第四章 贪心算法

苏 明

内容安排

- 4.1 区间调度：贪心算法领先
- 4.2 最小延迟调度：一个交换论证
- 4.3 最有高速缓存：一个更复杂的交换论证
- 4.4 一个图的最短路径
- 4.5 最小生成树问题
- 4.6 实现Kruskal 算法
- 4.7 聚类
- 4.8 Huffman 码与数据压缩

贪心算法

- 贪心（greedy）看起来会有一点负面的印象
- 从正反两面研究“目光短浅”的贪心法时，要有一种全面的理解。

贪心算法

- 直观上说，一个算法是贪心的，如果此算法是通过一些小的步骤来建立一个解，在每一步根据**局部情况选择**一个决定使得某些主要的指标能得到优化。
- 当一个贪心算法成功求解了一个非平凡的问题，通常隐含了某些有趣的，与问题本身结构有关的一些性质：**存在一个局部判断规则**可以用来构造问题的**最优解**。

贪心算法

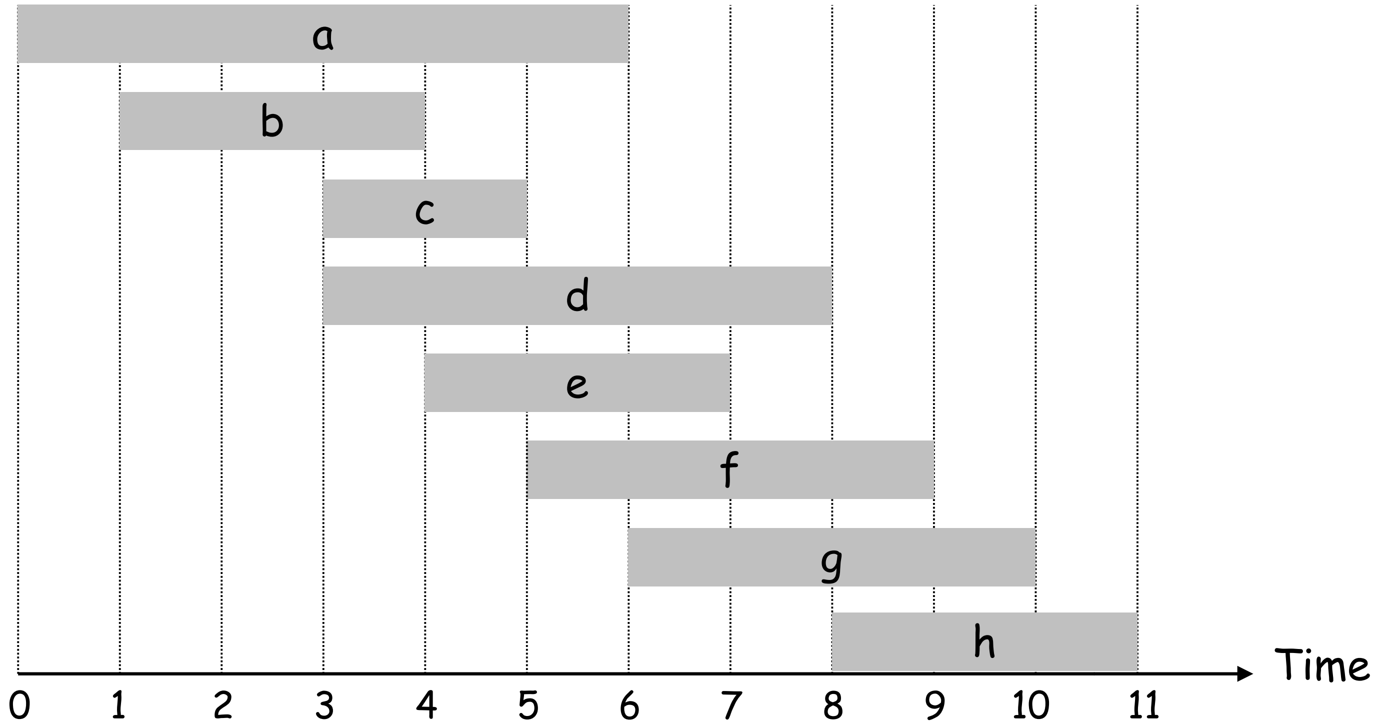
- 本章逐渐介绍两个基本的方法来证明一个贪心算法对一个问题能够提供一个最优解。
- 1. **贪心算法领先**的概念：每一步都比其他的算法好，从而证明产生了一个最优解。
- **2.交换论证**：考虑对这个问题的任何可能解，逐渐把它*转换*成由贪心算法找到的解，而且不影响解的质量。从而证明了贪心法找到了一个至少与其它解一样好的解。

4.1 区间调度：贪心算法领先

- 回顾第一章考虑的五个典型问题的第一个问题：一组需求{1,2,…,n}; 第i个需求与一个始于s(i)且止于f(i)的时间区间相对应。如果没有两个需求在时间上重叠，我们就说需求的子集是相容的。目标是给出一个最大的相容子集。

4.1 区间调度

- **区间调度问题：**
- 任务（需求）j 开始时间 s_j，结束时间 f_j.
- 两个任务是相容的如果任务的时间区间不相交
- 目的：找到数目尽可能多的相容任务



4.1 区间调度

- 解决问题的思路
- 按照某种顺序来安排任务
- 需要通过**探索**，来发现一个实用的规则

4.1 区间调度

- 第一次探索(One)
- 选最早开始的有效需求：即一个**具有最小开始时间s(i)**的需求；好处是按这种方式可以尽早开始使用我们的资源。

4.1 区间调度

- 这种方案可行吗？换句话说，是不是这种方案可以给出最大相容子集？

很可惜，这种方案不可行，上面的图例中，按照这种方案只能给出一个区间，但是实际上有更好的蓝色方案（4个相容区间）

4.1 区间调度

- 换一种方案(Two)
- 从接受最小时间区间的需求开始：**即时间区间=f(i)-s(i)尽可能小**。看起来可以尽可能多塞下一些需求。

4.1 区间调度

- 这种方案可行吗？换句话说，是不是这种方案可以给出最大相容子集？

这种方案也不可行，如上图所示，按照这种方案只能给出一个黄色区间，但是有更好的蓝色方案。

4.1 区间调度

- 换一种新方案(Three)

- 选择具有最少“**冲突**”的区间，也就是对每个需求i，我们计算i与其他需求不相容的数目，然后从小到大排序。

4.1 区间调度

- 这种方案可行吗？

从这个图中可以看出，如果选取了冲突最小的黄色方块，那么还是没有最上面的4个蓝色方块好。

4.1 区间调度

- 那么最好的方案应该是什么？
- 吸取前面的经验---与开始时间、区间的宽度、最少“冲突”都有一点联系
- 应该选择什么样的规则？
- 我们应该接受最早结束的需求，即f(i)尽可能小的需求i为第一个需求。这样的好处是**资源尽可能早被释放**，以便于安排下面的需求。

贪心算法

- 初始令R是所有需求的集合，设A为空
- While R 非空
- 选择一个最小结束时间的需求i∈R
- 把i加到A中
- 从R中删除与需求i不相容的所有需求
- Endwhile
- 返回集合A作为被接受的需求集合

分析算法

- 这个探索出来的贪心法是很自然的，但返回一个最优的区间集合却不是显而易见的。目前关于这个算法的最优性的论断仅仅是一种感觉；前面三种失败的方案从感觉上来说似乎可行，但验证后却容易举出反例。
- 下面用贪心算法领先的思路来**严格说明**这个贪心算法的最优性。

分析算法

- 命题4.1 由上面贪心规则返回的集合A中的区间都是相容的。

- 需要证明这个解是最优的，为了便于比较，令O**是一个最优的区间集合**。证明的主要思想是贪心算法生成的集合A“领先”于集合O.

分析算法

- 目标是为了证明|A|=|O|。

- 引入如下记号：设$i_1,i_2,...,i_k$是A中的需求，并按照加入到A的次序排列,|A|=k. 类似的，$j_1,j_2,...,j_m$是O中的需求。目标是为了证明k=m.

分析算法

- 我们的贪心规则保证：A中的每个区间至少与集合O中对应的区间结束得一样早。

- 命题4.2 对所有的指标$r<=k$, 我们有$f(i_r)<=f(j_r)$.

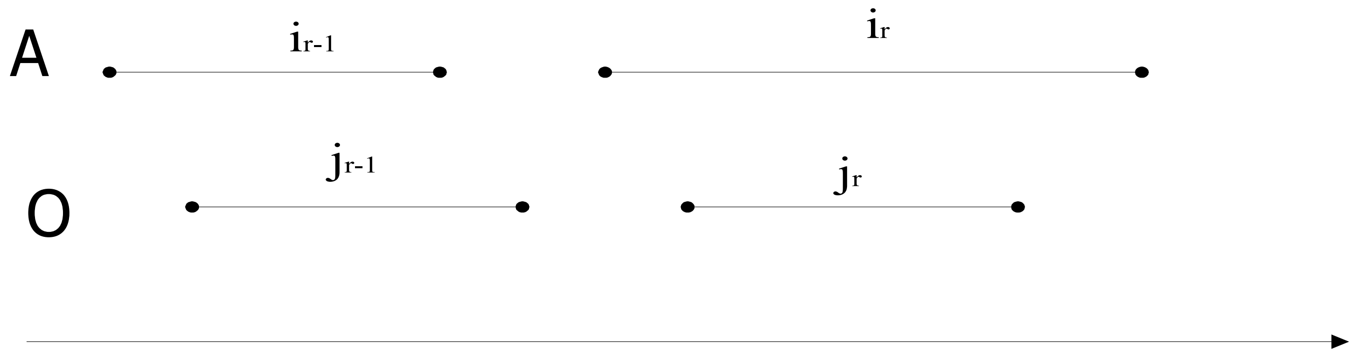
分析算法

- 证明：用归纳法
- 对于r=1，论断为真。
- 如果r>1,根据归纳假设，**假定这个论断对r-1为真，我们将试图证明对r也为真。**

如下图,由归纳假设我们知道$f(i_{r-1})<=f(j_{r-1})$,

为了使算法的第r个区间不更早结束，假设它处于“落后状态”, 即$f(i_r)>f(j_r)$,下面将导出矛盾。

分析算法



- 从上面可以看出： $f(j_{r-1})<=s(j_r)$; 又有 $f(i_{r-1})<=f(j_{r-1})$； 所以$f(i_{r-1})<= s(j_r)$. 若 $f(i_r)>f(j_r)$意味着A集合中应该选择j_r 而不是i_r, 矛盾。所以$f(i_r)<=f(j_r)$, 归纳成立。

分析算法

- 定理4.3 上面的贪心算法返回一个最优的集合A.
- 证明：如果存在一个更好的最优集合O满足更多的任务需求。即我们有m>k. 根据命题4.2，得到$f(i_r) <= f(j_r)$。在O中存在一个需求j_{r+1}, 在j_r后开始，从而也在i_r后开始. 从贪心算法的步骤中可以看出A*应该不在* *i_r处停止*，矛盾。

实现与运行时间

- 贪心算法. 把任务(需求)按照结束时间递增排序。依次选取与前面已选定任务相容的新任务.

算法时间复杂度?

实现与运行时间

- 开始按结束时间对n个需求排序：$i<j$时有$f(i)<=f(j)$, 这用O(nlogn)时间。再用O(n)时间构造一个数组S[1,2,…,n], S[i]包含s(i)的值;
- 算法执行中，总是选择第一个区间，然后按次序迭代通过区间直到满足$s(j)>=f(1)$的第一个区间j, 选择这个区间。**余下继续寻找。** 这部分代价是O(n).

推广

- 这里考虑的问题是相当简单的区间调度问题，实际背景中会有如下的变化， 比如：

- 我们假定调度算法选择相容子集时调度员**已经知道了所有的需求**。自然的， 调度员需要在获悉全部需求集合之间做出接收或者拒绝某个需求的决定。如果调度员为了搜集其它需求而等的太久，那么顾客(需求者)可能会失去耐心，从而放弃离开。

推广

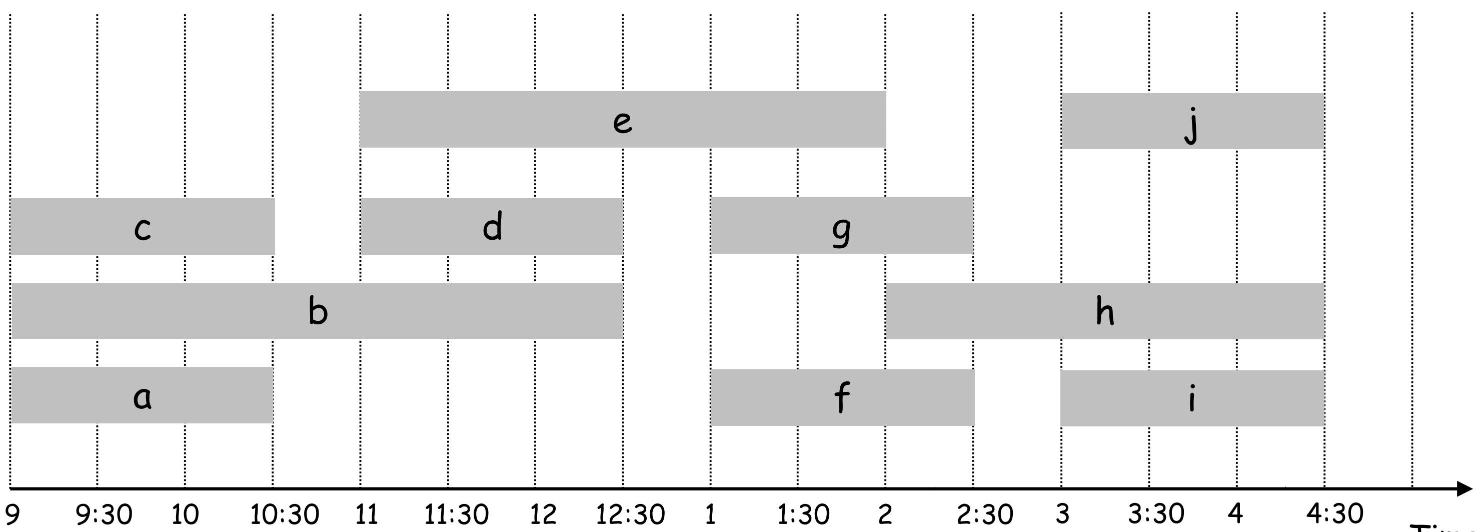
- 一个活跃的研究领域涉及到这种在线算法，调度员必须在不知道进一步输入的情况下随时间不断作出决定。
- 另外一种情况，每个需求会有一个不同的权值，比如每个需求所获得的钱数，目标是极大化我们的收入。这导致了加权的区间调度问题。

