**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по курсовой работе**

**по дисциплине «Алгоритмы и структуры данных»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 9301 |  | Гелета А.И.­­­  Русанова К.В. |
| Преподаватель |  | Тутуева А. В. |

­

Санкт-Петербург

2020

### Постановка задачи. Описание реализуемых алгоритмов

Реализовать класс графа, а также следующие методы:

1. bool IfEulerian(); //проверка наличия Эйлерова цикла
2. bool IfBipartite(); //проверка на двудольность
3. bool IfTree(); // проверка на дерево
4. int\* PruferCode(); //создание кода Прюфера по заданному графу
5. void PruferDecode(int\*, int); //задание графа по коду Прюфера
6. dualList\* StrongConnected(); //определение компонент сильной связности
7. int\* Dijkstra(int); //нахождение кротчайших путей от заданной вершины с помощью алгоритма Дейкстры
8. Iterator create\_dft\_iterator(int); // создание итератора, реализующего один из методов обхода в глубину (depth-first traverse)
9. Iterator create\_bft\_iterator(int) // создание итератора, реализующего методы обхода в ширину (breadth-first traverse)

### Оценка временной сложности

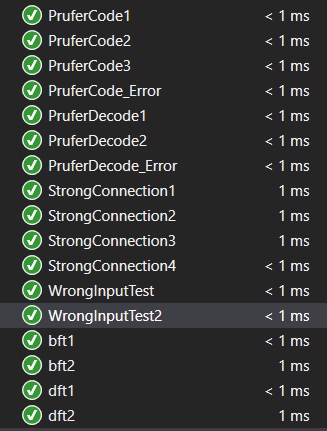
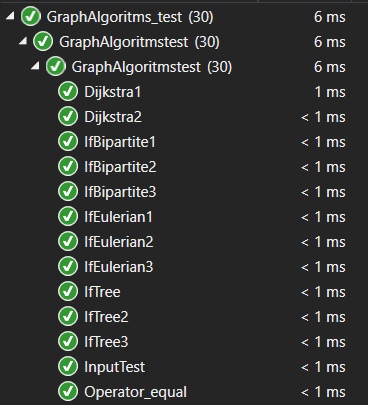
Реализованные алгоритмы рассчитаны на графы с относительно не большим количеством вершин, но большим количеством связей между ними. Поэтому хранение графа осуществляется с помощью матрицы смежности.

Табл.

|  |  |
| --- | --- |
| **Название метода** | **Сложность** |
| IfEulerian | O(n2) |
| IfBipartite | O(n2) |
| IfTree | O(n2) |
| PruferCode | O(n2) |
| PruferDecode | O(n2) |
| StrongConnected | O(n2) |
| Dijkstra | O(n2) |

### Реализованные unit-тесты

1. Operator\_equal — проверка перегрузки равенства графов
2. InputTest — корректный ввод графа
3. WrongInputTest — не корректный ввод графа
4. bft1 — обход графа в ширину
5. bft2 — обход графа в ширину
6. dft1 — обход графа в глубину
7. dft2 — обход графа в глубину
8. Dijkstra1 — проверка алгоритма Дейкстры
9. Dijkstra2 — проверка алгоритма Дейкстры
10. IfEulerian1 — проверка нахождения цикла Эйлера
11. IfEulerian2 — проверка нахождения цикла Эйлера
12. IfEulerian3 — проверка нахождения цикла Эйлера
13. IfBipartite1 — проверка является ли граф двудольным
14. IfBipartite2 — проверка является ли граф двудольным
15. IfBipartite3 — проверка является ли граф двудольным
16. IfTree — проверка является ли граф деревом
17. IfTree2 — проверка является ли граф деревом
18. IfTree3 — проверка является ли граф деревом
19. PruferDecode\_Error — проверка ошибки ввода при декодирование кода Прюфера
20. PruferDecode1 — проверка декодирования кода Прюфера
21. PruferDecode2 — проверка декодирования кода Прюфера
22. PruferCode\_Error — проверка ошибки ввода при создание кода Прюфера
23. PruferCode1 — проверка создания кода Прюфера
24. PruferCode2 — проверка создания кода Прюфера
25. PruferCode3 — проверка создания кода Прюфера
26. StrongConnection1 — проверка компонент сильной связности
27. StrongConnection2 — проверка компонент сильной связности
28. StrongConnection3 — проверка компонент сильной связности
29. StrongConnection4 — проверка компонент сильной связности



