Instructions:

- This project was written in C++ on Visual Studio and is saved as a .cpp file.
- Before running the program you need to have input files in the same folder as the .cpp so that you can read from them.
- On line number 307, which is shown here: “file.open("Input1.txt");”, you need to input the file that you want the program to read from.
- In order to write to a file, that file must be in the same folder as the .cpp
- To indicate which file you want the program to print the results to, you can change the filename on line 292, shown here: “myfile.open("Output1.txt");”.
- The file that is written to should appear in the same file folder that houses your .cpp file and your input files after the program is run.

Formulation:

- The set of variables X = {x1,x2,etc} consists of each square in the sudoku puzzle. Each square is a separate variable. There are 9 rows and 9 columns, so there are 81 different variables.
- The set of domains D = {d1,d2,etc} consists of the integers from 1 to 9 inclusive. The domains shift during computation for each variable but the values stay within 1 to 9 inclusive.
- The set of constraints C = {c1,c2,etc} is that all items in a shared row, in a shared column, and in a shared block have to be a different number. Every variable in a row, column, or block is constrained by the other variables that are also in that row, column, or block

Source Code:

```
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <sstream>
#include <tuple>
#include <algorithm>
using namespace std;

bool is_complete(vector<int>& content) { //checks to see if the sudoko board is completely filled out
	for (size_t i = 0; i < content.size(); ++i) {
		if (content[i] == 0) { //if an empty space is found its false
			return false;
```

```
                }
        }
        return true;
}

bool determine_poss_values(vector<int>& content, vector<tuple<int, int, int, vector<int>>>&
rows_columns) { //analyzes the board
        for (size_t i = 0; i < content.size(); ++i) { //determines for each variable its possible values
(iterates over the entire board)
                get<3>(rows_columns[i]).clear(); //clears whatever possible values were there
before
                if (content[i] != 0) { //if content is not zero it already has a value assigned to it
                        continue;
                }
                vector<int> curr_row_vals;
                vector<int> curr_column_vals;
                vector<int> curr_block_vals;
                vector<int> possible_vals;
                curr_row_vals.push_back(content[i]); //the variables in the same row as the
current
                curr_column_vals.push_back(content[i]); //same column as current
                curr_block_vals.push_back(content[i]);//same block as current
                for (size_t j = 0; j < content.size(); ++j) {
                        if ((i != j) && (get<0>(rows_columns[i]) == get<0>(rows_columns[j]))) { //if
each variable has the same row and isn't the current variable
                                curr_row_vals.push_back(content[j]);
                        }
                        if ((i != j) && (get<1>(rows_columns[i]) == get<1>(rows_columns[j]))) {//if
each variable has the same column and isn't the current variable
                                curr_column_vals.push_back(content[j]);
                        }
                        if ((i != j) && (get<2>(rows_columns[i]) == get<2>(rows_columns[j]))) {//if
each variable has the same 3x3 block and isn't the current variable
                                curr_block_vals.push_back(content[j]);
                        }
                }
                for (size_t j = 1; j < 10; ++j) { //iterates through every possible value (1 to 9)
                        bool already_exists = false;
                        for (size_t k = 0; k < curr_row_vals.size(); ++k) {
                                if (curr_row_vals[k] == j) { //sees if value is already in the current
variables (i) row
                                        already_exists = true;
                                        break;
                                }
```

```
                        }
                        if (already_exists != true) {
                                for (size_t k = 0; k < curr_column_vals.size(); ++k) {
                                        if (curr_column_vals[k] == j) { //sees if value is already in
the current var column
                                                already_exists = true;
                                                break;
                                        }
                                }
                        }
                        if (already_exists != true) {
                                for (size_t k = 0; k < curr_block_vals.size(); ++k) {
                                        if (curr_block_vals[k] == j) { //sees if value is already in the
