**EXPERIMENT 4 - Constraint Satisfaction - Graph/Map Colouring**

**BACKTRACKING CODE**

#include <stdbool.h>

#include <stdio.h>

// Number of vertices in the graph

#define V 4

void printSolution(int color[]);

/\* A utility function to check if

the current color assignment

is safe for vertex v i.e. checks

whether the edge exists or not

(i.e, graph[v][i]==1). If exist

then checks whether the color to

be filled in the new vertex(c is

sent in the parameter) is already

used by its adjacent

vertices(i-->adj vertices) or

not (i.e, color[i]==c) \*/

bool isSafe(

int v, bool graph[V][V],

int color[], int c)

{

for (int i = 0; i < V; i++)

if (

graph[v][i] && c == color[i])

return false;

return true;

}

/\* A recursive utility function

to solve m coloring problem \*/

bool graphColoringUtil(

bool graph[V][V], int m,

int color[], int v)

{

/\* base case: If all vertices are

assigned a color then return true \*/

if (v == V)

return true;

/\* Consider this vertex v and

try different colors \*/

for (int c = 1; c <= m; c++) {

/\* Check if assignment of color

c to v is fine\*/

if (isSafe(

v, graph, color, c)) {

color[v] = c;

/\* recur to assign colors to

rest of the vertices \*/

if (

graphColoringUtil(

graph, m, color, v + 1)

== true)

return true;

/\* If assigning color c doesn't

lead to a solution then remove it \*/

color[v] = 0;

}

}

**OUTPUT:**

Solution Exists: Following are the assigned colors

1 2 3 2

**FORWARD CHECKING CODE:**

#include<iostream>

#include<vector>

#include<map>

#include<string>

#define WITHOUT\_COLOR "NoColor"

using namespace std;

int c = 0;

int maxstep;

//Generates random number within 0 and given limit

int getRandom(int limit) {

return rand() % limit;

}

class Region {

public:

int number;

string color;

vector<int> neighbourRegion;

map<string, bool> possibleColors;

//Initialization of Forward Checking

void initFC(int i) {

number = i + 1;

color = WITHOUT\_COLOR; // The color assigned to the region, initially colorless

//Initially all colors are available for this region

possibleColors["Blue"] = true;

possibleColors["Yellow"] = true;

possibleColors["Green"] = true;

possibleColors["Red"] = true;

}

//Initialization of Min-Conflict Algo

void initMinConflict(int i) {

number = i + 1;

gencolor(getRandom(4));

}

//Assigns color to the region

void gencolor(int cnum) {

if (cnum == 0)

color = "Blue";

else if (cnum == 1)

color = "Yellow";

else if (cnum == 2)

color = "Green";

else if (cnum == 3)

color = "Red";

}

//Assigns neighbours from the generated map.

void genNeighbour(int i, int size, int\*\* Map) {

for (int j = 0; j < size; j++) {

if (Map[i][j] == 1) {

neighbourRegion.push\_back(j + 1);

}

}

}

//Returns the Region-number of neighbours

vector<int> getNeighbors() {

return neighbourRegion;

}

//Checks if the region has color or not

bool hasColor() {

if (color == WITHOUT\_COLOR)

return false;

return true;

}

//Updates colors of neighbour region in MRV

void updatePossibleColors(bool flag, string colorName) {

possibleColors[colorName] = flag;

}

//Returns to quantity of available cores for this region

int getPossibleColors() {

int cont = 0;

for (map<string, bool>::iterator it = possibleColors.begin(); it != possibleColors.end(); ++it) {

if (it->second)

cont += 1;

}

return cont;

}

//Returns to name of available colors for this region

vector<string> getPossibleColorsname() {

vector<string> col;

for (map<string, bool>::iterator it = possibleColors.begin(); it != possibleColors.end(); ++it) {

if (it->second)

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

}

return col;

}

};

class Map {

public:

vector<Region> regionList;

vector<string> colors;

//Initialization of colors

void MapcolorInt() {

colors.push\_back("Blue");

colors.push\_back("Yellow");

colors.push\_back("Green");

colors.push\_back("Red");

}

void printSolution() {

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

cout << regionList.at(i).number << " will have " << regionList.at(i).color << "\n";

}

}

//Chooses the region with the fewest possible colors

int selectVariableMVR() {

int mrv = colors.size() + 1;

int variable = 0;

