

## Lecture 0: 课程简介

编写人: 吴一航 [yhwu\\_is@zju.edu.cn](mailto:yhwu_is@zju.edu.cn)

### 0.1 声明

欢迎大家来到《高级数据结构与算法分析》(Advanced Data Structure & Algorithm Analysis, 简称 ADS) 课程! 这是计算机学院的同学不可或缺的专业基础课程, 将会介绍一些重要但相对于上个学期数据结构基础课程更加复杂的数据结构, 以及基本的算法设计思想与应用。这门课的部分章节可以视为理论计算机科学的入门, 之前的学长学姐们也或多或少都与你们吐槽过这门课的难度, 但相信在各位的努力之下, 这些都不是问题。

本讲义初稿编写于 2023-2024 学年春夏学期, 目前正在积极修订中。编写过程中主要参考浙江大学《高级数据结构与算法分析》课程组 PPT, 以及后面列出的参考书。注意, 本讲义可能存在错误, 请仔细甄别, 若发现错误可以通过邮件等方式联系编写人。

### 0.2 为什么学这门课

一个经典的等式可以概括:

$$\text{Programming} = \text{Data Structures} + \text{Algorithms}$$

算法是解决问题的核心, 而数据结构则可以通过有效管理数据, 减少算法中对输入数据或中间数据的搜索、插入、删除、合并等操作的时间。

### 0.3 这门课学什么

这门课主要包含以下主题:

#### 1. 高级数据结构

- 平衡搜索树: AVL 树, Splay 树, B+ 树, 红黑树
- 堆: 左式堆, 斜堆, 二项堆 (project 中还涉及斐波那契堆)
- 应用: 倒排索引 (搜索引擎算法)

- 时间复杂度分析方法：摊还分析

## 2. 算法分析

- 基本的算法设计思想：回溯法，分治法（含主定理），动态规划，贪心算法
- 难解问题的算法设计理论与方法：NP 问题，近似算法，局部搜索，随机算法
- 其他主题：并行算法，外部排序

## 0.4 有什么参考书

### 1. 《数据结构与算法分析：C 语言描述》(Data Structures and Algorithm Analyses in C), [美] Mark Allen Weiss

同数据结构基础教材。数据结构（AVL 树、Splay 树、左式堆、斜堆和二项堆等）以及部分的算法分析内容（回溯、分治和动态规划等）基本按照本教材思路进行。教材保持老外写书略啰嗦但比较清晰的特点，比 PPT 详细，如果只看 PPT 无法理解可以配合教材阅读。

### 2. 《算法导论》(Introduction to Algorithms) [美] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

令人闻风丧胆的鸿篇巨著，MIT 的算法课教材，有的地方也将其根据作者姓氏首字母简称 CLRS。在这门课中，摊还分析、红黑树、B+ 树、贪心算法、NP 和随机算法等章节基本参考本教材，除此之外，作业中也有难题源于此书，project 中关于斐波那契堆的内容也可以参考。相信很多同学对这本书早有耳闻且有敬畏之心，个人不是非常推荐初学者阅读这本书，因为其中存在不少跳步或者写作不够清晰之处，并且内容太多可能让人无法在一学期内把握核心想法，更适合当个字典挑选部分与课程内容相关的章节、例子等阅读，至少比课程组的 PPT 详细许多。

### 3. 算法设计 (Algorithm Design) [美] Jon Kleinberg, Éva Tardos

同样是算法类图书的经典之作，Jon Kleinberg 和 Éva Tardos 是康奈尔大学的教授，其中 Jon Kleinberg 不到四十岁就拿到了美国三院（美国科学院、工程院、艺术与科学院）院士，Éva Tardos 也是算法博弈论领域的开拓者之一，因此都是顶级大牛。在这门课中，近似算法、局部搜索等章节基本参考本教材，并且也有部分习题出自本教材。我个人非常推荐阅读这本教材，虽然偶尔有点啰嗦，但写作非常清楚且令人感到舒适，其中的例子和习题也都非常经典，也会介绍一些算法设计的思想。

除上述教材外，我个人也推荐基础比较薄弱的同学参考哥伦比亚大学教授（此前是斯坦福大学教授）Tim Roughgarden 的《算法详解》，这本书总共四卷（每卷都特别薄，所以可以看得很快），覆盖了 FDS 和 ADS 的较多内容。Tim 是前面提到的 Éva Tardos 的学生，同样也是算法博弈论领域的开拓者之一。这套书的特点就是相当简单易懂（绝对的入门书籍），作者已经把算法的设计思想、流程等给你嚼烂了喂嘴里了，所以读起来特别亲切（特别是动态规划部分，读者阅读后一定会有所体会）。特别是学 ADS 之前也可以用这本书再回顾一下 FDS 的内容，也会有所收获。

## 0.5 杂谈

可以说这门课是理论计算机方向的入门课程，但显然完全不是一门合格的入门课程。第一课程设计并不合理，在短短一学期内讲清楚学清楚这么多的内容简直是天方夜谭，特别是算法分析部分，很多章节都可以单独扩展成一个学期的课程；第二，课程组授课老师水平参差不齐，很多老师自己也没有专门学过这些理论，并不是做这些方向的，因此讲课也基本只能按照 PPT 来，但很遗憾，课程组的 PPT 非常简洁，只是列举一些例子和解法，没有逻辑关系的说明，也没有什么算法设计的思想，让人很难学明白，甚至不知道自己在学什么，只是看到几个例子，因此学生大部分也只能靠自学，或者不怎么学，总而言之学不明白；第三，这门课教考略分离，这是大课程组难以避免的现象（当然 ADS 至少比数据库、计算机网络等课程要良心一点），因为期末考试是每个老师负责一个小部分，很多老师也不知道自己上课希望给同学们传达什么重要的东西，所以考试无法与授课非常贴合，并且老师出题风格很随机，有的可能就喜欢出一些钻牛角尖的概念，有的喜欢挑难题。

总而言之，我相信大部分同学都认为这门课是梦魇，是计院最难学明白的专业课（之一），但我希望同学们认清一个事实，**这不完全是你们的问题，整个课程安排和课程组都存在非常大的问题**。如果同学们愿意跳出课程设计的种种缺陷，多花时间看一些前面推荐的教材还有一些学长写的讲义 / 笔记，整理一下这些算法背后的思想等，我认为会收获很多，并且会认识到这门课本质并不是非常难，而不是像现在一样学完之后压根不知道自己学了啥，只知道这些东西好像很难，以后再也不愿意碰理论计算机方向了。至于考试，除了部分老师抽风出的题之外，大部分题目应该也是不难的（特别是近年难度还下调了的情况）。

关于这门课的其它信息（如评分标准、其他参考资料和学习建议等），同学们可以参考我和其他学长学姐们共同建设的图灵课程指南的[高级数据结构与算法分析部分](#)。祝大家都能在这门课中学有所成，收获满意的成绩！

CS2045M: 高级数据结构与算法分析

2025-2026 学年秋冬学期

## Lecture 1: AVL 树、Splay 树与摊还分析

编写人: 吴一航 yhwu\_is@zju.edu.cn

### 1.1 引言

第一节课将介绍经典的两种平衡搜索树。作为第一节课，个人认为简单回顾上一学期的内容以引入本学期的内容是非常重要的。在课程简介中已经提到，数据结构的重要性在于可以通过有效管理数据，减少算法中对输入数据或中间数据的搜索、插入、删除、合并等操作的时间。因此，学习数据结构的基本要点就是了解其能支持的操作以及对时间复杂度的改进，从而可以在应用中面对合适的场景选择正确的数据结构。

在数据结构基础中，我们主要学习了如下数据结构以及它们对应的操作的时间复杂度分析，也学习了它们适用的场景：

1. 栈：实现常数级头部添加或删除对象，可用于深度优先搜索；
2. 队列：实现常数级尾部添加和头部删除对象，可用于广度优先搜索。

栈和队列称为线性结构（链表太简单不再重复），以上提到的关键例子（深搜和广搜）只需要用到首尾插入和删除的功能，因此简单的栈与队列是合适的选择。但在数据存储的场景中，数据的更新没有数据的搜索频繁，因此我们更重视搜索的效率，而线性结构往往需要线性时间完成这一点，除非使用有序数组可以用二分查找降低复杂度，但这样插入和删除的操作又变为线性。我们希望搜索、插入、删除的效率都比线性好，那么如何实现呢？

1. 二叉搜索树：遵循左小右大的原则进行插入，因此可以实现在树结构上的“二分”查找。设树高为  $h$ ，则搜索、插入和删除都可以在  $O(h)$  时间完成（还记得如何实现吗？），如果运气不差， $h = O(\log n)$ ，则实现了我们的理想。
2. 优先队列（二叉堆）：为经常要选取最小值或最大值的算法服务，例如操作系统进程调度，Dijkstra 算法（很多贪心算法都是如此）等。它并不支持高效的任意查找，因为只要求结点的儿子一定要比自己大（或小）即可，但可以支持  $O(1)$  的 FindMin 或 FindMax， $O(\log n)$  的插入（还记得吗？可以用数组实现，直接插入在最后一个位置，然后 percolate up）和 DeleteMin（用最后一个元素覆盖根结点，然后 percolate down）。

除此之外，为了支持更快的查询，我们介绍了散列表（hash table）的概念，可以在散列函数选择得较好的时候实现  $O(1)$  级别的查询、插入和删除速度，但如果设计不佳则查询和删除也是线性级别的时

间。这是很现实的问题，事实上前面的例子都告诉我们，要想实现一些特定功能的加速，那必然需要施加一些约束，这些约束可能不好实现，增加设计的复杂程度，也可能会影响其它操作的速度，这其中的 trade off 也使得数据结构的研究充满挑战和趣味。

本讲将介绍平衡搜索树，事实上设计平衡搜索树的目的是非常明确的，因为一般的二叉搜索树查询、插入和删除都可以在  $O(h)$  时间完成（树高为  $h$ ），但存在最差的情况使得  $h$  和  $n$  较为接近，这导致搜索树所有操作效率的降低。因此最简单的想法就是，在二叉搜索树左小右大的大小要求之上，再添加一些有关于树的平衡的要求，使得  $h = O(\log n)$  总是成立的，便可以真正实现搜索、插入和删除速度的提升。

在开始正式内容之前推荐一个数据结构可视化网站，包括了我们讲到的所有数据结构的基本操作可视化功能：<https://www.cs.usfca.edu/galles/visualization/Algorithms.html>

## 1.2 AVL 树

最简单的想法便是强制要求每次插入、删除之后都保证树的“绝对平衡”，这就是 AVL (Adelson-Velskii-Landis) 树的思想。AVL 树的平衡要求每个结点的左右子树高度差的绝对值不能大于 1，然后便可以证明以下引理。

**引理 1.1** 设 AVL 树的结点数量为  $n$ ，则树高  $h = O(\log n)$ .

证明时将给定结点个数求最高的树高问题，转化为给定树高，求最小结点个数的对偶问题，然后转化为斐波那契数列即可。

下面简要讨论搜索、插入和删除操作的具体实现，从而能得到其时间复杂度。搜索与普通的二叉搜索树完全一致，利用左小右大的性质递归或循环均可。因此时间复杂度显然就是  $O(h) = O(\log n)$ .

对于插入，需要  $O(\log n)$  的时间找到插入的位置，接下来的问题是，插入可能会破坏树的平衡结构。需要注意的是，被破坏平衡性的结点显然只能在新加入的结点到根结点的路径上，因为只有这些结点的子树高度变化了。因此需要从插入的结点开始向上走找到第一个被破坏平衡性质的结点。此时，平衡被破坏的原因可以分为四种情况：LL, LR, RL, RR。其中 LL 和 RR 只需一次旋转即可，LR 和 RL 需要两次旋转。此处读者应当结合 PPT 的动态图像，参考 *Data Structures and Algorithm Analyses in C* 理解具体代码实现，包括如何记录和更新结点信息，如何判断平衡条件被破坏，以及如何具体写 rotation 操作（2022 年期末考试编程题直接要求写 AVL 树的实现，得分情况很糟糕）。

**讨论：**回答以下两个问题，(1) 请举出合适的例子肯定或给出证明反驳，(2) 请举出合适的例子反驳或给出肯定的证明（证明无需教科书式的完整，但需要思路清晰，可以有图示辅助）：

1. 在插入的时候是不是有可能很多个结点的平衡性质都被破坏？

2. 如果是的话，一次旋转（Double rotation 也视为一个整体）操作能让所有平衡受到破坏的结点恢复吗？

上面的问题的意义在于，如果一条线上  $O(\log n)$  个结点平衡性质都被破坏，且需要每个顶点都旋转的话，操作代价是比较大的。事实上，两个问题的答案都是肯定的。第一个在课件第 6 页就可以找到例子，第二个问题，回忆只有新加入的结点到根结点的路径上的结点可能平衡属性被破坏，因为此时是插入操作，所以平衡被破坏是因为新加的结点太深了，但如果观察可以发现，无论是 LL、RR 还是 LR、RL，事实上都会使得插入的结点深度减少 1，所以一定能使得路径上所有结点平衡属性恢复（建议结合图示思考）。

至于删除，课件上并未作出要求，感兴趣的同學可以参考[维基百科](#)学习，事实上也是破坏平衡性质然后进行旋转。需要注意的是，删除不再有上面插入只需一次旋转即可恢复平衡的性质，具体可以参考[这个知乎回答](#)。

根据引理以及插入、删除的具体操作（每次旋转是常数级别的，删除最多  $O(\log n)$  次的旋转），我们很容易得到如下定理：

**定理 1.2** AVL 树的搜索、插入和删除操作的时间复杂度为  $O(\log n)$ .

### 1.3 Splay 树

Splay 树的想法一方面来源于希望可以不像 AVL 那样保持严格的平衡约束，但也能保证某种层面（均摊）的对数时间复杂度，另一方面 Splay 树在访问（特别注意访问包括搜索、插入和删除）时都需要将元素移动到根结点，这非常符合程序局部性的要求，即刚刚访问的数据很有可能再次被访问，因此在实现缓存和垃圾收集算法中有一定的应用。

接下来讨论具体操作，简而言之 Splay 树的操作就是在原先操作的基础上加上 splay 操作，而所谓 splay 操作就是通过一系列旋转将访问的结点移动到根结点的位置。因此可以将搜索、插入、删除描述如下：

- 搜索：使用普通二叉搜索树的方法找到结点，然后通过 splay 操作经过一系列旋转将搜索的结点移动到根结点的位置；
- 插入：使用普通二叉搜索树的方法找到要插入的位置进行插入，然后把刚刚插入的结点通过 splay 操作经过一系列旋转移动到根结点的位置；
- 删除：使用普通二叉搜索树的方法找到要删除的结点，然后通过 splay 操作经过一系列旋转将要删除的结点移动到根结点的位置，然后删除根结点（现在根结点就是要删除的点），然后和普通二叉搜索树的删除一样进行合理的 merge 即可。

于是接下来的问题就是 splay 操作的具体实现。PPT 的 12 页给出了一种非常 naive 的想法，就是不断地把访问的结点与其父结点更换父子关系，事实上就是不断用 SingleRotation 翻到根结点的位置。然

而这个例子告诉你，这么做之后虽然把要访问的结点放到了根结点，但其它有的结点被移动到了很深的位置，这是不好的（对比 PPT 的 15 页）。事实上可以论证这样的旋转无法保证从空树开始的连续  $M$  个操作是  $O(M \log n)$  的，这就是 PPT 第 13 页的例子的作用，这个例子从空树开始，通过  $N$  次插入（每次都是  $O(1)$ ）和  $N$  次查找（总共是  $N + \dots + 1 = O(N^2)$  的操作数）构造了长度为  $2N$  但操作  $O(N^2)$  的序列，不符合目标中从空树开始连续  $M$  个操作是  $O(M \log n)$  的要求，因此我们要放弃这种 naive 的、不断把访问的结点与其父结点更换父子关系，从而翻到根结点的方法（可以与 PPT 的 16 页正确的方法比较）。

因此在 PPT 的 14 页给出了合理的旋转方法，注意：

1. case 1 就是简单的交换  $X$  和  $P$  的父子关系，然后调整子树满足搜索树性质即可；
2. zig-zag 的操作方法与 AVL 树的 LR 或者 RL 是一致的，实际上与之前的 naive 的方法也是一样的，三者等价；
3. zig-zig 才是与 naive 的方法不一样的地方！特别注意 naive 的方法先交换  $X$  和  $P$  的位置关系，然后交换  $X$  和  $G$  的位置关系，但是 zig-zig 的标准操作方式是，先交换  $P$  和  $G$  的位置关系，再交换  $X$  和  $P$  的位置关系！这个区别就是它与 naive 方法的唯一区别，却能实现最终均摊的目标；
4. 注意虽然 zig-zig 在 PPT 上写的是 single rotation，但实际上转了两次，这里的 single 大概就是为了表示和 AVL 的 double rotation 不一样吧，zig-zag 有底气称为 double rotation 是因为它和 AVL 一样。

在此再次强调，Splay 树的访问包括搜索、插入和删除，因此 PPT 的 16 页连续插入 1, 2, 3, 4, 5, 6, 7 是会得到图示结果的，因为插入 2 就会把 2 翻到根结点，插入 3 就会把 3 翻到根结点，以此类推，最后 7 在根结点的位置，排成一条斜线。这里特别建议诸位动手推一遍这些操作，防止考试眼高手低。

具体的代码实现以及更加详细的分解动作讲解同样见维基百科。

## 1.4 摊还分析

### 1.4.1 概述

摊还分析的想法来源于我们希望估计一种数据结构经过一系列操作的平均花费时间。然而，平均时间非常难计算，因为每一步都有非常多的选择，连续  $m$  个操作，可能的操作路径是指数级别的。并且有时候平均涉及概率分布等，但我们并不知道确切的分布，因此比较难以计算。

一种最简单的估计方法就是用最差情况分析作为平均情况的上界，例如 Splay 树，每个操作最差都是  $O(n)$  ( $n$  为树中结点个数) 的，因此平均不会比最差情况差，所以也是  $O(n)$  的。然而这样的估计显然是放得太宽了，我们对这个复杂度是非常不满意的，因此需要进行摊还分析。事实上，在最差情况的分

析中，我们忽略了一件事情，就是有的序列是不可能出现的，例如直接在空的树上用  $O(n)$  时间删除。摊还分析则是希望排除掉最差情况分析中把所有不管可能不可能的情况，最差的路径挑出来的这种无脑行为，转而分析所有可能的从空结构开始的操作路径中，最差的平均时间，那么这一时间一定比最差情况分析好，因为排除掉了一些不可能出现的所谓最差序列，但又会大于等于平均时间，因为取的是所有可能序列中最差的那一种。因此摊还分析的时间复杂度一定是平均时间的上界，同时这个上界会比最差情况分析好，这也是 PPT 第 18 页不等式的内在含义。

注意摊还分析要从空结构开始，如果不从空结构开始，则必须要求连续的操作数量足够大，从而抵消初始步骤中可能出现的消耗较大的操作。否则可以思考从一个已经有很多元素的栈里面一次性 Multipop 出所有元素，这一步操作的复杂度显然不再是  $O(1 \cdot 1)$  的。回到 PPT 的 13 和 16 页的例子，我们构造的序列必须从空的开始插入然后才能 find，然后 13 页的 naive 的方法是能构造出很差的例子的，但 16 页重复 13 页的操作并不能构成反例。

接下来介绍三种方法：聚合分析、核算法以及势能法。聚合分析直接使用了上面提到的思想：“摊还分析是考虑可能出现的操作序列中的最差序列”（再次强调，这里的最差不是最差情况分析的最差，最差情况分析的最差会包括不可能出现的序列，但摊还分析要排除不可能的序列），而后两种方法则是基于另一种理解，具体展开时再介绍。

### 1.4.2 聚合分析

根据前面的介绍可知，“摊还分析是考虑可能出现的操作序列中的最差序列”，此时再审视 PPT 第 19 页的例子。我们希望计算出这个数据结构操作的平均时间，但连续  $n$  次操作的序列有  $3^n$  种（每一步都有 3 种操作选择），其中还有一些不可能的要剔除，而且我们也没有假定几种操作出现的概率分布，所以很难计算平均情况，因此需要其它方法。最差情况分析非常暴力，认为最差的操作就是一次 MultiPop  $n$  个元素，因此得到连续  $n$  次操作最差时间为  $O(n^2)$ ，然而怎么可能从空栈开始每次都 Pop  $n$  个元素呢？这种分析明显是严重放大了估计的上界，因此采用摊还分析，只考虑可能序列遇到的最差情况，排除掉最差情况分析臆想的不可能的序列。

于是需要思考这一支持 MultiPop 操作的栈，在从空栈开始的连续  $n$  次操作中，最差的情况是什么：实际上就是先 Push  $n - 1$  个然后最后一次操作一次性 Multipop 出所有元素。为什么这样是最差的呢？事实上简单想想就能理解，无法理解或者希望严谨一些可以看下面的解释：因为  $n$  次操作是固定的，所以目标是固定  $n$  的情况使得总的代价最大，普通的 Push 和 Pop 都是 1 次操作对应 1 个单位代价，所以必须寄希望于序列中单次 MultiPop 代价最大，那代价最大的情况就是只 MultiPop 一次，且就在最后一次，因为如果只 MultiPop 一次，不在最后一次，显然这次 MultiPop 代价比  $n - 1$  小；如果 MultiPop 多次，那么 Push 操作的个数少了，所以 MultiPop 能弹出的比  $n - 1$  少，所以代价也少。

综上，从空栈开始的连续  $n$  次操作中，最差的操作代价是  $2n - 2$ ，因此摊还分析复杂度为  $O(1)$ 。

### 1.4.3 核算法

核算法的思想来源于，我们希望计算平均成本，完美的平均成本就是截长补短，即把时间长的操作成本摊到时间短的操作上，但完美的截长补短很难做到，那么我们尽力做到，且要保证从长成本大的操作截取的部分都要分摊出去，时间不能变少了，否则会导致总时长变短，从而比平均时间短，那么摊还时间复杂度会成为平均时间的下界，就失去意义了。说起来有些抽象，直接写下来表达式吧。所谓截长补短，设第  $i$  种操作的真实成本是  $c_i$ ，截长补短的摊还成本是

$$\hat{c}_i = c_i + \Delta_i,$$

其中  $\Delta_i$  就是截的长（负值），或者补的短（正值），并且要保证摊还成本比平均成本大，这样分析出来的摊还成本才是平均成本的上界，即要求

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i,$$

也即  $\sum_{i=1}^n \Delta_i \geq 0$ 。有了这一基础之后再来审视 PPT 第 21 页的例子。因为希望一次操作摊还成本为  $O(1)$ ，所以希望这三种操作的摊还成本都是常数级别的，这样只要使得  $\sum_{i=1}^n \Delta_i \geq 0$ ，或者说  $\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$ ，那就直接证明了结论。但是我们知道，MultiPop 的代价比较大，把它调整为常数需要对应的  $\Delta < 0$ ，故必然需要代价小的操作  $\Delta > 0$ ，所以才有了例子中把 Push 操作代价调整为 2，然后我们可以利用  $\text{size}(S) \geq 0$  这一约束证明  $\sum_{i=1}^n \Delta_i \geq 0$ 。

事实上，核算法这一名字也非常形象。这就像你的银行卡，余额不可以为负数，所以你必须在花钱之前先存钱，你知道 MultiPop 是个花钱的事情，花钱之前必须存钱。当然你也可以说，你希望你的平均绩点保持在 4.0 以上，你知道 ADS 这门课你可能会挂科，所以你之前的微积分、线性代数、FDS 等就需要为这次挂科存下足够高的均绩。

### 1.4.4 势能法

事实上核算法尽管非常形象，但你要为每个操作设计一个摊还代价  $\hat{c}_i = c_i + \Delta_i$  是不一定像上面的例子那么简单的，况且你还要保证  $\sum_{i=1}^n \Delta_i \geq 0$ （比如思考 Splay 树这种复杂的结构）。所以希望有一个更统一的方法解决这个问题：我们不再把目光局限于每个操作，而是给整个结构定义一个势函数，这个势函数描述了这个结构不同状态。基于这一思想，形式化而言，规定第  $i$  次操作的摊还代价为

$$\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1})), \quad (1.1)$$

注意这里的  $i$  不再是核算法中表示第  $i$  种操作，而是  $n$  个操作组成的序列中的第  $i$  个。因此这里的含义是，每一步的摊还代价等于真实操作的代价加上势函数的变化，于是我们求和有

$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \left( \sum_{i=1}^n c_i \right) + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

为了使得摊还成本是平均成本的上界，仍然需要满足  $\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$ ，因此需要  $\Phi(D_n) \geq \Phi(D_0)$ ，这一点在设计函数的时候很容易满足，因为可以通过调整使得  $\Phi(D_0) = 0$ ，即初始状态势能为 0，则只需要其它任何状态的势能都不会小于 0 就可以了。

基于上述定义分析支持 MultiPop 操作的栈，自然地，我们希望真实操作成本高的 MultiPop 在势能法下的摊还代价（式1.1）回归到常数，从而达到总复杂度  $O(1)$  的目标。这就要求势能函数在经过 MultiPop 操作后有较大的下降，抵消 MultiPop 的真实操作代价。事实上在 MultiPop 后，栈中的元素个数下降了很多，因此一个自然的想法就是将势能函数定义为栈中的元素（并且这一定义符合  $\Phi(D_n) \geq \Phi(D_0)$ ），从而在这一势能函数定义下，三种操作的摊还代价都是常数，这也就是 PPT 第 23 页的思路。

读者不难发现势能法和核算法有些类似，都是通过对真实操作代价进行一些改变，实现最后证明的目标。只是核算法是对每一类操作添加  $\Delta_i$ （从而要求  $\sum_{i=1}^n \Delta_i \geq 0$ ），势能法是定义一个统一的势能函数来决定每一步对真实代价的改变量（从而要求  $\Phi(D_n) \geq \Phi(D_0)$ ）。

此外，从上面的例子中可以得到一个经验：为了实现摊还分析得到比较好的复杂度结果，在核算法和势能法中，必然需要对真实代价比较高的操作添加一个比较大的负数项，从而得到一个符合要求的摊还代价。特别地，在势能法中，这就要求势能函数在代价较高的操作后会有较大的下降，读者在后续章节的分析以及完成相关习题时都可以回忆这一基本的摊还分析原则。最后，其实势能法就是借用了物理里面的势能的思想，给数据结构定义一个与结构本身特点相关的量。比如重力势能是质量和高度的函数，但树的势能可能是树高的函数，也可能是树的结点总数的函数甚至更复杂等等。

#### 1.4.5 综合应用

总结一下上面的三种方法，第一种聚合分析建立在寻找可能出现的操作序列中的最差情况的想法，后两种建立在截长补短的思想，这两种思想是描述摊还分析等价的两种思想。然后要注意，在核算法中是对每个操作直接定义摊还代价，而势能法是利用势能函数对每一步操作间接定义摊还代价。还需要注意的是，聚合分析中所有操作的摊还代价是统一的，就是最后的平均值，因为是无差别的分析。但核算法不同操作的代价是直接定义的，可以是不同的，势能法也是可以不同的（栈的例子就可以看出）。

如果希望看到更多的例子应用这一美妙的思想的读者可以参考《算法导论》（第三版）17.4 节动态表的经典例子。这里给出一些考试可能出现的题目作为例子：

**例 1.3** A queue can be implemented by using two stacks  $S_A$  and  $S_B$  as follows:

- To enqueue  $x$ , we push  $x$  onto  $S_A$ .
- To dequeue from the queue, we pop and return the top item from  $S_B$ . However, if  $S_B$  is empty, we first fill it (and empty  $S_A$ ) by popping the top item from  $S_A$ , pushing this item onto  $S_B$ , and repeat until  $S_A$  is empty.

Assuming that push and pop operations take  $O(1)$  worst-case time, please select a potential function  $\Phi$  which can help us prove that enqueue and dequeue operations take  $O(1)$  amortized time (when starting from an empty queue).

A.  $\Phi = 2|S_A|$

B.  $\Phi = |S_A|$

C.  $\Phi = 2|S_B|$

D.  $\Phi = |S_B|$

**例 1.4** A sum list  $L$  is a data structure that can support the following operations:

- $Insert(x, L)$ : insert the item  $x$  into the list  $L$ . The cost is 1 dollar.
- $Sum(L)$ : sum all items in the list  $L$ , and replace the list with a list containing one item that is the sum. The cost is the length of the list  $|L|$  dollars.

Now we would like to show that any sequence of  $Insert$  and  $Sum$  operations can be performed in  $O(1)$  amortized cost per  $Insert$  and  $O(1)$  amortized cost per  $Sum$ . Which of the following statement is TRUE?

- A. We use the accounting method that charges an amortized cost of 2 dollars to  $Insert$  and 0 dollar for  $Sum$ .
- B. We use the potential function to be the number of elements in the list.
- C. We use the potential function to be the opposite number of elements in the list.
- D. Neither method can show the amortized cost for  $Insert$  and  $Sum$  is  $O(1)$ .

#### 1.4.6 Splay 树的摊还分析

接下来将完成本讲的最后一个使命：证明 Splay 树的每个操作的摊还代价是  $O(\log n)$  的。PPT 第 24 页给出了设计合理势函数的思想：

### 1. 回忆

$$\sum_{i=1}^n \hat{c}_i = \left( \sum_{i=1}^n c_i \right) + \Phi(D_n) - \Phi(D_0),$$

我们心里知道，最后合理的摊还分析结果是  $O(\log n)$  的，所以  $\Phi(D_n)$  应当是  $O(n \log n)$  量级，所以应该带个对数函数。

### 2. 我们还希望在计算的时候，旋转导致的势能的变化尽量简单，这需要两个状态之间势函数相减时很多项之间可以互相消去，因此定义树 $T$ 的势函数为

$$\Phi(T) = \sum_{i \in T} \log S(i),$$

即把所有结点的后代表数取对数然后求和，这样在旋转过后大部分结点后代表数不变，只有少量结点发生变化，于是减法时可以消去。

总而言之这两点可以说是一些直觉，但实际构造的时候必然要经过一系列的尝试，不可能一蹴而就。此外，在证明过程中需要用到一个引理，具体分析 PPT 或者维基百科已经比较详尽，此处不再赘述：

**引理 1.5** 若  $a + b \leq c$ , 且  $a$  和  $b$  均为正整数，则

$$\log a + \log b \leq 2 \log c - 2.$$

最后一个问题是，为什么不用每个结点对应子树的树高求和作为势函数？实际上代入后面的分析，会发现因为树高并非  $O(\log n)$  的，因此最后一步无法得到  $O(\log n)$  的摊还代价。总而言之，基于一系列的分析（具体见 PPT），可以得到如下结论：

**定理 1.6** 记  $R(X) = \log S(X)$ , 对于结点  $X$  在第  $i$  次操作时，有

1. *zig* 操作的摊还代价  $\hat{c}_i \leq 1 + (R_i(X) - R_{i-1}(X)) \leq 1 + 3(R_i(X) - R_{i-1}(X))$ ;
2. *zig-zag* 操作的摊还代价  $\hat{c}_i \leq 2(R_i(X) - R_{i-1}(X)) \leq 3(R_i(X) - R_{i-1}(X))$  (这一步用到了前面的引理);
3. *zig-zig* 操作的摊还代价  $\hat{c}_i \leq 3(R_i(X) - R_{i-1}(X))$ ;

然而这还不够，因为搜索、插入和删除操作需要经过一系列的旋转操作才能实现。于是需要进一步分析将  $X$  从它所在的位置连续移动到根结点总共需要的摊还代价。实际上，假设  $X$  的高度为  $H(X)$ , 则可能的旋转次数为

$$k = \begin{cases} H(X)/2 & H(X) \text{ 为偶数} \\ (H(X)-1)/2 + 1 & H(X) \text{ 为奇数} \end{cases},$$

其中偶数的情况都是 zig-zig 或 zig-zag，奇数会先进行  $(H(X) - 1)/2$  次 zig-zig 或 zig-zag，最后进行一次 zig。放缩成最差的情况，就是都要有  $k$  次 zig-zig 或 zig-zag，加上最后一次 zig，则将  $X$  旋转到根结点的操作摊还代价为

$$\begin{aligned}\sum_{i=1}^{k+1} \hat{c}_i &\leq 1 + 3(R_{k+1}(X) - R_k(X)) + \sum_{i=1}^k 3(R_i(X) - R_{i-1}(X)) \\ &= 1 + 3(R_{k+1}(X) - R_0(X)) = O(\log n).\end{aligned}$$

回忆搜索算法需要先找到  $X$ ，然后将  $X$  旋转到根结点，事实上寻找  $X$  不可能比旋转更差，因为只需要一路下行，不需要指针换来换去，而二者路径总长度还是一样的，所以寻找  $X$  的摊还复杂度也必定是  $O(\log n)$ ，因此整个搜索  $X$  的算法复杂度就是  $O(\log n)$ 。至于插入和删除，用同样的分析方式，结合它们的具体算法就可以发现也是  $O(\log n)$ 。但是要注意的是，插入的时候插入的结点到根结点路径上所有的结点的势函数值会增大，如果要严谨证明必须还要说明这些结点势函数的增量也是  $O(\log n)$  的，这里不给出详细证明，有兴趣的同学可以自行搜索资料或者自己尝试。总而言之，可以总结出如下定理：

**定理 1.7** *Splay* 树的搜索、插入和删除操作的摊还复杂度均为  $O(\log n)$ 。

## Lecture 2: 红黑树与 B+ 树

编写人: 吴一航 yhwu\_is@zju.edu.cn

## 2.1 红黑树

### 2.1.1 红黑树的引入与定义

为了实现平衡搜索树, AVL 树用一种非常简单暴力的思想, 即保持严格平衡 (任一结点左右子树高度差绝对值最多为 1) 然后用旋转维持平衡。红黑树在某种程度上是希望放松这一假设, 从另一角度——给结点染色, 定义其它平衡因子——也给出了一种解决方案。首先来看红黑树的五条性质, 或者说满足这五条性质的搜索树称为红黑树:

**定义 2.1** 一棵红黑树是满足下面性质的二叉搜索树:

1. 每个结点或者是红色的, 或者是黑色的;
2. 根结点是黑色的;
3. 每个叶结点 (*NIL*) 是黑色的;
4. 如果一个结点是红色的, 那么它的两个子结点都是黑色的;
5. 对每个结点, 从该结点到其所有后代叶结点的简单路径上, 均包含相同数目的黑色结点。

之后的讨论我们会经常使用上面的序号代表该序号对应的性质。PPT 第 2 页的右上角图示给出了红黑树结点的形式, 除了常规的父子以及键值信息之外, 红黑树还有一个颜色的属性, 即当左/右子结点为空时, 子结点必须是黑色的 *NIL* 结点 (它没有键值), 这与一般的 *NIL* 不一样。下面逐条解读以深入理解定义:

1. 第一条的言外之意是: 只有这两种颜色, 这一点或许很显然, 但在之后讨论红黑树删除的时候有用处。
2. 对于第二条, 之后会经常遇到在插入、删除调整时, 根结点变为红色的情况, 这时候如果直接将根结点染成黑色, 对于其它四个条件是完全没有影响的。因此后面其实并不会太关心根结点的颜色问题, 因为不是黑色直接染成黑色完全不影响其它性质。

3. 这一点或许有些迷惑，但既然定义如此必定有其作用，其作用一方面在于使得红黑树有一定的平衡性：如果没有 NIL 结点，完全 skew 的树也是可以符合其余几条规定的，另一方面将会在讨论删除的时候体现。

对于最后两条，它们的意义在于可以直接决定红黑树的树高是  $\log n$  级别的，因此是平衡搜索树。在介绍定理之前，需要先引入两个定义：

- 定义 2.2**
1. 称 NIL 为外部结点，其余有键值的结点为内部结点。
  2. 从某个结点  $X$  出发到达一个叶子结点 (NIL) 的任意一条简单路径上的黑色结点个数（不含  $X$  本身）称为  $X$  的黑高，记为  $bh(X)$ 。根据定义第五条，这一定义是合理的，因为从  $X$  出发出发到达一个叶子结点的任意一条简单路径上的黑色结点个数相同。除此之外，定义整棵红黑树的黑高为其根结点的黑高。

然后可以引入如下定理：

**定理 2.3** 一棵有  $n$  个内部结点的红黑树的高度至多为  $2 \log(n + 1)$ 。

具体证明这里不再重复，这里想说的是，这一定理事实上无需证明就能非常直观地得到。因为如果仔细观察定义第五条，就会明白，黑高实际上就是红黑树最关键的平衡因子：如果舍去全部的红色结点，剩下的树的结点个数一定大于等于高度为  $bh(\text{root})$  的完全平衡二叉树的结点个数，大于的情况看 PPT 第二页的例子就可以知道，因为删掉红色结点后可能不是二叉，这就是证明的第一个步骤：证明以  $X$  为子结点的子树至少有  $2^{bh(x)} - 1$  个内部结点。而根据第四条，任何路径上黑色结点必定占到至少一半的数量，因为红色不能是父子关系，所以有了证明的第二步，树高最多为黑高 2 倍。

总结而言，对于红黑树，黑高是绝对严格的平衡要求，而红色结点则是少量不平衡的因素，并且定义控制了红色结点的个数，也就控制了不平衡因素的影响，因此红黑树还是可以保持一定程度的平衡的。有了这一结论，至少现在可以知道对于搜索操作，红黑树的时间复杂度是  $O(\log n)$  的，至于插入和删除，还需要下面进一步讨论。

**【讨论 1】** 证明：在一棵红黑树中，从某结点  $X$  到其后代叶结点的所有简单路径中，最长的一条路径的长度至多是最短一条的 2 倍。

实际上这个问题和上述定理证明基于同样的思想，因为所有简单路径的黑色结点数量都是相同的，最短的路径就是全黑路径，最长的就是黑红相间的路径（因为红色不能是父子关系）。

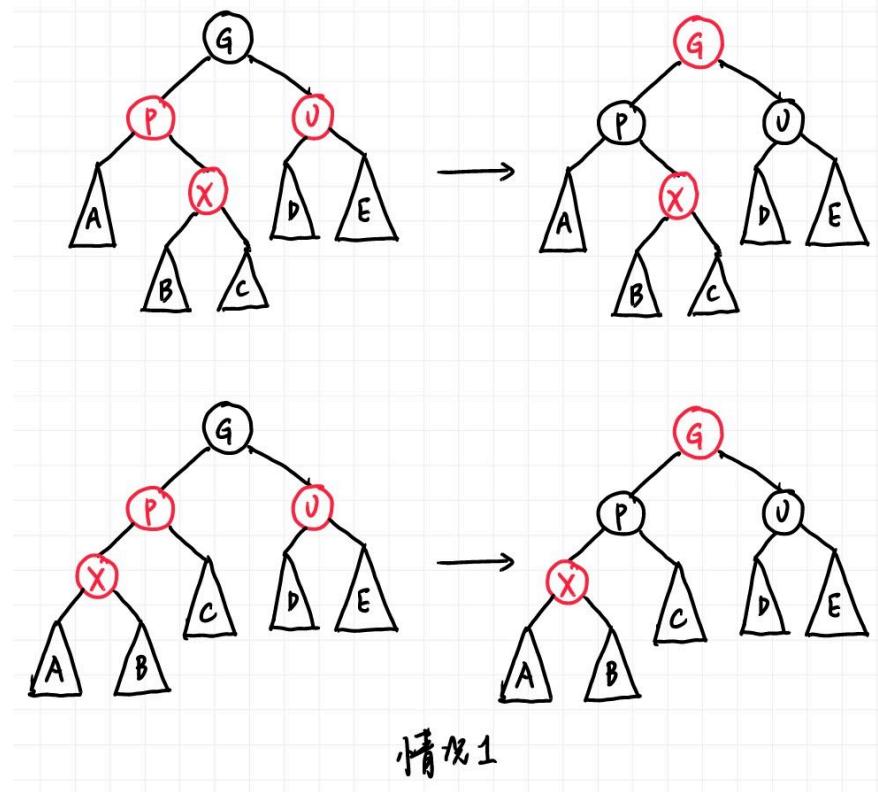
### 2.1.2 红黑树的插入

与 AVL 树类似，插入结点时有可能出现平衡性质被破坏，因此需要一定的调整。那么自然面临的第一个问题就是：插入应该插入黑色结点还是红色结点？假设每次都插入黑色结点，那么定义第五条一定

会被破坏，因此每次插入都必须调整（除了从空树插入的那次）；但是如果插入红色结点，定义第二条（仅在插入空树时）和定义第四条仅仅是有可能被破坏，因此直观来看插入红色结点会比插入黑色结点需要的调整更少。除此之外，在学完红黑树删除后，红黑树删除黑色结点时的复杂情况也暗示了插入红色应当是更好的选择。

接下来开始介绍红黑树的插入操作，根据前面的讨论，插入的结点首先都是红色的。最简单的情况是插入在黑色结点下面，这时无需任何调整，因为没有破坏任何性质；其次是插入空树时，这时直接把结点染黑即可恢复定义第二条。因此下面只讨论最复杂的情况，即插入到了红色结点的子结点的位置。此时唯一被违反的就是定义第四条。因此如果要解决这一问题，自然的想法就是，在不影响其它平衡性质（主要是第五条）的前提下，通过一些染色和旋转使得没有父子都是红色。主要分为以下三种情况讨论（假设插入的结点为  $X$ ，先讨论  $X$  插入在祖父  $G$  左侧的情况）：

**情况 1： $X$  的叔叔（即父亲的兄弟）是红色的， $X$  无论左右孩子都是该情况**



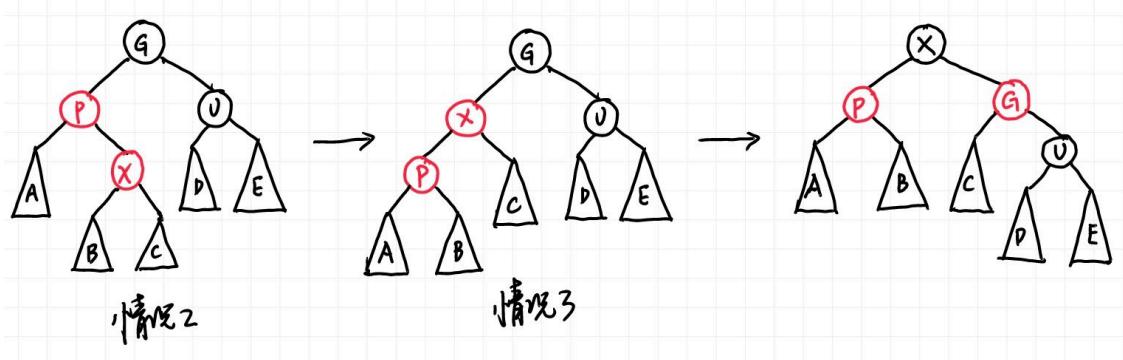
需要注意的是，这里所有结点都带子树，一方面至少有 NIL 结点，另一方面这可以不仅仅表示刚刚插入的情况，也可以表示经过几次调整后还在被这种情况困扰，因此更具一般性。注意  $G$  一定是黑色，因为  $P$  是红色，插入前它们就在红黑树中，因此不可能违背定义第四条。

下面开始解释解决方案，事实上此时解决方案非常直白：它的父亲和叔叔都是红色，如果把红色看成一种 debuff， $X$  都求不能把这个 debuff 甩给他们，只能求助于祖父，于是直接把  $X$  的父亲和叔叔染黑，祖父染红。此时完全不影响黑高性质，但问题并没有解决，因为祖父的父亲可能还是红色！但是，问题

被往上推了，最差的情况也是一直上推到把问题交给根结点，那么根结点此时只需要直接染黑就可以解决问题。当然也可能问题上推后变为下面两种情况，接下来我们进行解释：

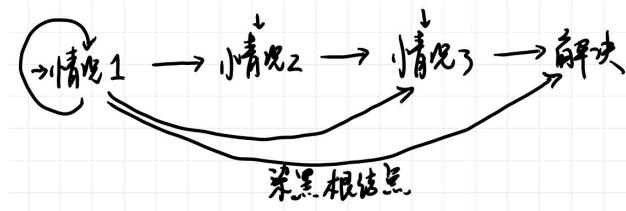
**情况 2:**  $X$  的叔叔（即父亲的兄弟）是黑色的，且  $X$  是右孩子

**情况 3:**  $X$  的叔叔（即父亲的兄弟）是黑色的，且  $X$  是左孩子



这里直接借用 AVL 树的想法，把红色视为 trouble，根据 trouble 是 LR 还是 LL 按 AVL 树的调整方式旋转，染色时，局部情况是根黑色，第二层红色。

当然还有  $X$  插入在祖父  $G$  右侧的情况，这是完全对称的，因此不再重复。总结而言，就是第一种情况叔叔红，直接改颜色，问题推给祖父，第二、三种情况叔叔黑且 LR、LL（对称的有 RL、RR），用 AVL 的旋转，最后局部情况是根黑色，第二层红色，大致流程如下：



即如果插入后直接落入情况三，只需要一次旋转染色即可解决，直接落入情况二，一次旋转进入情况三，再一次旋转染色即可解决，但如果落入情况一，一次调整后可能还在情况一，可能直到最后都是通过情况一加上染黑根结点解决，也可能几次调整后进入情况二或三后解决。根据这一流程可知，红黑树插入最多可能的旋转次数为 2（因为只有情况 2 和 3 会要旋转进入情况 2 后 1 次旋转必定进入情况 3，进入情况 3 后 1 次旋转必定解决），然后更改颜色最多是  $O(\log n)$  次，因为进入情况 2 或 3 只需要一次染色，在情况 1 最差也是每两层染一次色，而已经证明红黑树的最大高度是  $O(\log n)$  的。因此插入操作包括  $O(\log n)$  的搜索时间，加常数的旋转，加  $O(\log n)$  的染色，因此还是  $O(\log n)$  的时间复杂度，总结而言有如下结论：

**定理 2.4** 一棵有  $n$  个内部结点的红黑树插入一个结点的时间复杂度为  $O(\log n)$ 。

当然，上面的描述都是原理层面的讲解，如果读者希望学习具体代码实现加深印象，可以参考《算法导论》的伪代码。

**【讨论 2】** 考虑从空树开始连续插入  $n(n > 1)$  个结点得到一棵红黑树（每一步插入都要保证红黑树性质），试问这棵树一定会有红色结点吗？若是，请给出清晰的证明；若不是，请举出反例。

答案是一定会，可以使用数学归纳法证明： $n = 2$  时显然正确，根下面插入的结点一定是红色且无需调整；此后不需要调整，因为插入的是红色结点，因此红色结点只可能变多；如果需要调整，则根据三种情况的讨论，无论哪一种情况，在调整之后一定还保留着红色结点。有同学可能会质疑，情况 1 如果  $G$  到了根结点，则需要被染黑，但要注意的是，此时的  $X$  还是红色的，因此不管什么情况都是会保留红色结点。

### 2.1.3 红黑树的删除

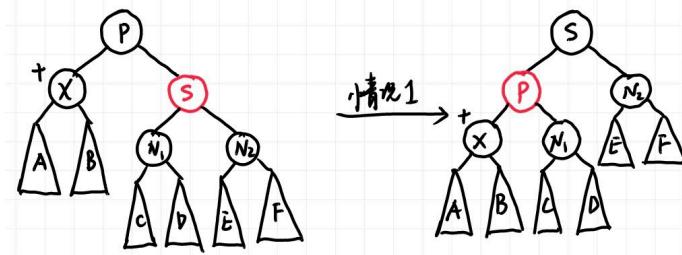
红黑树的删除是一个并不简单的操作，这其中有很多可能的情况需要不同的调整策略，接下来尽可能地清晰列举。在讲解红黑树删除之前，首先回顾一下普通平衡搜索树的删除操作，假设被删除的结点为  $X$ （如果不熟悉了可以回顾去年的 PPT 的例子）：

1. 如果  $X$  没有孩子，直接删除就好，没有任何后顾之忧；
2. 如果  $X$  只有一个孩子，那就让孩子接替  $X$  的位置；
3. 如果  $X$  有两个孩子，那就让  $X$  与其左子树的最大结点（或右子树最小结点）交换，然后删除  $X$ （这时  $X$  所在的位置一定最多只有一个子节点，因为左子树最大结点不可能有右孩子，右子树最小结点不可能有左孩子）。

事实上红黑树的删除是基于这些操作的，需要注意的是第三种情况， $X$  和与其交换的结点只交换键值，不交换颜色，否则如果两者颜色不同，在交换的时候就可能破坏第五条性质，这是很难令人满意的。总而言之，第三种情况可以通过一步交换直接转化为第一或第二种情况，因此只需要关心第一和第二种情况。在第一种情况中，接替被删除结点所在位置的结点是 NIL，第二种则是被删除结点的子结点。如果被删除的结点是红色，事实上无事发生，没有任何性质被破坏；如果被删除的是黑色，如果接替上来的结点是红色的，直接染黑也不会破坏任何性质。接下来就是问题的关键，如果接替的是 NIL 或是黑色结点应该怎么办？

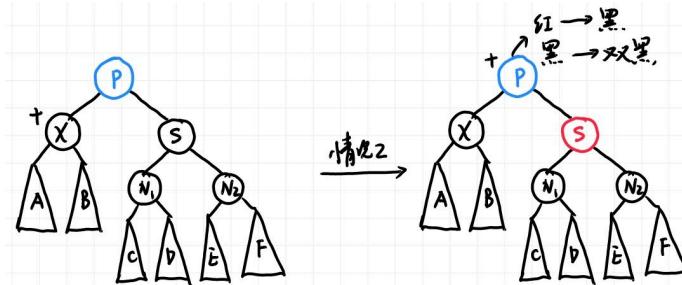
这个办法很聪明：直接给黑色结点或者 NIL（其实也是黑色结点）再加一重黑色，于是它的颜色变成了“双黑”。此时第五条性质没有被破坏，但是，第一条性质被破坏了！这里出现了非红也非黑的颜色！于是想法就是：把这个黑色的 debuff 扔给一个红色结点，或者一步一步往上扔给根结点，事实上根结点把双黑直接变成黑色完全不影响其它性质。可以将删除的情况分成以下四类（与插入相同，这里用子树表示更一般的情况， $X$  在此处则表示双黑结点，图中用黑色圆圈和圆旁边的加号表示双黑，蓝色表示颜色无所谓，可红可黑。注意这里  $X$  都是父亲的左孩子，右孩子情况对称）：

情况 1:  $X$  的兄弟是红色的



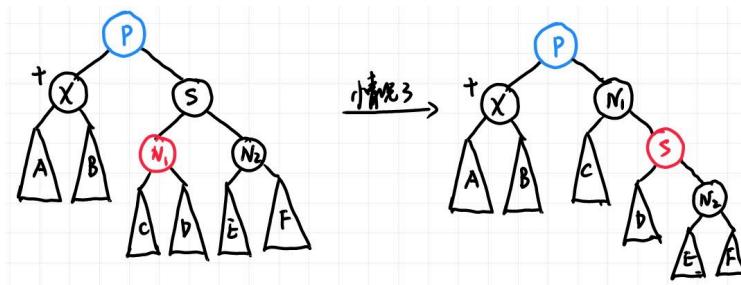
由于原先的树满足红黑树定义第四条，因此此时父结点一定是黑色。我们的想法很简单，兄弟是红色，那就希望兄弟能两肋插刀，把兄弟转上去，为了保持红黑树性质，很可惜只能把父亲染红，自己还承受双黑 debuff。但是好处在于，这个问题转化为了接下来的情况二三四中的一种，下面来看如何解决。

情况 2:  $X$  的兄弟是黑色的，且兄弟的两个孩子（根据距离划分为近、远侄子，用远近而不用左右是为了对称情况不混淆左右）都是黑色的



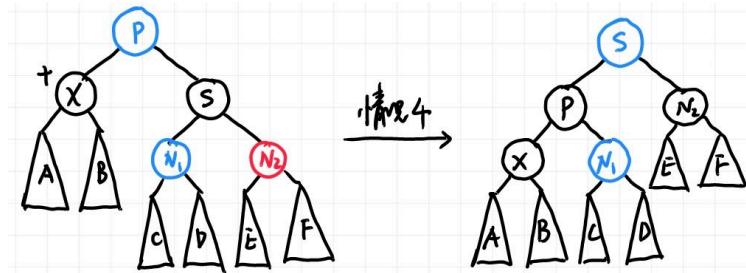
这时候没人是红色可以救了，那就想起根结点再怎么样都能救，所以把双黑往上推给父亲，为了保证红黑树性质，兄弟也要从黑变红，总而言之就是  $X$  这一层的黑色全部往上推。如果父亲原本是红色，那就染黑，问题解决；如果父亲原本是黑色，那父亲就变成双黑，让问题向根结点靠近。需要特别注意的是，如果是从情况 1 变成情况 2 的，父亲一定是红色，所以如果是 1 变为 2，则问题会马上解决。

情况 3:  $X$  的兄弟是黑色的，且近侄子是红色的，远侄子是黑色的

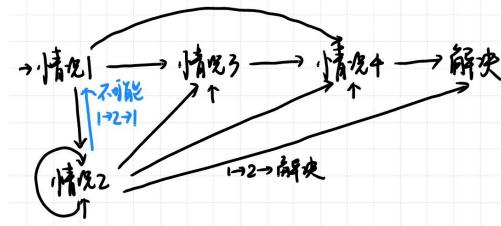


这时借用 AVL 树的想法，红色在父亲  $P$  的 RL 位置，因此做 double rotation: single rotation 后会变成情况 4 的 RR 的情况（也就意味着红色要给到 RR 的位置，这里有一个颜色的变化，用 RR 记忆很方便）。

情况 4:  $X$  的兄弟是黑色的，且远侄子是红色的，近侄子颜色任意



此时对应 AVL 树的 RR，于是再一次 single rotation 即可把双黑的一重黑丢给红色远侄子（即  $X$  和  $N_2$  都变成黑色），但要注意为了保证红黑树性质的颜色变化，即  $P$  和  $S$  还要交换颜色，此时问题解决。



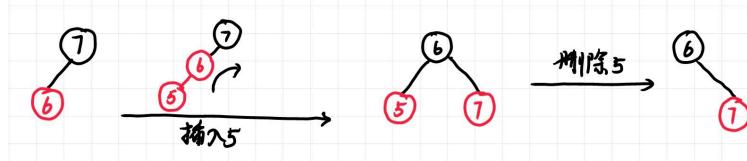
此时可以总结并计算出删除操作的时间复杂度。首先最多用  $O(\log n)$  的时间找到删除结点，最多 1 次交换和 1 个删除的操作。接下来如果删除后没有问题则到此结束；否则根据分析，情况 1、3 和 4 在问题解决前最多进去一次，因为 4 可以直接解决，3 直接进入 4 然后解决，1 如果进入 3 和 4 也可以马上解决，进入 2 后也因为父结点是红色可以马上解决。因此关键在于情况 2 可能出现很多次，但最多也只是树高  $O(\log n)$  次，因为每次都会上推 1 格。总而言之，因为情况 1、3 和 4 在问题解决前最多进去一次，所以最多 3 次旋转加上  $O(\log n)$  次颜色调整可以解决问题，因此有如下结论：

**定理 2.5** 一棵有  $n$  个内部结点的红黑树删除一个结点的时间复杂度为  $O(\log n)$ 。

总而言之，这里的情况的确非常复杂，但相信上面的解释会给出一些直观。考试的时候删除不会太难，很多时候依靠保证搜索树性质（左小右大）和红黑性质这两点就能猜出来操作是什么，当然能准确记住是更好。如果希望学习具体代码实现，同样可以参考《算法导论》的伪代码，尽管有些难看懂，但可以了解一下如何在不扩充红黑结点颜色的可取值范围的前提下表示双黑这种颜色。除此之外，这里的讲法与 PPT 不完全一致，但本质上是一样的，实际上不同老师的讲法都可能略有区别，但各位可以过一遍 PPT 上的例子就会发现这些方法都是一样的。

**【讨论 3】** 考虑将一个结点  $X$  插入红黑树  $T_0$ , 得到红黑树  $T_1$ , 然后紧接着下一步操作又立刻将  $X$  从  $T_1$  删除得到  $T_2$ , 请问  $T_0$  和  $T_2$  是否一定一样? 若是, 请给出清晰的证明; 若不是, 请举出反例。

显然不是; 考虑下面这一非常简单的例子即可:



#### 2.1.4 再论 AVL 树和红黑树的区别

在开头已经提到, AVL 树的平衡条件太严苛, 因此更新树 (即插入和删除) 操作会更频繁, 所以希望有一个条件更松的平衡要求但也能保证树高被控制在  $O(\log n)$  的量级。除此之外, AVL 树和红黑树似乎都是通过旋转恢复平衡, 没有很大的差别。但其实有一个很有趣的现象, 又非常多的库函数在选择平衡搜索树实现功能的时候, 会更常用红黑树, 例如大家最熟悉的 C++ 的 `std::map`, 以及 Java 8 开始的 HashMap 和 Microsoft .NET 框架的部分代码, 甚至 Linux 内核中内存管理也使用了红黑树 (可以参考[这个 GitHub 上的 Linux 文档](#))。那这其中的原因可能是什么呢?

