Cedric Williams, Janelle Zeng, Daniel Kim

In this constraint satisfaction problem, the group implemented three different algorithms to solve KenKen puzzles: the simple backtracking search solution, our best, advanced backtracking search solution, and our best local search solution (while limiting the number of iterations to ensure that the method terminates).

Before going into the explanations of each method, here are some clarifications on the terminology that we use to describe the KenKen puzzle. When we refer to a cell, we are referring to each partition that shares a mathematical expression. When we refer to a box, we are referring to individual squares of the KenKen puzzle. When we refer to a value, we are referring to a number that belongs in each box.

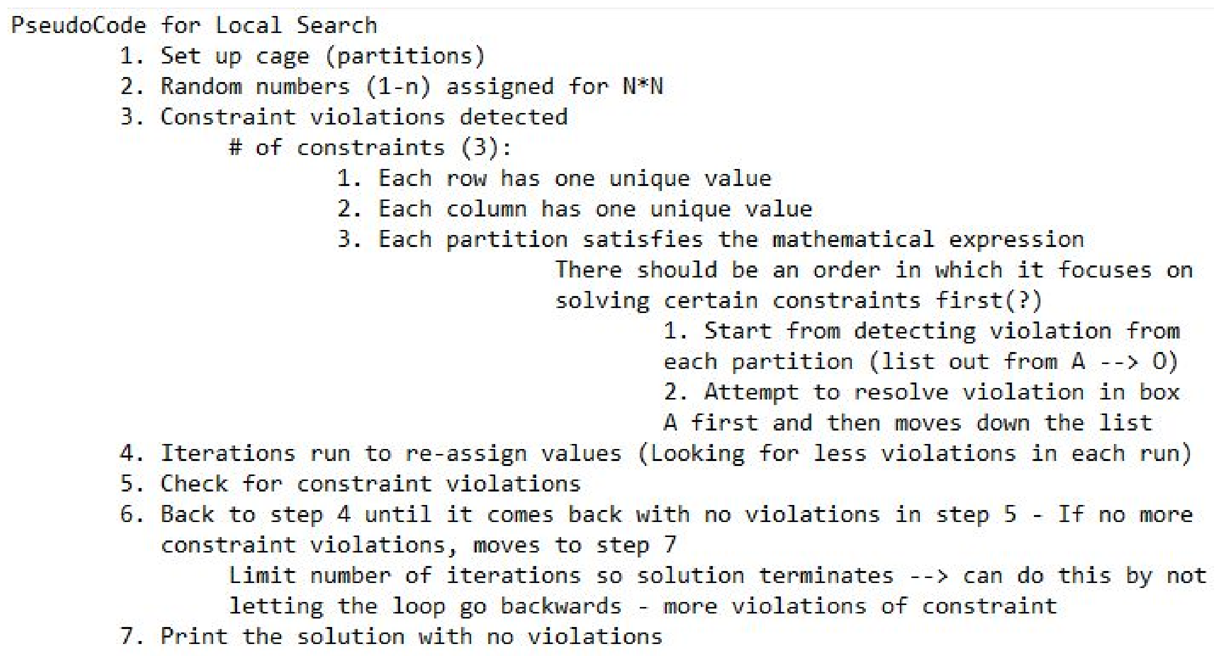
To implement our simple backtracking method, we created an N-ary tree called Search Tree, which is populated by nodes. Each node represents the values that have been tested for each cell of the inputted KenKen puzzle, and the depth level of the tree represents how many cells have been tested or completed. Once the max depth of the tree reaches N*N, the puzzle has been solved as all the values for the cells in the KenKen puzzle have been found. Therefore, the backtracking method runs until the search tree has a depth of N*N. The method checks to see if a value (1-N) meets the constraints of the puzzle for the next cell over from the cell represented by the current node (in the case of the current node being the root of the tree, the next cell is the first cell of the puzzle). If there is a possible value, a new node with that value is created and added as a child of the current node. If there is no possible value, the parent of the current node becomes the new current node, and other values that meet the constraints are tried. For the simple backtracking search, there are three constraints that the test value has to meet: no values can be the same in the row or column, and it has to satisfy the mathematical expression associated with the cell.

