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.

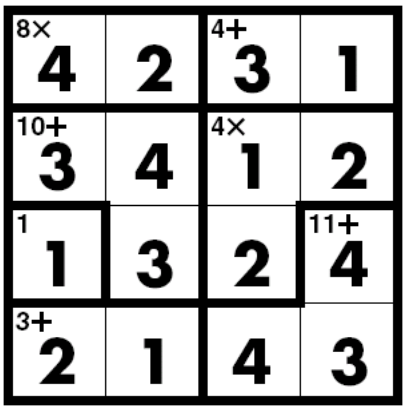

Figure 1: Example of a solved 4×4 puzzle with seven cages.

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.

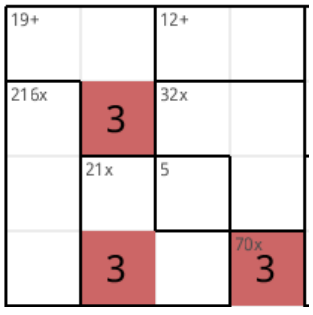

Figure 2: Row and column constraints not satisfied.

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.

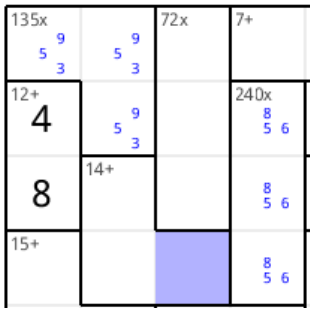

Figure 3: Note mode.

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 {

```

```
30    // Board constants
31    private static final int WINDOW_WIDTH = 480;
32    private static final int WINDOW_HEIGHT = 480;
33    private static final int BOARD_WIDTH = WINDOW_HEIGHT - 30;
34    private static final float LINE_WIDTH = 2.0f;
35    private static final int BOARD_OFFSET_X = 15;
36    private static final int BOARD_OFFSET_Y = 15;
37
38    // Clue constants
39    private static final int CLUE_OFFSET_X = 3;
40    private static final int CLUE_OFFSET_Y = 1;
41    private static final int CLUE_FONT_SIZE = 12;
42
43    // Guess variables
44    private int guess_offset_x;
45    private int guess_offset_y;
46    private static final int GUESS_FONT_SIZE = 25;
47
48    // Note constants
49    private static final int NOTE_OFFSET_X = 10;
50    private static final int NOTE_OFFSET_Y = 15;
51    private static final int NOTE_FONT_SIZE = 10;
52
53    // Help text constants
54    private static final int HELP_OFFSET_X = 19;
55    private static final int HELP_OFFSET_Y = 11;
56    private static final int HELP_FONT_SIZE = 20;
57    private static final String HELP_TEXT = "ESC:\n" + "F1:\n" + "F2:\n"
58      + "F3:\n" + "F4:\n" + "F5:\n" + "F6:\n" + "F7:\n" + "F8:\n" +
          "F9:\n"
59      + "F10:\n" + "F11:\n" + "F12:\n" + "OTHER:";
60    private static final String HELP_DESC = "EXIT\n" + "HELP\n" +
        "RESET\n"
61      + "NEW 3x3 PUZZLE\n" + "NEW 4x4 PUZZLE\n" + "NEW 5x5 PUZZLE\n"
62      + "NEW 6x6 PUZZLE\n" + "NEW 7x7 PUZZLE\n" + "NEW 8x8 PUZZLE\n"
63      + "NEW 9x9 PUZZLE\n" + "SOLVE (BRUTE FORCE)\n" + "SOLVE (DFS)\n"
64      + "ENABLE/DISABLE % CAGES\n" + "TOGGLE GUESS/NOTE MODE";
65
66    private static final String FONT_PATH = "res/DroidSans.ttf";
67
68    // Current problem
69    private Problem problem;
70
71    // Height (or width) of problem in cells
72    private int size;
73
74    // Grid of cage IDs
75    HashMap<Integer, Integer> cageIDs;
76
77    // Cell and cages relationship
78    private ArrayList<Cage> cellCages;
79
80    // Pixel width of a cell
81    private int cellWidth;
```

```
82
83      // Number fonts
84      private UnicodeFont clueFont;
85      private UnicodeFont guessFont;
86      private UnicodeFont noteFont;
87
88      // Help font
89      private UnicodeFont helpFont;
90
91      // Matrix of user's cell guesses
92      private HashMap<Integer, Integer> guessGrid;
93
94      // Matrix of user's cell notes
95      private HashMap<Integer, ArrayList<Boolean>> noteGrid;
96
97      // Matrix of incorrect cells
98      private HashMap<Integer, Boolean> incorrectGrid;
99
100     // Matrix of incorrect cell (cage)
101     private ArrayList<ArrayList<Boolean>> incorrectCellCages;
102
103     // Maps clue cells to clue text
104     private TreeMap<Integer, String> clueText;
105
106     // Guess/note history
107     private Stack<Integer> numHistory;
108     private Stack<Boolean> toggleHistory;
109     private Stack<Integer> hoverXHistory;
110     private Stack<Integer> hoverYHistory;
111
112     // Grid indices of the currently hovered cell
113     private int hoverCellX;
114     private int hoverCellY;
115
116     // Whether entry mode is "guess" or "note"
117     private boolean inGuessMode;
118
119     // Whether or not to show help on the board
120     private boolean showHelp;
121
122     // Whether or not problems with modulo cages can be generated
123     private boolean modEnabled;
124
125     // Whether main loop should be running
126     private boolean running;
127
128     // Used for checking whether player-filled board is solution
129     private HashMap<Integer, HashSet<Integer>> attempt;
130
131     // Used for displaying time player took to solve puzzle
132     private long startTime;
133
134     // Whether current guess/note entry is actually an undo action
135     private boolean isUndo;
```

```
136
137   public GUI(int startupSize) {
138     running = true;
139     modEnabled = false;
140     init();
141     setNewProblem(startupSize);
142   }
143
144   /**
145    * Initialize LWJGL and create the window.
146    */
147   @SuppressWarnings("unchecked")
148   private void init() {
149     // Create window
150     try {
151       Display.setDisplayMode(new DisplayMode(WINDOW_WIDTH,
              WINDOW_HEIGHT));
152       Display.setTitle("KenKen");
153       Display.create();
154     } catch (LWJGLException e) {
155       System.err.println("Display wasn't initialized correctly.");
156       System.exit(1);
157     }
158
159     // Create keyboard
160     try {
161       Keyboard.create();
162     } catch (LWJGLException e) {
163       System.out.println("Keyboard could not be created.");
164       System.exit(1);
165     }
166
167     glEnable(GL_TEXTURE_2D);
168     glShadeModel(GL_SMOOTH);
169     glDisable(GL_DEPTH_TEST);
170     glDisable(GL_LIGHTING);
171     glEnable(GL_BLEND);
172     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
173     glMatrixMode(GL_PROJECTION);
174     glLoadIdentity();
175     glOrtho(0, WINDOW_WIDTH, WINDOW_HEIGHT, 0, 1, -1);
176     glMatrixMode(GL_MODELVIEW);
177     glEnable(GL_COLOR_MATERIAL);
178
179     // Set background color to white
180     glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
181     glClear(GL_COLOR_BUFFER_BIT);
182
183     // Line thickness
184     glLineWidth(LINE_WIDTH);
185
186     try {
187       // Temporarily disable System.out
188       // System.setOut(new PrintStream(new OutputStream() {
```