current var block
                                                already_exists = true;
                                                break;
                                        }
                                }
                        }
                        if (already_exists == false) { //if the value does not exist in any of the
above then it's added to the possible values for that variable
                                get<3>(rows_columns[i]).push_back(j);
                        }
                }
                if (get<3>(rows_columns[i]).size() == 0) { //if there are no possible values left and
since it's established to be 0 then it's an inconsistency and failure
                        return false;
                }
        }
        return true;
}
int degree_test(vector<int>& content, vector<tuple<int, int, int, vector<int>>>& rows_columns,
vector<int>& mrv_winners) { //determines which mrv winner has the highest number of variables
it constrains
        int max = 0;
        vector<tuple<int, int>> constraint_vals; //first int is the cumulative number of variables it
constrains and second int is the original value to be able to return it
        vector<int> degree_winners;
        int rand_index = 0;
        for (size_t i = 0; i < mrv_winners.size(); ++i) {
                int num_in_rows = 0;
                int num_in_cols = 0;
                int num_in_blocks = 0;
                for (size_t j = 0; j < rows_columns.size(); ++j) {
```

```
                        if ((j == mrv_winners[i]) || (content[j] != 0)) { //if the iteration is the current
var or already assigned then skip
                                continue;
                        }
                        if (get<0>(rows_columns[j]) == get<0>(rows_columns[mrv_winners[i]])) {
//if row equals current val add one
                                num_in_rows += 1;
                        }
                        if (get<1>(rows_columns[j]) == get<1>(rows_columns[mrv_winners[i]]))
{//if column equals add one
                                num_in_cols += 1;
                        }
                        if ((get<2>(rows_columns[j]) == get<2>(rows_columns[mrv_winners[i]]))
&& (get<0>(rows_columns[j]) != get<0>(rows_columns[mrv_winners[i]])) &&
(get<1>(rows_columns[j]) != get<1>(rows_columns[mrv_winners[i]]))) {
                                num_in_blocks += 1; //if block equals and it is not in the same row
or column (those were already counted above) (it prevents repeat counting)
                        }
                }
                constraint_vals.push_back(make_tuple(num_in_rows + num_in_cols +
num_in_blocks, mrv_winners[i])); //holds the constraint values for each of the mrv winners
        }
        for (size_t i = 0; i < constraint_vals.size(); ++i) { //determines the max value out of the
constraint values
                if (get<0>(constraint_vals[i]) > max) {
                        max = get<0>(constraint_vals[i]);
                }
        }
        for (size_t i = 0; i < constraint_vals.size(); ++i) { //sees if there are any duplicate max
values in the list
                if (get<0>(constraint_vals[i]) == max) {
                        degree_winners.push_back(get<1>(constraint_vals[i])); //every var that
has that value is added to the list of winners
                }
        }
        if (degree_winners.size() > 1) { //if its still more than 1 left choose randomly
                srand(time(0)); //prevents random function from returning the same value each
time
                rand_index = degree_winners[rand() % degree_winners.size()]; //random winner
is chosen
        }
        else {
                rand_index = degree_winners[0]; //only 1 winner
        }
```

```
	return rand_index;
}
int select_unassigned_variables(vector<int>& content, vector<tuple<int, int, int, vector<int>>>&
rows_columns) { //determines which variable should be chosen next
	vector<int> mrv_winners;
	int min = 15;
	int index = 0;
	for (size_t i = 0; i < rows_columns.size(); ++i) { //determines and compares the number
of possible values that each variable has
		int curr_size = get<3>(rows_columns[i]).size();
		if (curr_size == 0) { //already assigned
			continue;
		}
		if (i == 0) { //first one to be tried
			min = curr_size;
		}
		else {
			if (curr_size < min) { //determines the minimum value of possible choices
out of the board
				index = i; //saves which variable has the lowest number of
possible value choices
				min = curr_size;
			}
		}
	}
	mrv_winners.push_back(index);
	for (size_t i = 0; i < rows_columns.size(); ++i) { //sees if there are any other variables with
the same min value
		if (get<3>(rows_columns[i]).size() == min) {
			if (i != index) { //if so it adds them to the winners list
				mrv_winners.push_back(i);
			}
		}
	}
