算法分析与设计

1绪论

数据结构：研究数据的逻辑结构、物理结构及其操作的学科
程序 = 数据结构 + 算法 + 语言 + 程序设计方法
数据结构是算法实现的 **基础**，算法总是依赖于数据结构来**实现**
人工智能的三要素: 数据、算法、算力
算法由操作、控制结构、数据结构三要素组成
算法就是问题的程序化解决方案；算法就是一个定义良好的可计算过程，它取一个或者一组值作为输入，并产生出一个或者一组值作为输出。因此，算法是一系列的计算步骤，用来将输入数据转换成输出结果。

算法具有的特征：

输入、输出、确定性、有穷性、正确性、通用性。

1. 输入：一个算法具有零个或者多个取自指定集合的输入值；
2. 输出：对算法的每一次输入，算法具有一个或多个与输入值相联系的输出值；
3. 确定性：算法的每一个指令步骤都是明确的；
4. 有限性/有穷性：对算法的每一次输入，算法都必须在有限步骤（即有限时间）内结束；
5. 正确性：对每一次输入，算法应产生出正确的输出值；
6. 通用性：算法的执行过程可应用于所有同类求解问题，而不是仅适用于特殊的输入。

算法与程序的区别：

程序并不都满足算法的所有特征，如有限性特征。算法代表了对特定问题的求解，而程序则是算法在计算机上的实现。

1. 一个程序不一定满足有穷性。例操作系统，只要整个系统不遭破坏，它将永远不会停止，即使没有作业需要处理，它仍处于动态等待中。因此，操作系统不是一个算法。
2. 程序中的指令必须是机器可执行的，而算法中的指令则无此限制。
3. 算法代表了对问题的解，而程序则是算法在计算机上的特定的实现。一个算法若用程序设计语言来描述，则它就是一个程序。

算法的描述

1. 基于文字描述
   优点：容易理解
   缺点：冗长、二义性

2. 基于流程图描述
   优点：流程直观
   缺点：缺少严密性、灵活性
3. 基于程序设计语言
   优点：能由计算机执行
   缺点：抽象性差，对语言要求高
4. 伪代码
   优点：表达能力强、抽象性强、容易理解
   与真实代码的差异：对特定算法的描述更加的清晰与精确；不需要考虑太多技术细节（数据抽象、模块、错误处理等）；用伪代码可以体现算法本质；永远不会过时

算法设计的要求：一个好算法

- 正确性(correctness)
- 可读性(readability)
- 健壮性(robustness)
- 效率与低存储等需求

算法分析

时间复杂度

算法中基本语句重复执行的次数是问题规模n的某个函数f(n),算法的时间量度记作：

$$T(n) = O(f(n))$$

表示随着n的增大，算法执行的时间的增长率和f(n)的增长率相同，称**渐近时间复杂度**

分析方法：

1. 找出语句频度最大的那条语句作为基本语句
2. 计算基本语句的频度得到问题规模n的某个函数f(n)
3. 取其数量级用符号“O”表示

> 例如：分析以下程序段的时间复杂度
>
> ```
> i = 1;
> while(i <= n)
>         i = i * 2;
> ```
>
> $2^{f(n)} \leq n$即$f(n) \leq \log_2 n$，取最大值$f(n) = \log_2 n$
> 所以该程序段的时间复杂度$T(n) = O(\log_2 n)$

$O(1)$常量阶 $O(n)$线性阶 $O(n2)$ 平方阶 $O(n^k)$多项式阶$O(logn)$对数阶 $O(2^n)$指数阶
常 量 阶： $O(1) = O(10)$
多项式阶： $2n^3 + 3n^2 + 4n + 5 = O(n^3)$
应当尽量选择多项式阶$O(n^k)$的算法

时间复杂度的多种表示

1. Θ，既是上界也是下界(tight)，等于的意思。
2. O，表示上界(tightness unknown)，小于等于的意思。
3. o，表示上界(not tight)，小于的意思。
4. Ω，表示下界(tightness unknown)，大于等于的意思。
5. ω，表示下界(not tight)，大于的意思。

总结: O是渐进上界，Ω是渐进下界。Θ需同时满足大O和Ω，称为确界(必须同时符合上界和下界)。O极其有用，因为它表示了最差性能

Letter（字母）	Bound（限制）	Growth（增长）
(theta) Θ	upper and lower, tight[1]	equal[2]
(big-oh) O	upper, tightness unknown	less than or equal[3]
(small-oh) o	upper, not tight	less than
(big omega) Ω	lower, tightness unknown	greater than or equal
(small omega) ω	lower, not tight	greater than

空间复杂度

研究算法的空间效率，只需要分析除输入和算法之外的额外空间。
空间复杂度(space complexity):算法所需存储空间的度量，记作:

$$S(n) = O(f(n))$$

其中n为问题的规模(或大小)

算法要占据的空间：

1. 算法本身要占据的空间，输入/输出，指令，常数，变量等
2. 算法要使用的辅助空间
3. 若额外空间相对于输入数据量是常数，则称该算法**原地工作**

Algorithm	Time Complexity			Space Complexi
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

算法设计

分治法

核心思想：分而治之，各个击破；
分治策略: 将原问题划分为n个规模较小而结构与原问题相似的子问题；递归地解决这些子问题，然后再合并其结果，就得到原问题的解。

1. 分解（Divide）： 将原问题分成成一系列子问题；
2. 解决（Conquer）：递归地解各个子问题。若子问题足够小，则直接求解；
3. 合并（Combine）：将子问题的结果合并成原问题的解

