迪杰斯特拉算法

- 引理8.5(**最短路径性质**): 在带权图G中，若从x到z的最短路径包含从x到y的路径P和从y到z的路径Q，那么P是从x到y的最短路径，Q是从y到z的最短路径。

Lemma 8.5 (Shortest path property) In a weighted graph G, suppose that a shortest path from x to z consists of path P from x to y followed by path Q from y to z. Then P is a shortest path from x to y, and Q is a shortest path from y to z. □

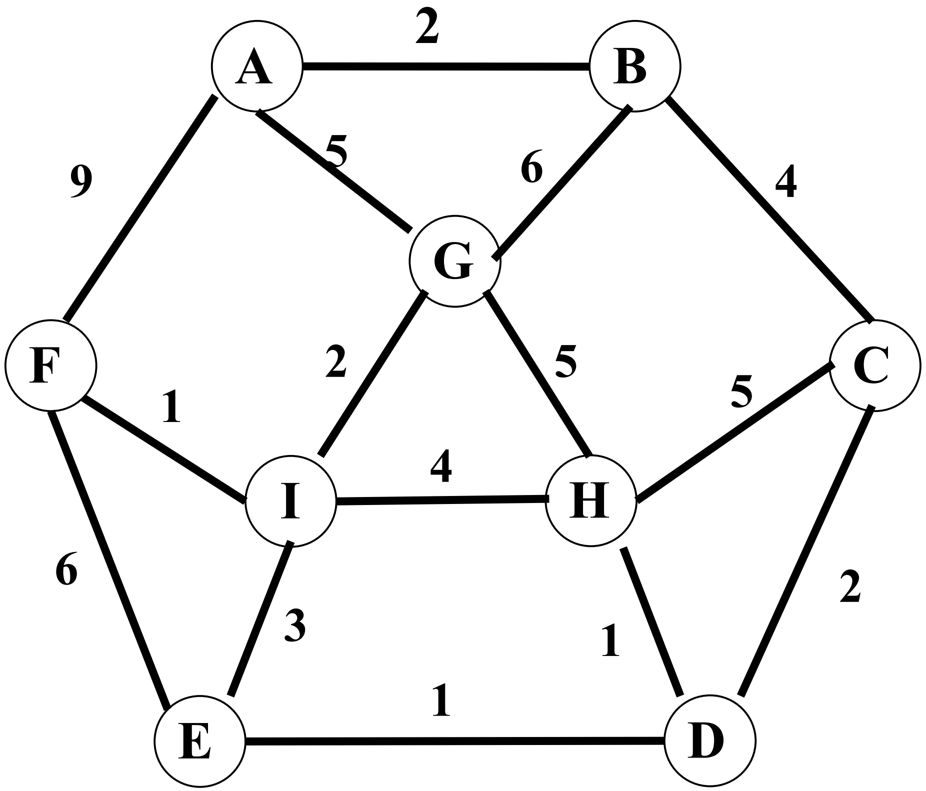
在带权图G中，若P是从x到y的最短路径，Q是从y到z的最短路径，P+Q是否为**从x到z的最短路径**？

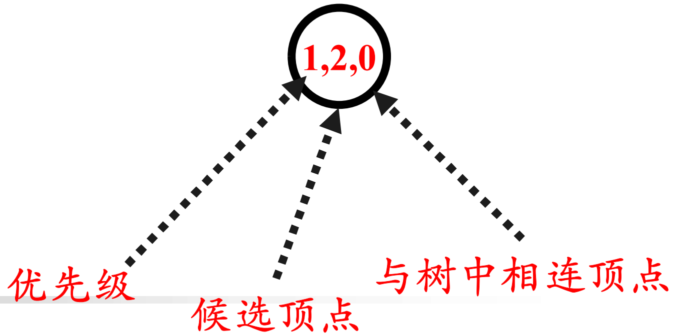
定义

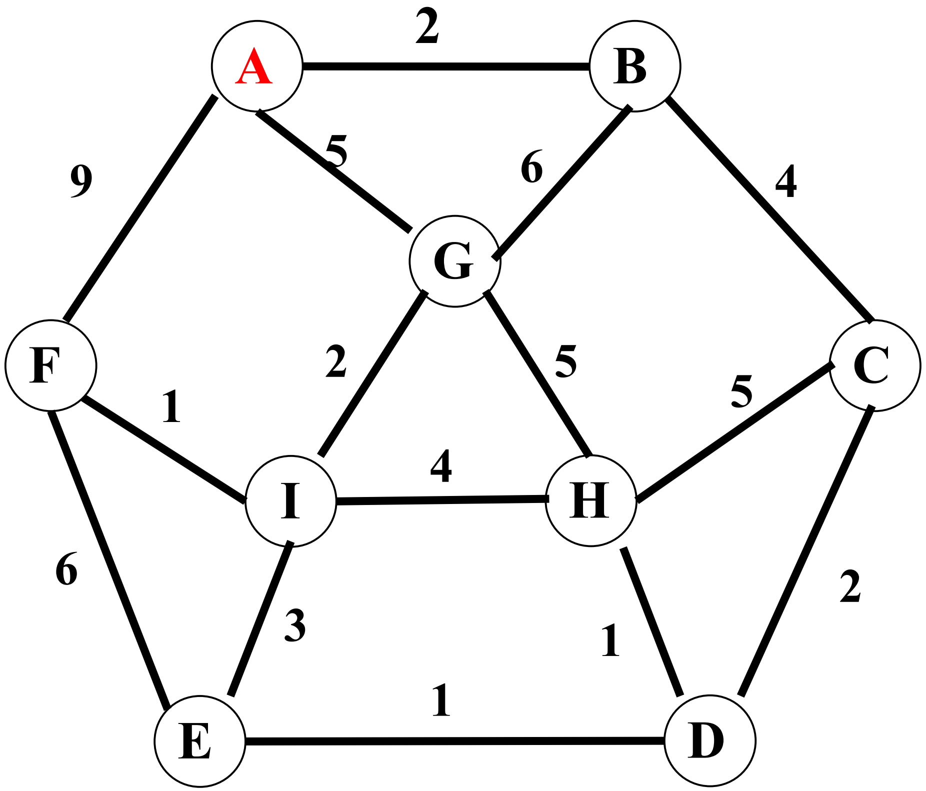
1. 边(y,z)的权值：w(yz)
2. 路径P=s, x_1, x_2, ⋯, x_r, y的路径长度w(P)= $w(sx_1)+w(x_1x_2)+\cdots+w(x_ry)$
3. 空路径长度为0

迪杰斯特拉算法

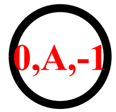
- 按路径长度递增的次序产生最短路径
- 设图G=(V,E), V={v_0, v_1,⋯, v_{n-1}},源点v_0,求v_0到其余各点的最短路径。
- 分析：设v_0到v_1,⋯, v_{n-1}的最短路径分别为P_1,P_2，⋯,P_{n-1}. 若这n-1条路径中最短的一条为P_i($1 \le i \le n-1$), 那么它一定是弧< v_0, v_i>；
- 若这n-1条路径中第二短的一条为P_j($1 \le i \ne j \le n-1$),那么它一定是弧< v_0, v_j>或者是路径< v_0, v_i> < v_i, v_j>；
- …….


