6. 图-2

- 图：基本概念和性质，基本操作
- 图的遍历：宽度优先，深度优先
- 图的表示和 Python 实现
- 生成树问题，DFS 生成树，BFS 生成树
- 最小生成树问题，Kruskal 算法，Prim 算法
- 最短路径问题，单源点最短路径和 Dijkstra 算法，所有顶点之间的最短路径和 Floyd 算法
- AOV 网，拓扑排序
- AOE 网，关键路径

最小生成树

- 网络 G（带权连通无向图）的每条边带有给定的权值，它的一棵生成树中各条边的权值之和称为该生成树的权
- 网络 G 可能有多棵不同生成树，其权值可能不同。其中权值最小的生成树称为 G 的最小生成树 (Minimum Spanning Tree，MST)

  显然，网络必有最小生成树，但最小生成树可能不唯一
- 最小生成树有许多实际应用：
  - 网络顶点看作城市，边的权看作连接城市的通讯线路的成本

    根据最小生成树建立的是城市间成本最低的通讯网
  - 可类似考虑成本最低的公路网；村庄间输电网络、有线电视网；城市输水管线、暖气管线、配送中心与线路规划；集成电路或印刷电路版的地线、供电线路，等等
- 表示这类应用的带权图中各结点到自身的权应取 0 值，到其他结点有边时有具体权值，无边时权值取无穷大 infinity。下面算法均如此假定

Kruskal 算法

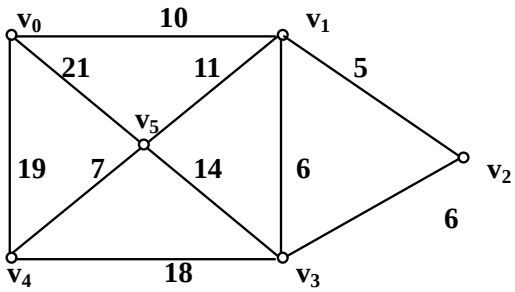
- Kruskal 算法是一种构造最小生成树的简单算法，其思想比较简单
- 设 G = (V，E) 是一个网络， Kruskal 算法构造最小生成树的过程是：
  - 初始时取只包含 G 中 n 个顶点而无边的孤立点子图 T = (V, {})，T 中每个顶点自成一个连通分量。下面考虑 T 的扩充
  - 将 E 中的边按权的递增顺序排序
  - 构造中每步按从小到大的顺序选择一条两端点位于 T 的两个不同连通分量的边 e，把 e 加入 T。这两个连通分量将由于该边的连接而变成一个连通分量（每次操作，T 减少一个连通分量）

    不断重复这个动作加入新边，直到 T 中所有顶点都包含一个连通分量里为止，这个连通分量就是 G 的一棵最小生成树
  - 如果这样做不能得到一个包含 G 的所有顶点的连通分量，则原图不连通，无最小生成树。算法做出的是 G 的最小连通树林
- 这一算法的正确性不难用归纳法证明（子图的生成树，最小，...）

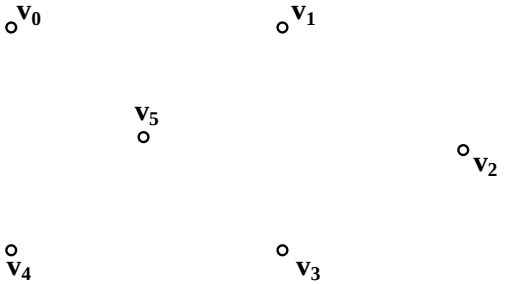
Kruskal 算法示例

- 用Kruskal方法构造右图的最小生成树
- E中的边按权递增顺序排列为：

  (v_1, v_2):5, (v_1, v_3):6, (v_2, v_3):6,
  (v_4, v_5):7, (v_0, v_1):10, (v_1, v_5):11,
  (v_3, v_5):14, (v_3, v_4):18, (v_0, v_4):19,
  (v_0, v_5):21

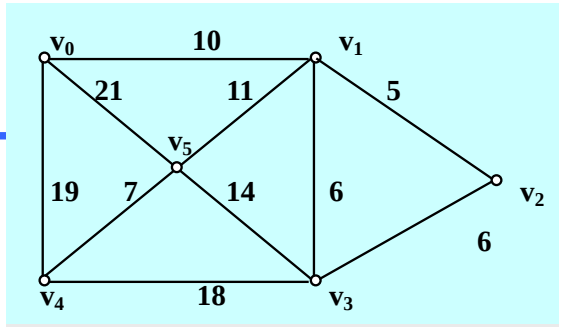


0，初始，T为只包含6个顶点的图

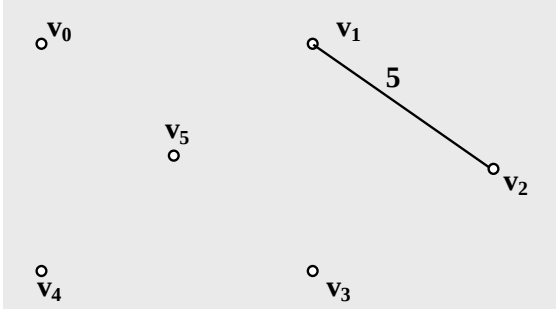


Kruskal 算法示例

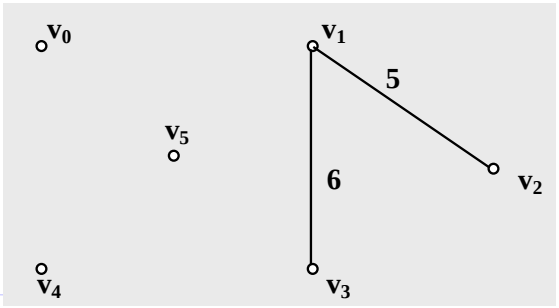
(v_1, v_2):5, (v_1, v_3):6, (v_2, v_3):6,
(v_4, v_5):7, (v_0, v_1):10, (v_1, v_5):11,
(v_3, v_5):14, (v_3, v_4):18, (v_0, v_4):19,
(v_0, v_5):21



