Ullmann算法

[main.c 1](#_Toc1673232569)

[graph.cpp 3](#_Toc1822366210)

[Graph.h 4](#_Toc144585011)

[matrix.h 6](#_Toc1989054197)

[Seperater.cpp 10](#_Toc713958818)

[Seperater.h 11](#_Toc2106661716)

[Ullman.h 12](#_Toc1290206685)

[Ullman\_build.cpp 14](#_Toc745732963)

[Ullman\_init.cpp 15](#_Toc916471191)

[Ullman\_query.cpp 18](#_Toc481824476)

[Ullman\_utils.cpp 23](#_Toc90467272)

[Ullman\_utils.h 25](#_Toc1582701107)

2015-11-18

## main.c

#include <string>

#include "sys/time.h"

#include "ullman.h"

#include "ullman\_utils.h"

#include "common.h"

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

{

ullman::parameters\_t parameters;

//./ullman -database ../data/ -query ../data/

size\_t nargh = ullman::get\_parameters(&parameters, argc, argv);

//[parameters] - [number of default parameters]

if (nargh != ullman::get\_narg() - 1) {

exit(ULLMAN\_ERROR);

}

std::string seperator;

ullman::get\_seperator(&parameters, seperator);

ullman::print\_parameters(&parameters);

timeval t1, t2;

double elapsed\_time = 0.0f;

gettimeofday(&t1, NULL);

ullman::Isomorphism isomorphism(parameters.database, parameters.query, seperator.c\_str());

if (ULLMAN\_SUCCESS != isomorphism.execute()) {

fprintf(stderr, "not successful execution!");

exit(ULLMAN\_ERROR);

}

gettimeofday(&t2, NULL);

elapsed\_time = (t2.tv\_sec - t1.tv\_sec) \* 1000.0;

elapsed\_time += (t2.tv\_usec - t1.tv\_usec) / 1000.0;

printf("elapsed time %f\n", elapsed\_time);

return 0;

}

## graph.cpp

#include "graph.h"

namespace ullman {

const std::map<int32\_t, std::vector<size\_t> > Graph::get\_vertex\_label\_map()

{

if (!\_m\_vertex\_label\_map.size()) {

construct\_vertex\_label\_map();

}

return \_m\_vertex\_label\_map;

}

void Graph::construct\_vertex\_label\_map()

{

for (size\_t i = 0; i < \_m\_vertice.size(); ++i) {

\_m\_vertex\_label\_map[\_m\_vertice[i].label].push\_back(\_m\_vertice[i].id);

}

}

const std::map<struct edge\_label\_list\_t, std::vector<struct edge\_t \*> >&

Graph::get\_edge\_label\_map()

{

if (!\_m\_edge\_label\_map.size()) {

construct\_edge\_label\_map();

}

return \_m\_edge\_label\_map;

}

//Todo

void Graph::construct\_edge\_label\_map()

{

for (size\_t i = 0; i < \_m\_vertice.size(); ++i) {

for (size\_t j = 0; j < \_m\_vertice[i].edges.size(); ++j) {

struct edge\_t \*edge = &\_m\_vertice[i].edges[j];

struct edge\_label\_list\_t edge\_label\_list;

edge\_label\_list.from\_label = \_m\_vertice[edge->from].label;

edge\_label\_list.edge\_label = edge->label;

edge\_label\_list.to\_label = \_m\_vertice[edge->to].label;

\_m\_edge\_label\_map[edge\_label\_list].push\_back(edge);

}

}

}

}//namespace ullman

## Graph.h

#ifndef GRAPH\_H

#define GRAPH\_H

#include <map>

#include <algorithm>

#include "common.h"

namespace ullman {

struct edge\_t {

size\_t from;

int32\_t label;

size\_t to;

size\_t id;

};

struct vertex\_t {

size\_t id;

int32\_t label;

std::vector<struct edge\_t> edges;

