哈密顿路径

班级：17数据科学与大数据技术1班  
  
黄希瑞（20170060123）

哈密顿路径

**17数据科学与大数据技术1班　　黄希瑞（20170060123）**

# ****需求分析****

本人计划编写一个图论计算程序，主要用来计算无权图的哈密顿路径。具体要求如下：

1. 创建无权图的邻接矩阵：能建立无权无向图和有向图，自行设计一个至少包含8个顶点（不超过10个）的无权图，从键盘或文件输入顶点个数、边的个数、边的2个顶点序号；
2. 显示：显示有向无权图（或无向无权图）的邻接矩阵；
3. 哈密顿路径：在图G中找出一条包含所有顶点的简单路径，该路径称为哈密顿路径。利用已经建好的图，寻找哈密顿路径，分析时间复杂度并输出结果；
4. 所有算法均报告成功与否，所有算法的输入、输出都保存并可复查。

# 概要设计

# 数据结构

本程序以图为基础建立1个类，其中包含图的方向性、邻接矩阵、顶点数量、邻边数量、节点访问记录、遍历路径共6个成员变量。邻接矩阵采用二维数组结构，模拟矩阵的行列定位。类定义如下：

|  |
| --- |
| #pragma once  #include <iostream>  #include <fstream>  #include <stack>  using namespace std;  // 最大顶点数量  constexpr int MAX\_VERTEX = 9;  class Graph {  private:  // 图的类型  bool isDirectional;  // 邻接矩阵  bool adjacency[10][10] = { 0 };  // 顶点数量  int vertexSize;  // 邻边数量  int edgeSize;  // 节点访问记录  bool isVisited[10] = { 0 };  // 深度优先遍历（内核）  void \_dft(const int);  // 重置节点访问记录（内核）  void \_rv();  // 遍历路径  stack<int> pathStack;  // 深度优先搜索（内核）  void \_dfs(const int);  public:  // 度的类型  const enum DegreeType {  in = 1, out = 2, all = 3  };  // 创建邻接矩阵  Graph(const bool, const int, const int);  // 析构函数  ~Graph() {}  // 显示邻接矩阵  void showGraph() const;  // 计算度数、入度、出度  int getDegree(int, DegreeType) const;  // 深度优先遍历（公开）  void deepFirstTraverse(const int);  // 哈密顿路径  void hamitonPath();  // 另存为  void saveAs(const string);  // 打开  Graph(const string);  }; |

# 函数主要调用关系图

本系统8个功能之间的主要调用关系如【图2-1】所示：



图2-1 系统函数调用关系图

# 详细设计

# “从键入生成无权图”功能包括如下函数：

|  |
| --- |
| // 创建邻接矩阵  Graph::Graph(const bool \_isDirectional, const int \_vertexSize, const int \_edgeSize)  {  isDirectional = \_isDirectional;  vertexSize = \_vertexSize;  edgeSize = \_edgeSize;  for (int i = 0; i < \_edgeSize; i++)  {  cout << "请输入邻边的2个顶点编号：" << ends;  // 起点与终点编号  int departure, arrival;  cin >> departure >> arrival;  adjacency[departure - 1][arrival - 1] = true;  // 若是无向图，需要将有向邻接矩阵进行对称  if (!\_isDirectional)  adjacency[arrival - 1][departure - 1] = true;  }  } |

# “从键入生成无权图”功能设计算法流程图如【图2-2】所示：



图3-1　从键入生成无权图

# “显示邻接矩阵”功能包括如下函数：

|  |
| --- |
| // 显示邻接矩阵  void Graph::showGraph() const  {  cout << "邻接矩阵为：" << endl;  for (int i = 0; i < vertexSize; i++)  {  for (int j = 0; j < vertexSize; j++)  cout << adjacency[i][j] << ends;  cout << endl;  }  } |

# “显示邻接矩阵”功能设计流程图如【图3-2】所示：



图3-2　显示邻接矩阵

# “计算度数、入度、出度”功能包括如下函数：

|  |
| --- |
| // 计算度数、入度、出度  int Graph::getDegree(int \_vertexID, DegreeType \_type = all) const  {  int degree = 0;  switch (\_type)  {  // 出度  case out:  for (const bool &edge : adjacency[\_vertexID - 1])  // 遍历特定行中的每一项  degree += edge;  break;  // 入度  case in:  for (const bool(&edgeList)[10] : adjacency)  // 遍历每一行中的特定项  degree += edgeList[\_vertexID - 1];  break;  // 总度  case all:  // 有向图：入度＋出度  degree = getDegree(\_vertexID, in) + getDegree(\_vertexID, out);  if (!isDirectional)  // 无向图：有向图÷２  degree = degree / 2;  break;  }  return degree;  } |

# “计算度数、入度、出度”功能设计流程图如【图3-3】所示：



图3-5　计算度数、入度、出度

# “深度优先遍历”功能包括如下函数：

|  |
| --- |
| // 深度优先遍历（公开）  void Graph::deepFirstTraverse(const int \_vertexID)  {  \_dft(\_vertexID);  \_rv();  } |
| // 深度优先遍历（内核）  void Graph::\_dft(const int \_vertexID)  {  // 输出当前顶点  cout << \_vertexID << ends;  // 将当前顶点标记为已访问  isVisited[\_vertexID - 1] = true;  // 遍历所有顶点  for (int i = 1; i <= vertexSize; i++)  // 若存在连通的边，且另一端的顶点未被访问  if (adjacency[\_vertexID - 1][i - 1] && !isVisited[i - 1])  // 递归地进行深度优先遍历  \_dft(i);  } |
| // 重置节点访问记录  void Graph::\_rv()  {  for (bool &vertex : isVisited)  vertex = false;  } |

# “深度优先遍历”功能设计流程图如【图3-4】～【图3-6】所示：



图3-4　深度优先遍历（公开）



图3-5　深度优先遍历（递归内核）



图3-6　重置节点访问记录

# “文件另存为”功能包括如下函数：

|  |
| --- |
| // 另存为  void Graph::saveAs(const string \_fileName)  {  ofstream fileWriter;  // 以输出方式打开文件  fileWriter.open(\_fileName, ios::out);  //若文件打开成功  if (fileWriter.is\_open())  {  // 导出方向性  fileWriter << isDirectional << endl;  // 导出顶点数量  fileWriter << vertexSize << endl;  // 导出邻边数量  fileWriter << edgeSize << endl;  // 导出顶点连接表  for (int i = 0; i < vertexSize; i++)  // 对于有向图  if (isDirectional)  {  // 遍历所有顶点（可能有反向边）  for (int j = 0; j < vertexSize; j++)  if (adjacency[i][j])  fileWriter << i + 1 << ends << j + 1 << endl;  }  else  {  // 跳过反向边（可能导致重复输出）  for (int j = i; j < vertexSize; j++)  if (adjacency[i][j])  fileWriter << i + 1 << ends << j + 1 << endl;  }  cout << "文件" << \_fileName << "导出成功！" << endl;  }  else  cerr << "错误：导出异常！" << endl;  fileWriter.close();  } |

