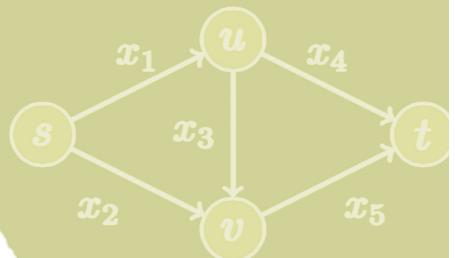


$$\begin{aligned}
 T(n) &= 2T(n/2) + cn \\
 &\leq 2c(n/2)\log_2(n/2) + cn \\
 &= 2c(n/2)\log_2 n - 2c(n/2) + cn \\
 &= cn\log_2 n
 \end{aligned}$$

\max
s.t.

$$\begin{array}{ll}
 x_1 - x_2 & f - f \leq 0 \text{ vertex } s \\
 -x_1 + x_3 & f - f \leq 0 \text{ vertex } t \\
 -x_2 - x_3 & f - f \leq 0 \text{ vertex } u \\
 -x_4 - x_5 & f - f \leq 0 \text{ vertex } v \\
 x_1 + x_2 + x_3 + x_4 + x_5 & \leq C_1
 \end{array}$$



算法讲义

卜东波 刘兴武 编著

关于问题求解方法的十八讲 LECTURES ON ALGORITHMS



国科大出版社

前　　言

公元 9 世纪，古波斯数学家 Al Khwarizmi 写了一本名为《The Compendious Book on Calculation by Completion and Balancing》的书。

这本书不仅记载了有关贸易、测量、遗产分配等方面的许多实际问题，还记载了抽象出来的线性方程甚至二次方程的求解步骤—这些求解方法步骤清晰、方便易行，被称之为“算法”，其英文 Algorithm 则源于 Khawarizmi 的拉丁文翻译，以此表示对 Al Khwarizmi 的敬意。

无独有偶。中国古代的数学著作《九章算术》以及《九章算术注释》也是记载许多实际问题以及相应的求解方法，甚至还使用“算”（运算次数）这一概念比较了不同算法的效率。中国古代数学重视解决来自现实生活的实际问题，而不像古希腊数学那样专注于证明定理；是故吴文俊先生称中国传统数学的特点是高度的机械化和算法化，这一点是和现代计算机科学相通的。

我们所写的这本讲义，是沿着“实际问题 → 抽象出的数学问题 → 算法设计”这条脉络总结我们的一些心得体会；如果说有一些特色的话，可能在于这本讲义不是简单地罗列现有的算法技术，而是试图强调“如何观察问题的结构”，强调“如何基于问题的结构进行算法设计”—求解问题的过程不应当只是逐个尝试各个算法技术或者纯粹依赖于灵感，而是应该依赖于我们对问题结构的认识；我们对问题结构认识得越深入，越有助于求解算法的设计。

我们在中国科学院大学的研究生班上多次讲授《算法设计与分析》课程，通常需要讲授一个学期、共十八次课。这本讲义是教学过程的一个产品；我们加上副标题“关于问题求解方法的十八讲”，是想表达这本书与教学之间的关系。

这本讲义的写作得到了李国杰、白硕、徐志伟等诸位老师的鼓励。他们奖掖后

进，拳拳之心，使人难生懈怠之意。我们谨以这本小书回馈他们的希冀。

我在滑铁卢大学听过 Timothy Chan 讲授的算法课。在把复杂的算法讲得清楚明白这一点上，我是从 Timothy Chan 那里领会到了很多。

有很多同事、同学担任了《算法设计与分析》课程的助教，对这本讲义有直接的贡献。我们在此对林宇、袁雄鹰、邵明富、王超、张海仓、黄春林、李锦、黄琴、凌彬、王耀军、许情、张任玉、巩海娥、杨飞、王冰、高枫、李艳博、朱建伟等同学表示诚挚的感谢 — 或许将他们列为本书的共同作者，更能彰显他们的贡献。

从写作风格上讲，我个人比较偏爱于《费恩曼物理学讲义》的写法：要揭示复杂事物背后的直观思想，自然的笔触或许更有助于理解本质性的东西。这份讲义是采用这种写法的一次尝试 — 由中国科学院大学的同学们依据课堂录音整理成文；我们在此对乔扬、申世伟、邵益文、黄斌、闫泽军、乔晶、袁伟超、李飞、孔鲁鹏、吴步娇、张敬玮、罗纯龙、曹晓然、梁志鹏、江涛等同学致以谢忱。

如何在讲清楚直观思想的同时又不失严谨性，是讲课和写作中最难把握的，也是最让我们困惑的 — 在这一点上，我们始终惴惴不安，只能静候同学们和读者们的建议和指正。

卜东波、刘兴武

2015 年于中关村

目录

前言	i
第一章 问题结构与基本算法思想	1
1.1 对问题可分解性的观察	1
1.1.1 子问题分解关系图	1
1.1.2 分而治之算法思想	1
1.2 对完全解空间的观察	1
1.2.1 完全解的邻域关系图	1
1.2.2 逐步改进算法思想	1
1.3 对解的形式的观察	2
1.3.1 部分解的枚举关系图	2
1.3.2 “聪明的”枚举算法思想	2
第二章 问题的分解及分而治之算法	3
2.1 排序问题	3
2.1.1 算法策略	3
2.2 逆序数计数问题	10
2.2.1 实际应用	10
2.2.2 分治策略	12
2.3 快速排序	14
2.3.1 快速排序的思想	14
2.3.2 快速排序复杂度 $O(n \log n)$ 的证明	15

2.3.3 改造快速排序	16
2.4 乘法问题	17
2.4.1 解决方法 1	18
2.4.2 新的分治算法	19
2.4.3 算法性能的比较	20
2.4.4 扩展：快速除法	20
2.5 矩阵乘法	21
2.5.1 解决方法 1	21
2.5.2 Strassen 算法	22
2.5.3 快速矩阵算法的进展	24
2.6 最近点对问题	24
2.6.1 解决方案	25
第三章 动态规划算法 (1)	32
3.1 矩阵链式乘法	32
3.1.1 解决问题的一般思路:	32
3.1.2 最优子结构性质:	35
3.1.3 记忆化搜索优化时间复杂度:	37
3.1.4 具体事例运算过程:	40
3.1.5 构建最优解方案:	42
3.1.6 问题总结:	43
3.2 0/1 背包问题:	43
3.2.1 具体事例:	44
3.2.2 动态规划求解思路:	44
3.2.3 时间复杂度的讨论:	48
3.2.4 另外一个子问题表示:	48
3.3 序列的连配问题	49
3.3.1 问题描述:	49
3.3.2 形式化定义:	49
3.3.3 动态规划求解:	53

3.3.4 构建最优解方案:	58
第四章 动态规划 (2)	59
4.1 上节课回顾	59
4.2 Hischberg 算法	59
4.2.1 第一个观察	59
4.2.2 第二个观察	62
4.2.3 第三个观察	62
4.2.4 算法	63
4.2.5 算法总结	64
4.3 alignment 问题的四个扩展	65
4.3.1 第一个扩展	65
4.3.2 第二个扩展	66
4.3.3 第三个扩展	67
4.3.4 第四个扩展	67
4.4 图上递归问题	68
4.4.1 TSP 问题	68
4.4.2 单源最短路问题	70
4.4.3 负圈判断问题	74
4.5 下节课内容	76
第五章 贪心算法	77
第六章 二项堆、斐波那契堆、贪心	99
6.1 回顾	99
6.2 二项堆	100
6.2.1 如何控制数的数目: 合并树	100
6.2.2 二项树及其性质	100
6.2.3 均摊分析	104
6.3 斐波那契堆	107
6.3.1 斐波那契堆的 Descreasekey 操作	108

6.3.2 斐波那契堆的插入操作	111
6.3.3 ExtractMin 取所有节点最小	111
6.3.4 对斐波那契堆做均摊分析: 这里的分析有些问题	112
6.4 Greedy 贪心算法	117
6.4.1 拟阵	119
6.4.2 Kruskal 算法	122
6.4.3 Prim 算法	124
第七章 线性规划问题 (1)	127
7.1 回顾	127
7.2 本章内容	128
7.3 几个例子	128
7.3.1 第一个例子: 饮食问题	128
7.3.2 第二个例子: 最大流问题	129
7.3.3 第三个例子: 最小费用流问题	130
7.3.4 第四个例子: 多物品流问题	131
7.3.5 第五个例子: 多物品流问题	132
7.3.6 补充例子: 基因组重排问题	133
7.4 动态规划问题的历史回顾	137
7.4.1 问题提出:	137
7.4.2 有关算法和分析:	137
7.4.3 NLP, Convex Programming, LP, Network flow, 和 ILP: . . .	138
7.4.4 求解 LP 问题的工具:	139
7.5 线性规划问题的求解	139
7.5.1 一般形式线性规划问题	139
7.5.2 标准形式线性规划问题	139
7.5.3 松弛形式线性规划问题	141
7.5.4 求解方法	142
7.5.5 线性规划问题的直观认识	143
7.5.6 约束条件 $\mathbf{x} \geq \mathbf{0}$ 的影响:	144

7.5.7 约束条件 $\min \mathbf{c}^T \mathbf{x}$ 的影响:	148
7.6 改进法求线性规划问题	149
7.6.1 例子:	150
7.6.2 为什么我们只用考虑多胞形的顶点就够了呢?	151
7.6.3 怎样选择初始点?	152
第八章 线性规划 (2)	157
第九章 对偶	175
9.1 内容引入	175
9.2 对偶的用处	175
9.3 DIET 问题	176
9.3.1 原始问题	176
9.3.2 对偶问题	177
9.3.3 DIET 问题总结	178
9.4 原始问题和对偶问题	179
9.4.1 线性规划矩阵	179
9.4.2 原始问题	180
9.4.3 对偶问题	181
9.4.4 原始问题与对偶问题之间的转换	181
9.5 拉格朗日乘子和 KKT 条件	182
9.5.1 拉格朗日乘子	182
9.5.2 KKT 条件	184
9.6 向线性规划中引入拉格朗日对偶	185
9.6.1 拉格朗日对偶	185
9.6.2 强边界	186
9.6.3 例子	186
9.6.4 拉格朗日——原问题和对偶问题之间的桥梁	187
9.6.5 对偶变量 y	188
9.7 对偶的四个性质	189

9.7.1 性质一：对偶的对偶就是原问题	189
9.7.2 性质二：弱对偶性	189
9.7.3 性质三：强对偶性	191
9.7.4 性质四：互补松弛性	191
9.8 原问题和对偶问题的 9 种情况	192
9.8.1 例子 1：原问题有无界最优解，对偶问题无解	193
9.8.2 例子 2：原问题和对偶问题都无解	193
9.9 对偶应用 1：Farkas 引理	194
9.9.1 引理内容	194
9.9.2 引理图例解释	194
9.9.3 引理证明	194
9.9.4 Farkas 引理变种	195
9.10 对偶应用 2：最短路径问题	196
9.10.1 最短路径问题回顾	196
9.10.2 线性规划形式	196
9.10.3 写出对偶问题	197
9.11 对偶单纯形算法	198
9.11.1 回顾原始问题单纯形算法	198
9.11.2 从另一个角度看原始问题单纯形算法	200
9.11.3 对偶单纯形算法	201
9.11.4 例子	202
9.11.5 对偶单纯形算法用处	204
9.12 原始对偶算法	204
9.12.1 原始对偶算法引入	204
9.12.2 原始对偶算法	204
9.12.3 最短路径：Dijkstra's algorithm 基本上是原始对偶算法	210
第十章 网络流算法	218
10.1 网络流：实际问题与算法发展脉络	218
10.1.1 概述	218

10.1.2 网络流的简短历史	219
10.1.3 最大流问题	220
10.1.4 Ford-Fulkerson algorithm	222
10.1.5 求解最小割问题的算法的发展史	227
10.1.6 最大流问题	228
10.2 Ford-Fulkerson 算法 [1956]	228
10.2.1 动态规划求解	228
10.2.2 Improvement 策略	229
10.2.3 正确性证明及时间复杂性分析	233
10.2.4 FORD-FULKERSON 算法的缺点	236
10.2.5 FORD-FULKERSON 算法的改进思想	241
10.3 改进策略一：Scaling 技术	241
10.3.1 Scaling FORD-FULKERSON 算法	242
10.3.2 Scaling 技术的时间复杂度	244
10.4 改进策略二：EDMONDS-KARP $O(m^2n)$ 算法	245
10.4.1 算法的创造者	245
10.4.2 EDMONDS-KARP 算法	245
10.4.3 时间复杂性分析	248
10.5 改进策略三：Dinitz 算法及 Dinic 算法	249
10.5.1 原始的 Dinitz 算法	249
10.5.2 Dinic 算法	250
10.6 从对偶的角度理解网络流	253
10.6.1 最大流 - 最小割：对偶问题	253
10.6.2 原始问题	254
10.6.3 FORD-FULKERSON 算法本质上是一个原始对偶算法	256
10.7 Push-relabel 算法	257
10.7.1 Push-relabel 算法简介	257
10.7.2 Push-relabel 算法描述	258
10.7.3 Push-relabel 算法实现	262

第十一章 网络流的应用	264
11.1 上节回顾	264
11.2 本节提要	264
11.3 网络流问题的扩展	265
11.3.1 无向图上的最大流问题	265
11.3.2 多源点、多汇点的流通问题	269
11.3.3 每条边有流量下界的流通问题	274
11.3.4 最小费用流问题	277
11.4 网络流的应用	284
11.4.1 集合划分	285
11.4.2 寻找路径	291
11.4.3 匹配	294
11.4.4 数字分解	297
第十二章 问题的难解性	301
12.1 NP 问题和难解性	301
12.1.1 问题和它的难度	301
12.1.2 ”问题”的定义	301
12.2 归约	303
12.2.1 多项式时间归约	303
12.2.2 多项式时间归约的图形表示及其意义	303
12.3 归约举例	304
12.3.1 INDEPENDENTSET \leq_P VERTEXCOVER	305
12.3.2 VERTEX COVER \leq_P SET COVER	309
12.3.3 通过“Gadget”归约: 3-SAT \leq_P INDEPENDENT SET	310
12.3.4 通过“Gadget”归约: SAT \leq_P HAMILTON CYCLE	314
12.3.5 HAMILTON CYCLE \leq_P TSP (TRAVELING SALESMAN PROBLEM)	320
12.3.6 SAT \leq_P GRAPH COLORING	325

第一章 问题结构与基本算法思想

TBA

1.1 对问题可分解性的观察

TBA

1.1.1 子问题分解关系图

TBA

1.1.2 分而治之算法思想

TBA

1.2 对完全解空间的观察

TBA

1.2.1 完全解的邻域关系图

TBA

1.2.2 逐步改进算法思想

TBA

1.3 对解的形式的观察

TBA

1.3.1 部分解的枚举关系图

TBA

1.3.2 “聪明的” 枚举算法思想

TBA

第二章 问题的分解及分而治之算法

本章主要内容

1. 第一个例子：归并排序
 - 用循环不变量的方法证明程序的正确性
 - 递归算法的时间复杂度
2. 其他例子：逆序对计数，最近点对，乘法和快速傅里叶变换。
3. 分治算法和随机算法结合在一起有很好的应用：快速排序算法，选择数算法。
分治算法的优势：有些问题使用暴力破解算法已经是多项式时间复杂度，但使用分治算法使得复杂度还能降低。比如最近点对的算法复杂度。

2.1 排序问题

2.1.1 算法策略

如果一个问题可以分成更小的子问题，我们有以下 2 种策略：

- (1) 增量式：一个数排序，增加一个数，2 个数排序，直到 n 个数排序。
- (2) 分治算法：大刀阔斧的分， n 个问题分两半， $n/2$ 的问题不会，则再分成 $n/4$ ，直到能解决为止。

增量式策略

基本思想：假设我们有一个部分解，例如 $A[0..j-1]$ 已经被正确排序，将 $A[j]$ 加入进来并放入正确的位置，那么 $A[0..j]$ 就被排好序了。

举例分析：插入排序。具体代码如下：

```

INSERTIONSORT( A )
1: for  $j = 0$  to  $n - 1$  do
2:    $key = A[j];$ 
3:    $i = j - 1;$ 
4:   while  $i \geq 0$  and  $A[i] > key$  do
5:      $A[i + 1] = A[i];$ 
6:      $i = i - 1;$ 
7:   end while
8:    $A[i + 1] = key;$ 
9: end for

```

- 最差的情况：数组是倒序排列的，那么每一个数加进来后，由于比前面所有的数都小，前面的数都要向后挪一个位置，因此 $T(n) = 1 + 2 + 3 + \dots + n$
- 插入排序的时间复杂度分析： $O(n^2)$.
- 迭代表达式表示为： $T(n) = T(n - 1) + cn = O(n^2)$.

分治策略

归并排序：第一次是被冯诺依曼在 1940s 提出的。

重要观察：一个问题能被划分成 2 个独立的子问题。具体步骤如下：

- Divide：将 n 个元素的序列分成 2 个分别有 $n/2$ 个元素的序列
- Conquer：假设每个子问题能被解决，实现方法是使用递归调用。
- Merge：将 2 个排序好的子序列合并成一个序列，得到原始问题的解。

算法伪代码如下：

```

MERGESORT( $A, l, r$ )
1: /* To sort part of the array  $A[l..r]$ . */
2: if  $l < r$  then
3:    $m = (l + r)/2;$  //  $m$  denotes the middle point;

```

```

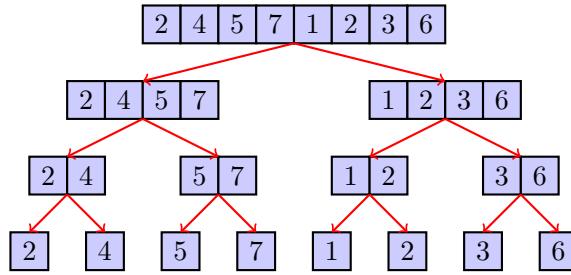
4: MERGESORT( A, l, m );
5: MERGESORT( A,m, r);
6: MERGE(A, l, m, r); // combining the sorted subsequences;
7: end if

```

第 4 步和第 5 步是将数组 $A[l..r]$ 分成 $A[l..m]$ 和 $A[m..r]$ ，并对两个子问题递归调用求解。第 6 步将 2 个部分解合并成最终解。

举例分析归并排序的具体做法

(一) 怎么分？



对于不会解的问题，将其分解到能被解决为止，在这里 2 个数的比较是算法的基础解，能被解决。

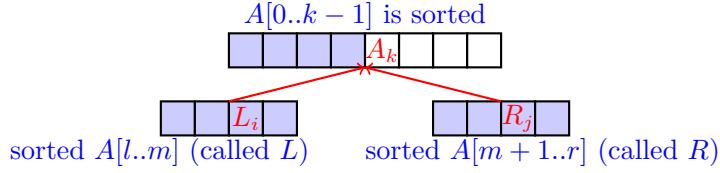
(二) 怎么合并？合并算法如下所示：

MERGE (A, l, m, r)

```

1: /* to merge  $A[l..m]$  (named as  $L$ ) and  $A[m + 1..r]$  (named as  $R$ ). */
2:  $i = 0; j = 0;$ 
3: for  $k = l$  to  $r$  do
4:   if  $L[i] < R[j]$  then
5:      $A[k] = L[i];$ 
6:      $i +=;$ 
7:   else
8:      $A[k] = R[j];$ 
9:      $j +=;$ 
10:  end if
11: end for

```



左边一半存入数组 L 和右边一半存入数组 R 都是已经排好的子序列，如何将 2 者合并成一个序列呢？

每次取两个数组最小的元素，取 2 者最小的填入数组 A 中，然后将数组下标右移一位。

a) 证明 merge 策略的正确性：循环不变量技术

- 循环不变量：类似于数学归纳法技术。
- 初始情况： $k = l$. 因为 $A[l..k-1]$ 为空，所以循环不变量成立。
- 特征保持：假设 $L[i] < R[j]$ ，而且 $A[l..k-1]$ 有 $k-1$ 个最小的数，那么把 $L[i]$ 放到 $A[k]$ ， $A[l..k]$ 就是拥有 $k-1+1$ 个最小的数

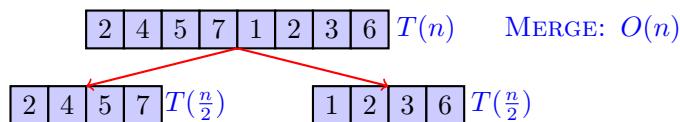
b) Merge 算法的时间复杂度

For 循环最多执行 n 次，故时间复杂度是 $O(n)$ 。

(三) 归并排序的时间复杂度

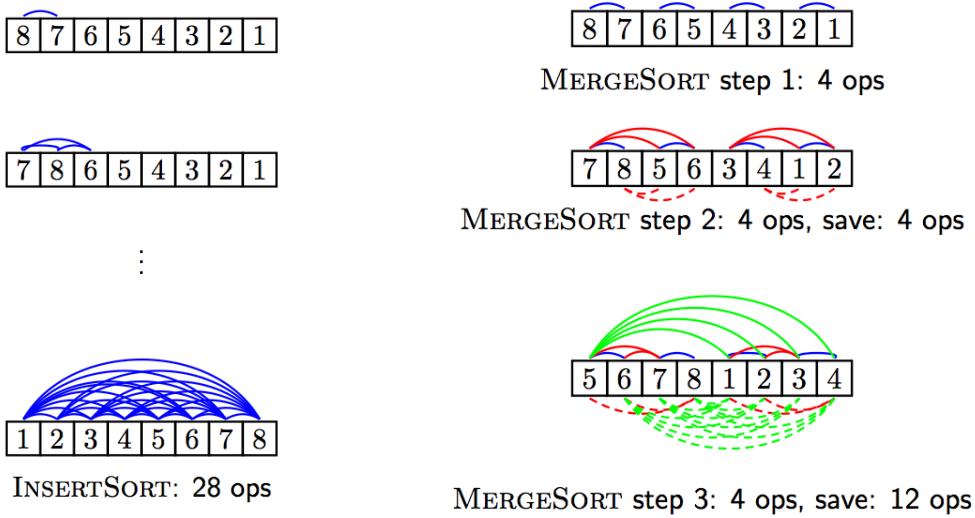
假设 $T(n)$ 是整个问题的时间复杂度，将其分成 2 半，左边需要 $T(n/2)$ ，右边需要 $T(n/2)$ ，那么可以写成如下：

$$T(n) = \begin{cases} c & n = 2 \\ T(n/2) + T(n/2) + cn & \text{otherwise} \end{cases} \quad (2.1.1)$$



从插入排序到归并排序，复杂度从 $O(n^2)$ 到 $O(n \log n)$ ，那么节省的究竟是什么呢？

下图表示的箭头数量是 2 种算法排序过程中需要进行的比较次数。



我们可以看出，插入排序需要 28 次比较。而归并排序中第 2 步省去了 4 次比较，第 3 步省去了 12 次比较。例如：第 3 步中，5 和 7 比较之后，不需要再和 8 进行比较。

分治算法的复杂度

归并排序的时间复杂度：如何精确计算分治算法的时间复杂度？

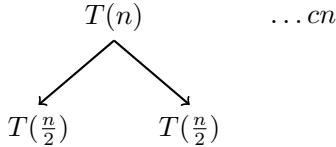
我们已经得到算法复杂度的递归表达式，如何将递归表达式写成精确地结果？主要有以下 3 种方法。

- 将表达式展开，展开一定程度之后就会观察到结果的模型。
- 猜测并证明猜测的正确性。
- 产生函数

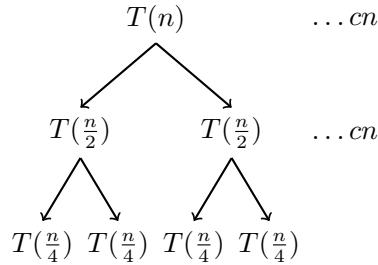
本书只讲前 2 种方法。

(1) 复杂度分析方法——展开

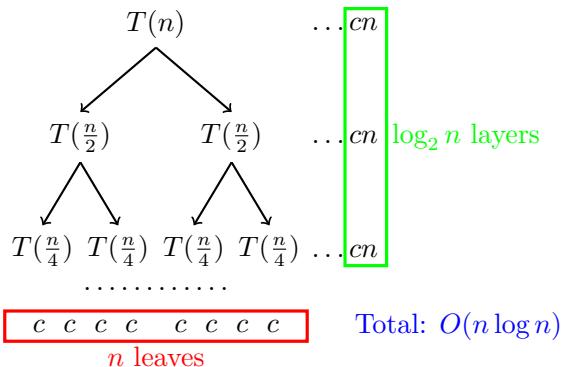
- 将 $T(n)$ 分成 $T(n/2)$ 的表达式，如果还看不出结果的话继续展开。



- 如果还不能解决，将 $T(n/2)$ 的问题展开成 $T(n/4)$ 的问题，直到能被解决为止。



- 直到展开的子问题的时间是常数。那么我们只需要计算最后一层常数时间的子问题的个数。



那么时间复杂度就由 2 部分组成：合并的时间和最后一层比较要花的时间。

那么究竟有多少层呢，很显然要层数是 $O(\log_2 n)$ 。那么合并花的时间就是 $cn * \log_2 n$ 。可以看出这棵树总共有 n 个叶子节点，花费时间是 cn 。

因此时间复杂度是 $O(n \log n)$

(二) 复杂度分析方法 2 —— 先猜测后证明

- 猜测一个解决方案并替换掉设定的系数，证明公式的正确性。
- 猜测: $T(n) \leq cn \log_2 n$ for all $n \geq 2$;
- 证明:

– 当 $n = 2$: $T(2) = c \leq cn \log_2 n$;

– 当 $n > 2$: 假设对所有 $m \leq n$ 都有 $T(m) \leq cm \log_2 m$ 。那么

$$T(n) = 2T(n/2) + cn \quad (2.1.2)$$

$$\leq 2c(n/2) \log_2(n/2) + cn \quad (2.1.3)$$

$$= 2c(n/2) \log_2 n - 2c(n/2) + cn \quad (2.1.4)$$

$$= cn \log_2 n \quad (2.1.5)$$

但事实上，我们经常猜的没有那么准，我们可能只猜到 $T(n) = O(n \log n)$ ，或者写成 $T(n) = k \log_b n$ 。K 和 b 我们不清楚，再去求解。

所以我们可能会猜测一个比较模糊的表达式，如下：

- 猜测和替换，我们只能猜测 k 和 b 可以等于多少，然后去替换掉。
- 模糊猜测: $T(n) = O(n \log n)$ 。时间复杂度这样描述 $T(n) = k \log_b n$, ***k, b 是待定系数***

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &\leq 2k(n/2) \log_b(n/2) + cn \quad (\text{set } b=2 \text{ for simplification}) \\ &= 2k(n/2) \log_2 n - 2k(n/2) + cn \\ &= kn \log_2 n - kn + cn \quad (\text{set } k=c \text{ for simplification again}) \\ &= cn \log_2 n \end{aligned}$$

后来，大家把递归表达式总结成一个定理。定理描述如下：

假设 $T(n)$ 定义为 $T(n) = aT(n/b) + f(n)$, 那么 $T(n)$ 会有如下性质：

1. If $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$;
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a}) \log n$;
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ and $af(n/b) \leq cf(n)$, then $T(n) = \Theta(f(n))$. Here, ϵ denotes a small, positive number.

举例分析如下

- Example 1: $T(n) \leq 3T(n/2) + cn$ (see a figure)

$$T(n) = O(n^{\log_2 3}) = O(n^{1.585})$$

- Example 2: $T(n) \leq 2T(\frac{n}{2}) + cn^2$ (see a figure)

$$T(n) = \sum_{j=0}^{\log n} \frac{cn^2}{2^j} = cn^2 \sum_{j=0}^{\log n} \frac{1}{2^j} = 2cn^2$$

(Note: not $O(n^2 \log n)$)

- Example 3: $T(n) \leq T(n/3) + T(2n/3) + cn$ (see a figure)

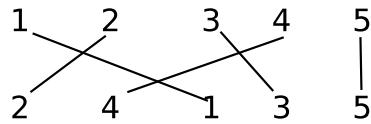
2.2 逆序数计数问题

2.2.1 实际应用

1. 识别 2 个人的相似度，比如比较 2 人对书本，电影等的排序。
2. meta search engine, 它把搜索请求提交给其他搜索引擎，比较各个搜索引擎的排序结果，计算它们的相似度。

逆序数问题的形式化表示

- 输入：一组 n 个不同的数的序列
- 输出：逆序对数，如果 $i < j$, 而 $a_i > a_j$, 那么就是逆序数对。



如上图表示，将正确的排好的序列 1,2,3,4,5 和序列 2,4,1,3,5 相同的数连起来，那么连线交叉的数量就是逆序数的对数。比如 1 和 2,1 和 4,3 和 4 总共 3 对。

逆序数的具体应用

- 基因序列的比较

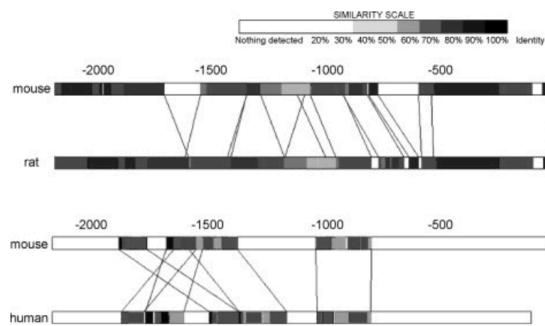


图 2.1: Sequence comparison of the 5' flanking regions of mouse, rat and human ER β .

在生物信息领域，比较 2 个基因组是否一样，基因组有很多基因的排序，假设有 n 个基因，上图是 mouse 和 rat 之间基因序列的比较，相同的基因连成一条线，交叉的地方就是逆序。老鼠和人的基因的相似情况同样如此。

- 求 2 个数列的相似度

常用方法: 1.Pearson 系数。2.Spearman 相关系数。新的策略:

$$W_1 = \sum_{i=1}^{n-k+1} (I_i^+, I_i^-)$$

表达式的含义是求相同长度子序列的相似度。举例分析

X : 1 3 4 2 5

Y : 1 4 5 2 3

$W_1 = 2$ when $k = 3$. 当设定比对的序列长度 (滑动窗口) 是 3 时, (1, 3, 4) 和 (1, 4, 5) 是没有逆序的, (3, 4, 2) 和 (4, 5, 2) 是没有逆序的, (4, 2, 5) 和 (5, 2, 3) 是有逆序的。那么相似度就是 2。这种算法的结果比 Pearson 系数要好。

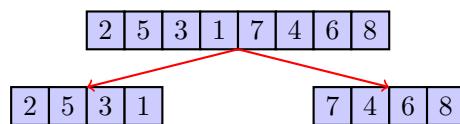
求逆序数的算法: 暴力方法: 枚举所有的点对, 然后得到逆序数的结果, 时间复杂度是 $O(n^2)$ 。那么有改善的算法吗?

2.2.2 分治策略

重要观察: 逆序数问题能够被划分成几个子问题。

分治策略:

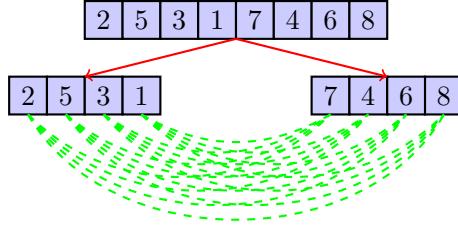
1. Divide: 分成 2 个序列: $A[0..n/2]$ 和 $A[n/2 + 1..n - 1]$;
2. Conquer: 递归调用计算各一半的逆序数;
3. Combine: 怎么比较分别位于左边和右边的数呢?



(一) 合并策略 1

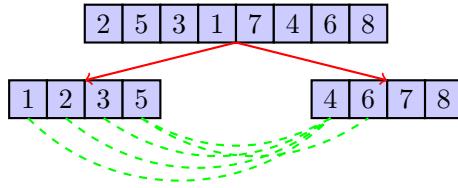
简单的枚举算法的话, 需要 $\frac{n^2}{4}$ 次比较。因此 $T(n) = 2T(\frac{n}{2}) + \frac{n^2}{4} = O(n^2)$.

对于通用的分治算法的递归表达式 $T(n) = aT(\frac{n}{b}) + f(n)$, 由于这里的合并时间消耗 $f(n) = \frac{n^2}{4}$, 所以时间复杂度太大。



(二) 合并策略 2: 假设左一半和右一半已经排好序了, 那么会大大减少比较的次数。

如果合并策略花费 $O(n)$ 时间, 那么整个分治算法的时间花费 $O(n \log n)$ 。



如图: 1 和 4 比较, $1 < 4$, 那么 1 不需要在和 4 后面的数比较。

实际上我们只需要在归并排序的基础上就行修改就可得到逆序数的解决方法。其算法伪代码如下:

SORT-AND-COUNT(A)

- 1: Divide A into two sub-sequences L and R ;
- 2: $(RC_L, L) = \text{SORT-AND-COUNT}(L);$
- 3: $(RC_R, R) = \text{SORT-AND-COUNT}(R);$
- 4: $(C, A) = \text{MERGE-AND-COUNT}(L, R);$
- 5: **return** $(RC = RC_L + RC_R + C, A);$

MERGE-AND-COUNT (L, R)

- 1: $RC = 0; i = 0; j = 0;$
- 2: **for** $k = 0$ to $\|L\| + \|R\| - 1$ **do**
- 3: **if** $L[i] > R[j]$ **then**
- 4: $A[k] = R[j];$
- 5: $j ++;$
- 6: $RC += (\frac{n}{2} - i);$
- 7: **else**

```

8:      $A[k] = L[i];$ 
9:      $i ++;$ 
10:    end if
11: end for
12: return ( $RC, A$ );

```

Time complexity: $T(n) = O(n \log n)$.

在合并的函数中，比较左边的数 $L[i]$ 和右边子数组的 $R[j]$ ，如果 $L[i] > R[j]$ ，那么 $L[i]$ 后面的数比 $R[j]$ 都要大，因此逆序数个数增加了 $\frac{n}{2} - i$ 。

逆序数问题的讨论：

排序的过程实际上就是减少逆序数的过程，假设我们记录了排序过程中减少的逆序数的数量，那么就得到了所有的逆序数的数量。

2.3 快速排序

2.3.1 快速排序的思想

快速排序和上次课讲的归并排序的思路比较像，但是分的办法不一样。

快速排序的分法：随便选择一个元素 $A[j]$ ，所有的其他元素和这个元素比较一次，那么就把比它小的放到 S_- ，比它大的元素放到 S_+ 。然后就是对 S_- 排序，对 S_+ 排序。接着就输出 $S_-, A[j], S_+$ 。

快速排序比归并排序的优势之一：代码比较简单。虽然代码写起来容易，但是分析就比较难了。理想情况下，是一次分 2 半，但是因为选择的随机性，我们不知道将数组分成的比例是多少。

- **Worst-case:** 选择了最大或者最小的元素作枢纽元；我们每次选择的都是数组中最大或者最小的数，分类操作需要一次循环，因此合并操作需要 $O(n)$ 的时间。那么复杂度就是 $O(n^2)$ 。

$$T(n) \leq T(n-1) + cn \Rightarrow T(n) = O(n^2)$$

- **Best-case:** 选择了中间元素作枢纽元; 那么复杂度是 $O(n \log n)$ 。

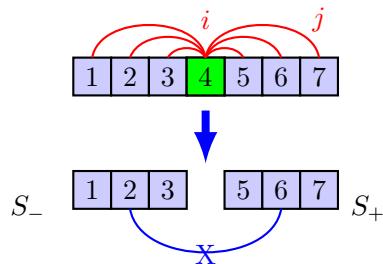
$$T(n) \leq 2T(n/2) + cn \Rightarrow T(n) = O(n \log n)$$

- **Most cases:** 事实上, 我们选择的情况不是最好也不是最差, 只会是一般的情况。我们接下来证明期望值是 $T(n) = O(n \log n)$ 。

2.3.2 快速排序复杂度 $O(n \log n)$ 的证明

我们在计数过程中只关心两两比较的次数, 用 X 表示对大小为 N 的数组排序比较的次数。

1. 第一个观察: 任意 i 和 j 最多比较一次, 假如选择了元素 4, 4 和所有元素比较了一次, 那么 4 以外的元素都被划分到了 2 个子数组, 4 不会再和其他元素比较。下面计算快速排序算法比较次数的期望值。



- 定义 $X_{ij} = I\{A[i]\text{ is compared with }A[j]\}$. 如果 $A[i]$ 和 $A[j]$ 比较了, $X_{ij} = 1$, 否则 $X_{ij} = 0$ 。

- 因此 $X = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}$.

$$\begin{aligned}
E[X] &= E\left[\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}\right] \\
&= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} E[X_{ij}] \\
&= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} Pr\{A[i] \text{ is compared with } A[j]\}
\end{aligned}$$

2. 第二个观察：当我们在处理 $A[i, i+1, \dots, j]$ 数组的时候， $A[i]$ 和 $A[j]$ 比较发生的条件是： $A[i]$ 或者 $A[j]$ 其中一个元素被选择作为枢纽元（划分数组的元素）。如果都没有被选中过，那么不会被比较。

例如：对于 1, 2, 3, 4, 5 这个数组，1 和 5 比较的条件是 1 或者 5 被选择，假如选择 2, 3, 4，那么 1 和 5 会被分到 2 个不同的子数组，就不会被比较。而且 1 和 5 比较的概率此时是 $2/5$ 。当 i, j 不是数组的边界时，这个概率小于 $\frac{2}{j-i+1}$ ，因此 $Pr\{A[i] \text{ is compared with } A[j]\} \leq \frac{2}{j-i+1}$ 。

那么我们重新计算前面的比较次数期望。

$$\begin{aligned}
E[X] &= \sum_{i=1}^n \sum_{j=i+1}^n Pr\{A[i] \text{ is compared with } A[j]\} \\
&\leq \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} \\
&= \sum_{i=1}^n \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^n \sum_{k=1}^n \frac{2}{k+1} \\
&= O(n \log n)
\end{aligned}$$

2.3.3 改造快速排序

我们把算法改造一下，让我们的分析更加简单。

MODIFIEDQUICKSORT(A)

```

1: while TRUE do
2:   randomly choose a splitter  $A[j]$ ;
3:   for  $i = 0$  to  $n - 1$  do
4:     Put  $A[i]$  in  $S_-$  if  $A[i] < A[j]$ ;

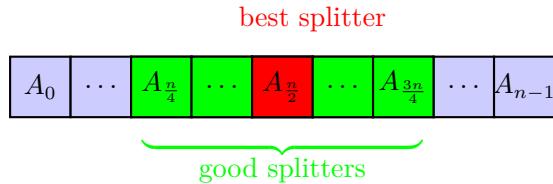
```

```

5:      Put  $A[i]$  in  $S_+$  if  $A[i] > A[j]$ ;
6:  end for
7:  if  $\|S_+\| > \frac{n}{4}$  and  $\|S_-\| > \frac{n}{4}$  then
8:    break;
9:  end if
10: end while
11: MODIFIEDQUICKSORT( $S_+$ );
12: MODIFIEDQUICKSORT( $S_-$ );
13: Output  $S_-$ , then  $A[j]$ , and finally  $S_+$ ;

```

我们在前面加了一个死循环，随机选择一个元素，并将整个数组分成 2 部分： S_- 和 S_+ 。如果 2 个子数组的大小都比 $n/4$ 要大，那么我们决定选择这个元素作枢纽；否则我们重新选择一个元素直到满足条件。



1. 选中中间元素 $A[n/2]$ 的概率是 $\frac{1}{n}$
2. 选中中间部分（绿色部分）的概率是 $\frac{1}{2}$ 。2 项分布的概率是 $\frac{1}{2}$ ，那么期望是 2。也就是循环 2 次就可以找到中间部分的枢纽元。那么合并的时间就是 $2n$ 。
 $T(n) = T(n/4) + T(3n/4) + 2n$ 的复杂度是 $O(n \log n)$ 。
3. 总结：

- 迭代深度是 $O(\log_{\frac{4}{3}} n)$.
- 每一次迭代找枢纽元的时间是 $O(n)$.
- $T(n) = O(n \log_{\frac{4}{3}} n)$.

2.4 乘法问题

一般做法如下图所示：

$$\begin{array}{r}
 & 1 & 2 \\
 \times & 3 & 4 \\
 \hline
 & 4 & 8 \\
 \hline
 3 & 6 & 8 \\
 \hline
 4 & 0 & 8
 \end{array}$$

提出问题： $O(n^2)$ 的复杂度是最优的吗？

2.4.1 解决方法 1

- 重要观察：2 个整数 x, y 可以表示成 n 位的二进制数，可以分成 2 部分。
- 分治策略：

1. **Divide:** $x = x_h \times 2^{\frac{n}{2}} + x_l, y = y_h \times 2^{\frac{n}{2}} + y_l,$

2. **Conquer:** calculate $x_h y_h, x_h y_l, x_l y_h$, and $x_l y_l$;

3. **Combine:**

$$xy = (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) \quad (2.4.1)$$

$$= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l \quad (2.4.2)$$

举例分析：计算 12×34

- Objective: to calculate 12×34
- $x = 12 = 1 \times 10 + 2, y = 34 = 3 \times 10 + 4$
- $x \times y = (1 \times 3) \times 10^2 + ((1 \times 4) + (2 \times 3)) \times 10 + 2 \times 4$

这个算法的复杂度：我们将其分成 4 个子问题，每个子问题的规模是 $\frac{n}{2}$ ，3 次加法。
 $T(n) = 4T(n/2) + cn \Rightarrow T(n) = O(n^2)$ 。如下表所示。

\times	y_h	y_l
x_h	$x_h y_h$	$x_h y_l$
x_l	$x_l y_h$	$x_l y_l$

我们可以发现以下问题：

- 我们的目标是计算 $x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l$ 。
- 我们发现没必要分别计算 $x_h y_l$ 和 $x_l y_h$, 只需要计算 $(x_h y_l + x_l y_h)$ 。
- 很明显 $(x_h y_l + x_l y_h) + (x_h y_h + x_l y_l) = (x_h + x_l) \times (y_h + y_l)$ 。所以 $(x_h y_l + x_l y_h)$ 只需要一次额外的乘法就能计算出来。

2.4.2 新的分治算法

- **Divide:** $x = x_h \times 2^{\frac{n}{2}} + x_l$, $y = y_h \times 2^{\frac{n}{2}} + y_l$,
- **Conquer:** 计算 $x_h y_h$, $x_l y_l$, 和 $P = (x_h + x_l)(y_h + y_l)$;
- **Combine:**

$$xy = (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) \quad (2.4.3)$$

$$= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l \quad (2.4.4)$$

$$= x_h y_h 2^n + (P - x_h y_h - x_l y_l) 2^{\frac{n}{2}} + x_l y_l \quad (2.4.5)$$

还是刚才那个例子：计算 12×34

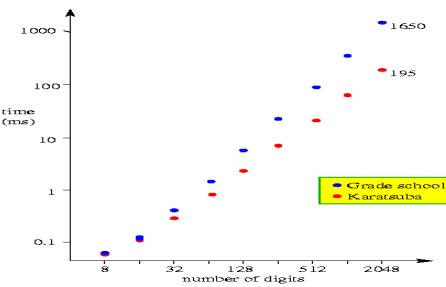
1. Objective: to calculate 12×34
2. $x = 12 = 1 \times 10 + 2$, $y = 34 = 3 \times 10 + 4$
3. $P = (1 + 2) \times (3 + 4)$
4. $x \times y = (1 \times 3) \times 10 \cdot 2 + (P - 1 \times 3 - 2 \times 4) \times 10 + 2 \times 4$

这个算法可以分成 3 个子问题，6 次加法，2 次移位。那么算法的复杂度变成 $T(n) = 3T(n/2) + cn \Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})$ 。

2.4.3 算法性能的比较

那么这个算法到底怎么样呢？

对于 n 很大时，Karatsuba 的算法效果是很明显的。但是当 n 的规模比较小时，额外的移位和加法操作将使这个算法变慢。当使用快速傅立叶变换的技术，乘法时间复杂度是 $O(n \log n)$ 。



- 牛顿迭代法:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (2.4.6)$$

$$= x_i - \frac{t - \frac{1}{x_i}}{\frac{1}{x_i^2}} \quad (2.4.7)$$

$$= -t \times x_i^2 + 2x_i \quad (2.4.8)$$

- 牛顿迭代法的迭代速度: 迭代次数只需要 $\log \log t = O(\log n)$ 。

4. 举例分析: 计算 $\frac{1}{13}$

#Iteration	x_i	ϵ_i
0	0.018700	-0.058223
1	0.032854	-0.044069
2	0.051676	-0.025247
3	0.068636	-0.008286
4	0.076030	-0.000892
5	0.076912	-1.03583e-05
6	0.076923	-1.39483e-09
7	0.076923	-2.77556e-17
8

2.5 矩阵乘法

问题: 给定 2 个 $n \times n$ 矩阵 A 和 B , 计算 $C = AB$ 。

2.5.1 解决方法 1

一般的方法: A 的第 i 行和 B 的第 j 列相乘得到 C_{ij} , 复杂度是 $O(n^3)$ 。因为根据定义矩阵 C 中每个元素都需要 $O(n)$ 次乘法, 总共有 n 个元素。

关键观察: 将矩阵分成 4 块, 每一块是 $\frac{n}{2} \times \frac{n}{2}$ 的大小。

分治算法:

1. **Divide:** 把 A , B , 和 C 分别划分成 4 个小矩阵;

2. **Conquer:** 计算子矩阵的乘积;

3. **Combine:**

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C_{11} = (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \quad (2.5.1)$$

$$C_{12} = (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \quad (2.5.2)$$

$$C_{21} = (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \quad (2.5.3)$$

$$C_{22} = (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \quad (2.5.4)$$

算法的复杂度：每个问题分成 8 个子问题和 4 次加法。每次加法花费 $O(n^2)$ 时间。所以时间复杂度是： $T(n) = 8T(n/2) + cn^2 \Rightarrow T(n) = O(n^3)$

问题：能降低矩阵乘法的复杂度吗？

2.5.2 Strassen 算法

Strassen 算法：第一次提出的算法比 $O(n^3)$ 要快。

算法的主要思想

减少求解子问题的数目：8 个子问题的结果并不一定都需要，只需要算出 7 个矩阵的结果就能表示出来。

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$P_1 = A_{11} \times (B_{12} - B_{22}) \quad (2.5.5)$$

$$P_2 = (A_{11} + A_{12}) \times B_{22} \quad (2.5.6)$$

$$P_3 = (A_{21} + A_{22}) \times B_{11} \quad (2.5.7)$$

$$P_4 = A_{22} \times (B_{21} - B_{11}) \quad (2.5.8)$$

$$P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) \quad (2.5.9)$$

$$P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \quad (2.5.10)$$

$$P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12}) \quad (2.5.11)$$

$$C_{11} = P_4 + P_5 + P_6 - P_2 \quad (2.5.12)$$

$$C_{12} = P_1 + P_2 \quad (2.5.13)$$

$$C_{21} = P_3 + P_4 \quad (2.5.14)$$

$$C_{22} = P_1 + P_5 - P_3 - P_7 \quad (2.5.15)$$

Strassen 算法的复杂度

那么 Strassen 算法的复杂度是 $T(n) = 7T(n/2) + cn^2 \Rightarrow T(n) = O(n^{\log_2 7}) = O(n^{2.807})$ 。

Strassen 算法的优劣

- **优势:**

1. 比一般的算法速度快很多。

2. Strassen 算法可以用来解决其他问题: 矩阵求逆, 行列式求值, 图的三角形个数计数。

- **劣势:**

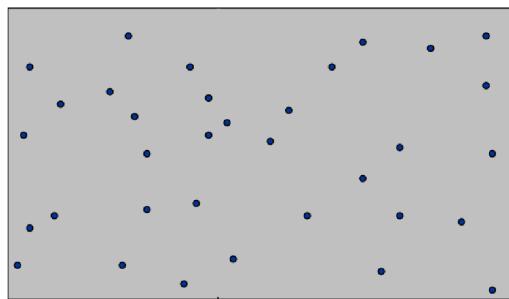
1. 只对于规模较大的问题复杂度有所提升。
2. 稳定性不足。
3. 相对于其他算法要求更多的内存资源。

2.5.3 快速矩阵算法的进展

- multiply two 2×2 matrices: 7 scalar sub-problems: $O(n^{\log_2 7}) = O(n^{2.807})$ [Strassen 1969]
- multiply two 2×2 matrices: 6 scalar sub-problems: $O(n^{\log_2 6}) = O(n^{2.585})$ (impossible)[Hopcroft and Kerr 1971]
- multiply two 3×3 matrices: 21 scalar sub-problems: $O(n^{\log_3 21}) = O(n^{2.771})$ (impossible)
- multiply two 20×20 matrices: 4460 scalar sub-problems: $O(n^{\log_{20} 4460}) = O(n^{2.805})$
- multiply two 48×48 matrices: 47217 scalar sub-problems: $O(n^{\log_{48} 47217}) = O(n^{2.780})$
- Best known: $O(n^{2.376})$ [Coppersmit-Winograd, 1987]
- Conjecture: $O(n^{2+\epsilon})$ for any $\epsilon > 0$;

2.6 最近点对问题

- **INPUT:** 平面上的 n 个点
- **OUTPUT:** 欧式距离最近的点对。



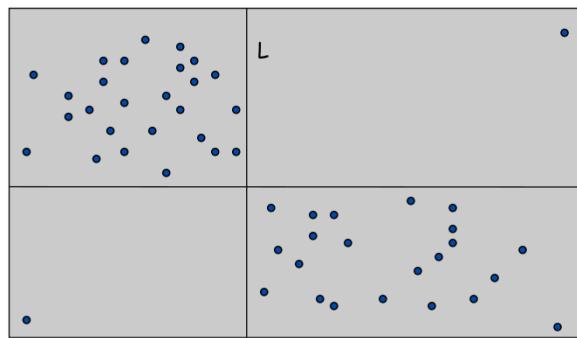
- 直线上: 将点进行排序, 找出 2 个距离最近的点, 复杂度是 $O(n \log n)$ 。

- 平面上: 比较所有的点对, 需要 $O(n^2)$ 的时间。

问题: 能够找出更快的算法吗?

2.6.1 解决方案

方案 1: 分成 4 个子问题

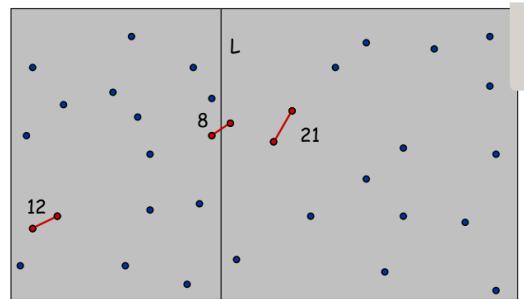


困难: 分成的子集可能是不均衡的。我们无法保证每个子集都有差不多 $n/4$ 个点。因此可能需要 $O(n^2)$ 的时间去合并子集, 最后的算法迭代表达式可能是这样的:
 $T(n) = 2T(\frac{n}{2}) + O(n^2)$ 。

方案 2: 分成 2 个子问题

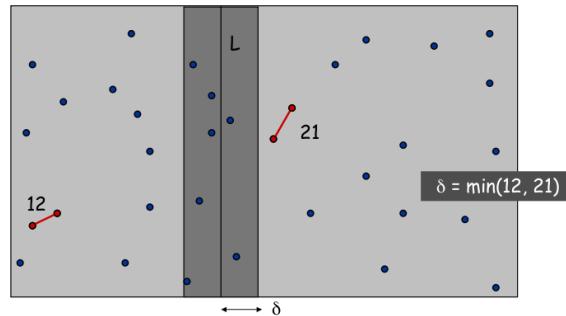
解决方法: 很容易将所有的点安装 x 坐标进行排序, 然后用 $x_{\lfloor \frac{n}{2} \rfloor}$ 将所有点分成 2 半。

- **Divide:** 将所有点分成 2 半;
- **Conquer:** 找出每个子问题的解: 左半部分的最近点对, 右半部分的最近点对;
- **Combine:** 最近点对可能存在于边界左边一点和右边一点的情况。

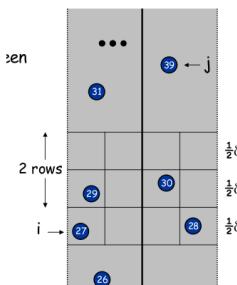
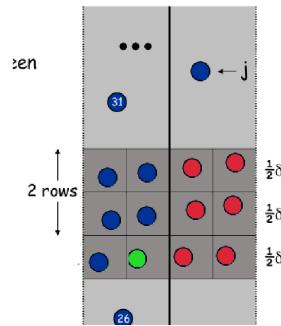


如果对左半边的每个点和右半边的每个点都做计算，那么时间开销很大：需要 $O(n^2)$ 的时间。

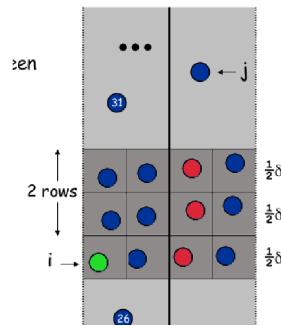
观察 1：实际上，我们只需要观察 L 条带里面的点对的距离。那么条带的范围是 2δ ， δ 是取左边的最近点对和右边的最近点对的距离的较小者。如图所示， $\delta = \min(12, 21)$ 。

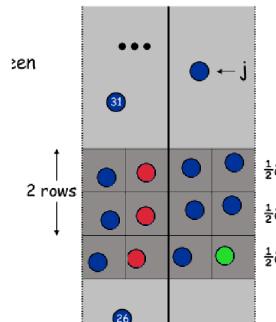
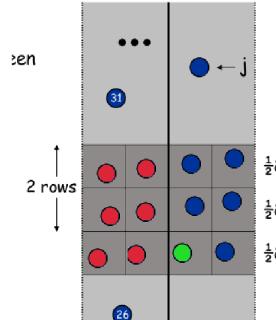


观察 2：即使在条带里面也没必要两两之间去计算。对中间的条带进行分割，每个格子的边长是 $\delta/2$ 。显然，一个格子里面只能装一个点。原因：当 2 个点在一个格子里面时，它们的距离将会小于 δ 。



接下来分析可能出现的最近点对的计算，下面 4 种情况是可能出现的最近点对的情况。每幅图中绿色的点只需要和红色的点继续比较即可。





对于左边的点，只需要对右边的最多 6 个点比较即可。我们将条带中的点按照 y 坐标进行排序，那么只需要找出临近的 11 个点即可，因为距离它最近的点一定在这 11 个点里面。算法伪代码表示如下：

CLOSESTPAIR(p_i, \dots, p_j) /* p_i, \dots, p_j have already been sorted according to x -coordinate; */

- 1: if $j - i == 1$ then
- 2: return $d(p_i, p_j)$;
- 3: end if
- 4: Use the x -coordinate of $p_{\lfloor \frac{i+j}{2} \rfloor}$ to divide p_i, \dots, p_j into two halves;
- 5: $\delta_1 = \text{CLOSESTPAIR}(\text{left half})$; $T(\frac{n}{2})$
- 6: $\delta_2 = \text{CLOSESTPAIR}(\text{right half})$; $T(\frac{n}{2})$
- 7: $\delta = \min(\delta_1, \delta_2)$;
- 8: Sort points within the 2δ strip by y -coordinate; $O(n \log(n))$

- 9: Scan points in y -order and calculate distance between each point with its next 11 neighbors. Update δ if finding a distance less than δ ; $O(n)$

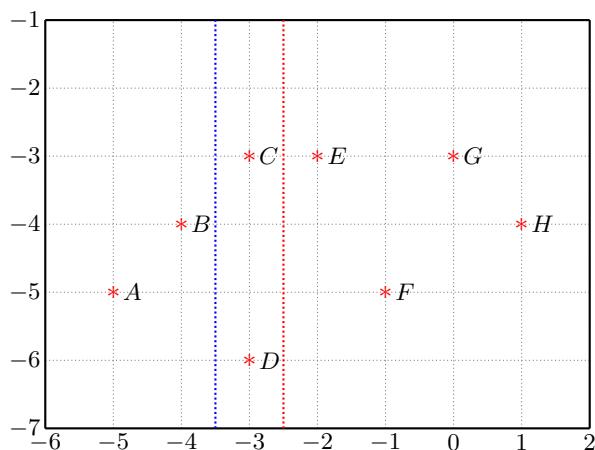
Time-complexity: $T(n) = 2T(\frac{n}{2}) + O(n \log n) = O(n \log^2(n))$.

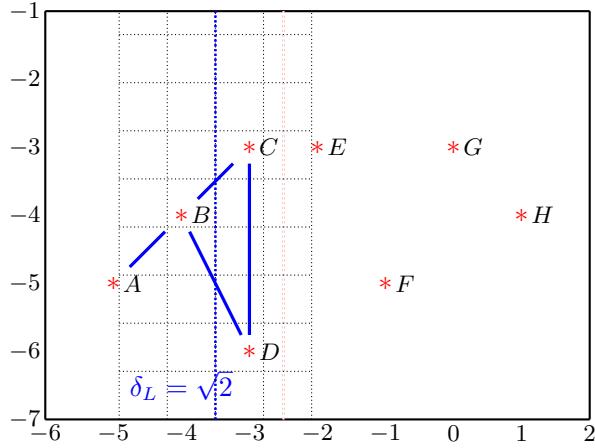
算法的提升

上面算法的合并时间：边界附近的点进行排序需要 $O(n \log n)$ 时间。

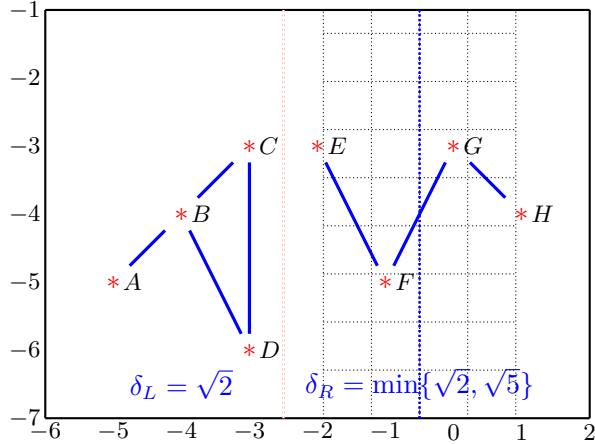
如果每一次迭代保持 2 个序列，一个按照 x 排序的序列，一个按照 y 排序的序列。像归并排序一样，将预先排序好的 2 个方向的序列合并起来，实际上只需要 $O(n)$ 的时间。因此时间复杂度是 $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

举例分析：8 个点的最近点对

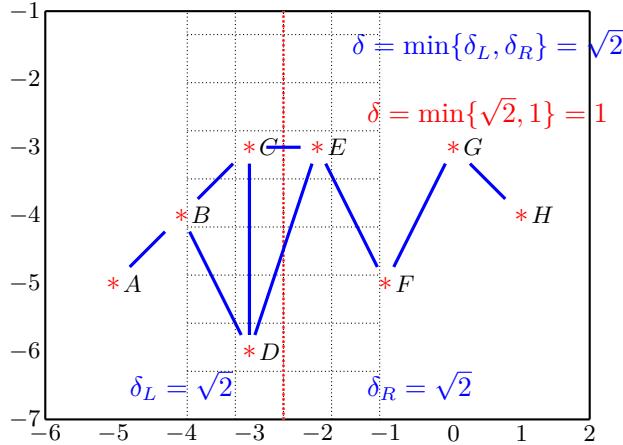




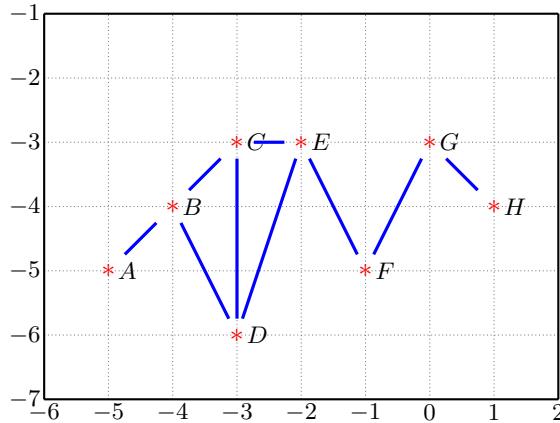
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.



- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow \min = \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.



- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10} \Rightarrow \delta = 1$.



- 我们只计算了 9 对点。剩下的 19 对是冗余的，原因如下：
 - 至少有一个点位于 2δ 条带之外。
 - 虽然 2 个点都在 2δ 条带内，但是它们之间的距离超过了 2 排格子 (格子大小: $\frac{\delta}{2} \times \frac{\delta}{2}$)。

第三章 动态规划算法 (1)

3.1 矩阵链式乘法

下面我们将从“矩阵链式乘法”这个简单的例子入手讲解动态规划。通过讲解这个例子，我们可以总结下如果要用动态规划的算法去解决实际问题，需要有哪些要素、解决问题的关键是什么以及怎样描述并定义子问题。

3.1.1 解决问题的一般思路：

首先我们先回忆下上一节提到的解决问题的三个基本思路。

碰到一个问题，如果这个问题太大了以至于搞不定，看能不能把它变小，把它规整成小问题去解决，这是考虑问题最基本的想法。

比如说给你一个长度为 n 的数组， n 个数太多了我们不会做，我们可以先尝试一个数能不能做，然后两个数能不能做，这样一直进行下去。

另外一种对待问题的思路是，我们可以把 n 个数分成左一半和右一半，然后看左半部分会不会做，右半部分会不会做。将大问题分解为小问题去解决，这就是分治的思想。

动态规划与分治算法的联系：

(1) 动态规划和分治是非常像的，都是要把大问题分解成子问题，然后将子问题的解进行合并起来求原问题的解。

(2) 动态规划一般会枚举所有的子问题，要把所有的子问题都解决一遍，但是它避免了对同一个子问题的重复计算，那它是怎么避免重复的呢，这就是 programming。programming 的意思是说生成一张表，不断的向表中填数，当访问到

表中单元时，如果表中有值则直接返回，没有则进行求解并将求到的值填在表中。

(3) 动态规划和贪心一样，都可以典型的求解最优化问题，但是动态规划又不仅用于最优化问题的求解，例如 p-value 的计算问题。

通常来说，只要我们发现一个问题当中能存在一种递归的性质，我们就可以把它分解成子问题，就能找到一种递归的关系，这个时候就可以用动态规划进行求解。

当我们在计算一个原始问题的时候，我们需要把原问题进行扩展，试图发现有意义的递推关系，而确定递推关系的关键就在于确定子问题的一般形式。

矩阵链式乘法的形式化描述：

- **Input:**

A sequence of n matrices A_1, A_2, \dots, A_n ; matrix A_i has dimension $p_{i-1} \times p_i$;

- **Output:**

Fully parenthesizing the product $A_1A_2\dots A_n$ in a way to minimize the number of scalar multiplications.

我们的目标是给我们 n 个矩阵 $A_1, A_2, A_3\dots, A_n$ ，其中 A_i 大小为 $P(i-1) * P(i)$ ，求最好的加括号方案使得整体的运算次数最少。

具体事例：

下面我们看下矩阵链式乘法这个例子：

比如说我们有如下四个矩阵 A_1, A_2, A_3, A_4 ：

$$\begin{aligned}
 A_1 &= \begin{bmatrix} 1 & 2 \end{bmatrix} A_2 = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} A_3 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix} A_4 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} \\
 &\quad 1 \times 2 \qquad\qquad\qquad 2 \times 3 \qquad\qquad\qquad 3 \times 4 \qquad\qquad\qquad 4 \times 5
 \end{aligned}$$

Solutions: $((A_1)(A_2))(A_3))A_4$ $((A_1)(A_2))((A_3)(A_4))$

$$\begin{array}{ll}
 \text{Cost:} & \begin{array}{ll} 1 \times 2 \times 3 & 1 \times 2 \times 3 \\ +1 \times 3 \times 4 & +3 \times 4 \times 5 \\ +1 \times 4 \times 5 & +1 \times 3 \times 5 \\ = 38 & = 81 \end{array}
 \end{array}$$

对这四个矩阵做运算，我们有很多种加括号的方案，比如：

方案一：先算 $A_1 * A_2$ ，然后再乘以 A_3 ，最后乘以 A_4 ，此时总共运算次数为 38 次。

方案二：先算 $A_1 * A_2$ ，然后算 $A_3 * A_4$ ，最后将两者得到的矩阵相乘，此时总的运算次数为 81 次。

我们使用两种不同的求解顺序，得到了不同的运算次数且差异很大。现在我们想知道第一种方案是最优的吗？是否存在更好的加括号的方案使得总共的运算次数最少。

解空间：

总共有多少种加括号的方案呢，实际上加括号的方案可以描述成一颗二叉树，二叉树的每个节点对应一个子问题。总共 n 个节点，则有 $\binom{2n}{n} - \binom{2n}{n-1}$ (Catalan number) 个从根节点到叶子节点的路径即 $\binom{2n}{n} - \binom{2n}{n-1}$ 个加括号的方案。

卡特兰数是指数级别的，解空间非常大，如果暴力枚举的话，速度会非常慢，所以暴力枚举这种策略是不可行的。

动态规划的一般思路：

下面我们研究下动态规划是怎么进行求解的。

问题的求解还是基于我们的观察，现在给我们 n 个矩阵，我们不会做，我们可以看能不能把它分解成小问题来求解。

我们的解就是加括号的方案，把求解过程想象成一系列的决策。每一步的决策是确定在哪个位置加括号，即确定先算哪些矩阵，再算哪些矩阵。

假如当前我们拿到了一个最优解，我们记做 O ，且假设我们将第一个括号加在第 k 个矩阵与第 $k+1$ 个矩阵之间，即 $(A_1 \dots A_k)(A_{k+1} \dots A_n)$ ，也就是说我们要先算 $(A_1 \dots A_k)$ ，再算 $(A_{k+1} \dots A_n)$ ，最后将两部分得到的矩阵乘起来。

这样我们就把原始问题分解成在 $(A_1 \dots A_k)$ 里面加括号使得运算次数最小和在 $(A_{k+1} \dots A_n)$ 加括号使得运算次数最小两个子问题了。

我们可以发现左半部分的右下标需要改变，右半部分的左下标需要改变，因此该问题便是从 A_i 到 A_j ，这样问题的左右下标都可以变化。因此子问题一般形式表示为在 (A_i, \dots, A_j) 加括号使得整体运算次数最少，我们记做 $OPT(i, j)$ ，显然原始问题可以表示成 $OPT(1, n)$ 。

因为左下标可以从 1 变化到 n ，而右下标可以从 i 变化到 n ，因此原问题的解空间为 $\sum_{i=1}^n (n-i+1) = \frac{n(n+1)}{2}$ ，即原问题包含 $O(n^2)$ 个子问题空间。

假如我们是在 A_k 和 A_{k+1} 矩阵之间加的括号，则在 A_i 到 A_j 所使用的运算次数就是

$$Cost(i, j) = Cost(i, k) + Cost(k+1, j) + p_i p_{k+1} p_{j+1}$$

该表达式对于任意的解都有这个性质。因此我们考虑最优解这个特殊情况有：

$$OPT(i, j) = OPT(i, k) + OPT(k+1, j) + p_i p_{k+1} p_{j+1}$$

即 (A_i, \dots, A_j) 最少的运算次数，等于 (A_i, \dots, A_k) 最少的运算次数加上 (A_k, \dots, A_j) 最少的运算次数加上最后两个矩阵相乘的运算次数。即原问题包含子问题的最优解，这就是最优子结构性质。

求解一个问题，如果这个问题可以规约，则分治大概可以解决问题。如果他还有最优子结构的性质，则动态规划大概可以解决问题。因此在碰到一个问题的时候，先观察该问题具有哪些性质，根据不同的性质我们使用不同的策略。

3.1.2 最优子结构性质：

那我们如何证明最优子结构呢：

如果对 (A_i, \dots, A_k) 有另外一种代价更小的加括号方案 $OPT^*(i, k) < OPT(i, k)$, 那将它替换到 (A_i, \dots, A_j) 的最优加括号的策略中, 就会产生另外一种加括号的方案, 且代价小于最优代价 $OPT(i, j)$, 这与原始定义 $OPT(i, j)$ 是最优的矛盾。因此原问题一定包含子问题的最优解。

本问题隐含了独立性假设即左边的求解与右边的求解是彼此不影响的。

但是现在我们还不能知道 k 具体在哪个位置, 因此需要在 i 和 j 区间内对 k 的所有情况进行枚举, 则我们有如下递推表达式:

$$OPT(i, j) = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{ OPT(i, k) + OPT(k+1, j) + p_i p_{k+1} p_{j+1} \} & \text{otherwise} \end{cases}$$

现在我们可以利用得到的递归表达式, 写出如下伪代码:

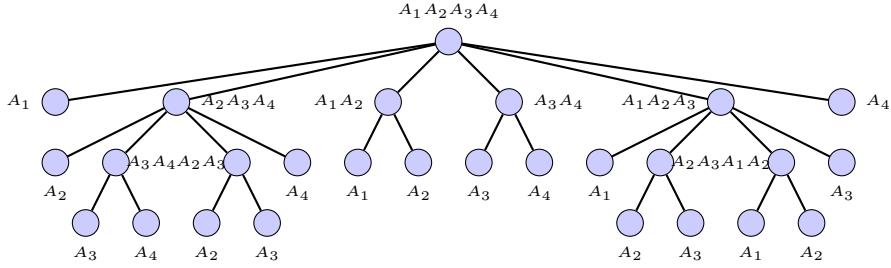
```

RECURSIVE_MATRIX_CHAIN( $i, j$ )
1: if  $i == j$  then
2:   return 0;
3: end if
4:  $OPT(i, j) = +\infty$ ;
5: for  $k = i$  to  $j - 1$  do
6:    $q = \text{RECURSIVE\_MATRIX\_CHAIN}(i, k)$ 
7:   +  $\text{RECURSIVE\_MATRIX\_CHAIN}(k + 1, j)$ 
8:   +  $p_i p_{k+1} p_{j+1}$ ;
9:   if  $q < OPT(i, j)$  then
10:      $OPT(i, j) = q$ ;
11:   end if
12: end for
13: return  $OPT(i, j)$ ;
```

注意最优解可以通过调用 $\text{RECURSIVE_MATRIX_CHAIN}(1, n)$ 得到。

通过伪代码的递归形式, 我们可以画出如下递归树:

递归树中的每个节点都代表一个子问题, 我们可以发现上面的递归树中有很多节点是重复的。



时间复杂度：

现在我们求下该程序的时间复杂度：

假设 $T(n)$ 为计算 n 个矩阵乘积的复杂度，则有

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \text{ for } n > 1$$

下面我们证明 $T(n) \geq 2^{n-1}$, 即原问题是指数时间的复杂度：

首先我们有 $T(1) \geq 1 = 2^{1-1}$ 然后针对 $n > 1$ 的情况，我们有：

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad (3.1.1)$$

$$= n + 2 \sum_{k=1}^{n-1} T(k) \quad (3.1.2)$$

$$\geq n + 2 \sum_{k=1}^{n-1} 2^{k-1} \quad (3.1.3)$$

$$\geq n + 2(2^{n-1} - 1) \quad (3.1.4)$$

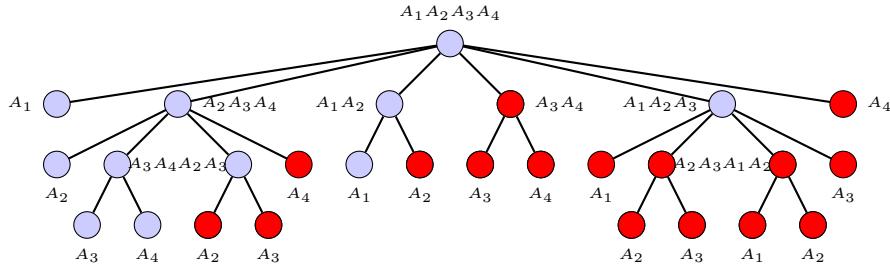
$$\geq n + 2^n - 2 \quad (3.1.5)$$

$$\geq 2^{n-1} \quad (3.1.6)$$

3.1.3 记忆化搜索优化时间复杂度：

现在有个问题，我们总共只有 $O(n^2)$ 个子问题，但是现在我们的程序竟然花费了 2^n 时间，表明这里面肯定有子问题被重复计算了。

比如下图的红色节点：



我们直观的想法是可以将已经计算过的子问题保存在一张表里面。下次如果再求解该子问题则直接将表中存放的值返回即可，这样便避免了重复计算。

因此我们有下面伪代码：

```

MEMORIZE_MATRIX_CHAIN( $i, j$ )
1: if  $OPT[i, j] \neq NULL$  then
2:   return  $OPT(i, j)$ ;
3: end if
4: if  $i == j$  then
5:    $OPT[i, j] = 0$ ;
6: else
7:   for  $k = i$  to  $j - 1$  do
8:      $q = MEMORIZE_MATRIX_CHAIN(i, k)$ 
9:     + $MEMORIZE_MATRIX_CHAIN(k + 1, j)$ 
10:    + $p_i p_{k+1} p_{j+1}$ ;
11:    if  $q < OPT[i, j]$  then
12:       $OPT[i, j] = q$ ;
13:    end if
14:   end for
15: end if
16: return  $OPT[i, j]$ ;
```

其实就是在原有伪代码的基础上添加了对表格中该单元的判断，如果该单元有值，则表明已经算过该子问题则直接返回，否则则对该子问题进行求解并保存在表格中。

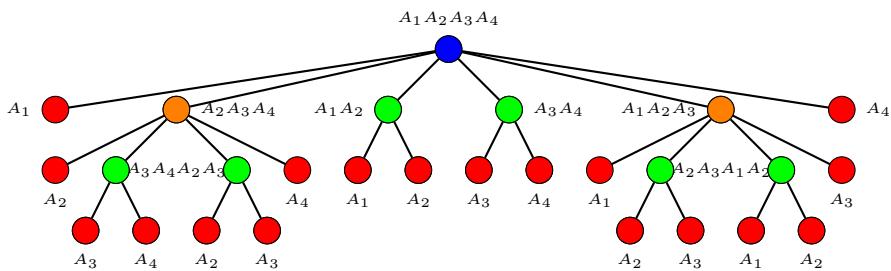
因为原问题有 $O(n^2)$ 个子问题，每个子问题有 $O(n)$ 种选择，因此原问题的时间复杂度为 $O(n^3)$ 。

由于递归需要使用内存中的栈结构，运算速度稍微慢些，因此我们可以尝试使用非递归的形式自底向上进行计算。因此有以下伪代码：

```

MATRIX_CHAIN_MULTIPLICATION( $P$ )
1: for  $i = 1$  to  $n$  do
2:    $OPT(i, i) = 0;$ 
3: end for
4: for  $l = 2$  to  $n$  do
5:   for  $i = 1$  to  $n - l + 1$  do
6:      $j = i + l - 1;$ 
7:      $OPT(i, j) = +\infty;$ 
8:     for  $k = i$  to  $j - 1$  do
9:        $q = OPT(i, k) + OPT(k + 1, j) + p_i p_{k+1} p_{j+1};$ 
10:      if  $q < OPT(i, j)$  then
11:         $OPT(i, j) = q;$ 
12:         $S(i, j) = k;$ 
13:      end if
14:    end for
15:  end for
16: end for
17: return  $OPT(1, n);$ 
```

因此我们有以下的运算过程：



(1) 求解红色(叶子)节点上的子问题。

(2) 求解绿色节点上的子问题。

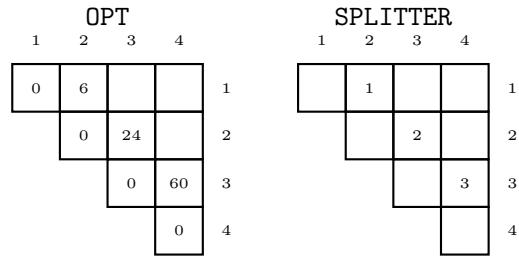
(3) 求解橘黄色节点上的子问题。

(4) 得到原问题的最优解。

我们可以看到动态规划是自低向上求解最优解问题的。

3.1.4 具体事例运算过程:

下面我们详细看下动态规划求解矩阵链式乘法的运算过程:



第一步:

$$OPT[1, 2] = p_0 \times p_1 \times p_2 = 1 \times 2 \times 3 = 6;$$

$$OPT[2, 3] = p_1 \times p_2 \times p_3 = 2 \times 3 \times 4 = 24;$$

$$OPT[3, 4] = p_2 \times p_3 \times p_4 = 3 \times 4 \times 5 = 60;$$

OPT				SPLITTER			
1	2	3	4	1	2	3	4
0	6	18			1	2	
	0	24	64	2		2	
		0	60	3		3	
			0	4			

第二步：

$$OPT[1, 3] = \min \begin{cases} OPT[1, 2] + OPT[3, 3] + p_0 \times p_3 \times p_4 (= 18) \\ OPT[1, 1] + OPT[2, 3] + p_0 \times p_2 \times p_4 (= 32) \end{cases} \quad \text{Thus, } SPLITTER[1, 2] = 2.$$

$$OPT[2, 4] = \min \begin{cases} OPT[2, 2] + OPT[3, 4] + p_1 \times p_2 \times p_4 (= 90) \\ OPT[2, 3] + OPT[4, 4] + p_1 \times p_3 \times p_4 (= 64) \end{cases} \quad \text{Thus, } SPLITTER[2, 4] = 3.$$

OPT				SPLITTER			
1	2	3	4	1	2	3	4
0	6	18	38		1	2	
	0	24	64	2		2	
		0	60	3		3	
			0	4			

第三步：

$$OPT[1, 4] = \min \begin{cases} OPT[1, 1] + OPT[2, 4] + p_0 \times p_1 \times p_4 (= 74) \\ OPT[1, 2] + OPT[3, 4] + p_0 \times p_2 \times p_4 (= 81) \quad \text{Thus, } SPLITTER[1, 4] = 3. \\ OPT[1, 3] + OPT[4, 4] + p_0 \times p_3 \times p_4 (= 38) \end{cases}$$

经过我们一步步的计算最后得到结果是 38，即我们运用动态规划得到的第一种方案是最好的方案。

3.1.5 构建最优解方案：

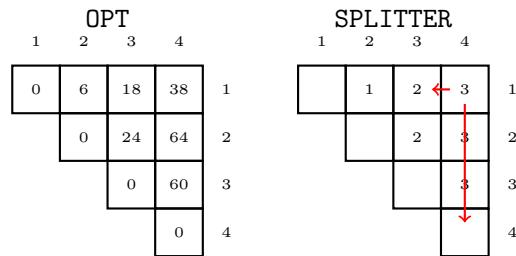
虽然我们找到了最优解的值，但是怎样求得与该最优解对应的解方案呢。

我们的解决方案是进行回溯，需要我们另外使用一个数组 S 保存当前节点最优解的来源。比如 $S[i, j]$ 是记录了对乘积 A_i, \dots, A_j 在 A_k 与 A_{k+1} 之间进行分开以取得最优加括号方案的 k 值。

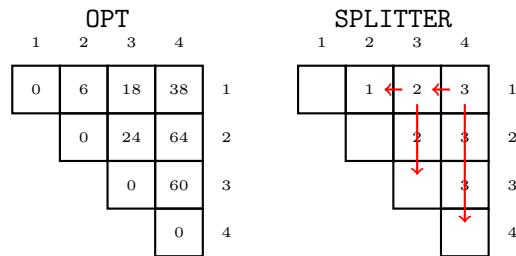
因此原问题 A_1, \dots, A_n 的最优解方案就是 $(A_1, \dots, A_{S[1,n]+1})(A_{S[1,n]+1}, \dots, A_n)$ 。

这里我们需要注意：原问题的最优解只有当所有的子问题都被算出来后才能得到。

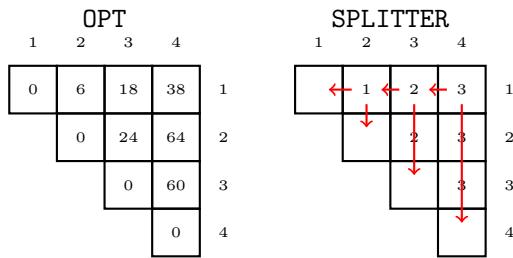
下面我们尝试回溯找到最优解方案：



第一步： $(A_1 A_2 A_3)(A_4)$



第二步： $((A_1 A_2)(A_3))(A_4)$



第三步: $((A_1)(A_2))(A_3))(A_4)$

根据我们的回溯, 知道了最优解方案就是先算 A_1A_2 然后结果乘以 A_3 , 最后乘以 A_4 .

