МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт Информационных технологий

Кафедра Математического обеспечения и стандартизации
информационных технологий

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 6
по дисциплине
«Структуры и алгоритмы обработки данных»

Тема: «Основные алгоритмы работы с графами. Применение графа в решении практических задач.»

Выполнил студент группы ИКБО-11-22 Гришин А. В.

Принял преподаватель Скворцова Л. А.

Самостоятельная работа выполнена «__»_____202__г. *(подпись студента)*

«Зачтено» «__»_____202__г. *(подпись руководителя)*

Москва 2023

Содержание

1. Цель работы

Получение практических навыков по выполнению операций над структурой данных граф.

2. Задача №1

2.1 Постановка задачи

Разработать приложение, которое использует хеш-таблицу для организации прямого доступа к записям двоичного файла, реализованного в практической работе 2.

2.2 Требования

1. Разработать структуру данных «Граф» в соответствии с вариантом индивидуального задания. обеспечивающий хранение и работу со структурой данных «граф». Вид графа (ориентированный или не ориентированный) используемого в реализации графа выбрать в соответствии с алгоритмом варианта. Граф может быть взвешенным.

2. Разработать и реализовать алгоритмы операций управления графом:

1) ввод графа с клавиатуры, наполнение графа осуществлять с помощью добавления одного ребер;

1) вывод графа монитор, представляя его списками смежных вершин:

2) алгоритмы задач, определенные вариантом индивидуального задания.

3. Разработать программу, демонстрирующую работу всех методов класса. Произвести тестирование программы.

4. Составить отчет.

Индивидуальное задание:

№	Представление графа в памяти	Задачи
15	Список смежных вершин	1) Определить степень вершины графа. 2) Составить программу нахождения кратчайшего пути в графе от заданной вершины к другой заданной вершине методом «Дейкстры». Вывести этот путь.

2.2 Реализация

2.2.1 Структура графа

```
// Структура для представления графа
struct Graph {
    int neighbor; // соседние вершины
    int thisTop;  // текущая вершина
    int weight;   // Вес ребра
};
```

Рисунок 1. Структура графа

2.2.2 Функция №1

```
// Функция для добавления ребра в граф
void addEdge(vector<Graph>& graph, int thisTop, int neighbor, int weight) {
    Graph edge;
    edge.thisTop = thisTop;
    edge.neighbor = neighbor;
    edge.weight = weight;
    graph.push_back(edge);
}
```

Рисунок 2. Функция №1

Эта функция добавляет новое ребро в граф. Она создает новую структуру **Graph** с указанными параметрами (**thisTop**, **neighbor** и **weight**) и добавляет её в вектор **graph**.

2.2.3 Функция №2

```
// Функция для вывода графа
void printGraph(const vector<Graph>& graph) {
    for (int i = 0; i < graph.size(); ++i) {
        cout << "Список смежных вершин для вершины " << graph[i].thisTop << ": ";
        for (const auto& edge : graph) {
            if (edge.thisTop == graph[i].thisTop) {
                cout << edge.neighbor << "(" << edge.weight << ") ";
            }
        }
        cout << "\n";

        // Пропуск оставшихся рёбер для той же вершины
        while (i + 1 < graph.size() && graph[i].thisTop == graph[i + 1].thisTop) {
            i++;
        }
    }
}
```

Рисунок 3. Функция №2

Эта функция выводит представление графа в виде списка смежности. Она перебирает каждую вершину в графе (**graph**) и для каждой вершины выводит её смежные вершины вместе с соответствующими весами ребер.

2.2.4 Функция №3

```cpp
// Функция для определения степени вершины графа
int degreeOfVertex(const vector<Graph>& graph, int vertex) {
    int degree = 0;
    for (const auto& edge : graph) {
        if (edge.thisTop == vertex || edge.neighbor == vertex) {
            degree++;
        }
    }
    return degree;
}
```

Рисунок 4. Функция №3

Эта функция определяет степень вершины в графе, то есть количество ребер, инцидентных данной вершине. Путем прохода по списку ребер графа она подсчитывает количество ребер, связанных с заданной вершиной. Возвращает число, обозначающее степень вершины в графе.