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

if (!regionList[i].hasColor()) {

if (regionList[i].getPossibleColors() < mrv) {

variable = i;

mrv = regionList[i].getPossibleColors();

}

}

}

return variable;

}

void ForwardChecking() {

for (int j = 0; j < regionList.size(); j++) {

//Find the region which has MINIMUM REMAINING VALUES

int mvr = selectVariableMVR();

//Get the possible color names for that region

vector<string> possColorsList = regionList[mvr].getPossibleColorsname();

//assign one of the possible colors to the region

regionList.at(mvr).color = possColorsList[0];

//Find the neighbours of the region to whom the color is just assigned.

vector<int> neighbor = regionList.at(mvr).getNeighbors();

//Remove the assigned color from the possible colors of its neighbours

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

if (!regionList[neighbor[i] - 1].hasColor()) {

regionList[neighbor[i] - 1].updatePossibleColors(false, regionList.at(mvr).color);

}

}

}

//Once all the regions are colored then the solution will be printed

cout << "\n\nsolution for Forward checking with MRV: \n";

printSolution();

}

};

//To print the map

void printMap(int\*\* M, int n) {

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

for (int j = 0; j< n; j++) {

cout << M[i][j] << " ";

}

cout << "\n";

}

}

//Create map function generates a practically feasible map.

int\*\* createMap(int n) {

int\*\* newMap = new int\*[n];

for (int i = 0; i < n; ++i)

newMap[i] = new int[n];

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

for (int j = 0; j < n; j++) {

if (i == j)newMap[i][j] = 0;

else

{

newMap[i][j] = -1;

}

}

}

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

for (int j = 0; j < n; j++) {

if (newMap[i][j] == -1) {

if (j == i - 4 || j == i - 3 || j == i - 2 || j == i - 1 || j == i + 1 || j == i + 2 || j == i + 3 || j == i + 4) {

newMap[i][j] = getRandom(2);

newMap[j][i] = newMap[i][j];

}

else

{

newMap[i][j] = 0;

}

}

}

}

printMap(newMap, n);

return newMap;

}

int main()

{

int n;

cout << "Enter number of region for a map: ";

cin >> n;

int\*\* M = new int\*[n];

M = createMap(n);

//Making 2 different regions because FC and Min-conflict algorithm starts with different initiallizations.

vector<Region> Region\_fc(n);

//initialization for the regions in FC includes giving numbers to the regions, giving "NoColor" to the present color of each region

//making possibility of all color=1 and put the values of neighbors from the map.

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

Region\_fc[i].initFC(i);

Region\_fc[i].genNeighbour(i, n, M);

}

Map map\_fc;

map\_fc.regionList = Region\_fc;

//Initialize colors with red, blue, green, yellow

map\_fc.MapcolorInt();

//Forward checking function of Map class actually assigns the values to all the regions.

map\_fc.ForwardChecking();

getchar();

return 0;

}

**OUTPUT:**

Enter number of region for a map: 4

0 1 0 1

1 0 1 1

0 1 0 1

1 1 1 0

solution for Forward checking with MRV:

1 will have Blue

2 will have Green

3 will have Blue

4 will have Red

|  |
| --- |
| **GREEDY CODE:**  #include <iostream>  #include <vector>  #include <unordered\_map>  #include <set>  using namespace std;    // Data structure to store a graph edge  struct Edge {  int src, dest;  };    class Graph  {  public:  // a vector of vectors to represent an adjacency list  vector<vector<int>> adjList;    // Constructor  Graph(vector<Edge> const &edges, int N)  {  // resize the vector to hold `N` elements of type `vector<int>`  adjList.resize(N);    // add edges to the undirected graph  for (Edge edge: edges)  {  int src = edge.src;  int dest = edge.dest;    adjList[src].push\_back(dest);  adjList[dest].push\_back(src);  }  }  };    // Add more colors for graphs with many more vertices  string color[] =  {  "", "BLUE", "GREEN", "RED", "YELLOW", "ORANGE", "PINK",  "BLACK", "BROWN", "WHITE", "PURPLE", "VOILET"  };    // Function to assign colors to vertices of a graph  void colorGraph(Graph const &graph, int N)  {  // keep track of the color assigned to each vertex  unordered\_map<int, int> result;    // assign a color to vertex one by one  for (int u = 0; u < N; u++)  {  // set to store the color of adjacent vertices of `u`  set<int> assigned;    // check colors of adjacent vertices of `u` and store them in a set  for (int i: graph.adjList[u])  {  if (result[i]) {  assigned.insert(result[i]);  }  }    // check for the first free color  int color = 1;  for (auto &c: assigned )  {  if (color != c) {  break;  }  color++;  }    // assign vertex `u` the first available color  result[u] = color;  }    for (int v = 0; v < N; v++)  {  cout << "The color assigned to vertex " << v << " is "  << color[result[v]] << endl;  }  }    // Greedy coloring of a graph  int main()  {  // vector of graph edges as per the above diagram  vector<Edge> edges = {  {0, 1}, {0, 4}, {0, 5}, {4, 5}, {1, 4}, {1, 3}, {2, 3}, {2, 4}  };    // total number of nodes in the graph  int N = 6;    // build a graph from the given edges  Graph graph(edges, N);    // color graph using the greedy algorithm  colorGraph(graph, N);    return 0;  } |

