第17章 贪婪算法

- 最优化问题
- 贪婪算法(greedy method)思想
- **17.3.3 拓扑排序**
- **17.3.5 单源最短路径**
- **17.3.6 最小耗费生成树**

17.3.6 最小耗费生成树

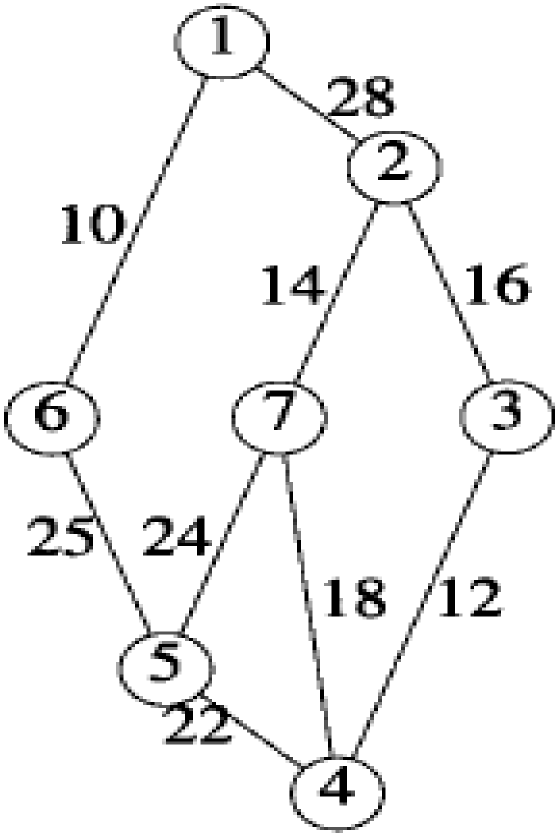
- 例[最小代价通讯网络]
  - 城市及城市之间所有可能的通信连接可被视作一个无向图，图的每条边都被赋予一个权值，权值表示建成由这条边所表示的通信连接所要付出的代价。
- 可行解：包含图中所有顶点（城市）的连通子图。设所有的权值都非负，则所有可能的可行解可表示成无向图的一组生成树。
- 最优解：具有最小代价的生成树。
- 限制条件：所有的边构成一个生成树。
- 优化函数：子集中所有边的权值之和。

- 最小耗费生成树(最小代价生成树/最小生成树)
- 具有*n*个顶点的**无向(连通)网络***G*的每个生成树刚好具有*n*-1条边。
- 最小耗费生成树问题是用某种方法**选择*n*-1条边**使它们形成*G*的**最小生成树**。
- 三种求解最小生成树的贪婪策略是:
  - **Kruskal (克鲁斯卡尔)算法**
  - **Prim(普里姆)算法**
  - Sollin算法

