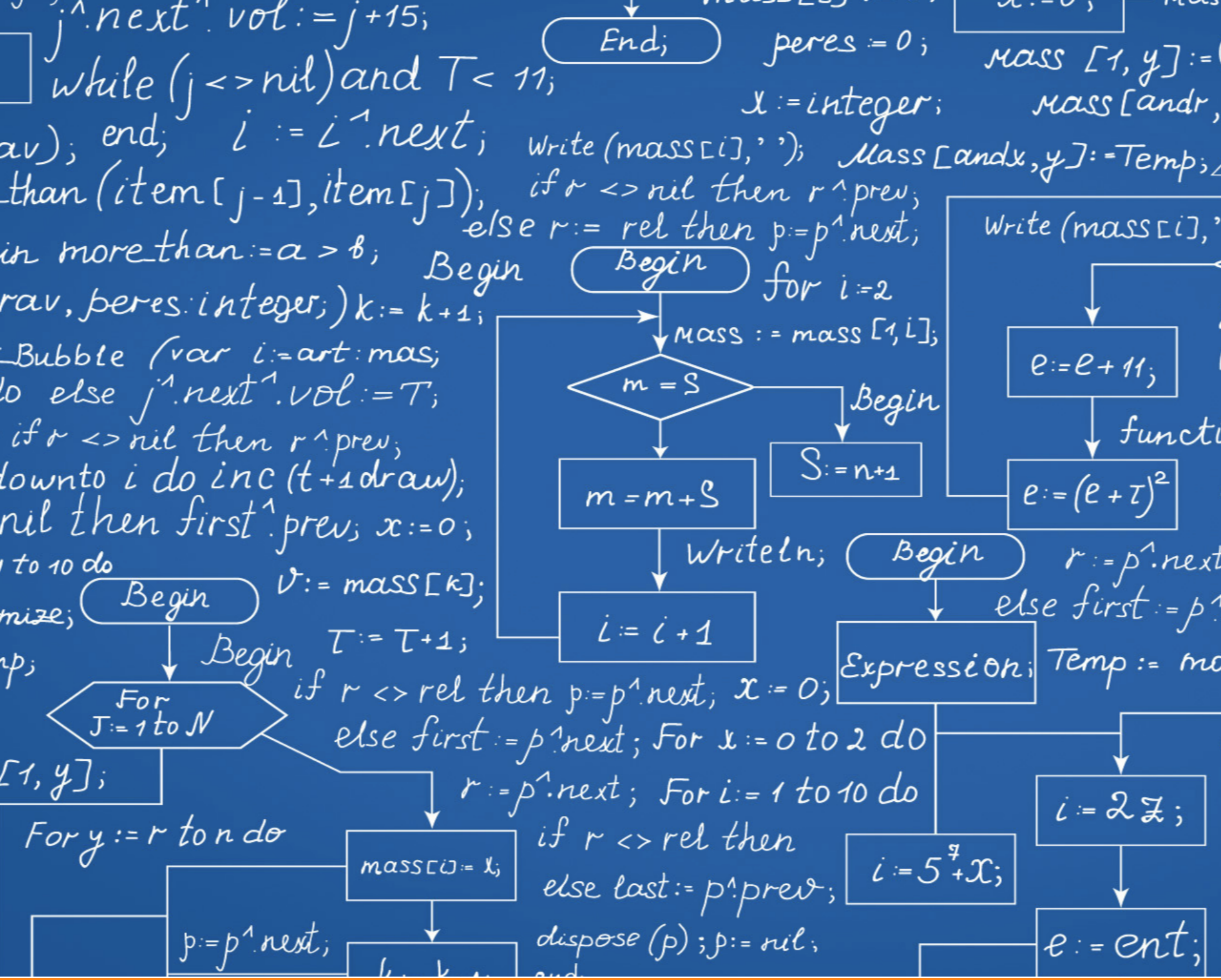


何老师算法课笔记

Always be patient, sharp and diligent.

作者：计卓1801全体

组织：华中科技大学

时间：2020 年 12 月 20 日

版本：1.6.2

目录

第 1 章 算法渐进分析

内容提要

- 渐进记号
- 渐进增长的性质
- 常见的渐进函数

1.1 关于算法的计算效率

当我们在了解计算效率或者研究算法的复杂度时，我们主要集中于运行算法的时间效率 –因为我们总是希望能够更快地运行算法。当然对于空间复杂度也是不能够忽略的，不过一下我们主要以算法的时间复杂度为载体，介绍算法的渐进分析，空间复杂度的分析可以进行类比。

一个算法在规模为n的输入下，最坏情况运行时间增长率最多与某个函数$f(n)$成正比，函数$f(n)$因此就成为了我们算法运行时间的一个界限，下面将对此进行详细讨论。

1.2 O, Ω, and Θ

在这里，我们希望寻找一种表达算法时间复杂或者是其他函数的方法，在这种方法中，常数系数和低次项对结果是没有影响的。比如 $1.62n^2 + 3.5n + 2.3$ 增长的方式和 n^2 一样。

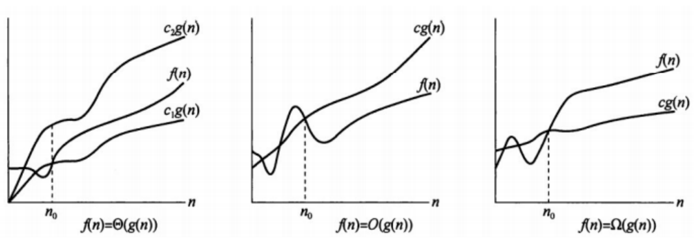


图 1.1: O, Ω, and Θ

1.2.1 渐进上界 O

令 $f(n)$ 为算法在输入规模为n的情况下的运行时间 (最坏情况)，给定另一个函数 $g(n)$，当n充分大，函数 $f(n)$ 不会超过 $g(n)$ 的常数倍，就有 $f(n) = O(g(n))$。数学定义如下：

定义 1.1. O (·)

$$O(g(n)) = \{f(n) : \exists\, c, n_0 \in R^+, such\ that : \forall n \geq n_0 : 0 \leq f(n) \leq cg(n)\}$$

♣

举个例子作为说明，假设算法的运行时间有着$f(n) = pn^2 + qn + r$的形式，其中p，q，r均为正常数，我们可以说任何具有这种形式的函数都是$O(n^2)$即$pn^2 + qn + r = O(n^2)$，证明如下：

证明

$$\begin{aligned}&\forall n \geq 1, then\ qn \leq qn^2, r \leq rn^2\\ \Longrightarrow\ & f(n) = pn^2 + qn + r\\ &\leq pn^2 + pn^2 + rn^2\\ &= (p+q+r)n^2\\ &= O(n^2)\end{aligned}$$

注意到$O(\cdot)$仅仅代表一个上界，并不代表函数准确的增长率，例如$pn^2+qn+r=O(n^3)$也是成立的但是它不是“最紧”的一个上界。

1.2.2 渐进下界 Ω

对于算法的渐进下界，我们同样可以对其进行说明：令$f(n)$为算法在输入规模为n的情况下的运行时间，给定另一个函数$g(n)$，如果对充分大的n，函数$f(n)$至少是函数$g(n)$的常数倍，就有$f(n)=\Omega(g(n))$

定义 1.2. $\Omega(\cdot)$

$$\Omega(g(n)) = \{f(n) : \exists\, c, n_0 \in R^+, such\ that : \forall n \geq n_0 : 0 \leq cg(n) \leq f(n)\}$$

♣

继续使用 $f(n)=pn^2+qn+r$ 的例子，其中q 、p、 r 均为正常数，我们可以说具有任何这种形式的函数都是$\Omega(n)$。证明如下：

证明

$$\begin{aligned}\forall n \geq 1,\ f(n) &= pn^2+qn+r\\ &\geq pn^2\\ &= \Omega(n^2)\end{aligned}$$

同$O(\cdot)$的情况，我们注意到$\Omega(\cdot)$仅仅代表一个下界，例如：$f(n)=pn^2+qn+r=\Omega(n)$也是成立的。

1.2.3 渐进紧界Θ

在上面知识的支持下，我们发现，对于同一个$f(n)$，其上下界，所对应的$g(n)$可能是相同的，即$f(n)=O(g(n))$并且$f(n)=\Omega(g(n))$ ，在这种情况下，我们可以说$f(n)=\Theta(g(n))$

定义 1.3. $\Theta(\cdot)$

$$\Theta(g(n)) = \{f(n) : \exists\, c_1, c_2, n_0 \in R^+, \text{such that:} \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

♣

沿用刚才的例子，对于$f(n)=pn^2+qn+r$，我们可以说任何具有该形式的函数都是$\Theta(n^2)$的，对于其证明就再简单不过了，我们只需要将上面两段合起来即可！在实际的计算中，对于显示的函数，我们为了求得$\Theta(\cdot)$，只需保留其最大幂的多项式或者主要成分，去掉常数项即可。

除了刚刚介绍的方法，我们还可以利用以下性质求得$\Theta(\cdot)$：
设f和g是两个函数，并且

$$\lim_{n\to+\infty}\frac{f(n)}{g(n)} = c\ (c \geq 0)$$

那么$f(n) = \Theta(g(n))$，关于该部分的证明，留给读者自行探索。

1.3 渐进增长的一些性质

下面将给出渐进增长的一些性质，对于性质的证明，可以自己证明，然后与[?]的P38-P40的证明进行对照

定理 1.1. Transitivity

(a) If $f = O(g)$ and $g = O(h)$, then $f = O(h)$

(b) If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$

(c) If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$

定理 1.2. Sum of Functions

假设f和g是两个函数，若对某个其他的函数h，都有:$f = O(h), g = O(h)$

那么，$f + g = O(h)$

推广开来：令k是确定的常数，$f_1, f_2, f_3, \ldots, f_k$和$h$是函数，且 $f_i = O(h) \quad i \in [1, k]$,

那么，$\sum_{i=1}^{k} f_i = O(h)$

定理 1.3

假设 f 和 g 是两个函数，(取非负值)，使得 $g = O(f)$

那么 $f + g = \Theta(f)$

1.4 常见的渐进函数

在一般的算法复杂度的分析中，我们常使用$O(\cdot)$进行渐进分析，其中常用的函数有以下几个：

算数级复杂度：

$$O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^2logn) < O(n^3) < \cdots$$

指数级复杂度：

$$O(2^n) < O(n!) < O(n^n)$$

如要了解更多的标准记号和常用函数，可以参考 [?]

第 2 章 骨牌问题 (Tiling Problem)

内容提要

- 定义
- 形状为2×1的骨牌

2.1 问题定义

定义 2.1. 骨牌问题

给出大小确定数量不限的骨牌，问能否不重叠的铺满整个平面

2.2 骨牌形状为2×1

骨牌形状为2×1(如图 2.1)，判断使用该种骨牌能不能不重叠的铺满给定的平面。形状为2×1的骨牌的填充问题属于P类问题。下面是一些用这种骨牌填充的例题。

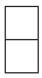


图 2.1: 形状为2×1的骨牌

2.2.1 形状为$M \times N$的矩形

这种完整的矩形(如图 2.2）是简单问题。只需要判断形状中的格子数目（$M \times N$）是否为偶数。如果格子的数目为偶数，则能填满。如果格子的数目为奇数则不能填满。

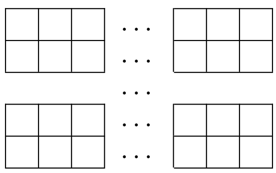


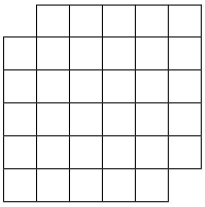
图 2.2: $M \times N$的矩形

如果格子的数目为奇数，显然不能使用2×1的骨牌铺满。而当格子的数目为偶数时，则行或列中必有一个为偶数。显然能用这种骨牌填满。

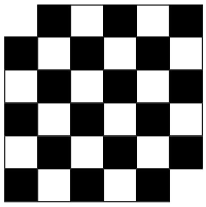
2.2.2 形状为6×6的矩形挖去两个对角

这种情况下（如图 2.3a)，格子的总数为34个，是个偶数。所以可以尝试通过对格子进行染色的方式判断。选取其中一个格子染成和黑色，对黑色上下左右相邻的格子染成白色，对白色上下左右相邻的格子染成黑色（结果如图 2.3b)。染色是判断能否被填充的一个方法。这种方法得到的不可填充的结论是可信的，但可填充的结论却不一定可信。

