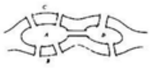
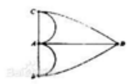
第四章 图论基础

第四章 图论基础

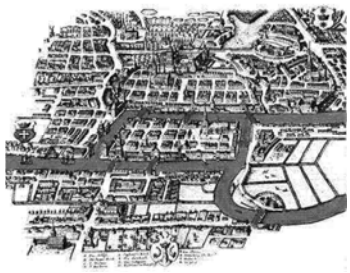
图论起源

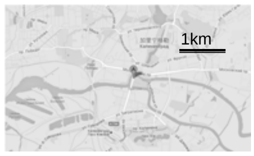
- 哥尼斯堡七桥问题；
- 1736年，29岁的欧拉发表了《哥尼斯堡的七座桥》的论文；
- 回答当地居民的散步问题，证明了更为广泛的有关一笔画的三条结论，称为“欧拉定理”。





哥尼斯堡/加里宁格勒的位置

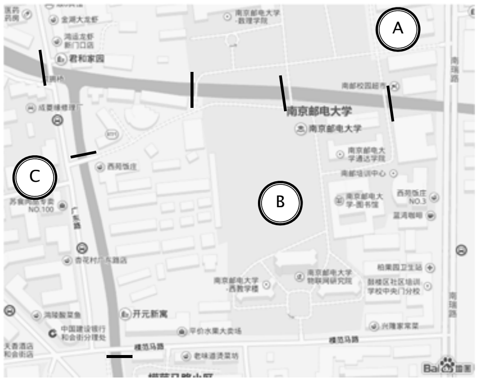




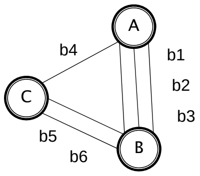
欧拉定理

- 连到一点的数目如是奇数条，就称为奇点，如果是偶数条就称为偶点；
- 连通图可以一笔画的重要条件是：奇点的数目不是0 个就是2 个；
- 要想一笔画成，必须中间点均是偶点，也就是有来路必有另一条去路，奇点只可能在两端，因此任何图能一笔画成，奇点要么没有要么在两端。

南邮三牌楼六桥问题



抽象化定义



E **:= {b1, b2, b3, b4, b5, b6}**
V **:= {A, B, C}**
G **:= (*V*, *E*)**，序偶

|*E*| = 6，size of graph（尺度）
|V| = 3，order of graph（阶数）

deg(A) = 4，节点A的度数
deg(B) = 5
deg(C) = 3
deg(*G*) = 12 = 2|E|，图G的度数

<deg(*v*)> = 2|*E*| / |*V*|
= 4

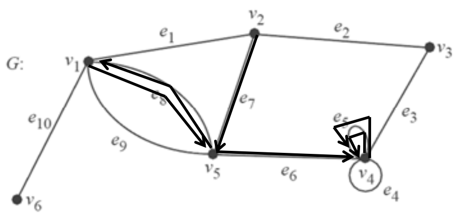
Q：证明

形式化定义

途径（Walk）

- 图中存在关联关系的点边，交替出现的序列

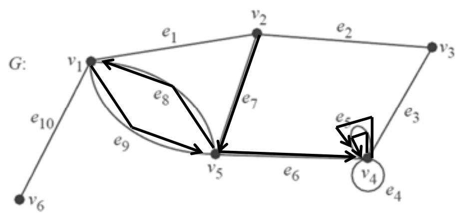
$v_2, e_7, v_5, e_8, v_1, e_8, v_5, e_6, v_4, e_5, v_4, e_5, v_4$



迹（Trail）

- 无重复边的walk

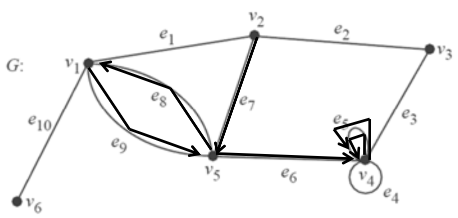
$v_2, e_7, v_5, e_8, v_1, e_8/e_9, v_5, e_6, v_4, e_5, v_4, \boxed{e_5, v_4}$



