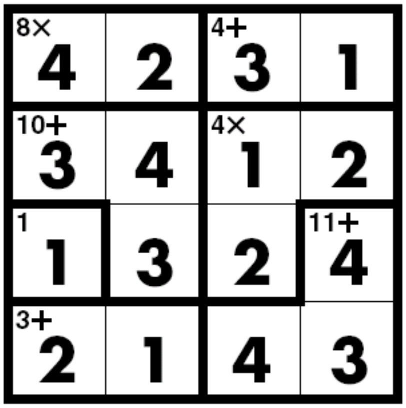
CS 3102 Term Project: KenKen Generator and Solver

Art Chaidarun (nc5rk) and Scott Tepsuporn (spt9np)

May 1, 2013

Introduction

KenKen is a popular puzzle game created in 2004 by Tetsuya Miyamoto, a Japanese math teacher. The puzzle shares many gameplay elements with Sudoku: the player must assign a number to each cell in a $n \times n$ grid such that each cell in a row or a column contains a unique number from 1 to n. In KenKen, however, cells of a "cage" must also satisfy the cage clue.



8× 4	2	4+ 3	1
10+ 3	4	4× 1	2
1 1	3	2	11+ 4
3+ 2	1	4	3

Figure 1: Example of a solved 4×4 puzzle with seven cages.

Controls

- Esc – Exit
- F1 – Show help information listed here
- F2 – Clear all guesses and notes from the current puzzle
- F3 – Create a new 3 × 3 puzzle
- F4 – Create a new 4 × 4 puzzle
- F5 – Create a new 5 × 5 puzzle

- F6 – Create a new 6 × 6 puzzle
- F7 – Create a new 7 × 7 puzzle
- F8 – Create a new 8 × 8 puzzle
- F9 – Create a new 9 × 9 puzzle
- F10 – Solve the puzzle using brute force
- F11 – Solve the puzzle using depth-first search
- F12 – Enable/disable generation of puzzles with modulo cages
- Backspace – Undo the last number entry action

To mark a cell with a number, hover the mouse cursor over the desired cell and type the number. To clear the number from the cell, type the same number again.

Hitting any key other than the ones listed above toggles between guess entry mode and note entry mode. The hovered cell is highlighted in gray during guess entry mode and in blue during note entry mode.

UI Features

Check As You Type

Guesses that violate any of the three uniqueness constraints (row, column, and cage) are highlighted in red upon entry.

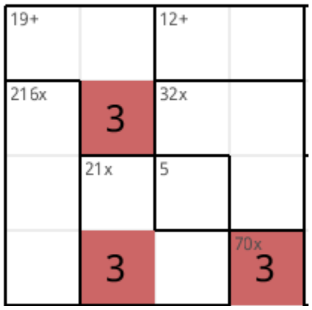


Figure 2: Row and column constraints not satisfied.

Cell Notes

To enter note entry mode, press any key that does not have a function assigned to it (see Controls section). The cursor will turn from gray to light blue to indicate the change in input mode. The user can then hover over any cell and type numbers to enter notes for that cell.

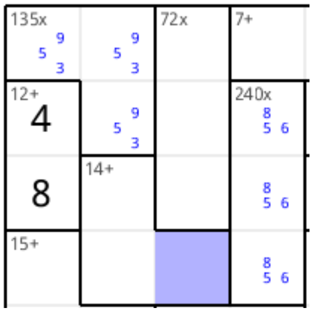


Figure 3: Note mode.

Undo History

Press backspace to undo the last action. The history is cleared whenever the puzzle is reset and whenever a new puzzle is generated.

Search Display

As the computer solves the puzzle using the specified search method, the window is updated with guess values from the current attempt. The refresh rate is set to once every k attempts, where k is small enough that the cell values update frequently yet large enough to have negligible impact on the search algorithm's running time.

The interval k is set to 65536 for brute force and 4096 for depth-first search, suggesting that brute force generates and checks each solution attempt approximately 16 times faster than depth-first search does. Of course, the efficiency of depth-first search more than compensates for the longer time spent on each individual attempt.

Puzzle Generator

To efficiently generate a new $n \times n$ puzzle, we start with an arbitrary legal solution (one that has each cell satisfying row and column constraints, i.e. a Latin square) and then shuffle the rows and columns. This ensures that the resulting solution is random yet still legal. The initial board we choose is the group table for addition modulo n with all entries shifted up by 1, due to the simplicity of the generating formula:

$$\text{cell}_{i,j} = ((i+j) \mod n) + 1$$

The next step is to construct cages. The algorithm to do this is as follows:

1. Start with a two-dimensional array that has each cell set to an 'uncaged' flag value (namely -1). Each cell in the array corresponds to a cell in the problem.

1	2	3	4
2	3	4	1
3	4	1	2
4	1	2	3

Figure 4: The initial solution template for 4 × 4 puzzles.

2. Randomly select the size of the cage (1-4 cells) to build. The probability of selecting a certain size can be changed in the program.

3. Select the first available uncaged cell in row-major order as the root node of a new cage. Mark the cell with an integer that uniquely identifies this cage.

4. Continue adding adjacent cells in random directions until we have either reached the pre-determined cage size or have run into a dead end where all adjacent cells have already been caged.

5. Randomly assign an operation to the cage. The probability of each operation can also be changed. We ensure that all cages with non-commutative operations have only two cells and that the contents of all division cages divide without remainder.

   Assignment of the modulo operation to cages is not a feature found in standard KenKen puzzles, and it may be enabled or disabled for new puzzles by pressing F12.

6. Repeat from step 2 until all cells have been caged.

Puzzle Solver

To have the computer automatically solve the current puzzle, the user may select between brute force and depth-first search. Both methods are completely fair in the sense that they never access the solution of the generated puzzle, even though it is contained in the same program.

Brute Force

The brute force method for iterating through candidate solutions is similar to the technique used for generating legal boards; the only difference is that we apply permutations in lexicographically increasing order to the rows and columns instead of shuffling them. This ensures that wrong attempts are never revisited.

Assuming that board solutions are uniformly distributed across all permutations in both dimensions, the solution is expected to be found after $\frac{(n!)^2}{2}$ attempts. For the 9×9 puzzle, this figure is nearly 66 billion. On a modern 2.3 GHz computer that can check 60 million boards per minute, the brute force solution will take an average time of 18.3 hours to complete.

Depth-First Search

The basic depth-first search implementation starts with the first available unknown cell. It then iterates through the set of possible values for that cell, hypothesizing a different value for the cell and spawning a new depth-first search at every iteration. Searches fail when the board contains at least one cell with no possible values. Eventually the search will terminate when each cell has exactly one possible value.

To make our solver more efficient, we first restrict the sets of possible values for all cells in unit cages to only the values specified by their clues. This causes depth-first search to skip over these cells. For multiplication cages, we remove numbers that are not factors of the specified product from their cells' lists of possible values. Similar preprocessing reductions are applied to the other operators.

We then take another pass through all the board cells, recursively removing the values of all known cells (those with only one remaining possible value) from the possible-value sets of their peer cells in the same row or column. The state space has been vastly reduced at this point.

Finally, we recursively call depth-first search. A heuristic is applied that prefers cells with fewer remaining possible values and, to a lesser extent, those in multiplication, division, and modulo cages. These cages tend to have fewer possible values than addition cages and subtraction cages.

Comparison

While the optimized depth-first search solver performs no faster than the brute force solver in the worst case, it is much faster than the brute force solver on average.

Both algorithms solve boards of size 6 or smaller almost instantaneously. Brute force on an 8×8 puzzle usually takes around 15 minutes, while depth-first usually takes only a few seconds. Brute force on a 9×9 puzzle would probably take around 18 hours; depth-first search typically solves it in less than ten seconds.

Although the running times for brute force were uniformly distributed, those of depth-first search showed an extremely positive skew. Some invocations on 9×9 puzzles finished in a millisecond, about half finished within five seconds, and still others took 30 minutes.

