++i) {
      // Iterates over each column of the current face
      for (int j = 0; j < 3; ++j) {
         if ((i == 1) && (j == 1))
            faceletColors[0][i][j] = currentSlice.getCentre();
         else if ((i + j) % 2 == 0)
            faceletColors[0][i][j] = currentSlice.getCorner(cornerPaintOrder[cornerIndex++]).getStickers()[0];
         else
            faceletColors[0][i][j] = currentSlice.getEdge(edgePaintOrder[edgeIndex++]).getStickers()[0];
      }
   }

   currentSlice = cube.getSlice(sliceIndices[1]);
```

```
   cornerIndex = 0;
   edgeIndex = 0;
   for (int i = 0; i < 3; ++i) {
      for (int j = 0; j < 3; ++j) {
         if ((i == 1) && (j == 1))
            faceletColors[1][i][j] = currentSlice.getCentre();
         else if ((i + j) % 2 == 0)
            /*
             * i.e. if it is the FLU or FRD corner, then get the second
             * sticker, and if it is the FUR or FDL corner, then get the
             * third sticker
             */
            faceletColors[1][i][j] =
currentSlice.getCorner(cornerPaintOrder[cornerIndex]).getStickers()[(cornerPaintOrder[cornerIndex++] % 2 == 0) ? 1
                  : 2];
         else
            /*
             * i.e. if it is the FU or FD edge, then get the second
             * sticker, otherwise get the first sticker.
             */
            faceletColors[1][i][j] =
currentSlice.getEdge(edgePaintOrder[edgeIndex]).getStickers()[(edgePaintOrder[edgeIndex++] % 2 == 0) ? 1
                  : 0];
      }
   }

   currentSlice = cube.getSlice(sliceIndices[2]);
   cornerIndex = 0;
   edgeIndex = 0;
   for (int i = 0; i < 3; ++i) {
      for (int j = 0; j < 3; ++j) {
         if ((i == 1) && (j == 1))
            faceletColors[2][i][j] = currentSlice.getCentre();
         else if ((i + j) % 2 == 0)
            faceletColors[2][i][j] =
currentSlice.getCorner(cornerPaintOrder[cornerIndex]).getStickers()[(cornerPaintOrder[cornerIndex++] % 2 == 0) ? 1
                  : 2];
         else
            faceletColors[2][i][j] = currentSlice.getEdge(edgePaintOrder[edgeIndex++]).getStickers()[1];
```

```
        }
    }
}
```

Main – Create the mouse listener for the cube

- See section 5.08 of the testing section.
- Go to page 132 of the appendix section to see Main class .

Returns:

a MouseListener for the cube panel so that the clicked piece can be solved when in Click-to-Solve mode

```
private static MouseListener getCubePanelMouseListener() {
   MouseListener cubePanelMouseListener = new MouseListener() {
      @Override
      public void mouseClicked(MouseEvent arg0) {}

      public void mouseReleased(MouseEvent e) {
         cubePanel.requestFocus();

         if (clickToSolve) {
            // Stores the index of the piece selected.
            int index = MouseSelectionSolver.getIndexOfPieceOnScreen(e.getX(), e.getY());
            // Stores an integer representing the choice made by the
            // user in the question dialog.
            int choice = -1;
            // Stores the colours of the stickers of the cubie selected.
            Color[] stickers;
            String solution = "";

            if (index < 0)
               return;

            MenuBar.setSolvePieceSelected(false);
            clickToSolve = false;

            if (index >= 8)
               stickers = cube.getEdge(index - 8).getStickers();
            else
               stickers = cube.getCorner(index).getStickers();
```

```
            colorSelection.setAlwaysOnTop(false);
            choice = MouseSelectionSolver.getQuestionDialogResponse(String.format(
                  "You have selected the %s-%s%s. Do you wish to continue?",
                  Cubie.getColorToWord(stickers[0]), Cubie.getColorToWord(stickers[1]),
                  (stickers.length == 3) ? "-" + Cubie.getColorToWord(stickers[2]) + " Corner" : " Edge"));
            colorSelection.setAlwaysOnTop(true);

            if (choice == 0) {
               selectionSolver.solvePiece(index);
            } else
               return;

            solution = selectionSolver.getSolution();

            timingDetailsTextArea.setText(String.format("Solving the %s-%s%s",
                  Cubie.getColorToWord(stickers[0]), Cubie.getColorToWord(stickers[1]),
                  (stickers.length == 3) ? "-" + Cubie.getColorToWord(stickers[2]) + " Corner" : " Edge"));

            if (!solution.equals(MouseSelectionSolver.BLANK)) {
               timingDetailsTextArea.setText(solution);
               timingDetailsTextArea.setCaretPosition(0);
            }

            if (MenuBar.paintCustomStateItem.isSelected())
               MenuBar.paintCustomStateItem.doClick();

            performRealTimeSolving();
            cubePanel.repaint();
         } else if (customPaintingInProgress && (realTimeSolutionTimer == null)) {
            // Stores the index of the facelet selected on screen.
            int index = MouseSelectionSolver.getIndexOfFaceletOnScreen(e.getX(), e.getY());
            // Stores the current colour which will be painted on the
            // selected sticker.
            Color currentCustomPaintingColor = colorSelection.getSelectedColor();

            if (index < 0)
               return;

            switch (index) {
```

```
            case 0:
               cube.getCorner(0).setSticker(0, currentCustomPaintingColor);
               break;
            case 1:
               cube.getEdge(0).setSticker(0, currentCustomPaintingColor);
               break;
            case 2:
               cube.getCorner(1).setSticker(0, currentCustomPaintingColor);
               break;
            case 3:
               cube.getEdge(3).setSticker(0, currentCustomPaintingColor);
               break;
            case 5:
               cube.getEdge(1).setSticker(0, currentCustomPaintingColor);
               break;
            case 6:
               cube.getCorner(3).setSticker(0, currentCustomPaintingColor);
               break;
            case 7:
               cube.getEdge(2).setSticker(0, currentCustomPaintingColor);
               break;
            case 8:
               cube.getCorner(2).setSticker(0, currentCustomPaintingColor);
               break;

            case 9:
               cube.getCorner(3).setSticker(1, currentCustomPaintingColor);
               break;
            case 10:
               cube.getEdge(2).setSticker(1, currentCustomPaintingColor);
               break;
            case 11:
               cube.getCorner(2).setSticker(2, currentCustomPaintingColor);
               break;
            case 12:
               cube.getEdge(7).setSticker(0, currentCustomPaintingColor);
               break;
            case 14:
               cube.getEdge(6).setSticker(0, currentCustomPaintingColor);
```

```
            break;
         case 15:
            cube.getCorner(6).setSticker(2, currentCustomPaintingColor);
            break;
         case 16:
            cube.getEdge(10).setSticker(1, currentCustomPaintingColor);
            break;
         case 17:
            cube.getCorner(7).setSticker(1, currentCustomPaintingColor);
            break;

         case 18:
            cube.getCorner(2).setSticker(1, currentCustomPaintingColor);
            break;
         case 19:
            cube.getEdge(1).setSticker(1, currentCustomPaintingColor);
            break;
         case 20:
            cube.getCorner(1).setSticker(2, currentCustomPaintingColor);
            break;
         case 21:
            cube.getEdge(6).setSticker(1, currentCustomPaintingColor);
            break;
         case 23:
            cube.getEdge(5).setSticker(1, currentCustomPaintingColor);
            break;
         case 24:
            cube.getCorner(7).setSticker(2, currentCustomPaintingColor);
            break;
         case 25:
            cube.getEdge(11).setSticker(1, currentCustomPaintingColor);
            break;
         case 26:
            cube.getCorner(4).setSticker(1, currentCustomPaintingColor);
            break;
         }

         cubePanel.repaint();
      }
```

```
        }

        @Override
        public void mouseEntered(MouseEvent arg0) {}
        @Override
        public void mouseExited(MouseEvent arg0) {}
        @Override
        public void mousePressed(MouseEvent arg0) {}
    };

    return cubePanelMouseListener;
}
```

MemberCompetition – Compare two ‘average of 5’ records

- See sections 3.11-3.14 of testing section. This correctly identifies the better of the two averages. If the averages are the exact same, then this returns false because it results in fewer comparisons being made by the calling sorting algorithm.
- Go to page 223 of the appendix section to see MemberCompetition class.

This method compares **this** average to another average. An average is better than another average if the 'average of 5' is faster, or the fastest time of **this** is faster than the fastest time of **other**. If the fastest times are the same, then the second fastest times are compared in the same way. If all times are the same, then **other** will be assumed to be the better average

Parameters:

`other` - the other MemberCompetition to which this MemberCompetition is compared

Returns:

true if **this** is better than **other**;
false otherwise

```
public boolean isBetterThan(MemberCompetition other) {
   /*
    * Stores the numerical representation of the average. For example, if
    * the times of this average were 12.00, 13.00, 14.00, 15.00, 16.00,
    * then 'thisAverage' would store 14.0
    */
   double thisAverage = get2DPAverage();
   double otherAverage = other.get2DPAverage();
   // If one of the average is DNF, then set it to infinity
   if (thisAverage == -1)
      thisAverage = 1e10;
   if (otherAverage == -1)
      otherAverage = 1e10;

   if (thisAverage < otherAverage)
      return true;
   else if (otherAverage < thisAverage)
      return false;
```

```
   /*
    * Stores the times of the average in their numerical representation.
    * For example, if the times were {12.00, 1:10.50, 55.23, 2:00.95,
    * 1:46.58} then the list would store {12.0, 70.5, 55.23, 120.95, 106.58}
    */
   double[] list1 = this.getNumericTimeArray();
   double[] list2 = other.getNumericTimeArray();

   // If a time is DNF, then set it to infinity
   for (int i = 0; i < 5; ++i) {
      if (list1[i] == -1)
         list1[i] = 1e10;
      if (list2[i] == -1)
         list2[i] = 1e10;
   }

   Sorter.quickSort(list1);
   Sorter.quickSort(list2);

   /*
    * i.e. if the fastest time of the first average is better than the
    * second average, then return true, or vice-versa return false. If the
    * fastest time of each is the same, the second times are compared etc.
    * until they are different.
    */
   for (int i = 0; i < 5; ++i) {
      if (list1[i] < list2[i])
         return true;
      else if (list2[i] < list1[i])
         return false;
   }

   /*
    * If this point is reached, then the averages are exactly the same, so
    * just return false.
    */
   return false;
}
```

OrientationSolver – Solve corner orientation

- See sections 3.05 - 3.10 of the testing section.
- Go to page 288 of the appendix section to see OrientationSolver class.

Performs the moves required to solve the corner orientation of **cube**

```
private void solveCornerOrientation() {
   // Stores the orientation of the current corner being examined.
   int orientation = 0;
   // Stores the number of trailing U moves performed.
   int numTrailing;
   // Stores the moves to be performed after inspection of the corner.
   String moves;

   for (int i = 0; i < 4; ++i) {
      moves = "";
      orientation = cube.getCorner(2).getOrientation();
      if (orientation == 1) {
         moves = "R' D' R D R' D' R D";
         solutionExplanation += String.format(
               "Bring the %s-%s-%s corner to the URF position then twist it anti-clockwise using %s%n",
               Cubie.getColorToWord(cube.getCorner(2).getStickers()[0]),
               Cubie.getColorToWord(cube.getCorner(2).getStickers()[1]),
               Cubie.getColorToWord(cube.getCorner(2).getStickers()[2]), "R' D' R D R' D' R D");

      } else if (orientation == -1) {
         moves = ("D' R' D R D' R' D R");
         solutionExplanation += String.format(
               "Bring the %s-%s-%s corner to the URF position then twist it clockwise using %s%n",
               Cubie.getColorToWord(cube.getCorner(2).getStickers()[0]),
               Cubie.getColorToWord(cube.getCorner(2).getStickers()[1]),
               Cubie.getColorToWord(cube.getCorner(2).getStickers()[2]), "D' R' D R D' R' D R");
      }

      cube.performAbsoluteMoves(moves);
      cube.performAbsoluteMoves("U");
      catalogMoves(moves + " U");
```

```
    }

    /*
     * Undo unnecessary U moves
     */
    numTrailing = getNumTrailingU();
    for (int i = 0; i < numTrailing; ++i) {
       cube.performAbsoluteMoves("U'");
    }
}
```

PermutationSolver – Solve edge permutation

- See sections 3.05 - 3.10 of the testing section.
- Go to page 293 of the appendix section to see Permutation class.

Performs and records the moves required to solve the edge permutation of **cube**

```
private void solveEdgePermutation() {
   // Stores the number of U moves performed so that no more than 3 are
   // perfomed during the initial inspection.
   int count = 0;
   // Stores the moves to be performed after inspection of the state.
   String moves = "";

   if (isEdgePermutationSolved()) {
      return;
   }

   // Get solid block on back
   while ((count < 4) && (!cube.getEdge(0).getSecondaryColor().equals(cube.getCorner(0).getStickers()[2]))) {
      ++count;
      cube.performAbsoluteMoves("U");
      catalogMoves("U");
   }

   if (count < 4) { // if block on back
      solutionExplanation += String.format("Bring the %s block to the back and then cycle the pieces ",
            Cubie.getColorToWord(cube.getEdge(0).getSecondaryColor()));

      if (edgesAreOpposite(cube.getEdge(1), cube.getEdge(2))) {
         solutionExplanation += String.format("anti-clockwise using  R U' R U R U R U' R' U' R2");
         moves = ("R U' R U R U R U' R' U' R2");
      } else {
         solutionExplanation += String.format("clockwise using  R2 U R U R' U' R' U' R' U R'");
         moves = ("R2 U R U R' U' R' U' R' U R'");
      }
   } else if ((cube.getEdge(0).getSecondaryColor().equals(cube.getCorner(3).getStickers()[1]))
         && (cube.getEdge(1).getSecondaryColor().equals(cube.getCorner(3).getStickers()[2]))) {
```

```
      solutionExplanation += "All edges needed swapped vertically or horizontally, so we must perform  M2 U M2 U2 M2
U M2";
      moves = ("M2 U M2 U2 M2 U M2");
   } else {
      solutionExplanation += "The edges need swapped in a z formation, so ";
      if (!cube.getEdge(2).getSecondaryColor().equals(cube.getCorner(1).getStickers()[2])) {
         moves = "U ";
         solutionExplanation += "set them up by performing  U  then ";
      }

      solutionExplanation += "perform  M2 U M2 U M' U2 M2 U2 M'";
      moves += "M2 U M2 U M' U2 M2 U2 M'";
   }

   cube.performAbsoluteMoves(moves);
   catalogMoves(moves);
}
```

Slice – Perform a move

- See sections 3.05 - 3.10 of the testing section.
- Go to page 323 of the appendix section to see Slice class.

Performs a move on the slice in the specified direction
1 - Clockwise 2 - 180 degree -1 - Anticlockwise This method handles an argument of '2'

Parameters:

`direction` - the direction in which to move the slice

```
public void performMove(int direction) {
   int tempDirection = (direction == 2) ? 1 : direction;

   for (int i = 0; i < (int) Math.abs(direction); ++i)
      pMove(tempDirection);
}
```

Performs a move on the slice in the specified direction
1 - Clockwise -1 - Anticlockwise

Parameters:

`direction` - the direction in which to move the slice

```
private void pMove(int direction) {
   // Stores a copy of the stickers of the last edge in the swap-cycle.
   Color[] tempEdgeStickers = Arrays.copyOf(edges[0].getStickers(), 2);
   // Stores a copy of the stickers of the last corner in the swap-cycle.
   Color[] tempCornerStickers = Arrays.copyOf(corners[0].getStickers(), 3);
   // Stores a copy of the orientation of the last edge in the swap-cycle.
   int tempEOrientation = edges[0].getOrientation();
   // Stores a copy of the orientation of the last corner in the swap-cycle.
   int tempCOrientation = corners[0].getOrientation();
   // Stores index of the next cubie in the swap cycle.
```

```
    int nextIndex = 0, end = (4 + direction) % 4;

    // Cycles around the stickers in the specified direction.
    for (int i = 0; i != end; i = (i - direction + 4) % 4) {
        nextIndex = (i - direction + 4) % 4;
        edges[i].setStickers(edges[nextIndex].getStickers());
        edges[i].setOrientation(edges[nextIndex].getOrientation());
        corners[i].setStickers(corners[nextIndex].getStickers());
        corners[i].setOrientation(corners[nextIndex].getOrientation());
    }

    // Completes the cycle
    edges[end].setStickers(tempEdgeStickers);
    edges[end].setOrientation(tempEOrientation);
    corners[end].setStickers(tempCornerStickers);
    corners[end].setOrientation(tempCOrientation);
}
```

Solve – Time format check

- See section 2.01 of the testing section. Only strings in the allowed format are accepted. Currently, times greater than or equal to one hour are not allowed.
- Go to page 331 of the appendix section to see Solve class.

Determines whether the specified time is in a valid. All valid formats are: MM:SS.sss
MM:SS.
MM:S.sss
MM:S.
M:SS.sss
M:SS.
M:S.sss
M:S.
SS.sss
SS.
S.sss
S.
DNF

Parameters:

`time` - the time to be analysed

Returns:

true if the specified time is valid;
false otherwise

```
public static boolean isValidTime(String time) {
   if (time == null)
      return false;

   if (time.equals("DNF"))
      return true;
   else if (!time.matches("(\\d{1,2}:)?\\d{1,2}\\.\\d*"))
      return false;
   // Times cannot be greater than or equal to one hour
   else if (getFormattedStringToDouble(time) > 3599.59)
```

```
      return false;
   else
      return true;
}
```

Solve – Generate a padded time-string

- See section 2.01 of the testing section. Times must be stored in the correct format, but this doesn't mean that other *input* styles shouldn't be accepted. For example, "1:5." is not the desired format for the program – "1:05.00" is the desired format, but rejecting this input would mean it would be more difficult for the user to input data. So, "1:5." is to be accepted, and this algorithm converts it correctly to the desired format for the program (MM:SS.ss).
- Go to page 331 of the appendix section to see Solve class.

Pads a time-string with leading and trailing zeros where appropriate. The argument must be in the form X:M.ss or similar so that it can be padded correctly.

Parameters:

`time` - the time-string to be padded

Returns:

the padded time-string

```
public static String getPaddedTime(String time) {
   // Stores the resulting padded time-string to be returned.
   String paddedTime = "";
   // Stores the index of the ':' character in the 'time'.
   int indexOfColon = 0;
   // Stores the index of the '.' character in the 'time'.
   int indexOfPeriod = 0;
   // Iterates over each character in 'time'.
   int j = 0;

   if ((indexOfColon = time.indexOf(":")) != -1) {
      // Finds the index of the first non-zero character
      while (time.substring(j, j + 1).equals("0"))
         ++j;

      if (j < indexOfColon)
         paddedTime += time.substring(j, indexOfColon + 1);
   }

   indexOfPeriod = time.indexOf(".");
```

```
   if (indexOfColon != -1) {
      /*
       * Adds leading zeros as required after the colon
       */
      for (int i = 0; i < 3 - indexOfPeriod + indexOfColon; ++i) {
         paddedTime += "0";
      }
   }

   paddedTime += time.substring(indexOfColon + 1);

   /*
    * Adds trailing zeros so that there are two digits after decimal point
    */
   int end = 3 - time.length() + indexOfPeriod;
   for (int i = 0; i < end; ++i)
      paddedTime += "0";

   return paddedTime;
}
```

Solve – Extract the number of seconds from a formatted time-string

– Go to page 331 of the appendix section to see Solve class.

Returns the numerical value represented by the formatted time-string

Parameters:

`time` - the formatted time-string from which the numerical time is to be calculated

Returns:

the numerical value represented by **time**

```
public static double getFormattedStringToDouble(String time) {
   if (time.equalsIgnoreCase("DNF"))
      return -1;

   // Stores the the index of the ':' character in 'time'.
   int index = 0;
   // Stores the resulting numerical representation of 'time'.
   double result = 0;

   /*
    * If the formatted string contains a minute component, then multiply
    * this part by 60 to get the number of seconds.
    */
   if ((index = time.indexOf(":")) != -1)
      result += 60 * Integer.valueOf("0" + time.substring(0, index));

   result += Double.parseDouble("0" + time.substring(index + 1));

   return result;
}
```

SolveMaster – Simplify the solution for the cross

- See sections 3.05 - 3.10 of the testing section. All of the specified simplifications are performed. Improvements to the algorithm could lead to solutions that are better for speed rather than move-count.
- Go to page 378 of the appendix section to see SolveMaster class.

Simplifies the cross solution so that cancellations occur,
e.g. F F' F = F
and so that there are no rotations,
e.g. R2 U y2 R = R2 U L

Parameters:

`originalMoves` - the moves to be simplified

```
private static void simplifyCross(LinkedList<String> originalMoves) {
   // Stores the current move being examined.
   String current;

   /*
    * Starts at the penultimate element since last element will be a z
    * rotation.
    */
   for (int i = originalMoves.size() - 2; i >= 0; --i) {
      current = originalMoves.get(i);

      /*
       * If y rotation is found then, alter all elements after the
       * rotation so that the rotation is not required.
       */
      if (current.substring(0, 1).equals("y") && (current.length() <= 2)) {
         for (int j = i + 1; j < originalMoves.size(); ++j) {
            originalMoves.set(j, applyRotationToMove(current, originalMoves.get(j)));
         }

         originalMoves.remove(i);
      }
```

```
    }
}
```

Returns the move that performs the same action on the cube if the cube was not rotated before the move.

Parameters:

`rotation` - the rotation that would have been made.

`move` - the move that would have been made after the rotation

Returns:

the move that performs the same action as the specified move *without* a rotation

```
private static String applyRotationToMove(String rotation, String move) {
   // Stores the moves so that
   String[] movePairings = { "D", "U", "R", "L", "B", "F" };
   // Stores the offset used in later calculations for the index of the
   // element to return.
   int offset = (rotation.contains("'") ? 1 : 0);
   // Stores the index of the element which is equal to the first character of 'move'
   int index;

   if (!rotation.contains("2")) {
      switch (rotation.substring(0, 1)) {
      case "x":
         switch (move.substring(0, 1)) {
         case "U":
            return movePairings[5 - offset] + move.substring(1);
         case "D":
            return movePairings[4 + offset] + move.substring(1);
         case "F":
            return movePairings[offset] + move.substring(1);
         case "B":
            return movePairings[1 - offset] + move.substring(1);
         default:
            return move;
         }
```

```
      case "y":
         switch (move.substring(0, 1)) {
         case "R":
            return movePairings[4 + offset] + move.substring(1);
         case "L":
            return movePairings[5 - offset] + move.substring(1);
         case "F":
            return movePairings[2 + offset] + move.substring(1);
         case "B":
            return movePairings[3 - offset] + move.substring(1);
         default:
            return move;
         }
      case "z":
         switch (move.substring(0, 1)) {
         case "U":
            return movePairings[3 - offset] + move.substring(1);
         case "D":
            return movePairings[2 + offset] + move.substring(1);
         case "L":
            return movePairings[offset] + move.substring(1);
         case "R":
            return movePairings[1 - offset] + move.substring(1);
         default:
            return move;
         }
      }
   } else {
      switch (rotation.substring(0, 1)) {
      case "x":
         if ("LR".contains(move.substring(0, 1)))
            return move;
      case "y":
         if ("UD".contains(move.substring(0, 1)))
            return move;
      case "z":
         if ("FB".contains(move.substring(0, 1)))
            return move;
      }
```

```
      index = LinearSearch.linearSearch(movePairings, move.substring(0, 1));
      return movePairings[index + ((index % 2 == 0) ? 1 : -1)];
   }

   return "-1";
}
```

SolveMaster – Cancel moves to simplify a solution

- See sections 3.05 - 3.10 of the testing section. All of the specified simplifications are performed. Improvements to the algorithm could lead to solutions that are better for speed rather than move-count.
- Go to page 378 of the appendix section to see SolveMaster class.

Cancels moves such as L2 L' -> L.

Parameters:

`originalMoves` - the moves to simplify

```
private static void cancelMoves(LinkedList<String> originalMoves) {
   // Stores the index of the current move being examined.
   int index = 0;
   // Stores the resulting combination the two moves being examined.
   String combination;
   // Stores the first element being examined.
   String one;
   // Stores the second element being examined.
   String two;
   // Stores the plane of the move, e.g. R2 has plane = R
   String moveType;

   while (index < (originalMoves.size() - 1)) {
      one = originalMoves.get(index).trim();
      two = originalMoves.get(index + 1).trim();

      // If the moves act on the same slice then
      if (one.substring(0, 1).equals(two.substring(0, 1))) {
         moveType = one.substring(0, 1);
         one = one.substring(1);
         two = two.substring(1);
         combination = getCombination(one, two);

         if (combination.equals("NOT_MATCHING"))
            ++index;
```

```
         else {
            if (combination.equals("-1")) {
               /*
                * The moves cancel, so remove each of them.
                */
               originalMoves.remove(index);
               originalMoves.remove(index);
            } else {
               /*
                * The two moves can be simplified to one, so remove the
                * second move and alter the first one.
                */
               originalMoves.remove(index + 1);
               originalMoves.set(index, moveType + combination);
            }
            /*
             * The moves have been simplified, so index needs to go back
             * to check if further simplifications will occur.
             */
            index = (index == 0) ? 0 : index - 1;
         }
      } else {
         ++index;
      }
   }

   /*
    * This simplifies "x y x" to "z2 y"
    */
   for (int i = 0; i < originalMoves.size() - 2; ++i) {
      if ((originalMoves.get(i).equals("x")) && (originalMoves.get(i + 1).equals("z"))
            && (originalMoves.get(i + 2).equals("x"))) {
         originalMoves.remove(i);
         originalMoves.set(i, "z2");
         originalMoves.set(i + 1, "y'");
      }
   }
}
```

SolveMaster – Record the moves to solve the cube

- See sections 3.05 - 3.10 of the testing section.
- Go to page 378 of the appendix section to see SolveMaster class.

Checks that the argument is not null, then records the moves.

Parameters:

`moves` - the moves to be recorded

```
public void catalogMoves(String moves) {
   if (moves == null)
      return;

   catMoves(moves.trim(), 0);
}
```

Recursively records the specified moves, one by one.
E.g. "R U R' F" will be recorded as "R", "U", "R'", "F"

Parameters:

`moves` - the moves to be recorded

`index` - the last index of a space

```
private void catMoves(String moves, int index) {
   // This indicates that all moves have been recorded.
   if (index >= moves.length())
      return;

   // Stores the remaining moves to be recorded.
   String remainingMoves = moves.substring(index).trim();
   // Stores the index of the first space in the remaining moves to be
   // recorded so that the next move can be identified.
```

```
   int indexOfSpace = remainingMoves.indexOf(" ");

   if (indexOfSpace == -1)
      indexOfSpace = remainingMoves.length();

   catalogMove(remainingMoves.substring(0, indexOfSpace));
   catMoves(moves, index + indexOfSpace + 1);
}
```

SolveMaster – Determine whether a piece is in the correct position (ignoring orientation)

- See sections 3.05 - 3.10 of the testing section.
- Go to page 378 of the appendix section to see SolveMaster class.

Parameters:

`cubie` - the cubie to be analysed

Returns:

true if the specified cubie is in the correct position on the cube (regardless of orientation);
false otherwise

```
public boolean isPieceIsInCorrectPosition(Cubie cubie) {
   // Stores the index of the cubie on the cube.
   int index = getIndexOf(cubie);
   // Stores the colours of stickers on the cubie so that the expected
   // centres around the cubie can be compared with the actual centres.
   Color[] centres = new Color[cubie.getStickers().length];
   // Stores true if the current centre sticker being examined is found.
   boolean found;
   // Stores the indices of the slices that immediately surround the cubie.
   int[] slicesIndex = (centres.length == 2) ? sliceEdgeSharing[index] : sliceCornerSharing[index];

   for (int i = 0; i < centres.length; ++i)
      centres[i] = cube.getSlice(slicesIndex[i]).getCentre();

   /*
    * This compares each expected centre colour with each colour of the
    * cubie. If all colours are the shared, then the piece is in the
    * correct position.
    */
   for (int i = 0; i < centres.length; ++i) {
      found = false;
      for (int j = 0; j < centres.length; ++j) {
         if (centres[i].equals(cubie.getStickers()[j])) {
            found = true;
            break;
```

```
            }
         }

         if (!found)
            return false;
      }

      return true;
}
```

SolveMaster – Get the index of a cubie's destination

- See sections 3.05 - 3.10 of the testing section.
- Go to page 378 of the appendix section to see SolveMaster class.

Parameters:

`cubie` - the cubie to be analysed

Returns:

the destination of the cubie on a solved cube according the current permutation of the centres.

```
protected int getIndexOfDestination(Cubie cubie) {
   // Stores the number of stickers on the cubie, indicating whether it is
   // a corner or edge.
   int numStickers = cubie.getStickers().length;
   // Stores the indices of the slices that surround each cubie at each
   // location.
   int[][] slicesIndices = (numStickers == 2) ? sliceEdgeSharing : sliceCornerSharing;
   // Stores true if the current centre has been found on the cubie.
   boolean foundCentre;
   // Stores true if the current slicesIndices element represents the
   // destination of the cubie.
   boolean foundPosition;

   /*
    * Goes through each position on the cube to see if the centre
    * surrounding that position match the stickers of the specified cubie.
    */
   for (int i = 0; i < slicesIndices.length; ++i) {
      foundPosition = true;
      for (int j = 0; j < slicesIndices[i].length; ++j) {
         foundCentre = false;
         for (int k = 0; k < numStickers; ++k) {
            if (cube.getSlice(slicesIndices[i][j]).getCentre().equals(cubie.getStickers()[k])) {
               foundCentre = true;
               break;
            }
```

```
         }

         if (!foundCentre) {
            foundPosition = false;
            break;
         }
      }
      if (foundPosition)
         return i;
   }

   return -1;
}
```

SolveMaster – Generate a state string for the current state of the cube

- See section 4.01 of the testing section.
- Go to page 378 of the appendix section to see SolveMaster class.

This returns a string representation of the current state of the cube so that it can be saved/loaded to/from file. The first two colours are top and front centres, and other colours are each sticker of each corner then each edge.

Returns:

the current state of the cube represented as a string

```
public String getStateString() {
   // Accumulates the resulting state-string
   String result = "";
   // Stores the stickers of the current cubie being examined.
   Color[] cStickers;

   /*
    * Stores the top and front centre colours.
    */
   result = Cubie.getColorToWord(cube.getSlice(0).getCentre()) + ","
         + Cubie.getColorToWord(cube.getSlice(4).getCentre()) + ",";

   /*
    * Stores the corner stickers.
    */
   for (int i = 0; i < 8; ++i) {
      cStickers = cube.getCorner(i).getStickers();

      for (int j = 0; j < 3; ++j)
         result += Cubie.getColorToWord(cStickers[j]) + ",";
   }

   /*
    * Stores the edge stickers.
    */
```

```
   for (int i = 0; i < 12; ++i) {
      cStickers = cube.getEdge(i).getStickers();

      for (int j = 0; j < 2; ++j)
         result += Cubie.getColorToWord(cStickers[j]) + ",";
   }

   return result;
}
```

Statistics – Calculate the average of x

- See section 1.07 of the testing section.
- Go to page 410 of the appendix section to see Statistics class.

Parameters:

`sizeOfAverage` - must be 5 or over

`times` - the times from which the average is calculated

Returns:

the average of the specified size.

