Day 7

拓扑排序

拓扑排序的描述

AOV网（Activity On Vertex NetWork）：用顶点表示活动，弧表示活动间的优先关系的有向图。

拓扑排序：有*N*项活动，用AOV网表示出来，A→B指A必须在B之前完成。请输出一个合理的活动顺序。

AOV网中不能存在有向环。如果存在环，则无法拓扑排序。所以，拓扑排序可用来检查有向图中是否有环。

```cpp
int vis[N]; // 结点访问情况：0表示未访问，1表示正在访问，2表示已访问。
int a[N], a_top;// 结果保存在数组a中
bool DFS(int v){
    vis[v]=1;
    for (int i=0; i<n; i++)
        if (G[i][v]!=INF){ // 找到前趋
            if (vis[i]==1) // 这个点进入两次，说明出现了环
                return false;
            else if (vis[i]==0)
                if (!DFS(i)) return false;
        }
    // 处理点v
    vis[v]=2;
    a[a_top++]=v;
    return true;
}
```

```cpp
bool topsort(){
    memset(visited, 0, sizeof(visited));
    a_top=0;
    for (int i=0;i<n;i++)
        if (!visited[i])
            if (!DFS(i)) return false;
    return true;
}
```

DFS [邻接矩阵]

使用辅助队列

① 记录每个结点入度。

② 将入度为零的点加入队列。

③ 依次对含有入度为零的点的边进行删边操作，同时将新得到的入度为零的点加入队列。

④ 继续对队列中未进行操作的点进行操作。

```cpp
bool topsort(){
    memset(cnt,0,sizeof(cnt));
    // 记录每个结点的入度。
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            if (G[i][j]<INF) cnt[j]++;
    // 将入度为零的点加入队列。
    for (int i=0;i<n;i++)
        if (cnt[i]==0) q.push(i);
    while (!q.empty())  {
        int i=q.front(); q.pop();
        a[a_top++]=i;                          // 记录结点
        for (int j=0;j<n;j++){
            if (G[i][j]<INF){
                cnt[j]--;                      // 依次对入度为零的点进行删边操作
                if (cnt[j]==0) q.push(j);      // 将新得到的入度为零的点加入队列
            }
        }
    }
    // 如果排序结束，但存在入度不为0的点，就说明有环。
    for (int i=0;i<n;i++) if (cnt[i]!=0) return false;
    return true;                               // 图中没有环
}
```

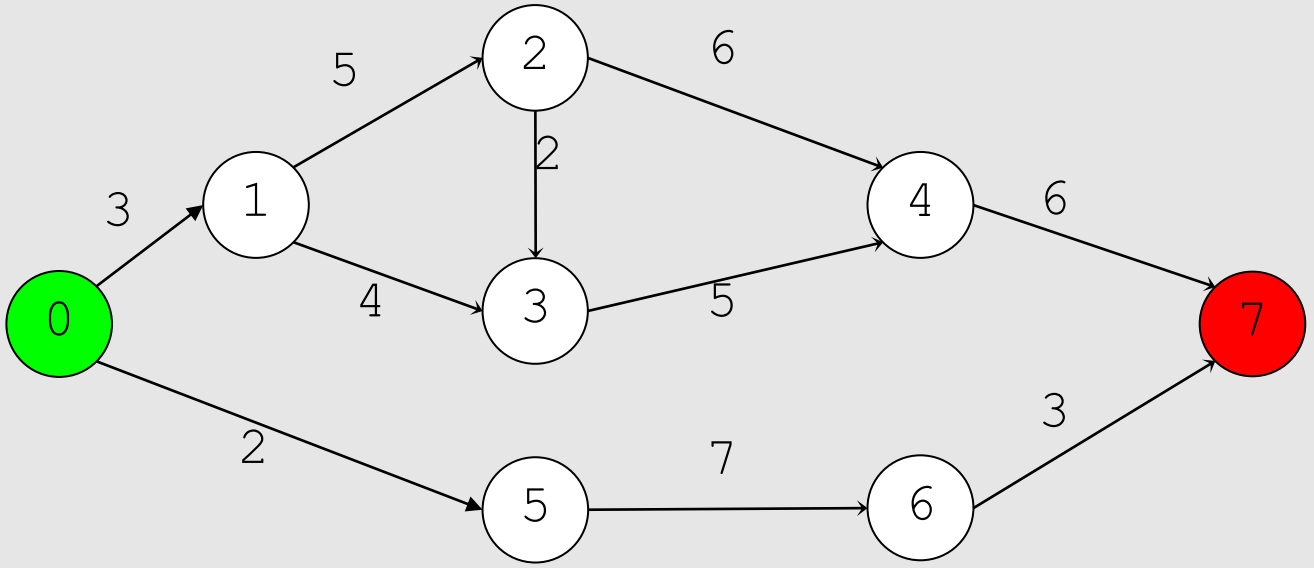
```cpp
queue<int> q;
int cnt[N];   // cnt[i]表示结点i的入度
int a[N], a_top;   // 结果保存在数组a中
```

