二项堆、斐波那契堆、贪心

闾泽军整理

2015 年 11 月 15 日

1 回顾

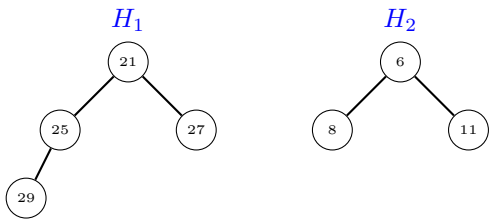
上一节我们从探究贪心和动态规划的关系，从最短路的动态规划算法转到了 Dijstra 算法，原先只是要求没有负圈，可以做动态规划。如果进一步加强条件，要求没有负的边，可以做 Dijstra 算法。
而关于 Dijstra 算法，其数据结构非常重要，尤其斐波那契堆这种数据结构是专门为 Dijstra 算法发明出来的。关于斐波那契堆还有一方面是其采用了均摊分析的方法。

首先来看下做这个数据结构的时候，最基本的出发点是什么。总体来说我们是要做一个优先队列，因为我们在 Dijstra 算法中，总是要在集合中找最小数。优先队列可以理解为，在队列中一个数越小就应该放在前面，所以实现这种数据结构，我们最直观简单的是开辟一个数组，但是存在一些问题，比如从 N 个数中找最小可能需

要从头到尾找一遍，即使预先做好排序，在插入时也会非常耗时。所以基于这些问题，提出了二叉堆。从数组到二叉堆的转变，对于原先严格的排序现在放松要求，二叉堆中不要求完全 n 个数有序，只要部分有序即可，只要一个节点比其儿子节点小就行，称为堆序。但是两个堆在合并时又比较麻烦，我们继续做思路上的转变，可以继续放松要求，不一定非得一棵树，如果可以有多棵数，两个堆的合并直接放一起就行。
基本思想：可以放松，但是不要太多，可以有多棵数，树的数目不要太多。

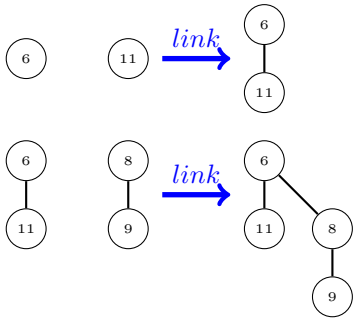
2 二项堆

树的数目不能太多，因为我们毕竟要找所有树中最小的数，每棵树最小都在根节点, 就直接把每个树的根节点比较下，显然如果有 n 棵树的话，就和数组没啥区别，所以要限制数的数目。图示为两个堆树。



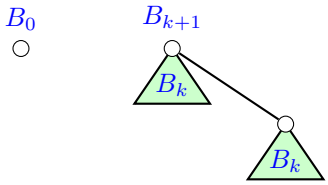
2.1 如何控制数的数目：合并树

为了控制数的数目不能太多，加了一个合并的操作。如图所示，每次合并时找同等级的树，合并的规则和堆合并的规则一样，合并完父节点的数比子节点的数都要小, 下图分别为一级的数和二级的树的合并操作。



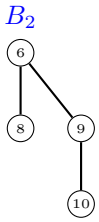
2.2 二项树及其性质

二项树是一个递归的定义，只有一个节点的为 B_0, 两个 B_k 等级的堆树合并为一个等级为 B_{k+1} 的堆树。如图：



我们通过一些直观的例子来理解下二项树

存在以下性质：（1）每棵树 B_k 的节点个数为 2^k （2）树的高度为 k。（3）从左往右儿子分别分 $B_0, B_1, B_2 ... B_k$，非常规则的一棵树可以观察到：对于每

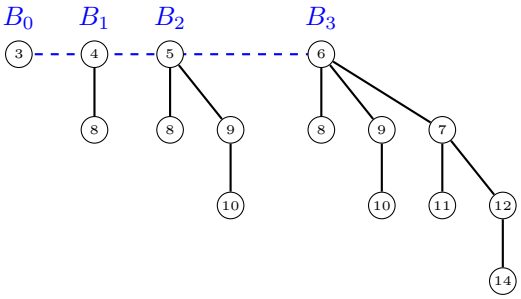


颗数每层的个数存在以下规则

$$B_0 : 1$$
$$B_1 : 11$$
$$B_2 : 121$$
$$B_3 : 1331$$
$$B_4 : 14641$$

刚好为二项式的系数，所以将这种结构称为二项树。那么我们一直强调控制树的个数，只要存在等级相同的树将其合并，那么最终形成的森林里，每个等级的数只有一个，如图例子中，B_0, B_1, B_2, B_3 各有一棵。

我们对两个堆树合并时只需要简单把两个堆树加起来，但是对于下图的例子，



加起来有两个 B_1，按照我们的规则最多只能有一个等级的树。就需要把两个 B_1 合并成一棵 B_2, 合并完之后又出现两棵 B_2, 再次合并两棵 B_2 成 B_3, 以此最终合成一棵 B_4。