区间划分(Interval Partitioning)

- 区间调度问题中，存在一种单一的资源和时间区间表示的很多需求：**如果我们拥有许多相同的资源可用**，而且想尽可能用少的资源安排所有的需求，那么产生一个问题：**区间划分问题**。
- 应用场景：每个需求与特定时间区间教室里安排的课程(演讲)对应，我们想尽可能用少的教室满足所有这些需求。

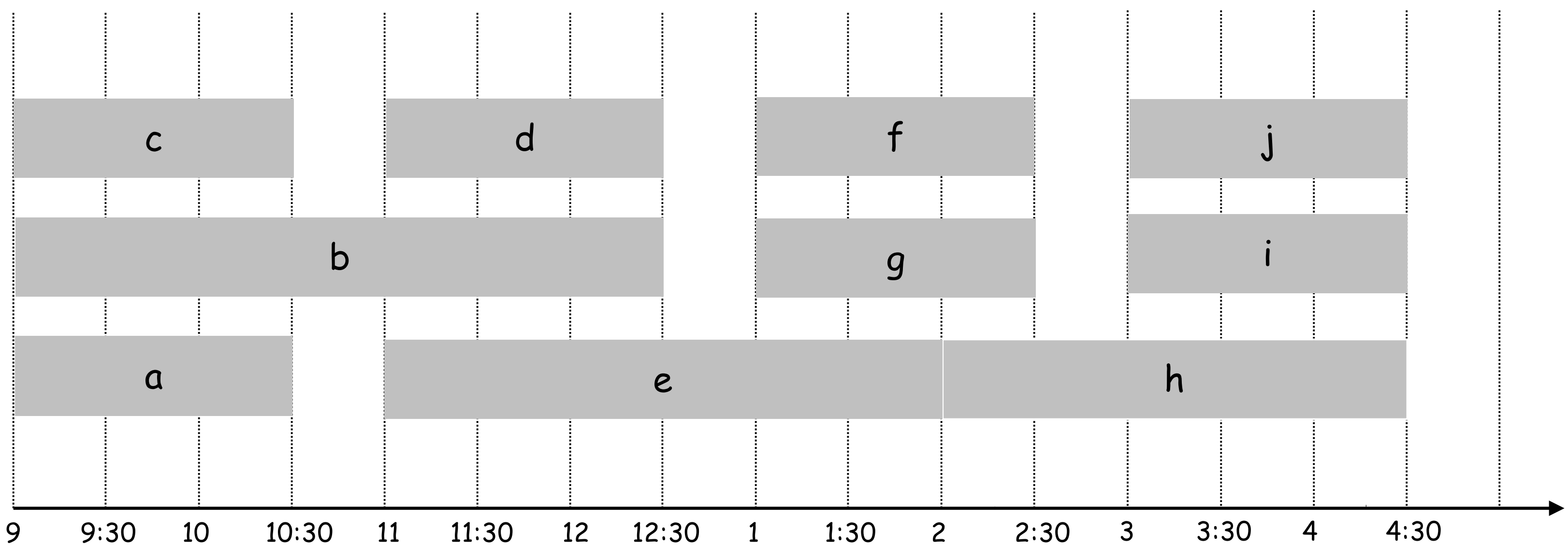
区间划分

- 区间划分问题描述.
  - 演讲 j 从 s_j 开始，在 f_j 结束。
  - 目标：寻找尽可能少的教室满足所有的这些演讲需求，其中同一个教室中没有演讲冲突。
- Ex: 下面的计划用4个教室安排了10个演讲需求。



区间划分

- 有没有更好的方案？用更少的教室？



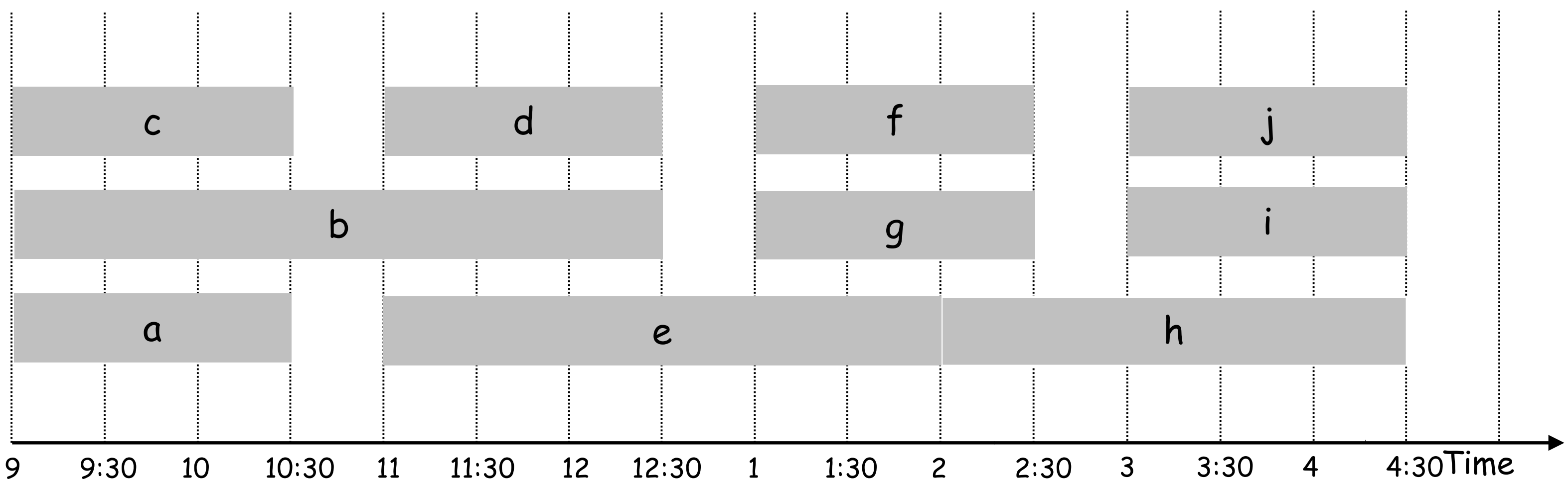
只需要3个教室！

区间划分

- 观察一些关键的因素
- 定义：一个区间集合的深度是通过时间线上任何一点的最大区间数。
- 命题4.4 在任何区间划分的实例中，资源数必须至少是区间集合的深度。

区间划分

- Ex: 下面例子中，区间集合的深度 = 3
  ⇒ 所以给出的划分安排就是最佳方案.
- 问题： 是不是总存在一个区间划分方案使得需要的资源数等于区间集合的深度？



设计算法

- 贪心算法要点：需求(演讲)按照开始时间排序，把需求安排到不冲突的教室中。

```
Sort intervals by starting time so that s1 ≤ s2 ≤ ... ≤ sn.
d ← 0

for j = 1 to n {
   if (lecture j is compatible with some classroom k)
      schedule lecture j in classroom k
   else
      allocate a new classroom d + 1
      schedule lecture j in classroom d + 1
      d ← d + 1
}
```

算法分析

- 命题4.5 如果我们采用上面的贪心算法，每个区间将被分配一个标签，且没有两个重叠的区间接受同样的标签。
- 证明：
- ✓ 没有两个重叠的区间分给同样的标签.
- ✓ 不会有区间在结束时标签不够用；

  考虑一个区间I，以及存在t个区间早于I且与I重叠，那么**t+1<=d**，从而d个标签中至少还剩一个没有分配给I.

算法分析

- 定理4.6 上面贪心算法使用与区间集合深度等量的资源为每个区间安排一个资源，这就是所需资源的最优数量。

算法分析与实现

- 实现方法：
  - 对于每一个教室 k, 记录下最后一个需求的结束时间；
  - 把(已分配)教室放在一个优先队列中(按照结束时间先后)。
- 实现复杂度代价？ O(n log n)

4.2 最小延迟调度

- 问题：我们有单一资源和一组使用资源的n个需求，每个需求需要一个时间区间。假定需求i更加灵活,不是开始时间与结束时间，需求i有一个截至时间d_i,要求一个长度为t_i的连续的时间区间。目标是需要尽可能多的满足需求。

最小延迟调度

描述

- 需求 j 需要长度为 t_j 时间段，截至时间为 d_j.
- 需求 j 如果在 s_j开始, 那么结束时间是$f_j = s_j + t_j$.
- 延迟的定义：$l_j = \max \{ 0, \; f_j - d_j \}$.
- 目标：安排所有的需求，使得计划具有最小的延迟调度：让 $L = \max l_j$最小.

例子

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15

设计算法

- 方案1：把任务按长度t_j增长的次序安排，使得短的任务尽快结束

- 方案可行吗？

设计算法

反例

	1	2
t_j	1	10
d_j	100	10

完全忽略了任务的截止时间！

设计算法

- 方案2 改进：考虑有效*松弛时间*d_i-t_i非常小的任务，按照松弛d_i-t_i增长的次序对任务排序。

- 方案可行吗？

设计算法

- 反例

	1	2
t_j	1	10
d_j	2	10

设计算法

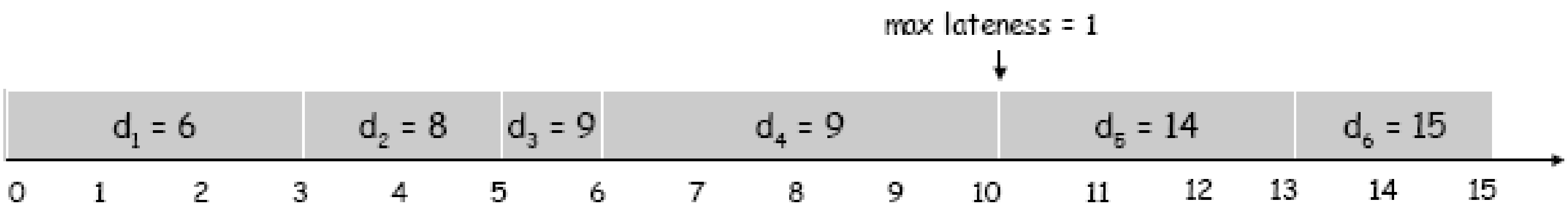
- 方案3 根据直觉基础，应该保证具有**最早截至时间**的任务完成的更早一些。

- 按照*结束时间*d_i*增长的次序*排序(最早截止时间优先)

设计算法

- 例子

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



设计算法

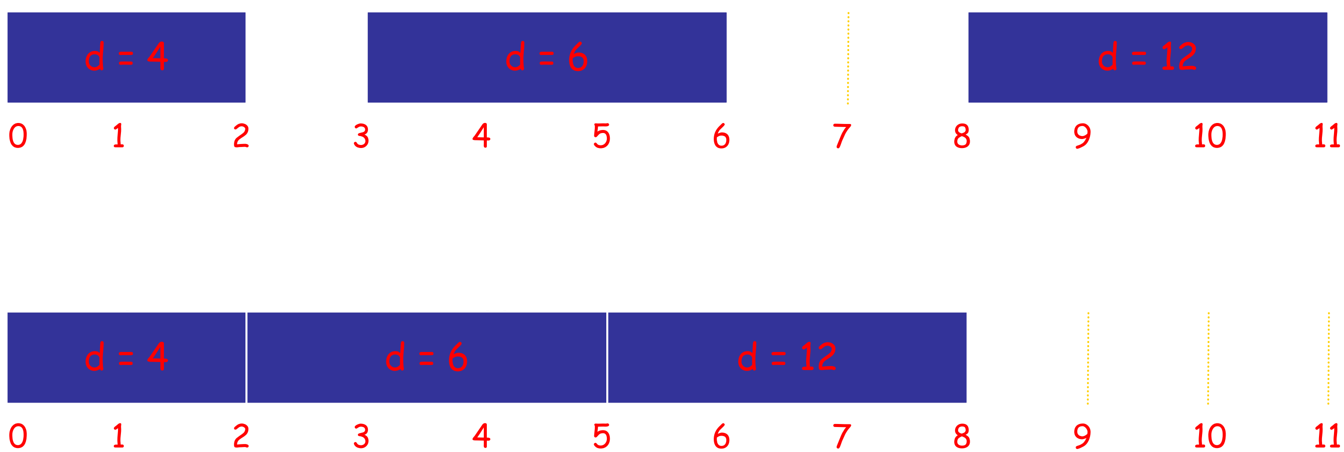
- 贪心算法(最早截至时间优先)

```
Sort n jobs by deadline so that d1 ≤ d2 ≤ … ≤ dn

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tj]
   sj ← t, fj ← t + tj
   t ← t + tj
output intervals [sj, fj]
```

分析算法

- 问题观察：不应有时间“空隙”存在



分析算法

- 命题4.7 存在一个**没有空闲时间**的最优调度

- 思路：开始将考虑一个最优调度O,我们的计划是逐步修改O,每步保持它的最优性，最终把它转换成一个与贪心算法得到的调度A相等的调度。这种类型的分析看作交换论证。

分析算法

- 尝试如下方式刻画调度：一个调度A’有一个逆序，如果具有截至时间d_i的任务i被安排在具有更早截止时间$d_j<d_i$的任务j的前边。

分析算法

- 如果存在很多具有<u>相等截至时间</u>的任务，可能存在很多不同的没有逆序的调度，**对这些任务安排次序的不同会不会有影响？**

命题4.8 所有没有逆序也没有空闲时间的调度有相同的最大延迟。

分析算法

- 证明：两个不同的调度仅可能在具有相同截止时间的任务安排次序上不一样。这两个不同调度中，具有截止时间d的任务全部被连续安排。在这些任务中，**最后一个有着最大的延迟**，而且这个延迟不依赖于这些任务的次序。

