|  |  |  |
| --- | --- | --- |
| Министерство науки и высшего образования  Российской Федерации | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования | | |
| «Новосибирский государственный технический университет» | | |
|  | | |
|  | | |
|  | | |
| Практическое задание №13 | | |
| 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 ветвей и границ |
| 7 | 8.2e-06 | 0.0002587 | 6.45e-05 |
| 8 | 1.13e-05 | 0.0022664 | 0.000848566 |
| 9 | 1.16e-05 | 0.0146126 | 0.00845085 |
| 10 | 2.45e-05 | 0.154859 | 0.037397 |
| 11 | 2.45e-05 | 1.45261 | 0.318263 |
| 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 | Длина пути метода ветвей и границ |
| 7 | 191 | 155 | 155 |
| 8 | 274 | 104 | 104 |
| 9 | 256 | 189 | 189 |
| 10 | 376 | 149 | 149 |

**2.Программа:**

**Tsp.cpp:**

#include <cmath>

#include <algorithm>

#include <numeric>

#include <random>

#include <stdexcept>

using namespace std;

vector<int> MinPath(const Graph &graph, vector<int> P1, vector<int> P2)

{

double lengthP1 = 0;

double lengthP2 = 0;

for (int i = 0; i < P1.size() - 1; i++)

{

lengthP1 += graph.edge\_weight(P1[i], P1[i + 1]);

}

for (int i = 0; i < P2.size() - 1; i++)

{

lengthP2 += graph.edge\_weight(P2[i], P2[i + 1]);

}

if (lengthP1 < lengthP2) return P1; else return P2;

}

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;

}

double LowerBound(const Graph &graph, vector<int> Visited)

{

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

vector<int> adjacent\_vertices;

double minSum = 0;

double minVertex1;

double minVertex2;

for (const auto &elem : Visited)

{

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

minVertex1 = numeric\_limits<double>::infinity();

minVertex2 = numeric\_limits<double>::infinity();

for (const auto &elem2 : adjacent\_vertices)

{

if (graph.edge\_weight(elem, elem2) < minVertex2)

{

if (graph.edge\_weight(elem, elem2) < minVertex1)

{

minVertex2 = minVertex1;

minVertex1 = graph.edge\_weight(elem, elem2);

}

else

{

minVertex2 = graph.edge\_weight(elem, elem2);

}

}

}

minSum += minVertex1 + minVertex2;

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

vertices.erase(it);

}

for (const auto &elem : vertices)

{

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

minVertex1 = numeric\_limits<double>::infinity();

minVertex2 = numeric\_limits<double>::infinity();

for (const auto &elem2 : adjacent\_vertices)

{

if (graph.edge\_weight(elem, elem2) < minVertex2)

{

if (graph.edge\_weight(elem, elem2) < minVertex1)

{

minVertex2 = minVertex1;

minVertex1 = graph.edge\_weight(elem, elem2);

}

else

{

minVertex2 = graph.edge\_weight(elem, elem2);

}

}

}

minSum += minVertex1 + minVertex2;

}

minSum = minSum \* 0.5;

return minSum;

}

vector<int> BnB(const Graph &graph, vector<int> Visited, vector<int> BestPath)

{

try

{

if (Visited.size() == graph.get\_vertices().size()) return MinPath(graph, BestPath, Visited);

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

vector<int> VNext;

vector<int> Path;

for (const auto &elem : Visited)

{

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

vertices.erase(it);

}

for (const auto &elem : vertices)

{

VNext = Visited;

VNext.push\_back(elem);

if (LowerBound(graph, VNext) < Length(graph, BestPath))

{

Path = BnB(graph, VNext, BestPath);

BestPath = MinPath(graph, BestPath, Path);

}

}

return BestPath;

}

catch (invalid\_argument a)

{

return {};

}

}

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

{

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

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

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

BestPath.erase(it);

BestPath.insert(BestPath.begin(), start\_vertex);

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

vector<int> result = BnB(graph, Visited, BestPath);

if (!result.size()<1)

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

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 "tsp.h"

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

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

vector<int> res;

Graph graph;

for (int i = 6; i <=6; i++)

{

random\_graph(graph,i);

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

res = tsp(graph, 0);

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 = 1; j < res.size(); j++)

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

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

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

res.clear();

}

return result;

}