统计黑色和白色格子的数目，黑色有18个，白色有16个。而将骨牌染成黑白两色，发现需要被填充的形状中黑色和白色的个数不同。所以这种形状无法被2×1的骨牌填充。



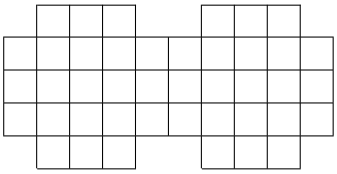
(a) 挖去两个对角的6 × 6的矩形



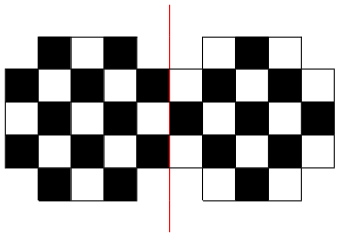
(b) 染色后的矩形

图 2.3

2.2.3 一个奇异的形状



(a) 一个例子



(b) 染色后

在这种情况下（如图 2.4a），使用上述方式染色后（如图 2.4b）发现黑色与白色的块数相同，均为21块。但这个图形却是不可被填充的。

现假设骨牌能填充这个形状，那么红线左侧和红线右侧都应该是可以被填充满的。但红线左侧的黑色方块比白色方块多4个，需要从右面借4个白色方块才能使左红线侧的褐色方块与白色方块数目相同。但红线右侧在不借出黑色方块的情况下最多借出2个白色方块，不满足需求。因此左边是不可能被填满的。所以整个平面也是不可被填满的。

这个问题还可以用网络流来求解。详见小节 7.2。

第 3 章 匹配问题（Matching Problem）

内容提要

- 定义
- 算法设计
- 算法分析

3.1 问题引入

现在有n个男性的一个集合$M = \{m_1, \ldots, m_n\}$和n个女性一个集合$W = \{w_1, \ldots, w_n\}$。用$M \times W$来表示(m, w)所有可能有序对的一个集合，其中$m \in M, w \in W$。一个匹配（Matching）S是一个有序对的集合，这些有序对来自于$M \times W$，并且每一个来自M中的元素和来自W中的元素最多在S中出现一次。一个完美匹配（Perfect Matching）S'是指每一个来自M和W中的元素都出现过一次的匹配。

定义 3.1. 匹配（Matching）

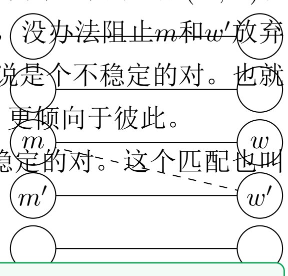
对于一个给定的0图$G = (V, E)$，这幅图的一个匹配M是图G的一个子图（由原来的图的一部分顶点和一部分边构成的图），其中每两条边都不相邻（没有公共顶点）。在匹配图中，一个顶点连出的边数至多是一条。如果这个顶点连出一条边，就称这个顶点是已匹配的。M中的一条边的两个端点叫做在M中是配对的。♣

定义 3.2. 完美匹配（Perfect Matching）

完美匹配是一个包括了图G中原来的所有顶点的匹配。♣

现在我们添加优先级的概念。每一个男性$m \in M$对所有的女性有一个排名。如果在m的排名中w比w'更高，我们说m相对于w'更倾向于w。我们把m的有序评价叫做他的倾向列表。在m的评价中所有人的排名都不相同。类似的，对于每一个$w \in W$也对所有的男性有个排名。

给定一个完美匹配S，我们应该担心下面的情况（如图 3.1）。现在有两个S中的匹配(m, w)和(m', w')，但m相对于w更倾向于w'，并且w'相对于m'更倾向于m。在这种情况下，没办法阻止m和w'放弃当前的对象然后组成新的一对。在这种情况下，我们说(m, w')对于S来说是个不稳定的对。也就是说，尽管(m, w')不存在于S中，但m和w'，相对于他们在S中的同伴，更倾向于彼此。



我们的目标是找到一个匹配，这个匹配是完美匹配，并且没有不稳定的对。这个匹配也叫稳定匹配。

定义 3.3. 稳定匹配（Stable Matching）

稳定匹配是一个满足以下条件的匹配：

(i) 是一个完美匹配

(ii) 不存在不稳定的的对

♣

3.2 算法设计

依据上文提到的问题，我们来考虑一下求解方法。

* 在最一开始，每个人都是未匹配的。设想一个未匹配的男性m选择了在他倾向列表中排名最高的女性w，并且提出了请求。所以我们能立刻声明(m,w)是我们最后稳定匹配中的一个对吗？不行，在后面有可能m'，他在w的排名更高，同样向w提出了请求。在另一方面，w立刻拒绝m也是有风险的。她可能不会再收到排名比m高的人发出的请求。所以，一个自然地想法是将(m,w)对置成约定状态（约会）。
* 假定我们现在处于某个状态，部分是没有匹配的，部分是处于约定状态的。下一步就是一个未匹配的m选择了 m还没有申请过的排名最高的w，向w提出申请。如果w是未匹配的，那么m和w就成为约定状态。如果w是跟m'处于约定状态，在这种情况下，w根据m和m'谁的排名更高来选择跟谁成为约定状态，而另一个人就成了未匹配状态。
* 最后，这个算法会在没有人处于未匹配状态时停止。这时，所有处于约定状态的就成为最终的结果，得到了一个稳定匹配。

算法 1就是盖尔-沙普利算法（Gale-Shapley algorithm）的具体描述。

算法 1: 盖尔-沙普利算法（Gale-Shapley algorithm）

```
Data: 男性集合M,女性集合W
begin
    将m ∈ M和w ∈ W置为未匹配状态。
    while 还有m ∈ M处于未匹配状态并且在他的倾向列表中还有未申请过的 do
        选择一个处于未匹配状态的m
        选择一个m还没有提出过申请的排名最高的w ∈ W
        if w是未匹配的 then
            (m, w)成为约定状态
        else w现在跟m′处于约定状态
            if 相比于m,w更倾向于m′ then
                m依然是未匹配的状态
            else 相比于m′,w更倾向于m
                (m, w)成为约定状态
                m′变成未匹配状态
Result: 将所有的约定状态转化成匹配状态后得到一个稳定匹配
```

3.3 算法分析

通过上述算法，我们可以得到如下命题

定理 3.1

w从第一次被申请后始终处于约定状态，并且w的对象会越来越好。

从m的视角会完全不同。随着时间的流逝，他会在未匹配和约定状态之间转换。但下面的命题始终成立

定理 3.2

在m的优先列表中，m请求的对象的排名越来越低

下面说明算法是会终止的。

定理 3.3

G-S算法最终会在n^2次循环后终止

给出一个显然的证明。

证明 在本算法中，每次迭代都会选择一个未匹配的m向他还没有申请过的w提出申请。我们令$P(t)$来表示在第t次迭代后，m向w提出过申请的对(m, w)的集合。对于任何的t，$P(t+1)$相对于$P(t)$一定是严格增长的。所有的M和W组成的对一共有n^2个，所以$P(t)$的值在本算法中最多为n^2。所以上面的算法最多会在n^2后停止。

下面是一些不显然的结论

定理 3.4

如果某个时刻m是自由的，那么一定有一个w他还没有申请过。

证明 假定某个时刻m是未匹配的，但他已经向所有的女性提出了申请。那么根据定理3.1,每一个女性都处于约定的状态。既然约定状态是一个匹配，那么就必然有n个男性处于约定的状态。但一共只有n个男性，而m是未匹配的，这就产生了矛盾。

定理 3.5

当算法结束时得到的匹配集合S一定是一个完美匹配。

证明 约定的对最后会形成匹配。我们G-S算法会在还有一个未匹配的m的时候终止。在终止的时候，m一定向所有的女性发出过请求，否则算法不会终止。但这与定理3.4矛盾，不可能有向所有的女性申请过后还有未匹配状态的男性。

最后，我们得到算法的结果是一个稳定匹配

定理 3.6

集合S是运行G-S算法后得到的配对的集合。则S是一个稳定匹配。

证明 由定理3.5，集合S是一个完美匹配。所以要证明S是一个稳定匹配，我们可以假定S中存在不稳定的对，然后推出矛盾。假定S中有两个匹配对(m, w)和(m', w')，并且具有下面的特点

* 相比于w，m更倾向于w'
* 相比于m'，w'更倾向于m

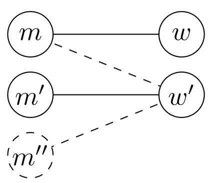


图 3.2

在执行算法的过程中，当m向w提出申请时，m是不是已经向w'提出过申请？如果没有，那么w在m的评价中比w'更高，这跟我们的假设“相比于w，m更倾向于w'”矛盾。如果m已经提出过申请，那么他被w'选择了更倾向的m''时拒绝了。m'是w'的最终选择，所以m'的在w'中的排名不会比m''低。这与假设“相比于m'，w'更倾向于m”矛盾。所以集合S中不存在不稳定对，S是个稳定匹配。

下面来说明一下G-S算法得到的解是相同的。书上说有一种简单的方法来证明这一点（~~我没看出来~~）。我们可以证明得到的解具有相同的特征。然后再得到解是相同的。

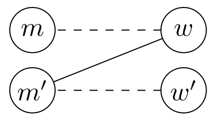
首先，如果存在一个稳定匹配包含对(m, w)，我们可以说w是m的一个有效同伴。如果w是m的一个有效同伴，并且在m的排名中比m的其他的有效同伴都要高，那么可以说w是m的最佳有效

同伴。我们用$best(m)$来表示m的最佳有效同伴。现在假定S^*是一个满足$S^* = \{(m, best(m)) : m \in M\}$的对集合。我们将要证明下面的定理。

定理 3.7

每一次执行G-S算法得到的结果都是集合S^*。

证明 我们假定，某次执行G-S算法得到的匹配S，其中存在男性没有匹配到他的最佳有效同伴。我们把这个过程记为ε。既然男性提出申请的次序是沿着倾向列表降序的，那也就是说存在男性在ε中被他的有效同伴被拒绝了。那么在ε中，记第一个被有效对象拒绝的为m，并且被他的有效对象w拒绝。考虑m被w拒绝的时刻，要么是因为w已经跟排名更高的m'成了约定状态，要么是因为排名更高的m'向w提出了申请。不管怎样，在那个时刻w跟m'形成或维持约定状态，m'在w的排名中比m要高。



-- 集合S'

— 集合S或过程ε

图 3.3

既然w是m的一个有效同伴，那么存在一个稳定匹配S'包含这个对(m, w)。那么，假定这个时候跟m'组成对的是w'，并且$w¸ \neq w$。

在ε中，m是第一个被w拒绝的有效对象，那么当m'向w提出申请的时候还没有被有效对象（比如w'）拒绝过，也就是说在m'看来w的排名比w'要高。并且m'在w的排名比m要高。那么(m, w')在S'中倾向成为一对，但$(m, w') \notin S'$，这就是个不稳定因素。

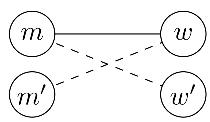
这跟我们关于S'是稳定匹配的声明是矛盾的，因此我们最初的声明是错误的。那么不存在任何过程使得m不与最佳有效同伴$best(m)$匹配。

对于男性，G-S算法得到的是理想的。不幸的是对于女性来说是不一样的。对于一个女性w，如果存在一个稳定匹配包含对(m, w)，那么我们说m是一个有效同伴。如果m是w的一个有效同伴，并且对于w来说没有有效同伴的排名比m低，那我们说m是w的最差有效同伴。

定理 3.8

在稳定匹配S^*中，每一个女性都跟她的最差有效同伴匹配。

证明 假定存在对(m, w)在稳定匹配S^*中，并且m不是w的最差有效同伴。那么存在一个稳定匹配S'，w跟一个比m差的人m'形成了对。在S'中，m跟w'形成了一对，且$w' \neq w$。由定理3.7，我们知道w是m的最佳有效对象，并且w'是m的有效对象，那么我们可以说相对于w'，m更倾向于w。但这会导致(m, w)成为S'中的不稳定因素，这与S'是稳定匹配是矛盾的。因此我们的假设是错误的。即对于在S^*中的(m, w)，m是w的最差有效同伴。



-- 集合S'

— 集合S^*

图 3.4

第 4 章 分治算法之平面最近点对问题

内容提要

- 平面最近点对问题定义
- 分治算法设计
- 分治算法时间复杂度分析
- 伪代码

4.1 平面最近点对问题定义

给定二维平面上的$n(n \geq 2)$个不同的点p组成点集$P = \{p_i|1 \leq i \leq n\}$，设计算法寻找欧式距离最近的点对$(A, B)$。

