宫水三叶的刷题日记

状压 DP

Author ：宫水三叶
Date ：2021/10/07
QQ Group：703311589
WeChat ：oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「状压 DP」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「状压 DP」即可获取最新下载链接。

💡下面介绍使用本合集的最佳使用实践：

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「状压 DP」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题‘
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **526. 优美的排列** ，难度为 **中等**。

Tag : 「位运算」、「状压 DP」、「动态规划」

假设有从 1 到 N 的 N 个整数，如果从这 N 个数字中成功构造出一个数组，使得数组的第 i 位 $(1 <= i <= N)$ 满足如下两个条件中的一个，我们就称这个数组为一个优美的排列。

条件：

- 第 i 位的数字能被 i 整除
- i 能被第 i 位上的数字整除

现在给定一个整数 N，请问可以构造多少个优美的排列？

示例1:

```
输入: 2

输出: 2

解释:
第 1 个优美的排列是 [1, 2]:
  第 1 个位置（i=1）上的数字是1，1能被 i（i=1）整除
  第 2 个位置（i=2）上的数字是2，2能被 i（i=2）整除

第 2 个优美的排列是 [2, 1]:
  第 1 个位置（i=1）上的数字是2，2能被 i（i=1）整除
  第 2 个位置（i=2）上的数字是1，i（i=2）能被 1 整除
```

说明:

- N 是一个正整数，并且不会超过15。

状态压缩 DP

利用数据范围不超过 15，我们可以使用「状态压缩 DP」进行求解。

使用一个二进制数表示当前哪些数已被选，哪些数未被选，目的是为了可以使用位运算进行加速。

我们可以通过一个具体的样例，来感受下「状态压缩」是什么意思：

例如 $(000...0101)_2$ 代表值为 1 和值为 3 的数字已经被使用了，而值为 2 的节点尚未被使用。

然后再来看看使用「状态压缩」的话，一些基本的操作该如何进行：

假设变量 $state$ 存放了「当前数的使用情况」，当我们需要检查值为 k 的数是否被使用时，可以使用位运算 `a = (state >> k) & 1`，来获取 $state$ 中第 k 位的二进制表示，如果 `a` 为 1 代表值为 k 的数字已被使用，如果为 0 则未被访问。

定义 $f[i][state]$ 为考虑前 i 个数，且当前选择方案为 $state$ 的所有方案数量。

一个显然的初始化条件为 $f[0][0] = 1$，代表当我们不考虑任何数（$i = 0$）的情况下，一个数都不被选择（$state = 0$）为一种合法方案。

不失一般性的考虑 $f[i][state]$ 该如何转移，由于本题是求方案数，我们的转移方程必须做到「不重不漏」。

我们可以通过枚举当前位置 i 是选哪个数，假设位置 i 所选数值为 k，首先 k 值需要同时满足如下两个条件：

- $state$ 中的第 k 位为 1；
- 要么 k 能被 i 整除，要么 i 能被 k 整除。

那么根据状态定义，位置 i 选了数值 k，通过位运算我们可以直接得出决策位置 i 之前的状态是什么：$state \& (\neg(1 << (k-1)))$，代表将 $state$ 的二进制表示中的第 k 位置 0。

最终的 $f[i][state]$ 为当前位置 i 选择的是所有合法的 k 值的方案数之和：

$$f[i][state] = \sum_{k=1}^{n} f[i-1][state \& (\neg(1 << (k-1)))]$$

> 一些细节：由于给定的数值范围为 $[1, n]$，但实现上为了方便，我们使用 $state$ 从右往左的第 0 位表示数值 1 选择情况，第 1 位表示数值 2 的选择情况 ... 即对选择数值 k 做一个 -1 的偏移。

