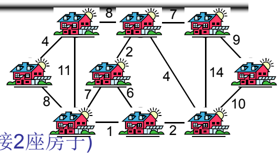
算法设计与分析
Algorithms Design & Analysis

第十一讲：最小生成树

最小生成树(Minimum Spanning Trees)

Problem



- A town has a set of houses and a set of roads
- A road connects 2 and only 2 houses(每条路连接且仅连接2座房子)
- A road connecting houses u and v has a repair cost w(u, v) (连接u和v房子的路的维修代价为w(u, v))

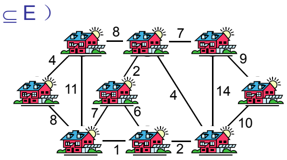
Goal: Repair enough (and no more) roads such that: (维修仅够需求的路,使得:)

1. Everyone stays connected: can reach every house from all other houses, and (每两座房子之间保持连通)
2. Total repair cost is minimum(维修代价最小)

最小生成树(Minimum Spanning Trees)

- **A connected, undirected graph: (连通无向图)**
  - **Vertices = houses, Edges = roads（顶点为房子，边为路）**
- **A weight w(u, v) on each edge (u, v) ∈ E （维修费用为边的权）**

Find T ⊆ E such that: （确定T ⊆ E ）

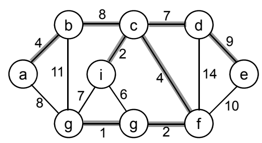


1. T connects all vertices

   （T连接所有的顶点）
2. $w(T) = \Sigma_{(u,v)\in T}\, w(u, v)$ is minimized （并且$w(T) = \Sigma_{(u,v)\in T}\, w(u, v)$最小）

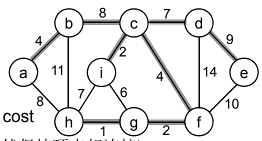
最小生成树(Minimum Spanning Trees)

- T forms a tree = **spanning tree (T为生成树)**
- A spanning tree whose weight is minimum over all spanning trees is called a ***minimum spanning tree***, or ***MST***. (所有生成树中边的权值之和最小的为最小生成树)



最小生成树特性(Properties of Minimum Spanning Trees)

- Minimum spanning trees are not unique(MST非唯一)
  - Can replace (b, c) with (a, h) to obtain a different spanning tree with the same cost (图中的(b, c) 和 (a, h)可以互换)
- MST have no cycles(没有回路)
  - We can take out an edge of a cycle, and still have the vertices connected while reducing the cost (如果存在回路,就可以取消一条边,而依然保持顶点相连接)



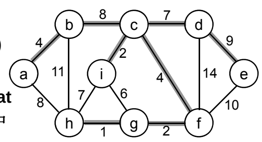
- \# of edges in a MST: (MST边的数目为顶点数目减1)
  - |V| - 1

MST生成(Growing a MST)

Minimum-spanning-tree problem: find a MST for a connected, undirected graph, with a weight function associated with its edges (问题:确定连通无向有权图的MST)

A generic solution: (一般方法)

- **Build a set A of edges (initially empty) (创建一个边的集合A,起始时为空)**
- **Incrementally add edges to A such that they would belong to a MST(逐渐向A中添加边并保持A属于某个MST)**
- **An edge (u, v) is safe for A if and only if A ∪ {(u, v)} is also a subset of some MST (当且仅当A ∪ {(u, v)}是MST的子集，我们说边(u, v)对于A是安全的）**

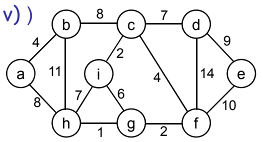


We will add only safe edges

MST一般算法(Generic MST algorithm)

1. A ← ∅
2. **while** A is not a spanning tree (没有形成生成树)
3. **do** find an edge (u, v) that is safe for A

   (寻找对A安全的边(u, v))
4. A ← A ∪ {(u, v)}
5. **return** A

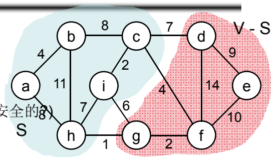


- How do we find safe edges?(如何找到安全的边?)