1，加入最小边 (v_1, v_2) 可以减少一个连通分量，图变成：

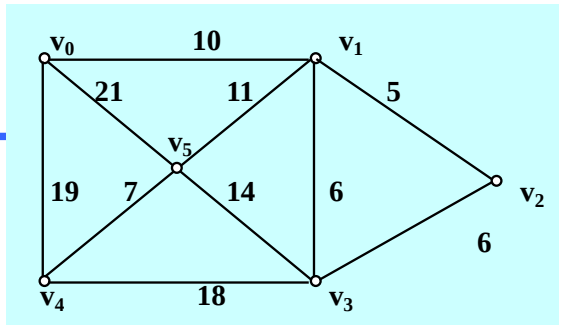


2，加入剩下的最小边 (v_1, v_3) 又可以减少一个连通分量，得到右图

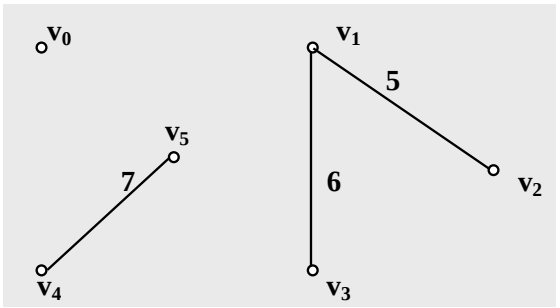


Kruskal 算法示例

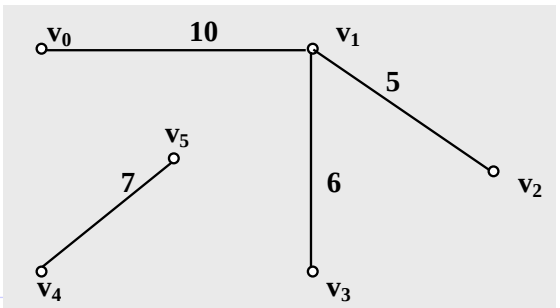
(v_2, v_3):6, (v_4, v_5):7, (v_0, v_1):10,
(v_1, v_5):11, (v_3, v_5):14, (v_3, v_4):18,
(v_0, v_4):19, (v_0, v_5):21



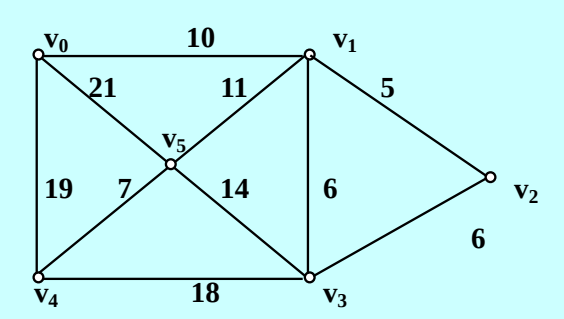
3，加入最小边 (v_2, v_3) 不能减少连通分量（ v_2, v_3 属同一连通分量），将它删除。加入剩下最小边 (v_4, v_5) 减少连通分量，得到右图：



4，加入剩下的边中权最小的 (v_0, v_1) 可以减少一个连通分量



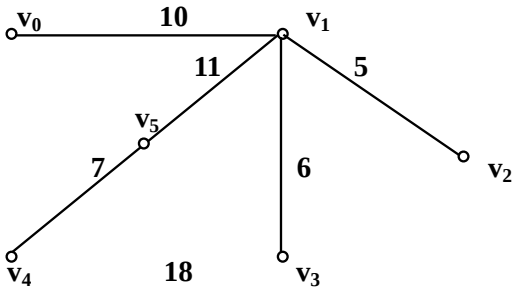
Kruskal 算法示例



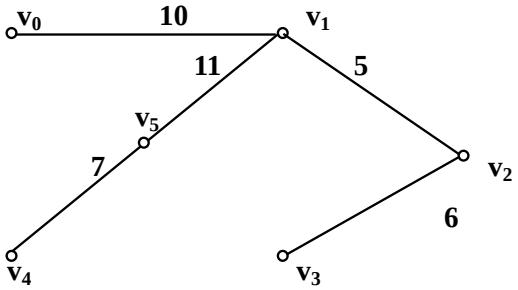
(v_1, v_5):11, (v_3, v_5):14, (v_3, v_4):18,

(v_0, v_4):19, (v_0, v_5):21

3，加入最小边 (v_1, v_5) 减少一个连通分量，得到右图。构造完成



图的最小生成树不唯一。例如图 G 中边 (v_1, v_3) 和 (v_2, v_3) 权都为 6，如果构造时选择 (v_2, v_3)，将得到另一最小生成树



Kruskal 算法

- 算法的抽象描述：

  T = (V, {})
  while T 中所含边数 **< n-1:**
  　从 **E** 中选取当前最小边 **(u,v)**, 将它从 **E** 中删去
  　if (u,v) 加入 **T** 中后不产生回路：将边 **(u,v)** 加入 **T** 中

