МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Государственное автономное образовательное учреждение высшего образования
«Новосибирский государственный технический университет»

Отчет по контрольной работе №2

«Коллекция данных – граф»

Вариант 8

Выполнила студентка группы ДТ-160

Буянкина Елизавета Алексеевна

Проверил преподаватель

Романенко Т.А.

Новосибирск – 2024

1. Задание

Спроектировать и реализовать АТД «Граф». АТД «Простой граф» реализуется в виде шаблонного класса. Параметрами шаблона является тип веса ребра графа. Интерфейс АТД «Граф» включает операции:

Конструктор пустого графа для заданных числа вершин, типа, и формы представления

V( ) - опрос числа вершин в графе,

E( ) - опрос числа ребер в графе,

Insert(v1,v2) вставка ребра, соединяющего вершины v1, v2,

Delete (v1,v2)удаление ребра, соединяющего вершины v1, v2,

Edge(v1,v2) опрос наличия ребра, соединяющего вершины v1, v2,

SetEdge(v1,v2, data) задание параметров ребра,

Task() решение задачи по варианту

Show() вывод структуры графа на экран.

Вариант 8

Реализация АТД « Взвешенный орграф». Граф представлен в виде списков смежности (L-граф). Определение радиуса и списка вершин для соответствующего радиусу пути на основе алгоритма Дейкстры. (радиус – минимальный эксцентриситет в графе, путь - последовательность вершин, лежащих на пути с суммарным весом ребер, равным радиусу).

2. Формат АТД

Граф задается в виде пары множеств G = (V, E), где V - конечное множество элементов, E - множество бинарных отношений между элементами. Элементы называются вершинами, а бинарные отношения - ребрами или дугами.

В ориентированном графе (орграфе) бинарные отношения между вершинами упорядочены, то есть отношения между парой вершин u, v (u, v) $\neq$ (v, u). В орграфе такие отношения называются дугами. Дуги направлены от одной вершины к другой.

2.1 Данные

Параметры:

WeightType – тип веса ребра графа

V – количество вершин графа

E – количество дуг графа

Структура хранения коллекции:

Представление графа G = (V, E) в виде списков смежности использует массив Adj из |V| списков, по одному для каждой вершины. Для каждой вершины односвязный список смежных

вершин Adj [u] содержит в произвольном порядке все смежные с ней вершины v, для которых существует ребро (u, v)E.

2.2 Операции:

1) Конструктор с заданным числом вершин Graph(size_t v)

   Вход: v – число вершин

   Предусловия: нет

   Процесс: создание графа с количеством вершин v

   Выход: нет

   Постусловия: создан граф с числом вершин V = v, числом дуг E = 0

2) Опрос числа вершин size_t V()

   Вход: нет

   Предусловия: нет

   Процесс: возврат текущего количества вершин V, содержащихся в графе

   Выход: количество вершин V

   Постусловия: нет

3) Опрос числа вершин size_t E()

   Вход: нет

   Предусловия: нет

   Процесс: возврат текущего количества дуг E, содержащихся в графе

   Выход: количество дуг E

   Постусловия: нет

4) Вставка дуги, соединяющей вершины v1, v2 Insert(int v1, int v2)

   Вход: v1 – начальная вершина, v2 – конечная вершина

   Предусловия: дуга (v1, v2) не существует

   Процесс: добавление дуги (v1, v2)

   Выход: булевое значение false, если дуга уже существует, иначе true

   Постусловия: добавлена дуга(v1, v2), E = E + 1

5) Удаление дуги, соединяющей вершины v1, v2 Insert(int v1, int v2)

   Вход: v1 – начальная вершина, v2 – конечная вершина

   Предусловия: дуга (v1, v2) существует

   Процесс: добавление дуги (v1, v2)

   Выход: булевое значение false, если дуга не существует, иначе true

   Постусловия: удалена дуга(v1, v2), E = E - 1

6) Опрос наличия дуги, соединяющей вершины v1, v2 Edge(int v1, int v2)