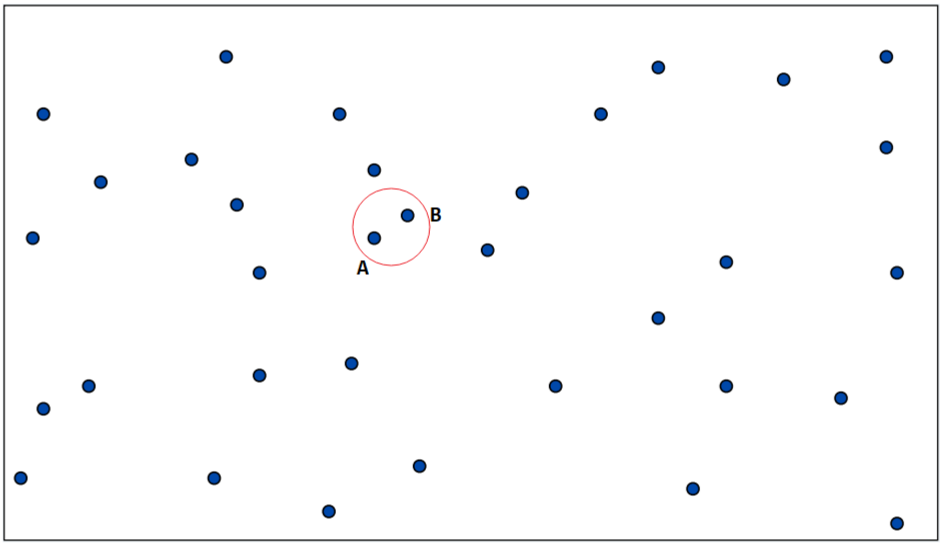


图 4.1: 问题定义图例

如图 4.1中点对(A, B)即为问题的答案。

4.2 分治算法设计

对于这样一个问题，我们很直接地可以使用BF (Brute Force)算法进行暴力求解，即二重循环计算所有点之间的距离，从而获得最小距离，显然该算法的时间复杂度为 $O(n^2)$。那么有没有更快的算法呢？本章我们使用经典的算法思想——分治，设计一个$O(n \log n)$的算法。

4.2.1 分治问题

遵循分治思想，我们首先要考虑如何分治问题使得问题规模约减。

我们使用X坐标作为第一关键字、Y坐标作为第二关键字，对点集P进行排序，并以点$p_{\lfloor \frac{n}{2} \rfloor}$作为分治点，获得如下两个点集：

$$P_1 = \{p_i \mid 1 \leq i \leq \lfloor \frac{n}{2} \rfloor\}$$

$$P_2 = \{p_i \mid \lfloor \frac{n}{2} \rfloor < i \leq n\}$$

这样就将当前问题约减为两个规模为$\frac{n}{2}$的子问题分治过程如图 4.2中所示。

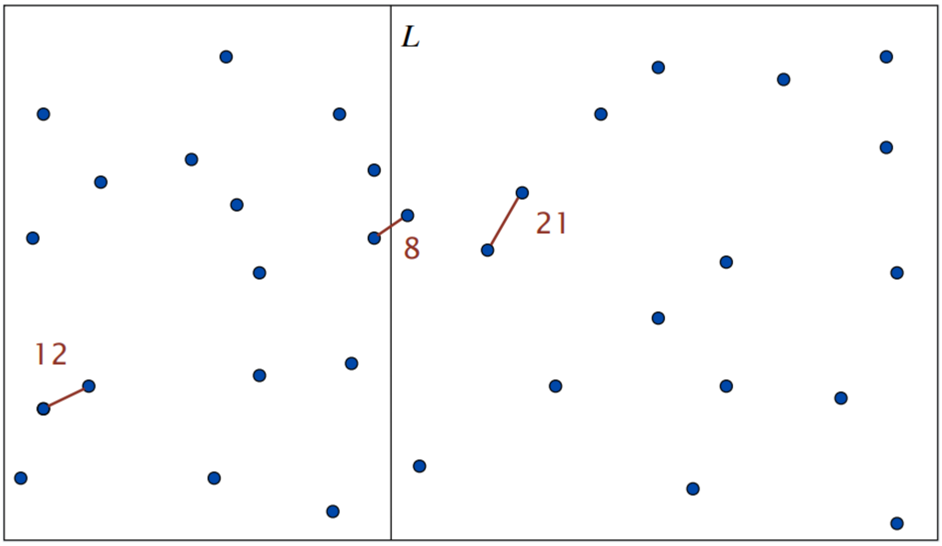


图 4.2: 分治过程图例

如此递归下去，我们可以求得两个点集相对应的最近点对距离δ_1, δ_2，取其中较小值记为$\delta = \min\{\delta_1, \delta_2\}$。

当分治到点集大小为2个或3个时，可以在常数时间内计算出子问题的解。

4.2.2 合并结果

接着，我们需要考虑如何合并子问题的解。

上述的δ一定是正确的合并结果嘛？显然不是，我们并没有考虑，一端在P_1，一端在P_2的线段。因此，在合并阶段，我们要将这种情况考虑在内。

这里，我们将所有横坐标与分治点$p_{\lfloor \frac{n}{2} \rfloor}$的横坐标 $x_{\lfloor \frac{n}{2} \rfloor}$差值小于$\delta$的点组成集合$B$，即

$$B = \{p_i \mid \left| x_i - x_{\lfloor \frac{n}{2} \rfloor} \right| \le \delta,\ 1 \le i \le n\}$$

因为只有B集合中的点之间的距离才有可能小于δ。 B集合如下图图 4.3中阴影部分所示:

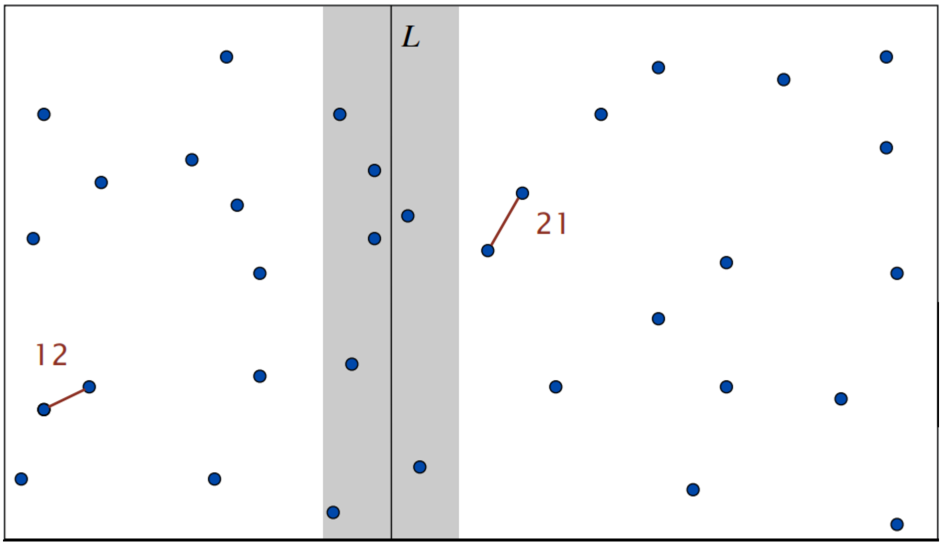


图 4.3: 合并过程图例

进一步，我们的目标是检验在B集合中是否存在距离比δ更近的点对，以此更新当前问题的

解。因此，对于每个$p_i = (x_i, y_i) \in B$遍历所有在其之下竖直距离不超过δ的点，即遍历集合

$$C(p_i) = \{p_j \mid y_i - \delta \le y_j \le y_i, p_j \in B\}$$

为了方便遍历，我们可能会想到对B集合中的点，以Y坐标为第一关键字，X坐标为第二关键字，进行排序。但是如此一来，每一次合并的时间复杂度为$O(n \log n)$，徒增时间消耗，因此我们采取合并策略，即按照Y坐标为关键字，进行P_1, P_2的归并来直接获得排序后的集合B，这样只需要$O(n)$的时间。

考虑到$C(p_i)$会因为归并操作而维持在$O(n)$数量级，其实不然，该集合的大小不会超过7。下面给出证明。

根据定义，$C(p_i)$中的点的纵坐标均处于$(y_i - \delta, y_i]$范围内，且其中的所有点的横坐标均处于$(x_m - \delta, x_m + \delta)$范围内。这样便构成了一个$2\delta \times \delta$的矩形。如下图图 4.4所示 。

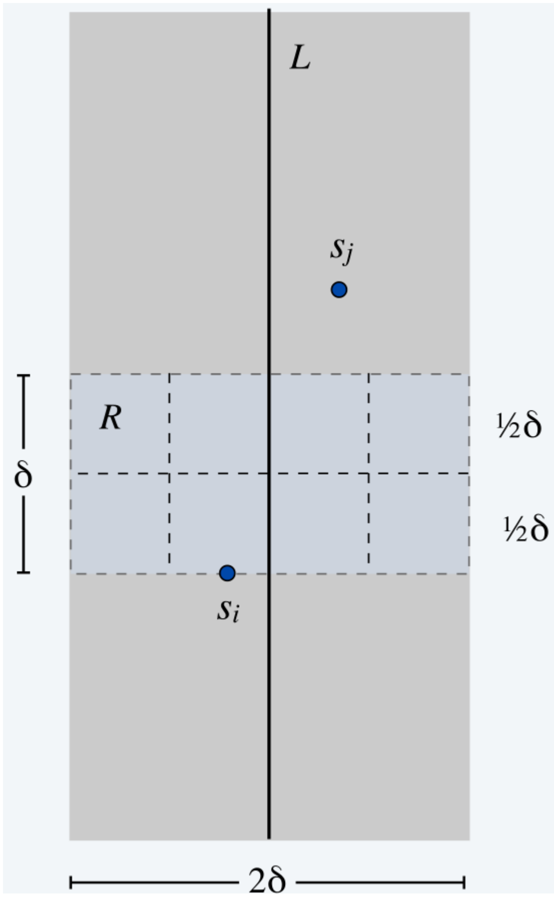


图 4.4: $C(p_i)$

接着，我们将这个矩形分拆成左右两个$\delta \times \delta$的正方形，左侧正方形的点集为$C(p_i) \cap P_1$，右侧正方形的点集为$C(p_i) \cap P_2$，从上述的分治过程可知，这两个点集内的点之间的距离一定不小于δ。

进一步，我们将$\delta \times \delta$正方形，分拆成四个$\frac{\delta}{2} \times \frac{\delta}{2}$ 小正方形，因为这个小正方形的对角线为$\frac{\delta}{\sqrt{2}} < \delta$，所以小正方形中最多只有一个点，而总共有8个小正方形，最多有8个点，除去p_i，则最多只有7个点。

至此，我们完成了父问题的分治与子问题的合并。

4.3 分治算法的时间复杂度分析

首先，第一次排序可以使用时间复杂度为$O(n\log n)$的排序算法，如快速排序或者归并排序。

接着，我们考虑分治过程，即通过分治，我们将规模为n的父问题，分为两个规模为$\frac{n}{2}$的子问题。

最后，归并过程中，根据采用的合并策略以及上述对更新操作的证明，我们需要$O(n)$级别的时间完成。

综上，给出递推式如下：

$$T(n)=\begin{cases}O(1) & 2\leq n\leq 3\\ 2T(\frac{n}{2})+O(n) & n>3\end{cases}$$

推导如下：

$$\begin{aligned}T(n)&=2T(\frac{n}{2})+O(n)\\&=2^2T(\frac{n}{2^2})+2O(\frac{n}{2})+O(n)\\&=2^2T(\frac{n}{2^2})+2O(n)\\&\vdots\\&=2^kT(\frac{n}{2^k})+kO(n)\ \ (n=2^k)\\&=O(n)+O(n\log n)\\&=O(n\log n)\end{aligned}$$

4.4 伪代码

算法 2: Nearest-Pair

Data: Point List $P=\{p_i \mid 1\leq i\leq n, p_i=(x_i,y_i)\}$
P should be sorted by x-coordinate in descending order.
Result: the minimum distance δ

```
begin
    if |P| <= 3 then
        Return the minimum Euclidean-Distance between each pair of points.
    m ← ⌊n/2⌋
    δ1 ← Nearest-Pair(P[1, ..., m])
    δ2 ← Nearest-Pair(P[m+1, ..., n])
    δ ← min{δ1, δ2}
    B ← MergeByY(P1, P2)
    foreach p_i ∈ B do
        foreach p_j ∈ C(p_i) do
            δ ← min{δ, Euclidean-Distance(p_i, p_j)}
    Return δ
```

第 5 章 分治法之大数乘法

内容提要

- 问题背景
- 改进分治法
- 直接分治法

5.1 问题描述

给定两个大数A和B，试计算 $A \times B$. 其中A和B分别表示为 $A = a_n a_{n-1} a_{n-2} \dots a_2 a_1$ ，$B = b_n b_{n-1} b_{n-2} \dots b_2 b_1$. 根据已学知识，给出如下引理。

引理 5.1

直接计算$A+B$，其复杂度为$O(n)$, 其中n为A和B的十进制位数。

直接计算$A \times B$时，我们将A与B的各位相乘，在将各中间结果相加，得到最终结果。不难得出，这一过程需要进行n次基本乘法与$n+1$次加法。根据引理5.1，有：

定理 5.1

直接计算$A \times B$的时间复杂度为$O(n^2)$.

