算法分析与设计

华中科技大学软件学院

2022年春

贪婪算法

- 通过做出一系列短视的决策来解决问题
- 每个决策本身都能最优地解决某些子问题
- 但这些子问题对整个问题来说未必最优
- 设计的关键：找到一种合适的方法，把问题分解成几个小的部分，然后把它们组合在 一起
- 作业调度、Dijkstra算法、最小生成树

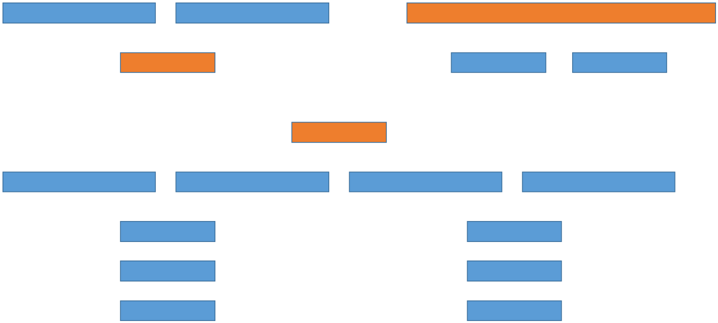
作业调度

- 给定：n个作业和一台机器，作业i有一个开始时间s_i和一个完成时间$f_i \geq s_i$
- 目标：以非重叠的方式找到可以在机器上调度的最大作业子集
- 对于任意两个计划作业i和j，$f_i \leq s_j$或$f_j \leq s_i$
- 贪婪方法：持续作业调度，确保没有新作业与现有的作业重叠。关键在于调度作业的顺序

贪婪策略

- 有几种可能的方法可以做到这一点，每种方法都试图尽量减少每个连续作业可能导致的潜在重叠次数
- 最短作业优先
- 最早到达优先
- 冲突最少优先
- 最早完成时间优先

贪婪策略的影响



最早完成时间优先算法

- 考虑任何不少于k个作业的解决方案S。对k归纳证明，贪婪算法G调度至少k个作业时，前k个作业不晚于所选解中的前k个作业完成。这一结论意味着贪婪算法调度的作业数至少与最优解相同
- 基本情况(k=0)显而易见。假设归纳假设适用于k−1
- 令S_k为S中的第k个作业，G_k是贪婪调度的第k个作业。显然，$s_{S_k} \geq f_{S_{k-1}} \geq f_{G_{k-1}}$。也就是说，$S_k$在$G_{k-1}$完成之后开始。另外，在贪婪中调度$G_{k-1}$时，$S_k$尚未被考虑。因此贪婪算法可以通过增加作业$S_k$来扩充其调度。因此，它会找到一个候选者来扩充它的解决方案，特别是选择一个不晚于S_k的解决方案

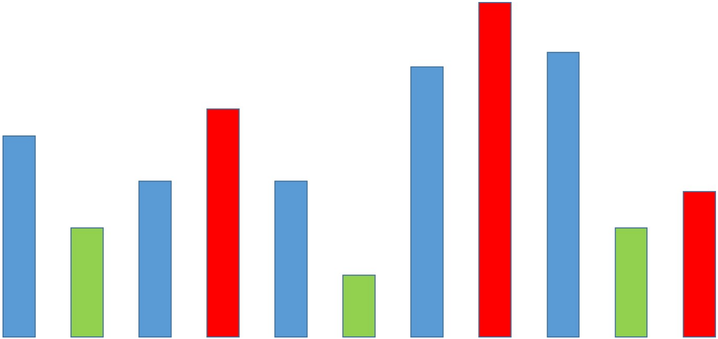
贪婪算法的特点

- 分步骤来构造一个优化问题的解，每一步需满足特定要求
  - 可行：不违反约束条件
  - 局部最优：当前步骤最优的局部解
  - 不可撤销：一旦做出选择，后续步骤中无法改变