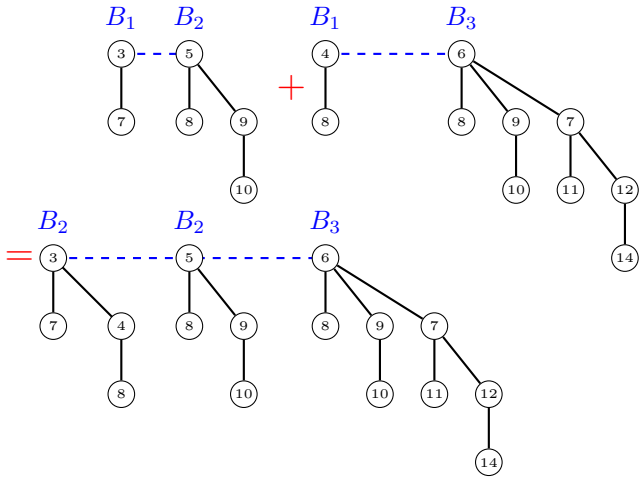


Figure 1: Consolidating two B_1 trees into a B_2 tree

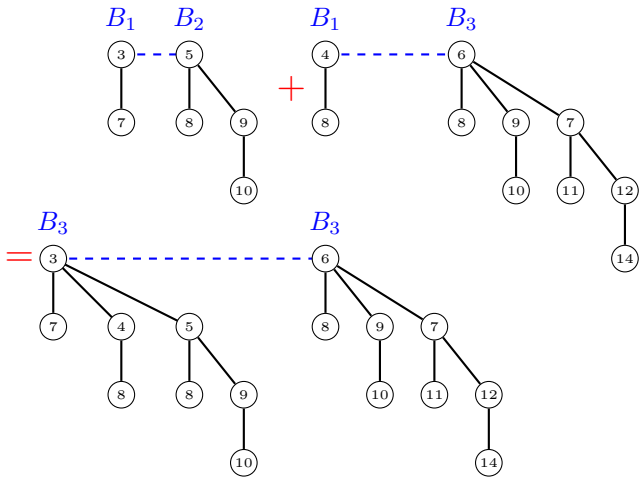
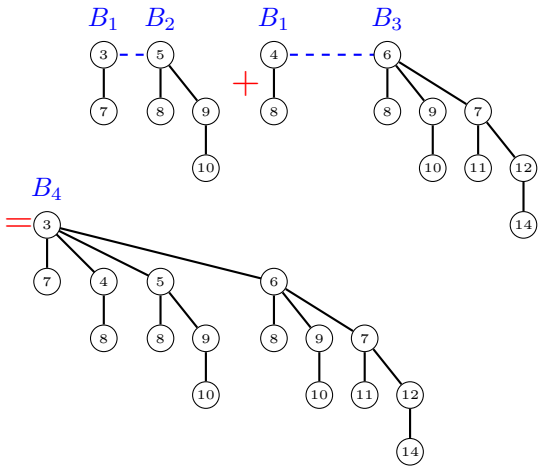


Figure 2: Consolidating two B_2 trees into a B_3 tree

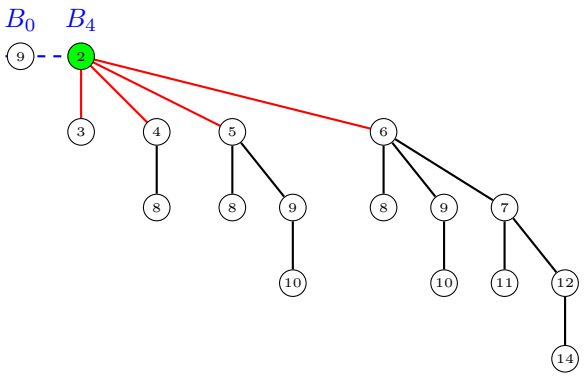


再次回顾下插入操作，只需要单建一棵树，这棵树只有一个节点既 B_0, 然后合并即可。

那么在森林中找最小，先看每棵树的根，如图，节点值为 2 的最小，那么就把这个最小的取走，儿子都独立成一棵树，然后继续合并。

ExtractMin()

1: find min of all roots;
2: remove the min node;
3: **while** there are two B_k trees for some k **do**
4: link them together into one B_{k+1} tree;
5: **end while**



2.3 均摊分析

上一节提到，二项树在插入操作只需要 $O(logn)$ 的时间，因为插入时考虑对同级的合并，因为最多有 log n 棵数。取最小一样，去掉最小之后，其儿子独立成一棵树，最多有 $logn$ 棵数，合并最多需要 $O(logn)$ 的时间。

但是对于时间复杂度的分析不是特别准确，实际上，插入操作只用了 $O(1)$ 的

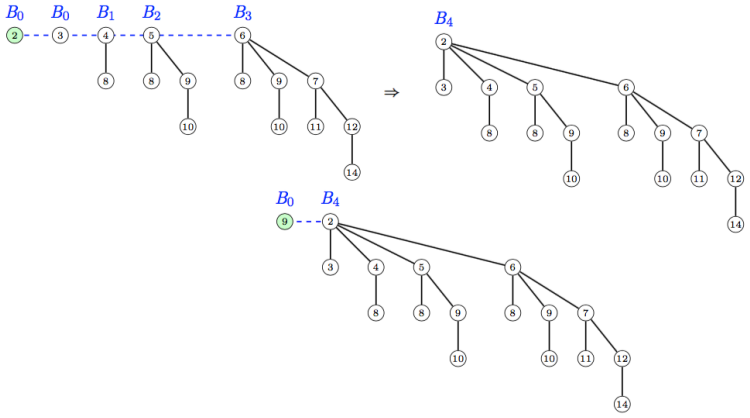
Operation	Linked List	Binary Heap	Binomial Heap
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$
EXTRACTMIN	$O(n)$	$O(\log n)$	$O(\log n)$
DECREASEKEY	$O(1)$	$O(\log n)$	$O(\log n)$
UNION	$O(1)$	$O(n)$	$O(\log n)$

时间 (常数级的操作，几次就可以做到)，这就是设计这个二项堆的目的，合并也是 O(1) 的时间。这种复杂度可以通过均摊分析来得到。

Operation	Linked List	Binary Heap	Binomial Heap	Binomial Heap *
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
EXTRACTMIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASEKEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
UNION	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$

* amortized cost

那么什么是均摊分析？以插入操作为例，一个插入操作可能会花费非常长的时间 O(log n), 如图所示，插入一个值为 2 的节点，即 B_1 大小的树，合并形成一个 B_2 的树，继续合并最终形成一个 B_4 的树。这次插入操作挺耗时间的，但是我们注意一下，这么一次很长的插入操作之后，以后的插入操作就会非常省事。一次插入操作之后合成一个很大的树了，那么之后再次插入就直接把 B_1 加入进去就可。所以，过去我们都是对单次操作做分析，而我们现在观察一串操作。均摊分析的思想就是，某次操作可能花费了很长的时间，针对本



例为 O(log n), 但是之后的操作可能就只需要 O(1),O(2)。所谓均摊就是对一串操作进行平均。下面我们用平均的思路再次看下插入操作

INSERT(x)

1: Create a B_0 tree for x;
2: **while** there are two B_k trees for some k **do**
3: link them together into one B_{k+1} tree;
4: **end while**

算法中第一句话花费 O(1) 的时间，While 循环几次花几次时间，用 w 表时 while 循环的此时，插入操作真实的时间为 1+w 的时间。然后我们考虑一个量 Φ, 称为势函数（*#tree* 或树的数目）。在插入的过程当中，树的数目增加了 1，whlie 每循环一次减少一棵树，总减少 w 棵树，真实时间为 $1+w = 1+\Phi$ 的减小值。类似的我们看下取最小