# “文件另存为”功能设计流程图如【图3-7】所示：



图3-7　文件另存为

# “文件打开”功能包括如下函数：

|  |
| --- |
| // 打开  Graph::Graph(const string \_fileName)  {  ifstream fileReader;  // 以输入方式打开文件  fileReader.open(\_fileName, ios::in);  if (fileReader.is\_open())  {  // 读取方向  fileReader >> isDirectional;  // 读取顶点数量  fileReader >> vertexSize;  // 读取邻边数量  fileReader >> edgeSize;  for (int i = 0; i < edgeSize; i++)  {  int departure, arrival;  fileReader >> departure >> arrival;  adjacency[departure - 1][arrival - 1] = true;  // 若是无向图，需要将有向邻接矩阵进行对称  if (!isDirectional)  adjacency[arrival - 1][departure - 1] = true;  }  cout << "文件" << \_fileName << "导入成功！" << endl;  }  else  cerr << "错误：导入异常！" << endl;  } |

# “文件打开”功能设计流程图如【图3-8】所示：



图3-8　文件打开

# “哈密顿路径”功能包括如下函数：

|  |
| --- |
| // 深度优先搜索（内核）  void Graph::\_dfs(const int \_vertexID)  {  // 将当前顶点加入路径栈  pathStack.push(\_vertexID);  // 标记当前顶点访问状态  isVisited[\_vertexID - 1] = true;  for (int i = 1; i <= vertexSize; i++)  {  if (adjacency[\_vertexID - 1][i - 1] && !isVisited[i - 1])  {  \_dfs(i);  }  // 存在哈密顿路径，退出循环  if (pathStack.size() == vertexSize)  {  // 留存路径并退出（后续再还原）  return;  }  }  // 将当前顶点撤出路径栈  pathStack.pop();  // 撤销当前顶点访问状态  isVisited[\_vertexID - 1] = false;  } |
| // 哈密顿路径  void Graph::hamitonPath()  {  // 从每个顶点开始  for (int i = 1; i <= vertexSize; i++)  {  // 深度优先搜索  \_dfs(i);  // 当路径栈满（一条路径遍历所有顶点）  if (pathStack.size() == vertexSize)  {  // 已找到路径，退出循环  break;  }  }  for (long long i = pathStack.size() - 1; i >= 0; i--)  {  cout << pathStack.\_Get\_container()[i] << ends;  }  // 重置路径栈  while (!pathStack.empty())  {  pathStack.pop();  }  // 重置顶点访问记录  \_rv();  } |

# “哈密顿路径”功能设计流程图如【图3-9】所示：



图3-9　哈密顿路径

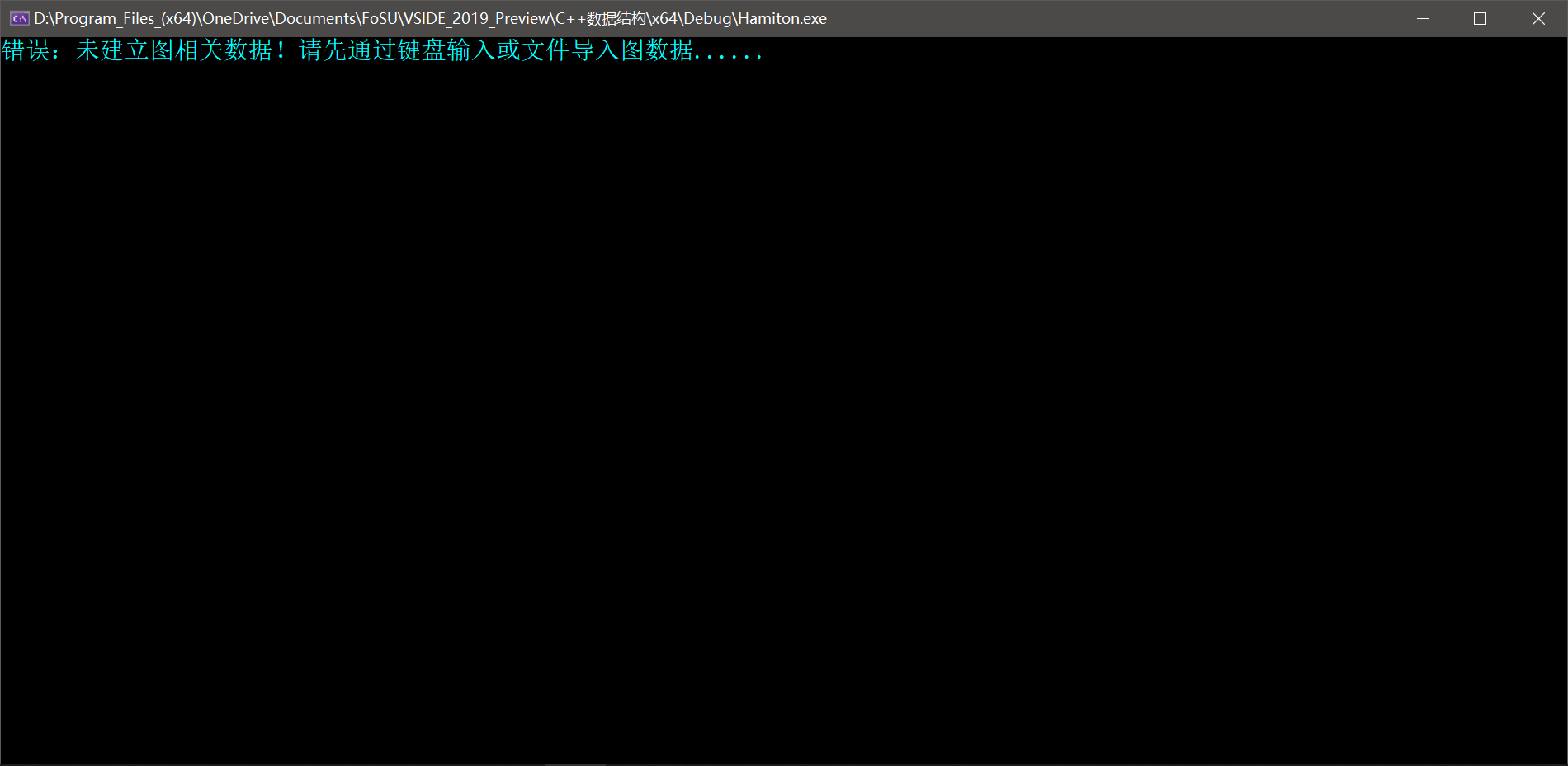
# 实验结果分析

# 问题与解决

# 启动时的非法选项屏蔽

在程序启动进入主界面时，此时程序中尚未为图建立对象，除了“从键入生成无权图”、“从文件读取”（对应2个构造函数）和“退出”3个功能选项能够正常使用，选择其余任意功能都将导致程序崩溃。

主函数启动程序时，图对象指针graph即被创建并赋值为std::nullptr（C++新标准中的空指针），可以通过判断graph对象是否为空指针，对功能菜单实现更深层次的管理控制。



# 文件导入错误时的对象变量重置

文件导入是通过对象的构造函数实现的，不论“图纸文件”是否被正常导入，图对象指针都一定不再是空指针。

graph非空，图相关的计算功能（功能2～6）将被启用，但由于图仍然存在异常，所以进行的运算也还是错误的。

在Graph类中添加公共数据成员isCorrectlyInited: bool，标记图是否被正常创建，如果图没有被正确创建（文件导入失败），此时关闭对象并重新将指针置空（相当于对象创建失败），继续让菜单对相关计算功能实现屏蔽。

