图

图是一种复杂的非线性结构，线性结构和树结构也可以看成简单的图结构。在非空线性表中，除开始结点和终端结点以外，任意结点都有唯一直接前驱和唯一直接后继，是一种“一对一”的线性关系；在树结构中，结点间具有“一对多”的层次关系，除根结点没有双亲外，每个结点只有一个双亲（直接前驱），除叶结点没有孩子外，每个结点都可以有多个孩子（直接后继）。在图结构中，结点之间的关系是任意的，是一种“多对多”的关系，即每个结点可以有零个或多个直接前驱、零个或多个直接后继。

图的概念

图结构被用于描述各种复杂的数据关系，其概念和术语说明如下：

1. 图(Graph)：由定点的非空集合V和边或弧的集合E组成，表示为$G=(V,E)$。以后将用$V(G)$和$E(G)$分别表示图G的顶点集和边集。若$E(G)$为空，则图G只有顶点没有边。
2. 无向图(Undirected Graph)：若图中顶点对是无序的，即边是无方向的，边集$E(G)$为无向边的集合，则图G称为无向图。在无向图中，以无序对(u,v)表示u和v之间存在一条无向边。在无向图中，边是对称的，(u,v)和(v,u)表示同一条边。
3. 有向图(Directed Graph)：若图中顶点是有序的，即边是有方向的，边集$E(G)$为有向边的集合，则图G称为有向图。在有向图中，一般将边称为弧(arc)，以有序对$\langle u,v\rangle$表示一条从顶点u出发到达顶点v的弧，其中u称为弧尾或起点，v称为弧头或重点。在有向图中，$\langle u,v\rangle$和$\langle v,u\rangle$是不一样的。
4. 无向完全图(Undirected Complete Graph)：在一个无向图中，如果任意两个顶点都有一条边直接相连，则称该图为无向完全图。具有n个顶点的无向图，边数的取值范围是$[0,\frac{n(n-1)}{2}]$。无向完全图具有$\frac{n(n-1)}{2}$条边。
5. 有向完全图(Directed Complete Graph)：在一个有向图中，如果任意一个顶点都有一条弧直接到达其他顶点，则称该图为有向完全图。具有n个顶点的有向图，边数的取值范围是$[0,n(n-1)]$。有向完全图具有$n(n-1)$条弧。
6. 子图(Subgraph)：假设有两个图G和G'，满足条件$V(G')$是$V(G)$的子集，$V(G')\subseteq V(G)$，且$E(G')$是$E(G)$的子集，$E(G')\subseteq E(G)$，则称G'是G的子图。
7. 权值(Weight)：有时，边或者弧上具有与之相关的某种含义的数值，称为权值。
8. 加权图(Weighted Graph)：若图中的边上带有权值，则称该图为加权图，或称为网。
9. 稀疏图与稠密图：有很少的边或弧（如$e\leq n\log n$）的图称为稀疏图，反之称为稠密图。
10. 度(Degree)：图G中，与顶点v相关联的边的数目，称为顶点v的度。
    - 在无向图中，顶点v的度是指与该顶点相关联的边的数目，记为$TD(v)$。
    - 在有向图中，弧$\langle u,v\rangle$和顶点u,v相关联，顶点v的度分为入度和出度。
      - 出度(Outdegree)：以v为起点的弧的数目称为v的出度，记为$OD(v)$。
      - 入度(Indegree)：以v为终点的弧的数目称为v的入度，记为$ID(v)$。
      - 在有向图中，顶点v的度是其出度与入度之和：$TD(v)=OD(v)+ID(v)$。

    一个具有n个顶点，e条边或弧的图，顶点的度之和是边数的2倍：

    $$e=\frac{1}{2}\sum_{i=1}^{n}TD(v_i)$$

11. 路径(Path)：在无向图G中，若存在一个从顶点v_1到v_n的顶点序列$v_1,v_2,\ldots,v_n$，且满足$(v_i,v_{i+1})\in E\ (1\leq i<n)$，则称从顶点$v_1$到顶点$v_n$存在一条路径。如果$G$是有向图，则路径也是有向的，顶点序列满足$\langle v_i,v_{i+1}\rangle\in E\ (1\leq i<n)$。