事实上这一问题应当是没有标准答案的, 毕竟是当年工程师的多方面考虑综合后的选择, 但可以通过这个问题看一看 AVL 树和红黑树的一些更细致的区别:

1. 我们都知道, AVL 树平衡条件更严格, 推导 AVL 树高的时候用到了斐波那契数列, 实际上, 可以验证的是 AVL 树最差高度大约为  $1.44 \log n$ , 红黑树最差则可以达到  $2 \log n$ , 事实上讨论题 1 隐含了这一点, 从这一层面来看, 如果对一棵树的查询操作居多, 那么 AVL 树会是更好的选择;
2. 但上一讲提到, AVL 树虽然插入只需要常数次旋转即可, 但在删除时可能需要  $O(\log n)$  次旋转, 而红黑树插入和删除都是常数次, 有人提到在代码实现时旋转是插入和删除最耗时的操作, 因此如果插入删除操作多, AVL 树不如红黑树快速, 而我们知道使用 `std::map` 时的确可能遇到较多插入删除操作;
3. AVL 树需要维护树高或者 balance factor 属性, 这是一个整数的大小, 而红黑树只需要 1 个 bit 存储颜色即可, 因此更省空间;
4. 红黑树是可持久化的数据结构, 因此在函数式编程中容易实现; 并且红黑树也可以支持分裂、合并等操作, 这使得它可以做批量并行的插入、删除操作 (实际上这与讲义最后红黑树与 B 树的关联是相关的), 具体已经超出课程范畴, 不再详细讨论。

上面提到的很多内容都可以在[维基百科](#)或者[StackOverflow 的讨论](#)上看到更详细的说明。事实上还有人将这一问题归结于《算法导论》这一著作没有写 AVL 树因此很多人不了解, 因为有些测试表明 AVL

树的效率更高。但这些测试很难有完美的说服力，实际上上面的很多理由也并不完全有说服力，或许这就是工程上的选择的特点吧，并非一两句话可以严谨解释的。

## 2.2 B+ 树

B+ 树的定义非常简单，这里不再重复，只需要记住孩子个数和结点中存储的键值个数的限制（为什么会有  $\lceil M/2 \rceil$  呢？是因为插入到爆炸的时候就是分裂到这个数量，为什么根结点最少是 2 个子结点呢？因为最开始插入到根结点爆炸的时候只能分裂成两个孩子），以及非叶结点中存储的键值的含义（实际上就是在向下搜索时区分小于键值和大于等于键值从而决定应该去第几个孩子结点）即可，然后需要理解 2-3-4 树或者 2-3 树这种简称的含义，实际上就是每个结点可能含有的子结点个数（内部可能的键值数则是可能的子结点数减 1）。需要注意的是，这里主要介绍 B+ 树的操作和意义。

### 2.2.1 B+ 树的操作

B+ 树的操作相比于红黑树直观很多，因为无需复杂的染色和旋转，也没有很多复杂的分类讨论。

1. 搜索：根据 B+ 树定义，需要在非叶结点层逐层和存储的键值比较从而确定去哪一个孩子结点。因此时间复杂度有两个重要因素：一个是树的高度，另一个是每一层搜索需要的时间。树的高度非常好计算，最差的情况也是每个结点都存  $\lceil M/2 \rceil$  个结点，因此最大高度是  $O(\log_{\lceil M/2 \rceil} N)$  的。然后每一层因为键值是排好序的，因此用二分查找找到要去哪个孩子结点，复杂度为  $O(\log_2 M)$ ，综合可得搜索的时间复杂度为

$$O(\log_2 M \cdot \log_{\lceil M/2 \rceil} N) = O\left((\log_2 M/2 + 1) \cdot \frac{\log_2 N}{\log_2 M/2}\right) = O(\log N),$$

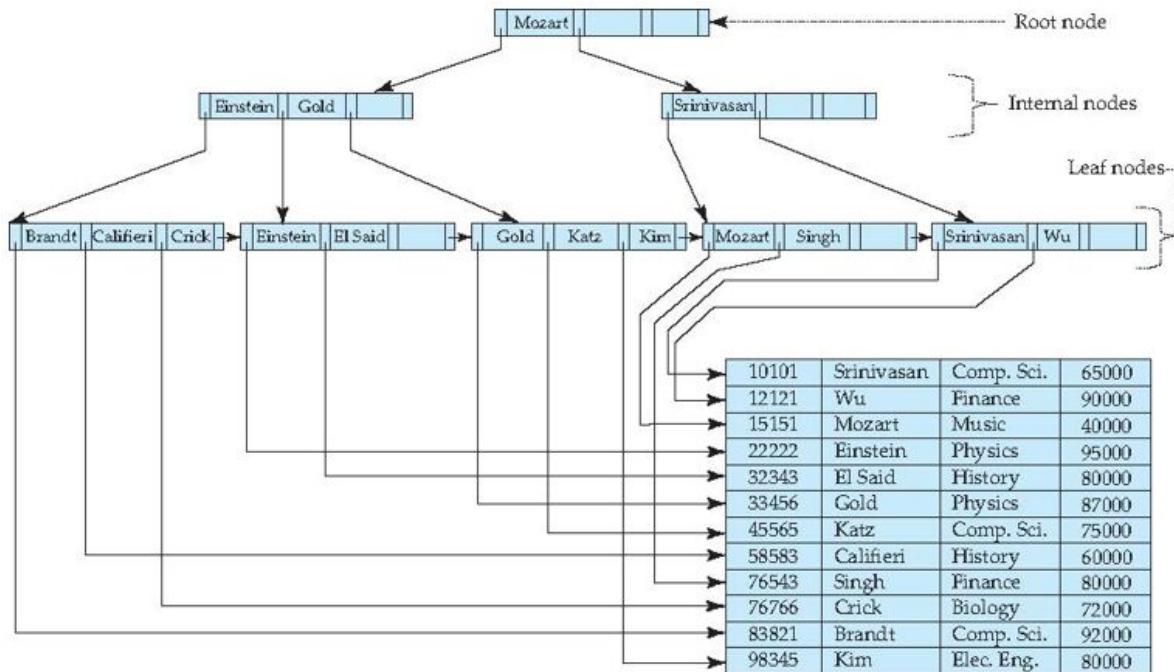
注意推导中使用了换底公式。

2. 插入：PPT 上的伪代码已经十分清楚，就是找到插入的位置，然后插入看结点是否放得下，放不下就分裂，如果分裂后子结点个数也过多则继续向上一层分裂，直到根结点孩子爆满则将根结点分裂并生成新的根结点，当然还要注意即使不分裂也可能需要按 B+ 树定义更新上层结点。我们知道树有  $O(\log_{\lceil M/2 \rceil} N)$  层，每层操作最多是  $O(M)$  的（如更新结点或者分裂，无非就是更改  $O(M)$  个键值以及修改  $O(M)$  个父子指针），因此整体时间复杂度为  $O(M \cdot \log_{\lceil M/2 \rceil} N) = O\left(\frac{M}{\log M} \log N\right)$ 。
3. 删除：PPT 没有要求，但想法很简单，因为只需把插入时分裂结点改为合并键值或孩子数量少的结点，当然需要注意的是，为了确保合并后键值数量不会超过  $M$  且减少合并次数，可以先看看兄弟结点是不是键值还很多，多的话拿一个过来即可，事实上整体时间复杂度和插入分析类似，也为  $O\left(\frac{M}{\log M} \log N\right)$ 。

**【讨论 4】** 总结 AVL 树、红黑树和 B+ 树的搜索、插入和删除的时间复杂度，绘制成表格。

## 2.2.2 B+ 树的意义

B+ 树与之前学过的搜索树的目的不完全一致，事实上在数据库系统课程中会学到 B+ 树在数据库索引中的应用。的确，B+ 树（或者 B 树族）自诞生之日起就注定是为数据库系统（或者文件系统）服务的。对于在内存中暂时存储的数据，使用一般的平衡二叉搜索树即可达到目的。根据前面的分析，B+ 树插入和查询的时间复杂度前面会带  $M / \log M$  的倍数， $M$  较大时这是一个比较大的常数，因此比二叉树耗时多，并且逐层分裂更新结点也是非常复杂的操作，所以在数据量很小的时候，B+ 树看起来并不是一个很好的选择。然而，当数据量非常大，不再存储在主存而是存储在机械硬盘时，情况恰好相反。假设有 1000000 条数据存在硬盘中，这时需要存储在硬盘中的搜索树（因为显然这棵树也太大了，而且这棵树也需要作为硬盘索引永久存储）。如果使用二叉树，则大约有  $\log_2 1000000 = 20$  层，因此需要经过约 20 个结点才能逐步下推到叶子结点找到真正的数据。然而，记住这棵树在磁盘里，因此每次从一个结点下降到其孩子时，磁盘都需要重新寻道、旋转，取出下孩子结点放在内存才能进行运算比较决定下一个结点，这里的时间是非常长的，可以达到毫秒级别，比内存中需要的时间要高出 5 个甚至更多数量级。所以现在时间瓶颈不在每一层的比较、重建的时间，因为这是在内存中的操作，相比于磁盘操作完全可以忽略，现在的瓶颈在在磁盘上重新定位孩子结点，因此在这种情况下，最好的选择应该是让树的高度越小越好，比如取  $M = 100$ （真实的情况一般在 50-2000 之间），则大约需要  $\log_{100} 1000000 = 3$  次磁盘操作就可以了，因此比二叉树效率高很多。



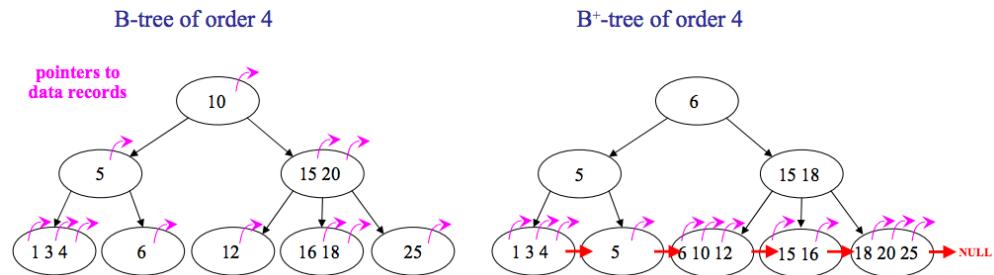
一个真实的数据库索引 B+ 树如上图所示，事实上从图中可以明确看出，本课程学习的 B+ 树并非真正应用的 B+ 树，一方面真正的 B+ 树叶子结点并非存的是真实的数据，而是数据表的主键（索引值，也就是寻找这条记录的依据）和指向主键对应的记录的指针，并非把真正的数据记录存储在叶子结点；

另一方面，真正的 B+ 树的叶子结点是互相连接的，这原因可以回顾删除操作：需要看兄弟结点是不是有盈余，所以叶子之间的指针可以方便直接找到兄弟结点。

### 2.2.3 B 树族

事实上，B+ 树这个名字就会让我们想到是不是有 B 树，的确如此，B+ 树是 B 树改进后的版本。如下图所示，B 树叶子结点之间没有指针，B+ 树对这一点做了改进；另一方面 B 树的非叶结点也有指向真实数据记录的指针，而中间结点有的，在叶结点就不需要再出现了，因为搜索的时候搜到非叶结点中存了主键即可直接通过指针找到数据记录，无需像 B+ 树那样搜索到底。但是为什么 B+ 树放弃了部分记录的搜索便利呢？因为在非叶结点还要存储指向数据记录的指针是很耗空间的事情，假设每个结点都是一个 page 的大小，磁盘每次读写也是 1 个 page 为单位，那么这一个 page 里面如果还要存储指向数据记录的指针就会大大减少指向孩子的指针个数，显然在分叉数非常大的时候孩子中键值的数目远远多于这一层存的键值，因此这一个 page 最终能索引到的数据记录会少很多，并且回忆磁盘为了取到这一个 page 要花很多的时间，因此在非叶结点存储指向数据记录的指针是因小失大。

- A B<sup>+</sup>-tree can be viewed as a B-tree in which each node contains only keys (not pairs), and to which an additional level is added at the bottom with linked leaves



B\* 树则是另一种 B 树族中的树，这种树要求每个结点中键值数目至少为  $2/3M$ ，比 B+ 树更满，因此直观理解插入时应当是相邻两个都插满了后分裂成三个。如何做到保证同时插满两个结点呢？这就需要在插满一个结点后看一看旁边的兄弟有没有满，如果没满直接给兄弟即可，满了就可以分裂。当然还有所谓的 B\*+ 树，感兴趣的同学可以自行搜索，这里不再展开。

当然，一个有趣的问题是为什将这类树命名为 B 树族，发明者并未给出明确解释，它可能是 Boeing (发明者所在实验室名) , balanced, between, broad, bushy 中的任何可能含义，事实上发明者之一 Edward M. McCreight 曾说 “the more you think about what the B in B-trees means, the better you understand B-trees”，因此这其中的奥秘还需同学们自行理解体会。

#### 2.2.4 红黑树与 B 树的关联

我想很多同学都会怀疑，红黑树这种复杂的结构是怎么就想到的呢？一个有趣的事情是，红黑树的提出时间是 1978 年，比 B 树的时间 1972 年晚了六年，并且其中都有 Rudolf Bayer 参与设计。事实上，红黑树的思想正是源于想法更加直观的 B 树。实际上，2-3-4 树和 B 树的操作多有相似之处，由于文档可视化比较麻烦，读者可以直接参考[这个网页](#)或者[这个网页](#)，实际上非常直观（但请时刻记住 B 和 B+ 树是不一样的，因此很多操作也不完全一致，不要弄混了，考试都是 B+ 树）。当然这也给我们一个启示，即基于分叉更多的 B 树可以设计彩色（多于两种颜色）树。当然有趣的是据作者所说，他们决定设计红黑树是因为红色是作者在 Xerox PARC 工作期间可用的彩色激光打印机打印的最好看的颜色，另一种说法是使用红色和黑色是因为他们可以用红笔和黑笔来画树。

## Lecture 3: 倒排索引

编写人: 吴一航 yhwu\_is@zju.edu.cn

### 3.1 内容概述

本讲是这门课程中最轻松的一讲。本讲的目标是应用之前学习的数据结构，讨论关于搜索引擎的设计问题。本讲主要涉及了如下主题：

1. 倒排索引的引入和定义：从两个 naive 的想法出发，其一是遍历搜索，这样太耗时间；其二是稀疏矩阵，这样存储比较浪费空间。所以改进为链表存储，这就是倒排索引。注意在倒排索引里保存单词的出现次数是因为当多个单词同时搜索时，从出现最少的词入手搜索会更快。
2. 如何构建倒排索引：逐个词语读入插入构建。其中会有很多问题，例如分词，stemming, stop words 等，还有通过搜索树或 hash 访问等；除此之外还有存储上的考量，因为内存不够需要存储到外存，外存可以分布式存储（两种方式），然后还有更新时可以用 cache 等改进存储效率。
3. 搜索引擎的评价：区分 Data Retrieval 和 Information Retrieval，了解准确率和召回率两个重要的衡量参数（与此对应还有假阳性和假阴性），其中阈值的设置是重要的影响因素。

希望进一步了解的同学可以参考 InvertedFileIndex.zip，其中有相关论文和参考资料等。因为本讲内容简单且宽泛，因此细节不在此赘述，对 PPT 中写得不够完善的部分也可以参考陈越老师的 MOOC。事实上搜索引擎设计中还有非常多具有影响力的算法，如 HITS, PageRank 等，感兴趣的读者可以自行了解。

## Lecture 4: 左式堆与斜堆

编写人: 吴一航 [yhwu\\_is@zju.edu.cn](mailto:yhwu_is@zju.edu.cn)

### 4.1 左式堆

#### 4.1.1 可合并堆的引入

在数据结构基础课程中，我们已经学过最简单的堆（优先队列）结构，该结构的目标在于能够快速访问全局最优的值（优先值），从而有利于一些需要不断选取最优值的场景，例如操作系统中的任务调度，或者在下一讲将会介绍的 Dijkstra 算法加速（以及最小生成树的 Prim 算法）等。作为一个数据结构，堆自然有如下两种性质需要规定：

1. 结构性质：堆是一棵完全二叉树。并且由于是完全二叉树，故树高  $h$  和总结点数之间的关系显然为  $h = O(\log n)$ 。
2. 序性质：对于最大堆，根结点就是最大元素，然后每个结点的孩子必须小于等于自身；反之对于最小堆，根结点是最小元素，孩子应当大于等于父亲。

之前的平衡搜索树也是由这两者规定的，序性质大部分都是左小右大，B+ 树略复杂，结构性质则是各显神通，读者可以自行回顾总结。

此前已经学习了基于数组的二叉堆实现，事实上是利用数组构造了一棵完全二叉树，通过数组索引除以 2 找到结点的父亲，乘以 2 和乘以 2 加 1 获得结点的左右孩子，因此非常方便，相比于指针需要寻址，这样的实现显然更加高效。下面简要回顾一下二叉堆的一些基本操作（以最小堆为例，本节无特殊说明都默认最小堆）：

1. Insert：直接插在完全二叉树的下一个空位上，然后 percolate up 找到它应当在的位置，显然最坏情况也与完全二叉树的高度成正比，即  $O(\log n)$ 。
2. FindMin：直接返回根结点即可，时间  $O(1)$ 。
3. DeleteMin：直接用完全二叉树的最后一个元素顶替根结点，然后 percolate down 找到新根结点的归宿，时间  $O(\log n)$ 。
4. BuildHeap：即对  $n$  个元素建堆存储。这是一个比较特别的操作，最原始的方法就是连续插入  $n$  次，但这样时间复杂度为  $O(n \log n)$ ，所以要有更好的手段。更好的方法是：无需管序性质，直接

任意插入这  $n$  个值，然后从完全二叉树倒数第二排有孩子的结点开始，往前依次检查是否有违反序性质的，有就 percolate down 到正确的位置，循环直到根结点也调整完毕为止，可以验证这样的算法复杂度为  $O(n)$ ，具体可见数据结构基础的教材与 PPT。

除此之外还有一些操作，这些操作在 Dijkstra 算法加速等场景中可能有应用，因此也展开介绍：

1. DecreaseKey / IncreaseKey：非常简单，直接用 percolate up / down 实现即可。
2. Delete：用 DecreaseKey 把 key 降低到最低，percolate up 到根结点后调用 DeleteMin 即可。

需要注意的是，这里的操作要求维护需要进行操作的结点（记为  $P$ ）的位置，否则需要  $O(n)$  的时间找到  $P$  然后再操作（因为堆不支持平衡搜索树那样左小右大的搜索便利性），显然是不合适的。所以这时候用指针实现是一种可行的办法，这样即使结点在一棵虚拟的树上的位置变了，但它在内存里的位置是不变的，这时以上操作的时间复杂度均为  $O(\log n)$ 。

看上去上面的操作需要的时间都十分完美，那么还需要在什么方面有改进呢？答案是有时候会考虑两个堆合并（merge）的问题。如果要对二叉堆进行合并，由于仍然需要保证合并后是完全二叉树，所以在基于数组的实现中能想到的最好的方法也只能是直接合并两个数组，然后调用 BuildHeap 在  $O(n)$  时间内完成。然而我们希望合并也是一个  $O(\log n)$  内可以实现的操作，所以需要定义新的结构（也就是接下来要介绍的三种可合并堆）来实现这一点。

#### 4.1.2 左式堆的定义与性质

下面首先介绍左式堆（leftist heap），这一数据结构仍然保持堆的序性质，但不再要求完全二叉树的结构性质。和它的名字一样，左式堆应当是整体向左倾斜的，那么如何严格定义这样的直观呢？这样的直观又有什么好处呢？下面来逐一介绍。首先需要定义一个 null path length 的概念以便于之后进一步定义左式堆的结构：

**定义 4.1** 把任一结点  $X$  的零路径长（null path length, NPL） $Npl(X)$  定义为从  $X$  到一个没有两个儿子的结点的最短路径的长。因此，具有 0 个或 1 个儿子的结点的  $Npl$  为 0，且规定  $Npl(null) = -1$ 。

根据这一定义可知，对于一个堆（或者二叉树），计算每个结点的  $Npl$  应当从叶子结点出发向上计算，因为每个结点的  $Npl$  就等于它的两个孩子的  $Npl$  的最小值 + 1。这一结论适用于没有两个孩子的结点，因为定义中  $Npl(null) = -1$ 。

**定义 4.2** 左式堆的结构性质是：每个结点的左孩子的  $Npl$  都要大于等于其右孩子的  $Npl$ 。注意这一定义适用于没有两个孩子的结点，因为有定义  $Npl(null) = -1$ 。

左式堆的例子和非例子可以参考 PPT。总结而言，我们基于  $Npl$  定义出了左式堆的结构性质，将向左倾斜的直观有了严格的表达。那么下一步需要研究的问题就是，向左倾斜有什么好处？

**定理 4.3** 在右路径上有  $r$  个结点的左式堆必然至少有  $2^r - 1$  个结点（右路径指从根结点出发一路找右孩子直到找到叶子的路径）。

**讨论 1:** 证明这一定理（提示：使用数学归纳法）。

**证明：** 使用数学归纳法证明。若  $r = 1$ ，则显然至少存在 1 个结点。设定理对右路径上有小于等于  $r$  个结点的情况都成立，现在考虑在右路径上有  $r + 1$  个结点的左式堆。此时，根的右子树恰好在右路径上有  $r$  个结点，因此右子树大小至少为  $2^r - 1$ 。考虑左子树，根据左式堆定义左子树的  $N_{pl}$  必须大于等于  $r - 1$ ，事实上  $N_{pl}$  大于等于  $r - 1$  的树右路径至少有  $r$  个结点，因此左子树大小也至少为  $2^r - 1$ ，因此整棵树的结点数至少为  $1 + (2^r - 1) + (2^r - 1) = 2^{r+1} - 1$ 。 ■

从这个定理立刻得到，共有  $N$  个结点的左式堆的右路径最多含有  $\lfloor \log(N + 1) \rfloor$  个结点。因此我们的想法是，将左式堆的操作的所有的工作都放到右路径上进行，因为它是  $O(\log n)$  的。当然很显然的一点是，对右路径的 Insert 和 Merge 可能会破坏左式堆性质，但接下来的分析将表明，恢复左式堆的性质是非常容易的。

有一个直觉是值得特别说明的，就是平衡搜索树中通常要求树越平衡越好，但堆却似乎不需要这一点，这是为什么呢。这是因为堆不支持 find 操作，所以左式堆左边的结点在操作中可以完全不被访问，而接下来的讨论会表明只在右路径上操作也完全可以解决插入、删除最小值和合并，因此完全不需要保持树的平衡。

### 4.1.3 左式堆的操作与实现

接下来的讨论主要解决三个核心操作：Insert、DeleteMin 和 Merge，其它操作和二叉堆区别不大或者没有必要。PPT 的 5-7 页详细展示了操作的动态过程以及代码实现，可以参考，特别是代码实现很可能作为考试中的程序填空题出现，请务必多加注意。

接下来简要地说明三种操作：

1. Merge：最核心的操作，事实上是接下来两种操作的基础。
2. Insert：可以视为一个堆和一个单结点的堆的 Merge，因此问题转化为 Merge。
3. DeleteMin：两个步骤实现：首先删除根结点，然后只需要将根结点的两个子树 Merge 即可，因此关键问题还是 Merge。

所以这三种操作最终都归结于 Merge，时间复杂度也完全来源于 Merge，所以接下来只对 Merge 进行详细的分析。下面介绍两个版本的解决方案，其一是递归，其二是迭代，事实上它们从结果上来看是等价的，我们分别来看它们在分析中的优劣。

【注：Delete 和 DecreaseKey 在另一份附件中有介绍，感兴趣的同學可以参考。】

#### 4.1.3.1 递归实现

递归实现分为如下步骤：

1. 如果两个堆中至少有一个是空的，那么直接返回另一个即可；
2. 如果两个堆都非空，比较两个堆的根结点 key 的大小，key 小的是  $H_1$ ，key 大的是  $H_2$ ；
3. 如果  $H_1$  只有一个顶点（根据左式堆的定义，只要它没有左孩子就一定是单点），直接把  $H_2$  放在  $H_1$  的左子树就完成任务了（很容易验证这样得到的结构符合左式堆性质，此时 Npl 也没有变化）；
4. 如果  $H_1$  不只有一个顶点，则将  $H_1$  的右子树和  $H_2$  合并（这是递归的体现，在 base case 设计良好，其它步骤也都合理的情况下你完全可以相信这一步递归帮你做对了），成为  $H_1$  的新右子树；
5. 如果  $H_1$  的 Npl 性质被违反，则交换它的两个子树；
6. 更新  $H_1$  的 Npl，结束任务。

如上的步骤对应于 PPT 的伪代码，直观的解释在伪代码的前一页也有展现，核心是上面的 4-5 步。需要注意的是，PPT 上的伪代码比较特别，它把整个过程拆成了两个互相调用的递归函数，实际上直接把第二个递归函数合并进第一个函数的代码中就会发现这就是一个普普通通的递归过程，这里只是更进一步地模块化，实际上是和普通递归完全等价的。

还有两个问题值得讨论，其一是一定要注意更新 Npl，否则所有的结点 Npl 都将是初始值 0，因此整个堆无论长什么样都是左式的，这显然不合理（得到的堆序性质仍满足但结构性质不满足）。其二是这一算法的复杂度如何，事实上可以走一遍这个递归流程，例子就是 PPT 第 5 页的例子：

1. 首先比较发现  $3 < 6$ ，因此递归要求  $H_1$  的右子树  $H'_1$  与  $H_2$  合并， $H_1$  的左子树不动；
2.  $H'_1$  的根结点  $8 > 6$ ，因此递归要求  $H_2$  的右子树  $H'_2$  与  $H'_1$  合并， $H_2$  的左子树不动；
3.  $H'_1$  根结点 8 大于  $H'_2$  根结点 7，因此递归要求  $H'_2$  的右子树  $H''_2$  与  $H'_1$  合并， $H'_2$  的左子树不动；
4.  $H'_1$  根结点 8 小于  $H''_2$  根结点 18，因此递归要求  $H'_1$  的右子树  $H''_1$  与  $H''_2$  合并， $H'_1$  的左子树不动；
5. 发现  $H''_1$  是 null，因此直接连接上  $H''_2$  即可，递归开始返回；
6. 1-4 步的每一步都在实现递归中的 `Merge(H1->Right, H2)` 步骤，因此接下来只需要将 merge 后的树接到父亲的右子树上，实现 `H1->Right = Merge(H1->Right, H2)`，然后判断是否需要交换子树并且更新 Npl 即可继续返回上一层递归，知道所有递归函数都返回就实现了整个过程。

在将递归过程展开之后，就可以很清楚地分析时间复杂度。首先分析递归的最大深度。不难发现，在 1-5 步的递归过程中，产生的递归层数不会超过两个左式堆的右路径长度之和，因为每次递归都会使得

两个堆的其中一个（根结点 key 更小的）向着右路径上下一个右孩子推进，并且直到其中一个推到了 null 结点就不再加深递归。注意加深一层的过程中的操作是常数的，因为只需要简单的大小比较和找孩子，加上右路径长度的限制，因此递归向下的过程是  $O(\log n)$  的。这一点可以更严谨地展开：假设  $H_1$  大小为  $N_1$ ,  $H_2$  大小为  $N_2$ , 两者路径之和

$$O(\log N_1 + \log N_2) = O(\log N_1 N_2) = O(\log \sqrt{N_1 N_2}) = O(\log(N_1 + N_2)),$$

上面的推导用到了基本不等式  $a + b \geq 2\sqrt{ab}$ 。总而言之，两个堆右路径长度之和仍然是两个堆大小的对数级别，因此递归层数是  $O(\log n)$  的是准确的。

接下来分析递归返回的操作，事实上每一层的操作也是常数的，因为只需要接上新的指针，判断、交换子树以及更新  $N_{pl}$ ，所以也是  $O(\log n)$  的，因此总的时间复杂度就是  $O(\log n)$  的。

#### 4.1.3.2 迭代实现

有趣的一点是，上面递归过程的展开实际上就等价于迭代算法的流程：每一次递归向下对应迭代中保留根结点更小的堆的左子树（就像是左子树不动，右子树等着接下来合并的结果），直到最后一次与 null 合并直接接上，递归返回过程实际上就是逐个检查新的右路径上的结点是否有违反  $N_{pl}$  性质的并且更新  $N_{pl}$  即可，其它结点无需关心是因为它们根本就不受影响（可以看 PPT 第 7 页的例子辅助理解这一过程）。因为已经说明了迭代和递归的每一步都是有对应关系的，只不过递归是最后返回时才接上每个结点的右子树，迭代过程中就已经接好了，因此二者时间复杂度是一样的。

当然在完成上面的流程后会有一个观察，就是在递归向下之后，或者说交换孩子调整左式堆性质之前，合并得到的堆的右路径是原来两个堆的右路径合并排序的结果，例如 PPT 上的 3、8 和 6、7、18 合并为了 3、6、7、8、19。通过上面的过程很容易证明这一结论是通用的，因为每次都在比较两个堆的右路径上两个点的大小，然后把小的作为根插入。有了这一规律，做题会更快捷一些，因为你只需要把两条右路径从小到大排序，然后从小到大依次带着左子树接入到新的右路径即可（但要注意在此之后还需要调整使得满足左式堆结构性质）。因此用代码实现迭代版本也并没有想象中复杂，只需要对两个堆右路径从小到大便利操作，然后再从右路径最后一个点返回根结点，过程中检查结构性质并更新  $N_{pl}$  即可。

## 4.2 斜堆

### 4.2.1 简介

斜堆与左式堆的关系就像是 splay 树和 AVL 树之间的关系。回顾 splay 树，它并不需要维护 AVL 树中的  $bf$  属性，只需要在访问一个结点之后就无脑地将它用 zig / zig-zig / zig-zag 三种情况将它翻到根结点即可。

斜堆也是类似的想法，它不用再维护  $N_{pl}$ ，因此在递归过程中左式堆所有维护结构性质以及更新  $N_{pl}$  的

操作不再需要，取而代之的是如下操作：

1. 在 base case 是处理  $H$  与 null 连接的情况时，左式堆直接返回  $H$  即可，但斜堆必须看  $H$  的右路径，要求  $H$  右路径上除了最大结点之外都必须交换其左右孩子。
2. 在非 base case 时，若  $H_1$  的根结点小于  $H_2$ ，如果是左式堆，需要合并  $H_1$  的右子树和  $H_2$  作为  $H_1$  的新右子树，最后再判断这样是否违反性质决定是否交换左右孩子，斜堆直接无脑交换，也就是说每次这种情况都把  $H_1$  的左孩子换到右孩子的位置，然后把新合并的插入在  $H_1$  的左子树上。

可以看 PPT 第 8 页的例子，如果像前面分析左式堆那样展开递归的每一步，前面的过程很好理解，就是无脑交换根的 key 更小的堆的左右孩子，关键在于当递归到最深的一层是 merge 一个 null 堆和一个 18 为根、35 为 18 的左孩子的堆，看上面操作的第一条，这个堆的右路径上除了最大结点外都要交换左右孩子，但幸运的是，这个堆右路径只有 18 一个结点，它是最大的，所以无需交换。这也就是 PPT 上说从递归角度来看这里不用交换的本质原因，特别注意务必以这上面说的方法为准，在维基百科等地方的斜堆 base case 之后都无需操作，但这里可能还有操作（尽管这个例子没有，但作业题有），作业和考试务必按照这里以及 PPT 所说的操作为准！

所以类似于左式堆，最后合并出的堆的左路径上包含两个原始堆的右路径排序后的结果，当然后面还可能连着原始堆右路径最大值的一些左孩子（因为这些左孩子是不被交换的），因此做题的时候这一规律也是可以利用的，熟练之后不一定要按照前面递归的方式逐步分析。

**讨论 2：** 判断以下两个说法是否正确，若正确，请给出详细的证明；若不正确，请举出反例：

1. 按顺序将含有键值  $1, 2, \dots, 2^k - 1$  的结点从小到大依次插入左式堆，那么结果将形成一棵完全平衡的二叉树。
2. 按顺序将含有键值  $1, 2, \dots, 2^k - 1$  的结点从小到大依次插入斜堆，那么结果将形成一棵完全平衡的二叉树。

本题两个结果都是正确的，证明只需要找规律归纳即可，没有什么特殊性。除此之外，斜堆的 Delete 和 DecreaseKey 操作一般不讨论。

### 4.2.2 摊还分析

根据上面的介绍，我们发现斜堆的好处很多，例如不需要多余空间存储 Npl，也不需要有很多判断、更新操作，但它真的能像我们希望的那样，像 splay 树一样具有  $O(\log n)$  的摊还操作代价吗？接下来开始分析。

首先需要明确的一点是，insert 和 delete 还是以 merge 为核心，所以一系列 insert、delete、merge 操作分析的关键还是 merge，因此只需分析 merge。事实上，因为斜堆的定义没有限制住右路径的长度，因此无法像左式堆那样直接用右路径长度 bound 住它的时间复杂度。

下面使用最强大的摊还分析工具：势函数法。为了定义这一势函数，需要给出一个辅助的定义：

**定义 4.4** 称一个结点  $P$  是重的 (*heavy*)，如果它的右子树结点个数至少是  $P$  的所有后代的一半（后代包括  $P$  自身）。反之称为轻结点 (*light node*)。

为了证明后面的主要定理，在此先给出一个引理

**引理 4.5** 对于右路径上有  $l$  个轻结点的斜堆，整个斜堆至少有  $2^l - 1$  个结点，这意味着一个  $n$  个结点的斜堆右路径上的轻结点个数为  $O(\log n)$ 。

**证明：**这里的证明和前面证明左式堆性质是十分类似的。对于  $l = 1$  显然成立，现在设小于等于  $l$  都成立，对于  $l+1$  的情况，找到右路径上第二个轻结点，那么它所在子树大小根据归纳假设至少有  $2^l - 1$  个结点。现在考虑第一个轻结点，根据轻结点定义它的左子树更大，而右路径上第二个轻结点所在的子树在其右子树中，因此它的左子树至少有  $2^l - 1$  个结点。故整个堆至少有  $1 + (2^l - 1) + (2^l - 1) = 2^{l+1} - 1$  个结点。 ■

**定理 4.6** 若有两个斜堆  $H_1$  和  $H_2$ ，它们分别有  $n_1$  和  $n_2$  个结点，则合并  $H_1$  和  $H_2$  的摊还时间复杂度为  $O(\log n)$ ，其中  $n = n_1 + n_2$ 。

**证明：**定义势函数  $\Phi(H_i)$  等于堆  $H_i$  的重结点 (heavy node) 的个数，并令  $H_3$  为合并后的新堆。设  $H_i(i = 1, 2)$  的右路径上的轻结点数量为  $l_i$ ，重结点数量为  $h_i$ ，因此真实的合并操作最坏的时间复杂度为  $c_i = l_1 + l_2 + h_1 + h_2$  (所有操作都在右路径上完成)。因此根据摊还分析可知摊还时间复杂度为

$$\hat{c}_i = c_i + \Phi(H_3) - (\Phi(H_1) + \Phi(H_2)).$$

事实上，在 merge 前可以记

$$\Phi(H_1) + \Phi(H_2) = h_1 + h_2 + h,$$

其中  $h$  表示不在右路径上的重结点个数。现在要考察合并后的情况，事实上有两个非常重要的观察：

1. 只有在  $H_1$  和  $H_2$  右路径上的结点才可能改变轻重状态，这是很显然的，因为其它结点合并前后子树是完全被复制的，所以不可能改变轻重状态；
2.  $H_1$  和  $H_2$  右路径上的重结点在合并后一定会变成轻结点，这是因为右路径上结点一定会交换左右子树，并且后续所有结点也都会继续插入在左子树上（这也表明轻结点不一定变为重结点）。

结合以上两点可知，合并后原本不在右路径上的  $h$  个重结点仍然是重结点，在右路径上的  $h_1 + h_2$  个重结点全部变成轻结点， $l_1 + l_2$  个轻结点不一定都变重，因此合并后有

$$\Phi(H_3) \leq l_1 + l_2 + h,$$

代入数据计算可得

$$\hat{c}_i \leq (l_1 + l_2 + h_1 + h_2) + (l_1 + l_2 + h) - (h_1 + h_2 + h) = 2(l_1 + l_2).$$

根据前面的引理， $l_1 + l_2 = O(\log n_1 + \log n_2) = O(\log(n_1 + n_2)) = O(\log n)$ （这里的等号之前有完全一样的说明），并且注意到初始（空堆）势函数一定为 0。且之后总是非负的，所以这一势函数定义满足要求，因此证明也就完成了。 ■

回忆摊还分析中强调的，势函数的想法就是使得复杂度大的操作恰好是能较多地降势能的，这里右路径很长的时候操作时间复杂度大，但很幸运的是，根据引理，右路径上的轻结点个数总是不可能太多，因此这时重结点更多，而操作之后右路径上重结点全部变轻，势函数又规定为重结点个数，因此非常完美地抵消了真实的复杂度。

需要说明的是，有一种很直观的势函数定义方法在此是行不通的。我们知道，一次合并的效果是处在右路径上的每一个结点都被移到左路径上，而其原左儿子变成新的右儿子。所以自然的一种想法是把每一个结点算为右结点或左结点来分类，这是根据结点是右儿子还是左儿子来决定的。将右结点的个数作为势函数，虽然这一势函数初始为 0 并且总是非负，但是问题在于，如果考虑一种最坏的情况，就是右路径长度为  $O(n)$  时，我们希望势函数减小很多使得摊还代价是对数的，然而很显然能找到例子使得在这种情况下操作之后右结点数量不变，所以这种情况下右结点并不适合于解决问题。因此考虑另一种基于轻重的分类来解决斜堆的摊还分析问题。

## Lecture 5: 二项堆

编写人: 吴一航 yhwu\_is@zju.edu.cn

## 5.1 二项堆

### 5.1.1 二项堆的引入与基本概念

二项堆的引入来源于我们希望插入建堆的操作有常数的平均时间，这一想法来源于二叉堆可以在  $O(n)$  时间内实现  $n$  个结点的插入建堆操作，而之前讨论的左式堆和斜堆不可以。因此我们希望在保持合并的对数时间的条件下优化插入的时间复杂度。

二项堆的定义如下：

1. 结构性质：

- (a) 二项堆不再是一棵树，而是多棵树构成的森林，其中每一棵树都称为二项树（后面能看到为什么是这个名字）；
- (b) 一个二项堆中的每棵二项树具有不同的高度（即每一个高度最多对应一棵二项树）；
- (c) 高度为 0 的二项树是一棵单节点树；高度为  $k$  的二项树  $B_k$  通过将一棵二项树  $B_{k-1}$  附接到另一棵二项树  $B_{k-1}$  的根上而构成，这一点根据 PPT 上的图非常直观。

2. 序性质：每棵二项树都保持堆的序性质，例如是最小堆（本节无特殊说明都默认最小堆）则根结点最小，孩子都比父亲大。

事实上进一步观察会发现更有趣的事情：二项树  $B_k$  实际上就是由一个带有儿子  $B_0, B_1, \dots, B_{k-1}$  的根组成。由此很容易根据定义归纳得到高度为  $k$  的二项树恰好有  $2^k$  个节点，而且在深度  $d$  处的节点数恰好就是二项系数  $\binom{k}{d}$ （下面给出简要证明），因此二项树的名字得来非常自然。

**证明：**数学归纳法。对于  $k = 0$  的情况显然正确，设直到  $k$  结论都是正确的，则对于  $B_{k+1}$ ，第一层和最后一层只有一个结点是可以直接得到的，在其它层中，回忆二项堆的定义是由两个  $B_k$  接起来的，因此第  $i$  层中根据归纳假设有  $\binom{k}{i} + \binom{k}{i-1} = \binom{k+1}{i}$  个结点，命题得证。 ■

根据二项堆结构性质（特别注意其中的 b），对于具有  $n$  个顶点的二项堆可以根据其二进制表示推出这个二项堆由哪些大小的二项树组成。

### 5.1.2 二项堆的操作与分析

最简单的操作是 FindMin，直接遍历全部  $O(\log n)$  棵树（回忆二进制表示）的根结点即可，因此时间复杂度是  $O(\log n)$ 。当然也可以通过专门记录最小的根结点来实现  $O(1)$  的时间复杂度，只是每次 DeleteMin 后要更新这一值。

然后是插入操作，和左式堆一样，插入只是特殊的合并，因此分析合并（merge）操作即可。实际上两个二项堆的合并想法也非常简单：既然每个二项堆都和唯一的二进制数对应，那合并后二项堆中二项树的分布情况不就符合二进制数的加法后的结果吗？结合 PPT 第 4 页的动画，merge 操作实际上就是从最小的堆开始（对应二进制最低位），如果无需进位（表明是  $1 + 0$  或  $0 + 1$ ）则直接留下称为新堆的一部分，如果需要进位则表明是  $1 + 1$ ，因此做合并后进位与下一位做加法，如此循环直到最高位完成操作。时间复杂度分析非常简单，在保证堆的存储顺序是按高度从小到大排列的前提下，时间复杂度很显然就是  $O(\log n)$ ，因为就是二进制逐位做操作。

插入（insert）是合并的特殊情况，操作不在此赘述，时间复杂度最坏为  $O(\log n)$ ，但从空开始建堆的时间复杂度是常数的，这一点稍后分析。DeleteMin 操作在 PPT 第 6 页也展示得很清楚了，实际上就是先用  $O(\log n)$  的时间找到根的最小值（或  $O(1)$ ，但这表明最后还需要更新最小值），设根最小的堆对应  $B_k$ ，于是可以得到两个堆，其一是整个二项堆移除  $B_k$  后剩下的堆，其二是  $B_k$  移除根结点后得到的堆，将这两个堆合并即可，时间复杂度显然是  $O(\log n)$ 。

对于从空开始插入建堆的时间复杂度需要做特殊的分析。因为是要平均时间的最坏情况，事实上也就是摊还代价，因此不难回想起三种方法。

**聚合法：**聚合法需要每一步的操作复杂度，实际上随便模拟几步再结合之前讨论的合并和二进制加法之间的关系就可以发现，插入的整个操作与二进制数加 1 有完全的对应关系：若是遇到了某一位是  $1 + 1$ ，则用常数操作完成简单的合并即可，如果遇到  $0 + 1$ ，那么当前所有的二项树合起来就是最后的结果。基于这一观察可知，因为  $0 + 1$  对应将 0 置 1， $1 + 1$  对应 1 置 0，这两种情况都对应于堆的常数时间操作，因此从空树连续插入  $n$  的顶点的时间复杂度与  $0 + 1 + 1 \dots$  ( $n$  个 1) 的过程中数据二进制表示中 0 和 1 比特翻转的次数总和。

于是算法复杂度就很好计算了，因为  $n$  对应于  $\lfloor \log n \rfloor + 1$  个二进制位，事实上最低位每次加 1 都会反转比特，次低位每两次运算反转比特，倒数第三位每 4 次运算反转比特……以此类推， $n$  次操作的整体时间复杂度与

$$n + \frac{n}{2} + \frac{n}{4} + \dots + \frac{n}{2^{\lfloor \log n \rfloor + 1}}$$

成正比，根据等比数列求和可知上述求和是小于  $2n$  的（取  $n \rightarrow \infty$  才能到  $2n$ ），所以单步操作的常数摊还时间也就得到了。

**核算法：**由于 PPT 上没有给出这一解法，在此留一个讨论题：

**讨论 1：**使用核算法证明二项堆从空堆开始连续插入  $n$  个结点，每一步操作的摊还代价为  $O(1)$ （提示：考虑二进制加法从 0 置 1（置位）和从 1 置 0（复位）两种操作）。

这里直接讨论置位和复位，从堆操作到二进制操作的对应是常数的，可以忽略。显然每次加 1 中置位操作只会出现 1 次，而复位操作可能有  $O(\log n)$  次，因此为了常数的摊还代价，复位操作的摊还代价设置为 0，因此每次复位欠下 1 个 credit 的债，置位为了补复位欠下的债，所以摊还代价必须要比真实操作 1 要大。实际上设置位的摊还代价为 2 即可，因为这样每次置位操作就可以为以后的复位操作存下 1 个 credit，也就是每次置位可以抵消未来一次复位欠的债。并且置位操作不可能比复位操作少，因为最开始所有位都是 0，没有 1，而最后的数字 1 的个数一定大于等于 0，这说明置位操作个数必定大于等于复位，因此置位存下的 credit 一定够复位消耗。再结合每次加 1 中置位操作只会出现 1 次，就可以得到每一步 insert 的摊还时间是常数的。

**势能法：**相信经历了这么多次势能函数的构造，读者对势能函数构造的方向应该很清楚了，那就是考虑进行复杂度最大的操作时定义怎样的势能函数能使得这一步势能下降很大。这里同样直接映射到二进制加法问题，不难发现复杂度很大的操作都对应于很多的复位和一个置位，复位导致 1 变 0。因此直接设势能函数等于加 1 后二进制中 1 的个数即可，对应于堆就是堆中插入后二项树的个数。正确性验证非常容易，此处不在赘述，如果还是会验证那可能说明在摊还分析上还存在较大的困难。

在本讲结束后会有堆加速 Dijkstra 算法的实验，在那里需要 DecreaseKey 的操作，事实上这在二项堆中是再简单不过的事情，因为每棵树都是二项树，所以类似于普通二叉堆的 DecreaseKey 操作（当然需要维护每个结点的指针，否则还要先找到这一结点，这是很耗时间的事情），只需要在违反序性质时将做了 DecreaseKey 的结点与父亲不断交换直到恢复序性质（percolate up）即可，时间复杂度显然也是  $O(\log n)$  的。删除任意结点操作显然也是和普通二叉堆一样顺其自然可以支持的，就是将想要删除的结点通过 DecreaseKey 将 key 值降低到最小，然后调用 DeleteMin 即可，显然时间复杂度为  $O(\log n)$ 。

### 5.1.3 二项堆的代码实现

因为每个结点的孩子数量可能不只有 2 个，因此使用 LeftChild 和 NextSibling 的组合实现。直观上来看用 LeftChild 和 NextSibling 是让二项树翻转了：原先是根的子树从左到右高度依次增大，现在依次减小了。并且为了方便索引每棵二项树，用一个数组存储每棵二项树的根，其中数组的索引就对应二项树的高度（见 PPT 第 7 页）。

因为插入、DeleteMin 的关键操作都是 merge，因此下面主要讨论 merge，而 merge 中经常需要合并两棵大小相同的二项树，因此首先介绍 combine 的实现。在 combine (PPT 第 9 页) 合并两棵相同大小的二项树时，结合堆的序性质以及 LeftChild 和 NextSibling 的实现，很容易想到其实只需要直接用根结点小的根作为新的根，根结点大的整棵树作为 LeftChild，NextSibling 接上根结点小的树除去根结点的其它部分即可。这里的时间复杂度显然是常数的。

接下来进入比较复杂的 merge 阶段 (PPT 第 10 页)。前面的内容很显然，下面重点分析循环中的 8 个 case。首先来看 case 是如何决定的，这里使用了双！，使用!! 的目的显然就是将可能不是 0 或 1 的值 bool 化，例如如果  $T_1$  存在 (即第一个堆对应高度为  $i$  的二项树存在)，则  $!!T_1$  做了两次取非操作后就变成了 1，如果  $T_1$  不存在，那么  $!!T_1$  ( $T_1$  实际上是 NULL) 就等于 0， $T_2$  和进位 carry 也是同理。

然后就能理解  $4*!!Carry + 2*!!T_2 + !!T_1$  的含义了，事实上这就是一个三位二进制数（当然 case 的标号还是十进制的，但我们心里要转化为二进制来分析），最高位表示是否有 carry，即之前的合并是否带来了进位（从堆的角度看也就是之前合并出了一棵新的更高的二项树），第二位代表第二个堆  $H_2$  是否有高度为  $i$  的二项树，最后一位代表  $H_1$  是否有高度为  $i$  的二项树。因此接下来的匹配过程实际上就是匹配竖式加法的各种情况，因为其中一位来源于  $H_1$ ，一位来源于  $H_2$ ，一位来源于进位，下面仔细分析各个情况（这里按照 0-7 的顺序，PPT 上先易后难把 3 和 4 顺序对调了，一定要看清楚）：

1. 000: 什么都不用做，只需等待循环结束后结束合并；
2. 001: 什么都不用做，只需等待循环结束后结束合并；
3. 010: 因为最后是要返回  $H_1$ （也就是说  $H_1$  是合并后的结果），清空  $H_2$ ，因此此时只需要将  $H_2$  中的树转移到  $H_1$  然后把  $H_2$  对应位置改为 NULL，最后等待循环结束后结束合并即可；
4. 011: 从加法操作来看此时没有进位，但会产生进位，当前位需要置 0，对应于堆操作就是， $H_1$  和  $H_2$  当前位变为 NULL，进位等于两个堆该高度的二项树合并后的结果；
5. 100: 类似于 010 的情况，但此时是把 carry 接到  $H_1$  上，然后只需等待循环结束后结束合并即可；
6. 101: 从加法操作看此时会产生进位，当前位需要置 0，因此堆操作就是  $H_1$  当前位变为 NULL，新的 carry 等于  $T_1$  和当前的 carry 合并的结果；
7. 110: 与 101 类似，只是  $H_2$  当前位变为 NULL，新的 carry 等于  $T_2$  和当前的 carry 合并的结果；
8. 111: 此时是  $1 + 1$  还要加上进位 1，因此求和后有新的进位，当前位也是 1，因此让  $H_1$  当前位变为 carry，新的 carry 等于  $T_1$  和  $T_2$  合并的结果，最后给  $H_2$  当前位变为 NULL 即可。

如此分析后不难发现，尽管代码看起来很复杂，但如果将以上情况与竖式加法一一对应，实际上是非常容易理解的。有一个细节是 for 循环的终止条件，是由变量 j 控制的，这里需要稍微理解一下，实际上很简单，就是等价地确定最多会循环多少次。最后是 DeleteMin 的实现（PPT 第 11 页），这里与先前讨论的步骤完全对应，所以并没有什么好强调的。

**讨论 2：**本题希望你设计支持可合并堆操作的 2-3-4 堆。这里的 2-3-4 与课上学习的 2-3-4 的 B+ 树是类似的，但也有部分不同：

1. 只有叶子结点存放 key，并且每个叶子结点只存放一个 key，并且叶子结点中的 key 也是无序的，不像 B+ 树中从左到右是从小到大的；
2. 每个非叶结点（包括当根结点不为叶子结点时的根结点） $x$  存放以  $x$  为根的子树中叶子结点存储的最小 key；
3. 根结点除了第 2 点外还需存储树的高度。

最后，2-3-4 堆设计放在内存中，磁盘读写是不需要的。你的任务是实现下面的 2-3-4 堆操作，在一个有  $n$  个元素的 2-3-4 堆上，1-5 每个操作都应当在  $\log n$  时间内完成，6 的合并应当在  $\log n$  时间内完成，其中  $n$  为两个堆元素个数之和。

1. FindMin: 返回指向堆中 key 最小的结点的指针；
2. DecreaseKey: 将给定结点的 key 减小为给定值；
3. Insert: 插入一个给定 key 的叶结点；
4. Delete: 删除一个给定 key 的叶结点；
5. DeleteMin: 删除最小结点；
6. Merge: 合并两个 2-3-4 堆，返回唯一的 2-3-4 堆，销毁原先的 2-3-4 堆。

(本题的可能实现肯定不止一种，不同方案之间可能有少部分细节的差别，合理即可)

【注：本题来源于《算法导论》（第三版）思考题 19-4，对答案感兴趣的同学善用互联网搜索即可得到答案。】

## 5.2 堆加速 Dijkstra 算法

### 5.2.1 回顾 Dijkstra 算法

Dijkstra 算法是著名的求解单源最短路径的算法，当然有一个重要的条件是没有负边。之后会在动态规划一讲中介绍存在负边的单源最短路径问题的解决思路，以及所有结点对的最短路径问题。

【提醒：这里并不是详解堆加速 Dijkstra 算法，只是叙述一下大致思路，具体实现还需靠各位读者自己努力或者查阅带图示的讲解加深理解】

现在简单回顾 Dijkstra 算法的流程，并从中找到需要用堆加速的理由。实际上 Dijkstra 算法的资料非常丰富，无论是上学期的 PPT 还是教材，还是网络上寻找都可以了解，例如[维基百科](#)，这里只把大致思想简单描述，不熟悉的读者请务必阅读其它地方更详细的版本。从算法的第一步开始，Dijkstra 算法维护两个集合， $X$  和  $V - X$ （假设  $V$  是所有顶点的集合）， $X$  中存的就是 Dijkstra 算法已经访问过的结点，也就是已经确定了最短路径的结点。同时算法维护每个顶点的一个 Dijkstra 得分，也就是当前状态下源结点到该顶点的最短路径长度，暂未到达的长度为  $+\infty$ 。算法的每一步都要从  $V - X$  中选择一个 Dijkstra 得分最低的点，也就是从当前状态下算法认为的  $V - X$  中与源结点距离最近的顶点，将其加入  $X$  中。就这样逐步加入，直到  $V = X$  结束。

### 5.2.2 堆加速 Dijkstra 算法

读者应当不难发现，在算法的执行过程中，每一轮循环都需要找到集合  $V - X$  中 Dijkstra 得分最低的顶点（也就是距离最近的顶点），将其取出，于是我们会很自然地想到最小堆的 DeleteMin 操作。因此可以维护一个存有  $V - X$  中所有顶点的最小堆，每个顶点的 key 就是当前的 Dijkstra 得分值。每一轮迭代可能需要对其中的顶点进行 DecreaseKey 操作，因为一个新的顶点加入  $X$  后可能使得  $V - X$  中顶点的 Dijkstra 得分降低（也可以直接删除然后插入新的，如果你没有支持 DecreaseKey 操作，例如斜堆中不应当有 DecreaseKey）。然后再调用 DeleteMin 取出最小顶点加入  $X$ 。

需要注意的是，回顾上一讲的讲义，DecreaseKey 操作需要维护每个顶点的位置信息（例如用指针），否则需要首先用线性时间找到它再进行下一步操作。

接下来讨论 Dijkstra 的算法复杂度。事实上，无论我们用何种数据结构，Dijkstra 算法的复杂度都可以表达为如下形式：

$$O(|E| \cdot T_{dk} + |V| \cdot T_{em}),$$

其中  $T_{dk}$  和  $T_{em}$  分别表示 DecreaseKey 操作和 DeleteMin 操作的时间复杂度。这一点并不难验证，因为每一条边的加入都可能导致一次 DecreaseKey 操作，而每一个结点都会在被 DeleteMin 操作取出。于是可以分别代入以下两种存储顶点的数据结构，得到如下结论：

1. 数组：用数组存储顶点，DecreaseKey 只需要降低数值即可，因此是  $O(1)$  的，但 DeleteMin 操作是  $O(|V|)$  的，因此整体复杂度为  $O(|E| + |V|^2) = O(|V|^2)$ 。
2. 堆：显然两种操作都是  $O(\log |V|)$  的，因此整体复杂度为  $O((|E| + |V|) \log |V|)$ ，如果所有结点都可以从源结点到达，这一复杂度为  $O(|E| \log |V|)$ 。

当图不太稠密的时候， $|E| = O(|V|^2 / \log |V|)$ ，使用堆的方法时间复杂度会更低（很容易验证），这也是堆加速的意义所在，因为在实际大型路线规划等场景中，更容易遇到稀疏图的情况，所以使用堆会使得算法性能更好。

### 5.2.3 斐波那契堆

如果我们希望在任何时候堆加速的 Dijkstra 算法都能在理论上优于数组版本，那么就要求 DecreaseKey 操作是常数的。这很难直接实现，但斐波那契度在理论上实现了 DecreaseKey 操作的常数摊还时间复杂度（但实际上常数非常大，因此实际性能完全不如其它操作简单的堆，同学们在 project 中会体会到这一点）。

如果你要完成本次 project 或者对这一内容感兴趣，那么关于斐波那契堆的具体设计与实现请参考《算法导论》等书籍、资料，这里不对现有资料做无趣的摘抄。下表是总结的各种堆的操作时间，注意表中的  $O(1)$  都代表摊还代价，二叉堆的建堆  $O(1)$  时间没有体现在表格中，因为回顾建堆操作，事实上这并不是通过普通的插入实现的。斜堆的删除和 DecreaseKey 操作不做讨论，因此也不要求大家实现。

	二叉堆	左式堆	斜堆	二项堆	斐波那契堆
Insert	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
Merge	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
DeleteMin	$O(\log n)$				
Delete	$O(\log n)$	$O(\log n)$		$O(\log n)$	$O(\log n)$
DecreaseKey	$O(\log n)$	$O(\log n)$		$O(\log n)$	$O(1)$

**讨论 3:** 回忆最小生成树的 Prim 算法，写出堆优化的 Prim 算法的大致思路（可以以伪代码形式呈现，或者有详细的文字描述），关键问题在于堆加速是加速了什么步骤，应当把什么存在堆中，需要对堆做什么操作。

## Lecture 6: 回溯法

编写人: 吴一航 [yhwu\\_is@zju.edu.cn](mailto:yhwu_is@zju.edu.cn)

从本讲起开始介绍算法设计策略, 将主要介绍回溯、分治、动态规划、贪心、NP 问题、近似算法、局部搜索、随机算法、并行算法以及外部排序, 其中从 NP 问题到随机算法这四讲将是这门课最具理论色彩的部分, 其中有无数闪烁着人类智慧的光芒伟大结论, 也有留待未来的天才解决的迷人猜想。当然这也意味着这几讲的内容颇具挑战性, 因为事实上在具有强大的理论计算机底蕴的学校, 这些内容每一章都可以用整整一门课来讲解, 上千页的教材都无法完全展现其魅力, 而我们只能在有限的课时中领略一些思想。

在接下来对算法设计的讨论中, 有两个要求将是讨论的核心:

1. 有效性 (effectiveness): 能保证算法的结果是正确的, 或者之后近似算法保证结果在一定近似比内, 或者随机算法保证一定概率内是合理的;
2. 有效率 (efficiency): 算法的运行速度应当是快速的。

在之后所有的算法中都会思考这两个问题 (当然对于分治、动态规划等问题, 有效性的保证看起来比较 trivial)。实际上, 在学习数据结构的时候也需要考虑操作保持数据结构的结构和序性质, 以及希望操作是比较快速的, 因此这些目标都是有共通性的。

本讲从思想最简单、看起来不那么应该出现在高级算法设计课程中的回溯法开始。需要注意的是, 接下来的讲义很多算法都不会像数据结构操作那样具体分析了, 因为理解算法过程非常简单, 更重要的是基本思想的传递以及思想的更丰富的应用, 以及在理论章节中会补充更多的理论知识, 供希望深入理解这些有趣但具有挑战性的专题的同学阅读。

## 6.1 回溯的基本思想与应用

### 6.1.1 基本思想

有些问题要求我们给出符合条件的解, 而所有的可能性是有限的, 这时可以通过所谓的暴力搜索 (exhausting search), 或称蛮力法 (Brute Force) 来对所有的可能性逐一检查得到正确的解。这样的方法显然可以保证算法有效性, 因为所有可能都被检查了, 然而在很多问题中, 所有可能性的个数 (搜索空间) 可能是相当巨大的, 这样的算法显然不是很有效率。

然而在很多情况下，我们可以很轻松地通过一些判断将很大一部分可能的结果排除在正确答案之外，这样的排除过程我们称为“剪枝”（pruning），至于为什么叫这个名字，是因为经常会将搜索正确答案的过程用一棵树表示，我们在八皇后问题中就能看清这一点。

### 6.1.2 八皇后问题

这是一个非常简单且经典的例子，因为算法太简单这里就不再赘述，这里只强调一个建树的思想。事实上树的建立过程就是模拟搜索决策过程：每一步都有很多种构建最终答案的选择的可能，在选择了其中一步后对于下一步构建答案又有新的可能，依此类推得到所有可能的决策路径。然后在这棵树上做剪枝即可，在八皇后问题中，剪枝的条件判别非常容易（见 PPT 第四页），因此不再赘述。

当然八皇后问题可以直接推广为  $n$  皇后问题，对具体实现感兴趣的读者可以自行阅读相关资料，例如[这个网页](#)。

当然说到复杂度分析不得不提到一些需要注意的结论。在回溯中经常遇到阶乘如  $n!$  的复杂度，也会碰到指数次方的复杂度。我相信读者只要简单回忆微积分或数学分析中学过的求极限的方法就能回忆起如下结论：

$$\lim_{n \rightarrow \infty} \frac{n!}{n^n} = \lim_{n \rightarrow \infty} \frac{a^n}{n!} = \lim_{n \rightarrow \infty} \frac{n^b}{a^n} = 0,$$

当然还有[斯特林公式](#)等更精确的估计，相信很多读者之前也有所了解。

### 6.1.3 收费公路问题

同样是一个很简单的算法，难点可能在于标题英文单词不认识（开个玩笑）。收费公路重建问题（turnpike reconstruction problem）是给定一个点集，以及其中点的两两之间距离然后重构出点集中各点位置的问题，这个名称得自于对美国西海岸公路上那些收税公路出口的模拟，它在物理学和分子生物学中都有应用。

想法非常简单，就是首先确定点的个数以及两个端点的位置（当然左端点可以默认为 0），接下来每一步考察目前剩余的最大的距离，这个距离应当是和两个端点之间的距离（为什么呢？例如 PPT 上的例子，第二次考察距离为 7 的点，有的同学可能会想这个距离 7 会不会是与之前选定的 8 之间的距离，即有一个点在 1 处呢？显然这是不可能的，因为 1 和 10 距离为 9，大于第一步的 8. 更抽象而言，每一次都会取出剩下的最大距离，因此当前的最大距离只可能是和端点的最大距离，否则新加入的点到端点的距离等于它到中间我们取的点的距离加上中间点到端点的距离，这比目前最大距离要大，而我们已经没有剩下这么大的距离了，因此不可能出现这样的情况），于是会有两种选择，从而可以逐步构造出一棵树。决定某个解是否成立的方法就是判断剩余的距离是否能满足这些点之间的距离。

伪代码实际上就是以上思路的直接体现，首先通过最大距离取点，然后判断这个点合不合要求，然后递归，递归成功则找到一种解，递归失败则撤回操作返回上一层，选择另一种可能。事实上这也引出了一

般回溯问题的框架，在 PPT 第 10 页有介绍。

#### 6.1.4 博弈

如果一个问题涉及双人乃至多人决策，那么就是一个博弈问题，Tic-tac-toe 就是一个典型的博弈问题。无论讨论什么博弈，如下几个条件都是必不可少的：

1. 参与人：在 Tic-tac-toe 中就是两个参与方，计算机与人类；
2. 策略：在 Tic-tac-toe 博弈中，策略就是计算机或人类在看到一个棋局后会作出的行动决策；
3. 效用函数：上过微观经济学课程的同学应该并不陌生，没上过的同学应当也知道所谓经济学的理性人假说。在一般博弈中，理性人追求的是自己的效用最大化，通俗来说就是自己越开心越好，当然在 Tic-tac-toe 中就是赢的可能越大越好。PPT 中将赢的可能用自己与对手的“number of potential wins at position P”之差来表达（number of potential wins 就是当前局势下，最大有多少种可能的赢法，也就是最后 8 种赢法中还剩几种是当前局势下能达到的，8 种赢法就是 3 横 3 竖 2 对角线）。

事实上，这上面的策略和效用函数是息息相关的，参与者希望他的行动能让他效用最大化，这就是参与者决定他的策略的出发点。在 PPT 中，我们为参与者制定的策略是所谓的“最大最小策略”（或“最小最大策略”），即已知别人是要最大化效用的，所以我的选择是要最小化别人最大化的效用（或者我知道别人要最小化我的效用，我就要最大化别人最小化的我的效用）。事实上，在两人零和博弈中（Tic-tac-toe 就是，只需看效用函数的定义就可知两个参与者的效用恰为相反数），这一最小最大策略得到的解比纳什均衡具有更好的性质。感兴趣的同学可以进一步了解博弈论相关的基本概念，毕竟同学们基本都是学习计算机专业的，祖师爷冯·诺依曼的很多著名工作都是与博弈论相关的（因此他也被尊称为“博弈论之父”），了解一些博弈论的常识还是有趣且有必要的。

当然我们的重点是回溯以及剪枝，这里重点介绍了  $\alpha - \beta$  剪枝。所谓  $\alpha$  剪枝就是当我要最大化别人最小化的我的效用时，如 PPT 第 16 页所示，我已经知道，如果我选左边这条路，最少也能得到 44 的效用，但如果我选择右边这条路，叶子有一个结点使得我只能得到 40 的效用，又因为别人一定是要最小化我的效用的，所以右边这条路不管右下方那个叶子是多少，我得到的效用也不超过 40 了，所以右下角那个点就被  $\alpha$  剪枝剪掉了。 $\beta$  剪枝是对称的情况，这里不再赘述，只强调这里要区分清楚二者的差别，并且自己要梳理清楚逻辑，然后注意被剪掉的是 PPT 图中染黑的点，这一问题在作业、考试中还是很常见的。

当然，在搜索空间太大的时候， $\alpha - \beta$  剪枝也是很复杂的，所以可能会随机采样几种情况进行模拟，最大化采样到的情况的效用，这也就是所谓的“蒙特卡罗方法”的基本想法。

## 6.2 讨论题

**讨论 1:** 在国际象棋中, 设  $B$  是棋盘的大小, 规定在  $R$  行  $C$  列上的马可以走到  $1 \leq R' \leq B$  行和  $1 \leq C' \leq B$  列处, 且要么

$$|R - R'| = 2 \text{ 及 } |C - C'| = 1,$$

要么

$$|R - R'| = 1 \text{ 及 } |C - C'| = 2,$$

(即马只能横跳两格加纵跳一格, 或者横跳一格加纵跳两格) 马的一次环游是马在棋盘上的一系列跳行, 它恰好访问所有的方格一次最后又回到开始的位置。

1. 如果  $B$  是奇数, 证明马的环游不存在 (提示: 考虑国际象棋棋盘有黑色和白色格子, 找到马跳和黑白格之间的关系);
2. 若存在马的环游, 给出一个回溯算法找出马的一次环游 (核心是给出剪枝策略, 最好能比最容易想到的条件有优化)。

**【注:】**这一问题被称为 closed knight's tour problem, 回溯算法对于这一问题而言显得非常暴力, 感兴趣的读者可以进一步参考[维基百科](#)等资料。

**讨论 2:** 证明: Tic-tac-toe 中, 下面三种情况有且只有一种情况成立:

1. 计算机 (画圈) 有一个制胜策略;
2. 人类 (画叉) 有一个制胜策略;
3. 双方都有一个至少保证平局的策略。

**【提示:】**可以搜索冯·诺依曼从国际象棋博弈出发得到的类似结论, 有两种常见的证明方法, 第一是从前往后的数学归纳法, 第二是使用逻辑表达式。

## Lecture 7: 分治法

编写人: 吴一航 yhwu\_is@zju.edu.cn

## 7.1 主定理

分治法我想诸位已经非常熟悉，因此这里也不再谈论一些基础的话题，直接开始讨论如何分析分治法的时间复杂度。通常而言，分治法递归的时间复杂度递推公式具有如下形式：

$$T(n) = aT(n/b) + f(n),$$

其中  $a$  是分了多少个子问题， $b$  与子问题划分方式有关，是子问题输入长度的收缩因子， $f(n)$  则是合并各个子问题的解需要的时间。当然练习中会看到不这么寻常的结构，这些留在讨论题中供诸位思考。

### 7.1.1 代入法

美其名曰代入法，其实就是猜证，显得有些耍无赖，但面对考试中都是选择判断的情况，这种方法显得尤为便捷。代入法一般分为两步：

1. 猜测解的形式；
2. 用数学归纳法求出解中的常数，并证明解是正确的。

这一方法在 PPT 的 8-9 页给出了例子，第 8 页关于归纳基础的讨论，希望表达的意思是如果常数取的足够大，那么在  $N$  小的情况下不管什么复杂度都是正确的，所以无需关心归纳基础。第 9 页的例子特别需要强调这里的常数  $c$  是要保持的，否则常数也会与  $N$  有关（ $N$  增加，常数增加），那就不再是常数了。至于如何猜测一个好的界，可以回忆一些熟悉的结论，例如归并排序这类，也可以猜测比较松的上下界，然后往中间逼近最紧的。

下面介绍一个常用的技巧，即换元法。来看下面这个问题：

【讨论 1:】  $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$ 。

为方便起见，不必担心值的舍入误差问题，只考虑  $\sqrt{n}$  是整数的情形即可。令  $m = \log n$ ，得到

$$T(2^m) = 2T(2^{m/2}) + m,$$

然后令  $S(m) = T(2^m)$ , 则有  $S(m) = 2S(m/2) + m$ , 这就是熟悉的归并排序结果, 因为  $S(m) = O(m \log m)$ , 因此  $T(n) = \log n \log \log n$ 。

下面是一个从算法导论截取的技巧, 对于一些情况会有所帮助:

### 微妙的细节

有时你可能正确猜出了递归式解的渐近界, 但莫名其妙地在归纳证明时失败了。问题常常出在归纳假设不够强, 无法证出准确的界。当遇到这种障碍时, 如果修改猜测, 将它减去一个低阶的项, 数学证明常常能顺利进行。

考虑如下递归式:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

我们猜测解为  $T(n) = O(n)$ , 并尝试证明对某个恰当选出的常数  $c$ ,  $T(n) \leq cn$  成立。将我们的猜测代入递归式, 得到

$$T(n) \leq c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1 = cn + 1$$

这并不意味着对任意  $c$  都有  $T(n) \leq cn$ 。我们可能忍不住尝试猜测一个更大的界, 比如  $T(n) = O(n^2)$ 。虽然从这个猜测也能推出结果, 但原来的猜测  $T(n) = O(n)$  是正确的。然而为了证明它是正确的, 我们必须做出更强的归纳假设。

直觉上, 我们的猜测是接近正确的: 只差一个常数 1, 一个低阶项。但是, 除非我们证明与归纳假设严格一致的形式, 否则数学归纳法还是会失败。克服这个困难的方法是从先前的猜测中减去一个低阶项。新的猜测为  $T(n) \leq cn - d$ ,  $d$  是大于等于 0 的一个常数。我们现在有

$$\begin{aligned} T(n) &\leq (c \lfloor n/2 \rfloor - d) + (c \lceil n/2 \rceil - d) + 1 \\ &= cn - 2d + 1 \leq cn - d \end{aligned}$$

只要  $d \geq 1$ , 此式就成立。与以前一样, 我们必须选择足够大的  $c$  来处理边界条件。

你可能发现减去一个低阶项的想法与直觉是相悖的。毕竟, 如果证明上界失败了, 就应该将猜测增加而不是减少, 更松的界难道不是更容易证明吗? 不一定! 当利用归纳法证明一个上界时, 实际上证明一个更弱的上界可能会更困难一些, 因为为了证明一个更弱的上界, 我们在归纳证明中也必须使用同样更弱的界。在当前的例子中, 当递归式包含超过一个递归项时, 将猜测的界减去一个低阶项意味着每次对每个递归项都减去一个低阶项。在上例中, 我们减去常数  $d$  两次, 一次是对  $T(\lfloor n/2 \rfloor)$  项, 另一次是对  $T(\lceil n/2 \rceil)$  项。我们以不等式  $T(n) \leq cn - 2d + 1$  结束, 可以很容易地找到一个  $d$  值, 使得  $cn - 2d + 1$  小于等于  $cn - d$ 。

【讨论 2:】  $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ 。

$n$  很大的时候, 17 是完全可以忽略的, 因此直接猜测  $O(n \log n)$  然后验证即可, 注意这里可能需要用到上面的技巧。

### 7.1.2 递归树方法

最直白且最自然的方法, 相信直接看出来树怎么画 (注意树每一层的复杂度就是合并复杂度, 每个叶子的复杂度就是 base case 的复杂度, 显然为常数), 知道基本的数列求和公式, 这个方法对于能用得上的场景都是很简单的, 这里也就不再赘述了 (PPT 第 10-11 页), 更具体的描述可以阅读算法导论。

【注:】PPT 第 11 页的例子可以用更高级的定理解决, 详情请看[Akra-Bazzi 定理的维基百科](#)。

需要注意的是, 第 10 页的例子与接下来的主定理有很大的关联, 在第 10 页的例子中, 最后求和分为两个部分, 第一个部分是每层的时间, 其实就是每次递归后合并解需要的时间和 (注意 PPT 的图上每层右边的大小是从下一层合并到该层需要的时间, 而非该层合并到上一层需要的时间), 第二个部分则是递归到 base case 的子问题个数, 每个需要常数的时间操作然后 return 然后开始合并。这两个部分谁更大, 谁就 master 了整个时间复杂度 (这一例子中是每层合并时间 master 了), 这也是接下来主定理 (master theorem) 的思想。除此之外, 通过画递归树得到的结果需要用替代法验证一下正确性。

【讨论 3:】给定一个整数  $M$ ,

$$T(n) = \begin{cases} 8T(n/2) + 1 & n^2 > M \\ M & \text{otherwise} \end{cases}.$$

有  $\log_2 n / \sqrt{M} + 1$  层, 第  $i$  层  $8^{i-1}$  个单位时间, 且有  $8^{\log_2 n / \sqrt{M}}$  个叶子, 每个叶子  $M$  个单位时间, 故总时间为

$$O(M \cdot 8^{\log_2 n / \sqrt{M}} + \sum_{i=0}^{\log_2 n / \sqrt{M}} 8^i) = O(n^3 / \sqrt{M}).$$

事实上考试中也出现过类似的问题:

Consider the following pseudo-code.

```
strange(a1, ..., an):
1. if n ≤ 2022 then return
2. strange(a1, ..., a⌊n/2⌋)
3. for i = 1 to n:
4.   for j = 1 to ⌊√n⌋:
5.     print(ai + aj)
6. strange(a⌊n/2+1⌋, ..., an)
```

What is the running time of this pseudo-code? Your answer should be as tight as possible. (You may assume that  $n$  is a power of 2.)