Complete Source Code

```java
package edu.virginia.kenken;

/**
 * @author artnc
 * @author scteps
 *
 */
public class Driver {

  public static void main(String[] args) {
    GUI gui = new GUI(6);
    gui.gameLoop();
    gui.destroy();
  }

}
```

src/Driver.java

```java
package edu.virginia.kenken;

import static org.lwjgl.opengl.GL11.*;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Stack;
import java.util.TreeMap;

import org.lwjgl.LWJGLException;
import org.lwjgl.input.Keyboard;
import org.lwjgl.input.Mouse;
import org.lwjgl.opengl.Display;
import org.lwjgl.opengl.DisplayMode;
import org.lwjgl.opengl.GL11;
import org.newdawn.slick.Color;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.UnicodeFont;
import org.newdawn.slick.font.effects.ColorEffect;

/**
 * @author art
 *
 */
public class GUI {

  // Board constants
  private static final int WINDOW_WIDTH = 480;
  private static final int WINDOW_HEIGHT = 480;
  private static final int BOARD_WIDTH = WINDOW_HEIGHT - 30;
  private static final float LINE_WIDTH = 2.0f;
  private static final int BOARD_OFFSET_X = 15;
  private static final int BOARD_OFFSET_Y = 15;

  // Clue constants
  private static final int CLUE_OFFSET_X = 3;
  private static final int CLUE_OFFSET_Y = 1;
  private static final int CLUE_FONT_SIZE = 12;

  // Guess variables
  private int guess_offset_x;
  private int guess_offset_y;
  private static final int GUESS_FONT_SIZE = 25;

  // Note constants
  private static final int NOTE_OFFSET_X = 10;
  private static final int NOTE_OFFSET_Y = 15;
  private static final int NOTE_FONT_SIZE = 10;

  // Help text constants
  private static final int HELP_OFFSET_X = 19;
  private static final int HELP_OFFSET_Y = 11;
  private static final int HELP_FONT_SIZE = 20;
  private static final String HELP_TEXT = "ESC:\n" + "F1:\n" + "F2:\n"
    + "F3:\n" + "F4:\n" + "F5:\n" + "F6:\n" + "F7:\n" + "F8:\n" + "F9:\n"
    + "F10:\n" + "F11:\n" + "F12:\n" + "OTHER:";
  private static final String HELP_DESC = "EXIT\n" + "HELP\n" + "RESET\n"
    + "NEW 3x3 PUZZLE\n" + "NEW 4x4 PUZZLE\n" + "NEW 5x5 PUZZLE\n"
    + "NEW 6x6 PUZZLE\n" + "NEW 7x7 PUZZLE\n" + "NEW 8x8 PUZZLE\n"
    + "NEW 9x9 PUZZLE\n" + "SOLVE (BRUTE FORCE)\n" + "SOLVE (DFS)\n"
    + "ENABLE/DISABLE % CAGES\n" + "TOGGLE GUESS/NOTE MODE";

  private static final String FONT_PATH = "res/DroidSans.ttf";

  // Current problem
  private Problem problem;

  // Height (or width) of problem in cells
  private int size;

  // Grid of cage IDs
  HashMap<Integer, Integer> cageIDs;

  // Cell and cages relationship
  private ArrayList<Cage> cellCages;

  // Pixel width of a cell
  private int cellWidth;
```

```java

    // Number fonts
    private UnicodeFont clueFont;
    private UnicodeFont guessFont;
    private UnicodeFont noteFont;

    // Help font
    private UnicodeFont helpFont;

    // Matrix of user's cell guesses
    private HashMap<Integer, Integer> guessGrid;

    // Matrix of user's cell notes
    private HashMap<Integer, ArrayList<Boolean>> noteGrid;

    // Matrix of incorrect cells
    private HashMap<Integer, Boolean> incorrectGrid;

    // Matrix of incorrect cell (cage)
    private ArrayList<ArrayList<Boolean>> incorrectCellCages;

    // Maps clue cells to clue text
    private TreeMap<Integer, String> clueText;

    // Guess/note history
    private Stack<Integer> numHistory;
    private Stack<Boolean> toggleHistory;
    private Stack<Integer> hoverXHistory;
    private Stack<Integer> hoverYHistory;

    // Grid indices of the currently hovered cell
    private int hoverCellX;
    private int hoverCellY;

    // Whether entry mode is "guess" or "note"
    private boolean inGuessMode;

    // Whether or not to show help on the board
    private boolean showHelp;

    // Whether or not problems with modulo cages can be generated
    private boolean modEnabled;

    // Whether main loop should be running
    private boolean running;

    // Used for checking whether player-filled board is solution
    private HashMap<Integer, HashSet<Integer>> attempt;

    // Used for displaying time player took to solve puzzle
    private long startTime;

    // Whether current guess/note entry is actually an undo action
    private boolean isUndo;
```

```

  public GUI(int startupSize) {
    running = true;
    modEnabled = false;
    init();
    setNewProblem(startupSize);
  }

  /**
   * Initialize LWJGL and create the window.
   */
  @SuppressWarnings("unchecked")
  private void init() {
    // Create window
    try {
      Display.setDisplayMode(new DisplayMode(WINDOW_WIDTH, WINDOW_HEIGHT));
      Display.setTitle("KenKen");
      Display.create();
    } catch (LWJGLException e) {
      System.err.println("Display wasn't initialized correctly.");
      System.exit(1);
    }

    // Create keyboard
    try {
      Keyboard.create();
    } catch (LWJGLException e) {
      System.out.println("Keyboard could not be created.");
      System.exit(1);
    }

    glEnable(GL_TEXTURE_2D);
    glShadeModel(GL_SMOOTH);
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_LIGHTING);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, WINDOW_WIDTH, WINDOW_HEIGHT, 0, 1, -1);
    glMatrixMode(GL_MODELVIEW);
    glEnable(GL_COLOR_MATERIAL);

    // Set background color to white
    glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // Line thickness
    glLineWidth(LINE_WIDTH);

    try {
      // Temporarily disable System.out
      // System.setOut(new PrintStream(new OutputStream() {
```

```java
      // @Override
      // public void write(int b) {
      // // Do nothing
      // }
      // }));

      clueFont = new UnicodeFont(FONT_PATH, CLUE_FONT_SIZE, false, false);
      clueFont.addAsciiGlyphs();
      clueFont.addGlyphs(400, 600);
      clueFont.getEffects().add(new ColorEffect());
      clueFont.loadGlyphs();

      guessFont = new UnicodeFont(FONT_PATH, GUESS_FONT_SIZE, false, false);
      guessFont.addAsciiGlyphs();
      guessFont.addGlyphs(400, 600);
      guessFont.getEffects().add(new ColorEffect());
      guessFont.loadGlyphs();

      noteFont = new UnicodeFont(FONT_PATH, NOTE_FONT_SIZE, false, false);
      noteFont.addAsciiGlyphs();
      noteFont.addGlyphs(400, 600);
      noteFont.getEffects().add(new ColorEffect());
      noteFont.loadGlyphs();

      helpFont = new UnicodeFont(FONT_PATH, HELP_FONT_SIZE, false, false);
      helpFont.addAsciiGlyphs();
      helpFont.addGlyphs(400, 600);
      helpFont.getEffects().add(new ColorEffect());
      helpFont.loadGlyphs();

      // Re-enable System.out
      // System.setOut(System.out);

    } catch (SlickException e) {
      System.out.println("Failed to create font. Exiting.");
      e.printStackTrace();
      System.exit(1);
    }
  }

  private void reset() {
    guessGrid = new HashMap<Integer, Integer>();
    noteGrid = new HashMap<Integer, ArrayList<Boolean>>();
    incorrectGrid = new HashMap<Integer, Boolean>();
    incorrectCellCages = new ArrayList<ArrayList<Boolean>>();
    attempt = new HashMap<Integer, HashSet<Integer>>();
    for (int i = 0; i < size; ++i) {
      incorrectCellCages.add(new ArrayList<Boolean>());
      for (int j = 0; j < size; ++j) {
        guessGrid.put(i * size + j, -1);
        noteGrid.put(i * size + j,
          new ArrayList<Boolean>(Collections.nCopies(size, false)));
        incorrectGrid.put(i * size + j, false);
        incorrectCellCages.get(i).add(false);
      }
    }

    inGuessMode = true;
    numHistory = new Stack<Integer>();
    toggleHistory = new Stack<Boolean>();
    hoverXHistory = new Stack<Integer>();
    hoverYHistory = new Stack<Integer>();

    Display.setTitle("KenKen");
    startTime = System.nanoTime();
  }

  /*
   * Load a new problem instance into the main window.
   */
  private void setNewProblem(int size) {
    this.size = size;
    cellWidth = BOARD_WIDTH / size;

    problem = new Problem(size, modEnabled);
    cageIDs = problem.getGrid();
    cellCages = problem.getCellCages();

    // Calculate guess offsets
    guess_offset_x = (int) (cellWidth * 0.5 - 8);
    guess_offset_y = guess_offset_x - 7;

    // Clear board
    reset();

    // Generate clue texts
    clueText = new TreeMap<Integer, String>();
    for (Cage c : problem.getCages()) {
      clueText.put(c.getCells().get(0), c.getClueText() + "");
    }
  }

  /**
   * Constantly refresh the window.
   */
  public void gameLoop() {
    while (!Display.isCloseRequested() && running) {
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      Display.sync(60);
      pollInput();
      renderFrame();
      Display.update();
    }
  }

  /**
   * Draw the given problem onto the main window.
   *
   * @param problem
   *           The problem instance
   */
  public void renderFrame() {
    // Draw cageIDs guides
    glColor3f(0.925f, 0.925f, 0.925f);

    for (int i = 1; i < size; ++i) {
      // Horizontal lines
      glBegin(GL_LINES);
      glVertex2i(BOARD_OFFSET_X, BOARD_OFFSET_Y + cellWidth * i);
      glVertex2i(BOARD_OFFSET_X + size * cellWidth, BOARD_OFFSET_Y +
