## CMPT307 Assignmnet6

Yuhan He 301385291

1. Is-Semiconnected(G)

C = Strongly-Connected-Components(G)

for i = 1 to s(C) do

int num = 0;

for j = 1 to s(C) do

if i is not equal to j then

if there is no edge between Ci and Cj then num = num + 1;

if num is equal to s(C)-1 then return false;

return true;

Correctness: C is the set of strongly connected components of G. If G is semiconnected graph, there is should at least one edge between Ci and other strongly connected components. So, we use num to count how many strongly connected components has no edge with Ci. If num equal to s(C)-1, it means that G is not semiconnected graph.

T(n) = 1 + n + n2 + n = O(n2) = O(V+E)

1. If any subset of edges that connects all vertices is not a tree, it means that there are cycles in the subset. To make it become a tree, we can always remove an edge. So, any subset of edges that connects all vertices must be tree.

If the weight could be negative, the cycle could have a weight smaller than path.

3

3

-7

3

3

1. Code:

C++:

List:

#pragma once

#include<iostream>

#include<chrono>

#include<time.h>

#include<queue>

#include<list>

#include<map>

using namespace std::chrono;

using namespace std;

class List\_Graph {

private:

map<int, list<int>> graph;

int\* vertex;

map<pair<int, int>, int> edge;

int size;

public:

List\_Graph();

List\_Graph(int\*, map<pair<int, int>, int>, int);

map<int, list<int>> getGraph();

int\* getVertex();

map<pair<int, int>, int> getEdge();

int getSize();

bool existVertex(int);

bool existEdge(int, int);

void addVertex(int);

void addEdge(int, int, int);

};

struct Node {

int v;

int p;

int d;

bool operator < (Node n) {

return this->d < n.d;

}

};

typedef Node\* nodeptr;

List\_Graph::List\_Graph() {

this->vertex = NULL;

this->size = 0;

map<int, list<int>> g;

map<pair<int, int>, int> e;

this->edge = e;

this->graph = g;

}

List\_Graph::List\_Graph(int\* v, map<pair<int, int>, int> e, int s) {

this->vertex = v;

this->edge = e;

this->size = s;

/\*for (auto p : e) {

pair<int, int> temp = p.first;

int w = p.second;

pair<int, int> t;

t = make\_pair(temp.second, temp.first);

if (this->edge.count(t) == 0) {

this->edge.insert(pair<pair<int, int>, int>(t, w));

}

}\*/

for (int i = 0; i < this->size; i++) {

list<int> l;

this->graph.insert(pair<int, list<int>>(v[i], l));

}

for (auto p : e) {

pair<int, int> temp = p.first;

map<int, list<int>>::iterator iter;

iter = this->graph.find(temp.first);

iter->second.push\_back(temp.second);

iter = this->graph.find(temp.second);

iter->second.push\_back(temp.first);

}

}

map<int, list<int>> List\_Graph::getGraph() {

return this->graph;

}

int\* List\_Graph::getVertex() {

return this->vertex;

}

map<pair<int, int>, int> List\_Graph::getEdge() {

return this->edge;

}

int List\_Graph::getSize() {

return this->size;

}

bool List\_Graph::existVertex(int v) {

for (int i = 0; i < this->size; i++) {

if (this->vertex[i] == v)

return true;

}

return false;

}

bool List\_Graph::existEdge(int v, int u) {

pair<int, int> e1;

pair<int, int> e2;

e1 = make\_pair(v, u);

e2 = make\_pair(u, v);

if (this->edge.count(e1) == 1 || this->edge.count(e2) == 1)

return true;

else

return false;

}

void List\_Graph::addVertex(int v) {

if (!existVertex(v)) {

this->vertex += v;

this->size++;

list<int> l;

this->graph.insert(pair<int, list<int>>(v, l));

}

}

void List\_Graph::addEdge(int v, int u, int w) {

if (!existEdge(v, u)) {

pair<int, int> e;

e = make\_pair(v, u);

this->edge.insert(pair<pair<int, int>, int>(e, w));

if (!existVertex(v)) {

int\* temp = new int[this->size + 1];

for (int i = 0; i < this->size; i++)

temp[i] = this->vertex[i];

temp[this->size] = v;

this->vertex = temp;

this->size++;

list<int> l;

this->graph.insert(pair<int, list<int>>(v, l));

}

if (!existVertex(u)) {

int\* temp = new int[this->size + 1];

for (int i = 0; i < this->size; i++)

temp[i] = this->vertex[i];

temp[this->size] = u;

this->vertex = temp;

this->size++;

list<int> l;

this->graph.insert(pair<int, list<int>>(u, l));