归并排序

1. 分解：把n个元素分成各含n/2个元素的子序列；
2. 解决：用归并排序算法对两个子序列递归地排序；
3. 合并：合并两个已排序的子序列以得到排序结果。

合并排序n个数最坏运行时间：

1. 当$n = 1$时，合并排序一个元素的时间是个常量；

2. 当$n > 1$时，运行时间的分解如下：
   分解：仅仅是计算出子数组的中间位置，需要常量时间，$D(n) = \Theta(1)$；
   解决：递归地求解两个规模为$n/2$的子问题，时间为$2T(n/2)$；
   合并：MERGE过程的运行时间为$C(n) = \Theta(n)$。

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

2排序

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	O(n²)	O(n)	O(n²)	O(1)	In-place	稳定
选择排序	O(n²)	O(n²)	O(n²)	O(1)	In-place	不稳定
插入排序	O(n²)	O(n)	O(n²)	O(1)	In-place	稳定
希尔排序	O(n log n)	O(n log² n)	O(n log² n)	O(1)	In-place	不稳定
归并排序	O(n log n)	O(n log n)	O(n log n)	O(n)	Out-place	稳定
快速排序	O(n log n)	O(n log n)	O(n²)	O(log n)	In-place	不稳定
堆排序	O(n log n)	O(n log n)	O(n log n)	O(1)	In-place	不稳定
计数排序	O(n + k)	O(n + k)	O(n + k)	O(k)	Out-place	稳定
桶排序	O(n + k)	O(n + k)	O(n²)	O(n + k)	Out-place	稳定
基数排序	O(n × k)	O(n × k)	O(n × k)	O(n + k)	Out-place	稳定

选泡插，
快归堆希桶计基，
n方n老n一三，
对n加kn乘k，
不稳稳稳不稳稳，
不稳不稳稳稳稳。

快排：

分治思想

防止最坏情况发生

1. 显示地对输入进行排列使得快速排序算法随机化
2. 采用随机取样（random sampling）的随机化技术，随机选主元

堆排

子结点的键值或索引总是小于（或者大于）它的父节点。

线性排序

计数排序、桶排序、基数排序
比较排序：排序结果中各元素的次序基于输入元素间的比较，这类算法是比较排序。最好的时间复杂度为$O(nlogn)$
线性排序：时间复杂度为$O(n)$（突破了比较排序的最好时间复杂度），达到线性，排序不基于比较。线性排序有三种：计数排序、桶排序、基数排序。

排序算法	时间复杂度	空间复杂度	是否稳定
桶排序	$O(n)$	$O(n)$	稳定
计数排序	$O(n)$	$O(n+k)$	稳定
基数排序	$O(nk)$	$O(n)$	稳定

排序算法时间的下界

任何一个以元素比较为基础，对n个元素排序的排序算法，所对应的决策树的高度至少为$\Omega(n\log n)$。
定理1 任意比较排序算法在最坏情况下，都需要做$\Omega(n\log n)$次比较。
堆排序和合并排序是渐近最优的排序算法（它们的运行时间上界$O(n\log n)$与 定理1给出的最坏情况下界$\Omega(n\log n)$一致）
快速排序不是渐近最优的比较排序算法。但是，快速排序其执行效率平均而言较堆排序和合并排序还好。

计数排序

基本思想：

对每一个输入元素x，统计出小于x的元素的个数。然后，根据这一信息直接把元素x放到它在最终输出数组中的位置上。
如果所有待排序的元素均为整数，介于1到k之间。当k = O(n), 时间复杂度：$O(n+k)$
分为三个步骤：计算每个数出现了几次；求出每个数出现次数的前缀和；利用出现次数的前缀和，从右至左计算每个数的排名。

桶排序

桶排序（Bucket Sort）是计数排序的泛化，当元素均匀地分布在某个区间时，可以在O(n)的时间内完成排序。

基本思想：

1. 把区间[0,1)划分成n个相同大小的子区间（称为桶）

2. 将n个输入数分布到各个桶中去
3. 先对各桶中元素进行排序，然后依次列出各桶中的元素

基数排序

Radix Sort(基数排序)：假设所有待排序元素均为整数，至多d位。先按最低有效位进行排序，再按次低有效位排序，重复这个过程，直到对所有的d位数字都进行了排序。
基数排序关键是**按位排序要稳定。**
$O(d(n+k))$如果d是常数，$k=O(n)$，则基数排序能在线性时间内完成排序。

3递归与分治

递归

递归(Recursion)，又译为递回，在数学与计算机科学中，是指在函数的定义中使用函数自身的方法。
直接或间接地调用自身的算法称为递归算法(直接递归、间接递归)。用函数自身给出定义的函数称为递归函数。

递归、递推、迭代

递推Inductive ： 一步步往后，从左往右。即有来无回。
递归Recursive：从最后面一步步往前嵌套，再从最前面一步步往后套。递归=递推+回归。即有来有回。
迭代Iteration：循环执行，每次把前面计算出的值套下一步。迭代是逐渐逼近，用新值覆盖旧值。

基本思想

把规模大的问题转化为规模小的相似的子问题来解决。
在函数实现时，因为解决大问题的方法和解决小问题的方法往往是同一个方法，所以就产生了函数直接或间接调用它自身的情况。这个解决问题的函数**必须有明显的结束条件**，这样就不会产生无限递归的情况。

