Ripple Effect

A Constraint Satisfaction Problem

Simon Joon-Hee Song: g2junhee

model design, problem encoding

Chae Min Ahn: g4ahncha

constraint reinforcement, experimental assessment

Project Background:

We tried to develop a CSP algorithm to solve a logic puzzle, called Ripple Effect[1], which is a Nikoli[2]-original logic puzzle. Ripple Effect uses a rectangular board divided into multiple areas of varying size and shape called "Rooms". The starting board contains a few cells with pre-assigned values. The goal of a Ripple Effect problem is to fill in the remaining empty cells with numbers so the final board satisfies all of the following rules:

1. A room contains numbers starting from 1 to the size of the room.
2. Each number in a room occurs once.
3. If a number is duplicated within a row or a column, the space between the two duplicates must be equal to or larger than the value of the reoccurring number.

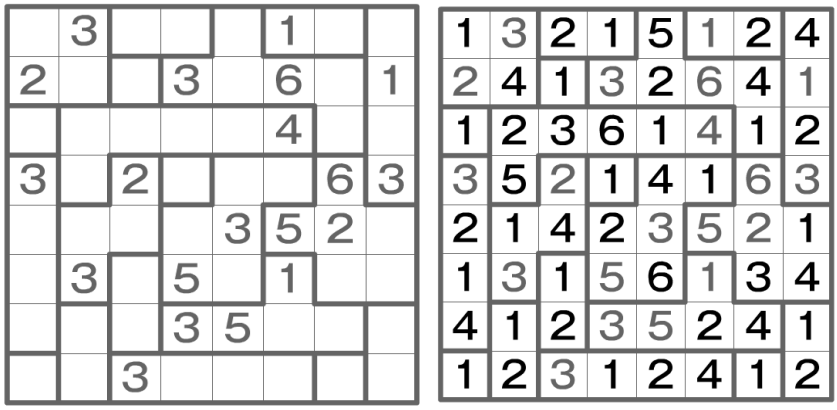

Figure 1. a typical Ripple Effect board and its solution

The CSP approach was well suited for this problem because each cell had a restricted number of possible values, which could be easily expressed as variables and their domains. Constraints could be used to enforce the last two rules. For example, we could see that for rule #2, we should have a n-ary all-different constraint scoping over the cells in the room, where n is the size of the room. We were also able to distinguish the goal state very easily, as all cells had to be filled in in a way that satisfies all of the constraints abovementioned. In addition, the Nikoli website provided numerous Ripple Effect puzzle boards and their solutions we could use to check the validity of our CSP algorithm.

Methods:

Since the CSP approach was chosen, we first had to formulate the adequate variables, then create constraints under the two rules.

1. Formulate Variables

   A Ripple Effect board has a size of M x N. Hence for each puzzle, we created M x N variables first and stored them in an array. We designed the problem so that each puzzle board is associated

[1] Ripple Effect, http://www.nikoli.com/en/puzzles/ripple_effect/

[2] Nikoli, a website hosting numerous logic puzzles, including original ones.

with an array, '`rooms`', which contains all the information about the rooms of the board, in a form of list of lists of tuples. Each sub-list contains a set of tuples representing the coordinates of the cells in each room. For example, the first room in Figure 1 would be `[(0, 0), (0, 1), (1, 0), (1, 1)]`. So when formulating each variable, we would take the size of the room the variable is within, and set their domain as 1 through `len(room)`, according to rule #1.

2. Find Constraints

   The last two rules of Ripple Effect puzzle could be converted to two different types of constraints. For rule #2, we introduced a n-ary all different constraint for each room. For each constraint, we took the range of possible values, which would again be 1 through `len(room)`, then computed all permutations of the values and added them to the constraint's satisfying tuples. For example, if a room contained three variables `v1`, `v2`, and `v3`, we would compute out all permutations of `(1, 2, 3)`, namely `[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]`, then create a constraint which scopes over `[v1, v2, v3]`, whose satisfying tuples would be the above permutated tuples. For faster performance, we removed already assigned values from the range of possible values, computed permutations, then inserted back the assigned value with respect to the assigned variable's order in the constraint's scope to the created permutations.

   For rule #3, we introduced multiple binary constraints between all possible combinations of variables within the same row for each row. We also did the same for each column. This rule was tricky to enforce since we had to know the assigned value x of $V_{(i,j)}$ to restrict the variables $V_{(i,j)}$ to $V_{(i+x,j)}$, $V_{(i,j)}$ to $V_{(i-x,j)}$, $V_{(i,j)}$ to $V_{(i,j+x)}$, and $V_{(i,j)}$ to $V_{(i,j-x)}$ from also being assigned x. So instead, we noticed how variables distanced by 6 or less would never be assigned `(6, 6)`, and variables distanced by 5 or less would never be assigned `(5, 5)`, and so on. So we created `max_rsize` number of lists, where `max_rsize` is the size of the biggest room in the board, thus the biggest assignable value in any variable's domain and the biggest distance apart any pair of same-valued variables would have to be by, and each list`[i]` would represent the satisfying tuples for `(i+1)`-distanced variables. Then we generated all possible pairs of assignable values for all of `max_rsize` lists, and removed same-valued pairs from appropriate lists. For example, `(6, 6)` pair would be removed from list`[0]` through list`[5]`, `(5, 5)` pair would be removed from list`[0]` through list`[4]`, and so on. When actually creating constraints for different combinations of variables within the same row or column, we calculated the distance apart between the two variables, say `(i+1)`, and assigned list`[i]` as the constraint's list of satisfying tuples.