**OUTPUT:**

The color assigned to vertex 0 is BLUE

The color assigned to vertex 1 is GREEN

The color assigned to vertex 2 is BLUE

The color assigned to vertex 3 is RED

The color assigned to vertex 4 is RED

The color assigned to vertex 5 is GREEN

**GENETIC ALGORITHM CODE :**

|  |  |
| --- | --- |
|  | import random as rnd |
|  | import copy |
|  | import matplotlib.pyplot as plt |
|  | from itertools import product |
|  |  |
|  | class Genetic: |
|  | def \_\_init\_\_(self, graph, num\_edges, population\_size, num\_colors): |
|  | """ |
|  | This constructor inits a model for the graph in order to solve it using genetic algorithm |
|  | """ |
|  | self.graph = graph |
|  | self.size = population\_size |
|  | self.length = len(graph) |
|  | self.edges = num\_edges |
|  | self.num\_colors = num\_colors |
|  | self.population = [[rnd.randint(1, num\_colors) for i in range(self.length)] for i in range(population\_size)] |
|  |  |
|  | def fitness(self, chrom): |
|  | """ |
|  | This function computes the fitness value for a given chromosome. |
|  | """ |
|  | f = 0.0 |
|  | for node, neighbors in enumerate(self.graph): |
|  | for nei in neighbors: |
|  | if self.population[chrom][node] != self.population[chrom][nei]: |
|  | f += 1 |
|  | f /= (2 \* self.edges) |
|  | return f |
|  |  |
|  | def selection(self, k): |
|  | """ |
|  | This function takes k member and selects the best one according to their fitness value. |
|  | The number of the tornuments depends on the population size and k and equals size/k |
|  | """ |
|  | parents = [] |
|  | torns = self.size // k |
|  | for i in range(torns): |
|  | candids = [rnd.randint(0, self.size - 1) for j in range(k)] |
|  | chosen = None |
|  | max\_fitness = -1.0 |
|  | for j in candids: |
|  | fit = self.fitness(j) |
|  | if fit > max\_fitness: |
|  | max\_fitness = fit |
|  | chosen = j |
|  | parents.append(chosen) |
|  | return parents |
|  |  |
|  | def new\_generation(self, parents): |
|  | """ |
|  | This function takes the selected parents and creates a number of new chromosomes. |
|  | The number of children equals the size of the original population. |
|  | Inputs |
|  | - parents: a list containing the index of the parent chromosomes |
|  | """ |
|  | children = [] |
|  | for i in range(self.size): |
|  | x = rnd.randint(0, len(parents) - 1) |
|  | y = 0 |
|  | while True: |
|  | y = rnd.randint(0, len(parents) - 1) |
|  | if x != y: break |
|  | children.append(self.crossover(self.population[parents[x]], self.population[parents[y]])) |
|  | self.population = children |
|  |  |
|  | def crossover(self, x, y): |
|  | """ |
|  | This function takes two parents and creates a child from them. |
|  | Inputs |
|  | - x: index of the first chromosome |
|  | - y: index of the second chromosome |
|  | """ |
|  | child = copy.deepcopy(x) |
|  | mask = [rnd.uniform(0, 1) for i in range(self.length)] |
|  | for i in range(self.length): |
|  | if mask[i] > 0.5: |
|  | child[i] = y[i] |
|  | return child |
|  |  |
|  | def mutation(self, mutation\_rate): |
|  | """ |
|  | This function mutates some of the genes in some of the chromosomes. |
|  | The number of genes to be mutated is computated with the following formula: (populationSize \* chromosomeLength \* mutaionRate) |