2.2.5 Функция №4

```cpp
// Функция для нахождения кратчайшего пути методом Дейкстры
static void shortestPathDijkstra(vector<Graph>& graph, int startVertex, int endVertex) {
    const int INF = std::numeric_limits<int>::max(); // Бесконечность
    int numVertices = 0;
    for (const auto& edge : graph) {
        numVertices = max(numVertices, max(edge.thisTop, edge.neighbor));
    }
    numVertices++; // Число вершин в графе
    vector<int> distance(numVertices, INF); // Массив расстояний до каждой вершины
    vector<int> previous(numVertices, -1); // Массив предыдущих вершин
    distance[startVertex] = 0; // Расстояние от начальной вершины до самой себя равно 0
    vector<bool> visited(numVertices, false); // Посещенные вершины
    while (true) {
        int closestVertex = -1;
        for (int i = 0; i < numVertices; ++i) {
            if (!visited[i] && (closestVertex == -1 || distance[i] < distance[closestVertex])) {
                closestVertex = i;
            }
        }
        if (closestVertex == -1 || closestVertex == endVertex) {
            break; // Дошли до конечной вершины или все вершины посещены
        }
        visited[closestVertex] = true;
        for (const auto& edge : graph) {
            if (edge.thisTop == closestVertex) {
                int to = edge.neighbor;
                int weight = edge.weight;
                if (distance[closestVertex] + weight < distance[to]) {
                    distance[to] = distance[closestVertex] + weight;
                    previous[to] = closestVertex;
                }
            }
        }
    }
    if (distance[endVertex] == INF) {
        cout << "Нет пути от вершины " << startVertex << " к вершине " << endVertex << "\n";
        return;
    }
    // Восстановление пути
    vector<int> path;
    for (int at = endVertex; at != -1; at = previous[at]) {
        path.push_back(at);
    }
    reverse(path.begin(), path.end());
    cout << "Кратчайший путь от вершины " << startVertex << " к вершине " << endVertex << ": ";
    for (int vertex : path) {
        cout << vertex << " ";
    }
    cout << "\n";
}
```

Рисунок 5. Функция №4

Данная функция реализует алгоритм поиска кратчайшего пути от начальной вершины до конечной в графе с весами на ребрах методом Дейкстры. Она использует массивы для хранения расстояний до каждой вершины и предыдущих вершин на пути к ним. Алгоритм просматривает все вершины графа, вычисляя расстояния до них от начальной вершины, обновляя их при необходимости и восстанавливая кратчайший путь по завершении. Если путь существует, выводит на экран кратчайший путь от начальной вершины до конечной.

2.3 Результаты тестирования

В качестве примера был взят граф номер 5.

