期末复习（精简版本复习2）

- 动态规划
  - 带权的区间调度
    - 如果不同区间有不同的权重：
    - 任务 j在 s_j时间开始，f_j时刻结束，且其权重为v_j .
    - 两个任务如果对应的时间区间不相交，称为相容.
    - 目标：寻找最大的不相容的区间子集，使得所选区间的权重之和最大。
    - 情形1: OPT 包含任务需求 j.
      - 不会包含不相容的任务需求$p(j)+1, p(j)+2, ..., j-1$,包含剩下的任务需求$1, 2, ..., p(j)$的最优解
    - 情形2: OPT不包含任务需求 j.
      - 一定包含任务需求$1, 2, ..., j-1$的最优解
    - 数学表示

$$OPT(j)=\begin{cases} 0 & \text{if } j=0 \\ \max\ \{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$

    - 如果已对开始时间排序M-Compute-Opt(n)的运行时间是$O(n)$.
    - 伪代码

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j)
{
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

    - 对于问题的优化
      - 递归的备忘录形式
        - 下面将用到一个数组$M[0...n]$保存中间计算结果

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0                                ← 全局数组

M-Compute-Opt(j) {
   if (M[j] is empty)
   {
    M[j] = max(w_j + M-Compute-Opt(p(j)),
                M-Compute-Opt(j-1))
    }
   return M[j]
}
```

- 按照结束时间排序: $O(nlogn)$.
- 算法本身的时间复杂度: $O(n)$

- 子集和问题
  - 给定n个项$1,...,n$,每个项有一个给定的非负的权w_i,以及给定一个界W.我们想选择项的一个子集S使得$\sum_{i\in S} w_i < W$，并且在这个前提下使得 $\sum_{i\in S}$ 达到最大。这个问题称为子集和问题。
  - 背包问题
    - 用物品$1,...,n$的子集装入一个容量W的背包使得它装的最满（或者装入的价值最大）。
    - 每一个需求i有一个值v_i与一个权w_i,在总权不超过W的限制下，选择一个最大总值的子集。
    - 动态规划方法:

$$OPT(i,w)=\begin{cases} 0 & \text{if } i=0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\quad v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

      - 定义$OPT(i,w)$表示使用物品$1,...,i$的子集且所允许的最大权是w的最优解的值。
      - （考虑到$1,...,i$，选取其中一个子集，最大容量是w）
      - 情形1 OPT 没有选择 i;
        - OPT选择子集$1,2,...,i-1$，采用w的最优解
      - 情形2 OPT 选择 i;
        - 余下部分OPT选择子集$1,2,...,i-1$，采用$w-w_i$的最优解
    - 自底向上的算法

```
Input: n, w_1,…,w_N, v_1,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```

- 上面算法正确的计算这个问题的最优值，并且运行在$\theta(nW)$时间。

- 序列对比
  - 空隙罚分 d;不匹配罚分 α_{pq}.
  - 总的罚分= 空隙罚分和不匹配罚分之和.
  - 定义 $OPT(i,j) = x_1x_2...x_i$ 与 $y_1y_2...y_j$ 比对的最小罚分

$$OPT(i,j)=\begin{cases} j\delta & \text{if } i=0 \\ \min\begin{cases}\alpha_{x_iy_j}+OPT(i-1,j-1) \\ \delta+OPT(i-1,j) \\ \delta+OPT(i,j-1)\end{cases} & \text{otherwise} \\ i\delta & \text{if } j=0\end{cases}$$

    - Case 1: $x_i - y_j$在OPT中
      - $x_i - y_j$不匹配罚分+$x_1x_2...x_{i-1}$和 $y_1y_2...y_{j-1}$最小比对罚分
    - Case 2a: OPT 中x_i 没有匹配
      - x_i处间隙罚分 +$x_1x_2...x_{i-1}$ 和 $y_1y_2...y_{j-1}y_j$最小比对罚分
    - Case 2b: OPT中y_j没有匹配
      - y_j处间隙罚分+ $x_1x_2...x_{i-1}x_i$和$y_1y_2...y_{j-1}$最小比对罚分
  - 时间，空间复杂度？ $\theta(mn)$

```
Sequence-Alignment(m, n, x_1x_2...x_m, y_1y_2...y_n, δ, α) {
   for i = 0 to m
      M[0, i] = iδ
   for j = 0 to n
      M[j, 0] = jδ

   for i = 1 to m
      for j = 1 to n
         M[i, j] = min(α[x_i, y_j] + M[i-1, j-1],
                        δ + M[i-1, j],
                        δ + M[i, j-1])
   return M[m, n]
}
```

