Combinatorial Optimization

基本思路：将求解实例转化为规模较小的实例，利用递推关系，从初始条件出发求解

最优化原理（Principle of Optimality）：一个n阶段过程的最优策略可以这样构成：首先求出以初始决策 造成的状态s_1为初始状态的$n-1$阶段子过程的最优策略，然后在附加第一阶段效益（或费用）的情形下，从所有可能初始决策得到的解中选择最优者

> 最短路的子路也是最短路

Knapsack

设最优解为$I(k,w)$，构造每一步情况为$C(k,w)$，根据第k个物品是否放入进行递推

背包问题的分枝定界法

- 实例求解过程可用一颗高度为n的二叉树表示。各层依次对应一个物品，最后一层的每个节点对应一个解
- 分枝：每个节点有两条出边连结两个子节点，分别代表该节点下一层对应物品放入背包与不放入背包
- 剪枝：
  - 该枝不存在可行解
  - 该枝不存在最优解
- 定界：
  - 任一个已获得的可行解的目标值均是最优值的下界
  - 对每一枝，求该枝所有可行解目标值的上界，若该上界不大于下界，则该枝不存在更好的可行解

可给出NP-hard问题任一实例最优解的算法总需要指数时间。较为现实的思路是“用精度换时间”，在多项式时间或可接受的实际运行时间内得到一个目标值与最优值较为接近的可行解：

- 近似算法：可以分析时间复杂度
- 启发式算法：不可分析时间复杂度

其中近似算法近似性能用**最坏情况比**来衡量：

任意实例I，算法A给出一个可行解，其目标值为$C^A(I)$，实例I的最优目标值为$C^*(I)$

- 称$r_A=inf\{r\geq 1|C^A(I)\leq C^*(I),\forall I\}$或$r^A=\frac{C^A(I)}{C^*(I)}$为算法的最坏情况界（worst-case ratio）
- 若算法的最坏情况界为r_A，则对该问题的任意实例I，均有$C^A(I)\leq r_AC^*(I)$
- 最坏情况界越接近于1，说明算法给出的可行解目标值越接近于最优值，算法近似性能越好
- 分析较难，适用范围较小，实际上的性能常常被低估
- 平均情况界更难计算
- 还有多项式时间近似方案、完全多项式时间近似方案，算法实现通常较为复杂，运行时间可能很长

相比之下，启发式算法适用范围更广，由于其通过数值模拟实际过程，总是能产出较好的结果

- 充分利用经验和技巧
- 数值模拟，免去理论证明
- 快速 准确 稳健 简单 可移植性强
- 需要测试算法的性能
    - 解的质量：最优解估计，需要估计解的下界，求得最坏情况比
    - 计算资源
    - 稳健性分析
    - 真实性和可重复性
    - 结果、图、表可读性
- 试验实例来源
    - 现实数据
    - 随机生成
    - 实例库

