Graph Coloring Algorithm on a Sudoku Puzzle

Hunter Lupro

May 2020

**1 Abstract**

A simple greedy algorithm is not sufficient enough to provide a solution to a preset Sudoku puzzle and results in improper coloring of the graph. To remedy this a new greedy algorithm is used that makes use of the technique known as backtracking. Further experimentation with this algorithm on three different sudoku puzzles of difficulties from easy to hard has resulted in correct solutions to each one showing that a graph coloring algorithm can be used to successfully to find solutions to Sudoku puzzles.

**2 Introduction**

The main idea of this paper is to document and describe the use of a graph coloring algorithm to solve a given Sudoku puzzle. The motivation behind implementing a graph coloring algorithm on a Sudoku puzzle is due in part to the historic problem of coloring a map in a way that no adjacent countries are the same color. Graph coloring can be traced back to this historic problem due to the ease of transforming a map into a graph. A map of countries can be easily translated into a graph by having each country be represented by a vertex and an edge is placed between two countries when they share a border. Eventually, graph coloring led to the Four Color Problem which asks if every planar graph can be colored with four colors and is famously known for being one of the first problems to be proved with the help of a computer(Voloshin).

A Sudoku grid, like a map of countries can be easily represented by a graph, with the chromatic number, the minimum amount of colors a map can be colored with, being nine.

**3 Topic Details**

Before a graph coloring algorithm can be implemented on a Sudoku puzzle, the Sudoku puzzle must be represented as an undirected graph and can be done as followed. A 9x9 Sudoku puzzle has 81 vertices each vertex will be labeled from 1-81 and can be visualized in a typeset pattern, left to right and top to bottom as seen in Table 1.

Table 1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 |
| 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 |
| 46 | 47 | 48 | 49 | 50 | 51 | 52 | 53 | 54 |
| 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 |
| 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 |

Next, an edge exists between two Sudoku cells if the two cells are in the same column or row and if they are in the same 3x3 square. Following these guidelines, every cell will share an edge with every cell in the same row, column and 3x3 square. This can be encoded with the use of the map container from the STD Library and for this approach two map containers will be used. The first container will map an integer to a vector of integers. The first number represents a vertex v in the graph, 1-81, and the vector will act as an adjacency list which will store the vertices that share an edge with vertex v. Due to the way edges in a Sudoku puzzle are created, each vertex in the map will have an adjacency list of twenty members.

The second map container will map a vertex v to another integer which will be its “color”. The number of colors available will be from 0-9 and represents what someone sees when they look at a Sudoku puzzle. Any “empty” cell that is any cell that does not have the number 1-9 will be labeled as 0 in the color map.

The last step in converting a Sudoku puzzle to a graph is to color in all the hints, or already known values, with the hints being given by any Sudoku generator or online database. By encoding a Sudoku puzzle into a text file where there are nine lines in the file with each line have nine numbers, a simple function to read in the values to the color map is used to set the map the vertices to a color. If there is no color, the number is zero like above. Next another vector is used to hold all uncolored cells and can be represented as vector unknowns. To populate this vector, iterate through the color map and each time a vertices color is found to be zero, insert it into the unknowns vector. This will be important when solving for a Sudoku solution.

Once the Sudoku puzzle has been successfully converted into a graph, coloring algorithms can be run to try and determine a possible solution. The first algorithm is a simple greedy algorithm that will choose the lowest value possible to color a vertex. The algorithm works as follows:

For vertices.begin up to end

Int c = 1

Int v = it->first

Bool sameColor = true

If(color[v] == 0)

While(sameColor)

Set same color to false

Iterate through v’s adjacency list and increment c until the color is not found in the color map of the adjacency list

The code works by going through map container that maps all 81 vertices to their adjacency lists and checking their color. If it’s zero, the bool sameColor is flipped to false to begin the while loop that will keep incrementing c until no vertex in v’s adjacency list has that color.

While this might initially seem promising, the algorithm will fail in its solution due to a point in time where 1-9 do not work as colors and the algorithm adds numbers past that. The reasoning behind this is because this algorithm lacks the idea of backtracking which is when if a vertex cannot be colored, the algorithm goes to the vertex before it in the map and tries another color and the process is repeated for that vertex if it can’t find a proper coloring.