- 算法的一个关键问题是判断两个顶点在当时的 **T** 里属于不同连通分量。可以通过检查两顶点间是否有路径完成判断，但这种做法太费时间
- 一种有效方法是为每个连通分量确定一个“代表元”，在算法过程中对每个顶点记录（和维护）其连通分量归属（记录和更新代表元）
  - 代表元相同的顶点相互连通（常量时间可判断）
  - 加入边减少了连通分量，需要选一个顶点，让合并的两个连通分量里的顶点都以它为代表元，为此要更新一些结点的代表元
- 表 **reps** 记录代表元关系，初始时各顶点以自己为代表元（**reps[i] = i**），算法中更新（**reps[v]** 总是 **v** 的代表元的下标）。表 **mst** 累积结果

Python 实现

```
def Kruskal(graph):
    vnum = graph.vertex_num()
    reps = [i for i in range(vnum)]
    mst, edges = [], []
    for vi in range(vnum): # 把所有的边加入表 edges
        for v, w in graph.out_edges(vi):
            edges.append((w, (vi, v)))
    edges.sort()                # 边按权值排序，O(n log n) time
    for w, e in edges:
        if reps[e[0]] != reps[e[1]]: # 两端点属于不同连通分量
            mst.append((e, w))       # 记录这条边
            if len(mst) == vnum-1: break # 已经有 |V|-1 条边，构造完成
            rep, orep = reps[e[0]], reps[e[1]]
            for i in range(vnum):    # 合并连通分量，统一代表元
                if reps[i] == orep: reps[i] = rep
    return mst
```


Kruskal 算法的复杂性和其他

- 算法中用了一个保存边的表，排序，选择，合并连通分量
- 时间复杂性（**|E|** 和 **|V|** 是边集合和顶点集合的大小）：
  - 建立边表 **edges** 用 **O(|E|)**，排序 **O(|E| log |E|)**（Python 函数）
  - 循环：
    - 整个循环不超过 **O(|E|)** 次，**O(|E|)** 时间
    - 记录元素 **n-1** 次，修改代表元 **|V|** 次每次 **O(|V|)**，计 $O(|V|^2)$
    - 时间复杂性 $O(max(|E|, |V|^2)) = O(|V|^2)$
  - 总的时间复杂性是 $O(max(|E| log|E|,|V|^2))$
- 空间复杂性主要是 **edges**，**reps** 和 **mst** 的开销，**O(max(|E|, |V|)**

  对于连通图，总有 **O(|E|) >= O(|V|)**，所以是 **O(|E|)**
- 注意：如果图用邻接矩阵表示，建立边表就需要 $O(|V|^2)$ 时间

Kruskal 算法的复杂性和其他

- 张乃孝老师的教科书上给出了 Kruskal 算法的另一个实现

  该实现的时间复杂性是 $O(|V|^3)$，空间复杂性是 $O(|V|^2)$

- 更好的实现方法采用了其他技术和辅助结构，可以使算法的时间复杂性降到 $O(|E| \log |V|)$（主要改进是集合合并的效率）

  注意：$O(\log |E|) = O(\log |V|)$，一般而言 $|E| < |V|^2$

- Kruskal 算法是一个抽象算法（或称为算法模式）
  - 实现时可以采用不同的具体辅助数据结构，不同的实现方法
  - 不同实现可能具有不同的时间和空间复杂性
- 下面考虑解决同一问题的另一种算法，称为 Prim 算法，其设计出发点与 Kruskal 算法完全不同，基于最小生成树的一种重要性质
  - 基本想法是从一个顶点出发逐步扩充包含该顶点的部分生成树 T
  - 开始时 T 只包含这个顶点且没有边，最终做出了一棵最小生成树

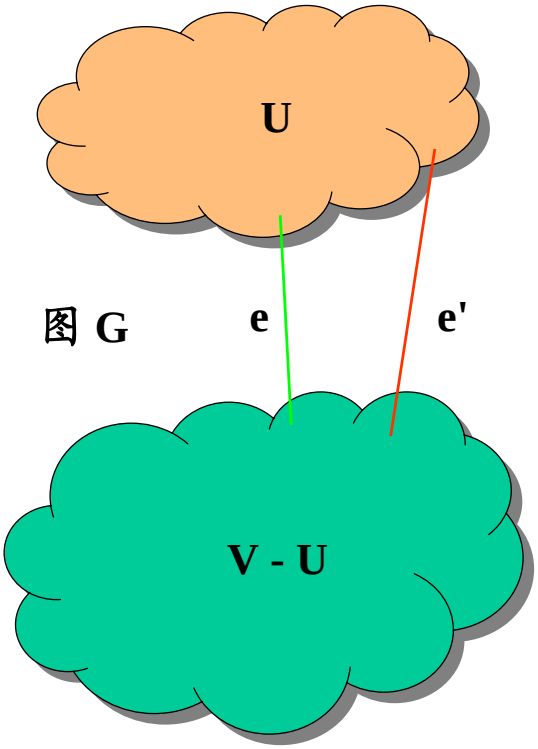
最小生成树：MST 性质

- 最小生成树有一个重要的 MST 性质：

  设 G = (V，E) 是网络，U 是 V 的任一真子集，e = (u,v) ∈E 且 u∈U，v∈V - U，而且 e 在 G 中所有一个端点在 U 另一端点在 V - U 的边中权值最小，那么：G 必有一棵包括边 e 的最小生成树