	if (mrv_winners.size() > 1) { //if its more than 1 in the winners list then you use the
degree test to determine which to pick
		int next_ind = degree_test(content, rows_columns, mrv_winners);
		return next_ind;
	}
	else {
		return mrv_winners[0]; //if its just 1 return that 1
	}
}
```

```
vector<int> begin_backtracking(vector<int>& content, vector<tuple<int, int, int, vector<int>>>&
rows_columns) { //start algorithm
	if (is_complete(content) == true) { //if board is full
		return content;
	}
	int next_var_index = select_unassigned_variables(content, rows_columns); //variable
next is chosen
	vector<int> temp_vals;
	for (size_t i = 0; i < get<3>(rows_columns[next_var_index]).size(); ++i) {
		temp_vals.push_back(get<3>(rows_columns[next_var_index])[i]);
	}
	sort(temp_vals.begin(), temp_vals.end()); //sorts the possible values of the current
variable in order to test from least to greatest
	for (size_t i = 0; i < temp_vals.size(); ++i) {
		content[next_var_index] = temp_vals[i]; //sets curr var to curr val
		bool consistent = determine_poss_values(content, rows_columns);
		if (consistent == true) { //if consistent continue
			vector<int> result = begin_backtracking(content, rows_columns);
//recursive call returns either state or failure
			if (result[0] != 99) { //99 indicates failure, if not return success
				return result;
			}
		}
		content[next_var_index] = 0; //if that val is inconsistent reset var to 0 and reset
state
		determine_poss_values(content, rows_columns);
	}
	vector<int> failure; //if no vals work return failure
	failure.push_back(99);
	return failure;
}

vector<int> change_to_ints(vector<char>& content) { //changes original char data from file to
ints
	vector<int> content_int;
	for (size_t i = 0; i < content.size(); ++i) {
		if (isdigit(content[i])) {
			stringstream str;
			str << content[i]; int x; str >> x;
			content_int.push_back(x);
		}
	}
	return content_int;
}
```

```
void fill_in_3x3(vector<tuple<int, int, int, vector<int>>>& rows_columns) { //determines for each var which 3x3 block it belongs in (1 to 9) (1 being leftmost top increasing horizontally with 9 being rightmost bottom)
	for (size_t i = 0; i < rows_columns.size(); ++i) {
		if ((get<0>(rows_columns[i]) < 4) && (get<1>(rows_columns[i]) < 4)) { //if var row is below the fourth one if col below the fourth one
			get<2>(rows_columns[i]) = 1; //assigns block number to var so block = 1
		}
		else if ((get<0>(rows_columns[i]) < 4) && (get<1>(rows_columns[i]) >= 4) && (get<1>(rows_columns[i]) < 7)) { //same concept with different specifics regarding placement on board
			get<2>(rows_columns[i]) = 2; //block = 2
		}
		else if ((get<0>(rows_columns[i]) < 4) && (get<1>(rows_columns[i]) >= 7)) {
			get<2>(rows_columns[i]) = 3; //block = 3
		}
		else if ((get<0>(rows_columns[i]) >= 4) && (get<0>(rows_columns[i]) < 7) && (get<1>(rows_columns[i]) < 4)) {
			get<2>(rows_columns[i]) = 4; //block = 4
		}
		else if ((get<0>(rows_columns[i]) >= 4) && (get<0>(rows_columns[i]) < 7) && (get<1>(rows_columns[i]) >= 4) && (get<1>(rows_columns[i]) < 7)) {
			get<2>(rows_columns[i]) = 5; //etc
		}
		else if ((get<0>(rows_columns[i]) >= 4) && (get<0>(rows_columns[i]) < 7) && (get<1>(rows_columns[i]) >= 7)) {
			get<2>(rows_columns[i]) = 6;
		}
		else if ((get<0>(rows_columns[i]) >= 7) && (get<1>(rows_columns[i]) < 4)) {
			get<2>(rows_columns[i]) = 7;
		}
		else if ((get<0>(rows_columns[i]) >= 7) && (get<1>(rows_columns[i]) >= 4) && (get<1>(rows_columns[i]) < 7)) {
			get<2>(rows_columns[i]) = 8;
		}
		else if ((get<0>(rows_columns[i]) >= 7) && (get<1>(rows_columns[i]) >= 7)) {
			get<2>(rows_columns[i]) = 9;
		}
	}
}
void fill_row_column_vals(vector<int>& content, vector<tuple<int, int, int, vector<int>>>& rows_columns) { //determines each variables initial column row and block
	for (size_t i = 0; i < content.size(); ++i) { //clears any possible values and initializes the rows and columns to 0
```

```
                vector<int> possible_values;
                possible_values.clear();
                rows_columns.push_back(make_tuple(0, 0, 0, possible_values));
        }
        for (size_t i = 0; i < rows_columns.size(); ++i) { //for every variable depending on its ith position on the puzzle, its row, column, and block were found
                if (i < 9) {
                        get<0>(rows_columns[i]) = 1; //row = 1
                }
                else if ((i >= 9) && (i < 18)) {
                        get<0>(rows_columns[i]) = 2; //row = 2
                }
                else if ((i >= 18) && (i < 27)) {
                        get<0>(rows_columns[i]) = 3; //row = 3
                }
                else if ((i >= 27) && (i < 36)) {
                        get<0>(rows_columns[i]) = 4; //row = 4
                }
                else if ((i >= 36) && (i < 45)) { //etc
                        get<0>(rows_columns[i]) = 5;
