**ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ**

**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

**«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»**

Факультет компьютерных наук

Департамент программной инженерии

|  |  |  |
| --- | --- | --- |
| СОГЛАСОВАНО  Научный руководитель,  Старший преподаватель департамента  программной инженерии факультет компьютерных наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ С.А. Шершаков  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г. |  | УТВЕРЖДАЮ  Академический руководитель образовательной программы «Программная инженерия»  профессор департамента программной инженерии, канд. техн. наук  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В.В. Шилов  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2019г. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | ***Подп. и дата*** |  | | ***Инв. № дубл.*** |  | | ***Взам. инв. №*** |  | | ***Подп. и дата*** |  | | ***Инв. № подл*** |  | | **ПРОГРАММА ПОСТРОЕНИЯ МОДЕЛИ ПРОЦЕССА АЛГОРИТМОМ FUZZY MINER**  **Текст программы**  **ЛИСТ УТВЕРЖДЕНИЯ**  **RU.17701729.04.01-01 12 01-1-ЛУ** | | |
|  |  | |
| Исполнитель  Студент группы БПИ 182  \_\_\_\_\_\_\_\_\_\_\_/Егоров И.С./  «\_\_\_»\_\_\_\_\_\_\_\_\_\_2019 г. | |
|  | | |
|  | |  |

**Москва 2019**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| УТВЕРЖДЕН  RU.17701729.04.01-01 12 01-1-ЛУ |  | |  | |
| |  |  | | --- | --- | | ***Подп. и дата*** |  | | ***Инв. № дубл.*** |  | | ***Взам. инв. №*** |  | | ***Подп. и дата*** |  | | ***Инв. № подл*** |  | | **ПРОГРАММА ПОСТРОЕНИЯ МОДЕЛИ ПРОЦЕССА АЛГОРИТМОМFUZZY MINER**  **Текст программы**  **RU.17701729.04.01-01 12 01-1**  **Листов 60** | | | | |
|  | |  | | |
|  | | |
|  | | | | |
|  | | | |  |

**Москва 2019**

**Содержание**

[**Код Form1.cs** 4](#_Toc21443660)

[**Код node.h** 13](#_Toc21443661)

[**Код node.cpp** 18](#_Toc21443662)

[**Код name\_similarity.h** 30](#_Toc21443663)

[**Код name\_similarity.cpp** 31](#_Toc21443664)

[**Код graph.h** 33](#_Toc21443665)

[**Код graph.cpp** 34](#_Toc21443666)

[**Код dotfile.h** 48](#_Toc21443667)

[**Код dotfile.cpp** 49](#_Toc21443668)

[**Код unary\_metric.h** 50](#_Toc21443669)

[**Код unary\_metric.cpp** 51](#_Toc21443670)

[**Код binary\_metric.h** 55](#_Toc21443671)

[**Код binary\_metric.cpp** 57](#_Toc21443672)

# **Код Form1.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Runtime.InteropServices;

using System.IO;

using System.Diagnostics;

using System.Drawing.Imaging;

namespace FuzzyMinerParser

{

public partial class Form1 : Form

{

const string allowedCharacters = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890";

const string imageName = "image.gif";

string filename\_to\_cpp;

bool draw1 = false;

bool draw2 = false;

string[] linesRead;

public Form1()

{

InitializeComponent();

}

[DllImport("FuzzyMinerLib.dll", EntryPoint = "Foo", CallingConvention = CallingConvention.StdCall)]

unsafe static extern void Execute(char\* file, int length, double\* metrics);

unsafe private void Calculate(object sender, EventArgs e)

{

//this.Enabled = false;

if(pictureBox1.Image != null)

{

pictureBox1.Image.Dispose();

pictureBox1.Image = null;

}

this.Cursor = Cursors.WaitCursor;

if (!buttonCalculate.Enabled)

return;

double[] metric = new double[7];

metric[0] = (double)param1.Value;

metric[1] = (double)param2.Value;

metric[2] = (double)param3.Value;

metric[3] = (double)param4.Value;

metric[4] = (double)param5.Value;

metric[5] = (double)autoParam1.Value / 1000;

metric[6] = (double)autoParam2.Value / 1000;

int sz = filename\_to\_cpp.Length;

Console.WriteLine(filename\_to\_cpp);

Console.WriteLine(sz);

Console.WriteLine(metric[6]);

fixed (char\* file\_to\_cpp = filename\_to\_cpp)

{

fixed (double\* metrics = metric)

{

Execute(file\_to\_cpp, sz, metrics);