# 特色及改进思想

本程序体积小、功能完善，且对于运行时错误考虑较为周全。通过指针动态调整程序占用的空间，避免运行时内存体积过大。

# 经验和体会

通过本次课程设计，我理解到了开发大型应用程序项目的繁杂与困难。大型应用程序的代码量普遍超过500行，有的甚至超过1000行。多文件引用层次、运行时异常处理需要考虑的东西非常多，部分异常与错误必须通过特殊数据的容错测试才能发现。

避免程序异常的方式是在开发时形成清晰的程序开发框架，明确程序在各个状态时需要执行的任务，以面向过程的思维理解面向对象程序的每一个细节，这样才能在开发时做到精益求精、游刃有余。

# 用户手册

编译并运行项目，进入主菜单，如【图5-1】所示，选择功能编号则直接跳转到相应功能，非法按键将被直接屏蔽，程序不会对非法输入做出响应。

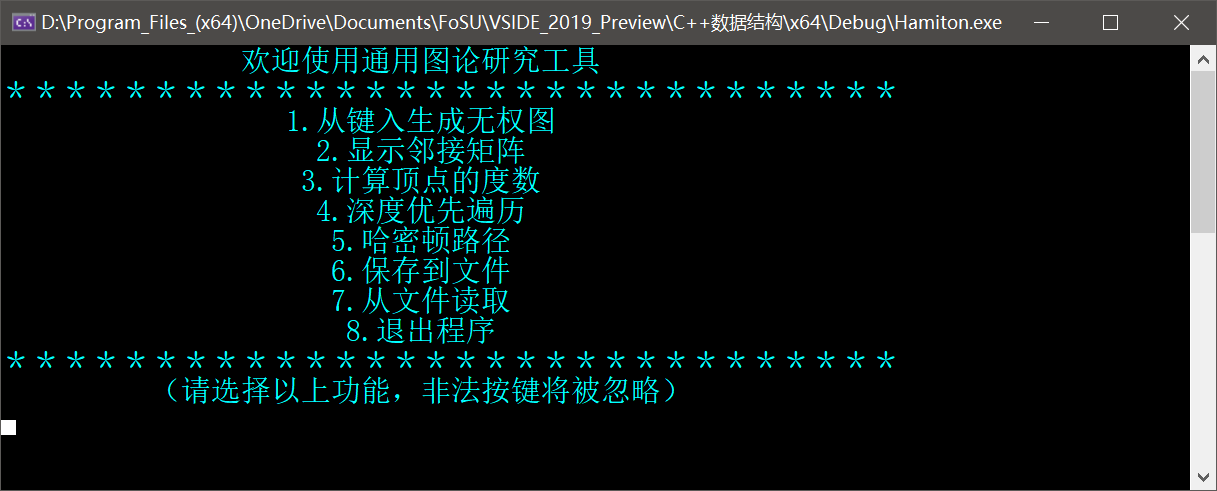


图5-1

选择功能2～6，此时数据未创建，出现【图5-2】所示的错误信息，延时3秒自动返回主菜单。

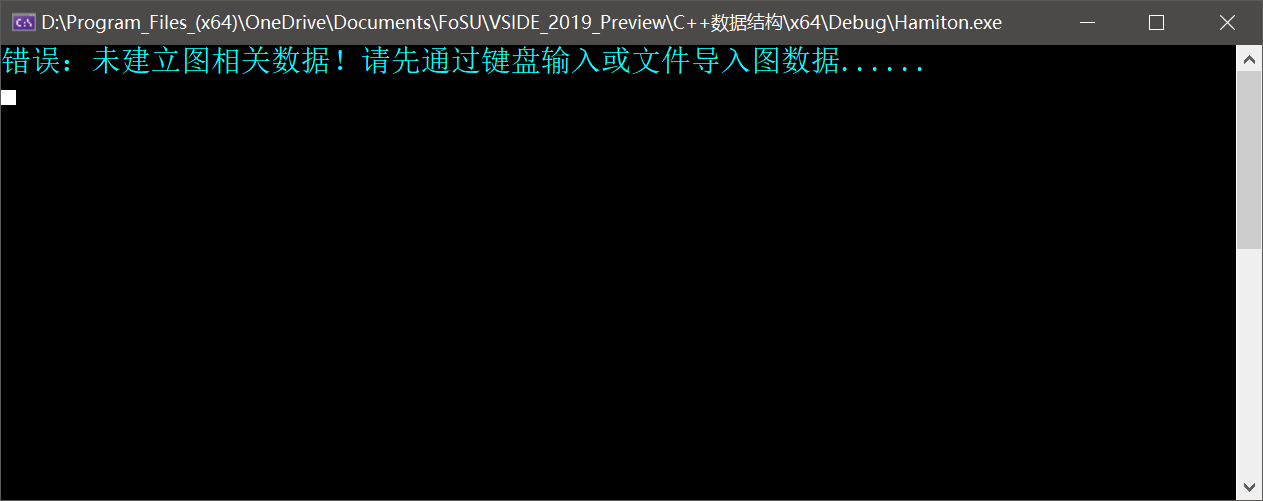


图5-2　图异常信息

选择功能7，从文件导入文件（建议使用绝对路径，相对路径是相对于项目根目录而言），如果文件异常，则输出异常信息，如【图5-3】。

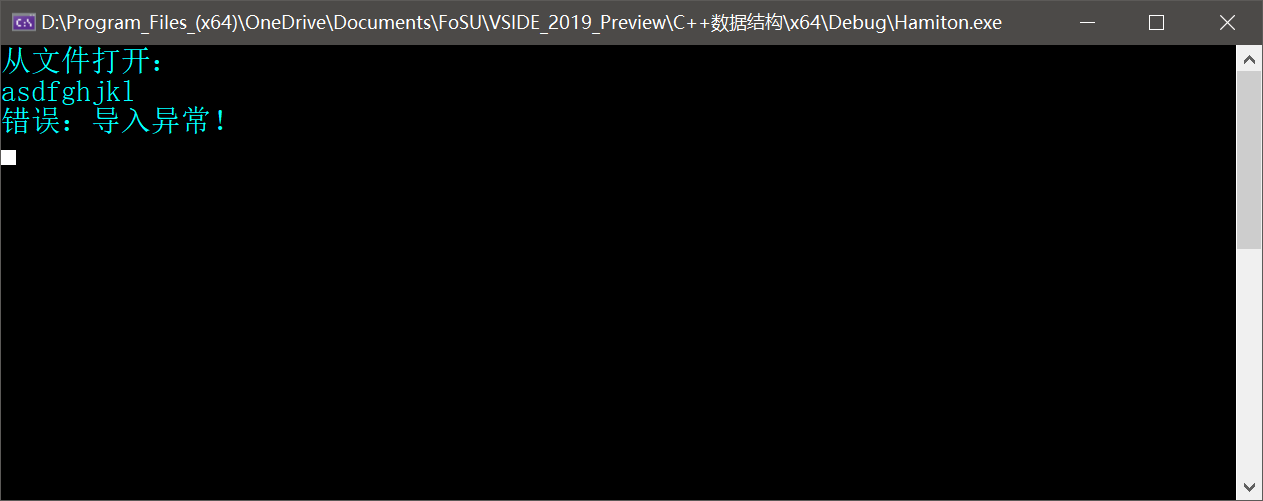
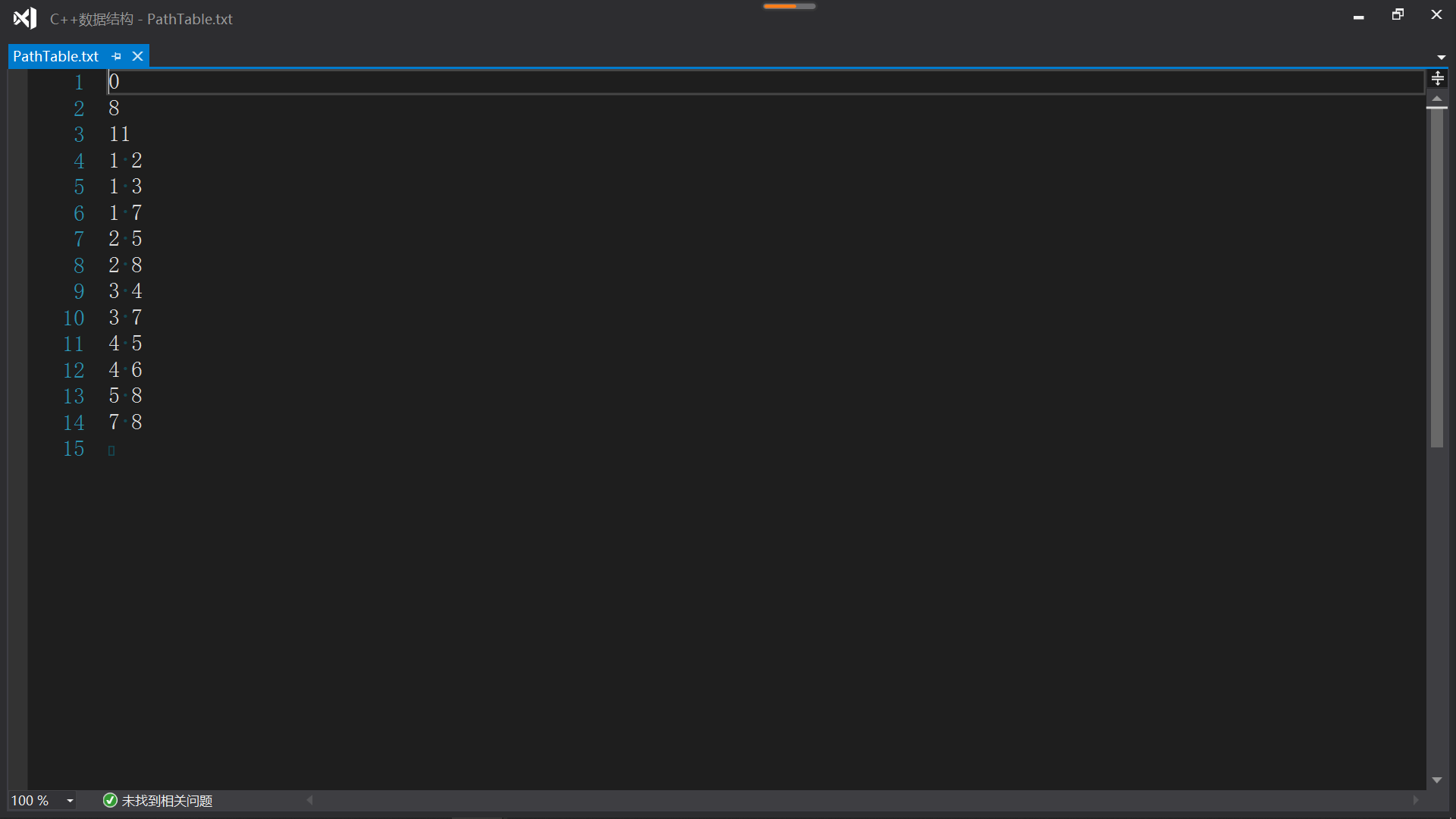