路（Path）

- 除始末点外无重复点的walk/trail

$v_2, e_7, v_5, e_8, v_1, \boxed{e_8, v_5, e_6, v_4, e_5, v_4, e_5, v_4}$



其他常用术语

- 开途径：起末点不相同的途径
- 闭途径：超末点相同的途径
- 开迹、闭迹/回路(Circuit)
- 开路、闭路/圈(Cycle)
- 途径长度：途径所含边的个数
- 奇/偶圈：圈的长度为奇数/偶数
- 最短路：两点间长度最小的路
- 图的直径：所有最短路的最大值
- 图的围长：最短圈的长度
- 图的周长：最长圈的长度

图的运算

图的关系运算

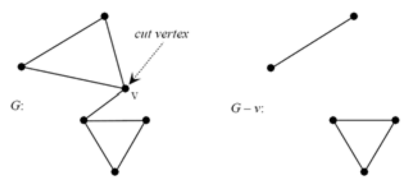
- $G_1(V_1, E_1)$，$G_2(V_2, E_2)$
- G_1为G_2的子图，iff.，V_1 是 V_2 子集，且 E_1是 E_2子集。
- G_1为G_2的生成子图， $V_1 = V_2$ 。
- G_1为G_2的边导出子图， V_1包含E_1的所有端点。
  - $G_1 = G_2[E_1]$
- G_1为G_2的点导出子图， E_1包含V_1的所有关联边。
  - $G_1 = G_2[V_1]$
- G_1为G_2补(图)， iff.，V_1=V_2，且E_1是E_2补集。

其他运算

- 并
  - $G_1(V_1, E_1) \cup G_2(V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2)$
- 交
  - $G_1(V_1, E_1) \cap G_2(V_2, E_2) = (V_1 \cap V_2, E_1 \cap E_2)$
- 异或/环和
  - $G_1(V_1, E_1)$^$G_2(V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2)[E_1$^$E_2]$

点割集(Vertex Cut)

- 点割集 F，是 E 的子集，当
  - $G(V, E) - F = (V - F, E - \{e_{ij} \mid i$ or j 属于$F\})$ 是不连通的；
  - $G(V, E) - H$ 是连通的，对于所有 H 是 F 的真子集。
- 割点v，
  - F 仅含点v



树与林

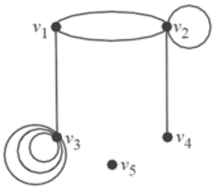
树的等价定义、生成树

- $G(V, E)$是树；
- G是连通的，且$|E| = |V| - 1$；
- G无圈，且$|E| = |V| - 1$；
- G的任意两点之间，存在唯一条路；
- G是连通的，删除任意一边后为非连通；
- G无圈，增加任意一边后，正好有一个圈。

第四章 图论基础

邻接矩阵

邻接矩阵



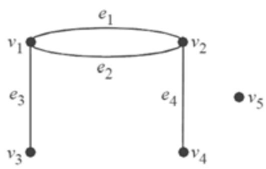
$$\mathbf{D} = \begin{array}{c} j \\ \begin{pmatrix} 0 & 2 & 1 & 0 & 0 \\ 2 & 1 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} i$$

$$\mathbf{D} = (d_{ij}).$$

$$d_{ij} = \text{number of edges between } v_i \text{ and } v_j.$$

$$V = \{v_1, \ldots, v_n\}$$

关联矩阵



$$\mathbf{A} = \begin{array}{c} \begin{array}{cccc} e_1 & e_2 & e_3 & e_4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array}$$

$$\mathbf{A} = (a_{ij})$$

$$a_{ij} = \begin{cases} 1 \text{ if } v_i \text{ is an end vertex of } e_j \\ 0 \text{ otherwise.} \end{cases}$$