}

}

//try

//{

System.Diagnostics.Process p = new System.Diagnostics.Process();

p.StartInfo.FileName = @"C:\proj\release\bin\dot.exe"; //позже зафиксить

p.StartInfo.Arguments = "-Tgif" + " -o" + Application.StartupPath + "\\" + imageName + ' ' + Application.StartupPath + "\\" + "test.dot";

p.StartInfo.CreateNoWindow = true;

p.StartInfo.UseShellExecute = false;

p.Start();

while (!p.HasExited) { }

//} catch (Exception ex)

//{

// MessageBox.Show("Файл dot.exe не найден!", "Ошибка!", MessageBoxButtons.OK, MessageBoxIcon.Error);

//}

OpenImage();

this.Cursor = null;

//this.Enabled = true;

}

private void OpenFile(object sender, EventArgs e)

{

buttonCalculate.Enabled = false;

OpenFileDialog dlg = new OpenFileDialog();

dlg.Title = "Открыть журнал событий...";

dlg.Filter = "log files(\*.log)|\*.log";

dlg.CheckPathExists = true;

if (dlg.ShowDialog() == DialogResult.OK)

{

string fileName = dlg.FileName;

if (!ParseFile(fileName))

{

MessageBox.Show("Формат файла не соответствует требуемому!", "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

return;

}

filename\_to\_cpp = fileName;

buttonCalculate.Enabled = true;

SaveFileButton.Enabled = true;

autoParam1.Enabled = true;

autoParam2.Enabled = true;

Calculate(sender, e);

}

}

private void SaveFile(object sender, EventArgs e)

{

SaveFileDialog dlg = new SaveFileDialog();

dlg.Title = "Сохранить визуализацию модели как...";

dlg.CheckPathExists = true;

dlg.OverwritePrompt = true;

dlg.Filter = "Image file(\*.gif)|\*.gif";

if (dlg.ShowDialog() == DialogResult.OK)

{

try

{

Bitmap bmpSave = (Bitmap)pictureBox1.Image;

bmpSave.Save(dlg.FileName, ImageFormat.Gif);

}

catch (Exception ex)

{

MessageBox.Show("Невозможно сохранить изображение!", "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

}

private bool ParseFile(string fileName)

{

string[] lines = File.ReadAllLines(fileName);

foreach (string line in lines)

{

Console.WriteLine(line);

if (!ParseLine(line))

return false;

}

linesRead = (string[])lines.Clone();

return true;

}

private bool ParseLine(string line)

{

string[] words = line.Split(';');

if (words.Last().Length != 0)

return false;

for (int i = 0; i < words.Length - 1; ++i)

{

string word = words[i];

if ((word.Length == 0) || !ParseWord(word))

{

Console.WriteLine($"Word: \"{word}\"");

return false;

}

}

return true;

}

private bool ParseWord(string word)

{

foreach (char c in word)

{

if (!allowedCharacters.Contains(c))

{

Console.Write(c);

return false;

}

}

return true;

}

private void OpenImage()

{

try

{

pictureBox1.Image = Image.FromFile(imageName);

}

catch (IOException ex)

{

MessageBox.Show("Файл изображения не найден!", "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Error);

buttonCalculate.Enabled = false;

SaveFileButton.Enabled = false;

}

}

private void autoParam1\_MouseDown(object sender, MouseEventArgs e)

{

draw1 = true;

}

private void autoParam1\_MouseUp(object sender, MouseEventArgs e)

{

if (!draw1)

return;

draw1 = false;

Calculate(sender, e);

}

private void autoParam1\_Scroll(object sender, EventArgs e)

{

EdgeCutoffValueLabel.Text = ((double)autoParam1.Value / 1000).ToString();

if (draw1)

return;

Calculate(sender, e);

}

private void autoParam2\_Scroll(object sender, EventArgs e)

{

NodeCutoffValueLabel.Text = ((double)autoParam2.Value / 1000).ToString();

if (draw2)

return;

Calculate(sender, e);

}

private void autoParam2\_MouseDown(object sender, MouseEventArgs e)

{

draw2 = true;

}

private void autoParam2\_MouseUp(object sender, MouseEventArgs e)

{

if (!draw2)

return;

draw2 = false;

Calculate(sender, e);

}

}

}

# **Код node.h**

#pragma once

#include <vector>

#include <string>

#include "unary\_metric.h"

#include "binary\_metric.h"