```
189         // @Override
190         // public void write(int b) {
191         // // Do nothing
192         // }
193         // }));
194
195         clueFont = new UnicodeFont(FONT_PATH, CLUE_FONT_SIZE, false,
                false);
196         clueFont.addAsciiGlyphs();
197         clueFont.addGlyphs(400, 600);
198         clueFont.getEffects().add(new ColorEffect());
199         clueFont.loadGlyphs();
200
201         guessFont = new UnicodeFont(FONT_PATH, GUESS_FONT_SIZE, false,
                false);
202         guessFont.addAsciiGlyphs();
203         guessFont.addGlyphs(400, 600);
204         guessFont.getEffects().add(new ColorEffect());
205         guessFont.loadGlyphs();
206
207         noteFont = new UnicodeFont(FONT_PATH, NOTE_FONT_SIZE, false,
                false);
208         noteFont.addAsciiGlyphs();
209         noteFont.addGlyphs(400, 600);
210         noteFont.getEffects().add(new ColorEffect());
211         noteFont.loadGlyphs();
212
213         helpFont = new UnicodeFont(FONT_PATH, HELP_FONT_SIZE, false,
                false);
214         helpFont.addAsciiGlyphs();
215         helpFont.addGlyphs(400, 600);
216         helpFont.getEffects().add(new ColorEffect());
217         helpFont.loadGlyphs();
218
219         // Re-enable System.out
220         // System.setOut(System.out);
221
222       } catch (SlickException e) {
223         System.out.println("Failed to create font. Exiting.");
224         e.printStackTrace();
225         System.exit(1);
226       }
227     }
228
229     private void reset() {
230       guessGrid = new HashMap<Integer, Integer>();
231       noteGrid = new HashMap<Integer, ArrayList<Boolean>>();
232       incorrectGrid = new HashMap<Integer, Boolean>();
233       incorrectCellCages = new ArrayList<ArrayList<Boolean>>();
234       attempt = new HashMap<Integer, HashSet<Integer>>();
235       for (int i = 0; i < size; ++i) {
236         incorrectCellCages.add(new ArrayList<Boolean>());
237         for (int j = 0; j < size; ++j) {
238           guessGrid.put(i * size + j, -1);
```

```
239         noteGrid.put(i * size + j,
240           new ArrayList<Boolean>(Collections.nCopies(size, false)));
241         incorrectGrid.put(i * size + j, false);
242         incorrectCellCages.get(i).add(false);
243       }
244     }
245
246     inGuessMode = true;
247     numHistory = new Stack<Integer>();
248     toggleHistory = new Stack<Boolean>();
249     hoverXHistory = new Stack<Integer>();
250     hoverYHistory = new Stack<Integer>();
251
252     Display.setTitle("KenKen");
253     startTime = System.nanoTime();
254   }
255
256   /*
257    * Load a new problem instance into the main window.
258    */
259   private void setNewProblem(int size) {
260     this.size = size;
261     cellWidth = BOARD_WIDTH / size;
262
263     problem = new Problem(size, modEnabled);
264     cageIDs = problem.getGrid();
265     cellCages = problem.getCellCages();
266
267     // Calculate guess offsets
268     guess_offset_x = (int) (cellWidth * 0.5 - 8);
269     guess_offset_y = guess_offset_x - 7;
270
271     // Clear board
272     reset();
273
274     // Generate clue texts
275     clueText = new TreeMap<Integer, String>();
276     for (Cage c : problem.getCages()) {
277       clueText.put(c.getCells().get(0), c.getClueText() + "");
278     }
279   }
280
281   /**
282    * Constantly refresh the window.
283    */
284   public void gameLoop() {
285     while (!Display.isCloseRequested() && running) {
286       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
287       Display.sync(60);
288       pollInput();
289       renderFrame();
290       Display.update();
291     }
292   }
```

```
293
294   /**
295    * Draw the given problem onto the main window.
296    *
297    * @param problem
298    *          The problem instance
299    */
300   public void renderFrame() {
301     // Draw cageIDs guides
302     glColor3f(0.925f, 0.925f, 0.925f);
303
304     for (int i = 1; i < size; ++i) {
305       // Horizontal lines
306       glBegin(GL_LINES);
307       glVertex2i(BOARD_OFFSET_X, BOARD_OFFSET_Y + cellWidth * i);
308       glVertex2i(BOARD_OFFSET_X + size * cellWidth, BOARD_OFFSET_Y +
              cellWidth
309         * i);
310       glEnd();
311
312       // Vertical lines
313       glBegin(GL_LINES);
314       glVertex2i(BOARD_OFFSET_X + i * cellWidth, BOARD_OFFSET_Y);
315       glVertex2i(BOARD_OFFSET_X + i * cellWidth, BOARD_OFFSET_Y +
              cellWidth
316         * size);
317       glEnd();
318     }
319
320     // Highlight errors in red
321     for (int i = 0; i < size; ++i) {
322       for (int j = 0; j < size; ++j) {
323         if (incorrectGrid.get(i * size + j) ||
               incorrectCellCages.get(i).get(j)) {
324           glColor3f(1.0f, 0.7f, 0.7f);
325           glBegin(GL_QUADS);
326           glVertex2f(BOARD_OFFSET_X + j * cellWidth, BOARD_OFFSET_Y + i
327             * cellWidth);
328           glVertex2f(BOARD_OFFSET_X + (j + 1) * cellWidth,
                BOARD_OFFSET_Y + i
329             * cellWidth);
330           glVertex2f(BOARD_OFFSET_X + (j + 1) * cellWidth,
                BOARD_OFFSET_Y
331             + (i + 1) * cellWidth);
332           glVertex2f(BOARD_OFFSET_X + j * cellWidth, BOARD_OFFSET_Y +
                (i + 1)
333             * cellWidth);
334           glEnd();
335         }
336       }
337     }
338
339     // Draw highlighted cell's background
340     if (!isUndo) {
```

```
341         if (hoverCellX >= 0 && hoverCellX < size && hoverCellY >= 0
342           && hoverCellY < size) {
343           // Highlight the new cell
344           if (inGuessMode) {
345             glColor3f(0.8f, 0.8f, 0.8f);
346           } else {
347             glColor3f(0.7f, 0.7f, 1.0f);
348           }
349           glBegin(GL_QUADS);
350           glVertex2f(BOARD_OFFSET_X + hoverCellX * cellWidth,
                BOARD_OFFSET_Y
351             + hoverCellY * cellWidth);
352           glVertex2f(BOARD_OFFSET_X + (hoverCellX + 1) * cellWidth,
353             BOARD_OFFSET_Y + hoverCellY * cellWidth);
354           glVertex2f(BOARD_OFFSET_X + (hoverCellX + 1) * cellWidth,
355             BOARD_OFFSET_Y + (hoverCellY + 1) * cellWidth);
356           glVertex2f(BOARD_OFFSET_X + hoverCellX * cellWidth,
                BOARD_OFFSET_Y
357             + (hoverCellY + 1) * cellWidth);
358           glEnd();
359         }
360       }
361
362       // Draw cell walls (note that when traversing the cageIDs in
            either the
363       // left-to-right or top-to-bottom direction, a wall needs to be
            placed if
364       // and only if the current cell belongs to a different cage from
            the
365       // previous cell)
366       glColor3f(0.0f, 0.0f, 0.0f);
367       int leftNeighborID = 0;
368       int topNeighborID = 0;
369       for (int i = 0; i < size; ++i) {
370         for (int j = 0; j < size; ++j) {
371           if (cageIDs.get(j * size + i) != leftNeighborID) {
372             glBegin(GL_LINES);
373             glVertex2i(BOARD_OFFSET_X + i * cellWidth, BOARD_OFFSET_Y +
                  cellWidth
374               * j);
375             glVertex2i(BOARD_OFFSET_X + (i + 1) * cellWidth,
                  BOARD_OFFSET_Y
376               + cellWidth * j);
377             glEnd();
378             leftNeighborID = cageIDs.get(j * size + i);
379           }
380           if (cageIDs.get(i * size + j) != topNeighborID) {
381             glBegin(GL_LINES);
382             glVertex2i(BOARD_OFFSET_X + j * cellWidth, BOARD_OFFSET_Y +
                  cellWidth
383               * i);
384             glVertex2i(BOARD_OFFSET_X + j * cellWidth, BOARD_OFFSET_Y +
                  cellWidth
385               * (i + 1));
```