A
B
C
D
E
F
G
H
I
2
9
5
6
4
2
5
5
1
4
6
3
1
1
2

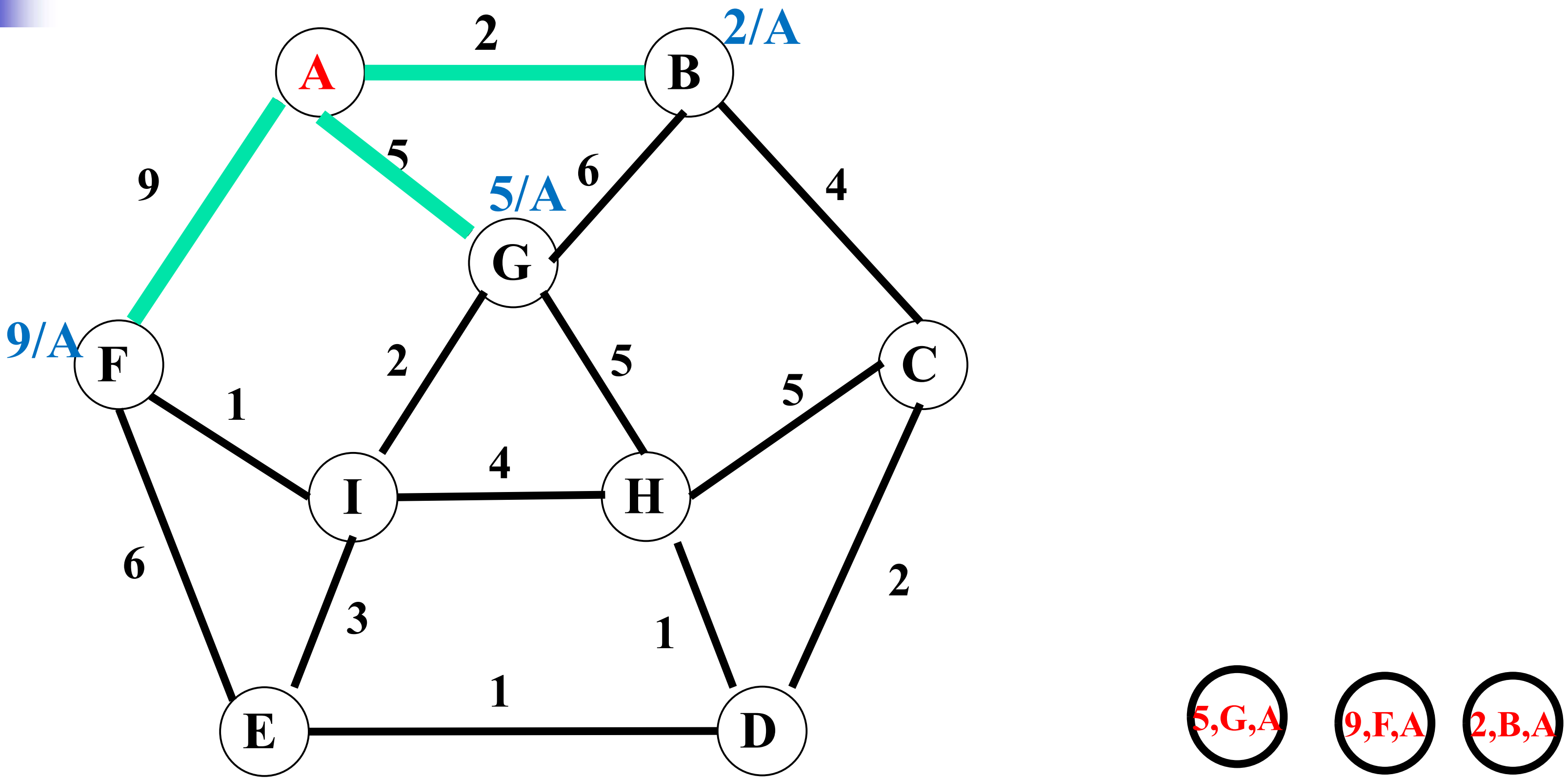




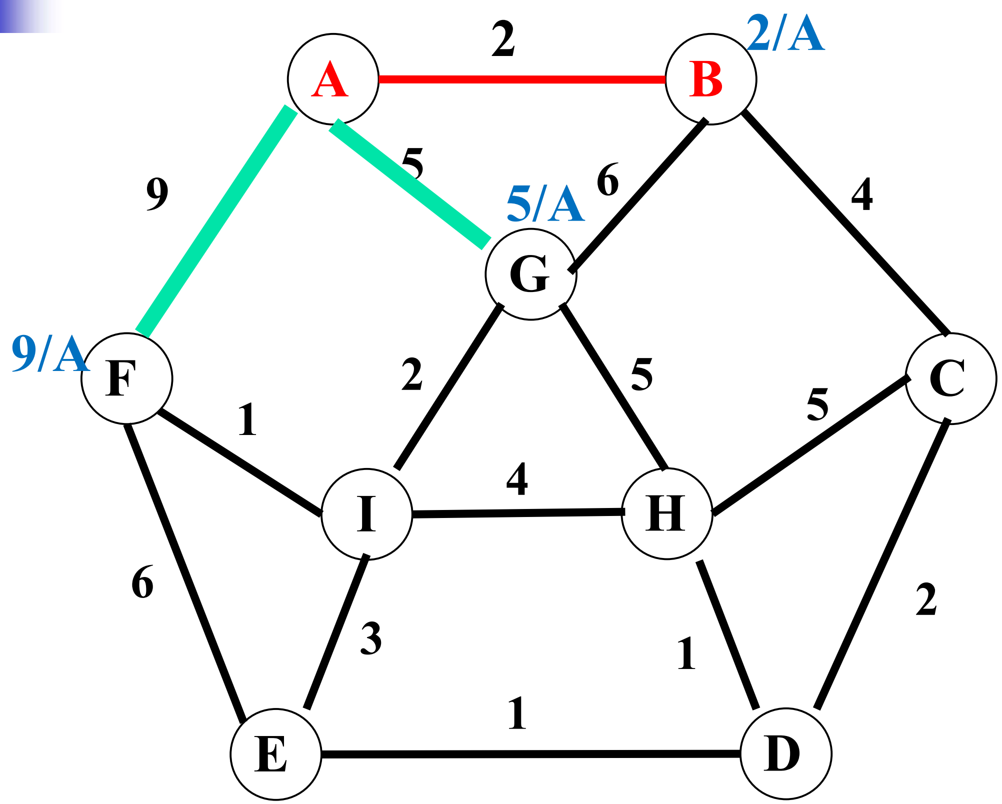
改用优先队列存放fringe 顶点



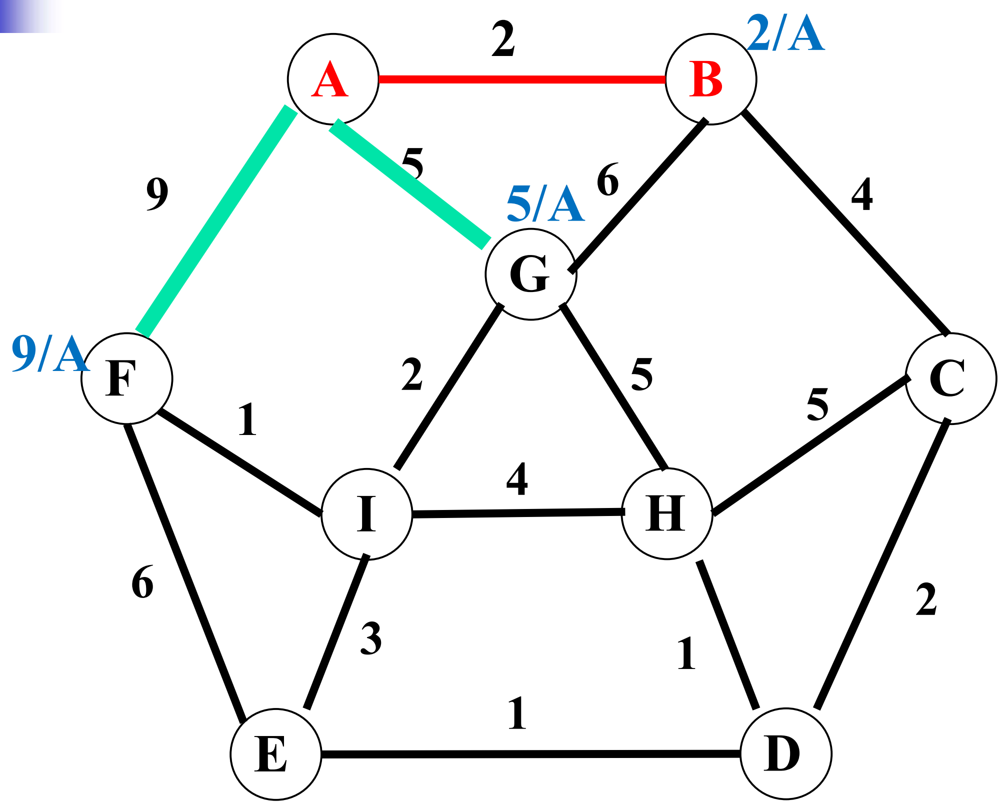
最小优先队列

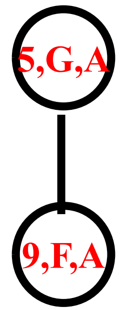


最小优先队列

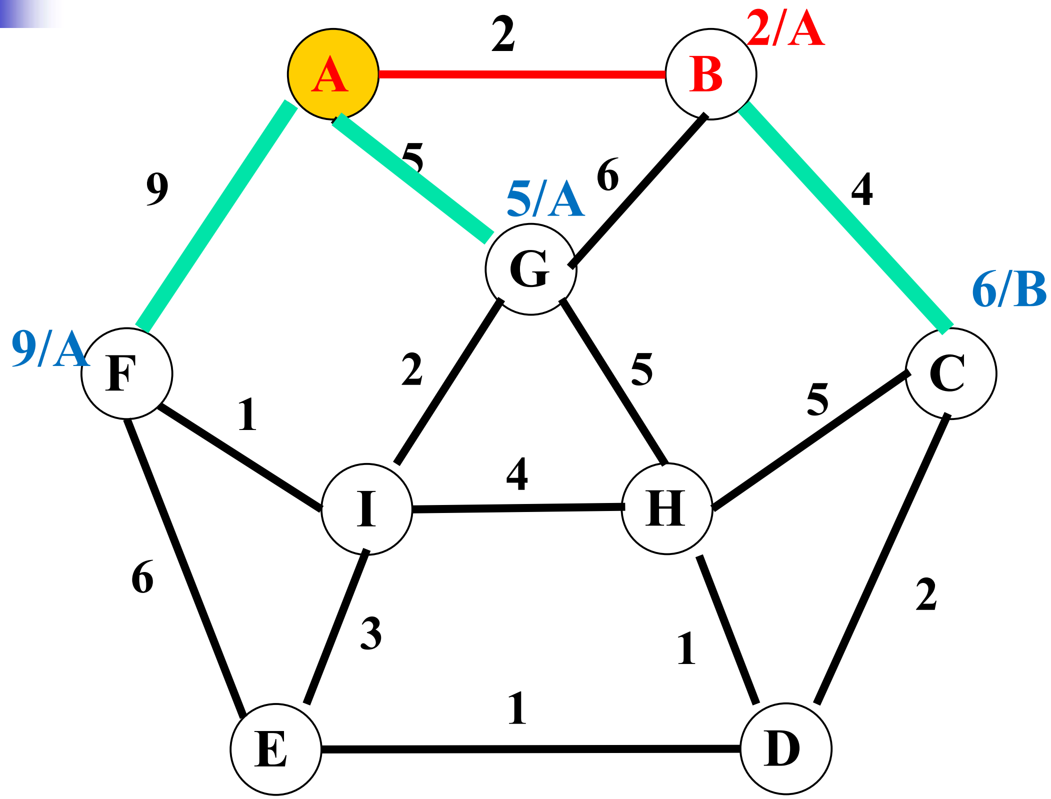


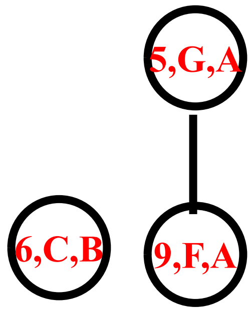
最小优先队列



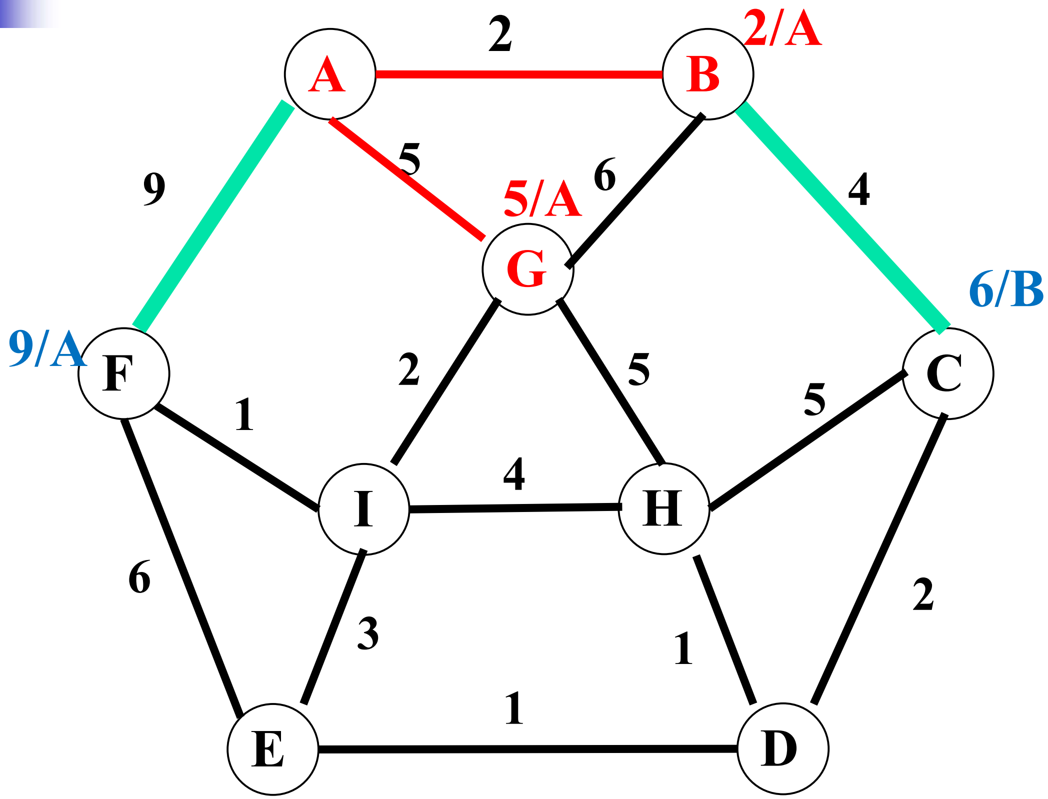


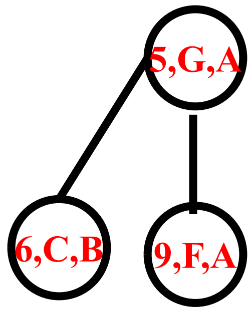
最小优先队列



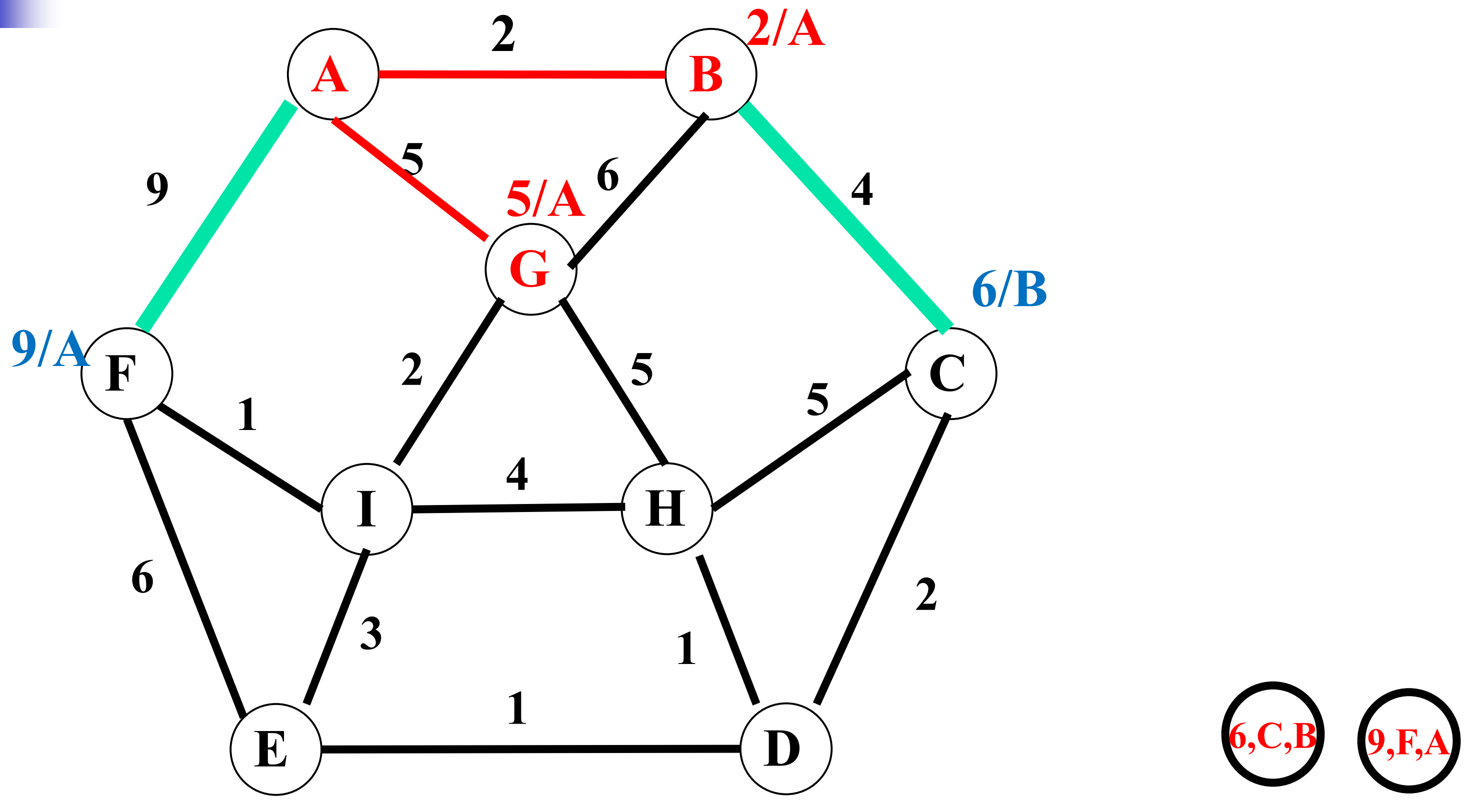


最小优先队列



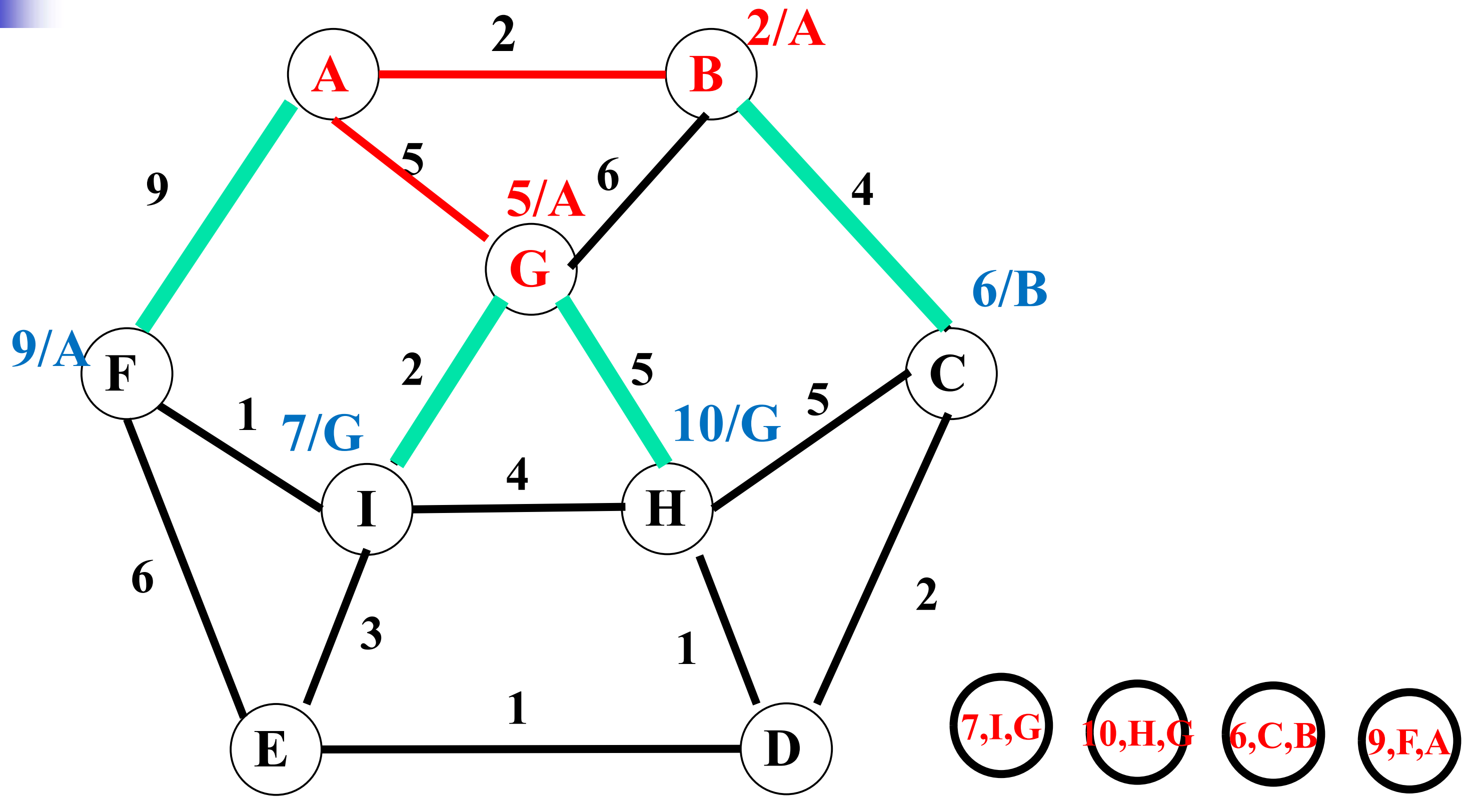


最小优先队列



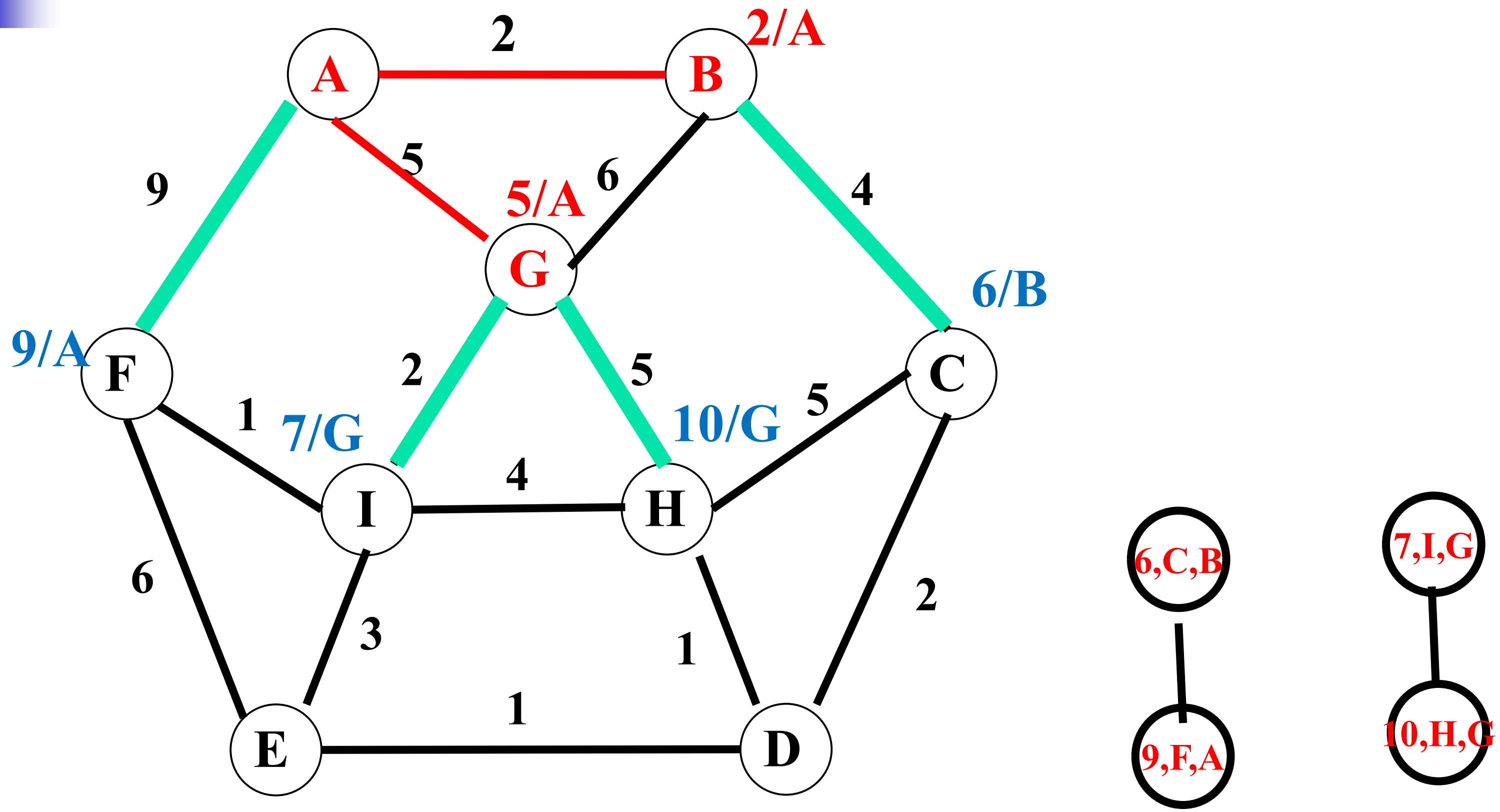
最小优先队列

1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.



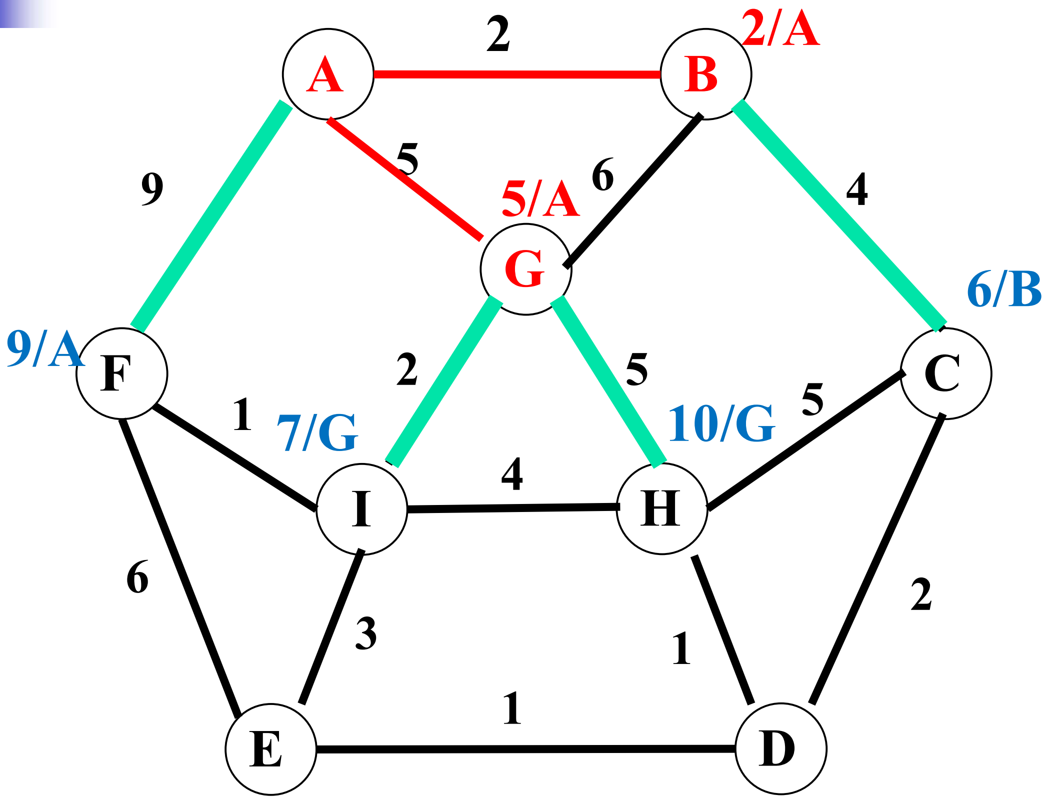
最小优先队列

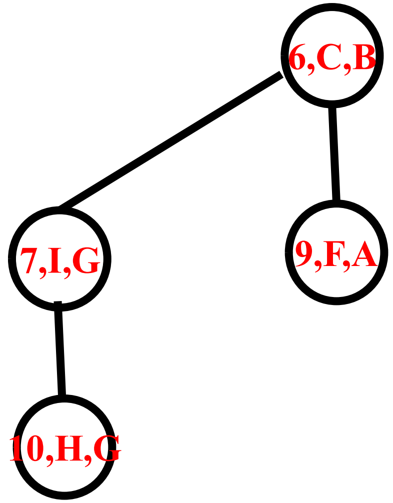
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.