- 证明：

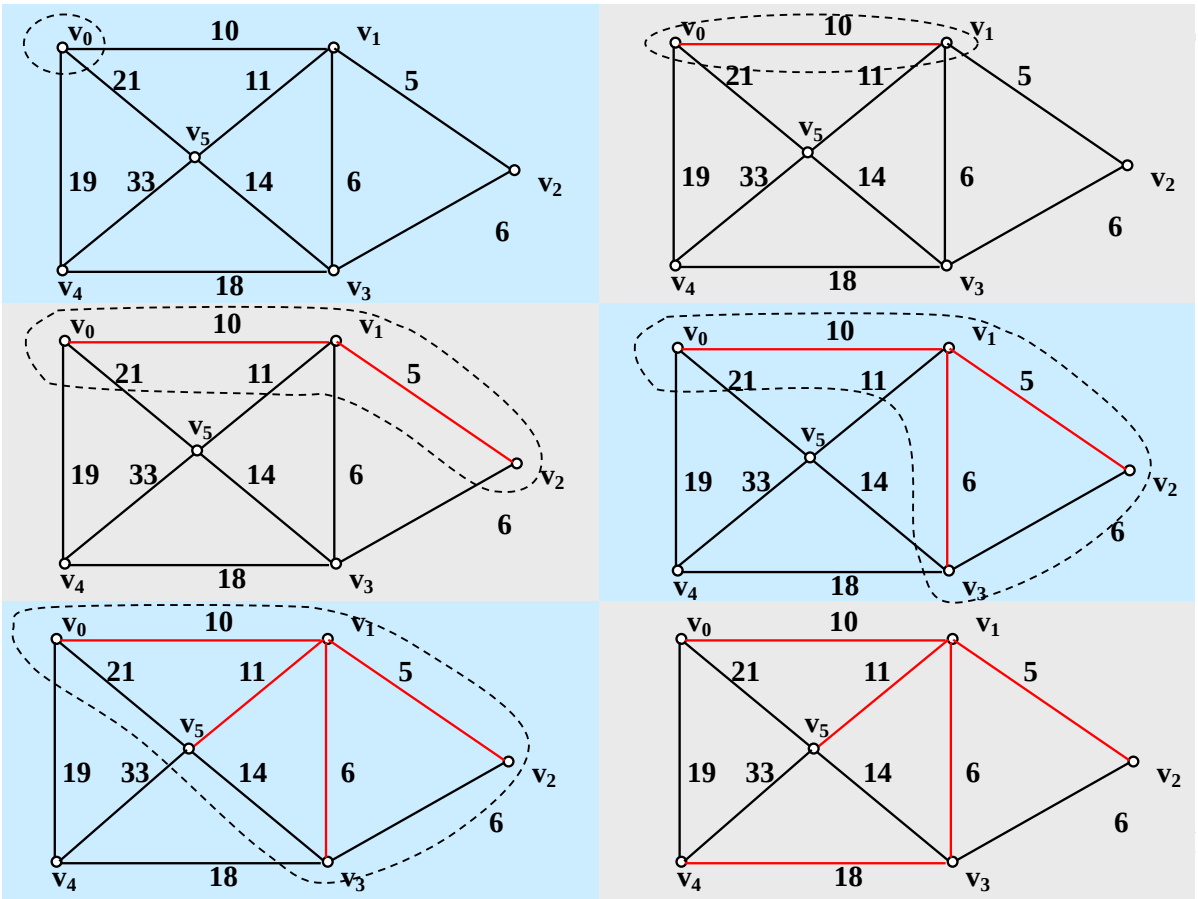
  取 G 的一棵最小生成树T（必定存在），若 e 属于 T 得证。否则将 e 加入 T 得到 T'，由于 T 连通 T' 中必定有环。设环中另一条一端在 U 另一端在 V-U 的边为 e'。去掉 e' 得到另一生成树。由于 e 的定义，新生成树的权不大于 T，即为所需



最小生成树：Prim 算法

- 基本思想：从一个顶点出发，利用 MST 性质选最短连接边扩充已连接的顶点集，直至该集合包含了所有顶点；或最终确定不是连接图
- 算法细节：
  - 从图 G 的顶点集 V 中任取一顶点（例如取顶点v_0）放入集合 U 中，这时 U = {v_0}，边集合 TE = { }，T = (U, TE) 是一棵树
  - 检查所有一个顶点在集合 U 里另一顶点在集合 V – U 的边，找出其中权最小的边 e = (u, v)（u∈U，v∈V-U），将顶点 v 加入集合 U，并将 e 加入 TE。(U, TE) 仍是一棵树
  - 重复上面步骤直至 U = V（树中包含所有顶点）。这时 TE 有 n-1 条边，T = (U, TE) 就是 G 的一棵最小生成树
- 这个算法直接利用 MST 性质构造最小生成树
  - 算法正确性：这一算法最后得到的必然是 G 的最小生成树
  - 请考虑如何证明这个算法的正确性

