P1 Written Document

Kakuro is a type of logic puzzle where the goal is to fill in a grid of cells with numbers from 1 to 9 such that each row and column contains unique numbers and the sum of numbers in each row and column is equal to a provided sum.

The solution process is initiated by first formulating the problem. The problem details are read from an input file. My Kakuro function splits the input into three parts, the commands, puzzle, and constraints; and makes use of the details provided to formulate the CSP. To formulate this Kakuro CSP, the problem components are defined as follows:

Variables:

There are 12 variables in total, one for each empty cell in the grid. In the code, each variable is represented with a tuple (i, j); i is the row number and j is the column number of the cell. Initially made with a 5 by 5 nested loops, the variables were reduced by deleting the elements that were not fillable in the provided puzzle input, leaving 12 tuples for the 12 spots in the puzzle.

Domains:

Each variable was assigned the domain set {1, 2, 3, 4, 5, 6, 7, 8, 9} as at the beginning all are possible values for each.

Constraints:

Firstly, all variables in a row or column must have distinct values (no repeated numbers)

The sum of variables in a row or column must be equal to the specified sum provided.

The variables in each clue must be distinct.

To represent the sum constraints, I use a list of tuples, where each tuple has 3 elements containing details of the constraint:

1. The first cell of the tuple contains another tuple for the coordinates (row, column) of the sum. It’s important to match the corresponding variables by row or column.
2. The second cell contains the specified sum that the assigned values should add up to
3. The third cell contains the orientation of the sum (horizontal or vertical), in order to determine whether the sum should constitute the corresponding rows or columns.

To add the constraints to the CSP problem, I made two different Constraint classes. The first, uniqConstraint() checks two different variables to ensure that they have distinct values. The second, sumConstraints() a variable, a sum to which the variable’s value is supposed to add to, and a list of the variables whose values should add up to the sum. By crosschecking assigned values with the expected sum, one is able to apply the sum constraint/ clue to solve the problem.

After formulating the problem, the solver selects the first variable to assign a value to. It picks the unassigned variable that appears earliest in the list, ignoring the minimum remaining value function since the minimum remaining value component is inputted as false.

Next, the solver selects a value to assign to the variable. It begins by assigning the first value in the domain set.

Next, the solver checks if the value assignment is consistent with the specified problem constraints. In this case, that is to check whether the value is distinct from others in its row and column and whether when added up with corresponding values, the sum is consistent with the provided sum.

If the value retains consistency, it is assigned to its designated variable and the solver recursively calls itself to begin the value assignment for the next variable. If not consistent, the value is not assigned to the variable and the process is backtracked to the previous variable where the solver goes on to the next value from the domain set and tries to assign it. If it is at the end of the domain set, the process backtracks to the previously assigns variable and attempts to reassign it.

If all the variables are assigned values that satisfy all constraints, the process terminates providing a solution. If there are no more variables to assign values to and a consistent solution is not found, the process terminates returning a failure.

Function Implementation:

Node Consistency:

For a value in the domain of a variable, the function checks whether the value, if assigned, would satisfy the constraints of the problem as well as be consistent with the currently formulated solution. It first selects the value and performs a check. If not consistent, the function returns False and the value is removed from the domain of potential values, If consistent, the value returns True allowing for value assignment. The function then grabs another value and begins the process again, repeating this process until all variables only have domains consistent with a solution that satisfies the constraints.

Minimum Remaining Values:

This function is applied in choosing what variable among unassigned variables to next assign a value to. It does its selection by checking for the variable with the least number of possible values that can be assigned to it. This is by determining the sizes of each variable’s domain set. The variable whose domain is smallest in size gets selected and returned for assignment. The function when called upon repeats this process until all variables are assigned.

Least Constraining Value:

This function is applied in choosing which value to assign. It selects the value by checking for which value would reduce the domains of possible values for the other variables by the smallest margin. The function selects a variable to assign a value to. It then determines the remaining unassigned variables. It then obtains the respective domains for these variables. The function then calculates the number of potential values for other variables each value in its current variable’s domain would eliminate. It then returns the value which eliminates the least number of potential values for corresponding variables. This value is assigned to the variable and the process is repeated until all variables are assigned.

*References*

My Kakuro CSP imports from the model generic CSP class adapted from the course textbook. As a beginner in Python programming, I struggled to implement the pseudo-code provided and get it to work. As such, I used the adapted Generic CSP class to formulate my constraint game, which is seen in Kakuro.py. This was the only method that provided me with a successful output. I however modified the functions in the Generic CSP file to suit my Kakuro constraint problem thus getting it to work.

The aspects of the generic CSP that I used are minimal and my Solver runs as required and prints the correct output into output.txt. I hope this does not invalidate my work. Thanks.