Sudoku Solver Utilizing Logical Solving Techniques and Recursive Backtracking

Authors: Jonathan Boyd, Department of Physics, Kent State University, Kent, OH 44242
Emily Hoopes, Department of Mathematics, Youngstown State University, Youngstown, OH 44504
Mentors: Dorothy Klein, Department of Mathematics, Kent State University, Kent, OH 44242
Dr. Thomas Wakefield, Department of Mathematics, Youngstown State University, Youngstown, OH 44504

Introduction

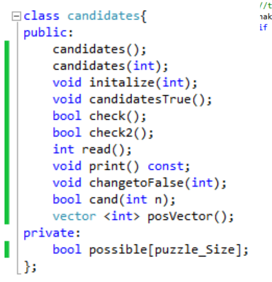
Sudoku is a puzzle games that is usually played on a 9x9 grid of boxes. The objective is to fill in the missing numbers following a simple rule: the numbers 1-9 may only appear once in each row, column, and 3x3 box of the grid. Some puzzles can be solved easily using a process of elimination in each set, however, some require complex strategies that utilize relations between cells. Our solver program applies both simple and complex methods to complete any puzzle.

Research Statement

Create a Sudoku solver that is capable of solving any 9x9 puzzle using logical and computational techniques.

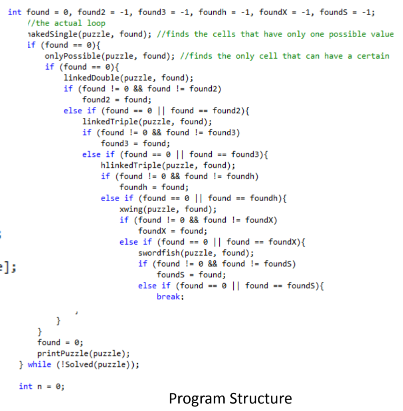
Methods

The puzzle begins as a 9x9 array of a custom class called cell. The cell class contains an integer value for the single as well as an array of Boolean values for possibilities. A function reads a txt file and begins filling in the known single values for a particular puzzle. After filling in these values a second function scans the puzzle for the inputted single values and begins changing the Boolean values in each row, column, and box that contains a single value. Following this the puzzle is passed to the solving portion of the code.



```
class candidates{
public:
    candidates();
    candidates(int);
    void initalize(int);
    void candidatesTrue();
    bool check();
    bool check2();
    int read();
    void print() const;
    void changetoFalse(int);
    bool cand(int n);
    vector <int> posVector();
private:
    bool possible[puzzle_Size];
};
```

Class Candidates Structure



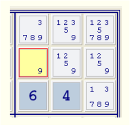
```
int found = 0, found2 = -1, found3 = -1, foundh = -1, foundX = -1, foundS = -1;
    //the actual loop
    nakedSingle(puzzle, found); //finds the cells that have only one possible value
    if (found == 0){
        onlyPossible(puzzle, found); //finds the only cell that can have a certain
        if (found == 0){
            linkedDouble(puzzle, found);
            if (found != 0 && found != found2)
                found2 = found;
            else if (found == 0 || found == found2){
                linkedTriple(puzzle, found);
                if (found != 0 && found != found3)
                    found3 = found;
                else if (found == 0 || found == found3){
                    hlinkedTriple(puzzle, found);
                    if (found != 0 && found != foundh)
                        foundh = found;
                    else if (found == 0 || found == foundh){
                        xwing(puzzle, found);
                        if (found != 0 && found != foundX)
                            foundX = found;
                        else if (found == 0 || found == foundX){
                            swordfish(puzzle, found);
                            if (found != 0 && found != foundS)
                                foundS = found;
                            else if (found == 0 || found == foundS){
                                break;

        }
    }
    found = 0;
    printPuzzle(puzzle);
} while (!Solved(puzzle));

int n = 0;
```

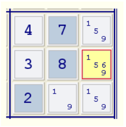
Program Structure

The first strategy utilized is naked single. The program searches the array for a cell with only one possibility in the Boolean values. It then takes this number and assigns it to the integer value and updates the row, column, and 3x3 box with the updated information for that assigned value.



Visual Representation of a Naked Single

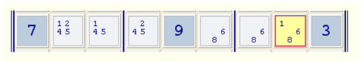
If the program finds a naked single it prints the updated puzzle and begins the solving function cycle again. If no naked singles are found It moves on to hidden singles. Hidden singles occur when there is only one of a given number as a possibility in any row, column, or box. The cell containing that possibility must have that value as its single value for the set to have all the numbers 1-9. The program then assigns it and updates the puzzle like before and begins the loop again.