#include "graph.h"

class graph;

class node

{

public:

node();

node(int \_frequency, std::string \_name);

int get\_relation(int num);

int get\_relation\_frequency(int num);

std::string get\_name();

int get\_node\_frequency();

void inc\_frequency();

void inc\_in\_relations();

void inc\_out\_relations();

void push\_relation(int num);

void push\_relation\_in(int num);

int get\_number\_of\_relations();

double get\_unary\_significanse();

void count\_unary\_frequency(int maximum);

double get\_unary\_frequency();

void count\_binary\_frequency(int maximum);

double get\_binary\_frequency(int pos);

bool get\_visible();

void set\_visible(bool val);

void count\_name\_similarity(graph \*fuzzy);

void count\_unary\_routing(graph\* fuzzy, int num);

double get\_unary\_routing();

void normalize\_unary\_routing(double maximum);

void normalize\_binary\_distance(double maximum);

void count\_binary\_distance(graph\* fuzzy, int num);

void count\_aggregate();

double get\_max\_aggregate();

void normalize\_aggregate(double maximum);

void count\_unary\_aggregate();

double get\_unary\_aggregate();

void normalize\_unary\_aggregate(double maximum);

double get\_max\_distance();

void normalize\_distance(double max);

void count\_binary\_significance();

double get\_max\_binary\_significance();

void normalize\_binary\_significance(double maximum);

void count\_unary\_significance();

void normalize\_unary\_significance(double maximum);

int get\_number\_of\_in\_relations();

int get\_in\_relation(int num);

double get\_binary\_significance(int num);

void set\_rel\_visible(int num, bool val);

bool get\_relation\_visible(int num);

void push\_user\_values(double binFrequency, double unFrequency,

double routingSig, double distanceSig, double nameSig);

private:

std::vector<std::pair<int, int>> relations;

std::vector<binary\_metric> binary;

std::vector<int> relations\_in;

std::vector<bool> visible\_relations;

unary\_metric unary;

std::string name;

bool visible;

int frequency, amount\_of\_edges\_in, amount\_of\_edges\_out;

};

# **Код node.cpp**

#include "stdafx.h"

#include "stdafx.h"

#include "node.h"

#include <vector>

#include <string>

#include "binary\_metric.h"

node::node() {

node::frequency = 0;

node::name = "";

node::visible = true;

}

node::node(int \_frequency, std::string \_name) {

node::frequency = \_frequency;

node::amount\_of\_edges\_in = 0;

node::amount\_of\_edges\_out = 0;

node::name = \_name;

node::visible = true;

unary;

}

void node::push\_relation(int num) {

for (int i = 0; i < (int)node::relations.size(); i++) {

if (node::relations[i].first == num) {

node::relations[i].second++;

return;

}

}

node::relations.push\_back(std::make\_pair(num, 1));

visible\_relations.push\_back(true);

}

void node::push\_relation\_in(int num)

{

for (int i = 0; i < (int)node::relations\_in.size(); i++) {

if (node::relations\_in[i] == num) {

return;

}

}

node::relations\_in.push\_back(num);

}

int node::get\_relation(int num) {

return node::relations[num].first;

}

int node::get\_number\_of\_relations() {

return (int)node::relations.size();

}

double node::get\_unary\_significanse()

{

return unary.get\_unary\_significance();

}

void node::count\_unary\_frequency(int maximum)

{

node::unary.count\_frequency(node::frequency, maximum);

}

double node::get\_unary\_frequency()

{

return unary.get\_normalized\_frequency();

}

void node::count\_binary\_frequency(int maximum)

{

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

binary[i].count\_binary\_frequency(relations[i].second, maximum);

}

}

double node::get\_binary\_frequency(int pos)

{

return binary[pos].get\_binary\_frequency();

}

bool node::get\_visible()

{

return visible;

}

void node::set\_visible(bool val)

{

visible = val;

}

void node::count\_name\_similarity(graph \*fuzzy)

{

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

binary[i].count\_name\_similarity(name, (\*fuzzy).get\_node(relations[i].first).get\_name());

}

}

void node::count\_unary\_routing(graph\* fuzzy, int num)

{

double outVal = 0;

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

double binSig = binary[i].get\_binary\_frequency();

double binCor = binary[i].get\_name\_similarity();

outVal += binCor \* binSig;

}

double inVal = 0;

for (int i = 0; i < relations\_in.size(); i++) {

node q = (\*fuzzy).get\_node(relations\_in[i]);

double binSig = 0;

double binCor = 0;

for (int j = 0; j < q.get\_number\_of\_relations(); j++) {

if (q.relations[j].first == num) {

binSig = q.binary[j].get\_binary\_frequency();

binCor = q.binary[j].get\_name\_similarity();

}

}

inVal += binCor \* binSig;