最佳股票交易时间

- 整型数组中的元素表示当天股票的价格
- 根据需要买入卖出，完成尽可能多的交易
- 不得同时进行多笔交，必须在再次购买之前卖出股票
- 设计算法以找到最大利润

局部性质

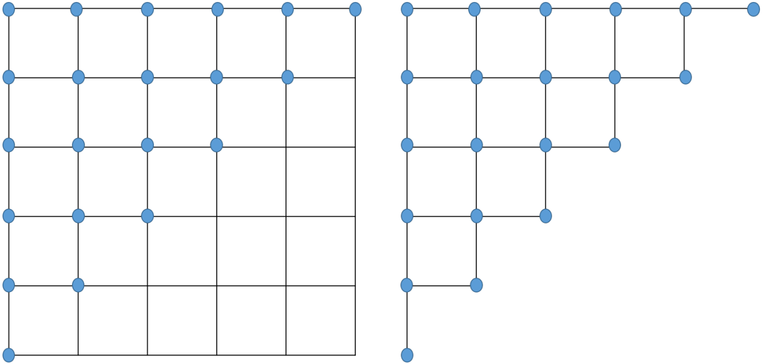


look-ahead: profits accumulation

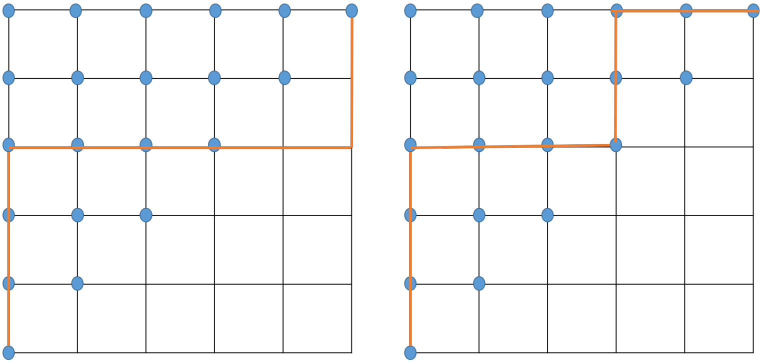
一趟扫描

```c
int profit (int a[], int n)
{
    int tmp = 0;

    for (int i = 1; i < n; i++)
    {
        if (a[i] > a[i-1])
            tmp += a[i] - a[i-1];
    }
    return (tmp);
}
```

Catlan计数问题



非法选择



Catlan数

- 总的路径数C_{2n}^{n}
- 非法路径数C_{2n}^{n-1}
- 合法路径数

$$\mathrm{Ctl}(2n) = \frac{1}{n+1} C_{2n}^{n} = C_{2n}^{n} - C_{2n}^{n-1}$$

动态视角

- 一共需要2n步
- 假设已经完成了k步，这k步一定合法
- 接下来还要走2n - k步，也必须合法
- 分解：先从$(0,0)$走到$(i, n-i), i \le n-i$，再从$(i, n-i)$到(n,n)，初始值$Ct(0,0)=1$

$$Ct(i,j) = Ct(i-1,j) + Ct(i,j-1), \quad \text{if } i < j$$

$$Ct(i,j) = Ct(i-1,j), \quad \text{if } i = j$$
$$Ct(i,j) = Ct(i,j-1), \quad \text{if } i = 0$$

递归代码

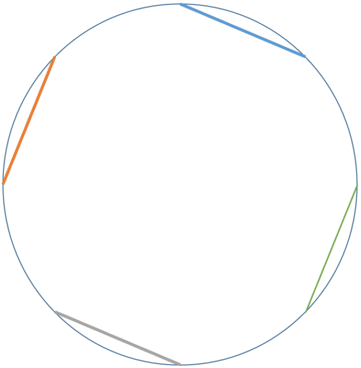
```
int Ctl (int i, int j)
{
    if (1 == i && 1 == j)  //base case
        return (1);

    if (i < j && i > 1 && j > 1)
        return (Ctl (i - 1, j)+ Ctl (i, j - 1));

    if (i == j)  //diagonal
        return (Ctl (i - 1, j));
    if (i == 1)  //border
        return (Ctl (i ,j - 1));

    printf ("Should have not reached here\n");
    return (0);
}
```