- A.  $O(n^2)$
- B.  $O(n^{1.5}) \log n$
- C. None of the other options is correct
- D.  $O(n^{1.5})$

可以将这一问题转化为上面的分段表达式, 然后求解。

### 7.1.3 主定理

主定理的证明属实有点老太太裹脚布, 感兴趣的同学可以阅读算法导论, 这里没必要重复这些纯技术性的过程。

考虑如下形式：

$$T(n) = aT(n/b) + f(n), \quad a \geq 1, b \geq 2$$

- 定理 7.1**
1. 若对于某个  $\varepsilon$  有  $f(n) = O(n^{\log_b a - \varepsilon})$ , 则  $T(n) = \Theta(n^{\log_b a})$ ;
  2. 若  $f(n) = \Theta(n^{\log_b a})$ , 则  $T(n) = \Theta(n^{\log_b a} \log n)$ ;
  3. 若对于某个  $\varepsilon$  有  $f(n) = \Omega(n^{\log_b a + \varepsilon})$ , 且对某个常数  $c < 1$  和所有足够大的  $n$  有  $af(n/b) \leq cf(n)$ , 则  $T(n) = \Theta(f(n))$ .

注意，这三种情况并未覆盖  $f(n)$  的所有可能性。情况 1 和情况 2 之间有一定间隙，情况 2 和情况 3 之间也有一定间隙。如果函数  $f(n)$  落在这两个间隙中，或者情况 3 中要求的正则条件不成立（关于正则条件可以看后面的主定理第二种形式的推导理解，也可以直接看证明理解），就不能使用主方法来求解递归式。下面看一个不符合正则条件的例子：

**例 7.2**  $T(n) = T(n/2) + n(2 - \cos n\pi)$ , 此时  $a = 1, b = 2$ , 且  $\cos$  函数的取值范围是  $-1$  到  $1$ , 因此满足上述第三种情况的第一个条件, 然而不满足第三种情况的正则条件。因为如果满足正则条件, 代入有

$$\frac{n}{2} \left(2 - \cos \frac{n\pi}{2}\right) \leq cn(2 - \cos n\pi)$$

对某个常数  $c < 1$  和充分大的  $n$  都成立。然而, 当  $n$  很大时, 总可以取  $n = 4k + 2$  这种形式破坏上面这一不等式。

实际上主定理并不需要死记硬背, 无论什么形式的主定理 (后面会介绍其它形式的), 其实关键都在于比较  $n^{\log_b a}$  和  $f(n)$  之间的关联, 如果前者大, 则前者“掌控了”整个时间复杂度, 所以时间复杂度就是  $T(n) = aT(n/b)$  对应的复杂度, 这也很符合直观, 因为此时  $a$  比较大, 分叉比较多, 树比较大 (结合前面递归树的例子), 所以更大的复杂度会落在叶子上; 反之后者大则每一层的复杂度“掌控了”整个时间复杂度, 故整体时间就是  $f(n)$  级别的。这也就是“主定理”这一名字的含义, 十分形象, 就是看前后两半谁 master 了整体时间复杂度, 具体为什么是比较这两个函数则是具体证明中可以很容易看出的。

#### 【讨论 4:】

1.  $T(n) = 4T(n/2) + n/\log n$ ;
2.  $T(n) = 2T(n/2) + n/\log n$ .

第一个直接使用主定理即可, 结果为  $\Theta(n^2)$ ; 第二个不能使用主定理, 因此用递归树, 递归树有  $\log_2 n$  层, 第  $i$  层时间复杂度  $n/\log(n/2^i)$ , 叶子有  $n$  个, 故整体复杂度为

$$\sum_{i=0}^{\log_2 n-1} \frac{n}{\log(n/2^i)} + \Theta(n) = \sum_{i=0}^{\log n-1} \frac{n}{\log_2 n - i} + \Theta(n) = \sum_{j=1}^{\log_2 n} \frac{n}{j} + \Theta(n) = O(n \log \log n).$$

最后代入检验也没问题。注意最后一步用了调和级数的估计结论，如果大家还能回忆起微积分中的欧拉常数，那么这个估计也是很熟悉的。

在这一问题中给出了一个主定理不适用的场景，实际上还有其它场景，例如递推式中  $a$  不是常数， $a$  和  $b$  取值不在规定范围内等，做题时需要多加小心（但事实上本题可以通过本节最后给出的主定理推广形式解决）。

下面介绍主定理相对比较简单的两种形式，第一种实际上是前面原始版本的推论：

- 定理 7.3**
1. 若对于某个常数  $c > 1$  有  $af(n/b) = cf(n)$ ，则  $T(n) = \Theta(n^{\log_b a})$ ；
  2. 若  $af(n/b) = f(n)$ ，则  $T(n) = \Theta(n^{\log_b a} \log n)$ ；
  3. 若对于某个常数  $c < 1$  有  $af(n/b) = cf(n)$ ，则  $T(n) = \Theta(f(n))$ .

不难看出是推论，例如对于第一种情况， $af(n/b) = cf(n)$  表明  $f(n) = a/cf(n/b)$ ，故大致可以递推得到

$$f(n) = \frac{a^{\log_b n}}{c^{\log_b n}} = \frac{n^{\log_b a}}{n^{\log_b c}} = O(n^{\log_b a - \varepsilon}),$$

正好适用于原始方法的第一种情况。第二、三种情况事实上也是同样的推导。从这里可以看出，对某个常数  $c < 1$ ，正则条件  $af(n/b) \leq cf(n)$  成立本身就意味着存在常数  $\varepsilon > 0$  使得  $f(n) = \Omega(n^{\log_b a + \varepsilon})$ ，因此前面的正则条件的意义在这里也可以看出。

第二种则在某些情况下比之前介绍的原始形式更强，首先介绍这一结论，简要说明其证明，然后给出一个经典的例子：

**定理 7.4** 对于递推式  $T(n) = aT(n/b) + \Theta(n^k \log^p n)$ ,  $a \geq 1, b > 1, k \geq 0, p \geq 0$ ,

1. 若  $a > b^k$ ，则  $T(n) = \Theta(n^{\log_b a})$ ；
2. 若  $a = b^k$ ，则  $T(n) = \Theta(n^k \log^{p+1} n)$ ；
3. 若  $a < b^k$ ，则  $T(n) = \Theta(n^k \log^p n)$ .

这里的第1种情况实际上就可以直接对应到主定理原始形式的第一种情况，第3种情况也对应原始形式第三种情况（包括正则条件），重点是第2种情况，相比于原先的形式更加强大，可以看下面这个例子：

**例 7.5** 考虑  $T(n) = 2T(n/2) + n \log n$ ，则原始版本和这里的第1种都是不可行的，但第3种形式可以使用。

至于第二种情况的证明，相对而言较为复杂。使用递归树进行分析，并且为了简化讨论，假设  $n$  是  $b$  的幂次。则很容易利用递归树得到（不妨设  $T(n) = aT(n/b) + cn^k \log^p n$ ）：

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_b n-1} a^i c \left(\frac{n}{b^i}\right)^k \log^p \frac{n}{b^i} + a^{\log_b n} T(1) \\ &= cn^k \sum_{i=0}^{\log_b n-1} \log^p \frac{n}{b^i} + n^k (\text{注意 } a = b^k), \end{aligned}$$

所以现在的问题就是求出对数和  $\sum_{i=0}^{\log_b n-1} \log^p \frac{n}{b^i}$ ，有一个简单的观察

$$\log^p \frac{n}{d} = (\log n - \log d)^p = \log^p n + O(\log^p n),$$

所以有

$$\begin{aligned} \sum_{i=0}^{\log_b n-1} \log^p \frac{n}{b^i} &= \sum_{i=0}^{\log_b n-1} (\log^p n + O(\log^p n)) \\ &= \log_b n \log^p n + \log_b n \cdot O(\log^p n) \\ &= \Theta(\log_b n \log^p n) = \Theta(\log^{p+1} n), \end{aligned}$$

代入前面的推导，可以得到  $T(n) = \Theta(n^k \log^{p+1} n)$  成立。

**【注：】** 上一主定理的形式还有更强的表达：

**定理 7.6** 对于递推式  $T(n) = aT(n/b) + \Theta(n^k \log^p n)$ ,  $a \geq 1, b > 1, k \geq 0$ , 而  $p$  为任意实数,

1. 若  $a > b^k$ , 则  $T(n) = \Theta(n^{\log_b a})$ .
2. 若  $a = b^k$ , 则
  - (a) 若  $p > -1$ ,  $T(n) = \Theta(n^k \log^{p+1} n)$ ;
  - (b) 若  $p = -1$ ,  $T(n) = \Theta(n^k \log \log n)$ ;
  - (c) 若  $p < -1$ ,  $T(n) = \Theta(n^k)$ .
3. 若  $a < b^k$ , 则
  - (a) 若  $p \geq 0$ ,  $T(n) = \Theta(n^k \log^p n)$ ;
  - (b) 若  $p < 0$ ,  $T(n) = \Theta(n^k)$ .

在这一形式下，讨论 4 的第二问实际上也在推广的主定理的讨论范围内了。更多的关于这一推广的描述与可以或不可以使用主定理的例子见[这个链接](#)，部分证明感兴趣的的同学可以参考[这个文档](#)。

## 7.2 分治法应用

### 7.2.1 曾经学过的问题

在学习数据结构基础甚至普通 C 程时我们已经多次接触分而治之的思想，这一节就先来回顾一下曾经学过的那些问题，并用主定理给出一些基本的分析。

#### 7.2.1.1 最大子序列和问题

相信大家对这个例子并不陌生——这是数据结构基础第一节课的内容！按照规律第一节课的内容应该是学习得比较认真的……用这个例子来回顾一下基本思想。假定要寻找子数组  $A[low, high]$  的最大子序列和，首先找到数组中央位置  $mid$ ，分治法很自然地将这个问题的解  $A[i, j]$  变为以下三种情况之一：

1. 最大子序列和完全在  $A[low, mid]$  中，即  $low \leq i \leq j \leq mid$ ；
2. 最大子序列和完全在  $A[mid + 1, high]$  中，即  $mid + 1 \leq i \leq j \leq high$ ；
3. 最大子序列和跨越  $mid$  两边，即  $low \leq i \leq mid < j \leq high$ 。

所以想法就是：

1. 分治：将原数组平分为两半  $A[low, mid]$  和  $A[mid + 1, high]$ ，然后分别对这两半求解最大子序列和；一定不能忘记递归有 base case，这里的 base case 就是数组只剩下一个元素，那就什么都不用操作，直接返回进入下一步合并阶段；
2. 合并：首先要计算跨越  $mid$  两边的最大子序列和，然后和左右两半的结果比较，选择最大的作为最终结果。关键在于计算跨越  $mid$  两边的最大子序列和，这其实是线性的，为什么？因为这里的子序列都必须跨越  $mid$ ，所以最大子序列和就是最大的  $A[i, mid]$  加上最大的  $A[mid + 1, j]$ ，这个只需要对  $i, j$  做遍历就行，所以是线性的。

那么得到的递推公式就是很简单的  $T(n) = 2T(n/2) + O(n)$ ，所以就是  $O(n \log n)$  的复杂度。

#### 7.2.1.2 归并排序与快速排序

这个例子出现的时间应该比最大子序列和更早——学习 C 语言的时候应该就有所了解了！归并排序的思想相当简单，分治直接将数组分成两半然后分别排序即可，base case 就是分到数组长度为 1 了，此时什么都不用做直接返回进入合并阶段，而合并就是将两个排序好的小数组合并成一个大数组，这一步显然是线性的复杂度，这也就很轻松地得到  $O(n \log n)$  的复杂度。

至于快速排序，我们选择一个 pivot，然后首先将小于 pivot 的都放左边，大于的都放右边，这一步只需线性时间，然后对 pivot 左右的数组递归继续排序。所以这里的流程和之前的问题不太一样，先做了额外操作后才进行分治。在数据结构基础的讨论中可知快速排序最差可能到平方级别的运行时间，在 ADS 中将会在随机算法一节中引入随机性证明平均时间复杂度为  $O(n \log n)$ 。

### 7.2.1.3 逆序对计数

逆序对在定义行列式时已经见到（当然有的同学不一定学习了这种定义方式，但只需简单搜索就可以知道逆序对的含义），显然分治法计算逆序对就是分成三类：一是全在左半边的逆序对个数，二是全在右半边的，三是跨越中点的。看起来很正常，但这个跨越中点的逆序对个数似乎最多有  $n/2 \times n/2 = n^2/4$  个，这样分治法就失去了时间上的优势。改进的想法是站在归并排序的肩膀上，如果每次计算逆序对后也进行了排序，那么合并过程中计算跨越中点的逆序对个数就只需要线性时间了：在合并两个已排序的子序列的过程中很容易完成逆序对的计算，相信读者只需稍加思考就可以理解这一点，如果实在不理解相信网上有大量资料能说明白这一点。

### 7.2.2 最近点对问题

这是 PPT 上的例子，也是相当经典的应用。同样分为三个部分，左最近点对、右最近点对和分离最近点对，关键在于线性时间找到分离最近点对。这是一个相当聪明的算法，首先取  $x$  坐标来看的中点，其  $x$  坐标记为  $\bar{x}$ ，然后考虑  $[\bar{x} - \delta, \bar{x} + \delta]$  ( $\delta$  是左右两半中最近点对距离) 之间的所有点  $q_1, q_2, \dots, q_l$ ，它们按  $y$  坐标排序。设  $q_i$  的  $y$  坐标为  $y_i$ ，那么只需要对  $q_i$  检查  $x$  坐标在  $[\bar{x} - \delta, \bar{x} + \delta]$  之间， $y$  坐标在  $[y_i, y_i + \delta]$  之间构成的长方形区域中的所有点是否有更近的点对即可（所有点都往上找就行，往下和下面的点往上找重合）。将这个长方形区域分为平均分割为 8 块，则每块内最多出现一个点，否则每块内两点距离不超过  $\frac{\sqrt{2}}{2}\delta$ ，并且每块都不跨中点，这与  $\delta$  是左右两半中最近点对距离矛盾，因为可以找到距离更短的两点。所以对于每个  $q_i$ ，只需找向上 7 个点即可，所以找分离最近点对的时间复杂度是线性的。

但是，7 个点看起来可能有点太多了。实际上当前研究的点所在的一侧的所有点都完全可以不考虑，例如当前我们循环到了一个在左半边的点，那么整个左半边的点都不需要考虑，只需要考虑右半边 4 个格子最多的 4 个点。但要注意的是这种情况你需要提前维护好左右两边各自的按  $y$  坐标排序的点列，然后要维护  $y$  坐标紧跟着左半边的每个点的四个右半边的点，对称的右半边也要维护，这并不复杂。

然而实际上还可以进一步将 4 这个数字降为 3。因为每在右半边放一个点，那么在右半边，以这个点为圆心半径为  $\delta$  的圆内不可能再有另一个点，事实上最差的情况就是有四个这样的圆心，此时四个圆心在右半边区域的四个角上（也就是  $(0, 0), (0, \delta), (\delta, 0), (\delta, \delta)$ ），那这时其实找三个也就够了（其实最好是  $(0, 0)$  最好），而其它情况不可能有四个圆心（因为四个圆心互相的最近距离都只能是  $\delta$ ，最极端的情况也是一个正方形），最多三个圆心那么找三个也就足够了。

当然，其实三个点还是有点多，两个点也是可以说明的，诸位可以自己思考这一点，实际上如果理解了3个点的算法这一点也并不难。除此之外，这一问题还有一个经典的线性随机算法，在随机算法一节中会简单介绍。

### 7.2.3 从整数乘法到矩阵乘法

提到分治法，不得不提的就是 Strassen 矩阵乘法。这是一个看起来有点魔法，能将矩阵乘法从最简单的三次方算法复杂度下降的方法。为了使这一方法看起来不那么魔法，可以先介绍 Karatsuba 整数乘法算法作为基础。

考虑两个长度为  $n$  的整数做乘法（长度不同则短的高位补 0 补齐长度即可），如果不做优化，用传统的竖式乘法显然复杂度为  $O(n^2)$ 。如果用分治法，想法很简单，就是将两个整数分为两半分别计算乘法，然后组合起来即可。假设  $n$  是偶数（奇数就是不能平分，记号上麻烦一点，但本质是一样的），则可以设第一个乘数为  $a = 10^{n/2}a_1 + a_2$ ，第二个乘数为  $b = 10^{n/2}b_1 + b_2$ ，那么二者相乘的结果就是

$$ab = 10^n a_1 b_1 + 10^{n/2} (a_1 b_2 + a_2 b_1) + a_2 b_2.$$

因此在分治阶段需要计算  $a_1 b_1, a_1 b_2, a_2 b_1, a_2 b_2$  四个长度为  $n/2$  的乘法，合并阶段只需要简单移位和加法就行，复杂度是线性的，所以整体时间复杂度递推公式为

$$T(n) = 4T(n/2) + O(n).$$

尴尬的事情出现了，根据主定理这个时间复杂度是  $O(n^2)$ ，似乎优化失败了。其实关键就在于这里分治要计算四个乘法，如果能计算更少的乘法呢？注意到

$$a_1 b_2 + a_2 b_1 = (a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2,$$

而  $a_1 b_1$  和  $a_2 b_2$  是递归本来就要计算的，所以事实上分治时只要计算  $(a_1 + a_2)(b_1 + b_2)$ 、 $a_1 b_1$  和  $a_2 b_2$  三个乘法即可算出最后的乘法，然后合并时计算多了一步计算  $(a_1 + a_2)(b_1 + b_2) - a_1 b_1 - a_2 b_2$ ，其实就是减法，所以复杂度还是线性的，故递推式变为了

$$T(n) = 3T(n/2) + O(n),$$

很高兴，复杂度变为了  $O(n^{\log_2 3})$ ，这相比平方有了一定改进。

有了上面的基础，下面来看矩阵乘法。分治的想法显然是使用分块矩阵，考虑简单的情况，即矩阵是方阵的情况，设矩阵阶数  $n$  为偶数（同理奇数只是记号复杂，偶数不失一般性），将两个矩阵进行如下分块：

$$A = \begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix}, \quad B = \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix},$$

设结果矩阵

$$C = \begin{pmatrix} C_1 & C_2 \\ C_3 & C_4 \end{pmatrix},$$

则有

$$\begin{aligned} C_1 &= A_1B_1 + A_2B_3 \\ C_2 &= A_1B_2 + A_2B_4 \\ C_3 &= A_3B_1 + A_4B_3 \\ C_4 &= A_3B_2 + A_4B_4 \end{aligned}$$

于是分治需要计算 8 个阶数为  $n/2$  的矩阵乘法，然后用平方时间做矩阵加法即可，即时间复杂度递推公式为

$$T(n) = 8T(n/2) + O(n^2).$$

很遗憾，时间复杂度仍然是  $O(n^3)$ 。所以类似于 Karatsuba 整数乘法，我们希望用更少的乘法来表达出 8 个乘法。天才的 Strassen 在 1969 年给出了这一算法，他构造了如下七个矩阵

$$\begin{aligned} M_1 &= (A_1 + A_4)(B_1 + B_4) \\ M_2 &= (A_3 + A_4)B_1 \\ M_3 &= A_1(B_2 - B_4) \\ M_4 &= A_4(B_3 - B_1) \\ M_5 &= (A_1 + A_2)B_4 \\ M_6 &= (A_3 - A_1)(B_1 + B_2) \\ M_7 &= (A_2 - A_4)(B_3 + B_4) \end{aligned}$$

通过组合上述 7 个新矩阵得到：

$$\begin{aligned} C_1 &= M_1 + M_4 - M_5 + M_7 \\ C_2 &= M_3 + M_5 \\ C_3 &= M_2 + M_4 \\ C_4 &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

于是只需要线性时间计算加法和减法，然后分治只需要计算 7 个乘法即可，因此时间复杂度递推公式为

$$T(n) = 7T(n/2) + O(n^2),$$

因此复杂度变为了  $O(n^{\log_2 7})$ ，这相比三次方有了一定改进。

一个有趣但无解的问题是，Strassen 是怎么想出这 7 个矩阵的——他的论文也没给出解释，且当这是智慧与尝试的结晶吧。有些地方给了基于代数几何和表示论的解释，表明这样的组合一定存在，从更高的视角下给这个答案一个合理的解释。

CS2045M: 高级数据结构与算法分析

2025-2026 学年秋冬学期

## Lecture 8: 动态规划

编写人: 吴一航 yhwu\_is@zju.edu.cn

我想这毫无疑问将是期中之前这门课最令人感到噩梦的一章，因为动态规划并不是一个很 trivial 的方法，不像回溯、分治以及之后的贪心自然直观。但我相信，经过本讲的讲解与推导，以及诸位自己的仔细思考，至少对这门课的考试、未来的研究和工作都能打下坚实的基础。

## 8.1 动态规划的背景

### 8.1.1 动态规划的来由

1966 年，Dynamic Programming 登上国际顶级期刊《Science》，理查德·贝尔曼（Richard Bellman）为唯一作者，该论文被引 28421 次。贝尔曼在他的自传中写道：“1950 年秋季，我在兰德公司工作。我的第一个任务是为多阶段决策过程找到一个名称。”

“一个有趣的问题是，“动态规划”这个名字是从哪里来的？20 世纪 50 年代不是搞数学研究的好年代。当时在华盛顿有一位非常有趣的人，他叫威尔逊（Charles E. Wilson, 1953-1957 年任美国国防部长），是当时的美国国防部长，他对“研究”这个词有病态的恐惧和憎恨。提到研究，他的脸会涨得通红，如果有人在他面前用“研究”这个词，他就会变得很暴力。你可以想象他对数学这个词的感受。兰德公司受雇于空军，而空军的老板基本上是威尔逊。因此，我觉得我必须做点什么来瞒过威尔逊和空军，不让他们知道我实际上是在兰德公司做数学研究。”

“所以，给这个研究起个什么名字好呢？首先，我对 planning, decision making 和 thinking 比较感兴趣。但是由于种种原因，planning 这个词并不合适。因此，我决定使用 programming 这个词。我想让大家明白，这是动态的，是多级的，是时变的——我想，这个名字可以起到一石二鸟的效果。作为形容词，dynamic 还有一个很有趣的特性，那就是 dynamic 这个词不可能有贬义。我们不可能想出一些可能使它具有贬义的组合。因此，我认为 dynamic programming 是个好名字。这是一件连国会议员都不会反对的事情。所以我把它作为我研究活动的“保护伞”。（以上内容摘自[这个知乎文章](#)）

### 8.1.2 基础问题：爬楼梯

首先看动态规划思想的最简单的应用，从中体会动态规划解决问题的基本思想。

**【爬楼梯】**假设你正在爬楼梯。需要爬  $n$  阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶，求有多少种

不同的方法爬到楼顶，要求算法在线性时间内解决这一问题。

现在尝试分析这一问题，有的同学可能会想这是一个计数问题，所以使用中学的排列组合思想即可解决——当然可以，但这里希望讨论算法设计，所以不会讨论这种直白的方法，当然感兴趣的读者应该不难想到如何使用排列组合解决这一问题（通过上 2 级台阶的次数分类是一种办法）。至于算法设计方面，首先会想到曾经学过的回溯和分治思想，回溯法解决这一问题的想法非常简单，或许与暴力枚举区别不大，时间复杂度显然不可能是线性的。如果本题要求列出所有的可能情况，那么回溯是合理的，但只需要计数，因此除非找不到更好的办法，否则不会采用效率低下的回溯法。

分治或许是一个降低复杂度的好方法，若  $n$  为偶数，那么想法可以是先爬前一半的台阶，再爬后一半的台阶，然后两种方法数相乘。然而这并不完整，因为一次可以爬两个台阶，因此第  $n/2$  个台阶不一定是分界线，此时相当于先走了  $n/2 - 1$  个台阶，然后一次跨两个台阶，再走后  $n/2 - 1$  个台阶，因此有

$$f(n) = f(n/2)^2 + f(n/2 - 1)^2,$$

其中  $f(n)$  表示爬  $n$  阶楼梯的方法数。因此偶数的情况分治只需要计算  $f(n/2)$  和  $f(n/2 - 1)$  即可，递归直到 base case，时间复杂度的递推式大致是  $T(n) = 2T(n/2) + O(1)$ ，看起来很美好，因为用上一节课的知识知道这是线性的复杂度，满足要求。然而如果考虑奇数，需要计算  $f(\lceil n/2 \rceil)$ 、 $f(\lfloor n/2 \rfloor)$ 、 $f(\lceil n/2 \rceil - 1)$  和  $f(\lfloor n/2 \rfloor - 1)$ ，这样复杂度递推式就大致变成了  $T(n) = 4T(n/2) + O(1)$ ，这样看来复杂度是  $O(n^2)$  的，显然不是我们想要的。

分治法的失败迫使我们思考新的解法，或许可以对这个问题的结构进行一些观察。可以看为什么这个问题分治法并没有之前学习的那些问题合适：实际上是因为问题对半分了之后还无法解决这一问题，还要递归解决很多近似于对半分的子问题才能将所有问题解决（比如还需要考虑  $n/2 - 1$  的情况）——不像归并排序，对半分之后两边问题解决了，然后只需要解决一个线性合并问题即可。为了算法的效率性，利用子问题的组合来帮助解决最后的问题显然是一条正确的道路，既然对半分不适合，但这个问题的解应该是从小问题的解出发开始逐步得到最后整个问题的解。那么解的最后一步应该是一个好的出发点，因为它可能依赖于前面小问题的解，这样或许可以得到一些新的关于最终解的结构的观察。不难知道爬楼梯的方法中，最后一步有两种可能：

1. 最后一步爬 1 个台阶；
2. 最后一步爬 2 个台阶。

如果能知道这两种情况下爬楼梯的方法数，那么总的爬楼梯的方法数就是把以上两种情况的方法数求和。事实上很容易发现，如果最后一步爬 1 个台阶，那么爬楼梯的方法数就是爬  $n - 1$  阶楼梯的方法数；如果最后一步爬 2 个台阶，那么爬楼梯的方法数就是爬  $n - 2$  阶楼梯的方法数。由此就得到了一个递推关系： $f(n) = f(n - 1) + f(n - 2)$ 。这个递推关系是不是很熟悉？没错，这就是斐波那契数列的定义！于是可以怀着激动的心情，像 PPT 第 2 页那样写出一个一个美好的递归的解法，如果这是 PTA 的一个习题，我想邪恶的出题老师一定会让你感受到运行超时的绝望！为什么呢？事实上递归算法如此慢的原因在于算法模仿了递归。为了计算  $F_N$ ，存在一个对  $F_{N-1}$  和  $F_{N-2}$  的调用。然而，由于  $F_{N-1}$

递归地对  $F_{N-2}$  和  $F_{N-3}$  进行调用，因此存在两个单独的计算  $F_{N-2}$  的调用。如果探试整个算法，那么可以发现， $F_{N-3}$  被计算了 3 次， $F_{N-4}$  计算了 5 次，而  $F_{N-5}$  则是 8 次，等等，因此冗余计算的增长是爆炸性的。

因此需要一个实现上的技术来帮助减少冗余的计算。其实这一想法非常简单，只要用一个数组把已经计算出来的值存起来：首先初始化斐波那契数列的第 0 和 1 项，放在数组下标 0 和 1 的位置，然后基于此可以计算出第 2 项，放在数组下标 2 的位置，然后基于存储的第 1 和 2 项算出第 3 项放在数组下标 3 的位置，以此类推直到算出第  $n$  项。显然这一迭代的算法复杂度是线性的，但因为使用了数组存储，所以使用了额外的空间，所以是一种典型的“空间换时间”的算法（当然其实递归也很耗空间），这一重要的思想称其为“记忆化”，非常形象，因为就是利用额外的空间记住曾经算过的结果，从而避免了重复计算的问题。换句话说，原先递归的方法实际上是一种“自顶向下”的方法，它符合直接思路：最后长度为  $n$  的问题需要长度为  $n-1$  和  $n-2$  的求和，那代码直接利用递归计算递归式显得非常自然。但是重复的计算迫使我们放弃这种自然的想法，转而采用自底向上从小问题逐步迭代构建原问题的解法。

总结一下上面的讨论：在爬楼梯这一简单而经典的问题中，通过对最后一步两种情况的分类，将整个问题的解转化为了两个子问题解的求和，即有一个从子问题的解到原问题的解的递推公式，然后通过记忆化的方式求解递推式。将上述特点抽象出来：一个问题，它的最优解可以表达为一些合适的子问题的最优解的递推关系，则称这一问题具有**最优子结构性质**（因为大问题的最优解可以直接依赖于小问题的最优解）。然后求解这一递推式，通过设置好 base case（这里也用 base case 指代最简单的情况，但注意这时不是递归了），然后通过记忆化的方法，使用迭代算法而非费时的递归算法避免冗余计算，得到一个时间复杂度令人满意的算法，这就是动态规划的基本想法。

更一般的来说，上面的问题只是借用了动态规划的简单思想，接下来的例子才是正式的开始。在开始旅程之前，先给出动态规划的一般范式。动态规划方法通常用来求解最优化问题（optimization problem）。这类问题可以有很多可行解，每个解都有一个值，我们希望寻找具有最优值（最小值或最大值）的解。称这样的解为问题的一个最优解（an optimal solution），而不是最优解（the optimal solution），因为可能有多个解都达到最优值。

通常按如下 4 个步骤来设计一个动态规划算法：

1. 刻画一个最优解的结构特征；
2. 递归地定义最优解的值；
3. 计算最优解的值，通常采用自底向上的方法；
4. 利用计算出的信息构造一个最优解。

步骤 1-3 是动态规划算法求解问题的基础。如果仅仅需要一个最优解的值，而非解本身，可以忽略步骤 4（上面的例子就没有这一步骤，当然上面的例子也不是最优化问题，但接下来马上就可以看到包含步骤 4 的例子）。如果确实要做步骤 4，有时就需要在执行步骤 3 的过程中维护一些额外信息，以便用来构造一个最优解。

更精炼地，动态规划就是为一个具有所谓最优子结构性质（即原问题最优解可以由子问题最优解递推得到）的最优化问题寻找一个子问题到原问题的递推式，然后用记忆化方法求解，最后有时需要构造出这一最优解。接下来将开始正式的旅途，我们将会从易到难，观察一些经典的例子的解法。动态规划是一个经验和 trick 都很重要的算法设计思想，因此见识很多不同风格、不同特点的例子并加以自己的思考是非常重要的。

## 8.2 加权独立集合问题

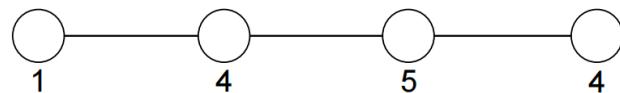
前面的爬楼梯问题只能说是一个简单的热身，引出了动态规划的基本想法，即最优子结构的递推以及记忆化。接下来将正式研究如何将动态规划应用到最优化问题中，展示上面给出的求解一般流程。

### 8.2.1 问题描述

第一个问题称之为加权独立集合问题。可以从一个带故事背景的问题出发：

**【打家劫舍】**你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

将这一问题抽象出来，实际上就是：考虑一个无向图  $G$ ，其上所有点都在一条线上（这种图称其为路径图），每个点都有一个非负权重。称  $G$  的独立集合是指顶点互不相邻的子集（换句话说独立集合不会同时包含一条边上的两个点），然后要求解一个具有最大顶点权重和的独立集合。这一问题称为一维的加权独立集合问题，当然显然有对一般的图的版本，但那一问题之后会知道是很困难的，所以这里只介绍一维的。



上图是一个简单的例子，在这个图中有 8 个独立子集：空集、四个单点集、第 1 和第 3 个点、第 1 和第 4 个点、第 2 和第 4 个点三个两点集。其中最大的独立集合显然是第 2 和第 4 个点构成的集合，其权重和为 8，即小偷的最优选择是偷 2 和 4 两家，最大收益是 8。需要注意的是，题目的假设中每个顶点都有非负权重，所以最优解的顶点应当是越多越好。

### 8.2.2 最优子结构

开始尝试解决这一问题。在使用动态规划之前，当然可以先看看其它方法是否可以。回溯当然可行，但复杂度显然太高；分治法读者可以自己尝试一下，事实上又会落入前面爬楼梯那样子问题纠缠的情况，因此也并不是很合适。所以尝试动态规划，寻找最优子结构和递推关系。

为了找到这一递推关系，可以做一个简单的自顶向下的思维实验（这在之后的问题中经常用到）：如果你拿到一个路径图，那么你对于这一问题的解首先会想什么？你应该想知道最后一个点是不是在最优解中，因为只有知道了这个点在不在才能进一步向前继续构造出整个问题的解！更具体地说，设  $G = (V, E)$  表示  $n$  个顶点的路径图，具有边  $(v_1, v_2), \dots, (v_{n-1}, v_n)$ ，并且每个顶点  $v_i$  都有非负权重  $w_i$ 。根据前面的分析，应当考虑  $v_n$  是否在最优解  $S$  中，显然可以分为如下两种情况讨论：

1. 如果  $v_n$  不在最优解  $S$  中，那么  $S$  实际上可以看成前  $n - 1$  个点构成的路径图  $G_{n-1}$  组成的子问题的一个可行解。事实上  $S$  一定是  $n - 1$  个点的路径图的最大加权独立子集（即最优解），否则如果能找到更优的解  $S^*$  是  $G_{n-1}$  的最优加权独立集合，那么  $S^*$  作为子问题的最优解，是原问题的可行解，因此一定不会优于原问题的最优解  $S$ ，这就产生了矛盾：前面假设  $S^*$  优于  $S$ ，现在推出  $S^*$  不会优于  $S$ ；
2. 如果  $v_n$  在最优解  $S$  中，那么由于是独立集合问题，那么  $v_{n-1}$  就不能在最优解中，因为  $v_{n-1}$  和  $v_n$  相邻。所以类似于第一种情况的分析， $S - \{v_n\}$  一定是前  $n - 2$  个点构成的路径图  $G_{n-2}$  的最优解，所以整个问题的最优解就是  $G_{n-2}$  的最优解加上  $v_n$ 。

由此，整个问题的最优解依赖于前  $n - 1$  和  $n - 2$  个点构成的子图的最优解，或者说，这两个子问题的最优是构成原问题最优的基础，这就是这一问题对应的最优子结构性质。设前  $i$  个点的最优加权独立集合的权重之和为  $W_i$ ，可以写出递推关系：

$$W_n = \max\{W_{n-1}, W_{n-2} + w_n\},$$

这里是最后一步的递推关系，事实上可以不断利用这一递推关系自顶向下向前构建出整个问题的解，即有更一般的表达

$$W_i = \max\{W_{i-1}, W_{i-2} + w_i\},$$

其中  $i = 2, 3, \dots, n$ ,  $W_0 = 0$ 。于是，到目前为止完成了动态规划四步框架的前两步，接下来需要写代码实现这一算法。

利用之前介绍的记忆化方法，从 base case 出发，自底向上构建出整个问题的解，否则自顶向下递归会重复计算太多的子问题。很显然需要一个循环，从开始的结点遍历到最后，用递推式逐步向后计算出截止于每个点的最优解，最后得到整个问题的解。但需要设置好 base case，否则将无从开始我们的迭代。事实上 base case 就是最 trivial 的情况，即 0 个点和 1 个点的情况，有了这两种情况，接下来利用递推式就可以算出所有子问题以及原问题的解了，伪代码如下图。很显然，这一算法的时间复杂度是  $O(n)$ ，是一个非常高效的算法。

```

 $A := \text{length-}(n+1) \text{ array} \quad // \text{subproblem solutions}$ 
 $A[0] := 0 \quad // \text{base case } \#1$ 
 $A[1] := w_1 \quad // \text{base case } \#2$ 
for  $i = 2$  to  $n$  do
     $// \text{use recurrence from Corollary 16.2}$ 
     $A[i] := \max\{\underbrace{A[i-1]}_{\text{Case 1}}, \underbrace{A[i-2] + w_i}_{\text{Case 2}}\}$ 
return  $A[n] \quad // \text{solution to largest subproblem}$ 

```

### 8.2.3 解的重构

现在还剩下动态规划框架的最后一步，即有时候可能不仅需要最优解的值，还需要最优解本身包含了哪些点，这就需要解的重构。事实上这一问题想要重构出解非常简单，只需要观察上面伪代码的数组  $A$  留下的线索，结合递推式即可。从数组  $A$  中，首先可以看出最后一个点是否在最优解中，因为

$$W_n = \max\{W_{n-1}, W_{n-2} + w_n\},$$

如果  $W_n = W_{n-1}$ ，那么最后一个点不在最优解中，此时回退一格，继续重建图  $G_{n-1}$  的最优解；否则最后一个点在最优解中，回退两格重建图  $G_{n-2}$  的最优解。具体伪代码实现中还有 base case 的考量，读者可以自己思考然后与下图中的参考答案比对：

```

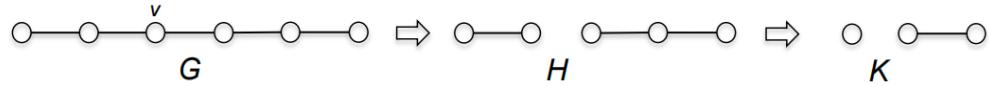
 $S := \emptyset \quad // \text{vertices in an MWIS}$ 
 $i := n$ 
while  $i \geq 2$  do
    if  $A[i-1] \geq A[i-2] + w_i$  then  $// \text{Case 1 wins}$ 
         $i := i - 1 \quad // \text{exclude } v_i$ 
    else  $\quad // \text{Case 2 wins}$ 
         $S := S \cup \{v_i\} \quad // \text{include } v_i$ 
         $i := i - 2 \quad // \text{exclude } v_{i-1}$ 
    if  $i = 1$  then  $\quad // \text{base case } \#2$ 
         $S := S \cup \{v_1\}$ 
return  $S$ 

```

总而言之，这是一个简单而经典的动态规划的例子，这一例子将之前给出的框架完整地展示了一遍。接下来是一些思考题，检验读者是否至少理解了本题的解法：

**【思考：】**本问题中每个顶点都有非负的权重，假设允许顶点的权重为负数，那么上面的解法是否仍然适用？如果不适用，你能否构造出一个不适用的图并给出一个新的解法？

**【思考：】**这个问题规划了一种方法，该方法解决比路径图更为复杂的图的加权独立集合问题。考虑一个任意的无向图  $G = (V, S)$ ，所有顶点的权重均不为负，并且任意一个顶点  $v \in V$  具有权重  $w_v$ 。通过从  $H$  中删除  $v$  的相邻顶点以及相关联的边得到  $K$ ，如下图所示：



设  $W_G$ 、 $W_H$  和  $W_K$  分别表示  $G$ 、 $H$  和  $K$  的最大加权独立集合的总权重，并考虑下面这个公式：

$$W_G = \max\{W_H, W_K + w_v\},$$

下面哪些说法是正确的（选择所有正确的答案）：

1. 这个公式对于路径图并不总是正确的；
2. 这个公式对于路径图总是正确的，但对于树并不总是正确的；
3. 这个公式对于树总是正确的，但对于任意的图并不总是正确的；
4. 这个公式对于任意的图都是正确的；
5. 这个公式对于树的加权独立集合问题可以产生一种线性时间的算法；
6. 这个公式对于任意图的加权独立集合问题可以产生一个线性时间的算法。

**【思考：】**（最大子序列和问题）同上一讲分治的最大子序列和问题，要求写出递推关系，并且算法要给出这一最大子序列。

**【思考：】**（零钱兑换问题）给你一个整数数组 `coins`，表示不同面额的硬币；以及一个整数 `amount`，表示总金额。计算并返回可以凑成总金额所需的最少的硬币个数，以及最少硬币个数对应的兑换方式。如果没有任何一种硬币组合能组成总金额，返回 `-1`。你可以认为每种硬币的数量是无限的。

例如：`coins = [1, 2, 5]`, `amount = 11`, 则答案为 3, 最优兑换为  $11 = 5 + 5 + 1$ 。

## 8.3 背包问题

### 8.3.1 问题描述

背包问题是一个最经典的动态规划问题，这里首先介绍最基础的背包问题，即 0-1 背包问题。这一问题的描述如下：有  $n$  个物品，每个物品的重量为  $s_i$ ，价值为  $v_i$ ，有一个容量为  $C$  的背包，希望找到一个最优的装载方案，使得背包中的物品总价值最大。这一问题的特点是每个物品你要么不放进包里，要么完整的 1 个放进去，因此称为 0-1 背包问题。

一个简单的例子如下：假设有 4 个物品，重量分别为  $s = [2, 1, 3, 2]$ ，价值分别为  $v = [12, 10, 20, 15]$ ，背包容量为  $C = 5$ ，那么最优的装载方案是将第 1、2、4 个物品放进背包，总价值为  $12 + 10 + 15 = 37$ 。

### 8.3.2 最优子结构：一维似乎不再适用

完全可以仿照着前面讨论的两个问题的解法来解决背包问题。首先考虑最后一个物品是否在最优解中，然后根据这一情况划分子问题，然后递推：

1. 如果第  $n$  个物品不在最优解  $S$  中，即最优方案排除了最后一件物品，因此它可以看成仅由前  $n-1$  个物品组成的一种可行解决方案，并且  $S$  必定是这个子问题的最优解，否则可以找到一个更优的解  $S^*$ ， $S^*$  作为子问题的最优解，是原问题的可行解，因此一定不会优于原问题的最优解  $S$ ，这就产生了矛盾（事实上理由和前面加权独立集合问题一样）；
2. 如果第  $n$  个物品在最优解  $S$  中，这种情况只有在  $s_n \leq C$  时才有意义。类似于加权独立集合问题，我们希望  $S - \{n\}$  是前  $n-1$  个物品组成的子问题的最优解，但这显然是错误的！如果  $S - \{n\}$  就已经把  $W$  几乎占满，那最后  $n$  根本就放不进来了。因此需要对子问题的设置略做调整： $S - \{n\}$  应当是前  $n$  个物品在背包容量为  $C - s_n$  的情况下的最优解！因为我们知道  $n$  在解中，那么除去  $n$  之外的其它解的重量之和最多就是  $C - s_n$ ，因此  $S - \{n\}$  应当是前  $n$  个物品在背包容量为  $C - s_n$  的情况下的可行解，并且应当是最优解，否则又可以同前面情况的讨论构造出矛盾。

因此根据上述讨论，此时的子问题不再仅仅是只由单个参数（即是第几个输入）控制，还由背包容量控制，因此需要一个二维的数组来存储子问题的解。设  $V_{i,c}$  表示总重量不超过  $c$  的前  $i$  件物品组成的子集的最大总价值，那么可以得到如下递推关系：

$$V_{i,c} = \begin{cases} V_{i-1,c}, & s_i > c; \\ \max\{V_{i-1,c}, V_{i-1,c-s_i} + v_i\}, & s_i \leq c. \end{cases}$$

所以分析完了最优子结构（原问题和子问题最优解的依赖关系），写出了递推式，接下来就是写代码实现这一算法，首先需要设置好 base case，然后利用递推式自底向上构建出整个问题的解。根据递推式，只要  $i-1$  的所有情况都确定了，递推就可以往  $i$  推进，所以 base case 需要对  $i$  的最小情况（即  $i=0$ ）设置好就可以了，而  $i=0$  时  $V_{i,c}$  显然是 0，因为根本没有物品可以放。伪代码如下页图所示。一个显然的事情是，这一算法的时间复杂度是  $O(nC)$ 。

```

// subproblem solutions (indexed from 0)
A :=  $(n + 1) \times (C + 1)$  two-dimensional array
// base case ( $i = 0$ )
for  $c = 0$  to  $C$  do
     $A[0][c] = 0$ 
// systematically solve all subproblems
for  $i = 1$  to  $n$  do
    for  $c = 0$  to  $C$  do
        // use recurrence from Corollary 16.5
        if  $s_i > c$  then
             $A[i][c] := A[i - 1][c]$ 
        else
             $A[i][c] :=$ 
             $\max\{\underbrace{A[i - 1][c]}_{\text{Case 1}}, \underbrace{A[i - 1][c - s_i] + v_i}_{\text{Case 2}}\}$ 
return  $A[n][C]$  // solution to largest subproblem

```

### 8.3.3 解的重构

和前面的加权独立集合问题一样，还是根据递推式从数组中挖掘出最优解，建议读者自己思考，下面给出参考代码：

```

 $S := \emptyset$            // items in an optimal solution
 $c := C$                  // remaining capacity
for  $i = n$  downto 1 do
    if  $s_i \leq c$  and  $A[i - 1][c - s_i] + v_i \geq A[i - 1][c]$  then
         $S := S \cup \{i\}$       // Case 2 wins, include  $i$ 
         $c := c - s_i$         // reserve space for it
    // else skip  $i$ , capacity stays the same
return  $S$ 

```

### 8.3.4 降维

事实上，对于这种多维的背包问题，经常可以考虑这样一种优化策略：即有没有可能数组的维数可以下降，例如这里是不是一维数组就能帮我们做好所有的事情。事实上如果不考虑解的重建是完全可以的（考虑解的重建可以参考后面矩阵乘法的方法，那样可能又要维护新的二维数组），因为事实上第  $i$  个物品对应的所有容量的解只依赖于第  $i - 1$  个物品的所有容量的解，因此事实上数组完全可以不用  $i$  这一维，每次循环对某个  $i - 1$ ，算出所有容量的解，保存在只有容量这一个 index 的数组中，然后再算下一个  $i$  的所有容量的解覆盖掉前面的解。这样看起来很完美，但有一点需要注意，就是第  $i$  轮数组的

$A[c]$  会依赖于更小的  $c$  在  $i - 1$  轮的解，所以如果  $c$  的遍历还是从小到大的，那么就可能提前把需要的  $i - 1$  轮的解覆盖，所以对  $c$  应当从大到小遍历，这样就不会出现错误的覆盖。

【思考:】（背包问题的变体）背包问题有大量有趣的变体，读者可以参考网上的资料进行学习，此处限于体量和时间无法展开讲解。

## 8.4 矩阵乘法的计算顺序

前面的问题似乎带给我们一些错觉：似乎动态规划就是看输入的最后一个元素是否在最优解中，然后就有两种最优解的构成形式，对应到两个子问题递推式取最优即可。事实上显然不是所有问题都这么简单的，下面这一问题将会展示子问题可能有很多种的情况。

### 8.4.1 问题描述

PPT 第 4 页描述了这一问题，简单而言就是要计算一系列矩阵链相乘的顺序，使得总的计算代价最小。PPT 上给出了一个非常极端的例子，表明最优和最差的计算顺序的差距是巨大的。

### 8.4.2 最优子结构：比较的子问题不一定只有两种

考虑矩阵链相乘  $M_1 M_2 \cdots M_n$ ，每个矩阵的大小为  $r_{i-1} \times r_i$ 。记

$$M_{ij} = M_i M_{i+1} \cdots M_j,$$

显然前面的三个问题给我们了不少的经验：应当考虑最后一个矩阵，然后依据这一矩阵可能的相乘方式划分最优解的可能性。很简单的，第一种情况对应的计算方式是  $M_{1,n-1} M_n$ ，即先用算前  $n - 1$  个矩阵相乘的最优方式计算前  $n - 1$  个矩阵的乘积，最后与  $M_n$  相乘；第二种情况的计算方式是  $M_{1,n-2} M_{n-1,n}$ ，即先用算前  $n - 2$  个矩阵相乘的最优方式计算前  $n - 2$  个矩阵的乘积，最后与  $M_{n-1,n}$  的结果相乘。这是根据前面的问题的经验得到的，看起来很完美，但事实上真的只有这两种情况吗？显然不是的！注意，请回到分类的根源标准：此时分类的依据是  $M_n$  的计算方式，那么对于如下乘积形式：

$$M_{1i} M_{i+1,n-1} M_n,$$

只要是不同的  $i$ ，事实上都可能是不同的乘积方式（对于  $M_n$  而言，不同的  $M_{i+1,n-1}$  与其相乘时如果各个  $M_{i+1,n-1}$  的行数不一样，那就会带来不同的复杂度），因此情况远远没有我们想的那么简单。

所以不应当简单利用最后一个矩阵的相乘方式进行讨论，而是思考其它的子问题划分方式。应当停下来回到最开始讨论动态规划的想法，而非局限于一些过拟合的错误思路：进行自顶向下的思维实验，当你面对一个完全没有任何划分的矩阵链乘法时，你的一个很直接的想法就应当是首先确定这第一刀的

划分应该出现在哪里——正如面对一个加权独立集合问题时，需要首先知道最后一个点是否在最优解中一样，否则将无法自顶向下推进得到最优解的剩余部分是什么样的。于是考虑所有的

$$M_{1i} M_{i+1,n}, \quad 1 \leq i \leq n-1,$$

分解，那么整个问题的最优解就是所有上式中  $n-1$  个分解的最优的那个对应的情况。因为是自顶向下的思考，那么在之前的步骤中应当已经得到了所有的  $M_{1i}$  和  $M_{i+1,n}$  的最优乘法顺序，因此对于每一种分解，总的最长时间应当等于  $M_{1i}$  和  $M_{i+1,n}$  的最长时间求和加上  $M_{1i}$  和  $M_{i+1,n}$  相乘所需的时间（即  $r_0 r_i r_n$ ），然后可以得到这  $n-1$  种情况的最优时间，接下来只需取出它们中的最优值就是整个问题的最优解。或许说起来有点难懂，看下面这一递推式就会非常清楚：

$$m_{1n} = \min\{m_{1i} + m_{i+1,n} + r_0 r_i r_n\},$$

其中  $m_{ij}$  表示  $M_{ij}$  的最优乘法时间。更一般地有

$$m_{ij} = \begin{cases} 0, & i = j; \\ \min_{i \leq k < j} \{m_{ik} + m_{k+1,j} + r_{i-1} r_k r_j\}, & i < j. \end{cases}$$

其中第一条是接下来代码中不可或缺的 base case。从递推式中很容易看出这一问题的最优子结构，实际上就是对于  $M_{ij}$ ，其计算的最后一定是  $M_{ik}$  和  $M_{k+1,j}$  相乘，因此所有分解的情况构成了需要比较的全体情况，然后必须取出每个分解的最优情况（否则最后可能得到的不是最优解），然后取出所有分解的最优情况中最好的就是大问题的最优解。因此  $M_{ij}$  的最优乘法顺序依赖于子问题  $M_{ik}$  和  $M_{k+1,j}$  的最优值，这就是这一问题的最优子结构。

和之前的问题一致，这一递推式很适合于递归，但冗余计算很多，因此自底向上记忆化。这里的记忆化和之前也不太一样，因为思考递推式并不是以前那样按输入顺序每个输入是否是子问题解的一部分，那时的代码可以同理可以做一个遍历，根据输入顺序逐个判断当前输入是否应当是某个子问题的解的一部分。读者可以思考，如果按顺序遍历我们的代码一定会出现混乱，因为你会发现你还要依赖很多你当前并没算出来的情况。但是只要你想到这一点就可以：如果短的矩阵链最优解被构造，那么可以基于此计算更长的矩阵链的最优解，即应当按照矩阵链的长度从小到大的顺序计算最优解，即最外层循环应当是按  $M_{ij}$  中  $i - j$  的大小递增的顺序。这样 base case 是矩阵链长度为 1 ( $i = j$ ) 的全 0，然后是长度为 2 ( $j - i = 1$ ) 的情况，实际上也只有一种乘法方式，所以也是 trivial 的，接下来长度为 3 的情况完全可以基于长度为 1 和 2 的情况得到，然后逐步迭代到最后的长度为  $n$  的情况。这样就得到了一个非常简单的动态规划算法，其时间复杂度是  $O(n^3)$ ，这里同样建议读者自己写一遍代码（可以参考 PPT 第六页，体会 base case 开始自底向上构建解的想法），可以加深理解。

### 8.4.3 解的重构

这一问题重构解的方法如果用和前面的问题一样简单直接的挖掘方法显然会比较耗时间，因为每次要比较很多种可能的情况才能重构出解，所以需要做一些优化，优化的方法也非常自然，实际上关键

就是用一个二维数组记住每个子问题

$$m_{ij} = \min_{i \leq k < j} \{m_{ik} + m_{k+1,j} + r_{i-1}r_k r_j\}$$

对应的最优的分点  $k$ , 然后当算出总问题  $m_{1n}$  的最优解及其对应的最优分点后, 再找左右两半子问题的最优分点, 以此类推, 用中序遍历的思想将分点输出即可, 只需线性时间。

**【思考:】**(钢条切割问题) Serling 公司购买长钢条, 将其切割为短钢条出售。切割工序本身没有成本支出。公司管理层希望知道最佳的切割方案。假定已知 Serling 公司出售一段长度为  $i$  英寸的钢条的价格为  $p_i$  ( $i = 1, 2, \dots, n$ ), 单位为美元)。钢条的长度均为整英寸。下图给出了一个价格表的样例。

长度 $i$	1	2	3	4	5	6	7	8	9	10
价格 $p_i$	1	5	8	9	10	17	17	20	24	30

钢条切割问题是这样的: 给定一段长度为  $n$  英寸的钢条和一个价格表  $p_i$  ( $i = 1, 2, \dots, n$ ), 求切割钢条方案, 使得销售收益  $r_n$  最大。注意, 如果长度为  $n$  英寸的钢条的价格  $p_n$  足够大, 最优解可能就是完全不需要切割。

**【思考:】**(完全平方数的和) 给你一个整数  $n$ , 返回和为  $n$  的完全平方数的最少数量。例如  $n = 13$ , 则  $n$  至少需要写成两个完全平方数相加的形式, 即  $n = 4 + 9$ 。

## 8.5 最长公共子序列问题

### 8.5.1 问题描述

### 8.5.2 序列对齐问题

### 8.5.3 最优子结构: 字符串问题的特点

### 8.5.4 解的重构

**【思考:】**(回文字符串) 如果字符串的反序与原始字符串相同, 则该字符串称为回文字符串。现在给你一个字符串  $s$ , 找到  $s$  中最长的回文子串。

**【思考:】** RNA

## 8.6 最优二叉搜索树

### 8.6.1 问题描述

这一问题给考虑下列输入：给定一列单词  $w_1, w_2, \dots, w_n$ （它们的顺序已经按字典序排好）和它们出现的固定概率  $p_1, p_2, \dots, p_n$ 。问题是要求以一种方法在一棵二叉查找树中安放这些单词使得总的期望存取时间最小。在一棵二叉查找树中，访问深度  $d$  处的一个元素所需要的比较次数是  $d + 1$ ，因此如果  $w_i$  被放在深度  $d_i$  上，就要将  $\sum_{i=1}^N p_i(1 + d_i)$  极小化。

### 8.6.2 算法描述

这一问题最直接的想法便是贪心或者使用平衡树，但 PPT 第 7 页中的例子告诉我们它们都是不一定可行的。一种也很自然的想法是分治法，可以将问题分为两个子问题来解决，然而分治法有一个很大的问题：这两半的子问题应该如何划分？如果提前知道最优的根结点是哪个单词，那很容易进行问题的划分，然而可惜的是我们没有预知未来的能力。但从这一特点能回忆到曾经熟悉的问题：在矩阵链乘法中，需要知道第一个分点在哪里才能构建后续的解，在权独立集合问题中，需要知道最后一个点是否在最优解中才能推出整个解。这一问题也是一样的，需要知道最优的根结点在哪里才能划分问题，然后可以得到两个子问题，然后我们可以得到最优解。并且知道最初给定的输入已经按字典序排好，所以如果知道根结点的最优解为  $w_k$ ，那么在整个问题的最优解中，其左子树必定是  $w_1, \dots, w_{k-1}$  对应的最优二叉搜索树，右子树是  $w_{k+1}, \dots, w_n$  对应的最优二叉搜索树。因此可以得到一个递推式（符号含义见 PPT 第 8 页）：

$$c_{ij} = \sum_{k=i}^j p_k + \min_{i \leq k \leq j} \{c_{i,k-1} + c_{k+1,j}\}, \quad i \leq j.$$

注意第一项的意义在于，因为根结点的存在，导致所有  $i$  到  $j$  的单词都加深一层，是两个子问题结合起来增加的代价。第二项则是前面讨论的左右子树子问题的情况。此时的 base case 没有写在递推式中，但读者仔细观察应当不难看出是  $j = i - 1$  时一定为 0（记住 base case 就是递推式的最小出发点，是最 trivial 的情况），当然也可以把  $i = j$  的情况，即只有一个点的情况作为 base case，然后对上面的递推式略作修改。基于此，可以得到如下伪代码：

图中为代码的复杂度显然是立方级别的：尽管只写了两个循环，但显然最后取  $\min$  的步骤也是要遍历的——这与前面的矩阵乘法问题完全一致。总结而言，这一问题的最优子结构也是不言自明的，即最优解依赖于每个分割得到的子树的的最优解。这一问题的解的重构也是非常简单的，只需记住每个子问题的最优分点即可，然后回溯建树即可，和矩阵乘法顺序问题也是一致的，只需线性时间。

或许这一问题从很多方面看来都与矩阵乘法顺序问题一致，但事实上可以为这一问题构建一个平方级别的算法，读者可以跟随下一思考题来实现这一解法：

```

// subproblems (i indexed from 1, j from 0)
A := (n + 1) × (n + 1) two-dimensional array
// base cases (i = j + 1)
for i = 1 to n + 1 do
    A[i][i - 1] := 0
// systematically solve all subproblems (i ≤ j)
for s = 0 to n - 1 do      // s=subproblem size-1
    for i = 1 to n - s do   // i + s plays role of j
        // use recurrence from Corollary 17.5
        A[i][i + s] :=
             $\sum_{k=i}^{i+s} p_k + \min_{r=i}^{i+s} \underbrace{\{A[i][r-1] + A[r+1][i+s]\}}_{\text{Case } r}$ 
return A[1][n] // solution to largest subproblem

```

【思考:】设

$$c_{ij} = \begin{cases} 0, & i = j; \\ w_{ij} + \min_{i < k \leq j} \{c_{i,k-1} + c_{kj}\}, & i < j. \end{cases}$$

设  $W$  满足四边形不等式, 即对所有的  $i \leq i' \leq j \leq j'$ , 有

$$w_{ij} + w_{i'j'} \leq w_{ij'} + w_{i'j}.$$

进一步假设  $W$  是单调的: 如果  $i \leq i'$  且  $j \leq j'$ , 则  $w_{ij} \leq w_{i'j'}$ 。

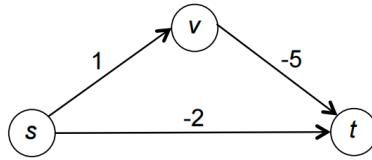
1. 证明:  $C$  满足四边形不等式;
2. 令  $R_{ij}$  是使  $C_{i,k-1} + C_{kj}$  最小值的最大的  $k$  (即出现平局的情况选择最大的  $k$ ), 证明:

$$R_{ij} \leq R_{i,j+1} \leq R_{i+1,j+1};$$

3. 证明:  $R$  沿着每一行和列是非减的;
4. 证明:  $C$  中所有项的计算可以在  $O(n^2)$  时间内完成;
5. 用上述结论描述一个最优二叉搜索树的  $O(n^2)$  时间算法。

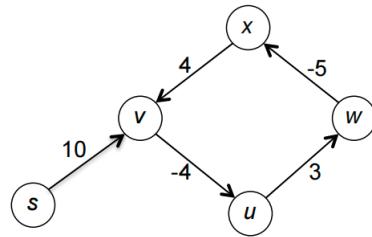
## 8.7 最短路问题再讨论

相信各位在 FDS 中已经对经典的单源 (即只考虑一个顶点出发到任意其它顶点的距离) 无负边最短路问题算法——Dijkstra 算法非常熟悉, 但这一问题对于带负边的问题可能给不出正确的解, 如下面的例子:



如果用 Dijkstra 算法求解这一问题，会得到最短路径长为  $-2$ ，但事实上显然是  $-4$ ，为什么 Dijkstra 算法在有负边时会失效则要回顾这一贪心算法的正确性证明，具体的读者可以回顾离散或 FDS 中学习过的证明，总体原因是正确性证明的确是依赖于边长非负这一条件才能实现的。

负边带来的更痛苦的事情是可能出现负环：



这样即使有一个能求出带负边的最短路的算法，但因为事实上的最短路应当在负环中无限循环直至负无穷，所以算法也会失效。当然有一种面对负环的解决方案，即计算只考虑无环路最短路径，但在负环存在时这一问题是 NP 困难的（之后的章节会介绍，总之是一类非常难解的问题）。看起来我们无路可走了，但还可以考虑另一种修订版本，即希望这一算法要么能计算出正确的最短路径长度，要么能正确地判断出输入图中包含了负环（只需要判断即可）。这就带来了下面的 Bellman-Ford 算法。

**【注：本节的讲解很多内容参考拉夫加登的《算法详解》，感兴趣的同学可以读一下原书】**

### 8.7.1 Bellman-Ford 算法

事实上大家在 FDS 中就已经见过这一算法（可以回过头去翻一翻 PPT），只是当时我相信老师并不会说这是一个动态规划算法。那么现在用动态规划的思想来重新设计和理解这一算法。

既然是动态规划，就需要找到最优子结构。但是对于一般的图而言，因为具有线性结构，所以定义子结构并不是很容易的事情（所以加权独立集合考虑的是路径图，这样点就有线性的先后顺序，子结构很好定义）。于是需要考虑一些特别的子问题、子结构定义方式。在 Bellman-Ford 算法中，子问题被巧妙地定义为：对于每个顶点  $v$ ，考虑从源点  $s$  到  $v$  的所有路径中，最多经过  $k$  条边的路径的最短路径长度。为什么这一子问题是合理的呢？来看递推关系的推导就知道了，即能否用最多经过  $k-1$  条边的路径的最短路径长度推到最多经过  $k$  条边的路径的最短路径（记为  $P$ ）长度。事实上完全可以，并且显然地，递推依赖于如下两种情况取最小值：

1.  $P$  还是只有  $k-1$  条甚至更少的边，那么直接继承子问题的解即可；

2.  $P$  有  $k$  条边, 设  $P'$  为  $P$  中前  $k-1$  条边, 并且  $(w, v)$  是其最后一次的跳跃, 那么因为  $P$  是  $s$  到  $v$  的  $k$  条边的最短路径, 所以  $P'$  也是  $s$  到  $w$  的  $k-1$  条边的最短路径, 否则能取到一个更短的  $P''$  从  $s$  到  $w$ , 那  $P''$  接上  $(w, v)$  是比  $P$  更短的路径, 显然矛盾。因此递推时只需要考虑所有能一步到达  $v$  的顶点  $w_1, \dots, w_m$ , 然后取  $s$  到这些  $w_i (i = 1, \dots, m)$  的最短路长度, 再加上  $(w_i, v)$  的长度得到  $m$  条  $s$  到  $v$  的路径长度, 然后取出这些长度的最小值即可。

记  $D^k[s][v]$  为从  $s$  到  $v$  的最多经过  $k$  条边的最短路径长度, 整个图为  $G = (V, E)$ , 如果  $a$  到  $b$  有一条边, 那么这条边的长度记为  $l_{ab}$ , 那么根据上面的讨论有递推式:

$$D^k[s][v] = \min \begin{cases} D^{k-1}[s][v], & \text{case 1;} \\ \min_{(w,v) \in E} \{D^{k-1}[s][w] + l_{wv}\}, & \text{case 2.} \end{cases}$$

以上递推式的正确性来源于考虑到了所有长度小于等于  $k+1$  的  $s$  到  $v$  的路径的情况, 最优子结构体现在长度至多为  $k$  的路径的最短路径长度一定是依赖于长度至多为  $k-1$  的路径的最短路径长度的。这是最短路问题的一个非常重要的性质 (即最短路径的子路径也是对应两点间的最短路径), 通过定义如上的子问题  $D^k[s][v]$ , 恰好利用了这一重要性质, 找到了合适的最优子结构, 十分精妙。

似乎我们的工作已经做完了? 先别高兴太早, 还需要检测负环! 可以看上面的递推式, 事实上现在还没有讨论  $k$  上升到多少的时候停止递推, 如果有负环, 事实上上面的递推式可以无限递推下去, 这显然不是我们想要的。因此现在考虑, 上面的算法最多到哪一步停止, 停止时是否可以判断出有无负环。事实上有下一重要引理:

**引理 8.1** 设起点为  $s$ , 如果对于某个  $k \geq 0$ , 对于所有目标顶点  $v$  都有  $D^k[s][v] = D^{k+1}[s][v]$ , 那么

1. 对于每个  $i \geq k$  和任意的目标顶点  $v$ ,  $D^i[s][v] = D^k[s][v]$  都成立;
2. 对于每个目标顶点  $v$ ,  $D^k[s][v]$  就是  $s$  到  $v$  的最短路径长度。

**证明:** 引理第一条, 只需看清前面的递推式即可。对于所有目标顶点  $v$  都有  $D^k[s][v] = D^{k+1}[s][v]$ , 那么当考虑  $D^{k+2}[s][v]$  时, 递推式面对的输入和  $D^{k+1}[s][v]$  完全一致! 所以显然  $D^{k+2}[s][v] = D^{k+1}[s][v]$ , 这样递推下去, 就得到了第一条的结论。

对于第二条, 用反证法非常显然, 如  $D^k[s][v]$  不是  $s$  到  $v$  的最短路径长度, 但按定义这是  $k$  步之内的最短路, 那么只能是存在一条更长的路径比  $D^k[s][v]$  更短。但根据引理第一条, 任意长度大于等于  $k$  的  $s$  到  $v$  的路径的最短路径长度都是  $D^k[s][v]$ , 显然矛盾。 ■

基于这一引理可以得到如下的判断是否包含负环的定理:

**定理 8.2** 输入图  $G$  不包含负环当且仅当  $D^n[s][v] = D^{n+1}[s][v]$  对所有的目标顶点  $v$  都成立。其中  $n$  是图  $G$  的顶点数。

**证明:** 若  $D^n[s][v] = D^{n+1}[s][v]$  对所有的目标顶点  $v$  都成立, 根据引理知道对于每个  $i \geq n$  和任意的目标顶点  $v$ ,  $D^i[s][v] = D^n[s][v]$  都成立, 如果存在负环, 那么显然  $i$  增大的时候某些顶点的最短路径长度会减小, 矛盾!