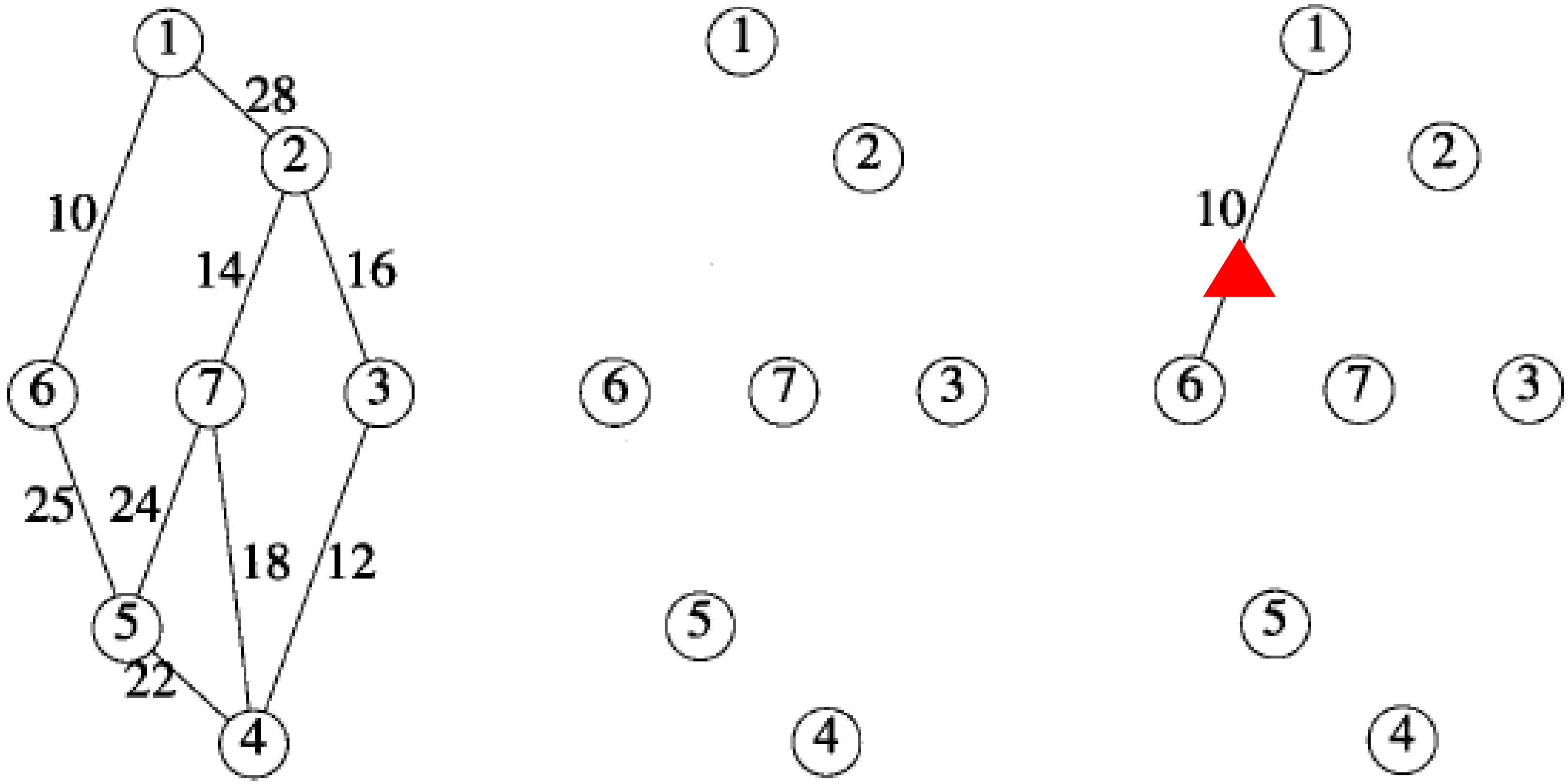
Kruskal算法

- Kruskal算法思想：
- 开始，初始化含有 n个顶点 0条边的森林.
- Kruskal算法所使用的贪婪准则是：从剩下的边中选择一条**不会产生环路**的**具有最小耗费**的**边**加入已选择的边的集合中。
  - Kruskal算法分*e*步(*e*是网络中边的数目)。
  - 按耗费递增的顺序来考虑这*e*条边，每次考虑一条边。
  - 当考虑某条边时，若将其加入到已选边的集合中会出现环路，则将其抛弃，否则，将它选入。