```
public static double getAverageOf(int sizeOfAverage, LinkedList<Solve> times) {
   // Stores the number of DNF times in the past 'sizeOfAverage' times.
   int numDNF = getNumDNF(sizeOfAverage, times);
   int size = times.size();

   if ((times.size() < sizeOfAverage) || (sizeOfAverage < 5) || (numDNF > 1))
      return -1;

   // Stores the raw numerical representation of the times.
   double[] raw = new double[sizeOfAverage];
   // Accumulates the sum of the times to be used in the mean, i.e. the
   // middle (x - 2) times.
   double sum = 0;
   // Stores the number of times to be used in the mean so that the mean
   // can be found.
   double factor = /* (x < 5) ? x : */sizeOfAverage - 2;
   // Stores the index of the first element to be used in the mean.
   int start;
   // Stores the index of the last element to be used in the mean.
   int end;

   start = 1;
   end = sizeOfAverage - 1;
```

```
   /*
    * This allows the number of times specified to be processed.
    */
   for (int i = 0; i < sizeOfAverage; ++i)
      raw[i] = times.get(size - sizeOfAverage + i).getNumericTime();

   /*
    * This gets all DNFs (-1s) at the start and sorts other times.
    */
   Sorter.quickSort(raw);

   /*
    * If there is a DNF time, then the first element will represent the
    * slowest time and the second element the fastest, so the last three
    * elements need to be processed.
    */
   if (numDNF > 0) {
      ++start;
      ++end;
   }

   for (int i = start; i < end; ++i) {
      sum += raw[i];
   }

   return (sum / factor);
}
```

Statistics – Generate a string containing the formatted statistics

– Go to page 410 of the appendix section to see Statistics class.

Returns a formatted average (i.e. all times are separated by commas and the fastest and slowest times are surrounded with brackets) of the specified size as the string

Parameters:

`sizeOfAverage` - the size of the average to be returned

`times` - the times from which the average is calculated

Returns:

a formatted string of the average in the form "[average] [time_1], [time_2], ..., [time_n]"

```
public static String getFormattedAverage(int sizeOfAverage, LinkedList<Solve> times) {
   int size = times.size();
   // Stores the times in order of their numerical value.
   double[] orderedTimes = new double[sizeOfAverage];
   // Stores a copy of the times in the average.
   double[] timesCopy = new double[sizeOfAverage];
   /*
    * The first element indicates whether or not the slowest time has been
    * found and enclosed in brackets; The second element indicates whether
    * or no the fastest time has been found and enclosed in brackets.
    */
   boolean[] found = { false, false };
   String formattedStatistics = "";

   if (size < sizeOfAverage)
      return "";

   for (int i = 0; i < sizeOfAverage; ++i)
      orderedTimes[i] = times.get(size - sizeOfAverage + i).getNumericTime();

   timesCopy = Arrays.copyOf(orderedTimes, orderedTimes.length);
   Sorter.quickSort(orderedTimes);
```

```
   sortByDNF(orderedTimes);

   for (int i = 0; i < timesCopy.length; ++i) {
      /*
       * If the current time is the slowest time the average, and the
       * slowest time has not already been identified, then put brackets
       * around it.
       */
      if ((!found[1]) && (timesCopy[i] == orderedTimes[orderedTimes.length - 1])) {
         found[1] = true;
         formattedStatistics += String.format("(%s)",
               (timesCopy[i] == -1) ? "DNF" : Solve.getSecondsToFormattedString(timesCopy[i]));
      }
      /*
       * If the current time is the fastest time the average, and the
       * fastest time has not already been identified, then put brackets
       * around it.
       */
      else if ((timesCopy[i] == orderedTimes[0]) && (!found[0])) {
         found[0] = true;
         formattedStatistics += String.format("(%s)",
               (timesCopy[i] == -1) ? "DNF" : Solve.getSecondsToFormattedString(timesCopy[i]));
      } else
         formattedStatistics += String.format("%s",
               (timesCopy[i] == -1) ? "DNF" : Solve.getSecondsToFormattedString(timesCopy[i]));

      formattedStatistics += ", ";
   }
   /*
    * Remove the last comma
    */
   if (formattedStatistics.contains(","))
      formattedStatistics = formattedStatistics.substring(0, formattedStatistics.lastIndexOf(","));

   return formattedStatistics;
}
```

Statistics – Sort an array of values in ascending order where DNF represents infinity

– Go to page 410 of the appendix section to see Statistics class.

Sorts the specified list so that '-1' elements are at the end
Parameters:

`list` - the list to be sorted

```
private static void sortByDNF(double[] list) {
   // Stores the index of the element whose whose value will become list[j]
   int i = 0;
   // Stores the index of the element being examined.
   int j = 0;

   /*
    * Pushes all non-DNF times to start of array.
    */
   while ((i < list.length) && (j < list.length)) {
      if (list[j] != -1) {
         list[i] = list[j];
         ++i;
      }

      ++j;
   }

   /*
    * All non-DNF times are before i, so set all elements after to -1
    */
   for (; i < list.length; ++i) {
      list[i] = -1;
   }
}
```

Tutorial – Load a tutorial from file

- See sections 1.17-1.22 of the testing section. All data in the text file containing the tutorial information is extracted and interpreter correctly so that the user can explore the tutorial interactively. This algorithm has some validation checks present to stop infinite loops and crashing, but there are probably erroneous files that will cause problems.
- Go to page 451 of the appendix section to see Tutorial class.

Loads the tutorial stored in the file at the specified file path. The fields store the data from the file.

Parameters:

`filePath` - the path of the file which contains the tutorial

Throws:

`java.lang.Exception` - if the file cannot be accessed properly or is not formatted correctly

```
public void loadTutorial(String filePath) throws Exception {
   // Stores the number of lines in the the file.
   int numLines;
   // Stores the index of the current line in the file being processed.
   int currentIndex = 0;
   // Stores the index of the next heading in the file.
   int headingIndex;
   hintIndex = -1;
   subTutorialIndex = 0;

   currentFile = new TextFile();
   currentFile.setFilePath(filePath);
   currentFile.setIO(TextFile.READ);

   fileData = currentFile.readAllLines();
   numLines = fileData.length;
   currentFile.close();

   numSubTutorials = getNumSubTutorials();
   scrambles = new String[numSubTutorials];
```

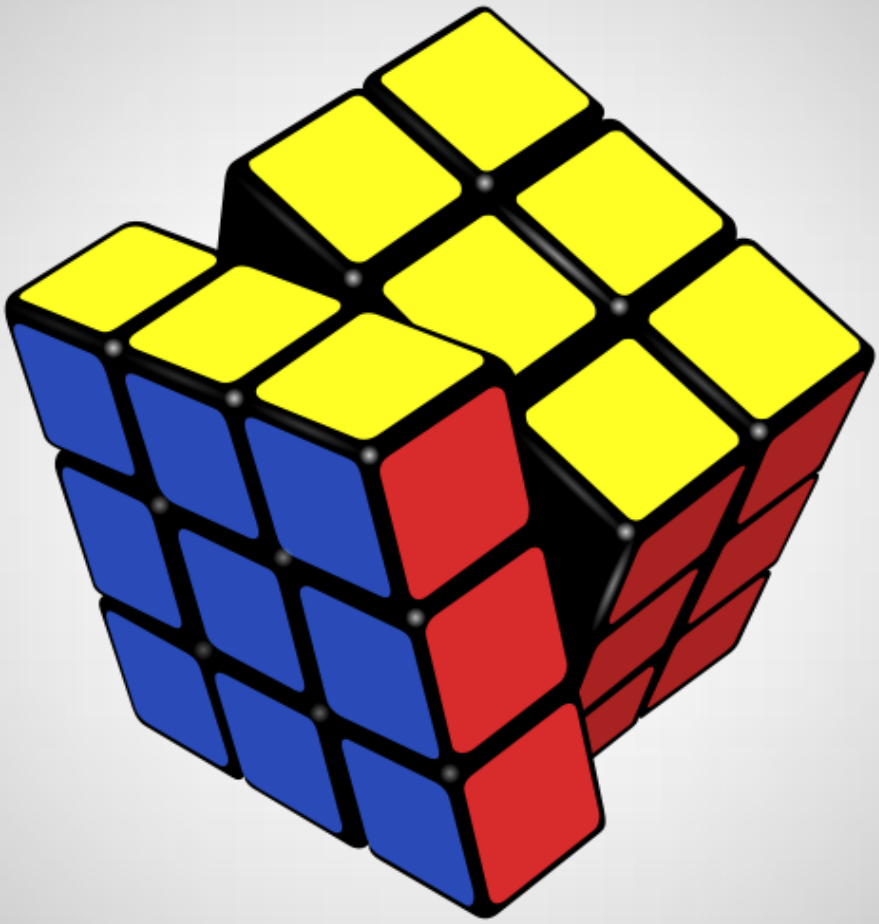
```
   descriptions = new String[numSubTutorials];
   expectedSolvedPieces = new String[numSubTutorials][20];
   hints = new String[numSubTutorials][20];
   optimalSolutions = new String[numSubTutorials][20];
   explanations = new String[numSubTutorials];
   tutorialsRequiringUserAction = new boolean[numSubTutorials];
   tutorialsRequiringUserSolution = new boolean[numSubTutorials];

   for (int i = 0; i < numSubTutorials; ++i) {
      if (fileData[currentIndex].contains("ENABLE"))
         tutorialsRequiringUserAction[i] = true;
      else
         tutorialsRequiringUserAction[i] = false;

      currentIndex += 2;
      scrambles[i] = fileData[currentIndex];

      currentIndex += 2;

      /*
       * This finds the next section in the sub-tutorial
       */
      descriptions[i] = "";
      while ((!fileData[currentIndex].equals("EXPECTED FINAL STATE:")) && (!fileData[currentIndex].equals("*"))) {
         descriptions[i] += fileData[currentIndex] + "\n";
         ++currentIndex;
      }

      descriptions[i] = descriptions[i].substring(0, descriptions[i].lastIndexOf("\n"));

      if (fileData[currentIndex].equals("EXPECTED FINAL STATE:")) {
         tutorialsRequiringUserSolution[i] = true;
         ++currentIndex;

         /*
          * Finds and stores the pieces that are expected to be solved.
          * It does this by finding the index of the next heading and
          * then storing each line before that heading.
          */
```

```
        headingIndex = LinearSearch.linearSearchStartsWith(
              Arrays.copyOfRange(fileData, currentIndex, numLines), "HINT:");
        expectedSolvedPieces[i] = new String[headingIndex];
        for (int j = 0; j < headingIndex; ++j) {
           expectedSolvedPieces[i][j] = fileData[currentIndex];
           ++currentIndex;
        }
        ++currentIndex;

        /*
         * Finds and stores the hints. It does this by finding the index
         * of the next heading and then storing each line before that
         * heading.
         */
        headingIndex = LinearSearch.linearSearchStartsWith(
              Arrays.copyOfRange(fileData, currentIndex, numLines), "OPTIMAL SOLUTION:");
        hints[i] = new String[headingIndex];
        for (int j = 0; j < headingIndex; ++j) {
           hints[i][j] = fileData[currentIndex];
           ++currentIndex;
        }
        ++currentIndex;

        /*
         * Finds and stores the optimal solutions. It does this by
         * finding the index of the next heading and then storing each
         * line before that heading.
         */
        headingIndex = LinearSearch.linearSearchStartsWith(
              Arrays.copyOfRange(fileData, currentIndex, numLines), "EXPLANATION:");
        optimalSolutions[i] = new String[headingIndex];
        for (int j = 0; j < headingIndex; ++j) {
           optimalSolutions[i][j] = fileData[currentIndex];
           ++currentIndex;
        }
        ++currentIndex;

        /*
         * Finds and stores the explanation. It does this by finding the
```

```
          * index of the next heading and then appending each line before
          * the heading to explanations[i].
          */
         headingIndex = LinearSearch.linearSearchStartsWith(
               Arrays.copyOfRange(fileData, currentIndex, numLines), "*");
         explanations[i] = "";
         for (int j = 0; j < headingIndex; ++j) {
            explanations[i] += fileData[currentIndex] + "\n";
            ++currentIndex;
         }
         ++currentIndex;

         explanations[i] = explanations[i].substring(0, explanations[i].lastIndexOf("\n"));
      } else {
         ++currentIndex;
      }
   }

