宫水三叶的刷题日记

图论 BFS

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「图论 BFS」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「图论 BFS」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「图论 BFS」；
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 **127. 单词接龙** ，难度为 **困难**。

Tag : 「双向 BFS」

字典 wordList 中从单词 beginWord 和 endWord 的 转换序列 是一个按下述规格形成的序列：

- 序列中第一个单词是 beginWord 。
- 序列中最后一个单词是 endWord 。
- 每次转换只能改变一个字母。
- 转换过程中的中间单词必须是字典 wordList 中的单词。

给你两个单词 beginWord 和 endWord 和一个字典 wordList ，找到从 beginWord 到 endWord 的 最短转换序列 中的 单词数目 。如果不存在这样的转换序列，返回 0。

示例 1：

```
输入：beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
输出：5
解释：一个最短转换序列是 "hit" -> "hot" -> "dot" -> "dog" -> "cog", 返回它的长度 5。
```

示例 2：

```
输入：beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
输出：0
解释：endWord "cog" 不在字典中，所以无法进行转换。
```

提示：

- 1 <= beginWord.length <= 10
- endWord.length == beginWord.length
- 1 <= wordList.length <= 5000
- wordList[i].length == beginWord.length
- beginWord、endWord 和 wordList[i] 由小写英文字母组成
- beginWord != endWord
- wordList 中的所有字符串 互不相同

基本分析

根据题意，每次只能替换一个字符，且每次产生的新单词必须在 `wordList` 出现过。

一个朴素的实现方法是，使用 BFS 的方式求解。

从 `beginWord` 出发，枚举所有替换一个字符的方案，如果方案存在于 `wordList` 中，则加入队列中，这样队列中就存在所有替换次数为 1 的单词。然后从队列中取出元素，继续这个过程，直到遇到 `endWord` 或者队列为空为止。

同时为了「防止重复枚举到某个中间结果」和「记录每个中间结果是经过多少次转换而来」，我们需要建立一个「哈希表」进行记录。

哈希表的 KV 形式为 `{ 单词 : 由多少次转换得到 }`。

当枚举到新单词 `str` 时，需要先检查是否已经存在与「哈希表」中，如果不存在则更新「哈希表」并将新单词放入队列中。

这样的做法可以确保「枚举到所有由 `beginWord` 到 `endWord` 的转换路径」，并且由 `beginWord` 到 `endWord` 的「最短转换路径」必然会最先被枚举到。

双向 BFS

经过分析，BFS 确实可以做，但本题的数据范围较大：`1 <= beginWord.length <= 10`

朴素的 BFS 可能会带来「搜索空间爆炸」的情况。

想象一下，如果我们的 `wordList` 足够丰富（包含了所有单词），对于一个长度为 10 的 `beginWord` 替换一次字符可以产生 $10 * 25$ 个新单词（每个替换点可以替换另外 25 个小写字母），第一层就会产生 250 个单词；第二层会产生超过 $6 * 10^4$ 个新单词 ...

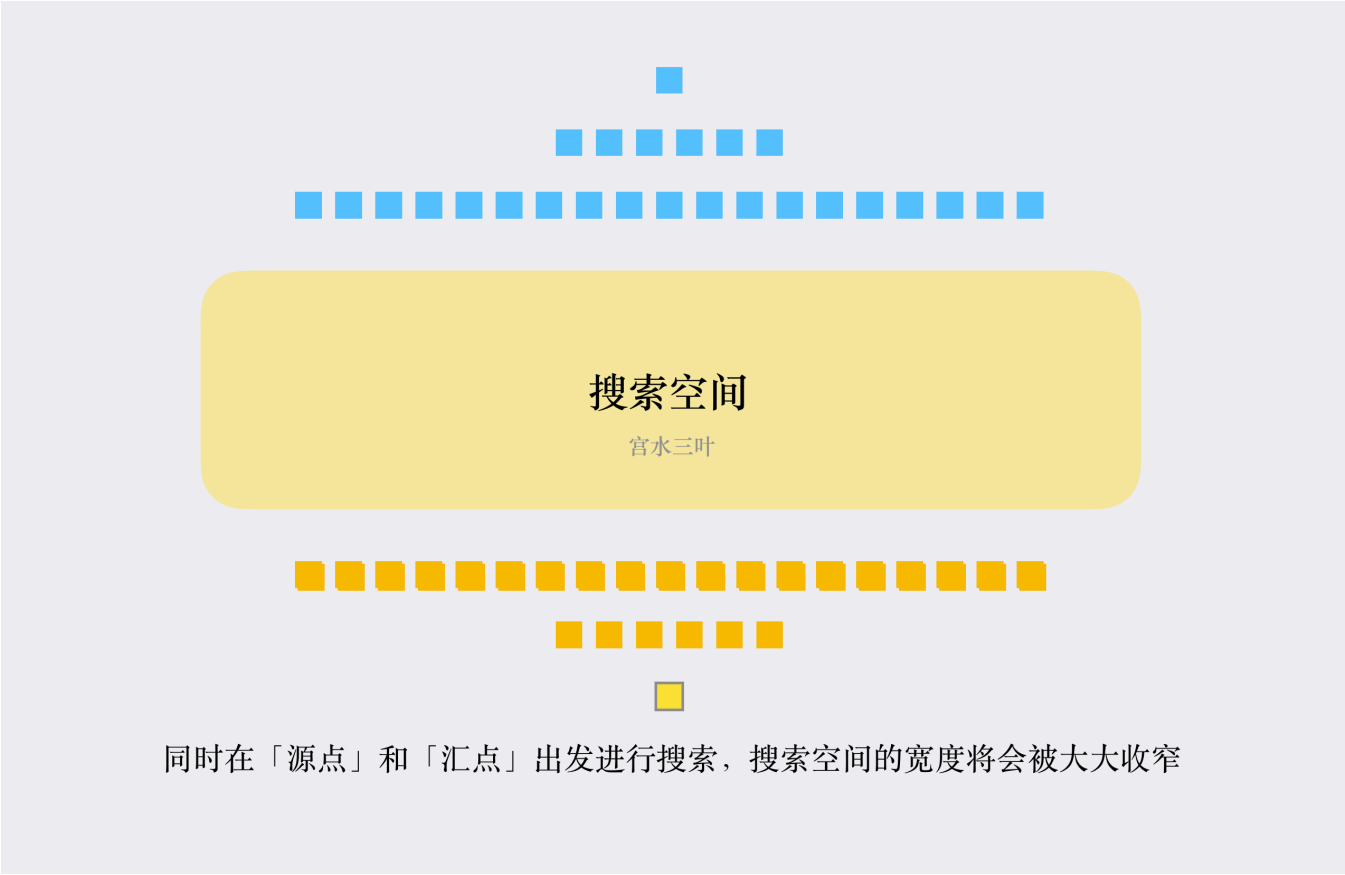
随着层数的加深，这个数字的增速越快，这就是「搜索空间爆炸」问题。

在朴素的 BFS 实现中，空间的瓶颈主要取决于搜索空间中的最大宽度。

那么有没有办法让我们不使用这么宽的搜索空间，同时又能保证搜索到目标结果呢？

「双向 BFS」可以很好的解决这个问题：

同时从两个方向开始搜索，一旦搜索到相同的值，意味着找到了一条联通起点和终点的最短路径。



同时在「源点」和「汇点」出发进行搜索，搜索空间的宽度将会被大大收窄

「双向 BFS」的基本实现思路如下：

1. 创建「两个队列」分别用于两个方向的搜索；
2. 创建「两个哈希表」用于「解决相同节点重复搜索」和「记录转换次数」；
3. 为了尽可能让两个搜索方向“平均”，每次从队列中取值进行扩展时，先判断哪个队列容量较少；
4. 如果在搜索过程中「搜索到对方搜索过的节点」，说明找到了最短路径。

「双向 BFS」基本思路对应的伪代码大致如下：

```
d1、d2 为两个方向的队列
m1、m2 为两个方向的哈希表，记录每个节点距离起点的

// 只有两个队列都不空，才有必要继续往下搜索
// 如果其中一个队列空了，说明从某个方向搜到底都搜不到该方向的目标节点
while(!d1.isEmpty() && !d2.isEmpty()) {
    if (d1.size() < d2.size()) {
        update(d1, m1, m2);
    } else {
        update(d2, m2, m1);
    }
}

// update 为从队列 d 中取出一个元素进行「一次完整扩展」的逻辑
void update(Deque d, Map cur, Map other) {}
```

回到本题，我们看看如何使用「双向 BFS」进行求解。

估计不少同学是第一次接触「双向 BFS」，因此这次我写了大量注释。

建议大家带着对「双向 BFS」的基本理解去阅读。

代码：

```java
class Solution {
    String s, e;
    Set<String> set = new HashSet<>();
    public int ladderLength(String _s, String _e, List<String> ws) {
        s = _s;
        e = _e;
        // 将所有 word 存入 set，如果目标单词不在 set 中，说明无解
        for (String w : ws) set.add(w);
        if (!set.contains(e)) return 0;
        int ans = bfs();
        return ans == -1 ? 0 : ans + 1;
    }

    int bfs() {
        // d1 代表从起点 beginWord 开始搜索（正向）
        // d2 代表从结尾 endWord 开始搜索（反向）
        Deque<String> d1 = new ArrayDeque<>(), d2 = new ArrayDeque();

        /*
         * m1 和 m2 分别记录两个方向出现的单词是经过多少次转换而来
         * e.g.
         * m1 = {"abc":1} 代表 abc 由 beginWord 替换 1 次字符而来
         * m2 = {"xyz":3} 代表 xyz 由 endWord 替换 3 次字符而来
         */
        Map<String, Integer> m1 = new HashMap<>(), m2 = new HashMap<>();
        d1.add(s);
        m1.put(s, 0);
        d2.add(e);
        m2.put(e, 0);

        /*
         * 只有两个队列都不空，才有必要继续往下搜索
         * 如果其中一个队列空了，说明从某个方向搜到底都搜不到该方向的目标节点
         * e.g.
         * 例如，如果 d1 为空了，说明从 beginWord 搜索到底都搜索不到 endWord，反向搜索也没必要进行了
         */
        while (!d1.isEmpty() && !d2.isEmpty()) {
            int t = -1;
            // 为了让两个方向的搜索尽可能平均，优先拓展队列内元素少的方向
            if (d1.size() <= d2.size()) {
                t = update(d1, m1, m2);
            } else {
                t = update(d2, m2, m1);
            }
            if (t != -1) return t;
        }
```

```
        return -1;
    }

    // update 代表从 deque 中取出一个单词进行扩展，
    // cur 为当前方向的距离字典；other 为另外一个方向的距离字典
    int update(Deque<String> deque, Map<String, Integer> cur, Map<String, Integer> other)
        // 获取当前需要扩展的原字符串
        String poll = deque.pollFirst();
        int n = poll.length();

        // 枚举替换原字符串的哪个字符 i
        for (int i = 0; i < n; i++) {
            // 枚举将 i 替换成哪个小写字母
            for (int j = 0; j < 26; j++) {
                // 替换后的字符串
                String sub = poll.substring(0, i) + String.valueOf((char)('a' + j)) + poll
                if (set.contains(sub)) {
                    // 如果该字符串在「当前方向」被记录过（拓展过），跳过即可
                    if (cur.containsKey(sub)) continue;

                    // 如果该字符串在「另一方向」出现过，说明找到了联通两个方向的最短路
                    if (other.containsKey(sub)) {
                        return cur.get(poll) + 1 + other.get(sub);
                    } else {
                        // 否则加入 deque 队列
                        deque.addLast(sub);
                        cur.put(sub, cur.get(poll) + 1);
                    }
                }
            }
        }
        return -1;
    }
}
```

- 时间复杂度：令 `wordList` 长度为 n，`beginWord` 字符串长度为 m。由于所有的搜索结果必须都在 `wordList` 出现过，因此算上起点最多有 $n+1$ 节点，最坏情况下，所有节点都联通，搜索完整张图复杂度为 $O(n^2)$；从 `beginWord` 出发进行字符替换，替换时进行逐字符检查，复杂度为 $O(m)$。整体复杂度为 $O(m*n^2)$
- 空间复杂度：同等空间大小。$O(m*n^2)$

总结

这本质其实是一个「所有边权均为 `1` 」最短路问题：将 `beginWord` 和所有在 `wordList` 出现过的字符串看做是一个点。每一次转换操作看作产生边权为 `1` 的边。问题求以 `beginWord` 为源点，以 `endWord` 为汇点的最短路径。

