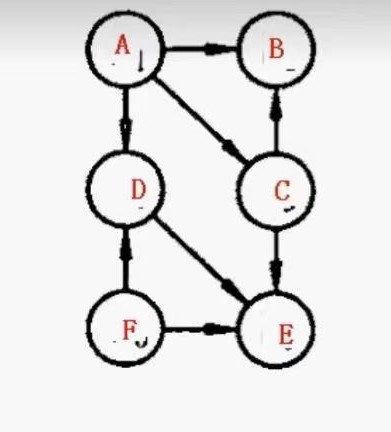
**有向无环图及其应用**

1. **拓扑排序**

AOV网络（关心的是顶点的活动）

过程：

由没有入度的顶点开始，删除顶点后要删除由该顶点出发的边，然后接着排没有入度的点。



//修改的插入边程序 //改为有向图

void InsertEdge(GraphLnk&gl, T v1, T v2) {

int posv1 = GetVertexPos(gl, v1);

int posv2 = GetVertexPos(gl, v2);

if (v1 == -1 || v2 == -1) {

return;

}

Edge\*s;

//V1-->V2

s = (Edge\*)malloc(sizeof(Edge));

assert(s != nullptr);

s->dest = posv2;

//这里采用头插的方式

s->link = gl.NodeTable[posv1].adj;

gl.NodeTable[posv1].adj = s;

//有向图只插入一条边

}

//拓扑排序程序

void TopologicalSort(GraphLnk &g) {

int n = g.NumVertices;

int \*count = (int\*)malloc(sizeof(int)\*n);

assert(count != nullptr);

for (int i = 0; i < n; i++) {

count[i] = 0; //初始化全为0

}

//统计入度

Edge\*p;

for (int i = 0; i < n; i++) {

p = g.NodeTable[i].adj;

while (p != nullptr) {

count[p->dest]++;

p = p->link;

}

}

//找到入度为0的顶点，开始对其进行访问

int top = -1;

for (int i = 0; i < n; i++) {

if (count[i] == 0) {

//假想一个栈

count[i] = top; //将入度为0的顶点“入栈”

top = i; //top指向栈顶元素

}

}

int v, w;

for (int i = 0; i < n; i++) {

if (top == -1) {

cout << "网络中有环" << endl;

return;

}

else {

v = top; //模拟出栈

top = count[top];

cout << g.NodeTable[v].data << "-->";

w = GetFirstNeighbor(g, g.NodeTable[v].data);

while(w != -1) {

count[w]--;

if (count[w] == 0) {

//模拟入栈

count[w] = top;

top = w;

}

w = GetNextNeighbor(g, g.NodeTable[v].data, g.NodeTable[w].data);

