**实 验 报 告 （ 3 ）**

|  |  |
| --- | --- |
| **实验名称**：图的应用 | **实验地点**：宿舍，信息楼211 |
| **所使用的工具软件及环境：**  windows 11版本 10.0.22631 版本 22631:  GNU gdb (GDB) 7.6.1,  gcc version 6.3.0 (MinGW.org GCC-6.3.0-1) win32,  Vscode 1.94.2 | |
| **一、实验目的：**  1、掌握图的结构特征，以及各种存储结构的特点及使用范围。  2、掌握图的创建、遍历的算法。  3、掌握图的应用算法。 | |
| **二、实验内容及要求：**  **1.编写完整的程序，构建一个有n（自己定义）个顶点和e条边的带权图G。该图使用邻接表存储，并遍历邻接表，输出指定顶点的邻接点信息。**  **1）设计思路**  1．询问有向图还是无向图（无向图则添加边的时候要额外添加反向的边）  2.读入顶点数n和边数e，初始化图G。  3.根据边数，读入每条边/弧的信息（起点-终点-权重），完成图G邻接表的构造。  4.利用循环，输出指定顶点的邻接点信息，并输出相应边的权和相应节点的地址  //其他设计   1. 人性化的输入输出，以空格分隔序列的输入 2. 添加了程序错误运行日志，并将其重定向到txt文件（利用freopen） 3. 对程序进行模块化设计，将声明统一写在头文件内并加以详细注释（使头文件有较好的描述数据结构的效果），定义统一写在源文件内，变量命名使用下划线法增加可读性。 4. 引入了双端队列和DFS（但目前没有用到）。   **2）数据结构的描述**  个人认为在节点结构内定义邻接点地址和路径权重会更容易理解，单独定义边的结构在代码层面比较复杂，但是临近期末时间有限，并没有精力从结构出发重构整套代码并完善内存回收机制。况且邻接表存储时将边作为额外一个结构体使用也不会有特别大的困难。  //graph.h  图的相关声明  #pragma once  #include "headers.h"  typedef int vertex;  typedef int weight\_type;  #define max\_vertex\_num 100       //最大顶点数目  typedef struct AVNode \*p\_AVNode;    /\*  @brief 邻接表表头,不存储数据  \*/  typedef struct VNode  {      p\_AVNode first;     //该节点的邻接表  }adj\_list[max\_vertex\_num];  typedef struct GNode \*LGraph;  /\*  @brief 邻接表存储的图  \*/  struct GNode  {      int n\_v; // 顶点数      int n\_e; // 边数      adj\_list G; // 邻接表      bool directed;          //是否是有向图  };    /\*  @brief 邻接点  \*/  struct AVNode  {      vertex v; // 终点      weight\_type weight; // 边的权重      p\_AVNode next;      //下一个临界点  };    /\*  @brief 边的定义  \*/  typedef struct ENode \*PtrToENode;  struct ENode{      vertex v1, v2;      /\* 有向边<V1, V2> \*/      weight\_type weight;  /\* 权重 \*/  };  typedef PtrToENode Edge;  /\*  @brief 初始化一个有VertexNum个顶点但没有边的图  @param int 顶点个数  @return LGraph 图的指针  \*/  LGraph CreateGraph( int vertexNum );  void Visit( vertex V,p\_AVNode address);  void DFS( LGraph Graph, vertex V, void (\*Visit)(vertex,p\_AVNode) );  LGraph BuildGraph();  **3）程序代码**  **//err.h**  //用于错误输出 23010341杳泽  #pragma once  #include "headers.h"  /\*  @brief perror(str) 并写入stderr  @param char\* str要写入的错误信息  @return None  \*/  void print\_err(char \*str);  /\*  @brief 重定向stderr至 str.log  @param  char\* 重定向后的文件主名  @return None  \*/  void redirect\_stderr(char \*str) ;    //err.c  #include "err.h"    void print\_err(char \*str)        //perror(str)    并写入stderr  {      perror(str);      fprintf(stderr,"errno:%d\n\n",errno) ;      fprintf(stdout,"%s.errno:%d\n",str,errno);      return ;  }  void redirect\_stderr(char \*str)      //重定向stderr 至str.log  {      char location[20];      sprintf(location,"./%s.log",str);       //生成存储文件名      if(freopen(location, "a", stderr) == NULL)fprintf(stdout,"error redirecting stdout\n");      return ;  }    //graph.h  //graph.h  图的相关声明  #pragma once  #include "headers.h"  