图论1　最短路/最小生成树

目录

- 图的基础知识
- 最短路算法
- 最小生成树算法
- 例题选讲

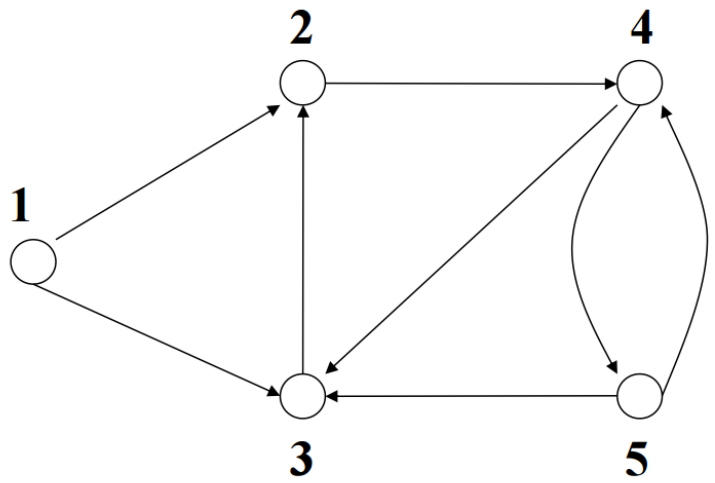
目录

- **图的基础知识**
  - 图的储存
  - 部分概念
  - 图的遍历
- 最短路算法
- 最小生成树算法
- 例题选讲

图的储存

- **邻接矩阵**
- g[i][j]的值表示点i到j的连边情况

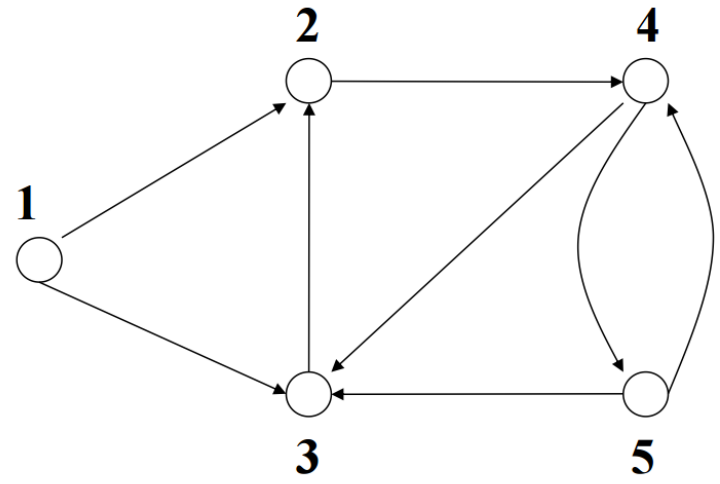
$$g[i][j]=\begin{cases}1 & \text{点i与点j之间存在一条边}\\ 0 & \text{点i与点j之间不存在一条边或者i=j}\end{cases}$$



$$\begin{pmatrix}0 & 1 & 1 & 0 & 0\\ 0 & 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 1 & 0\end{pmatrix}$$

图的储存

- **邻接表**
- g[i][j]表示i点出发的第j条边为g[i][j]



$$\begin{pmatrix} 2 \\ 1 \\ 1 \\ 2 \\ 2 \end{pmatrix} \begin{pmatrix} 2 & 3 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 5 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 \end{pmatrix}$$

图的储存

- 代码实现

```
int g[V][V];
// g[i][j]=0 表示图内没有i->j的边.
// 否则图内有一条i->j的边,且边权为g[i][j],无权则一般设为1.

void init()
{
    memset(g,0,sizeof(g));
}

g[i][j]=w; //增加一条i->j,边权为w的边.
g[i][j]=0; //删除i->j的边.

for(int i=0;i<V;++i)    //遍历与节点u相邻的点.
    if(g[u][v])
    {
        //图中有一条u->v的边.
        /*do something... */
    }
```

```
vector<int>g[V];

void init()     //初始化:把边全部清空.
{
    for(int i=0;i<V;++i)
        g[i].clear();
}

void add_edge(int u,int v)  //新增一条u->v的边.
{
    g[u].push_back(v);
}

for(int v:g[u])     //遍历与节点u相邻的点.
{
    //图中有一条u->v的边.
    /*do something... */
}
```

图的储存

- **链式前向星**
- 这种存图方式的数据结构主要是**边集数组**，顾名思义，图的边是用数组来存储的。

edge_num表示边的数量

head[i]表示点i出发(按输入顺序)最后一条边的编号

edge[i].next表示以edge[i]同起点的下一条边的编号

edge[i].to表示边edge[i]的终点

head[i]和e[i].next等于-1则表示没有边

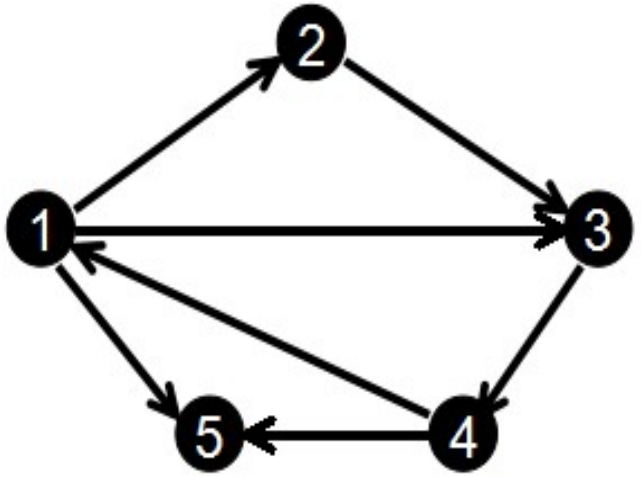
```
struct EDGE
{
    int to;
    int next;
}edge[E];

int head[V];
int edge_num;
```

图的储存

- 一个例子

```
edge[0].to = 2;    edge[0].next = -1;    head[1] = 0;
edge[1].to = 3;    edge[1].next = -1;    head[2] = 1;
edge[2].to = 4;    edge[2],next = -1;    head[3] = 2;
edge[3].to = 3;    edge[3].next = 0;     head[1] = 3;
edge[4].to = 1;    edge[4].next = -1;    head[4] = 4;
edge[5].to = 5;    edge[5].next = 3;     head[1] = 5;
edge[6].to = 5;    edge[6].next = 4;     head[4] = 6;
```



输入边的顺序：

1 2
2 3
3 4
1 3
4 1
1 5
4 5

```
head[1] = 5;
head[2] = 1;
head[3] = 2;
head[4] = 6;
```

图的储存

- 代码

```
void init()  //初始化，把边清空，边数赋值0，head初始化成-1.
{
    edge_num=0;
    memset(head,-1,sizeof(head));
}

void add_edge(int u,int v)
{
    edge[edge_num].to=v;
    edge[edge_num].next=head[u];
    //贪图方便可以写成edge[edge_num]={v,head[u]};
    head[u]=edge_num;
    edge_num++;
}

for(int i=head[u];~i;i=edge[i].next) //遍历顶点u所有的边
{
    int v=edge[i].to;  //此时图内有一条边 u->v 存在。
    /*do something*/
}
```

图的储存

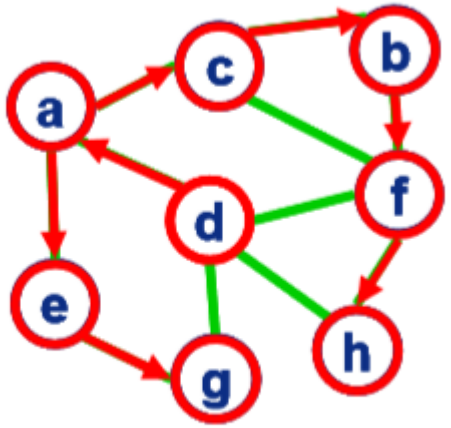
- **总结**
- 对于邻接矩阵存图来说，由于内存消耗的局限性，它的适用范围比较狭窄，几乎只能在简单图论题目中见到。
- 邻接表存图是最为常见的一种，绝大部分采用C++STL中的vector实现，一般情况下大部分图论题目都能使用该存图方式（注意重边）。
- 链式前向星其实是一种较好替代邻接表来存图的数据结构，在邻接表存图不能使用时可以使用，几乎可以用于全部图论题目。

部分概念

- **顶点的度**：
  - 在无向图中，某个顶点的度是与它相关联的边的数目。
  - 在有向图中，一个顶点的出度是以它为起始的边的数目，入度是以它为终止的边的数目。
- **简单路径**：顶点不重复的路径。
- **自环**：从某个顶点出发连向它自身的边。
- **环**：从某个顶点出发再回到自身的路径，又称回路。
- **重边**：从一个顶点到另一个顶点有两条边直接相连。

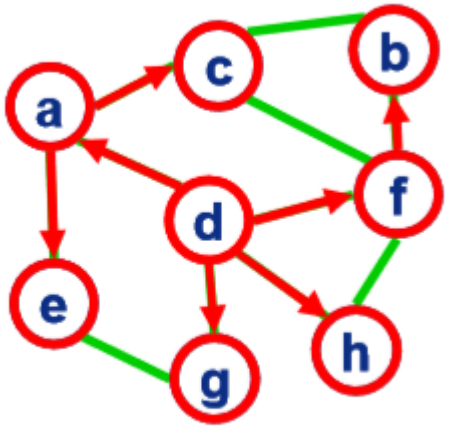
图的遍历

Depth First Search (d)

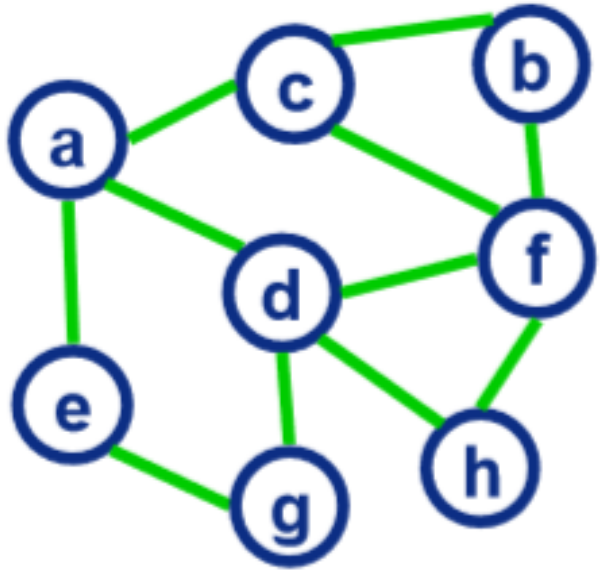


d a c b f h e g

Breadth First Search (d)



d a f g h c e b



图的遍历

```
bool vis[V];
void dfs(int u)
{
    vis[u]=1;                   //把此前没访问过，但现在访问到的点做一个标记。
    /* do something... */

    //遍历与节点u相邻的所有点：
    //1.邻接矩阵：
    for(int i=0;i<V;++i)
        if(g[u][i]==1&&vis[i]==0)
            dfs(i);

    //2.vector邻接表：
    for(int v:g[u])
        if(vis[v]==0)
            dfs(v);

    //3.链式前向星：
    for(int i=head[u];~i;i=edge[i].next)
        if(vis[edge[i].to]==0)
            dfs(edge[i].to);

    /* do something... */
    return;
}
```

```
bool vis[V];
void bfs(int root)              //从root开始搜索.
{
    queue<int>q;                //使用先进先出的队列来存储遍历顺序.

    memset(vis,0,sizeof(vis));
    vis[root]=1;
    q.push(root);
    while(!q.empty())           //表示还有点需要访问.
    {
        int u=q.front(); q.pop();       //取出队列头.

        /* do something... */
        //遍历与节点u相邻的所有点：然后扔到待访问队列里面
        //1.邻接矩阵：
        for(int i=0;i<V;++i)
            if(g[u][i]==1&&vis[i]==0)
            {
                vis[i]=1;
                q.push(i);
            }
        //2.vector邻接表：
        for(int v:g[u])
            if(vis[v]==0)
            {
                vis[v]=1;
                q.push(v);
            }

        //3.链式前向星：
        for(int i=head[u];~i;i=edge[i].next)
            if(vis[edge[i].to]==0)
            {
                vis[edge[i].to]=1;
                q.push(edge[i].to);
            }
        /* do something... */
    }
    return;
}
```

