清華大學 Tsinghua University

数据结构 第十一讲

图（下）

刘烨斌

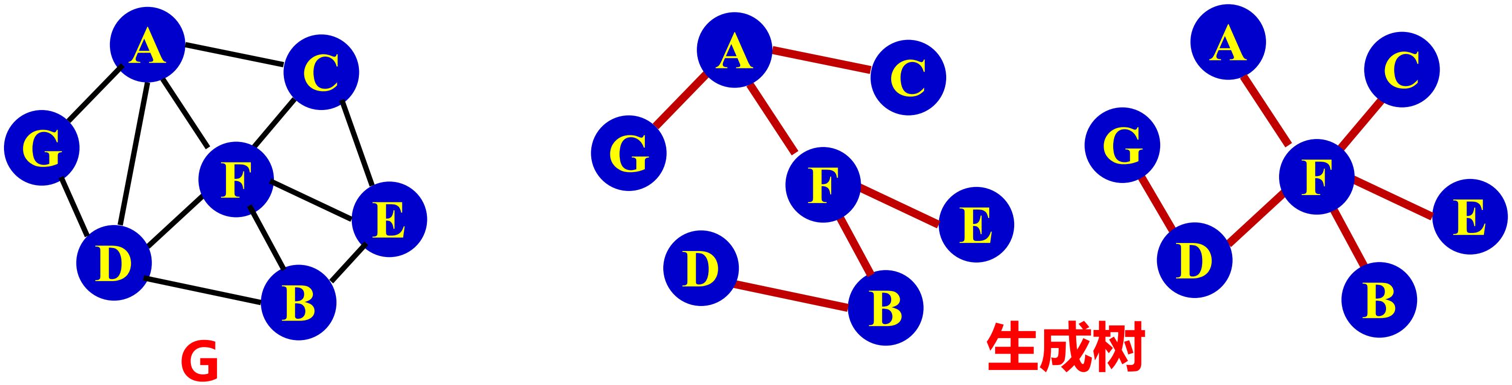
清华大学自动化系

2024年5月28日

最小支撑树

■ 最小支撑树（最小生成树, Minimal Spanning Tree）

- ✓ 连通图G的某一无环连通子图T若覆盖G中所有的顶点，则称作G的一棵支撑树或生成树

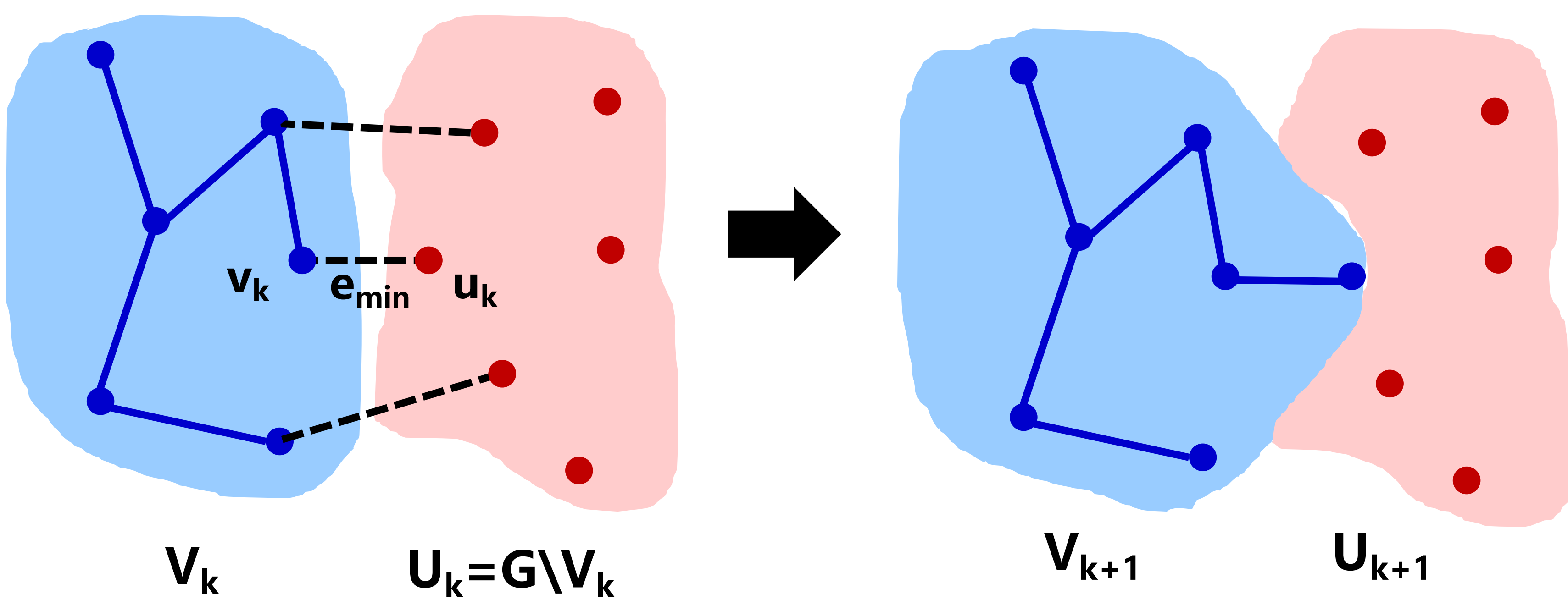


- ✓ n个顶点的连通网络的生成树有n个顶点、n-1条边
- ✓ 不能使用产生回路的边
- ✓ 各边上的权值的总和达到最小
- ✓ 应用：假设有一个网络，用以表示 n 个城市之间架设通信线路，边上的权值代表架设通信线路的成本。如何架设才能使线路架设的成本达到最小?

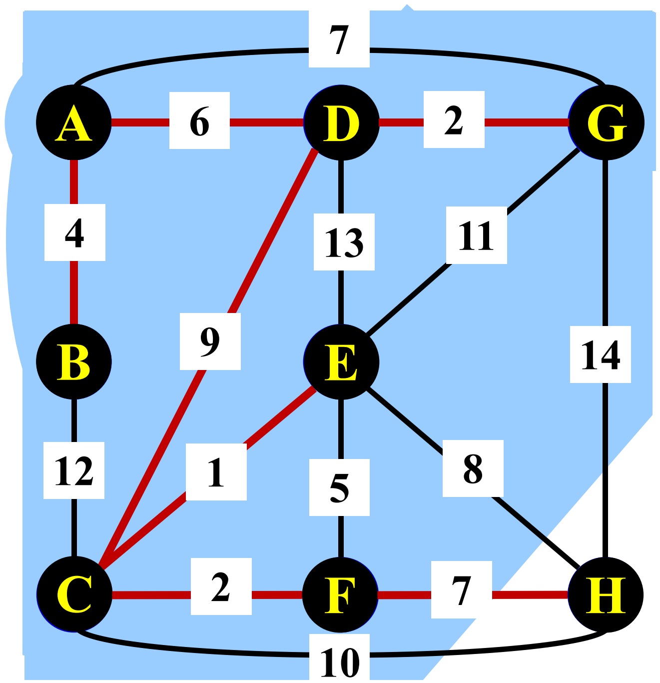
最小支撑树

■ 普里姆算法（ Prim ）

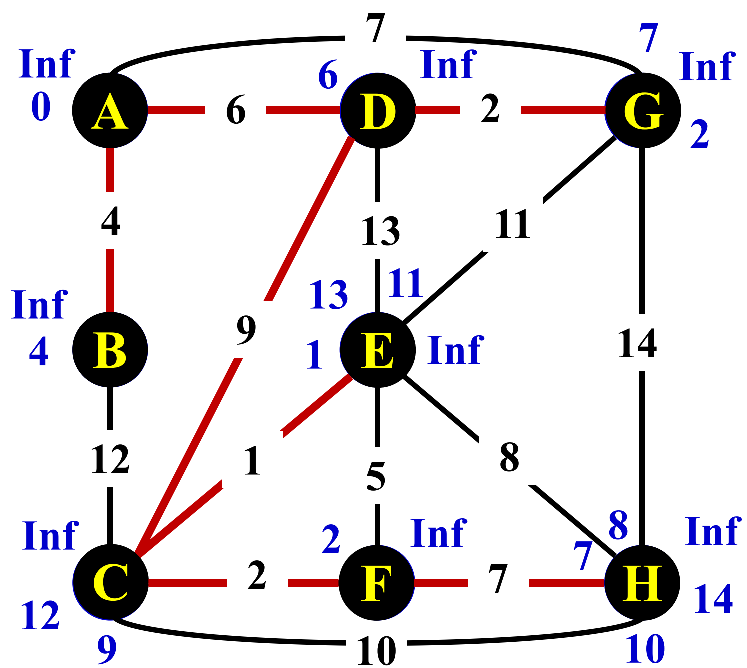
- ✓ 下图中顶点集V和顶点集U构成G={V,U}的一个割（cut）
- ✓ 最小生成树总是采用联接每一割的最短跨越边
- ✓ 采用贪心迭代法，设k时刻的最小生成树为$T_k=\{V_k,E_k\}$，此时最小割为$e_{min}=e_k=\{v_k,u_k\}$, 则第k+1时刻的最小生成树为$T_{k+1}=\{V_k+v_k, E_k+e_k\}$



最小支撑树



最小支撑树


7
Inf
0
A
6
6
D
Inf
2
7
G
Inf
2
4
13
11
Inf
4
B
9
13
11
1
E
Inf
14
12
1
5
8
Inf
C
12
9
2
2
F
Inf
7
7
8
H
Inf
14
10
10

最小支撑树

Prim的实现(非教材版本)