1. 将求解的较大规模的问题分割成k个更小规模的子问题。
2. 对这k个子问题分别求解。如果子问题的规模仍然不够小，则再划分为k个子问题，如此递归的进行下去，直到问题规模足够小，很容易求出其解为止。
3. 将求出的小规模的问题的解合并为一个更大规模的问题的解，自底向上逐步求出原来问题的解。

分治法的设计思想：将一个难以直接解决的大问题，分割成一些规模较小的相同问题，再各个击破，分而治之。

边界条件与递归方程是递归函数的二个要素，递归函数只有具备这两个要素，才能在有限次计算后得出结果。

> **Fibonacci数列：**
>
> $$F(n):\begin{cases}1 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

总结

优点：结构清晰，可读性强，容易用数学归纳法来证明算法的正确性，能为设计算法、调试程序带来很大方便。
缺点：递归算法的运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多。
解决方法：
在递归算法中消除递归调用，使其转化为非递归算法。

1. 采用一个用户定义的栈来模拟系统的递归调用工作栈。该方法通用性强，但本质上还是递归，只不过人工做了本来由编译器做的事情，优化效果不明显。
2. 用递推来实现递归函数。
3. 通过变换能将一些递归转化为尾递归，从而迭代求出结果。

分治

1. 分解，将原问题分解成若干个与原问题结构相同但规模较小的子问题；
2. 解决，解决这些子问题。如果子问题规模足够小，直接求解，否则递归地求解每个子问题；
3. 合并，将这些子问题的解合并起来，形成原问题的解。

分治法所能解决的问题一般具有以下几个特征

1. 该问题的规模缩小到一定的程度就可以容易地解决；
2. 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质；
3. 利用该问题分解出的子问题的解可以合并为该问题的解；
4. 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子问题。

优点：能简单地求解复杂的问题；并行性(并行计算、多处理器系统)；内存访问(利用内存缓存机制，不需要访问存取速度较慢的主存)
缺点：分治法不能适应于所有问题；递归的效率较慢(具体的实现方式)；分治法比迭代方法更复杂(例子：n个数求和)。

矩阵连乘

传统方法$O(n^3)$，分治法$O(n^{2.81}) \Longrightarrow O(n^{2.376})$

尾递归

尾递归函数中，递归调用是递归函数中的最后一条指令，并且在函数中只有一次递归调用。

4动态规划

1. 多阶段决策过程
   问题的活动过程分为若干相互联系的阶段，任一阶段i以后的行为仅依赖于i阶段的过程状态，而与i阶段之前的过程如何达到这种状态的方式无关。在每一个阶段都要做出决策，这一系列的决策称为多阶段决策过程 (Multistep Decision Process, MDP) 。
2. 最优化问题
   问题的每一阶段可能有多种可供选择的决策，必须从中选择一种决策。各阶段的决策构成一个决策序列。决策序列不同，所导致的问题的结果可能不同。
   过程的**最优决策序列**具有如下性质：无论过程的初始状态和初始决策是什么，其余的决策都必须相对于初始决策所产生的状态构成一个最优决策序列。
3. 多阶段决策的最优化问题
   求能够获得问题最优解的决策序列——最优决策序列。

利用动态规划求解问题的前提

1. **证明问题满足最优性原理**。如果对所求解问题证明满足最优性原理，则说明用动态规划方法有可能解决该问题。
2. **获得问题状态的递推关系式**。获得各阶段间的递推关系式是解决问题的关键。

动态规划

基本概念

1. 状态：表示每个阶段开始时，问题或系统所处的客观状况。状态既是该阶段的某个起点，又是前一个阶段的某个终点。通常一个阶段有若干个状态。
   状态无后效性：如果某个阶段状态给定后，则该阶段以后过程的发展不受该阶段以前各阶段状态的影响，也就是说状态具有马尔科夫性。
2. 策略：各个阶段决策确定后，就组成了一个决策序列，该序列称之为一个策略。由某个阶段开始到终止阶段的过程称为子过程，其对应的某个策略称为子策略。

总体思想

动态规划的思想实质是分治思想和解决冗余（**DP解决冗余数据问题**）
动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题。动态规划法用一个表来记录所有已解决的子问题的答案。

步骤

1. 找出最优解的性质，并刻划其结构特征。$\Longrightarrow$ **划分子问题**
2. 递归地定义最优解的值。$\Longrightarrow$ **给出最优解的递归式**
3. 按自底向上的方式计算最优解的值。
4. 由计算出的结果构造一个最优解。

适用条件

适合采用动态规划方法的最优化问题中的两要素：**最优子结构**和**重叠子问题**

最优子结构

如果问题的最优解是由其子问题的最优解来构造的，则称该问题具有最优子结构。

重叠子问题

递归算法求解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次，该性质称为子问题的重叠性质。

动态规划算法，充分利用重叠子问题，对每一个子问题只解一次，而后将其解保存在一个表格中，当再次需要解此子问题时，只是简单地用常数时间查看一下结果。
通常不同的子问题其个数随问题的大小呈多项式增长。因此用动态规划算法只需要多项式时间，从而获得较高的解题效率。

设计技巧

动态规划的设计技巧：**阶段的划分、状态的表示和存储表的设计**；

总结

与非线性规划相比，动态规划的优点：

1. 易于确定全局最优解。动态规划方法是一种逐步改善法，它把原问题化为一系列结构相似的最优化子问题，而每个子问题的变量个数比原问题少的多，约束集合也要简单得多。
2. 能得到一簇解，有利于分析结果
3. 能利用经验，提高求解的效率。动态规划方法反映了过程逐段演变的前后联系，较之非线性规划与实际过程联系得更紧密。