12. 路径长度(Path Length)：是指该路径上经过的边或弧的数目。对于加权图，路径长度是指该路径中各边权值之和。
13. 回路或环(Cycle)：若一条路径上的第一个顶点和最后一个顶点相同，则该路径称为回路或环。
14. 简单路径：若一条路径上所有顶点均不重复，则该路径称为简单路径。
15. 简单回路或简单环：处第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路，称为简单回路或简单环。
16. 连通、连通图和连通分量：在无向图G中，若从u到v $(u \neq v)$存在路径，则称u到v是连通的。若$V(G)$中的每对不同顶点u和v都连通，则称G是连通图。无向图G中的极大连通子图称为图G的连通分量。连通分量可以是连通图自身。
17. 强连通图和强连通分量：在有向图G中，若对于$V(G)$中的每对不同的顶点u, v $(u \neq v)$，都存在从u到v以及从v到u的路径，则称G是强连通图。有向图G中极大的强连通子图称为图G的强连通分量。强连通分量可以是强连通图自身。
18. 生成树：是连通图的极小连通子图，它含有图中全部n个顶点，但只有足以构成一棵树的$n-1$条边。在生成树中添加一条边之后，必然会形成回路或环。
19. 有向树：只有一个顶点的入度为0，其余顶点的入度为1的有向图，称为有向树。有向树是弱连通图。将有向图的所有有向边替换为无向边，所得到的图称为原图的基图。如果一个有向图的基图是连通图，则该有向图是弱连通图。

图的抽象数据类型定义如下：

```
template <class VertexType, class EdgeType>
class Graph{
    struct mstEdge{ // 最小生成树的边结点类型
        int vex1, vex2;
        EdgeType weight; // 边的三元组:(起点,终点, 权重)
        bool operator<(const mstEdge& e) const {
            return weight < e.weight;
        }
    } *TE; // 最小生成树的边集
    int verNum, edgeNum; // 图的顶点数和边数
    bool *visited; // 访问标记数组
public:
    int numOfVertex() const {return verNum;} // 返回图的顶点数
    int numOfEdge() const {return edgeNum;} // 返回图的边数
    virtual void createGraph(const VertexType V[], const EdgeType E[]) = 0; // 创建图
    virtual void dfsTraverse() const = 0; // 深度优先遍历
    virtual void bfsTraverse() const = 0; // 广度优先遍历
    virtual void topSort() const = 0; // 拓扑排序
    virtual void prim(EdgeType noEdge) = 0; // Prim算法求最小生成树
    virtual void kruskal() const = 0; // Kruskal算法求最小生成树
    virtual void printMst() const = 0; // 打印最小生成树
    virtual bool searchEdge(int from, int to) const = 0; // 查找边
    virtual bool insertEdge(int from, int to, EdgeType w) = 0; // 插入权值为w的边
    virtual bool removeEdge(int from, int to) = 0; // 删除边
    virtual void printGraph() const = 0; // 打印图
protected:
    struct mstEdge{
        int vex1, vex2;
        EdgeType weight; // 边的三元组:(起点,终点, 权重)
        bool operator<(const mstEdge& e) const { // 使用优先级队列需要重载<运算符
            return weight < e.weight;
        }
    } *TE; // 最小生成树的边集
    int verNum, edgeNum; // 图的顶点数和边数
    bool *visited; // 访问标记数组
```

```
};
```

图的存储结构

图的存储结构除了要存储图中各个顶点本身的数据信息外，同时还要存储顶点间的关系，即边的信息。图中任意两个顶点之间都可能存在关系，因此图的结构比较复杂。图的常用存储结构有邻接矩阵、邻接表、十字链表和邻接多重表。

邻接矩阵

图的邻接矩阵表示法

设图$G=(V,E)$含有n $(n\geq 1)$个顶点，需要存放n个顶点信息及n^2个边或弧的信息。图的邻接矩阵(Adjacency Matrix)表示法：用$n\times n$的二维数组（矩阵）来表示顶点间的邻接关系，用一维数组存储图的顶点数据信息。图G的邻接矩阵是具有如下性质的n阶方阵：

$$A[i][j]=\begin{cases}1 & (v_i,v_j)\text{或}\langle v_i,v_j\rangle\in E(G)\\ 0 & (v_i,v_j)\text{或}\langle v_i,v_j\rangle\notin E(G)\end{cases}$$