typedef int vertex;  typedef int weight\_type;  #define max\_vertex\_num 100       //最大顶点数目  typedef struct AVNode \*p\_AVNode;    /\*  @brief 邻接表表头,不存储数据  \*/  typedef struct VNode  {      p\_AVNode first;     //该节点的邻接表  }adj\_list[max\_vertex\_num];  typedef struct GNode \*LGraph;  /\*  @brief 邻接表存储的图  \*/  struct GNode  {      int n\_v; // 顶点数      int n\_e; // 边数      adj\_list G; // 邻接表      bool directed;          //是否是有向图  };    /\*  @brief 邻接点  \*/  struct AVNode  {      vertex v; // 终点      weight\_type weight; // 边的权重      p\_AVNode next;      //下一个临界点  };    /\*  @brief 边的定义  \*/  typedef struct ENode \*PtrToENode;  struct ENode{      vertex v1, v2;      /\* 有向边<V1, V2> \*/      weight\_type weight;  /\* 权重 \*/  };  typedef PtrToENode Edge;  /\*  @brief 初始化一个有VertexNum个顶点但没有边的图  @param int 顶点个数  @return LGraph 图的指针  \*/  LGraph CreateGraph( int vertexNum );  void Visit( vertex V,p\_AVNode address);  void DFS( LGraph Graph, vertex V, void (\*Visit)(vertex,p\_AVNode) );  LGraph BuildGraph();  //graph.c  // graph.c  #include "graph.h"  #include "err.h"  bool Visited[max\_vertex\_num]={false};       //访问过的顶点  LGraph CreateGraph(int vertexNum)  { /\* 初始化一个有vertexNum个顶点但没有边的图 \*/      vertex V;      LGraph Graph;      Graph = (LGraph)malloc(sizeof(struct GNode)); /\* 建立图 \*/      Graph->n\_v = vertexNum;      Graph->n\_e = 0;      /\* 初始化邻接表头指针 \*/      /\* 注意：这里默认顶点编号从0开始，到(Graph->n\_v - 1) \*/      for (V = 0; V < Graph->n\_v; V++)          Graph->G[V].first = NULL; // 开始的链表为空      return Graph;  }  void InsertEdge(LGraph Graph, Edge E)  {      p\_AVNode n\_ewNode;      /\* 插入边 <v1, v2> \*/      /\* 为v2建立新的邻接点 \*/      n\_ewNode = (p\_AVNode)malloc(sizeof(struct AVNode));      n\_ewNode->v = E->v2;      n\_ewNode->weight = E->weight;      /\* 将v2插入v1的表头 \*/      n\_ewNode->next = Graph->G[E->v1].first;      Graph->G[E->v1].first = n\_ewNode;      if (Graph->directed == false)       //无向图      {          /\* 若是无向图，还要插入边 <v2, v1> \*/          /\* 为v1建立新的邻接点 \*/          n\_ewNode = (p\_AVNode)malloc(sizeof(struct AVNode));          n\_ewNode->v = E->v1;          n\_ewNode->weight = E->weight;          /\* 将v1插入v2的表头 \*/          n\_ewNode->next = Graph->G[E->v2].first;          Graph->G[E->v2].first = n\_ewNode;      }  }  LGraph BuildGraph()  {      LGraph Graph;      Edge E;      vertex V;      int n\_v, i;      printf("请输入顶点个数：");      scanf("%d", &n\_v); /\* 读入顶点个数 \*/      getchar();      for (i = 0; i < n\_v; i++)          Visited[i] = 0;      Graph = CreateGraph(n\_v); /\* 初始化有n\_v个顶点但没有边的图 \*/      printf("有向图请输入y,无向图n:");      char is\_directed='n';      scanf("%c",&is\_directed);      getchar();      if(is\_directed=='y')    Graph->directed=true;      else if (is\_directed=='n')  Graph->directed=false;      else print\_err("方向错误.