缺点：

1. 没有一个统一的标准模型可供应用。
2. 应用的局限性。要求状态变量满足“无后效性”条件，不少实际问题在取其自然特征作为状态变量往往不能满足这条件。
3. 在数值求解中，存在“维数障碍”。在计算机中，每递推一段，必须把前一段的最优值函数在相应的状态集合上的全部值存入内存中。当维数增大时，所需的内存量成指数倍增长。

5贪心

旅行商问题(Traveling Salesman Problem, TSP)

需要访问所有顶点，不能有重复

中国邮递员问题 (Chinese Postman Problem, CPP)

需要访问所有边，可以重复

最优化问题

问题有n个输入，问题的解是由这n个输入的某个子集组成，这个子集必须满足某些事先给定的条件。

- **约束条件**：子集必须满足的条件；
- **可行解**：满足约束条件的子集；可行解可能不唯一；
- **目标函数**：用来衡量可行解优劣的标准，一般以函数形式给出；
- **最优解**：能够使目标函数取极值（极大或极小）的可行解。

贪心方法：

一种改进的分级的处理方法，可对满足上述特征的某些问题方便地求解，属于近似算法。即每次在做选择时，总是先选择具有相同特征的那个解，即“贪心解”。
贪心方法问题的一般特征：问题的解是由n个输入的、满足某些事先给定的条件的子集组成。
能够得到某种量度意义下的最优解的**分级处理**方法称为贪心方法。
使用贪心策略求解的关键：**选取能够得到问题最优解的量度标准**。

贪心算法

贪心算法总是作出在当前看来最好的选择，并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。单源最短路经问题，最小生成树问题等
在一些情况下，贪心算法不能得到整体最优解，但其最终结果是最优解的近似。

贪心VS动态规划

贪心算法和动态规划算法都要求问题具有最优子结构性质，但是两者存在着巨大的差别。

1. 动态规划是先分析子问题，再做选择。而贪心算法是先做贪心选择，做完选择后，生成子问题，然后再去求解子问题；
2. 动态规划每一步可能会产生多个子问题，而贪心算法每一步只会产生一个子问题(为非空)；
3. 从特点上看，动态规划是**自底向上**解决问题，而贪心算法是**自顶向下**解决问题。

基本思想

从问题的某一个初始解出发，通过一系列的贪心选择，即当前状态下的局部最优选择，逐步逼近给定的目标，尽可能快地求得更好的解。
在贪心算法(Greedy Method)中采用**逐步构造/分级最优解**的方法。在每个阶段，都作出一个按某个评价函数最优的决策，该最优评价函数称为**贪心准则(Greedy Criterion)**
贪心算法的正确性，要证明按贪心准则求得的解是全局最优解。

基本步骤

1. 决定问题的最优子结构；
2. 设计出一个递归解；
3. 证明在递归的任一阶段，最优选择之一总是贪心选择, 那么做贪心选择总是安全的。

4. 证明通过做贪心选择，所有子问题(除一个以外)都为空, 即只产生一个子问题。
5. 设计出一个实现贪心策略的递归算法。
6. (性能角度) 将递归算法转换成迭代算法。

基本要素

贪心选择性质和**最优子结构性质**

贪心选择性质

所谓贪心选择性质是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。
对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的整体最优解，否则得到的是近优解。

最优子结构性质

当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。
问题的最优子结构性质是该问题可用动态规划算法或贪心算法求解的关键特征。但是，需要注意的是，**并非所有具有最优子结构性质的问题都可以采用贪心策略来得到最优解。**
贪心算法只需考虑一个选择(即贪心的选择)；在做贪心选择时，子问题之一必须是空的，因此只留下一个非空子问题。

与递归、分治的联系

- 分治策略用于解决原问题与子问题结构相似的问题，对于各子问题相互独立的情况，一般用递归实现；
- 动态规划用于解决子问题有重复求解的情况，既可以用递归实现，也可以用迭代实现；
- 贪心算法用于解决具有贪心选择性质的问题，既可以用递归实现，也可以用迭代实现，因为很多递归贪心算法都是尾递归，很容易改成迭代贪心算法；
- 递归是实现手段，分治策略是解决问题的思想，动态规划和贪心算法很多时候会使用记录子问题运算结果的递归实现。

Dijkstra

基本思想

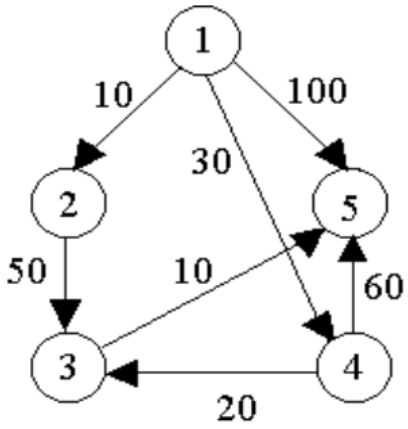
设置顶点集合S并不断地作贪心选择来扩充这个集合。一个顶点属于集合S当且仅当从源到该顶点的最短路径长度已知。

步骤

1. 初始时，S中仅含有源。设u是G的某一个顶点，把从源点到u且中间只经过S中顶点的路称为从源点到u的特殊路径，并用数组dist记录当前每个顶点所对应的最短特殊路径长度。
2. 每次从V-S中取出具有最短特殊路长度的顶点u (贪心策略)，将u添加到S中，同时对数组dist作必要的修改(why?)。
3. 直到S包含了所有V中顶点，此时，dist就记录了从源到所有其它顶点之间的最短路径长度。