}

map<int, list<int>>::iterator iter;

iter = this->graph.find(v);

iter->second.push\_back(u);

iter = this->graph.find(u);

iter->second.push\_back(v);

}

}

int\* Join(int\* S, int\* V, int s1, int s2) {

int\* join = new int[s2 - s1];

int s = 0;

for (int i = 0; i < s2; i++) {

int n = 0;

for (int j = 0; j < s1; j++) {

if (S[j] == V[i]) {

n = 1;

break;

}

}

if (n == 0) {

join[s] = V[i];

s++;

}

}

return join;

}

void Graph\_Heapify(int\* W, pair<int, int>\* edge, int i, int n) {

int l = i \* 2 + 1, r = i \* 2 + 2;

int min = 0;

if (l < n && W[l] < W[i])

min = l;

else

min = i;

if (r < n && W[r] < W[min])

min = r;

if (min != i) {

swap(W[min], W[i]);

swap(edge[min], edge[i]);

Graph\_Heapify(W, edge, min, n);

}

}

void Build\_Graph\_Heap(int\* W, pair<int, int>\* edge, int n) {

for (int i = (n - 1) / 2; i >= 0; i--)

Graph\_Heapify(W, edge, i, n);

}

void Heap\_Decrease\_key(int\* W, pair<int, int>\* edge, int i, int w, pair<int, int> e) {

if (w > W[i])

return;

else {

W[i] = w;

edge[i] = e;

while (i > 1 && W[(i + 1) / 2 - 1] > W[i]) {

swap(W[(i + 1) / 2 - 1], W[i]);

swap(edge[(i + 1) / 2 - 1], edge[i]);

i = (i + 1) / 2 - 1;

}

}

}

pair<int, int> bestEdge(List\_Graph graph, int\* S, int\* V, int s) {

/\*int size = 0;

int\* w = new int[size];

pair<int, int>\* P = new pair<int, int>[size];\*/

int w = 10000;

pair<int, int> P;

for (auto p : graph.getEdge()) {

int a = p.second;

pair<int, int> b = p.first;

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

if (b.first == S[i]) {

for (int j = 0; j < graph.getSize() - s; j++) {

if (b.second == V[j]) {

if (w > a) {

w = a;

P = pair<int, int>(b.first, b.second);

}

}

}

}

else {

if (b.second == S[i]) {

for (int j = 0; j < graph.getSize() - s; j++) {

if (b.first == V[j]) {

if (w > a) {

w = a;

P = pair<int, int>(b.second, b.first);

}

}

}

}

}

}

}

return P;

}

pair<int, int>\* MST\_Prim(List\_Graph graph, int r) {

pair<int, int>\* A = new pair<int, int>[graph.getSize() - 1];

int size = 1;

int\* S = new int[size];

S[0] = r;

int i = 0;

while (size != graph.getSize()) {

int\* temp = Join(S, graph.getVertex(), size, graph.getSize());

pair<int, int> P = bestEdge(graph, S, temp, size);

A[i] = P;

int a = P.second;

temp = new int[size + 1];

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

temp[j] = S[j];

temp[size] = a;

S = temp;

size++;

i++;

}

return A;

}

Matrix:

#define MAX 1000000

#define SIZE 1000

class Graph

{

private:

int\* vertices;

int\*\* edges;

int size;

int NumVertices;

int NumEdges;

public:

Graph();

Graph(int\*, int\*\*, int, int);

int getVertexIndex(int);

int getNumVertices();

int\* getVertices();

int\*\* getEdge();

void addVertex(int\* label, int size);

void removeVertex(int label);

void addEdge(int label1, int label2, int w);

void removeEdge(int label1, int label2, int w);

void print();

};

Graph::Graph()

{

size = SIZE;

NumVertices = NumEdges = 0;

vertices = NULL;

edges = new int\* [size];

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

edges[i] = new int[size];

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

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

edges[i][j] = -1;

}

Graph::Graph(int\* v, int\*\* e, int nv, int ne)

{

vertices = v;

edges = e;

size = SIZE;

NumVertices = nv;

NumEdges = ne;

}

int Graph::getVertexIndex(int v)

{

int i = 0;

for (i; i < NumVertices; i++)

{

if (vertices[i] == v)

return i;

}

return -1;

}

int Graph::getNumVertices() {

return this->NumVertices;

}

int\* Graph::getVertices() {

return this->vertices;

}

int\*\* Graph::getEdge() {

return this->edges;

}

void Graph::addVertex(int\* label, int size)

{

this->vertices = label;

this->NumVertices = size;

}

void Graph::removeVertex(int label)

{

int p = getVertexIndex(label);

if (p == -1)

return;

for (int i = p; i < NumVertices - 1; i++)

vertices[i] = vertices[i + 1];

int n = 0;