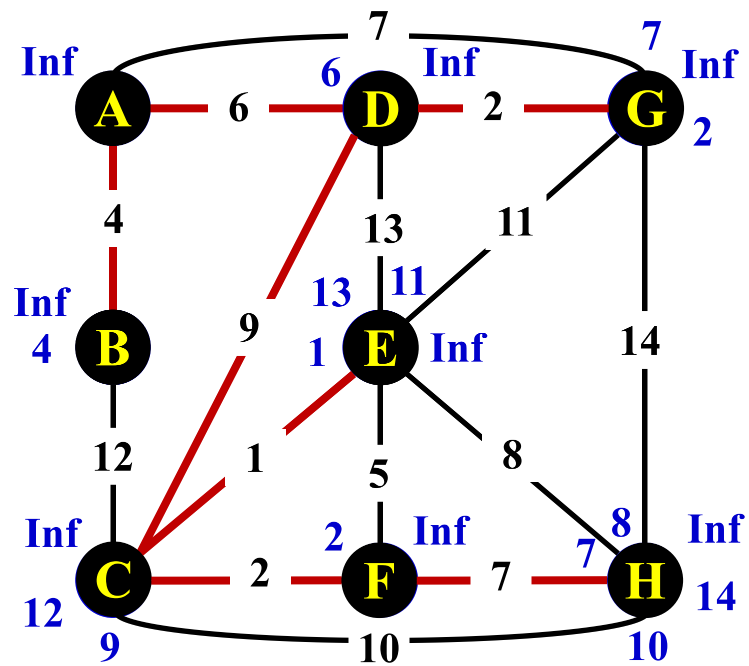
```
void minSpanTreePrim(MGraph g){
    int min, i, j, k; int pred[MAXVEX];int cutcost[MAXVEX];
    cutcost[0] = 0; //0节点直接加入生成树，其割边权重为0
    pred[0] = 0; //0节点前驱为自己
    for (i = 1; i<g.numVertexes; i++){
        //将节点0加入后，更新其他每个节点的割边权重
        cutcost[i] = g.arc[0][i]; //割边权重更新(初始化)
        pred[i] = 0; // 初始化各节点的前驱为节点0
    }
    for (i = 1; i<g.numVertexes; i++){ //循环nV-1次,查找最小割边加入节点加入
        min = INFINITY; // 设置最小
        for (j = 1; j<g.numVertexes; j++){ // 循环节点1至节点nV-1，判断哪个节点加入生成树
            if (cutcost[j] != 0 && cutcost[j]<min){
                min = cutcost[j]; // 更新最小割边权重为j的割边权重
                k = j; // 设置j为当前生成树准备加入的节点
            }
        }
        printf("(%d,%d)", pred[k], k); //k加入生成树，并输出其前驱
        cutcost[k] = 0;//表示该点k已经加入生成树
        for (j = 1; j<g.numVertexes; j++){//该循环检查在k加入后，各顶点割边是否权重是否更新
            if (cutcost[j] != 0 && g.arc[k][j]<cutcost[j]){
                cutcost[j] = g.arc[k][j]; // j点的割边权重更新
                pred[j] = k; // 设置j点的割边连接的点k
            }
        }
}
```

时间复杂度O(n^2)，可通过优先级队列降低

最小支撑树

Prim的实现(非教材版本)

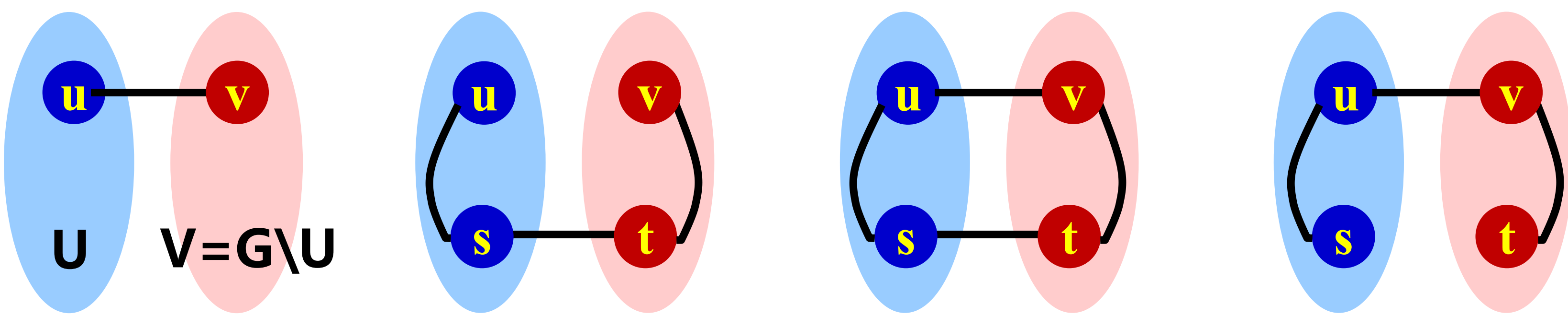
```
void minSpanTreePrim(MGraph g){
    int min, i, j, k; int pred[MAXVEX];int cutcost[MAXVEX];
    cutcost[0] = 0;
    pred[0] = 0;
    for (i = 1; i<g.numVertexes; i++){
        cutcost[i] = g.arc[0][i];
        pred[i] = 0;
    }
    for (i = 1; i<g.numVertexes; i++){
        min = INFINITY;
        for (j = 1; j<g.numVertexes; j++){
            if (cutcost[j] != 0 && cutcost[j]<min){
                min = cutcost[j];
                k = j;
            }
        }
        printf("(%d,%d)", pred[k], k);
        cutcost[k] = 0;
        for (j = 1; j<g.numVertexes; j++){
            if (cutcost[j] != 0 && g.arc[k][j]<cutcost[j]){
                cutcost[j] = g.arc[k][j];
                pred[j] = k;
            }
        }
    }
}
```



最小支撑树

■ 普里姆算法（Prim）正确性证明

- ✓ (a) 反证：假设uv是割(U:G\U)的最小跨越边，而最小生成树未采用
- ✓ (b) 则必有另一跨越边st联接该割（可能s=u或v=t，但不同时成立）
- ✓ (c) 若uv和st同时存在，则构成环
- ✓ (d) 与(b)实现相同的功能，相同的边数，但代价比(b)小，所以(b)不成立



(a) (b) (c) (d)

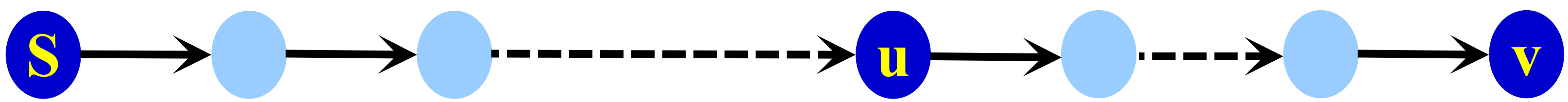
最短路径(树)

- 最短路径(Shortest Path)
  - ✓ 如果从图中某一顶点（称为源点）到另一顶点（称为终点）的路径可能不止一条，如何找到一条路径使得沿此路径上各边上的权值总和达到最小
- 边上权值非负情形的单源最短路径问题
  - ✓ Dijkstra算法 (重点算法)
- 边上权值为任意值的单源最短路径问题
  - ✓ Bellman和Ford算法
- 所有顶点之间的最短路径
  - ✓ Floyd算法

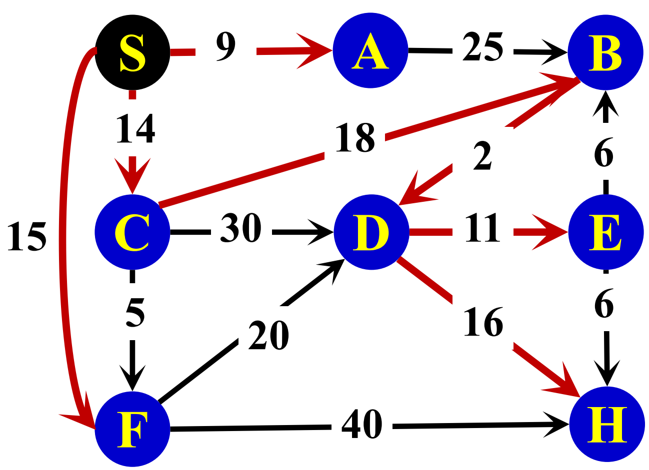
最短路径(树)

最短路径树

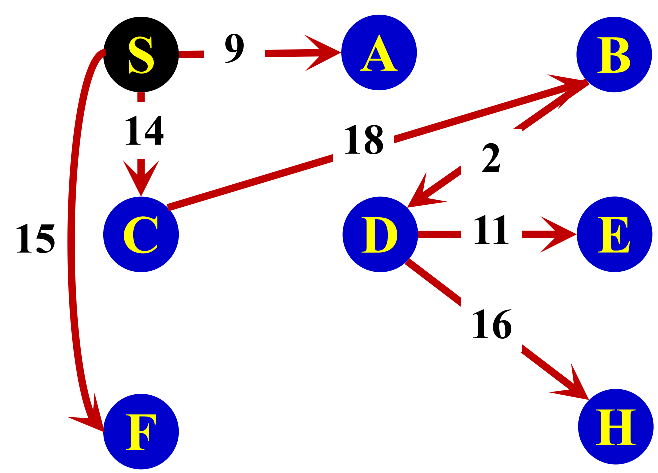
- ✓ 单调性：最短路径的任意前缀也是最短路径；S到v的最短路径经过u，则沿着该路径从S到u也是u的最短路径（可反证）



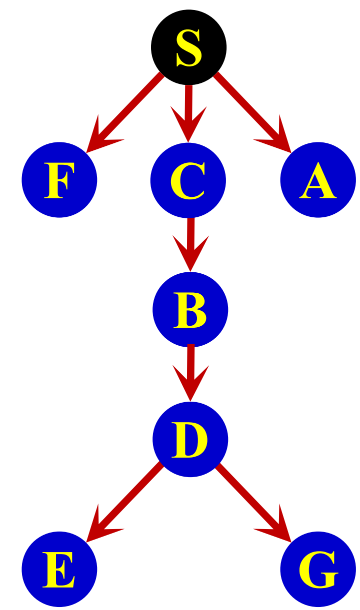
- ✓ 无环性：S到图中其它各点的最短路径的集合必无环
- ✓ 最短路径树（SPT, Shortest Path Tree）