Catlan数

0	1	2	3	4	5	6	7	8	9	10
1	1	2	5	14	42	132	429	1430	4862	16796

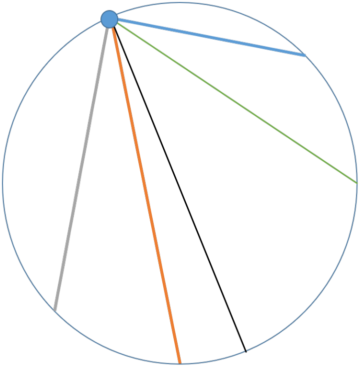
圆与弦

- 圆上有2n个点，可以连成n个不相交的弦
- 不同的连接方式有多少？



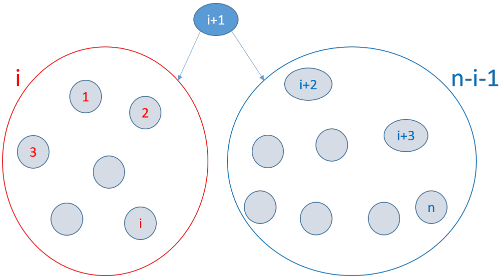
问题分解

- 固定一个点，可以连接n个点，使这条弦的两侧各有偶数个点
- 归结为更小的子问题
  $\mathrm{arcs}(n) = \sum_{i=0}^{n-1} \mathrm{arcs}(i) \times \mathrm{arcs}(n-i-1)$



二叉树数量

- 给定二叉树的中序遍历结果：1, 2, 3, . . . , n
- 一共有多少种不同的二叉树可能产生这种中序遍历？
- 观察视角：根节点在哪里？



递归

```
int count (int n)
{
    int sum = 0;

    if (n == 0)
        return (1);

    for (int i = 0; i < n; i++)
        sum += count (i)* count (n - i - 1);

    return (sum);
}
```

出栈顺序

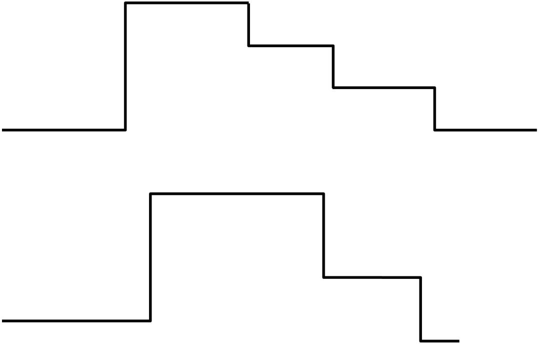
- n个符号分别依次入栈
- 一共有多少种不同可能的出栈方式?
- 归结为已知问题：在$n \times n$的方格中，入栈=向上，出栈=向右，起点$(0, 0)$，终点(n, n)

动态时间拉伸

- 多普勒效应：波在波源移向观察者时接收频率变高，而在波源远离观察者时接收频率变低
- 模板匹配
- 时间序列对准
- 序列相似度度量

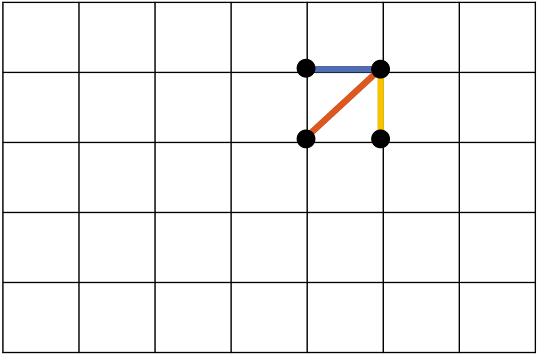
Dynamic Time Warping

- 两个有限长度的序列$X = \{x_1, \ldots, x_m\}, Y = \{y_1, \ldots, y_n\}$
- 非负代价函数$d(x_i, y_j)$
- 如何发现X和Y的最佳对应关系？



动态规划求解

- 转化为从$(0,0)$到(m,n)的最短路径问题
- 已知$D(0,0)=0$，求$D(m,n)$
- 递推过程：$D(i,j)=$
$d(i,j)+\min\{D(i,j-1),D(i-1,j-1),D(i-1,j)$