Вход: v1 – начальная вершина, v2 – конечная вершина

Предусловия: нет

Процесс: проверка наличия дуги (v1, v2)

Выход: булевое значение true, в случае наличия дуги, иначе false

Постусловия: нет

7) Установка веса weight для дуги (v1, v2) SetEdge(int v1, int v2, WeightType weight)

Вход: v1 – начальная вершина, v2 – конечная вершина, weight – вес дуги

Предусловия: дуга (v1, v2) существует

Процесс: установка веса weight для дуги (v1, v2)

Выход: булевое значение false, если дуга не существует, иначе true

Постусловия: вес дуги (v1, v2) = weight

8) Определение центра орграфа Task()

Вход: нет

Предусловия: граф не пуст

Процесс: определение вершины, соответствующей минимальному эксцентриситету в графе, и соответствующего пути

Выход последовательность вершин в виде массива, лежащих на пути с суммарным весом дуг, равным радиусу, или генерация исключения при невыполнении предусловия

Постусловия: нет

9) Вывод графа на экран Show()

Вход: нет

Предусловия: нет

Процесс: вывод структуры графа на экран

Выход: нет

Постусловия: нет

3. Справочное определение класса для коллекции «Граф»

```
template<class WeightType>
class Graph {
      Graph(size_t v); // конструктор с заданным числом вершин
      size_t V(); // опрос числа вершин в графе
      size_t E(); // опрос числа ребер в графе
      bool Insert(int v1, int v2);// вставка дуги, соединяющей вершины v1, v2
      bool Delete(int v1, int v2); // удаление дуги, соединяющей вершины v1, v2
      bool Edge(int v1, int v2); //опрос наличия дуги, соединяющей вершины v1, v2
      bool SetEdge(int v1, int v2, WeightType weight); // установка веса weight
для дуги (v1, v2)
      vector<int> Task(); // определение центра орграфа
      void Show(); // вывод графа на экран
};
```

4. Описание алгоритма, заданного вариантом

Эксцентриситет вершины – это длина максимального из наикратчайших путей до данной вершины от всех остальных вершин графа. Центром орграфа называется вершина с минимальным эксцентриситетом, т.е. это вершина, для которой максимальное расстояние (длина пути) от других вершин минимально.

Алгоритм Дейкстры находит кратчайшие пути от заданной вершины до всех остальных в графе без ребер отрицательного веса. Основная идея алгоритма состоит в том, что для каждой вершины графа отмечается минимальное известное расстояние до этой вершины, которое хранится в множестве D. Изначально отметки о расстоянии неизвестны – ∞. Алгоритм на каждом шаге при нахождении меньшего значения обновляет информацию о расстоянии на основании данных из структуры смежности. Посещенные вершины хранятся в отдельном множестве. Выбор следующей не посещённой вершины для перехода производится на основе информации из множества D, где выбирается вершина с минимальным весом. После прохождения алгоритма в множестве D будут хранится данные о минимальном весе и предыдущей вершине для каждой вершины в графе. Таким образом, для каждой вершины можно восстановить пройденный к ней путь, обращаясь к данным о предыдущих узлах.

Пройдемся по всем вершинам графа и найдем минимальные пути до остальных вершин. После каждой итерации будем отмечать для каждой вершины наибольшее из найденных минимальных расстояний. В конечном итоге выберем вершину с минимальным эксцентриситетом. Она и будет центром графа.

5. Список использованной литературы

1. Романенко, Т. А. Программные коллекции данных. Проектирование и реализация : учебник для вузов / Т. А. Романенко. — Санкт-Петербург : Лань, 2021. — 152 с.

2. Фрэнк М. Каррано, Джанет Дж. Причард. Абстракция данных и решение задач на C++. Стены и зеркала. - М. - СПб – Киев: «Вильямс», 2003 г. – 848 с.

3. Т. Кормен, Ч. Лейзерсон, Р. Ривест Алгоритмы. Анализ и построение. - М: «БИНОМ», 2000 г. – 960 с.