Meta-heuristic 元启发式算法

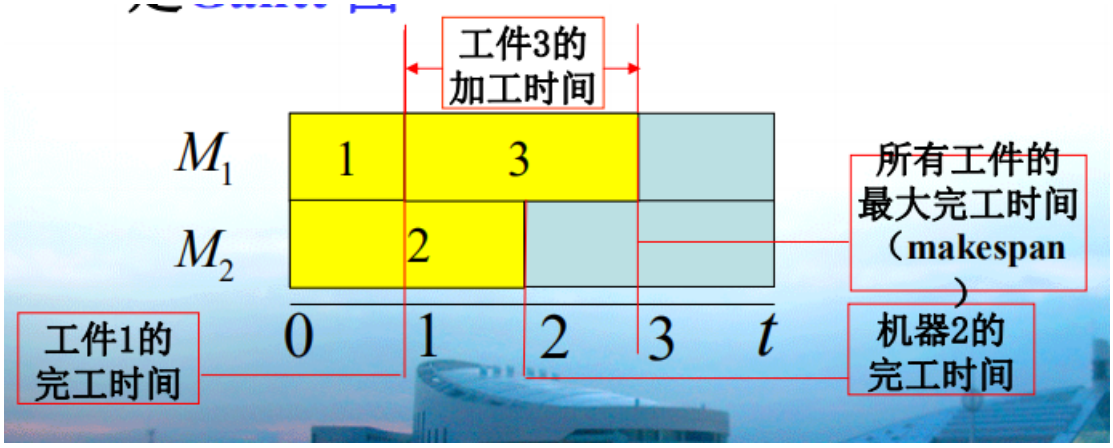
Vehicle Routing Problem

- n个顾客，分别位于地点i，需求为q_i
- K辆车，容量均为Q
- 仓库位于地点O，从地点i到地点j的距离为c_{ij}
- 将顾客分配给各个车，每车配送的顾客需求和不超过Q，确定每辆车从仓库出发，完成分配给该车的所有顾客的配送，最后回到仓库的路线，使得所有车行驶的总路程最短
- NP-Hard问题

Scheduling

- 主要研究如何利用有限资源，在给定的限制条件下，将一批任务安排在某些时间段内完成，并使效益最大
- 习惯上把可用的资源称为机器(machine), 需要完成的任务叫做工件(job)

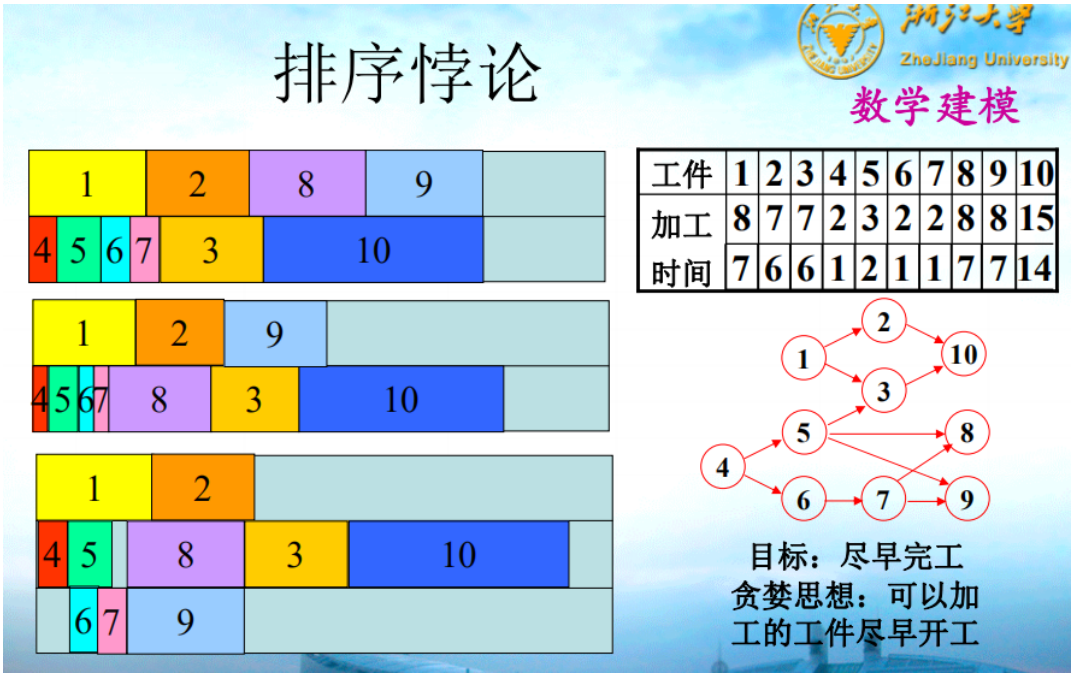
Gantt 图



单台机问题

现有n个工件，工件j的加工时间为p_j，预定交工期（duedate）为d_j。机器在同一时刻只能加工一个工件，工件加工不可中断。如何确定工件的加工顺序，可使得

- 误工（完工时间大于预定交工期）的工件数最少
  - P问题
- 各工件延误时间（完工时间与预定交工期之差）的最大值最小
  - EDD原则：将工件按照预定交工期的非减序排列
- 所有误工工件的总延误时间最小
  - NPC问题