3.1.6 问题总结:

经过对矩阵链式乘法的整个求解过程, 我们有如下总结:

如果大问题搞不定, 我们可以将其分解成更小的子问题。在动态规划求解问题时关键是如何定义子问题, 我们可以将求解过程想象成多步决策的过程, 再假设已经拿到了最优解, 然后考察第一个决策是做什么的, 此时可能会有多种情况, 那我们就枚举所有情况, 然后观察子问题的所有形式, 并进行总结便会得到子问题的一般形式即递归表达式。

最后看该问题是否满足最优子结构的性质。如果满足则根据递归表达式, 我们便可以写出源代码。

3.2 0/1 背包问题:

现在有个物品集, 每个物品都有重量和价值, 现在希望选择一个物品子集, 使得总的重量小于给定的重量并且总的价值最大。

问题的形式化描述:

该问题的形式化描述如下:

- **Input:**

A set of items. Item i has weight w_i and value v_i , and a total weight limit W ;

- **Output:**

A sub-set of items to maximize the total value with a total weight below W .

这里 0/1 的意思是只能装或者不装这个物品，装记做 1，不装记做 0.

3.2.1 具体事例：

下面我们看下一个生活中的例子：

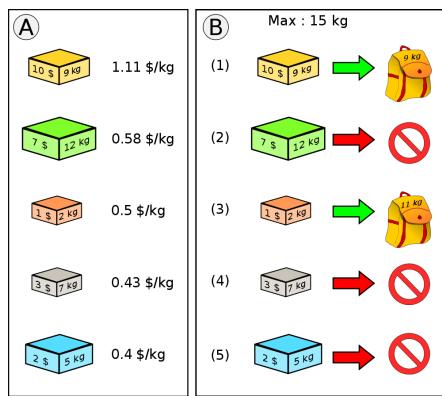


图 3.1: 物品从上到下依次为金银铜铁锡

我们直观的想法可能是先装单价贵重的物品，比如图中的物品，我们可能会先装金块，因为最大只有 15kg，装完金块后还剩 6 kg，此时还能装铜块和 x i 块。按照我们先装单价贵重的物品则会装铜块，此时还剩下 4 kg，则没有物品可以装的下了。
按照我们启发式的装东西想法得到的总价为 11\$，可是我们启发式得到的一定是最优的吗？下面我们看下动态规划是怎么求解的。

3.2.2 动态规划求解思路：

现在我们还是按照之前解决矩阵链式乘法的求解思路来：

当物品集是 n 个物品的时候，我们不知道怎么解决，可以先考虑 $n - 1$ 个物品，或者 $n/2$ 个物品，看可不可以将物品数目变少。

我们解方案是选哪些物品，是物品的子集。我们可以把问题的求解想象成一系列的决策，在第 i 步，我们决定第 i 个物品是装还是不装。

现在假设我们已经得到子问题的最优解，当前考虑第一个决策即考虑最后一个物品是装还是不装。

如果装，则原问题变成从前 $n - 1$ 个物品中选择限重 $W - w_n$ 的物品子集使得价值最大。

如果不装，则问题变成在前 $n - 1$ 个物品中选择限重 W 的物品子集使得价值最大。

此时对子问题的两种情况进行总结，我们可以得到子问题的一般形式，在前 i 个物品中选择物品，选物品的价值越大越好，将子问题的最优解记做 $OPT(i, w)$.

我们对 (i, w) 的两种情况进行分析，并取最大值，则有如下递归表达式：

$$OPT(i, w) = \max\{OPT(i - 1, w), OPT(i - 1, w - w_n) + v_n\}$$

根据递归表达式，我们可以写出如下伪代码：

KNAPSACK(n, W)

```

1: for  $w = 1$  to  $W$  do
2:    $OPT[0, w] = 0;$ 
3: end for
4: for  $i = 1$  to  $n$  do
5:   for  $w = 1$  to  $W$  do
6:      $OPT[i, w] = \max\{OPT[i - 1, w], v_i + OPT[i - 1, w - w_i]\};$ 
7:   end for
8: end for
```

我们列了一张 4 行 7 列的一张表格， $dp[i][j]$ 表示前 i 个物品，容量为 j 的时候最多可以装多少价值的物品。

程序的开始需要对数组进行初始化，对前 0 个物品装，无论容量多大，最大效益都是 0。

因此我们有如下求解过程：

Backtracking: step 1

Initially all $\text{OPT}[0,w] = 0$

	W = 0	1	2	3	4	5	6
i=3							
i=2							
i=1							
i=0	0	0	0	0	0	0	0

$$\begin{aligned} \text{OPT}[1,2] &= \max\{ \\ &\quad \text{OPT}[0,2] (=0), \\ &\quad \text{OPT}[0,0] + V_1 (=0+2) \} \\ &= 2 \end{aligned}$$

	W = 0	1	2	3	4	5	6
i=3							
i=2							
i=1	0	0	2	2	2	2	2
i=0	0	0	0	0	0	0	0

$$\begin{aligned} \text{OPT}[2,4] &= \max\{ \\ &\quad \text{OPT}[1,4] (=2), \\ &\quad \text{OPT}[1,2] + V_2 (=2+2) \} \\ &= 4 \end{aligned}$$

	W = 0	1	2	3	4	5	6
i=3							
i=2	0	0	2	2	4	4	4
i=1	0	0	2	2	2	2	2
i=0	0	0	0	0	0	0	0

$$\begin{aligned} \text{OPT}[3,3] &= \max\{ \\ &\quad \text{OPT}[2,3] (=2), \\ &\quad \text{OPT}[2,0] + V_3 (=0+3) \} \\ &= 3 \end{aligned}$$

	W = 0	1	2	3	4	5	6
i=3	0	0	2	3	4	5	5
i=2	0	0	2	2	4	4	4
i=1	0	0	2	2	2	2	2
i=0	0	0	0	0	0	0	0

	W = 0	1	2	3	4	5	6
Backtracking	0	0	2	3	4	5	5
OPT[3,6] = max{	0	0	2	2	4	4	4
OPT[2,6] (=4),	0	0	2	2	2	2	2
OPT[2,3] + V3 (=2+3) }	0	0	0	0	0	0	0
= 5							
Decision: Select item 3							

	W = 0	1	2	3	4	5	6
Backtracking	0	0	2	3	4	5	5
OPT[3,6] = max{	0	0	2	2	4	4	4
OPT[2,6] (=4),	0	0	2	2	2	2	2
OPT[2,3] + V3 (=2+3) }	0	0	0	0	0	0	0
= 5							
Decision: Select item 3							

OPT[2,3] = max{
 OPT[1,3] (=2),
 OPT[1,1] + V2 (=0+2) }
 = 2

Decision: Select item 2

Backtracking: step 2

最后我们得到 $dp[3][6] = 5$ ，那么如何找到最优解方案呢。

根据之前我们做矩阵链式乘法的经验知道通过对中间过程进行记录，一步步进行回溯找到该最优解方案。

3.2.3 时间复杂度的讨论：

该问题的时间复杂度为 $O(nW)$ 。因为原问题有 nW 个子问题，每个子问题进行 2 次比较，则共有 $O(2nW)$ 运算次数，因此原问题的时间复杂度就是 $O(nW)$ 。

如果 W 过大，则该算法将不是很高效。因为数值是用 2 进制表示的，则 W 将使用 $\log W$ 个 bit 表示，则原问题的时间复杂度为 $O(n2^{\log W}) = O(n2^{input\ length})$ ，是与输入长度相关的指数表达式，这种形式的表达式叫做伪多项式时间的算法。

3.2.4 另外一个子问题表示：

开始我们假设所有的问题给了一个排序。现在我们假设对 n 个物品不排序，则子问题可以定义成 $OPT(s, W)$ 即在 S 这个物品集合里，当包的容量最大是 W 时如何使装的物品的价值最大。

现在考察包里面的任何一个物品，则该物品有装和不装两种情况，因此有递归表达式

$$OPT(S, W) = \max\{OPT(S - \{i\}, W - w_i) + v_i, OPT(S - \{i\}, W)\}$$

这种思路在理论上是 ok 的，但是时间复杂度会非常高。因为任何一个物品的子集我们都要作为 S 进行求解，如果有 n 个物品，则有 2^n 个子集，则问题的时间复杂度将是指数级的。

而我们之前定义的 $OPT(i, w)$ 是指在前 i 个物品中选且包的容量最大为 w 时的装的物品价值最大。此时只考虑前 i 个物品，并不考虑前 i 个物品到底是哪 i 个物品，则该子问题只有 $O(n)$ 个子问题，可见之前对问题的表示比上述表示要简洁的多。

可见子问题的表示对动态规划的时间复杂度有很重要的影响，如果定义的不够好将导致时间复杂度特别高。

3.3 序列的连配问题

3.3.1 问题描述:

关于序列连配问题的实际需求比较多，比如生物信息领域中 DNA 序列的匹配问题。生活中常见的例子就是我们在 word 里面打英文单词，如果有错误，它会划红线进行提醒，有时甚至可以自动修改。

关于判断输入的英文单词是否错误，我们可以在系统后台存储一个词典。如果打出的英文单词不在该词典中，则该词就判断为错误，但是如何自动化修改呢？如何知道该词与词典中的哪个词最近呢？比如我们敲入的是 teh 这个单词，如何自动修改成 the 这个单词呢？

关键问题在于如何评判键盘敲入的单词与词典背后的单词的相似程度呢？比如我们敲入的是 OCURRANCE 这个单词，我们通过插入一个字母 C 和修改 A 这个字母为 E 变成词典中的 OCCURRENCE 这个单词，即我们可以经过有限次的修改，删除，添加操作，可以将敲出的单词变成词典中某个正确的单词。

3.3.2 形式化定义:

因此我们引出序列连配问题的形式化定义：

- **Input:**

Two sequence S and T , $|S| = m$, and $|T| = n$;

- **Output:**

To identify an alignment of S and T that maximizes a scoring function.

Note: for the sake of simplicity, the following indexing schema is used: $S = S_1S_2\dots S_m$.

这里 Alignment 是左右对齐的意思，经常表示产生式过程，表示上面的单词 S 是怎样通过下面的单词 T 变成的。

我们的目标就是通过添加空格使上下序列一样长，例如在 S 中添加一些空格变成 S' ， T 中添加一些空格变成 T' 。

如果变了之后 T ‘的第 i 个是空格，则表示 S 该位置上的字母是 T 通过插入操作得到的。

如果变了之后 S ‘的第 i 个是空格，则表示 S 是将 T 中该位置的字母删除了。

因此 Alignment 就是记录了 T 是经过怎样一系列的变化变成 S 的。

现在假设我们有两个序列：S : teh 和 T: the

我们认为敲的 S 与字典里面的 T 是最相似的。为什么最相似呢，肯定是我们对单词之间的相似度进行了打分，那怎么打分的呢？

我们假设有如下打分原则：

$$d(S, T) = \sum_{i=1}^{|S'|} \delta(S'[i], T'[i])$$

这里的 $\delta(a, b)$ is:

1. Match: +1 , e.g. $\delta('C', 'C') = 1.$
2. Mismatch: -1, e.g. $\delta('E', 'A') = -1.$
3. Ins/Del: -3, e.g. $\delta('C', '-') = -3.$

$\delta(a, b)$ 表示如果当前 S 位置上的字母与 T 位置上的字母 match 到了，则 +1 分，如果没有 match 到则 -1 分，少敲则 -3 分。这里这样设计打分原则只是为了教学方便。

一个问题可以建模成组合优化或者统计问题，这两个问题是密切相关的。

使用 Alignment，我们可以知道你最想敲的是什么。我们系统后台有个词典，词典中的每个单词都与你想敲的单词进行相似度计算，则找到与你敲的单词最相似的就是你最想敲的，也就是得分最高的那个单词。

具体事例：

例如下面一个例子：

- 1. $T = "OCCUPATION": d(S', T') = 1+1-3+1-3-3-1+1-3-3-3+1-3-3 =$

S' : OC_URRA_NCE
 | | | |
 T' : OCCU_PATION__

S' : O_CURRANCE
 | | | | | | |
 T' : OCCURRENCE

-28.

2. $T = \text{"OCCURRENCE"}: d(S', T') = 1 - 3 + 1 + 1 + 1 + 1 - 1 + 1 + 1 + 1 = 4.$

- 我们对单词“OCCUPATION”和“OCURRANCE”通过加空格使得它们等长，然后通过计分的方案得到“OCCUPATION”与“OCURRANCE”的得分是 -28 分。

同样假设我们在词典中遇到了另外一个词“OCCURRENCE”，和上面一样操作，最后得到“OCCURRENCE”与“OCURRANCE”的得分是 4 分。

S' : O_CURRANCE

| | | | | | |

T' : OCCURRENCE

S' : O_CURR_ANCE

| | | | | | |

T' : OCCURRE-NCE

因为单词“OCCURRENCE”得分比较高，所以我们猜测敲错的单词“OCURRANCE”极有可能是从“OCCURRENCE”这个单词过来的。

另外我们还能知道你是怎么敲的？

即使我们找到与“OCURRANCE”最相似的是“OCCURRENCE”这个单词，我们仍然有很多种加空格的方案使得他们等长。

- 1. Alignment 1: $d(S', T') = 1 - 3 + 1 + 1 + 1 + 1 - 1 + 1 + 1 + 1 = 4.$
- 2. Alignment 2: $d(S', T') = 1 - 3 + 1 + 1 + 1 + 1 - 3 - 3 + 1 + 1 + 1 = -1.$
- 这两种加空格的方案都可以使得单词“OCURRANCE”与“OCCURRENCE”等长，但是得分却是不同的。通过 Alignment 我们可以知道单词“OCURRANCE”中的字母 A 最可能是把字母 E 敲错了得到的，而不是少敲了一个 E 且多敲了一个 A 得到的。

因此我们知道 Alignment 十分重要，不仅可以识别出字典中哪个单词与你敲的单词相似度最大，而且可以推测出你是怎么敲错的。

3.3.3 动态规划求解：

现在我们对问题进行总结：

给我们两个字符串 S 与 T，问如何通过加空格使得得分最高。

采用我们之前求解问题的一般思路：首先给我们的字符串很长，我们不好解决，我们可以将该字符串分成更小的字符串，我们的 solution 是通过加空格使两个字符串对齐的方案。我们把求解过程当做一系列的决策，对每个决策部分我们决定 $s[i]$ 是怎样通过 $T[j]$ 变来的，比如 OCURRANCE 是 S, OCCURRENCE 是 T，我们尝试考虑 S 是经过什么样的操作变化到 T 的。

我们首先考虑 S 单词最后的字母 E，那 S 的 E 是通过 T 怎么变化得到的呢？

- (1) 从 T 的最后一个单词直接过来的，即 match 到了。
- (2) 这个 E 是我们多敲的，即 T 通过插入操作变化过来的。
- (3) 这个 E 是我们少敲的，即 T 通过删除操作变化过来的。

我们的总体目标是给我们一个词，我们想知道该词是否与词典中的某个词相似，给我们的词太长不好解决，可以先考虑最后一个字母，看它有哪些来源。

如果是多敲的，则 S 除了最后一个字母剩余的部分，是由 T 的整体变化而来的。则问题变成假设我们找到一个最优解，现在我们只考虑最后一个决策，即 S 的最后一个字母是怎么过来的呢？我们由上面的分析知道，其来源三种可能。

- (1) 如果 $S[m]$ 与 $T[n]$ 形成匹配，则子问题变成了 $S[1, \dots, m-1]$ 与 $T[1, \dots, n-1]$ 的对齐问题。
- (2) 如果 $S[m]$ 与空格匹配，则表示 $S[m]$ 是 T 通过插入操作过来的，则子问题变成了 $S[1, \dots, m-1]$ 与 $T[1, \dots, n]$ 的对齐问题。
- (3) 如果 $T[n]$ 与空格匹配，则表示 $S[m]$ 是 T 通过删除操作过来的，则子问题变成了 $S[1, \dots, m]$ 与 $T[1, \dots, n-1]$ 的对齐问题。

则我们可以总结得到子问题的一般形式就是 S 的前缀与 T 的前缀的对齐方案，我们将该问题的最优解记做 $OPT(i, j) = S[1 \dots i] alignment T[1 \dots j]$.

因此我们可以得到如下递归表达式：

Score: $d("OCU", "") = -9$	Score: $d("", "OC") = -6$
Alignment: $S' = OCU$	Alignment: $S' = --$
T' = ---	T' = OC

$$OPT(i, j) = \max \begin{cases} \delta(S_i, T_j) + OPT(i - 1, j - 1) \\ \delta(' ', T_j) + OPT(i, j - 1) \\ \delta(S_i, ' ') + OPT(i - 1, j) \end{cases}$$

即枚举当前单元的三种可能来源，在其中取最大值就可以。

因此有如下伪代码：

```

NEEDLEMAN_WUNCH( $S, T$ )
1: for  $i = 0$  to  $m$ ; do
2:    $OPT[i, 0] = -3 * i;$ 
3: end for
4: for  $j = 0$  to  $n$ ; do
5:    $OPT[0, j] = -3 * j;$ 
6: end for
7: for  $i = 1$  to  $m$  do
8:   for  $j = 1$  to  $n$  do
9:      $OPT[i, j] = \max\{OPT[i - 1, j - 1] + \delta(S_i, T_j), OPT[i - 1, j] - 3, OPT[i, j - 1] - 3\};$ 
10:  end for
11: end for
12: return  $OPT[m, n]$  ;

```

Note: the first row is introduced to describe the alignment of prefixes $T[1..i]$ with an empty sequence ϵ , so does the first column.

我们通过伪代码可以得到下面的表格， S 中每个字母一列， T 中每个字母一行，其中单元 (i, j) 表示 $S[1, \dots, j]$ 与 $T[1, \dots, i]$ 对齐所能得到的最大的分数，比如图中的单元 (1,2):

S:	'	O	C	U	R	R	A	N	C	E
T:	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
O	-3									
C	-6									
U	-9									
R	-12									
R	-15									
E	-18									
N	-21									
C	-24									
E	-27									
	-30									

S:	'	O	C	U	R	R	A	N	C	E
T:	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
O	-3	1	-2	-5	-8	-11	-14	-17	-20	-23
C	-6	-2	2	-1	-4	-7	-10	-13	-16	-19
U	-9	-5	-1	1	-2	-5	-8	-11	-12	-15
R	-12	-8	-4	0	0	-3	-6	-9	-12	13
R	-15	-11	-7	-3	1	1	-2	-5	-8	-11
E	-18	-14	-10	-6	-2	2	-	-3	-6	-9
N	-21	-17	-13	-9	-5	-1	1	-1	-4	-5
C	-24	-20	-16	-12	-8	-4	-2	2	-1	-4
E	-27	-23	-19	-15	-11	-7	-5	-1	3	0
	-30	-26	-22	-18	-14	-10	-8	-4	0	4

Score: $d("OC", "O") = \max \begin{cases} d("OC", "") & -3 \quad (= -9) \\ d("O", "") & -1 \quad (= -4) \\ d("O", "O") & -3 \quad (= -2) \end{cases}$

Alignment: $S' = OC$

$T' = O-$

S:	'	O	C	U	R	R	A	N	C	E
T:	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
O	-3	1	-2	-5	-8	-11	-14	-17	-20	-23
C	-6	-2	2	-1	-4	-7	-10	-13	-16	-19
U	-9	-5	-1	1	-2	-5	-8	-11	-12	-15
R	-12	-8	-4	0	0	-3	-6	-9	-12	13
R	-15	-11	-7	-3	1	1	-2	-5	-8	-11
E	-18	-14	-10	-6	-2	2	-	-3	-6	-9
E	-21	-17	-13	-9	-5	-1	1	-1	-4	-5
N	-24	-20	-16	-12	-8	-4	-2	2	-1	-4
C	-27	-23	-19	-15	-11	-7	-5	-1	3	0
E	-30	-26	-22	-18	-14	-10	-8	-4	0	4

同样的我们可以得到最后一个单元的来源情况。

单元 (1,2) 表示将 T 的前缀 O 敲成 S 的前缀 OC 时的 Alignment 情况，根据之前的分析我们知道有上述三个来源，即 (1, 2) 单元仅与 (1, 1), (0, 2), (0, 1) 这三个单元有关。

$$\text{Score: } d(\text{"OCURRANCE"}, \text{"OCCURRENCE"}) = \max \left\{ \begin{array}{ll} d(\text{"OCURRANCE"}, \text{"OCCURRENC"}) & -3 \\ d(\text{"OCURRANC"}, \text{"OCCURRENC"}) & +1 \\ d(\text{"OCURRANC"}, \text{"OCCURRENCE"}) & -3 \end{array} \right.$$

Alignment: $S' = \text{O-CURRANCE}$

S:	'	O	C	U	R	R	A	N	C	E
T:	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
O	-3	1	-2	-5	-8	-11	-14	-17	-20	-23
C	-6	-2	2	-1	-4	-7	-10	-13	-16	-19
C	-9	-5	-1	1	-2	-5	-8	-11	-12	-15
U	-12	-8	-4	0	0	-3	-6	-9	-12	13
R	-15	-11	-7	-3	1	1	-2	-5	-8	-11
R	-18	-14	-10	-6	-2	2	-	-3	-6	-9
E	-21	-17	-13	-9	-5	-1	1	-1	-4	-5
N	-24	-20	-16	-12	-8	-4	-2	2	-1	-4
C	-27	-23	-19	-15	-11	-7	-5	-1	3	0
E	-30	-26	-22	-18	-14	-10	-8	-4	0	4

从上面的例子我们可以知道中间的单元都是由相邻的三个单元变换而来的，那么第一行和第一列是怎么来的呢？

第一行是说 T 是空时，是怎么一步步从 "" 变化到 $S[1,..,n]$ 的。相当于每次都插入一个字母变成与 S 对应的字母，即每增加一个字母就扣三分。

第一列是说字典里有个词 T，你敲成空字符串了，即表示 T 是怎么样变化到空字符串的，所以每次都少敲了，因此将 T 变成 S 需要每次都删除一个字母，即每删除一个字母就扣三分。

3.3.4 构建最优解方案：

那么我们如何知道最后的得分是怎么来的呢？常用的想法就是回溯，我们的最后的得分 4 本身有三个来源，我们开个表格记录该单元的得分是从三个来源中具体的哪个单元过来的，不断的向前回溯就可以找到一个对应的 alignment，使得 S 变成 T。

但是我们经常对分的评判不是很准确，这时候可以随机回溯多次，然后取平均，这样可以取一个比较常见的模式，使得我们最终得到的结果更稳定些。

现在对我们的问题再陈述下：我们在文档里面敲了一个词，该词不在词典中，这时候我们知道自己敲错了，但是我们想知道是怎么错的，我们想用这个分来衡量词典中的词经过最少的几次操作才能变成我们敲的单词 S 呢。

第四章 动态规划 (2)

4.1 上节课回顾

上堂课动态规划，我们讲到如果一个大问题可以分解成为小的问题，且有最优子结构的性质，这种时候可以用尝试动态规划。动态规划最简单的例子是矩阵的链式乘法，它是一个序列乘的问题。这个问题的划分可以有很多种，我们不知道怎么分，所以使用枚举。上次我们还讲到单词出错如何进行更改，以及作业查重的方法。这个算法的问题在于需要构造很大的矩阵，比如一篇文章有 10K 字符，矩阵就是 $10K \times 10K$ 这么大，这个空间开销是非常大的。

4.2 Hischberg 算法

4.2.1 第一个观察

为了解决这个问题，Hischberg 在 1975 年提出了一个非常精彩的算法，可把空间开销变成 $O(m + n)$ 。该算法核心在三个观察。首先，对于两个字符串，如果只关心最后的分，不需要用到全部的矩阵，只要用到两列就可以。比如对于下面这个图

S:	'	O	C	U	R	R	A	N	C	E
T:	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
O	-3	1	-2	-5	-8	-11	-14	-17	-20	-23
C	-6	-2	2	-1	-4	-7	-10	-13	-16	-19
U	-9	-5	-1	1	-2	-5	-8	-11	-12	-15
R	-12	-8	-4	0	0	-3	-6	-9	-12	13
R	-15	-11	-7	-3	1	1	-2	-5	-8	-11
E	-18	-14	-10	-6	-2	2	-	-3	-6	-9
E	-21	-17	-13	-9	-5	-1	1	-1	-4	-5
N	-24	-20	-16	-12	-8	-4	-2	2	-1	-4
C	-27	-23	-19	-15	-11	-7	-5	-1	3	0
E	-30	-26	-22	-18	-14	-10	-8	-4	0	4

我通过初始化知道了最左边一列的分，然后下一列的分怎么算？我们知道任意一个单元的分依赖于临近的三个单元。所以，知道第一列以后，我们可以算出第二列。这时候，我只需要放弃第一列空间，计算第三列，以此类推，我们只是依赖了两个数组，就可以计算到最后的分。下面的图展现了计算过程。

S:	'	O	C	U	R	R	A	N	C	E
T:	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
O	-3	1	-2	-5	-8	-11	-14	-17	-20	-23
C	-6	-2	2	-1	-4	-7	-10	-13	-16	-19
U	-9	-5	-1	1	-2	-5	-8	-11	-12	-15
R	-12	-8	-4	0	0	-3	-6	-9	-12	13
R	-15	-11	-7	-3	1	1	-2	-5	-8	-11
E	-18	-14	-10	-6	-2	2	-	-3	-6	-9
E	-21	-17	-13	-9	-5	-1	1	-1	-4	-5
N	-24	-20	-16	-12	-8	-4	-2	2	-1	-4
C	-27	-23	-19	-15	-11	-7	-5	-1	3	0
E	-30	-26	-22	-18	-14	-10	-8	-4	0	4

S:	'	O	C	U	R	R	A	N	C	E	
T:	'	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
O	-3	1	-2	-5	-8	-11	-14	-17	-20	-23	
C	-6	-2	2	-1	-4	-7	-10	-13	-16	-19	
C	-9	-5	-1	1	-2	-5	-8	-11	-14	-15	
U	-12	-8	-4	0	0	-3	-6	-9	-12	13	
R	-15	-11	-7	-3	1	1	-2	-5	-8	-11	
R	-18	-14	-10	-6	-2	2	-	-3	-6	-9	
E	-21	-17	-13	-9	-5	-1	1	-1	-4	-5	
N	-24	-20	-16	-12	-8	-4	-2	2	-1	-4	
C	-27	-23	-19	-15	-11	-7	-5	-1	3	0	
E	-30	-26	-22	-18	-14	-10	-8	-4	0	4	

对于上面的观察，我们可以写一个算法计算最后的分

PREFIX_SPACE_EFFICIENT_ALIGNMENT(S, T, SCORE)

```

1: for  $i = 0$  TO  $m$  do
2:    $score[i] = -3 * i;$ 
3: end for
4: for  $i = 1$  TO  $m$  do
5:   for  $j = 1$  TO  $n$  do
6:      $newscore[j] = \max\{score[j - 1] + \delta(S_i, T_j), score[j] - 3, newscore[j - 1] - 3\};$ 
7:   end for
8:    $newscore[0] = 0;$ 
9:   for  $j = 1$  TO  $n$  do
10:     $score[j] = newscore[j];$ 
11:   end for
12: end for
13: return  $score[n]$  ;

```

4.2.2 第二个观察

我们刚才做的算法，是从左往右的，都是 S 的前缀和 T 的前缀进行计算。但是其实也是可以 S 的后缀进行计算。一开始我问的是，S 的最后一个字符 E 是怎么来的。我们定义的子问题用到了 S 的前缀和 T 的前缀。当基于 S 和 T 的后缀定义子问题的时候，我们问的是：S 的第一个字符是怎么来的？这也会有三种情况。这样子问题，就变成 S 和 T 的后缀的最优连配。这时候，矩阵要变成从下往上，从右往左开始算。算到最后我们发现分数是一样的，也是 4。

4	0	-4	-10	-12	-16	-18	-22	-26	-30
5	3	-1	-7	-9	-13	-15	-19	-23	-27
3	6	2	-4	-6	-10	-12	-16	-20	-24
-1	2	5	-1	-3	-7	-9	-13	-17	-21
-5	-2	1	4	0	-4	-6	-10	-14	-18
-9	-6	-3	0	3	-1	-3	-7	-11	-15
-13	-10	-7	-4	-1	2	0	-4	-8	-12
-15	-12	-9	-6	-3	0	3	-1	-5	-9
-19	-16	-13	-10	-7	-4	-1	2	-2	-6
-23	-20	-17	-14	-11	-8	-5	-2	1	-3
-27	-24	-21	-18	-15	-12	-9	-6	-3	0

O
C
C
U
R
R
E
N
C
E
T
S

但是如果需要知道某个字符是在哪里进行插入和删除的，按照过去的方法，我们需要一个矩阵记录回溯的信息。但是如果只开两个数组，我们没有办法做到这点。所以我们只可以算分，但是不知道如何回溯，应该怎么办？

4.2.3 第三个观察

接下是最重要的技巧：假如已经有了最优的连配，我们问 S 最中间的字符是 align 到谁？比如说对这里的 S 和 T，我会问 R 从哪里来的？我们把 S 分成两半，即蓝框和红框，那么前一部分总是和 T 的一部分 align。

$\frac{m}{2}$

S: OCURRANCE

T: OCCURRENCE
 $1 \leq q \leq n$

我们有这个式子：

$$OPT(S, T) = OPT(S[1.. \frac{m}{2}], T[1..q]) + OPT(S[\frac{m}{2}+1..m], T[q+1..n])$$

假设我们有了最优连配，其总体的分等于左一半的分和右一半的分相加；我们假设知道前一半是 *align* 到具体的哪个 q 那么就可以直接算。假如我们知道 q ，那么一切都有了，那么 q 到底是什么？

4.2.4 算法

我们来看这个算法

LINEAR_SPACE_ALIGNMENT(S, T)

- 1: ALLOCATE TWO ARRAYS f AND b ; EACH ARRAY HAS A SIZE OF m .
- 2: PREFIX_SPACE_EFFICIENT_ALIGNMENT($S, T[1..\frac{n}{2}], f$);
- 3: SUFFIX_SPACE_EFFICIENT_ALIGNMENT($S, T[\frac{n}{2}+1, n], b$);
- 4: LET $q^* = argmax_q(f[q] + b[q])$;
- 5: FREE ARRAYS f AND b ;
- 6: RECORD $IN ARRAY A ;$
- 7: LINEAR_SPACE_ALIGNMENT($S[1..q^*], T[1..\frac{n}{2}]$);
- 8: LINEAR_SPACE_ALIGNMENT($S[q^*+1..n], T[\frac{n}{2}+1, n]$);
- 9: **return** A ;

我们先申请两个数组，一个是 f ，一个是 b 。 f 代表前向数组， b 代表后向数组。首先对 S 的左一半和 T 整体进行 *align*，结果放到 f 中， S 的右一半和 T 进行 *align*，结果放到 b 中。 S 的一半，*align* 到 T 的那个 q 呢？我们发现 q 是把 f 和 b 加起来，之间最大的那个数的位置。

S: ""	O	C	U	R								
T:	0	-3	-6	-9	-12	-24	-12	-16	-18	-22	-26	-30
" "	-3	1	-2	-5	-8	-17	-9	-13	-15	-19	-23	-27
O	-6	-2	2	-1	-4	-10	-6	-10	-12	-16	-20	-24
C	-9	-5	-1	1	-2	-5	-3	-7	-9	-13	-17	-21
C	-12	-8	-4	0	0	0	0	-4	-6	-10	-14	-18
U	-15	-11	-7	-3	1	4	3	-1	-3	-7	-11	-15
R	-18	-14	-10	-6	-2	-3	-1	2	-4	-8	-12	
R	-21	-17	-13	-9	-5	-8	-3	0	3	-1	-5	-9
E	-24	-20	-16	-12	-8	-15	-7	-4	-1	2	-2	-6
N	-27	-23	-19	-15	-11	-22	-11	-8	-5	-2	1	-3
C	-30	-26	-22	-18	-14	-29	-15	-12	-9	-6	-3	0
E							R	A	N	C	E	""
							f	f+b	b			S

S 的左一半和 T 的 OCCUR 部分做 align, 也即我们想敲 T 的 OCCUR 字符是敲成了 OCUR 这样。在上面的算法中, 我们知道了 q_* , 记录 $S < q, n/2 >$, 表示 $n/2$ 是从 q 这里变来的, 剩下的我们递归调用即可。

S: ""	O	C	U	R								
T:	0	-3	-6	-9	-12	-24	-12	-16	-18	-22	-26	-30
" "	-3	1	-2	-5	-8	-17	-9	-13	-15	-19	-23	-27
O	-6	-2	2	-1	-4	-10	-6	-10	-12	-16	-20	-24
C	-9	-5	-1	1	-2	-5	-3	-7	-9	-13	-17	-21
C	-12	-8	-4	0	0	0	0	-4	-6	-10	-14	-18
U	-15	-11	-7	-3	1	4	3	-1	-3	-7	-11	-15
R	-18	-14	-10	-6	-2	-3	-1	2	-4	-8	-12	
R	-21	-17	-13	-9	-5	-8	-1	2	-4	-8	-12	
E	-24	-20	-16	-12	-8	-15	-3	0	3	-1	-5	-9
N	-27	-23	-19	-15	-11	-22	-7	-4	-1	2	-2	-6
C	-30	-26	-22	-18	-14	-29	-11	-8	-5	-2	1	-3
E							-15	-12	-9	-6	-3	0
							R	A	N	C	E	""
							f	f+b	b			S

4.2.5 算法总结

这样的话这个算法的思想分为三点:

- 只用两个数组就可以得到最后的分数。

2. 可以从前往后算也可以从后往前算，最终结果是一样的。
3. 假设知道了最优的连配，那么 S 的左一半和右一半是从哪里得来的呢？我们假设这个位置是 q ，并且提供了定位的方法。

这个算法声称解决了动态规划的空间消耗问题，其使用了 $O(m+n)$ 的空间。

那么这个算法会不会增加了时间？过去的算法是 $O(m^*n)$ ，这是因为 m^*n 个单元，每个单元是从三个数中取最大，每个要进行三次比较所以是 $3mn$ 。这个新算法还是 $O(m^*n)$ 的时间，怎么证明？这个超过了第一堂课递归主定理的范畴，因为这里的递归调用依赖于 S 的左一半和 T 的前一半，前一半到哪里不能事先知道。那应该怎么办？我们采用连蒙带猜的方法，猜并且带入验证。我们设 $T(m,n)$ 是 $O(mn)$ 的时间，首先假设 $m' < m \ n' < n$

则 $T(m',n') \leq km'n'$ 对于任意的 $m' < m$ AND $n' < n$ 成立。故而有以下证明：

$$T(m,n) = cm + T(q, \frac{n}{2}) + T(m-q, \frac{n}{2}) \quad (4.2.1)$$

$$\leq cm + kq\frac{n}{2} + k(m-q)\frac{n}{2} \quad (4.2.2)$$

$$= cm + kq\frac{n}{2} + km\frac{n}{2} - kq\frac{n}{2} \quad (4.2.3)$$

$$\leq (c + \frac{k}{2})mn \quad (4.2.4)$$

$$= kmn \quad (\text{set } k = 2c) \quad (4.2.5)$$

这个递归的证明和分治的时候不同，每个子问题的大小，即 q 的大小不知道，以后类似的问题可以这么做。另外，我们通过这个例子知道知道动态规划的内存是可以降下来的。

4.3 alignment 问题的四个扩展

4.3.1 第一个扩展

关于这个联配问题，有四个扩展，第一点是全局的连配到局部的连配，全局的连配是两个单词进行连配，但是这个是不够的。之前的算法只适合单词改错，不适合判断抄袭。比如说我只有一道题是抄的，其他部分我都很老实，这样总体的分还是比较低。只关心其中部分的分，就是局部的连配。这是 SMITH-WATERMAN 在 1981 年做的工作。

局部的连配公式和原来的区别在于在原来的基础上加了 0，也就是一旦也就是罚的分够狠，小于 0，就重新从 0 开始。以前的分是指总体的 S 来自于 T 的概率，由于抄袭只有一部分，应该只关心一部分的分。

局部连配公式：

$$d(S, T) = \max \left\{ \begin{array}{l} \delta(S_n, T_n) + d(S[1..n-1], T[1..n-1]) \\ \delta(' \text{---}', T_n) + d(S, T[1..n-1]) \\ 0 \\ \delta(S_n, ' \text{---}') + d(S[1..n-1], T) \end{array} \right.$$

全局连配的公式：

$$d(S, T) = \max \left\{ \begin{array}{l} \delta(S_n, T_n) + d(S[1..n-1], T[1..n-1]) \\ \delta(' _ ', T_n) + d(S, T[1..n-1]) \\ \delta(S_n, ' _ ') + d(S[1..n-1], T) \end{array} \right.$$

4.3.2 第二个扩展

上次说的罚分，实际使用的时候，如果 $match +1$ ，如果是插入 -3，删除 -3， $mismatch -1$ 。上堂课我们讲为什么不是扣 3.1415926 分，这是大家需要考虑的。实际工作中会用下面这个表格，一个字母变成另外一个字母的分是通过概率统计出来的，如 A 变成 R 表示 -2 分。这些是怎么来的呢？我们统计这样一个概率：P('A' | 'A')，然取 \log ，然后取整，就是这个表格里的数。如果我们做的算法不结合统计，是得不到好的效果的。

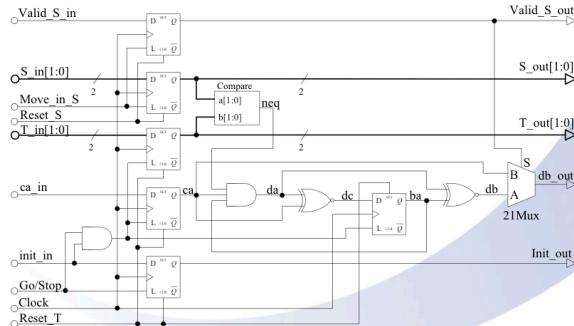
4.3.3 第三个扩展

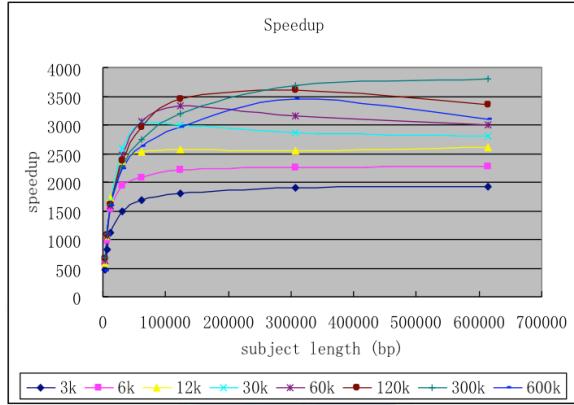
刚才的例子，得到的分数是 4，4 分是高还是低，如何评判分高和低？我们提出 $P(S, T|RANDOM)$ ，即想写 T 的时候，写成 S 的概率有多大。我们有这样的计算公式：

分数大于 S 的概率是 $1 - e^{-y}$ ，其中 $y = Kmne^{-\lambda S}$. 具体可以参考论文。

4.3.4 第四个扩展

最后一个扩展是我的工作：这个连配的算法非常简单，只是三个数求 MAX，不需要用 CPU 来算，并且可以同时算有并行性。所以做了一个 FPGA 的卡来实现这个算法，从图中可以看出，对于 1000K 的字符串，一个卡能顶 1000 多个 CPU，以后当我们从算法设计的角度无法提高性能的时候，可以考虑这种方法。





4.4 图上递归问题

我们对数据结构进行分类

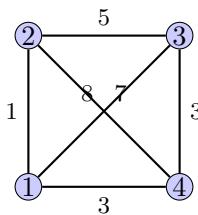
第一类是 *sequences*(数组), 比如第一堂课讲的 N 个数排序, 可以分成左一半, 右一半。

第二类是 *graph*(图), 图的特例是树, 图的递归的例子是旅行商问题。

第三类是 *set*(集合)。总体上来看, 我们在这三种数据结构上进行递归, 我们接下来讲图上递归的问题。

4.4.1 TSP 问题

TSP 是一个图上递归的例子: 我们来回顾一下 TSP 问题。

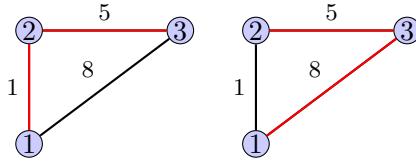


给定图, 从 1 号节点, 经过所有节点回到 1, 找路径最短。假设了从 1 号节点出发。要考虑这个问题, 只要考虑一个跟它相关的问题: 我们定义 $D(S, e)$, 这是一个子问题, 表示我们从 1 出发, 旅行过 S 中所有的节点, 到达 e 距离最短, 这个

距离是 $D(S, e)$ 。为什么要解决这个子问题？因为只要解决了这个子问题，就能解决前面的问题：因为我们是想从 1 出发，经过所有城市，最后回到 1。那么是从哪里回来的？可能是 2 3 4。最后的旅程可以这么表示：

$$\begin{aligned} & \min\{D(\{2, 3, 4\}, 2) + d_{2,1}, \\ & D(\{2, 3, 4\}, 3) + d_{3,1}, \\ & D(\{2, 3, 4\}, 4) + d_{4,1}\} \end{aligned}$$

但是这个要怎么算？我们还是老的套路，先从最简单的例子来看，如果 S 只包含一个或者两个城市，会不会很简单？我们在这种情况下求最小就可以了。怎么做呢？

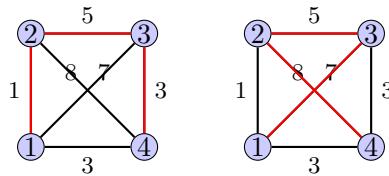


我们有如下结论：

$$D(\{2\}, 2) = d_{12}; D(\{3\}, 3) = d_{13};$$

这个是显然的； D_2 表示经过 S 中的所有节点一次，最后到达 2。我们能解决简单的问题以后，复杂的问题该怎么办？比如 $D(\{1, 2, 3, 4\}, 4)$ ？

这里最终到达 4，要么从 2 号来，要么从 3 号来，我们有了最优子结构的式子：



- $D(\{1, 2, 3, 4\}, 4) = \min\{D(\{1, 2, 3\}, 3) + d_{34}, D(\{1, 2, 3\}, 2) + d_{24}\};$

- 最优子结构性质：

$$D(S, e) = \begin{cases} d_{1e} & \text{IF } S = \{e\} \\ \min_{m \in S - \{e\}} (D(S - \{e\}, m) + d_{me}) & \text{OTHERWISE} \end{cases}$$

有了这个递归表达式，就可以写一个算法如下：

```

function D(S, e)
1: if S = {e} then
2:   return d1e;
3: end if
4: d = ∞;
5: for all city m ∈ S, and m ≠ e do
6:   if D(S - {e}, m) + dme < d then
7:     d = D(S - {e}, m) + dme;
8:   end if
9: end for
10: return d;

```

这是图上递归的例子，我原来是 4 个节点的图 G，通过递归把图变小了。过去的递归都是在数组上比较好理解，现在是一个图，对于一个四个节点的图不会做，可以变成三个节点... 但是这个算法性能不怎么好。这个算法的时间复杂度和空间复杂度如下：

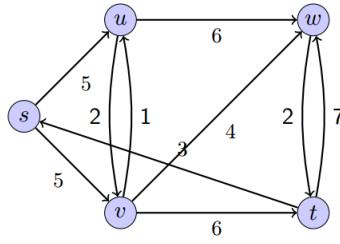
- 空间复杂度： $\sum_{k=2}^{n-1} k \binom{n-1}{k} + n - 1 = (n-1)2^{n-2}$
- 时间复杂度： $\sum_{k=2}^{n-1} k(k-1) \binom{n-1}{k} + n - 1 = O(2^n n^2)$.

我们先说空间复杂度：我们要枚举所有的子图，它有多少个呢？我们有 D(S,e), s 属于 1,2, ...n,

e=1,2,3...; 我们考虑子图规模为 k, k 取 1 到 n-1。对于规模为 k 的子图，其有 k 个子问题。我们最终算出来是 2 的 n 次方这个量级。中间的每个子问题都要存下来。那么时间复杂度呢？对于动态规划的算法的时间复杂度，我们看有多少个子问题，以及每个子问题要做多少次运算。由于我们要对 all city 求最小，所以我们在时间复杂度的式子基础上乘 k-1，结果是 $O(2^n n^2)$ 。

4.4.2 单源最短路问题

我们接着往下看，还是在图上做递归的改进方法：单源最短路问题，它加了一点东西做改进。



我们想求从 S 到 T 的最短路径，这个问题和 TSP 的不同在于不需要每个城市都走到（假设没有负圈）。我们先来尝试定义子问题。

- 从 S 到 T 的路径，我们把这个过程看成一系列的决策，在每一个决策步考虑下一步往哪里走。
- 假如已经拿到了最优解 O，考察 O 中的第一个决策：所有 s 的邻居都是可选项，这样选择了从 s 到 v 的一条路。
- 剩下的问题是，从 v 中怎么达到 t，路径越短越好；

这样，图变小了，定义了子问题。

但是按照这个递归表达式写程序有点慢，因为子问题的数量太多了，图上的递归，是指数级的，跟前面的问题遇到了相同的困难。所以我们做动态规划的时候要注意，有的时候定义的子问题不是特别合适，导致子问题数目为指数级。我们做了如下改进：

引入了新的观察，因为图中没有负圈，从 S 到 T 最短路最多只有 n 个节点。假设我们拿到了最优解 O，考虑 O 中的第一个决策。所有和 s 直连的点都有可能。当决定了第一步以后，问题变成从 v 到 t 的最短路最多经过 n-2 条边。从而我们修改了子问题，写出了最优子结构：

$$OPT[v, t, k] = \min \begin{cases} OPT[v, t, k - 1], \\ \min_{\langle v, w \rangle \in E} \{ OPT[w, t, k - 1] + d(v, w) \} \end{cases}$$

由于上面定义的是最多 k 条边，那么可能只走了 k-1 条边，k-2 条边 ...，所以有了上面那一项。

算法如下 BELLMAN_FORD(G, s, t)

1: **for** ANY NODE $v \in V$ **do**

```

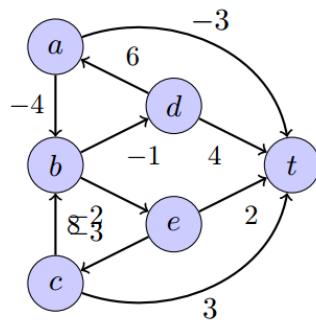
2:    $OPT[v, t, 0] = \infty;$ 
3: end for
4: for  $k = 0$  TO  $n - 1$  do
5:    $OPT[t, t, k] = 0;$ 
6: end for
7: for  $k = 1$  TO  $n - 1$  do
8:   for all NODE  $v$  (IN AN ARBITRARY ORDER) do
9:      $OPT[v, t, k] = \min \begin{cases} OPT[v, t, k - 1] \\ \min_{<v, w> \in E} \{OPT[w, t, k - 1] + d(v, w)\} \end{cases}$ 
10:  end for
11: end for
12: return  $OPT[s, t, n - 1];$ 

```

我们仔细看一下这个算法：我们先看第 7 行到第 10 行，第 9 行是我们的递归表达式。初始化阶段， v 到 t 经过 0 步无法到达，所以设置无穷。自己到自己，无论经过几步，都是 0。

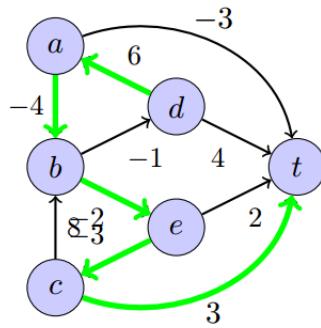
”Richard Bellman on the birth of dynamic programming” (S.DREYFUS, 2002) 大家有时间可以看一下这篇 paper，对理解动态规划很有帮助。

我们看一个例子，对这么一个道路交通网络，我们问到达 t 的最短路径是什么。



Source node	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
t	0	0	0	0	0	0
a	-	-3	-3	-4	-6	-6
b	-	-	0	-2	-2	-2
c	-	3	3	3	3	3
d	-	4	3	3	2	0
e	-	2	0	0	0	0

我们的子问题是，从任意一个节点出发，经过 k 步到达 t 的路程这样我们有了第一行和第一列。我们发现 a 到 t 可以直达，所以表格的 $(1,1) = -3$; 这个可以解释为 $\text{OPT}(a,t,1)=\min \text{OPT}(a,t,0)$, $\text{OPT}(w,t,0)+daw$ 这里的 $w=t$, 剩下的部分以此类推。



Source node	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
t	0	0	0	0	0	0
a	-	-3	-3	-4	-6	-6
b	-	-	0	-2	-2	-2
c	-	3	3	3	3	3
d	-	4	3	3	2	0
e	-	2	0	0	0	0

这幅图给大家扩展一个小知识，我们发现把所有的点到达 t 的最短路径都画出来，他们不会一个复杂的图，是一个 tree，这个将来会有用。

4.4.3 负圈判断问题

如果 v 可以到达 t ，从 v 到 t 有负圈，那么我么有这个式子： $\lim_{k \rightarrow \infty} OPT(v, t, k) = -\infty$ 。 k 越来越大，它会越来越小。我们给出下面这样一个图：

源点	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$	$k=11$
t	0	0	0	0	0	0	0	0	0	0	0	0
a	-	-3	-3	-4	-6	-6	-6	-6	-6	-6	-6	-6
b	-	-	0	-2	-2	-2	-2	-2	-2	-2	-2	-2
c	-	3	3	3	3	3	3	3	3	3	3	3
d	-	4	3	3	2	0	0	0	0	0	0	0
e	-	2	0	0	0	0	0	0	0	0	0	0

所以出现了图的负圈判断问题。如果里面没有负圈，我们可以得到这样一个结

论：对于 $k > n$, 结果是一样的。我们回到一个有负圈的图，如果我接着计算下去，可以发现值越来越小。所以判断有没有负圈，只要把这个 *Bellman–Ford* 算法多跑几次就可以。多跑几次，可以看到周期是 2，所以圈是三个节点。任意给我们一个图，问里面有没有负圈，应该怎么办？我们可以把这个图做一个扩展，加入一个节点，然所有的节点都指向它，边长为 0。然后我们运行上面的算法，如果有负圈，到 t 的最短路径会越来越小，这就是负圈的判断方法。

说到最短路径，大家第一个想到 *dijkstra*。这个算法有了，为什么还要用 *Bellman–Ford*? 我们来看算法在路由器上面的应用：有一个复杂的网络的图，中间每个节点是一个 *router*，我们如果在网络中找到 *google* 的最短路径我们如果用 *dijkstra* 算法，会有问题。找最短路径需要全局信息，但是这个信息是拿不到的。而 *Bellman–Ford* 算法只要知道 LOCAL 的信息就可以了。

下面是每台路由器上跑的程序，一开始是每台路由器只能自己到自己。任何一个路由器 w ，发现到达 *google* 有条近路，我就把这个消息告诉所有的邻居。我就把邻居 w 到 *google* 的距离 + 我到 w 的距离，和我过去到达 *google* 要花的时间求最小，并且更新。这里，任意一个路由器发现了最短路，只要告诉邻居就够了，从来不需要关心整个 *internet*。

```

ASYNCHRONOUSSHORTESTPATH( $G, s, t$ )
1: INITIALLY, SET  $OPT[t, t] = 0$ , AND  $OPT[v, t] = \infty$ ;
2: LABEL NODE  $t$  AS "ACTIVE";
3: while EXISTS AN ACTIVE NODE do
4:   ARBITRARILY SELECT AN ACTIVE NODE  $w$ ;
5:   REMOVE  $w$ 'S ACTIVE LABEL;
6:   for all EDGES  $< v, w >$  (IN AN ARBITRARY ORDER) do
7:      $OPT[v, t] = \min \begin{cases} OPT[v, t] \\ OPT[w, t] + d(v, w) \end{cases}$ 
8:     if  $OPT[v, t]$  WAS UPDATED then
9:       LABEL  $v$  AS "ACTIVE";
10:    end if
11:   end for
```

12: **end while**

接着我们来看一个相关的问题，最长路径问题。这个问题和最短路径问题对应，需要找从源点到目标节点的最长路径。这是一个 NP 难问题，非常非常难。动态规划的 *Bellman–Ford* 方法在这里不行，因为这个问题不好分。假设把从 q 到 t 的问题变成两个问题 q 到 r 和 r 到 t :

1. $P(q, r) = q \rightarrow s \rightarrow t \rightarrow r$
2. $P(r, t) = r \rightarrow q \rightarrow s \rightarrow t,$

我们可能得到的解是: $q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$, 这不是简单路径。

在最长路径问题中，不能分解因为两个子问题可能有联系。在最短路径问题中就不会遇到这个问题。如果把最短路径问题分解成两个子问题，假设两个子问题 q 到 r 和 r 到 t 之间有共享一个点 w ，那么就会形成一个圈，把这个圈去掉以后会得到一个更短的路径，因为我们假设了没有负圈。

4.5 下节课内容

我们在讲动态规划的时候，分解子问题，由于不知道要怎么分，所以需要枚举。如果我们加入一点更严的限制的话，我们就不用枚举了。分治的时候，我们就这么分，不用枚举。动态规划的时候，我们不知道怎么分，所以要枚举。如果我们的问题更特殊一点，就不用枚举了，直接就知道选谁。这是什么算法呢？这就是贪心算法。它是动态规划算法的一个加强的版本。我们在 *Bellman–Ford* 算法中用到枚举，如果我们加一个条件，我们就知道下家走谁，根本就不用枚举，这就是 *Dijkstra* 算法。

第五章 贪心算法

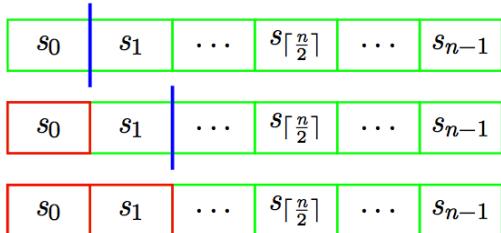
今天我们要讲新的内容：贪心算法。

我们从组合优化的问题开始讲，这个时候我们已经把一个问题给形式化之后，形成了一个组合优化的问题。那接下来我们怎么做呢？我们先看我们能不能正面地解决问题，也就是看这个问题能不能进行规约，及我们能不能把问题变成一个小的问题。如果可以，那我们意识到大概我们能做 DIVIDE & CONQUER。或者，整体这一类问题都叫 SELF-REDUCTION。如果在 DIVIDE&CONQUER 之后，我们还能够再观察到一个结构，也就是上节课讲的 OPTIMAL STRUCTURE，就意味着我们可以试试动态规划。今天，我们进一步深入，如果这个问题不仅仅可以规约成小的问题，并且它有最优子结构性质，同时它还有第三个性质，贪心选择（GREEDY SELECTION）的性质，那我们就能采取第三类方法：贪心算法。当然，这个贪心算法是一个充满灵感的技巧，以后我们不一定绕这么多，当我们经验丰之后，我们可能一开始就能想到这个问题可以用贪心算法来解决。好，今天我们来看看贪心算法。

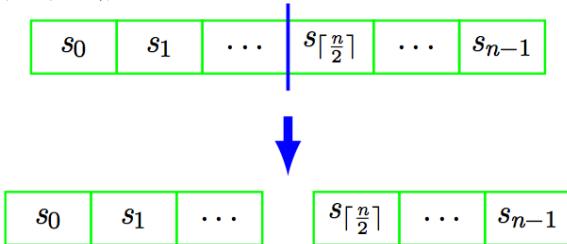
这堂课我们这么来讲：首先我们从上堂课讲过的最短路径和区间调度（INTERVAL SCHEDULING）问题开始，这两个问题足够让我们了解贪心算法；接着我们总结一下贪心算法的基本技术；我们还会讲一些其他例子 -HUFFMAN 编码（HUFFMAN CODE），我们都学过，也比较简单，我就跳过去了 -只讲讲这个 SPANNING TREE。为什么要讲它呢？因为我们要讲一点贪心算法的理论基础：拟阵（MATROID），讲拟阵的时候，顺便讲一下最小支撑树；最后我们顺带补充一下为了做最短路径算法而发明的一些数据结构，像：二项堆（BINOMIAL HEAP），FIBONACCI 堆，还有这个我们下次再讲的数据结构：UNION-FIND。

首先我们做一个声明：这几堂课都是沿着开始上课时我们所讲的解决问题的思路

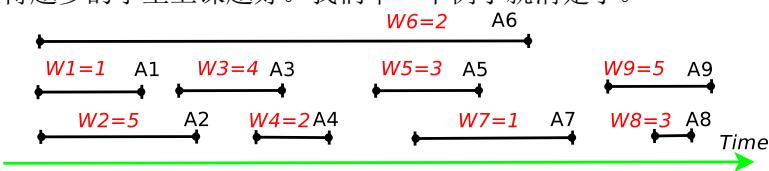
来走。所以首先我们还是要看看一个问题能不能分，这是非常重要的一个性质，也是我们拿到一个问题之后要做的第一个观察。我们一再说如果一个问题能分呢，我们脑子里可以想想看，会有两种分法，拿这个数组为例子：这个问题能不能被规约成更小的问题呢？我们从最简单的做起， n 个东西不会做，那我们看看一个会不会做，一个会做了，再看看两个会不会做，这是一种规约的策略。



给我 n 个东西不会做，那我把它一刀砍成两半，看看两个小问题会不会做，非常直白的思路。



好，为了讲贪心算法，我们来看看一个非常实际的问题的两个版本。就拿我们教室来说：可能有很多门课都要来预定这个教室。假如说第 i 门课程，我们叫它 A_i ，它要从 S_i 这个时候开始上课，到 F_i 结束，那教务处的老师就要利用这个时间信息来安排课程。我们已经知道有很多的课程都想来申请我们这间教室，教务处的需求是：能使得更多的学生上课越好。我们举一个例子就清楚了。



这是时间轴，每一条黑线代表一门课。我们用 W 来表示课程有多少学生。我们看看教务处如何安排。我们看看教务可能的安排：

1. 选择 $\{A_1, A_3, A_5, A_8\}$ ，这些可的确在时间上不冲突，那能容纳多少学生呢？把这几门课的学生加起来，一共 11 个学生。

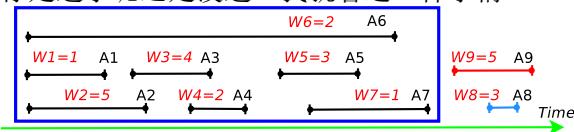
2. 选择 $\{A_6, A_9\}$, 每门课之间也不冲突, 总共 7 个学生可以上课。

我想到此, 这个问题已经被描述得很清楚了, 就是给我们每门课上课的时间和下课的时间, 以及每门课上课的学生, 你怎么调度, 使得更多的学生上课越好。非常清晰哈。

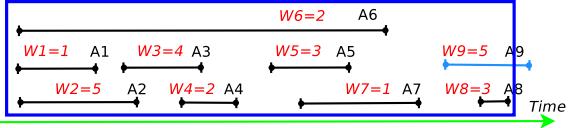
我对这个问题做了一下形式化, 但这个形式化大概可以不看, 我们只说两个重点吧, 其它的我们可以不看。

- COMPATIBLE: 两门课 A_i 和 A_j COMPATIBLE, 如果它们在时间上没有冲突。
- 我们先假设, 所有的课程已经按照它们的结束时间排序, 这个要求有点不起眼, 但是它很重要, 原因我们慢慢再说。

大家想想, 教务处应该怎么处理这九门课? 我们做此思维: “还是那个老问题, 给我们这么多门课, 的确看来, 我们搞不定这个事情, 那我们看我们能不能把这个问题给变小一点呢? 反正九门课我搞不定。能不能把它变小一点?” 所以, 很多时候, 我们碰到的问题的规模很大, 一下子可能我们无从下手, 我们就想能不能把它变成小的问题, 千方百计要琢磨一下这件事情, 直到我们发现它的确不好规约, 我们再想其他办法, 那是啥呢? 那等我们下下堂课会讲。现在我们先看, 这个问题假如可以被转化成小一点的子问题, 那么这个问题的解是啥呢? 就是从中间选出来的一个子集。我们把这个求解过程想象成一系列的决策, 在每个步骤, 我们都要选择一门课。假如我们已经拿到了最优解, 我们就问最优解中最后一个决策到底是什么? 也就是后一门课 A_n 你是选了呢还是没选? 我就看这一件事情!



假如说选了这么课 A_9 , 那么 A_8 不能选了, 因为跟 A_9 时间有冲突, 那其它的这些课呢? 都跟它没有冲突, 那是啥意思呢? 就是在你上课之前, 我就已经下课了。好, 那我剩下的, 就是在这蓝框里面所有的课, 我们再去选去。对不对? 你看, 一开始给我们九门课, 我们搞不定。选了一门课之后, 我们就把问题规约成了一个七门课的问题, 我们再去做, 因此把问题变小了。那还有一种可能呢? 假如说 A_9 这门课我没选, 怎么办呢?



那意味着，前 8 门课，甭管跟我打不打架，我都可以来考虑。那剩下的问题就是在前 8 门课中，我再去选课。那你看：无论是我们选，还是不选 A_9 ，我们的确是吧大问题变小了。那变小了这个子问题是什么样子呢？你把这两种情况一总结之后，就可以很清晰地看出这个样子：从 A_1, A_2, \dots, A_i 当中选择一些课，使能容纳的学生越多越好。

现在我们就可以插一句话呢，我们问什么要求这些课要按结束时间排好序呢？假如一开始我们不将它排序，则我们看是否要选一门课，就会有点麻烦。问题变成：

$$B(S) = \max \begin{cases} B(S - \{a\}) \\ B(S - \{a\} - \{\text{与 } a \text{ 冲突}\}) + W_i \end{cases}$$

那这两种情况就会很麻烦，你看子问题数目是多少？子问题数目就变成 2^n 了。虽然这个推理是对的，但是就是使得子问题数目比较多。有很多时候，我们要把问题稍微变一下型。

好，我们现在再讲回来。我们观察两种情况之后，把子问题定义成：在前 i 门课当中，我们怎么进行选择，使得容纳的学生越多越好，我们把这个最优解记作 $opt(i)$ ，那我们就有这个式子了：

$$OPT(i) = \max \begin{cases} OPT(pre(i)) + W_i \\ OPT(i - 1) \end{cases}$$

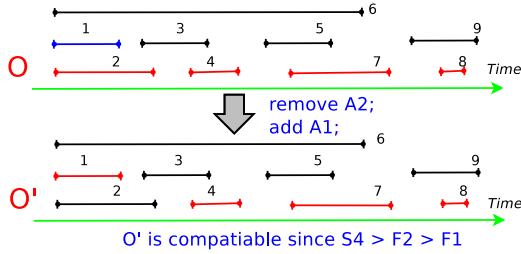
其中 $pre(i)$ 表示第 i 门课上课之前已经结束的所有课。大家不妨拿这个公式与前一个公式比较一下，看看差别。都是对的，但是前一个子问题数目太多了，后一个子问题数目是线性的。有了递归表达式之后，我们就能够很容易地写出一个动态规划的算法出来。好，下面我们来看看这个算法的复杂度。

一开始我们对所有的课进行排序，复杂度为 $O(n \log(n))$ 。这个动态规划的时间呢是 $O(n)$ 的，为什么呢？我们看，总共有 n 个子问题，每个子问题只要在两个数中找出最大值，所花的时间是 $2n$ ，加上前面的 $O(n \log(n))$ ，总体的时间还是 $O(n \log(n))$ 的。

现在对这个问题，我们已经做了一个非常好的 $O(n \log(n))$ 的动态规划的算法。

下面我们看看这个问题的一点特殊的清晰：假如每门课只有一个学生，我们的目标还是和以前一样，选择一些不冲突的课，容纳的学生越多越好，该怎么办？这相当于把以前的问题变严格了，所以原先的性质还成立，最优子结构的性质还有，不过又多了一个性质，贪心选择的性质。

那什么是贪心选择的性质？对这个问题来说，可以这么陈述：假如 A_1 是下课最早的这门课，那么 A_1 肯定会出现在最优解里。我们来用交换论证（EXCHANGE ARGUMENT）的方法来证明：

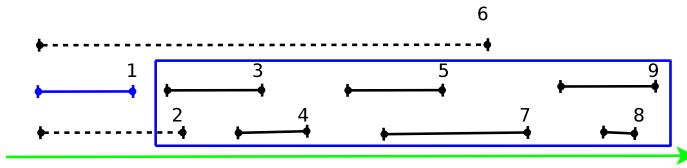


假如有另外的一个最优解 O ，没有选择 A_1 ，而是选择了 A_2 。那我们可以自信的说把 A_2 这么课换成 A_1 的话（形成一个新的解 O' ），肯定不会问题，因为 A_1 比 A_2 下课早，所以 A_1 也不会跟 A_2 后面的课冲突。所以 O' 肯定不会比 O 差。有了以上的这些性质，我们就可以将动态规划的算法变成贪心算法。假如所有的课还是按下课时间进行了排序。我们首先选择下课时间最早的这么课 A_1 ，接着，我们把所有与 A_1 冲突的课都给去了，然后再不断地往下做。这样我们就把原先的动态规划给简化了。

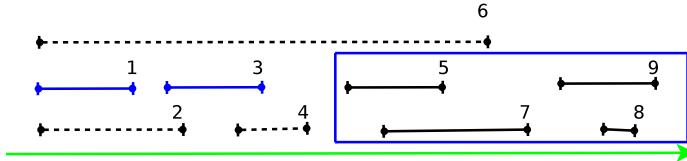
$$OPT(i) = \max \begin{cases} OPT(\text{pre}(i)) + W_i \\ OPT(i - 1) \end{cases}$$

在动态规划里面，我们还要枚举两种 CASE，用来判断是否要选 A_i ，现在我们不用枚举了，只要 A_i 是当下下课最早的课，那我们就直接选。现在这个复杂度还是 $O(n \log(n))$ ，因为最开始我们还是要将课程进行一下排序。现在贪心算法就变得很简单了：

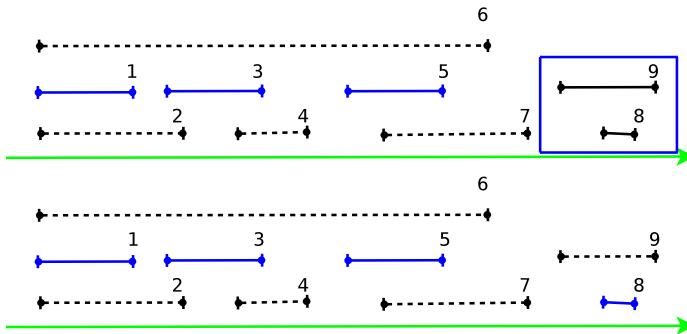
- 首先，我们选择下课最早的 A_1 ，剩下的就只要在蓝色的方框中找最优解就行，就归结成一个更小的子问题了。



- 接下来，就选上一个步骤中蓝色方框里下课最早的课 A_3 ，再将蓝色方框缩小。

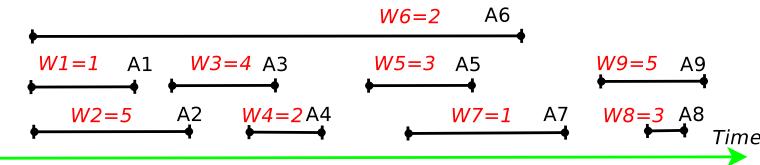


- 然后，跟之前一样，一步一步构造出一个最优解出来。



在动态规划中，最后要的出我们到底选了那些可，还要进行回溯。现在呢，也不用回溯，直接就能够看出了。

现在问一个小的问题，这种贪心的策略在一般的情况下还能够成立吗？不会了。我们看，还是这个一般的情况，及每门课的学生都不一样



按照贪心的办法，我们会选择 $\{A_1, A_3, A_5, A_8\}$ ，总共是 11 个学生。而最优解是 $\{A_2, A_4, A_5, A_9\}$ ，总共 15 个学生，只能够通过动态规划算法算出来，贪心算法无能为力。为什么在一般的情况下，贪心算法就不灵光了呢？是因为贪心选择的性质不成立了。所以，大家要注意一下，这两个问题，陈述起来很像，只是这个 WEIGHT 变了一下，一下就导致算法改变了。

讲到这就能总结一下了，我们看看动态规划和贪心到底有什么异同。首先看相同点：

- 都是来求解优化的问题
- 都有最优子结构这个性质。不要以为最优子结构是动态规划所独有的，其实不是，贪心算法也要利用到这个性质。
- 最后一点是最深刻的话，“在每一个贪心算法的背后，几乎总会有一个动态规划的算法，算法这个动态规划算法比较笨拙”。这也是为什么诸位本科学算法，通常先是先将贪心算法。但我还是倾向于将贪心算法放在动态规划算法之后将，因为前者的确是对后者的一个加强。很多同学都有 MIT 的那本算法导论，我建议大家看一下贪心算法那章的最后一节（SECTION），写动态规划和贪心算法之间的关系，写得非常好，是其他书里面所没有的。

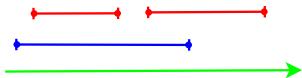
我们接着看不同点：

- 动态规划算法一般需要在所有的决策步（DECISION STEP）枚举所有可能，而且要在所有的子问题被解决之后，才还不能做决定。而贪心根本就没有枚举这一步，而是直接基于局部最优解来做决定，不用考虑这个子问题将来会怎样。（因为无论子问题的结果如何，基于局部最优解所做出的决定总是能够导致全局最优解，这也是贪心选择性质的另外一种表述。）

那到底如何知道该选择贪心算法呢？第一种方法是沿着经典路线：先做 DIVIDE&CONQUER，再做动态规划，再做贪心。另外一种：当我们做了很多问题之后，有经验了，我们就试错吧，非常讲究灵感。我们就把求解过程想象成一系列的决策，在每一部，部分解逐步往上涨的时候，尝试不同的贪心选择性质。再次强调，这个需要灵感，在这个过程中，我们可能会犯很多错误。比如说，假如教务处给然我们安排可，我们已经知道，动态规划是可以解决这个问题的，我们再看看贪心算法行不行，而不是沿着经典路线逐步地改善。如何看呢，我们尝试不同的贪心规则：

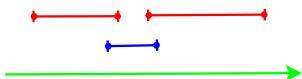
- 假如一门课上课越早，就优先安排。

初看这个想法还不错，但一分析，就发现有问题，比如说这样三门课：



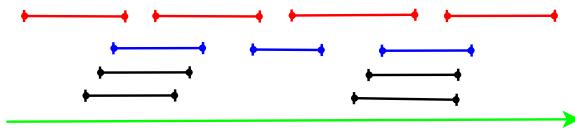
我们看到蓝色的课上课最早，我们就安排它，但实际上是对的，应该要安排那两门红色的课才对，这样能安排更多。

- 这个就有点 TRICKY 了，这个是说如果一门课上课时间短，我们就安排它。



我们发现这也不行，比如安排了一门蓝色的课后（因为它历时短），两门红色的课就不能选了。

- 我们想个更复杂的：如果一门课与其它课冲突很少，就越好。初看这个想法，的确合情合理，大家安排课的时候，的确有可能想到这个方法，可惜还是不对。



大家看，中间蓝色那门课只与两门课冲突，根据这个贪心规则，我们首先选它，导致它上面两门红色的课不能选了。于是我们只好选择它旁边两门蓝色的课，这样我们一共选择三门课，但最优解确是选择最上面红色的那四门课。

大家看，我们指望我们有灵感，还弄出很复杂的贪心的规则，可最后还是找出了它们的反例。所以，很多时候，我们的灵感真的靠不住，需要我们严谨的验证。这是大家将来写程序会碰到的问题，将来大家碰到一个问题，可能第一想用的办法就是贪心算法，因为它特别的简单，也特别地能体现我们的聪明才智，可是有时候它既是经不起推敲。当然，有时候我们想要找一个反例还挺难的，比如说上面最后一个例子，我是费老大的劲才找出一个，前面两个到还很 EASY。

接下来，我们要跟上堂课挂起钩来，我们最短路径问题，很多时候我们从 DIJKSTRA 算法开刀，那我们上次课为什么要绕一个弯先讲 BELLMAN-FORD 这个动态规划算法呢？就是为了和今天做一些勾连。问题我再简单陈述一下，就是有很多城市，城市之间有不同距离的道路，给了两个城市 s 和 t ，问从 s 到 t 的最短距离是多少？如果没有负圈呢，我们就可以用 BELLMAN-FORD 算法，当然，若有负圈，它

也能检测出来，所以说它很强大。那如果把条件再加强，不仅没有负圈，而且还没有负边呢？就用不着这个动态规划了，有点费劲，直接做了一个更快的算法，就是 DIJKSTRA 贪心算法。

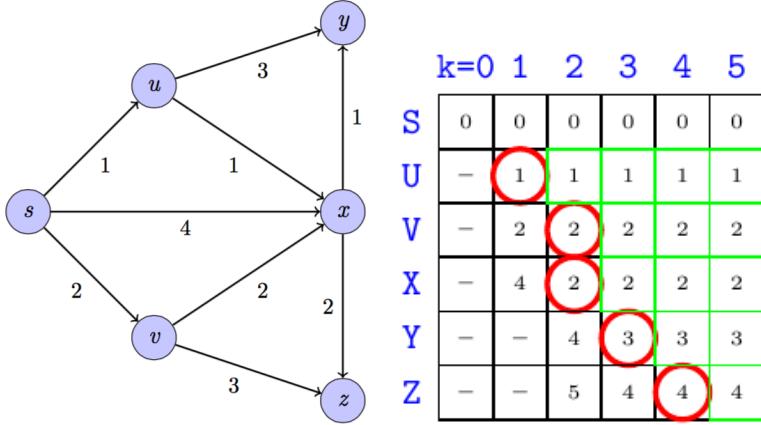
下面我们先回顾一下上节课我们怎么做的这个动态规划算法，然后再看，当改了一下条件，所有边都是正的之后，怎么样进行简化。上堂课我们是这么讲的：我们的解是从 s 到 t 的一条路径，我们还知道 s 到 t 当中，最多只有 $n - 1$ 条边，其实这个时候已经把原先的问题变了。原来的问题是这样陈述的：在图 G 当中，我们要找从 s 到 t 的最短路径，记作 $d(G, s, t)$ 。一旦没有负圈的话，我们将问题转换成：求从 s 到 t 当中，最多有 k 条边的路径，记作 $d(s, t, k)$ 。在此，还是再次强调一下，为什么我们要转换问题的描述呢？因为前者的子问题数目为： 2^n ，而后者是 $O(n)$ 。

好，回顾来，我们把求解过程想象成一系列的决策，每一次决策步决定下一步往哪走。假如我们已经得到一个最优解 O ，我们就观察当中的最后一个决策，是从哪个城市到达 t ，容易得出：凡是能直接到达 t 的点都是有可能的。假如是从 v 到达 t ，剩下的就变成这么一个子问题：如何从 s 到达 v ，最多经过 $n - 2$ 条边。所以，我们就可以将子问题的形式写出来了，就是：求从 s 到 v 的距离，最多有 k 步的路径，记作 $OPT(v, k)$ 。那最优子结构的性质就出来了：

$$OPT(v, k) = \min \begin{cases} OPT(v, k - 1) \\ \min_{\langle u, v \rangle \in E} \{OPT(u, k - 1) + d_{u,v}\} \end{cases}$$

上面式子的第一项想要体现的是“至多”，下面一项是枚举所有能到达 v 的 u ，看从哪点过来使得总体的距离最短。这个算法的时间的复杂度是： $O(mn)$ ，其中 m, n 分别是图中边和节点的数目。