(a) 网络



(b) SPT

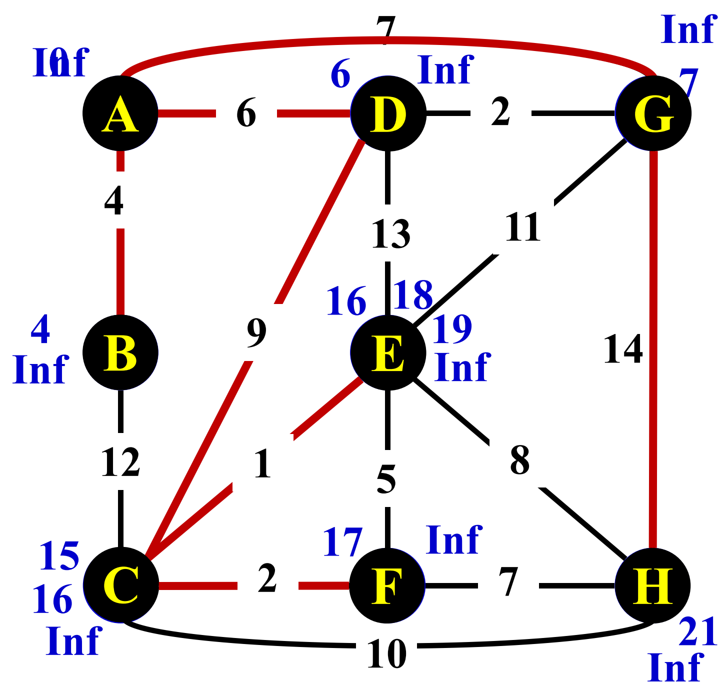


(c) SPT

最短路径(树)

- 迪杰斯特拉(Dijkstra) 算法
  - ✓ 给定一个带权有向图G与源点s，求从s到G中其他顶点的最短路径
  - ✓ 限定各边上的权值大于或等于0
  - ✓ 按路径长度的递增次序， 逐步产生最短路径
  - ✓ 首先求出长度最短的一条最短路径(s,u)，更新SPT的顶点集及s到其他各边的最短距离（更新u的邻域），再求出s到其它顶点长度次短的一条最短路径，依次类推，直到所有顶点进入SPT集合

最短路径(树)

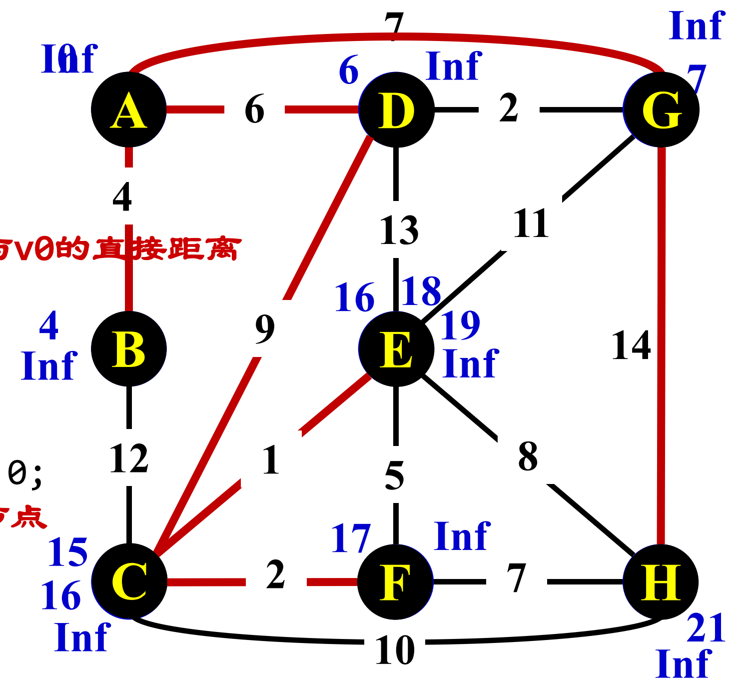


A
B
C
D
E
F
G
H
7
6
2
4
13
11
9
14
12
1
5
8
2
7
10
Inf
6
Inf
7
4
Inf
16
18
19
Inf
15
16
Inf
17
Inf
21
Inf

最短路径(树)

■ Dijkstra的实现（非教材版本）

```
void Dijkstra(MGraph g,int v0){
    int min,i,j,k; bool inTree[MAXVEX];
    int mindist[MAXVEX]; //当前每个节点的最短路径
    int pred[MAXVEX];    //每个节点的前驱
    for (int i = 0; i < g.numVertexes; ++i){
        mindist[i] = g.arc[v0][i]; //节点的初始最短距离为与v0的直接距离
        inTree[i] = false;    //初始都未用过该点
        if (mindist[i] == INFINITY) pred[i] = -1;
        else pred[i] = v0;  //设置前驱
    }
    inTree[v0] = true;  //节点v0为起始顶点  mindist[v0] = 0;
    for (i = 1; i < g.numVertexes; i++){//循环nV-1次加新节点
        min = INFINITY;
        for (j = 0; j < g.numVertexes; ++j)
        if ((!inTree[j]) && mindist[j]<min){
            k = j; // 保存当前节点号
            min = mindist[j]; //更新最小长度
        }
        inTree[k] = true; // 设置节点k进入最短路径树
        printf("(%d,%d,%d)", pred[k], k, mindist[k]); //k加入最短路径树，并输出其前驱
        for (j = 0; j < g.numVertexes; j++) // 循环k的所有邻域节点进行
        if ((!inTree[j]) && g.arc[k][j]<INFINITY){ //k的邻域节点
            if (mindist[k] + g.arc[k][j] < mindist[j]){ //通过k点找更短的路径
                mindist[j] = mindist[k] + g.arc[k][j];  //更新j的最短路径
                pred[j] = k;    //记录j的前驱顶点为k
            }
        }
```



时间复杂度$O(n^2)$，可通过优先级队列降低

最短路径(树)与最小支撑树

普里姆(Prim) 算法实现（教材提供代码，非书上版本）

```
template <typename Tv, typename Te> void Graph<Tv, Te>::prim ( int s ) {
   reset(); priority ( s ) = 0;
   for ( int i = 0; i < n; i++ ) { //共需引入n个顶点和n-1条边
      status ( s ) = VISITED;
      if ( -1 != parent ( s ) ) type ( parent ( s ), s ) = TREE; //引入当前的s
      for ( int j = firstNbr ( s ); -1 < j; j = nextNbr ( s, j ) ) //枚举s的所有邻居j
         if ( ( status ( j ) == UNDISCOVERED ) && ( priority ( j ) > weight ( s, j ) ) )
            { priority ( j ) = weight ( s, j ); parent ( j ) = s; } //与Dijkstra唯一不同
      for ( int shortest = INT_MAX, j = 0; j < n; j++ ) //选出下一极短跨边
         if ( ( status ( j ) == UNDISCOVERED ) && ( shortest > priority ( j ) ) )
            { shortest = priority ( j ); s = j; }
   }
}
```

迪杰斯特拉(Dijkstra) 算法实现（教材提供代码，非书上版本）

```
template <typename Tv, typename Te> //最短路径Dijkstra算法：适用于一般的有向图
void Graph<Tv, Te>::dijkstra ( int s ) { //assert: 0 <= s < n
   reset(); priority ( s ) = 0;
   for ( int i = 0; i < n; i++ ) { //共需引入n个顶点和n-1条边
      status ( s ) = VISITED;
      if ( -1 != parent ( s ) ) type ( parent ( s ), s ) = TREE; //引入当前的s
      for ( int j = firstNbr ( s ); -1 < j; j = nextNbr ( s, j ) ) //枚举s的所有邻居j
         if ( ( status ( j ) == UNDISCOVERED ) && ( priority ( j ) > priority (s) + weight(s, j)))
            { priority ( j ) = priority ( s ) + weight ( s, j ); parent ( j ) = s; }
      for ( int shortest = INT_MAX, j = 0; j < n; j++ ) //选出下一最近顶点
         if ( ( status ( j ) == UNDISCOVERED ) && ( shortest > priority ( j ) ) )
            { shortest = priority ( j ); s = j; }
   }
}
```

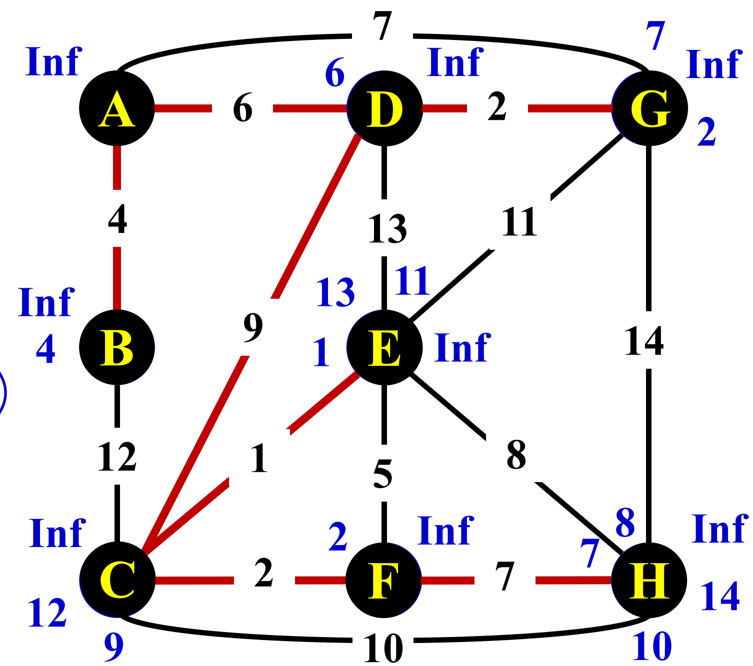
回顾：最小支撑树

Prim的实现(非教材版本)

```
void minSpanTreePrim(MGraph g){
    int min, i, j, k; int pred[MAXVEX];int cutcost[MAXVEX];
    cutcost[0] = 0;
    pred[0] = 0;
    for (i = 1; i<g.numVertexes; i++){
        cutcost[i] = g.arc[0][i];
        pred[i] = 0;
    }
    for (i = 1; i<g.numVertexes; i++){
        min = INFINITY;
        for (j = 1; j<g.numVertexes; j++){
            if (cutcost[j] != 0 && cutcost[j]<min){
                min = cutcost[j];
                k = j;
            }
        }
        printf("(%d,%d)", pred[k], k);
        cutcost[k] = 0;
        for (j = 1; j<g.numVertexes; j++){
            if (cutcost[j] != 0 && g.arc[k][j]<cutcost[j]){
                cutcost[j] = g.arc[k][j];
                pred[j] = k;
            }
        }
    }
}
```