图5-3　文件打开异常

打开一个正常的文件（已经预置数据，如【图5-4】），如果文件打开全部正常，则成功导入，如【图5-5】。



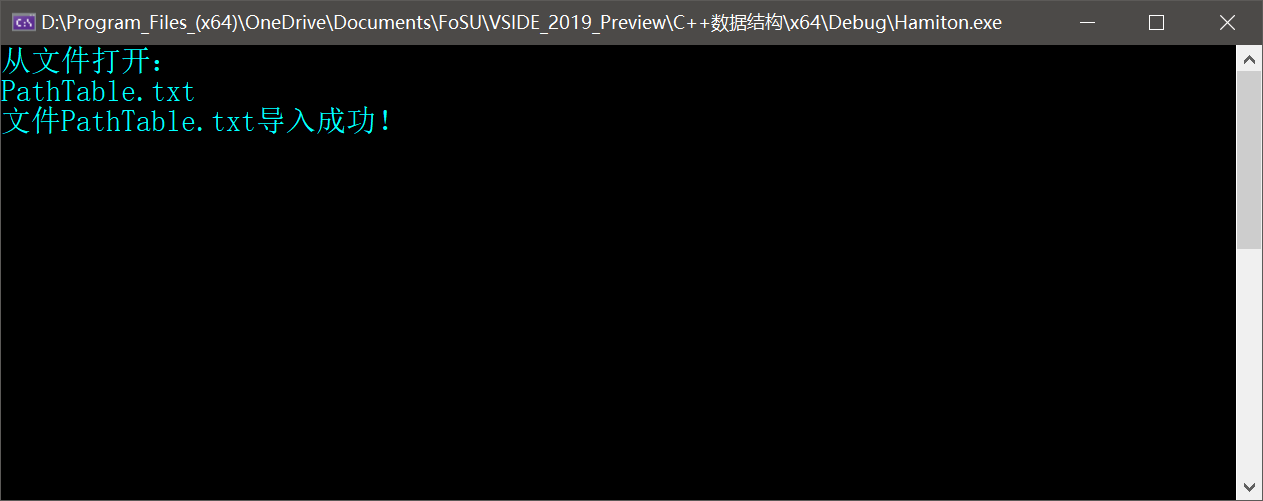


图5-5　文件导入成功

现在，重新选择功能2，展示邻接矩阵，如【图5-6】。

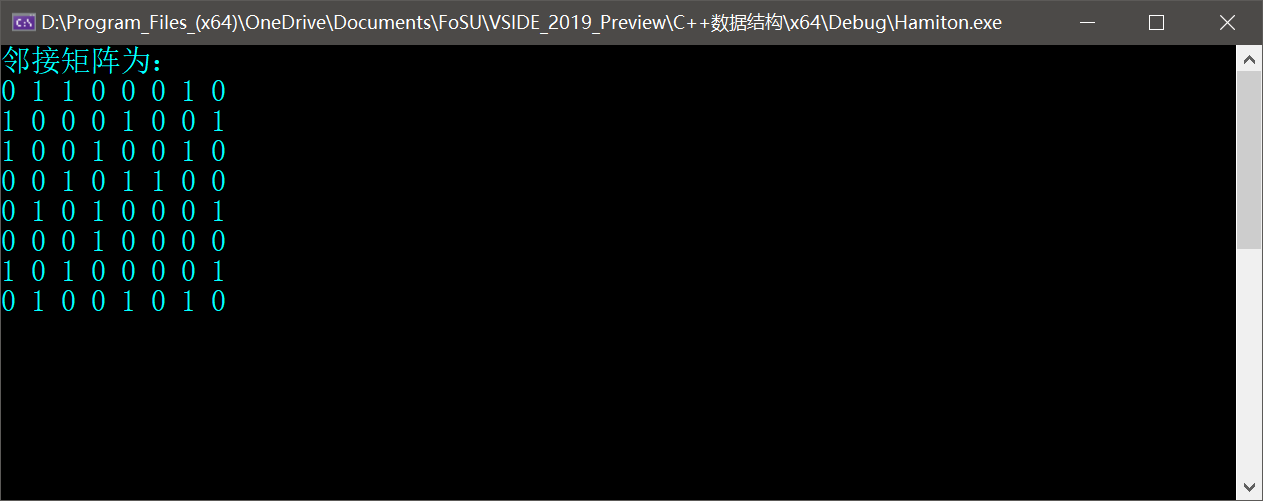


图5-6　邻接矩阵

选择功能3，输入定点编号并进入菜单，如【图5-7】。

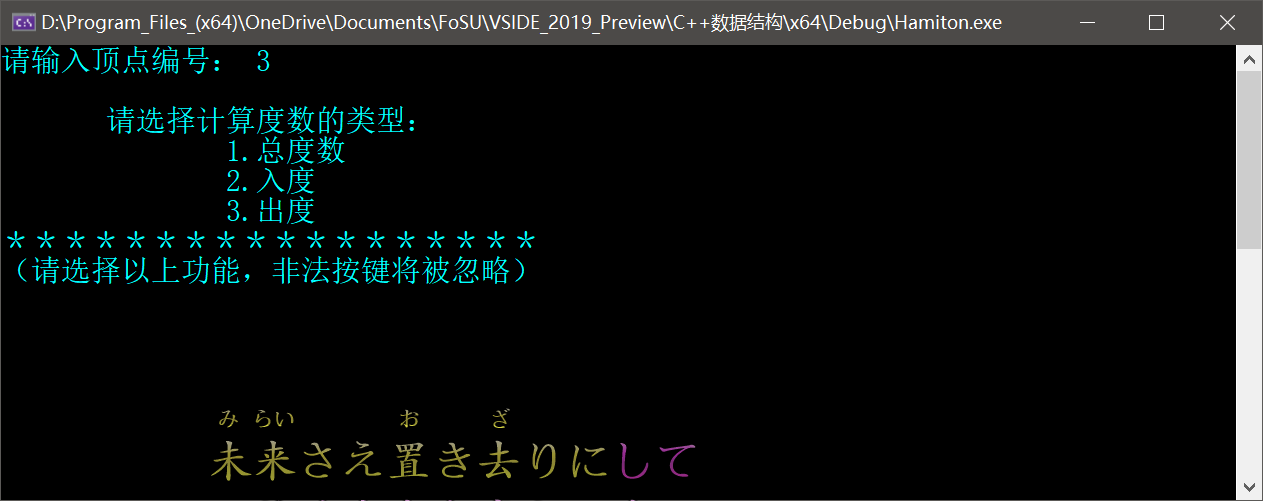


图5-7　计算度数