Prim 算法的过程示例



Prim 算法实现的思想

- 生成树用类似构造 DFS 生成树的数据表示，用 vnum 个元素（vnum 为顶点数）的表 mst 保存最小生成树的边，mst[1] = ((1, 2), 10) 表示顶点 1 到 2 的边在 mst 且权值为 10，mst[i] = None 表示 i 不属于 U
- 用一个优先队列 cands 记录候选的最短边，元素形式为 (w, i, j)，表示从顶点 i 到 j 的边为候选，其权值为 w
- 算法过程：
  - 初始把 (0, 0, 0) 放入优先队列，表示从顶点 0 到自身的长 0 的边
  - 循环的第一次迭代将把顶点 0 记入最小生成树顶点集 U，方法是设 mst[0] = (0, 0)。还会把顶点 0 到其余顶点的边按权值存入优先队列 cands，表示待考察的候选边集合
  - 执行中反复选择 cands 里记录的最短边 (u, v) 。如确定它是连接 U 中顶点与 V-U 顶点的边（发现 v 是 V-U 的顶点），就把这条边及其权记入 mst[v]，并把 v 的出边存入 cands；否则直接丢掉
  - 结束时 mst 中是最小生成树的 n 条边（包括 (0, 0) 边以方便实现）

Prim 算法的实现

```
def Prim(graph):
    vnum = graph.vertex_num()
    mst = [None]*vnum
    cands = PrioQueue([(0, 0, 0)]) # 记录候选边，形式为 (w, vi, vj)
    count = 0
    while count < vnum and not cands.is_empty():
        w, v, vmin = cands.dequeue() # 取当时的最短边
        if mst[vmin]: continue          # 邻接顶点 vmin 已在 mst，继续
        mst[vmin] = ((vmin, v), w)      # 记录新的 MST 边和顶点
        count += 1
        for edge in graph.out_edges(vmin): # 考察 vmin 可达的顶点和边
            if not mst[v]:                 # 如果 v 还不在 mst，记录它
                cands.enqueue((w, vmin, v))
    return mst
```

Prim 算法的复杂性分析

- 时间复杂性：初始化部分是 O(|V|)，算法时间主要用在选择最小生成树的边。循环次数与考察和加入优先队列的元素个数有关，也与顶点个数 |V| 有关。考虑到每条边至多进入弹出队列一次，这个角度的时间开销是 O(|E| log |E|)；构造 O(|V|) 条边的时间开销不低于 O(|V|)。一般而言前者大于后者，所以时间复杂性为 O(|E| log |E|)
- 空间复杂度：算法里用了一个表 mst 和一个优先队列，大小是 O(|V|) 和 O(|E|)。对连通图有 |E| > |V|，由此空间复杂度是 O(|E|)
- Prim 算法也是一种抽象算法。张老师的教材里给了另一个实现，其时间复杂度是 $O(|V|^2)$。请注意 O(log |E|) = O(log |V|)，也可以说上面 Prim 算法的时间复杂度是 O(|E| log |V|)。在边比较稀疏的图，也就是说，如果 |E| = O(|V|)，这里的算法效率更高
- 注意：如果图用邻接矩阵表示，提取一个顶点的邻接边就是 O(n) 操作，整个算法的时间开销将为 $O(\max(|V|^2, |E| \log |E|))$

  可见，数据结构采用不同的实现方法，同样算法的效率也不一样

Prim 算法的改进

- 前面 Prim 算法的缺点是可能把没价值的边存入队列 cands

  空间复杂性达到 O(|E|)（通常大于 O(|V|)）

  在实际中加入队列的边可能不同，可以做些试验研究有关的情况
- 如果算法里能只记录连接 U 和 V-U 的边，那么在整个计算中，需要保存的边不超过 |V| 条，就可以把空间开销降到 O(|V|) 。为保证效率，要求算法中使用的数据结构能有效支持下面操作
  - 获得这些边中的最短边。显然，采用堆结构，复杂性为 O(log|V|)
  - 发现了到 V-U 中顶点的更短边时高效更新。要求在 O(log|V|) 时间内从结点找到堆里的元素，修改原有的权值并恢复堆结构
- 也就是说，需要一种类似优先队列的结构，支持（以元素个数为 n）：
  - O(n) 的建堆，O(log n) 的 getmin 操作（堆支持这些）
  - 从某种 index（这里是顶点标号）出发的 O(1) 的 weight(ind)，从 id 和新权值出发的 O(log n) 的减权值操作 dec_weight(ind, w)

Prim 算法的改进

- 称这种结构为“可减权堆”，设定义的类是 **DecPrioHeap**，支持：
  - **DecPrioHeap(list)**，其中 **list** 的项形式为 **(w, index, other)**，这个类的对象 **decheap** 是以 **list** 的项为元素以 **w** 为权的小顶堆
  - **decheap.getmin()** 弹出堆中最小元并恢复堆，**O(log n)** 复杂性
  - **decheap.weight(ind)** 得到 **ind** 为 **index** 的元素权值，**O(1)**
  - **decheap.dec_weight(ind, w, other)** 在 **w** 小于 **decheap** 里具有 **ind** 的元素的权值时，将该元素改为 **(w, ind, other)** 并恢复堆的结构，复杂性为 **O(log n)**
  - **decheap.is_empty()** 在 **decheap** 为空时返回 **True**

