|  |  |  |
| --- | --- | --- |
| Министерство науки и высшего образования  Российской Федерации | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования | | |
| «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра прикладной математики | | |
|  | | |
| Домашнее задание № 3 | | |
| по дисциплине «Методы построения и анализа алгоритмов» | | |
|  | | |
| **NP-полные задачи** | | |
|  | | |
| Группа: | ПМИ-81 |
|  |  |
| Студенты: | Демидович Егор |
|  | Вариант - 3 |
|  |  |
| Преподаватели: | Щукин Георгий Анатольевич |
|  |  |
|  | | |
| Новосибирск | | |
| 2019 | | |

1. **Условие задачи**

Для решения NP-полной задачи из варианта требуется разработать (реализовать) два алгоритма:

1. алгоритм, решающий задачу методом полного перебора/комбинаторного поиска и гарантированно находящий оптимальное решение задачи. С большой вероятностью этот алгоритм будет иметь экспоненциальную временную сложность
2. алгоритм, находящий субоптимальное (приближенное) решение задачи, но имеющий полиномиальную временную сложностью. Классы алгоритмов, находящих субоптимальное решение: жадные алгоритмы, рандомизированные алгоритмы, эвристические алгоритмы и т.д.

Дана электрическая сеть, представляющая из себя множество электрогенераторов, между которыми протянуты провода. Провод имеет ток, если работает хотя бы один генератор на одном из концов провода. Найти минимальное по размеру множество генераторов, которые потребуется включить, чтобы обеспечить током всю сеть.

Задача представляет из себя NP-полную задачу - задачу о минимальном вершинном покрытии (minimum vertex cover)

1. **Использованные алгоритмы**

Оптимальный алгоритм основан на методе полного перебора.   
Его сложность: **O(n!),**

В качестве субоптимального алгоритма - жадный алгоритм.  
Его сложность: **O(n\*log(n))**

1. **Текст программы**

**Source.cpp**

#define CATCH\_CONFIG\_RUNNER

#include "Library.h"

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

{

Catch::Session().run(argc, argv);

int a, b;

int swap;

vector<Wire> wires = read\_from\_file("input.txt");

vector<int> greedy = getcover\_greedy(wires);

vector<int> bruted = getcover\_brute(wires);

result\_to\_files(greedy, bruted);

printf("==========GREEDY==========\n");

printf("Nodes\tEdges\tTime\n");

for (int i = 0; i <= 3; i++)

for (int j = 1; j < 10; j++)

measure\_greedy(pow(10, i) \* j, pow(10, i) \* j);

printf("\n==========BRUTE===========\n");

printf("Nodes\tEdges\tTime\n");

for (int i = 0; i <= 30; i++)

measure\_brute(i, i);

printf("\n ACCURACY TEST \n");

printf("Nodes\tEdges\tBrute\tGreedy\n");

for (int i = 1; i <= 30; i++)

measure\_accuracy(i, i\*4);

\_getch();

return 0;

}

**electricity.h**

#pragma once

#include <vector>

#include <algorithm>

#include <iostream>

#include "timer.h"

using namespace std;

class Wire {

public:

int a;

int b;

Wire(int A, int B);

};

// If int node in vector

bool isContains(vector<int> set, int node);

// If wire node in vector

bool isContains(vector<Wire> set, Wire node);

// If set covers generators with wires

bool isCovers(vector<Wire> wires, vector<int> generators, vector<int> set);

// Returns count of adjacent nodes in node

int vertexAdjCount(vector<Wire> wires, int node);

// Creates new vector of vectors that contains new unique elements from vertexes, little first

vector<vector<int>> appendix(vector<vector<int>> previous, vector<int> vertexes);

// Returns unique vertexes in wire vector

vector<int> getVertexes(vector<Wire> wires);

// Returns minimum cover set using greedy algorithm

vector<int> getcover\_greedy(vector<Wire> wires);

// Returns minimum cover set using brute algorithm

vector<int> getcover\_brute(vector<Wire> wires);

**Library.h**

#pragma once

#define REPEATS 5

#include "catch.hpp"

#include "timer.h"

#include "electricity.h"

#include <fstream>

#include <conio.h>

#include <map>

#include <set>

// Generates random wire vector with n nodes and m wires

vector<Wire> gen\_random\_wires(int n, int m);

// Measuring brute algorithm test with given counts of nodes and arcs

void measure\_brute(int nodes, int arcs);

// Measuring greedy algorithm test with given counts of nodes and arcs

void measure\_greedy(int nodes, int arcs);