};

typedef std::vector<struct vertex\_t> Vertice;

struct edge\_label\_list\_t {

size\_t from\_label;

int32\_t edge\_label;

size\_t to\_label;

bool operator < (const struct edge\_label\_list\_t& edge\_label\_list) const

{

if (from\_label != edge\_label\_list.from\_label) {

return from\_label < edge\_label\_list.from\_label;

} else {

if (edge\_label != edge\_label\_list.edge\_label) {

return edge\_label < edge\_label\_list.edge\_label;

} else {

return to\_label < edge\_label\_list.to\_label;

}

}

}

};

class Graph {

public:

explicit Graph() : id(0), \_m\_nedges(0) {};

explicit Graph(size\_t size) : id(0), \_m\_nedges(0), \_m\_vertice(size) {};

size\_t size() const

{

return \_m\_vertice.size();

}

void resize(size\_t s)

{

\_m\_vertice.resize(s);

}

void set\_id(size\_t id)

{

this->id = id;

}

size\_t get\_id() const

{

return id;

}

void set\_nedges(size\_t size)

{

\_m\_nedges = size;

}

size\_t get\_nedges() const

{

return \_m\_nedges;

}

void set\_vertice(const Vertice& vertice)

{

this->\_m\_vertice = vertice;

}

struct vertex\_t& get\_vertex(size\_t idx) {

return \_m\_vertice[idx];

}

const struct vertex\_t& get\_vertex(size\_t idx) const {

return \_m\_vertice[idx];

}

const std::map<struct edge\_label\_list\_t, std::vector<struct edge\_t \*> >& get\_edge\_label\_map();

const std::map<int32\_t, std::vector<size\_t> > get\_vertex\_label\_map();

void clear()

{

id = 0;

\_m\_vertice.clear();

}

void sort\_vertex\_by\_degree() {

std::sort(\_m\_vertice.begin(), \_m\_vertice.end(), sort\_vertex);

}

private:

struct sort\_vertex\_t {

bool operator () (const struct vertex\_t& vertex\_a,

const struct vertex\_t& vertex\_b)

{

return vertex\_a.edges.size() > vertex\_b.edges.size();

}

} sort\_vertex;

void construct\_edge\_label\_map();

void construct\_vertex\_label\_map();

private:

size\_t \_m\_nedges;

size\_t id;

Vertice \_m\_vertice;

std::map<struct edge\_label\_list\_t, std::vector<struct edge\_t \*> > \_m\_edge\_label\_map;

std::map<int32\_t, std::vector<size\_t> > \_m\_vertex\_label\_map;

};

}//namespace ullman

#endif

## matrix.h

#ifndef MATRIX\_H

#define MATRIX\_H

#include <vector>

#include <map>

#include <string>

#include <sstream>

namespace ullman {

//efficient matrix

template<typename T>

class Matrix {

public:

Matrix<T>(size\_t nrows, size\_t ncolumn) :

\_m\_nrows(nrows), \_m\_ncolumns(ncolumn), \_m\_change(true),

\_m\_value(nrows, std::vector<T>(ncolumn, 0)) {};

Matrix<T>() : \_m\_nrows(0), \_m\_ncolumns(0), \_m\_change(true) {};

inline Matrix<T> operator \* (const Matrix<T>& other) const;

inline Matrix<T> transposition() const;

inline void set(size\_t i, size\_t j, T v);

inline T get(size\_t i, size\_t j) const;

inline void fill(T v);

inline void fill(size\_t row, T v);

void resize(size\_t size\_i, size\_t size\_j);

inline void clear();

inline const char \* c\_str();

//{nrows, ncolumns}

std::pair<size\_t, size\_t> size() const

{

return std::make\_pair(\_m\_nrows, \_m\_ncolumns);

}

private:

size\_t \_m\_ncolumns, \_m\_nrows;

std::string \_m\_str;

bool \_m\_change;

std::vector<std::vector<T> > \_m\_value;

