NJU-Open-Resource

Dynamic Programming

Basic Idea

Subproblem Scheduling

动态规划的动机：递归求解问题时可能会有许多子问题被重复计算，合理调度子问题的求解顺序可以使得每个子问题仅被求解一次，之后再求解时只需查询一个 dictionary 获取其解即可。如何合理地调度子问题就是 Dynamic Programming 的关键

Subproblem Graph

> Subproblem Graph
>
> The subproblem graph for a recursive algorithm A of some problem is defined as:
>
> - Vertex: the instance of the problem
> - Directed edge: $I \to J \iff$ when A invoked on I, it makes a recursive call directly on instance J

对于一个递归算法，如果它总能终止，则其 subprogram graph 是个 DAG

Smart recursion：Compute each subproblem **only once**

如何做到 smart recursion：在 subproblem graph 中寻找逆拓扑序，按照逆拓扑序求解子问题，并将其解记录在一个 dictionary 中（DFS 递归实现——尽头节点的概念）

将 subproblem graph 中的调用关系看作依赖关系，按照**逆拓扑序**处理问题可以保证处理任意子问题时其依赖的子问题已被求解，直接查询其解即可

Minimum Cost Matrix Multiplication

task：求一个矩阵序列的乘积：$A_1 \times A_2 \times \cdots \times A_n$ ，由于矩阵乘法的结合律，无论以什么顺序计算，得到的答案都是一样的，但是不同的计算顺序的代价差别很大。如何求出一个代价最小的计算序列

> 两个矩阵相乘的代价：$C_{p\times r} = A_{p\times q} \times B_{q\times r}$

> $$C_{ij} = \sum_{k=1}^{q} A_{ik} B_{kj}$$
>
> 计算一个元素的代价为 q ， C 中有 pr 个元素，总代价为 pqr ，即三个维度值的乘积

Brute Force

若矩阵链中有 n 个矩阵，计算其第一次相乘的代价，则其总代价即为第一次相乘的代价假设递归求解相乘后剩余 $n-1$ 个矩阵相乘的最小代价，遍历所有可能的第一次相乘的位置，求其最小值。显然递归方程为

$$T(n) = (n-1)T(n-1) + n$$
$$T(n) = \Theta((n-1)!)$$

上述递归算法子问题规模降低速度过慢。从另一角度考虑问题，考虑最后一次相乘，其必然是两个矩阵相乘，而这两个矩阵是两个连续的矩阵链的乘积。连续的矩阵链可以用首尾的 index 唯一标识。

用 low 和 high 标识矩阵链，则设最后一次在 k 处相乘，在此之前序列 $(0, k)$ 和 (k, n) 均已完成相乘。由此可递归求出子问题的最优相乘代价，再加上最后一次相乘的代价，即为最后一次在 k 处相乘的代价。遍历所有可能的最后一次相乘的位置，取最优即可。

$$W(n) = 2W(n-1) + n$$
$$W(n) = \Theta(2^n)$$

代价仍然很大，但从上述递归过程中可以看出存在很多次重复计算的子问题，且越短的矩阵序列被重复计算的次数就越多。可以利用 DP 思想消除冗余计算

Dynamic Programming

Smart Recursion: 使用 DFS 遍历 subproblem graph，如果未求解则递归求解，否则使用以求得的解。

对于所有可能的矩阵序列 (i, j) ，其最优相乘代价的解可以存放在一个二维数组中。对于子问题 $(low, high)$ ，其依赖子问题 $(low, k), (k, high)$ 的解，其中 $low < k < high$ ，在二维数组中，可表示为一个元素的求解依赖于其左边的元素(low, k) 和下边的元素 $(k, high)$

故按行从下向上计算，同一行从左向右计算，即可保证求解每个子问题时其依赖的子问题均已求解。使用两重循环遍历即可。

算法共要求解 $O(n^2)$ 个子问题，而每个子问题求解时必须在 $O(n)$ 个可能的最后一次相乘位置对应的子问题中找出最小的一个，故算法时间复杂度为 $O(n^3)$ ，同时还需要 $O(n^2)$ 的空间以存储子问题的解

Weighted Binary Search Tree

为 BST 中每个节点 K_i 赋一个权值 p_i ，设其到根的路径长度为 c_i，定义树的权值

$$A(T)=\sum_{i=1}^{n}p_ic_i$$

Task：给定节点，求 BST 的最小权

平衡的 BST 有时候没有不平衡的 BST 权值小

Subproblem

一颗最优 BST 其左右子树也是最优 BST

对于所有顶点，将其 Key 按升序排列

$$K_1,K_2,\dots,K_n$$

则定义子问题 $(low,high)$ 为使用节点 $K_{low},\dots,K_{high}$ 生成的最优 BST 的权

类比矩阵链乘法，可将这个子问题生成的 BST 分为根 K_r 和由 $K_{low},K_{low+1},\dots K_{r-1}$ 与 $K_{r+1},K_{r+2},\dots K_{high}$ 生成的子树，遍历所有可能的 K_r ，取最优，即可得到子问题 $(low,high)$ 的解。定义：