  有了这个类，改造 **Prim** 算法的工作就可以完成
- 这种结构是可以实现的，但有点复杂，留作练习（有点困难）

  课程主页里有个链接，是最近出的一本 **Python** 数据结构教材（美国）的相关材料，作者没做出这个结构，给了一个 **O(n)** 操作

Prim 算法的改造

改造后的算法：

```
def Prim(graph):
    vnum = graph.vertex_num()
    wv_seq = [[graph.get_edge(0, v), v, 0] for v in range(vnum)]
    connects = DecPrioHeap(wv_seq)  # 最近连接的顶点堆，|V| 个元素
    mst = [None]*vnum
    while not connects.is_empty():
        w, mv, u = connects.getmin() # 取得最近顶点和连接边
        if w == infinity: break # 最近顶点已不连通，说明该图没有生成树
        mst[mv] = ((u, mv), w)                  # 这就是新的 MST 边
        for v, w in graph.out_edges(mv): # 检查 mv 的邻接边
            if not mst[v] and w < connects.weight(v): 如果更短就修改
                connects.dec_weight(v, w, mv)
    return mst
```

空间复杂性 **O(|V|)**，时间复杂性是 **O(max(|V| log |V|, |E| log |V|))**

最小生成树问题的复杂性

- 最小生成树是一个非常重要的问题，人们对它做了很多进一步研究，取得的许多成果（参看相关 **wiki** 页）
  - 有一个证明了复杂度为 **O(|E|)** 的随机算法（概率意义的复杂度）
  - 有一个证明了复杂度为 **O(|E| α(|E|, |V|))** 的已知最优算法，其中的 α 是 **Ackermman(n, n)** 的逆函数（几乎是常数，**2002**年）
  - 有并行算法方面的工作
  - **...**

$$\text{Achermman}(4,4) = 2^{2^{10^{19729}}}$$

- 最小生成树（**MST**）研究还没结束。研究已经确定该问题的时间复杂度下界为 **O(|E|)**，但还没找到这样的算法，也没找到更高的下界
- 可以自己想想上面两个算法的实现方法，或者自己开发其他算法，有兴趣的同学可以查看 **wiki** 和其他相关材料，例如：

  Introduction to Algorithms，**MIT Press**（高教出版社影印）

最短路径（shortest path）

- 定义（最短路径问题）：
  - 在网络或带权有向图里，从顶点 **v** 到 **v'** 的一条路径上各条边的长度之和称为该路径的长度
  - 从 **v** 到 **v'** 的所有路径中长度最短的路径就是最短路径，最短路径的长度称为从 **v** 到 **v'** 的距离，记为 **dis(v, v')**
- 最短路径在实际应用中特别有价值，许多调度问题与此有关：
  - 运输（最短里程，最短运费，最低成本，最少时间等）
  - 加工或者工作的流程，等等
- 通过从 **v** 出发，遍历能到达 **v'** 所有可能路径，可以从中找出最短路径，显然可行（例如用深度或宽度优先搜索），但这不是很有效的方法，其中还有一些麻烦（例如，如果存在环，可达路径就有无穷多条）。请自己考虑如何写出这样的算法，并分析算法的复杂度
- 人们已经开发了一些更为有效的算法

单源点最短路径

- 单源点最短路径问题要求求出从一给定顶点到其他所有顶点的最短路径
- **Dijkstra** 算法能求出一个顶点到所有其他结点的最短路径。这个算法也顺便解决了对给定的 **v** 到 **v'** 求最短路径的问题
  - 该算法要求所有边的权不小于 **0**
  - 有人提出了在允许负数权边的图中的单源点最短路径问题，可以参看 **wiki** 的有关页面，或参考其他相关文献
- **Dijkstra** 算法的工作过程与 **Prim** 算法类似，利用了另一与 **MST** 类似的性质。假设要找从顶点 v_0 到其他顶点的最短路径
  - 算法执行过程中把图中顶点分为两个集合：当时已知最短路径的顶点集合 **U** 和尚不知道最短路径的顶点集合 **V - U**
  - 算法逐步扩充已知最短路径的顶点集合，每次从 **V – U** 中找到一个新顶点（它是当时已能确定最短路径的顶点）加入 **U**
  - 反复这样做，直到找出从 v_0 到所有顶点的最短路径。算法能同时给出这些最短路径及其长度（距离）

Dijkstra 算法

- 如何为 **U** 扩充顶点，即，如何在 **V-U** 里找到下一个能确定最短路径的顶点？
- 定义从 v_0 到 $v' \in V-U$ 的当前已知最短路经长度 $cdis(v_0, v')$ 定义为：

$$cdis(v_0, v') = \min\{dis(v_0,v) + w(v,v') \mid v \in U \wedge (v, v') \in E\}$$

