```
/*图的邻接矩阵存储结构定义*/
#define MaxVertexNum 100
typedef char VertexType;
typedef int EdgeType;
typedef struct {
    VertexType Vex[MaxVertexNum];//顶点表
    EdgeType Edge[MaxVertexNum][MaxVertexNum];//邻接矩阵，边表
    int vexnum, arcnum;          //图的当前顶点数和弧数
} MGragh;

/*图的邻接表存储结构定义*/
typedef struct ArcNode {         //边表结点
    int adjvex;                  //该弧所指向的顶点的位置
    struct ArcNode *next;        //指向下一个条弧的指针
    //InfoType info;             //网的边权值
} ArcNode;
typedef struct VNode {           //顶点表结点
    VertexType data;             //顶点信息
    ArcNode *first;              //指向第一条依附该顶点的弧的指针
} VNode, AdjList[MaxVertexNum];
typedef struct {
    AdjList vertices;            //邻接表
    int vexnum, arcnum;          //邻接表的顶点数和弧数
} ALGraph;                       //ALGraph 是以邻接表存储的图类型

/*图的十字链表存储结构定义(有向图)*/
typedef struct xArcNode {//边表结点
    int tailvex, headvex;//该弧的头尾结点
    struct xArcNode *hlink, *tlink;//分别指向弧头相同，弧尾相同的结点
    //InfoType info;
} xArcNode;
typedef struct xVNode {//顶点表结点
    VertexType data;//顶点信息
    xArcNode *firstin, *firstout;//指向第一条入弧和出弧
} xVNode;
typedef struct {
    xVNode xlist[MaxVertexNum];//邻接表
    int vexnum, arcnum;//图的顶点数和弧数
} GLGraph;

/*图的邻接多重表（无向图）*/
typedef struct mArcNode {//边表结点
    bool mark;//访问标记
    int ivex, jvex;//分别指向该弧的两个结点
    struct mArcNode *ilink, *jlinik;//分别指向两个顶点的下一条边
    //InfoType info;
} mArcNode;
typedef struct mVNode {//顶点表结点
    VertexType data;//顶点信息
    mArcNode *firstedge;//指向第一条依附该顶点的边
} mVNode;
typedef struct {
    mVNode adjmulist[MaxVertexNum];//邻接表
    int vexnum, arcnum;
} AMLGraph;

/*NextNeighbor:邻接矩阵*/
int NextNeighbor(MGragh &G, int x, int y) {
    if (x != -1 && y != -1)
        for (int col = y + 1; col < G.vexnum; col++)
            if (G.Edge[x][col] > 0 && G.Edge[x][col] < maxWeight)
                return col;
    return -1;
}

/*NextNeighbor:邻接表*/
int NextNeighbor(ALGraph &G, int x, int y) {
    if (x != -1) {
        ArcNode *p = G.vertices[x].first;//对应链表第一个顶点
```

```
            while (p != NULL && p->adjvex != y)//寻找邻接顶点 y
                p = p->next;
            if (p != NULL && p->next != NULL)
                return p->next->adjvex;//返回下一个邻接顶点
        }
        return -1;
}

/*邻接表转化成邻接矩阵*/
void Conver(ALGraph &g, int arcs[M][N]) {
        for (int i = 0; i < n; i++) {
            ArcNode p = g.vertices[i].first;//依次遍历各顶点表结点为头的边链表
            while (p != NULL) {//遍历边链表
                arcs[i][p.adjvex] = 1;
                p = p.next;
            }
        }
}

/*广度优先搜索算法*/
bool visited[MaxVertexNum];

void BFSTraverse(Graph G) {
        for (int i = 0; i < G.vexnum; i++) visited[i] = false;
        InitQueue(Q);
        for (int i = 0; i < G.vexnum; i++)//对每个连通分量调用一次 BFS
            if (!visited[i])
                BFS(G, i);
}

void BFS(Graph G, int v) {
        visit(v);
        visited[v] = true;
        EnQueue(Q, v);
        while (!isEmpty(Q)) {
            DeQueue(Q, v);