### Пример работы

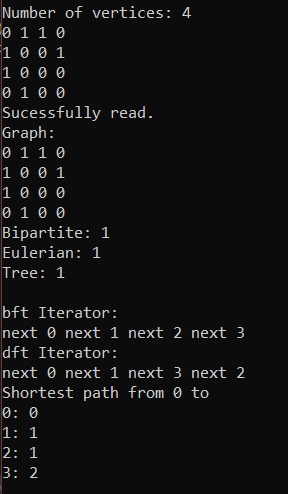


Рис. 1 — Пример работы

Лист. 1 — Программа для демонстрации примера работы

#include <iostream>

#include <fstream>

#include "Graph.h"

#include "Iterator.h";

int main()

{

std::cout << "Hello World!\n";

Graph g\_test(1);

cout << g\_test;

//bipatite

cout << "Bipartite: " << g\_test.IfBipartite() << endl;

//euler

cout << "Eulerian: " << g\_test.IfEulerian() << endl;

//tree

cout << "Tree: " << g\_test.IfTree() << endl;

//bft iterator

cout << endl << "bft Iterator:\n";

Iterator\* g\_bft\_iterator = g\_test.create\_bft\_iterator(0);

std::cout << "next " << g\_bft\_iterator->next() << " ";

while (g\_bft\_iterator->has\_next() != 0)

{

std::cout << "next " << g\_bft\_iterator->next() << " ";

}

//dft iterator

cout << endl << "dft Iterator:\n";

Iterator\* g\_dft\_iterator = g\_test.create\_dft\_iterator(0);

std::cout << "next " << g\_dft\_iterator->next() << " ";

while (g\_dft\_iterator->has\_next() != 0)

{

std::cout << "next " << g\_dft\_iterator->next() << " ";

}

//Dijkstra

int\* dij1\_arr = new int[g\_test.GetSize()];

dij1\_arr = g\_test.Dijkstra(0);

cout << "\nShortest path from 0 to\n";

for (int j = 0; j < g\_test.GetSize(); j++)

cout << j << ": " << dij1\_arr[j] << " \n";

cout << endl;

}

### Листинг

Лист. 2 — Заголовочный файл

#pragma once

#include "Iterator.h";

#include "dualList.h"

#include "Stack.h"

#include "Queue.h"

#include <stdlib.h>

#include <iostream>

using namespace std;

class Graph

{

private:

int\*\* G;

int V;

bool Directed;

bool DirectedCheck();

bool IfCycleNoDir();

public:

Graph(int\*\* new\_graph = NULL, int ver\_num = 0) {

G = new\_graph;

V = ver\_num;

Directed = DirectedCheck();

};

Graph(int choice, string filename = "in.txt");

int GetSize();

bool IfEulerian();

bool IfBipartite();

bool IfTree();

int\* PruferCode();

void PruferDecode(int\*, int);

dualList\* StrongConnected();

int\* Dijkstra(int);

friend std::ostream& operator << (std::ostream&, const Graph&);

friend bool operator == (Graph& left, int right[]) {

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

for (int j = 0; j < left.V; j++)

if (left.G[i][j] != right[i \* left.V + j]) return false;

return true;

}

Iterator\* create\_dft\_iterator(int); // depth-first traverse iterator

Iterator\* create\_bft\_iterator(int); // breadth-first traverse iterator

class dft\_Iterator : public Iterator // depth-first traverse

{

private:

bool\* visited;

int\*\* ItrG;

Stack\* stack;

int Icurrent;

int sizeV;

int before;

bool connection;

public:

dft\_Iterator(int\*\* Gr, int max, int start = 0) {

stack = new Stack();

sizeV = max;

ItrG = Gr;

Icurrent = start;

visited = new bool[max];

stack->push(Icurrent);

for (size\_t i = 0; i < max; i++)

visited[i] = false;

visited[Icurrent] = true;

before = -1;

connection = true;

};

int next();

bool has\_next();

bool newconnection();

int beforecur();

~dft\_Iterator() {

delete visited;

delete\* ItrG;

}

};

class bft\_Iterator : public Iterator // depth-first traverse

{

private:

bool\* visited;

int\*\* ItrG;

Queue\* queue;

int Icurrent;

int sizeV;

int before = -1;

bool connection;

public:

bft\_Iterator(int\*\* Gr, int max, int start = 0) {

queue = new Queue();

sizeV = max;