// Measuring accuracy of algorithms with given counts of nodes and arcs

void measure\_accuracy(int nodes, int arcs);

// Reads given file and returns vector of wires

vector<Wire> read\_from\_file(const char\* chars);

// Writes out results of both algorithms into two files

void result\_to\_files(vector<int> greedy, vector<int> brute);

**timer.h**

#pragma once

#include <chrono>

typedef std::chrono::high\_resolution\_clock hrc;

namespace {

class Timer {

public:

Timer() {}

void start() {

begin = hrc::now();

}

double getTime() {

return std::chrono::duration<double>(hrc::now() - begin).count();

}

private:

hrc::time\_point begin;

};

}

**electricity.cpp**

#pragma once

#include "electricity.h"

Wire::Wire(int A, int B) {

a = A;

b = B;

};

bool isContains(vector<int> set, int node) {

for (int w : set)

if (w == node)

return true;

return false;

}

bool isContains(vector<Wire> set, Wire node) {

for (Wire w : set)

if (w.a == node.a && w.b == node.b)

return true;

return false;

}

bool isCovers(vector<Wire> wires, vector<int> generators, vector<int> set) {

vector<int> lighted;

for (int p : set)

for (Wire wire : wires)

if (wire.a == p || wire.b == p)

{

if (!isContains(lighted, wire.a))

lighted.push\_back(wire.a);

if (!isContains(lighted, wire.b))

lighted.push\_back(wire.b);

}

sort(lighted.begin(), lighted.end());

sort(generators.begin(), generators.end());

return lighted == generators;

}

int vertexAdjCount(vector<Wire> wires, int node) {

int ans = 0;

for (Wire wire:wires)

if (wire.a == node || wire.b == node)

ans++;

return ans;

}

vector<vector<int>> appendix(vector<vector<int>> previous, vector<int> vertexes) {

sort(vertexes.begin(), vertexes.end());

int hs = previous.size();

int vs = vertexes.size();

vector<vector<int>> newious;

vector<int> buffer;

for (int i = 0; i < hs; i++)

{

for (int j = 0; j < vs; j++)

{

buffer = previous[i];

while (buffer.size() != 0 && buffer.back() > vertexes[j])

j++;

if (buffer.size() == 0 || buffer.back() < vertexes[j])

{

buffer.push\_back(vertexes[j]);

newious.push\_back(buffer);

}

}

}

return newious;

}

vector<int> getVertexes(vector<Wire> wires) {

vector<int> vertexes;

for (Wire wire : wires) {

if (!isContains(vertexes, wire.a))

vertexes.push\_back(wire.a);

if (!isContains(vertexes, wire.b))

vertexes.push\_back(wire.b);

}

return vertexes;

}

vector<int> getcover\_greedy(vector<Wire> wires) {

vector<int> ans;

vector<int> vertexes = getVertexes(wires);

vector<int> workset = vertexes;

vector<int> weight;

for (int p : workset)

weight.push\_back(vertexAdjCount(wires, p));

int vs = workset.size();

int imax;

while (vs>0)

{

imax = 0;

for (int i = 1; i < vs; i++)

if (weight[i] > weight[imax])

imax = i;

ans.push\_back(workset[imax]);

weight.erase(weight.begin() + imax);

workset.erase(workset.begin() + imax);

vs = workset.size();

if (isCovers(wires, vertexes, ans))

break;

}

return ans;

}

vector<int> getcover\_brute(vector<Wire> wires) {

vector<int> ans;

vector<int> vertexes = getVertexes(wires);

int vs = vertexes.size();

int ws = wires.size();

vector<vector<int>> combinations = { {} };

for (int i = 1; i <= vs; i++)

{

combinations = appendix(combinations, vertexes);

for (vector<int> p : combinations) {

if (isCovers(wires, vertexes, p))

{

ans = p;

i = vs;

break;

}

}

}

return ans;

}

**Library.cpp**

#pragma once

#include "Library.h"

vector<Wire> gen\_random\_wires(int nodes, int arcs) {

vector<Wire> ans;

int a, b, swap;

for (int i = 0; i < arcs; i++) {

a = rand() % nodes;

b = rand() % nodes;

if (a < b) {

swap = a;

a = b;

b = swap;

}

ans.push\_back(Wire(a, b));

}

return ans;