EXTRACTMIN()

1: find min of all roots;
2: remove the min node;
3: **while** there are two B_k trees for some k **do**

4: link them together into one B_{k+1} tree;
5: **end while**

把所有树的根节点拿来找最小，把最小的去掉，儿子单独成一棵树，检查将相同等级的树合并。分析时间，用 w 表示循环的次数，d 表示取掉的最小的那个节点的儿子个数，我们同样考虑势函数 $\Phi(\#tree$ 或树的数目)，在这个过程中，增加了 d，减小了 w。真实的时间为 d+w=d+Φ 的减小值。

现在我们考虑一串的操作，包括 n 次插入操作和 m 次寻找最小的操作，总体时间花费了 n*(1+decrease $\#tree$)+m*(d+decrease $\#tree$)，总时间为 n+m*d+ 所有的 decrease $\#tree$,d 是值删了节点会有多少节点，一个树最多有 log n 个儿子，所以总体时间为 n+mlogn+ 所有减少的树。首先注意，所有树减少的数目肯定小于增加的树的个数。那么整个树的个数增加多少？一次插入操作增加一棵，一次查最小最多增加有 log n 个儿子变成树，所以为 n+mlogn。所以总时间 <= 2(n+mlogn), 所以平均下来，每次插入操作花费 2 个单位的时间，每次找最小花费 2logn 的时间。所以我们说均摊下来，插入操作只花费了 O(1) 的时间，找做小值花费了 O(log n) 的时间

均摊分析的定义，我们考虑一系列操作,n_1 次的操作 1，n_2 次的操作 2..., 如果总共花费的时间为 $O(n_1T_1 + n_1T_2, ...)$，那么我们就说操作 1 花费了平均 T_1 的时间，操作 2 花费了平均 T_2 的时间。再次回顾下一次很耗时 logn 的插入操作之后，下面有 logn 级别的次数的插入操作会很小, 每次几乎就是 2 个单位时间耗时。

3 斐波那契堆

但是二项堆中还有一个操作 DecreaseKey(修改指定的一个值), 如图把 B_4 树中的 17 改为 1，1 比 16 小，要调整一次，比 13 还小再调整一次，再继续和 5 调整。最后树有多高花费多少时间，即 O(log n) 的时间。那么 DecraseKey 操作能不能也快一些。确实可以，下面我们看以下斐波那契堆这种数据结构。那么斐波那契堆是怎么实现的呢？之前我们总是和父节点比，和父节点换，那么现在我们就不换了，单独把这个节点砍掉，称为一棵树。这种数据结构是 1986 年 Robert Tarjan 提出来的，我们看下他的基本思想，同之前一样，我们要放松下要求，一旦我们的堆序被违背之后，我们不做调整，直接砍掉。但是放松成什么样呢？过去我们的树虽然是斜着的，但是非常完美，是规整的二项式系数排列的树。现在我们不要求那么规整，但是也要和原来差不多。二项树的节点数刚好为 2^n, 现在斐波那契堆树的节点个数为 $2^{1.618}$。

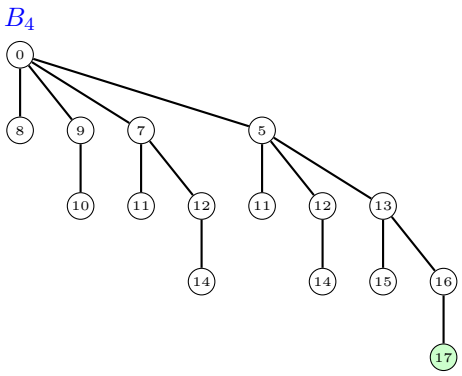


Figure 3: DecreaseKey: 17 to 1

3.1 斐波那契堆的 Descreasekey 操作

下面看下斐波那契堆是如何操作 Descreasekey 的。

DecreaseKey(v, x)

1: $key(v) = x$;
2: **if** heap order is violated **then**
3: $u = v's$ parent;
4: cut subtree rooted at node v;
5: **while** u is marked **do**
6: cut subtree rooted at node u, and insert it into the root list;
7: unmark u;
8: $u = u's$ parent;
9: **end while**
10: mark u;
11: **end if**

首先，把要求修改的那个节点 v 的数修改为指定的数，v 的父亲节点为 u，如果违反序，就把 v 砍掉。第五行，如果 u 已经被标记，干脆把 u 也砍掉也单独成一棵树。标记是说明该节点已经失去了一个儿子，最终的目标是每个节点最多只能丢掉一个儿子，这样树的形状不会变化的太厉害。如图是原始的斐波那契堆（从二项堆过来的）：