ItrG = Gr;

Icurrent = start;

visited = new bool[max];

queue->push(Icurrent);

for (size\_t i = 0; i < max; i++)

visited[i] = false;

visited[Icurrent] = true;

connection = true;

};

int next();

bool has\_next();

bool newconnection();

int beforecur();

~bft\_Iterator() {

delete visited;

delete\* ItrG;

}

};

~Graph() {

for (size\_t i = 0; i < V; i++)

free(G[i]);

free(G);

}

};

Лист. — Файл с реализованными методами

#include "Graph.h"

#include <iostream>

#include <stdlib.h>

#include <string>

#include <fstream>

Graph::Graph(int choice, string filename) {

//choice 1 -> reading from console

//choice 2-> reading from file

fstream f;

//reading vertices

if (choice == 1) {

cout << "Number of vertices: ";

cin >> V;

}

else if (choice == 2) {

f.open(filename, ios::in);

f >> V;

if (f.eof() || V == 0) throw std::out\_of\_range("Graph is empty");

}

else {

cout << "Wrong input. Try again\n";

throw std::invalid\_argument("Invalid Option");

return;

}

//memory

int\*\* p;

p = (int\*\*)malloc(sizeof(int\*) \* V);

if (!p) {

throw std::out\_of\_range("Allocation error");

return;

}

else {

G = p;

}

//reading graph

if (choice == 1) {

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

{

p[i] = (int\*)malloc(V \* sizeof(int));

if (!p[i]) { throw std::out\_of\_range("Allocation error"); return; }

else G[i] = p[i];

//reading

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

{

int temp; cin >> temp;

if (temp == 0 || temp == 1) G[i][j] = temp;

else {

throw std::invalid\_argument("Input error");

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

free(G[k]);

free(G);

V = 0;

}

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

}

}

cout << "Sucessfully read.\n";

}

else if (choice == 2) {

//we believe that people can write input properly i guess

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

{

p[i] = (int\*)malloc(V \* sizeof(int));

if (!p[i]) { throw std::out\_of\_range("Allocation error"); return; }

else G[i] = p[i];

//reading

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

{

if (f.eof()) {

throw std::invalid\_argument("End of file");

}

int temp; f >> temp;

if (temp == 0 || temp == 1) G[i][j] = temp;

else {

throw std::invalid\_argument("Input error");

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

free(G[k]);

free(G);

V = 0;

}

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

}

}

cout << "Sucessfully read." << V << endl;

f.close();

}

//checking if graph is directed

Directed = DirectedCheck();

}

bool Graph::DirectedCheck() {

for (size\_t i = 0; i < V; i++)

for (size\_t j = i; j < V; j++)

if (G[i][j] != G[j][i])

return true; //symmetry check

return false;

}

bool Graph::IfCycleNoDir() {

if (Directed) {

throw std::invalid\_argument("Invalid Graph Type");

}

else {

Iterator\* g\_dft\_iterator = create\_dft\_iterator(0);

bool\* visited = new bool[V];

for (size\_t i = 0; i < V; i++)

visited[i] = false;

int cur,bef;

while (g\_dft\_iterator->has\_next())

{

bef = g\_dft\_iterator->beforecur();

cur = g\_dft\_iterator->next();

visited[cur] = true;

for (size\_t i = 0; i < V; i++)

if ((G[cur][i] == 1) && (visited[i] == true) && (i != bef)) //we can go to the already visited peak, but not to the previous one

return true;

}

}

return false;

}

bool Graph::IfEulerian() {

//check for connections

Iterator\* g\_dft\_iterator = create\_dft\_iterator(0);

int max\_connect = 0, cur\_connect = 0;

while (g\_dft\_iterator->has\_next())

{

g\_dft\_iterator->next();

if (!g\_dft\_iterator->newconnection()) cur\_connect++;

else {

if (max\_connect > 1) { if (cur\_connect > 1) return false; }

if (cur\_connect > max\_connect) max\_connect = cur\_connect;

cur\_connect = 0;

}

}

if (cur\_connect > 1 && max\_connect > 1) return false;

//count

int\* count\_rows, \*count\_col;

count\_rows = new int[V];

count\_col = new int[V];

for (int j = 0; j < V; j++) count\_col[j] = 0;