选取优先级
最高顶点

邻域优先级更新



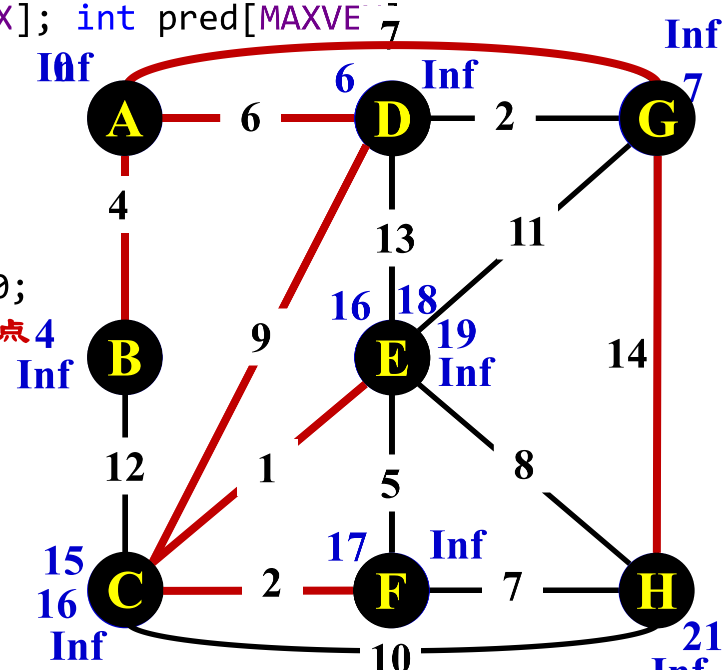
回顾：最短路径(树)

Dijkstra的实现（非教材版本）

```
void Dijkstra(MGraph g,int v0){
    int min,i,j,k; bool inTree[MAXVEX]; int mindist[MAXVEX]; int pred[MAXVE
    for (int i = 0; i < g.numVertexes; ++i){
        mindist[i] = g.arc[v0][i]; inTree[i] = false;
        if (mindist[i] == INFINITY) pred[i] = -1;
        else pred[i] = v0;   //设置前驱
    }
    inTree[v0] = true;  //节点v0为起始顶点  mindist[v0] = 0;
    for (i = 1; i < g.numVertexes; i++){//循环nV-1次加新节点
        min = INFINITY;
        for (j = 0; j < g.numVertexes; ++j)
        if ((!inTree[j]) && mindist[j]<min){
            k = j; // 保存当前节点号
            min = mindist[j]; //更新最小长度
        }
        inTree[k] = true; // 设置节点k进入最短路径树
        printf("(%d,%d,%d)", pred[k], k, mindist[k]); //k加入最短路径树，并输出其前驱
        for (j = 0; j < g.numVertexes; j++) // 循环k的所有邻域节点进行
        if ((!inTree[j]) && g.arc[k][j]<INFINITY){ //k的邻域节点
            if (mindist[k] + g.arc[k][j] < mindist[j]){ //通过k点找更短的路径
                mindist[j] = mindist[k] + g.arc[k][j];  //更新j的最短路径
                pred[j] = k;    //记录j的前驱顶点为k
            }
        }
    }
}
```

选取优先级最高顶点

邻域优先级更新



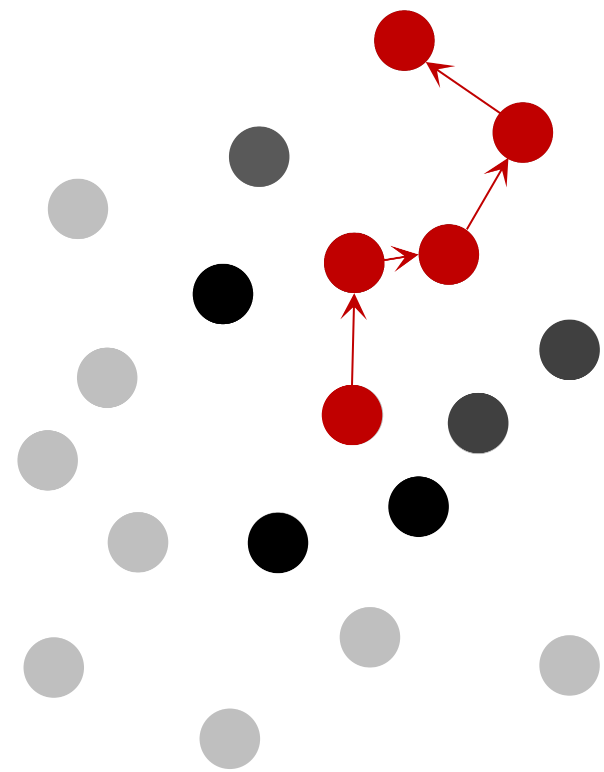
图搜索的统一框架

图的遍历搜索

顶点邻域优先级更新

迭代

选取最高优先级顶点



回顾：广度优先搜索

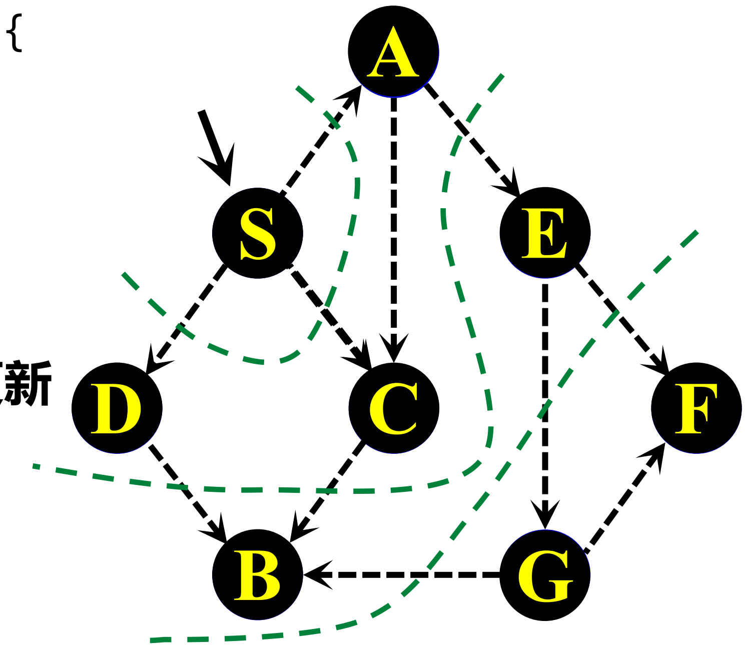
G F B E D C A

```
void Graph<Tv, Te>::BFS ( int v, int& clock ) {
   Queue<int> Q;
   status ( v ) = DISCOVERED;
   Q.enqueue ( v );
   dTime ( v ) = ++clock;
   while ( !Q.empty() ) {
      int v = Q.dequeue();
      for ( int u = firstNbr ( v );
            -1 < u; u = nextNbr ( v, u ) )
         if ( UNDISCOVERED == status ( u ) ) {
            status ( u ) = DISCOVERED;
            Q.enqueue (u);
            parent ( u ) = v;
            dTime ( u ) = ++clock; type ( v, u ) = TREE;
         } else {
            type ( v, u ) = CROSS;
      status ( v ) = VISITED;
   }
}
```

选取优先级最高顶点

邻域优先级更新

借助队列实现隐式、高效的优先级更新



S UNDISCOVERED

S DISCOVERED

回顾：深度优先搜索

栈实现

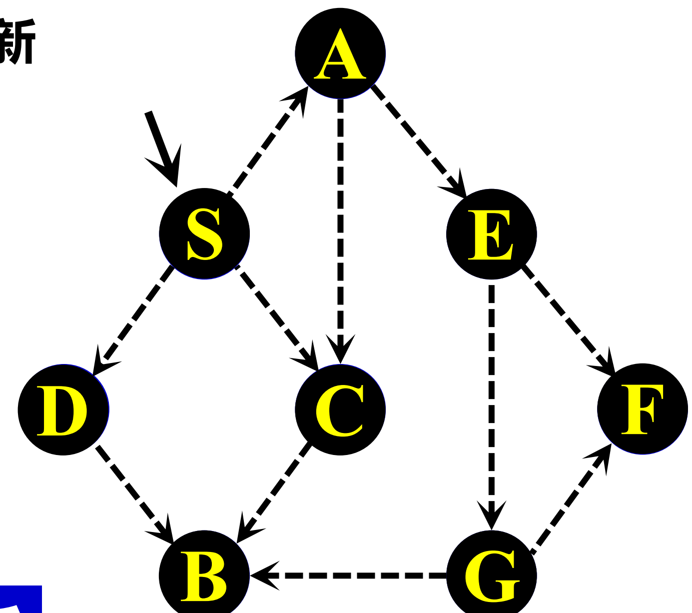
D C C B F

S A E F G B C D

```
void Graph<Tv, Te>::DFS ( int v) {
   Stack<int> S;
   S.push ( v );
   while ( !S.empty() ) {
      int v = S.pop();
      status ( v ) = DISCOVERED;
      for ( int u = firstNbr ( v );
            -1 < u; u = nextNbr ( v, u ) )
         if ( UNDISCOVERED == status ( u ) ) {
            S.push (u);
            status (u) = DISCOVERED;
         }
   }
}
```

选取优先级最高顶点

邻域优先级更新



S UNDISCOVERED

S DISCOVERED

借助栈，实现隐式、高效的优先级更新

图的广义搜索框架

■ 搜索框架

基于队列的广度优先遍历	基于栈的深度优先遍历	基于优先级队列的图优先级遍历
从队列中取顶点v	从栈中取顶点v	遍历顶点 取优先级最高点v
↕	↕	↕
对v邻域顶点u 入队列	对v邻域顶点u 入栈	对v邻域顶点u 优先级更新

推广 ➡

使用队列和栈，简化选取最高优先级顶点步骤复杂度

统一的处理框架，支持更复杂的优先计数方法

基于优先级队列的最小生成树与最短路

■ 最小生成树

```
void Prim ( Node start ) {
   priority_queue<Node> PQ;
   status (start) = DISCOVERED;
   PQ.push (start);
   while ( !PQ.empty() ) {
      int v = PQ.pop();
      if (VISITED == status (v))
          continue;
      for ( int u = firstNbr (v);
            -1 < u; u = nextNbr (v, u))
         if (VISITED != status (u) &&
             u.priority > weight(v,u))
         {
            u.priority = weight(v,u);
            PQ.push (u);
            parent ( u ) = v;
         }
      status ( v ) = VISITED;
   }
}
```

