哈密顿路径实验报告

杜金鸿,15338039

2017 年 5 月 22 日

目录

1 实验目的

1.1 问题描述

哈密顿路径是指图 G 中一条包含所有顶点的简单路径，即哈密顿路径是一条连接 n 个结点、长度为 $n-1$ 的路径。

1.2 实验要求

1. 判断非完全有向图 G 是否存在哈密顿路径；
2. 若存在哈密顿路径，则输出；
3. 分析算法时间复杂度.

2 实验内容

2.1 哈密顿路径问题的存储结构

用数组记录结点类型，并采用邻接矩阵结构存储图中结点之间的关系。

2.2 哈密顿路径求解算法

采用取消回溯的深度优先遍历算法求解哈密顿路径。

若从某个结点出发进行遍历可以找到哈密顿路，则考虑哈密顿路径的搜寻过程。利用深度优先的遍历算法从某结点 v 出发搜寻到某一结点时，则从该结点 v' 出发对剩余结点进行遍历，若部分图遍历完成时，已判断部分图中存在从 v' 出发的哈密顿路，则原图存在哈密顿路，且该路已经可以确定；若部分图遍历完成时，已判断该部分图不存在从 v' 点出发的哈密顿路径，则返回上一步，排除结点 v'，将前一步中下一个未遍历结点出发，继续查找。

停止查找的条件是，遍历结果数组中已经有 n 个结点。同样地，若整个图最终查找过程中，遍历数组中结点数恰好达到过 n，则说明改图存在从 v 出发的哈密顿路径且就是当前路径。反之，若遍历结果数组结点个数未达到过 n 个，则不存在从该结点出发的哈密顿路径。

2.3 分析算法时间复杂度

3 设计与编码

3.1 存储结构

在 "*graph.hpp*" 文件中定义图结构。

```
#ifndef graph_hpp
#define graph_hpp

#include <stdio.h>
```

```
#include <iostream>
using namespace std;
const int Maxsize=10;

template<class T>
class Graph{
public:
    Graph(T a[], int n, int e); // 构造函数1
    Graph(T a[], int n, int g[]); // 构造函数2
    ~Graph(); // 析构函数
    void DFS(int v); // 取消回溯的深度优先遍历算法
    void DFS_find(); // 遍历各初始结点
    void show_graph(); // 输出图
    void show(); // 输出
private:
    T vertex[Maxsize]; // 结点数组
    int arc[Maxsize][Maxsize]; // 邻接矩阵
    int vertexNum,arcNum; // 结点数、边数
    int* visited; // 遍历标志数组
    int* S; // 遍历结果数组
    int count; // 计数器
    bool flag; // 哈密顿路存在标志
};

#endif /* graph_hpp */
```

3.2 算法流程

Algorithm 1 取消回溯的深度优先遍历算法

各参数初始化
输入遍历的初始结点 v, 执行函数 DFS(v){
$visited[v] = 1; S[count++] = v;$
for $0 \leqslant j < vertexNum$ do
  if $v != j$ && $arc[v][j] == 1$ && $visited[j] == 0$ then
    执行函数 DFS(j)
  end if
end for
if $count == vertexNum$ then
  return;
else
  if $j == vertexNum$ then
    $visited[v] = 0; count--;$
  end if
end if
}