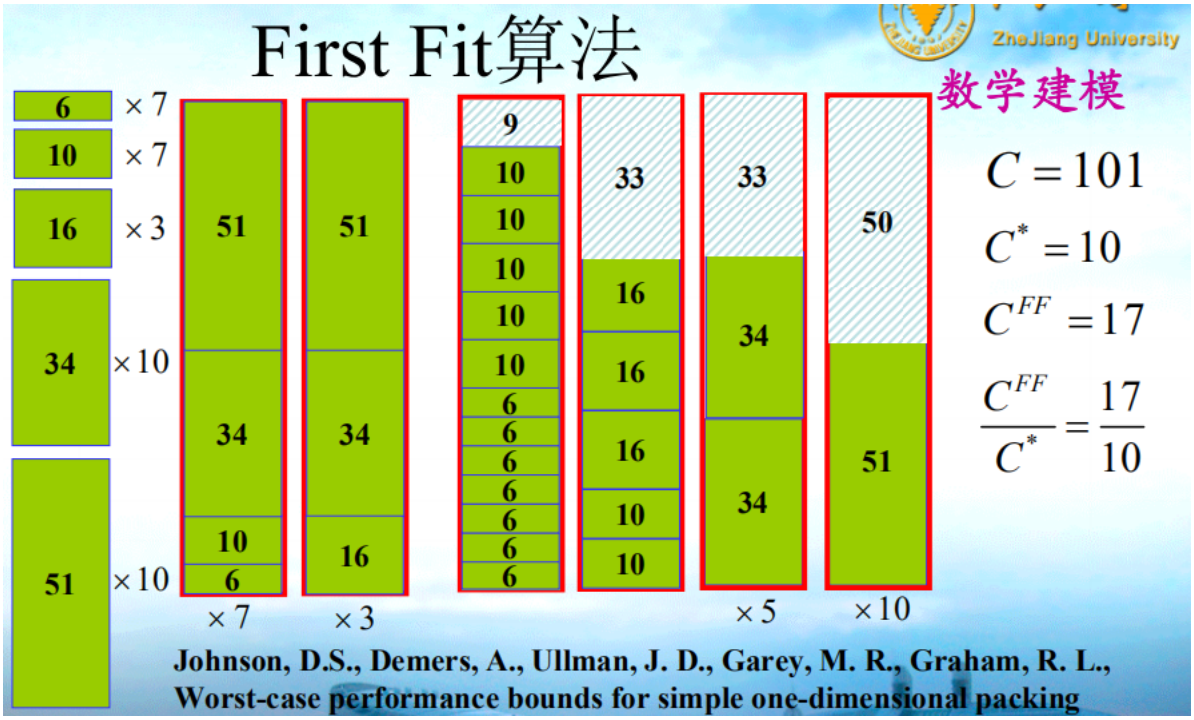


症结：算法有问题

Bin-packing

研究如何将一系列物品放入容量一定的若干箱子中，使得在放入每个箱子中的物品大小之和不超过箱子容量的前提下，所用箱子数尽可能少

- 一维装箱：
  - First Fit（FF）算法：将物品放在按箱子启用顺序第一个能放下的箱子中
  - 先把小的全塞进去，再放大的



用规划的方法很难解决二维、三维的装箱问题

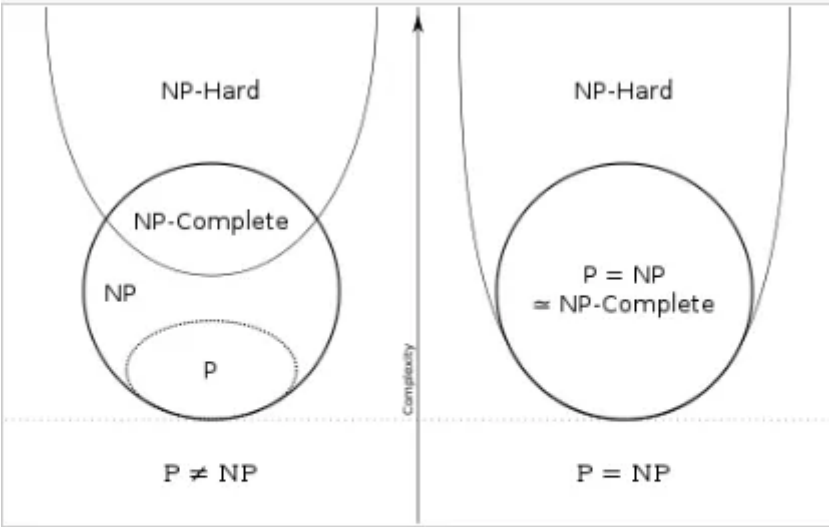
Roof

用不同类型的光伏电池铺设给定长宽的矩形屋顶

- 设类型为i的光伏电池是长为a_i，宽为b_i的矩形，每块该类型电池预期效益为p_i
- 铺设时光伏电池不能互相覆盖，也不能超出屋顶之外，但不必覆盖屋顶所有区域
- 采用怎样的铺设方案可使效益最大
- 二维的背包问题
- 长宽容纳，面积容纳，效益最大
- **用规划的方法很难解决二维、三维的装箱问题**

P NP

$$P \subseteq NP$$



Euler diagram for **P**, **NP**, **NP**-complete, and **NP**-hard set of problems (excluding the empty language and its complement, which belong to **P** but are not **NP**-complete)

NP: **nondeterministic polynomial problem**

P: **polynomial problem**

> 在这里强调（回到我竭力想澄清的误区上），NP问题不是非P类问题。NP问题是指可以在多项式的时间里验证一个解的问题。NP问题的另一个定义是，可以在多项式的时间里猜出一个解的问题。比方说，我RP很好，在程序中需要枚举时，我可以一猜一个准。现在某人拿到了一个求最短路径的问题，问从起点到终点是否有一条小于100个单位长度的路线。它根据数据画好了图，但怎么也算不出来，于是来问我：你看怎么选条路走得最少？我说，我RP很好，肯定能随便给你指条很短的路出来。然后我就胡乱画了几条线，说就这条吧。