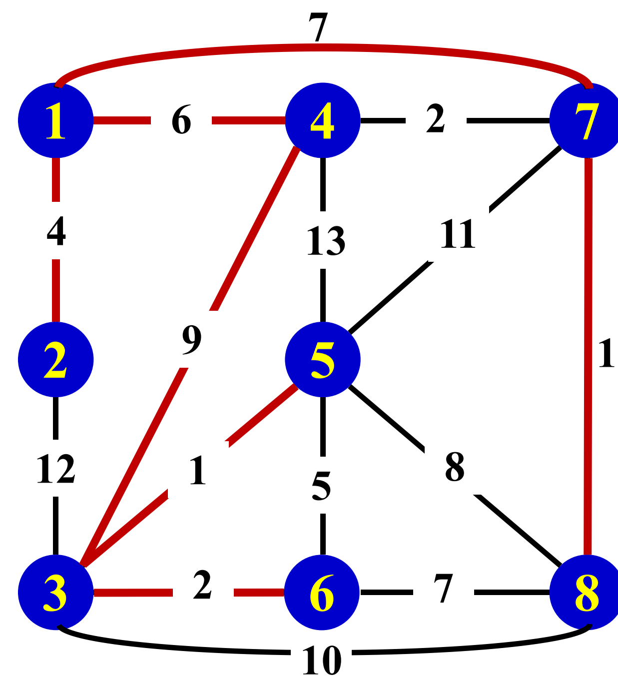
■ 最短路径树

```
void Dijkstra ( Node start ) {
   priority_queue<Node> PQ;
   status (start) = DISCOVERED;
   PQ.push (start);
   while ( !PQ.empty() ) {
      int v = PQ.pop();
      if (VISITED == status (v))
          continue;
      for (int u = firstNbr (v);
            -1 < u; u = nextNbr (v, u))
         if(VISITED != status (u) &&
            u.priority>weight(v,u)+v.priority)
         {
            u.priority=weight(v,u)+v.priority;
            PQ.push (u);
            parent ( u ) = v;
         }
      status ( v ) = VISITED;
   }
}
```

基于优先级队列的最短路

Dijkstra的优先级队列实现

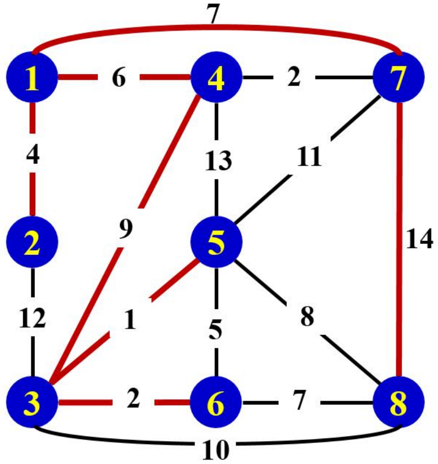
```
struct Node{
    int end; int weight;
    friend bool operator < (Node A, Node B){
        return  A.weight > B.weight;
    }
};
bool visited[maxn];
int m, n;
vector< vector<Node> > G; // 构建邻接表

int main(){
    Node P;
    fscanf_s(infile, "%d %d\n", &n, &m);
    G.clear(); G.resize(n + 1);
    for (int i = 0; i<m; i++){
        int a, b, c;
        fscanf_s(infile, "%d %d %d\n", &a, &b, &c);
        P.end = b;
        P.weight = c;
        G[a].push_back(P);
        P.end = a;
        G[b].push_back(P);
    }
    Dij(1);
}
```



输入:	3 5 1	
8 15	3 6 2	4 7 2
1 2 4	1 7 7	5 7 11
2 3 12	4 5 13	5 8 8
1 4 6	5 6 5	6 8 7
3 4 9	3 8 10	7 8 14

基于优先级队列的最短路

■ Dijkstra的优先级队列实现




```
int Dij(int Star, int End){
     Node P, Pn;
     P.end = Star; P.weight = 0;
     priority_queue<Node> Q;
     Q.push(P);              //把顶点放入优先级队列
     while (!Q.empty()){
         P = Q.top();  Q.pop(); //提取优先级最高顶点
         if (visited[P.end]) continue;  // 若该顶点被访问过，则返回
         visited[P.end] = true; // 设置该顶点访问标记
         int nEdge = G[P.end].size(); // 该顶点的邻域表个数
         for (int i = 0; i< nEdge; i++){
             Pn.end = G[P.end][i].end; // 取出第i个邻域顶点的秩
             Pn.weight = G[P.end][i].weight + P.weight; // 对应权重修改
             if (!visited[Pn.end])    // 若该邻域顶点未被访问，则放入队列
                    Q.push(Pn);
         }
     }
     return -1;
}
```

基于优先级队列的最短路

■ Dijkstra的优先级队列实现

```
int Dij(int Star, int End){
     Node P, Pn;
     P.end = Star; P.weight = 0;
     priority_queue<Node> Q;
     Q.push(P);                //把顶点放入优先级队列
     while (!Q.empty()){ //至多e次提取
         P = Q.top();  Q.pop(); //提取优先级最高顶点(维护堆序性, 下滤, 复杂度O(loge))
         if (visited[P.end]) continue;  // 若该顶点被访问过, 则返回
         visited[P.end] = true; // 设置该顶点访问标记
         int nEdge = G[P.end].size(); // 该顶点的邻域表个数
         for (int i = 0; i< nEdge; i++){
             Pn.end = G[P.end][i].end; // 取出第i个邻域顶点的秩
             Pn.weight = G[P.end][i].weight + P.weight; // 对应权重修改
             if (!visited[Pn.end])    // 若该邻域顶点未被访问, 则放入队列
                    Q.push(Pn);
         }                        // 放入该顶点进入优先级队列, 不对重复顶点进行合并,
     }                            // 每个顶点可能重复放入, 队列中元素至多为边的数目e
     return -1;                   // 此处至多放入e次
}
```

整体时间复杂度： O(eloge)=O(elogn)

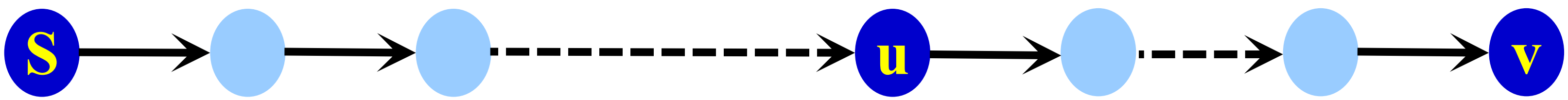
最短路径(树)

- 最短路径(Shortest Path)
  - ✓ 如果从图中某一顶点（称为源点）到另一顶点（称为终点）的路径可能不止一条，如何找到一条路径使得沿此路径上各边上的权值总和达到最小
- 边上权值非负情形的单源最短路径问题
  - ✓ Dijkstra算法 (重点算法)
- 边上权值为任意值的单源最短路径问题
  - ✓ Bellman和Ford算法
- 所有顶点之间的最短路径
  - ✓ Floyd算法

最短路径(树)