现在我们跑一下 BELLMAN-FORD 这个算法，同时，我们假设每条边都是正的。

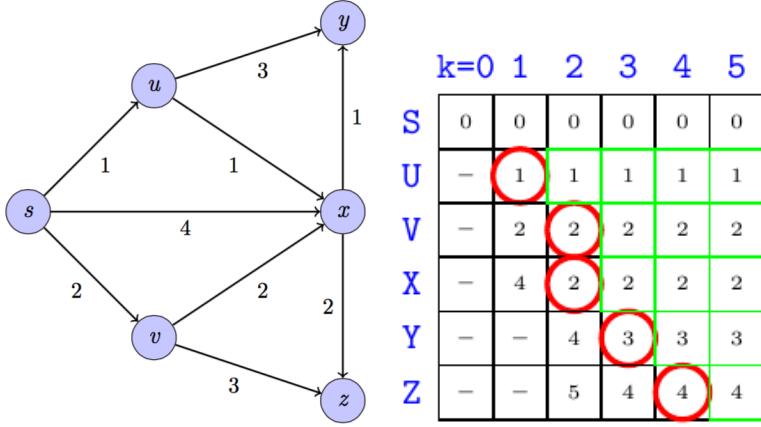


我们可以得到这样一张表，表的行和列分别是 $OPT(v, k)$ 中的第一，第二个参数。这个表的用处，我们待会再看。

我们再看第二个版本，假如说这个边上都是正的，会出什么问题。在算法的第 k 步，我们考察一个特殊的节点 v^* ， v^* 是从 s 出发最多经过第 $k - 1$ 步能够到达的最近的一个节点。对于 v^* ，最优子结构的性质也成立，所以有：

$$OPT(v^*, k) = \min \begin{cases} OPT(v^*, k - 1) \\ \min_{u \in E} \{OPT(u, k - 1) + d_{u, v^*}\} \end{cases}$$

我们仔细看看有什么特殊的地方。光成数学形式上看，我们就可以看出点蹊跷来。我们会发现，上面式子下面的那一部分根本就不用看了，因为下面式子肯定要比上面的大。因为 v^* 是是从 s 出发最多经过第 $k - 1$ 步能够到达的最近的一个节点，所以 $OPT(v^*, k - 1)$ 肯定要小于 $\min_{u \in E} \{OPT(u, k - 1)\}$ ，我们假设所有边都是正的，所以 $d_{u, v^*} > 0$ ，所以更加有 $OPT(v^*, k - 1) > \min_{u \in E} \{OPT(u, k - 1) + d_{u, v^*}\}$ ，所以 $OPT(v^*, k) = OPT(v^*, k - 1)$ 。这个性质在图中怎么体现呢？大家看图：



从列开始看：一旦在一列中间找到最小的值之后（红圈中的值），在看最小值所在行右面的值（绿色方框中的值），永远不变了。最优子结构的性质： $OPT(v, k) = \min \begin{cases} OPT(v, k - 1) \\ \min_{u \in N(v)} \{OPT(u, k - 1) + d_{u,v}\} \end{cases}$

是对任何的节点都成立的。但若我们在所有边都不为负的情况下，对于那些特殊的节点 v^* ，最优子结构的性质还用不着，就是 $OPT(v^*, k) = OPT(v^*, k - 1)$ 就够了。若是按照 BELLMAN-FORD 算法，表中的每一个数值都是要计算的，其实，在所有边都不为负的情况下，所有绿色方框中的值的计算都没有必要，是不用算的。只要计算红色圆圈以及其左边的值就够了，那我们如何来计算这些值呢？我们来看一下一个贪心选择的性质。

我们用 S 来表示最短距离已经被确定的点。我们从所有不在 S 中的点中找一个离起始点最近的结点 u^* ，所以，从起始点到 u^* 的距离可以表示为： $d'(u) = \min_{w \in S} \{d(w) + d(w, u)\}$ ，并且：最短路径就是： $P = s \rightarrow \dots \rightarrow w \rightarrow u^*$ 。

下面是算法：DIJKSTRA(G, s)

```

1:  $S = \{s\}$ ; //S DENOTES THE SET OF EXPLORED NODES,
2:  $d(s) = 0$ ; // $d(u)$  STORES AN UPPER BOUND OF THE SHORTEST-PATH WEIGHT
   FROM  $s$  TO  $u$ ;
3: for all NODE  $v \neq s$  do
4:    $d(v) = +\infty$ ;
5: end for

```

```

6: while  $S \neq V$  do
7:   for all NODE  $v \notin S$  do
8:      $d(v) = \min_{u \in S} \{d(u) + d(u, v)\};$ 
9:   end for
10:  SELECT THE NODE  $v^*$  ( $v^* \notin S$ ) THAT MINIMIZES  $d(v);$ 
11:   $S = S \cup \{v^*\};$ 
12: end while

```

在课程网站上有这个算法的动画演示：LEC7-DEMO-DIJKSTRA.PDF。我就即使不知道 DIJKSTRA 当年是怎么想到这个算法的，但是假设我们学了动态规划，学了 BELLMAN-FORD 之后，我们也可以写出类似 DIJKSTRA 的算法出来，只需要不计算绿色方框中的值就行。



这位就是 DIJKSTRA。

我们再来看看 DIJKSTRA 算法和 BELLMAN-FORD 算法，我们只是把对边的值的限制稍微改了一下，假如没有负圈，我们有这样一个表达式： $OPT[v, k] = \min \begin{cases} OPT[v, k - 1], \\ \min_{u \in E} \{OPT[u, k - 1] + d(u, v)\} \end{cases}$

如果再加强一下，根本就没有负边，这个约束更强了，导致我们可以用贪心算法。

DIJKSTRA 算法的复杂度是 $m + n \log n$ ，为什么呢？我们将 DIJKSTRA 用一个更接近计算机的伪代码描述出来：DIJKSTRA(G, s)

```

1:  $key(s) = 0; // key(u)$  STORES AN UPPER BOUND OF THE SHORTEST-PATH WEIGHT
   FROM  $s$  TO  $u;$ 
2:  $PQ.$  INSERT ( $s$ );
3:  $S = \{s\}; //$  LET  $S$  BE THE SET OF EXPLORER NODES;
4: for all NODE  $v \neq s$  do

```

```

5:    $key(v) = +\infty$ 
6:    $PQ.\text{INSERT}(v)$  N TIMES
7: end for
8: while  $S \neq V$  do
9:    $v = PQ.\text{EXTRACTMIN}()$ ; N TIMES
10:   $S = S \cup \{v\}$ ;
11:  for EACH  $w \notin S$  AND  $\langle v, w \rangle \in E$  do
12:    if  $key(v) + d(v, w) < key(w)$  then
13:       $PQ.\text{DECREASEKEY}(w, key(v) + d(v, w))$ ; M TIMES
14:    end if
15:  end for
16: end while

```

我们定义了一个优先队列 (PRIORITY QUEUE)，是什么意思呢？就是里面存了一堆数，我们可以很快找出一个最小来。一开始，把所有的不在 S 中的节点，每个节点都有一个值 $+\infty$ ，我们把它插到优先队列里面去。然后，我们从优先队列里面取一个最小，把最小的数放到 S 当中。接着，对所有墨水没有洇到的那些节点，我们看一下会不会从墨水已经洇得结点当中，重新更新一下路径。所以，一旦能找到一个新的方案，我们更新一下距离。对优先队列，我们有 n 次插入的操作，有 n 次取出最小值的操作，还有 m 次，要把这个值改一下。

优先队列我们怎么做呢？有很多种方案。如果大家没有学过优先队列的话，大家也能找出一种方案，本质上即使一堆数之间找最小。

OPERATION	LINKED	BINARY	BINOMIAL	FIBONACCI
	LIST	HEAP	HEAP	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	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

我们可以写一个数组或链表，讲所有的数存下来。数组和链表，插入一个数只需要 $O(1)$ 的时间，取出最小值得从头到尾都比一遍，需要 $O(n)$ 的时间。现在这个讲的有点快，大家可以先听一下，之后我们会有一个 100 页的 SLIDE 来详细讲。总个 DIJKSTRA 算法，需要 $O(n*1+m*1+n*n) = O(n^2)$ 的时间，是所有方法中最老土，最大的。其实，做 DIJKSTRA 算法，单是数组不太好，经过一些列的改进，我们最后采用 FIBONACCI 数列，插入只需要 $O(1)$ 的时间，取出最小值要 $O(\log n)$ ，改一下值需要 $O(1)$ ，最终需要 $O(m + n \log n)$ 。就相当于把所有的边枚举一次，把所有的结点排一下序的速度。值得说的是，这个 FIBONACCI 堆就是专门为加快 DIJKSTRA 算法专门设计的一种数据结构。

好，上节课我们讲到，为了做 DIJKSTRA 算法，最老土的办法是数组或链表，但是稍微有点慢。再往下呢，我们可以用一个树结构来存储所有的数，也是本科学到的二叉堆。再往下，我们可以用很多颗树，就是一个森林。再往下，我们可以不仅仅用很多颗树，而且对树的样子不要有太严格的要求，就是 FIBONACCI 堆，做得更快。这里面有很多的设计思想在里面。

我们看一下优先队列的核心功能，就是怎么能够存一堆书，以方便我们之后如何能够最快地找最少，而且要考虑到这一堆数还在不断地变。所以，优先队列要支持能这几个功能：

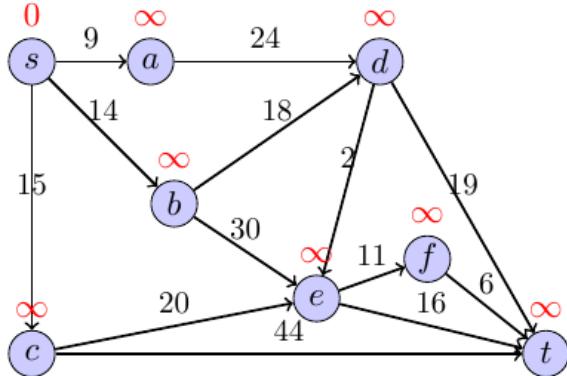
1. 添加一个数
2. 在已有的数中找最小

3. 对数进行修改

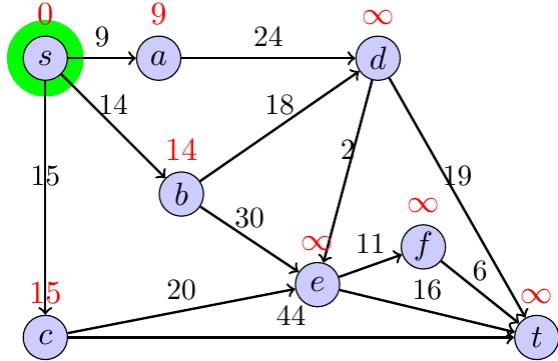
优先队列很有用，很多算法都有用到，比如：

- DIJKSTRA 算法
- PRIM 的最小支撑树
- HUFFMAN 编码
- A^* 搜索算法
- HEAPSORT

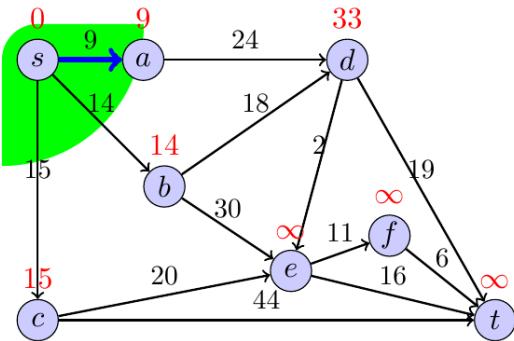
反正只要是在一堆数中间找最小，就要用到它。它就干这事。我们下面具体看看 DIJKSTRA 算法中是如何用到优先队列的。



一开始优先队列是这个： $PQ = \{s(0), a(\infty), b(\infty), c(\infty), d(\infty), e(\infty), f(\infty), t(\infty)\}$ ，除了 s ，其它的数都是正无穷，接着从里面找最小，执行一次，是 $\text{Y}=s$ ，接着把与具有当前最小值节点 s 连接的节点的值改一下，比如 a ，原来是正无穷，现在要变成 9。



现在优先队列变成了： $PQ = \{a(9), b(14), c(15), d(\infty), e(\infty), f(\infty), t(\infty)\}$ ，然后再取出最小值，然后和之前一样，把与具有当前最小值节点 a 连接的节点的值改一下，



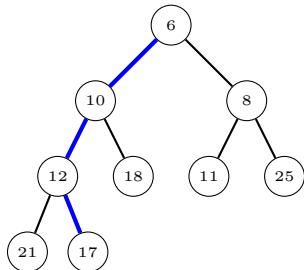
优先队列是： $PQ = \{b(14), c(15), d(33), e(\infty), f(\infty), t(\infty)\}$ 。之后再不停的执行这个步骤，每次都是找最小，然后再改一些数。直到优先队列中具有最小值的节点是目标节点 t 。

我们怎样来实现这个优先队列呢？先看看最老土的办法，数列。假如我们有一堆数：[8, 1, 6, 2, 4]，这是一个没有排序的数组。如果新加一个数，我们直接将它放在数组最后面就行了，要花 $O(1)$ 的时间。若找最小呢，那要从头到尾都要找一遍，所以要花 $O(n)$ 的时间。如果我们能够时刻保持数组有序，那求最小就会变得很简单，有序数组为：[1, 2, 4, 6, 8] 只要找第一个数就行，要花 $O(1)$ 的时间，不过插入就麻烦了，比如要插入一个 5，要从 1 开始比，比到 8 后，在决定放在 8 的前面，可能要与所有的数都比一遍，要花 $O(n)$ 的时间。

所以没排序的数列的时间复杂度是这样的：

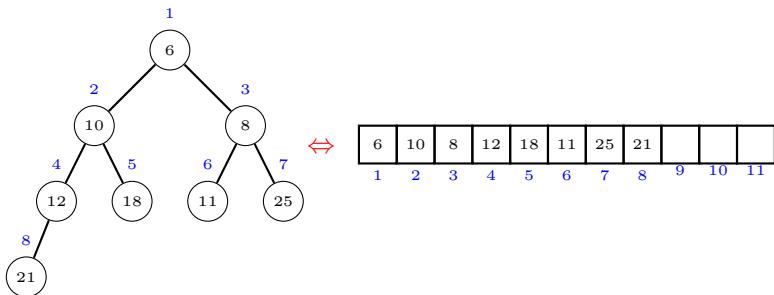
Operation	Linked List
INSERT	$O(1)$
EXTRACTMIN	$O(n)$
DECREASEKEY	$O(1)$
UNION	$O(1)$

这有点慢，那怎么办？提出一个新的数据结构：二叉堆，发明人是：R. W. FLOYD。它的设计思想：我要放松一下我的要求，但是别放松得太狠。因为有的过强的约束是没有必要的。这也是以下我们要讲的几个数据结构的共同思想。我们的目标是找最小，难道需要所有的数都严格地排序吗？没有必要。但也不能放松得太狠，我们还是要求稍微有点序。



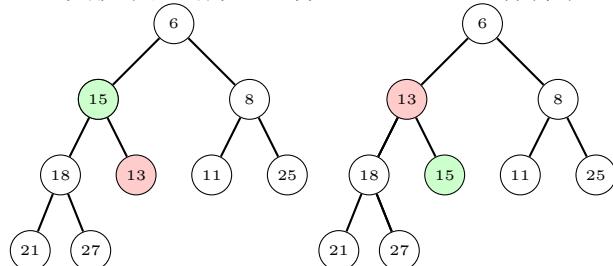
这个二叉树的特性是：任何一个父亲都比两儿子要小。我们并不要求所有的数都有一个完美的排序，只要求任何一条路径上有序就够了。为什么这样更好呢？因为这是二叉树，任何一条路径只有 $\log n$ 这么高。

二叉树有两种实现方式：



第一种方法是老老实实的存下两个子节点的指针，还可以干脆将其弄成一个数组就行，这样，第 k 个节点的父亲就是第 $k/2$ 个节点。假如我们采取第二种方式。这样，我们就只要这个数组中部分有序就够了，相当于放松了一下要求。我们还要对二项堆进行一些基本操作，当 HEAPORDER（父节点要比其子节点小）被违反了之后，则

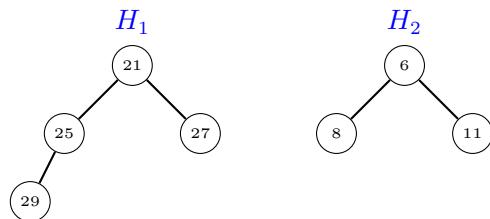
通过交换节点的方法，讲 HEAPORDER 保持下去。



比如上面这个图，左边的 HEAPORDER 被打乱了，于是我们交换值为 15 与值为 13 的节点的位置。若用二叉堆表示了一堆数之后，则求最小的问题就变得很简单了，因为父亲的值总比儿子的小，那么求最小就只要看根节点就好了。当我们先插入一个数时，先把它放到树的最下面，这可能会破坏 HEAPORDER 规则，但我们可以通过一系列的交换来重新维护规则。该值也是同样的道理。

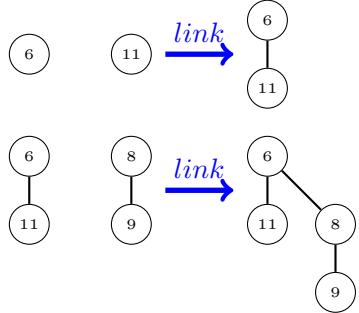
假如我们在 DIJKSTRA 算法中用到了二项堆，那算法的时间复杂度是多少呢？算法中最多只有 m 项插入，换值和取出最小值操作，每次操作最多花 $\log n$ 时间，所以总体的复杂度是 $O(m \log n)$ 。

我们看，二叉堆相比用单纯的数组或是链表，它放松了一些，不要求所有的数都有序。但是用二叉堆，两个堆的合并要花 $O(n)$ 的时间。



比如上面是两个二叉堆，我们想把它们合成一个。一种方法是把一个中的每一个数一个个地插入到另外一个二叉堆中，这是可以得，但是很慢。另外一种呢，把所有的数都拿过来，放在一起，然后重新建堆，这个要花 $O(n)$ 的时间。那我们想有没有更快的方法呢？我们再想想我们一定需要将两个堆合并成一个吗，我们可不可以多棵数来装下所有数呢？这就是下面我们要讲的二项树（BINOMIAL HEAP）的思想。这就是更进一步的放松了，我们之前要求要用一棵树来表示所有的树，现在我们放松到可以用多棵数来表示。这样，合并操作相当于就只要花 $O(n)$ 的时间。但是我们还要记得一点，我们不能过分放松了，因为之后我们还要在所有的数中间找最小呢。用多棵

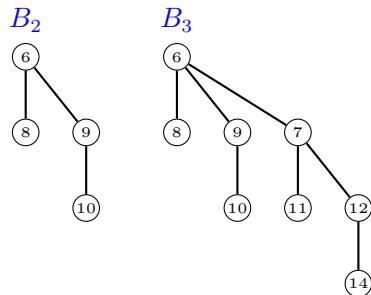
树表示的话，我们如何在所有数之间找最小呢？由于每棵树的最小值都是它的根，那我们只要在所有树的根之间找最小就行。这种方法最极端的例子是什么？那就是每个数字单独形成一个数，这样就相当于回到了一开始用数组的方法，我们需要在所有数当中取出最小值。这有很麻烦。所以二项树思想最光辉的思想就在这，可以允许用多棵树来表示，但不要有太多，太多了又麻烦了。那怎么做到这一点呢？于是我们发现树太多了，我们就将他们合并一下。具体方法叫做 CONSOLIDATING：

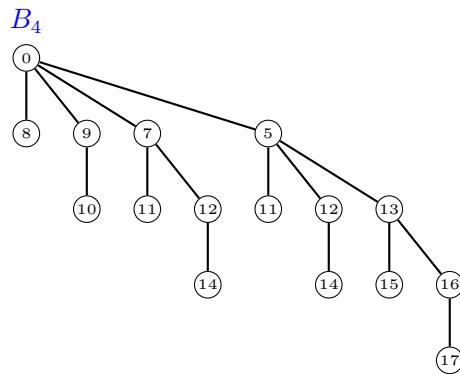


那具体怎么合并呢？我们只要比较一下两颗树根的大小，把大的那棵树接在小的那一颗树的根节点的下面，以保持 HEAPORDER。我们下面给出二项树一个严格的定义，这是一个递归定义的：

- 单个节点可以叫做一个二项树。
- 一个二项树一定由两个相同大小的二项树按照规则（比较一下两颗树根的大小，把大的那棵树接在小的那一颗树的根节点的下面）组成。

比如，下面几个图都是二项树：

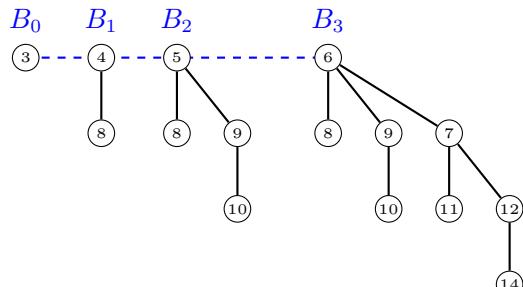




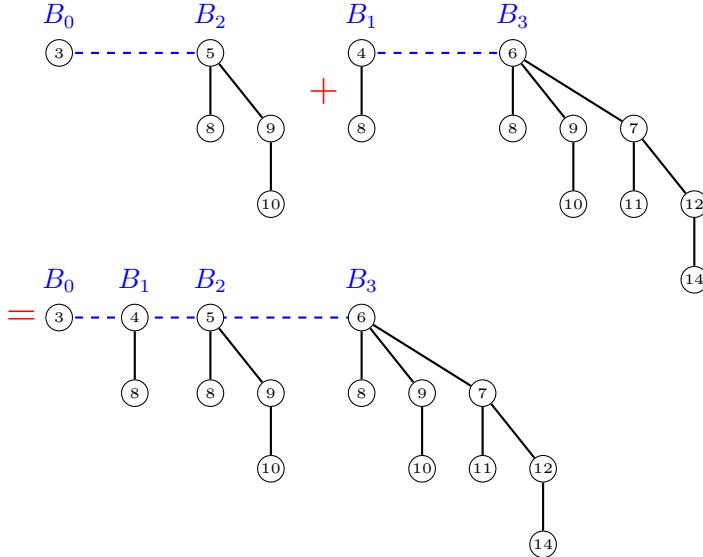
我们看看这个数有什么特点：任何一个树都不是特别平衡，但每个数的节点都有这样一些性质：

1. $|B_k| = 2^k$;
2. $\text{height}(B_k) = k$;
3. $\text{degree}(B_k) = k$;
4. 第 i 个儿子有 $i - 1$ 个儿子；

一个二项堆其实不是一棵树了，它是很多棵树：



每棵树是同样满足一定的排序的。而且每一个 k 级树最多只有一颗，因为若有多颗，我们需要将它们合并。所以要容纳 n 棵树，整个森林中最多只有 $\lfloor \log_2 n \rfloor + 1$ 棵树，并且最高的那棵树的高度是 $\lfloor \log_2 n \rfloor$ 。所以在所有数中间找最小，我们只需要在树的根节点中间找，只要花 $O(\log n)$ 的时间。接下来我们看看讲两组树合并需要花多少时间？



上面这种情况最简单，把两组树放在一起之后，也没有两两大小一样的树，我们只需要将他们简单地放在一起就好了，花 $O(n)$ 的时间。若是有两组树放在一块后有两两大小相同的树，那我们就需要将相同大小的树合并，只到一组数之间没有两两大小相同的树为止。比如两组树合并后有四组二项堆： $\{B_1, B_2, B_1, B_3\}$ ，我们先要将两个 B_1 变成一个 B_2 ：

$\{B_2, B_2, B_3\}$

再将两个 B_2 变成一个 B_3 ：

$\{B_3, B_3\}$

再将两个 B_3 变成一个 B_4 。

将两个二项堆变成一个很简单，只要花 $O(1)$ 的时间。而合并两个森林时，我们只花了 $O(\log n)$ 的时间。那插入一个数呢，若原先森林里没有单个数的树，我们放进去就行，若有了，我们就合并，所以也是 $O(\log n)$ 的时间。取出最小值呢，当我们在一棵树中取出它的最小值（根节点）后，它就变成了两棵树，这可能会和其它的树大小相同，所以可能还需要再合并，也需要 $O(\log n)$ 的时间。

总结一下：二项堆（BINOMIAL HEAP）相比二叉堆（BINARY HEAP），合并的时间改善了。

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	$O(1)$	$O(n)$	$O(\log n)$

第六章 二项堆、斐波那契堆、贪心

6.1 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。

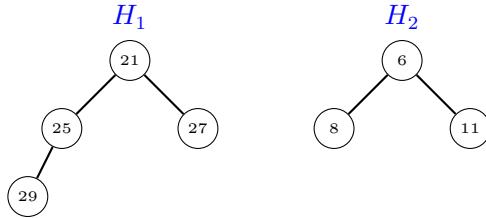
首先来看下做这个数据结构的时候，最基本的出发点是什么。总体来说我们是要做一个优先队列，因为我们在 DIJSTRA 算法中，总是要在集合中找最小数。优先队列可以理解为，在队列中一个数越小就应该放在前面，所以实现这种数据结构，我们最直观简单的是开辟一个数组，但是存在一些问题，比如从 N 个数中找最小可能需要

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。

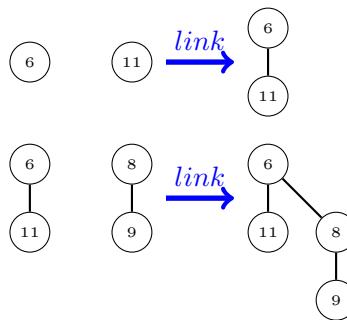
6.2 二项堆

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。



6.2.1 如何控制数的数目：合并树

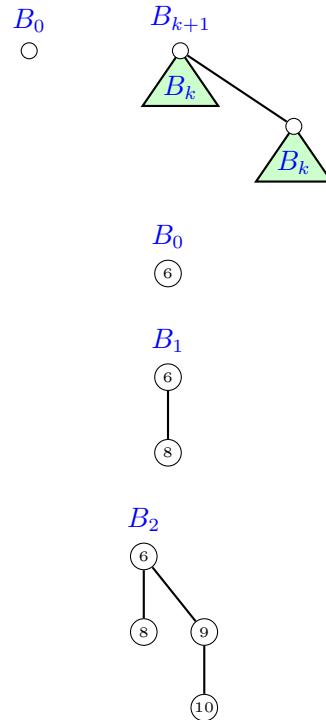
为了控制数的数目不能太多，加了一个合并的操作。如图所示，每次合并时找同等级的树，合并的规则和堆合并的规则一样，合并完父节点的数比子节点的数都要小，下图分别为一级的数和二级的树的合并操作。



6.2.2 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 κ 。(3) 从左往右儿子分别分 $B_0, B_1, B_2 \dots B_k$ ，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

$$B_0 : 1$$

$$B_1 : 11$$

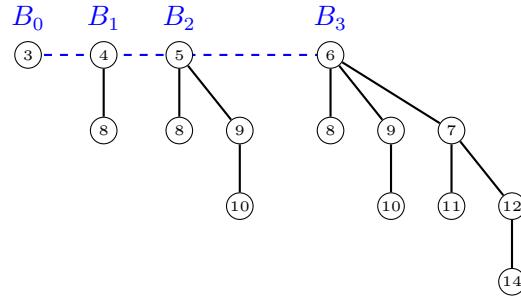
$$B_2 : 121$$

$$B_3 : 1331$$

$$B_4 : 14641$$

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 。

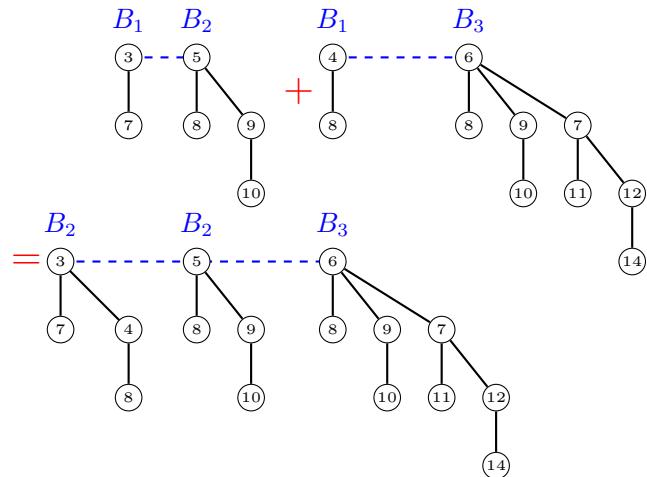
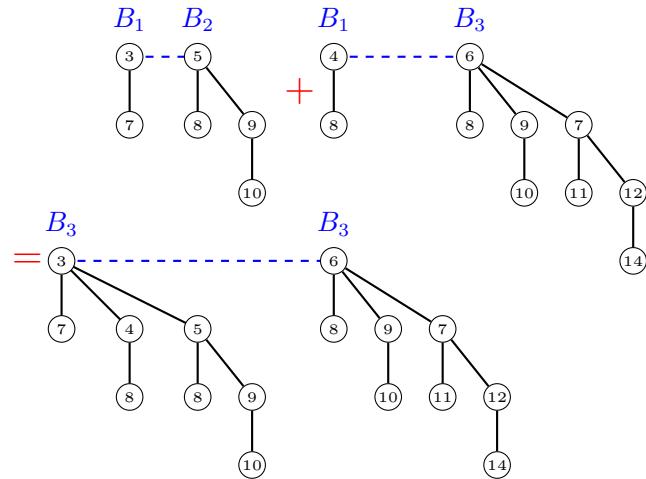
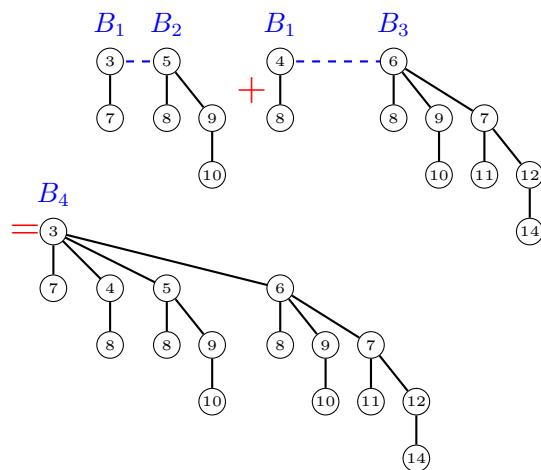


图 6.1: Consolidating two B_1 trees into a B_2 tree

图 6.2: 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。

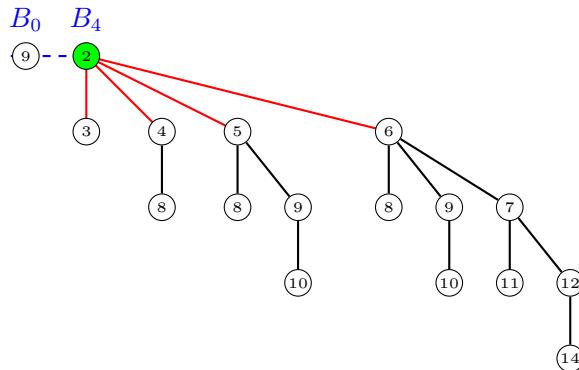
那么在森林中找最小，先看每棵树的根，如图，节点值为 2 的最小，那么就把这个最小的取走，儿子都独立成一棵树，然后继续合并。

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

```



6.2.3 均摊分析

上一节提到，二项树在插入操作只需要 $O(\log n)$ 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 $\log n$ 棵数，合并最多需要 $O(\log n)$ 的时间。

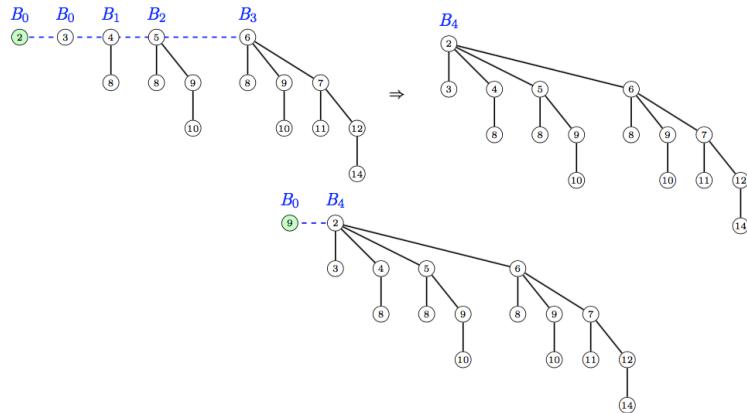
但是对于时间复杂度的分析不是特别准确，实际上，插入操作只用了 $O(1)$ 的时间（常数级的操作，几次就可以做到），这就是设计这个二项堆的目的，合并也是 $O(1)$ 的时间。这种复杂度可以通过均摊分析来得到。

Operation	Linked List	Binary Heap	Binomial Heap
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	$O(1)$	$O(n)$	$O(\log n)$

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

$\text{INSERT}(x)$

- 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 表示 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，WHILE 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

$\text{EXTRACTMIN}()$

- 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 。真实时间为 $d+w=d+\Phi$ 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 $N*(1+\text{DECREASE } \#tree)+M*(d+\text{DECREASE } \#tree)$ ，总时间为 $N+M*D+ \text{所有的 DECREASE } \#tree, d$ 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 $N+M\log N + \text{所有减少的树}$ 。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 $N+M\log N$ 。所以总时间 $\leq 2(N+M\log N)$ ，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 $2\log N$ 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 $O(n_1 T_1 + n_2 T_2, \dots)$ ，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 $\log N$ 的插入操作之后，下面有 $\log N$ 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。

6.3 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 DECREASEKEY 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

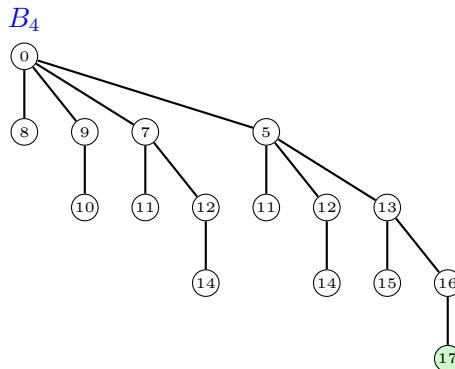


图 6.3: DECREASEKEY: 17 to 1

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 $2^{1.618}$ 。

6.3.1 斐波那契堆的 Decreasekey 操作

下面看下斐波那契堆是如何操作 DECREASEKEY 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。

现在把 15 变成 2，2 比 13 小砍掉，13 做上标记。

再把 12 变为 8，满足规则不做操作。

再把 14 变为 1，比 8 小砍掉，8 做上标记。

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：

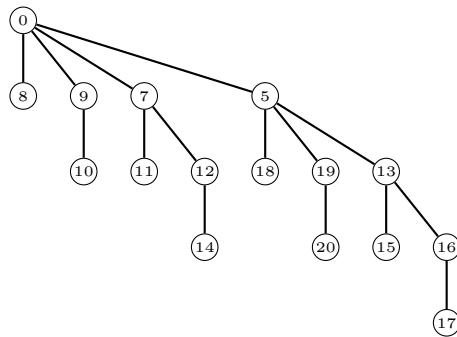


图 6.4: The original Fibonacci heap

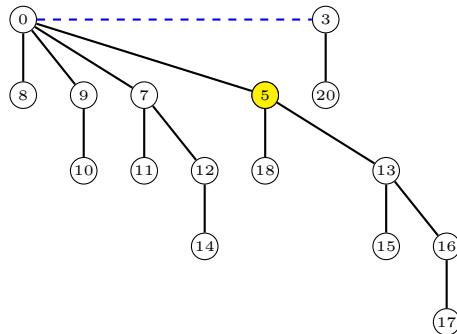


图 6.5: DECREASEKEY: 19 to 3

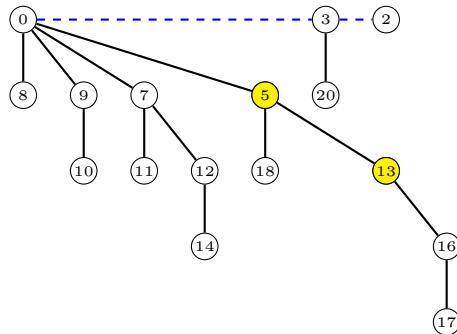


图 6.6: DECREASEKEY: 15 to 2

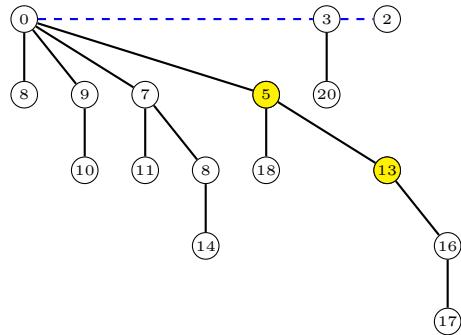


图 6.7: DECREASEKEY: 12 to 8

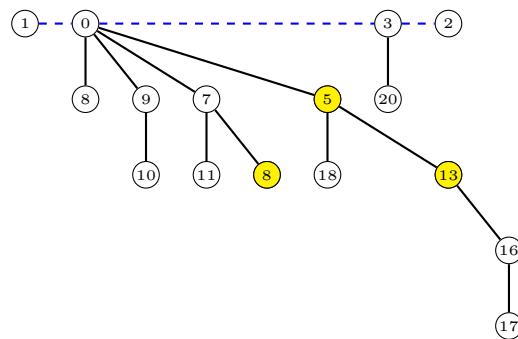


图 6.8: DECREASEKEY: 14 to 1

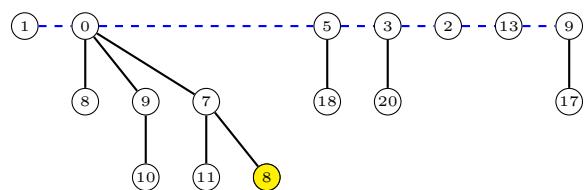


图 6.9: DECREASEKEY: 16 to 9

6.3.2 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：

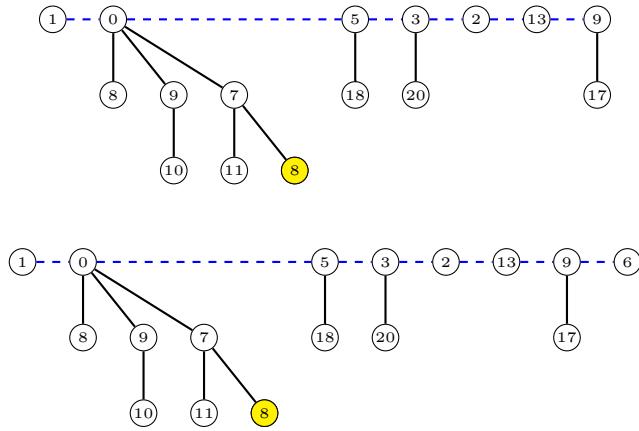


图 6.10: INSERT(6): creating a new tree, and insert it into the root list

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。

6.3.3 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。

6.3.4 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 DECREASEKEY 单次操作实际到底花费多长时间，观察 3.1 中 DECREASEKEY 算法，前 4 行花费为 $O(1)$ 时间，WHILE 循环中也是单位时间，真实时间为 $1+w, w$ 为 WHILE 循环的数目。

我们考察下面这个量 $\Phi = \#trees + 2\#\text{marks}$ ，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: $1+w=1+\Phi$ 减量，但是如果我们可以把 w 均摊到其他操作上，那么 DECREASEKEY 操作只需要花费 $O(1)$ 的时间。

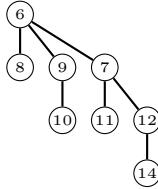
再看下 EXTRACTMIN 花费多少时间，观察 3.3 中的算法，第一行花费 $O(\log n)$ 时间，第二行花费 $O(\log n)$ 时间， WHILE 循环花费 w 时间。真实运行时间为 $d+w$ ，其中 d 表示有多少儿子升级，放到根里面。我们仍考察 $\Phi = \#trees + 2\#\text{marks}$ ，每次增加 d 棵树，每次 WHILE 循环就减少一棵，共 w 棵。运行时间: $d+w=d+\text{DECREASE IN } \Phi$ 。

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。

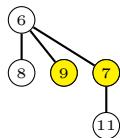
假如考虑一个操作序列包括 n 次插入， m 次找最小， r 次修改操作。整体真实运行时间为: $n + md_{max} + r +$ 整体 Φ 的减少量，我们再变一下，总时间中 Φ 减小量肯定小于增加的量，因为 Φ 最开始为 0。所以，总运行时间小于 $n + md_{max} + r +$ 整体 Φ 增量。那么 Φ 总共增加的量 $n*1$ (每次插入操作增加为 1) + MD_MAX(每次找最小操作中增加为 d) + 3R(每次修改操作中为 3，这个值有些问题)。所以整体运行时间为: $n + md_{max} + r + n + md_{max} + 3r = 2n + 2md_{max}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

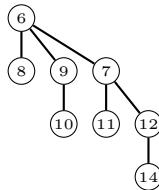


而对于斐波那契堆，9 有可能损失儿子，7 有可能丢了两个儿子，最终这个树被砍成如图所示：

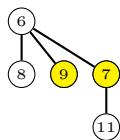


我们砍的没太狠，每个节点最多丢失一个儿子，所以 D 比二项树大一点， $\log_{\Phi} n \geq d \geq \log_2 n$ ，其中 $\Phi = \frac{1+\sqrt{5}}{2} = 1.618$

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 1 个儿子有几个孙子，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 1 个儿子有准确的 $l-1$ 个孙子。如图：

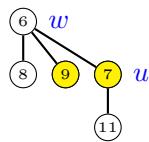


而在斐波那契堆树中，没有这么好的性质，第 1 个儿子至少有 $l-2$ 个孙子，如图



为什么会这样？我们考察 U 时 w 的第 1 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 $\text{DEGREE}(u) = \text{DEGREE}(w) \geq i - 1$ ，而后面的 u 最多失去一个儿子，所以 $\text{DEGREE}(u) \geq i - 2$



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



图 6.11: $|B_1| = 2^1$ and $|F_0| = 1 \geq \phi^0$

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



图 6.12: $|B_2| = 2^2$ 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

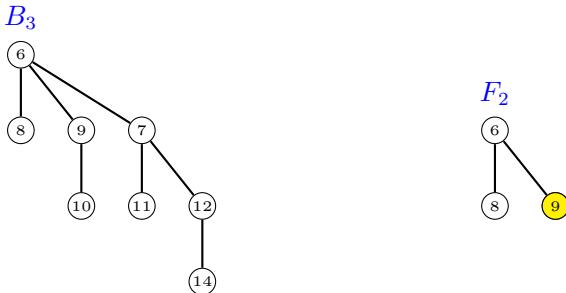
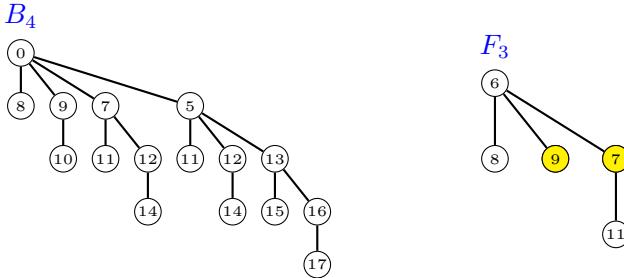
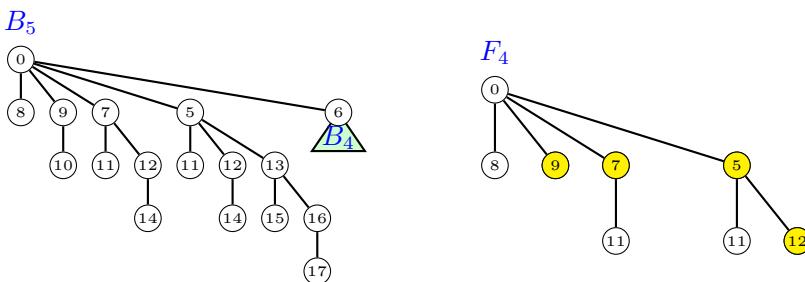


图 6.13: $|B_3| = 2^3$ and $|F_2| = 3 \geq \phi^2$

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

图 6.14: $|B_4| = 2^4$ and $|F_3| = 5 \geq \phi^3$

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 $d_{max} \leq \log_{\Phi} n$ 。

图 6.15: $|B_5| = 2^5$ and $|F_4| = 8 \geq \phi^4$

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

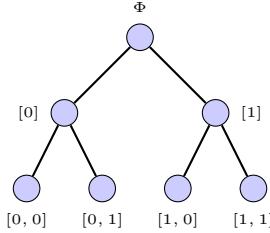
Operation	Linked	Binary	Binomial	Fibonacci
	list	heap	heap	heap
MAKEHEAP	1	1	1	1
INSERT	1	$\log n$	$\log n$	1
EXTRACTMIN	n	$\log n$	$\log n$	$\log n$
DECREASEKEY	1	$\log n$	$\log n$	1
DELETE	n	$\log n$	$\log n$	$\log n$
UNION	1	n	$\log n$	1
FINDMIN	n	1	$\log n$	1
DIJKSTRA	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

那么 DECREASEKEY 的操作变为 $O(1)$ 后对我们的算法有什么影响呢？DIJKSTRA 算法的运行时间从最开始数组结构的 $O(n^2)$ ，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.

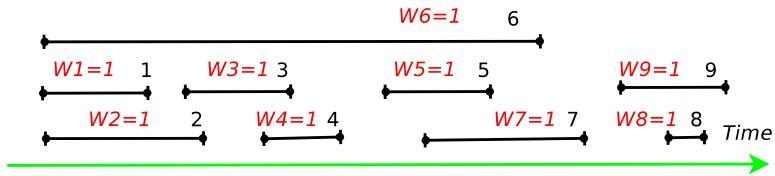
6.4 Greedy 贪心算法

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 $X = [x_1, x_2, \dots, x_n]$ ，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。

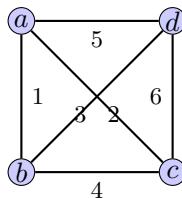


下面我们看一个例子，给我们上课起止的时间，每堂课容纳的学生都为 $w=1$ ，让我们找选择那些课使得不冲突可以使最多的学生上课。我们遵循是 $X = [x_1, x_2, \dots, x_n]$ ，那么

$x_1 \in [1, 2, \dots, 9]$ ，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。

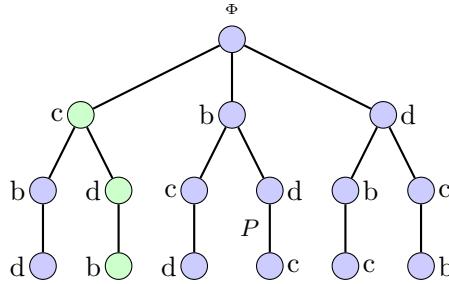


我们再来看 TSP 问题，如图



$X = [x_1, x_2, \dots, x_n]$ ，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。



6.4.1 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。

$$A_1 = [\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}]$$

$$A_2 = [\begin{array}{ccccc} 1 & 4 & 9 & 16 & 25 \end{array}]$$

$$A_3 = [\begin{array}{ccccc} 1 & 8 & 27 & 64 & 125 \end{array}]$$

$$A_4 = [\begin{array}{ccccc} 1 & 16 & 81 & 256 & 625 \end{array}]$$

$$A_5 = [\begin{array}{ccccc} 2 & 6 & 12 & 20 & 30 \end{array}]$$

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, 初始解为

$$A_1 = [\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array}] \quad W_1 = 9$$

$$A_2 = [\begin{array}{ccccc} 1 & 4 & 9 & 16 & 25 \end{array}] \quad W_2 = 7$$

$$A_3 = [\begin{array}{ccccc} 1 & 8 & 27 & 64 & 125 \end{array}] \quad W_3 = 5$$

$$A_4 = [\begin{array}{ccccc} 1 & 16 & 81 & 256 & 625 \end{array}] \quad W_4 = 3$$

$$A_5 = [\begin{array}{ccccc} 2 & 6 & 12 & 20 & 30 \end{array}] \quad W_5 = 1$$

空, 每次取权重最大的那个, 本例中先选 A_1 , 然后选 A_2 , 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 $O(N \log N + N C(N))$, $O(N \log N)$ 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  $A$ ;
```

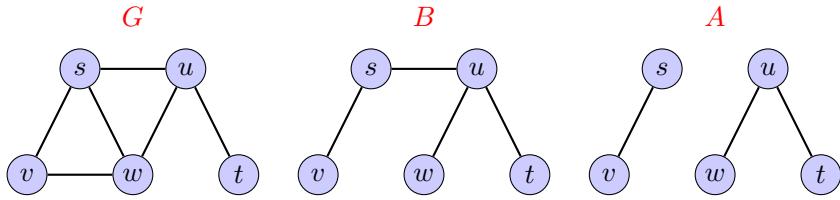
上述算法是对的么? 首先这个算法第一是具有最优子结构的性质, 第二个是具有贪心选择性质。首先 v 是权重最大的向量, 并且 v 本身就是线性无关的。假设 A 是最优的解, 且 A 包含 v 。

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。

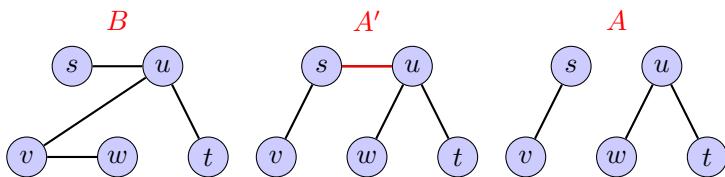
极大线性无关组并不仅仅存在在矩阵中，同样存在在其他问题中，我们称为“拟阵”(MATROID，由 HAUSSLER 在 1935 年提出)。他发现矩阵中的线性无关概念和图论中的森林很像。如果一个图不包含回路，就称为独立。在图当中的无环子图(森林)，连同且无圈称为树，图中 B 所示。如果只要求无圈称为森林，图中 A 所示。

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：



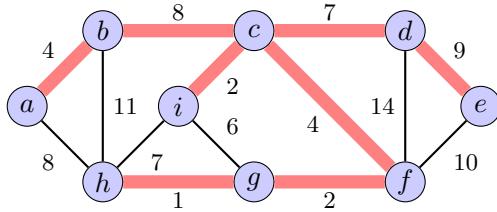
扩展性：如果 A 和 B 都是无环森林，并且 A 是个小森林，那么从 B 中挑出一些边放到 A 中使得 A 还是森林。如图从 B 中添加 E_{su} 到 A 中形成 A'。

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。

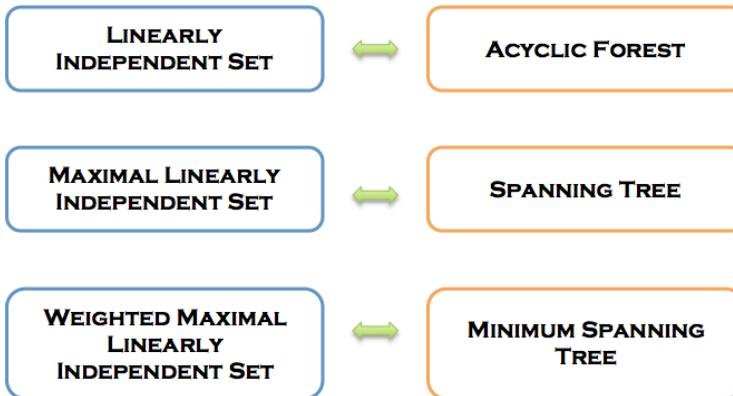


6.4.2 Kruskal 算法

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。加权的极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。



GENERIC SPANNING TREE(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：

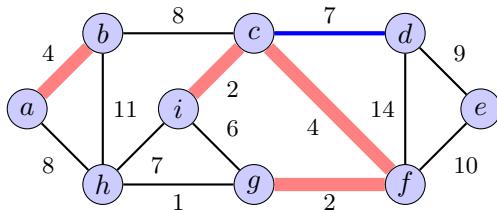


图 6.16: Safe edge

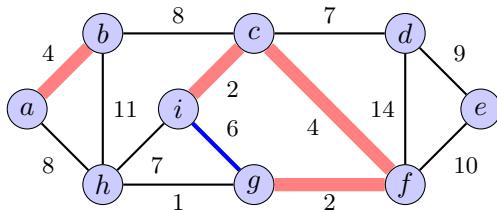


图 6.17: Unsafe edge

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**

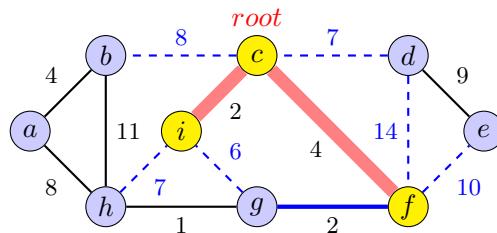
注意红色部分，首先按照权重把所有的边 E 按照非降序排列。一开始把所有节点建立一个集合 SET ，每个节点单独成一个集合。 v 在集合 V 中， u 在集合 U 中，每加一条边，判断 v 和 u 在不在一个集合里，在一个集合中的话就会形成圈。这里使用 $Union - Find$ （并查集）这种数据结构来判断会不会形成环（两个元素是不是在一个集合当中）。

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 $O(m \log m)$ 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 $\aleph(n) = O(\log n)$ ，总体时间为 $O(m \log n)$

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。

6.4.3 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:    $\Pi[v] = \text{NULL}$ ; //  $\Pi(v)$  DENOTES THE PREDECESSOR NODE OF  $v$ 
4:    $PQ.\text{INSERT}(v)$ ; // N TIMES

```

```

5: end for
6:  $key[root] = 0;$ 
7:  $PQ.\text{INSERT}(root);$ 
8: while  $PQ \neq \text{NULL}$  do
9:    $u = PQ.\text{EXTRACTMIN}();$  // N TIMES
10:  for all  $v$  ADJACENT WITH  $u$  do
11:    if  $W(u, v) < key(v)$  then
12:       $\Pi(v) = u;$ 
13:       $PQ.\text{DECREASEKEY}(W(u, v));$  // 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

Operation	Linked	Binary	Binomial	Fibonacci
	list	heap	heap	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	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(m + n \log n)$

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。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。

第七章 线性规划问题 (1)

7.1 回顾

到现在为止，我们可以整理一下这门课到底讲什么。回顾一下已经学过的内容，理清思路。

我们把一个问题建模成组合优化的问题后，首先想到的是能否正面攻击它。第一个观察是，这个问题是否可以分成独立的子问题呢？如果是，基本的思路是规约 (REDUCTION)，如使用 DIVIDE AND CONQUER 算法。如果还有最优子结构的性质，可用动态规划 (DYNAMIC PROGRAMMING) 算法。如果进一步还有贪心选择的性质，可用贪心 (GREEDY) 算法。当然也可以在观察到问题可以分解时，直接应用贪心算法。上节课讲过，可以用部分解的枚举树，直接做贪心算法。以上这些是我们已经学习过的内容，通过对一个问题怎样观察，观察到怎样的性质，来决定我们使用什么算法。

今天所学的内容是我们思路的一个转折点，如果我们观察到一个问题不能够分解，或者我们不愿意进行分解，这个时候怎么办呢？那就只有一种办法，逐步改进法 (IMPROVEMENT)。这里面包括线性规划算法 (LINEAR PROGRAMMING ALGORITHM)，网络流算法 (NETWORK FLOW ALGORITHM)，局部搜索算法 (LOCAL SEARCH ALGORITHMS) 中的蒙特·卡罗方法 (MONTE CARLO METHOD) LS/MC，分支限界方法 (BRANCH AND BOUND METHOD)，等等。

7.2 本章内容

我们先从实际问题讲起，然后讲这些实际问题的数学建模。这些问题包括：饮食问题，最大流问题，最小费用流，多物品流，SAT 问题。接着我们考虑怎样把这些问题建模成数学的线性规划的问题，以及对于线性规划的直观认识和求解方法，包括单纯型算法，内联法，椭球法。

最后，我们介绍领域内一个新的重要的方法，平滑复杂度算法。它完美的回答了，为什么单纯型算法本身具有指数级复杂度，却往往在实际应用中可以做到在线性复杂度的时间内解决问题。

7.3 几个例子

7.3.1 第一个例子：饮食问题

问题：

Food	Energy	Protein	Calcium	Price
Oatmeal	110	4	2	3
Whole milk	160	8	285	9
Cherry pie	420	4	22	20
Pork with beans	260	14	80	19

大家考虑这样一个问题，假设市场上有四种食品，分别给出了它们所含的能量，蛋白质含量，钙离子含量，以及对应的价格。现在有一个家庭主妇，她采购一天的食物，使这些食物加起来至少要含有 2000KCAL 的能量，55G 的蛋白质和 800MG 的钙离子。请问她采用怎样的采购方案能够花费最少？

下面给出这个问题的两个解决方案：

- 方案一：10 份猪肉，花费 190
- 方案二：8 份全奶 + 2 份草莓派，花费 112

很显然，第二种方案比较省钱。请问，现在能不能找到一个更省钱的方案？

将这个问题形式化后，就变得简单了。我们假设买燕麦 x_1 份，全奶 x_2 份，草莓派 x_3 份，猪肉 x_4 份。分别写出它们的约束条件和目标，我们的目标是花的钱最少。

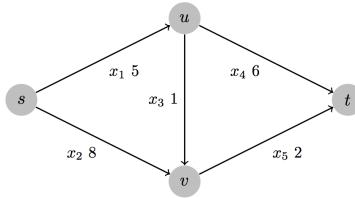
LP Formulation:

$$\begin{aligned}
 & \min \quad 3x_1 + 9x_2 + 20x_3 + 19x_4 && \text{MONEY} \\
 & s.t. \quad 110x_1 + 160x_2 + 420x_3 + 260x_4 \geq 2000 && \text{ENERGY} \\
 & \quad \quad \quad 4x_1 + 8x_2 + 4x_3 + 14x_4 \geq 55 && \text{PROTEIN} \\
 & \quad \quad \quad 2x_1 + 285x_2 + 22x_3 + 80x_4 \geq 800 && \text{CALCIUM} \\
 & \quad \quad \quad x_1, x_2, x_3, x_4 \geq 0
 \end{aligned}$$

因为每一个约束函数以及目标函数都是线性的，所以这样的问题就是线性规划问题。

7.3.2 第二个例子：最大流问题

问题：



如图，假设有四个城市， s , u , v , t 。连线表示，城市之间有道路。连线上的数字表示这条道路每天最多能够运送货物的吨数。比如，城市 s , u 之间有条路，这条路每天最多运 5 吨货物。

如果你是调度员，请设计一个调度方案，使得从 s 到 t 一天之内能够运送的货物越多越好。

现在我们形式化这个问题，将 5 条路的运输量分别为 x_1 , x_2 , x_3 , x_4 , x_5 。然后写出约束条件和目标函数。

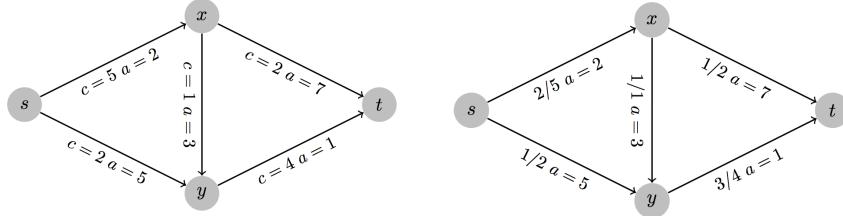
LP Formulation:

$$\begin{array}{ll}
 \max & x_1 + x_2 && \text{OUTPUT FROM } s \\
 \text{s.t.} & x_1 - x_3 - x_4 = 0 && \text{NODE } u \\
 & x_2 + x_3 - x_5 = 0 && \text{NODE } v \\
 & 5 \geq x_1 \geq 0 && \text{EDGE } (s, u) \\
 & \dots && \dots
 \end{array}$$

注意：因为 u , v 是中间节点，所以必须满足运入 u , v 的货物必须在当天运出。另外要满足每条路的运输量不超过其最大运量。我们的目标是要从 s 运出的货物越多越好。

7.3.3 第三个例子：最小费用流问题

问题：

图 7.1: Cost = $2 \times 2 + 1 \times 7 + 1 \times 3 + 1 \times 5 + 3 \times 1 = 18$

这个问题比刚才那个问题多了一些条件，每条路除了有最大运量的限制，另外对每条路都给出了运输每吨货物的运价。

现在有 D 吨的货物要从 s 运到 t ，怎样运输能使运费最少？比如上图右边的运输方案，花费为 18。

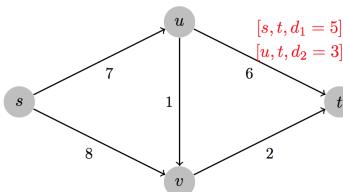
下面形式化这个问题，**LP Formulation**:

$$\begin{aligned} \min \quad & \sum_{(u,v) \in E} a(u,v) f(u,v) \\ \text{s.t.} \quad & f(u,v) \leq C(u,v) \quad \text{FOR EACH } (u,v) \in E \\ & f(u,v) \geq 0 \quad \text{FOR EACH } (u,v) \in E \\ & \sum_{u,(u,v) \in E} f(u,v) = \sum_{w,(v,w) \in E} f(v,w) \quad \text{FOR EACH } v \in V - \{s,t\} \\ & \sum_{v,(s,v) \in E} f(s,v) = d \end{aligned}$$

注意： $a(u,v)$ 表示 u 到 v 之间的运输成本， $f(u,v)$ 表示 u 到 v 之间运送货物的吨数， $C(u,v)$ 表示 u 到 v 之间允许的最大运货量。

7.3.4 第四个例子：多物品流问题

问题：



四个城市的交通网络，要从 s 到 t 运送 5 吨货物 D_1 ，同时要从 u 到 t 运送 3 吨货物 D_2 。这个问题与之前不同的是，要同时运送两种货物。

LP Formulation:

$$\begin{aligned}
 \max \quad & 0 \\
 s.t. \quad & \sum_{i=1}^k f_i(u, v) \leq c(u, v) \quad \text{FOR EACH } (u, v) \\
 & f_i(u, v) \geq 0 \quad \text{FOR EACH } i, (u, v) \\
 & \sum_{u, (u, v) \in E} f_i(u, v) = \sum_{w, (v, w) \in E} f_i(v, w) \quad \text{FOR EACH } i, v \in V - \{s_i, t_i\} \\
 & \sum_{v, (s_i, v) \in E} f_i(s_i, v) = d_i \quad \text{FOR EACH } i
 \end{aligned}$$

注意：目标函数为 $\max 0$, 这个看起来有些奇怪, 其实这个问题只需要解出满足这些约束条件的解就可以了。

讲到这里我们就可以回答为什么这堂课是我们思维的一个转折点了。我们以前遇到一个问题可以分解, 我们就很高兴, 有很多的办法去解决, 但是上面这个问题是不容易分解的。值得指出的是, 线性规划是到目前为止唯一的一个多项式算法。也就是说, 我们或许可以写出一个多物品的贪心算法, 但它的时间复杂度不是多项式时间的。

7.3.5 第五个例子: 多物品流问题

SAT 问题值得讲, 因为它是一个整数线性规划问题。

问题:

INPUT:

A SET OF m CONJUNCTION NORMAL FORMULA (CNF) CLAUSES OVER n BOOLEAN VARIABLES x_1, x_2, \dots, x_n

OUTPUT:

WHETHER ALL CLAUSES CAN BE SATISFIED BY AN TRUE/FALSE ASSIGNMENT OF THE n VARIABLES.

- A SAT INSTANCE:

$$\begin{aligned}
 \Phi = & (x_1 \vee \neg x_2 \vee x_3) \wedge \\
 & (\neg x_1 \vee x_2 \vee \neg x_3) \wedge \\
 & (x_1 \vee x_2 \vee \neg x_3)
 \end{aligned}$$

- AN ASSIGNMENT TO MAKE ALL CLAUSES TRUE:

$$x_1 = \text{TRUE}, x_2 = \text{TRUE}, x_3 = \text{TRUE}$$

注意：每个 x_i 都为一个布尔变量，取值只能是 TRUE 或者 FALSE 两者之一。用 \vee 连起来的式子叫做子式，这些子式连接成一个和取范式。能不能找到一种组合，使得所有的子式都为 TRUE？这个是可以做到的，比如我们取 $x_1 = \text{TRUE}, x_2 = \text{TRUE}, x_3 = \text{TRUE}$ ，就可以使得三个子句都为 TRUE。

对于这个问题，诸位怎样求解呢？先看能不能分解？答案是肯定的，我们下堂课介绍分解的方法。我们先来看不分解的方法，把这个问题写成下面的形式：

LP Formulation:

$$\begin{aligned} \max \quad & c_1 + c_2 + c_3 \\ s.t. \quad & x_1 + (1 - x_2) + x_3 \geq c_1 \\ & (1 - x_1) + x_2 + (1 - x_3) \geq c_2 \\ & x_1 + x_2 + (1 - x_3) \geq c_3 \\ & x_1, x_2, x_3 = 0/1 \\ & c_1, c_2, c_3 = 0/1 \end{aligned}$$

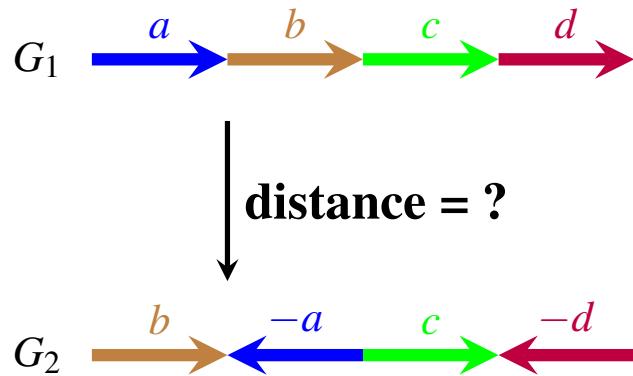
注意：第一行的 $\max c_1 + c_2 + c_3$ 表明， c_1, c_2, c_3 均取 1。所以对于这个例子来说，当且仅当 $c_1 + c_2 + c_3 = 3$ 时才成立。

在这个问题中每个 x_i 只能取 0 或 1 两个值，这样的问题叫做整数线性规划。

7.3.6 补充例子：基因组重排问题

问题：

- THE MINIMUM NUMBER OF OPERATIONS TO TRANSFORM G_1 INTO G_2

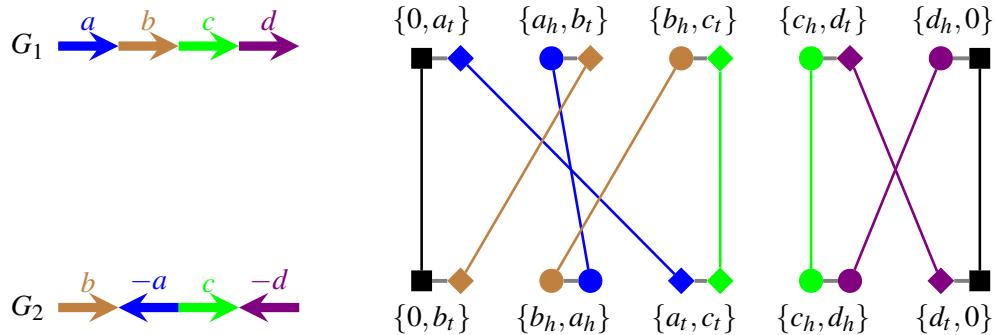


OPERATIONS: REVERSE A FRAGMENT OF THE GENOME;

假设人的基因组 G_1 上有 a, b, c, d 这样四个基因，老鼠的基因组 G_2 上 a, d 这两个基因是反过来的，问基因组 G_1 翻转几次可以得到基因组 G_2 ？

补充：如果诸位熟悉计算机历史的话，这个问题是否曾经见过？这是 BILL GATES 退学前和老师一起研究的一个翻煎饼的问题。（这里有一个 BILL 的故事和翻煎饼的说明）

形式化模型：我们给这个问题建立一个邻接图形式化的模型。

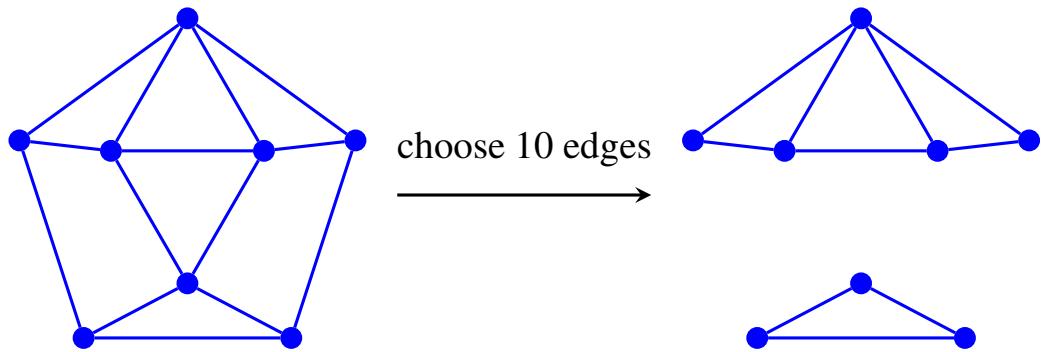


- 可以证明：DCJ DISTANCE = (#ADJACENCIES) – (#CYCLES).
- 在这个例子中，DCJ DISTANCE = 3
- 所以在这个图中，因为 ADJACENCIES 为固定值，所以求最小的 DCJ DISTANCE 的问题就转换为求图中有多少个环的问题。

问题描述：

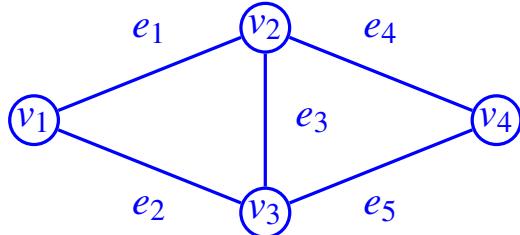
Problem: GIVEN AN UNDIRECTED GRAPH $G = (V, E)$, TO CHOOSE k EDGES AND REMOVE OTHERS, SUCH THAT THE NUMBER OF CONNECTED COMPONENTS IN THE REMAINING GRAPH IS MAXIMIZED.

(Formulate this problem as an ILP.)



假定要从一个图中选出一些边，使得连通的点越多越好。

例子：比如我们要选出 3 条边，使得连通的点越多越好。



- 用 x_i 表示第 i 条边是否被选择，选则为 1，不选为 0。

$$x_1 + x_2 + x_3 + x_4 + x_5 = 3$$

- 用 y_i 来表示第 i 个顶点的标签。

$$1 \leq y_1 \leq 1$$

$$1 \leq y_2 \leq 2$$

$$1 \leq y_3 \leq 3$$

$$1 \leq y_4 \leq 4$$

- 如果一条边被选择，它的两个顶点需要有相同的标签。

$$y_1 \leq y_2 + 1 \cdot (1 - x_1); \quad y_2 \leq y_1 + 2 \cdot (1 - x_1) \quad (\text{for } e_1)$$

$$y_1 \leq y_3 + 1 \cdot (1 - x_2); \quad y_3 \leq y_1 + 3 \cdot (1 - x_2) \quad (\text{for } e_2)$$

$$y_2 \leq y_3 + 2 \cdot (1 - x_3); \quad y_3 \leq y_2 + 3 \cdot (1 - x_3) \quad (\text{for } e_3)$$

$$y_2 \leq y_4 + 2 \cdot (1 - x_4); \quad y_4 \leq y_2 + 4 \cdot (1 - x_4) \quad (\text{for } e_4)$$

$$y_3 \leq y_4 + 3 \cdot (1 - x_5); \quad y_4 \leq y_3 + 4 \cdot (1 - x_5) \quad (\text{for } e_5)$$

- 同样的连通单元中的顶点有相同的标签。

- SINCE ALL VERTICES HAVE DISTINCT UPPER BOUNDS, IN EACH CONNECTED COMPONENT, AT MOST ONE VERTEX CAN REACH ITS UPPER BOUND. THUS, WE CAN USE THE NUMBER OF VERTICES WHOSE UPPER BOUND IS REACHED, TO COUNT THE NUMBER OF CONNECTED COMPONENTS.
- <2-> WE USE A BINARY VARIABLE z_j TO INDICATE WHETHER THE LABEL OF v_j REACHES ITS UPPER BOUND:

$$1 \cdot z_1 \leq y_1$$

$$2 \cdot z_2 \leq y_2$$

$$3 \cdot z_3 \leq y_3$$

$$4 \cdot z_4 \leq y_4$$

WE CAN VERIFY THAT, $z_j = 1$ ONLY IF $y_j = j$, I.E., THE LABEL OF v_j REACHES ITS UPPER BOUND.

- THE OBJECTIVE FUNCTION OF THE ILP FORMULATION CAN BE SET TO MAXIMIZE THE NUMBER OF VERTICES WHOSE UPPER BOUND CAN BE REACHED:

$$\max z_1 + z_2 + z_3 + z_4$$

7.4 动态规划问题的历史回顾

看了这些问题之后，大家一定会有感觉，我们生活中的很多问题，都可以写成一个最优化的问题，不论我们想要最优化的是目标函数，还是问题的约束条件，它们都是一些线性组合。这是一类很常见的问题，我们简单看一下这类问题的历史。

7.4.1 问题提出：

1946 年，在美国空军的指挥部，GEORGE B. DANTZIG 作为数学顾问，他的同事问了他一个问题关于部队部署的问题：怎样快速调度训练计划和物流。在前电子计算机的时代，人们是用穿孔卡和一些模拟的设备来制造机械式的计算机器。“PROGRAM”这个词用来表示规划和调度，而不是现在所说的编程。在 1947 年，DANTZIG 把这个问题抽象成了一个数学问题。

7.4.2 有关算法和分析：

1949 年，DANTZIG 又提出了单纯形算法。大家回忆一下我们第一堂课就讲过的，解决一个问题的三个步骤。首先我们要确定要解决的实际的问题是什么，然后将它用公式描述为一个数学问题，最后提出算法。DANTZIG 的这个故事向我们清晰的展示了这三个步骤，首先在 1946 年他的同事向他提出了这个问题，1947 年他为这个问题建立了数学模型，然后在 1949 年，他提出了解决这个问题的单纯形算法（SIMPLEX ALGORITHM）。

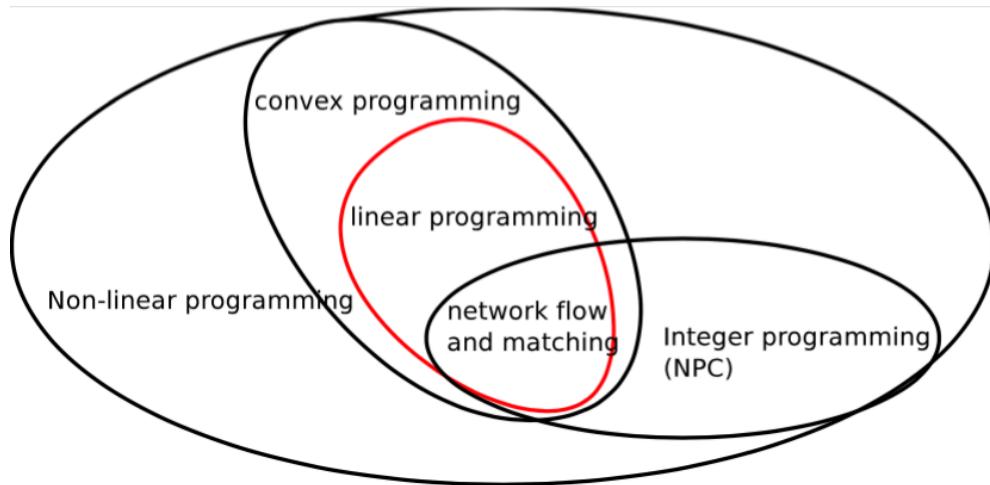
人们发现单纯形算法太好了，运行的飞快，所有的问题都解决的非常好。所以人们一直以为这个算法是多项式时间复杂度的算法，然而很不幸，在 1971 年，KLEE 和 MINTY 提出了第一个反例，说明单纯型算法不是多项式时间复杂度。这下大家产生了怀疑，线性规划问题应该是一个难以分解（NP HARD）的问题，因为单纯型算法的成功，很多人都在用它。典型的例子是，1975 年 L. V. KANTOROVICH 和 T. C. KOOPMANS 将线性规划的方法用于经济学领域的资源分配问题上，得到了诺贝尔经济学奖。

在 1971 年后，虽然人们怀疑线性规划问题是 NP 完全问题，但是在 1979 年，峰回路转，苏联科学家 L. G. KHANCHIAN 提出了多项式时间复杂度的椭球算法，这

是一个可以分解的问题。椭球法理论上非常漂亮，常常用在其他领域，但是实际效果却很慢，不实用。1984 年，另一个苏联科学家 N. KARMARKAR 提出了内联法，是多项式时间复杂度的算法，同时运行起来也非常快。

但是这个问题还是没有完全解决，人不知道为什么单纯型算法理论上是指指数型复杂度，但实际运行时却常常是多项式时间复杂度。直到 2001 年，D. SPIELMAN 和 S. TENG 提出了平滑型复杂度理论 (SMOOTHED COMPLEXITY)，完美地证明了这个问题。

7.4.3 NLP, Convex Programming, LP, Network flow, 和 ILP:



我们发现生活中的很多问题都能够描述成一个优化的问题，诸位以后研究生生涯中碰到的问题可能 80% 的情况是优化问题，所以最外面这个大圈，就可以覆盖大家研究问题 80% 的情况。这里面有一类凸的规划是非常特殊的，目标函数是凸的，可行域也是凸的，这就是凸规划问题。凸规划问题里面又有一个线性规划问题，目标函数和约束都是线性组合。还有另一类问题是整数型规划问题，我们学过的网络流和匹配这两类问题是整数型规划问题，但是它们又很特殊，网络流问题中假设 $0 \leq x_i \leq 1$ ，但是线性规划问题 x_i 只有 0 或 1 两种取值。

7.4.4 求解 LP 问题的工具：

现在有很多求解线性规划的软件包，典型的一个包是 GLPK，还有一个是 GUROBI，是目前运行最快的软件包。下面给大家看两个例子：

55:25-第一部分结束，演示

7.5 线性规划问题的求解

前面介绍了建模的方法，下面介绍求解的过程。

线性规划问题的形式有：一般形式，标准形式和松弛形式。

7.5.1 一般形式线性规划问题

- GENERAL FORM: MIXTURE OF LINEAR INEQUALITIES AND EQUALITIES

$$\begin{aligned} \min \quad & c_1x_1 + c_2x_2 + \dots + c_nx_n \\ s.t. \quad & a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n \geq b_i \quad i \in M \\ & a_{j1}x_1 + a_{j2}x_2 + \dots + a_{jn}x_n = b_j \quad j \in \overline{M} \\ & x_i \geq 0 \quad i \in N \end{aligned}$$

7.5.2 标准形式线性规划问题

- STANDARD FORM: LINEAR INEQUALITIES;

$$\begin{aligned} \min \quad & c_1x_1 + c_2x_2 + \dots + c_nx_n \\ s.t. \quad & a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq b_2 \\ & \dots \quad \dots \quad \dots \quad \dots \\ & a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq b_m \\ & x_i \geq 0 \quad \text{FOR } \forall i \end{aligned}$$

- STANDARD FORM IN MATRIX LANGUAGE:

$$\min \quad \mathbf{c}^T \mathbf{x}$$

$$s.t. \quad \mathbf{A}\mathbf{x} \leq \mathbf{b}$$

$$\mathbf{x} \geq \mathbf{0}$$

- HERE $\mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}$, $\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}.$$

注意：在一般形式的线性规划问题中，约束条件函数可能有等式和不等式的形式，我们希望把这些约束条件统一写成标准的方程形式，比如都写成用 \leq 表示的方程，并且满足 $x_i \geq 0$ 。我们做如下变换：

一般形式转换为标准形式：

- TRANSFORMATIONS:

1. **Variables:** A FREE VARIABLE \Rightarrow TWO NON-NEGATIVEIVE VARIABLES;

x_i MAY OR MAY NOT BE POSITIVE \Rightarrow REPLACING x_i WITH $x'_i - x''_i$

AND ADDING CONSTRAINTS: $x'_i \geq 0; x''_i \geq 0$

2. **Constraints:** AN EQUALITY \Rightarrow TWO INEQUALITIES;

$a_{j1}x_1 + a_{j2}x_2 + \dots + a_{jn}x_n = b_j \Rightarrow$

$a_{j1}x_1 + a_{j2}x_2 + \dots + a_{jn}x_n \geq b_j$

$a_{j1}x_1 + a_{j2}x_2 + \dots + a_{jn}x_n \leq b_j$

7.5.3 松弛形式线性规划问题

- SLACK FORM: LINEAR EQUALITY;

$$\begin{aligned}
 \min \quad & c_1x_1 + c_2x_2 + \dots + c_nx_n \\
 \text{s.t.} \quad & a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\
 & a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\
 & \dots \quad \dots \quad \dots \quad \dots \\
 & a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m \\
 & x_i \geq 0 \text{ FOR } \forall i
 \end{aligned}$$

- SLACK FORM IN MATRIX LANGUAGE:

$$\begin{aligned}
 \min \quad & \mathbf{c}^T \mathbf{x} \\
 \text{s.t.} \quad & \mathbf{Ax} = \mathbf{b} \\
 & \mathbf{x} \geq \mathbf{0}
 \end{aligned}$$

注意：松弛形式也是一种比较常见的形式，所有的约束条件方程都采用等式的形式。假如一开始我们写出的目标函数是 $a_{j1}x_1 + a_{j2}x_2 + \dots + a_{jn}x_n \leq b_j$ 的形式，可以引入一个整数 s ，使之变为等式形式， s 称为松弛变量。

- TRANSFORMATIONS:

1. **Variables:** CHANGING “INEQUALITY ON PARTIAL SOLUTION (x_1, \dots, x_n) ” TO “EQUALITY ON FULL SOLUTION (s, x_1, \dots, x_n) ” BY INTRODUCING A SLACK VARIABLE s .

$$\begin{aligned}
 a_{j1}x_1 + a_{j2}x_2 + \dots + a_{jn}x_n &\leq b_j \Rightarrow \\
 a_{j1}x_1 + a_{j2}x_2 + \dots + a_{jn}x_n + s &= b_j
 \end{aligned}$$

2. **Constraint:** $s \geq 0$. (s IS CALLED A SLACK VARIABLE)

例子：标准形式 VS 松弛形式

- STANDARD FORM:

$$\begin{aligned} -x_3 + 2x_4 &\leq 2 \\ 3x_3 - 2x_4 &\leq 6 \\ x_3, x_4 &\geq 0 \end{aligned}$$

- SLACK FORM:

$$\begin{aligned} x_1 &- x_3 + 2x_4 = 2 \\ x_2 + 3x_3 - 2x_4 &= 6 \\ x_1, x_2, x_3, x_4 &\geq 0 \end{aligned}$$

注意：大家不要小看这种变换，把不等式的约束变为等式的约束，是线性规划中常常要考虑的问题。

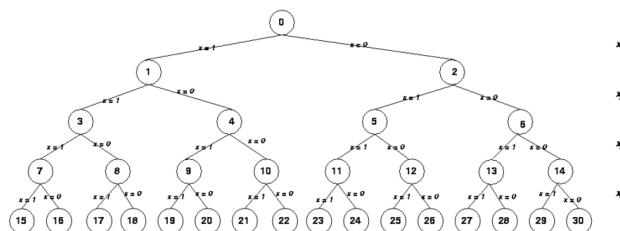
7.5.4 求解方法

下面我们讨论怎样求解。

$$\begin{aligned} \max \quad & c_1 + c_2 + c_3 \\ \text{s.t.} \quad & x_1 + (1 - x_2) + x_3 \geq c_1 \\ & (1 - x_1) + x_2 + (1 - x_3) \geq c_2 \\ & x_1 + x_2 + (1 - x_3) \geq c_3 \\ & x_1, x_2, x_3 = 0/1 \\ & c_1, c_2, c_3 = 0/1 \end{aligned}$$

方法一：首先观察解的形式， $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4](x_i = 0/1)$ ，一定可以画出一个部分解的枚举树，叶子节点即为所求的完整解。如下：

- SOLUTION: $X = [x_1, x_2, \dots, x_n], x_i = 0/1$
- PARTIAL SOLUTION ENUMERATION TREE:

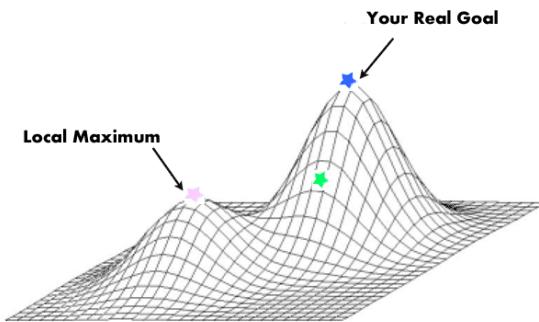


- WE WILL TALK ABOUT “INTELLIGENT ENUMERATION”, SAY BRANCH-AND-BOUND, AND BACKTRACKING, LATER.