};

template <typename T>

inline Matrix<T> Matrix<T>::transposition() const

{

Matrix<T> matrix(\_m\_ncolumns, \_m\_nrows);

for (size\_t i = 0; i < \_m\_nrows; ++i) {

for (size\_t j = 0; j < \_m\_ncolumns; ++j) {

matrix.set(j, i, this->get(i, j));

}

}

return matrix;

}

template <typename T>

inline Matrix<T> Matrix<T>::operator \* (const Matrix<T>& other) const

{

std::pair<size\_t, size\_t> other\_size = other.size();

if (other\_size.first != \_m\_ncolumns)

return Matrix<T>();

Matrix<T> matrix(\_m\_nrows, other\_size.second);

for (size\_t i = 0; i < \_m\_nrows; ++i) {

for (size\_t j = 0; j < other\_size.second; ++j) {

T tmp = 0;

for (size\_t k = 0; k < \_m\_ncolumns; ++k) {

tmp += this->get(i, k) \* other.get(k, j);

}

matrix.set(i, j, tmp);

}

}

return matrix;

}

template <typename T>

inline void Matrix<T>::set(size\_t i, size\_t j, T v)

{

if (i < \_m\_nrows && i >= 0 && j < \_m\_ncolumns && j >= 0) {

\_m\_value[i][j] = v;

\_m\_change = true;

}

}

template <typename T>

inline T Matrix<T>::get(size\_t i, size\_t j) const

{

if (i < \_m\_nrows && i >= 0 && j < \_m\_ncolumns && j >= 0) {

return \_m\_value[i][j];

}

return 0;

}

template <typename T>

inline void Matrix<T>::fill(T v)

{

for (size\_t i = 0; i < \_m\_nrows; ++i) {

\_m\_value[i].assign(\_m\_ncolumns, v);

}

\_m\_change = true;

}

template <typename T>

inline void Matrix<T>::fill(size\_t nrow, T v)

{

if (nrow < \_m\_nrows) {

\_m\_value[nrow].assign(\_m\_ncolumns, v);

}

}

template <typename T>

void Matrix<T>::resize(size\_t size\_i, size\_t size\_j)

{

if (\_m\_value.size() < size\_i) {

\_m\_value.resize(size\_i, std::vector<T>(\_m\_ncolumns, 0));

}

\_m\_nrows = size\_i;

if (\_m\_value[0].size() < size\_j) {

for (size\_t i = 0; i < \_m\_nrows; ++i) {

\_m\_value[i].resize(size\_j, 0);

}

}

\_m\_ncolumns = size\_j;

\_m\_change = true;

}

template <typename T>

inline void Matrix<T>::clear()

{

for (size\_t i = 0; i < \_m\_nrows; ++i) {

\_m\_value[i].clear();

}

\_m\_ncolumns = 0;

\_m\_nrows = 0;

\_m\_change = true;

}

template <typename T>

inline const char \* Matrix<T>::c\_str()

{

if (\_m\_change) {

std::stringstream ss;

for (size\_t i = 0; i < \_m\_nrows; ++i) {

for (size\_t j = 0; j < \_m\_ncolumns; ++j) {

ss << \_m\_value[i][j] << " ";

}

ss << "\n";

}

\_m\_str = ss.str();

}

\_m\_change = false;

return \_m\_str.c\_str();

}

}//namespace ullman

#endif

## Seperater.cpp

#include "seperator.h"

namespace ullman {

static const uint32\_t MAX\_LENGTH = 1024;

uint32\_t Seperator::seperate(const char\* file\_path, Buffer& stream) {

char line[MAX\_LENGTH];

FILE \*fp = fopen(file\_path, "r+");

if (fp == NULL) {

fprintf(stderr, "error occurs when reading file %s\n", file\_path);

exit(ULLMAN\_ERROR);