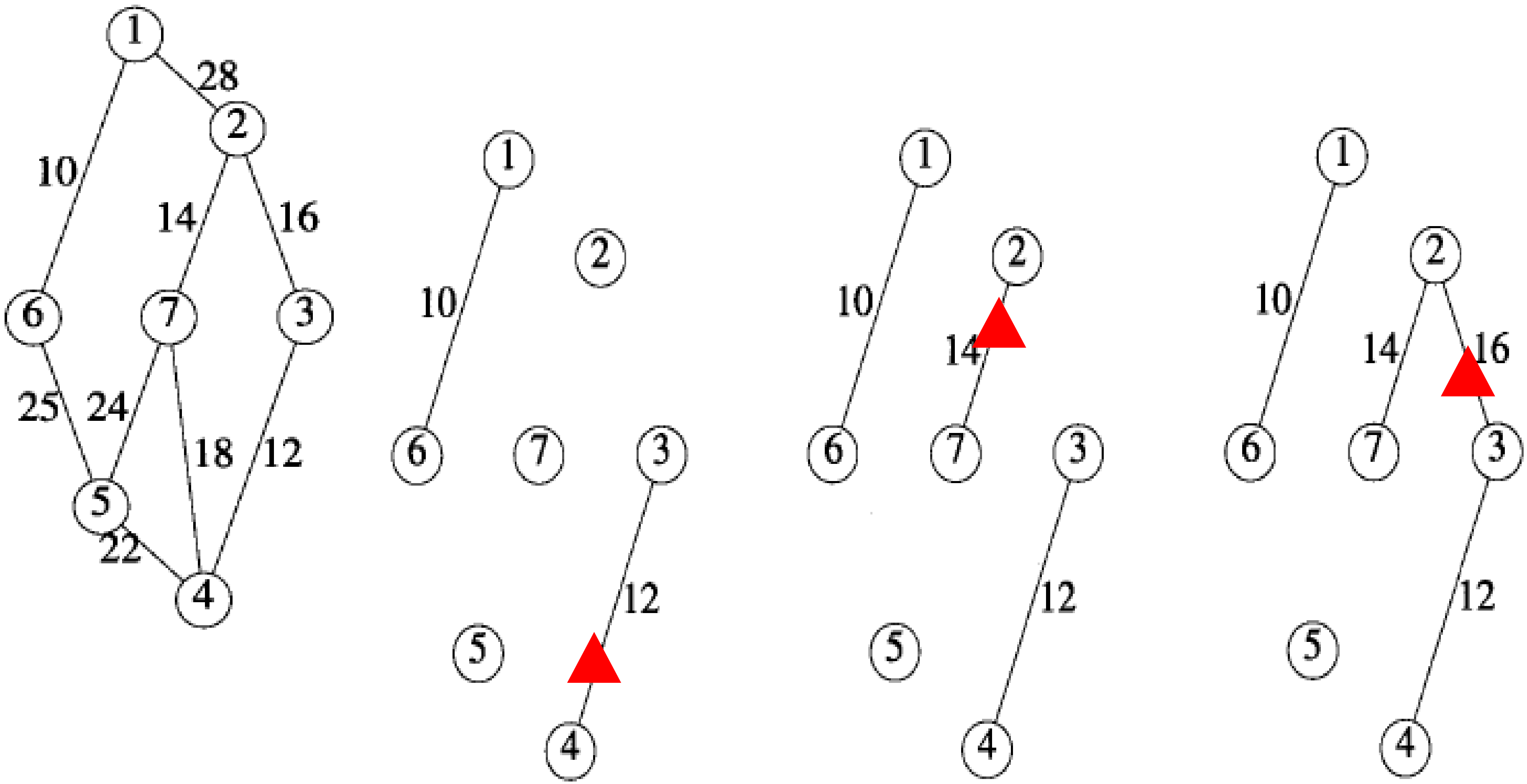
Kruskal算法示例



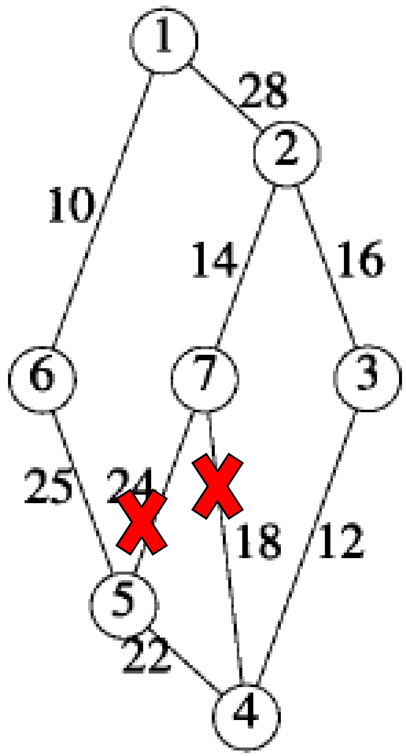
Kruskal算法示例

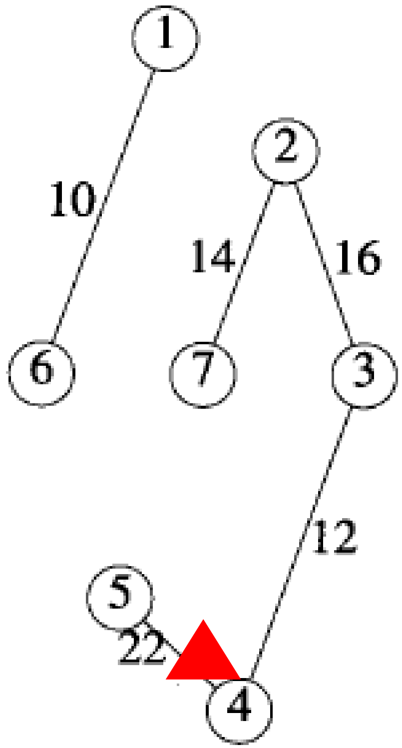


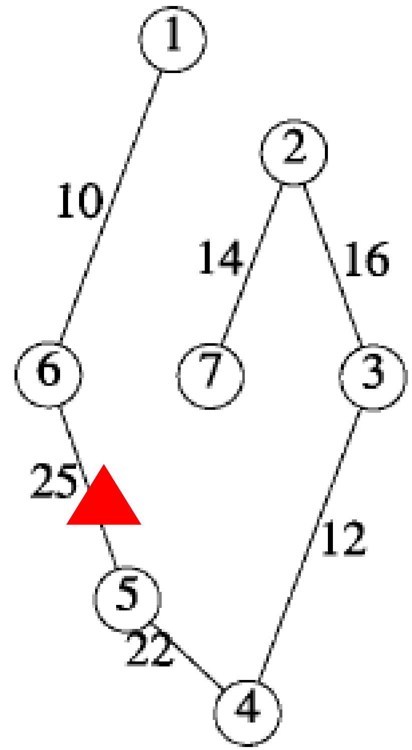
Kruskal算法示例



Kruskal算法示例







Kruskal算法的伪代码

- //在一个具有*n*个顶点的网络中找到一棵最小生成树
- 令*T*为所选边的集合，初始化T =∅
- 令E为网络中边的集合
- *While* (E≠∅ && |T| ≠ n-1){
  - 令(*u*,*v*)为*E*中代价最小的边
  - *E*=*E*-{(*u*,*v*) } //从*E*中删除边
  - *If* ((*u*,*v*)加入*T*中不会产生环路) 将(*u*,*v*)加入*T*
  - }
- *if* (|T| = = n-1) T是最小耗费生成树
- *Else* 网络不是互连的，不能找到生成树

•Kruskal算法的正确性证明 见P438

数据结构的选择及复杂性分析

- 边集 E：使用边的最小堆(小根堆)描述边集 E.
  - E 是否为空?
  - 选择和删除*E*中代价最小的边.
- 使用边的最小堆描述边集 E.
  - 初始化. O(e) .
  - 选取和删除代价最小的边： O(loge) .

数据结构的选择及复杂性分析

- 被选择的边的集合T:
  - T 中是否n -1 条边?
  - 将边 (u, v) 加入T是否产生环路?
  - 将边 (u, v) 加入T 中.
- 用数组spanningTreeEdges 来实现
  - 在数组的一端(右端)进行添加：O(1) .最多在*T*中加入*n*-1条边，因此对*T*的添加操作总时间为O(*n*)

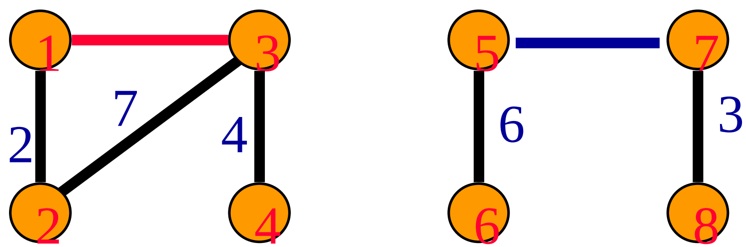
U					
V					
weight					

最小堆的元素及生成树数组 *t* 的数据类型

```
template <class T>
class EdgeNode {
  ▪ public:
      operator T() const {return weight;}
  ▪ private:
      T weight;//边上的权值
      int u, v;//边的端点
} ;
```

将边(u, v)加入T是否产生环路?