1. 无向图的邻接矩阵表示法的特点如下：
    1. 无向图的邻接矩阵一定是对称矩阵，因此，含有n个顶点的无向图，可以只存储其下三角或上三角的元素，将其压缩到大小是$\frac{n(n-1)}{2}$的空间内。
    2. 第i行（或第i列）非零元素个数，表示的是顶点v_i的度。
2. 有向图的邻接矩阵表示法的特点如下：
    1. 有向图的邻接矩阵不一定是对称的。
    2. 第i行非零元素个数，表示的是顶点v_i的出度。
    3. 第i列非零元素个数，表示的是顶点v_i的入度。
3. 网的邻接矩阵

    若图G是带权图（网），则邻接矩阵可以定义为：

$$A[i][j]=\begin{cases}w_{i,j} & (v_i,v_j)\text{或}\langle v_i,v_j\rangle\in E(G)\\ \infty & (v_i,v_j)\text{或}\langle v_i,v_j\rangle\notin E(G)\end{cases}$$

    其中$w_{i,j}$表示边上的权值

邻接矩阵表示法的特点

1. 图的邻接矩阵表示是唯一的。
2. 含有n个顶点的图，其邻接矩阵的空间代价都是$O(n^2)$，与图的顶点数相关，与边数无关。
3. 当邻接矩阵是稀疏矩阵时，可以采用三元组表的形式存储。

图的邻接矩阵表示法的定义与实现

图中顶点信息和顶点之间关系（边或弧）分别用一维数组`vertexs`和二维数组`edges`存储，其形式描述如下：

```
template <class VertexType, class EdgeType>
class adjMatrix : public Graph<VergexType, EdgeType> {
    VertexType* vertexs; // 顶点向量
    EdgeType** edges; // 邻接矩阵
    EdgeType noEdge; // 无边标志
    void dfs(int start) const; // 从start顶点开始深度优先遍历
```

```
public:
    adjMatrix(int size, EdgeType noEdgeFlag);
    ~adjMatrix(); // 析构函数
    void createGraph(const VertexType V[], const EdgeType E[]); // 创建图
    void printGraph() const; // 打印图
    bool searchEdge(int from, int to) const; // 查找边
    bool insertEdge(int from, int to, EdgeType w); // 插入边
    bool removeEdge(int from, int to); // 删除边
    void dfsTraverse() const; // 深度优先遍历
    void bfsTraverse() const; // 广度优先遍历
    void topSort() const; // 拓扑排序
    void prim(EdgeType noEdge); // Prim算法求最小生成树
    void kruskal() const; // Kruskal算法求最小生成树
    void printMst() const; // 打印最小生成树
    void floyd() const; // Floyd算法求最短路径
    void printFloyed(EdgeType** D, int** pre) const; // 打印Floyd算法的结果
};
```

构造函数：

```
template <class VertexType, class EdgeType>
adjMatrix<VertexType, EdgeType>::adjMatrix(int size, EdgeType noEdgeFlag){
    verNum = size;
    edgeNum = 0;
    noEdge = noEdgeFlag;
    vertex = new VertexType[verNum];
    edges = new EdgeType*[verNum];
    for(int i = 0; i < verNum; i++){
        edges[i] = new EdgeType[verNum];
        for(int j = 0; j < verNum; j++){
            edge[i][j] = noEdge; // 邻接矩阵初始值
        }
    }
    visited = new bool[verNum]; // 访问标志数组
    TE = new mstEdge[verNum - 1]; // 最小生成树的边集
}
```

析构函数：

```
template <class VertexType, class EdgeType>
adjMatrix<VertexType, EdgeType>::~adjMatrix(){
    delete [] vertexs; // 删除顶点向量
    for(int i = 0; i < verNum; i++){
        delete [] edges[i];
    }
    delete [] edges; // 删除邻接矩阵
    delete [] visited; // 删除访问标志数组
    delete [] TE; // 删除最小生成树的边集
}
```