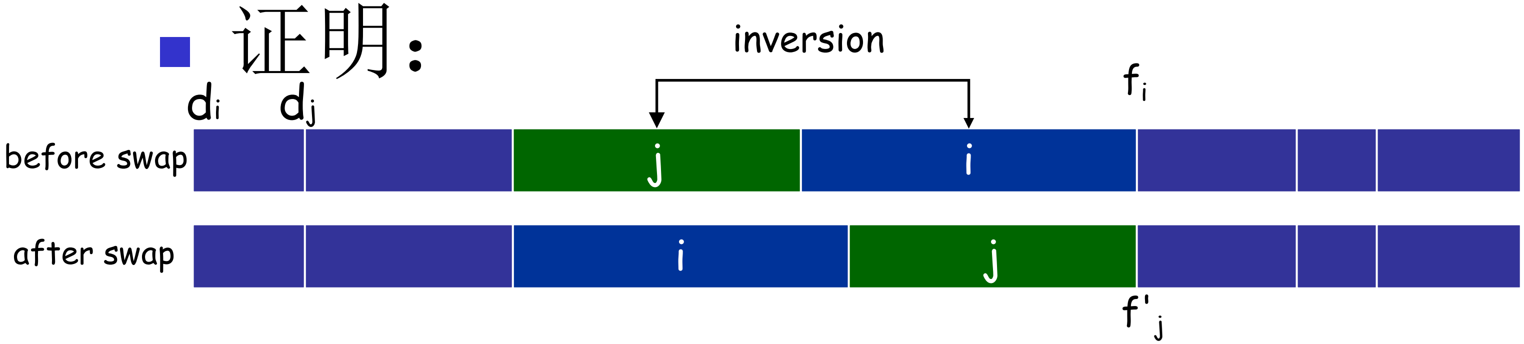
分析算法

- 先考虑最简单情形(没有空闲时间)：如果存在一个**相邻**的逆序任务调度，那么交换以后，所得到的调度方案是不是更好？

- 命题：如果调度中存在一个相邻的逆序任务i,j，那么交换以后，所得到的新调度方案不会增加最大延迟。

分析算法

- 证明：



设ℓ是交换之前的延迟，ℓ'是交换后的延迟。

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- 那么

$$
\begin{aligned}
\ell'_j &= f'_j - d_j && \text{(definition)} \\
&= f_i - d_j && (j \text{ finishes at time } f_i) \\
&\leq f_i - d_i && (i < j) \\
&= \ell_i && \text{(definition)}
\end{aligned}
$$

分析算法

- 定理4.9 存在一个即没有逆序，也没有空闲时间的最优调度。
- 证明

(1)如果调度O有一个**逆序**，那么存在一对**相邻的逆序i,j.**

(2)交换i和j之后，我们得到具有减少一个逆序的新调度。

(3)新的被交换的调度的最大延迟不大于O的最大延迟;

(4)消除所有逆序。

分析算法

- 定理4.10 前面贪心算法产生的调度有最优的最大延迟。
- 证明： 定理4.9证明存在一个没有逆序的最优调度；又根据命题4.8，所有没有逆序的最优调度有着**相同**的最大延迟。所以贪心算法产生最优解。

贪心算法分析要点

- 贪心算法领先(Greedy algorithm stays ahead)
- 交换论证(Exchange argument)
- 结构论证(Structural) 发现一个所有解具有的界限("structural" bound)，然后严格证明算法能够达到这个界。

推广

- 对于需求i,除了截止时间d_i,需要的时间t_i,可能还有一个最早可能的开始时间r_i,最早开始的时间被看成释放时间。具有释放时间的调度问题:
- 需求:能在 *下午1点到5点*之间的某个时间预订教室用于一次*两个小时*的讲座吗?
- 求这个更一般问题的最优解非常困难

4.3 最优超高速缓存

- 问题的产生

- 比如向图书馆借书，每个学生有数量的限制，为了使你在图书馆交换书的次数达到最少，应该采用什么方法？

超高速缓存

- 处理存储分层：少量数据可以被快速的存取，而大量数据需要更多的时间存取；你必须合理的安排数据
- 计算机发展史

内存中存取数据比硬盘存储数据要快得多，但是硬盘有更多的存储容量。

因此内存管理要做的事情：把经常使用的数据块保存在主存中，并且尽可能少访问硬盘。

超高速缓存

- 类似的，处理器的单片超高速缓存比起主存有着更快的存取时间
- 类似的，人们使用Web浏览器时，硬盘常常作为频繁访问的Web网页的超高速缓存，因为访问硬盘的速度大于网上下载的速度
- 速度上分层：

处理器的单片超高速缓存>主存>硬盘>网上下载

超高速缓存

- 超高速缓存 一个快速存储器中存储少量数据以便减少与一个慢速存储器的交互而花费的时间
- 比如：
  - ✓ 主存对于硬盘像超高速缓存
  - ✓ 硬盘对于因特网像超高速缓存
  - ✓ 你的办公桌对于校图书馆像超高速缓存

超高速缓存

- 目的是为了让超高速缓存尽可能有效，当你访问某块数据时，数据应该尽可能在超高速缓存里

- 需要一个超高速缓存维护算法，确定数据何时存在超高速缓存里，何时从超高速缓存中收回。

超高速缓存

- 问题抽象：
- 主存中有n块数据的集合U
- 更快的超高速缓存，一次能够保存$k<n$个数据。从U中取出一系列数据项：$D=d_1,d_2,...,d_n$;　　需要决定在每个时刻哪k个项保存在超高速缓存中。

超高速缓存

- 对某项d_i,如果已经在超高速缓存中(访问命中，cache hit)，那么访问速度很快；否则，需要从主存中调配过来。这时，需要收回在超高速缓存中别的数据，为d_i空出位置。这种情形叫做超高速缓存缺失(cache miss)。我们希望尽可能少出现这些缺失，少做交换。

超高速缓存

- 对于一个特定的存储访问序列，一个超高速缓存算法决定一个**收回调度**。在哪些点把哪些项从超高速缓存中收回，这样就确定了超高速缓存的内容和随时间的交换数目。

超高速缓存

- Ex: k = 2, 初始 cache = ab,
  访问序列: a, b, c, b, c, a, a, b.

requests	cache	
a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b

Cache交换数目？

2

超高速缓存

- 系统研究人员关注的

- 给定一个存储访问的完全序列，什么是使得超高速缓存交换尽可能少的收回调度？

超高速缓存

- "最远将来规则"（ Farthest-in-future ）
  When d_i**需要**被放入超高速缓存
  收回在**最远的将来**被需要的那个项

- 日常生活中的一些经验

超高速缓存

- 例子

current cache:	a	b	c	d	e	f

future queries: g a b c e d a b b a c d e a f a d e f g h . . .

cache miss

eject this one

超高速缓存

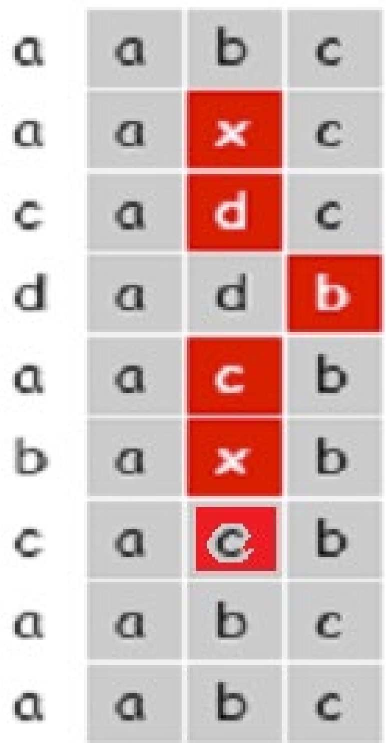
- Theorem. [Bellady, 1960s] FF（Farthest-in-future ）调度方案是最优调度方案，得到最小的**交换**次数。

FF调度分析

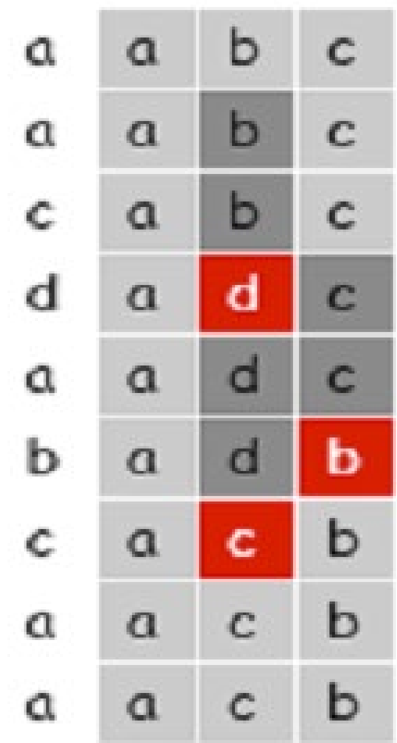
- 定义：如果在第i步有对d的需求，而且d不在超高速缓存中，那么考虑在第i步只放入项d的调度，这种调度被称为简化调度。

- 直觉：能够把非简化调度转化成简化调度，而且高速缓存交换的次数不会增加。

FF调度分析



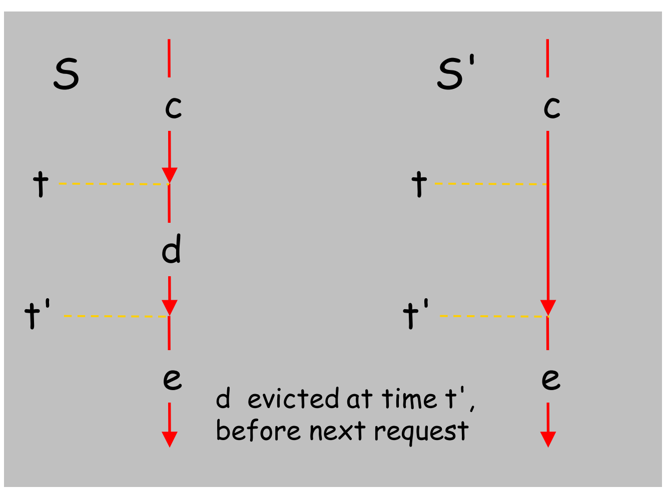
an unreduced schedule



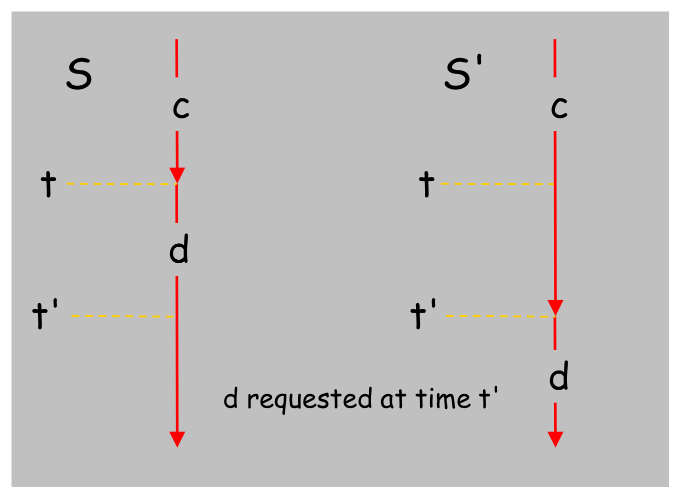
a reduced schedule

FF调度分析

- Claim. 给定任意一个非简化调度S, 可以将其转换成一个简化调度 S'，并不增加缓存交换次数.
- Pf. 假设S在时刻t，把d带入缓存中交换c，尽管这时没有需求。
  - 情形1: **假设在下次需求d之前，** d 在 t'发生交换。
  - 情形2: **假设在时刻t'需求d.**



Case 1



Case 2

FF调度分析

- 定理4.12 对于某个数j,令S是在序列中的前j项与S_{FF}做同样收回决定的简化调度，那么存在一个在序列中的前j+1项与S_{FF}做同样收回决定的简化调度S’,并且S’不比S产生更多的**交换**。

FF调度分析

证明： 通过对需求j进行归纳（存在一个最优的简化调度S，在前面j+1个需求中与S_{FF}的缓存调度的策略完全一样）

设S是一个简化调度，与S_{FF}在前j个调度完全一样。下面我们将构造一个S′使得在前j+1个需求中与S_{FF}完全一样.

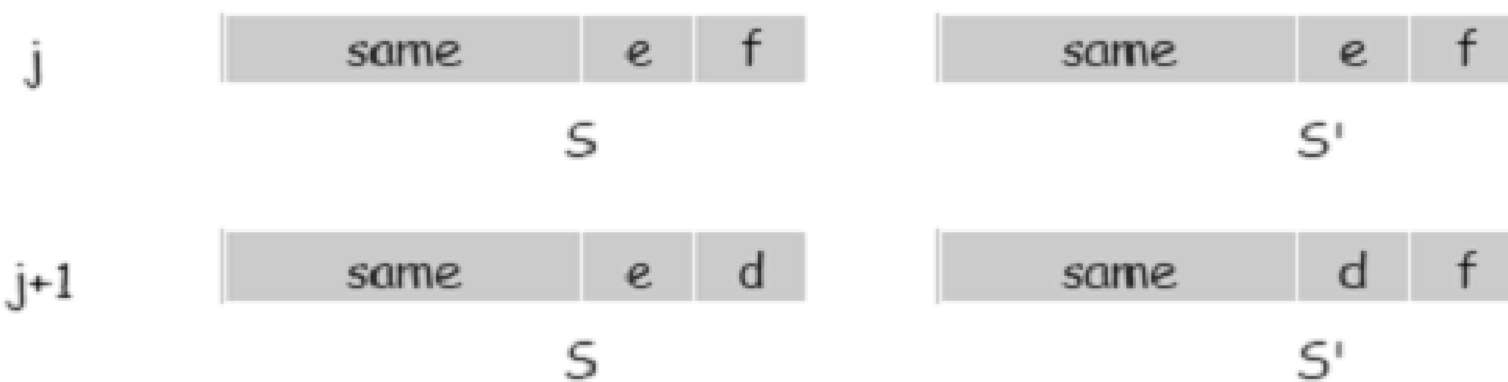
- 考虑第(j+1)个需求$d=d_{j+1}$.
- 既然S和与S_{FF}在前j个调度完全一样，那么它们在第j+1个需求之前缓存中是完全一样的。
- 情形1：(d已经在缓存中) S′=S，满足所需性质.
- 情形2： (d不在缓存中但是S和S_{FF}交换同样的元素)S′=S，满足所需性质.

FF调度分析

证明（续）

情形3： (d不在缓存中，S_{FF}交换e;S交换f ≠ e)

-于是我们构造S′,与S不同的是,交换的是e而不是f



现在S′与S_{FF}在前面j+1个需求都是一样的：下面我们说明在缓存中保留f不比保留e差。

FF调度分析

设j′是第一次在j+1个需求后S和S′采用不同的动作(意味着必须包括e,f的操作)，设g是在时刻j′需要的元素。

Case 3a: g=e 这是不可能的，违反了Farthest-In-Future的性质。

Case 3b: g=f. 因为f不可能在S调度的缓存中，设交换的元素是e′.

-如果e′=e,注意到S′中已经有f;于是此时开始S与S′有相同的缓存。

- 如果e' ≠ e，S′用e′交换e，此时开始S与S′有相同的缓存。(尽管S′此时不是简化调度，但是可以转换成前j+1步与SFF相同的简化调度)

FF调度分析

设j′是第一次在j+1个需求后S和S′采用不同的动作(意味着必须包括e,f的操作)，设g是在时刻j′需要的元素。

Case 3c: $g \neq e, f$. S 必须用 e交换.

我们让S'用f交换；此时S和S'具有完全相同的缓存。

FF调度分析

- 体现了交换论证的思想，如果S’含有“贪心”的性质，那么不比S 产生更多的交换。
- 这个定理说明可以存在一个**连续的**简化调度优化，变成最后的S_{FF}.