关键路径

AOE网

AOE网（Activity on Edge）：它是一个带权的有向无环图，可用来估算工程的完成时间。

以下图为例。绿色的0叫做源点，红色的7叫做汇点；其他的每一个点都叫做一个事件；一条边叫做一项活动，权代表活动持续的时间。



其中，路径最长的路径叫做关键路径，影响工程进度的活动叫关键活动，并且关键路径上的活动一定是关键活动。最早开工时间和最晚开工时间相等的工程是关键活动。

问题描述

假设以AOE网表示一个施工流程图，弧上的权值表示完成该项子工程所需的时间。求：

① 完成整个工程的最短时间；

② 在满足依赖关系，并且不影响最短时间的前提下，每项工程的最早开工时间和最晚开工时间；

③ 关键路径

对DAG求关键路径

这是DAG上的动态规划。注意，**只有图的拓扑排序序列为0,1,2,……时才能用这种算法！**

1. **状态表示：** 设$f(i)$为事件i的最早开工时间，$g(i)$为事件i的最晚开工时间。
2. **状态转移方程：**
   $f(i)=\max\{f(j)+G[j][i]\}$
   $g(j)=\min\{g(j)-G[j][i]\}$
   其中j是i的前趋。
3. **边界条件：** $f(0)=0$，$g(n)=f(n)$
4. 计算时先顺推出$f(n)$，然后令$g(n)=f(n)$，再倒推出$g(0)$。如果关键路径通过点i，则$f(i)=g(i)$。

```
f[0]=0;
for (int i=1; i<=n; i++){
    int max=0;
    for (int j=0; j<=n; j++)
        if (G[j][i]!=INF)
            if (G[j][i]+f[j]>max) max=G[j][i]+f[j];
    f[i]=max;
}

g[n]=f[n];
for (int i=n-1; i>=0; i--){
    int min=INF;
    for (int j=0; j<=n; j++)
        if (G[i][j]!=INF)
            if (g[j]-G[i][j]<min) min=g[j]-G[i][j];
    g[i]=min;
}

// 寻找关键路径
for (int i=1; i<n; i++) if (et[i]==eet[i]) cout<<i<<"->";
cout<<n;
```

对拓扑序列求关键路径

这仍然需要动态规划。

1. **划分阶段**：以**拓扑序列**划分阶段，一项工程为一个阶段。
2. **状态表示**：设$f(i)$为事件i的最早开工时间，$g(i)$为事件i的最晚开工时间。
3. **状态转移方程**：