目录

- 图的基础知识
- **最短路算法**
  - Dijkstra算法
  - Bellman-Ford算法
  - SPFA算法
  - Floyed算法
- 最小生成树算法
- 例题选讲

单源最短路

- 从一个点出发，到达其他顶点的最短路径的长度。
- 基本操作：**松弛**
- d[u]+map[u, v]< d[v]这样的边(u,v)称为紧的(tense),可以对它进行松弛(relax):
- d[v] = d[u]+w, pred[v] = u
- 最开始给每一个点一个很大的d值从d[s]=0开始，不断的对可以松弛的点进行松弛，不能松弛的时候就已经求出了最短路了。

Dijkstra算法

- Dijkstra(迪杰斯特拉)算法是典型的单源最短路径算法，用于计算一个节点到其他所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止
- 注意该算法要求图中不存在负权边。
- 可以证明，**具有最小的d[i]（临时最短路）值的（还没加入最短路）点在此以后无法松弛**
- 所以每次找最近的点进行松弛操作

Dijkstra算法

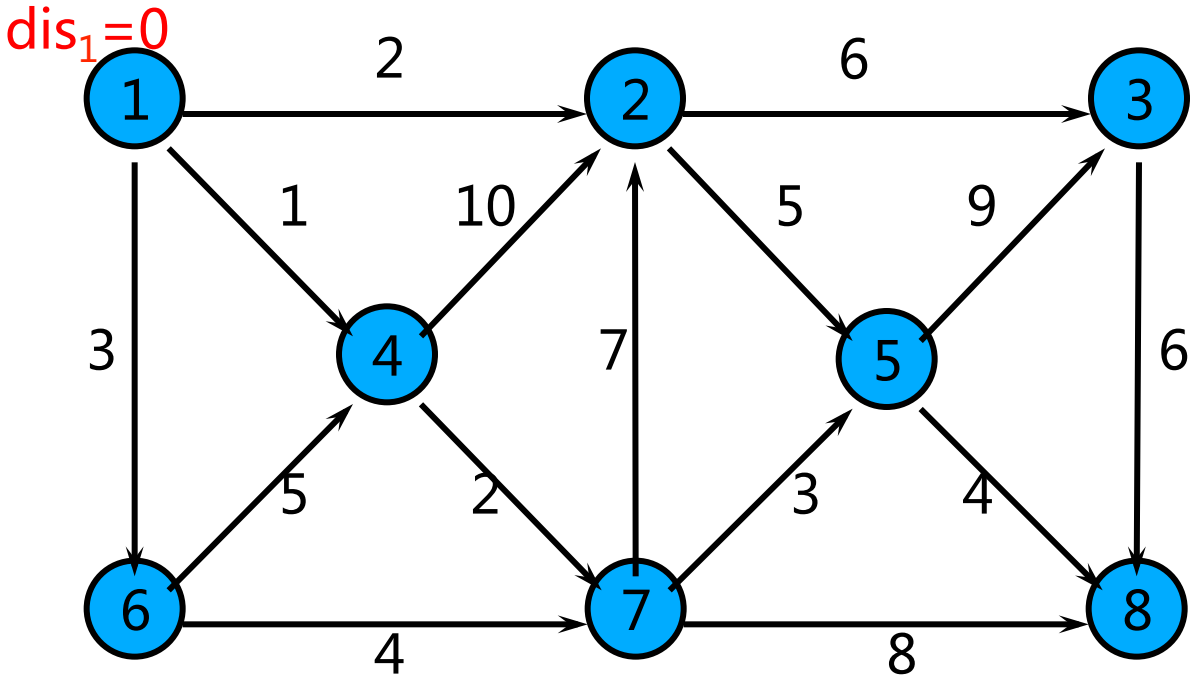
- 在开始之前，认为所有的点都没有进行过计算，dis[]全部赋值为∞(dis[]表示各点当前到源点的最短距离），源点s的dis[s]=0。把点集V分成两部分S与T，S中的点为已经确定最短路的点。S初始化为空集，T为全集V。
- 每次在T中**选择dis值最小的点u**，它的dis值就是最短路径值，将它**加入S中**，然后遍历与u相邻的点v，对所有v做一次松弛操作。
- 重复n次上一操作，算法完成。

- 正确性如何证明？

Dijkstra算法

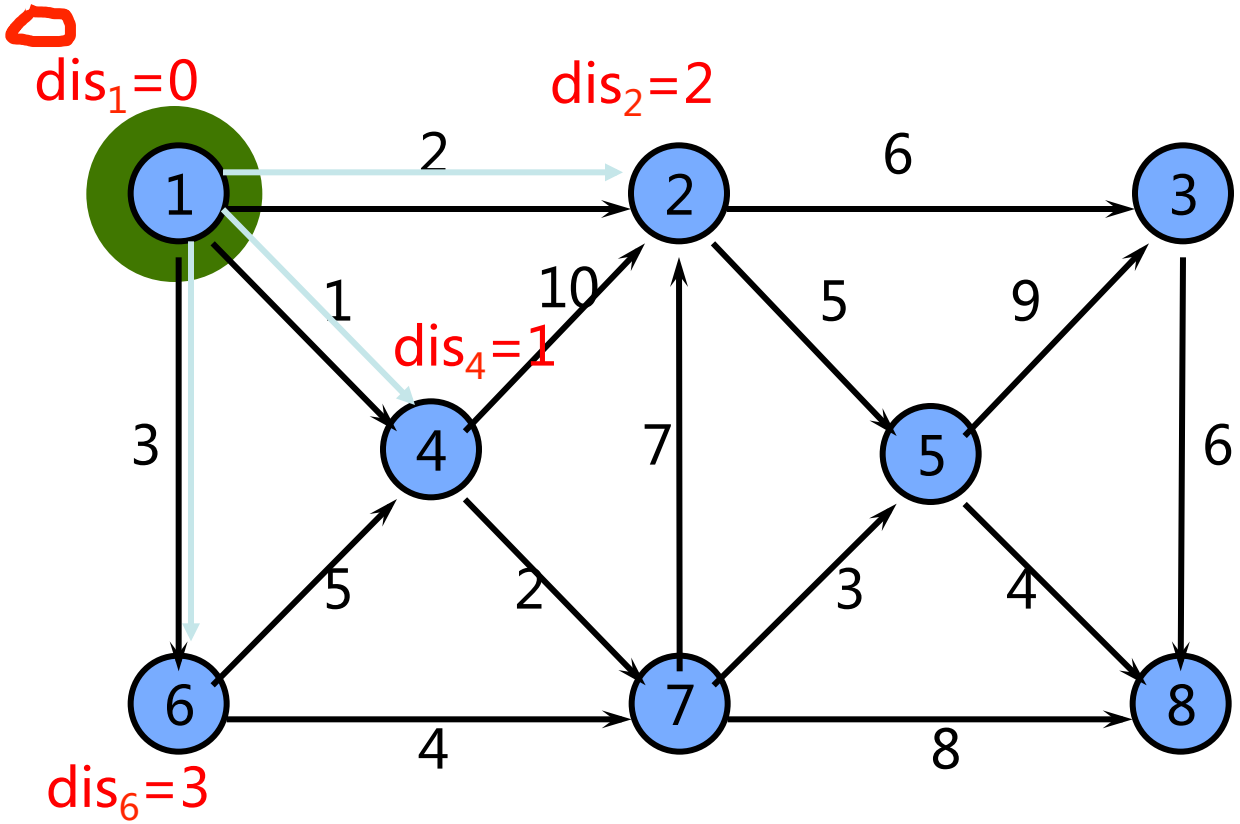
演示：求从1到8的最短路径

S={},dis[1]=0;



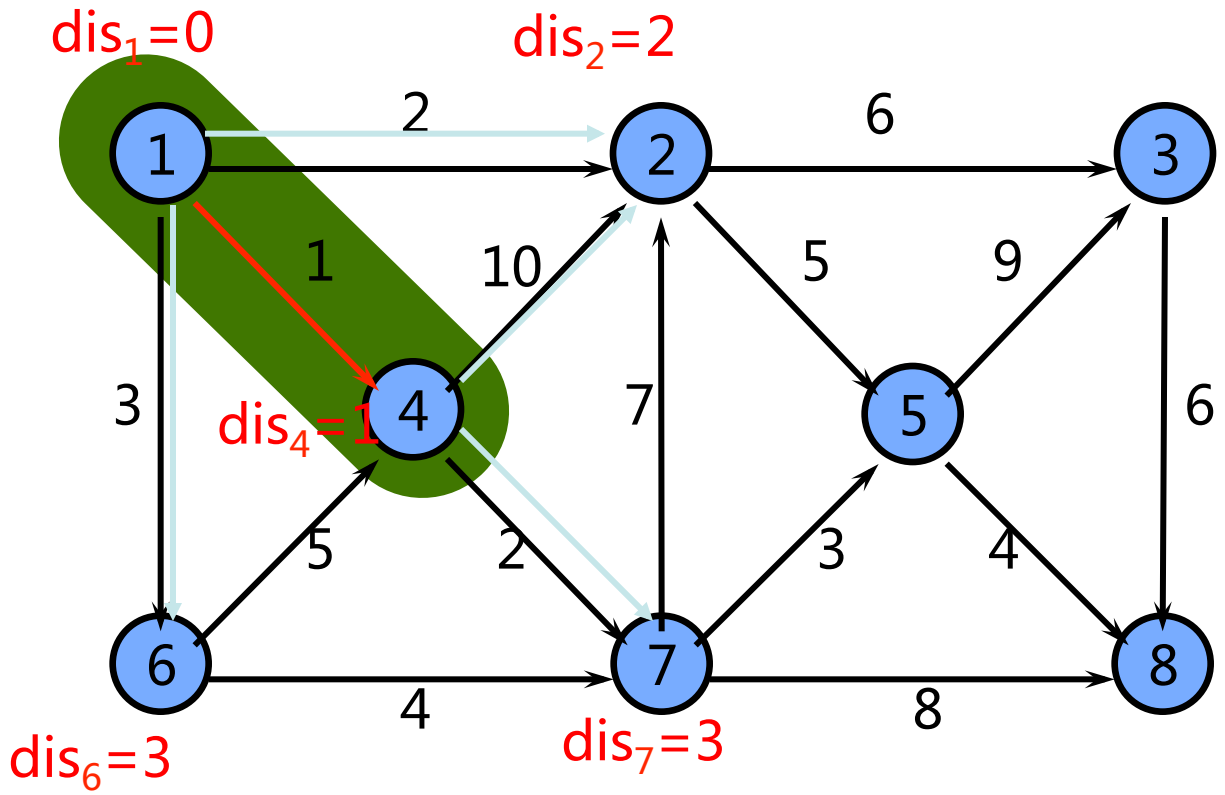
Dijkstra算法

S={1}; dis[2]=2,dis[4]=1,dis[6]=3;