FF调度分析

- 定理4.13 S_{FF}比其他任何的调度S'不产生更多的交换，因此是最优的。

推广

- Belady的最优算法对于超高速缓存的性能提供了一个基准.
- 应用的局限性:
  - ✓ 需要知道所有的需求序列
  - ✓ 真实的情况：必须在不知道将来需求的情况下匆忙做出收回的决定

推广

- 经验上，实际应用的是最近最少使用原则（LRU: ）
- 建议从超高速缓存中收回最久以前被访问的页面（least recently used ）
- 把时间方向倒过来：过去的最久而不是将来的最远，恰好是Belady算法
- 实际应用中，一般频繁访问的是*刚刚被访问过的内容*

推广

- LIFO. 后进先出的替换算法.

- 定理. FF 是最好的离线(off line)高速缓存替换算法.

  - LRU is k-competitive. [Section 13.8]
  - LIFO is arbitrarily bad.

4.4 一个图的最短路径

- Edsger W. Dijkstra(1972, Turing Prize)
- 主要的研究工作

✓ 最早指出“goto“有害

✓ 首创结构化程序设计

✓ PV操作

✓ Algol60编译器

✓ 1956，有障碍物的两个地点之间找出最短路径---

一个图的最短路径

- 问题描述

- 从网络中一个点到另一个点的旅行，经过一系列的交叉点，基本问题是确定相应的图中结点之间的最短路径。

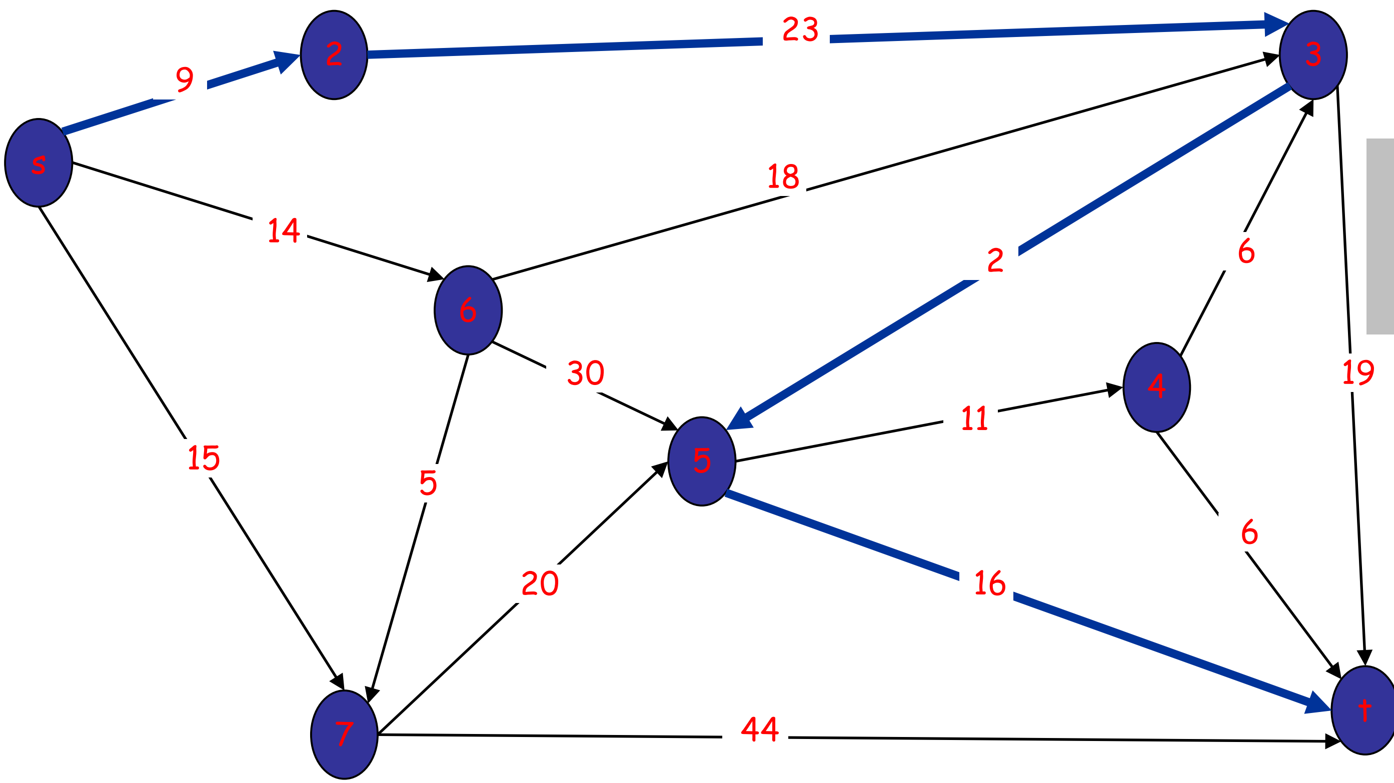
一个图的最短路径

- Shortest path network
  - Directed graph G = (V, E).
  - Source s, destination t.
  - Length ℓ_e = length of edge e.

- 最短路径问题: 寻找有向图中 s 到 t的最短路径

一个图的最短路径

最短路径长度是多少？



s-2-3-5-t
= 9 + 23 + 2 + 16
= 50.

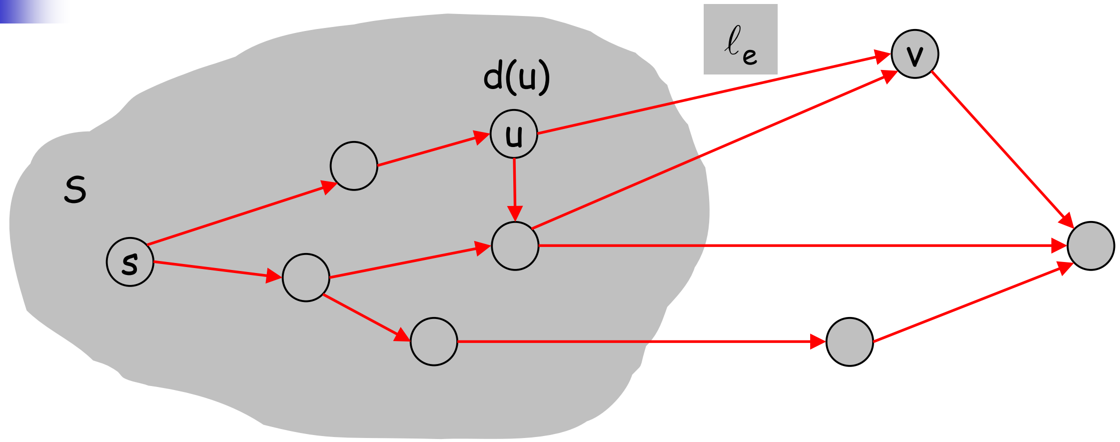
Dijkstra算法

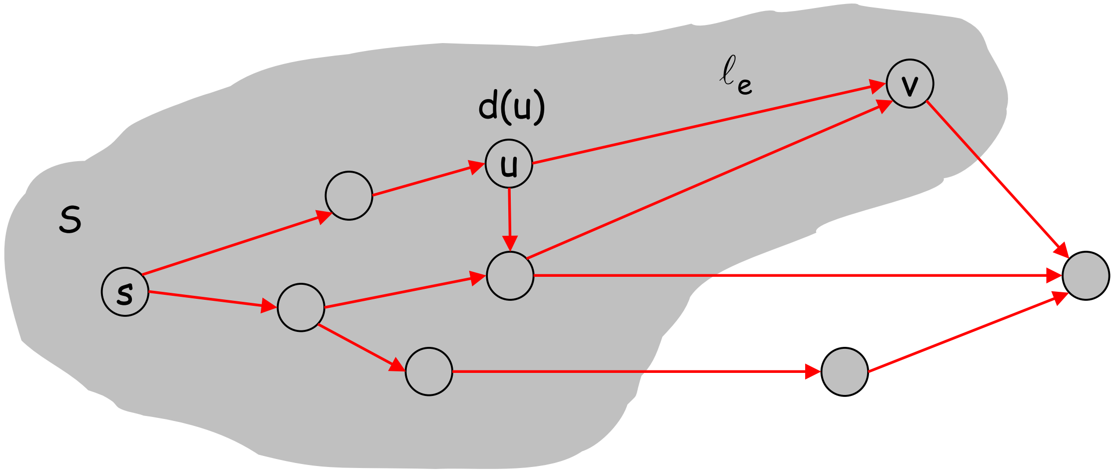
- 保持一个结点集S，如果源点s到u的最短路径一旦确定，那么u就加到S中。
- 最开始 S = { s }, d(s) = 0.
- 循环寻找结点v，使得 $\pi(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e$,

从已经探索过的某个结点u 到v的边 (u, v)

最小，然后把v 加入S, 并设 d(v) = π(v).

Dijikstra算法





Dijkstra算法

Dijkstra算法(G,l)

设S是被探查的结点集合

对每个S中的u,存储一个d(u);

初始S={s}且d(s)=0

While S!=V

选择一个结点不在S中的结点v,使得从S到v至少有一条边连接并且

$d'(v)=\min_{\{e=(u,v),\ u\ in\ S\}} d(u)+l_e$ 最小

将v加入S并且定义d(v)=d'(v)

Endwhile

分析算法

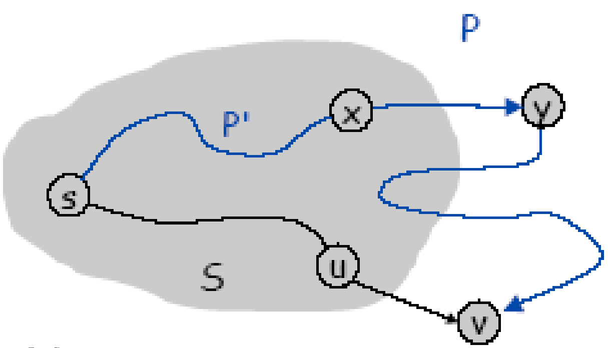
- 当Dijkstra算法加入一个结点v时，我们就得到了正确的到v的最短路径距离，这总是真的吗？

- 定理4.14 算法执行中任意一点的集合S, 对每个S中的u,路径P_u是最短的s-u路径。

分析算法

- Pf. 采用归纳法
- $|S| = 1$ 易证.
- 假设 $|S| = k \geq 1$命题成立。
  - Let v be next node added to S, and let u-v be the chosen edge.
  - The shortest s-u path plus (u, v) is an s-v path of length $\pi(v)$.
  - Consider any s-v path P. We'll see that it's no shorter than $\pi(v)$.
  - Let x-y be the first edge in P that leaves S, and let P' be the subpath to x.
  - P is already too long as soon as it leaves S.

$$\ell(P) \underset{\text{nonnegative weights}}{\geq} \ell(P') + \ell(x,y) \underset{\text{inductive hypothesis}}{\geq} d(x) + \ell(x,y) \underset{\text{defn of } \pi(y)}{\geq} \pi(y) \underset{\text{Dijkstra chose v instead of y}}{\geq} \pi(v)$$



算法实现

- 对于每个“无关顶点"，不做改变
- 对于“相邻顶点”，当v被加入“核心”时, 对于边e = (v, w), 更新临时最短距离

$$\pi(w) = \min\{\ \pi(w),\ \pi(v)+\ell_e\}.$$

- 提高效率的方法：采用**优先队列**来存放没有探索的结点，根据π(v)来排序.

算法分析

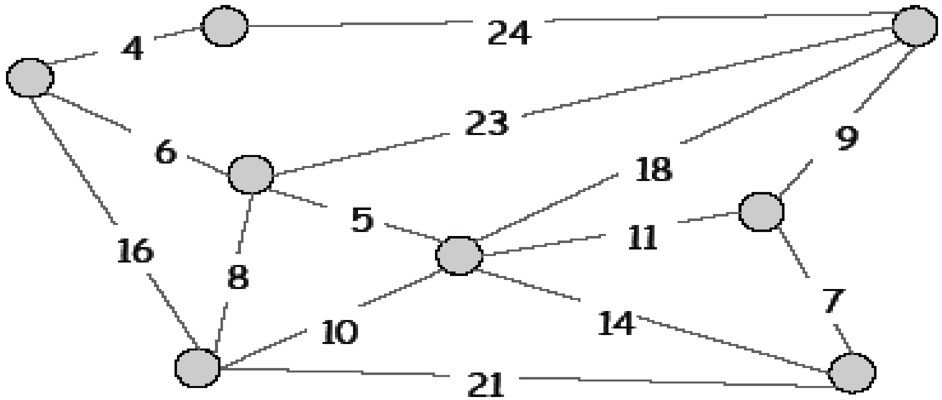
- 普通实现代价
  每次考虑增加一个结点;
  求到源点最小距离.
  复杂度：O(mn)
- 采用优先队列实现代价?
  O(m)次考虑边，n次ExtractMin
  m次ChangeKey;
  复杂度：O(mlogn)

4.5 最小生成树问题

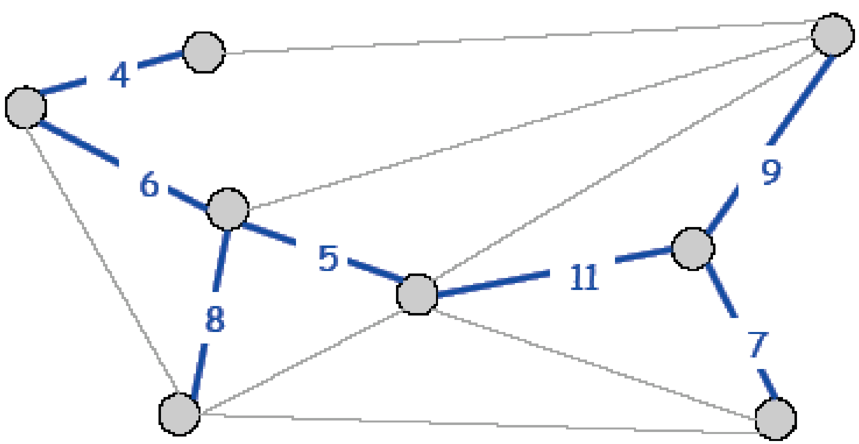
- 假设我们有位置集合$V=\{v_1,v_2,\ldots,v_n\}$,我们想建立一个通信网络，网络应该是连通的，我们希望尽可能便宜的建立它。
- 问题抽象：

对于确定的边$e=(v_i,v_j)$,存在一个正的费用C_e, 问题是对于图$G=(V,E)$,寻找 $T \subseteq E$ 使得图(V,T)是连通的，而且 $\sum_{e \in T} C_e$ 最小。

最小生成树问题



$G = (V, E)$



$T, \ \Sigma_{e \in T} c_e = 50$

应用场景

- MST(Minimum Spanning Tree) 是许多应用场景中的基本问题
  - Network design.
    - telephone, electrical, TV cable, computer, road
  - Approximation algorithms for NP-hard problems.
    - traveling salesperson problem, Steiner tree

应用场景

- 间接应用
  - max bottleneck paths
  - LDPC codes for error correction
  - learning features for real-time face verification
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
  - Cluster analysis.
  - …

最小生成树问题

- 命题4.16 设T是满足上面网络设计问题的最小费用解，那么(V,T)是一棵树。

最小生成树问题

- Exhaustive Search?