借助这个题，我向你介绍了「双向 BFS」，「双向 BFS」可以有效解决「搜索空间爆炸」问题。

对于那些搜索节点随着层数增加呈倍数或指数增长的搜索问题，可以使用「双向 BFS」进行求解。

【补充】启发式搜索 AStar

可以直接根据本题规则来设计 A* 的「启发式函数」。

比如对于两个字符串 `a` `b` 直接使用它们不同字符的数量来充当估值距离，我觉得是合适的。

因为不同字符数量的差值可以确保不会超过真实距离（是一个理论最小替换次数）。

注意：本题数据比较弱，用 A* 过了，但通常我们需要「确保有解」，A* 的启发搜索才会发挥真正价值。而本题，除非 `endWord` 本身就不在 `wordList` 中，其余情况我们无法很好提前判断「是否有解」。这时候 A* 将不能带来「搜索空间的优化」，效果不如「双向 BFS」。

代码：

```java
class Solution {
    class Node {
        String str;
        int val;
        Node (String _str, int _val) {
            str = _str;
            val = _val;
        }
    }
    String s, e;
    int INF = 0x3f3f3f3f;
    Set<String> set = new HashSet<>();
    public int ladderLength(String _s, String _e, List<String> ws) {
        s = _s;
        e = _e;
        for (String w : ws) set.add(w);
        if (!set.contains(e)) return 0;
        int ans = astar();
        return ans == -1 ? 0 : ans + 1;
    }
    int astar() {
        PriorityQueue<Node> q = new PriorityQueue<>((a,b)->a.val-b.val);
        Map<String, Integer> dist = new HashMap<>();
        dist.put(s, 0);
        q.add(new Node(s, f(s)));

        while (!q.isEmpty()) {
            Node poll = q.poll();
            String str = poll.str;
            int distance = dist.get(str);
            if (str.equals(e)) {
                break;
            }
            int n = str.length();
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < 26; j++) {
                    String sub = str.substring(0, i) + String.valueOf((char)('a' + j)) + s
                    if (!set.contains(sub)) continue;
                    if (!dist.containsKey(sub) || dist.get(sub) > distance + 1) {
                        dist.put(sub, distance + 1);
                        q.add(new Node(sub, dist.get(sub) + f(sub)));
                    }
                }
            }
        }
        return dist.containsKey(e) ? dist.get(e) : -1;
```

```
    }
    int f(String str) {
        if (str.length() != e.length()) return INF;
        int n = str.length();
        int ans = 0;
        for (int i = 0; i < n; i++) {
            ans += str.charAt(i) == e.charAt(i) ? 0 : 1;
        }
        return ans;
    }
}
```

> **🍭 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 **403. 青蛙过河** ，难度为 **困难**。

Tag：「DFS」、「BFS」、「记忆化搜索」、「线性 DP」

一只青蛙想要过河。 假定河流被等分为若干个单元格，并且在每一个单元格内都有可能放有一块石子（也有可能没有）。 青蛙可以跳上石子，但是不可以跳入水中。

给你石子的位置列表 stones（用单元格序号 升序 表示）， 请判定青蛙能否成功过河（即能否在最后一步跳至最后一块石子上）。

开始时， 青蛙默认已站在第一块石子上，并可以假定它第一步只能跳跃一个单位（即只能从单元格 1 跳至单元格 2 ）。

如果青蛙上一步跳跃了 k 个单位，那么它接下来的跳跃距离只能选择为 k - 1、k 或 k + 1 个单位。 另请注意，青蛙只能向前方（终点的方向）跳跃。

示例 1：

```
输入：stones = [0,1,3,5,6,8,12,17]

输出：true

解释：青蛙可以成功过河，按照如下方案跳跃：跳 1 个单位到第 2 块石子, 然后跳 2 个单位到第 3 块石子, 接着 跳 2 个单位到第 4 块
```

示例 2：

```
输入：stones = [0,1,2,3,4,8,9,11]

输出：false

解释：这是因为第 5 和第 6 个石子之间的间距太大，没有可选的方案供青蛙跳跃过去。
```

提示：

- 2 <= stones.length <= 2000
- 0 <= stones[i] <= 2^{31} - 1
- stones[0] == 0

DFS（TLE）

根据题意，我们可以使用 `DFS` 来模拟/爆搜一遍，检查所有的可能性中是否有能到达最后一块石子的。

通常设计 `DFS` 函数时，我们只需要不失一般性的考虑完成第 i 块石子的跳跃需要些什么信息即可：

- 需要知道当前所在位置在哪，也就是需要知道当前石子所在列表中的下标 u。
- 需要知道当前所在位置是经过多少步而来的，也就是需要知道上一步的跳跃步长 k。

代码：

```Java
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    public boolean canCross(int[] ss) {
        int n = ss.length;
        // 将石子信息存入哈希表
        // 为了快速判断是否存在某块石子，以及快速查找某块石子所在下标
        for (int i = 0; i < n; i++) {
            map.put(ss[i], i);
        }
        // check first step
        // 根据题意，第一步是固定经过步长 1 到达第一块石子（下标为 1）
        if (!map.containsKey(1)) return false;
        return dfs(ss, ss.length, 1, 1);
    }

    /**
     * 判定是否能够跳到最后一块石子
     * @param ss 石子列表【不变】
     * @param n  石子列表长度【不变】
     * @param u  当前所在的石子的下标
     * @param k  上一次是经过多少步跳到当前位置的
     * @return 是否能跳到最后一块石子
     */
    boolean dfs(int[] ss, int n, int u, int k) {
        if (u == n - 1) return true;
        for (int i = -1; i <= 1; i++) {
            // 如果是原地踏步的话，直接跳过
            if (k + i == 0) continue;
            // 下一步的石子理论编号
            int next = ss[u] + k + i;
            // 如果存在下一步的石子，则跳转到下一步石子，并 DFS 下去
            if (map.containsKey(next)) {
                boolean cur = dfs(ss, n, map.get(next), k + i);
                if (cur) return true;
            }
        }
        return false;
    }
}
```

- 时间复杂度：$O(3^n)$
- 空间复杂度：$O(3^n)$

但数据范围为 10^3，直接使用 DFS 肯定会超时。

我们需要考虑加入「记忆化」功能，或者改为使用带标记的 `BFS`。

记忆化搜索

在考虑加入「记忆化」时，我们只需要将 `DFS` 方法签名中的【可变】参数作为维度，`DFS` 方法中的返回值作为存储值即可。

通常我们会使用「数组」来作为我们缓存中间结果的容器，

对应到本题，就是需要一个 `boolean[石子列表下标][跳跃步数]` 这样的数组，但使用布尔数组作为记忆化容器往往无法区分「状态尚未计算」和「状态已经计算，并且结果为 `false`」两种情况。

因此我们需要转为使用 `int[石子列表下标][跳跃步数]`，默认值 0 代表状态尚未计算，-1 代表计算状态为 `false`，1 代表计算状态为 `true`。

接下来需要估算数组的容量，可以从「数据范围」入手分析。

根据 `2 <= stones.length <= 2000`，我们可以确定第一维（数组下标）的长度为 2009，而另外一维（跳跃步数）是与跳转过程相关的，无法直接确定一个精确边界，但是一个显而易见的事实是，跳到最后一块石子之后的位置是没有意义的，因此我们不会有「跳跃步长」大于「石子列表长度」的情况，因此也可以定为 2009（这里是利用了由下标为 i 的位置发起的跳跃不会超过 $i + 1$ 的性质）。

至此，我们定下来了记忆化容器为 `int[][] cache = new int[2009][2009]`。

但是可以看出，上述确定容器大小的过程还是需要一点点分析 & 经验的。

那么是否有思维难度再低点的方法呢？

答案是有的，直接使用「哈希表」作为记忆化容器。「哈希表」本身属于非定长容器集合，我们不需要分析两个维度的上限到底是多少。

另外，当容器维度较多且上界较大时（例如上述的 `int[2009][2009]`），直接使用「哈希表」可以有效降低「爆空间/时间」的风险（不需要每跑一个样例都创建一个百万级的数组）。

代码：

```java
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    // int[][] cache = new int[2009][2009];
    Map<String, Boolean> cache = new HashMap<>();
    public boolean canCross(int[] ss) {
        int n = ss.length;
        for (int i = 0; i < n; i++) {
            map.put(ss[i], i);
        }
        // check first step
        if (!map.containsKey(1)) return false;
        return dfs(ss, ss.length, 1, 1);
    }
    boolean dfs(int[] ss, int n, int u, int k) {
        String key = u + "_" + k;
        // if (cache[u][k] != 0) return cache[u][k] == 1;
        if (cache.containsKey(key)) return cache.get(key);
        if (u == n - 1) return true;
        for (int i = -1; i <= 1; i++) {
            if (k + i == 0) continue;
            int next = ss[u] + k + i;
            if (map.containsKey(next)) {
                boolean cur = dfs(ss, n, map.get(next), k + i);
                // cache[u][k] = cur ? 1 : -1;
                cache.put(key, cur);
                if (cur) return true;
            }
        }
        // cache[u][k] = -1;
        cache.put(key, false);
        return false;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

动态规划

有了「记忆化搜索」的基础，要写写出来动态规划就变得相对简单了。

我们可以从 `DFS` 函数出发，写出「动态规划」解法。

我们的 `DFS` 函数签名为：

```
boolean dfs(int[] ss, int n, int u, int k);
```

其中前两个参数为不变参数，后两个为可变参数，返回值是我们的答案。

因此可以设定为 $f[][]$ 作为动规数组：

1. 第一维为可变参数 u，代表石子列表的下标，范围为数组 `stones` 长度；
2. 第二维为可变参数 k，代表上一步的的跳跃步长，前面也分析过了，最多不超过数组 `stones` 长度。

这样的「状态定义」所代表的含义：当前在第 i 个位置，并且是以步长 k 跳到位置 i 时，是否到达最后一块石子。

那么对于 $f[i][k]$ 是否为真，则取决于上一位置 j 的状态值，结合每次步长的变化为 `[-1,0,1]` 可知：

- 可从 $f[j][k-1]$ 状态而来：先是经过 $k-1$ 的跳跃到达位置 j，再在原步长的基础上 `+1`，跳到了位置 i。
- 可从 $f[j][k]$ 状态而来：先是经过 k 的跳跃到达位置 j，维持原步长不变，跳到了位置 i。
- 可从 $f[j][k+1]$ 状态而来：先是经过 $k+1$ 的跳跃到达位置 j，再在原步长的基础上 `-1`，跳到了位置 i。

只要上述三种情况其中一种为真，则 $f[i][j]$ 为真。

至此，我们解决了动态规划的「状态定义」&「状态转移方程」部分。

但这就结束了吗？还没有。

我们还缺少可让状态递推下去的「有效值」，或者说缺少初始化环节。

因为我们的 $f[i][k]$ 依赖于之前的状态进行“或运算”而来，转移方程本身不会产生 $true$ 值。因此为了让整个「递推」过程可滚动，我们需要先有一个为 $true$ 的状态值。

这时候再回看我们的状态定义：**当前在第 i 个位置，并且是以步长 k 跳到位置 i 时，是否到达最后一块石子。**

显然，我们事先是不可能知道经过「多大的步长」跳到「哪些位置」，最终可以到达最后一块石

子。

这时候需要利用「对偶性」将跳跃过程「翻转」过来分析：

我们知道起始状态是「经过步长为 1」的跳跃到达「位置 1」，如果从起始状态出发，存在一种方案到达最后一块石子的话，那么必然存在一条反向路径，它是以从「最后一块石子」开始，并以「某个步长 k」开始跳跃，最终以回到位置 1。

因此我们可以设 $f[1][1] = true$，作为我们的起始值。

这里本质是利用「路径可逆」的性质，将问题进行了「等效对偶」。表面上我们是进行「正向递推」，但事实上我们是在验证是否存在某条「反向路径」到达位置 1。

建议大家加强理解 ~

代码：

```
class Solution {
    public boolean canCross(int[] ss) {
        int n = ss.length;
        // check first step
        if (ss[1] != 1) return false;
        boolean[][] f = new boolean[n + 1][n + 1];
        f[1][1] = true;
        for (int i = 2; i < n; i++) {
            for (int j = 1; j < i; j++) {
                int k = ss[i] - ss[j];
                // 我们知道从位置 j 到位置 i 是需要步长为 k 的跳跃

                // 而从位置 j 发起的跳跃最多不超过 j + 1
                // 因为每次跳跃，下标至少增加 1，而步长最多增加 1
                if (k <= j + 1) {
                    f[i][k] = f[j][k - 1] || f[j][k] || f[j][k + 1];
                }
            }
        }
        for (int i = 1; i < n; i++) {
            if (f[n - 1][i]) return true;
        }
        return false;
    }
}
```

- 时间复杂度：$O(n^2)$

- 空间复杂度：$O(n^2)$

BFS

事实上，前面我们也说到，解决超时 `DFS` 问题，除了增加「记忆化」功能以外，还能使用带标记的 `BFS`。

因为两者都能解决 `DFS` 的超时原因：大量的重复计算。

但为了「记忆化搜索」&「动态规划」能够更好的衔接，所以我把 `BFS` 放到最后。

如果你能够看到这里，那么这里的 `BFS` 应该看起来会相对轻松。

它更多是作为「记忆化搜索」的另外一种实现形式。

代码：

