Національний Технічний Університет України
“Київський Політехнічний Інститут”
Фізико-Технічний Інститут

ПРИКЛАДНІ АЛГОРИТМИ

КОМП’ЮТЕРНИЙ ПРАКТИКУМ №1
Графи Варіант 7

Виконав студент 3-го курсу
групи ФІ-14
Дорошенко Юірй

Київ 2023

1 Вступ

Звіт з першого комп'ютерного практикума по прикладним алгоритмам. Увесь код можна знайти за посиланням на GitHub.

1.1 Мета

Опанувати способи представлення різних типів графів (неорієнтовані, орієнтовані, зважені) та їх ефективної реалізації. Реалізувати алгоритм пошуку сильних компонент зв'язності орієнтованого графу, проаналізувати отримані з нього данні.

1.2 Виконання завдання

1) Я реалізував базовий клас граф матрицею суміжності (з конвертором в список суміжності). В ньому також реалізовані функції додавання ввершини, видалення вершини, виводу матриці та списку суміжності.

2) Реалізував дочірні класи Орієнтований граф, неорієнтований граФ та зважений граф (у формі неорієнтованого).

3) На них реалізовані такі функції:

    1. додавання нового ребра
    2. видалення ребра
    3. генерація рандомного графу за вказаною щільністю (для зваженого графа також є вхідні мінімальна та максимальна вага)

4) Реалізував "Алгоритм пошуку компонент сильної зв'язності за dfs"для орієнтовного графа та функцію підрахунку середньої ваги та середньої кількості ксз для великої кількості графів, задля подальшого аналізу цих данних.