\n");      printf("请输入边数：");      scanf("%d", &(Graph->n\_e)); /\* 读入边数 \*/      getchar();      printf("请输入边\n|start|end|weight|\n");      if (Graph->n\_e != 0)      {                                           /\* 如果有边 \*/          E = (Edge)malloc(sizeof(struct ENode)); /\* 建立边结点 \*/          /\* 读入边，格式为"起点 终点 权重"，插入邻接矩阵 \*/          for (i = 0; i < Graph->n\_e; i++)          {              scanf("%d %d %d", &E->v1, &E->v2,&E->weight);              getchar();              /\* 注意：如果权重不是整型，weight的读入格式要改 \*/              InsertEdge(Graph, E);          }      }      /\* 如果顶点有数据的话，读入数据      for (V=0; V<Graph->n\_v; V++)          scanf(" %c", &(Graph->G[V].Data));\*/      return Graph;  }  void Visit( vertex V ,p\_AVNode address)  {      printf("------->V%d \n  [0x%p]\n\n", V,address);  }  /\* Visited[]为全局变量，已经初始化为false \*/  void DFS( LGraph Graph, vertex V, void (\*Visit)(vertex,p\_AVNode) )  {   /\* 以V为出发点对邻接表存储的图Graph进行DFS搜索 \*/      p\_AVNode W;        Visit( V,Graph->G[V].first ); /\* 访问第V个顶点 \*/      Visited[V] = 1; /\* 标记V已访问 \*/      for( W=Graph->G[V].first; W; W=W->next ) /\* 对V的每个邻接点W->AdjV \*/          if ( !Visited[W->v] )    /\* 若W->AdjV未被访问 \*/              DFS( Graph, W->v, Visit );  /\* 则递归访问之 \*/  }  //queue.h  //queue.h  #include "headers.h"  typedef char ElementType ;  typedef int Position;  typedef struct QNode \*Deque;  struct QNode {      //队列结构      ElementType \*Data;      //数据      Position Front, Rear;       //首元素的下标和尾元素+1的下标      int MAXSIZE;        //双端队列容纳的最大数据容量  };    /\*  @brief 初始化双端队列  @param int 队列能容纳的最大元素个数  @return 队列的指针  \*/  Deque init\_deque(int max\_size);  /\*  @brief 判断是否队满  @param Deque 双端队列指针  @return bool  \*/  bool IsFull(Deque Q);  /\*  @brief 判断是否队空  @param Deque 双端队列指针  @return bool  \*/  bool IsEmpty(Deque Q);    /\*  @brief 删除队首数据  @param Deque 要操作的队列指针  @return ElementType 返回删除的元素  \*/  ElementType Pop(Deque Q);    /\*  @brief 尾插数据  @param Deque  要操作的双端队列指针  @return bool 成功 ->true  \*/  bool Inject(Deque Q,ElementType x);    /\*  @brief 头插数据  @param Deque,ElementType  要插入的双端队列,元素  @return bool succeed ->true  \*/  bool Push(Deque Q, ElementType x);    /\*  @brief  尾部删除数据  @param Deque 要操作的双端的队列  @return ElementType 删除的数据  \*/  ElementType Eject(Deque Q);    /\*  @brief 打印Deque内的元素  @param Deque 要操作的双端队列  @return None  \*/  void print\_deque(Deque Q);    /\*  @brief 删除队列(释放内存)  @param Deque 要操作的队列  @return None  \*/  void del\_Deque(Deque Q) ;    //删除队列    //queue.c  //queue.c  #include "queue.h"  const int ERROR=-1;    /\*  @brief 初始化双端队列  @param int 队列能容纳的最大元素个数  @return 队列的指针  \*/  Deque init\_deque(int max\_size)  {      Deque head=(Deque)malloc(sizeof(struct QNode));      head->Data=(char \*)malloc(sizeof(char)\*max\_size);      