```Java
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    public boolean canCross(int[] ss) {
        int n = ss.length;
        for (int i = 0; i < n; i++) {
            map.put(ss[i], i);
        }
        // check first step
        if (!map.containsKey(1)) return false;

        boolean[][] vis = new boolean[n][n];
        Deque<int[]> d = new ArrayDeque<>();
        vis[1][1] = true;
        d.addLast(new int[]{1, 1});

        while (!d.isEmpty()) {
            int[] poll = d.pollFirst();
            int idx = poll[0], k = poll[1];
            if (idx == n - 1) return true;
            for (int i = -1; i <= 1; i++) {
                if (k + i == 0) continue;
                int next = ss[idx] + k + i;
                if (map.containsKey(next)) {
                    int nIdx = map.get(next), nK = k + i;
                    if (nIdx == n - 1) return true;
                    if (!vis[nIdx][nK]) {
                        vis[nIdx][nK] = true;
                        d.addLast(new int[]{nIdx, nK});
                    }
                }
            }
        }

        return false;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

题目描述

这是 LeetCode 上的 **752. 打开转盘锁** ，难度为 **中等**。

Tag：「双向 BFS」、「启发式搜索」、「AStar 算法」、「IDAStar 算法」

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有10个数字： ‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’ 。每个拨轮可以自由旋转：例如把 ‘9’ 变为 ‘0’，‘0’ 变为 ‘9’ 。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 ‘0000’ ，一个代表四个拨轮的数字的字符串。

列表 deadends 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 target 代表可以解锁的数字，你需要给出解锁需要的最小旋转次数，如果无论如何不能解锁，返回 -1 。

示例 1:

```
输入：deadends = ["0201","0101","0102","1212","2002"], target = "0202"
输出：6
解释：
可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" -> "0202"。
注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的，
因为当拨动到 "0102" 时这个锁就会被锁定。
```

示例 2:

```
输入: deadends = ["8888"], target = "0009"
输出：1
解释：
把最后一位反向旋转一次即可 "0000" -> "0009"。
```

示例 3:

```
输入: deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"], target = "8888"
输出：-1
解释：
无法旋转到目标数字且不被锁定。
```

示例 4:

```
输入: deadends = ["0000"], target = "8888"
输出：-1
```

提示：

- 1 <= deadends.length <= 500
- deadends[i].length == 4
- target.length == 4
- target 不在 deadends 之中
- target 和 deadends[i] 仅由若干位数字组成

基本分析

首先，我建议你先做「127. 单词接龙」，然后再回过头将本题作为「练习题」。

「127. 单词接龙」定位困难，而本题定位中等。主要体现在数据范围上，思维难度上「127. 单词接龙」并不比本题难，大胆做。

「127. 单词接龙」原题链接在 这里，相关题解在 这里。

回到本题，根据题意，可以确定这是一个「最短路/最小步数」问题。

此类问题，通常我们会使用「BFS」求解，但朴素的 BFS 通常会带来搜索空间爆炸问题。

因此我们可以使用与 (题解)127. 单词接龙 类似的思路进行求解。

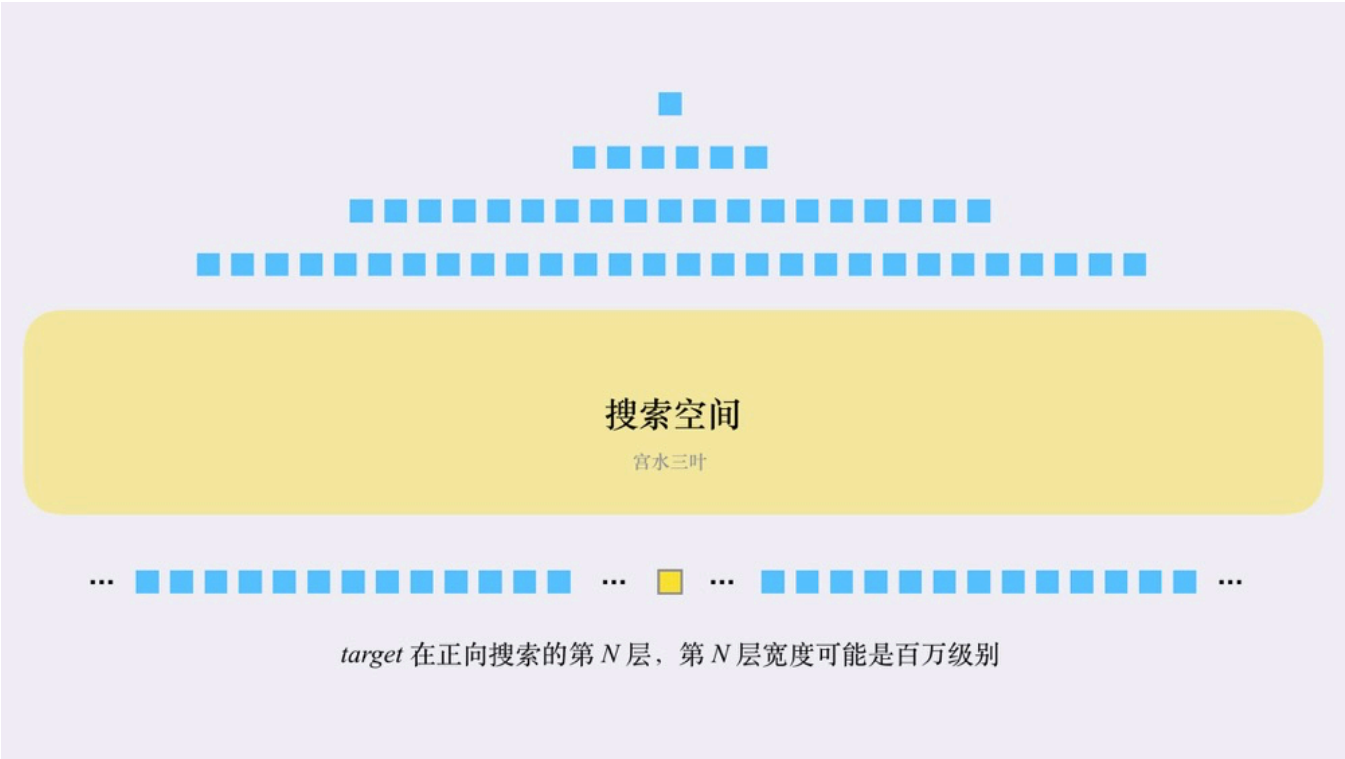
双向 BFS

我们知道，递归树的展开形式是一棵多阶树。

使用朴素 BFS 进行求解时，队列中最多会存在“两层”的搜索节点。

因此搜索空间的上界取决于 **目标节点所在的搜索层次的深度所对应的宽度**。

下图展示了朴素 BFS 可能面临的搜索空间爆炸问题：



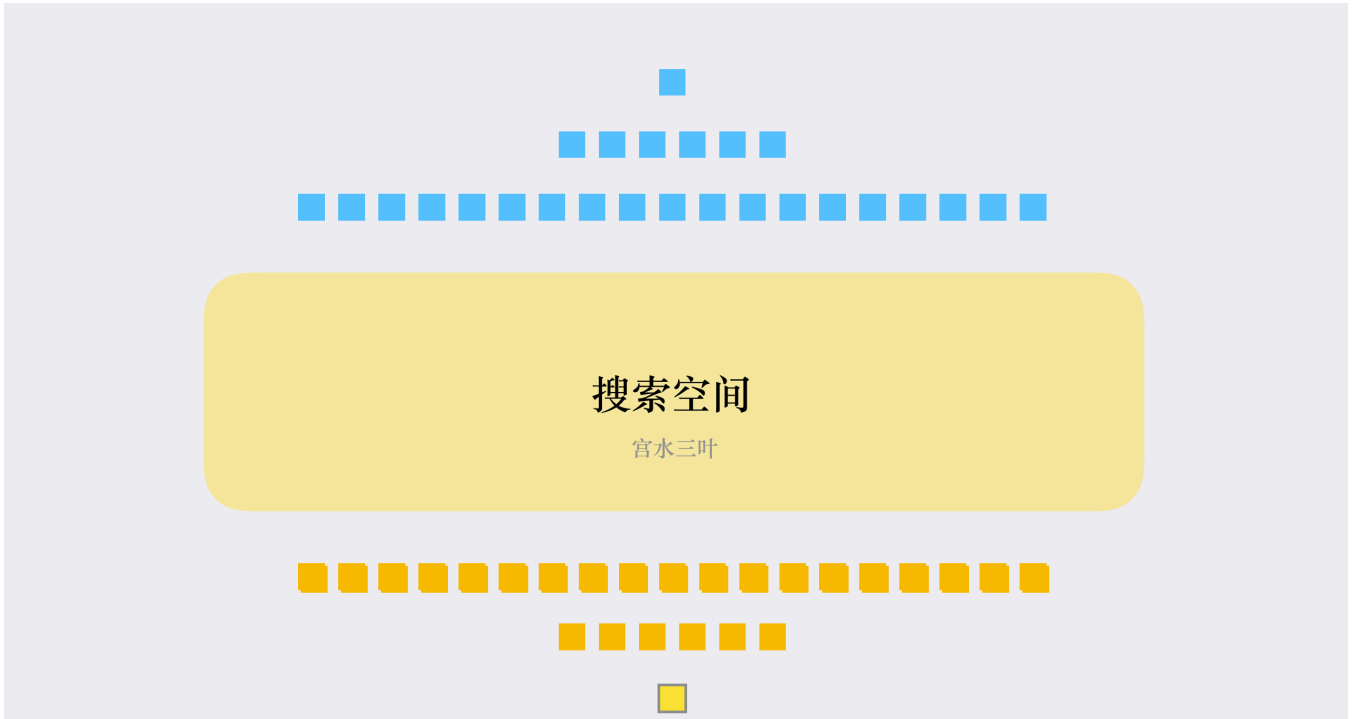
在朴素的 BFS 实现中，空间的瓶颈主要取决于搜索空间中的最大宽度。

那么有没有办法让我们不使用这么宽的搜索空间，同时又能保证搜索到目标结果呢？

「双向 BFS」 可以很好的解决这个问题：

同时从两个方向开始搜索，一旦搜索到相同的值，意味着找到了一条联通起点和终点的最短路径。

对于「有解」、「有一定数据范围」同时「层级节点数量以倍数或者指数级别增长」的情况，「双向 BFS」的搜索空间通常只有「朴素 BFS」的空间消耗的几百分之一，甚至几千分之一。



同时在「源点」和「汇点」出发进行搜索，搜索空间的宽度将会被大大收窄

「双向 BFS」的基本实现思路如下：

1. 创建「两个队列」分别用于两个方向的搜索；
2. 创建「两个哈希表」用于「解决相同节点重复搜索」和「记录转换次数」；
3. 为了尽可能让两个搜索方向“平均”，每次从队列中取值进行扩展时，先判断哪个队列容量较少；
4. 如果在搜索过程中「搜索到对方搜索过的节点」，说明找到了最短路径。

「双向 BFS」基本思路对应的伪代码大致如下：

```
d1、d2 为两个方向的队列
m1、m2 为两个方向的哈希表，记录每个节点距离起点的

// 只有两个队列都不空，才有必要继续往下搜索
// 如果其中一个队列空了，说明从某个方向搜到底都搜不到该方向的目标节点
while(!d1.isEmpty() && !d2.isEmpty()) {
    if (d1.size() < d2.size()) {
        update(d1, m1, m2);
    } else {
        update(d2, m2, m1);
    }
}

// update 为从队列 d 中取出一个元素进行「一次完整扩展」的逻辑
void update(Deque d, Map cur, Map other) {}
```