由于导入的是无向无权图，所以3种类型计算的数值应该都相同，如【图5-8】～【图5-10】，延迟3秒后返回顶层菜单。

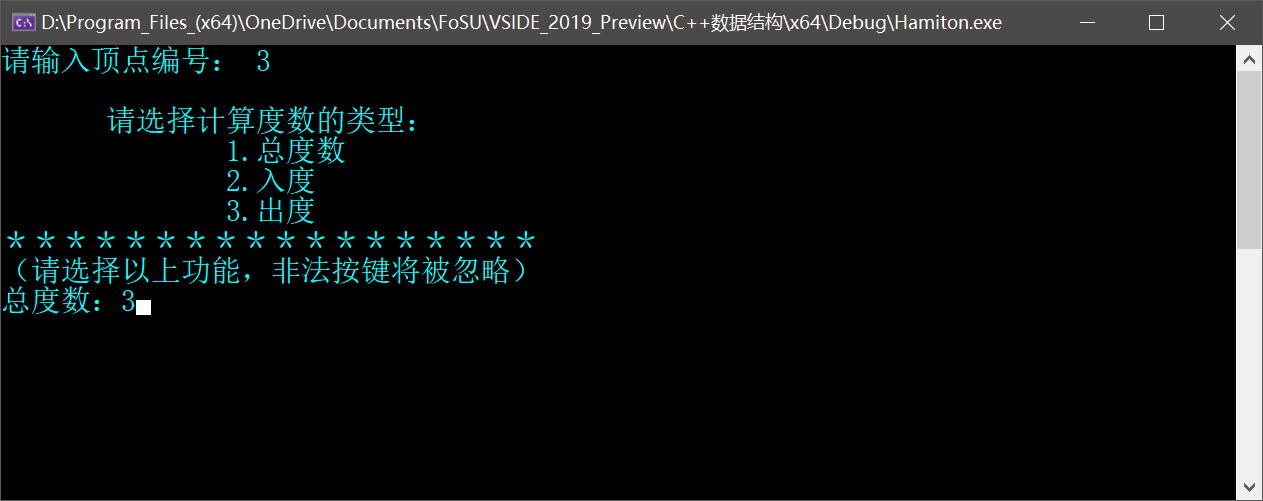


图5-8　总度数

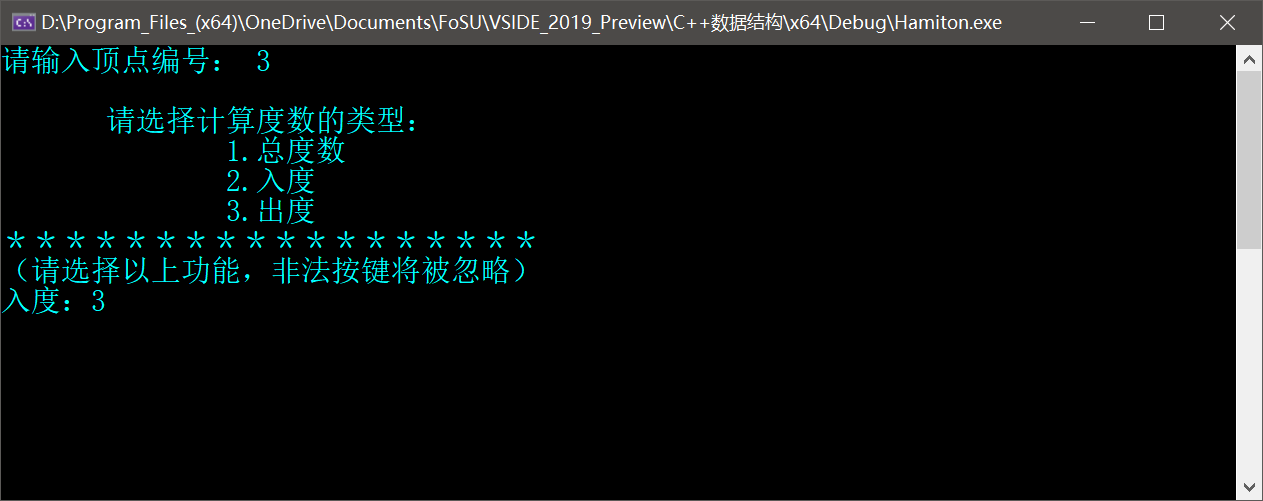


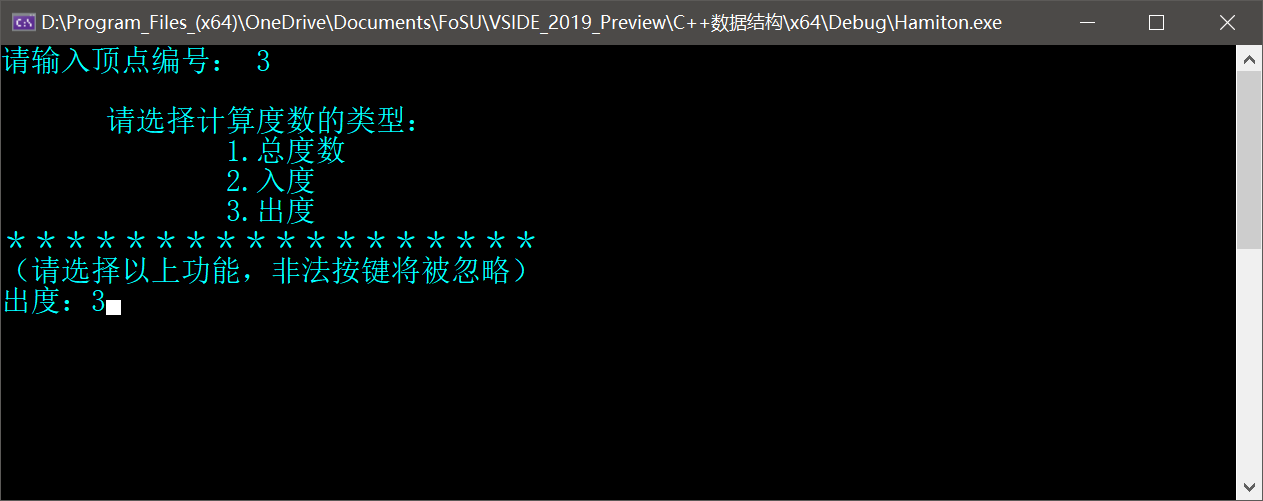
图5-9　入度  


图5-10　出度

选择功能4，输入遍历起点，显示结果如【图5-11】。

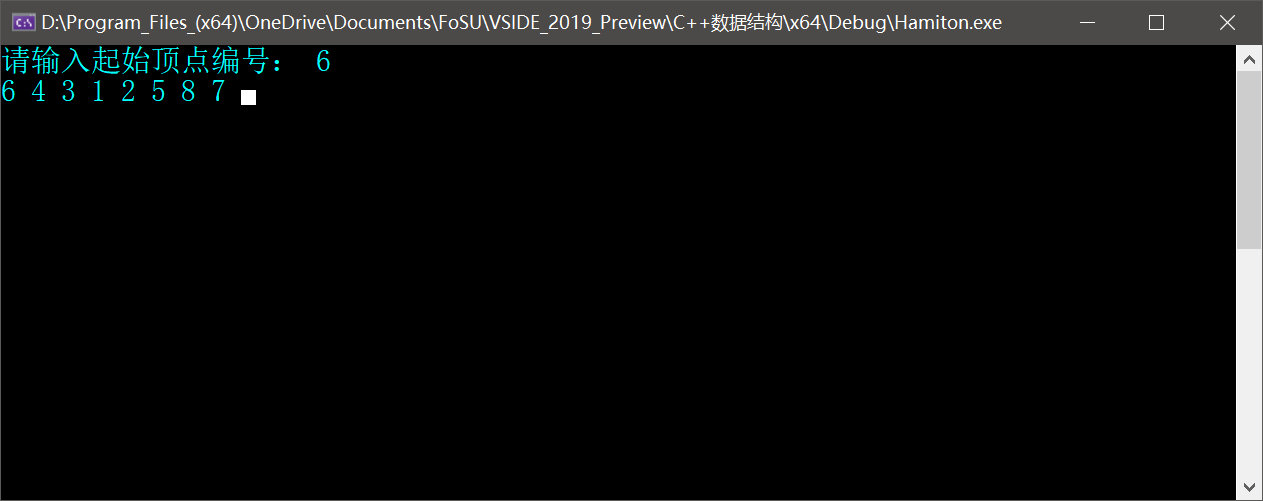


图5-11　深度优先遍历

选择功能5，输出哈密顿路径如【图5-12】。