}

uint32\_t ncount = 0;

while (fgets(line, MAX\_LENGTH - 1, fp) != NULL) {

stream.resize(ncount + 1);

char \*pch = NULL;

pch = strtok(line, \_m\_token);

while (pch != NULL) {

stream[ncount].push\_back(std::string(pch));

pch = strtok(NULL, \_m\_token);

}

ncount++;

}

fclose(fp);

return ncount;

};

}//namespace ullman

## Seperater.h

#ifndef SEPERATOR\_H

#define SEPERATOR\_H

#include "common.h"

namespace ullman {

class Seperator {

public:

Seperator(const char\* token): \_m\_token(token) {

};

uint32\_t seperate(const char \*file\_path, Buffer& stream);

private:

const char\* \_m\_token;

};

}//namespace ullman

#endif //SEPERATOR\_H

## Ullman.h

#ifndef ULLMAN\_H

#define ULLMAN\_H

#include <set>

#include "graph.h"

#include "matrix.h"

#include "seperator.h"

#include "common.h"

namespace ullman {

class Graph;

class Database {

public:

void push\_graph(const Graph& graph)

{

graphs.push\_back(graph);

}

const Graph& get\_graph(size\_t id) const

{

return graphs[id];

}

Graph& get\_graph(size\_t id)

{

return graphs[id];

}

size\_t size() const

{

return graphs.size();

}

void sort()

{

for (size\_t i = 0; i < graphs.size(); ++i)

graphs[i].sort\_vertex\_by\_degree();

}

private:

std::vector<Graph> graphs;

};

class Isomorphism {

public:

explicit Isomorphism(const char \*database, const char \*query, const char \*sep\_type) :

\_m\_file\_data(database), \_m\_file\_query(query) , \_m\_seperator(sep\_type) {};

UllmanReturnCode execute();

private:

UllmanReturnCode read\_input(const Buffer& input, Database& database);

UllmanReturnCode query();

bool construct\_match(Graph& query\_graph, Graph& entry\_graph);

void dfs\_search(size\_t idx, Matrix<bool> matrix);

void build\_matrix(Matrix<bool>& matrix, size\_t nrows, size\_t ncolumns);

void build\_matrix(Matrix<bool>& matrix, const Graph& graph);

bool judge(const Matrix<bool>& matrix);

void refine(Matrix<bool>& matrix, size\_t start);

UllmanReturnCode output();

private:

const char \*\_m\_file\_data;

const char \*\_m\_file\_query;

Seperator \_m\_seperator;

Database \_m\_database;

Database \_m\_query;

//whether current columns are used

Matrix<bool> \_m\_columns;

//whether all the possible olumns

std::vector<int32\_t> \_m\_columns\_used;

//corresponding matrix b

Matrix<bool> matrix\_b;

//corresponding matrix a

Matrix<bool> matrix\_a;

std::vector<std::vector<size\_t> > \_m\_output;

size\_t \_m\_cur\_graph\_id;

size\_t \_m\_cur\_query\_id;

bool \_m\_cur\_find;

};

}//namespace ullman

#endif

## Ullman\_build.cpp

#include "ullman.h"

namespace ullman {

//Todo

void Isomorphism::build\_matrix(Matrix<bool>& matrix, size\_t nrows, size\_t ncolumns)

{

matrix.clear();

matrix.resize(nrows, ncolumns);

for (size\_t i = 0; i < \_m\_columns\_used.size(); ++i) {

if (\_m\_columns\_used[i] != -1)

matrix.set(\_m\_columns\_used[i], i, 1);

}

}

//Todo

void Isomorphism::build\_matrix(Matrix<bool>& matrix, const Graph& graph)

{

matrix.clear();

matrix.resize(graph.size(), graph.size());

for (size\_t i = 0; i < graph.size(); ++i) {

const struct vertex\_t& vertex = graph.get\_vertex(i);

size\_t from\_id = vertex.id;

for (size\_t j = 0; j < vertex.edges.size(); ++j) {

size\_t to\_id = vertex.edges[j].to;

matrix.set(from\_id, to\_id, 1);

