实验题第 1 题

第 1 小题

(1) 设计思路

本题要求**基于图的深度优先搜索策略写一个算法，判别有向图中是否存在由顶点 vi 到顶点 vj 的路径（i!=j）**。

- 根据测试程序给出的形式(if(ExistPathDFS(graph,start,end)))可知，函数需要返回一个 bool 类型的返回值，故将函数定义为 bool 类型。
- 由于有向图的 DFS 过程是从起始点开始顺着路径的方向往下遍历的，故只要在 DFS 序列中出现目标点，则代表存在从 a 到 b 的路径。因此可以把问题转化为从 a 开始进行 DFS，判断 DFS 序列中是否存在 e，存在则返回 true，不存在则返回 false。
- 根据先前写的 DFS 算法，可以直接在其基础上进行修改，完成本题要求的算法，得到代码如下 i。因为输入变量的类型变了，所以在递归过程中一些顺序要相应改变，并且要增加返回的条件和步骤。根据题目，设置 a=b 为结束递归的条件，返回 true。
- 需要注意的是，题目要求多次调用 ExistPathDFS，而 DFS 是一个递归的过程，于是需要利用一个递归之外的接口函数去完成重置访问数组的工作。

(2) 源代码

bool ExistPathDFSh(const ALDirNetwork<DataType,WeightType> &graph, DataType a, DataType b){}; **//基于 DFS 判别有向图中是否存在由 a 到 b 的路径的 DFS 辅助函数**

```
template <class DataType, class WeightType>
bool ExistPathDFSh(const ALDirNetwork<DataType,WeightType> &graph, DataType a,
DataType b){
  int v1 = graph.GetOrder(a);          //获取 a 的地址
  int v2 = graph.GetOrder(b);          //获取 b 的地址
  if(a == b) return true;              //如果 ab 相同，返回 true（递归的结束条件）

  graph.SetVisitedTag(v1,VISITED);     //标记 v1 为已访问
  //遍历邻接点递归调用 DFS
  for(int w1 = graph.GetFirstAdjvex(v1); w1 != -1; w1 = graph.GetNextAdjvex(v1,w1)){
    DataType e;                        //设置 e 来存放获取的节点信息
    graph.GetElem(w1, e);              //获取 v1 邻接点 w1 的信息，存放到 e 中
    //若 w1 未被访问过，则从 w1 开始递归调用 DFS
    if(graph.GetVisitedTag(w1) == UNVISITED)
          //如果递归结束条件满足，返回 true 回溯到上一层
          if(ExistPathDFSh(graph, e, b))
                return true;
   }
   return false;  //遍历完所有节点，没有返回 true 的，最终返回 false
}
```

bool ExistPathDFS(const ALDirNetwork<DataType,WeightType> &graph, DataType a, DataType b){}; **//基于 DFS 判别有向图中是否存在由 a 到 b 的路径的接口函数**

```
template <class DataType, class WeightType>
bool ExistPathDFS(const ALDirNetwork<DataType,WeightType> &graph, DataType a,
DataType b){
```

```
  //如果辅助函数成功，获取到目标节点
  if(ExistPathDFSh(graph, a, b)){
    //初始化访问数组并返回 true
    for (int v = 0; v < graph.GetVexNum(); v++) {
          graph.SetVisitedTag(v, UNVISITED);
      }
    return true;
  }
  //否则，初始化访问数组并返回 false
  else{
    for (int v = 0; v < graph.GetVexNum(); v++) {
          graph.SetVisitedTag(v, UNVISITED);
      }
    return false;
  }
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

在设计 DFS 寻找目标点（路径）的程序时，要重点关注递归的返回条件，避免陷入死循环。同时还要注意输入和实际操作时数据类型的转换，确保可以利用现有资源对给定目标进行针对性的解决。

第 2 小题

(1) 设计思路

本题要求**基于图的广度优先搜索策略写一个算法，判别有向图中是否存在由顶点 vi 到顶点 vj 的路径（i!=j）**。

- 根据测试程序给出的形式(`if(ExistPathBFS(graph,start,end))`)可知，函数需要返回一个 bool 类型的返回值，故将函数定义为 bool 类型。
- 由于有向图的 BFS 过程是从起始点开始顺着路径的方向往下遍历的，故只要在 BFS 序列中出现目标点，则代表存在从 a 到 b 的路径。因此可以把问题转化为从 a 开始进行 BFS，判断 BFS 序列中是否存在 e，存在则返回 true，不存在则返回 false。
- 根据先前写的 BFS 算法，可以直接在其基础上进行修改，完成本题要求的算法，得到代码如下。
- 需要注意的是，由于本题多次调用此函数，但 BFS 算法本身并没有重置访问数组的过程，因此要手动对其进行重置。对此，我在函数的开头对访问数组进行了重置，

(2) 源代码

bool ExistPathBFS(const ALDirNetwork<DataType,WeightType> &graph, DataType a, DataType b){}; **//基于 BFS 判别有向图中是否存在由 a 到 b 的路径**