- $A(low,high,r)$ 为子问题 $(low,high)$ 对应的 BST 的权值，且其根为 K_r
- $A(low,high)=\min\{A(low,high,r)\mid low\leqslant r\leqslant high\}$ ，为子问题 $(low,high)$ 的解
- $p(low,high)=p_{low}+p_{low+1}+\cdots+p_{high}$

对于一颗子问题 $(low,high)$ 生成的 BST T ，设 $A(T)=W$ ，则当其成为某个 root 的子树时，其权值变为 $W+p(low,high)$ （参考树的高度和在成为子树后的变化）

故得到递归关系式

$$\begin{aligned}A(low,high,r)&=p_r+p(low,r-1)+A(low,r-1)+p(r+1,high)+A(r+1,high)\\&=p(low,high)+A(low,r-1)+A(r+1,high)\end{aligned}$$

根据上文对 $A(low,high)$ 的定义可得

$$A(low,high)=\min_{low\leqslant r\leqslant high}\{A(low,r-1)+A(r+1,high)\}+p(low,high)$$

由此可得子问题 $(low,high)$ 的解依赖于 $(low,k-1),(k+1,high)$ 的解，其中 $low\leqslant k\leqslant high$ ，最终所求即为 $(1,n)$

Dynamic Programming

根据上述子问题的定义和依赖关系，显然可以用之前解决矩阵链相乘时的方法解决该问题，维护一个二维数组，按行从下向上计算，同一行从左向右计算，即可保证求解每个子问题时其依赖的子问题均已求解。使用两重循环遍历即可。代价为 $O(n^3)$

APSP & SSSP from the DP perspective

Floyd-Warshall

定义子问题 $dist(u, v, r)$ 为从 u 到 v 的最短路径，且路径上中继顶点不超过 r ，显然

$$\text{dist}(u,v,r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min\{dist(u,v,r-1), \text{dist}(u,r,r-1) + dist(r,v,r-1)\} & \text{otherwise} \end{cases}$$

根据子问题间的依赖关系可以得到一个朴素的 DP 解法：维护一个三维数组，按照 r 的升序来求解子问题。但是可以得到提升的 DP 解法，只用维护二维数组，具体原因可见 Path in Graph 中的 APSP 部分

SSSP over a DAG

定义子问题 $dist(v)$ 为源点到 v 的最短路径（假设源点可达图中所有顶点），则子问题间的依赖关系为

$$dist(v) = \min_{u \to v \in E} \{dist(u) + w(u \to v)\}$$

对于 DAG 可知其存在逆拓扑序，且满足 $u \to v \in E \iff$ 逆拓扑序中 u 在 v 之前，故只要按照逆拓扑序处理图中顶点即可保证计算子问题时其依赖的子问题均已求解完毕。

Edit Distance

两个字符串间的编辑距离定义为：将一个单词变为另一个单词所需的最少“编辑操作”的个数，其中编辑操作包括字母的插入，删除和替换

Task：求两个字符串 $A[1..m], B[1..n]$ 的编辑距离

Subproblem

定义子问题 (i, j) 为 $A[1..i], B[1..j]$ 的编辑距离

将 $A[1..i]$ 编辑为 $B[1..j]$ 有三种情况

- 将 $A[1..i-1]$ 编辑为 $B[1..j]$ 再删除 $A[i]$，即 $(i-1,j)+1$
- 将 $A[1..i]$ 编辑为 $B[1..j-1]$ 再插入 $B[j]$，即 $(i,j-1)+1$
- 将 $A[1..i-1]$ 编辑为 $B[1..j-1]$，再将 $A[i]$ 替换为 $B[j]$，即 $(i-1,j-1)+I\{A[i]\neq B[j]\}$

由此可得子问题间的依赖关系

$$(i,j)=\begin{cases} i & j=0 \\ j & i=0 \\ \min\begin{cases}(i-1,j)+1\\(i,j-1)+1\\(i-1,j-1)+I\{A[i]\neq B[j]\}\end{cases} & \text{otherwise}\end{cases}$$