|  | """ |
|  | mutated\_genes = int(self.size \* self.length \* mutation\_rate) |
|  | if mutated\_genes <= 0: |
|  | return |
|  | options = product(range(self.size), range(self.length)) |
|  | options = rnd.sample(list(options), mutated\_genes) |
|  | for i in range(mutated\_genes): |
|  | self.population[options[i][0]][options[i][1]] = rnd.randint(1, self.num\_colors) |
|  |  |
|  | def exec(self, num\_iters, k, mutation\_rate): |
|  | """ |
|  | This part executes the algorithm for num\_iters iterations. |
|  | """ |
|  | high\_hist = [] |
|  | low\_hist = [] |
|  | middle\_hist = [] |
|  | for i in range(num\_iters): |
|  | high, low, middle = self.stats() |
|  | high\_hist.append(high) |
|  | low\_hist.append(low) |
|  | middle\_hist.append(middle) |
|  | parents = self.selection(k) |
|  | self.new\_generation(parents) |
|  | self.mutation(mutation\_rate) |
|  | return high\_hist, low\_hist, middle\_hist |
|  |  |
|  | def stats(self): |
|  | high, low, ave = -1, -1, 0.0 |
|  | for i, chrom in enumerate(self.population): |
|  | fit = self.fitness(i) |
|  | ave += fit |
|  | if fit > high: |
|  | high = fit |
|  | if fit < low or low == -1: |
|  | low = fit |
|  | return high, low, (ave/self.size) |
|  |  |
|  | def get\_graph(): |
|  | ver = int(input('enter number of vertices: ')) |
|  | graph = [] |
|  | edges = 0 |
|  | for i in range(ver): |
|  | temp\_list = [] |
|  | face = input() |
|  | face = face.split(' ') |
|  | for j in face: |
|  | temp\_list.append(int(j)) |
|  | edges += 1 |
|  | graph.append(temp\_list) |
|  |  |
|  | return graph, ver, edges // 2 |
|  |  |
|  | def show\_data(high, low, middle, title): |
|  | plt.figure() |
|  | plt.title(title) |
|  | plt.plot(high, label='Maximum') |
|  | plt.plot(low, label='Minimum') |
|  | plt.plot(middle, label='Average') |
|  | plt.legend() |
|  | plt.savefig('C:\\Users\\Mohammad\\Desktop\\figures\\' + title + '.png') |
|  | print(title, 'saved.') |
|  |  |
|  | def main(): |
|  |  |
|  | # params |
|  | # generations = [50, 500, 5000] |
|  | # mutation\_rates = [0.01, 0.02, 0.05, 0.1] |
|  | # population\_sizes = [10, 100, 1000] |
|  | # ks = [2, 5, 10] |
|  | # colors = 4 |
|  |  |
|  | # graph, N, M = get\_graph() |
|  |  |
|  | # for gen in generations: |
|  | # for mut in mutation\_rates: |
|  | # for p in population\_sizes: |
|  | # for k in ks: |
|  | # if p == 10 and k > 2: |
|  | # continue |
|  | # genet = Genetic(graph, M, p, colors) |
|  | # data = genet.exec(gen, k, mut) |
|  | # title = 'generations=%d, population=%d, mutation rate=%.2f, k=%d' % (gen, p, mut, k) |
|  | # show\_data(\*data, title) |
|  |  |
|  | # params |
|  | generations = 50 |
|  | mutation\_rate = 0.01 |
|  | population\_size = 100 |
|  | k = 5 |
|  | colors = 4 |
|  |  |
|  | graph, N, M = get\_graph() |
|  | genet = Genetic(graph, M, population\_size, colors) |
|  | genet.exec(generations, k, mutation\_rate) |
|  |  |
|  | high = -1 |
|  | best = None |
|  | for i, chrom in enumerate(genet.population): |
|  | fit = genet.fitness(i) |
|  | if fit > high: |
|  | high = fit |
|  | best = chrom |
|  |  |
|  | print() |
|  | print('best coloring') |
|  | print(best) |
|  |  |
|  | if \_\_name\_\_ == '\_\_main\_\_': |
|  | main() |