3.3 类的实现

在”graph.cpp” 文件完成对类的实现：

```
#include ”graph.hpp”
#include <iostream>
using namespace std;

template<class T>
Graph<T>::Graph(T a[], int n, int e){
    vertexNum = n;arcNum = e;count = 0;flag = false;
    visited = new int[vertexNum]();
    S = new int[vertexNum]();
    for (int i=0; i<vertexNum; i++) {
        vertex[i]=a[i];
    }
    for (int i=0; i<vertexNum; i++) {
        for (int j=0;j<vertexNum; j++) {
            arc[i][j] = 0;
        }
    }
    for (int i,j,k = 0; k<arcNum; k++) {
        cout<<”请输入边的两个顶点的序号：”;
        cin>>i>>j;
        arc[i][j] = 1;arc[j][i] = 1;
    }
}

template<class T>
Graph<T>::Graph(T a[], int n, int* g){
    vertexNum = n;count = 0; arcNum = 0;
    visited = new int[vertexNum];
    S = new int[vertexNum];
    for (int i=0; i<vertexNum; i++) {
        vertex[i]=a[i];
        visited[i] = 0;
    }
    for (int i=0; i<vertexNum; i++) {
        for (int j=0;j<vertexNum; j++) {
            arcNum++;
            arc[i][j]=g[i*vertexNum+j];
        }
    }
}

template<class T>
Graph<T>::~Graph(){
    delete[] visited;
    delete[] S;
}

template<class T>
void Graph<T>::DFS(int v){ //指定初始节点，取消回溯的深度优先遍历
    visited[v] = 1; S[count++]=v; int j=0;
    for (j = 0; j<vertexNum; j++) {
        if (v!=j && arc[v][j] == 1 && visited[j] == 0) {
```

```
            DFS(j);
        }
    }
    if (count==vertexNum) {
        flag = true;
        return;
    }
    else if (j == vertexNum) {
        visited[v] = 0; count--;
    }

}

template<class T>
void Graph<T>::DFS_find(){ // 遍历各初始节点
    for (int i=0; i<vertexNum; i++) {
        flag = false;
        count = 0;
        for (int j=0; j<vertexNum; j++) {
            visited[j]=0;
        }
        DFS(i);
        show();
    }
}

template<class T>
void Graph<T>::show_graph(){
    cout<<"图： "<<endl<<'\t';
    for (int k=0; k<vertexNum; k++) {
        cout<<vertex[k]<<'\t';
    }
    cout<<endl;
    for (int i=0; i<vertexNum; i++) {
        cout<<vertex[i]<<'\t';
        for (int j=0; j<vertexNum; j++) {
            cout<<arc[i][j]<<'\t';
        }
        cout<<endl;
    }
}

template<class T>
void Graph<T>::show(){
    if (flag) {
        cout<<"汉密顿路： ";
        for (int i=0; i<vertexNum; i++) {
            cout<<vertex[S[i]];
            if (i!=vertexNum-1) {
                cout<<"-->";
            }
        }
        cout<<endl;
    }
    else
```

```
        cout<<"不存在从"<<vertex[S[0]]<<"出发的哈密顿路"<<endl;
}
```

4 运行与测试

4.1 运行结果

创建一个含有 6 个结点的非完全有向图，结点为：

$$vertex = \{'A','B','C','D','E','F'\}$$

邻接矩阵为：

$$arc = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

主函数如下：

```
#include <iostream>
#include "graph.cpp"

int main(int argc, const char * argv[]) {
    char ch[]={'A','B','C','D','E','F'};
    int g[6][6]={{1,0,1,0,1,0},
        {1,1,0,1,0,0},
        {1,1,1,0,0,0},
        {0,1,0,1,0,1},
        {1,0,0,0,1,1},
        {0,0,0,0,0,1}};
    Graph<char> a(ch,6,g[0]);
    a.show_graph();
    a.DFS_find();

    return 0;
}
```

输出结果：

```
图:
    A   B   C   D   E   F
A   1   0   1   0   1   0
B   1   1   0   1   0   0
C   1   1   1   0   0   0
D   0   1   0   1   0   1
E   1   0   0   0   1   1
F   0   0   0   0   0   1
不存在从A出发的哈密顿路
不存在从B出发的哈密顿路
不存在从C出发的哈密顿路
不存在从D出发的哈密顿路
汉密顿路: E-->A-->C-->B-->D-->F
不存在从F出发的哈密顿路
Program ended with exit code: 0
```

4.2 算法分析

从某个结点出发寻找哈密顿路径，最佳情况的时间复杂度为 n，最坏情况的时间复杂度为 $n!$（有向非完全图）。

由于算法寻找哈密顿路径需要遍历各结点，判断是否存在从每个结点出发的哈密顿路径。所以算法在最佳情况下的时间复杂度是 n^2，在最坏情况的时间复杂度为 $n \cdot n!$。

4.3 算法扩展

若将上述取消回溯的深度优先遍历算法应用到哈密顿回路求解问题中，则只需在确定每个结点出发的哈密顿路后，判断是否存在 $S[verNum-1]$ 到 $S[0]$ 的路径。若有便存在哈密顿回路。当然这种算法在时间复杂度方面的表现并不优秀。

5 总结

哈密顿路径求解问题，需要对图的结点进行遍历，显然利用深度优先遍历算法比广度优先遍历算法更合理。查找图的哈密顿路问题可以分解成查找各子图哈密顿路径这一子问题。使用深度优先遍历算法进行遍历，我们需要在查找子图哈密顿路径失败时回溯并且判断遍历完成的标志。