由定理5.1和引理5.1可知，如果我们直接相乘两个大数，其时间复杂度相比加法运算高出一个量级。由于乘法在计算机中大量存在，我们希望找到更好的算法来降低乘法计算的时间复杂度，以提升计算机的性能。分治法为我们提供了一条途径。

5.2 直接分治法

5.2.1 算法描述

这是一种简单的分治方法，将两个大数分为前后两部分，进行相乘。不失一般性，这里假设n为偶数。将A与B分割为A_2, A_1, B_2, B_1,即：

$$
\begin{aligned}
A_2 &= a_n a_{n-1} \dots a_{\frac{n}{2}+2} a_{\frac{n}{2}+1} \\
A_1 &= a_{\frac{n}{2}} a_{\frac{n}{2}-1} \dots a_2 a_1 \\
B_2 &= b_n b_{n-1} \dots b_{\frac{n}{2}+2} b_{\frac{n}{2}+1} \\
B_1 &= b_{\frac{n}{2}} b_{\frac{n}{2}-1} \dots b_2 b_1
\end{aligned}
$$

则A可以写为$A = A_2 \times 2^{\frac{n}{2}} + A_1$. B可以写为$B = B_2 \times 2^{\frac{n}{2}} + B_1$. 计算$A \times B$的问题在进行上述转换后表示为：

$$
\begin{aligned}
A \times B &= (A_2 \times 2^{\frac{n}{2}} + A_1) \times (B_2 \times 2^{\frac{n}{2}} + B_1) \\
&= A_2 B_2 \times 2^n + (A_2 B_1 + A_1 B_2) \times 2^{\frac{n}{2}} + A_1 B_1
\end{aligned}
$$

此时将两个大数相乘的问题转化为4个乘法子问题和3个加法子问题。显然，分治策略还可以对子问题使用，继续减小问题的规模。

5.2.2 伪代码

算法 3: DirectDAC

Input: Two large numbers A, B, which both have n decimal digits
Result: $A \times B$

begin
  $n \leftarrow$ Number of Decimal Digits of A and B
  if $n \neq 1$ **then**
    Divide A, B into A_2, A_1, B_2 and B_1
    $C_3 \leftarrow DirectDAC(A_2, B_2)$
    $C_2 \leftarrow DirectDAC(A_2, B_1)$
    $C_1 \leftarrow DirectDAC(A_1, B_2)$
    $C_0 \leftarrow DirectDAC(A_1, B_1)$
    return $C_3 \ll n + (C_2 + C_1) \ll (\frac{n}{2}) + C_0$
  else
    return $A \times B$

5.2.3 复杂度分析

由上述的算法描述可知，算法的主要开销来自于每次分支带来的4个乘法子问题和3个加法子问题，由于移位可在机器中由一个简单的指令完成，我们忽略这个操作的时间。
假设$T(n)$表示两个n位大数相乘所需的时间开销，则在直接分治法中：

$$\begin{aligned} T(n) &= 4T(\frac{n}{2}) + 3n \\ &= 4T(\frac{n}{2}) + O(n) \end{aligned}$$

根据主方法，$\log_2 4 = 2 > 1$, 推出如下定理：

定理 5.2
用直接分治法计算$A \times B$的时间复杂度为$O(n^2)$.

根据定理5.2,直接分治法的性能是令人失望的，因为其并不能提供时间上优于直接相乘的性能。但分治策略提示我们，这个算法的性能与乘法子问题的数目强相关。我们如果能够用一些其他的开销换取更少的乘法子问题数目，也许能得到更好的算法。

5.3 改进分治法

5.3.1 改进思路

在直接分治法中，通过对大数进行分割，我们有：

$$A \times B = A_2B_2 \times 2^n + (A_2B_1 + A_1B_2) \times 2^{\frac{n}{2}} + A_1B_1$$

这个过程中，引入了4次乘法运算；在上一节中提到，分治策略和主定理提示我们尽可能减少乘法的次数。但换取更低的乘法子问题数，需要其他的开销。一种想法是，由于加法的复杂度为$O(n)$，我们也许可以用略多的加法子问题，来减少乘法子问题数。基于此想法，我们对直接分治法作出一些改进。首先将直接分治法中的计算式修改为：

$$\begin{aligned} A \times B &= A_2B_2 \times 2^n + (A_2B_1 + A_1B_2) \times 2^{\frac{n}{2}} + A_1B_1 \\ &= A_2B_2 \times 2^n + ((A_2 + A_1) \times (B_2 + B_1) - A_2B_2 - (A_1B_1)) \times 2^{\frac{n}{2}} + A_1B_1 \end{aligned}$$

观察上式，我们只需要做3次乘法，即计算A_2B_2, A_1B_1, $(A_2 + A_1) \times (B_2 + B_1)$, 以及4次加法，2次减法。考虑到加法和减法本质上等同，我们成功地将这一问题转化为了3个乘法子问题和6个加法子问题。相比于直接分治法，我们降低了乘法的数量。

下面给出该算法的伪代码及复杂度分析。

5.3.2 伪代码

算法 4: ModifiedDAC

```
Input: Two large numbers A, B, which both have n decimal digits
Result: A × B
begin
    n ← Number of Decimal Digits of A and B
    if n ≠ 1 then
        Divide A, B into A2, A1, B2 and B1
        C2 ← DirectDAC(A2, B2)
        C1 ← DirectDAC(A1, B1)
        C0 ← DirectDAC(A2 + A1, B2 + B1)
        return C2 ≪ n + (C0 − C2 − C1) ≪ (n/2) + C1
    else
        return A × B
```

5.3.3 复杂度分析

同上节的复杂度分析，我们此处也忽略移位操作带来的开销。改进分治法中，我们将问题分解为3个乘法子问题与6个加法子问题。因此有：

$$\begin{aligned} T(n) &= 3T(\frac{n}{2}) + 6n \\ &= 3T(\frac{n}{2}) + O(n) \end{aligned}$$

根据主方法，$\log_2 3 > 1$. 推出如下定理：

定理 5.3

用改进分治法计算$A \times B$的时间复杂度为$O(n^{\log_2 3}) \approx O(n^{1.585})$.

第 6 章 动态规划（1）

内容提要

- ❑ 带权区间调度
- ❑ 矩阵链乘法

6.1 概述

在前面的课程中，已经学习过了贪心法和分治法两种策略，本章将开始动态规划（Dynamic programming）的学习。简要比较一下几种算法策略的不同。

- 贪心法 -基于贪心策略，每次总是选取眼前最优的选项，同时期待最终的结果最优。优点在于思考和模型建立较为简单，难点在于如何证明算法的正确性。
- 分治法 -分而治之，子问题不存在重叠。难点在于如何分割问题，以及如何合并解。
- 动态规划 -思想类似于分治，与贪心法相反，子问题之间存在重叠，算法执行过程中记录子问题的解。难点在于如何找到转移方程。

本章将介绍动态规划在带权区间调度 6.2 、矩阵链乘法 6.3 两个问题上的应用。

6.2 带权区间调度

6.2.1 问题描述

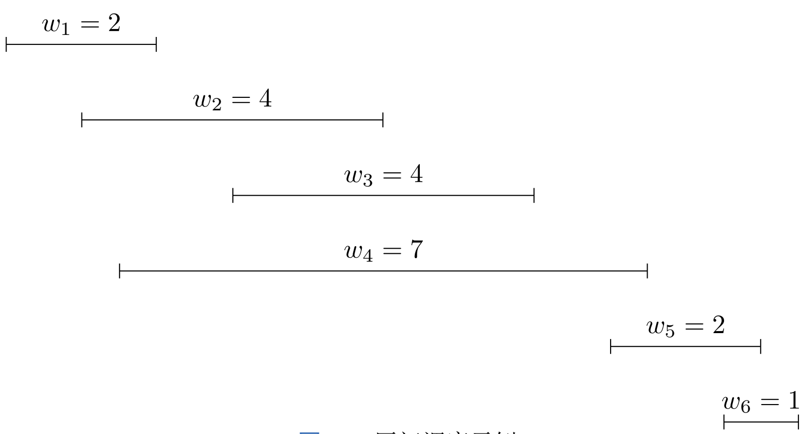


图 6.1: 区间调度示例

下面给出带权区间调度问题的定义。

定义 6.1. 带权区间调度问题

给定区间$I_1, I_2, \ldots, I_n$，s_i为I_i开始时间，f_i为I_i的结束时间（$f_i > s_i$），$w_i > 0$, 假设$\forall i < j$，$s_i < s_j$。

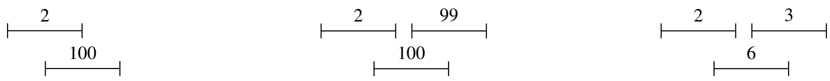
- $OPT(k)$：表示区间集合$\{I_1, I_2, \ldots, I_k\}$上的最优解权值。
- $P(i)$：表示I_i的前驱，当$P(i) = j$时，有$f_j = \max\limits_{1 \le k < i} \{f_k | f_k < s_i\}$，当$I_i$没有前驱时，$P(i) = 0$

目标：寻找一个$\sum w_i$最大的区间子集R，满足$\forall I_m, I_n \in R, m < n$都有$f_m < s_n$。

通俗来讲，就是找出一个彼此时间不重叠的区间序列，使得这个序列的权重在所有可能的序列中，权重最大。

注 以图 6.1为例，$P(6) = 4, P(5) = 3, P(4) = 0, P(3) = 1, P(2) = 0, P(1) = 0$

6.2.2 贪心



(a) (b) (c)

图 6.2: 贪心反例

1. 以最早完成时间排序，反例如图 6.2a
2. 以最大权重排序，反例如图 6.2b
3. 选择冲突最少的区间，反例如图 6.2c

常见的贪心思考方向无法解决带权区间调度问题。

6.2.3 动态规划

6.2.3.1 算法设计

观察带权区间调度问题，对于最优解$OPT(n)$，可以得到如下两个结论：

- 子集R要么包含I_n，要么不包含I_n
- 对于区间I_n
  - 如果$I_n \in R$，$OPT(n) = w_n + OPT(P(n))$
  - 如果$I_n \notin R$，$OPT(n) = OPT(n-1)$

于是我们可以得到该问题的状态转移方程：

$$OPT(n) = \begin{cases} w_n + OPT(P(n)), & I_n \in R \\ OPT(n-1), & I_n \notin R \end{cases} \tag{6.1}$$

因为需要找到最大的权值，上式也可以写成

$$OPT(n) = \max\{w_n + OPT(P(n)), OPT(n-1)\} \tag{6.2}$$

6.2.3.2 算法分析

时间复杂度 在记录最优解的情况下，仅需要填满大小为n的一维数组即可，每次计算$OPT(n)$时会从数组中取已计算的$OPT(n-1)$，因此时间复杂度为$O(n)$。考虑不记录子问题解的情况，在最坏情况下$T(n) = T(n-1) + T(n-2)$，可以发现这是一个斐波那契序列，因此求解该问题的时间复杂度约为$O(1.618^n)$。

空间复杂度 算法执行过程中，会开辟一个空间记录子问题最优解，即空间复杂度为$O(n)$。

6.3 矩阵链乘法

6.3.1 问题描述

假设有矩阵乘法$A \cdot B \cdot C$。其中$A = (n \times m), B = (m \times n), C = (n \times m)$。根据矩阵乘法性质，有$(A \cdot B) \cdot C = A \cdot (B \cdot C)$。前者的时间复杂度为$O(2n^2m)$，后者的时间复杂度为$O(2m^2n)$。于是我们可以看出，不同的矩阵相乘顺序，对结果没有影响，但是对计算时间却有较大的影响，矩阵链乘法即是找到一个最优的相乘顺序。

下面给出矩阵链乘法问题的相关定义。

定义 6.2. 矩阵链乘法问题

给定n个矩阵的链$< A_1, A_2, \dots, A_n >$矩阵A_i的规模为$p_{i-1} \times p_i, (1 \le i \le n)$。求完全括号化方案，使得计算矩阵乘积$A_1 \cdot A_2 \cdot A_3 \cdots A_n$所需的标量乘法次数最少。

- $C(i,j,k)$：记$\underbrace{(A_i \cdot A_{i+1} \cdots A_j)}_{p \times q} \cdot \underbrace{(A_{j+1} \cdot A_{j+2} \cdots A_k)}_{q \times r}$相乘的代价为$C(i,j,k) = pqr$。
- $OPT(i,j)$：记矩阵链$< A_i \cdots A_j >$之间的最优相乘成本为$OPT(i,j)$。

♣

6.3.2 算法设计

对于矩阵链$< A_i \cdots A_j >$，我们可以在i到j之间找到一个切分点k，将问题分解为$< A_i \cdots A_k >$和$< A_{k+1} \cdots A_j >$两个子问题。假设这两个子问题的最优解已知（在动态规划中，可以理解为已存在子问题解的记录），那么可以得到如下的公式。

$$OPT(i,j) = \begin{cases} 0, & i = j \\ \min\limits_{i \le k < j} \{OPT(i,k) + OPT(k+1,j) + C(i,k,j)\}, & i < j \end{cases} \tag{6.3}$$