Q：从关联矩阵得到邻接矩阵？

第四章 图论基础

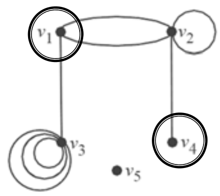
可达性

可达性矩阵

- 图 $G(V, E)$，邻接矩形为 $\boldsymbol{D}$
- 可达性矩阵：$\boldsymbol{R} = (r_{ij})$， $r_{ij} = (\boldsymbol{R})_{ij}$

$$r_{ij} = \begin{cases} 1 \textit{ if } G \textit{ has a } v_i\text{–}v_j \textit{ path} \\ 0 \textit{ otherwise.} \end{cases}$$

直接算法：邻接矩阵相乘



$$\begin{array}{c} \quad\quad j=1 \\ D = \begin{pmatrix} 0 & 2 & 1 & 0 & 0 \\ 2 & 1 & 0 & 1 & 0 \\ 1 & 0 & 3 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \begin{array}{l} \\ \\ \\ \\ i = 4 \\ \\ \end{array}$$

$D{\times}D = [\mathrm{sum}_k(d_{ik} \cdot d_{kj})]$ $O(N^2)$

$D{\times}D{\times}D = \ldots$ $O(N^2)$

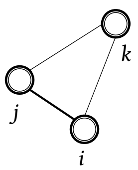
$D{\times}D{\times}D{\times}D = \ldots$ $O(N^2)$

$D{\times}D{\times}D{\times}D{\times}D = \ldots$ $O(N^2)$

$O(N^3)$

Warshall算法

- $\mathbf{E}_0$：$\mathbf{D}$中大于0的元素用1替代
- $\mathbf{R} = \mathbf{E}_n$

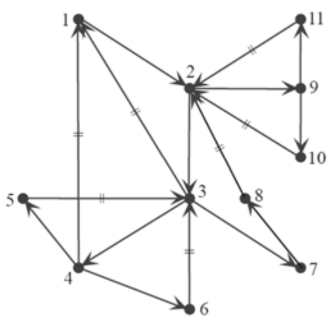


```
procedure Warshall
begin
   E := E_0
   for i := 1 to n do
      for j := 1 to n do
         if (E)_ji = 1 then for k := 1 to n do
            (E)_jk := max((E)_jk, (E)_ik)
         fi
      od
   od
end
```

[Prob]时间复杂度？

从某点起查找特定的点或边

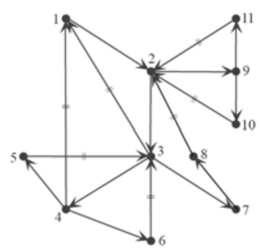
深度优先查找

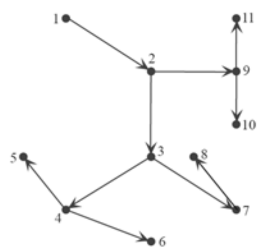


- r：起始点
- e：关联边
- v：下一点
- r = FAHTER(v)

1. 所有点 x 关联边查找完成时，返回父节点；
2. 否则选择x的关联边e，到下一点y
   a) 如果y已查过，则e为回退边；
   b) 如果y未查过，e为树边，以y替代x重复第1步