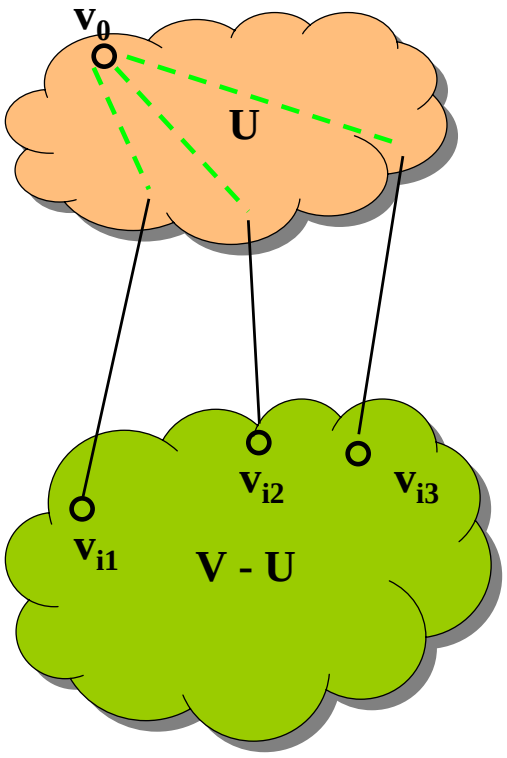
  其中 $w(v,v')$ 是边 (v, v') 的权

  如果没有这种 **v** 则令 $cdis(v_0, v') = \infty$

- 性质：如果 **v'** 是 **V-U** 中 **cdis** 值最小的顶点，那么 $dis(v_0, v') = cdis(v_0, v')$

  也就是说：从 v_0 到 **v'** 的最短路径已知，可以立刻把 **v'** 加入集合 **U**

  这一性质不难证明（请自己想想）

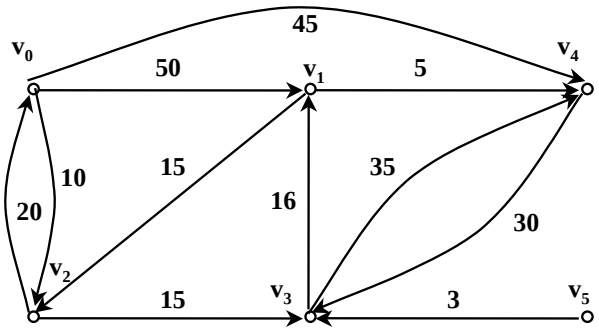


Dijkstra 算法梗概

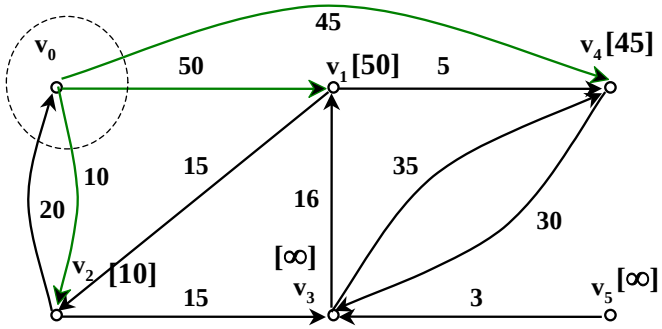
- 初始：在集合 U 中放入顶点 v_0
  - v_0 到 v_0 的距离为 0
  - 对 V – U 里的顶点 v，如果 $(v_0, v) \in E$，则 v 的已知最短路经长为 $w(v_0, v)$，否则令 v 的已知最短路经长为 ∞
- 从 V – U 中选出当时已知最短路长度最小的顶点 v_{min} 加入 U（这时到 v_{min} 的已知最短路长 $cdis(v_0, v_{min})$ 就是 v_0 到 v_{min} 的距离 ）
- 由于 v_{min} 加入，V – U 中某些顶点的已知最短路经长度可能改变：
  - 如果从 v_0 经过 v_{min} 到 v' 的路径长度比原已知最短路长更短，这就是到 v' 的新的已知最短路经长度，该路径经过 v_{min} 到 v'
  - 记录新的最短路径及距离，支持下面继续选择 V-U 的最近顶点
- 反复选择顶点并记录新的最短路径信息，直到从 v_0 可达的所有顶点都在集合 U 中为止。如果存在未加入 U 的顶点，就说明图不连通

Dijkstra 算法的过程示例

例：已知带权图及其邻接矩阵A，求顶点 v_1 到其它各顶点的最短路径



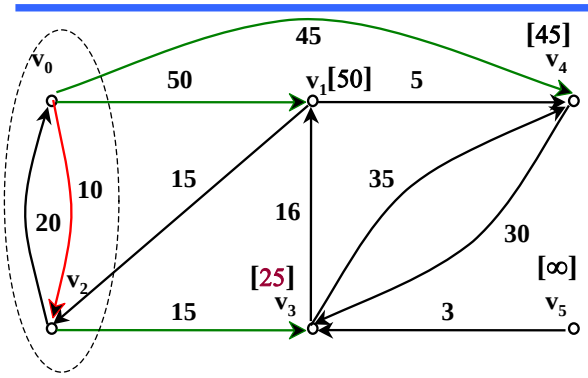
$$A = \begin{bmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 5 & \infty \\ 20 & \infty & 0 & 15 & \infty & \infty \\ \infty & 16 & \infty & 0 & 35 & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{bmatrix}$$