- 边的集合*T*与*G*中的顶点一起定义了一个由至多*n*个连通子图(树)构成的图。
- 当u 和 v处于同一子图时，将边 (u, v) 加入T会产生环路
- 当u 和 v处于不同的子图中时，将边 (u, v) 加入T则不会产生环路



■ 将边 (u, v) 加入T是否产生环路

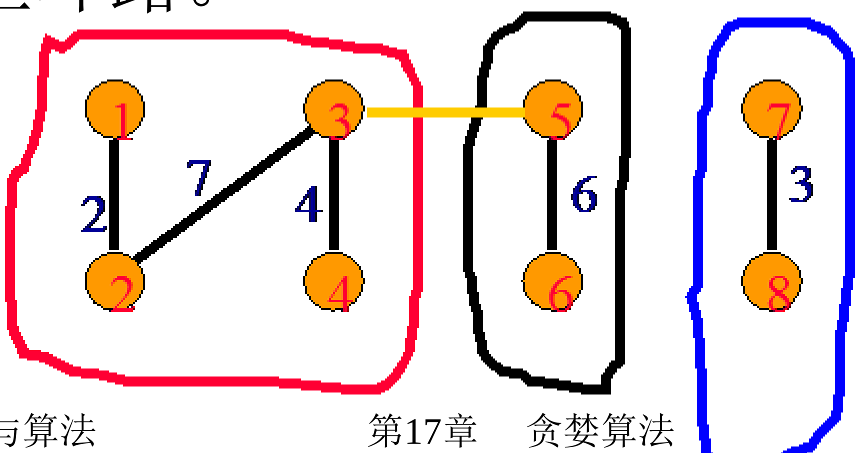
u 和 v是否处于同一子图

将边(u, v)加入T是否产生环路?

- 用子图中的顶点集合来描述每个连通子图(树)，这些顶点集合没有公共顶点。
- 两个顶点在同一个子图中 *当且仅当* 它们在同一个顶点集合中
- **两个顶点是否在同一个集合中：**可利用并查集中的find操作实现`s1 = find(u); s2 = find(v);`

将边(u, v)加入T是否产生环路?

- ***在T中加一条边，两个子图被合并：可***利用并查集中的unite操作实现
- 当s1 =s2 ;即顶点u 和顶点v在同一个顶点集合中，则说明顶点u 和顶点v处于同一子图中，将边(u, v) 加入T会产生环路;
- 当s1 !=s2 ;即顶点u 和顶点v在不同的顶点集合中，说明u 和 v处于不同的子图中，将边 (u, v) 加入T则不会产生环路。



复杂性分析

- 使用并查集(11.9.2).
  - 初始化.：O(n) .
  - find操作的次数最多为2*e*，Unite操作的次数最多为*n*-1(若网络是连通的，则刚好是*n*-1次)。
  - 比O(n+e)稍大一点。
- 使用边的最小堆，按耗费递增的顺序来考虑*e*条边：O(eloge).
- Kruskal算法的渐进复杂性：O(n+eloge).

Graph::Kruskal 1/3

```
bool kruskal(weightedEdge<T> *spanningTreeEdges)
{//使用Kruskal算法寻找最小代价(耗费)生成树
 //如果不连通则返回false;
 //如果连通,则在spanningTreeEdges[0:n-2]中
 //返回最小生成树
      if (directed() || !weighted())  throw…….;

      int n = numberOfVertices();
      int e = numberOfEdges();
```

Graph::kruskal 2/3

```
// 建立图中的边数组
 weightedEdge<T> *edge = new weightedEdge<T> [e + 1];
  int k = 0;        // edge[]的索引(游标)
  for (int i = 1; i <= n; i++)
      {//获得所有关联至顶点 i的边
         vertexIterator<T> *ii = iterator(i);
          int j;
          T w;
          while ((j = ii->next(w)) != 0)
             if (i < j)  // 加入到边数组
                edge[++k] = weightedEdge<int> (i, j, w);
        }
   // 数组edge[1:e]初始化为最小堆(小根堆)
  minHeap<weightedEdge<T> > heap(1);
  heap.initialize(edge, e);
  fastUnionFind uf(n); //建立n个元素的并查集对象
```

Graph::kruskal 3/3

```
//按照权值的递增顺序来抽取边,然后决定选入或舍弃
k = 0;  //作为spanningTreeEdges中的游标
while (e > 0  && k < n - 1)
    {//生成树未完成 并且 尚有剩余边
     weightedEdge<T> x =  heap.top();
      heap.pop();
      e--;
      int a = uf.find(x.vertex1());
      int b = uf.find(x.vertex2());
      if  (a != b)
          {// 选择边x
            spanningTreeEdges[k++] = x;
            uf.unite(a,b);
          }
      }
        return (k == n - 1);
      }
```

Prim 算法

- Prim (普里姆)算法思想：
  - 从具有一个**单一顶点**(可以是原图中任意一个顶点)的树*T*开始
  - 重复地加一条边和一个顶点.
    - ➢往*T*中加入一条代价最小的边(u,v) 使 $T \cup \{(u, v)\}$仍是一棵树.
    - ⇒对于边(u,v)， u 、 v 中正好有一个顶点位于*T*中.
  - 直到 T 中包含 n-1 条边为止.