                }
                else if ((i >= 45) && (i < 54)) {
                        get<0>(rows_columns[i]) = 6;
                }
                else if ((i >= 54) && (i < 63)) {
                        get<0>(rows_columns[i]) = 7;
                }
                else if ((i >= 63) && (i < 72)) {
                        get<0>(rows_columns[i]) = 8;
                }
                else if ((i >= 72) && (i < 81)) {
                        get<0>(rows_columns[i]) = 9;
                }
                if ((i == 0) || (i == 9) || (i == 18) || (i == 27) || (i == 36) || (i == 45) || (i == 54) || (i == 63) || (i == 72)) { // column = 1
                        get<1>(rows_columns[i]) = 1;
                }
                else if ((i == 1) || (i == 10) || (i == 19) || (i == 28) || (i == 37) || (i == 46) || (i == 55) || (i == 64) || (i == 73)) { //column = 2
                        get<1>(rows_columns[i]) = 2;
                }
                else if ((i == 2) || (i == 11) || (i == 20) || (i == 29) || (i == 38) || (i == 47) || (i == 56) || (i == 65) || (i == 74)) { //column = 3
                        get<1>(rows_columns[i]) = 3;
```

```
                }
                else if ((i == 3) || (i == 12) || (i == 21) || (i == 30) || (i == 39) || (i == 48) || (i == 57) ||
(i == 66) || (i == 75)) { //column = 4
                        get<1>(rows_columns[i]) = 4;
                }
                else if ((i == 4) || (i == 13) || (i == 22) || (i == 31) || (i == 40) || (i == 49) || (i == 58) ||
(i == 67) || (i == 76)) { //etc
                        get<1>(rows_columns[i]) = 5;
                }
                else if ((i == 5) || (i == 14) || (i == 23) || (i == 32) || (i == 41) || (i == 50) || (i == 59) ||
(i == 68) || (i == 77)) {
                        get<1>(rows_columns[i]) = 6;
                }
                else if ((i == 6) || (i == 15) || (i == 24) || (i == 33) || (i == 42) || (i == 51) || (i == 60) ||
(i == 69) || (i == 78)) {
                        get<1>(rows_columns[i]) = 7;
                }
                else if ((i == 7) || (i == 16) || (i == 25) || (i == 34) || (i == 43) || (i == 52) || (i == 61) ||
(i == 70) || (i == 79)) {
                        get<1>(rows_columns[i]) = 8;
                }
                else if ((i == 8) || (i == 17) || (i == 26) || (i == 35) || (i == 44) || (i == 53) || (i == 62) ||
(i == 71) || (i == 80)) {
                        get<1>(rows_columns[i]) = 9;
                }
        }
        fill_in_3x3(rows_columns); //determines the block number for each variable
        determine_poss_values(content, rows_columns); //determines the initial (from input)
possible values for every variable
}
void format_solution(vector<int>& solution) {
        ofstream myfile; //creates output file
        myfile.open("Output1.txt");
        for (size_t i = 0; i < solution.size(); ++i) { //writes to the output file the final solution to the
sudoko puzzle
                if ((i == 8) || (i == 17) || (i == 26) || (i == 35) || (i == 44) || (i == 53) || (i == 62) || (i
== 71) || (i == 80)) {
                        myfile << solution[i] << endl;
                }
                else {
                        myfile << solution[i] << " ";
                }
        }
        myfile.close();
```

```
}
int main() {
	ifstream file;
	char position = 0;
	vector<char> content;
	file.open("Input1.txt"); //opens file to read from
	if (!file.is_open()) {
		cerr << "Could not open the file.";
		return 1;
	}
	while (file.get(position)) { //reads from the file and adds each char val to a vector
		content.push_back(position);
	}
	file.close();
	vector<int> contint;
	contint = change_to_ints(content); //initial char vals are changed to ints for easier work
	vector<tuple<int, int, int, vector<int>>> rows_columns; //first int is each var's row, second int is each's column, third int is each's block number, and fourth vector<int> are each's possible values
	fill_row_column_vals(contint, rows_columns); //initializies the rows_columns fields for each variable
	vector<int> solution = begin_backtracking(contint, rows_columns); //returns the final solution or failure
	if (solution[0] != 99) { //99 is the failure message so if its not that then write to an output file
		format_solution(solution); //write to output file and format the solution puzzle
	}
}
```

Output 1:

1 3 2 5 6 9 7 8 4
6 8 5 2 7 4 1 9 3
4 9 7 8 3 1 2 6 5
8 5 6 4 9 2 3 1 7
3 7 1 6 8 5 9 4 2
9 2 4 7 1 3 6 5 8
2 4 9 3 5 6 8 7 1
5 1 8 9 2 7 4 3 6
7 6 3 1 4 8 5 2 9

Output 2:

4 5 3 6 7 8 9 1 2
2 8 1 5 3 9 7 6 4
9 6 7 4 1 2 3 5 8
3 7 5 1 6 4 2 8 9
6 9 4 2 8 3 5 7 1
1 2 8 7 9 5 6 4 3
8 3 6 9 5 1 4 2 7
5 4 9 8 2 7 1 3 6
7 1 2 3 4 6 8 9 5

Output 3:

5 7 6 3 4 1 9 2 8
8 2 1 9 6 5 7 4 3
9 4 3 8 7 2 5 6 1
1 6 8 4 5 7 3 9 2
2 9 7 1 3 8 6 5 4
4 3 5 2 9 6 1 8 7
3 5 2 7 8 9 4 1 6
6 1 4 5 2 3 8 7 9
7 8 9 6 1 4 2 3 5