}

}

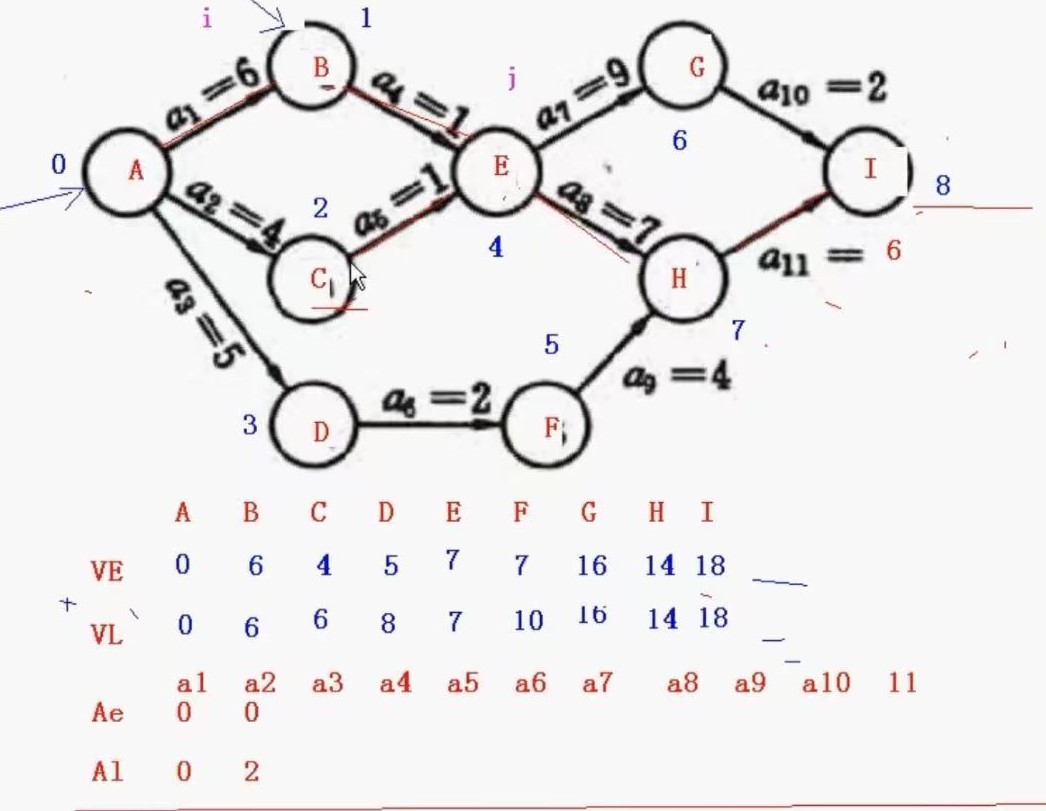
}

}

1. **关键路径**

AOE网络（关心的是边的概念）：完成整项工程至少需要多少时间？哪些活动是影响工程进度的关键？

由于AOE中的活动可以并行进行，所以完成工程的最短时间是从开始点到完成点最长路径的长度，最长路径即为关键路径。



这里的a1,a2,a3指的是活动，活动就是（以a1为例）就是AB之间的边

Ae记录边的最早开始时间，就是尾顶点的最早开始时间

Al记录边的最晚开始时间，就是头顶点的最晚开始时间减去这条活动边上的权值

E GetWeight(GraphMtx& g, int v1, int v2) {

if (v1 == -1 || v2 == -1) {

return 0;

}

return g.Edge[v1][v2];

}

void CriticalPath(GraphMtx& g) {

int n = g.NumVertices;

int \*ve = (int\*)malloc(sizeof(int)\*n); //最早开始时间

int \*vl = (int\*)malloc(sizeof(int)\*n); //最晚开始时间

assert(ve != nullptr&&vl != nullptr);

for (int i = 0; i < n; i++) {

ve[i] = 0;

vl[i] = 0x7FFFFF; //初始化

}

//正向求解出发点最早开始时间

int j,w;

for (int i = 0; i < n; i++) {

j = GetFirstNeighbor(g, g.verticeList[i]);

while (j != -1) {

w = GetWeight(g, i, j);

if (ve[i] + w > ve[j]) { //第一个核心点

ve[j] = ve[i] + w;

}

j = GetNextNeighbor(g, g.verticeList[i], g.verticeList[j]);

}

}

//反向求出发点最晚开始时间

vl[n - 1] = ve[n - 1]; //最后一个顶点肯定为关键活动，所以满足最早开始时间等于最晚开始时间（保证中间没有时间浪费）！其他的点不一定

for (int i = n - 2; i > 0; i--) { //举个例子，一个顶点（活动）要等待18个单位时间才能开始，而他前面一个活动到他要4个单位时间，因此前面一个活动最晚开始时间是14

j = GetFirstNeighbor(g, g.verticeList[i]);

while (j != -1) {

w = GetWeight(g, i, j);

if (vl[j] - w < vl[i]) { //后一个顶点最晚时间减权值是求得该条支路上的最晚开始时间，要与其他支路上（或者是已经求得的最晚开始时间）比,选最小值

vl[i] = vl[j] - w;

}

j = GetNextNeighbor(g, g.verticeList[i], g.verticeList[j]);

}

}

//找关键路径

//即找到每一个关键顶点（最晚开始和最早开始时间相同）

int Ae, Al; //记录边的最早开始时间和最晚开始时间

for (int i = 0; i < n; i++) {

j = GetFirstNeighbor(g, g.verticeList[i]);

while (j != -1) {

Ae = ve[i]; //最早开始时间等于顶点的开始时间

Al = vl[j]- GetWeight(g, i, j); //这里其实是在找下一个顶点

if (Ae == Al) {

cout << "<" << g.verticeList[i] << "," << g.verticeList[j] << ">" << "是关键路径" << endl;

}

j = GetNextNeighbor(g, g.verticeList[i], g.verticeList[j]);

}

}

free(ve);

free(vl);

}