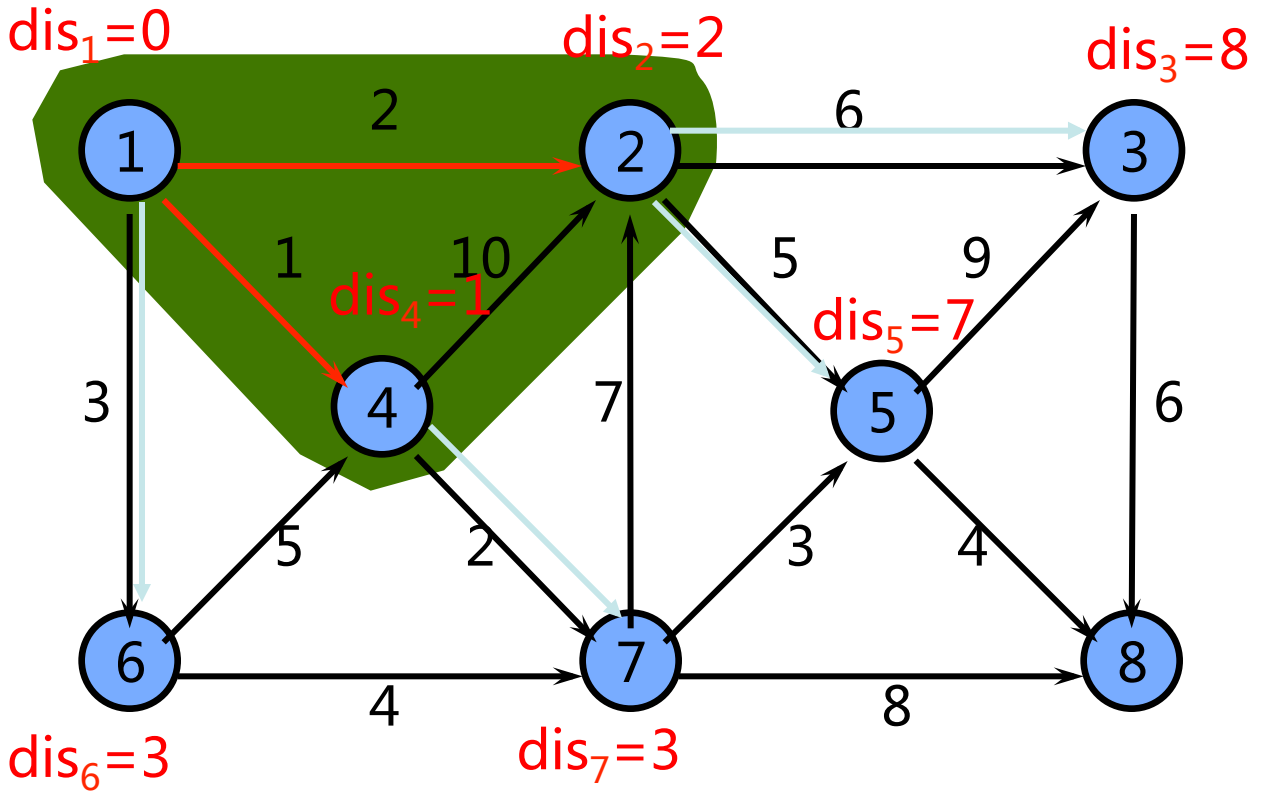
Dijkstra算法

S={1,4},dis[7]=3;