注意：我们的目标就是从 8 个完整解中找出最好的解。另外除了使用枚举的方法，贪心法，回溯法，分支限界法均可以使用。所以，我们的解题思路是，只要解的形式为 $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4](x_i = 0/1)$ ，就可以使用各种方法来找最优解。

扩展：如果 $x_i \in [0, 1]$ ，我们可以用分支限界的方法。分支限界实际上就是枚举，只不过要枚举的聪明一点，要进行大幅度的剪支，加快运行速度。这种方法会在后续学习中讨论。

方法二：我们换一种思路，考虑完全解的情况。我们不对问题进行分类，也不对解进行分类，每次都考虑完整的解。画出解空间如下，



每个格子点都是一个完整解，如果两个格子点之间 \mathbf{X} 的距离如果非常小，表示对一个解进行很小的改变就可以到达另一个解。

7.5.5 线性规划问题的直观认识

我们讨论第二种求解方法，每次都考虑完整解的情况。考察一个线性规划如下，

$$\min \quad \mathbf{c}^T \mathbf{x}$$

$$s.t. \quad \mathbf{Ax} = \mathbf{b}$$

$$\mathbf{x} \geq \mathbf{0}$$

大家观察一下会发现，我们已经学过如何求解 $\mathbf{Ax} = \mathbf{b}$ ，但没有学过优化 $\mathbf{c}^T \mathbf{x}$ ，另外还要考虑 $\mathbf{x} \geq \mathbf{0}$ 这个约束条件。下面我们分别来讨论。

7.5.6 约束条件 $x \geq 0$ 的影响:

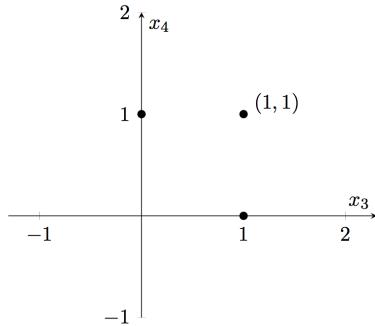
回顾线性方程 $\mathbf{Ax} = \mathbf{b}$ 的求解:

$$\begin{aligned} x_1 & - x_3 + 2x_4 = 2 \\ x_2 + 3x_3 - 2x_4 & = 6 \\ 2x_1 + x_2 + x_3 + 2x_4 & = 10 \end{aligned}$$

通过高斯消元法, 我们得到:

$$\begin{aligned} x_1 & - x_3 + 2x_4 = 2 \\ x_2 + 3x_3 - 2x_4 & = 6 \end{aligned}$$

4 个未知量, 两个方程, 我们只能确定两个独立变量, 得到的解平面上的任意一个点都是这个方程组的完整解。如下图:



约束条件 $x \geq 0$ 的影响:

下面我们考虑对这个线性方程加一个约束条件,

- 例如: $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$

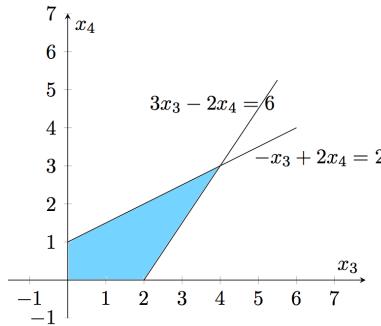
$$\begin{aligned} x_1 & - x_3 + 2x_4 = 2 \\ x_2 + 3x_3 - 2x_4 & = 6 \\ 2x_1 + x_2 + x_3 + 2x_4 & = 10 \\ x_1, x_2, x_3, x_4 & \geq 0 \end{aligned}$$

- 通过高斯消元法，我们得到：

$$\begin{array}{rcl} x_1 & - & x_3 + 2x_4 = 2 \\ x_2 + 3x_3 - 2x_4 = 6 \\ \textcolor{red}{x_1, x_2, x_3, x_4 \geq 0} \end{array}$$

- 得到一个线性不等式组：

$$\begin{array}{rcl} -x_3 + 2x_4 \leq 2 \\ 3x_3 - 2x_4 \leq 6 \\ \textcolor{red}{x_3, x_4 \geq 0} \end{array}$$



对于这个线性不等式组，分别画出 $x_3 + 2x_4 = 2$ 和 $3x_3 - 2x_4 = 6$ 之后，就可以确定出它的解域如上图的阴影区域。与刚才的平面解域不同，这个解域为多胞形。

多胞形 \Leftrightarrow 可行域

定理： Any polytope $P \subset \mathbf{R}^{n-m}$ corresponds to the feasible region of a linear program $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$ (denoted as $F = \{\mathbf{x} : \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$), and vice versa.

在空间 \mathbf{R}^{n-m} 中，任何一个多胞形都对应着 $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$ 的一个可行域 $F = \{\mathbf{x} : \mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ ，反之亦然。

也就是说，等式约束条件： $x_2 + 3x_3 - 2x_4 = 6$ ，等价为不等式约束条件 $3x_3 - 2x_4 \leq 6$ 。

下面我们证明一下这个定理。

证明：

注意：变量数为 N ，约束条件有 M 个。

Proof: feasible region \Rightarrow polytope:

- CONSIDER A **feasible full solution x** OF THE FOLLOWING LP:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\ &\dots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &= b_m \\ x_1, x_2, \dots, x_n &\geq 0 \end{aligned}$$

- 一定可以使用高斯消元法化简为：

$$\begin{aligned} x_1 &+ a'_{1,m+1}x_{m+1} + \dots + a'_{1n}x_n = b'_1 \\ x_2 &+ a'_{2,m+1}x_{m+1} + \dots + a'_{2n}x_n = b'_2 \\ &\dots &\dots \\ x_m &+ a'_{m,m+1}x_{m+1} + \dots + a'_{mn}x_n = b'_m \\ x_1, x_2, x_m, x_{m+1}, \dots, x_n &\geq 0 \end{aligned}$$

- BY **removing positive variables** x_1, x_2, \dots, x_m , WE HAVE THE FOLLOWING **linear inequalities**:

$$\begin{aligned} a'_{1,m+1}x_{m+1} + \dots + a'_{1n}x_n &\leq b'_1 \\ a'_{2,m+1}x_{m+1} + \dots + a'_{2n}x_n &\leq b'_2 \\ &\dots \\ a'_{m,m+1}x_{m+1} + \dots + a'_{mn}x_n &\leq b'_m \\ x_{m+1}, \dots, x_n &\geq 0 \end{aligned}$$

- DEFINE A POLYTOPE $P \subset \mathbf{R}^{n-m}$ AS THE INTERSECTION OF m HALF-SPACES:

$$HS_j : a'_{j,m+1}x_{m+1} + \dots + a'_{jn}x_n \leq b'_j, 1 \leq j \leq m. \text{ (BY } x_j \geq 0\text{)}$$

- THUS, **any feasible full solution** $x = (x_1, x_2, \dots, x_n) \Rightarrow$ **partial solution** $\mathbf{x}_N = (x_{m+1}, \dots, x_n) \in P$.

Proof: polytope \Rightarrow feasible region:

注意：反过来，我们加入一个松弛变量 s_j ，将一个多胞形变成了 $AX = B$ 的形式。

- BASIC IDEA: CHANGING **inequality** TO **equality** THROUGH INTRODUCING **slack variables**.

— SUPPOSE P IS THE INTERSECTION OF m HALF-SPACES (INEQUALITIES),
SAY:

$$HS_j : a_{j1}x_1 + a_{j2}x_2 + \dots + a_{jn}x_n \leq b_j \quad (1 \leq j \leq m)$$

— INTRODUCING A NON-NEGATIVE SLACK VARIABLE s_j TO EACH INEQUALITY, WE HAVE:

$$a_{j1}x_1 + a_{j2}x_2 + \dots + a_{jn}x_n + s_j = b_j \quad (s_j \geq 0)$$

- THUS WE CHANGE

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &\leq b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &\leq b_2 \\ &\dots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &\leq b_m \\ x_1, x_2, \dots, x_n &\geq 0 \end{aligned}$$

INTO

$$\begin{aligned} s_1 &+ a_{1,1}x_1 + \dots + a_{1n}x_n = b_1 \\ s_2 &+ a_{2,1}x_1 + \dots + a_{2n}x_n = b_2 \\ &\dots &\dots \\ s_m &+ a_{m,1}x_1 + \dots + a_{mn}x_n = b_m \\ s_1, s_2, s_m, x_1, \dots, x_n &\geq 0 \end{aligned}$$

- THUS, A **partial solution** $(x_1, x_2, \dots, x_n) \in P \Rightarrow$ A **feasible full solution** $(s_1, s_2, \dots, s_m, x_1, x_2, \dots, x_n) \geq 0$.

我们引入一些记号、常用的称谓。

记号：

- 超平面: $\{\mathbf{x} : a_1x_1 + a_2x_2 + \dots + a_nx_n = b\}$ (线性等式约束)

- 半空间: $\{\mathbf{x} : a_1x_1 + a_2x_2 + \dots + a_nx_n \leq b\}$ (线性不等式约束)
- 多面体: 一些半空间的组合;
- 多胞形: 有界非空的多面体;

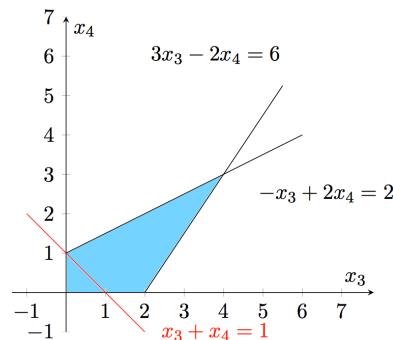
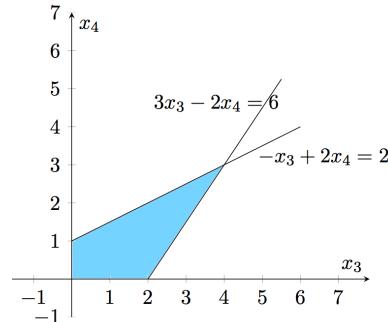
7.5.7 约束条件 $\min \mathbf{c}^T \mathbf{x}$ 的影响:

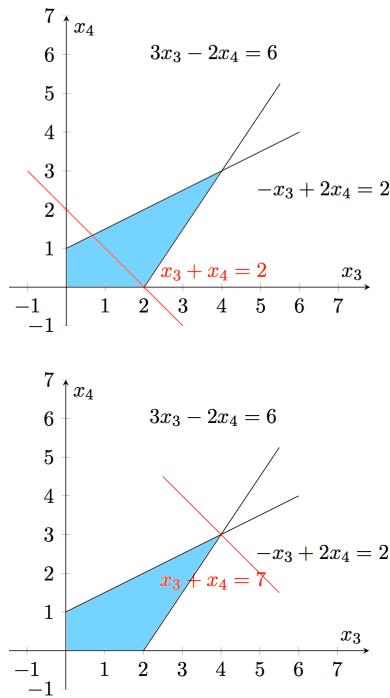
我们来看第二个差异, 我们想要求下面这个问题,

$$\begin{aligned} \max \quad & x_3 + x_4 \\ s.t. \quad & -x_3 + 2x_4 \leq 2 \\ & 3x_3 - 2x_4 \leq 6 \\ & x_3, x_4 \geq 0 \end{aligned}$$

对这个不等式, 我们已经不怕它了, 只要添加变量, 就可以把不等式变为等式。

画出它的可行域如下:





在可行域的所有点中，我们要挑一个满足约束，并使得 $\max x_3 + x_4$ 的点。从上面的图中，可以很直观的看出，只要找顶点就可以了，不用考虑内部点。

7.6 改进法求线性规划问题

经过上面这些准备工作之后，接下来我们来看怎样正式的求解线性规划。

改进策略就三句话：

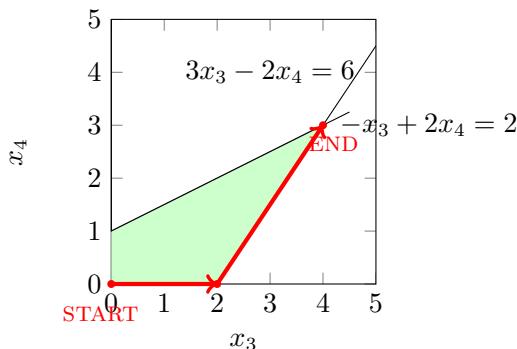
第一步，随便找一个初始的解；第二步，从这个初始点开始往一个方向稍微改进一点；第三步，判断改进是否足够好，若足够好，则返回。

大家注意，如果大家在学运筹学，或者数值计算方法的话，那些里面的方法全都是这个套路。比如牛顿法，梯度下降法等等，都不考虑是否可分的问题，都是通过这样一种改进初始解的方法做。

我们过去的思路是，拿到一个问题，我们先琢磨这个问题好不好分。现在拿到这个问题，我们先不考虑怎么分（讲完线性规划后会讲怎么分）。

下面这个例子说明这个套路。

7.6.1 例子：



IMPROVEMENT()

```

1: x = x0; //STARTING FROM A VERTEX;
2: while TRUE do
3:   x = IMPROVE(x); //MOVE TO ANOTHER VERTEX VIA AN EDGE;
4:   if STOPPING(x) then
5:     BREAK; //STOP WHEN x IS OPTIMAL
6:   end if
7: end while
8: return x;

```

第一步：找到一个初始可行解，如 $0,0$ ；第二步：找到一个结果比 $0,0$ 改进一点的点，比如 $2,0$ ；发现还可以再改进，于是接着再改进一点，找到 $4,3$ ；第三步：发现没有办法再改进了，则返回。

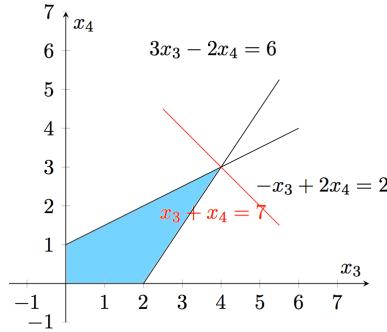
注意：琢磨清楚三句话，初始点怎么找，怎样改进，结束条件是什么。线性规划的单纯型算法、牛顿法之类的算法全都是这三句话。

我们来看一下，单纯型算法回答了哪些问题呢？

1. 为什么我们只用考虑多胞形的顶点就够了呢？
2. 怎样选择初始点？
3. 怎样改进？
4. 什么时候结束？

我们来一一回答。

7.6.2 为什么我们只用考虑多胞形的顶点就够了呢？

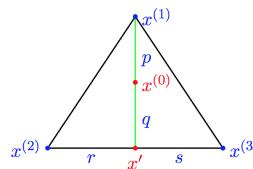


直观地看，我们发现最优解一定在顶点上，不需要考虑内部点，问题得到了极大的简化。

定理：THERE EXISTS A VERTEX IN \mathbf{P} THAT TAKES THE OPTIMAL VALUE.

证明：

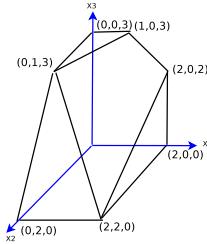
- SINCE \mathbf{P} IS A BOUNDED CLOSE SET, $\mathbf{c}^T \mathbf{x}$ REACHES ITS OPTIMUM IN \mathbf{P} .
- DENOTE THE OPTIMAL SOLUTION AS $\mathbf{x}^{(0)}$. WE WILL SHOW THERE IS A VERTEX AT LEAST AS GOOD AS $\mathbf{x}^{(0)}$. WHY?
 - $\mathbf{x}^{(0)}$ CAN BE REPRESENTED AS THE CONVEX COMBINATION OF VERTICES OF \mathbf{P} , I.E. $\mathbf{x}^{(0)} = \lambda_1 \mathbf{x}^{(1)} + \lambda_2 \mathbf{x}^{(2)} + \dots + \lambda_k \mathbf{x}^{(k)}$, WHERE $\lambda_i \geq 0$, $\lambda_1 + \dots + \lambda_k = 1$. (SEE APPENDIX FOR DETAILS.)
 - THUS $\mathbf{c}^T \mathbf{x}^{(0)} = \lambda_1 \mathbf{c}^T \mathbf{x}^{(1)} + \lambda_2 \mathbf{c}^T \mathbf{x}^{(2)} + \dots + \lambda_k \mathbf{c}^T \mathbf{x}^{(k)}$
 - LET $\mathbf{x}^{(i)}$ BE THE VERTEX WITH THE MINIMAL OBJECTIVE VALUE $\mathbf{c}^T \mathbf{x}^{(i)}$;
 - $\mathbf{c}^T \mathbf{x}^{(0)} = \lambda_1 \mathbf{c}^T \mathbf{x}^{(1)} + \lambda_2 \mathbf{c}^T \mathbf{x}^{(2)} + \dots + \lambda_k \mathbf{c}^T \mathbf{x}^{(k)} \geq \mathbf{c}^T \mathbf{x}^{(i)}$.
- THUS, VERTEX $\mathbf{x}^{(i)}$ IS ALSO AN OPTIMAL SOLUTION SINCE $\mathbf{c}^T \mathbf{x}^{(i)} \leq \mathbf{c}^T \mathbf{x}^{(0)}$



- SUPPOSE $\mathbf{x}^{(0)}$ IS AN OPTIMAL SOLUTION.

- CONNECTING $\mathbf{x}^{(0)}$ AND $\mathbf{x}^{(1)}$ WITH A LINE. SUPPOSE THE LINE INTERSECTS LINE SEGMENT $(\mathbf{x}^{(2)}, \mathbf{x}^{(3)})$ AT POINT \mathbf{x}' .
- WE HAVE $\mathbf{x}^{(0)} = \lambda_1 \mathbf{x}^{(1)} + (1 - \lambda_1) \mathbf{x}'$, WHERE $\lambda_1 = \frac{p}{p+q}$.
- WE ALSO HAVE $\mathbf{x}' = \lambda_2 \mathbf{x}^{(2)} + (1 - \lambda_2) \mathbf{x}^{(3)}$, WHERE $\lambda_2 = \frac{r}{r+s}$.
- THUS, WE HAVE $\mathbf{x}^{(0)} = \lambda_1 \mathbf{x}^{(1)} + (1 - \lambda_1)\lambda_2 \mathbf{x}^{(2)} + (1 - \lambda_1)(1 - \lambda_2) \mathbf{x}^{(3)}$.
- SUPPOSE $\mathbf{c}^T \mathbf{x}^{(1)}$ IS THE MINIMUM OF $\mathbf{c}^T \mathbf{x}^{(1)}, \mathbf{c}^T \mathbf{x}^{(2)}, \mathbf{c}^T \mathbf{x}^{(3)}$.
- NOTICE THAT $\lambda_1 + (1 - \lambda_1)\lambda_2 + (1 - \lambda_1)(1 - \lambda_2) = 1$.
- WE HAVE: $\mathbf{c}^T \mathbf{x}^{(1)} \leq \mathbf{c}^T \mathbf{x}^{(0)}$. THUS, A VERTEX $\mathbf{x}^{(1)}$ IS FOUND NOT WORSE THAN $\mathbf{x}^{(0)}$.

7.6.3 怎样选择初始点?



考虑一个三维的多胞体，我们已经知道，对于我们所要求解的问题，只用考虑顶点就够了，内部点不用考虑。

我们人通过看图很容易得到这些顶点，那么，怎样写程序找到多胞体的顶点？

定理：A VERTEX OF \mathbf{P} CORRESPONDS TO A BASIS OF MATRIX \mathbf{A} .

一个多边形的顶点就对应着一个矩阵的基，这下就好做了，只要给一个矩阵求基就行了。

例子：

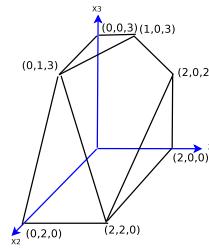
$$\begin{array}{llllll}
 \min & -x_1 & - & 14x_2 & - & 6x_3 \\
 s.t. & x_1 & + & x_2 & + & x_3 & \leq & 4 \\
 & x_1 & & & & & \leq & 2 \\
 & & & & & x_3 & \leq & 3 \\
 & & & & 3x_2 & + & x_3 & \leq & 6 \\
 & x_1 & , & x_2 & , & x_3 & \geq & 0
 \end{array}$$

证明:

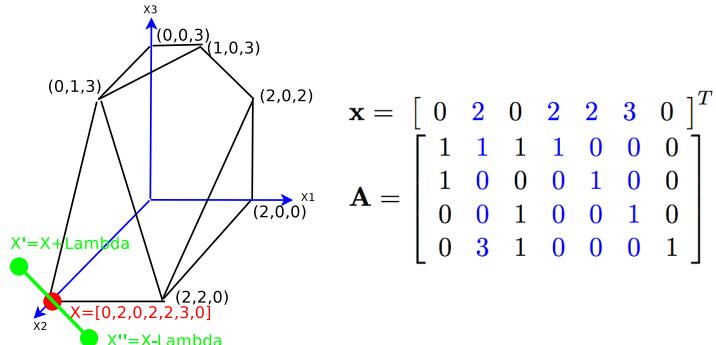
Part 1: Vertex \Rightarrow basic feasible solution

- WE WILL FIRST SHOW THAT **any vertex** OF THE POLYTOPE CORRESPONDS TO A BASIS OF THE MATRIX \mathbf{A} .
- AN EXAMPLE (SLACK FORM):

$$\begin{array}{lllllll} \min & -x_1 & - & 14x_2 & - & 6x_3 & \\ \text{s.t.} & x_1 & + & x_2 & + & x_3 & + x_4 = 4 \\ & x_1 & & & & x_3 & + x_5 = 2 \\ & & & & & x_3 & + x_6 = 3 \\ & 3x_2 & + & x_3 & & & + x_7 = 6 \\ & x_1, & x_2, & x_3, & x_4, & x_5, & x_6, & x_7 \geq 0 \end{array}$$



矩阵描述:



补充: 关于矩阵的描述, 推荐大家看一本书, *Linear Algebra and Its Applications* (David C. Lay)。

- TAKE THE VERTEX $(x_1, x_2, x_3) = (0, 2, 0)$ AS AN EXAMPLE. THE CORRESPONDING FULL SOLUTION IS $(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = (0, 2, 0, 2, 2, 3, 0)$.

- WE WILL SHOW THAT THE COLUMN VECTORS CORRESPONDING TO **non-zero** x_i , I.E. $\{\mathbf{a}_2, \mathbf{a}_4, \mathbf{a}_5, \mathbf{a}_6\}$, ARE LINEARLY INDEPENDENT.
- SUPPOSE $\exists(\lambda_2, \lambda_4, \lambda_5, \lambda_6) \neq 0$ SUCH THAT $\lambda_2\mathbf{a}_2 + \lambda_4\mathbf{a}_4 + \lambda_5\mathbf{a}_5 + \lambda_6\mathbf{a}_6 = 0$, I.E. $\mathbf{A}\lambda = \mathbf{0}$, WHERE $\lambda = [0, \lambda_2, 0, \lambda_4, \lambda_5, \lambda_6, 0]$.
- THEN WE CAN CONSTRUCT **two other points:** $\mathbf{x}' = \mathbf{x} + \theta\lambda$ AND $\mathbf{x}'' = \mathbf{x} - \theta\lambda$.
- IT IS EASY TO DEDUCE THAT BOTH \mathbf{x}' AND \mathbf{x}'' LIE INSIDE P SINCE:
 - $\mathbf{Ax}' = \mathbf{Ax} + \mathbf{0} = \mathbf{b}$ AND $\mathbf{Ax}'' = \mathbf{Ax} - \mathbf{0} = \mathbf{b}$
 - IN ADDITION, WE CAN GUARANTEE $\mathbf{x}' \geq \mathbf{0}$ AND $\mathbf{x}'' \geq \mathbf{0}$ VIA SETTING θ TO BE SUFFICIENTLY SMALL SINCE $\mathbf{x} \geq \mathbf{0}$, AND $\lambda_1 = \lambda_3 = \lambda_7 = 0$.
- CONTRADICTION: IT IS IMPOSSIBLE FOR A VERTEX TO BE CENTER OF TWO INNER POINTS OF \mathbf{P} .

现在我们已经知道顶点可以表示成基，我们来把过去的东西用矩阵的语言表示。对于任何一个顶点，都对应一个基 B ，我们把非基的列都叫做 N 。于是有，

- FOR A VERTEX \mathbf{x} OF THE POLYTOPE, A BASIS \mathbf{B} CAN BE DERIVED VIA EXTRACTING THE COLUMN VECTORS CORRESPONDING TO NON-ZERO x_i . THE NON-BASIS COLUMN VECTORS ARE DENOTED AS \mathbf{N} .
- THEN THE ORIGINAL LP CAN BE REPRESENTED AS:
- HERE, \mathbf{x} IS DECOMPOSED AS $\mathbf{x} = \begin{bmatrix} \mathbf{x}_B \\ \mathbf{x}_N \end{bmatrix}$. THEN WE HAVE $\mathbf{x}_N = \mathbf{0}$, AND $\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b}$ (REASON: $\mathbf{Ax} = \mathbf{b}$, I.E. $\mathbf{Bx}_B + \mathbf{Nx}_N = \mathbf{b}$)
- THE CORRESPONDING OBJECTIVE VALUE IS $\mathbf{c}^T\mathbf{x} = \mathbf{c}_B^T\mathbf{x}_B + \mathbf{c}_N^T\mathbf{x}_N = \mathbf{c}_B^T\mathbf{B}^{-1}\mathbf{b}$.

一个例子：

- FOR A VERTEX $\mathbf{x} = \begin{bmatrix} 0 & 2 & 0 & 2 & 2 & 3 & 0 \end{bmatrix}^T$, THE COLUMNS CORRESPONDING TO NON-ZERO x_i ARE EXTRACTED TO FORM A BASIS $\mathbf{B} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 \end{bmatrix}$.
- LET'S DECOMPOSE $\mathbf{x} = \begin{bmatrix} 0 & 2 & 0 & 2 & 2 & 3 & 0 \end{bmatrix}^T$ ACCORDINGLY INTO $\mathbf{x}_B = [2 \ 2 \ 2 \ 3]^T$ AND $\mathbf{x}_N = [0 \ 0 \ 0]^T$.
- IT IS EASY TO VERIFY THAT $\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b}$. IN THIS EXAMPLE, $\mathbf{b} = \begin{bmatrix} 4 \\ 2 \\ 3 \\ 6 \end{bmatrix}$.

Part 2: Basic feasible solution \Rightarrow vertex

- GIVEN A BASIS \mathbf{B} OF MATRIX \mathbf{A} , WE CALL $\mathbf{x} = \begin{bmatrix} \mathbf{B}^{-1}\mathbf{b} \\ \mathbf{0} \end{bmatrix}$ A **basic solution respect to \mathbf{B}** .
- IF WE FURTHER HAVE $\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{b} \geq \mathbf{0}$, \mathbf{x} IS CALLED A **basic feasible solution respect to \mathbf{B}** .
- WE WILL SHOW THAT A **basic feasible solution \mathbf{x} respect to \mathbf{B}** IS A VERTEX OF THE POLYTOPE \mathbf{P} .

证明:

- IT SUFFICES TO SHOW THAT \mathbf{x} CANNOT BE REPRESENTED AS A CONVEX COMBINATION OF ANY TWO POINTS IN \mathbf{P} .
- BY CONTRADICTION, SUPPOSE THERE ARE TWO DIFFERENT POINTS $\mathbf{x}^{(1)}$ AND $\mathbf{x}^{(2)}$ IN \mathbf{P} SUCH THAT $\mathbf{x} = \lambda_1\mathbf{x}^{(1)} + \lambda_2\mathbf{x}^{(2)}$, WHERE $0 < \lambda_1, \lambda_2 < 1$.
- NOTE THAT $\lambda_1\mathbf{x}_N^{(1)} + \lambda_2\mathbf{x}_N^{(2)} = \mathbf{x}_N = 0$.
- SO $\mathbf{x}_N^{(1)} = \mathbf{x}_N^{(2)} = 0$.

- WE HAVE $\mathbf{x}_B^{(1)} = \mathbf{x}_B^{(2)} = \mathbf{B}^{-1}\mathbf{b} = \mathbf{x}_B$ (BY $\mathbf{A}\mathbf{x}^{(1)} = \mathbf{0}$ AND $\mathbf{A}\mathbf{x}^{(2)} = \mathbf{0}$). A CONTRADICTION.

一个例子：

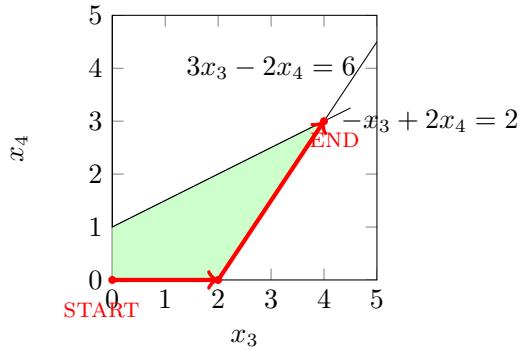
- FOR MATRIX $\mathbf{A} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 3 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$, AND $\mathbf{b} = \begin{bmatrix} 4 \\ 2 \\ 3 \\ 6 \end{bmatrix}$, WE FIRST

CALCULATE A BASIS OF \mathbf{A} AS $\mathbf{B} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 \end{bmatrix}$.

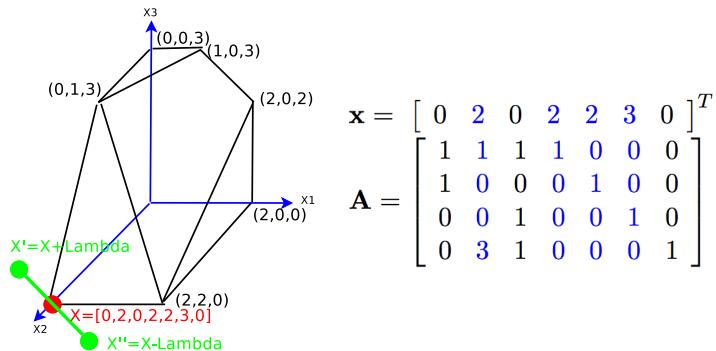
- THE **basic feasible solution x respect to B** IS $\mathbf{x} = \begin{bmatrix} \mathbf{B}^{-1}\mathbf{b} \\ \mathbf{0} \end{bmatrix} = \begin{bmatrix} 0 & 2 & 0 & 2 & 2 & 3 & 0 \end{bmatrix}^T$.
- IT IS EASY TO VERIFY THAT $(x_1, x_2, x_3) = (0, 2, 0)$ IS A VERTEX OF THE POLYTOPE \mathbf{P} .

第八章 线性规划 (2)

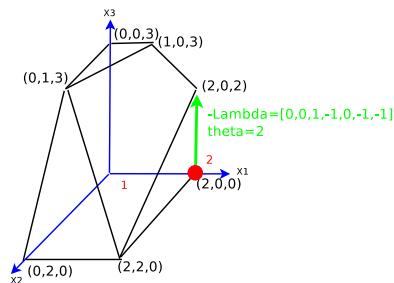
上堂课讲线性规划，通过线性规划学习一个新的套路，当我们遇到一个问题，解不能分的时候，或者我们不愿意分的时候，怎么办。这种情况下，只能从完整解开始，我们要构造这样一张图。所有的完整解都画在一张图里，每个节点是一个完整解，每 2 个点经过一些很少的变换就能转化，我们就连一条边。边两端的解靠的很近。举一个简单的例子，真实情况比较复杂。每个解标记为 x_i ，都有相应的值 $f(x_i)$ ，我们的目标是找 $\min f(x_i)$ ，怎么办？随便找一个初始解，看他的邻居是不是比他小，然后就走到邻居，再接着判断他的邻居是否能改进，这是非常典型的做法，叫 IMPROVEMENT。一切都是从完整解出发，所以上节课突出这种典型的路数，也就说当问题的解不好分解，但我们能观察出可行解之间的关系，这个图叫完全解邻域关系图。我们判断每个解的邻域的关系。所以一旦画出这样的图，很容易用这种套路求解。第一步，随便给一个初始点。第二部，循环判断当前点的邻域能否移动，比自己好，就 IMPROVE。最后，判断一下是不是结束，是不是足够的好。用到线性规划中去，就要多加一小问。第一点，可行解是谁？可行解是在整个区域，特别大最优解在某个顶点，内部不用管，只用管这些顶点，把范围大大减小。剩下的就是刚才讲过的问题。怎么样找初始点，怎样从一个解到另一个解，什么时候停，上次课前两个问题解答了，最优解肯定在顶点出现（如果在内部出现，肯定能找到顶点比他更好）。



找初始解，就是随便找一个顶点，怎么通过计算找一个顶点，任意一个顶点（0, 2, 0）带进去，变成一个完整解。都带进去之后，只找那些非零的解对应的列，也就是基。什么是顶点，顶点就是一个基。



开始看第三个问题，第一个问题只管顶点就可以了，第二个问题顶点怎么找，就是找 A 的基，找基很简单，高斯消元法可以解决，现在回答第三个问题。如果当前解不太好，要 IMPROVE，找和他相邻的一个解，找相邻的顶点，问题是怎样从一个顶点到另一个顶点呢，直观上看就是经过一条边，走合适的距离，怎么实现。



看一个例子就很清楚了，从 $(2, 0, 0)$ 开始，完整带入求出完整解，非零列挑出来，线性无关，构成基。其他向量可以用基表示，表示唯一，其他写成这样，系数拿出来，可以看出，系数恰好对应一条边。 $(0, 0, -1, 1, 0, 1, 1)$ ，在三维空间， $(0, 0, -1)$ ，向下方向，就是这条边，沿这条边走合适的距离，就是 $(2, 0, 0)$ ，走一段距离， x' 沿着 λ 走 θ 距离，设 $\theta = 2$ ，后面解释为什么，带进去看看。 $(2, 0, 2)$ ，恰好是这个点，也就是说，我们思路很清晰，一开始给定顶点，蓝色的对应一组基，这是第一步，第二步，其他非基向量可以表示成基的线性组合，系数恰好对应边的方向，我们只需要沿着边走合适的距离就会到达一个顶点。

严格的证明一下。 x 是一个顶点，顶点对应这组基，考察一个非基向量，非基向量 a_e 表示成基的组合，计算沿这个方向走多远，就是 θ ，先按式子计算，得到一 θ ，得到 x' 对应一个顶点，每个顶点都对应一个基，新的顶点对应的基就是 $B' = B - a_l + a_e$ 。

下面证明一下，证明 2 页 SLIDES。原先 x 是可行解， $ax = b$ 是满足的，拆成向量的形式，把绿色的 a_e 拆成基的线性组合，上面式子减去下面式子的 θ 倍，这个式子是满足的，新得到的解 $x'_i = x_i - \theta\lambda_i$ ，证明新的解是可行解，可行解满足 2 个条件，一个是 $ax = b$ ，一个是 $x \geq 0$ ，现在只需证明第二个条件，分两种情况讨论，一种是所有 $\lambda_i \leq 0$ ，所有的 $x_i \geq 0, \theta$ 取正数，就能满足条件。第二种情况是，存在 $\lambda_i > 0$ ，不能把 θ 取的太大，否则某些 $x_i < 0$ ，设置 $\theta = \min_{\mathbf{a}_i \in \mathbf{B}, \lambda_i > 0} \frac{x_i}{\lambda_i} = \frac{x_l}{\lambda_l}$ 推导如下：

$$x_i - \theta\lambda_i \geq 0$$

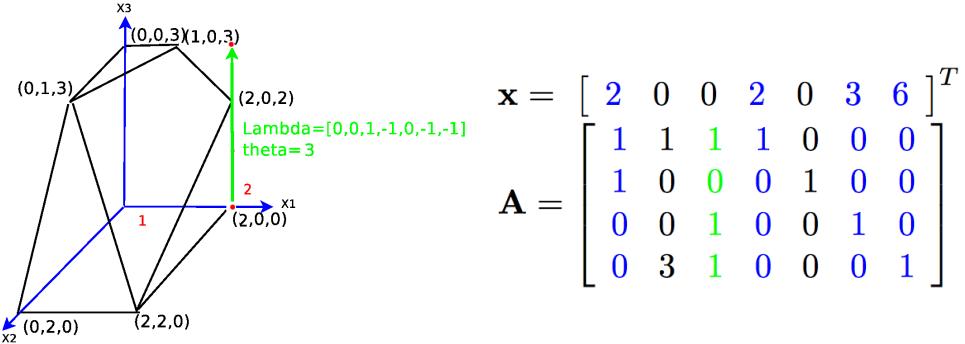
$$x_i \geq 0$$

$$\Rightarrow \theta \leq \frac{x_i}{\lambda_i}$$

θ 至多设置这么大，达到最大的向量给予下标 l ，这里面 E 代表 ENTER，L 代表 LEAVE。看一下例子，只用考虑 λ 中大于 0 的位置， θ 至多是多少可以算出来。在这里 $\theta = 2, l = 4$ 。这样取了之后， x' 对应一个可行解。满足上面 2 个条件。

再从具体的例子看一下， θ 的设置是从这个点出发，走多远恰好到一个顶点。刚才 $\theta = 2$ ，假如 $\theta = 3$ ，直观上沿这个点走 3 步，走出去，不在可行解的区域内。反应到式子里，具体就是 $x' = x - \theta\lambda$ ，刚才 θ 至多是 2，如果取 3 的话，导致 x' 存在分量小于 0。我们要求所有 x 的分量都是大于等于 0 的。所以不是可行

解。再看，如果沿着这个方向走，只走一步，直观上看没有到达这个顶点，在这



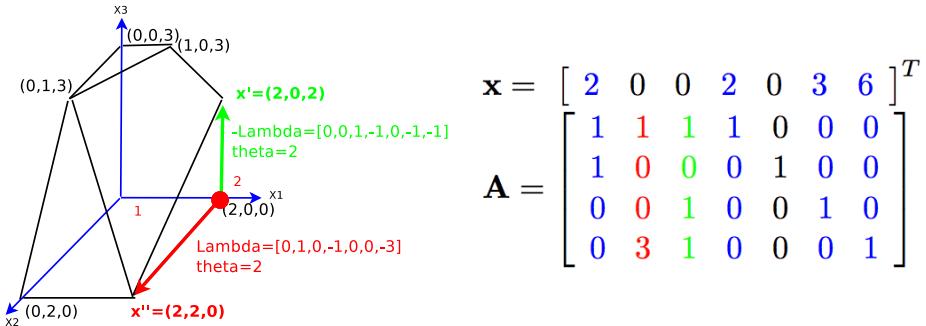
个边的中间， $\mathbf{x}' = \mathbf{x} - \theta\lambda$ 不是一个顶点，也可以从数值上面看，取出非零列，是线性无关的，构成一个基，现在他有 5 个，肯定不是一组基。现在证完了，从 \mathbf{x} 出发，沿一个方向走一定的步数，到达一个可行解。下面说明，这个可行解沿边走 2 步，必定到达一个顶点，对应一组基，怎么证明他是一个顶点呢？只需要证明 \mathbf{x}' 的非零列对应的是一组基，是线性无关的。换句话说，我们想证明：在这个例子中，新的一组基怎么得到的，绿色的 a_3 放进来， a_4 拿出去， a_1, a_3, a_6, a_7 对应一组基，怎么证明？假如说不是一组基，是线性相关的，存在一组不为 0 的数，使得 $d_1 a_1 + d_3 a_3 + d_6 a_6 + d_7 a_7 = 0$ ，注意 a_3 原先是非基向量，可以用原来的基唯一线性表示，带进去，得到新的等式，而原来的基是线性无关的，所以系数都为 0，因此，得到新的基也是线性无关的。

总结一下，一组非基向量表示成基向量的组合，系数就是方向，沿这个方向走多远呢，只要算一下 θ 就行，走固定长度之后必定是一组基，证明就是刚才上面的过程。

现在回顾一下，刚才讲的，随便给的初始解后，把基标出来，现在，沿着一条边到达一个顶点，得到新的基，原先的基去掉一个，新添加一个向量。看下面的例子，蓝色是旧的基，绿色是新添加的基，把这种操作叫做旋转 PIVOT。蓝色是出基的向量 E ，绿色的是入基的向量 L ，后面讲解 PIVOT 如何实现。关于 PIVOT 的由来，翻译有，旋转，QUICKSORT 中的枢纽元，很古怪的用法。不好翻译。小布什的 US PIVOT CHINA。关于南海的讲话，重返亚太。

刚才说，从任意一个顶点选择一条边，可以到达一个顶点，边对应非基向量，拆了之后， λ 对应边的方向，红色非基向量表示成蓝色的线性组合，系数对应边。从

(2,0,0) 出发，可以到达三个方向，(有一个方向没画)，沿着那个方向走？试试看？到达 x', x'' 怎么样。



先说我们走到 a_2 ，把他表示成基的线性组合，边的方向就是系数，设置合理的 θ ，得到 x'' ，目标函数的变化，考虑到目标函数值，我们的旋转就是为了改变目标函数值。我们不知道选择哪个点，所以看目标函数值变化。 x'' 目标函数值是 $c^T x''$, x 是 $c^T x$, 都是已知的得到 -14θ , 就是目标函数值的改变，从 x 到 x'' , 目标函数值大幅度降低了。再往后看，走另一条边，也是上面的方法，到达 x' ，计算目标函数的改变值，这里是 -6θ , 从 x MOVE x' ，目标函数的改变是 -6θ 。一般我们采用最大梯度的规则， -6θ 和 -14θ 所以沿着边到达 x'' 的方向走，这是启发式的规则，一般这么用，也可以不用。

现在只剩最后一个问题了，我们会从一个顶点到另外一个顶点，怎么过去呢？找非基向量就可以，什么时候停呢？假如说我们从一个顶点 x 到达 x' ，看目标函数改进， $c^T x' - c^T x$ ，求这个函数的差。

$$x' = x - \theta \lambda$$

$$c^T x' - c^T x = -\theta c^T \lambda$$

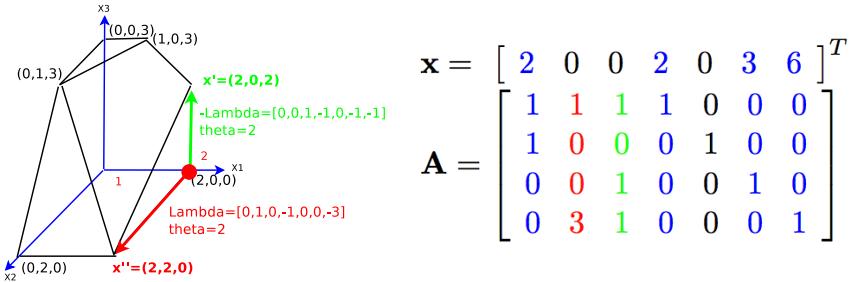
详细的写开， λ 到底是什么，入基的 c 减去其他向量，这是目标函数的改进，一旦这个数小于等于 0，以为这从 x 到 x' 后，我们得到改善，IMPROVE，我们想 MINIMIZE，每次都是下降，当得不到改善，就停。当上面的目标函数改进大于等于 0，我们就停。这个公式 λ 的形式是怎么推出来的。我们有好多列，蓝色的列是基，绿色的入

基 a_e , 出基 a_l , 有 $a_e = \lambda_1 a_1 + \lambda_2 a_2 + \dots + \lambda_m a_m$, $\vec{\lambda} = [\lambda_1, \lambda_2, \dots, -1, \dots, \lambda_m]$, 然后

$$x' = x - \theta \vec{\lambda}$$

$$\begin{aligned} c^T x' - c^T x \\ &= -\theta c^T \vec{\lambda} \\ &= -\theta [-ce + \sum_{i=1}^m \lambda_i c_i] \\ &\quad \theta [ce - \sum_{i=1}^m \lambda_i c_i] \end{aligned}$$

我们什么时候停呢? 如果这个数对任意的非基向量都大于等于 0, 意味着带进去目标函数没有改善, 我们就停了。所以, 我们把这个数叫做检验数。检验数写成矩阵的形式就是 $\bar{c}^T = c^T - c_B^T B^{-1} A$, λ 替换成 $B^{-1}A$, 后面解释为什么进行替换。现在先记住! 停止条件也清楚了, 任意非基向量, 把他替换成 λ , 带进去算出目标函数, 大于等于零就停止, 也就是最优了! 为什么呢? 我们严格的证明一下! 考察一个线性



规划, 他的目标函数是 $\min \mathbf{c}^T \mathbf{x}$, 约束是 $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$, 假如说 \mathbf{x} 是一个顶点, 顶点就对应一组基 B , 如果算出检验数大于等于 0, 则 \mathbf{x} 就是最优解。为什么是最优解, 假如说我们找到一个可行解 \mathbf{y} , 满足约束条件, 我们可以证明任意可行解的目标函数都比 \mathbf{x} 的目标函数大, 为什么呢? 因为 $c^T \geq c_B^T B^{-1} A, y \geq 0, B^{-1} b = x_B$, 任意顶点都对应一个基, 顶点和基的对应关系, 后面看例子理解, 换句话说, 任意可行解的目标函数值都比 \mathbf{x} 的要大, 所以是最优解。

有点绕, 大家回去后仔细看! 有了这四点之后, 单纯性算法就完全做完了。回顾一下! 我把它扩展成了五点, 又加了第一句琐碎的话! 对线性规划, 观察五点性质, 第一点: 可行解就是多面体内部, 包括边界, 最优解肯定实在定点上拿到, 所以我们不用关心内部, 只关心顶点, 这比较省事! 如何得到顶点呢? 只要做线性变换就

可以，算出基，基就是顶点！如果当前一个顶点不够好，想 IMPROVE，在找一个点，比我好，怎么 IMPROVE，怎么找到下一个点？沿着某一条边到下一个顶点，怎么沿着边走？边是什么？边就对应非基向量，拿什么时候停呢，一旦做到最后，一个检验数全大于零，就停。检验数是 $c^T - c^T B^{-1} A$ ，为什么是这样？稍等一下，看这五点全有了。有这五点，我们就能把单纯性算法写完。写完看一下主要步骤。输入是矩阵 A，向量 b，MINIMIZE $c^T x$ ，第一步调用初始化，找一个初始点，这个初始点，把 A 的基给找出来，叫做 B，下标都放在 B 里面，这件事稍微有点麻烦，先假设会做这一步，待会会说完整的扩展，其他的都是 EASY。得到一个初始点，按照过去的规范，IMPROVE 初始化，就是写一个死循环，里面先判断一下是不是要停，如果不存在一个非基向量， $c_e < 0$ 的话，检验数都大于等于 0，我们就返回，返回的时候返回一个值。如果存在一个小于 0，可以 IMPROVE，我们随便找一个向量，也就是绿色的那个向量，拆成其他向量线性组合，算一下那个 θ ， θ 等于所有的 $\frac{b_i}{a_{ie}}$ 的最小值，若果 $a_{ie} > 0$ ，否则 $\Delta_i = \infty$ ， θ 去最小的 Δ_i ，改成 θ_i ，如果 $\theta = \infty$ ，则是无界的，否则做一个 PIVOT，PIVOT 是说旧基，E 为入基，L 为出基，调用这个过程，这就是线性规划单纯性算法，里面有 3 步没解决，一个是怎么找初始点，放到最后再说，一个是找到一个基，我们怎么算 x，接下讲解，还有一个是 IMPROVE 的时候，一个入基，一个出基，怎么操作。

```

SIMPLEX(A, b, c)
1: ( $B_I, A, b, c, z$ ) = INITIALIZESIMPLEX(A, b, c);
2: //IF THE LP IS FEASIBLE, A VERTEX x IS RETURNED WITH  $B_I$  STORING THE INDICES OF VECTORS IN THE
   CORRESPONDING BASIS B; OTHERWISE, "INFEASIBLE" IS REPORTED.
3: while TRUE do
4:   if THERE IS NO INDEX e (1 ≤ e ≤ n) HAVING  $c_e < 0$  then
5:     x = CALCULATEX( $B_I, A, b, c$ );
6:     return (x, z);
7:   end if;
8:   CHOOSE AN INDEX e HAVING  $c_e < 0$  ACCORDING TO A CERTAIN RULE;
9:   for EACH INDEX i (1 ≤ i ≤ m) do
10:    if  $a_{ie} > 0$  then
11:       $\Delta_i = \frac{b_i}{a_{ie}}$ ;
12:    else
13:       $\Delta_i = \infty$ ;
14:    end if
15:  end for

```

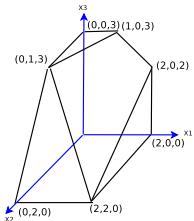
```

16:   CHOOSE AN INDEX  $l$  THAT MINIMIZES  $\Delta_i$ ;
17:   if  $\Delta_l = \infty$  then
18:     return "UNBOUNDED";
19:   end if
20:    $(B_I, \mathbf{A}, \mathbf{b}, \mathbf{c}, z) = \text{PIVOT}(B_I, \mathbf{A}, \mathbf{b}, \mathbf{c}, z, e, l);$ 
21: end while

```

先说这一个，假如说我们知道一个矩阵 A ，基是 B ， I 是 INDEX 下标，由基础确认的 x 是怎样的呢？是这样的？对每个 x_j 都检查一下，如果 x_j 不在基里面，我们说什么是基，什么是顶点？把 x 找出来，非零的那些拿出来对应一组基，换句话说，0 的那些都是非基的，所以不在基里面的那些 $x_j = 0$ ，其他那些，任意的一个向量的解就是 B 。这里写法有点绕，待会看一下例子就清楚了。这么写是没错的，更加鲁棒性一些，这里存疑，待会再回来看一下。我们讲了那么多，看个例子！假如说，我们要求这个线性规划，目标函数和约束如下，如何从这个约束把这个图画出来已经讲过了，最终是一个多面体。STANDARD FORM:

$$\begin{aligned}
\min \quad & -x_1 - 14x_2 - 6x_3 \\
s.t. \quad & x_1 + x_2 + x_3 \leq 4 \\
& x_1 \leq 2 \\
& x_3 \leq 3 \\
& 3x_2 + x_3 \leq 6 \\
& x_1, x_2, x_3 \geq 0
\end{aligned}$$



我们先把他加一些松弛变量，变成等式约束。一旦做完松弛变量，我们就画了如下的一些 TABLE，单纯型算法，他讲线性规划，也是纸上作业，画一个表格，这个表格叫单纯型表，表格主体就是矩阵 A , c 写在上面，对一下上面的例子， B 写在左边，有的写在右边，写在左边比较方便，左上角是目标函数的负数，一开始是 0。

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$-z = 0$	$\bar{c}_1 = -1$	$\bar{c}_2 = -14$	$\bar{c}_3 = -6$	$\bar{c}_4 = 0$	$\bar{c}_5 = 0$	$\bar{c}_6 = 0$	$\bar{c}_7 = 0$
$\mathbf{x}_{B1} = b'_1 = 4$	1	1	1	1	0	0	0
$\mathbf{x}_{B2} = b'_2 = 2$	1	0	0	0	1	0	0
$\mathbf{x}_{B3} = b'_3 = 3$	0	0	1	0	0	1	0
$\mathbf{x}_{B4} = b'_4 = 6$	0	3	1	0	0	0	1

单纯型表的设计，我们都很清楚，A, B, C, -Z 总共 4 块，事实上，我们要看的仔细一些，矩阵 A 始终要维护一个基 B 都是单位阵，始终要保持这一点，只要作高斯线性变换就可以，关键是这样保持之后，能推出很多很好的性质，很好的性质如下：这个矩阵 A 做高斯线性变换，基是 B，实际上是 $B^{-1}A$ ，矩阵 A，基 B，保持单位阵，整个矩阵乘以 B^{-1} ，就变成单位阵，这块就是 $B^{-1}A$ ，下一页讲的更清楚。

为什么单纯型表这样设计？为什么基对应的矩阵是单位阵，原始的矩阵 A 放在这，基的叫做 B，非基的叫做 N，对应的左边是向量 b，上面是 c，c 也分块，一个叫 c_B ，一个叫 c_N ，这个很简单！假如说 B 刚开始不是单位阵的形式，这样不太好！我们先把他变成单位阵，整个矩阵乘以 B^{-1} 就行了，所有向量全部乘 B^{-1} 就行。然后变成 $B^{-1}B$ 单位阵， $B^{-1}N$ ，上面也参与行变换， $c_B = 0$ ，仔细看一下右上角变成什么， c_i 要变成 0，要用下面的单位阵乘以 c_i ，再减去上面的就变成 0，依次累加，右上角就是 $c_N^T - c_B^T B^{-1}N$ ，左上角就是 $-c_B^T B^{-1}b$ 。矩阵 A，得到基是 B，要转换为单位阵，好处很多，变成单位阵就是所有乘以 B^{-1} ，如上变换，怎么达到这一点，就是高斯行变换，线性代数里面有讲解。上面的 c 也加入高斯行变换，每列分别减去下面单位阵的 $c_i, 1 \leq i \leq m$ 倍，就变为 0，就是乘以 c_B^T ，然后减去。左边也要进行高斯行变换，注意左上角跟右上角的对应关系。这就是为什么在单纯型算法里面要千方百计的保证对应基为单位阵，单位阵带来好处，B, C, z 是固定的。 $B^{-1}b$ 对应顶点， $c_b^T B^{-1}b$ 对应目标函数值， $c_N^T - c_B^T B^{-1}N$ 对应检验数。分离的形式好理解，矩阵的形式需要仔细思考，矩阵的乘法怎么理解！最后说一件事，就是 PIVOT。PIVOT 是说一开始得到一个顶点，把他对应的基都画起来了，非基向量进行拆，让他入基。PIVOT 直观上看就是，基已经写成单位阵的形式，上面的对应的 c 都是 0，非基的向量不一定是单位阵，最终想让他入基，把他变成单位阵，单位向

$$\begin{array}{c|ccc|cc}
 & \mathbf{c}_B^T & & \mathbf{c}_N^T & & \\
 \hline
 0 & c_1 & c_2 & \cdots & c_m & \cdots & c_n \\
 b_1 & a_{11} & a_{12} & \cdots & a_{1m} & \cdots & a_{1n} \\
 b_2 & a_{21} & a_{22} & \cdots & a_{2m} & \cdots & a_{2n} \\
 \vdots & \vdots & \ddots & \ddots & \vdots & \ddots & \vdots \\
 b_m & a_{m1} & a_{m2} & \cdots & a_{mm} & \cdots & a_{mn} \\
 \hline
 \mathbf{b} & \mathbf{B} & & \mathbf{N} & &
 \end{array} \xrightarrow{\text{Row Operation}} \begin{array}{c|cc|c}
 -\mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{b} & 0 & 0 & \cdots & 0 \\
 \hline
 \mathbf{B}^{-1} \mathbf{b} & 1 & 0 & \cdots & 0 \\
 & 0 & 1 & \cdots & 0 \\
 & \vdots & \vdots & \ddots & \vdots \\
 & 0 & 0 & \cdots & 1
 \end{array} \xrightarrow{\mathbf{B}^{-1} \times} \begin{array}{c|cc|c}
 \bar{\mathbf{c}}^T & & & \\
 \hline
 \mathbf{c}_N^T - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{N} & & & \\
 \hline
 \mathbf{B}^{-1} \mathbf{B} & & & \\
 \mathbf{B}^{-1} \mathbf{N} & & &
 \end{array}$$

$$\begin{array}{c|ccccc}
 0 & \dots & 0 & \cdots & \mathbf{c}_e & \cdots \\
 \hline
 b_1 & \dots & 0 & \cdots & a_{1e} & \cdots \\
 b_2 & \dots & 0 & \cdots & a_{2e} & \cdots \\
 \dots & \dots & \dots & \dots & \dots & \dots \\
 b_l & \dots & 1 & \cdots & a_{le} & \cdots \\
 \dots & \dots & \dots & \dots & \dots & \dots \\
 b_m & \dots & 0 & \cdots & a_{me} & \cdots
 \end{array} \Rightarrow \begin{array}{c|ccccc}
 -\frac{a_{me}}{a_{le}} b_l & \dots & -\frac{\mathbf{c}_e}{a_{le}} & \cdots & 0 & \cdots \\
 b_1 - \frac{a_{1e}}{a_{le}} b_l & \dots & -\frac{a_{1e}}{a_{le}} & \cdots & 0 & \cdots \\
 b_2 - \frac{a_{2e}}{a_{le}} b_l & \dots & -\frac{a_{2e}}{a_{le}} & \cdots & 0 & \cdots \\
 \dots & \dots & \dots & \dots & \dots & \dots \\
 \frac{1}{a_{le}} b_l & \dots & \frac{1}{a_{le}} & \cdots & 1 & \cdots \\
 \dots & \dots & \dots & \dots & \dots & \dots \\
 b_m - \frac{a_{me}}{a_{le}} b_l & \dots & -\frac{a_{me}}{a_{le}} & \cdots & 0 & \cdots
 \end{array}$$

量，怎么做，就是高斯行变换，看一下伪代码的意思，选定入基和出基，入基变成单位向量，然后高斯行变换，左边的 B 也要变，写的比较绕，实际上很简单，仔细理解一下。

$\text{PIVOT}(B_I, \mathbf{A}, \mathbf{b}, \mathbf{c}, z, e, l)$

```

1: //SCALING THE  $l$ -TH LINE
2:  $b_l = \frac{b_l}{a_{le}}$ ;
3: for  $j = 1$  TO  $n$  do
4:    $a_{lj} = \frac{a_{lj}}{a_{le}}$ ;
5: end for
6: //ALL OTHER LINES MINUS THE  $l$ -TH LINE
7: for  $i = 1$  TO  $m$  BUT  $i \neq l$  do
8:    $b_i = b_i - a_{ie} \times b_l$ ;
9:   for  $j = 1$  TO  $n$  do
10:     $a_{ij} = a_{ij} - a_{ie} \times a_{lj}$ ;
11:   end for
12: end for
13: //THE FIRST LINE MINUSES THE  $l$ -TH LINE
14:  $z = z - b_l \times c_e$ ;
15: for  $j = 1$  TO  $n$  do
16:    $c_j = c_j - c_e \times a_{lj}$ ;
17: end for
18: //CALCULATING  $\mathbf{x}$ 

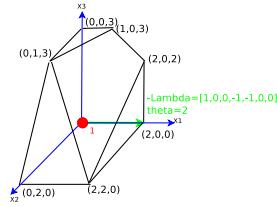
```

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$-z = 0$	$\bar{c}_1 = -1$	$\bar{c}_2 = -14$	$\bar{c}_3 = -6$	$\bar{c}_4 = 0$	$\bar{c}_5 = 0$	$\bar{c}_6 = 0$	$\bar{c}_7 = 0$
$\mathbf{x}_{B1} = b'_1 = 4$	1	1	1	1	0	0	0
$\mathbf{x}_{B2} = b'_2 = 2$	1	0	0	0	1	0	0
$\mathbf{x}_{B3} = b'_3 = 3$	0	0	1	0	0	1	0
$\mathbf{x}_{B4} = b'_4 = 6$	0	3	1	0	0	0	1

19: $B_I = B_I - \{l\} \cup \{e\}$;

20: **return** $(B_I, \mathbf{A}, \mathbf{b}, \mathbf{c}, z)$;

看个例子，这样大家的困惑得会解决一些！



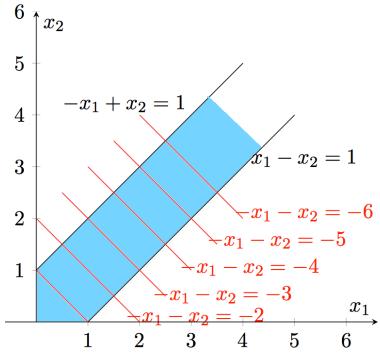
三步骤，第一步找一个初始点，第二步找谁进去，第三部进去之后再变一次，第四部就是什么时候停。公式比较多，看的麻烦！就解前面那个线性规划的例子！画出单纯型表，写上 A , b , c , z , 没什么问题。第一步找一个初始点，初始点就是基，蓝色的就是基，而且就是单位阵，基对应的顶点写成矩阵的形式就是 $\mathbf{x} = \begin{bmatrix} \mathbf{B}^{-1}\mathbf{b} \\ \mathbf{0} \end{bmatrix} = [0, 0, 0, 4, 2, 3, 6]^T$, 非基的都是 0, 基对应的就是 B , 可以写出 \mathbf{x} 。找到基了，基就是顶点，这点不太好，然后进行改进，随便找个非基向量，绿色的，让他入基，沿着他走多远呢？随便找一列，只要上面对应的 c 小于 0, 就可以。为什么小于 0, 刚才说了，先不管他。这就是入基，然后是出基，这一列里面找大于 0 的，除法计算最小的 θ 值，对应的行的单位阵也就是旧基里面为 1 的那一列为出基，单纯型算法 3 步，第一步：初始点：基，第二部：WHILE{T} 找一个负的 c_i , 用 $\frac{b_i}{coef}$, 找最小的，然后行变换。如果找不到负的就结束。在这个例子里面接下来就是做高斯行变换，新的基就产生了，跟原来不一样。对应图里面就是，原始的点事 x_1 , 走到右边那个点。接着再找 c 里面谁是负的，这一列里面找正的，然后做高斯行变换，写

程序比较简单。这里面插一句话：这里 B 是 0，比较奇怪，导致 $\theta = 0$ ，这很特殊，也就是说当前点沿着找的方向走 0 步，还是自己这个点。这是什么意思？这可能带来危险！可能一直在一个点呆着，这种情况叫做退化！接下来往下走，最后所有的 C 都是大于等于 0 的，停了，当前点是最优点。最优点的值是什么？基对应相应的 B ，非基都是 0，所以写程序比较简单！但是，为什么能方便的写程序实现单纯型算法呢，就是因为单纯型表有那些优秀的性质。始终把基表示成单位阵， B 对应解，上面的 C 对应检验数，左上角是目标函数值，那一页 SLIDES 就是最核心的。

在单纯型算法过程当中，我们有可能在一个点不断的呆着。因为 $\theta = 0$ ，怎么办？有很多办法处理这个事情，处理的规则暂时不讲，大家感兴趣可以看一下附加的 SLIDES。先往下讲。对单纯型算法强调几点：第一点，写程序是很 EASY 的，只是背后的知识，为什么要这么实现，讲课的难点在这里，为什么矩阵是这样的。第二个比较难的地方还是在矩阵，很多的术语写成分离的形式很好理解，但写成矩阵的形式，不好理解。熟悉一下矩阵乘法表达的意思。剩下的困难就不多了。这次课后有个作业实现单纯型算法，第一因为单纯型算法非常经典，第二这个套路在其他地方很有用。第三点，学生实现过后，我现在很多研究，在基因组上的线性规划，基因组上有上千万个点 x_i ，调用所有的软件包都不行。因为内存空间占用太大，但是结构很奇怪！这里说明为什么让大家练的原因？因为起码对我的工作是有用的，对大家将来有可能有用。基因组是一个长长的线段，上面有一些点 x_i ，点的数目非常多，10M，上千万左右，知道有些点之间的距离 d_{ij} 。把整体的 x_i 求出来。线性规划如下：

$$\begin{aligned} \min \quad & E \\ \text{s.t.} \quad & x_i - x_j = d_{ij} + \varepsilon_{ij} \\ & |\varepsilon_{ij}| \leq E \end{aligned}$$

ε_{ij} 是误差， d_{ij} 已知，使误差最小化。最简单的，还没用二次平方误差！数据量比较大！按道理来说，应该是 MSE 准则！现在线性的都搞不定，不能用平方！这个线性规划，上千万的时候，现有的包都搞不定！GOURBI,GLPK，都不行！要用特定的算法解决！这个样子比较奇怪，这样单纯型表的矩阵 A 是极度稀疏的，而且维数很大！内存根本放不下。我们要重新编写。下面来看一个例子！这个例子比过去的要直观



一些！是下面这样子的

$$\begin{aligned}
 & \min -x_1 - x_2 \\
 \text{s.t.} \quad & x_1 - x_2 \leq 1 \\
 & -x_1 + x_2 \leq 1 \\
 & x_1, x_2 \geq 0
 \end{aligned}$$

在图上看一下约束是怎么样的，具体的对应关系！这个解区域有点问题，不是封闭的！把目标函数的等高线画出来。可以看到目标函数可以取到负无穷，这是在图上直观的看法，那运行单纯型算法会怎么样呢？先把他写成松弛型，添加松弛变量，变成等式约束。

STANDARD FORM:

$$\begin{aligned}
 & \min -x_1 - x_2 \\
 \text{s.t.} \quad & x_1 - x_2 \leq 1 \\
 & -x_1 + x_2 \leq 1 \\
 & x_1, x_2 \geq 0
 \end{aligned}$$

SLACK FORM:

$$\begin{aligned}
 & \min -x_1 - x_2 \\
 \text{s.t.} \quad & x_1 - x_2 + x_3 = 1 \\
 & -x_1 + x_2 + x_4 = 1 \\
 & x_1, x_2, x_3, x_4 \geq 0
 \end{aligned}$$

画出单纯型表，写出矩阵的形式，照抄，很简单！第一步先找基，这很简单，松弛变量天然的就是一组基。由基可以反解出 x ，然后开始 WHILE 循环，先找 C 里面小于 0 的列，对应 A 里面的正数，求出 θ 的最小值！负的不管。做高斯行变换，入基和出基。得到新的基。再重复，找 C 里面负的！找到了，然后 A 里面对应的没正

的，原来找正的，是因为走太远会出圈，现在全是负的。

$$\begin{aligned} x &= [1, 0, 0, 0] \\ \vec{\lambda} &= [-1, 0, 0, 0] \\ x' &= x - \theta \vec{\lambda} \geq 0 \end{aligned}$$

直观上看，可也沿着这条边走无穷远，都是可行解！每走一步，目标函数都会变小，所以无界，想多小就多小！特征就是 A 里面对应的列全是小于等于 0 的。到现在为止，单纯型算法都会跑了，单纯型算法里面 3 块，现在解决 2 块，算法里面：找初始点，死循环，死循环里面都找有没有小于等于 0 的，若有，则沿着这个方向，找到最小的 θ ，做高斯行变换就完了，现在还剩初始点怎么找？初始时给我们矩阵 A，向量 b 和 c，怎样找一个基，使得对应基的 b 是大于等于 0 的。最后一个问题，我们会 IMPROVE，但是初始顶点怎么找？换句话说，让我们解 $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$ ，怎么解？用以前的知识能不能解？只会 $\mathbf{Ax} = \mathbf{b}$ ，但不一定满足要求！找这个点怎么做？不用线性规划是没法做的！过去学的那一套做不了。让我们看怎么做！假如说解下面的问题，找一个初始解。怎么办？我们把转换成解下面的辅助线性规划！辅助线性规划是这样子的，添加变量 x_0 ，MINIMIZE x_0 ，注意这里变换后还是小于等于 0 的，再陈述一次，求解 $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$ ，不会做，变成解辅助线性规划，加了一个目标函数，每个约束都减了个 x_0 。为什么是这样子的？假如说原先的线性规划有可行解，现在加上 x_0 ，其他值不变，这就是辅助线性规划的一个可行解。原来解满足约束，加上 $x_0 = 0$ 后也满足新的辅助线性规划的约束。搞到一个可行解，就是最优解。假如说原来的 L 没有可行解，对于任意取值，至少有一个约束不可满足，假如是第 1 行约束，那么根据辅助方程组，就有 $x_0 > 0$ 。MINIMIZE x_0 ， x_0 取不到 0。辅助线性规划有解，但是解不是 $x_0 = 0$ 。大家回去琢磨一下，这是非常巧妙的构造。上面有可行解，下面最优解肯定是 0。那怎么解辅助线性规划呢？很简单，先加上松弛变量，变成松弛型。因为松弛变量天然对应基。

$$\begin{array}{lllllll} \min & & & x_0 & & & \\ \text{s.t.} & a_{11}x_1 + \dots + a_{1n}x_n - x_0 + s_1 & & & & = b_1 \\ & a_{21}x_1 + \dots + a_{2n}x_n - x_0 + s_2 & & & & = b_2 \\ & & \dots & & & & \\ & a_{m1}x_1 + \dots + a_{mn}x_n - x_0 + s_m & & & & = b_m \\ & x_1, \dots, x_n, x_0, s_1, s_2, \dots, s_m & & & & \geq 0 \end{array}$$

之后，如果所有的 b_i 都大于等于 0，我们直接取他们作为基。其他的非基都是 0，这就是可行解。如果有负的，比较麻烦，不能直接作为基。作为基之后，不是一个可行解。怎么办呢？一旦 b_i 有负的之后，找 b_i 里面的最小值，负的最厉害的，其他所有的全减去负的最小的数，其他都是正的，自己再乘负号就行。

具体的过程不用管，回头看看就行，先看后面的例子，就清楚了！

```

1: LET  $l$  BE THE INDEX OF THE MINIMUM  $b_i$ ;
2: SET  $B_I$  TO INCLUDE THE INDICES OF SLACK VARIABLES;
3: if  $b_l \geq 0$  then
4:   return  $(B_I, \mathbf{A}, \mathbf{b}, \mathbf{c}, 0)$ ;
5: end if
6: CONSTRUCT  $L_{aux}$  BY ADDING  $-x_0$  TO EACH CONSTRAINT, AND USING  $x_0$  AS THE OBJECTIVE FUNCTION;
7: LET  $(\mathbf{A}, \mathbf{b}, \mathbf{c})$  BE THE RESULTING SLACK FORM FOR  $L_{aux}$ ;
8: //PERFORM ONE STEP OF PIVOT TO MAKE ALL  $b_i$  POSITIVE; ;
9:  $(B_I, \mathbf{A}, \mathbf{b}, \mathbf{c}, z) = \text{PIVOT}(B_I, \mathbf{A}, \mathbf{b}, \mathbf{c}, z, l, 0)$ ;
10: ITERATE THE WHILE LOOP OF SIMPLEX ALGORITHM UNTIL AN OPTIMAL SOLUTION TO  $L_{aux}$  IS FOUND;
11: if THE OPTIMAL OBJECTIVE VALUE TO  $L_{aux}$  IS 0 then
12:   RETURN THE FINAL SLACK FORM WITH  $x_0$  REMOVED, AND THE ORIGINAL OBJECTIVE FUNCTION OF  $L$ 
      RESTORED;
13: else
14:   return "INFEASIBLE";
15: end if

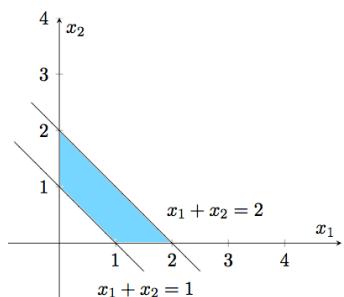
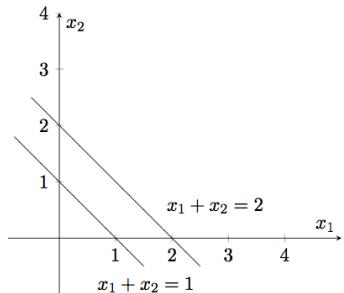
```

给我们解下面的线性规划，首先要求一个可行解！先画出可行域是什么样子的。既要在上面，又要在下面，所以直观上是没有可行解的。

LP L :

$$\begin{aligned}
 \min \quad & x_1 + 2x_2 \\
 \text{s.t.} \quad & x_1 + x_2 \geq 2 \\
 & x_1 + x_2 \leq 1 \\
 & x_1, x_2 \geq 0
 \end{aligned}$$

给我们线性规划，先求辅助线性规划，形式如前面讲解的那样，添加 x_0 ，然后把辅助线性规划转换成松弛型，这都使基本步骤。接着，写出单纯型表，矩阵照抄，松弛变量是基，但是对应的 $b_i < 0$ ，不是可行解，怎么办呢？所有的行都减去这个负的，然后自己再乘一个负号。然后就得到新基。且这个基对应的 $b_i > 0$ 是可行解。所有约束都有 $-x_0$ ，减去之后，除了自己有 $-x_0$ 外，其他都没有。然后自己变成 $+x_0$ 。大家体会一下基的变化！找到新基，且对应的 \mathbf{B} 是可行解。再往下面跑一



步，找一下 c 有没有负的，然后取一列，找出 θ ，做高斯行变换。最后停了！上面的 c 都是大于等于 0 的，STOP，得到最优解！最优解的值是 $-\frac{1}{2}$ ，我们的目标函数值是 0 的话，意味着原始的有可行解！现在不是 0，是负的，原先没有可行解！到现在为止，大家终于会本科学的东西有扩展了。 $Ax = b$ 会做，加上限制 $x \geq 0$ 就不会了，现在怎么做？变成辅助线性规划做！

$$\begin{aligned} & \min x_0 \\ \text{s.t. } & Ax + x_0 = b \\ & x \geq 0 \\ & x_0 \geq 0 \end{aligned}$$

所以很多时候，我们稍微加一点限制，过去的方法就不能做，必须要有更高级的工具才能解！ 我们再来看，刚才没有可行解，再看个有可行解的例子！

LP L:

$$\begin{aligned} & \min x_1 + 2x_2 \\ \text{s.t. } & x_1 + x_2 \leq 2 \\ & x_1 + x_2 \geq 1 \\ & x_1, x_2 \geq 0 \end{aligned}$$

