**综合设计报告**

**课程名称：**  数据结构

**题 目：**  图的建立和遍历

**指导教师：**  蔡英

**设计起始日期：** 2013.xx.xx-2013.xx.xx

**学 院：**  计算机学院

**系 别：** 计算机科学与技术系

**学生姓名：**  刘鸿喆

**班级/学号：**  计科1201/2012011107

**成 绩：**

**一、需求分析**

建立无向非连通图的邻接表存储结构，要求顶点个数不少于15个。

用DFS及BFS对此邻接表进行遍历，打印出两种遍历的顶点访问顺序。

给定图中任意两个顶点v1和v2及整数k，判断是否存在从v1到v2的路径长度为k的简单路径，若有打印出路径上的顶点序列（要求路径上不含回路）。进一步：找出从v1到v2的所有路径长度为k的简单路径。（简单路径：顶点序列中不含重现的顶点的路径。）

**二、概要设计**

**1、数据结构**

Map, List (Adjacency List), Queue, Stack, Vector

**2、使用算法**

BFS, DFS, 路径搜索（也是DFS）

**三、详细设计**

**1、数据结构详细设计**

Map：用于存储顶点名与顶点的键值对集合：std::map<Key, V> vs;

List：用于存储边：std::list<E> es;

Queue：用于BFS遍历图：std::queue<Key> ks;

Stack：用于记录和存储路径：std::stack<Key> \_path;

Vector：用于存储路径集合：std::vector<std::stack<Key>> \_\_paths;

**2、算法**

建立与删除：

加入点：同Map。

删除点：先删除其它点与其关联的边，再删除整个点与点上的边。

加入边：检测是否已连接，若已链接，更新权值，否则链接，在两个点上都加上边。

删除边：删除两点上对应边。

BFS：

1. 循环所有点，若点未访问过，以其作为开始点开始2~3过程，完成后返回1，继续循环
2. 将开始点加入队列中，访问并标记访问过
3. 若队列为空，结束，若不为空，指定当前点为队头点，重复进行下列步骤
   1. 访问当前点，对头出列
   2. 检测当前点所有临接的点，若未访问过，加入到队列中，并标为访问过

DFS：

1. 循环所有点，若点未访问过，以其作为开始点开始2~3过程，完成后返回1，继续循环
2. 访问节点（若初始则为开始点），并标记访问过
3. 检查邻接点，若为访问过，调用2~3过程，完成后返回3

路径搜索（也是DFS）：与上一个的不同之处在于记录路径和剩余长度，搜索失败后，将访问改为未访问状态。若剩余长度为0，且路径顶点对接，则成功找到一条路径，加入到Vector里。

**四、调试分析**

**1、调试过程中遇到的问题**

无。

**2、算法的时空分析**

整体存储空间：O(V+E)

建立点：

时间：O(1)

空间：O(1)

删除点：

时间：O(E)

空间：O(1)

建立边：

时间：O(E)

空间：O(1)

删除边：

时间：O(E)

空间：O(1)

BFS：

时间：O(V)

空间：O(V)

DFS：

时间：O(V)

空间：O(V)

路径搜索：

时间：O(V!)

空间：O(V)

**五、使用说明和测试结果**

**1、使用说明**

无。

**2、测试结果**

输出

A(192:168:0:1): P(8) C(4)

B(10:16:45:124): C(2) P(2)

C(211:123:23:12): B(2) A(4)

D(43:32:180:197): J(3) E(9) F(8)

E(234:222:123:1): G(4) D(9)

F(32:23:65:233): D(8)

G(12:168:173:90): E(4) H(2)

H(233:168:0:55): I(5) G(2)

I(128:168:164:64): H(5)

J(164:168:7:6): K(3) D(3)

K(3:168:11:56): J(3) O(7)

L(40:168:0:43): O(1) N(5)

M(45:168:23:136): O(9)

N(8:8:8:8): L(5)

O(12:168:45:23): K(7) L(1) M(9)

P(76:45:239:234): A(8) B(2)

DFS Result:

A: 192:168:0:1

P: 76:45:239:234

B: 10:16:45:124

C: 211:123:23:12

D: 43:32:180:197

J: 164:168:7:6

K: 3:168:11:56

O: 12:168:45:23

L: 40:168:0:43

N: 8:8:8:8

M: 45:168:23:136

E: 234:222:123:1

G: 12:168:173:90

H: 233:168:0:55

I: 128:168:164:64

F: 32:23:65:233

BFS Result:

A: 192:168:0:1

P: 76:45:239:234

C: 211:123:23:12

B: 10:16:45:124

D: 43:32:180:197

J: 164:168:7:6

E: 234:222:123:1

F: 32:23:65:233

K: 3:168:11:56

G: 12:168:173:90

O: 12:168:45:23

H: 233:168:0:55

L: 40:168:0:43

M: 45:168:23:136

I: 128:168:164:64

N: 8:8:8:8

Paths from I to M (length = 42):

I -> H -> G -> E -> D -> J -> K -> O -> M

Paths from A to P (length = 8):

A -> P

A -> C -> B -> P

A(192:168:0:1): C(4)

B(10:16:45:124): C(2) P(2)

C(211:123:23:12): B(2) A(4)

E(234:222:123:1): G(4)

F(32:23:65:233):

G(12:168:173:90): E(4) H(2)

H(233:168:0:55): I(5) G(2)

I(128:168:164:64): H(5)

J(164:168:7:6): K(3)

K(3:168:11:56): J(3) O(7)

L(40:168:0:43): O(1) N(5)

M(45:168:23:136): O(9)

N(8:8:8:8): L(5)

O(12:168:45:23): K(7) L(1) M(9)

P(76:45:239:234): B(2)

**六、心得体会**

图最重要的就是结构，好的结构编码也会简单。

**七、附录**

**1、程序**

//

// main.cpp

// ds-ex4-graph

//

// Created by Darren Liu on 13-12-11.

// Copyright (c) 2013年 mashiroLab. All rights reserved.

//

#include <iostream>

#include <string>

#include "include/ipv4-address.h"

#include "include/graph.h"

void print(const char &vertex, IPv4Address &ip) {

std::cout << vertex << ": " << ip << std::endl;

}

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

UDG<char, IPv4Address> graph;

graph['A'] = IPv4Address(192, 168, 0, 1);

graph['B'] = IPv4Address(10, 16, 45, 124);

graph['C'] = IPv4Address(211, 123, 23, 12);

graph['D'] = IPv4Address(43, 32, 180, 197);

graph['E'] = IPv4Address(234, 222, 123, 1);

graph['F'] = IPv4Address(32, 23, 65, 233);

graph['G'] = IPv4Address(12, 168, 173, 90);

graph['H'] = IPv4Address(233, 168, 0, 55);

graph['I'] = IPv4Address(128, 168, 164, 64);

graph['J'] = IPv4Address(164, 168, 7, 6);

graph['K'] = IPv4Address(3, 168, 11, 56);

graph['L'] = IPv4Address(40, 168, 0, 43);

graph['M'] = IPv4Address(45, 168, 23, 136);

graph['N'] = IPv4Address(8, 8, 8, 8);

graph['O'] = IPv4Address(12, 168, 45, 23);

graph['P'] = IPv4Address(76, 45, 239, 234);

graph.link('C', 'B', 2);

graph.link('K', 'J', 3);

graph.link('O', 'K', 7);

graph.link('J', 'D', 3);

graph.link('H', 'I', 5);

graph.link('P', 'A', 8);

graph.link('L', 'O', 1);

graph.link('G', 'E', 4);

graph.link('G', 'H', 2);

graph.link('O', 'M', 9);

graph.link('D', 'E', 9);

graph.link('C', 'A', 4);

graph.link('B', 'P', 2);

graph.link('D', 'F', 8);

graph.link('L', 'N', 5);

graph.showAdjList();

std::cout << std::endl;

std::cout << "DFS Result:" << std::endl;

graph.dfs(print);

std::cout << std::endl;

std::cout << "BFS Result:" << std::endl;

graph.bfs(print);

std::vector<std::stack<char>> paths = graph.paths('I', 'M', 42);

std::cout << std::endl;

std::cout << "Paths from I to M (length = 42): " << std::endl;

for (std::stack<char> path : paths) {

if (!path.empty()) {

std::cout << path.top();

path.pop();

}

while (!path.empty()) {

std::cout << " -> " << path.top();

path.pop();

}

std::cout << std::endl;

}

paths = graph.paths('A', 'P', 8);

std::cout << std::endl;

std::cout << "Paths from A to P (length = 8): " << std::endl;

for (std::stack<char> path : paths) {

if (!path.empty()) {

std::cout << path.top();

path.pop();

}

while (!path.empty()) {

std::cout << " -> " << path.top();

path.pop();

}

std::cout << std::endl;

}

std::cout << std::endl;

graph.erase('D');

graph.unlink('A', 'P');

graph.showAdjList();

return 0;