The next algorithm implemented to find a solution to a Sudoku puzzle involves the use of backtracking to correct potential errors and move forward. The code works very similar to the first one but instead iterates through the vector unknowns instead of all 81 vertices. When the algorithm comes across a coloring problem where no number 1-9 will work, it colors that vertex 0 and calls the function backtrack(int I, int stop).

Backtrack works as follows:

While(I != stop)

Color is set to the next possible color, if c is 1 then it is now 2

Iterate through the adjacency list until c is a proper coloring or past 9

If v is colored increment i

If v is not able to find a proper color decrement i

The algorithm works by using two integers, I and stop, I is position in the vector that we are trying to color, and stop is the position in the vector not to go past. If a coloring can’t be found, then i is decremented so that the previous vertex will be recolored. Once a vertex is properly colored, i is incremented to retry and color the later vertices that depended on the vertices that are earlier in the vector. This process goes back and forth, brute forcing the proper colorings until the entire solution is available and colored.

This algorithm was tested on three different Sudoku puzzles of different difficulties and was able to provide the solution in a matter of seconds, showing that a graph coloring algorithm can be successfully used to determine the solution to a Sudoku Puzzle.

**4 Conclusion**

A simple greedy algorithm is not sufficient enough to provide a solution to a preset Sudoku puzzle. To remedy this a new greedy algorithm is used that makes use of the technique known as backtracking. Further experimentation with this algorithm on three different sudoku puzzles of difficulties from easy to hard has resulted in correct solutions to each one showing that a graph coloring algorithm can be used to successfully to find solutions to Sudoku puzzles.

**Bibliography**

Maltby, H., Thanh, K. N., Williams, C., & Khim, J. (n.d.). Graph Coloring and Chromatic Numbers. Retrieved from https://brilliant.org/wiki/graph-coloring-and-chromatic-numbers/

Rioux-Maldague, L. (2014). Graph Coloring Algorithms.

Voloshin, Vitaly. (2009). Graph Coloring: History, results and open problems. Alabama Journal of Mathematics.

**Appendix**

**Source Code for graph.h**

#ifndef GRAPH\_H

#define GRAPH\_H

#include <iostream>

#include <map>

#include <vector>

#include <cmath>

#include <iomanip>

#include <fstream>

#include <chrono>

class Graph{

private:

/\* will map an int to a list of its neighbors \*/

std::map<int, std::vector<int>> vertices; //adjacency list using vector class

std::map<int, int> color; //map that maps two ints, the first int is the vertex and the second is the color

std::vector<int> unknowns; //A vector of all vertices whose coloring is zero

int dimension; //int that is the dimension of the puzzle ie a 9x9

public:

Graph(int n);

void addVertex(int x); // add a vertex to the graph

void addEdge(int x, int y); // add an undirected edge to the graph

void print(); // prints the adjacency list of each vertex, to show the structure

void printBoard(); //Prints the color of each vertex in a Sudoku board format

void connectSquare(int x, int sqr, int n); //

void ColorGreedy(); //A simple greedy algorithm that colors the graph

void ColorBackTrack();

void easyHint(); //precolors the graph to match an easy sudoku puzzle

void intermediateHint(); //precolors the graph to match an intermediate sudoku puzzle

void hardHint(); //precolors the graph to match a hard sudoku puzzle

void addUnknowns(); //Populates the vector unknowns with all the vertices who's color is 0.

void backtrack(int i, int stop);

int numHints(); //returns the number of hints a sudoku puzzle begins with

};

**Graph.cpp**

#include "graph.h"

Graph :: Graph(int n) {

dimension = n;

for(int i = 1; i <=(n\*n); i++) { //adds n^2 vertices to the graph

addVertex(i);

}

int sqr = sqrt(n); //square root of n which is used when determining the size of the squares in a sudoku puzzle

//For loop that finds every upper left corner of each square in a sudoku game

for(int i = 1; i <= (n\*n); i+= (n\*sqr)) { //iterates through rows

for(int j = 0; j < n; j+= sqr) { //iterates through every sqrt(n) columns

connectSquare(i + j, sqr, n); //function to add an edge

}

}

//Connects all the vertices in a single column or vertical line of a sudoku grid, since every individual square was