}

if (inVal == 0 && outVal == 0) {

unary.count\_unary\_routing(0);

}

else {

double res = ((inVal - outVal) / (inVal + outVal));

if (res < 0.0) {

res = -res;

}

unary.count\_unary\_routing(res);

}

}

double node::get\_unary\_routing()

{

return unary.get\_normalized\_routing();

}

void node::normalize\_unary\_routing(double maximum)

{

unary.normalize\_unary\_routing(maximum);

}

void node::normalize\_binary\_distance(double maximum)

{

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

binary[i].normalize\_binary\_distance(maximum);

}

}

void node::count\_binary\_distance(graph\* fuzzy, int num)

{

double our\_agg = get\_unary\_aggregate();

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

double rel\_agg = (\*fuzzy).get\_node(relations[i].first).get\_unary\_aggregate();

double bin\_agg = binary[i].get\_aggregate();

binary[i].count\_binary\_distance(our\_agg, rel\_agg, bin\_agg);

}

}

void node::count\_aggregate()

{

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

binary[i].count\_aggregate();

}

}

double node::get\_max\_aggregate()

{

double maximum = 0;

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

if (binary[i].get\_aggregate() > maximum) {

maximum = binary[i].get\_aggregate();

}

}

return maximum;

}

void node::normalize\_aggregate(double maximum)

{

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

binary[i].normalize\_aggregate(maximum);

}

}

void node::count\_unary\_aggregate()

{

unary.count\_aggregate();

}

double node::get\_unary\_aggregate()

{

return unary.get\_aggregate();

}

void node::normalize\_unary\_aggregate(double maximum)

{

unary.normalize\_aggregate(maximum);

}

double node::get\_max\_distance()

{

double maximum = 0;

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

if (binary[i].get\_binary\_distance() > maximum) {

maximum = binary[i].get\_binary\_distance();

}

}

return maximum;

}

void node::normalize\_distance(double maximum)

{

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

binary[i].normalize\_binary\_distance(maximum);

}

}

void node::count\_binary\_significance()

{

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

binary[i].count\_binary\_significance();

}

}

double node::get\_max\_binary\_significance()

{

double maximum = 0;

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

if (binary[i].get\_binary\_significance() > maximum) {

maximum = binary[i].get\_binary\_significance();

}

}

return maximum;

}

void node::normalize\_binary\_significance(double maximum)

{

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

binary[i].normalize\_binary\_significance(maximum);

}

}

void node::count\_unary\_significance()

{

unary.count\_unary\_significance();

}

void node::normalize\_unary\_significance(double maximum)

{

unary.normalize\_unary\_significance(maximum);

}

int node::get\_number\_of\_in\_relations()

{

return relations\_in.size();

}

int node::get\_in\_relation(int num)

{

return relations\_in[num];

}

double node::get\_binary\_significance(int num)

{

return binary[num].get\_binary\_significance();

}

void node::set\_rel\_visible(int num, bool val)

{

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

if (relations[i].first == num) {

visible\_relations[i] = val;

return;

}

}

}

bool node::get\_relation\_visible(int num)

{

return visible\_relations[num];

}

void node::push\_user\_values(double binFrequency, double unFrequency, double routingSig, double distanceSig, double nameSig)

{

binary.resize(relations.size());

unary.set\_user\_values(unFrequency, routingSig);

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

binary[i].set\_user\_values(binFrequency, distanceSig, nameSig);

}

}

int node::get\_relation\_frequency(int num) {

return node::relations[num].second;

}

std::string node::get\_name() {

return node::name;

}

int node::get\_node\_frequency() {

return node::frequency;

}

void node::inc\_frequency() {

node::frequency++;

}

void node::inc\_in\_relations()

{

node::amount\_of\_edges\_in++;

}

void node::inc\_out\_relations()

{

node::amount\_of\_edges\_out++;

}

# **Код name\_similarity.h**

#pragma once

#include <string>

#include <vector>

#include <algorithm>

class name\_similarity

{

public:

double get\_name\_similarity(std::string first, std::string second);

};

# **Код name\_similarity.cpp**

#include "stdafx.h"

#include "name\_similarity.h"

double name\_similarity::get\_name\_similarity(std::string first, std::string second)

{

int n = first.length();

int m = second.length();

if (n == 0) {

return m;

}

else if (m == 0) {

return n;

}

int MAX\_N = n + m;

int cost;

std::vector<int> d(MAX\_N + 1), p(MAX\_N + 1);

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

p[i] = i;

