Kakuro generator algorithm

Objective of the algorithm:

The objective is to generate a valid board for the Kakuro game, meaning that playing it by the Kakuro rules, it can be solved. Our generator aims to create boards of unique solution, but it can fail to do so, and in such case it creates a board which will have more than one solution.
In any case, we try to minimize the probability of generating a board of multiple solutions.

Basic idea behind the algorithm:

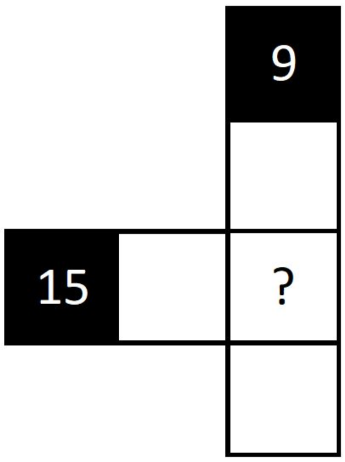
We define as an **ambiguity**, the case where a given cell of the board can take more than one different value and for each of those values the Kakuro has a valid solution.

So we need to minimize the number of ambiguities as much as possible.
If you investigate a little on Kakuro puzzles there are some known possible combinations such that for a given row size and column size and their sums you can compute what values can be assigned to the cell where this row and this column cross. We are interested in finding the combinations where it only leaves one possible value available to the cell, that way we ensure that that cell will not cause an ambiguity.

Let's look at some interesting cases:

- We define as a **starting point** the case where a cell value can be determined only with the row size, column size, row sum and column sum. This happens when there is only one value in common between all the values in combinations for the row and all the values in combinations for the column.

Example of starting point:



Combinations for Space 2, Sum 15:

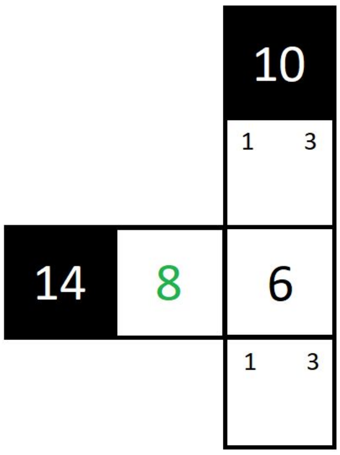
6 + 9, 7 + 8

Combinations for Space 3, Sum 9:

1 + 2 + **6**, 1 + 3 +5, 2 + 3 + 4

The only common value between the first set of combinations and the second one is the number 6, so the cell "?" must take the value 6, there is no ambiguity.

- Another interesting case is when we have some values in a row and/or column assigned, and some not assigned, we can find the possible combinations that fit into that row and/or column taking into account the assigned values. Let's see a simple example:



Combinations for Space 2, Sum 14:

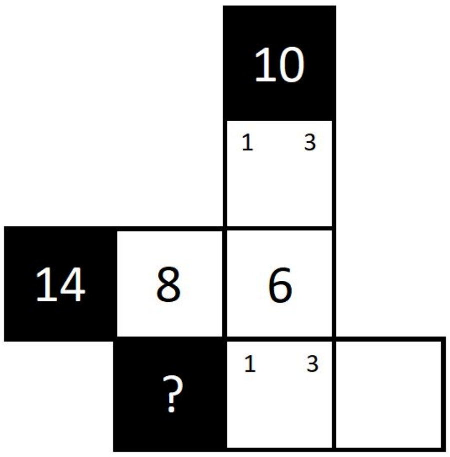
~~5 + 9~~, **6** + 8

Combinations for Space 3, Sum 10:

~~1 + 2 + 7~~, 1 + 3 + **6**, ~~1 + 4 + 5~~, ~~2 + 3 + 5~~

We see that there is only one combination left for the row so at the next step we will assign that value to the corresponding cell.

- But when we are generating a new puzzle we don't have row sums and column sums, sometimes we need to make a decision to assign a value to a certain row or column. This is what we can do:



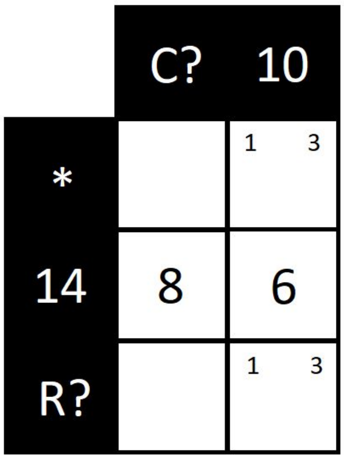
To break the ambiguity of the cells of the column (both can take value 1 or 3) we assign a sum to the row marked with the symbol "?" that given the space of 2, among all of the combinations there is at least one that uses either 1 or 3 but there isn't any combination that uses the other. For example 6:

Combinations for space 2, sum 6:

1 + 5, ~~2 + 4~~

From these two combinations only the first one is possible because the column of the cell doesn't have any possible combinations with 6 and 2 or 6 and 4. So at the next step we assign 1 to the value.

