|  |  |  |
| --- | --- | --- |
| Министерство науки и высшего образования  Российской Федерации | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования | | |
| «Новосибирский государственный технический университет» | | |
|  | | |
|  | | |
|  | | |
| Практическое задание №12 | | |
| по дисциплине «Методы построения и анализа алгоритмов» | | |
|  | | |
| **кафедра теоретической и прикладной информатики** | | |
|  | | |
|  | Факультет: | ПМИ |
| Группа: | ПМИ-03 |
| Бригада: | Место для ввода текста. |
| Студенты: | Сидоров Даниил, |
|  | Малыгин Сергей |
|  |  |
| Преподаватель: | Щукин Георгий Анатольевич |
|  |  |
|  | | |
| Новосибирск | | |
| 2021 | | |

**1.Результаты замеров:**

|  |  |  |
| --- | --- | --- |
| Вершины | Жадный алгоритм | Полный перебор  O(nn\*n) |
| 10 | 2.45e-05 | 0.154859 |
| 11 | 2.05e-05 | 1.83657 |
| 12 | 5.58e-05 | 20.6258 |
| 100 | 0.0007679 | - |
| 1 000 | 0.153868 | - |
| 2 000 | 0.630635 | - |
| 3 000 | 1.61405 | - |
| 4 000 | 2.97576 | - |
| 5 000 | 4.82397 | - |

|  |  |  |
| --- | --- | --- |
| Вершины | Длина пути  жадного алгоритма | Длина пути  полного перебора |
| 7 | 191 | 155 |
| 8 | 274 | 104 |
| 9 | 256 | 189 |
| 10 | 376 | 149 |

**2.Программа:**

**Tsp.cpp:**

#include "tsp.h"

#include <stdexcept>

#include <cmath>

#include <algorithm>

#include <numeric>

#include <random>

using namespace std;

void Swap(vector<int> &temp, int a, int b)

{

int t = temp[a];

temp[a] = temp[b];

temp[b] = t;

}

void reshuffle\_create(int k, vector<int> &a, vector<vector<int>> &reshuffle)

{

if (k == a.size())

{

reshuffle.push\_back(a);

}

else

{

for (int j = k; j < a.size(); j++)

{

Swap(a, k, j);

reshuffle\_create(k + 1, a, reshuffle);

Swap(a, k, j);

}

}

}

vector<int> tsp(const Graph &graph, int begin\_vertex) {

if (graph.get\_vertices().size() <= 1)

return vector<int> {};

if (graph.get\_vertices().size() == 2) {

auto vertex = graph.get\_vertices();

return vector<int> {vertex[0], vertex[1]};

}

double min\_weight = std::numeric\_limits<double>::infinity();

vector<int> result;

vector<int> temp\_result;

int node = begin\_vertex;

bool flag; double weight;

vector<vector<int>> reshuffle;

auto vertex = graph.get\_vertices();

vertex.erase(find(vertex.begin(), vertex.end(), begin\_vertex));

vector<vector<int>> Permutations;

if (vertex.size() > 2)

reshuffle\_create(0, vertex, reshuffle);

else {

reshuffle.push\_back({ vertex[0], vertex[1] });

reshuffle.push\_back({ vertex[1], vertex[0] });

}

for (auto &i : reshuffle) {

flag = 1;

node = begin\_vertex;

temp\_result = {};

weight = 0;

for (auto &j : i) {

if (graph.has\_edge(node, j)) {

temp\_result.push\_back(j);

weight += graph.edge\_weight(node, j);

node = j;

}

else { flag = 0; break; };

}

if (flag and weight < min\_weight) {

result = temp\_result;

min\_weight = weight;

}

}

if (result.size() > 0)

result.insert(result.begin(), begin\_vertex);

else return {};

if (!graph.has\_edge(\*(result.begin()), \*(result.end() - 1))) return {};

return result;

}

vector<int> greedy\_tsp(const Graph &graph, int start\_vertex)

{

if (graph.get\_vertices().size() < 2) return {};

vector<int> vertices = graph.get\_vertices();

auto it = find(vertices.begin(), vertices.end(), start\_vertex);

vertices.erase(it);

vector<int> result = { start\_vertex };

vector<int> adjacent\_vertices;

double min;

int vertex = start\_vertex;

while (vertices.size() != 0)

{

min = numeric\_limits<double>::infinity();

adjacent\_vertices = graph.get\_adjacent\_vertices(vertex);

for (const auto &elem : adjacent\_vertices)

{

if (graph.edge\_weight(vertex, elem) < min && find(vertices.begin(), vertices.end(), elem) != vertices.end())

{

min = graph.edge\_weight(vertex, elem);

vertex = elem;

}

}

if (min == numeric\_limits<double>::infinity())

return {};

result.push\_back(vertex);

auto it = find(vertices.begin(), vertices.end(), vertex);

vertices.erase(it);

}

return result;

}

void random\_graph(Graph &graph, int num\_vertices)

{

for (int i = 0; i < num\_vertices; i++)

graph.add\_vertex(i);

default\_random\_engine generator;

uniform\_int\_distribution<int> distributionEdge(1, 100);

for (int i = 0; i < num\_vertices - 1; i++)

for (int j = i + 1; j < num\_vertices; j++)

graph.add\_edge(i, j, distributionEdge(generator));

}

**main.cpp:**

#define CATCH\_CONFIG\_RUNNER

#include "catch.hpp"

#include <iostream>

#include <chrono>

#include "tsp.h"

using namespace std;

int main(int argc, char\* argv[]) {

int result = Catch::Session().run(argc, argv);

Graph graph;

vector<int> res;

for (int i = 1000; i <=5000; i+=1000)

{

random\_graph(graph, i);

cout << "i: " << i << endl;

auto t1 = chrono::high\_resolution\_clock::now();

res = greedy\_tsp(graph, 0);

auto t2 = chrono::high\_resolution\_clock::now();

auto seconds = chrono::duration<double>(t2 - t1).count();

cout << "Time of greedy\_tsp: " << seconds << endl;

double sum = 0;

for (int j = 0; j < i-1; j++)

sum += graph.edge\_weight(res[j], res[j + 1]);

sum += graph.edge\_weight(res[res.size()-1], res[0]);

cout << "sum greedy: " << sum << endl;

t1 = chrono::high\_resolution\_clock::now();

res = tsp(graph, 0);

t2 = chrono::high\_resolution\_clock::now();

seconds = chrono::duration<double>(t2 - t1).count();

cout << "Time of tsp: " << seconds << endl;

sum = 0;

for (int j = 0; j < i - 1; j++)

sum += graph.edge\_weight(res[j], res[j + 1]);

sum += graph.edge\_weight(res[res.size()-1], res[0]);

cout << "sum naive: " << sum << endl;

}

return result;

}