|  |  |  |
| --- | --- | --- |
| Министерство науки и высшего образования  Российской Федерации | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования | | |
| «Новосибирский государственный технический университет» | | |
|  | | |
|  | | |
|  | | |
| Практическое задание №14 | | |
| 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 | Метод локального поиска |
| 7 | 8.2e-06 | 0.0002587 | 1.64e-05 |
| 8 | 1.13e-05 | 0.0022664 | 1.56e-05 |
| 9 | 1.16e-05 | 0.0146126 | 3.16e-05 |
| 10 | 2.45e-05 | 0.154859 | 2.9e-05 |
| 11 | 2.45e-05 | 1.45261 | 3.96e-05 |
| 100 | 0.0007679 | - | 0.334892 |
| 150 | 0.0020238 | - | 1.82167 |
| 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 | Длина пути метода локального поиска |
| 7 | 191 | 69 | 122 |
| 8 | 274 | 91 | 157 |
| 9 | 256 | 103 | 211 |
| 10 | 376 | 119 | 286 |
| 11 | 385 | 113 | 319 |

**2.Программа:**

**Tsp.cpp:**

#include "tsp.h"

#include <cmath>

#include <algorithm>

#include <numeric>

#include <random>

#include <stdexcept>

using namespace std;

double Length(const Graph &graph, vector<int> Path)

{

double lengthPath = 0;

for (int i = 0; i < Path.size() - 1; i++)

lengthPath += graph.edge\_weight(Path[i], Path[i + 1]);

return lengthPath;

}

vector<int> Transform(vector<int> CurrentPath, int A, int B, int C, int D)

{

vector<int> result= CurrentPath;

vector<int>::iterator itB = find(result.begin(), result.end(), B);

vector<int>::iterator itC = find(result.begin(), result.end(), C);

iter\_swap(itB, itC);

return result;

}

vector<int> TwoOptImprove(const Graph &graph, vector<int> CurrentPath)

{

vector<pair<int, int>> nonAdjacentPairs;

vector<pair<pair<int, int>, pair<int, int>>> result;

for (int i = 0; i < CurrentPath.size() - 1; i++)

for (int j = i + 1; j < CurrentPath.size(); j++)

nonAdjacentPairs.push\_back(make\_pair(CurrentPath[i], CurrentPath[j]));

for (int i = 0; i < nonAdjacentPairs.size() - 1; i++)

for (int j = i + 1; j < nonAdjacentPairs.size(); j++)

if (nonAdjacentPairs[i].first != nonAdjacentPairs[j].first && nonAdjacentPairs[i].first != nonAdjacentPairs[j].second && nonAdjacentPairs[i].second != nonAdjacentPairs[j].first && nonAdjacentPairs[i].second != nonAdjacentPairs[j].second)

result.push\_back(make\_pair(nonAdjacentPairs[i], nonAdjacentPairs[j]));

double OldWeight;

double NewWeight;

for (int i = 0; i < result.size(); i++)

{

OldWeight = graph.edge\_weight(result[i].first.first, result[i].first.second) + graph.edge\_weight(result[i].second.first, result[i].second.second);

NewWeight = graph.edge\_weight(result[i].first.first, result[i].second.first) + graph.edge\_weight(result[i].first.second, result[i].second.second);

if (NewWeight < OldWeight)

return Transform(CurrentPath, result[i].first.first, result[i].first.second, result[i].second.first, result[i].second.second);

}

return CurrentPath;

}

vector<int> tsp(const Graph &graph)

{

if (graph.get\_vertices().size() < 2)

return {};

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

srand(unsigned(std::time(0)));

random\_shuffle(CurrentPath.begin(), CurrentPath.end());

vector<int> ImprovedPath;

while (true)

{

try

{

ImprovedPath = TwoOptImprove(graph, CurrentPath);

}

catch (invalid\_argument a)

{

return {};

}

if (Length(graph, ImprovedPath) < Length(graph, CurrentPath))

CurrentPath = ImprovedPath;

else

return CurrentPath;

}

}

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 <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 = 7; i <= 10; i++)

{

random\_graph(graph, i);

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

res = tsp(graph);

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

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

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

double sum = 0;

for (int j = 0; j < res.size() - 1; j++)

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

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

cout << "sum tsp: " << sum << endl;

}

return result;

}