DFS树





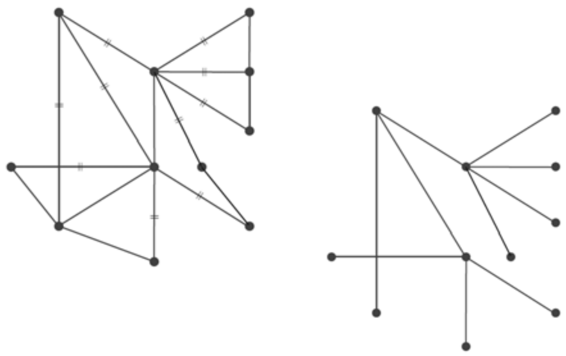
DFS形式化算法

1. Set TREE $\leftarrow \emptyset$, BACK $\leftarrow \emptyset$ and $i \leftarrow 1$. For every vertex x of G, set FATHER$(x) \leftarrow 0$ and $K(x) \leftarrow 0$.
2. Choose a vertex r for which $K(r) = 0$ (this condition is needed only for disconnected graphs, see step #6). Set DFN$(r) \leftarrow i$, $K(r) \leftarrow 1$ and $u \leftarrow r$.
3. If every edge incident to u has been examined, go to step #5. Otherwise, choose an edge $e = (u, v)$ that has not been examined.
4. We direct edge e from u to v and label it examined.
   4.1 If $K(v) = 0$, then set $i \leftarrow i + 1$, DFN$(v) \leftarrow i$, TREE $\leftarrow$ TREE $\cup \{e\}$, $K(v) \leftarrow 1$, FATHER$(v) \leftarrow u$ and $u \leftarrow v$. Go back to step #3.
   4.2 If $K(v) = 1$, then set BACK $\leftarrow$ BACK $\cup \{e\}$ and go back to step #3.
5. If FATHER$(u) \neq 0$, then set $u \leftarrow$ FATHER(u) and go back to step #3.
6. (Only for disconnected graphs so that we can jump from one component to another.) If there is a vertex r such that $K(r) = 0$, then set $i \leftarrow i + 1$ and go back to step #2.
7. Stop.

宽度/广度优先查找

1. In the beginning, no vertex is labeled. Set $i \leftarrow 0$.
2. Choose a (unlabeled) starting vertex r (root) and label it with i.
3. Search the set J of vertices that are not labeled and are adjacent to some vertex labeled with i.
4. If $J \neq \emptyset$, then set $i \leftarrow i + 1$. Label the vertices in J with i and go to step #3.
5. (Only for disconnected graphs so we can jump from one component to another.) If a vertex is unlabeled, then set $i \leftarrow 0$ and go to step #2.
6. Stop.

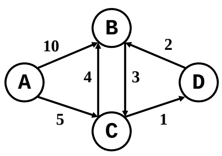
BFS树



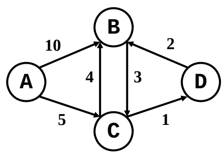
最轻/短路径

目标问题

- 边权重(weight)、路径权重
- 权重最小路，按字面为最轻路
  - 通信网中，把最短称为最短跳数，最轻称为最小成本
- 如果不存在特定两点间最轻路，也需明确



Dijkstra算法

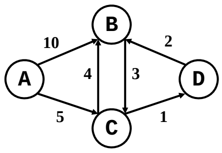



```
Dijkstra(G)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);      ← Q中d[u]最小者
      S = S U {u};
      for each v ∈ u->Adj[]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);      } Relaxation Step
Q->DecreaseKey()完成排序
```

计算过程,step0,step1

- d[A]=0, d[B]=d[C]=d[D]=∞,
- S = ∅; Q = {A,B,C,D}

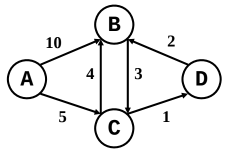
- u=A; S={A}; Q={B,C,D}
- u->Adj[]={B,C}; d[B]=10,d[C]=5



计算过程,step2

- u=A; S={A}; Q={B,C,D}
- u->Adj[]={B,C}; d[B]=10,d[C]=5

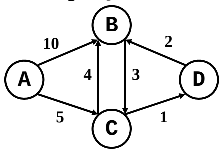
- u=C; S={A,C};Q={B,D};d[D]=6,d[B]=9



计算过程,step3,step4

- u=C; S={A,C};Q={B,D};d[D]=6,d[B]=9

- u=D;S={A,C,D};Q={B};d[B]=8
- d=B;S={A,B,C,D};Q=∅



同时得到最小生成树

[Prob]时间复杂度?