回到本题，我们看看如何使用「双向 BFS」进行求解。

估计不少同学是第一次接触「双向 BFS」，因此这次我写了大量注释。

建议大家带着对「双向 BFS」的基本理解去阅读。

代码：

```java
class Solution {
    String t, s;
    Set<String> set = new HashSet<>();
    public int openLock(String[] _ds, String _t) {
        s = "0000";
        t = _t;
        if (s.equals(t)) return 0;
        for (String d : _ds) set.add(d);
        if (set.contains(s)) return -1;
        int ans = bfs();
        return ans;
    }
    int bfs() {
        // d1 代表从起点 s 开始搜索（正向）
        // d2 代表从结尾 t 开始搜索（反向）
        Deque<String> d1 = new ArrayDeque<>(), d2 = new ArrayDeque<>();
        /*
         * m1 和 m2 分别记录两个方向出现的状态是经过多少次转换而来
         * e.g
         * m1 = {"1000":1} 代表 "1000" 由 s="0000" 替换 1 次字符而来
         * m2 = {"9999":3} 代表 "9999" 由 t="9996" 替换 3 次字符而来
         */
        Map<String, Integer> m1 = new HashMap<>(), m2 = new HashMap<>();
        d1.addLast(s);
        m1.put(s, 0);
        d2.addLast(t);
        m2.put(t, 0);

        /*
         * 只有两个队列都不空，才有必要继续往下搜索
         * 如果其中一个队列空了，说明从某个方向搜到底都搜不到该方向的目标节点
         * e.g.
         * 例如，如果 d1 为空了，说明从 s 搜索到底都搜索不到 t，反向搜索也没必要进行了
         */
        while (!d1.isEmpty() && !d2.isEmpty()) {
            int t = -1;
            if (d1.size() <= d2.size()) {
                t = update(d1, m1, m2);
            } else {
                t = update(d2, m2, m1);
            }
            if (t != -1) return t;
        }
        return -1;
    }
    int update(Deque<String> deque, Map<String, Integer> cur, Map<String, Integer> other)
```

```
            String poll = deque.pollFirst();
            char[] pcs = poll.toCharArray();
            int step = cur.get(poll);
            // 枚举替换哪个字符
            for (int i = 0; i < 4; i++) {
                // 能「正向转」也能「反向转」，这里直接枚举偏移量 [-1,1] 然后跳过 0
                for (int j = -1; j <= 1; j++) {
                    if (j == 0) continue;

                    // 求得替换字符串 str
                    int origin = pcs[i] - '0';
                    int next = (origin + j) % 10;
                    if (next == -1) next = 9;

                    char[] clone = pcs.clone();
                    clone[i] = (char)(next + '0');
                    String str = String.valueOf(clone);

                    if (set.contains(str)) continue;
                    if (cur.containsKey(str)) continue;

                    // 如果在「另一方向」找到过，说明找到了最短路，否则加入队列
                    if (other.containsKey(str)) {
                        return step + 1 + other.get(str);
                    } else {
                        deque.addLast(str);
                        cur.put(str, step + 1);
                    }
                }
            }
            return -1;
        }
    }
```

AStar 算法

可以直接根据本题规则来设计 A* 的「启发式函数」。

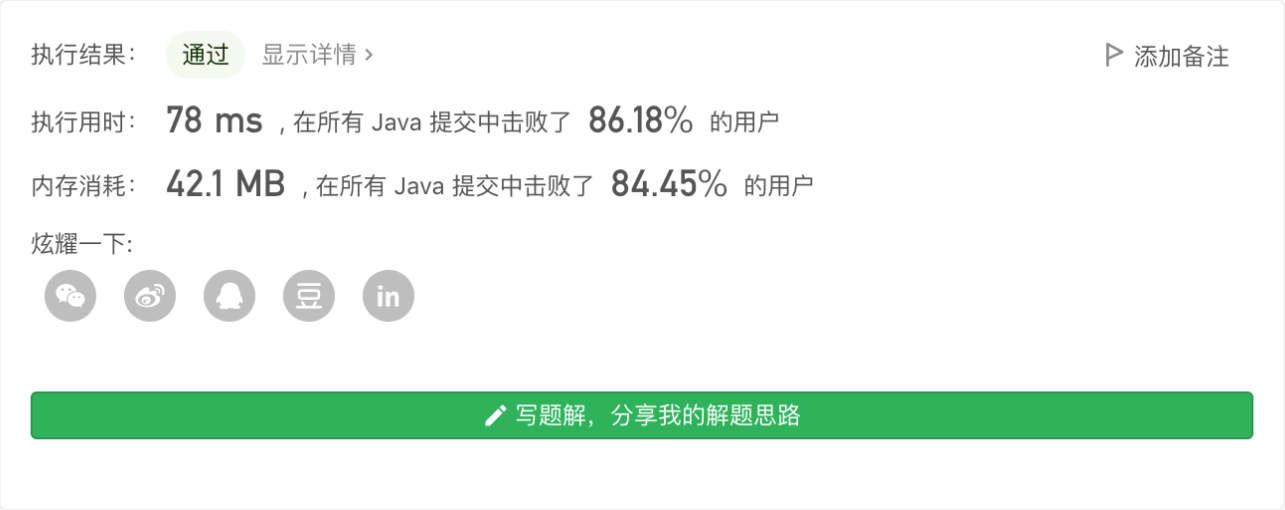
比如对于两个状态 `a` 和 `b` 可直接计算出「理论最小转换次数」：**不同字符的转换成本之和** 。

需要注意的是：由于我们衡量某个字符 `str` 的估值是以目标字符串 `target` 为基准，因此我们只能确保 `target` 出队时为「距离最短」，而不能确保中间节点出队时「距离最短」，因此

我们不能单纯根据某个节点是否「曾经入队」而决定是否入队，还要结合当前节点的「最小距离」是否被更新而决定是否入队。

这一点十分关键，在代码层面上体现在 `map.get(str).step > poll.step + 1` 的判断上。

注意：本题用 A* 过了，但通常我们需要先「确保有解」，A* 的启发搜索才会发挥真正价值。而本题，除非 `t` 本身在 `deadends` 中，其余情况我们无法很好提前判断「是否有解」。对于无解的情况 A* 效果不如「双向 BFS」。



代码：

```java
class Solution {
    class Node {
        String str;
        int val, step;
        /**
         *  str : 对应字符串
         *  val : 估值（与目标字符串 target 的最小转换成本）
         *  step: 对应字符串是经过多少步转换而来
         */
        Node(String _str, int _val, int _step) {
            str = _str;
            val = _val;
            step = _step;
        }
    }
    int f(String str) {
        int ans = 0;
        for (int i = 0; i < 4; i++) {
            int cur = str.charAt(i) - '0', target = t.charAt(i) - '0';
            int a = Math.min(cur, target), b = Math.max(cur, target);
            // 在「正向转」和「反向转」之间取 min
            int min = Math.min(b - a, a + 10 - b);
            ans += min;
        }
        return ans;
    }
    String s, t;
    Set<String> set = new HashSet<>();
    public int openLock(String[] ds, String _t) {
        s = "0000";
        t = _t;
        if (s.equals(t)) return 0;
        for (String d : ds) set.add(d);
        if (set.contains(s)) return -1;

        PriorityQueue<Node> q = new PriorityQueue<>((a,b)->a.val-b.val);
        Map<String, Node> map = new HashMap<>();
        Node root = new Node(s, f(s), 0);
        q.add(root);
        map.put(s, root);
        while (!q.isEmpty()) {
            Node poll = q.poll();
            char[] pcs = poll.str.toCharArray();
            int step = poll.step;
            if (poll.str.equals(t)) return step;
            for (int i = 0; i < 4; i++) {
```

```
                for (int j = -1; j <= 1; j++) {
                    if (j == 0) continue;
                    int cur = pcs[i] - '0';
                    int next = (cur + j) % 10;
                    if (next == -1) next = 9;

                    char[] clone = pcs.clone();
                    clone[i] = (char)(next + '0');
                    String str = String.valueOf(clone);

                    if (set.contains(str)) continue;
                    // 如果 str 还没搜索过，或者 str 的「最短距离」被更新，则入队
                    if (!map.containsKey(str) || map.get(str).step > step + 1) {
                        Node node = new Node(str, step + 1 + f(str), step + 1);
                        map.put(str, node);
                        q.add(node);
                    }
                }
            }
        }
        return -1;
    }
}
```

IDA* 算法

同样我们可以使用基于 `DFS` 的启发式 IDA* 算法：

- 仍然使用 `f()` 作为估值函数
- 利用旋转次数有限：总旋转次数不会超过某个阈值 $\max$。
- 利用「迭代加深」的思路找到最短距离

理想情况下，由于存在正向旋转和反向旋转，每一位转轮从任意数字开始到达任意数字，消耗次数不会超过 5 次，因此理想情况下可以设定 $\max = 5 * 4$ 。

但考虑 `deadends` 的存在，我们需要将 $\max$ 定义得更加保守一些：$\max = 10 * 4$ 。

但这样的阈值设定，加上 IDA* 算法每次会重复遍历「距离小于与目标节点距离」的所有节点，会有很大的 TLE 风险。

因此我们需要使用动态阈值：不再使用固定的阈值，而是利用 `target` 计算出「最大的转移成

本」作为我们的「最深数量级」。

PS. 上述的阈值分析是科学做法。对于本题可以利用数据弱，直接使用 $\max = 5 * 4$ 也可以通过，并且效果不错。

但必须清楚 $\max = 5 * 4$ 可能是一个错误的阈值，本题起点为 `0000`，考虑将所有正向转换的状态都放入 `deadends` 中，`target` 为 `2222`。这时候我们可以只限定 `0000` 先变为 `9999` 再往回变为 `2222` 的通路不在 `deadends` 中。

这时候使用 $\max = 5 * 4$ 就不对，但本题数据弱，可以通过（想提交错误数据拿积分吗？别试了，我已经提交了 🤣

代码：

```java
class Solution {
    String s, t;
    String cur;
    Set<String> set = new HashSet<>();
    Map<String, Integer> map = new HashMap<>();
    public int openLock(String[] ds, String _t) {
        s = "0000";
        t = _t;
        if (s.equals(t)) return 0;
        for (String d : ds) set.add(d);
        if (set.contains(s)) return -1;

        int depth = 0, max = getMax();
        cur = s;
        map.put(cur, 0);
        while (depth <= max && !dfs(0, depth)) {
            map.clear();
            cur = s;
            map.put(cur, 0);
            depth++;
        }
        return depth > max ? -1 : depth;
    }
    int getMax() {
        int ans = 0;
        for (int i = 0; i < 4; i++) {
            int origin = s.charAt(i) - '0', next = t.charAt(i) - '0';
            int a = Math.min(origin, next), b = Math.max(origin, next);
            int max = Math.max(b - a, a + 10 - b);
            ans += max;
        }
        return ans;
    }
    int f() {
        int ans = 0;
        for (int i = 0; i < 4; i++) {
            int origin = cur.charAt(i) - '0', next = t.charAt(i) - '0';
            int a = Math.min(origin, next), b = Math.max(origin, next);
            int min = Math.min(b - a, a + 10 - b);
            ans += min;
        }
        return ans;
    }
    boolean dfs(int u, int max) {
        if (u + f() > max) return false;
        if (f() == 0) return true;
```

```
        String backup = cur;
        char[] cs = cur.toCharArray();
        for (int i = 0; i < 4; i++) {
            for (int j = -1; j <= 1; j++) {
                if (j == 0) continue;
                int origin = cs[i] - '0';
                int next = (origin + j) % 10;
                if (next == -1) next = 9;
                char[] clone = cs.clone();
                clone[i] = (char)(next + '0');
                String str = String.valueOf(clone);
                if (set.contains(str)) continue;
                if (!map.containsKey(str) || map.get(str) > u + 1) {
                    cur = str;
                    map.put(str, u + 1);
                    if (dfs(u + 1, max)) return true;
                    cur = backup;
                }
            }
        }
        return false;
    }
}
```

> **🍭 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 **773. 滑动谜题** ，难度为 **困难**。

Tag : 「BFS」、「最小步数」、「AStar 算法」、「启发式搜索」

在一个 2 x 3 的板上（board）有 5 块砖瓦，用数字 1~5 来表示, 以及一块空缺用 0 来表示.

一次移动定义为选择 0 与一个相邻的数字（上下左右）进行交换.

最终当板 board 的结果是 1,2,3],[4,5,0 谜板被解开。

给出一个谜板的初始状态，返回最少可以通过多少次移动解开谜板，如果不能解开谜板，则返回 -1 。

示例：

```
输入：board = [[1,2,3],[4,0,5]]
输出：1
解释：交换 0 和 5 ，1 步完成
```

```
输入：board = [[1,2,3],[5,4,0]]
输出：-1
解释：没有办法完成谜板
```

```
输入：board = [[4,1,2],[5,0,3]]
输出：5
解释：
最少完成谜板的最少移动次数是 5 ，
一种移动路径:
尚未移动: [[4,1,2],[5,0,3]]
移动 1 次: [[4,1,2],[0,5,3]]
移动 2 次: [[0,1,2],[4,5,3]]
移动 3 次: [[1,0,2],[4,5,3]]
移动 4 次: [[1,2,0],[4,5,3]]
移动 5 次: [[1,2,3],[4,5,0]]
```

```
输入：board = [[3,2,4],[1,5,0]]
输出：14
```

提示：

- board 是一个如上所述的 2 x 3 的数组.
- board[i][j] 是一个 [0, 1, 2, 3, 4, 5] 的排列.