Prim算法的伪代码

//假设网络中至少具有一个顶点

设 T 为所选择的边的集合，初始化 $T=\varnothing$

设 TV 为已在树中的顶点的集合，置 $TV=\{1\}$

令 E 为网络中边的集合

While ($E \neq \varnothing$) & & ($|T| \neq n$-1) {

  令(u, v)为最小代价边，其中$u \in TV, v \notin TV$

  If (没有这种边) break

  $E=E - \{(u,v)\}$ / /从E中删除此边

  在 T 中加入边(u, v)

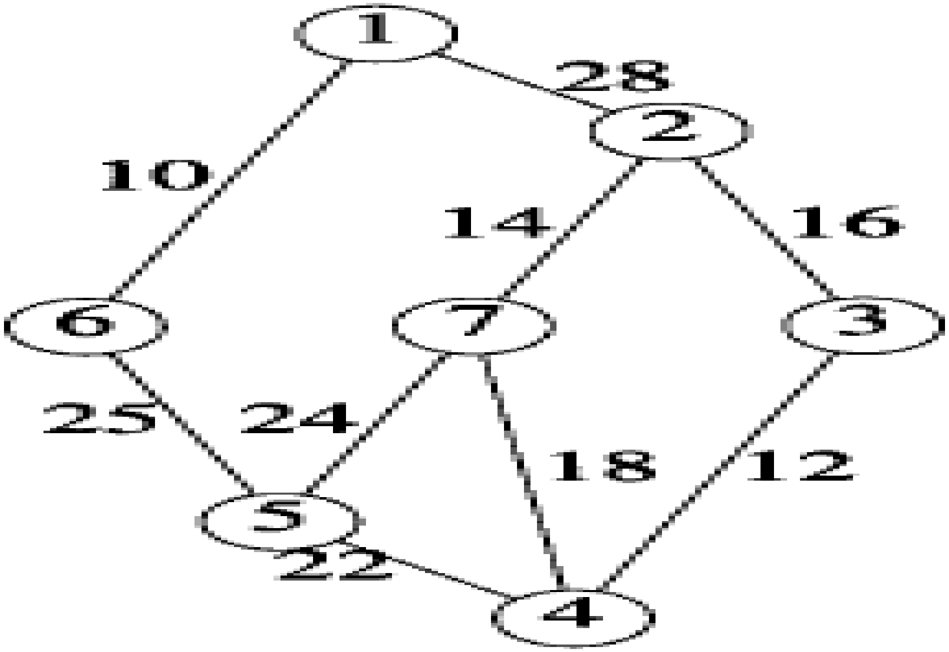