第四章 图论基础

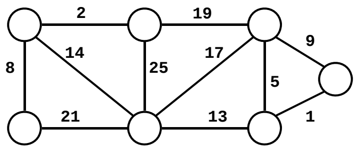
Kruskal算法

1. 将一条权最小的边加入子图T中，并保证不形成圈。
2. 如果当前弧加入后不形成圈，则加入这条弧，如果当前弧加入后会形成圈，则不加入这条弧，并考虑下一条弧。

Kruskal算法

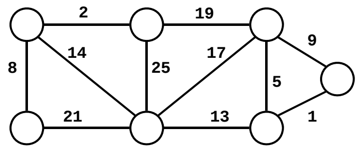
```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，计算用例



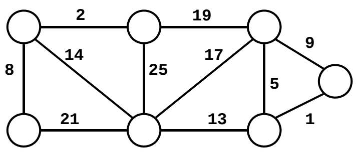

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，7个单节点子树



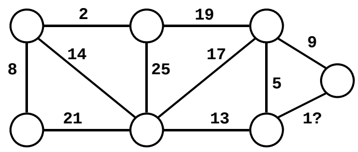

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，边集E排序



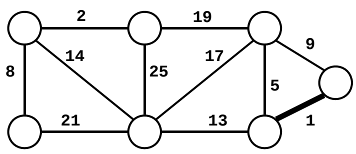

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，选取权重最小边



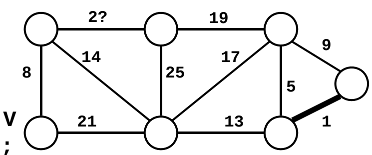

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，合并子树




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

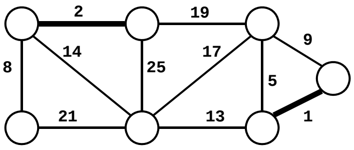
Kruskal算法，再选次小权重边




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

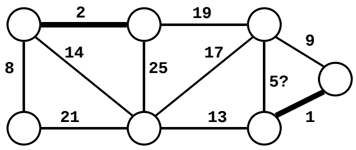
Kruskal算法，再合并子树

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```



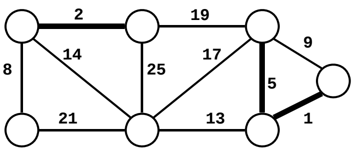
Kruskal算法，重复选边

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```



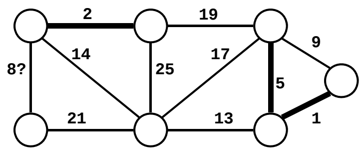
Kruskal算法，再合并子树

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```



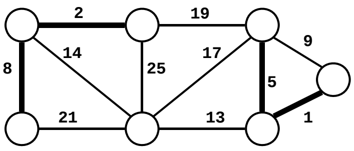
Kruskal算法，再选边

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```



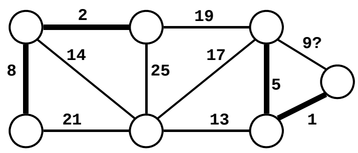
Kruskal算法，再次合并子树

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

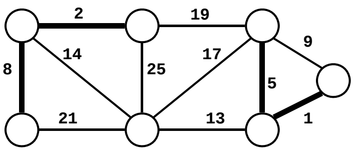