4. Кубенский А.А. Структуры и алгоритмы обработки данных: объектно-ориентированный подход и реализация на C++. – СПб.: БХВ-Петербург, 2004 г. – 464 с.

5. Коллинз У.Дж. Структуры данных и стандартная библиотека шаблонов. – М.: ООО «Бином-Пресс», 2004. – 624 с.

6. Роберт Сэджвик. Фундаментальные алгоритмы на C++. Части 1-5. - М: «DiaSoft», 2001 г. – 688 с.

6. Приложение с текстами программ

6.1 Graph.h

```
#include <iostream>
#include <forward_list>
#include <queue>
using namespace std;
template<class WeightType>
class Graph {
private:
	class Arc { // Ребро
	public:
		int to; // Конечная вершина
		WeightType weight{}; // Вес ребра
		Arc(int v) : to(v) {}
	};
	size_t e = 0; // Количество ребер
	size_t v = 0; // Количество вершин
	vector<forward_list<Arc>> Lgraph;
	Arc* findArc(int v1, int v2) { // Поиск ребра
		if (Lgraph.size() <= v1) { // Такой вершины нет
			return nullptr;
		}
		forward_list<Arc>& vertex = Lgraph[v1];
		auto iter = find_if(vertex.begin(), vertex.end(), [&v2](Arc arc) { return
arc.to == v2; });
		if (iter == vertex.end()) { // Ребро не найдено
			return nullptr;
		}
		return &(*iter);
	}
	vector<pair<WeightType, vector<int>>> exstrisitetsForGraph() { // Поиск
эксцентриситетов взвешенного графа
		vector<pair<WeightType, vector<int>>> exstrisitetsForGraph(v);
		for (int from = 0; from < Lgraph.size(); from++) { // Проходимся по всем
вершинам
			queue<int> q;
			vector<WeightType> distance(v, numeric_limits<WeightType>::max()); //
максимальное расстояние до вершин
			vector<vector<int>> paths(v); // путь с максимальным расстоянием до
вершин
			vector<bool> visited(v, false);
			visited[from] = true;
			distance[from] = {};
			paths[from].push_back(from);
			q.push(from);
			while (!q.empty()) { // Пока не пройдемся по всем вершинам
				int v = q.front();
				q.pop();
				for (const Arc& arc : Lgraph[v]) {
					if (!visited[arc.to] || distance[arc.to] > distance[v] +
arc.weight) { // если вершину еще не посещали или посещали более долгим путем
						distance[arc.to] = distance[v] + arc.weight;
						paths[arc.to] = paths[v];
						paths[arc.to].push_back(arc.to);
						visited[arc.to] = true;
						q.push(arc.to);
					}
				}
			}
```

```
                  for (int i = 0; i < Lgraph.size(); i++) {
                        if   (exstrisitetsForGraph[i].first   <   distance[i])   {   //
максимальное значение в каждом столбце
                              exstrisitetsForGraph[i]      =      make_pair(distance[i],
paths[i]);
                        }
                  }
            }
            return exstrisitetsForGraph;
      }
public:
      Graph(size_t v);
      size_t V(); // опрос числа вершин в графе
      size_t E(); // опрос числа ребер в графе
      bool Insert(int v1, int v2);//  вставка ребра, соединяющего вершины v1, v2
      bool Delete(int v1, int v2); // удаление ребра, соединяющего вершины v1, v2
      bool Edge(int v1, int v2); //   опрос наличия ребра, соединяющего вершины v1, v2
      bool SetEdge(int v1, int v2, WeightType weight); // задание параметров ребра
      vector<int> Task(); //    решение задачи по варианту
      void Show(); // вывод графа на экран
};
template<class WeightType>
bool Graph<WeightType>::Insert(int v1, int v2) {
      if (Lgraph.size() <= v1 || v1 == v2 || Edge(v1, v2)) {
            return false;
      }
      Lgraph[v1].push_front(Arc(v2));
      e++;
      return true;
}
template<class WeightType>
bool Graph<WeightType>::Delete(int v1, int v2) {
      if (!Edge(v1, v2)) {
            return false;
      }
      forward_list<Arc>& vertex = Lgraph[v1];
      vertex.remove_if([&v2](Arc arc) { return arc.to == v2; });
      e--;
      return true;
}
template<class WeightType>
bool Graph<WeightType>::Edge(int v1, int v2) {
      return findArc(v1, v2) != nullptr;
}
template<class WeightType>
bool Graph<WeightType>::SetEdge(int v1, int v2, WeightType weight) {
      Arc* arc = findArc(v1, v2);
      if (arc == nullptr) {
            return false;
      }
      arc->weight = weight;
      return true;
}
template<class WeightType>
vector<int> Graph<WeightType>::Task() {
      vector<pair<WeightType, vector<int>>> exstr = exstrisitetsForGraph();
      if (exstr.empty()) {
            throw exception("Пустой граф");
      }
      auto radius = min_element( // радиус – минимальный эксцентриситет в графе
            exstr.begin(), exstr.end(),
```

```
            [](const   pair<WeightType,   vector<int>>&   p1,   const   pair<WeightType,
vector<int>>& p2) {return p1.first < p2.first; });
      return radius->second;
}
template<class WeightType>
void Graph<WeightType>::Show() {
      for (int i = 0; i < Lgraph.size(); i++) {
            cout << i;
            for (const Arc& arc : Lgraph[i]) {
                  cout << " " << arc.to << "," << arc.weight;
            }
            cout << endl;
      }
}
template<class WeightType>
Graph<WeightType>::Graph(size_t v){
      this->v = v;
      Lgraph.clear();
      Lgraph.resize(v, {});
}
template<class WeightType>
size_t Graph<WeightType>::V() {
      return v;
}
template<class WeightType>
size_t Graph<WeightType>::E() {
      return e;
}
```