画表格

例如，对右图中的有向图，应用迪杰斯特拉算法计算从源顶点1到其它顶点间最短路径的过程列如下表所示。



迭代	S	u	dist[2]	dist[3]	dist[4]	dist[5]
初始	{1}	-	10	maxint	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

时间复杂度分析

对于一个具有n个顶点和e条边的带权有向图，如果用带权邻接矩阵表示这个图，那么Dijkstra算法的主循环体需要O(n)时间。这个循环需要执行n-1次，所以完成循环需要O(n2)时间。算法的其余部分所需要时间不超过O(n2)。

最小生成树

设$G=(V,E)$是无向连通带权图，即一个网络。E中每条边(v,w)的权为$c[v][w]$。如果G的子图G'是一棵包含G的所有顶点的树，则称G'为G的生成树。生成树上各边权的总和称为该生成树的耗费。在G的所有生成树中，耗费最小的生成树称为G的最小生成树(MST: minimum-cost spanning tree)。

Prim算法vs Kruskal算法

算法思想的角度

- 两者都是贪心的思想，但是考虑的角度不同；
- Prim算法从**顶点的角度出发**，每次选择距离当前节点最近的节点加入解集合中，直到所有节点都加入。
- Kruskal算法**从边的角度出发**，每次总是选择权重最小的边加入，直到加入n-1条边为止。（如果加入一条边后出现回路，skip这条边）
- 对于图的生成树，主要有无向图的最小生成树、次小生成树和有向图的最小树形图三种问题。

从代码实现的角度

- Prim算法需要维持一个堆数据结构

- Prim算法在实现中用priority_queue比较多
- Kruskal算法需要union-find的数据结构

总结

- 适合用贪心法的问题应具有最优子结构性质：原问题的最优解包含其子问题的最优解。
- 不是所有的问题都能找到贪心算法。例如，0/1背包问题
- 对于一个具体问题，要确定它是否具有贪心选择性质，必须证明每一步所作的贪心选择最终导致问题的一个整体最优解。

如何证明?
首先考察问题的一个整体最优解，并证明可修改这个最优解，使其贪心选择开始。而且作了贪心选择后，原问题简化为一个规模更小的类似子问题。然后，用数学归纳法证明，通过每一步作贪心选择，最终可得到问题的一个整体最优解。其中，证明贪心选择后的问题简化为规模更小的类似子问题的关键在于利用该问题的最优子结构性质。

6回溯法

概述

回溯法是一个既带有系统性又带有跳跃性的搜索算法；
系统性：它在包含问题的所有解的解空间树中，按照深度优先的策略，从根结点出发搜索解空间树。
跳跃性：算法搜索至解空间树的任一结点时，判断该结点为根的子树是否包含问题的解，如果肯定不包含，则跳过以该结点为根的子树的搜索，逐层向其祖先结点回溯。否则，进入该子树，继续深度优先的策略进行搜索。
这种以深度优先的方式系统地搜索问题的解的算法称为回溯法，它适用于解一些组合数较大的问题

问题的解空间

- 问题的解向量：回溯法希望一个问题的解能够表示成一个n元式$(x1, x2, ..., xn)$的形式。
- 显约束：对分量xi的取值限定。
- 隐约束：为满足问题的解而对不同分量之间施加的约束。

解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）。

基本思想

1. 搜索从开始结点(根结点) 出发，以深度优先搜索整个解空间。
2. 这个开始结点成为活结点，同时也成为当前的扩展结点。在当前的扩展结点处，搜索向纵深方向移至一个新结点。这个新结点就成为新的活结点，并成为当前扩展结点。
3. 如果在当前的扩展结点处不能再向纵深方向扩展，则当前扩展结点就成为死结点。

4. 此时，应往回移动(回溯)至最近的一个活结点处(回溯点)，并使这个活结点成为当前的扩展结点；直到找到一个解或全部解。

基本步骤

1. 针对所给问题，定义问题的解空间；
2. 确定易于搜索的解空间结构；
3. 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。
   常用剪枝函数：
   1. 用约束函数在扩展结点处剪去不满足约束的子树；
   2. 用限界函数剪去得不到最优解的子树。

解空间树

子集树

当所给的问题是从n个元素的集合S中找出满足某种性质的子集时，相应的解空间树称为子集树。
子集树通常有2^n个叶子结点，其总结点个数为$2^{n+1}-1$，遍历子集树时间为$\Omega(2^n)$。如0-1背包问题，叶结点数为2^n，总结点数2^{n+1}。

排列树

当所给问题是确定n个元素满足某种性质的排列时，相应的解空间树称为排列树。
如果解空间是由n个元素的排列形成，即n个元素的每一个排列都是解空间中的一个元素，那么，最后解空间的组织形式是排列树。
排列树通常有$n!$个叶子结点，因此，遍历排列树需要$\Omega(n!)$的计算时间。如TSP问题(Traveling Salesman Problem，推销员问题)，叶结点数为$n!$，遍历时间为$\Omega(n!)$。

用回溯法解题的一个显著特征是**在搜索过程中动态产生问题的解空间**。在任何时刻，算法只保存从根结点到当前扩展结点的路径。如果解空间树中从根结点到叶结点的最长路径的长度为$h(n)$，则回溯法所需的计算空间通常为$O(h(n))$。而显式地存储整个解空间则需要$O(2^{h(n)})$或$O(h(n)!)$内存空间。