   tutorialLoaded = true;
}
```

SECTION 5

USER MANUALS



Student User Manual

Contents

Introduction

Kuubik was developed to help people learn how to solve the Rubik's cube. There are features for both learning and practising, so the system can be used by people with all levels of experience. The system was intended to provide a comprehensive environment that would enable users to further their skill rapidly. The system can be used by anyone who wishes to learn how to solve the Rubik's cube.

The program allows you to:

- Learn how to solve the Rubik's cube using built-in and custom-made tutorials.
- Automatically generate solutions for a given cube state.
- Paint custom states onto a virtual Rubik's cube.
- Practise solving the Rubik's cube using randomly generated scrambles.
- Save/load cube states, solve information, statistics and scrambles.
- View a graph of your times and save this graph as an image.

Installation

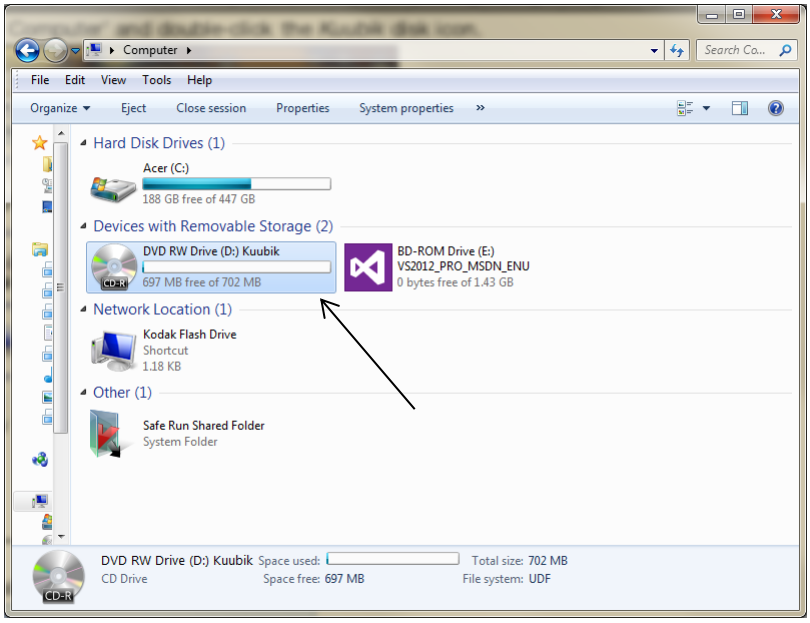
The minimum requirements for the system are:
OS: Windows XP/Vista/7
Processor: 1.0 GHz
Memory: 128 MB RAM
Graphics: N/A
Hard Drive: 8 MB available space

1. Ensure your version of java is up to date by going to *www.java.com/en* and downloading any relevant updates.

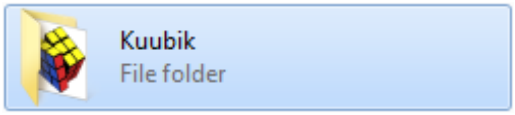
2. If you do not have a PDF software, download Adobe Reader from *get.adobe.com/uk/reader*

3. Insert the installation disc into the computer.

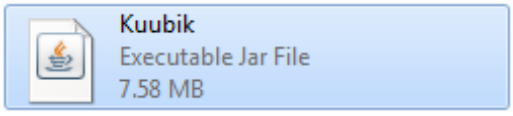
4. Go to 'My Computer' and double-click the *Kuubik* disc icon.



5. A folder called '*Kuubik*' is saved on the disc. Drag the *Kuubik* folder into your My Documents.



6. Once the *Rubik's Cube* folder is stored on your computer, you can open the folder and double-click the *Rubik's Cube* JAR file to open the program.

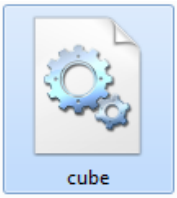


7. To access the program through the desktop, right-click on the *Rubik's Cube* JAR file and select 'Create shortcut', then drag the shortcut onto the desktop.

Backup and Recovery

To back up the data in the database, you can copy the *cube* database file to a different location.

1. Locate the *Kuubik* folder, which you saved during installation.
2. In the *Kuubik* folder, go the *res* folder.
3. Copy the *cube* database file to a different location, such as an external disk or flash drive.



If you need to recover the database, e.g. after re-installing the system, just copy and paste the *cube* database file that you saved as a back up to the *res* file in the *Kuubik* folder.

How to Use the System

Shortcuts

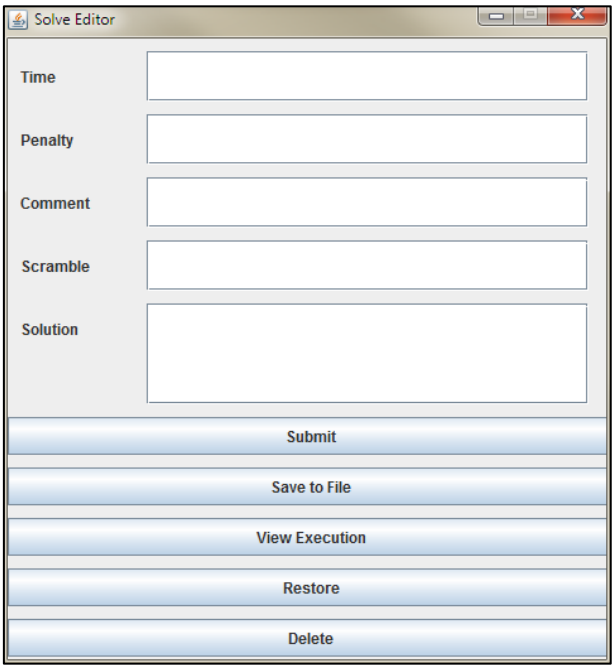
Action	Shortcut
Start new solve	Alt + 1
Save statistics	Ctrl + S
Load cube state	Ctrl + O
Add solve	Ctrl + D
Edit selected solve	Ctrl + E
Cancel solve	Ctrl + Q
Solve cube	Ctrl + R
Time graph – Save as image	Ctrl + S
Time graph – Close window	Ctrl + W
Time graph – Reset zoom	Ctrl + R
Time graph – 2D	Ctrl + 2
Time graph – 3D	Ctrl + 3

How to perform moves on the virtual cube

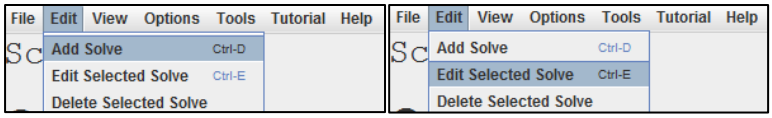
You can make the virtual cube perform moves by pressing the corresponding keys on the keyboard:

Move	Key
U	j
U’	f
D	s
D’	l
R	i
R’	k
L	d
L’	e
F	h
F’	g
B	w
B’	o
M	x
M’	.
Rw	u
Rw’	m
Lw	v
Lw’	r
x	y
x’	n
y	;
y’	a
z	p
z’	q

How to use the Solve Editor form

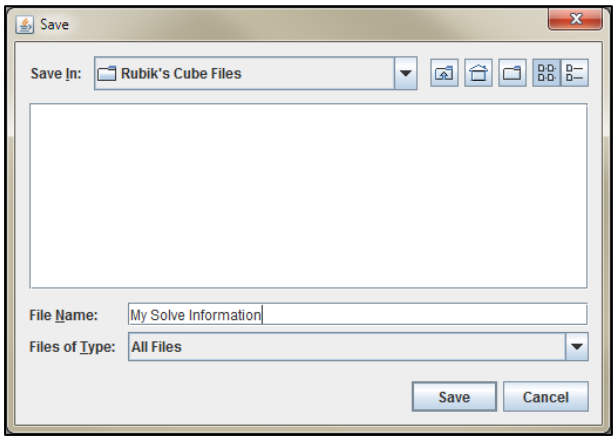


- You can access the Solve Editor form by selecting the *Add Solve* menu item or *Edit Selected Solve* menu item.

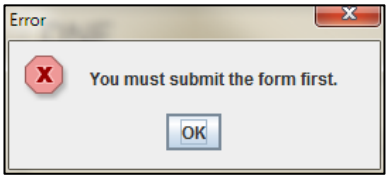


- *Submit Button*: The data in the form will be submitted and the corresponding in the list at the right-hand side of the screen will be updated.

- *Save to File Button*: If the data in the form is valid, then a window will appear asking you to choose a location to save the file.



- *View Execution Button*: The cube will be scrambled instantly using the moves provided in the *scramble field*, and then the moves provided in the *solution field* will be performed in real-time. The rate at which the moves are performed is specified in the preferences. This feature is available only if you are **editing** the solve; if you are adding the solve and you click this button, then the following error message will appear:



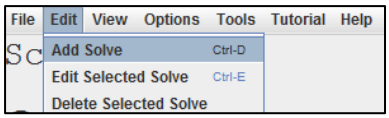
- *Restore Button*: Any changes made will be discarded and the original data will be shown in the form.

- *Delete Button*: The window will close and the corresponding item in the list at the right-hand side of the screen will be removed.

For more information, see *How to Add Solve Information*, *How to Edit Solve Information*, and *How to Delete Solve Information*

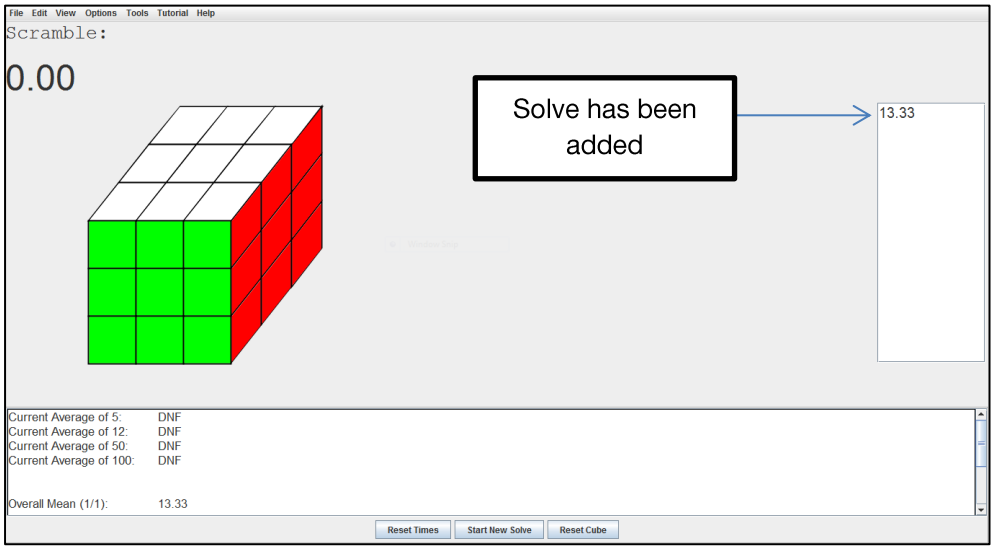
How to add solve information

1. *Main window menu bar → Edit → Add Solve*

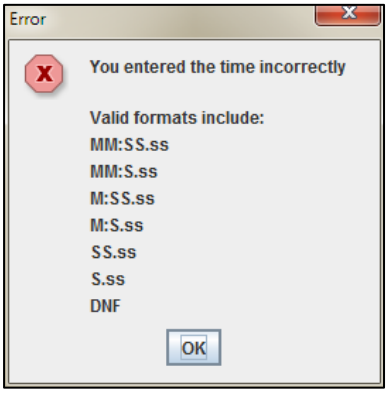


2. Enter the solve information into the *Solve Editor* form then click the *Submit* button. The *time field* must not be left blank, but the other fields may contain any type of data (including blank data).

3. The updated solve should then appear in the list on the right-hand side of the screen.



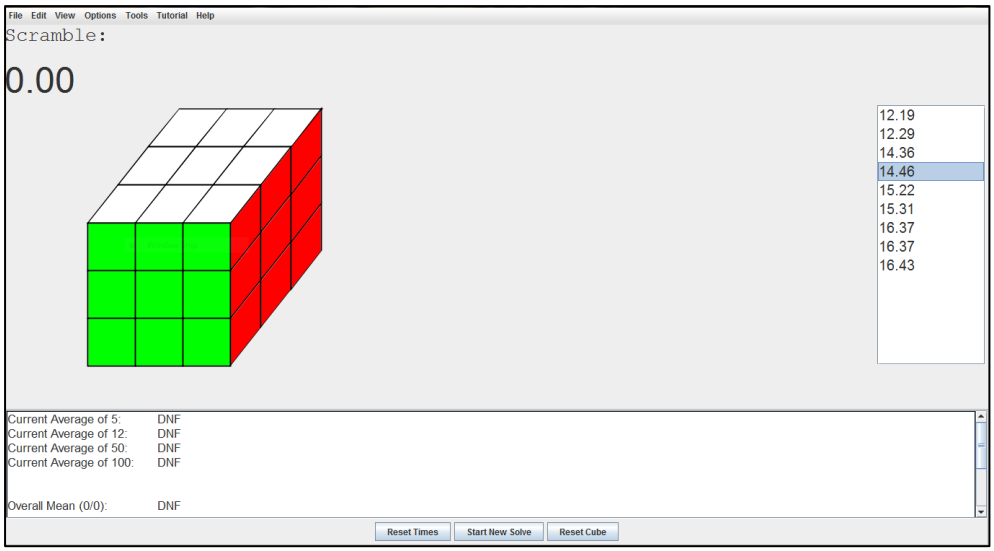
4. If the data in the *time field* is incorrect, then the following error message will be shown:



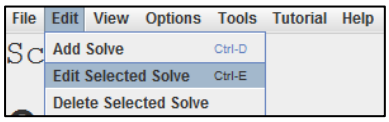
See the *Data Input Guidelines* section for help on how to enter valid data.

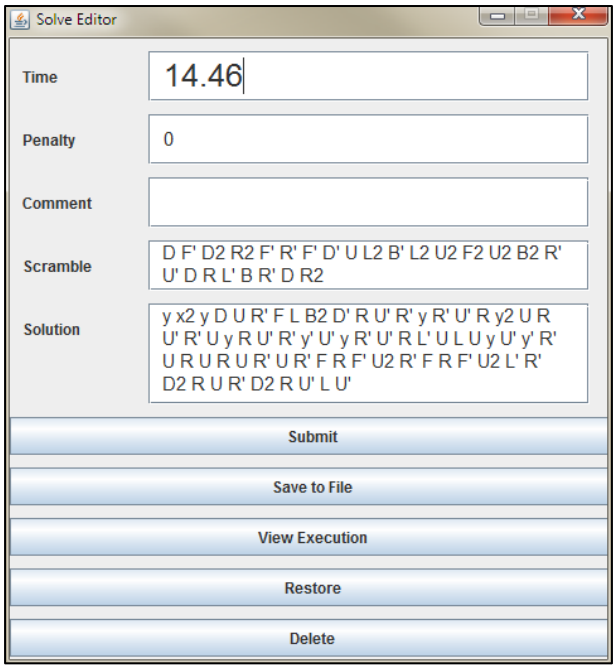
How to edit solve information

1. Select the solve in the list that you want to edit.



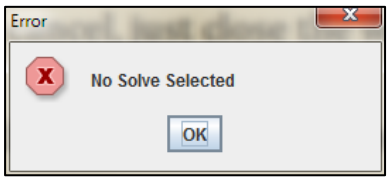
2. *Main window menu bar → Edit → Edit Selected Solve*





3. Edit the information as required then click the *Submit* button; the corresponding item in the list will be updated. The *time field* must not be left blank, but the other fields may contain any type of data (including blank data). If you wish to cancel, just close the window and any changes will be discarded.

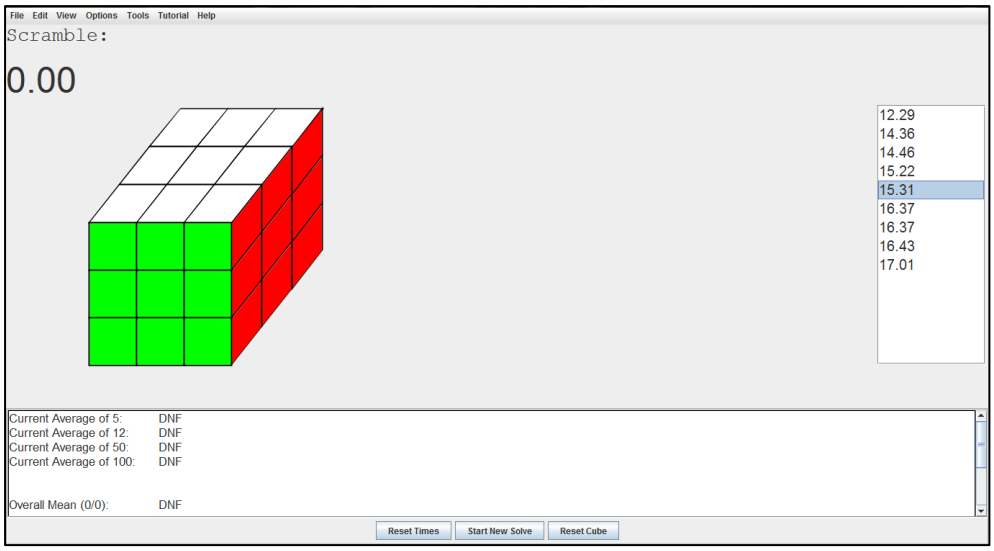
4. If no item in the list is selected, then the following error message will be shown:



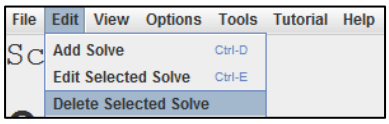
See the *Data Input Guidelines* section for help on how to enter valid data.

How to delete solve information

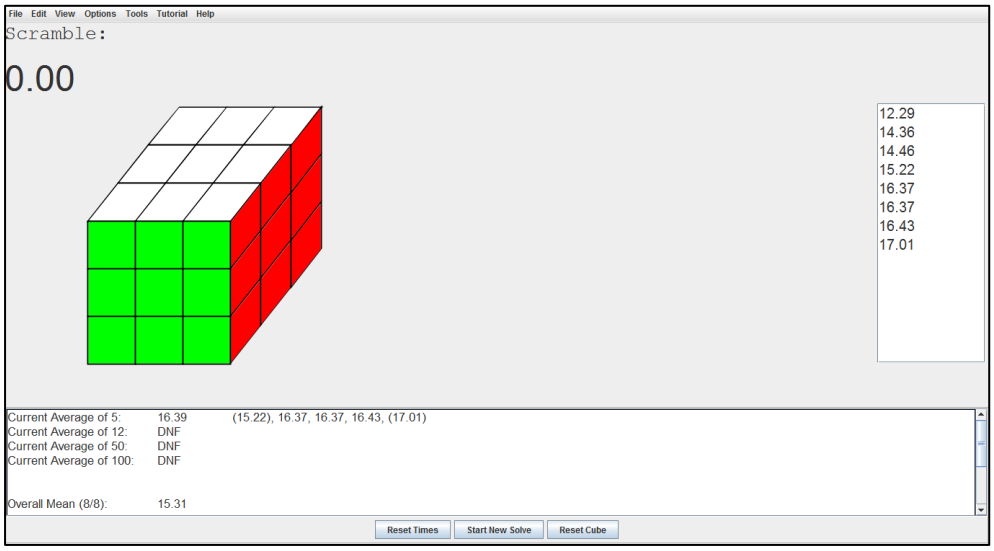
1. Select the solve in the list that you want to delete.



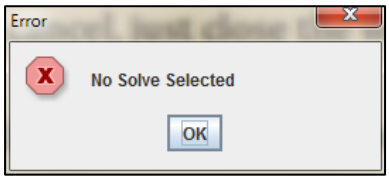
2. *Main window menu bar → Edit → Delete Selected Solve*



3. The selected solve will be removed from the list.

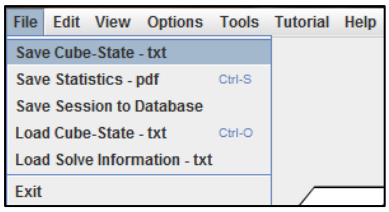


4. If no item in the list is selected, then the following error message will be shown:

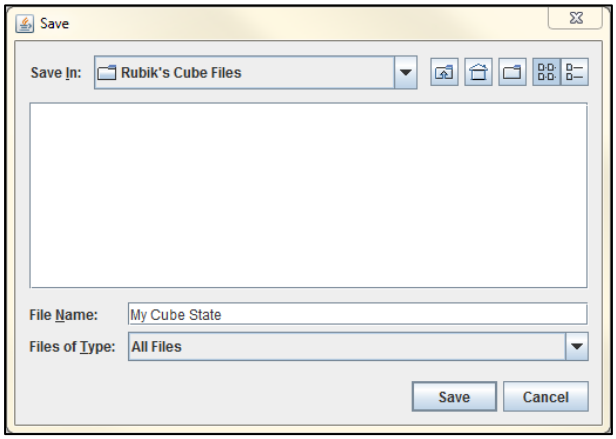


How to save a cube state

1. *Main window menu bar → File → Save Cube-State – txt*

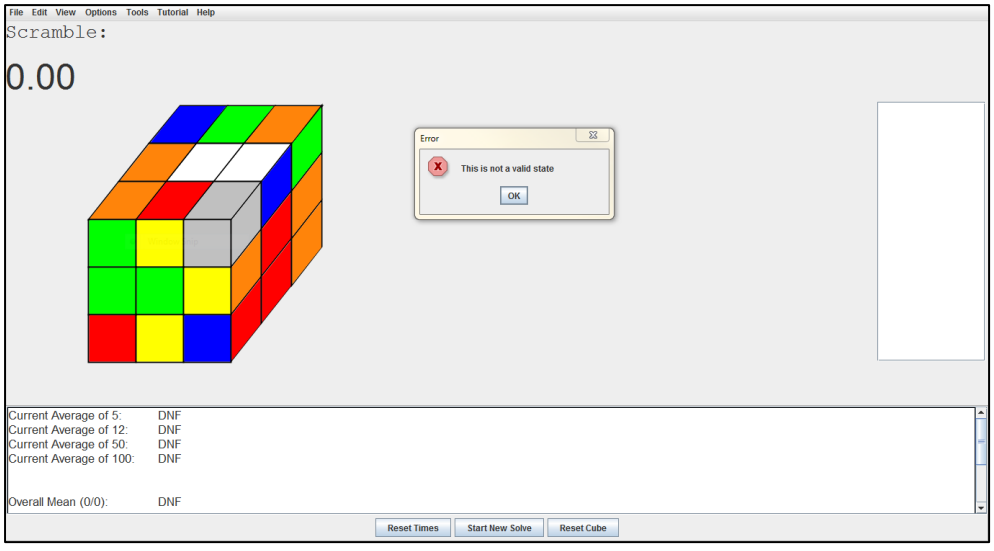


2. Choose the location for the file.



3. The current state/permutation of the cube will be saved in this file.

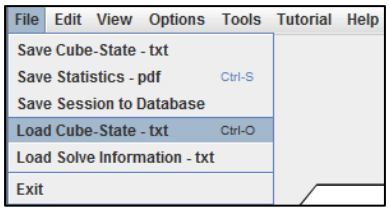
4. If the cube is not in a valid state when trying to save, then the following error message will be shown, and the corresponding invalid stickers will be removed.



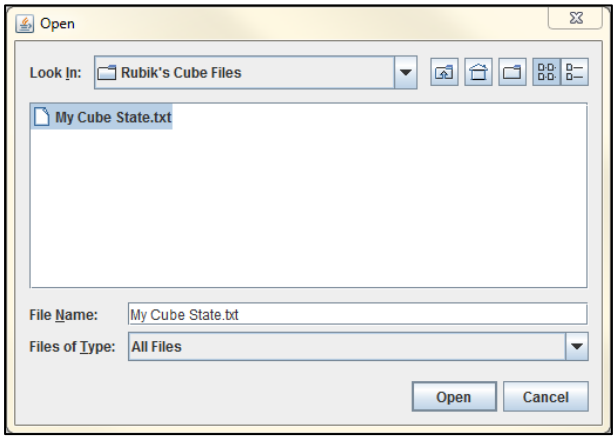
See the *Data Input Guidelines* section for help on how to enter valid cube states.

How to load a cube state

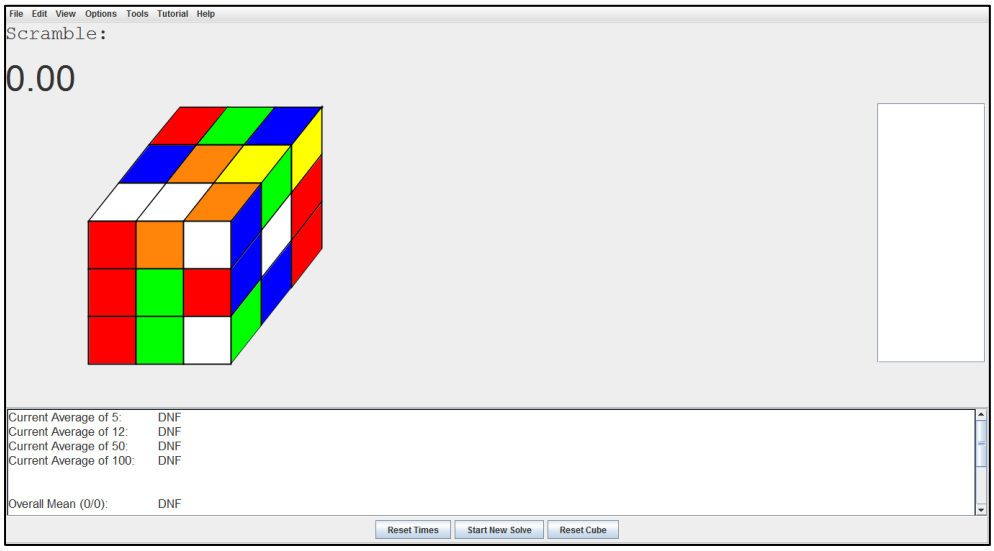
1. *Main window menu bar → File → Load Cube State – txt*



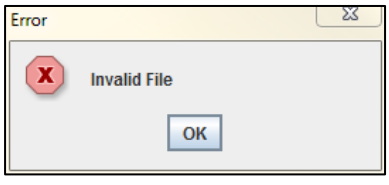
2. Choose the file to load.



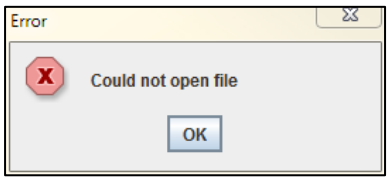
3. The cube's state will change to match the data stored in the text file.



4. If the text file contains data that is not in the correct format, then the following error message will be shown:



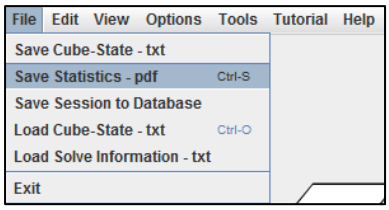
5. If the file cannot be opened, then the following error message will be shown:



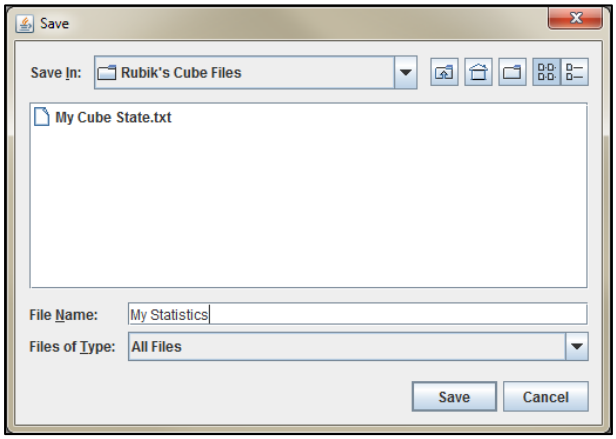
See the *Troubleshooting* section for advice on how to deal with these errors.

How to save statistics

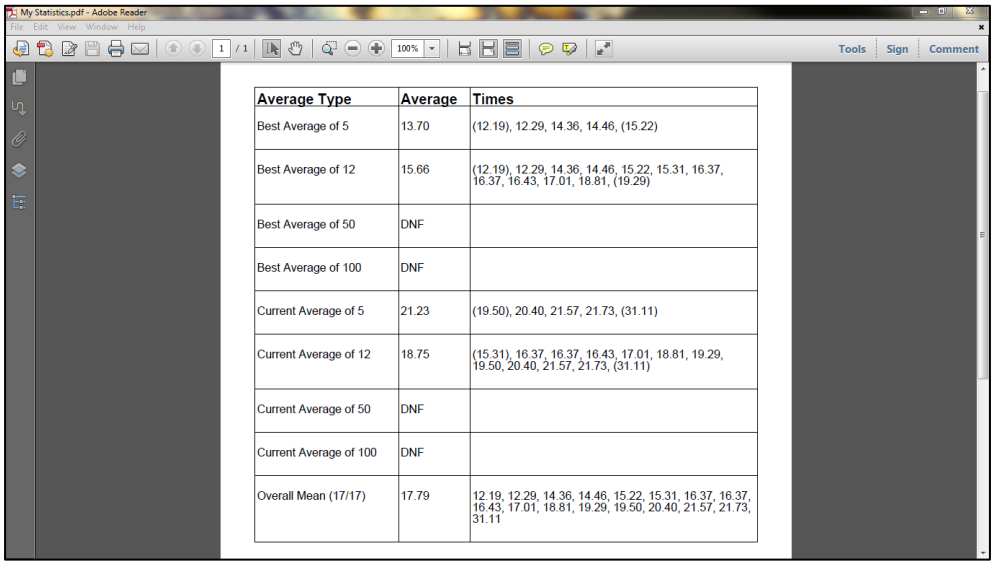
1. *Main window menu bar → File → Save Statistics – pdf*



2. Choose a location for the file.



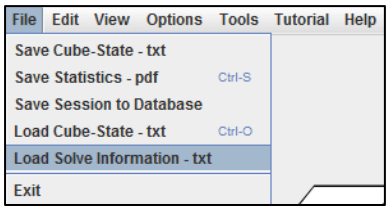
3. The statistics will be saved as a pdf file in the location you choose.



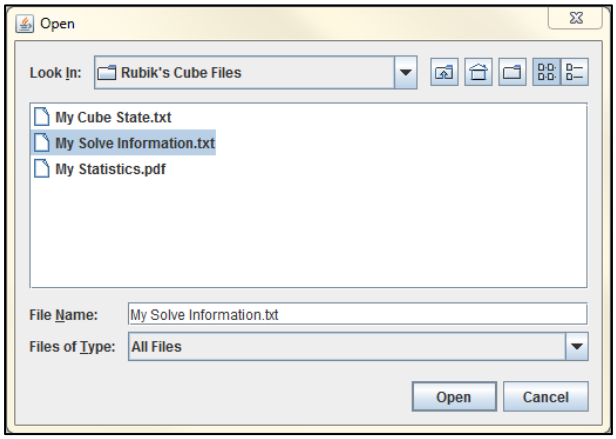
Average Type	Average	Times
Best Average of 5	13.70	(12.19), 12.29, 14.36, 14.46, (15.22)
Best Average of 12	15.66	(12.19), 12.29, 14.36, 14.46, 15.22, 15.31, 16.37, 16.37, 16.43, 17.01, 18.81, (19.29)
Best Average of 50	DNF	
Best Average of 100	DNF	
Current Average of 5	21.23	(19.50), 20.40, 21.57, 21.73, (31.11)
Current Average of 12	18.75	(15.31), 16.37, 16.37, 16.43, 17.01, 18.81, 19.29, 19.50, 20.40, 21.57, 21.73, (31.11)
Current Average of 50	DNF	
Current Average of 100	DNF	
Overall Mean (17/17)	17.79	12.19, 12.29, 14.36, 14.46, 15.22, 15.31, 16.37, 16.37, 16.43, 17.01, 18.81, 19.29, 19.50, 20.40, 21.57, 21.73, 31.11

How to load solve information

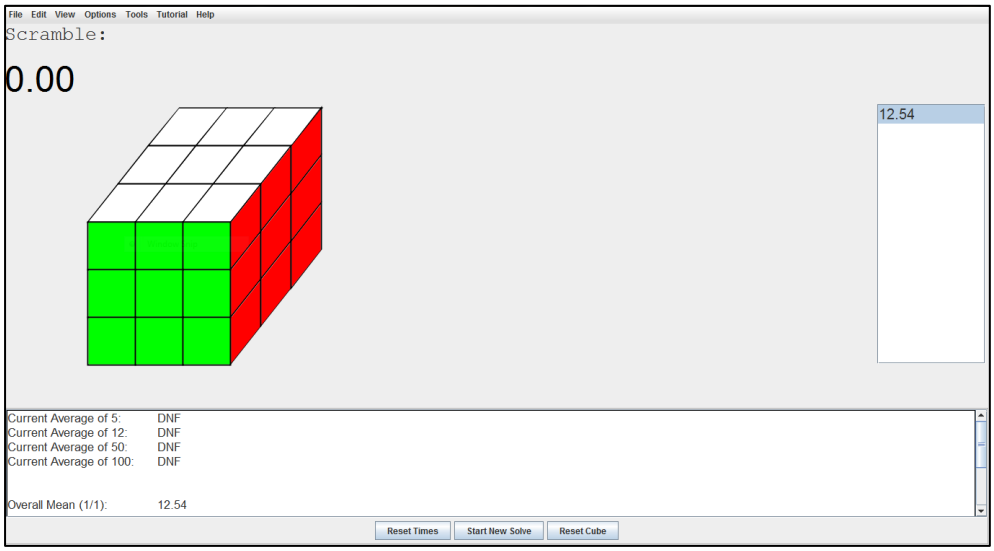
1. *Main window menu bar → File → Load Solve Information - txt*



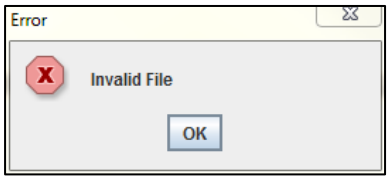
2. Choose the file to load.



3. The solve information will be added to the list at the right-hand side of the screen.



4. If the text file contains data that is not in the correct format, then the following error message will be shown:

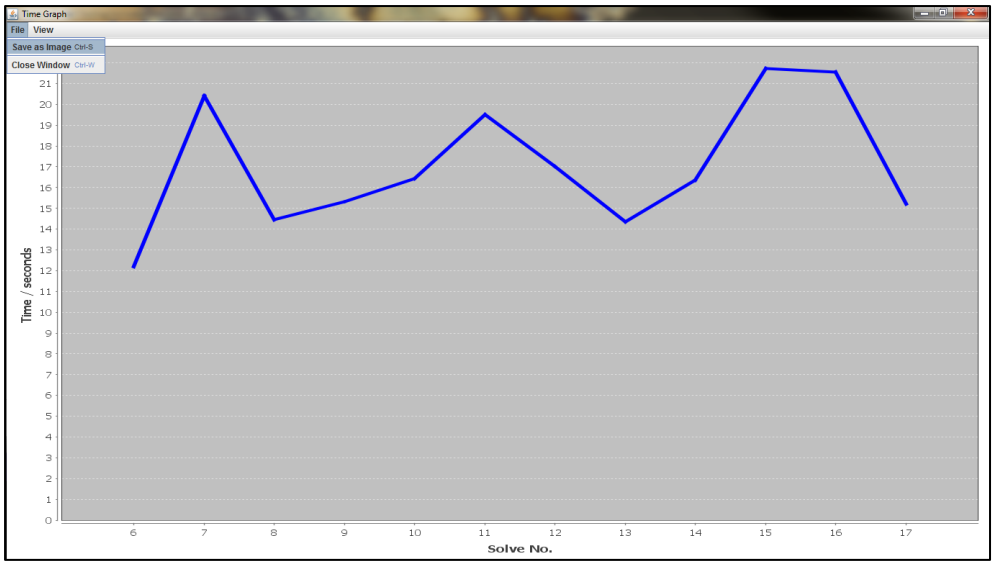


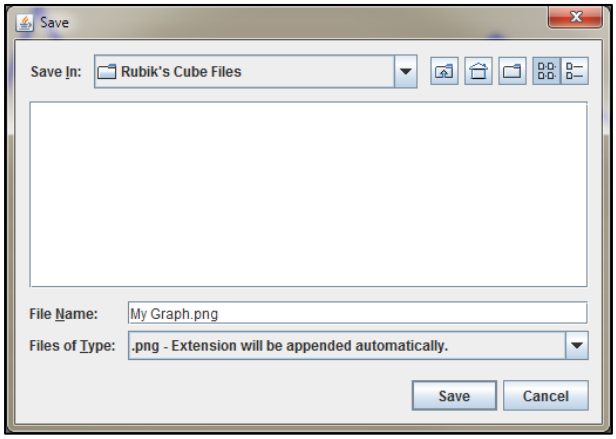