基本分析

这是八数码问题的简化版：将 $3 * 3$ 变为 $2 * 3$，同时将「输出路径」变为「求最小步数」。

通常此类问题可以使用「BFS」、「AStar 算法」、「康拓展开」进行求解。

由于问题简化到了 $2 * 3$，我们使用前两种解法即可。

BFS

为了方便，将原来的二维矩阵转成字符串（一维矩阵）进行处理。

这样带来的好处直接可以作为哈希 `Key` 使用，也可以很方便进行「二维坐标」与「一维下标」的转换。

由于固定是 $2 * 3$ 的格子，因此任意的合法二维坐标 (x, y) 和对应一维下标 idx 可通过以下转换：

- $idx = x * 3 + y$
- $x = idx / 3, y = idx \% 3$

其余的就是常规的 `BFS` 过程了。

代码：

```Java
class Solution {
    class Node {
        String str;
        int x, y;
        Node(String _str, int _x, int _y) {
            str = _str; x = _x; y = _y;
        }
    }
    int n = 2, m = 3;
    String s, e;
    int x, y;
    public int slidingPuzzle(int[][] board) {
        s = "";
        e = "123450";
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                s += board[i][j];
                if (board[i][j] == 0) {
                    x = i; y = j;
                }
            }
        }
        int ans = bfs();
        return ans;
    }
    int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
    int bfs() {
        Deque<Node> d = new ArrayDeque<>();
        Map<String, Integer> map = new HashMap<>();
        Node root = new Node(s, x, y);
        d.addLast(root);
        map.put(s, 0);
        while (!d.isEmpty()) {
            Node poll = d.pollFirst();
            int step = map.get(poll.str);
            if (poll.str.equals(e)) return step;
            int dx = poll.x, dy = poll.y;
            for (int[] di : dirs) {
                int nx = dx + di[0], ny = dy + di[1];
                if (nx < 0 || nx >= n || ny < 0 || ny >= m) continue;
                String nStr = update(poll.str, dx, dy, nx, ny);
                if (map.containsKey(nStr)) continue;
                Node next = new Node(nStr, nx, ny);
                d.addLast(next);
                map.put(nStr, step + 1);
            }
```

```
        }
        return -1;
    }
    String update(String cur, int i, int j, int p, int q) {
        char[] cs = cur.toCharArray();
        char tmp = cs[i * m + j];
        cs[i * m + j] = cs[p * m + q];
        cs[p * m + q] = tmp;
        return String.valueOf(cs);
    }
}
```

A* 算法

可以直接根据本题规则来设计 A* 的「启发式函数」。

比如对于两个状态 `a` 和 `b` 可直接计算出「理论最小转换次数」：**所有位置的数值「所在位置」与「目标位置」的曼哈顿距离之和（即横纵坐标绝对值之和）**。

注意，我们只需要计算「非空格」位置的曼哈顿距离即可，因为空格的位置会由其余数字占掉哪些位置而唯一确定。

A* 求最短路的正确性问题：由于我们衡量某个状态 `str` 的估值是以目标字符串 `e=123450` 为基准，因此我们只能确保 `e` 出队时为「距离最短」，而不能确保中间节点出队时「距离最短」，因此我们不能单纯根据某个节点是否「曾经入队」而决定是否入队，还要结合当前节点的「最小距离」是否被更新而决定是否入队。

这一点十分关键，在代码层面上体现在 `map.get(nStr) > step + 1` 的判断上。

我们知道，A* 在有解的情况下，才会发挥「启发式搜索」的最大价值，因此如果我们能够提前判断无解的情况，对 A* 算法来说会是巨大的提升。

而对于通用的 $N * N$ 数码问题，判定有解的一个充要条件是：**「逆序对」数量为偶数，如果不满足，必然无解，直接返回 -1 即可。**

对该结论的充分性证明和必要性证明完全不在一个难度上，所以建议记住这个结论即可。

代码：

```Java
class Solution {
    class Node {
        String str;
        int x, y;
        int val;
        Node(String _str, int _x, int _y, int _val) {
            str = _str; x = _x; y = _y; val = _val;
        }
    }
    int f(String str) {
        int ans = 0;
        char[] cs1 = str.toCharArray(), cs2 = e.toCharArray();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                // 跳过「空格」，计算其余数值的曼哈顿距离
                if (cs1[i * m + j] == '0' || cs2[i * m + j] == '0') continue;
                int cur = cs1[i * m + j], next = cs2[i * m + j];
                int xd = Math.abs((cur - 1) / 3 - (next - 1) / 3);
                int yd = Math.abs((cur - 1) % 3 - (next - 1) % 3);
                ans += (xd + yd);
            }
        }
        return ans;
    }
    int n = 2, m = 3;
    String s, e;
    int x, y;
    public int slidingPuzzle(int[][] board) {
        s = "";
        e = "123450";
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                s += board[i][j];
                if (board[i][j] == 0) {
                    x = i; y = j;
                }
            }
        }

        // 提前判断无解情况
        if (!check(s)) return -1;

        int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
        Node root = new Node(s, x, y, f(s));
        PriorityQueue<Node> q = new PriorityQueue<>((a,b)->a.val-b.val);
        Map<String, Integer> map = new HashMap<>();
```

```
        q.add(root);
        map.put(s, 0);
        while (!q.isEmpty()) {
            Node poll = q.poll();
            int step = map.get(poll.str);
            if (poll.str.equals(e)) return step;
            int dx = poll.x, dy = poll.y;
            for (int[] di : dirs) {
                int nx = dx + di[0], ny = dy + di[1];
                if (nx < 0 || nx >= n || ny < 0 || ny >= m) continue;
                String nStr = update(poll.str, dx, dy, nx, ny);
                if (!map.containsKey(nStr) || map.get(nStr) > step + 1) {
                    Node next = new Node(nStr, nx, ny, step + 1 + f(nStr));
                    q.add(next);
                    map.put(nStr, step + 1);
                }
            }
        }
        return 0x3f3f3f3f; // never
    }
    String update(String cur, int i, int j, int p, int q) {
        char[] cs = cur.toCharArray();
        char tmp = cs[i * m + j];
        cs[i * m + j] = cs[p * m + q];
        cs[p * m + q] = tmp;
        return String.valueOf(cs);
    }
    boolean check(String str) {
        char[] cs = str.toCharArray();
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < n * m; i++) {
            if (cs[i] != '0') list.add(cs[i] - '0');
        }
        int cnt = 0;
        for (int i = 0; i < list.size(); i++) {
            for (int j = i + 1; j < list.size(); j++) {
                if (list.get(i) < list.get(j)) cnt++;
            }
        }
        return cnt % 2 == 0;
    }
}
```

> **🍭 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 **815. 公交路线** ，难度为 **困难**。

Tag：「图论 BFS」、「双向 BFS」、「图论搜索」

给你一个数组 `routes` ，表示一系列公交线路，其中每个 `routes[i]` 表示一条公交线路，第 `i` 辆公交车将会在上面循环行驶。

- 例如，路线 `routes[0] = [1, 5, 7]` 表示第 0 辆公交车会一直按序列 `1 -> 5 -> 7 -> 1 -> 5 -> 7 -> 1 -> ...` 这样的车站路线行驶。

现在从 `source` 车站出发（初始时不在公交车上），要前往 `target` 车站。 期间仅可乘坐公交车。

求出 最少乘坐的公交车数量 。如果不可能到达终点车站，返回 `-1` 。

示例 1：

```
输入：routes = [[1,2,7],[3,6,7]], source = 1, target = 6

输出：2

解释：最优策略是先乘坐第一辆公交车到达车站 7 , 然后换乘第二辆公交车到车站 6 。
```

示例 2：

```
输入：routes = [[7,12],[4,5,15],[6],[15,19],[9,12,13]], source = 15, target = 12

输出：-1
```

提示：

- 1 <= routes.length <= 500.
- 1 <= routes[i].length <= 10^5
- routes[i] 中的所有值 互不相同
- sum(routes[i].length) <= 10^5
- 0 <= routes[i][j] < 10^6

- 0 <= source, target < 10^6

基本分析

为了方便，我们令每个公交站为一个「车站」，由一个「车站」可以进入一条或多条「路线」。

问题为从「起点车站」到「终点车站」，所进入的最少路线为多少。

抽象每个「路线」为一个点，当不同「路线」之间存在「公共车站」则为其增加一条边权为 1 的无向边。

单向 BFS

由于是在边权为 1 的图上求最短路，我们直接使用 `BFS` 即可。

起始时将「起点车站」所能进入的「路线」进行入队，每次从队列中取出「路线」时，查看该路线是否包含「终点车站」：

- 包含「终点车站」：返回进入该线路所花费的距离
- 不包含「终点车站」：遍历该路线所包含的车站，将由这些车站所能进入的路线，进行入队

一些细节：由于是求最短路，同一路线重复入队是没有意义的，因此将新路线入队前需要先判断是否曾经入队。

执行结果： 通过 显示详情 >

添加备注

执行用时： 83 ms ，在所有 Java 提交中击败了 41.82% 的用户

内存消耗： 84.9 MB ，在所有 Java 提交中击败了 12.68% 的用户

炫耀一下：

写题解，分享我的解题思路

代码：

```java
class Solution {
    int s, t;
    int[][] rs;
    public int numBusesToDestination(int[][] _rs, int _s, int _t) {
        rs = _rs; s = _s; t = _t;
        if (s == t) return 0;
        int ans = bfs();
        return ans;
    }
    int bfs() {
        // 记录某个车站可以进入的路线
        Map<Integer, Set<Integer>> map = new HashMap<>();
        // 队列存的是经过的路线
        Deque<Integer> d = new ArrayDeque<>();
        // 哈希表记录的进入该路线所使用的距离
        Map<Integer, Integer> m = new HashMap<>();
        int n = rs.length;
        for (int i = 0; i < n; i++) {
            for (int station : rs[i]) {
                // 将从起点可以进入的路线加入队列
                if (station == s) {
                    d.addLast(i);
                    m.put(i, 1);
                }
                Set<Integer> set = map.getOrDefault(station, new HashSet<>());
                set.add(i);
                map.put(station, set);
            }
        }
        while (!d.isEmpty()) {
            // 取出当前所在的路线，与进入该路线所花费的距离
            int poll = d.pollFirst();
            int step = m.get(poll);

            // 遍历该路线所包含的车站
            for (int station : rs[poll]) {
                // 如果包含终点，返回进入该路线花费的距离即可
                if (station == t) return step;

                // 将由该线路的车站发起的路线，加入队列
                Set<Integer> lines = map.get(station);
                if (lines == null) continue;
                for (int nr : lines) {
                    if (!m.containsKey(nr)) {
                        m.put(nr, step + 1);
                        d.add(nr);
```

```
                    }
                }
            }
        }
        return -1;
    }
}
```

- 时间复杂度：令路线的数量为 n，车站的数量为 m。建图的时间复杂度为 $O(\sum_{i=0}^{n-1} len(rs[i]))$；`BFS` 部分每个路线只会入队一次，最坏情况下每个路线都包含所有车站，复杂度为 $O(n * m)$。整体复杂度为 $O(n * m + \sum_{i=0}^{n-1} len(rs[i]))$。
- 空间复杂度：$O(n * m)$

双向 BFS（并查集预处理无解情况）

另外一个做法是使用双向 `BFS`。

首先建图方式不变，将「起点」和「终点」所能进入的路线分别放入两个方向的队列，如果「遇到公共的路线」或者「当前路线包含了目标位置」，说明找到了最短路径。

另外我们知道，双向 `BFS` 在无解的情况下不如单向 `BFS`。因此我们可以先使用「并查集」进行预处理，判断「起点」和「终点」是否连通，如果不联通，直接返回 -1，有解才调用双向 `BFS`。

由于使用「并查集」预处理的复杂度与建图是近似的，增加这样的预处理并不会越过我们时空复杂度的上限，因此这样的预处理是有益的。一定程度上可以最大化双向 `BFS` 减少搜索空间的效益。

代码：

```java
class Solution {
    static int N = (int)1e6+10;
    static int[] p = new int[N];
    int find(int x) {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }
    void union(int a, int b) {
        p[find(a)] = p[find(b)];
    }
    boolean query(int a, int b) {
        return find(a) == find(b);
    }
    int s, t;
    int[][] rs;
    public int numBusesToDestination(int[][] _rs, int _s, int _t) {
        rs = _rs; s = _s; t = _t;
        if (s == t) return 0;
        for (int i = 0; i < N; i++) p[i] = i;
        for (int[] r : rs) {
            for (int loc : r) {
                union(loc, r[0]);
            }
        }
        if (!query(s, t)) return -1;
        int ans = bfs();
        return ans;
    }
    // 记录某个车站可以进入的路线
    Map<Integer, Set<Integer>> map = new HashMap<>();
    int bfs() {
        Deque<Integer> d1 = new ArrayDeque<>(), d2 = new ArrayDeque<>();
        Map<Integer, Integer> m1 = new HashMap<>(), m2 = new HashMap<>();

        int n = rs.length;
        for (int i = 0; i < n; i++) {
            for (int station : rs[i]) {
                // 将从起点可以进入的路线加入正向队列
                if (station == s) {
                    d1.addLast(i);
                    m1.put(i, 1);
                }
                // 将从终点可以进入的路线加入反向队列
                if (station == t) {
                    d2.addLast(i);
                    m2.put(i, 1);
```

```
                }
                Set<Integer> set = map.getOrDefault(station, new HashSet<>());
                set.add(i);
                map.put(station, set);
            }
        }

        // 如果「起点所发起的路线」和「终点所发起的路线」有交集，直接返回 1
        Set<Integer> s1 = map.get(s), s2 = map.get(t);
        Set<Integer> tot = new HashSet<>();
        tot.addAll(s1);
        tot.retainAll(s2);
        if (!tot.isEmpty()) return 1;

        // 双向 BFS
        while (!d1.isEmpty() && !d2.isEmpty()) {
            int res = -1;
            if (d1.size() <= d2.size()) {
                res = update(d1, m1, m2);
            } else {
                res = update(d2, m2, m1);
            }
            if (res != -1) return res;
        }

        return 0x3f3f3f3f; // never
    }
    int update(Deque<Integer> d, Map<Integer, Integer> cur, Map<Integer, Integer> other) {
        // 取出当前所在的路线，与进入该路线所花费的距离
        int poll = d.pollFirst();
        int step = cur.get(poll);

        // 遍历该路线所包含的车站
        for (int station : rs[poll]) {
            // 遍历将由该线路的车站发起的路线
            Set<Integer> lines = map.get(station);
            if (lines == null) continue;
            for (int nr : lines) {
                if (cur.containsKey(nr)) continue;
                if (other.containsKey(nr)) return step + other.get(nr);
                cur.put(nr, step + 1);
                d.add(nr);
            }
        }

        return -1;
```

```
    }
}
```