稍微改一下，是这样子的！变成下面的线性规划！跟图进行对照，直观上看，是有可行解的！求解一下过程！也是刚才的步骤，构造辅助线性规划，转化成松弛型，非常 EASY。转换之后，写出单纯型表，松弛变量是基，但是基对应的 B 不是可行解，做初始化，减去负的最小的，然后自己乘以负号，得到新基和可行解。不断的 WHILE 循环，最后一步！所有的 c 都是正的，就返回了！得到最优的可行解！对应的目标函数值是 0，意味着辅助线性方程组的解达到最优解 0. 所以原始的方程是有可行解的。直接带进去就可以了。下一页是如何把初始可行解转化为转化为输入，可以暂时跳过，实现的时候比较方便，可以完全不管他。

我们讲完单纯型算法，大家回去可以用熟悉的语言实现。PYTHON 语言不到 200 行，就能实现！我们会写单纯型算法，实际情况，他会跑多快呢？

这一页，大家看一下介绍，单纯型算法 1949 年 DANTZIG 提出来的，很多人都在用，发现很快！做高斯行变换，基要换入和换出，要换多少次呢？实践当中发现的次数，基本上跟 M 成比例。和 N 的关系很小。有人说，固定 M 之后，运行次数跟 $\log n$ 成比例。这都是实践的经验！迭代的次数跟 M 有关，WHILE 循环的次数，随意总体的运行时间是 $O(m^2n)$ ，因为 WHILE 循环每一次都要做高斯行变换，矩阵的规模是 mn 的， $O(m)$ 轮，所以是 $O(m^2n)$ 。对稀疏矩阵来说，就像刚才的基因序列例子，运行时间 $O(Km^\alpha nd^{0.33})$ ， K 是常数， M 是行数， N 是线性的， D 是有多少个非零元。看稀疏程度怎么样。来实际看一下，我写了个程序，随机的产生很多线性规划，跑 GLPK，跑单纯型算法，看总共要几轮结束，首先固定约束 M ，变量数不断增长，观察有多少次！随着变量增加，换入换出增加非常缓慢！的确很像 $\log n$ 。又换另外一个，假如我们固定 N ，约束不断的增加。每个都随机的产生了一堆！每个至少产生 1000 个。每个点都是均值！看到迭代的次数跟 M 显著增加，很像线性的 $O(m)$ ，大概是这样的。

但是，非常不幸的是，单纯型算法在实际生活中跑的非常快，基本上是 $O(m^2n)$ ，但是不是多项式时间算法。这件事，直到 1972 年，V. KLEE AND G. L. MINTY 构造了一个稀奇古怪的例子，在他上面跑单纯型算法，就要花指数的时间。看一下例子是怎么样子的。

具体的例子如下，有这些数学表达式，画出多面体。如下图。这个样子比较难画！这个稀奇古怪的多面体会导致什么情况呢？他总共有 8 个点，跑单纯型算法，

所有的顶点都会遍历一次。看附加的 SLIDES 演示。各个棱是不一样长的。每一次 MOVE 都是下降的。总共 8 个顶点，都要旅行到，才可以。这是讲单纯型算法难点最简单的一个例子。实际的跑一下例子。的确要跑 2^n 的。看一下 GLPK 是不是有进行优化。根据 V. KLEE AND G. L. MINTY 的例子进行构造，产生最多 100 万个，看一下运行时间。是指数上升的。SCRIP 也放在网上，可以自己运行！沿着这条线，是支书时间，只是存在这种可能性，GLPK 大概按照这条线走的。花了指数时间，我们怎么不让他犯错呢，加一点 ERROR，马上时间就降下来。为什么单纯型算法在实际的例子中跑的很快，只要 $O(m^2n)$ ，但是的确存在一个例子，这个例子需要跑指数多的时间。这中间就有矛盾！怎么解决？有一个新的解决办法，叫平滑分析，平滑复杂度。平滑分析是 DANIEL A. SPIELMAN, SHANG-HUA TENG 在 2001 年得出的分析方法。找准自己感兴趣的方向！2001 年得到哥德尔奖。他们说单纯型算法是指数时间复杂度，但是他有多项式时间的平滑复杂度。换了一个度量。不用最坏时间，用平滑的，他就是多项式时间。那什么是平滑时间复杂度？讲个例子！到现在讲了 3 个分析方法，一个是平均时间分析，一个叫最坏时间分析，还有一个均摊分析。这 3 件事。平均时间分析解决最坏时间分析的困难，但是平均时间分析有问题。平滑时间复杂度是平均时间分析和最坏时间分析的一个折中。看一下画的图。X 轴是 INSTANCE 例子，每个点事固定 N，Y 轴是时间。最坏时间复杂度是最慢的例子，平均下来很低，AVERAGE 范围太广，拿不到所有的例子，不太可能，实际中做不动。怎么办？平滑时间复杂度是说稍微做点扰动，求平均，是很小的。在小邻域内做平均。复杂度很低。这是我们的解释，V. KLEE AND G. L. MINTY 构造的例子非常古怪，我们日常生活中拿到的例子，都有 ERROR，也就说现实生活中很难遇到 V. KLEE AND G. L. MINTY 的例子，总是带点噪声的例子。看一下带噪声的例子！看一下有噪声会出什么事！构造一个例子。解 $Ax = b$ ，A 加上一个 NOISE，用条件数解释，都一样，无法解释，很生动！大家做图像处理或者信号处理，总是很讨厌噪声，总是要想办法把噪声去掉，但是，滕老师说：“噪声是个好东西！”。

第九章 对偶

9.1 内容引入

任何一个线性规划或者写一个线性规划其实你得到的不是一个东西，而是两个东西，这另外一个东西就是他的对偶。

9.2 对偶的用处

我们之所以要讲对偶，对偶的作用如下：

1. 第一点用处是再往后了，等到我们讲近似算法的时候，非常重要，我们经常碰到一些优化一个目标函数，比如说最小化 $f(\mathbf{x})$ ，但是最小化 $f(\mathbf{x})$ ，有时候可能是 NP 完全的，是非常难的，所以我们只好求近似，不求最优求个差不多，求近似的时候如果我们一开始就知道 $f(\mathbf{x})$ 的界，就非常有帮助，那怎么得到这种界呢，这种界还有一个用处就是在分支限界里面，千方百计得到一个界，对偶和松弛是非常有力的一种形式的界，所以我们要将对偶。
2. 第二点用处是线性规划一般是成对出现的，我们写了一个 LP 之后，马上就可以写出对偶形式出来，所以我们就得到两个。那写了这个对偶之后怎么得到这个界呢？同时得到两个问题，对于其中一个问题的可行解就天然的是另外一个问题的一个界。

9.3 DIET 问题

9.3.1 原始问题

我们还是先从上面这最简单的例子看起，从家庭主妇买食品的例子入手，我们先回顾一下家庭主妇买食品，家庭主妇呢在市场上有这四种食品，每种食品他的成分都清楚，价格也知道，家庭主妇呢又想知道这一天呢我每种食品各买多少，使我花的钱最少，同时满足我能量的基本需求，我们已经讲过了，碰见这种问题我们怎么构建动态规划，这里面 WILL? 对象联系起来，还得稍微重塑一下。

问题： 家庭主妇想知道她每天必须花费在食品上多少钱，以获得所有的能量（2000 大卡），蛋白质（55 克），钙（800 毫克）。

表格：

Food	Energy	Protein	Calcium	Price	Quantity
Oatmeal	110	4	2	3	x_1
Whole milk	160	8	285	9	x_2
Cherry pie	420	4	22	20	x_3
Pork beans	260	14	80	19	x_4

线性规划方程：

$$\begin{aligned}
 & \min \quad 3x_1 + 9x_2 + 20x_3 + 19x_4 && \text{MONEY} \\
 \text{s.t. } & 110x_1 + 160x_2 + 420x_3 + 260x_4 \geq 2000 && \text{ENERGY} \\
 & 4x_1 + 8x_2 + 4x_3 + 14x_4 \geq 55 && \text{PROTEIN} \\
 & 2x_1 + 285x_2 + 22x_3 + 80x_4 \geq 800 && \text{CALCIUM} \\
 & x_1, x_2, x_3, x_4 \geq 0
 \end{aligned}$$

我们的目标函数是什么呢？目标函数就是 $3x_1 + 9x_2 + 20x_3 + 19x_4$ 这是我们的总价格，我们总共要花多少钱，约束是什么呢？我们只看一个吧，能量约束，买 x_1 份

燕麦我们要得到这么多能量，所以得到能量为 $110x_1 + 160x_2 + 420x_3 + 260x_4$ ，最小需求呢是 2000 大卡，所以约束是这个意思。

9.3.2 对偶问题

这个市场上总是有两类商品，不是只有一类，不仅有卖原始的燕麦、牛奶这些商品，还有一些公司呢分解的成果，比如说他是卖蛋白粉，单卖蛋白粉，单卖能量棒，单卖钙片。

问题表格：

Food	Energy	Protein	Calcium	Price (cents)
Oatmeal	110	4	2	3
Whole milk	160	8	285	9
Cherry pie	420	4	22	20
Pork with beans	260	14	80	19
Price	y_1	y_2	y_3	

线性规划方程：

$$\begin{aligned}
 \max \quad & 2000y_1 + 55y_2 + 800y_3 && \text{MONEY} \\
 s.t. \quad & 110y_1 + 4y_2 + 2y_3 \leq 3 && \text{OATMEAL} \\
 & 160y_1 + 8y_2 + 285y_3 \leq 9 && \text{MILK} \\
 & 420y_1 + 4y_2 + 22y_3 \leq 20 && \text{PIE} \\
 & 260y_1 + 14y_2 + 80y_3 \leq 19 && \text{PORK\&BEANS} \\
 & y_1, y_2, y_3 \geq 0
 \end{aligned}$$

那假如说你是卖蛋白粉的商家，如果让你来定价格，你应该怎么来定？首先说这个价格，你的总体目标当然是想赚的钱越多越好，每个家庭主妇都要赚的越多钱越好，另外当然你这个价格有一些限制条件，如果你定的价格与原始商品的价格之间有竞争

力，家庭主妇就可能考虑购买你的商品，比如不吃燕麦了买一堆的蛋白粉，一堆的能量棒和钙片，混在一块效果和麦片一样。但是你价格定的过高呢，那家庭主妇还不如直接去买麦片，这是一个很简单的道理。

接着我们会考虑到如果你是这个商家，让你来定一个合理的价格，你怎么定呢？你只好写一个这样的线性规划，我们假设能量棒是 y_1 , 1 大卡的能量棒我们的定价是 y_1 , 1 克的蛋白我们定价 y_2 , 1 毫克的钙是 y_3 , $2000y_1 + 55y_2 + 800y_3$ 就是我在一个家庭主妇身上至少赚这么多钱，我的目标就是至少赚的钱，最大化它。下面第一个约束是说各种成分的价格总和一定小于燕麦的价格，不然家庭主妇就会直接购买燕麦。那这个商家要定这个价格，并且在市场上价格越高越好，那么他就必须解这个线性规划。

9.3.3 DIET 问题总结

从 DIET 这个例子中我们可以看出，假如将原先的家庭主妇的问题定为原问题，那这个商家定价的问题就定为对偶问题，这两个是非常紧密关联的，实际上原问题和对偶问题只不过是矩阵 A 的两种看法：家庭主妇是按照行看，商家问题是按照列看。家庭主妇问题按行看写成矩阵的样子就是： $\text{MIN } c^T x$ 。商家问题按列看携程矩阵的样子就是： $\text{MAX } y^T b$ 。基本上两个式子是完全对应的。

9.4 原始问题和对偶问题

9.4.1 线性规划矩阵

$$\begin{array}{cccc|c} & c_1 & c_2 & \dots & c_n \\[1ex] & a_{11} & a_{12} & \dots & a_{1n} & b_1 \\[1ex] & a_{21} & a_{22} & \dots & a_{2n} & b_2 \\[1ex] & & & \dots & & \\[1ex] & a_{m1} & a_{m2} & \dots & a_{mn} & b_m \end{array}$$

原始问题和对偶问题只是对于线性规划矩阵 A 两种不同的角度：

- 原始问题：从行看
- 对偶问题：从列看

9.4.2 原始问题

详细表示（按行看）：

$$\min \quad c_1 \mathbf{x}_1 + c_2 \mathbf{x}_2 + \dots + c_n \mathbf{x}_n$$

$$a_{11} \mathbf{x}_1 + a_{12} \mathbf{x}_2 + \dots + a_{1n} \mathbf{x}_n \geq b_1$$

$$a_{21} \mathbf{x}_1 + a_{22} \mathbf{x}_2 + \dots + a_{2n} \mathbf{x}_n \geq b_2$$

...

$$a_{m1} \mathbf{x}_1 + a_{m2} \mathbf{x}_2 + \dots + a_{mn} \mathbf{x}_n \geq b_m$$

$$x_i \geq 0 \text{ FOR EACH } i$$

矩阵表示：

$$\min \quad \mathbf{c}^T \mathbf{x}$$

$$s.t. \quad \mathbf{A} \mathbf{x} \geq \mathbf{b}$$

$$\mathbf{x} \geq \mathbf{0}$$

9.4.3 对偶问题

详细表示（按列看）：

$$\begin{array}{ccccc}
 c_1 & c_2 & \dots & c_n & \\
 \vee | & \vee | & & \vee | & \max \\
 y_1 a_{11} & y_1 a_{12} & \dots & y_1 a_{1n} & y_1 b_1 \\
 + & + & & + & + \\
 y_2 a_{21} & y_2 a_{22} & \dots & y_2 a_{2n} & y_2 b_2 \\
 + & + & & + & + \\
 \vdots & \vdots & \dots & \vdots & \vdots \\
 + & + & & + & + \\
 y_m a_{m1} & y_m a_{m2} & \dots & y_m a_{mn} & y_m b_m \\
 & & & y_j & \geq 0 \quad \text{FOR EACH } j
 \end{array}$$

矩阵表示：

$$\begin{aligned}
 \max \quad & \mathbf{y}^T \mathbf{b} \\
 s.t. \quad & \mathbf{y} \geq \mathbf{0} \\
 & \mathbf{y}^T \mathbf{A} \leq \mathbf{c}^T
 \end{aligned}$$

9.4.4 原始问题与对偶问题之间的转换

写出一个线性规划的时候，可以同时得到他的对偶。原问题为 MIN 时，那么对偶就是 MAX：

- 一旦原问题是 $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ ，那么对偶一定是 $\mathbf{y} \leq 0$ ；一旦原问题是 $\mathbf{x} \geq 0$ ，那么对偶问题就是 $\mathbf{y}^T \mathbf{A} \leq \mathbf{c}$ 。

Primal:	Dual:
$\min \quad \mathbf{c}^T \mathbf{x}$ $s.t. \quad \mathbf{A}\mathbf{x} \leq \mathbf{b}$ $\mathbf{x} \geq 0$	$\max \quad \mathbf{y}^T \mathbf{b}$ $s.t. \quad \mathbf{y} \leq 0$ $\mathbf{y}^T \mathbf{A} \leq \mathbf{c}^T$
<i>change sign</i>	

- 如果原始问题是 $\mathbf{A}\mathbf{x} = \mathbf{b}$, 那么对偶问题对 \mathbf{y} 没有约束, 你可以 $\mathbf{y} \leq 0$ 也可以 $\mathbf{y} \geq 0$; $\mathbf{x} \geq 0$ 一样需要反号。

Primal:	Dual:
$\min \quad \mathbf{c}^T \mathbf{x}$	$\max \quad \mathbf{y}^T \mathbf{b}$
$s.t. \quad \mathbf{A}\mathbf{x} = \mathbf{b}$	$s.t. \quad \mathbf{y}^T \mathbf{A} \leq \mathbf{c}^T$
$\mathbf{x} \geq 0$	

Change Sign

9.5 拉格朗日乘子和 KKT 条件

我们写出一个线性规划, 那么我们就可以机械的写出他的对偶。对偶为何要这样生成, 我们可以很容易的从拉格朗日对偶的观点中得出。

9.5.1 拉格朗日乘子

引入

拉格朗日是: 求 $\min f(\mathbf{x})$, 我们可以通过倒数等于零, 如果给我们约束 $\min f(\mathbf{x})$, 一定要满足等式 $g_i(\mathbf{x}) = 0$ 的条件, 再增加难度, 再加入不等式条件 $h_i(\mathbf{x}) \leq 0$ 。

即以下三步:

1. 首先没有约束条件。

$$\min f(\mathbf{x})$$

2. 加入等式条件 $g_i(\mathbf{x}) = 0$ 。

$$\begin{aligned} \min & \quad f(\mathbf{x}) \\ s.t. & \quad g_i(\mathbf{x}) = 0 \quad i = 1, 2, \dots, m \end{aligned}$$

3. 再加入不等式条件 $h_i(\mathbf{x}) \leq 0$ 。

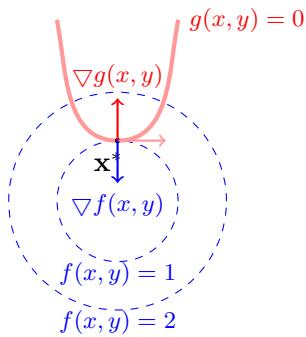
$$\begin{aligned} \min & \quad f(\mathbf{x}) \\ s.t. & \quad g_i(\mathbf{x}) = 0 \quad i = 1, 2, \dots, m \\ & \quad h_i(\mathbf{x}) \leq 0 \quad i = 1, 2, \dots, p \end{aligned}$$

求解

我们考虑以下问题:

$$\begin{aligned} \min \quad & f(x, y) \\ \text{s.t.} \quad & g(x, y) = 0 \end{aligned}$$

我们把 $g(x, y) = 0$ 这条曲线画在图上, 把 $f(x, y)$ 画等高线, 比如说图上等于 1 和 2, 我们的目标是要在 $g(x, y) = 0$ 这条曲线上找一个点使得 $f(x, y)$ 越小越好。



我们就把它想成气球, 这个气球一开始比较大, 这上面与 $g(x, y) = 0$ 交了两个点, 这时候能找到两个点且 $f(x, y)$ 等于 2, 然后我们尝试 $f(x, y)$ 变小, 直到与 $g(x, y) = 0$ 有一个交点, 最后变成没有交点, 这时候 $f(x, y)$ 达到最小, 所以可以得到最后最小的点肯定是 $g(x, y)$ 与等高线若近若离的, 这个若近若离的表示方法就是 $g(x, y) = 0$ 与等高线相切的, 那个点就是最优点。

所以得到以下式子:

- $\nabla f(x, y) = \lambda \nabla g(x, y)$
- $L(x, y, \lambda) = f(x, y) - \lambda g(x, y)$
- $\nabla L(x^*, y^*, \lambda) = 0$

我们将相切继续表示，表示成它们的法线，也就是梯度，这个梯度都会垂直于这个切线，所以我们直观上看，假如说 (x^*, y^*) 是最优解的话，那么在这个点上 $f(x, y)$ 不会再改变值了，沿着这条线走的话 $f(x, y)$ 数值不变，也就是说他两的梯度要在同一条直线上，有可能同向也有可能反向。可以表示为两个梯度成一定比例，我们把比例用 λ 表示 $\nabla f(x, y) = \lambda \nabla g(x, y)$ ，接着就可以得到拉格朗日方程 $L(x, y, \lambda) = f(x, y) - \lambda g(x, y)$ ，接着对于拉格朗日方程求导，等于 0 可以得到 $\nabla f(x, y) = \lambda \nabla g(x, y)$ ，所以得到求有约束的最小值，可以转化成 $\nabla L(x^*, y^*, \lambda) = 0$ 。

朗格朗日方程式原始的优化目标减去一个东西，这个东西可以看作是违反约束的阀，因为我们要求约束的结果等于 0，如果 $g(x, y)$ 不等于 0 那么我们就要减去后面这个式子，这个 λ 就是拉格朗日乘子。

拉格朗日乘子：

$$L(x, y, \lambda) = f(x, y) - \lambda g(x, y)$$

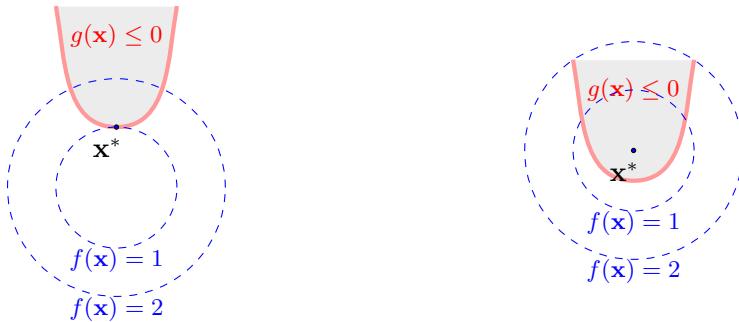
9.5.2 KKT 条件

前面我们已经完成了约束为 $g(x) = 0$ 的求解方法，那如果是小于等于 0，那就不是拉格朗日了，是拉格朗日的拓展，叫 KKT 条件。

引入

要求 $g(x, y) \leq 0$ ，也就是说图中的阴影区我们是要在阴影区或者边界上找一个点，使得 $f(x, y)$ 最小，我们把他分成两种情况：

1. 如果最优解恰好在这条边界上，那么这个就可以通过拉格朗日来求解。
2. 如果最优解在阴影区内部，我们是要找最优解，这个最优解不在边界上，在他内部，如果在内部那这个解一定是单纯考虑 $f(x, y)$ 的最优解，也就是直接求 $\nabla f(x, y) = 0$ 的解。



满足条件

KKT 条件可以这样来表示，首先先写一个拉格朗日 $L(x, y, \lambda) = f(x, y) - \lambda g(x, y)$ ，这个最优解需要满足的条件就是：

1. 与拉格朗日一样，使得式子的倒数等于 0。
2. 满足原先的可行性，原先的 $g(x, y) \leq 0$ 。
3. 对偶的可行性 $\lambda_i \leq 0$ 。
4. 满足互补的松弛性。每个 $\lambda_i g_i(x) = 0$ 。

9.6 向线性规划中引入拉格朗日对偶

9.6.1 拉格朗日对偶

对于线性规划：

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A} \mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

这是一个约束优化问题，反正对于约束优化问题都可以写出拉格朗日，拉格朗日是对每一个约束加了一个 λ ，变成：

$$L(\mathbf{x}, \lambda) = \mathbf{c}^T \mathbf{x} - \sum_{i=1}^m \lambda_i (a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n - b_i)$$

如果我们要求 $\lambda \geq 0$, 并且 \mathbf{x} 是可行解的话, 那么我们原先的目标函数肯定比 $L(\mathbf{x}, \lambda)$ 要大。

对于 $g(\lambda) = \inf_{\mathbf{x}} L(\mathbf{x}, \lambda)$, 就是对于任意的 λ 我都把所有的 \mathbf{x} 都枚举一遍, 算出它的下界当中最大的下界, 所以 $L(\mathbf{x}, \lambda)$ 一定大于等于 $g(\lambda)$ 。这里的 $g(\lambda)$ 就是拉格朗日对偶。

现在对于拉格朗日对偶进行解释, 得到以下求解过程:

$$\begin{aligned} g(\lambda) &= \inf_{\mathbf{x}} L(\mathbf{x}, \lambda) \\ &= \inf_{\mathbf{x}} (\mathbf{c}^T \mathbf{x} - \sum_{i=1}^m \lambda_i (a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n - b_i)) \\ &= \inf_{\mathbf{x}} (\mathbf{c}^T \mathbf{x} - \lambda^T (\mathbf{A}\mathbf{x} - \mathbf{b})) \\ &= \inf_{\mathbf{x}} (\mathbf{c}^T \mathbf{x} - \lambda^T \mathbf{A}\mathbf{x} + \lambda^T \mathbf{b}) \\ &= \inf_{\mathbf{x}} (\lambda^T \mathbf{b} + (\mathbf{c}^T - \lambda^T \mathbf{A})\mathbf{x}) \\ &= \begin{cases} \lambda^T \mathbf{b} & \text{IF } \mathbf{c}^T \geq \lambda^T \mathbf{A} \text{ AND } \mathbf{x} \geq \mathbf{0} \\ -\infty & \text{OTHERWISE} \end{cases} \end{aligned}$$

9.6.2 强边界

现在我们给出任意一个 $f(\mathbf{x})$, 我们现在可以通过拉格朗日搭一下桥, 我们想 $\min f(\mathbf{x})$, 现在可以得到 $f(\mathbf{x}) \geq L(\mathbf{x}, \lambda) \geq g(\lambda)$, 所以得到我要 $\min f(\mathbf{x})$, 最小最小不可能比 $g(\lambda)$ 要小了, 所以我们就 $\max g(\lambda)$, 任何一个 $g(\lambda)$ 都要比 $f(\mathbf{x})$ 要小, 所以最大的值就是 $f(\mathbf{x})$ 的最小值, 所以将 λ 换成 \mathbf{y} 就是我们所要得到的对偶。

9.6.3 例子

接着看一个例子, 原问题是:

$$\begin{aligned} \min \quad & x \\ \text{s.t.} \quad & x \geq 2 \\ & x \geq 0 \end{aligned}$$

写出他的拉格朗日为：

$$L(x, y) = x - y * (x - 2) = 2y + x * (1 - y)$$

所以拉格朗日对偶为：

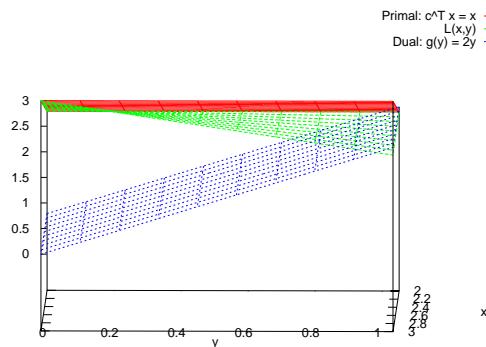
$$g(y) = \inf_x L(x, y) = \begin{cases} 2y & \text{IF } x \geq 0 \text{ AND } (1 - y) \geq 0 \\ -\infty & \text{OTHERWISE} \end{cases}$$

所以对偶问题为：

$$\begin{aligned} \max \quad & 2y \\ \text{s.t.} \quad & y \leq 1 \\ & y \geq 0 \end{aligned}$$

9.6.4 拉格朗日——原问题和对偶问题之间的桥梁

核心的问题是，我们原先要 MIN 的目标函数大于等于拉格朗日，拉格朗日大于等于对偶目标函数，也就是说 $\text{MIN } \mathbf{c}^T \mathbf{x} \geq \mathbf{L}(\mathbf{x}, \mathbf{y}) \geq \mathbf{y}^T \mathbf{b}$ ，所以通过拉格朗日搭了一下桥，所以我们原先的目标函数一定大于对偶的目标函数。图下图所示：



9.6.5 对偶变量 y

对偶用途非常广，尤其在经济学领域，经常把它叫做一个价格，经济学领域中常常用到线性规划，常常把对偶变量 y 叫做影子价格或者叫边界成本。边界成本就是说，当我们放松或者加强一个东西的时候，你要多少钱。边界成本就是拉格朗日乘子。

拉格朗日乘子就是说我违反了任意一个约束的时候你要罚我多少钱，我们原本的约束是 $Ax \geq b$ ，违反了以后就变成了 $b_i + \Delta b_i$ ，变了一点点，那要罚我都多少钱呢？那罚的钱数目就是 $\frac{\partial L(\mathbf{x}, \lambda)}{\partial b_i} = \lambda_i$ 。

通过例子可以看出对偶变量就是拉格朗日乘子。拉格朗日乘子就是约束稍微改变一点，对于目标函数有什么变化，有的改变对于目标函数没有变化，有的有变化，这个变化多少就是拉格朗日乘子就是我们的对偶变量。

例子：

DIET 问题的 b 的值为 $b_1 = 2000, b_2 = 55, b_3 = 800$ ，原始问题的可行解：

$$\mathbf{x} = (14.24, 2.70, 0, 0)$$

$$\mathbf{c}^T \mathbf{x} = 67.096$$

对偶问题的可行解：

$$\mathbf{y} = (0.0269, 0, 0.0164)$$

$$\mathbf{y}^T \mathbf{b} = 67.096$$

稍稍改变 b ，看对于 $\max \mathbf{c}^T \mathbf{x}$ 的影响：

- $b_1 = 2001: \max \mathbf{c}^T \mathbf{x} = 67.123 \quad (\mathbf{y}_1 = 0.0269 = 67.123 - 67.096)$
- $b_2 = 56: \max \mathbf{c}^T \mathbf{x} = 67.096 \quad (\mathbf{y}_2 = 0 = 67.096 - 67.096)$

- $b_3 = 801: \max \mathbf{c}^T \mathbf{x} = 67.112 (\mathbf{y}_3 = 0.0164 = 67.112 - 67.096)$

9.7 对偶的四个性质

9.7.1 性质一：对偶的对偶就是原问题

原始问题 (P):

$$\begin{aligned} & \min \quad \mathbf{c}^T \mathbf{x} \\ s.t. \quad & \mathbf{A} \mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

对偶问题 (D):

$$\begin{aligned} & \max \quad \mathbf{y}^T \mathbf{b} \\ s.t. \quad & \mathbf{x} \geq \mathbf{0} \\ & \mathbf{y}^T \mathbf{A} \leq \mathbf{c}^T \end{aligned}$$

对偶的改变 (D'):

$$\begin{aligned} & \min \quad \mathbf{y}^T (-\mathbf{b}) \\ s.t. \quad & \mathbf{x} \geq \mathbf{0} \\ & \mathbf{y}^T (-\mathbf{A}) \geq (-\mathbf{c}^T) \end{aligned}$$

对偶的对偶 ($D'D$):

$$\begin{aligned} & \max \quad \mathbf{x}^T (-\mathbf{c}^T) \\ s.t. \quad & \mathbf{x} \geq \mathbf{0} \\ & \mathbf{x}^T (-\mathbf{A}) \leq -\mathbf{b}^T \end{aligned}$$

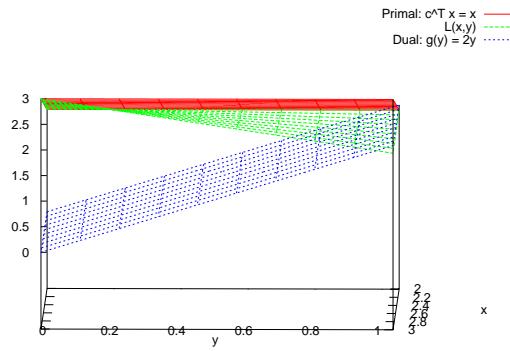
与原问题相同

9.7.2 性质二：弱对偶性

内容

对于一般问题，对偶问题的任意一个可行解总是原问题的一个下界。

例子



证明

原始问题：

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A} \mathbf{x} \geq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

对偶问题：

$$\begin{aligned} \max \quad & \mathbf{y}^T \mathbf{b} \\ \text{s.t.} \quad & \mathbf{y} \geq \mathbf{0} \\ & \mathbf{y}^T \mathbf{A} \leq \mathbf{c}^T \end{aligned}$$

推导：

- 由于 $\mathbf{y}^T \mathbf{A} \leq \mathbf{c}^T$ 并且 $\mathbf{x}^T \geq \mathbf{0}$, 所以 $\mathbf{c}^T \mathbf{x} \geq \mathbf{y}^T \mathbf{A} \mathbf{x}$ 。
- 由于 $\mathbf{A} \mathbf{x} \geq \mathbf{b}$ 并且 $\mathbf{y} \geq \mathbf{0}$, 所以 $\mathbf{c}^T \mathbf{x} \geq \mathbf{y}^T \mathbf{A} \mathbf{x} \geq \mathbf{y}^T \mathbf{b}$ 。

9.7.3 性质三：强对偶性

内容

对于线性规划，原问题的存在最优解，那对偶问题也存在一个最优解，它们的数值相等。

证明

- 线性规划的最优解肯定能够写成 $\mathbf{x}^* = \begin{bmatrix} \mathbf{B}^{-1}\mathbf{b} \\ 0 \end{bmatrix}$ ，最优解肯定在顶点上，顶点就是一个基，假如这个基是 \mathbf{B} 则 \mathbf{x} 一定写成 $\mathbf{B}^{-1}\mathbf{b}$ ，其它的非基向量为 0。且停止条件是 $\mathbf{c}^T - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{A} \geq 0$ 。
- 我们定义： $\mathbf{y}^{*T} = \mathbf{c}_B^T \mathbf{B}^{-1}$ 。 \mathbf{y}^{*T} 是对偶的最优解。
- 带入得到， $\mathbf{y}^{*T}\mathbf{b} = \mathbf{c}_B^T \mathbf{B}^{-1}\mathbf{b} = \mathbf{c}^T \mathbf{x}^*$ 。
- 因为所有的 \mathbf{y} 带入对偶问题都小于等于原始问题，现在找到一个相等的了，所以这个 \mathbf{y}^* 一定是最优解。

9.7.4 性质四：互补松弛性

内容

假如 \mathbf{x} 是原问题的可行解， \mathbf{y} 为对偶问题的可行解，那么 \mathbf{x} 和 \mathbf{y} 是最优解时，当且仅当：

- 对于任意 $1 \leq i \leq m$ ， $u_i = y_i(a_{i1}x_1 + a_{i2}x_2 + \dots + a_{in}x_n - b_i) = 0$ 。
- 对于任意 $1 \leq j \leq n$ ， $v_j = (c_j - a_{1j}y_1 - a_{2j}y_2 - \dots - a_{mj}y_m)x_j = 0$ 。

例子

DIET 问题的可行解和他的对偶问题的可行解是: $\mathbf{x} = (14.244, 2.707, 0, 0)$ 和 $\mathbf{y} = (0.0269, 0, 0.0164)$ 。可以得到:

$$\begin{aligned} 110x_1 + 160x_2 + 420x_3 + 260x_4 &= 2000 \\ 4x_1 + 8x_2 + 4x_3 + 14x_4 &> 55 \Rightarrow y_2 = 0 \\ 2x_1 + 285x_2 + 22x_3 + 80x_4 &= 800 \\ x_1, x_2, x_3, x_4 &\geq 0 \end{aligned}$$

证明

证:

- 对于任意 i 和 $j, u_i = 0$ 并且 $v_j = 0$
- $\Leftrightarrow \sum_i u_i = 0$ 并且 $\sum_j v_j = 0$ (因为 $u_i \geq 0, v_j \geq 0$)
- $\Leftrightarrow \sum_i u_i + \sum_j v_j = 0$
- $\Leftrightarrow (\mathbf{y}^T \mathbf{A} \mathbf{x} - \mathbf{y}^T \mathbf{b}) + (\mathbf{c}^T \mathbf{x} - \mathbf{y}^T \mathbf{A} \mathbf{x}) = 0$
- $\Leftrightarrow \mathbf{y}^T \mathbf{b} = \mathbf{c}^T \mathbf{x}$
- $\Leftrightarrow \mathbf{y}$ 和 \mathbf{x} 是可行解 (通过强对偶性质可以得到, $\mathbf{y}^T \mathbf{b}$ 和 $\mathbf{c}^T \mathbf{x}$ 到达了他们自身的边界)

9.8 原问题和对偶问题的 9 种情况

给我们任何的实际问题, 我们可以写出他的线性规划, 然后我们就可以机械的写出他的对偶, 然后分别进行求解。

Primal Dual	Bounded Optimal Objective Value	Unbounded Optimal Objective Value	Infeasible
Bounded Optimal Objective Value	Possible	Impossible	Impossible
Unbounded Optimal Objective Value	Impossible	Impossible	Possible
Infeasible	Impossible	Possible	Possible

9.8.1 例子 1：原问题有无界最优解，对偶问题无解

- 原问题：

$$\begin{aligned}
 \min \quad & -2x_1 - 2x_2 \\
 \text{s.t.} \quad & x_1 - x_2 \leq 1 \\
 & -x_1 + x_2 \leq 1 \\
 & x_1 \geq 0 \\
 & x_2 \geq 0
 \end{aligned}$$

- 对偶问题

$$\begin{aligned}
 \max \quad & y_1 + y_2 \\
 \text{s.t.} \quad & y_1 \leq 0 \\
 & y_2 \leq 0 \\
 & y_1 - y_2 \leq -2 \\
 & -y_1 + y_2 \leq -2
 \end{aligned}$$

9.8.2 例子 2：原问题和对偶问题都无解

- 原问题：

$$\begin{aligned}
 \min \quad & x_1 - 2x_2 \\
 \text{s.t.} \quad & x_1 - x_2 \geq 2 \\
 & -x_1 + x_2 \geq -1 \\
 & x_1 \geq 0 \\
 & x_2 \geq 0
 \end{aligned}$$

- 对偶问题:

$$\begin{aligned}
 \max \quad & 2y_1 - y_2 \\
 \text{s.t.} \quad & y_1 \geq 0 \\
 & y_2 \geq 0 \\
 & y_1 - y_2 \leq 1 \\
 & -y_1 + y_2 \leq -2
 \end{aligned}$$

9.9 对偶应用 1: Farkas 引理

9.9.1 引理内容

给我一堆的向量 $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m, \mathbf{c} \in \mathbb{R}^n$, 那么就可以得到:

1. $\mathbf{c} \in \mathbf{C}(\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m)$, 或者
2. 存在一个向量 $\mathbf{y} \in \mathbb{R}^n$, 对于任意的 i , $\mathbf{y}^T \mathbf{a}_i \geq 0$ 但是 $\mathbf{y}^T \mathbf{c} < 0$ 成立。

9.9.2 引理图例解释

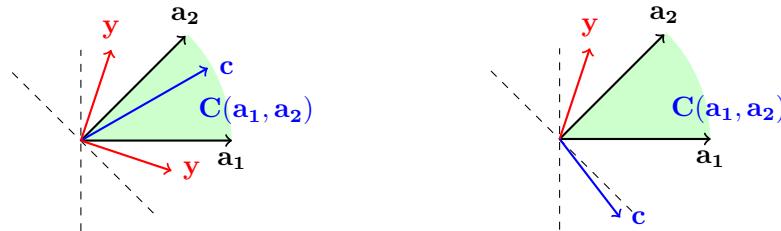


图 9.1: 第一种情况: $\mathbf{c} \in \mathbf{C}(\mathbf{a}_1, \mathbf{a}_2)$

图 9.2: 第二种情况: $\mathbf{c} \notin \mathbf{C}(\mathbf{a}_1, \mathbf{a}_2)$

9.9.3 引理证明

证:

- 假如对于任意向量 $\mathbf{y} \in \mathbb{R}^n$, $\mathbf{y}^T \mathbf{a}_i \geq 0$ ($i = 1, 2, \dots, m$), 同时和任意的边界都大于等于 0, 即 $\mathbf{y}^T \mathbf{c} \geq 0$ 。那么 \mathbf{c} 一定在 $\mathbf{C}(\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m)$ 中。

- 考虑到以下的线性规划的原始问题:

$$\begin{aligned} \min \quad & \mathbf{c}^T \mathbf{y} \\ \text{s.t.} \quad & \mathbf{a}_i^T \mathbf{y} \geq \mathbf{0} \quad i = 1, 2, \dots, m \end{aligned}$$

- 由于 $\mathbf{c}^T \mathbf{y} \geq \mathbf{0}$, 所以显然可以得到 $\mathbf{c}^T \mathbf{y}$ 有一个下界为 0, 所以可以得到原问题的一个可行解是 $\mathbf{y} = \mathbf{0}$ 。

- 因此可以得到他的对偶为:

$$\begin{aligned} \max \quad & 0 \\ \text{s.t.} \quad & \mathbf{x}^T \mathbf{A}^T = \mathbf{c}^T \\ & \mathbf{x} \geq \mathbf{0} \end{aligned}$$

- 因此我们可以得到, 存在一个向量 \mathbf{x} 使得 $\mathbf{c} = \sum_{i=1}^m x_i \mathbf{a}_i$ 。所以得 \mathbf{c} 可以分解为 \mathbf{A} 的一个线性组合, 即 \mathbf{c} 在 \mathbf{A} 的内部。

9.9.4 Farkas 引理变种

FARKAS 引理是线性规划的核心, 它可以推出很多东西, 比如分离引理, 博弈论里的 MINMAX 引理等等。

变种 1

如果 \mathbf{A} 是一个 $m \times n$ 的矩阵, 并且 $\mathbf{b} \in \mathbb{R}^m$, 那么可以得到:

1. $\mathbf{Ax} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}$ 是一个可行解, 或者
2. 存在一个向量 $\mathbf{y} \in \mathbb{R}^m$ 使得 $\mathbf{y}^T \mathbf{A} \geq \mathbf{0}$ 但是 $\mathbf{y}^T \mathbf{b} < 0$ 。

变种 2

给一些向量 $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m \in \mathbb{R}^n$ 。如果 $\mathbf{x} \in \mathbf{C}(\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m)$, 那么就存在一个线性堵路的向量集 $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m$, 对于向量集 $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_r$, 可以得到 $\mathbf{x} \in \mathbf{C}(\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_r)$ 。

变种 3

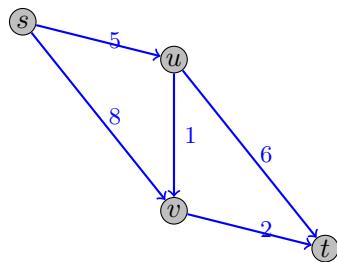
如果 $\mathbf{C} \subset \mathbb{R}^n$ 是一个闭的凸集，并且 $\mathbf{x} \in \mathbb{R}^n$. 如果 $\mathbf{x} \notin \mathbf{C}$, 那么肯定存在一个超平面分离 \mathbf{x} 和 \mathbf{C} 。

9.10 对偶应用 2：最短路径问题

9.10.1 最短路径问题回顾

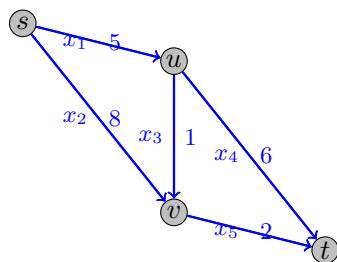
输入： n 个城市和一些道路的集合。一条从城市 i 到城市 j 的路的距离记作 $d(i, j)$ 。两个特殊的城市： s 和 t 。

输出： 城市 s 和城市 t 之间的最短路径。



9.10.2 线性规划形式

每条边我都设一个变量，可以得到以下图：



如果我用了这条边，我就让 x_i 的值等于 1，没走这条边 x_i 等于 0，所以可以写成一个 0/1 的线性规划，即一个整数线性规划。

写出整数线性规划方程：

$$\begin{aligned} \min \quad & 5x_1 + 8x_2 + 1x_3 + 6x_4 + 2x_5 \\ s.t. \quad & x_1 + x_2 = 1 \quad \text{向量 } s \\ & -x_4 - x_5 = -1 \quad \text{向量 } t \\ & -x_1 + x_3 + x_4 = 0 \quad \text{向量 } u \\ & -x_2 - x_3 + x_5 = 0 \quad \text{向量 } v \\ & x_1, x_2, x_3, x_4, x_5 = 0/1 \end{aligned}$$

由于纯在全单模条件，所以可以将这个 ILP 问题转化成 LP 问题，可以得到以下的线性规划方程：

$$\begin{aligned} \min \quad & 5x_1 + 8x_2 + 1x_3 + 6x_4 + 2x_5 \\ s.t. \quad & x_1 + x_2 = 1 \quad \text{向量 } s \\ & -x_4 - x_5 = -1 \quad \text{向量 } t \\ & -x_1 + x_3 + x_4 = 0 \quad \text{向量 } u \\ & -x_2 - x_3 + x_5 = 0 \quad \text{向量 } v \\ & x_1, x_2, x_3, x_4, x_5 \geq 0 \\ & x_1, x_2, x_3, x_4, x_5 \leq 1 \end{aligned}$$

9.10.3 写出对偶问题

通过原始问题我们可以机械的写出它的对偶问题，得到对偶的线性规划方程为：

$$\begin{aligned} \max \quad & y_s - y_t \\ s.t. \quad & y_s - y_u \leq 5 \quad x_1 : \text{边}(s, u) \\ & y_s - y_v \leq 8 \quad x_2 : \text{边}(s, v) \\ & y_u - y_v \leq 1 \quad x_3 : \text{边}(u, v) \\ & -y_t + y_u \leq 6 \quad x_4 : \text{边}(u, t) \\ & -y_t + y_v \leq 2 \quad x_5 : \text{边}(v, t) \end{aligned}$$

根据下图，我们从对偶问题中可以看出，现在我们对于每个城市设一个变量 y_s ，我们用 y_s 来表示这个城市的海拔高度， y_u 来表达另一个城市的海拔高度，现在我们知道 s 到 u 有一条路走 5 公里，那么他们两的海拔高度差肯定不会超过 5 公里，目标函数是 y_s 减去 y_t 就是 s 到 t 的最小的海拔差，我们 MAX 下它，可以得到它的下界。这就是对偶问题的直观含义。

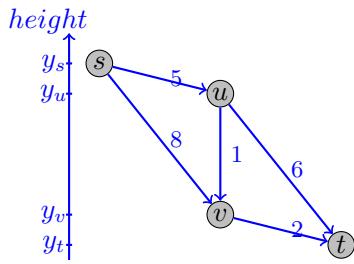


图 9.3: 对偶问题的直观含义

9.11 对偶单纯形算法

9.11.1 回顾原始问题单纯形算法

原始问题：

$$\begin{aligned}
 \min \quad & x_1 + 14x_2 + 6x_3 \\
 \text{s.t.} \quad & x_1 + x_2 + x_3 \leq 4 \\
 & x_1 \leq 2 \\
 & x_3 \leq 3 \\
 & 3x_2 + x_3 \leq 6 \\
 & x_1, x_2, x_3 \geq 0
 \end{aligned}$$

单纯形算法很简单，你把矩阵 \mathbf{A} 照抄下来，上面把 c 都照抄下来， b 写在左边，初始化的最优值写左边为 0，可以得到下图所示表格：

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$-z = 0$	$\bar{c}_1 = 1$	$\bar{c}_2 = 14$	$\bar{c}_3 = 6$	$\bar{c}_4 = 0$	$\bar{c}_5 = 0$	$\bar{c}_6 = 0$	$\bar{c}_7 = 0$
$\mathbf{x}_{B1} = b'_1 = 4$	1	1	1	1	0	0	0
$\mathbf{x}_{B2} = b'_2 = 2$	1	0	0	0	1	0	0
$\mathbf{x}_{B3} = b'_3 = 3$	0	0	1	0	0	1	0
$\mathbf{x}_{B4} = b'_4 = 6$	0	3	1	0	0	0	1

先找一个基可行解，所有都对他做高斯行变换，包括右边的部分都变成 0。然后写一个 WHILE 循环，里面分三步：

- 先找上面 c_i 内有没有负数的
- 有负数的话找对应的一列，在里面正的里面找最小，做高斯行变换
- 直到 c_i 内全是非负为止。

最终就能得到最优解。

原始的解为 \mathbf{x} ，什么时候可行呢？即 \mathbf{x} 等于 $\mathbf{B}^{-1}\mathbf{b}$ ，那么 $\mathbf{B}^{-1}\mathbf{b} \geq 0$ 是显然的。

将线性规划写出他的对偶出来，可以得到：

$$\begin{aligned}
 \max \quad & 4y_1 + 2y_2 + 3y_3 + 6y_4 \\
 s.t. \quad & y_1 + y_2 \leq 1 \\
 & y_1 + 3y_4 \leq 14 \\
 & y_1 + y_3 + y_4 \leq 6 \\
 & y_1, y_2, y_3, y_4 \leq 0
 \end{aligned}$$

对应下表：

可以得到：

- 对偶变量是： $\mathbf{y}^T = \mathbf{c}_B^T \mathbf{B}^{-1}$ ，可行解为 $\mathbf{y}^T \mathbf{A}$ 且可行性是 $\mathbf{y}^T \mathbf{A} \leq \mathbf{c}^T$ 。

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$-z = 0$	$\bar{c}_1 = 1$	$\bar{c}_2 = 14$	$\bar{c}_3 = 6$	$\bar{c}_4 = 0$	$\bar{c}_5 = 0$	$\bar{c}_6 = 0$	$\bar{c}_7 = 0$
$\mathbf{x}_{B1} = b'_1 = 4$	1	1	1	1	0	0	0
$\mathbf{x}_{B2} = b'_2 = 2$	1	0	0	0	1	0	0
$\mathbf{x}_{B3} = b'_3 = 3$	0	0	1	0	0	1	0
$\mathbf{x}_{B4} = b'_4 = 6$	0	3	1	0	0	0	1

- 如果一个基称为对偶可行是说 $\bar{\mathbf{c}}^T = \mathbf{c} - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{A} = \mathbf{c}^T - \mathbf{y}^T \mathbf{A} \geq 0$ 。即只要 c_i 行都大于等于 0，就是对偶可行解。

9.11.2 从另一个角度看原始问题单纯形算法

可以将单纯形算法看做：

- 起始点：**一开始的时候我们找一个可行解，这个可行解是 $\mathbf{x}_B = \mathbf{B}^{-1} \mathbf{b}$ 且 $\mathbf{B}^{-1} \mathbf{b} \geq 0$ 。
- 保持：**我们在做的过程中，始终要保持左边大于等于 0，原问题是可行解。
- 停止：**停止是最上面的 \mathbf{c}_i 行要大于等于 0，即 $\bar{\mathbf{c}}^T = \mathbf{c}^T - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{A} \geq 0$ 。这个从另外一个角度看就是对偶是可行的。

我们把我们过去的算法从对偶的观点重新再解释一遍，从一个原问题可行解出发，始终保持原问题是可行的，不断地做直到对偶也可行，一旦对偶可行我们就知道了这两者都可行了，我们就得到了最优解了。

我们也可以这样做，假如我们始终保持对偶可行，初始的数值就让对偶可行，在做的过程中始终保持对偶可行，直到最后原问题也可行，因为这两个问题都可行的话就得到最优解了，所以这么做也可以。

9.11.3 对偶单纯形算法

通过上述的推倒，我们可以得到对偶单纯形算法为：

DUAL SIMPLEX($B_I, z, \mathbf{A}, \mathbf{b}, \mathbf{c}$)

```

1: //DUAL SIMPLEX STARTS WITH A DUAL FEASIBLE BASIS. HERE,  $B_I$  CONTAINS THE INDICES OF THE BASIC
   VARIABLES.
2: while TRUE do
3:   if THERE IS NO INDEX  $l$  ( $1 \leq l \leq m$ ) HAS  $b_l < 0$  then
4:      $\mathbf{x} = \text{CALCULATEX}(B_I, \mathbf{A}, \mathbf{b}, \mathbf{c})$ ;
5:     return ( $\mathbf{x}, z$ );
6:   end if;
7:   CHOOSE AN INDEX  $l$  HAVING  $b_l < 0$  ACCORDING TO A CERTAIN RULE;
8:   for EACH INDEX  $j$  ( $1 \leq i \leq n$ ) do
9:     if  $a_{lj} < 0$  then
10:       $\Delta_j = -\frac{c_j}{a_{lj}}$ ;
11:    else
12:       $\Delta_j = \infty$ ;
13:    end if
14:   end for
15:   CHOOSE AN INDEX  $e$  THAT MINIMIZES  $\Delta_j$ ;
16:   if  $\Delta_e = \infty$  then
17:     return "'NO FEASIBLE SOLUTION'";
18:   end if
19:    $(B_I, \mathbf{A}, \mathbf{b}, \mathbf{c}, z) = \text{PIVOT}(B_I, \mathbf{A}, \mathbf{b}, \mathbf{c}, z, e, l)$ ;
20: end while
```

9.11.4 例子

- 标准型:

$$\begin{aligned} \min \quad & 5x_1 + 35x_2 + 20x_3 \\ s.t. \quad & x_1 - x_2 - x_3 \leq -2 \\ & -x_1 - 3x_2 \leq -3 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

- 松弛型:

$$\begin{aligned} \min \quad & 5x_1 + 35x_2 + 20x_3 \\ s.t. \quad & x_1 - x_2 - x_3 + x_4 = -2 \\ & -x_1 - 3x_2 + x_5 = -3 \\ & x_1, x_2, x_3, x_4, x_5 \geq 0 \end{aligned}$$

第一步

	x_1	x_2	x_3	x_4	x_5
$-z = 0$	$\bar{c}_1 = 5$	$\bar{c}_2 = 35$	$\bar{c}_3 = 20$	$\bar{c}_4 = 0$	$\bar{c}_5 = 0$
$\mathbf{x}_{B1} = b'_1 = -2$	1	-1	-1	1	0
$\mathbf{x}_{B2} = b'_2 = -3$	-1	-3	0	0	1

- 基 (蓝色): $\mathbf{B} = \{\mathbf{a}_4, \mathbf{a}_5\}$ 。

- 解: $\mathbf{x} = \begin{bmatrix} \mathbf{B}^{-1}\mathbf{b} \\ \mathbf{0} \end{bmatrix} = [0, 0, 0, -2, -3]^T$ 。

- 推进: 选择 \mathbf{a}_5 是因为 $b'_2 = -3 < 0$; 选择 \mathbf{a}_1 是因为 $\min_{j, a_{2j} < 0} \frac{\bar{c}_j}{-a_{2j}} = \frac{\bar{c}_1}{-a_{21}}$ 。

第二步

- 基 (蓝色): $\mathbf{B} = \{\mathbf{a}_1, \mathbf{a}_4\}$ 。

- 解: $\mathbf{x} = \begin{bmatrix} \mathbf{B}^{-1}\mathbf{b} \\ \mathbf{0} \end{bmatrix} = [3, 0, 0, -5, 0]^T$ 。

	x_1	x_2	x_3	x_4	x_5
$-z = -15$	$\bar{c}_1 = 0$	$\bar{c}_2 = 20$	$\bar{c}_3 = 20$	$\bar{c}_4 = 0$	$\bar{c}_5 = 5$
$\mathbf{x}_{B1} = b'_1 = -5$	0	-4	-1	1	1
$\mathbf{x}_{B2} = b'_2 = 3$	1	3	0	0	-1

- 推进: 选择 \mathbf{a}_4 是因为 $b'_1 = -5 < 0$; 选着 \mathbf{a}_2 是因为 $\min_{j, a_{1j} < 0} \frac{\bar{c}_j}{-a_{1j}} = \frac{\bar{c}_2}{-a_{12}}$ 。

第三步

	x_1	x_2	x_3	x_4	x_5
$-z = -40$	$\bar{c}_1 = 0$	$\bar{c}_2 = 0$	$\bar{c}_3 = 15$	$\bar{c}_4 = 5$	$\bar{c}_5 = 10$
$\mathbf{x}_{B1} = b'_1 = \frac{5}{4}$	0	1	$\frac{1}{4}$	$-\frac{1}{4}$	$-\frac{1}{4}$
$\mathbf{x}_{B2} = b'_2 = -\frac{3}{4}$	1	0	$-\frac{3}{4}$	$\frac{3}{4}$	$-\frac{1}{4}$

- 基 (蓝色): $\mathbf{B} = \{\mathbf{a}_1, \mathbf{a}_2\}$ 。
- 解: $\mathbf{x} = \begin{bmatrix} \mathbf{B}^{-1}\mathbf{b} \\ \mathbf{0} \end{bmatrix} = [\frac{5}{4}, -\frac{3}{4}, 0, 0, 0]^T$ 。
- 推进: 选择 \mathbf{a}_1 是因为 $b'_2 = -\frac{3}{4} < 0$; 选择 \mathbf{a}_3 是因为 $\min_{j, a_{2j} < 0} \frac{\bar{c}_j}{-a_{2j}} = \frac{\bar{c}_3}{-a_{23}}$ 。

第四步

	x_1	x_2	x_3	x_4	x_5
$-z = -55$	$\bar{c}_1 = 20$	$\bar{c}_2 = 0$	$\bar{c}_3 = 0$	$\bar{c}_4 = 20$	$\bar{c}_5 = 5$
$\mathbf{x}_{B1} = b'_1 = 1$	$\frac{1}{3}$	1	0	0	$-\frac{1}{3}$
$\mathbf{x}_{B2} = b'_2 = 1$	$-\frac{4}{3}$	0	1	-1	$\frac{1}{3}$

- 基 (蓝色): $\mathbf{B} = \{\mathbf{a}_2, \mathbf{a}_3\}$ 。
- 解: $\mathbf{x} = \begin{bmatrix} \mathbf{B}^{-1}\mathbf{b} \\ \mathbf{0} \end{bmatrix} = [0, 1, 1, 0, 0]^T$ 。

- 完成！

9.11.5 对偶单纯形算法用处

1. 对偶单纯形，如果我们原始对偶可行解好找，我们就跑对偶单纯形。很多问题我们的 c 都大于等于 0，那么一开始就可以找到对偶可行解，就不用一开始费劲找原始问题的可行解了。
2. 我们日常会碰到很多的需求，比如解一个特别大的线性规划，但是解了之后别人又提了一些新要求，又要添加新约束或者更改参数，使用对偶单纯形就不用重新计算。
3. 如果我们约束的数目比变量的数目大很多，跑对偶单纯形要好，速度非常快。
4. 如果我们的线性规划是退化的情形，我们也要试一下对偶单纯形，这点是非常快的。

9.12 原始对偶算法

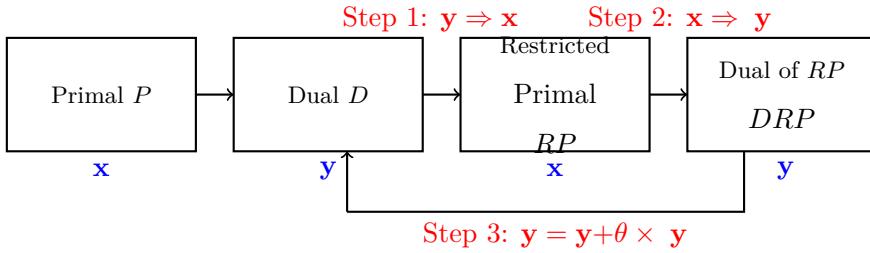
9.12.1 原始对偶算法引入

今天我们讲一个很重要的 TECHNIQUE，叫原始对偶。原始对偶也是一种迭代式的，逐步改进式的迭代方法。这两节课都比较难，但是当你觉得它有用的时候，你就会觉得它不这么难了。原始对偶方法也是一种对偶方法，他也要解一个对偶。它充分利用问题的下界信息。原始对偶和原来的对偶单纯型不太一样，它不像对偶单纯型，一定要从一个对偶基可行解出发。原始对偶方法只要求对偶可行就行。第二，在原始对偶方法中，每次都要迭代解一个 DRP。DRP 是一个小线性规划，但很多时候，我们不用单纯型去解它。因为它有很多的直观的组合解释，尤其对于图论问题。

9.12.2 原始对偶算法

原始对偶算法基本思路：

我们用这幅图来说明原始对偶方法：



假如我们要求解远线性规划问题 P ，它的变量是 X 。我们先把它的对偶问题写出来。假如我们拿到对偶问题的一个可行解 Y ，我们把 Y 带入到对偶问题中，看看 Y 能指导我们得到 X 多少的信息，得到的 X 的信息又能够知道我们如何改进 Y 。通过这样不断迭代改进 Y ，找到最优解。这里有很多名词，大家先不要着急，一个个的看下去。

得到 RP

看个例子。

- PRIMAL P :

$$\begin{aligned}
 & \min \quad c_1 x_1 + c_2 x_2 + \dots + c_n x_n \\
 & s.t. \quad a_{11} x_1 + a_{12} x_2 + \dots + a_{1n} x_n = b_1 \quad (\mathbf{y}_1) \\
 & \quad a_{21} x_1 + a_{22} x_2 + \dots + a_{2n} x_n = b_2 \quad (\mathbf{y}_2) \\
 & \quad \dots \\
 & \quad a_{m1} x_1 + a_{m2} x_2 + \dots + a_{mn} x_n = b_m \quad (\mathbf{y}_m) \\
 & \quad x_1, x_2, \dots, x_n \geq 0
 \end{aligned}$$

- DUAL D :

$$\begin{aligned}
 & \max \quad b_1 y_1 + b_2 y_2 + \dots + b_m y_m \\
 & s.t. \quad a_{11} y_1 + a_{21} y_2 + \dots + a_{m1} y_m \leq c_1 \\
 & \quad a_{12} y_1 + a_{22} y_2 + \dots + a_{m2} y_m \leq c_2 \\
 & \quad \dots \\
 & \quad a_{1n} y_1 + a_{2n} y_2 + \dots + a_{mn} y_m \leq c_n
 \end{aligned}$$

原始问题如上所示，我们将它的对偶问题写出来。怎么写对偶呢，我们再重复一遍。对偶就是拉格朗日乘子，每一行的约束做一个拉格朗日乘子，就是变量，叫做 y_1, y_2, \dots 。然后写出对偶问题。

假如我们拿到了对偶问题的一个可行解 Y ，我们首先验证 Y 是不是最优解。如何验证呢？

首先，如果 Y 是最优解，则 X 要满足一个条件。什么条件呢？就是这样一个条件：

- DUAL PROBLEM D:

$$\begin{array}{lllllllll} \max & b_1 y_1 & + & b_2 y_2 & + & \dots & + & b_m y_m \\ s.t. & a_{11} y_1 & + & a_{21} y_2 & + & \dots & + & a_{m1} y_m & \leq c_1 \quad ('= \Rightarrow x_1 \geq 0) \\ & & & & & \dots & & & \\ & a_{1n} y_1 & + & a_{2n} y_2 & + & \dots & + & a_{mn} y_m & \leq c_n \quad ('< \Rightarrow x_n = 0) \end{array}$$

假如我们拿到一个 y , 我们把 y 带入对偶问题的约束中, 对于每一条约束, 如果约束 1 取等号, 则相当于没有告诉我 x_1 的任何信息, $x_1 \geq 0$ 是本来就已知的。如果约束 N 取小于号, 大家回忆一下我们讲的互补松弛性, 如果约束取小于, x_N 必定等于 0。我们用 J 表示满足等于号的约束:

c_1	c_2	\dots	c_n
\parallel	\parallel	\dots	\vee
$y_1 a_{11}$	$y_1 a_{12}$	\dots	$y_1 a_{1n}$
+	+	\dots	+
$y_2 a_{21}$	$y_2 a_{22}$	\dots	$y_2 a_{2n}$
+	+	\dots	+
\vdots	\vdots	\dots	\vdots
+	+	\dots	+
$y_m a_{m1}$	$y_m a_{m2}$	\dots	$y_m a_{mn}$

我们回到原始问题，如果 Y 是一个最优解，那么红框内表示满足的等号约束，篮框内的取小于号，对应的 $x_{N=0}$ 。也就是说给我一个 Y ，如果是最优的，带进去， X 必须要满足这些条件：

- RP:

$$\begin{aligned}
 a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\
 a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\
 &\dots \\
 a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &= b_m \\
 x_i &= 0 \quad i \notin J \\
 x_i &\geq 0 \quad i \in J
 \end{aligned}$$

我们只需要解这个限制性的原问题 RESTRICTED PRIMAL (RP) 就可以了。没有目标函数，只需要 X 满足这些约束就够了。解这个不等式约束问题，我们把它转化成线性规划问题来做。加一些松弛变量:s1,s2,...,sm, 变成下面这样一个问题：

$$\begin{aligned}
 \min \quad \epsilon &= s_1 + s_2 + \dots + s_m \\
 \text{s.t.} \quad s_1 &+ a_{11}x_1 + \dots + a_{1n}x_n = b_1 \\
 s_2 &+ a_{21}x_1 + \dots + a_{2n}x_n = b_2 \\
 &\dots \\
 s_m &+ a_{m1}x_1 + \dots + a_{mn}x_n = b_m \\
 x_i &= 0 \quad i \notin J \\
 x_i &\geq 0 \quad i \in J \\
 s_i &\geq 0 \quad \forall i
 \end{aligned}$$

最优值如果等于 0，表明我们能找到一个 X 满足 RP。如果最优值大于 0，RP 没有可行解，所以 Y 就不是最优解。

如何求解 RP: DRP

现在，我们回顾一下。如果 Y 是最优解，对应的 X 满足原先的约束，并且有些 $X_i=0$ (蓝框内)。我们只要找到这些 X, Y 就是最优的。如何找这些 X 呢？求解 RP 对应的线性规划，当然，我们不一定需要直接去解 RP 对应的线性规划，我们解 RP 对应的线性规划的对偶 (DRP)，也是等价的：

- DRP:

$$\begin{aligned}
 \max \quad w &= b_1 y_1 + b_2 y_2 + \dots + b_m y_m \\
 \text{s.t.} \quad a_{11} y_1 + a_{21} y_2 + \dots + a_{m1} y_m &\leq 0 \\
 a_{12} y_1 + a_{22} y_2 + \dots + a_{m2} y_m &\leq 0 \\
 &\dots \\
 a_{1|J|} y_1 + a_{2|J|} y_2 + \dots + a_{m|J|} y_m &\leq 0 \\
 y_1, y_2, \dots, y_m &\leq 1
 \end{aligned}$$

当最优值是 0 的时候，Y 就是最优的。否则 Y 不是最优的。当 Y 不是最优时，应该怎么办。这个时候虽然 Y 不是最优，但是我们对这个问题的求解所花的功夫没有白费。它提供了有用的信息，它可以告诉我们怎么改进 Y。

DRP 优化 y

为什么说我们求得的 DRP 的解可以改进 Y 的呢？我们构造一个解： $\mathbf{y}' = \mathbf{y} + \theta \mathbf{y}$, $\theta > 0$

- 如果说它要是改进的话，我们只要理解两点。第一点目标函数的确是变大啦。第二点，Y 是满足约束的，Y' 也应该是满足约束的。

我们先看第一点。DRP 问题的目标函数和对偶问题的目标函数是一样的。所以，如果 DRP 问题的目标函数能找到一个最优解，它是等于 0 的，那我们就已经 STOP 了，如果是大于 0 的话，我们可以知道， $\mathbf{y}^T \mathbf{b} = w_{OPT} > 0$, $\mathbf{y}'^T \mathbf{b} = \mathbf{y}^T \mathbf{b} + \theta w_{OPT} > \mathbf{y}^T \mathbf{b}$.

第二点，Y 本来是满足约束的，会不会变大以后就不满足约束了呢。对于任意的 $j \in J$, $a_{1j} \Delta y_1 + a_{2j} \Delta y_2 + \dots + a_{mj} \Delta y_m \leq 0$ (依据 DRP 的约束)。所以我们有 $\mathbf{y}'^T \mathbf{a}_j = \mathbf{y}^T \mathbf{a}_j + \theta \mathbf{y}^T \mathbf{a}_j \leq \mathbf{c}_j$ 对于任意的 $\theta > 0$. 对于 $j \notin J$, 又分两种情况：

第一种，对于 $\forall j \notin J$, $a_{1j} \Delta y_1 + a_{2j} \Delta y_2 + \dots + a_{mj} \Delta y_m \leq 0$:

\mathbf{y}' 肯定是一个可行解，对于 $\forall \theta > 0$, $\forall 1 \leq j \leq n$:

$$a_{1j} y'_1 + a_{2j} y'_2 + \dots + a_{mj} y'_m \quad (9.12.1)$$

$$= a_{1j} y_1 + a_{2j} y_2 + \dots + a_{mj} y_m \quad (9.12.2)$$

$$+ \theta(a_{1j} \Delta y_1 + a_{2j} \Delta y_2 + \dots + a_{mj} \Delta y_m) \quad (9.12.3)$$

$$\leq c_j \quad (9.12.4)$$

换句话说，对偶问题是无界的，原问题是不可行的。因为原问题的解可以任意大。

第二种， $\exists j \notin J$, $a_{1j} \Delta y_1 + a_{2j} \Delta y_2 + \dots + a_{mj} \Delta y_m > 0$:

我们可以设置 $\theta \leq \frac{c_j - (a_{1j}y_1 + a_{2j}y_2 + \dots + a_{mj}y_m)}{a_{1j}\Delta y_1 + a_{2j}\Delta y_2 + \dots + a_{mj}\Delta y_m} = \frac{c_j - \mathbf{y}^T \mathbf{a}_j}{\mathbf{y}^T \mathbf{a}_j}$ 从而使得 $\mathbf{y}'^T \mathbf{a}_j = \mathbf{y}^T \mathbf{a}_j + \theta \mathbf{y}^T \mathbf{a}_j \leq c_j$.

原始对偶算法

讲完这些，原始对偶算法就出来啦：

- 原始对偶算法：

```

1: INFEASIBLE = "No"
    OPTIMAL = "No"
     $\mathbf{y} = \mathbf{y}_0$ ; //  $\mathbf{y}_0$  IS A FEASIBLE SOLUTION TO THE DUAL PROBLEM  $D$ 
2: while TRUE do
3:   FINDING TIGHT CONSTRAINTS INDEX  $J$ , AND SET CORRESPONDING  $x_j = 0$  FOR  $j \notin J$ .
4:   THUS WE HAVE A SMALLER RP.
5:   SOLVE DRP. DENOTE THE SOLUTION AS  $\Delta \mathbf{y}$ .
6:   if DRP OBJECTIVE FUNCTION  $w_{OPT} = 0$  then
7:     OPTIMAL = "Yes"
8:     return  $y$ ;
9:   end if
10:  if  $\mathbf{y}^T \mathbf{a}_j \leq 0$  (FOR ALL  $j \notin J$ ) then
11:    INFEASIBLE = "Yes";
12:    return ;
13:  end if
14:  SET  $\theta = \min \frac{c_j - \mathbf{y}^T \mathbf{a}_j}{\mathbf{y}^T \mathbf{a}_j}$  FOR  $\mathbf{y}'^T \mathbf{a}_j > 0$ ,  $j \notin J$ .
15:  UPDATE  $\mathbf{y}$  AS  $\mathbf{y} = \mathbf{y} + \theta \mathbf{y}$ ;
16: end while

```

下面我们看一下原始对偶算法的优点。

1. 原始对偶算法肯定会结束。如果用一些防止退化的规则的话，肯定会结束的。(如何使用 BLAND 法则防止退化的 SLIDES 放到了网上)
2. 我们会看到无论是 RP 还是 DRP 都不显式的依赖于 c 。实际上那个 c 已经表达在那个 J 当中了。 J 就是我们得到的那些约束。

这就导致了原始对偶算法的一个很重要的优点。那就是：RP 经常是一个纯粹性的组合问题。在最短路径问题当中，RP 可以直接转化为一个组合问题—连通可达性问题。我们把一个可行解带到对偶问题后，约束域的约束分成了两部分。一部分约

束取等于号（我们看到的图中的红框）；另一部分取严格小于号（图中的蓝框），这一部分对应的 $X_1=0$ ，这意味着这些约束将不需要再考虑。问题的规模一下子大大缩小。我们只需要考虑红框的问题。原始对偶的优势就在于，每次都把问题缩小一块儿。我们现在比较一下对偶问题和 DRP 问题：

- 对偶问题：

$$\begin{aligned} \max \quad & b_1y_1 + b_2y_2 + \dots + b_my_m \\ s.t. \quad & a_{11}y_1 + a_{21}y_2 + \dots + a_{m1}y_m \leq c_1 \\ & a_{12}y_1 + a_{22}y_2 + \dots + a_{m2}y_m \leq c_2 \\ & \dots \\ & a_{1n}y_1 + a_{2n}y_2 + \dots + a_{mn}y_m \leq c_n \end{aligned}$$

- DRP：

$$\begin{aligned} \max \quad & w = b_1y_1 + b_2y_2 + \dots + b_my_m \\ s.t. \quad & a_{11}y_1 + a_{21}y_2 + \dots + a_{m1}y_m \leq 0 \\ & a_{12}y_1 + a_{22}y_2 + \dots + a_{m2}y_m \leq 0 \\ & \dots \\ & a_{1|\mathcal{J}|}y_1 + a_{2|\mathcal{J}|}y_2 + \dots + a_{m|\mathcal{J}|}y_m \leq 0 \\ & y_1, y_2, \dots, y_m \leq 1 \end{aligned}$$

非常好的形式，DRP 的目标函数和对偶问题一模一样。约束有三点不一样，第一个，约束不再是小于等于 c_i 了，而是小于等于 0。第二点，只需要管红框里的约束，蓝框里的不用管了。第三点，每个 y_i 都小于等于 1。我们可以总结下如果要用动态规划的算法去解决实际问题，需要有哪些要素、解决问题的关键是什么以及怎样描述并定义子问题。

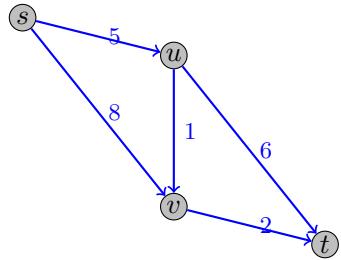
9.12.3 最短路径：Dijkstra's algorithm 基本上是原始对偶算法

我们再回顾一下 DIJKSTRA'S ALGORITHM，我们说它不适合第一节课学，因为他太精巧，太聪明了，而我们很难从中学到东西。现在我们对这个问题跑一下原始对偶。原始对偶是一个套路。假如要求一个原始问题 P，我们可以很机械的把它的对偶问题 D 写出来。然后我们把一个 Y 带进去，直接求出针对这个 Y 的 DRP。

如果 DRP 求出的解是 0, 那就是最优解, 如果大于 0, 则更新 Y, 不断重复。这么一个机械性的东西竟然就是 DIJKSTRA'S ALGORITHM。

最短路径问题

我们看一下这个最短路径问题, 四个城市:s,u,v,t。城市之间有路连接, 求最短路径。



最短路径问题的对偶以及简化

写出它的原始线性规划:

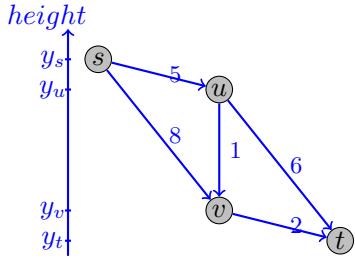
$$\begin{aligned}
 \min \quad & 5x_1 + 8x_2 + 1x_3 + 6x_4 + 2x_5 \\
 \text{s.t.} \quad & x_1 + x_2 - x_4 - x_5 = 1 \quad \text{VERTEX } s \\
 & -x_1 + x_3 + x_4 = -1 \quad \text{VERTEX } t \\
 & -x_2 - x_3 + x_4 + x_5 = 0 \quad \text{VERTEX } u \\
 & x_1, x_2, x_3, x_4, x_5 \geq 0 \\
 & x_1, x_2, x_3, x_4, x_5 \leq 1
 \end{aligned}$$

对于约束条件, 我们要求 x_i 应该取 0 或者 1。如果这样的话, 问题挺难的。对于这个问题, 我们可以松弛一下, 变成大于等于 0 小于等于 1. 由于全单模条件, 这两个条件的解是一样的。

写出原问题的对偶:

$$\begin{aligned}
 \max \quad & y_s - y_t \\
 \text{s.t.} \quad & y_s - y_u \leq 5 \quad x_1 : \text{EDGE } (s, u) \\
 & y_s - y_v \leq 8 \quad x_2 : \text{EDGE } (s, v) \\
 & y_u - y_v \leq 1 \quad x_3 : \text{EDGE } (u, v) \\
 & -y_t + y_u \leq 6 \quad x_4 : \text{EDGE } (u, t) \\
 & -y_t + y_v \leq 2 \quad x_5 : \text{EDGE } (v, t)
 \end{aligned}$$

原问题相当于对每个边设一个变量，对偶问题相当于对城市设置变量。 y_1 可以理解为城市 s 的海拔高度。优化目标为 $y_s - y_t$ ，是原问题的下边界。求它的最大值。



跑原始对偶算法，首先简化原始问题，将 y_t 固定为 0。问题变为：

$$\begin{aligned}
 & \max \quad y_s \\
 \text{s.t.} \quad & y_s - y_u \leq 5 \quad x_1 : \text{EDGE } (s, u) \\
 & y_s - y_v \leq 8 \quad x_2 : \text{EDGE } (s, v) \\
 & y_u - y_v \leq 1 \quad x_3 : \text{EDGE } (u, v) \\
 & y_u \leq 6 \quad x_4 : \text{EDGE } (u, t) \\
 & y_v \leq 2 \quad x_5 : \text{EDGE } (v, t)
 \end{aligned}$$

第一次迭代

设置一个初始可行解： $\mathbf{y}^T = (0, 0, 0)$ 。将其带入约束，判断哪些约束取等号，哪些取不等。根据互补松弛性，确定哪些 $x_i = 0$ 。

$$\begin{aligned}
 y_s - y_u &< 5 \Rightarrow x_1 = 0 \\
 y_s - y_v &< 8 \Rightarrow x_2 = 0 \\
 y_u - y_v &< 1 \Rightarrow x_3 = 0 \\
 y_u &< 6 \Rightarrow x_4 = 0 \\
 y_v &< 2 \Rightarrow x_5 = 0
 \end{aligned}$$

验证可知在 D 中， $J = \Phi$ ，即 $x_2, x_3, x_4, x_5 = 0$ 。

把当前的 RP 写出来:

$$\begin{aligned}
 \min \quad & s_1 + s_2 + s_3 \\
 \text{s.t.} \quad & s_1 + x_1 + x_2 = 1 \quad \text{NODE } s \\
 & s_2 - x_1 + x_3 + x_4 = 0 \quad \text{NODE } u \\
 & s_3 - x_2 - x_3 + x_5 = 0 \quad \text{NODE } v \\
 & s_1, s_2, s_3, \geq 0 \\
 & x_1, x_2, x_3, x_4, x_5 = 0
 \end{aligned}$$

蓝色的 x_i 是为了突出表示 $x_i=0$ 。

写出 DRP:

$$\begin{aligned}
 \max \quad & y_s \\
 \text{s.t.} \quad & y_s \leq 1 \\
 & y_u \leq 1 \\
 & y_v \leq 1
 \end{aligned}$$

如果原先的 $y(0,0,0)$ 是最优解的话, 对应的 x 一定满足 RP, Δy 满足 DRP.