}

std::string s, f;

for (int j = 1; j <= m; j++) {

s = second[j - 1];

d[0] = j;

f = "";

for (int i = 1; i <= n; i++) {

f = first[i - 1];

if (s == f)

cost = 0;

else

cost = 1;

if (d[i - 1] + 1 < p[i] + 1) {

d[i] = d[i - 1] + 1;

}

else {

d[i] = p[i] + 1;

}

if (d[i] > p[i - 1] + cost) {

d[i] = p[i - 1] + cost;

}

}

std::swap(p, d);

}

return 1.0 - (p[n] / MAX\_N);

}

# **Код graph.h**

#pragma once

#include <map>

#include <vector>

#include "node.h"

#include <string>

class node;

class graph

{

public:

graph(std::vector<std::string> log);

graph();

node get\_node(int i);

int get\_size();

void count\_binary\_frequency();

void count\_unary\_frequency();

void count\_unary\_routing();

void cut\_nodes(double cut\_off);

void count\_all\_metrics();

void count\_name\_similarity();

void count\_binary\_distance();

void count\_aggregate();

void count\_unary\_aggregate();

void count\_binary\_significance();

void count\_unary\_significance();

void solve\_conflicts(double cut\_value);

void solve\_conflict(int node1, int node2, double cut\_value);

double count\_rel(int node1, int node2, double sig);

void cut\_edges(double cut\_value);

double check\_number\_of\_components();

void dfs(int num);

void push\_user\_values(double binFrequency, double unFrequency,

double routingSig, double distanceSig, double nameSig);

private:

std::map<std::string, int> all\_names;

int size;

std::vector<node> nodes;

std::vector<std::string> parse\_line(std::string trace);

std::vector<bool> node\_visited;

};

# **Код graph.cpp**

#include "stdafx.h"

#include "graph.h"

#include "node.h"

#include <vector>

#include <map>

#include <algorithm>

std::vector<std::string> graph::parse\_line(std::string trace) {

std::vector<std::string> all;

int pointer = 0;

while (pointer < (int)trace.size()) {

std::string current\_name = "";

while (trace[pointer] != ';') {

current\_name += trace[pointer];

pointer++;

}

pointer++;

all.push\_back(current\_name);

}

return all;

}

graph::graph(std::vector<std::string> log) {

graph::size = 0;

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

std::vector<std::string> trace = graph::parse\_line(log[i]);

if ((int)trace.size() == 1) {

if (graph::all\_names.count(trace[0]) == 0) {

graph::all\_names[trace[0]] = graph::size;

graph::size++;

graph::nodes.push\_back(node(0, trace[0]));

}

graph::nodes[graph::all\_names[trace[0]]].inc\_frequency();

}

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

if (graph::all\_names.count(trace[j]) == 0) {

graph::all\_names[trace[j]] = graph::size;

graph::size++;

graph::nodes.push\_back(node(0, trace[j]));

}

if (graph::all\_names.count(trace[j + 1]) == 0) {

graph::all\_names[trace[j + 1]] = graph::size;

graph::size++;

graph::nodes.push\_back(node(0, trace[j + 1]));

}

int start = graph::all\_names[trace[j]];

int finish = graph::all\_names[trace[j + 1]];

nodes[start].inc\_out\_relations();

nodes[finish].inc\_in\_relations();

graph::nodes[start].push\_relation(finish);

graph::nodes[finish].push\_relation\_in(start);

if (j == 0) {

graph::nodes[start].inc\_frequency();

graph::nodes[finish].inc\_frequency();

} else

graph::nodes[finish].inc\_frequency();

}

}

}

graph::graph() {

graph::size = 0;

}

node graph::get\_node(int i) {

return graph::nodes[i];

}

int graph::get\_size() {

return (int)graph::nodes.size();

}

void graph::count\_unary\_frequency() {

int maximum = 0;

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

if (nodes[i].get\_node\_frequency() > maximum) {

maximum = nodes[i].get\_node\_frequency();

}

}

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

nodes[i].count\_unary\_frequency(maximum);

}

}

void graph::count\_unary\_routing()

{

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

nodes[i].count\_unary\_routing(this, i);

}

double maximum = 0;

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

if (nodes[i].get\_unary\_routing() > maximum)

maximum = nodes[i].get\_unary\_routing();

}

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

nodes[i].normalize\_unary\_routing(maximum);

}

}

void graph::cut\_nodes(double cut\_off)

{

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

if (nodes[i].get\_unary\_significanse() < cut\_off) {

nodes[i].set\_visible(false);