  }

if ($|T| = = n - 1$) T是一棵最小生成树

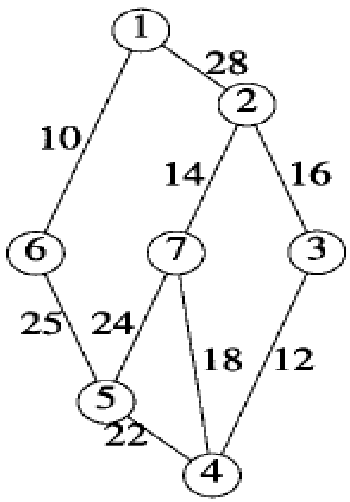
else 网络是不连通的，没有最小生成树

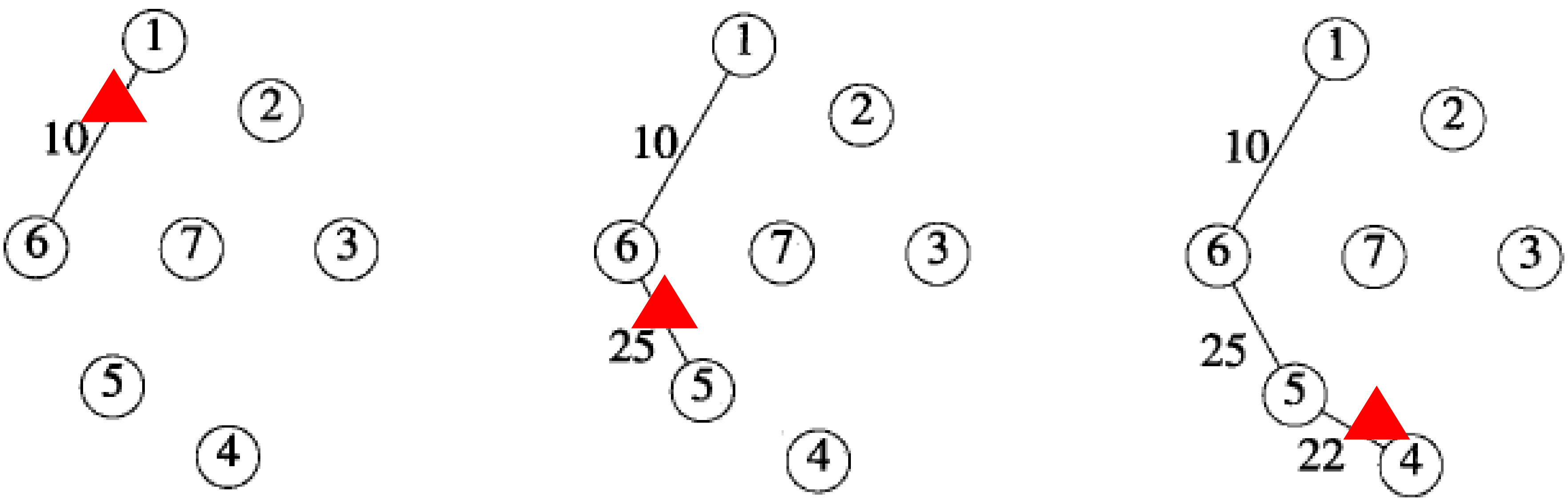
时间复杂性：$O(n^2)$

Prim算法示例

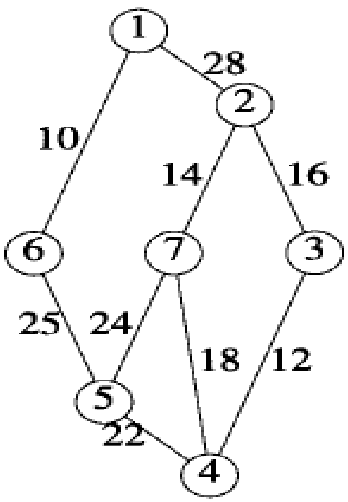


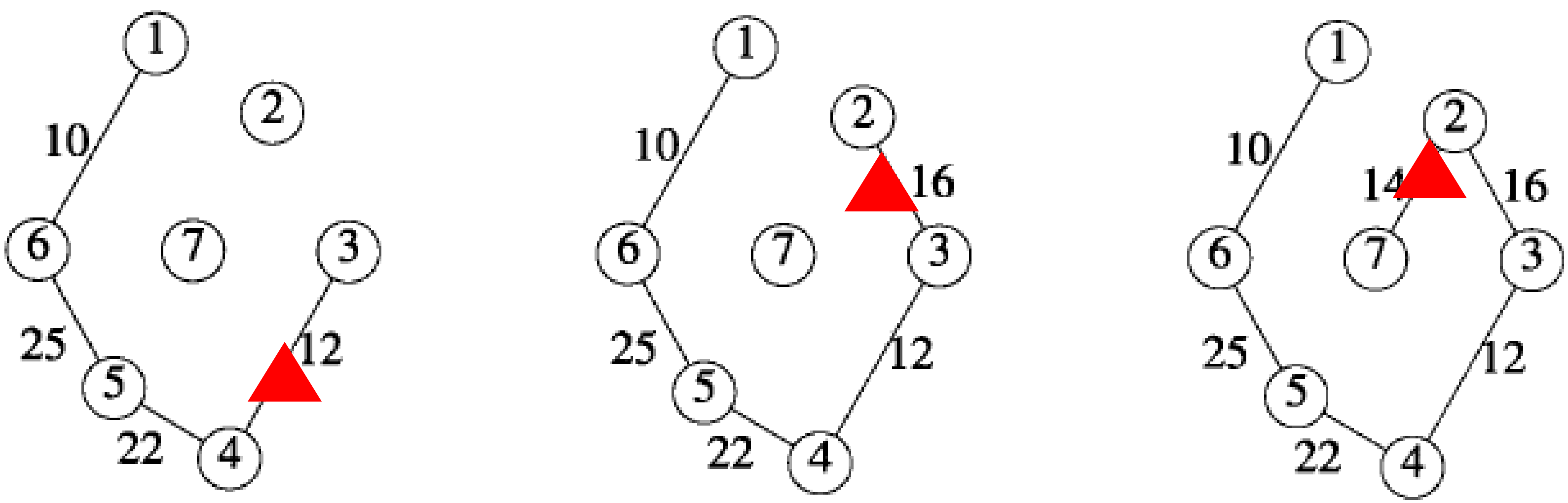
例





例

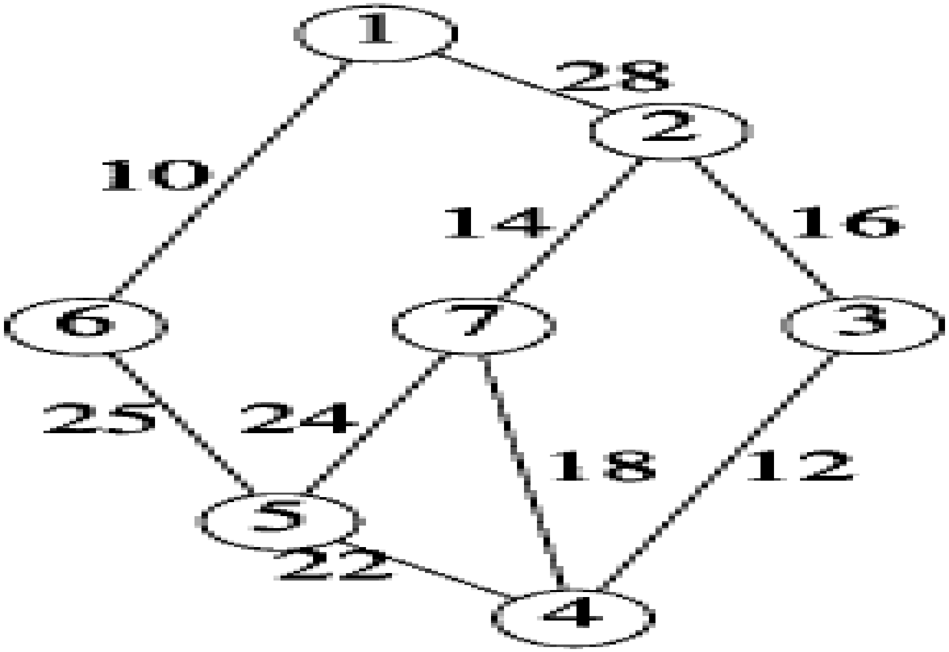