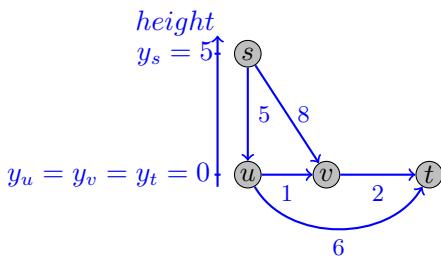
原始对偶算法应用到图论上, 常常是这样的: 写出的线性规划, 对应的 DRP 形式很特殊, 解能够很明显的看出来。

采用组合技术求解 DRP, 通过 DRP 求解, 可以知道当前的 y 不是最优。对于 DRP 的最优值, 找一个最优解 $\Delta y^T = (1, 0, 0)$ 。(最优解不唯一)

计算步长 $\theta: \theta = \min\{\frac{c_1 - y^T a_1}{y^T a_1}, \frac{c_2 - y^T a_2}{y^T a_2}\} = \min\{5, 8\} = 5$

更新 y : $y^T = y^T + \theta \Delta y^T = (5, 0, 0)$.

经过一次迭代更新以后, 城市之间的状态如图:



我们就拿这一步来看, 为什么说原始对偶算法就是 DIJKSTRA'S ALGORITHM。

从 DIJKSTRA'S ALGORITHM 的角度来看:

DRP 的最优解: $\Delta \mathbf{y}^T = (1, 0, 0)$, 对应着 DIJKSTRA'S ALGORITHM 滴墨水的起始位置, 也是被染的点集合 $S = \{s\}$ 。第一次的最优解对应染的第一个点。DRP 的实际目的找到墨水所要染的点。

步长 $\theta = \min\left\{\frac{c_1 - \mathbf{y}^T \mathbf{a}_1}{\mathbf{y}^T \mathbf{a}_1}, \frac{c_2 - \mathbf{y}^T \mathbf{a}_2}{\mathbf{y}^T \mathbf{a}_2}\right\} = \min\{5, 8\} = 5$: 对于现在墨水所染得点集合 S , 下一次墨水能染的最短距离。

第二次迭代

将新的可行解 $\mathbf{y}^T = (5, 0, 0)$ 带入, 检查约束:

$$\begin{aligned} y_s - y_u &= 5 \\ y_s - y_v &< 8 \Rightarrow x_2 = 0 \\ y_u - y_v &< 1 \Rightarrow x_3 = 0 \\ y_u - y_v &< 6 \Rightarrow x_4 = 0 \\ y_v &< 2 \Rightarrow x_5 = 0 \end{aligned}$$

可知在 D 中: $J = \{1\}$, 即 $x_2, x_3, x_4, x_5 = 0$.

对应的 RP:

$$\begin{array}{llllll} \min & s_1 + s_2 + s_3 & & & & \\ \text{s.t.} & s_1 + x_1 + x_2 & = 1 & \text{NODE } s \\ & s_2 - x_1 + x_3 + x_4 & = 0 & \text{NODE } u \\ & s_3 - x_2 - x_3 + x_5 & = 0 & \text{NODE } v \\ & s_1, s_2, s_3, & & & & \geq 0 \\ & x_1, x_2, x_3, x_4, x_5 & = 0 & & & \end{array}$$

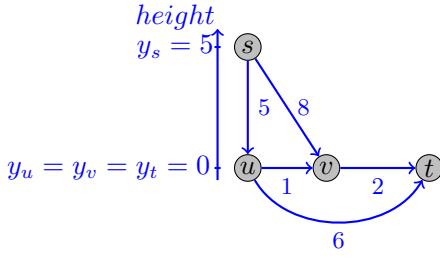
写出 DRP:

$$\begin{array}{ll} \max & y_s \\ \text{s.t.} & y_s \leq 1 \\ & y_u \leq 1 \\ & y_v \leq 1 \end{array}$$

采用组合技术求解 DRP, 通过 DRP 求解, 可以知道当前的 \mathbf{y} 不是最优。对于 DRP 的最优值, 找一个最优解 $\Delta \mathbf{y}^T = (1, 0, 0)$ 。(最优解不唯一) 步长 θ : $\theta = \min\left\{\frac{c_1 - \mathbf{y}^T \mathbf{a}_1}{\mathbf{y}^T \mathbf{a}_1}, \frac{c_2 - \mathbf{y}^T \mathbf{a}_2}{\mathbf{y}^T \mathbf{a}_2}\right\} = \min\{5, 8\} = 5$ 。

更新 \mathbf{y} : $\mathbf{y}^T = \mathbf{y}^T + \theta \Delta \mathbf{y}^T = (5, 0, 0)$.

经过第二次迭代更新以后, 城市之间的状态如图:



从 DIJKSTRA's ALGORITHM 的角度来看:

DRP 的最优解: $\Delta \mathbf{y}^T = (1, 1, 0)$, 对应着被染的点集合 $S = \{s, u\}$ 。事实上, DRP 的求解通过从 s 可达的节点中寻找确定。

步长 $\theta = \min\{\frac{\mathbf{c}_2 - \mathbf{y}^T \mathbf{a}_2}{\mathbf{y}^T \mathbf{a}_2}, \frac{\mathbf{c}_3 - \mathbf{y}^T \mathbf{a}_3}{\mathbf{y}^T \mathbf{a}_3}, \frac{\mathbf{c}_4 - \mathbf{y}^T \mathbf{a}_4}{\mathbf{y}^T \mathbf{a}_4}\} = \min\{3, 1, 6\} = 1$: 对于现在墨水所染得点集合 S , 下一次墨水能染的最短距离。

第三次迭代

将新的可行解 $\mathbf{y}^T = (6, 1, 0)$. 带入, 检查约束:

$$\begin{aligned}
 y_s - y_u &= 5 \\
 y_s - y_v &< 8 \Rightarrow x_2 = 0 \\
 y_u - y_v &= 1 \\
 y_u &< 6 \Rightarrow x_4 = 0 \\
 y_v &< 2 \Rightarrow x_5 = 0
 \end{aligned}$$

可知在 D 中: $J = \{1, 3\}$, 即 $x_2, x_4, x_5 = 0$.

对应的 RP:

$$\begin{array}{lllll}
 \min & s_1 & +s_2 & +s_3 & \\
 \text{s.t.} & s_1 & +x_1 & +x_2 & = 1 \quad \text{NODE } s \\
 & s_2 & -x_1 & +x_3 & +x_4 = 0 \quad \text{NODE } u \\
 & s_3 & -x_2 & -x_3 & +x_5 = 0 \quad \text{NODE } v \\
 & s_1, & s_2, & s_3, & \geq 0 \\
 & & x_2, & x_4, & x_5 = 0
 \end{array}$$

写出 DRP:

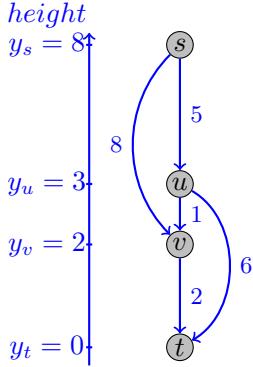
$$\begin{aligned} \max \quad & y_s \\ \text{s.t.} \quad & y_s - y_u \leq 0 \\ & y_u - y_v \leq 0 \\ & y_s, y_u, y_v \leq 1 \end{aligned}$$

采用组合技术求解 DRP, 通过 DRP 求解, 可以知道当前的 y 不是最优。对于 DRP 的最优值, 找一个最优解 $\Delta y^T = (1, 1, 1)$ 。(最优解不唯一) 步长 θ :

$$\theta = \min\left\{\frac{c_4 - y^T a_4}{y^T a_4}, \frac{c_5 - y^T a_5}{y^T a_5}\right\} = \min\{5, 2\} = 2.$$

更新 y : $y^T = y^T + \theta \Delta y^T = (8, 3, 2)$..

经过第三次迭代更新以后, 城市之间的状态如图:



从 DIJKSTRA'S ALGORITHM 的角度来看:

DRP 的最优解: $\Delta y^T = (1, 1, 1)$, 对应着被染的点集合 $S = \{s, u, v\}$ 。事实上, DRP 的求解通过从 s 可达的节点中寻找确定。

步长 $\theta = \min\left\{\frac{c_4 - y^T a_4}{y^T a_4}, \frac{c_5 - y^T a_5}{y^T a_5}\right\} = \min\{5, 2\} = 2$: 对于现在墨水所染得点集合 S , 下一次墨水能染的最短距离。

第四次迭代

将新的可行解 $y^T = (8, 3, 2)$. 带入, 检查约束:

$$\begin{aligned} y_s - y_u &= 5 \\ y_s - y_v &< 8 \Rightarrow x_2 = 0 \\ y_u - y_v &= 1 \\ y_u &< 6 \Rightarrow x_4 = 0 \\ y_v &= 2 \end{aligned}$$

可知在 D 中: $J = \{1, 3\}$, 即 $x_2, x_4, x_5 = 0$.

对应的 RP:

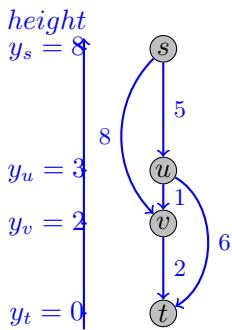
$$\begin{array}{llllll} \min & s_1 & +s_2 & +s_3 & & \\ \text{s.t.} & s_1 & & & & = 1 \quad \text{NODE } s \\ & s_2 & & & & = 0 \quad \text{NODE } u \\ & & s_3 & & & = 0 \quad \text{NODE } v \\ & & & -x_2 & -x_3 & +x_5 \\ & s_1, & s_2, & s_3, & & \geq 0 \\ & & & x_2, & & x_4 \\ & & & & & = 0 \end{array}$$

写出 DRP:

$$\begin{array}{llll} \max & y_s \\ \text{s.t.} & y_s - y_u & \leq 0 \\ & y_u - y_v & \leq 0 \\ & y_v & \leq 0 \\ & y_s, y_u, y_v & \leq 1 \end{array}$$

采用组合技术求解 $\Delta y^T = (0, 0, 0)$, 可知当前时刻 y 为最优解。

经过第四次迭代更新以后, 城市之间的状态如图:



从 DIJKSTRA's ALGORITHM 的角度来看:

DRP 的最优解: $\Delta y^T = (0, 0, 0)$, 表示能找到一个路径 PATH 从 s 到 t , 强迫 $y_s = 0$ 。这对应 DIJKSTRA's ALGORITHM 中那滴墨水把所有的点都染到了。

DIJKSTRA's ALGORITHM 的另一个直观解释为: 用一些绳子连着一些球, 拧起 s 点, 然后让 t 点在最下面, 求两者最短距离。

第十章 网络流算法

10.1 网络流：实际问题与算法发展脉络

10.1.1 概述

我们来讲网络流问题：

- MAXIMUMFLOW PROBLEM: FORD-FULKERSON ALGORITHM, MAXFLOW-MINCUT THEOREM;
- A DUALITY EXPLANATION OF FORD-FULKERSON ALGORITHM AND MAXFLOW-MINCUT THEOREM(实际上就是强对偶性);
- SCALING TECHNIQUE TO IMPROVE FORD-FULKERSON ALGORITHM (值得大家学习)；
- SOLVING THE DUAL PROBLEM: PUSH-RELABEL ALGORITHM;
- EXTENSIONS OF MAXIMUMFLOW PROBLEM: LOWER BOUND OF CAPACITY, MULTIPLE SOURCES & MULTIPLE SINKS, INDIRECT GRAPH;

10.1.2 网络流的简短历史

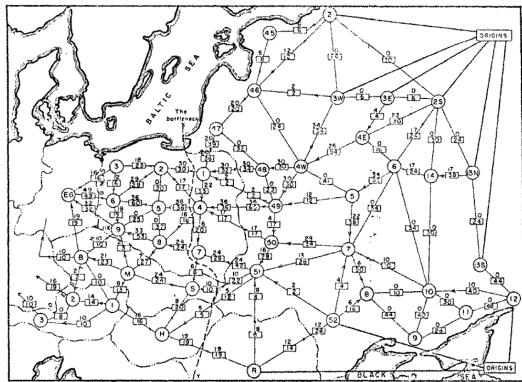


图 10.1: Soviet Railway network, 1955

.....1955 年，美国开始在想，如何轰炸铁路，来阻断苏联同社会主义国家之间的联系。.....

- “.... From Harris and Ross [1955]: Schematic diagram of the railway network of the Western Soviet Union and Eastern European countries, with a maximum flow of value 163,000 tons from Russia to Eastern Europe, and a cut of capacity 163,000 tons indicated as “The bottleneck””
- A recently declassified U.S. Air Force report indicates that the original motivation of minimum-cut problem and Ford-Fulkerson algorithm is to disrupt rail transportation the Soviet Union [A. Shrijver, 2002].(2002 年的解密文档)

1955 年提出的问题，到了 1956 年，FORD AND FULKERSON 就给了一个算法。从这个事情中又能够体现着出这件事情：原始问题的实际问题是什么，数学的抽象 - 建模是第二部，第三步是算法设计。

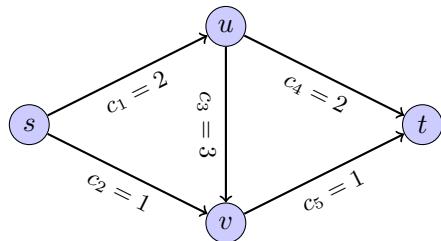
Year	Developers	Time-complexity
1956	Ford and Fulkerson	$O(mC)$ and $O(m^2 \log C)$
1972	Edmonds and Karp	$O(m^2 n)$
1970	Dinitz	$O(n^2 m)$
1974	Karzanov	$O(n^3)$
1983	Sleator and Tarjan	$O(nm \log n)$
1988	Goldberg and Tarjan	$O(n^2 m \log(\frac{n^2}{m}))$
2012	Orlin	$O(nm)$

10.1.3 最大流问题

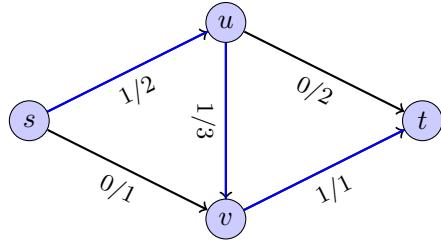
问题描述

- 输入: 一个有向图 $G = \langle V, E \rangle$. 每个顶点 v 表示一个城市, 每条边 e 表示城市之间的路, 每条边 e 有个容量限制 C_e . 两个特殊的点: 起点 **source** s 和终点 **sink** t ;
- 输出: 对于每一条边 $e = (u, v)$, 分给一条流 $f(u, v)$ 最终使得 $\sum_{u, (s, u) \in E} f(s, u)$ 最大.

具体举例如下:



目标: 从 s 点运尽量多的货物到目的地 t 。



定义：flow

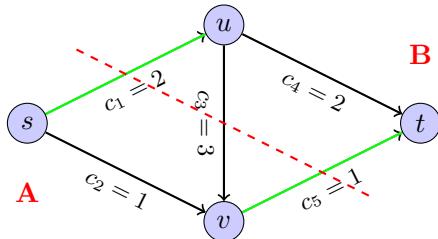
$f : E \rightarrow R^+$ 是一个 $s - t$ flow 如果：

1. (CAPACITY CONSTRAINTS): $0 \leq f(e) \leq C_e$ 对于全部的 e 成立；
2. (CONSERVATION CONSTRAINTS): 对于任何中间节点 $v \in V - \{s, t\}$, $f^{in}(v) = f^{out}(v)$, 其中 $f^{in}(v) = \sum_{e \text{ INTO } v} f(e)$ 并且 $f^{out}(v) = \sum_{e \text{ OUT OF } v} f(e)$. (直观: 输入 = 输出对于任何节点.)

flow 的值 f 被定义为 $V(f) = f^{out}(s)$.

定义： $s - t$ cut

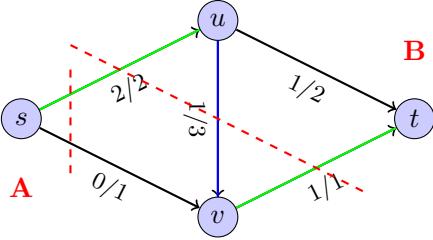
一个 $s - t$ cut 是一个划分 V 的 (A, B) 从而使得 $s \in A$ AND $t \in B$. 割 cut (A, B) 的 capacity 被定义为 $C(A, B) = \sum_{e \text{ FROM } A \text{ TO } B} C(e)$.



$C(A, B) = 3$, 只计从 A 到 B 的, 不计从 B 到 A 的

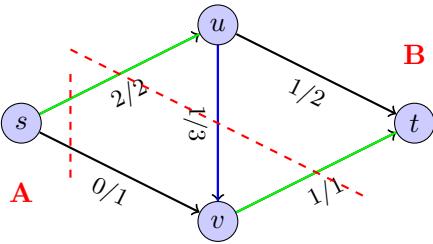
定义：流值引理

给定一个流 f . 对于 任何 $s - t$ 割 $CUT (A, B)$, 通过这个割的流是一个常量 $V(f)$. 通常, $V(f) = f^{out}(A) - f^{in}(A)$.



$$V(f) = 2 + 0 = 2$$

$$f^{out}(A) - f^{in}(A) = 2 + 1 - 1 = V(f)$$



引理证明

- 我们有: $0 = f^{out}(v) - f^{in}(v)$ 对于任何的 $v \neq s$ 和 $v \neq t$. 这是条件。
- 因此我们有:

$$\begin{aligned} V(f) &= f^{out}(s) - f^{in}(s) && // \text{提示: } f^{in}(s) = 0; \\ &= \sum_{v \in A} (f^{out}(v) - f^{in}(v)) \\ &= (\sum_{\text{E FROM A TO B}} f(e) + \sum_{\text{E FROM A TO A}} f(e)) \\ &\quad - (\sum_{\text{E FROM B TO A}} f(e) + \sum_{\text{E FROM A TO A}} f(e)) \\ &= f^{out}(A) - f^{in}(A) \end{aligned}$$

以上是一些定义和引理。现在回到原问题。依据我们现在所学的知识，采用什么方法使得流最大。贪心可以，但肯定不太好，分支特别多，不是一个太好的选择。线性规划肯定可以，是万能的。首先这个问题不好分，是图的问题，不好规约。下面看一下 1956 年的 FORD-FULKERSON ALGORITHM。

10.1.4 Ford-Fulkerson algorithm

Lester Randolph Ford Jr. 和 Delbert Ray Fulkerson



图 10.2: Lester Randolph Ford Jr. and Delbert Ray Fulkerson

尝试 1: 动态规划技术

- 动态规划似乎不太好用.
- 实际上, 当前不存在一个最大流 问题可以真正的被看做是属于动态规划问题.
- 我们知道最大流 问题是在 **P** 中因为它可以写成动态规划 (见 LECTURE 8).
- 然而, 网络结构存在它自己的属性使得能够有一个更有效的算法, 非正式的称作 **network simplex**, 等等.

尝试 2: 改进 策略

问题不好分, 我们尝试改进的策略。

$\text{IMPROVEMENT}(f)$

```
1:  $x = x_0$ ; //STARTING FROM AN INITIAL SOLUTION;  
2: while TRUE do  
3:    $x = \text{IMPROVE}(x)$ ; //MOVE ONE STEP TOWARDS OPTIMUM;  
4:   if STOPPING( $x, f$ ) then
```

```

5:      BREAK;
6:  end if
7: end while
8: return x;

```

迭代框架的三个关键问题

三个问题:

1. 如何构建一个初始解?

- 对于最大流 问题, 一个 0-流可以通过通过设置 $f(e) = 0$ 得到, 对于任意 e .
- 很容验证 H CONSERVATION 和 CAPACITY 约束都被满足, 对于 0-流来说.

2. 如何改进这个解决办法?

3. 何时停止?

先看一个随便一想就能想到的办法。

- 假定 p 是一个在网络 G 中的简单 $s - t$ PATH.

```

1: INITIALIZE  $f(e) = 0$  FOR ALL  $e$ .
2: while THERE IS AN  $s - t$  PATH IN GRAPH  $G$  do
3:   arbitrarily CHOOSE AN  $s - t$  PATH  $p$  IN  $G$ ;
4:    $f = \text{AUGMENT}(p, f)$ ;
5: end while
6: return  $f$ ;

```

我们定义 $bottleneck(p, f)$ 作为 PATH p 上边的最小 CAPACITY.

$\text{AUGMENT}(p, f)$:

```

1: LET  $b = bottleneck(p, f)$ ;
2: for EACH EDGE  $e = (u, v) \in P$  do
3:   if  $(u, v)$  IS A FORWARD EDGE then

```

```

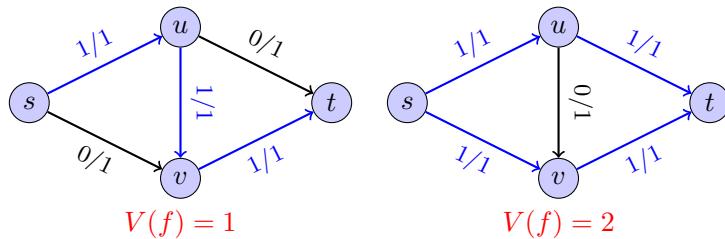
4:     INCREASE  $f(u, v)$  BY  $b$ ;
5: else
6:     DECREASE  $f(u, v)$  BY  $b$ ;
7: end if
8: end for

```

这是一个失败的算法。

为什么会失败呢？

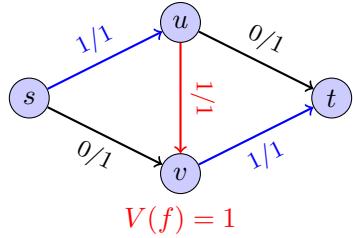
- 我们从 0-流开始。为了减小 f 的值，我们找到一条 $s - t$ PATH，比如说 $p = s \rightarrow u \rightarrow v$, 来运更多的商品
- 这三条边上的流可以被增加 1 同时满足 CONSERVATION 和 CAPACITY 限制。
- 然而，我们不能发现一条 $s - t$ PATH 存在于 G 中，使得 f 增加更多（左半部分）即使最大流是 2 (右半部分)。



Ford-Fulkerson algorithm: “**复原 undo**” 功能

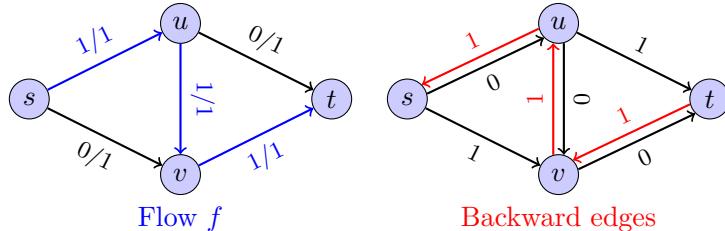
关键性观察：

- 当构建一个流 f 时，调度商品可能会犯错，即，有些边不应该用来运输商品。举个例子，图中的边 $u \rightarrow v$ 就不应该使用。



- 为了改进流 f , 我们应该使用一些手段 **更正在写错误**, 即“复原 UNDO”边上做过的运输任务。
- 如何实现”UNDO”功能呢?
- 增加反向边!**
- 假定我们增加一条 **反向** 边 $v \rightarrow u$ 到原始图. 接着我们可以更正这次运输, 通过退回从 v 到 u 的商品.

加了退货边的图就叫剩余图 (RESIDUAL GRAPH)。



剩余图 Residual Graph

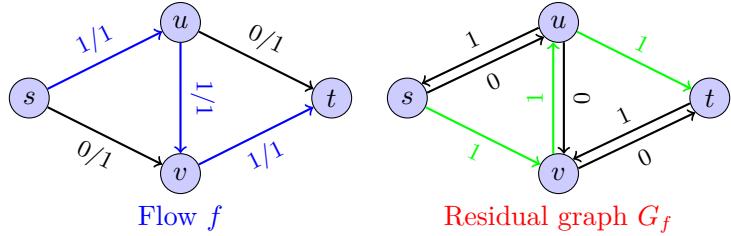
给定一个有向图 $G = < V, E >$, 有一个流 f , 我们定义 **剩余图 residual graph** $G_f = < V, E' >$. 对于任何一条边 $e = (u, v) \in E$, 按照下面的要求将两条边加入到 E' :

- (**正向边** (u, v) 标注剩余的运输容量):

如果 $f(e) < C(e)$, 添加一条边 $e = (u, v)$ 标注运输容量 $C(e) = C(e) - f(e)$.

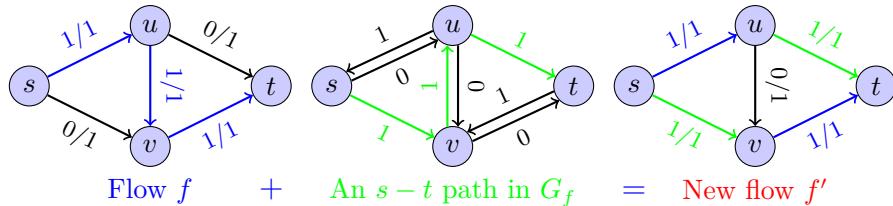
- (**反向边** (v, u) 标注回退容量):

如果 $f(e) > 0$, 添加一条边 $e' = (v, u)$ 标注回退容量 $C(e') = f(e)$.



提示：路径 PATH 中包含反向边 (v, u) 。

沿着路径增广流：从 f 到 f'



注释：

- 通过使用反向边 $v \rightarrow u$, 原始的从 u 到 v 的运输被退回.
- 更具体的, 第一次商品运输流 f 将会改变它的路径 (从 $s \rightarrow u \rightarrow v \rightarrow t$ 到 $s \rightarrow u \rightarrow t$), 当第二次使用路 $s \rightarrow v \rightarrow t$ 的时候.

10.1.5 求解最小割问题的算法的发展史

年	提出者	时间复杂度
1956	Ford and Fulkerson	$O(mC)$ and $O(m^2 \log C)$
1972	Edmonds and Karp	$O(m^2 n)$
1970	Dinitz	$O(n^2 m)$
1974	Karzanov	$O(n^3)$
1983	Sleator and Tarjan	$O(nm \log n)$
1988	Goldberg and Tarjan	$O(n^2 m \log(\frac{n^2}{m}))$
2012	Orlin	$O(nm)$

10.1.6 最大流问题

问题描述: 有向图 $G = \langle V, E \rangle$, 边 e 上有大小为 C_e 的容量限制, G 中有两个特殊的点: 起点 s 和目的地 t 。现给每条边 $e = (u, v)$ 指定流值 $f(u, v)$, 问如何指定可使得总的流值 $\sum_{u, (s, u) \in E} f(s, u)$ 最大。例如下图 (图 1.1):

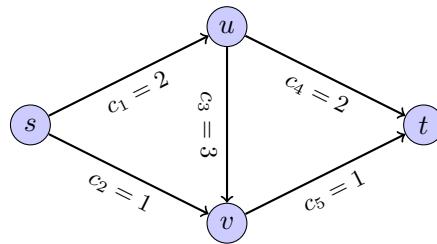


图 10.3: 图 1.1 求解 $s \rightarrow t$ 可运输的最多货物

目标: 从起点 s 运输尽可能多的货物到目的地 t 。

10.2 Ford-Fulkerson 算法 [1956]



图 10.4: Lester Randolph Ford Jr. and Delbert Ray Fulkerson

10.2.1 动态规划求解

遗憾的是, 该问题并不存在动态规划的多项式时间算法。但在第 8 节中也可以看出, 该问题是 P 类的, 因为可以用线性规划求解。

10.2.2 Improvement 策略

(1) IMPROVEMENT 策略大致思想

先确定一个初始可行解，然后再改进，直到不能改进。该问题用 IMPROVEMENT 策略实现的伪代码如下：

IMPROVEMENT(f)

```

1:  $x = x_0$ ; //STARTING FROM AN INITIAL SOLUTION;
2: while TRUE do
3:    $x = \text{IMPROVE}(x)$ ; //MOVE ONE STEP TOWARDS OPTIMUM;
4:   if STOPPING( $x, f$ ) then
5:     BREAK;
6:   end if
7: end while
8: return  $x$ ;

```

(2) 算法需要考虑的 3 个关键因素：

- 如何构造初始可行解？一个很直接的方法是初始化为 0 流，每条边均有 $f(e) = 0$
- 如何对接进行改进
- 何时需要停止？

(3) 一个失败的尝试

对于上节课所讲的一个笨拙的改进算法，随机选择可行流并在图上进行改进所得到的流，可能最终并不能终止于最优解。例如如下左图（图 2.2）的可行流，该可行流无法通过增加流值再得到改进，但该可行流并不是最大流，因为右图（图 2.3）是一个流值为 2 的可行流。

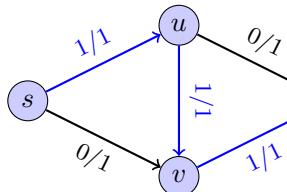


图 2.2 $V(f) = 1$

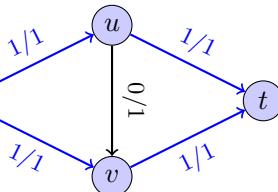


图 2.3 $V(f) = 2$

(4) FORD-FULKERSON 算法引入反向边，让运出去的货物还可以有机会退回。将加入了退货边的图称为剩余图，构造剩余图的大致思路为，若 $u \rightarrow v$ 运了 f 吨货物，且容量限制为 $C(u, v)$ ，则剩余图中 u 到 v 的正向弧 $u \rightarrow v$ 权值为 $C(u, v) - f$ ，表示最多还可再运 $C(u, v) - f$ 吨货，并构造权值为 f 的反向弧，表示最多可退大小为 f 的货物。例如，下面两个图分别代表网络的流图（图 2.4）及剩余图（图 2.5）。

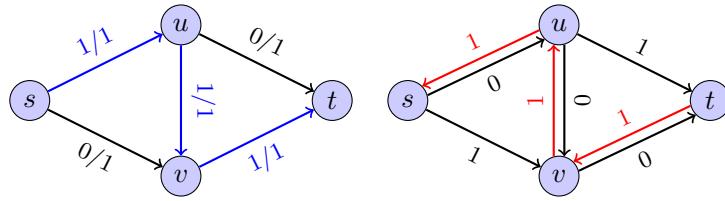
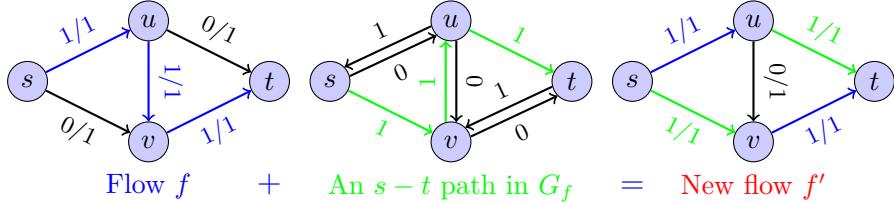
图 2.4 Flow f

图 2.5 Backward edges

FORD-FULKERSON 算法的实现过程可以简单地用如下所示的图来刻画：



简单来说，就是原始流 + 剩余图中的可行路 = 原始流的一个改进。

令 p 表示剩余图 G_f 中的一条简单路径，称为增广路径。并定义 $bottleneck(p, f)$ 为路径 p 上的最小容量边。则 FORD-FULKERSON 算法可以描述为：

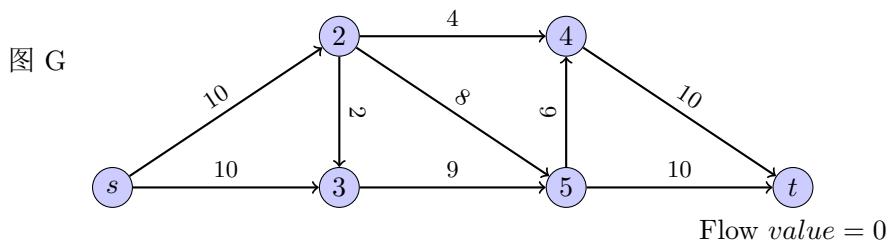
FORD-FULKERSON ALGORITHM

- 1: INITIALIZE $f(e) = 0$ FOR ALL e .
- 2: **while** THERE IS AN $s - t$ PATH IN RESIDUAL GRAPH G_f **do**
- 3: **arbitrarily** CHOOSE AN $s - t$ PATH p IN G_f ;
- 4: $f = AUGMENT(p, f)$;
- 5: **end while**
- 6: **return** f ;

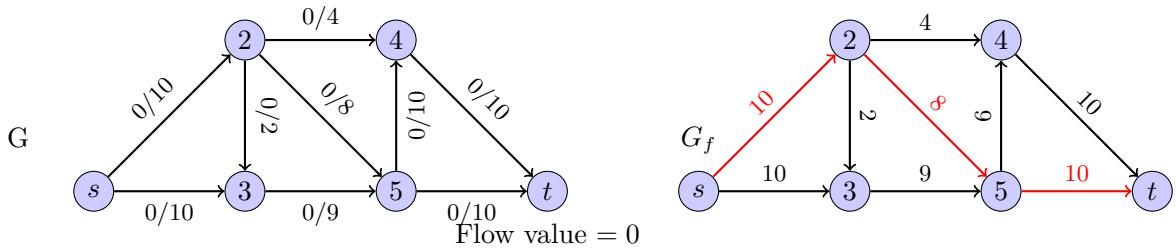
与上节课所讲的随机行走的算法的唯一区别是剩余图有起点到终点的路而不是原网络图有起点到终点的路径。但这一小小的修改却可以使算法最终终止于最优解（网络的最大流）。

下面是一个 FORD-FULKERSON 算法的实现。

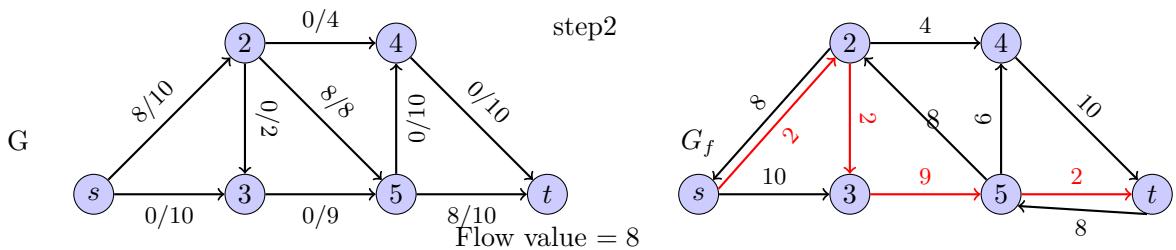
Ford-Fulkerson 算法实例

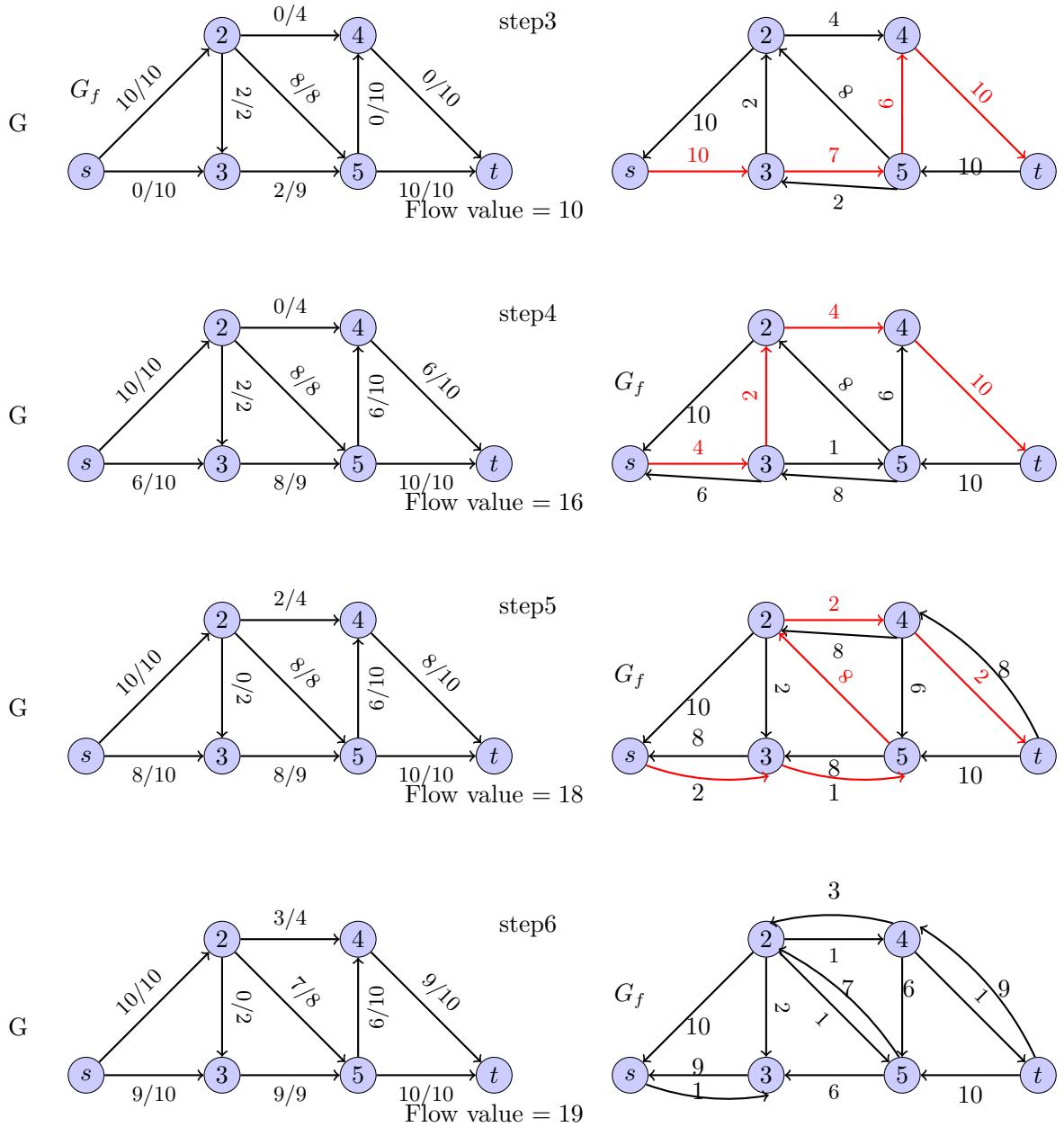


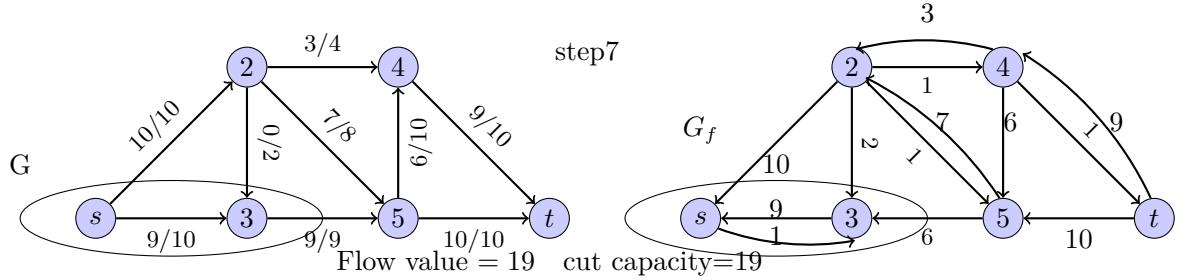
step1



step2







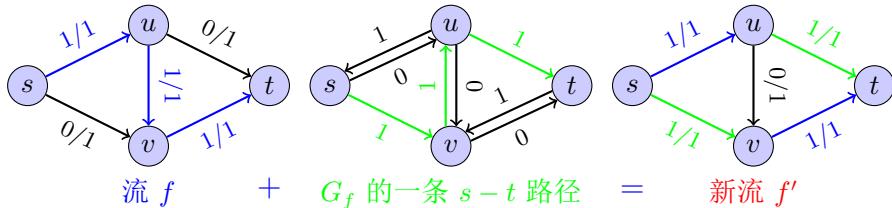
上图中 STEP7 中的流即为最大流, G_f 中阴影部分无可到达外面的有向边, 称阴影部分与外面所形成的割为最小割。

10.2.3 正确性证明及时间复杂性分析

增广操作将产生一个新的流

定理 1. 操作 $f' = AUGMENT(p, f)$ 将生成 G 的一个新流 f' 。

如下图所示:



证明. • 检查 f' 是否满足容量限制 (capacity constraints) $0 \leq f'(e) \leq C(e)$, 对于路径 p 上的边 $e = (u, v)$, 可以分为以下两种类型

(a) (u, v) 是前向边 ($(u, v) \in E$):

$$0 \leq f(e) \leq f'(e) = f(e) + bottleneck(p, f) \leq f(e) + (C(e) - f(e)) \leq C(e)$$

(b) (u, v) 是反向边 ($(v, u) \in E$):

$$C(e) \geq f(e) \geq f'(e) = f(e) - bottleneck(p, f) \geq f(e) - f(e) = 0$$

• 检查储存限制 (conservation constraints)(流入 = 流出)

对 f' 中的每个中间节点 v (除去起点 s , 终点 t), 流入 v 的流的总和等于流出 v 的总和。

□

单调递增

定理 2. (单调递增) $V(f') > V(f)$

证明. $V(f') = V(f) + bottleneck(p, f) > V(f)$ □

一个平凡的上界

定理 3. $C = \sum_{e \text{ out of } s} C(e)$ 是 $V(f)$ 的一个上界。

证明. 显然, $V(f) \leq f^{out}(s) = C$ 。 □

增广次数

定理 4. 假定所有的边的容量值都是整数, 则执行 Ford-Fulkerson 算法时, 每次迭代的流值及容量值都是整数。因此, $bottleneck(p, f) \geq 1$, 且循环至多进行 C 次。

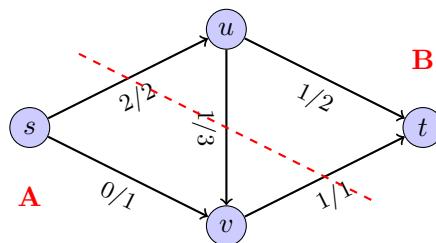
证明. 在假定容量值为整数的前提下, 循环的每步均有 $bottleneck(p, k) \geq 1$, 因此, $V(f') \geq V(f) + 1$, 则循环至多进行 C 次。

时间复杂性为 $O(mC)$. (C 次循环, 每次循环需要使用 DFS 或 BFS 寻找一条 $s - t$ 的路径, 花费时间为 $O(m + n)$) □

一个更精确的上界

定理 5 (精确的上界). 给定流 f , 对于任意 $s - t$ 割 $cut(A, B)$, 有 $V(f) \leq C(A, B)$ 。

下图给出了一个例子:



$$V(f) = 2 \leq C(A, B) = 3$$

证明.

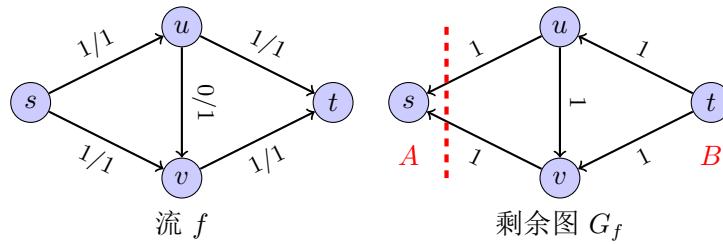
$$\begin{aligned} V(f) &= f^{out}(A) - f^{in}(A) \quad (\text{由流值引理}) \\ &\leq f^{out}(A) \quad (\text{由 } f^{in}(A) \geq 0) \\ &= \sum_{e \in A \rightarrow B} f(e) \\ &\leq \sum_{e \in A \rightarrow B} C(e) \quad (\text{由 } f(e) \leq C(e)) \\ &= C(A, B) \end{aligned}$$

□

正确性证明

定理 6. Ford-Fulkerson 算法终止于最大流 f , 也即为最小割 $cut(A, B)$ 。

如图所示:



证明. • 当剩余图 G_f 中无 $s - t$ 的路径时, FORD-FULKERSON 算法终止

- 令 A 为 G_f 中从 s 可达的顶点集, 并令 $B = V - A$, 则 (A, B) 形成了一个 $s - t$ 割 ($A \neq \emptyset, B \neq \emptyset$)。
- 位于割 $cut(A, B)$ 之间的边可分为以下两种类型:
 1. $u \in A, v \in B$: 则有 $f(e) = C(e)$ 。否则, 由 $(u, v) \in G_f$ 可得 $v \in A$,
 2. $u \in B, v \in A$: 则有 $f(e) = 0$ 。否则, 由 $(v, u) \in G_f$ 可得 $u \in A$

- 于是有最大流等于最小割

$$\begin{aligned}
 V(f) &= f^{out}(A) - f^{in}(A) \\
 &= f^{out}(A) \quad (\text{由 } f^{in}(A) = 0) \\
 &= \sum_{e \in A \rightarrow B} f(e) \\
 &= \sum_{e \in A \rightarrow B} C(e) \quad (\text{由 } f(e) = C(e)) \\
 &= C(A, B)
 \end{aligned}$$

□

10.2.4 FORD-FULKERSON 算法的缺点

整数约束

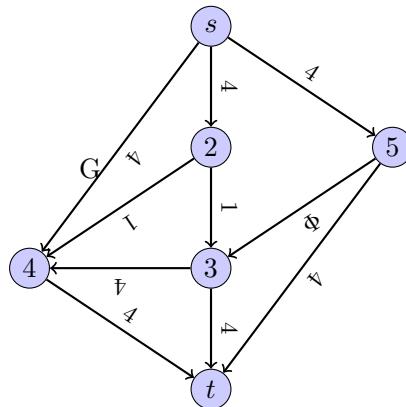
容量的整数限制是该算法不可或缺的条件，因为这可使得边的瓶颈 (BOTTLE-NECK) 增量每次至少增加 1。

但当容量限制为无理数 (有理数总可以通过乘以某个数转化为整数)，该算法有可能会处于无限循环中，而无法终止于最大流。

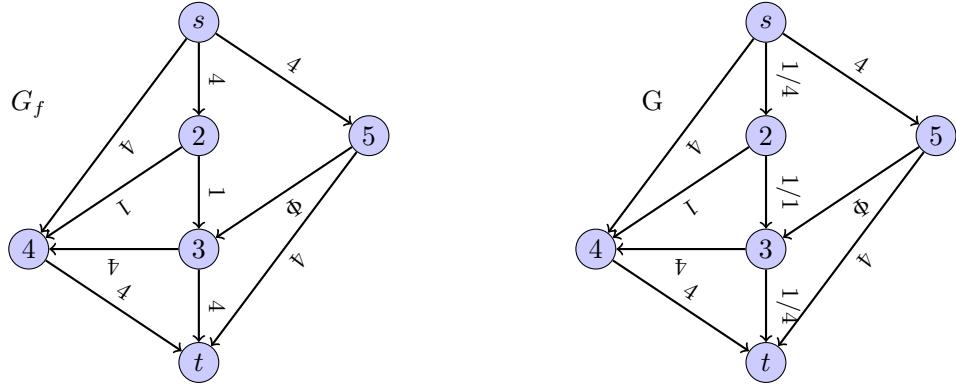
下面的例子显示了当容量约束条件存在无理数时，会出现无限循环而无法停止的结果：

其中， $1 - \Phi = \Phi^2$

最大流实例



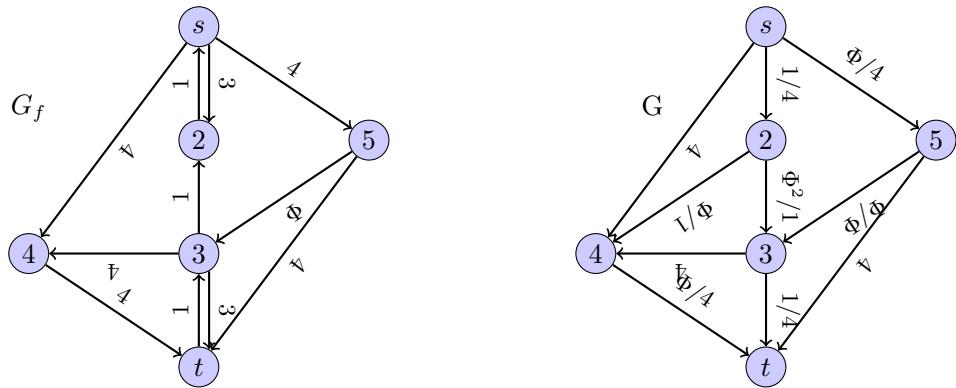
Step 1



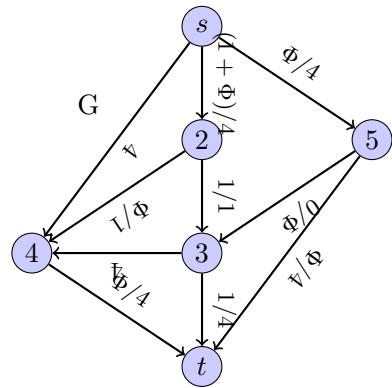
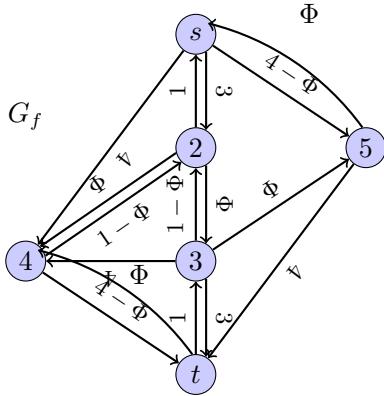
流值 = 1

(1)

Step 2

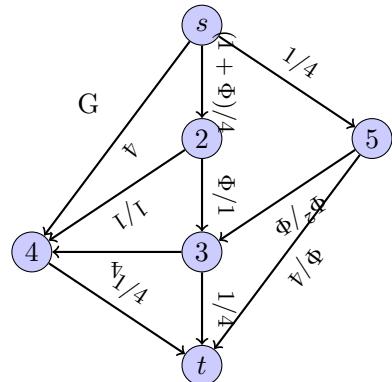
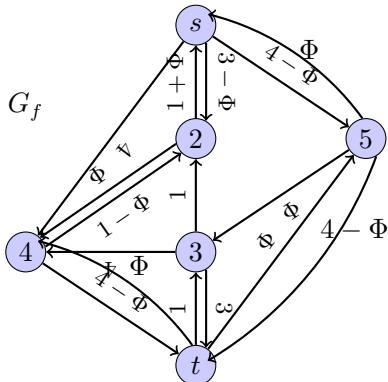
流值 = $1 + \Phi$

Step 3



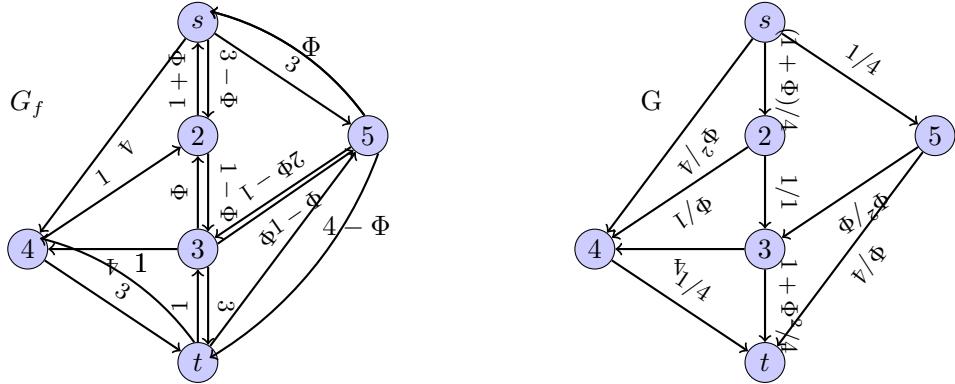
$$\text{流值} = 1 + \Phi + \Phi$$

Step 4



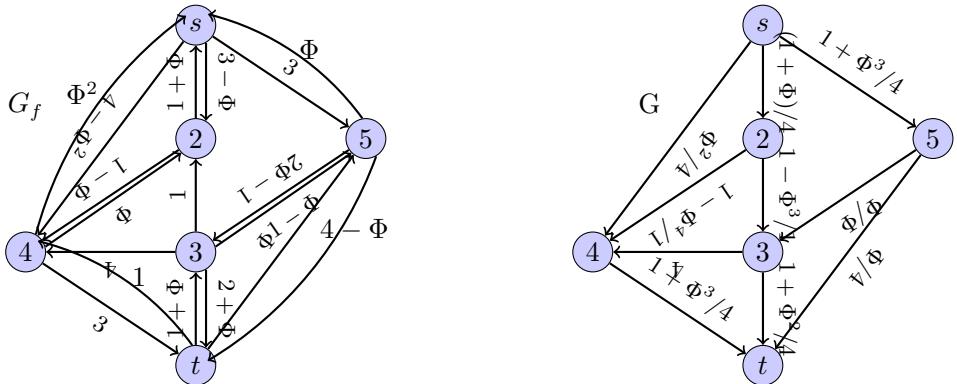
$$\text{流值} = 1 + \Phi + \Phi + \Phi^2$$

Step 5



$$\text{流值} = 1 + \Phi + \Phi + \Phi^2 + \Phi^3$$

Step 6



$$\text{流值} = 1 + \Phi + \Phi + \Phi^2 + \Phi^3 + \Phi^4$$

```

1: if FLOW(2, 4) == 1 then
2:   CHOOSE STEP 5 AS THE PATH.;
3: else
4:   if FLOW (5, 3) == Φ then
5:     CHOOSE STEP 3 AS THE PATH.;
6:   end if

```

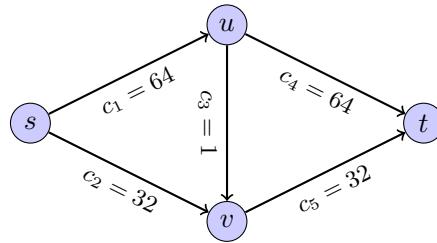
```

7: if FLOW (2,3) == 1 then
8:   CHOOSE STEP 4(6) AS THE PATH. ;
9: end if
10: end if

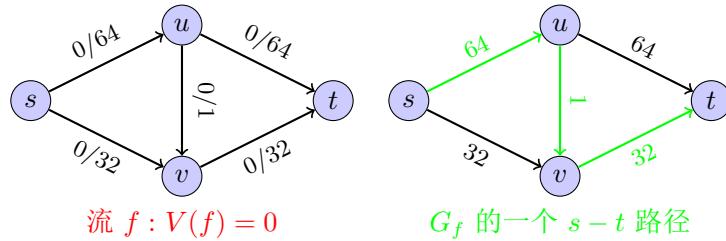
```

一个时间复杂度很高的实例

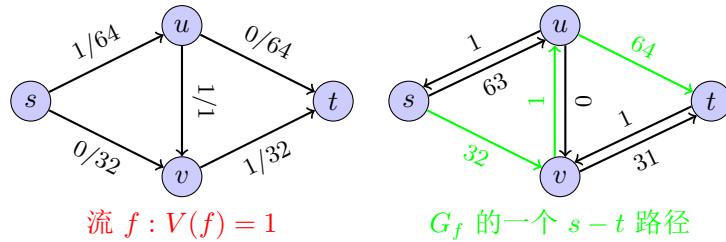
由于 FORD-FULKERSON 算法并未指定如何选取路径，算法运行时间依赖于每次所选的路径，比如下面的例子，当选取的增广路径不太好时，运行时间就会大大的增大。



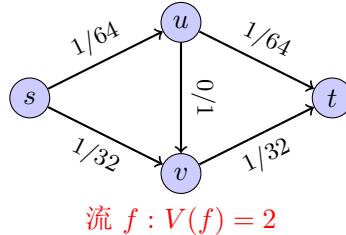
FORD-FULKERSON 算法: STEP 1



FORD-FULKERSON 算法: STEP 2



FORD-FULKERSON 算法：STEP 3



经过两次迭代后，该问题类似于原问题，只需修改原问题的容量限制，使得每条边的容量限制减 1 即可。

因此，利用 FORD-FULKERSON 算法，经过 $32 + 32 + 1$ 次迭代就可以得到最大流。但实际上该问题若选取合适的扩展路，只需要 $2 + 1$ 次迭代即可。

10.2.5 FORD-FULKERSON 算法的改进思想

由于 FORD-FULKERSON 算法并未确定如何选择增广路径，故可能会使得每次的瓶颈容量值选取的很小。

改进思路

(1) 尽量走大路

每次选择具有最大瓶颈容量的增广路径（实际应用较少）。

SCALING 技术：一种可以寻找增广路径的有效方法，每次可以得到大的改进。

(2) 走最短路径

EDMONDS-KARP：在 BFS 树中选择最短 s-t 路径。

DINITZ 算法：在分层网络中寻找一条路径，并执行

10.3 改进策略一：Scaling 技术

问题：如何选取一个大的增广路径？ $bottleneck(p, f)$ 越大，迭代次数越少。

- 1) 可以通过二进制搜索，或者以 $O(m + n \log n)$ 时间对 DIJKSTRA 算法稍加修改得到最大的 $bottleneck(p, f)$ 。但从某种程度来说（速度很慢），这种策略是不可行的。

b) 一种朴素的思想：放松条件，将“最大”放松为“足够大”。

借助该思想，可以建立 $bottleneck(p, f)$ 的一个下界 Δ ：删除小边。即：删除容量小于 Δ 的边，得到的图记为 $G_f(\Delta)$ 。

10.3.1 Scaling FORD-FULKERSON 算法

```

1: INITIALIZE  $f(e) = 0$  FOR ALL  $e$ .
2: LET  $\Delta = C$ ;
3: while  $\Delta \geq 1$  do
4:   while THERE IS AN  $s - t$  PATH IN  $G_f(\Delta)$  do
5:     CHOOSE AN  $s - t$  PATH  $p$ ;
6:      $f = AUGMENT(p, f)$ ;
7:   end while
8:    $\Delta = \Delta/2$ ;
9: end while
10: return  $f$ ;
```

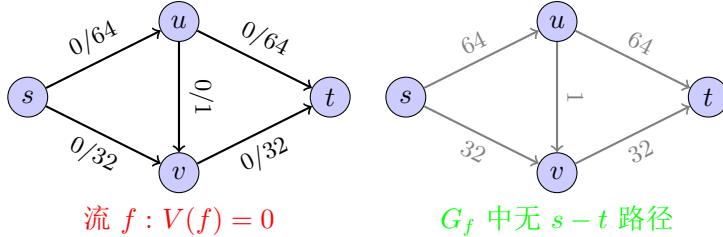
除了 STEP2,3,4,8 外其余与 FORD-FULKERSON 算法步骤相同。

由于在 G_f 中删去了小于 Δ 的边，因此每次增流的规模都很大。

Δ 最终必须减为 1，这样才能保证剩余图中无反向边（不漏掉容量较小的边）。

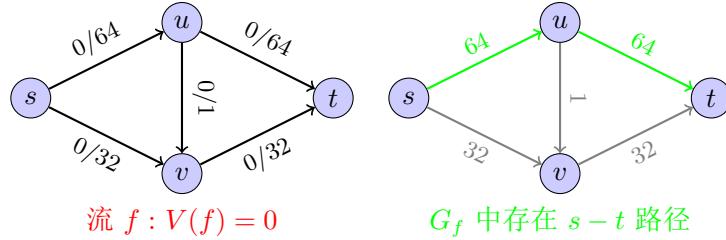
用该改进算法求上述实例：

实例：STEP 1



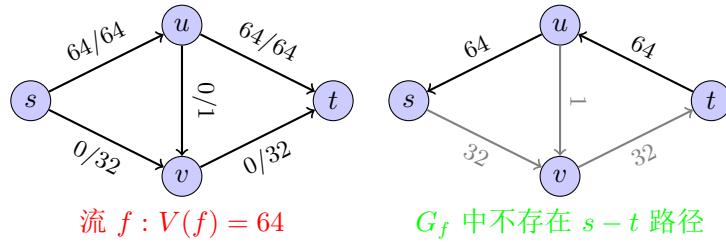
构造 0 流， $\Delta = 96$ ，由于 $G_f(\Delta)$ 图中无 $s - t$ 的路径，令 $\Delta = \Delta/2 = 48$ 。

实例：STEP 2



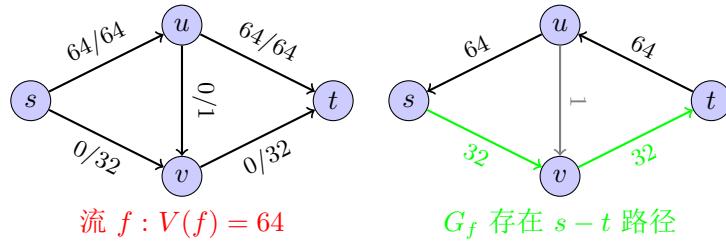
0 流, $\Delta = 48$, 存在 $s - t$ 的路径 ($s-U-T$), 执行增广操作。

实例: STEP 3



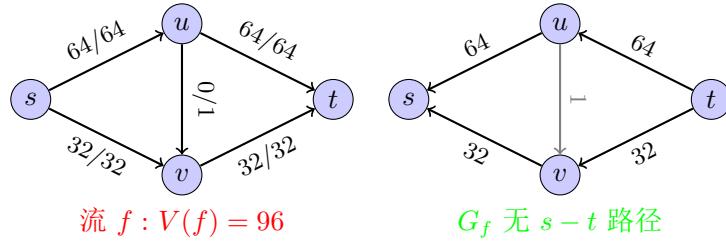
流: $V(f) = 64$, $\Delta = 48$, 不存在 $s - t$ 路径, 令 $\Delta = \Delta/2 = 24$ 。

实例: STEP 4



流: 64, $\Delta = 224$, 存在 $s - t$ 路径: $s - v - t$, 执行增广操作。

实例: STEP 5



流: 96, 已得到最大流。 $\Delta = 24$, 不存在 $s - t$ 路径。

10.3.2 Scaling 技术的时间复杂度

SCALINGFF 的时间复杂度为 $O(m^2 \log C)$, 因为算法依赖于 C 的长度, 因此该算法并不是多项式算法。

定理 7 (外循环次数). *while* 迭代次数至多进行 $1 + \log_2 C$ 。

定理 8 (内循环次数). *Scaling* 阶段, 增广操作的次数至多进行 $2m$ 次。

证明. 1) 令 f 表示 Δ Scaling 阶段结束时的流, f^* 表示最大流。则有 $V(f) \geq V(f^*) - m\Delta$ 。

2) 在下个 $\frac{\Delta}{2}$ Scaling 阶段, 每次增广, $V(f)$ 至少增加 $\frac{\Delta}{2}$ 。

则在 $\frac{\Delta}{2}$ Scaling 至多进行 $2m$ 次增广。

□

下面来证明 $V(f) \geq V(f^*) - m\Delta$ 。

证明. 令 A 表示剩余图中从 s 可到达的顶点集, $B = V - A$, 则 (A, B) 构成了最小割。

考察 $e = (u, v) \in E$:

1) $u \in A, v \in B$: 则有 $f(e) \geq C(e) - \Delta$ 。否则, A 应该包含 v , 若 G_f 中正向边的流值 $\geq \Delta$, $(u, v) \in G_f(\Delta)$

2) $v \in A, u \in B$: 则有 $f(e) \leq \Delta$ 。否则, A 应该包含 v , 若 G_f 中反向边的流值大于 Δ , $(u, v) \in G_f(\Delta)$ 。

因此:

$$\begin{aligned} V(f) &= \sum_{e \in A \rightarrow B} f(e) - \sum_{e \in B \rightarrow A} f(e) \\ &\geq \sum_{e \in A \rightarrow B} (C(e) - \Delta) - \sum_{e \in B \rightarrow A} \Delta \\ &\geq \sum_{e \in A \rightarrow B} C(e) - m\Delta \\ &= C(A, B) - m\Delta \\ &\geq V(f^*) - m\Delta \end{aligned}$$

□

SCALING 技巧对以后的发展很有用，后面在近似算法的背包问题也会提到。

10.4 改进策略二：EDMONDS-KARP $O(m^2n)$ 算法

10.4.1 算法的创造者



图 10.5: Jack Edmonds, and Richard Karp

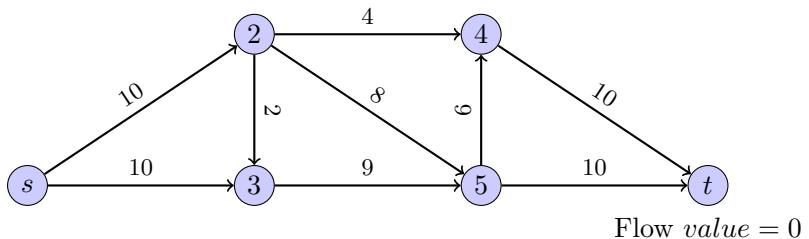
10.4.2 EDMONDS-KARP 算法

- 1: INITIALIZE $f(e) = 0$ FOR ALL e .
- 2: **while** THERE IS A $s - t$ PATH IN G_f **do**
- 3: CHOOSE **the shortest $s - t$** PATH p IN G_f USING BFS ;
- 4: $f = \text{AUGMENT}(p, f)$;
- 5: **end while**
- 6: **return** f ;

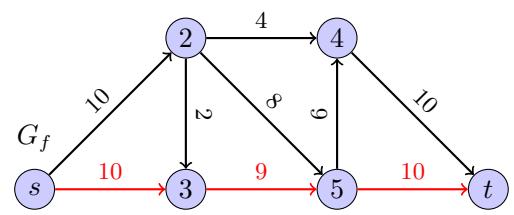
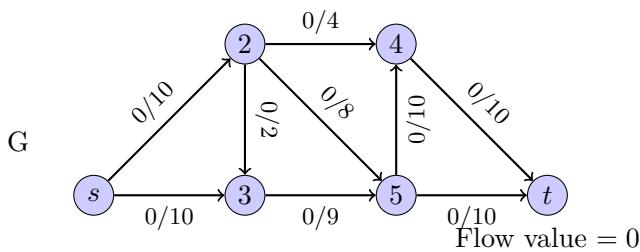
下面是该算法的一个运行实例：

Edmonds-Karp 算法实例

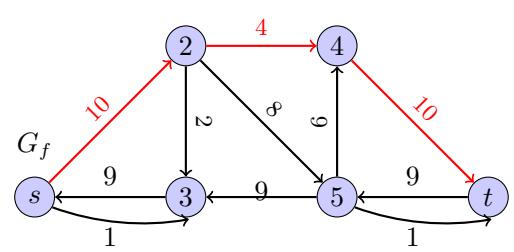
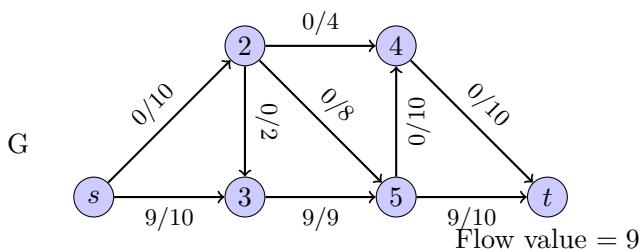
图 G



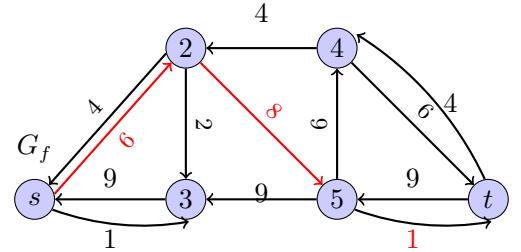
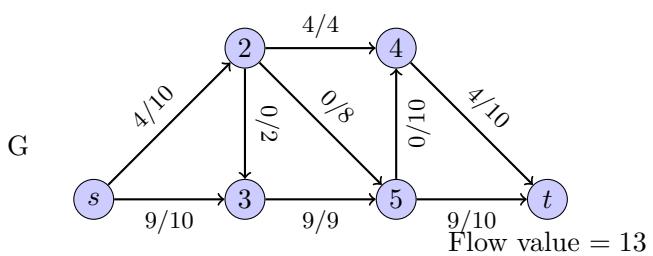
step1



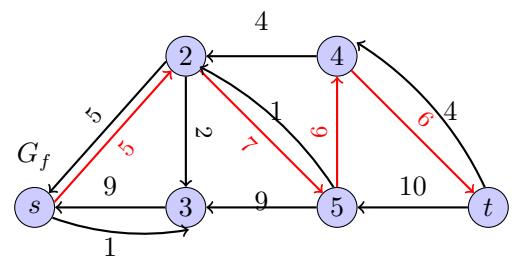
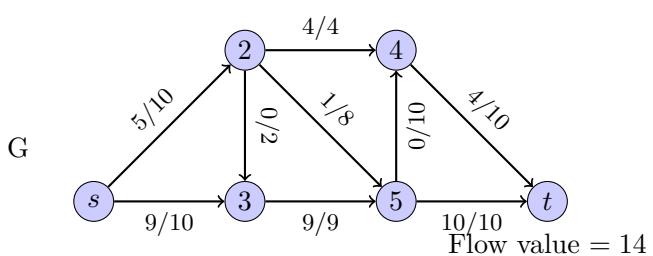
step2



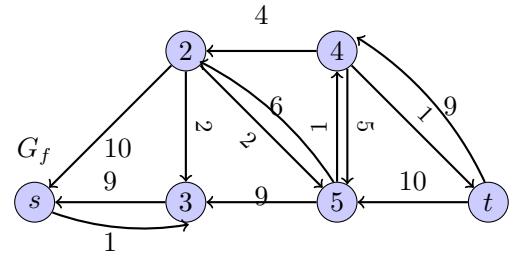
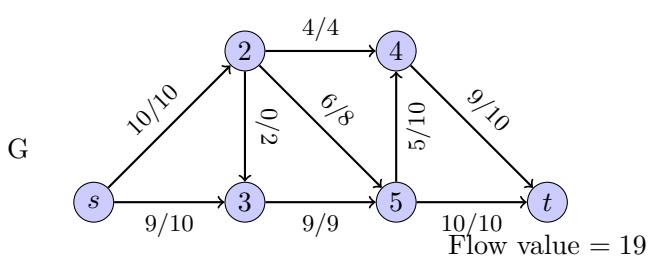
step3

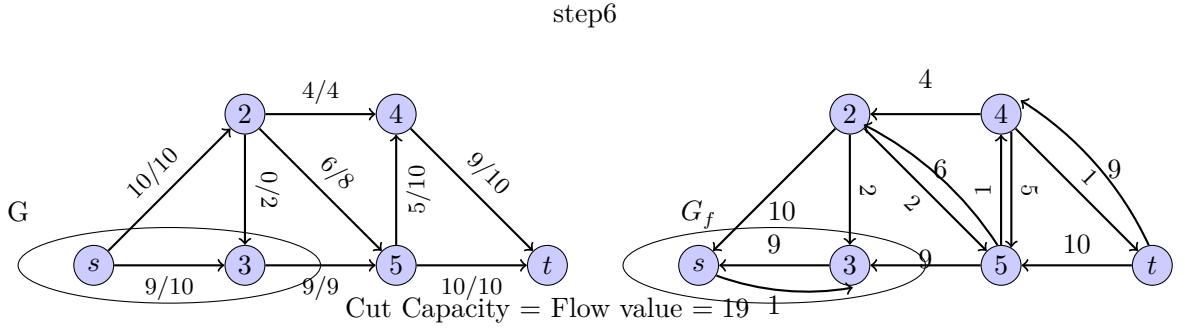


step4



step5





10.4.3 时间复杂性分析

引理 1. EDMONDS-KARP 算法中, 一条边 $e = (u, v)$ 作为 bottleneck 边的次数至多为 $\frac{n}{2}$ 次。

证明.

- 将剩余图 G_f 的节点分层为 L_0, L_1, \dots , 其中 $L_0 = s$, L_i 表示从 s 出发, 最短路径为 i 的所有点的集合, 用 $L(u)$ 表示节点 u 的层数。
- 假设一条边 $e = (u, v)$ 在 G_f 中连续两次作为 bottleneck, 依次记为 while 循环中的第 k 步, 第 k''' 步。
- 在第 k 步: $L(v) = L(u) + 1$, 经过这一步流的增广, bottleneck 边 $e = (u, v)$ 的反向边将出现在 G_f 中 (只有反向边 $e' = (v, u)$ 没有正向边, 因为该边是瓶颈 (bottleneck) 边)。
- 在第 k''' 中, $e = (u, v)$ 又成为 bottleneck 边。这意味着中间某一次循环, 记为 $k''(k < k'' < k''')$ 中, $e' = (v, u)$ 出现在最短路径上 (此次增广后, $e = (u, v)$ 将出现在 G_f 中)。因此有 $L''(u) = L''(v) + 1$, 即 $L''(u) = L''(v) + 1 > L(v) + 1 = L(u) + 2$ 。
- 又由于对任意节点, 层数至多为 n , 因此有引理成立。

□

定理 9. EDMONDS-KARP 算法运行时间为 $O(m^2n)$ 。

证明.