反之, 如果  $G$  不包含负环, 那么从  $s$  出发到一个不同的点  $v$ , 最多也就是把所有顶点都遍历了一遍, 所以最短路径最长也只能是  $n - 1$  (否则更长就有环, 但此时环都是非负的)。换句话说, 把边预算从  $n - 1$  上升到  $n$  对所有顶点的结果一定是没有影响的。 ■

接下来是代码实现, 因为是单源最短路, 所以  $s$  无需在数组索引中, 数组形式为  $A[i][v]$ , 表示从  $s$  到  $v$  的边数最多为  $i$  的最短路长度这一子问题。下面设置 base case, 很显然此时 base case 是  $i = 0$  的情况, 此时只有  $s$  到  $s$  最短路就是 0, 其余都设置为无穷大, 然后就可以用递推式进行计算了。如果到了某一步发现对于所有的  $v$  都有  $A[i][v] = A[i - 1][v]$ , 那么根据引理第二条可知所有的最短路长度就出来了, 如果知道  $n$  步后都无法满足这一点, 根据定理则存在负环 (因此需要维护一个 flag 表明是否达到稳定状态)。如果需要重构最短路径, 可以存下每次取 min 的时候距离最小的那个点, 最后做个 traceback 即可, 并且最短路径的子路径也是最短路径, 所以这一重建过程遍历一次就可以得到所有顶点的最短路径了。

下面分析一下 Bellman-Ford 算法的时间复杂度, 设输入图的顶点数为  $n$ , 边数为  $m$ , 可以大致知道有两层循环, 其中第一层  $i$  最多需要遍历  $n$  个值, 第二层  $n$  个顶点也需要全部遍历, 然后第二层循环内需要遍历对应的顶点  $v$  的所有  $(w, v)$ , 于是实际上整个第二层循环计算、比较了每个顶点的入度之和这个层级的量, 而所有顶点入度之和在有向图中数量就是边数, 所以总的复杂度就是  $O(mn)$ 。

Bellman-Ford 算法应用非常广泛, 事实上早期 Internet 路由协议 RIP 就是使用的这一算法。为什么使用 Bellman-Ford 算法呢? 因为如果要考虑一个新的互联网中节点  $w$  和路由器的最短路, 那么只需要知道和  $w$  直接相连的顶点的最短路就可以直接利用递推式推导即可, 这就意味着 Bellman-Ford 算法即使对于像 Internet 这样规模的网络图也是可以实现的, 每台计算机只需要与直接邻居通信、只执行本地计算即可得知需要的一切信息, 而 Dijkstra 算法每台主机还要计算一个相当大的 Internet 规模的网络图的最短路, 显然是不合适的。

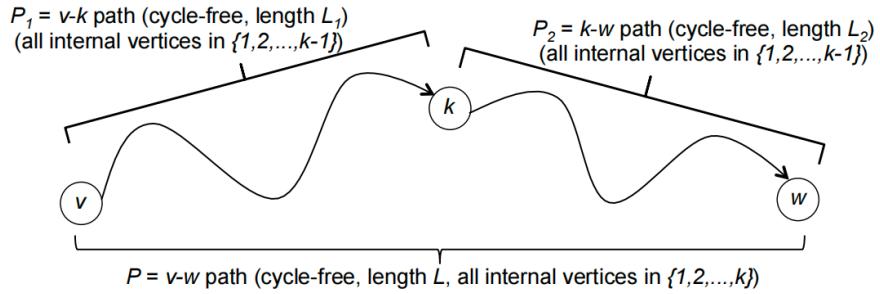
### 8.7.2 Floyd-Warshall 算法

在考虑了单源、找出最短路或负环的算法后, 最后一种推广是考虑所有点对之间的最短路, 也就不是单源, 每个点都可能是源, 并且有负环也需要指出。最简单的想法就是机械地调用  $n$  次 Bellman-Ford 算法, 但这样复杂度就是  $O(n^2m)$ , 在稠密图上是四次方级别的, 有点不可接受, 所以考虑尝试新的动态规划思想来重新解决这一问题。

Floyd-Warshall 算法定义的子问题更加有趣 (图上的算法总是让人觉得很巧妙)。首先需要给图上的每个顶点一个编号:  $1, 2, \dots, n$ , 保持记号  $D^k[i][j]$ , 但现在它的含义变成了: 从点  $i$  出发到  $j$  的, 只使用  $1, 2, \dots, k$  作为内部顶点的 (内部顶点就是除起点和终点外的其它点, 即从  $i$  到  $j$  中间只能经过这些点), 不包含环的最短路径长度。

这一子问题的定义显然导致了以下递推的想法：当我们要从  $D^{k-1}[i][j]$  推到  $D^k[i][j]$  对应的路径  $P$  时，显然又是老调重弹的分类方法：要么  $k$  在  $P$  中，要么不在：

1. 若  $k$  不是  $P$  的内部顶点，那么最好的解就是继承  $D^{k-1}[i][j]$ ；
2. 若  $k$  是  $P$  的内部顶点，那么可以把  $P$  分成两段： $i \rightarrow k$  和  $k \rightarrow j$ ，如图所示。即  $P$  的总长度就等于这两段的长度之和，即  $D^k[i][j] = D^{k-1}[i][k] + D^{k-1}[k][j]$ 。



有一个值得考虑的问题是，有没有可能这两段路径纠缠起来形成一个环？如果形成负环，无需关心这一情况，因为最终会检测出来；如果是正环，那么能画出一个不经过  $k$  的更短的路径，所以这种情况也不会在最终的最短路径中出现。因此下述递推式是合理的：

$$D^k[i][j] = \min \begin{cases} D^{k-1}[i][j], & \text{case 1;} \\ D^{k-1}[i][k] + D^{k-1}[k][j], & \text{case 2.} \end{cases}$$

接下来的问题和 Bellman-Ford 算法一样，需要考虑如何判断是否存在负环：

**定理 8.3** 图  $G$  包含负环当且仅当存在一个顶点  $v$ ，使得  $D^n[v][v] < 0$ 。

**证明：**若图  $G$  不包含负环，根据前面的推理，Floyd-Warshall 算法一定会返回正确的最短路径长度，所以  $D^n[v][v] = 0$ 。若包含负环，那么根据负环的定义，存在一个顶点  $v$ ，使得从  $v$  出发回到自己的最短路径长度小于 0。所以在递推过程中一定会出现  $D^n[v][v] = D^{k-1}[v][k] + D^{k-1}[k][v] < 0$  的情况，这时就会更新  $v$  到  $v$  的最小值，且之后这个值只能越来越小（因为一直在取  $\min$ ），所以  $D^n[v][v] < 0$ 。 ■

基于以上讨论可以写出 Floyd-Warshall 算法的伪代码。最 naive 的方式就是用三维数组  $D[k][i][j]$  表达  $D^k[i][j]$  的含义，那么首先要讨论 base case，因为递推是  $k$  逐渐增大的过程，所以 base case 就是  $k = 0$  的情况，接下来用三层循环构建全部的解即可，因此时间复杂度是  $O(n^3)$ ，相比于调用  $n$  次 Bellman-Ford 算法更好。PPT 第 11 页给出了一种降维的方式：其实  $k$  没有必要出现在下标，只需要在增大  $k$  的过程中更新  $i$  和  $j$  之间的最短路长度即可，因为随着  $k$  增大最短路一定是递减的，对所有的  $k$  维护一个值足矣（但 Bellman-Ford 算法不能这么降维，因为判断负边需要二维子问题）。至于重建最短路径，在循环的时候不断更新每对顶点  $i$  和  $j$  形成最短路径的最优的  $k$ ，因为这样最后一次  $k$

的更新就对应于  $i$  和  $j$  之间最短路的必经之路。然后可以回溯在线性时间内重建给定的两点之间的最短路径。

事实上关于最短路的问题的研究时至今日仍在进行, STOC’23, FOCS’24 和 STOC’25 仍然有相关的文章发表, 这一领域的研究仍然是非常活跃的。事实上, 所有顶点对的最短路径是否存在  $O(n^{2.99})$  的算法仍然是一个很著名的开放性问题, 留待后人解决。

## 8.8 总结

### 8.8.1 动态规划、贪心算法、运筹学

我想在看过这么多的例子之后, 可以尝试总结一下动态规划解决问题的范式。实际上, 使用动态规划解决的问题一般都有一个特点, 即是希望求解在满足一定条件下某个问题的最优解, 事实上这是数学规划的最经典的问题, 也是运筹学的出发点。例如 0-1 背包问题, 事实上可以建模成如下混合整数规划问题:

$$\begin{aligned} & \max \sum_{i=1}^n v_i x_i, \\ \text{s.t. } & \sum_{i=1}^n w_i x_i \leq W, \\ & x_i \in \{0, 1\}. \end{aligned}$$

当然这里并非要讨论运筹学, 毕竟这是一个更广且更深的话题, 只是本讲和下一讲贪心算法中遇到的问题大都是运筹学问题, 很多问题在运筹学中都有数学规划的“通法”解决, 但这两章中遇到的问题在结构上具有一定的特殊性, 使得我们能用更漂亮的办法解决。

### 8.8.2 动态规划与分治法

下一讲将讨论动态规划和贪心算法能解决的问题的结构上有什么区别, 现在回到本讲开头遇到的一个情况, 即爬楼梯问题不适合分治算法, 因此这引出了一个分治法和动态规划方法比较的问题: 两者似乎都是组合子问题的解, 那么更具体的区别在哪里呢?

在拉夫加登的《算法详解》中, 他做出了如下总结, 诸位可以对照之前学过的例子(分治的经典就是归并排序)体会:

1. 在典型的分治算法中, 每个递归调用只提供一种方法把输入划分为更小的子问题。在动态规划中, 如果采用递归, 每个递归调用的选择就比较开放, 可以对更小子问题的多种定义方法进行考虑并从中选择最优的一种。

2. 由于动态规划算法的每个递归调用会对更小子问题的多个选择进行试验，因此子问题一般会在不同的递归调用中重复出现（或者说子问题之间有 overlapping）。因此，将子问题的解决方案进行缓存是一种可以无脑进行的优化。在大多数分治算法中，所有的子问题都是不同的，因此没有必要特它们的解决方案进行缓存。
3. 分治算法的大多数经典应用把任务的多项式复杂度的简单算法替换为更快的分治版本。动态规划得力的应用是把需要指数级时间复杂度的任务（例如分举搜索）优化为多项式时间级的复杂度，因此是一种基于子问题的聪明的枚举法。
4. 相对来说，分治算法的子问题大小一般不会超过输入的某个比例（例如 50%）。动态规划的子问题只要小于输入就没有问题，只要满足正确性。
5. 分治算法可以看成动态规划的一种特殊情况，每个递归调用选择一个固定的子问题集合以递归的方式解决。作为一种更为复杂的算法范例，动态规划相比分治算法适用于范围更广的问题，但它技术要求也更高（至少需要足够的实践）。

面临一个新问题时，应该选择哪种算法范例呢？如果能够发现一种分治算法解决方案，就应毫不犹豫地使用它。如果分治算法无能为力，尤其当失败的原因是组合步骤总需要大量从头开始的计算时，就可以尝试使用动态规划（例如之前用分治法分析过爬楼梯、加权独立集合、最优二叉搜索树等问题，它们子问题的划分要么会纠缠，要么会不知如何划分）。

### 8.8.3 最后的话

或许现在的讨论总是强调回溯法的效率低下以及分治算法的不可行性，这容易使得我们认为动态规划是一种“万全之法”。当然事实上显然不是这样的，动态规划对于很多问题而言也是不适用的。PPT 第 17 页给出了两种不适用的情况，第一是没有最优子结构，例如出现了 History-dependency，那么你如果从 base case 出发迭代，当迭代到打叉的点时，会因为前面的最优路径经过了冲突的点而舍弃这条路，但你可能因此舍弃了一个很好的解。其二是子问题如果没有 overlapping，那么递推计算可能是不完备的，这时用分治算法可能会更好。

总结而言，动态规划重点在于两个步骤：其一是寻找问题的最优子结构，即原问题的最优解会依赖于一些子问题的最优解，其二是根据递推式的特点，选取合适的循环变量自底向上记忆化逐步构建出原问题的解。我想这两步都多少有些 trick 在：不同问题可能有不同的最优子结构，可能一维就能解决，可能二维甚至更高维才能解决，需要解决的子问题个数可能是 2 个、3 个或者很多个。对于一般的问题，都可以通过思维实验的方式思考解的构成：我首先需要判断什么，才能构建出接下来的解。当然这需要偶尔的灵光一闪以及一定的经验积累，但我相信同学们会在一些问题中感受到其中的乐趣。除此之外，代码实现也是不可缺少的进一步理解动态规划的方式，这是针对于第二步记忆化而言的，这里 base case 的选取，以及循环变量的选取（可能就是顺序遍历，也可能是子问题的长度等）都是在代码实现中不可或缺的。

总而言之，除了最后最短路这种图上的问题比较特别外，其它很多问题使用动态规划并不是很困难的事情。或许使得动态规划是一个让人觉得有点费解的方法的很大一部分理由其实是因为这个不明所以却在当年又迫不得已的名字！倘若动态规划改名递推法，我想从一开始的讨论诸位就会觉得上面的方法并没有多么特殊，对吗？这种方法会和回溯、分治、贪心一样自然，甚至比回溯和贪心更加有章可循。

# 动态规划补充

2023-2024 学年春夏学期高级数据结构与算法分析

吴一航

2024 年 4 月 22 日

# 动态规划的来由

1966 年，Dynamic Programming 登上国际顶级期刊《Science》，理查德·贝尔曼（Richard Bellman）为唯一作者，该论文被引 28421 次。贝尔曼在他的自传中写道：“1950 年秋季，我在兰德公司工作。我的第一个任务是为多阶段决策过程找到一个名称。”

“一个有趣的问题是，“动态规划”这个名字是从哪里来的？20 世纪 50 年代不是搞数学研究的好年代。当时在华盛顿有一位非常有趣的人，他叫威尔逊（Charles E. Wilson, 1953-1957 年任美国国防部长），是当时的美国国防部长，他对“研究”这个词有病态的恐惧和憎恨。提到研究，他的脸会涨得通红，如果有人在他面前用“研究”这个词，他就会变得很暴力。你可以想象他对数学这个词的感受。兰德公司受雇于空军，而空军的老板基本上是威尔逊。因此，我觉得我必须做点什么来瞒过威尔逊和空军，不让他们知道我实际上是在兰德公司做数学研究。”

# 动态规划的来由 (Cont'd)

“所以，给这个研究起个什么名字好呢？首先，我对 planning, decision making 和 thinking 比较感兴趣。但是由于种种原因，planning 这个词并不合适。因此，我决定使用 programming 这个词。我想让大家明白，这是动态的，是多级的，是时变的——我想，这个名字可以起到一石二鸟的效果。作为形容词，dynamic 还有一个很有趣的特性，那就是 dynamic 这个词不可能有贬义。我们不可能想出一些可能使它具有贬义的组合。因此，我认为 dynamic programming 是个好名字。这是一件连国会议员都不会反对的事情。所以我把它作为我研究活动的“保护伞”。”

# 动态规划的范式

动态规划方法通常用来求解最优化问题，这类问题可以有很多可行解，每个解都有一个值，我们希望寻找具有最优值（最小值或最大值）的解。我们通常按如下 4 个步骤来设计一个动态规划算法：

- ① 刻画一个最优解的结构特征；
- ② 递归地定义最优解的值；
- ③ 计算最优解的值，通常采用自底向上的方法；
- ④ 利用计算出的信息构造一个最优解。

更精炼地，动态规划就是为一个具有所谓最优子结构性质（即原问题最优解可以由子问题最优解递推得到）的最优化问题寻找一个子问题到原问题的递推式，然后用记忆化方法求解，最后有时我们需要构造出这一最优解。

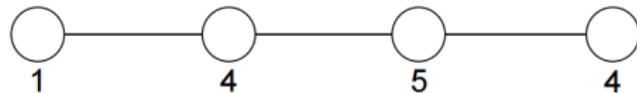
# 加权独立结合问题 (WIS)

**【打家劫舍】**你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

# 加权独立结合问题 (WIS)

**【打家劫舍】**你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。给定一个代表每个房屋存放金额的非负整数数组，计算你不触动警报装置的情况下，一夜之内能够偷窃到的最高金额。

问题抽象：考虑一个无向图  $G$ ，其上所有点都在一条线上（这种图我们称其为路径图），每个点都有一个非负权重。我们称  $G$  的独立集合是指顶点互不相邻的子集（换句话说独立集合不会同时包含一条边上的两个点），然后要求解一个具有最大顶点权重和的独立集合，这一问题我们称为一维的加权独立集合问题。



## 加权独立结合问题 (WIS) (Cont'd)

分为如下两种情况讨论：

- ① 如果  $v_n$  不在最优解  $S$  中，那么  $S$  实际上可以看成前  $n-1$  个点构成的路径图  $G_{n-1}$  组成的子问题的一个可行解。事实上  $S$  一定是  $n-1$  个点的路径图的最大加权独立子集（即最优解）；
- ② 如果  $v_n$  在最优解  $S$  中，那么由于是独立集合问题，那么  $v_{n-1}$  就不能在最优解中，因为  $v_{n-1}$  和  $v_n$  相邻。 $S - \{v_n\}$  一定是前  $n-2$  个点构成的路径图  $G_{n-2}$  的最优解，所以整个问题的最优解就是  $G_{n-2}$  的最优解加上  $v_n$ 。

## 加权独立结合问题 (WIS) (Cont'd)

分为如下两种情况讨论：

- ① 如果  $v_n$  不在最优解  $S$  中，那么  $S$  实际上可以看成前  $n-1$  个点构成的路径图  $G_{n-1}$  组成的子问题的一个可行解。事实上  $S$  一定是  $n-1$  个点的路径图的最大加权独立子集（即最优解）；
- ② 如果  $v_n$  在最优解  $S$  中，那么由于是独立集合问题，那么  $v_{n-1}$  就不能在最优解中，因为  $v_{n-1}$  和  $v_n$  相邻。 $S - \{v_n\}$  一定是前  $n-2$  个点构成的路径图  $G_{n-2}$  的最优解，所以整个问题的最优解就是  $G_{n-2}$  的最优解加上  $v_n$ 。

问题：

- ① 为什么一定要子问题的最优解？
- ② 子问题最优一定能得到全问题最优吗？

## 加权独立结合问题 (WIS) (Cont'd)

由此，整个问题的最优解依赖于前  $n - 1$  和  $n - 2$  个点构成的子图的最优解，或者说，这两个子问题的最优是构成原问题最优的基础，这就是这一问题对应的最优子结构性质。

设前  $i$  个点的最优加权独立集合的权重之和为  $W_i$ ，我们可以写出递推关系：

$$W_n = \max\{W_{n-1}, W_{n-2} + w_n\},$$

这里是最后一步的递推关系，事实上我们可以不断利用这一递推关系自顶向下向前构建出整个问题的解，即我们有更一般的表达

$$W_i = \max\{W_{i-1}, W_{i-2} + w_i\},$$

其中  $i = 2, 3, \dots, n$ ,  $W_0 = 0$ 。

## 加权独立结合问题 (WIS) (Cont'd)

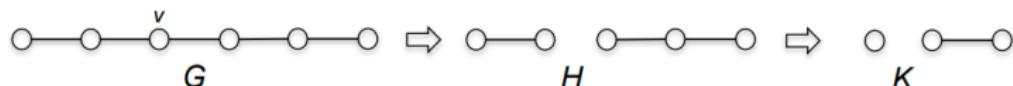
```
A := length-(n + 1) array // subproblem solutions
A[0] := 0 // base case #1
A[1] := w1 // base case #2
for i = 2 to n do
    // use recurrence from Corollary 16.2
    A[i] := max{A[i - 1], A[i - 2] + wi}
        Case 1           Case 2
return A[n] // solution to largest subproblem
```

## 加权独立结合问题 (WIS) (Cont'd)

```
S :=  $\emptyset$                                 // vertices in an MWIS
i := n
while i ≥ 2 do
    if  $A[i - 1] \geq A[i - 2] + w_i$  then    // Case 1 wins
        i := i - 1                          // exclude  $v_i$ 
    else                                    // Case 2 wins
        S := S ∪ { $v_i$ }                  // include  $v_i$ 
        i := i - 2                          // exclude  $v_{i-1}$ 
if i = 1 then                                // base case #2
    S := S ∪ { $v_1$ }
return S
```

# 加权独立结合问题 (WIS): 另一种解法

考虑一个任意的无向图  $G = (V, S)$ , 所有顶点的权重均不为负, 并且任意一个顶点  $v \in V$  具有权重  $w_v$ 。通过从  $H$  中删除  $v$  的相邻顶点以及相关联的边得到  $K$ , 如下图所示:

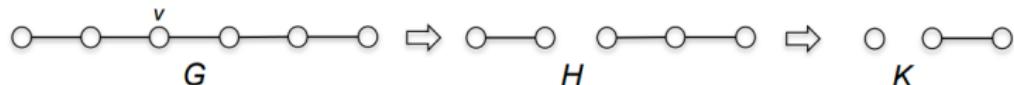


设  $W_G$ 、 $W_H$  和  $W_K$  分别表示  $G$ 、 $H$  和  $K$  的最大加权独立集合的总权重, 考虑下面这个公式:

$$W_G = \max\{W_H, W_K + w_v\},$$

# 加权独立结合问题 (WIS): 另一种解法

考虑一个任意的无向图  $G = (V, S)$ , 所有顶点的权重均不为负, 并且任意一个顶点  $v \in V$  具有权重  $w_v$ 。通过从  $H$  中删除  $v$  的相邻顶点以及相关联的边得到  $K$ , 如下图所示:



设  $W_G$ 、 $W_H$  和  $W_K$  分别表示  $G$ 、 $H$  和  $K$  的最大加权独立集合的总权重, 考虑下面这个公式:

$$W_G = \max\{W_H, W_K + w_v\},$$

回忆最优二叉搜索树 (矩阵乘法顺序、钢条切割都类似):

$$c_{ij} = \sum_{k=i}^j p_k + \min_{i \leq k \leq j} \{c_{i,k-1} + c_{k+1,j}\}, \quad i \leq j.$$

# 最优二叉搜索树代码

```
// subproblems (i indexed from 1, j from 0)
A := (n + 1) × (n + 1) two-dimensional array
// base cases (i = j + 1)
for i = 1 to n + 1 do
    A[i][i - 1] := 0
// systematically solve all subproblems (i ≤ j)
for s = 0 to n - 1 do          // s=subproblem size-1
    for i = 1 to n - s do      // i + s plays role of j
        // use recurrence from Corollary 17.5
        A[i][i + s] :=
             $\sum_{k=i}^{i+s} p_k + \min_{r=i}^{i+s} \underbrace{\{A[i][r-1] + A[r+1][i+s]\}}_{\text{Case } r}$ 
return A[1][n] // solution to largest subproblem
```

# 背包问题

0-1 背包问题：我们有  $n$  个物品，每个物品的重量为  $s_i$ ，价值为  $v_i$ ，我们有一个容量为  $C$  的背包，我们希望找到一个最优的装载方案，使得背包中的物品总价值最大。这一问题的特点是每个物品你要么不放进包里，要么完整的 1 个放进去，因此称为 0-1 背包问题。

# 背包问题

0-1 背包问题：我们有  $n$  个物品，每个物品的重量为  $s_i$ ，价值为  $v_i$ ，我们有一个容量为  $C$  的背包，我们希望找到一个最优的装载方案，使得背包中的物品总价值最大。这一问题的特点是每个物品你要么不放进包里，要么完整的 1 个放进去，因此称为 0-1 背包问题。

我们设  $V_{i,c}$  表示总重量不超过  $c$  的前  $i$  件物品组成的子集的最大总价值，那么我们可以得到如下递推关系：

$$V_{i,c} = \begin{cases} V_{i-1,c}, & s_i > c, \\ \max\{V_{i-1,c}, V_{i-1,c-s_i} + v_i\}, & s_i \leq c. \end{cases}$$

# 背包问题 (Cont'd)

```
// subproblem solutions (indexed from 0)
A := (n + 1) × (C + 1) two-dimensional array
// base case (i = 0)
for c = 0 to C do
    A[0][c] = 0
// systematically solve all subproblems
for i = 1 to n do
    for c = 0 to C do
        // use recurrence from Corollary 16.5
        if  $s_i > c$  then
            A[i][c] := A[i - 1][c]
        else
            A[i][c] :=
                max{ $\underbrace{A[i - 1][c]}_{\text{Case 1}}, \underbrace{A[i - 1][c - s_i] + v_i}_{\text{Case 2}}$ }
return A[n][C] // solution to largest subproblem
```

# 背包问题 (Cont'd)

```
// subproblem solutions (indexed from 0)
A := (n + 1) × (C + 1) two-dimensional array
// base case (i = 0)
for c = 0 to C do
    A[0][c] = 0
// systematically solve all subproblems
for i = 1 to n do
    for c = 0 to C do
        // use recurrence from Corollary 16.5
        if  $s_i > c$  then
            A[i][c] := A[i - 1][c]
        else
            A[i][c] :=
                 $\max\left\{\underbrace{A[i - 1][c]}_{\text{Case 1}}, \underbrace{A[i - 1][c - s_i] + v_i}_{\text{Case 2}}\right\}$ 
return A[n][C] // solution to largest subproblem
```

能否降成只用一维数组解决?

## 背包问题 (Cont'd)

$$\begin{aligned} & \max \sum_{i=1}^n v_i x_i, \\ \text{s.t. } & \sum_{i=1}^n s_i x_i \leq C, \\ & x_i \in \{0, 1\}. \end{aligned}$$

可以有很多的变体，例如  $x_i$  可以是任意整数，可以是  $[0, 1]$  之间的任意实数，优化目标可以是最小化，约束条件可以是大于等于的不等式等等。

# 最长公共子序列

给定两个字符串 `text1` 和 `text2`，返回这两个字符串的最长 **公共子序列** 的长度。如果不存在 **公共子序列**，返回 `0`。

一个字符串的 **子序列** 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

- 例如，`"ace"` 是 `"abcde"` 的子序列，但 `"aec"` 不是 `"abcde"` 的子序列。

两个字符串的 **公共子序列** 是这两个字符串所共同拥有的子序列。

## 示例 1：

输入: `text1 = "abcde"`, `text2 = "ace"`

输出: 3

解释: 最长公共子序列是 `"ace"`，它的长度为 3。

<https://leetcode.cn/problems/longest-common-subsequence/>

## 最长公共子序列 (Cont'd)

与矩阵链乘法问题相似，设计 LCS 问题的递归算法首先要建立最优解的递归式。我们定义  $c[i, j]$  表示  $X_i$  和  $Y_j$  的 LCS 的长度。如果  $i=0$  或  $j=0$ ，即一个序列长度为 0，那么 LCS 的长度为 0。根据 LCS 问题的最优子结构性质，可得如下公式：

$$c[i, j] = \begin{cases} 0 & \text{若 } i = 0 \text{ 或 } j = 0 \\ c[i - 1, j - 1] + 1 & \text{若 } i, j > 0 \text{ 且 } x_i = y_j \\ \max(c[i, j - 1], c[i - 1, j]) & \text{若 } i, j > 0 \text{ 且 } x_i \neq y_j \end{cases} \quad (15.9)$$

# 序列对齐问题

DNA 序列相似性：AGGGCT 和 AGGCA 需要对齐，可以插入空格，其中空格和非空格扣 1 分，两个字母对不上扣 2 分，求最小扣分。

# 序列对齐问题

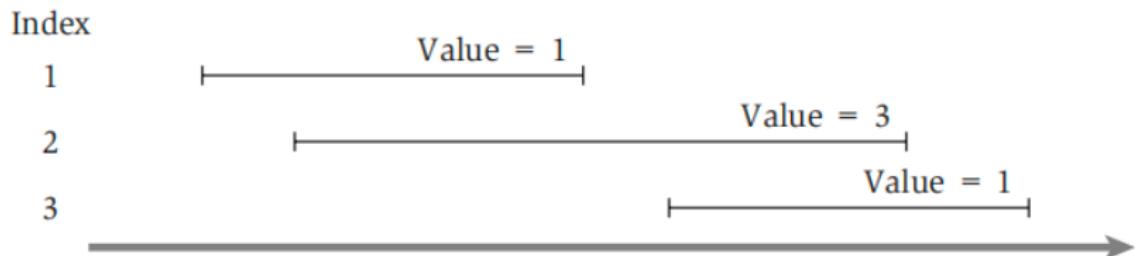
DNA 序列相似性：AGGGCT 和 AGGCA 需要对齐，可以插入空格，其中空格和非空格扣 1 分，两个字母对不上扣 2 分，求最小扣分。

$$P_{i,j} = \min \underbrace{\{P_{i-1,j-1} + \alpha_{x_i y_j}\}}_{\text{Case 1}}, \underbrace{\{P_{i-1,j} + \alpha_{gap}\}}_{\text{Case 2}}, \underbrace{\{P_{i,j-1} + \alpha_{gap}\}}_{\text{Case 3}}.$$

编辑距离（类似题目）：<https://leetcode.cn/problems/edit-distance/>

# 加权活动选择问题

给定一个活动集合  $S = \{a_1, a_2, \dots, a_n\}$ , 其中每个活动  $a_i$  都有一个开始时间  $s_i$  和结束时间  $f_i$ , 且  $0 \leq s_i < f_i < \infty$ 。每个活动  $a_i$  也有一个权重值  $w_i$ 。如果活动  $a_i$  和  $a_j$  满足  $f_i \leq s_j$  或者  $f_j \leq s_i$ , 则称活动  $a_i$  和  $a_j$  是兼容的 (即二者时间不会重合)。活动选择问题就是要找到一个权重最大的兼容活动子集。



# 其它问题

最大子序列和

最长递增子序列：

<https://leetcode.cn/problems/longest-increasing-subsequence/>

最长回文子串：<https://leetcode.cn/problems/longest-palindromic-substring/>

分割回文串：<https://leetcode.cn/problems/palindrome-partitioning-ii/>

图上相关的问题：Bellman-Ford 算法、Floyd-Warshall 算法

红黑树 project

## Lecture 9: 贪心算法

编写人: 吴一航 [yhwu\\_is@zju.edu.cn](mailto:yhwu_is@zju.edu.cn)

说到贪心算法，想必诸位也不陌生。实际上我们已经学过很多贪心算法了，例如 Dijkstra 算法以及最小生成树的两种算法。在这些算法中，我们的任务是要得到一个最优的结果，于是每一步都选取满足某些性质的最好结果（例如 Dijkstra 每一步都找目前能找到的最近点）。如果能证明这些局部最优就是全局最优，那么贪心算法就是正确的。当然很可能贪心算法不一定能带来最优结果，那么这时可能要更换策略，或者本身问题就很难得到最优解，那么贪心作为一种高效率的启发式算法，可以尝试计算一个贪心算法能和最优解之间的差的最大比例（将在近似算法一节讨论）。接下来将讨论几个经典的贪心算法的应用，讨论设计贪心算法的关键以及证明其正确性的思路。

### 9.1 活动选择问题

给定一个活动集合  $S = \{a_1, a_2, \dots, a_n\}$ ，其中每个活动  $a_i$  都有一个开始时间  $s_i$  和结束时间  $f_i$ ，且  $0 \leq s_i < f_i < \infty$ 。如果活动  $a_i$  和  $a_j$  满足  $f_i \leq s_j$  或者  $f_j \leq s_i$ ，则称活动  $a_i$  和  $a_j$  是兼容的（即二者时间不会重合）。活动选择问题就是要找到一个最大的兼容活动子集。

假设输入中是按照  $n$  个活动结束时间从小到大排列的。在学完动态规划后，相信读者应该对这一问题并不陌生——这里又遇到了类似于加权独立集合中有冲突的情况，所以同样可以设计出两种动态规划的递推式来解决这一问题：

- 设  $S_{ij}$  表示活动  $a_i$  和  $a_j$  之间的最大兼容活动集合（开始时间在  $a_i$  结束之后，结束时间在  $a_j$  开始之前），其大小记为  $c_{ij}$ ，那么有

$$c_{ij} = \max\{c_{ik} + c_{kj} + 1 \mid f_i \leq s_k < f_k \leq s_j\} \quad (9.1)$$

这一解法的思想更接近矩阵乘法顺序问题，会选择中间的最优解然后分为左右子问题递归。

- 设  $S_i$  表示活动  $a_1, a_2, \dots, a_i$  的最大兼容活动集合，其大小记为  $c_i$ ，那么有

$$c_i = \max\{c_{i-1}, c_{k(i)} + 1\} \quad (9.2)$$

其中  $k(i)$  表示在  $1 \leq k \leq i$  中， $f_k \leq s_i$  且  $c_k$  最大的  $k$ ，即不与  $a_i$  冲突的最晚结束的活动。这一思想更接近背包问题的思路，即考察最后一个是否在解中，分成两种情况考虑。

看起来第二种办法更加便捷（也更加符合常人思维，特别是和加权独立集合问题比较），但也需要  $O(n^2)$  的时间来解决这一问题（第二种递推式中  $k(i)$  的计算也是需要时间的）。第一种方法这里也要提一下，

看起来时间复杂度是  $O(n^3)$  的，但实际上上一讲最优二叉搜索树的平方优化也可以用在这里，所以复杂度也可以降到  $O(n^2)$ 。但对速度的追求不止于此，我们希望有一个更高效的算法，因此还需要新的思路，即使用贪心的思路，因为很多时候贪心只需要一个遍历即可确定出答案，所以时间上有很大的优势。

PPT 上给出了几种错误的贪心想法，这也表明了贪心并不是那么容易就能找到正确的贪心策略，需要不断地举反例否定自己，如果觉得自己逻辑足够严密就可以开始尝试证明。在这里可以找到一个可能正确的贪心策略为从前到后每次选择不冲突的最早结束的活动——这其实是一个很自然的想法，因为结束越早越能给后面的活动安排留出余地，因此也顺理成章能让更多活动都安排进来。下面来证明这一策略的正确性：令  $S_k = \{a_i \in S \mid s_i \geq f_k\}$ ，即为在  $a_k$  结束后开始的任务集合。当做出贪心选择，选择了  $a_1$  后，剩下只需要求解  $S_1$  这一子问题即可。如果  $a_1$  确实在最优解中，那么原问题的最优解显然就由活动  $a_1$  及子问题  $S_1$  的最优解构成（这一点称为贪心算法的最优子结构，根据前面讨论的动态规划的最优子结构性质可知是显然的，当然后面也马上会给出严谨证明）。然后  $S_1$  内又可以按照贪心策略选择新的结束时间最早的活动，以此类推得到全部的解。现在还剩下一大问题：贪心选择最早结束的活动真的就是最优解的一部分吗？下面的定理证明了这一点。

**定理 9.1 (活动选择问题贪心选择性质)** 考虑任意非空子问题  $S_k$ ，令  $a_m$  是  $S_k$  中结束时间最早的活动，则  $a_m$  在  $S_k$  的某个最大兼容活动子集中。

**证明：**令  $A_k$  是  $S_k$  的一个最大兼容活动子集，且  $a_j$  是  $A_k$  中结束时间最早的活动。若  $a_j = a_m$ ，则已经证明  $a_m$  在  $S_k$  的某个最大兼容活动子集中。若  $a_j \neq a_m$ ，令集合  $A'_k = A_k - \{a_j\} \cup \{a_m\}$ ，即将  $A_k$  中的  $a_j$  替换为  $a_m$ 。很显然的， $A'_k$  中的活动是不相交的，因为  $A_k$  中的活动是不相交的，而  $a_m$  是  $A_k$  中结束时间最早的活动，即  $f_m \leq f_j$ ，所以  $a_m$  不可能和  $A_k$  后面的活动冲突。由于  $|A_k| = |A'_k|$ ，因此得出结论  $A'_k$  也是  $S_k$  的一个最大兼容活动子集，且它包含  $a_m$ 。 ■

这一证明思想称为“交换参数法”，即可以假设存在一个最优选择，其中某个元素可能不在贪心选择中，然后通过交换贪心选择和最优选择的元素来构造一个不可能变差的解。这一思想在很多贪心算法的证明中都有应用，在后面的讨论中也会看到。在说明了这一性质（称为贪心选择性质）后，需要将之前挖的最优子结构的坑填上，只有结合这两点才能保证不断选择贪心选择的确是能得到最优解的：

**定理 9.2 (活动选择问题最优子结构)** 在活动选择问题中，用贪心策略选择  $a_1$  之后得到子问题  $S_1$ ，那么  $a_1$  和子问题  $S_1$  的最优解合并一定可以得到原问题的一个最优解。

**证明：**反证法：假设  $a_1$  和子问题  $S_1$  的最优解  $C_1$  合并得到的解  $C$ ，不是原问题的一个最优解。假设  $C'$  是原问题的一个最优解，则  $|C'| > |C|$ 。根据贪心选择性质可知，将  $a_1$  替换掉  $C'$  中的第一个元素得到  $C''$  不会使得结果变差，即  $|C''| \geq |C'| > |C|$ ，那么将  $C''$  除去  $a_1$  后，剩余的部分其实也是子问题  $S_1$  的一个解  $C''_1$ ，由于  $|C''| > |C|$ ，所以  $|C''_1| > |C_1|$ ，这和  $C_1$  是子问题  $S_1$  的最优解矛盾，因此原问题的最优解一定是  $a_1$  和子问题  $S_1$  的最优解合并得到的解。 ■

事实上思路非常简单，就是反证法，如果综合起来不是最优解，那  $C_1$  也不可能子问题最优解，因为能找到一个更优的解。事实上以前的动态规划问题的最优子结构也可以采用这样的证明方法，只是动态规划中都认为最优子结构性质太显然所以没有特别强调。

综合以上贪心选择性质和最优子结构性质，最优解的构造方式就是：首先根据贪心选择性质找到最早结束的活动  $a_1$ ，根据最优子结构性质可知，它和  $S_1$  的最优解一起可以形成全问题的最优解，然后任务就变成了找  $S_1$  的最优解。重复贪心选择性质，要找到  $S_1$  中最早结束的  $a_{i_1}$  作为贪心选择，这样根据最优子结构， $S_1$  的最优解又可以表达为  $a_{i_1}$  和剩余子问题  $S_{i_1}$  的最优解的组合，然后找  $S_{i_1}$  的最优解，用贪心选择然后得到剩余子问题……以此类推，利用归纳法即可得到，直到子问题为空时可以得到整个问题的最优解。

当然，对称的也可以从后往前选择最晚开始的活动，看起来非常合理，事实上是否正确呢？答案是肯定的，证明方法也和上面差不多，留作讨论题：

**【讨论】** 证明：从后往前依次选择不冲突的最晚开始的活动是一个正确的贪心策略。

最后来看时间复杂度，很显然的，在输入活动已按结束时间排序的前提下，只需要按结束时间升序遍历一遍所有活动即可，因此只需  $O(n)$  的时间就能找到最优解。这一算法的时间复杂度是线性的，因此是非常高效的。如果未排序，可以在  $O(n \log n)$  的时间内完成排序，因此总时间复杂度是  $O(n \log n)$ 。

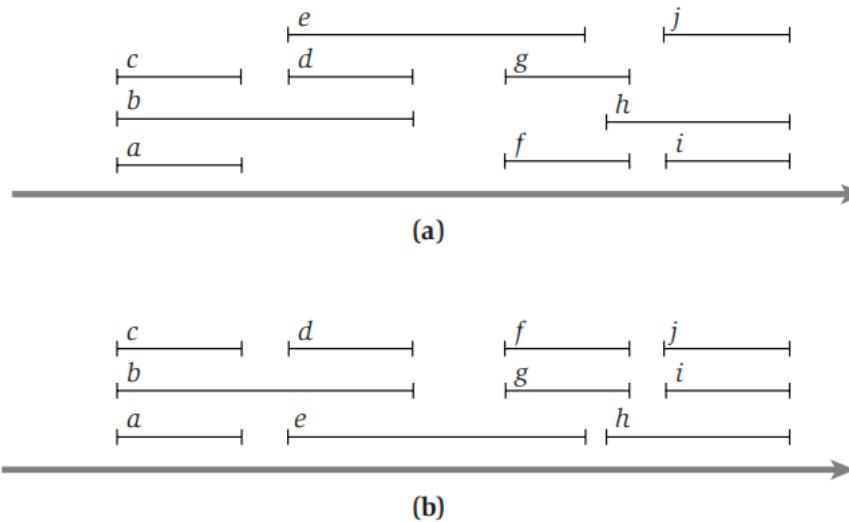
当然这一问题有很多有趣且非常常见的变体，可以在这里简单说明一下：

**【变体 1】**（加权活动选择问题）现在的问题不再是希望能包容进来的活动越多越好，而是每个活动都有一个权重  $w_i$ ，希望找到一个最大权重的兼容活动集合（前面讨论的问题实际上只是所有活动权重相等的特例）。事实上，很容易举出反例说明一般的贪心算法失效，所以现在可以转向最开头给出的动态规划算法，只是所有的加 1 应当改为加权重。

**【变体 2】**（区间调度问题）现在的问题不再是最化兼容集合的大小或者权重，而是所有活动都必须举办，考虑将所有活动分配到最少的教室中，使得每个教室内的活动不冲突。

事实上这一问题只需要最简单的贪心算法就可以了。首先给出算法，然后给出简要的正确性说明。首先将所有活动按照开始时间排序，设置初始教室数量为 1，然后从前往后遍历。每次选择一个活动时，都看当前的教室中的活动有没有不冲突的，如果有就直接放进对应的教室，如果全部冲突则新开一个教室。

这样的算法为什么正确？可以看下页图中的例子。图 (a) 展示了 10 个输入的活动，事实上可以立刻断言：至少也需要三个教室才能完成活动。原因很简单，因为最多出现了三个活动同时进行的情况。而根据刚刚的方法，不可能分配出超过 3 个教室的情况，因为是出现冲突时才新开教室的，如果开出了四个教室那就说明在某个时刻有四个活动同时进行，这是不可能的。因此算法在这一场景下一定是正确的，显然更进一步的推广到任意冲突的情况也是正确的。图 (b) 给出了一个可行的解。这里贪心算法正确性的证明没有像前面那样用贪心选择和最优子结构，因为有更显然的教室数量上下界相等的性质可以更方便地说明。



【拓展】给你一个非负整数数组 `nums`, 你最初位于数组的第一个下标。数组中的每个元素的值代表你在该位置可以跳跃的最大长度。

1. 设计一个线性时间算法判断你是否能够到达最后一个下标, 如果可以, 返回 `true`; 否则, 返回 `false`;
2. 假设题目给定的都是可以到达最后一个下标的数组, 那么如何设计一个算法, 使得你到达最后一个下标的步数最少? 一个平方级别的算法可以用动态规划实现, 但可以用贪心算法实现一个线性级别的算法。

举个例子, `nums = [2,3,1,1,4]`, 那么显然第一个问题答案为 `true`, 第二个问题答案为 2, 实现方法为从下标为 0 跳到下标为 1 的位置, 跳 1 步, 然后跳 3 步到达数组的最后一个位置。

本题来源于 leetcode, 感兴趣的同学可以利用互联网找到问题的解答。这类题目往往思路比较巧妙, 因为在线算法的设计往往需要一些观察和技巧。

## 9.2 调度问题

假设现在有  $n$  个任务, 每个任务  $i$  都有一个正的完成需要的时间  $l_i$  和一个权重  $w_i$ 。假定只能按一定顺序依次执行这些任务, 不能并行。

很显然的, 有  $n!$  种调度方法, 记  $\sigma$  为某一种调度, 那么在调度  $\sigma$  中, 任务  $i$  的完成时间  $C_i(\sigma)$  是  $\sigma$  中在  $i$  之前的任务长度之和加上  $i$  本身的长度。换句话说, 在一种调度中, 一个任务的完成时间就是这个任务从整个流程开始到它自己被执行完毕总共执行的时间。目标是最小化加权完成时间之和:

$$T = \min_{\sigma} \sum_{i=1}^n w_i C_i(\sigma).$$

举个例子，有三个任务， $l_1 = 1, l_2 = 2, l_3 = 3, w_1 = 3, w_2 = 2, w_3 = 1$ 。如果把第一个任务放在最前，第二个放在其次，第三个放在最后，那么三个任务的完成时间就是 1、3、6，因此加权时间和为  $3 \times 1 + 2 \times 3 + 1 \times 6 = 15$ 。如果读者检查全部的六种排序，这的确是最小的那个。

接下来希望设计一个贪心算法高效地对所有情况都返回一个正确的结果，因为贪心算法看着就非常适合这一问题：可以按某个标准不断确定下一个任务是哪个，直至最后选出完整的最优解。思考贪心策略并不是一件很简单的事情，在后面会总结一些技巧例如考虑极端情况或者控制变量，这里就尝试控制变量：如果所有任务长度相同，应该把权重大还是小的放在前面？如果每个任务权重相同，应该先放时间短的还是长的？答案太显然了，就是先放权重大、时间短的。所以自然而然得到了如下两种可能的贪心策略：

1. 计算每个任务  $i$  的  $w_i - l_i$ ，按从大到小降序调度任务；
2. 计算每个任务  $i$  的  $w_i/l_i$ ，按从大到小降序调度任务。

显然二者都符合前面的直观，但哪个是正确的结果呢？如果回顾加权完成时间之和的公式，会发现权重和时间是会相乘的，所以第二种直观上看起来合理一些，事实上也的确如此，考虑一个例子：有两个任务， $l_1 = 5, l_2 = 2, w_1 = 3, w_2 = 1$ ，两种方法会返回不同的结果，而第二种才能返回最优解。

当然这只能说明第一种肯定是错误的，第二种的正确性还需要严谨证明。类似于前面的活动选择问题，这里仍然分为贪心选择性质和最优子结构两个部分进行证明：

**定理 9.3 (调度问题的贪心选择性质)** 令  $i$  是当前  $w_i/l_i$  最大的任务，则在当前问题下，则一定存在将  $i$  排在首位的最优调度方式。

**证明：**仍然使用“交换参数法”：假设有一个最优解  $C$ ，如果它的第一个任务是  $i$ ，结束证明。如果不是，考虑将  $i$  不断与前一个任务交换，直到换到第一个位置的过程。假设现在排在  $i$  前面一位的是  $j$ ，则一定有

$$\frac{w_i}{l_i} \geq \frac{w_j}{l_j},$$

又所有数都是正数，故移项有  $w_i l_j - w_j l_i \geq 0$ 。不难看出， $i$  和  $j$  交换前后其它的任务加权时间完全没有变化，变化只在  $i$  和  $j$ ，设在  $i$  和  $j$  前面的总时间为  $t$ ，则交换前  $i$  和  $j$  的加权时间和为

$$t_1 = w_j(t + l_j) + w_i(t + l_j + l_i),$$

交换后为

$$t_2 = w_i(t + l_i) + w_j(t + l_i + l_j),$$

二者作差化简有

$$t_1 - t_2 = w_i l_j - w_j l_i \geq 0,$$

故把  $i$  往前换，加权时间和一定不会变大，故仍然保证最优解。那么就可以不断把  $i$  往前换，直到它在第一个位置，这样就证明了贪心选择性质。 ■

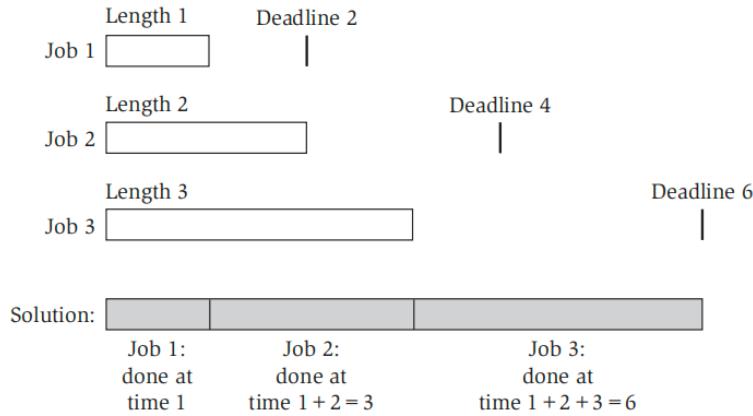
**定理 9.4 (调度问题的最优子结构)** 在调度问题  $S$  中, 用贪心策略首先选择了最大的  $w_i/l_i$  对应的任务  $i$  后, 剩下的子问题  $S_1$  (即在除  $i$  外的任务中寻找一个最小化加权完成时间之和的解) 的最优解  $C_1$  和  $i$  一起一定构成了原问题的一个最优解  $C$ 。

**证明:** 和活动选择问题完全一致, 非常直观的结论就用反证法秒杀: 如果  $C$  不是最优解, 那么一定有一个最优解  $C'$ , 它对应的加权完成时间之和  $T' < T$ , 其中  $T$  是  $C$  对应的加权完成时间之和。

根据贪心选择性质, 如果把  $C'$  中的  $i$  不断通过相邻交换换到第一个位置, 情况一定不会变差, 因此还是最优解, 将这一新的最优解记为  $C''$ 。于是  $C''$  在选择了  $i$  之后, 剩下的选择实际上也是  $S_1$  的一个解, 由于  $T' < T$ , 这表明  $C''$  中对应的  $S_1$  的解必定比  $C_1$  更好, 但  $C_1$  是最优解, 因此得到矛盾。所以  $C_1$  和  $i$  一起一定构成了原问题的一个最优解  $C$ 。 ■

综合以上两点, 就很容易递推得到整个问题的最优解可以通过按  $w_i/l_i$  降序进行选择的方式得到。这一问题的时间复杂度显然最耗时的部分也在排序上, 不再重复。

**【变体 1】(最小化最大延时)** 设有  $n$  个任务, 每个任务  $j$  具有一个完成需要的时间  $t_j$ , 以及一个截止日期  $d_j$ 。只能依次次完成这些任务, 不能并行, 如下图所示:



三个任务长度分别为 1、2、3, 截止日期分别为 2、4、6, 按如图方式排序, 所有任务都能在截止前完成。当然有的时候可能没这么幸运, 如果有任务超时, 定义延迟时间  $l_j$  为其完成的时间减去截止日期  $d_j$ , 如果没有超时则延迟定义为 0, 目标是最小化所有任务的最大延迟。

对这一问题已经有前面基础调度问题的经验, 可以很容易想到如下几种贪心策略:

- 按完成时间  $t_j$  从小到大排序, 然后依次完成任务; 这很符合小学的时候学习的接水的角度问题, 但这一策略在这一问题下是不正确的, 因为完成时间短的可能截止日期很长, 例如考虑两个任务, 一个需要 1 天, 截止日期 100 天后; 另一个需要 10 天, 截止日期 10 天后, 那么按照这里的调度方式显然不对;
- 也可以根据前面基础调度问题的思想设计, 看起来按照  $d_j - t_j$  从小到大排序是个很好的选择: 这表征了某个任务的冗余时间量, 冗余时间越短应该越早安排你完成任务。这看起来很完美, 但事

事实上也是错的！例如考虑两个任务，一个需要 1 天，截止日期 2 天后；另一个需要 10 天，截止日期 10 天后，那么按照这里的调度方式也不对。

事实上上面两种思路出现了一个矛盾：第一个表明完成时间  $t_j$  越短越应该先完成，看起来很合理；第二个  $t_j$  越短反而冗余时间变大，应该晚完成——我们陷入了一个尴尬的局面，这是前面的基础调度问题没有的：这个  $t_j$  究竟应该在贪心中占据什么地位呢？答案是，没有地位就行。正确的贪心策略是直接按截止日期  $d_j$  从小到大排序进行调度：这看起来有些不可思议，因为完全忽略了  $t_j$ ，但事实证明如果利用“交换参数”法是可以证明这一贪心选择的合理性的，过程和上面基础调度问题类似，可以找到一组相邻逆序对然后交换，很容易发现交换后情况不会变差，即交换后二者延时最大值不会上升，那么整体最大的延时也不会上升（如果还是不太会可以参考《Algorithm Design》）。说明了这一贪心选择性质后，就可以利用显然的最优子结构性质导出得到最优解的方式：首先选择截止时间最早的任务  $a_{i_1}$  完成，然后剩下的问题调度是一个子问题  $S_{i_1}$ ，即除去  $a_{i_1}$  后剩下的任务的最小化最大延迟的问题，显然  $a_{i_1}$  和  $S_{i_1}$  的最优解放在一起能构成全问题的最优解（如果不相信，可以仿照前面问题的证明方式自己尝试写证明），所以继续取  $S_{i_1}$  中截止时间最早的任务即可，以此类推，每次都选择剩下的任务中截止最早的任务就能得到全问题最优解。

**【变体 2】**（最小化延时惩罚）设有  $n$  个任务，每个任务  $j$  都只需要单位时间即可完成，但是每个任务  $j$  都有一个截止日期  $d_j$ ，如果超出截止日期则会受到惩罚  $w_j$ ，希望找到一个调度方案，使得总的惩罚最小。这一问题见《算法导论》16.5 节，可以在初步了解后续介绍的拟阵后给出一个非常通用有效的贪心解法。

**【变体 3】**（完成工时最小化）假设有  $n$  项作业，每个作业  $j$  具有已知的长度  $l_i$ ，不同于前面的问题都只有一台机器顺序执行所有工作，这里有  $m$  台相同的机器可以同时处理作业。希望找到一个调度方案，使得所有作业的完成时间最小。

举个简单的例子，假如有两台一样的机器，作业长度分别为  $[1, 2, 2, 3]$ ，那么可以将作业分配为  $[1, 2], [2, 3]$ ，这样完成时间最小为 5。如果将作业分配为  $[1, 3], [2, 2]$ ，那么完成时间最小为 4。这一问题直接使用贪心算法无法保证最优解，将在近似算法一讲中讨论这一问题。

**【拓展】**（离线最优缓存问题）如果本学期各位同学同时在学习计算机组成/计算机系统 II，那么这一问题是一个令人兴奋的联动。简而言之就是 cache 容量有限，在 cache miss 且 cache 已被装满的时候如何调整 cache 内的元素使得 cache miss 最小化。感兴趣的读者可以在《算法导论》的思考题 16-5 找到本题的详细版本，那里叙述了一种所谓“将来最远”的贪心策略，要求诸位证明其具有最优性质。

## 9.3 贪心算法的讨论

### 9.3.1 贪心算法的范式

贪心算法通过做出一系列选择来求出问题的最优解。在每个决策点，它做出在当时看来最佳的选择。这种启发式策略并不保证总能找到最优解，但对有些问题确实有效，如活动选择问题。一般地，可以按如下步骤设计贪心算法：

1. 将最优化问题转化为这样的形式：对其做出一次选择后，只剩下一个子问题需要求解；
2. 证明做出贪心选择后，原问题总是存在最优解，即贪心选择总是安全的；
3. 证明做出贪心选择后，剩余的子问题满足性质：其最优解与贪心选择组合即可得到原问题的最优解，这样就得到了最优子结构。

可以代入活动选择问题来看这一范式的应用，回忆对子问题  $S_k$  的定义以及正确性的证明，上述思路是非常清晰的。下面的问题是如何证明一个贪心算法是否能求解一个最优化问题呢？并没有适合所有情况的方法，例如之前涉及的“交换参数法”并不是唯一的证明贪心选择正确性的方法，在活动选择变体 2 中已经见到了对于特定问题更简单的证明方式。如果读者回顾 Dijkstra 算法，就会发现在那里是利用了数学归纳法来证明算法的正确性的。而最小生成树的两种算法正确性证明则涉及了图问题的一些更复杂的特性。当然这里学到的“交换参数法”是对于很大一部分问题比较直观的一种证明手段，但贪心这一思想实在是应用太广泛，所以无法保证这种证明方法对所有问题都适用。

但对于大部分问题贪心算法的证明而言，贪心选择性质和最优子结构是两个关键要素（读者可以代入前面已经讲到的问题进行理解）：

1. 贪心选择性质：可以通过做出局部最优（贪心）选择来构造全局最优解。换句话说，当进行选择时，直接做出在当前问题中看来最优的选择，而不必考虑子问题的解。

这也是贪心算法与动态规划的不同之处。在动态规划方法中，每个步骤都要进行一次选择，但选择通常依赖于子问题的解。但在贪心算法中，总是做出当时看来最佳的选择，然后求解剩下的唯一的子问题。贪心算法进行选择时可能依赖之前做出的选择，但不依赖任何将来的选择或是子问题的解。

如果进行贪心选择时不得不考虑众多选择，通常意味着可以改进贪心选择，使其更为高效。例如，在活动选择问题中，假定已经将活动按结束时间单调递增顺序排好序，则对每个活动能够只需处理一次。通过对输入进行预处理或者使用适合的数据结构（通常是优先队列），通常可以使贪心选择更快速，从而得到更高效的算法。

2. 最优子结构：一个问题的最优解包含其子问题的最优解。事实上在动态规划中就提到这一名词，事实上二者的含义是完全类似的。如前活动选择问题所述，通过对原问题应用一次贪心选择即可得

到子问题，那么最优子结构的工作就是论证：将子问题的最优解与贪心选择组合在一起的确能生成原问题的最优解。这种方法隐含地对子问题使用了数学归纳法，证明了在每个步骤对子问题进行贪心选择，一步一步推进就会生成原问题的最优解。

当然还有一个很困难的问题就是如何找到一个合理的贪心思路：这的确有些困难，有时候你可以去考虑一些极端情况，从中会发现具有哪些性质的解是最想要的（例如调度问题用这一思想就很容易想到可能的解法）。在多参数（例如调度问题有权重、完成时间、截止时间等参数）的情况下，也可以固定某些参数看另一些参数的变化等，这些小技巧都能帮助对问题的基本结构有一个观察。最后在证明时也是不应完全被套路束缚，有时候也应当从一些直观中找到灵感，特别是有的时候最优子结构其实非常显然，例如前面的两个问题就是如此，所以直观的感受结合将直观转化为严谨的叙述的过程是很重要的。

**【讨论】**（分数背包问题）给定  $n$  个物品和一个容量为  $C$  的背包，物品  $i$  的重量为  $w_i$ ，价值为  $v_i$ 。希望找到一个最优的方案，使得背包中物品的总价值最大。注意和 0-1 背包问题的区别在于，可以取物品的一部分，而不一定要取全部。