```
386             glEnd();
387             topNeighborID = cageIDs.get(i * size + j);
388           }
389         }
390       }
391
392       // Draw board boundaries
393       glBegin(GL_LINES); // Top
394       glVertex2i(BOARD_OFFSET_X, BOARD_OFFSET_Y);
395       glVertex2i(BOARD_OFFSET_X + size * cellWidth, BOARD_OFFSET_Y);
396       glEnd();
397
398       glBegin(GL_LINES); // Bottom
399       glVertex2i(BOARD_OFFSET_X, BOARD_OFFSET_Y + cellWidth * size);
400       glVertex2i(BOARD_OFFSET_X + size * cellWidth, BOARD_OFFSET_Y +
              cellWidth
401         * size);
402       glEnd();
403
404       glBegin(GL_LINES); // Left
405       glVertex2i(BOARD_OFFSET_X, BOARD_OFFSET_Y);
406       glVertex2i(BOARD_OFFSET_X, BOARD_OFFSET_Y + cellWidth * size);
407       glEnd();
408
409       glBegin(GL_LINES); // Right
410       glVertex2i(BOARD_OFFSET_X + size * cellWidth, BOARD_OFFSET_Y);
411       glVertex2i(BOARD_OFFSET_X + size * cellWidth, BOARD_OFFSET_Y +
              cellWidth
412         * size);
413       glEnd();
414
415       // All fonts must be rendered last!
416       // TODO Make overlay dimensions dependent on text size, not window
              size
417       if (showHelp) {
418         // Fade board
419         glColor4f(0.0f, 0.0f, 0.0f, 0.8f);
420         glBegin(GL_QUADS);
421         glVertex2f(0, 0);
422         glVertex2f(WINDOW_WIDTH, 0);
423         glVertex2f(WINDOW_WIDTH, WINDOW_HEIGHT);
424         glVertex2f(0, WINDOW_HEIGHT);
425         glEnd();
426
427         // Modal overlay
428         glColor3f(1.0f, 1.0f, 1.0f);
429         glBegin(GL_QUADS);
430         glVertex2f(WINDOW_WIDTH * 0.1f, WINDOW_HEIGHT * 0.13f);
431         glVertex2f(WINDOW_WIDTH * 0.9f, WINDOW_HEIGHT * 0.13f);
432         glVertex2f(WINDOW_WIDTH * 0.9f, WINDOW_HEIGHT * 0.87f);
433         glVertex2f(WINDOW_WIDTH * 0.1f, WINDOW_HEIGHT * 0.87f);
434         glEnd();
435
436         helpFont.drawString(HELP_OFFSET_X + WINDOW_WIDTH * 0.1f,
```

```
          HELP_OFFSET_Y
437         + WINDOW_HEIGHT * 0.13f, HELP_TEXT, Color.black);
438       helpFont.drawString(HELP_OFFSET_X + WINDOW_WIDTH * 0.1f + 85,
439         HELP_OFFSET_Y + WINDOW_HEIGHT * 0.13f, HELP_DESC, Color.black);
440     } else {
441       // Draw clue text
442       for (Map.Entry<Integer, String> e : clueText.entrySet()) {
443         clueFont.drawString(
444           BOARD_OFFSET_X + CLUE_OFFSET_X + cellWidth * (e.getKey() %
                size),
445           BOARD_OFFSET_Y + CLUE_OFFSET_Y + cellWidth * (e.getKey() /
                size),
446           e.getValue(), Color.darkGray);
447       }
448       // Draw guess text and note text
449       for (int i = 0; i < size; ++i) {
450         for (int j = 0; j < size; ++j) {
451           if (guessGrid.get(i * size + j) > 0) {
452             guessFont.drawString(BOARD_OFFSET_X + j * cellWidth
453               + guess_offset_x,
454               BOARD_OFFSET_Y + i * cellWidth + guess_offset_y,
455               Integer.toString(guessGrid.get(i * size + j)),
                    Color.black);
456           } else {
457             for (int k = 0; k < size; ++k) {
458               if (noteGrid.get(i * size + j).get(k)) {
459                 noteFont.drawString(BOARD_OFFSET_X + j * cellWidth
460                   + NOTE_OFFSET_X + 12 * (k % 3), BOARD_OFFSET_Y + i
461                   * cellWidth + NOTE_OFFSET_Y + 10 * (2 - k / 3),
462                   Integer.toString(k + 1), Color.blue);
463               }
464             }
465           }
466         }
467       }
468     }
469
470     // Call this last, after rendering fonts
471     GL11.glDisable(GL11.GL_TEXTURE_2D);
472   }
473
474   /**
475    * Detect user input from keyboard and mouse.
476    */
477   private void pollInput() {
478     // Need "+ cellWidth ... - 1" to make -0.5 round to -1 instead of 0
479     hoverCellX = (Mouse.getX() - BOARD_OFFSET_X + cellWidth) /
          cellWidth - 1;
480     hoverCellY =
481       (WINDOW_HEIGHT - Mouse.getY() - BOARD_OFFSET_Y + cellWidth) /
            cellWidth
482         - 1;
483
484     // Draw only if mouse is over board
```

```
485     while (Keyboard.next()) {
486       // Discard keydown events
487       if (Keyboard.getEventKeyState()) {
488         continue;
489       }
490       isUndo = false;
491       switch (Keyboard.getEventKey()) {
492         case Keyboard.KEY_ESCAPE:
493           running = false;
494           break;
495         case Keyboard.KEY_1:
496         case Keyboard.KEY_NUMPAD1:
497           type(1);
498           break;
499         case Keyboard.KEY_2:
500         case Keyboard.KEY_NUMPAD2:
501           type(2);
502           break;
503         case Keyboard.KEY_3:
504         case Keyboard.KEY_NUMPAD3:
505           type(3);
506           break;
507         case Keyboard.KEY_4:
508         case Keyboard.KEY_NUMPAD4:
509           type(4);
510           break;
511         case Keyboard.KEY_5:
512         case Keyboard.KEY_NUMPAD5:
513           type(5);
514           break;
515         case Keyboard.KEY_6:
516         case Keyboard.KEY_NUMPAD6:
517           type(6);
518           break;
519         case Keyboard.KEY_7:
520         case Keyboard.KEY_NUMPAD7:
521           type(7);
522           break;
523         case Keyboard.KEY_8:
524         case Keyboard.KEY_NUMPAD8:
525           type(8);
526           break;
527         case Keyboard.KEY_9:
528         case Keyboard.KEY_NUMPAD9:
529           type(9);
530           break;
531         case Keyboard.KEY_F1:
532           showHelp = !showHelp;
533           break;
534         case Keyboard.KEY_F2:
535           showHelp = false;
536           reset();
537           break;
538         case Keyboard.KEY_F3:
```

```
539             showHelp = false;
540             setNewProblem(3);
541             break;
542           case Keyboard.KEY_F4:
543             showHelp = false;
544             setNewProblem(4);
545             break;
546           case Keyboard.KEY_F5:
547             showHelp = false;
548             setNewProblem(5);
549             break;
550           case Keyboard.KEY_F6:
551             showHelp = false;
552             setNewProblem(6);
553             break;
554           case Keyboard.KEY_F7:
555             showHelp = false;
556             setNewProblem(7);
557             break;
558           case Keyboard.KEY_F8:
559             showHelp = false;
560             setNewProblem(8);
561             break;
562           case Keyboard.KEY_F9:
563             showHelp = false;
564             setNewProblem(9);
565             break;
566           case Keyboard.KEY_F10:
567             showHelp = false;
568             BruteForceSolver bf = new BruteForceSolver(this, problem);
569             bf.startTimer();
570             bf.solve();
571             bf.stopTimer();
572             bf.printElapsedTime();
573             Display.setTitle("KenKen - Brute Force Solver took "
574               + String.format("%.3f", bf.getElapsedTime() * 0.000000001)
575               + " seconds");
576             break;
577           case Keyboard.KEY_F11:
578             showHelp = false;
579             DepthFirstSolver dfs = new DepthFirstSolver(this, problem);
580             dfs.startTimer();
581             dfs.solve();
582             dfs.stopTimer();
583             dfs.printElapsedTime();
584             Display.setTitle("KenKen - DFS Solver took "
585               + String.format("%.3f", dfs.getElapsedTime() * 0.000000001)
586               + " seconds");
587             break;
588           case Keyboard.KEY_F12:
589             modEnabled = !modEnabled;
590             setNewProblem(size);
591             break;
592           case Keyboard.KEY_BACK:
```