$f(i)=\max\{f(j)+G[j][i]\}$

$g(j)=\min\{g(j)-G[j][i]\}$

其中$j<i$。由于序列由拓扑排序产生，j要么是i的前趋，要么和i无依赖关系。

计算时先顺推出$f(n)$，然后令$g(n)=f(n)$，再倒推出$g(0)$。

```cpp
#include <iostream>
#include <cstring>
using namespace std;

const int INF=100000000, N=10000;
int G[N][N];
int a[N];  // 拓扑排序序列
int f[N],g[N],prev[N];
int n,m;

bool topsort();
// 拓扑排序，所有的“<n”要改成“<=n”。
void print(int i){
    if (i==-1) return;
    print(prev[i]);
    cout<<i<<" ";
}
```

```cpp
int main(){
    cin>>n>>m;
    for (int i=0; i<=n; i++)
        for (int j=0; j<=n; j++)
            G[i][j]=INF;
    for (int i=1; i<=m; i++){
        int u,v,w;
        cin>>u>>v>>w;
        G[u][v]=w;
    }
    memset(f,0,sizeof(f));
    memset(g,0,sizeof(g));
    memset(prev,-1,sizeof(prev));
    if (!topsort()) return 0;
```

```
    for (int i=1; i<=n; i++)
        for (int j=0; j<i; j++) {
            int &x=a[j], &y=a[i];
            if (G[x][y]!=INF && f[y]<f[x]+G[x][y])  {
                f[y]=f[x]+G[x][y];
                prev[y]=x;
            }
        }
    g[n]=f[n];
    for (int i=n-1; i>=0; i--)
        for (int j=n; j>i; j--) {
            int &x=a[j], &y=a[i];
            if (G[y][x]!=INF && g[y]<g[x]-G[y][x])
                g[y]=g[x]-G[y][x];
        }

    cout<<"Len="<<f[a[n]]<<endl;
    print(a[n]);
    return 0;
}
```

最小生成树

生成树

给定一个无向图，如果它的某个子图任意两个顶点都互相连通并且是一棵树，那么这棵树就叫生成树；

如果边上有权值，那么使得边权之和最小的生成树就叫做最小生成树

显然生成树是否存在和图是否连通是等价的经典问题

最小生成树问题

有n个村庄，把村庄看作顶点，村庄之间的道路看作边，边权为两个村庄之间修建道路需要的花费

现在需要在这n个村庄中修建道路使得所有的村庄互通，请问最少需要的花费是多少？

Prim算法

Prim算法是贪心算法，贪心策略为：找到目前情况下能连上的权值最小的边的另一端点，加入之，直到所有的顶点加入完毕。

Prim适用于稠密图。

朴素Prim的时间复杂度是$O(n^2)$，因为在寻找离生成树最近的未加入顶点时浪费了很多时间。所以，可以用堆进行优化。堆优化后的Prim算法的时间复杂度为$O(m\log n)$。

堆优化Prim的代码比较复杂，并查集优化的Kruskal算法与它相比，要好很多。

```cpp
int minEdge[N], cloest[N];      // 与点N连接的最小边

int Prim(int start=0){      // start的出度不能为0!
    int ans=0, k=0, min;
    // 加入第一个点
    for (int i=0;i<n;i++){
        minEdge[i]=G[start][i];
        cloest[i]=start;
    }
    minEdge[start]=0;
    for (int i=0;i<n-1;i++) {
        min=INF; // 寻找离生成树最近的未加入顶点k
        for (int j=0;j<n;j++)
            if (minEdge[j]!=0 && minEdge[j]<min)
                min=minEdge[k=j];
        // 把找到的边加入到MST中
        ans+=minEdge[k];
        minEdge[k]=0; // 加入完毕。以后不用再处理这个点。
        // 重新计算最短边
        for (int j=0;j<n;j++)
            if (G[k][j]<minEdge[j]){
                minEdge[j]=G[k][j];
                cloest[j]=k;
            }
    }
    return ans;
}
```

Kruskal算法

Kruskal算法是贪心算法，贪心策略为：选目前情况下能连上的权值最小的边，若与以生成的树不够成环，加入之，直到n-1条边加入完毕。

时间复杂度为O($n$log$m$)，最差情况为O($m$log$n$)。相比于Prim，这个算法更常用。

```cpp
int parent[N], rank[M];  // p代表并查集，r是边的序号

int comp (const int i, const int j) {return w[i]<w[j];} // 排序时使用
int find (int x){   // 带路径压缩的查找函数
    return parent[x]==x ? x : parent[x] = find(parent[x]);
}

int Kruskal(){
    int ans = 0;
    for (int i=0;i<n;i++) parent[i]=i;       // 初始化并查集
    for (int i=0;i<m;i++) rank[i]=i;// 边的序号（下面要按照边的权值大小来排序）
    sort(rank, rank+m, comp);   // 按照边的权值大小排序

    for (int i=0;i<m;i++){
        int e=rank[i];
        int x=find(u[e]), y=find(v[e]);     // 找出当前边的两个端点所在集合的编号
        if (x!=y){ // 如果不在同一集合，合并
            ans += w[e];
            parent[x] = y;
        }
    }
    return ans;
}
```