Finding Safe Edges

- Let's look at edge (h, *g*)
  - Is it safe for A initially?

(考察边(h, g)，它在初始时对于A是否是安全的?)

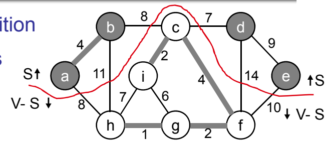


- Later on:
  - Let S ⊂ V be any set of vertices that includes h but not *g* (so that *g* is in V - S) (S ⊂ V是包含顶点h但不包含顶点*g*的集合, *g* ⊂ V - S)
  - In any MST, there has to be one edge (at least) that connects S with V - S (在MST上, 至少有一条边连接S和V - S)

一些定义(Definitions)

- A **cut** (S, V - S) is a partition of vertices into disjoint sets S and V – S

(切割:将图的顶点分成两个互不相连的两个集合S和V – S)



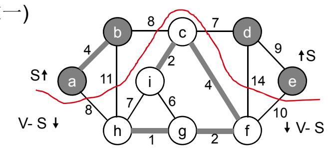
- An edge **crosses** the cut

(S, V - S) if one endpoint is in S and the other in V – S (如果一个边的两个端点分别在切割的两个集合S和 V - S中,我们说该边横跨该切割)

一些定义(Definitions)

- A cut **respects** a set *A* of edges ⇔ no edge in A crosses the cut (一个切割不干预集合A ⇔集合A中没有边与该切割相交)
- An edge is a **light edge** crossing a cut ⇔ its weight is minimum over all edges crossing the cut (横跨切割的轻边⇔所有横跨切割的边中权值最小的边)
  – For a given cut, there can be > 1 light edge crossing it

  (light edge > 1可以不唯一)



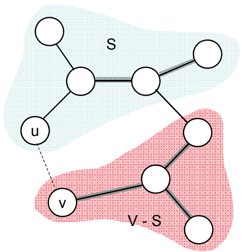
定理(Theorem)

- Let A be a subset of some MST, (S, V - S) be a **cut** that respects A, and (u, v) be a **light edge** crossing (S, V - S). Then (u, v) is safe for A. (A是MST的子集， (S, V - S)是不干预A的一个切割， (u, v)是横跨该切割的轻边，那么， (u, v)对于A是安全的)

Proof:

- Let T be a MST that includes A

  (假设T是包含A的MST)
  – Edges in A are shaded
- Assume T does not include the edge (u, v) (T不包含边(u, v))
- **Idea**: construct another MST T' that includes A ∪ {(u, v)} (构建另一MST T'，它包括A ∪ {(u, v)})



定理证明(Theorem – Proof)

- **T contains a unique path p between u and v**

(因为T是MST，所以u 和 v存在唯一的连通路径p)

- **(u, v) forms a cycle with edges on p**

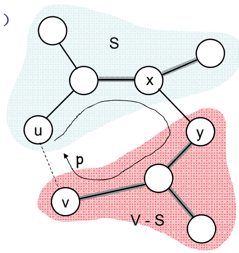
((u, v)与路径p形成回路)

- **(u, v) crosses the cut ⇒ path p must cross the cut (S, V - S) at least once: let (x, y) be that edge**

(由于(u, v)横跨切割，要形成回路， p中至少存在一条边横跨该切割，设为(x, y))

- **Let's remove (x, y) ⇒ breaks T into two components. (去掉(x, y) ，变成两个部分)**
- **Adding (u, v) reconnects the components (添加(u, v) ，使之再连接)**

  T' = T - {(x, y)} ∪ {(u, v)} (构造T')



定理证明续Theorem – Proof (cont.)

$T' = T - \{(x, y)\} \cup \{(u, v)\}$

Have to show that T' is a MST:

（证明T'是MST）

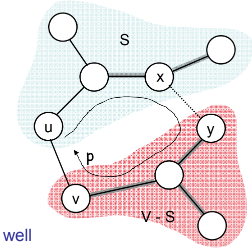
- (u, v) is a light edge （因为(u, v)是LE）

  $\Rightarrow w(u, v) \le w(x, y)$
- $w(T') = w(T) - w(x, y) + w(u, v)$

  $\le w(T)$
- Since T is a MST （又因为T是MST）

  $w(T) \le w(T') \Rightarrow T'$ must be an MST as well

（两方面综合可得T'是MST）



13

定理证明续Theorem – Proof (cont.)