         cellWidth
        * i);
      glEnd();

      // Vertical lines
      glBegin(GL_LINES);
      glVertex2i(BOARD_OFFSET_X + i * cellWidth, BOARD_OFFSET_Y);
      glVertex2i(BOARD_OFFSET_X + i * cellWidth, BOARD_OFFSET_Y +
         cellWidth
        * size);
      glEnd();
    }

    // Highlight errors in red
    for (int i = 0; i < size; ++i) {
      for (int j = 0; j < size; ++j) {
        if (incorrectGrid.get(i * size + j) ||
           incorrectCellCages.get(i).get(j)) {
          glColor3f(1.0f, 0.7f, 0.7f);
          glBegin(GL_QUADS);
          glVertex2f(BOARD_OFFSET_X + j * cellWidth, BOARD_OFFSET_Y + i
            * cellWidth);
          glVertex2f(BOARD_OFFSET_X + (j + 1) * cellWidth,
             BOARD_OFFSET_Y + i
            * cellWidth);
          glVertex2f(BOARD_OFFSET_X + (j + 1) * cellWidth,
             BOARD_OFFSET_Y
            + (i + 1) * cellWidth);
          glVertex2f(BOARD_OFFSET_X + j * cellWidth, BOARD_OFFSET_Y +
             (i + 1)
            * cellWidth);
          glEnd();
        }
      }
    }

    // Draw highlighted cell's background
    if (!isUndo) {
      if (hoverCellX >= 0 && hoverCellX < size && hoverCellY >= 0
        && hoverCellY < size) {
        // Highlight the new cell
        if (inGuessMode) {
          glColor3f(0.8f, 0.8f, 0.8f);
        } else {
          glColor3f(0.7f, 0.7f, 1.0f);
        }
        glBegin(GL_QUADS);
        glVertex2f(BOARD_OFFSET_X + hoverCellX * cellWidth, BOARD_OFFSET_Y
          + hoverCellY * cellWidth);
        glVertex2f(BOARD_OFFSET_X + (hoverCellX + 1) * cellWidth,
          BOARD_OFFSET_Y + hoverCellY * cellWidth);
        glVertex2f(BOARD_OFFSET_X + (hoverCellX + 1) * cellWidth,
          BOARD_OFFSET_Y + (hoverCellY + 1) * cellWidth);
        glVertex2f(BOARD_OFFSET_X + hoverCellX * cellWidth, BOARD_OFFSET_Y
          + (hoverCellY + 1) * cellWidth);
        glEnd();
      }
    }

    // Draw cell walls (note that when traversing the cageIDs in either the
    // left-to-right or top-to-bottom direction, a wall needs to be placed if
    // and only if the current cell belongs to a different cage from the
    // previous cell)
    glColor3f(0.0f, 0.0f, 0.0f);
    int leftNeighborID = 0;
    int topNeighborID = 0;
    for (int i = 0; i < size; ++i) {
      for (int j = 0; j < size; ++j) {
        if (cageIDs.get(j * size + i) != leftNeighborID) {
          glBegin(GL_LINES);
          glVertex2i(BOARD_OFFSET_X + i * cellWidth, BOARD_OFFSET_Y + cellWidth
            * j);
          glVertex2i(BOARD_OFFSET_X + (i + 1) * cellWidth, BOARD_OFFSET_Y
            + cellWidth * j);
          glEnd();
          leftNeighborID = cageIDs.get(j * size + i);
        }
        if (cageIDs.get(i * size + j) != topNeighborID) {
          glBegin(GL_LINES);
          glVertex2i(BOARD_OFFSET_X + j * cellWidth, BOARD_OFFSET_Y + cellWidth
            * i);
          glVertex2i(BOARD_OFFSET_X + j * cellWidth, BOARD_OFFSET_Y + cellWidth
            * (i + 1));
          glEnd();
          topNeighborID = cageIDs.get(i * size + j);
        }
      }
    }

    // Draw board boundaries
    glBegin(GL_LINES); // Top
    glVertex2i(BOARD_OFFSET_X, BOARD_OFFSET_Y);
    glVertex2i(BOARD_OFFSET_X + size * cellWidth, BOARD_OFFSET_Y);
    glEnd();

    glBegin(GL_LINES); // Bottom
    glVertex2i(BOARD_OFFSET_X, BOARD_OFFSET_Y + cellWidth * size);
    glVertex2i(BOARD_OFFSET_X + size * cellWidth, BOARD_OFFSET_Y + cellWidth
      * size);
    glEnd();

    glBegin(GL_LINES); // Left
    glVertex2i(BOARD_OFFSET_X, BOARD_OFFSET_Y);
    glVertex2i(BOARD_OFFSET_X, BOARD_OFFSET_Y + cellWidth * size);
    glEnd();

    glBegin(GL_LINES); // Right
    glVertex2i(BOARD_OFFSET_X + size * cellWidth, BOARD_OFFSET_Y);
    glVertex2i(BOARD_OFFSET_X + size * cellWidth, BOARD_OFFSET_Y + cellWidth
      * size);
    glEnd();

    // All fonts must be rendered last!
    // TODO Make overlay dimensions dependent on text size, not window size
    if (showHelp) {
      // Fade board
      glColor4f(0.0f, 0.0f, 0.0f, 0.8f);
      glBegin(GL_QUADS);
      glVertex2f(0, 0);
      glVertex2f(WINDOW_WIDTH, 0);
      glVertex2f(WINDOW_WIDTH, WINDOW_HEIGHT);
      glVertex2f(0, WINDOW_HEIGHT);
      glEnd();

      // Modal overlay
      glColor3f(1.0f, 1.0f, 1.0f);
      glBegin(GL_QUADS);
      glVertex2f(WINDOW_WIDTH * 0.1f, WINDOW_HEIGHT * 0.13f);
      glVertex2f(WINDOW_WIDTH * 0.9f, WINDOW_HEIGHT * 0.13f);
      glVertex2f(WINDOW_WIDTH * 0.9f, WINDOW_HEIGHT * 0.87f);
      glVertex2f(WINDOW_WIDTH * 0.1f, WINDOW_HEIGHT * 0.87f);
      glEnd();

      helpFont.drawString(HELP_OFFSET_X + WINDOW_WIDTH * 0.1f,
         HELP_OFFSET_Y
        + WINDOW_HEIGHT * 0.13f, HELP_TEXT, Color.black);
      helpFont.drawString(HELP_OFFSET_X + WINDOW_WIDTH * 0.1f + 85,
        HELP_OFFSET_Y + WINDOW_HEIGHT * 0.13f, HELP_DESC, Color.black);
    } else {
      // Draw clue text
      for (Map.Entry<Integer, String> e : clueText.entrySet()) {
        clueFont.drawString(
          BOARD_OFFSET_X + CLUE_OFFSET_X + cellWidth * (e.getKey() %
             size),
          BOARD_OFFSET_Y + CLUE_OFFSET_Y + cellWidth * (e.getKey() /
             size),
          e.getValue(), Color.darkGray);
      }
      // Draw guess text and note text
      for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
          if (guessGrid.get(i * size + j) > 0) {
            guessFont.drawString(BOARD_OFFSET_X + j * cellWidth
              + guess_offset_x,
              BOARD_OFFSET_Y + i * cellWidth + guess_offset_y,
              Integer.toString(guessGrid.get(i * size + j)),
                 Color.black);
          } else {
            for (int k = 0; k < size; ++k) {
              if (noteGrid.get(i * size + j).get(k)) {
                noteFont.drawString(BOARD_OFFSET_X + j * cellWidth
                  + NOTE_OFFSET_X + 12 * (k % 3), BOARD_OFFSET_Y + i
                  * cellWidth + NOTE_OFFSET_Y + 10 * (2 - k / 3),
                  Integer.toString(k + 1), Color.blue);
              }
            }
          }
        }
      }
    }

    // Call this last, after rendering fonts
    GL11.glDisable(GL11.GL_TEXTURE_2D);
  }

  /**
   * Detect user input from keyboard and mouse.
   */
  private void pollInput() {
    // Need "+ cellWidth ... - 1" to make -0.5 round to -1 instead of 0
    hoverCellX = (Mouse.getX() - BOARD_OFFSET_X + cellWidth) /
       cellWidth - 1;
    hoverCellY =
      (WINDOW_HEIGHT - Mouse.getY() - BOARD_OFFSET_Y + cellWidth) /
         cellWidth
        - 1;

    // Draw only if mouse is over board
    while (Keyboard.next()) {
      // Discard keydown events
      if (Keyboard.getEventKeyState()) {
        continue;
      }
      isUndo = false;
      switch (Keyboard.getEventKey()) {
        case Keyboard.KEY_ESCAPE:
          running = false;
          break;
        case Keyboard.KEY_1:
        case Keyboard.KEY_NUMPAD1:
          type(1);
          break;
        case Keyboard.KEY_2:
        case Keyboard.KEY_NUMPAD2:
          type(2);
          break;
        case Keyboard.KEY_3:
        case Keyboard.KEY_NUMPAD3:
          type(3);
          break;
        case Keyboard.KEY_4:
        case Keyboard.KEY_NUMPAD4:
          type(4);
          break;
        case Keyboard.KEY_5:
        case Keyboard.KEY_NUMPAD5:
          type(5);
          break;
        case Keyboard.KEY_6:
        case Keyboard.KEY_NUMPAD6:
          type(6);
          break;
        case Keyboard.KEY_7:
        case Keyboard.KEY_NUMPAD7:
          type(7);
          break;
        case Keyboard.KEY_8:
        case Keyboard.KEY_NUMPAD8:
          type(8);
          break;
        case Keyboard.KEY_9:
        case Keyboard.KEY_NUMPAD9:
          type(9);
          break;
        case Keyboard.KEY_F1:
          showHelp = !showHelp;
          break;
        case Keyboard.KEY_F2:
          showHelp = false;
          reset();
          break;
        case Keyboard.KEY_F3:
          showHelp = false;
          setNewProblem(3);
          break;
        case Keyboard.KEY_F4:
          showHelp = false;
          setNewProblem(4);
          break;
        case Keyboard.KEY_F5:
          showHelp = false;
          setNewProblem(5);
          break;
        case Keyboard.KEY_F6:
          showHelp = false;
          setNewProblem(6);
          break;
        case Keyboard.KEY_F7:
          showHelp = false;
          setNewProblem(7);
          break;
        case Keyboard.KEY_F8:
          showHelp = false;
          setNewProblem(8);
          break;
        case Keyboard.KEY_F9:
          showHelp = false;
          setNewProblem(9);
          break;
        case Keyboard.KEY_F10:
          showHelp = false;
          BruteForceSolver bf = new BruteForceSolver(this, problem);
          bf.startTimer();
          bf.solve();
          bf.stopTimer();
          bf.printElapsedTime();
          Display.setTitle("KenKen - Brute Force Solver took "
            + String.format("%.3f", bf.getElapsedTime() * 0.000000001)
            + " seconds");
          break;
        case Keyboard.KEY_F11:
          showHelp = false;
          DepthFirstSolver dfs = new DepthFirstSolver(this, problem);
          dfs.startTimer();
          dfs.solve();
          dfs.stopTimer();
          dfs.printElapsedTime();
          Display.setTitle("KenKen - DFS Solver took "
            + String.format("%.3f", dfs.getElapsedTime() * 0.000000001)
            + " seconds");
          break;
        case Keyboard.KEY_F12:
          modEnabled = !modEnabled;
          setNewProblem(size);
          break;
        case Keyboard.KEY_BACK:
```

```java
          isUndo = true;
          if (toggleHistory.size() > 0) {
            inGuessMode = toggleHistory.pop();
            hoverCellX = hoverXHistory.pop();
            hoverCellY = hoverYHistory.pop();
            markCell(numHistory.pop());
          }
          break;
        default:
          inGuessMode = !inGuessMode;
          break;
      }
    }
  }

  private void markCell(int n) {
    boolean isRemoval;
    if (boardHovered()) {
      if (inGuessMode) {
        // Mark guess
        if (guessGrid.get(hoverCellY * size + hoverCellX) == n) {
          guessGrid.put(hoverCellY * size + hoverCellX, -1);
          isRemoval = true;
        } else {
          if (!isUndo && guessGrid.get(hoverCellY * size + hoverCellX)
             > 0) {
            boolean tmp1;
            int tmp2;

            tmp1 = toggleHistory.pop();
            toggleHistory.push(inGuessMode);
            toggleHistory.push(tmp1);

            tmp2 = numHistory.pop();
            numHistory.push(guessGrid.get(hoverCellY * size +
               hoverCellX));
            numHistory.push(tmp2);

            tmp2 = hoverXHistory.pop();
            hoverXHistory.push(hoverCellX);
            hoverXHistory.push(tmp2);

            tmp2 = hoverYHistory.pop();
            hoverYHistory.push(hoverCellY);
            hoverYHistory.push(tmp2);
          }

          guessGrid.put(hoverCellY * size + hoverCellX, n);

          // Return if board contains solution
          boolean boardComplete = true;
          int guess;
          HashSet<Integer> guessSet;
          for (int i = 0; i < size * size; ++i) {
            guess = guessGrid.get(i);
            if (guess < 1) {
              boardComplete = false;
              break;
            }
            guessSet = new HashSet<Integer>();
            guessSet.add(guess);
            attempt.put(i, guessSet);
          }
          if (boardComplete && problem.checkGrid(attempt)) {
            Display.setTitle("KenKen - Player solved in "
              + String.format("%.3f",
                (System.nanoTime() - startTime) * 0.000000001) + " seconds!");
            return;
          }

          isRemoval = false;
        }
      } else {
        // Mark note
        // TODO Decide what to do with this.. nice feature but breaks history
        if (!isUndo && guessGrid.get(hoverCellY * size + hoverCellX) > 0) {
          boolean tmp1;
          int tmp2;

          tmp1 = toggleHistory.pop();
          toggleHistory.push(true);
          toggleHistory.push(tmp1);

          tmp2 = numHistory.pop();
          numHistory.push(guessGrid.get(hoverCellY * size + hoverCellX));
          numHistory.push(tmp2);

          tmp2 = hoverXHistory.pop();
          hoverXHistory.push(hoverCellX);
          hoverXHistory.push(tmp2);

          tmp2 = hoverYHistory.pop();
          hoverYHistory.push(hoverCellY);
          hoverYHistory.push(tmp2);
        }
        guessGrid.put(hoverCellY * size + hoverCellX, -1);
        if (noteGrid.get(hoverCellY * size + hoverCellX).get(n - 1)) {
          noteGrid.get(hoverCellY * size + hoverCellX).set(n - 1, false);
          isRemoval = false;
        } else {
          noteGrid.get(hoverCellY * size + hoverCellX).set(n - 1, true);
          isRemoval = true;
        }
      }
      // Verify row
      ArrayList<Integer> currRow = new ArrayList<Integer>();
      for (int i = 0; i < size; ++i) {
        currRow.add(guessGrid.get(hoverCellY * size + i));
      }
      for (int i = 0; i < size; ++i) {
        if (currRow.get(i) < 0) {
          incorrectGrid.put(hoverCellY * size + i, false);
        } else {
          if (currRow.lastIndexOf(Integer.valueOf(currRow.get(i))) !=
             i) {
            incorrectGrid.put(hoverCellY * size + i, true);
            incorrectGrid.put(
              hoverCellY * size
                +
                   currRow.lastIndexOf(Integer.valueOf(currRow.get(i))),
                   true);
          }
          if (Collections.frequency(currRow, currRow.get(i)) < 2
            && incorrectGrid.get(hoverCellY * size + i) == true) {
            incorrectGrid.put(hoverCellY * size + i, false);
          }
        }
      }

      // Verify column
      ArrayList<Integer> currCol = new ArrayList<Integer>();
      for (int i = 0; i < size; ++i) {
        currCol.add(guessGrid.get(i * size + hoverCellX));
      }
      for (int i = 0; i < size; ++i) {
        if (currCol.get(i) < 0) {
          incorrectGrid.put(i * size + hoverCellX, false);
        } else {
          if (currCol.lastIndexOf(Integer.valueOf(currCol.get(i))) !=
             i) {
            incorrectGrid.put(i * size + hoverCellX, true);
            incorrectGrid.put(
              currCol.lastIndexOf(Integer.valueOf(currCol.get(i))) *
                 size
                + hoverCellX, true);

          }
          if (Collections.frequency(currCol, currCol.get(i)) < 2
            && Collections.frequency(currRow, currCol.get(i)) < 2
            && incorrectGrid.get(i * size + hoverCellX) == true) {
            incorrectGrid.put(i * size + hoverCellX, false);
          }
        }
      }

      // Yes, recheck ALL the rows again
```

```java
      ArrayList<Boolean> modifiedCols =
        new ArrayList<Boolean>(Collections.nCopies(size, false));
      for (int j = 0; j < size; ++j) {
        ArrayList<Integer> row = new ArrayList<Integer>();
        for (int m = 0; m < size; ++m) {
          row.add(guessGrid.get(j * size + m));
        }
        for (int k = 0; k < size; ++k) {
          if (row.get(k) < 0) {
            incorrectGrid.put(j * size + k, false);
            modifiedCols.set(k, true);
          } else {
            if (row.lastIndexOf(Integer.valueOf(row.get(k))) != k) {
              incorrectGrid.put(j * size + k, true);
              incorrectGrid.put(
                j * size +
                   row.lastIndexOf(Integer.valueOf(row.get(k))), true);
            }
          }
        }
      }

      // verify all changed columns
      for (int i = 0; i < size; ++i) {
        if (modifiedCols.get(i)) {
          ArrayList<Integer> col = new ArrayList<Integer>();
          for (int j = 0; j < size; ++j) {
            col.add(guessGrid.get(j * size + i));
          }
          for (int k = 0; k < size; ++k) {

            if (col.get(k) < 0) {
              incorrectGrid.put(k * size + i, false);
            } else {
              if (col.lastIndexOf(Integer.valueOf(col.get(k))) != k) {
                incorrectGrid.put(k * size + i, true);

                incorrectGrid.put(col.lastIndexOf(Integer.valueOf(col.get(k)))
                  * size + i, true);
              }
            }
          }
        }
      }

      // verify cell of user input once more
      if (isRemoval) {
        incorrectGrid.put(hoverCellY * size + hoverCellX, false);
      }

      // Deal with cages
      Cage cageToCheck = cellCages.get(hoverCellY * size + hoverCellX);
      if (guessGrid.get(hoverCellY * size + hoverCellX) > -1) {
        if (cageToCheck.isFilled(size, guessGrid)
          && !cageToCheck.isSatisfied(size, guessGrid)) {
          for (Integer i : cageToCheck.getCells()) {
            incorrectCellCages.get(i / size).set(i % size, true);

          }
        } else if (cageToCheck.isFilled(size, guessGrid)
          && cageToCheck.isSatisfied(size, guessGrid)) {
          for (Integer i : cageToCheck.getCells()) {
            incorrectCellCages.get(i / size).set(i % size, false);
          }
        }
      } else {
        for (Integer i : cageToCheck.getCells()) {
          incorrectCellCages.get(i / size).set(i % size, false);
        }
      }
    }
  }

  private boolean boardHovered() {
    return hoverCellX >= 0 && hoverCellX < size && hoverCellY >= 0
      && hoverCellY < size;
  }

  private void type(int n) {
    if (n <= size) {
      numHistory.push(n);
      hoverXHistory.push(hoverCellX);
      hoverYHistory.push(hoverCellY);
      toggleHistory.push(inGuessMode);
      markCell(n);
    } else {
      inGuessMode = !inGuessMode;
    }
  }

  public void showProgress(HashMap<Integer, HashSet<Integer>> state) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    Display.sync(60);
    for (int i = 0; i < size * size; ++i) {
      guessGrid.put(i, (state.get(i).size() == 1) ?
         state.get(i).iterator()
        .next() : -1);
    }
    renderFrame();
    Display.update();
  }

  /**
   * Tear down the window
   */
  public void destroy() {
    Display.destroy();
  }
```

```
}
```