- 图中的最短路径
  - 如果存在负圈，s-t中不存在“**最短**”路径
  - Bellman-Ford最短路径算法

- 如果G没有负圈，那么存在一条从s到t的简单的最短路径(没有重复结点)，因此它至多有n-1条边。
- $OPT(i,v)$ 表示至多使用i条边的$v-t$最短路径的最小费用,我们目的是计算s到t的最短路径：$OPT(n-1,s)$
  - 多重选择，$OPT(i,v)$
  - 如果路径P至多用$i-1$条边，那么$OPT(i,v)=OPT(i-1,v)$
  - 如果路径P用i条边,并且第一条边是(v,w),那么余下的是$w-t$路径至多使用$i-1$条边：

    $$OPT(i,v)=c_{vw}+OPT(i-1,w)$$

  - 如果图中没有负圈，那么$OPT(n-1,v)$就是最短的$v-t$路径长度。

    $$OPT(i,v)=\begin{cases} 0 & \text{if } i=0 \\ \min\left\{OPT(i-1,v),\ \min\limits_{(v,w)\in E}\{OPT(i-1,w)+c_{vw}\}\right\} & \text{otherwise} \end{cases}$$

- 网络流
  - 流的定义：
  - $s-t$流是一个函数f,它把每条边e映射到一个非负实数$f:E->R^+$,值$f(e)$表示由边e携带的流量，一个流f必须满足下面两个性质：
    - (容量条件)$0<=f(e)<=C_e$
    - (守恒条件)除了s,t外，对每个结点v,满足

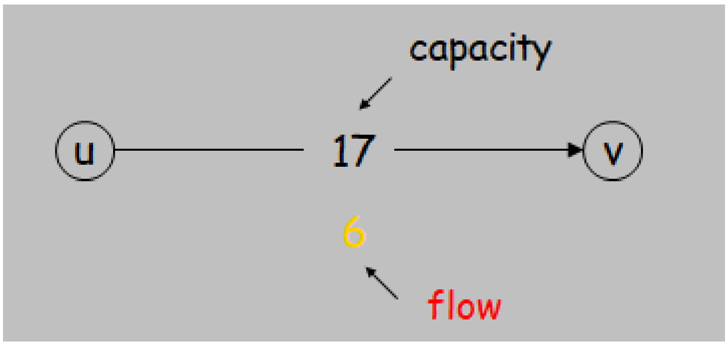
      $$\sum_{e\_in\_v} f(e)=\sum_{e\_out\_v} f(e)$$

  - 最大流问题：给定一个流网络，自然的目标就是安排交通使得有效容量尽可能得到有效使用：找出一个具有最大值的流

    定义 $f^{out}(v)=\sum_{e\_out\_v} f(e), f^{in}(v)=\sum_{e\_in\_v} f(e)$

    我们记 $v(f)=f^{out}(s)$

  - 剩余图
    - 原始边
      - $e=(u,v)\in E$ ，流$f(e)$,容量$c(e)$.



- 剩余边
  - $e=(u,v)$ and $e^R=(v,u)$.
  - 剩余容量:

$$c_f(e)=\begin{cases} c(e)-f(e) & \text{for } e \\ f(e) & \text{for } e^R \end{cases}$$

- 剩余图: $G_f=(V,E_f)$.
  - 具有正的剩余容量的剩余边.
  - $E_f=\{e\}\cup\{e^R\}$.
- 对G的每条边$e=(u,v)$,其中$f(e)<c(e)$,那么存在$c(e)-f(e)$的剩余的容量，我们还可以尝试在这个容量往前推，于是G_f中包含这条边e,容量为$c(e)-f(e)$，称为前向边。
- 对G的每条边$e=(u,v)$,其中$f(e)>0$,我们可以通过向后推这个流来“撤销”它，于是G_f中包含边$e'=(v,u)$,容量是$f(e)$,称为后向边。
- 剩余图中的增广路径
  - 令P是G_f中一条简单的$s-t$路径。定义$bottleneck(P,f)$是P上任何边关于流f的 最小剩余容量。如下算法$augment(f,P)$在G中产生一个新的流f'. $(f->f')$
  - 增广路径 (Augmenting Path)： 在剩余图中，从源节点 s 到汇聚节点 t 的一条简单路径。这条路径上的所有边的剩余容量都大于 0。

```
Augment(f, P) {
   b ← bottleneck(P)
   foreach e ∈ P {
      if (e ∈ E) f(e) ← f(e) + b      前向边
      else       f(eR) ← f(e) - b     后向边
   }
   return f
}
```

  - 下面考虑Ford-Fulkerson算法的运行时间。
  - n表示G中的结点数，m表示G中的边数,那么Ford-Fulkerson算法在至多C次While循环的迭代后终止。
  - 定理7.5 假设在流网络G中的所有容量都是整数，那么Ford-Fulkerson算法可以在$O(mC)$时间内实现