查找图中是否存在从`from`到`to`的权值为`w`的边，其中`from`和`to`是顶点在`vertexs`数组中的下标：

```
template <class VertexType, class EdgeType>
bool adjMatrix<VertexType, EdgeType>::searchEdge(int from, int to) const {
    if(from < 0 || from >= verNum || to < 0 || to >= verNum) {
        cout << "Invalid vertex index." << endl;
        return false;
    }
    return edges[from][to] != noEdge; // 返回是否存在边
}
```

在图中插入从from到to的边，其中from和to是顶点在vertexs数组中的下标。

算法分析：

1. 当该边已经存在且权值为w时，返回false。
2. 当该边不存在时，置该边的权值为w，边数计数器增大，返回true。
3. 当改变已经存在且权值不等于w时，更新边的权值为w，返回true。

```
template <class VertexType, class EdgeType>
bool adjMatrix<VertexType, EdgeType>::insertEdge(int from, int to, EdgeType w){
    if(from < 0 || from >= verNum || to < 0 || to >= verNum) {
        cout << "Invalid vertex index." << endl;
        return false;
    }
    if(edges[from][to] == noEdge) {
        edges[from][to] = w; // 插入边
        ++edgeNum;
        return true;
    } else {
        cout << "Edge already exists." << endl;
        return false; // 边已存在
    }
}
```

删除从from到to的边，其中from和to是顶点在vertexs数组中的下标：

```
template <class VertexType, class EdgeType>
bool adjMatrix<VertexType, EdgeType>::removeEdge(int from, int to){
    if(from < 0 || from >= verNum || to < 0 || to >= verNum) {
        cout << "Invalid vertex index." << endl;
        return false;
    }
    if(edges[from][to] != noEdge) {
        edges[from][to] = noEdge; // 删除边
        --edgeNum;
        return true;
    } else {
        cout << "Edge does not exist." << endl;
        return false; // 边不存在
    }
}
```

创建图，其中V为顶点数组，E为经过降维的邻接矩阵：

```
template <class VertexType, class EdgeType>
```

```
void adjMatrix<VertexType, EdgeType>::createGraph(const VertexType V[], const EdgeType E[]){
    int i, j;
    for(i = 0; i < verNum; i++){
        vertexs[j] = V[i]; // 初始化顶点
    }
    for(i = 0; i < verNum; i++){
        for(j = 0; j < verNum; j++){
            if(E[i * verNum + j] > 0){
                insertEdge(i, j, E[i * verNum + j]); // 插入边
            }
        }
    }
}
```

输出图：

```
template <class VertexType, class EdgeType>
void adjMatrix<VertexType, EdgeType>::printGraph() const {
    int i, j;
    for(i = 0; i < verNum; i++){
        cout << vertexs[i] << ":";
        for(j = 0; j < verNum; j++){
            cout << edges[i][j] << " "; // 打印邻接矩阵
        }
    cout << endl;
    }
}
```

邻接表

用邻接矩阵表示图的方法，所占存储空间只与顶点的个数有关，与边的个数无关，在存储某些稀疏图或无根树时会造成空间的浪费。当图上的两个顶点之间存在不止一条边时，用邻接矩阵无法同时表示两个顶点间两条以上的边。

图的邻接表表示法

邻接表(Adjacency List)：这是将图的顶点的顺序存储结构和各顶点的邻接点的链式存储结构相结合的存储方式，类似于树的孩子链表表示法。

边表：在邻接表表示法中，为图中每个顶点建立一个单链表，每个单链表上附设一个头结点。对于无向图，第i个链表将图中与顶点v_i有邻接关系的所有顶点连接起来，也就是说，第i个边表中结点的个数等于顶点v_i的度。对于有向图，第i个链表链接了以顶点v_i为弧尾（起点）的所有弧头（终点）顶点，也就是说，第i个边表中结点的个数等于顶点v_i的出度。

图的边表结点：邻接点域`to`表示与顶点v_i相邻接的顶点在顶点向量中的序号（下标）；指针域`next`指向与顶点v_i相邻接的下一个顶点的边表结点。带权图（网）的边表结点增加了数据域`weight`表示边上的权值等信息。

顶点表：每个链表设立一个头结点，头结点有两个域，数据域`vertex`存储顶点v_i的数据信息，指针域`firstEdge`指向顶点v_i的第一个临界点。为了便于运算，将各个顶点的头结点以顺序存储结构或链式存储结构存储。