head->MAXSIZE=max\_size;      head->Front=0;      head->Rear=0;       //初始化      return head;    //返回  }  /\*  @brief 判断是否队满  @param Deque 双端队列指针  @return bool  \*/  bool IsFull(Deque Q)  {      if((Q->Rear+1)%Q->MAXSIZE == Q->Front)      {          return true;      }      return false;  }  /\*  @brief 判断是否队空  @param Deque 双端队列指针  @return bool  \*/  bool IsEmpty(Deque Q)  {      if(Q->Rear == Q->Front)      {          return true;      }      return false;  }  /\*  @brief 删除队首数据  @param Deque 要操作的队列指针  @return ElementType 返回删除的元素  \*/  ElementType Pop(Deque Q)  {      if(IsEmpty(Q))      {          printf("队列空！\n");          return ERROR;      }      ElementType data = Q->Data[Q->Front];      Q->Front=(Q->Front+1)%Q->MAXSIZE; // 向后移动front指针      return data;  }  /\*  @brief 尾插数据  @param Deque  要操作的双端队列指针  @return bool 成功 ->true  \*/  bool Inject(Deque Q,ElementType x)  {      if(IsFull(Q))      {          printf("队列满！\n");          return false;      }      Q->Data[Q->Rear] = x; // 数据插入      Q->Rear = (Q->Rear+1)%Q->MAXSIZE; // 向后移动rear指针      return true;  }  /\*  @brief 头插数据  @param Deque,ElementType  要插入的双端队列,元素  @return bool succeed ->true  \*/  bool Push(Deque Q, ElementType x)  {      if(IsFull(Q))      {          printf("队列已满！\n");          return false;      }      // 计算要插入的位置（数组下标）      Q->Front = (Q->Front-1+Q->MAXSIZE)%Q->MAXSIZE;      // 数据插入      Q->Data[Q->Front] = x;      return true;  }  /\*  @brief  尾部删除数据  @param Deque 要操作的双端的队列  @return ElementType 删除的数据  \*/  ElementType Eject(Deque Q)  {      if(IsEmpty(Q))      {          printf("队列空！\n");          return ERROR;      }      // 向前移动rear指针      Q->Rear=(Q->Rear-1+Q->MAXSIZE)%Q->MAXSIZE;      ElementType data=Q->Data[Q->Rear];      return data;  }  /\*  @brief 打印Deque内的元素  @param Deque 要操作的双端队列  @return None  \*/  void print\_deque(Deque Q)  {      for(int i=0;i<Q->MAXSIZE;i++)      {          if(Q->Data[i]<33||Q->Data[i]>126)      //占位符          {              printf("占位符    |");          }          else printf("%c         |",Q->Data[i]);      //打印数据     用\t总是莫名其妙对不齐      }      printf("\n");      for(int i=0;i<Q->MAXSIZE;i++)      {          printf("[%p]|",&Q->Data[i]);    //打印地址      }      printf("\t Front:%p,Rear:%p",&Q->Data[Q->Front],&Q->Data[Q->Rear]);      printf("\n\n");  }  /\*  @brief 删除队列(释放内存)  @param Deque 要操作的队列  @return None  \*/  void del\_Deque(Deque Q)     //删除队列  {      free(Q->Data);      Q->Data=NULL;      free(Q);      Q=NULL;  }  //headers.h  //类似于 #include <std/bitsc++.h>       集合了所需的所有头文件的头文件  #pragma once  #include <errno.h>  #include <stdio.h>  #include <stdbool.h>  #include <stdlib.h>  #include <string.h>  #include <math.h>  #include <time.h>  #include <ctype.h>    //main.c  #include "headers.h"  #include "graph.h"  #include "err.h"  