现在想把 19 变成 3，比父节点小砍掉，那么 5 失掉了一个儿子，标记为黄色。

现在在把 15 变成 2，2 比 13 小砍掉，13 做上标记。

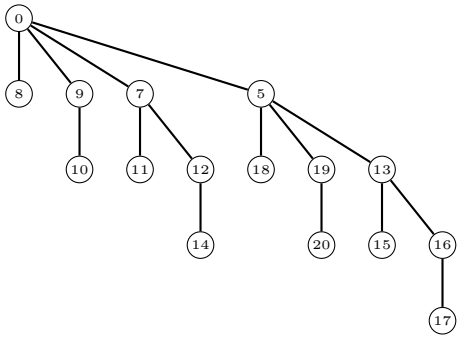


Figure 4: The original Fibonacci heap

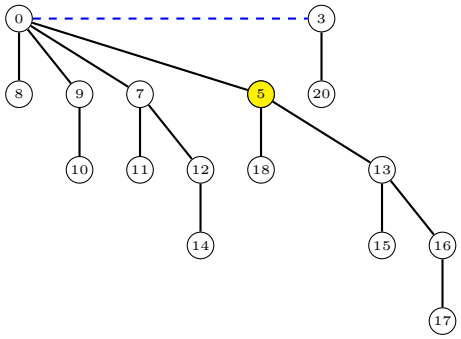


Figure 5: DecreaseKey: 19 to 3

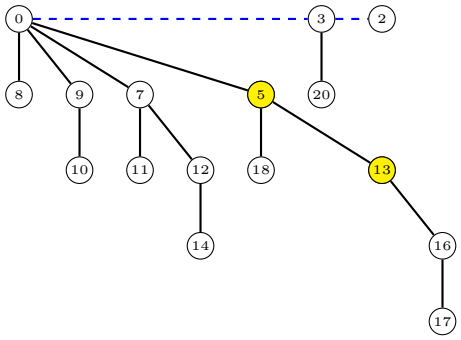


Figure 6: DecreaseKey: 15 to 2

再把 12 变为 8，满足规则不做操作。

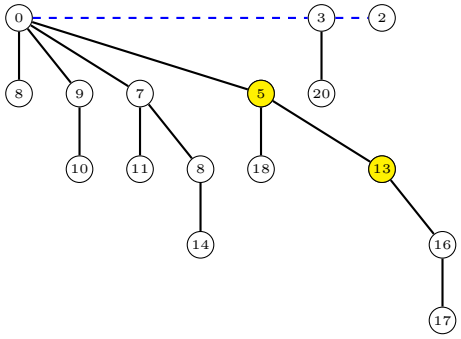


Figure 7: DecreaseKey: 12 to 8

再把 14 变为 1，比 8 小砍掉，8 做上标记。

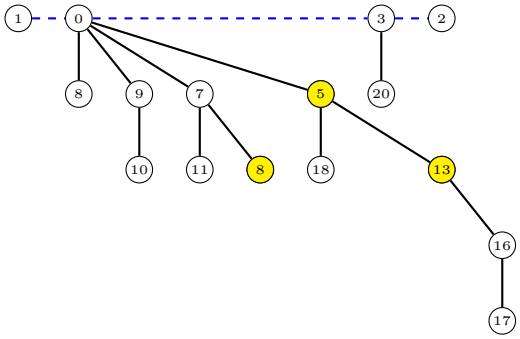


Figure 8: DecreaseKey: 14 to 1

但是当把 16 变成 9 的时候，比 13 小要砍掉，但是 13 已经被标记了，所以把 13 也砍掉，5 也被标记过，所以 5 也要被砍掉，所以最终结果为：

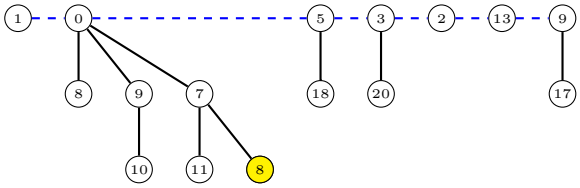


Figure 9: DecreaseKey: 16 to 9

3.2 斐波那契堆的插入操作

斐波那契堆对插入操作也做了修改。如图：

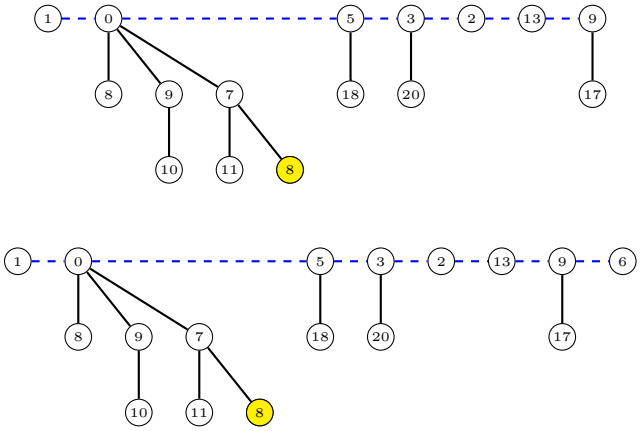


Figure 10: INSERT(6): creating a new tree, and insert it into the root list

比如新插入一个元素 6，直接放进去就行，不像之前还需要把同级的合并，把所有的归并操作放到 ExtractMin 中去。这个技巧称为"Being Lazy！"，举个例子，大家书架放书，上课之前取了一本书，回来之后插入进去，但是有些同学要把书按照字母序，但是这样很累。Being Lazy 就是插入就插入，不用管它，过一周之后就自动恢复漂亮的序了。

3.3 ExtractMin 取所有节点最小

EXTRACTMIN()

1: find min of all root nodes;
2: remove the min node;
3: **while** there are two roots u and v of the same degree **do**
4: consolidate the two trees together;
5: **end while**

和二项树结构找最小的操作是一样的。

3.4 对斐波那契堆做均摊分析：这里的分析有些问题

分析看每个操作单独花费的时间

观察 DecraseKey 单次操作实际到底花费多长时间，观察 3.1 中 DecraseKey 算

法，前 4 行花费为 O(1) 时间，Whlie 循环中也是单位时间，真实时间为 1+w,w 为 whlie 循环的数目。
我们考察下面这个量 $\Phi = \#trees + 2\#marks$，表示有几棵树以及几个节点丢失了儿子。在这个过程当中，每循环一次新增了 1 棵树，共 w 棵，新增了一个标记，所以 Φ 增量为:w+2（暂时认为为 3 ）。暂且认为总运行时间为：1+w=1+Φ 减量，但是如果我们可以把 w 均摊到其他操作上，那么 DecraseKey 操作只需要花费 O(1) 的时间。

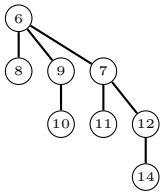
再看下 ExtractMin 花费多少时间，观察 3.3 中的算法，第一行花费 O(logn) 时间，第二行花费 O(logn) 时间，While 循环花费 w 时间。真实运行时间为 d+w, 其中 d 表示有多少儿子升级，放到根里面。我们仍考察 $\Phi = \#trees + 2\#marks$，每次增加 d 棵树，每次 whlie 循环就减少一棵，共 w 棵。运行时间：d+w=d+decrease in Φ。