效率分析

回溯法的效率在很大程度上依赖于以下因素：

1. 产生$x[t]$的时间；
2. 满足显约束的$x[t]$值的个数；
3. 计算约束函数constraint的时间；
4. 计算上界函数bound的时间；
5. 满足约束函数和上界函数约束的所有$x[k]$的个数。

好的约束函数能显著地减少所生成的结点数，但这样的约束函数往往计算量较大。因此，在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷。
在其它条件相当的前提下，让可取值最少的$x[i]$优先。

7分支限界法

与回溯法的区别

1. 求解目标不同：
   一般而言回溯法的求解目标是找出解空间树中满足约束条件的所有解，而分支限界法的求解目标是尽快地找出解空间树中满足约束条件的一个解。
2. 搜索方法不同：
   回溯法使用深度优先方法搜索，而分支限界一般用宽度优先或最佳优先方法来搜索。
3. 对扩展结点的扩展方式不同：
   分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点；
4. 存储空间的要求不同：
   分支限界法的存储空间比回溯法大得多，因此当内存容量有限时，回溯法解决问题成功的可能性更大。

基本思想

分支限界法**常以广度优先或以最小耗费(最大效益)优先**的方式搜索问题的解空间树。

1. 对已处理的各结点根据限界函数估算目标函数的可能取值，
2. 从中选出目标函数取得极大(极小) 值的结点优先进行广度优先搜索，
3. 不断地调整搜索方向，尽快找到解，裁剪那些不能得到最优解的子树以提高搜索效率。

特点：限界函数一般是基于问题的目标函数，适用于求解最优化问题。

搜索策略

在扩展结点处，首先生成其所有的儿子结点(分支)，然后从当前的活结点表中选择下一个扩展结点。
为了有效地选择下一个扩展结点，以加速搜索的进程，在每一个活结点处，计算一个函数值(优先值)，并根据这些已计算出的函数值，从当前活结点表中选择一个最有利的结点作为扩展结点，使搜索朝着解空间树上有最优解的分支推进，以便尽快地找出一个最优解。

求解步骤

1. 定义解空间(对解编码)；
2. 确定解空间的树结构；
3. 按BFS等方式搜索：
   1. 每个活结点仅有一次机会变成扩展结点；
   2. 由扩展结点生成一步可达(即宽度搜索)的新结点；
   3. 在新结点中，删除不可能导出最优解的结点； **限界策略**
   4. 将剩余的新结点加入活动表(队列)中；

5. 从活动表中选择结点再扩展； **分支策略**
6. 直至活动表为空；

常见的两种分支限界法

队列式(FIFO)分支限界法

从活结点表中取出结点的顺序与加入结点的顺序相同，因此活结点表的性质与队列相同；

优先队列(代价最小或效益最大)分支限界法

每个结点都有一个对应的耗费或收益，以此决定结点的优先级：

- 如果查找一个具有**最小耗费**的解，则活结点可用**小根堆**来建立，下一个扩展结点就是具有最小耗费的活结点；
- 如果希望搜索一个具有**最大收益**的解，则可用**大根堆**来构造活结点表，下一个扩展结点是具有最大收益的活结点。

解空间树的动态搜索

1. 回溯求解0/1背包问题，剪枝能减少搜索空间，但整个搜索按深度优先机械地进行，是盲目搜索（因为该方法不能预测本结点以下的结点进行得如何）。
2. 回溯求解TSP问题也是盲目的（虽有目标函数，也只有找到一个可行解后才有意义）。

分支限界法**首先**确定一个合理的限界函数，并根据限界函数确定目标函数的界[down, up]；**然后**按照广度优先策略遍历问题的解空间树，在某一分支上，依次搜索该结点的所有孩子结点，**分别估算这些孩子结点的目标函数的可能取值（注意：对于最小化问题，估算结点的down，对最大化问题，估算结点的up）。**
如果某孩子结点的目标函数值超出目标函数的上界或下界，则将其丢弃（即基于该结点生成的解不会比目前已得的更好），否则入待处理表。
从0-1背包问题的搜索过程可看出：与回溯法相比，分支限界法可**根据限界函数不断调整搜索方向**，选择**最可能得最优解的子树**优先进行搜索→找到问题的解。

8随机算法

$$
\text{算法}\begin{cases}\text{确定的}\begin{cases}\text{传统算法：以100\%成功概率求解问题的最优解}\\ \text{近似算法：一种对解质量的让步}\end{cases}\\ \text{非确定的}\begin{cases}\text{随机算法：一种对求解成功概率和解质量的让步}\\ \text{智能算法：遗传算法、模拟退火法、拟人拟物算法等}\end{cases}\end{cases}
$$

随机算法是指在算法中执行某些步骤或某些动作时，所进行的选择是随机的。
三要素：输入实例、随机源和停止准则。
特点：简单、快速和易于并行化。

常见的4类随机算法

数值随机化算法：

常用于**数值问题**的求解，所得到的往往是**近似解**，解的精度随着计算时间增加而不断提高；
在许多情况下，计算问题的精确解是不可能或者没有必要的，数值随机算法能得到相当满意的解；**最优化问题的近似解**

蒙特卡罗算法：

用于**求问题的准确解**。该方法总可以得到问题的解，但是该解**未必是正确的**。求得正确解的概率依赖于算法所用的时间。比较难以判断解是否正确；

拉斯维加斯算法：

不会得到不正确的解，但是有时会找不到解。找到正确解的概率随着所用的计算时间的增加而提高。对任一实例，反复调用算法求解足够多次，可使求解失效的概率任意小；
核心思想：随机生成答案并检测答案正确性。

