Министерство науки и высшего образования Российской Федерации
Российский химико-технологический университет имени Д. И. Менделеева

Факультет информационных технологий и управления
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

«Графы»

Выполнил студент группы КС-30 Колесников Артем Максимович

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/AMKolesnikov_30_ALG

Приняли: аспирант кафедры ИКТ Пысин Максим Дмитриевич
аспирант кафедры ИКТ Краснов Дмитрий Олегович

Дата сдачи: 04.04.2022

Москва
2022

Содержание

Описание задачи

В рамках лабораторной работы необходимо реализовать генератор случайных графов, генератор должен содержать следующие параметры:

- Максимальное/Минимальное количество генерируемых вершин
- Максимальное/Минимальное количество генерируемых ребер
- Максимальное количество ребер связанных с одной вершины
- Генерируется ли направленный граф
- Максимальное количество входящих и выходящих ребер

Сгенерированный граф должен быть описан в рамках одного класса (этот класс не должен заниматься генерацией), и должен обладать обязательно следующими методами:

- Выдача матрицы смежности
- Выдача матрицы инцидентности
- Выдача список смежности
- Выдача списка ребер

В качестве проверки работоспособности, требуется сгенерировать 10 графов с возрастающим количеством вершин и ребер (количество выбирать в зависимости от сложности расчета для вашего отдельно взятого ПК). На каждом из сгенерированных графов требуется выполнить поиск кратчайшего пути или подтвердить его отсутствие из точки А в точку Б, выбирающиеся случайным образом заранее, поиском в ширину и поиском в глубину, замерев время, требуемое на выполнение операции. Результаты замеров наложить на график и проанализировать эффективность применения обоих методов к этой задаче.

Описание структуры

Граф – совокупность точек, соединенных линиями. Точки называются вершинами, или узлами, а линии – ребрами, или дугами.

Степень входа вершины – количество входящих в нее ребер, степень выхода – количество исходящих ребер.

Граф, содержащий ребра между всеми парами вершин, является полным.

В ориентированном графе ребра являются направленными, т.е. существует только одно доступное направление между двумя связными вершинами.

В неориентированном графе по каждому из ребер можно осуществлять переход в обоих направлениях.

Граф может быть представлен (сохранен) несколькими способами:

1) Матрица смежности, это двумерная таблица, для которой столбцы и строки соответствуют вершинам, а значения в таблицы соответствуют ребрам, для невзвешенного графа они могут быть просто 1 если связь есть и идет в нужном направлении и 0 если ее нет, а для взвешенного графа будут стоять конкретные значения.
2) Матрица инцидентности, это матрица, в которой строки соответствуют вершинам, а столбцы соответствуют связям, и ячейки ставиться 1 если связь выходит из вершины,

-1 если входит и 0 во всех остальных случаях.

3) Список смежности, это список списков, содержащий все вершины, а внутренние списки для каждой вершины содержат все смежные ей.
4) Список ребер, это список строк, в которых хранятся все ребра вершины, а внутренние значение содержит две вершины к которым присоединено это ребро.

Выполнение задачи

Я реализовал граф на языке C++. Моя программа состоит из функции «GenerateGraphs» и класса «GraphViews», который содержит 6 функций: adjacencyMatrix, adjacencyList, edgeList, incidenceMatrix, bfs, dfs.