- Cayley's 定理. K_n一共存在 n^{n-2} 种支撑树.
- 由此可见，如果用穷举暴力搜索的方法是不可行的
- 需要有效的设计算法

贪心算法设计

- Kruskal's algorithm. 初始 T = ϕ. 边按照费用递增次序排列。通过不断插入边来建立一棵生成树：把边e插入T 只要不构成圈。

- Reverse-Delete algorithm(反向删除算法). 初始 T = E. 依照费用递减的次序开始删除边。当达到每条边e时(从最贵的边开始)，只要这样做不破坏当前图的连通性。

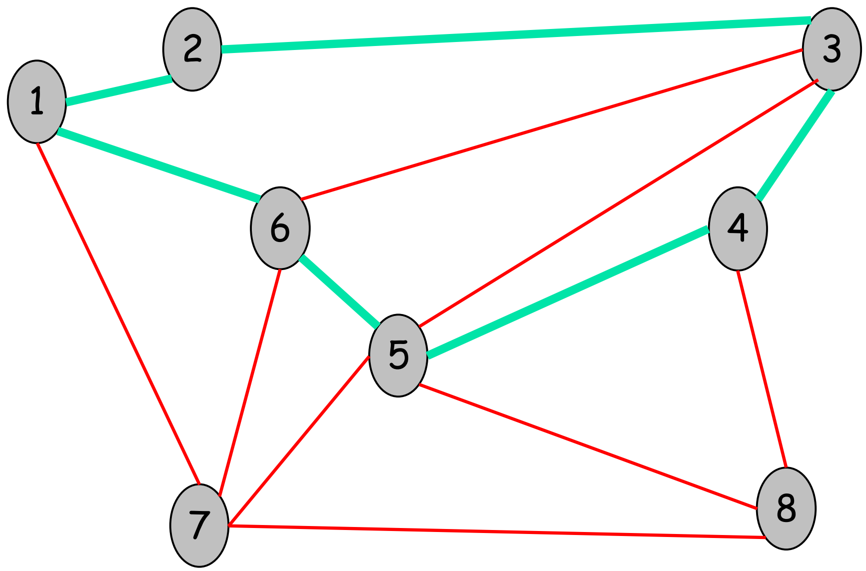
贪心算法设计

- Prim's algorithm. 初始S={s}，然后贪心选择的增长树T. 每步选择一端在T中，费用最小的边与 T连接。

殊途同归，三种算法都可以产生最小生成树(MST)！

圈和割

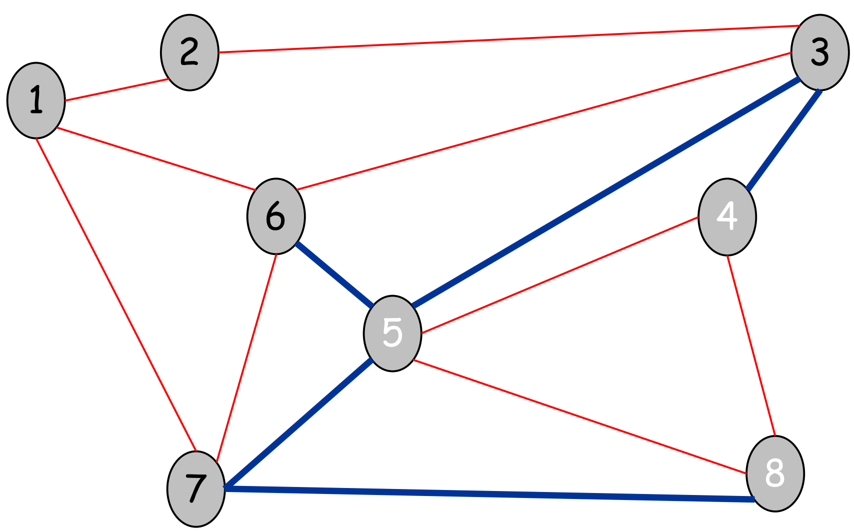
- 圈(Cycle). Set of edges the form a-b, b-c, c-d, …, y-z, z-a.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

圈和割

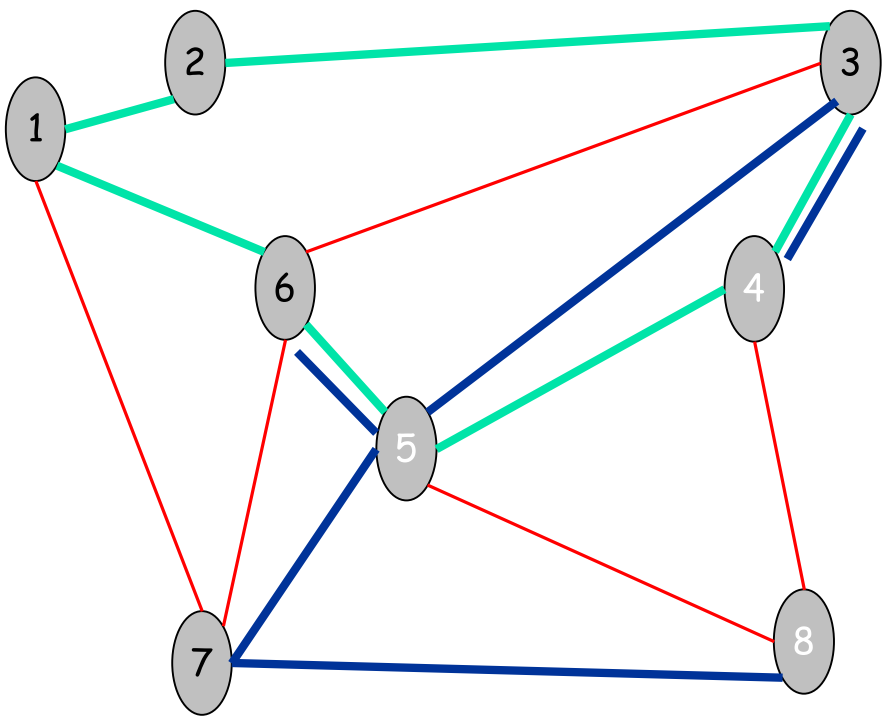
- 割 (Cutset) A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

圈和割

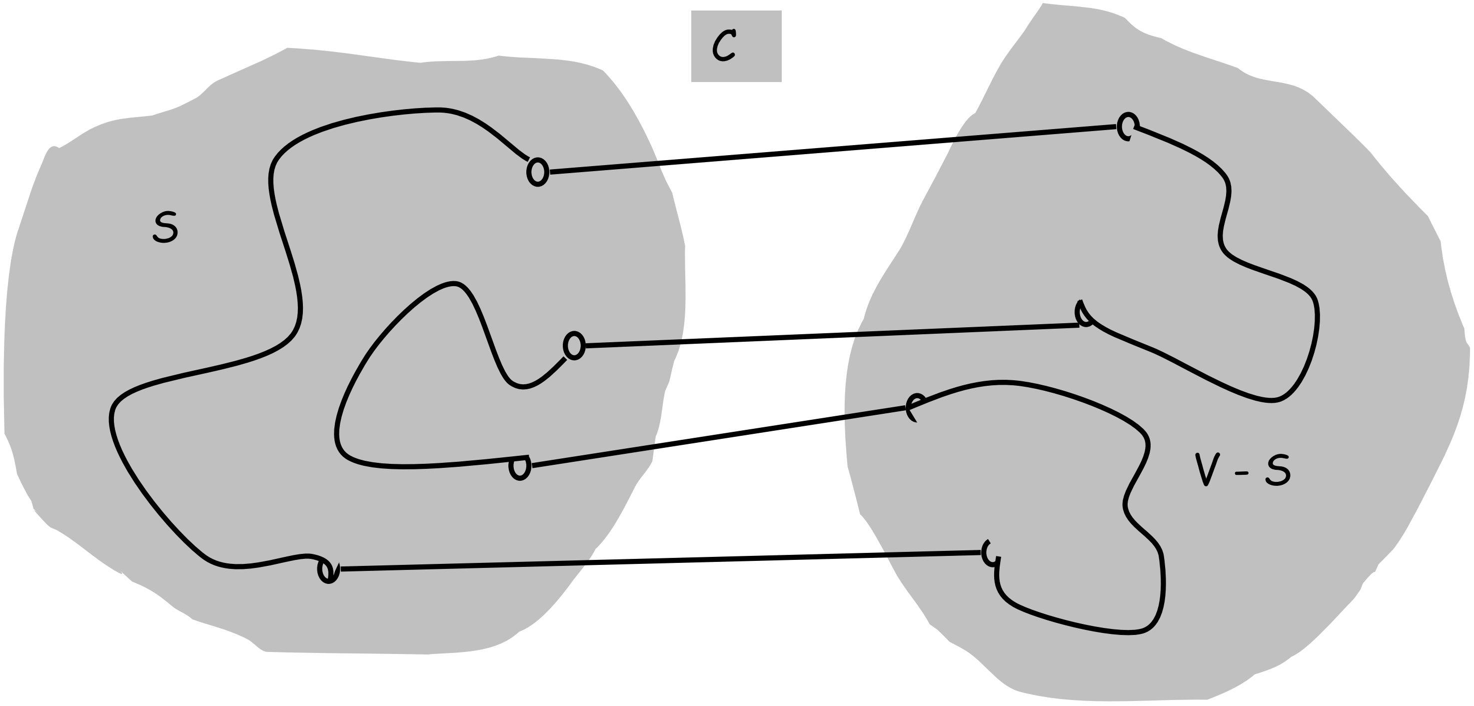
- 命题：一个圈和一个割有偶数条相交边。



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

圈和割

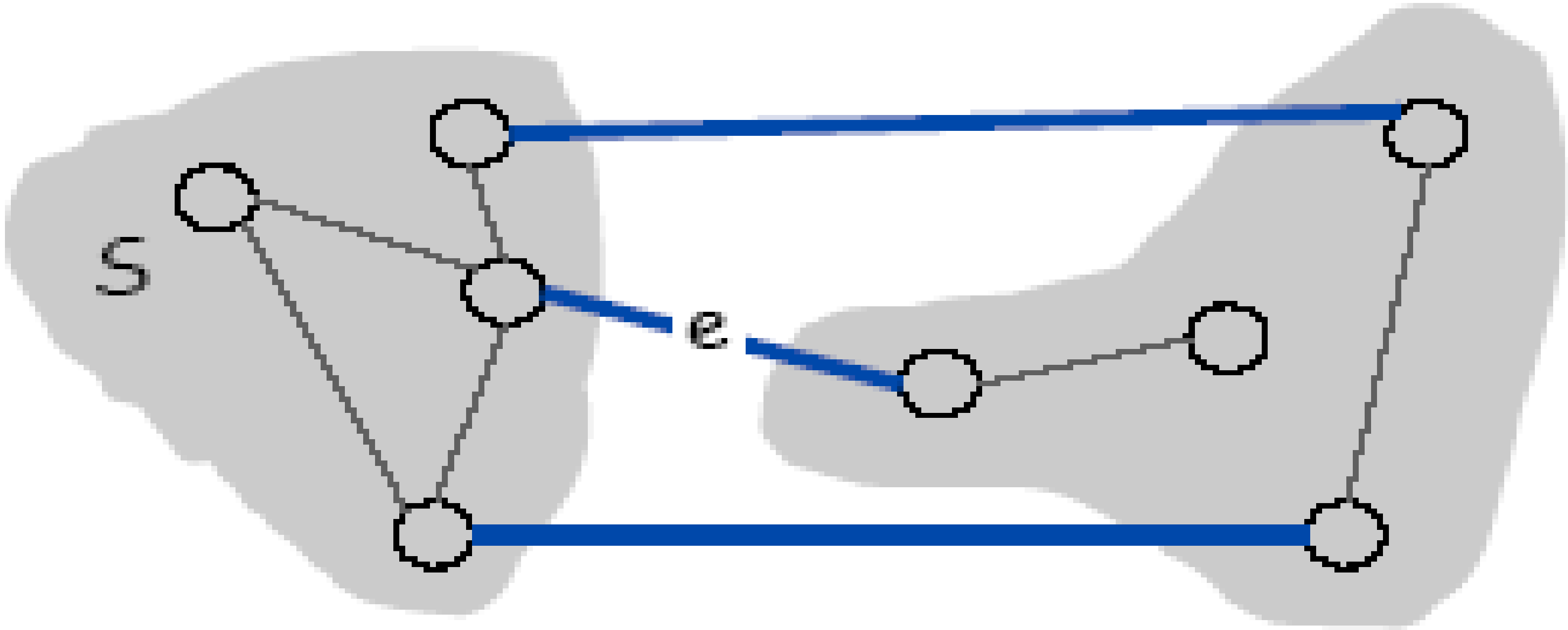
- 证明：



算法分析

- 什么时候在最小生成树中包含一条边是安全的？
- 定理4.17 假设所有边的费用都是不等的。令S是任意节点子集，$S \neq \phi, V$，令边e=(u,v)是一端在S中，另一端在V-S中的最小费用边。那么每棵最小生成树都包含边e.