}

void measure\_brute(int nodes, int arcs) {

Timer t;

vector<Wire> wires;

double time = 0;

for (int i = 0; i < REPEATS; i++) {

wires = gen\_random\_wires(nodes, arcs);

t.start();

getcover\_brute(wires);

time += t.getTime();

}

printf("%d\t%d\t%10f\n", nodes, arcs, time / REPEATS);

}

void measure\_greedy(int nodes, int arcs) {

Timer t;

vector<Wire> wires;

double time = 0;

for (int i = 0; i < REPEATS; i++) {

wires = gen\_random\_wires(nodes, arcs);

t.start();

getcover\_greedy(wires);

time += t.getTime();

}

printf("%d\t%d\t%10f\n", nodes, arcs, time / REPEATS);

}

void measure\_accuracy(int nodes, int arcs) {

vector<Wire> wires = gen\_random\_wires(nodes, arcs);

vector<int> brute = getcover\_brute(wires);

vector<int> greedy = getcover\_greedy(wires);

printf("%d\t%d\t%d\t%d\n", nodes, arcs, brute.size(), greedy.size());

}

vector<Wire> read\_from\_file(const char\* chars) {

vector<Wire> wires;

int a, b, swap;

ifstream in(chars);

if (!in)

return {};

while (in && !in.eof()) {

in >> a >> b;

if (a > b)

{

swap = a;

a = b;

b = swap;

}

Wire buffer(a, b);

if (!isContains(wires, buffer))

wires.push\_back(buffer);

in.close();

return wires;

}

}

void result\_to\_files(vector<int> greedy, vector<int> brute) {

ofstream fout;

fout.open("greedy.txt");

for (int p : greedy)

fout << p << " ";

fout.close();

fout.open("brute.txt");

for (int p : brute)

fout << p << " ";

fout.close();

}

**4.Текст юнит тестов**

#pragma once

#include "Library.h"

template <typename Container>

bool is\_equal(const Container& c1, const Container& c2) {

const auto s1 = set<typename Container::value\_type>(c1.begin(), c1.end());

const auto s2 = set<typename Container::value\_type>(c2.begin(), c2.end());

return s1 == s2;

}

TEST\_CASE("[Electricity] Test empty set", "[isContains<int>]") {

vector<int> set;

CHECK\_FALSE(isContains(set,0));

}

TEST\_CASE("[Electricity] Test filled set", "[isContains<int>]") {

vector<int> set{ 1,2,3,5,6,7 };

CHECK(isContains(set, 3));

CHECK\_FALSE(isContains(set, 4));

}

TEST\_CASE("[Electricity] Test filled wireset", "[isContains<Wire>]") {

vector<Wire> set{ {1,2},{2,3},{4,6} };

CHECK(isContains(set, { 1,2 }));

CHECK\_FALSE(isContains(set, {1,3}));

}

TEST\_CASE("[Electricity] Test if set isCovers", "[isCovers]") {

vector<Wire> wires{ {1,2},{2,3},{4,6} };

vector<int> generators{ 1,2,3,4,6 };

CHECK(isCovers(wires, generators, { 2,4 }));

CHECK\_FALSE(isCovers(wires, generators, { 4 }));

}

TEST\_CASE("[Electricity] Test adjacent vertex count", "[vertexAdjCount]") {

vector<Wire> wires{ {1,2},{2,3},{4,6} };

vector<int> generators{ 1,2,3,4,6 };

CHECK(vertexAdjCount(wires, 4) == 1);

CHECK(vertexAdjCount(wires, 2) == 2);

}

TEST\_CASE("[Electricity] Test appendix", "[appendix]") {

vector<vector<int>> prev{ {1},{2},{3} };

vector<int> nodes{ 1,2,3 };

vector<vector<int>> wanted{ {1,2},{1,3},{2,3} };

CHECK(is\_equal(appendix(prev, nodes), wanted));

}

TEST\_CASE("[Electricity] Test get vertexes", "[getVertexes]") {

vector<Wire> wires{ {1,2},{2,3},{4,6} };

CHECK(is\_equal(getVertexes(wires), { 1,2,3,4,6 }));

CHECK\_FALSE(is\_equal(getVertexes(wires), { 1,2,3,4 }));

}

TEST\_CASE("[Electricity] Test one station no wires (greedy)", "[getcover\_greedy]") {

vector<Wire> wires{ {1,1} };

CHECK(is\_equal(getcover\_greedy(wires), {1}));

}

TEST\_CASE("[Electricity] Test one station no wires (brute)", "[getcover\_brute]") {

vector<Wire> wires{ {1,1} };