```
template <class DataType, class WeightType>
bool ExistPathBFS(const ALDirNetwork<DataType,WeightType> &graph, DataType a,
DataType b){
  //重置访问数组
  for (int v = 0; v < graph.GetVexNum(); v++) { //遍历所有顶点节点，初始化为未访问
        graph.SetVisitedTag(v, UNVISITED);
    }

  int v1 = graph.GetOrder(a);                    //获取 a 的地址
  int v2 = graph.GetOrder(b);                    //获取 b 的地址
  if(a == b) return true;                        //如果 ab 相同，返回 true（不考虑自回路）

  LinkQueue<int> vexq;                           //创建队列来辅助完成 BFS
  int u,w;                                       //创建 u，w 辅助完成邻接点的获取
  DataType e;                                    //创建 e 来获取

  graph.SetVisitedTag(v1,VISITED);               //标记 v1 为已访问
  vexq.EnQueue(v1);                              //v1 入队

  while(!vexq.IsEmpty()){                        //当队列非空
    vexq.DelQueue(u);                            //队头元素出队，存放到 u 中
    //按照邻接链表的顺序遍历 u 的邻接点
    for(w = graph.GetFirstAdjvex(u); w != -1; w = graph.GetNextAdjvex(u,w)){
      //如果邻接点未访问过，执行以下操作
      if (graph.GetVisitedTag(w) == UNVISITED){
        graph.GetElem(w, e);                     //获取邻接点 w 的值存放到 e 中
        graph.SetVisitedTag(w, VISITED);         //标记 w 为已访问
        vexq.EnQueue(w);                         //w 入队
        //如果 w 的值与目标值一致，即 e=b，则找到路径，返回 true
        if(e == b) {
          return true;                           //返回 true
```

```
        }
      }
    }
  }
  return false;//从 a 出发的所有节点遍历结束，未找到从 a 到 b 的路径，返回 false
}
```

(3) 说明

上述代码可以应用测试程序进行测试并得到正确结果。

BFS 不同于 DFS，其不需要递归，思维的深度较浅，代码实现较简单。需要注意的是要根据需求自己补全重置的步骤。

第 3 小题

(1) 设计思路

本题要求**利用 Dijkstra 算法求指定源点到其余各顶点的最短路径,要求输出最短路径及其长度**。

- 首先需要熟悉 Dijkstra 算法的原理和操作流程，识别到 dist 和 path 数组的特征，才能利用其来完成题目所给出的要求。
- 根据 Dijkstra 的特征，直接利用 path 和 dist 数组就可以完成最短路径和长度的输出。
- 长度比较好输出，直接输出 dist[v]即可，而路径的输出就需要通过 path 数组进行反向回溯，获取到完整的从 v 到 v0 的路径，这也是本函数的一个设计难点。
  - 由于是从 path 数组获取路径，则需要先从到目标点 v 开始，再进行回溯直到 v0;
  - 利用栈的 LIFO 原理，可以利用数组模拟一个栈的入栈和出栈，即正向遍历存入数组，反向遍历输出数组。这样就可以保证最终输出的路径顺序正确。
- 根据以上思想，得到代码如下。

(2) 源代码

void Dijkstra(const ALDirNetwork<DataType,WeightType> &g, int v0,WeightType dist,int path){}; **//Dijkstra 算法实现函数**