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

count\_rows[i] = 0;

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

if (G[i][j] == 1) { count\_rows[i]++; count\_col[j]++; }

}

}

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

if (!Directed && (count\_col[i] + count\_rows[i]) % 2 != 0) return false;

else if (Directed && (count\_col[i] != count\_rows[i])) return false;

return true;

}

bool Graph::IfBipartite() {

int\* color = new int[V];

for (size\_t i = 0; i < V; i++)

color[i] = 2; //1-black,0-white,2-grey

Iterator\* g\_bft\_iterator = create\_bft\_iterator(0);

int cur;

while (g\_bft\_iterator->has\_next())

{

if (g\_bft\_iterator->newconnection()) //if switched to another connected component

{

cur = g\_bft\_iterator->next();

color[cur] = 1; //paint in black

}

else

cur = g\_bft\_iterator->next();

for (size\_t i = 0; i < V; i++)

if (G[cur][i] == 1)

{

switch (color[i])

{

case 2: //if painted gray

if (color[cur] == 1) //paint in the opposite color

color[i] = 0;

else color[i] = 1;

break;

default:

if (color[i] == color[cur]) //if the colors match

{

return false;

}

break;

}

}

}

return true;

}

bool Graph::IfTree() {

Iterator\* g\_dft\_iterator = create\_dft\_iterator(0);

if (Directed)

{

int countDeg;

int count = 0;

for (size\_t i = 0; i < V; i++)

{

countDeg = 0;

for (size\_t j = 0; j < V; j++)

countDeg += G[j][i];

if (countDeg>1) //entry degree greater than 1

{

return false;

}

else if (countDeg==0)

{

count++;

if (count > 1) //entry degree 0 only at one vertex

return false;

}

}

if (count==0) //no root

{

return false;

}

}

else {

if (IfCycleNoDir()) return false; //no cycles

while (g\_dft\_iterator->has\_next())

{

g\_dft\_iterator->next();

if (g\_dft\_iterator->newconnection()) return false; //one component of connectivity

}

}

return true;

}

int\* Graph::PruferCode() {

//code only for not directed trees

if (!IfTree() || Directed) throw std::invalid\_argument("Invalid Graph Type");

if (V <= 2) return NULL;

int\* PrufC = new int[V - 2];

int\*\* G\_copy = new int\*[V];

//copy the array to work with

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

{

G\_copy[i] = new int[V];

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

G\_copy[i][j] = G[i][j];

}

int cur\_ver = 0, connect\_num = 0, next\_ver = 0, cur\_Pruf = 0;

bool exit = false;

while (cur\_Pruf < V - 2) {

for (int j = 0; j < V; j++) { //searching for the smallest 'leaf' with 1 connection

if (G\_copy[cur\_ver][j] == 1) {

connect\_num++;

if (connect\_num > 1) { cur\_ver++; exit = true; break; }

next\_ver = j;

}

}

if (connect\_num == 0) { exit = true; cur\_ver++; }

if (!exit) { //found the leaf, deleting it and adding number to prufer code

G\_copy[next\_ver][cur\_ver] = 0;

G\_copy[cur\_ver][next\_ver] = 0;

PrufC[cur\_Pruf] = next\_ver;

if (next\_ver < cur\_ver) cur\_ver = next\_ver;

else cur\_ver++;

cur\_Pruf++;

}

connect\_num = 0, next\_ver = 0; exit = false;

}

return PrufC;

}

void Graph::PruferDecode(int\* PrufC, int Pruf\_length) {

for (int i = 0; i < Pruf\_length; i++)

if (PrufC[i] >= Pruf\_length + 2) throw std::invalid\_argument("Invalid Prufer Code");

V = Pruf\_length + 2;

int\* ver\_num = new int[V];

int\*\* p = (int\*\*)malloc(sizeof(int\*) \* V);

int pos = 0, pos\_pruf = 0;

//two arrays; prufer code and vertices in order

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

{

ver\_num[i] = i;

p[i] = (int\*)malloc(V \* sizeof(int));

for (int j = 0; j < V; j++) p[i][j] = 0;

}

for (; pos\_pruf < V - 2; pos\_pruf++) {

//connecting a vertex from prufer with smallest vertex from other array

//second vertex shouldn't be in prufer code

for (int i = pos\_pruf; i < V - 2; i++) {

if (ver\_num[pos] == PrufC[i] || ver\_num[pos] == -1) { i = pos\_pruf -1; pos++; }