}

else {

int it = 0;

while (it < nodes[i].get\_number\_of\_relations()) {

if (nodes[nodes[i].get\_relation(it)].get\_unary\_significanse() < cut\_off && nodes[i].get\_relation\_visible(it)) {

int to = nodes[i].get\_relation(it);

for (int j = 0; j < nodes[to].get\_number\_of\_relations(); j++) {

if (nodes[to].get\_relation\_visible(j))

nodes[i].push\_relation(nodes[to].get\_relation(j));

}

//nodes[i].set\_rel\_visible(it, false);

}

it++;

}

}

}

}

void graph::count\_all\_metrics()

{

count\_binary\_frequency();

count\_unary\_frequency();

count\_name\_similarity();

count\_unary\_routing();

count\_binary\_distance();

count\_binary\_significance();

count\_unary\_significance();

}

void graph::count\_name\_similarity()

{

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

nodes[i].count\_name\_similarity(this);

}

}

void graph::count\_binary\_distance()

{

count\_aggregate();

count\_unary\_aggregate();

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

nodes[i].count\_binary\_distance(this, i);

}

double maximum = 0;

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

double val = nodes[i].get\_max\_distance();

if (val > maximum) {

maximum = val;

}

}

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

nodes[i].normalize\_distance(maximum);

}

}

void graph::count\_aggregate()

{

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

nodes[i].count\_aggregate();

}

double maximum = 0;

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

double val = nodes[i].get\_max\_aggregate();

if (val > maximum) {

maximum = val;

}

}

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

nodes[i].normalize\_aggregate(maximum);

}

}

void graph::count\_unary\_aggregate()

{

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

nodes[i].count\_unary\_aggregate();

}

double maximum = 0;

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

if (nodes[i].get\_unary\_aggregate() > maximum) {

maximum = nodes[i].get\_unary\_aggregate();

}

}

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

nodes[i].normalize\_unary\_aggregate(maximum);

}

}

void graph::count\_binary\_significance()

{

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

nodes[i].count\_binary\_significance();

}

double maximum = 0;

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

double val = nodes[i].get\_max\_binary\_significance();

if (val > maximum) {

maximum = val;

}

}

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

nodes[i].normalize\_binary\_significance(maximum);

}

}

void graph::count\_unary\_significance()

{

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

nodes[i].count\_unary\_significance();

}

double maximum = 0;

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

if (nodes[i].get\_unary\_significanse() > maximum) {

maximum = nodes[i].get\_unary\_significanse();

}

}

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

nodes[i].normalize\_unary\_significance(maximum);

}

}

void graph::solve\_conflicts(double cut\_value)

{

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

for (int j = 0; j < nodes[i].get\_number\_of\_relations(); j++) {

int to = nodes[i].get\_relation(j);

for (int k = 0; k < nodes[i].get\_number\_of\_in\_relations(); k++) {

if (to == nodes[i].get\_in\_relation(k)) {

solve\_conflict(i, to, cut\_value);

}

}

}

}

}

double graph::count\_rel(int node1, int node2, double sig)

{

double sum\_of\_out\_node1\_significances = 0;

double sum\_of\_in\_node2\_significances = 0;

for (int i = 0; i < nodes[node1].get\_number\_of\_relations(); i++) {

sum\_of\_out\_node1\_significances += nodes[node1].get\_binary\_significance(i);

}

for (int i = 0; i < nodes[node2].get\_number\_of\_in\_relations(); i++) {

int in\_rel = nodes[node2].get\_in\_relation(i);

for (int j = 0; j < nodes[in\_rel].get\_number\_of\_relations(); j++) {

if (node2 == nodes[in\_rel].get\_relation(j)) {

sum\_of\_in\_node2\_significances += nodes[in\_rel].get\_binary\_significance(j);

}

}

}

return 0.5 \* (sig / sum\_of\_out\_node1\_significances) + 0.5 \* (sig / sum\_of\_in\_node2\_significances);

}

void graph::cut\_edges(double cut\_value)

{

int cnt\_at\_all = 0;

node\_visited.resize(size);

node\_visited.assign(size, false);

int cnt = check\_number\_of\_components();

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

for (int j = 0; j < nodes[i].get\_number\_of\_relations(); j++) {

if (nodes[i].get\_binary\_significance(j) < cut\_value && nodes[i].get\_relation\_visible(j)) {

nodes[i].set\_rel\_visible(j, false);

int cnt2 = check\_number\_of\_components();

if (cnt != cnt2) {

nodes[i].set\_rel\_visible(j, true);

}

}

}

}

}

double graph::check\_number\_of\_components()

{

double cnt = 0;

node\_visited.assign(size, false);

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

if (!node\_visited[i]) {

dfs(i);

cnt++;