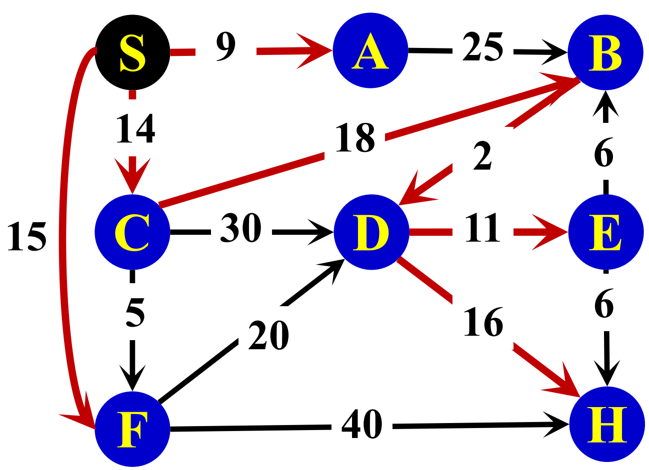
■ 最短路径树

✓ 单调性：最短路径的任意前缀也是最短路径；S到v的最短路径经过u，则沿着该路径从S到u也是u的最短路径（可反证）

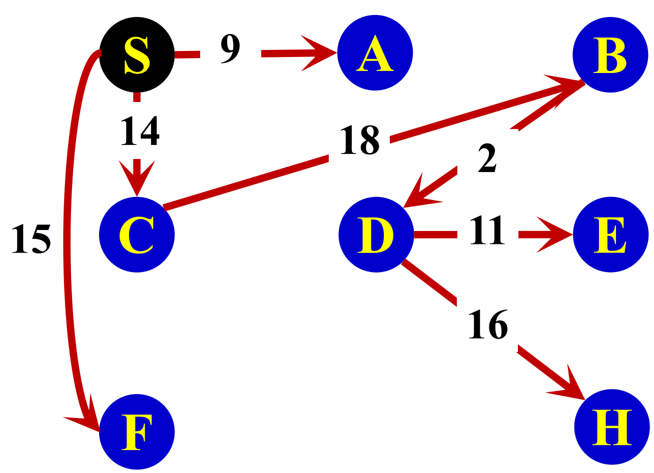


✓ 无环性：S到图中其它各点的最短路径的集合必无环

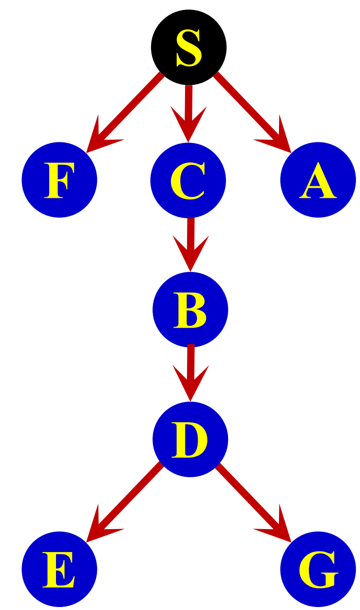
✓ 最短路径树（SPT, Shortest Path Tree）



(a) 网络



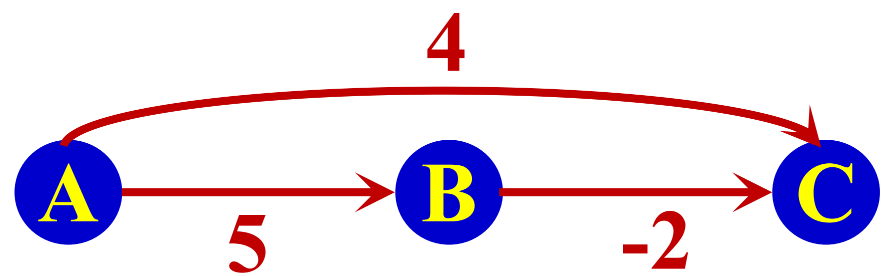
(b) SPT



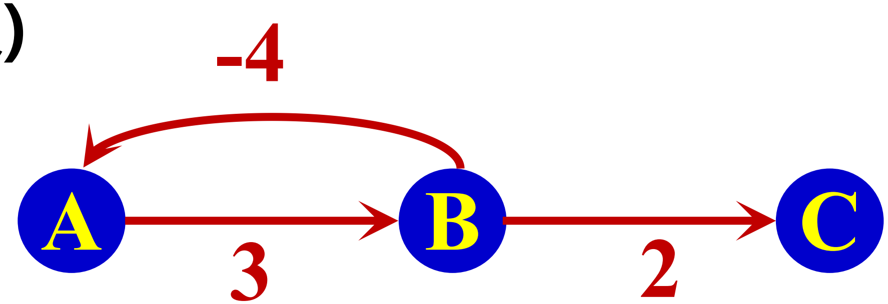
(c) SPT

边有负值的最短路径问题

- Dijkstra的局限
  - ✓ BC边权值为负，使用Dijkstra求从A到C最短路径为AC，代价为4
  - ✓ 从A到B再到C则代价为3



- Bellman和Ford算法
  - ✓ 解决负值问题。限制条件：图中不能包含负权回路（回路的权值和为负)



负值和回路

边有负值的最短路径问题

Bellman和Ford算法

- ✓ 没有负权和回路时，n个顶点的图中任意两个顶点之间如果存在最短路径，此路径最多有n-1条边
- ✓ 构造最短路径长度数组序列$\text{dist}^1[v], \text{dist}^2[v], \ldots, \text{dist}^{n-1}[v]$。
- ✓ $\text{dist}^1[v]$是从源点u到终点v的只经过一条边的最短路径的长度$\text{dist}^1[v] = \text{Edge}[u][v]$
- ✓ $\text{dist}^k[v]$是从源点u出发最多经过k条边到达终点v的最短路径长度
- ✓ 算法的最终目的是计算出 $\text{dist}^{n-1}[v]$，采用**递推计算**

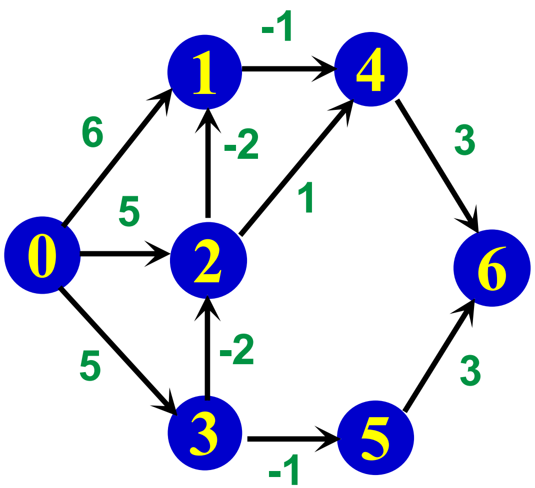
$$\text{dist}^1[v] = \text{Edge}[u][v];$$

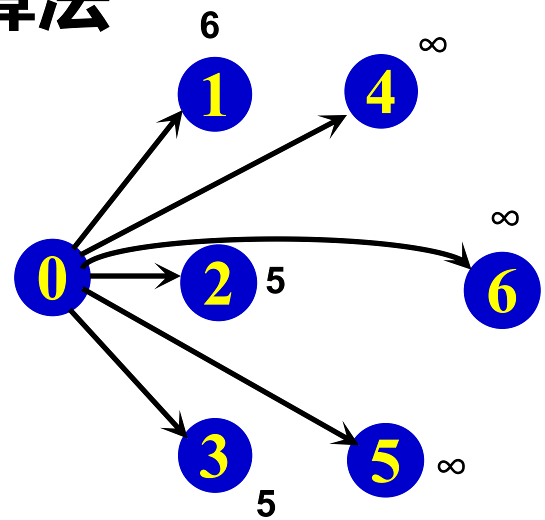
$$\text{dist}^k[v] = \min\{\text{dist}^{k-1}[v], \min_j\{\text{dist}^{k-1}[j]+\text{Edge}[j][v]\}\}$$

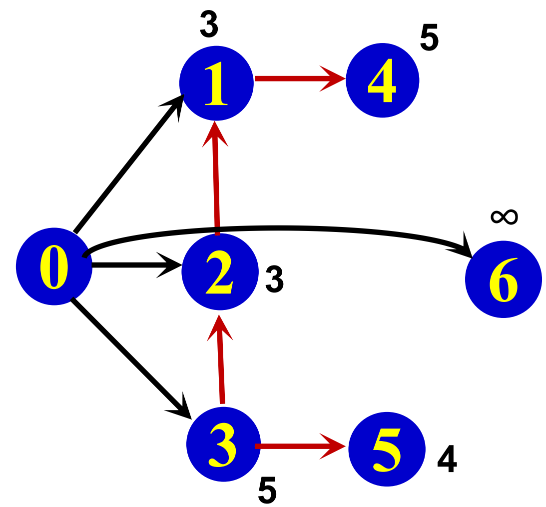
- ✓ 算法需判断是否存在负权和回路，可在n-1次迭代后再做一迭代，若某节点最小距离仍能更新，则存在负值和回路

边有负值的最短路径问题

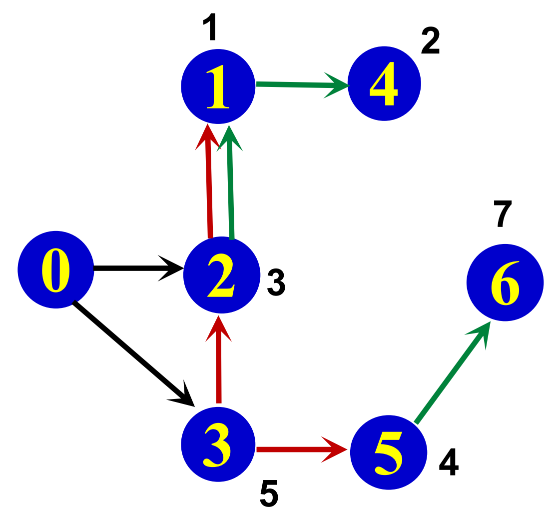
Bellman和Ford算法





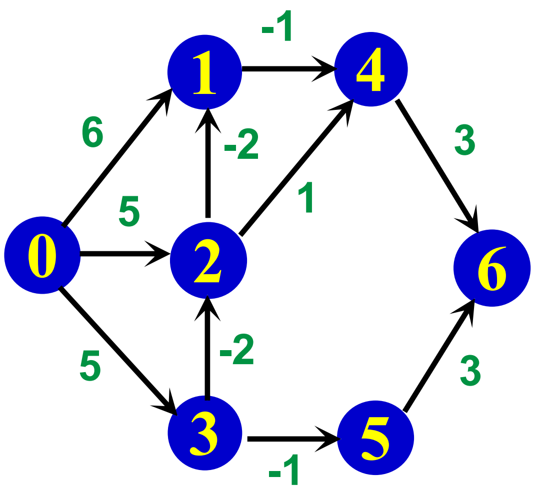


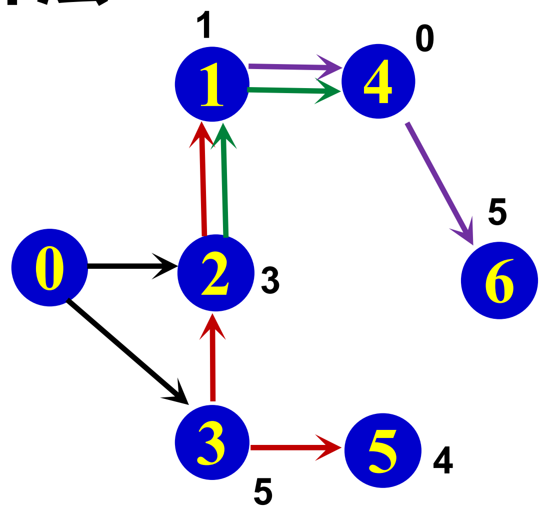
k	$d^k[0]$	$d^k[1]$	$d^k[2]$	$d^k[3]$	$d^k[4]$	$d^k[5]$	$d^k[6]$
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7