2 Код

```
#include <iostream>
#include <vector>
#include <random>
#include <stack>
#include <algorithm>
#include <cmath>


class Graph {

public:

    int vertex;
    std::vector<std::vector<int>> matrix;
    std::vector<std::vector<int>> list;

    Graph(int vertex) {
        this->vertex = vertex;
        matrix.resize(vertex, std::vector<int>(vertex, false));
        list.resize(vertex);
    }

    void addVertex(int newVertex) {
        int oldVertex = vertex;
```

```
    std::vector<std::vector<int>> newMatrix(newVertex, std::vector<int>(newVertex,
        false));


    for (int i = 0; i < oldVertex; ++i) {
        for (int j = 0; j < oldVertex; ++j) {
            newMatrix[i][j] = matrix[i][j];
        }
    }

    vertex = newVertex;
    matrix = newMatrix;
}

void removeVertex(int v) {
    if (v < 0 || v >= vertex) {
        std::cout << "Invalid vertex index." << std::endl;
        return;
    }

    int oldVertex = vertex;


    std::vector<std::vector<int>> newMatrix(oldVertex - 1, std::vector<int>(oldVertex -
        1, false));

    int newRow = 0;
    for (int i = 0; i < oldVertex; ++i) {
        if (i == v) {

            continue;
        }

        int newCol = 0;
        for (int j = 0; j < oldVertex; ++j) {
            if (j == v) {

                continue;
            }


            newMatrix[newRow][newCol] = matrix[i][j];
            ++newCol;
        }
        ++newRow;
    }

    vertex = oldVertex - 1;
    matrix = newMatrix;
}


void printMatrix() {
    for (int i = 0; i < vertex; ++i) {
        for (int j = 0; j < vertex; ++j) {
            std::cout << matrix[i][j] << " ";
        }
        std::cout << std::endl;
    }
}
void printList() {
```

```
        std::cout << "Adjacency List:\n";
        for (int i = 0; i < vertex; ++i) {
            std::cout << i << " -> ";
            for (int j : list[i]) {
                std::cout << j << " ";
            }
            std::cout << "\n";
        }
    }



};

class undirectedGraph : public Graph {
public:
    undirectedGraph(int vertex) : Graph(vertex) {}

    void addEdge(int source, int destination) {
        if (source >= 0 && source < vertex && destination >= 0 && destination < vertex) {
            matrix[source][destination] = true;
            matrix[destination][source] = true;
        }
    }

    void undirectedGenerator(double edgeProbability) {
        if (edgeProbability < 0.0 || edgeProbability > 1.0) {
            std::cout << "Invalid edge probability." << std::endl;
            return;
        }

        std::random_device rd;
        std::default_random_engine generator(rd());
        std::uniform_real_distribution<double> distribution(0.0, 1.0);

        for (int i = 0; i < vertex; ++i) {
            for (int j = i + 1; j < vertex; ++j) {
                if (distribution(generator) < edgeProbability) {
                    addEdge(i, j);
                }
            }
        }
    }
    void undirectedtoList() {
        for (int i = 0; i < vertex; ++i) {
            for (int j = 0; j < vertex; ++j)
            {
                if (matrix[i][j] == 1) {
                    list[i].push_back(j);

                }
                else {
                    j = j + 1;
                }
            }
        }
    }


};

class directedGraph : public Graph {
```

```
public:
    directedGraph(int vertex) : Graph(vertex) {}

    void addEdge(int source, int destination) {
        if (source >= 0 && source < vertex && destination >= 0 && destination < vertex) {
            matrix[source][destination] = true;
        }
    }

    void directedGenerator(double edgeProbability) {
        if (edgeProbability < 0.0 || edgeProbability > 1.0) {
            std::cout << "Invalid edge probability." << std::endl;
            return;
        }

        std::random_device rd;
        std::default_random_engine generator(rd());
        std::uniform_real_distribution<double> distribution(0.0, 1.0);

        for (int i = 0; i < vertex; ++i) {
            for (int j = 0; j < vertex; ++j) {
                if (i != j && distribution(generator) < edgeProbability) {
                    addEdge(i, j);
                }
            }
        }
    }
    void directedtoList() {
        for (int i = 0; i < vertex; ++i) {
            for (int j = 0; j < vertex; ++j)
            {
                if (matrix[i][j] == 1) {
                    list[i].push_back(j);
                    list[j].push_back(i);

                }
                else {
                    j = j + 1;
                }

            }
        }
    }
};

class weightedGraph : public Graph {
public:
    weightedGraph(int vertex) : Graph(vertex) {}

    void addEdge(int source, int destination, int weight) {
        if (source >= 0 && source < vertex && destination >= 0 && destination < vertex) {
            matrix[source][destination] = weight;
        }
    }

    void weightedGenerator(int minWeight, int maxWeight, double probability) {
        if (minWeight > maxWeight || probability < 0.0 || probability > 1.0) {
            std::cout << "Invalid weight range or probability." << std::endl;
            return;
        }

        std::random_device rd;
```

```
        std::default_random_engine generator(rd());
        std::uniform_int_distribution<int> weightDistribution(minWeight, maxWeight);
        std::uniform_real_distribution<double> probDistribution(0.0, 1.0);

        for (int i = 0; i < vertex; ++i) {
            for (int j = 0; j < vertex; ++j) {
                if (i != j) {
                    double randValue = probDistribution(generator);
                    if (randValue <= probability) {
                        int weight = weightDistribution(generator);
                        addEdge(i, j, weight);
                    }
                }
            }
        }
    }

    void weightedtoList() {
        for (int i = 0; i < vertex; ++i) {
            for (int j = 0; j < vertex; ++j)
            {
                if (matrix[i][j] > 1) {
                    list[i].push_back(j);

                }
                else {
                    j = j + 1;
                }

            }
        }
    }

};

void dfs(int v, const std::vector<std::vector<int>>& graph, std::vector<bool>& visited,
     std::stack<int>& stack) {
    visited[v] = true;

    for (int i = 0; i < graph.size(); ++i) {
        if (graph[v][i] && !visited[i]) {
            dfs(i, graph, visited, stack);
        }
    }

    stack.push(v);
}

std::vector<std::vector<int>> transposeGraph(const std::vector<std::vector<int>>& graph) {
    int numVertices = graph.size();
    std::vector<std::vector<int>> transposed(numVertices, std::vector<int>(numVertices, 0));

    for (int i = 0; i < numVertices; ++i) {
        for (int j = 0; j < numVertices; ++j) {
            transposed[i][j] = graph[j][i];
        }
    }

    return transposed;
}
```

```
void dfsUtil(int v, const std::vector<std::vector<int>>& graph, std::vector<bool>& visited,
     std::vector<int>& component) {
    visited[v] = true;
    component.push_back(v);

    for (int i = 0; i < graph.size(); ++i) {
        if (graph[v][i] && !visited[i]) {
            dfsUtil(i, graph, visited, component);
        }
    }
}

std::vector<std::vector<int>> findStronglyConnectedComponents(const
     std::vector<std::vector<int>>& graph) {
    int numVertices = graph.size();
    std::vector<bool> visited(numVertices, false);
    std::stack<int> stack;

    for (int i = 0; i < numVertices; ++i) {
        if (!visited[i]) {
            dfs(i, graph, visited, stack);
        }
    }

    std::vector<std::vector<int>> transposed = transposeGraph(graph);

    visited.assign(numVertices, false);
    std::vector<std::vector<int>> stronglyConnectedComponents;

    while (!stack.empty()) {
        int v = stack.top();
        stack.pop();

        if (!visited[v]) {
            std::vector<int> component;
            dfsUtil(v, transposed, visited, component);
            stronglyConnectedComponents.push_back(component);
        }
    }

    return stronglyConnectedComponents;
}

int main() {
    int numVertices = 10;
    double probability = 0.3;
    int numGraphs = 100;

    double totalNumComponents = 0.0;
    double totalAverageSize = 0.0;

    for (int graphIndex = 1; graphIndex <= numGraphs; ++graphIndex) {
        directedGraph graph(numVertices);
        graph.directedGenerator(probability);

        std::vector<std::vector<int>> stronglyConnectedComponents =
             findStronglyConnectedComponents(graph.matrix);

        int numComponents = stronglyConnectedComponents.size();
        double averageSize = 0.0;
        for (const auto& component : stronglyConnectedComponents) {
            averageSize += component.size();
```

```
        }
        if (numComponents > 0) {
            averageSize /= numComponents;
        }

        totalNumComponents += numComponents;
        totalAverageSize += averageSize;

    }

    double averageNumComponents = totalNumComponents / numGraphs;
    double averageAverageSize = totalAverageSize / numGraphs;

    std::cout << "Average Number of Strongly Connected Components for " << numGraphs << "
        Graphs: " << averageNumComponents << std::endl;
    std::cout << "Average Average Size of Strongly Connected Components for " << numGraphs <<
        " Graphs: " << averageAverageSize << std::endl;

    return 0;
}
```