代码：

```java
class Solution {
    public int countArrangement(int n) {
        int mask = 1 << n;
        int[][] f = new int[n + 1][mask];
        f[0][0] = 1;
        for (int i = 1; i <= n; i++) {
            // 枚举所有的状态
            for (int state = 0; state < mask; state++) {
                // 枚举位置 i（最后一位）选的数值是 k
                for (int k = 1; k <= n; k++) {
                    // 首先 k 在 state 中必须是 1
                    if (((state >> (k - 1)) & 1) == 0) continue;
                    // 数值 k 和位置 i 之间满足任一整除关系
                    if (k % i != 0 && i % k != 0) continue;
                    // state & (~(1 << (k - 1))) 代表将 state 中数值 k 的位置置零
                    f[i][state] += f[i - 1][state & (~(1 << (k - 1)))];
                }
            }
        }
        return f[n][mask - 1];
    }
}
```

- 时间复杂度：共有 $n * 2^n$ 的状态需要被转移，每次转移复杂度为 $O(n)$，整体复杂度为 $O(n^2 * 2^n)$
- 空间复杂度：$O(n * 2^n)$

状态压缩 DP（优化）

通过对朴素的状压 DP 的分析，我们发现，在决策第 i 位的时候，理论上我们应该使用的数字数量也应该为 i 个。

但这一点在朴素状压 DP 中并没有体现，这就导致了我们在决策第 i 位的时候，仍然需要对所有的 $state$ 进行计算检查（即使是那些二进制表示中 1 的出现次数不为 i 个的状态）。

因此我们可以换个思路进行枚举（使用新的状态定义并优化转移方程）。

定义 $f[state]$ 为当前选择数值情况为 $state$ 时的所有方案的数量。

这样仍然有 $f[0] = 1$ 的初始化条件，最终答案为 $f[(1 << n) - 1]$。

不失一般性考虑 $f[state]$ 如何计算：

从当前状态 $state$ 进行出发，检查 $state$ 中的每一位 1 作为最后一个被选择的数值，这样仍然可以确保方案数「不重不漏」的被统计，同时由于我们「从小到大」对 $state$ 进行枚举，因此计算 $f[state]$ 所依赖的其他状态值必然都已经被计算完成。

同样的，我们仍然需要确保 $state$ 中的那一位作为最后一个的 1 需要与所放的位置成整除关系。

因此我们需要一个计算 $state$ 的 1 的个数的方法，这里使用 $lowbit$ 实现即可。

最终的 $f[state]$ 为当前位置选择的是所有合法值的方案数之和：

$$f[state] = \sum_{i=0}^{n} f[state \& (\neg(1 << i))]$$

代码：

```
class Solution {
    int getCnt(int x) {
        int ans = 0;
        while (x != 0) {
            x -= (x & -x); // lowbit
            ans++;
        }
        return ans;
    }
    public int countArrangement(int n) {
        int mask = 1 << n;
        int[] f = new int[mask];
        f[0] = 1;
        // 枚举所有的状态
        for (int state = 1; state < mask; state++) {
            // 计算 state 有多少个 1（也就是当前排序长度为多少）
            int cnt = getCnt(state);
            // 枚举最后一位数值为多少
            for (int i = 0; i < n; i++) {
                // 数值在 state 中必须是 1
                if (((state >> i) & 1) == 0) continue;
                // 数值（i + 1）和位置（cnt）之间满足任一整除关系
                if ((i + 1) % cnt != 0 && cnt % (i + 1) != 0) continue;
                // state & (~(1 << i)) 代表将 state 中所选数值的位置置零
                f[state] += f[state & (~(1 << i))];
            }
        }
        return f[mask - 1];
    }
}
```

- 时间复杂度：共有 2^n 的状态需要被转移，每次转移复杂度为 $O(n)$，整体复杂度为 $O(n * 2^n)$
- 空间复杂度：$O(2^n)$

总结

不难发现，其实两种状态压缩 DP 的思路其实是完全一样的。

只不过在朴素状压 DP 中我们是显式的枚举了考虑每一种长度的情况（存在维度 i），而在状压 DP（优化）中利用则 $state$ 中的 1 的个数中蕴含的长度信息。

