CSCI 555 - AI
Professor Alden Wright
06dec2012

Sudoku Solver

Objective

The goal of this project was to learn about constraint satisfaction problems by solving Sudoku as a specific case. The techniques to be used in this software are arc consistency, path consistency, and backtracking. The final output of the program is the original puzzle and the solution or the final state from which the puzzle could not be solved.

Project Description

```
program input file          program output

x x x  x 5 4  x 2 x         687|954|321
x x x  6 2 x  x x x         913|627|584
x 4 x  x 1 3  x x 9         245|813|679
                            -----------
x x x  x x x  x x x         458|271|936
3 7 x  x x 8  x 4 2         379|568|142
x x x  4 x x  x 5 x         162|439|758
                            -----------
8 x x  x x 5  x x x         896|345|217
x 2 1  x x x  x x 3         521|786|493
7 x x  x 9 2  x 6 x         734|192|865
figure 1                    figure 2
```

The software takes a single text file in the form shown by figure 1. It then uses the above techniques to solve the puzzle. These techniques are powerful when applied together, but so effective enough when applied individually.

The rules of Sudoku are as follows

1) Each cell must be assigned a value from 1 to 9.
2) Each row, column, and sub-square must have only one occurrence of each number.

Constraints

In a constraint satisfaction problem there are different ways of defining larger constraints; however, all such constructs must have some rule that reduces that constraint to a set of binary constraints. A binary constraint $\{X_i, X_j\}$ is said to be true when $X_i \neq X_j$. In the case of Sudoku there are 27 AllDiff constraints, which are the rows, columns, and sub-squares. Each of these constraints form 45 binary constraints for a total of 1215 binary constraints, but among these 1215 constraints there is some duplication because from the perspective of any single cell each sub-square AllDiff constraint shares 4 cells in common with the row and column constraints as shown by the highlighting in figure 1.

Arc Consistency

An arc is formed when two cells are joined by a binary constraint, and any two cells joined by an arc are said to be neighbors regardless of their positions in the puzzle. Specifically in Sudoku, each cell has 20 unique neighbors.

The AC-3 algorithm acts on each cell only through the domains of the individual cells of the puzzle. The domain of each cell is defined to be the possible values that a cell could be while still satisfying all of the binary constraints. For example in figure 1, the top left cell has a domain of {1,6,9} because the row already has {5,4,2}, the column has {3,8,7}, and the sub-square has {4}.

A basic summary of the algorithm's logic is as follows. Put every binary constraint into a queue. Take a binary constraint (X_i, X_j) from the front of the queue and attempt to revise the domain of X_i. If the domain of X_i is revised, then check to make sure there is some values remaining in X_i's domain. If there aren't any, then fail the entire algorithm immediately. Otherwise, add a binary constraint (X_k, X_i) to the end of the queue for every neighbor X_k of X_i except for X_j, which we just checked. If the domain of X_i is not revised, then do nothing and move on to the next binary constraint in the queue. The revise step removes conflicting values from the domain of X_i with respect to X_j. After the entire queue has been processed and each cell is arc consistent with the smallest domain possible, the final step is to assign values to any cell with a domain of size one.

Path Consistency

I wasn't able to find enough information to implement path consistency, but the basic idea is that additional information can be inferred beyond the basic binary constraints that the problem starts with by using hypothetical syllogism. This additional information either allows the complete inference step to be completed while checking fewer constraints or to make significantly more reductions to the domain of each cell during the inference step.

Backtracking

The basic idea of the backtracking algorithm is to guess the value of each cell one by one until the solution is found, but back out of the choice when it doesn't work out. All alone backtracking is extremely inefficient as I will describe in the graph theory analysis; however, in combination with a good inference algorithm backtracking can make a few very smart guesses to solve a Sudoku puzzle very efficiently. The backtracking algorithm works in a two step process, guess and infer. Any valid inference algorithm can be used for the infer step. This program uses the AC-3 algorithm described above.

Before the backtracking algorithm is run, the program will first run the inference algorithm in case that alone is enough to solve the puzzle. But even if the inference algorithm isn't enough to solve the puzzle outright, it will minimize the domain of each cell in the puzzle, giving the backtracking algorithm the most information possible to make a good guess.

The guess step uses the domains of each cell to make its guess in the following way. First it chooses a cell on which to make a guess. It does this by finding a cell with the smallest possible domain (ideally of size two) because it will always have a higher probability of guessing correctly on these cells. Once a cell has been chosen the actual guess is done by prioritizing the values in that cell's domain. This is done by following all of the arcs leading out of the cell to peek at the cell's neighbor's domains. The algorithm will count how often each value (1 through 9) occurs in the neighbor's domains. These counts are then used to prioritize the value in the cell's own domain because any value that shows up often in the neighbor's domains is less likely to be the correct choice for this cell.

A basic summary of the backtracking algorithm's logic is as follows. First, attempt to make a guess. If every value in the cell's domain has been tried attempt to backtrack, but if backtracking isn't possible report that the puzzle has no solution. Once a guess has been made, run the inference algorithm of your choice to reduce the domains of all unassigned cells, then assign values to any cell whose domain was reduced to size one. Repeat until the puzzle is solved.