再考虑插入操作，Insert 操作只需要花费 O(1) 的时间，Φ 增量为 1 ，因为增加的树的数目为 1, 标记没变所以 Φ 减少量为 0 。

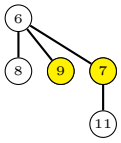
假如考虑一个操作序列包括 n 次插入，m 次找最小，r 次修改操作。整体真实运行时间为：$n + md_{max} + r +$ 整体 Φ 的减少量, 我们再变一下，总时间中 Φ 减小量肯定小于增加的量，因为 Φ 最开始为 0。所以，总运行时间小于 $n + md_{max} + r +$ 整体 Φ 增量。那么 Φ 总共增加的量 n*1(每次插入操作增加为 1)+md_max(每次找最小操作中增加为 d)+3r(每次修改操作中为 3 , 这个值有些问题)。所以整体运行时间为：$n + md_{max} + r + n + md_{max} + 3r = 2n + 2md_{max} + 4r$, 所以平均下来，每次插入操作为 O(1) 单位时间，找最小值操作花费 O(d_max), 修改操作花费 O(1) 的时间。

那么最后一个问题,d_{max} 到底是多少？

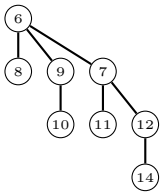
对于二项树很规整，n 个节点最多有 $log_2 n$ 个儿子。



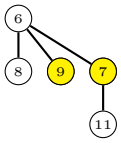
而对于斐波那契堆， 9 有可能损失儿子， 7 有可能丢了一个儿子，最终这个树被砍成如图所示：



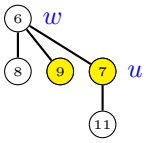
我们砍的没太狠，每个节点最多丢失一个儿子，所以 d 比二项树大一点，$log_\Phi n \geqslant d \geqslant log_2 n$，其中 $\Phi = \frac{1+\sqrt{5}}{2} = 1.618$
下面严格证明下斐波那契堆这种结构的性质。在二项树的第 i 个儿子有几个孙子，第第一个儿子有 0 个孙子，第 1 个儿子有 1 个孙子，第二个儿子有 2 个孙子，所以第 i 个儿子有准确的 i-1 个孙子。如图：



而在斐波那契堆树中，没有这么好的性质，第 i 个儿子至少有 i- 2 个孙子, 如图



为什么会这样？我们考察 u 时 w 的第 i 个儿子, 如果 w 当前不是一个根节点而且它最多丢失一个儿子。u 是什么时候称为 w 的儿子呢，在合并的时候。w 一开始应当有 8, 9 这些儿子，新来的 u 这棵树合并起来。我们考虑把 u 合并到 w 上那个时刻，w 至少有 i-1 个儿子 (u 是第 i 个，此时 u 还没合并进来)。u 当时肯定有两个儿子，不过后来丢掉了一个。因为相同级别的才会合并，所以合并的时刻 degree(u)=degree(w)$\geqslant i-1$，而后面的 u 最多失去一个儿子，所以 degree(u)$\geqslant i-2$



我们考察如下图所示的树，B_1 有两个儿子，B_1 最多可以丢掉一个儿子，所以 B 变成了 F_0

Figure 11: $|B_1| = 2^1$ and $|F_0| = 1 \geq \phi^0$

再看这幅图，我们考考虑 B_2 丢失儿子最大的情况，把 9 那个儿子丢掉，变成 F_1

Figure 12: $|B_2| = 2^2$ and $|F_1| = 2 \geq \phi^1$

再考虑 B_3, 考虑损失儿子最狠的情况，6 可以把儿子 7 丢掉，而 9 也还可以丢掉儿子 10, 所以最终结果为 F_2

对于 B_4 来说，丢失儿子最恨的情况，首先可以丢失最大的儿子 5，然后节点 7 还可以丢失儿子 12, 然后 9 还可以再丢掉儿子 10, 最终为 F_3。对于 B_4 这棵树来说非常完美，节点个数为 24, 而 F_3 有 5 个节点，个数肯定大于等于 Φ^3

再看这个递归的画法,B_5 的节点个数为 25,F_4 的节点个数为 8 , 肯定小于等于 Φ^4。我们可以观察到，从 F_0 到 F_4, 节点个数为 1,2,3,5,8, 也就是说满足这种规则的最小的树的节点的数目为斐波那契数。所以任何一棵树的规模大于等于 Φ^k, 所以 $d_max \leqq log_\Phi n$。

所以斐波那契堆的复杂度：插入仍未为 O(1), 取最小仍未 O(nlogn), 合并

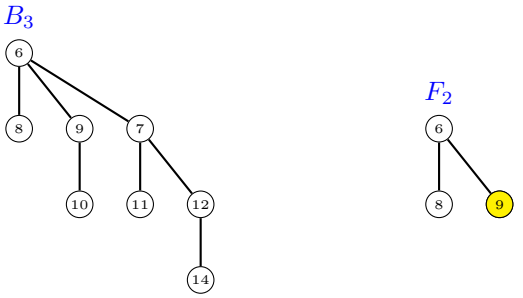


Figure 13: $|B_3| = 2^3$ and $|F_2| = 3 \geq \phi^2$

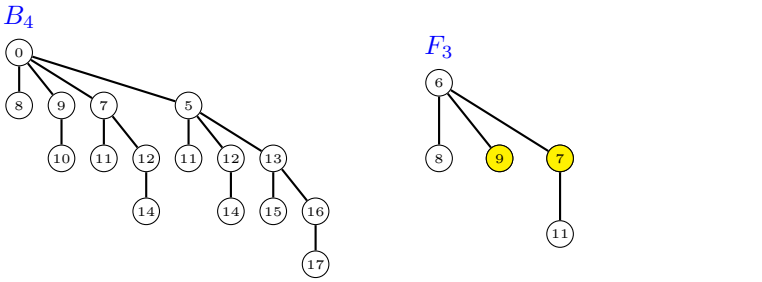


Figure 14: $|B_4| = 2^4$ and $|F_3| = 5 \geq \phi^3$

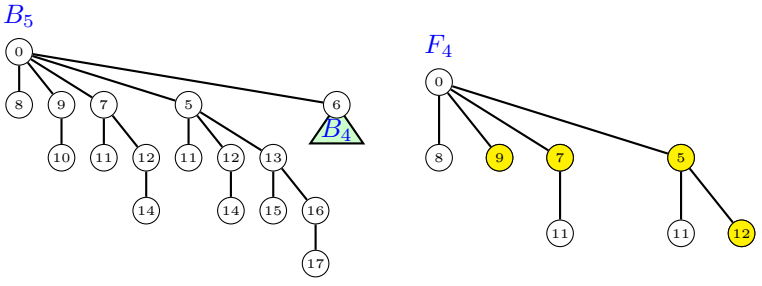


Figure 15: $|B_5| = 2^5$ and $|F_4| = 8 \geq \phi^4$

Operation	Linked list	Binary heap	Binomial heap	Fibonacci heap
MakeHeap	1	1	1	1
Insert	1	$\log n$	$\log n$	1
ExtractMin	n	$\log n$	$\log n$	$\log n$
DecreaseKey	1	$\log n$	$\log n$	1
Delete	n	$\log n$	$\log n$	$\log n$
Union	1	n	$\log n$	1
FindMin	n	1	$\log n$	1
Dijkstra	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