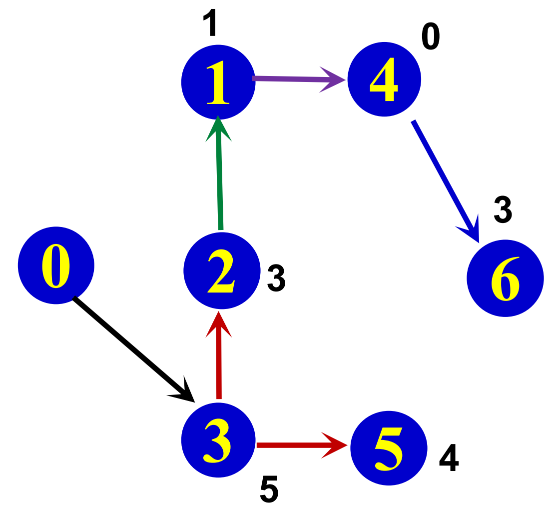


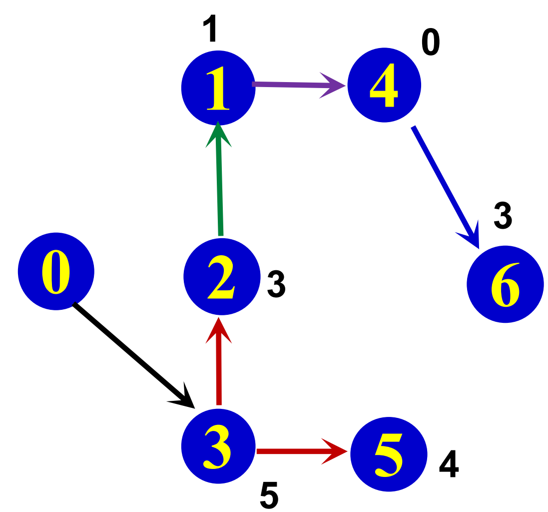
边有负值的最短路径问题

Bellman和Ford算法







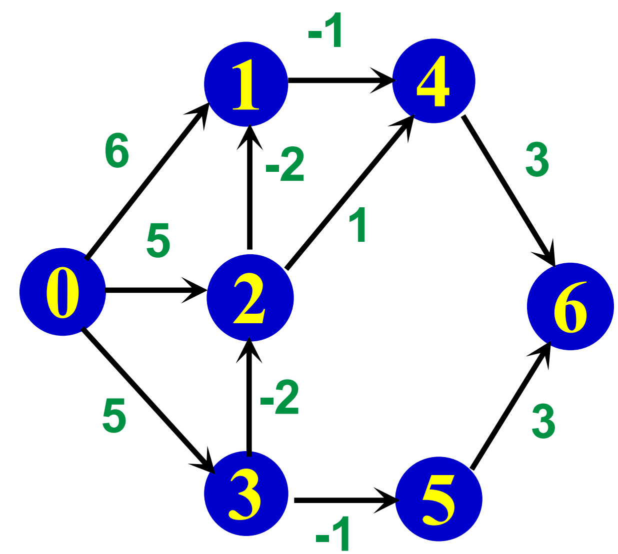


k	d^k[0]	d^k[1]	d^k[2]	d^k[3]	d^k[4]	d^k[5]	d^k[6]
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

边有负值的最短路径问题

Bellman和Ford算法

- ✓ 固定k情况下，计算 min { $dist^{k-1}[j]$+ Edge[j][v] }，仍需遍历每个v，内层遍历j，复杂度为O(n²)
- ✓ 可直接遍历所有边，只对每条的终端节点距离进行更新，为此复杂度O(e)
- ✓ 整体复杂度O(ne)



k	$d^k[0]$	$d^k[1]$	$d^k[2]$	$d^k[3]$	$d^k[4]$	$d^k[5]$	$d^k[6]$
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

- ✓ 进一步优化：SPFA算法 (Shortest Path Faster Algorithm) (可自学)

边有负值的最短路径问题

■ Bellman和Ford算法实现

```
#define MAX 0x3f3f3f3f
#define N 1010
int nodenum, edgenum, original; //点，边，起点
typedef struct Edge {
      int u, v; //u为起点，v为终点
      int cost; //边的权重
}Edge;

Edge edge[N];
int dis[N], pre[N];

void print_path(int root) { //打印最短路的路径（反向）
    while (root != pre[root]) { //前驱
        printf("%d-->", root);  root = pre[root];
    }
    if (root == pre[root])  printf("%d\n", root);
}
```

边有负值的最短路径问题

Bellman和Ford算法实现

```
int main(){
    scanf("%d%d%d", &nodenum, &edgenum, &original);
    pre[original] = original;
    for (int j = 1; j <= edgenum; ++j){
        scanf("%d%d%d",&edge[j].u, &edge[j].v, &edge[j].cost);
    }
    if (!Bellman_Ford())
        for(int i = 1; i <= nodenum; ++i) {//每个点最短路
            printf("%d\n", dis[i]);
            printf("Path:");
            print_path(i);
        }
    else
        printf("have negative circle\n");
    return 0;
}
```

边有负值的最短路径问题

Bellman和Ford算法实现

```
bool Bellman_Ford(){
    for (int i = 1; i <= nodenum; ++i) //初始化
        dist[i] = (i == original ? 0 : MAX);
    for (int k = 1; k <= nodenum - 1; ++k) //k次迭代
        for (int j = 1; j <= edgenum; ++j)
                            // 对原公式n²次松弛，简化为e次边的松弛
            if (dist[edge[j].v]>dist[edge[j].u]+edge[j].cost){
                dist[edge[j].v]=dist[edge[j].u]+edge[j].cost;
                pre[edge[j].v] = edge[j].u;
            }
    bool negative = false; //判断是否含有负权回路
    for (int j = 1; j <= edgenum; ++j)
            // 再做一次迭代看是否有任意边可改进，若是则有负权和回路
        if(dis[edge[j].v] > dis[edge[j].u]+edge[j].cost){
            negative = true; break;}
    return negative;
}
```

边有负值的最短路径问题

Bellman和Ford算法正确性证明

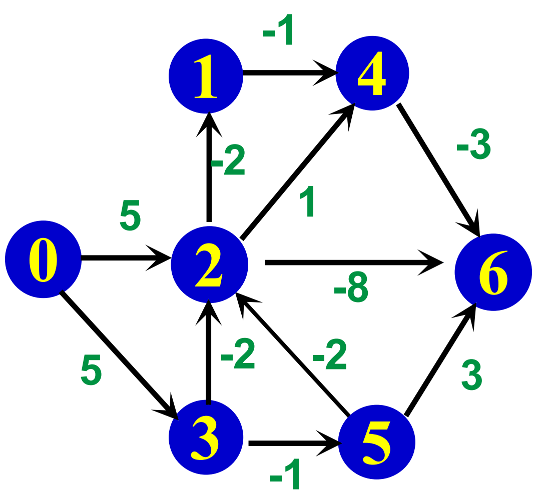
- ✓ 为何能检测负权和环路并返回正确的negative判断?
  negative=0， 无负权回路； negative=1, 有负权回路;

- ✓ 若图中有负权和环路，而negative=0
- ✓ 则设该环路为包含m个节点$c=\{v_0,v_1,v_2\ldots v_m\}$, $v_m=v_0$，则有$\sum_{i=1}^{m}cost(v_{i-1},v_i)<0$。假设算法返回negative=0，则对任意 i 都有 $dist(v_i)\le dist(v_{i-1})+cost(v_{i-1},v_i)$, 这里 $i=1,2\ldots,m$。将环路c上的所有这些不等式相加，有$\sum_{i=1}^{m}dist(v_i)\le\sum_{i=1}^{m}dist(v_{i-1})+\sum_{i=1}^{m}cost(v_{i-1},v_i)$，由于$v_m=v_0$，上式得到$\sum_{i=1}^{m}cost(v_{i-1},v_i)\ge 0$，与环路为负权和的结论矛盾，故算法必定返回1

填空题 1分

设置

针对下图进行Bellman&Ford算法，已知某个迭代步骤k时，到达各节点的最短路长度值为下表，求步骤k+1时，各节点的最短路长度值的总和为 [填空1] 。



	$d^k[0]$	$d^k[1]$	$d^k[2]$	$d^k[3]$	$d^k[4]$	$d^k[5]$	$d^k[6]$
k	0	1	2	5	2	4	-5

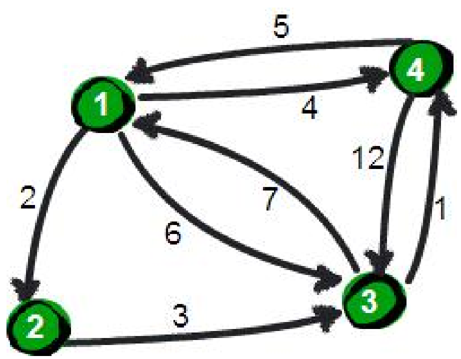
正常使用填空题需3.0以上版本雨课堂

作答

多源最短路径问题

Floyd算法（只有六行代码的算法）

- ✓ 寻找图中多源点之间（任意两点之间）最短路径的算法
- ✓ Robert W. Floyd于1962年发表



E=

	1	2	3	4
1	0	2	6	4
2	∞	0	3	∞
3	7	∞	0	1
4	5	∞	12	0

P=

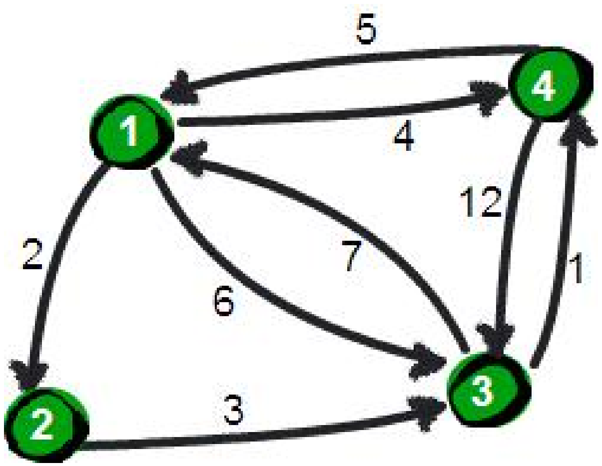
	1	2	3	4
1	1	2	3	4
2	1	2	3	4
3	1	2	3	4
4	1	2	3	4

- ✓ 引入第三顶点k，是否能缩短a到b的距离？ a->k->b
- ✓ k究竟是1到n中哪个呢？
- ✓ 引入多个顶点呢？ $a\text{->}k_1\text{->}k_2\text{....->}b$
- ✓ 如顶点4到顶点3，e[4][3]=12;
  引入节点1，e[4][1]+e[1][3]=5+6=11;
  再引入节点2，e[4][1]+e[1][2]+e[2][3]=5+2+3=10

多源最短路径问题

■ Floyd算法（只有六行代码的算法）

初始化：



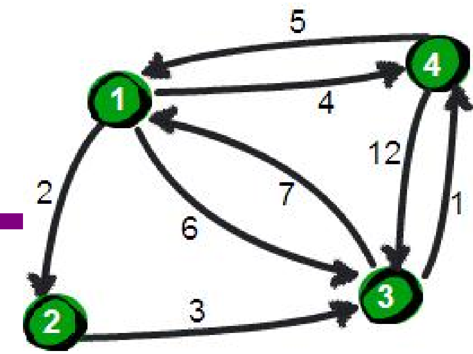
E=

	1	2	3	4
1	0	2	6	4
2	∞	0	3	∞
3	7	∞	0	1
4	5	∞	12	0

P=

	1	2	3	4
1	1	2	3	4
2	1	2	3	4
3	1	2	3	4
4	1	2	3	4

多源最短路径问题



■ Floyd算法（只有六行代码的算法）

✓ 若只允许经过1号顶点中转，求任意两点最小距离

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
         if (e[i][j]>e[i][1]+e[1][j]){
             e[i][j]=e[i][1]+e[1][j];
             p[i][j]=p[i][1];
        }
```

E=

	1	2	3	4
1	0	2	6	4
2	∞	0	3	∞
3	7	9	0	1
4	5	7	11	0

P=

	1	2	3	4
1	1	2	3	4
2	1	2	3	4
3	1	1	3	4
4	1	1	1	4

✓ 只允许经过1号和2号顶点中转的任意两点最小距离