邻接表表示法的特点

1. 图的邻接表表示不唯一。
2. 对于无向图，顶点v_i的度为第i个链表中的结点个数。
3. 对于有向图，顶点v_i的出度为第i个链表中的结点个数，若要求顶点v_i的入度，则需遍历整个邻接表。

4. 存储含有n个结点e条边的无向图，需用n个顶点结点和$2e$个边表结点：存储含有n个结点e条边的有向图，需用n个顶点结点和e个边表结点。因此，邻接表的空间代价是$O(n+e)$。
5. 在边稀疏($e \ll \frac{n(n-1)}{2}$)的情况下，用邻接表表示图比用邻接矩阵节约存储空间。
6. 在邻接表上容易找到任意顶点的第一个邻接点和下一个邻接点，但要判定任意两个结点(v_i, v_j)之间是否有边或弧相连，则需要遍历第i个或第j个链表，在这方面不如邻接矩阵方便。

图的邻接表表示法的定义与实现

一个图的邻接表存储结构可描述如下，其中edgeNode为边表结点类型，verNode为顶点结点类型：

```
template <class VertexType, class EdgeType>
class adjList : public Graph<VertexType, EdgeType>{
    struct edgeNode{ // 边表结点类型
        int to; // 边的终点编号(在顶点表中的下标)
        EdgeType weight; // 边的权重
        edgeNode* next; // 指向下一条边表结点
        edgeNode(){}
        edgeNode(int t, EdgeType w, edgeNode* n = nullptr){
            to = t;
            weight = w;
            next = n;
        }
    };
    struct verNode{ // 顶点结点类型
        VertexType vertex; // 顶点信息
        edgeNode* firstEdge; // 指向第一个临界点的指针
        verNode(edgeNode* h = nullptr){
            firstEdge = h; // 初始化顶点结点
        }
    };
    verNode* verList; // 顶点表
    int* topOrder; // 保存拓扑排序，用于求关键路径
    void dfs(int start) const; // 从start号顶点出发深度优先遍历图
public:
    adjList(int size);
    ~adjList();
    void createGraph(const VertexType V[], const EdgeType E[]); // 创建图
    void printGraph() const; // 打印图
    bool searchEdge(int from, int to) const; // 查找边
    bool insertEdge(int from, int to, EdgeType w); // 插入边
    bool removeEdge(int from, int to); // 删除边
    void dfsTraverse() const; // 深度优先遍历
    void bfsTraverse() const; // 广度优先遍历
    void topSort() const; // 拓扑排序
    void prim(EdgeType noEdge); // Prim算法求最小生成树
    void kruskal() const; // Kruskal算法求最小生成树
    void printMst() const; // 打印最小生成树
    bool criticalPath() const; // 关键路径算法
    bool dijkastra(int start, EdgeType noEdge) const; // 求单源点最短路径
    bool printDijPath(int from, int to, int pre[]) const; // 输出源点到其他顶点的最短路径
};
```

构造函数：

```
template <class VertexType, class EdgeType>
adjList<VertexType, EdgeType>::adjList(int size){
    verNum = size;
    edgeNum = 0;
    verList = new verNode[verNum]; // 初始化顶点表
    topOrder = new int[verNum]; // 初始化拓扑排序数组
    visited = new bool[verNum]; // 访问标志数组
    TE = new mstEdge[verNum - 1]; // 最小生成树的边集
}
```

析构函数：

```
template <class VertexType, class EdgeType>
adjList<VertexType, EdgeType>::~adjList(){
    for(int i = 0; i < verNum; i++){
        edgeNode* p = verList[i].firstEdge;
        while(p != nullptr){
            edgeNode* temp = p;
            p = p->next;
            delete temp; // 删除边表结点
        }
    }
    delete [] verList; // 删除顶点表
    delete [] topOrder; // 删除拓扑排序数组
    delete [] visited; // 删除访问标志数组
    delete [] TE; // 删除最小生成树的边集
}
```