}

}

}

}//namespace ullman

## Ullman\_init.cpp

#include "sys/time.h"

#include "ullman.h"

namespace ullman {

UllmanReturnCode Isomorphism::execute()

{

Buffer ullman\_database;

Buffer ullman\_query;

\_m\_seperator.seperate(\_m\_file\_data, ullman\_database);

\_m\_seperator.seperate(\_m\_file\_query, ullman\_query);

if (ULLMAN\_SUCCESS != read\_input(ullman\_database, \_m\_database)) {

fprintf(stderr, "read input database error!\n");

return ULLMAN\_ERROR;

}

//To-do: sort the vertices

if (ULLMAN\_SUCCESS != read\_input(ullman\_query, \_m\_query)) {

fprintf(stderr, "read input query error!\n");

return ULLMAN\_ERROR;

}

timeval t1, t2;

double elapsed\_time = 0.0f;

gettimeofday(&t1, NULL);

if (ULLMAN\_SUCCESS != query()) {

fprintf(stderr, "find isomorphism error!\n");

return ULLMAN\_ERROR;

}

gettimeofday(&t2, NULL);

elapsed\_time = (t2.tv\_sec - t1.tv\_sec) \* 1000.0;

elapsed\_time += (t2.tv\_usec - t1.tv\_usec) / 1000.0;

//printf("elapsed time->execute %f\n", elapsed\_time);

if (ULLMAN\_SUCCESS != output()) {

fprintf(stderr, "output error!\n");

return ULLMAN\_ERROR;

}

return ULLMAN\_SUCCESS;

}

UllmanReturnCode Isomorphism::read\_input(const Buffer& buffer, Database& database)

{

Graph graph;

Vertice vertice;

size\_t graph\_idx = 0;

size\_t edge\_id = 0;

for (size\_t i = 0; i < buffer.size(); ++i) {

if (buffer[i][0] == "t") {

if (i != 0) {

graph.set\_nedges(edge\_id);

graph.set\_vertice(vertice);

edge\_id = 0;

database.push\_graph(graph);

graph.clear();

vertice.clear();

}

char indicator, seperator;

size\_t idx;

indicator = buffer[i][0][0];

seperator = buffer[i][1][0];

sscanf(buffer[i][2].c\_str(), "%zu", &idx);

if (graph\_idx != idx) {

fprintf(stderr, "reading buffer warning! %zu %zu\n", graph\_idx, idx);

return ULLMAN\_WARNING;

}

//debug

//printf("t # %zu\n", idx);

graph.set\_id(idx);

++graph\_idx;

} else if (buffer[i][0] == "v") {

char indicator;

size\_t id;

int32\_t label;

indicator = buffer[i][0][0];

sscanf(buffer[i][1].c\_str(), "%zu", &id);

sscanf(buffer[i][2].c\_str(), "%d", &label);

//debug

//printf("v %zu %d\n", id, label);

struct vertex\_t vertex;

vertex.id = id;

vertex.label = label;

vertice.push\_back(vertex);

} else if (buffer[i][0] == "e") {

char indicator;

size\_t from, to;

int32\_t label;

indicator = buffer[i][0][0];

sscanf(buffer[i][1].c\_str(), "%zu", &from);

sscanf(buffer[i][2].c\_str(), "%zu", &to);

sscanf(buffer[i][3].c\_str(), "%d", &label);

//debug

//printf("e %zu %zu %d\n", from, to, label);

struct edge\_t edge;

edge.from = from;

edge.to = to;

edge.label = label;

edge.id = edge\_id;

++edge\_id;

//first edge

vertice[from].edges.push\_back(edge);

//second edge

edge.from = to;

edge.to = from;

vertice[to].edges.push\_back(edge);

} else {

fprintf(stderr, "reading buffer warning!\n");