最小优先队列

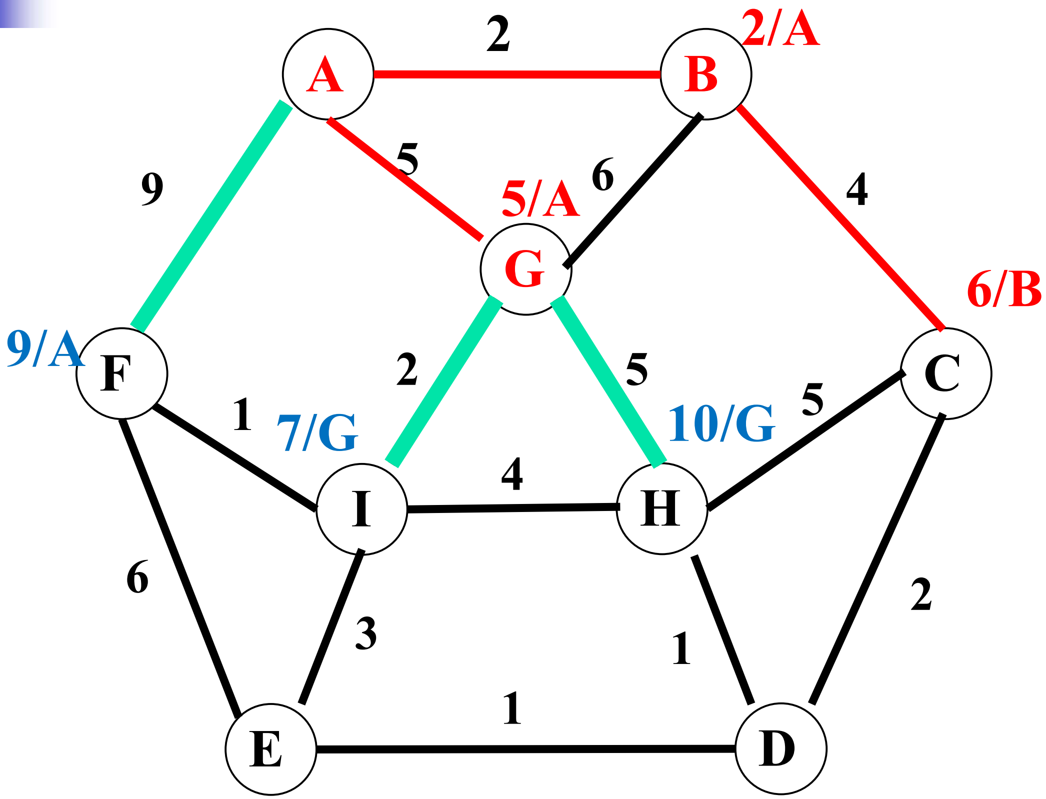
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

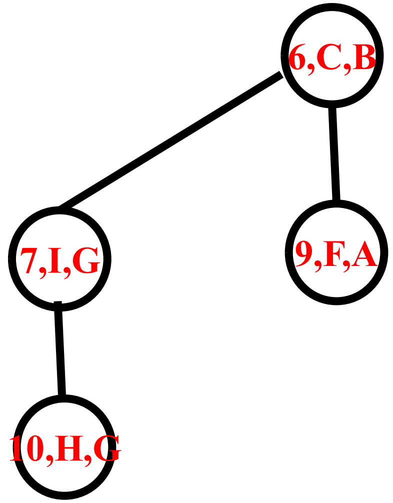




最小优先队列

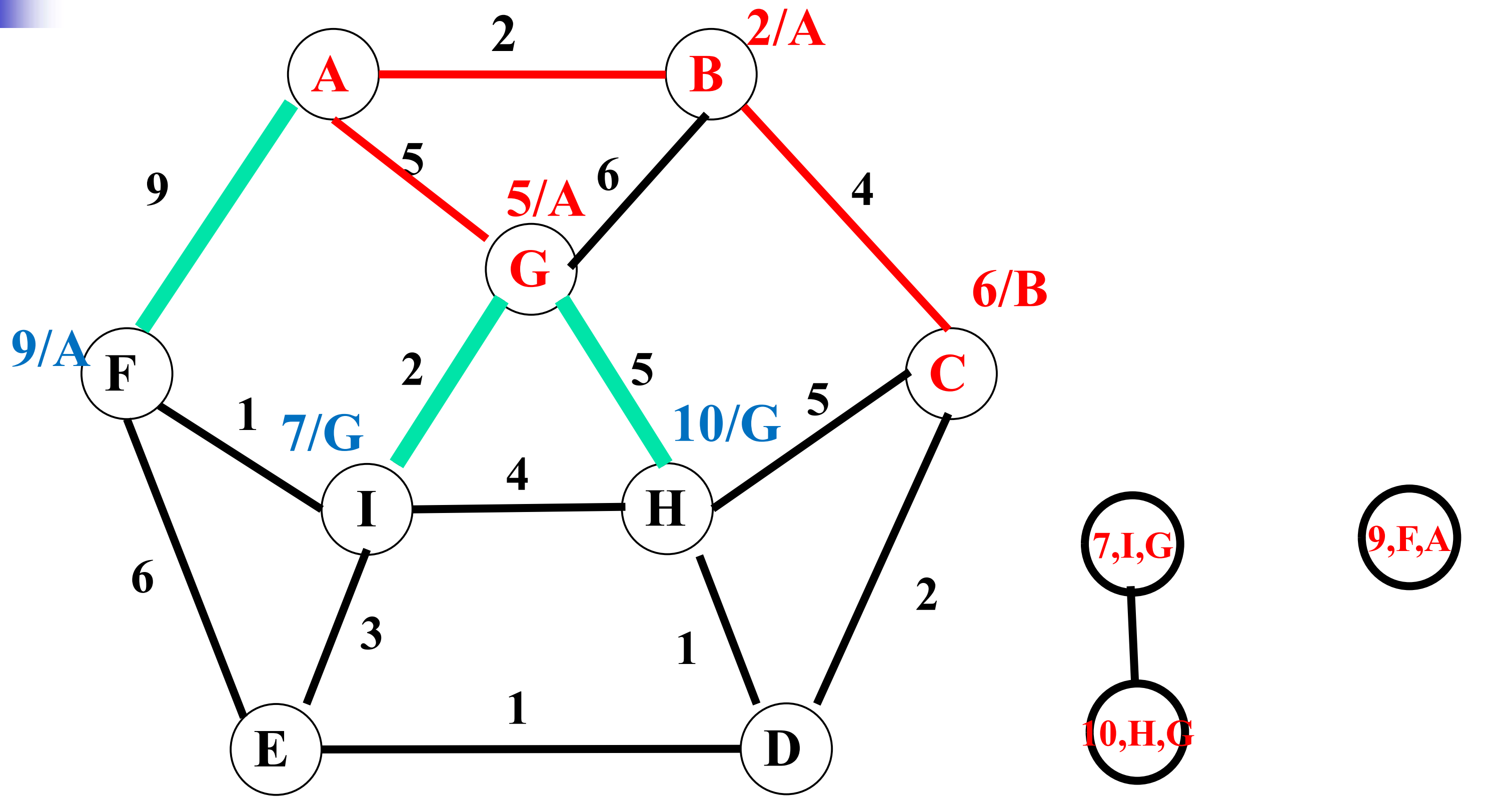
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.





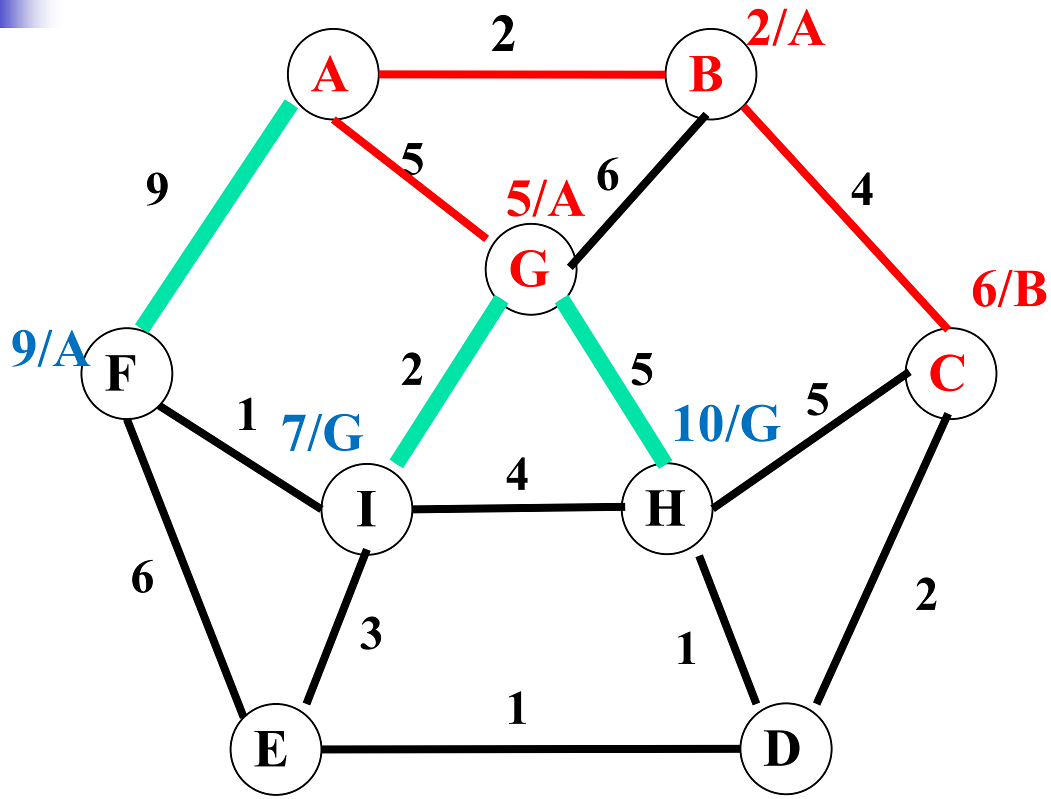
最小优先队列

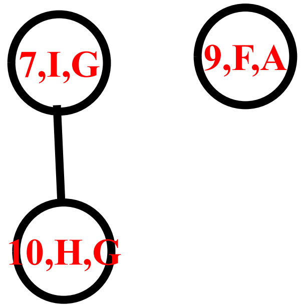
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.



最小优先队列

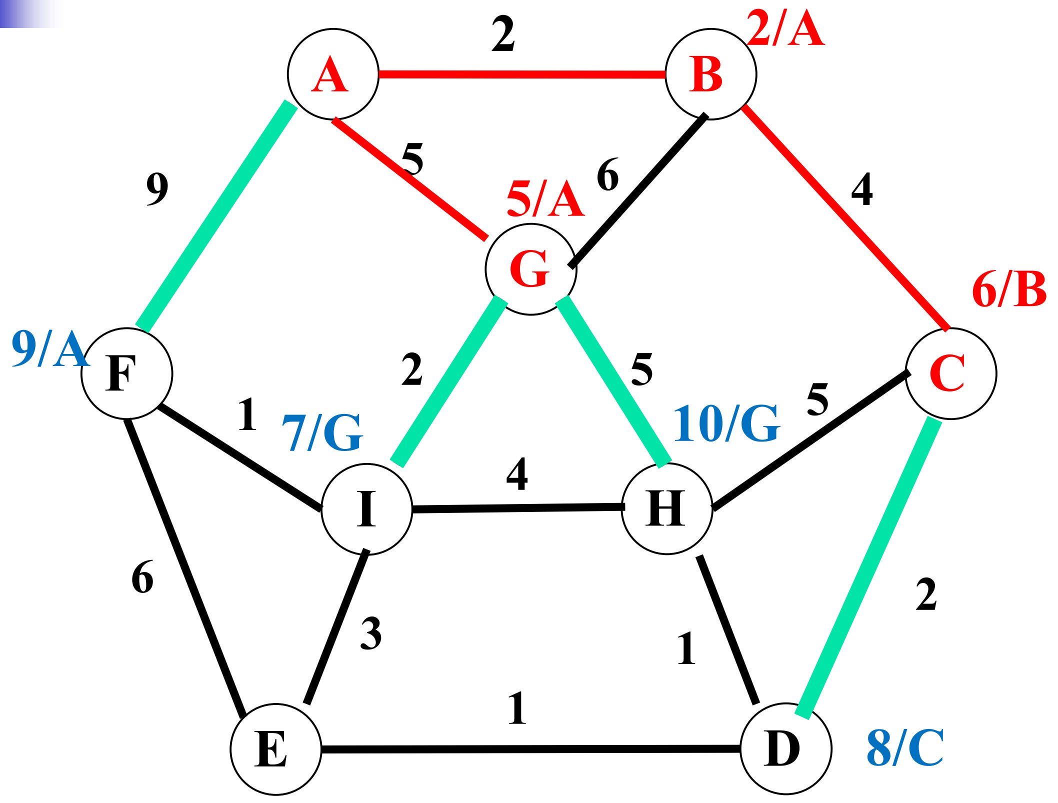
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

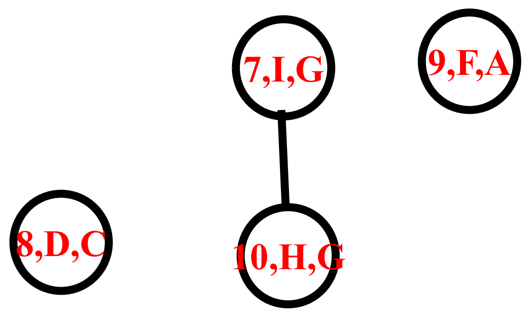




最小优先队列

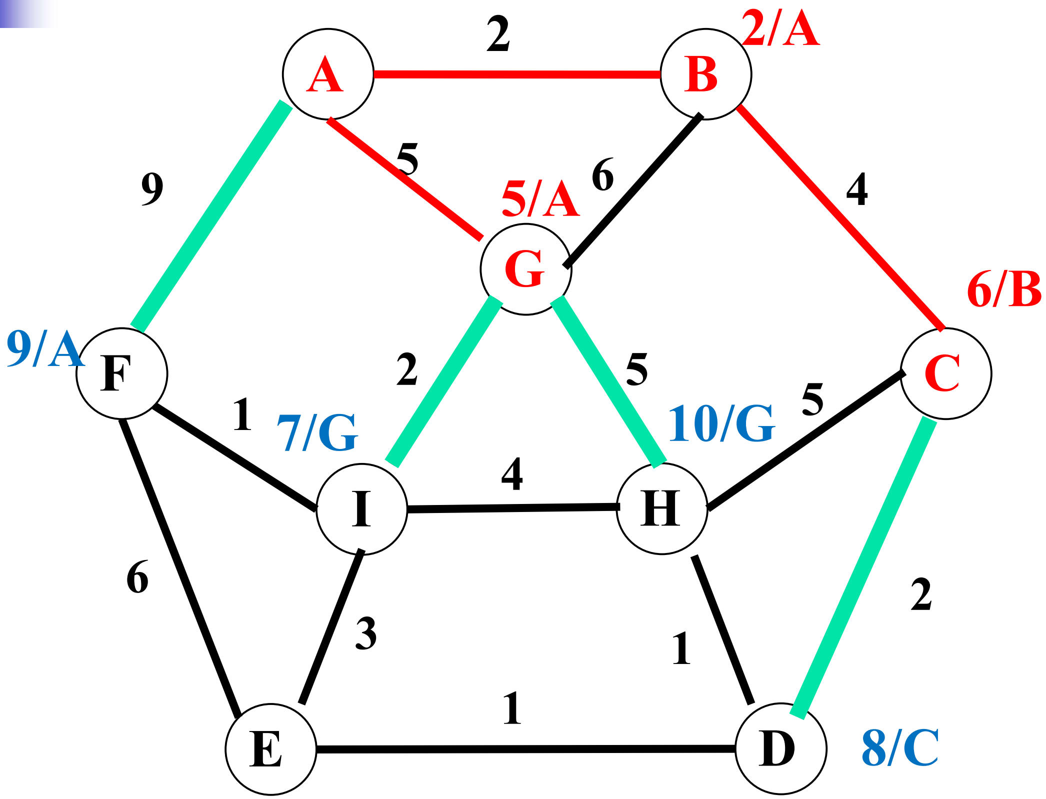
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