Evaluation and Results:

We want to evaluate the success of our Ripple Effect CSP algorithm, first by comparing the algorithm-computed solutions to actual solutions of the puzzles, and secondly by asserting some expected hypotheses:

1. Performance of Ripple Effect CSP algorithm is inversely proportional to size of board.
2. Performance of Ripple Effect CSP algorithm is inversely proportional to size of rooms in board.
3. Performance of Ripple Effect CSP algorithm is directly proportional to # of pre-assigned values.

We employed the two strategies to see whether our CSP algorithm was solving logic puzzles right, thus proving successful, and to see if it assumed expected behaviors of CSP algorithms. Also by observing performance results, we could see the efficiency of our algorithm.

So we have created several test boards and utilized the forward checking and generalized arc consistence constraint propagators to compute solutions. All test boards were taken from the Nikoli website, and we compared the solutions of our CSP algorithm with the provided solution on the website to test validity of our algorithms. Given the same pre-assigned values on the same board, all solutions matched.

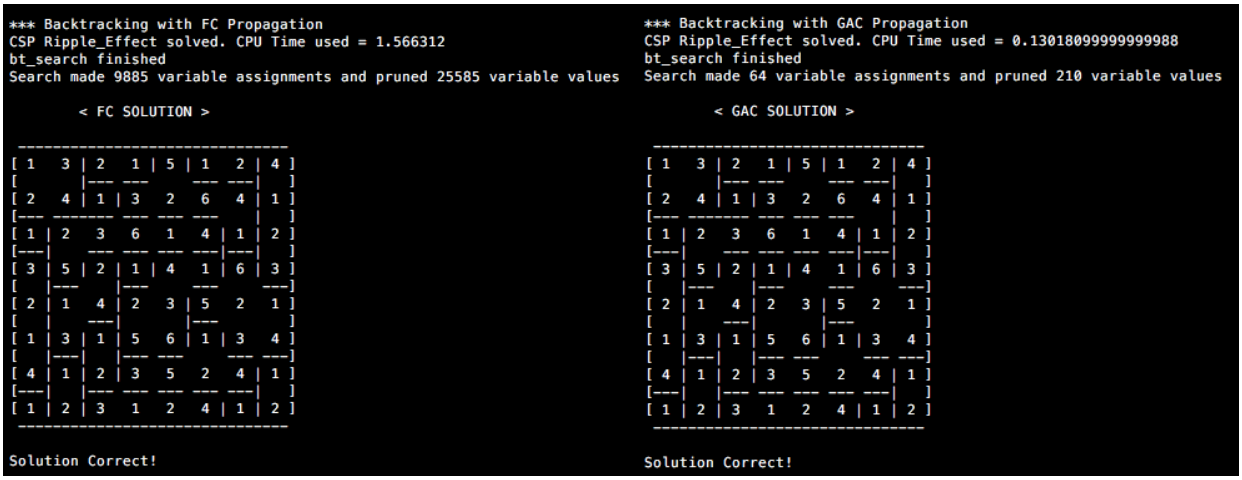

Figure 2. typical solution output, for each FC and GAC propagation methods

To assert the abovementioned hypotheses, we have planned to observe the performances of different propagation methods on different boards, as well as discover the effects of various aspects of a board such as size of board, size of rooms in board, and number of pre-assigned values in board. We measured performance in terms of low CPU time used, fewer number of assigned variables and fewer number of pruned variables.

propagation method	**CPU time used**				
	puzzle 1 (small)	puzzle 2 (medium)	puzzle 3 (medium)	puzzle 4 (medium)	puzzle 5 (large)
FC	1.5663	0.0263	0.1439	0.2361	157.4750
GAC	0.1302	0.1078	0.2990	0.3366	0.7125

Table 1. CPU time used for different puzzles

propagation method	puzzle 1 (small)		puzzle 2 (medium)		puzzle 3 (medium)		puzzle 4 (medium)		puzzle 5 (large)	
	assigned	**pruned**	**assigned**	**pruned**	**assigned**	**pruned**	**assigned**	**pruned**	**assigned**	**pruned**
FC	9,885	25,585	100	148	474	1,029	948	2,236	506,804	1,097,493
GAC	64	210	100	148	100	264	100	274	180	475

Table 2. number of assigned variables and pruned variables for different puzzles

Puzzle 1 is of size 8 x 8, and contains rooms with size up to 6, and has 20 pre-assigned values.
Puzzle 2 is of size 10 x 10, and contains rooms with size up to 3, and has 0 pre-assigned values.
Puzzle 3 is of size 10 x 10, and contains rooms with size up to 6, and has 6 pre-assigned values.
Puzzle 4 is of size 10 x 10, and contains rooms with size up to 6, and has 4 pre-assigned values.
Puzzle 5 is of size 18 x 10, and contains rooms with size up to 5, and has 11 pre-assigned values.