src/GUI.java

```
package edu.virginia.kenken;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Random;

public class Problem {

  private final int size;
  private final HashMap<Integer, Integer> grid;
  private final HashMap<Integer, Integer> solution;
  private int numCages;
  private ArrayList<Cage> cages;
  private final ArrayList<Cage> cellCages;
  private final Random rand;

  public Problem(int size, boolean modEnabled) {
    this.size = size;
    grid = new HashMap<Integer, Integer>();
    numCages = 0;
    cages = new ArrayList<Cage>();
    rand = new Random();
    cellCages =
      new ArrayList<Cage>(Collections.nCopies(size * size, new
          Cage()));
    ArrayList<ArrayList<Integer>> solutionArray =
      new ArrayList<ArrayList<Integer>>();

    // Start with a legal, non-random board

    for (int i = 0; i < size; ++i) {
      solutionArray.add(new ArrayList<Integer>());
      for (int j = 0; j < size; ++j) {
        solutionArray.get(i).add((i + j) % size + 1);
      }
    }

    // Shuffle rows

    Collections.shuffle(solutionArray);

    // Transpose board matrix

    int tmp;
    for (int i = 0; i < size; ++i) {
      for (int j = 0; j < i; ++j) {
        tmp = solutionArray.get(i).get(j);
        solutionArray.get(i).set(j, solutionArray.get(j).get(i));
        solutionArray.get(j).set(i, tmp);
      }
    }

    // Shuffle rows (which were the columns before transposition) again

    Collections.shuffle(solutionArray);

    // Print matrix (for testing only)

    System.out.println("Generated solution:");
    for (int i = 0; i < size; ++i) {
      for (int j = 0; j < size; ++j) {
        System.out.print(solutionArray.get(i).get(j));
      }
      System.out.print("\n");
    }
    System.out.println("");

    // Copy temporary solution arrays into hashmap
    solution = new HashMap<Integer, Integer>();
    for (int i = 0; i < size; ++i) {
      for (int j = 0; j < size; ++j) {
        solution.put(i * size + j, solutionArray.get(i).get(j));
      }
    }

    // Initialize cageIDs

    for (int i = 0; i < size * size; ++i) {
      grid.put(i, -1);
    }

    ArrayList<String> directions = new ArrayList<String>();
    directions.add("N");
    directions.add("E");
    directions.add("S");
    directions.add("W");

    int curID = 0;
    int curX = -1;
    int curY = -1;
    int nextX = -1;
    int nextY = -1;

    int cageSize;
    int maxCageSize = -1;
    float cageCutoff;
    float opCutoff;

    boolean boardFull;
    boolean growable;
```

```java

    // TODONE Remove all references to sizeDistribution (it's just for testing)
    // ArrayList<Integer> sizeDistribution = new ArrayList<Integer>();
    // sizeDistribution.add(0);
    // sizeDistribution.add(0);
    // sizeDistribution.add(0);
    // sizeDistribution.add(0);

    cages = new ArrayList<Cage>();
    Cage cage;

    // ArrayList used to keep track of which cells belong to the current cage
    ArrayList<Integer> cageCells = new ArrayList<Integer>();

    // Each iteration generates a new cage
    while (true) {
      cageCells.clear();
      // Select first available uncaged cell to be "root node" of new cage
      boardFull = true;
      for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
          if (grid.get(i * size + j) < 0) {
            curX = j;
            curY = i;
            boardFull = false;
            break;
          }
        }
        if (!boardFull) {
          break;
        }
      }

      // ...Unless all cells are caged already; then quit
      if (boardFull) {
        break;
      }

      // Predetermine the maximum number of cells this cage will contain,
      // assuming nothing gets in the way of its growth
      cageCutoff = rand.nextFloat();
      if (cageCutoff < 0.07) {
        maxCageSize = 1;
      } else if (cageCutoff < 0.55) {
        maxCageSize = 2;
      } else if (cageCutoff < 0.9) {
        maxCageSize = 3;
      } else {
        maxCageSize = 4;
      }

      // Add current cell to new cage
      cage = new Cage();

      // Add method is used for positioning of the cells based on ID. Do not
      // change!
      cage.add(curY * size + curX);
      cageCells.add(curY * size + curX);
      cage.addPosition(curY, curX);
      cage.addElement(solution.get(curY * size + curX));
      grid.put(curY * size + curX, curID);
      cageSize = 1;

      // Grow cage, cell by cell
      while (true) {
        // Stop when maximum cage size is reached
        if (cageSize >= maxCageSize) {
          break;
        }

        growable = false;

        // Randomly choose growth direction
        Collections.shuffle(directions);
        for (String s : directions) {
          switch (s) {
            case "N":
              nextX = curX;
              nextY = curY - 1;
              break;
            case "E":
              nextX = curX + 1;
              nextY = curY;
              break;
            case "S":
              nextX = curX;
              nextY = curY + 1;
              break;
            case "W":
              nextX = curX - 1;
              nextY = curY;
              break;
          }
          if (nextX >= 0 && nextX < size && nextY >= 0 && nextY < size) {
            if (grid.get(nextY * size + nextX) == -1) {
              growable = true;
              break;
            }
          }
        }

        // If next cell is valid, add it to cage and move to it
        if (growable && cageSize < maxCageSize) {
          cage.add(nextY * size + nextX);
          cageCells.add(nextY * size + nextX);
          cage.addPosition(nextY, nextX);
          cage.addElement(solution.get(nextY * size + nextX));
          grid.put(nextY * size + nextX, curID);
          curX = nextX;
          curY = nextY;
          cageSize += 1;
        } else {
          break;
        }
      }

      // Assign operator to cage
      Cage operationCage;
      switch (cage.getCells().size()) {
        case 1:
          operationCage = new UnitCage(cage);
          break;
        case 2:
          opCutoff = rand.nextFloat();
          if (opCutoff < 0.1) {
            operationCage = new AdditionCage(cage);
          } else if (opCutoff < 0.2) {
            operationCage = new MultiplicationCage(cage);
          } else {
            if (modEnabled) {
              if (opCutoff < 0.5) {
                operationCage = new SubtractionCage(cage);
              } else {
                int smaller = cage.getCellElements().get(0);
                int larger = cage.getCellElements().get(1);
                if (larger < smaller) {
                  int temp = smaller;
                  smaller = larger;
                  larger = temp;
                }
                if (larger % smaller == 0 && opCutoff < 0.95) {
                  operationCage = new DivisionCage(cage);
                } else {
                  operationCage = new ModuloCage(cage);
                }
              }
            } else {
              int smaller = cage.getCellElements().get(0);
              int larger = cage.getCellElements().get(1);
              if (larger < smaller) {
                int temp = smaller;
                smaller = larger;
                larger = temp;
              }
              if (larger % smaller == 0 && opCutoff < 0.95) {
                operationCage = new DivisionCage(cage);
```

```java
              } else {
                operationCage = new SubtractionCage(cage);
              }
            }
          }
          break;
        default:
          operationCage =
            (rand.nextBoolean() ? new MultiplicationCage(cage)
              : new AdditionCage(cage));
          break;
      }
      cages.add(operationCage);

      // Assign each cell, referenced by ID, to the appropriate cage
      for (Integer i : cageCells) {
        cellCages.set(i, operationCage);
      }

      // sizeDistribution
      // .set(cageSize - 1, sizeDistribution.get(cageSize - 1) + 1);
      curID += 1;
    }

    numCages = curID + 1;

    // System.out.println("Number of cages: " + numCages);
    // System.out.println("Cage size distribution: " + sizeDistribution);
  }

  public int getSize() {
    return size;
  }

  public HashMap<Integer, Integer> getGrid() {
    return grid;
  }

  public int getNumCages() {
    return numCages;
  }

  public ArrayList<Cage> getCages() {
    return cages;
  }

  public boolean checkGrid(HashMap<Integer, HashSet<Integer>> attempt) {
    // TODO Ensure rows and columns are also valid
    for (Cage c : cages) {
      if (!c.isSatisfiedHashMapVersion(attempt, size)) {
        return false;
      }
    }

    boolean generatedSolutionFound = true;
    for (int i = 0; i < size; ++i) {
      for (int j = 0; j < size; ++j) {
        if (attempt.get(i * size + j).iterator().next() != solution.get(i
          * size + j)) {
          generatedSolutionFound = false;
          break;
        }
      }
      if (!generatedSolutionFound) {
        break;
      }
    }

    if (generatedSolutionFound) {
      System.out.println("Generated solution found!");
    } else {
      System.out.println("Different solution found!");
    }
    return true;
  }

  // Method to check for valid row and columns
  public boolean checkRowAndColumn(ArrayList<ArrayList<Integer>> attempt) {
    // Create HashSet; when adding duplicates, the add method will return false
    HashSet<Integer> test = new HashSet<Integer>();

    // First check rows
    for (int i = 0; i < size; ++i) {
      test.clear();
      for (int j = 0; j < size; ++j) {
        if (!test.add(attempt.get(i).get(j))) {
          return false;
        }
      }
    }

    // Then check columns
    for (int i = 0; i < size; ++i) {
      test.clear();
      for (int j = 0; j < size; ++j) {
        if (!test.add(attempt.get(j).get(i))) {
          return false;
        }
      }
    }
    return true;
  }

  public ArrayList<Cage> getCellCages() {
    return cellCages;
  }

}
```