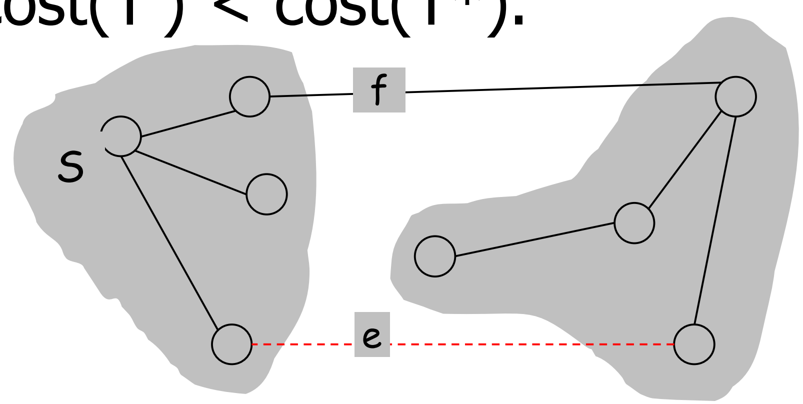
算法分析



e is in the MST

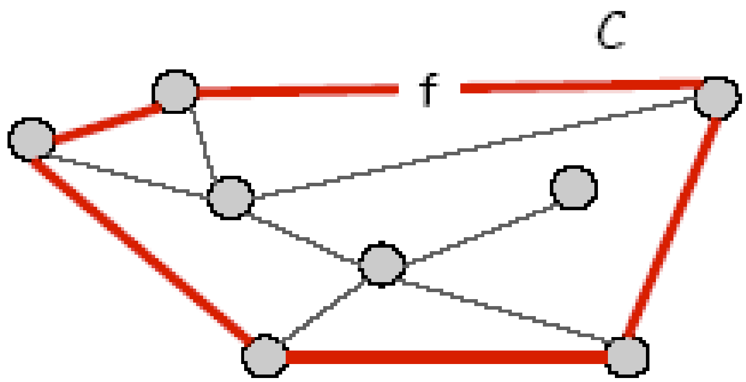
算法分析

- Pf. (采用交换论证)
  - 假设 e 不属于 T*(最小生成树).
  - 增加 e 到 T* 形成了一个圈C.
  - 边e 即属于圈 C，也属于与S对应的割 D ； $\Rightarrow$ 一定存在另一条边f，即属于C 也属于 D.
  - T' = T* $\cup$ { e } - { f } **也是一个生成树.**
  - 因为 $c_e < c_f$, cost(T') < cost(T*).
  - 导出矛盾. ▪



算法分析

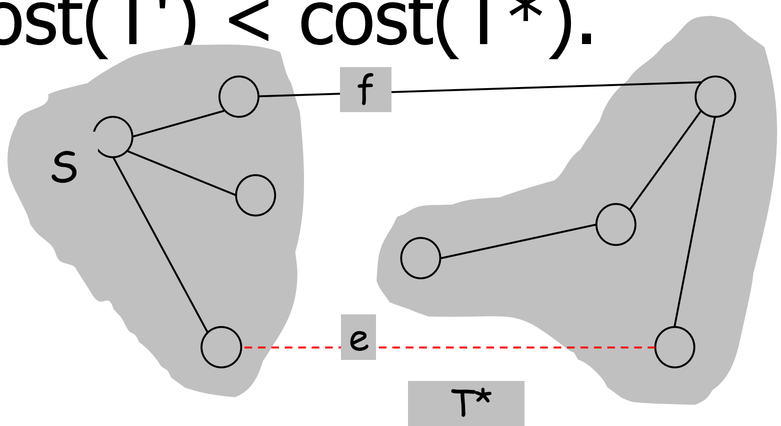
- 定理4.20 假设所有边的费用都是不等的。令C是G的一个圈，令边f=(v,w)是属于C的最贵的边。那么f不属于G的任何最小生成树。



f is not in the MST

算法分析

- 证明：(采用交换论证) 假设f属于T*.
  - 把f 从T* 中删除得到一个集合S .
  - f 即在圈 C 中，也在与S对应的割 D 中⇒
  - 一定存在另一条边e,即属于C 也属于 D.
  - T' = T* ∪ { e } - { f } 也是一个生成树.
  - 因为 $c_e < c_f$, cost(T') < cost(T*).
  - 导出矛盾. ■



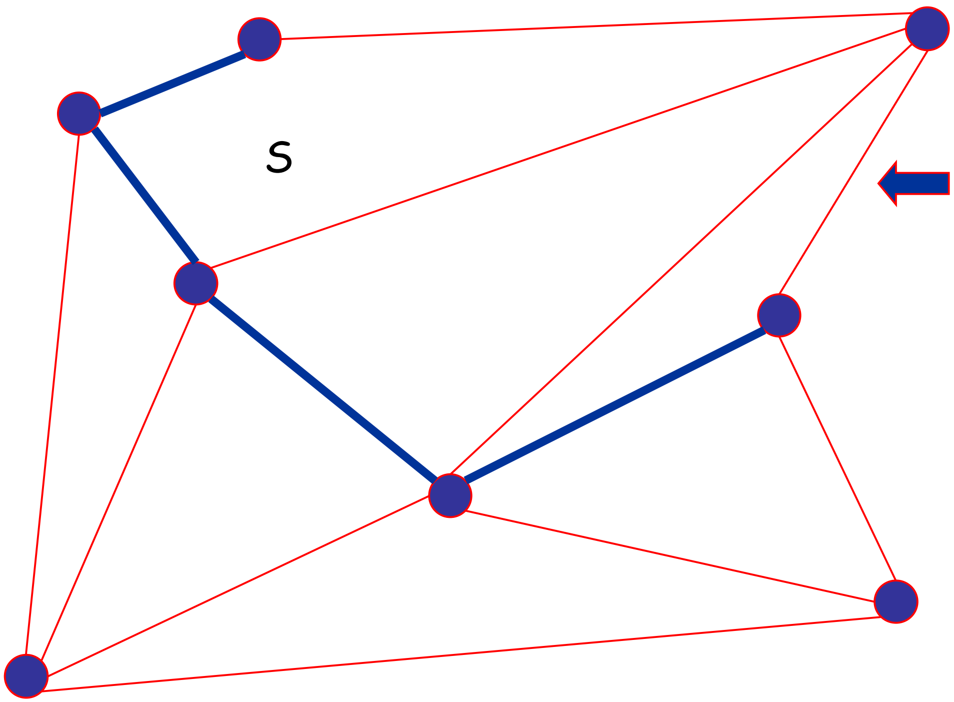
逆删除算法

- 定理4.21 逆删除算法产生G的一棵最小生成树。
- 证明：**最后算法输出(V,T)是连通的**。如果(V,T)包含一个圈C.考虑C上最贵的边e,它将是算法遇到的第一条边，移走应该不会破坏图的连通性，与逆删除算法的行为矛盾。

Prim 算法

- Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]
  - 初始 S = 任意结点.
  - 每次把与S对应的割中最小费用边加入 T, 同时把新的探索结点u 加入S.

Prim算法



Prim算法

- 定理4.19 Prim算法产生一棵最小生成树。
- 证明： 对于Prim算法，容易知道它只加了那些属于每棵最小生成树的边。根据定义，e是使得一个结点在S中而另一个结点在V-S中费用最低的边($\min_{e=(u,v),u\in S} C_e$),

所以根据割性质4.17，e在每一棵最小生成树中。

Prim算法

- 实现．采用与Dijkstra 类似的算法，使用优先队列.
  - 保持一些已经探索过的结点集合S.
  - 对于每一个没有探索过的结点v, cost a[v] = cost of cheapest edge v to a node in S.
  - 复杂度估计：　　采用优先队列（堆），O(m log n).

Prim算法

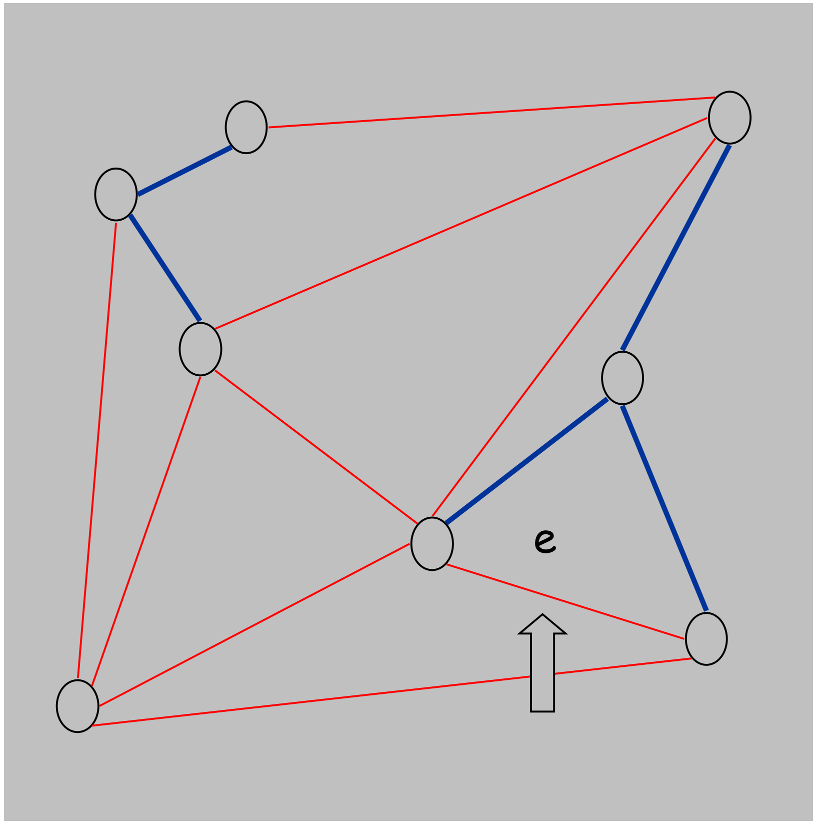
```
Prim(G, c) {
   foreach (v ∈ V) a[v] ← ∞
   Initialize an empty priority queue Q
   foreach (v ∈ V) insert v onto Q
   Initialize set of explored nodes S ← φ

   while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v) incident to u)
          if ((v ∉ S) and (c_e < a[v]))
             decrease priority a[v] to c_e
}
```

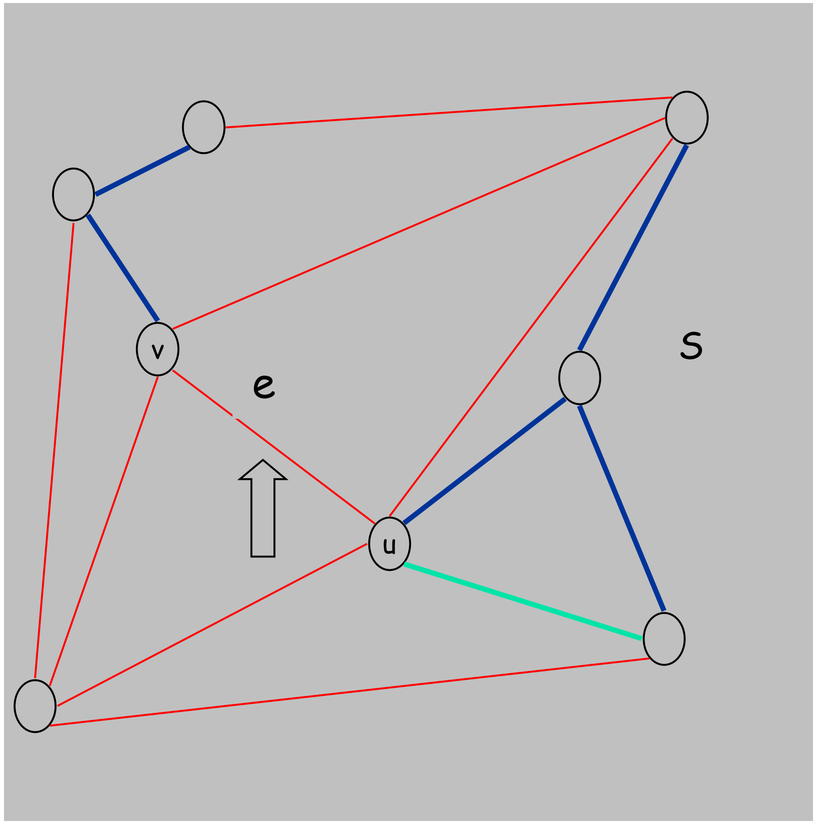
Kruskal算法

- Kruskal's algorithm. [Kruskal, 1956]
  - 边按照费用递增的顺序排序.
  - Case 1: 如果把边e加入T时构成了一个圈，那么跳过边e.
  - Case 2: 否则，把边 e = (u, v) 加入 T 中(根据割性质)， 其中 S 是与u连通的结点集合。

Kruskal算法



Case 1



Case 2

Kruskal算法

- 算法实现

  采用 union-find 数据结构.

  - 生成一个最小生成树T.
  - 保持每一个连通分支的集合.
  - 算法代价：O(m log n).

Kruskal算法

```
Kruskal(G, c) {
   Sort edges weights so that c₁ ≤ c₂ ≤ ... ≤ cₘ.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m
      (u,v) = eᵢ
      if (u and v are in different sets) {
         T ← T ∪ {eᵢ}
         merge the sets containing u and v
      }
   return T
}
```

讨论

排除所有边的费用不同的假设

- 原来假设所有边的费用不同，现在推广:
- 对于边费用相同的情形加一个*微小的扰动*，这样避免了“**平分**”的情形；对于原来不同的费用次序没有影响。注意到相关的算法基于边的费用的**相对关系**，所以同样产生对原始实例也是最优的MST.