See the *Troubleshooting* section for advice on how to deal with these errors.

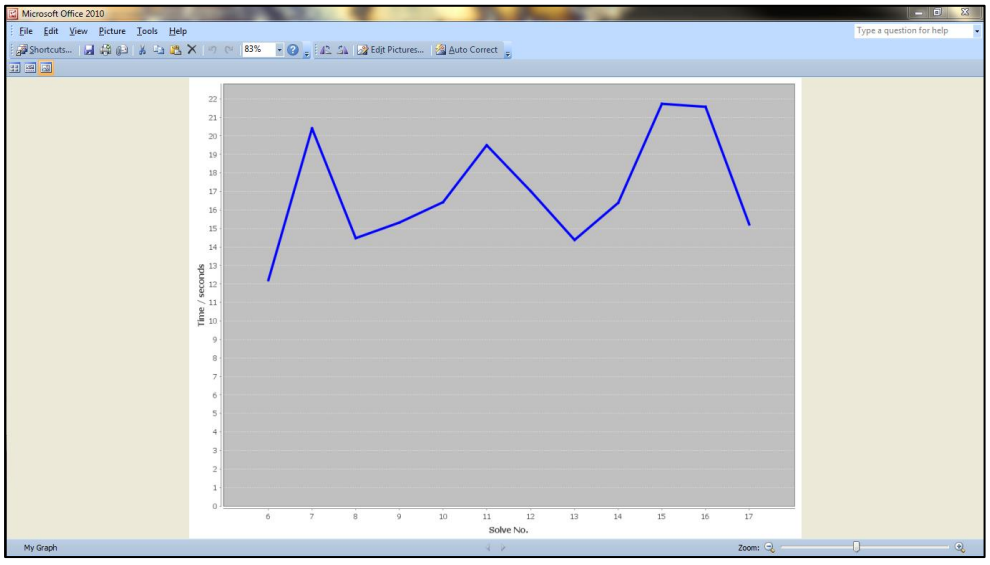
How to use Time Graph window

- You can access the time graph by selecting the *Show Time Graph* menu item (*Main window menu bar → View → Show Time Graph*)

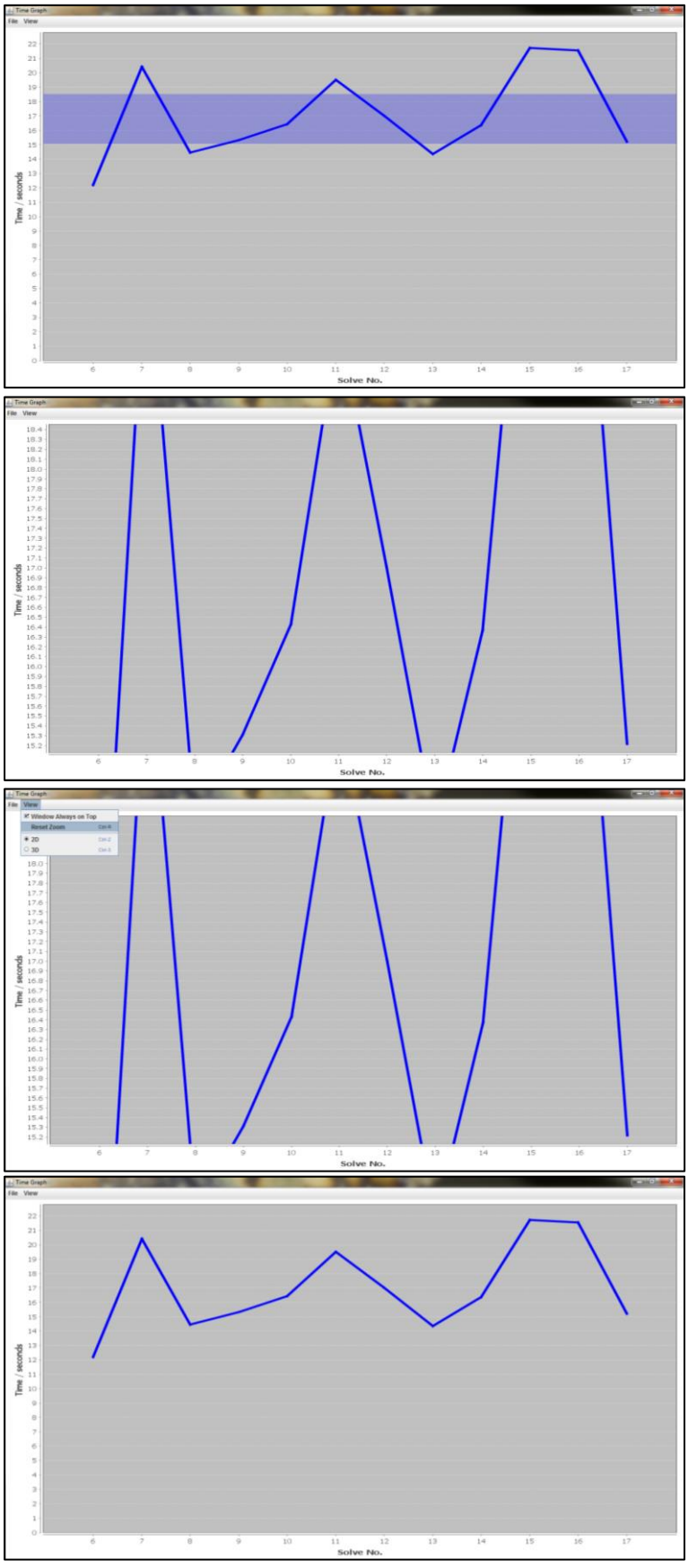
- **Saving a Graph**: You can save the graph as an image by following (*File → Save as Image*) and then choosing a location to save the graph. The image will be saved with the extension .png





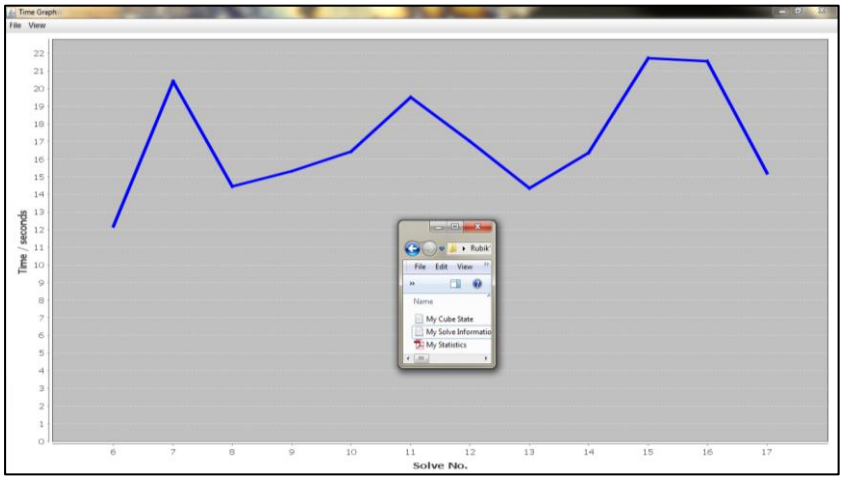


- **Changing Zoom**: You can zoom in on a section of the graph by clicking and dragging the mouse and highlighting the section. If you want to reset the zoom to the default zoom, then you can go (*View → Reset Zoom*).

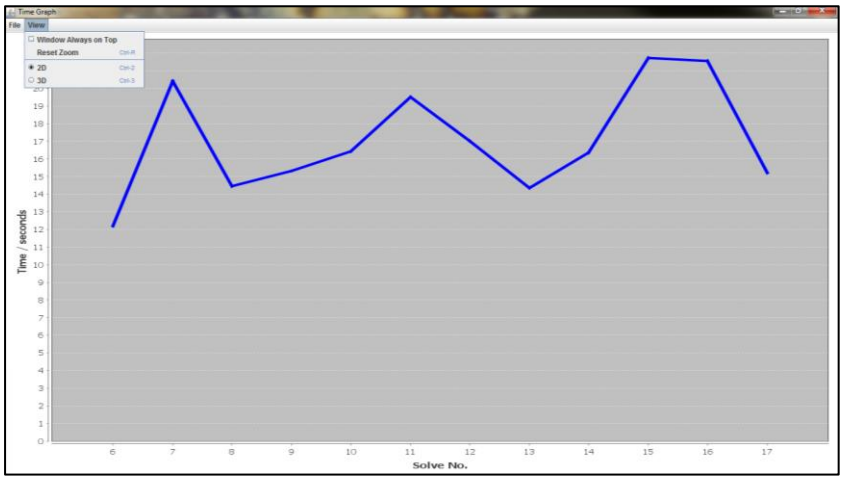


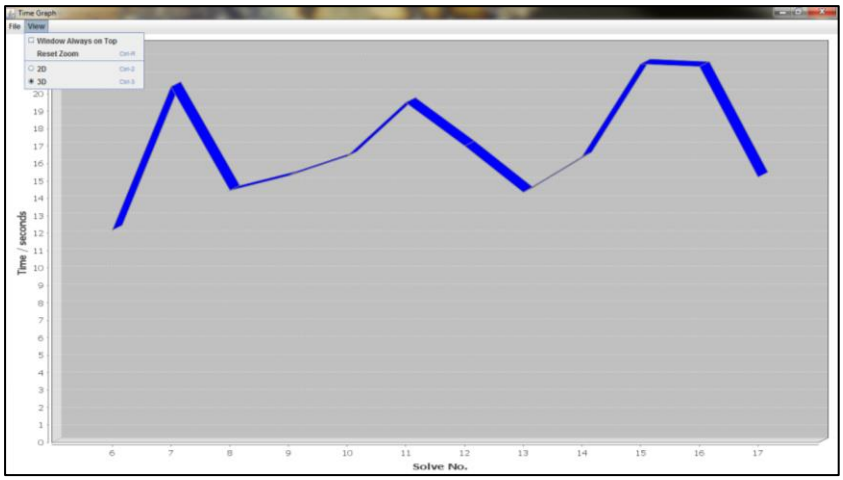
- **Change *Always on Top* option**: If you want the time graph window to be on top of all other windows, you can check the *Always on Top* menu item by following (*View → Window Always on Top*). If the box is checked, then the window will always be on top, i.e. no other window can be placed on top of this window, otherwise it will not always be on top, i.e. other windows can be placed on top of the window.

  (Here the *Window Always on Top* is not checked, so the Windows Explorer window can be placed on top of the *Time Graph* window)

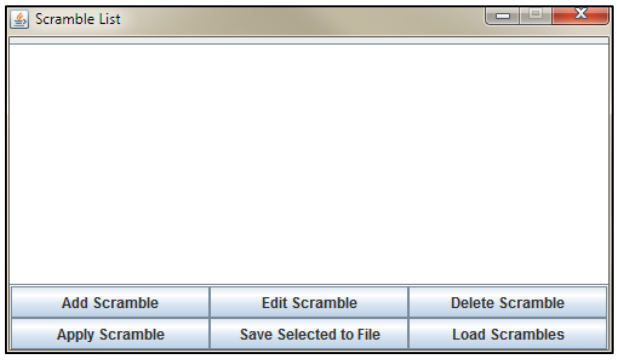


- **2D ↔ 3D**: You can select whether the graph is shown in a 2D or 3D fashion by selecting the corresponding item in the view menu (*View → 2D*, or, *View → 3D*).



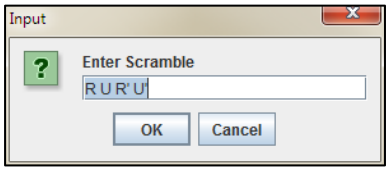


How to use the Scramble List window

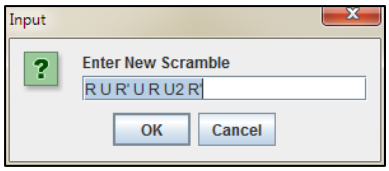


- You can access the Scramble List window by following (*Main window menu bar → View → Show Scramble List*)

- *Add Scramble Button*: After clicking this button, a window will be shown into which you can enter the scramble.



- *Edit Scramble Button*: After clicking, an input window will be shown with the original scramble in the text field; you can edit the scramble and submit it.



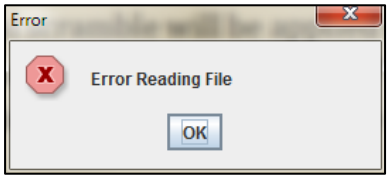
- *Delete Scramble Button*: After clicking, the selected scrambles will be removed from the list.

- *Apply Scramble Button*: The selected scramble will be applied to the virtual cube. For example, if the selected scramble consists of the text “R2 U2 F”, then the moves “Right Right Up Up Front” will be performed on the cube (instantly). This has the effect of scrambling or ‘mixing up’ the cube with the moves specified.

- *Save Selected to File*: A window will be shown asking you to choose a location for the file. The selected scrambles will be saved to a text file: one scramble per line.

- *Load Scrambles Button*: A window will be shown asking you to choose the file containing the scrambles you wish to load. After selecting the file, the scrambles stored in the text file will be appended to the list.

- If the file cannot be opened, then the following error message will be shown:

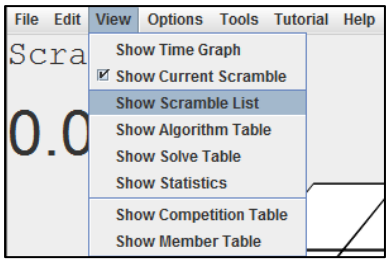


- Note: the data in this window will be discarded when the program is closed.

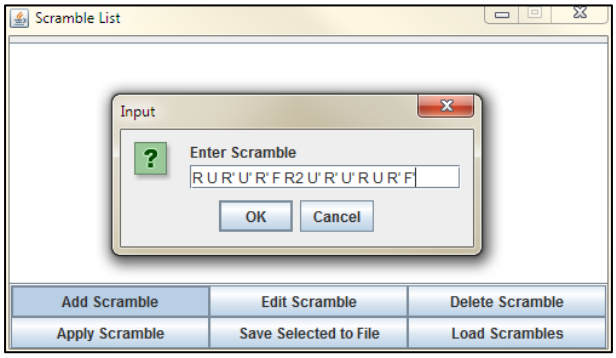
For more information, see *How to add a scramble, How to edit a scramble,* and *How to delete a scramble*.

How to add a scramble

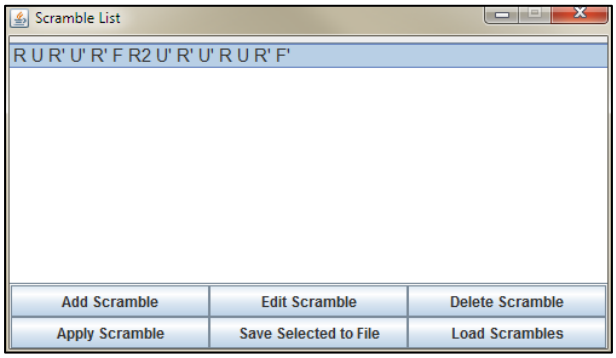
1. Open the Scramble List window (*Main window menu bar → View → Show Scramble List*)



2. Click the *Add Scramble* button and enter the desired scramble into the window that appears.



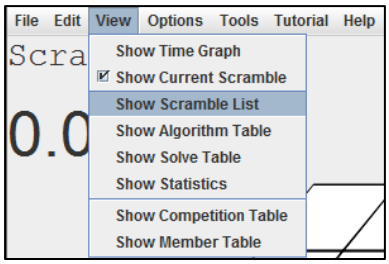
3. Click *OK* and the scramble will be appended to the list.



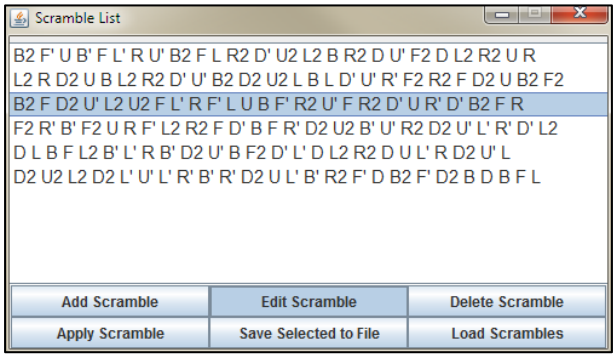
4. If the scramble you enter is blank or consists only of spaces, then nothing will be appended to the list. No other validation checks are run on the data you enter, so a typical scramble, such as "R U R' U'", is valid and an input of "$%^" is also valid.

How to edit a scramble

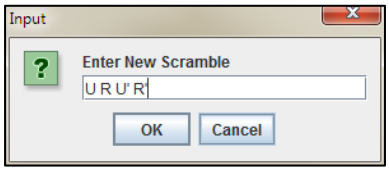
1. Open the Scramble List window (*Main window menu bar → View → Show Scramble List*)



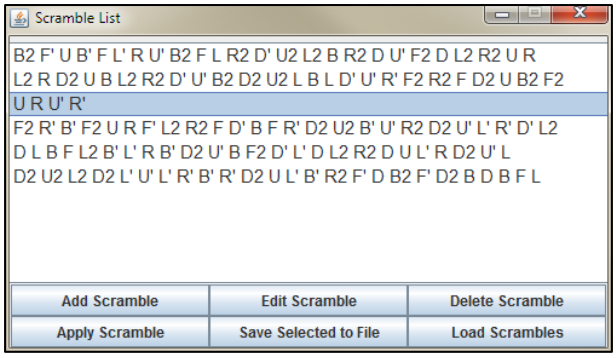
2. Select the scramble that you want to enter and click the *Edit Scramble* button.



3. Enter the new scramble in the window that appears.



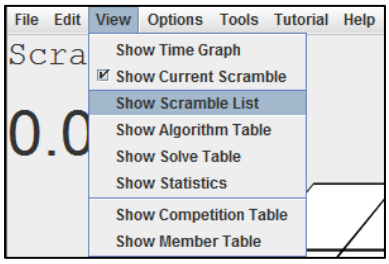
4. Click *OK* and the corresponding item in the list will be updated.



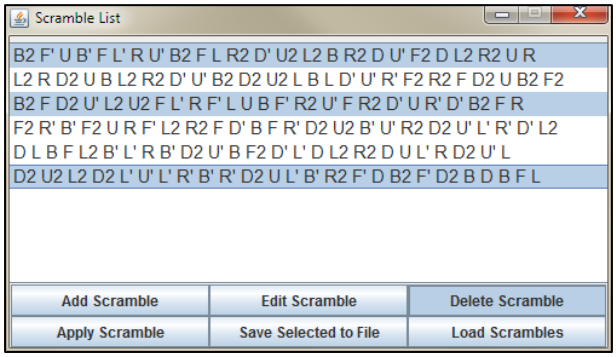
5. If the scramble you enter is blank or consists only of spaces, then the selected item will not be changed. No other validation checks are run on the data you enter, so a typical scramble, such as “R U R’ U’”, is valid and an input of “$%^” is also valid.

How to delete a scramble

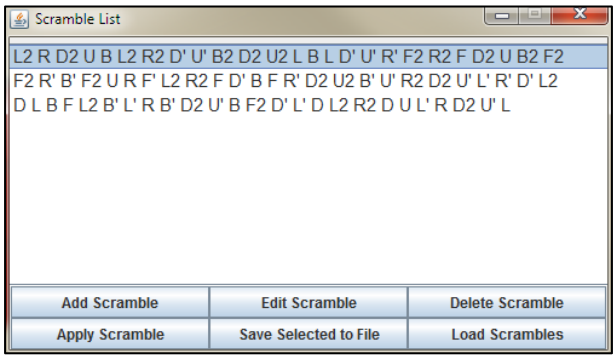
1. Open the Scramble List window (*Main window menu bar → View → Show Scramble List*)



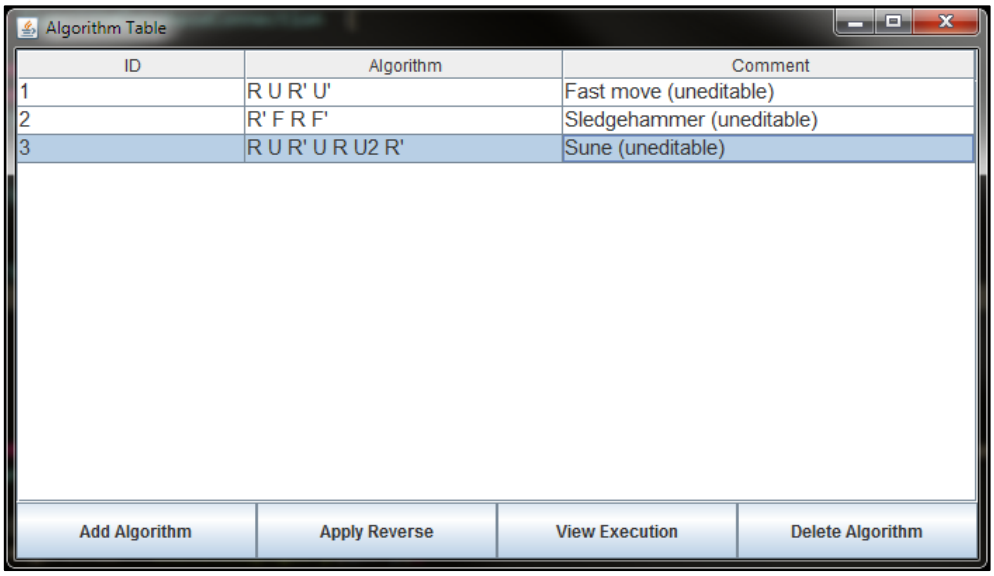
2. Select the scramble/s that you want to delete and click the *Delete Scramble* button.



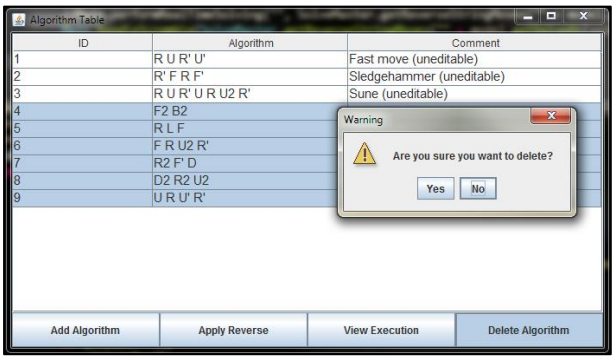
3. The selected scrambles will be removed from the list.



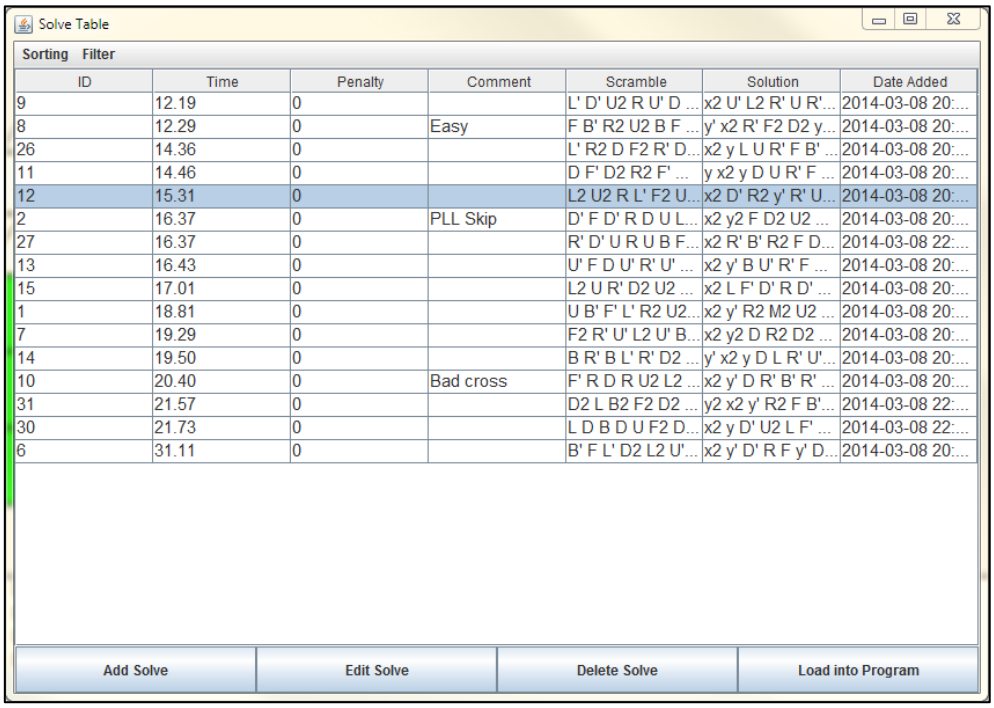
How to use the Algorithm Table



- You can access the Solve Table by following (*Main window menu bar → View → Show Algorithm Table*)
- The three algorithms shown are preset algorithms; these rows cannot be edited or deleted.
- *Add Algorithm Button*: A row will be added to the table.
- To edit a cell, double-click on the cell and then start typing. You cannot edit the ID column.
- *Apply Reverse Button*: The 'reverse' of the selected algorithm will be applied to the cube. For example, if the algorithm was "R2 B2 L U F", then "F' U' L' B2 R2" would be applied to the cube (instantly). This can be used so that you can practise performing algorithms.
- *View Execution*: The selected algorithm will be performed in real-time on the virtual cube.
- *Delete Algorithm Button*: The selected algorithms will be removed from the table.
    - If more than five rows are selected, then a warning message will be shown asking you whether you are sure you want to continue.

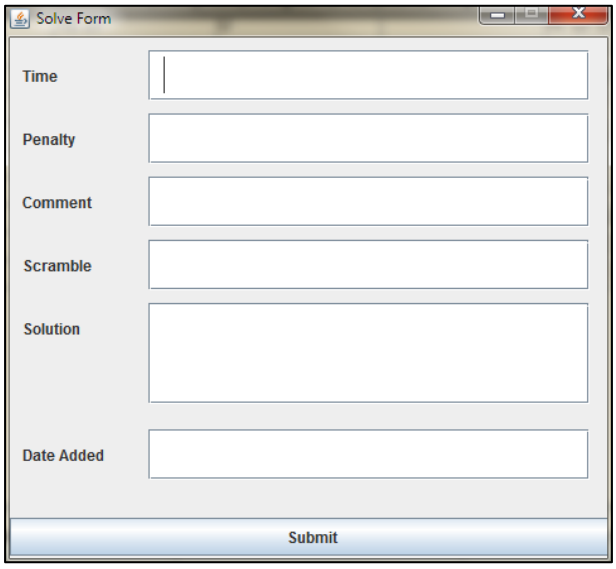


How to use the Solve Table



- You can access the Solve Table by following (*Main window menu bar → View → Show Solve Table*)

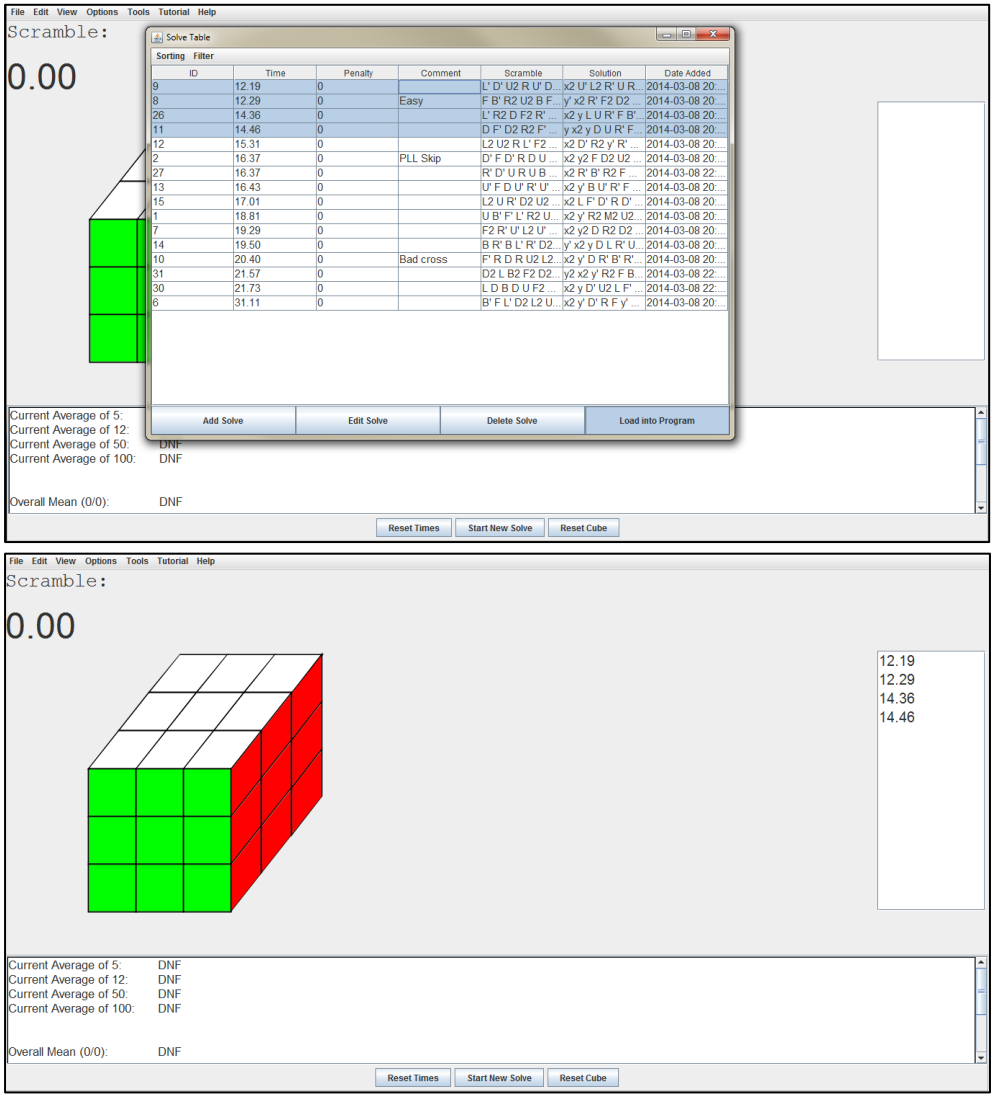
- *Add Solve Button*: The *Solve Form* window will be shown into which you can enter information about a solve.



- *Edit Solve Button*: The *Solve* Form window will be shown with the selected solve's information in the text fields.

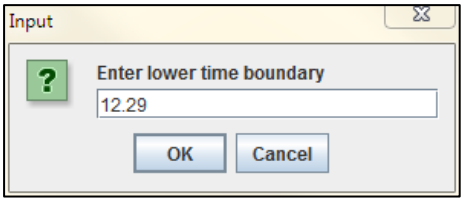
- *Delete Solve Button*: The selected rows from the table will be removed.

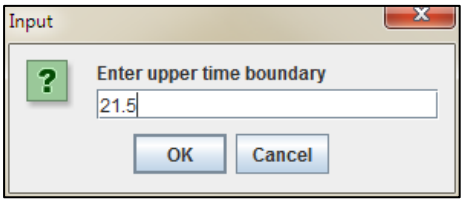
- *Load into Program Button*: the selected solves will be appended into the main window's list of solves. The time, penalty, comment, scramble and solution will be transferred to the list of solves.

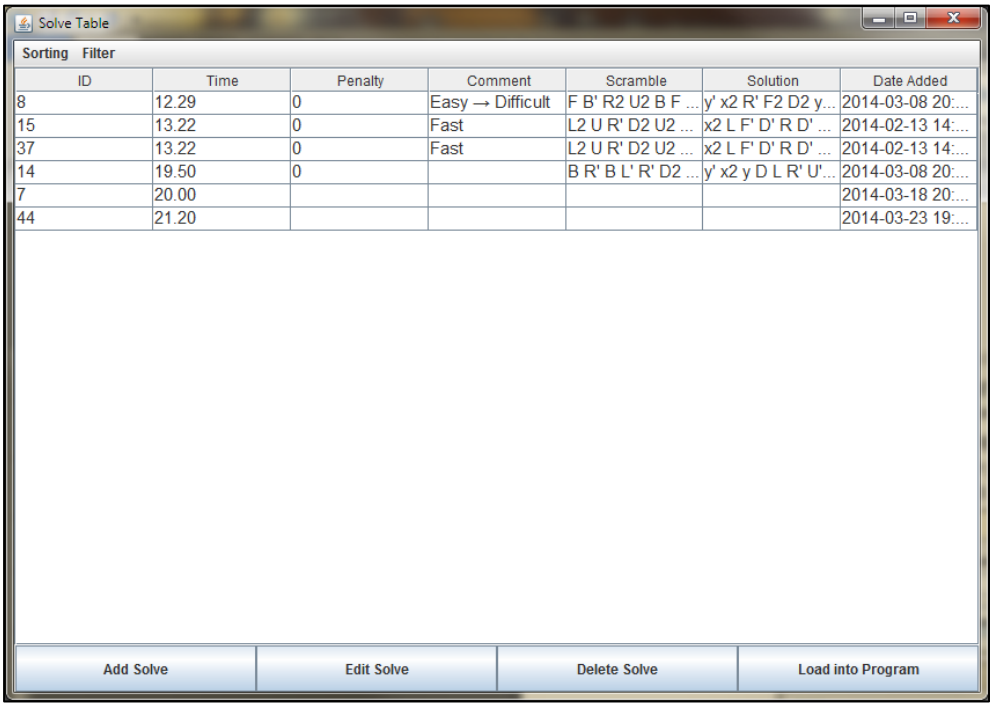


- *Sorting Menu*: There are six menu items in the *Sorting* menu:
  - **Sort by Time (Ascending)**: The rows in the table will be sorted with the fastest solve in the first row and the slowest solve in the bottom row.
  - **Sort by Time (Descending)**: The rows in the table will be sorted with the fastest solve in the first row and the slowest solve in the bottom row.
  - **Sort by Date (Ascending)**: The rows in the table will be sorted with the earliest date in the first row and the latest date in the bottom row.
  - **Sort by Date (Descending)**: The rows in the table will be sorted with the latest date in the first row and the earliest date in the bottom row.
  - **Sort by ID (Ascending)**: The rows in the table will be sorted with the least ID in the first row and the greatest ID in the bottom row.
  - **Sort by ID (Descending)**: The rows in the table will be sorted with the greatest ID in the first row and the least ID in the bottom row

- *Filter Menu*: There are two items in the *Filter* menu:
  - **Filter by Time**: Clicking this menu item triggers a two-step input process. In the first pop-up window, you enter the lower boundary for the times to be shown; in the second pop-up window, you enter the upper boundary for the times to be shown. For example, if you enter the following data in the pop-up windows, the times shown in the table will be between 12.29 seconds and 21.50 seconds inclusive.



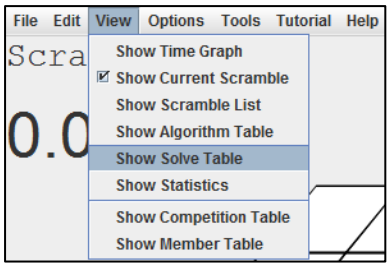




  - **Remove Filter**: Clicking this menu item clears any filter present on the data, i.e. all data will be shown in the table.

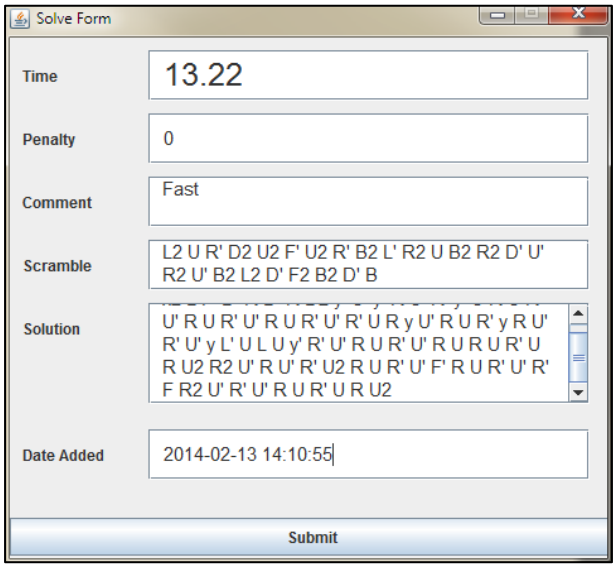
How to add a solve to the database

1. Open the Solve Table (*Main window menu bar → View → Show Solve Table*)

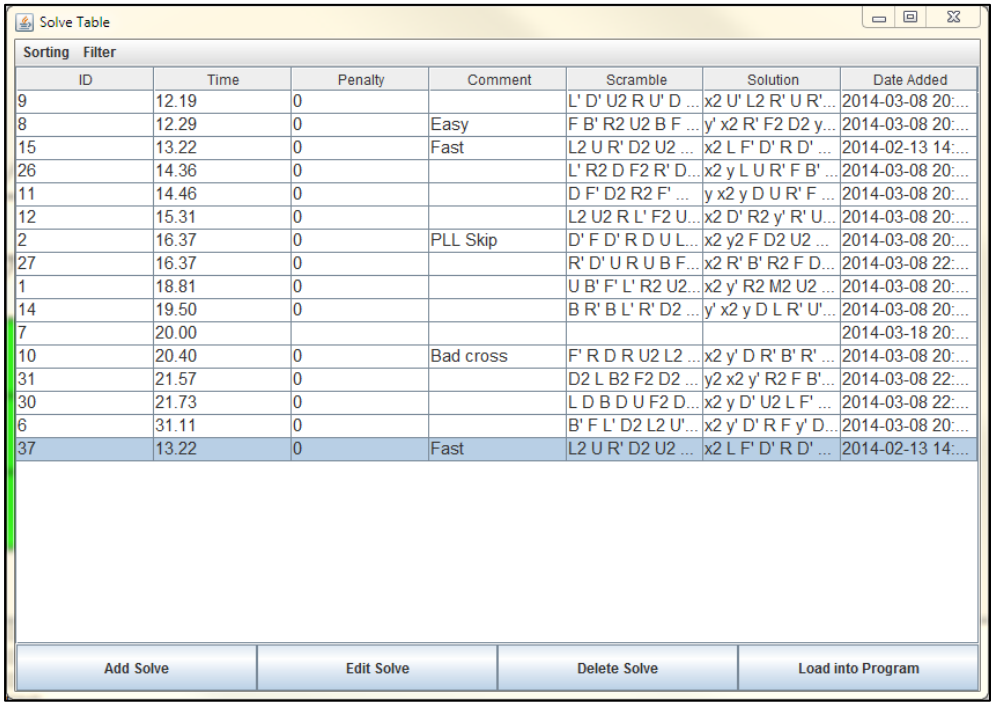


2. Click the *Add Solve* button.

3. The *Solve* Form window will appear. Enter the desired information into the form and click the *Submit* button.

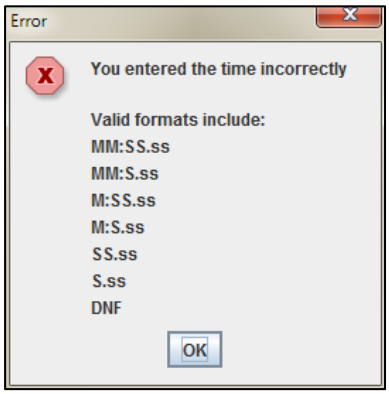


4. A row containing the information will be appended to the table.

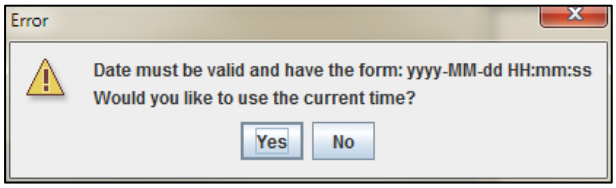


ID	Time	Penalty	Comment	Scramble	Solution	Date Added
9	12.19	0		L' D' U2 R U' D ...	x2 U' L2 R' U R'...	2014-03-08 20:...
8	12.29	0	Easy	F B' R2 U2 B F ...	y' x2 R' F2 D2 y...	2014-03-08 20:...