- The last case we want to mention is that we might get into a situation where neither the row nor the column have a defined sum. If there are no values assigned to cells of these row and column we can create a new starting point, but if there are values we should try to use them to find a good assignation for the row and column sums that isn't a starting point (because if we add a bunch of starting points the difficulty of the puzzle will decrease for the user).

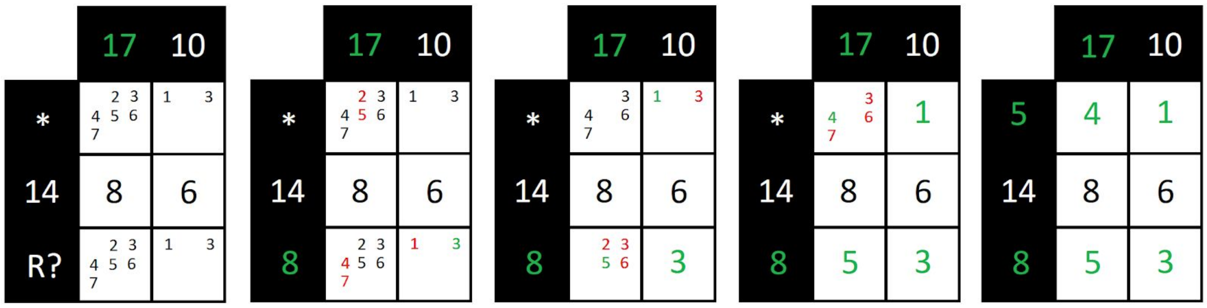


For example, if we take the value 17 for column “C?” it has the following combinations available:
~~1+7+9~~, ~~2+6+9~~, 2+7+8, ~~3+5+9~~, 3+6+8, 4+5+8, ~~4+6+7~~

Only three of them have an eight.

For the row “R?” we will take into account that it has to be a value that solves the ambiguity for the column of sum 10 as well as the new one for “C?” that no has sum value 17.

For instance we can assign 8 for the row “R?” sum. This will end up in a bunch of cells being affected, looking something like so:



Now it is clear that an assignation on a row sum, column sum or cell value can have repercussions in a lot of other cells’ possible values. For this example that we have shown we have chosen values that worked for the purposes of the explanation but if a wrong value was chosen it would maybe reach a point where a cell had no more possible values to be assigned. In that case the assignation would fail and we should be able to do a rollback to the state previous to the wrong assignation.

With all this clear we can proceed to explain the different parts of the algorithm.

Also, we have created a class that manages all the consulting operations related to searching for all the possible known cases given a space and sum, or given a space and some values, etc. This class is named **KakuroConstants** and this is the name we’ll use to refer to it from now on.

The algorithm:

The parameters: We ask for the number of rows, the number of columns and the difficulty of the puzzle.

- **Step 1:** Generating black and white cells
    - First of all we start with a board of all white cells and we set the first row and column of the board to black.

    - With the aim of creating interesting looking kakuros we've decided we will ensure the symmetry along the horizontal axis of the placement of black cells so every time we turn a cell black in the top half we will also do so in the bottom half. So we traverse the top half of the remaining white cells and (according to a certain probability defined taking into account the difficulty) we decide if we should attempt to turn it black, we consider:
        - If it isn't two squares apart, in any direction, of another black cell or of the symmetric position.
        - If turning that cell and its symmetric partner black would result in a disconnected board (two or more groups of white cells). We can know this because the only way that it would cause a disconnection is if for any one of its adjacent white cells there wasn't a path connecting it to any other of its adjacent white cells. So we do a modification of the BFS algorithm with a priority queue that visits first the cells that are closer to the one we're evaluating, this way it goes a little faster than just doing a BFS or DFS.
    - Once this traversal is complete it can happen that we have rows or columns with a size greater than nine, so we traverse the whole board again and for every row/column with size greater than nine we cut it according to the following priorities (always taking into account the modification of the symmetric cell):
        - First we look to turn black a cell that verifies the two conditions we explained earlier (doesn't produce size 1, doesn't disconnect).
        - If there's not such a cell we look for one of the remaining cells that would not disconnect the white cell connected group but that would then create a row/column of size 1.
        - If none of the cells can be converted to black without disconnecting the graph, then we find ourselves obligated to choose a random cell and create a disconnection.