//already had all its vertices added as neighbors it starts at the lowest square of i.

int skip = 0;

for(int i = 1; i <= n; i++){ //increments to the beginning of the next column

skip = 0;

for(int j = i; j <=(n\*n)-(sqr \*n); j+=n) { //increments down the row until the second to last square is finished

if(j >= skip) { //if the skip variable is too old it is updated

skip = j+(n\*sqr);

}

for(int k = skip; k <=(n\*n); k+=n) { //starts at the square below i and begins going down the rows

addEdge(j,k);

}

}

}

// Connects all the vertices in a horizontal line, since every individual square was already counted as a neighbor it

// finds the vertices in the other squares down the columns

skip = 0; //int that is used to keep track of what square to skip over when adding edges

for(int i = 1; i <= (n\*n); i+=n) { //increments to the beginning of the next row

for(int j = i; j <= i+n-1-sqr; j++) { //increments to next column until the end of the second to last square

if(j%sqr == 1) { //if j has incremented sqr columns edit the skip variable

skip = j+sqr;

}

for(int k = skip; k <= i+n-1; k++) { //finds the start of the vertices to connect at the square that starts at vertex(skip)

addEdge(j,k);

}

}

}

}

void Graph :: connectSquare(int x, int sqr, int n) {

int\* arr = new int[n]; //initialize array that will be used to connect all the edges

int inc = -1; //array increment

for(int i = 0; i < sqr; i++) { //for loop that populates the array with the vertices in the square

for(int j = 0; j < sqr; j++) { //moves across the columns of the square

inc++; //array index incremented

arr[inc] = x+j; //vertex is stored into the temporary array. all vertices inside a sqrt(n) x sqrt(n) sqaure will be added

}

x = x + n; //next row is accessed

}

for(int i = 0; i < n-1; i++) {

for (int j = i+1; j < n; j++) { //inner for loops makes it so the indexices before i are not access again

addEdge(arr[i], arr[j]);

}

}

delete [] arr; //deletion of temporary array

}

void Graph :: addVertex(int x) {

if(vertices.find(x) == vertices.end()) { //Searches the map for vertex x

std::vector<int> neighbors; //if not found initalize an empty vector

vertices.insert({x, neighbors}); //insert x and its empty vector into the map

color.insert({x, 0});

}

else {

std::cout << "Vertex " << x << " is already in the graph." << "\n"<< std::endl; //if already found print statement

}

}

void Graph :: addEdge(int x, int y) {

auto it = vertices.find(x); //iterators that are set to where x and y are in the map if found

auto it2 = vertices.find(y);

bool exist = false; //bool acting as a flag for inserting an edge

if(it != vertices.end() && it2 != vertices.end()) //If statement to check if both vertices are in the graph

{

for(int n : it->second) { //for loop that checks adjacency list of x. if y is found then the edge already exists

if(n == y) { //Only need to check one list since this is an undirected graph

std::cout << "An edge between the vertices, " << x << " and " << y << ", already exists." << "\n" << std::endl;

exist = true; //if they exist the flag is set to true

}

}

if(!exist) { //if bool is false then the edge is added by means of inserting x into y's adjacency list and vice versa.

vertices[x].push\_back(y);

vertices[y].push\_back(x);

}

}

else {

std::cout << "Error. One or both of the requested verticies are not in the graph." << "\n" << std::endl;

}

}

void Graph :: print() { //Prints a vertex, and all of its neighbors

for(auto it = vertices.begin(); it != vertices.end(); it++) { //For loop is set to the first vertex in the map

std::cout << "Neighbors of vertex " << it->first <<std::endl; //it->first points to the key which is the value of the vertex

if(it->second.empty()) //it->second points to the vector or the adjaceny list of all the neighboring vertices

{

std::cout << "\nThis vertex has no neighbors" << std::endl;

}

else {

for(int n : it->second) { //ranged based for loop that iteratates through the vector

std::cout << n << " ";

}

std::cout << "\n" << std::endl;