```
593           isUndo = true;
594           if (toggleHistory.size() > 0) {
595             inGuessMode = toggleHistory.pop();
596             hoverCellX = hoverXHistory.pop();
597             hoverCellY = hoverYHistory.pop();
598             markCell(numHistory.pop());
599           }
600           break;
601         default:
602           inGuessMode = !inGuessMode;
603           break;
604       }
605     }
606   }
607
608   private void markCell(int n) {
609     boolean isRemoval;
610     if (boardHovered()) {
611       if (inGuessMode) {
612         // Mark guess
613         if (guessGrid.get(hoverCellY * size + hoverCellX) == n) {
614           guessGrid.put(hoverCellY * size + hoverCellX, -1);
615           isRemoval = true;
616         } else {
617           if (!isUndo && guessGrid.get(hoverCellY * size + hoverCellX)
                  > 0) {
618             boolean tmp1;
619             int tmp2;
620
621             tmp1 = toggleHistory.pop();
622             toggleHistory.push(inGuessMode);
623             toggleHistory.push(tmp1);
624
625             tmp2 = numHistory.pop();
626             numHistory.push(guessGrid.get(hoverCellY * size +
                    hoverCellX));
627             numHistory.push(tmp2);
628
629             tmp2 = hoverXHistory.pop();
630             hoverXHistory.push(hoverCellX);
631             hoverXHistory.push(tmp2);
632
633             tmp2 = hoverYHistory.pop();
634             hoverYHistory.push(hoverCellY);
635             hoverYHistory.push(tmp2);
636           }
637
638           guessGrid.put(hoverCellY * size + hoverCellX, n);
639
640           // Return if board contains solution
641           boolean boardComplete = true;
642           int guess;
643           HashSet<Integer> guessSet;
644           for (int i = 0; i < size * size; ++i) {
```

```
645             guess = guessGrid.get(i);
646             if (guess < 1) {
647               boardComplete = false;
648               break;
649             }
650             guessSet = new HashSet<Integer>();
651             guessSet.add(guess);
652             attempt.put(i, guessSet);
653           }
654           if (boardComplete && problem.checkGrid(attempt)) {
655             Display.setTitle("KenKen - Player solved in "
656               + String.format("%.3f",
657                 (System.nanoTime() - startTime) * 0.000000001) + "
                    seconds!");
658             return;
659           }
660
661           isRemoval = false;
662         }
663       } else {
664         // Mark note
665         // TODO Decide what to do with this.. nice feature but breaks
              history
666         if (!isUndo && guessGrid.get(hoverCellY * size + hoverCellX) >
              0) {
667           boolean tmp1;
668           int tmp2;
669
670           tmp1 = toggleHistory.pop();
671           toggleHistory.push(true);
672           toggleHistory.push(tmp1);
673
674           tmp2 = numHistory.pop();
675           numHistory.push(guessGrid.get(hoverCellY * size +
                hoverCellX));
676           numHistory.push(tmp2);
677
678           tmp2 = hoverXHistory.pop();
679           hoverXHistory.push(hoverCellX);
680           hoverXHistory.push(tmp2);
681
682           tmp2 = hoverYHistory.pop();
683           hoverYHistory.push(hoverCellY);
684           hoverYHistory.push(tmp2);
685         }
686         guessGrid.put(hoverCellY * size + hoverCellX, -1);
687         if (noteGrid.get(hoverCellY * size + hoverCellX).get(n - 1)) {
688           noteGrid.get(hoverCellY * size + hoverCellX).set(n - 1,
                false);
689           isRemoval = false;
690         } else {
691           noteGrid.get(hoverCellY * size + hoverCellX).set(n - 1,
                true);
692           isRemoval = true;
```

```
693           }
694         }
695         // Verify row
696         ArrayList<Integer> currRow = new ArrayList<Integer>();
697         for (int i = 0; i < size; ++i) {
698           currRow.add(guessGrid.get(hoverCellY * size + i));
699         }
700         for (int i = 0; i < size; ++i) {
701           if (currRow.get(i) < 0) {
702             incorrectGrid.put(hoverCellY * size + i, false);
703           } else {
704             if (currRow.lastIndexOf(Integer.valueOf(currRow.get(i))) !=
                  i) {
705               incorrectGrid.put(hoverCellY * size + i, true);
706               incorrectGrid.put(
707                 hoverCellY * size
708                   +
                      currRow.lastIndexOf(Integer.valueOf(currRow.get(i))),
                      true);
709             }
710             if (Collections.frequency(currRow, currRow.get(i)) < 2
711               && incorrectGrid.get(hoverCellY * size + i) == true) {
712               incorrectGrid.put(hoverCellY * size + i, false);
713             }
714           }
715         }
716
717         // Verify column
718         ArrayList<Integer> currCol = new ArrayList<Integer>();
719         for (int i = 0; i < size; ++i) {
720           currCol.add(guessGrid.get(i * size + hoverCellX));
721         }
722         for (int i = 0; i < size; ++i) {
723           if (currCol.get(i) < 0) {
724             incorrectGrid.put(i * size + hoverCellX, false);
725           } else {
726             if (currCol.lastIndexOf(Integer.valueOf(currCol.get(i))) !=
                  i) {
727               incorrectGrid.put(i * size + hoverCellX, true);
728               incorrectGrid.put(
729                 currCol.lastIndexOf(Integer.valueOf(currCol.get(i))) *
                     size
730                   + hoverCellX, true);
731
732             }
733             if (Collections.frequency(currCol, currCol.get(i)) < 2
734               && Collections.frequency(currRow, currCol.get(i)) < 2
735               && incorrectGrid.get(i * size + hoverCellX) == true) {
736               incorrectGrid.put(i * size + hoverCellX, false);
737             }
738           }
739         }
740
741         // Yes, recheck ALL the rows again
```

```
742         ArrayList<Boolean> modifiedCols =
743           new ArrayList<Boolean>(Collections.nCopies(size, false));
744         for (int j = 0; j < size; ++j) {
745           ArrayList<Integer> row = new ArrayList<Integer>();
746           for (int m = 0; m < size; ++m) {
747             row.add(guessGrid.get(j * size + m));
748           }
749           for (int k = 0; k < size; ++k) {
750             if (row.get(k) < 0) {
751               incorrectGrid.put(j * size + k, false);
752               modifiedCols.set(k, true);
753             } else {
754               if (row.lastIndexOf(Integer.valueOf(row.get(k))) != k) {
755                 incorrectGrid.put(j * size + k, true);
756                 incorrectGrid.put(
757                   j * size +
                        row.lastIndexOf(Integer.valueOf(row.get(k))), true);
758               }
759             }
760           }
761         }
762
763         // verify all changed columns
764         for (int i = 0; i < size; ++i) {
765           if (modifiedCols.get(i)) {
766             ArrayList<Integer> col = new ArrayList<Integer>();
767             for (int j = 0; j < size; ++j) {
768               col.add(guessGrid.get(j * size + i));
769             }
770             for (int k = 0; k < size; ++k) {
771
772               if (col.get(k) < 0) {
773                 incorrectGrid.put(k * size + i, false);
774               } else {
775                 if (col.lastIndexOf(Integer.valueOf(col.get(k))) != k) {
776                   incorrectGrid.put(k * size + i, true);
777
778                   incorrectGrid.put(col.lastIndexOf(Integer.valueOf(col.get(k)))
779                     * size + i, true);
780                 }
781               }
782             }
783           }
784         }
785
786         // verify cell of user input once more
787         if (isRemoval) {
788           incorrectGrid.put(hoverCellY * size + hoverCellX, false);
789         }
790
791         // Deal with cages
792         Cage cageToCheck = cellCages.get(hoverCellY * size + hoverCellX);
793         if (guessGrid.get(hoverCellY * size + hoverCellX) > -1) {
794           if (cageToCheck.isFilled(size, guessGrid)
```