extern bool Visited[];      //声明 访问过的节点  int main()  {      LGraph Graph;      vertex V;        Graph=BuildGraph();        printf("请输入要查看的顶点序号:");      int check=0;      scanf("%d",&check);      getchar();      if (check>=Graph->n\_e)      {          char tmp\_str[20];          sprintf(tmp\_str,"不存在编号为%d的顶点",check);          print\_err(tmp\_str);          exit(-1);      }      //DFS(Graph,check,Visit);        p\_AVNode tmp=Graph->G[check].first;      printf("V%d [0x%p]\n",check,tmp);      //int visited\_ad      for(;tmp!=NULL;tmp=tmp->next)      {          printf("\t\t------->V%d [0x%p]\n\t\t [%d]\n\n",tmp->v,tmp,tmp->weight);      }          return 0;  }  **2.实现下图的“邻接矩阵存储的Dijkstra算法”，要求调试程序时，邻接矩阵使用文件的方式读入，程序功能：能够输出V0到任意顶点的最短路径。**    **1）设计思路**   1. 利用fscanf,fgets和strtok读取文件中的数据 2. 将顶点数、边数、邻接矩阵存储的图初始化并存入数据。 3. 利用Dijkstra求v0点到其他点的dist 和path 4. 利用for循环结合printf输出dist和path   //其他设计   1. 人性化的输入输出，以空格分隔序列的读取 2. 对程序进行模块化设计，将声明统一写在头文件内并加以详细注释（使头文件有较好的描述数据结构的效果），定义统一写在源文件内，变量命名使用下划线法增加可读性。   **2）数据结构的描述**  //详情见Dijkstra\_algo.h  #define MaxVertexNum 100    /\* 最大顶点数设为100 \*/  #define INFINITY 65535      /\* ∞设为双字节无符号整数的最大值65535\*/  //#define ERROR -1  typedef int Vertex;         /\* 用顶点下标表示顶点,为整型 \*/  typedef int WeightType;     /\* 边的权值设为整型 \*/  typedef char DataType;      /\* 顶点存储的数据类型设为字符型 \*/  /\* 图结点的定义 \*/  typedef struct GNode \*PtrToGNode;  struct GNode{      int Nv;  /\* 顶点数 \*/      int Ne;  /\* 边数   \*/      WeightType G[MaxVertexNum][MaxVertexNum]; /\* 邻接矩阵 \*/      //DataType Data[MaxVertexNum];      /\* 存顶点的数据 \*/   无数据      /\* 注意：很多情况下，顶点无数据，此时Data[]可以不用出现 \*/  };  typedef PtrToGNode MGraph; /\* 以邻接矩阵存储的图类型 \*/  /\* 边的定义 \*/  typedef struct ENode \*PtrToENode;  struct ENode{      Vertex V1, V2;      /\* 有向边<V1, V2> \*/      WeightType Weight;  /\* 权重 \*/  };  typedef PtrToENode Edge;  /\*  @brief 读取数据  @param void  @return Mgraph   存储数据的图  \*/  MGraph read\_data(void);  /\*  @brief  初始化一个有VertexNum个顶点但没有边的图  @param int 定点数  @return Mgraph 图  \*/  MGraph CreateGraph(int VertexNum);  /\*  @brief 向图中插入边  @param Mgraph 要插入的图  @param 插入的边  @return None  \*/  void InsertEdge(MGraph Graph, Edge E);  /\*  @brief 返回未被收录顶点中dist最小者  @param Mgraph 图  @param int[] dist数组  @param int[] 收录过的数组  @return Vertex 未被收录顶点中dist最小者  \*/  Vertex FindMinDist(MGraph Graph, int dist[], int collected[]);  /\*  @brief Dijkstra算法  @param Mgraph 图  @param int[]        dist[]  @param int[]        path[]  @param Vertex   要寻找最小路径的顶点  @return bool  \*/  bool Dijkstra(MGraph Graph, int dist[], int path[], Vertex S);  **3）程序代码**  //Dijkstra\_algo.h  #pragma once  #include<stdio.h>  #include<stdlib.h>  #include <stdbool.h>  #include <windows.h>  #define MaxVertexNum 100    /\* 最大顶点数设为100 \*/  #define INFINITY 65535      /\* ∞设为双字节无符号整数的最大值65535\*/  //#define ERROR -1  typedef int Vertex;         /\* 用顶点下标表示顶点,为整型 \*/  typedef int WeightType;     /\* 边的权值设为整型 \*/  typedef char DataType;      /\* 顶点存储的数据类型设为字符型 \*/  /\* 图结点的定义 \*/  typedef struct GNode \*PtrToGNode;  struct GNode{      int Nv;  /\* 顶点数 \*/      int Ne;  /\* 边数   \*/      WeightType G[MaxVertexNum][MaxVertexNum]; /\* 邻接矩阵 \*/      //DataType Data[MaxVertexNum];      /\* 存顶点的数据 \*/   无数据      /\* 注意：很多情况下，顶点无数据，此时Data[]可以不用出现 \*/  };  typedef PtrToGNode MGraph; /\* 以邻接矩阵存储的图类型 \*/  /\* 边的定义 \*/  typedef struct ENode \*PtrToENode;  struct ENode{      Vertex V1, V2;      /\* 有向边<V1, V2> \*/      WeightType Weight;  /\* 权重 \*/  };  typedef PtrToENode Edge;  /\*  @brief 读取数据  @param void  @return Mgraph   存储数据的图  \*/  MGraph read\_data(void);  /\*  @brief  初始化一个有VertexNum个顶点但没有边的图  @param int 定点数  @return Mgraph 图  \*/  MGraph CreateGraph(int VertexNum);  /\*  @brief 向图中插入边  @param Mgraph 要插入的图  @param 插入的边  @return None  \*/  void InsertEdge(MGraph Graph, Edge E);  /\*  @brief 返回未被收录顶点中dist最小者  @param Mgraph 图  @param int[] dist数组  @param int[] 收录过的数组  @return Vertex 未被收录顶点中dist最小者  \*/  Vertex FindMinDist(MGraph Graph, int dist[], int collected[]);  /\*  @brief Dijkstra算法  @param Mgraph 图  @param int[]        dist[]  @param int[]        path[]  @param Vertex   要寻找最小路径的顶点  @return bool  \*/  bool Dijkstra(MGraph Graph, int dist[], int path[], Vertex S);  //main.c  #include "Dijkstra\_algo.h"  int main()  {      MGraph Graph;      int i;      Graph=read\_data();      int dist[MaxVertexNum],path[MaxVertexNum];      Dijkstra(Graph,dist,path,0);      for(i=0; i<Graph->Nv;i++)         printf("V%d的dist是%d,path是%d.\n",i,dist[i],path[i]);      return 0;  }  // Dijkstra.c  #include "Dijkstra\_algo.h"  MGraph CreateGraph(int VertexNum)  { /\* 初始化一个有VertexNum个顶点但没有边的图 \*/      Vertex V, W;      MGraph Graph;      Graph = (MGraph)malloc(sizeof(struct GNode)); /\* 建立图 \*/      Graph->Nv = VertexNum;      Graph->Ne = 0;      /\* 初始化邻接矩阵 \*/      /\* 注意：这里默认顶点编号从0开始，到(Graph->Nv - 1) \*/      for (V = 0; V < Graph->Nv; V++)          for (W = 0; W < Graph->Nv; W++)              Graph->G[V][W] = INFINITY;      return Graph;  }  void InsertEdge(MGraph Graph, Edge E)  {      /\* 插入边 <V1, V2> \*/      Graph->G[E->V1][E->V2] = E->Weight;      /\* 若是无向图，还要插入边<V2, V1> \*/      Graph->G[E->V2][E->V1] = E->Weight;  }  MGraph read\_data()  {      MGraph Graph;      FILE \*fp = NULL;      fp = fopen("./graph.txt", "r");      if (!fp)      {          printf("fopen err!.");          exit(-1);      }      int Nv=0,Ne=0;      fscanf(fp, "%d %d", &Nv, &Ne); // 读取顶点个数      fscanf(fp,"\n");      Graph=CreateGraph(Nv);      Graph->Ne=Ne;      for (int i = 0; i < Graph->Nv; i++)     //i  行      {          char str[200];          fgets(str,200,fp);      //读取一行          char \*tmp=strtok(str," ");          if(\*tmp!