注 根据算法执行的迭代方向不同，公式可以有多种写法，几种迭代方向参见图 6.3

算法 5: MATRIX-CHAIN-ORDER

Input: 序列$p =< p_0, p_1, \dots, p_n >$，长度为$n+1$，$p_{i-1} \times p_i$为第$i$个矩阵的规模
Output: 代价表$OPT[1..n, 1..n]$，分割表$s[1..n-1, 2..n]$记录$OPT(i,j)$的分割点k

$n = p.length - 1$;
let $OPT[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables;
for $i = 1$ **to** n **do**
    $OPT[i,i] = 0$;
for $l = 2$ **to** n **do**
    for $i = 1$ **to** $n - l + 1$ **do**
        $j = i + l - 1$;
        $OPT[i,j] = \infty$;
        for $k = i$ **to** $j - 1$ **do**
            $q = OPT[i,k] + OPT[k+1,j] + p_{i-1}p_kp_j$;
            if $q < OPT[i,j]$ **then**
                $OPT[i,j] = q$;
                $s[i,j] = k$;
return *OPT and s*

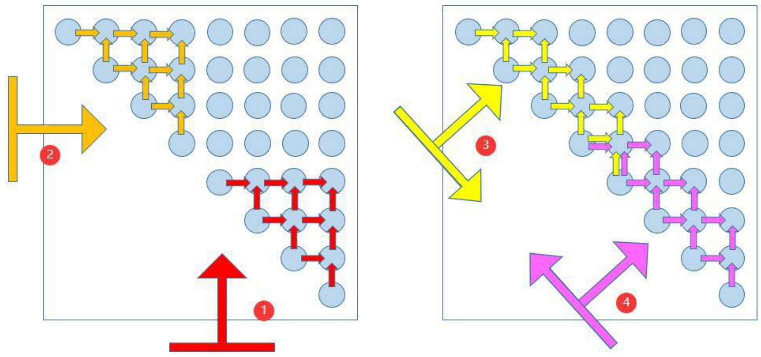


图 6.3: 几种不同的迭代方向

6.3.3 算法分析

时间复杂度 简单分析算法 5的嵌套循环结构，可以得到算法的运行时间为$O(n^3)$。循环嵌套的深度为三层，每层的循环变量(i、j和k)最多取$n-1$个值。

空间复杂度 需要$O(n^2)$的空间来保存OPT和s。

第 7 章 网络流应用

内容提要

- 最大二分匹配问题
- 棒球比赛
- Tiling Problem
- 项目选择

本章是在基于网络流FF算法的基础上，学习网络流的应用。

7.1 最大二分匹配问题

定义 7.1. 二分图

对于无向图$G = (V, E)$，若顶点集V可以分割为两个互不相交子集(X, Y)，使得边集E中任意一条边$e = (u, v)$，都可以满足$u \in X$，$v \in Y$，则称图G为二分图。♣

问题：对于给定的$G = (V, E)$,V为$V_l \cup V_r$,$V_l \cap V_r = \phi$,找到V_l到V_r的最大匹配。

定义 7.2. 匹配

一边集M为边集E的子集，且M中任意两条边无公用顶点（不相交），则称M为图G的一个匹配。♣

定义 7.3. 极大匹配

不是其他任何匹配的子集的匹配。♣

定义 7.4. 最大匹配

极大匹配中包含最多边数的一个匹配称为最大匹配。♣

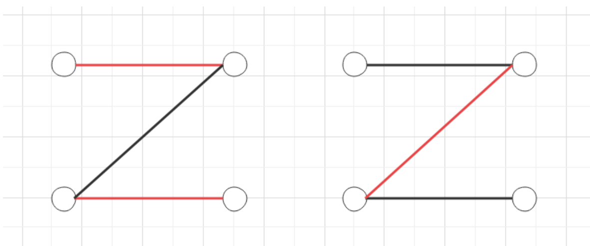


图 7.1: 极大匹配举例

如图 7.1所示，二者都为极大匹配，但是只有左图为最大匹配。
使用如图 7.2的方法构建网络流,之后使用FF算法求解。
网络流解决最大二分匹配问题的时间复杂度为：$O(mn)$。

定理 7.1. 最大流最小割定理

指在一个网络流中，能够从源点到达汇点的最大流量等于如果从网络中移除就能够导致网络流中断的边的集合（对割的另一种理解）的最小容量和。即在任何网络中，最大流的值等于最小割的容量。♡

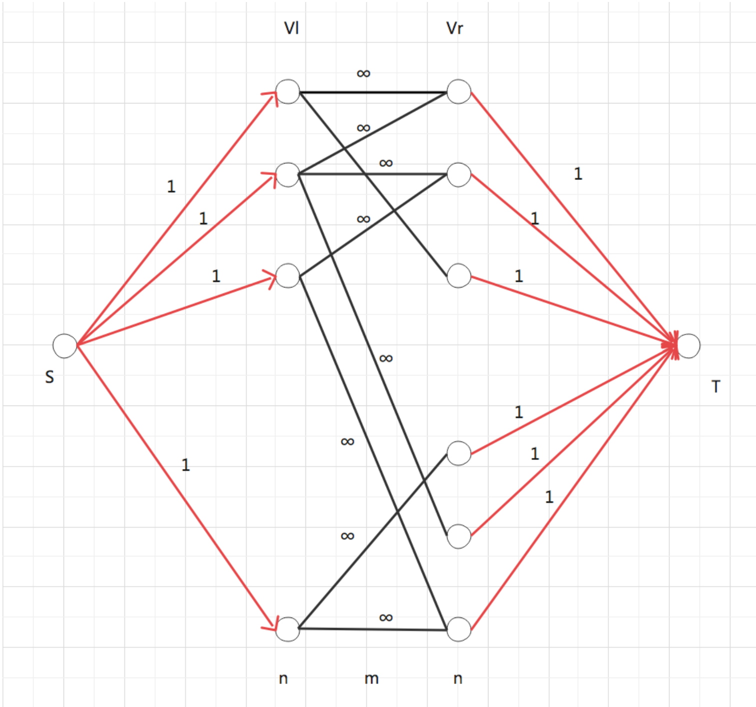


图 7.2: FF算法建模

使用下面一个例子说明该算法的正确性：

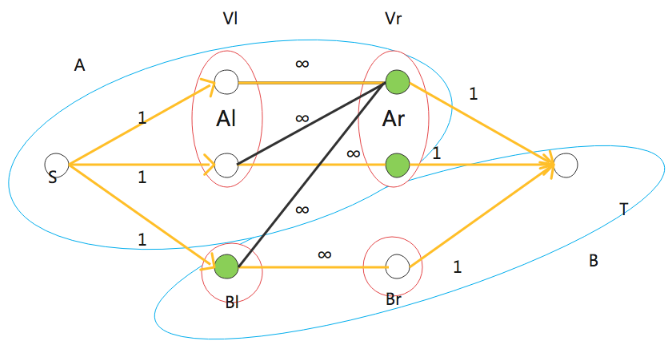


图 7.3: 算法正确性说明

例7.1 如图 7.3所示，给定二分图中，点被分为了V_l和V_r两个部分，构造网络流算法，设置一个起点S，且令S到所有的V_l中的点均有路径，容量为1；设置一个终点T，且从V_r中的点到T均有路径，容量为1。同时，所有V_l与V_r之间的路径容量均设为无穷大，由此可以找到该网络流的最大流和最小割。

- 因为最大流等价于最小割，对于残差图Gf，从S点可以到达的点构成集合A，其他点构成集合B。
- 因为该图为二分图，所以不存在从A_l到B_l的路径，同样的，也不存在从A_r到B_r的路径。
- 因为找到了最小割，而且最小割是有限数，因此不存在从A_l到B_r的路径（已知A_l到B_r的边的容量为正无穷）。
- 最小割值 $C(A, B) = |B_l|$(实际为S流入B_l) + $|A_r|$ (实际为 A_r流入T) = 最大流 = 最大匹配。

对于二分图来说 |最大匹配|=|最小顶点覆盖|。

定义 7.5. 完美匹配

所有的点均被匹配，不存在未被匹配的点。♣

定义 7.6. T (A)

$A \subseteq V_l$,$T(A) = \{b \in Vr|(a,b) \in E\}$。♣

对于一个二分图来说，如果$|V_l| = |V_r| = n$,并且存在完美匹配，那么任取集合$A \subseteq V_l$, $|T(A)| \geq |A|$。

定理 7.2. 霍尔定理

任取$A \subseteq V_l$，若$|T(A)| \geq |A|$,则存在完美匹配。♡

证明霍尔定理的正确性 (证明其逆否命题正确):

- 逆否命题：若不存在完美匹配，则存在集合A，$|T(A)| < |A|$。
- $|V_l| = |V_r| = n$,所以最大匹配$k < n$。
- 找到A，B（即S-T割），$C(A,B) = f(A,B) = k < n$。
- $|A_l| + |B_l| = n$, $|B_l| + |A_r| = k < n$。由此可以推出$|A_l| > |A_r|$,可以推出$|T(A_l)| < |A|$。
- 因此，逆否命题成立，霍尔定理得证。

7.2 骨牌问题

回顾本课程开头提到的骨牌问题，我们试着用网络流的方法再次解决。

例7.2 在$m \times n$的残缺棋盘（有方格缺失）上，填入1×2大小的矩形骨牌，能否用骨牌将棋盘填满？

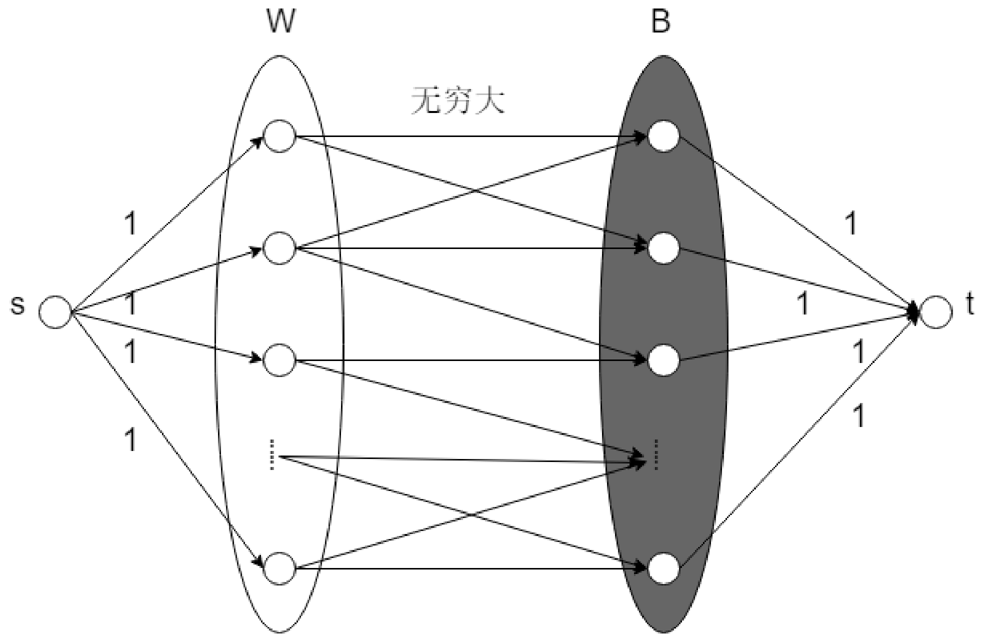


图 7.4: 骨牌问题建模

问题建模： 沿用之前的处理方法，黑白相间地将棋盘做上标记，这样棋盘的方格会被分为两个集合(B,W)。若$|B|$不等于$|W|$，我们可以得出否定的结论；若$|B| = |W| = n$，则在B和W集

之间，相邻的方格连上边，容量正无穷，这样一来， (B,W) 相当于二分匹配中的(V_l, V_r)，就相当于解决一个最大二分匹配问题，建模如图 7.4。

求得最大流$f = n$，则表示该棋盘可以用骨牌填满。

- 算法复杂度分析：F-F算法复杂度为$O(mC)$，在这个模型中，m=n，$C \le 4n$,所以用网络流解决棋盘问题的算法时间复杂度为$O(n^2)$。

7.3 棒球比赛

球队	分数	剩余场次
A	93	A-B......1
B	89	A-C......6
C	88	A-D......1
D	86	B-D......3
		C-D......1