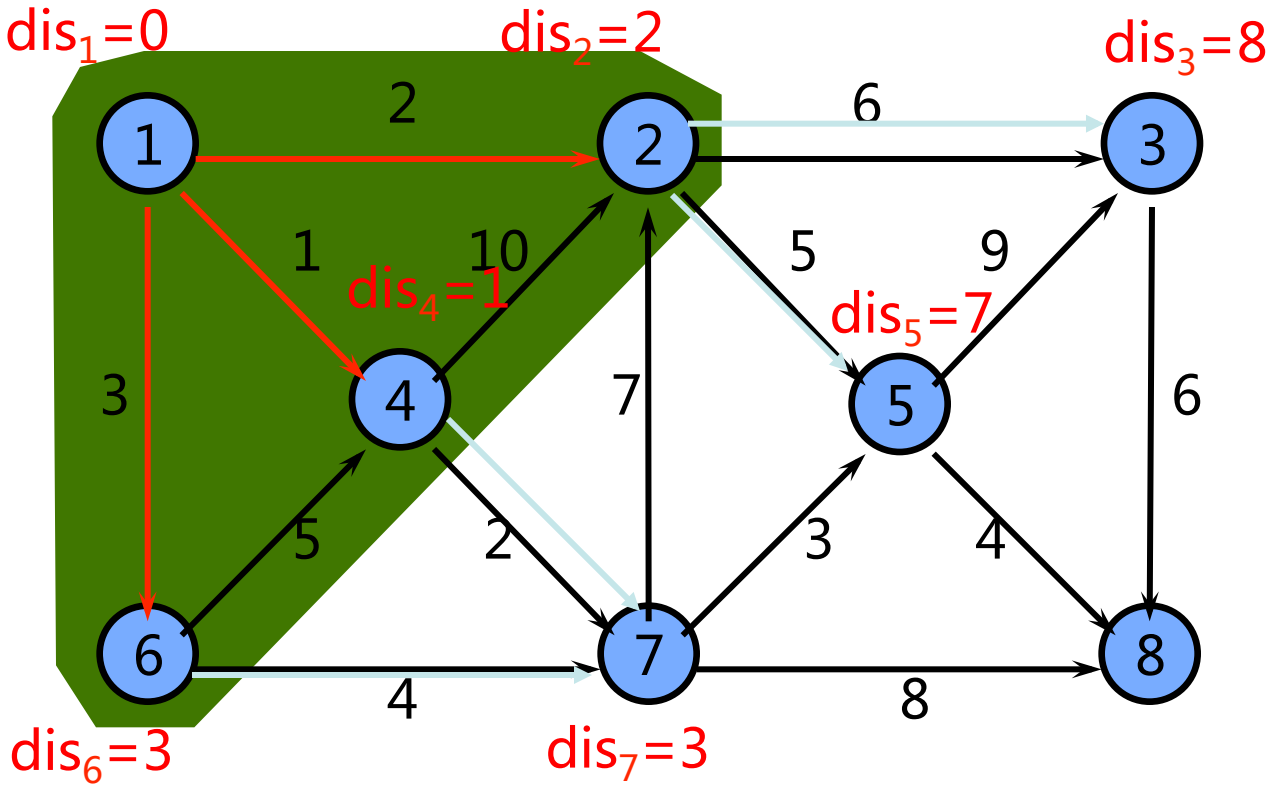
Dijkstra算法

S={1,2,4}



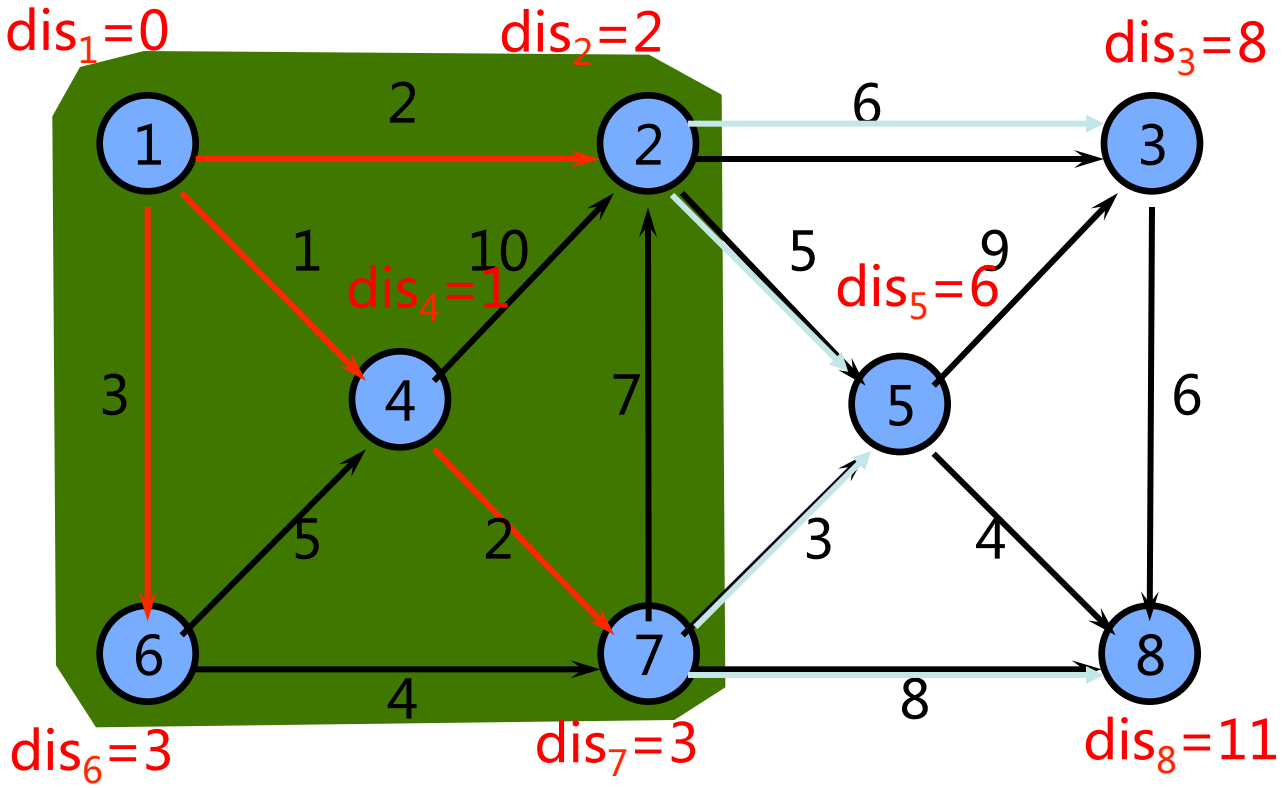
Dijkstra算法

S={1,2,4,6}



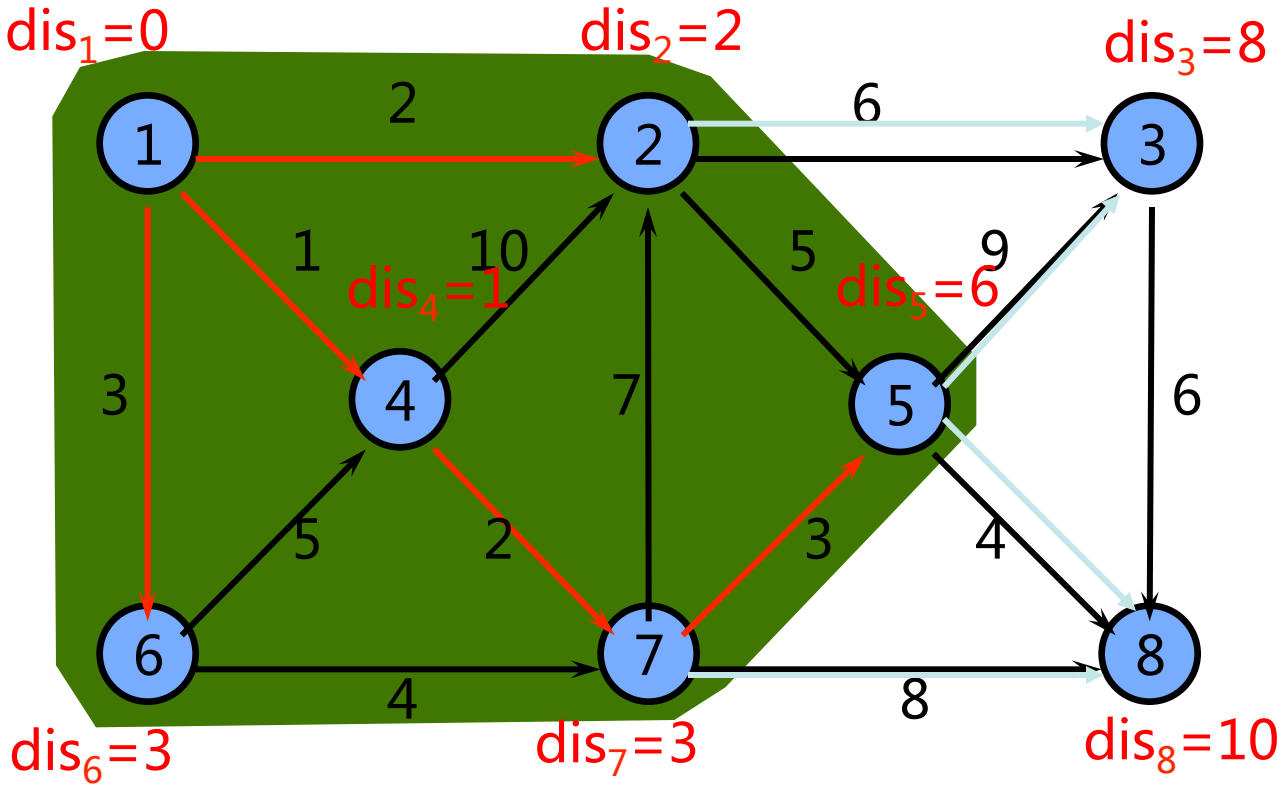
Dijkstra算法

S={1,2,4,6,7}



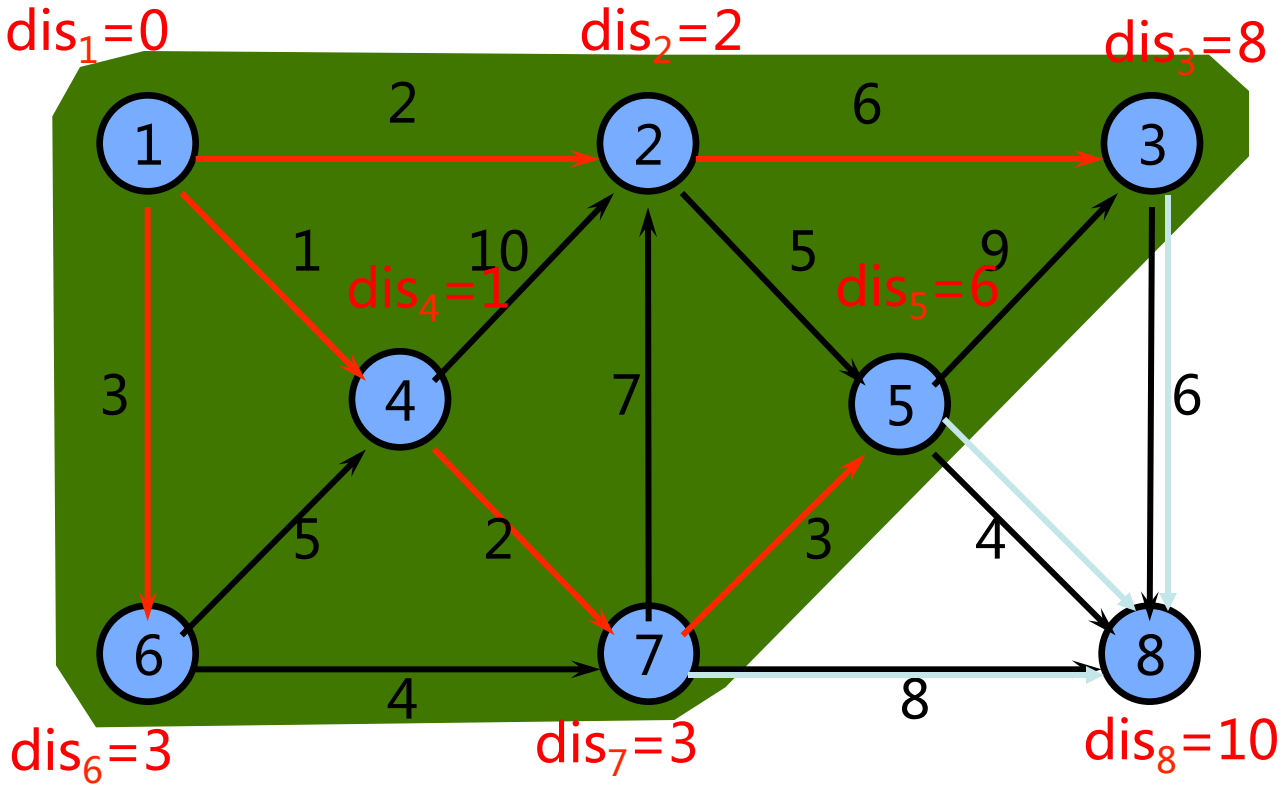
Dijkstra算法

S={1,2,4,5,6,7}



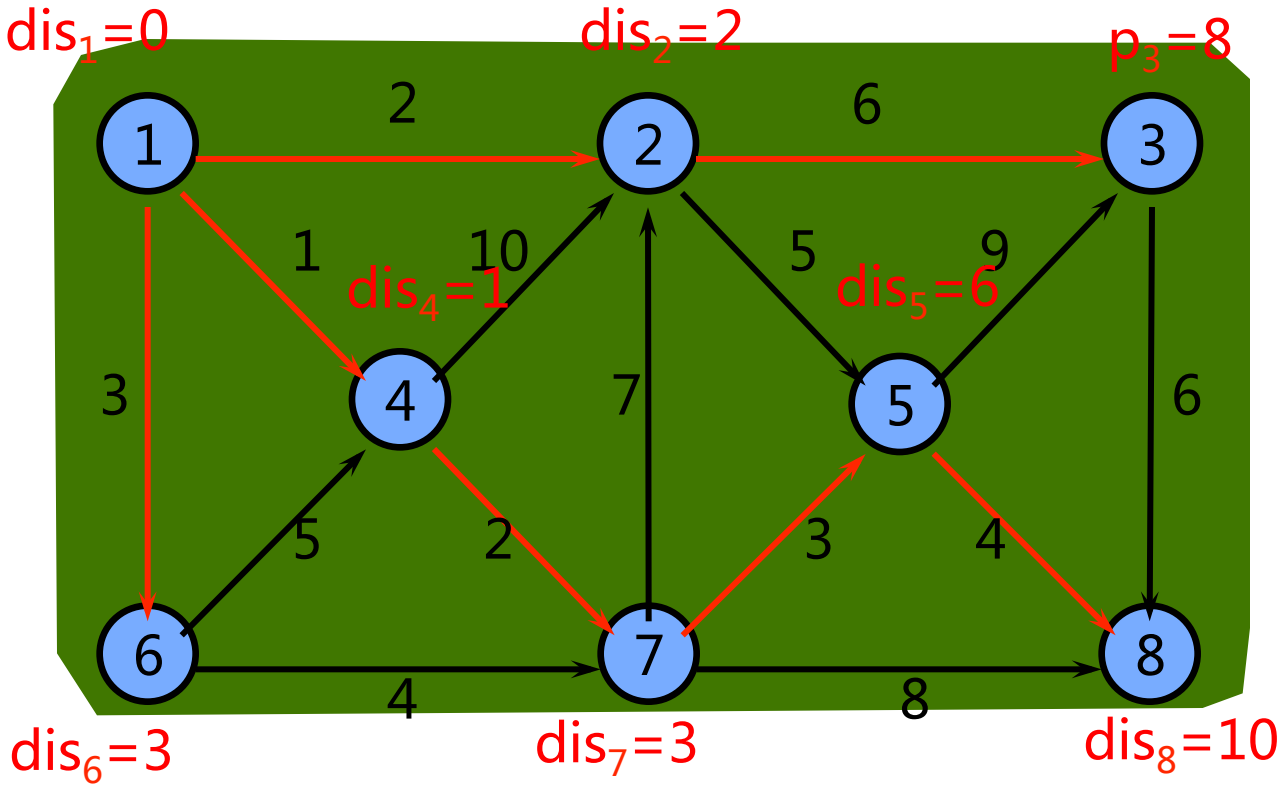
Dijkstra算法

S={1,2,3,4,5,6,7}



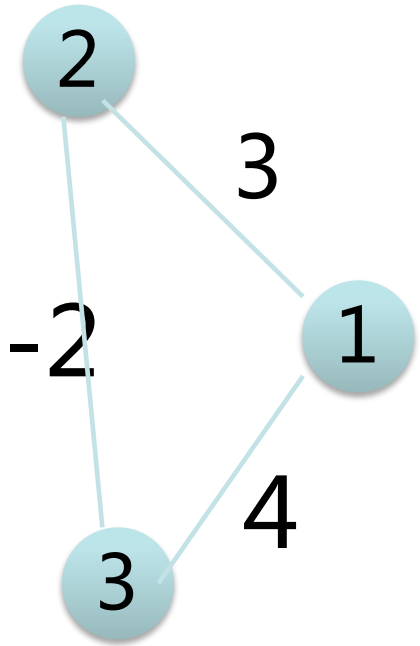
Dijkstra算法

S={1,2,3,4,5,6,7,8}



Dijkstra算法

- Dijkstra算法也适用于无向图，但不适用于有负权边的图。
- d[1,2] = 2
- 但用Dijkstra算法求得 d[1,2] = 3



Dijkstra算法

```
int dis[V];
void Dijkstra(int s)
{
    bool mark[V];        //mark[u]=1表示点u在点集S内.
    for(int i=0;i<V;++i)
    {
        mark[i]=0;
        dis[i]=inf*(i!=s);
    }
    for(int i=0;i<V;++i)
    {
        int mindist=inf,u;
        for(int j=0;j<V;++j)    //找到点集T中dis值最小的点
            if(mark[i]==0&&mindist>dis[j])
            {
                mindist=dis[j];
                u=j;
            }
        mark[u]=1;              //把u加入点集S
        for(int j=0;j<V;++j)    //松弛u相邻的所有边
            if(g[u][j]&&dis[j]>dis[u]+g[u][j])
                dis[j]=dis[u]+g[u][j];
    }
}
```