1. 设计一个解决这一问题的贪心算法，并计算其时间复杂度；
2. 证明你设计的贪心算法能返回问题的全局最优解；
3. 你设计的贪心算法对 0-1 背包问题是否适用？如果适用请给出证明，不适用请举出反例。

### 9.3.2 拟阵

在上一讲中已经简单探讨了动态规划与运筹学的关联，这两讲的优化问题很多都可以转变为数学规划问题求解，但因为有些问题有更特殊的结构，所以可以用更高效的算法实现。动态规划的结构已经非常熟悉，并且已经形成了某种“套路”，但贪心算法看起来似乎无章可循。事实上，对于一类贪心算法（并非全部，因为贪心实在是太常用），其背后也有一个比较复杂但美丽的统一结构，称其为拟阵。相关内容可以在《算法导论》中找到对应，当然也可以阅读组合优化相关教材学习，这里不展开描述。

## 9.4 哈夫曼编码

哈夫曼编码希望找到一个字母表的期望长度最小（依据字母出现频率）的前缀编码，在之前最优二叉搜索树的讨论中似乎有一个类似的目标，但那里是希望最小化搜索的期望时间，并没有前缀编码的需求。事实上，哈夫曼编码具有非常强的信息论背景。相信读者应该对于香农定义的“信息熵”并不陌生：

**定义 9.5 (信息熵)** 对于一个离散随机变量  $X$ ，其信息熵定义为

$$H(X) = - \sum_x p(x) \log_2 p(x) \quad (9.3)$$

其中  $p(x)$  是  $X$  取值为  $x$  的概率。

使用以 2 为底的对数是为了保证信息熵的单位是比特，在平均意义上，信息熵可以理解为对于一个随机变量  $X$ ，需要多少比特来表示它。举几个简单的例子：

**例 9.6** 考虑一个服从均匀分布且有 32 种可能取值的随机变量  $X$ 。为了确定一个结果，需要一个能容纳 32 个不同值的标识，因此用 5 个比特足矣。而其信息熵为

$$H(X) = - \sum_{i=1}^{32} \frac{1}{32} \log_2 \frac{1}{32} = 5 \quad (9.4)$$

这也是预期的结果。

**例 9.7** 假定有 8 匹马参加的一场赛马比赛，它们的获胜概率分别为  $(\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64}, \frac{1}{64})$ 。那么这场比赛的信息熵为

$$H(X) = - \sum_{i=1}^8 p_i \log_2 p_i = 2 \quad (9.5)$$

假定我们要把哪匹马会赢的信息告诉给别人，其中一个策略是发送胜出的马的编号，这样对于任何一匹马都需要 3 个比特。但由于概率不是均等的，明智的方法是对概率大的马用更短的编码，对概率小的马用更长的编码。例如使用以下编码：0, 10, 110, 1110, 111100, 111101, 111110, 111111，这样平均每匹马需要 2 个比特，比等长的比特数更短。

事实上，在信息论中，可以证明随机变量的任一前缀 0 – 1 编码的期望长度必定大于等于其信息熵。而哈夫曼编码就是一种前缀编码，其期望长度等于信息熵。因此有很多人都尝试从信息论角度研究最优的前缀编码。1951 年，哈夫曼和他在 MIT 信息论的同学需要选择是完成学期报告还是期末考试。导师 Robert M. Fano 给他们的学期报告的题目是，寻找最有效的二进制编码。由于无法证明哪个已有编码是最有效的，哈夫曼放弃对已有编码的研究，转向新的探索，最终发现了基于有序频率二叉树编码的想法，并很快证明了这个方法是最有效的。由于这个算法，学生终于青出于蓝，超过了他那曾经和信息论创立者香农共同研究过类似编码的导师。事实上哈夫曼的构造思想在编码理论中算得上完独树一帜的，因为这一贪心算法的构造的确与当时主流的想法完全不同。当然算法过程过于简单不再赘述。在此直接跳跃到正确性证明，对 PPT 中两个看起来不明所以的证明正确性引理给出一些解释。事实上这两个证明恰好也对应了之前讨论的贪心算法的两个关键要素。

**引理 9.8 (贪心选择性质)**  $C$  为一个字母表，其中每个字符  $c \in C$  都有一个频率  $c.freq$ 。令  $x$  和  $y$  是  $C$  中频率最低的两个字符。那么存在  $C$  的一个最优前缀码， $x$  和  $y$  的码字长度相同，且只有最后一个二进制位不同。

这一引理说明在哈夫曼编码中使用的贪心选择（即把两个频率最小的结合成一个新结点）是可以保证最优解仍然存在的。证明比较简单，在这里也就只叙述思想了。思路仍然是交换参数：令  $T$  表示任意一个最优前缀码所对应的编码树，对其进行修改，得到表示另外一个最优前缀码的编码树，使得在新树中， $x$  和  $y$  是深度最大的叶结点，且它们为兄弟结点。如果可以构造这样一棵树，那么  $x$  和  $y$  的码字

将有相同长度，且只有最后一位不同。幸运的是 PPT 第 14 页的构造就满足这一点，也很容易验证新的树的代价只可能小于等于原来的树，而原来的树是最优的，因此新的树必定也是最优的。

接下来，很自然地需要证明最优子结构性质，即贪心选择加上剩下的子问题的最优解构成整体最优解：

**引理 9.9 (最优子结构性质)**  $C$  为一个字母表，其中每个字符  $c \in C$  都有一个频率  $c.freq$ 。令  $x$  和  $y$  是  $C$  中频率最低的两个字符。令  $C'$  为  $C$  去掉字符  $x$  和  $y$ ，加入一个新字符  $z$  后的字母表。给  $C'$  也定义频率集合，不同之处只是  $z.freq = x.freq + y.freq$ 。令  $T'$  为  $C'$  的任意一个最优前缀码树，那么可以将  $T'$  中叶结点  $z$  替换为一个以  $x$  和  $y$  为孩子的内部结点得到一个  $C$  的一个最优前缀码树  $T$ 。

如果上面这一命题正确，那么每次合并  $x$  和  $y$  得到  $z$  之后，按照没有  $x$  和  $y$ ，只有  $z$  的子问题继续推进贪心算法可以得到  $T'$  这一子问题的最优解，它和合并  $x$  和  $y$  得到  $z$  这一前面已经验证正确性的贪心选择一起，就构成了整体的最优解。下面证明这一引理，关键在于反证法，即如果最后不能得到最优解，那么子问题也不可能是最优解：

**证明：**记  $B(T)$  树  $T$  的代价，即所有字母的期望编码长度（即 PPT 上的 cost）。事实上很容易验证  $B(T') = B(T) - x.freq - y.freq$ ，因为  $T'$  就相当于把  $T$  中  $x$  和  $y$  的频率代价上移一层。然后可以用反证法证明这一引理。假设  $T$  不是  $C$  的最优前缀编码树，即存在树  $T''$  使得  $B(T'') < B(T)$ 。根据前面的引理，将  $T''$  中的  $x$  和  $y$  和它们的父结点替换成  $z$ ，得到一个新树  $T'''$ ，其中  $z.freq = x.freq + y.freq$ 。那么有

$$B(T''') = B(T'') - x.freq - y.freq < B(T) - x.freq - y.freq = B(T'),$$

这与  $T'$  是  $C'$  的最优前缀编码树矛盾。因此假设是错误的， $T$  必定是  $C$  的最优前缀编码树。 ■

所以这一证明其实非常简单，就是很符合直觉的如果  $T$  不是最优那么  $T'$  必定不可能是从最优解来的。综上两个引理就证明了哈夫曼编码的正确性。这一证明也是贪心算法证明的一个典型例子，可以看到贪心选择性质和最优子结构性质是贪心算法证明的两个关键要素。

## 9.5 一些拓展问题

### 9.5.1 田忌赛马问题

田忌赛马是中国历史上有名的扬长避短而在竞技中获胜的故事，下面是这一故事的改编：田忌和齐王赛马，他们各有  $n$  匹马，每次双方各派出一匹马进行赛跑，获胜一方记 1 分，失败一方记 -1 分，平局不计分。假设每匹马只能出场一次，每匹马都有一个速度值，比赛中速度快的马一定会获胜。田忌知道所有马的速度值，且田忌可以安排每轮赛跑双方出场的马，问田忌如何安排马的出场顺序，使得最后获胜的比分最大？

**【提示】** 贪心法求解田忌赛马的贪心策略是保证每一场比赛都是最优方案，分别考虑如下情况：

1. 田忌最快的马比齐王最快的马快，则拿两匹最快的马进行赛跑，因为田忌最快的马一定能赢一场，此时选齐王最快的马是最优的；
2. 田忌最快的马比齐王最快的马慢，则拿田忌最慢的马和齐王最快的马进行赛跑，因为齐王最快的马一定能赢一场，此时选田忌最慢的马是最优的；
3. 田忌最快的马与齐王最快的马速度相等，考虑以下两种情况：
  - (a) 田忌最慢的马比齐王最慢的马要快，则拿两匹最慢的马进行赛跑，因为齐王最慢的马一定会输一场，此时田忌选最慢的马一定是最优的；
  - (b) 否则用田忌最慢的马与齐王最快的马赛跑，因为田忌最慢的马一定不能赢一场，而齐王最快的马一定不会输一场，此时选田忌最慢的马一定是最优的。

显然，这一时间复杂度是线性的，因此要做  $n$  次选择，每次都是常数时间就能决定。当然目前只是直观上认为上述策略是合理的，如果要严谨说明仍然需要按照两个要素来证明。

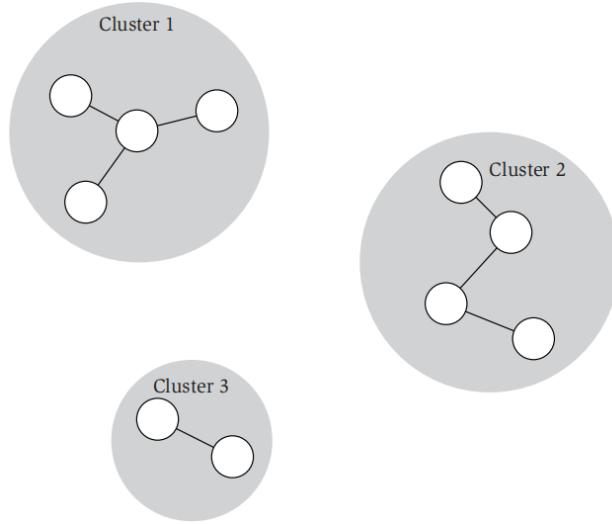
### 9.5.2 聚类问题

聚类问题是一个非常经典的无监督机器学习问题，它来源于一个非常自然的场景，即希望将一组数据分成若干个类别，使得每个类别内的数据尽可能相似，而不同类别之间的数据尽可能不同。在聚类问题中，通常会使用一些距离度量来衡量数据之间的相似性，具体的距离函数可以忽略，只需要满足度量的基本性质即可。

然后可以将聚类问题形式化：给定  $n$  个点，希望将这  $n$  个点分成  $k$  个类别  $C_1, \dots, C_k$ （其中  $k$  是给定的，称为  $k$ -聚类）。能选择的优化目标很多，这里首先定义  $k$ -聚类的“间隙值”为处于不同类别的两个点的最近距离，然后希望最大化聚类结果的间隙值（还有另一种对称的优化目标，即最小化每个点到聚类中心的距离，这一问题将在近似算法中讨论）。

有一个比较直观的做法，就是按照点之间的距离从小到大排序，然后将依次将最短距离对应的边加入图中，最后连上边的就认为在同一个聚类中，实际上知道每个聚类就是一个联通分支，于是直到最后只剩下  $k$  个联通分支时就停止。当然要注意的是如果下一个最短边将要连接的是已经在同一个联通分支中的两个点，就不用再连接这条边了。有趣的是，这一算法事实上就是用 Kruskal 算法得到的最小生成树的过程，只不过最后还剩下  $k - 1$  条边没有连接罢了。

算法正确性的证明并不困难，记根据上述贪心算法得到的结果为  $C$ ，如果有另一种聚类方式  $C'$  的话（还是考虑交换参数），那么  $C'$  中一定存在  $C$  中处于同一聚类的两个点到了不同聚类中，又这两个点的距离按照 Kruskal 算法的计算顺序一定小于等于那些在  $C$  中没有连接的边（因为  $C$  的连接顺序是按照 Kruskal 算法的计算顺序来的），因此  $C'$  的间隙值首先至少小于等于这两点的距离，然后它们的距离又小于等于  $C$  的间隙值，因此算法是最优的。



### 9.5.3 匹配问题

**【房屋交换问题】**有  $n$  个人，每个人初始时都有一个房子，但每个人不一定最喜欢他自己当前住的房子。每个人对所有房子都有个偏好顺序，即他最喜欢的的房子排在第一位，次喜欢的房子排在第二位，以此类推。现在要求找到一个房子分配新方案，使得每个人能得到比现在更喜欢的房子（或者至少保持原样）。如何设计一个贪心算法来解决这个问题？

**【稳定匹配问题】**有  $n$  个男生和  $n$  个女生，每个人对异性都有一个偏好顺序，即他最喜欢的异性排在第一位，次喜欢的异性排在第二位，以此类推。现在要求找到一个匹配方案，使得没有一对人会同时对这个匹配方案不满意（即男生  $v$  和女生  $w$  没有匹配成功，但相比于他们当前的匹配，他们更喜欢对方）。如何设计一个贪心算法来解决这个问题？

以上两个有趣的问题属于经典的算法博弈论领域的基本问题，感兴趣的读者可以阅读《斯坦福算法博弈论二十讲》的 9、10 两章，或者利用互联网搜索相关资料。

除此之外，如果读者还对贪心算法感兴趣的话，可以参考《算法导论》、《Algorithm Design》等教材上的例题与习题等，作业中的找零钱问题就源于《算法导论》，多阶段的贪心算法（最小成本树状图）等可见《Algorithm Design》，限于篇幅和时间这里不能详细展开描述。

CS2045M: 高级数据结构与算法分析

2025-2026 学年秋冬学期

## Lecture 10: NP 完全性

编写人: 吴一航 yhwu\_is@zju.edu.cn

### 10.1 引言

激动人心的，我们来到了这门课的一个新的转折点。之前的讨论介绍了几种算法设计策略，看到了很多能巧妙利用问题结构从而可以快速解决的问题，介绍了非常多如何“解决”某个问题的方法。现在需要考虑的问题是，一个问题有多难，是否存在一种方法使得它能够被高效地解决？这个问题显然比先前的考虑更加抽象，也需要更多的技巧和思考，事实上也是理论计算机科学的终极目标之一。在通常的意义下认为能“快速解决”或“高效解决”的问题是存在一个多项式时间的算法能够解决的问题（后面会解释原因），而所谓多项式时间，是指算法的时间复杂度与输入的长度之间是多项式关系。下面来看三个经典的存在多项式时间算法问题：

1. 最短路径问题：给定一个有向图  $G = (V, E)$ ，即使是带负权的，都可以在  $O(|V||E|)$  的时间内找到从单一源顶点开始的最短路径；这是多项式时间的，因为图的输入规模可以视为  $|V| + |E|$ （如果使用邻接表）；
2. 欧拉回路问题：是否存在一个回路（即起点终点是同一个顶点）使得它恰好经过图中每条边一次？这个问题可以在  $O(|E|)$  的时间内用深度优先搜索解决，这在数据结构基础课程中已经熟悉；
3. 2-CNF 可满足性问题（简称 2-SAT 问题）：回忆离散数学，称一个逻辑表达式是  $k$ -CNF（即  $k$ -合取范式）的，如果它是由  $k$  个子句的合取 ( $\wedge$ ) 构成的，每个子句是由  $k$  个变量或者它们的否定构成的析取 ( $\vee$ )。2-SAT 问题就是判断一个 2-CNF 逻辑表达式是否存在对其变量的某种 0 和 1 的赋值，使得它的值为 1，这个问题可以在多项式时间内解决。例如

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$$

满足赋值条件  $x_1 = 1, x_2 = 0, x_3 = 1$ 。这一问题与图论中的强连通分量有关，在后续介绍了归约的概念后会详细讨论，总之也是多项式时间可解的。

然而，有一个经典的容易出现的混淆的问题是 0-1 背包问题：回忆 0-1 背包问题的动态规划算法，最终的时间复杂度为  $O(nC)$ ，其中  $n$  是输入的物品个数， $C$  是背包最大容量。乍一看这是给出了多项式时间的算法，然而需要注意的是， $C$  在输入时是以二进制的方式表达的，其二进制表示的长度是  $\log C$ ，因此这个算法的时间复杂度实际上是指数级别的（当然这个复杂度的确非常有迷惑性，因此称其为伪多

项式的)。当然对于  $n$  而言,  $n$  个物品实际上对应于  $n$  个二进制物品价值和物品重量输入, 前面的图问题中  $n$  个顶点和边实际上也是  $n$  个二进制编码, 所以不会发生  $C$  这种单独一个数字导致的的窘境。

然而, 这一窘境引出了一个自然的问题: 0-1 背包问题是否存在一个多项式时间内可以解决的算法, 很遗憾, 至今无法对这个问题给出答案, 这也就引入了今天要讨论的关于问题的困难程度的话题。前面的三个多项式时间的例子以及前面关于算法设计的讨论仿佛给了我们很大的信心, 仿佛所有问题都能找到高效的算法, 但只要对上面三个问题稍微做一点点变化, 就会发现事情并不是那么简单:

1. 最短路问题变体: 如果带负圈的最短路问题中我们的要求不是找到负圈就结束, 而是给出具有最短路的无环路径, 那么这一问题就变得困难了, 目前无法在多项式时间内解决这个问题;
2. 哈密顿回路问题: 起点和终点是一个顶点, 途中经过图中所有其他节点且只经过一次。即将欧拉回路每条边经过一次改为了每个点经过一次, 然而判断一个图是否存在哈密顿回路目前也不存在多项式时间的算法, 尽管看起来和欧拉回路区别不大;
3. 3-CNF 可满足性问题 (简称 3-SAT 问题): 判断一个 3-CNF 逻辑表达式是否存在对其变量的某种 0 和 1 的赋值, 使得它的值为 1。这个问题目前也没有多项式时间的算法, 尽管和 2-SA 问题仅有一字之差。

由此可以发现, 似乎问题与问题之间亦有差别: 有的问题能找到很巧妙的方法在多项式时间内解决, 而有的问题虽然看起来和前者差不多, 但至今也没有找到多项式时间的算法。这也就带来了这一节讨论的核心: 如何形式化地定义问题的困难程度, 如何对这些问题按照困难程度进行分类, 这些困难程度背后又有什么有趣的内涵? 为了讨论这些问题, 需要首先形式化地定义问题以及计算这些问题的形式化计算模型。接下来开始正式的讨论。

## 10.2 形式语言

为了形式化地定义问题以及计算模型, 首先需要给出某种编码。这就是形式语言要干的事情。首先, 考虑一个至多可数的集合  $\Sigma$ , 称为字母表 (alphabet), 其上的字符串 (string) 定义为有限个元素 (包括 0 个) 的有序连接。称所有这样的字符串记作  $\Sigma^*$ 。

但往往, 这样的东西本身并不具备含义。一方面, 需要一个解释来表明它到底在表达什么, 比方说, 下面这个二进制字符串 ( $\{0, 1\}$  上的字符串):

0100100001100101011011000110110001101111

可以表示十进制整数 310939249775 或者 ASCII 字符串 Hello: 需要明确指出它表示的是什么东西。另一方面, 这样的结构过于简单, 以至于不能表达任何有意义的东西, 这就像是语言的无意义连接, 比方说:

军进犯大林嘴鸟军强脾气穷就怕嗲气

这是一段脸滚键盘打出来的字，它也是中文字符上面的字符串，但我们认为它毫无意义。因此要取所有字符串  $\Sigma^*$  的一个子集  $L$ ，并将其称为形式语言（formal language）。这个子集的定义在这里是由自然语言给出的，而如果是在描述复杂度那里，则需要对它给出更加精细的刻画。

下面主要讨论两种问题：

1. 给定语言  $L$ ，判断某个字符串  $\omega$  是否在  $L$  中，这被称为判定问题（decision problem）；
2. 给定语言  $L$ ，寻找某个字符串  $\omega$  使得  $\omega$  在  $L$  中，这被称为搜索问题（search problem）；
3. 给定语言  $L_1, L_2$ ，计算某个从  $L_1$  到  $L_2$  的函数，这在后面会以另外一种方式实现。

在此需要注意形式语言的几个性质：

1. 至多可数的集合上的有限字符串是可数的；这就是为什么在此定义的时候用了至多可数，而课件上用的是有限——在这里的语境下，这不会有什么影响，而且至多可数和后面讨论的逻辑系统能够联系起来。
2. 语言可以做衔接、并、交、补、Kleene 星等运算，这些运算的定义参见课件。

最后来看一个例子：如果给出一个自然语言下的问题，如何将其转变成一个形式语言的判定问题？显然，能够完成翻译的问题只有那些答案是“是”或者“否”的问题，例如：

给定一个图，判定它是否有 Hamilton 回路。

首先需要说明：

**引理 10.1** 所有图构成的集合是可数的。

这个引理的证明不难。显然，它的顶点数有限，固定顶点数的图个数有限；可数个有限集的并集依然是可数的，因此可以构造一个  $\Sigma^*$  包含所有图而不包含所有别的东西：只要把这个可数集到自然数的双射变成到  $\{1\}^*$  的双射即可，即取  $\Sigma = \{1\}$ ，这样字符串由几个 1 组成就意味着它是编号为几的图。而形式语言  $L$  就定义为所有有 Hamilton 回路的图对应的字符串构成的集合。在这里可以看到，它是  $\Sigma^*$  的一个子集，而它的定义是通过一个谓词（predicate）“有 Hamilton 回路”给出的，即这一形式语言可以写为

$$L = \{x \in \Sigma^* \mid Q(x)\},$$

其中  $Q(x)$  是谓词“ $x$  是一个有 Hamilton 回路的图”。因此判定问题就是给定一个字符串  $x$ ，判定它是否在  $L$  中。于是，一个判定问题的复杂度就被转变成了另一个问题：一个可数集的子集能有“多复杂”？这就需要接下来进一步定义计算模型后进行讨论。

## 10.3 计算模型

考虑计算过程，最基本的想法就是所谓的“算法”的概念：一个有限的指令集中取出的有限条指令，能在有限时间内确定性地完成。在 1930 年以前，从来没有过可计算性这样的概念，当给出一个问题时，自然假定它是可计算的。然而，自 Hilbert 的幻梦被 Gödel 打破以来，其他一些要求给出某种算法的问题也遭到质疑：是否有一些问题用算法是不能解决的？但是，这样的直觉想要得到严格的证明，天才的图灵发明了一种泛用的计算模型用来描述可计算函数：

**定义 10.2 (图灵机)** 考虑一个双向无限的磁带 (*tape*)，将其划分成一个一个的方格 (*cell*)，有一个磁头 (*head*)，磁头中有一个有限状态机，其中包含有限个状态  $q \in Q$ 。每个方格要么是空的 ( $\square$ )，要么是写有字符  $s \in S$  的 ( $S$  为有限集)。在每一步操作中，它可以完成：

- 磁头内部状态的转变；
- 把扫描到的符号  $s$  改写成  $s' \in S \cup \{\square\}$ ；
- 往左 ( $L$ ) 或往右 ( $R$ ) 移动磁头；

这由一个偏函数（只在定义域的一个子集上有定义的函数） $\delta : Q \times (S \cup \{\square\}) \rightarrow Q \times (S \cup \{\square\}) \times \{L, R\}$  决定（即当我们处于某个状态、读取到某个字符时，这个函数告诉我们下一个状态、应该写下的字符以及应该往哪个方向移动磁头），将这个偏函数称为图灵程序 (*Turing program*)。

每一个计算开始之前，都将磁头置于起始方格 (*starting cell*)，它是写有内容的最左侧方格，并让其处于起始状态 (*starting state*)，然后将输入放在起始方格右侧，并保证纸带其余部分为空。如果磁头抵达状态  $q_h \in Q$ ，称作停机状态 (*halting state*)，则视作计算结束，并将磁带上从起始方格开始的内容视作输出。将图灵机的构型 (*configuration*) 记作：

$$c = t_m t_{m-1} \dots t_2 t_1 \underline{q_i} s_1 s_2 \dots s_k$$

其中  $s_i$  和  $t_i$  分别是一直到最右侧（最左侧）的非空格子上的内容，当然，如果没有则可以写个  $\square$ ； $q_i$  指当前磁头所处的状态，当前磁头指向  $s_1$  所在的方格。

如果磁头进入状态  $q \neq q_h$  而  $\delta$  未定义（不移动），则称计算中断，不将其视作停机。

所谓的图灵计算 (*Turing computation*) 就指图灵机的一列构型  $c_0, c_1, \dots, c_n$ ，它按照如上的叙述由一个图灵程序所描述。

来看两个基本的图灵机的例子：

**例 10.3** 计算函数  $f(x) = x + 2$ ，如果用纸带上连续的 1 的个数表示数字大小（例如输入  $x$  就是纸带上有  $x$  个连续的 1），于是要计算这一函数，实际上是希望图灵机能在输入后面多放两个 1。可以用如

当前状态	当前读入字符	下一状态	写下的字符	磁头移动方向
$q_1$	1	$q_1$	1	R
$q_1$	□	$q_2$	1	R
$q_2$	□	$q_h$	1	R

下的图灵机状态转移函数计算这一函数：从上表可以看出  $q_1$  是起始状态，如果看到 1 则说明现在还在输入  $x$  的区域内，直到读到了一个空格 □ 时，就要开始写入 1 来做加法了：写下一个 1 就进入状态  $q_2$ ，然后  $q_2$  时再写下一个 1，此时就完成了加法，进入停机状态  $q_h$ 。

**例 10.4** 因为需要讨论判定问题，因此图灵机最终应该回答“是”或“否”。实际上就是最终图灵机会在纸带上留下一个 1 或 0 代表“是”或“否”。在有些图灵机的定义中，会规定写下 1 时代表一个  $q_{accept}$  状态表示接收，写下 0 时代表一个  $q_{reject}$  状态表示拒绝，实际上如果在进入接受和拒绝状态后再加一个转移函数到停机状态  $q_h$ ，就和前面的定义相容了。

事实上前面的定义的图灵机称为确定性图灵机 (deterministic Turing machine)，还有一种非确定性图灵机 (non-deterministic Turing machine)。所谓的确定性图灵机和非确定性图灵机的差异就在于偏函数上。按照上面的定义，它可以给每个状态都赋予一个转移，也就是说，确定原来的  $(q, s) \in Q \times (S \cup \{\square\})$ ，它就只能达到唯一的一个  $Q \times (S \cup \{\square\}) \times \{L, R\}$  中的元素。非确定性图灵机中，这个函数的值域是原来值域的幂集，也就是说，它有许多可能的路径可选，如果有一条路径可以停机，那它称就会停机。在判定性问题中，只要有一条路径接受，那么整个图灵机就接受，也就是说只有所有路径都拒绝，整个图灵机才拒绝。在后面会看到，非确定性图灵机和确定性图灵机在可计算性上是等价的，但是在复杂度上可能大相径庭。

**定理 10.5** 任何一个非确定性图灵机都可以被一个确定性图灵机模拟，即它们可以计算的函数是一致的。

**证明:**[sketch] 为了实现这一模拟，最直接的想法就是：因为非确定性图灵机执行时，每一步会产生多种选择，因此所有路径会生成一颗树，确定性图灵机只需要进行 BFS 搜索这棵树即可。

用一个三条纸带的图灵机来模拟即可：第一条纸带放输入，第二条纸带模拟非确定性图灵机读到输入的时候的状态和行动，但 BFS 事实上是有一个特定顺序的，图灵机并不知道，所以还需要第三条纸带来记录 BFS 的进程，提示第二条纸带下一步应该模拟哪条路径。这样就可以模拟非确定性图灵机的行为了。然后说明三条纸带的图灵机（更广泛的，多条纸带的图灵机）实际上与单条纸带的是等价的，这一点比较显然，最简单的想法就是把纸带分成三个部分即可。上述说明的详细版本都是比较技术性的，读者可以参考计算理论的教材。

因为非确定性图灵机的所有计算都可以被确定性图灵机模拟，所以显然二者可以计算的函数是一致的。



**定理 10.6 (Church-Turing 论题)** 存在一个现实可行的计算某个函数的方式当且仅当存在一个图灵机计算这个函数。

通俗而言就是每个算法都有一个图灵机实现。事实上 Church-Turing 论题或多或少是个猜想，所以这还是个论题，并不能称之为定理。并且这一论题不全是数学问题，其中还有丰富的物理内涵。这一论题还有一个推广形式。

**定理 10.7 (推广的 Church-Turing 论题)** 现实可行的计算系统都可以被图灵机高效地模拟。

## 10.4 Gödel 数

下面介绍一种典范的方式对形式语言进行编码。记  $\mathbb{P} \subset \mathbb{N}$  为所有素数的集合。则对于至多可数的字母表  $\Sigma$ ，可以建立双射  $f : \Sigma \rightarrow \mathbb{N}$ 。然后，我们将每个字符串映到：

$$x_1 x_2 \dots x_n \mapsto p_1^{f(x_1)} p_2^{f(x_2)} \dots p_n^{f(x_n)}$$

其中  $p_n$  表示第  $n$  个  $\mathbb{P}$  中的素数。这样就能建立一个有限长度字符串到  $\mathbb{N}$  的对应，而且它是单的（质因数分解的唯一性）。这样就建立了一个  $L \rightarrow \mathbb{N}$  的对应，其中  $L$  是任意语言。这样的对应（在更广泛的意义上讲，任何  $C \rightarrow \mathbb{N}$  的单射，其中  $C$  为可数个数学对象的集合）就被称为一个 Gödel 数。

**例 10.8** 图灵机是可以被指派 Gödel 数的，因为图灵程序的每一个分量都是可数的，所以只有可数个图灵机。记  $M_\alpha$  为编码为  $\alpha$  的图灵机。不难发现：

1. 对于任意计算，都存在无穷多个  $\alpha \in \mathbb{N}$  对应完成这种计算的图灵程序，这只要通过插入一些无意义状态就能实现。这个结论通常被称为指标集 (*indice set*) 的无限性。
2. 存在一个图灵机模拟任意编号  $\alpha$  的图灵机的执行。这个图灵机称为通用图灵机 (*universal Turing machine*)。可以表明，如果原来的图灵机计算时间是  $f(n)$ ，那么它花费的时间是  $O(f(n) \log f(n))$ ，这在后面较复杂的几节会用到。

Gödel 定义这套东西的主要目标是完成 Gödel 不完备性的证明。在此我们呈现 Tarski 的版本，称为 Tarski 不可定义性定理 (Tarski's undefinability theorem)，它蕴含了 Gödel 的结果：

**定义 10.9** 称  $T : L \rightarrow L$  是语言  $L$  中的一个谓词 (*predicate*)，如果它将字符串  $x$  映到

$$s_1 x s_2 x \dots s_{n-1} x s_n,$$

其中  $s_n \in \Sigma^*$ ，且对于任意  $x \in L$  都有  $T(x) \in L$ 。

**定理 10.10 (Tarski 不可定义性定理)** 设  $L$  是一个形式语言，存在  $\mathbb{N}$  到  $\Sigma^*$  的嵌入，记作  $\iota$ 。其中的字符串可以用  $g : L \rightarrow \mathbb{N}$  对应到其 Gödel 数，满足以下条件：

1. 存在函数  $f : L \rightarrow \{0, 1\}$ ，称为字符串的真值；
2. 存在符号  $\neg \in \Sigma$ ，使得  $f(\neg x) = 1 - f(x)$ ；
3. 对角线引理成立：对于任意谓词  $B : L \rightarrow L$ ，都有一个字符串  $a$  使得  $a = B(\iota(g(a)))$ ；

那么，不存在谓词  $T : L \rightarrow L$  满足  $f(T(\iota(g(x)))) = f(x), \forall x \in L$ 。

**证明：**反证法。根据对角线引理，考虑一个  $s \in L$ ，使得  $s = \neg T(\iota(g(s)))$ （即取  $B = \neg T$ ，先调用  $T$  然后在字符串前面加一个  $\neg$ ），然后两边应用  $f$  求真值，就有  $f(s) = 1 - f(T(\iota(g(s)))) \neq f(T(\iota(g(s))))$ 。■

其中的对角线引理暗示的是这个语言可以自指。也就是说，我们可以构造一个谓词，它的真值是它自己的 Gödel 数的真值的否定。这个定理的证明是一个典型的应用对角线法的证明，它的证明方法是构造一个谓词，使得它的真值是它自己的 Gödel 数的真值对应的谓词的否定。

这个定理意味着：

1. 不存在一个谓词  $T$  使得  $f \circ T \circ \iota \circ g$  能够判定任意一个字符串的真值；也就是说， $\iota \circ g$  的“逆”是不可完全在  $L$  中定义的；
2. Gödel 不完备性定理：一个包含基本算数（自然数及其等性）的公理系统中，存在一个不可证明也不可证伪的命题：取  $L$  为所有可能的逻辑公式，它可以叙述自然数的性质。而一个逻辑公式可以用  $g$  来编码。所以，对角线引理就是表明，我们有一个谓词  $S(n)$  表达  $\varphi(n)$  为假，其中  $\varphi$  是 Gödel 数为  $n$  的逻辑公式。然后，将  $S$  应用于它自身的 Gödel 数  $g(S)$ ，即可得出矛盾，因为若  $S(g(S))$  为真，则  $\varphi(g(S))$  为假，其中  $\varphi$  是 Gödel 数为  $g(S)$  的逻辑公式，又  $g$  是单射，故  $S = \varphi$ ，所以若  $S(g(S))$  为真，则  $\varphi(g(S)) = S(g(S))$  为假，这显然是荒谬的；
3. 停机问题：存在一个不可被图灵机计算的问题：判定任意一个图灵机是否停机。这个问题的证明是通过构造一个图灵机，它会在某个 Gödel 数  $g$  上停机当且仅当这个 Gödel 数对应的图灵机  $M_g$  在自己 Gödel 数  $g$  上不停机，这个图灵机的存在性就是对角线引理的推论。实际上，证明也是如出一辙的。

上面的证明方法，对角线法，是一种非常重要的方法。后面会在复杂度的语境下重新使用这种方法，并得出一些有意思的结果。

当然，我想提到哥德尔，大家都不会拒绝那段有趣的数学史。在二十世纪初，野心勃勃的 Hilbert 提出了一句振聋发聩的口号：我们必须知道，我们终将知道。因此，他的二十三个问题中的第二个就是算数公理系统的一致性。但是，这个命题很快被 Gödel 所证伪，虽然 Hempel 等当时知名的哲学家并不认可他的证明，Gödel 不完备性定理很快成为了一个重要的结果，并随后由 Church 的  $\lambda$ -演算和 Turing

的图灵机，进一步表明了算数公理系统不但存在不可证明的命题，也不存在一种算法来判断一个命题是否为真。进一步地，他们提出了 Church-Turing 论题，就此划定了可计算性的疆界。而后续要讨论的复杂度理论肇源于 Hartmanis 和 Stearns 的论文，在后面有所提及，而且这构成了本讲以及以后几讲中谈论复杂度理论的基本框架。另一条涉及复杂度的理论大约起源于 1970s 早期，是由 Blum 等人提出的公理化的方法，这套方法我们鲜有涉及，但也是饶有趣味的。

实际上，对复杂度的意识早在 1950 年代就有苏联科学家做出过探讨，他们探讨了暴力搜索优化问题的解的方法下界。这个问题在西方国家受到关注应当是大约 1965 年 Edmonds 的论文。有意思的是，Gödel 在他与 von Neumann 1956 年的通信中，提出了  $P = NP$  的一个原始形式。而在本讲中最具代表性的人物是 Cook (1971) 和 Karp (1972)，有兴趣的读者可以参见 Sipser 在 1992 年一篇综述 *The history and status of the P versus NP question*，其中详细解释了  $P = NP$  问题的历史，并且对 Gödel 的信件做出了翻译。

## 10.5 复杂类

在有了前面几节的铺垫后，将正式开始讨论最开始提出的问题，即如何形式化地定义问题的困难程度，也就是需要定义一些困难程度可能不同复杂类，将问题归于这些复杂类中。下面定义几个经典的复杂度类，对于读者来说，前两个类大概是最熟悉的，也是最容易接受的，但也是目前看来相当复杂的两个复杂度类。

### 10.5.1 DTIME 和 NTIME 系列

我们考虑一族函数  $f(n)$ 。称一个问题是：

1.  $DTIME(f(n))$  的，如果求解规模为  $n$  的问题的确定性图灵机能在  $f(n)$  步之内停机；
2.  $NTIME(f(n))$  的，如果求解规模为  $n$  的问题的非确定性图灵机能在  $f(n)$  步之内停机。

一个形式语言  $L$  的判定问题的规模指的是输入的长度。这两个类的定义来源于 Hartmanis & Stearns 的论文，在这里给出的形式是稍有不同的，忽略了常数的问题，只要注意到他们证明的一个加速定理：

**定理 10.11 (Hartmanis-Stearns, 1966)** 如果  $f$  可以被图灵机  $M$  在  $T(n)$  时间内计算，那么对于任意常数  $c \geq 1$ ，都有一个图灵机  $\tilde{M}$  能够在  $T(n)/c$  的时间内完成同样的计算。

**证明：**[sketch] 仅证明  $c = 2$  的情况，其余情况类似。将图灵机的磁头中的有限状态机的两步状态转移合并成一步转移，这样的话字母表也需要同步拓宽，即将原先的字符（包括  $\square$ ）两两组合。显然计算时间减半。 ■

在此基础上，定义几个复杂度类：

1.  $\text{P} = \bigcup_{k \geq 1} \text{DTIME}(n^k);$
2.  $\text{NP} = \bigcup_{k \geq 1} \text{NTIME}(n^k);$
3.  $\text{EXP} = \bigcup_{k \geq 1} \text{DTIME}(2^{n^k});$
4.  $\text{NEXP} = \bigcup_{k \geq 1} \text{NTIME}(2^{n^k}).$

结合前面的定义理解此处的定义，P 就是确定性图灵机能在多项式时间内停机解决的问题，其余类似。

基于此，不难发现  $\text{P} \subset \text{NP}$ ,  $\text{EXP} \subset \text{NEXP}$ ，因为确定性图灵机就是一种特殊的非确定性图灵机。而另一个平凡的性质  $\text{NP} \subset \text{EXP}$  则需要注意到用确定性图灵机模拟非确定性图灵机所需的开销是指数级别的，或者使用另一个等价定义：

**定理 10.12** 一个语言  $L$  是 NP 的当且仅当存在多项式  $p$  和一个多项式时间的确定性图灵机  $M$ ，使得对于任意  $x$  都有

$$x \in L \iff \exists u \in \{0, 1\}^{p(|x|)}, s.t. M(x, u) = 1.$$

$u$  被称为  $x$  关于语言  $L$  的证明 (*certificate*)。故上式的含义为  $x$  在  $L$  中当且仅当存在一个多项式长度的证明使得  $M$  在多项式时间内接受。

这个定理的证明（也就是两个 NP 的定义等价）实际上并不复杂，在此给出一个直觉描述。如果  $L$  是 NP 的，那么给出一个非确定性图灵机的选择序列即可作为证明，这个序列长度一定是多项式规模的，因为非确定性图灵机在多项式时间内至多完成多项式次分支。而验证只要通过一个确定性图灵机模拟这个选择序列下的执行即可完成。如果存在这样一个证明，那么非确定性图灵机只要不断分支，一定能在多项式时间内分支多项式次，进而凑出这样的一个证明。

除了这几个平凡的关系以外，其它层级的关系（相等还是真包含）大多还是模糊不清的（P 和 EXP 除外）。下面很快会重新回到这个问题上来。

在这一节的最后，将给出为什么 EXP 和 NEXP 类重要的一个原因：

**定理 10.13** 如果  $\text{EXP} \neq \text{NEXP}$ ，那么  $\text{P} \neq \text{NP}$ 。

证明不难，但为了缩减篇幅在此省略。这表明，指教级别的复杂度类研究实际上有助于厘清多项式时间的复杂度类之间的关系。最后给出一个例子：

**例 10.14**

$$L = \{(m, r) \mid \exists s < r, s \mid m\}$$

这个语言是 NP 的。这用等价定义很容易看出来，因为  $s$  就是所需的证明。因为问题的输入是  $m, r$ ，那么输入的长度就是  $\log m + \log r$ ，而证明  $s$  是一个小于  $r$  的数，它的长度是  $O(\log r)$  的，所以  $s$  作为

证明首先是多项式长度的。然后可以在  $\log r$  的多项式时间内做除法验证，这样就能验证给出的证明是否正确。

如果用非确定性图灵机，直观而言直接在第一步尝试所有可能的  $s$  即可。因为这些尝试只要有一个成功，那么我们就能得到一个正确的分解。

尽管知道这个问题是 NP 的，但是并不知道它是否是 NP-完全（后面会提及这一概念）的。值得一提的是，质因数分解的困难性是当前许多密码学基础设施（如 RSA 等）被认为安全的基础。因此如果  $P = NP$ ，现在的大量密码的攻击难度将会大大降低。

### 10.5.2 co-NP 类

下面这个类比较特殊，它是由形式语言的补集所定义的。实际上不妨定义一个更广泛的东西。回顾一下，一个形式语言  $L$  的补（complement），记作  $\bar{L}$ ，是所有在  $\Sigma^*$  中而不在  $L$  中的字符串构成的语言。

**定义 10.15** 设  $C$  是一个复杂度类，则  $co-C$  称为它的补类（complement class），其中的语言定义为：

$$co-C = \{L \mid \bar{L} \in C\}$$

注意， $co-C$  并不意味着在所有问题意义上的补，而是其中所有语言的补。下面的结果显然：

**定理 10.16**  $P = co-P$ 。

这是因为只要将回答中的“是”和“否”互换就可以了——这是确定性图灵机的特权。对于非确定性图灵机，问题就变得没那么简单了。如果有一条路径给出了“是”的回答，那么整体回答就是“是”；而如果简单翻转是和否，那么如果原来有一条路径给出了“否”的回答，整体的回答依然是“是”。因此， $co-NP$  的结构就显得扑朔迷离了起来。事实上，用另外一种方式给出定义可能更加轻松，这对应 NP 的等价定义：

**定理 10.17** 一个语言  $L$  是  $co-NP$  的当且仅当存在多项式  $p$  和一个多项式时间的确定性图灵机  $M$ ，使得对于任意  $x$  都有

$$x \in L \iff \forall u \in \{0, 1\}^{p(|x|)}, s.t. M(x, u) = 0.$$

注意这里把  $\exists$  换成了  $\forall$ 。实际上， $P \subset NP \cap co-NP$ ，这个结果很简单，留作读者练习。更进一步地，有  $P = NP \implies NP = co-NP$ 。这里同样给出一个例子：

**例 10.18** 在布尔公式中，所有永真式的集合是  $co-NP$  的。这是因为可以用确定性图灵机在多项式时间内完成一个布尔公式的求值。如果它是永真式，那么任意求值的结果都是 1，然后翻转 0 和 1 即可匹配上面的定义。

### 10.5.3 空间复杂度类 PSPACE

在这里，需要考虑的是图灵机的纸带花费多大的空间。同样定义一个问题：

1.  $\text{DSPACE}(f(n))$  的，如果求解规模为  $n$  的问题的确定性图灵机花费  $f(n)$  长度的纸带；
2.  $\text{NSPACE}(f(n))$  的，如果求解规模为  $n$  的问题的非确定性图灵机花费  $f(n)$  长度的纸带。

注意，非确定性图灵机花费的纸带长度是最多的那个分支花费的长度。当然，这里的“加速定理”更加显然，只需“几步并成一步”，也就是在一格纸带里多填充一些东西，扩充字母表即可。因此，这样的定义是良定的。

在这里，首先应该考虑以下结果：

**定理 10.19 (Savitch 定理)**

$$\text{NSPACE}(f(n)) = \text{DSPACE}(f^2(n))$$

这个估计的证明并不困难，只要考虑怎么用确定性图灵机模拟非确定性图灵机即可。类似地定义：

1.  $\text{PSPACE} = \bigcup_{k \geq 1} \text{DSPACE}(n^k)$ ;
2.  $\text{NPSPACE} = \bigcup_{k \geq 1} \text{NSPACE}(n^k)$ 。

但是这个时候，Savitch 定理就告诉我们， $\text{PSPACE} = \text{NPSPACE}$ 。而由于确定性图灵机的对称性，也有  $\text{co-PSPACE} = \text{PSPACE}$ 。所以，空间复杂度层级中， $\text{PSPACE}$  就成为了几乎唯一最重要的一个。当然，还有一些亚线性复杂度的问题，在此略过不表。

接下来要考虑的就是空间复杂度和时间复杂度之间的关系。很显然有

$$\text{P} \subset \text{NP} \subset \text{PSPACE} \subset \text{EXP}$$

为什么？因为一方面，图灵机在  $n$  步内之内至多能够经过  $n$  个纸带格子，所以  $\text{NP} \subset \text{NPSPACE} = \text{PSPACE}$ ；另一方面，一个图灵机如果只经过  $n^k$  个格子，那么它的构型（可能的状态）的总数只有  $O(2^{n^k})$  个，而如果它进入了完全相同的构型，那它就会循环（永不停机）。因此， $\text{PSPACE} \subset \text{EXP}$ 。但是，到目前为止，我们并不知道  $\text{PSPACE}$  和  $\text{EXP}$  之间是否是真包含关系。最后依然以一个例子收尾：

**例 10.20 (TQBF 问题)** 在全量词（所有变元都是受限变元）的布尔公式中，永真式的集合是  $\text{PSPACE}$  的。

**例 10.21 TQBF 以及 TQBF 博弈与 PSPACE 完全性**

### 10.5.4 其它有点意思的复杂度类

在 P 和 NP 中间的小复杂度类很多。其中，有一些直接以问题的名字命名，比如以后可能会提到一点的 UGC (unique game conjecture)，以及 PPAD (polynomial parity arguments on directed graphs)，后者在算法博弈论中有一些作用。在这里不会深入介绍这些复杂度类之间复杂的结构，因为这样的定义多到不可胜数。感兴趣的读者可以参照 [Complexity Zoo](#) 网站给出的图示。

## 10.6 归约与 NP 完全性

讨论了这么多抽象的复杂度类之后，终归是要回到问题层面上来。下面要讨论的依然是一个问题有多复杂。现在，已经建立的理论告诉我们，可以用一个相对简单的方法来完成描述，也就是将其看成某一个类的成员。而接下来要考虑的问题就是：

1. 在某一个类中，不同问题的难度如何比较？
2. 如何更好地确定不同的类之间的包含关系？

这两个问题就是在这一部分想要探讨的重点。我们会首先定义 Karp 归约，然后指出在一个复杂度类中，有一些问题是相对来说“最为困难的”，通过对它们的研究，就可以给出复杂度类层级的研究。然后会表明如何建立一个复杂度层级，即对角线方法，并且藉此证明一个有趣的定理：Ladner 定理。最后将通过另一种方式表述归约，并表明对角线法所面临的障碍，同时建立多项式复杂度分层。

### 10.6.1 Karp 归约

首先从一个生活中的例子开始。考虑一道数学题 A，如果告诉你它等价于 B，那么你觉得问题 A 难还是问题 B 难？从直观上，一般会说问题 A 更难，这是因为在做数学题的时候，往往把难题归约到简单的问题。但再仔细想想，这其实是有问题的：我们不能排除还有一种转化 C 使得问题 A 变成平凡的；而如果问题 B 能解决，那么问题 A 就能解决，因此，问题 A 的难度一定不超过问题 B——这是一个反直觉的事实。另一个更平易近人的例子是，如果想要在 ADS 课上考高分，那么我们可以认真学习，也可以黑了 PTA；但是，认真学习和黑了 PTA 都比考高分难，因为二者都可以保证考高分，而考高分不一定真的认真学习了，也不一定真把 PTA 黑了。这样，下面的定义就会显得比较自然：

**定义 10.22** 称一个语言  $A$  可被多项式地归约（或 *Karp 归约*，*reduction*）到  $B$ ，如果存在一个可以在多项式时间内计算的函数  $f$  使得

$$x \in A \iff f(x) \in B$$

记作  $A \leq_P B$ 。

实际上这就是说，当用解决问题  $B$  的方法去解决问题  $A$  时，可以在多项式时间内完成问题的转化，并且保证给出的答案是正确的（不会错判漏判）——这也表明了之后自己书写归约证明时的关键两步：首先写出归约方法，然后证明给出的答案是正确的（即上面的等价）。有的时候，这种函数也被称为是一个转译器（transducer）。很显然，归约关系是自反的、传递的。因此，它就自然给出了一种层级：

**定义 10.23** 考虑复杂度类  $C$ 。如果问题  $P$  满足  $\forall C \in C, C \leq_P P$ ，则称  $P$  是  $C$ -难的 ( $C$ -hard)；如果同时  $P \in C$ ，则称  $P$  是  $C$ -完全的 ( $C$ -complete)。

结合我们前面对归约中问题难度的讨论，完全问题显然就是一个复杂度类中“最复杂”的那些问题。因此不难把  $P = NP$  的问题归约成以下几个步骤：

1. 找到一个  $NP$ -完全的问题（下一节就会做到）；
2. 证明这个问题是  $P$  的（很不幸，还没人能做到）。

这样因为所有的  $NP$  问题都可以在多项式时间内归约到  $NP$ -完全的问题，因此如果  $NP$ -完全的问题可以在多项式时间内解决，那么所有的  $NP$  问题都可以在多项式时间内解决，那么  $NP$  就是  $P$  了。反之， $P \neq NP$  的问题归约成以下几个步骤：

1. 找到一个  $NP$ -完全的问题；
2. 证明这个问题不是  $P$  的（很不幸，也还没人能做到）。

当然，正如我们一直强调的，归约并不意味着一定要这么来处理这个问题，所以证明  $P \neq NP$  的方式其实也不止一种。同学们闲着没事也可以想想，万一就拿了图灵奖呢（不是）。

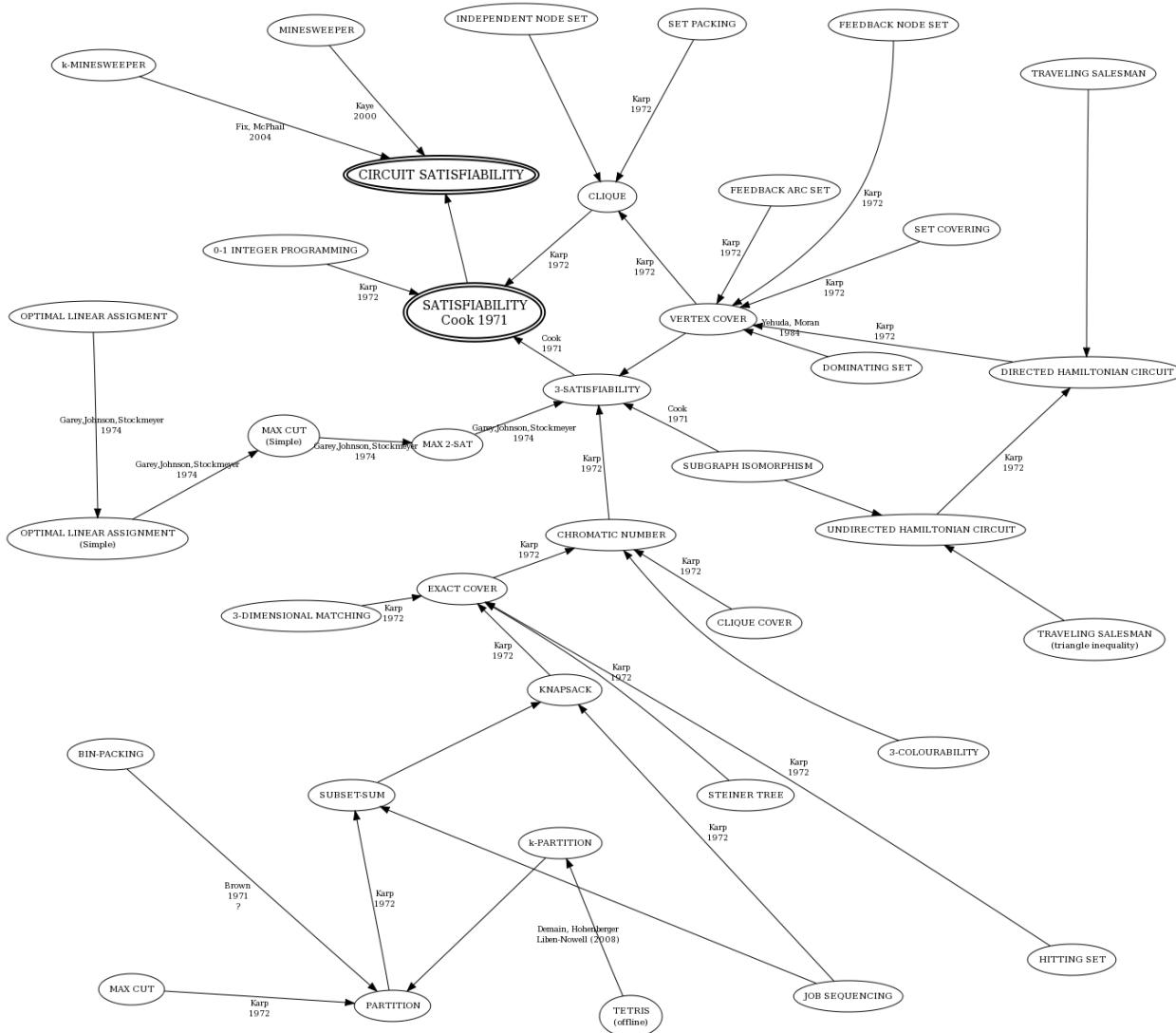
有的读者可能会疑惑，本节的标题为 Karp 归约，其中有什么背景呢？事实上，这源于 Richard M. Karp 在 1972 年发表的文章 *Reducibility among Combinatorial Problems*，其中介绍了 21 个  $NP$  完全问题之间的归约。下面的大图给出了更多的  $NP$  完全以及  $NP$  困难问题（回忆上面的定义， $NP$  困难就是指所有的  $NP$  问题都能归约到它，但它本身可能不在  $NP$  中）的归约路径。

## 10.6.2 完全问题及案例

言归正传，首先要想办法找到一个完全问题。实际上已经给出了一个例子：质因数分解（将在后面给出证明）。但是，最初的  $NP$ -完全问题实际上并不是这个，而是 Cook-Levin 定理给出的 SAT 问题——它考虑所有 CNF 公式，在其中定义语言：

$$SAT = \{L \mid L \text{ 可满足}\}$$

实际上这和最开始定义的 2-SAT 和 3-SAT 问题类似，只是这里不再限制合取范式中每个子句的长度。在清楚问题的定义后，就可以开始我们本节的一个核心定理的讲解：



**定理 10.24 (Cook 定理)** SAT 问题是 NP-完全的。

接下来的很大篇幅都将是这个定理的证明。不感兴趣的读者当然可以跳过这个部分，因为这个部分确实显得有些枯燥。显然，SAT 问题是 NP 的，所以下面要表明它是 NP-困难的，即所有 NP 问题都能归约到它：

**证明：**下面考虑任意一个 NP 中的语言  $L$ 。根据定义，存在一个非确定性图灵机能够在  $p(n)$  的时间内完成对它的判定。接下来，要为这个语言  $L$  中的元素  $x$  构造一个 CNF，形成一个多项式归约。在此为了简洁起见，仅给出对应的 CNF 的构造而不去解释这为什么是多项式的，因为这应当是自明的。

设这个图灵机在某条计算路径上得到的前  $p(n)$  个构型是  $C_0, C_1, \dots, C_{p(n)}$ ，定义以下逻辑谓词，它们都是用来描述这一串状态转移的：

1.  $S(q, t)$ : 构型  $C_t$  的状态是  $q$ ;
2.  $H(h, t)$ : 构型  $C_t$  的读写头在纸带上的第  $h$  格;
3.  $T(b, h, t)$ : 构型  $C_t$  的纸带第  $h$  格写有符号  $b$ ;
4.  $I(j, t)$ : 在  $t$  到  $t + 1$  的状态转移过程中，完成了操作  $j$ 。

这些逻辑谓词都是不那么正式的，其中的  $q, h, b, j$  实际上都以布尔变量编码。注意，因为至多用到  $O(p(n))$  个格子，所以  $h$  是有多项式的范围的。在下面的描述中，为了不那么冗长，也采用不那么形式化的写法，读者应当可以自己用这些谓词构造出下面描述中对应的 CNF。要求它满足：

1. 在时刻 0，状态是起始状态  $q_0$ ，此时的纸带只有前  $n$  格放了输入  $x$ ，其他部分都是空白;
2. 在时刻  $i$ ，状态唯一、纸带的每一格上的符号唯一、读写头的位置唯一;
3. 在时刻  $p(n)$ ，状态是接受状态;
4. 在时刻  $i$ ，没有被读写头访问的方格  $h$  上的符号与  $i + 1$  时刻相同;
5. 在时刻  $i$ ，仅有一个操作被执行，且它符合有限状态机的转移规则。

事实上这样构造的 CNF 长度为  $O(p(n)^3)$ ，它是满足要求的。这里的长度主要来源于第二条要求的唯一性的要求。例如考虑状态唯一性，设状态集合为  $\{q_1, \dots, q_k\}$ ，其中  $k$  是  $O(p(n))$  级别的（因为最多只能跑这么多步，故只能有这么多状态），那么要求某一时刻的状态唯一，其实就是要以下 CNF 成立（这显然等价于下面出现的谓词  $S(q_i, t)$  只能有一个为真）：

$$(S(q_1, t) \vee \dots \vee S(q_k, t)) \wedge \left( \bigwedge_{1 \leq i < j \leq k} (\neg S(q_i, t) \vee \neg S(q_j, t)) \right)$$

这个 CNF 的长度是  $O(p(n)^2)$  的，其它的唯一性要求同理，并且一共有  $O(p(n))$  个时刻都需要满足这样的唯一性，所以这个要求的长度是  $O(p(n)^3)$  的。

总而言之，根据前面的描述，将这一问题根据图灵机构型需要满足的要求转化为了一个 CNF，并且根据上述要求第三条显然有  $x \in L \iff$  CNF 可满足（CNF 可满足等价于存在一系列构型组成的转移路径使得最后一步接受），所以就完成了证明。 ■

下一步就是完成 Cook-Levin 定理：

**引理 10.25**  $\text{SAT} \leq_P \text{3-SAT}$ 。

**证明：**考虑  $x$  是一个 CNF，对每一个子句  $F = u_1 \vee u_2 \vee \dots \vee u_m$ ，给出一族：

$$(u_1 \vee u_2 \vee z_1) \wedge (\neg z_1 \vee u_3 \vee z_2) \wedge (\neg z_2 \vee u_4 \vee z_3) \wedge \dots \wedge (\neg z_{m-3} \vee u_{m-1} \vee u_m)$$

其中  $z_i$  是新引入的变量，不难验证这样的拆分是合理的。如果  $m < 3$ ，新引入变量即可，在此略过。这个族是可满足的当且仅当  $F$  是可满足的。 ■

结合上述定理和引理，有 Cook-Levin 定理：

**定理 10.26 (Cook-Levin 定理)** 3-SAT 是 NP-完全的。

**证明：**首先显然 3-SAT 是 NP 的。然后，由于  $\text{SAT} \leq_P \text{3-SAT}$ ，并且因为 SAT 是 NP-完全的，所以所有的 NP 问题都能归约到 SAT，根据归约的传递性可知所有的 NP 问题都能归约到 3-SAT，所以 3-SAT 是 NP-完全的。 ■

因此这一定理也给出了一种证明 NP-完全性的方法：只要证明一个问题 是 NP 的，然后证明它可以从一个已知的 NP-完全问题归约而来即可。现在，重新看一眼这个让人头晕的证明。注意到：

1. 它给出的转化  $x \in L \iff f(x) \in \text{SAT}$  实际上给出了从  $x$  的证明到  $f(x)$  的证明的一个单射，因为  $x$  的证明作为非确定性图灵机的一个分支在证明中被给出了。这种归约被称为 Levin 归约；
2. 实际上，它给出的是所有  $x$  的可能证明到所有  $f(x)$  的可能证明的集合的双射，这表明它们大小相同。这种规约被称为约俭归约 (parsimonious reduction)。

许多类似的 NP-完全问题都有从任意语言  $L \in \text{NP}$  出发的约俭 Levin 归约，这在有的时候是一个非常有用的事实。下面就是一个有意思的例子：

**推论 10.27** 如果  $P = NP$ ，那么对于任意  $L \in NP$ ，存在一个图灵机能够在多项式时间内计算出任意一个字符串  $x \in L$  的证明。

**证明：**因为  $L$  到 SAT 的归约是约俭 Levin 归约，所以只要表明对于 SAT 这个命题成立即可。为此，考虑一个确定性图灵机  $M$  在多项式时间内给出任意 CNF 的可满足性，这是  $P = NP$  的直接推论，然后将利用它来构造一个能够搜索出证明的图灵机：首先，用  $M$  来检查输入  $\varphi$  是否是可满足的，如果不是，那当然没有证明；如果是，那考虑固定第一个变量，判断剩下的是否是可满足的，以此类推。 ■

在接下来关于 NP 完全的讨论中，将看到的问题基本都是判定问题，但显然不是所有问题都是判定问题，例如讲义前面提到的搜索问题。对于一个搜索问题（最优化、排序事实上都可以视为搜索一个解的问题），可以将其转化为对应的判定问题。根据上述引理，如果要搜索某个 NP 语言中的成员（这是搜索问题），可以写下对应的判定问题。然后可以把这个 NP 语言归约到 SAT，这个归约是 Levin 归约，因此 SAT 的一个证明可以转化为该判定问题的一个证明，然后这个证明可以在多项式时间内转化为搜索问题的一个解。

举个例子，考虑团问题，有一个 NP 语言

$$\{(G, k) \mid G \text{有一个大小为 } k \text{ 的团}\}$$

有一个搜索问题：找到一个大小为  $k$  的团，对应的判定问题为  $G$  是否存在一个大小为  $k$  的团。可以将这个判定问题归约到 SAT，然后根据 Levin 归约的性质，SAT 的证明可以转化为团判定问题的证明，然后这个证明就是一个大小为  $k$  的团。

除此之外，这个证明事实上表明 SAT 是向下自归约的（downward self-reducible）。给定一个解决长度小于  $n$  的算法，就能给出解决长度  $n$  的算法。实际上，Levin 归约的性质使得所有 NP-完全问题都有类似的特性，但由于其形式化相对麻烦，在此不做过多的解释。

另外，上面对 co-NP 和 PSPACE 给出的两个例子也都是完备的。接下来给出一些比较经典的归约的例子供读者参考，体会归约的一些基本方法。

首先是 PPT 上的例子：

**例 10.28** 已知哈密顿回路问题 NP-完全的，现在要证明旅行商问题 (TSP) 也是 NP-完全的。

- 哈密顿回路问题：给定一个图  $G = (V, E)$ ，判断是否存在一个哈密顿回路，即一个遍历所有节点恰好一次的回路；
- 旅行商问题：给定一个完全图  $G = (V, E)$  和一个整数  $k$ ，判断是否存在一个哈密顿回路，使得其长度不超过  $k$ 。

证明：

1. 第一步：证明 TSP 是 NP 的。这是显然的，因为如果给了一个哈密顿回路，显然可以在多项式时间内验证它的长度是否不超过  $k$ 。
2. 第二步：证明哈密顿回路问题可以归约到 TSP，也就是说已经有一个可以解决 TSP 的万能方法，现在要解决哈密顿回路问题，要想办法把解决哈密顿回路问题转化为解决 TSP 问题，让转化而来的 TSP 返回给我的答案就是我想知道的哈密顿回路问题的答案——也就是说，存在哈密顿回路等价于转化后的旅行商问题的答案是“是”。