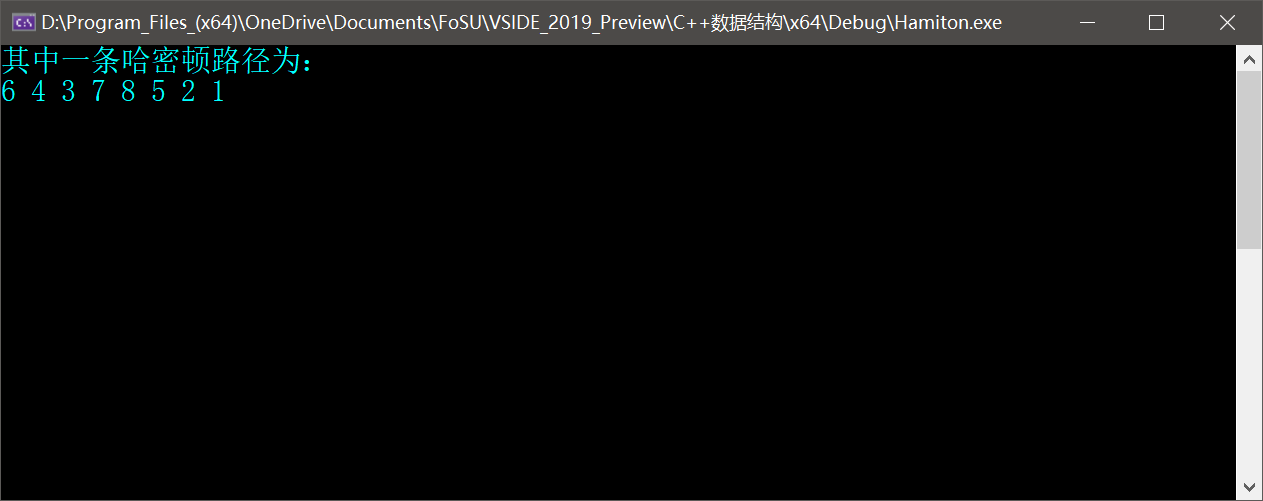


图5-12　哈密顿路径

选择功能1，重新创建一张图（简单九边形），如【图5-13】所示。

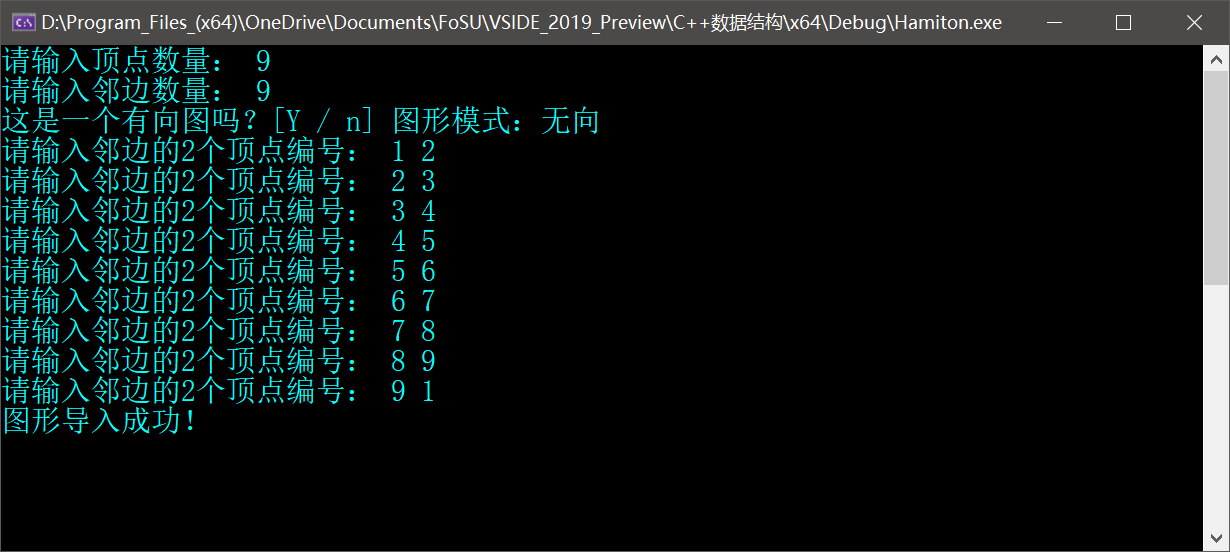


图5-13　从键盘输入数据

选择功能6，将图写入文件（如【图5-14】），导出的文件如【图5-15】所示。

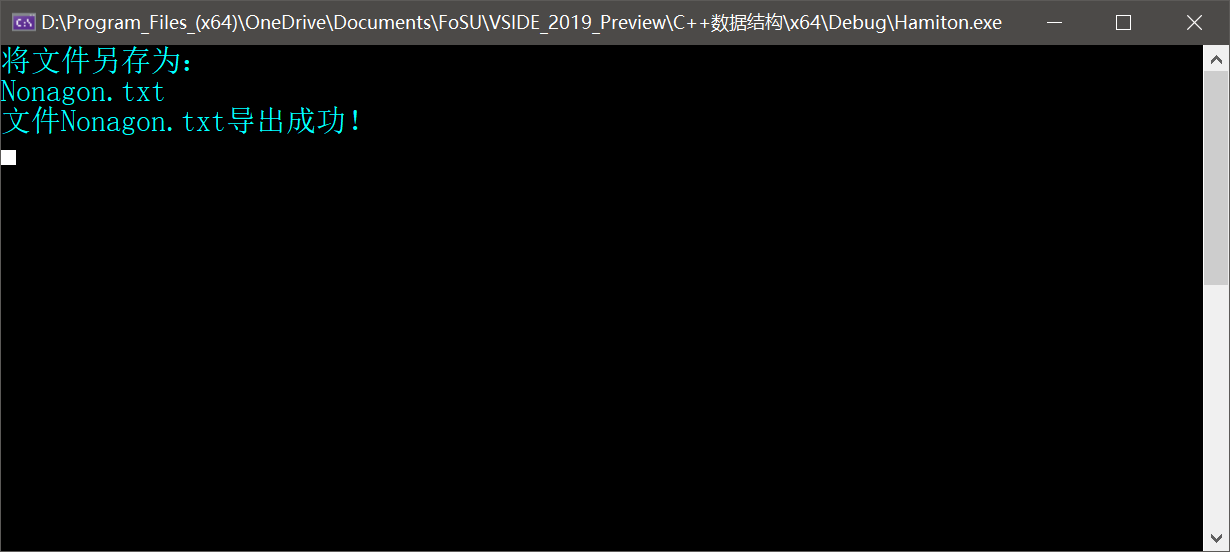


图5-14　导出文件

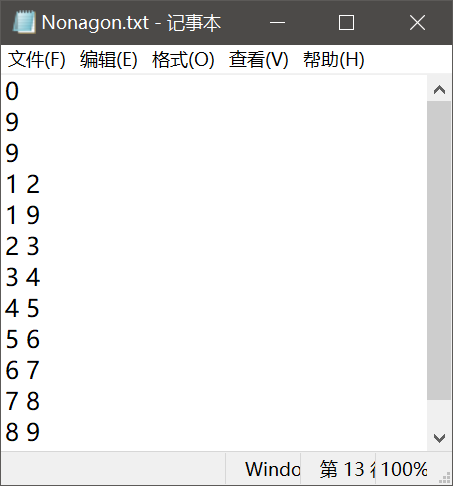
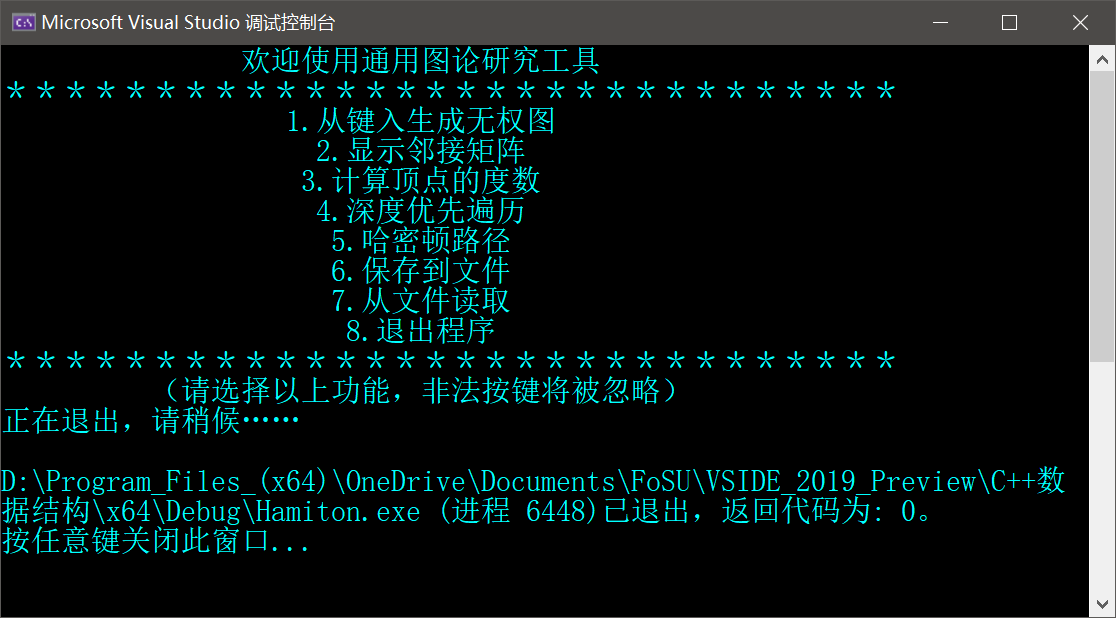