为 O(1), 而 DecreaseKey 的操作变成了 O(1) 的时间，为什么，变化某个数之后，只要违反了堆序，根本不去调整，直接当成一棵树，所以平均下来为 O(1) 的时间。

Operation	Linked List	Binary Heap	Binomial Heap	Binomial Heap *	Fibonacci Heap *
INSERT	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
EXTRACTMIN	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
DECREASEKEY	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
UNION	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$

* amortized cost

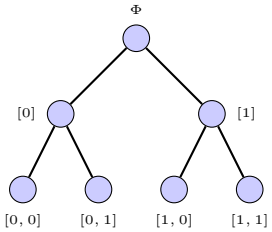
那么 DecreaseKey 的操作变为 O(1) 后对我们的算法有什么影响呢？Dijstra 算法的运行时间从最开始数组结构的 O(n2)，到二叉堆的 O(mlogn), 到二项堆的 O(mlogn), 到最终斐波那契堆的 O(m+nlogn)。时间复杂度好了很多。

Dijkstra algorithm: n Insert, n ExtractMin, and m DecreaseKey.

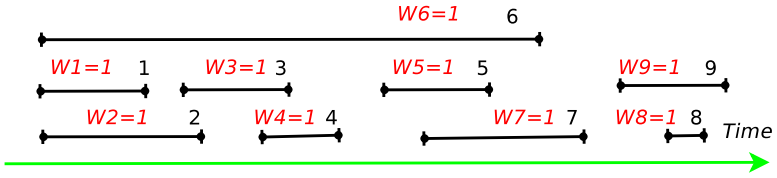
4 Greedy 贪心算法

贪心算法经常是每一步都选择局部最优的，是非常短视的 (Near Sight)，希望每次局部的最优最终能够达到全局最优。那么一个贪心算法应该怎么做呢，什么时候用？假如说我们观察到我们问题的解是 $X = [x_1, x_2, ...x_n]$, 每个 $x_i \in S_i$，我们的目标时要找 X^* 使得 X^* 为最大值。凡是我们碰到问题的解是由多项组成的，我们要考虑两招：一招是枚举，另一招就是贪心。这里有一个关键概念

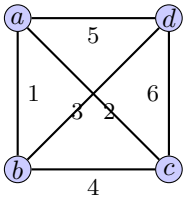
称为“部分解树”，我们的解太大不好找，我们让解慢慢的长起来, 如图，解从空开始，不断往下延伸，每个中间节点都是一个部分解，叶子节点是一个完整的解，我们最终从叶子节点中找到最优的。每次遇到这里问题可以画类似的图。第一招枚举所有的叶子节点，但是太慢，我们第一节提到多可以剪枝来减少规模，不用枚举所有的。第二招我们在第一个节点可以选择到底时让 x 等于 1 还是 0 , 根据当前的状态判断必然选 0 , 那么选 0 后在根据状态来判断必然要往下走，所以这是沿着一个或者多条路径来往下走。



下面我们看一个例子, 给我们上课起止的时间，每堂课容纳的学生都为 w= 1 ，让我们找选择那些课使得不冲突可以使最多的学生上课。我们遵循是 $X = [x_1, x_2, ...x_n]$，那么
$x_1 \in [1, 2, ..9]$，如果 x_1 选择 1 之后，$x_2 \in [3, 4, 5, 7, 8, 9]$, 以此我们可以把整个树枚举出来，最后在叶子节点中找到最大的那个。那么贪心怎么操作，对于空的时候，就觉得第一个选择 1 ，不枚举所有的树，其他都不用扩展，贪心选择 1 之后，我们选择 3 ， 3 再往下扩展到最终的叶子节点。所谓贪心就是在部分解树中找了一条路径。

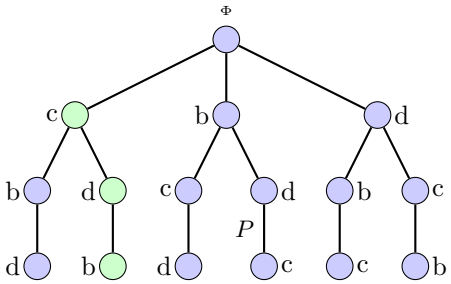


我们再来看 TSP 问题, 如图



$X = [x_1, x_2, ...x_n]$, 还是按照部分解树，先画出完整的解树，一开始解是空

的 Φ, 部分解树如图, 那么贪心规则是什么呢？我们就说从 a 出发到下一个节点谁最短选择谁, 也就是图示沿着绿色节点的路径。而其余的分支都不再考虑。所以贪心比较快，贪心每次选择局部最优。



我们主要目的不是讲各种技巧，而是观察给定的问题的结构：一类是可以规约成子问题可以分，一类不可以规约就 Impove, 还有一类就是枚举。

4.1 拟阵

问题：给定一个矩阵，求极大线性无关组。如图，给定五个向量, 我们还是按照上述的部分解树来看。贪心就是每次从里面选出一条路径出来，最终到叶子节点。

$$
\begin{aligned}
A_1 &= [\quad 1 \quad 2 \quad 3 \quad 4 \quad 5\ ] \\
A_2 &= [\quad 1 \quad 4 \quad 9 \quad 16 \quad 25\ ] \\
A_3 &= [\quad 1 \quad 8 \quad 27 \quad 64 \quad 125\ ] \\
A_4 &= [\quad 1 \quad 16 \quad 81 \quad 256 \quad 625\ ] \\
A_5 &= [\quad 2 \quad 6 \quad 12 \quad 20 \quad 30\ ]
\end{aligned}
$$

算法描述如下，初始解为空，对任意的行向量，加进去一个向量，如果是线性无关，就往解集合中加。

IndependentSet(M)