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

{

if (edges[p][i] > 0)

n++;

}

for (int i = p; i < NumVertices - 1; i++)

for (int j = 0; j < NumVertices; j++)

edges[i][j] = edges[i + 1][j];

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

for (int j = p; j < NumVertices - 1; j++)

edges[i][j] = edges[i][j + 1];

NumVertices--;

NumEdges -= n;

}

void Graph::addEdge(int label1, int label2, int w)

{

int p1 = getVertexIndex(label1);

int p2 = getVertexIndex(label2);

if (p1 == -1 || p2 == -1)

return;

edges[p1][p2] = edges[p2][p1] = w;

NumEdges++;

}

void Graph::removeEdge(int label1, int label2, int w)

{

int p1 = getVertexIndex(label1);

int p2 = getVertexIndex(label2);

if (p1 == -1 || p2 == -1)

return;

if (edges[p1][p2] == -1)

return;

edges[p1][p2] = edges[p2][p1] = -1;

NumEdges--;

}

void Graph::print()

{

cout << "\t";

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

cout << vertices[i] << "\t";

cout << endl;

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

{

cout << vertices[i] << "\t";

for (int j = 0; j < NumVertices; j++)

{

if (edges[i][j] != -1)

cout << edges[i][j] << "\t";

else

cout << "n" << "\t";

}

cout << endl;

cout << endl;

}

}

pair<int, int> bestEdge2(Graph g, int\* S, int\* V, int size) {

int w = 10000;

pair<int, int> P;

int\*\* temp = g.getEdge();

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

for (int j = 0; j < g.getNumVertices() - size; j++) {

int k = g.getVertexIndex(S[i]);

int l = g.getVertexIndex(V[j]);

pair<int, int> d;

d = make\_pair(S[i], V[j]);

if (g.getEdge()[k][l] > -1) {

if (w > g.getEdge()[k][l]) {

w = g.getEdge()[k][l];

P = d;

}

}

}

}

return P;

}

pair<int, int>\* MST\_Prim(Graph graph, int r) {

pair<int, int>\* A = new pair<int, int>[graph.getNumVertices() - 1];

int\* S = new int[1];

int size = 1;

S[0] = r;

int i = 0;

while (size != graph.getNumVertices()) {

int\* temp = Join(S, graph.getVertices(), size, graph.getNumVertices());

pair<int, int> P = bestEdge2(graph, S, temp, size);

A[i] = P;

int a = P.second;

temp = new int[size + 1];

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

temp[j] = S[j];

temp[size] = a;

S = temp;

size++;

i++;

}

return A;

}

|  |  |  |  |
| --- | --- | --- | --- |
| node | Edge | Time for list graph | Time for matrix graph |
| 100 | 300 | 230.06ms | 86.46ms |
| 100 | 1000 | 543.27ms | 80.35ms |
| 100 | 4950 | 1706.91ms | 91.34ms |
| 200 | 600 | 1055.29ms | 953.04ms |
| 200 | 2828 | 2624.99ms | 909.90ms |
| 200 | 19900 | 14208.68ms | 970.51ms |
| 400 | 1200 | 4671.52ms | 11429.63ms |
| 400 | 8000 | 13016.64ms | 11215.49ms |
| 400 | 79800 | 137278.54ms | 11320.06ms |
| 800 | 2400 | 25415.47ms | 124683.45ms |
| 800 | 22627 | 75974.46ms | 133697.30ms |
| 800 | 319600 | 2499564.88ms | 129132.54ms |

1. Structure:

The minimum distance between s and f should be minimum sum of weight of edge of adjacent node and the distance from the node and f.

Bellman equation:

Min(x, y) =

Distance-Vector(G,s,v)

v.d = min{opt(i-1,s,v), Distance-Vector(G,s,w)+opt(i-1,w,v)};

Distance-Vector-Every(G, s,v)

for every vertex v except s do

Distance-Vector(G,s,v);

T(n) = O(V\*N)

1. The way for building graph is same with question3.

List:

void Graph\_Heapify(Node\* vertices, int i, int n) {

int l = i \* 2 + 1, r = i \* 2 + 2;

int min = 0;

if (l < n && vertices[l].d < vertices[i].d)

min = l;

else

min = i;

if (r < n && vertices[r].d < vertices[min].d)

min = r;

if (min != i) {

swap(vertices[min], vertices[i]);

Graph\_Heapify(vertices, min, n);

}

}

void Build\_Graph\_Heap(Node\* vertices, int n) {

for (int i = (n - 1) / 2; i >= 0; i--)

Graph\_Heapify(vertices, i, n);