```
for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        if (e[i][j]>e[i][1]+e[1][j]){
            e[i][j]=e[i][1]+e[1][j];
            p[i][j]=p[i][1];}

for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
        if (e[i][j]>e[i][2]+e[2][j]){
            e[i][j]=e[i][2]+e[2][j];
            p[i][j]=p[i][2];}
```

E=

	1	2	3	4
1	0	2	5	4
2	∞	0	3	∞
3	7	9	0	1
4	5	7	10	0

P=

	1	2	3	4
1	1	2	2	4
2	1	2	3	4
3	1	1	3	4
4	1	1	1	4

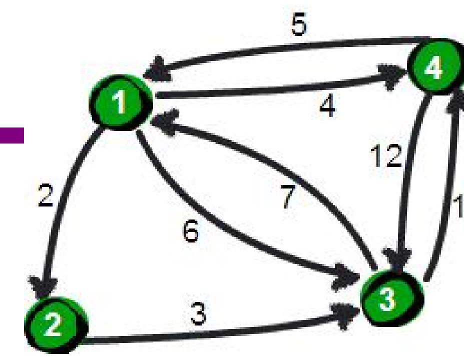
顶点4到3，查询P[4][3]得1，
再查询P[1][3]得2，再查询P[2][3]得3，
故最短路径为4->1->2->3

多源最短路径问题

Floyd算法（只有六行代码的算法）

- ✓ 疑问：方案B（先寻找中转2，再寻找中转1），是否能获得与方案A（先寻找中转1，再寻找中转2）相同的正确的4到3的最短路径4->1->2->3?
- ✓ 可以找到！原因如下两点
- ✓ 在方案B进行搜索2中转时，因4->1->2->3为最短路，为此得到1到3的最短路径必经过2，此时E_{13}更新为E_{123}
- ✓ 在方案B进行搜索1中转时,因方案A进行1搜索时已证明4到3的最短路径经过1，基于这结论，方案B搜索1也会得到4到3的最短路径经过1，为此E_{43}更新为$E_{413}=E_{41}+E_{13}=E_{41}+E_{123}=E_{4123}$，所以此时可得到4到3最短路径代价为路径4->1->2->3的代价，即找到该最短路

多源最短路径问题



Floyd算法（只有六行代码的算法）

✓ 为此，最终算法如下

```
for (k = 1; k <= n; k++)
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            if (e[i][j]>e[i][k]+e[k][j]){
                e[i][j]=e[i][k]+e[k][j];
                p[i][j]=p[i][k];
            }
```

k=3

E=

	1	2	3	4
1	0	2	5	4
2	10	0	3	4
3	7	9	0	1
4	5	7	10	0

P=

	1	2	3	4
1	1	2	2	4
2	3	2	3	3
3	1	1	3	4
4	1	1	1	4

k=4

E=

	1	2	3	4
1	0	2	5	4
2	9	0	3	4
3	6	8	0	1
4	5	7	10	0

P=

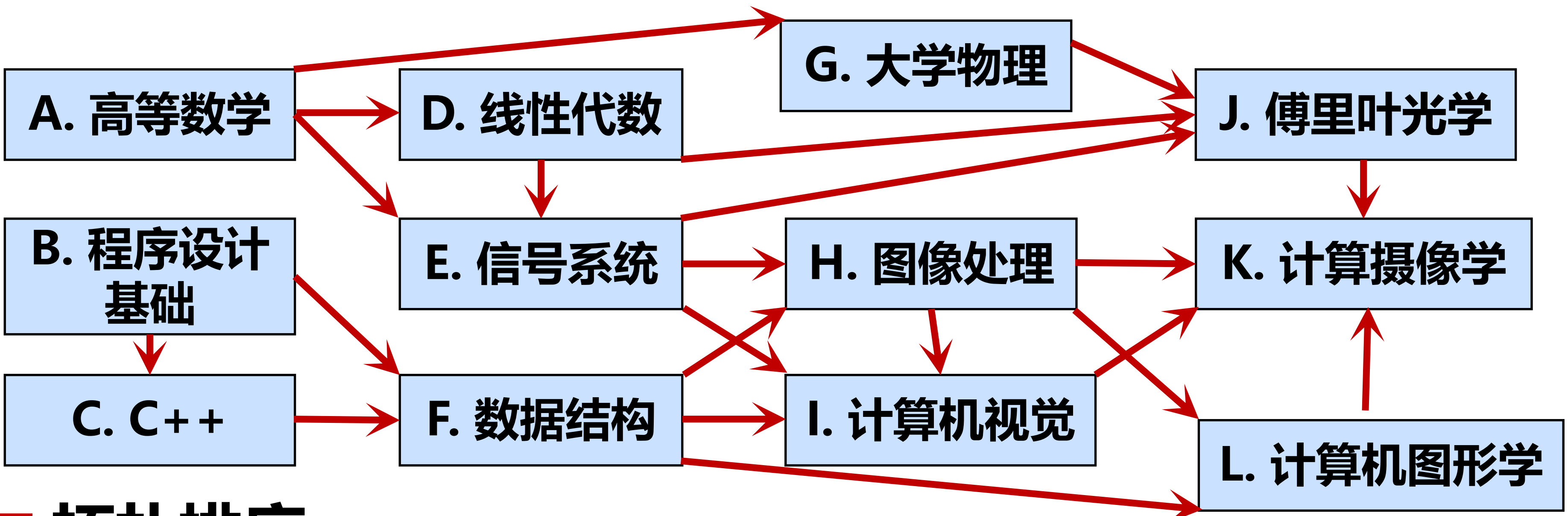
	1	2	3	4
1	1	2	2	4
2	3	2	3	3
3	4	4	3	4
4	1	1	1	4

例如，从2到1的最短路径，查P[2][1]得3，再查P[3][1]得4，再查P[4][1]得1，因此最短路径为2->3->4->1，代价为E[2][1]=9

拓扑排序

■ AOV(Activity on Vertex)网

✓ 顶点表示活动，有向边代表优先级，有向边起始端活动须早于末端活动



■ 拓扑排序

✓ 设G=(V,E)是一个具有n个顶点的有向图，若序列$v_1,\dots,v_n$满足G中每条有向边的活动时间先后要求，则称$v_1,\dots,v_n$为G的拓扑排序

✓ G存在拓扑排序必定保证G无环

✓ G的拓扑排序不唯一：ABCDEFGHIJLK为上图的其中一个拓扑排序

拓扑排序

■ 拓扑排序算法

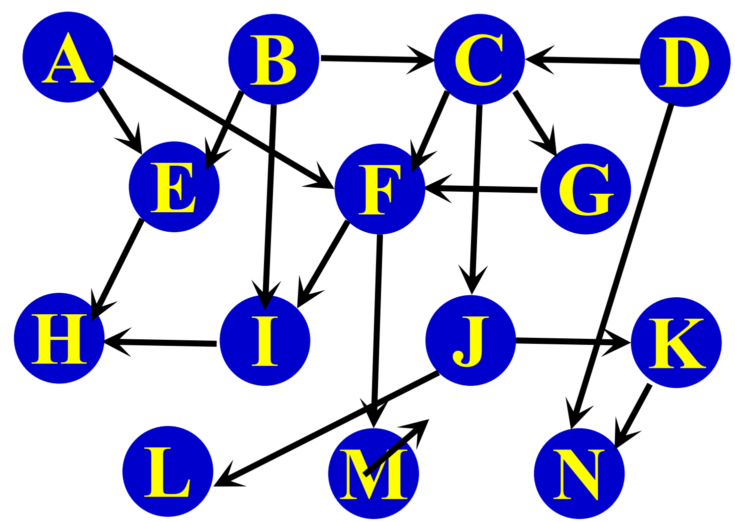
- ✓ 在AOV网络中选一个没有直接前驱的顶点, 并放入已排序集TS; (选取最优)
- ✓ 从图中删去该顶点, 同时删去所有它发出的有向边，更新邻域入度; (邻域更新)
- ✓ 重复以上直到

a) 全部顶点均已输出，拓扑有序序列形成，拓扑排序完成；或

b) 图中还有未输出的顶点(所有剩余顶点入度都不为0)，这时网络中必存在有向环

D A B C E G F

I H M J L K N



拓扑排序

拓扑排序算法

- 在AOV网络中选一个没有直接前驱的顶点, 并放入已排序集TS; (选取最优)
- 从图中删去该顶点, 同时删去所有它发出的有向边，更新邻域入度; (邻域更新)
- 重复以上直到
  - a) 全部顶点均已输出，拓扑有序序列形成，拓扑排序完成；或
  - b) 图中还有未输出的顶点(所有剩余顶点入度都不为0)，这时网络中必存在有向环

优先级搜索

顶点邻域优先级更新 ⟷ 迭代 ⟷ 选取最高优先级顶点入TS集

顶点优先值为该顶点的入度，每次对顶点邻域的入度降1

采用栈结构降低选取复杂度，所有入度为0的顶点入栈

拓扑排序

拓扑排序算法实现

```
bool Graph<Tv, Te>::TS () {
    Stack<int> S;
    for(int i=0; i<n; i++){
        status ( s ) = UNDISCOVERED;
        if(V[i].inDegree==0) S.push(i);
            status ( s ) = DISCOVERED;
     }
    if(S.size()==0) return false;
    while ( !S.empty() ) {
        int s = S.pop();
        status ( s ) = VISITED;
        for ( int u = firstNbr ( s );
                -1 < u; u = nextNbr ( s, u ) )
            if ( UNDISCOVERED == status ( u ) )
                if((--V[u].inDegree)==0) S.push(u);
         }
    }
    for(int i=0; i<n; i++)
        if(status (s) == UNDISCOVERED) return false;
    return true;
}
```