推广

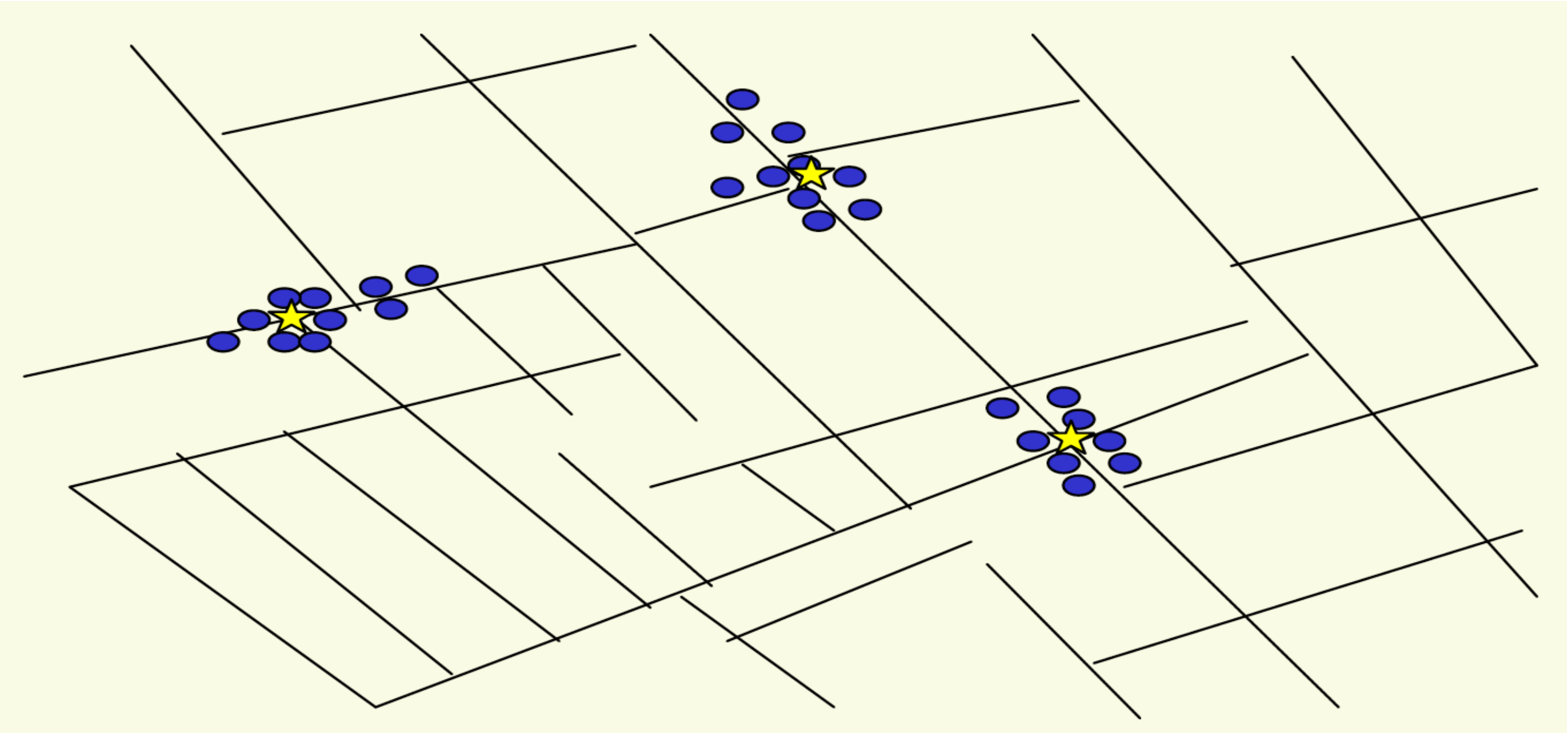
- 还关心点到点的距离
- 介意边上的拥塞，给定需要在结点对之间运输的交通量，需要寻找一棵生成树，其中每条边运输不超过确定的交通量。
- 真实的网络设计 **鲁棒性:**寻找集合上的最便宜的连通网络，删除一条边后仍保持畅通

4.7 聚类

- 问题

- 给定一组个体，比如说一些照片，文件，微生物等等，需要试图把它们分类或者组成相关的群体，产生聚类问题

聚类



Outbreak of cholera deaths in London in 1850s.
Reference: Nina Mishra, HP Labs

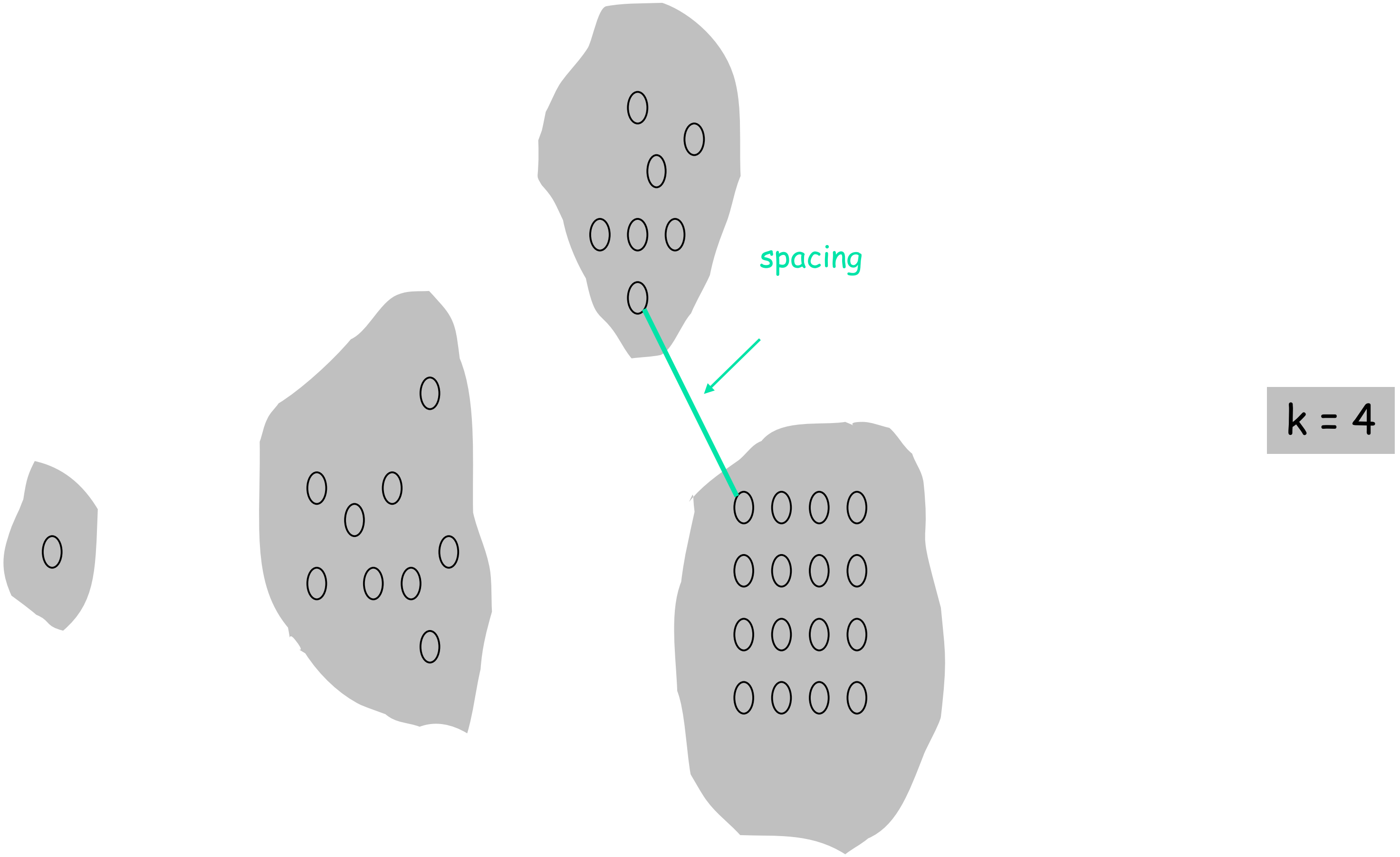
聚类

- 需要定义距离函数：描述关系的远近
- 基本问题. Divide into clusters so that points in different clusters are far apart.
  - Routing in mobile ad hoc networks.
  - Identify patterns in gene expression.
  - Document categorization for web search.
  - Similarity searching in medical image databases
  - Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

聚类

- 最大间隔聚类
- K聚类．把对象分成k个非空的部分.
- 距离函数:
  - $d(p_i, p_j) = 0$ iff $p_i = p_j$
  - $d(p_i, p_j) \geq 0$
  - $d(p_i, p_j) = d(p_j, p_i)$
- 间隔．处在不同聚类中的任何一对点之间的最小距离
- 对于某个k, 寻找具有最大可能间隔的 k聚类.

聚类



设计算法

- 按照距离$d(p_i,p_j)$增加的次序在点对之间增加边；
- 如果发现p_i与p_j已经属于同一个聚类，那么应该避免加这条边；因为它不会改变连通分支的集合；
- 每次添加一条横跨两个不同连通分支的边，就把两个对应的聚类合并。
- 单链聚类

设计算法

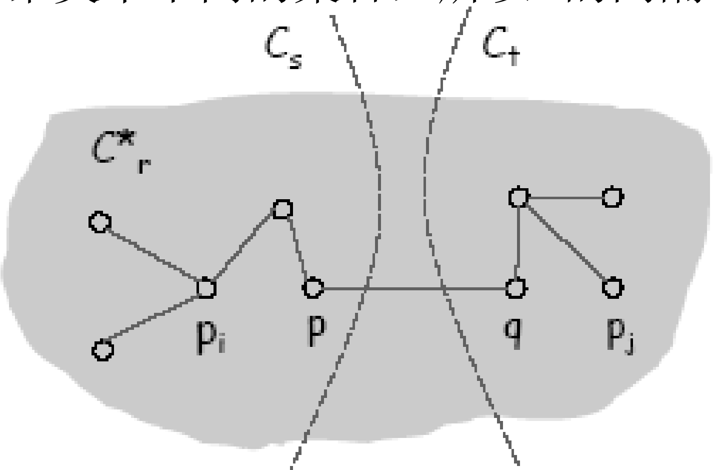
- 可以发现，实现的过程仿佛就是Kruskal最小生成树算法
- 也就是，运行Kruskal算法，但是就在它加最后的k-1条边之前停止。
- 等价于取整棵最小生成树T,然后删除k-1条最贵的边，并且定义k聚类是所得到的连通分支$C_1, C_2, \ldots, C_k$.

分析算法

- 定理4.26 由删除最小生成树T的k-1条最贵的边所构成的连通分支$C*: C_1^*, C_2^*, \ldots, C_k^*$组成一个最大间隔的k聚类。

分析算法

- 证明： 设C：$C_1, \ldots, C_k$是另外一个聚类
  - C* 的间隔就是第k-1条最贵边的边，长度为d*.
  - 设 p_i, p_j 在 C*的同一个聚类C^*_r中，但是却在C不同的聚类中，比如 C_s 与 C_t.
  - 由于pi,pj属于同一个连通分支C^*_r，所以一定在我们停止之前Kruskal算法添加了pi-pj路径上所有的边。特别的，这里pi-pj路径上每条边的长度至多是d*。
  - 因为相邻的p, q属于聚类中不同的集合，所以C的间隔 $\le$ d* ▪

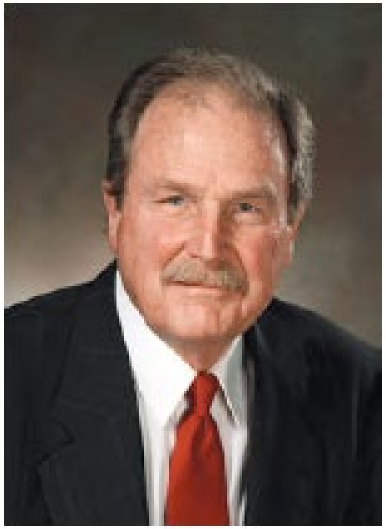


4.8 Huffman码与数据压缩

- 数据压缩，就是用最少的数码来表示信源所发出的信号，减少容纳给定消息集合或数据采样集合的信号空间

- 数字通信的基础

D. A. Huffman



问题的提出

- 数据压缩领域的基本问题之一：
- 字母用二进制表示，字母的频率不同，如何选择编码的方式，使得效率最高？

问题的提出

- 可变长编码模式(Samuel Morse)

  0对应一个点；

  1对应一个长划；

  一般的将比较频繁的字母对应到比较短的位串。

  问题：译码的**不确定**性；

  于是引入了暂停的符号

问题的提出

前缀码

- 对于一组字母S的前缀码是把每个字母 $x \in S$ 以下述方式映射到一个0，1序列的函数 γ，对于不同的 $x, y \in S$，序列 $\gamma(x)$ 不是序列$\gamma(y)$的前缀。

比如 $S = \{a, b, c, d, e\}$ ，定义如下的编码方式

$$\gamma_1(a) = 11, \gamma_1(b) = 01, \gamma_1(c) = 001, \gamma_1(d) = 10, \gamma_1(e) = 000$$

问题的提出

- 最优前缀码
- 假设对每一个字母 $x \in S$，文本中等于字母 x 的字母频率为 f_x ，字母总量为 n 。
  那么编码总的长度为：$\sum_{x \in S} n f_x |\gamma(x)| = n \sum_{x \in S} f_x |\gamma(x)|$

我们的目标使得每个字母的平均位数

$ABL(\gamma) = \sum_{x \in S} f_x |\gamma(x)|$ 最小。

问题的提出

- 1948 Claude Shannon 奠定了编码的信息论基础
- 宏观的结论：
- 如果 γ 是一种可译编码方案 (Instantaneous code), 那么

$$ABL(\gamma) >= H_r(f_1, f_2, \ldots)$$

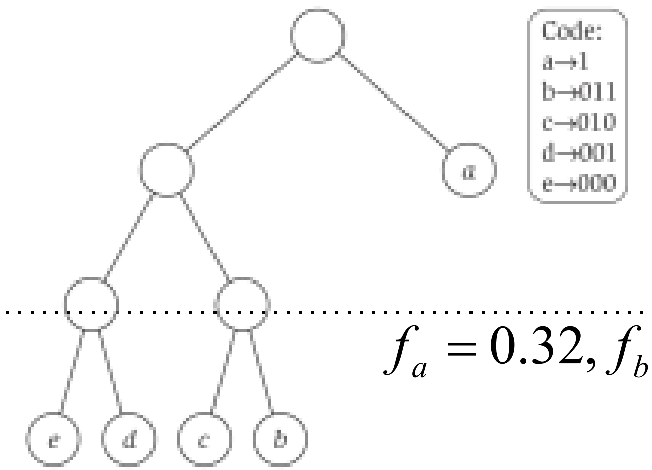
其中 $H_r(f_1, f_2, \ldots) = \sum_{x \in S} f_x \log_r \frac{1}{f_x}$

问题的提出

- 虽然给出了一个宏观的结论，但是当时并没有给出好的具体的编码算法。
- Shannon 和 Fano知道他们的方法不能给出最优前缀码，不知道如何不用蛮力搜索来计算最优编码。
- David Huffman 解决了这个问题，当时是一个研究生