Dijkstra算法的堆优化

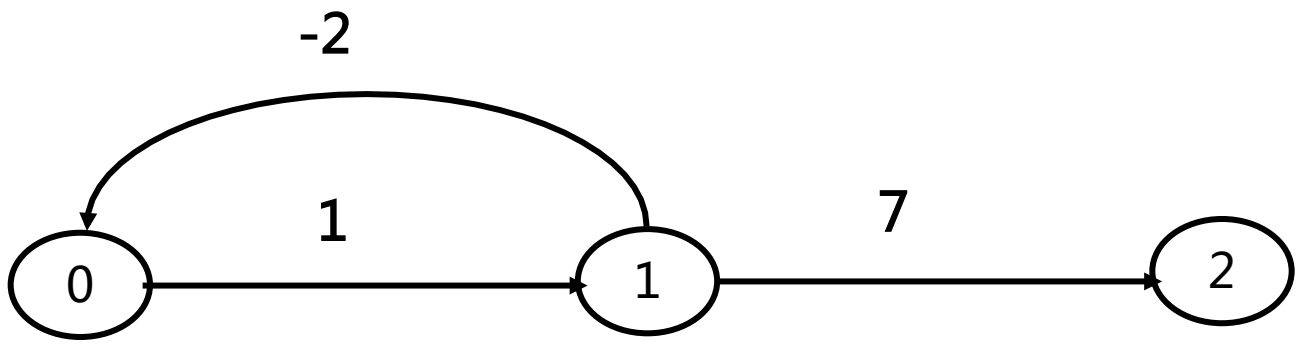
```
using pii=pair<int,int>;
int dis[V];
void Dijkstra(int s)
{
    priority_queue<pii,vector<pii>,greater<pii> >q;      //用优先队列存储节点，节点用pair表示
                                                         //pair里面的first存储dis值
                                                         //second存储节点编号
    for(int i=0;i<V;++i)
        dis[i]=inf*(i!=s);
    q.push(pii(dis[s],s));
    while(!q.empty())
    {
        pii now=q.top();q.pop();
        int u=now.second;
        if(dis[u]<now.first)     //取出来的pair是旧的，直接忽略
            continue;
        for(int i=head[u];~i;i=edge[i].next)
        {
            int v=edge[i].to;
            if(dis[v]>dis[u]+edge[i].w)
            {
                dis[v]=dis[u]+edge[i].w;
                q.push(pii(dis[v],v));
            }
        }
    }
}
```

Bellman-Ford算法

- Bellman-Ford算法：为了能够求解含负权边的带权有向图的单源最短路径问题，Bellman(贝尔曼)和Ford(福特)提出了从源点逐次绕过其他顶点，以缩短到达终点的最短路径长度的方法。
- 开始前把除源点外所有点的dis[]初始化成∞。
- 每次操作对所有边做一次松弛操作，每操作一次都能将一个点的dis更新到最终状态，最多操作|V|-1次就能完成。

Bellman-Ford算法

- 限制条件：要求图中不能包含权值总和为负值回路(负权值回路)，如下图所示。



Bellman-Ford算法

```
int dis[V];
bool Bellman_Ford(int s)
{
    for(int i=0;i<V;++i)
        dis[i]=inf*(i!=s);
    for(int i=1;i<V;++i)
        for(int j=0;j<E;++j)
        {
            int u=edge[j].from;
            int v=edge[j].to;
            int w=edge[j].w;
            dis[v]=min(dis[v],dis[u]+w);     //松弛
        }
    for(int i=0;i<E;++i)
    {
        int u=edge[j].from;
        int v=edge[j].to;
        int w=edge[j].w;
        if(dis[v]>dis[u]+w)                  //判负环
            return false;
    }
    return true;
}
```

SPFA算法

- Bellman-ford的队列优化
- 假设我们现在已经得到了Bellman-Ford 算法某个阶段的dist 数组，然后我们发现了一条s 到u 的距离比dist[u] 更加短的路径。我们更新了dist[u]。
- 那么接下来**直接受到影响的就是与u 直接关联的顶点v**，也就是如果dist[u] + w[u][v] < dist[v] 的话，s 到v 的最短路就可以利用s 到u 的最短路加上u 到v 的边来更新。
- 这样的话与v 直接关联的顶点又会受到影响……不断这样持续下去直到最后没有顶点能被影响。

SPFA算法

- 在竞赛中大多数人会选择用SPFA 作为单源最段路的算法，主要原因在于它比较好写，而且通常情况下跑得比较快。但是SPFA 的时间复杂度实际上是没有保证的，最坏情况基本和Bellman-Ford 相同，但是最好的时候可以到达O(|V|+|E|)。
- 非常重要的一点就是SPFA 的队列需要使用循环队列，虽然最多队列里只会有n各点，但是每个点可能会入队多次。

SPFA算法

```
dis[s]=0; q.push(s); //起点入队
while(!q.empty())
{
    s=q.front(); q.pop(); in[s]=false; //取出队头
    for(i=last[s];i;i=e[i].next) //枚举由s出发的边的中点
    {
        if(e[i].v+dis[s]<dis[e[i].to]) //更新最短路
        {
            dis[e[i].to]=e[i].v+dis[s];
            if(!in[e[i].to]) //更新后的点入队
            {
                q.push(e[i].to);
                in[e[i].to]=true;
            }
        }
    }
}
```

Floyd算法

- **动态规划思想**
  - F[i,j]表示i到j的经过小于k的点所能得到的临时最短路
  - 枚举中转点k
  - if (f[i][k] + f[k][j] <= f[i][j]) f[i][j] = f[i][k] + f[k][j];
- **代码实现**

```
int dis[V][V];
void Floyd()
{
    for(int k=0;k<V;++k)
        for(int i=0;i<V;++i)
            for(int j=0;j<V;++j)
                dis[i][j]=min(dis[i][j],dis[i][k]+dis[k][j]);
}
```

目录

- 图的基础知识
- 最短路算法
- 最小生成树算法
  - 并查集
  - Prim算法
  - Kruskal算法
- 例题选讲

并查集

- 有这样的问题：初始时n个元素分属不同的n个集合，通过不断的给出元素间的联系，要求实时的统计元素间是否直接或间接地存在联系。

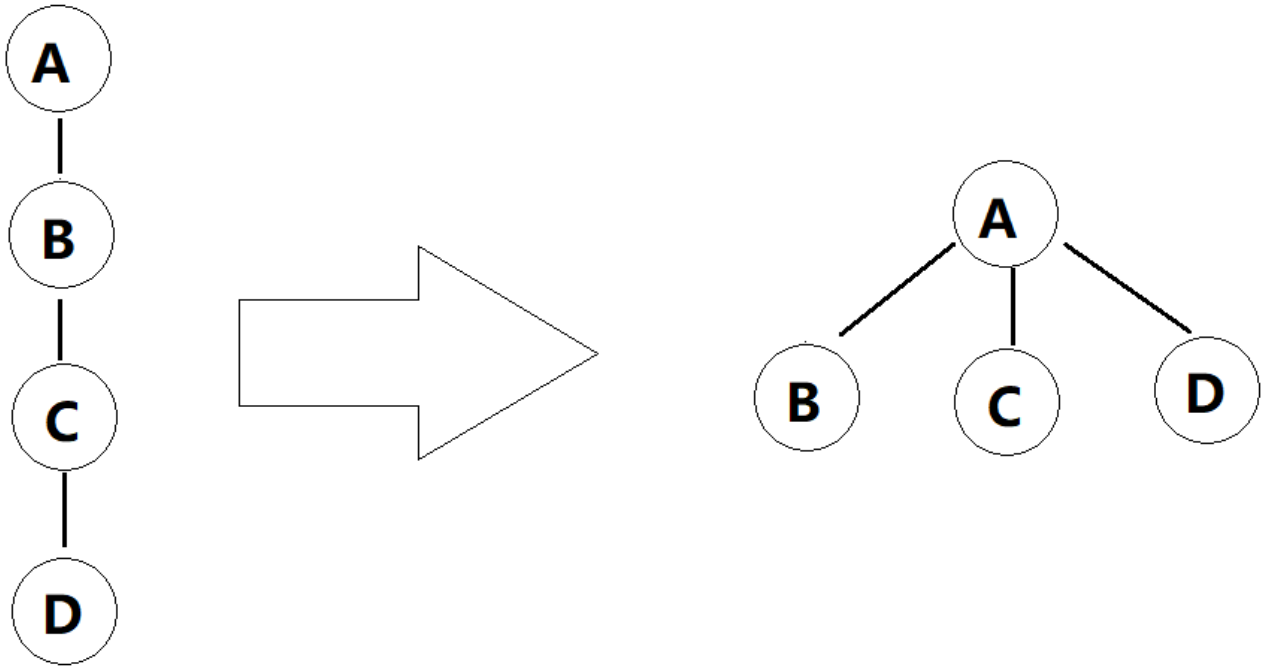
输入关系	分离集合
初始状态	{1} {2} {3} {4} {5} {6} {7} {8} {9} {10}
(2,4)	{1} {2,4} {3} {5} {6} {7} {8} {9} {10}
(5,7)	{1} {2,4} {3} {5,7} {6} {8} {9} {10}
(1,3)	{1,3} {2,4} {5,7} {6} {8} {9} {10}
(8,9)	{1,3} {2,4} {5,7} {6} {8,9} {10}
(1,2)	{1,2,3,4} {5,7} {6} {8,9} {10}
(5,6)	{1,2,3,4} {5,6,7} {8,9} {10}
(2,3)	{1,2,3,4} {5,6,7} {8,9} {10}

并查集

```
int fa[V];
void find(int x)
{
    return x==fa[x]?x:find(fa[x]);
}
void merge(int x,int y)
{
    x=find(x);
    y=find(y);
    if(x!=y)
        fa[y]=x;
}
```

并查集

- 优化：**路径压缩**
- 每次寻找最远祖先时，在找到最远祖先后，寻找的路径上的点的最远祖先都获取到了，不妨直接丢弃当前父节点，记录最远祖先。那么以后访问他们的最远祖先的复杂度都是O(1)了。



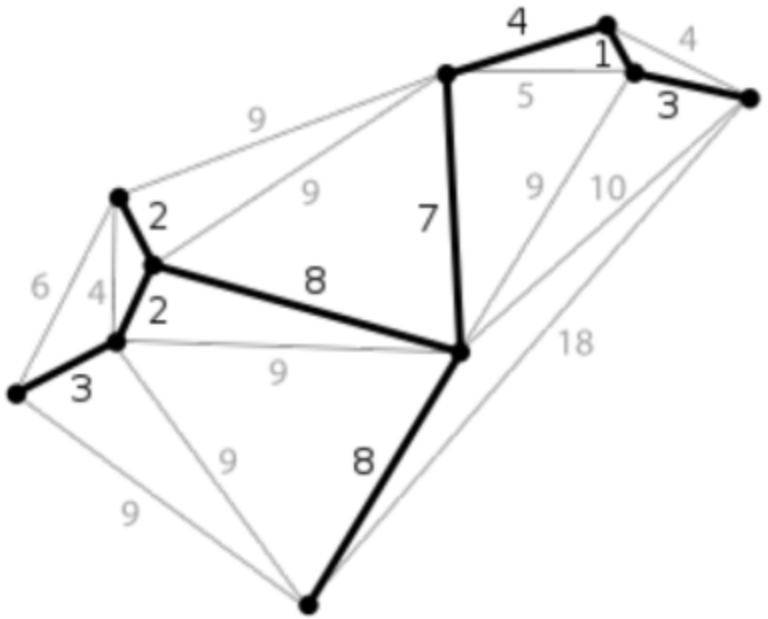
并查集

```
int fa[V];
void find(int x)
{
    return x==fa[x]?x:fa[x]=find(fa[x]);
}
void merge(int x,int y)
{
    x=find(x);
    y=find(y);
    if(x!=y)
        fa[y]=x;
}
```

```
int fa[V];
void find(int x)
{
    return x==fa[x]?x:find(fa[x]);
}
void merge(int x,int y)
{
    x=find(x);
    y=find(y);
    if(x!=y)
        fa[y]=x;
}
```