}

}

}

//Prints the graph in the shape of a Sudoku board

void Graph :: printBoard() { //Prints a vertex, and all of its neighbors

int h = 0; //variables to keep track of position in order to print a new line or vertical line

int v = 0;

int z = 0; //responsible for printing new line after n vertices are printed

int s = sqrt(dimension);

std::cout << std::setfill('-') << std::setw(3\*(dimension+s-1)) << "-" << std::endl;

std::cout << std::setfill(' ');

for(auto it = color.begin(); it != color.end(); it++) { //For loop is set to the first vertex in the map

h++;

z++;

std::cout << std::setw(3) << std::left;

std::cout << it->second;

if(z == dimension) {

v++;

z = 0;

std::cout << std::endl;

h = 0;

}

if(h == s) {

std::cout << std::setw(3) << std::left;

std::cout << "|";

h = 0;

}

if(v == s) {

std::cout << std::setfill('-') << std::setw(3\*(dimension+s-1)) << "-" << std::endl;

std::cout << std::setfill(' ');

v = 0;

}

}

std::cout << std::endl;

}

//Takes the graph and attempts to color a Sudoku solution, //Iterates through each vertex and labels the smallest value of c possibe

//if an adjacent vertex has that color n, c is incremented to the next color and the loop runs again

//Assigns vertices colors from left to right top to bottom in a sudoku board

//Doesnt work for 3x3

//Need to create one that invloves the use of backtracking

void Graph :: ColorGreedy() {

for(auto it = vertices.begin(); it != vertices.end(); it++) {

int c = 1;

int v = it->first;

bool sameColor = true;

if(color[v] == 0) {

while(sameColor != false) {

sameColor = false;

for(int n : it->second) {

if(color[n] == c) {

sameColor = true;

}

}

if(sameColor) {

c++;

}

}

color[it->first] = c;

}

}

}

//Graph coloring algorithm that iterates through a vector that contains every uncolored vertex in the graph

//In each loop two iterators are set to the correspond adjacency list map container vertices and color map color

//int c will designate the starting color that the algorithm will try to color the vertex located at unknowns[i]

//if color[i] is zero, c will be one.

//A while loop runs that stops as soon as c is not the same color as any vertex in the adjacency list

//if c is greater than 9 that means that a coloring couldnt be found for that vertex and the vertex is colored 0 and then

//backtrack is called to go down the vector towards the beginning of the vector and fix any potential coloring errors by trying

//new color values

void Graph :: ColorBackTrack() {

for(int i = 0; i < unknowns.size(); i++) {

auto it = color.find(unknowns[i]); //returns an iterator pointing to the location where color is stored

auto v = vertices.find(unknowns[i]); //points to the current vertex's adjacency list

int c = it->second+1; //current color of unknown at v[i];

bool sameColor = true;

while(sameColor != false) {

sameColor = false;

for(int n : v->second) {

if(color[n] == c) {

sameColor = true;

}

}

if(sameColor) { //if any vertex in the adjacency list has the color c, that color is taken and c is incremented

c++;

}

}

if(c > 9) {

color[unknowns[i]] = 0;

backtrack(i-1, i+1); //if no number fits go back to previous and try next option

}

else {

color[unknowns[i]] = c;

}

}

}

//Similar to ColorBackTrack this function breaks down the vector unknowns into a range of 0 to the int stop.

//If i can not find a valid coloring, the loop iterates up the vector until a coloring is accepted and then the

//vertices down the vector are attempted to be colored again. This stops until i is equal to the stop index which is designated

//above

void Graph :: backtrack(int i, int stop) {

if(i == -1) {

std::cout << "Error: No Solution Found" << std::endl;

}

while(i != stop) {

auto it = color.find(unknowns[i]); //returns an iterator pointing to the location where color is stored

auto v = vertices.find(unknowns[i]); //points to the current vertex's adjacency list

int c = it->second+1; //current color of unknown at v[i];

bool sameColor = true;

while(sameColor != false) {

sameColor = false;

for(int n : v->second) {

if(color[n] == c) {

sameColor = true;

}

}

if(sameColor) { //if any vertex in the adjacency list has the color c, that color is taken and c is incremented

c++;

}

}

if(c > 9) { //if a coloring wasn't found iterate up the vector towards the start

color[unknowns[i]] = 0;

i--;

}

else { //if a coloring was found move down the vector towards the stopping point

color[unknowns[i]] = c;

i++;

}

}

}

//Reads in from hint3.txt which contains an easy sudoku puzzle.