```
795           && !cageToCheck.isSatisfied(size, guessGrid)) {
796           for (Integer i : cageToCheck.getCells()) {
797             incorrectCellCages.get(i / size).set(i % size, true);
798
799           }
800         } else if (cageToCheck.isFilled(size, guessGrid)
801           && cageToCheck.isSatisfied(size, guessGrid)) {
802           for (Integer i : cageToCheck.getCells()) {
803             incorrectCellCages.get(i / size).set(i % size, false);
804           }
805         }
806       } else {
807         for (Integer i : cageToCheck.getCells()) {
808           incorrectCellCages.get(i / size).set(i % size, false);
809         }
810       }
811     }
812   }
813
814   private boolean boardHovered() {
815     return hoverCellX >= 0 && hoverCellX < size && hoverCellY >= 0
816       && hoverCellY < size;
817   }
818
819   private void type(int n) {
820     if (n <= size) {
821       numHistory.push(n);
822       hoverXHistory.push(hoverCellX);
823       hoverYHistory.push(hoverCellY);
824       toggleHistory.push(inGuessMode);
825       markCell(n);
826     } else {
827       inGuessMode = !inGuessMode;
828     }
829   }
830
831   public void showProgress(HashMap<Integer, HashSet<Integer>> state) {
832     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
833     Display.sync(60);
834     for (int i = 0; i < size * size; ++i) {
835       guessGrid.put(i, (state.get(i).size() == 1) ?
             state.get(i).iterator()
836         .next() : -1);
837     }
838     renderFrame();
839     Display.update();
840   }
841
842   /**
843    * Tear down the window
844    */
845   public void destroy() {
846     Display.destroy();
847   }
```

```
848 }
```

src/GUI.java

```
1  package edu.virginia.kenken;
2
3  import java.util.ArrayList;
4  import java.util.Collections;
5  import java.util.HashMap;
6  import java.util.HashSet;
7  import java.util.Random;
8
9  public class Problem {
10
11   private final int size;
12   private final HashMap<Integer, Integer> grid;
13   private final HashMap<Integer, Integer> solution;
14   private int numCages;
15   private ArrayList<Cage> cages;
16   private final ArrayList<Cage> cellCages;
17   private final Random rand;
18
19   public Problem(int size, boolean modEnabled) {
20     this.size = size;
21     grid = new HashMap<Integer, Integer>();
22     numCages = 0;
23     cages = new ArrayList<Cage>();
24     rand = new Random();
25     cellCages =
26       new ArrayList<Cage>(Collections.nCopies(size * size, new
           Cage()));
27     ArrayList<ArrayList<Integer>> solutionArray =
28       new ArrayList<ArrayList<Integer>>();
29
30     // Start with a legal, non-random board
31
32     for (int i = 0; i < size; ++i) {
33       solutionArray.add(new ArrayList<Integer>());
34       for (int j = 0; j < size; ++j) {
35         solutionArray.get(i).add((i + j) % size + 1);
36       }
37     }
38
39     // Shuffle rows
40
41     Collections.shuffle(solutionArray);
42
43     // Transpose board matrix
44
45     int tmp;
46     for (int i = 0; i < size; ++i) {
47       for (int j = 0; j < i; ++j) {
```

```
48          tmp = solutionArray.get(i).get(j);
49          solutionArray.get(i).set(j, solutionArray.get(j).get(i));
50          solutionArray.get(j).set(i, tmp);
51        }
52      }
53
54      // Shuffle rows (which were the columns before transposition) again
55
56      Collections.shuffle(solutionArray);
57
58      // Print matrix (for testing only)
59
60      System.out.println("Generated solution:");
61      for (int i = 0; i < size; ++i) {
62        for (int j = 0; j < size; ++j) {
63          System.out.print(solutionArray.get(i).get(j));
64        }
65        System.out.print("\n");
66      }
67      System.out.println("");
68
69      // Copy temporary solution arrays into hashmap
70      solution = new HashMap<Integer, Integer>();
71      for (int i = 0; i < size; ++i) {
72        for (int j = 0; j < size; ++j) {
73          solution.put(i * size + j, solutionArray.get(i).get(j));
74        }
75      }
76
77      // Initialize cageIDs
78
79      for (int i = 0; i < size * size; ++i) {
80        grid.put(i, -1);
81      }
82
83      ArrayList<String> directions = new ArrayList<String>();
84      directions.add("N");
85      directions.add("E");
86      directions.add("S");
87      directions.add("W");
88
89      int curID = 0;
90      int curX = -1;
91      int curY = -1;
92      int nextX = -1;
93      int nextY = -1;
94
95      int cageSize;
96      int maxCageSize = -1;
97      float cageCutoff;
98      float opCutoff;
99
100     boolean boardFull;
101     boolean growable;
```

```
102
103     // TODONE Remove all references to sizeDistribution (it's just for
            testing)
104     // ArrayList<Integer> sizeDistribution = new ArrayList<Integer>();
105     // sizeDistribution.add(0);
106     // sizeDistribution.add(0);
107     // sizeDistribution.add(0);
108     // sizeDistribution.add(0);
109
110     cages = new ArrayList<Cage>();
111     Cage cage;
112
113     // ArrayList used to keep track of which cells belong to the
            current cage
114     ArrayList<Integer> cageCells = new ArrayList<Integer>();
115
116     // Each iteration generates a new cage
117     while (true) {
118       cageCells.clear();
119       // Select first available uncaged cell to be "root node" of new
            cage
120       boardFull = true;
121       for (int i = 0; i < size; ++i) {
122         for (int j = 0; j < size; ++j) {
123           if (grid.get(i * size + j) < 0) {
124             curX = j;
125             curY = i;
126             boardFull = false;
127             break;
128           }
129         }
130         if (!boardFull) {
131           break;
132         }
133       }
134
135       // ...Unless all cells are caged already; then quit
136       if (boardFull) {
137         break;
138       }
139
140       // Predetermine the maximum number of cells this cage will
            contain,
141       // assuming nothing gets in the way of its growth
142       cageCutoff = rand.nextFloat();
143       if (cageCutoff < 0.07) {
144         maxCageSize = 1;
145       } else if (cageCutoff < 0.55) {
146         maxCageSize = 2;
147       } else if (cageCutoff < 0.9) {
148         maxCageSize = 3;
149       } else {
150         maxCageSize = 4;
151       }
```

```
152
153         // Add current cell to new cage
154         cage = new Cage();
155
156         // Add method is used for positioning of the cells based on ID.
                Do not
157         // change!
158         cage.add(curY * size + curX);
159         cageCells.add(curY * size + curX);
160         cage.addPosition(curY, curX);
161         cage.addElement(solution.get(curY * size + curX));
162         grid.put(curY * size + curX, curID);
163         cageSize = 1;
164
165         // Grow cage, cell by cell
166         while (true) {
167           // Stop when maximum cage size is reached
168           if (cageSize >= maxCageSize) {
169             break;
170           }
171
172           growable = false;
173
174           // Randomly choose growth direction
175           Collections.shuffle(directions);
176           for (String s : directions) {
177             switch (s) {
178               case "N":
179                 nextX = curX;
180                 nextY = curY - 1;
181                 break;
182               case "E":
183                 nextX = curX + 1;
184                 nextY = curY;
185                 break;
186               case "S":
187                 nextX = curX;
188                 nextY = curY + 1;
189                 break;
190               case "W":
191                 nextX = curX - 1;
192                 nextY = curY;
193                 break;
194             }
195             if (nextX >= 0 && nextX < size && nextY >= 0 && nextY <
                    size) {
196               if (grid.get(nextY * size + nextX) == -1) {
197                 growable = true;
198                 break;
199               }
200             }
201           }
202
203           // If next cell is valid, add it to cage and move to it
```