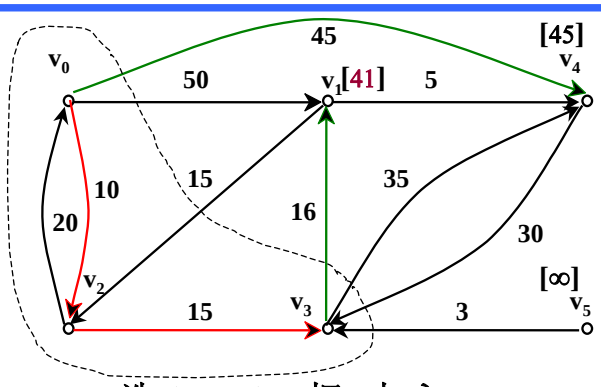
绿色为从 U 到 V-U 的边，

选 (v_0, v_2) 并把 v_2 加入U

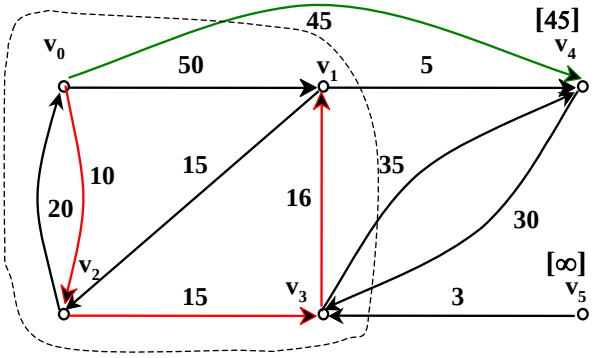
Dijkstra 算法的过程示例



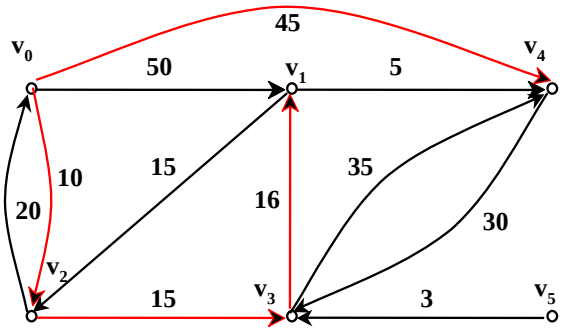
选 (v_2, v_3)，把v_3加入U



选 (v_3, v_1)，把v_1加入U



选 (v_0, v_4)，把v_4加入U



Dijkstra 算法的实现

- 性质：如果 v' 是初始点 v_0 到 v 的最短路径 p 上 v 前一个顶点，那么 p 去掉最后顶点 v 得到的路径 p' 也是 v_0 到 v' 的最短路径

  这个性质很容易通过反证法证明：如果 p' 不是 v_0 到 v' 的最短路径，那么 p 也不是 v_0 到 v 的最短路径

- 下面算法利用了这个性质
  - 记录 v_0 到 v 的最短路径，只需记录 v_0 到 v 最短路径上的前一顶点
  - 所以，记录所有最短路径只需要一个 n-1 元的边集合
- vnum 元表 paths 的元素 paths[v] = (v', p) 记录从 v_0 到顶点 v 的最短路径和路径长度，v' 是 v_0 到 v 最短路径上 v 的前一顶点，p 是最短路径长度。此外，paths[v] = None 也表示 v 还不在 U 里
- 最短路径的候选边集按路径长度记录在优先队列 cands 里，元素形式 (p, v, v') 表示从 v0 经 v 到 v' 的最短路径长度为 p。按 p 值在 conds 里排序。如最短候选边的终点 v' 在 V-U，将其加入 paths，并将经由 v' 可达的其他顶点及其路径长度记入 cands

Dijkstra 算法的实现

```
def dijkstra_shortest_paths(graph, v0):
    vnum = graph.vertex_num()
    assert 0 <= v0 < vnum
    paths = [None]*vnum
    count = 0
    cands = PrioQueue([(0, v0, v0)])      # 初始队列
    while count < vnum and not cands.is_empty():
        plen, u, vmin = cands.dequeue() # 取路径最短顶点
        if paths[vmin]: continue          # 如果其最短路径已知则继续
        paths[vmin] = (u, plen)           # 记录新确定的最短路径
        for v, w in graph.out_edges(vmin): # 考察经由新 U 顶点的路径
            if not paths[v]: # 是到尚未知最短路径的顶点的路径，记录它
                cands.enqueue((plen + w, vmin, v))
        count += 1
    return paths
```

算法的基本结构和 Prim 算法相同，只是记入优先队列的“权值”不同

Dijkstra 算法的复杂度

- Dijkstra 算法的时间复杂度：
  - 算法中的初始化部分的时间复杂度不超过 O(|V|)
  - 算法主体部分与 Prim 算法类似，复杂性是 O(|E| log |E|)
- 张老师的教科书给出了一个时间复杂度为 $O(|V|^2)$ 的算法，可参考
- 算法的空间复杂度是 O(max(|E|, |V|)) = O(|E|)，主要是 paths 需要保存 |V| 个顶点的信息，优先队列 cands 可能保存 O(|E|) 条边的信息
- 这个算法里需要找到更短的路并更新相关信息，与 Prim 算法里的情况类似。也可以采用前面的“可减权堆” 做出 O(|V|) 空间的算法

  有关算法请自己考虑
- 下次课研究图中各对顶点之间的最短路径问题

  一个显然的解法是利用 Dijkstra 算法 |V| 次。但下次课会给出另一个想法与此完全不同的算法，它一下做出所有最短路径

问题，方法，算法和程序

- 以求网络的最小生成树为例，这是一个问题。**Prim** 和 **Pruskal** 提出了解决方法，分别基于网络的 **MST** 性质和简单连通分支扩充。两者都是抽象的算法。基于它们都可能设计出多种不同的具体算法
  - 其中可能选用不同的数据结构，具体过程也可能有异
  - 不同的实现（实际算法）又可能具有不同的复杂性
- 对最小生成树问题，已知其算法时间复杂度的下界 **O(|E|)**，但
  - 尚未证明这是下确界（是否可能达到？），也没找到 **O(|E|)** 算法
  - 因此最小生成树问题的最快算法仍是一个 **open problem**
  - 可见，这个问题既有实际价值，也有理论追求
- 基于某种算法，可以（用某个编程语言）写出具体的程序
  - 虽然，编程不当也可能达不到算法可能达到的最高效率（复杂度）
  - 需要理解所用的语言机制和数据结构