Табл. 1: Серед. к-ть комп. зв'язності+дисперсія

щільність/к-ть вершин	5	10	15	25	40	50	Середнє	Дисперсія
0.1	4.85	9.03	10.39	6.64	2.35	1.49	5.1533	8.5799
0.2	4.36	5.08	2.48	1.36	1.01	1	2.5873	2.3697
0.3	3.46	2.13	1.22	1.01	1	1	1.9703	1.2142
0.4	2.44	1.26	1.01	1	1	1	1.4033	0.2688
0.5	1.73	1.02	1.01	1	1	1	1.0033	0.0336
0.6	1.29	1	1	1	1	1	1	0

Табл. 2: Серед. розмір комп. зв'язності

щільність/к-ть вершин	5	10	15	25	40	50	Середнє	Дисперсія
0.1	1.04083	1.13635	1.67196	5.24946	23.4767	38.9167	15.690208	174.660
0.2	1.25	2.86437	8.43893	21.1667	39.8	50	16.25885	193.1504
0.3	1.79333	6.18345	13.5625	24.875	40	50	18.87581	193.0473
0.4	2.66167	8.84167	14.925	25	40	50	19.39917	172.8026
0.5	3.72583	9.9	14.925	25	40	50	19.3	142.83
0.6	4.39583	10	15	25	40	50	19.61667	112.9667

3 Приколи з КСЗ

Побудуємо таблиці залежності кількості компонент сильної зв'язності та к-ть вершин в них, в залежності від щільності графу та кількості його ребер. Для тестів всіх значень будувалось по 100 графів.
Так як всі розрахунки в тестах були рівноімовірними, таблиці репрезентують математичне сподівання для відповідних значень щільності та кількості вершин.
Розрахуємо ще середні значення по таблиці та дисперсію за формулою:

$$\text{Дисперсія} = \frac{\sum_{i=1}^{n}(x_i - \text{середнє})^2}{n} \tag{1}$$

4 Висновок

Я опанував представлення графів та реалізацію алгоритмів для них, а саме: алгоритм пошуку компонент сильної зв'язності та генератори рандомних графів. Навчився працювати з данними, що залежать від характеристик згенерованих мною графів.