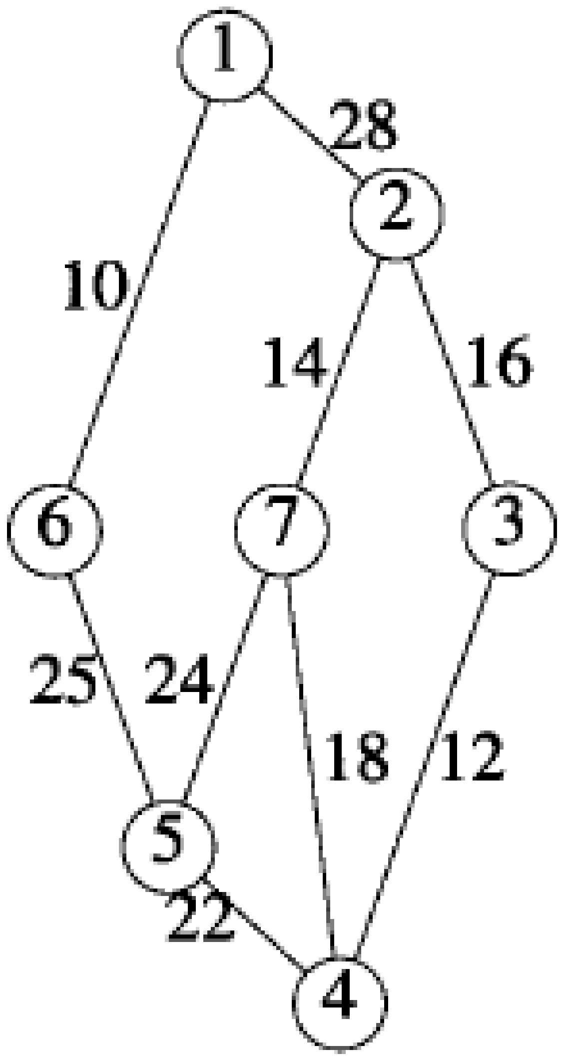
Sollin算法

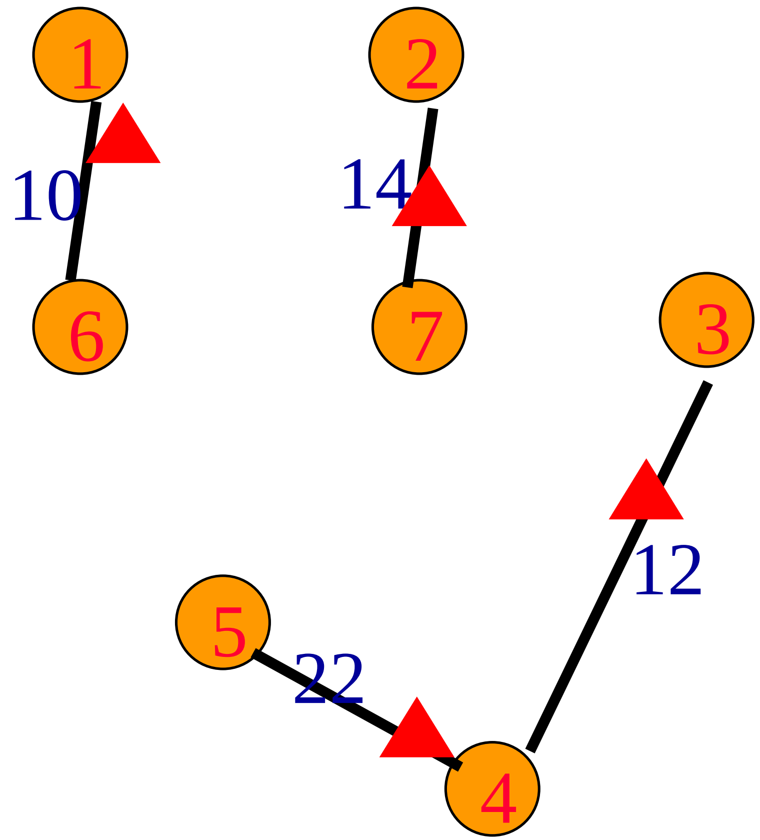
- Sollin算法思想.
- 从含有 n个顶点的森林开始.
- 每一步中为森林中的每棵树选择一条边，这条边刚好有一个顶点在树中且边的代价最小。将所选择的边加入要创建的生成树中。
  - 一个森林中的两棵树可选择同一条边。
  - 当有多条边具有相同的耗费时，两棵树可选择与它们相连的不同的边。
  - 丢弃重复的边和构成环路的边。
- 直到仅剩下一棵树或没有剩余的边可供选择时算法终止。