那人按我指的这条把权值加起来一看，嘿，神了，路径长度98，比100小。于是答案出来了，存在比100小的路径。别人会问他这题怎么做出来的，他就可以说，因为我找到了一个比100 小的解。在这个题中，找一个解很困难，但验证一个解很容易。验证一个解只需要O(n)的时间复杂度，也就是说我可以花O(n)的时间把我猜的路径的长度加出来。那么，只要我RP好，猜得准，我一定能在多项式的时间里解决这个问题。我猜到的方案总是最优的，不满足题意的方案也不会来骗我去选它
>
> 所以要定义NP问题，是因为通常只有NP问题才可能找到多项式的算法。我们不会指望一个连多项式地验证一个解都不行的问题存在一个解决它的多项式级的算法。
>
> 很显然，所有的P类问题都是NP问题。也就是说，能多项式地解决一个问题，必然能多项式地验证一个问题的解——既然正解都出来了，验证任意给定的解也只需要比较一下就可以了。关键是，人们想知道，是否所有的NP问题都是P类问题
>
> NPC问题的定义非常简单。同时满足下面两个条件的问题就是NPC问题。首先，它得是一个NP问题；然后，所有的NP问题都可以约化到它。证明一个问题是 NPC问题也很简单。先证明它至少是一个NP问题，再证明其中一个已知的NPC问题能约化到它（由约化的传递性，则NPC问题定义的第二条也得以满足；至于第一个NPC问题是怎么来的，下文将介绍），这样就可以说它是NPC问题了。　既然所有的NP问题都能约化成NPC问题，那么只要任意一个NPC问题找到了一个多项式的算法，那么所有的NP问题都能用这个算法解决了，NP也就等于P 了。

> 顺便讲一下NP-Hard问题。NP-Hard问题是这样一种问题，它满足NPC问题定义的第二条但不一定要满足第一条（就是说，NP-Hard问题要比 NPC问题的范围广）。NP-Hard问题同样难以找到多项式的算法，但它不列入我们的研究范围，因为它不一定是NP问题。即使NPC问题发现了多项式级的算法，NP-Hard问题有可能仍然无法得到多项式级的算法

P问题一般计算容易，计算类的问题常为P

NP问题一般计算困难，验证容易，需要搜索的常为NP

证明NP-Hard

- 查资料
- 找NP-Hard特殊情况
- 证明归约
    - 由I_1构造I_2
    - 由I_2的算法A解I_2，也可解I_1
    - 证明I_1与I_2会同时成立