}

p[ver\_num[pos]][PrufC[pos\_pruf]] = 1;

p[PrufC[pos\_pruf]][ver\_num[pos]] = 1;

ver\_num[pos] = -1;

pos = 0;

}

int row = -1, col = -1;

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

if (ver\_num[i] != -1 && row == -1) row = ver\_num[i];

else if (ver\_num[i] != -1) { col = ver\_num[i]; break; }

p[row][col] = 1; p[col][row] = 1;

G = (int\*\*)realloc(G, sizeof(int\*) \* V);

G = p;

}

dualList\* Graph::StrongConnected() {

int\* number = new int[V];

bool\* visited = new bool[V];

Stack\* stack = new Stack();

int backcur;

bool canGo=true;

int Icurrent = 0, count = 0,temp;

stack->push(Icurrent);

for (size\_t i = 0; i < V; i++) {

number[i] = -1;

visited[i] = false;

}

cout << endl;

while (canGo) //depth-first traversal in a transposed graph

{

visited[Icurrent] = true;

temp = Icurrent;

for (size\_t i = 0; i < V; i++)

if ((visited[i] == false) && (G[i][Icurrent] == 1))

{

Icurrent = i;

stack->push(Icurrent);

break;

}

if (count > 0) { //number the vertices upon exit

if ((temp == Icurrent) && (number[count - 1] != Icurrent))

{

number[count] = Icurrent;

count++;

}

}

else {

if (temp == Icurrent)

{

number[count] = Icurrent;

count++;

}

}

while ((temp == Icurrent) && !stack->isEmpty())

{

backcur = stack->pop();

for (size\_t i = 0; i < V; i++)

if ((visited[i] == false) && (G[i][backcur] == 1))

{

Icurrent = i;

stack->push(backcur);

stack->push(Icurrent);

break;

}

if (count > 0) { //number the vertices upon exit

if ((temp == Icurrent) && (number[count - 1] != backcur))

{

number[count] = backcur;

count++;

}

}else{

if (temp == Icurrent)

{

number[count] = backcur;

count++;

}

}

}

if (temp == Icurrent)

{

for (size\_t i = 0; i < V; i++)

if (visited[i] == false)

{

Icurrent = i;

stack->push(Icurrent);

break;

}

}

canGo = false;

if (!stack->isEmpty())

canGo = true;

else {

for (size\_t i = 0; i < V; i++)

if (visited[i] == false) {

canGo = true;

break;

}

}

}

for (size\_t i = 0; i < V; i++) {

visited[i] = false;

}

canGo = true;

int countSv = 0;

Icurrent = number[count-1]; //starting with the last one on the way out

stack->clear();

dualList\* Result = new dualList[V];

stack->push(Icurrent);

while (canGo) //number the vertices upon exit

{

visited[Icurrent] = true;

for (size\_t i = 0; i < V; i++)

if (number[i] == Icurrent)

{

Result[countSv].push\_back(Icurrent); //add to list

number[i] = -1;

break;

}

temp = Icurrent;

for (size\_t i = 0; i < V; i++)

if ((visited[i] == false) && (G[Icurrent][i] == 1))

{

Icurrent = i;

stack->push(Icurrent);

break;

}

while ((temp == Icurrent) && !stack->isEmpty())

{

backcur = stack->pop();

for (size\_t i = 0; i < V; i++)

if ((visited[i] == false) && (G[backcur][i] == 1))

{

Icurrent = i;

stack->push(backcur);

stack->push(Icurrent);

break;

}

}

if (temp == Icurrent)

{

for (int i = V-1; i >= 0; i--)

if (number[i] != -1) //we go to the next not visited from the last one on the way out

{

if (visited[number[i]] == false)

{

countSv++; //came out of the tree

Icurrent = number[i];

stack->push(Icurrent);

break;

}

}

}

canGo = false;

if (!stack->isEmpty())

canGo = true;

else {

for (size\_t i = 0; i < V; i++)

if (visited[i] == false) {

canGo = true;

break;

}

}

}

return Result;

}

int Graph::GetSize() {

return V;

}

int\* Graph::Dijkstra(int A) {

int\* distance = new int[V];

bool\* visited = new bool[V];

int\*\* G\_copy = new int\* [V];

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

{

G\_copy[i] = new int[V];