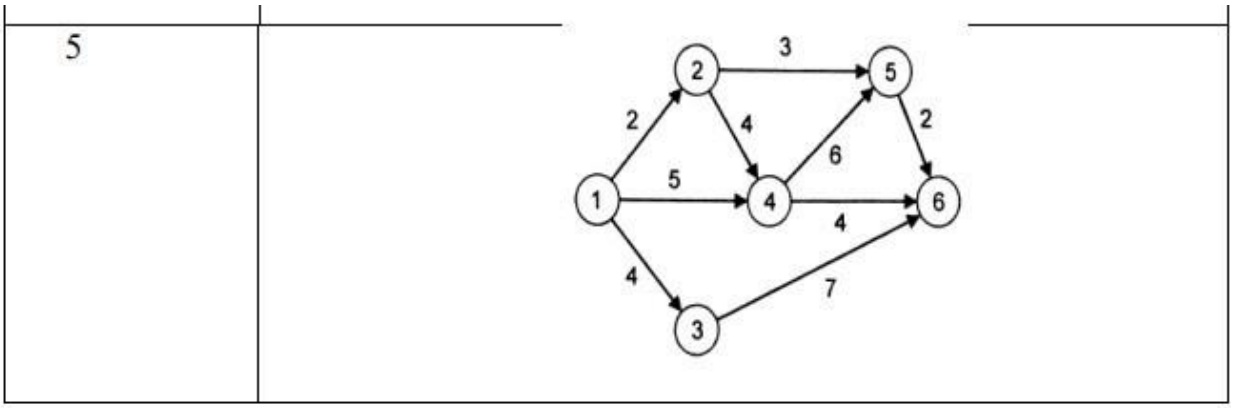

Рисунок 6. Выбранный вариант

```
// Добавление ребер
addEdge(graph, 1, 2, 2);
addEdge(graph, 1, 3, 4);
addEdge(graph, 1, 4, 5);
addEdge(graph, 2, 4, 4);
addEdge(graph, 2, 5, 3);
addEdge(graph, 3, 6, 7);
addEdge(graph, 4, 6, 4);
addEdge(graph, 4, 5, 6);
addEdge(graph, 5, 6, 2);
```