Sollin算法示例

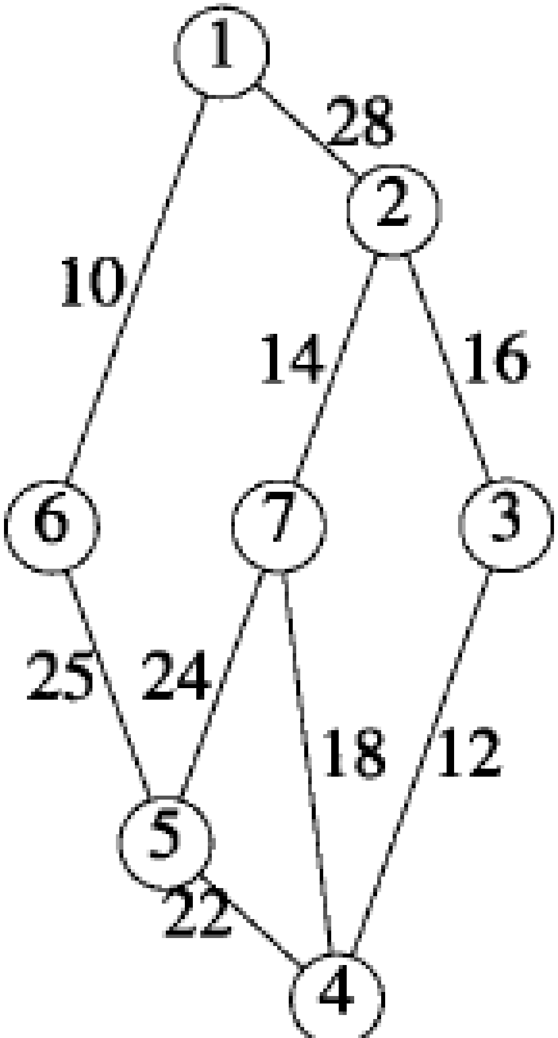


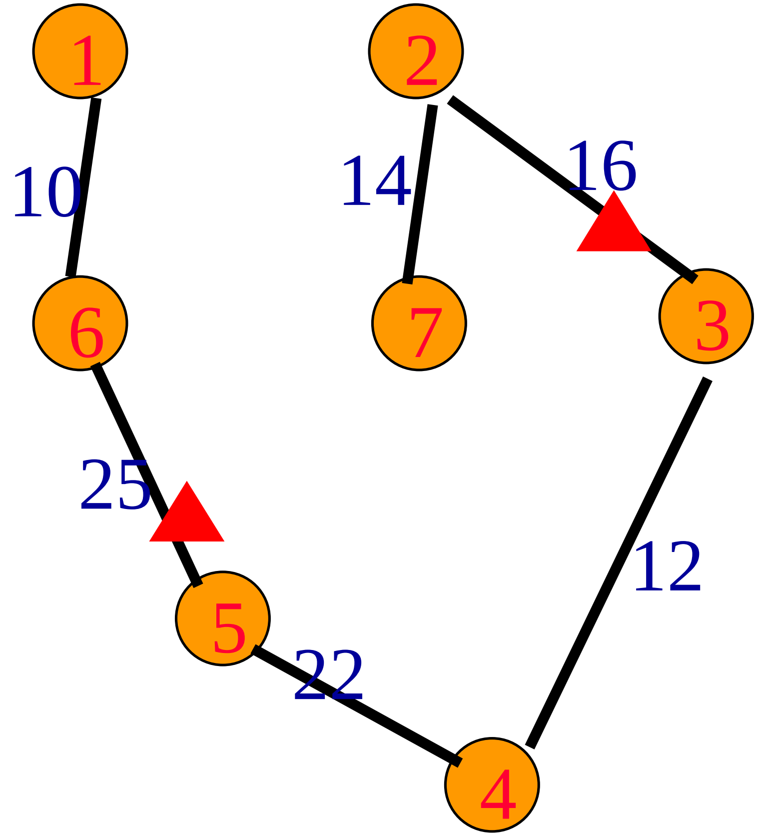
Sollin算法示例





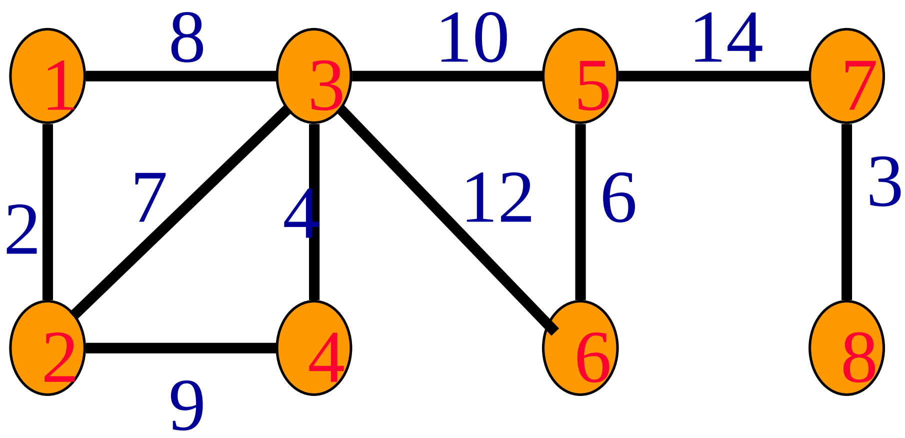
Sollin算法示例





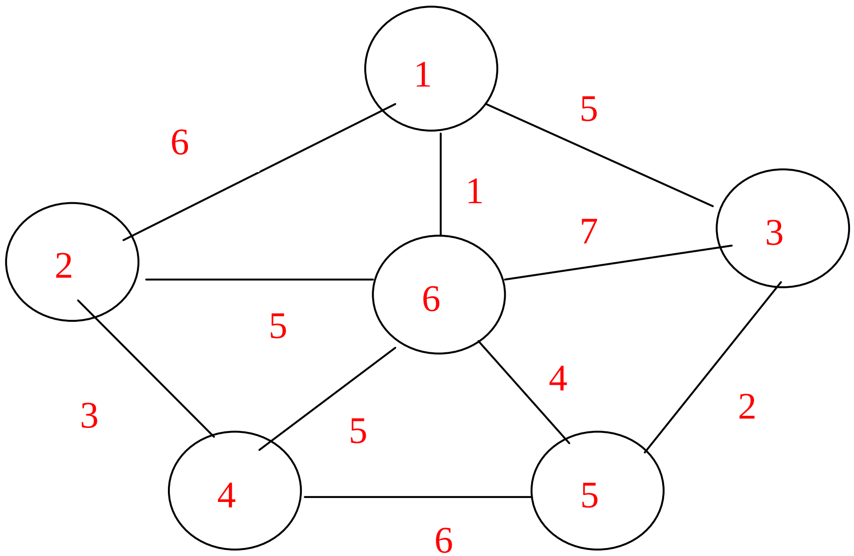
练习

- 1、描述图的普里姆算法和克鲁斯卡尔算法；分别用普里姆算法和克鲁斯卡尔算法求出下图的一棵最小代价生成树。给出构造最小生成树的过程.



练习

- 2、描述图的普里姆算法和克鲁斯卡尔算法；分别用普里姆算法和克鲁斯卡尔算法求出下图的一棵最小代价生成树。给出构造最小生成树的过程.



练习

- 3、有如下的网络邻接矩阵，画出该图；给出图的邻接链表表示；画出一棵最小生成树。
- ∞ 17 ∞ ∞ 20 22
- 17 ∞ 6 7 ∞ 12
- ∞ 6 ∞ 11 ∞ ∞
- ∞ 7 11 ∞ 19 15
- 20 ∞ ∞ 19 ∞ 34
- 22 12 ∞ 15 34 ∞
-