15	13.22	0	Fast	L2 U R' D2 U2 ...	x2 L F' D' R D' ...	2014-02-13 14:...
26	14.36	0		L' R2 D F2 R' D...	x2 y L U R' F B' ...	2014-03-08 20:...
11	14.46	0		D F' D2 R2 F' ...	y x2 y D U R' F ...	2014-03-08 20:...
12	15.31	0		L2 U2 R L' F2 U...	x2 D' R2 y' R' U...	2014-03-08 20:...
2	16.37	0	PLL Skip	D' F D' R D U L...	x2 y2 F D2 U2 ...	2014-03-08 20:...
27	16.37	0		R' D' U R U B F...	x2 R' B' R2 F D...	2014-03-08 22:...
1	18.81	0		U B' F' L' R2 U2...	x2 y' R2 M2 U2 ...	2014-03-08 20:...
14	19.50	0		B R' B L' R' D2 ...	y' x2 y D L R' U'...	2014-03-08 20:...
7	20.00					2014-03-18 20:...
10	20.40	0	Bad cross	F' R D R U2 L2 ...	x2 y' D R' B' R' ...	2014-03-08 20:...
31	21.57	0		D2 L B2 F2 D2 ...	y2 x2 y' R2 F B'...	2014-03-08 22:...
30	21.73	0		L D B D U F2 D...	x2 y D' U2 L F' ...	2014-03-08 22:...
6	31.11	0		B' F L' D2 L2 U'...	x2 y' D' R F y' D...	2014-03-08 20:...
37	13.22	0	Fast	L2 U R' D2 U2 ...	x2 L F' D' R D' ...	2014-02-13 14:...

5. If the data in the *time field* is invalid, then the following error message will be shown:



6. If the data in the *date added field* is invalid, then the following warning message will be shown:

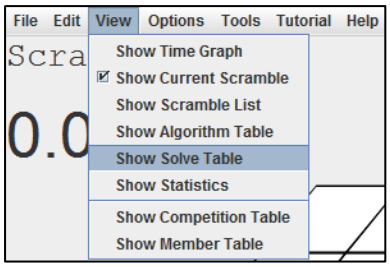


By selecting *Yes*, the information will be submitted and the current time will be used for the *Date Added* field. If *No* is selected, then you will have to enter a different datum for the date.

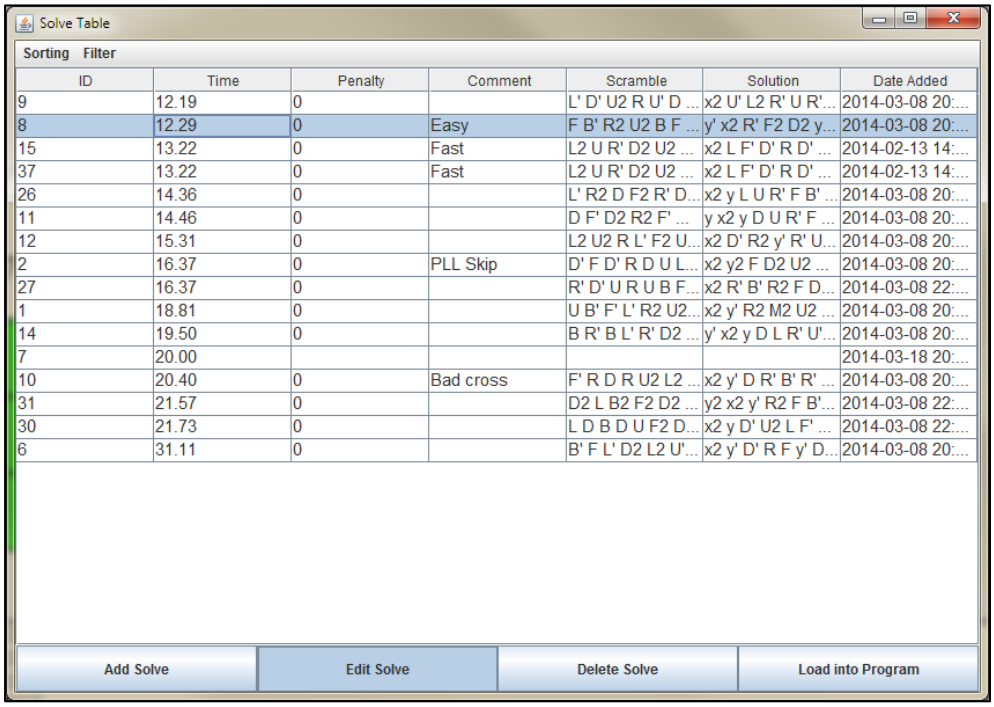
See the *Data Input Guidelines* section for help on how to enter valid data.

How to edit a solve in the database

1. Open the Solve Table (*Main window menu bar → View → Show Solve Table*)

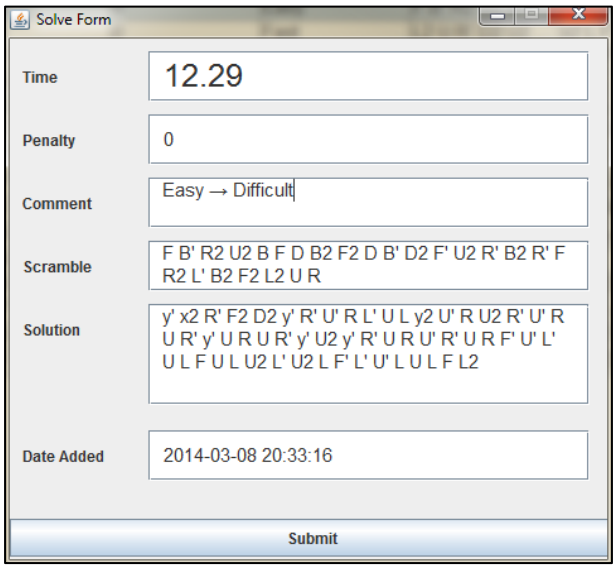


2. Select the solve you want to edit, and then click the *Edit Solve* button.

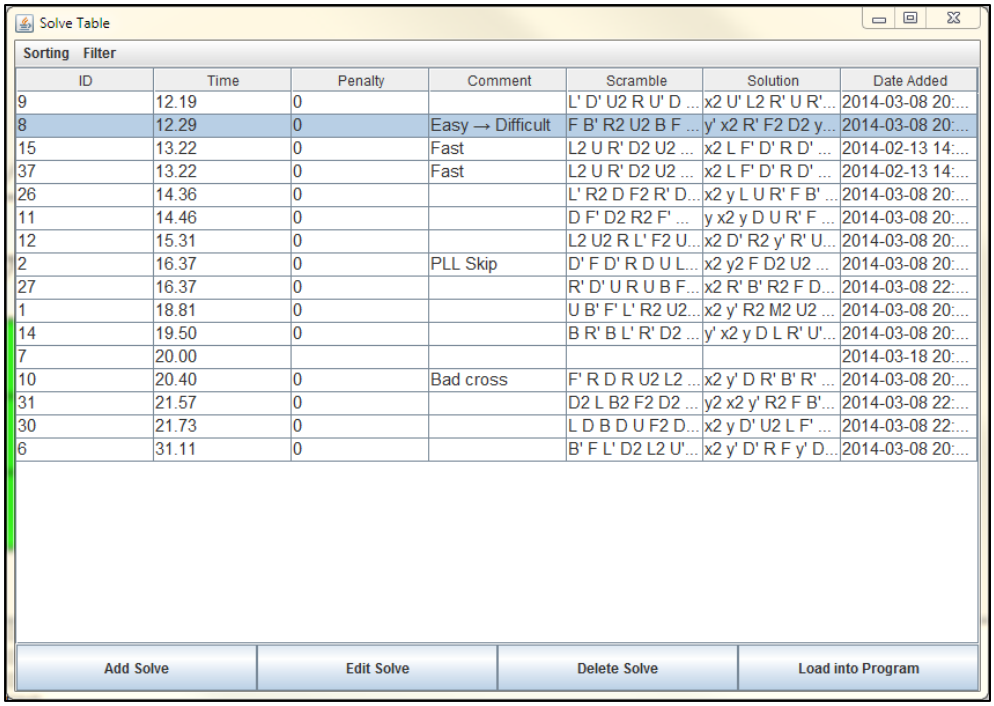


ID	Time	Penalty	Comment	Scramble	Solution	Date Added
9	12.19	0		L' D' U2 R U' D ...	x2 U' L2 R' U R'...	2014-03-08 20:...
8	12.29	0	Easy	F B' R2 U2 B F ...	y' x2 R' F2 D2 y...	2014-03-08 20:...
15	13.22	0	Fast	L2 U R' D2 U2 ...	x2 L F' D' R D' ...	2014-02-13 14:...
37	13.22	0	Fast	L2 U R' D2 U2 ...	x2 L F' D' R D' ...	2014-02-13 14:...
26	14.36	0		L' R2 D F2 R' D...	x2 y L U R' F B' ...	2014-03-08 20:...
11	14.46	0		D F' D2 R2 F' ...	y x2 y D U R' F ...	2014-03-08 20:...
12	15.31	0		L2 U2 R L' F2 U...	x2 D' R2 y' R' U...	2014-03-08 20:...
2	16.37	0	PLL Skip	D' F D' R D U L...	x2 y2 F D2 U2 ...	2014-03-08 20:...
27	16.37	0		R' D' U R U B F...	x2 R' B' R2 F D...	2014-03-08 22:...
1	18.81	0		U B' F' L' R2 U2...	x2 y' R2 M2 U2 ...	2014-03-08 20:...
14	19.50	0		B R' B L' R' D2 ...	y' x2 y D L R' U'...	2014-03-08 20:...
7	20.00					2014-03-18 20:...
10	20.40	0	Bad cross	F' R D R U2 L2 ...	x2 y' D R' B' R' ...	2014-03-08 20:...
31	21.57	0		D2 L B2 F2 D2 ...	y2 x2 y' R2 F B'...	2014-03-08 22:...
30	21.73	0		L D B D U F2 D...	x2 y D' U2 L F' ...	2014-03-08 22:...
6	31.11	0		B' F L' D2 L2 U'...	x2 y' D' R F y' D...	2014-03-08 20:...

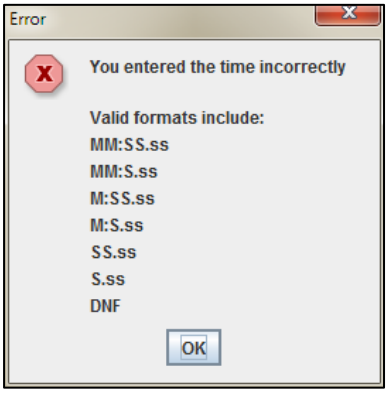
3. The *Solve Form* window will appear; the text fields will be populated with the information of the selected solve. Edit the desired information the click the *Submit* button.



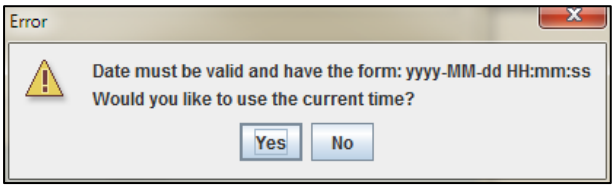
4. The corresponding row in the table will be updated.



5. If the data in the *time field* is invalid, then the following error message will be shown:



6. If the data in the *date added field* is invalid, then the following warning message will be shown:

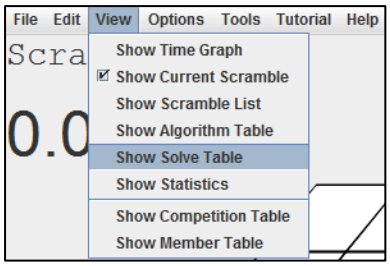


By selecting *Yes*, the information will be submitted and the current time will be used for the *Date Added* field. If *No* is selected, then you will have to enter a different datum for the date.

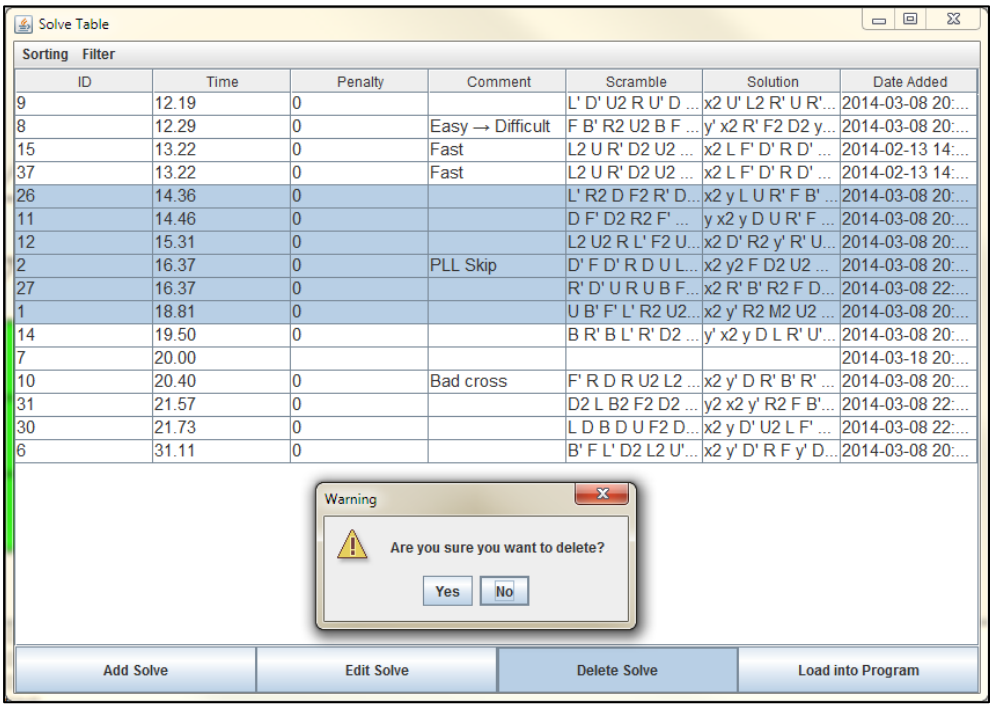
See the *Data Input Guidelines* section for help on how to enter valid data.

How to delete a solve from the database

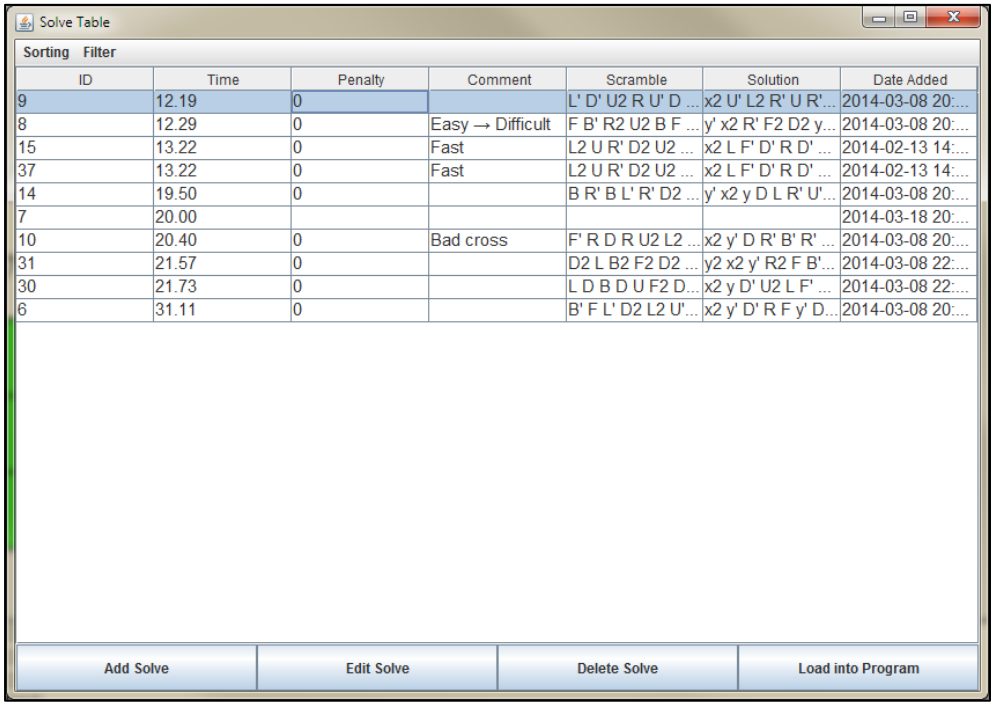
1. Open the Solve Table (*Main window menu bar → View → Show Solve Table*)



2. Select the solve that you want to delete and click the *Delete Solve* button.
   - If more than four rows are selected, then a warning message with the text, “Are you sure you want to delete?” will be shown.

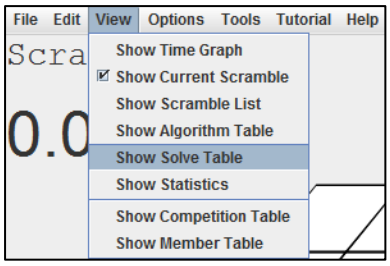


3. The selected rows will be removed from the table.

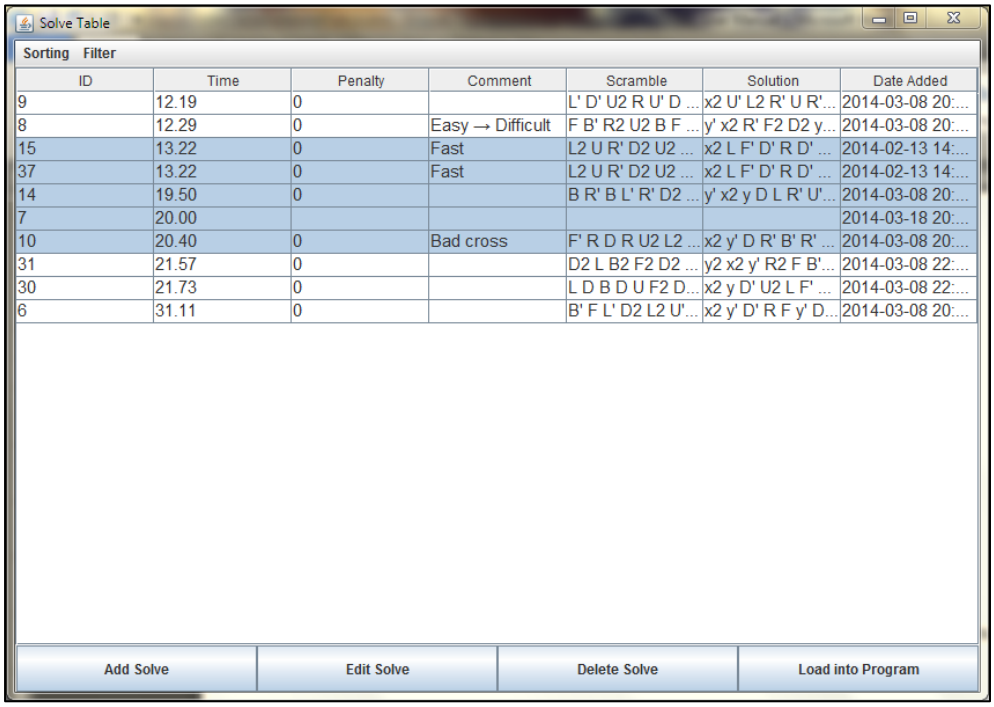


How to load a solve from the database into the main window

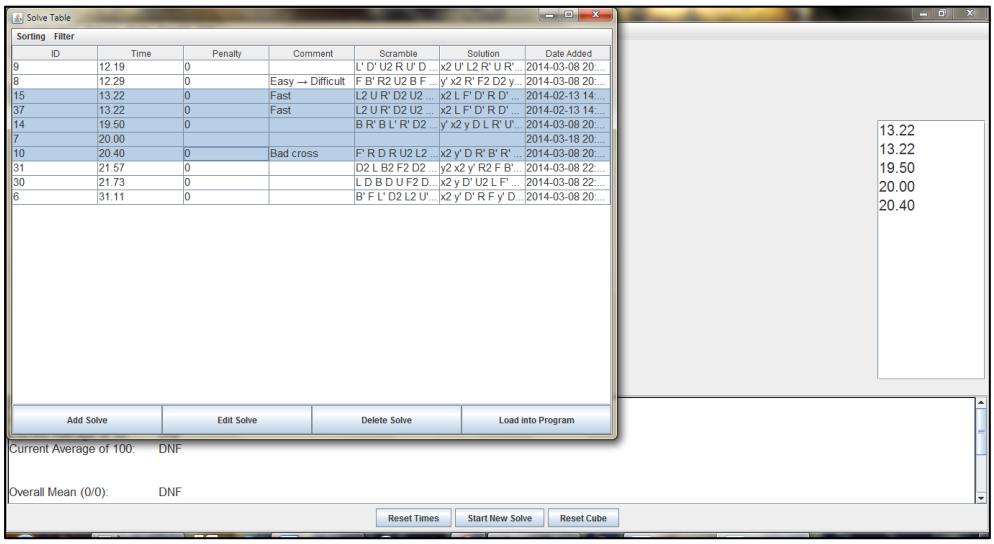
1. Open the Solve Table (*Main window menu bar → View → Show Solve Table*)



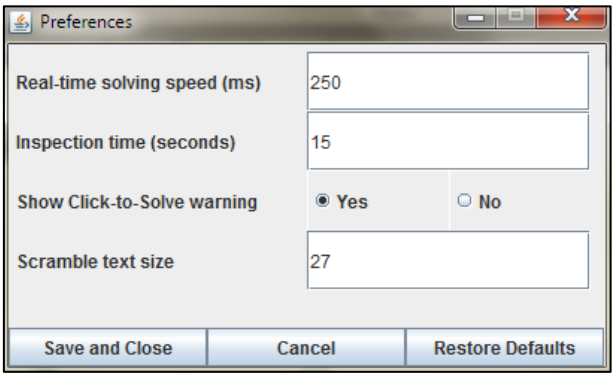
2. Select the solves that you want to load into the main window.



3. Click the *Load into Program* button. The solves, with their respective information, will be loaded into the main window's solve list.



How to use the Preferences window



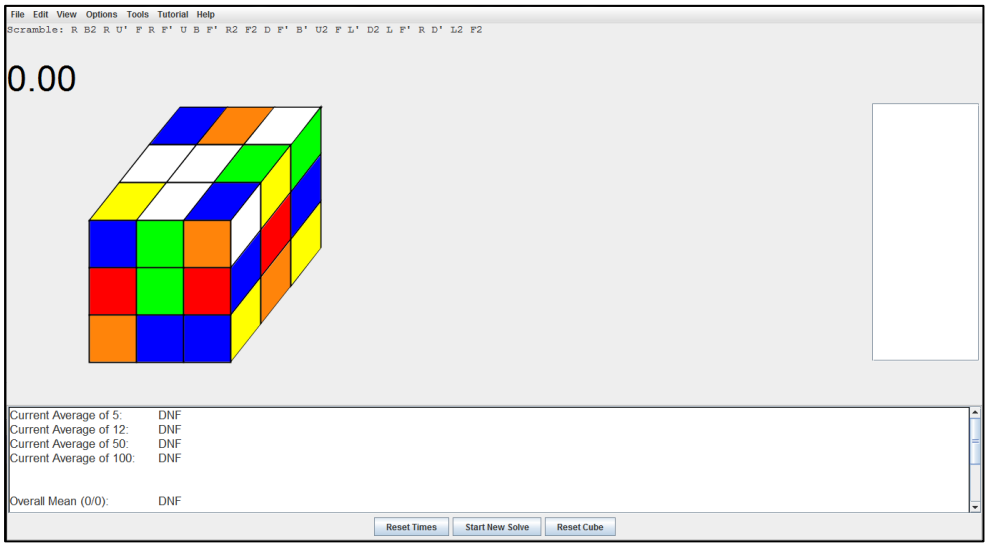
- You can access the Preferences window by following (*Main window menu bar* → *Options* → *Preferences*)

- *Real-time solving speed* field: The value in this field indicates the time taken per move when automatic execution (see page 6 for information on automatic execution) is taking place. For example, if the value was '125', then a move would be performed on the cube every 125 milliseconds, or every 0.125 seconds. The value entered can be between 1 and 9999 inclusive.

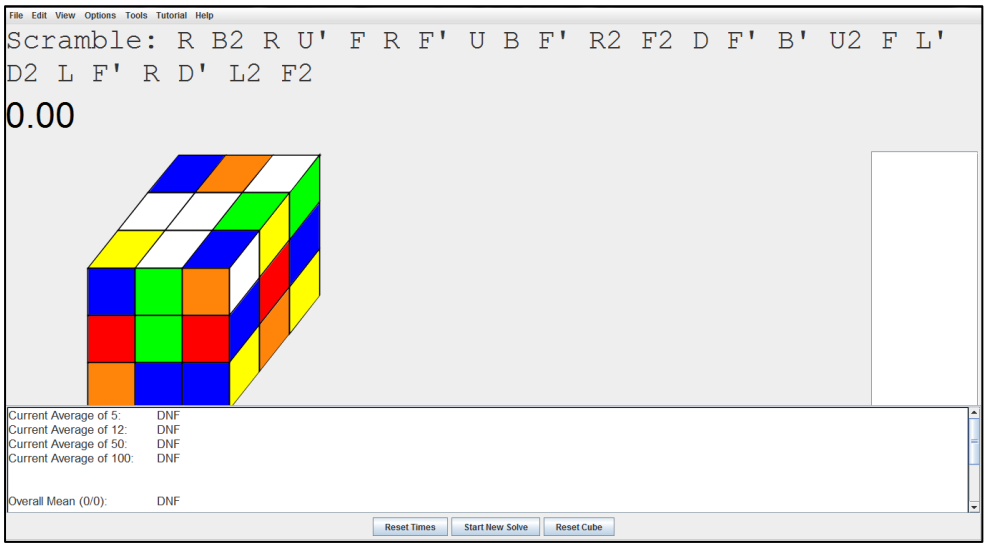
- *Inspection time (seconds)* field: The value in this field indicates how many seconds you have to inspect the cube before starting a solve. For example, if you click the *Start New Solve* button in the main window, the cube will be scrambled and the inspection timer will start with *x* seconds (where *x* is the value in the *Inspection time* field) and will begin to count down. You can inspect the cube while the timer is counting down. When the counter reaches zero, a two second penalty will be applied; after two further seconds, the solve will be disqualified. The value entered can be between 1 and 99 inclusive.

- *Show Click-to-Solve Warning* option: If *Yes* is selected, then an information/warning window will be shown after you click the *Solve Piece* menu item in the main window. If *No* is selected, then no information/warning window will be shown. It is recommended that this option remain *Yes*. If the *Yes* option is selected, then after clicking the *Solve Piece* menu item, this window will be shown.

- *Scramble text size* field: The value in this field indicates the size of the scramble-text in the main window. The value entered can be between 1 and 99 inclusive.
  This window shows a *Scramble text size* value of 15:



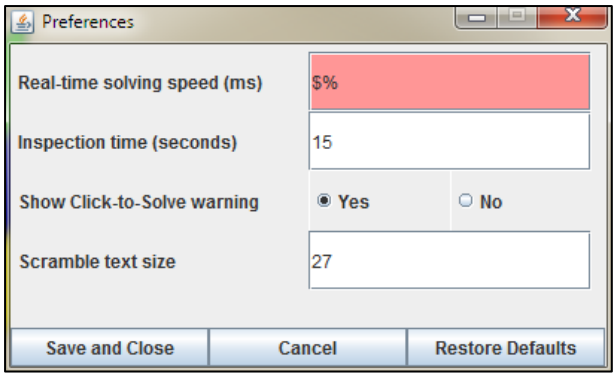
  This window shows a *Scramble text size* value of 40:



- *Restore Defaults* button: After clicking this button, the default preferences will be restored and saved:
  - *Real-time solving speed (ms)*: 250
  - *Inspection time (seconds)*: 15
  - *Show Click-to-Solve warning*: Yes
  - *Scramble text size*: 27

- *Cancel* Button: After clicking this button, the window will close and any changes will be discarded.

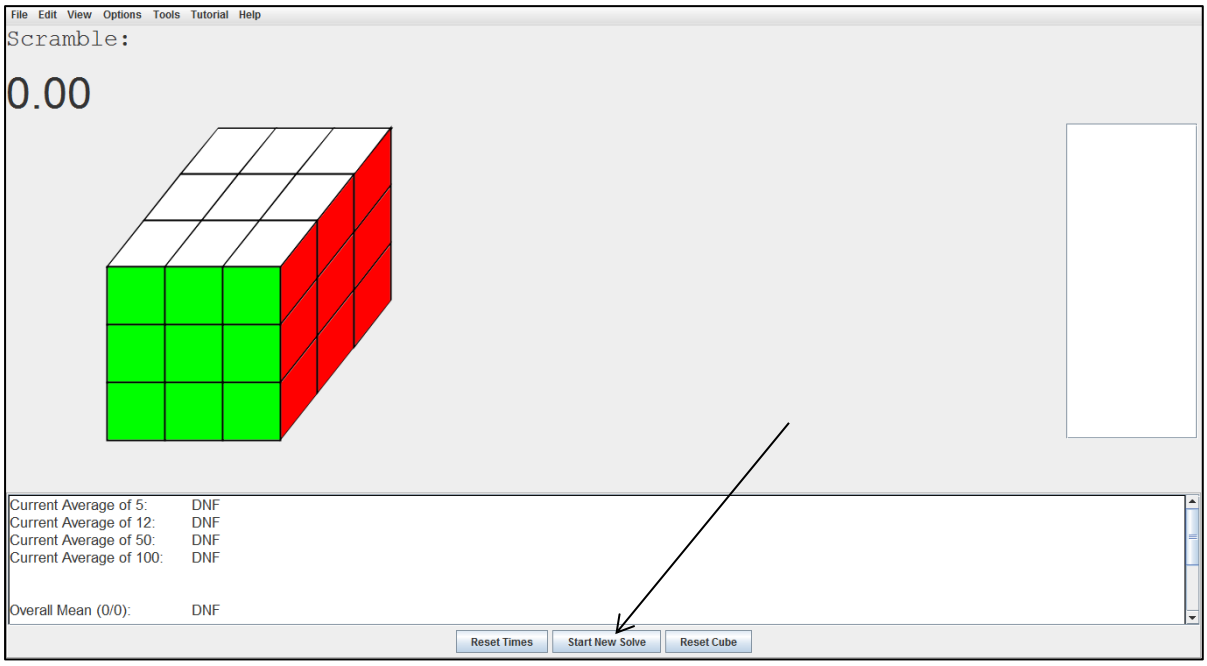
- *Save and Close* Button: After clicking this button, the information entered will be saved. If any data is invalid, the corresponding text field will become red:



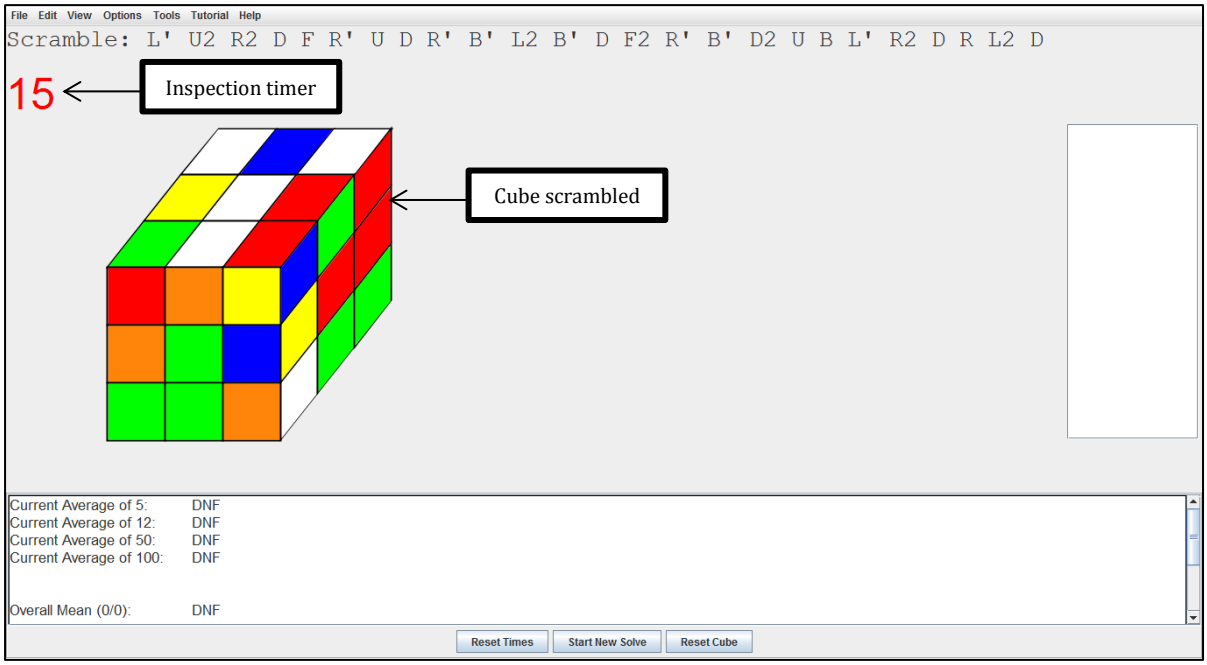
See the *Data Input Guidelines* section for help on how to enter valid data.

How to start a solve

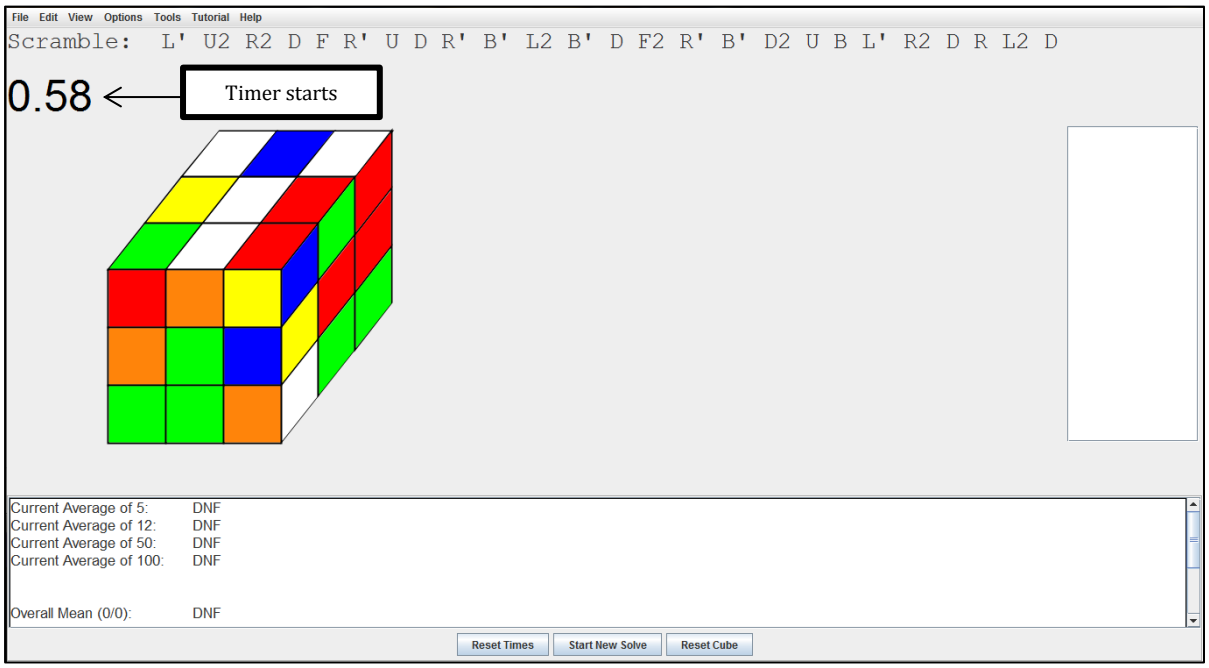
1. In the main window, click the *Start New Solve* button.



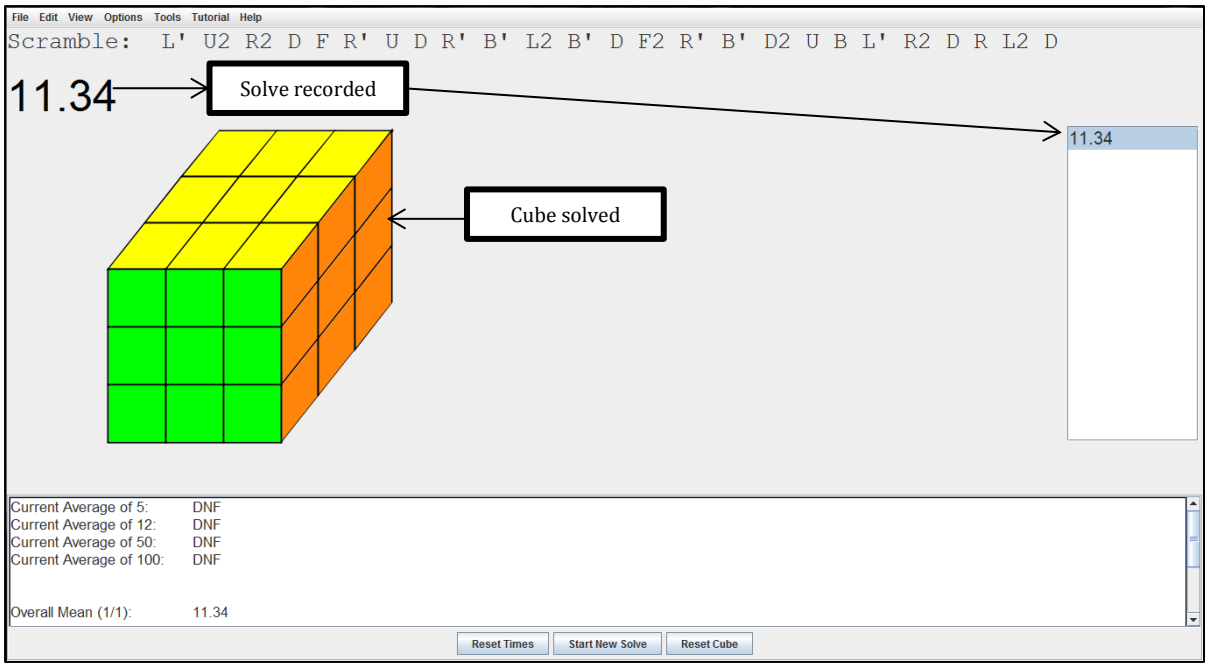
2. The cube will be scrambled randomly and the inspection timer will start counting down. You cannot perform moves while the inspection timer is running, but you are permitted to perform rotations.



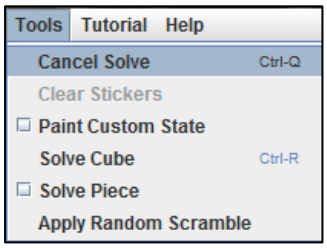
3. To start the timer (and the solve), press and release spacebar. After releasing the spacebar, the solve timer will start and you can begin performing moves.



4. The timer will stop and the solve will be recorded in the solve list after you have solved the cube.

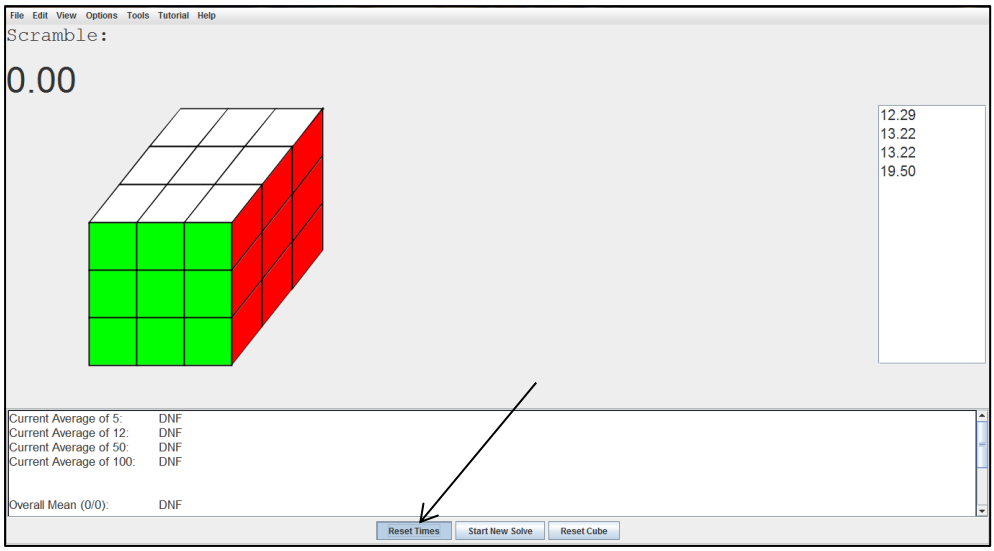


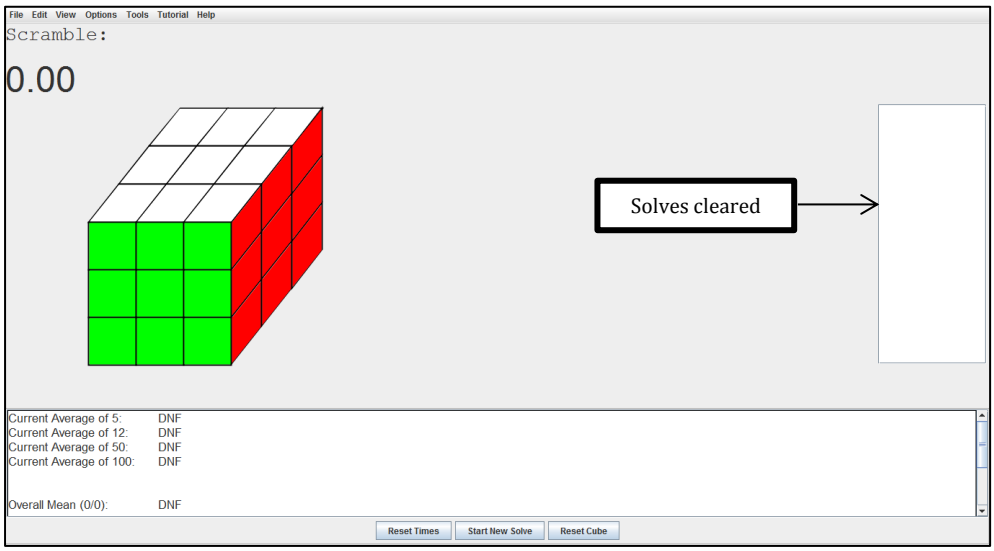
5. If you wish to cancel the current solve, i.e. reset the timer and continue with other tasks, you can click the *Cancel Solve* menu item (*Main window menu bar → Tools → Cancel Solve*).



How to delete all solves in the solve list

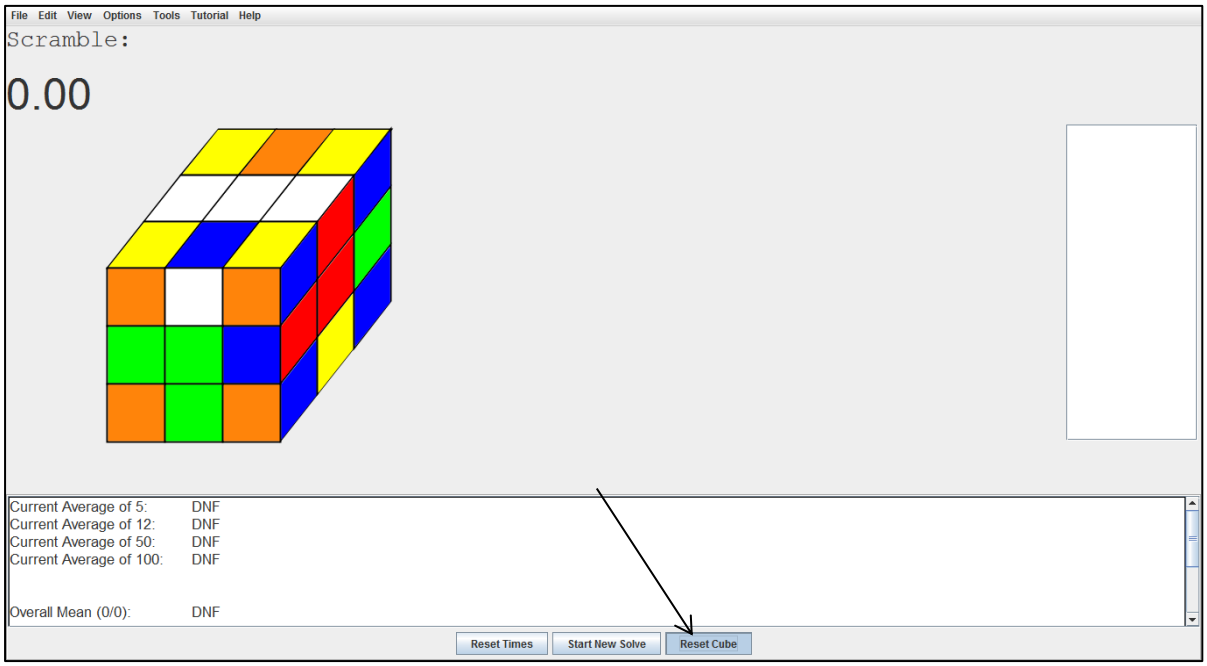
1. Click the *Reset Times* button and all solves in the solve list will be cleared.

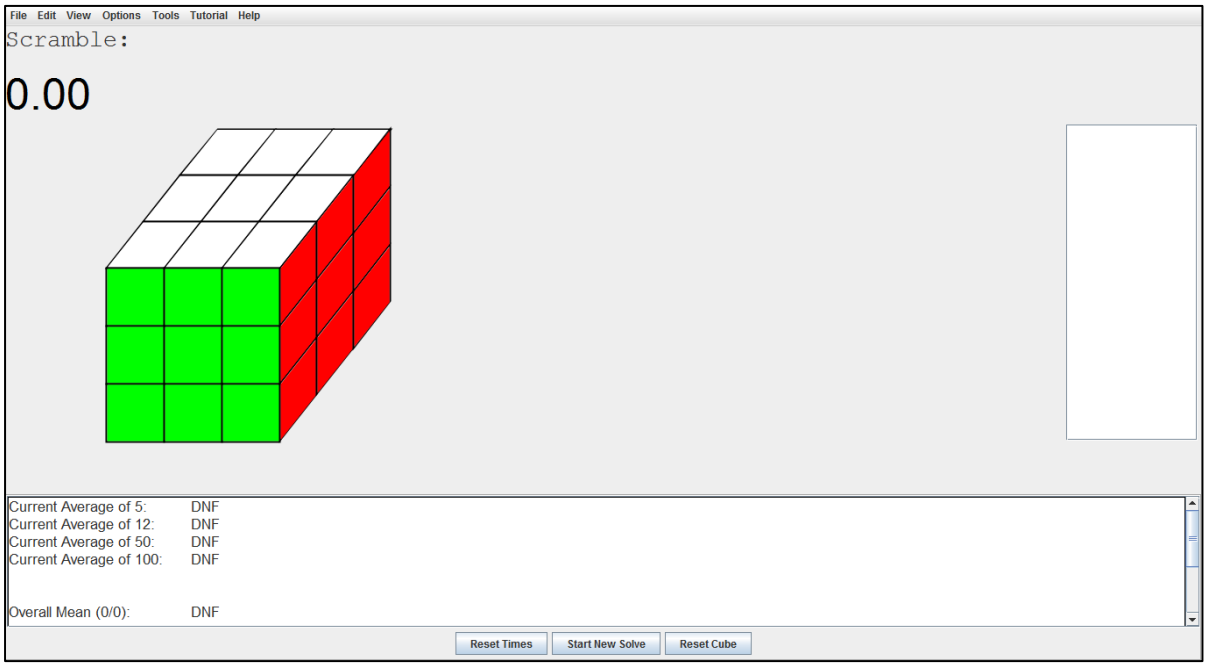




How to reset the cube to a solved state

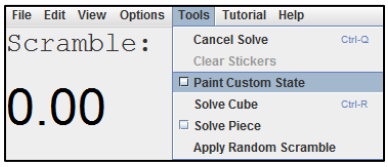
1. Click the *Reset Cube* button and the cube will be reset to a solved state with white on top and green on front.

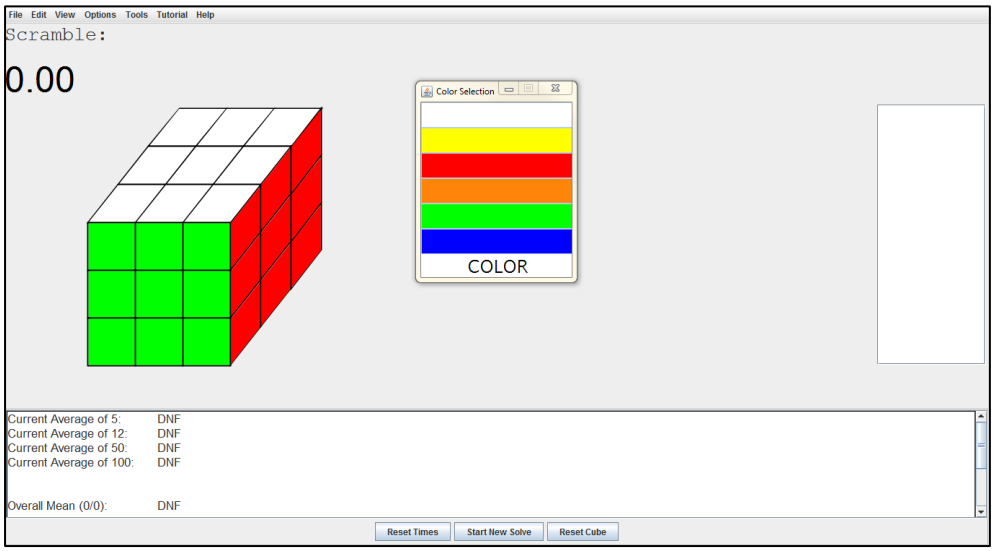




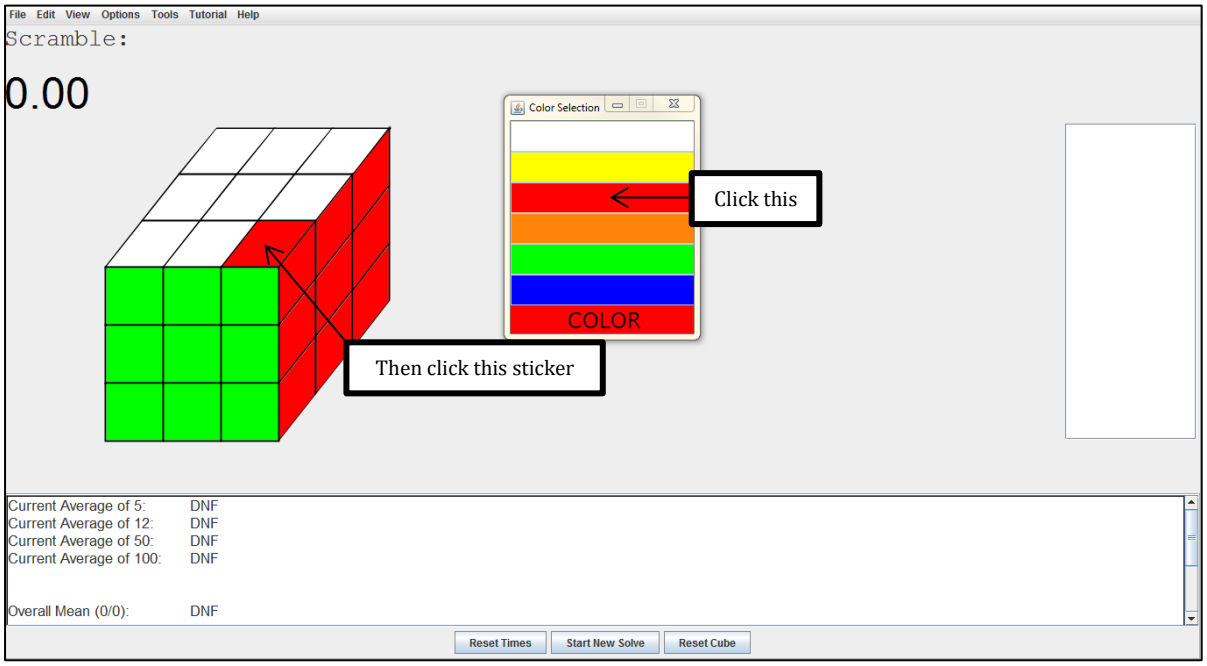
How to paint a custom state

1. *Main window menu bar → Tools → Paint Custom State*. The *Color Selection* window will then open.

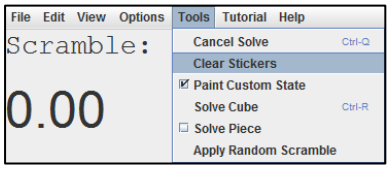


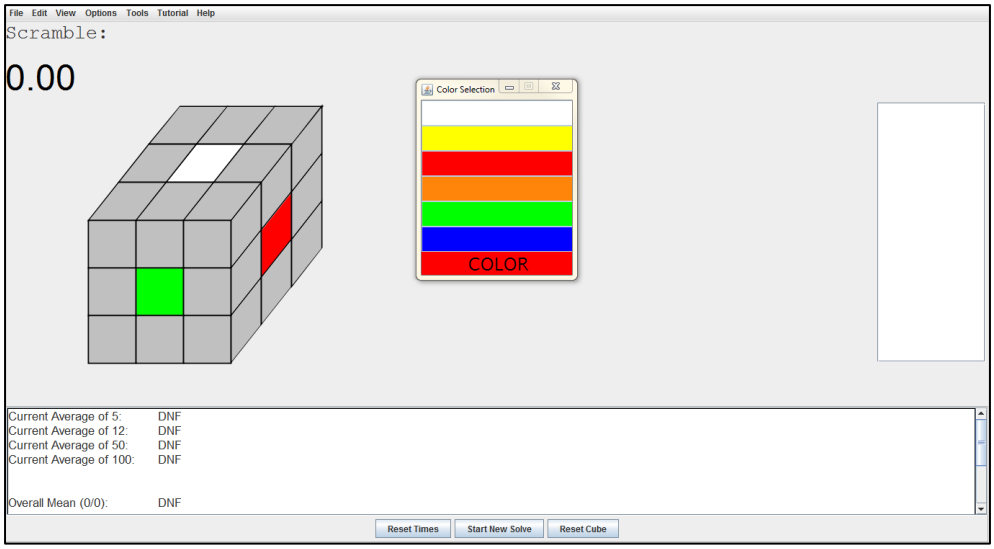


2. If, for example, you want to paint a sticker red, click on the red rectangle in the Color Selection window, then click on the sticker that you want to paint red.



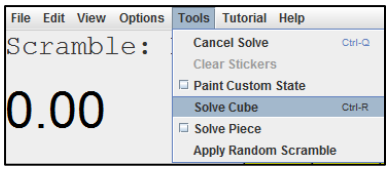