舍伍德算法：

总能求得问题的一个解，且所求得的解总是正确的。它用于改善确定性算法的时间复杂度。
当一个确定性算法最坏情况下的计算复杂性与其在平均情况下的计算复杂性有较大差别时，可在这个确定性算法中引入随机性将它改造成一个舍伍德算法，消除或者减少这种差别。
其主要思想是：避免算法最坏情况，设法消除最坏情形和特定实例之间的关联。
随机快排属于舍伍德算法。

总结

确定型算法 vs 随机算法

- 确定型算法：对某个特定输入的每次运行过程是**可重复的**，运行结果是**一样的**。
- 随机算法：对某个特定输入的每次运行过程是**随机的**，运行结果也可能是**随机的**。

随机算法的优势

- 在许多情况下，随机性选择通常比最优选择省时，可以很大程度上降低算法的计算复杂度。
- 在运行时间或者空间需求上随机算法比确定型算法往往有较好的改进。
- 随机算法设计简单。

随机算法的比较

算法	有解	解正确	改进复杂性的优势	设计要点
舍伍德	有	是	可能改善最坏情况	随机选择
拉斯维加斯	不一定	是	可能改善平均情况	与确定算法相结合
蒙特卡罗	有	不一定	解决目前困难的问题	概率$>1/2$多次执行

随机序列提高算法的平均复杂度—— 舍伍德(Sherwood)算法
随机生成答案并检测答案正确性—— 拉斯维加斯(Las Vegas)算法
考虑正确几率的算法—— 蒙特卡罗(Monte Carlo)算法
蒙特卡罗算法：采样越多，越近似最优解；拉斯维加斯算法：采样越多，越有机会找到最优解。

---

9 NP理论

计算模型

在进行问题的计算复杂性分析之前，首先必须建立求解问题所用的计算模型，包括定义该计算模型中所用的基本运算。
建立计算模型的目的是为了使问题的计算复杂性分析有**一个共同的客观尺度。**
3个基本计算模型：这3个计算模型在计算能力上是等价的，但在计算速度上是不同的。

1. 随机存取机RAM (Random Access Machine)
2. 随机存取存储程序机RASP (Random Access Stored Program Machine)
3. 图灵机 (Turing Machine)。

RAM和RSAP

随机存取机RAM、随机存取存储程序机RASP的差异：
结构：RASP的整体结构类似于RAM，所不同的是RASP的程序是存储在寄存器中的。每条RASP指令占据2个连续的寄存器。第一个寄存器存放操作码的编码，第二个寄存器存放地址. RASP指令用整数进行编码。
复杂性：不管是在均匀耗费标准下，还是在对数耗费标准下，RAM程序和RASP程序的复杂性只差一个常数因子。在一个计算模型下$T(n)$时间内完成的输入-输出映射可在另一个计算模型下模拟，并在$kT(n)$时间内完成。其中k是一个常数因子。空间复杂性的情况也是类似的。

图灵机

确定型图灵机DTM：若对任一个状态和符号，要执行的动作都是唯一的
非确定型图灵机NTM：若执行的动作是有穷多个可供选择

- 与RAM模型类似，图灵机既可作为语言接受器，也可作为计算函数的装置。
- 图灵机M的**时间复杂性$T(n)$是它处理所有长度为n的输入所需的最大计算步数。**如果对某个长度为n的输入，图灵机不停机，$T(n)$对这个n值无定义。
- 图灵机的**空间复杂性$S(n)$是它处理所有长度为n的输入时，在k条带上所使用过的方格数的总和。**如果某个读写头无限地向右移动而不停机，$S(n)$也无定义。

P 、NP及NPC类问题

P类问题：

一类问题的集合，对其中的任一问题，都存在一个确定型图灵机M和一个多项式p，对于该问题的任何(编码)长度为n的实例，M都能在$p(n)$步内，给出对该实例的回答。即：**多项式时间内可被解决的问题**

NP类问题：

一类问题的集合，对其中的任一问题，都存在一个非确定型图灵机M和一个多项式p，对于该问题的任何(编码) 长度为n的实例，M都能在$p(n)$步内，给出对该实例的回答。

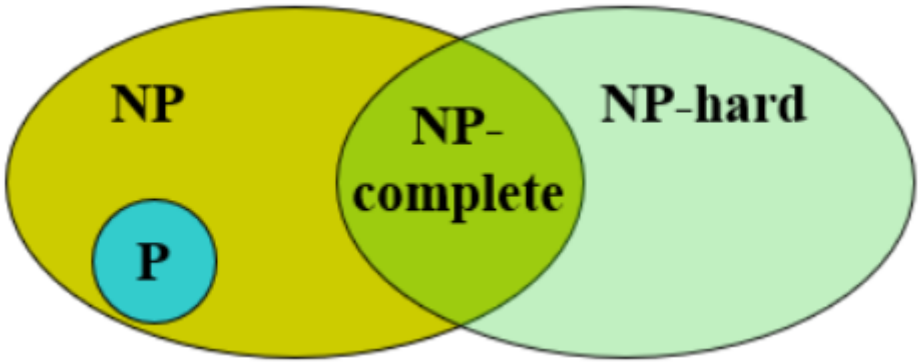
NPC问题：

对于一个(判定性)问题q，若 (1) $q \in NP$; (2) NP中任一问题均可多项式时间多一归约到q，则称问题q为NP-完全的(NP-complete，NPC)

