Kakuro CSP solver

This program is intended to solve a Kakuro puzzle. A Kakuro puzzle is a grid with specific squares blocked out with a number on it referring to a row or column going down or right of the blocked square (can be both a row and col coming out of a blocked square). Each non blocked out square needs a number from 1 to 9 that when summed with the rest of its group (the row or col between an blocked square and an edge or between 2 blocked squares) equals the number in the blocked square that correlates with said row or col as well as each of these squares in each group holding a unique number.

Variables, domains, and Constraint formulation

Variables: variables for this function are the squares on the grid $X_{i,j}$

- Where $X_{i, j}$ is a coordinate with $0 < i, j < N$ as the row, col on a N by N grid
    - $0 < I, j$ because row and col 1 will be constraints and won't have values

Domain: {1, 2, 3, 4, 5, 6, 7, 8, 9}

- each variable $X_{i, j}$ can take one of these values which represents its contribution to the sum constraint

Constraints: the constraints of the kakuro CSP are

- $\forall$ i, j ($X_{i, j}$) is an element of {1, 2, 3, 4, 5, 6, 7, 8, 9}
    - Where X is a var with coordinates i, j
    - The value on the grid can be '#' meaning it's the start or end of a constraint or just an edge piece and not a variable.
    - If the value is not '#' it is a variable that is part of a constraint and must be in domain {1, 2, 3, 4, 5, 6, 7, 8, 9}
- For each sum constraint C that constrains from i,j->n,m there exists a set of values from $X_{i,j->n,m}$ such that $\sum \{X_{i,j}, \ldots, X_{n,m}\} = SUMC_{i,j->n,m}$
    - Where $\{X_{i,j}, X_{n,m}\}$ is a set of variables in the range i,j to n,m with $X_{i,j}$ as the first variable constrained and $X_{n,m}$ is the last in the row/col group being constrained
    - $SUMC_{i,j->n,m}$ is the number that constrains the range i,j to n,m
- AllDiff($X_{i, j, \ldots}$, $X_{n, m}$)
    - Where $X_{i,j}$ is the start of a constraint row and $X_{n,m}$ is the end of one

CSP Implementation

To solve the kakuro puzzle the program will, using the function get_csp, create a multidimensional list called grid where each node is accessed using grid[row][col] (row = I, col = j from constraint section). This grid will be the CSP as all the information related to the CSP formulation is accessed through this grid. Each node is an instance variable called var containing its current val (grid[row][col].num, which is initialized as '0') and the domain of remaining valid values (grid[row][col].domain) as well as the other var's they share constraints with. The shared constraints come in the form of 2 lists of coordinates for each direction (grid[row][col].constH.nodes and grid[row][col].constV.nodes), the coordinates can be retrieved and unpacked to use in later functions. If var an edge piece or a constraint piece it will have '#' as its num which is both used in determining the domain of each constraint and for visual representation at the beginning and end. Finally for when FC or MAC is being utilized the domain has a

list that saves the previous domain each time it gets pruned (grid[row][col].rollBack). The constraints are also instance variables that contain their own starting block (const[x].row, const[x].col), the items its constraining (const.nodes, which is empty initially and filled by find_rect()), what direction the constraint constrains (const.direction) and the sum constraint itself (const.num). finally for the get_csp function the constraints get their row/col groups assigned by find_rect() which takes input lists grid and const and assigns a const for each direction to the corresponding var.

Backtracking search implementation

The backtracking search algorithm is essentially the given pseudocode adapted to the implementation of the optional inferences. It first finds a unassigned var (var.num == '0') using the find() function which returns either the coordinates to the first unassigned var or uses MRV() to return the var with the smallest domain, if find returns None the program must be done as all variables have a number assigned. Otherwise the coordinates are saved. Next the domain must be determined, if LCV is enabled the domain is set to the sorted list that LCV() returns, else the domain of the selected var is used. Once the variable and the domain are determined the algorithm loops through the given domain and calls check_violations() which returns true or false. If true and any inference is enabled it will check those next, if inferences pass or there are none to check recursively call backtrack again to solve the CSP. If an inference or check_violations() fails it rolls back the changes made by the inferences as well as the value that var.num was assigned and backtracks by returning false to the recursive call to backtrack(). The backtracking function is done when no more vars can be found by find().