CHECK(is\_equal(getcover\_brute(wires), { 1 }));

}

TEST\_CASE("[Electricity] Test three negative stations no wires (greedy)", "[getcover\_greedy]") {

vector<Wire> wires{ {-1,-1},{1,1},{-7,-7} };

CHECK(is\_equal(getcover\_brute(wires), { -1,1,-7 }));

}

TEST\_CASE("[Electricity] Test three negative stations no wires (brute)", "[getcover\_brute]") {

vector<Wire> wires{ {-1,-1},{1,1},{-7,-7} };

CHECK(is\_equal(getcover\_brute(wires), { -1,1,-7 }));

}

TEST\_CASE("[Electricity] Test three stations one wire (greedy) fail", "[getcover\_greedy]") {

vector<Wire> wires{ {-1,1},{-7,-7} };

vector<int> result = getcover\_greedy(wires);

bool br = is\_equal(result, { -1,-7 }) || is\_equal(result, { 1,-7 });

CHECK\_FALSE(br);

}

TEST\_CASE("[Electricity] Test three stations one wire (brute)", "[getcover\_brute]") {

vector<Wire> wires{ {-1,1},{-7,-7} };

vector<int> result = getcover\_brute(wires);

bool br = is\_equal(result, { -1,-7 }) || is\_equal(result, { 1,-7 });

CHECK(br);

}

TEST\_CASE("[Electricity] Test three stations two wires (greedy)", "[getcover\_greedy]") {

vector<Wire> wires{ {-1,1},{1,-7} };

vector<int> result = getcover\_greedy(wires);

bool br = is\_equal(result, { 1 });

CHECK(br);

}

TEST\_CASE("[Electricity] Test three stations two wires (brute)", "[getcover\_brute]") {

vector<Wire> wires{ {-1,1},{1,-7} };

vector<int> result = getcover\_brute(wires);

bool br = is\_equal(result, { 1 });

CHECK(br);

}

TEST\_CASE("[Electricity] Test three stations three wires, triangle (greedy)", "[getcover\_greedy]") {

vector<Wire> wires{ {-1,1},{1,-7},{-7,-1} };

vector<int> result = getcover\_greedy(wires);

bool br = is\_equal(result, { -1 }) || is\_equal(result, { 1 }) || is\_equal(result, { -7 });

CHECK(br);

}

TEST\_CASE("[Electricity] Test three stations three wires, triangle (brute)", "[getcover\_brute]") {

vector<Wire> wires{ {-1,1},{1,-7},{-7,-1} };

vector<int> result = getcover\_brute(wires);

bool br = is\_equal(result, { -1 }) || is\_equal(result, { 1 }) || is\_equal(result, { -7 });

CHECK(br);

}

TEST\_CASE("[Electricity] Test five stations four wires, flat (greedy) fail", "[getcover\_greedy]") {

vector<Wire> wires{ {1,2},{2,3},{3,4},{4,5} };

vector<int> result = getcover\_greedy(wires);

bool br = is\_equal(result, { 1,4 }) || is\_equal(result, { 2,4 }) || is\_equal(result, { 2,5 });

CHECK\_FALSE(br);

}

TEST\_CASE("[Electricity] Test five stations four wires, flat (brute)", "[getcover\_brute]") {

vector<Wire> wires{ {1,2},{2,3},{3,4},{4,5} };

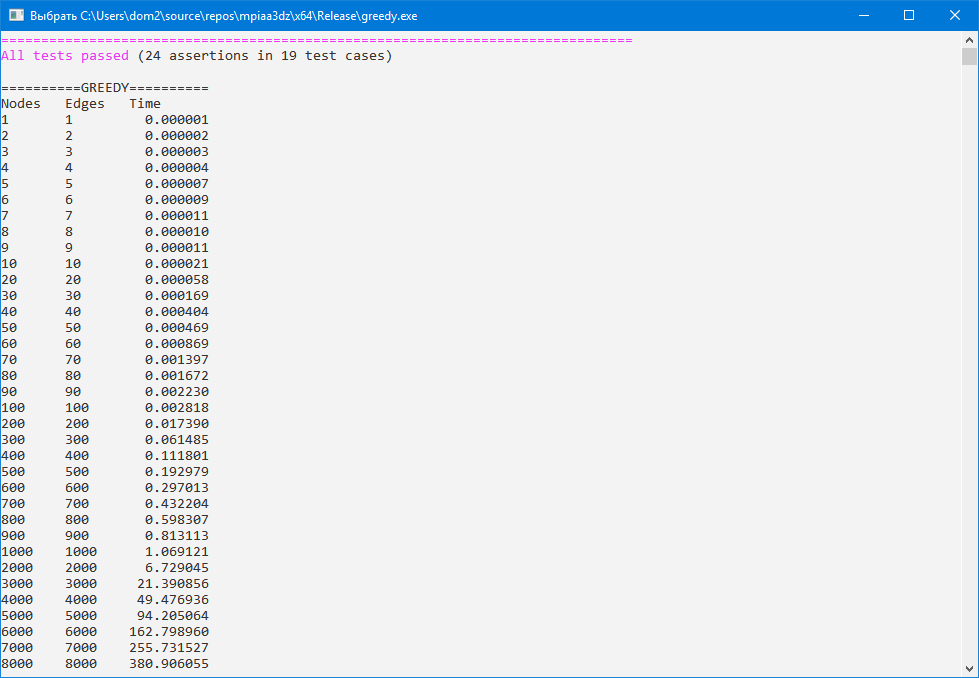
vector<int> result = getcover\_brute(wires);