Функция «GenerateGraphs» ~ генерирует случайный граф с указанными параметрами вершин, ребер, ориентированный / неориентированный в виде матрицы смежности.

```
vector<vector<int>> GenerateGraphs(int min_vertex, int max_vertex, int min_edge, int
max_edge, int max_num_edge_one_vortex, bool is_directed) {

	int vertex, edges;

	vertex = numGenerator(min_vertex, max_vertex);
	int max_possible_num_edges = vertex * (vertex - 1) / 2;
	if ((min_edge <= max_possible_num_edges) && (max_edge <= max_possible_num_edges))
		edges = numGenerator(min_edge, max_edge);
	else if (min_edge <= ((vertex * (vertex - 1)) / 2)) {
		cout << "Максимальное число ребер превышает возможное значение ребер!" <<
endl;
		edges = numGenerator(min_edge, max_possible_num_edges);
	}
	else {
		cout << "Минимальное число ребер превышает возможное значение ребер! Поэтому
число ребер будет равно максимально возможному: " << max_possible_num_edges << endl;
		edges = max_possible_num_edges;
	}

	edges = numGenerator(min_edge, max_edge);

	vector<vector<int>> graph(vertex, vector<int>(vertex));

	for (int i = 0; i < edges; i++)
	{
		int u, v;
		bool flag = false;

		do {
			u = rand() % vertex;
			v = rand() % vertex;

			int sum1 = 0;
			int sum2 = 0;
			flag = false;

			for (int i = 0; i < vertex; i++)
			{
				sum1 += graph[u][i];
				sum2 += graph[v][i];
			}

			if (u == v)
				flag = true;


		} while (flag);

		if (is_directed)
			graph[u][v] = 1;
```

```
            else
                graph[u][v] = graph[v][u] = 1;

        }

        return graph;

         }
```

Класс «GraphViews» ~ содержит 4 функции возможного представления графа и 2 функции поиска пути между вершинами:

```
class GraphViews
{
public:
        GraphViews (vector<vector<int>> G) {
                adjMatrix = G;
                N = G.size();
                for (int i = 0; i < N; i++)
                {
                        used.push_back(0);
                }
        };
                ~ GraphViews () {};

         vector<vector<int>> adjacencyMatrix() {…}
         vector<vector<int>> adjacencyList() {…}
         vector<vector<int>> edgeList() {…}
         vector<vector<int>> incidenceMatrix() {…}
         struct Edge {…}
         void bfs(int start_ind, int finish_ind) {…}
         void dfs(int start_ind, int finish_ind) {…}

private:
        vector<vector<int>> adjMatrix;
        int N;
        vector<bool> used;
         };
```

Функция «adjacencyMatrix» ~ возвращает и выводит на экран матрицу смежности графа.

```
        vector<vector<int>> adjacencyMatrix() {

                cout << "\nМатрица смежности" << endl;

                for (int i = 0; i < N; i++) {
                        for (int j = 0; j < N; j++) {
                                cout << adjMatrix[i][j] << "\t";
                        }

                        cout << endl;
                }

                cout << endl;

                return adjMatrix;
                }
```

Функция «adjacencyList» ~ возвращает и выводит на экран список смежности графа.

```
vector<vector<int>> adjacencyList() {
		vector<vector<int>> g(N);
		for (int i = 0; i < N;++i)
			for (int j = 0;j < N;++j) {
				if (adjMatrix[i][j])
					g[i].push_back(j);
			}

		cout << "\nСписок смежности" << endl;

		for (int i = 0; i < N; i++)
		{
			for (int n : g[i])
				cout << n << "\t";

			cout << endl;
		}

		cout << endl;

		return g;
	}
```

Функция «edgeList» ~ возвращает и выводит на экран список ребер графа.

```
vector<vector<int>> edgeList() {

		vector<vector<int>> edge;

		for (int i = 0; i < N; i++)
		{
			for (int j = 0; j < N; j++) {

				if (adjMatrix[i][j]) {
					vector<int> v(2);
					v[0] = i;
					v[1] = j;
					edge.push_back(v);
				}
			}
		}

		cout << "\nСписок ребер" << endl;

		for (int i = 0; i < edge.size(); i++)
		{
			for (int n : edge[i])
				cout << n << "\t";

			cout << endl;

		}

		cout << endl;

		return edge;
		}
```

Функция «incidenceMatrix» ~ возвращает и выводит на экран матрицу инцидентности графа.

```
template<typename T>
bool MyStack<T>::empty()
{
	return _size == 0 ? true : false;
}
```