Рисунок 7. Добавление ребер

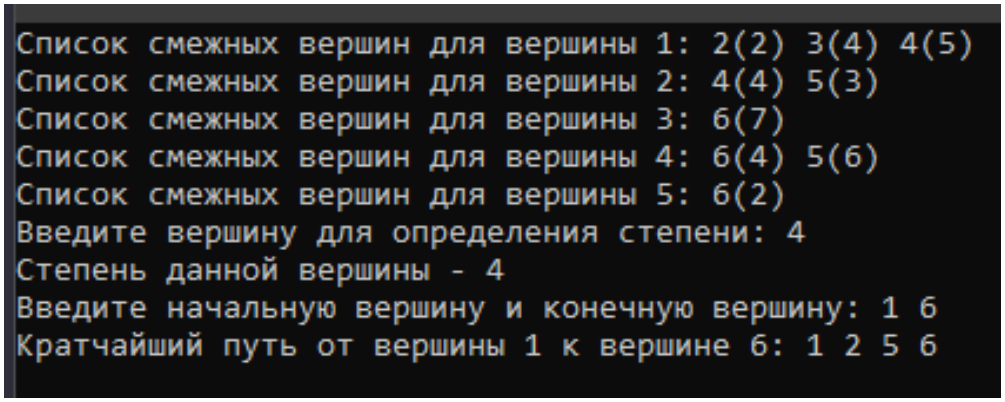

Рисунок 8. Результат тестирования

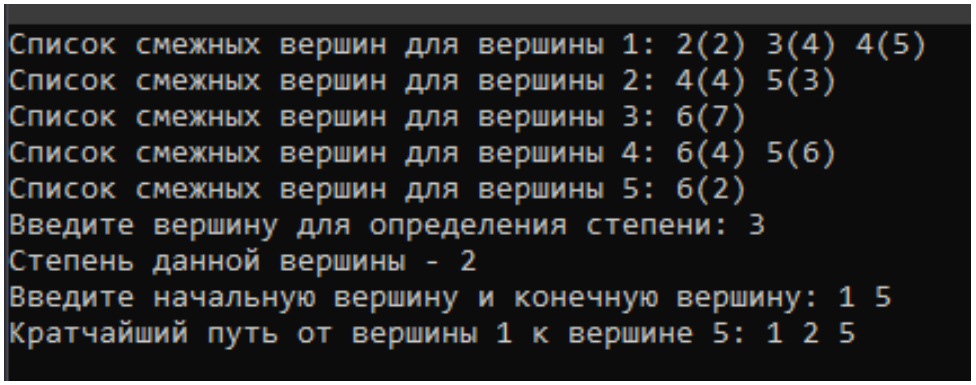

Рисунок 8. Результат тестирования

3 Исходный код программы

Листинг 1 – Исходный код программы для Задания 1

```
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>
#include <limits>
using namespace std;

// Структура для представления графа
struct Graph {
	int neighbor; // соседние вершины
	int thisTop; // текущая вершина
	int weight; // Вес ребра
};

// Функция для добавления ребра в граф
void addEdge(vector<Graph>& graph, int thisTop, int neighbor, int weight) {
	Graph edge;
	edge.thisTop = thisTop;
	edge.neighbor = neighbor;
```

```
	edge.weight = weight;
	graph.push_back(edge);
}
// Функция для вывода графа
void printGraph(const vector<Graph>& graph) {
	for (int i = 0; i < graph.size(); ++i) {
		cout << "Список смежных вершин для вершины " << graph[i].thisTop << ":
";
		for (const auto& edge : graph) {
			if (edge.thisTop == graph[i].thisTop) {
				cout << edge.neighbor << "(" << edge.weight << ") ";
			}
		}
		cout << "\n";
		// Пропуск оставшихся рёбер для той же вершины
		while (i + 1 < graph.size() && graph[i].thisTop == graph[i + 1].thisTop)
{
			i++;
		}
	}
}

// Функция для определения степени вершины графа
int degreeOfVertex(const vector<Graph>& graph, int vertex) {
	int degree = 0;
	for (const auto& edge : graph) {
		if (edge.thisTop == vertex || edge.neighbor == vertex) {
			degree++;
		}
	}
	return degree;
}

// Функция для нахождения кратчайшего пути методом Дейкстры
static void shortestPathDijkstra(vector<Graph>& graph, int startVertex, int
endVertex) {
	const int INF = std::numeric_limits<int>::max(); // Бесконечность
	int numVertices = 0;
	for (const auto& edge : graph) {
		numVertices = max(numVertices, max(edge.thisTop, edge.neighbor));
	}
	numVertices++; // Число вершин в графе

	vector<int> distance(numVertices, INF); // Массив расстояний до каждой вершины
	vector<int> previous(numVertices, -1); // Массив предыдущих вершин

	distance[startVertex] = 0; // Расстояние от начальной вершины до самой себя
равно 0

	vector<bool> visited(numVertices, false); // Посещенные вершины

	while (true) {
		int closestVertex = -1;
		for (int i = 0; i < numVertices; ++i) {
			if (!visited[i] && (closestVertex == -1 || distance[i] <
distance[closestVertex])) {
				closestVertex = i;
			}
		}

		if (closestVertex == -1 || closestVertex == endVertex) {
			break; // Дошли до конечной вершины или все вершины посещены
		}

		visited[closestVertex] = true;
```

```
            for (const auto& edge : graph) {
                  if (edge.thisTop == closestVertex) {
                        int to = edge.neighbor;
                        int weight = edge.weight;
                        if (distance[closestVertex] + weight < distance[to]) {
                              distance[to] = distance[closestVertex] + weight;
                              previous[to] = closestVertex;
                        }
                  }
            }
      }

      if (distance[endVertex] == INF) {
            cout << "Нет пути от вершины " << startVertex << " к вершине " <<
endVertex << "\n";
            return;
      }

      // Восстановление пути
      vector<int> path;
      for (int at = endVertex; at != -1; at = previous[at]) {
            path.push_back(at);
      }

      reverse(path.begin(), path.end());

      cout << "Кратчайший путь от вершины " << startVertex << " к вершине " <<
endVertex << ": ";
      for (int vertex : path) {
            cout << vertex << " ";
      }
      cout << "\n";
}

int main() {
      setlocale(LC_ALL, "Russian");
      locale::global(std::locale("en_US.UTF-8"));
      // Пример использования
      vector<Graph> graph;
      // Добавление ребер
      addEdge(graph, 1, 2, 2);
      addEdge(graph, 1, 3, 4);
      addEdge(graph, 1, 4, 5);
      addEdge(graph, 2, 4, 4);
      addEdge(graph, 2, 5, 3);
      addEdge(graph, 3, 6, 7);
      addEdge(graph, 4, 6, 4);
      addEdge(graph, 4, 5, 6);
      addEdge(graph, 5, 6, 2);

      // Вывод содержимого графа
      printGraph(graph);

      int vertex;
      cout << "Введите вершину для определения степени: ";
      cin >> vertex;
      cout << "Степень данной вершины - " << degreeOfVertex(graph, vertex) << endl;

      int source, destination, maxWeight;
      cout << "Введите начальную вершину и конечную вершину: ";
      cin >> source >> destination;

      shortestPathDijkstra(graph, source, destination);
      return 0;
}
```

4. Вывод

В результате выполнения работы я получил практические навыки по выполнению операций над структурой данных граф.