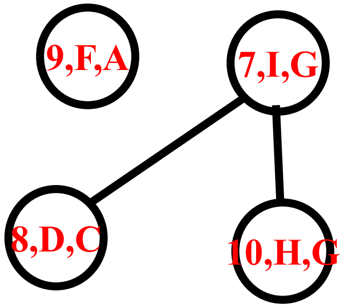




最小优先队列

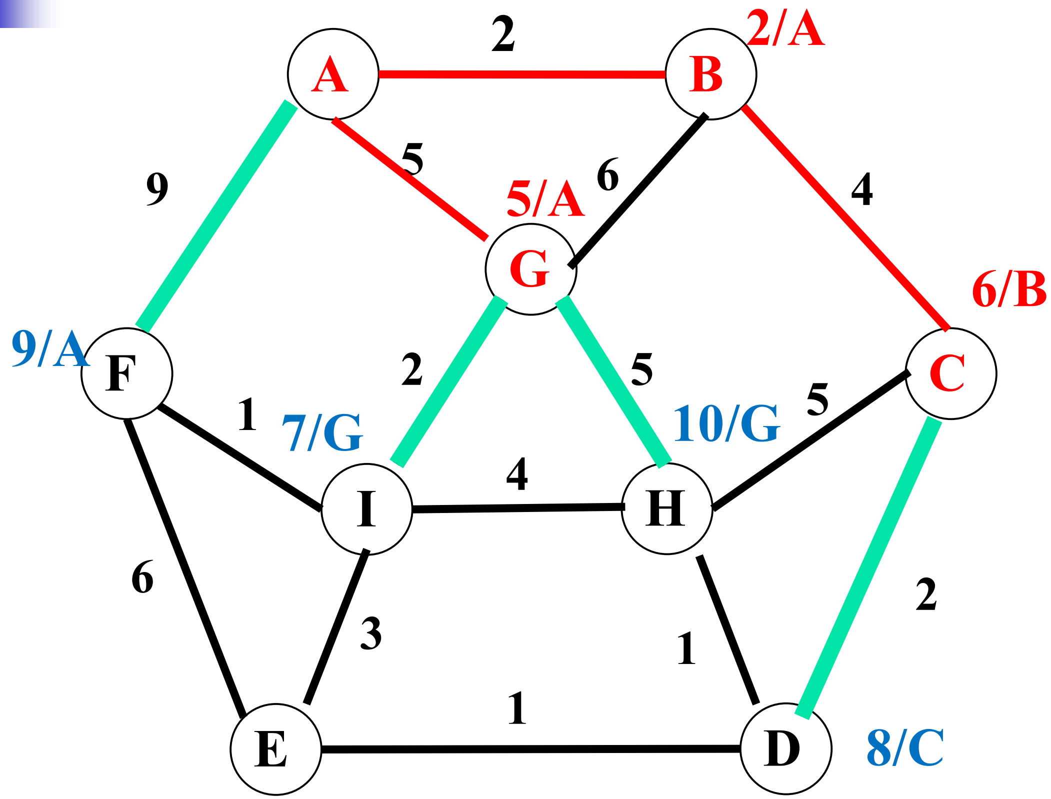
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

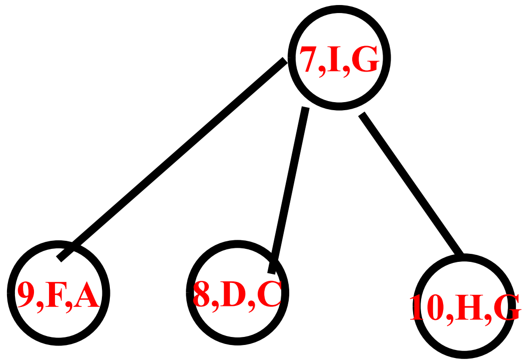




最小优先队列

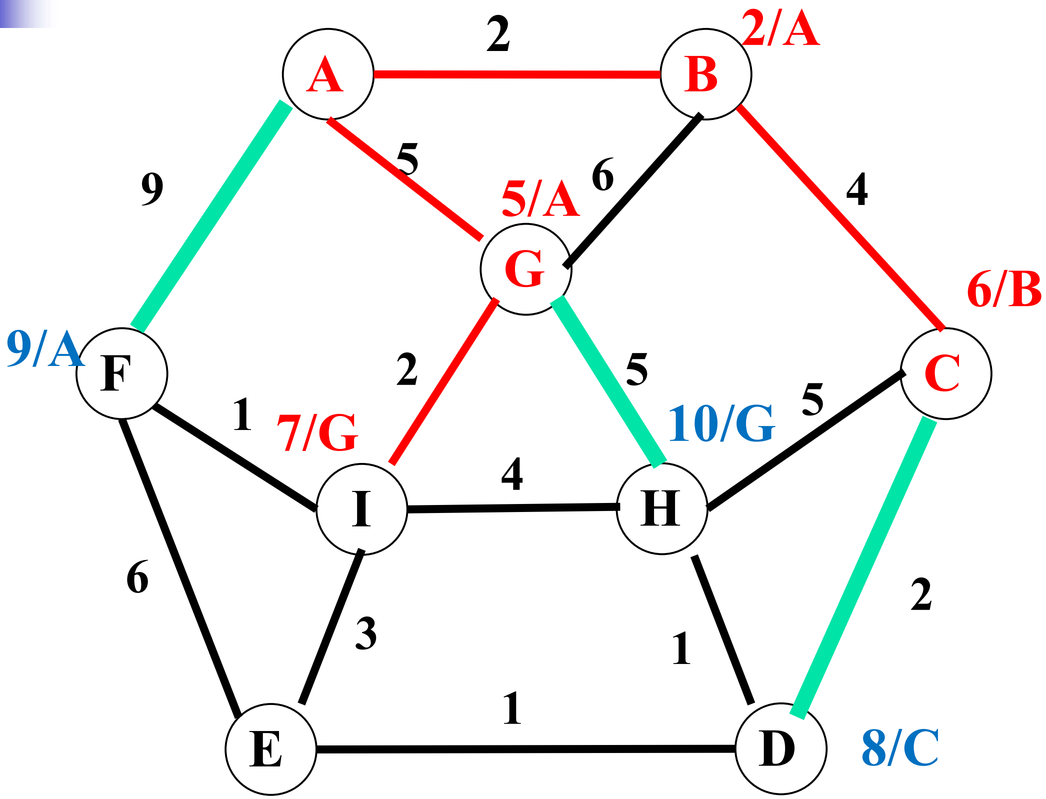
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

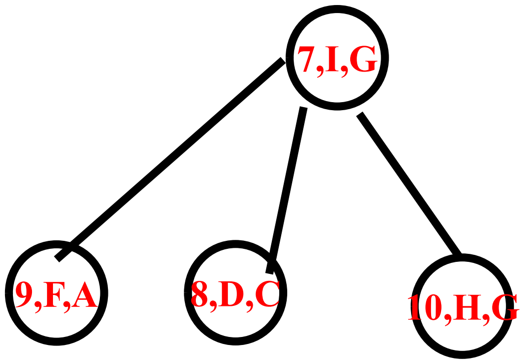




最小优先队列

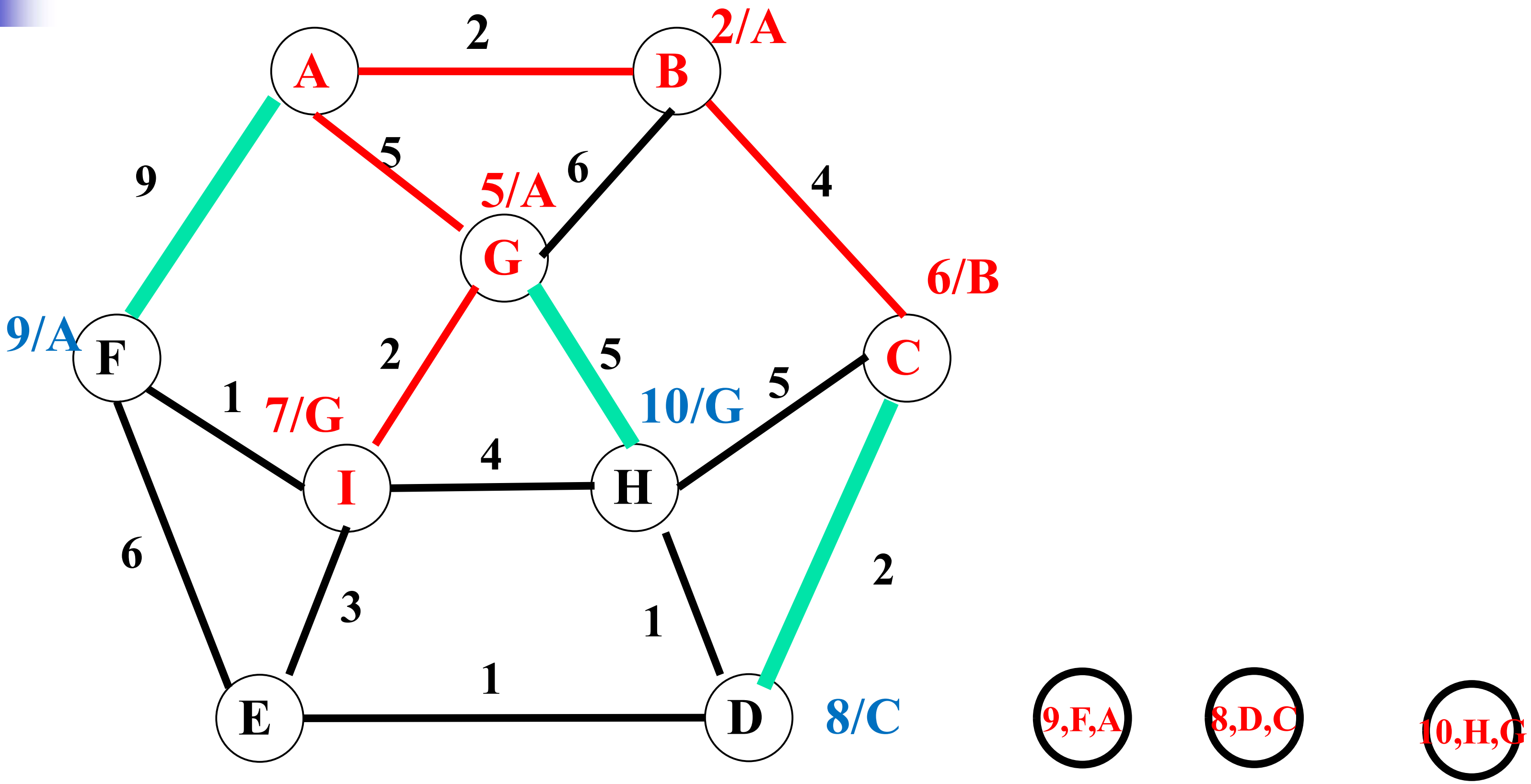
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.





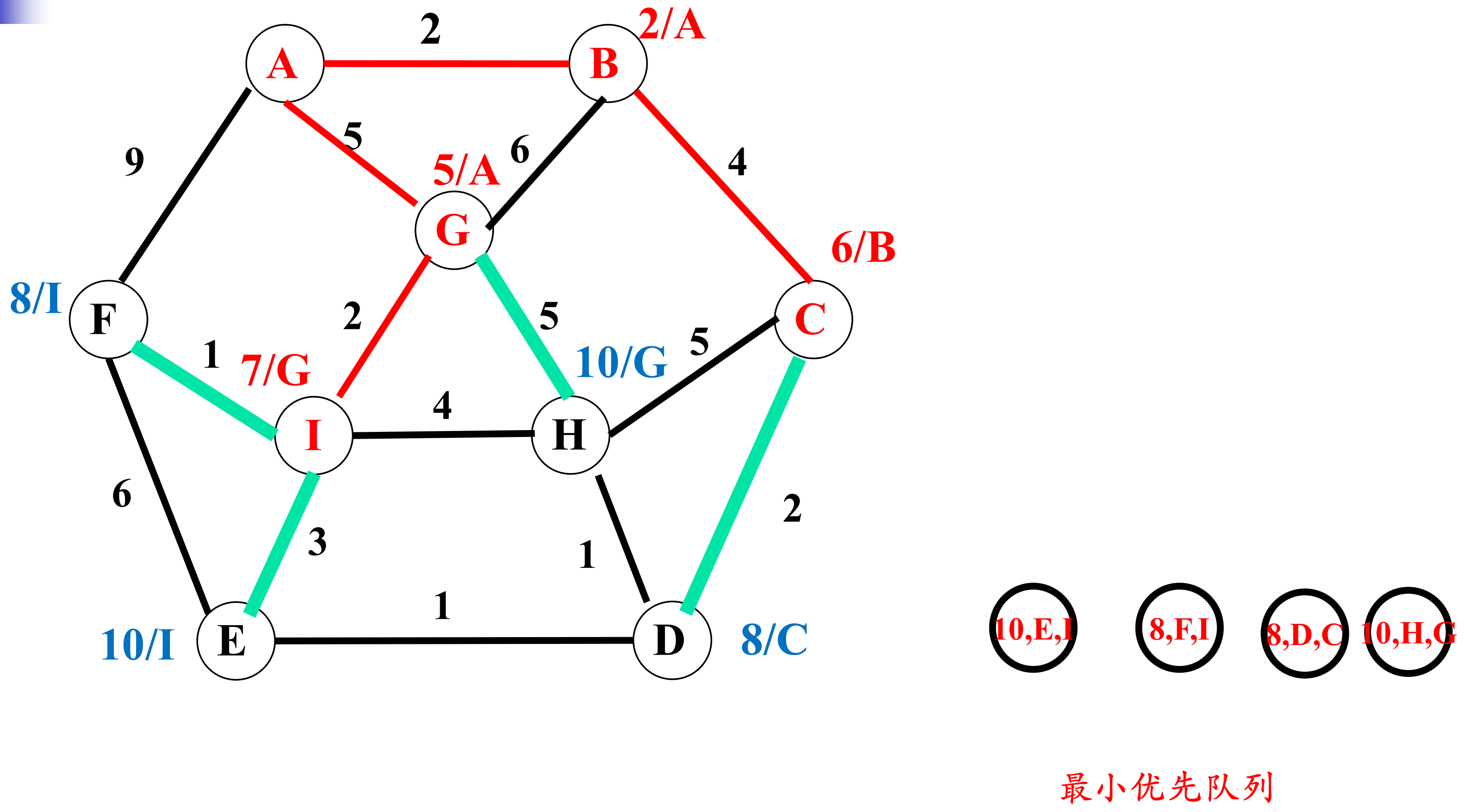
最小优先队列

1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

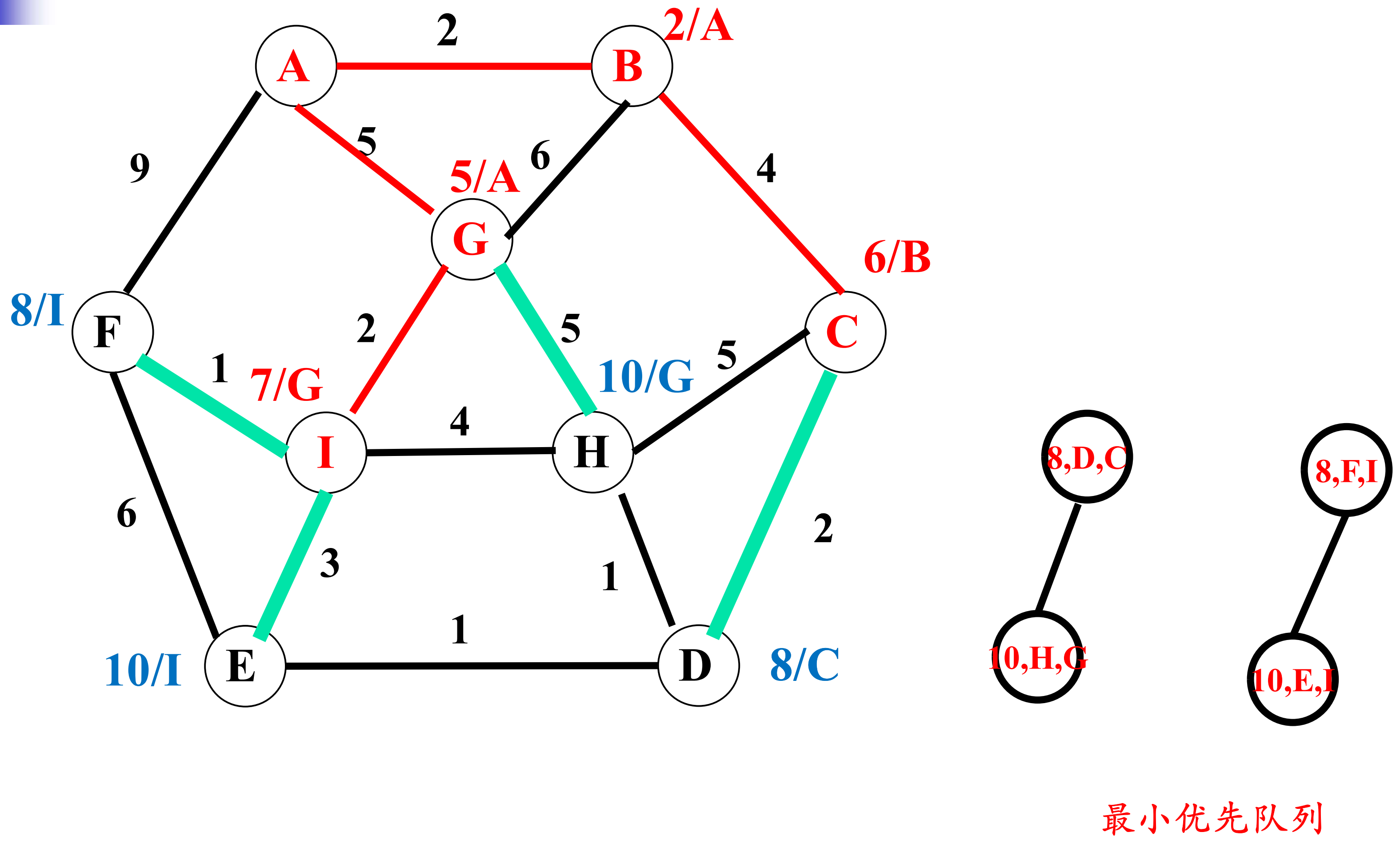


最小优先队列

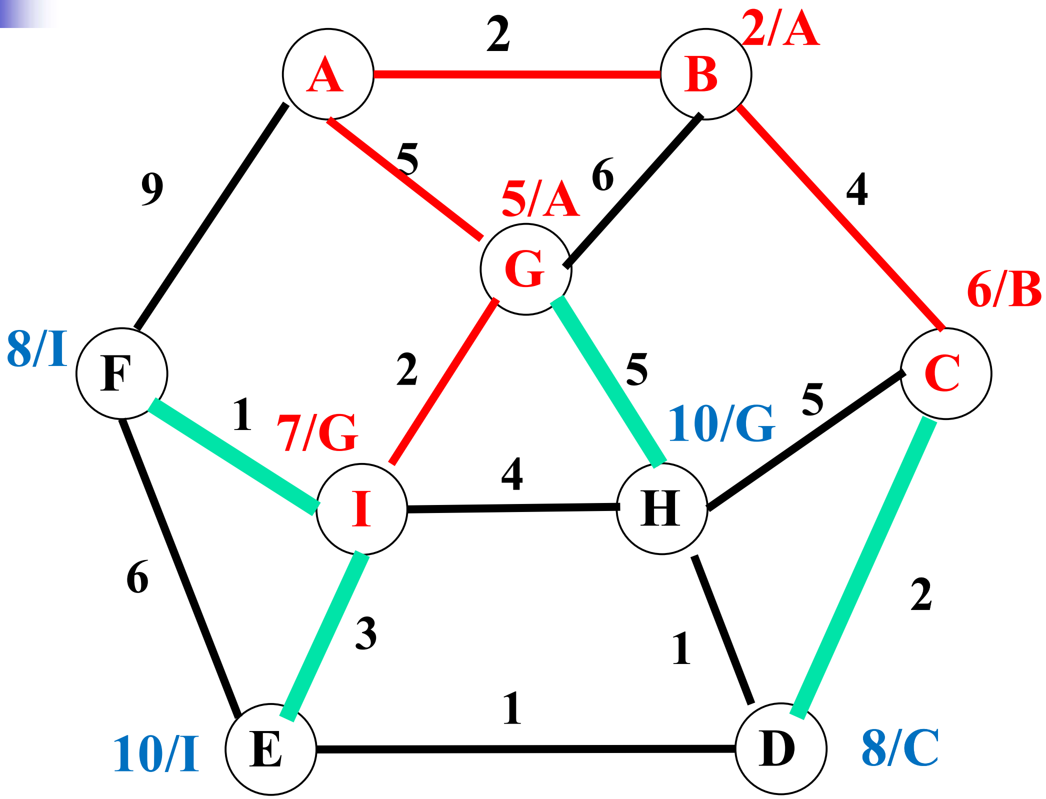
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

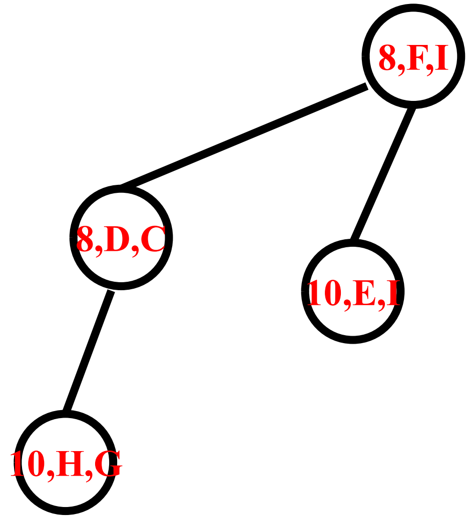


1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.