1: $A = \{\}$;
2: **for all** row vector v **do**
3: **if** $A \cup \{v\}$ is still independent **then**
4: $A = A \cup \{v\}$;
5: **end if**
6: **end for**
7: **return** A;

因为线性无关有两个性质:
(1) 继承性：如果 B 是一个线性无关组，如果 $A \subset B$, 那么 A 也是一个线性无关组。
(2) 扩展性: 如果 A 和 B 都是线性无关组，并且 |A|<|B|, 那么从 B 当中肯定能够找到一些向量加到 A 当中，使得 A 还是线性无关向量组。

考虑另一个问题: 加入每个向量都关联了一个权重 w, 我们在这里选出一些向量出来是线性无关的，同时权重之和是最大的。如图：我们按照贪心的想法，

$$
\begin{aligned}
A_1 &= [\ 1\ \ 2\ \ 3\ \ 4\ \ 5\ ] \quad W_1 = 9 \\
A_2 &= [\ 1\ \ 4\ \ 9\ \ 16\ \ 25\ ] \quad W_2 = 7 \\
A_3 &= [\ 1\ \ 8\ \ 27\ \ 64\ \ 125\ ] \quad W_3 = 5 \\
A_4 &= [\ 1\ \ 16\ \ 81\ \ 256\ \ 625\ ] \quad W_4 = 3 \\
A_5 &= [\ 2\ \ 6\ \ 12\ \ 20\ \ 30\ ] \quad W_5 = 1
\end{aligned}
$$

初始解为空，每次取权重最大的那个，本例中先选 A1, 然后再选 A2, 直到线性相关就不选。算法如下, 首先解为空，我们把所有的向量的按照权重做降序排列，然后和无权重的算法一样。算法复杂度为 O(nlogn+ nC(n)),O(nlogn) 是排序时间，C(n) 是每次检查线性无关花费多少时间。

MATROID_GREEDY(M, W)

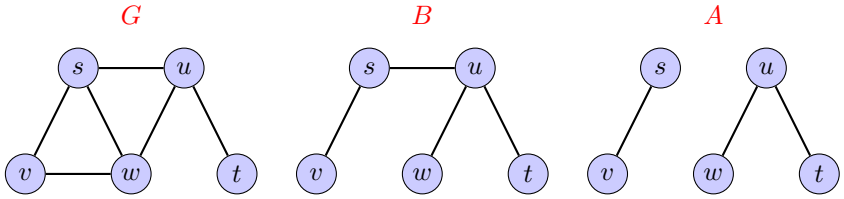
1: $A = \{\}$;
2: **Sort row vectors in the decreasing order of their weights;**
3: **for all** row vector v **do**
4: **if** $A \cup \{v\}$ is still independent **then**
5: $A = A \cup \{v\}$;
6: **end if**
7: **end for**
8: **return** A;

上述算法是对的么？首先这个算法第一是具有最优子结构的性质，第二个是具有贪心选择性质。首先 v 是权重最大的向量，并且 v 本身就是线性无关的。假设 A 是最优的解，且 A 包含 v。
证明：我们假设存在另外一个最优结构，但是 v $\notin$ B; 我们从 B 中开始构造一个新的 $A = v$, 因为 B 比 A 大，B 中肯定可以挑出一个向量添加到 A 中，直到 A 和 B 一样多，因为极大线性无关组的数目肯定一样多（即矩阵的秩）。最后 A 和 B 元素只有一个元素不同，因为 A 选择了 v, 所以 B 肯定选了另一个 v', 而 v 的权重最大，所以 A 的权重肯定比 B 大。那么剩下的自问题就是：和 v 线性相关的全部去掉，剩下的作为子问题 A'。因为 A' 中每个向量都是和 v 线性无

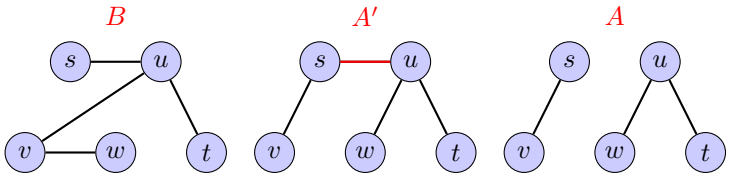
关的，所以 $A = A' \cup v$, 所以 A 肯定也是线性无关的。这样选择每次的 v 都具有最大的权重，所以最终会得到最优的解。

极大线性无关组并不仅仅存在在矩阵中，同样存在在其他问题中，我们称为“拟阵”(matroid, 由 Haussler 在 1935 年提出)。他发现矩阵中的线性无关概念和图论中的森林很像。如果一个图不包含回路，就称为独立。在图当中的无环子图（森林），连同且无圈称为树, 图中 B 所示。如果只要求无圈称为森林，图中 A 所示。

继承性: 如果一个森林，挑其中几个边，仍然是个森林（和线性无关组相似）。如图：



扩展性: 如果 A 和 B 都是无环森林，并且 A 是个小森林，那么从 B 中挑出一些边放到 A 中使得 A 还是森林。如图从 B 中添加 E_{su} 到 A 中形成 A'。如果森林 B 比 A 的边数要多 ($\#Tree = |V| - |E|$,V 是节点，E 是边)，那么 B 的树的数目比 A 要少，B 中肯定有一棵树是 A 中两棵树连接而成的。图例中的为 (u,v) 是跨了两棵树，所以加入到 A 中肯定不会形成圈。

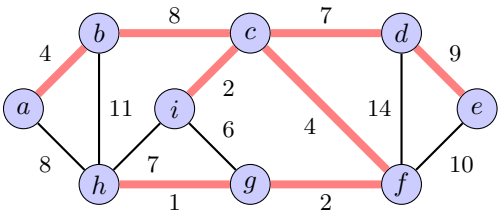


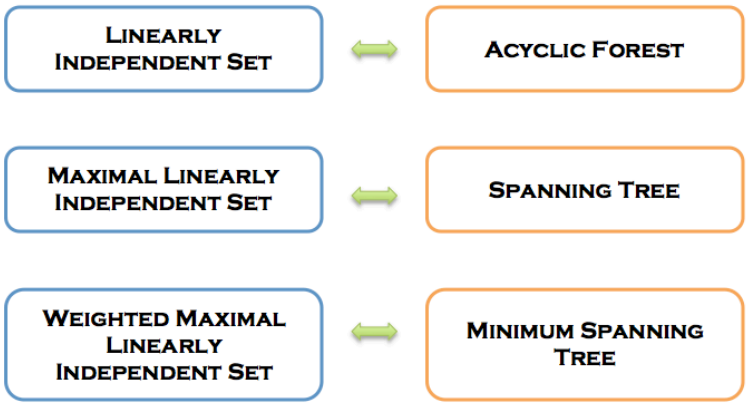
4.2 Kruskal 算法

那么拟帧有什么用呢？我们可以看下最小支撑树的问题：在实际电路板中，焊点之间都有距离，选择那些可以把这些焊点全部连接起来使得距离最小？

我们看下线性无关组和森林之间的关系：线性无关组对于森林。极大线性无关组对应支撑树。加权的加大线性无关组对应最小支撑树。所以我们就可以和处理加权极大线性无关组一样来处理最小支撑树。