}

}

graph.set\_vertice(vertice);

database.push\_graph(graph);

return ULLMAN\_SUCCESS;

}

UllmanReturnCode Isomorphism::output()

{

size\_t sum = 0;

for (size\_t i = 0; i < \_m\_output.size(); ++i) {

printf("t # %zu : %zu\n", i, \_m\_output[i].size());

sum += \_m\_output[i].size();

for (size\_t j = 0; j < \_m\_output[i].size(); ++j) {

printf("%zu ", \_m\_output[i][j]);

}

printf("\n\n");

}

printf("\nsum: %zu\n", sum);

return ULLMAN\_SUCCESS;

}

}//namespace ullman

## Ullman\_query.cpp

#include "ullman.h"

namespace ullman {

UllmanReturnCode Isomorphism::query()

{

\_m\_output.resize(\_m\_query.size());

for (size\_t i = 0; i < \_m\_query.size(); ++i) {

Graph& query\_graph = \_m\_query.get\_graph(i);

\_m\_cur\_query\_id = query\_graph.get\_id();

build\_matrix(matrix\_a, query\_graph);

for (size\_t j = 0; j < \_m\_database.size(); ++j) {

Graph& entry\_graph = \_m\_database.get\_graph(j);

\_m\_cur\_graph\_id = entry\_graph.get\_id();

#ifdef DEBUG

printf("query\_id %zu, graph\_id %zu\n", \_m\_cur\_query\_id, \_m\_cur\_graph\_id);

//printf("query\_graph.size() %zu, entry\_graph.size() %zu\n", query\_graph.size(), entry\_graph.size());

#endif

\_m\_columns.resize(query\_graph.size(), entry\_graph.size());

\_m\_columns\_used.resize(entry\_graph.size(), -1);

build\_matrix(matrix\_b, entry\_graph);

if (!construct\_match(query\_graph, entry\_graph)) {

\_m\_columns.clear();

continue;

}

#ifdef DEBUG

// printf("%s\n", \_m\_columns.c\_str());

// getchar();

// getchar();

#endif

\_m\_cur\_find = false;

for (size\_t k = 0; k < \_m\_columns.size().second; ++k) {

if (!\_m\_columns.get(0, k))

continue;

Matrix<bool> matrix = \_m\_columns;

matrix.fill(0, 0);

matrix.set(0, k, 1);

\_m\_columns\_used[k] = 0;

dfs\_search(1, matrix);

\_m\_columns\_used[k] = -1;

}

\_m\_columns\_used.clear();

\_m\_columns.clear();

}

}

return ULLMAN\_SUCCESS;

}

bool Isomorphism::construct\_match(Graph& query\_graph, Graph& entry\_graph)

{

//prune : the size

if (query\_graph.size() > entry\_graph.size())

return false;

const std::map<struct edge\_label\_list\_t, std::vector<struct edge\_t \*> >& query\_list =

query\_graph.get\_edge\_label\_map();

std::map<struct edge\_label\_list\_t, std::vector<struct edge\_t \*> >::const\_iterator query\_list\_it =

query\_list.begin();

const std::map<struct edge\_label\_list\_t, std::vector<struct edge\_t \*> >& entry\_list =

entry\_graph.get\_edge\_label\_map();

std::map<struct edge\_label\_list\_t, std::vector<struct edge\_t \*> >::const\_iterator entry\_list\_it =

entry\_list.begin();

while (query\_list\_it != query\_list.end() && entry\_list\_it != entry\_list.end()) {

if (query\_list\_it->first < entry\_list\_it->first) {

++query\_list\_it;

} else if (entry\_list\_it->first < query\_list\_it->first) {

++entry\_list\_it;