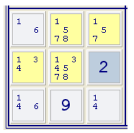
Visual Representation of a Hidden Single

The next function is linked doubles. The doubles can be naked or hidden. If they are naked, then the rest of the cells in the group have those candidate values changed to false. If two values occur exactly twice in a group, then the cells are compared. If the cells share the same two possibilities, then those candidate values are set to false for the rest of the cells in the group.

Visual Representation of a Naked Double Pair

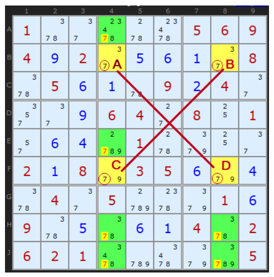


The program moves on to finding triples if no doubles are found. Triples, like doubles, relies on looking at the candidate values for cells. Between three cells, there are three possible values. Not all cells need to contain all three possible values. The program works by scanning every row, column and box. If a value occurs two or three times, that value is stored in a vector. Once it finds three values that have returned true two or three times, it then compares to see if they all occur within the same three cells. Then it changes those values to false in the rest of the group.



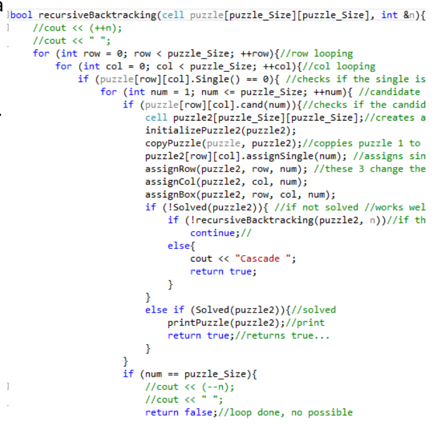
Visual Representation of Triple Set

The next to last technique is X-wing. In X-wing two rows (or columns) have the same two possibilities in the same two columns (or rows). Then all other candidates for this value can be eliminated from the columns (or rows). The code works by checking one row and seeing if a possible value is true for two cells. Then, it checks all of the rows after it to see if any other rows have only two possible cells for the value. If these statements are true, then it changes that possibility to false for every cell in that column.



Visual Representation of X-wing

The final technique is a brute force recursive backtracking function that is capable of solving any puzzle by itself. It begins by scanning the puzzle starting with the top left corner. It checks to see if the cell is assigned a single. If not, it makes a copy of the puzzle makes a guess using the truth values of the candidates for that cell, and passes the new puzzle to another iteration of the function. When it gets a contradiction, it "backtracks" to the previous cell and tries another guess. This process continues until the correct solution path is found.



```
bool recursiveBacktracking(cell puzzle[puzzle_Size][puzzle_Size], int &n){
    //cout << (++n);
    //cout << " ";
    for (int row = 0; row < puzzle_Size; ++row){//row looping
        for (int col = 0; col < puzzle_Size; ++col){//col looping
            if (puzzle[row][col].Single() == 0){ //checks if the single is
                for (int num = 1; num <= puzzle_Size; ++num){ //candidate
                    if (puzzle[row][col].cand(num)){//checks if the candid
                        cell puzzle2[puzzle_Size][puzzle_Size];//creates a
                        initializePuzzle2(puzzle2);
                        copyPuzzle(puzzle, puzzle2);//coppies puzzle 1 to
                        puzzle2[row][col].assignSingle(num); //assigns sin
                        assignRow(puzzle2, row, num); //these 3 change the
                        assignCol(puzzle2, col, num);
                        assignBox(puzzle2, row, col, num);
                        if (!Solved(puzzle2)){ //if not solved //works wel
                            if (!recursiveBacktracking(puzzle2, n))//if th
                                continue;//
                            else{
                                cout << "Cascade ";
                                return true;
                            }
                        }
                        else if (Solved(puzzle2)){//solved
                            printPuzzle(puzzle2);//print
                            return true;//returns true...
                        }
                    }
                    if (num == puzzle_Size){
                        //cout << (--n);
                        //cout << " ";
                        return false;//loop done, no possible
```

Recursive Backtracking Structure

Results

The solver is a success on many puzzles of varying difficulty, however gets stuck on complex puzzles. These puzzles are very sparse with given values and require the recursive backtracking function. This function works for any puzzle, however, given the number of possible branches in a Sudoku grid, it takes longer to run than desired, multiple hours or days.

Conclusions

Additional solving techniques applied before recursive backtracking would cut down on the number of paths to be checked and therefore cut back on the required time.

Future Work

-Develop additional functions from other techniques (Swordfish).
-Expand it to work with any size puzzle (i.e. 16x16).
-Create a graphical user interface.
-Develop it to help people solve puzzles on their own with hints.

References

Johnson, A. Solving Sudoku. Retrieved February 17, 2015, from http://angusj.com/sudoku/hints.php
Main Page. (n.d.). Retrieved February 17, 2015, from http://www.sudokuwiki.org/