Kruskal算法，选到回路边

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

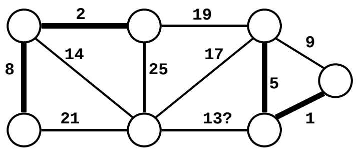


Kruskal算法，不需合并也不加集



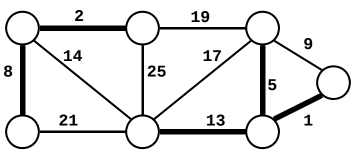

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，再选边



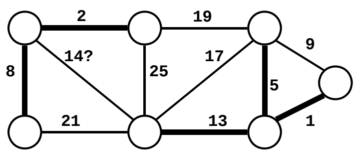

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，再合并子树



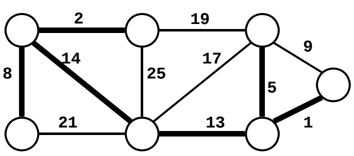

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，再选边



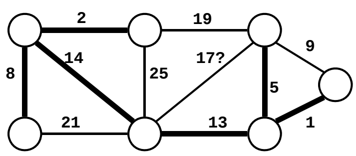

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，再合并



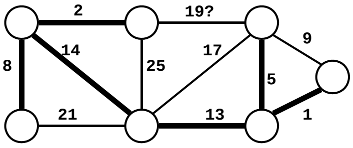

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，再选边、不合并



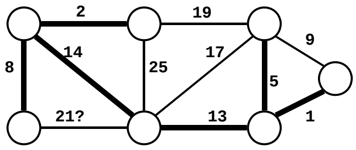

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，同样结果



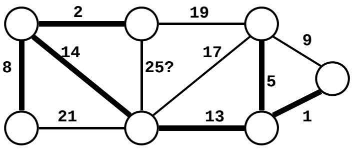

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，仍然不合并



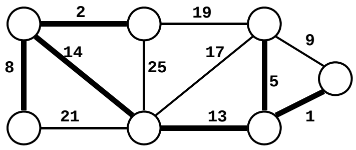

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，最后一条同样



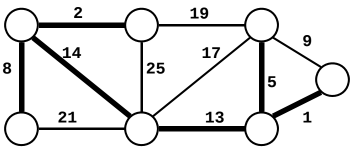

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法，得到MST




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal算法




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

[Prob]时间复杂度？

Prim算法

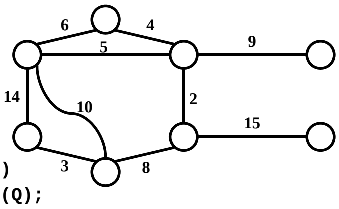
1. 不断扩展一棵子树 $T=(S, F)$, F 为E子集，直到S包括全部顶点，得到最小生成树T。
2. 每次增加一条边，使得这条边是由当前子树结点集S 及其补集S 所形成的边割集的最小边。

Prim算法

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

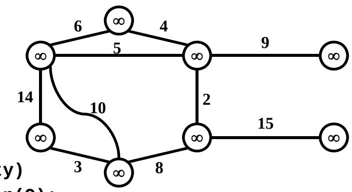
Prim算法，计算用例