}

}

return cnt;

}

void graph::dfs(int num)

{

node\_visited[num] = true;

for (int i = 0; i < nodes[num].get\_number\_of\_relations(); i++) {

if (!node\_visited[nodes[num].get\_relation(i)] && nodes[num].get\_relation\_visible(i)) {

dfs(nodes[num].get\_relation(i));

}

}

}

void graph::push\_user\_values(double binFrequency, double unFrequency, double routingSig, double distanceSig, double nameSig)

{

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

nodes[i].push\_user\_values(binFrequency, unFrequency, routingSig, distanceSig, nameSig);

}

}

void graph::solve\_conflict(int node1, int node2, double cut\_value)

{

double sigA\_B = 0;

for (int i = 0; i < nodes[node1].get\_number\_of\_relations(); i++) {

if (nodes[node1].get\_relation(i) == node2) {

sigA\_B = nodes[node1].get\_binary\_significance(i);

}

}

double sigB\_A = 0;

for (int i = 0; i < nodes[node2].get\_number\_of\_relations(); i++) {

if (nodes[node2].get\_relation(i) == node1) {

sigB\_A = nodes[node2].get\_binary\_significance(i);

}

}

double relA\_B = (node1, node2, sigA\_B);

double relB\_A = (node2, node1, sigB\_A);

if (std::abs(relA\_B - relB\_A) > cut\_value) {

if (relA\_B > relB\_A) {

nodes[node2].set\_rel\_visible(node1, false);

}

else {

nodes[node1].set\_rel\_visible(node2, false);

}

}

}

void graph::count\_binary\_frequency()

{

int maximum = 0;

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

for (int j = 0; j < nodes[i].get\_number\_of\_relations(); j++) {

int cur = nodes[i].get\_relation\_frequency(j);

if (cur > maximum)

maximum = cur;

}

}

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

nodes[i].count\_binary\_frequency(maximum);

}

}

# **Код dotfile.h**

#pragma once

#include <fstream>

#include <string>

#include "graph.h"

class dotfile

{

public:

void create\_dot(std::string name, graph model);

private:

std::ofstream fout;

};

# **Код dotfile.cpp**

#include "stdafx.h"

#include "dotfile.h"

#include <iomanip>

void dotfile::create\_dot(std::string name, graph model) {

dotfile::fout.open(name);

fout << "digraph model {" << '\n';

fout << "ratio = fill;" << '\n' << "node [style = filled];" << '\n';

for (int i = 0; i < model.get\_size(); i++) {

if (model.get\_node(i).get\_visible()) {

dotfile::fout << model.get\_node(i).get\_name()

<< " [label = \"" << std::fixed << std::setprecision(3) << model.get\_node(i).get\_name()

<< ':' << model.get\_node(i).get\_unary\_significanse()

<< "\", color=\"0.650 0.200 1.000\"]" << '\n';

for (int j = 0; j < model.get\_node(i).get\_number\_of\_relations(); j++) {

if (model.get\_node(model.get\_node(i).get\_relation(j)).get\_visible() && model.get\_node(i).get\_relation\_visible(j)) {

dotfile::fout << model.get\_node(i).get\_name() << " -> " <<

model.get\_node(model.get\_node(i).get\_relation(j)).get\_name() << '\n';

}

}

}

}

fout << "}";

fout.close();

}

# **Код unary\_metric.h**

#pragma once

class unary\_metric

{

public:

unary\_metric();

void count\_frequency(int frequency, int maximum);

double get\_normalized\_frequency();

double get\_normalized\_routing();

void count\_unary\_routing(double val);

void normalize\_unary\_routing(double maximum);

void count\_aggregate();

double get\_aggregate();

void normalize\_aggregate(double maximum);

void count\_unary\_significance();

double get\_unary\_significance();

void normalize\_unary\_significance(double maximum);

void set\_user\_values(double frequency, double routing);

private:

double normalized\_frequncy;

double normalized\_routing;

double aggregate;

double unary\_significance;

double user\_frequency;

double user\_routing;

};

# **Код unary\_metric.cpp**

#include "stdafx.h"

#include "unary\_metric.h"

unary\_metric::unary\_metric()

{

unary\_metric::normalized\_frequncy = 0;

unary\_metric::normalized\_routing = 0;

unary\_significance = 0;

aggregate = 0;

user\_frequency = 0;

user\_routing = 0;

}

void unary\_metric::count\_frequency(int frequency, int maximum) {

unary\_metric::normalized\_frequncy = (double)frequency / (double)maximum;

}

double unary\_metric::get\_normalized\_frequency()