//Puzzle was found on https://dingo.sbs.arizona.edu

void Graph :: easyHint() {

std::ifstream inFile;

inFile.open("hint.txt");

for(int i = 1; i <= 81; i++) {

inFile >> color[i];

}

inFile.close();

addUnknowns();

}

//Reads in from hint3.txt which contains an Intermediate sudoku puzzle.

//Puzzle was found on https://dingo.sbs.arizona.edu

void Graph :: intermediateHint() {

std::ifstream inFile;

inFile.open("hint2.txt");

for(int i = 1; i <= 81; i++) {

inFile >> color[i];

}

inFile.close();

addUnknowns();

}

//Reads in from hint3.txt which contains a very difficult sudoku puzzle.

//Puzzle was found on https://dingo.sbs.arizona.edu

void Graph :: hardHint() {

std::ifstream inFile;

inFile.open("hint3.txt");

for(int i = 1; i <= 81; i++) {

inFile >> color[i];

}

inFile.close();

addUnknowns();

}

//Returns the number of hints each sudoku puzzle has.

int Graph :: numHints() {

return 81-unknowns.size();

}

//Creates a vector that holds all the vertices that are uncolored.

//This vector is used to find solutions to the sudoku puzzle given without altering the vertices already colored

void Graph :: addUnknowns() {

for(auto it = color.begin(); it != color.end(); it++) {

if(it->second == 0) { //every vertex uncolored is added to the vector

unknowns.push\_back(it->first);

}

}

}

#endif

**projectDriver.cpp**

#include "graph.h"

int main() {

std::cout << "===========GRAPH\_COLORING SUDOKU APPLICATION=================" << std::endl;

std::cout << "Simple Greedy Algorithm" << std::endl;

Graph simple(9);

simple.easyHint();

std::cout << "Number of Hints: " << simple.numHints() << std::endl;

std::cout << "\nEasy Difficulty Sudoku Board" << std::endl;

simple.printBoard();

// auto start = std::chrono::system\_clock::now();

simple.ColorGreedy();

// auto end = std::chrono::system\_clock::now();

//std::chrono::duration<double> elasped\_seconds = end-start;

std::cout << "Solution" << std::endl;

simple.printBoard();

std::cout << "Greedy Algorithm with Backtracking" << std::endl;

Graph sudoku(9);

sudoku.easyHint();

std::cout << "Number of Hints: " << sudoku.numHints() << std::endl;

sudoku.printBoard();

auto start = std::chrono::system\_clock::now();

sudoku.ColorBackTrack();

auto end = std::chrono::system\_clock::now();

std::chrono::duration<double> elasped\_seconds = end-start;

std::cout << "Solution" << std::endl;

sudoku.printBoard();

std::cout << "Solution found in " << std::fixed << elasped\_seconds.count() << " seconds\n" << std::endl;

std::cout << "Intermediate Difficulty Sudoku Board" << std::endl;

Graph sudoku2(9);

sudoku2.intermediateHint();

std::cout << "Number of Hints: " << sudoku2.numHints() << std::endl;

sudoku2.printBoard();

start = std::chrono::system\_clock::now();

sudoku2.ColorBackTrack();

end = std::chrono::system\_clock::now();

elasped\_seconds = end-start;

std::cout << "Solution" << std::endl;

sudoku2.printBoard();

std::cout << "Solution found in " << std::fixed << elasped\_seconds.count() << " seconds\n" << std::endl;

std::cout << "Hard Difficulty Sudoku Board" << std::endl;

Graph sudoku3(9);

sudoku3.hardHint();

std::cout << "Number of Hints: " << sudoku3.numHints() << std::endl;

sudoku3.printBoard();

start = std::chrono::system\_clock::now();

sudoku3.ColorBackTrack();

end = std::chrono::system\_clock::now();

elasped\_seconds = end-start;

std::cout << "Solution" << std::endl;

sudoku3.printBoard();

std::cout << "Solution found in " << std::fixed << elasped\_seconds.count() << " seconds\n" << std::endl;

}