- 网络中的最大流与最小割
    - 我们说一个$s-t$割是结点集合V的一个划分(A,B)，使得 $s\in A,t\in B$.一个割(A,B)的容量记为$c(A,B)$. 也就是从A出来的所有边的容量之和。
    - 令f是任意$s-t$流，且(A,B)是任意$s-t$割，那么$v(f)<=c(A,B)$
- 二分匹配问题
    - 最大流的构造.
        - 构造图 $G'=(L\cup R\cup\{s,t\},E')$.
        - 连接原图L到R的每条边, 每条边赋予单位容量.
        - 增加一个源点s, 从s到L中的每个结点连接一条边，每条边赋予单位容量.
        - 增加一个终点t,从R中的每个结点到t连接一条边，每条边赋予单位容量.
    - 令$n=|L|=|R|$,m是G的边数,时间复杂度:
    - 注意到$C=|L|=n$,根据以前$O(mC)$的界
    - 定理7.38　可以用Ford-Fulkerson算法在$O(mn)$时间内找到二部图中的一个最大匹配。

- NP与计算的难解性
    - 面对一些困难的问题，我们即不知道这些问题存在多项式时间算法，也不能证明问题不存在多项式时间算法，这里将会对“困难”问题提出一个清晰的概念：在计算上实际上是难的，虽然我们不能证明它---NP完全
    - 多项式时间归约
        - $Y\leq_P X$;
            - 读作“Y多项式时间可归约到X”;或“X至少像Y一样的难(相对于多项式时间)”
        - 假设$Y\leq_P X$，如果X能在多项式时间内求解，则Y也能在多项式时间内求解。
        - 假设$Y\leq_P X$，如果Y不能在多项式时间内解决，则X不能在多项式时间内解决。
        - 独立集：在图$G=(V,E)$中，如果顶点集合$S\subseteq V$ 中的任意两点之间没有边，则称S是独立的。
        - 独立集问题：给定图G和数k,问G包含大小至少为k的独立集吗?
        - 顶点覆盖：给定图$G=(V,E)$,如果每一条边$e\in E$至少有一个端点在S中，则称S是一个顶点覆盖。
        - 顶点覆盖问题：给定图G和数k,问G是否包含大小至多为k的顶点覆盖?
    - 集合覆盖问题
        - 给定n个元素的集合U, U的子集$S_1,S_2,...,S_m$以及数k, 是否存在数目至多为k的子集合，其并集等于U
        - 集合覆盖是顶点覆盖的自然推广
        - 顶点覆盖$\leq_P$集合覆盖
    - 集合包装问题
        - 希望把大量集合包装在一起，限制他们中的任意两个都不重叠。
        - 给定n个元素的集合U, U的子集以及数k, 问在这些子集中至少有k个两两不相交吗?

- 独立集$\leq_P$集合包装
- 可满足性问题: (SAT)
  - 给定变量集合$X = \{x_1, x_2...x_n\}$上的一组子句$C_1, ..., C_k$,或者是对于下面这个式子(CNF)，问存在满足的真值赋值吗?

    $$\phi = c_1 \wedge \cdots \wedge c_k$$

  - $3 - SAT$:三元可满足性:
    - 要求每一个子句的长为3
    - $3 - SAT \leq_P$ 独立集.
- 3-SAT $\leq_P$独立集$\leq_P$ 顶点覆盖 $\leq_P$ 集合覆盖.
- NP的定义
  - 如果存在多项式$p(.)$使得对每一个输入串s, 算法A对s的计算在至多$O(p|s|)$步内终止，则称A有多项式运行时间。
  - 根据这一概念，就形成了问题类：存在多项式时间解法的问题的集合P
  - NP(Nondeterministicpolynomial time)是所有存在有效验证程序的问题的集合
- P类问题: 有多项式时间算法，“判定问题”这个行为是否能在多项式时间内求解
  - P类回答：
  - ✓"是否存在解？"
  - ✓"解是否满足某个条件？"
  - P类不一定回答：
  - ? "解是什么？"
  - ? "所有解都是什么？"
  - 只解决存在性问题
- NP类问题: 验证解需要多项式时间
  - NP类关注的核心是"验证的容易性"而不是"求解的容易性"
  - 验证复杂度，而非求解复杂度
  - 给定一个候选解(证书)，能否在多项式时间内验证它是否正确?
- NP完全问题: 目前只有指数时间算法
  - $X \in NP$
  - 对于所有的$Y \in NP$, Y $\leq_P$ X.
- 一般的经验是，NP问题要么是P,要么是NP完全问题