**算法分析与设计课程设计报告书 评分：\_\_\_\_\_\_\_\_\_\_\_\_**

**题目：基于Ullmann算法的图同构识别**

设计人：何树深

班级：13网络2班 学号：1314080903212

1. 实验环境和地点：

1、硬件环境：个人机，CPU主频：3.1GHz 内存：8G

2、软件环境：操作系统：ubuntu

编程语言：C++

3、地点：5号楼1楼智能信息处理实验室

1. 实验任务解决方案：

1、Ullmann算法的流程图。

开始

结束

输出结果

进行深度优先搜索

构造矩阵M

构造图B的邻接矩阵

构造图A的邻接矩阵

输入图B的数据

输入图A的数据

2、Ullmann算法实现的关键代码。

1). 构造矩阵:

A.

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

B.

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);

}

}

2).构造矩阵M的代码:

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;

}

1. . 深度优先搜索:

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

#endif

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

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

#ifdef DEBUG

#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;

}

}

1. Ullmann算法的计算复杂度分析：
2. . 在最坏情况下, Ullmann算法的时间复杂度是: O(n^4).
3. . 算法中, 剪枝是影响算法性能的重要因素.
4. . Ullmann算法以深度优先的方式进行搜索, 搜索过程表示为一个bool矩 阵. 当节点不匹配时, 回溯到最近匹配的节点, 寻找其他方向.
5. . 算法会检查匹配点对的邻接点的匹配情况, 尽早地识别出不匹配的节点, 提高算法效率.
6. 总结综合设计心得体会：

这个课程设计好难, 把别人的算法打印下来, 看了好久都不懂.

百度上资料很少, google上才有一些, 才查了好些资料才开始看懂一点点.