GenericSpanningTree(G)





1: $F = \{\}$;
2: **while** F does not form a spanning tree **do**
3: find an edge (u, v) that is **safe** for F;
4: $F = F \cup \{(u, v)\}$;
5: **end while**

上述算法为求支撑树，算法每次找一条边如果是 safe 的话就加入到 F 中。Safe 意思就是不形成环。我们会发现和求矩阵线性无关组的算法基本一样。看下 Safe 和 Unsafe 的例子：

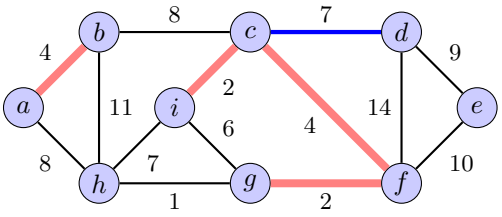


Figure 16: Safe edge

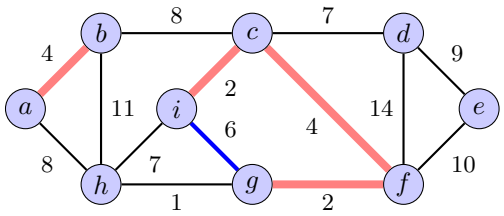


Figure 17: Unsafe edge

求支撑树我们已经会求了，那么最小支撑树呢？Kruskal 在 1956 年提出来的算法。

MST-KRUSKAL(G, W)

1: $F = \{\}$;
2: **for all** vertex $v \in V$ **do**
3: MAKESET(v);
4: **end for**
5: sort the edges of E into nondecreasing order by weight W;
6: **for** each edge $(u, v) \in E$ in the order **do**
7: **if** FINDSET$(u) \neq$ FINDSET(v) **then**
8: $F = F \cup \{(u, v)\}$;
9: UNION (u, v);
10: **end if**
11: **end for**

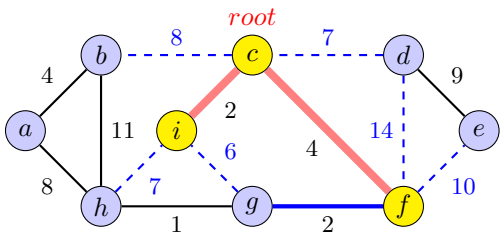
注意红色部分，首先按照权重把所有的边 E 按照非降序排列。一开始把所有节点建立一个集合 SET，每个节点单独成一个集合。v 在集合 V 中，u 在集合 U 中，每加一条边，判断 v 和 u 在不在一个集合里，在一个集合中的话就会形成圈。这里使用 $Union - Find$（并查集）这种数据结构来判断会不会形成环（两个元素是不是在一个集合当中）。

分析时间复杂度：第 2-3 行：n 次建立集合操作。第 5 行：排序花了 O(mlogm) 次的。for 循环中，检查两个端点，共 2m 次。增加边：n-1 次合并操作。最终为 $O((m+n)\aleph(n))$,$\aleph(n)$ 是逆阿克曼函数（非原始递归函数），增长非常慢，所以 $\aleph(n) = O(logn)$, 总体时间为 $O(mlogn)$

假如我们不知道 Kruskal 算法，我们怎么处理最小支撑树问题呢。我们还是按照部分解树来分析，每次都是从局部最优往下找。

4.3 Prim 算法

求解最小支撑树还可以用 Prim 算法，其最有解也是逐步构成，每一步始终保持肯定是一棵树，比 Kruskal 好些，不用检查是不是构成了圈。如图：



算法描述如下, 数据结构使用了优先队列，每次找已经添加到集合中的点和不在集合中的点的最小的边。

1: **for all** node $v \in V$ and $v \neq root$ **do**
2: $key[v] = \infty$;
3: $\Pi[v] =$ NULL; //$\Pi(v)$ denotes the predecessor node of v
4: PQ.INSERT(v); // n times
5: **end for**
6: $key[root] = 0$;
7: PQ.INSERT($root$);
8: **while** $PQ \neq$ NULL **do**
9: $u = PQ$.EXTRACTMIN(); // n times
10: **for all** v adjacent with u **do**
11: **if** $W(u, v) < key(v)$ **then**
12: $\Pi(v) = u$;
13: PQ.DECREASEKEY($W(u, v)$); // m times
14: **end if**
15: **end for**
16: **end while**

值得说的是，Prim 算法的不同数据结构上的复杂度，我们会发现 Prim 和 Dijstra 算法一样快，前提是使用了斐波那契堆。如图：PRIM algorithm: n INSERT, n EXTRACTMIN, and m DECREASEKEY.

拟阵在我们决定是否使用贪心非常有用的性质。如果我们在分析问题是发现问题的结构就和求解极大线性无关组一样，我们就可以直接用拟阵来解决。

Operation	Linked list	Binary heap	Binomial heap	Fibonacci heap
MakeHeap	1	1	1	1
Insert	1	$\log n$	$\log n$	1
ExtractMin	n	$\log n$	$\log n$	$\log n$
DecreaseKey	1	$\log n$	$\log n$	1
Delete	n	$\log n$	$\log n$	$\log n$
Union	1	n	$\log n$	1
FindMin	n	1	$\log n$	1
Prim	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

但是它不是万能的，比如哈夫曼编码、区间调度问题。因为哈夫曼编码每次都是把两个最小的合并成一个新的数值，在合并的过程中的子问题被改了。