} else {

for (size\_t i = 0; i < (query\_list\_it->second).size(); ++i) {

struct edge\_t \*query\_edge = (query\_list\_it->second)[i];

size\_t query\_from = query\_edge->from;

size\_t query\_to = query\_edge->to;

for (size\_t j = 0; j < (entry\_list\_it->second).size(); ++j) {

struct edge\_t \*entry\_edge = (entry\_list\_it->second)[j];

size\_t entry\_from = entry\_edge->from;

size\_t entry\_to = entry\_edge->to;

\_m\_columns.set(query\_from, entry\_from, 1);

\_m\_columns.set(query\_to, entry\_to, 1);

}

}

++entry\_list\_it;

++query\_list\_it;

}

}

//prune : degree

for (size\_t i = 0; i < \_m\_columns.size().first; ++i) {

for (size\_t j = 0; j < \_m\_columns.size().second; ++j) {

if (\_m\_columns.get(i, j) == 1) {

size\_t degree\_from = query\_graph.get\_vertex(i).edges.size();

size\_t degree\_to = entry\_graph.get\_vertex(j).edges.size();

if (degree\_to < degree\_from)

\_m\_columns.set(i, j, 0);

}

}

}

//prune : refine

refine(\_m\_columns, 0);

#ifdef DEBUG

//printf("after refine\n%s\n", \_m\_columns.c\_str());

#endif

//prune : the mapping of zeros

for (size\_t i = 0; i < \_m\_columns.size().first; ++i) {

bool find = false;

for (size\_t j = 0; j < \_m\_columns.size().second; ++j) {

if (\_m\_columns.get(i, j) == 1) {

find = true;

break;

}

}

if (!find) {

return false;

}

}

return true;

}

void Isomorphism::dfs\_search(size\_t idx, Matrix<bool> matrix)

{

//prune 3: if only one of the mapping has found

if (\_m\_cur\_find)

return;

#ifdef DEBUG

// printf("idx %zu, size %zu\n", idx, \_m\_columns.size().first);

// printf("query\_id %zu, graph\_id %zu\n", \_m\_cur\_query\_id, \_m\_cur\_graph\_id);

//printf("%s\n", matrix.c\_str());

#endif

if (idx == matrix.size().first) {

Matrix<bool> matrix\_c = matrix \* ((matrix \* matrix\_b).transposition());

#ifdef DEBUG

// printf("idx %zu, size %zu\n", idx, \_m\_columns.size().first);

// printf("query\_id %zu, graph\_id %zu\n", \_m\_cur\_query\_id, \_m\_cur\_graph\_id);

// printf("m\n%s\n", \_m\_columns.c\_str());

// printf("b\n%s\n", matrix\_b.c\_str());

// printf("m \* b\n%s\n", (\_m\_columns \* matrix\_b).c\_str());

// printf("m \* b.trans\n%s\n", ((\_m\_columns \* matrix\_b).transposition()).c\_str());

// printf("m \* m \* b.trans\n%s\n", (\_m\_columns \* ((\_m\_columns \* matrix\_b).transposition())).c\_str());

// printf("c\n%s\n", matrix\_c.c\_str());

// printf("a\n%s\n", matrix\_a.c\_str());

#endif

if (judge(matrix\_c)) {

\_m\_cur\_find = true;

\_m\_output[\_m\_cur\_query\_id].push\_back(\_m\_cur\_graph\_id);

}

return ;

}

refine(matrix, idx);

#ifdef DEBUG

//printf("after refine\n%s\n", matrix.c\_str());

#endif

for (size\_t i = 0; i < matrix.size().second; ++i) {

if (\_m\_columns\_used[i] != -1 || !matrix.get(idx, i))

continue;

Matrix<bool> next\_matrix = matrix;

next\_matrix.fill(idx, 0);

next\_matrix.set(idx, i, 1);

\_m\_columns\_used[i] = idx;

dfs\_search(idx + 1, next\_matrix);

\_m\_columns\_used[i] = -1;

}

}

bool Isomorphism::judge(const Matrix<bool>& matrix\_c)