图5-15　导出后的文件

选择功能8，退出程序，如【图5-16】。



# 附录

# 文件清单与说明

Definition.hpp：无权图的类定义（数据成员定义和方法成员声明）；

Relize.hpp：无权图的类实现（方法成员实现）；

User\_Interface.cpp：与类无关的UI控制函数和主函数；

PathTable.txt：无权图的邻接表。

# 主要程序算法

|  |
| --- |
| // Definition.hpp  #pragma once  #include <iostream>  #include <fstream>  #include <stack>  using namespace std;  // 最大顶点数量  constexpr int MAX\_VERTEX = 9;  class Graph {  private:  // 图的类型  bool isDirectional;  // 邻接矩阵  bool adjacency[10][10] = { 0 };  // 顶点数量  int vertexSize;  // 邻边数量  int edgeSize;  // 节点访问记录  bool isVisited[10] = { 0 };  // 深度优先遍历（内核）  void \_dft(const int);  // 重置节点访问记录（内核）  void \_rv();  // 遍历路径  stack<int> pathStack;  // 深度优先搜索（内核）  void \_dfs(const int);  public:  // 记录对象是否被正确初始化  bool isCorrectlyInited;  // 度的类型  const enum DegreeType {  in = 1, out = 2, all = 3  };  // 创建邻接矩阵  Graph(const bool, const int, const int);  // 析构函数  ~Graph() {}  // 显示邻接矩阵  void showGraph() const;  // 计算度数、入度、出度  int getDegree(int, DegreeType) const;  // 深度优先遍历（公开）  void deepFirstTraverse(const int);  // 哈密顿路径  void hamitonPath();  // 另存为  void saveAs(const string);  // 打开  Graph(const string);  }; |
| // Definition.hpp  #pragma once  #include <iostream>  #include <fstream>  #include <stack>  using namespace std;  // 最大顶点数量  constexpr int MAX\_VERTEX = 9;  class Graph  {  private:  // 图的类型  bool isDirectional;  // 邻接矩阵  bool adjacency[10][10] = {0};  // 顶点数量  int vertexSize;  // 邻边数量  int edgeSize;  // 节点访问记录  bool isVisited[10] = {0};  // 深度优先遍历（内核）  void \_dft(const int);  // 重置节点访问记录（内核）  void \_rv();  // 遍历路径  stack<int> pathStack;  // 深度优先搜索（内核）  void \_dfs(const int);  public:  // 记录对象是否被正确初始化  bool isCorrectlyInited;  // 度的类型  const enum DegreeType {  in = 1,  out = 2,  all = 3  };  // 创建邻接矩阵  Graph(const bool, const int, const int);  // 析构函数  ~Graph() {}  // 显示邻接矩阵  void showGraph() const;  // 计算度数、入度、出度  int getDegree(int, DegreeType) const;  // 深度优先遍历（公开）  void deepFirstTraverse(const int);  // 哈密顿路径  void hamitonPath();  // 另存为  void saveAs(const string);  // 打开  Graph(const string);  }; |
| // User\_Interface.cpp  #include "Realize.hpp"  #include <conio.h>  #include <cstdlib>  #include <Windows.h>  using namespace std;  // 选项合法性校验（是否）  bool selected()  {  char choice = '\0';  do  {  choice = \_getch();  } while (choice != 'Y' && choice != 'y' && choice != 'n' && choice != 'N');  if (choice == 'Y' || choice == 'y')  {  cout << "图形模式：有向" << endl;  return true;  }  else  {  cout << "图形模式：无向" << endl;  return false;  }  }  // 选项合法性校验（选项）  char selected(const char \_min, const char \_max)  {  char choice = '\0';  do  {  choice = \_getch();  } while (choice < \_min || choice > \_max);  return choice;  };  // 用户主菜单  void graphicsInterface()  {  cout << " 欢迎使用通用图论研究工具 " << endl;  cout << "＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊" << endl;  cout << " 1.从键入生成无权图 " << endl;  cout << " 2.显示邻接矩阵 " << endl;  cout << " 3.计算顶点的度数 " << endl;  cout << " 4.深度优先遍历 " << endl;  cout << " 5.哈密顿路径 " << endl;  cout << " 6.保存到文件 " << endl;  cout << " 7.从文件读取 " << endl;  cout << " 8.退出程序 " << endl;  cout << "＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊" << endl;  cout << " （请选择以上功能，非法按键将被忽略） " << endl;  }  // 主函数  int main()  {  // 图论对象  Graph \*graph = nullptr;  while (true)  {  system("cls");  graphicsInterface();  switch (selected('1', '8'))  {  case '1':  {  system("cls");  cout << "请输入顶点数量：" << ends;  int vertexSize;  cin >> vertexSize;  cout << "请输入邻边数量：" << ends;  int edgeSize;  cin >> edgeSize;  cout << "这是一个有向图吗？[Y / n]" << ends;  bool isDirectional = selected();  graph = new Graph(isDirectional, vertexSize, edgeSize);  cout << "图形导入成功！" << endl;  Sleep(1000);  break;  }  case '2':  {  system("cls");  if (graph != nullptr)  {  graph->showGraph();  }  else  {  cerr << "错误：未建立图相关数据！请先通过键盘输入或文件导入图数据......" << endl;  }  Sleep(3000);  break;  }  case '3':  {  system("cls");  if (graph != nullptr)  {  cout << "请输入顶点编号：" << ends;  int vertexID;  cin >> vertexID;  cout << endl;  cout << " 请选择计算度数的类型： " << endl;  cout << " 1.总度数 " << endl;  cout << " 2.入度 " << endl;  cout << " 3.出度 " << endl;  cout << "＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊" << endl;  cout << "（请选择以上功能，非法按键将被忽略）" << endl;  switch (selected('1', '3'))  {  case '1':  cout << "总度数：" << graph->getDegree(vertexID, Graph::all);  break;  case '2':  cout << "入度：" << graph->getDegree(vertexID, Graph::in);  break;  case '3':  cout << "出度：" << graph->getDegree(vertexID, Graph::out);  break;  }  }  else  {  cerr << "错误：未建立图相关数据！请先通过键盘输入或文件导入图数据......" << endl;  }  Sleep(3000);  break;  }  case '4':  {  system("cls");  if (graph != nullptr)  {  cout << "请输入起始顶点编号：" << ends;  int vertexID;  cin >> vertexID;  graph->deepFirstTraverse(vertexID);  }  else  {  cerr << "错误：未建立图相关数据！请先通过键盘输入或文件导入图数据......" << endl;  }  Sleep(3000);  break;  }  case '5':  {  system("cls");  if (graph != nullptr)  {  cout << "其中一条哈密顿路径为：" << endl;  graph->hamitonPath();  }  else  {  cerr << "错误：未建立图相关数据！请先通过键盘输入或文件导入图数据......" << endl;  }  Sleep(3000);  break;  }  case '6':  {  system("cls");  if (graph != nullptr)  {  cout << "将文件另存为：" << endl;  string filePath;  cin >> filePath;  graph->saveAs(filePath);  }  else  {  cerr << "错误：未建立图相关数据！请先通过键盘输入或文件导入图数据......" << endl;  }  Sleep(3000);  break;  }  case '7':  {  system("cls");  cout << "从文件打开：" << endl;  string filePath;  cin >> filePath;  // 如果存在已经被打开的图，则关闭（释放）图对象  if (graph != nullptr)  {  delete graph;  // 图指针置空  graph = nullptr;  }  // 导入数据  graph = new Graph(filePath);  // 判断图是否被成功创建（不论导入成功与否，图指针必不为空）  if (!graph->isCorrectlyInited)  {  // 图没有被正确创建（打开），其他功能仍无法正常使用，  // 所以还需要关闭（释放）错误的图对象，让菜单屏蔽相应功能  delete graph;  // 图指针置空  graph = nullptr;  }  Sleep(3000);  break;  }  case '8':  {  cout << "正在退出，请稍候……" << endl;  Sleep(1000);  // 程序退出前销毁所有数据  delete graph;  graph = nullptr;  exit(0);  }  }  }  return 0;  } |