哈夫曼

哈夫曼（Huffman）树

哈夫曼树（最优二叉树）：带权路径长度最小的二叉树。

- 树的路径长度：一棵树的每一个叶结点到根结点的路径长度的和。
- 带权二叉树：给树的叶结点赋上某个实数值（称叶结点的权）。
- 带权路径长度：各叶结点的路径长度与其权值的积的总和。

如何构建哈夫树：贪心策略——权越大离根越近。

① 首先，创建*n*个初始的Huffman树，每棵树只包含单一的叶结点，叶结点记录对应字母。

② 接着拿走权最小但没有被处理的两棵树，再把它们标记为Huffman树的叶结点。

③ 把这两个叶结点标记为一个分支结点的两个子结点，而这个结点的权即为两个叶结点的权之和。

④ 重复上述步骤，直到序列中只剩下一个元素。

```
int n;
struct node{
    int w;
    int parent,
        leftchild,
        rightchild;
} h[M];
```

```
node * buildtree(int *weight){  // weight[i]表示结点i的权值。返回值是Huffman的树根。
    int p1, p2;
    int min1, min2;
    memset(h,-1,sizeof(h));
    for (int i=0; i<n; i++) h[i].w=weight[i];

    int m=2*n-1;
    for (int i=n; i<m; i++) {
        min1=min2=INF;
        for (int j=0; j<i; j++)
            if (h[j].parent==-1){
                if (h[j].w < min1)
                    min2=min1, min1=h[j].w, p2=p1, p1=j;
                else if (h[j].w < min2)
                    min2=h[j].w, p2=j;
            }
        h[p1].parent=i;
        h[p2].parent=i;
        h[i].leftchild=p1;
        h[i].rightchild=p2;
        h[i].w=h[p1].w+h[p2].w;
    }
    return &h[2*n-2];
}
```

哈夫曼编码

哈夫曼码：哈夫曼树的非叶结点到左右孩子的路径分别用0，1 表示，从根到叶的路径序列即为哈夫曼码。它的特点如下：

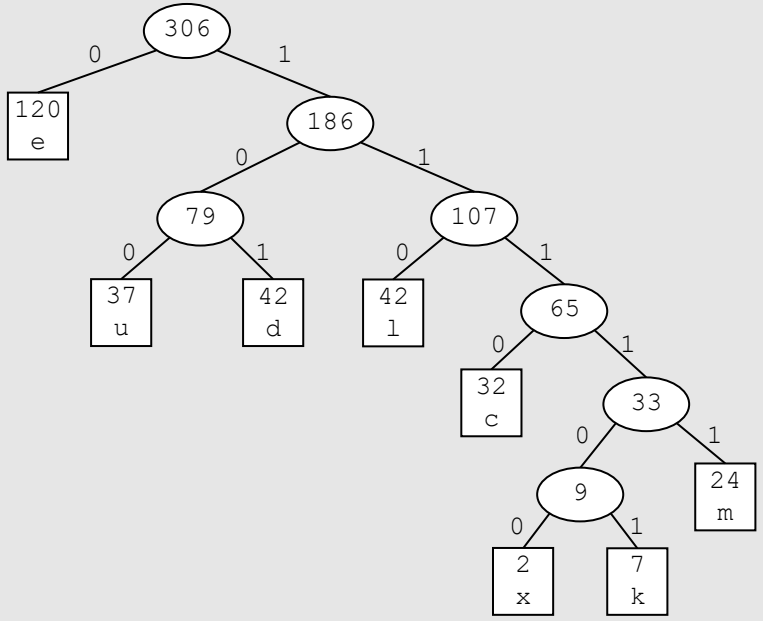
① 代码长度取决于对应字母的相对使用频率（或者“权”），因此它是一种变长编码。

② 任何一字符的编码不是更长编码的前缀部分（否则在解码时会发生混淆）。

假如有几个字母，代码如下

字母	频率	代码	位数
c	32	1110	4
d	42	101	3
e	120	0	1
k	7	111101	6
l	42	110	3
m	24	11111	5
u	37	100	3
z	2	111100	6

则代码对应的哈夫曼树为



那么按照此规则，“cell”对应的代码就是“11100110110”，“1011001110111101”对应的单词就是“duck”。

解码过程如下：从左到右逐位判别代码串，直到确定一个字母。具体的字母需要通过Huffman树确定。