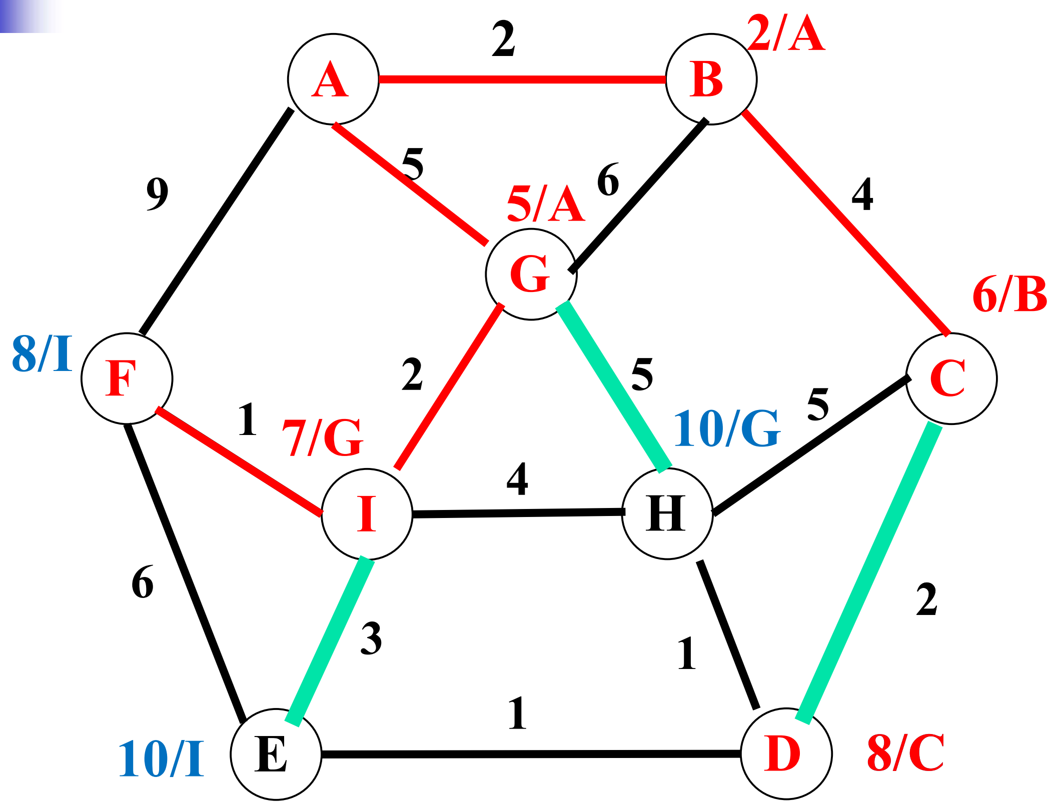
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

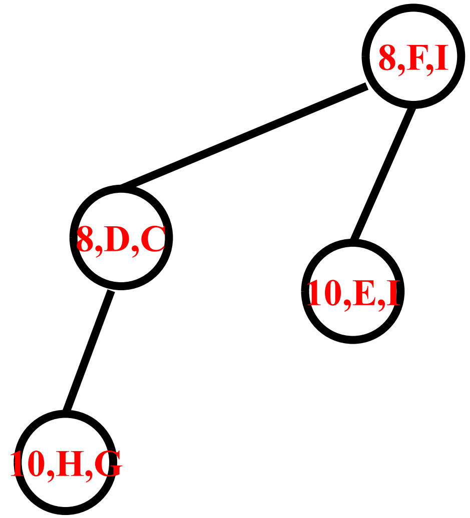




最小优先队列

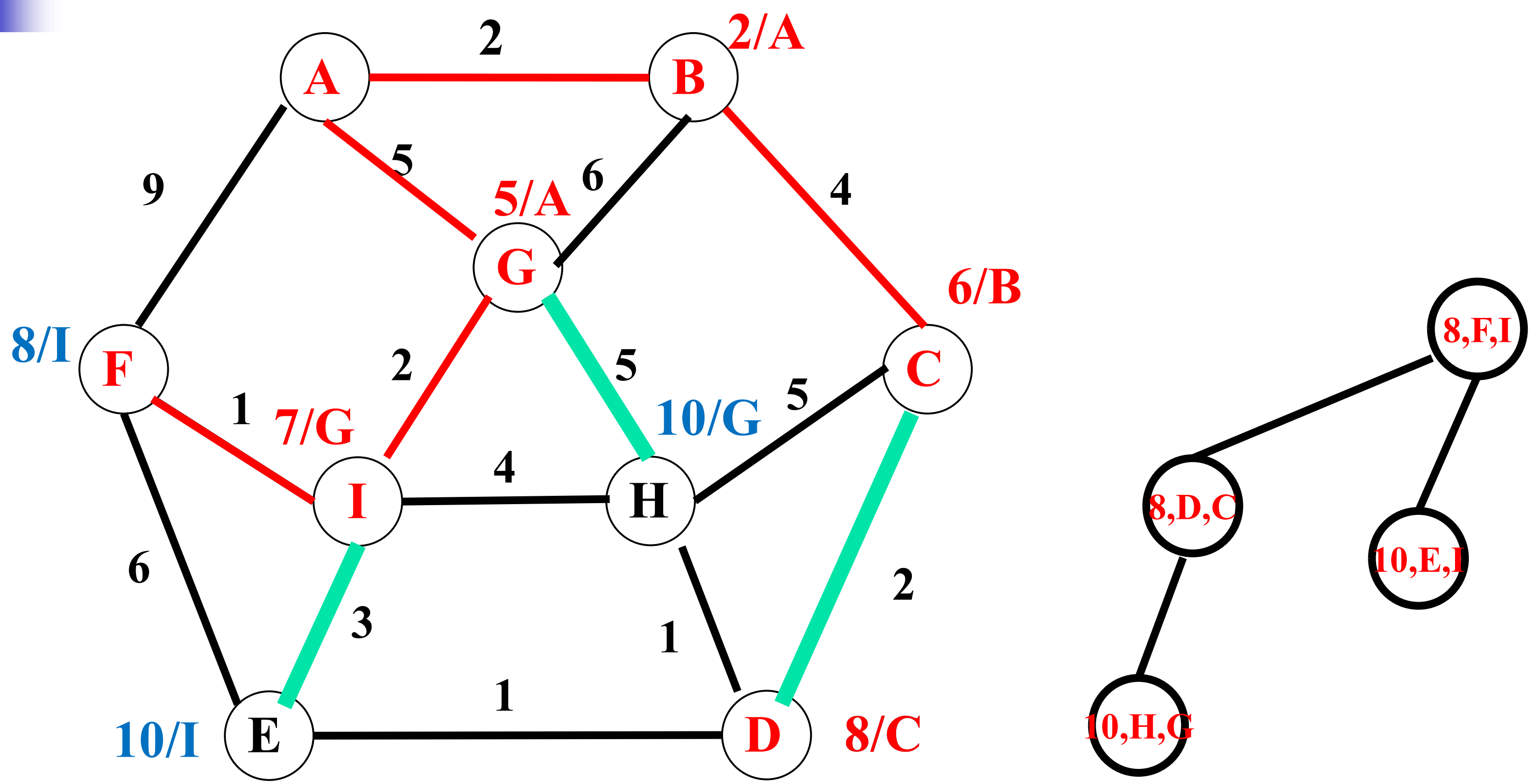
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.





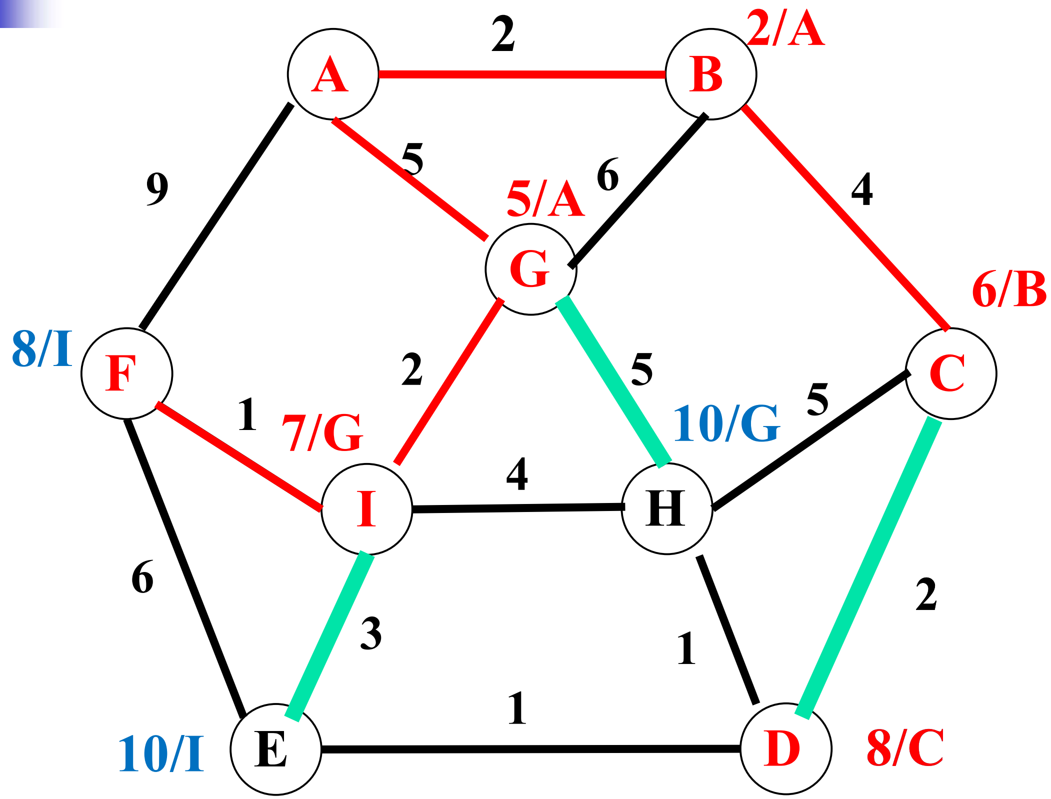
最小优先队列

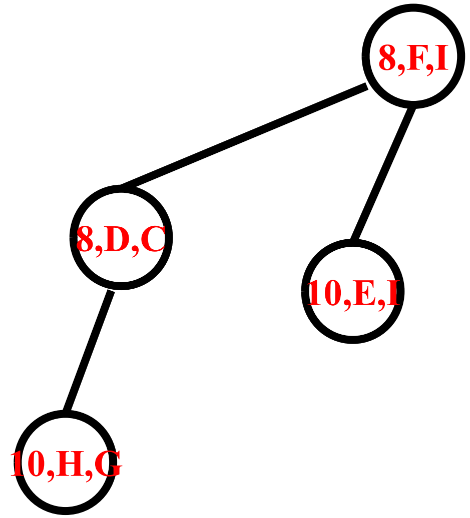
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.



最小优先队列

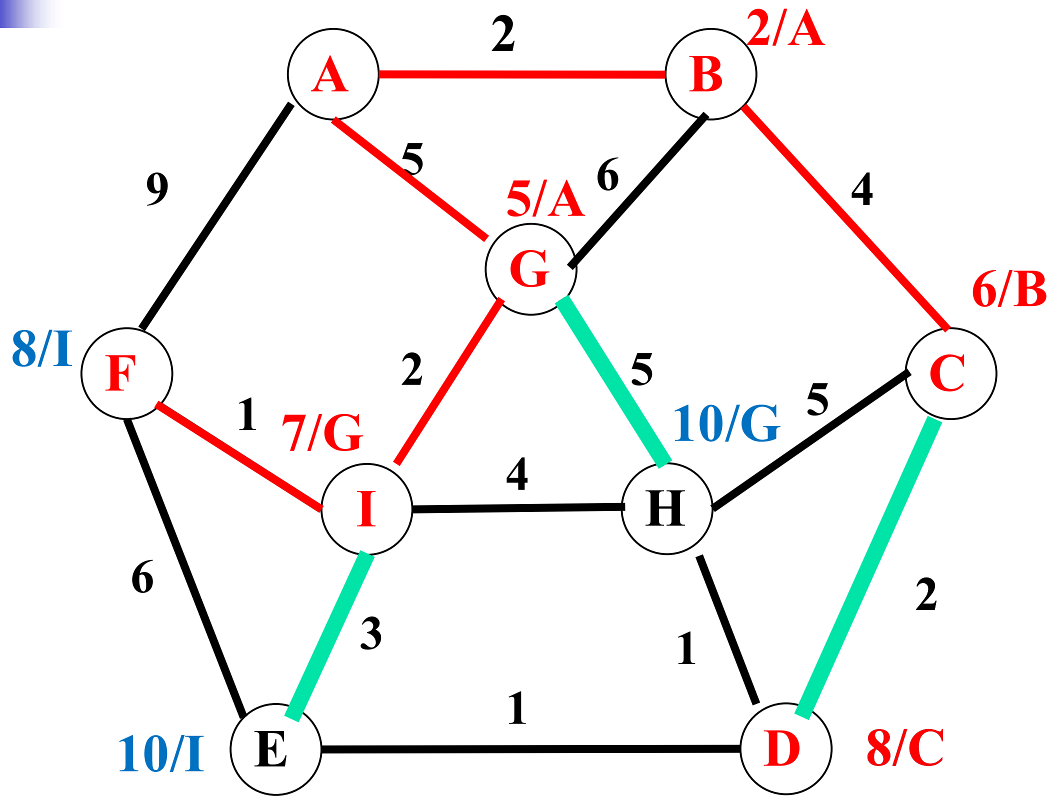
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

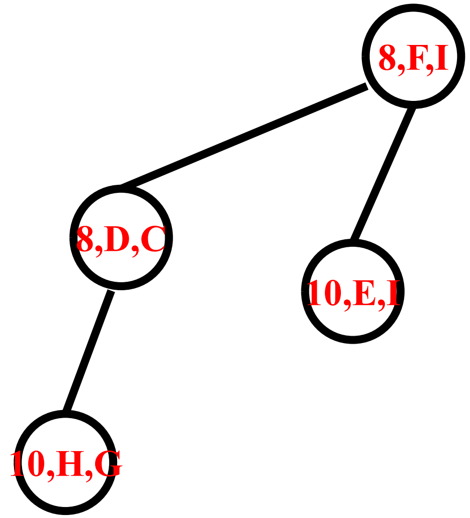




最小优先队列

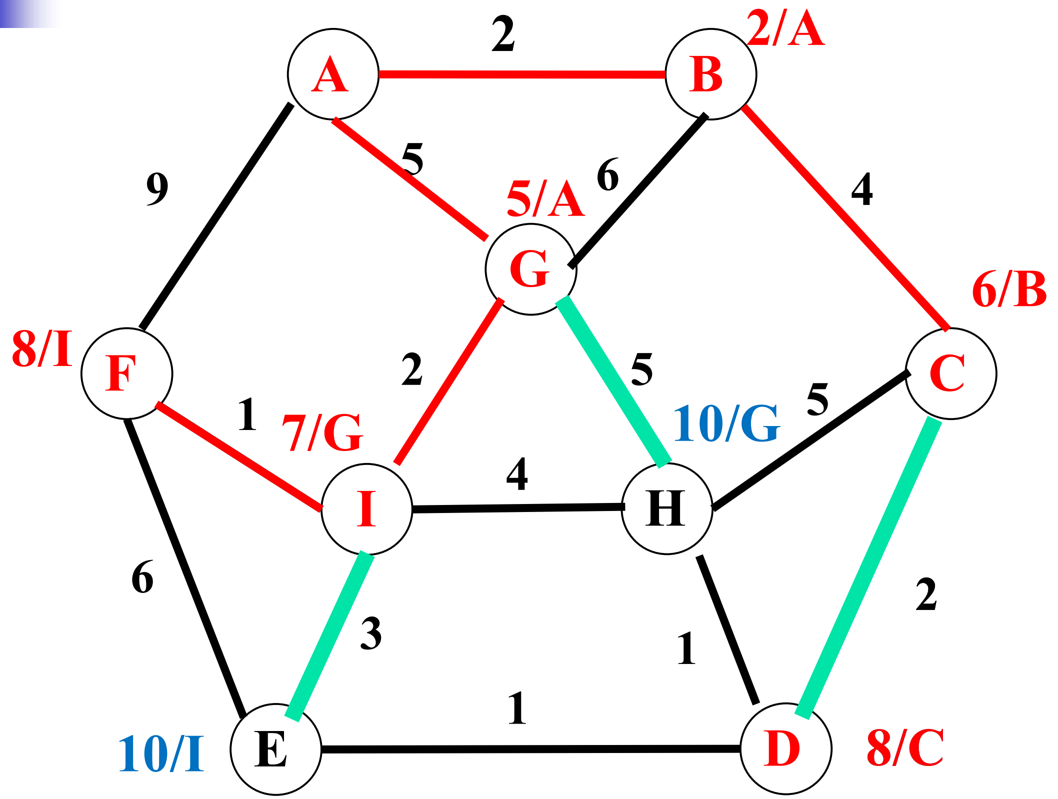
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