如何归约呢？其实是比较平凡的。当收到一个哈密顿回路的输入  $G = (V, E)$  时，为了转化为 TSP 问题，需要构造一个带权完全图  $G' = (V, E')$ ，然后把  $G$  中的边权重都设为 1，而原先不在  $G$  中的边（因为  $G$  不一定是完全图）权重设为 2。这样归约后答案是否一致呢？事实上也是显然的：

- (a) 如果  $G'$  中存在一个哈密顿回路的长度不超过  $|V|$ ，这意味着存在一个边权全为 1 的哈密顿回路，而边权为 1 表明这条边原先就在  $G$  中，因此这就意味着  $G$  中存在一个哈密顿回路；
- (b) 如果  $G'$  不存在长度不超过  $|V|$  的哈密顿回路，那么  $G$  中必定也不存在哈密顿回路，否则若存在哈密顿回路，那么  $G'$  中也存在哈密顿回路只经过边权为 1 的边，故长度不超过  $|V|$ ，与假设  $G'$  不存在长度不超过  $|V|$  的哈密顿回路矛盾，故  $G$  中必定也不存在哈密顿回路。

因此，通过上面的归约，我们把一个图  $G$  的哈密顿回路问题转化为了图  $G'$  在  $k = |V|$  的情况下的 TSP 问题，并且只要 TSP 返回是，那就的确有哈密顿回路，返回否就的确没有。并且归约的过程显然也是多项式时间的，因为只需把图赋权重即可，这样就完成了归约。 ■

PPT 的最后还给出了一个团问题和顶点覆盖问题的归约，读者可以自行阅读 PPT，应当是容易理解的。下面来看一个归约，尽管不是 NPC 问题，但是给出了一种 SAT 问题和图问题之间的归约的经典方法。

**例 10.29** 2-SAT 问题是多项式时间可解的。

**证明：**显然需要将 2-SAT 问题转化为多项式时间可解的图问题。为了转化为图，假设全部出现的逻辑变量为  $x_1, \dots, x_n$ ，于是每个变量和它的反一共有  $2n$  个变量，将它们作为图的  $2n$  个顶点即可。我们心里有一个声音，就是这些点之间边的连接情况应该就和可满足性有些联系。如何表达出来呢？事实上只需要利用一个基本的逻辑等价式：

$$x_1 \vee x_2 \iff (\neg x_1 \rightarrow x_2) \wedge (\neg x_2 \rightarrow x_1)$$

于是可以将 2-SAT 的所有  $x_i \vee x_j$  字句转化为图中的两条边，即  $(\neg x_i \rightarrow x_j)$  和  $(\neg x_j \rightarrow x_i)$ （其它形式的子句类似）。这样就把图的顶点和边都构造好了，然后只需要用多项式时间找强联通分量即可，因为表达式可满足当且仅当图中强连通分量不存在矛盾，即不会有同一个变量和它的反在同一个强连通分量中：

1. 如果对于某个  $i$  有  $x_i$  和  $\neg x_i$  在同一个强连通分量中，则意味着有一条路径从  $x_i$  到  $\neg x_i$ ，也有一条路径从  $\neg x_i$  到  $x_i$ ，根据逻辑蕴含的传递性可知， $x_i \rightarrow \neg x_i$  和  $\neg x_i \rightarrow x_i$  都成立，即有  $\neg x_i \vee \neg x_i$  且  $x_i \vee x_i$ ，这显然是不可能的；
2. 如果对于所有的  $i$ ， $x_i$  和  $\neg x_i$  都不在同一个强连通分量中，那么直接尝试赋值即可，并且一定是可以实现的，因为不会出现矛盾的赋值。

所以用多项式时间把可满足性问题转化为了图的强连通分量问题，并且强联通分量也有多项式时间算法，这样就完成了证明。 ■

### 10.6.3 对角线方法和 Ladner 定理

接下来要考虑怎么充分地建立一种复杂度分层。对角线方法 (diagonalization) 在这里是重要的——当然，在这里只能给出一些粗浅的认识，更深层次的形式化需要的抽象超越了在此可以讨论的范围。首先定义：

**定义 10.30** 称  $f$  是一个时间可构造函数 (*time-constructible function*)，如果存在一个图灵机，对于任意输入  $n$ ，可以在  $O(f(n))$  的时间内输出  $f(n)$  个 1。

这个定义粗看似乎有点莫名其妙。直观上看，最明显的反例是不可计算的函数，它当然不是时间可构造的，而可计算的反例在现在还给不出来。粗略地理解的话，这个定义想要保证的是有一台图灵机可以对这个  $f(n)$  进行计时。在下面的定理证明过程中，会注意到它的重要性：

**定理 10.31 (Time Hierarchy Theorem, THT, Seiferas-Fischer-Meyer, 1978, Žák, 1983)** 设  $f(n)$  和  $g(n)$  都是时间可构造的，如果  $f(n) = o(g(n))$ ，则  $\text{DTIME}(f(n)) \subsetneq \text{DTIME}(g(n) \log g(n))$ 。

**证明：**为了证明这个定理，首先需要考虑构造一个图灵机，然后证明它不在  $\text{DTIME}(f(n))$  中，但是在  $\text{DTIME}(g(n) \log g(n))$  中。前者，需要采用对角线方法，而后者只需要完成一个模拟。定义函数：

$$p(\alpha) = \begin{cases} 1 - M_\alpha(\alpha) & \text{如果模拟 } M_\alpha \text{ 的通用图灵机在 } g(|\alpha|) \log g(|\alpha|) \text{ 步以内停机} \\ 0 & \text{否则} \end{cases}$$

一方面， $p$  在  $O(g(n) \log g(n))$  的时间内是可计算的。这是因为直接用一台通用图灵机模拟  $M_\alpha$  的执行即可，其花费的开销正好是这个值，这样就完成了这个模拟。另一方面，假定  $p$  在  $O(f(n))$  中可以计算，花费的时间是  $c_1 f(n)$ ，用通用图灵机模拟它执行的开销是  $c_2 c_1 f(n) \log f(n)$ ，而由于  $f(n) = o(g(n))$ ，总存在一个整数  $n_0$  使得对于任意  $n \geq n_0$  都有

$$g(n) \log g(n) > c_2 c_1 f(n) \log f(n).$$

取一个  $|\beta| > n_0$  长度的能计算函数  $p$  的图灵机  $M_\beta$ ，即  $M_\beta(\alpha) = p(\alpha)$ ，它花费的时间是  $c_1 f(n)$ ，则根据上面的不等式，模拟  $M_\beta$  的通用图灵机能在  $g(|\beta|) \log g(|\beta|)$  的时间内停机，这是由假定得出的，因此用两种方式计算  $p(\beta)$ ，根据  $p$  的定义，有  $p(\beta) = 1 - M_\beta(\beta)$ ，但是因为  $M_\beta$  的定义， $p(\beta) = M_\beta(\beta) \neq 1 - M_\beta(\beta)$ 。这就产生了矛盾，因此  $p$  不可能在  $O(f(n))$  的时间内计算，即  $p \notin \text{DTIME}(f(n))$ ，但是  $p \in \text{DTIME}(g(n) \log g(n))$ ，因此  $\text{DTIME}(f(n)) \subsetneq \text{DTIME}(g(n) \log g(n))$ 。 ■

这个定理有一个直接的推论，即  $\text{P} \subsetneq \text{EXP}$ 。因此此前研究的包含关系链尽管中间的包含是否是真包含未知，但是  $\text{P}$  和  $\text{EXP}$  之间的关系是确定的。这也从某种层面上给出了为什么认为多项式时间算法是高效的算法的原因。

为了表明可构造性的意义，陈述（但不证明）以下定理：

**定理 10.32 (Borodin-Trakhtenbrot)** 对于任意一个可计算函数  $g : \mathbb{N} \rightarrow \mathbb{N}$  满足  $g(n) \geq n$ , 存在一个函数  $f : \mathbb{N} \rightarrow \mathbb{N}$  使得  $\text{DTIME}(f(n)) = \text{DTIME}(g(f(n)))$ 。

显然, 取  $g$  是时间可构造的, 如果  $f$  也是时间可构造的, 那么它和 THT 是矛盾的。对于非确定性图灵机的版本有:

**定理 10.33** 设  $f(n)$  和  $g(n)$  都是时间可构造的, 则  $\text{NTIME}(f(n)) \subsetneq \text{NTIME}(g(n))$ , 如果  $f(n+1) = o(g(n))$ 。

对于空间复杂度的版本, 有类似的空间可构造和以下定理:

**定理 10.34** 设  $f(n)$  和  $g(n)$  都是空间可构造的, 则  $\text{DSPACE}(f(n)) \subsetneq \text{DSPACE}(g(n))$ , 如果  $f(n) = o(g(n))$ 。

这几个结果的证明是类似的, 在此略过。现在以一个对角线方法不太寻常的应用收尾:

**定理 10.35 (Ladner)** 如果  $P \neq NP$ , 则一定存在一个语言  $L \in NP - P$ , 且不是 NP-完全的。

**证明:** 很显然, 如果  $P \neq NP$ , 那么 NP-完全的语言一定在  $NP - P$  中, 这个定理表明, 其中事实上不只有 NP 完全的问题。所以希望从 SAT 出发去构造一个不是 NP-完全的语言。首先定义:

$$\text{SAT}_H = \{x1^{n^{H(n)}} \mid x \in \text{SAT}, n = |x|\}$$

这个语言类看起来很寻常, 它无非就是对 SAT 做了一点加边的操作。当  $H$  有界时, 它显然也是 NP-完全的, 因为它的加边无非是多项式的。而当  $H$  无界时, 声称它不是 NP-完全的:

如果存在一个从 SAT 到  $\text{SAT}_H$  的多项式归约  $f$ , 它的时间复杂度是  $O(n^i)$ , 那么它一定是把一个长度  $n$  的表达式映到一个长度  $O(n^i)$  的表达式  $x1^{|x|^{H(|x|)}}$ , 他的长度是  $|x| + |x|^{H(|x|)}$ 。因此  $|x| = o(n)$ , 形象地说, 就将长度为  $n$  的表达式在多项式时间内越归约越短了, 这样的话最多  $n$  次归约就能完成对这个表达式的判定任务, 因为长度已经变成 1 了, 这意味着 SAT 是 P 的, 与  $P \neq NP$  的假设矛盾。

因此要做的就是构造函数  $H$ , 使得它在无界 (保证不是 NP-完全的) 的同时增长不是太快 (保证不是 P) 的。事实上, 取  $H(n)$  为最小的、小于  $\log \log n$  的整数  $i$ , 使得对于任意长度不超过  $\log n$  的字符串, 指标  $i$  的图灵机  $M_i$  都会在  $i|x|^i$  步之内停机, 而且  $M_i = 1 \iff x \in \text{SAT}_H$ , 也就是说  $i$  是判定  $\text{SAT}_H \cap \{x \mid |x| \leq \log n\}$  的图灵机的指标集中的元素。如果这样的  $i$  不存在, 那么取  $H(n) = \log \log n$ 。

接下来表明  $\text{SAT}_H \notin P$ 。如果它属于 P, 那么存在一个图灵机  $M$  在至多  $cn^c$  步中解决这个问题。这意味着存在一个整数  $i > c$  使得  $M = M_i$ 。由  $H$  的定义, 对于任意  $n > 2^{2^i}$ , 都有  $H(n) \leq i$ , 则  $n^{H(n)}$  有多项式界, 这意味着对于足够长的输入 (因为要求  $i > c$  且  $n > 2^{2^i}$ ),  $\text{SAT}_H$  就是 SAT 加上多项式长度的边, 所以这就表明 SAT 是 P 的。

为了表明它不是 NP-完全的，只要表明  $H$  在  $n$  趋于无穷时趋于无穷。也就是说，对于任意整数  $i$ ，只有有限个  $n$  使得  $H(n) = i$ 。因为  $SAT_H \notin P$ ，对于任意  $i$ ，存在一个输入  $x$  使得给定  $i|x|^i$  的时间， $M_i$  不能给出正确的结果。故结合  $H$  的定义，对于任意满足  $\log n > |x|$  的整数  $n$ ， $H(n) \neq i$ ，故只有有限个  $n$  使得  $H(n) = i$ 。 ■

## 10.7 NP 困难

之前已经提到了 NP-困难问题，它是一个非常重要的概念。很遗憾，限于时间与篇幅无法在这里展开太多，但非常建议有兴趣的同学去阅读《Algorithm Design》的第 10 章，其中介绍了一些处理 NP 困难问题的方法。在这里只给出一个例子：

**例 10.36** 回顾 0-1 背包问题：给定  $n$  个物品，每个物品有一个重量  $s_i$  和一个价值  $v_i$ ，以及一个背包容量  $C$ ，问如何选择物品放入背包使得总价值最大，但总重量不超过  $C$ 。

1. 动态规划一讲中给出了一个算法，请问那个算法是多项式时间的吗？
2. 如果每个物品的重量都小于等于  $n^2$ ，请问 0-1 背包问题是否存在多项式时间算法；
3. 0-1 背包问题判定版本：给定  $n$  个物品，每个物品有一个重量  $s_i$  和一个价值  $v_i$ ，以及一个背包容量  $C$  和一个价值  $V$ ，问是否存在一种选择使得总价值不小于  $V$  且总重量不超过  $C$ 。证明：0-1 背包问题判定问题是 NP 完全的（提示：可以利用互联网搜索任意可能的归约）。

实际上，本题表明了 0-1 背包问题是 NP 困难的，但不是强 NP 困难的。

## 10.8 致谢

在此我需要非常感谢刘泓健同学为本章讲义的绝大部分内容提供的基本框架。我们真诚地希望这几章的讲义的作用不仅仅是让读者理解课程需要掌握的内容，更重要的是在这片理论计算机科学尚不繁荣的土地上——毕竟在这里笔者感兴趣的算法博弈论都能算得上很理论的——种下一颗种子，让读者去探索更多相关的知识。也由衷感谢 Prof. Arora 和 Prof. Barak 的 *Computational Complexity: A Modern Approach* 一书，本章的许多内容都参考了这本书。

作为世界一流大学的世界一流专业，在计算机科学与技术学院没有复杂度理论的专题课程是非常令人惊讶的。这个领域本应该是计算机科学的重要组成部分，其中也反映了许多计算机科学的基本思想和方法，诞生了许多图灵奖得主（包括图灵本人、Cook 和 Karp 等我们在本节中见到的名字）。但是，我们遗憾地看到，计算机科学与技术的学生对科学一无所知，对技术也知之甚少。我们希望，这份讲义（尤其是其中的对角线方法）能够让大家体会到计算机科学本身的精妙之处，并且能够更多地了解相关的内容，比方说可计算性理论、数理逻辑等等。在这一讲中，我们尚未能明确给出数理逻辑和复杂度理论

之间的直接关联，虽然我们通过几个例子将其展现了一部分。有兴趣的读者可以寻找一些多项式层级、描述复杂度等关键词的文献，它更明显地表达了复杂度和数理逻辑之间的联系，对于自动定理证明、理论求解等领域也有诸多应用。此外，复杂度理论尚有一种更加现代化、形式化的表述方式，即所谓复杂度测度。一些物理层面的复杂度，比如 Kolgomorov 复杂度等也能在某种意义上得到统一，并且得到一些热力学的结果。此外，谈到数理逻辑，值得一提的就是与之并称的类型论和范畴论。也有一套基于范畴论的所谓综合复杂度理论（synthetic complexity theory）正在形成，虽然它远未能称得上成型。但是，这种未定的形态更适合那些有志于计算机科学的研究者，毕竟，这种仿佛来自虚空的和谐之美，是我们不变的追求。

## Lecture 11: 近似算法

编写人: 吴一航 [yhwu\\_is@zju.edu.cn](mailto:yhwu_is@zju.edu.cn)

### 11.1 基本思想与概念

上一讲讨论了 **NP** 完全性，引入了一类“很难解决”的问题，这些问题至今无法找到多项式时间的算法。因此需要做一些妥协。整体而言，对一个最优化问题的算法有如下三个期望：

1. 算法要能找到确切的最优解 (optimality)；
2. 算法能高效（通常是多项式时间）运行 (efficiency)；
3. 算法是通用的，能够解决所有问题 (all instances)。

对于 **NP** 完全问题而言，目前无法同时做到以上三点，除非  $P = NP$ ，因此需要做一些妥协。如果算法舍弃第二个期望，则是用例如回溯等方法在指数时间内解决问题，这对于输入不大的情况是可以接受的；如果舍弃第三个期望，相当于为问题找到一些容易解决的特例；如果舍弃第一个期望，但能保证高效找到的解是和真正的最优解“相差不大”的，那么称这类算法为近似算法。

本讲的目标就是介绍几个基本的近似算法的例子，从中体会近似算法的基本设计思路——因为目前的基础有限，时间也有限，因此只能接触非常基础的内容，毕竟近似算法是值得一个长学期一门大课来吸收其背后深刻内涵的主题。

首先定义近似算法的抽象的基本范式和概念：

**定义 11.1** 假设有某类问题  $\mathcal{I}$ （例如背包问题），其中的一个具体实例记为  $I$ （当背包问题的参数给定的时候即为一个实例），且有一个复杂度为多项式的近似算法  $A$ 。定义：

- $A(I)$  为算法  $A$  在实例  $I$  上得到的解；
- $OPT(I)$  为实例  $I$  的最优解。

考虑  $\mathcal{I}$  是最小化问题，若存在  $r \geq 1$ ，对任意的  $I$  都有

$$A(I) \leq r \cdot OPT(I),$$

那么称  $A$  为该问题的  $r$ -近似算法（即对于任何可能的问题实例， $A$  给出的解都不会比最优解的  $r$  倍更大）。我们特别关心其中可以取到的最小  $r$ ，称

$$\rho = \inf\{r : A(I) \leq r \cdot OPT(I), \forall I\}$$

为近似比 (*approximation ratio*, 即算法  $A$  最紧的近似界)。它可以等价定义为：

$$\rho = \sup_I \frac{A(I)}{OPT(I)},$$

即这一比值最大的实例对应的比值就是最紧的界。反之，如果是极大化问题，那么上式应该改为

$$\rho = \sup_I \frac{OPT(I)}{A(I)},$$

将两者合并起来，可以统一写作

$$\rho = \sup_I \left\{ \frac{OPT(I)}{A(I)}, \frac{A(I)}{OPT(I)} \right\}.$$

需要注意的是，上面只讨论了近似比为常数的情况。事实上有时候近似比也可能与输入规模有关，例如下面会讨论的调度问题等，但大部分情况下的结果都是常数的，因此一般而言定义都是常数。

给定一类问题  $\mathcal{I}$  和算法  $A$ ，事实上很难根据定义求出  $A$  的近似比，因为  $OPT(I)$  一般未知。因此，只能通过  $OPT(I)$  的范围来确定近似比。以最小化问题为例，确定近似比需要以下两个步骤：

1. 首先寻找一个  $r \geq 1$ ，对于任何实例  $I$ ，都有  $A(I) \leq r \cdot OPT(I)$ （可以首先寻找到  $OPT(I)$  的一个下界  $LB(I) \leq OPT(I)$ ，然后让  $A(I) \leq r \cdot LB(I)$  即可）；
2. 接下来证明  $r$  是不可改进的，即对任意的  $\varepsilon \geq 0$ ，都存在一个实例  $I_\varepsilon$ ，使得  $A(I_\varepsilon) \geq (r - \varepsilon) \cdot OPT(I_\varepsilon)$ 。

在  $P \neq NP$  的假设下，没有多项式算法解决  $NP$  问题，因此近似比不可能为 1；不过，我们希望设计近似比尽可能小（尽可能接近 1）的近似算法。那么什么样的近似是最好可能的呢？设  $|I|$  代表问题  $I$  的规模， $f$  是一个可计算 (computable) 函数，不一定为多项式函数，那么有

1. PTAS (多项式时间近似方案, Polynomial time approximation scheme)：存在算法  $A$ ，使得对每一个固定的  $\varepsilon \geq 0$ ，对任意的实例  $I$  都有

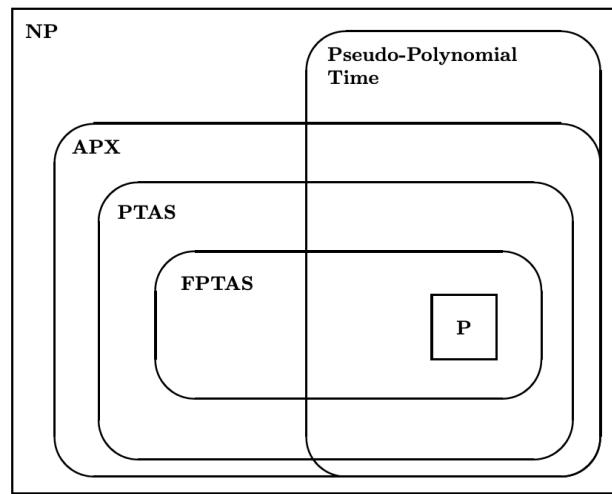
$$A(I) \leq (1 + \varepsilon) \cdot OPT(I),$$

且算法  $A$  的运行时间以问题规模  $|I|$  的多项式为上界，则称  $A$  是该问题类的一个 PTAS。

理论上  $A$  在多项式时间内可以无限近似；不过对于不同的  $\varepsilon$ ， $A$  的运行时间上界也可能不同，例如可以令  $A$  的复杂度为  $O(|I|^{1/\varepsilon})$  甚至  $O(|I|^{\exp(1/\varepsilon)})$ ，这样算法的表现就会很糟糕，因为指数过大。一般可以将 PTAS 的复杂度记为  $O(|I|^{f(1/\varepsilon)})$ 。

2. EPTAS (Efficient PTAS): 在 PTAS 的基础上, 要求算法  $A$  的复杂度是  $O(|I|^c)$  的, 其中  $c \geq 0$  是与  $\varepsilon$  无关的常数。可以将 EPTAS 的复杂度记为  $|I|^{O(1)}f(1/\varepsilon)$ 。
3. FPTAS (Fully PTAS): 在 PTAS 的基础上, 要求算法  $A$  的运行时间关于  $|I|$  和  $\varepsilon$  都呈多项式, 即可以将 FPTAS 的复杂度记为  $|I|^{O(1)}(1/\varepsilon)^{O(1)}$ 。

除了上面三者外, 还有范围最广的 APX, 代表可近似(approximable), 如果某个 NP 问题存在近似比为常数的多项式时间近似算法, 则称该问题属于 APX; 除此之外还有 QPTAS 即 Quasi PTAS, 对于每个固定的  $\varepsilon$ , 要求算法  $A$  的复杂度是  $O(n^{(\log n)^c})$  的。事实上, 除非  $P = NP$ , 否则都有  $FPTAS \subsetneq PTAS \subsetneq APX$ , 并且 APX 困难问题是没有 PTAS 的。



当面临一个具体的问题类时, 有时能得到乐观的结果, 例如:

- 0-1 背包问题存在 FPTAS;
- 欧氏空间中的 TSP 存在 PTAS。

另一方面, 在  $P \neq NP$  或者  $NP \neq ZPP$  的假设下, 也可能得到悲观的结果, 例如顶点覆盖 (Vertex Cover) 不存在 PTAS (这是一个 APX 困难问题)。

当然上面主要是通过近似比定义近似算法的优良程度, 事实上还有一种更自然却不常用的定义方式如下:

**定义 11.2** 对一个最优化问题  $I$  的一个绝对近似算法是关于问题规模  $|I|$  的多项式时间算法  $A$ , 使得对任意实例  $I$  都有

$$|A(I) - OPT(I)| \leq k,$$

其中  $k$  是一个常数。

尽管这一定义看起来非常合理，但只对极少数经典的 NP 困难优化问题才已知存在绝对近似算法，例如着色问题。

漫长的定义结束后，将开始对几个经典例子的讨论。在开启正式旅程之前，容许我感谢知乎用户金鱼马（也是 20 级图灵班的同学）对讲义的支持，前面的定义部分以及装箱问题都参考了他的[知乎专栏](#)，也很推荐对优化等领域感兴趣的读者关注他的文章。

## 11.2 最小化工时调度问题

第一个例子来源于之前贪心算法中的调度问题的变种：假设有  $n$  个作业，分别具有正的长度  $l_1, \dots, l_n$ ，有  $m$  台相同的机器。需要求出作业与机器的一种分配方案，使得耗时最长的机器的运行时间最短。

很显然的，假设一个作业能被拆分到不同机器上执行，那最优解就是在每个机器上平分，最短时间也就是  $\sum_{i=1}^n l_i / m$ 。然而要求每个作业不能被拆分，举个简单的例子，假如有两台一样的机器，作业长度分别为  $[1, 2, 2, 3]$ ，那么可以将作业分配为  $[1, 2], [2, 3]$ ，这样完成时间为 5。如果将作业分配为  $[1, 3], [2, 2]$ ，那么完成时间为 4。这一问题直接使用贪心算法无法保证最优解，所以考虑能不能有近似解。

在开始正式的讨论之前，先来看一个动态规划算法，问题的设置略有不同。将问题场景设置为只有两台不同的机器 A 和 B，第  $i$  个任务在处理机 A 上处理需要的时间为  $a_i$ ，在处理机 B 上处理的时间为  $b_i$ ，需要求耗时最长的机器的最短运行时间（称最小化的最长机器运行时间为最小工时）。

很显然的，这是原始版本  $m = 2$  的更一般化的情况。动态规划需要找到最优子结构并写出递推式。对于这个问题，可以考虑当完成第  $k$  个任务时，有两种可能：

1. A 处理机完成了第  $k$  个任务，那么 B 处理机完成  $k$  个任务的最短时间就与 B 处理机完成  $k - 1$  个任务所需的最短时间是相同的；
2. B 处理机完成了第  $k$  个任务，那么 B 处理机完成  $k$  个任务的最短时间就等于 B 处理机完成  $k - 1$  个任务的最短时间加上 B 处理机完成第  $k$  个任务所需要的时间。

设  $F[k][x]$  表示完成第  $k$  个任务时 A 耗费的时间为  $x$  的情况下，B 所花费的最短时间，其中  $0 \leq k \leq n$ ， $0 \leq x \leq \sum_{i=1}^n a_i$ ，那么，状态转移方程为

$$F[k][x] = \min\{F[k - 1][x - a_k], F[k - 1][x] + b_k\},$$

最终的结果即是完成  $n$  个任务时 A 和 B 所需时间的较大值，即  $\min_x \max\{F[n][x], x\}$ 。实际上这和 0-1 背包问题的动态规划设计思路是类似的，只是思路上绕了一些。

看到有动态规划算法，似乎找到了多项式时间的解决方案。然而只要仔细观察  $x$  的取值范围，就会发现这里其实出现了和 0-1 背包同样的问题，即  $\sum_{i=1}^n a_i$  其实是输入的指数函数，因为输入只需要  $\log \sum_{i=1}^n a_i$

级别的二进制编码长度，所以也是伪多项式的算法。事实上，即使是 A 和 B 是同样的机器（即所有作业在其上运行时间一致），也有如下结论：

**定理 11.3** 给定若干个作业，两台一样的机器，判定问题：是否存在一种分配方案使得最小工时不超过  $T$  是 NP 完全的。

**证明：**可由划分问题 (Partition problem) 规约。划分问题是 Karp 的 21 个 NP 完全问题之一，该问题表述为：给定若干个正整数，问可否将其划分成和相等的两个部分；换言之，给定  $c_1, \dots, c_n \in \mathbb{Z}^+$ ，划分问题问是否存在一个  $S \subseteq \{1, \dots, n\}$  使得  $\sum_{i \in S} c_i = \sum_{i \notin S} c_i$ 。

为了将划分问题归约到最小化工时调度问题，构造一个最小化工时调度问题的实例。设输入的作业长度为

$$\frac{2Tc_i}{\sum_{j=1}^n c_j},$$

那么这些作业的总时长为  $2T$ ，显然这些作业能用两台机器在  $T$  时间内完成当且仅当两台机器上安排的任务时长一致，这也等价于划分问题存在解。由此实现了从划分问题到最小化工时调度问题的多项式归约，因此定理中的判定问题是 NP 完全的，这也表明最小化工时调度问题是 NP 困难的。 ■

在证明了困难性后，就可以名正言顺地讨论近似算法了——除非你有自信解决  $P = NP$ 。最简单的设计近似算法的思路就是用一些看起来很有道理但不能在所有解上返回最优解的贪心算法。在这里的目标显然是  $m$  个机器上尽可能做到负载均衡，所以会自然地有如下算法：

**定理 11.4 (Graham, 1969)** 对每个作业进行调度的时候，选择当前工作量最小的机器进行调度。这样的算法近似比为  $2 - 1/m$ 。

下面就来证明这一结论。在证明近似比的时候，需要做的就是利用贪心算法的特点。

**证明：**设  $i$  表示贪心算法的调度最后得到的具有最大负载的那台机器，设其工时为  $M$ ， $j$  是分配给这个机器的最后一个作业。我们知道算法每次找到的机器的负载都是当时最小的，现在退回到选择  $j$  之前，则当时全体机器的负载之和为  $\sum_{k=1}^{j-1} l_k$ ，因此当时最小负载的机器的负载应当是小于等于  $\frac{1}{m} \sum_{k=1}^{j-1} l_k$  的。因此有

$$M \leq l_j + \frac{1}{m} \sum_{k=1}^{j-1} l_k \leq l_j + \frac{1}{m} \sum_{k \neq j} l_k,$$

将上式变形有

$$M \leq (1 - \frac{1}{m})l_j + \frac{1}{m} \sum_{k=1}^n l_k,$$

我们知道  $l_j \leq M^*$ （最优解一定比每个作业单独的长度更长），并且  $\frac{1}{m} \sum_{k=1}^n l_k \leq M^*$ ，因为这是代表了所有作业均匀分配到每台机器上的情况，一定是最优解的下界。将这两个结果带入上面的不等式有

$$M \leq (1 - \frac{1}{m})M^* + M^* = (2 - \frac{1}{m})M^*,$$

然后作为近似比一定要是下界，即要对任意的  $m$  找到一个例子是满足这一近似比的。考虑  $m$  台机器， $m(m - 1)$  个长度为 1 的作业和 1 个长度为  $m$  的作业作为输入，那么显然最优解是  $m$ ，但如果用 Graham 算法解为  $2m - 1$ ，恰好符合上述近似比。因此贪心算法的近似比为  $2 - 1/m$ 。 ■

或许这个近似比还有改进的余地：如果观察恰好符合近似比的例子，我们发现问题出在最后才把最长的作业进行调度。如果先对输入的作业根据长度从大到小排序，然后再调用 Graham 算法，对于这个例子是能得到最优解的。事实上这样先排序后贪心的算法也的确能给出更紧的近似比：

**定理 11.5** 上述先排序后调用 Graham 算法的算法得到的近似比为  $\frac{4}{3} - \frac{1}{3m}$ 。

对这一结果感兴趣的读者可以阅读蒂姆·拉夫加登的《算法详解》第四卷，这里因为篇幅和编写时间限制无法给出证明。

### 11.3 装箱问题

提到近似算法，不可能不提到装箱这一经典的问题。经典的（一维）装箱问题具体描述为：给定若干个带有尺寸的物品，要求将所有物品放入容量给定的箱子中，使得每个箱子中的物品尺寸之和不超过箱子容量并使所用的箱子数目最少。简单起见，一般将上述模型标准化：给定  $n$  尺寸在  $(0, 1]$  内的物品  $a_1, a_2, \dots, a_n$ ，目标是使用数量尽可能少的单位容量箱子装下所有物品，每个箱子中物品尺寸和都不超过 1。该问题是 NP 完全的；例如，给定若干个物品，问两个箱子是否能够装下，这个看起来简单的事情在多项式时间内不可解。

**定理 11.6** 给定若干个物品，判断它们是否可由两个箱子装下是 NP 完全的。

**证明：**再次使用划分问题。回顾一下划分问题的表述：给定若干个正整数，问可否将其划分成和相等的两个部分；换言之，给定  $c_1, \dots, c_n \in \mathbb{Z}^+$ ，划分问题问是否存在一个  $S \subseteq \{1, \dots, n\}$  使得  $\sum_{i \in S} c_i = \sum_{i \notin S} c_i$ 。

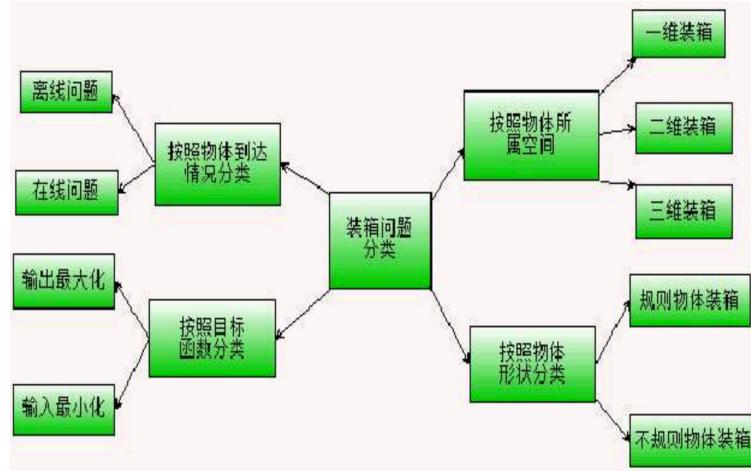
根据一个划分问题实例，可以构造一个装箱问题，令物品的尺寸为

$$a_i = \frac{2c_i}{\sum_{j=1}^n c_j},$$

同时假设这些  $a_i \in (0, 1]$ ，由于  $\sum_{i=1}^n a_i = 2$ ，那么显然这些物品能用两个大小为 1 的箱子装下当且仅当两个箱子都正好装满，这也同时解答了划分问题。这样就把一个已知的 NP 完全问题规约为了两个箱子的装箱问题。而显然一个给定的装箱问题的解可在多项式时间内验证，因此该问题是 NP 完全的。 ■

事实上上面的证明和最小化工时调度问题是类似的，其实最小化工时调度问题可以视为装箱个数固定最小化最大箱子中的物品大小的问题，因此和这里固定箱子最大的容积求最小化箱子个数的问题是对偶的。

装箱问题有极多变体，例如多维的装箱，包括几何装箱和向量装箱，以及变尺度装箱等。本节并不考虑这些变体，只介绍经典一维装箱的近似算法。一维装箱问题发展至今，其研究已经相对比较成熟。



之前衡量近似算法采用的都是“近似比 (approximation ratio)”，也被称作“绝对近似比”，定义为最坏情况下算法的求解值与问题的最优值之间的比率。对任意实例  $I$ ，用  $A(I)$  表示针对实例  $I$  运行算法  $A$  后所用箱子数； $OPT(I)$  表示装完实例  $I$  中所有物品所用的最少的箱子数。若存在常数  $\alpha \geq 1$ ，对任意实例  $I$  有  $A(I) \leq \alpha \cdot OPT(I)$ ，则  $A$  的近似比至多是  $\alpha$ 。关于一维装箱问题，有如下结论：

**定理 11.7** 除非  $P = NP$ ，否则装箱算法不存在多项式算法有小于  $3/2$  的绝对近似比。

**证明：**反证法。如果存在近似比小于  $\frac{3}{2}$  的多项式时间近似算法  $A$ ，那么直接用它判断物品是否可由两个箱子装下：

1. 如果确实可以由两个箱子装下，即  $OPT(I) = 2$ ，那么近似算法根据近似比要求返回的解必定是  $A(I) < 3$ ，但这一近似算法返回的值必定是整数，于是  $A(I) = 2$ ，于是近似算法直接就返回了正确答案；
2. 如果  $OPT(I) \geq 3$ ，那么  $A(I) \geq 3$ ，此时因为近似比低于  $3/2$ ，表明最优解不可能是 2，所以通过近似算法的解也能判断出物品是否可由两个箱子装下。

这样就回答了两个箱子的装箱问题，是通过多项式时间的归约（因为这里甚至不需要归约）到一个多项式时间的算法实现的。而它是一个  $NP$  完全问题，因此只可能有  $P = NP$ 。 ■

这里证明用到的 trick 在之后的不可近似的证明中也是常用的，其实就是通过在解一定是整数这一条件下，近似的答案只能是标准答案这样的观察来实现。当然上述结论的  $3/2$  是有算法可以恰好压线的，后面会看到，FFD (First Fit Decreasing) 算法的绝对近似比就是  $3/2$ ，不能再改进。

在装箱问题中，一般不关心全部的实例，而关心  $OPT(I)$  较大的那些实例。因此定义“渐近近似比 (asymptotic approximation ratio)”如下：

**定义 11.8** 如果对任意常数  $\alpha \geq 1$ , 对任意实例  $I$ , 存在一个常数  $k$ , 满足

$$A(I) \leq \alpha \cdot OPT(I) + k$$

称所有满足上式的  $\alpha$  的下确界为  $A$  的渐近近似比。

可以看出  $\alpha$  决定了当  $OPT(I)$  充分大时  $A(I)$  与  $OPT(I)$  的比值。 $k$  除了可以是某些固定的常数, 也可以是  $o(OPT(I))$ , 只需要在  $OPT(I)$  充分大时  $k/OPT(I) \rightarrow 0$  即可。当  $k = 0$  时渐近近似比就成了绝对近似比。

还有一个概念需要强调, 装箱问题中, 若所有的物品信息在开始装箱前已知, 则它是离线 (offline) 问题; 若初始时物品信息并不全部给出, 例如物品在传送带上逐个到达, 需要即时安排, 且对未到达物品信息一无所知, 同时做出的决定无法更改, 此时称为在线 (online) 问题。“近似比”通常用来描述离线问题近似算法的性能。而对于在线问题, 一般用“竞争比 (competitive ratio)”的概念。近似比的存在来自于计算资源有限, 而竞争比的存在来自于对问题信息所知有限。在线算法领域里, 装箱问题也是一个十分重要的问题, 不过本讲不多做介绍。PPT 第 9 页给出了关于在线算法的最差情况的一个  $5/3$  的结论 (第 8 页给出了对应的构造), 第 10 页则讨论了离线的 FFD 算法, 这一点后面会展开介绍。

这里主要讨论贪心的 Fit 算法。这是一类按照某种简单规则依次将物品放入箱子的启发式算法, 其中包括

1. Nest Fit (NF): 打开一个新箱子, 将第一个物品装进箱子。当后续物品到达时, 如果当前打开的箱子有足够的空间, 就直接放入; 否则关闭该箱子 (不再接受任何新的物品), 再打开一个新的箱子, 将该物品放进去。

- 不难看出对任意实例  $I$ ,  $NF(I) \leq 2OPT(I) - 1$ 。PPT 第 6 页给出了证明, 实际上就是利用 Next Fit 的特点: 相邻两个箱子内物品尺寸之和大于 1 (否则它们应当放到同一个箱子里), 因此如果用到了  $2OPT(I)$  个箱子, 那一定表明所有物品总的大小超过了  $OPT(I)$ , 然而每个箱子大小为 1, 如果所有物品总的大小超过了  $OPT(I)$ , 那么即使每个箱子都恰好装满也要用多于  $OPT(I)$  个箱子, 从而  $OPT(I)$  不再是最优解, 矛盾。

- 事实上装箱的渐近近似比就是 2。上界来自于上面的推导; 下界来自这样的一个实例: 所有物品的大小为  $\frac{1}{2}, \varepsilon, \frac{1}{2}, \varepsilon, \dots$ , 共有  $m$  组  $\frac{1}{2}, \varepsilon$ , 其中  $\varepsilon > 0$  充分小,  $m$  充分大, 那么 NF 解为  $m$ , 最优解为  $m/2 + 1$ 。

需要注意的是, 这种取  $\varepsilon$  来控制近似比的样例在近似算法中非常常见, 之后还会见到。

- NF 的近似效果很差, 因为其仅保持一个打开的箱子, 提前关闭的箱子之空间没有得到利用。

2. Any Fit: 这是一类 Fit 方案, 满足如下性质: 当物品到达时, 除非所有目前打开的箱子都无法装下该物品, 才允许打开一个新箱子。包括下面几种:

- First Fit (FF): 选择最早打开的箱子优先填入;
- Best Fit (BF): 选择最满的箱子 (剩余空间最小) 优先填入;

- Worst Fit (WF): 选择最空的箱子（剩余空间最大）有限填入。

前面的所有 Fit 算法都是在线算法。此外，Any Fit 的三种算法都满足相邻两个箱子物品尺寸之和大于 1，因此它们都不会比 NF 差。而前面 NF 的下界实例也适用于 WF，因此 WF 和 NF 一样差。

从定义来看 BF 算法似乎比 FF 算法有更高的箱子利用率，但并非总是如此。

**例 11.9** 考虑下面 5 个物品的实例：0.5、0.7、0.1、0.4、0.3。此实例 FF 需要 2 个箱子，而 BF 则需要 3 个；考虑下面 4 个物品的实例：0.5、0.7、0.3、0.5，此实例用 FF 需要 3 个箱子，而 BF 只需要 2 个。这两个例子说明不会存在“BF 总比 FF 好”或者“FF 总比 BF”好的结论。

与 WF 不同，FF 和 BF 算法满足一个更强的性质：设第  $k$  个箱子是当前打开箱子中剩余空间最大且最晚打开的一个。亦即，第  $k$  个打开的箱子剩余空间最大，和它有相同剩余空间的箱子打开得都比它早；那么除非当前物品无法装进前  $k - 1$  个箱子里，否则它不会装进第  $k$  个箱子里。满足此性质的算法称为 Almost Any Fit (AAF) 算法。FF 和 BF 是 AAF 算法，而 NF 和 WF 不是。

可以很容易修正 WF 将其变成 AAF 算法。修正后的算法记为 Almost Worst Fit (AWF)：将当前物品放入能装下它的剩余空间第二大的箱子中；若这样的箱子不存在，便将其放入能装它的剩余空间最大的箱子中；若上述两种情况都不存在，则新打开一个箱子将其装入。AWF 算法看起来只是对 WF 算法的微小修正，但可观察到 AWF 属于 AAF。

有如下渐近比事实：NF 和 WF 的渐近比都是 2，而任意的 AAF (FF、BF、AWF) 都可以达到相同的渐近比：1.7。特别地，对于 FF 算法，设  $FF(I)$  表示对  $I$  运用 FF 得到的装箱数。可以证明，对于任意实例  $I$ ，有

$$FF(I) \leq 1.7OPT(I) + \frac{4}{5},$$

这可以说明 First Fit 有 1.7 的渐进比。对于这一结论的证明感兴趣的读者可以阅读[这篇知乎文章](#)。

如果允许首先将所有物品从大到小排序，再使用 First Fit，这种方法叫做 First Fit Decreasing (FFD)。显然这是一种离线算法，因为要在知道所有的输入的前提下才能排序，有两个结论：

- 绝对近似比  $FFD(I) \leq \frac{3}{2}OPT(I)$ ；
- 渐近近似比  $FFD(I) \leq \frac{11}{9}OPT(I) + \frac{6}{9}$ 。

第一个证明比较简单：

**证明：**设所有的物品为  $a_1 \geq a_2 \geq \dots \geq a_n > 0$ ，考虑第  $j = \left\lceil \frac{2}{3}FFD(I) \right\rceil$  个箱子  $B_j$ ，如果它包含了一个  $a_i > 1/2$  的物体，那么  $B_j$  前面的箱子中物品体积都超过  $1/2$ ，由于排在  $a_i$  前面的物体体积都超过  $1/2$ ，因此至少有  $j$  个体积超过  $1/2$  的物品，这些物品都得放在不同的箱子里，于是

$$OPT(I) \geq j \geq \frac{2}{3}FFD(I),$$

若不然，若  $B_j$  里面没有体积超过  $1/2$  的物品，那么除了最后一个箱子  $B_{FFD(I)}$ ， $B_j$  及其之后的箱子  $B_j, B_{j+1}, \dots, B_{FFD(I)-1}$  内至少都有两个体积不超过  $1/2$  的物品。于是至少有  $2(FFD(I) - j) + 1$  个物品都无法放入  $B_1, B_2, \dots, B_{j-1}$ ，所以

$$\begin{aligned} OPT(I) &> \min \{j - 1, 2(FFD(I) - j) + 1\} \\ &\geq \min \left\{ \left\lceil \frac{2}{3} FFD(I) \right\rceil - 1, 2 \left( FFD(I) - \left( \frac{2}{3} FFD(I) + \frac{2}{3} \right) \right) + 1 \right\} \\ &= \left\lceil \frac{2}{3} FFD(I) \right\rceil - 1 \end{aligned}$$

由于  $OPT(I)$  是整数，所以  $OPT(I) > \left\lceil \frac{2}{3} FFD(I) \right\rceil - 1$  意味着  $OPT(I) \geq \left\lceil \frac{2}{3} FFD(I) \right\rceil \geq \frac{2}{3} FFD(I)$ ，即  $FFD(I) \leq \frac{3}{2} OPT(I)$ 。

另一方面， $3/2$  是紧的，考虑实例  $\left\{ \frac{1}{2}, \frac{1}{4}, \frac{1}{4}, \frac{1}{3}, \frac{1}{3}, \frac{1}{3} \right\}$  即可。 ■

这说明 FFD 是装箱问题的最优绝对近似比算法。

第二个结论证明较为复杂，见：Dósa, G. (2007). The Tight Bound of First Fit Decreasing Bin-Packing Algorithm Is  $FFD(I) \leq 11/9OPT(I) + 6/9$ . \*Combinatorics, Algorithms, Probabilistic and Experimental Methodologies\*。

除此之外，一维装箱问题还有渐进 PTAS，感兴趣的读者同样可以阅读[这篇知乎文章](#)进行初步的了解。

## 11.4 0-1 背包问题

### 11.4.1 基础的 2-近似算法

在贪心算法一节中已经看到了贪心算法解决分数背包问题的方法，实际上思想非常简单，就是让背包的每一单位重量的价值最大化。于是有一个自然的问题：对于 0-1 背包问题而言，这样的贪心算法是最优的吗？

这个问题很好回答，只需要构造一个简单的反例即可。考虑一个 0-1 背包问题，有两个物品，第一个物品的价值为 1，重量为 1，第二个物品的价值为 2，重量为 3，背包的容量为 3。贪心策略是让每单位重量的价值最大化，因此贪心算法的选择结果是选择第一个物品，但实际上最优解显然是选择第二个物品，这样背包的价值为 2，而贪心算法的解为 1。因此贪心算法并不是最优的。

事实上这是整数规划转线性规划的应用，原先 0-1 背包对应于整数规划，因为每个物品只能取 0 或 1 这样的整数个，但分数背包问题的贪心方法则可能选取分数个物品，而分数背包问题的线性规划最优解实际上也就是贪心算法对应的解。当然这只是一些思想上的说明，感兴趣的同学可以进一步了解。

为了得到近似比，需要进一步改进贪心算法：有两个贪心策略，其一是根据这里的单位重量的价值（即 PPT 上的 profit density），其二是直接贪心选择最大价值的物品，运行两个算法选择最优解，可以证明，这样结合后的近似比为 2。设分数背包问题的最优解为  $P_{frac}$ ，0 - 1 背包问题的最优解为  $P_{OPT}$ ，0-1 背包问题贪心解（即分数背包舍去选分数个的物品）为  $P_{greedy}$ ，所有装得下的物品中的最大价值为  $p_{max}$ ，那么有

$$p_{max} \leq P_{greedy} \leq P_{OPT} \leq P_{frac}.$$

第一个不等式来源于算法是取两种贪心策略的最优值，其中之一就是按照价值从大到小贪心选择的，因此价值最大的（当然前提是能放下）物品一定会被选择，所以贪心算法至少比这个价值大；第二个不等式来源于贪心策略是一个可行解，可行解一定小于等于最优解。于是近似比有上界

$$\frac{P_{OPT}}{P_{greedy}} \leq \frac{P_{frac}}{P_{greedy}} \leq \frac{P_{greedy} + p_{max}}{P_{greedy}} = 1 + \frac{p_{max}}{P_{greedy}} \leq 2.$$

其中第二个不等式是因为贪心策略是把分数背包的解的那个取了分数个的物品舍去了，而舍去的这一物品的价这部分值不可能比最大价值还大，因此有  $P_{greedy} + p_{max} \geq P_{frac}$ 。

为了说明 2 是近似比，需要构造一个例子。实际上并不困难，假设背包容量为 4，有三个物品，价值分别为  $2 + 2\epsilon, 2, 2$ ，重量分别为  $2 + \epsilon, 2, 2$ ，其中  $\epsilon$  是一个很小很小的正数。可以发现，贪心策略会选择第一个物品，而最优解是选择后两个物品，因此有近似比为 2，不可能进一步优化。

#### 11.4.2 FPTAS 算法

现在讨论一个更美好的算法，即一个可以无限近似的算法。利用动态规划一讲的算法可知 0-1 背包问题是有多项式算法的，即其复杂度是  $O(nC)$  的，其中  $n$  是物品的数量， $C$  是背包的容量。但现在可以换个角度进行动态规划，数组第二个维度不再是背包的容量，而是价值，即原先的数组  $A[i][c]$  表达的是前  $i$  个物品放入容量为  $c$  的背包的最大价值，现在的数组是  $A[i][v]$  表达的是前  $i$  个物品放入价值为  $v$  的背包的最小重量。PPT 第 13 页给出了转移方程，事实上和原先非常类似。显然，这一动态规划的复杂度是  $O(nV)$  的，其中  $V$  是所有物品的价值之和。做个简单的变换，设  $v_{max}$  是所有物品的价值的最大值，那么显然有  $V \leq nv_{max}$ ，因此复杂度是  $O(n^2 v_{max})$  的。

化成这一形式的目的非常明确，因为可以非常容易地看出，如果  $v_{max}$  的大小是  $n$  的多项式级别的，就可以得到一个多项式时间的算法。然而并非所有实例都能满足这一条件，因此需要对输入的价值做一些技术性的处理，即对输入的所有价值做同比例的缩小，使得  $v_{max}$  能缩小到  $n$  的多项式级别。

接下来需要分析这样缩小的合理性。先不管这个同比例缩小的比例是多少，姑且设为  $b$ 。设输入的价值为  $v_1, \dots, v_n$ ，缩小后的价值为  $v_1/b, \dots, v_n/b$ ，但请注意这些值并不能直接放到动态规划中运行，因为这些值不一定是整数，但动态规划算法会用到这些值作为数组的下标。因此需要将这些值统一向上取整（向下取整也可以），则得到的价值集合为  $\lceil v_1/b \rceil, \dots, \lceil v_n/b \rceil$ ，这样就可以将这些值放入动态规划中运行。

显而易见的，对于全体价值同时放缩一个比例  $b$ ，是不会影响动态规划选择的最优物品集合的（可以

看递推式，实际上就是里面出现的所有数都被放缩了一个比例，因此其中的大小关系不会变化），因此上面的动态规划算法选出的最优物品集合和当价值为  $\lceil v_1/b \rceil b, \dots, \lceil v_n/b \rceil b$  时的最优物品集合是一样的，只是后者的最优价值扩大了  $b$  倍。记  $\lceil v_i/b \rceil b$  为  $v'_i$ ，那么显然  $v'_i$  和原问题的  $v_i$  差距应当不大（后面会形式化这一差距），因此有理由相信  $v'_1, \dots, v'_n$  的最优解是  $v_1, \dots, v_n$  的最优解的一个近似。

总结一下思路：首先将全体价值缩小一个比例  $b$ ，使得最大价值是  $n$  的多项式级别的，然后做一个向上取整后调用动态规划算法得到准确的最优解  $v$  就是多项式级别的了，然后把向上取整后的价值放大回来，利用价值放缩不改变最优物品选择的特点，可知这样只是把前面动态规划得到的最优值扩大了  $b$  倍，因此  $bv$  就是放大回来后的最优解，然后因为放大回来的价值和原先的价值应当是接近的，因此有理由相信  $bv$  是原问题最优值的一个近似。

接下来就要确定比例  $b$ ，然后形式化算法并证明它的确是 FPTAS：

1. 给定想要达到的近似比率  $\varepsilon$ （为了分析方便，假设  $1/\varepsilon$  是整数，如果不是整数也可以向下找一个满足条件的数替代），令放缩比例  $b = \frac{\varepsilon v_{\max}}{n}$ ；
2. 将所有价值放缩为  $\lceil v_i/b \rceil$ ，然后运行动态规划算法得到最优解  $v$ ；
3. 然后将所有价值放大为  $v'_i = \lceil v_i/b \rceil b$ ，此时最优解为  $bv$ ，然后  $bv$  就是近似最优解了。

**定理 11.10** 上述算法是  $0-1$  背包问题的一个 FPTAS。

**证明：**首先时间复杂度是显然的，缩小价值后的的动态规划算法是  $O(n^2 v_{\max}/b) = O(n^3/\varepsilon)$  的，因此是符合 FPTAS 的。接下来需要证明这一算法的近似比是符合 FPTAS 的。

首先需要将前面的一个直观形式化，即  $v'_i = \lceil v_i/b \rceil b$  和  $v_i$  是接近的，事实上因为在向上取整时最多只会加上 1，因此有

$$v_i \leq v'_i \leq v_i + b.$$

设对于  $v_1, \dots, v_n$  而言的最优物品集合为  $S^*$ ，而对于  $v'_1, \dots, v'_n$  而言的最优物品集合为  $S$ ，那么有

$$\sum_{i \in S} v'_i \geq \sum_{i \in S^*} v'_i,$$

这是因为  $S$  是  $v'_1, \dots, v'_n$  的最优解，而  $S^*$  是一个可行解。然后结合上述舍入的不等式有如下不等式链：

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} v'_i \leq \sum_{i \in S} v'_i \leq \sum_{i \in S} (v_i + b) \leq nb + \sum_{i \in S} v_i = \varepsilon v_{\max} + \sum_{i \in S} v_i,$$

比较链首尾，事实上和目标只差一步（首是真正的最优解，尾加号后的部分是近似的解），只要能估计出  $v_{\max}$  和  $\sum_{i \in S} v_i$  的关系，就能得到近似比的上界。显然  $v_{\max} \leq \sum_{i \in S} v_i$ ，因为根据  $1/\varepsilon$  是整数的假设， $v_{\max}/b$  实际上就是一个整数，向上取整对其没有影响，因此在动态规划算法中，选取  $v_{\max}/b$  当然是一个可行解，所以最优解的价值应当大于等于  $v_{\max}/b$ ，再放大  $b$  倍回来其实就是不等式  $v_{\max} \leq \sum_{i \in S} v_i$ ，综

上有

$$\sum_{i \in S^*} v_i \leq \varepsilon v_{\max} + \sum_{i \in S} v_i \leq (1 + \varepsilon) \sum_{i \in S} v_i,$$

因此近似比满足 FPTAS 的要求。 ■

FPTAS 的存在性与所谓强 NP 困难的概念相关，所谓强 NP 困难，通俗（不够严谨）而言就是存在多项式函数使得即使把输入限制到这一多项式长度也是 NP 困难的（因此 0-1 背包显然不是强 NP 困难的）。关于强 NP 困难和 FPTAS，有如下结论：

**定理 11.11 (Garey, Johnson, 1978)** 一个具有整目标函数的强 NP 困难问题，若对某一多项式和所有实例  $I$  都满足

$$OPT(I) \leq p(\text{size}(I)), \text{largest}(I),$$

则只有当  $P = NP$  时才会存在 FPTAS。

## 11.5 聚类问题

接下来讨论的问题也是一个已经见过的问题的变种（如果读者阅读了贪心算法一讲中的聚类问题）。当时的目标是让不同聚类之间越远越好，且直接使用 Kruskal 算法来解决问题，最后剩下几个联通分支一定是相距最远的可能。但是如果转换目标，希望同一个聚类之间越近越好，那么就需要使用不同的算法来解决问题。

首先给出问题的准确描述：输入  $n$  个点的位置  $s_1, \dots, s_n$ ，以及一个整数  $k$ ，要求选择  $K$  个中心，使得每个点到最近的中心的距离最大值  $r(C)$  最小化。这个问题被称为  $k$ -中心问题 ( $k$ -center problem)。PPT 的 15 页给出了清晰的图示，16 页给出了一些距离的定义，包括  $\text{dist}(s, C)$  和  $r(C)$ ，相信理解起来还是不困难的。

直觉告诉我们，可以有一个非常简单的贪心算法（PPT 第 17 页），即把第一个点放在最合适的位置上（即使得第一个点到所有点的最大距离最小化）。PPT 上给出了这一算法合理的例子，但同时也给出了可以使得算法结果任意差的例子，因此还需要转换方向。

### 11.5.1 思维实验：如果我们已知最优解？

一个可行的解法来源于一个思维实验：假设已经知道最优解  $r(C^*)$ （也许是神谕），这是否会有帮助呢？这一利用思维实验，假设已知最优解并基于此构造算法的方式在近似算法设计中也是常见的。事实上，在知道了最优解  $r(C^*)$  后，还是无法直接得到最优的中心放置的位置，所以现在如果任意放  $K$  个中心，以  $r(C^*)$  为半径，很有可能做不到将所有点覆盖。但可以考虑这样一个事实：在知道了最优解  $r(C^*)$  后，某个点  $s$  到最优解中最近的中心  $C_s$  的距离是不超过  $r(C^*)$  的，既然任意选取  $s$  为一个中心时用  $r(C^*)$

为半径无法覆盖，那么如果放大半径，将真正最优解  $C_s$  为中心， $r(C^*)$  为半径的圆内所有点都囊括进来，那不就还是可以实现覆盖吗？

接下来的问题是，半径需要放到多大。事实上结论是显然的：在半径设置为  $2r(C^*)$  时就可以实现。这是因为  $s$  到  $C_s$  的距离为  $r(C^*)$ ，而  $C_s$  到最远的点的距离也不会超过  $r(C^*)$ ，因此  $s$  到  $C_s$  为中心， $r(C^*)$  为半径的任意点的距离都不会超过  $2r(C^*)$ 。因此可以得到 PPT 第 19 页给出的 2-近似算法（Greedy-2r）：首先设置一个最优解  $r(C^*)$ ，然后从输入点集中随机选取第一个点作为第一个中心，然后删除该点为中心， $2r(C^*)$  为半径的所有点，然后在剩余点中随机选择第二个中心，以此类推。根据前面的讨论，如果的确有神谕告诉我们最优解，那么这一算法在  $K$  步之内必然停止，且得到的解是最优解的 2 倍：

**定理 11.12** Greedy-2r 算法在给定最优解  $r(C^*)$  的情况下是一个 2-近似算法。

**证明：**因为每次都是删除选取的中心  $2r(C^*)$  为半径的圆内所有点，所以如果算法能在  $K$  步之内停止，那么得到的解一定是小于等于最优解的 2 倍，因此只需要证明算法能在  $K$  步之内停止即可。根据前面的讨论，如果最优解是  $r(C^*)$ ，那么在上面的算法中，每次随机选择一个剩余的点作为中心， $2r(C^*)$  为半径的圆至少会带走一个真正的最优解中的点为中心， $r(C^*)$  为半径的圆内所有点，因此  $K$  步之后必然最优解覆盖的所有点都被算法覆盖，因此必然停止。 ■

考虑逆否命题，这一定理及其证明表明：如果 Greedy-2r 算法在  $K$  步之内不停止，那么最优解一定不是  $r(C^*)$ （即会大于  $r(C^*)$ ）。为什么要讨论这一逆否命题呢？因为事实上并没有神谕，因此 Greedy-2r 算法输入的最优解需要自己猜，PPT 的 20 页给出了二分查找的思路，根据逆否命题不断地判断选择的  $r(C^*)$  是否合适即可。

### 11.5.2 不使用思维实验的近似算法设计

在尝到神谕的甜头后，可以来讨论一个抛弃神谕的方案：如果不知道最优解  $r(C^*)$ ，能否设计一个近似算法呢？事实上也是可以的，PPT 第 21 页给出了这一算法（Greedy-Kcenter），其基本思路是：首先从输入点集中随机选取一个点作为第一个中心，加入中心点集  $C$ 。然后每轮循环在剩余的点中找到一个点  $s$  的  $\text{dist}(s, C)$  最大，即  $s$  是到现有中心最短距离最大的点，将其加入中心点集  $C$ ，直到  $C$  中有  $K$  个点。这一算法的近似比是 2，需要依赖于前面 Greedy-2r 算法的正确性：

**定理 11.13** Greedy-Kcenter 算法是一个 2 -近似算法。

**证明：**反证法，设最优解为  $r(C^*)$ ，并假设 Greedy-Kcenter 算法给出的最优解大于  $2r(C^*)$ ，这说明 Greedy-Kcenter 算法结束后，一定存在一个点  $s$  距离所有的中心的距离大于  $2r(C^*)$ ，否则所有点都落在某个中心的  $2r(C^*)$  范围之内，最优解一定不会大于  $2r(C^*)$ 。

回顾 Greedy-Kcenter 算法的步骤，每一步都在选择一个距离现有中心  $C'$  最远的点，既然  $s$  每一步都

没有被选到，这就说明每一步选取的点  $c$  都有（假设最终的中心为  $C$ ）

$$\text{dist}(c, C') \geq \text{dist}(s, C') \geq \text{dist}(s, C) > 2r(C^*),$$

回顾 Greedy-2r 算法，在 Greedy-2r 算法中，每一步删除选择的中心的  $2r(C^*)$  范围内所有点，然后随机挑选一个剩余点作为中心。而前面得到的  $\text{dist}(c, C') > 2r(C^*)$  表明在假设的情况下（存在一个点  $s$  距离所有的中心的距离大于  $2r(C^*)$ ），Greedy-Kcenter 算法中每一步选择的点也是符合 Greedy-2r 算法的选择条件的，因为每一步都选择的是当前选过的中心  $2r(C^*)$  开外的点，所以 Greedy-Kcenter 算法选择  $K$  个点就相当于 Greedy-2r 算法中选择  $K$  个点，而根据 Greedy-2r 算法的性质， $K$  步之后还剩下点  $s$  没有被覆盖，说明真正最优解一定大于  $r(C^*)$ ，与假设矛盾。 ■

### 11.5.3 聚类问题的最优近似比

有同学可能很感兴趣，前面我们给出的算法近似比都是 2，那么这个 2 是最优的吗？

**定理 11.14** 除非  $P = NP$ ，否则  $K$ -center 问题不存在  $\rho$ -近似算法 ( $\rho < 2$ )。

为了证明这一命题，需要首先引入一个  $NP$  完全问题：dominating set 问题：给定一个图  $G = (V, E)$  和一个整数  $k$ ，需要判断是否存在一个大小为  $k$  的集合  $S \subset V$ ，使得每个点要么在  $S$  中，要么与  $S$  中的某个点相邻。现在通过将 dominating set 问题归约到寻找  $\rho$ -近似  $k$ -center 问题 ( $\rho < 2$ ) 来证明上述定理。

**【讨论】** dominating set 问题和 vertex cover 问题是一样的吗？如果不一样，能给出例子说明吗？

**证明：**为了归约，给定一个 dominating set 问题的实例  $G = (V, E)$  和整数  $k$ ，构造一个  $k$ -center 问题的实例：将  $G$  中相邻的点之间的距离设为 1，不相邻的点之间的距离设为 2，这一距离不难检验符合距离的三条定义。

显然的，此时  $k$ -center 问题的解只可能是 1 或 2（因为所有的边长度只有 1 和 2 两种情况），并且存在半径为 1 的解当且仅当 dominating set 问题存在大小为  $k$  的解（由此看出  $k$ -center 问题是  $NP$  困难的）。现在假设存在一个  $\rho$ -近似算法  $A$  ( $\rho < 2$ ) 来解决  $k$ -center 问题，则有：

1. 如果  $A$  返回的解  $r$  在  $1 \leq r \leq \rho$ ，则说明存在一个半径为 1 的解（记住解只有 1 和 2 两种可能，此时不可能是 2），根据前面的分析有 dominating set 问题存在大小为  $k$  的解；
2. 如果  $A$  返回的解  $r$  在  $2 \leq r \leq 2\rho$ ，则说明不存在一个半径为 1 的解，即 dominating set 问题不存在大小为  $k$  的解。