6.2 Menu.cpp

```
#include <iostream>
#include "graph.h"
using namespace std;
int main() {
      setlocale(LC_ALL, "Russian");
      Graph<int> g = Graph<int>(0);
      int n;
      while (true) {
            cout << endl << "Меню. Выберите действие:" << endl << endl;
            cout << "0. Создать граф" << endl;
            cout << "1. Вставка" << endl;
            cout << "2. Удаление" << endl;
            cout << "3. Число вершин" << endl;
            cout << "4. Число ребер" << endl;
            cout << "5. Опрос наличия ребра" << endl;
            cout << "6. Задание параметров ребра" << endl;
            cout << "7. Вывод структуры графа на экран" << endl;
            cout << "8. Определение центра и списка вершин для соответствующего радиусу
пути на основе алгоритма Дейкстры" << endl;
            cout << endl;
            cin >> n;
            cout << endl;
            try {
                  switch (n) {
                  case 0: {
                        int v;
                        cin >> v;
                        g = Graph<int>(v); break;
                  }
                  case 1: {
                        int v1, v2;
```

```
                        cin >> v1 >> v2;
                        cout << g.Insert(v1, v2) << endl; break;
                }
                case 2: {
                        int v1, v2;
                        cin >> v1 >> v2;
                        cout << g.Delete(v1, v2) << endl; break;
                }
                case 3: {
                        cout << g.V() << endl; break;
                }
                case 4: {
                        cout << g.E() << endl; break;
                }
                case 5: {
                        int v1, v2;
                        cin >> v1 >> v2;
                        cout << g.Edge(v1, v2) << endl;
                        break;
                }
                case 6: {
                        int v1, v2, weight;
                        cin >> v1 >> v2 >> weight;
                        cout << g.SetEdge(v1, v2, weight) << endl; break;
                }
                case 7: {g.Show();break;}
                case 8: {
                        vector<int> radius = g.Task();
                        cout << "Центр: " << *radius.rbegin() << endl;
                        cout << "Путь:";
                        for (const int& vertex : radius) {
                                cout << " " << vertex;
                        }
                        break;
                }
                }
        }
        catch (exception e) {
                cout << e.what() << endl;
        }
    }
}
```