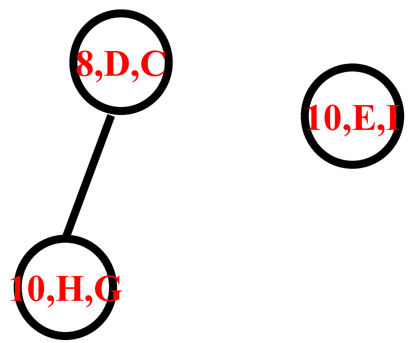




最小优先队列

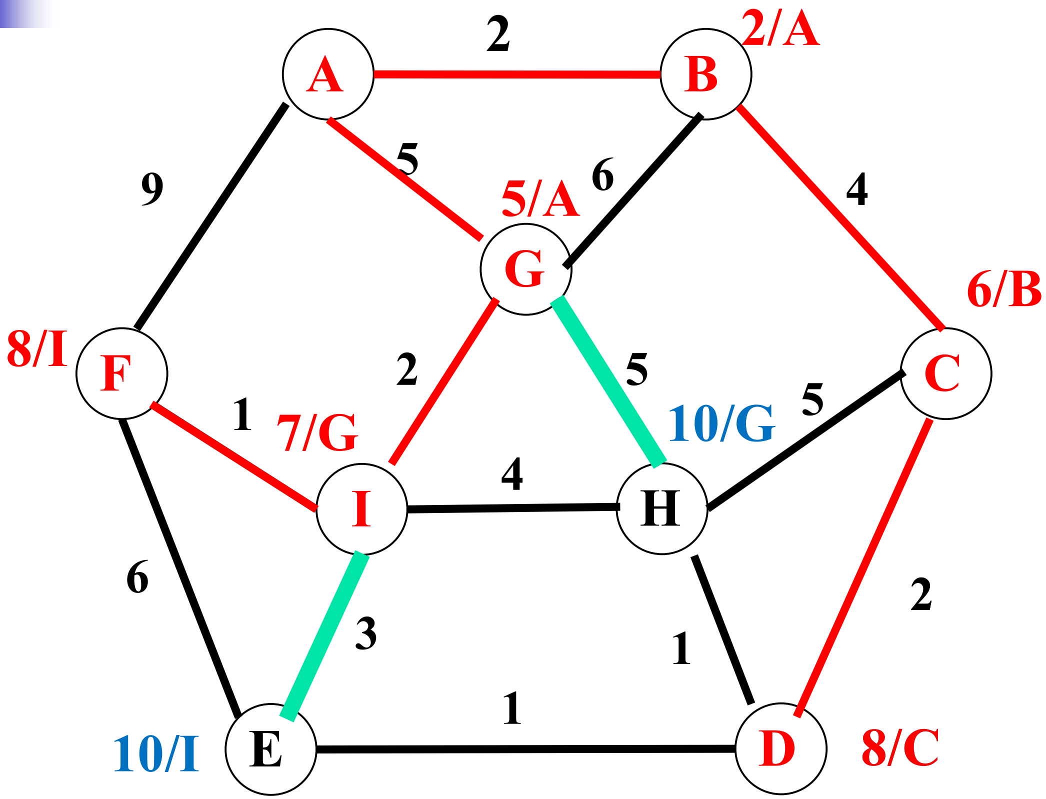
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

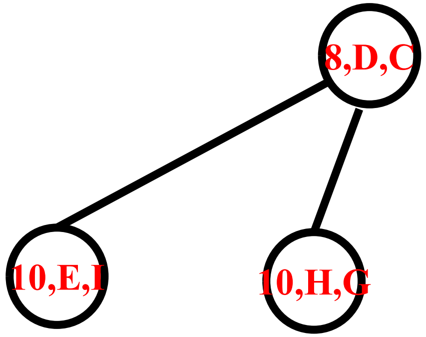




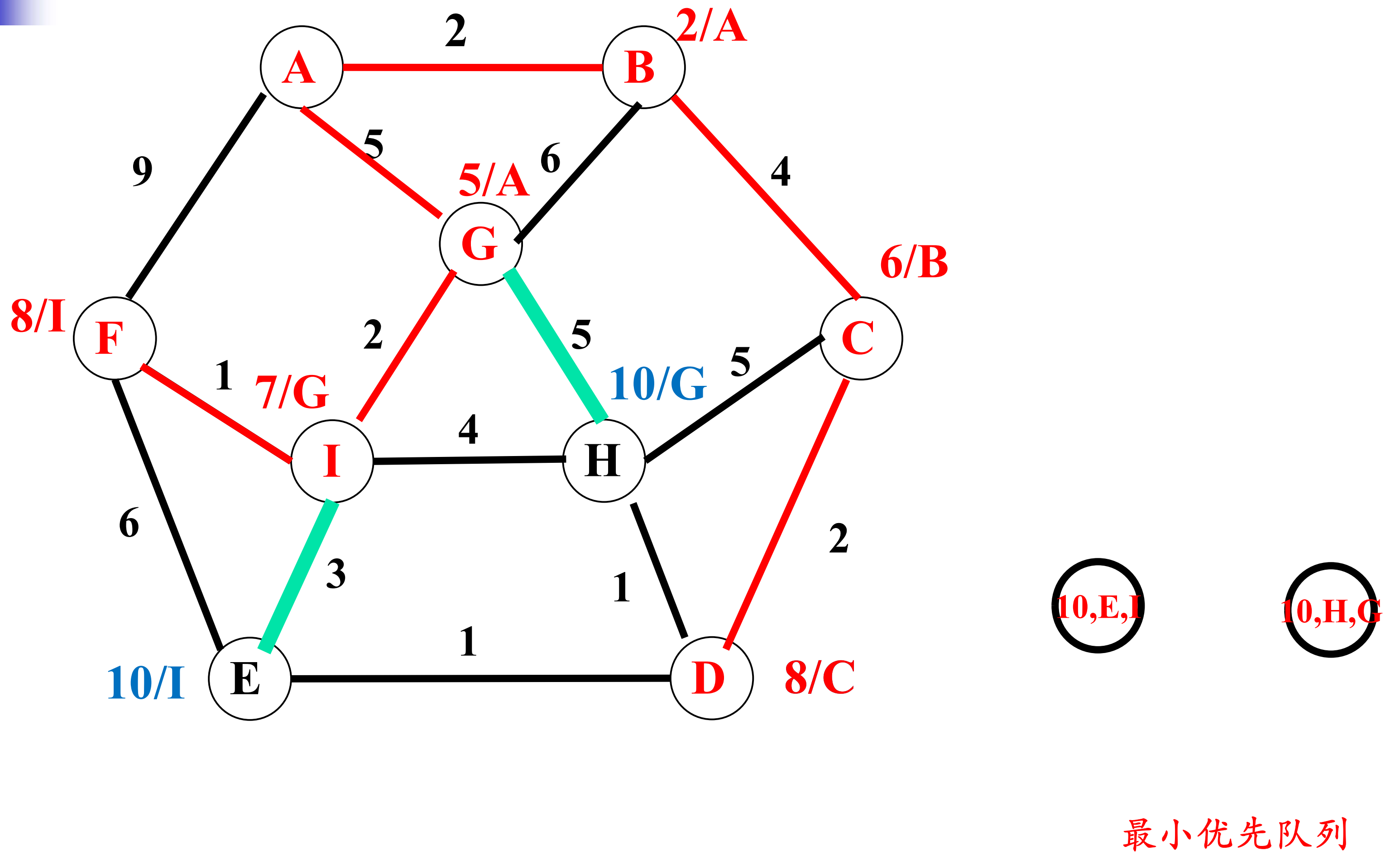
最小优先队列

1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

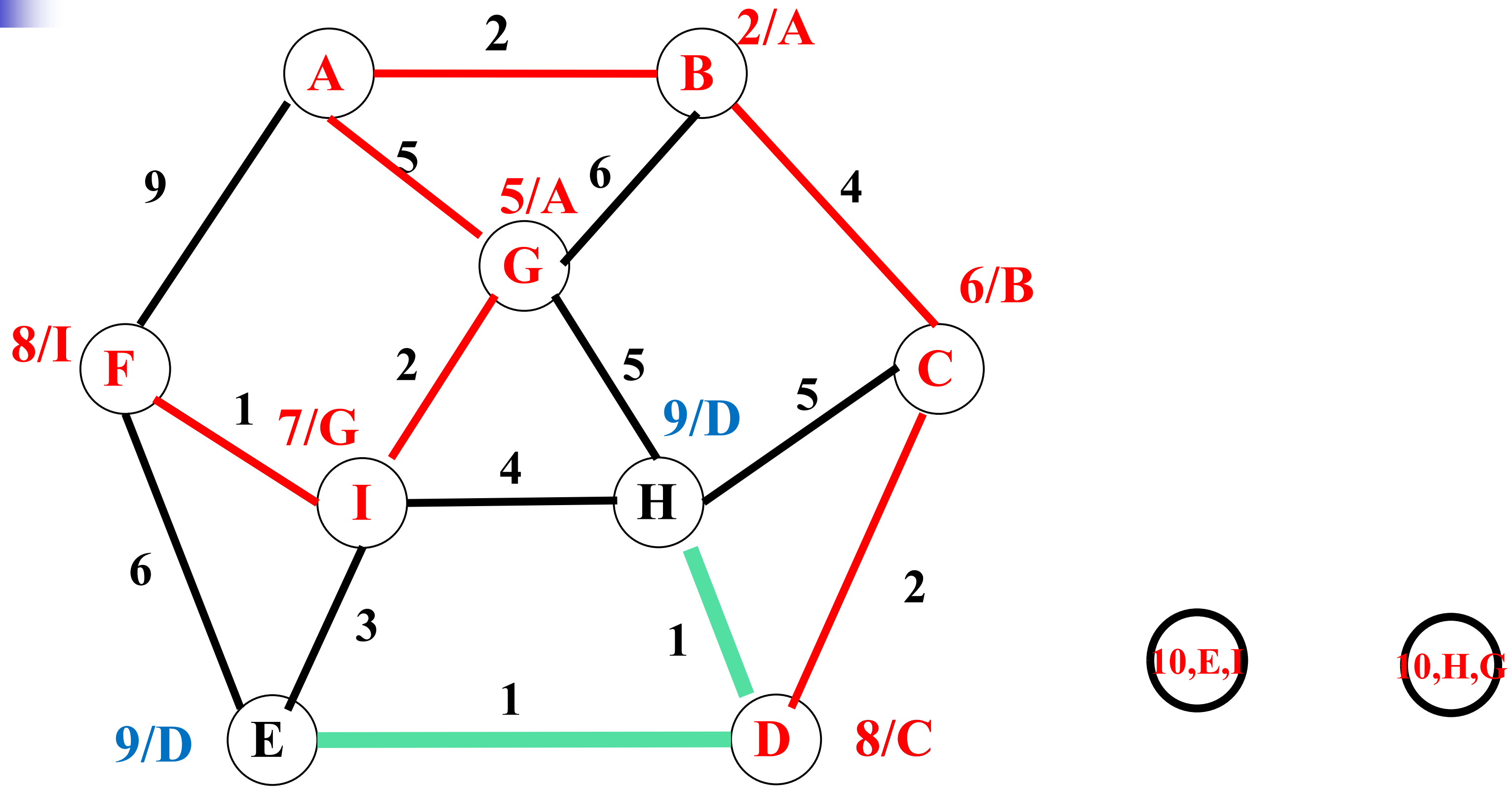




1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

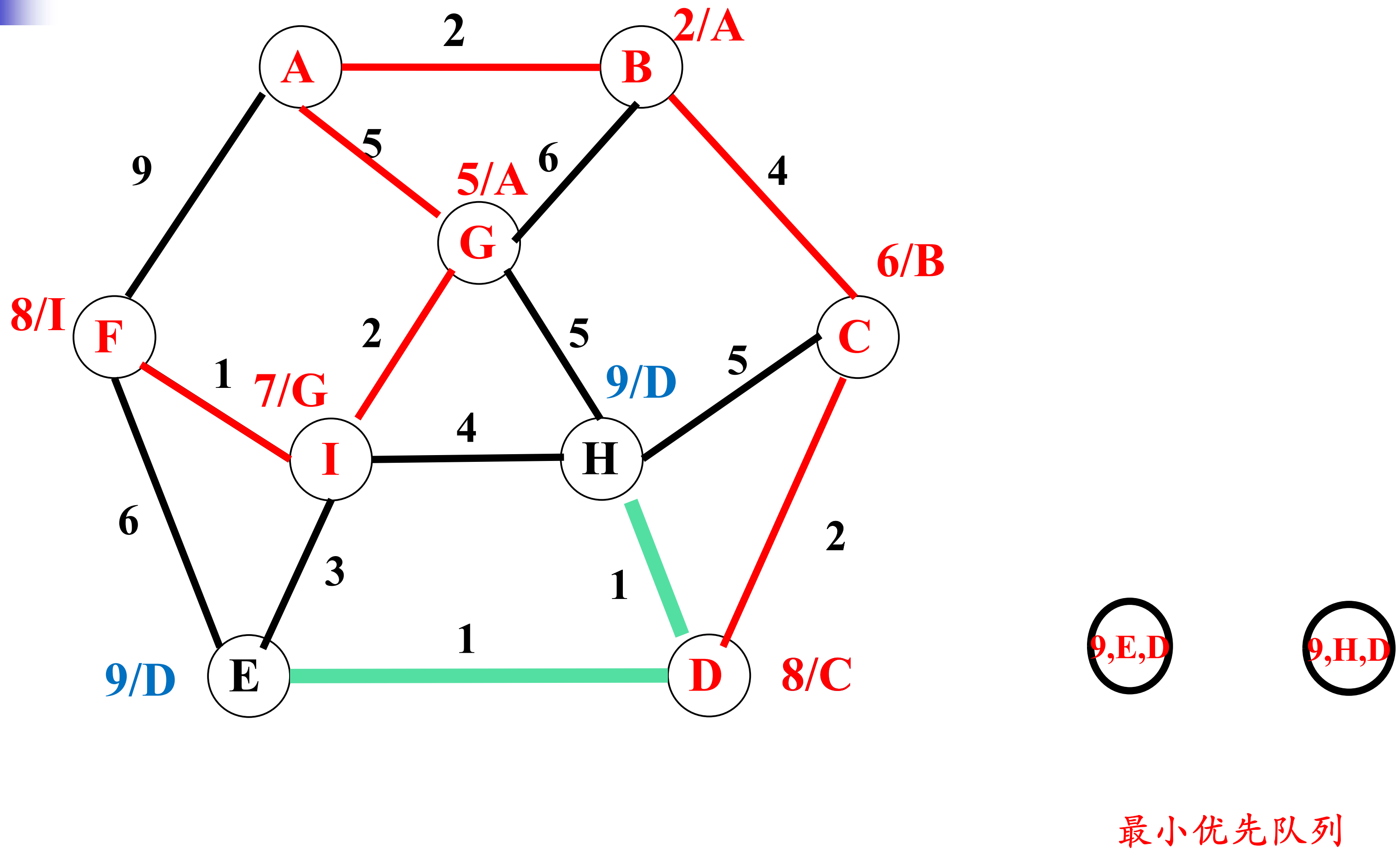


1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

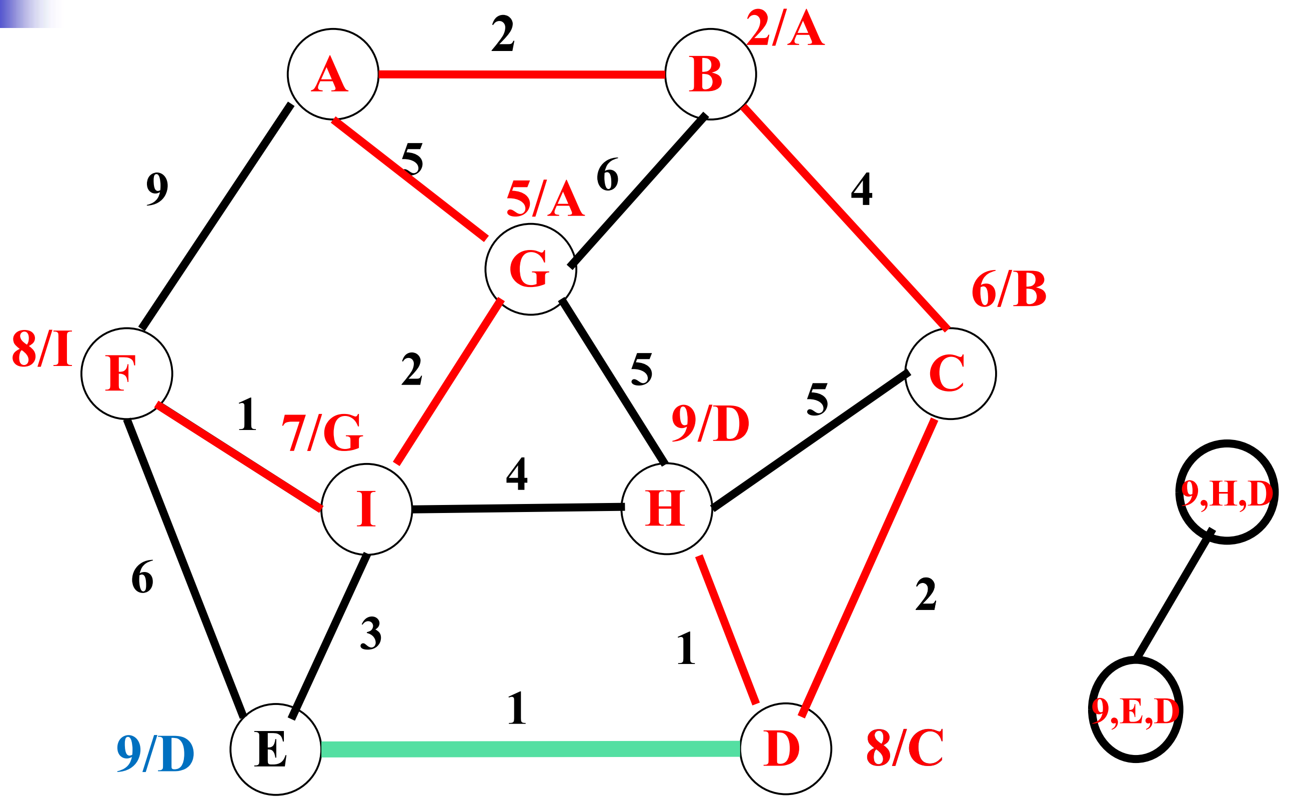


最小优先队列

1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.