查找图中是否存在从from到to的权值为w的边，其中from和to是顶点在vertexs数组中的下标：

```
template <class VertexType, class EdgeType>
bool adjList<VertexType, EdgeType>::searchEdge(int from, int to) const {
    if(from < 0 || from >= verNum || to < 0 || to >= verNum){
        cout << "Invalid vertex index." << endl;
        return false;
    }
    edgeNode* p = verList[from].firstEdge; // 从from顶点的边表开始查找
    while(p != nullptr && p->to != to){
        p = p->next; // 遍历边表
    }
    if(p == nullptr) return false; // 没有找到边
    return true; // 找到边
}
```

在图中插入从from到to的边，其中from和to是顶点在vertexs数组中的下标。

算法分析：由于每个顶点的（边表）单链表中均无头结点，因此插入边表结点时要对首元结点单独处理。插入边可分为三种情况。

1. 当该边已经存在且权值为w时，返回false。
2. 当该边不存在时，置该边的权值为w，边数计数器增大，返回true。
3. 当改变已经存在且权值不等于w时，更新边的权值为w，返回true。

```
template <class VertexType, class EdgeType>
bool adjList<VertexType, EdgeType>::insertEdge(int from, int to, EdgeType w){
    if(from < 0 || from >= verNum || to < 0 || to >= verNum){
        cout << "Invalid vertex index." << endl;
        return false;
    }
    edgeNode *p = verList[from].firstEdge, *pre, *s; // 从from顶点的边表开始查找
    while(p != nullptr && p->to != to){
        pre = p; // 记录前一个结点
        p = p->next; // 遍历边表
    }
    if(p != nullptr && p->to == to){
        if(p->weight != w) p->weight = w; // 如果边已存在，更新权重
        else return false;
    }else{
        s = new edgeNode(to, w, p);
        if(p == verList[from].firstEdge)
            verList[from].firstEdge = s; // 插入到边表头部
        else
            pre->next = s; // 在链表其他位置上插入结点
        ++edgeNum; // 新增一条边，边数加一
    }
    return false;
}
```

删除从from到to的边，其中from和to是顶点在vertexs数组中的下标。同样要注意，由于每个结点的单链表中均无头结点，因此删除边表结点时要对首元结点单独处理：

```
template <class VertexType, class EdgeType>
bool adjList<VertexType, EdgeType>::removeEdge(int from, int to){
    if(from < 0 || from >= verNum || to < 0 || to >= verNum){
        cout << "Invalid vertex index." << endl;
        return false;
    }
    edgeNode *p = verList[from].firstEdge, *pre = nullptr;
    while(p != nullptr && p->to < to){
        pre = p;
        p = p->next;
    }
    if((p == nullptr) || (p->to > to))
        return false;
    if(p->to == to){
        if(p == verList[from].firstEdge)
            verList[from].firstEdge = p->next;
        else pre->next = p->next;
        delete p;
        --edgeNum;
        return true;
    }
}
```

创建图，其中V为顶点数组，E为经过降维的邻接矩阵：

```
template <class VertexType, class EdgeType>
void adjList<VertexType, EdgeType>::createGraph(const VertexType V[], const EdgeType E[]){
    int i, j;
    for(i = 0; i < verNum; i++)
        verList[i].vertex = V[i];
    for(i = 0; i < verNum; i++)
        for(int j = 0; j < verNum; j++){
            if(E[i * verNum + j] > 0)
                insertEdge(i, j, E[i * verNum + j]); // insertEdge插入边按to值排序
        }
}
```

输出图：

```
template <class VertexType, class EdgeType>
void adjList<VertexType, EdgeType>::printGraph() const {
    int i;
    for(i = 0; i < verNum; i++){
        cout << verList[i].vertex << ":";
        edgeNode *p = verList[i].firstEdge;
        while(p != nullptr){
            cout << verList[p->to].vertex << ' ';
            p = p->next;
        }
        cout << endl;
    }
}
```

图的遍历

图的遍历：对于给定图$G=(V,E)$，从顶点v $(v\in V(G))$出发，按照某种次序访问G中的所有顶点，使得每个顶点被访问一次且仅被访问一次，这一过程称为图的遍历。通常有两种遍历图的方法，深度优先遍历和广度优先遍历，它们对有向图和无向图都一样适用。相比于树的遍历过程，图的遍历过程要复杂得多，主要表现在以下两个方面：