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

if (G[i][j] == 0)

{

G\_copy[i][j] = INT\_MAX;

}

else {

G\_copy[i][j] = G[i][j];

}

}

for (size\_t i = 0; i < V; i++) {

distance[i] = G\_copy[A][i];

visited[i] = false;

}

int u=0,index=0,min;

distance[A] = 0;

for (size\_t i = 0; i < V; i++)

{

min = INT\_MAX;

for (size\_t j = 0; j < V; j++)

{

if ((visited[j]==false)&&(distance[j]<min)) //finding a shortcut

{

min = distance[j];

index = j;

}

}

u = index;

visited[u] = true;

for (size\_t j = 0; j < V; j++)

{

if ((visited[j]==false)&&(G\_copy[u][j]!=INT\_MAX) && (distance[u] != INT\_MAX) &&((distance[u]+G\_copy[u][j]) < distance[j])) //if shorter bypass

{

distance[j] = distance[u] + G\_copy[u][j];

}

}

}

for (size\_t i = 0; i < V; i++)

{

delete G\_copy[i];

}

delete G\_copy;

return distance; //if distance[i] == INT\_MAX, the way doesn't exist

}

Iterator\* Graph::create\_dft\_iterator(int start = 0) {

return new dft\_Iterator(G,V,start);

}

std::ostream& operator << (std::ostream& out, const Graph& Gr) {

cout << "Graph: " << endl;

for (int i = 0; i < Gr.V; i++) {

for (int j = 0; j < Gr.V; j++) {

cout << Gr.G[i][j] << " ";

}

cout << endl;

}

return out;

}

bool Graph::dft\_Iterator::has\_next() {

if (!stack->isEmpty())

return true;

for (size\_t i = 0; i < sizeV; i++)

if (visited[i] == false)

return true;

return false;

}

int Graph::dft\_Iterator::next() {

if (!has\_next()) {

throw std::out\_of\_range("No more elements");

}

visited[Icurrent] = true;

int temp = Icurrent;

for (size\_t i = 0; i < sizeV; i++)

if ((visited[i] == false) && (ItrG[Icurrent][i] == 1)) //go deeper

{

before = Icurrent;

Icurrent = i;

stack->push(Icurrent);

connection = false;

break;

}

int backcur;

while ((temp == Icurrent) && !stack->isEmpty())

{

backcur = stack->pop(); //pop the top off the stack

for (size\_t i = 0; i < sizeV; i++)

if ((visited[i] == false) && (ItrG[backcur][i] == 1)) //go deeper

{

before = backcur;

Icurrent = i;

stack->push(backcur);

stack->push(Icurrent);

connection = false;

break;

}

}

if (temp == Icurrent)

{

for (size\_t i = 0; i < sizeV; i++)

if (visited[i] == false) //go to the next component of connectivity

{

before = -1;

Icurrent = i;

stack->push(Icurrent);

connection = true;

break;

}

}

return temp;

}

int Graph::dft\_Iterator::beforecur() {

return before;

}

bool Graph::dft\_Iterator::newconnection() {

return connection;

}

Iterator\* Graph::create\_bft\_iterator(int start = 0) {

return new bft\_Iterator(G, V, start);

}

bool Graph::bft\_Iterator::has\_next() {

if (!queue->isEmpty())

return true;

for (size\_t i = 0; i < sizeV; i++)

if (visited[i] == false)

return true;

return false;

}

int Graph::bft\_Iterator::next() {

if (!has\_next()) {

throw std::out\_of\_range("No more elements");

}

visited[Icurrent] = true;

int temp = Icurrent;

for (size\_t i = 0; i < sizeV; i++)

if ((visited[i] == false) && (ItrG[Icurrent][i] == 1)) { //put all unvisited neighboring vertices in the queue

queue->push(i);

}

while (!queue->isEmpty() && (visited[Icurrent] == true))

{

Icurrent = queue->pop(); //getting the top out of the queue

connection = false;

}

if (queue->isEmpty() && (visited[Icurrent] == true))

{

for (size\_t i = 0; i < sizeV; i++)

if (visited[i] == false) //go to the next component of connectivity

{

Icurrent = i;

queue->push(Icurrent);

connection = true;

break;

}

}

return temp;

}

int Graph::bft\_Iterator::beforecur() {

return before;

}

bool Graph::bft\_Iterator::newconnection() {

return connection;

}