```
204         if (growable && cageSize < maxCageSize) {
205           cage.add(nextY * size + nextX);
206           cageCells.add(nextY * size + nextX);
207           cage.addPosition(nextY, nextX);
208           cage.addElement(solution.get(nextY * size + nextX));
209           grid.put(nextY * size + nextX, curID);
210           curX = nextX;
211           curY = nextY;
212           cageSize += 1;
213         } else {
214           break;
215         }
216       }
217
218       // Assign operator to cage
219       Cage operationCage;
220       switch (cage.getCells().size()) {
221         case 1:
222           operationCage = new UnitCage(cage);
223           break;
224         case 2:
225           opCutoff = rand.nextFloat();
226           if (opCutoff < 0.1) {
227             operationCage = new AdditionCage(cage);
228           } else if (opCutoff < 0.2) {
229             operationCage = new MultiplicationCage(cage);
230           } else {
231             if (modEnabled) {
232               if (opCutoff < 0.5) {
233                 operationCage = new SubtractionCage(cage);
234               } else {
235                 int smaller = cage.getCellElements().get(0);
236                 int larger = cage.getCellElements().get(1);
237                 if (larger < smaller) {
238                   int temp = smaller;
239                   smaller = larger;
240                   larger = temp;
241                 }
242                 if (larger % smaller == 0 && opCutoff < 0.95) {
243                   operationCage = new DivisionCage(cage);
244                 } else {
245                   operationCage = new ModuloCage(cage);
246                 }
247               }
248             } else {
249               int smaller = cage.getCellElements().get(0);
250               int larger = cage.getCellElements().get(1);
251               if (larger < smaller) {
252                 int temp = smaller;
253                 smaller = larger;
254                 larger = temp;
255               }
256               if (larger % smaller == 0 && opCutoff < 0.95) {
257                 operationCage = new DivisionCage(cage);
```

```
258                 } else {
259                   operationCage = new SubtractionCage(cage);
260                 }
261               }
262             }
263             break;
264           default:
265             operationCage =
266               (rand.nextBoolean() ? new MultiplicationCage(cage)
267                 : new AdditionCage(cage));
268             break;
269       }
270       cages.add(operationCage);
271
272       // Assign each cell, referenced by ID, to the appropriate cage
273       for (Integer i : cageCells) {
274         cellCages.set(i, operationCage);
275       }
276
277       // sizeDistribution
278       // .set(cageSize - 1, sizeDistribution.get(cageSize - 1) + 1);
279       curID += 1;
280     }
281
282     numCages = curID + 1;
283
284     // System.out.println("Number of cages: " + numCages);
285     // System.out.println("Cage size distribution: " +
        sizeDistribution);
286   }
287
288   public int getSize() {
289     return size;
290   }
291
292   public HashMap<Integer, Integer> getGrid() {
293     return grid;
294   }
295
296   public int getNumCages() {
297     return numCages;
298   }
299
300   public ArrayList<Cage> getCages() {
301     return cages;
302   }
303
304   public boolean checkGrid(HashMap<Integer, HashSet<Integer>> attempt)
        {
305     // TODO Ensure rows and columns are also valid
306     for (Cage c : cages) {
307       if (!c.isSatisfiedHashMapVersion(attempt, size)) {
308         return false;
309       }
```

```
310       }
311
312       boolean generatedSolutionFound = true;
313       for (int i = 0; i < size; ++i) {
314         for (int j = 0; j < size; ++j) {
315           if (attempt.get(i * size + j).iterator().next() !=
                solution.get(i
316             * size + j)) {
317             generatedSolutionFound = false;
318             break;
319           }
320         }
321         if (!generatedSolutionFound) {
322           break;
323         }
324       }
325
326       if (generatedSolutionFound) {
327         System.out.println("Generated solution found!");
328       } else {
329         System.out.println("Different solution found!");
330       }
331       return true;
332     }
333
334     // Method to check for valid row and columns
335     public boolean checkRowAndColumn(ArrayList<ArrayList<Integer>>
          attempt) {
336       // Create HashSet; when adding duplicates, the add method will
             return false
337       HashSet<Integer> test = new HashSet<Integer>();
338
339       // First check rows
340       for (int i = 0; i < size; ++i) {
341         test.clear();
342         for (int j = 0; j < size; ++j) {
343           if (!test.add(attempt.get(i).get(j))) {
344             return false;
345           }
346         }
347       }
348
349       // Then check columns
350       for (int i = 0; i < size; ++i) {
351         test.clear();
352         for (int j = 0; j < size; ++j) {
353           if (!test.add(attempt.get(j).get(i))) {
354             return false;
355           }
356         }
357       }
358       return true;
359     }
360
```

```
361   public ArrayList<Cage> getCellCages() {
362     return cellCages;
363   }
364 
365 }
```

src/Problem.java

```
1  package edu.virginia.kenken;
2  
3  public abstract class Solver {
4    private final GUI gui;
5    private final Problem problem;
6    private long startTime;
7    private long endTime;
8    private long elapsedTime;
9  
10   public Solver(GUI gui, Problem problem) {
11     this.gui = gui;
12     this.problem = problem;
13   }
14 
15   public GUI getGUI() {
16     return gui;
17   }
18 
19   public Problem getProblem() {
20     return problem;
21   }
22 
23   public void startTimer() {
24     startTime = System.nanoTime();
25   }
26 
27   public void stopTimer() {
28     endTime = System.nanoTime();
29     elapsedTime = endTime - startTime;
30   }
31 
32   public void printElapsedTime() {
33     System.out.println("Elapsed time: " + elapsedTime * 0.000000001
34       + " seconds");
35   }
36 
37   public long getElapsedTime() {
38     return elapsedTime;
39   }
40 
41 }
```

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
```

```
54        if (!nextPermutation(rowPermutation)) {
55          rowPermutation = new ArrayList<Integer>();
56          for (int k = 0; k < size; ++k) {
57            rowPermutation.add(k + 1);
58          }
59          nextPermutation(colPermutation);
60        }
61
62        // Reassign attempt grid values as specified by permutations
63        for (int i = 0; i < size; ++i) {
64          for (int j = 0; j < size; ++j) {
65            attempt.put(
66              i * size + j,
67              template.get((colPermutation.get(i) - 1) * size
68                + rowPermutation.get(j) - 1));
69          }
70        }
71      }
72
73      solution = attempt;
74      getGUI().showProgress(solution);
75    }
76
77    // public long getStatesChecked() {
78    // return statesChecked;
79    // }
80
81    // public HashMap<Integer, HashSet<Integer>> getSolution() {
82    // return solution;
83    // }
84
85    // public void printSolution() {
86    // for (int i = 0; i < size; ++i) {
87    // System.out.println(solution.get(i));
88    // }
89    // }
90
91    /**
92     * @param p
93     *          Input list
94     * @return Whether input is not the last permutation
95     */
96    private static boolean nextPermutation(ArrayList<Integer> p) {
97      int a = p.size() - 2;
98      while (a >= 0 && p.get(a) >= p.get(a + 1)) {
99        a--;
100     }
101     if (a < 0) {
102       return false;
103     }
104
105     int b = p.size() - 1;
106     while (p.get(b) <= p.get(a)) {
107       b--;
```

```
108       }
109
110       int t = p.get(a);
111       p.set(a, p.get(b));
112       p.set(b, t);
113
114       for (int i = a + 1, j = p.size() - 1; i < j; ++i, --j) {
115         t = p.get(i);
116         p.set(i, p.get(j));
117         p.set(j, t);
118       }
119       return true;
120     }
121   }
```