From Table 1, we could see the clear disparity of CPU time results between the two propagation methods. GAC, in general, showed consistent results with respect to board sizes, with some notable trend and reasonable difference between each results. FC, on the other hand, had drastic difference in some of its results. This is due to the fact that FC only forward checks constraints with one uninstantiated variable, instead of looping through all relevant constraints and enforcing consistency, hence FC is rather dependent on what the actual solution is and how soon it can be reached within the search space than the GAC. We could also inspect from Table 2, that GAC propagation method was very efficient and only carried out the least necessary number of assignments, which is the total number of variables in a board, thus proving the efficiency of our Ripple Effect CSP implementation.

The general trend was that, as the board size grew larger, the more time a backtracking routine took. Although the FC took longer time for puzzle 1 (8 x 8) than it did for the three 10 x 10 boards, we could inspect the general trend of CPU time being directly proportional to board size through the rather consistent GAC results. Hence we have asserted hypothesis #1 true. We also want to note the exceptionally low CPU time results for puzzle 2 with both FC and GAC methods. This is likely due to the fact that despite its 10 x 10 size, puzzle 2 contains rooms with size up to only 3, meaning the largest domain for any variable in the board could be `[1, 2, 3]`, which is very small. From this discovery, we have also asserted hypothesis #2 true.

# of pre-assigned values	FC					
	CPU time		**# assigned**		**# pruned**	
	trial 1	trial 2	trial 1	trial 2	trial 1	trial 2
20	1.5776	1.5948	9,885	9,885	25,585	25,585
16	13.7956	8.2625	88,881	52,627	224,508	137,511
12	46.2340	169.7583	280,156	1,053,661	733,891	2,730,283
8	515.8965	242.8838	3,030,911	1,496,885	7,620,364	3,857,042
4	63.9226	124.2913	397,812	769,843	949,411	1,931,887
0	0.4120	0.3929	2,549	2,549	5,717	5,717

Table 3. FC results for different number of pre-assigned values

# of pre-assigned values	GAC					
	CPU time		**# assigned**		**# pruned**	
	trial 1	trial 2	trial 1	trial 2	trial 1	trial 2
20	0.1314	0.1273	64	64	210	210
16	0.1370	0.1387	64	64	209	210
12	0.1937	0.1839	66	65	228	214
8	0.2049	0.2065	66	66	231	219
4	0.3798	0.2862	90	65	399	209
0	0.3986	0.3929	65	65	209	209

Table 4. GAC results for different number of pre-assigned values

From Table 3 and Table 4, we could observe the effect of number of pre-assigned values on CSP performance. To test this, we looped through the process of running a backtracking routine, then un-assigning 4 pre-assigned values at random from the previous board used. We went through this process twice, both trials on puzzle 1. In Table 4, which show all GAC results, we could see the CPU time gradually grow as the number of pre-assigned values decreased, in both trials. So no matter which variables were assigned or un-assigned, the directly proportional relationship between the number of pre-assigned values and performance maintained. (hypothesis #3)

One interesting thing to note was the unregulated FC results with respect to number of pre-assigned values. The results were all over the place as FC depends largely on which variables are assigned or un-assigned, but eventually performance improved with only a few pre-assigned values. A possible explanation for this may be that GAC takes larger calculation overhead to enforce consistency between all constraints, and then takes a move, whereas FC could just get lucky if the solution takes place early in its search space.

Limitations:

One of the shortcomings of the way we modeled our Ripple Effect CSP is the `rooms` array. We found it challenging to generate `rooms` array from the board input, with all border representations such as '-', '|' and ' ', and have implemented the CSP model so that `rooms` have to be manually given. This is greatly time consuming, and is apt for human errors such as same variable appearing twice. Also the amount of effort and time as the board size grows will be unbearable.

Conclusion:

We have developed a CSP algorithm to solve the Ripple Effect logic puzzle. We learned that Ripple Effect rules are well suited for conversion to constraints in a constraint satisfaction problem, and we were able to successfully implement Ripple Effect CSP algorithm. Our success was measured in terms of validity, as tested by actual solutions from Nikoli website, and also in terms of efficiency, as seen in all GAC results that make only the least necessary number of assignments in less than 0.3 of a second in most cases.

One interesting extension to our project would be to approach rule #3 differently. Currently, the constraints under rule #3 involve generating all valid pairs of assignments and associating them with binary constraints scoping over all differently-distanced variables within a row or a column. It is very space-expensive to create all such constraints beforehand, especially as board size grows. Instead of using pre-defined constraints that prohibit all cases of violation, we can modify our propagators to add distance-related constraints as the values get assigned, and undo the constraint if we were to un-assign variables. This approach might compute solutions slower, but will use much less space overhead. We may be interested in the trade-offs between the compute time and space overhead.

Also, we should try to find a way to simplify the board representation and implement function to generate `rooms` array from the board input. This would reduce the preprocess effort greatly.