图 7.5: 棒球问题说明

例7.3 有四个棒球队比赛，目前赛场上的得分情况及剩余的场次如图 7.5，问B队有没有机会赢得比赛（并列第一也算赢）？

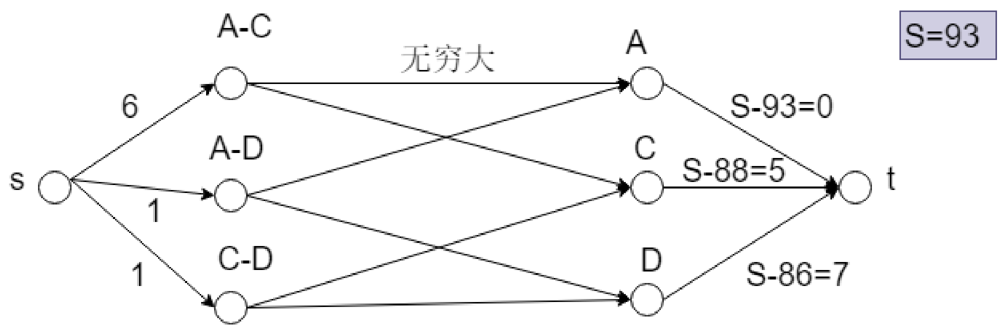


图 7.6: 棒球问题建模

问题建模： 先计算若B队赢得了剩下所有可赢得的分后的总得分S。称参赛双方相同的比赛为一类比赛。将每类比赛作为顶点，通过容量为该类比赛剩余场数的入边交于源点s，出边指向参赛队伍，容量为正无穷；参赛队伍出边指向汇点t，容量为S-（该队伍的当前得分）。如图 7.6所示。

求得最大流$f = C_{in}(t)$，$C_{in}(t)$即t的流入边容量之和，则表示B有机会赢。

- 用F-F算法求模型的最大流和最小割，残差图如图 7.7。

例7.4 如果初始状态，B的得分为90，那么B能否获胜？（B有机会获胜）

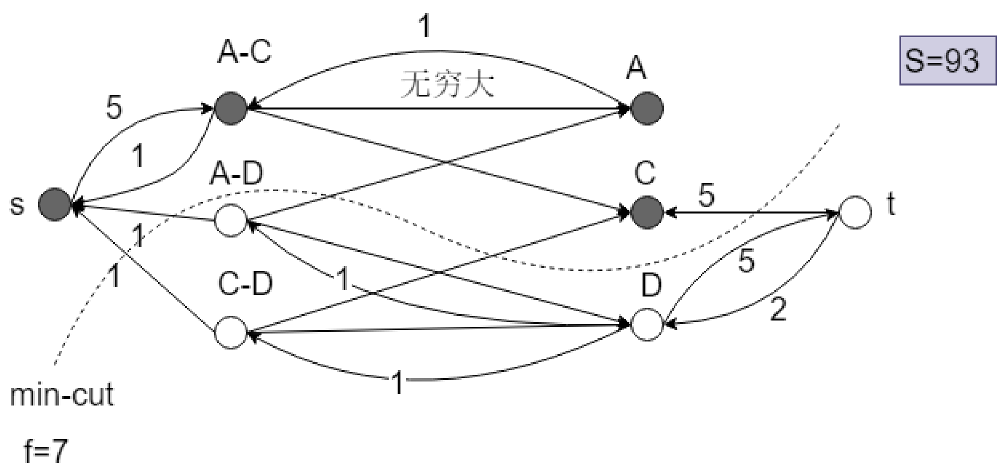


图 7.7: 棒球问题残差图

7.4 项目选择问题

例7.5 有一项目的集合{1，2，……，n}，项目之间有一定相互制约关系，完成每个项目都有一个价值v_i，v_i可正可负。现要求一子集A，使得总价值p最大，且A中任一项目的前驱任务（完成一项目之前所必须完成的项目）也必须在A中。

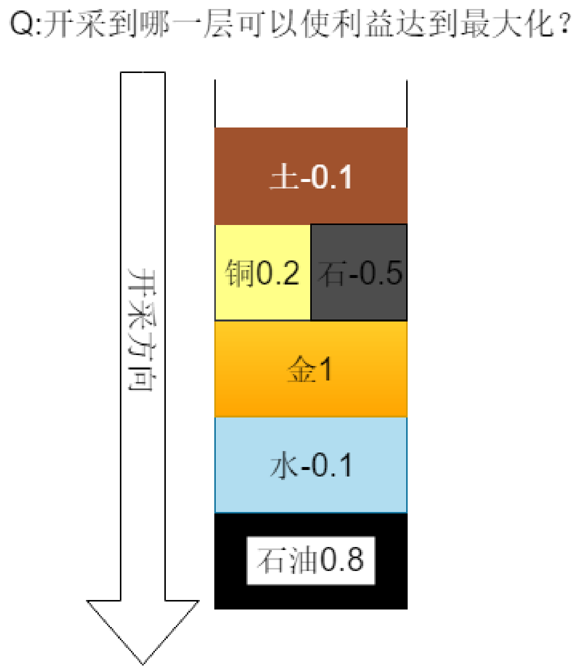


图 7.8: 项目选择示例

为更好地理解问题，给出一更加具体地例子，如图 7.8所示。

问题建模： 由于有先后的制约关系，所以原图应为一个有向图，在此基础上，引入源点s，指向所有价值为负的项目，容量为该项目价值的绝对值；所有价值为正的项目指向汇点t，容量为该项目的价值。对一实例的建模如图 7.9。

- 用F-F算法求模型的最大流和最小割，残差图如图 7.10所示，其中最小割中的集合B所包含的项目即为问题的解。

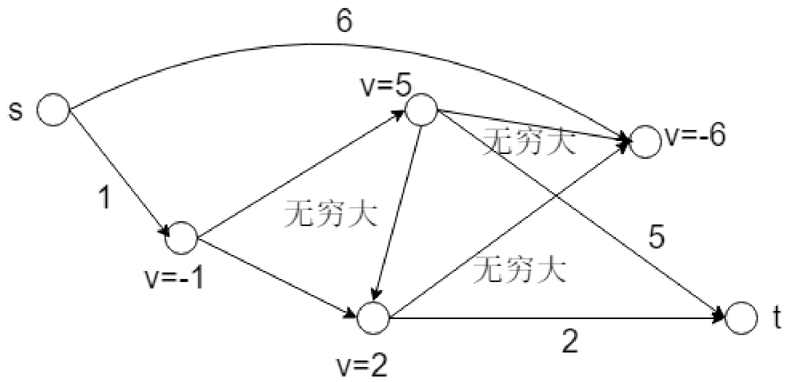


图 7.9: 项目选择建模

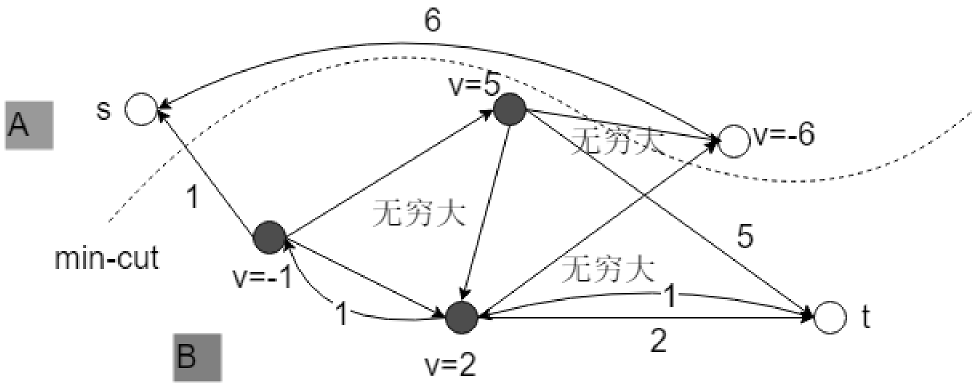


图 7.10: 项目选择问题解决

结果正确性分析：

- 解的可行性：该模型s的所有出边容量为有限值，最坏的情况就是所有负价值点都被选入，此时最小割为s所有出边，因此最小割是包含不到无穷大的边的，此解法也就是可行的。
- 解为最优解：如图 7.11为此模型获得的一般割情形。

从图 7.11可求此结果的价值：

$$v = (p_{B+}) - (p_{B-}) = (p_+) - (p_{A+}) - (p_{B-}) = (p_+) - [(p_{A+}) + (p_{B-})] = (p_+) - cut \tag{7.1}$$

所以所得价值为一常数减去割，要求$maxv$，则割要为$mincut$，即$maxv = (p_+) - mincut$

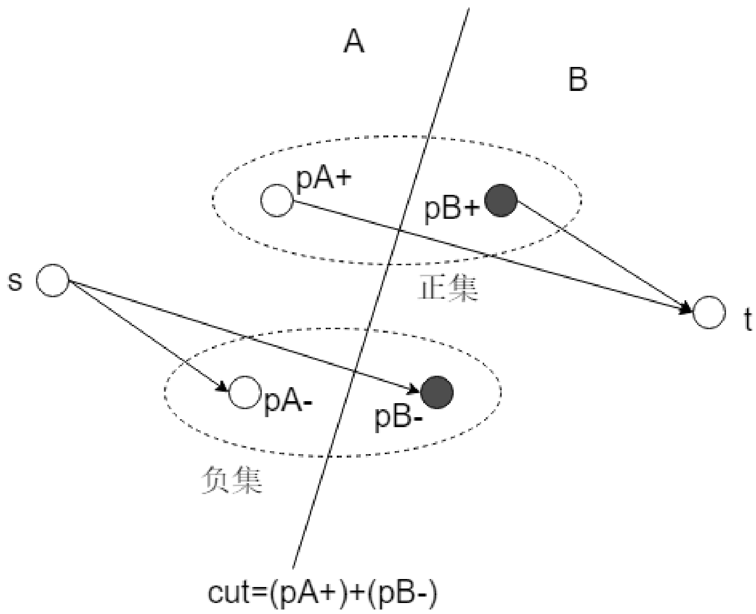


图 7.11: 项目选择问题一般情形

第 8 章 网络流应用之图像分割

本章简单介绍网络流在图像分割上的应用。

定义 8.1. 背景知识

图像是可以看作由一个个像素组成的巨大图，将像素一一用边连接起来，则这些像素点会成为这个巨大图网络的顶点．一个图由前景和背景组成，假设顶点上的值用 a_i 表示，$0 \le a_i \le 1$, a_i 趋近于 0 表示 a_i 为图的背景, a_i 趋近于 1 表示 a_i 为图的前景, 并且设所有属于前景的顶点 a_i 构成集合 A, 所有属于背景的顶点 a_j 构成集合 B. 假设边上的值用 w_{ij} 表示, w_{ij} 设为边的惩罚值, w_{ij} 趋向于 0 表示 "分离" (即 w_{ij} 连接的两个点分别属于前景和背景), w_{ij} 趋向于 1 表示 "在一起" (即 w_{ij} 连接的两个点都属于前景或者背景) 设总的惩罚值为 $A = \min\left(\sum_{i\epsilon B} a_i + \sum_{i\epsilon A}(1-a_j) + \sum_{i\epsilon A, j\epsilon B} w_{ij}\right)$ ♣

8.1 问题实例

8.1.1 问题描述

- 对于下面这个图，利用网络流求解该图前景和背景的最大可能

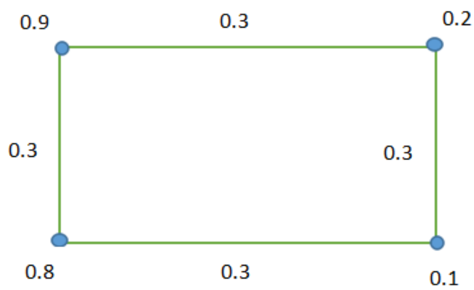


图 8.1: 图片前景背景识别

8.1.2 思路描述

- 图形切割算法通过向图 G (V,E) 添加 S 点和 T 点，将图中所有的顶点，与 S 和 T 建立边，如果一个点与 S 相连，则对应边的权值为该点的值 a_i, 如果一个点与 T 相连，则对应边的权值为1减去该点的值 $1 - a_j$。可以得到下面这个图:
- 根据最大流最小割, 可以得到得到二分图的最大匹配, 可以得到集合A和B, 保证总的惩罚值 $A = \min\left(\sum_{i\epsilon B} a_i + \sum_{i\epsilon A}(1-a_j) + \sum_{i\epsilon A, j\epsilon B} w_{ij}\right)$ 最小, 最小为 $(0.2+0.1)+(1-0.9)+(1-0.8)+0.3+0.3 = 1.2$, 而 A 和 B 分别对应图的前景和背景。

8.2 问题扩展

- 假如一个图的前景不是一个整体，而是有两个分开的部分，比如两只在两个不同位置的猫在一个图中。这样一个算法能否将图像上的前景和背景分开？

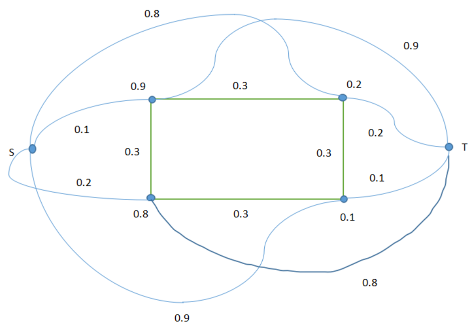


图 8.2: 图片前景背景识别

图 8.3: 图片前景背景识别

第 9 章 近似算法

内容提要

- ❑ 近似算法介绍
- ❑ 顶点覆盖
- ❑ 任务调度
- ❑ 最小带权覆盖
- ❑ MAX-K-SAT

本章讲述了NPC问题的一些近似算法及其质量分析。

9.1 近似算法介绍

9.1.1 引入与定义

求解NPC问题的思路通常包括：

1. 设计通用的指数级时间复杂度算法
2. 针对特例设计多项式时间复杂度算法
3. 根据问题特点设计启发式算法，或借用元启发式算法的框架求解（如蚁群、遗传、退火等算法）
4. 设计近似算法