Dijkstra最短路算法

- 荷兰计算机科学家、软件工程师Edsger Dijkstra在1959年提出的一种解决最短路径问题的贪婪算法
- Dijkstra算法贪婪地探索从s开始的路径，每次移动到下一个最近的节点。这种方式实际上构造了从s到图中每个其他节点的最短路径。
- 给定：带权图以及特殊节点s和t
- 目标：找到s和t之间的最短路径

Dijkstra算法描述

- 初始化$K=\{s\}$，令$Path(s)=\varnothing$，$d(s)=0$
- 对不在K中的每一个结点v，计算距离

$$d(v)=\min_{u\in K}\{d(u)+w(u,v)\}$$

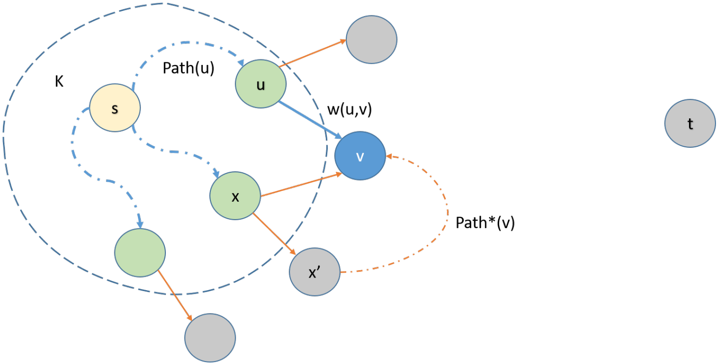
- 令v^*为

$$v^*=\arg\min_{v\notin K} d(v)$$

$e=(u,v)$为对应边

- 令$K=K\cup\{v^*\}$，$Path(v^*)=Path(u)\cup\{e^*\}$，$d(v^*)=d(u)+w(e^*)$.

算法过程



Dijkstra算法的最优性

通过对K的大小的归纳来证明这个结论。$|K| = 1$的基本情况是平凡的：在这种情况下K只包含s和路径$Path(s) = \varnothing$。假设结论一直保持到步骤$k-1$仍成立，并考虑第k步把点v添加到集合K，且$e = (u, v)$为对应边。 为了造成矛盾，假设路径$Path(v)$并非s到v的最短路径，而$Path^*(v)$为相应的最短路径。设x为$Path^*(v)$上集合K中最后一个结点，x'是$Path^*(v)$上紧随x之后的结点。

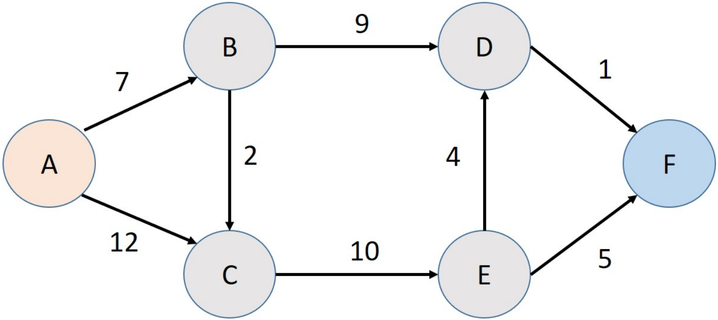
Dijkstra算法的最优性

由定义可知$d(x) + w(x, x')$不超过为$Path^*(v)$的长度。但是，通过构造，

$$d(u) + w(u, v) \leq d(x) + w(x, x')$$

且$d(v) = d(u) + w(u, v)$ 意味着$d(v)$不超过$Path^*(v)$的长度。于是导出矛盾。

最短路径举例



$$s = A, t = F$$

最小生成树问题

- $G=(V,E)$是一个无向连通图，代价函数w将边映射为正实数
- 生成树是一棵连接G的所有顶点的无向树
- 生成树的代价等于树中所有边的代价之和
- 最小生成树是代价为G的所有可能生成树的最小代价的一棵生成树
- 一个图可以有许多具有代价相同的MST
- 构建MST的两个主要算法，Kruskal和Prim算法，都是贪婪算法