NP-hard问题：

若问题q仅满足条件(2)而不一定满足条件(1)，则问题q称为NP-难的(NP-hard，NPH)。显然：$NPC \subseteq NP-hard$

NPC和NP-hard关系：



NP-hard问题至少跟NPC问题一样难。
NPC问题肯定是NP-hard的，但反之不一定。
所有NP-complete问题都是NP-hard问题 (如：旅行推销员问题)；
但是NP-hard问题不一定是NP-complete问题(如：程式停止问题，它不是NP问题)

NP-完全性理论

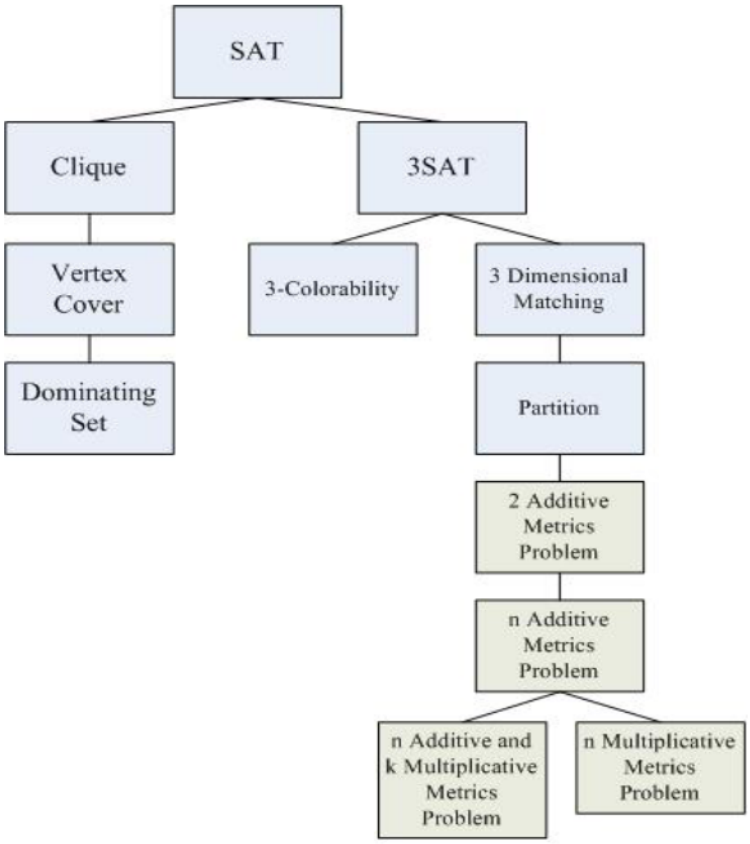
可满足性问题(Satisfiability problem) 是NP完全问题，亦即 SAT∈NPC

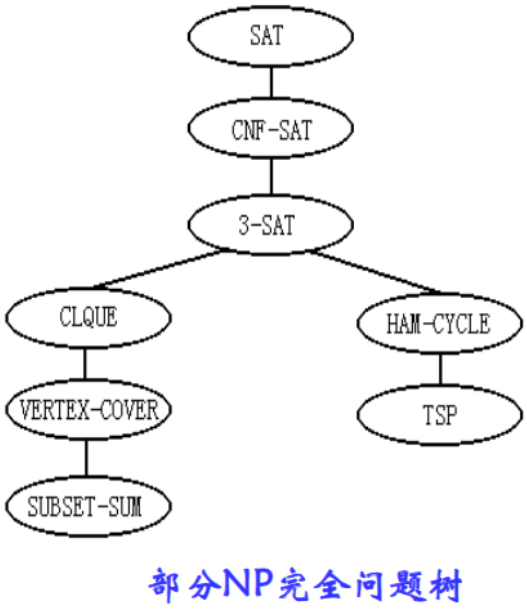
局限性

易解问题：可多项式时间内求解的问题
难解问题：需超多项式时间求解的问题
NP-完全性理论既没有找到第二类问题的多项式时间的算法，也没有证明这样的算法就不存在，而是证明了这类问题计算难度之等价性 (彼此间困难程度相当)。因此，NPC具有如下性质：若其中1个问题多项式可解当且仅当其他所有NPC问题亦多项式可解。

求解

1. 减少搜索量
   简单算法是穷举搜索，时间为指数
   减少搜索量：分枝限界法，隐枚举法、动态规划等。可以提高效率，但时间复杂度不变
2. 优化问题
   降低优化要求，求近似解，以得到一个多项式时间的算法。即：找寻在容许的时间内得到容许精度的近似最优解的算法。





部分NP完全问题树

NP完全问题的近似算法

迄今为止，所有的NP完全问题都还没有多项式时间算法。
对于这类问题，通常可采取以下几种解题策略。
(1)只对问题的特殊实例求解
(2)用动态规划法或分支限界法求解
(3)用概率算法求解
(4)只求近似解
(5)用启发式方法求解

设计思想

- 放弃求解最优解，用近似最优解代替最优解，以此换取：
  算法设计上的简化
  时间复杂性的降低
- 近似算法是可行的：
  问题的输入数据是近似的；

问题的解允许有一定程度的误差；
近似算法可在很短的时间内得到问题的近似解。

总结

近似算法放弃求最优解，用近似解代替最优解，以换取算法设计上的简化和时间复杂性的降低。
近似算法通常采用两个标准来衡量性能：

- 算法的时间复杂性
- 解的近似程度
  - 近似比η
  - 相对误差λ
  - 相对误差界$\varepsilon(n)$