其中设计近似算法时便要求时间复杂度是多项式级，得到的解可以保证与最优解比差别有限，具体定义如下。

定义 9.1. 近似算法

对一个最小最优问题（最大最优则变为大于号）有多项式级时间复杂度，并对任意实例均有$ALG \leqslant \alpha \cdot OPT$，其中$\alpha$为一个常数，则称该算法为此问题的近似算法。（ALG为该算法结果的质量，OPT为最优解的质量）

9.1.2 近似算法常用证明方法

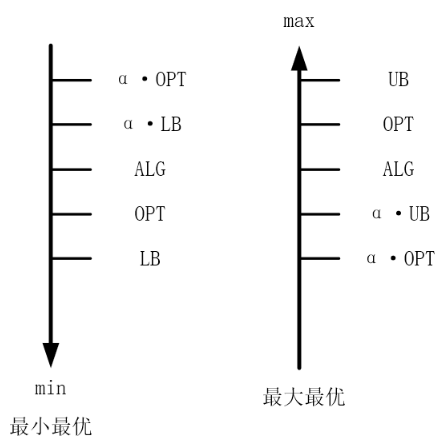


图 9.1: 近似算法证明思路

9.1.2.1 最小最优证明

一般利用图 9.1的证明思路，首先找到OPT下界LB，证明$OPT \geqslant LB$，再想办法证明$ALG \leqslant \alpha \cdot LB$，从而得到$ALG \leqslant \alpha \cdot OPT$，即可证明该近似算法的正确性。

9.1.2.2 最大最优证明

同样利用图 9.1的证明思路，首先找到OPT上界UB，证明$OPT \leqslant UB$，再想办法证明$ALG \geqslant \alpha \cdot UB$，从而得到$ALG \geqslant \alpha \cdot OPT$，即可证明该近似算法的正确性。

9.2 顶点覆盖

本节将介绍一个顶点覆盖的近似算法。

9.2.1 问题描述

定义 9.2. 顶点覆盖问题

对于给定的图(V, E)，找到一个点集$S \subset V$，使得该图所有边都至少有一个端点在点集S中。 ♣

9.2.2 算法描述

算法步骤如下：

1. 找到极大匹配M,相关定义如下：

   定义 9.3. 匹配

   给定一个图G，在G的一个子图M中，任意两边都没有相同的端点，且每个点都有边相连。 ♣

   定义 9.4. 极大匹配

   一个匹配无法再增加任何点和边，则称之为极大匹配。 ♣

2. 输出M中的所有点作为解的点集S

9.2.3 正确性证明

命题 9.1. 求证

该算法始终有$ALG \leqslant 2 \cdot OPT$，在该问题中OTP即为最优解点的数量，ALG即为算法求解的点的数量。 ♠

证明：

1. 证明$OPT \geqslant |M|$（其中|M|为极大匹配的边数）:对于M中任意一条边，其必定至少有一点在OPT中，否则这条边就未被覆盖，与顶点覆盖的要求矛盾。故$OPT \geqslant |M|$
2. 证明$ALG = 2 \cdot |M|$:极大匹配中任意一点度为1，故点的数量即为边的数量的两倍，得证$ALG = 2 \cdot |M|$。
3. 根据上述证明可以得到$2 \cdot OPT \geqslant 2 \cdot |M| = ALG$，得证$ALG \leqslant 2 \cdot OPT$

9.3 任务调度

9.3.1 问题描述

定义任务T为集合$\{t_1, t_2, \ldots, t_n\}$，有$m$台机器$\{M_1, M_2, \ldots, M_m\}$，而一任务调度即将任务$T$中的所有任务分配给这$m$台机器，假设对第$i$台机器，设其被分配的任务为集合$A(i), i \in [1, m]$，则其执行任务的总时间可定义为

$$T_i = \sum_{j \in A(i)} t_j$$

所有机器并行执行任务，则完成所有任务的时间T可表示为

$$T = \max_{i=1}^{m} T_i$$

要求寻找一个算法，使得T尽可能的小，设实际最优解为T_0
该问题为$NP-hard$问题，所以无法在多项式找出最优解，以下给出的均为近似算法，并讨论解与最优解的关系。
例9.1 假设有6个任务，其所需执行时间依次为2,3,4,6,2,2，有三台机器，则其最优解$T_0 = 7$。

9.3.2 贪心算法一[在线算法]

该问题最直接的思路就是贪心算法，将依次取$t_1, t_2, \ldots, t_n$，让m台机器执行，每次挑选的机器的T_i都是最小的，即每次t_i都加入到当前负载T_k最小的机器，设该算法给出的近似解为T_1^*
算法的伪代码如下

算法 6: Greedy-Algorithm1

Result: the minimum T_1^*
begin
  Start with no tasks assigned
  Set $T_i = 0$ and $A(i) = \{\}$ for all machines M_i **foreach** $j \in [1, n]$ **do**
    Let M_i be a machine which T_i the minimum at present
    Aissign task j to machine M_i
    Set $A(j) = A(i) \cup \{j\}$
    Set $T_j = T_i + t_j$

该算法在所给出的例子中运行的结果如下图所示，这里给出的近似解为$T_1^* = 6 + 2 = 8$
实际上我们可以证明，T_1^*和T_0有如下关系：

$$T_1^* \leq 2T_0$$

证明首先需要用到两个显而易见的定理。

定理 9.1

最优解T_0不可能比所有任务在所有机器执行的平均时间还小，即

$$T_0 \geq \frac{\sum_j t_j}{m}$$

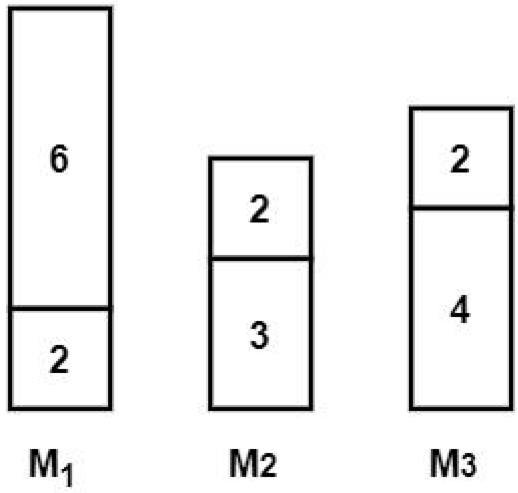


图 9.2: 贪心算法一的结果

定理 9.2

最优解T_0不可能比任务时间最长的那个任务的时间小，即

$$T_0 \geq \max_j t_j$$

因此T_0会大于等于任一t_i。

进一步假设由该贪婪算法所得到的T_1^*由第i台机器产生，如下图所示

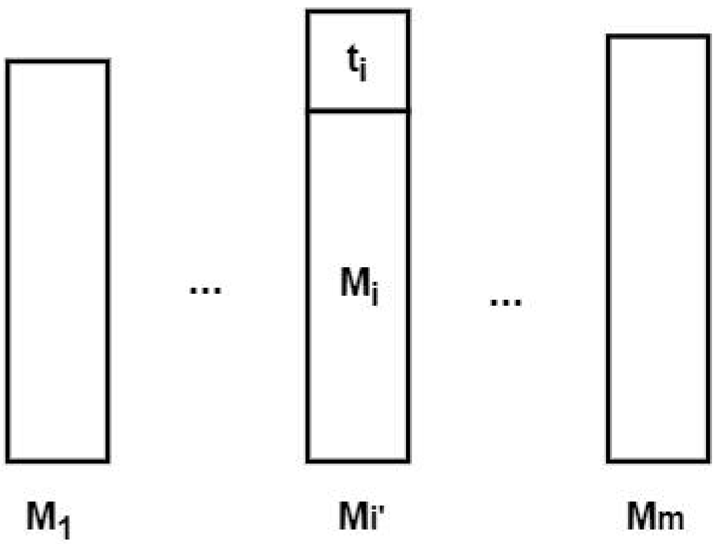


图 9.3: 贪心算法一近似比的证明

证明 由定理2可知，最上面的那个任务t_i的执行时间不会比T_0更大，即

$$t_i \leq \max_j t_j \leq T_0$$

而M_i是所执行的剩下任务的总时间，根据贪婪算法的执行的策略，会导致在使M_i执行最上面的那个任务之前，剩下的任务的时间之和是最小的，由定理1可知，该部分时间和不会超过T_0，即

$$M_i \leq AVG \leq T_0$$

因此，$T_1^* \le 2T_0$成立。

9.3.3 贪心算法二[先排序再贪心]

针对贪心算法一，一个很自然的改进措施，将所有任务按照从大到小的时间进行排序，然后再执行贪心算法一。

算法的伪代码如下

算法 7: Greedy-Algorithm2

Result: the minimum T_2^*

begin

Start with no tasks assigned

Sort tasks in decreasing order of processing time t_j

Assume that $t_1 \ge t_2 \ge \ldots \ge t_n$ Set $T_i = 0$ and $A(i) = \{\}$ for all machines M_i **foreach** $j \in [1, n]$ **do**

Let M_i be a machine which T_i the minimum at present

Aissign task j to machine M_i

Set $A(j) = A(i) \cup \{j\}$

Set $T_j = T_i + t_j$

该算法在所给出的例子中运行的结果如下图所示，这里给出的近似解为$T_2^* = 3 + 2 + 2 = 7$

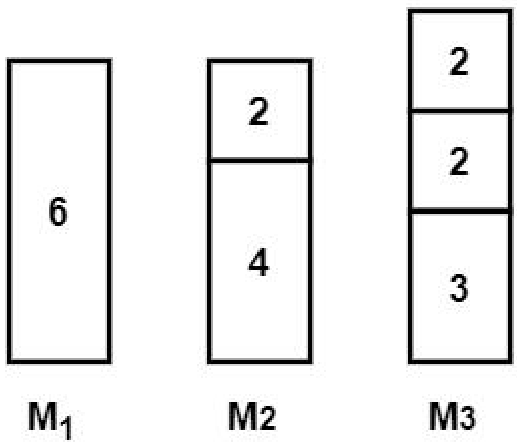


图 9.4: 贪心算法二的结果

实际上我们可以证明，T_2^*和T_0有如下关系：

$$T_1^* \le \frac{3}{2}T_0$$

进一步假设由该贪婪算法所得到的T_2^*由第i台机器产生，如下图所示

证明 首先，如果任务数n小于等于机器数m，那么显然有$T_2^* = T_0$，结论成立。

如果任务数n大于机器数m，则有$T_0 \ge 2t_{m+1}$，因为一开始第1到第m个任务会被依次放在第1到第m台机器上，而第$m+1$个任务必然会放在这几个任务之后执行，而由于任务已经按降序排序，故第$m+1$个任务放好之后，必然会大于$2t_{m+1}$，而它必然小于等于T_0，即$T_0 \ge 2t_{m+1}$，则有

$$t_{m+1} \le \frac{1}{2}T_0$$

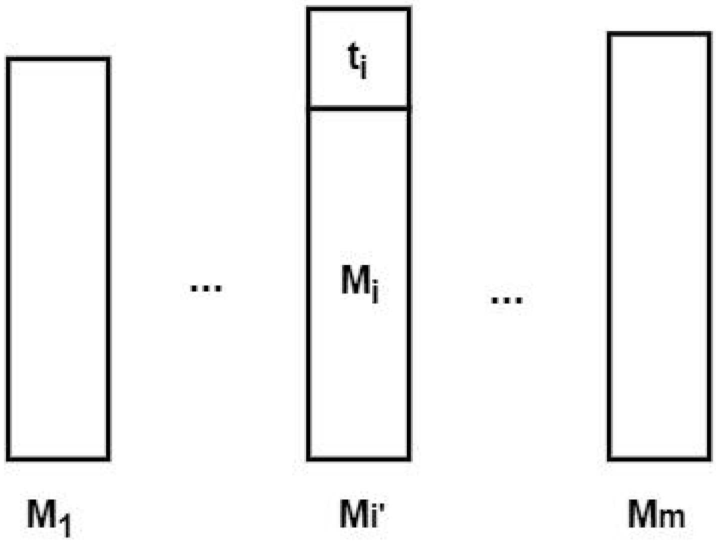


图 9.5: 贪心算法二近似比的证明

在图中，最上面的那个任务t_i的执行时间不会比t_{m+1}更大（因为序号在m+1后面），即

$$t_i \le t_{m+1} \le \frac{1}{2}T_0$$

而M_i是所执行的剩下任务的总时间，根据贪婪算法的执行的策略，会导致在使M_i执行最上面的那个任务之前，剩下的任务的时间之和是最小的，由定理1可知，该部分时间和不会超过T_0，即