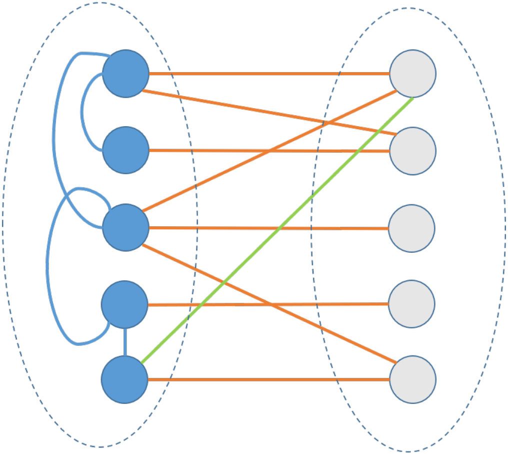
通用算法

```
Input:   Graph G, cost function w
Output:  A minimum spanning tree of G

A ← ∅
while A ≠ MST do
    find a safe edge {u, v} for A
    A ← A ∪ {u, v}
end
return A
```

安全边

- 什么是安全边？如何寻找安全边？
- 图的割(cut of graph)：图$G = (V, E)$的割$(S, V\backslash S)$是对结点集合V的一个划分
- 穿过割的边：一条边$(u, v) \in E$穿过割$(S, V\backslash S)$，如果其端点分别在S和$V\backslash S$
- 轻边：穿过割的所有边中权重最小的一条边
- 结论：假设$A \subset E$，且包含在一些MST中，如果$(S, V\backslash S)$是一个关于A的割，(u, v)是一条穿过割$(S, V\backslash S)$的轻边，则(u, v)对A来说是安全的

安全边



$S, V \setminus S$

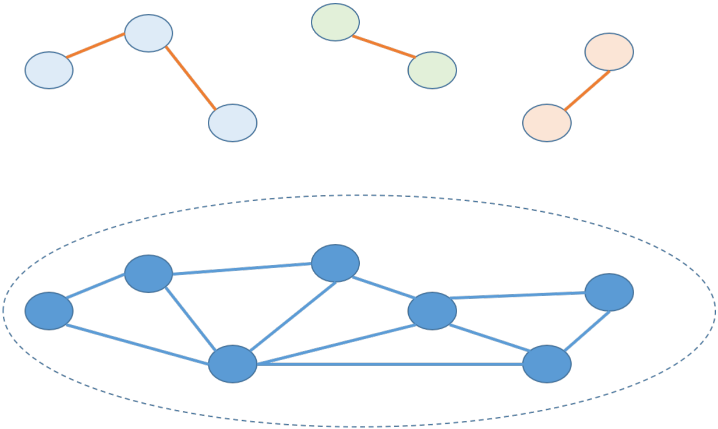
安全边

假设T是一个包含A的MST。如果T包含边(u, v)，则(u, v)对A是安全的。如果T不包含(u, v)，可以证明T'，另一个MST包含$A \cup \{(u, v)\}$，(u, v)对A也是安全的。因为T是生成树，所以在T中有一条从顶点u到顶点v的路径，$Path(u, v)$，如果加入(u, v)则导致了一个循环。因为u和v位于割的不同侧，因此至少有一条边$(x, y) \in Path(u, v)$穿过$(S, T \backslash T)$。此外，$(x, y) \notin A$，其权重不少于边(u, v)的权重。删除T中的边(x, y)并插入边(u, v)，可以构造新树T'，成本最多是和树T的成本一样。此外，$A \cup \{(u, v)\} \subset T'$ 且$(x, y) \notin A$，所以边(u, v)对A是安全的。

Kruskal算法

- Kruskal在1956年所提出的一种MST算法
- 集合A初始时只包括孤立的结点
- 对E的边按权重大小升序排序。
- 顺序考虑边；如果这条边连接两个不同部分，则添加该边
- 无向图$G = (V, E)$的MST可以在时间复杂度$O(|V| \log |V| + |E| \log |E|)$ 完成