            for (w = FirstNeighbor(G, v); w >= 0; w = NextNeighbor(G, v, w))
                if (!visited[w]) {
                    visit(w);
                    visited[w] = true;
                    EnQueue(Q, w);
                }
        }
}

/*BFS 求单源最短路径*/
void BFS_MIN_Distance(Graph G, int u) {
        //d[i]表示从 u 到 i 结点的最短路径
        for (i = 0; i < G.vexnum; i++)
            d[i] = INF;
        visited[u] = true;
        d[u] = 0;
        EnQueue(Q, u);
        while (!isEmpty(Q)) {
            DeQueue(Q, u);
            for (w = FirstNeighbor(G, u); w >= 0; w = NextNeighbor(G, u, w))
                if (!visited[w]) {
                    visited[w] = true;
                    d[w] = d[u] + 1;
                    EnQueue(Q, w);
                }
        }
}

/*DFS 深度优先搜索算法*/
void BFSTraverse(Graph G) {
        for (v = 0; v < G.vexnum; ++v) visited[v] = false;
        for (v = 0; v < G.vexnum; ++v)
            if (!visited[v])
```

```
                DFS(G, v);
}

void DFS(Graph G, int v) {
    visit(v);
    visited[v] = true;
    for (w = FirstNeighbor(G, u); w >= 0; w = NextNeighbor(G, u, w))
        if (!visited[w])
            DFS(G, w);
}

/*判断一个无向图是否是一棵树*/
/*G 必须是无回路的连通图或者是有 n-1 条边的连通图*/
/*这里采用后者*/
bool isTree(Graph &G) {
    for (int i = 1; i <= G.vexnum; i++) visited[i] = false;
    int Vnum = 0, Enum = 0;//记录顶点数和边数
    DFS(G, 1, Vnum, Enum, visited);
    if (Vnum == G.vexnum && Enum == 2 * (G.vexnum - 1))
        return true;
    else
        return false;
}

void DFS(Graph &G, int v, int &Vnum, int &Enum, int visited[]) {
    visited[v] = true;
    Vnum++;//顶点计数
    int w = FirstNeighbor(G, v);
    while (w != -1) {//当邻接顶点存在
        Enum++;//边存在，边计数
        if (!visited[w])
            DFS(G, w, Vnum, Enum, visited);
    }
}

/*DFS 非递归算法，采用邻接表*/
void DFS_Non_RC(ALGraph &G, int v) {
    //从顶点 v 开始进行 DFS，一次遍历一个连通分量的所有顶点
    int w;
    InitStack(S);
    for (i = 0; i < G.vexnum; i++) visited[i] = false;
    Push(S, v);
    visited[v] = true;
    while (!IsEmpty(S)) {
        k = Pop(S);
        visit(k);//先访问，再将其子结点入栈
        for (w = FirstNeighbor(G, k); w >= 0; w = NextNeighbor(G, k, w))
            if (!visited[w]) {
                Push(S, w);
                visited[w] = true;
            }
    }
}//由于使用了栈，使得遍历的方式是从右端到左端进行的

/*DFS，邻接表，判断是否存在从 Vi 到 Vj 的路径*/
int visited[MaxSize] = {0};

int Exist_Path_DFS(ALGraph G, int i, int j) {
    int p;//顶点序号
    if (i == j) return 1;
    else {
        visited[i] = 1;
        for (p = FirstNeighbor(G, i); p >= 0; p = NextNeighbor(G, i, p))
            if (!visited[p] && Exist_Path_DFS(G, p, j))
                return 1;
    }
    return 0;
}
```

```
/*BFS，邻接表，判断是否存在从 Vi 到 Vj 的路径*/
int Exist_Path_BFS(ALGraph G, int i, int j) {
	InitQueue(Q);
	EnQueue(Q, i);
	while (!isEmpty(Q)) {
		DeQueue(Q, u);
		visited[u] = 1;
		for (p = FirstNeighbor(G, i); p; p = NextNeighbor(G, i, p)) {
			k = p.adjvex;