src/BruteForceSolver.java

```
1   package edu.virginia.kenken;
2
3   import java.util.ArrayList;
4   import java.util.HashMap;
5   import java.util.HashSet;
6
7   public class DepthFirstSolver extends Solver {
8     private final int size;
9     private final ArrayList<Cage> cages;
10    private boolean solutionFound;
11    private HashMap<Integer, HashSet<Integer>> solution;
12    private int statesChecked;
13    private HashMap<Integer, Integer> gainScores;
14
15    public DepthFirstSolver(GUI gui, Problem problem) {
16      super(gui, problem);
17
18      size = problem.getSize();
19      cages = problem.getCages();
20      solutionFound = false;
21      statesChecked = 0;
22    }
23
24    public void solve() {
25
26      // Initialize grid of guesses to all empty
27      HashMap<Integer, HashSet<Integer>> root =
28        new HashMap<Integer, HashSet<Integer>>();
29      for (int i = 0; i < size * size; ++i) {
30        root.put(i, new HashSet<Integer>());
31        // TODO Make this iterate upwards (currently set to iterate
              downwards to
32        // improve naive information gain, since large cell guesses
              typically fail
33        // faster)
```

```
34        for (int j = size; j > 0; --j) {
35          root.get(i).add(j);
36        }
37      }
38      // Get easy stuff done first - mark all UnitCages and recurse
           through
39      // their peers, marking them if possible too
40      for (Cage c : cages) {
41        c.preprocess(size, root);
42        if (c.getCells().size() == 1) {
43          trimPeers(c.getCells().get(0), c.getTotal(), root);
44        }
45      }
46
47      // Assign expected information gain scores to cells
48      gainScores = new HashMap<Integer, Integer>();
49      int operationScore = -1;
50      for (Cage c : cages) {
51        switch (c.getClass().getSimpleName()) {
52          case "AdditionCage":
53            operationScore = 35;
54            break;
55          case "DivisionCage":
56            operationScore = 50;
57            break;
58          case "ModuloCage":
59            operationScore = 35;
60            break;
61          case "MultiplicationCage":
62            operationScore = 50;
63            break;
64          case "SubtractionCage":
65            operationScore = 35;
66            break;
67          case "UnitCage":
68            operationScore = -1;
69            break;
70          default:
71            System.out.println("Wtf");
72            break;
73        }
74
75        if (operationScore < 0) {
76          continue;
77        }
78
79        for (Integer cellID : c.getCells()) {
80          gainScores.put(cellID,
81            (int) (operationScore - 12 * Math.pow(1.5, c.getNumCells() -
                 1)));
82        }
83      }
84
85      // Call the root instance of DFS on the cell with highest info gain
```

```
86      DFS(maxGain(root), root);
87
88      if (solution == null) {
89        System.out.println("No solution found.");
90      } else {
91        // Update display with current state
92        getGUI().showProgress(solution);
93
94        // HashMap<Integer, Integer> matrix = new HashMap<Integer,
              Integer>();
95        // for (int i = 0; i < size; ++i) {
96        // for (int j = 0; j < size; ++j) {
97        // matrix.put(i * size + j, (solution.get(i * size + j).size()
              == 1)
98        // ? solution.get(i * size + j).iterator().next() : -1);
99        // }
100       // }
101       // getProblem().checkGrid(matrix);
102       getProblem().checkGrid(solution);
103       System.out.println("States checked: " + statesChecked);
104     }
105   }
106
107   /**
108    * Recursively called DFS algorithm - should be called only on
          undetermined
109    * cells.
110    *
111    * @param cellID
112    * @param state
113    */
114   private void DFS(int cellID, HashMap<Integer, HashSet<Integer>>
          state) {
115     // Check whether this is a solution
116     if (solutionFound) {
117       return;
118     }
119
120     // Loop through possible values for this cell
121     int markedInCage;
122     boolean cagesSatisfied;
123     HashMap<Integer, HashSet<Integer>> child;
124
125     for (Integer v : state.get(cellID)) {
126       // Quit if this branch's left sibling found a solution
127       if (solutionFound) {
128         return;
129       }
130
131       statesChecked += 1;
132       if (statesChecked % 4096 == 0) {
133         // Update display with current state
134         getGUI().showProgress(state);
135       }
```

```
136
137         // Copy parent state into a new child state
138         child = cloneState(state);
139
140         // Mark cell with DFS hypothesis
141         child.get(cellID).clear();
142         child.get(cellID).add(v);
143
144         // Trim peers
145         trimPeers(cellID, v, child);
146
147         // Check for cage conflicts (note that we don't need to check for
148         // row/column conflicts since we previously called
                makeAndTrimPeers on the
149         // HashSet we're iterating through)
150         cagesSatisfied = true;
151         for (Cage c : cages) {
152           if (!cagesSatisfied) {
153             break;
154           }
155
156           // Check this cage
157           markedInCage = 0;
158           for (Integer i : c.getCells()) {
159             if (child.get(i).size() < 1) {
160               // This might occur if a wrong solution is given to
                      trimPeers
161               cagesSatisfied = false;
162               break;
163             }
164             if (child.get(i).size() == 1) {
165               markedInCage += 1;
166             }
167           }
168           if (cagesSatisfied && markedInCage == c.getNumCells()) {
169             if (!c.isSatisfiedHashMapVersion(child, size)) {
170               cagesSatisfied = false;
171               break;
172             }
173           }
174         }
175         if (!cagesSatisfied) {
176           continue;
177         }
178
179         // Check whether child is solution
180         if (isSolution(child)) {
181           solution = child;
182           solutionFound = true;
183           return;
184         }
185
186         // Recursively call DFS
187         DFS(maxGain(child), child);
```

```
188       }
189     }
190
191     /**
192      * Mark the given cell, remove its value from its peers' sets of
             possible
193      * values, and recursively continue marking peers whose sizes of
             sets of
194      * possible values become 1.
195      *
196      * @param cellID
197      *          Cell to mark
198      * @param value
199      *          Value to mark
200      * @param state
201      *          Current state
202      */
203     private void trimPeers(int cellID, int value,
204       HashMap<Integer, HashSet<Integer>> state) {
205       int row = cellID / size;
206       int col = cellID % size;
207       int peerID;
208
209       // Trim this cell's designated value from its peer cells
210       // TODO Factor out the common loop bodies
211       for (int i = 0; i < size; ++i) {
212         peerID = row * size + i;
213         if (peerID != cellID) {
214           if (state.get(peerID).remove(value)) {
215             // Peer newly became determined, so trim *its* peers
216             if (state.get(peerID).size() == 1) {
217               trimPeers(peerID, state.get(peerID).iterator().next(),
                       state);
218             }
219           }
220         }
221
222         peerID = size * i + col;
223         if (peerID != cellID) {
224           if (state.get(peerID).remove(value)) {
225             // Peer newly became determined, so trim *its* peers
226             if (state.get(peerID).size() == 1) {
227               trimPeers(peerID, state.get(peerID).iterator().next(),
                       state);
228             }
229           }
230         }
231       }
232     }
233
234     /**
235      * Check whether all cells in the state have 1 possible value.
236      *
237      * @param state
```

```
238     * @return Whether state is a solution
239     */
240   private boolean isSolution(HashMap<Integer, HashSet<Integer>> state)
          {
241     boolean allCellsMarked = true;
242     for (HashSet<Integer> s : state.values()) {
243       if (s.size() > 1) {
244         allCellsMarked = false;
245         break;
246       }
247     }
248     return allCellsMarked;
249   }
250
251   private HashMap<Integer, HashSet<Integer>> cloneState(
252     HashMap<Integer, HashSet<Integer>> state) {
253     HashMap<Integer, HashSet<Integer>> clone =
254       new HashMap<Integer, HashSet<Integer>>();
255     HashSet<Integer> possibleValues;
256     for (Integer i : state.keySet()) {
257       possibleValues = new HashSet<Integer>();
258       for (Integer j : state.get(i)) {
259         possibleValues.add(j);
260       }
261       clone.put(i, possibleValues);
262     }
263     return clone;
264   }
265
266   private int maxGain(HashMap<Integer, HashSet<Integer>> state) {
267     // for (int i = 0; i < size * size; ++i) {
268     // if (state.get(i).size() > 1) {
269     // return i;
270     // }
271     // }
272     // return -1;
273     int maxGain = -1;
274     int cellID = -1;
275     int gain;
276     for (int i = 0; i < size * size; ++i) {
277       if (state.get(i).size() > 1) {
278         gain = gainScores.get(i) + 700 * (size - state.get(i).size())
              / size;
279         if (gain > maxGain) {
280           maxGain = gain;
281           cellID = i;
282         }
283       }
284     }
285     return cellID;
286   }
287 }
```