Kruskal算法



$$V_a \neq V_b$$

Kruskal算法

```
A ← ∅
foreach v ∈ V do
    MakeSet(v)
end
sort the edges in E in non-decreasing order
foreach e = (a, b) ∈ E taken in sorted order do
    V_a ← Find(a)
    V_b ← Find(b)
    if V_a ≠ V_b then
        A ← A ∪ {e}
        Merge(V_a, V_b)
    end
end
return A
```

Prim算法

- Prim的算法与Dijkstra的单源最短路径算法非常相，具有相同的复杂度
- 在算法的任何阶段，集合A都会形成一棵树，而不是像Kruskal那样由连接的组件组成的森林
- 每个阶段中，都会向树添加一条轻边，将A连接到$V\backslash A$中的顶点
- 用边的权重组成一个优先队列负责查找轻边

P和NP

- 在分析算法的复杂性时，可以把问题转化为一个决策问题：答案为是/否的可计算问题
- 例如，把在一种语言中生成字符串的问题转化为验证给定字符串在一种语言中的成员身份的问题
- P指的是是一类语言，它的隶属度问题可以用输入字符串大小的时间多项式来确定（或者说，多项式时间内可解的决策问题）
- 对于某些语言，则没有多项式时间的成员算法。但是，我们可以用另一个“见证”字符串在多项式时间内验证该字符串是否使用这一语言
- NP是一类语言，给定一个多项式长度的见证字符串，其成员资格可以在多项式时间内进行验证（具有有效验证工具的决策问题）

求解与验证

- 求解问题看起来比验证答案要难（更加耗时）
- 例如，数独，给定初始设置，一般情况下求解数独比验证填好的数字是否满足数独的规则要更为困难
- 但是，如果需要验证的解的数量非常大，即使单个答案可以很快验证，总统上也需要花费很长时间
- NP是nondeterministic polynomial time的缩写，指在非确定性图灵机上可以以多项式时间精确求解的问题
- 有些问题似乎相似，实际分属P和NP：欧拉旅行问题（访问所有边），汉密尔顿环（一次性访问所有节点）

NP和Co-NP

- $L \in NP$ iff $\exists$ a poly-time verifier V such that
  $\forall x \in L\ \exists w$ with $|w| = poly(|x|)$ such that $V(x; w)$ accepts
  $\forall x \notin L\ \exists w$ with $|w| = poly(|x|)$ $V(x; w)$ rejects
- NP问题的YES可以在多项式时间验证，集合Co-NP是NP语言的补集，NO可以在多项式时间验证
  $L \in Co - NP$ iff $\exists$ a poly-time verifier V such that
  $\forall x \notin L\ \exists w$ with $|w| = poly(|x|)$ such that $V(x; w)$ accepts
  $\forall x \in L\ \exists w$ with $|w| = poly(|x|)$ $V(x; w)$ rejects

P–时间可约性

- 可约性的概念允许我们在多项式时间内将一个问题转化为另一个问题
- 如果我们能解决后者，那么我们也能解决前者的问题。相反，如果前者是NP难问题，后者也是NP难问题
- 在讨论决策问题时，如果存在一个多项式时间算法，该算法将问题A的一个实例作为输入，并输出一个保证与问题A的实例具有相同结果的问题B的实例，这种归约称为Cook归约
- 就是说，如果问题B存在一个有效算法，那么问题A可以通过将其实例转换为问题B的实例，并对其应用有效算法来解决

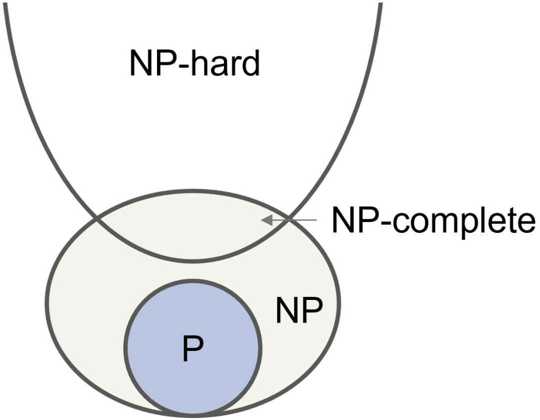
P=NP?