In order to check for duplicate values in a row, the depth of the current node is found. The depth of the cell we are trying to find values for is actually the depth of the current node + 1. Since we know the size of the KenKen puzzle, the column value of the current node is the depth % N (and if this returns a 0, the column value is N). We iterate through the row by checking if the test value is equal to the value of a traverse node that is first set to the current node, then setting the traverse node equal to its parent. If there is a duplicate, the method checkRow(int test) returns false, and we repeat that depth % N - 1 times in order to test all the values in the row that have been declared for the puzzle solution. If the method has successfully iterated all the way through, the method returns true as there are no duplicate values found.

Similar to the previous method, in order to check for duplicate values in a column, we have to get the depth of the cell we are trying to find. We know that the cell that we are trying to find is in the same column as the cell represented by a node that is depth - N. We iterate through the row by checking if the test value is equal to the value of the node at depth - N in the tree. If there is a duplicate, the method checkColumn(int test) returns false, and we repeat that until the depth is negative after subtracting N from it some number of times. If the method has successfully iterated all the way through, the method returns true as there are no duplicate values found.

In order to implement our best backtracking search, we referenced our simple backtracking search. We knew that in order to refine our simple backtracking to make it advanced, we could add more constraints. We created a new class for the advanced backtrack algorithm, which is the subclass/extension of our simple backtrack. First, we established a hash table to store a cage coordinate <key> and a set of potential numbers that fit both the total and the operator constraints. This step was to be done before the algorithm attempted to solve the problem, and we decided to call it the preCheck. After the preCheck, the puzzle starts at single-cell cages and reduces the set of numbers in the domain, making the domain even smaller. Additionally, we created a method called cellReduction, an extension of the preCheck, which uses the cage domain and the temporary variables to test if the test values placed in a specific cell cause another cell further in the puzzle to not have a value. We determined that the most constrained variable in the KenKen puzzle was the single-cell cages. In order to identify them, we created a method called findSingle.

In order to implement our local search, we started by populating the empty N*N puzzle with random values (1-N). The puzzle started with initial, random values which most likely had multiple violations of each constraint of the puzzle - no repeated number in each row, no repeated number in each column, and each cell satisfying the mathematical expression associated with it. In each iteration of the local search method, a random box is swapped with another random box. After the swap, the total number of constraint violations is counted by adding the number of each violation. If the sum of new violations after the swap is less than the sum of violations prior to the swap, that indicates that we are getting closer to the solution and the swap accepted. However, if a new number of violations after the swap is higher than the number of violations prior to the swap, then the swap is rejected, and we revert the swapping of the two values and try a different swap of random boxes. We set the random number swapping to iterate 10,000 times so that it goes through enough iterations to get closer to the goal of solving the puzzle with no constraint violations but also so that it eventually comes to an end, even without finding a solution. We used minimum-conflicts heuristic as a utility function to guide our local search. As the local search method gets closer to having fewer constraint violations, the closer we get to the solution which would have zero violations and the puzzle fully solved.



```
PseudoCode for Local Search
        1. Set up cage (partitions)
        2. Random numbers (1-n) assigned for N*N
        3. Constraint violations detected
              # of constraints (3):
                      1. Each row has one unique value
                      2. Each column has one unique value
                      3. Each partition satisfies the mathematical expression
                                      There should be an order in which it focuses on
                                      solving certain constraints first(?)
                                              1. Start from detecting violation from
                                              each partition (list out from A --> O)
                                              2. Attempt to resolve violation in box
                                              A first and then moves down the list
        4. Iterations run to re-assign values (Looking for less violations in each run)
        5. Check for constraint violations
        6. Back to step 4 until it comes back with no violations in step 5 - If no more
           constraint violations, moves to step 7
              Limit number of iterations so solution terminates --> can do this by not
              letting the loop go backwards - more violations of constraint
        7. Print the solution with no violations
```

Individual contributions of each group member were the following:
Cedric - wrote code for processing input file & Advanced Backtracking, organizing team meetings
Janelle - wrote code for LocalSearch, Simple & Advanced Backtracking
Daniel - wrote pseudocode for LocalSearch, wrote project paper and the ReadMe file.