Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет информатики и радиоэлектроники»

Институт информационных технологий

Факультет компьютерных технологий

Лабораторная работа № 13

«ПОИСК МАРШРУТОВ НА ОРИЕНТИРОВАННЫХ ГРАФАХ»

Вариант 1

Выполнил: студент гр. 281073

Буйновский М.В.

Проверил: Потоцкий Д.С.

Минск 2023

**Цель работы:** научиться обрабатывать элементы ориентированных графов, находить в них заданные маршруты.

**Задание:** представить ориентированный граф, состоящий из 7-10 вершин, с помощью матрицы смежности, а затем выполнить следующие операторы над его элементами. Указать вершину-источник, а затем решить следующие задачи.

1. Кратчайшие пути от вершины-источника до всех вершин орграфа на основе алгоритма Дейкстры.

**Код программы:**

**dijkstra.h**

#include <algorithm>

#include <iomanip>

#include <iostream>

#include <unordered\_map>

#include <vector>

#include "priority\_queue.h"

template <typename Graph> class Dijkstra

{

private:

// Reconstruct a path from start to goal

static std::vector<std::pair<typename Graph::location\_t, typename Graph::cost\_t>> reconstruct\_path(

const typename Graph::location\_t &start,

const typename Graph::location\_t &goal,

std::unordered\_map<typename Graph::location\_t, typename Graph::location\_t> &came\_from,

std::unordered\_map<typename Graph::location\_t, typename Graph::cost\_t> &cost\_so\_far

)

{

std::vector<std::pair<typename Graph::location\_t, typename Graph::cost\_t>> path;

typename Graph::location\_t current = goal;

if (came\_from.find(goal) == came\_from.end())

{

return path; // no path can be found

}

while (current != start)

{

path.push\_back({current, cost\_so\_far[current]});

current = came\_from[current];

}

path.push\_back({start, (typename Graph::cost\_t)0});

// reconstructed path will start from end, reverse

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

return path;

}

public:

static void show(

std::vector<std::pair<typename Graph::location\_t, typename Graph::cost\_t>> path,

const typename Graph::location\_t &start,

const typename Graph::location\_t &goal

)

{

std::cout << "\033[31m" << std::setw(2) << start << "\033[0m"

<< " -> "

<< "\033[36m" << std::setw(2) << goal << "\033[0m"

<< " | ";

if (path.size() == 0)

{

std::cout << "No path can be found from "

<< "\033[36m"

<< "'" << start << "'"

<< "\033[0m"

<< " to "

<< "\033[36m"

<< "'" << goal << "'"

<< "\033[0m"

<< "." << std::endl;

return;

}

else if (path.size() == 1)

{

std::cout << "Moving to the same edge. (cost: "

<< "\033[33m" << std::setw(2) << path[0].second << "\033[0m"

<< ")." << std::endl;

return;

}

std::cout << "\033[31m" << std::setw(2) << start << "\033[0m";

for (size\_t i = 1; i < path.size(); i++)

{

// cost from the first edge to current is stored, so subtract cost of prev from current

std::cout << " -("

<< "\033[33m" << std::setw(2) << path[i].second - path[i - 1].second << "/" << std::setw(3)

<< path[i].second << "\033[0m"

<< ")-> ";

if (path[i].first == goal)

{

std::cout << "\033[32m";

}

else

{

std::cout << "\033[36m";

}

std::cout << std::setw(2) << path[i].first;

std::cout << "\033[0m";

}

std::cout << std::endl;

}

static std::vector<std::pair<typename Graph::location\_t, typename Graph::cost\_t>> search(

Graph &graph, const typename Graph::location\_t &start, const typename Graph::location\_t &goal

)

{

std::unordered\_map<typename Graph::location\_t, typename Graph::cost\_t> cost\_so\_far;

PriorityQueue<typename Graph::location\_t, typename Graph::cost\_t> frontier;

std::unordered\_map<typename Graph::location\_t, typename Graph::location\_t> came\_from;

frontier.push(start, typename Graph::cost\_t(0));

came\_from[start] = start;

cost\_so\_far[start] = typename Graph::cost\_t(0);

while (!frontier.empty())