- P与NP是否相等是计算机科学中最突出、最重要的开放性问题之一。考虑这个问题的有效工具：一个问题对于一个类来说是完全的概念
- 可以说，NP中最难的问题称为NP-Hard，是NP中的每一个问题都可以被多多项式时间规约到的问题
- 因此，任何一个问题的多项式时间算法都意味着NP中的每一个问题都可以在多项式时间内求解，即NP $\subseteq$ P
- 我们已经知道P $\subseteq$ NP，因为每个P算法都可以被看作是一个NP算法
- P $=$ NP iff 存在一个决定任何一个NP完全问题的多项式时间算法

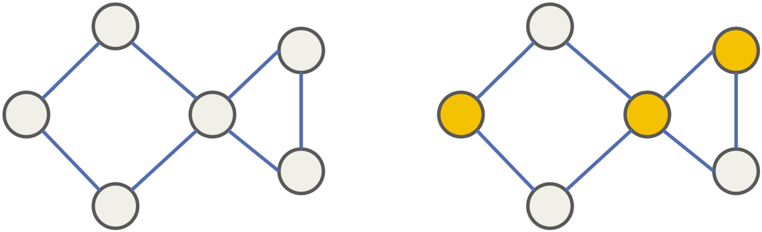
SAT问题

- 第一个被证明是NP完全的问题是Boolean-SAT：给定一个布尔表达式，是否有一组变量可以使整个表达式的值为真？显然，存在一些不可满足的布尔表达式
- 可满足性：给定共轭范式的布尔表达式，寻找变量的值使该表达式为TRUE
- Cook和Levin独立地证明了SAT的NP完全性，称为Cook-Levin定理。Cook-Levin定理证明了SAT是NP完全的，证明了对于NP中的任何问题SAT都存在一个约简

P和NP



顶点覆盖问题



SAT问题定义

布尔可满足性：给定一个CNF（共轭范式）中的布尔公式Φ，该公式可以满足吗？换句话说，我们给出了一个带有n个变量$x_1, x_2, \ldots, x_n$的布尔公式$\Phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$，其中每个$C_i$是一个形式为$(l_{i1} \vee l_{i2} \vee \ldots \vee l_{il})$的子句，每个$l_{ij}$是从集合$\{x_1, x_2, \ldots, x_n; \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n\}$抽取的文字。我们需要决定是否存在一些变量的设置使得Φ得到满足

2-CNF和3-CNF

- 重要性：NP-完全问题是一类任何NP问题可以以多项式时间转化的问题。任一NP完全问题可以多项式时间求解，则所有NP-完全问题都变成了P问题
- 2-CNF：子句包含1或2个变量的满足性决策问题
- 3-CNF：子句可以包含3个变量
- 2-CNF是P问题，而3-CNF属于NPC

SAT的NP完全性质

要证明SAT问题是NP完全的，只需证明NP中的任何问题都可以在多项式时间内规约为SAT问题

- 首先把问题限制为决策问题：把原始问题转化为一个成员问题，即给定的输入是否属于某种语言。那么，P表示可以在多项式时间内确定成员隶属的语言类，而NP代表存在一个可以在多项式时间内验证其成员隶属的语言类。
- 其次，需要将一个问题规约为另一个问题。问题A可以规约到问题B，则如果得到B的一个解，可以多项式时间内调用这个解还原性为A的解。

Cook-Levin定理表明，对于NP中的任何问题，都存在到SAT的一种规约，从而证明SAT是NP完全的。

Cook-Levin定理

Theorem

SAT is NP-complete

假设L是一个NP问题，根据定义，则L具有一个多项式时间的验证算法V：

1. If $x \in L$，$\exists$见证者y，$V(x, y) = 1$
2. If $x \notin L$，$\forall$见证者y，$V(x, y) = 0$

可以为V构造一个多项式大小的电路，由AND、OR、NOT组成。该电路包含$|x| + |y|$个输入，其中$|x|$对应x的每一位的值，$|y|$代表可变变量

Cook-Levin定理证明思路

要求解问题L，只需要找到输入中的$|y|$个变量的一种设置使电路的输出为1。这样就把问题L归结为决定电路是否能输出1的问题。接下来证明满足电路的问题可被归为SAT的一个实例。电路中的每一个门可以表示成一个3-CNF（三元CNF，每个子句仅包含三项）。例如