**SIMULATED ANNEALING ALGORITHM CODE :**

|  |  |
| --- | --- |
|  | import random as rnd |
|  | import copy |
|  | import math |
|  | import matplotlib.pyplot as plt |
|  | from genetic import get\_graph |
|  |  |
|  | class SimulatedAnealing: |
|  | """ |
|  | This class helps to use simulated anealing algorithm to solve map coloring. |
|  | To use this we fist have to initialize it using the constructor. |
|  | Then we can call exec method witch executes the algorithm for the given number of iterations. |
|  | """ |
|  | def \_\_init\_\_(self, graph, num\_edges, num\_colors, schedule, T0, alpha): |
|  | self.length = len(graph) |
|  | self.state = [rnd.randint(1, num\_colors) for i in range(self.length)] |
|  | self.edges = num\_edges |
|  | self.colors = num\_colors |
|  | self.graph = graph |
|  | self.schedule = schedule |
|  | self.T0 = T0 \* 1.0 |
|  | self.alpha = alpha \* 1.0 |
|  |  |
|  | def fitness(self, chrom): |
|  | """ |
|  | This function computes the fitness value for a given chromosome. |
|  | """ |
|  | f = 0.0 |
|  | for node, neighbors in enumerate(self.graph): |
|  | for nei in neighbors: |
|  | if chrom[node] != chrom[nei]: |
|  | f += 1 |
|  | f /= (2 \* self.edges) |
|  | return f |
|  |  |
|  | def exec(self, count): |
|  | """ |
|  | For "count" times we take new successors and choose to go there or not depending |
|  | on their fitness value. |
|  | """ |
|  | hist\_t = [] |
|  | hist\_fit = [] |
|  | for t in range(count): |
|  | T = self.schedule(self.T0, self.alpha, t) |
|  | hist\_t.append(T) |
|  | hist\_fit.append(self.fitness(self.state)) |
|  | if T == 0: |
|  | return self.state |
|  |  |
|  | successor = copy.deepcopy(self.state) |
|  | rand = rnd.randint(0, self.length - 1) |
|  | new\_color = rnd.randint(1, self.colors) |
|  | while new\_color == self.state[rand]: |
|  | new\_color = rnd.randint(1, self.colors) |
|  | successor[rand] = new\_color |
|  |  |
|  | e1 = self.fitness(self.state) |
|  | e2 = self.fitness(successor) |
|  | delta\_e = (e2 - e1) \* 1.0 |
|  |  |
|  | if delta\_e >= 0: |
|  | self.state = successor |
|  | else: |
|  | prob = math.exp(delta\_e / T) |
|  | unif = rnd.uniform(0, 1) |
|  | if unif <= prob: |
|  | self.state = successor |
|  |  |
|  | return self.state, hist\_t, hist\_fit |
|  |  |
|  | def mapping\_1(T0, alpha, t): |
|  | T = T0 \* math.pow(alpha, t) |
|  | return T |
|  |  |
|  | def mapping\_2(T0, alpha, t): |
|  | T = (T0) / (1 + alpha \* math.log(1 + t)) |
|  | return T |
|  |  |
|  | def mapping\_3(T0, alpha, t): |
|  | T = (T0) / (1 + alpha \* t) |
|  | return T |
|  |  |
|  | def mapping\_4(T0, alpha, t): |
|  | T = (T0) / (1 + alpha \* t \* t) |
|  | return T |
|  |  |
|  | def main(): |
|  | # params |
|  | colors = 4 |
|  | iters = 10000 |
|  | T0 = 500 |
|  | alpha = 40 |
|  |  |
|  | graph, N, M = get\_graph() |
|  | sa = SimulatedAnealing(graph, M, colors, mapping\_2, T0, alpha) |
|  | state, t, fit = sa.exec(iters) |
|  | print(state) |
|  |  |
|  | fig, (ax1, ax2) = plt.subplots(2, 1) |
|  | ax1.plot(fit) |
|  | ax2.plot(t) |
|  | plt.show() |
|  |  |
|  | if \_\_name\_\_ == '\_\_main\_\_': |
|  | main() |

**Presentation Link:**

[**https://docs.google.com/presentation/d/180Gp3uGrkhOLcwfu0ThubMd\_0zQzvi-huE3NxJ0kdvE/edit?usp=sharing**](https://docs.google.com/presentation/d/180Gp3uGrkhOLcwfu0ThubMd_0zQzvi-huE3NxJ0kdvE/edit?usp=sharing)

**Presented by Team Elite**