1. 图中任意一个顶点都可能和其余顶点相邻接，因此在访问了某个顶点之后，在后面的访问过程中，可能再次返回该顶点，即图中可能存在回路。
2. 对于非连通图，从某个顶点出发，不能到达图中的所有顶点。

为了避免同一个顶点被多次访问，通常对已经访问的顶点加标记。对于含有n个顶点的图，我们用大小为n的访问标志数组`visited`来标志顶点是否被访问过，其初始值为`false`，表示未访问。在遍历的过程中，当顶点v_i被访问时，置`visited[i] = true`，表示顶点v_i已经被访问。

深度优先遍历

深度优先遍历(Depth First Search, DFS)：又称为深度优先搜索，类似于树的前序遍历，尽可能先对纵深方向进行搜索。其遍历过程如下：

1. 首先选定一个未被访问的顶点v，访问此顶点并加上已访问标志。
2. 然后依次从顶点v的未被访问的邻接点出发深度优先遍历图。
3. 重复上述过程直到图中所有和顶点v有路径相通的顶点都被访问到。

4. 如果还有顶点没有被访问到（非连通图），则再选取其他未被访问的顶点，重复以上遍历过程，直到访问完所有顶点位置。

基于邻接表的深度优先遍历算法如下：

深度优先遍历图的公共接口函数：

```
template <class VertexType, class EdgeType>
void adjList<VertexType, EdgeType>::dfsTraverse() const {
    int i, count = 0;
    for(i = 0; i < verNum; i++)
        visited[i] = false; // 初始化访问标志数组
    for(i = 0; i < verNum; i++){
        if(visited[i] == false){
            dfs(i); // 从未访问的顶点开始深度优先遍历
            count++;
        }
    }
    cout << endl;
}
```

私有递归函数dfs：访问从顶点start出发能够深度优先遍历到的所有顶点：

```
template <class VertexType, class EdgeType>
void adjList<VertexType, EdgeType>::dfs(int start) const {
    visited[start] = true; // 标记当前顶点为已访问
    cout << verList[start].vertex << " "; // 输出当前顶点
    edgeNode* p = verList[start].firstEdge; // 获取当前顶点的边表
    while(p != nullptr){ // 遍历边表
        if(!visited[p->to]){ // 如果边的终点未被访问
            dfs(p->to); // 递归访问
        }
        p = p->next; // 移动到下一条边
    }
}
```

假设图G有n个顶点和e条边，深度优先遍历算法将对图中所有的顶点和边进行访问，因此它的时间代价和顶点数n和边数e是相关的，算法的时间复杂度为$O(n+e)$。如果以邻接矩阵作为图的存储结构，则算法的时间复杂度为$O(n^2)$。

基于邻接矩阵的深度优先遍历算法如下：

由于深度优先遍历图的公共接口函数dfsTraverse与邻接表的dfsTraverse代码实现一致，因此不再赘述。

私有递归函数dfs：访问从顶点start出发能够深度优先遍历到的所有顶点：

```
template <class VertexType, class EdgeType>
void adjMatrix<VertexType, EdgeType>::dfs(int start) const {
    visited[start] = true; // 标记当前顶点为已访问
    cout << vertexs[start] << " "; // 输出当前顶点
    for(int i = 0; i < verNum; i++) {
        if(edges[start][i] != noEdge && visited[i] == false) { // 如果存在边且未访问
            dfs(i); // 递归访问
        }
    }
}
```

广度优先遍历

广度优先遍历(Vreadth First Search, BFS)：又称为广度优先搜索，类似于树的层次遍历，图的广度优先遍历的过程如下：

1. 首先选定一个未被访问的顶点v，访问此顶点并加上已访问标志。
2. 依次访问与顶点v相邻接的未被访问的全部邻接点，然后从这些访问过的邻接点出发依次访问它们各自的未被访问的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问。
3. 重复上述过程，直至图中所有与顶点v有路径相连的顶点都被访问到。
4. 若图中还有其他顶点未访问到，则任选其中一个作为源点，继续进行广度优先搜索，直到访问完所有的顶点为止。