最小生成树

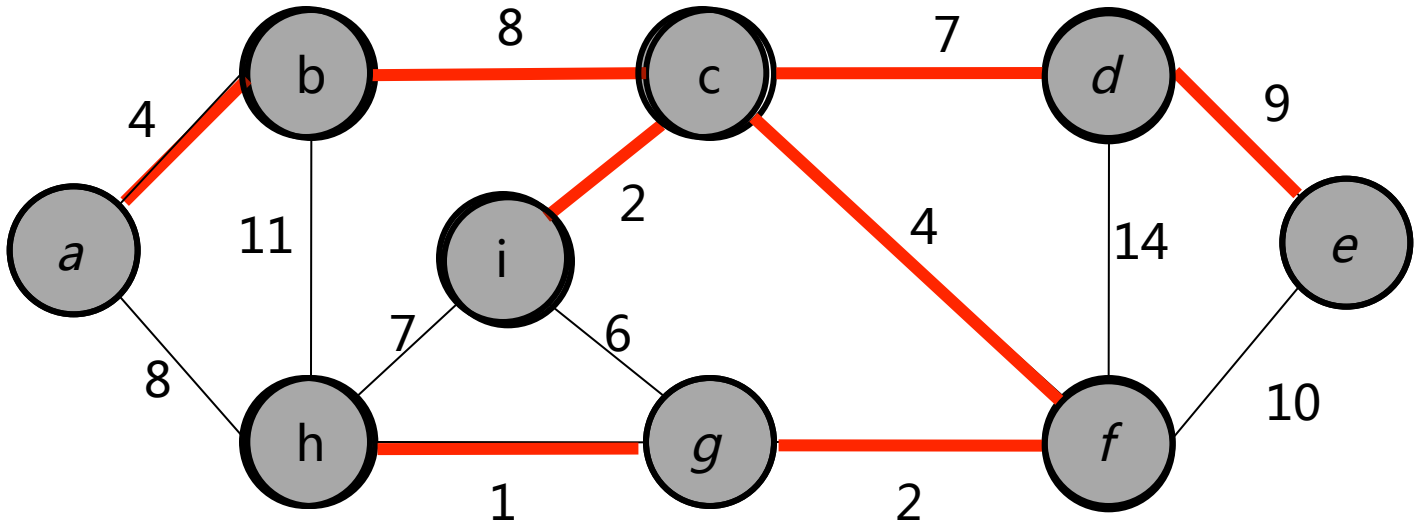
- **生成树**：无向连通图G 的一个子图如果是包含G 的所有顶点的树，那么就称这个子图为G 的生成树。
- 我们称生成树各边权值和为该树的权。对于无向连通图来说，权最小的生成树被成为**最小生成树**。



Prim算法

- 先建立一个只有一个结点的树，这个结点可以是原图中任意的一个结点。
- 使用一条边扩展这个树，要求这条边一个顶点在树中另一个顶点不在树中，并且这条边的**权值要求最小**。
- 重复上一个步骤直到所有顶点都在树中。

Prim算法



Prim算法

- 类似dijkstra

```
int dis[V];
int Prim(int s)     //第一个加进树的点是s
{
    int ret=0;
    bool mark[V];       //mark[u]=1表示点u在树内.
    for(int i=0;i<V;++i)
    {
        mark[i]=0;
        dis[i]=inf*(i!=s);
    }
    for(int i=0;i<V;++i)
    {
        int mindist=inf,u;
        for(int j=0;j<V;++j)    //在树以外的点中，找到dis值最小的点
            if(mark[i]==0&&mindist>dis[j])
            {
                mindist=dis[j];
                u=j;
            }
        ret+=mindist;
        mark[u]=1;              //把u加入树
        for(int j=head[u];~j;j=edge[j].nxt) //更新dis值
        {
            int v=edge[j].to;
            dis[v]=min(dis[v],edge[j].w);
        }
    }
    return ret;
}
```

Kruskal算法

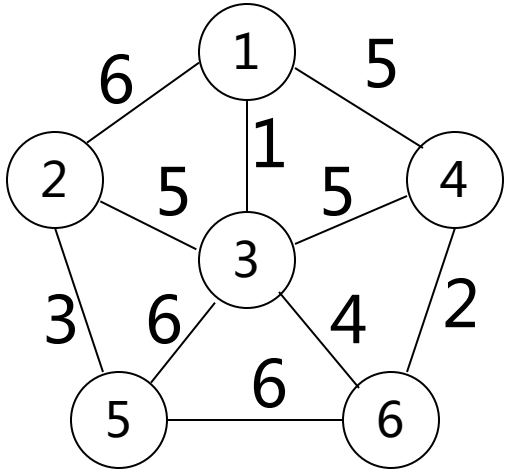
- Kruskal 算法是能够在O(mlogm) 的时间内得到一个最小生成树的算法。

它主要是基于贪心的思想：

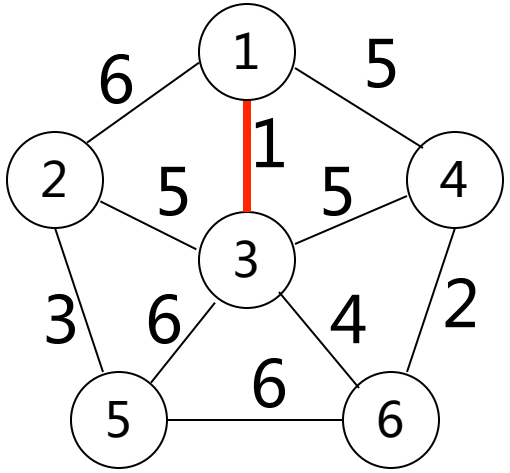
  - 将边按照边权从小到大排序，并建立一个没有边的图T。
  - 选出一条没有被选过的边权最小的边。
  - 如果这条边两个顶点在T 中所在的连通块不相同，那么将它加入图T。
  - 重复上述2个步骤直到图T 连通为止。