Функция «printStack» ~ выводит все элементы стека в консоль.

```
vector<vector<int>> incidenceMatrix() {

		int num_edge = 0, j_b = 0, col_nodirect = 0, col_direct = 0;
		for (int i = 0; i < N; i++) {
			for (int j = 0; j < N; j++) {
				if ((adjMatrix[i][j] == adjMatrix[j][i]) && (adjMatrix[i][j]))
					col_nodirect++;
				else if(adjMatrix[i][j])
					col_direct++;
			}
		}

		num_edge = col_direct + col_nodirect / 2;

		vector<vector<int>> matrix_incidence(N, vector<int>(num_edge));

		for (int i = 0; i < N; i++) {
			for (int j = i + 1; j < N; j++) {
				if ((adjMatrix[i][j] == adjMatrix[j][i]) && (adjMatrix[i][j]))
				{
					matrix_incidence[i][j_b] = 1;
					matrix_incidence[j][j_b] = 1;
					j_b++;
				}
				else if ((adjMatrix[i][j] == 1) && (adjMatrix[j][i] == 0))
				{
					matrix_incidence[i][j_b] = 1;
					matrix_incidence[j][j_b] = -1;
					j_b++;
				}
				else if ((adjMatrix[i][j] == 0) && (adjMatrix[j][i] == 1))
				{
					matrix_incidence[i][j_b] = -1;
					matrix_incidence[j][j_b] = 1;
					j_b++;
				}
			}
		}

		cout << "\nМатрица инцидентности" << endl;

		for (int i = 0; i < N; i++)
		{
			for (int j = 0; j < matrix_incidence[0].size(); j++)
			{
				cout << matrix_incidence[i][j] << "\t";
			}
			cout << endl;
		}

		cout << endl;
```

```
        return matrix_incidence;
    }
```

Функция «bfs» ~ осуществляет поиск пути в ширину между двумя вершинами.

```
    void bfs(int start_ind, int finish_ind) {

        queue<int> Queue;
        stack<Edge> Edges;
        Edge e;
        vector<int> nodes(N); // вершины графа
        int node;
        bool flag = false;
        Queue.push(start_ind); // помещаем в очередь первую вершину

        while (!Queue.empty())
        { // пока очередь не пуста
            node = Queue.front(); // извлекаем вершину
            Queue.pop();
            nodes[node] = 2; // отмечаем ее как посещенную

            if (node == finish_ind) {
                flag = true;
                break;
            }

            for (int j = 0; j < N; j++)
            { // проверяем для нее все смежные вершины
                if (adjMatrix[node][j] == 1 && nodes[j] == 0)
                { // если вершина смежная и не обнаружена
                    Queue.push(j); // добавляем ее в очередь
                    nodes[j] = 1; // отмечаем вершину как обнаруженную
                    e.begin = node; e.end = j;
                    Edges.push(e);
                }
            }
        }

        cout << "Поиск в ширину:" << endl;

        if (flag) {
            cout << "Путь между вершинами " << start_ind << " и " << finish_ind <<
" найден!" << endl;
            cout << finish_ind;

            int ind_edge = finish_ind;

            while (!Edges.empty()) {
                e = Edges.top();
                Edges.pop();
                if (e.end == ind_edge) {
                    ind_edge = e.begin;
                    cout << " <- " << ind_edge;
                }
            }

            cout << endl;
        }
        else
            cout << "Пути между вершинами " << start_ind << " и " << finish_ind <<
" нет!" << endl;

    }
```