3. To make painting easier, you can first clear all stickers so that they all show grey. You can do this by following (*Main window menu bar → Tools → Clear Stickers*).



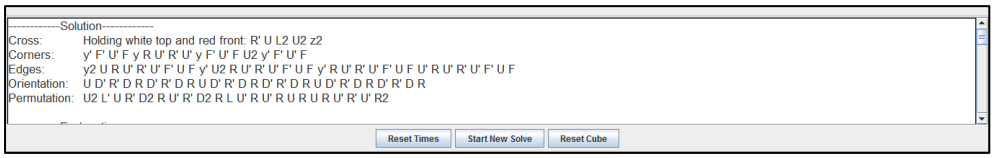


How to generate a solution for the current state

1. *Main window menu bar → Tools → Solve cube*



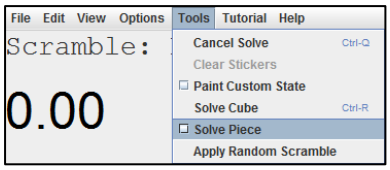
2. A solution will be generated and shown in the text area at the bottom of the main window. The solution will be performed automatically on the cube.



3. If you click the *Solve Cube* menu item while a solve is in progress, the solve will be cancelled and then the solution will be generated.

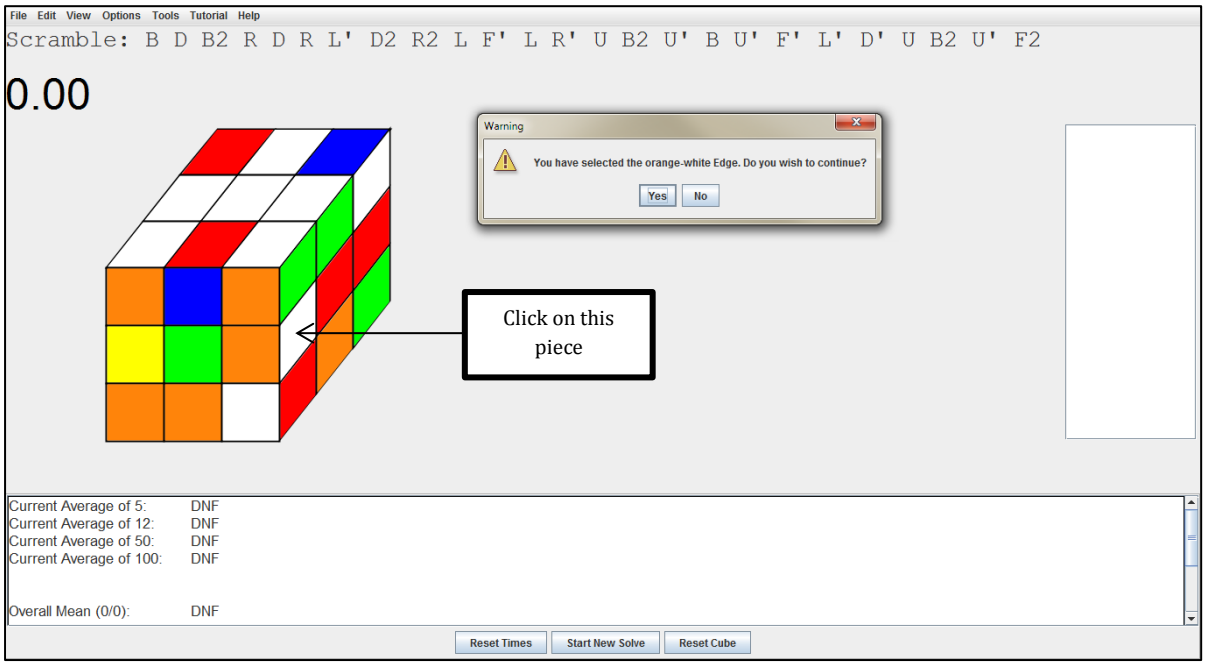
How to solve a selected piece

1. Click the *Solve Piece* menu item (*Main window menu bar → Tools → Solve Piece*)

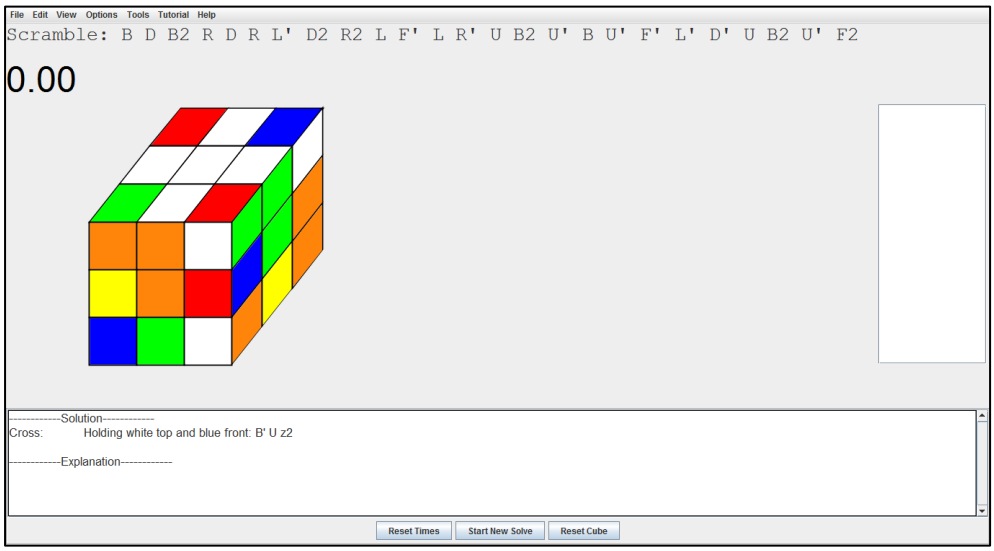


2. If specified in the preferences, a warning window will be shown:

3. Click on the piece for which a solution should be generated. A warning window will be shown, asking you to confirm your choice. For example, if you click on the orange-white edge in this situation:



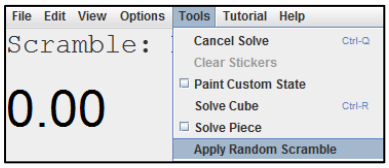
4. If you wish to continue, select *Yes*. The solution will be generated for that piece, and the solution will be shown in the text area at the bottom of the main window.



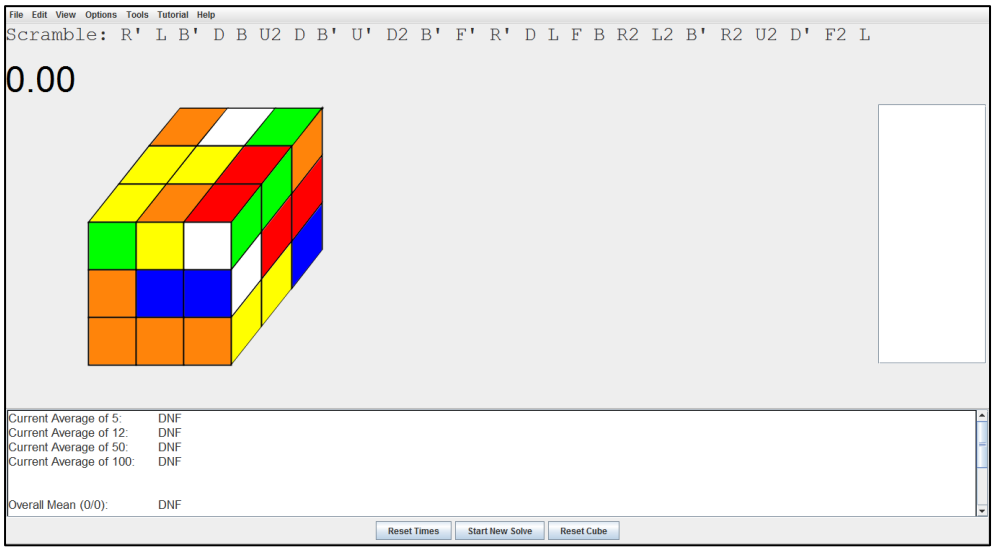
5. If you accidentally select the wrong piece, you can choose not to continue by clicking *No*. You will then have to click the *Solve Piece* menu item again and select the correct piece.

How to apply a random scramble

1. Click the *Apply Random Scramble* menu item (*Main window menu bar → Tools → Apply Random Scramble*)



2. A random scramble will be generated and the cube will be scrambled accordingly.

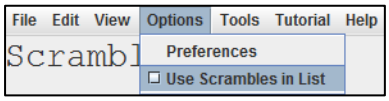


3. You cannot apply a random scramble when any timers are running or when you are painting a custom state.

How to use the scrambles in the scramble list

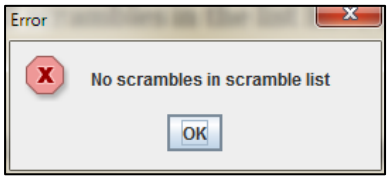
If you have scrambles saved in the scramble list window (see page 22), you can use the scrambles to scramble the cube for each solve. This is useful for competitions; five scrambles could be saved in the scramble list and then these could be used to scramble the cube each time (instead of a random scramble).

1. Click the *Use Scrambles in List* menu item (*Main window menu bar → Options → Use Scrambles in List*).



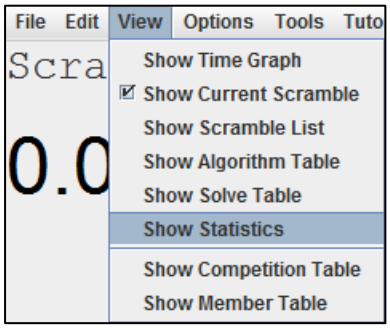
2. If you click *Start New Solve*, the first scramble in the scramble list will be used to scramble the cube. After solving the cube (or clicking *Cancel Solve*), the next scramble in the list will be used to scramble the cube for the next solve. After all scrambles in the list have been used, the first one will be reused, then the second etc.

3. If there are no scrambles in the list, the following error message will be shown:



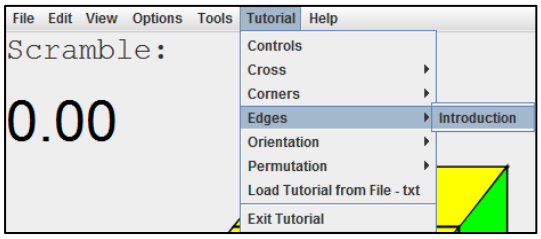
How to show statistics

1. If the text area at the bottom of the screen is showing something other than statistics, then you can show the statistics by selecting the *Show Statistics* menu item (*Main window menu bar → View → Show Statistics*)



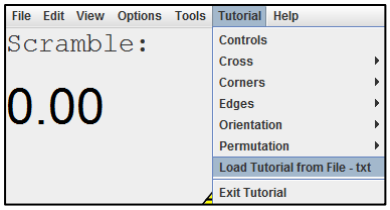
How to open a tutorial

1. Select the desired tutorial from the tutorial menu (*Main window menu bar → Tutorial → Desired tutorial*)

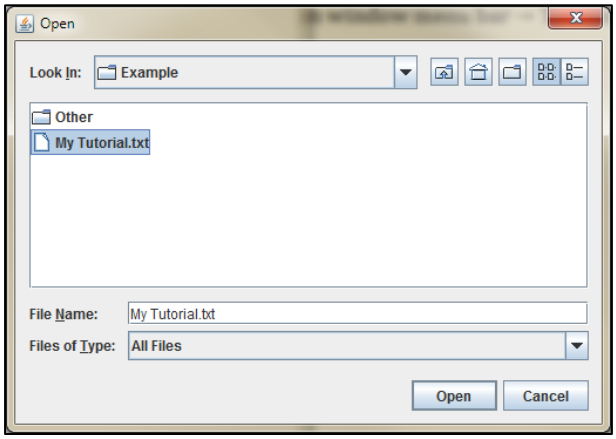


How to load a tutorial from file

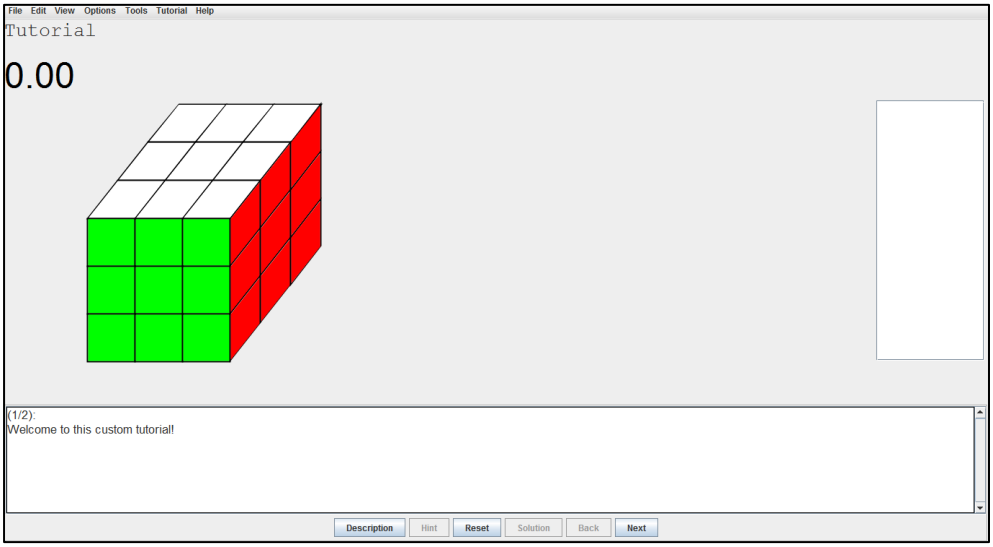
1. Click the *Load Tutorial from File* menu item (*Main window menu bar → Tutorial → Load Tutorial from File*).



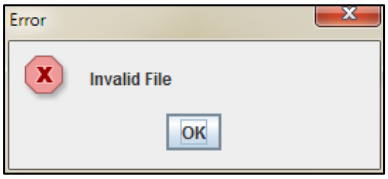
2. Choose the file to load.



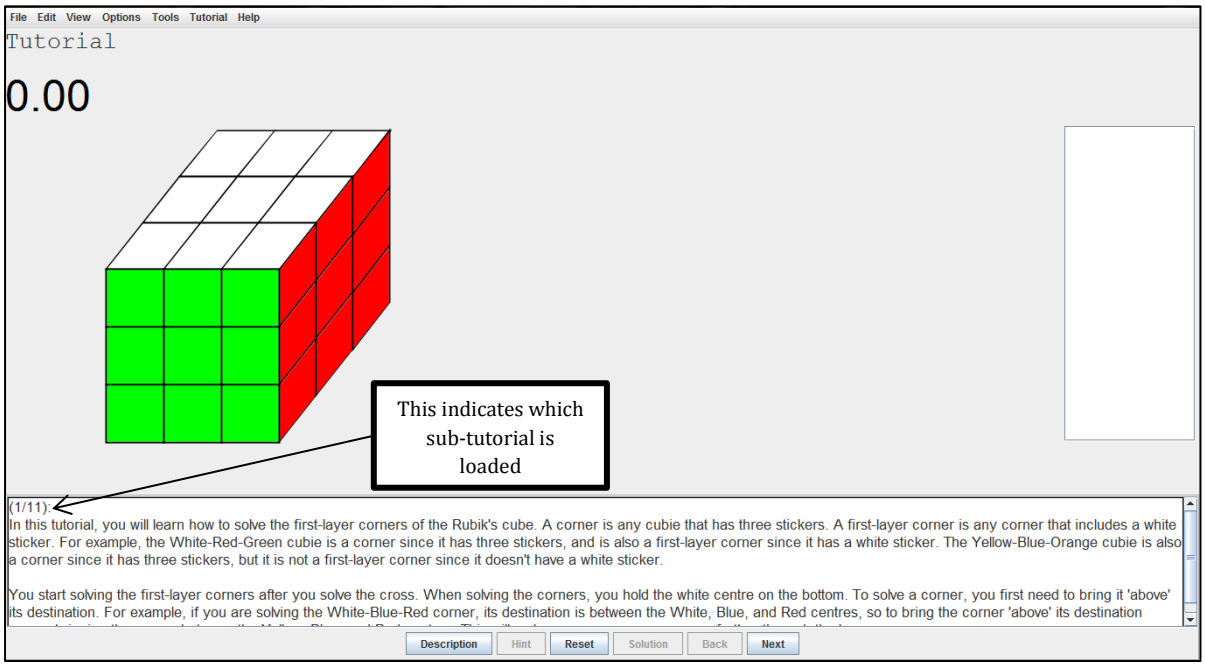
3. The tutorial will be loaded into the main window.



4. If the tutorial file is invalid, then the following error message will be shown:

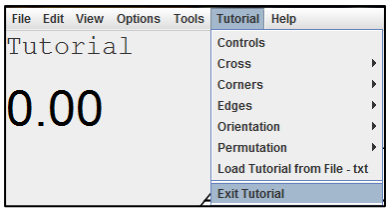


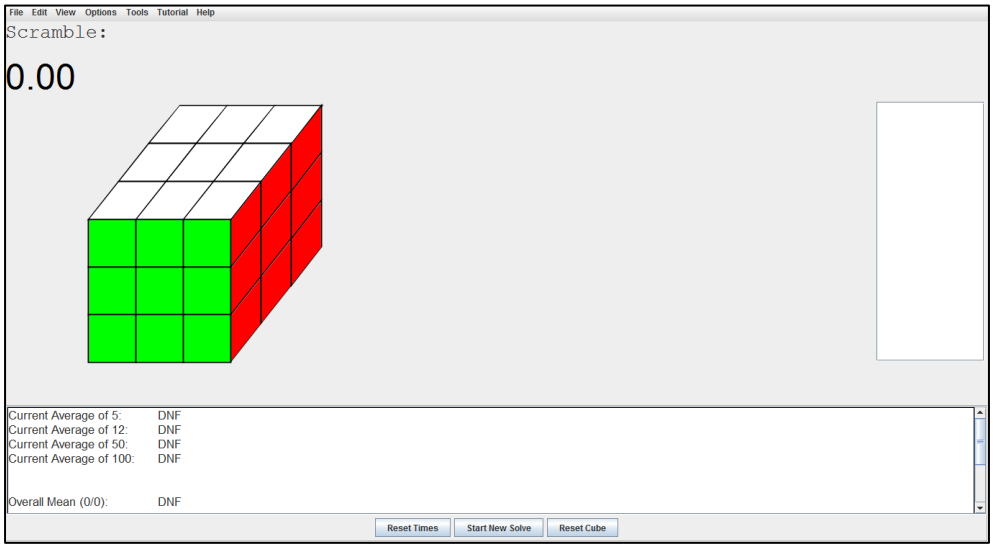
How to use tutorial mode



- *Description* Button: The description for the current step of the tutorial will be shown.
- *Hint* Button: The n^{th} hint for the current step of the tutorial will be shown. The next time this button is clicked, the $(n + 1)^{th}$ hint for the current step will be shown etc. After all hints have been shown, the first hint will be shown again.
- *Reset* button: The cube will be reset to its original state for the current step of the tutorial. This is useful if the tutorial is asking you to solve a certain piece, then you make a mistake; you can simply reset the cube and start again.
- *Solution* button: The solution for the current step of the tutorial will be shown.
- *Back* button: The previous sub-tutorial will be loaded.
- *Next* button: The next sub-tutorial will be loaded.

- To exit tutorial mode, (*Main window menu bar → Tutorial → Exit Tutorial*). The normal 'Timing' mode will be shown.





Data Input Guidelines

Entering times

Times are entered in the *Solve Editor* form (see page 6) and in the *Solve Form* window (see page 31). Times must be of the form MM:SS.ss and must be between 00:00.00 and 59:59.59 inclusive. The time can also be 'DNF' (Did Not Finish) which indicates a disqualified time.

Here are some examples of valid and invalid times:

Valid	Invalid	Explanation
21.57	$^%	All characters in this input are illegal, so it is obviously invalid.
12.19	5.5.	There are two '.' characters in this input, so it is invalid. 5.5 would be valid.
13.22	.	This input consists only of a '.', it has no digits, so it is invalid. '0.' would be valid.
5.55	2:40:50	This input has a colon between '40' and '50' instead of a '.' character; 2:40.50 would be valid.
1:23.91	20	This has no decimal point, so it is invalid. 20. would be valid.
1:4. (= 1:04.00)	-1.2	The presence of the '-' character makes this input invalid. 1.2 would be valid.
59:59.59	60:00.00	This is outside the range of valid times, so it invalid.
DNF	rAnDoM	The only non-'MM:SS.ss' input allowed is 'DNF', so 'rAnDoM' is invalid.

Entering dates

Dates are entered in the *Solve Form* window (see page 31). Dates must be entered in the form YYYY-MM-DD HH:MM:SS, and they must be valid, for example, you could not have 2003-02-29 00:00:00, because 2003 was not a leap year. The time must be between 00:00:00 and 23:59:59 inclusive.

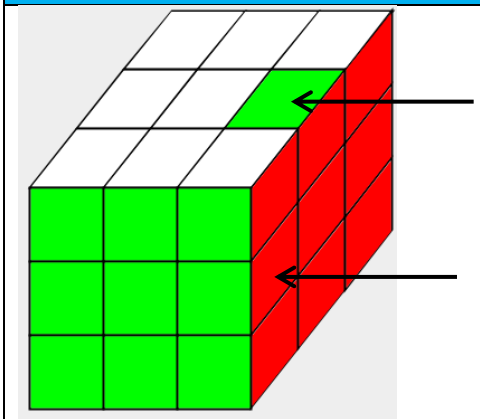
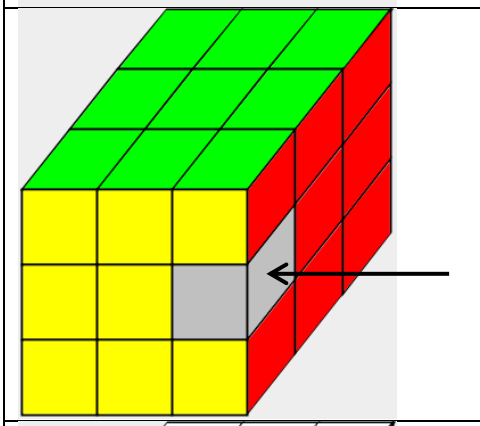
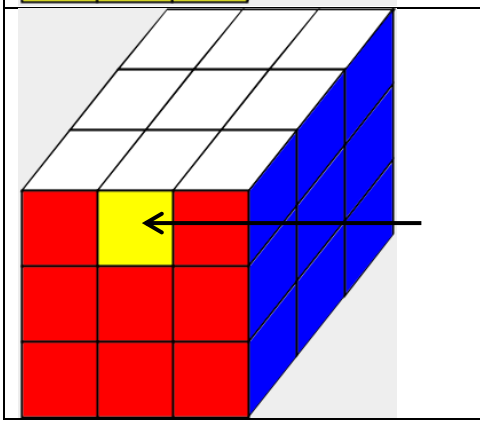
Here are some examples of valid and invalid dates:

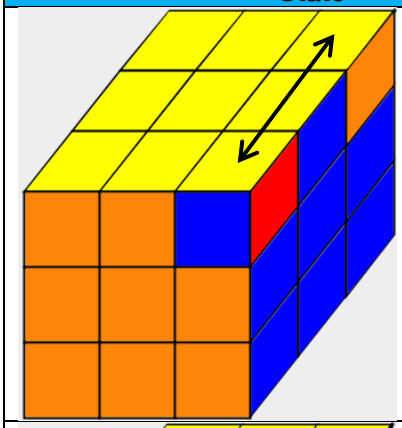
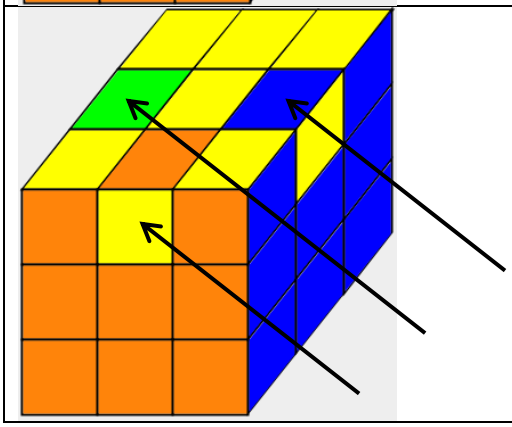
Valid	Invalid	Explanation
2002-01-24 20:33:16	$^&”*	This input is not in the correct format, so it will not be accepted.
1998-12-13 12:04:46	2001-11--15 00:00:00	This has two ‘-‘ characters adjacent to each other, so it is not in the correct format.
2001-05-22 00:00:00	1998-27-5	This input is in the correct format, but there is no 27th month, so this input is invalid. In addition, this input does not have HH:MM:SS included, only YYYY-MM-DD
1998-01-18 23:59:59	2000-01-32 00::00:00	This input is in the correct format, but there is no month with 32 days. There are also two ‘:’ characters adjacent to each other, so this is not valid.
2001-1-1 1:1:1	2014-02-14 30:00:00	This input is in the correct format, but the time ’30:00:00’ is not valid
2004-02-29 00:00:00	2003-02-29 00:00:00	29 February only occurs during leap years; 2003 was not a leap year, so this input will be rejected.

Entering cube states

You can enter cube states by using the 'Paint Custom State' tool. Valid cube states solvable states. States that cannot be solved include those with duplicate pieces, missing pieces, unknown pieces, invalid permutations/orientations, and combinations thereof.

Examples of invalid states:

State	Explanation
	The two green-red edges are duplicates, so this state is invalid.
	The yellow-red edge is missing, so this state is not valid.
	The white-yellow edge is an unknown edge, i.e. it does not exist on a standard Rubik's cube, so this state is invalid.

State	Explanation
	Only the yellow-red-blue and yellow-blue-orange corners have been swapped. Basic theory tells us that this state is unsolvable; if we solve these two corners, we will destroy the permutation of other pieces.
	Only the yellow-green, yellow-orange, and yellow-blue edges have been flipped. Basic theory tells us that this state is unsolvable; there must be an even number of edges flipped for the orientation to be solved.

Entering preferences

- *Real-time solving speed (ms)* field:
  This value can be between 1 and 9999 inclusive. The input must consist of digits only, e.g. 20 not 'twenty'. If the input contains any character other than 0, 1, 2, … 9, then the input will be invalid. If you enter a fractional value, such as 125.6, then it will be rounded down to the nearest integer (125 in this case).

- *Inspection time (seconds)* field:
  This value can be between 1 and 99 inclusive. The input must consist of digits only, e.g. 15 not 'fifteen'. If the input contains any character other than 0, 1, 2, …, 9, or '.' then the input will be invalid. If you enter a fractional value, such as 15.4, then it will be rounded down to the nearest integer (15 in this case).

- *Scramble text size* field:
  This value can be between 1 and 99 inclusive. The input must consist of digits only, e.g. 27 not 'twenty seven'. If the input contains any character other than 0, 1, 2, …, 9, or '.' then the input will be invalid. If you enter a fractional value, such as 27.9, then it will be rounded down to the nearest integer (27 in this case).

Troubleshooting

1. Why is the cube not performing moves when I press keys on the keyboard?

This problem can usually be fixed by clicking somewhere on the cube. After clicking, the cube will have focus and your keyboard input will be acknowledged.

If the cube is still not performing moves, make sure you check the following points:

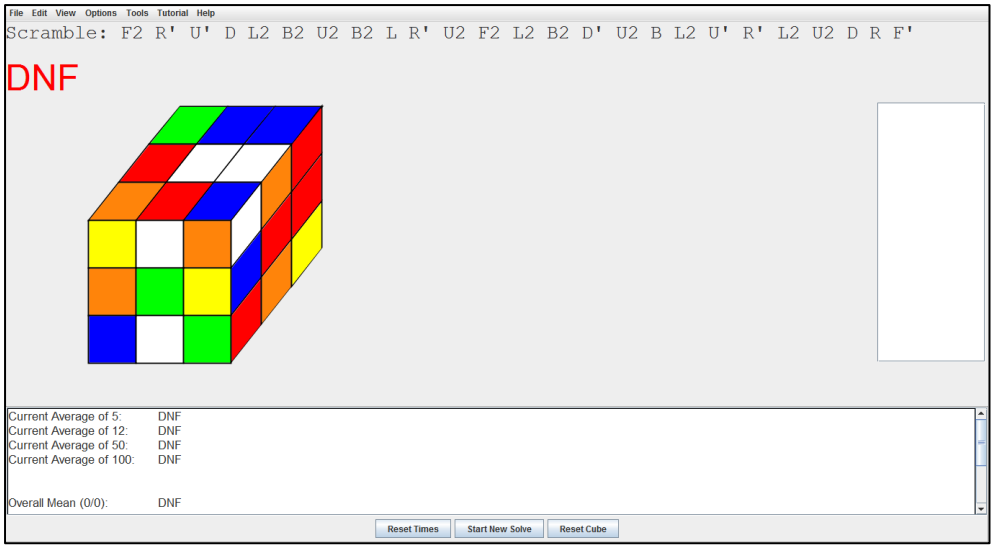
- Make sure the *Color Selection* window is not open – you are not allowed to perform moves when this window is open.
- Permission to perform moves can be restricted if the system is in tutorial mode – this may not be an error, just a feature of the particular tutorial.
- If the cube is being animated automatically, then you will not be allowed to perform moves. Click the *Cancel Solve* menu item (*Main window menu bar → Tools → Cancel Solve*) to cancel the current animation and you will be able to perform moves.
- You are allowed to perform rotations, but not moves, when the inspection timer is running.

Also, make sure that you are pressing valid keys on the keyboard (See page 6).

2. Why are some of my solves being recorded with a penalty of 2?

Solves are given a penalty of 2 if you take more than *x* seconds to inspect the cube, where *x* is the number of seconds allowed for inspection – this is specified in the preferences. For example, if you allow yourself 15 seconds for inspection, then a solve will be given a penalty of +2 if you take more than 15 seconds. You need to make sure that you don't exceed the allowed time so that you don't get these penalties.

In addition, if the timer shows 'DNF' during inspection, then your time will be disqualified. The DNF penalty will be incurred after (*x* + 2) seconds.

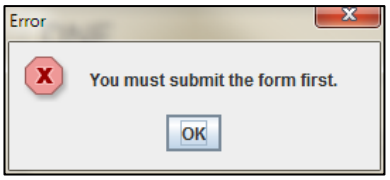


3. When I try to solve an individual piece, the solution solves other pieces as well.

If you choose to solve a piece, such as an edge, without having the previous sub-steps solved, such as cross and corners, the solution will have to complete these steps first. Solving other pieces is not an error or a limitation; these other pieces need to be solved for the solution to be sensible.

4. When I click the View Execution button in the Solve Editor window, an error message appears.

This feature is available only if you are **editing** the solve; if you are adding the solve and you click this button, then the following error message will appear. To avoid this error message, click the 'Submit' button, then you can reopen the window to access this feature.



5. An error message appears when I try to enter a time.

Times must be entered in a specific format (see page 57). If the time you enter is incorrect, then an error message will be shown (see below). Try re-entering the time in a format closer to the accepted formats to avoid this error message.

6. An error message appears when I try to edit or delete a solve in the list in the main window.

This error message appears because there is no selected item in the list. Make sure there is an item selected and then try to edit it.

7. The program is saying that the cube state I enter/load is invalid.

A strict validation process is carried out to determine whether the cube state you enter is valid or not, so ensure that *all* stickers are definitely correct (see page 59). If you are replicating the state of a physical cube, ensure that the physical cube has the same colour scheme and is solvable before continuing.

8. The program is saying that the solve information I'm trying to load is invalid.

Ensure that the file is valid, i.e. has the format {Time, Penalty, Comment, Scramble, Solution} with each of these on a separate line in the text file.

9. I cannot open a file containing tutorial/cube-state/solve information.

Make sure the directory in which the file is stored is accessible to programs, i.e. make sure it is not password-protected or similar, and ensure that the file is valid.

10. The program is saying that the date I enter is invalid.

Ensure that the date is in the form yyyy-MM-dd HH:mm:ss (see page 58), e.g. 2014-03-08 20:33:16, and is a date that actually exists, i.e. you cannot enter the 2014-02-29 00:00:00 because 2014 is not a leap year.

11. The data I enter in the Preferences window is not being accepted.

Ensure that you enter a number in each of the fields (not words, such as "fourteen") and that the value lies within the valid range (see page 61 for ranges).

12. I cannot filter the times in the Solve Table.

Ensure that you enter the times in the pop-up windows correctly (see page 57). In addition, do not enter an upper boundary that is lower than the lower boundary.

13. When I try to start a new solve, an error message appears saying “No scrambles in scramble list”.

This error message appears when you have selected the ‘Use Scrambles in List’ menu item (*Main window menu bar → Options → Use Scrambles in List*) but have not provided data in the Scramble List window (*Main window menu bar → View → Show Scramble List*). You can fix this problem either by entering scrambles into the Scramble List window, or by unchecking the ‘Use Scrambles in List’ menu item.

Limitations of the System

- The system generates solutions using only the beginners' method. Advancements of the system could include more-advanced methods such as CFOP, ZZ, and Roux etc.
- Custom statistics are currently not available, e.g. calculating the average of 15, or average of 20 etc.
- The time graph shows 12 times or fewer – it does not show times over a custom range.
- The key-mappings for performing moves are fixed – they cannot be customised.
- The sticker colours of the cube cannot be customised, e.g. you cannot change the yellow face to show purple stickers.
- The system does not currently support custom notation; only WCA notation is supported.
- Certain tutorial files or solve-information files may cause the system to crash.
- There is currently no English-like explanation for solving the cross.

Administrator User Manual

Contents

Introduction

Kuubik was developed to help the administrator of a school's mathematics club to record information about members in the club, and record information about competitions, including the date of the competition and the times achieved by each member in that competition.

The program allows you to:

- Record details about members, such as name, date of birth, gender, etc.
- Create new competitions and record the dates for these competitions.
- Record the times that the members achieve in each competition.
- Automatically sort members by best average for each competition.

Installation

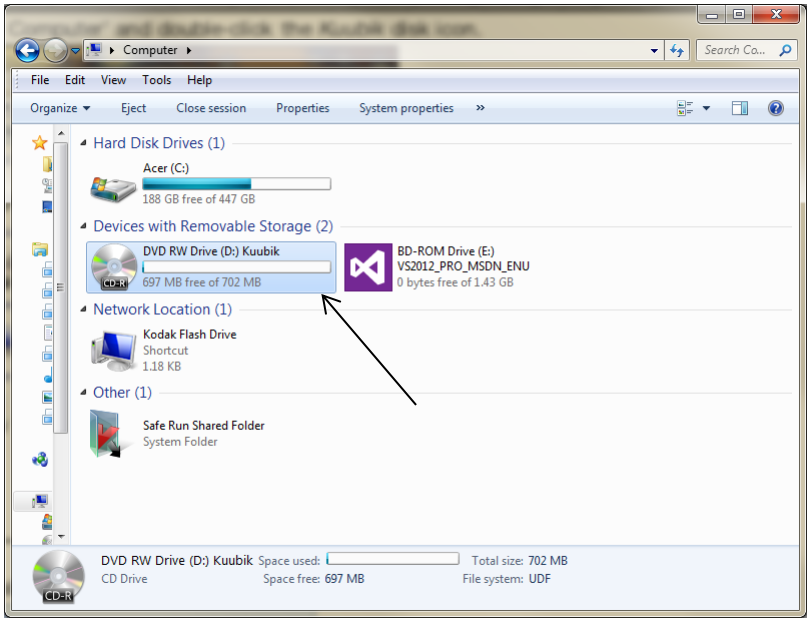
The minimum requirements for the system are:
OS: Windows XP/Vista/7
Processor: 1.0 GHz
Memory: 128 MB RAM
Graphics: N/A
Hard Drive: 8 MB available space

1. Ensure your version of java is up to date by going to *www.java.com/en* and downloading any relevant updates.

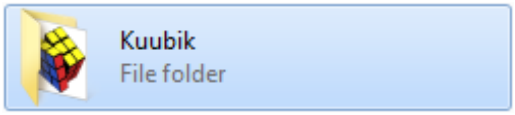
2. If you do not have a PDF software, download Adobe Reader from *get.adobe.com/uk/reader*

3. Insert the installation disc into the computer.

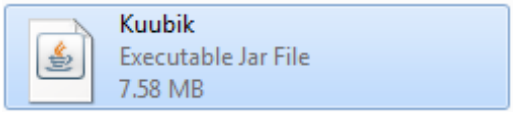
4. Go to 'My Computer' and double-click the *Kuubik* disc icon.



5. A folder called '*Kuubik*' is saved on the disc. Drag the *Kuubik* folder into your My Documents.



6. Once the *Rubik's Cube* folder is stored on your computer, you can open the folder and double-click the *Rubik's Cube* JAR file to open the program.

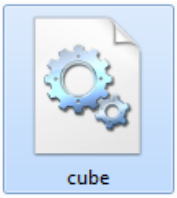


7. To access the program through the desktop, right-click on the *Rubik's Cube* JAR file and select 'Create shortcut', then drag the shortcut onto the desktop.

Backup and Recovery

To back up the data in the database, you can copy the *cube* database file to a different location.

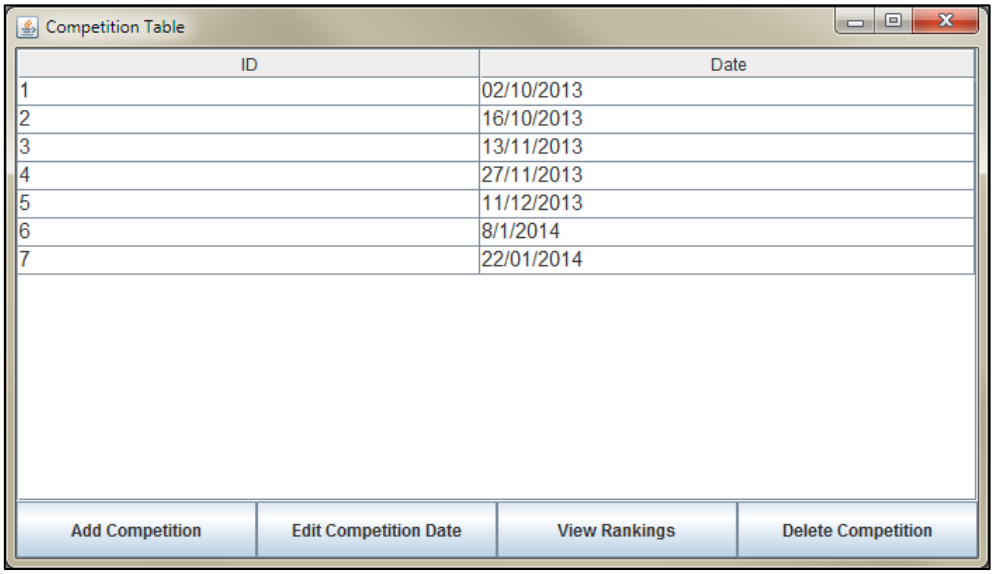
1. Locate the *Kuubik* folder, which you saved during installation.
2. In the *Kuubik* folder, go the *res* folder.
3. Copy the *cube* database file to a different location, such as an external disk or flash drive.



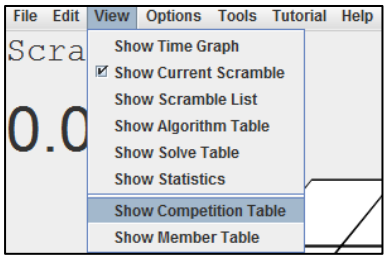
If you need to recover the database, e.g. after re-installing the system, just copy and paste the *cube* database file that is saved on your computer to the *res* file in the *Kuubik* folder.

How to Use the System

How to use the Competition Table window

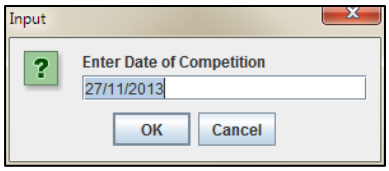


- You can access the competition table by following (*Main window menu bar → View → Show Competition Table*).

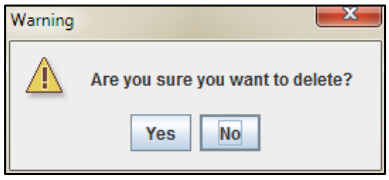


- *Add Competition* button: A new row will be added to the table. The ID will be generated automatically and the date will be blank.

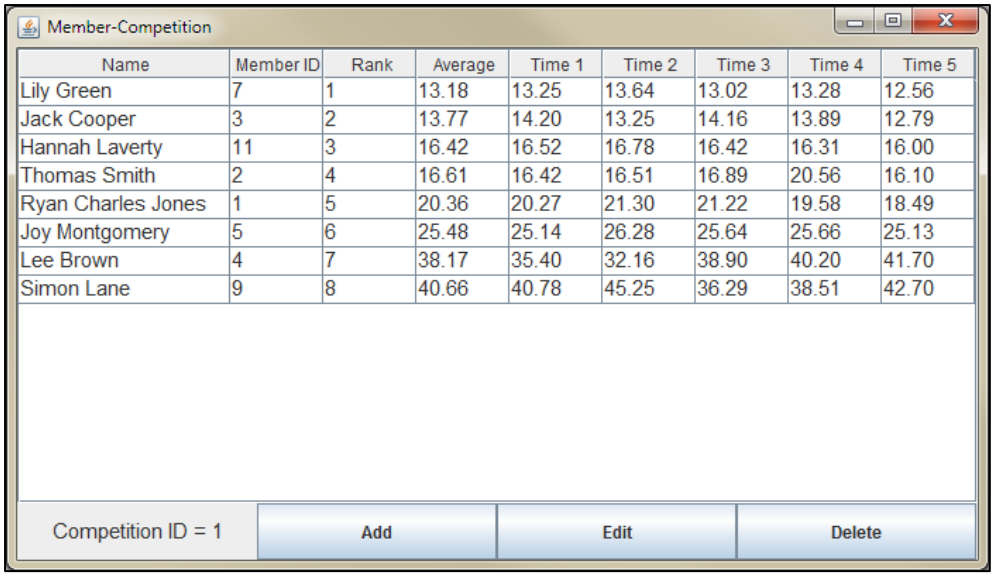
- *Edit Competition Date* button: A window will appear, asking you to enter a new date for the selected competition. Clicking the *OK* button will update the corresponding competition's date. (See the *Data Input Guidelines* section for help on how to enter valid dates).



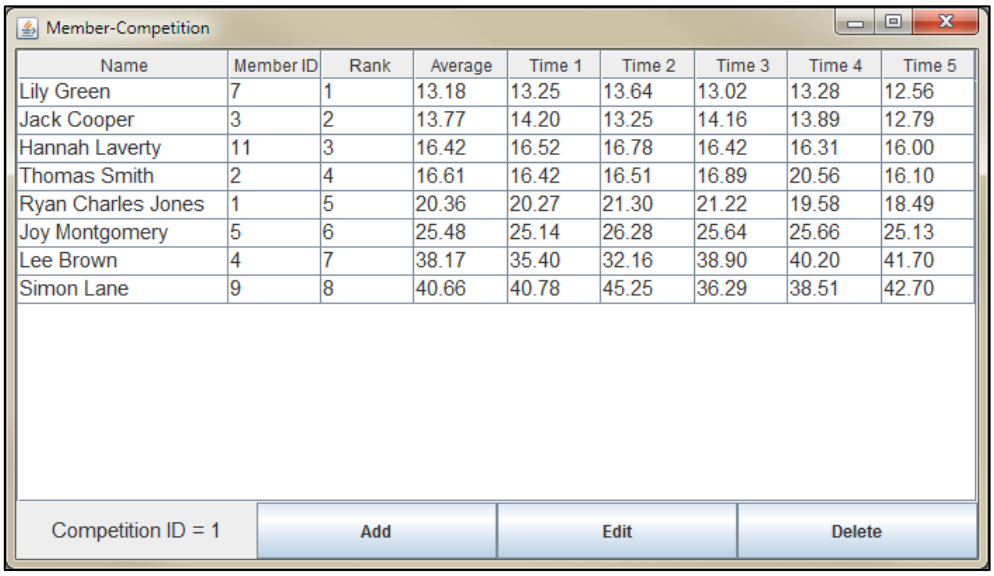