='N') Graph->G[i][0]=\*tmp-48;          for(int j=1;j<Graph->Ne;j++)          {              tmp=strtok(NULL," ");          //分隔字符              if(!tmp)  break;              if(\*tmp!='N')  Graph->G[i][j]=\*tmp-48;       //存储数据          }      }      return Graph;  }  Vertex FindMinDist(MGraph Graph, int dist[], int collected[])  { /\* 返回未被收录顶点中dist最小者 \*/      Vertex MinV, V;      int MinDist = INFINITY;      for (V = 0; V < Graph->Nv; V++)      {          if (collected[V] == false && dist[V] < MinDist)          {              /\* 若V未被收录，且dist[V]更小 \*/              MinDist = dist[V]; /\* 更新最小距离 \*/              MinV = V;          /\* 更新对应顶点 \*/          }      }      if (MinDist < INFINITY) /\* 若找到最小dist \*/          return MinV;        /\* 返回对应的顶点下标 \*/      else          return ERROR; /\* 若这样的顶点不存在，返回错误标记 \*/  }  bool Dijkstra(MGraph Graph, int dist[], int path[], Vertex S)  {      int collected[MaxVertexNum];      Vertex V, W;      /\* 初始化：此处默认邻接矩阵中不存在的边用INFINITY表示 \*/      for (V = 0; V < Graph->Nv; V++)      {          dist[V] = Graph->G[S][V];          if (dist[V] < INFINITY)              path[V] = S;          else              path[V] = -1;          collected[V] = false;      }      /\* 先将起点收入集合 \*/      dist[S] = 0;      collected[S] = true;      while (1)      {          /\* V = 未被收录顶点中dist最小者 \*/          V = FindMinDist(Graph, dist, collected);          if (V == ERROR)                 /\* 若这样的V不存在 \*/              break;                      /\* 算法结束 \*/          collected[V] = true;            /\* 收录V \*/          for (W = 0; W < Graph->Nv; W++) /\* 对图中的每个顶点W \*/                                          /\* 若W是V的邻接点并且未被收录 \*/              if (collected[W] == false && Graph->G[V][W] < INFINITY)              {                  if (Graph->G[V][W] < 0) /\* 若有负边 \*/                      return false;       /\* 不能正确解决，返回错误标记 \*/                  /\* 若收录V使得dist[W]变小 \*/                  if (dist[V] + Graph->G[V][W] < dist[W])                  {                      dist[W] = dist[V] + Graph->G[V][W]; /\* 更新dist[W] \*/                      path[W] = V;                        /\* 更新S到W的路径 \*/                  }              }      } /\* while结束\*/      return true; /\* 算法执行完毕，返回正确标记 \*/  } | |
| **三、程序运行结果（分析计算时空复杂度，运行界面截图）**  **1.exp-3-1**  **//有向图**    //有向图    **//无向图**    **2. exp-3-2**  **//测试用例graph.txt文件见附件压缩包**  **//测试用例无向图**    **//运行结果**    **时空复杂度分析**  **邻接矩阵下使用Dijkstra算法**  时间复杂度T(n)=O(|V|^2)  若使用邻接表 T(n)可改进为 O(|E| log |V|)  邻接表实现的Dijkstra算法的空间复杂度是S(n)=O(|E| +| V|)  邻接矩阵实现的Dijkstra算法的空间复杂度是S(n)=O(|V|^2)  **邻接表的遍历**  遍历邻接表的时间复杂度是O(|V| +|E|)  遍历邻接表的空间复杂度是O(1) | |

**成绩：\_\_\_\_\_\_\_\_\_\_\_\_** **任课教师签名：\_\_\_\_\_\_\_\_\_\_\_\_** 2024年 月 日