src/Problem.java

```java
package edu.virginia.kenken;

public abstract class Solver {
  private final GUI gui;
  private final Problem problem;
  private long startTime;
  private long endTime;
  private long elapsedTime;

  public Solver(GUI gui, Problem problem) {
    this.gui = gui;
    this.problem = problem;
  }

  public GUI getGUI() {
    return gui;
  }

  public Problem getProblem() {
    return problem;
  }

  public void startTimer() {
    startTime = System.nanoTime();
  }

  public void stopTimer() {
    endTime = System.nanoTime();
    elapsedTime = endTime - startTime;
  }

  public void printElapsedTime() {
    System.out.println("Elapsed time: " + elapsedTime * 0.000000001
      + " seconds");
  }

  public long getElapsedTime() {
    return elapsedTime;
  }

}
```

src/Solver.java

```java
package edu.virginia.kenken;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;

public class BruteForceSolver extends Solver {
  private HashMap<Integer, HashSet<Integer>> solution;
  private final int size;
  private long statesChecked;

  public BruteForceSolver(GUI gui, Problem problem) {
    super(gui, problem);

    size = problem.getSize();
    solution = new HashMap<Integer, HashSet<Integer>>();
    statesChecked = -1;
  }

  public void solve() {
    if (solution.size() > 0) {
      System.out.println("The board has already been solved.");
      return;
    }

    HashMap<Integer, HashSet<Integer>> attempt =
      new HashMap<Integer, HashSet<Integer>>();
    HashMap<Integer, HashSet<Integer>> template =
      new HashMap<Integer, HashSet<Integer>>();

    // Start with a legal, non-random board
    ArrayList<Integer> rowPermutation = new ArrayList<Integer>();
    ArrayList<Integer> colPermutation = new ArrayList<Integer>();
    HashSet<Integer> tmp;
    for (int i = 0; i < size; ++i) {
      rowPermutation.add(i + 1);
      colPermutation.add(i + 1);
      for (int j = 0; j < size; ++j) {
        tmp = new HashSet<Integer>();
        tmp.add((i + j) % size + 1);
        attempt.put(i * size + j, tmp);
        template.put(i * size + j, tmp);
      }
    }
    statesChecked = 1;
    while (!getProblem().checkGrid(attempt)) {
      statesChecked += 1;

      if (statesChecked % 65536 == 0) {
        getGUI().showProgress(attempt);
      }

      // Get next permutations of rows and columns
      if (!nextPermutation(rowPermutation)) {
        rowPermutation = new ArrayList<Integer>();
        for (int k = 0; k < size; ++k) {
          rowPermutation.add(k + 1);
        }
        nextPermutation(colPermutation);
      }

      // Reassign attempt grid values as specified by permutations
      for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
          attempt.put(
            i * size + j,
            template.get((colPermutation.get(i) - 1) * size
              + rowPermutation.get(j) - 1));
        }
      }
    }

    solution = attempt;
    getGUI().showProgress(solution);
  }

  // public long getStatesChecked() {
  // return statesChecked;
  // }

  // public HashMap<Integer, HashSet<Integer>> getSolution() {
  // return solution;
  // }

  // public void printSolution() {
  // for (int i = 0; i < size; ++i) {
  // System.out.println(solution.get(i));
  // }
  // }

  /**
   * @param p
   *          Input list
   * @return Whether input is not the last permutation
   */
  private static boolean nextPermutation(ArrayList<Integer> p) {
    int a = p.size() - 2;
    while (a >= 0 && p.get(a) >= p.get(a + 1)) {
      a--;
    }
    if (a < 0) {
      return false;
    }

    int b = p.size() - 1;
    while (p.get(b) <= p.get(a)) {
      b--;
```

```java
    }

    int t = p.get(a);
    p.set(a, p.get(b));
    p.set(b, t);

    for (int i = a + 1, j = p.size() - 1; i < j; ++i, --j) {
      t = p.get(i);
      p.set(i, p.get(j));
      p.set(j, t);
    }
    return true;
  }
}
```