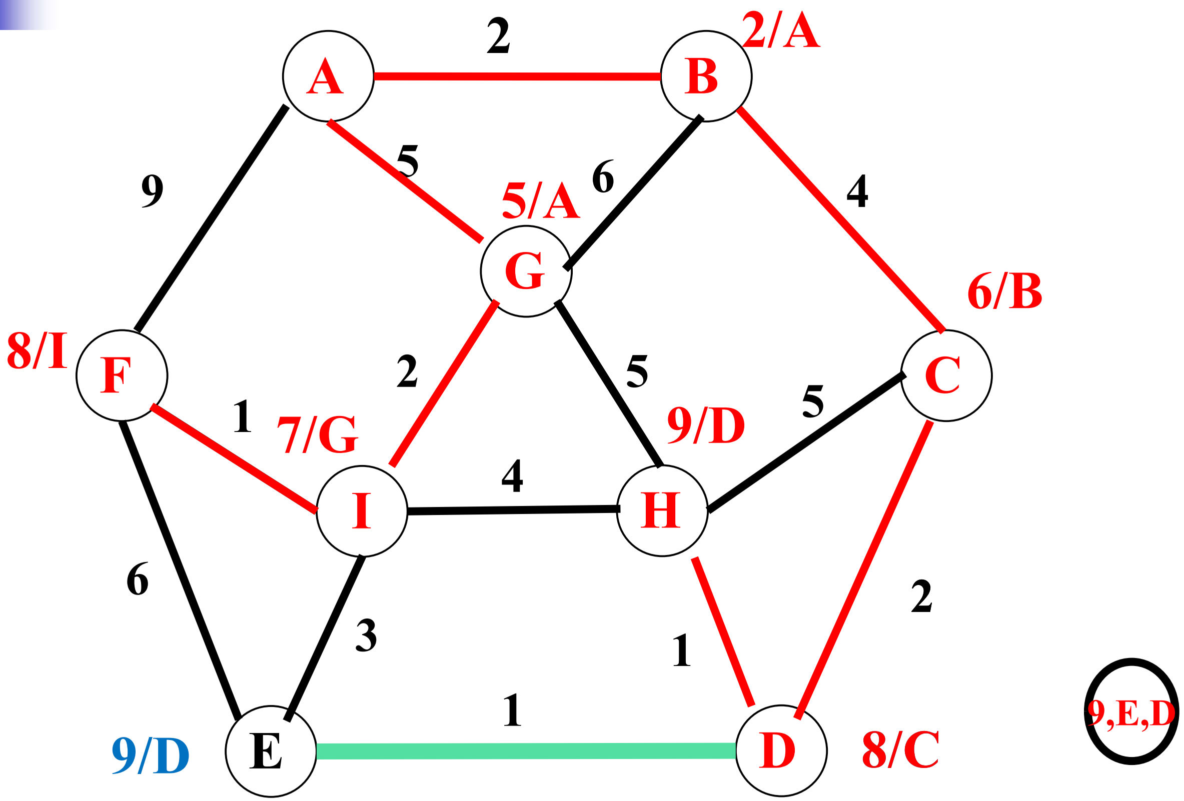


1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.



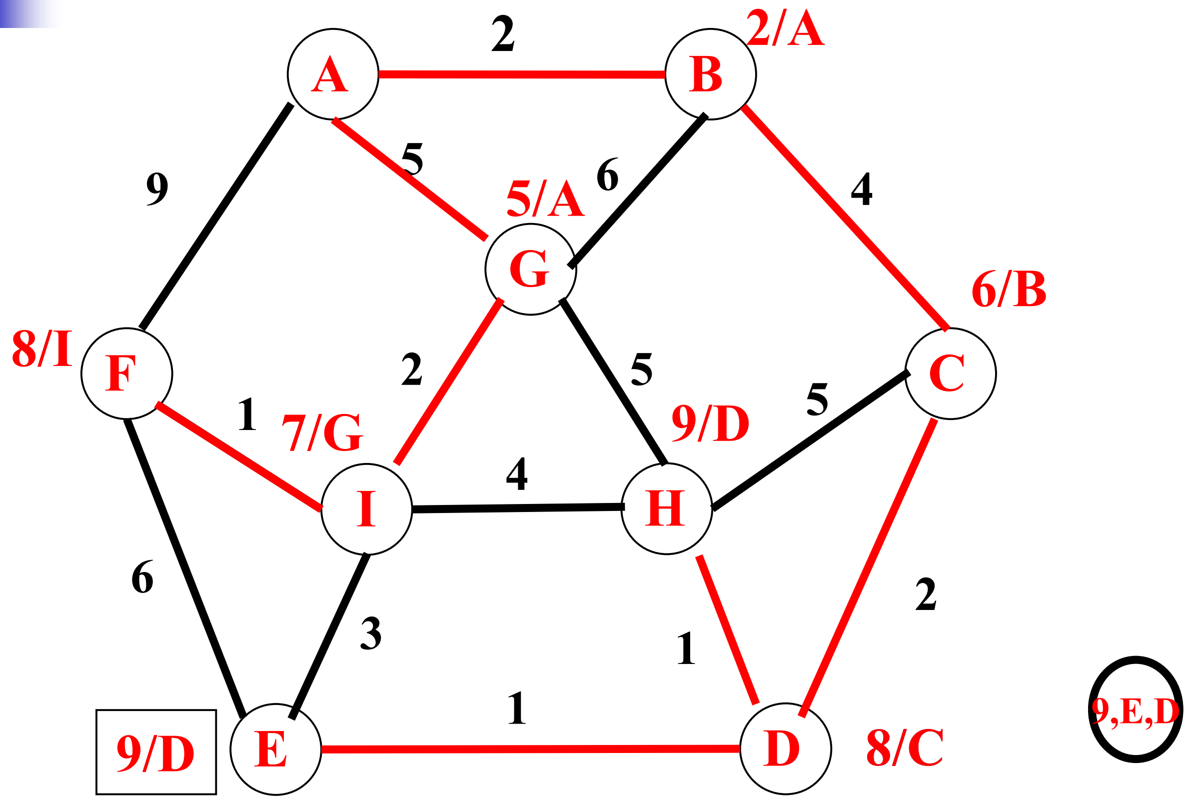
最小优先队列

1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.



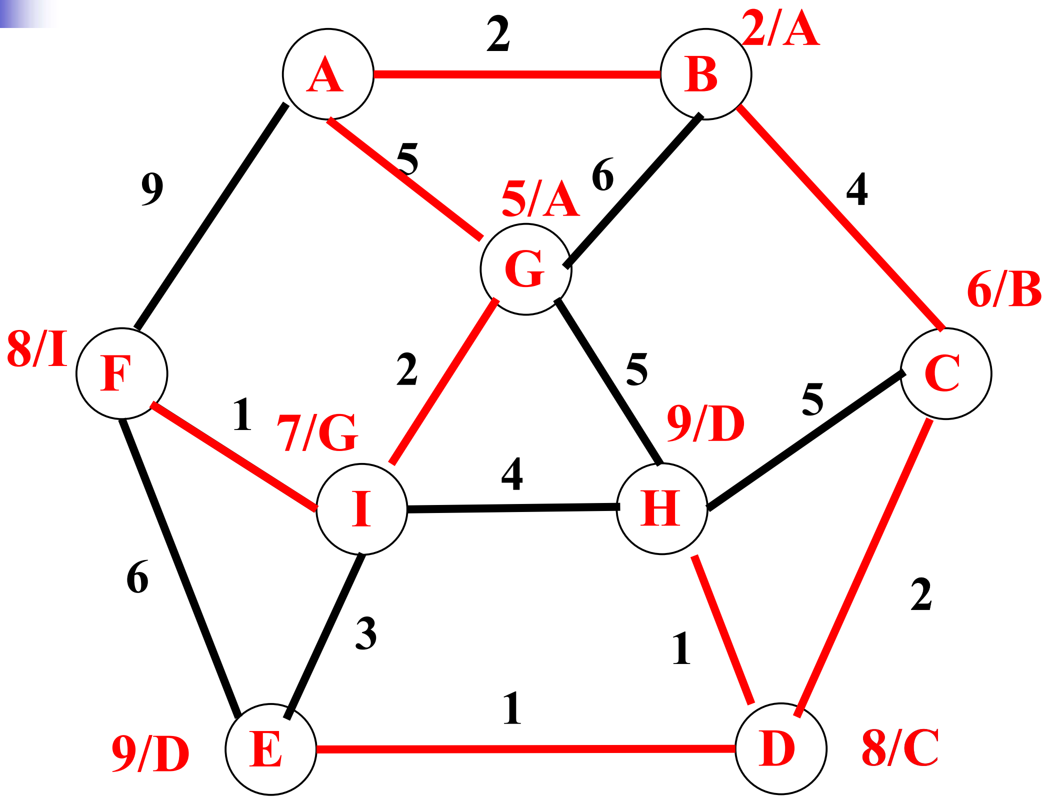
最小优先队列

1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.



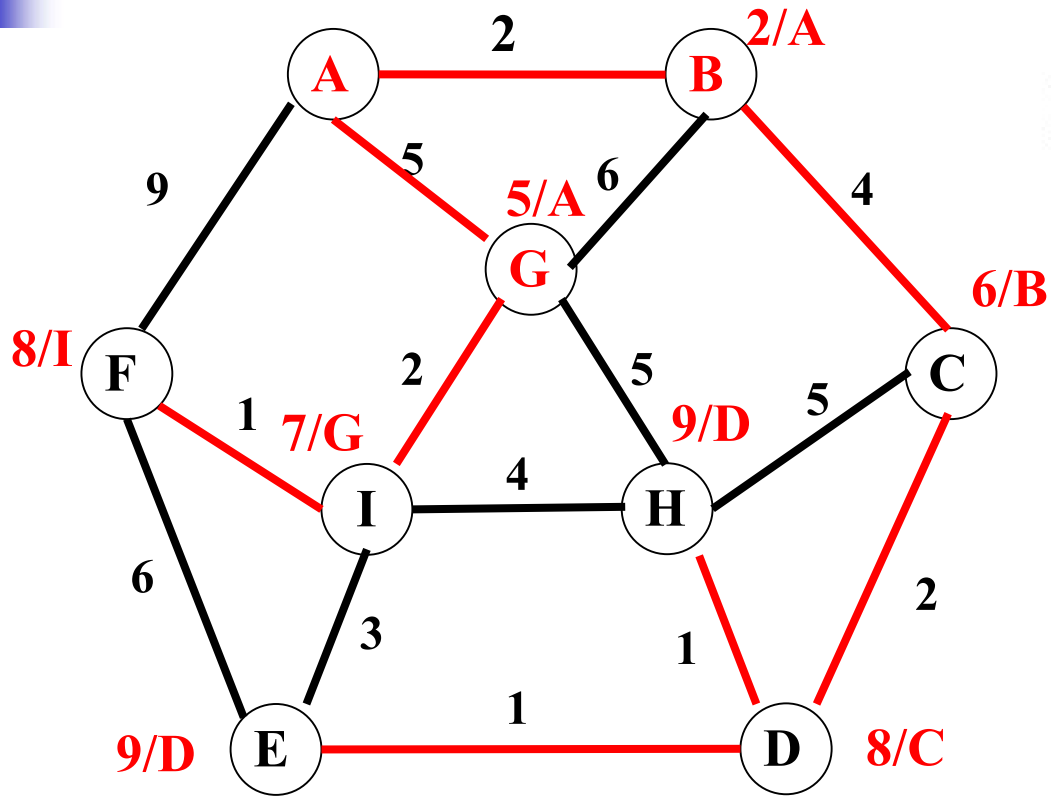
最小优先队列

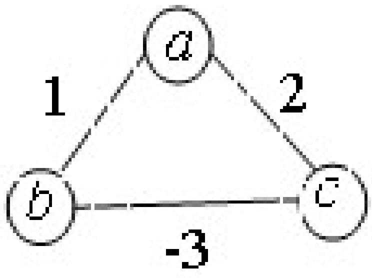
1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.



最小优先队列

1. *tree* vertices: in the tree constructed so far,
2. *fringe* vertices: not in the tree, but adjacent to some vertex in the tree,
3. *unseen* vertices: all others.





最短路径树

该算法只适用于静态网络

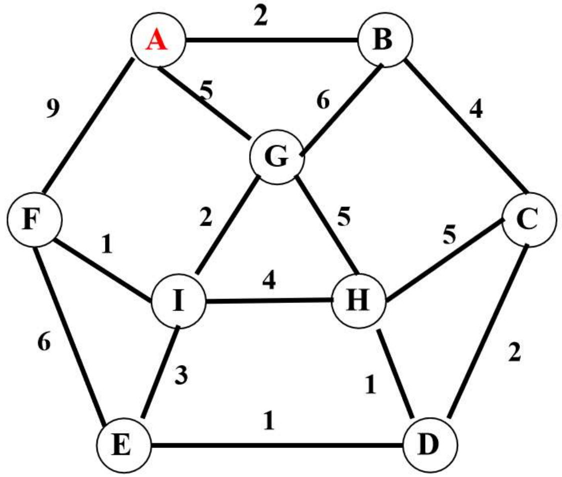
网络上边的权值不能为负数

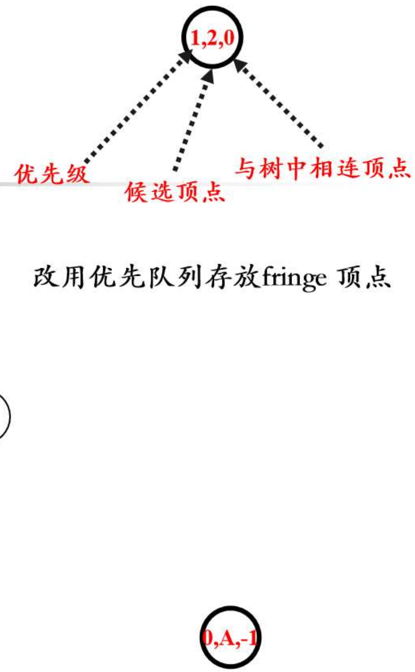
```
dijkstraSSSP(G, n) // OUTLINE
    Initialize all vertices as unseen.
    Start the tree with the specified source vertex s; reclassify it as tree;
    define d(s, s) = 0.
    Reclassify all vertices adjacent to s as fringe.
    While there are fringe vertices:
        Select an edge between a tree vertex t and a fringe vertex v such that
        (d(s, t) + W(tv)) is minimum;
        Reclassify v as tree; add edge tv to the tree;
        define d(s, v) = (d(s, t) + W(tv)).
        Reclassify all unseen vertices adjacent to v as fringe.
```

1初始化一个空的最小优先队列（候选节点），加入出发点s
2 从候选节点（最小优先队列）中选一个最小出队，将其加到最短路径树上，确定了其最短路径
3更新候选节点
4 重复2、3直到队列为空

```
void  shortestPaths(EdgeList[] adjInfo, int n, int s, int[] parent, float[] fringeWgt)
    int[]  status = new int[n+1];
    MinPQ  pq = create(n, status, parent, fringeWgt);

    insert(pq, s, -1, 0);
    while (isEmpty(pq) == false)
        int  v = getMin(pq);
        deleteMin(pq);
        updateFringe(pq, adjInfo[v], v);
    return;
```