}

#include <iostream>

#include <list>

#include <map>

#include <vector>

#include <queue>

#include <stack>

#include <algorithm>

template <typename Key, typename Value>

class UDG {

private:

class E {

public:

E() {}

E(const Key &k, const unsigned int &w = 1) : k(k), w(w) {}

Key k;

int w;

};

class V {

public:

V() {}

V(const Value &value) : value(value) {}

Value value;

std::list<E> es;

bool visited;

};

std::map<Key, V> vs;

public:

UDG() {}

~UDG() {}

Value &operator[](Key &&k) {

return vs[k].value;

}

std::size\_t erase(const Key &k) {

for (const auto &e : vs[k].es) {

vs[e.k].es.remove\_if([k](const E &e) {

return e.k == k;

});

}

return vs.erase(k);

}

bool link(const Key &k1, const Key &k2, const unsigned int &w) {

if (w == 0) {

return false;

}

typename std::map<Key, V>::iterator v1 = vs.find(k1);

typename std::map<Key, V>::iterator v2 = vs.find(k2);

if (v1 == vs.end() || v2 == vs.end()) {

return false;

}

typename std::list<E>::iterator e1 =

find\_if(v1 -> second.es.begin(),

v1 -> second.es.end(),

[k2](const E & e) {

return e.k == k2;

});

typename std::list<E>::iterator e2 =

find\_if(v2 -> second.es.begin(),

v2 -> second.es.end(),

[k1](const E & e) {

return e.k == k1;

});

if (e1 != v1 -> second.es.end() && e2 != v2 -> second.es.end()) {

e1 -> w = w;

e2 -> w = w;

} else {

v1 -> second.es.push\_back(E(k2, w));

v2 -> second.es.push\_back(E(k1, w));

}

return true;

}

bool unlink(const Key &k1, const Key &k2) {

typename std::map<Key, V>::iterator v1 = vs.find(k1);

typename std::map<Key, V>::iterator v2 = vs.find(k2);

if (v1 == vs.end() || v2 == vs.end()) {

return false;

}

v1 -> second.es.remove\_if([k2](const E & e) {

return e.k == std::move(k2);

});

v2 -> second.es.remove\_if([k1](const E & e) {

return e.k == std::move(k1);

});

return true;

}

private:

std::stack<Key> \_path;

std::vector<std::stack<Key>> \_\_paths;

public:

std::vector<std::stack<Key>> paths(const Key &k1, const Key &k2, int l) {

while (!\_path.empty()) {

\_path.pop();

}

\_\_paths.clear();

for (auto &v : vs) {

v.second.visited = false;

}

\_paths(k1, k2, l);

return \_\_paths;

}

private:

void \_paths(const Key &k1, const Key &k2, int l) {

if (l < 0) {

return;

}

if (l == 0 && k1 == k2) {

\_path.push(k1);

\_\_paths.push\_back(\_path);

\_path.pop();

return;

}

\_path.push(k2);

vs[k2].visited = true;

for (const auto &e : vs[k2].es) {

if (!vs[e.k].visited) {

\_paths(k1, e.k, l - e.w);

}

}

vs[k2].visited = false;

\_path.pop();

}

public:

void bfs(void visit(const Key &, Value &)) {

if (vs.size() == 0) {

return;

}

std::queue<Key> ks;

for (auto &v : vs) {

v.second.visited = false;

}

for (auto &v : vs) {

if (!v.second.visited) {

ks.push(v.first);

v.second.visited = true;

}

while (!ks.empty()) {

Key k = ks.front();

ks.pop();

visit(k, vs[k].value);

for (const auto &e : vs[k].es) {

if (!vs[e.k].visited) {

ks.push(e.k);

vs[e.k].visited = true;

}

}

}

}

}

public:

void dfs(void visit(const Key &, Value &)) {

for (auto &v : vs) {

v.second.visited = false;

}

for (const auto &v : vs) {

dfs(visit, v.first);

}

}

private:

void dfs(void visit(const Key &, Value &), Key k) {

if (vs[k].visited) {

return;

}

vs[k].visited = true;

visit(k, vs[k].value);

for (const auto &e : vs[k].es) {

dfs(visit, e.k);

}

}

public:

void showAdjList() {

for (const auto &v : vs) {

std::cout << v.first << '(' << v.second.value << ')' << ":";

for (const auto &e : v.second.es) {

std::cout << ' ' << e.k << '(' << e.w << ')';

}

std::cout << std::endl;

}

}

};