1. 或门OR是一个输入a和b以及输出Z_i的函数，表示为$(a \vee b \vee \bar{Z}_i) \wedge (Z_i \vee \bar{a}) \wedge (Z_i \vee \bar{b})$
2. 非门NOT是一个输入a且输出Z_i的函数，表示为$(a \vee \bar{Z}i) \wedge (\bar{a} \vee Z_i)$

即使有些子句只包含少于3项，可以通过填充独立文字来构造3-CNF。独立文字的值不影响从句的布尔量

Cook-Levin定理证明

假设V中共有q个门，记为$Z_1, Z_2, ..., Z_q$，其中Z_q是V的最后输出。 这些门要么直接使用输入，要么使用中间结果Z_i为输入。因此，整个电路可以表示为CNF形式的公式： $\Phi = C_1 \wedge C_2 \wedge ... \wedge C_q \wedge Z_q$ 其中$C_i = (t_1 \vee t_2 \vee t_3)$，$t_1, t_2, t_3 \in \{x, y, Z_1, Z_2, ..., Z_q, \bar{Z}_1, \bar{Z}_2, ...\bar{Z}_q\}$。 如前所述，即使最后一个子句只含一项$Z_q$，也可等价转换为3-CNF。因此，该电路被归结为$\Phi$，3-CNF形式的公式。$\Phi$被满足当且仅当原电路输出1。所以，$L \leq_p SAT$。SAT是NP完全问题。

顶点覆盖问题的NP完全性

- Vertex Covering Problem属于NP问题
- 任何一个NP完全问题可以规约为顶点覆盖问题

顶点覆盖问题属于NP

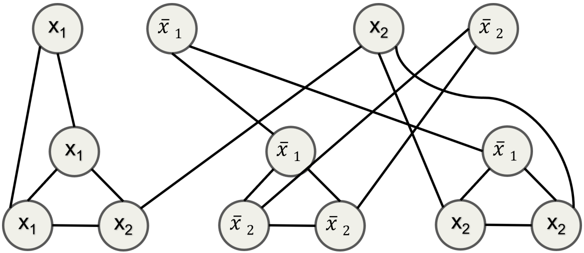
- 设计一个多项式时间的验证算法，检验：
  - 给定的顶点集合是原图顶点的一个子集
  - 该顶点集合覆盖了所有的边
- 上述步骤都可以在多项式时间完成。如果每一步回答都是YES，则返回YES，否则返回NO

SAT规约为顶点覆盖问题

- 给定一个3-CNF：

$$\Phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$$

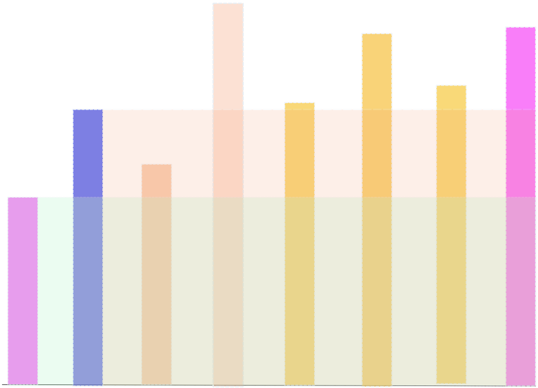
- 设计一个顶点覆盖问题与之等价



接雨水

- 给定一个长度为n的整形数组，元素均大于0
- 以其中任意两个作为隔板，且位置不变
- 如何使容纳水的容积最大?

接雨水



选择短板算容量并淘汰

```
int water (int a[], int n)
{
    int m = 0, tmp;
    int i = 0, j = n - 1;

    while (i < j)
    {
        if (a[i] < a[j])  //select the shorter
            tmp = a[i]*(j - i++);
        else
            tmp = a[j]*(j-- - i);  //for volume
       if (tmp > m)  /*update max*/
            m = tmp;
    }
    return (m);
}
```