Solution 1: Basic Backtracking

Our barebones backtracking solution is based around the solve() method which recursively calls itself in order to implement the backtracking functionality. In the event that a dead end is reached, the value of the most recently modified point is set back to 0 and the recursion for that point is completed, essentially backtracking to the next most recently modified point.

Our solution finds new points to fill using the findBlank method, which returns a point with a value of 0. findBlank returns the 0 with the lowest x value, breaking ties (points within the same row) using the lowest y value. Once a blank is found, the isValid method is attempted on the blank point using values 1 to N. isValid returns true if and only if the checkCol, checkRow, and checkCage methods return true. The checkCol and checkRow methods check if there are duplicates within the current point's column or row. The checkCage method uses the current point's getCage to determine the operation being used by a given cage, and uses different logic for each operation to determine whether or not the cage contains valid values for its operation. If the isValid method returns true, the selected point's value is set to the value from 1 to N which passed isValid.

solve(m) is then called recursively, and if solve(m) returns false for a given point, the method backtracks through the matrix of points as many steps as necessary until isValid returns true for a given point-value combination, at which point the method begins stepping deeper into the recursion. This process repeats until all points have been filled with a value, when the base case returns true. At this point, the puzzle is solved.

Solution 2: Improved backtracking

Our improved backtracking solution is based around the solveBestBack() method which, like the basic backtracking method, recursively calls itself in order to implement the backtracking functionality. We improved on the basic backtracking by implementing MRV functionality. To accomplish this we created a new method called findMRV() which returned the Point with the fewest possible values. We noticed that even in large 7x7 or 8x8 KenKen puzzles the majority of cages had size 2. We decided to treat cages of size 3 or greater differently than those cages with size 2 or 1. In our Cage class, before we began inserting values and recurring, we generated all possible values 1 to N that satisfied the Cage's associated expression. For example, if a Cage's expression is 5+, then that Cage would have {4,1,2,3} as its list of possible values. This list would then be given to each Point in the Cage, thus limiting the values we attempt for each Point to just the ones in the list. The solveBestBack() method would find the Point with the shortest list of possible values and iterate through those. Then, it would check if it was valid to insert one value from that list. If it was valid, the Point's value would be updated and consequently the value would be removed from all other Points' lists of possible values in that row and column. Then, the method would call itself and pass the updated Matrix with this new value in it. When it encounters that no value from a Point's list is valid, then it starts backtracking and returns the Matrix to a previous state where it can start trying to solve again.

When compared to the basic backtracker solver, the main improvements came from finding the Point with the fewest possible values each time the solveBestBack() method was called. The basic solver would naively iterate through the Matrix going from one point to its neighbor. The improved solver would jump around the Matrix. Another improvement made was having a list of possible values for each Point and Cage. This addition saved a lot of time since the solver didn't have to naively iterate through every value 1 to N like before.

Solution 3: Local Search

Our local search implementation cycles through N^5 randomized matrices, swapping values between rows if they are equally or less restraining. We picked N^5 as the max number of randomized matrices because it seemed to give us the best balance between efficiency and actually reaching a solution, although we also could have implemented a while loop to keep creating new random matrices until a solution was found. The random matrices are created using the populateMatrix method which generates a semi-randomized matrix of points. Each row of an NxN matrix produced by populateMatrix contains one of each number from 1 to N randomly placed within the row. The following is a 6x6 matrix produced by the populateMatrix class:

342651
156243
345261
312654
153462
342615

Each point also has an associated num_constraints property which represents whether or not a point has a conflict within the column and/or the cage. A num_constraints value of 1 represents a point that a.) has the same value as another point in its column or b.) cage expression that is invalid at that point. A num_constraints value of 2 corresponds to both of those two cases, and a num_constraints value of 0 corresponds to neither of these cases. A num_constraints value of 0 does not necessarily mean that the point has the correct value unless the entire matrix's points have num_constraints values of 0. The populateMatrix class fills in these num_constraints values by looping through each point and running the updateConstraints method on it.

updateConstraints is a method of the KenKen class uses checkCol and checkCageLocal to come up with a num_constraints value and set it. checkCol has already been discussed, and checkCageLocal is essentially the same as the earlier checkCage, but since every point in the local search matrix has a value, there is no val property being passed or evaluated. Instead

checkCageLocal is run using the arraylist of points contained in the current point's getCage method.

Once the matrix is populated, the solveLocalSearch() method is called 4N times on a given randomly generated matrix. We reached this number by experimenting with different values, and 4N seemed to give us the best balance between efficiency and reaching a solution fairly consistently. Each of these passes up to 4N represent one "pass" through the entire matrix, trying to swap the values using solveLocalSearch.

The solveLocalSearch method attempts to swap each value in a given row with another value in the same row using the trySwap method. trySwap swaps the values of the two points it is passed and checks to see if making this change reduces or maintains the sum of the two points' num_constraints property. If swapping doesn't increase the total num_constraints value of the two points, the values are maintained and updateConstraints is run on each point in the column and cage of both of the swapped points. If swapping increases the total num_constraints value, the two points are restored back to their old values and num_constraints.

The matrix class also contains a totalNumConstraints field which contains the total value of num_constraints across the entire matrix. This value is updated after every trySwap call in solveLocalSearch, and if the value of this field is zero, the puzzle has been solved. If the value is equal to zero, a message is printed to the console confirming that local search has found a solution, and a value of true is returned to the localSearchStarter class, which in turn returns a value of true to the initial call in main. If the outer loop contained within localSearchStarter runs N^5 times without producing a value, we exit the loop and print a statement saying that the program failed to find a solution.

Our utility function attempted to minimize the totalNumConstraints field by minimizing the combined value of num_constraints for each attempted swap.

Statement of Contribution:

We both spent roughly equal time working on the basic backtracking together, and then we spent equal time planning how to go about doing the improved backtracking. Initially, we both spent all of our time working on the improved backtracking, but as the assignment went on, Angello was primarily responsible for the improved backtracking solution and Brian was primarily responsible for the local search solution. With that being said, there was a significant amount of help being given both ways on these last two solutions.