- 时间复杂度：令路线的数量为 n，车站的个数为 m。并查集和建图的时间复杂度为 $O(\sum_{i=0}^{n-1} len(rs[i]))$；`BFS` 求最短路径的复杂度为 $O(n * m)$。整体复杂度为 $O(n * m + \sum_{i=0}^{n-1} len(rs[i]))$。
- 空间复杂度：$O(n * m + \sum_{i=0}^{n-1} len(rs[i]))$

> **🍥 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**。

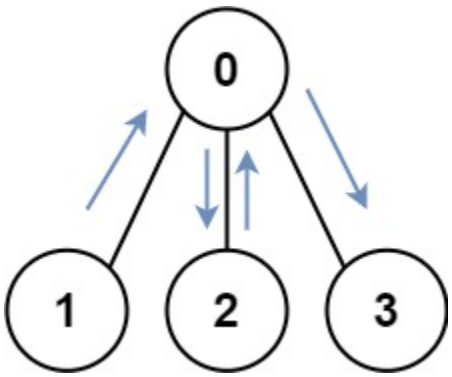
Tag：「图」、「图论 BFS」、「动态规划」、「状态压缩」

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：

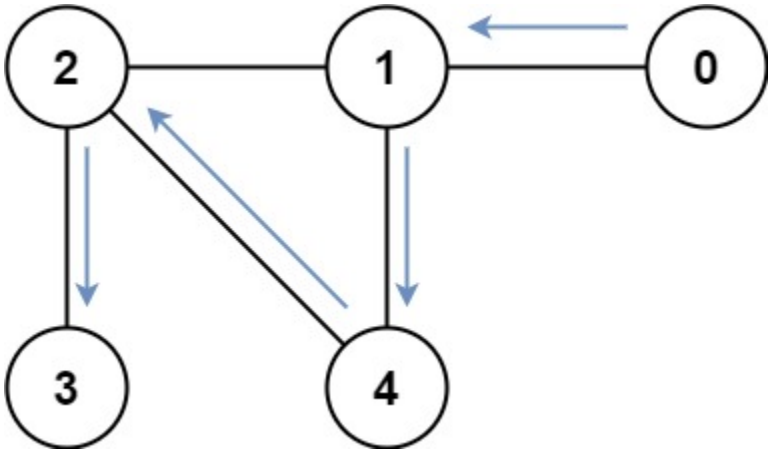


```
输入：graph = [[1,2,3],[0],[0],[0]]

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。

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？

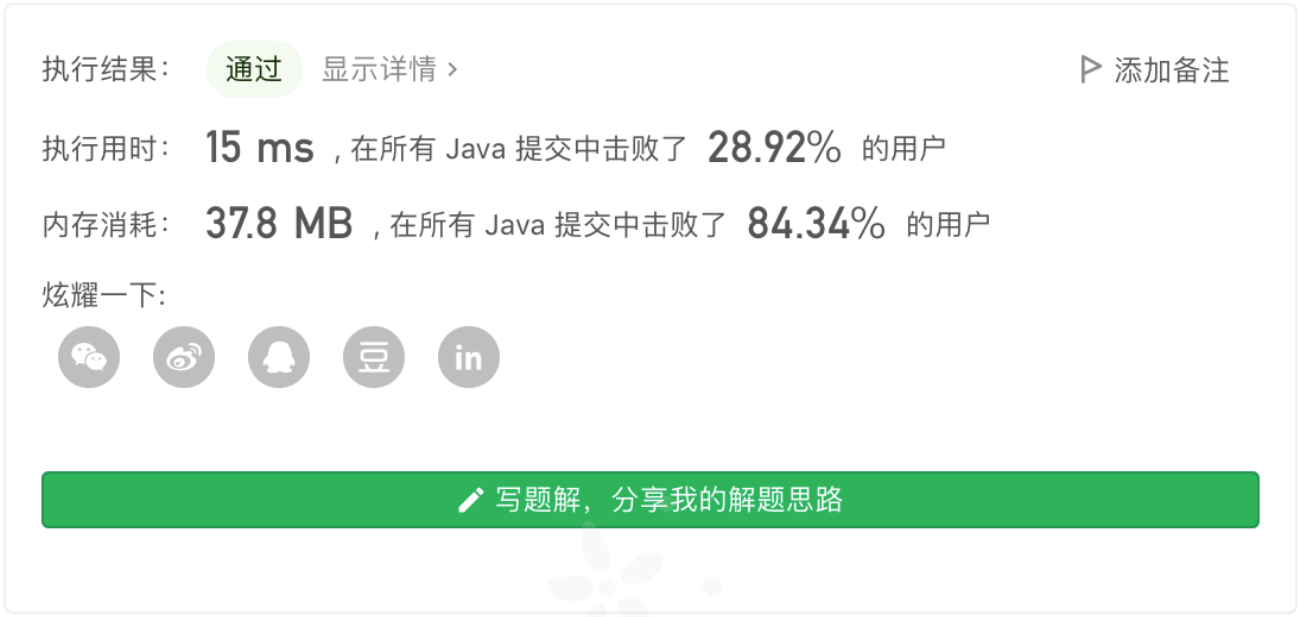
是因为如果从「常规的 DP 转移思路」出发，状态之间不存在拓扑序（有环），这就导致了我们在计算某个 $f[state][u]$ 时，它所依赖的状态并不确保已经被计算/更新完成，所以我们无法使用常规的 DP 手段来求解。

> 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;
    }
}
```

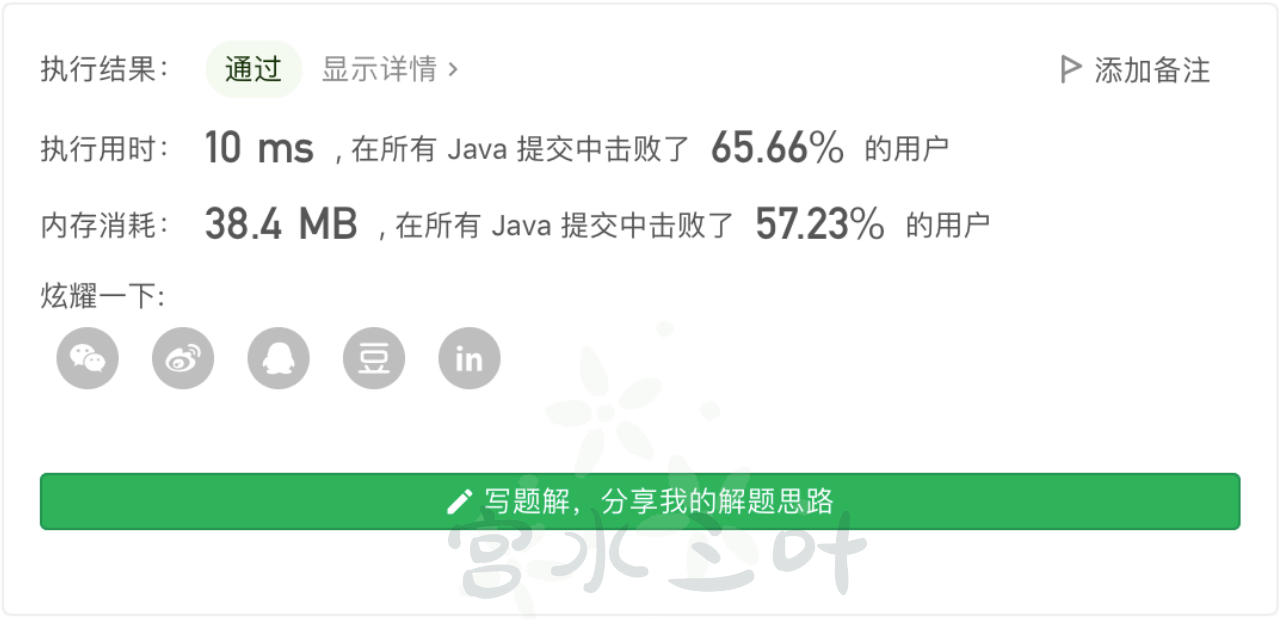
- 时间复杂度：Floyd 复杂度为 $O(n^3)$；DP 共有 $n * 2^n$ 个状态需要被转移，每次转移复杂度为 $O(n)$，总的复杂度为 $O(n^2 * 2^n)$。整体复杂度为 $O(\max(n^3, n^2 * 2^n))$
- 空间复杂度：$O(n * 2^n)$

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
    }
}
```

题目描述

这是 LeetCode 上的 **863. 二叉树中所有距离为 K 的结点** ，难度为 **中等**。

Tag：「图论 BFS」、「图论 DFS」、「二叉树」

给定一个二叉树（具有根结点 root），一个目标结点 target ，和一个整数值 K 。

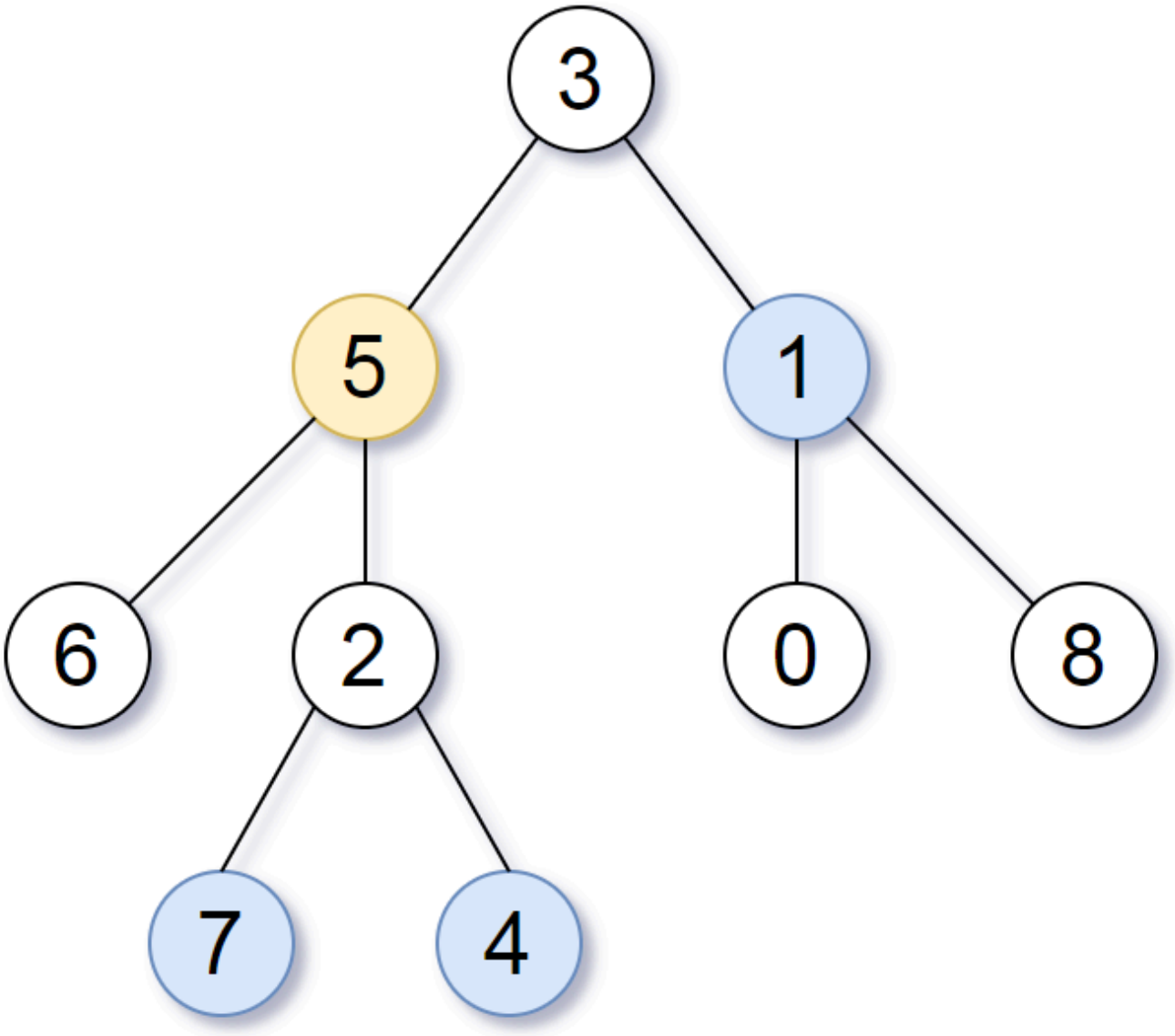
返回到目标结点 target 距离为 K 的所有结点的值的列表。 答案可以以任何顺序返回。

示例 1：