- *Delete Competition* button: The selected competitions will be removed from the database (and from the table). A warning message will appear to confirm your decision. If you select *Yes*, then the selected rows will be removed from the table; if you select *No* then the warning window will close and nothing will be deleted.



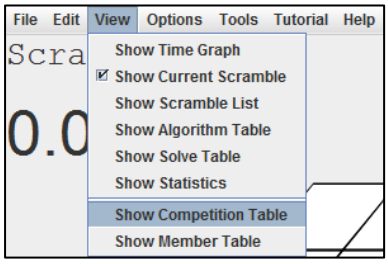
- *View Rankings* button: The *Member-Competition* window will open, showing the rankings for the selected competition.

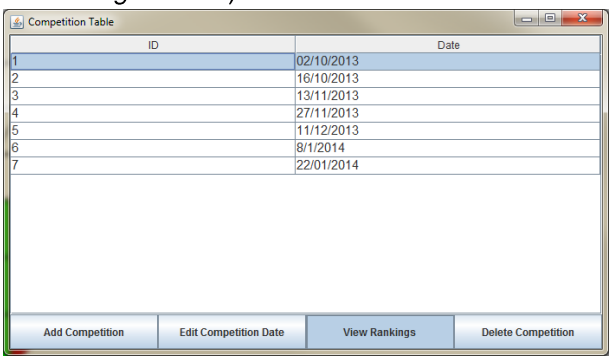


How to use the Member-Competition window

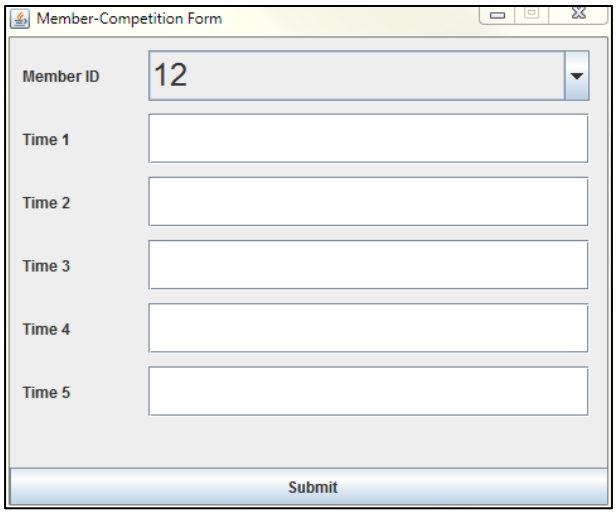


- You can access the Member-Competition window by following (*Main window menu bar* → *View* → *Show Competition Table* → *View Rankings Button*).

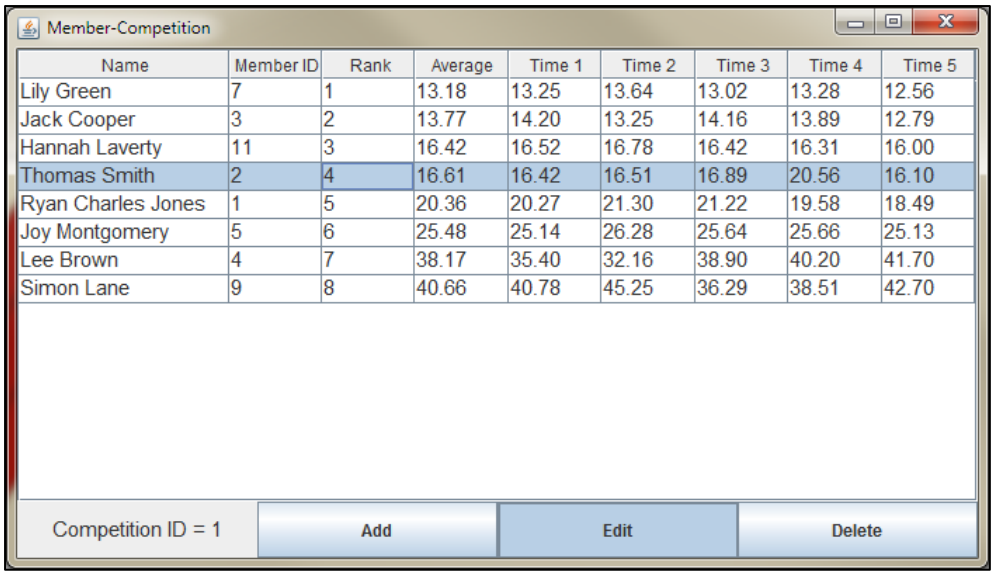


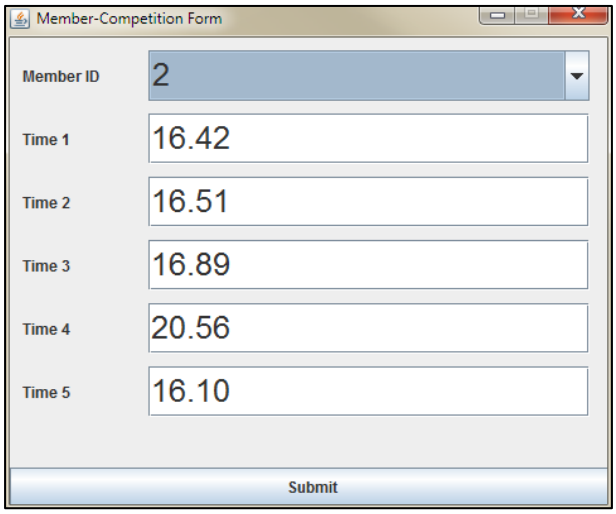


- *Add* button: The *Member-Competition* form window will open, allowing you to enter the information for the new record.

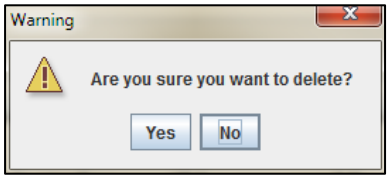


- *Edit* button: The information from the selected row will be placed in the fields of the *Member-Competition Form* window. You can then edit the information as required.

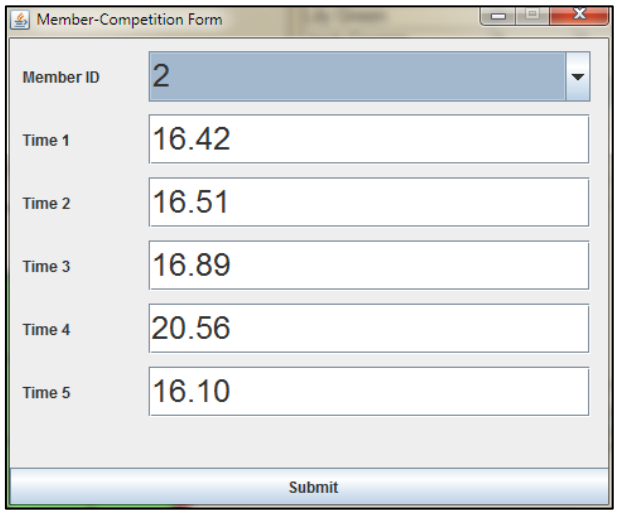




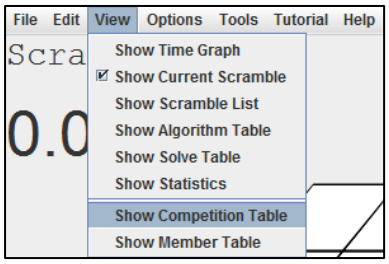
- *Delete* button: The selected row will be removed from the table and the corresponding information will be removed from the database. A warning message will appear before deletion to confirm your decision.

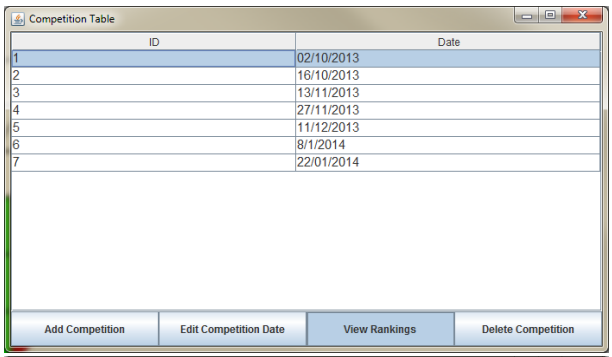


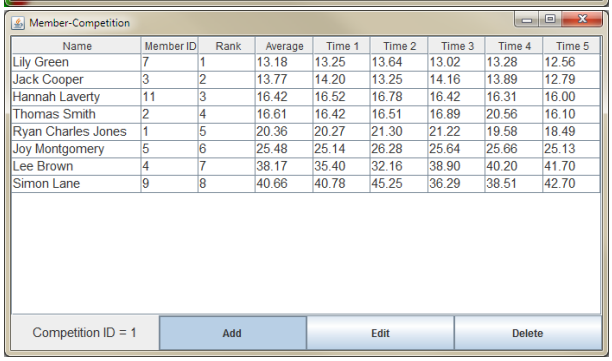
How to use the Member-Competition Form window



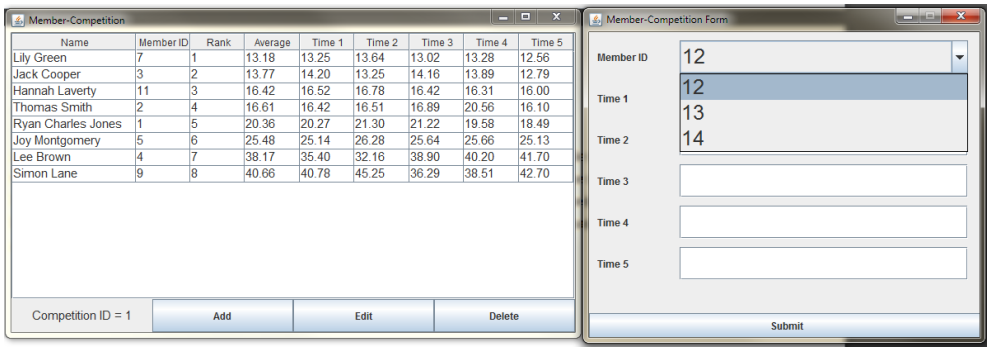
- You can access the Member-Competition window by following (*Main window menu bar → View → Show Competition Table → View Rankings Button → Add/Edit*).







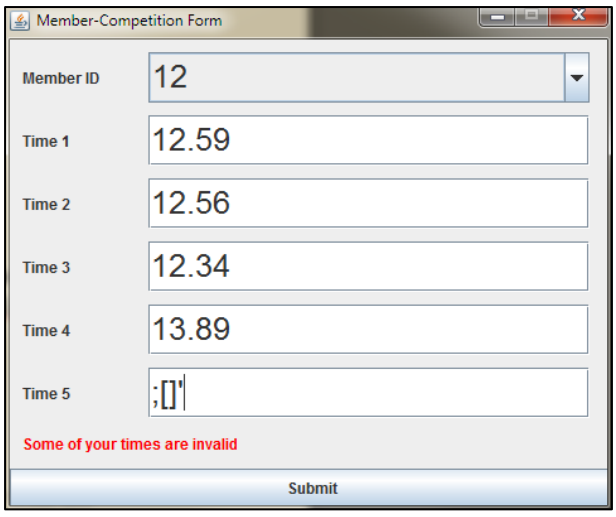
- *Member ID Drop-Down List*: This list shows the available member IDs. Unavailable member IDs are those that have already been used for the current competition. For example, if the member with Member ID = 4 already has data recorded about him/her for this competition, then this drop-down list will not contain the number 4, but if another member with Member ID = 10 has no data recorded about him/her for this competition, then the number 10 *will* be in the list. In the image shown below, members with IDs 1, 2, 3, 4, 5, 7, 9, 11 have data recorded about them for the current competition, but members with IDs 12, 13, 14 do *not* have data recorded about them for the current competition, so numbers 12, 13, 14 are in the drop-down list.



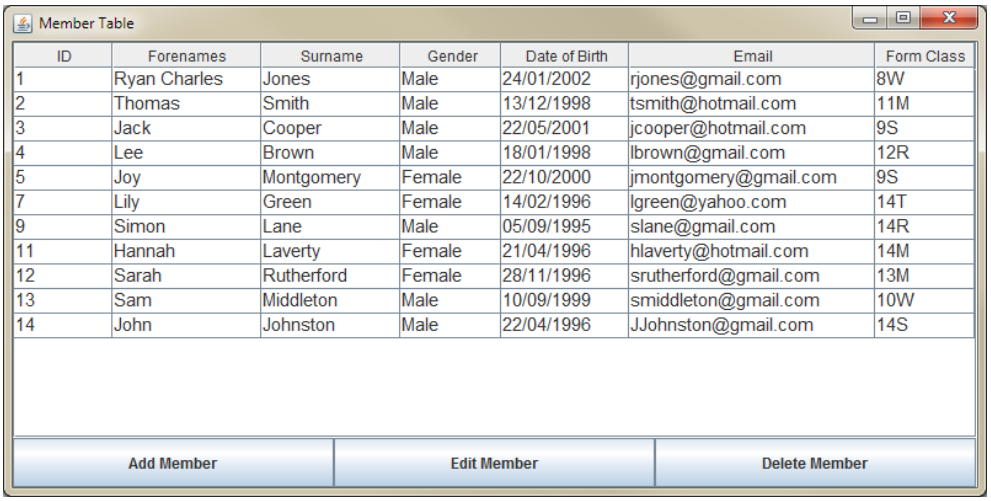
- *Time x* fields: These indicate the five times of the competitor's average. Each time must be entered in the format specified in the *Data Input Guidelines* section, but the general format is MM:SS.ss.
  Examples:
  1:12.47 – indicates one minute 12.47 seconds
  00:15.98 – indicates 15.98 seconds
  15.98 – indicates 15.98 seconds

- *Submit* button: If you are adding a new record, then a new row will be added to the table with the information you provided; if you are editing a record, then the corresponding row will be updated. The rankings will be updated accordingly.

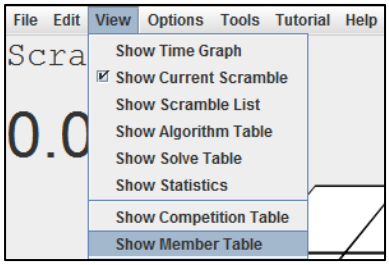
- If any of the times are invalid, then after clicking the *Submit* button, an error message will appear at the bottom of the window.



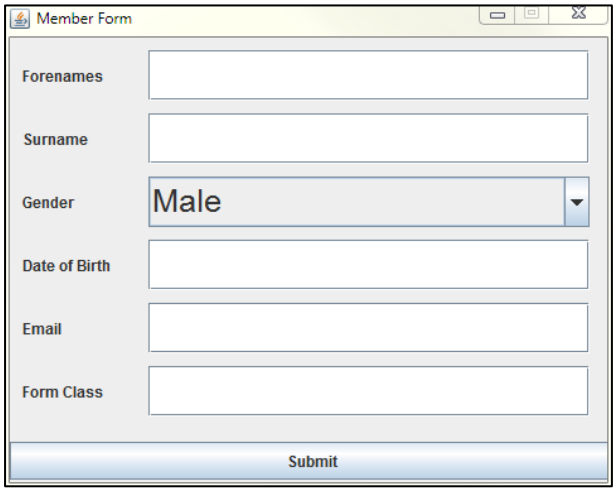
How to use the Member Table window



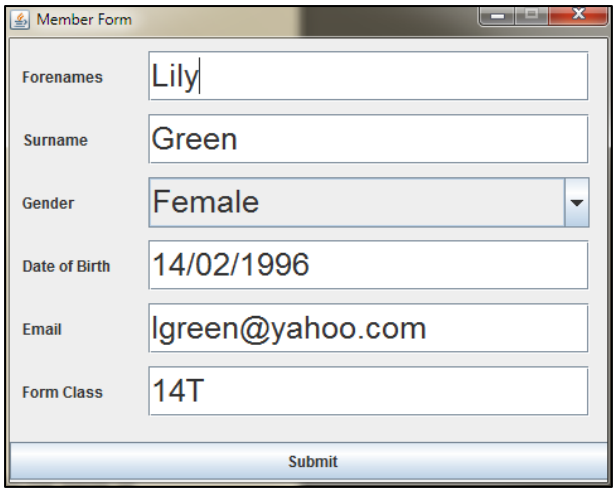
- You can access the Member Table window by following (*Main window menu bar → View → Member Table*).



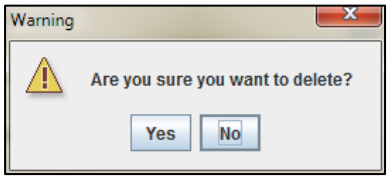
- *Add Member* button: The *Member Form* window will open, allowing you to enter the information about the new member.



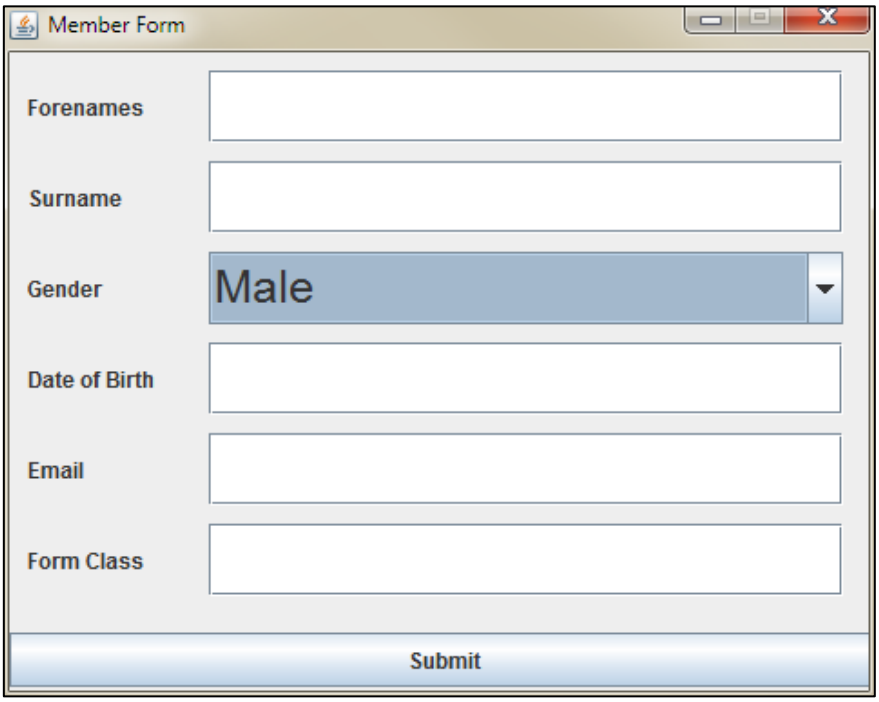
- *Edit Member* button: The information from the selected row will be placed in the fields of the *Member Form* window. You can then edit the information as required.



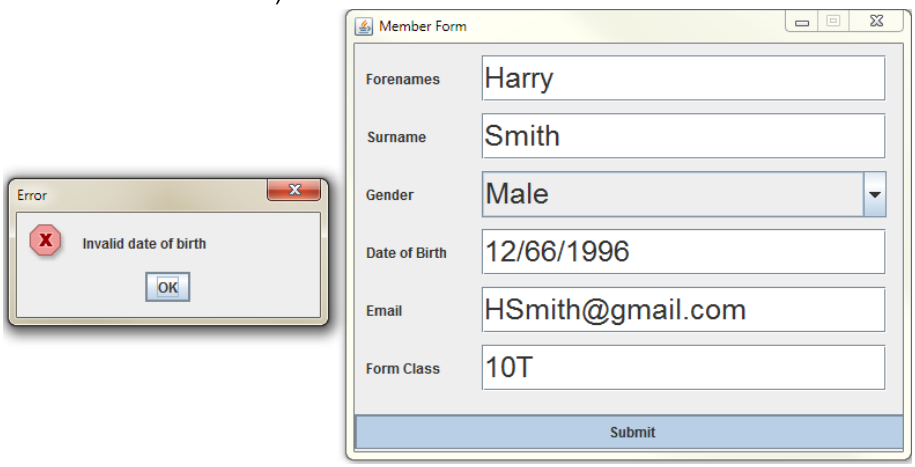
- *Delete Member* button: The selected member will be removed from the database and from the table. A warning window will be shown before deletion in order to confirm your decision.



How to use the Member Form window



- You can access the Member Form window by following (*Main window menu bar → View → Member Table → Add Member/Edit Member*)
- No field may be left blank.
- *Date of Birth* field: Date of birth must be in the format “DD/MM/YYYY”, e.g. 12/11/1996
- *Email* field: The email must be in a valid format, e.g. me@example.com
- *Form Class* field: The form class must be one of 8M, 8R, 8S, 8T, 8W, 9M…, 14M, 14R, 14S, 14T, 14W
- *Submit* button: If any of the data in the form is incorrect, then an error message will be shown, informing you that certain data is incorrect. (See the *Data Input Guidelines* section for help on how to enter valid data).



Data Input Guidelines

Entering dates

Dates are entered in the *Competition Table* window (see page 5) and the *Member Form* window (see page 13). Dates must be entered in the form DD/MM/YYYY, they must be before the current date, and they must be valid, for example, you could not have 29/02/2003, because 2003 was not a leap year.

Here are some examples of valid and invalid dates:

Valid	Invalid	Explanation
24/01/2002	$^&”*	This input is not in the correct format, so it will not be accepted.
13/12/1998	15//11/2001	This has two backslash characters adjacent to each other, so it is not in the correct format.
22/05/2001	5/27/1998	This input is in the correct format, but there is no 27th month, so this input is invalid.
18/01/1998	32/01/2000	This input is in the correct format, but there is no month with 32 days.
1/1/2001	01/01/2100	This input is in the correct format, but is after the current date, so it will be rejected.
29/02/2004	29/02/2003	29 February only occurs during leap years; 2003 was not a leap year, so this input will be rejected.

Entering email addresses

Emails are entered in the *Member Form* window (see page 13). Emails must be valid, but do not necessarily need to exist. For example, the email address 8393481@DoesNotExist.com might not exist, but is still a valid.

Here are some examples of valid and invalid emails:

Valid	Invalid
jmontgomery@gmail.com	address@
hlaverty@hotmail.com	.com

There are too many possible inputs to show here, but the general form is x@y.z

Entering times

Times are entered in the *Member-Competition Form* window (see page 9). Times must be of the form MM:SS.ss and must be between 00:00.00 and 59:59.59 inclusive.

Here are some examples of valid and invalid times:

Valid	Invalid	Explanation
21.57	$^%	All characters in this input are illegal, so it is obviously invalid.
12.19	5.5.	There are two '.' characters in this input, so it is invalid. 5.5 would be valid.
13.22	.	This input consists only of a '.', it has no digit, so it is invalid. 0. would be valid.
5.55	2:40:50	This input has a colon between '40' and '50' instead of a '.' character; 2:40.50 would be valid.
1:23.91	20	This has no decimal point, so it is invalid. 20. would be valid.
1:4. (= 1:04.00)	-1.2	The presence of the '-' character makes this input invalid. 1.2 would be valid.
59:59.59	60:00.00	This is outside the range of valid times, so it invalid.
DNF	rAnDoM	The only non-'MM:SS.ss' input allowed is 'DNF', so 'rAnDoM' is invalid.

Troubleshooting

1. An error message appears when I try to enter a time.

Times must be entered in a specific format (see page 15). If the time you enter is incorrect, then an error message will be shown. Try re-entering the time in a format closer to the accepted formats to avoid this error message.

2. An error message appears when I try to enter a form class.

Make sure you enter one of the following as the form class: 8M, 8R, 8S, 8T, 8W, 9M, 9R, 9S, 9T, 9W, 10M, 10R, 10S, 10T, 10W, 11M, 11R, 11S, 11T, 11W, 12M, 12R, 12S, 12T, 12W, 13M, 13R, 13S, 13T, 13W, 14M, 14R, 14S, 14T, 14W.

3. An error message appears when I try to enter an email address.

The email address you enter does not have to exist, but it must have the format x@y.z, e.g. me@example.com (see page 14).

4. An error message appears when I try to enter a date of birth.

Ensure the date you enter is in the form DD/MM/YYYY, is valid, and is before the current date (see page 14).

5. Some of the records in the Member-Competition window show the name UNKNOWN.

Records that show the name ‘UNKNOWN’ represent data stored about a member that no longer exists in the database. The data has not been deleted from the Member-Competition table because usually you will want to keep this data. You cannot edit these records, you can only delete them.

Limitations of the System

- Emails can be validated, but not verified; this means you can enter an email address that does not actually exist. If the system were online, then features like this could be added.

- Since competitors perform solving on different laptops during a competition, there is no way to add their times to the database automatically. If a network was in place, this would be possible, but currently the system does not support this.

- Different events are not available, such as FMC and BLD, since some events require a mean of 3 rather than an average of 5, while others require a numerical answer (FMC).