由于归约是多项式时间的（只需要设置一些距离），然后可以通过  $A$  来在多项式时间解决 dominating set 问题，这就说明可以在多项式时间内解决  $NP$  完全的 dominating set 问题，这就说明了  $P = NP$ 。 ■

有的同学可能会疑惑，如果证明不存在 3-近似算法（甚至任意近似比的算法），好像所有的构造都是一样的，那为什么还能得到 2-近似算法呢？这里特别要注意距离的三角不等式条件，3-近似的构造就不再满足了。

## 11.6 旅行商问题

回顾旅行商问题：给定一个完全图  $G = (V, E)$ ，每条边  $e \in E$  有一个权重  $w(e)$ ，求一条经过所有点的最短路径，即一条权重之和最小的哈密顿回路。

上一讲已经说明了旅行商问题的判定版本是一个 NP 完全问题，那么对应的旅行商问题是一个 NP 困难问题。自然地，我们希望找到一个多项式时间的近似算法来解决这个问题。然而有如下定理：

**定理 11.15** 除非  $P = NP$ ，否则对于任意的  $k \geq 1$ ，TSP 不存在  $k$ -近似算法。

类似于聚类问题，这里将哈密顿回路问题归约到 TSP 的  $k$ -近似算法：

**证明：**反证法：假设存在一个  $k$ -近似算法  $A$ ，则可以推出哈密顿回路问题有多项式时间的算法，根据哈密顿回路问题是 NP 完全问题可知  $P = NP$ 。

给定一个哈密顿回路问题的输入图  $G = (V, E)$ ，将其转化为一个 TSP 问题的实例，即一个完全图  $G'$ ，将  $G'$  的边权重定义如下：

$$w(e) = \begin{cases} 1 & e \in E \\ 2 + (k-1)n & e \notin E \end{cases},$$

其中  $n = |V|$ 。可以看到，如果  $G$  中存在哈密顿回路，则其对应的 TSP 问题的最优解为  $n$ ，否则回路长度至少为  $(n-1) + (2 + (k-1)n) = kn + 1$ 。而我们知道存在  $A$  是一个  $k$ -近似算法，使用  $A$  进行计算，则有：

1. 如果算法多项式时间内返回的解为  $n$ ，则说明  $G$  存在哈密顿回路，这一点显然；
2. 如果  $G$  存在哈密顿回路，那么  $A$  在多项式时间内也必定返回  $n$ ，这是因为  $A$  是一个  $k$ -近似算法，因此  $A$  的解至多为  $kn$ ，而其它任意解都是  $kn + 1 > kn$ 。

因此完成了从哈密顿回路到 TSP 的多项式时间归约（因为归约过程只需要对边赋权重），而用算法  $A$  可以在多项式时间内返回正确结果，这就说明了哈密顿回路问题可以在多项式时间内解决，这就说明了  $P = NP$ 。 ■

尽管这一定理的结果十分悲观，但实际上这里给出的归约构造在现实问题中并不合理：现实中的 TSP 边的权重（也就是旅行商在两点之间通行的距离）应当满足下述三角不等式：

$$w(u, v) \leq w(u, x) + w(x, v),$$

其中  $u, v, x$  是图中的任意三个点。将权重满足三角不等式的 TSP 问题称为度量 TSP 问题（因为三角不等式是度量空间的性质）。这让我们想起了上面的聚类问题，在那里也是通过度量空间的三角不等式限制从而允许了 2-近似算法的存在。那么在旅行商问题中，在添加度量空间的条件后，能得到怎样的结果呢？

**定理 11.16** 旅行商问题存在如下 2-近似算法：首先求出图  $G$  的最小生成树  $T$ ，然后在  $T$  上按深度优先搜索（先序遍历）的顶点遍历顺序得到一条哈密顿回路，其权重不超过最小生成树的权重的两倍。

**证明：**设  $H^*$  表示在给定顶点集合上的一个最优旅行路。显然最小生成树的权值是最优旅行路线代价的一个下界：

$$c(T) \leq c(H^*).$$

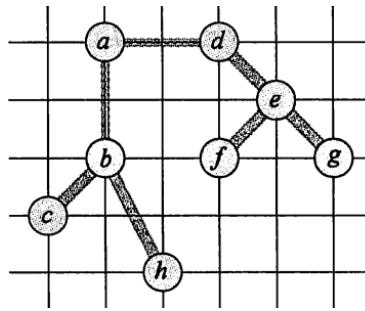
可以对  $T$  进行完全遍历，实际上就是深度优先搜索（记为  $W$ ），显然深度优先搜索恰好会经过  $T$  的每条边两次，所以有

$$c(W) = 2c(T),$$

从而有

$$c(W) \leq 2c(H^*),$$

即  $W$  的代价在最优旅行路线代价的 2 倍之内。但是， $W$  一般来说不是一个旅行路线，因为它对于某些顶点的访问次数超过一次。然而可以通过如下变换得到一条哈密顿回路（参考下面的图）：深度优先搜索（也就是先序遍历）的顺序是  $a \rightarrow b \rightarrow c \rightarrow b \rightarrow h \rightarrow b \rightarrow a \rightarrow d \dots$ ，而如果将所有顶点除了第一次访问外的访问都去掉，得到的路线为  $a \rightarrow b \rightarrow c \rightarrow h \rightarrow d \rightarrow \dots$ ，显然这样能得到哈密顿回路。并且这样的去除是不会增加原先深度优先搜索得到的解的代价的，这是由三角不等式保证的。此时不是按先序遍历进行遍历，而是按先序遍历的取点顺序每个点只取第一次出现，这样遍历的代价不超过最优解的二倍，并且得到的一定是一个哈密顿回路，所以的确得到了一个 2-近似算法。 ■



事实上上面证明的关键就是原先的先序遍历的路线就是一个不超过最优解两倍的路线，只是不是哈密顿回路罢了，然后通过三角不等式说明对路线做点的删除变成哈密顿回路是不会增加代价的即可。

事实上旅行商问题还有  $3/2$ -近似算法，需要一定的其它知识背景故不在此介绍。此外，之前讨论的都是离散图上的旅行商问题，事实上还可以定义欧氏空间中的旅行商问题，在这一问题中点与点之间的距离是由欧氏距离给出的。限于难度以及背景知识需要，只能在这里陈述结论，即欧氏空间中的旅行商问题存在多项式时间近似方案，由 Arora (1998) 和 Mitchell (1999) 给出。

## 11.7 其它问题

由于时间和篇幅限制，无法在讲义中讨论所有经典的近似算法问题。顶点覆盖和集合覆盖问题是两个同样非常经典的例子，可以参考《算法导论》或《Algorithm Design》。《算法导论》的章末习题中也有很多经典的例子（包括作业题中的近似最大生成树），当然在前面已经讨论了一些。当然，这里给出的都是在目前知识范围内能接受的例子，如果希望进一步了解更深入、复杂的近似算法设计，可以选择相关的课程或教材。

## Lecture 12: 局部搜索

编写人: 吴一航 [yhwu\\_is@zju.edu.cn](mailto:yhwu_is@zju.edu.cn)

### 12.1 基本思想

本讲将讨论一个与过往风格不太一致，但在解决最优化问题时非常通用的方法，我们称之为局部搜索。在前面的讨论中，我们经常讨论最优化问题，即目标是最大化或最小化某个目标函数——例如 0-1 背包问题，其中目标是最大化背包中的物品权重总和。

之前的讨论通常是考虑一些算法设计策略来解决这一问题，但现在请退回到一无所知的状态，将问题也退回到最原始的状态。假如我们在一座山上，想要走到这座山海拔最低的位置，那么你会怎么办呢？很显然，最自然的想法就是看看我站的这个点附近能不能再往下走，直到我无法走到更低的位置为止。

当然，很显然的问题是如果这座山是一个山顶一个山谷这样连绵起伏的，我们很有可能走到的只是一个局部的山谷，也就是局部的最低点，并非整座山脉的最低点。所以这样的搜索最优解的方式非常自然地称为**局部搜索 (local search)** ——因为找最优解的方式是看看当前位置的附近位置能不能进一步优化解，直到附近没有最优解时才会停止，并且最终搜索到的结果是一个局部最优解。PPT 第 2 页给出了一个图示，展示了类似的思想。

#### 12.1.1 自然科学中的局部搜索

相信局部搜索的想法会很容易让我们联想到物理学中的势能等概念。的确如此，在物理学中的局部搜索是非常常见的。我们知道，自然是“懒惰”的，它总是更加倾向于采纳那些能量更低的状态（最低势能），或者受到影响更小的情形（最小作用量）。因此，某个（经典物理）问题的解往往是某个局部最小值的情形。基于此，局部搜索的算法的物理背景往往可以十分清晰。比方说，模拟退火就是一个非常简单的例子（之后会展开讨论）。一个非常简单的观察来源于 Euler-Lagrange 方程的推导，即一个泛函的极值通过一个偏微分方程来描述。而这个泛函的极值本身的求法，由直接的搜索也能得出。这样的思路在计算化学中相当常见，比如说非常经典的密度泛函理论 (DFT)，其中可以选择许多组不同的“基”作为计算的前提，例如 6-31G\*\* 等，这是一种非常常用的加速计算的方法，它比从头计算 (ab initio) 方法要快上许多。

在生物学中，所谓的物竞天择就是某种更广义的局部搜索。基于 Darwin 的进化论和较为晚近的基因理论，人们发展出了进化计算 (evolutionary computation) 来解决一些问题。考虑以下对应：

生物学	计算机
环境	问题
个体	可行解
适应性	开销

这是进化计算的基本隐喻。接下来，考虑初始化一个种群（population），然后通过其中的一次选择（亲代选择，parent selection）选出亲代，然后通过基因型的重组（recombination）和变异（mutation）来产生后代（offspring），再通过子代选择（survivor selection）使得后代重新构成种群。通过这样的一代一代的循环，可以找到一个“适应环境”的种群结果，这就是遗传算法的整体思路。

接下来的问题是：如何做选择，以及如何做变异。在这个过程中，有很多比较复杂的构造。实际上，第一性的问题是，如何表达某个个体的基因型。这将其分成了许多种类，例如所谓的遗传算法（genetic algorithm）、进化策略（evolution strategy）、进化规划（evolutionary programming）和遗传规划（genetic programming）等等，这些区分是纯粹历史性的，实际上也不是非常重要，基本上，用字符串、树、向量、有限状态机等等，都是可以的。

从生物学的角度看，亲代选择（或称配偶选择）是根据个体的性质判断某个个体能否找到配偶、以及能否繁衍后代。典型的亲代选择都是概率性的，尤其是在能否找到配偶方面。而子代选择则也被称为替换策略或环境选择，它反映的是某个个体能够存活到发生繁衍的时候，它不应该在出生之后立刻死亡，等等。当然，这样的事情也是概率性的。

在进化计算的设计中，这样的选择往往是重要的。一个非常经典的笑话就来源于评估函数设计的不当：如果要让一只电子狼去在尽可能短的时间内避开障碍，追上一只仿生人梦到的电子羊，那么一只最厉害的电子狼可能是直接撞向障碍物的一只。它花费了最短的时间，抵达了狼生的终点。哪怕它没有获得抓羊的福利，减小的时间开销也足以让其成为狼群中的佼佼者，这是一个非常错误的结果，但也反映了某种有意思的现实情境。

这样的算法最常用的地方在于软件工程中的模糊测试（fuzzing），即通过一代代的测试样本变异来找出可能发生漏洞的地方，这往往也就是启发式的，比方说输入是一个结构化的东西（XML，等等），然后通过某些变异规则使得它能够大概率成为合法的测试样本，然后供分词器等完成测试，进而发现代码中可能存在的问题。

举个例子，我们希望通过进化计算来解决 0-1 背包问题。这时，候选解的表示方式非常自然，可以表示为一个二进制字符串（这就是可行解的基因型，表现型就是最后的物品选择）。有两种方法来解读一个字符串，第一个是，如果一个字符是 1，就表示它包含这个物体；反之则不包含这个物体。第二个是，如果它是 1，就表示它如果不会超过容量限制，则包含这个物体。很显然，第二个比第一个的利用率更高，这是因为其中没有“即死”的个体，即那些本身重量超过背包容量的个体，这样的个体在子代选择的时候就让他直接丢弃。但不同的“基因型”可能会对应相同的“表现型”，例如两个基因型在前几个物品上选择一致，并且此时已经塞满了，那么后面的 01 串就不会影响最终物品的选择。然后，通过现有的最优解判断亲代选择的几率，即更接近较优解的更容易留下后代。子代通过某个给定的继承概率从

亲代中继承某个对应值。当然，这是一个“无性生殖”的例子，也就是说，没有配偶选择的过程。实际上，有研究表明，有性生殖更能提升种群的基因多样性，进而有利于进一步的搜索。

### 12.1.2 全局优化的复杂度界

进行到这里，我们已经看到了局部搜索作为一个优化方法的合理性——它是一种来源于自然界的合理性。但是有的读者可能还有另一个疑惑，即为什么明知道局部搜索很可能只能带来局部最优解，在很差的情况下可能与真正的最优解相去甚远，但还是选择它呢？也许当你要找一座山脉最低点的时候你可以爬上最近的山峰先看看，然后决定往哪里走最好。但有时候山脉可能绵延数千里，你爬上了一个山峰也无法断定哪里是全局最优。当然这只是一个比喻，在解决一般优化问题时，有时候可行解的个数可能太多（一般是指数级别），因此穷举或者回溯找到全局最优解的方式非常耗时，而很多优化问题又是NP困难的，暂时也无法给出高效算法，并且近似算法可能需要精巧的设计和精妙的证明，或者有些问题本身就无法获得很好的近似比，那么此时很自然地就会求助于使用局部搜索这一非常自然的方式。

这里来看一个优化问题的例子来印证上面的说法。考虑问题  $\min_{\mathbf{x} \in \mathbb{B}^n} f(\mathbf{x})$ ，其中可行解的集合  $\mathbb{B}^n$  是  $\mathbb{R}^n$  中的一个  $n$  维盒子  $n$ -dimensional box)

$$\mathbb{B}^n = \{\mathbf{x} \in \mathbb{R}^n \mid 0 \leq \mathbf{x}^{(i)} \leq 1, i = 1, \dots, n\}$$

假如我们对  $f$  一无所知，那么显然是无法求解的。现在加入一个限制。首先设使用  $\ell_\infty$  对  $\mathbb{R}^n$  中的距离进行度量（即切比雪夫距离）

$$\|\mathbf{x}\|_\infty = \max_{1 \leq i \leq n} |\mathbf{x}^{(i)}|$$

并假设目标函数  $f : \mathbb{R}^n \mapsto \mathbb{R}$  在  $\mathbb{B}^n$  上满足利普希茨连续 (Lipschitz continuous) 条件，即对于某个常数  $L$ ，

$$|f(\mathbf{x}_1) - f(\mathbf{x}_2)| \leq L \|\mathbf{x}_1 - \mathbf{x}_2\|_\infty, \quad \forall \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{B}^n,$$

其中， $L$  称为（关于  $\|\cdot\|_\infty$  的）Lipschitz 常数。那么问题类  $\mathcal{P} = (\Sigma, \mathcal{O}, \mathcal{T}_\varepsilon)$  描述如下，

模型 $\Sigma$	$\min_{\mathbf{x} \in \mathbb{B}^n} f$ 其中 $f$ 在 $\mathbb{B}^n$ 上是 $\ell_\infty$ Lipschitz 连续的
Oracle $\mathcal{O}$	与优化目标相关的唯一局部信息是给定点能返回函数值
停止准则 $\mathcal{T}_\varepsilon$	$\bar{\mathbf{x}} \in \mathbb{B}^n, \quad f(\bar{\mathbf{x}}) - f^* \leq \varepsilon$

有一个很朴素的想法，以二维形式为例， $\mathbb{B}^2$  是一个边长为 1 的正方形；将其边长均分为  $p$  份，将正方形划分成网格，得到了  $(p+1)^2$  个交点（如果是  $n$  维，则为  $(p+1)^n$ ），接下来逐个计算这些点的函数值，选择其中最小的一个值作为  $f$  的最小值。这种方法被称作均匀网格法 (uniform grid method)，算法如下：

1. 生成  $(p+1)^n$  个点： $\mathbf{x}_\alpha = \left( \frac{i_1}{2p}, \frac{i_2}{2p}, \dots, \frac{i_n}{2p} \right)$ ，其中  $\alpha \equiv (i_1, i_2, \dots, i_n) \in \{0, 1, \dots, p\}^n$ ；
2. 在所有的点  $x_\alpha$  中找到使目标函数达到最小值的点  $\bar{\mathbf{x}}$ ；

3. 算法输出  $(\bar{\mathbf{x}}, f(\bar{\mathbf{x}}))$ 。

这是一种累计信息对测试点顺序没有任何影响的零阶优化算法。直觉上，当  $p$  足够大时，一定能找到一个足够逼近  $f$  最小值的点：

**定理 12.1** 设  $f^*$  是  $f$  的全局最优值， $\bar{\mathbf{x}}$  是均匀网格法的输出，那么

$$f(\bar{\mathbf{x}}) - f^* \leq \frac{L}{2p}.$$

证明：设  $\mathbf{x}_*$  是最小值点，那么存在  $n$  维坐标  $(i_1, i_2, \dots, i_n)$  使得

$$\mathbf{x}_1 \equiv \mathbf{x}_{(i_1, i_2, \dots, i_n)} \leq \mathbf{x}_* \leq \mathbf{x}_{(i_1+1, i_2+1, \dots, i_n+1)} \equiv \mathbf{x}_2,$$

即  $\mathbf{x}_*$  一定落在某个小立方体之内。注意到对于  $i = 1, \dots, n$ ，有  $\mathbf{x}_2^{(i)} - \mathbf{x}_1^{(i)} = \frac{1}{p}$ ，且  $\mathbf{x}_1^{(i)} \leq \mathbf{x}_*^{(i)} \leq \mathbf{x}_2^{(i)}$ 。

定义  $\hat{\mathbf{x}} = (\mathbf{x}_1 + \mathbf{x}_2)/2$ ，

$$\hat{\mathbf{x}} = \begin{cases} \mathbf{x}_2^{(i)}, & \text{if } \mathbf{x}_*^{(i)} \geq \hat{\mathbf{x}}^{(i)} \\ \mathbf{x}_1^{(i)} & \text{otherwise} \end{cases}$$

显然有  $|\hat{\mathbf{x}}^{(i)} - \mathbf{x}_*^{(i)}| \leq \frac{1}{2p}$ ， $i = 1, \dots, n$ ，即  $\mathbf{x}_*$  落在的小正方体的  $2^n$  个顶点中，一定有一个顶点，其到  $\mathbf{x}_*$  的  $\ell_\infty$  距离不大于  $\frac{1}{2p}$ ；因此，

$$\|\hat{\mathbf{x}} - \mathbf{x}_*\|_\infty = \max_{1 \leq i \leq n} |\hat{\mathbf{x}}^{(i)} - \mathbf{x}_*^{(i)}| \leq \frac{1}{2p}$$

$$f(\bar{\mathbf{x}}) - f^* \leq f(\hat{\mathbf{x}}) - f^* \leq L \|\hat{\mathbf{x}} - \mathbf{x}_*\|_\infty \leq \frac{L}{2p}$$

■

通常而言，算法需要“找到问题类  $\mathcal{P}$  有精度  $\varepsilon > 0$  的近似解，现将其定义为：寻找一个  $\bar{\mathbf{x}} \in \mathbb{B}^n$ ，使得  $f(\bar{\mathbf{x}}) - f^* \leq \varepsilon$ ”。有如下结论：

**定理 12.2** 对于问题类  $\mathcal{P}$  的方法  $\mathcal{G}$ ，其分析复杂度最多为

$$\mathcal{A}(\mathcal{G}) = \left( \left\lfloor \frac{L}{2\varepsilon} \right\rfloor + 2 \right)^n$$

因为只需要在上面的算法中取  $p = \left\lfloor \frac{L}{2\varepsilon} \right\rfloor + 1$  即可保证

$$f(\bar{\mathbf{x}}) - f^* \leq \frac{L}{2p} \leq \varepsilon,$$

而且我们需要在  $(p+1)^n$  个点调用 Oracle（即获取当前位置的函数值）。所以  $\mathcal{A}(\mathcal{G})$  就确定了问题类的复杂度的上界。可以看出，除了精度  $\varepsilon$  外，均匀网格法的分析复杂度与划分点数  $p$  与维数  $n$  有关。

但目前还存在一些问题。我们没有证明方法  $\mathcal{G}(p)$  的这个上界能否取到，也许事实上其性能会更好；另外，还可能存在比  $\mathcal{G}(p)$  更好的算法。要回答这个问题，就需要计算  $\mathcal{P}$  的复杂度下界，该下界满足

- 基于黑箱概念；
- 对于所有合理的迭代算法都有效；
- 采用对抗 Oracle (Resisting Oracle) 思想。

一个“对抗 Oracle”思想是，为每个具体方法创造一个尽可能最差的问题。也就是说，对于每一个  $x_k$ ，返回一个尽可能差的  $\mathcal{O}(x_k)$ 。但是该返回值必须与之前的答案相符，也与问题类  $\mathcal{P}$  的描述相符。基于这一思想有如下定理：

**定理 12.3** 对于  $\varepsilon < \frac{1}{2}L$ ， $\mathcal{P}$  的分析复杂度下界是  $\left(\left\lfloor \frac{L}{2\varepsilon} \right\rfloor\right)^n$ 。

**证明：**令  $p = \left(\left\lfloor \frac{L}{2\varepsilon} \right\rfloor\right)^n \geq 1$ ，假设存在一个方法，对于  $\mathcal{P}$  中的任意问题，都只需要  $N < p^n$  次对 Oracle 的调用；根据鸽巢原理，一定有一个小立方体中没有测试点，故可以从这里下手构造函数。

将该算法应用于下面的对抗策略：Oracle 在任意测试点处都返回  $f(\mathbf{x}) = 0$ 。因此，该方法只能求出  $\bar{\mathbf{x}} \in \mathbb{B}^n$ ， $f(\bar{\mathbf{x}}) = 0$ 。然而，注意到存在  $\hat{\mathbf{x}} \in \mathbb{B}^n$ ，使得

$$\hat{\mathbf{x}} + \frac{1}{p}\mathbf{e} \in \mathbb{B}^n, \quad \mathbf{e} \in (1, \dots, 1)^T \in \mathbb{R}^n$$

且在立方体  $\mathbb{B} = \left\{ \mathbf{x} \mid \hat{\mathbf{x}} \leq \mathbf{x} \leq \hat{\mathbf{x}} + \frac{1}{p}\mathbf{e} \right\}$  内没有任何测试点。令  $\mathbf{x}_* = \hat{\mathbf{x}} + \frac{1}{2p}\mathbf{e}$ 。考虑函数

$$f_p(\mathbf{x}) = \min\{0, L\|\mathbf{x} - \mathbf{x}_*\|_\infty - \varepsilon\}$$

易知  $f_p$  满足  $\ell_\infty$ -Lipschitz 连续，其 Lipschitz 常数为  $L$ ，全局最优为  $-\varepsilon$ 。然而， $f_p$  仅在立方体  $\mathbb{B}' = \left\{ \mathbf{x} \mid \|\mathbf{x} - \mathbf{x}_*\|_\infty \leq \frac{\varepsilon}{L} \right\}$  内不为 0，由于  $2p \leq L/\varepsilon$ ，因此  $\mathbb{B}' \subseteq \mathbb{B}$ 。

而在我们的方法中，对于所有的测试点， $f_p(\mathbf{x})$  都会返回 0。由于我们方法的精度为  $\varepsilon$ ，可以得出结论：如果 Oracle 的调用次数小于  $p^n$ ，那么结果的精度不可能小于  $\varepsilon$ 。定理得证。 ■

现在可以对  $\mathcal{G}$  进行总结

$$\mathcal{G} : \left( \left\lfloor \frac{L}{2\varepsilon} \right\rfloor + 2 \right)^n, \quad \text{下界 } \left( \left\lfloor \frac{L}{2\varepsilon} \right\rfloor \right)^n$$

如果  $\varepsilon \leq O\left(\frac{L}{n}\right)$ ，那么上界和下界是一致的；因此对于问题类  $\mathcal{P}$ ，方法  $\mathcal{G}(p)$  是最优的。因此如果执意要对一个函数求解全局最优（在如上条件下），那么时间复杂度的底线也将是指数级别的。因此会求助于局部搜索，以更快速、更精妙的方式找到一个局部最优解。

## 12.2 第一个例子：顶点覆盖问题

### 12.2.1 局部搜索的范式

在讨论了局部搜索的大致思想后，下面开始正式的讨论。在正式讨论开始之前，先将前面的大致思想形式化。首先分别讨论局部搜索中的“局部”和“搜索”两个关键词：

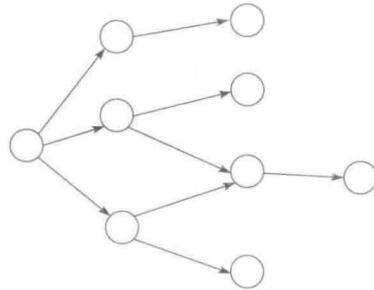
1. 为了定义“局部”，实际上就是定义“邻居（neighborhood）”这一概念。假设  $S$  是一个可行解，所谓  $S$  的邻居就是一个集合

$$N(S) = \{S' \mid S' \text{ 是 } S \text{ 经过一个小的更改之后得到的解}\},$$

这里“小的更改”需要算法设计者自己根据问题的特点定义，例如背包问题可以设置为进行一个物品的替换，旅行商问题可能是交换两个顶点的途径顺序等等；于是所谓的解  $S$  的局部也就是  $S$  的邻居集合  $N(S)$ ；

2. 所谓的搜索就是从任意一个可行解开始，不断在当前所处的解的局部（也就是邻居）中选择一个最优解，直到无法在局部找到更优解为止，这样就能得到一个局部最优解。

可以对局部搜索做一个可视化的理解，展示为下图中的有向图上的一个行走。图中每条有向边表示一个更优的局部移动，并且显然这些局部移动间不会有环，所以下图是一个有向无环图。图中没有出边的节点（吸收点）表示局部最优。局部搜索算法事实上就是在图中不断地沿着出边进行行走，直到进入一个吸收点为止。



### 12.2.2 顶点覆盖问题

在讲完了抽象的范式之后，马上来看一个具体的基础的例子进行应用。考虑顶点覆盖问题：给定一个无向图  $G = (V, E)$ ，找到一个最小的顶点集合  $S \subseteq V$ ，使得对于每一条边  $(u, v) \in E$ ，至少有一个顶点在  $S$  中。

这个问题是一个 NP 困难问题，这是在 NP 问题一讲中给出的结果，也就是说暂时没有找到一个多项式时间的算法来解决这个问题。在给出局部搜索解前先简单回顾一下近似算法。这里考虑一个简单的

贪心算法（记为算法  $A$ ）：任意一条边  $(u, v)$ ，然后将  $u$  和  $v$  同时加入到  $C$  中，然后把  $u$  和  $v$  所在的所有边全部移除，下一轮继续在剩下的边中随机选择一条重复上述操作，直到所有的边都被删除为止。

**定理 12.4** 上述顶点覆盖问题的贪心算法  $A$  是一个 2-近似算法。

**证明：**设  $S$  为算法  $A$  运行过程中选出的所有边的集合。为了覆盖  $S$  中的边，任意一个顶点覆盖（特别是最优覆盖  $C^*$ ）都必须至少包含  $S$  中每条边的一个端点。又因为每次把边选进  $S$  后就会把所有与选入的边的端点相关联的所有边，所以  $S$  中不存在两条边具有共同的端点，所以有

$$|C^*| \geq |S|.$$

设算法  $A$  选出的顶点覆盖为  $C$ ，因为一条边的两个端点都被选入到  $C$  中，并且没有两条边共同的端点，所以

$$|C| = 2|S|.$$

于是有

$$|C| = 2|S| \leq 2|C^*|,$$

■

再来看看一下上述证明过程。事实上与上一讲中见到的问题类似，我们需要在不知道最优顶点覆盖的规模到底是多少的情况下证明算法  $A$  所返回顶点覆盖的规模至多为最优顶点覆盖规模的 2 倍。为此，我们巧妙地利用了最优顶点覆盖规模的一个下界（就是贪心算法选出的边数），从而使证明过程不牵涉最优顶点覆盖的实际规模。

接下来开始介绍局部搜索算法。显然首先需要定义这一问题中的邻居是什么。在顶点覆盖问题中，目标是找到一个最小的顶点覆盖，然后需要每一步在邻居中找一个更优解，所以不妨让邻居就是比当前解少一个顶点的解。这样就可以定义局部搜索算法了：选择的初始解就是图  $G$  的所有顶点，即  $S = V$ ，然后每次从  $S$  中随机选择一个顶点  $v$ ，然后将  $v$  从  $S$  中移除，检查新的解  $S - \{v\}$  是否是一个顶点覆盖，如果是则继续后续删除操作，否则就得到了一个局部最优解。

称这一算法为“梯度下降法”。相信读者对这一词汇并不陌生，它是求解一个函数的最小值时方法，常用于机器学习、优化理论中。所谓的梯度下降就是因为梯度反映的是函数下降最快的方向，因此沿着梯度方向寻找局部更优的解是合理的。在这里借用了其中的思想，这里的梯度下降其实就是任选一个点从现有解中删除。

PPT 第 6 页给出了“梯度下降法”合适与不合适的例子。对于第一种情况而言，实际上就对应于局部最优与全局最优完全等价的情况；第二种则表明局部最优（选择外面一圈的所有点）相比于全局最优（选择中心点）可以任意差；第三种则是有很多种情况，可以通过不同的下降选择找到不同的局部最优解。由此也看出局部搜索与近似算法的一个很重要的区别，即局部搜索得到的解可能是没有任何近似比保证的——局部搜索只是一种启发式的算法。所以接下来希望对这一简单的方法做一些改进：我们希望有时候我们能跳出一些很差的局部最优解，换条路继续搜索。

### 12.2.3 Metropolis 算法与模拟退火

这种改进的方法由 Metropolis, Rosenbluth 和 Teller 于 1953 年提出，被称为 Metropolis 算法。他们希望利用统计力学中的原理模拟物理系统的行为。在统计物理中有一个假设，当一个系统的能量为  $E$  时，它出现的概率为  $e^{-E/kT}$ ，其中  $T > 0$  是温度， $k$  是玻尔兹曼常数。这个假设被称为 Boltzmann 分布。显然的，当  $T$  固定时，能量越低的状态出现的概率越大，因此一个物理系统也有更大的概率处于能量低的状态。然后考虑温度  $T$  的影响，当  $T$  很大时，根据指数函数的特点，不同能量对应的概率其实差别可能不是很大；但  $T$  较低的时候，不同的能量对应的概率差别就会很大。

基于玻尔兹曼分布，可以将 Metropolis 算法描述如下：

1. 初始化：随机选择一个可行解  $S$ ，设  $S$  的能量为  $E(S)$ ，并确定一个温度  $T$ ；
2. 不断进行如下步骤：
  - (a) 随机选择  $S$  的一个邻居  $S'$ （可以按均匀分布随机选择）；
  - (b) 如果  $E(S') \leq E(S)$ ，则接受  $S'$  作为新的解；
  - (c) 如果  $E(S') > E(S)$ ，则以概率  $e^{-(E(S') - E(S))/kT}$  接受  $S'$  作为新的解；如果接受了则更新解，继续下一轮迭代，否则保持原解  $S$ ，继续迭代。

直观地说，Metropolis 算法就是在当前解  $S$  的邻居中随机选择一个解  $S'$ ，如果  $S'$  的能量更低则接受，否则以一定概率接受一个更差的解——这就使得我们有可能跳出一个局部最优解。当然有一个注意的问题是，上面没有写算法的停止点，实际上进行到一个满意的结果中断算法即可。Metropolis 等人证明了他们的算法具有如下性质（证明不属于这门课的范畴，涉及一些简单的随机过程）：

**定理 12.5** 设  $Z = \sum_S e^{-E(S)/kT}$ 。对于任一状态  $S$ ，记  $f_S(t)$  为在  $t$  轮迭代中选到了状态  $S$  的比例，则当  $t \rightarrow \infty$  时， $f_S(t)$  将以概率 1 收敛于  $e^{-E(S)/kT}/Z$ 。

这一结论表明，Metropolis 算法在足够长的时间后，在每个状态上停留的时间比例将和玻尔兹曼分布近似，也就是说这一算法成功模拟了玻尔兹曼分布，有更大的可能停留在低能量状态，即有更大的概率获得很好的解。

PPT 第 7 页给出了针对顶点覆盖问题的 Metropolis 算法。对于 PPT 第 6 页的 case 1，有了 Metropolis 算法，就有可能跳出局部最优解，找到全局最优解。然而如果看 case 0，就会发现当进行到没剩下几个点的时候，这时因为是均匀选取下一个状态，那么有非常大的概率会抽到一个比当前解差的解，然后又有一定概率接受这个解，这样就会偏离最优解（一个点都不选），甚至陷入一种在几个解之间来回跳但就是找不到最优解的境地，但是不用 Metropolis 算法都能解出最优解，现在反而解不出了，这是很难让人接受的，因此还需要进一步优化 Metropolis 算法。

事实上还有一个关键的因素没有考虑：Metropolis 算法的一个重要参数是温度  $T$ 。之前也分析了，当温度很高的时候，不同能量对应的概率差别不大，因此高温非常适合于跳出局部最优解；而当温度很低的

时候，不同能量对应的概率差别很大，这时候就有更大的概率进入好的解。因此我们希望在开始的时候温度很高，然后逐渐降低温度，这样就能在开始的时候跳到一个比较好的局部，然后在降温后降到能到的最低点。这就是模拟退火（Simulated Annealing）的思想，它的思想来源于模拟降温结晶过程。有一个有趣的问题，就是为什么不一开始就选择低温，因为前面的定理表明当  $t \rightarrow \infty$  时，Metropolis 算法的解会收敛到玻尔兹曼分布，这样就能大概率找到很好的解。但在实际中发现温度较低的时候这一过程会非常漫长，所以是不切实际的，因此采取先用高温偏向于好的局部，然后降温找到最优解这样的过程。当然模拟退火也不能保证一定能找到最优解，毕竟一切选择都是概率性的。

## 12.3 旅行商问题

在顶点覆盖中，直观感觉是局部搜索就是一种启发式的、没有效果保证的算法，那么接下来的几个例子会给出一些有保证的局部搜索算法：我们会看到局部最优解就是想要的解的情况，还会看到局部最优解是有可证明的近似比的解。接下来要看的问题是老熟人：旅行商问题，当然这里是建立在上一讲的 2-近似算法的基础上做进一步的局部搜索改进，所以自然也是一个近似比有保证的算法。

这已经是第三次遇见旅行商问题了，这次将从局部搜索的角度提供一种解决方案。首先来回顾一下旅行商问题的定义：给定一个无向完全图  $G = (V, E)$ ，每条边  $(u, v) \in E$  有一个权重  $w(u, v)$ ，希望找到一个最短的回路，使得每个顶点恰好被访问一次。这个问题是一个 NP 困难问题，暂时没有找到一个多项式时间的算法来解决这个问题。

### 12.3.1 一个准确的动态规划算法

在正式讨论局部搜索算法前，我希望在这里先补全一个之前遗漏的关于旅行商问题的讨论。事实上对于这类 NP 困难问题，之前提到过，在输入不算太大的时候也可以接受指数时间的精确算法。回顾 0-1 背包问题和最小化工时调度问题，我们都提到了动态规划算法，这是一个可以支持指级别精确算法的比较精巧算法设计策略（回溯过于暴力了），所以也可以尝试用动态规划来解决旅行商问题。

动态规划的关键在于找出最优子结构。仍然用一个思维实验来开始讨论，假如上帝给了你一个解，那么上帝自己一定是一个一个点慢慢选出来的，但你只能从最后选的那个点观察一些选点的原则。假设给所有顶点编号  $1, \dots, n$ ，然后我们的目标是找一条最短的经过每个点恰好一次从 1 回到 1（从任意点到自己都可以，但其实是等价的，所以用 1 举例子说明）的哈密顿回路。那么这个路径的最后一跳应当是从某个点  $j \neq 1$  到 1，那么问题就转换为了找最优的  $j$  作为最后一跳，即

$$\text{最优路线的成本} = \min_{j=2, \dots, n} \text{访问每个顶点的无环 } 1 \rightarrow j \text{ 路径的最低成本} + c_{j1},$$

其中用  $c_{ij}$  表示从  $i$  到  $j$  的成本。

于是问题变成了怎么找到从 1 出发到任意顶点  $j \neq 1$  的最短的经过每个顶点恰好一次的路径。如果能解决这一问题，那么再代入上面的表达式即可求出旅行商的最短路径。这里就要用最短路问题自带的最

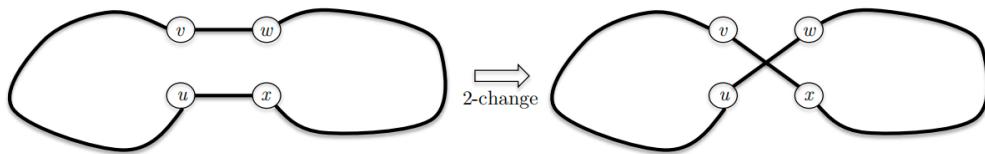
优子结构来设计动态规划算法了（在 Bellman-Ford 和 Floyd-Warshall 算法中已经实践过）：即最短路径的某一段也一定是对应的起点终点的最短路径。因此如果要求从 1 到  $j$  的最短路径，可以考虑最后一跳的选取，假设最短路径最后一跳是从  $k$  到  $j$ ，那么可以将问题分解为从 1 到  $k$  的最短路径加上从  $k$  到  $j$  的成本，当然我们并不能直接知道哪个  $k$  最好，所以还需要遍历所有的  $k$ ，这样就决定了最后一跳。然后递归解决倒数第二跳，以此类推。不难发现，这样的动态规划算法的子问题可以表达为  $C_{S,j}$ ，表示从 1 到  $j \in S$  经过  $S$  中所有点恰好一次的最短路径。这样就可以写出动态规划的递推式：

$$C_{S,j} = \min_{k \in S, k \neq 1, j} C_{S - \{j\}, k} + c_{kj}.$$

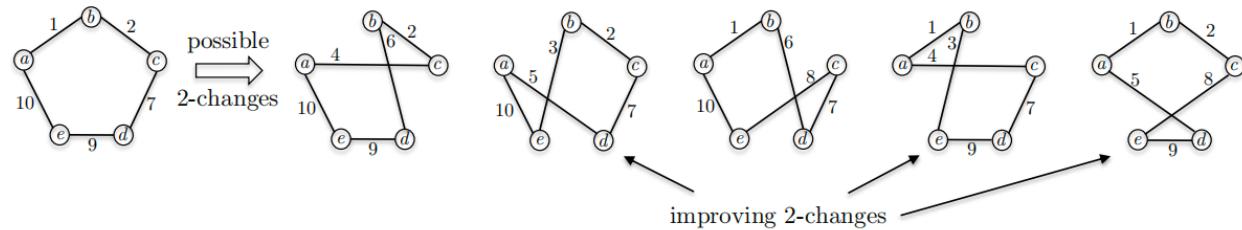
接下来就是要写一个迭代的程序填满子问题的表格，根据递推式的特点，只要按照  $S$  的大小从小到大依次填表，就能保证解每个问题的时候，需要的子问题是已经填满了的。这样就可以得到最终的最优解。这一算法称为 Held-Karp 算法，复杂度分析也很简单：有  $n \cdot 2^n$  个子问题，每个问题的计算复杂度是  $O(n)$ （求  $\min$  需要遍历所有可能），所以总的复杂度是  $O(n^2 \cdot 2^n)$ （当然不要忘了算法最后一步还需要回归到 1 到 1 的环路，这里需要遍历  $n - 1$  种最后一跳的可能，是线性时间的）。

### 12.3.2 局部搜索算法

接下来回归正题，即设计一个局部搜索算法。在上一讲已经介绍了旅行商问题的一个 2-近似算法，可以从这个 2-近似算法开始，然后用局部搜索来改进得到的解。为了设计局部搜索算法，首先需要定义邻居关系，对于一条路线来说，怎样的邻居关系是自然的呢？首先有一个基本的观察，对于两条长度为  $n$  的不同的哈密顿回路，显然它们之间能共享的最大边数为  $n - 2$ 。因此可以自然地定义两条哈密顿回路  $S$  和  $S'$  之间的邻居关系为： $S$  和  $S'$  之间的边有两条不一样，如下图所示：



称这样的变换为 2-变换（2-change）。于是接下来就可以在 2-近似的解的基础上做 2-变换进一步改进解（即检查改变后的受到影响的边的长度是否减小）。当然要注意，上图中二变换不能做成  $v$  和  $u$  相连， $w$  和  $x$  相连，这样变换之后就不是哈密顿回路了。下图给出了一个真实的例子，读者可以参照理解：



## 12.4 Hopfield 神经网络的稳定构型

Hopfield 神经网络是一种全连接的反馈神经网络，于 1982 年由 J.Hopfield 教授提出（他也因此获得 2024 年诺贝尔物理学奖）。关于 Hopfield 神经网络的介绍并非本门课程要求的内容，感兴趣的读者利用互联网自行搜索即可。与优化问题相关的一点是，Hopfield 神经网络有一个能量函数，如果能将能量函数与一个优化问题绑定，那么 Hopfield 神经网络就可以用来解决这个优化问题，例如旅行商问题。

当然这并不是讨论的重点，这里希望讨论的是 Hopfield 神经网络的稳定构型（stable configurations）。为了研究这一问题，需要首先进行一些定义：

1. Hopfield 神经网络可以抽象为一个无向图  $G = (V, E)$ ，其中  $V$  是神经元的集合， $E$  是神经元之间的连接关系，并且每条边  $e$  都有一个权重  $w_e$ ，这可能是正数或负数；
2. Hopfield 神经网络的一个构型（configuration）是指对网络中每个神经元（即图的顶点）的状态的一个赋值，赋值可能为 1 或 -1，记顶点  $u$  的状态为  $s_u$ ；
3. 如果对于边  $e = (u, v)$  有  $w_e > 0$ ，则希望  $u$  和  $v$  具有相反的状态；如果  $w_e < 0$ ，则希望  $u$  和  $v$  具有相同的状态；综合而言，我们希望

$$w_e s_u s_v < 0.$$

4. 将满足上述条件的边  $e$  称为“好的（good）”，否则称为“坏的（bad）”；
5. 称一个点  $u$  是“满意的（satisfied）”，当且仅当  $u$  所在的边中，好边的权重绝对值之和大于等于坏边的权重绝对值之和，即

$$\sum_{v:e=(u,v) \in E} w_e s_u s_v \leq 0,$$

反之，如果  $u$  不满足这一条件，称  $u$  是“不满意的（unsatisfied）”；

6. 最后，称一个构型是“稳定的（stable）”，当且仅当所有的点都是满意的。

这些定义或许太多太杂，但如果读者看到 PPT 第 10 页的例子，应当就会熟悉这些定义。现在的问题是，给定一个 Hopfield 神经网络，是否存在一个稳定构型？如果存在，如何找到这个稳定构型？

当然有的读者可能感兴趣这一问题的研究背景究竟是什么，当然这超出了我们的讨论范围，但简而言之就是 Hopfield 神经网络这样的反馈神经网络在接受输入后会不断自我迭代，最终稳定于一个稳定构型。希望了解细节的读者可以自行查阅相关的人工神经网络的资料。

### 12.4.1 局部搜索算法

如果要设计一个局部搜索算法，有一个非常简单直接的方式。在这一问题中，我们自然地就会定义一个构型的邻居就是将其中一个点的状态取反得到的新构型，然后就可以设计一个局部搜索算法了：从一

从一个随机初始构型开始，然后检查每个点是否满意，如果有不满意的点，就翻转这个点的状态（那么这个点自然就变得满意了），然后继续检查，直到所有的点都满意为止。如果这个算法会停止，那么停止的时候得到的就是一个稳定构型：因为所有点都满意了。PPT 第 11 页给出了这一局部搜索算法的伪代码（称其为 `State_flipping` 算法），以及一个例子。

那么自然的问题就是，这个算法一定会停止吗？这一问题的回答并不是很显然，这里需要引入一个非常常用的工具来解答这一问题，即势能函数。首先定义这一函数，然后来看这么定义的合理性。定义势能函数  $\Phi$ ，它输入一个构型  $S$ ，返回这个构型下所有好边的权重绝对值的和，即

$$\Phi(S) = \sum_{e \text{是好的}} |w_e|.$$

显然的一点是，对于任意的构型， $\Phi(S) \geq 0$ ，并且最大值也就是所有边权重绝对值之和，即  $\Phi(S) \leq \sum_{e \in E} |w_e|$ 。下面来观察翻转一个不满意的点后势能函数的变化。设当前状态为  $S$ ，有一个不满意点  $u$ ，翻转  $u$  的状态后得到  $S'$ ，那么有

$$\Phi(S') - \Phi(S) = \sum_{e=(u,v) \in E, e \text{是坏的}} |w_e| - \sum_{e=(u,v) \in E, e \text{是好的}} |w_e|.$$

这是因为翻转后原先与  $u$  相连的好边都变成了坏边，坏边都变成了好边，其余边没有变化。又因为  $u$  是不满意的，因此与  $u$  相连的坏边比好边权重绝对值之和大，所以上式大于 0，即  $\Phi(S') > \Phi(S)$ 。又因为势能函数只能取整数值，故  $\Phi(S') \geq \Phi(S) + 1$ 。这就意味着每次翻转一个不满意的点，势能函数就会增加至少 1。因为势能函数的取值范围是有限的（0 到所有边权重绝对值之和），所以局部搜索算法一定会停止：

**定理 12.6** *State\_flipping* 算法最多翻转  $\sum_{e \in E} |w_e|$  次后会停止。

由此可以看出定义的势能函数的合理性：这一势能函数随着不满意顶点的翻转，值一定增大，并且势能函数取值有限，因此局部搜索算法一定会停止。此外，不难证明势能函数的局部最大值其实就对应于一个稳定构型：

**定理 12.7** 设  $S$  是一个构型，如果  $\Phi(S)$  是局部最大值，则  $S$  是一个稳定构型。

**证明：**假设  $S$  不是一个稳定构型，即存在一个不满意的点  $u$ ，那么可以翻转  $u$  的状态，得到  $S$  的一个邻居  $S'$ ，并且有  $\Phi(S') > \Phi(S)$ ，这与  $\Phi(S)$  是局部最大值矛盾。 ■

所以通过势能函数将 Hopfield 神经网络的稳定构型的寻找转化为了利用 `State_flipping` 这一局部搜索算法找到势能函数的局部最优解，并且局部最优解在这一问题中其实就是要找到的解（稳定状态）。事实上所谓的稳定构型就是一种纳什均衡，而前面定义的势函数就是算法博弈论中势函数求解纳什均衡的方法，事实上这就解释了为什么会突然冒出一个神奇的势能函数可以证明上述结论，事实上这背后有更一般的方法支撑，感兴趣的读者可以了解相关的概念，这里不展开叙述。

### 12.4.2 PLS 完全性

到目前为止的讨论似乎都忽略了一件事情：那就是局部搜索的时间复杂度，倘若仔细思考，便会发现其中有很多深入的问题。前面证明了 `State_flipping` 算法最多经过  $\sum_{e \in E} |w_e|$  次翻转后会停止——相信到今天你应当能一眼看出这是伪多项式时间复杂度，考虑到每条边的长度输入的二进制编码长度，这一时间复杂度实际上是指数的。当然在边权都比较小，例如是点的个数的多项式级别的时候，这一时间复杂度还能退回多项式级别——这些分析和 0-1 背包的动态规划算法完全一致。

于是有个很自然的问题：寻找 Hopfield 神经网络的稳定构型是否存在多项式级别的局部搜索算法呢？答案是不知道——这与 PLS 这一复杂度类有关。事实上类似于 NP，PLS 也是一个复杂度类，只不过 PLS 是对寻找优化问题局部最优解的困难性进行建模。1988 年，Johnson, Papadimitriou 和 Yannakakis 在他们的文章 *How easy is local search?* 引入了复杂度类 PLS。一个局部搜索问题在 PLS 中，如果：

1. 每个解的大小是输入的多项式级别的；
2. 可以在多项式时间内找到一个可行解（不一定是局部最优解）；
3. 可以在多项式时间内计算每个解的代价；
4. 可以在多项式时间内找到每个解的所有邻居。

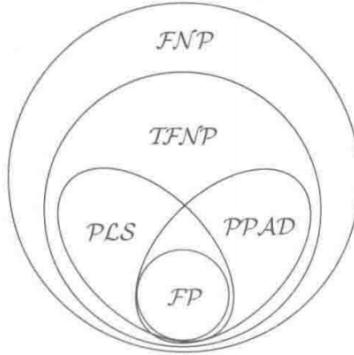
这些要求表明，一个问题在 PLS 中，则可以在多项式时间内判断一个解是不是局部最优解。利用 NP 一讲中学到的知识，可知 PLS 中最困难的问题，也就是所有问题都能归约到它的问题，称之为 PLS-完全问题。在上面的论文中，他们也给出了第一个 PLS-完全问题，即 circuit problem，这里不再深入讲述。关于 PLS 的形式语言定义、归约等内容，读者可以点击这个[维基百科链接](#)。这里只介绍一些比较有趣的事。

为了介绍一些事实，还需要定义一些复杂类（当然我们可能是损失一些严谨性，目的仅仅是介绍一些有趣的事）。首先定义 FP (functional P)，我们知道 P 问题是判定问题，只需要回答是和否，FP 则是在判断是的时候还要给出证据。例如判断两数相加是不是偶数是一个 P 问题，这时只要输出 0/1 即可；但如果是偶数时还要求输出两个数的和，这就是一个 FP 问题。进一步定义 FNP (functional NP)，这是 FP 的 NP 版本，即在判断是的时候还要给出证据。具有至少一个证据的 FNP 问题子集称为 TFNP。

除了 PLS，还有一个与搜索相关的复杂类 PPAD (Polynomial Parity Arguments on Directed graphs)。这一名称非常怪异，是多项式时间的有向图上的奇偶校验的含义，当然也直接表明了这一复杂类的来源。之前的讨论已经看到，PLS 的搜索可以视为一个有向无环图对吸收点的搜索，PPAD 则是在一条有向路径图或一个圆环图上的搜索。具体的定义比较繁杂，但关联就是，纯策略纳什均衡的搜索属于 PLS 完全问题，而一些混合策略纳什均衡的搜索属于 PPAD 完全问题。

这些复杂类之间是什么关联呢？可以用一张图来形象地说明：

具体相关的结论与讨论如下（事实上图中所有的包含关系都应当是暂时未被确认的）：



1. 最关键的直观: PLS 处于 P 和 NP 的难度之间, 这也符合我们对局部搜索问题的直观;
2. Schäffer 和 Yannakakis 在 1991 年证明了 Hopfield 神经网络的稳定构型的局部搜索是 PLS 完全的, 因此目前还没有找到一个多项式时间的算法来解决这一问题;
3. 如果所有 TFNP 问题都是 FNP 完全的, 那么  $\text{co-NP} = \text{NP}$ ;
4. TFNP 是没有完全问题的, 这是因为它是一个语义定义的复杂类, 而别的复杂类都是语法定义的;
5. 如果所有 PLS 问题都是 FNP 完全的, 那么  $\text{co-NP} = \text{NP}$ .

最后一个结论是很新的, 发表在 STOC 2021 上, John Fearnley, Paul W. Goldberg, Alexandros Hollender, Rahul Savani 证明了连续局部搜索 (例如凸优化中的梯度下降等问题) 对应的复杂类 CLS (continuous local search) 是 PPAD 和 PLS 的交集, 非常有趣:

$$\text{CLS} = \text{PPAD} \cap \text{PLS}.$$

总而言之, 上述提到的复杂类之间的关系仍待解决, 特别是算法博弈论这一学科的兴起使得对 PLS 和 PPAD 的研究变得更加重要, 这也是一个非常有趣的研究方向。

## 12.5 最大割问题

最大割问题也是一个经典的 NP 困难问题, 在这一问题中, 我们希望将一个边权全为正的无向图  $G = (V, E)$  的顶点集合  $V$  分成两个集合  $A$  和  $B$  (这时的解记为  $(A, B)$ ), 使得割边的权重和最大, 其中割边的含义就是一条边的两个端点分别在  $A$  和  $B$  中, 即要最大化

$$w(A, B) = \sum_{(u,v) \in E, u \in A, v \in B} w_{uv}.$$

### 12.5.1 局部搜索算法

下面希望给出一个局部搜索解法——这是非常自然的，因为它和 Hopfield 神经网络问题之间有一个很直接的关联。有一个很简单的观察，对于任意的解  $(A, B)$ ，将  $A$  中的点赋状态 -1， $B$  中的点赋状态 1，因为所有边的权重都是正数，所以最大化割边权重和对应于 Hopfield 神经网络问题其实就是最大化好边的总权重和  $\Phi(S)$ ，这就和 Hopfield 神经网络问题一样了。在 Hopfield 神经网络问题中，使用 **State\_flipping** 算法来找到局部最大值，在那时局部最大值对应的构型就是一个稳定构型，现在则对应一个最大割的局部最优解。我们想要知道这个解的质量如何，事实上令人惊喜的是，这个局部搜索算法给出的局部最优解最差也不会低于最优解的一半：

**定理 12.8** 设  $(A, B)$  是如上局部搜索算法得出的一个局部最优解， $(A^*, B^*)$  是最优解，则

$$\frac{w(A, B)}{w(A^*, B^*)} \geq \frac{1}{2}.$$

**证明：**记图  $G$  中所有边的权重之和为  $W = \sum_{(u,v) \in E} w_{uv}$ 。因为  $(A, B)$  是一个局部最优解，即把  $u$  放到  $B$  中会使得割边权重和降低，即

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv},$$

这是因为把  $u$  从  $A$  换到  $B$  后，割边总权重只是增加了不等式左边权重，减小了不等式右边权重。对所有的  $u \in A$  都有这样的不等式，将这些不等式求和有

$$\sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv},$$

很显然，等式左边就是 2 倍的  $A$  中所有边的权重之和，因为每条边都被计算了两次，等式右边就是所有割边的权重之和，所以有

$$2 \sum_{(u,v) \in A} w_{uv} = \sum_{u \in A} \sum_{v \in A} w_{uv} \leq \sum_{u \in A} \sum_{v \in B} w_{uv} = w(A, B),$$

对称地，如果开始选取  $u \in B$ ，也有

$$2 \sum_{(u,v) \in B} w_{uv} \leq w(A, B),$$

又知道  $W = \sum_{(u,v) \in A} w_{uv} + \sum_{(u,v) \in B} w_{uv} + w(A, B)$ ，并且最大割的权重和不可能超过全体边的权重和  $W$ ，所以有

$$w(A^*, B^*) \leq W = \sum_{(u,v) \in A} w_{uv} + \sum_{(u,v) \in B} w_{uv} + w(A, B) \leq 2w(A, B),$$

从而得证。 ■

这一证明看起来有些巧妙，但事实上这里的套路仍然来源于算法博弈论中更一般的框架，实际上就是证明纳什均衡的 POA (price of anarchy, 无秩序代价，即纳什均衡下的代价（局部最优解）与最优化

价（全局最优解）的比值）的标准证明方法：第一步利用局部最优的特点得到任意一个元素具有的表达式（之后就是纳什均衡），第二步将它们求和，第三步则是利用一些巧妙的观察或技术性的操作得到最终的结论。不难看出，第一步和第二步是非常标准的，第三步则是不同问题需要不同巧妙的想法。

上面的结论表明，通过局部搜索找到的局部最优解和最优解之间是可能有可以证明的近似比的。但是注意，这并不一定是 2-近似算法。注意这里的局部搜索算法和 Hopfield 神经网络一样，都是通过 **State\_flipping** 寻找好边权重和  $\Phi(S)$  的局部最优点，而我们已经提及，这一问题是 PLS 完全的，所以目前还没有找到一个多项式时间的算法来解决这一问题。PPT 第 17 页给出了最大割问题真正的多项式时间的近似算法的研究历史，但背后需要的基础知识目前我们并未涉及，所以这里不再展开。这里能给出的是一个类似于 FPTAS 的近似方案，只是近似比是  $2 + \varepsilon$ 。

这里的想法非常简单，事实上在控制迭代次数引起的复杂度时非常常用（这里就是求  $\Phi(S)$  的局部最优值需要的迭代次数可能太多）。我们要求算法在找不到一个能对解有“比较大的提升”的时候就停止，即使当时的解不是局部最优解：当处于解  $w(A, B)$  时，要求下一个解的权重至少要增大  $\frac{\varepsilon}{n}w(A, B)$ ，其中  $n$  是图  $G$  的顶点数。对于这一算法，有如下结论：

**定理 12.9** 设  $(A, B)$  是上述算法给出的解， $(A^*, B^*)$  是最优解，则

$$\frac{w(A, B)}{w(A^*, B^*)} \geq \frac{1}{2 + \varepsilon}.$$

并且这一算法会在  $O(\frac{n}{\varepsilon} \log W)$  次状态翻转后停止，其中  $W$  是所有边的权重之和。

**证明：**近似比的证明事实上非常简单，只需要对前一个近似比为 2 的证明做一些修改即可。因为必须要一个“比较大的改进”才会继续搜索，所以当处于算法的结束状态时，解  $(A, B)$  应当满足，对于任意的  $u \in A$  有

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv} + \frac{\varepsilon}{n} w(A, B),$$

这是因为这时将  $u$  从  $A$  换到  $B$  后，增加的割边权重和不会超过  $\frac{\varepsilon}{n}w(A, B)$ 。然后接下来所有的证明都基于此改进继续推进即可证明。

接下来证明时间复杂度的结论。因为每次状态翻转都会使得割边权重和增加  $1 + \frac{\varepsilon}{n}$  倍，又因为  $(1 + \frac{1}{x})^x \geq 2, \forall x \geq 1$ ，所以有  $(1 + \frac{\varepsilon}{n})^{\frac{n}{\varepsilon}} \geq 2$ ，即目标函数  $\Phi$  变为原先的 2 倍需要的状态翻转次数至多为  $\frac{n}{\varepsilon}$  次，所以算法在  $O(\frac{n}{\varepsilon} \log W)$  次状态翻转后能翻转  $W$  倍（这也是从目标函数最低值 1 翻转到目标函数最大值  $W$  所需的最大倍数），因此状态翻转次数的上界得证。 ■

### 12.5.2 更优的邻居关系

事实上进行到现在我们已经见到很多局部搜索的例子了，但还没有对某个问题的邻居关系的选择有很深入的探讨，都是选择了最直观的邻居关系。事实上，在最开始引入局部搜索的时候就提到，局部搜索

有两个关键点，第一是选取好的邻居关系，第二是每一步找到一个好的邻居继续搜索。在 Metropolis 算法和模拟退火中特别讨论了后者（特别针对于顶点覆盖问题），现在需要仔细讨论前者（特别针对最大割问题）。

很自然地，选取邻居关系时应当注意以下两点：

1. 每个解的邻居数目不应该太少，否则很容易在短短几步之内就陷入一个不佳的局部最优解；
2. 每个解的邻居数目不应该太多，否则搜索空间太大，很难在短时间内找到一个好的邻居——考虑极端情况，如果每个解的邻居数目是所有可能的解，那么就退化成了穷举搜索。

这些原则说起来容易，但真的找到一个更好的邻居关系并非易事。考虑最大割问题，一个非常简单的想法就是，原先的邻居关系是把一个点从一个集合换到另一个集合，现在可以考虑把  $k$  个点从一个集合换到另一个集合，这样就有了一个  $k$ -flip 的邻居关系。一个显然的关系是， $(A, B)$  和  $(A', B')$  是  $k$ -flip 邻居，那么必然也是  $k'$ -flip 邻居，其中  $k' > k$ 。因此如果  $(A, B)$  是  $k'$ -flip 邻居关系下的局部最优解，那么它也是  $k$ -flip 邻居关系下的局部最优解（其中  $k' > k$ ）。这样就可以通过不断增大  $k$  来在每次翻转时找到一个更好的邻居关系，最后解的近似比可能更低。

然而，当  $k$  增大时，每一步翻转找到一个更好的邻居的时间复杂度也会增大——因为搜索空间将是  $\Theta(n^k)$  的，在  $k$  稍微增大一些时这一复杂度就是不可接受的了。

所以需要找到一个权衡。Kernighan 和 Lin (1970) 提出了一种定义邻居关系的方式，它通过如下算法得到  $(A, B)$  的邻居集合：

1. 第一步，选取一个点进行翻转，要求选取的这个点是让翻转后结果的割边权重和最大的那个点（即使有可能最大的权重也会比现有权重低），标记这个被翻转的点，令翻转后的解为  $(A_1, B_1)$ ；
2. 之后第  $k$  ( $k > 1$ ) 步时，有前一步得到的结果  $(A_{k-1}, B_{k-1})$ ，以及有  $k-1$  个已经被标记的点。然后选择一个没有被标记的点进行翻转，要求选取的这个点是让翻转后结果的割边权重和最大的那个点（即使有可能最大的权重也会比现有权重低），标记这个被翻转的点，令翻转后的解为  $(A_k, B_k)$ ；
3.  $n$  步之后所有点都被标记了，也就是说所有的点都恰好被翻转了一次，因此实际上就有  $(A_n, B_n) = (B, A)$ ，这本质上与  $(A, B)$  并无区别。因此上述过程（称为 K-L 启发式算法）得到了  $n-1$  个邻居，即  $(A_1, B_1), (A_2, B_2), \dots, (A_{n-1}, B_{n-1})$ 。因此  $(A, B)$  是一个局部最优解当且仅当对于任意的  $1 \leq i \leq n-1$ ，有  $w(A, B) \geq w(A_i, B_i)$ 。

简单分析即可发现上述寻找邻居的过程是一个  $O(n^2)$  的过程，因此相比于简单的进行  $k$  较大的  $k$ -flip 的邻居关系，K-L 启发式算法的时间复杂度是可以接受的，并且实践表明，局部搜索使用 K-L 启发式算法给出的邻居的效果很好，尽管这一事实暂时还没有理论推导来说明。总而言之，K-L 启发式算法给出的邻居比较好地权衡了算法效率和结果准确性。

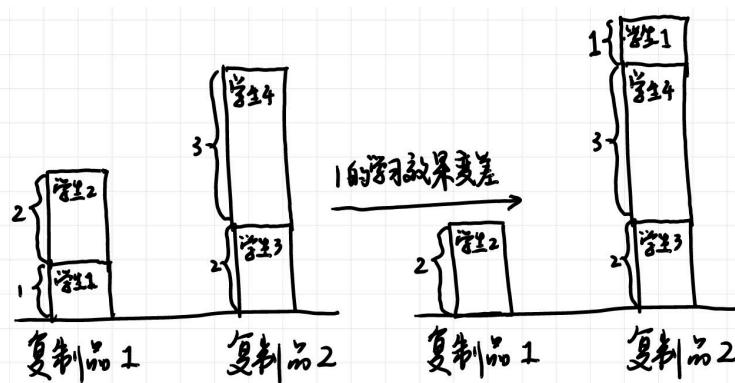
问题：如果助教可以复制

期末考试的脚步逐渐逼近，有  $k$  个正在经受 ADS 折磨的同学（编号为  $1, \dots, k$ ）希望能找 ADS 助教补习功课。每个同学  $i$  都自带一个正整数  $w_i$ ，代表他/她学习 ADS 的困难程度，并且这个数字是公开的：每个人不仅知道自己学习的困难程度，也知道别人学习的困难程度。