src/BruteForceSolver.java

```java
package edu.virginia.kenken;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;

public class DepthFirstSolver extends Solver {
  private final int size;
  private final ArrayList<Cage> cages;
  private boolean solutionFound;
  private HashMap<Integer, HashSet<Integer>> solution;
  private int statesChecked;
  private HashMap<Integer, Integer> gainScores;

  public DepthFirstSolver(GUI gui, Problem problem) {
    super(gui, problem);

    size = problem.getSize();
    cages = problem.getCages();
    solutionFound = false;
    statesChecked = 0;
  }

  public void solve() {

    // Initialize grid of guesses to all empty
    HashMap<Integer, HashSet<Integer>> root =
      new HashMap<Integer, HashSet<Integer>>();
    for (int i = 0; i < size * size; ++i) {
      root.put(i, new HashSet<Integer>());
      // TODO Make this iterate upwards (currently set to iterate downwards to
      // improve naive information gain, since large cell guesses typically fail
      // faster)
      for (int j = size; j > 0; --j) {
        root.get(i).add(j);
      }
    }
    // Get easy stuff done first - mark all UnitCages and recurse through
    // their peers, marking them if possible too
    for (Cage c : cages) {
      c.preprocess(size, root);
      if (c.getCells().size() == 1) {
        trimPeers(c.getCells().get(0), c.getTotal(), root);
      }
    }

    // Assign expected information gain scores to cells
    gainScores = new HashMap<Integer, Integer>();
    int operationScore = -1;
    for (Cage c : cages) {
      switch (c.getClass().getSimpleName()) {
        case "AdditionCage":
          operationScore = 35;
          break;
        case "DivisionCage":
          operationScore = 50;
          break;
        case "ModuloCage":
          operationScore = 35;
          break;
        case "MultiplicationCage":
          operationScore = 50;
          break;
        case "SubtractionCage":
          operationScore = 35;
          break;
        case "UnitCage":
          operationScore = -1;
          break;
        default:
          System.out.println("Wtf");
          break;
      }

      if (operationScore < 0) {
        continue;
      }

      for (Integer cellID : c.getCells()) {
        gainScores.put(cellID,
          (int) (operationScore - 12 * Math.pow(1.5, c.getNumCells() - 1)));
      }
    }

    // Call the root instance of DFS on the cell with highest info gain
    DFS(maxGain(root), root);

    if (solution == null) {
      System.out.println("No solution found.");
    } else {
      // Update display with current state
      getGUI().showProgress(solution);

      // HashMap<Integer, Integer> matrix = new HashMap<Integer, Integer>();
      // for (int i = 0; i < size; ++i) {
      // for (int j = 0; j < size; ++j) {
      // matrix.put(i * size + j, (solution.get(i * size + j).size() == 1)
      // ? solution.get(i * size + j).iterator().next() : -1);
      // }
      // }
      // getProblem().checkGrid(matrix);
      getProblem().checkGrid(solution);
      System.out.println("States checked: " + statesChecked);
    }
  }

  /**
   * Recursively called DFS algorithm - should be called only on undetermined
   * cells.
   *
   * @param cellID
   * @param state
   */
  private void DFS(int cellID, HashMap<Integer, HashSet<Integer>> state) {
    // Check whether this is a solution
    if (solutionFound) {
      return;
    }

    // Loop through possible values for this cell
    int markedInCage;
    boolean cagesSatisfied;
    HashMap<Integer, HashSet<Integer>> child;

    for (Integer v : state.get(cellID)) {
      // Quit if this branch's left sibling found a solution
      if (solutionFound) {
        return;
      }

      statesChecked += 1;
      if (statesChecked % 4096 == 0) {
        // Update display with current state
        getGUI().showProgress(state);
      }

      // Copy parent state into a new child state
      child = cloneState(state);

      // Mark cell with DFS hypothesis
      child.get(cellID).clear();
      child.get(cellID).add(v);

      // Trim peers
      trimPeers(cellID, v, child);

      // Check for cage conflicts (note that we don't need to check for
      // row/column conflicts since we previously called
        makeAndTrimPeers on the
      // HashSet we're iterating through)
      cagesSatisfied = true;
      for (Cage c : cages) {
        if (!cagesSatisfied) {
          break;
        }

        // Check this cage
        markedInCage = 0;
        for (Integer i : c.getCells()) {
          if (child.get(i).size() < 1) {
            // This might occur if a wrong solution is given to
              trimPeers
            cagesSatisfied = false;
            break;
          }
          if (child.get(i).size() == 1) {
            markedInCage += 1;
          }
        }
        if (cagesSatisfied && markedInCage == c.getNumCells()) {
          if (!c.isSatisfiedHashMapVersion(child, size)) {
            cagesSatisfied = false;
            break;
          }
        }
      }
      if (!cagesSatisfied) {
        continue;
      }

      // Check whether child is solution
      if (isSolution(child)) {
        solution = child;
        solutionFound = true;
        return;
      }

      // Recursively call DFS
      DFS(maxGain(child), child);
```

```
    }
  }

  /**
   * Mark the given cell, remove its value from its peers' sets of possible
   * values, and recursively continue marking peers whose sizes of sets of
   * possible values become 1.
   *
   * @param cellID
   *          Cell to mark
   * @param value
   *          Value to mark
   * @param state
   *          Current state
   */
  private void trimPeers(int cellID, int value,
    HashMap<Integer, HashSet<Integer>> state) {
    int row = cellID / size;
    int col = cellID % size;
    int peerID;

    // Trim this cell's designated value from its peer cells
    // TODO Factor out the common loop bodies
    for (int i = 0; i < size; ++i) {
      peerID = row * size + i;
      if (peerID != cellID) {
        if (state.get(peerID).remove(value)) {
          // Peer newly became determined, so trim *its* peers
          if (state.get(peerID).size() == 1) {
            trimPeers(peerID, state.get(peerID).iterator().next(), state);
          }
        }
      }

      peerID = size * i + col;
      if (peerID != cellID) {
        if (state.get(peerID).remove(value)) {
          // Peer newly became determined, so trim *its* peers
          if (state.get(peerID).size() == 1) {
            trimPeers(peerID, state.get(peerID).iterator().next(), state);
          }
        }
      }
    }
  }

  /**
   * Check whether all cells in the state have 1 possible value.
   *
   * @param state
```

```java
   * @return Whether state is a solution
   */
  private boolean isSolution(HashMap<Integer, HashSet<Integer>> state) {
    boolean allCellsMarked = true;
    for (HashSet<Integer> s : state.values()) {
      if (s.size() > 1) {
        allCellsMarked = false;
        break;
      }
    }
    return allCellsMarked;
  }

  private HashMap<Integer, HashSet<Integer>> cloneState(
    HashMap<Integer, HashSet<Integer>> state) {
    HashMap<Integer, HashSet<Integer>> clone =
      new HashMap<Integer, HashSet<Integer>>();
    HashSet<Integer> possibleValues;
    for (Integer i : state.keySet()) {
      possibleValues = new HashSet<Integer>();
      for (Integer j : state.get(i)) {
        possibleValues.add(j);
      }
      clone.put(i, possibleValues);
    }
    return clone;
  }

  private int maxGain(HashMap<Integer, HashSet<Integer>> state) {
    // for (int i = 0; i < size * size; ++i) {
    // if (state.get(i).size() > 1) {
    // return i;
    // }
    // }
    // return -1;
    int maxGain = -1;
    int cellID = -1;
    int gain;
    for (int i = 0; i < size * size; ++i) {
      if (state.get(i).size() > 1) {
        gain = gainScores.get(i) + 700 * (size - state.get(i).size()) / size;
        if (gain > maxGain) {
          maxGain = gain;
          cellID = i;
        }
      }
    }
    return cellID;
  }
}
```

src/DepthFirstSolver.java

```java
package edu.virginia.kenken;

import java.util.ArrayList;

public class Constraint {
  private ArrayList<Integer> cells;
  private final ArrayList<Integer> cellElements;
  private ArrayList<Integer> cellPositions;

  public Constraint() {
    // Contains the cells (row-major) that this cage holds
    cells = new ArrayList<Integer>();
    // Note: cellElements is only applicable to cages that have not been
    // assigned to operations yet
    cellElements = new ArrayList<Integer>();
    // Stores the position of each cell in the cage in alternating col, row
    // order
    cellPositions = new ArrayList<Integer>();
  }

  public ArrayList<Integer> getCells() {
    return cells;
  }

  public void setCells(ArrayList<Integer> cells) {
    this.cells = cells;
  }

  public void setCellPositions(ArrayList<Integer> cellPositions) {
    this.cellPositions = cellPositions;
  }

  public void add(Integer cellID) {
    cells.add(cellID);
  }

  public void addPosition(Integer cellX, Integer cellY) {
    cellPositions.add(cellX);
    cellPositions.add(cellY);
  }

  public void addElement(Integer cellVal) {
    cellElements.add(cellVal);
  }

  public ArrayList<Integer> getCellElements() {
    return cellElements;
  }

  public ArrayList<Integer> getCellPositions() {
    return cellPositions;
  }

  public int getNumCells() {
    return cells.size();
  }

}
```

src/Constraint.java

```java
package edu.virginia.kenken;

import java.util.HashMap;
import java.util.HashSet;

public class Cage extends Constraint {
  private int total;

  public Cage() {
    super();
  }

  public Cage(Cage src) {
    super();
    setCells(src.getCells());
    setCellPositions(src.getCellPositions());
  }

  public String getClueText() {
    return Integer.toString(total);
  }

  public int getTotal() {
    return total;
  }

  public void setTotal(int total) {
    this.total = total;
  }

  public void preprocess(int size, HashMap<Integer, HashSet<Integer>>
      state) {
    return;
  }

  public boolean isSatisfiedHashMapVersion(
    HashMap<Integer, HashSet<Integer>> state, int size) {
    System.out.println("This was supposed to be abstract.");
    return false;
  }

  public boolean isSatisfied(int size, HashMap<Integer, Integer>
      entryGrid) {
    System.out.println("This was supposed to be abstract.");
    return false;
  }

  // TODO Make size a field instead of a parameter
  public boolean isFilled(int size, HashMap<Integer, Integer>
      entryGrid) {
    for (int i = 0; i < getCellPositions().size(); i = i + 2) {
      if (entryGrid.get(getCellPositions().get(i) * size
        + getCellPositions().get(i + 1)) < 1) {
        return false;
      }
    }
    return true;
  }
}
```