Need to show that (u, v) is safe for A:

（证明(u, v)对于A是安全的，即是某个MST的部分）

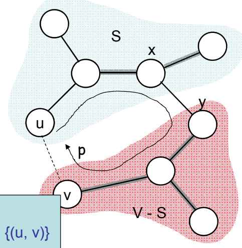
i.e., (u, v) can be a part of a MST

- $A \subseteq T$ and $(x, y) \notin A \Rightarrow A \subseteq T'$
- $A \cup \{(u, v)\} \subseteq T'$
- Since T' is an MST

$\Rightarrow$ (u, v) is safe for A

WHY?

$T' = T - \{(x, y)\} \cup \{(u, v)\}$



14

讨论(Discussion)

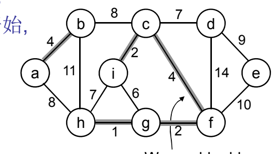
In GENERIC-MST:

- *A* is a forest containing connected components (A 是森林，包含连通的几个组件，在初始时刻，每个组件只是一个顶点）
  - Initially, each component is a single vertex
- Any safe edge merges two of these components into one （加入安全边将其中的两个组件为一）
  - Each component is a tree （每个部分是一棵树）
- Since an MST has exactly |V| - 1 edges - after iterating |V| - 1 times, we have only one component （MST有|V| - 1条边，因此需要进行|V| - 1次安全边的加入，这时候只有一个连通的组件，即MST）

15

Kruskal 算法(The Algorithm of Kruskal)

- Start with each vertex being its own component (从每个顶点开始，每个顶点即是个组件)
- Repeatedly merge two components into one by choosing the light edge that connects them(通过选择LE，不断的将两个组件连接起来）



We would add edge (c, f)

- Scan the set of edges in monotonically increasing order by weight（依权值递增顺序检查边的集合）
- Uses a disjoint-set data structure to determine whether an edge connects vertices in different components（利用不相关集合的数据结构判断边是否连接不同的组件）

16

不相关集合的操作（Operations on Disjoint Data Sets）

- MAKE-SET(u) – creates a new set whose only member is u （创建集合，仅含元素u）
- FIND-SET(u) – returns a representative element from the set that contains u （包含u，返回代表元素）
  - May be any of the elements of the set that has a particular property （集合的任何元素具有某个特定的属性）
  - *E.g.:* $S_u = \{r, s, t, u\}$, the property is that the element be the first one alphabetically （集合S_u的特性是按字母顺序的第一个字母）

    FIND-SET(u) = r FIND-SET(s) = r
  - FIND-SET has to return the same value for a given set （对于给定的集合，返回同样的值）

17

不相关集合的操作（Operations on Disjoint Data Sets）

- UNION(u, v) – unites the dynamic sets that contain u and v, say S_u and S_v
  - *E.g.:* $S_u = \{r, s, t, u\}$, $S_v = \{v, x, y\}$

    UNION (u, v) = {r, s, t, u, v, x, y}

合并操作

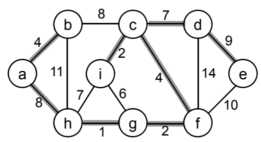
18

KRUSKAL(V, E, w)

```
A ← ∅
for each vertex v ∈ V
      do MAKE-SET(v)
sort E into non-decreasing order by weight w（权值递增排序）
for each (u, v) taken from the sorted list
      do if FIND-SET(u) ≠ FIND-SET(v) （属于不同的组件）
            then A ← A ∪ {(u, v)}
                  UNION(u, v)
return A
```

Running time: O(E lgV) – dependent on the implementation of the disjoint-set data structure

Example



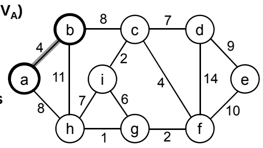
1: (h, g)	8: (a, h), (b, c)
2: (c, i), (g, f)	9: (d, e)
4: (a, b), (c, f)	10: (e, f)
6: (i, g)	11: (b, h)
7: (c, d), (i, h)	14: (d, f)