```
输入：root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2

输出：[7,4,1]

解释：
所求结点为与目标结点（值为 5）距离为 2 的结点，
值分别为 7，4，以及 1
```



注意，输入的 “root” 和 “target” 实际上是树上的结点。
上面的输入仅仅是对这些对象进行了序列化描述。

提示：

- **给定的树是非空的。**
- **树上的每个结点都具有唯一的值** 0 <= node.val <= 500 **。**
- **目标结点** target **是树上的结点。**
- 0 <= K <= 1000.

基本分析

显然，如果题目是以图的形式给出的话，我们可以很容易通过「`BFS` / 迭代加深」找到距离为 k 的节点集。

而树是一类特殊的图，我们可以通过将二叉树转换为图的形式，再进行「`BFS` / 迭代加深」。

由于二叉树每个点最多有 2 个子节点，点和边的数量接近，属于稀疏图，因此我们可以使用「邻接表」的形式进行存储。

建图方式为：对于二叉树中相互连通的节点（`root` 与 `root.left`、`root` 和 `root.right`），建立一条无向边。

建图需要遍历整棵树，使用 `DFS` 或者 `BFS` 均可。

由于所有边的权重均为 1，我们可以使用「`BFS` / 迭代加深」找到从目标节点 `target` 出发，与目标节点距离为 k 的节点，然后将其添加到答案中。

> 一些细节：利用每个节点具有唯一的值，我们可以直接使用节点值进行建图和搜索。

建图 + `BFS`

由「基本分析」，可写出「建图 + `BFS`」的实现。

代码：

```java
class Solution {
    int N = 1010, M = N * 2;
    int[] he = new int[N], e = new int[M], ne = new int[M];
    int idx;
    void add(int a, int b) {
        e[idx] = b;
        ne[idx] = he[a];
        he[a] = idx++;
    }
    boolean[] vis = new boolean[N];
    public List<Integer> distanceK(TreeNode root, TreeNode t, int k) {
        List<Integer> ans = new ArrayList<>();
        Arrays.fill(he, -1);
        dfs(root);
        Deque<Integer> d = new ArrayDeque<>();
        d.addLast(t.val);
        vis[t.val] = true;
        while (!d.isEmpty() && k >= 0) {
            int size = d.size();
            while (size-- > 0) {
                int poll = d.pollFirst();
                if (k == 0) {
                    ans.add(poll);
                    continue;
                }
                for (int i = he[poll]; i != -1 ; i = ne[i]) {
                    int j = e[i];
                    if (!vis[j]) {
                        d.addLast(j);
                        vis[j] = true;
                    }
                }
            }
            k--;
        }
        return ans;
    }
    void dfs(TreeNode root) {
        if (root == null) return;
        if (root.left != null) {
            add(root.val, root.left.val);
            add(root.left.val, root.val);
            dfs(root.left);
        }
        if (root.right != null) {
            add(root.val, root.right.val);
```

```
            add(root.right.val, root.val);
            dfs(root.right);
        }
    }
}
```

- 时间复杂度：通过 `DFS` 进行建图的复杂度为 $O(n)$；通过 `BFS` 找到距离 $target$ 为 k 的节点，复杂度为 $O(n)$。整体复杂度为 $O(n)$
- 空间复杂度：$O(n)$

建图 + 迭代加深

由「基本分析」，可写出「建图 + 迭代加深」的实现。

迭代加深的形式，我们只需要结合题意，搜索深度为 k 的这一层即可。

代码：

```java
class Solution {
    int N = 1010, M = N * 2;
    int[] he = new int[N], e = new int[M], ne = new int[M];
    int idx;
    void add(int a, int b) {
        e[idx] = b;
        ne[idx] = he[a];
        he[a] = idx++;
    }
    boolean[] vis = new boolean[N];
    public List<Integer> distanceK(TreeNode root, TreeNode t, int k) {
        List<Integer> ans = new ArrayList<>();
        Arrays.fill(he, -1);
        dfs(root);
        vis[t.val] = true;
        find(t.val, k, 0, ans);
        return ans;
    }
    void find(int root, int max, int cur, List<Integer> ans) {
        if (cur == max) {
            ans.add(root);
            return ;
        }
        for (int i = he[root]; i != -1; i = ne[i]) {
            int j = e[i];
            if (!vis[j]) {
                vis[j] = true;
                find(j, max, cur + 1, ans);
            }
        }
    }
    void dfs(TreeNode root) {
        if (root == null) return;
        if (root.left != null) {
            add(root.val, root.left.val);
            add(root.left.val, root.val);
            dfs(root.left);
        }
        if (root.right != null) {
            add(root.val, root.right.val);
            add(root.right.val, root.val);
            dfs(root.right);
        }
    }
}
```

- 时间复杂度：通过 `DFS` 进行建图的复杂度为 $O(n)$；通过迭代加深找到距离 $target$ 为 k 的节点，复杂度为 $O(n)$。整体复杂度为 $O(n)$
- 空间复杂度：$O(n)$

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

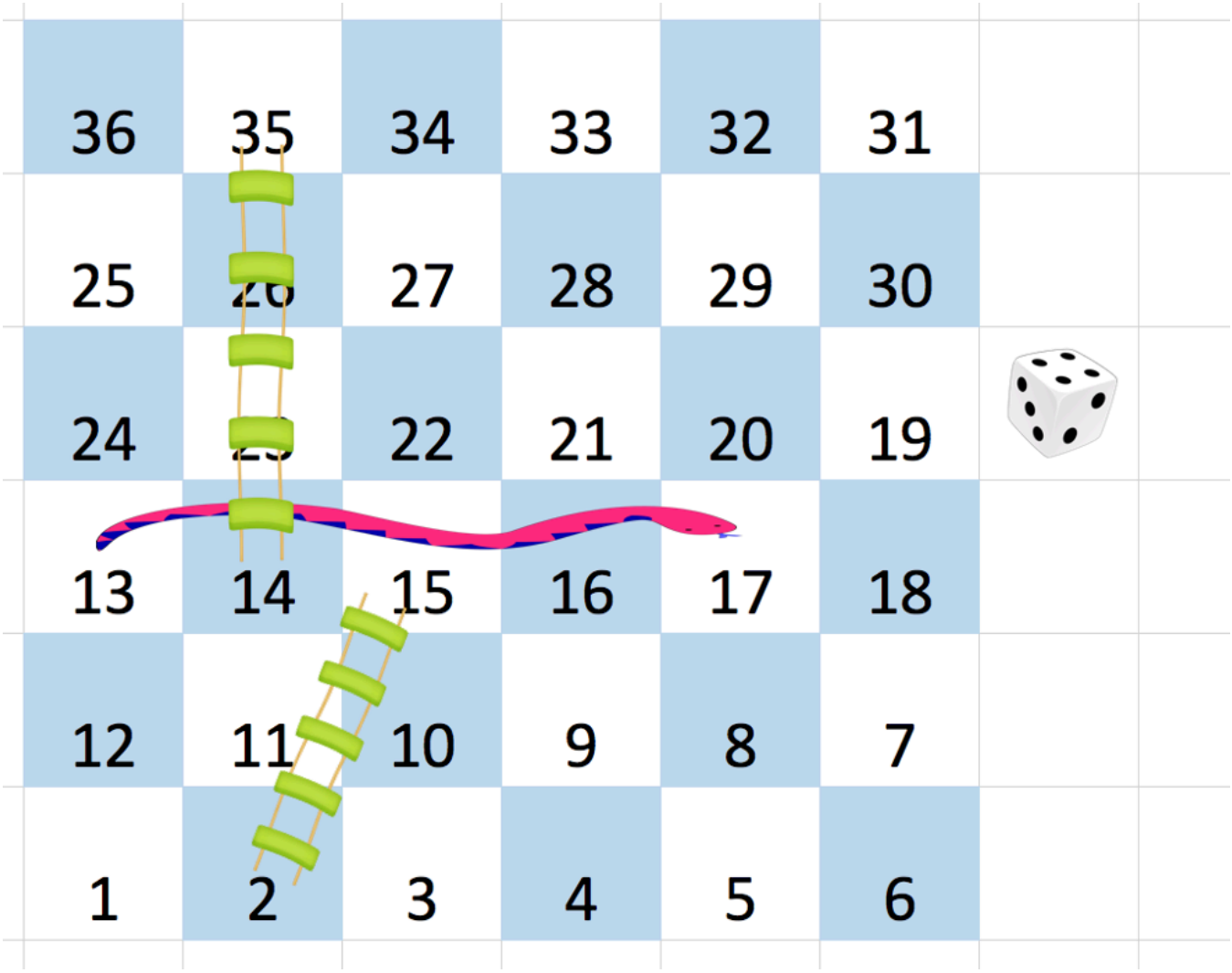
题目描述

这是 LeetCode 上的 **909. 蛇梯棋** ，难度为 **中等**。

Tag : 「图论 BFS」

$N * N$ 的棋盘 board 上，按从 1 到 $N * N$ 的数字给方格编号，编号 从左下角开始，每一行交替方向。

例如，一块 $6x6$ 大小的棋盘，编号如下：



r 行 c 列的棋盘，按前述方法编号，棋盘格中可能存在 “蛇” 或 “梯子”；如果 `board[r][c] != -1`，那个蛇或梯子的目的地将会是 `board[r][c]`。

玩家从棋盘上的方格 1（总是在最后一行、第一列）开始出发。

每一回合，玩家需要从当前方格 x 开始出发，按下述要求前进：

- 选定目标方格：选择从编号 `x+1`，`x+2`，`x+3`，`x+4`，`x+5`，或者 `x+6` 的方格中选出一个目标方格 s，目标方格的编号 <= $N * N$。
    - 该选择模拟了掷骰子的情景，无论棋盘大小如何，你的目的地范围也只能处于区间 [x+1, x+6] 之间。
- 传送玩家：如果目标方格 S 处存在蛇或梯子，那么玩家会传送到蛇或梯子的目的地。否则，玩家传送到目标方格 S。
  注意，玩家在每回合的前进过程中最多只能爬过蛇或梯子一次：就算目的地是另一条蛇或梯子的起点，你也不会继续移动。

返回达到方格 $N * N$ 所需的最少移动次数，如果不可能，则返回 -1。

示例：

```
输入：[
[-1,-1,-1,-1,-1,-1],
[-1,-1,-1,-1,-1,-1],
[-1,-1,-1,-1,-1,-1],
[-1,35,-1,-1,13,-1],
[-1,-1,-1,-1,-1,-1],
[-1,15,-1,-1,-1,-1]]

输出：4

解释：
首先，从方格 1 [第 5 行，第 0 列] 开始。
你决定移动到方格 2，并必须爬过梯子移动到到方格 15。
然后你决定移动到方格 17 [第 3 行，第 5 列]，必须爬过蛇到方格 13。
然后你决定移动到方格 14，且必须通过梯子移动到方格 35。
然后你决定移动到方格 36, 游戏结束。
可以证明你需要至少 4 次移动才能到达第 N*N 个方格，所以答案是 4。
```

提示：

- 2 <= board.length = board[0].length <= 20
- `board[i][j]` 介于 1 和 $N * N$ 之间或者等于 -1。
- 编号为 1 的方格上没有蛇或梯子。
- 编号为 $N * N$ 的方格上没有蛇或梯子。

BFS

最多有 $20 * 20$ 个格子，直接使用常规的单向 `BFS` 进行求解即可。

为了方便我们可以按照题目给定的意思，将二维的矩阵「扁平化」为一维的矩阵，然后再按照规则进行 `BFS`。

代码：