$$M_i \le AVG \le T_0$$

因此，$T_2^* \le \frac{3}{2}T_0$成立。

综上所述，$T_2^* \le \frac{3}{2}T_0$成立。

9.3.4 补充

实际上，对于算法二而言，$\frac{3}{2}$并非下确界，实际上的下确界为

$$T_2^* \le \frac{4}{3}T_0$$

此处下确界的含义是，不可能再找到一个比它更小的正数满足上式。

进一步假设由该贪婪算法所得到的T_2^*由第i台机器产生，如下图所示

证明 首先，如果任务数n小于等于机器数m，那么显然有$T_2^* = T_0$，结论成立。

然后，如果任务数n与机器数m的关系满足$n > 2m$时，则有$T_0 \ge 3t_{2m+1}$，对于前$2m+1$个任务，由鸽笼原理，必有一个机器分到3个任务，而由于任务已经按降序排序，故第$2m+1$个任务放好之后，这个机器上的3个任务和必然会大于$3t_{2m+1}$，而它必然小于等于T_0，即$T_0 \ge 3t_{2m+1}$，则有

$$t_{2m+1} \le \frac{1}{3}T_0$$

在图中，最上面的那个任务t_i的执行时间不会比t_{2m+1}更大（因为序号在2m+1后面），即

$$t_i \le t_{2m+1} \le \frac{1}{3}T_0$$

而M_i是所执行的剩下任务的总时间，根据贪婪算法的执行的策略，会导致在使M_i执行最上面的那个任务之前，剩下的任务的时间之和是最小的，由定理1可知，该部分时间和不会超过T_0，即

$$M_i \le AVG \le T_0$$

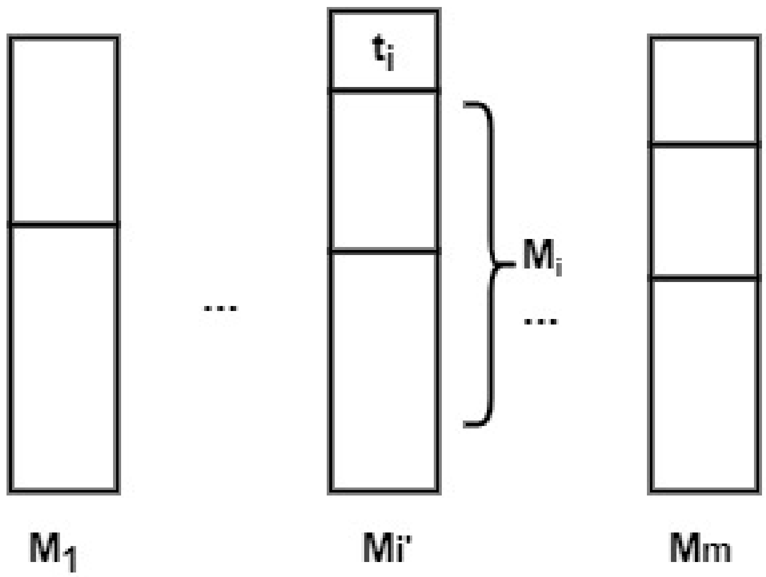


图 9.6: 贪心算法二近似比的证明

因此，$T_2^* \leq \frac{4}{3}T_0$成立。
最后，当$m < n \leq 2m$时，也不能得到T_0，只能得到$\frac{4}{3}T_0$，此部分证明复杂。
综上所述，$T_2^* \leq \frac{4}{3}T_0$成立。
例9.2 假设有$2m+1$个任务，执行时间分别为m到$2m$和$m+1$到$2m$，求解贪心算法二的近似比。该问题的特殊性使得其最优解可以凭直觉得到为$3m+2$，调度策略如下图所示。

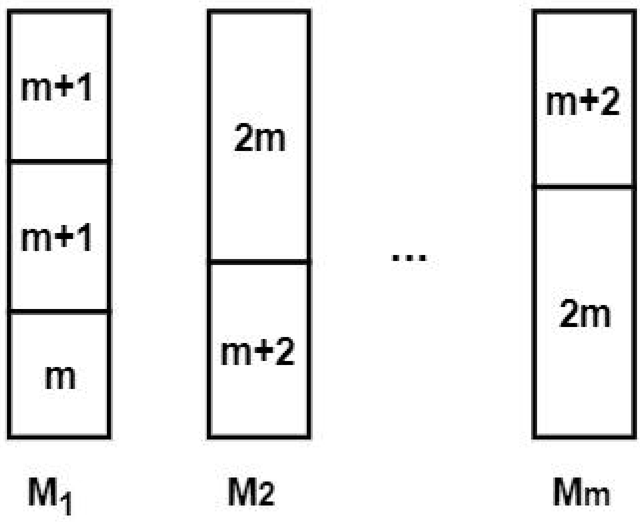


图 9.7: 最优解调度

而利用贪婪算法二得出的近似解T_2^*则为$4m+1$，证明如下。
证明 贪心算法二调度策略如下表所示。
贪心算法二的调度顺序为从第一排从左到右，第二排从右到左，第三排剩一个m，由表可以得到最大时间为$4m+1$。
因此贪心算法的近似比为$\lim_{m\to\infty}\frac{4m+1}{3m+2}=\frac{4}{3}$
更一般的，对于算法解ALG和最优解OPT，有$\frac{ALG}{OPT} \leq \frac{4}{3}-\frac{1}{3m}$。

M_1	M_2	M_3	M_4	...	M_{m-1}	M_m	m的奇偶性
$2m$	$2m$	$2m-1$	$2m-1$	...	$2m-\frac{m}{2}+1$	$2m-\frac{m}{2}+1$	偶数m
					$2m-\frac{m+1}{2}+2$	$2m-\frac{m+1}{2}+1$	奇数m
$m+1$	$m+1$	$m+2$	$m+2$	...	$2m-\frac{m}{2}$	$2m-\frac{m}{2}$	偶数m
					$2m-\frac{m+1}{2}$	$2m-\frac{m+1}{2}+1$	奇数m
m							
4m+1	3m+1	3m+1	3m+1	...	3m+1	3m+1	

9.4 最小带权覆盖

本节将介绍一个最小带权覆盖的近似算法。

9.4.1 问题描述

定义 9.5. 最小带权覆盖问题

对于给定的图(V,E)，各个点有权重w，找到一个点集$S \subset V$，使得该图所有边都至少有一个端点在点集S中，且S中所有点的权重之和比所有可行的解都小。

9.4.2 算法描述

算法步骤如下：

1. 将原问题建模为线性规划问题：原问题是$\forall e=(u,v)\epsilon E$，有$v\epsilon S$或$u\epsilon S$，求$\min\sum\limits_{v\epsilon G} x_v w_v$其中

$$x_v = \begin{cases} 0 & v \notin S \\ 1 & v\epsilon S \end{cases}$$

将其转化为线性规划问题，可变为：

$$\begin{cases} x_v^* + x_u^* \geqslant 1 & \forall e=(u,v)\epsilon E \\ x_v^* \geqslant 0 & \forall v\epsilon G, x_v\epsilon[0,1] \\ \min\sum\limits_{v\epsilon G} x_v^* w_v & \end{cases}$$

2. 使用线性规划求解器求解，再将得到的解转化为原问题的解：

$$x_v = \begin{cases} 0 & x_v^* < 0.5 \\ 1 & x_v^* \geqslant 0.5 \end{cases}$$

9.4.3 正确性证明

命题 9.2. 求证

该算法得到的解是一个顶点覆盖

证明：因为$\forall e=(u,v)\epsilon E, x_v^*+x_u^* \geqslant 1$，故$x_v^* \geqslant 0.5$或$x_u^* \geqslant 0.5$，故$x_v$和$x_u$至少有一个为1，即至少有一点覆盖该边e。得证该算法得到的解是一个顶点覆盖。

命题 9.3. 求证

$ALG \leqslant 2\cdot OPT$

证明：$OPT \geqslant \sum\limits_{v\epsilon G} x_v^* w_v^*$，而又有$x_v \leqslant 2 \cdot x_v^*$，故有 $ALG = \sum\limits_{v\epsilon G} x_v w_v \leqslant 2 \cdot \sum\limits_{v\epsilon G} x_v^* w_v^* \leqslant 2 \cdot OPT$，得证。

9.5 MAX-K-SAT

本节将介绍三个MAX-K-SAT的算法。

9.5.1 问题描述

定义 9.6. K-STA问题

对于一个公式F，其由n个子句$C_1, \cdots, C_n$与运算构成，每个子句又恰好由三个文字或运算构成，即$C_i = L_{i1} \bigvee L_{i2} \bigvee L_{i3}$。每个文字的值取一个变元$X_i$的值或取其非。求一组变元赋值方案，使得公式F为真。♣

定义 9.7. MAX-K-SAT问题

对于一个K-SAT问题，求一组赋值方案使得值为真的子句数量最多。♣

9.5.2 随机算法

9.5.2.1 算法描述

对所有文字$X_i(i = 1, \cdots, n)$等概率随机赋值

$$X_i = \begin{cases} 0 & P = 0.5 \\ 1 & P = 0.5 \end{cases}$$

9.5.2.2 算法分析

- $P(C_i = 1) = 1 - \frac{1}{2^K}$
- 故有$E(ALG) = E(\sum\limits_{i=1}^{n} C_i) = \sum\limits_{i=1}^{n} E(C_i) = n \cdot (1 - \frac{1}{2^K})$
- 又由$ALG \leqslant OTP \leqslant n$
- 可得$\frac{E(ALG)}{OPT} \geqslant \frac{E(ALG)}{n} = 1 - \frac{1}{2^K} \geqslant 0.5$
- 注意这里的分子并不是ALG，而是ALG的期望

9.5.3 确定性贪心算法

9.5.3.1 算法描述

对于随机算法有该递推式：$E(ALG1) = \frac{1}{2}E(ALG1|X_i = 0) + \frac{1}{2}E(ALG1|X_i = 1)$。本算法便基于这一点让本算法的E(ALG2)不小于随机算法的E(ALG1),记变元数为m。

9.5.3.2 算法分析

由算法描述可知，对任何$i = 1, \cdots, m$都有

$E(ALG2|X_{i-1}, \cdots, X_1) = max(E(ALG1|X_i = 0, X_{i-1}, \cdots, X_1), E(ALG1|X_i = 1, X_{i-1}, \cdots, X_1))$

for $i = 1$**to** m **do**
  if $E(ALG1|X_i = 0, X_{i-1}, \cdots, X_1) > E(ALG1|X_i = 1, X_{i-1}, \cdots, X_1)$ **then**
    $X_i = 0$
  else
    $X_i = 1$

故有
$$E(ALG2) = max(E(ALG1|X_i = 0), E(ALG1|X_i = 1)) \geqslant E(ALG1)$$

9.5.4 线性规划算法

9.5.4.1 算法描述

在线性规划建模中，记q_i为C_i的值，y_i为L_i的值，f_{ij}为变元x_i在C_i中的符号。则变为线性规划问题
$$\begin{cases} q_i, y_i \epsilon [0,1] \\ q_i \leqslant \sum\limits_{f_{ij}>0} y_j + \sum\limits_{f_{ij}<0} (1-y_j) \\ \max \sum\limits_{i=1,\cdots,n} q_i \end{cases}$$
线性规划求解完成后，取
$$x_i = \begin{cases} 1 & P = y_i \\ 0 & P = 1 - y_i \end{cases}$$

9.5.4.2 算法分析

线性规划求解完成后，对任一子句不妨假设其符号全为正，便于证明推导：
$$\begin{aligned} &\because q_i \leqslant \sum_{j=1,\cdots,K} y_j \\ &\therefore 1 - \frac{q_i}{K} \geqslant \frac{1}{K} \sum_{j=1,\cdots,K} (1-y_j) \quad (1) \end{aligned}$$
故有
$$\begin{aligned} P(C_i = 1) &= 1 - \prod_{j=1}^{K} (1-y_j) \\ &\geqslant [\frac{1}{K} \sum_{j=1}^{K} (1-y_j)]^K \\ (1) \Rightarrow &\geqslant 1 - (1 - \frac{q_i}{K})^K \\ q_i \leqslant 1 \Rightarrow &\geqslant q_i[1 - (1 - \frac{1}{K})^K] \\ &\geqslant q_i(1 - \frac{1}{e}) \end{aligned}$$

故有

$$
\begin{aligned}
E(ALG) = E(\sum_{i=1}^{n} C_i) &= \sum_{i=1}^{n} E(C_i) \\
&= (1-\frac{1}{e}) \sum_{i=1,\cdots,n} q_i \\
&= (1-\frac{1}{e}) \cdot OPT(LP) \\
&\geqslant (1-\frac{1}{e}) OPT
\end{aligned}
$$

即 $\frac{E(ALG)}{OPT} \geqslant 1-\frac{1}{e}$