Функция «dfs» ~ осуществляет поиск пути в глубину между двумя вершинами.

```
void dfs(int start_ind, int finish_ind) {

	stack<int> Stack;
	stack<Edge> Edges;
	Edge e;
	vector<int> nodes(N); // вершины графа
	int node;
	bool flag = false;

	Stack.push(start_ind); // помещаем в очередь первую вершину
	while (!Stack.empty())
	{ // пока стек не пуст

		node = Stack.top(); // извлекаем вершину
		Stack.pop();
		if (nodes[node] == 2) continue;
		nodes[node] = 2; // отмечаем ее как посещенную

		if (node == finish_ind) {
			flag = true;
			break;
		}

		for (int j = N-1; j >= 0; j--)
		{ // проверяем для нее все смежные вершины
			if (adjMatrix[node][j] == 1 && nodes[j] != 2)
			{ // если вершина смежная и не обнаружена
				Stack.push(j); // добавляем ее в стек
				nodes[j] = 1; // отмечаем вершину как обнаруженную
				e.begin = node; e.end = j;
				Edges.push(e);
			}
		}
	}

	cout << "Поиск в глубину:" << endl;

	if (flag) {
		cout << "Путь между вершинами " << start_ind << " и " << finish_ind <<
" найден!" << endl;
		cout << finish_ind;

		int ind_edge = finish_ind;

		while (!Edges.empty()) {
			e = Edges.top();
			Edges.pop();
			if (e.end == ind_edge) {
				ind_edge = e.begin;
				cout << " <- " << ind_edge;
			}
		}

		cout << endl;
	}
	else
		cout << "Пути между вершинами " << start_ind << " и " << finish_ind <<
" нет!" << endl;
}
```

Функция «main» ~ запускает генерацию разнообразных графов и тестирует на них методы поиска пути, замеряя время выполнения каждого метода.

```
int main()
{
	srand(time(0));
	setlocale(LC_ALL, "Rus");

	int num_test = 10;

	for (int i = 0; i < num_test; i++)
	{
		int min_vertex = 2 * (i + 1);
		int max_vertex = 4 * (i + 1);
		vector<vector<int>> graph = GenerateGraphs(2 * (i + 1), 4 * (i + 1), 2 * (i +
1), 5 * (i + 1), 1, false);
		GraphViews g(graph);
		g.adjacencyMatrix();
		g.incidenceMatrix();
		g.adjacencyList();
		g.edgeList();

		int rand_num = numGenerator(0, min_vertex - 1);

		chrono::high_resolution_clock::time_point start_dfs =
chrono::high_resolution_clock::now();
		g.dfs(rand_num, min_vertex);
		chrono::high_resolution_clock::time_point end1 =
chrono::high_resolution_clock::now();
		chrono::duration<double, milli> milli_diff_dfs = end1 - start_dfs;
		cout << "Время поиска в глубину: " << milli_diff_dfs.count() << " мс" << endl
<< endl;

		chrono::high_resolution_clock::time_point start_bfs =
chrono::high_resolution_clock::now();
		g.bfs(rand_num, min_vertex);
		chrono::high_resolution_clock::time_point end2 =
chrono::high_resolution_clock::now();
		chrono::duration<double, milli> milli_diff_bfs = end2 - start_bfs;
		cout << "Время поиска в ширину: " << milli_diff_bfs.count() << " мс" << endl;

	}

}
```

Один из результатов работы программы:

```
Матрица смежности
0       1       0       0       1       1       1       0
1       0       0       1       0       1       0       0
0       0       0       0       1       0       0       0
0       1       0       0       0       1       0       0
1       0       1       0       0       1       1       0
1       1       0       1       1       0       0       0
1       0       0       0       1       0       0       0
0       0       0       0       0       0       0       0


Матрица инцидентности
1       1       1       1       0       0       0       0       0       0
1       0       0       0       1       1       0       0       0       0
0       0       0       0       0       0       1       0       0       0
0       0       0       0       1       0       0       1       0       0
0       1       0       0       0       0       1       0       1       1
0       0       1       0       0       1       0       1       1       0
0       0       0       1       0       0       0       0       0       1
0       0       0       0       0       0       0       0       0       0
```

```
Список смежности
1       4       5       6
0       3       5
4
1       5
0       2       5       6
0       1       3       4
0       4
```

```
Список ребер
0       1
0       4
0       5
0       6
1       0
1       3
1       5
2       4
3       1
3       5
4       0
4       2
4       5
4       6
5       0
5       1
5       3
5       4
6       0
6       4
```

```
Поиск в глубину:
Путь между вершинами 0 и 6 найден!
6 <- 4 <- 5 <- 3 <- 1 <- 0
Время поиска в глубину: 3.137 мс

Поиск в ширину:
Путь между вершинами 0 и 6 найден!
6 <- 0
Время поиска в ширину: 2.7988 мс
```