src/Cage.java

```java
package edu.virginia.kenken;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;

public class AdditionCage extends Cage {
  public AdditionCage(Cage src) {
    super(src);
    int sum = 0;
    for (Integer d : src.getCellElements()) {
      sum += d;
    }
    setTotal(sum);
  }

  @Override
  public String getClueText() {
    return getTotal() + "+";
  }

  @Override
  public void preprocess(int size, HashMap<Integer, HashSet<Integer>>
      state) {
    Iterator<Integer> it;
    int minPossible = getTotal() - size * (getNumCells() - 1);
    int value;
    for (Integer cellID : getCells()) {
      it = state.get(cellID).iterator();
      while (it.hasNext()) {
        value = it.next();
        if (value >= getTotal() || value < minPossible) {
          it.remove();
        }
      }
    }
    return;
  }

  @Override
  public boolean isSatisfiedHashMapVersion(
    HashMap<Integer, HashSet<Integer>> entryGrid, int size) {
    int guessSum = 0;
    for (int i = 0; i < getCellPositions().size(); i = i + 2) {
      guessSum +=
        entryGrid
          .get(getCellPositions().get(i) * size +
             getCellPositions().get(i + 1))
          .iterator().next();
    }
    return (guessSum == getTotal());
  }

  @Override
  public boolean isSatisfied(int size, HashMap<Integer, Integer>
     entryGrid) {
    int guessSum = 0;
    for (int i = 0; i < getCellPositions().size(); i = i + 2) {
      guessSum +=
        entryGrid.get(getCellPositions().get(i) * size
          + getCellPositions().get(i + 1));
    }
    return (guessSum == getTotal());
  }
}
```

src/AdditionCage.java

```java
package edu.virginia.kenken;

import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;

public class DivisionCage extends Cage {
  public DivisionCage(Cage src) {
    super(src);
    setTotal(Collections.max(src.getCellElements())
      / Collections.min(src.getCellElements()));
  }

  @Override
  public void preprocess(int size, HashMap<Integer, HashSet<Integer>>
     state) {
    Iterator<Integer> it;
    int value;
    for (Integer cellID : getCells()) {
      it = state.get(cellID).iterator();
      while (it.hasNext()) {
        value = it.next();
        if (value * getTotal() > size && value > getTotal()
          && value % getTotal() > 0) {
          it.remove();
        }
      }
    }
    return;
  }

  @Override
  public String getClueText() {
    return getTotal() + "/";
  }

  @Override
  public boolean isSatisfiedHashMapVersion(
    HashMap<Integer, HashSet<Integer>> entryGrid, int size) {
    int a =
      entryGrid
        .get(getCellPositions().get(0) * size +
           getCellPositions().get(1))
        .iterator().next();
    int b =
      entryGrid
        .get(getCellPositions().get(2) * size +
           getCellPositions().get(3))
        .iterator().next();
    return (Math.max(a, b) % Math.min(a, b) == 0 && Math.max(a, b)
      / Math.min(a, b) == getTotal());
  }

  @Override
  public boolean isSatisfied(int size, HashMap<Integer, Integer>
     entryGrid) {
    int a =
      entryGrid.get(getCellPositions().get(0) * size
        + getCellPositions().get(1));
    int b =
      entryGrid.get(getCellPositions().get(2) * size
        + getCellPositions().get(3));
    return (Math.max(a, b) / Math.min(a, b) == getTotal());
  }
}
```

src/DivisionCage.java

```
package edu.virginia.kenken;

import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;

public class ModuloCage extends Cage {
  public ModuloCage(Cage src) {
    super(src);
    setTotal(Collections.max(src.getCellElements())
      % Collections.min(src.getCellElements()));
  }

  @Override
  public String getClueText() {
    return getTotal() + "%";
  }

  @Override
  public void preprocess(int size, HashMap<Integer, HashSet<Integer>>
     state) {
    Iterator<Integer> it;
    int value;
    for (Integer cellID : getCells()) {
      it = state.get(cellID).iterator();
      while (it.hasNext()) {
        value = it.next();
        if (value > size - getTotal() && value <= getTotal()) {
          it.remove();
        }
      }
    }
    return;
  }

  @Override
  public boolean isSatisfiedHashMapVersion(
    HashMap<Integer, HashSet<Integer>> entryGrid, int size) {
    int a =
      entryGrid
        .get(getCellPositions().get(0) * size +
           getCellPositions().get(1))
        .iterator().next();
    int b =
      entryGrid
        .get(getCellPositions().get(2) * size +
           getCellPositions().get(3))
        .iterator().next();
    return (Math.max(a, b) % Math.min(a, b) == getTotal());
  }

  @Override
```

```java
  public boolean isSatisfied(int size, HashMap<Integer, Integer>
     entryGrid) {
    int a =
      entryGrid.get(getCellPositions().get(0) * size
        + getCellPositions().get(1));
    int b =
      entryGrid.get(getCellPositions().get(2) * size
        + getCellPositions().get(3));
    return (Math.max(a, b) % Math.min(a, b) == getTotal());
  }

}
```

src/ModuloCage.java

```java
package edu.virginia.kenken;

import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;

public class MultiplicationCage extends Cage {
  public MultiplicationCage(Cage src) {
    super(src);
    int product = 1;
    for (Integer d : src.getCellElements()) {
      product *= d;
    }
    setTotal(product);
  }

  @Override
  public String getClueText() {
    return getTotal() + "x";
  }

  @Override
  public void preprocess(int size, HashMap<Integer, HashSet<Integer>>
     state) {
    Iterator<Integer> it;
    int value;
    int minPossible =
      (int) Math.ceil(getTotal() / Math.pow(size, getTotal() - 1));

    for (Integer cellID : getCells()) {
      it = state.get(cellID).iterator();
      while (it.hasNext()) {
        value = it.next();
        if (getTotal() % value > 0 || value > getTotal() || value <
           minPossible) {
          it.remove();
        }
      }
    }
    return;
  }

  @Override
  public boolean isSatisfiedHashMapVersion(
    HashMap<Integer, HashSet<Integer>> entryGrid, int size) {
    int guessProduct = 1;
    for (int i = 0; i < getCellPositions().size(); i = i + 2) {
      guessProduct *=
        entryGrid
          .get(getCellPositions().get(i) * size +
             getCellPositions().get(i + 1))
          .iterator().next();
    }
    return (guessProduct == getTotal());
  }

  @Override
  public boolean isSatisfied(int size, HashMap<Integer, Integer>
     entryGrid) {
    int guessProduct = 1;
    for (int i = 0; i < getCellPositions().size(); i = i + 2) {
      guessProduct *=
        entryGrid.get(getCellPositions().get(i) * size
          + getCellPositions().get(i + 1));
    }
    return (guessProduct == getTotal());
  }
}
```

src/MultiplicationCage.java

```
package edu.virginia.kenken;

import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Iterator;

public class SubtractionCage extends Cage {
  public SubtractionCage(Cage src) {
    super(src);
    setTotal(Collections.max(src.getCellElements())
      - Collections.min(src.getCellElements()));
  }

  @Override
  public String getClueText() {
    return getTotal() + "-";
  }
```

```java

  @Override
  public void preprocess(int size, HashMap<Integer, HashSet<Integer>> state) {
    Iterator<Integer> it;
    int value;
    for (Integer cellID : getCells()) {
      it = state.get(cellID).iterator();
      while (it.hasNext()) {
        value = it.next();
        if (value > size - getTotal() && value <= getTotal()) {
          it.remove();
        }
      }
    }
    return;
  }

  @Override
  public boolean isSatisfiedHashMapVersion(
    HashMap<Integer, HashSet<Integer>> entryGrid, int size) {
    return (Math.abs(entryGrid
      .get(getCellPositions().get(0) * size + getCellPositions().get(1))
      .iterator().next()
      - entryGrid
        .get(getCellPositions().get(2) * size + getCellPositions().get(3))
        .iterator().next()) == getTotal());
  }

  @Override
  public boolean isSatisfied(int size, HashMap<Integer, Integer> entryGrid) {
    return (Math.abs(entryGrid.get(getCellPositions().get(0) * size
      + getCellPositions().get(1))
      - entryGrid.get(getCellPositions().get(2) * size
        + getCellPositions().get(3))) == getTotal());
  }
}
```

src/SubtractionCage.java

```java
package edu.virginia.kenken;

import java.util.HashMap;
import java.util.HashSet;

public class UnitCage extends Cage {
  public UnitCage(Cage src) {
    super(src);
    setTotal(src.getCellElements().get(0));
  }

  @Override
  public String getClueText() {
    return getTotal() + "";
  }

  @Override
  public void preprocess(int size, HashMap<Integer, HashSet<Integer>> state) {
    state.get(getCells().get(0)).clear();
    state.get(getCells().get(0)).add(getTotal());
    return;
  }

  @Override
  public boolean isSatisfiedHashMapVersion(
    HashMap<Integer, HashSet<Integer>> entryGrid, int size) {
    return (entryGrid
      .get(getCellPositions().get(0) * size + getCellPositions().get(1))
      .iterator().next() == getTotal());
  }

  @Override
  public boolean isSatisfied(int size, HashMap<Integer, Integer> entryGrid) {
    return (entryGrid.get(getCellPositions().get(0) * size
      + getCellPositions().get(1)) == getTotal());
  }
}
```

src/UnitCage.java