```
template <class DataType, class WeightType>
void Dijkstra(const ALDirNetwork<DataType,WeightType> &g, int v0,WeightType*
dist,int* path){
  WeightType mindist, infinity = g.GetInfinity();    //定义最小距离和无穷大
  int u, v;                                          //定义 u，v 来辅助完成算法
  //初始化距离和路径数组
  for(v = 0; v < g.GetVexNum(); v++){    //从 v 出发遍历所有点
    dist[v] = g.GetWeight(v0, v);        //获取从 v0 出发到每个节点的弧长权值，存到 dist 里
    if (dist[v] == infinity)             //初始化 path 数组，如果存在弧，则存入 v0，否则记-1
      path[v] = -1;
    else
      path[v] = v0;
    g.SetVisitedTag(v, UNVISITED);       //检查完一个点后，标记为未访问，表示未确定最短路径
  }

  g.SetVisitedTag(v0, VISITED);          //v0 本身不用找最短路径
  //遍历 v0 以外的所有点，寻找 v0 到其他点的最短路径
  for (int i = 1; i < g.GetVexNum(); i++){
    u = v0;                              //记录当前路径最短的点，初始化为 v0
    mindist = infinity;                  //初始化最短距离为 infinity
    //在尚未确定最短路径的点中寻找路径最短点
    for(v = 0; v < g.GetVexNum(); v++)
      if (g.GetVisitedTag(v) == UNVISITED && dist[v] < mindist){
        u = v;
        mindist = dist[v];
      }
    g.SetVisitedTag(u, VISITED);         //标记 u 为已访问，表示确定了其最短路径
    //更新从起始点 v0 到 u 的邻接点的最短路径
    for(v = g.GetFirstAdjvex(u); v != -1; v = g.GetNextAdjvex(u,v ))
      if (g.GetVisitedTag(v) == UNVISITED && mindist + g.GetWeight(u, v) < dist[v]){
        dist[v] = mindist + g.GetWeight(u, v);
        path[v] = u;
```

```
    }
  }
}
```

void OutputShortestPath(const ALDirNetwork<DataType, WeightType> &g, int v0, WeightType dist, int path); //配合 Dijkstra 函数的输出函数

```
template <class DataType, class WeightType>
void OutputShortestPath(const ALDirNetwork<DataType, WeightType> &g, int v0,
WeightType* dist, int* path) {
  int vexNum = g.GetVexNum();
  WeightType infinity = g.GetInfinity();
  DataType a,b;
  //遍历所有节点，检查其最短路径
  for (int v = 0; v < vexNum; v++) {
    if (v == v0) continue; //跳过起点自身
    //检查是否可达
    if (dist[v] == infinity) {              //若不可达，则输出不可达信息
      g.GetElem(v0,a);
      g.GetElem(v,b);
        cout << "There is no path between " << a << " and " << b << endl;
     } else {                               //若可达，则输出路径和长度
      //构建路径数组
      int pathArray[vexNum];              //存储路径的数组
      int count = 0;                      //记录路径中的节点数
      int current = v;                    //初始化 current 来跟踪当前节点的地址
      //通过 path 数组向前求得路径
      while (current != -1) {
        pathArray[count++] = current;     //将 current 存入路径数组并且向后移动
        current = path[current];          //通过 path 数组获得前驱节点的地址
      }
      //反向输出路径数组
       g.GetElem(v0,a);
       g.GetElem(v,b);
       cout << "The shortest path between " << a << " and " << b << " is:" << endl;
       //遍历路径数组并输出对应元素，用空格隔开
       for (int i = count - 1; i >= 0; i--) {
         DataType e;                        //定义辅助变量 e 来输出路径节点
         g.GetElem(pathArray[i], e);        //输出反向遍历路径数组中的节点值
         cout << e;
         if(i > 0)
          cout << " ";
       }
       cout << endl;
       //输出最短路径长度，直接取 dist[v]即可
       cout << "The distance is: " << dist[v] <<endl;
      }
   }
}
```

(3)说明

上述代码可以应用测试程序进行测试并得到正确结果。

值得注意的是，需要先熟悉 dijkstra 算法以及结果的存储结构，才能掌握对具体变量的操作。