Graph Theory Analysis

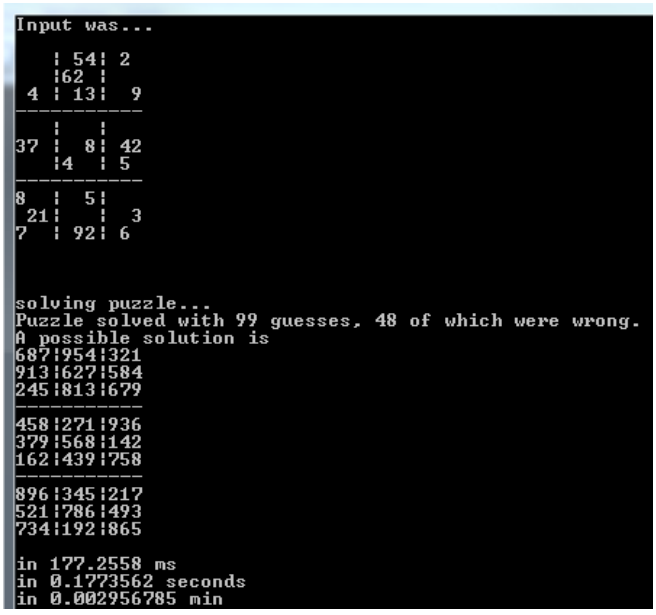
Now to tie some of these concepts together with graph theory, giving a clearer picture of the problem size that Sudoku represents. This problem can be thought of as a graph search problem where every possible board state (legal or not) is a vertex and every possible guess from a board state (legal or not) is an edge. It's easy to see that this would be a very naive way to solve Sudoku and that a person using this strategy would be unlikely to actually solve the puzzle. Likewise, a computer attempting a brute force a Sudoku puzzle in this way would have to search through trillions and trillions of game states to find the answer. No modern computer could do this in a human lifetime for a problem like Sudoku.

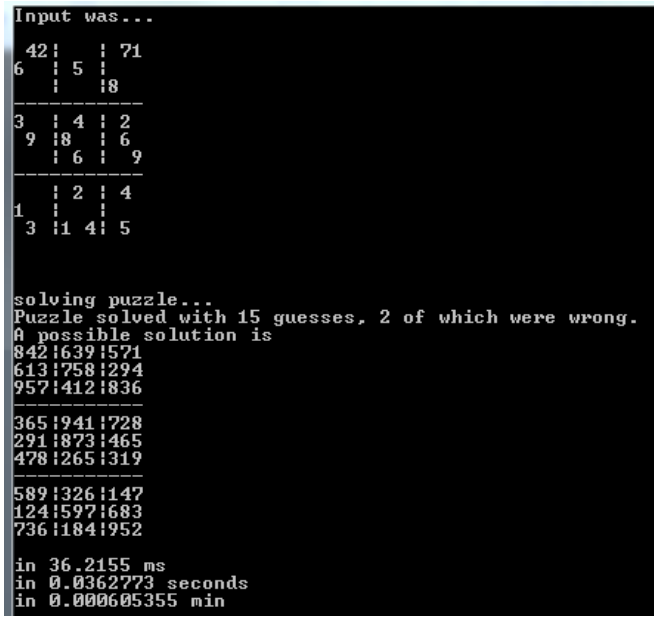
There is good news! We can remove our naivety one step at a time. For example if we only include legal board states among the vertices and only make legal moves for our edges, our graph becomes a tree of considerably less size. This would be like solving Sudoku by using only guesses without assigning any values by inference. In other words, this is still an extremely naive solution, but we've reduced the size of the problem enough that a computer could theoretically do this search in minutes to hours depending on how lucky we get on a particular puzzle.

The final step where we add inferences in between each guess reduces the size of our search tree from an average height of 60-70 to an average height of 10-20. This means that instead of searching through possibly 70^2 or 70^4 guesses, we only need to look at 10^2 or so guesses and that we'll be very likely to find the answer early on in that search. For example, if a puzzle that takes 100 guesses to solve is processed in .2 seconds, then a puzzle that takes 70^4 guesses would be processed in 33 days 8 hours and 20 minutes. However, a puzzle that takes 70^2 guesses would still be processed in 9.8 seconds, so depending on how good a job we do with our book keeping on the cell's domains even a fairly naive solution can solve Sudoku efficiently.

Results

I believe that there are no Sudoku puzzles that the solver fails to solve when a solution exists because the search done by backtracking only gives up searching a particular branch of the search space when it's 100% certain that no solution exists on that branch. For some of the most challenging Sudoku puzzles I could find, the solver is able to find the solution in .04 to .2 seconds with most very hard puzzles taking about .1 seconds. Additionally if more than one solution does exist, the program will stop when the first solution is found, so the results will be somewhat random.





Images of the program's output solving two different Sudoku puzzles.

Finally, I decided to introduce a small amount of randomness into the backtracking algorithm. When choosing a cell to guess on by smallest domain size, I now break ties between cells with the smallest domain at random. For most puzzles, this was a noticeable improvement in running speed. Over 2000 trials per puzzle the average running times seem to vary between 40ms and 100ms with a max of around 400ms in some cases but a min of around 10ms in others. I also considered adding weights to the domain values rather than ordering them outright because I could then choose the guess from the weighted domain at random, but over 2000 trials on several puzzles the average running time had almost no variation with and without randomized weighted domain values. A difference of at most 4 ms on one of the puzzles.

References

S. J Russel and P. Norvig. (2009). *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall.

Y. Zhang, R. H. C. Yap 2001. *Making AC-3 an Optimal Algorithm*. In Proceedings of IJCAI-01. pg 316-321.