bool br = is\_equal(result, { 1,4 }) || is\_equal(result, { 2,4 }) || is\_equal(result, { 2,5 });

CHECK(br);

}

**5. Тесты**



**6.Графики зависимости времени от количества проводов**

**1) Алгоритм полного перебора**

**2) Жадный алгоритм**

**Сравнение точности алгоритмов:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Соединений | n | | 2n | | 4n | |
| Вершин\Алгоритм | **Полный** | **Жадный** | **Полный** | **Жадный** | **Полный** | **Жадный** |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| 3 | 2 | 2 | 1 | 1 | 1 | 1 |
| 4 | 1 | 1 | 2 | 3 | 1 | 1 |
| 5 | 2 | 2 | 1 | 1 | 1 | 1 |
| 6 | 2 | 3 | 2 | 2 | 2 | 2 |
| 7 | 3 | 3 | 2 | 3 | 2 | 2 |
| 8 | 2 | 5 | 3 | 8 | 2 | 2 |
| 9 | 3 | 6 | 3 | 8 | 2 | 2 |
| 10 | 4 | 5 | 3 | 5 | 2 | 4 |
| 11 | 4 | 8 | 3 | 8 | 2 | 6 |
| 12 | 4 | 7 | 3 | 3 | 2 | 3 |
| 13 | 4 | 6 | 3 | 6 | 2 | 9 |
| 14 | 5 | 10 | 4 | 8 | 2 | 3 |
| 15 | 5 | 6 | 4 | 6 | 3 | 4 |
| 16 | 5 | 10 | 5 | 12 | 3 | 5 |
| 17 | 5 | 14 | 4 | 6 | 3 | 5 |
| 18 | 6 | 11 | 4 | 7 | 3 | 8 |
| 19 | 6 | 11 | 5 | 10 | 3 | 9 |
| 20 | 6 | 16 | 4 | 9 | 3 | 5 |
| 21 | 6 | 12 | 6 | 13 | 4 | 6 |
| 22 | 6 | 16 | 5 | 9 | 4 | 21 |
| 23 | 7 | 17 | 6 | 11 | 4 | 16 |
| 24 | 7 | 17 | 6 | 12 | 5 | 13 |
| 25 | 8 | 18 | 6 | 13 | 4 | 7 |
| 26 | 7 | 14 | 6 | 18 | 5 | 6 |
| 27 | 8 | 14 | 8 | 15 | 4 | 7 |
| 28 | 9 | 22 | 7 | 24 | 5 | 9 |
| 29 | 8 | 20 | 7 | 12 | 5 | 6 |
| 30 | 8 | 19 | 6 | 25 | 5 | 6 |
|  | Количество вершин | | | | | |

**7.Вывод**

В ходе выполнения работы была выполнена:

* 1. Реализация алгоритма полного перебора для данной задачи.
  2. Реализация жадного алгоритма для данной задачи.
  3. Проведено тестирование на проверку корректности реализации алгоритмов и программы.
  4. Проведено исследование времени работы алгоритма на данных разного размера.

Результаты:

1. Тестирование работает корректно.
2. Алгоритм полного перебора имеет сложность: **O(n!)**
3. Жадный алгоритм имеет сложность: **O(n\*log(n))**
4. Полный перебор даёт точный результат, в то время как жадный алгоритм даёт приближенный результат.
5. Проведено исследование точности работы жадного алгоритма относительно полного перебора