图

图的数据结构(邻接矩阵)

```
typedef struct GNode *PtrToGNode;
struct GNode{
    int Nv;  /* 顶点数 */
    int Ne;  /* 边数   */
    VertexType vex[Maxsize]; /* 顶点表*/
    EdgeType G[MaxVertexNum][MaxVertexNum]; /* 邻接矩阵, 边表*/
};
typedef PtrToGNode MGraph; /* 以邻接矩阵存储的图类型 */
```

深度优先遍历（递归）

```
void DFS(MGraph graph, int v){
    visited[v] = true;
    visit(v);
    for(int i = 0; i < graph->Nv; ++i){
        if(graph->G[v][i] == 1 && !visited[vex[i]])
            DFS(graph, vex[i]);
    }
}
```

深度优先遍历（迭代）

```
void DFS(MGraph graph, int v){
    stack<int> st;
    visit(v);
    visited[v];
    st.push(v);
    while(!st.empty()){
        int data, i;
        data = st.top();
        for(i = 0; i < graph->Nv; ++i){
            if(graph->G[data][i] == 1 && visited[vex[i]] == 1){
                visit(vex[i]);
                visited[vex[i]] = true;
                st.push(v);
                break;
            }
        }
        if(i == graph->Nv) st.pop();
    }
}
```

图的广度优先遍历(迭代)

```
void BFS(MGraph graph, int v){
    queue<int> que;
    int vertex;
    visit(v);
    visited[v] = true;
    que.push(v);
    while(!que.empty()){
        vertex = que.front();
        que.pop();
        for(int i = 0; i < graph->Nv; ++i){
            if(graph->G[v][i] == 1 && !visited[vex[i]]){
                visit(vex[i]);
                visited[vex[i]] = true;
                que.push(vex[i]);
            }
        }
    }
}
```

图的数据结构描述(邻接表)

```
typedef struct ArcNode{      // 边表结点
    int adjvex;              // 该弧所指向的顶点的位置
    struct ArcNode *nextArc;    // 指向下一条弧的指针
}ArcNode;

typedef struct VNode{        // 顶点表结点
    int data;                // 顶点信息
    ArcNode* firstArc;          // 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;              // 顶点数目
    int arcnum;              // 边数目
}ALGraph;
```

深度优先遍历(递归)

```
vector<bool> visited;
void DFS(ALGraph graph, int v){
    visited[v] = true;
    ArcNode* p;
    visit(graph.vertices[v].data);
    p = G.verties[v].firstArc;
    while(p){
        if(!visited[p->adjvex])
            DFS(graph, p->adjvex);
        p = p->nextArc;
    }
}
```

深度优先遍历(迭代)

```
void DFSTraverse(Graph graph,int v){ //图的非递归深度优先遍历
    int i,visited[MaxSize],top;
    ArcNode *stack[MaxSize],*p;
    for(i = 0; i < graph.vexnum; i++){     //将所有顶点都添加未访问标志0
        visited[i] = 0;
    }
    printf("%4c",graph.vertices[v].data); //访问顶点v并将访问标志置为1
    visited[v] = 1;

    top = -1; //初始化栈
    p = graph.vertices[v].firstArc;     //p指向顶点v的第一个邻接点
    while(top > -1 || p != NULL){
        while(p!=NULL){
            if(visited[p->adjvex] == 1){
                p = p->nextarc;
            }else{
                printf("%4c",graph.vertices[p->adjvex].data);
                visited[p->adjvex]=1;
                stack[++top] = p;
                p = graph.vertices[p->adjvex].firstArc;
            }
        }
        if(top > -1){
            p = stack[top--];
            p = p->nextArc;
        }
    }
}
```

广度优先遍历

```
// 图的广度优先遍历
int visited[maxsize];
void BFS(ALGraph *G, int v){
    ArcNode* p;
    int que[maxsize], front = 0, rear = 0;
    int j;
    Visit(v);
    visited[v] = 1;
    rear = (rear + 1) % maxsize;
    que[rear] = v;
    while (front != rear)
    {
        front = (front + 1) % maxsize;
        j = que[front];
        p = G->adjlist[j].firstArc;
        while(p){
            if(visited[p->adjvex] == 0){
                Visit(p->adjvex);
                visited[p->adjvex] = 1;
                rear = (rear + 1) % maxsize;
                que[rear] = p->adjvex;
            }
            p = p->nextarc;
        }
    }
}
```

拓扑排序

```
bool TopologicalSort(Graph G){
    //若G存在拓扑排序，返回true， 否则返回false
    InitStack(S);
    for(int i=0; i<G.vexnum; ++i){
        if(indegree[i] == 0)
            S.push(i);
        int count = 0;
        while(!IsEmpty(S)){
            Pop(S,i);
            print[count++] = i;
            for(p = G.vertices[i].firstarc; p; p = p->next){
                //将所有i所指向的顶点的入度减1， 并且将入度减为0的顶点压入栈S
                    v = p->adjvex;
                    if(!(--indegree[v]))
                        S.push(v);      // 入度为0 则入栈
            }//for
        }//while
        if(count < G.vexnum)     // 拓扑排序失败 有回路
            return false;
        else
            return true;
    }
}
```

Dijstra

```
int g[N][N];  // 存储每条边
int dist[N];  // 存储1号点到每个点的最短距离
bool st[N];   // 存储每个点的最短路是否已经确定

// 求1号点到n号点的最短路，如果不存在则返回-1
int dijkstra()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    for (int i = 0; i < n - 1; i ++ )
    {
        int t = -1;     // 在还未确定最短路的点中，寻找距离最小的点
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        // 用t更新其他点的距离
        for (int j = 1; j <= n; j ++ )
            dist[j] = min(dist[j], dist[t] + g[t][j]);

        st[t] = true;
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

Floyd算法

初始化：

```
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

// 算法结束后，d[a][b]表示a到b的最短距离
void floyd()
{
    for (int k = 1; k <= n; k ++ )
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

prim

```
int n;      // n表示点数
int g[N][N];        // 邻接矩阵，存储所有边
int dist[N];        // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中


// 如果图不连通，则返回INF(值是0x3f3f3f3f), 否则返回最小生成树的树边权重之和
int prim()
{
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    for (int i = 0; i < n; i ++ )
    {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[t] > dist[j]))
                t = j;

        if (i && dist[t] == INF) return INF;

        if (i) res += dist[t];
        st[t] = true;

        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }

    return res;
}
```