- 由于只需要维护连通性，可以不需要真正建立图T，可以用**并查集**来维护。

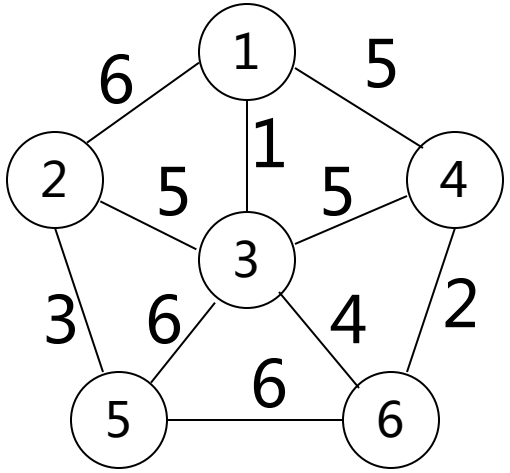
Kruskal算法



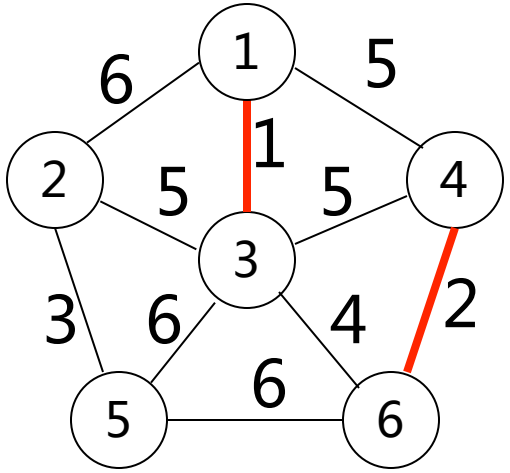
原始图



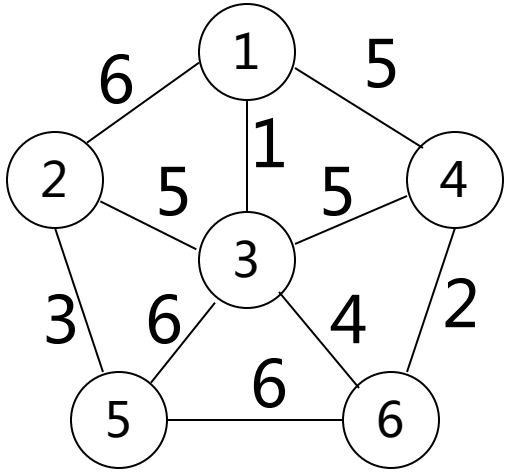
Kruskal算法



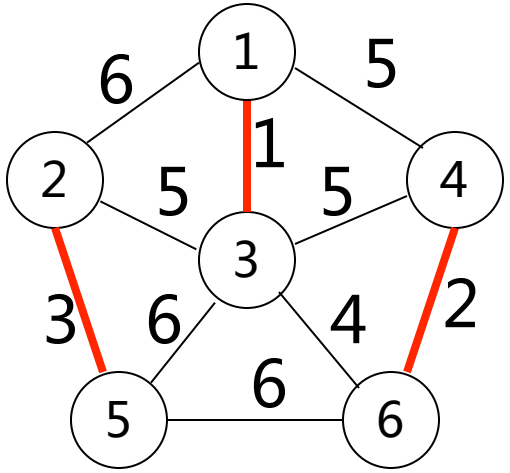
原始图



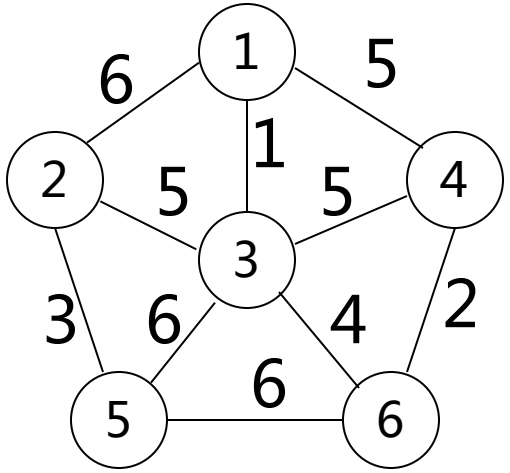
Kruskal算法



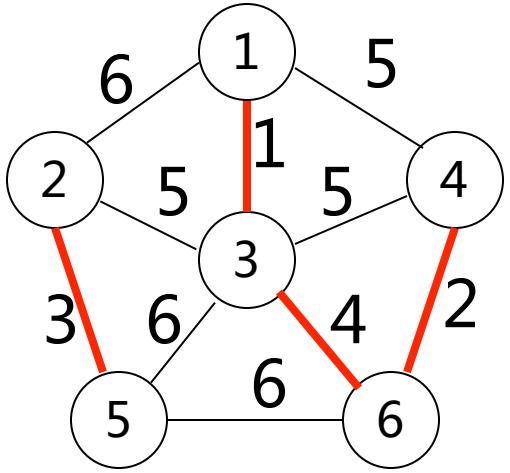
原始图



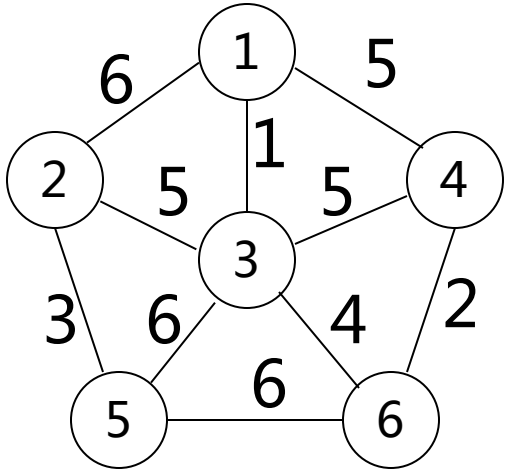
Kruskal算法



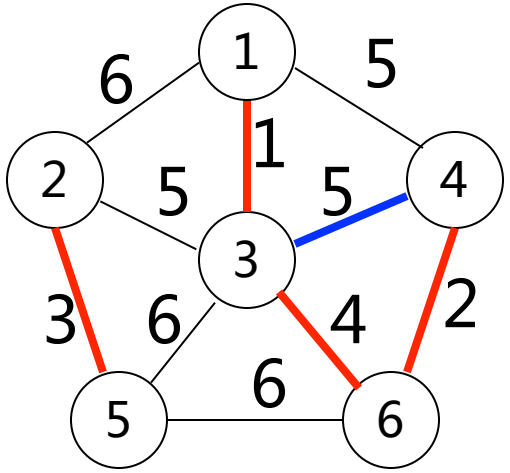
原始图



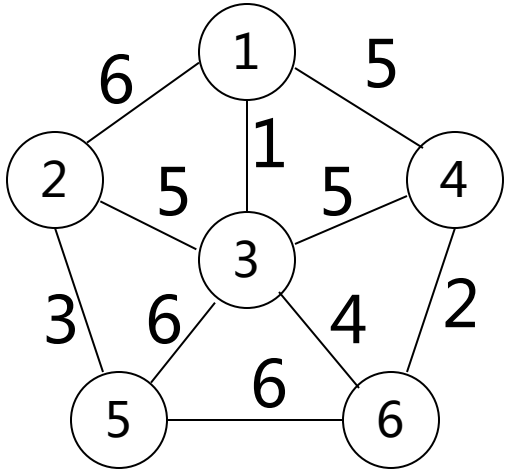
Kruskal算法



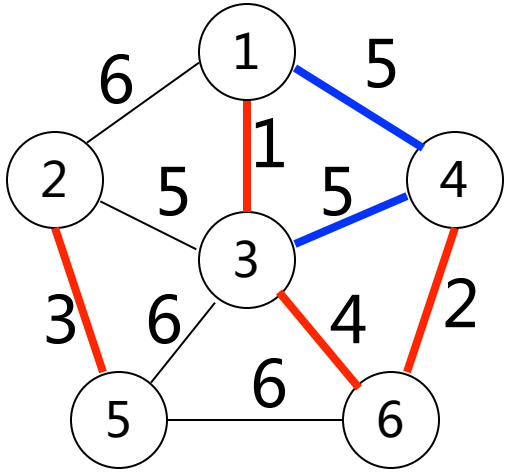
原始图



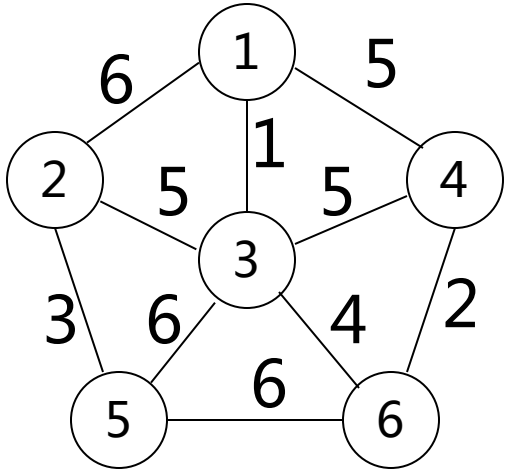
Kruskal算法



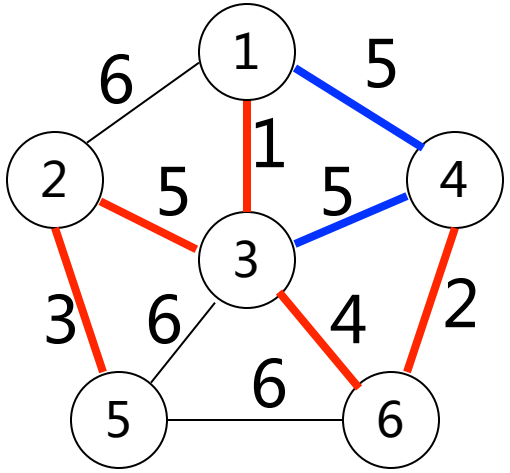
原始图



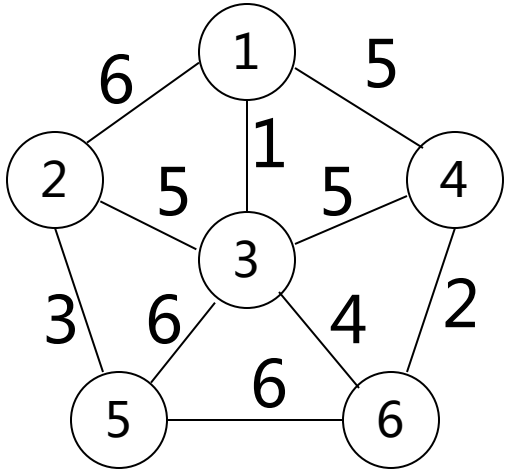
Kruskal算法



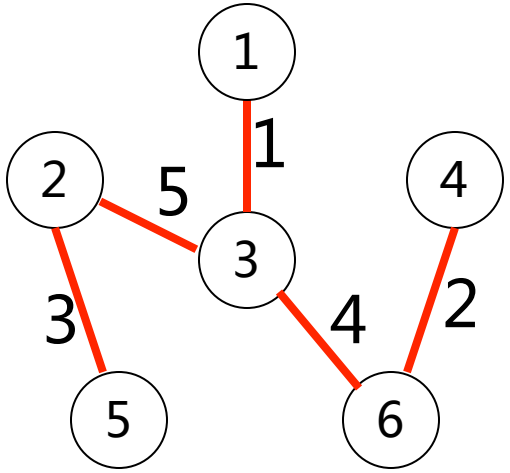
原始图



Kruskal算法



原始图



最小生成树

Kruskal算法

```
struct EDGE
{
    int u,v,w;
}edge[E];
int fa[V];
int find(int x){return x==fa[x]?fa[x]=find(fa[x]);}
void init()
{
    for(int i=0;i<V;++i)
        fa[i]=i;
}
bool merge(int x,int y)
{
    x=find(x);
    y=find(y);
    if(x==y) return 0;  //表示x和y已经在同一联通块中;
    fa[y]=x; return 1;  //合并;
}
```

```
int Kruskal()
{
    init();
    sort(edge,edge+E,[](const edge& a,const edge& b)->bool{return a.w<b.w;})
    int cnt=0;  //记录加边次数，加够V-1条边的时候生成树完成.
    int ret=0;  //记录答案：最小生成树权值和.
    for(int i=0;cnt<V-1;++i)
    {
        EDGE& e=edge[i];
        if(merge(e.u,e.v))
        {
            cnt++;
            ret+=e.w;
        }
    }
    return ret;
}
```

比较

- Prim和Kruskal的贪心策略是一样的，都是选耗费最小的边：
  - 对于Prim，其选取的边(u,v)必有一个顶点已经被覆盖，另一个顶点未被覆盖。
  - 而对于Kruskal,其选取的边(u,v)任意，只要这个边的加入不能使被覆盖的顶点构成回路。
- prim 算法　　(复杂度O(n^2))——适用于稠密图
- kruskal算法　　(复杂度O(m*lg(m))——适用于稀疏图

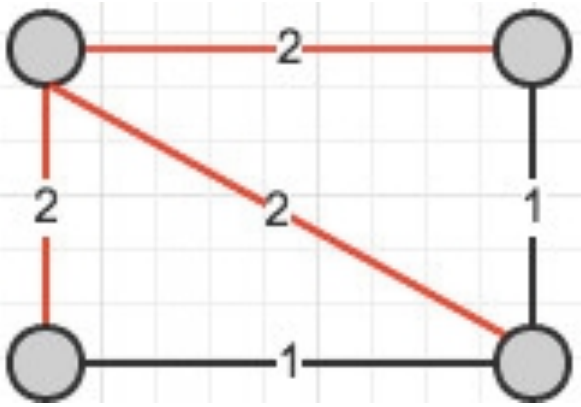
瓶颈生成树

- 无向图G的一颗瓶颈生成树是这样的一颗生成树T，它最大的边权值在G的所有生成树中是最小的。瓶颈生成树的值为T中最大权值边的权。
- 一个结论：无向图的最小生成树一定是瓶颈生成树，但瓶颈生成树不一定是最小生成树。

瓶颈生成树

- 命题：无向图的最小生成树一定是瓶颈生成树。
- 证明：可以采用反证法予以证明。假设最小生成树不是瓶颈树，设最小生成树T的最大权边为e，则存在一棵瓶颈树Tb，其所有的边的权值小于w(e)。删除T中的e，形成两棵数T1; T2，用Tb中连接T1; T2的边连接这两棵树，得到新的生成树，其权值小于T，与T是最小生成树矛盾。

- 命题：瓶颈生成树不一定是最小生成树。



[UVA 11374]Airport Express

- N个点，从起点S到终点E，有M条经济和K条商业线，只能乘坐一次商业线，求最短时间。
- $N \leq 500$, $M + K \leq 2000$。

[UVA 11374]Airport Express

- 枚举选择商业线。
- 以S和E为起点分别跑一次最短路即可。
- 在不使用商务车票的情况下计算两次最短路，分别算出起点s到每一点的最短路ds[N]， 以及终点t到每一点的最短路dt[N]，并记录起点到终点的最短路Min。然后枚举商务车票（商务车票初始站点u， 终止站点v， 花费cos），使Min = min（ds[u] + dt[v] + cos, ds[v] + dt[u] + cos）,并记录下使用的商务车票的u和v。维护过后的Min就是在最优的情况下使用商务车票的最短路花费，然后根据pre数组，以及记录的u和v输出路径。

[BZOJ 4152]The Captain

- 给定平面上的n个点，定义(x1, y1)到(x2, y2)的费用
- 为min(|x1 – x2|, |y1 – y2|)，求从1号点走到n号点的最小费用。
- 2 ≤ n ≤ 2 × 10^5。

[BZOJ 4152]The Captain

- 考虑建边，如果暴力建边则需要建n^2条边，显然不可接受。
- 可以分析一下边权的性质：对于(x1,y1)(x2,y2) (x3,y3)这三个点若x1 ≤ x2 ≤ x3，则从x方向由P1– > P3的边可以由(P1– > P2) + (P2– > P3)组成，所以P1– > P3的边可以不用显式的建出来，而是由P1– > P2 和P2– > P3 的边构成。
- 注意：本题最短路算法卡SPFA。

[BZOJ 2763][LUOGU 4568]飞行路线

- Alice和Bob现在要乘飞机旅行，他们选择了一家相对便宜的航空公司。该航空公司一共在n个城市设有业务，设这些城市分别标记为0到n – 1，一共有m种航线，每种航线连接两个城市，并且航线有一定的价格。Alice和Bob现在要从一个城市沿着航线到达另一个城市，途中可以进行转机。航空公司对他们这次旅行也推出优惠，他们可以免费在最多k种航线上搭乘飞机。那么Alice和Bob这次出行最少花费多少？
- $2 \le n \le 10000$; $1 \le 5 \times 10^4$; $0 \le k \le 10$。

[BZOJ 2763][LUOGU 4568]飞行路线

- 经典的分层图。
- 可以发现免费搭乘的次数k非常小，自然的想到使用拆点法进行建图。
- 把每个节点i拆成k + 1个节点(i,j)其中0 ≤ j ≤ k，表示从起点到第i个点，且恰使用了j次免费搭乘的最小花费。对每层k按原图权值建边，对每条边(u,v,c)额外建边(u,j)−>(v, j+1)，费用为0，表示使用了一次免费飞行。再跑从起点到终点的最短路即可。