设计算法

- 使用二叉树T表示前缀码，那么叶结点就是我们需要的编码方式。



$$f_a = 0.32, f_b = 0.25, f_c = 0.20, f_d = 0.18, f_e = 0.05$$

- 可以知道，从T构成的S的编码是一个前缀码

设计算法

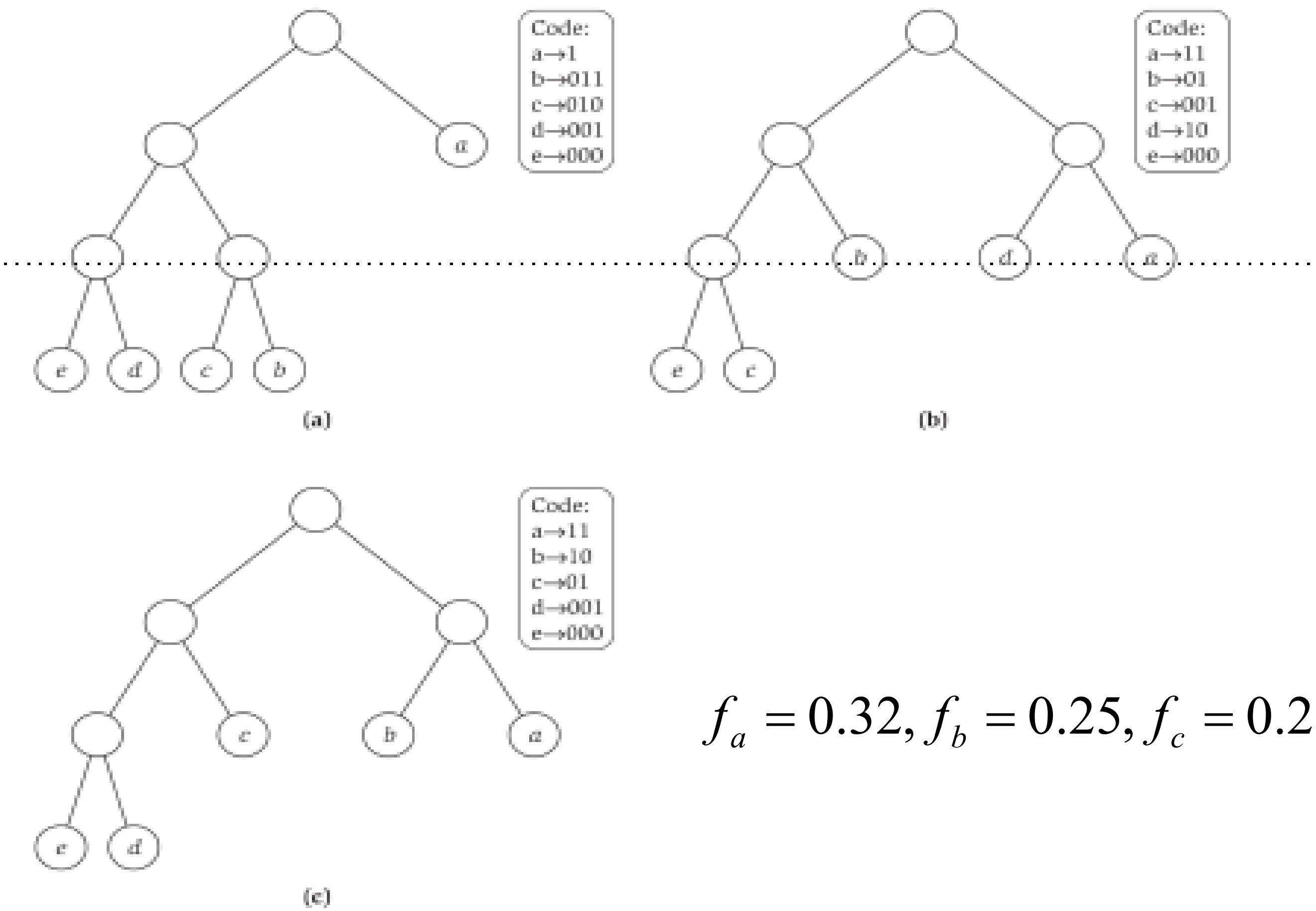
- 可以知道，

$$\gamma(x) = depth_T(x), ABL(T) = \sum_{x \in S} f_x \cdot depth_T(x)$$

- 现在我们关心的问题：与最优前缀码对应的二叉树应该是什么样子？

设计算法

- 几种不同的二叉树编码方式



$$f_a = 0.32, f_b = 0.25, f_c = 0.20, f_d = 0.18, f_e = 0.05$$

设计算法

- 第一个尝试：***自顶向下***的方法
- 尽可能“塞满”树叶，把字母表S分成两个集合S_1, S_2 ,试图划分尽可能使它们“接近均衡”，这样的编码方式称为Shannon-Fano码。
- 比如图(b)中，$f_a + f_d = 0.5, f_e + f_c = 0.25$

设计算法

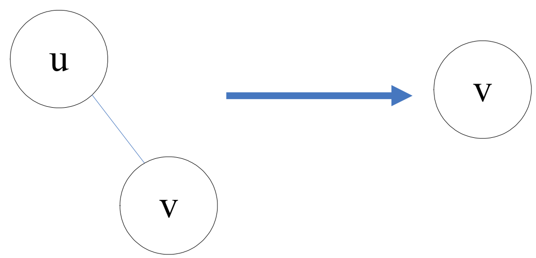
- 对于(b), $ABL(\gamma_b) = 0.25\times 3 + 0.25\times 2 + 0.5\times 2 = 2.25$
- 对于(c), $ABL(\gamma_c) = 0.23\times 3 + 0.77\times 2 = 2.23$

可见自顶向下的方法不是最优的

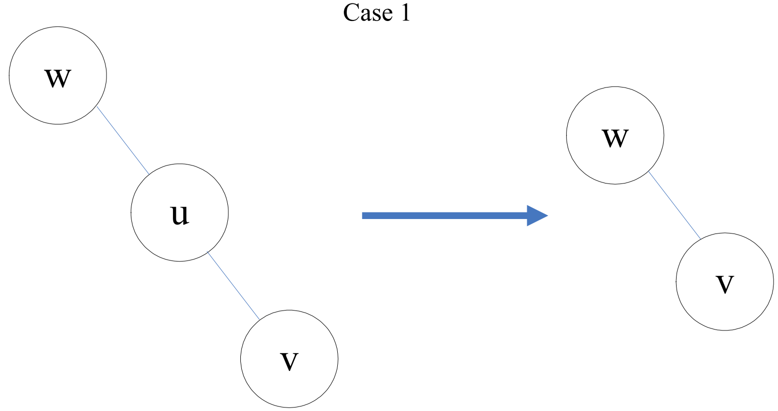
如果我们知道最优前缀码的结构，会是什么样子？设对应的二叉树是T*.

设计算法

- 与最优前缀码对应的二叉树是满的
  (*除了叶结点外每一个结点都有左右孩子*)



Case 1



Case 2

设计算法

- 我们的直观，频率大的字母，对应的编码(树高)短。
- 假设u与v是T*的树叶，使得depth(u)<depth(v). 树叶u对应于$y \in S$，树叶v对应于$z \in S$，那么$f_y \geq f_z$.

设计算法

- 证明：$ABL(T^*) = \sum_{x \in S} f_x \cdot depth(x)$ ，交换结点u与v,那么总和的改变是$(depth(v) - depth(u))(f_y - f_z)$,
  因为depth(u)<depth(v), 而这个改变需要是非负数，所以$f_y \geq f_z$.

事实上，对于图(b),(c), 发现交换b，d对应的叶结点，就可以获得更小的ABL.

设计算法

- 考虑T*中具有**最大深度**的一片树叶v, v有一个父亲u, 根据前述命题，T*是满的二叉树，所以v还有一个兄弟w.
  那么w是T*的一片树叶。

  证明：用反证法，如果w下面还有一片树叶w’, 那么depth(w’)>depth(v),与v具有最大深度相矛盾。

设计算法

- 存在一个与树T*对应的最优前缀码，其中**两个最低频率**的字母被指定为树叶，这两片树叶是T*中的兄弟。
- 据此我们有了自底向上的算法策略：设y*与z*是最低频率的两个字母，用其父亲节点“元字母”w替代y*,z*(这样字母表缩小), 递归的寻找前缀码。

设计算法

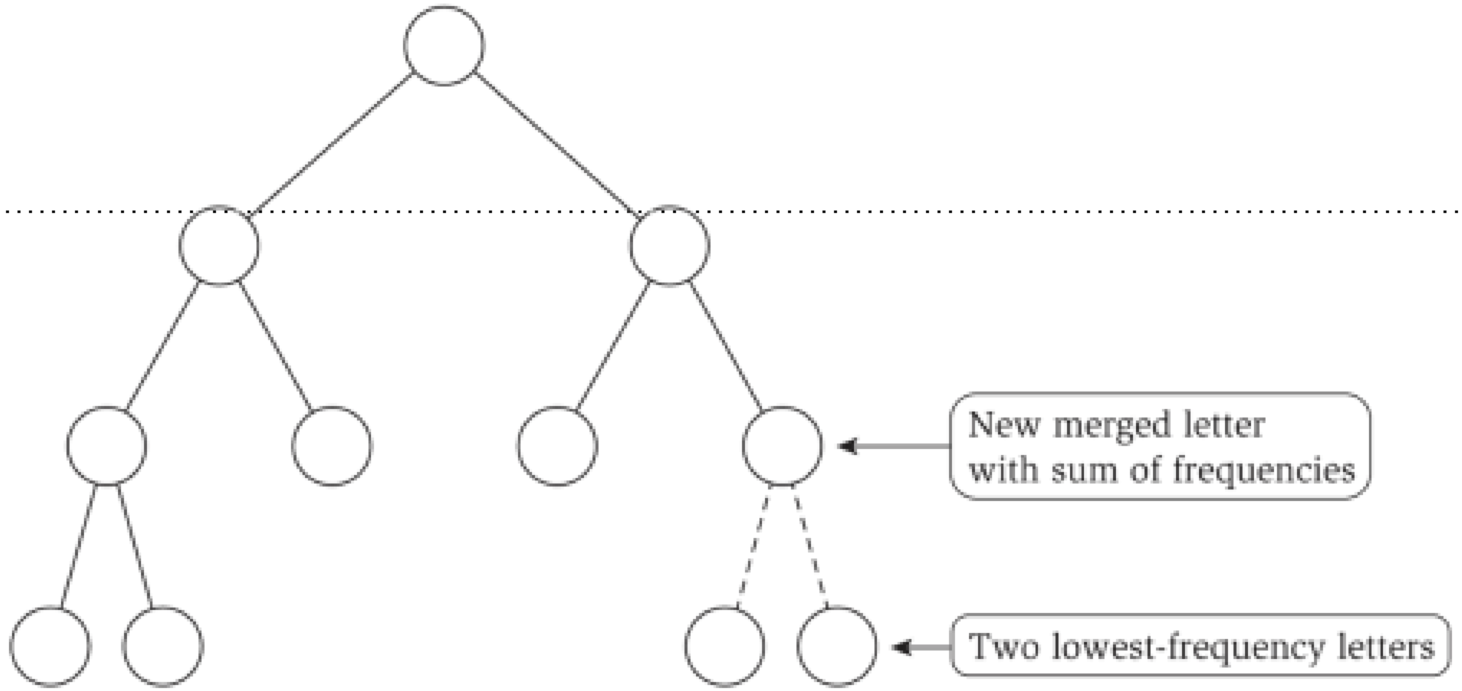


Figure 4.17 There is an optimal solution in which the two lowest-frequency letters label sibling leaves; deleting them and labeling their parent with a new letter having the combined frequency yields an instance with a smaller alphabet.

设计算法

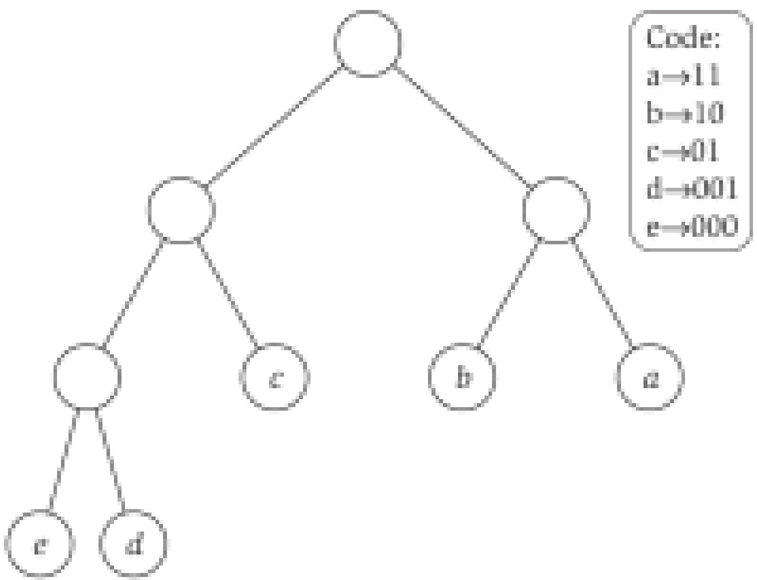
- 具体的算法步骤(Huffman算法):

```
To construct a prefix code for an alphabet S, with given frequencies:
  If S has two letters then
    Encode one letter using 0 and the other letter using 1
  Else
    Let y* and z* be the two lowest-frequency letters
    Form a new alphabet S' by deleting y* and z* and
      replacing them with a new letter ω of frequency f_{y*} + f_{z*}
    Recursively construct a prefix code γ' for S', with tree T'
    Define a prefix code for S as follows:
      Start with T'
      Take the leaf labeled ω and add two children below it
        labeled y* and z*
  Endif
```

设计算法

- S={a,b,c,d,e}, $f_a = 0.32, f_b = 0.25, f_c = 0.20, f_d = 0.18, f_e = 0.05$
- Step1 d,e 合并
- Step2 c, (de) 合并
- Step3 a,b 合并
- Step4 (ab), (c,(de))合并

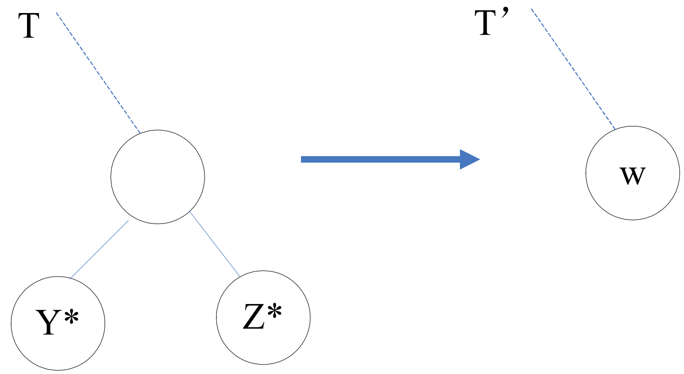
这样就得到了图(c)



(c)

分析算法

- T与T'



- 命题：$ABL(T') = ABL(T) - f_w$

- 定理：对于给定字母表得到的Huffman码就是最优前缀码。

分析算法

- 证明：**归纳步骤中用反证法**。假设贪心算法产生的树T不是最优的，存在树Z, 使得ABL(Z)<ABL(T),根据前面的命题，存在这样一棵树Z,其中y*与z*(频率最低的两个字母)是兄弟。类似的，我们可以得到Z’, 满足$ABL(Z') = ABL(Z) - f_w$. 于是就得到 $ABL(Z') < ABL(T')$，与作为S’前缀码的T’最优性相矛盾。

分析算法

- **实现与运行时间**
- 一般的代价

  识别最低频率的字母在O(k)时间内，

  迭代求和总的代价为O(k^2)
- 采用优先队列(用堆实现)

  每次插入和最小元素的取出调整时间O(logk),

  总运行时间O(klogk)

推广

- 理论上编码最少位数是熵值；

  “半位”的概念：算术编码
- 缺点：对文本的变化不能作出改变；

---自适应压缩模式
- 研究的方向：压缩技术的能力，计算代价之间找到一个合适的平衡点。

  LZW, grammar based coding, etc