{

return normalized\_frequncy;

}

double unary\_metric::get\_normalized\_routing()

{

return normalized\_routing;

}

void unary\_metric::count\_unary\_routing(double val)

{

normalized\_routing = val;

}

void unary\_metric::normalize\_unary\_routing(double maximum)

{

if (maximum != 0) {

normalized\_routing /= maximum;

}

}

void unary\_metric::count\_aggregate()

{

aggregate = normalized\_frequncy + normalized\_routing;

}

double unary\_metric::get\_aggregate()

{

return aggregate;

}

void unary\_metric::normalize\_aggregate(double maximum)

{

if (maximum != 0)

aggregate = aggregate / maximum;

}

void unary\_metric::count\_unary\_significance()

{

//Here must be coef

unary\_significance = user\_frequency \* normalized\_frequncy + user\_routing \* normalized\_routing;

}

double unary\_metric::get\_unary\_significance()

{

return unary\_significance;

}

void unary\_metric::normalize\_unary\_significance(double maximum)

{

if (maximum != 0) {

unary\_significance = unary\_significance / maximum;

}

}

void unary\_metric::set\_user\_values(double frequency, double routing)

{

user\_frequency = frequency;

user\_routing = routing;

}

# **Код binary\_metric.h**

#pragma once

#include <string>

class binary\_metric

{

public:

binary\_metric();

void count\_binary\_frequency(int frequency, int maximum);

double get\_binary\_frequency();

double get\_binary\_significance();

void count\_binary\_significance();

void count\_name\_similarity(std::string first\_name, std::string second\_name);

double get\_name\_similarity();

void normalize\_binary\_distance(double maximum);

double get\_binary\_distance();

void count\_aggregate();

void normalize\_aggregate(double maximum);

void count\_binary\_distance(double me, double rel, double mid);

double get\_aggregate();

void normalize\_binary\_significance(double maximum);

void set\_user\_values(double binFrequency, double distanceSig, double nameSig);

private:

double binary\_frequency;

double binary\_distance;

double binary\_significance;

double name\_similar;

double aggregate;

double user\_frequency;

double user\_distance;

double user\_nameSig;

};

# **Код binary\_metric.cpp**

#include "stdafx.h"

#include "binary\_metric.h"

#include "name\_similarity.h"

binary\_metric::binary\_metric()

{

binary\_significance = 0;

binary\_frequency = 0;

name\_similar = 0;

binary\_distance = 0;

aggregate = 0;

user\_distance = 0;

user\_frequency = 0;

user\_nameSig = 0;

}

void binary\_metric::count\_binary\_frequency(int frequency, int maximum)

{

binary\_frequency = (double)frequency / (double)maximum;

}

double binary\_metric::get\_binary\_frequency()

{

return binary\_frequency;

}

double binary\_metric::get\_binary\_significance()

{

return binary\_significance;

}

void binary\_metric::count\_binary\_significance()

{

//here must be coef

binary\_significance = user\_nameSig \* name\_similar

+ user\_frequency \* binary\_frequency

+ user\_distance \* binary\_distance;

}

void binary\_metric::count\_name\_similarity(std::string first\_name, std::string second\_name)

{

name\_similarity name\_sim;

name\_similar = name\_sim.get\_name\_similarity(first\_name, second\_name);

}

double binary\_metric::get\_name\_similarity()

{

return name\_similar;

}

void binary\_metric::normalize\_binary\_distance(double maximum)

{

if (maximum != 0) {

binary\_distance = binary\_distance / maximum;

}

}

double binary\_metric::get\_binary\_distance()

{

return binary\_distance;

}

void binary\_metric::count\_aggregate()

{

aggregate = name\_similar + binary\_frequency;

}

double binary\_metric::get\_aggregate()

{

return aggregate;

}

void binary\_metric::normalize\_aggregate(double maximum)

{

if (maximum > 0) {

aggregate = aggregate / maximum;

}

}

void binary\_metric::normalize\_binary\_significance(double maximum)

{

if (maximum != 0) {

binary\_significance = binary\_significance / maximum;

}

}

void binary\_metric::set\_user\_values(double binFrequency, double distanceSig, double nameSig)

{

user\_distance = distanceSig;

user\_frequency = binFrequency;

user\_nameSig = nameSig;

}

void binary\_metric::count\_binary\_distance(double me, double rel, double mid)

{

if ((me + rel) != 0) {

binary\_distance = ((me - mid) + (rel - mid)) / (me + rel);

if (binary\_distance < 0) {

binary\_distance = -binary\_distance;

}

}

else

binary\_distance = 0;

}