[BZOJ 2763][LUOGU 4568]飞行路线

- 解法二？
- 一条路径中将边权大的边使用免费搭乘一定更划算。
- 二分mid，将便权大于mid的边设为0，其余设为1，求最短路是否不超过K。

[BZOJ 1774]过路费

- n个点，m条双向边，每个点有权值c_i，每条边有权值w_i，一条路径的费用=每条边的权值和+各个点的权值的最大值，即sum(w_i)+max(c_i)。q个询问，问x到y的最小费用。
- n ≤ 250; m ≤ 10000。

[BZOJ 1774]过路费

- floyd的三重循环中，第一层k限定了当前任意的i到j的最短路径都是由1到k所更新的，也就是i到j的路径中不经过除此以外的点。
- 点按c_i排序，k循环按点从小到大循环，即i到j的路径中不经过c_x > c_k的点。所以我们可以知道当前更新dis[i][j]
- 的dis[i][k]和dis[k][j]这两条路径中点权的最大值max{ci,cj,ck}

[LUOGU 1119] 灾后重建

- 给出B地区的村庄数N，村庄编号从0到N − 1，和所有M条公路的长度，公路是双向的。并给出第i个村庄重建完成的时间t[i]，你可以认为是同时开始重建并在第t[i]天重建完成，并且在当天即可通车。若t[i]为0则说明地震未对此地区造成损坏，一开始就可以通车。之后有Q个询问(x; y; t)，对于每个询问你要回答在第t天，从村庄x到村庄y的最短路径长度为多少。如果无法找到从x村庄到y 村庄的路径，经过若干个已重建完成的村庄，或者村庄x或村庄y在第t天仍未重建完成，则需要返回−1。
- 数据保证t[i] < t[2] < · · · < t[n]。
- 1 ≤ n ≤ 200, 1 ≤ Q ≤ 5 × 10^5。

[LUOGU 1119] 灾后重建

- 在第i 个村庄重建完成时，前面村庄都重建完成了，后面的村庄都没有重建完成。
- 最短路只能经过前i 个村庄。
- Floyed!

[POJ 3660]Cow Contest

- 有n头牛，给你m对关系(a, b)表示牛a能打败牛b，求在给出的这些关系下，能确定多少牛的排名。
- 1≤ n≤ 200

[POJ 3660]Cow Contest

- Floyed算法的拓展：Floyd-Wareshall。
- 初始时，若两头牛关系确定则f[i,j] = 1。
- 对于一头牛若确定的关系=n-1这说明这头牛的排名是确定的。

[BZOJ 2143]飞飞侠

- 一个N × M的矩形方阵，每个格子代表一个街区，对于第i行第j列的街区，使用有Aij的费用就可以弹跳到距离不超过Bij 的街区。告诉你三个人的坐标，需要前往其中的某个人那里集合，问到哪一个人那儿的总费用最低。
- 1 ≤ N,M ≤ 150; 0 ≤ Bij ≤ 100; 0 ≤ Aij ≤ 1000。

[BZOJ 2143]飞飞侠

- 这道题点很少，但是边可能很多，直接建图做最短路显然不可行。
- d[i][j][k]表示在(i,j)这个点可以走k步的最少费用，每次只能向四个方向走，或者原地不动即可。
- 等同于对于距离h的一个点，建5条边，分别连向四个方向和当前点能量为h − 1的点。这就体现了你在某个点弹一次就可以到达一些点。
  - 走到相邻一格或不走：d[i][j][k]− > d[x][y][k − 1]，其中(i; j)和(x; y)相邻或相等，k > 0。
  - 选择一次弹射：d[i][j][0] + a[i][j]− > d[x][y][b[i][j]]。
- 这样做的好处是建的边不会太多，点数和边数同级，用三遍SPFA最短路算法解决即可。

[BZOJ 1232]安慰奶牛cheer

- n个点m条双向边，要求去掉一些边仅保留n – 1条边，每个点有各自的点权，每条边有各自的边权，要求从一个点出发遍历每一个点再回到起点，每经过一次点i，代价就加上一次该点点权，每经过一次边j，代价就加上一次该边的边权，求最小的代价。

[BZOJ 1232]安慰奶牛cheer

- 每一条边对答案的贡献= 该边的边权×2 + 两端点点权以此作为它的边权跑最小生成树。
- 然后我们会发现，作为起点的点又额外加了一次，而对于我们求出的最小生成树，无论选哪个点为起点，所需的代价是一样的即新边权的和。
- 我们只需要选择点权最小的点作为起点，最后将答案加上它一次即可。

[BZOJ 1601][Usaco2008 Oct]灌水

- Farmer John已经决定把水灌到他的n(1<=n<=300)块农田，农田被数字1到n标记。把一块土地进行灌水有两种方法，从其他农田饮水，或者这块土地建造水库。建造一个水库需要花费wi,连接两块土地需要花费pij。计算Farmer John所需的最少代价。
- 1 ≤ N ≤ 300; 1 ≤ wi ≤ 10^5; 1 ≤ pij ≤ 10^5。

[BZOJ 1601][Usaco2008 Oct]灌水

- 每个水库要么选择自己这里建造水库，要么选择连一条边到已建成的水库。
- 假设所有的水库最终都选择好了一个决策的话，那么整个图就是，分成m块，每一块有一个点是自己建造水库的。其他都是顺着边连到这个点的。也就是在这个子图当中做最小生成树。
- 加一个超级源，每个点向源连花费wi边。然后在整个图中做最小生成树。超级源的连通保证了至少有一个点建造了水库

[BZOJ 2654]tree

- 给你一个V个点E条边无向带权连通图，每条边是黑色或白色。让你求一棵最小权的恰好有need条白色边的生成树。题目保证有解。
- V ≤ 50000; E ≤ 100000。

[BZOJ 2654]tree

- 白边的权值越大，被选入的条数会越少。
- 二分一个值，然后每次给白边加上这个值。看一下最小生成树中白边的个数。最后答案再把它减去。
- 二分的正确性？即如果给白色边边权加上mid，则所选白色边> need，如果加上mid + 1，则所选白色边< need。
- 解决方法是，在排序的时候，我们将白色边放在相同长度的黑色边之前。这样，因为mid + 1时白边< mid，所以一定有若干= mid的黑边。在mid时，我们多选的白边就可以被黑边替换掉。所以在最后统计答案的时候，只需要ans−= mid × need即可。

[UVA 10816]Travel in Desert

- 一群人在沙漠中，给定了n个点，m条路，双向的。每条路有一定的长度且路上的温度也不一样。现在这群人想从s到t去，要使路径中的最高温度最低，有多条路径的情况下选择路程最短的，输出路径，最高温度最低的路程。
- $1 \leq n \leq 100; 1 \leq m \leq 10000$。

[UVA 10816]Travel in Desert

- 如果只考虑最小的最大热度，那么本题就是一个最小瓶颈路问题，只需按照热度找一棵最小生成树即可。但是，如果这样的路径有多个，实际上是最小生成树有多个时，要找到最短路径，还得把热度不大于最小生成树中最大热度的边并且没在生成树中的边加到最小生成树中，然后再找最短路。

[BZOJ 2200]道路与航线

- Farmer John 正在一个新的销售区域对他的牛奶销售方案进行调查。他想把牛奶送到 T （1≤T≤2.5×10^4）个城镇 ，编号为 1 到 T。这些城镇之间通过 R 条道路（编号为 1 到 R）和 P 条航线（编号为 1 到 P）连接。每条道路 i 或者航线 i 连接城镇 Ai到 Bi，花费为 Ci。对于道路，0≤Ci≤10^4，然而航线的花费很神奇，花费 Ci可能是负数。道路是双向的，可以从 Ai到 Bi，也可以从 Bi到 Ai，花费都是 Ci。然而航线与之不同，只可以从 Ai到 Bi。
- 事实上，由于最近恐怖主义太嚣张，为了社会和谐，出台了一些政策保证：如果有一条航线可以从 Ai到 Bi，那么保证不可能通过一些道路和航线从 Bi回到 Ai。由于 FJ 的奶牛世界公认十分给力，他需要运送奶牛到每一个城镇。他想找到从发送中心城镇 S 把奶牛送到每个城镇的最便宜的方案，或者知道这是不可能的。
- ,1≤R,P≤5×10^4

[BZOJ 2200]道路与航线

- 单源最短路问题，但有负边权不能使用Dijkstra，若用SPFA求解，因为测试数据经过了特殊构造，会超时。
- 题目条件：①双向边边权非负数②单向边边权可能为负但不构成环
- 双向边构成若干连通块，加上单向边连通块构成有向无环图。
- 堆优化的dijkstra+拓扑排序

[POJ1734]Sightseeing trip

- 给定图的N个点M条边，求出图中的最小环(无向图，有重边),至少包含3个点。
- 1≤N≤100。

[POJ1734]Sightseeing trip

- 第k层循环开始前，d[i,j]保存这编号仅经过1到k-1的最短路。
- min{d[i,j]+a[j,k]+a[k,i]}

[POJ 3613]Cow Relays

- 给定一个M(2 <= M <= 100)条边的无向图，求S到E恰好经过N(2 <= N <= 1000000)条边的最短路。

[POJ 3613]Cow Relays

- 将每条边的两个点进行映射为1到P的整数（P<=2M）。
- A[i,j]表示从i到j经过经过若干条边的最短路(1<=i,j<=P)。
- a矩阵表示经过n条边，b矩阵表示经过m条边，c表示经过m + n条边
- 满足广义矩阵乘法：c[i][j] = min(c[i][j],a[i][k]+b[k][j]);

[CH 6201]走廊泼水节

- 在一棵最小生成树的基础上增加一些边变成一张完全图，但是这张图的最小生成树仍然是原来的树，求增加的边的边权的和最小是多少。
- N≤6000

[CH 6201]走廊泼水节

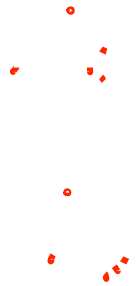
- 类似Kruskal。
- 设当前的边（x,y,z），x所在的并查集为Sx，y所在的并查集为Sy，此时应当合并这两个S。
- 假设u∈Sx,v∈Sy,若（u,v）≠（x,y），则在最终的完全图上，我们肯定要在uv之间加上一条边，于是，这条边与uv在生成树上的路径构成了一个环，但是为了保证(x,y)依然在最小生成树上，那么(u,v)的边权一定大于(x,y)的边权，为了让答案最小化，这个边权为z+1。
- 那么，Sx，Sy之间一共会增加|Sx|*|Sy|-1条边，我们只需要将 (z+1)*(|Sx|*|Sy|-1)累加到答案中即可。

[CH 6202]黑暗城堡

- 黑暗城堡有N个房间 (1≤N≤1000)，M条可以制造的双向通道，以及每条通道的长度。
- 城堡满足下面的条件：设 D[i] 为如果所有的通道都被修建，第 i 号房间与第1号房间的最短路径长度；而 S[i] 为实际修建的树形城堡中第 i 号房间与第1号房间的路径长度；要求对于所有整数 i(1≤i≤N)，有 S[i]=D[i] 成立。
- 问有多少种不同的城堡修建方案，答案对 2^31–1 取模。

[CH 6202]黑暗城堡

- 若x是y的父亲，d[y]=d[x]+w[x][y]
- 按d数组排序，枚举统计每一步转移满足的数量，累乘即可。



谢 谢

欢迎提问