```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

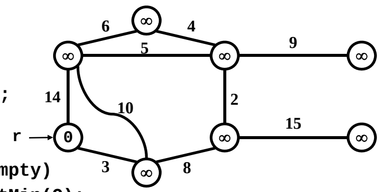
Prim算法,初始化


```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Prim算法，根选择

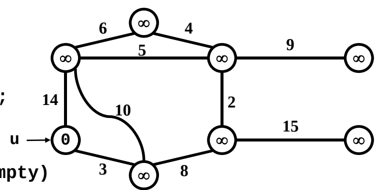

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Pick a start vertex r

Prim算法，集合Q更新

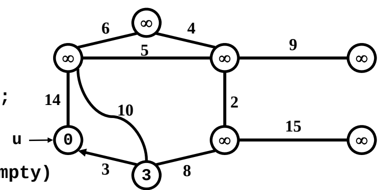

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Red vertices have been removed from Q

Prim算法，下一节点v

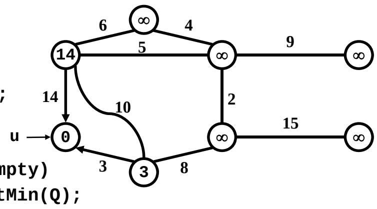

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Red arrows indicate parent pointers

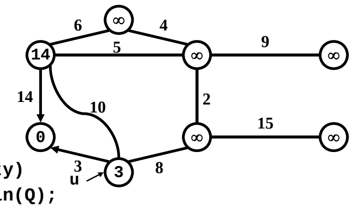
Prim算法，其他节点

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



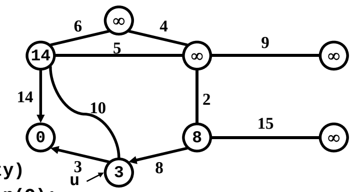
Prim算法，Q中Key最小节点

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



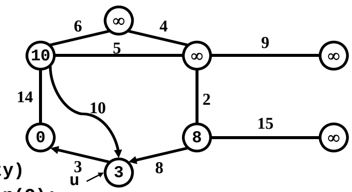
Prim算法，下一节点

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



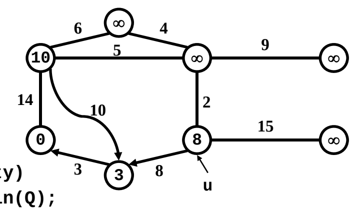
Prim算法，下一节点，更新

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



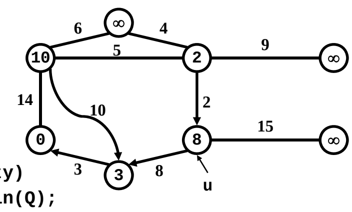
Prim算法，再选最小节点

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



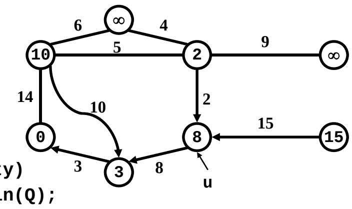
Prim算法，最小节点发生变化

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



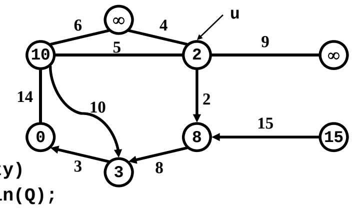
Prim算法

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



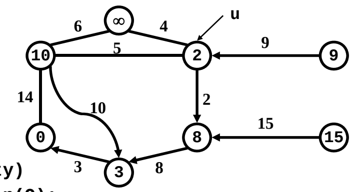
Prim算法，选最小节点

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



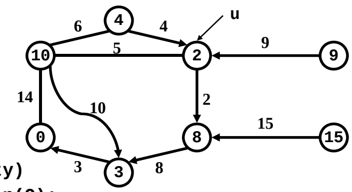
Prim算法

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



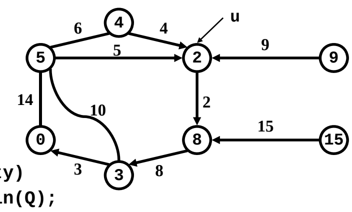
Prim算法，下一节点

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



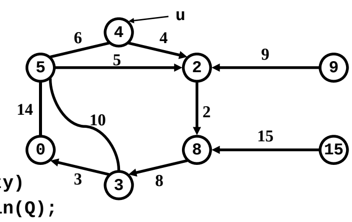
Prim算法，权值和路更新

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



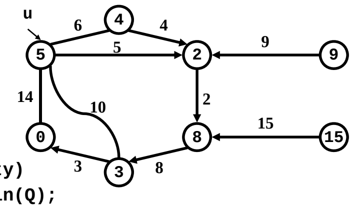
Prim算法，再选无更新

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



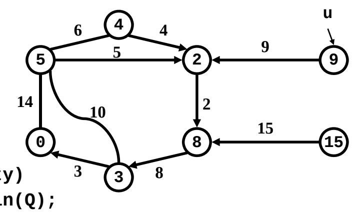
Prim算法，同样结果

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



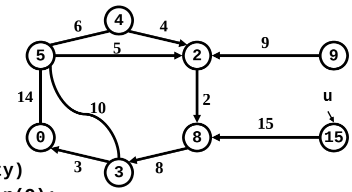
Prim算法，依然同样

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Prim算法，Q中最后一个

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Prim算法

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

What is the hidden code?

Prim算法,Q排序

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

Prim算法，复杂度推算

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

How often is ExtractMin() called?
How often is DecreaseKey() called?

Prim算法

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Depends on queue，Q
binary heap: O(E lg V)
Fibonacci heap: O(V lg V + E)

第四章 图论基础

4.1 证明图的平均节点度为2倍的图尺度与图阶数之商。

4.2 如何从图的关联矩阵求得邻接矩阵？

4.3 推算Warshall算法的时间复杂度。

4.4 推算Dijkstra算法的时间复杂度。