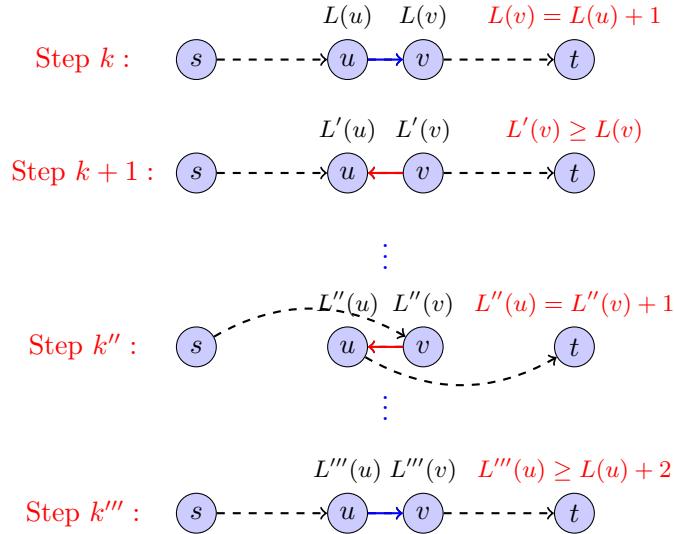
- 由引理 1 知, 每条边 $e = (u, v)$ 至多会被作为 bottleneck 边 $\frac{n}{2}$ 次。因此, while 循环至多进行 $\frac{n*m}{2}$ 次 (总共有 m 条边)。

- 使用 BFS 寻找最短路径耗时为 $O(m)$ 。

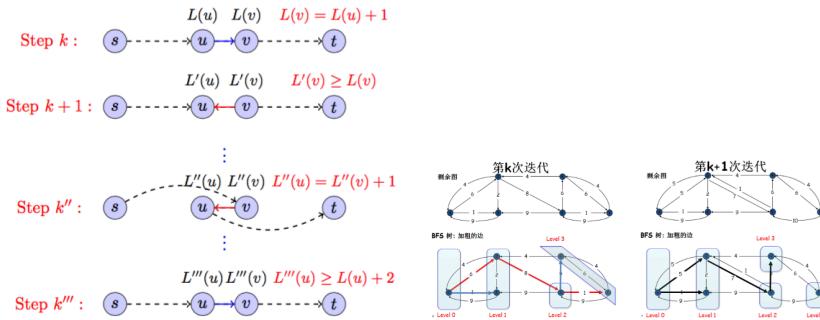
则有总的时间复杂度为 $O(m^2n)$ 。

□

以下为该定理的图形化解释



还有一个很关键的点是层数不会下降，如下图所示：



10.5 改进策略三：Dinitz 算法及 Dinic 算法

10.5.1 原始的 Dinitz 算法

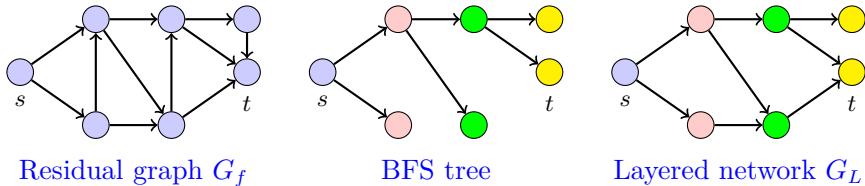
算法思想与 FORD-FULKERSON 算法类似，区别在于借助了一个数据结构。

在 FORD-FULKERSON 算法中寻找一条增广路径需要 $O(m)$ 的时间，但利用 BFS TREE 就可以存储子序列的迭代过程，就可很快的进行搜索。

DINITZ 算法就是将 BFS TREE 的思想加入到分层网络中：

- BFS TREE: 仅包含到达 v 的第一条边
- LAYERED NETWORK: 包含剩余图中 $s - v$ 的所有最短路径上的边。

如下图所示:



操作较为繁琐，因此应用不太广泛。

10.5.2 Dinic 算法

算法简述

SHIMON EVEN 和 ALON ITAI 在 Y.DINITZ 的基础上又融合了 A.KARZANOV 的思想。主要修改了以下两部分:

- 阻塞流 (A.KARZANOV 首先提出): 用 $O(m)$ 的时间构造分层网络 G_L , 每得到一个 G_L , 不断地进行增广, 直到没有路径。
- 用 DFS 搜索增广路径。

算法实现

DINIC'S ALGORITHM

```

1: INITIALIZE  $f(e) = 0$  FOR ALL  $e$ .
2: while TRUE do
3:   CONSTRUCT LAYERED NETWORK  $G_L$  FROM RESIDUAL GRAPH  $G_f$ ;
4:   if  $dist(s, t) = \infty$  then
5:     BREAK;
6:   end if
7:   FIND A BLOCKING FLOW  $f'$  IN  $G_L$  USING DFS TECHNIQUE;

```

8: AUGMENT FLOW f BY f' ;

9: **end while**

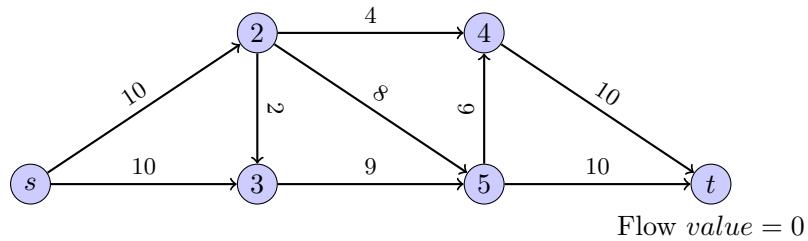
10: **return** f ;

- 这里，阻塞流就代表了在 G_L 网络中经过该流的增广，不再存在 $s - t$ 的路径。
- 而在使用 $O(m)$ 时间得到一个分层网络后，EDMONDS-KARP 每次只增广一条路径（效率较低）。

算法实例：

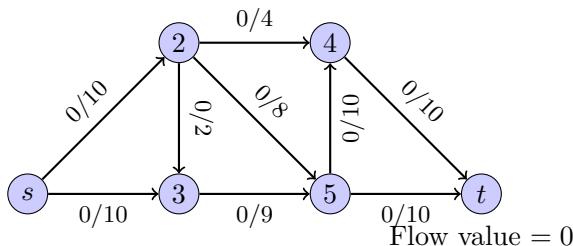
Dinic 算法实例

图 G



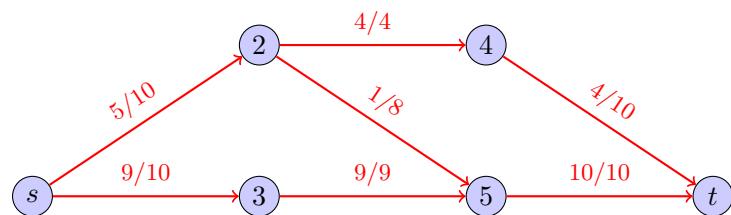
step1

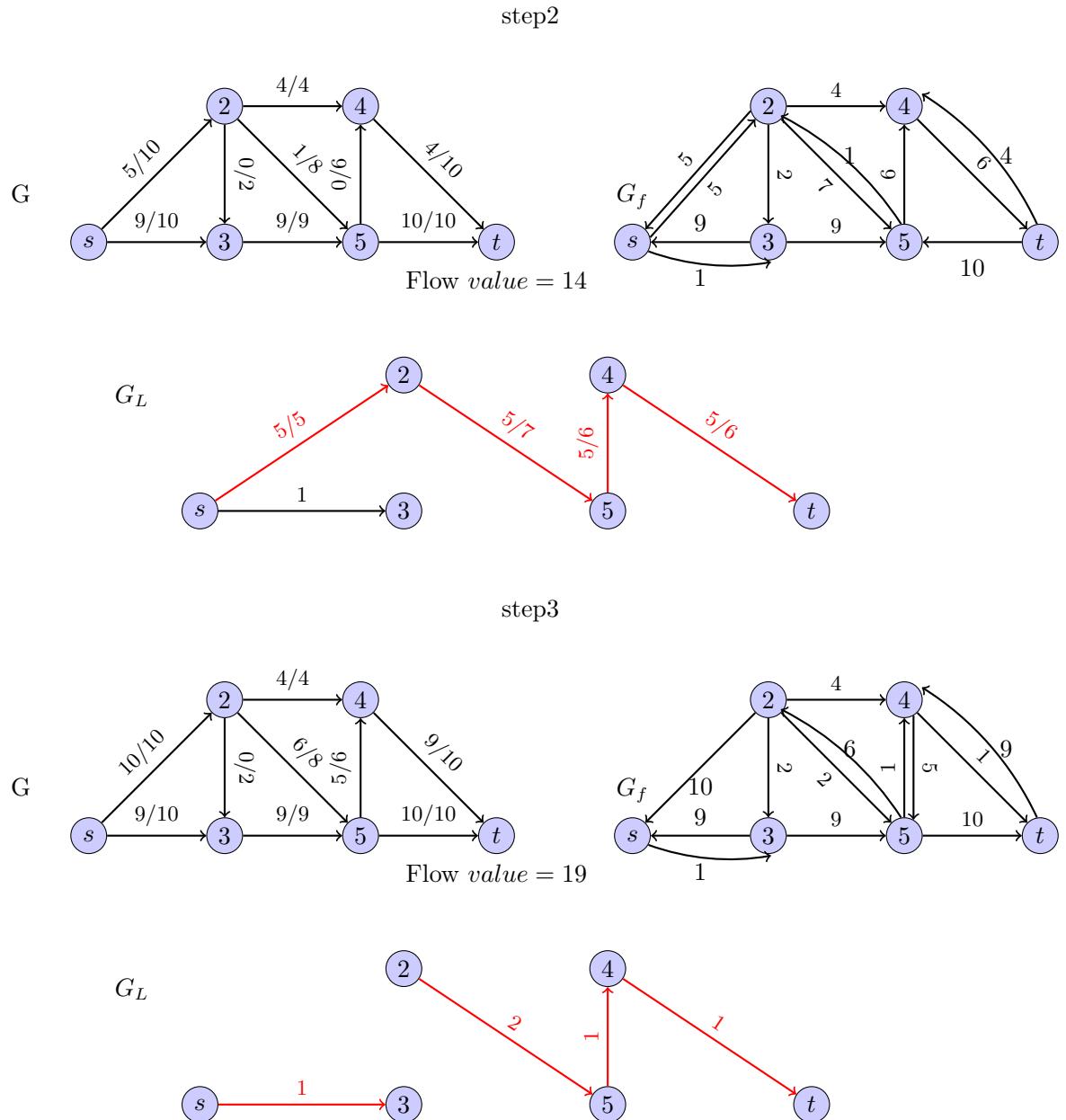
G

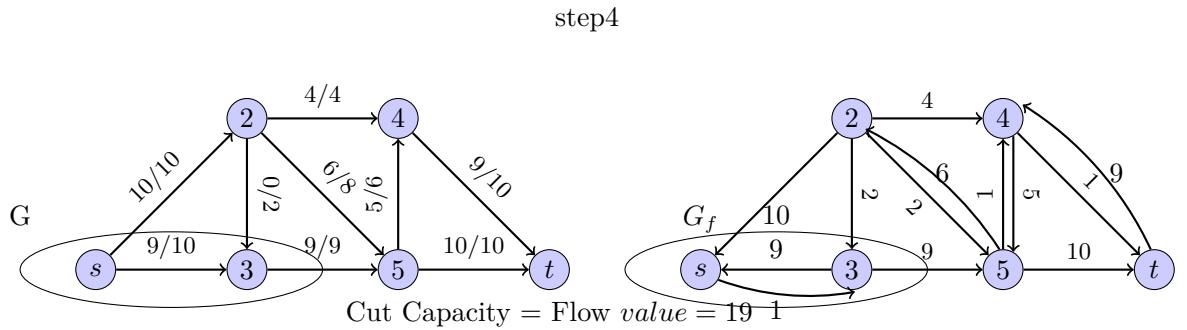


G_f

G_L







算法分析

总的时间复杂度为 $O(mn^2)$

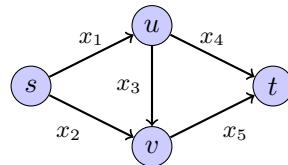
- 构造分层网络: $O(m)$ (扩展的 BFS)
- 寻找阻塞流: $O(mn)$ 。其中,
 - 1) 用 DFS 在分层网络中寻找一条 $s - t$ 路径需要 $O(n)$ 时间。
 - 2) 每条路径至少饱和一个 BOTTLENECK 边。因此, 寻找一条阻塞流至多需要 M 次迭代。
- $\#while = O(n)$ (与 EDMOND-KARP 增广过程分析相同)

因此总的时间复杂度为 $O(mn^2)$

10.6 从对偶的角度理解网络流

另一个较为简单的思想是利用线性规划求解。

10.6.1 最大流 - 最小割：对偶问题



DUAL: 给边设置变量 (x_i 表示边 i 的 flow)

$$\begin{array}{lllll}
 \max & & f \\
 \text{s.t.} & x_1 + x_2 & -f = 0 & \text{VERTEX } s \\
 & -x_4 - x_5 + f = 0 & & \text{VERTEX } t \\
 & -x_1 + x_3 + x_4 & = 0 & \text{VERTEX } u \\
 & -x_2 - x_3 + x_5 & = 0 & \text{VERTEX } v \\
 & x_1 & \leq C_1 \\
 & x_2 & \leq C_2 \\
 & x_3 & \leq C_3 \\
 & x_4 & \leq C_4 \\
 & x_5 & \leq C_5 \\
 & x_1, x_2, x_3, x_4, x_5 & \geq 0
 \end{array}$$

则可通过单纯形法得到得到最大流，但由于该问题较为特殊，因此还有其他解决方法。以下是一个等价的线性规划表示版本：

$$\begin{array}{lllll}
 \max & & f \\
 \text{s.t.} & x_1 + x_2 & -f \leq 0 & \text{VERTEX } s \\
 & -x_4 - x_5 + f \leq 0 & & \text{VERTEX } t \\
 & -x_1 + x_3 + x_4 \leq 0 & & \text{VERTEX } u \\
 & -x_2 - x_3 + x_5 \leq 0 & & \text{VERTEX } v \\
 & x_1 & \leq C_1 \\
 & x_2 & \leq C_2 \\
 & x_3 & \leq C_3 \\
 & x_4 & \leq C_4 \\
 & x_5 & \leq C_5 \\
 & x_1, x_2, x_3, x_4, x_5 & \geq 0
 \end{array}$$

由于约束 (1),(2),(3),(4) 中任意三个不等式相加，所得到的不等式与第四个不等式联立，即为第四个等式，所以这两组线性规划是等价的。

10.6.2 原始问题

PRIMAL: 给 节点 设定变量

$$\begin{array}{lllllll}
 \min & C_1 z_1 & + C_2 z_2 & + C_3 z_3 & + C_4 z_4 & + C_5 z_5 \\
 \text{s.t.} & y_s & -y_u & +z_1 & & & \geq 0 \\
 & y_s & & -y_v & +z_2 & & \geq 0 \\
 & & y_u & -y_v & & +z_3 & \geq 0 \\
 & & -y_t & +y_u & & +z_4 & \geq 0 \\
 & & -y_t & & +y_v & & +z_5 \geq 0 \\
 & -y_s & +y_t & & & & \geq 1 \\
 & y_s, & y_t, & y_u, & y_v, & z_1, & z_2, & z_3, & z_4, & z_5 \geq 0
 \end{array}$$

对该线性规划的分析：

- 因为约束条件中 y_i 的出现是相对的，因此可以设定其中一个 y_i 的值，不妨令 $y_s = 0$ ，则由约束条件 (6) 可得 $y_t \geq 1$ 。
- 由约束条件 (4) $z_4 \geq y_t - y_u$ ，又目标函数是使得 $C_4 z_4$ 尽可能的小，因此有 $y_t = 1$ 。
- 由约束条件 (1) $z_1 \geq y_u$ ，又目标函数是使得 $C_1 z_1$ 尽可能的小，因此有 $z_1 = y_u$ 。

因此原问题可化为下述等价版本：

$$\begin{array}{lllll}
 \min & C_1 z_1 & + C_2 z_2 & + C_3 z_3 & + C_4 z_4 & + C_5 z_5 \\
 \text{s.t.} & -y_u & +z_1 & & & = 0 \\
 & -y_v & & +z_2 & & = 0 \\
 & y_u & -y_v & & +z_3 & \geq 0 \\
 & y_u & & & +z_4 & \geq 1 \\
 & & y_v & & +z_5 & \geq 1 \\
 & y_s & & & & = 0 \\
 & y_t & & & & = 1 \\
 & y_u, & y_v, & z_1, & z_2, & z_3, & z_4, & z_5 \geq 0
 \end{array}$$

由于约束的系数矩阵是全单模的，因此最优解一定是整数解。所以变量是 0/1 变量。原问题又可以等价为：

$$\begin{array}{lllll}
 \min & C_1 z_1 & + C_2 z_2 & + C_3 z_3 & + C_4 z_4 & + C_5 z_5 \\
 \text{s.t.} & -y_u & +z_1 & & & = 0 \\
 & -y_v & & +z_2 & & = 0 \\
 & y_u & -y_v & & +z_3 & \geq 0 \\
 & y_u & & & +z_4 & \geq 1 \\
 & & y_v & & +z_5 & \geq 1 \\
 & y_s & & & & = 0 \\
 & y_t & & & & = 1 \\
 & y_u, & y_v, & z_1, & z_2, & z_3, & z_4, & z_5 & = 0/1
 \end{array}$$

直观的理解, z_i 表示第 i 条边, $z_i = 1$ 表示选择第 i 条边作为割边, 则最终目标函数就是求最小割。即最大流的对偶问题是最小割问题。

- 如果节点 i 在 A 中, 则 $y_i = 0$ (由 $z_i = 1$ 可得), 否则 $y_i = 1$ 。
- 由弱对偶性, $f \leq c$ (流 \leq 割), 即最大流最小割定理。

10.6.3 FORD-FULKERSON 算法本质上是一个原始对偶算法

$$\begin{array}{lllll}
 \max & & & f \\
 \text{s.t.} & x_1 + x_2 & -f & \leq 0 & \text{VERTEX } s \\
 & -x_4 - x_5 + f & \leq 0 & & \text{VERTEX } t \\
 & -x_1 + x_3 + x_4 & & \leq 0 & \text{VERTEX } u \\
 & -x_2 - x_3 + x_5 & & \leq 0 & \text{VERTEX } v \\
 & x_1 & & \leq C_1 \\
 & x_2 & & \leq C_2 \\
 & x_3 & & \leq C_3 \\
 & x_4 & & \leq C_4 \\
 & x_5 & & \leq C_5 \\
 & x_1, x_2, x_3, x_4, x_5 & & \geq 0
 \end{array}$$

对该对偶问题转化为 DRP 形式

- 将约束右边的 C_i 换成 0。
- 增加约束 $x_i \leq 1, f \leq 1$ 。
- 将 J 中的约束条件分为两类, $J = J^S \cup J^E$, 其中, $J^S = i | x_i = C_i$ 在对偶 D 中, $J^E = j | x_j = C_j$ 在对偶 D 中, 直观上来看, J^S 表示饱和边, 而 J^E 表示空边 (边上流值为 0)。

DRP 描述:

- DRP:

$$\begin{array}{llllll}
 \max & & & f \\
 s.t. & x_1 + x_2 & -f = 0 & \text{VERTEX } s \\
 & -x_4 - x_5 + f = 0 & \text{VERTEX } t \\
 & -x_1 + x_3 + x_4 & = 0 & \text{VERTEX } u \\
 & -x_2 - x_3 + x_5 & = 0 & \text{VERTEX } v \\
 & x_i & \leq 0 & i \in J^S \\
 & x_j & \geq 0 & j \in J^E \\
 & x_1, x_2, x_3, x_4, x_5, f & \leq 1
 \end{array}$$

- FORD-FULKERSON 算法本质上是原始对偶算法。
- 由于 $\omega_{OPT} \leq 1$, 又全单模系数矩阵使得解的值为整数, 因此 ω_{OPT} 只有以下两种情况
 - $\omega_{OPT} = 0$ 意味着已找到最优解。
 - $\omega_{OPT} = 1$ 代表在剩余图 G_f 中的一条 $s - t$ 路径。
- 为什么会构成剩余图 G_f ?

$x_i \leq 0, i \in J^S$ 代表反向边, $x_j \geq 0, j \in J^E$ 代表正向边, 对于其他的边 x_i 没有约束。

10.7 Push-relabel 算法

PUSH-RELABEL 算法是求最大流的一个更高效的算法。初始版本的复杂性为 $O(n^2m)$ (与 DINIC 算法时间复杂性相当). 而选用如下数据结构可使得算法复杂性明显提升。

- 选用 FIFO 顶点选择规则 $O(n^3)$,
- 选用最高活跃顶点选择规则 $O(n^2\sqrt{m})$,
- 选用 SLEATOR,TARJAN 动态树数据结构 $O(mn \log(n/m))$

10.7.1 Push-relabel 算法简介

基本思想: 增广流的基本思想是每次维持对偶线性规划的可行性, 而 PUSH-RELABEL 方法每次维持原问题的可行性。

先来介绍几个定义。

预流 (pre-flow)

定义 1. 若 f 满足以下两条, 称 f 是预流

- 容量条件 (*Capacity condition*): $f(e) \leq C(e)$
- 超额条件 (*Excess condition*): 任意节点 $v \neq s, E_f(v) = \sum_{e \in intov} f(e) - \sum_{e \in outofv} f(e) \geq 0$

若所有仓库都是空的 ($\forall v, E_f(v) = 0$), 则称预流 f 是流。

标号 (label)

定义 2. (有效标号) 对每个节点 $v \in V$ 和一个预流 f , 其高度 $h(v)$ 满足:

- $h(t) = 0, h(s) = n$
- 对于剩余图 G_f 的每条边 (u, v) , 均有 $h(u) \leq h(v) + 1$ (G_f 中的有向边 (u, v) , v 比 u 最多低 1)

10.7.2 Push-relabel 算法描述

- FORD-FULKERSON: 对边设置变量, 每次对 G_f 中的边更新流, 直到 G_f 中不存在 $s - t$ 路径。
- PUSH-RELABEL: 对节点设置变量, 每次更新预流 f , 维持 G_f 中午 $s - t$ 路径的性质, 直到 f 是一个流。

与 FORD-FULKERSON 算法的区别形式化语言描述:

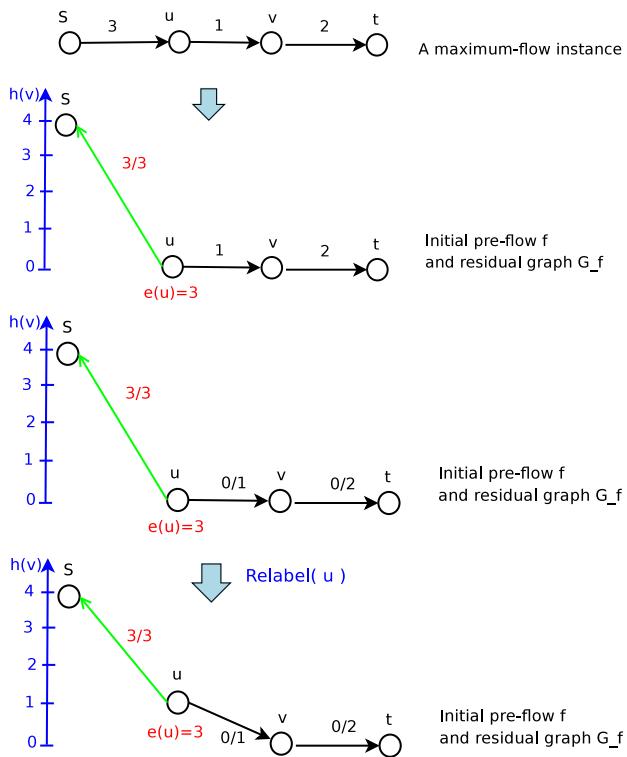
1 FORD-FULKERSON	PUSH-RELABEL
2 F IS A FLOW	F IS A PREFLOW
3 while (EXIST A S-T PATH){	while (F IS NOT A FLOW
4){	
5 SUSTAIN:F TO A FLOW	SUSTAIN: NOT EXIST
5 S-T PATH	
5 }	}

还剩下一个问题，如何标识 G_f 中无 $s - t$ 路径？标号法

定理 10. 若在 G_f 中存在有效标号 (valid label)，则 G_f 中不存在 $s - t$ 路径。

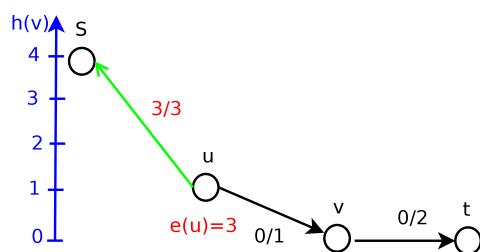
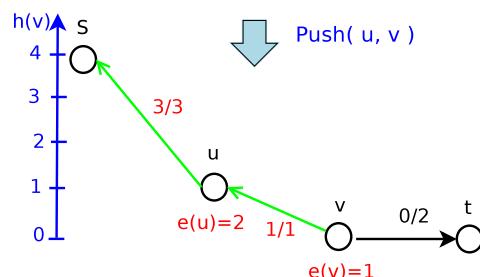
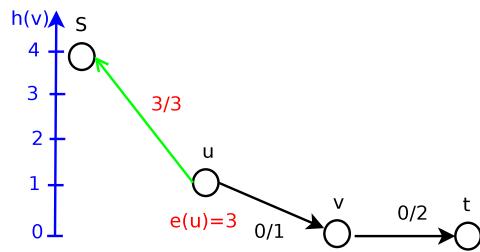
证明. 假设 G_f 中存在 $s - t$ 路径，则 $s - t$ 路径包含至多 $n - 1$ 条边。由于 $h(s) = n$ 且 $h(u) \leq h(v) + 1$ ，则 t 的高度应大于 0，与 $h(t) = 0$ 矛盾。 \square

下面用图形来描述，这里只描述了网络中的其中一条路径。



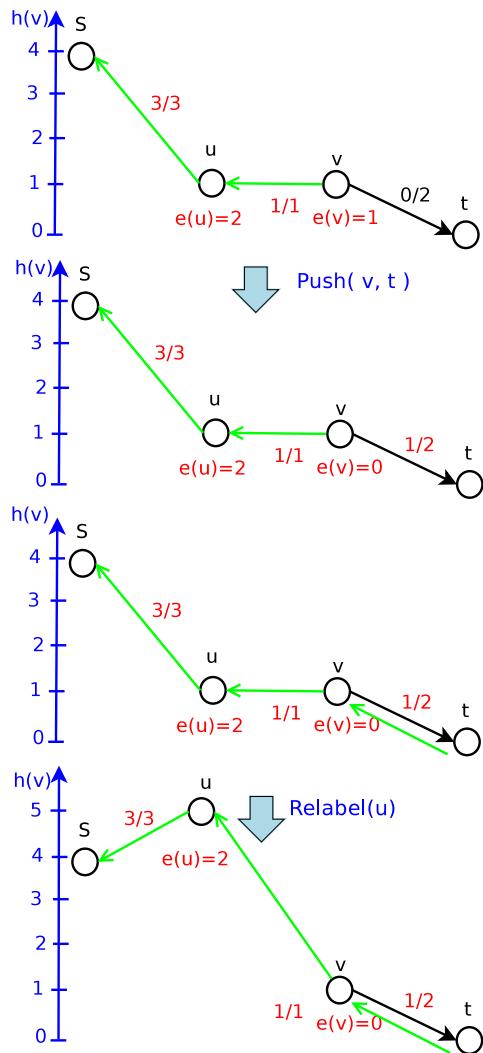
初始化

STEP1



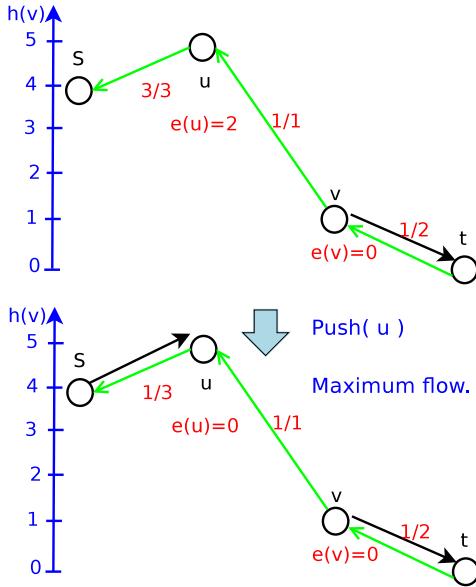
STEP2

STEP3



STEP4

STEP5



STEP6

10.7.3 Push-relabel 算法实现

HIGH-LEVEL 算法:

初始化:

- 预流设置: 初始时将所有货物均运出, $f(s, u) = C(s, u)$, 且对其他边有 $f(u, v) = 0$ 。
- 标号设置: $h(s) = n$, 其他顶点有 $h(v) = 0$ 。

迭代过程: 每一步, 对于 $E(f) > 0$ 的节点 v

- 若存在一个标号小于 v 的邻居节点 u , 则增加 v 指向 u 的流。
- 否则, 在标号合理前提下, 增加其高度 $h(v)$ 。

PUSH-RELABEL ALGORITHM:

- 1: $h(s) = n;$
- 2: $h(v) = 0;$ FOR ANY $v \neq s;$

```

3:  $f(e) = C(e)$  FOR ALL  $e = (s, u)$ ;
4:  $f(e) = 0$ ; FOR OTHER EDGES;
5: while THERE EXISTS A NODE  $v$  WITH  $E_f(v) > 0$  do
6:   if THERE EXISTS AN EDGE  $(v, w) \in G_f$  S.T.  $h(v) > h(w)$ ; then
7:     //PUSH EXCESS FROM  $v$  TO  $w$ ;
8:     if  $(v, w)$  IS A FORWARD EDGE; then
9:        $e = (v, w)$ ;
10:       $bottleneck = \min\{E_f(v), C(e) - f(e)\}$ ;
11:       $f(e) += bottleneck$ ;
12:    else
13:       $e = (w, v)$ ;
14:       $bottleneck = \min\{E_f(v), f(e)\}$ ;
15:       $f(e) -= bottleneck$ ;
16:    end if
17:  else
18:     $h(v) = h(v) + 1$ ; //RELABEL NODE  $v$ ;
19:  end if
20: end while

```

时间复杂度: $T = O(n^2m)$

第十一章 网络流的应用

11.1 上节回顾

上一节我们讲到了网络流，以及计算网络流的很多种算法，包括最原始的 FORD-FURKERSON，后面的 EDMONDS-KARP 算法，DINITZ 算法，以及 TARJAN 在 1983 年提出的 PUSH-RELABEL 算法。其中有一个很关键的技巧就是缩放 (SCALING)。

11.2 本节提要

本节我们要讲网络流的应用。因为网络流和线性规划是非常强有力的武器，掌握了他们之后，一大类的问题都可以归结成这种技术。在这里提醒大家一点，建模的本领是我们的看家本领之一，而建模的本事在以下几个地方特别需要强调：

1. 线性规划，还有我们后面要讲的半正定规划
2. 网络流，也就是本节的主题
3. 问题的规约，下节课我们将 NP-HARD 的时候将会提到

本节要将以下几个部分：

- 扩展的最大流问题：无向图上的最大流；多源点、多汇点的流通问题；每条边有流量下界的流通问题；最小费用流 MINIMUM COST FLOW；
- 使用网络流和原始对偶技术解决实际问题：

1. 集合划分: 给我们一个集合, 让我们把集合分成两堆, 有可能可以建模成网络流, 常见的问题有: IMAGESEGMENTATION, PROJECTSELECTION, PROTEINDOMAINPARSING;
 2. 在网络中寻找路径: 常见的有这些问题: FLIGHTSCHEDULING, DISJOINT PATHS, BASEBALLELIMINATION;
 3. 拆分数字: BASEBALLELIMINATION;
 4. 构造匹配: BIPARTITEMATCHING, SURVEYDESIGN;
- 匹配的扩展: 这是计算机算法历史上十分重要的一个部分, 比如二分图匹配 BIPARTITEMATCHING, 加权二分图匹配 WEIGHTEDBIPARTITEMATCHING, 一般图匹配 GENERALGRAPHMATCHING, 加权一般图匹配 WEIGHTEDGENERALGRAPHMATCHING;
 - 网络流发展的简要历史.

11.3 网络流问题的扩展

网络流问题有以下几个扩展:

1. 无向图上的最大流 (MAXIMUMFLOW) 问题;
2. 多源点、多汇点的流通 (CIRCULATION) 问题;
3. 每条边有流量下界的流通 (CIRCULATION) 问题;
4. 最小费用流问题 (MINIMUM COST FLOW);

11.3.1 无向图上的最大流问题

问题的定义

一个无向图上的最大流问题可以形式化定义如下:

输入: 一个无向图 $G = \langle V, E \rangle$, 每条边 e 有容量 $C(e) > 0$. 两个特殊的节点: 源点 s 和汇点 t ;

输出: 对每条边 e , 指定一个流量值 $f(e)$, 使得总的流量值 $\sum_{e=(s,v)} f(e)$ 最大.

流性质: 只有这里和之前讲的有向图上的最大流问题不同

1. 容量限制: 从 u 到 v 和从 v 到 u 的流量加起来不能超过 (u, v) 上的容量, 即 $0 \leq f(u, v) + f(v, u) \leq C(u, v), \forall (u, v) \in E$;
2. 存储 (CONSERVATION) 限制: $f^{in}(v) = f^{out}(v), \forall v \in V \setminus \{s, t\}$.

一个例子

下面我们举一个例子:

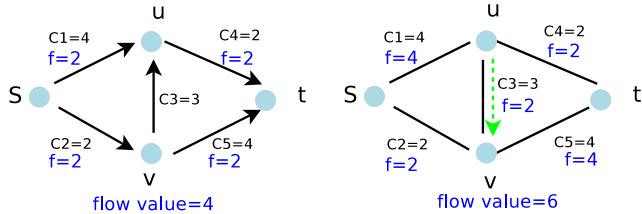


图 11.1: 有向图与无向图上的最大流

如图 11.1 所示, 左边是上一节我们遇到的有向图上的最大流, 使用上一节的方法, 可以得到总流量是 4. 而右边是一个无向图, 比如 s 到 u , 可以看做是一个双向的铁路, 可以从 u 往 s 运, 也可以从 s 往 u 运, 它们加起来不能超过 4 吨. 这是一个很自然的改变, 我们也经常遇到. 这时在使用上一节讲到的那些算法就会出问题了, 如图 11.1 右边的所示, 可以运 6 吨. 也就是说我们上一节的算法不能直接使用, 需要做一些修改.

算法

那么, 有向图我们会做了, 无向图不会做。我们能不能将无向图转换成有向图? 首先, 将无向图转换成有向图, 在我们会做的有向图上面做完之后, 再把它改一下, 回归到原始的问题。这种思想在后面两个问题中也将用到。算法大意如下:

无向图 G 上的最大流算法

- 1: 将无向图 G 转换成有向图 G' ;
- 2: 使用 FORD-FULKERSON 算法计算 G' 上的最大流;
- 3: 校正 (REVISING) 每条边上的流量, 使其满足无向图上的流量限制;

大意明白了, 接下来我们一步一步具体看, 首先将无向图 G 转换成有向图 G' :

1. 添加边: 对图 G 中每一条边 (u, v) , 在 G' 中引入两条新的边 $e = (u, v)$ 和 $e' = (v, u)$;
2. 设置容量: $C(e') = C(e) = C(u, v)$.

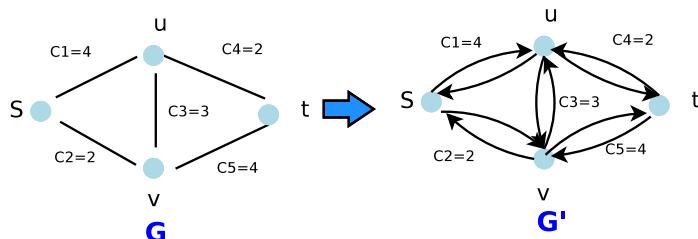
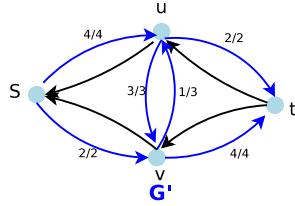


图 11.2: 无向图到有向图的转换

接下来, 在 G' 上使用有向图的最大流算法, 得到如图 11.3 上蓝色的边所示的一个最大流。唯一的问题是在边 (u, v) 上, 从 u 到 v 运 3 吨, 从 v 到 u 运 1 吨, 总共加起来是 4 吨, 超过了原图 G 中的容量限制 3.

图 11.3: G' 上的一个最大流

最后, 校正每条边上的流量, 已消除像 (u, v) 边上这样违反容量限制的问题. 如图 11.4 所示, 只要有 $u - v$ 这样的一个圈, 那么从 v 到 u 就不运送了, 而从 u 到 v 少运 1 吨, 运 2 吨就可以了, 其他的边没有变化. 简单来说, 双向铁路上流量小的边就不运了, 而流量大的边的流量减少相同的值. 形式化的, 两条边 $e = (u, v)$ 和 $e' = (v, u)$, 如果有 $f(e) = a > b = f(e')$, 则设定 $f(e) = a - b$ 且 $f(e') = 0$

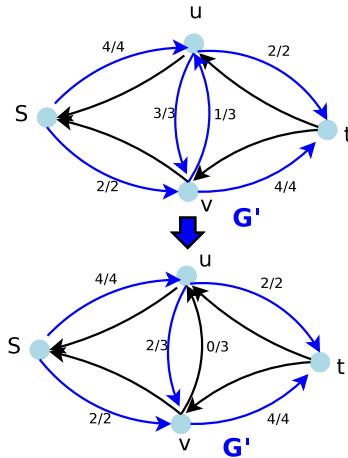


图 11.4: 校正每条边上的流量

算法的正确性

由此, 我们可以得到如下的定理:

定理 11. 在有向图 G 上存在一个最大流 f , 使得 $f(u, v) = 0$ 或 $f(v, u) = 0$.

这么改之后我们要验证两个条件。第一，每个城市是不是流入等于流出；第二，每条边上的运量之和不超过边上的容量。第一个是很显然的。第二个，我们有 $f(u, v) \leq C$ 且 $f(v, u) \leq C$ ，假设 $f(v, u) > f(u, v)$ ，我们有 $f'(u, v) = f(u, v) - f(v, u) \leq C$ ，满足容量条件，如图 11.5 所示。

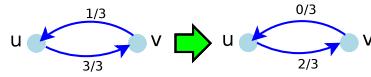


图 11.5: 对 (u, v) 边进行流量校正

Proof.

假设 f' 是有向图 G' 上一个的最大流，满足 $f'(u, v) > 0$ 且 $f'(v, u) > 0$ 。我们将流 f' 转换成无向图上的最大流 f 如下：

令 $\delta = \min\{f'(u, v), f'(v, u)\}$.

令 $f(u, v) = f'(u, v) - \delta$, 令 $f(v, u) = f'(v, u) - \delta$. 我们有 $f(u, v) = 0$ 或者 $f(v, u) = 0$.

显然这样定义的无向图上的流量满足容量限制和存储限制。

f 的值和 f' 相同，所以 f 是最优的。 \square

11.3.2 多源点、多汇点的流通问题

问题的定义

一个多源点、多汇点的流通问题可以形式化定义如下：

输入： 一个图 $G = \langle V, E \rangle$, 每条边 e 有容量 $C(e) > 0$. 存在多个源点 s_i 和多个汇点 t_j . 每个汇点 t_j 都有一个需求 $d_j > 0$, 每个源点 s_i 可以提供 d_i (表示为负的需求 $d_i < 0$).

输出： 一个可行流通 (feasible circulation) f , 它满足所有节点的要求，也就是：

1. 容量限制: $0 \leq f(e) \leq C(e)$;
2. 需求限制: $f^{in}(v) - f^{out}(v) = d_v$;

为表述简单, 我们使用一个小技巧, 对所有既不是源点也不是汇点的节点 v , 我们定义 $d_v = 0$. 由此, 我们有 $\sum_i d_i = 0$. 同时我们令 $D = \sum_{d_v > 0} d_v$ 为总需求量.

注意, 流通 CIRCULATION 问题和多物品流 MULTICOMMODITIES 问题是不同的:

1. 流通问题: 流网络中只存在一种货物. 一个汇点 t_i 可以从任意一个源点接受货物. 也就是说, 汇点 t_i 的需求是由所有源点来的货物组合而来.
2. 多物品流问题: 流网络中有多种物品. 比方说, 在我们需要在同一个网络中运送食物和石油. 汇点 t_i (假设需要食物) 只接受从 s_i (假设供应食物) 运出的货物 k_i . 两种货物共享这个网路. 到目前为止, 现行规划是解决多物品流的唯一的多项式时间算法.

我们来看一个例子, 如图 11.6 所示. s_1, s_2 都是货源地, 都想往外运 3 吨货物, t_1 需要 2 吨货物, t_2 需要 4 货物, 问题就是我们该怎么运?

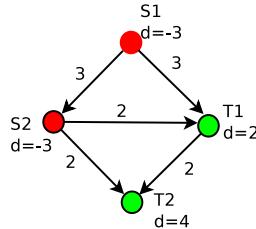


图 11.6: 两个源点 s_1, s_2 和两个汇点 t_1, t_2 的流通问题

算法

现在我们来解这个问题. 再回到原初的想法, 给我们一个问题, 我们首先考虑这个问题能不能分解, 对这个问题, 可以分. 比如, 从 s_1 向 s_2 运 3 吨, 然后再把这条边删掉. 但是这个分法会引来很多麻烦, 子问题的数目可能非常多. 所以我们就先不分, 考虑改进 (IMPROVEMENT) 的做法.

其实，使用改进的做法，就算没有下面的算法，我们也可以使用之前所学的一个强力的武器—线性规划来解这个问题。我们很容易把它写成线性规划，写完之后，如果不嫌麻烦的话就机械性的把这个问题的原始对偶以及 DRP 写出来，这个问题就求解了。事实上，我们接下来将的方法都可以通过原始对偶推出来，只不过是它的一个简化的实现罢了。

那么我们首先先把这个问题写成线性规划，对每条边 (u, v) 定义一个变量 $f(u, v)$ 表示这条边上的流。由边上有容量限制和节点的需求限制，可以得到如下的线性规划表达式（由于只要求可行解，目标函数为 0）：

$$\begin{aligned} \text{MIN } & 0 \\ \text{s.t. } & f(u, v) \leq C(u, v), & \forall (u, v) \in E \\ & \sum_v f(v, u) - \sum_v f(u, v) = d_u, \forall u \in V \\ & f(v, u) \geq 0, & \forall (u, v) \in E \end{aligned}$$

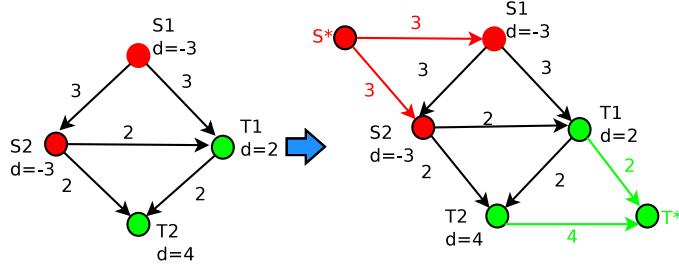
接下来，我们考虑更加简单的做法。老样子，这里有两个收货地，我们过去做的只有单个发货收货地。跟之前一样，能不能把这个不会做的问题转化成会做 的问题呢？我们构造一个图，这个图上只有一个发货地，一个收货地，然后做网络流，最后在返回到原始问题。就是如下的做法：

- 1: 对原网络增加一个超级源点 S^* 和超级汇点 T^* ，构成一个扩展网络 G' ；
- 2: 使用 FORD-FULKERSON 算法计算得到 G' 上的一个最大流 f ；
- 3: 如果最大流的值等于 $D = \sum_{v:d_v>0} d_v$ ，则返回流 f 。

接下来，我们一步一步来看。

第一步：构造扩展网络 G'

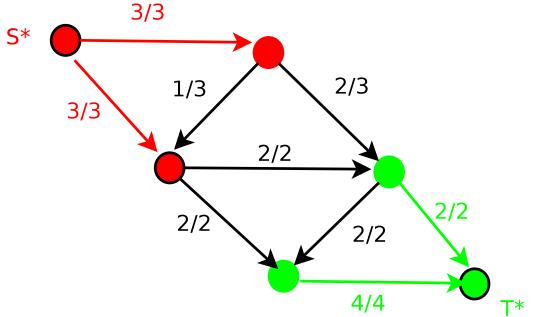
转化：如图 11.7 所示，构造网络 G' ：增加一个超级源点 s^* ，指向所有的货源地 s_i ，链接的边上的容量为 $C(s^*, s_i) = -d_i$ 。类似地，增加的一个超级汇点 t^* ，所有汇点 t_j 都指向它，对应边上权值为 $C(t_j, t^*) = d_j$ 。

图 11.7: 构造扩展网络 G'

这种转化方法我们可以这样看：比如 s_1 , 有 3 吨货要往外运, 我们就假设这 3 吨货是从 s^* 运来的. 同理, 对收货地 t_1 和 t_2 最终也要运到 t^* .

第二步：计算 G' 上的最大流

接下来, 问题就转化成我们会做的问题了, 在 G' 上跑 FORD-FULKERSON, 得到最大流如图 11.8 所示.

图 11.8: 计算 G' 上的最大流

第三步：检查 G' 上的最大流

如图 11.9 所示, 我们得到的最大流是 6 吨, 等于 S^* 运出货物的总量, 即 $6 = \sum_{v, d_v > 0} d_v$. 因此, 原始问题是解. 只要把边原封不动地拷过去就得到了一个原始流

通问题的可行解。但是如果求得的最大流不是 6，那就意味着原问题没有一种可能的流通方案

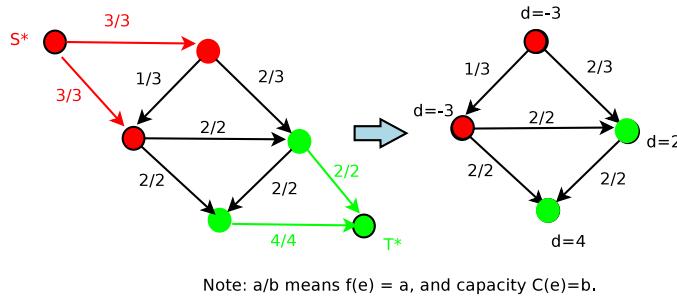


图 11.9: 检查 G' 上的最大流

算法的正确性

定理 12. 流通问题存在一个可行解当且仅当图 G' 上的最大 $s^* - t^*$ 流等于 D .

Proof.

\Leftarrow :

删除所有的 (s^*, s_i) 和 (t_j, t^*) 边。显然所有 s_i 和 t_j 都满足容量限制和需求限制。

\Rightarrow : 即如果我们能找到一个可行解, 那最大流肯定是 D

我们构造一个 $s^* - t^*$ 流并证明这个流是最大流:

1. 定义一个流 f 如下: $f(s^*, s_i) = -d_i$ 且 $f(t_j, t^*) = d_j$.
2. 考虑一个割 (A, B) , 其中 $A = \{s^*\}$, $B = V - A$, 就是超级源点和其他顶点之间的割, 显然这个割的值是 $C(A, B) = D$.
3. 我们有 $C(A, B) = D$. 因为 f 达到了最大, 所以 f 是一个最大流.

□

11.3.3 每条边有流量下界的流通问题

问题的定义

简单来说, 这里我们要求每条边上至少要运一定的货物, 完整的定义如下:

输入: 一个图 $G = \langle V, E \rangle$, 每条边 e 都有一个容量上界 $C(e)$ 和容量下界 $L(e)$. 存在多个源点 s_i 和多个汇点 t_j . 每个汇点 t_j 都有一个需求 $d_j > 0$, 每个源点 s_i 可以提供 d_i (可以表示为负的需求 $d_i < 0$).

输出: 一个可行流通 (feasible circulation) f , 它满足所有节点的需求限制, 也就是:

1. 容量限制: $L(e) \leq f(e) \leq C(e);$
2. 需求限制: $f^{in}(v) - f^{out}(v) = d_v;$

为表述简单, 对所有既不是源点也不是汇点的节点 v , 我们定义 $d_v = 0$. 由此, 我们有 $\sum_i d_i = 0$. 同时我们令 $D = \sum_{d_v > 0} d_v$ 为总需求.

一个例子

下面我们来举一个例子, 如图 11.10 所示, 跟之前问题的差别仅仅在于边 (s_1, s_2) 至少运 2 吨, 至多运 3 吨.

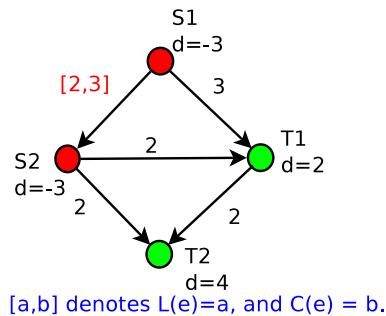


图 11.10: 带有流量下界的流通问题一个实例

流量的下界有一个很大的用途。当我们设置了边 e 上的流量下界 $L(e) > 0$, 我们强制流使用边 e . 例如, 如图 11.10 所示, 流必须使用边 EDGE (s_1, s_2) . 大家在之后工作、研究中很有可能会遇到这样的问题: 这条路我必须得选. 这是一个很有意义的扩展.

算法

这个问题大家肯定可以做, 把之前写的线性规划中的 $f(v, u) \geq 0$ 条件改成 $f(v, u) \geq L(u, v)$ 就可以解决这个问题. 我们下面要讲的仅仅是让它更快一点.

解决这个问题的思想和之前几个问题是一样的, 有下界的我不会做, 我们看看能不能把这个下界给去了. 怎么去呢? 加一个初始的流. 过去我们从 0 流开始解网络流, 我们现在则从初始流开始. 具体如下:

- 1: 建立一个初始流 f_0 : 对所有边 $e = (u, v)$, 令 $f_0(e) = L(e)$;
- 2: 在图 G' 上解一个不带流量下界的流通问题. 特别的, 图 G' 是通过校正带有流量下界的边 $e = (u, v)$ 建立的, 校正方法如下:

1. 顶点: $d'_u = d_u + L(e)$, $d'_v = d'_v - L(e)$,
2. 边: $L(e) = 0$, $C(e) = C(e) - L(e)$.

令 f' 为图 G' 的可行流通.

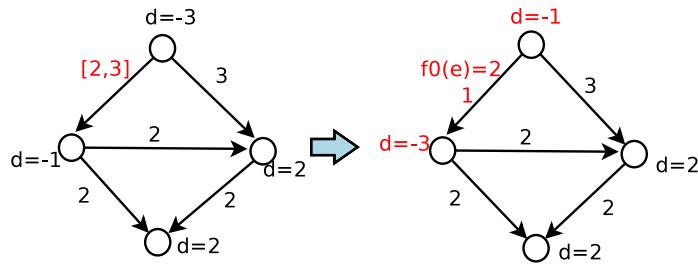
- 3: 返回 $f = f' + f_0$.

第一步: 建立初始流

如图 11.11 所示, (s_1, s_2) 边上至少要运 2 吨, 那我就让你把这 2 吨先运了, 这条边剩下的容量就是 1. 而 s_2 本身就有 1 吨要往外运, 而预先运来了 2 吨, 就变成要运 3 吨了. 其他一切都没有变.

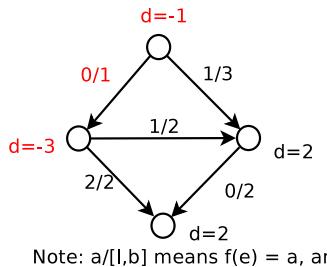
第二步: 解一个新的流通问题

经过初始流之后的问题就是我们之前讲过的流通问题, 再在这个网络 G' 上求得一个可行流通 f' 就行了, 如图 11.12.



Note: $a/[l,b]$ means $f(e) = a$, and capacity $L(e)=l$, and $C(e)=b$.

图 11.11: 建立初始流, 最上面的顶点是 s_1 , 左面的是 s_2



Note: $a/[l,b]$ means $f(e) = a$, and capacity $L(e)=l$, and $C(e)=b$.

图 11.12: 解 G' 上的流通问题

第三步: 将 f_0 和 f' 加起来

最后, 如图 11.13 所示, 左侧第一个是初始流 f_0 , 第二个是上一步得到的可行流通 f' . 把他们加起来, 我们就得到了原始问题的解 $f = f_0 + f'$.

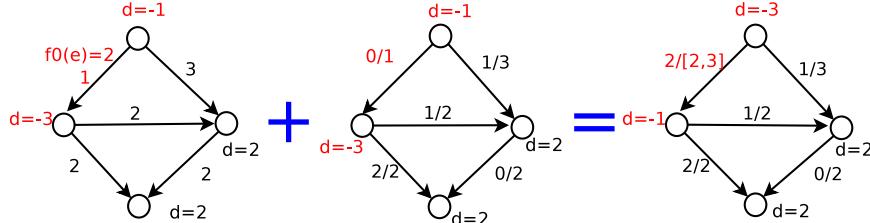


图 11.13: 将 f_0 和 f' 加起来

算法正确性

接下来我们验证，我们所得到的流满足问题所给的条件。

定理 13. 图 G (带有流量下界) 上有一个流通 f 当且仅当图 G' (没有流量下界) 上有一个流通 f' .

Proof.

令 $f'(e) = f(e) + L_e$.

容易验证其满足容量限制和流通限制。 \square

11.3.4 最小费用流问题

问题的定义

输入: 图 $G = \langle V, E \rangle$, 每条边 e 都有一个容量 $C(e) > 0$ 和一个通过这条边上运送单位物品的花费 $w(e)$. 存在两个特殊的节点: 源点 s 和汇点 t . 从 s 到 t 运送流量为 v_0 .

输出: 找一个流通 f , 使得总流量等于 v_0 且花费最少。

一个例子

我们来看一个例子, 如图 11.14 所示, 我们的目标是: 用最少的花费将 $v_0 = 2$ 单位的货物从 s 运到 t .

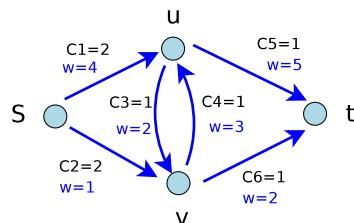


图 11.14: 最小费用流的一个实例

这个问题就不是那么容易了, 我们需要运一定的货, 使得花费的钱越少越好. 之前的一些例子我们一眼就能看出答案, 而这个例子, 没有办法直接看出答案. 为什么这么难呢, 主要是每条边上有个价钱 w_e . 它把这个问题复杂化了. 同时我们也不能像之前几个问题那样简单的将图 G 转换为另一个 G' . 怎么办呢? 我们就回到原始对偶的想法上去.

原始对偶

我们给每条边上设一个流量 y_i 代表边 i 运几吨, 由此写出图 11.14 中最小费用流的线性规划表达式如下:

$$\begin{aligned}
 \min \quad & 4y_1 + y_2 + 2y_3 + 3y_4 + 5y_5 + 2y_6 \\
 \text{s.t.} \quad & y_1 + y_2 = 2 \quad \text{NODE } s \\
 & -y_5 - y_6 = -2 \quad \text{NODE } t \\
 & -y_1 + y_3 - y_4 + y_5 = 0 \quad \text{NODE } u \\
 & -y_2 - y_3 + y_4 + y_6 = 0 \quad \text{NODE } v \\
 & y_i \leq C_i \\
 & y_i \geq 0
 \end{aligned}$$

我们已经把这个问题写成线性规划的形式了, 可以说这个问题就已经解决了. 下面的问题就只是有没有更快的解法? 我们使用下面的规则将这个线性规划重写为标准的对偶形式.

- 目标函数: 将 \min 替换为 \max .
- 限制: 将 “=” 替换为 “ \leq ”. (注意到如果所有的不等式都满足的话, 它们都应当取等号. 例如, 不等式 (2), (3) 和 (4) 使得 $y_1 + y_2 \geq 2$. 因此, 它将不等式 (1) 的 \leq 变成 $=$. 其他不等式也是一样的.)

可以得到如下的对偶形式:

$$\begin{aligned}
 \max \quad & -4y_1 - y_2 - 2y_3 - 3y_4 - 5y_5 - 2y_6 \\
 \text{s.t.} \quad & y_1 + y_2 \leq 2 \quad \text{NODE } s \\
 & -y_5 - y_6 \leq -2 \quad \text{NODE } t \\
 & -y_1 + y_3 - y_4 + y_5 \leq 0 \quad \text{NODE } u \\
 & -y_2 - y_3 + y_4 + y_6 \leq 0 \quad \text{NODE } v \\
 & y_i \leq C_i \\
 & y_i \geq 0
 \end{aligned}$$

回到我们的目标, 从 s 到 t 运 2 吨货物且花费最少. 那么首先, 第一个问题是不管价钱多少, 从 s 到 t 能不能运 2 吨货物. 第二问题, 在运 2 吨的条件下费用最小.

我们首先看第一个问题, 我们可以找到一个流量为 2 的流通. 这个很容易, 这就是一个流通问题, 我们之前已经解决了. 这样, 就相当于我们得到了原始对偶问题 D 的一个初始可行解.

接下来, 我们把这个初始可行解代进去, 看能不能接着进行改进, 这个就是我们原始对偶里最核心东西: DRP. 接下来, 我们就由对偶把这个问题的 DRP 写出来, 回想从原始对偶构造 DRP 的核心思想: 给一堆初始解 y_i , 把他们全部代进对偶里去, 看是小于号还是等于号成立. 如果是等于, 在 DRP 中是对应的约束是小于等于:

$$\begin{aligned}
 y: \text{ DUAL: } & \sum ** = C_1 \\
 \Rightarrow \Delta y: \text{ DRP: } & \sum ** \leq 0
 \end{aligned}$$

如果是严格小于, DRP 中这个约束就没了:

$$\begin{aligned}
 y: \text{ DUAL: } & \sum ** \leq C_2 \\
 \Rightarrow \Delta y: \text{ DRP: } & \text{EMPTY}
 \end{aligned}$$

这个的直观含义是, 我们要得到 $y' = y + \theta \Delta y$, 而 y' 也要满足之前的约束, 即:

$$\begin{aligned}
 \sum ** & \leq C_1 \\
 \sum ** & \leq C_2
 \end{aligned}$$

如果在对偶中是等于号 (即 C_1 这个约束), 在 DRP 中是小于号, 加起来肯定小于等于 C_1 . 如果对偶中是小于号, 即已经小于 C_2 了, 这就意味着不等式左边和 C_2 总有差值, 在 DRP 中就不用约束了, 因为我们总可以调节 θ 使得对 y' , C_2 这个不等约束成立.

对这个问题而言, 具体规则如下:

- 将右边的项替换为 0.
- 将 J 中没有的约束去除 (J 包含了 D 中使 $=$ 成立的那些约束).
- 对每条边增加约束 $y_i \geq -1$.

我们可以得到如下的 DRP:

$$\begin{aligned}
 \max \quad & -4y_1 - y_2 - 2y_3 - 3y_4 - 5y_5 - 2y_6 \\
 \text{s.t.} \quad & y_1 + y_2 \leq 0 \quad \text{NODE } s \\
 & y_5 + y_6 \leq 0 \quad \text{NODE } t \\
 & y_1 - y_3 + y_4 - y_5 \leq 0 \quad \text{NODE } u \\
 & y_2 + y_3 - y_4 - y_6 \leq 0 \quad \text{NODE } v \\
 & y_i \leq 0 \quad \text{FOR FULL ARC} \\
 & -y_i \leq 0 \quad \text{FOR EMPTY ARC} \\
 & y_i \leq 1 \quad \text{FOR ANY ARC}
 \end{aligned}$$

接下来, 先引入一个概念, 再回到 DRP 上.

定义 3 (环流 (Cycle flow)). 一个流 f , 如果对任意节点 (包括 s 和 t), 流入等于流出, 那么它就被称为环流 (cycle flow) .

如果我们已经得到了图 N 上的一个流, 那么我们把这个流带进去, 把它的 DRP 写出来, 接下来就是解这个 DRP. 解这个 DRP, 可以跑单纯性算法, 但对这个问题, 可以不用老老实实那么干, 这个问题是一个组合问题. 实际上, 解这个 DRP 与在一个新的图 $N'(f)$ 中找一个圈等价. 图 $N'(f)$ 构造方法如下:

1. 对 N 中的每条边 $e = (u, v)$, 在 $N'(f)$ 中引入两条边 $e = (u, v)$ 和 $e' = (v, u)$;
2. 将 $N'(f)$ 中的边 e 和边 e' 的容量分别设置为 $C(e) - f(e)$ 和 $-f(e)$;
3. 费用为 $w(e') = -w(e)$;

对每条边, 如图 11.15 所示, $-w$ 对应的是退货边, 可以解释为最多可以把这 f 吨都退回来, 退一吨挽回之前的损失 w 元钱 (过去的调度出错了), 也就是节省 w 元钱.

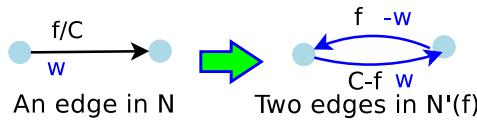


图 11.15: 构造 N'

由上, 我们不用去解这个现行规划, 只要把原先的图变一下, 然后在这个图上面找有没有负圈. 这个正确性由如下的定理保证.

定理 14. f 是图 N 上的最小费用流 \Leftrightarrow 图 $N'(f)$ 中不包含带有负花费的圈.

直觉上说, 如果网络 $N'(f)$ 存在一个圈. 如果沿着这个圈推进一个单位流, 那么这个流就是一个环流 (表示为 \bar{f}). 那么 $f + \bar{f}$ 仍然是原网络 N 的一个流.

Proof. f 是图 N 上的最小费用流

$$\begin{aligned} &\Leftrightarrow DRP \text{ 的最优解等于 } 0. (\text{也就是 } \Delta y = 0) \\ &\Leftrightarrow N'(f) \text{ 中不包含带有负花费的环流 (cycle flow).} \\ &\Leftrightarrow N'(f) \text{ 中不包含带有负花费的圈.} \end{aligned}$$

□

关于, DRP 的最优解为 0 对应于 N' 中没有负圈, 大家可以结合后面的例子, 来看下面的说明: 这个 DRP 表达式就对应着几何的直观表达, 即这个图中有负圈. 首先第一点, DRP 式子中, 对每个节点有 ≤ 0 , 我们也已经说过, 这个等价于 $= 0$, 也就是说, 对每个节点而言, 流入等于流出. 这说明这个式子所表示的流是一个环流. 第二点, 它肯定是 N' 中的环流. N' 体现在 DRP 不等式中关于边的那三组不等式. 如果该

边流量用完了，那这条边就别用了，即 $y_i \leq 0$. 如果如果是空的边，那还可以用. 第三点，负圈，体现在目标函数上.

算法

M. KLEIN 在 1967 年发明了一个最小费用流算法. 这个算法和我们之前看到的一样，都是“改进”这个套路. 就是说，一开始找一个初始解，不断改进，直到达到某个条件停止. 这个算法可以表示为：

Algorithm 1 Klein algorithm

- 1: 使用最大流算法（比如 Ford-Fulkerson 算法）得到一个值为 v_0 的流 f ;
 - 2: **while** $N'(f)$ 包含了一个负花费的圈 C **do**
 - 3: 令 b 为圈 C 的 bottleneck.
 - 4: 定义 \bar{f} 为 C 上的单位流.
 - 5: $f = f + b\bar{f}$;
 - 6: **end while**
 - 7: **return** f .
-

注意：

1. 流上的花费随着迭代逐渐减少，但同时流值保持不变.
2. 带有负花费的圈可以通过 BELLMAN-FORD 算法来寻找.

下面我们来看一个例子：

第一步

建立一个初始流 f_0 : 流值为 2, 流上的费用为: 17. 如图 11.16 所示.

接下来，我们看这种方案能不能改进. 构造新图 $N'(f)$, 如图 11.17 所示. 可以发现一个负费用圈: $s \rightarrow v \rightarrow u \rightarrow s$ (红色的部分). 费用为 -5, 相当于挽回经济损失 5 元钱.

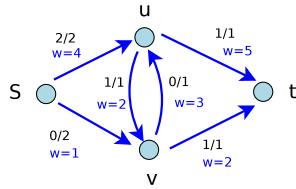
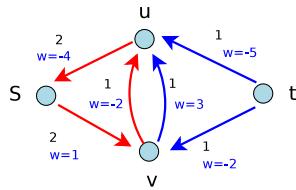
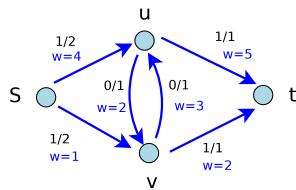


图 11.16: 建立初始流

图 11.17: 构造新图 $N'(f)$

第二步

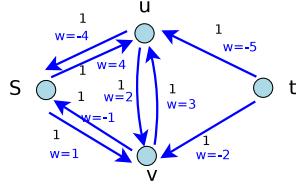
经过一轮迭代，把上面得到的两个流加起来，得到 $f = f + \bar{f}$ ，流值保持 $2 - 0 = 2$ ，费用为: $17 - 5 = 12$ ，如图 11.18 所示。

图 11.18: 迭代后的 N

再次构造 $N'(f)$ ，如图 11.19 所示。可以发现图中没有负费用的圈了，算法结束。

扩展

最小费用流有一个扩展，HITCHCOCK 运输 (TRANSPORTATION) 问题，这个问题在 1941 年就已经提出来了，问题定义如下：

图 11.19: 迭代后的 N'

输入: 存在 n 个源点 s_1, s_2, \dots, s_n 和 n 个汇点 t_1, t_2, \dots, t_n . 每个源点 s_i 可以供应 a_i , 每个汇 t_j 有一个需求 b_j . s_i 到 t_j 的费用是 c_{ij} .

输出: 设计一种调度使得费用最少.

FRANK L. HITCHCOCK 在 1941 年提出了这个问题. 这个问题等价于最小费用流 [WAGNER, 1959]. 这个问题是一个最小费用流的特例, 这个图是一个二部图, 而最小费用流是一般图. 当每条边 $c_{ij} = 0/1$ 时, 这个问题就变成了分配问题.

在 1956 年, L. R. FORD JR. 和 D. R. FULKERSON 提出了一种“标签”法来解决运输问题. 这个算法比单纯形效率高很多, 可以参见“SOLVING THE TRANSPORTATION PROBLEM”, L. R. FORD JR. AND D. R. FULKERSON.

11.4 网络流的应用

为了将一个问题转化为网络流问题, 我们需要做以下几个工作:

1. 我们首先应该定义一个网络(图). 有时原始问题有图, 我们只要改一下就行了, 而有时我们需要完全从头开始建立这个图.
2. 接着我们需要定义边上的权值. 有时原始问题中权值在顶点上, 这时我们需要将顶点上的权值移到边上.
3. 定义源点 s 和汇点 t . 有时我们需要定义超级源点 s^* 和超级汇点 t^* .

4. 最后我们需要证明最大流 (寻找路径、匹配) 或最小割 (划分节点) 正好是我们想要的, 这个是最重要的.

注意: 大多数问题都用到了这么一个性质: 存在一个整数值的最大流当且仅当存在一个最大流.

那什么样的问题天然能想到它是一个最大流呢? 一方面, 考虑最大流. 所谓流, 就是一堆 s 到 t 的路径 (或者说通路), 所以如果让我们在图中找路径, 我们可以想到最大流. 如果找到一条边, 这是一个匹配的话, 那这也是一个天然的最大流问题. 另一方面, 考虑最小割. 所谓割就是把顶点分成两堆, 所以如果原始问题让我们把顶点分成几个部分的话, 那我们就可以想到割.

11.4.1 集合划分

第一个问题, 加入问题让我们把一个集合分成两半, 这个时候我们就可以想到网络流 (最小割). 我们来看几个例子:

图片分割问题



图 11.20: 一幅像素图

我们首先来看一张图片, 如图 11.20 所示, 相信大家都看得出来这是谁, 为什么能看出来呢? 因为虽然我们没有意识到, 我们的大脑对每一像素都分辨了一次这个是前景还是背景. 那些很亮的点基本都是前景, 很暗的点基本都是背景. 而有的点我们可能

一下子无法分辨, 但是他们周围存在一些跟他们很像的点, 这是我们可以说, 这个点和田周围的这个点同属于背景或同属于前景. 这就是图片分割 (IMAGESEGMENTATION) 问题, 定义如下:

输入: 给定一个像素图格式的图像 (比如图 11.20). 像素 $i, i \in P$ 有一定概率 f_i 是前景, 有有一定概率 b_i 是北京. 另外, 两个相邻的像素 i 和 j 相似的概率是 l_{ij} ;

目标: 将前景从背景中辨识出来. 形式化的, 我们希望得到一个划分 $P = F \cup B$ (F 表示前景, B 表示背景), 使得 $Q(F, B) = \sum_{i \in F} f_i + \sum_{j \in B} b_i + \sum_{i \in F} \sum_{j \in N(i) \cap F} l_{ij} + \sum_{i \in B} \sum_{j \in N(i) \cap B} l_{ij}$ 最大.

目标中的 $Q(F, B)$ 是什么意思呢, 是以下三者之和: 你认为是前景的那些像素是前景的概率, 你认为是背景的那些像素是背景的概率, 以及所有前景中邻居也是前景的概率和所有背景中邻居也是背景的概率. 为什么要最大化这个呢? 如果大家学过机器学习有关的课程, 大家就会明白他是再做一个极大似然. 我们假设 $x_i = 0$ 表示 i 像素是背景, 而 $x_i = 1$ 表示是前景. X 表示所有像素的 x_i 组成的向量. 我们其实是在 $\max P(X|f, l)$. 我们把这个概率写成能量的指数 (或者说能量是概率的负对数), 再把能量展开, 有:

$$\begin{aligned} & \max P(X|f, l) \\ &= e^{-E(x)} \\ &= \exp \left\{ - \left(\sum_i P_i(x_i) + \sum_i \sum_j h_{ij}(x_i, x_j) \right) \right\} \end{aligned}$$

很多东西都能写成这个样子, $E(x)$ 展开玻尔兹曼分布, 为什么呢? 它是从最大熵来的. 这个玻尔兹曼分布, 我们一般就写单体项和两体项, 三体项样本数量已经不够了. 而上面那个最大似然就相当于:

$$\min \sum_i P_i(x_i) + \sum_i \sum_j h_{ij}(x_i, x_j)$$

这里补充一点:

$$\begin{cases} P(x) = e^{-E(x)} \\ E(x) = -\log P(x) \end{cases}$$

大家熟悉的是概率取负对数是熵，在这里概率取负对数是能量。其实，熵就是不能做功的能量。

回到目标中的目标函数 $Q(F, B)$ ，大家知道这个是由极大似然来的，不是瞎设的。现在我们就不管为什么写成这样的，考虑怎么解这个问题。这个问题大家可以写成优化问题，但是，很不幸的，无法直接写成线性规划的形式，直接写是一个二次优化的问题（两个求和那里得用 $x_i x_j$ ）。

接下来，我们先看一个有 9 个像素的例子，如图 11.21 所示。

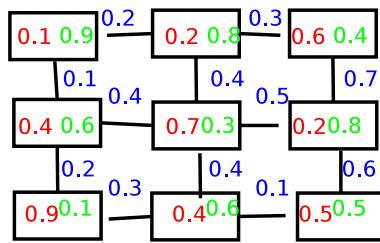


图 11.21：一个图片分割的实例

每个格子代表一个像素，其中：

- 红色（每个格子的左侧）：像素 i 是前景的概率 f_i ；
- 绿色（每个格子的右侧）：像素 i 是背景的概率 b_i ；
- 蓝色（格子之间）：像素 i 和 j 在同一类的概率 l_{ij} ；

这里需要把节点分成两堆，这个我们就可以想到最大流最小割。接下来我们按之前的四步把这个问题转化成一个网络流问题。

1. 网络：如图 11.22 所示，增加两个节点：源点 s 和汇点 t ， s 指向所有的像素，所有的像素都指向 t ；
2. 容量：将像素（顶点）上的权值移到边上，即 $C(s, v) = f_v$, $C(v, t) = b_v$;
 $C(u, v) = l_{uv}$;

3. 割: 对应一个划分. 割容量 $C(F, B) = M - Q(F, B)$, 其中 $M = \sum_i(b_i + f_i) + \sum_i \sum_j l_{ij}$ 是一个常数.

4. 最小割: 原问题的最优解

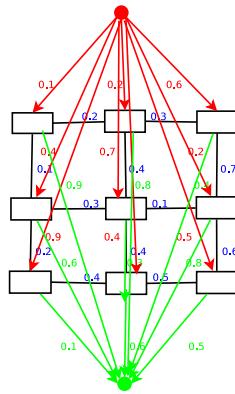


图 11.22: 将图片分割问题转化成网络流

其中, 对割, 我们举一个例子, 比如第二三行之间的一个割, 我们认为上方的都是前景, 下方的都是背景的. 考察这个割所割的各种颜色的边的数目, 结合目标函数 $Q(F, B)$ 中计数的边的数目, 将对应颜色的边相加, 正好是所有边的数目, 是个常数, 如表 11.1 所示.

表 11.1: 割与目标函数对应的边的比较

	红边	蓝边	绿边
$C(F, B)$	3	3	6
$Q(F, B)$	6	9	3
$M = C + Q$	9	12	9

而我们想要最大化 $Q(F, B)$, 就等价于最小化 $C(F, B)$, 也就是再求最小割. 这样, 这个问题就可以用最大流来解了, 这是一个比较快的方法. 但是, 我个人并不是特别推崇这个方法, 这个东西有点巧妙, 想出来就想出来了, 想不出来还是想不出来, 启发

性不是很大. 我还是希望大家可以把这个问题写成线性规划或是二次规划. 事实上对之前提到的二次规划, 稍加技巧就可以把它写成现行规划.

工程选择问题

我们首先给出工程选择 PROJECT SELECTION 问题的定义

输入: 给定一个有向无环图 (DAG). 一个节点 i 代表一个工程, 完成这个工程将获得一定收益 (表示为 $p_i > 0$), 或者一定损失 (表示为 $p_i < 0$). 一条有向边 $u \rightarrow v$ 表示先序条件 (PREREQUISITES), 比如 v 应当在 u 之前完成 (即要想干 u 必须先干 v).

目标: 选择一个工程的子集 A 使得:

1. 可行: 如果一个工程被选中了, 那么他所有的先决工程也要被选中;
2. 最优: 最大化利润 $\sum_{v \in A} p_v$;

我们来看一个例子, 如图 11.23 左面所示, 这个大家可以看做找工作时一年挣 10 万, 但是得先交 3 年学费, 一年 1 万. 问题就是在满足先序关系的条件下选择一些工程, 使得赚得钱越多越好.

先看一个巧妙的办法, 这个问题有是让我们在一个集合中选一个子集, 这就是割. 我们把它转换成一个网络流的问题, 构造流网络, 如图 11.23 所示, 具体方法如下:

1. 网络: 原网络中没有源点和汇点, 引入连个节点: 源 s 和汇 t , s 指向所有赚钱 ($p_i > 0$) 的节点, 所有交学费 ($p_i < 0$) 的节点指向 t ;
2. 容量: 将顶点上的权值移到边上, 并设 $C(u, v) = \infty, \forall u, v \in E$.
3. 割: 顶点集的划分.

最终我们想说明最小割等价于最大利润. 图 11.24 显示了实例上的一个割.

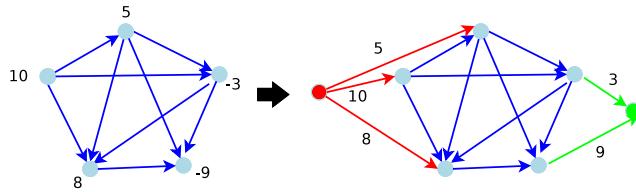


图 11.23: 构造流网络

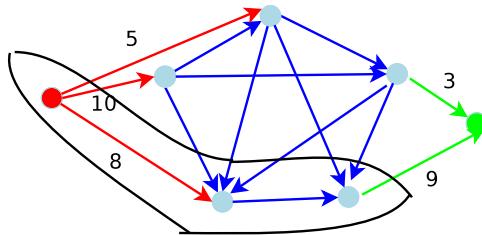


图 11.24: 流网络上的一个割, 表示我们选择下面连个工程

这个例子所示的割容量为 $C(A, B) = 5 + 10 + 9$, 而这个割表示选择下面两个 (即全出来的两个) 工程, 这个割对应的收益是 $\sum_{i \in A} p_i = 8 - 9 = -1$. 这两个数加起来是 $C = (5 + 10 + 9) + (8 - 9) = 5 + 10 + 8$, 是一个常数, 这个是所有能赚钱的项目之和, 就是图 11.24 左边三条红线上权值的和. 这个问题跟之前一个很像, 加起来等于常数, 要最大化收益, 只要最小化割就可以了. 对应于一般情况, 我们可以表述如下:

1. 割容量: $C(A, B) = C - \sum_{i \in A} p_i$, 其中 $C = \sum_{v \in V} p_v$ ($p_v > 0$) 是一个常数.
2. 最小割: 由于容量和利润的和是一个常数, 所以最小割对应于最大利润.

最后, 还有一点微妙的东西, 我们为什么要把内部的边容量设成是 \inf 呢? 因为我们要保证这个解是可行的. 我们考察一个如图 11.25 所示的割, 在这个割中, 工程 u 被选中了, 但他的前导工程 v 没有被选上, 也就是说, 这个割是不可行的. 而在正常情况下, 那么边 $< u, v >$ 就是一个割边, 这将使得我们得到的割容量为无穷. 这也就意味着, 求最小割我们是永远求不出这种情况的. 这也就强迫我们选取一个工程的前导工程, 这也是一个比较巧妙的地方.

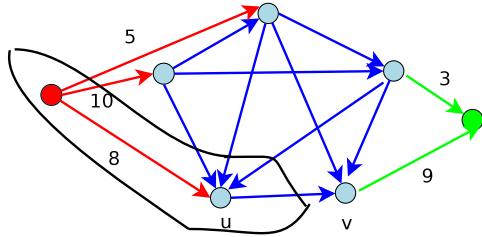


图 11.25: 一种不可行的割

那么, 这个设计这么巧妙, 想不出来怎么办? 我们还是回到线性规划上来. 我们对每一个工程设一个 x_i 表示该工程选还是不选. 而对先序关系, 比如完成 x_1 需要先完成 x_2 , 我们有逻辑表达式: IF x_1 THEN x_2 , 这个等价于 $x_2 \geq x_1$, 其他的边也同理. 这样对例子中的问题, 我们有以下的线性规划表达式:

$$\begin{aligned} \text{MAX } & 10x_1 + 5x_2 - 3x_3 - 9x_4 + 8x_5 \\ \text{S.T. } & x_i = 0/1, i = 1, \dots, 5 \\ & x_2 \geq x_1 \\ & x_3 \geq x_1 \\ & \dots \end{aligned}$$

这样, 如果大家觉得之前的精巧的做法想不到, 我们还可以用如上的线性规划来求解.

11.4.2 寻找路径

之前我们已经讲了: 割就是把一个集合分成两堆. 那么流呢? 流就是从 s 到 t 经过一堆路径. 所以如果让我们从 s 到 t 找一堆路径的话, 我们可以想到最大流最小割.

不相交路径问题

我们来看一个问题: 不相交路径. 定义如下:

输入: 给定一张图 $G = \langle V, E \rangle$, 两个顶点 s 和 t , 一个整数 k .

目标: 指定 k 个互不相交的 $s - t$ 路径;

图 11.26 展示了不相交路径问题的一个实例, 在这个例子中, 最多可以找到 4 条从 s 到 t 的不相交路径。这个问题与图连接问题也有关。

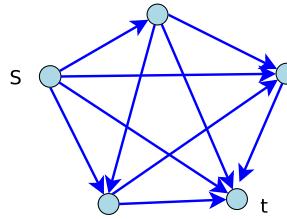


图 11.26: 不相交路径问题的一个实例

为了用网络流解决这个问题, 我们构造网络如下:

1. 边: 跟原图一样 (原图已经天然给定了 s 和 t);
2. 容量: $C(u, v) = 1$;
3. 流: 如下所述

注意, 每条边的流量都是 1, 这就限制了每条边只能走一遍, 这就限制了各个路径不会有重叠 (即相交)。

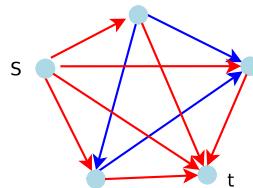


图 11.27: 例子中得到的 4 个不相交的路径

定理 15. 图 G 中存在 k 割不相交的路径 \Leftrightarrow 最大的 $s - t$ 流值至少是 k .

Proof. 1. 注意: 最大 $s - t$ 流值是 k 意味着这是一个值为 k 的整数流 (不会出现 0.5 这样的非整数解)。

2. 选择 $f(e) = 1$ 的边就行.

□

关于时间复杂度, FORD-FULKERSON 算法的复杂度为 $O(mC)$, 而这里的 C 最多是 n , 所以这个算法的时间复杂度为 $O(mn)$.

上面这个问题很简单, 我们介绍一点扩展, MENGER 在 1927 年做了一个定理, 这个定理就是说不相交路径的最大数目等于我们小时候做的一个游戏: 把这些边想象成火柴棍, 拿掉几根以后 s 和 t 就不通了. 最小的火柴棍的数目等于最大的不相交路径数目, 形式化的表述如下:

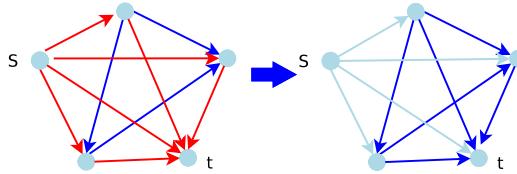


图 11.28: •

定理 16 ((Menger theorem)). 最大不相交路径的数目等于将 s 和 t 分隔开所需要删除的最小边数.

Proof.

1. 最大不相交路径的数目等于最大流;
2. 存在一个割 (A, B) , 使得 $C(A, B)$ 等于不相交路径的数目;
3. 这个割的割边就是我们想要的那些边.

□

调查问卷设计

寻找路径问题还有一个应用, 调查问卷的设计, 定义如下:

输入: 顾客的集合 A , 产品的集合 P . 令 $B(i) \subseteq P$ 表示顾客 i 购买的产品. 一个整数 k .

目标: 设计一个有 k 个问题的调查问卷, 使得对顾客 i , 问题的数量至少是 c_i , 至多是 c'_i . 另一方面, 对每种产品, 问题的数目至少是 p_i , 至多是 p'_i .

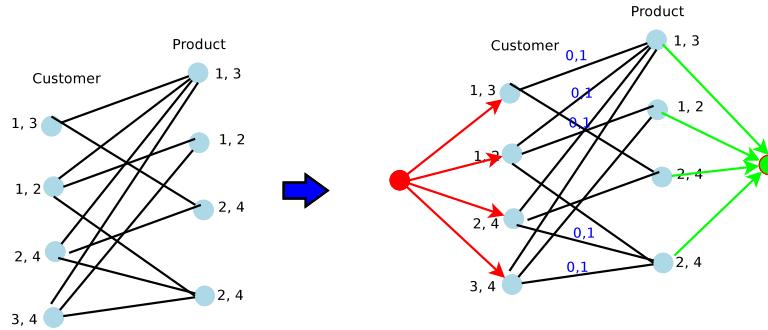


图 11.29: 将调查问卷设计问题转换为网络流问题

接下来我们建立这个问题对应的流网络:

1. 边: 引入两个节点 s AND t . 将顾客连接到 s 并将产品连接到 t . s 点可以看所商场发信部门, 把问题发送到顾客, 而 t 点相当于商场的收件箱.
2. 容量: 将节点上的权值移到边上; 并令中间这些边的权值均为 $C(i, j) = 1$;
3. 流通: 一个原问题的可行解.

老样子, 这个问题也一样可以用线性规划求解, 我们这里就不多说了. 这个问题也很有用, 将来大家工作如果到商场电信部门, 估计就让大家干这事.