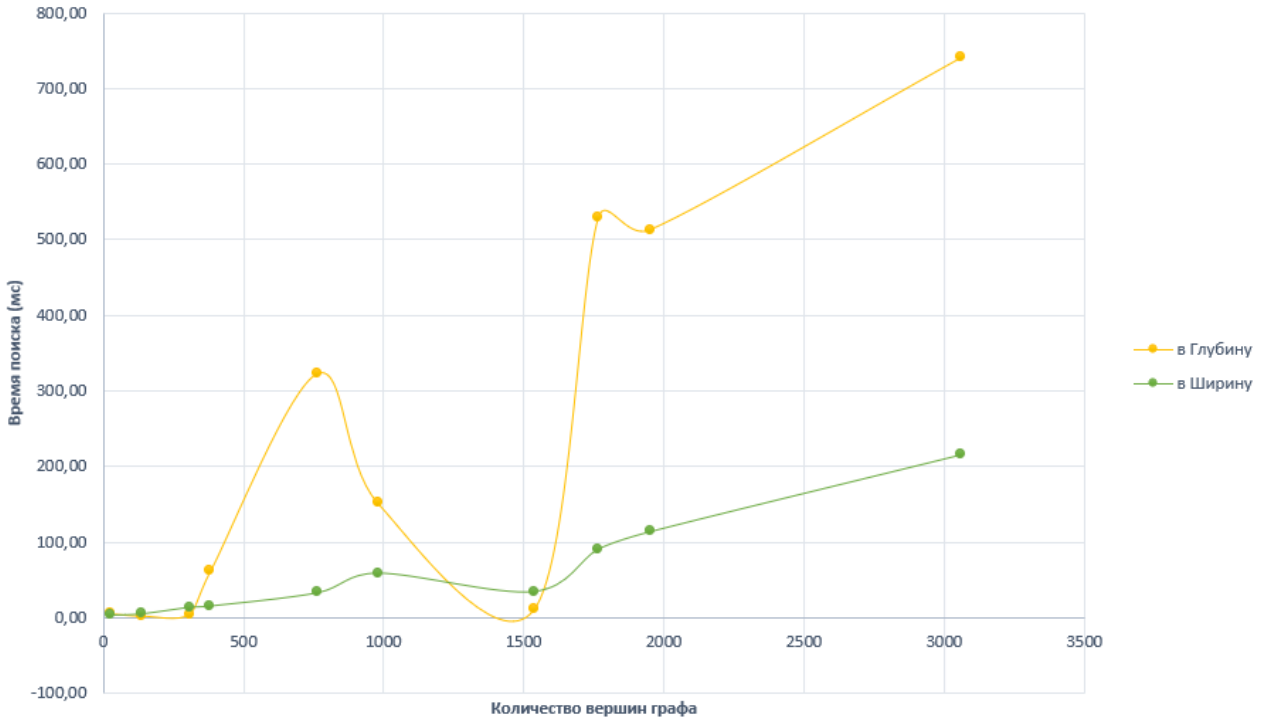

Рисунок 1 – График времени поиска пути относительно количества вершин графа.

Заключение

В этой лабораторной работе я познакомился с такой структурой, как граф. Мне удалось реализовать его на языке C++. Я разобрал методы поиска пути между вершинами графа и могу сделать вывод о том, что эффективность поиска очень зависит от расположения искомой вершины. Главное преимущество поиска в ширину состоит в том, что решение всегда будет найдено, если оно существует. Поиск в ширину просматривает все без исключения узлы, постепенно удаляясь от начального узла. Поиск в глубину резко теряет свою эффективность в случае, если искомый узел будет находиться наверху правой стороны графа, в то время как поиск начнется с левой. Поиск в ширину находит кротчайший путь между вершинами, а поиск в глубину – первый найденный.