正在加紧完成毕业论文的助教见此情形觉得压力太大，于是复制了  $m$  个一模一样的自己来辅导同学。助教把这  $m$  个复制品放在教室后就离开了，留下同学们自由选择  $m$  个复制品中的一个辅导自己。当然复制品也是一条生命，因此一个复制品的培训效果会随着它辅导的同学的困难程度之和增加而减小，因此每个同学都希望自己选择的辅导助教的困难程度之和尽可能小。于是我们可以定义一个“均衡状态”：在这一状态下，每个同学都选择了一个复制品，且每个同学都不愿意换一个复制品来辅导自己，因为这样一定会使得自己选择的新复制品的困难程度之和大于等于原先的困难程度之和，学习效果不会提升。

注意我们有一个重要的假设。即在选择复制品期间每个同学不能互相讨论，也就是每个人做出的决定都是独立完成的，每个同学只是根据当前状态和自身需求来行动，不能和别人商量着一起做什么。

举个简单的例子，假如助教复制了两个一模一样的自己，有四个同学  $1, 2, 3, 4$ ，他们学习 ADS 的困难程度分别为  $1, 2, 2, 3$ ，那么有一种均衡状态是：同学 1 和 2 选择了第一个复制品，同学 3 和 4 选择了第二个复制品，此时第一个复制品的困难程度之和为  $1+2=3$ ，第二个复制品的困难程度之和为  $2+3=5$ 。这一选择为什么是均衡状态呢？因为每个同学都不愿意换一个复制品来辅导自己，比如对于同学 1，现在选择的复制品的困难程度之和为 3，如果他换成选择 2，那么变成了 1、3、4 都选择了第二个复制品，第二个复制品的困难程度之和变为  $1+2+3=6 > 3$ ，所以同学 1 完全没必要从 1 换到 2，因为会使自己的学习效果变差（如下图所示）；又例如对于同学 3，现在所在的第二个复制品的困难程度之和为 5，如果他换成第一个复制品，那么变成了 1、2、3 都选择了第一个复制品，则第一个复制品的困难程度之和变为  $1+2+2=5$ ，所以同学 3 也完全没必要换一个复制品，因为无法改变自己的学习效果。对于同学 2 和 4 也是类似的，所以不再赘述。



1. 自然地，可以用一个局部搜索算法来寻找均衡状态：每个同学首先都随机选择辅导自己的复制品，然后在局部搜索算法的每一步，其中的一个同学会表达自己的不满，因为它自己为了提升学习效果提出要换成更换之后困难程度之和更小的复制品辅导自己。然后不断会有同学表达不满，直到没有同学表达不满时，算法停止。

- (a) 请写出这一局部搜索算法中的邻居关系是什么；  
(b) 证明：如果算法会停止，那么算法停止的时候，得到的状态一定是一个均衡状态；  
(c) 这一算法对于任意情况都会停止吗？如果是，请给出证明，如果不是，请给出一个反例。
2. 现在放宽假设，假如每个同学在做决策的时候是可以互相交流的，也就是他们可以一起商定最后的选择。这些同学内心的良知告诉他们，他们应该让最后的局势尽可能对每个人公平，也就是希望最小化困难程度之和最大的复制品的困难程度（简称困难瓶颈）。还是前面的例子，商量后的结果可以是第 1、4 个同学选择第一个复制品，第 2、3 个同学选择第二个复制品，这样第一个复制品的困难程度之和为  $1 + 3 = 4$ ，第二个复制品的困难程度之和为  $2 + 2 = 4$ ，此时的困难瓶颈为 4（并且显然是最优的了，因为平摊了全部困难程度），而之前的选择导致的困难瓶颈为 5，不如这一分配。
  - (a) 最小化困难瓶颈的选择结果一定是一个均衡状态吗？如果是，请给出证明，如果不是，请给出一个反例；
  - (b) 若限制  $m = 2$ ，最小化困难瓶颈的选择结果一定是一个均衡状态吗？如果是，请给出证明，如果不是，请给出一个反例；
  - (c) 证明：任意一个均衡状态的困难瓶颈一定不大于最小困难瓶颈的两倍。

提示：

1. 1(c) 和 2(b) 可以利用上课所学 Hopfield 神经网络的定义势函数的方法解决；
2. 2(c) 可以参考近似算法的讲义中的最小化工时调度问题，实际上整个问题都只是从那个问题改编过来的，所以如果你仔细阅读了过往的讲义，这题的大部分小问应当是不困难的。

## Lecture 14: 并行算法

编写人: 吴一航 yhwu\_is@zju.edu.cn

穿过了 4 个困难而又精彩的理论专题，我们来到了这门课程的尾声。在期末各大课程大作业等压力骤增的情况下，ADS 的最后两章难度的下降给了诸位温柔的怀抱。言归正传，本讲的主题在这门课程中相对比较独立：我们将进入一个全新的领域，讨论高性能计算在算法方面的基础优化方法，即设计所谓并行算法。本讲将在几个例子中讨论设计并行算法的几种基本思想与技巧。因为本讲在考试中也是比较基础的部分，因此重点将课件中的内容描述清楚，除了开头并行计算基础模型外，不会像前几章一样有过多的拓展。

### 14.1 何谓“并行”

【注】以下有较多内容参考了[这个知乎页面](#)，感兴趣的读者可以参考原链接学习更多。

规模最大且运算能力最强的计算机被称为“超级计算机”。在过去的二十年里，超级计算机的概念无一例外地指向拥有多个 CPU 且可同时处理一个问题的机器——并行计算机。

我们很难精确定义并行的概念，因为它在不同的层面上有着不同的含义。在计算机体系结构中，会见到如下三种并行模式：

1. 指令级并行 (Instruction-Level Parallelism, ILP)：在一个 CPU 内部，多条指令可以同时执行；例如大家现在就已经学过的流水线 CPU，以及将来可能会学到的乱序、多发射以及超长指令字 (VLIW，即把不相关的指令封装到一条超长的指令字中、超标量（例如有多个 ALU，可以同时运行没有相关性的多条指令）等）；
2. 数据级并行 (Data-Level Parallelism, DLP)：即将相同的操作同时应用于一些数据项的编程模型，例如经典的 SIMD (Single Instruction, Multiple Data) 架构，即一条指令同时作用于多个数据，例如用一条指令实现向量加法，两个向量中每对对应的元素相加互不干扰，所以可以同时进行所有的加法；
3. 线程级并行 (Thread-Level Parallelism, TLP)：一种显式并行，程序员要设计并行算法，写多线程程序，这也是本讲将要讨论的内容。线程级并行主要指同时多线程 (SMT，是一种指令级并行的基础上的扩展，可以在一个核上运行多个线程，多个线程共享执行单元，以便提高部件的利用率，提高吞吐量) / 超线程 (HT，一个物理 CPU 核心被伪装成两个以上的逻辑 CPU 核心，看起来就像多个 CPU 在同时执行任务，是通过在一个物理核心中创建多个线程实现的) 以及多核和多处理器。

这里需要澄清三个很相似但又不一致的词语：并发，平行和分布式。其中平行是指任务真的在通过上述三种并行方式在一台机器的处理器上同时执行，分布式是指同一个网络里面的多台机器上同时完成多个任务。并发与上述两种有较大区别，实际上并发并非真的同时执行，而是通过时间片轮转的方式，让多个任务在一个处理器上交替执行，这样看起来就像是同时执行了，诸位在数据库和操作系统中学习的锁等都属于进程、线程调度并发控制的工具。

## 14.2 并行算法基本模型

说了这么多关于并行的基本内容，下面转向今天的主题：并行算法设计。使用并行算法显然是希望利用现代计算机多核的优势，同时执行一些可以不互相干扰的算法部分，加速算法的运行。因此有一个重要的参数称为加速比：

**定义 14.1 加速比 (Speedup)** 是指在并行计算中，使用  $p$  个处理器时相对于使用一个处理器时的性能提升比例，即

$$S(p) = \frac{T_1}{T_p},$$

其中  $T_1$  是使用一个处理器时的运行时间， $T_p$  是使用  $p$  个处理器时的运行时间。

显然最理想的情况下，加速比应当是  $p$ ，即使用  $p$  个处理器时的运行时间是使用一个处理器时的  $1/p$ 。然而实际情况中，由于并行算法的设计和实现都有一定的困难：首先，使用多个处理器意味着额外的通信开销；其次，如果处理器并未分配到完全相同的工作量，则会产生一部分的闲置，就会造成负载不平衡 (load unbalance)，再次降低实际速度；最后，代码运行可能依赖其原有顺序，不能完全并行。

### 14.2.1 演近论

如果我们忽略一些限制，比如处理器的数量，或者它们之间的互连的物理特性，就可以推导出关于并行计算效率极限的理论结果。本节将简要介绍这些结果，并简要讨论它们与现实中高性能计算的联系。

考虑一个很简单的例子，两个  $n$  阶方阵  $A$  与  $B$  相乘得到矩阵  $C$ ，显然总的操作数是  $2n^3$ （所有需要的加法和乘法）。如果有  $n^2$  个处理器，那么有一个自然的并行策略，也就是让每个处理器负责结果矩阵  $C$  的一个元素的计算，而每个元素的计算显然互不干扰，这样并行需要的时间就是计算一个元素的时间，即  $2n$ ，因此这里的加速比就是  $n^2$ ，看起来非常完美。

当然可以用更多的处理器实现更快的算法，实际上矩阵乘法有一个步骤是要求  $n$  个对应元素乘法的和，这一步骤可以并行化，见 PPT 第 5 页，可以用  $n/2$  个处理器实现在  $O(\log n)$  的时间内完成这一步骤。因此如果一共有  $n^3/2$  个处理器，即给每个元素的计算分配  $n/2$  个处理器，那么  $n$  个乘法也可以并行到 2 步之内结束，因为乘法之间互不干扰，加法的时间则下降为  $O(\log n)$ ，因此总的时间为  $O(\log n)$ ，相比于  $O(n)$  的时间有了提升。

对这些理论界线的一个反对意见是，它们隐含地假定了某种形式的共享内存。不难发现，上述矩阵相乘并行算法可能有很多个 CPU 同时访问了原始矩阵的同一个元素，比如同时得到  $C$  的第一行的所有元素需要同时访问  $A$  的第一行的所有元素。事实上，这种算法模型被称为 PRAM 模型 (Parallel Random Access Machine)，是 RAM 模型 (Random Access Machine) 在共享内存系统上的扩展。该模型假设所有处理器共享一个连续的内存空间。此外，模型还允许同一位置上同时进行多个访问。这在实际应用中，特别是在扩大问题规模和处理器数量的情况下是不可能的。对 PRAM 模型的另一个反对意见是，即使在单个处理器上，它也忽略了内存的层次结构（即忽略了 cache 等）。但在接下来的理论讨论中都会忽略这些问题，因为我们的目的是讨论并行算法的基本思想。我们使用 PRAM 模型，这一模型有规定如下三种内存共享方式：

1. EREW (Exclusive Read Exclusive Write): 每个内存位置在任意时刻只能被一个处理器读取或写入；
2. CREW (Concurrent Read Exclusive Write): 每个内存位置在任意时刻可以被多个处理器读取，但只能被一个处理器写入；
3. CRCW (Concurrent Read Concurrent Write): 每个内存位置在任意时刻可以被多个处理器读取或写入，因为写入涉及到同时写入不同值可能造成的冲突，因此写入策略又可以分为如下三种：
  - (a) CRCW-C (Common): 所有处理器写入的值相同时才会写入；
  - (b) CRCW-A (Arbitrary): 所有处理器写入的值可以不同，任意选取其中一个写入即可；
  - (c) CRCW-P (Priority): 所有处理器写入的值可以不同，但是有一个优先级，只有优先级最高的写入才会生效。

如上讨论了内存模型与实际的差别，实际上还有一个具有较大争议的问题是处理器的个数。例如矩阵乘法问题，如果有无穷个处理器，能否设计出更高效的并行算法？当然这一问题的答案并不是今天要关心的，我们要关心的是，无穷个处理器，或者说很多个处理器真的是符合现实的吗？显然不是，处理器之间也是有通信时间的，电子的传输速度是有上界的，或者说目前相信光速是有界的，然后处理器阵列的排列方式也会影响通信时间，实际的布线如何实现也有很大的挑战。实际上，有人证明了，在三维世界和有限的光速下，对于  $n$  处理器上的问题，无论互连方式如何，速度都被限制在  $\sqrt[4]{n}$ 。这是因为有一个结论是，如果每个处理器占用一个单位体积的空间，在单位时间内产生一个结果，并且在单位时间内可以发送一个数据项。那么，在一定的时间内，最多只有半径为  $t$  的球中的处理器，即  $O(t^3)$  的处理器可以对最终结果做出贡献；所有其他处理器都离得太远，这也意味着过多的处理器实际上是毫无意义的。那么，在时间  $T$  内，能够对最终结果做出贡献的操作数

$$\int_0^T t^3 dt = O(T^4),$$

在时间  $T$  内，这意味着，最大的可实现的速度提升是串行时间的四次方根。当然，在我们的模型中，同样因为只是讨论基本的并行算法设计思想，因此不会过多关注这些细节，这里介绍只是希望强调一下实际情况与理想模型的区别。

在讨论了理想模型后，下面进一步讨论一些关于近似比的理论。前面矩阵乘法  $n^2$  个处理器的情况太过于理想，因为所有处理器之间的计算都互相不影响，因此达到了理想加速比。但现实中，部分代码依赖固有顺序执行，因此无法达到理想加速比。假设有 5% 的代码必须串行执行，那么这部分的时间将不会随着处理器的数量增加而减少。因此，对该代码的提速被限制在 20 的系数。这种现象被称为阿姆达尔定律（Amdahl's Law），下面对其进行表述：

**定理 14.2 (阿姆达尔定律, 1967)** 设  $0 \leq f \leq 1$  是一个程序中必须串行执行的部分的比例，那么使用  $p$  个处理器的最大加速比  $S(p)$  满足

$$S(p) = \frac{1}{f + (1 - f)/p}.$$

**证明：**设整个程序串行需要的时间为  $t$ ，那么串行部分的时间为  $ft$ ，并行部分的时间为  $(1 - f)t$ 。使用  $p$  个处理器时，串行部分的时间不变，而并行部分的时间最少为  $(1 - f)t/p$ ，因此总时间最小为  $ft + (1 - f)t/p$ ，因此加速比最大为

$$S(p) = \frac{t}{ft + (1 - f)t/p} = \frac{1}{f + (1 - f)/p}.$$

■

当  $p \rightarrow \infty$  时， $S(p) \rightarrow 1/f$ ，看起来这是一个很糟糕的结论，因为如果一个程序可并行的部分占到 90%，看起来非常多，但加速比最多只能达到 10 倍，是一个并不大的数字，因此一定程度上给研究并行计算的人们带来了挫败感，因为这个定律让人们认为并行可能不是一个很有前途的方向。

然而我们可以怀疑这一定律的合理性：阿姆达尔定律认为只增加处理器数量并不会对并行加速结果有明显的提升，其隐含假设是：越来越多的处理器上执行同一个固定计算。然而在实际中情况并非如此：人们根据可用处理器的数量来调整问题规模的大小，处理器很多的时候可以处理更多的数据，并且有一个现实的假设是，串行部分可以是独立于问题大小的，因此数据量越大，并行部分越大，而阿姆达尔定律则是假设数据集的大小是固定的。首个打破阿姆达尔定律这一假设的是 John L. Gustafson，他在 1988 年于 Communications of the ACM 上发表论文提出了如下定理：

**定理 14.3 (古斯塔夫森定律, 1988)** 设某个程序使用  $p$  个处理器并行执行时，串行部分使用的时间为  $f_1$ （不是比例），并行部分使用的时间为  $f_2$ ，那么使用  $p$  个处理器的最大加速比  $S(p)$  满足

$$S(p) = \frac{f_1 + f_2 \cdot p}{f_1 + f_2}.$$

**证明：**由假设可知，使用  $p$  个处理器并行执行的时间为  $T_p = f_1 + f_2 \cdot p$ ，使用一个处理器串行执行的时间最大为  $T_1 = f_1 + f_2 \cdot p$ ，因此加速比为

$$S(p) = \frac{T_1}{T_p} = \frac{f_1 + f_2 \cdot p}{f_1 + f_2}.$$

■

实际上这里的关键就是没有规定串行和并行时间所占的比例，而是可以根据处理器的数量调整它们各自的占比。显然，这里当  $f_2 \rightarrow \infty$  时， $S(p) \rightarrow p$ ，这就与前面的理想加速比相符合了，因此做并行计算看起来还是有前途的！

阿姆达尔定律和古斯塔夫森定律看起来似乎水火不容，但 1990 年，密西根州立大学的博士生孙贤和（现为伊利诺理工大学教授）与倪明选（现为澳门大学教授）在 Supercomputing 会议上提出了一种 Memory-Bounded 并行加速模型，成功地统一并扩展了 Amdahl 定律与 Gustafson 定律。该模型后被称为“Sun-Ni's Law”，写入并行计算教科书：

**定理 14.4 (孙-倪定律, 1990)** 设某个程序串行部分占比为  $f$ ，并行部分占比为  $1 - f$ ，内存受限函数为  $G(p)$ （后面会解释），那么使用  $p$  个处理器的最大加速比  $S(p)$  满足

$$S(p) = \frac{f + (1 - f) \cdot G(p)}{f + \frac{(1 - f) \cdot G(p)}{p}}.$$

假设  $G(p) = 1$ ，则问题大小是固定的或与资源增加无关，内存限制加速模型简化为 Amdahl 定律；假设  $G(p) = p$ ，则内存限制加速模型简化为 Gustafson 定律，这意味着当内存容量增加  $p$  倍，工作负载也增加  $p$  倍。

然而，前面所说的定律都只是非常粗糙的估计，因为很多实际的程序并不能很干净地分成串行和并行部分，我们需要一个更精细的模型，也就是关键路径（Critical Path）和相应的布伦特定理（Brent's Theorem）。我们将关键路径定义为最长度的依赖关系链（可能是非唯一的），这个长度有时被称为跨度（span），在本课程中统一记为  $D$ ，即深度 depth。由于关键路径上的任务需要一个接一个地执行，关键路径的长度是并行执行时间的一个下限，因此这自然也就是理想情况下有无穷个处理器的时候需要的执行时间，即有

$$D = T_\infty.$$

另外有一个参数——工作量（work），记为  $W$ ，就是并行中所有任务的总时间，实际上就是只用一个处理器的时候的执行时间（忽略不同处理器之间的通信等），即

$$W = T_1.$$

有了这些概念，可以将算法的平均并行度（average parallelism）定义为  $W/D = T_1/T_\infty$ 。

PPT 第 9 页给出了这两个参数在求和问题中更直观的展示，实际上  $D$  就是二叉树的高度， $W$  就是二叉树的所有节点的数量，即  $D = O(\log n)$ ， $W = O(n)$ 。

现在我们讨论了  $T_1$  和  $T_\infty$ ，但没有像前面的定理那样讨论  $T_p$ ，即对于一般的处理器数量，一个任务的处理时间如何。那么接下来的布伦特定理就是要讨论这一问题：

**定理 14.5 (布伦特定理, Brent's Theorem)** 设一个并行计算问题的工作量为  $W$ ，关键路径长度为  $D$ ，那么使用  $p$  个处理器的并行时间具有如下上下界：

$$\frac{W}{p} \leq T_p \leq \frac{W - D}{p} + D.$$

**证明:** 下界是显然的, 因为  $T_p \geq T_\infty$ , 而  $T_\infty$  也是受到理想加速比的限制的, 因此  $T_p \geq W/p$ 。

对于上界, 因为关键路径长度为  $D$ , 因此整个算法的执行可以分为  $D$  个内部并行, 互相之间串行的阶段, 设每个阶段的工作量为  $W_i$ , 则有

$$W = \sum_{i=1}^D W_i,$$

使用  $p$  个处理器时, 每一个阶段所需的时间为  $\lceil W_i/p \rceil$ , 向上取整的原因在于, 每个阶段的工作量可能不能被  $p$  整除, 比如大任务被分成了 7 个子任务, 7 个任务分配给 3 个处理器, 那么每个处理器分配到的任务数应当是 2, 2, 3。

因此需要的总时间就是

$$T_p = \sum_{i=1}^D \left\lceil \frac{W_i}{p} \right\rceil = \sum_{i=1}^D \left( \left\lfloor \frac{W_i - 1}{p} \right\rfloor + 1 \right) \leq \sum_{i=1}^D \left( \frac{W_i - 1}{p} + 1 \right) = \frac{W - D}{p} + D.$$

其中使用了等式

$$\left\lceil \frac{x}{y} \right\rceil = \left\lfloor \frac{x-1}{y} \right\rfloor + 1.$$

■

根据这一结论, 你也可以得到加速比  $T_1/T_p$  的具体公式, 事实上前面的定理都只是在不断地给出加速比的上界, 但这里引入关键路径这一更加贴合实际程序运行的理论模型同时给出了上下界, 因此是一个较强的结论。

### 14.3 前缀和问题

前面的讨论介绍了并行计算的基础理论, 引入了 PRAM 等简化计算模型, 以及基于关键路径的理论, 接下来将在这些理论的基础上讨论一些具体的并行算法设计思路。通常有五种设计并行算法的范式, 包括平衡二叉树、划分范式、双对数范式、加速级联范式以及流水线范式, 其中流水线范式不展开介绍, 感兴趣的读者可以阅读 PPT 最后给出的参考资料, 其中给出了 2-3 树的插入和删除的流水线算法 (笔者并没有仔细阅读, 觉得有些许抽象)。而平衡二叉树在求和问题中已经见到了, 因此接下来相当于给出一个更复杂的例子进行应用。

下面要讨论的是前缀和问题, 即给定一个长度为  $n$  的数组  $A$ , 需要设计并行算法, 求出  $A$  的前  $k$  个元素的和, 其中  $k$  取遍 1 到  $n$ 。PPT 第 12-13 页给出了这一算法, 关键步骤就在于设

$$C(h, i) = \sum_{k=1}^{\alpha} A(k),$$

其中  $\alpha$  是点  $(h, i)$  为根的子树最右下角的叶子的  $i$  值, 那么可以从上往下计算这个  $C$ , 每一层计算都可以并行, 直到算完  $C(0, i)$  实际上就是 1 到  $i$  的前缀和。因此关键就是怎么从上往下计算, 并且保证每一层都能并行。事实上非常简单, 因为很容易观察到根结点  $C$  值就是整个数组的和, 因此这只需要把自底向上求和的结果 (存在数组  $B$  中) 保留即可。然后再往下走的过程中, 有如下三个观察:

1. 如果  $i = 1$ , 那么  $C(h, i) = B(h, i)$ , 这是因为  $i = 1$  的时候, 根据定义,  $C(h, 1)$  是从第 1 个元素加到以这一点为根的子树右下角的叶子, 而  $B(1, h)$  事实上也就是从第一个元素加到以这一点为根的子树右下角的叶子 (看图可以看得很清楚);
2. 如果  $i$  是偶数, 这表明这一点是某个点的右儿子, 因此它和它的父亲最右下角的叶子是同一个, 因此有  $C(h, i) = C(h + 1, i/2)$ ;
3. 如果  $i$  是奇数且不是 1, 这表明这一点是某个点的左儿子, 首先它自己的值  $B(h, i)$  不是从 1 开始加的, 所以要选一个左边的点, 把从 1 开始加到这个点对应的之前的部分补上, 即  $C(h, i) = C(h + 1, (i - 1)/2) + B(h, i)$ 。需要注意的是, 如果不是并行算法,  $C(h + 1, (i - 1)/2)$  可以替换为  $C(h, i - 1)$ , 但因为并行算法要求每一行是一起算出来的, 所以要用上一行的才合理。

算法的伪代码见 PPT 第 13 页, 显然这一并行算法的深度是  $O(\log n)$ , 总工作量是  $O(n)$ , 因为无非就是在完全二叉树上先从下往上, 然后再从上往下分层遍历了两轮。

## 14.4 归并问题

接下来将介绍第二种并行算法的设计思路, 因为二叉树对于更复杂的问题而言有较大的局限性。讨论的第一个例子是归并问题, PPT 第 14 页给出了问题的描述, 事实上与归并排序中“归并”步骤是一致的, 除了增加了几条比较平凡的假设。下面将讨论如何设计一个并行算法来解决这个问题。

### 14.4.1 简单的想法

既然可以设计并行算法, 那么如果能同时将所有  $A$  和  $B$  中的元素在最后的数组  $C$  中的位置找到, 然后同时将它们放到正确的位置上, 那么就可以以很快的速度解决这一问题。

这里的关键步骤就是同时找到  $A$  和  $B$  中的元素在  $C$  中的位置, 这是一个并不困难的问题, 例如对  $A$  中的元素  $A[i]$ , 它在  $A$  中是第  $i + 1$  个元素 (下标从 0 开始), 即在它前面有  $i$  个元素。如果能找到它在  $B$  中处于

$$B[j] < A[i] < B[j + 1]$$

的位置, 那么  $B$  中有  $j + 1$  个元素在合并后的  $C$  中应当在  $A[i]$  前面, 因此最后  $A[i]$  在  $C$  中的位置就是  $C[i + j + 1]$ , 这样它前面就有  $i + (j + 1)$  个元素了。对  $B$  中的元素  $B[k]$  也有类似的分析。

由此将归并问题转化为了找到某个数组中的一个元素在另一个数组中的位置的问题。显然有两种策略:

1. 逐个元素二分查找: 使用二分查找找到一个元素在另一个数组中的位置只需要  $O(\log n)$  的时间, 即深度  $D = O(\log n)$ , 总工作量为  $O(n \log n)$ ; 然后需要将所有元素放到正确的位置上, 这一深度为  $O(1)$ , 总工作量为  $O(n)$  (如果数组长度不同则是  $O(n + m)$ ), 两步合起来深度为  $O(\log n)$ , 总工作量为  $O(n \log n)$ ;

2. 整体线性查找: PPT 第 16 页给出了伪代码, 实际上和归并排序的操作并无本质差别, 就是两个数组的元素从头依次向后比较, 因此完全没有并行, 深度和总工作量都是  $O(n)$  (或者数组长度不同就是  $O(m + n)$ )。

但这两种方法都不令人满意, 因为第一种方法的总工作量太大, 而第二种方法的深度太大。我们希望能够找到一个更好的方法, 这就需要引入一种更加高级的并行算法设计思路, 即划分范式。

#### 14.4.2 划分范式 (Partitioning Paradigm)

划分范式的想法非常简单, 且实际运用起来非常有效, 分为如下两个步骤:

1. 划分 (partitioning): 把原问题划分为很多 (设为  $p$  个) 很小的子问题, 每个子问题大小大致为  $n/p$ ;
2. 真实工作 (actual work): 同时对所有子问题进行处理, 得到最终结果。

有心的同学应当观察到了, 前面的二分查找方法相当于将原问题划分为了  $n$  个子问题, 即  $p = n$ , 而线性的方法则完全没有划分, 即  $p = 1$ 。因此这两种方法分别对应于划分范式的两个极端, 所以有理由相信合理的  $p$  可以得到一个更好的结果, 能让两个极端取长补短。

#### 14.4.3 运用划分范式

目标非常明确, 就是取长补短。因此算法应当按照如下方式套用划分范式:

1. 划分: 将每个数组划分为  $p$  份 (此时  $p$  未知, 分析后可以选取到最优的  $p$ ),  $p$  是一个很大的值, 每个子问题很小, 如 PPT 第 17 页图所示。首先对每个子问题的第一个元素求出它们在另一个数组的位置, 这一步应当使用二分查找, 否则因为子问题很多, 使用线性查找会导致总工作量达到  $O(p \cdot n)$ , 这几乎是平方级别的工作量, 显然是不可接受的; 因此使用二分查找, 这样深度为  $O(\log n)$ , 总工作量为  $O(p \log n)$ ;
2. 真实工作: 如 PPT 第 17 页图所示, 现在剩下的工作就是相邻两个箭头之间的部分需要在这一很小的区域内确认相对位置, 因为这些位置确定后直接可以根据上一步划分中得到的结果确定在整个最终的数组中的位置。

注意, 很显然的一点是, 这些箭头不会交叉, 这是由箭头代表的大小关系决定的。另一方面, 相邻两个箭头之间的距离一定不超过  $n/p$ , 因为一旦超过  $n/p$ , 那其中一定会有一个箭头 (箭头出现的频率是每  $n/p$  个元素一次), 并且一个数组上有  $p$  个出发的位置和  $p$  个到达的位置, 所以在任意一个数组上, 出发点和到达点一共最多有  $2p$  个, 然后两个数组的这些划分区域按图中形式一一对应进行计算, 每个区域实际上就是一个子问题, 即现在还剩下  $2p$  个大小为  $O(n/p)$  的子问

题。这时并行对每个子问题直接使用线性查找即可，因为每个子问题都非常小（大小为  $O(n/p)$ ），此时深度为  $O(n/p)$  也很小，总工作量为  $2p \cdot O(n/p) = O(n)$  是符合要求的。而如果使用二分查找，总工作量为

$$2p \cdot \frac{n}{p} \cdot O(\log \frac{n}{p}) = 2n \log \frac{n}{p},$$

这是因为有  $2p$  个子问题，每个子问题都至多有  $n/p$  个元素要用至多  $\log(n/p)$  的时间确定位置，这并无法保证是  $O(n)$  的，因此不使用二分查找。

现在的问题就在于，还有两个时间复杂度未定：第一步划分的总工作量为  $O(p \log n)$ ，第二步真实工作的深度为  $O(n/p)$ 。我们希望找到一个最优的  $p$ ，使得第一步总工作量是  $O(n)$  的，第二步深度是  $O(\log n)$  的。这样就可以得到一个深度为  $O(\log n)$ ，总工作量为  $O(n)$  的并行算法，成功做到取长补短。事实上很容易看出， $p = \frac{n}{\log n}$  就是满足条件的，因此就选取这个  $p$ ，得到了一个深度为  $O(\log n)$ ，总工作量为  $O(n)$  的并行算法。

总而言之，划分范式就是通过划分（在大数据量下使用二分查找）和真实工作（在小工作量下使用线性查找）统一了两种极端情况，然后希望找到一个折中的合适的  $p$ ，使得两个极端取长补短，得到一个更好的结果。

## 14.5 寻找最大元素

下面希望设计并行算法找到一个数组中的最大元素。我们将会讨论第三种并行算法的设计范式，即双对数范式，这是二叉树的一种扩展；然后讨论加速级联范式，它可以综合不同的并行算法得到更好的并行算法。

### 14.5.1 简单的想法

在讨论复杂的范式前先来讨论一些简单的想法。第一种最简单的想法就是类似于前面求和问题的做法，构造一棵二叉树，初始元素两两分组比较，然后逐层上递，最终得到最大值。这样的算法深度为  $O(\log n)$ ，总工作量为  $O(n)$ 。

然而我们可以更激进：为什么不一次性把所有元素之间都比较一次呢？这样深度只需要  $O(1)$ 。如果不考虑总工作量会很大的危害，这种方式是可行的：并行比较每一对元素  $A[i]$  和  $A[j]$ ，只要  $A[i]$  在比较中是较小的一方，就往新的数组  $B[i]$  处写 1 ( $B$  的所有位置初始化为 0)，最后那个最大的元素肯定从来没被写过 1，因此还保持着 0，所以并行判断  $B$  中哪个元素是 0，那它就是最大值。

这里有一个需要注意的细节是，在并行比较每一对元素时，可能会有多个线程同时往数组  $B$  中进行写入，实际上只需要使用前面提到的 CRCW 策略允许同时写入，然后按 common rule 写入即可，因为这里所有线程往任意一个  $B[i]$  只会写入 1 这个数字，所以写入的内容都是一样的，故用 common 就能实现写入。

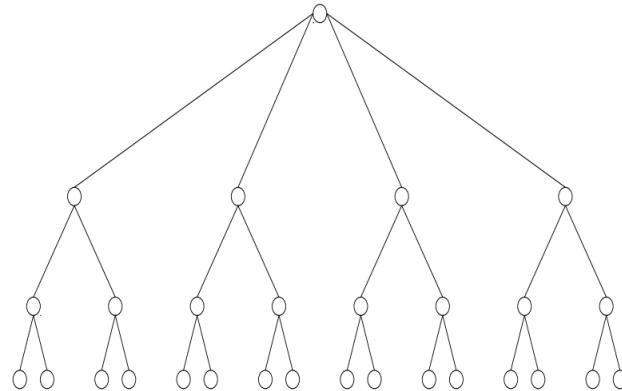
很显然，这样的算法深度非常完美，是  $O(1)$  的，但总工作量重点需要比较每一对元素的值，因此是  $O(n^2)$  的。上面的两种方法在总工作量和深度上如果能取长补短，必定是非常完美的，接下来就来逐步尝试优化它们。

### 14.5.2 双对数范式 (Doubly-logarithmic Paradigm)

双对数范式是一种可以二叉树的扩展。在完全二叉树中，设叶子的数量为  $n$ ，那么树高是  $\log n$  级别的，这里双对数则是希望构造一棵树，使得树高是  $\log \log n$  级别的。为了构造这样一颗树，首先设树中每个节点的 level 为从根到该节点的距离（需要经过的边数），根的 level 为 0。接下来构造这棵树如下：

1. 设某个节点的 level 为  $s$ ，当  $s \leq \log \log n - 1$  时，则它有  $2^{2^{h-s-1}}$  个孩子；
2. 当  $s = \log \log n$  时，它有 2 个孩子作为树的叶子。

不难验证这样一颗树的叶子数量的确是  $n$ ，高度也是  $\log \log n$  级别的。下图给出了一个 16 个叶子的双对数树的例子：



事实上还可以有一个观察，即一个不在倒数两行的结点的 level 为  $s$  时，根据定义它有  $2^{2^{h-s-1}}$  个孩子，然后以它为根的子树的叶子数量可以算出来是  $2^{2^{h-s}}$ ，而我们知道

$$2^{2^{h-s-1}} = \sqrt{2^{2^{h-s}}},$$

也就是说在每个节点相当于把问题分成了  $\sqrt{m}$  个子问题，其中  $m$  为该节点对应的子树的叶子数量。而我们知道，以双对数树某一层单个节点为根的子树的叶子数，其实是上一层某个节点为根的子树的叶子数开根号（因为层数为  $s$  的某个节点，以它为根的子树的叶子数量是  $2^{2^{h-s}}$ ， $s$  每增加 1 实际上就是开一次根号），因此层数越往下，就相当于子问题大小逐步开根号。假设原问题输入大小为  $n$ ，那么可以通过加一些 dummy 节点使得可以构造出以这  $n$  个节点为叶子的双对数树，这样就可以使用双对数范式来设计一个并行算法，即在每一层将问题分为  $\sqrt{m}$  个子问题， $m$  为这一层的单个节点为根的子树的叶子数量，这样每个子问题（其实就是树上某个节点）对应的叶子数也就是  $\sqrt{m}$  了，这就和双对数树结合起来了。我们熟知的二叉树则是每层分成两个子问题，是更简单的策略。

将这一思路套用在当前的问题上：首先将数组划分为  $\sqrt{n}$  份，每一份的大小为  $\sqrt{n}$ ，然后并行递归找到每一份中的最大值，最后归并阶段再并行找到这  $\sqrt{n}$  个最大值中的最大值即可。设整个算法深度为  $D(n)$ ，工作量为  $W(n)$ ，它们都是关于数组长度的函数。每一份中的最大值采用递归的策略，因此长度为  $\sqrt{n}$  的数组的最大值的寻找需要  $D(\sqrt{n})$  的深度和  $W(\sqrt{n})$  的总工作量。以上是递归阶段，在归并阶段，返回的  $\sqrt{n}$  个最大值中的最大值采用前面的两两比较的方法即可，深度为  $O(1)$ ，总工作量为  $O((\sqrt{n})^2) = O(n)$ 。于是按照分治法的方式可以写出递推式

$$\begin{aligned} D(n) &= D(\sqrt{n}) + O(1), \\ W(n) &= \sqrt{n}W(\sqrt{n}) + O(n). \end{aligned}$$

第一个式子这样的类型在分治算法中应当已经见过，换元法即可破解。设  $n = 2^m$ ，则有

$$D(2^m) = D(2^{m/2}) + O(1),$$

设  $D(2^m) = S(m)$ ，则有

$$S(m) = S(m/2) + O(1),$$

肉眼即可解得  $S(m) = O(\log m)$ ，因此  $D(n) = O(\log \log n)$ 。

对于  $W$  的递推式，直接暴力展开找规律即可。同样设  $n = 2^m$ ， $S(m) = W(2^m)$ ，则有

$$S(m) = 2^{m/2}S(m/2) + O(2^m),$$

进一步展开  $S(m/2)$ ，有

$$S(m/2) = 2^{m/4}S(m/4) + O(2^{m/2}),$$

代入上式，有

$$S(m) = 2^{3m/4}S(m/4) + O(2 \cdot 2^m),$$

继续展开，有

$$S(m) = 2^{7m/8}S(m/8) + O(3 \cdot 2^m),$$

以此类推，直到  $S(1)$  有

$$S(m) = 2^{m-1}S(1) + O(\log m \cdot 2^m),$$

代入  $S(1) = O(1)$ ，有

$$S(m) = O(\log m \cdot 2^m),$$

即  $W(n) = O(n \log \log n)$ 。因此得到了一个深度为  $O(\log \log n)$ ，总工作量为  $O(n \log \log n)$  的并行算法。

当然可以用更直观的方式来理解上面两个递推式的结果。事实上  $D$  的递推式就对应于求叶子数为  $n$  的双对数树的高度，因为子问题大小每开一次根号，大小就加一，这和双对数树每往下一层，单个节点为根的子树的叶子数开根号完全对应。而  $W$  的递推式表明，在最高层需要多做  $O(n)$  的工作，第二层应当多做  $O(\sqrt{n} \cdot \sqrt{n}) = O(n)$  的工作 ( $\sqrt{n}$  个子问题，每个子问题在解决时，综合出这个子问题的解都需要  $\sqrt{n}$  的时间)，以此类推，每一层都需要做  $O(n)$  的工作，因此总工作量是  $O(n \log \log n)$ ，实际上这就是在利用递归树方法求解通项。

注：其实上面不一定要分成  $\sqrt{n}$  份，分成  $n^{1/3}$  份也可以用相同的方式证明最后的复杂度满足前面的结论。

### 14.5.3 加速级联范式 (Accelerating Cascades Paradigm)

我们仍然希望改进上面的算法，那么加速级联范式通常是一个值得尝试的手段——即可以考虑将现有的并行算法，例如上面讨论的双对数范式的算法与其它策略结合，得到更好的算法。方法如下（至于怎么凑的，那也是一些智慧与尝试的结晶了）：

1. 将数组分为  $n/\log \log n$  份，即每一份的大小为  $\log \log n$ ，如 PPT 第 20 页；实际上每一份的大小都很小了，所以可以直接利用线性查找的方式找到每一份的最大值，则每一份的深度和工作量都是  $O(\log \log n)$  的；
2. 然后对上面求出的  $n/\log \log n$  个最大值使用双对数范式的算法。

于是总的深度为

$$D(n) = O(\log \log n) + O(\log \log(n/\log \log n)) = O(\log \log n),$$

总的工作量为

$$W(n) = O(n/\log \log n \cdot \log \log n) + O(n/\log \log n \cdot \log \log(n/\log \log n)) = O(n).$$

当然也可以尝试其它级联的方式，比如第一步用二叉树的方法等，但采用上面的步骤已经能实现一定的改进。同学们也可以探究不同的划分对于这一算法的影响，这里就不再深入讨论了。

### 14.5.4 随机算法

看起来还有改进的空间，毕竟当初暴力的两两比较方法能实现  $O(1)$  的深度。接下来将介绍一种随机算法，它可以保证以非常高的概率在  $O(1)$  的深度和  $O(n)$  的工作量内找到最大值。

首先给出算法的描述，然后进行分析：

1. 第一步：从长度为  $n$  的数组  $A$  中，依照均匀分布取出  $n^{\frac{7}{8}}$  个元素，得到新的数组记为  $B$ ；这一步需要  $n^{\frac{7}{8}}$  个处理器各自负责抽一个然后放到内存中某个位置，深度为  $O(1)$ ，总工作量为  $O(n^{\frac{7}{8}})$ ；
2. 第二步：求出  $B$  中的最大值，使用确定性的算法实现，通过如下三个子步骤实现：
  - (a) 把  $B$  分成  $n^{\frac{3}{4}}$  个长度为  $n^{\frac{1}{8}}$  的子数组，然后使用两两比较的暴力算法并行找到每个子数组的最大值，这一步深度为  $O(1)$ ，总工作量为  $O(n^{\frac{3}{4}} \cdot (n^{\frac{1}{8}})^2) = O(n)$ ，因为有  $n^{\frac{3}{4}}$  个子数组，每个子数组用两两比较的办法的复杂度是数组长度平方级别的；然后将这  $n^{\frac{3}{4}}$  个最大值放在新数组  $C$  中；

- (b) 把  $C$  分成  $n^{\frac{1}{2}}$  个长度为  $n^{\frac{1}{4}}$  的子数组，然后使用两两比较的暴力算法并行找到每个子数组的最大值，这一步深度为  $O(1)$ ，总工作量为  $O(n^{\frac{1}{2}} \cdot (n^{\frac{1}{4}})^2) = O(n)$ ，然后将这  $n^{\frac{1}{2}}$  个最大值放在新数组  $D$  中；
- (c) 数组  $D$  直接使用两两比较的方法找到最大值，这一步深度为  $O(1)$ ，总工作量为  $O((n^{\frac{1}{2}})^2) = O(n)$ 。

综合上述方法，整个第二步相当于一个三轮的淘汰赛，整体而言第二步的深度为  $O(1)$ ，总工作量为  $O(n)$ ，并且能求出  $B$  中的最大值，也就是抽取的  $n^{\frac{7}{8}}$  个元素中的最大值；

**【注】** 实际上不难验证最开始抽  $n^{\frac{15}{16}}$  或者其它类似值的也是可以的，当然不能取太高破坏复杂度。

3. 第三步：目前只得到了  $n^{\frac{7}{8}}$  个元素中的最大值（记为  $M$ ），如何进一步提高概率呢？答案是再来一轮，但是这再来一轮是有讲究的。这一步首先均匀分布取出  $n^{\frac{7}{8}}$  个元素，得到数组  $B$ ；然后用  $n$  个处理器，每个处理器放  $A$  的一个元素，与前一步得到的  $M$  进行比较，如果小于  $M$  则什么也不用做，大于  $M$  则往数组  $B$  的一个随机位置写入这一更大的值（为什么要随机写入呢？因为一共  $n$  个处理器，但只有  $n^{\frac{7}{8}}$  个位置可供选择，不能一一对应位置供每个处理器使用，但又要在  $O(1)$  时间内完成位置分配，只能是随机了）。写完后，再次求出更新后的  $B$  中的最大值，这一步的深度和工作量和第二步一样，因此是  $O(1)$  的深度和  $O(n)$  的工作量。

那么这样成功的概率如何呢？如果第二步得到的  $M$  本来就在原数组中排名靠前，例如位列前  $n^{\frac{1}{4}}$ ，那么第三步随机放入更大的元素时，出现冲突的概率就会降低，如果完全没有冲突，那么所有比  $M$  大的元素都会被放入  $B$  中，这样  $B$  中的最大值就是原数组的最大值。但如果出现冲突且导致真正的最大元素因为冲突没有写入  $B$ ，那么我们的算法就失败了。

总而言之，关于算法的正确性有如下定理：

**定理 14.6** 以上算法以非常高的概率在  $O(1)$  的深度和  $O(n)$  的工作量内找到数组  $A$  中的最大值：存在常数  $c$ ，使得算法只有  $\frac{1}{n^c}$  的概率无法在这一时间复杂度内找到最大值。

在这一复杂度内找到最大值的含义可以简单理解为，上面的第三步只执行一次——事实上为了进一步提高正确概率，第三步可以被重复实施很多次，但很多次后将会破坏算法的时间复杂度，上面的定理就是表明，在找到最大值之前破坏这一时间复杂度的概率是很小的。

最后以这一定理的证明来结束本讲的内容（但这一证明并不要求掌握）：

**证明：**事实上，上述算法第三步一轮之内能找到最大值的概率大于等于如下两部分的乘积：

1. 第二步返回的结果排名在整个长度为  $n$  的数组的前  $n^{\frac{1}{4}}$ ；
2. 在上一条的条件下，第三步随机放入的元素都没有冲突。

大于等于的原因是，即使不满足上述条件也可能找到最大值，比如冲突的元素不是最大值，但这么粗略的估计就能够证明最后的结论。

首先来看第一部分的概率。考虑这一问题的补，即抽到的  $n^{\frac{7}{8}}$  个元素中最大的那个元素排名在前  $n^{\frac{1}{4}}$  之外的概率。这事实上等价于所有这  $n^{\frac{7}{8}}$  个元素的排名都在前  $n^{\frac{1}{4}}$  之外。因为是均匀分布抽取的，故每一个元素的排名在前  $n^{\frac{1}{4}}$  之外的概率是

$$\frac{n - n^{\frac{1}{4}}}{n} = 1 - n^{-\frac{3}{4}},$$

并且不同的元素之间是独立的，因此所有元素的排名都在前  $n^{\frac{1}{4}}$  之外的概率是

$$(1 - n^{-\frac{3}{4}})^{n^{\frac{7}{8}}} = ((1 - n^{-\frac{3}{4}})^{n^{\frac{3}{4}}})^{n^{\frac{1}{8}}},$$

回忆  $(1 - \frac{1}{n})^n$  是单调递增趋于  $\frac{1}{e}$  的，因此一定存在一个常数  $c_1 < 1$  使得上式小于等于  $c_1^{n^{\frac{1}{8}}}$ 。由于这里求的是原问题的补问题的概率，故原问题的概率大于等于  $1 - c_1^{n^{\frac{1}{8}}}$ 。

接下来求解第二部分概率，即仅有小于等于  $n^{\frac{1}{4}}$  个元素会写入新数组的情况下，写入没有冲突的概率，这个概率的一个下界是不难得到的。虽然所有元素是并行写入的，但可以串行思考。第一个元素写入时，没有冲突的概率显然是 1；第二个元素写入时，不与第一个元素冲突的概率是

$$\frac{n^{\frac{7}{8}} - 1}{n^{\frac{7}{8}}} = 1 - n^{-\frac{7}{8}},$$

当写入第  $i + 1$  个元素时，它不发生冲突的概率最低的情况就是前  $i$  个元素都没有冲突的情况，因为此时前  $i$  个元素都写入了不同位置，因此冲突的概率最高，故不冲突的概率最低。因此第  $i + 1$  个元素不发生冲突的概率大于等于

$$1 - i \cdot n^{-\frac{7}{8}},$$

因此所有元素都不冲突的概率大于等于

$$(1 - n^{-\frac{7}{8}})(1 - 2n^{-\frac{7}{8}}) \cdots (1 - n^{\frac{1}{4}} \cdot n^{-\frac{7}{8}}) \geq (1 - n^{\frac{1}{4}} \cdot n^{-\frac{7}{8}})(1 - n^{\frac{1}{4}} \cdot n^{-\frac{7}{8}}) \cdots (1 - n^{\frac{1}{4}} \cdot n^{-\frac{7}{8}}) = (1 - n^{-\frac{5}{8}})^{n^{\frac{1}{4}}},$$

又有

$$(1 - n^{-\frac{5}{8}})^{n^{\frac{1}{4}}} = ((1 - n^{-\frac{5}{8}})^{n^{\frac{5}{8}}})^{n^{-\frac{3}{8}}},$$

因为  $(1 - \frac{1}{n})^n$  单调递增趋于  $\frac{1}{e} \geq \frac{1}{3}$ ，故  $n$  较大时，上式大于等于  $(\frac{1}{3})^{n^{-\frac{3}{8}}}$ 。

为了得到最后的结果，需要考察 1 和  $(\frac{1}{3})^{n^{-\frac{3}{8}}}$  之间的距离，设  $1 - f(n) = (\frac{1}{3})^{n^{-\frac{3}{8}}}$ ，即  $(1 - f(n))^{\frac{8}{3}} = \frac{1}{3}$ 。下面希望据此推出  $f(n) = O(\frac{1}{n^{c_2}})$ ，其中  $c_2 > 0$  是一个常数。事实上，如果取  $f(n) = n^{-\frac{3}{16}}$ ，那么  $(1 - n^{-\frac{3}{16}})^{\frac{8}{3}}$  会趋于 0，因为这实际上就是  $(1 - \frac{1}{n})^{n^2}$ ，而这一值趋向于  $\frac{1}{e^n}$ ，故趋近于 0。因此不可能等于  $1/3$ ，想要始终恒等于  $1/3$ ， $f(n)$  的下降速度必须比  $n^{-\frac{3}{16}}$  更快，在  $n$  较大的时候值也就更小，因此  $f(n)$  一定是  $O(\frac{1}{n^{c_2}})$  的，其中  $c_2 > 0$  是一个常数。

综合上面的讨论，第三步一轮之内不能找到最大值的概率就会小于等于 1 减去前面算出的两部分的乘积，即

$$P \leq 1 - (1 - c_1^{n^{\frac{1}{8}}}) \cdot (1 - f(n)) = f(n) + c_1^{n^{\frac{1}{8}}} \cdot (\frac{1}{3})^{n^{-\frac{3}{8}}},$$

我们知道,  $f(n)$  是  $O(\frac{1}{n^{c_2}})$  的, 而  $c_1^{n^{\frac{1}{8}}}$  在  $n$  较大时会趋于 0,  $(\frac{1}{3})^{n^{-\frac{3}{8}}}$  会趋于 1, 因此  $P$  最大也就是  $f(n) = O(\frac{1}{n^{c_2}})$  级别的, 因此命题得证。 ■

## Lecture 15: 外部排序

编写人: 吴一航 yhwu\_is@zju.edu.cn

很高兴, 我们走到了这门课, 也是浙江大学数据结构与算法课程系列的最后一个章节。作为两个学期数据结构与算法的收尾, 我们将第一次讨论考虑外存的算法 (当然之前的 B+ 树和倒排索引讨论了外存中的数据结构)。

### 15.1 外部排序的引入

大部分内部排序算法都用到内存可直接寻址的事实。希尔排序用一个时间单位比较元素  $A[i]$  和  $A[i - h_k]$ 。堆排序用一个时间单位比较元素  $A[i]$  和  $A[i * 2 + 1]$ 。使用三数中值分割法的快速排序以常数个时间单位比较  $A[Left]$ 、 $A[Center]$  和  $A[Right]$ 。如果输入数据在磁带上, 那么所有这些操作就失去了它们的效率, 因为磁带上的元素只能被顺序访问 (磁头只能顺序移动, 如果不顺序访问那就很耗时间)。即使数据在一张磁盘上, 由于转动磁盘和移动磁头所需的延迟, 仍然存在实际上的效率损失。

基于这一原因, 在外部排序时适合使用的是归并排序, 因为归并排序的时间复杂度来源于归并, 而归并只需要顺序扫描两个有序数组, 然后写入的时候也是顺序写入, 因此适用于外部排序。

### 15.2 简单算法

#### 15.2.1 算法描述

PPT 第 3 页给出了一个简单的归并想法, 更一般化的描述如下: 设有四盘磁带,  $T_{a_1}, T_{a_2}, T_{b_1}, T_{b_2}$ , 它们是两盘输入磁带和两盘输出磁带, 并且功能是交替的 (后面会看到)。

设数据最初在  $T_{a_1}$  上, 并设初始数组有  $N$  个元素, 并且内存可以一次容纳 (和排序)  $M$  个记录。一种自然的做法是第一步从输入磁带一次读入  $M$  个记录, 在内部将这些记录排序, 然后再把这些排过序的记录交替地写到  $T_{b_1}$  或  $T_{b_2}$  上。我们将每组排序好的记录叫做一个顺串 (run)。

现在  $T_{b_1}$  和  $T_{b_2}$  各包含一组顺串, 下一步,  $b$  就变成了输入磁带,  $a$  因为上面的内容没用了所以可以变成输出磁带。将  $T_{b_1}$  和  $T_{b_2}$  上每个磁带的第一个顺串取出并将二者合并, 把结果写到  $T_{a_1}$  上, 该结果是一个二倍长的顺串。然后, 再从每盘磁带取出下一个顺串, 合并, 并将结果写到  $T_{a_2}$  上。继续这个过程, 交替使用  $T_{a_1}$  和  $T_{a_2}$ , 直到  $T_{b_1}$  或  $T_{b_2}$  为空。此时, 或者它们均为空, 或者剩下一个顺串。对于后者, 把剩下的顺串拷贝到适当的磁盘末尾即可。

进一步地，重复相同的步骤，这一次用两盘  $a$  磁带作为输入，两盘  $b$  磁带作为输出，结果得到一些  $4M$  的顺串。继续这个过程直到得到长为  $N$  的一个顺串。显然，因为一趟工作顺串长度加倍，该算法将需要  $\lceil \log_2(N/M) \rceil$  趟工作（这里的趟就是 PPT 上的 pass，表示一轮从输入磁盘到输出磁盘的工作），外加一趟构造初始的顺串。例如 PPT 上的例子，第一趟构造出长度为 3 的顺串，接下来是三趟工作：顺串长度依次由 3 变 6，6 变 12，12 变 13，因此共 4 趟操作。

**【讨论】**若有 1000 万个记录，每个记录 128 个字节，并有 4 兆字节的内存，需要多少趟操作？

事实上第一趟将建立 320 个顺串，然后套用公式再需要 9 趟以完成排序，因此共需要 10 趟操作。

### 15.2.2 优化目标

显然这一基础算法还有很多值得优化的地方，PPT 第 4 页给出了一些优化方向，分别对应后面要讨论的优化策略，那么闲话少说，直接进入优化策略的讨论。

## 15.3 多路合并

第一个优化方向就是减少工作的趟数，显然如果使用  $k$  路的归并，也就是每次归并  $k$  条纸带上对应位置的顺串，那么每次合并后顺串长度增加  $k$  倍，因此加上初始的 1 趟，只需要  $1 + \lceil \log_k(N/M) \rceil$  趟即可完成排序，减少了趟数，因此减少了磁带移动的次数，从而减少总时间。这样的算法称为  $k$  路合并。

$k$  路合并有一个实现上需要注意的点，因为是  $k$  个顺串要合并，因此需要不断的在  $k$  个元素中选取最小值放到输出的磁带上，这个操作可以使用优先队列来实现，PPT 第 5 页给出了这样的例子。

## 15.4 多相 (Polyphase) 合并

尽管多路合并看起来很有吸引力，但是需要注意的是， $k$  路合并需要的磁带数目是  $2k$ ，因为每一趟工作需要  $k$  个输入磁带和  $k$  个输出磁带。因此，如果  $k$  很大，那么磁带数目就会很多，这是不太合理的（后面会解释为什么不合理）。因此讨论另一种优化方案：多相合并，能够大大减小归并需要的磁带数目： $k$  路合并只需要  $k+1$  条磁带。

作为例子，下面阐述只用三盘磁带如何完成 2 路合并。设有三盘磁带  $T_1$ 、 $T_2$  和  $T_3$ ，在  $T_1$  上有一个输入文件，它将产生 34 个顺串。一种选择是在  $T_2$  和  $T_3$  的每一盘磁带中放入 17 个顺串。然后可以将结果合并到  $T_1$  上，得到一盘有 17 个顺串的磁带。由于所有的顺串都在一盘磁带上，因此现在必须把其中的一些顺串放到  $T_2$  上以进行另一次的合并。执行合并的逻辑方式是将前 8 个顺串从  $T_1$  拷贝到  $T_2$  并进行合并。这样的效果是，为了所做的每一趟合并，不得不附加额外的复制工作，而复制也需要磁头的移动，是很昂贵的操作，因此这种方法并不好。如果接着把所有的步骤都画完，会发现总共需要 1 趟初始 + 6 趟工作，外加 5 次复制操作。

另一种选择是把原始的 34 个顺串不均衡地分成两份。设把 21 个顺串放到  $T_2$  上而把 13 个顺串放到  $T_3$  上。然后，在  $T_3$  用完之前将 13 个顺串合并到  $T_1$  上。然后可以将具有 13 个顺串的  $T_1$  和 8 个顺串的  $T_2$  合并到  $T_3$  上。此时，合并 8 个顺串直到  $T_2$  用完为止，这样，在  $T_1$  上将留下 5 个顺串而在  $T_3$  上则有 8 个顺串。然后，再合并  $T_1$  和  $T_3$ ，等等，直到合并结束。PPT 第 6 页给出了动态图示展示上述全部过程。

顺串最初的分配有很大的关系。例如，若 22 个顺串放在  $T_2$  上，12 个在  $T_3$  上，则第一趟合并后得到  $T_1$  上的 12 个顺串以及  $T_2$  上的 10 个顺串。在另一次合并后， $T_1$  上有 10 个顺串而  $T_3$  上有 2 个顺串。此时，进展的速度慢了下来，因为在  $T_3$  用完之前只能合并两组顺串。这时  $T_1$  有 8 个顺串而  $T_2$  有 2 个顺串。同样，只能合并两组顺串，结果  $T_1$  有 6 个顺串且  $T_3$  有 2 个顺串。再经过三趟合并之后， $T_2$  还有 2 个顺串而其余磁带均已没有任何内容。此时必须将一个顺串拷贝到另外一盘磁带上，然后结束合并，所以非常复杂。

事实上，最初给出的  $21 + 13$  的分配是最优的，否则都可能出现上面的需要复制或者有长短不对齐导致浪费的情况。如果顺串的个数是一个斐波那契数  $F_n$ ，那么分配这些顺串最好的方式是把它们分裂成两个斐波那契数  $F_{n-1}$  和  $F_{n-2}$ 。否则，为了将顺串的个数补足成一个斐波那契数就必须用一些哑顺串 (dummy run) 来填补磁带。

还可以把上面的做法扩充到  $k$  路合并，此时需要第  $k$  阶斐波那契数用于分配顺串，其中  $k$  阶斐波那契数定义为

$$F^{(k)}(n) = F^{(k)}(n-1) + F^{(k)}(n-2) + \cdots + F^{(k)}(n-k)$$

辅以适当的初始条件  $F^{(k)}(1) = F^{(k)}(2) = \cdots = F^{(k)}(k-2) = 0$ ,  $F^{(k)}(k-1) = 1$ 。这样就可以用  $k+1$  盘磁带完成  $k$  路合并。具体操作需要说明一下，因为并不是那么平凡的。事实上经过第一步拆分之后，每个磁带上的顺串个数应当分别为

$$\text{磁带1: } F^{(k)}(n-1) + F^{(k)}(n-2) + \cdots + F^{(k)}(n-k)$$

$$\text{磁带2: } F^{(k)}(n-1) + F^{(k)}(n-2) + \cdots + F^{(k)}(n-k+1)$$

...

$$\text{磁带}k: F^{(k)}(n-1)$$

$$\text{磁带}k+1: 0$$

然后合并磁带 1 到  $k$  的前  $F^{(k)}(n-1)$  个顺串，将合并后的结果放到磁带  $k+1$  上，此时不同磁带上的顺串个数变为

$$\text{磁带1: } F^{(k)}(n-2) + F^{(k)}(n-3) + \cdots + F^{(k)}(n-k)$$

$$\text{磁带2: } F^{(k)}(n-2) + F^{(k)}(n-3) + \cdots + F^{(k)}(n-k+1)$$

...

$$\text{磁带}k: 0$$

$$\text{磁带}k+1: F^{(k)}(n-1) = F^{(k)}(n-2) + F^{(k)}(n-3) + \cdots + F^{(k)}(n-k-1)$$

此时的顺串个数分布与之前的是类似的，只是下标减了 1，因此可以继续合并除磁带  $k$  外每个磁带的前  $F^{(k)}(n - 2)$  个顺串，将合并后的结果放到磁带  $k$  上，以此类推，直到顺串个数为 1，此时就完成了排序。

## 15.5 缓存并行处理

PPT 第 8-9 页考虑了实际实现外部排序的时候的一个问题。在实现外部排序的时候，肯定是一块一块地读入数据的，然后并不是每比较了一次就要往磁盘写一个元素，这样每次比较完就要等磁盘处理很长时间才能进行下一次比较，是非常耗时间的。实际上在 2 路合并中，应该是把内存划分为输入 2 个缓存区（buffer），输出 1 个缓存区，输入缓存区用来存放从输入磁盘读入的数据，然后两个输入缓存区的数据比较之后的排序结果先放到输出缓存区，当输出缓存区满了之后再一次性写入输出磁盘。

然而这样的实现仍然存在问题，那就是当输出缓存区要写回磁盘的时候，内存里也是什么事情都做不了。所以解决方案是，拆分成两个输出缓存区，当其中一个满了写回磁盘的时候，另一个输出缓存区继续接收数据，这样就可以实现并行处理了——一个在内存里操作，一个进行 I/O 交互。

事实上输入缓存也是一样的，如果仍然是 2 个输入缓存，当所有输入缓存中的数据都比较完了，这时也要被迫停止等待新的数据从输入磁盘中读入。因此要进一步划分为 4 个输入缓存，这样当其中两个输入缓存中的数据正在比较的时候，另外两个输入缓存可以同时并行地读入新的数据。

事实上这里就可以回答前面为什么  $k$  太高时， $k$  路合并尽管减少了趟数，但是并不一定更优的原因了。因为在  $k$  路合并中，根据前面的讨论，应当讲整个内存区域划分成  $2k$  个输入缓存区和 2 个输出缓存区，这样当  $k$  很大的时候，输入缓存就会被划分得很细，一次能读入输入缓存的数据量就会减小（也就是 block 大小降低），那么 I/O 操作就会变多，因此  $k$  太大的时候尽管趟数降低导致 I/O 成本降低，但并不一定更优。因此这其中一定有一个最优的  $k$ ，当然这是与具体的机器硬件有关的。

## 15.6 替换选择

最后要考虑的优化方向是优化顺串的构造。迄今用到的策略是每个顺串的大小都是内存的大小，然而事实上这一长度还可以进一步扩展。原因在于某个内存位置中的元素一旦写入磁盘了，这个内存位置就可以空出来给数组后续的元素使用。只要后续的元素比现在写入的更大，就可以继续往顺串后面添加该元素。PPT 第 10 页给出了很清晰的动态例子，首先在内存中维护一个优先队列，每次从队列中取出比当前顺串中最后一个元素更大的最小的元素写入磁盘，然后再从磁盘中读入一个元素放入队列中，直到队列中的元素都比顺串的最后一个要小，就开启一个新的顺串。

上述方法通常称为替换选择（replacement selection）。显然，当原数组的顺序已经比较接近最终的顺序时，替换选择能够得到更长的顺串。当然，在平均情况下，替换选择生成的顺串长度为  $2M$ ，其中  $M$  是内存的大小。这意味着使用替换选择算法能使得顺串的数量降低，因此之后合并的趟数也会减少，从

而减少了 I/O 操作的次数。

当然，在生成了不同长度的顺串之后，还需要考虑如何合并这些顺串是最优的。PPT 第 11 页给出了一个例子，显然最优解就是哈夫曼树，因为目标是想让长的顺串尽可能少地参与合并，那么显然贪心地不断选择最小的两个顺串合并是最优的，当然感兴趣的读者可以用之前贪心算法的贪心选择性质和最优子结构两步证明套路来证明这一点，实际上贪心选择性质的证明，只需要交换两个顺串的合并次序发现长的放后面合并一定会让整体操作变小即可证明，最优子结构仍然是用反证法，读者只需要回顾贪心算法一节就能知道如何严谨证明，这里就不再赘述了，因为其实很显然。

祝贺大家顺利完成了这门课程的学习，希望这本讲义能够提供理解上的帮助，给予一些启发。尽管这门课的学习充满了痛苦，但我相信只要是平常认真学习了这些内容的就一定能有所收获。祝愿大家在期末考试中取得好成绩，再见！