{

typename Graph::location\_t current = frontier.pop();

if (current == goal)

{

break;

}

for (typename Graph::location\_t next : graph.neighbors(current))

{

typename Graph::cost\_t new\_cost = cost\_so\_far[current] + graph.cost(current, next);

if (cost\_so\_far.find(next) == cost\_so\_far.end() || new\_cost < cost\_so\_far[next])

{

cost\_so\_far[next] = new\_cost;

came\_from[next] = current;

frontier.push(next, new\_cost);

}

}

}

return Dijkstra<Graph>::reconstruct\_path(start, goal, came\_from, cost\_so\_far);

}

};

**graph.h**

#include <algorithm>

#include <iostream>

#include <map>

#include <stdexcept>

#include <string>

#include <vector>

class Graph

{

public:

// Types for graphs

typedef size\_t location\_t;

typedef int cost\_t;

// Some constants

static const Graph::location\_t INF = 0;

private:

// Map of edges that stores every available edge from it

std::map<location\_t, std::vector<std::pair<Graph::location\_t, Graph::cost\_t>>> edges;

public:

// Constructor

Graph(std::vector<std::vector<cost\_t>> matrix)

{

// Convert matrix representation of a graph to map

for (location\_t i = 0; i < matrix.size(); i++)

{

if (matrix.size() != matrix[i].size())

{

throw std::invalid\_argument(

"Matrix representation of a graph must be a square. Matrix size: " + std::to\_string(matrix.size()) + "; Matrix[i] size: " + std::to\_string(matrix[i].size()));

}

for (location\_t j = 0; j < matrix[i].size(); j++)

{

if (matrix[i][j] > 0)

{

edges[i].push\_back({j, matrix[i][j]});

}

}

}

}

// Get all neighbors

std::vector<Graph::location\_t> neighbors(int id)

{

std::vector<Graph::location\_t> result;

for (auto location : edges[id])

{

result.push\_back(location.first);

}

return result;

}

// Print graph to a console

void show()

{

for (auto it : edges)

{

std::cout << "\033[36m" << it.first << "\033[0m" << std::endl;

for (location\_t i = 0; i < it.second.size(); i++)

{

std::cout << " -("

<< "\033[33m" << it.second[i].first << "\033[0m"

<< ")-> "

<< "\033[36m" << it.second[i].second << "\033[0m" << std::endl;

}

}

}

// Get a cost of moving from one edge to another

Graph::cost\_t cost(Graph::location\_t first, Graph::location\_t second)

{

auto result = std::find\_if(

edges[first].begin(),

edges[first].end(),

[second](const std::pair<Graph::location\_t, Graph::cost\_t> &element) { return element.first == second; }

);

if (result == edges[first].end())

{

throw std::invalid\_argument(

"First edge '" + std::to\_string(first) + "' does not a have a path to edge '" + std::to\_string(second)

+ "'."

);

}

return result->second;

}

};

**priority\_queue.h**

#include <queue>

#include <utility>

#include <vector>

// Modified version of std::priority\_queue. Use vector instead of heap and change priority direction

template <typename T, typename priority\_t> struct PriorityQueue

{

public:

typedef std::pair<priority\_t, T> PQElement;

std::priority\_queue<PQElement, std::vector<PQElement>, std::greater<PQElement>> elements;

inline bool empty() const

{

return elements.empty();

}

inline void push(T item, priority\_t priority)

{

elements.emplace(priority, item);

}

T pop()

{

T best\_item = elements.top().second;

elements.pop();

return best\_item;

}

};

**main.cpp**

#include <chrono>

#include <iostream>

#include <random>

#include <stdlib.h>

#include <vector>

#include "dijsktra.h"

#include "graph.h"

int main(void)

{

// Some random C++ bullshit

std::random\_device rd;

// seed value is designed specifically to make initialization

// parameters of std::mt19937 (instance of std::mersenne\_twister\_engine<>)

// different across executions of application

std::mt19937::result\_type seed

= rd()

^ ((std::mt19937::result\_type

)std::chrono::duration\_cast<std::chrono::seconds>(std::chrono::system\_clock::now().time\_since\_epoch())

.count()