        - Other mechanisms might be considered to make the process better in the final version if we have time left such as checking not only where we can place a black cell to cut the long line, but also checking, if it is a conflictive position, if we can fix it by turning neighbouring black cells back to white.

- **Step 2:** Taking into account the difficulty we randomly choose a number of white cells and for each one we ask KakuroConstants what are some possible sum assignations for the row and column of that cell that would make it a **starting point**.

At this point we have some assignations that force certain values in some cells. We have created a data-structure called **SwappingCellQueue** that acts like a priority queue that keeps track of all the white cells in the board being developed. The first element of the queue will always be the cell that has fewer possible values left, so the one that should most likely be defined next by assigning a certain sum to its row or column (remember interesting case number three).

Once we have some values defined, there are cells that have fewer possible values than other ones, so we give them priority:

- **Step 3:** Take the first white cell in our SwappingCellQueue, which will in no case have a value assigned.

  We have four possibilities:
  - Only the sum of its row is undefined.
  - Only the sum of its column is undefined.
  - Both the sum of its row and the sum of its column are undefined.
  - Both the sum of its row and the sum of its column are defined.

  - In the three first cases we have options to make the cell take a certain value (as we have seen in the examples above), so we should ask our KakuroConstants class for the possible combinations for the space of the row/column/both and the values that have already been used in the row/column/both. We will get as an answer a list of values that if we assigned to the row/column would force the cell we are considering to take a certain value.
  So we try to assign each of the possibilities to the row/column sum until one assignment is successful. (We will talk about the assigning process later, for now just know that an assignation can be successful or it can fail, as we commented earlier).
  If there were no possibilities or none of them ends in an assignation of the value of the cell (all fail, which is unlikely) then this cell could cause an **ambiguity**, any cell that we can't assign successfully might get assigned later as consequence of another assignation, but just in case we'll add it to a list of "possibly ambiguous cells" and we'll check on it later.

  - In the last case we can't force the cell to take any value by assigning a sum to its row/column because they already have defined sums. The only thing we can do is find a cell in the same row/column such that one of its possible values coincides with one of the possible values of the cell we are considering, and that this "neighbouring" cell's row or column is undefined. Then if we force that cell into taking the possible value that it has in common, our possible value will no longer be possible and we might force the cell we are considering into a value.
  If this process doesn't work the cell could cause an **ambiguity** that could potentially be solved by its own but maybe not, so we add it to our "possibly ambiguous cells" list and move on to the next one.

  We repeat Step 3 until the SwappingCellQueue is empty, meaning that we have either looked or assigned all of the cells.
  Notice that up until this point, any cell value assignment has been forced by us choosing specific sum values for the rows and columns. Now it's time to check if any of our "possibly ambiguous cells" still hasn't been assigned a value and thus the ambiguity remains.

- **Step 4:** We iterate through the possibly ambiguous cells and check if they have a value. If they do (as all assignations we have done so far have been forced) then the ambiguity was resolved. If they don't, however, we have two options: we either accept the fact that our generated board won't have a unique solution and we assign one of its possibilities to it (which will resolve recursively the cells that were sharing this ambiguity) or we could have a final board that has some cell values already pre-defined (solving all ambiguities, meaning unique solution).

Finally: what do we mean by the different value **assignation** processes

We have three different assignation scenarios that we need to manage, but they have strong similarities that let us reuse a couple of functions.

- **Row sum assignation:** It assigns a value to the specified row sum, but then the possibilities for the cells in the row are affected, which means we have to update the row possibilities.
- **Column sum assignation:** It assigns a value to the specified column sum, but then the possibilities for the cells in the column are affected, which means we have to update the column possibilities.
- **Cell value assignation:** It assigns a value to the given cell (should happen because it is forced by an assignation to a row or column). But then, the possibilities for the cells in the same row and column are affected, which means we have to update the row possibilities and the column possibilities.