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **847. 访问所有节点的最短路径** ，难度为 **困难**。

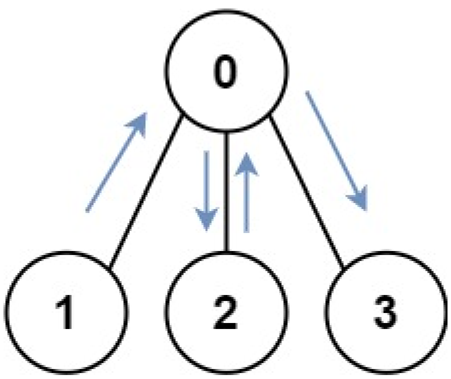
Tag : 「图」、「图论 BFS」、「动态规划」、「状态压缩」

存在一个由 n 个节点组成的无向连通图，图中的节点按从 0 到 n - 1 编号。

给你一个数组 graph 表示这个图。其中，graph[i] 是一个列表，由所有与节点 i 直接相连的节点组成。

返回能够访问所有节点的最短路径的长度。你可以在任一节点开始和停止，也可以多次重访节点，并且可以重用边。

示例 1：

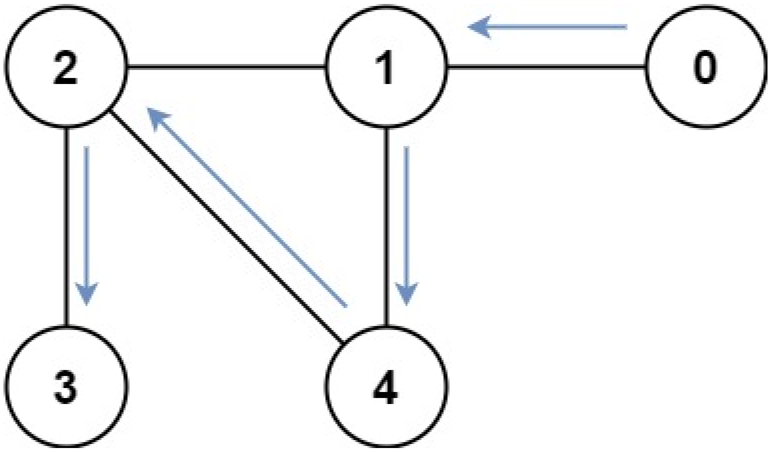



```
输入：graph = [[1,2,3],[0],[0],[0]]

输出：4

解释：一种可能的路径为 [1,0,2,0,3]
```

示例 2：




```
输入：graph = [[1],[0,2,4],[1,3,4],[2],[1,2]]

输出：4

解释：一种可能的路径为 [0,1,4,2,3]
```

提示：

- n == graph.length
- 1 <= n <= 12
- 0 <= graph[i].length < n
- graph[i] 不包含 i
- 如果 graph[a] 包含 b ，那么 graph[b] 也包含 a
- 输入的图总是连通图

基本分析

为了方便，令点的数量为 n，边的数量为 m。

这是一个等权无向图，题目要我们求从「**一个点都没访问过**」到「**所有点都被访问**」的最短路径。

同时 n 只有 12，容易想到使用「状态压缩」来代表「当前点的访问状态」：**使用二进制表示长度为 32 的 `int` 的低 12 来代指点是否被访问过。**

我们可以通过一个具体的样例，来感受下「状态压缩」是什么意思：

例如 $(000...0101)_2$ 代表编号为 0 和编号为 2 的节点已经被访问过，而编号为 1 的节点尚未被访问。

然后再来看看使用「状态压缩」的话，一些基本的操作该如何进行：

假设变量 $state$ 存放了「当前点的访问状态」，当我们需要检查编号为 x 的点是否被访问过时，可以使用位运算 `a = (state >> x) & 1`，来获取 $state$ 中第 x 位的二进制表示，如果 `a` 为 1 代表编号为 x 的节点已被访问，如果为 0 则未被访问。