			if (k == j) return 1;
			if (!visited[k]) EnQueue(Q, k);
		}
	}
	return 0;
}

/*邻接表，输出从顶点 Vi 到 Vj 的所有简单路径*/
/*采用 DFS
 * 设置 path 数组存放路径上的结点，初始为空
 * d 表示路径长度，初始为-1
 * 设置查找函数名 FindPath()
 * 1)FindPath(G,u,v,path,d): d++; path[d]=u;
 *    若找到 u 的未访问相邻结点 u1，则继续下去，否则 visited[u]=0 并返回
 * 2)FindPath(G,u1,v,path,d): d++; path[d]=u1;
 *    若找到 u1 的未访问相邻结点 u2，则继续下去，否则 visited[u]=0
 * 3)依次类推，继续上述递归过程，直到 Ui=V,输出 path*/
void FindPath(ALGraph *G, int u, int v, int path[], int d) {
	int w, i;
	ArcNode *p;
	d++;//路径长度+1
	path[d] = u;//将当前结点添加到路径当中
	visited[u] = 1;
	if (u == v) print(path[]);
	p = G->vertices[u].first;//p 指向 u 的第一个相邻点
	while (p != NULL) {
		w = p->adjvex;//若顶点 w 未访问，递归访问它
		if (visited[w] == 0)
			FindPath(G, w, v, path, d);
		p = p->next;//p 指向下一个相邻点
	}
	visited[u] = 0;//恢复环境，使该顶点可以重新使用
}

/*拓扑排序算法实现*/
bool TopologicalSort(Graph G) {
	InitStack(S);
	for (int i = 0; i < G.vexnum; i++)
		if (indegree[i] == 0)
			Push(S, i);//将所有入度为 0 的点进栈
	int count = 0;
	while (!IsEmpty(S)) {
		Pop(S, i);
		print[count++] = i;
		for (p = G.vertices[i].firstarc; p; p = p->nextarc) {
			v = p->adjvex;
			if (!(--indegree[v]))
				Push(S, v);
		}
	}
	if (count < G.vexnum) return false;
	else return true;
}

/*DFS 实现有向无环图拓扑排序*/
/*对于有向无环图 G 中的任意结点 u,v 他们之间的关系必然是三种之一：
 * 1)假设 u 是 v 的祖先，则在调用 DFS 访问 u 的过程中，必然会在这个过程结束之前递归的归 v 调用 DFS 访问，也就是说 v 的 DFS 函数结束时间先于 u 的 DFS 结束时间。从而可以考虑再 DFS 调用的过程中设定一个时间标记，再 DFS 调用结束时，对各个结点计时。祖先的结束时间必然大于子孙的结束时间
 * 2)如果 u 是 v 的子孙，则 v 是 u 的祖先，v 的结束时间大于 u 的结束时间
```

```
 * 3)如果 uv 没有关系，则 u，v 在拓扑排序中关系任意
 * 从而按照结束时间从大到小，就可以得到拓扑排序序列*/
/*实际上和深度遍历一样，只不过加入了 time 变量*/
bool visited[MaxVertexNum];

void BFSTraverse(Graph G) {
    for (v = 0; v < G.vexnum; v++)
        visited[v] = false;
    time = 0;
    for (v = 0; v < G.vexnum; v++)
        if (!visited[v])
            DFS(G, v);
}

void DFS(Graph G, int v) {
    visited[v] = true;
    visit(v);
    for (w = FirstNeighbor(G, v); w >= 0; w = NextNeighbor(G, v, w))
        if (!visited[w])
            DFS(G, w);
    time = time + 1;
    finishTime[v] = time;
}

/*Dijkstra 和 Prim 算法比较，基于邻接矩阵 G[N][N]*/
void compare() {
    bool closed[N] = {false};
    int Min[N] = {INF};
//对应 Dijkstra 中的从 start 点出发到其余各点的最短路径或加入 Prim 算法中最小生成树的边。初始化的时候，都为正无穷
    close[start] = true;
    Min[start] = 0;
//表示从 start 点开始执行 Dijkstra 或 Prim 算法
    for (int i = 1; i < N; i++) {
        //执行 N-1 次，即开始链接其余的 N-1 个结点
        //保存尚未求解出的结点中与起点距离最短的结点或者到已球出来的最小生成树中距离最小的那个结点
        int k = -1;
        for (int j = 0; j < N; j++)
            if (!closed[j] && (k == -1 || Min[k] > Min[j]))
                k = j;
        closed[k] = true;
        //得到了 k，这里考虑了图是连通的，所以认为 k 一定存在，不加判定条件
        for (int j = 0; j < N; j++) {
            //Dijkstra 算法对应的更新 Min 算法
            if (Min[j] > Min[k] + G[k][j])
                Min[j] = Min[k] + G[k][j];
            //Prim 算法对应的更新 Min 算法
            if (Min[j] > G[k][j])
                Min[j] = G[k][j];
        }


    }
}
```