+ (std::mt19937::result\_type)std::chrono::duration\_cast<std::chrono::microseconds>(

std::chrono::high\_resolution\_clock::now().time\_since\_epoch()

)

.count());

std::mt19937 gen(seed);

std::uniform\_int\_distribution<Graph::cost\_t> location\_distribution(1, 12);

std::uniform\_real\_distribution<double> percentage\_distribution(0, 1);

std::cout << "Enter amount of edges: ";

unsigned edges\_amount;

std::cin >> edges\_amount;

std::cout << "Enter chance of an edge: ";

double edge\_chance;

std::cin >> edge\_chance;

std::vector<std::vector<Graph::cost\_t>> matrix(edges\_amount);

for (unsigned i = 0; i < edges\_amount; i++)

{

// matrix.push\_back(std::vector<Graph::cost\_t>(edges\_amount));

for (unsigned j = 0; j < edges\_amount; j++)

{

if (i != j && percentage\_distribution(gen) <= edge\_chance)

{

matrix[i].push\_back(location\_distribution(gen));

}

else

{

matrix[i].push\_back(Graph::INF);

}

}

}

Graph graph(matrix);

std::cout << "Graph: " << std::endl;

graph.show();

Graph::location\_t start = 0;

std::cout << "Path: " << std::endl;

for (Graph::location\_t i = 0; i < matrix.size(); i++)

{

auto path = Dijkstra<Graph>::search(graph, start, i);

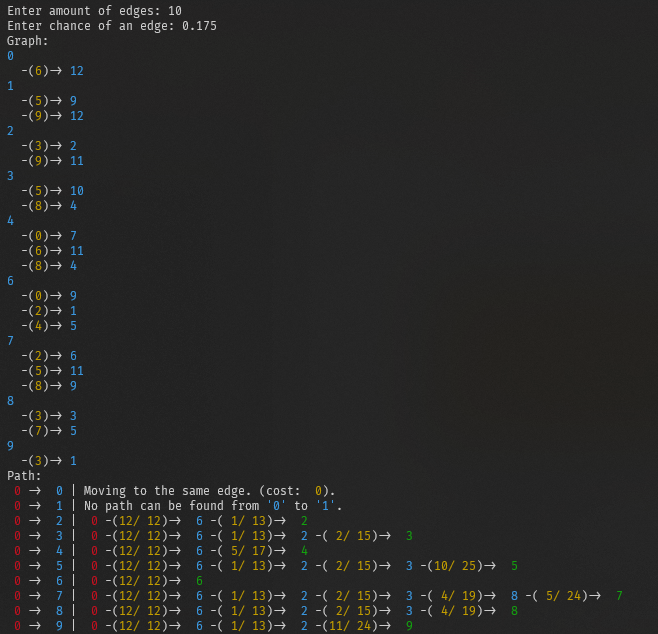
Dijkstra<Graph>::show(path, start, i);

}

return EXIT\_SUCCESS;

}

**Скриншот работы программы, демонстрация результатов:**

****

**Контрольные вопросы:**

**1. Для решения какой задачи на ориентированном графе удобно использовать алгоритм Дейкстры?**

Алгоритм Дейкстры удобно использовать для решения задачи нахождения кратчайшего пути с одним источником на ориентированном графе. Этот алгоритм позволяет найти кратчайшие пути от заданной исходной вершины ко всем остальным вершинам графа.

**2. Для решения какой задачи на ориентированном графе удобно использовать алгоритм Флойда?.**

Алгоритм Флойда удобно использовать для решения задачи нахождения кратчайших путей между всеми парами вершин на ориентированном графе. Он позволяет найти кратчайший путь между любой парой вершин графа и строит матрицу, в которой содержатся длины этих кратчайших путей.

**3. Какая вершина графа называется его центром?**

Центром ориентированного графа является вершина с минимальным эксцентриситетом. Эксцентриситет вершины определяется как максимальное расстояние (длина пути) от этой вершины до других вершин графа. Центральная вершина имеет наименьший эксцентриситет и расстояния от неё до других вершин являются минимальными по сравнению с другими вершинами графа.

**Вывод:** в результате выполнения лабораторной работы я научился обрабатывать элементы ориентированных графов и находить в них заданные маршруты