{

for (size\_t i = 0; i < matrix\_a.size().first; ++i) {

for (size\_t j = 0; j < matrix\_a.size().second; ++j) {

if (matrix\_a.get(i, j) == 0)

continue;

if (matrix\_c.get(i, j) == 0)

return false;

}

}

return true;

}

void Isomorphism::refine(Matrix<bool>& matrix, size\_t start)

{

const Graph& query\_graph = \_m\_query.get\_graph(\_m\_cur\_query\_id);

//Todo: optimize refine

while (true) {

bool change = false;

for (size\_t i = start; i < matrix.size().first; ++i) {

for (size\_t j = 0; j < matrix.size().second; ++j) {

if (matrix.get(i, j) == 0)

continue;

const struct vertex\_t& vertex = query\_graph.get\_vertex(i);

bool find = true;

for (size\_t k = 0; k < vertex.edges.size(); ++k) {

size\_t x = vertex.edges[k].to;

bool non\_zero = false;

for (size\_t y = 0; y < matrix.size().second; ++y) {

if (matrix.get(x, y) && matrix\_b.get(y, j)) {

non\_zero = true;

break;

}

}

if (!non\_zero) {

find = false;

break;

}

}

if (!find) {

change = true;

matrix.set(i, j, 0);

}

}

}

if (!change)

break;

}

}

}//namespace ullman

## Ullman\_utils.cpp

#include "ullman\_utils.h"

namespace ullman {

size\_t get\_parameters(struct parameters\_t \*p\_parameters, int argc, char \*argv[])

{

char \*database = NULL;

char \*query = NULL;

int sep\_type = 0;

size\_t n\_argh = 0;

for (size\_t i\_argv = 1; i\_argv < argc - 1; i\_argv += 2)

{

for (size\_t i\_argh = 0; i\_argh < N\_ARG; i\_argh++)

{

if (strcmp(argv[i\_argv], ARGH[i\_argh]) != 0)

{

continue;

}

switch (i\_argh)

{

case 0: database = argv[i\_argv + 1];

n\_argh++;

break;

case 1: query = argv[i\_argv + 1];

n\_argh++;

break;

case 2: sep\_type = atoi(argv[i\_argv + 1]);

if (sep\_type < 0 || sep\_type >= SEP\_TYPE\_NCOUNT) {

usage();

exit(-1);

}

break;

default:

usage();

exit(-1);

}

break;

}

}

p\_parameters->database = database;

p\_parameters->query = query;

p\_parameters->sep\_type = sep\_type;

return n\_argh;

}

void get\_seperator(const struct parameters\_t \*p\_parameters,

std::string& seperator)

{

if (p\_parameters->sep\_type == 0) {

//To-do: seperator factory

seperator = " ";

} else {

//do nothing

}

}

void print\_parameters(const struct parameters\_t \*p\_parameters)

{

printf("-database: %s\n", p\_parameters->database);

printf("-query: %s\n", p\_parameters->query);

printf("-sep: %d\n", p\_parameters->sep\_type);

}

size\_t get\_narg()

{

return N\_ARG;

}

void usage()

{

printf("usage!\n");

}

}//namespace ullman

## Ullman\_utils.h

#ifndef ULLMAN\_UTILS\_H

#define ULLMAN\_UTILS\_H

#include "common.h"

namespace ullman {

const static size\_t N\_ARG = 3;

const static char \*ARGH[N\_ARG] = { "-database", "-query", "-sep"};

struct parameters\_t {

char \*database;

char \*query;

int sep\_type;

};

enum SEP\_TYPE {

DEFAULT,

SEP\_TYPE\_NCOUNT

};

size\_t get\_parameters(struct parameters\_t \*p\_parameters, int argc, char \*argv[]);

void print\_parameters(const struct parameters\_t \*p\_parameters);

void get\_seperator(const struct parameters\_t \*p\_parameters, std::string& seperator);

size\_t get\_narg();

void usage();

}//namespace ullman

#endif

//此程序共1100行左右.