Dynamic Programming

维护一个二维数组，按照行从上往下，同一行从左往右计算即可。代价为 $O(n^2)$

Highway Restaurants

在高速路上有 n 个餐馆的位置 $m_1,m_2,\ldots,m_n$

在每个位置开餐馆可以获得利润 $p_1,p_2,\ldots,p_n$

任意两餐馆之间必须间隔至少 k 公里

Task：求餐馆的分配使得利润最大

Subproblem

定义子问题 $P(j)$ 为只使用前 j 个位置所能得到的最大利润

定义 $prev[j]$ 为距离 m_j 至少 k 公里的最近的餐馆（$prev[j]=\max\{i\mid m_j-m_i\geqslant k\}$）

则可以考虑两种情况

- 在 m_j 处开餐厅，问题退化为 $p_j+P(prev[j])$
- 不在 m_j 处开餐厅，问题退化为 $P(j-1)$

即

$$P(j)=\max\{P(j-1),p_j+P(prev[j])\}$$

Dynamic Programming

注意到这个问题与之前问题的不同，该问题的子问题空间是一维的。按照 j 升序逐个解决子问题即可。

Words into Lines

有一系列单词长度为 $w_1, w_2, \ldots, w_n$

行宽为 W

需满足若 $w_i, w_{i+1}, dots, w_j$ 在同一行则 $w_i + w_{i+1} + \cdots + w_j \leqslant W$

Task：合理排版使得除最后一行外每行的空格（penalty）总数最少

Subproblem

可定义子问题 (i, j) 表示将单词 i 到单词 j 以最小空格分配在行中。需要特别注意对于任意 (k, n) 最后一行空格记为 0

显然对于某个常数 k，$(1, k), (k+1, n)$ 构成了 $(1, n)$ 的解。遍历所有可能的 k 取最优即可

注意到这个子问题空间是二维的，然而也可使用一维的子问题空间描述这个问题：

子问题 (k, n) 可简化为 (k)，即使用 k 到 n 的单词以最小空格分配在行中。而对于问题 $(1, n)$ 可看作在满足限制的条件下将前 k 个单词放在首行，然后递归解决子问题 $(k+1, n)$，即 (k)

于是便将问题的规模降到了一维

$$(i) = \begin{cases} 0 & \sum_{i}^{n} w_i \leqslant W \\ \min\{W - \sum_{i}^{i+k-1} w_i + (k) \mid W - \sum_{i}^{i+k-1} w_i \geqslant 0\} & \text{otherwise} \end{cases}$$

Dynamic Programming

观察可看出任意子问题依赖于比其大的子问题，按照 i 的降序求解即可

算法的代价为 $O(Wn)$，一般行宽为常数，故代价为 $O(n)$，且空间复杂度为 $O(n)$

Making Change

有不同面额的硬币，其面额为 $d_1, d_2, \ldots, d_n$

Task：给定金额 N，能否用最少的硬币换取金额 N

Subproblem

定义子问题 (i, j) 为使用面额为 $d_1, d_2, \ldots, d_i$ 的硬币换取 j 所需的最少硬币数。显然可将其分为两种情况考虑

- 至少使用了一枚 d_i ，问题降级为 $(i, j-d_i)+1$ 。注：若 $j-d_i<0, (i, j-d_i)=\infty$
- 没有使用 d_i ，问题降级为 $(i-1, j)$

在以上两种情况中取最优即可

$$(i, j)=\begin{cases} 0 & j=0 \\ \infty & j<0 \\ \infty & j>0 \text{ and } i=0 \\ \min\{(i-1, j), (i, j-d_i)+1\} & \text{otherwise} \end{cases}$$

(n, N) 的解即为所求

Dynamic Programming

维护一个二维矩阵存储子问题的解，一个子问题只依赖其左边和上边的子问题的解，按照行从上往下，同一行从左往右计算即可

代价为 $O(nN)$

Other DP Problems

String problems

Longest common subsequence

Longest common substring

Longest increasing subsequence

etc.

One dimensional problems

Arrangements along a straight line

Graph problems

Vertex cover

Hard problems

Knapsack problems

Elements of Dynamic Programming

Symptoms of DP

能使用 DP 解决的问题一般具有以下两个特征

- Overlapping subproblems：这是使用 DP 的基本动机——避免在递归求解中对于子问题的重复计算，可以增加一定的空间开销用于组织子问题和存储子问题的解，但是这些增加的开销会带来显著的性能提升，消除大量的重复计算
- **Optimal substructure**：这是 DP 正确性的关键。即一个问题的最优解必然是由其子问题的最优解组成的，必须原始问题和子问题都求最优解，递归求解才能正常进行，后续的 DP 才能成为可能

How to use DP

Brute Force：DP 基于的递归往往是基于蛮力遍历所有可能取最优，虽然看上去不够高效，但是对于一些比较有难度的优化问题，遍历是无法避免的。正是由于蛮力递归才会带来大量的 overlapping subproblems

Smart Programming：基于子问题间的依赖关系，以 subproblem graph 中拓扑序求解子问题，合理规划调度以避免对子问题的重复计算。