同理，当我们需要将标记编号为 x 的节点已经被访问的话，可以使用位运算 `state | (1 << x)` 来实现标记。

状态压缩 + BFS

因为是等权图，求从某个状态到另一状态的最短路，容易想到 `BFS`。

同时我们需要知道下一步能往哪些点进行移动，因此除了记录当前的点访问状态 $state$ 以外，还需要记录最后一步是在哪个点 u，因此我们需要使用二元组进行记录 $(state, u)$，同时使用 $dist$ 来记录到达 $(state, u)$ 使用的步长是多少。

> 一些细节：由于点的数量较少，使用「邻接表」或者「邻接矩阵」来存图都可以。对于本题，由于已经给出了 $graph$ 数组，因此可以直接充当「邻接表」来使用，而无须做额外的存图操作。


执行结果： 通过 显示详情 >
添加备注
执行用时： 9 ms ，在所有 Java 提交中击败了 78.31% 的用户
内存消耗： 38.2 MB ，在所有 Java 提交中击败了 68.68% 的用户
炫耀一下：
写题解，分享我的解题思路

代码：

```java
class Solution {
    int INF = 0x3f3f3f3f;
    public int shortestPathLength(int[][] graph) {
        int n = graph.length;
        int mask = 1 << n;

        // 初始化所有的 (state, u) 距离为正无穷
        int[][] dist = new int[mask][n];
        for (int i = 0; i < mask; i++) Arrays.fill(dist[i], INF);

        // 因为可以从任意起点出发，先将起始的起点状态入队，并设起点距离为 0
        Deque<int[]> d = new ArrayDeque<>(); // state, u
        for (int i = 0; i < n; i++) {
            dist[1 << i][i] = 0;
            d.addLast(new int[]{1 << i, i});
        }

        // BFS 过程，如果从点 u 能够到达点 i，则更新距离并进行入队
        while (!d.isEmpty()) {
            int[] poll = d.pollFirst();
            int state = poll[0], u = poll[1], step = dist[state][u];
            if (state == mask - 1) return step;
            for (int i : graph[u]) {
                if (dist[state | (1 << i)][i] == INF) {
                    dist[state | (1 << i)][i] = step + 1;
                    d.addLast(new int[]{state | (1 << i), i});
                }
            }
        }
        return -1; // never
    }
}
```

- 时间复杂度：点（状态）数量为 $n * 2^n$，边的数量为 $n^2 * 2^n$，`BFS` 复杂度上界为点数加边数，整体复杂度为 $O(n^2 * 2^n)$
- 空间复杂度：$O(n * 2^n)$

Floyd + 状压 DP

其实在上述方法中，我们已经使用了与 DP 状态定义分析很像的思路了。甚至我们的元祖设计 $(state, u)$ 也很像状态定义的两个维度。

那么为什么我们不使用 $f[state][u]$ 为从「没有点被访问过」到「访问过的点状态为 $state$」，并最后一步落在点 u 的状态定义，然后跑一遍 DP 来做呢？

是因为如果从「常规的 DP 转移思路」出发，状态之间不存在拓扑序（有环），这就导致了我们在计算某个 $f[state][u]$ 时，它所依赖的状态并不确保已经被计算/更新完成，所以我们无法使用常规的 DP 手段来求解。

> 这里说的常规 DP 手段是指：枚举所有与 u 相连的节点 v，用 $f[state'][v]$ 来更新 $f[state][u]$ 的转移方式。

常规的 DP 转移方式状态间不存在拓扑序，我们需要换一个思路进行转移。

对于某个 $state$ 而言，我们可以枚举其最后一个点 i 是哪一个，充当其达到 $state$ 的最后一步，然后再枚举下一个点 j 是哪一个，充当移动的下一步（当然前提是满足 $state$ 的第 i 位为 1，而第 j 位为 0）。