```Java
class Solution {
    int n;
    int[] nums;
    public int snakesAndLadders(int[][] board) {
        n = board.length;
        if (board[0][0] != -1) return -1;
        nums = new int[n * n + 1];
        boolean isRight = true;
        for (int i = n - 1, idx = 1; i >= 0; i--) {
            for (int j = (isRight ? 0 : n - 1); isRight ? j < n : j >= 0; j += isRight ? 1
                nums[idx++] = board[i][j];
            }
            isRight = !isRight;
        }
        int ans = bfs();
        return ans;
    }
    int bfs() {
        Deque<Integer> d = new ArrayDeque<>();
        Map<Integer, Integer> m = new HashMap<>();
        d.addLast(1);
        m.put(1, 0);
        while (!d.isEmpty()) {
            int poll = d.pollFirst();
            int step = m.get(poll);
            if (poll == n * n) return step;
            for (int i = 1; i <= 6; i++) {
                int np = poll + i;
                if (np <= 0 || np > n * n) continue;
                if (nums[np] != -1) np = nums[np];
                if (m.containsKey(np)) continue;
                m.put(np, step + 1);
                d.addLast(np);
            }
        }
        return -1;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

题目描述

这是 LeetCode 上的 **1162. 地图分析** ，难度为 **中等**。

Tag：「图论 BFS」、「多源 BFS」

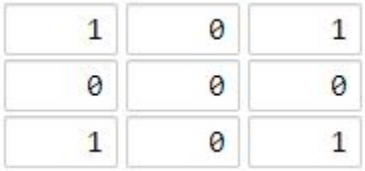
你现在手里有一份大小为 $N * N$ 的 网格 $grid$，上面的每个 单元格 都用 0 和 1 标记好了。

其中 0 代表海洋，1 代表陆地，请你找出一个海洋单元格，这个海洋单元格到离它最近的陆地单元格的距离是最大的。

我们这里说的距离是「曼哈顿距离」：$(x0, y0)$ 和 $(x1, y1)$ 这两个单元格之间的距离是 $|x0 - x1| + |y0 - y1|$。

如果网格上只有陆地或者海洋，请返回 -1。

示例 1：

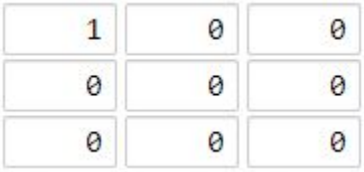


```
输入：[[1,0,1],[0,0,0],[1,0,1]]

输出：2

解释：海洋单元格 (1, 1) 和所有陆地单元格之间的距离都达到最大，最大距离为 2。
```

示例 2：



```
输入：[[1,0,0],[0,0,0],[0,0,0]]

输出：4

解释：海洋单元格 (2, 2) 和所有陆地单元格之间的距离都达到最大，最大距离为 4。
```

提示：

- 1 <= grid.length == grid[0].length <= 100
- grid[i][j] 不是 0 就是 1

单源 BFS

通常我们使用 BFS 求最短路，都是针对如下场景：从特定的起点出发，求解到达特定终点的最短距离。

这是一类特殊的「单源最短路」问题：本质是在一个边权为 1 的图上，求从特定「源点」出发到达特定「汇点」的最短路径。

对于本题，如果套用「单源最短路」做法，我们需要对每个「海洋」位置做一次 BFS：求得每个「海洋」的最近陆地距离，然后在所有的距离中取 max 作为答案。

单次 BFS 的最坏情况需要扫描完整个矩阵，复杂度为 $O(n^2)$。

同时，最多有 n^2 个海洋区域需要做 BFS，因此这样的做法复杂度为 $O(n^4)$，并且 $O(n^4)$ 可直接取满。

PS. 数据范围为 10^2，理论上是一定会超时，但本题数据较弱，Java 2021/06/28 可过。

一些细节：为了方便，我们在使用哈希表记录距离时，将二维坐标 (x, y) 转化为对应的一维下标 $idx = x * n + y$ 作为 key 进行存储。

代码：

```Java
class Solution {
    int n;
    int[][] grid;
    public int maxDistance(int[][] _grid) {
        grid = _grid;
        n = grid.length;
        int ans = -1;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 0) {
                    ans = Math.max(ans, bfs(i, j));
                }
            }
        }
        return ans;
    }
    // 单次 BFS：求解海洋位置 (x,y) 最近的陆地距离
    int bfs(int x, int y) {
        int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
        Deque<int[]> d = new ArrayDeque<>();
        Map<Integer, Integer> map = new HashMap<>();
        d.addLast(new int[]{x, y});
        map.put(x * n + y, 0);
        while (!d.isEmpty()) {
            int[] poll = d.pollFirst();
            int dx = poll[0], dy = poll[1];
            int step = map.get(dx * n + dy);
            if (grid[dx][dy] == 1) return step;
            for (int[] di : dirs) {
                int nx = dx + di[0], ny = dy + di[1];
                if (nx < 0 || nx >= n || ny < 0 || ny >= n) continue;
                int key = nx * n + ny;
                if (map.containsKey(key)) continue;
                d.addLast(new int[]{nx, ny});
                map.put(key, step + 1);
            }
        }
        return -1;
    }
}
```

- 时间复杂度：$O(n^4)$
- 空间复杂度：$O(n^2)$

多源 BFS

这其实还是道「多源 BFS」入门题。

与「单源最短路」不同，「多源最短路」问题是求从「多个源点」到达「一个/多个汇点」的最短路径。

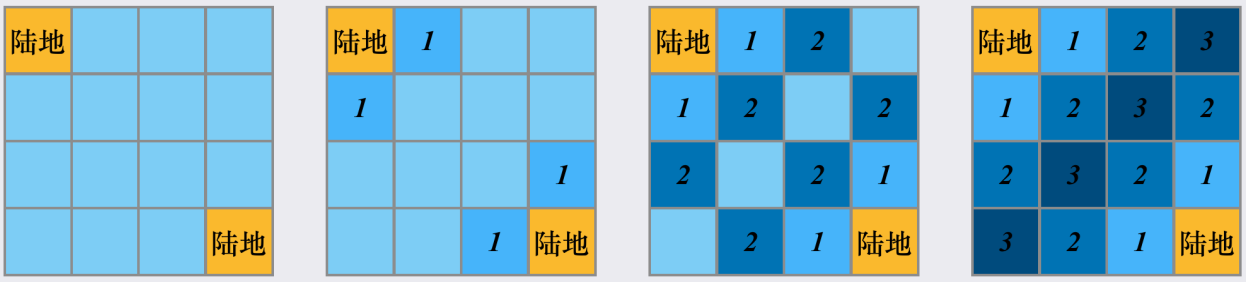
在实现上，最核心的搜索部分，「多源 BFS」与「单源 BFS」并无区别。

并且通过建立「虚拟源点」的方式，我们可以「多源 BFS」转换回「单源 BFS」问题。

什么意思？

以本题为例，题面要我们求每个「海洋」区域到最近的「陆地」区域的最大值。

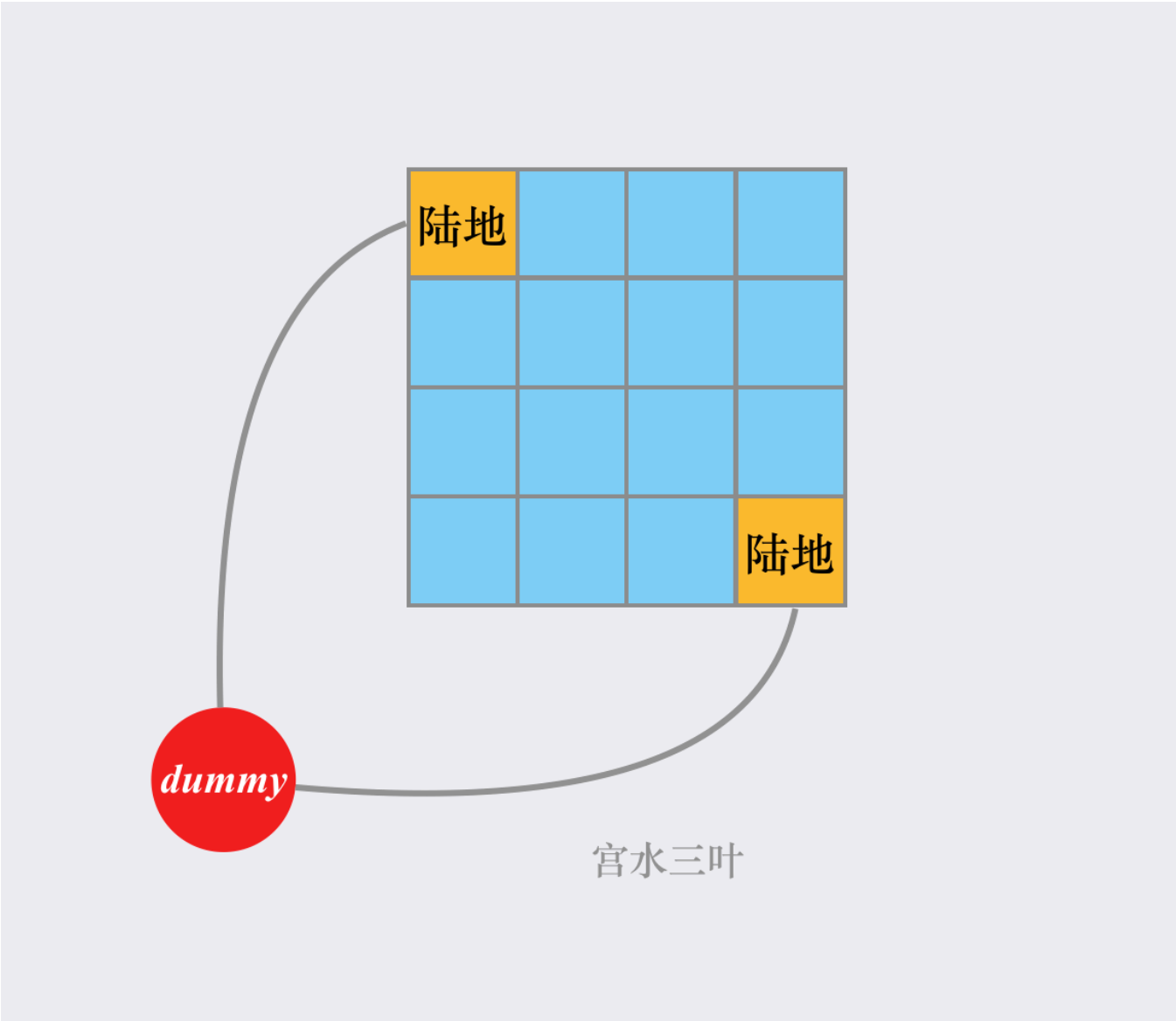
我们可以将「源点/起点」和「汇点/终点」进行反转：**从每个「陆地」区域出发，多个「陆地」区域每次同时向往扩散一圈，每个「海洋」区域被首次覆盖时所对应的圈数，就是「海洋」区域距离最近的「陆地」区域的距离。**



宫水三叶

不过，这是如何与「单源 BFS」联系起来的呢？

我们可以想象存在一个「虚拟源点」，其与所有「真实源点」（陆地）存在等权的边，那么任意「海洋」区域与「最近的陆地」区域的最短路等价于与「虚拟源点」的最短路：



实现上，我们并不需要真的将这个虚拟源点建立出来，只需要将所有的「真实源点」进行入队即可。

这个过程相当于从队列中弹出「虚拟源点」，并把它所能到点（真实源点）进行入队，然后再进行常规的 BFS 即可。

一些细节：实现上为了方便，在进行常规 BFS 时，如果一个「海洋」区域被访问到，说明其被离它「最近的陆地」覆盖到了，修改值为最小距离。这样我们只需要考虑那些值仍然为 0 的「海洋」区域即可（代表尚未被更新）。

代码：

```Java
class Solution {
    public int maxDistance(int[][] grid) {
        int n = grid.length;
        Deque<int[]> d = new ArrayDeque<>();
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) {
                    d.add(new int[]{i, j});
                    map.put(i * n + j, 0);
                }
            }
        }
        int ans = -1;
        int[][] dirs = new int[][]{{1,0},{-1,0},{0,1},{0,-1}};
        while (!d.isEmpty()) {
            int[] poll = d.poll();
            int dx = poll[0], dy = poll[1];
            int step = map.get(dx * n + dy);
            for (int[] di : dirs) {
                int nx = dx + di[0], ny = dy + di[1];
                if (nx < 0 || nx >= n || ny < 0 || ny >= n) continue;
                if (grid[nx][ny] != 0) continue;
                grid[nx][ny] = step + 1;
                d.add(new int[]{nx, ny});
                map.put(nx * n + ny, step + 1);
                ans = Math.max(ans, step + 1);
            }
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

总结

今天我们介绍了「多源 BFS」，通过建立「虚拟源点」，我们可以将其转化回「单源 BFS」问题。

实现上我们只需要将所有的「真实源点」进行入队，然后再进行 BFS 即可。

看起来两者区别不大，但其本质是通过源点/汇点转换，应用常规的 Flood Fill 将多次朴素 BFS 转化为一次 BFS，可以有效降低我们算法的时间复杂度。

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

💡**更新 Tips：本专题更新时间为 2021-10-07，大概每 2-4 周 集中更新一次。**

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「图论 BFS」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：

"给作者手机充个电"

YOLO 的赞赏码