{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}

1. Add (h, g)	{g, h}, {a}, {b}, {c}, {d}, {e}, {f}, {i}
2. Add (c, i)	{g, h}, {c, i}, {a}, {b}, {d}, {e}, {f}
3. Add (g, f)	{g, h, f}, {c, i}, {a}, {b}, {d}, {e}
4. Add (a, b)	{g, h, f}, {c, i}, {a, b}, {d}, {e}
5. Add (c, f)	{g, h, f, c, i}, {a, b}, {d}, {e}
6. Ignore (i, g)	{g, h, f, c, i}, {a, b}, {d}, {e}
7. Add (c, d)	{g, h, f, c, i, d}, {a, b}, {e}
8. Ignore (i, h)	{g, h, f, c, i, d}, {a, b}, {e}
9. Add (a, h)	{g, h, f, c, i, d, a, b}, {e}
10. Ignore (b, c)	{g, h, f, c, i, d, a, b}, {e}
11. Add (d, e)	{g, h, f, c, i, d, a, b, e}
12. Ignore (e, f)	{g, h, f, c, i, d, a, b, e}
13. Ignore (b, h)	{g, h, f, c, i, d, a, b, e}
14. Ignore (d, f)	{g, h, f, c, i, d, a, b, e}

Prim 算法（The algorithm of Prim）

- **The edges in set A always form a single tree** （集合A中的边形成一棵树）
- **Starts from an arbitrary "root": V_A = {a}** （从任意选定的根开始）
- **At each step:** （每次）
  - **Find a light edge crossing cut (V_A, V - V_A)**
    （确定切割(V_A, V - V_A)的LE）
  - **Add this edge to A（加入A）**
  - **Repeat until the tree spans all vertices**
    （直到树包含所有顶点）

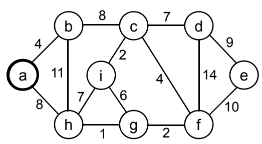


- **Greedy strategy（贪婪策略）**
  - **At each step the edge added contributes the minimum amount possible to the weight of the tree** （每次加入的边使树的边的权值和增加最小）

如何迅速确定LE（How to Find Light Edges Quickly?）

Use a priority queue Q:（利用优先队列）

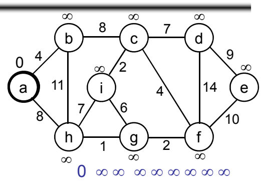
- **Contains all vertices not yet included in the tree (V – V_A)**
  （包含所有不在树中的顶点(V – V_A)）
  - **V_A = {a} , Q = {b, c, d, e, f, g, h, i}**



- **With each vertex we associate a key:**
  每个顶点赋予一个关键值，即树中连接至v的任何边(u，v)的最小权值）
  key[v] = minimum weight of any edge (u, v) connecting v to a vertex in the tree
  - **Key of v is ∞ if v is not adjacent to any vertices in V_A（如果v与V_A任何的顶点不相连，则关键值为∞）**
  - **After adding a new node to V_A we update the weights of all the nodes adjacent to it（增加节点后，所有与之相连的节点的关键值要做调整）**
  - **We added node a ⇒ key[b] = 4, key[h] = 8**

PRIM(V, E, w, r)

```
Q ← ∅
for each u ∈ V
    do key[u] ← ∞
       π[u] ← NIL
key[r] ←0
Q ← V
while Q ≠ ∅
      do u ← EXTRACT-MIN(Q)
         for each v ∈ Adj[u]
              do if v ∈ Q and w(u, v) < key[v]
                    then π[v] ← u
                         key[v] ← w(u, v)
```



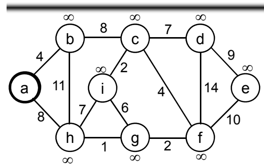
0 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞

Q = {a, b, c, d, e, f, g, h, i}

V_A = ∅

Extract-MIN(Q) ⇒ a

Example

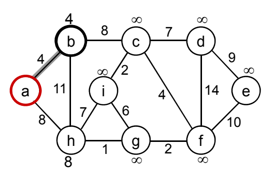


0 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞

Q = {a, b, c, d, e, f, g, h, i}

V_A = ∅

Extract-MIN(Q) ⇒ a



key [b] = 4   π [b] = a

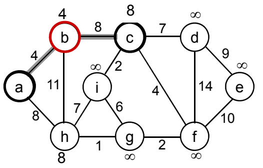
key [h] = 8   π [h] = a

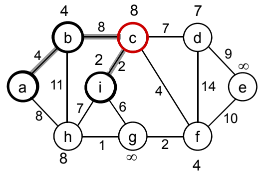
4 ∞ ∞ ∞ ∞ ∞ 8 ∞

Q = {b, c, d, e, f, g, h, i} V_A = {a}

Extract-MIN(Q) ⇒ b

Example

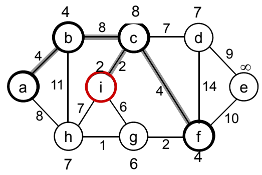


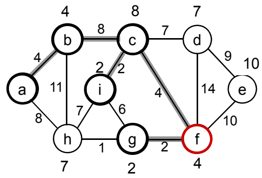


key [c] = 8 π [c] = b
key [h] = 8 π [h] = a - unchanged
8 ∞ ∞ ∞ ∞ 8 ∞
Q = {c, d, e, f, g, h, i} V_A = {a, b}
Extract-MIN(Q) $\Rightarrow$ c

key [d] = 7 π [d] = c
key [f] = 4 π [f] = c
key [i] = 2 π [i] = c
7 ∞ 4 ∞ 8 2
Q = {d, e, f, g, h, i} V_A = {a, b, c}
Extract-MIN(Q) $\Rightarrow$ i

Example

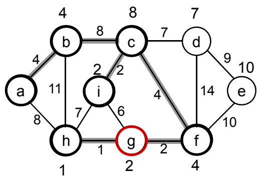


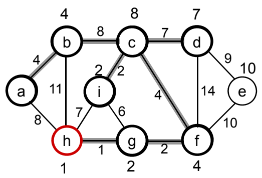


key [h] = 7 π [h] = i
key [g] = 6 π [g] = i
7 ∞ 4 ∞ 8
Q = {d, e, f, g, h} V_A = {a, b, c, i}
Extract-MIN(Q) $\Rightarrow$ f

key [g] = 2 π [g] = f
key [d] = 7 π [d] = c unchanged
key [e] = 10 π [e] = f
7 10 2 8
Q = {d, e, g, h} V_A = {a, b, c, i, f}
Extract-MIN(Q) $\Rightarrow$ g

Example

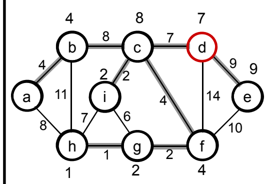




key [h] = 1 π [h] = g
7 10 1
Q = {d, e, h} V_A = {a, b, c, i, f, g}
Extract-MIN(Q) $\Rightarrow$ h

7 10
Q = {d, e} V_A = {a, b, c, i, f, g, h}
Extract-MIN(Q) $\Rightarrow$ d

Example



key [e] = 9 π [e] = f
9
Q = {e} V_A = {a, b, c, i, f, g, h, d}
Extract-MIN(Q) $\Rightarrow$ e
Q = $\varnothing$ V_A = {a, b, c, i, f, g, h, d, e}

PRIM(V, E, w, r)

```
Q ← ∅                                          Total time: O(VlgV + ElgV) = O(ElgV)
for each u ∈ V
    do key[u] ← ∞                              O(V) if Q is implemented
       π[u] ← NIL                              as a min-heap
key[r] ← 0
Q ← V
while Q ≠ ∅            ←—— Executed |V| times   } Min-heap operations:
    do u ← EXTRACT-MIN(Q)  ←—— Takes O(lgV)      } O(VlgV)
       for each v ∈ Adj[u]  ←—— Executed O(E) times   }
           do if v ∈ Q and w(u, v) < key[v]  ←—— Constant   } O(ElgV)
                 then π[v] ← u                ——— Takes O(lgV) }
                      key[v] ← w(u, v)
```