11.4.3 匹配

二部图匹配

我们首先来看最简单的二部图匹配:

输入: 二部图 $G = \langle V, E \rangle$;

目标: 找出最大匹配;

一个常用的比喻是：如图图 11.30 所示，左边是男士，右边是女士，如果互相喜欢就连一条边，如果不喜欢就不连边。我们就相当于红娘，问最多能成几对。

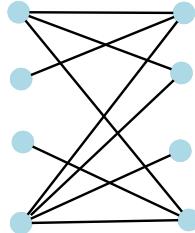


图 11.30：一个二部图匹配的例子

对于这个问题，我们有一比较简单的做法。按如下方式建立流网络，如图 11.31 所示

1. 边：增加两个节点 s 和 t ；将 s 连接到 U 并将 t 连接到 V ；
2. 容量： $C(e) = 1, \forall e \in E$ （这就保证了中间两排节点，每个节点最多使用一次）；
3. 流：最大流对应于最大匹配；

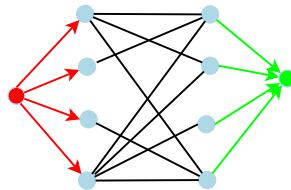


图 11.31：二部图匹配问题对应的流网络

算法时间复杂度为 $O(mn)$

完美匹配

上面我们已经解决了二部图上的最大匹配，二部图上的匹配是很简单的。但即便是对于二部图，也有不是那么简单的东西：HALL 在 1935 所给出的定理 (KONIG-EGERVARY 在 1931 年给出)。我们首先给出完美匹配的定义 (简单来讲就是所有人都有对象)，一个例子如图 11.32 所示。

定义 4 (完美匹配). 给定一个二部图 $G = \langle V, E \rangle$, 其中 $V = X \cup Y$, $X \cap Y = \emptyset$, $|X| = |Y| = n$. 一个匹配 M 是完美匹配当且仅当 $|M| = n$.

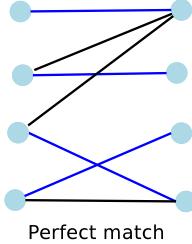


图 11.32: 一个二部图完美匹配的例子

而这个定理如下所述, 这个定理不是很显然.

定理 17. 一个二部图存在完美匹配 \Leftrightarrow 对任意 $A \subseteq X$, $|\Gamma(A)| \geq |A|$, 其中 $\Gamma(A)$ 表示 A 中匹配节点的集合.

用红娘的例子来说, 对一个如图 11.32 所示二部图, 假设左边是男士, 右边是女士, 每个人都有对象当且仅当: 任意挑一些男士 (即 A 集合), 它们喜欢的女士 (即 $\Gamma(A)$) 的数量大于等于这些男士的数量.

这个定理反过来很好理解, 不如图 11.33 右边所示, 三位男士喜欢两位女士, 那肯定有一个人打光棍, 也就是不存在完美匹配. 正过来就不好理解了, 为什么所有集合的男士喜欢的女士比男士多的话就存在完美匹配? 所有男士的集合有 2^n 种, 不是那么好理解.

Proof. 我们只证明如果存在完美匹配, 那么 $|\Gamma(A)| < |A|$.

1. 假设不存在完美匹配, 也就是说, 最大匹配 (成的对数) 是 M 且 $|M| < n$;
2. 我们考虑之前的流图, 考察它的最大流和最小割, 那么肯定存在一个割使得 $C(A', B') < n$. 定义 $A = A' \cap X$. 这个仅仅是把不存在完美匹配翻译成最大流的语言;

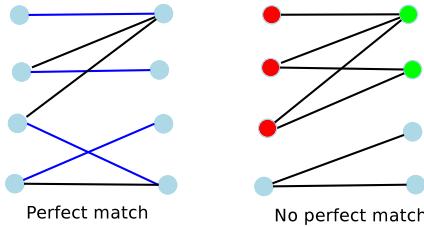


图 11.33: 左侧的例子存在完美匹配, 而右侧的不存在完美匹配

3. $C(A', B') = |X \cap B'| + |Y \cap A'| = n - |\Gamma(A)|$, 如图 11.34 所示.

4. 因为 $C(A', B') < n$, 所以我们有 $|\Gamma(A)| < |A|$.

□

注意, 如果有必要, 我们可以改变 A' 使得 $\Gamma(A) \subseteq A'$. 这个算法时间复杂度为: $O(mn)$

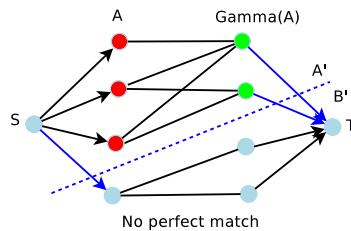


图 11.34: 图 11.33 中右图 (不存在完美匹配) 对应的网络图

也就是说, 这个定理告诉我们, 给了一堆喜欢的关系, 我们没有找到完美匹配, 原因就是其中几个人喜欢的人数太少了, 我们用上面的方法还能找到到底是哪些人. 这个定理很好理解, 但是它很深刻.

11.4.4 数字分解

还有什么问题可以让我们想到最大流最小割呢? 把一个数字拆分. 一个应用是 BASEBALL ELIMINATION 问题. 不知道大家现在看不看足球, 有时当一个赛季快结束

时, 有的球报上居然会出数学公式, 推算什么什么队 (比如北京国安队) 与冠军无缘. 定义如下:

输入: n 个球队 T_1, T_2, \dots, T_n . 一个球队 T_i 已经赢了 w_i 场球, 对 T_i 球队和 T_j 球队, 还有 g_{ij} 场球要踢.

目标: 我们是否可以确定一支球队, 比方说 T_i , 已经无法赢得冠军? 如果可以赢得冠军, 我们能给出证据吗?

下面我们给一个例子: 一共有 4 只球队: *New York, Baltimore, Toronto, Boston*, 它们的 w_i 和 g_{ij} 如下:

1. w_i : 纽约 NY(90), 巴尔迪莫 BALT(88), 多伦多 TOR(87), 波士顿 Bos(79).
2. g_{ij} : NY:BALT 1, NY:TOR 6, BALT:TOR 1, BALT:Bos 4, TOR:Bos 4, NY:Bos 4.

赛程过半, 我们就可以得到以上的数据, 这时我们已经可以肯定 *Boston* 已经无法取得冠军了, 因为一方面, *Boston* 最多可以以 $79 + 12 = 91$ 场胜利结束比赛 (已经赢了的加上剩下的). 另一方面, 我们可以找到一个球队的子集 (先不管怎么找的), 比如说 $\{NY, Tor\}$, 子集中球队结束时胜利的总数为 $90 + 87 + 6 = 183$, 因此结束时至少有一支球队赢了 $\frac{183}{2} = 91.5 > 91$ 场球. 注意, 另一个子集 $\{NY, Tor, Balt\}$ 就无法作为 *Bos* 已经无法取得冠军的证据.

我们可以把 BASEBALL ELIMINATION 问题形式化地表述如下:

问题: 对一个特定的球队 z . 我们能否确定是否存在一个球队的子集 $S \subseteq T - \{z\}$, 使得:

1. z 可以以最多 m 场赢球结束所有比赛;
2. $\frac{1}{|S|}(\sum_{x \in S} w_x + \sum_{x,y \in S} g_{xy}) > m$;

换句话说, S 集合中至少有一只球队赢场多余 z .

我们以 $z = Boston$ 为例来展示如何使用网络流解决这个问题. 首先, 我们令 $m = w_z + \sum_{x \in T} g_{xz} = 91$, 也就是 z 球队可能赢的总场数. 接下来, 我们按如下的方式构造网络, 如图 11.35 所示:

1. 令 $S = T - \{z\}$ 且 $g^* = \sum_{x,y \in S} g_{xy} = 1 + 6 + 1 = 8$.
2. 节点: 对每两支球队, 构造一个节点 $x : y$ (左边那三个点), 对每个球队 x , 构造一个节点 x (右边三个点).
3. 边:
 - s 指向 $x : y$, 设置容量为 $g_{x,y}$. 假设赢一场一分, 那么 $s -> \{NY:TOR\}$ 就相当于从 s 发射 6 分.
 - 连接 $x : y - x$, 连接 $x : y - y$, 容量都设为 $g_{x,y}$. 对 $\{NY:TOR\}$ 节点, 它们一共要踢 6 场(赢 6 分), 它一定要分解成 $\{NY:TOR\}$ 到 NY 和 $\{NY:TOR\}$ 到 TOR .
 - 连接 $x - t$, 容量设为 $m - w_x$. 比如 NY 到 t 的边, 纽约已经赢了 90 场了, 再赢 1 场就比波士顿最多可能赢得要高了.

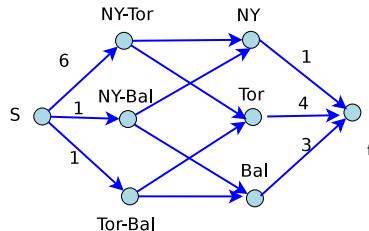


图 11.35: BASEBALL ELIMINATION 问题的网络构造

首先, 直觉上考虑, 沿着边 $s - x : y$, 我们运送 $g_{x,y}$ 次赢场. 在节点 $x : y$, 这个数被分解成了两个数, 也就是两只球队的赢场. 这样, 这个问题就是一个数字分解问题.

情况 1: 最大流的值是 $g^* = 8$

定理 18. 网络中存在一个值为 $g^* = 8$ 的流当且仅当球队 $z = Boston$ 还有可能赢得冠军.

Proof. • \Leftarrow 如果有一个值为 g^* 的流, 那么边 $x - t$ 上的容量保证了没有球队可以赢得超过 m 场球. 因此, 球队 z 还有可能赢得冠军 (只要 z 赢得剩下的所有比赛).
• \Leftarrow 如果球队 z 还有可能赢得冠军, 我们总可以找到一个值为 g^* 的流.

□

情况 2: 最大流的值小于 $g^* = 8$

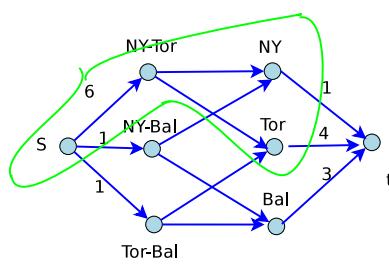


图 11.36: 第二种情况

定理 19. 如果最大流值严格小于 g^* , 最小割描述了一个子集 $S \subseteq T - \{z\}$, 其使得 $\frac{1}{|S|}(\sum_{x \in S} w_x + \sum_{x,y \in S} g_{xy}) > m$.

Proof. (See extra slides)

□

匹配问题还有很多扩展, 比如: 指派 (ASSIGNMENT) 问题, 带权指派 (WEIGHTED ASSIGNMENT) 问题的 HUNGARIAN 算法, 开花 (BLOSSOM) 算法.

第十二章 问题的难解性

12.1 NP 问题和难解性

12.1.1 问题和它的难度

本节课总共要讲两件事情：

1. 难度是问题本身固有的属性，复杂度是算法的复杂度，没有问题的复杂度这个概念。
2. 归约：比较两个问题谁比谁更难

问题分类：

1. 简单求解问题（存在多项式时间的算法，比如 STABLEMATCHING）
2. 不可能求解问题（HALT 问题）
3. TRULY HARD 问题（存在算法可求解问题，但可证明不存在多项式时间的算法，只存在指数时间算法）即给定图灵机，能否在 K 步内停止。
4. NP-HARD 问题（问题很难，但是不能证明它很难，只能证明它的相对难度）

12.1.2 ”问题”的定义

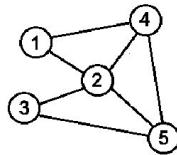
我们把问题这么来表示：一个问题经过数学建模后，抽象出的数学问题，必定包含一个 INPUT 及 OUTPUT，最终我们想问，对于这个问题任何一个 INPUT，能够给出一个 OUTPUT。一个具体的 INPUT 称为一个实例。

例 1：s-t 连通性问题

输入：一个图 $G = \langle V, E \rangle$, 两个节点 s , 和 t ;

输出：一条路径 s 到 t ; (或者返回 “NO” 如果图中不存在这样的路径);

具体实例：一个五个点的图，连通性如下图所示，问存不存在 s 到 t 的一条路径



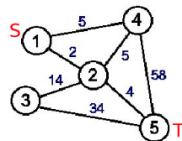
因此，PROBLEM 是抽象的表示，INSTANCE 是具体的实例。

我们一般会碰到两类问题，第一类是优化问题：对这个问题当中任何一个实例 (INSTANCE) $x \in I$ ，我们找一个最优的解 (SOLUTION) y^* ，如最短路径问题。

例 2：最短路径问题

输入：一个图 $G = \langle V, E \rangle$ ，两个将节点， s 和 t ；

输出：从 s 到 t 的最短路径，或者返回 “NO” 如果图中不存在 s 到 t 的路径；

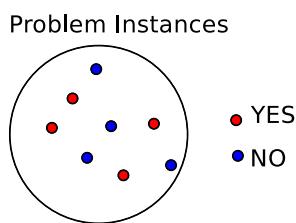


我们遇到的很多问题都是优化问题，但是优化问题在分析难度时并不容易，因为优化问题需要得到目标函数值。而这里存在一个比较好分析的问题类型，称为判定问题：对于任何一个问题的实例 (INSTANCE) 答案只有 YES, NO。

例 3：路径问题 (path problem)

输入：一个图 $G = \langle V, E \rangle$ ，两个节点 s 和 t ，以及 一个整数 k ；

输出：一条从 s 到 t 的路径，并且长度最大为 k ；



如果我们能找到这条路径回答 YES，否则回答 NO。我们通过观察可以发现，

为优化问题添加一个限制条件，如最大长度不超过 k 等，可以将优化问题转换为判定问题，这样答案简单，便于分析。对于判定问题我们很容易进行一些描述。对例 3，我们知道问题就是一堆实例的集合，我们把所有的是实例拿过来放在这里，形成一个文氏图，每个点代表一个实例（INSTANCE）。针对每个实例，答案只有两种（YES OR NO），这比优化问题描述起来简单多了。

12.2 归约

现在我们对问题怎么表示已经很清楚了，就是把 INPUT 及 OUTPUT 描述清楚就行。这个对于同学们没有什么困难，接下来需要考虑的是问题之间的难度关系怎么比较，怎么说一个问题比另外一个问题更难呢？这个办法叫做归约，它的目的是揭示两个问题之间的关系，就是比较两个问题谁难谁容易。我们看一下详细的归约是什么意思。

12.2.1 多项式时间归约

多项式时间的归约是一个变换的过程，我们记作 f ，它把问题 A 的任何一个实例（INSTANCE） α 都能快速变成问题 B 的一个实例 $\beta = f(\alpha)$ ，变换 f 包含两点要求：

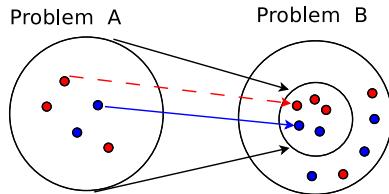
1. 快速变换：变换过程只需要花费多项式时间。
2. 等价性：变换前后答案需要相同。（如果实例 α 答案是 YES，则实例 $\beta = f(\alpha)$ 答案也应为 YES。）

我们将满足上述两点要求的过程记作 $A \leq_P B$ ，读作 “**A is reducible to B**”。

那么上面的符号都代表什么意义呢？ P 是 POLYNOMIAL，代表多项式时间， \leq 看作两个问题之间的难度关系，如 $A \leq_P B$ 表示 B 比 A 更难。

12.2.2 多项式时间归约的图形表示及其意义

上述多项式时间归约的定义可以用如下图所示的文氏图来表示：



问题 A 可以用一个圈表示一个集合，里面包含问题 A 的所有实例，每个实例的答案只有 YES 或 NO，归约就是能够把问题 A 中的任意一个实例变作问题 B 中的一个实例，这样我们就把 A 中的每个实例映射到 B 中的实例上，值得注意的是，如果 A 的某个实例答案是 YES，则相应的映射到 B 中的实例答案也必须是 YES，A 中实例答案是 NO 则映射的实例的答案也必须是 NO。

到现在为止我们只介绍了问题之间怎么进行归约，接下来将要解释为什么归约之后就可以认为 B 比 A 更难。

定理 1：如果问题 B 能在多项式时间内求解，则问题 A 也能在多项式时间内求解。

我们可以证明这个定理，但是我们先看它反过来的定理。

定理 2：相反的，如果问题 A 是难的，则问题 B 也是难的。

所以我们可以说明 B 比 A 难，那么为什么说定理 1 是对的呢？假如说我们对问题 B 有个快速求解的算法，则我们能得到一个 A 的快速求解算法，即给定一个问题 A 的实例 α ，首先通过多项式时间归约将实例 α 转换成问题 B 的实例 $\beta = f(\alpha)$ ，运行问题 B 的快速求解算法我们就可以得到实例 β 的答案。最后根据等价性，可以通过 β 的答案得到问题 A 的实例 α 的答案。

抽象的东西讲的太多了，接下来看一下实际的例子。

12.3 归约举例

在分析这个问题之前插一句话，今天这堂课非常强调大家数学建模的能力，强调数学建模能力有三个地方。第一个是线性规划，我们需要将一个实际问题抽象成一个线性规划，第二个是网络流，很多问题可以转换成网络流问题，第三个就是将要讲的这个。我们首先训练一下如何将一个实际问题抽象出来。所以接下来大家会看到很多

的名词和很多问题的抽象定义，但是大家不要怕，我们都会讲一个实际的问题是什么。

12.3.1 INDEPENDENTSET \leq_P VERTEXCOVER

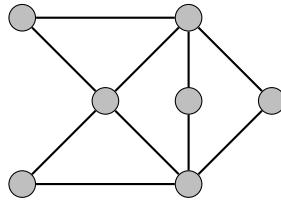
独立集问题

我们先看什么是独立集问题，先不管抽象的定义，先看实际的需求，假设我们有 n 个朋友，但是有些朋友之间相处不好，容易发生冲突。那么如何在避免人际关系紧张的前提下邀请至少 k 个朋友吃饭。我们将这个实际问题抽象成一个数学问题。

输入： 给定一个图 $G = \langle V, E \rangle$ ，和一个整数 k ；

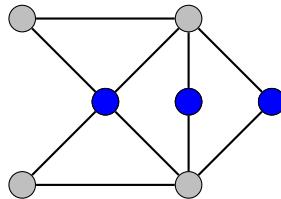
输出： 是否存在一个点的集合 $S \subseteq V$, $|S| = k$, 使得集合 S 中任意两点都没有边相连？

我们用每个点表示一个人，如果两个人关系不好则在图中用一条边连接两个点，则可以用下图表示 7 个人的人际关系图，问如果要请 3 个人吃饭，请哪些人。



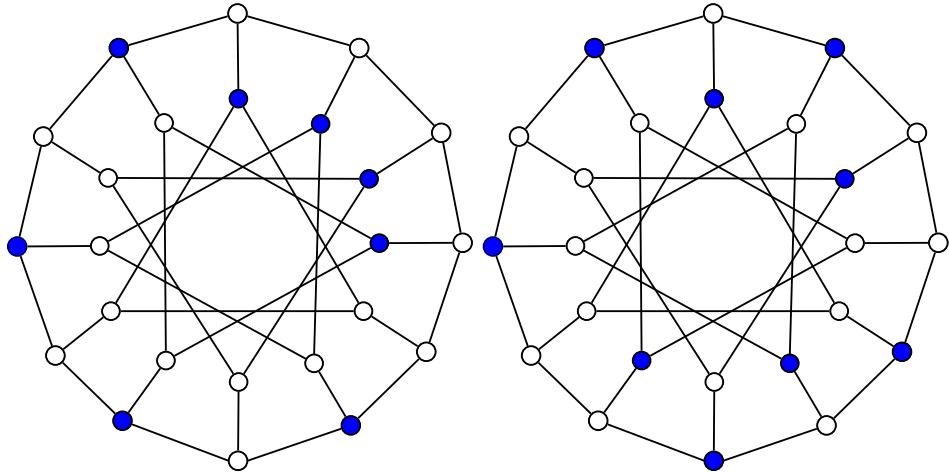
问题很清楚了，就是给我们一个图，在图中找 k 个节点，使得各个节点间没有边连接，我们称这样的点的集合为独立集，独立的意思是说两个点之间没有边。我们看看其他的情形

$k = 3$: 蓝色标示的三个节点是独立的。



上面这个情况大家一眼就可以看出来，那么出现如下所示的情形呢。

$k = 8, k = 9$: 蓝色的节点形成一个独立集, 左边 8 个节点, 右边 9 个节点。



左边的图是非常非常有名的图, 让我们在中间请 8 个人吃饭, 怎么请。这幅图是这么画的, 外面是 12 个点, 如同时钟, 里面也是 12 个点, 每个都连接一条边, 里面的点之间画了三角形, 总共 4 个三角形。这幅图可以表示请八个人吃饭, 也可以表示请九个人吃饭。我们可以验证, 蓝色的任意两个点之间不存在边连接。大家先不慌问怎么找出来的, 这个不好找。这个图找 10 个人就不可能了, 虽然你有 24 个点, 找不出 10 个人。

顶点覆盖问题

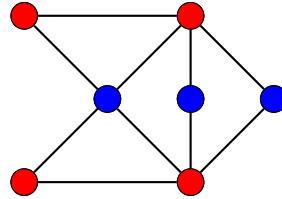
介绍完独立集问题再来介绍顶点覆盖问题。那么什么是顶点覆盖呢, 还是先不看抽象的定义, 先看现实的一个例子。我们有 N 个地点, 每个地点之间有一条道路, 问如何挑选地点安装摄像头使得我们可以监控所有的道路。这是一个很实际的例子, 我们不可能给所有的道路安装摄像头, 肯定是要尽可能少的使用摄像头。

我们可以把这个实际问题抽象成一个数学问题。

输入: 给定一个图 $G = \langle V, E \rangle$, 和一个整数 k ;

输出: 是否存在点的集合 $S \subseteq V$, $|S| = k$, 使得任意一条边至少能被一个节点覆盖 (COVER) ?

以下图为例, 我们可以找到 4 个节点 (用红色标注) 覆盖所有的边。



归约——变换

那么这两个问题我们解释的很清楚了，一个问题是请客吃饭，一个问题是装摄像头。我们为什么可以说装摄像头问题比请客吃饭问题要难呢？要证明这个观点，我们需要构造一个变换，使得独立集问题中的每个实例 $\langle G, k \rangle$ 都可以转换成顶点覆盖问题中的实例 $\langle G', k' \rangle$ ，并且对应问题的答案一致。

接下来我们看怎么构造这样的变换。原先的请客吃饭问题（独立集问题）给了我们一个人际关系图，每个节点代表一个人，假如左边的图代表请客吃饭问题的一个实例，我们把它变成如右图所示的装摄像头问题的一个实例。请客吃饭问题的实例是一个人际关系图，装摄像头问题的实例是地点和道路之间的关系图。

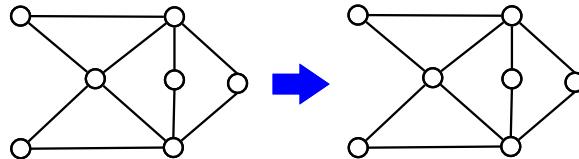


图 12.1: Transformation from an INDEPENDENT SET instance ($G, k = 3$) into a VERTEX COVER instance ($G' = G, k' = 4$)

那么我们怎么构造一个装摄像头问题的关系图呢，针对这个问题很简单，直接将人际关系图复制作为装摄像头问题的关系图。但是我们需要注意，问题里不仅需要定义一个图，还需要定义一个参数 k ，那么 k 怎么定义呢。对于原始的请客吃饭问题是需要我们请 k 个人，而装摄像头问题需要我们找 $n - k$ 个地方装摄像头。

正如图中的那句话所说：变换过程将独立集问题变换成顶点覆盖问题，并且图还是那个图，但是独立集问题 $k = 3$ ，顶点覆盖问题 $k' = n - k = 7 - 3 = 4$

归约——等价性

现在变换过程有了，我们需要证明等价性。

定理：图 G 有独立集 S ($|S| = k$) $\Leftrightarrow G'$ 有顶点覆盖集 S' ($|S'| = n - k$).

用通俗的话说就是，我们能请 k 个人吃饭，当且仅当我们能在装摄像头问题中找到 $n - k$ 个节点装摄像头覆盖到所有的边。

为什么呢，我们这么看。假如说对原始的请客吃饭问题，我们回答一个 YES，即能找到 k 个人一起吃饭，且不会发生冲突，我们把这些人叫做 s (蓝色标注)。如下图中找到的三个人不会发生冲突，因为对于任意的一条边 $e = (u, v)$ ，假设两个端点是 u 和 v ，则两个端点不会同时在 S 中，因此可以得到 $u \in V - S$ 或者 $v \in V - S$ 。我们定义 $S' = V - S$ (红色标注)，我们就在这些地方装摄像头。在图中可以表示我们请蓝色节点代表的朋友吃饭，而在剩下的红色节点代表的地方装摄像头覆盖所有的边。因为蓝色节点之间没有边，则剩下的红色节点肯定可以覆盖所有的边。

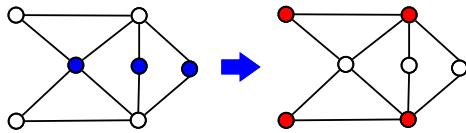


图 12.2: The complement of an independent set(in blue) form a vertex cover (in red)

在构建快速的变换过程以及证明等价性之后，我们就证明了装摄像头问题（顶点覆盖问题）比请客吃饭问题（独立集问题）要难。实际上请客吃饭问题就足够的难，它属于 NP 类问题中最难最难的一类问题。我们也可以证明独立集问题比顶点覆盖问题要难，即两个问题的难度系数是一样的，这个问题我们以后再说。

刚才的例子让我们知道了怎么说明一个问题比另一个问题更难。但是因为分析问题的复杂度有时候会比较麻烦，因此这个例子的分析可能会给大家带来困扰。我们来看看牛人 C. PAPADIMITRIOU 是怎么分析问题的难度，怎么做归约的：

要证明一个问题 NP 完全的，我们可以从问题的简单实例开始仔细琢磨，直到我们找到一个小例子，它表现出非常有趣的行为。有时候，对这个例子的观察可以直接得到一个 NP 完全问题的简单证明.... 有时候称其为“机关建设”。

(EXCERPTED FROM COMPUTER AND COMPLEXITY)

12.3.2 VERTEX COVER \leq_P SET COVER

再来看一个简单的归约，集合覆盖问题比顶点覆盖问题还要难。

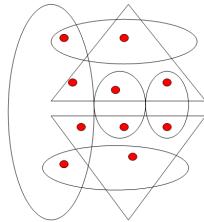
集合覆盖问题

对于集合覆盖问题，我们来看一个实际例子。IBM 公司要开发一个杀毒软件，杀毒软件识别病毒是通过检查代码中是否存在某些关键字，每种病毒都有特殊的关键字，但是任何一种关键字可能对应很多种病毒。为了降低杀毒软件病毒库的大小，我们感兴趣的是能不能找出有代表性的关键字，而不是每种病毒都单独对应一个关键字。我们来看一下它抽象成什么问题。

输入：一个集合 U 有 n 个元素，一些 U 的子集 S_1, S_2, \dots, S_m ，和一个数 k .

输出：能否找到 k 个子集，使得它们的并集等于全集 U ?

我们来举个简单的例子解释这个问题。

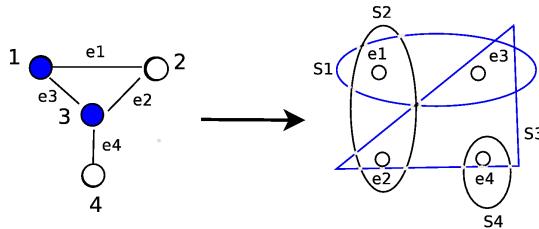


在这个实例中我们假设有 10 个病毒，每个病毒用一个红点表示，每个框表示一个关键词，表示包含在框内的病毒共享这个关键词，扫描病毒时发现这个关键词，我们就知道是框内的某个病毒出现。我们的目标是找尽可能少的子集使得所有的病毒都包含在这些框内，这样我们就能减少病毒库的规模，而在这个例子中我们能找到最少 3 个集合使得它们的并集能够包含所有红点。而这个就是集合覆盖问题中“覆盖”一词的实际意义，即子集的集合能将所有红点包含在内。

在弄清楚集合覆盖问题后，我们需要证明集合覆盖问题比顶点覆盖问题要难。

归约——变换，等价性

我们首先构造一种变换，即任何一个道路和地点的连接图我们总能变换为杀毒的事情，怎么变。假如原先的地方有四个地点，连接图如下图左边所示，令 $k = 2$ ，则我们可以找出两个地方覆盖所有的道路。我们就把这个实例变换成杀毒的问题的一个实例，即连接图中的每条边对应一个病毒（右图中的点）。左图中 1 号顶点能覆盖边 e_1, e_3 两条边，则在右图中构造一个集合覆盖顶点 e_1 和 e_3 ，同理左图中 3 号顶点能覆盖边 e_2, e_3 和 e_4 ，我们同样在右图中构造一个集合覆盖这些点，以此类推。所以如果我们能在左图中找到两个点覆盖所有的道路，则必定可以在右图中找到两个集合覆盖所有的病毒（点）。因此如果我们顶点覆盖问题的实例回答 YES，则我们同样可以在集合覆盖问题中找到相应的解，使得答案同样是 YES，使得该变换满足等价性。



上面这个简单的例子是让大家熟悉一下什么时候归约，下面我们讲一个稍微难点的例子。

12.3.3 通过“Gadget”归约: $3\text{-SAT} \leq_P \text{INDEPENDENT SET}$

观察了两个简单的归约之后，我们来看一个不是那么直接的归约问题，独立集问题比 3-SAT 问题更难。

SAT (Satisfiability) Problem

什么是 3-SAT? 3-SAT 问题是数理逻辑中很重要的一个问题，很基本的一个问题。很多实际的问题都能转换成 3-SAT 问题，它的全称是可满足性问题。在人工智

能领域表达一组变量的约束，或者在超大规模集成电路测试时验证电路是否符合需求方面等问题都可以转换成 3-SAT 问题。

我们来看一下可满足性问题的定义。这个定义稍微有点繁琐，大家耐心往下看。

输入： 给定一个合取范式 (CNF) $\phi = C_1 \wedge C_2 \dots \wedge C_k$;

输出： 是否存在对所有变量 x_i 的一个合适赋值，使得所有的子句 C_j 都是 TRUE(可满足)?

注意：

1. 布尔变量: x_1, x_2, \dots, x_n , $x_i = \text{TRUE/FALSE}$;
2. 文字 (项) : 一个变量 x_i , 或者它的非 $\neg x_i$, 表示一个文字;
3. 子句: 一些文字的或: $C_1 = x_1 \vee \neg x_2 \vee \dots \vee x_3$;
4. 合取范式 (CNF) : 一系列子句的与 $\phi = C_1 \wedge C_2 \dots \wedge C_k$;

我们来看一个例子来理解这个定义。

CNF: $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2 \vee \neg x_3)$;

TRUE ASSIGNMENT: $x_1 = \text{FALSE}, x_2 = \text{FALSE}, x_3 = \text{FALSE}$;

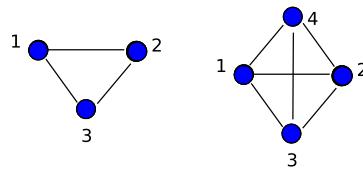
上面这个例子的合取范式由三个子句组成，第一个子句是 $x_1 \vee \neg x_2$ ，第二个子句是 $\neg x_1 \vee \neg x_3$ ，第三个子句是 $x_2 \vee \neg x_3$ 。三个子句之间用 \wedge (与) 进行连接，这就形成了一个输入。因此我们的目标对布尔变量的值进行一种组合，使得三个子句都等于 TRUE。而当 $x_1 = \text{FALSE}, x_2 = \text{FALSE}, x_3 = \text{FALSE}$ 时我们可以验证三个子句均等于 TRUE，从而合取范式也等于 TRUE，即这个输入是可满足的。

我们对 SAT 问题有两种看法，让我们找一堆 TRUE 和 FALSE 的赋值组合，使得整体等于 TRUE，这个求解过程可以从两个角度来思考。第一个角度是从变量的角度来思考，我们可以尝试变量为 TRUE 或 FALSE，从而找到一种组合使得所有子句都等于 TRUE；第二个角度是从子句的角度来思考，我们假设每个子句都等于 TRUE，则我们从每个子句中选择一个或多个布尔变量为 TRUE，使得该子句为 TRUE（因为子句是一些文字的或，所以只要有一个文字为 TRUE 则该子句为 TRUE），但是我们从每个子句中挑选的布尔变量之间不能发生冲突（如第一个子句 $x_1 = \text{TRUE}$ ，则第二个子句不能选择 $\neg x_1 = \text{TRUE}$ ）。

SAT 问题与独立集问题之间的联系

谈到冲突，大家可能联想到请客吃饭的问题，实际上与请客吃饭问题是类似的。有了这点知识，我们就可以证明请客吃饭比 SAT 问题要难。

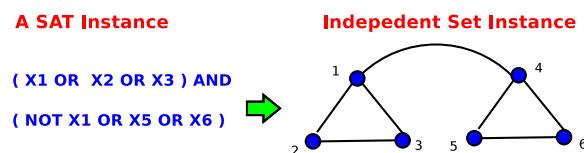
在证明过程中我们需要一点精巧设计的零件 (GADGET)。在这里我们先回顾一下 GADGET 的定义：一个很小的，非常有用的，精巧设计的机关或工具；有明确定义的功能，可以用来模拟另外一个问题。例如我们可以人际关系图来模拟 SAT 问题中的变量或者子句。在请客吃饭问题中，输入是一个人际关系图，在这里我们设计一个非常巧妙的人际关系图，就是一个三角形，三个人之间任意两个之间都有矛盾，所以如果我们请这三个人吃饭，则只能选择其中一人，右图中也是同样的。



大家注意我们刚才的描述中就暗含了 *or* 这种关系在里面。事实上，在 SAT 问题中子句内的变量是通过或进行相连，即逻辑关系中核心是或关系。而在人际关系图中，我们同样可以在图中表示或关系，即在左图中我们可以**选择 1 或选择 2 或选择 3**，所以实质上是一样的。换句话说如果我们画一个三角形就能模拟出 OR 的关系了，所以逻辑关系我们可以用逻辑变量来表示，也可以用画一幅图来表示。

归约——变换

基于这个认识，我们就可以把 SAT 问题的任何一个实例变成一个人际关系图。假如一个 SAT 实例是 $(x_1 \vee x_2 \vee x_3) \text{ AND } (\neg x_1 \vee x_5 \vee x_6)$ ，所以我们需要找一个对变量的赋值方案，使得第一个子句为 TRUE，第二个子句也为 TRUE。我们可以把它表示成如下图所示的独立集问题。

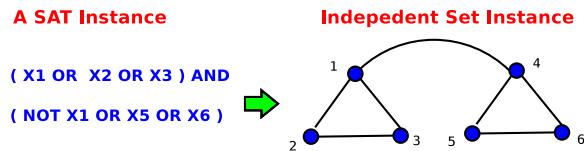


我们来观察一下是怎么变换的，在这边 x_1 对应人际关系图中 1 号点， x_2 对应 2 号点， x_3 对应 3 号点，然后我们先添加一个 4 号点，然后 x_5 , x_6 分别对应一个点。需要注意的是，在右图中画了两个三角形，以第一子句为例，子句的逻辑关系是 x_1 或 x_2 或 x_3 ，而右图第一个三角形同样表示要么选 1 号，要么选 2 号，要么选 3 号。另外 x_1 对应 1 号，而 x_4 对应 $\neg x_1$ ，所以在右图中我们可以在 x_1 和 x_4 之间连一条边，表示要么选 x_1 ，要么选 x_4 ，即如果 $x_1 = \text{TRUE}$ 则 $\neg x_1 = \text{FALSE}$ 。

我们从分析中可以看出来，这样的变换是非常简单的，每个子句对应一个三角形，同时在 x_i 和 $\neg x_i$ 之间连一条边。

归约——等价性

现在我们就可以证明这个问题了。假如说在 SAT 问题中存在一种赋值安排，使得问题的答案为 TRUE，则在请客吃饭问题中肯定可以找两个人吃饭，而不会发生冲突。



我们来看看证明为什么是这样的。假如这个例子是可满足的，则必定存在一种赋值安排使得每个子句都等于 TRUE(只安排一个文字等于 TRUE)，所以我们就选等于 TRUE 的文字所对应的节点，且节点之间不会发生冲突。所以我们只要选那些使得问题可满足的文字的对应节点就不会发生冲突。

现在我们考虑反过来证明这个观点。假如构造的图中我们能请两个人吃饭，则对应的文字的赋值必定是可满足的。因为这些图都是精心构造的图，是一堆的三角形把它们连起来的，每个三角形里面最多最多选一个人，现在我们需要选两个人吃饭，则我们只好在左边的三角形选一个人，右边的三角形选一个人。现在我们假设请 x_1 和 x_6 吃饭，显然是不会发生冲突的，则我们相应的令 $x_1 = \text{TRUE}, x_6 = \text{TRUE}$ ，其余没选的等于 FALSE，然后我们将这些赋值代入两个子句中，则显然两个子句均等于 TRUE。

那么这个核心在哪呢？核心有两点，第一点是我们一定要构造三角形，这就是所

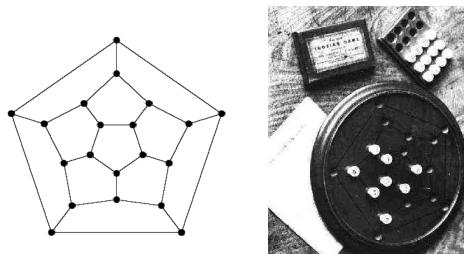
谓的“GADGET”，一种非常精巧的结构，保证如果我们选择两个人，则每个三角形必定选一个。第二点是 x_i 和 $\neg x_i$ 有边存在，一定不会同时选。这个例子大家回去再看一下，我们先往下走。

12.3.4 通过“Gadget”归约: SAT \leq_P HAMILTON CYCLE

下面我们同样利用精心构造的东西来证明哈密尔顿圈问题比 SAT 问题要难。

哈密尔顿圈

什么是哈密尔顿圈？哈密尔顿圈是英国的威廉·哈密尔顿爵士在 1857 年发明的一款游戏，这个游戏的图如下图所示。

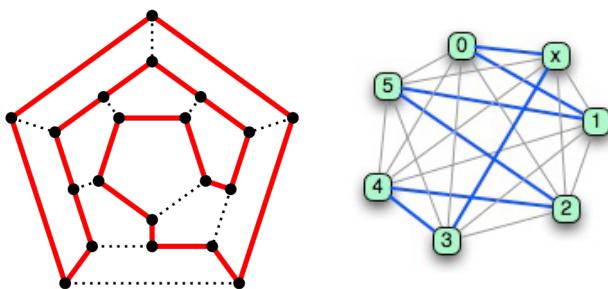


大家注意左图是一幅非常有名的图，这幅图是怎么画的呢？我来给大家演示一下。这个图有个名称，叫 ICONIAN（正十二面体顶点图），即将正十二面体投影到平面上形成的图。它的外边是一个五边形，里边也有一个五边形，然后外面的五边形和里面的五边形之间也会构造成五边形。下面我们给出哈密尔顿圈的抽象定义。

输入: 给定一个图 $G = \langle V, E \rangle$ ；

输出: 是否存在一个环路使得该路径经过图中所有的节点，且每个节点只经过一次？

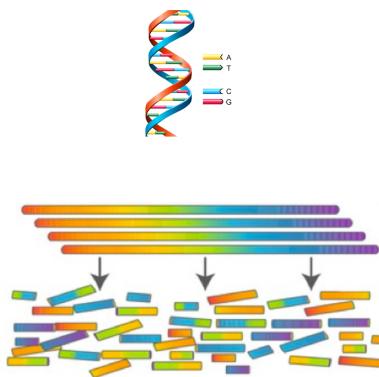
虽然这个图已经很有规律了，但是并不容易找到这样的路径，所以这个问题是挺难的一个问题。但是它是可以存在一条满足要求的环路的，结果如左图图所示。



我们从左图可以得知它是存在哈密尔顿圈的，而右图是一幅没有规律的图形，它也存在一个哈密尔顿圈。但是大家知道左图是一幅非常规整的图，而右图也是凑巧找到一个。现在如果给我们一般一幅图，问能否画出满足要求的环路。大家会觉得不知道怎么做，会很难。但是这个问题会非常非常有用，在这里我插入一点其他的知识。

哈密尔顿圈应用——人类基因组测序

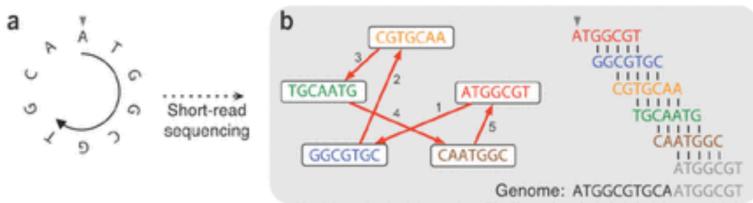
我们人类基因组测序，就完全得益于对这个问题的快速求解。那么什么叫 DNA 测序呢？我们人体内的遗传物质是染色体，染色体的核心是 DNA，DNA 是双螺旋结构，由两条链构成，彼此之间是互补链，因此我们只要知道其中一条链就可以了。我们体内是 23 条染色体，加起来是 3GB 大小，也就是说平均下来每个都很长。DNA 其实类似字符串，是由 A, T, C, G 四个字母组成的很长的字符串。



那人类基因组测序是什么意思呢？就是希望得到 DNA 链的碱基排列顺序。一开始由生物学家发起成立的国际组织——人类基因组联盟（HGP），他们首先将很长

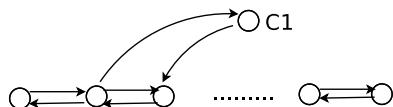
的 DNA 链人工进行切分成很短很短的 DNA 序列，然后将这些很短的 DNA 序列分配给各个国家进行同步测序。与此同时，有一位叫 VENTER 的人也在同步开展这项工作，不同的是他把 DNA 链用超声波随机的进行切分测序，然后转换成哈密尔顿路径问题，交给超级计算机进行求解运算，从而得到完整的 DNA 链的碱基序列。下面我们介绍一下第二种方法。

我们假设细菌的基因组如图 A 所示，是一个环状的，然后把它随机的打断。例如从 A 处打断，一直到 T，则会形成 ATGGCGT 这样的一个 DNA 片段。然后构造如图 B 的一幅图，每个节点就是一个片段，如果一个片段的尾巴和另一个片段的头重合了，就在两个节点之间连一条边。现在节点和边都有了，寻找完整的 DNA 链的碱基序列，就可以转换成在图中找一个环路，使得路径经过所有节点，且每个节点只经过一次，这样哈密尔顿圈就对应了基因组。



哈密尔顿圈问题与 SAT 问题之间的联系

这个问题很难，上面的 DNA 片段装配问题往往需要利用超级计算机才能完成。人们认为哈密尔顿圈问题比 SAT 问题还要难，这是为什么呢？我们先来观察一些小的例子，最后从中识别出一些特定的属性，就如同之前的几个例子中表现的那样。

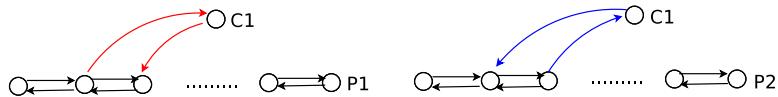


现在我们来观察一个例子。这个例子中画了一排的节点（先不考虑 C_1 节点），节点之间双向连接。现在假如需要我们遍历所有的节点，且每个节点只能经过一次，我们需要怎样遍历？第一种方法，从最左边的节点向右走，直到最右端的节点，这样我们就遍历了所有节点一次且仅有一次，我们先不考虑回来的问题；第二种方法，从右往左走，直到最左边的节点，这样我们遍历了所有节点一次且仅有一次。那么如果

我们选择从中间出发，则必然会发生一个节点经过两次的情况。

所以想要经过所有的节点且每个节点只经过一次，只有两种方案，要么从最左边走到最右边，要么从最右边走到最左边，而不能从中间开始。

我们可以注意到在上面这段话中蕴含着或（OR）关系。现在我们在图中再加入一个点 C_1 ，并进行如上图所示的连接。那么针对这种情况，如何做到遍历所有的点且每个点只经过一次？刚才已经分析过，如果只是底下的一排节点，则要么从最左边开始从左往右，要么从最右边开始，从右往左。现在我们从右边尝试一下，但是如果我们要经过 C_1 ，则必然发生一个节点经过两次这种情况。那我们只好从左边开始遍历，很容易验证存在一条路径，它能够遍历所有的点，且所有节点只遍历一遍。



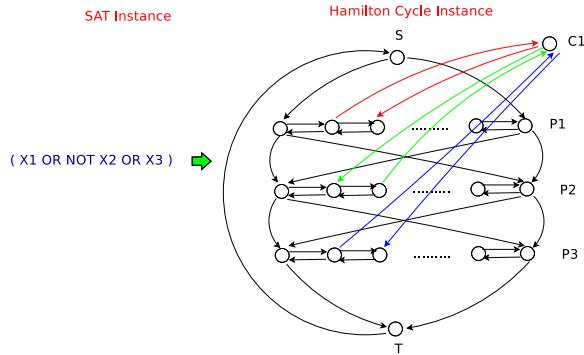
如 P1 所示精心构造的小零件，使得原来或的关系，变成只能按照一个方向走。如果我们想从右往左遍历，则可以按 P2 的方式构造这个图，将连接 C_1 节点的路径方向反转一下即可，这样只能选择从右往左走遍历所有节点。

归约——变换

现在我们构造了一个小的零件，可以用路径选择方式来表示或关系，而 SAT 问题里面同样存在或关系。在图中我们可以用从左往右走还是从右往左走来表示或关系。这样我们就可以构造一个变换，即对任何一个 SAT 问题的实例，就可以构造一个哈密尔顿圈实例。我们用下面这个实例来说明怎么构造这样一幅图。

下图中 SAT 实例是 $X_1 \text{ OR NOT } X_2 \text{ OR } X_3$ ，我们这么来构造相应的图。任何一个变量对应一排节点，即 X_1 对应第一排的节点， X_2 对应第二排的节点， X_3 对应第三排的节点。整个图是一个子句，我们构造一个特殊的节点 C_1 表示这个子句，如果子句中包含 X_j ，则第 j 排节点按顺时针方向连接 C_1 节点（如下图中第 1, 3 排节点），如果子句中包含 $\neg X_j$ ，则第 j 排节点按逆时针方向连接 C_1 节点（如下图中第 2 排节点），再构造两个节点 S 和 T 。现在来看整体是个什么样子，第一个变量构造一排的节点，一排要么从左往右走，要么从右往左走。如此，我们不妨规定，从左往右走相应的布尔变量（不是子句中的文字的取值）取 TRUE，从右往左走

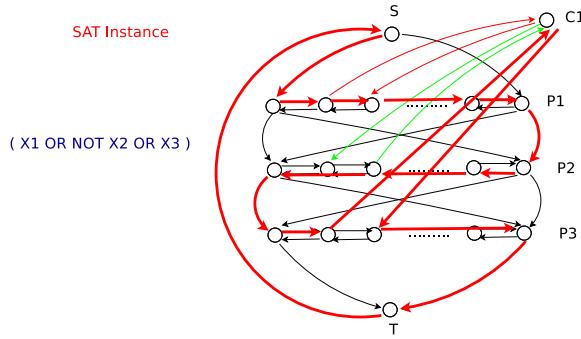
相应的布尔变量（不是子句中文字的取值）取 FALSE。



现在这幅图我们分解出来一层一层的看，第一层是 X_1 一排的节点，第二层是 X_2 一排的节点，第三层 X_3 一排的节点。我们从点 S 出发，经过所有的节点，即 S 可以连接到第一排最左边的节点，然后往右走，也可以指向第一排最右边的节点然后往左走。假如从 S 节点走到第一排的最左边的节点，然后往右走走到最右边的节点，接下来如果第二排是从右往左走则直接连接下面的节点，下一排也可能从左往右走，则我们需要将第一排最右边的节点连接到第二排最左边的节点，类似的，第一排的最左边的节点也要连到第二排最左边的节点和第二排最右边的节点，因为我们并不知道第一排会选择从哪个方向走。路径最终指向节点 T ， T 再回来连接到 S ，这样就可以了。

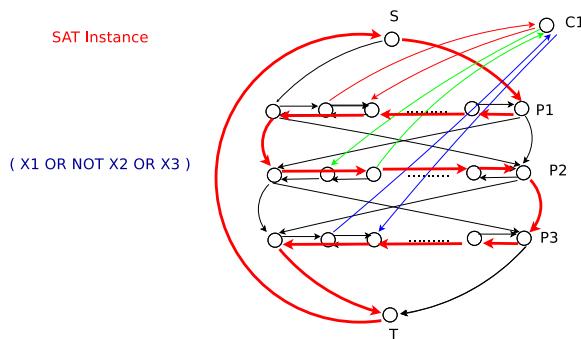
还有这个 C_1 ，这个 C_1 我们必定要走，怎么办呢？此时我们令子句中的每个文字都等于 TRUE，假定第一排从左往右走， $X_1 = \text{TRUE}$ ，则我们可以沿着红色的路径经过 C_1 。第二排表示 $\neg X_2 = \text{TRUE}$, $X_2 = \text{FALSE}$ ，所以我们从右往左走，然后沿着绿色的路径经过 C_1 。我们令 $X_3 = \text{TRUE}$ ，则我们选择沿着蓝色的路径经过 C_1 。这样对任何一个 SAT 问题的实例，我们构造了一个如上图所示的稀奇古怪的图。

如果对原始问题有个 TRUE 赋值，即使得子句结果为 TRUE，则我们在图中一定可以找到一个圈。如果原始问题有个 FALSE 赋值，则在图中一定不存在一个圈，我们来看看为什么。子句 TRUE 赋值，我们可以令 $X_1 = \text{TRUE}$, $X_2 = \text{FALSE}$, $X_3 = \text{TRUE}$ ，则子句整体都等于 TRUE。



在图中，我们从节点 S 出发，因为 $X_1 = \text{TRUE}$ ，所以第一排从左往右走。 $X_2 = \text{FALSE}$ ，所以从第一排最右边节点直接到第二排最右边节点，然后从右往左走。 $X_3 = \text{TRUE}$ ，所以从第二排最左边的节点直接到第三排最左边的节点，然后从左往右走。随后到节点 T ，最终回到节点 S 。不过我们注意到节点 C_1 并没有经过，所有点都经过了，只剩下 C_1 节点，怎么办呢？我们观察子句中哪个文字使得子句等于 TRUE ， X_1 可以让子句等于 TRUE ， X_3 可以让子句等于 TRUE ， X_2 也可以让子句等于 TRUE ，我们可以如图中选择的路径一样，从第三排进入节点 C_1 然后再走到 S 即可。大家可能觉得这种路径的选择是顺理成章，比较显然，那么接下来我们看看 FALSE 赋值，你会发现并不显然。

这里我们选择相同的 SAT 问题实例，即 $X_1 \text{ OR NOT } X_2 \text{ OR } X_3$ ，我们存在一种赋值，使得子句等于 FALSE ，我们会发现如果按照 FALSE 所代表的路径走，没有办法达到目标。我们令 $X_1 = \text{FALSE}$, $X_2 = \text{TRUE}$, $X_3 = \text{FALSE}$ ，此时子句等于 FALSE 。



现在我们按照 TRUE 和 FALSE 对应的方向在图中寻找路径，我们从 S 出发，

因为 $X_1 = \text{FALSE}$, 所以第一排选择从右往左走, $X_2 = \text{TRUE}$, 所以第二排选择从左往右走, 第三排同理选择从右往左走, 再到节点 T , 最后返回节点 S , 现在我们来观察一下有没有可能经过节点 C_1 , 我们可以发现三排均不可能在限制条件下经过节点 C_1 , 所以从这个例子中我们可以发现, 一个 FALSE 的赋值没有办法在对应的图中找到一个满足限制条件 (经过所有节点且每个节点只经过一次) 的环路。

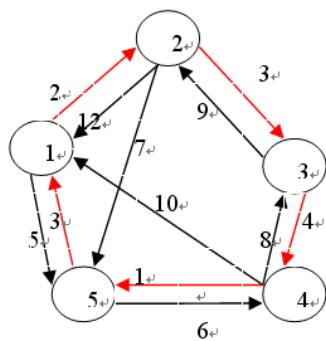
归约——等价性

看了刚才详细的剖析之后, 我们就可以证明它们的等价性了, 假如 SAT 问题的实例 ϕ 是可满足的, 也就是说我们能找到 $X_i = \text{TRUE/FALSE}$ 的一种组合, 使得子句的结果为 TRUE, 我们就从节点 S 出发, 每一排如果是 TRUE, 则选择从左往右走, 如果是 FALSE, 则选择从右往左走, 从刚才的分析过程我们知道这样的路径肯定会经过每排的节点以及 S, T 节点。那么对于子句 C_j , 由于它是可满足的, 所以至少有一个子句中的文字等于 TRUE, 则我们在相应的那一排肯定可以经过节点 C_j 。

12.3.5 HAMILTON CYCLE \leq_P TSP (TRAVELING SALESMAN PROBLEM)

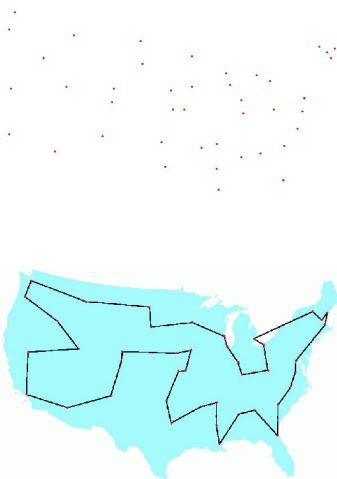
之前讲的归约属于比较复杂的归约过程, 现在我们讲一个比较简单的归约, 旅行商问题比哈密尔顿圈问题更难。旅行商问题我们第一节课就讲了, 我们再看看这个例子是什么意思?

旅行商问题 (TSP)



我们有五个城市，城市之间有道路，经过道路所花费的时间都有，例如 1 号和 2 号城市需要 2 小时，2 号到 1 号需要 12 小时，等等。问题是说，这个商人能否从 1 号城市出发，经过所有的城市且每个城市只经过一次，再回到 1 号城市，并且花费的时间最短。这是一个优化问题，使得我们花费的时间最短，对于上图所示的例子，是沿着红色的路径 (1, 2, 3, 4, 5, 1) 走花费的时间最短，可能存在其他的路径，但是花费的时间都会长于红色方案。

如今 TSP 问题被人们广泛研究，因为它在实际生活中用途广泛，在很久以前。DANTZIG(线性规划)，FULKERSON(网络流)，和 JOHNSON(《计算机与难解性》作者) 都研究过这个问题。他们在 1954 年对美国的 49 个州的首府所抽象出来的问题，找到了最短路径，其抽象问题及其结果如下图所示。



大家知道他们是怎么解决这个问题的吗？他们在每个节点上钉上一颗图钉，找一些细绳，在图钉之间进行缠绕，最后找到一条最短的路径出来。这个问题得到非常多的研究，我们可以从下图中了解这个问题的研究进展。

Year	Research Team	Size of Instance	Name
1954	G. Dantzig, R. Fulkerson, and S. Johnson	49 cities	dantzig42
1971	M. Held and R.M. Karp	64 cities	64 random poi
1975	P.M. Camerini, L. Fratta, and F. Maffioli	67 cities	67 random poi
1977	M. Grötschel	120 cities	gr120
1980	H. Crowder and M.W. Padberg	318 cities	lin318
1987	M. Padberg and G. Rinaldi	532 cities	att532
1987	M. Grötschel and O. Holland	666 cities	gr666
1987	M. Padberg and G. Rinaldi	2,392 cities	pr2392
1994	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	7,397 cities	pla7397
1998	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	13,509 cities	usa13509
2001	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	15,112 cities	df15112
2004	D. Applegate, R. Bixby, V. Chvátal, W. Cook, and K. Helsgaun	24,978 cities	sw24798

(引用自 <HTTP://WWW.TSP.GATECH.EDU/HISTORY/MILESTONE.HTML.>)

从中我们可以看到有许多著名专家学者孜孜不倦的寻找更优的答案，到现在为止能够解决 24000 多个城市。接下来我们看看旅行商问题的形式化定义。

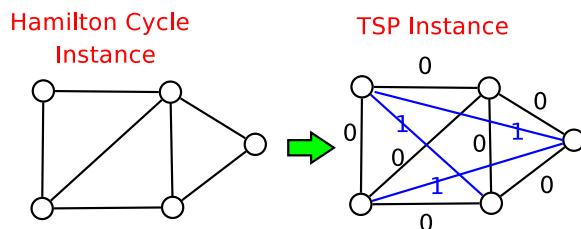
输入： 给定一个图 $G = \langle V, E \rangle$, 距离 $d : E \rightarrow R$, 和一个界限 B ;

输出： 是否存在一个哈密尔顿圈使得路径总长度小于等于 B ?

在这里稍微提示一下，原先是一个优化问题，为转换成判定问题，我们增加了一个参数 (阈值) B 。

归约——变换，等价性

为什么我们说 TSP 问题比哈密尔顿圈问题更难呢？因为任何一个哈密尔顿圈问题我们都能转换成一个 TSP 问题。给我们一幅左图，哈密尔顿圈问题是问这个里面是否存在一个环路，相应的我们构造一个如右图所示的旅行商问题。



我们把节点之间所有可能的连接都画出来，原先的边距离都是 0，新添加的边距

离都是 1，问在右图中能否旅行所有的城市且每个城市只经过一次，回到原点且总里程数小于等于 0。很明显啊，在右图中是存在一条满足要求的环路（沿着左图中外围的边走一次即可）。所以这一下就证明了，旅行商问题比哈密尔顿圈问题要难，因为旅行商问题很容易就转换成哈密尔顿圈问题。左图存在一个环路，则右图就存在一条路径总里程小于等于 0。

扩展阅读：使用 DNA 计算机计算哈密尔顿圈问题

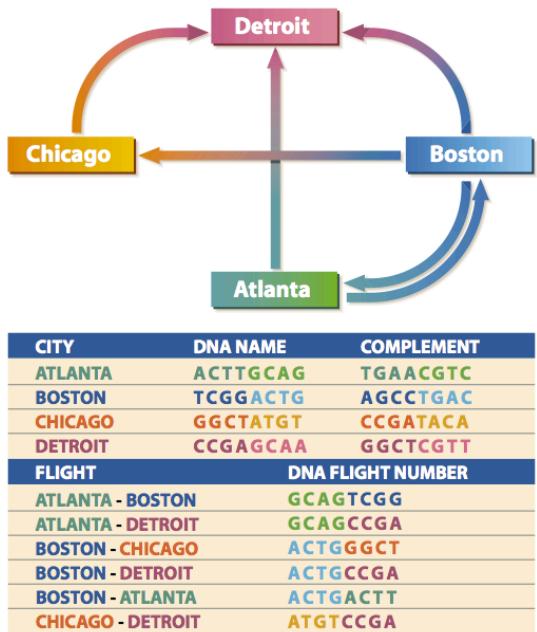
现在我们知道旅行商问题和哈密尔顿圈问题都很难，用电子计算机很难求解，因此大家就考虑能否使用 DNA 计算机进行快速求解。

在这里给大家补充一点重要知识，在 1994 年的时候，LEONARD M. ADLEMAN 发明了世界上第一台 DNA 计算机，这台计算机就能求解哈密尔顿圈问题。LEONARD M. ADLEMAN 同时也是 RSA 加密算法的发明者之一，而这两项工作也是他最著名的工作。



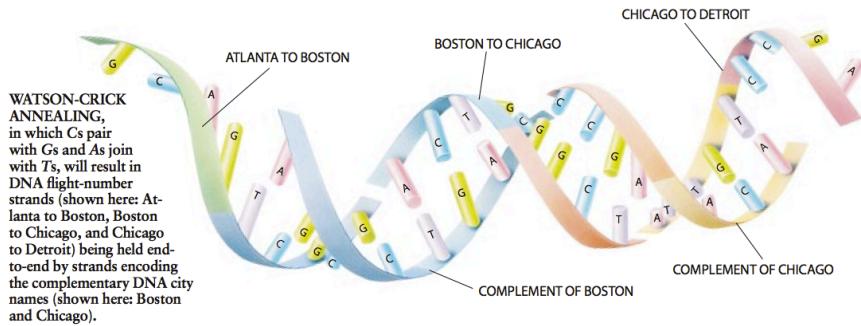
他在《科学美国人》杂志上曾撰写一篇文章，来讲述他发明 DNA 计算机的过程，他用分子机制来求解哈密尔顿圈问题，我们来看看他这个东西是怎么做的。

我们给出一个哈密尔顿圈的实例。



图中有芝加哥，底特律，波士顿和亚特兰大四个城市，城市之间的航班如图中所示，现在我们需要旅行所有城市且每个城市只经过一次。并且我们降低难度，不需要形成环路，而只需要旅行所有城市且每个城市只经过一次即可，大家想想怎么旅行。唯一一条满足要求的路径为亚特兰大——波士顿——芝加哥——底特律。现在我们用DNA计算机来求解这个问题，DNA计算机是怎么构造的呢？

每个城市都随机合成了一小段8碱基DNA，同时合成它的互补链，以亚特兰大为例，合成的8碱基DNA链为 $-ACTTGCAG-$ ，则其互补链为 $-TGAACGTC-$ 。对于8碱基的DNA链，我们分成前4个碱基和后4个碱基。对于图中的每个航班我们也合成一段DNA，以亚特兰大到波士顿的航班为例，我们合成一段DNA $-GCAGTCGG-$ ，它是由亚特兰大的后4位碱基 $-GCAG-$ 和波士顿前4位碱基 $-TCGG-$ 连接而成，其他航班也类似的合成一段DNA。然后把这些合成的DNA放入试管中，加入DNA聚合酶，此时我们就可以解决了哈密尔顿圈问题。



从图中合成的 DNA 我们可以看到，前 4 位是亚特兰大的后 4 位，接着再是波士顿的前 4 位，由于存在互补 DNA 链，因此进行碱基互补配对后，波士顿的前 4 位碱基会和波士顿的后 4 位碱基进行连接，形成完整的代表波士顿的 8 位碱基 DNA。而波士顿的后 4 位又和芝加哥的前 4 位连接到一起，如此这般，所有能够连接到一起的 DNA 片段都连接到了一起，而其中最长的 DNA 链就是我们的哈密尔顿路径，剩下的问题就是把这个最长的链找出来即可。

DNA 计算机刚刚问世时，中国科学院计算技术研究所的老师们曾对 DNA 计算机技术进行过研究，认为 DNA 计算是一种专用计算机，且需要消耗大量的 DNA，所以当时认为 DNA 计算机还不能进行大规模应用，而量子计算机更有可能投入应用。

12.3.6 SAT \leq_P GRAPH COLORING

下面我们再来讲一个证明，图着色问题比 SAT 问题更难。为什么我们一直在讨论比逻辑问题，可满足性问题要难的问题呢？因为到最后我们要证明逻辑问题比所有问题都要难，它是最难最难的问题，换句话说它比其他问题都要难，所以大家的难度系数是一样的。下堂课我们将介绍为什么一直在讨论 SAT 问题。

图着色问题

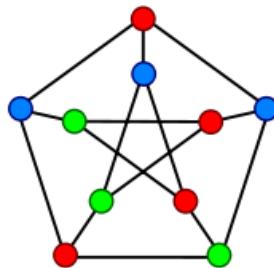
我们先看一个可抽象成图着色问题的实际问题——室内定位。室内定位现在主要依靠 WIFI，假设现在有 N 个 WIFI，每个 WIFI 都有好几个波段，如果两个 WIFI 离的很近的话，则两个 WIFI 不能用同一个波段，否则就会产生干扰。现在问题就很

清晰了，假设有 N 个 WIFI，WIFI 信号源的地理位置已知，我们应该怎么安排每个 WIFI 信号源的波段，使得相互之间不会发生干扰现象。我们总共可用的波段很少，比如每个 WIFI 只有 K 个波段可以使用，我们应该怎么设计？我们把上面描述的实际问题抽象成一个数学问题。

输入：一个图 $G = \langle V, E \rangle$ ，一个整数 k ；

输出：是否存在图 G 的 $k -$ 使得每个节点都有一个颜色，但任何一条边的两个端点颜色不同？

我们举个例子来说明一下。



这个图叫 PETERSEN 图，外面是个五边形，里面是一个五角星，对应的节点进行连接形成一个 PETERSEN 图。我们问能否进行 3 着色？这里如果将每种颜色想象成一个波段，就类似于我们上面讲的实际例子了，所以这是一个很有用的问题。这个例子可以进行 3 着色，按图所示的方式进行着色，可以保证任何一条边的两个端点颜色都不相同。

图着色问题分类及其关系

图着色问题有很多种类，刚才我们提到的例子是顶点着色，也就是说每个顶点我们给一个颜色，使得相邻的顶点颜色不同。类似的还有边着色问题，我们能否对边进行着色，使得任何相邻的边都没有相同的颜色。我们最熟悉的是第三类问题，称为面着色问题，著名的四色地图就是说对于美国地图，相邻的州着不一样的颜色，所以称为面着色问题，当然对于地图的形式还是有一些要求。这三类问题我们用图说明。

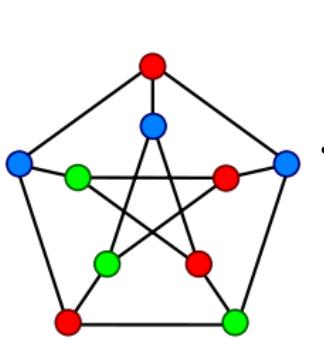


图 12.3: Peterson graph the complement of Peter-
son graph

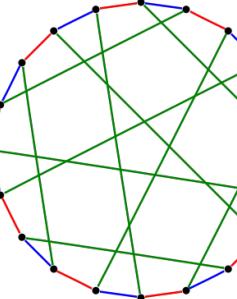


图 12.4: Desargues graph:

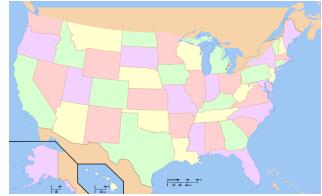
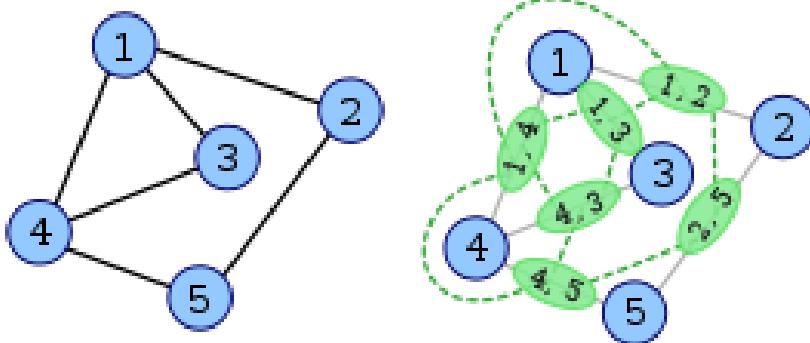


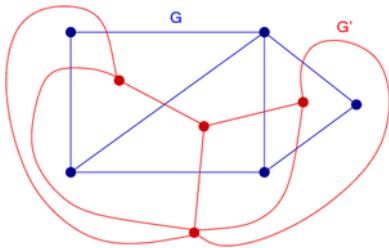
图 12.5: USA map

我们说这三类问题中，第一个顶点着色问题是最本质的，边着色和面着色问题都可以归结为第一个顶点着色问题，所以我们只需要研究顶点着色问题即可。

那么为什么边着色问题可以归结为顶点着色问题呢？假如我们需要对左图进行边着色，则任意相邻的边不能有相同的颜色，我们把它转换成顶点问题，我们对每条边单独做一个顶点，然后相邻的边所代表的顶点之间连接一条边，所以对左图进行边着色就等价于对右图中的绿色的顶点进行顶点着色。

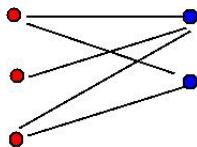


面着色也是同样的，假设下图蓝色部分表示一幅地图，我们需要对左，中，右，以及外面进行着色，使得四个面的颜色不相同。



我们也能转换成顶点的问题。我们构造一个新的图，每个面是一个顶点，两个顶点之间有边相连，所以总共有四个顶点，七条边。所以对蓝色的地图进行面着色等价于对外面红色图进行顶点着色。所以顶点着色是最本质的。

接下来我们看一个顶点着色最简单的例子，给我们一幅图，能否进行 2 着色？这种特殊的情况是很快就可以解决的，没有什么难度。因为这幅图一旦用两种颜色着色，这幅图肯定是一幅二部图，以下图为例。

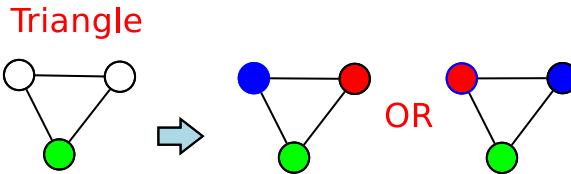


我们把红色的放左边，蓝色的放右边，相同的颜色的顶点之间没有边连接，所以边都在红色节点和蓝色节点之间，所以它肯定是一个二部图。判定一个图是否为二部图很简单，运行广度优先搜索即可。

我们先粗略的看一下进展，判定一个图 2 着色是很容易的，那么判定 K 着色呢？一幅图我们可以使用动态规划的方法，因为一幅图还是可以进行分割的，但是这个动态规划方法很慢，时间复杂度为 $O(2.445^n)$ ，是指数级的运算时间。后来又使用了一些办法使得 K 着色问题的算法时间复杂度降低为 $O(2^n n)$ ，还是指数级。对于三着色或四着色，有一些比较好的进展，时间复杂度降低为 $O(1.3289^n)$ 和 $O(1.7504^n)$ ，还是指数级的时间复杂度。以上就是图着色问题的最新进展，大家可以看出图着色问题很难，因为到现在为止设计的所有算法都是指数级的算法。下面我们来说明图着色问题比 SAT 问题更难。

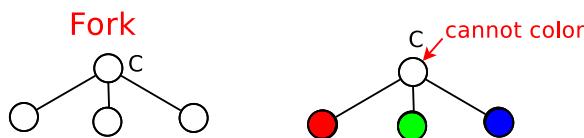
归约——构造三角形、分叉结构

为什么说图着色问题比 SAT 问题更难？我们还是要设计一些精巧的东西，它能模拟 SAT 问题中的逻辑关系。我们还是先对图着色问题画一些小的例子，发现三角形很特殊，假如我们画一个三角形，然后对它进行 3 着色，我们用图形表示如下。



不妨假设底部的节点为绿色，那么顶部的两个节点不能为绿色，要么是红色，要么是蓝色，所以要么左边节点为红色，右边节点为蓝色；要么左边节点为蓝色，则右边节点必须为红色，两种情形均已表现在上图中。大家可能注意到，在上述描述中仍然蕴含着逻辑或关系，即我们可以用这两种情况模拟 SAT 问题中的 TRUE OR FALSE，我们不妨设左边的情形为 FALSE，右边的情形为 TRUE。这样我们就可以考虑用这个来模拟 SAT 问题中的或关系 (OR)， $X_i = \text{TRUE/FALSE}$ 。所以我们需要观察一些小的例子，这些例子当中有些特性和其他问题的特性很像，这样我们能用一些小的例子来模拟这些特性，在这个例子中我们可以模拟或关系 (OR)。

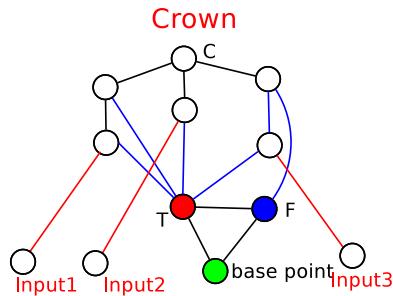
我们再来尝试构造另外一种精巧的东西，一个类似于叉子的树状结构，它分为三个分叉。



给我们如上图所示的图进行着色，则下面点不能把三种颜色都用尽。假如下面三个节点的颜色为红，绿，蓝，这个怎么着色呢？所以如果想对图进行着色，则下面三个节点只能用两种颜色，一种也可以。

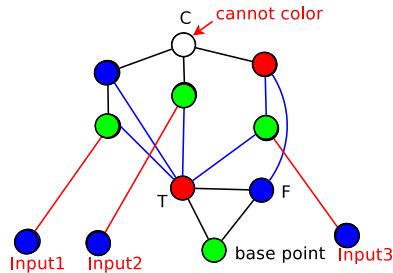
归约——构造组合“王冠”

把上面讲的两个例子组合到一块，又构成了一个稀奇古怪的东西，一个王冠。



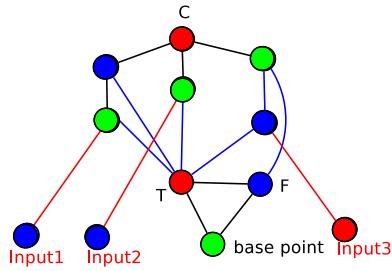
大家注意，在这里可能提前要给大家说一下，我们在证明一个问题比另外一个问题难的时候，我们找的都是稀奇古怪的情况，都是精心设计的图。这里也是精心设计的王冠，上面一部分是顶部一个点，下面分成 3 个叉，3 个叉分别垂下来一个节点，底部我们先画好一个三角形，这个三角形当作基准放在这里，因为这个三角形三个顶点分别着三种颜色，所以作为一个基准放在这里。这种设计的确是很古怪的设计，但是也是很巧妙的一种设计。接下来我们看看精巧的地方在哪里。

第一种情况，我们对三个 INPUT 均赋值 BLUE，则我们会发现没有办法对节点 C 进行着色。

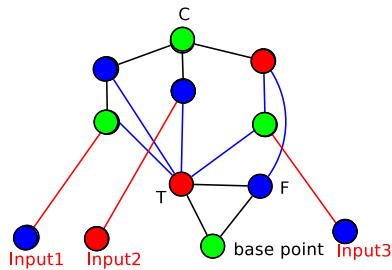


即如果三个输入均为 FALSE，则顶部节点没有办法进行着色，接下来我们详细分析一下。在这幅图中三角形的颜色是固定的，然后导致分叉下面的三个节点必须为绿色，从而导致分叉的三个节点为红，蓝，绿三种颜色，所以节点 C 不能进行着色。所以我们可以得出结论，三个输入不能同时着蓝色，即 SAT 问题的 X_1, X_2, X_3 不能同时取 FALSE，

第二种情况，假设我们三个输入的取值为蓝，蓝，红，则我们可以验证对节点 C 能够进行着色。



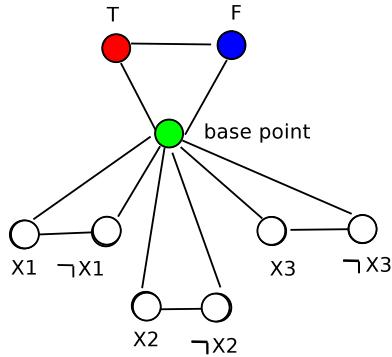
第三种情况，假设我们三个输入的取值为蓝，红，蓝，我们同样可以验证对节点 C 能够进行着色。



总共八种可能，除了第一个以外，其余的情况都可以，这就是王冠的特性即三个输入至少有一个输入为 TRUE 才能对节点 C 进行着色。

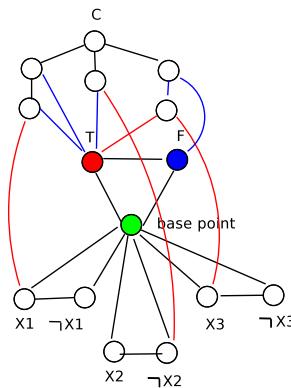
归约——变换，等价性

我们讲完王冠，终于讲到怎么为 SAT 问题构造一幅图出来。对 SAT 问题，每个变量构造一个三角形。

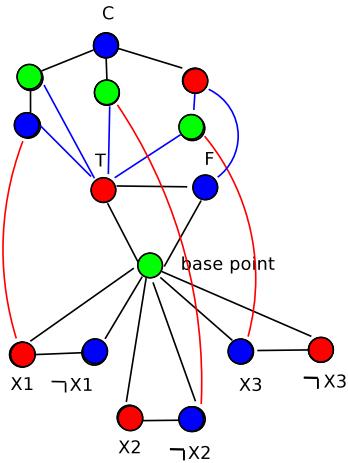


将基准三角形放在这里后，为每个变量构造一个三角形，即我们为 X_1 构造一个三角形，两个节点 $X_1, \neg X_1$ 都连接到绿色的顶点，第二个三角形是代表 $X_2, \neg X_2$ 的两个节点连接到绿色的顶点，变量 X_3 同理构造。所以我们如果给第一个三角形左边的代表 X_1 的节点着红色，则我们只能给旁边的代表 $\neg X_1$ 的顶点着蓝色，反之亦然，假如红色表示 TRUE，则第一个三角形可以表示要么 $X_1 = \text{TRUE}$ ，要么 $\neg X_1 = \text{TRUE}$ 。任何一个变量我们都可以按照这种方式进行分析。

接着我们怎么对子句进行构造呢？假如给我们一个 SAT 问题的实例，这个实例只有一个子句 $C = (x_1 \vee \neg x_2 \vee x_3)$ ，我们先画一个王冠放在上部，基准三角形也是固定的，并且王冠和基准三角形要进行相对应的连接。我们注意到子句中第一个文字是 X_1 ，则第一个输入也指向 X_1 ，同理第二个输入连向 $\neg X_2$ ，第三个输入指向 X_3 。



最后我们只要说一句话就行了，子句假如是可满足的话，则对应的图必定存在一种着色方案，进行 3 着色。下面是一个 3 着色的例子。



在这个结果中，我们令 $X_1 = \text{TRUE}, X_2 = \text{TRUE}, X_3 = \text{FALSE}$ ，子句的确是可满足的，则在对应的图中，我们令 $X_1, X_2, \neg X_3$ 三个对应的节点为红色，其余为蓝色。通过观察我们可以发现，三个输入不都为蓝色，所以可以对节点 C 进行着色。那么假如赋值导致子句是 FALSE，即 $X_1 = \text{FALSE}, X_2 = \text{TRUE}, X_3 = \text{TRUE}$ ，所以在对应的图中， $X_1, X_2, \neg X_3$ 对应的节点为蓝色，其余为红色。因为三个输入均为蓝色，所以节点 C 不可能进行着色。

以上分析说明了，对于任意一个 SAT 问题的实例，我们就能构造一个精巧的图，假如不把颜色告诉大家，让大家对这个图进行着色，任意一条边两个端点不会有同一个颜色。假如你在这种情况下能找到一种着色方案，你就能解决 SAT 问题。事实上 SAT 问题是非常非常难的，下堂课我们再介绍为什么说 SAT 问题非常非常难。