求解任意两点最短路径，可以使用 Floyd 算法，复杂度为 $O(n^3)$。

代码：

```java
class Solution {
    int INF = 0x3f3f3f3f;
    public int shortestPathLength(int[][] graph) {
        int n = graph.length;
        int mask = 1 << n;

        // Floyd 求两点的最短路径
        int[][] dist = new int[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                dist[i][j] = INF;
            }
        }
        for (int i = 0; i < n; i++) {
            for (int j : graph[i]) dist[i][j] = 1;
        }
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }

        // DP 过程，如果从 i 能够到 j 的话，使用 i 到 j 的最短距离（步长）来转移
        int[][] f = new int[mask][n];
        // 起始时，让所有状态的最短距离（步长）为正无穷
        for (int i = 0; i < mask; i++) Arrays.fill(f[i], INF);
        // 由于可以将任意点作为起点出发，可以将这些起点的最短距离（步长）设置为 0
        for (int i = 0; i < n; i++) f[1 << i][i] = 0;

        // 枚举所有的 state
        for (int state = 0; state < mask; state++) {
            // 枚举 state 中已经被访问过的点
            for (int i = 0; i < n; i++) {
                if (((state >> i) & 1) == 0) continue;
                // 枚举 state 中尚未被访问过的点
                for (int j = 0; j < n; j++) {
                    if (((state >> j) & 1) == 1) continue;
                    f[state | (1 << j)][j] = Math.min(f[state | (1 << j)][j], f[state][i]
                }
            }
        }

        int ans = INF;
        for (int i = 0; i < n; i++) ans = Math.min(ans, f[mask - 1][i]);
```

```
        return ans;
    }
}
```

- 时间复杂度：Floyd 复杂度为 $O(n^3)$；DP 共有 $n * 2^n$ 个状态需要被转移，每次转移复杂度为 $O(n)$，总的复杂度为 $O(n^2 * 2^n)$。整体复杂度为 $O(\max(n^3, n^2 * 2^n))$
- 空间复杂度：$O(n * 2^n)$

AStar

显然，从 $state$ 到 $state'$ 的「理论最小修改成本」为两者二进制表示中不同位数的个数。

同时，当且仅当在 $state$ 中 1 的位置与 $state'$ 中 0 存在边，才有可能取到这个「理论最小修改成本」。

因此直接使用当前状态 $state$ 与最终目标状态 `1 << n` 两者二进制表示中不同位数的个数作为启发预估值是合适的。

代码：

```java
class Solution {
    int INF = 0x3f3f3f3f;
    int n;
    int f(int state) {
        int ans = 0;
        for (int i = 0; i < n; i++) {
            if (((state >> i) & 1) == 0) ans++;
        }
        return ans;
    }
    public int shortestPathLength(int[][] g) {
        n = g.length;
        int mask = 1 << n;
        int[][] dist = new int[mask][n];
        for (int i = 0; i < mask; i++) {
            for (int j = 0; j < n; j++) {
                dist[i][j] = INF;
            }
        }
        PriorityQueue<int[]> q = new PriorityQueue<>((a,b)->a[2]-b[2]); // state, u, val
        for (int i = 0; i < n; i++) {
            dist[1 << i][i] = 0;
            q.add(new int[]{1<< i, i, f(i << 1)});
        }
        while (!q.isEmpty()) {
            int[] poll = q.poll();
            int state = poll[0], u = poll[1], step = dist[state][u];
            if (state == mask - 1) return step;
            for (int i : g[u]) {
                int nState = state | (1 << i);
                if (dist[nState][i] > step + 1) {
                    dist[nState][i] = step + 1;
                    q.add(new int[]{nState, i, step + 1 + f(nState)});
                }
            }
        }
        return -1; // never
    }
}
```

最新专题合集资料下载，可关注公众号「宫水三叶的刷题日记」，回台回复「状压 DP」获取下载链接。

觉得专题不错，可以请作者吃糖🍭🍭🍭：