Check_violations()

This function is the part of the backtracking search that determines whether the value placed from the domain of the current var violates any constraints. It takes the CSP, the value that was selected and the position at which it was placed. This function first checks each row/col group for the var at the given position for unassigned variables. If there is even 1 unassigned var that means the sum constraint isn't ready to be checked. If there are no unassigned vars in a row or col group that means this value filled that last slot and the can be checked against the constraint. To check the sum constraint all the values in the constraint group are summed and if they do not equal the sum constraint the function returns false. For alldiff, the function checks the given num against the var.nums of each var in the row/col group and that the node being checked isn't the node passed to check_violations(). If both are true it also returns false otherwise if all tests are passed no violations are found and true is returned.

FC implementation

The FC() function takes the CSP, the value being placed and the position of the var that the value is placed in as its arguments. This function starts with looking for and counting unassigned vars in both row/col groups for the given var. If any unassigned var gets found its domain is saved to rollback as it's likely it will get changed when pruning occurs. If just a single unassigned val remains that means this row can be forward checked to predict what the sum should be. It does this by assigning a bool to each direction for the var. If either or both directions have just one unassigned val the bool is false and the function removes the values that when combined with the others in its group don't add to the sum constraint of that group. If the whole domain gets pruned, then we can backtrack early as there can be no valid value when the backtrack algorithm reaches this unassigned var. If there are more than 1 unassigned vars in a row the function will just check alldiff by pruning the domains of the nodes in the

row/col groups that have the same value as the val being placed at this step. Again if the domain is pruned until it is empty return false.

MRV implementation

The minimum remaining values function MRV() sits inside of the normal find function which returns the first unassigned variable. Of course instead of the first var MRV finds the first of the unassigned vars with the smallest domain by looping through each Var that still has '0' as its num and gets the length of its domain and storing the value only if its smaller than the current smallest. Meaning if (1, 2) and (4, 4) are both '0' and have the same domain the var with position (1, 2) will be selected.

LCV implementation

For the LCV implementation LCV(grid, pos) takes the grid and the variable selected by either the sequential find or the MRV find as its input and loops through its domain checking for each number how many times it shows up in the neighbors (under the same constraint) then returns a sorted array with each entry being a number from the domain and its occurrences, sorted from least to most by the occurrences. When this list is returned the backtracking will occur as normal but looping through this order instead.

AC3 implementation

AC3() is implemented in the form of a function AC3() which takes the CSP as an argument. AC3 is a preprocessing step to reduce the initial domain and has every arc from the entire CSP in the initial queue. This algorithm simply loops through all the nodes and loads all their arcs and then loops through them passing RIV() the most recently popped arc. If RIV() returns removed, reload the arcs involving the var that was just passed.

MAC implementation

The underlying logic is very similar for both AC3() and MAC() with the main differences being that MAC() gets called after every assignment with just the arcs involving the assignment being initialized in the queue to prune future domains (meaning it requires a positional argument as well as the CSP) additionally while each arc is being loaded in MAC() each var in said arc gets its domain inserted in the first position of rollback as it's being used in backtrack. finally if the RIV() function returns an arc where the domain is empty for arc[0] it will return false to backtracking.

RIV() (for AC3 and MAC)

The RIV() function takes the csp and all the arcs provided by AC3 or MAC and determines the range of valid values in the domain of the of arc[0] based on the domain of arc[1] for the sum and alldiff constraints. If we assume the other members of the constraint group are at the highest values they can take (ex: if there is 3 vars in the group, the 'rowTotal' will be 9 as arc[0] and arc[1] will loop through their domains pruning arc[0], if there was 4 'rowTotal' would be 9 + 8 and onward) and then see which numbers from arc[0] + each number in arc[1] + rowtotal => (rowTotal) > the sum. For any number in that formula that is greater than the sum is potentially a valid number. For the alliff it simply measures the domain of each arc and if num is in the domain of arc[1] and it's length is 1 prune that number from arc[0]. If anything was removed set removed to True and return it otherwise its set to false.

Helper functions:

find()

find() is a function that takes the CSP and the list of bools and returns the next var for the backtrack algorithm. It does this either by looping sequentially through the grid and finding the first unassigned var if MRV is not enabled. If MRV is set to true, MRV() is called instead and as described before returns the first of the unassigned vars with the that have the minimum domain size (of all vars that == minimum domain length chose the first found).

find_rect()

find_rect() is a part of the process in making the CSP. It takes all the constraints from a list and adds each node that is between the constraints position and the next var with the value of '#' in the direction determined by const[x].direction. This forms the row/col groups that represents each constraints affect on either a row or a column.

rollback() and rollbackMAC()

These functions are called by backtracking search and both simply look for unassigned vars in the current vars row/col group (because if the value is unassigned its likely it was changed) and pop off the first list in rollback. It does this because as the algorithm recursively backtracks it will pop off the most recent change each time which will match with how far back the algorithm needs to backtrack.