}

Node Extract\_Node(nodeptr& node, int& size) {

Node n = node[0];

nodeptr temp = new Node[size - 1];

for (int i = 1; i < size; i++)

temp[i-1] = node[i];

node = temp;

size--;

return n;

}

void Add\_Node(nodeptr& node, int& size, Node n) {

nodeptr temp = new Node[size + 1];

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

temp[i] = node[i];

temp[size] = n;

node = temp;

size++;

}

nodeptr Dijkstra\_List(List\_Graph graph, int s) {

int sizeQ = graph.getSize();

int size = graph.getSize();

int sizeS = 0;

nodeptr V = new Node[sizeQ];

nodeptr S = new Node[size];

list<Node> Q;

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

Node n;

if (graph.getVertex()[i] == s) {

n.v = graph.getVertex()[i]; n.p = -1; n.d = 0;

}

else {

n.v = graph.getVertex()[i]; n.p = -1; n.d = 10000;

}

Q.push\_back(n);

}

while (!Q.empty()) {

Q.sort();

Node temp = Q.front();

Q.pop\_front();

S[sizeS] = temp;

sizeS++;

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

auto iter = begin(Q);

std::advance(iter, i);

pair<int, int> a;

a = make\_pair(temp.v, iter->v);

pair<int, int> b;

b = make\_pair(iter->v, temp.v);

if (graph.getEdge().count(a) == 1) {

int w = graph.getEdge().find(a)->second;

if (iter->d > w + temp.d) {

iter->p = temp.v;

iter->d = w + temp.d;

}

}

else {

if (graph.getEdge().count(b) == 1) {

int w = graph.getEdge().find(b)->second;

if (iter->d > w + temp.d) {

iter->p = temp.v;

iter->d = w + temp.d;

}

}

}

}

}

return S;

}

Matrix:

nodeptr Dijkstra\_Matrix(Graph graph, int s) {

int sizeQ = graph.getNumVertices();

int size = graph.getNumVertices();

int sizeS = 0;

nodeptr V = new Node[sizeQ];

nodeptr S = new Node[0];

nodeptr Q = new Node[sizeQ];

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

Node n;

if (graph.getVertices()[i] == s) {

n.v = graph.getVertices()[i]; n.p = -1; n.d = 0;

}

else {

n.v = graph.getVertices()[i]; n.p = -1; n.d = 10000;

}

Q[i] = n;

}

while (sizeQ != 0) {

Build\_Graph\_Heap(Q, sizeQ);

Node temp = Extract\_Node(Q, sizeQ);

Add\_Node(S, sizeS, temp);

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

int a = graph.getVertexIndex(temp.v);

int b = graph.getVertexIndex(Q[i].v);

if (graph.getEdge()[a][b] != -1) {

int w = graph.getEdge()[a][b];

if (Q[i].d > w + temp.d) {

if (Q[i].d > w + temp.d) {

Q[i].p = temp.v;

Q[i].d = w + temp.d;

}

}

}

}

}

return S;

}

|  |  |  |  |
| --- | --- | --- | --- |
| node | Edge | Time for list graph | Time for matrix graph |
| 100 | 300 | 3974.48ms | 1.77ms |
| 100 | 1000 | 11539.51ms | 2.08ms |
| 100 | 4950 | 40796.27ms | 1.60ms |
| 200 | 600 | 30670.59ms | 10.91ms |
| 200 | 2828 | 140914.74ms | 11.86ms |
| 200 | 19900 | 507864.81ms | 11.68ms |
| 400 | 1200 | 401568.20ms | 76.98ms |
| 400 | 8000 | 1592460.32ms | 73.48ms |
| 400 | 79800 | 7436563.39ms | 77.37ms |
| 800 | 2400 | 3965148.12ms | 570.90ms |
| 800 | 22627 | 12450484.33ms | 554.40ms |
| 800 | 319600 | 73246654.65ms | 566.69ms |

1. Extend-Shortest-Paths(L(r−1), W, ∏)

let L(r)[1..n, 1..n] be a new n × n matrix;

Π=π[I, j];

for i = 1 to n do

for j = 1 to n do

L(r)[i, j] = ∞;

for k = 1 to n do

L(r)[i, j] = min{L(r)[i, j], L(r−1)[i, k] + W[k, j]

Π[i, j] = k;

return L(r) and Π.

Slow-All-Pairs-Shortest-Paths(W)

L(1) = W;

for r = 2 to n − 1 do

Π=π[I, j];

let L(r) be a new n × n matrix;

L(r) =Extend-Shortest-Paths(L(r−1), W, Π);

return L(n−1).

1. Faster-All-Pairs-Shortest-Paths(W)

L = W; r = 1;

while r < n − 1 do

for i = 0 to V do

L = Extend-Shortest-Paths(L, L);

r = 2r;

return L(r).