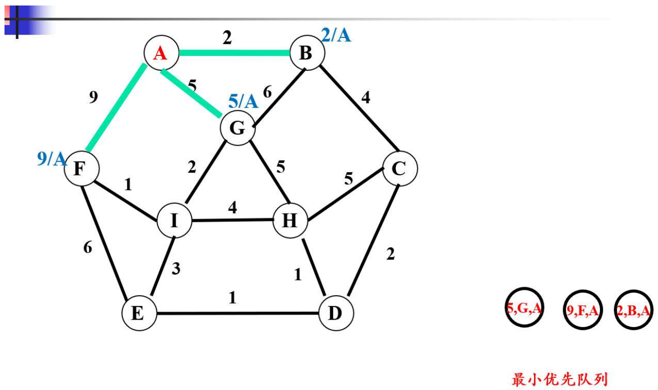


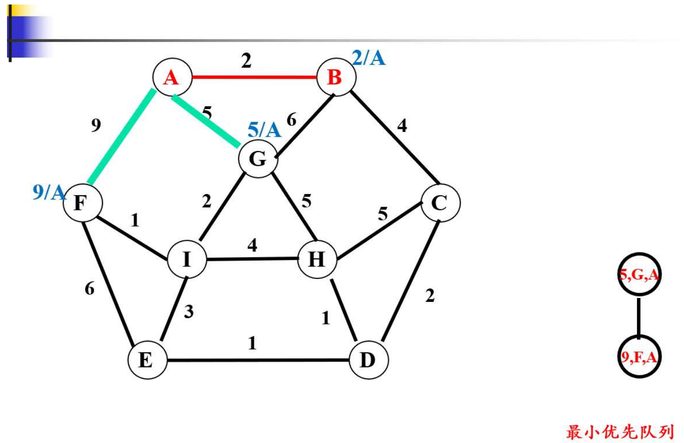


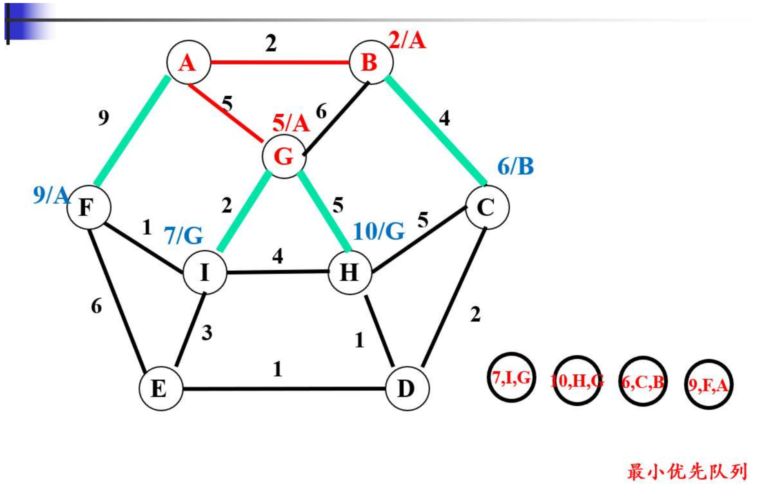
改用优先队列存放fringe 顶点

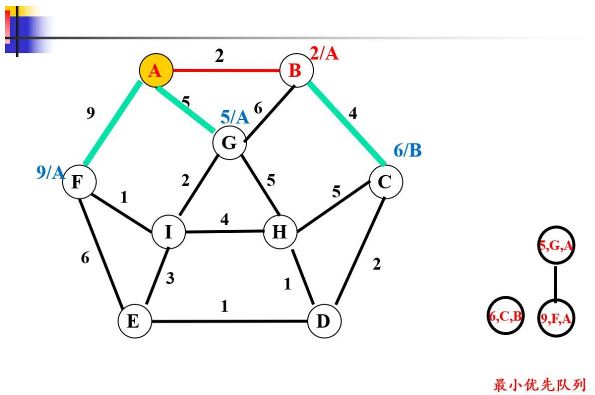
最小优先队列

```
void updateFringe(MinPQ pq, EdgeList adjInfoOfV, int v)
    float myDist = pq.fringeWgt[v];
    EdgeList remAdj;
    remAdj = adjInfoOfV;
    while (remAdj ≠ nil)
        EdgeInfo wInfo = first(remAdj);
        int w = wInfo.to;
        float newDist = myDist + wInfo.weight;
        if (pq.status[w] == unseen)
            insert(pq, w, v, newDist);
        else if (pq.status[w] == fringe)
            if (newDist < getPriority(pq, w))
                decreaseKey(pq, w, v, newDist);
        remAdj = rest(remAdj);
    return;
```



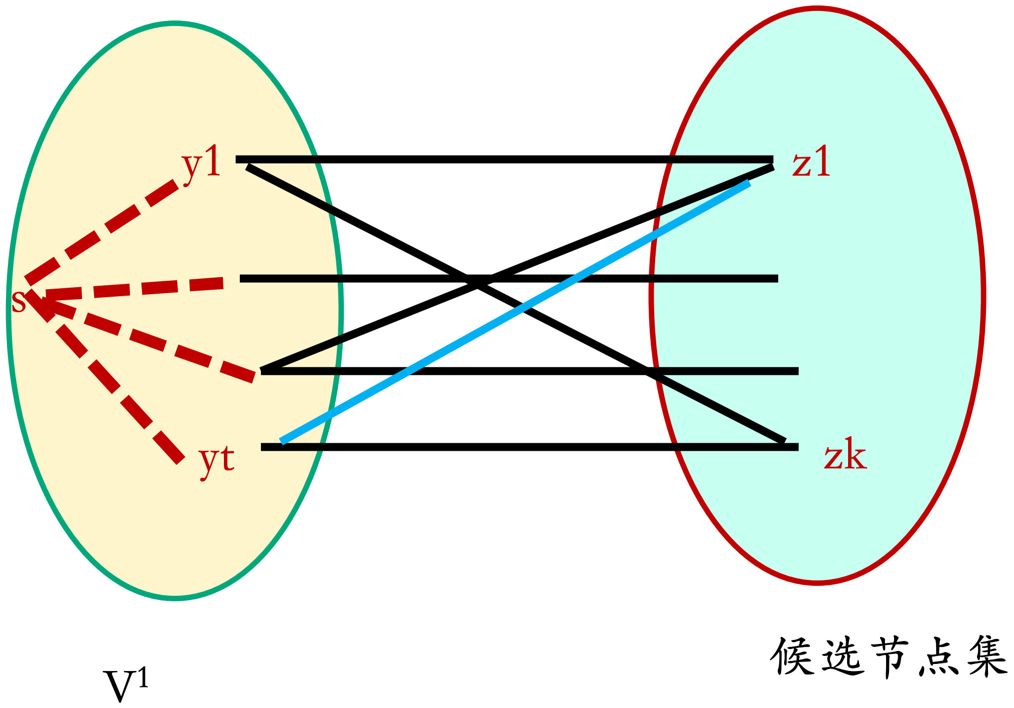






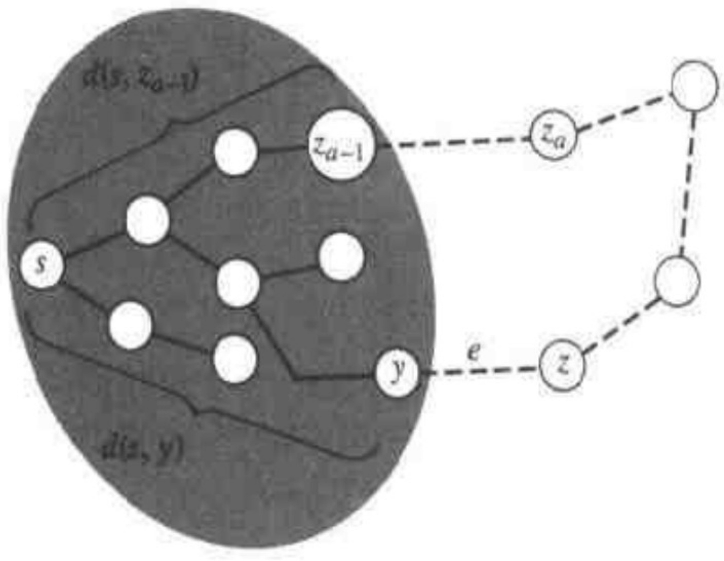
迪杰斯特拉算法

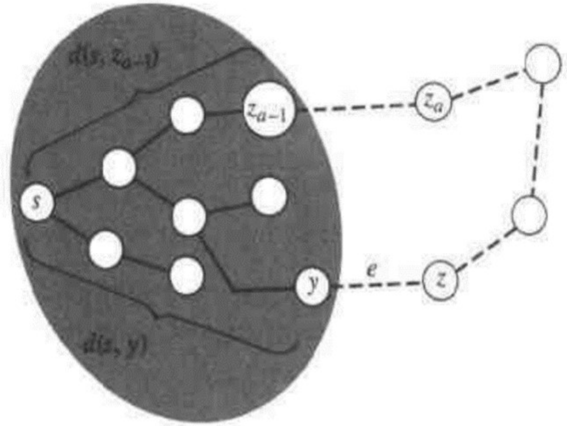
- 设在G=(V, E, W)为带权无向图，且权值非负。令$V^1 \subseteq V$, $s \in V^1$
- V^1是目前迪杰斯特拉算法已经加入最短路径树上的点的集合，s是出发点
- 令d(s,y)为G中从s到y的最短距离
- 迪杰斯特拉算法执行过程中每个候选顶点z的当前的最好情况：$\min\{d(s,y)+w(yz) \mid y \in V^1\}$，算法在最小优先队列中只保留每一候选节点的最好情况
- 迪杰斯特拉算法每一次迭代从所有候选顶点选一个，相当于选择：
- $d(s,y)+w(yz)=\min\{d(s,y^*)+w(y^*z^*) \mid y^* \in V^1, z^* \in V-V^1\}$。


y1
s
yt
z1
zk
V1
候选节点集

Theorem 8.6 Let $G = (V, E, W)$ be a weighted graph with nonnegative weights. Let V' be a subset of V and let s be a member of V'. Assume that $d(s, y)$ is the shortest distance in G from s to y, for each $y \in V'$. If edge yz is chosen to minimize $d(s, y) + W(yz)$ over all edges with one vertex y in V' and one vertex z in $V - V'$, then the path consisting of a shortest path from s to y followed by the edge yz is a shortest path from s to z.

- 定理8.6：设在G=(V, E, W)为带权无向图，且权值非负。令$V^1 \subseteq V$, $s \in V^1$。对每一$y \in V^1$，令d(s,y)为G中从s到y的最短距离。若边yz满足$d(s,y)+w(yz)=\min\{d(s,y^*)+w(y^*z^*) \mid y^* \in V^1,\ z^* \in V- V^1\}$。那么从s到y的最短路径+(y,z)为从s到z的最短路径。





- 令e=yz，且$s, x_1, x_2, \cdots, x_r, y$为从s到y的最短路径(可能y=s)。
- 令$P=s, x_1, x_2, \cdots, x_r, y, z$。$w(P)= d(s,y)+w(yz)$。假设P不是从s到z的最短路径，令$P1=s, z_1, z_2, \cdots, z_a, \cdots, z$为从s到z的一条最短路径。$z_a$是这条路径上第一个不在$V^1$中的顶点(可能$z_a=z$)。
- 则$w(P)>w(P1)$
- 根据定理条件中e=yz的确定，可知

  $w(P)= d(s,y)+w(yz)=d(s,y)+w(e)\le d(s, z_{a-1})+w(z_{a-1}z_a)$------------(1)
- 根据定理8.5，$P1=s, z_1, z_2, \cdots, z_a, \cdots, z$为从s到z的一条最短路径，则$s, z_1, z_2, \cdots, z_{a-1}$为从s到$z_{a-1}$的一条最短路径，其路径长度为$d(s, z_{a-1})$。
- 由于$s, z_1, z_2, \cdots, z_a$为路径P1的一部分，且该路径上的其余边权值非负，所以：$d(s, z_{a-1})+w(z_{a-1}z_a) \le w(P1)$-------------------------(2)
- 由(1),(2)可得$w(P) =d(s,y)+w(e)\le d(s, z_{a-1})+w(z_{a-1}z_a)\le w(P1)$,与$w(P)>w(P1)$矛盾，所以假设不成立，即P是从s到z的最短路径。

Theorem 8.7 Given a directed weighted graph G with nonnegative weights and a source vertex s, Dijkstra's algorithm computes the shortest distance (weight of a minimum-weight path) from s to each vertex of G that is reachable from s.

- 令迪杰斯特拉算法求出最短路径点的顺序为$s=v_0, v_1, v_2, \cdots, v_{n-1}$。下面证明算法循环了k次后，正确求出了从s到$v_0, v_1, v_2, \cdots, v_k$最短路径最短路径。
- 对k采用数学归纳法证明。
- 当k=0时，从s到s的最短路径$d(s,s)=0$;
- 当k>0时，假设定理对k-1成立。根据定理8.6，$V^1=\{v_0, v_1, v_2, \cdots, v_{k-1}\}$，$v_k=z$，若$d(s,y)+w(yv_k)=\min\{d(s,y^*)+w(y^*v^*_k) \mid y^*\in V^1, v^*_k \in V-V^1\}$，则$d(s,y)+w(yv_k)$是从s到$v_k$的最短路径长度→这也是算法求出的从s到$v_k$的最短路径长度。
- 若循环找不出候选边，算法结束，余下的距离均是∞。

Ch9.4 弗洛伊德算法

- 求每一对顶点之间的最短路径
- 从 v_i 到 v_j 的所有可能存在的路径中，选出一条长度最短的路径。

Ch9.4 弗洛伊德算法

- **若**$<v_i,v_j>$**存在，则存在路径**(v_i,v_j)
- **若**$<v_i,v_1>,<v_1, v_j>$**存在，则存在路径**(v_i,v_1, v_j)
- **若**$(v_i,\ldots,v_2), (v_2,\ldots, v_j)$**存在，则存在一条路径**$(v_i, \ldots, v_2, \ldots v_j)$
- ...
- **依次类推，则** v_i **至** v_j**的最短路径应是上述这些路径中，路径长度最小者。**

Ch9.4 弗洛伊德算法--顶点v_i到顶点v_j的最短路径

- 如果$<v_i,v_j>\in E(G)$，则从v_i到v_j存在一条路径（v_i,v_j）；
- 该路径是否为最短路径尚需进行n次试探。
- 首先考虑路径（v_i,v_1,v_j），若其存在，比较路径（v_i,v_1,v_j）和路径（v_i,v_j）的长度，取其中较小者为从v_i到v_j的中间顶点序号不大于1的最短路径；
- 在路径上再加一个顶点v_2，若（v_i，…，v_2）和（v_2，…，v_j）分别是当前找到的中间顶点的序号不大于2最短路径，那么将（v_i，…，v_2，…，v_j）和已找到的中间结点的序号不大于1的最短路径比较，取其中较小的为从v_i到v_j的中间顶点序号不大于2的最短路径；
- 再增加一个顶点v_3，继续进行试探，依次类推。经过n次试探后可求得从顶点v_i到顶点v_j的最短路径。

$D^{(k)}[i][j]$表示从v_i到v_j的中间顶点序号不大于k的最短路径的长度；$D^{(n)}[i][j]$表示从v_i到v_j的最短路径的长度

Ch9.4 弗洛伊德算法

- 弗洛伊德算法递推地产生一个n阶矩阵序列：

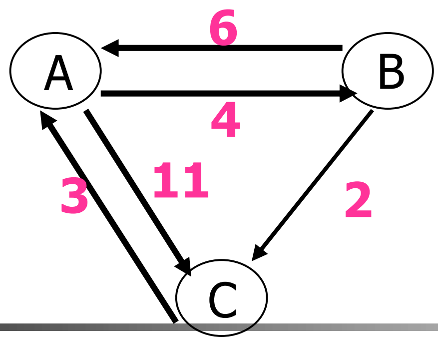
  $D^{(0)}, D^{(1)}, \ldots, D^{(k)}, \ldots, D^{(n)}$

  $D^{(0)}[i][j]=w_{ij}$;

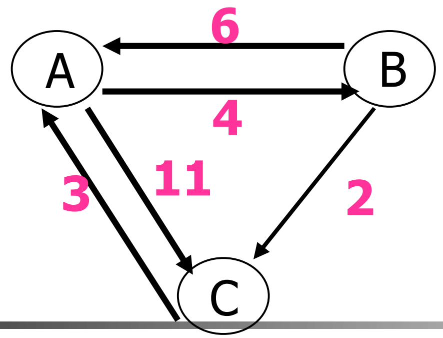
  $D^{(k)}[i][j]=\min\{D^{(k-1)}[i][j], D^{(k-1)}[i][k]+D^{(k-1)}[k][j]\}$ $(1\le k\le n)$

$$w_{ij}=\begin{cases} w(v_i v_j) & i\neq j,\ v_i v_j\in E \\ \infty & i\neq j,\ v_i v_j\notin E \\ 0 & i=j \end{cases}$$

Lemma 9.3 For each k in $0,\ldots,n$, let $d_{ij}^{(k)}$ be the weight of a shortest simple path from v_i to v_j with highest-numbered intermediate vertex v_k, and let $D^{(k)}[i][j]$ be defined by Equation (9.3). Then, $D^{(k)}[i][j]\le d_{ij}^{(k)}$. □



$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$



$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

$$D^{(0)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix} \quad D^{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \quad D^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix} \quad D^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

弗洛伊德算法

- Void allPairsShortestPaths(float[ ][ ] w,int n, float[ ][ ] D)

```
int i,j,k;
D=w;
for(k=1;k≤n;k++)
    for(i=1;i ≤n;i++)
        for(j=1;j≤n;j++) D[i][j]=min(D[i][j],D[i][k]+D[k][j])
```

当边上的权值存在负数时，FLOY算法是否成立？

当边上的权值存在负数时，FLOY算法是否成立？

$$
\begin{bmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & \infty & 3 \\ 5 & \infty & 0 & -3 \\ \infty & -1 & 4 & 0 \end{bmatrix}
$$

$$
\begin{bmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & 7 & 3 \\ 5 & 7 & 0 & -3 \\ \infty & -1 & 4 & 0 \end{bmatrix}
\begin{bmatrix} 0 & 2 & 4 & 3 \\ 3 & 0 & 7 & 3 \\ 5 & 7 & 0 & -3 \\ 2 & -1 & 4 & 0 \end{bmatrix}
\begin{bmatrix} 0 & 2 & 4 & 1 \\ 3 & 0 & 7 & 3 \\ 5 & 7 & 0 & -3 \\ 2 & -1 & 4 & 0 \end{bmatrix}
\begin{bmatrix} 0 & 0 & 4 & 1 \\ 3 & 0 & 7 & 3 \\ -1 & -4 & 0 & -3 \\ 2 & -1 & 4 & 0 \end{bmatrix}
$$

As long as there are no negative-weight cycles, there is always a *simple* shortest path between any pair of nodes. Algorithm 9.4 is guaranteed to find the shortest simple path, regardless of whether weights are negative or not.