那么，怎样才能实现步骤2中所指定的结点访问顺序呢？除了需要设置访问标志数组，还需要借助队列存放顶点的邻接点。此时广度优先遍历算法可以描述如下：

1. 初始化一个队列。
2. 遍历从某个未被访问过的顶点开始，访问这个顶点并加上已访问标记，然后将该顶点入队。
3. 在队列不空情况下，反复进行以下操作：队头元素出队，访问该元素的所有未被访问的邻接点并加上已访问标志，然后将这些邻接点依次入队，这一操作一直进行到队列为空为止。
4. 如果图中还有未被访问的顶点，说明图不是连通图，则再选择任意一个未被访问过的顶点，重复上述过程，直到所有顶点都被访问到为止。

基于邻接表的广度优先遍历算法如下：

```
template <class VertexType, class EdgeType>
void adjList<VertexType, EdgeType>::bfsTraverse() const {
    int v, i, count = 0;
    queue<int> Q;
    edgeNode* p;
    for(i = 0; i < verNum; i++) visited[i] = false; // 初始化访问标志数组
    for(i = 0; i < verNum; i++){
        if(visited[i] == true) continue;
        cout << verList[i].vertex << " "; // 输出当前顶点
        visited[i] = true; // 标记当前顶点为已访问
        Q.push(i); // 将当前顶点入队
        count++;
        while(!Q.empty()){
            v = Q.front(); // 获取队首元素
            Q.pop(); // 出队
            p = verList[v].firstEdge; // 获取当前顶点的边表
            while(p != nullptr){ // 遍历边表
                if(visited[p->to] == false){ // 如果边的终点未被访问
                    cout << verList[p->to].vertex << " "; // 输出边的终点
                    visited[p->to] = true; // 标记为已访问
                    Q.push(p->to); // 将边的终点入队
                }
                p = p->next; // 移动到下一条边
            }
        }
    }
    cout << endl;
}
```

基于邻接矩阵的广度优先遍历算法如下：

```cpp
template <class VertexType, class EdgeType>
void adjMatrix<VertexType, EdgeType>::bfsTraverse() const {
    int v, i, j, count = 0;
    queue<int> Q;
    for(i = 0; i < verNum; i++) visited[i] = false; // 初始化访问标志数组
    for(i = 0; i < verNum; i++){
        if(visited[i] == true) continue;
        cout << vertexs[i] << ' ';
        visited[i] = true; // 标记当前顶点为已访问
        Q.push(i); // 将当前顶点入队
        count++;
        while(!Q.empty()){
            v = Q.front(); // 获取队首元素
            Q.pop(); // 出队
            if(visited[v] == false){
                for(j = 0; j < verNum; j++){ // 查找顶点v未被访问的邻接点
                    if(edges[v][j] != noEdge && visited[j] == false){
                        cout << vertexs[j] << ' '; // 输出当前顶点
                        visited[j] = true; // 标记为已访问
                        Q.push(j); // 将顶点入队
                    }
                }
            }
        }
    }
    cout << endl;
}
```

图的连通分量和生成树

图的连通性问题实际上是图的遍历的一种应用，我们可以利用图的遍历算法求图的连通分量以及连通分量的个数，计数器count就用来统计图的连通分量的个数。

生成树

一个连通图的生成树是一个极小的连通子图，它含有图中全部n个顶点，但只有足以构成一棵树的$n-1$条边。在生成树中添加一条边之后，必然会形成回路或环。一般来着，一个连通图的生成树并不是唯一的，除非原图本身是一棵树。

若无向图G是连通图，对其遍历时就可以从图中任意一个顶点出发，进行深度优先遍历或广度优先遍历，就能访问到图中的所有顶点。在遍历过程中，如果将每次前进途中路过的结点和边记录下来，就得到一个子图，该子图为以源点为根的生成树。采用DFS算法遍历图所得到的生成树称为深度优先生成树；采用BFS算法遍历图所得到的生成树称为广度优先生成树。

生成森林

若无向图G是非连通图，则从图中某个顶点出发遍历图，不能访问到该图的所有顶点，而需要依次对图中的每个连通分量进行深度优先遍历或广度优先遍历，即需要从多个顶点出发进行DFS或BFS。在遍历过程中，如果将每次前进途中路过的结点和边记录下来，就得到多棵树，从而构成森林。采用DFS算法遍历图所得到的森林称为深度优先生成森林；采用BFS算法遍历图所得到的森林称为广度优先生成森林。