So what happens when we update the row or the columns possibilities?
Well to understand that we must realize that all cells start with all possibilities being valid ones (1 through 9) but as we do any kind of assignations these possible values can stop being possible for a certain cell. In no case a cell can "win" possible values from an assignation, it can only restrict more and more the options for it.

- **Row/Column update:** Called when there is an attempt to do an assignation, we ask KakuroConstants which are the possible values for the row/column given the space, the sum (if it is assigned) and the values used so far.
    - (Possible scenario) If the sum is not assigned and the call to KakuroConstants only returns possibilities for a specific sum then the assignation responsible for this update has forced this row/column to have only one possible sum assignation and the success of it depends on the success of the sum assignation to this row/column.

  We check which of the possibilities can actually be assigned to the row/column by checking if the values that are needed can be assigned to the remaining cells according to their set of still possible values. This can reduce a lot the number of different possibilities for the row/column.
  If we find a cell in that row/column with an annotated possible value that is no longer possible for any of the cells in the row/column (we have the possibilities from the call we just made), then we have to erase that value from that cell's possible values list.
  We also have to erase the values that will surely need to go to certain cells in the row. For example, if a cell has possible values 1,2,3, but two others have possible values 2,3, then it is clear that these two cells will be assigned these two values and thus, the first cell will no longer have them as possible, and 1 will be assigned to it.

When we finnish checking for these possible changes, we have a list of cells that have been affected (have lost possibilities) by the update.
There are three important scenarios that we need to check for each affected cell:

- ➢ It has no possible values left: The assignation responsible for this update has forced a situation where a cell can't have any values, this means that the assignation can't be done. We return that the assignation failed.
- ➢ It has only one possible value left: The assignation responsible for this update is forcing this cell to take a value, so the success of the operation will be determined by the success of assigning this forced value to this cell (which will call the corresponding updates).
- ➢ It has more than one possible value left but at least one of the possibilities that was removed was essential to a possible row or column sum combination, in which case the success of the operation will be determined by the success of updating the row and column of the affected cell. For efficiency purposes, if we update a certain row/column because of one of these changes and no other changes occur, then for the rest of cells modified in the same row/column we don't need to call the update function for that row/column because we already know there will be no changes.

If an assignation process is not successful we have been annotating all the changes into some data structures that, in this case, will tell us exactly how the state of the board was before the assignation attempt and we are able to rollback the changes.

Pseudocode of the general idea for the algorithm:

```
generate() {
      // Step1
      prepareBoard(); // Makes the assignation of white/black cells
                      // into a new board, no values are assigned.
                      // This process is well documented above.

      preprocessBoard(); // Indexes every row/column in the board
                      // and initializes the data to be used during
                      // the generation process.

      initializeDataStructures(); // We use the SwappingCellQueue
                      // and KakuroFunctions classes to assist in
                      // the generation process.

      // Step2
      findStartingPoints(); // Assigns values to certain
rows/columns
                      // to have some white cells with forced
                      values // (defined as starting points). The
                      amount
                      // depends on the difficulty.
```

```
        // Step3
        while (!swappingCellQueue.isEmpty()) {
              WhiteCell c = swappingCellQueue.getFirstElement();
              if (!rowSumIsAssigned(c) or !colSumIsAssigned(c)) {
                    forcingSumAssignation(c); // Assign a value to the
                                // row or column (or both) that is not
                                // assigned to force a value into c.
              } else if (c.notations.size() == 2) {
                    forceSumAssignationToNeighbour(c); // Recursive
                                // function that tries to find a cell in
                                // c’s row or column that can be forced
                                // into taking a value in c’s
                                // possibilities. As c has only 2
                                // possibilities this would force c to
                                // take the other value.
              }
        }

        // Step4
        for (c : whiteCellsWithoutValuesAssigned) {
              for (n : c.notations) {
                    if (tryToAssignN(c).isSuccessful) {
                          // SwappingCellQueue will get updated.
                          if (!swappingCellQueue.isEmpty())
                                go to Step3;
                    }
              }
        }

        if (!whiteCellsWithoutValuesAssigned.isEmpty()){
              fillRemainingWhiteCells() // No possible assignment
  left,
                          // we fill the board with legal values that
                          // don’t respect the sum assignments, this
                          // results in most probably ambiguous
                          kakuros.
        }

        generatedBoard = buildFinalKakuroBoard(); // Gathers all the
                          // data from the “working board” and builds
                          // the board to be delivered in initial
                    state.
  }

  getGeneratedBoard() {
        return generatedBoard;
```

```
}
```