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;
```

```
52    }
53
54    public int getNumCells() {
55      return cells.size();
56    }
57
58  }
```

src/Constraint.java

```
1   package edu.virginia.kenken;
2
3   import java.util.HashMap;
4   import java.util.HashSet;
5
6   public class Cage extends Constraint {
7     private int total;
8
9     public Cage() {
10      super();
11    }
12
13    public Cage(Cage src) {
14      super();
15      setCells(src.getCells());
16      setCellPositions(src.getCellPositions());
17    }
18
19    public String getClueText() {
20      return Integer.toString(total);
21    }
22
23    public int getTotal() {
24      return total;
25    }
26
27    public void setTotal(int total) {
28      this.total = total;
29    }
30
31    public void preprocess(int size, HashMap<Integer, HashSet<Integer>>
        state) {
32      return;
33    }
34
35    public boolean isSatisfiedHashMapVersion(
36      HashMap<Integer, HashSet<Integer>> state, int size) {
37      System.out.println("This was supposed to be abstract.");
38      return false;
39    }
40
41    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

```
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) {
```

```
32            it.remove();
33          }
34        }
35      }
36      return;
37    }
38
39    @Override
40    public boolean isSatisfiedHashMapVersion(
41      HashMap<Integer, HashSet<Integer>> entryGrid, int size) {
42      int guessSum = 0;
43      for (int i = 0; i < getCellPositions().size(); i = i + 2) {
44        guessSum +=
45          entryGrid
46            .get(getCellPositions().get(i) * size +
                 getCellPositions().get(i + 1))
47            .iterator().next();
48      }
49      return (guessSum == getTotal());
50    }
51
52    @Override
53    public boolean isSatisfied(int size, HashMap<Integer, Integer>
        entryGrid) {
54      int guessSum = 0;
55      for (int i = 0; i < getCellPositions().size(); i = i + 2) {
56        guessSum +=
57          entryGrid.get(getCellPositions().get(i) * size
58            + getCellPositions().get(i + 1));
59      }
60      return (guessSum == getTotal());
61    }
62  }
```

src/AdditionCage.java

```
1   package edu.virginia.kenken;
2
3   import java.util.Collections;
4   import java.util.HashMap;
5   import java.util.HashSet;
6   import java.util.Iterator;
7
8   public class DivisionCage extends Cage {
9     public DivisionCage(Cage src) {
10      super(src);
11      setTotal(Collections.max(src.getCellElements())
12        / Collections.min(src.getCellElements()));
13    }
14
15    @Override
16    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

```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
```

```
51    public boolean isSatisfied(int size, HashMap<Integer, Integer>
          entryGrid) {
52      int a =
53        entryGrid.get(getCellPositions().get(0) * size
54          + getCellPositions().get(1));
55      int b =
56        entryGrid.get(getCellPositions().get(2) * size
57          + getCellPositions().get(3));
58      return (Math.max(a, b) % Math.min(a, b) == getTotal());
59    }
60
61  }
```

src/ModuloCage.java

```
1   package edu.virginia.kenken;
2
3   import java.util.HashMap;
4   import java.util.HashSet;
5   import java.util.Iterator;
6
7   public class MultiplicationCage extends Cage {
8     public MultiplicationCage(Cage src) {
9       super(src);
10      int product = 1;
11      for (Integer d : src.getCellElements()) {
12        product *= d;
13      }
14      setTotal(product);
15    }
16
17    @Override
18    public String getClueText() {
19      return getTotal() + "x";
20    }
21
22    @Override
23    public void preprocess(int size, HashMap<Integer, HashSet<Integer>>
          state) {
24      Iterator<Integer> it;
25      int value;
26      int minPossible =
27        (int) Math.ceil(getTotal() / Math.pow(size, getTotal() - 1));
28
29      for (Integer cellID : getCells()) {
30        it = state.get(cellID).iterator();
31        while (it.hasNext()) {
32          value = it.next();
33          if (getTotal() % value > 0 || value > getTotal() || value <
                minPossible) {
34            it.remove();
35          }
```

```
36        }
37      }
38      return;
39    }
40
41    @Override
42    public boolean isSatisfiedHashMapVersion(
43      HashMap<Integer, HashSet<Integer>> entryGrid, int size) {
44      int guessProduct = 1;
45      for (int i = 0; i < getCellPositions().size(); i = i + 2) {
46        guessProduct *=
47          entryGrid
48            .get(getCellPositions().get(i) * size +
                   getCellPositions().get(i + 1))
49            .iterator().next();
50      }
51      return (guessProduct == getTotal());
52    }
53
54    @Override
55    public boolean isSatisfied(int size, HashMap<Integer, Integer>
        entryGrid) {
56      int guessProduct = 1;
57      for (int i = 0; i < getCellPositions().size(); i = i + 2) {
58        guessProduct *=
59          entryGrid.get(getCellPositions().get(i) * size
60            + getCellPositions().get(i + 1));
61      }
62      return (guessProduct == getTotal());
63    }
64  }
```

src/MultiplicationCage.java

```
1   package edu.virginia.kenken;
2
3   import java.util.Collections;
4   import java.util.HashMap;
5   import java.util.HashSet;
6   import java.util.Iterator;
7
8   public class SubtractionCage extends Cage {
9     public SubtractionCage(Cage src) {
10      super(src);
11      setTotal(Collections.max(src.getCellElements())
12        - Collections.min(src.getCellElements()));
13    }
14
15    @Override
16    public String getClueText() {
17      return getTotal() + "-";
18    }
```

```
19
20    @Override
21    public void preprocess(int size, HashMap<Integer, HashSet<Integer>>
          state) {
22      Iterator<Integer> it;
23      int value;
24      for (Integer cellID : getCells()) {
25        it = state.get(cellID).iterator();
26        while (it.hasNext()) {
27          value = it.next();
28          if (value > size - getTotal() && value <= getTotal()) {
29            it.remove();
30          }
31        }
32      }
33      return;
34    }
35
36    @Override
37    public boolean isSatisfiedHashMapVersion(
38      HashMap<Integer, HashSet<Integer>> entryGrid, int size) {
39      return (Math.abs(entryGrid
40        .get(getCellPositions().get(0) * size +
              getCellPositions().get(1))
41        .iterator().next()
42        - entryGrid
43          .get(getCellPositions().get(2) * size +
                getCellPositions().get(3))
44          .iterator().next()) == getTotal());
45    }
46
47    @Override
48    public boolean isSatisfied(int size, HashMap<Integer, Integer>
          entryGrid) {
49      return (Math.abs(entryGrid.get(getCellPositions().get(0) * size
50        + getCellPositions().get(1))
51        - entryGrid.get(getCellPositions().get(2) * size
52          + getCellPositions().get(3))) == getTotal());
53    }
54  }
```

src/SubtractionCage.java

```
1  package edu.virginia.kenken;
2
3  import java.util.HashMap;
4  import java.util.HashSet;
5
6  public class UnitCage extends Cage {
7    public UnitCage(Cage src) {
8      super(src);
9      setTotal(src.getCellElements().get(0));
```

```
10    }
11
12    @Override
13    public String getClueText() {
14      return getTotal() + "";
15    }
16
17    @Override
18    public void preprocess(int size, HashMap<Integer, HashSet<Integer>>
          state) {
19      state.get(getCells().get(0)).clear();
20      state.get(getCells().get(0)).add(getTotal());
21      return;
22    }
23
24    @Override
25    public boolean isSatisfiedHashMapVersion(
26      HashMap<Integer, HashSet<Integer>> entryGrid, int size) {
27      return (entryGrid
28        .get(getCellPositions().get(0) * size +
             getCellPositions().get(1))
29        .iterator().next() == getTotal());
30    }
31
32    @Override
33    public boolean isSatisfied(int size, HashMap<Integer, Integer>
          entryGrid) {
34      return (entryGrid.get(getCellPositions().get(0) * size
35        + getCellPositions().get(1)) == getTotal());
36    }
37  }
```

src/UnitCage.java