BFS 算法框架套路详解

https://mp.weixin.qq.com/s?__biz=MzAxODQxMDM0Mw==&mid=2247485134&idx=1&sn=fd345f8a93dc4444bcc65c57bb46fc35&scene=21#wechat_redirect

BFS 的核心思想应该不难理解的，就是把一些问题抽象成图，从一个点开始，向四周开始扩散。一般来说，我们写 **BFS** 算法都是用「队列」这种数据结构每次将一个节点周围的所有节点加入队列。

BFS 相对 DFS 的最主要的区别是：**BFS** 找到的路径一定是最短的，但代价就是空间复杂度比 **DFS** 大很多，至于为什么，我们后面介绍了框架就很容易看出来了。

本文就由浅入深写两道 BFS 的典型题目，分别是「二叉树的最小高度」和「打开密码锁的最少步数」，手把手教你怎么写 BFS 算法。

一、算法框架

要说框架的话，我们先举例一下 **BFS** 出现的常见场景好吧，问题的本质就是让你在一幅「图」中找到从起点**start**到终点**target**的最近距离，这个例子听起来很枯燥，但是 **BFS** 算法问题其实都是在干这个事儿。

把枯燥的本质搞清楚了，再去欣赏各种问题的包装才能胸有成竹嘛。

这个广义的描述可以有各种变体，比如走迷宫，有的格子是围墙不能走，从起点到终点的最短距离是多少？如果这个迷宫带「传送门」可以瞬间传送呢？

再比如说两个单词，要求你通过某些替换，把其中一个变成另一个，每次只能替换一个字符，最少要替换几次？(move one step only)

再比如说连连看游戏，两个方块消除的条件不仅仅是图案相同，还得保证两个方块之间的最短连线不能多于两个拐点。你玩连连看，点击两个坐标，游戏是如何判断它俩的最短连线有几个拐点的？

再比如……

净整些花里胡哨的，这些问题都没啥奇技淫巧，本质上就是一幅「图」，让你从一个起点，走到终点，问最短路径。这就是 BFS 的本质，框架搞清楚了直接默写就好。

记住下面这个框架就 OK 了：

// 计算从起点 start 到终点 target 的最近距离

```
int BFS(Node start, Node target) {
    Queue<Node> q; // 核心数据结构
    Set<Node> visited; // 避免走回头路 letter 256 array, 2 D matrix

    q.offer(start); // 将起点加入队列
    visited.add(start);
    int step = 0; // 记录扩散的步数

    while (q not empty) {
        int sz = q.size();
        // 将当前队列中的所有节点向四周扩散
```

```
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            // 划重点：这里判断是否到达终点
            if (cur is target)
                return step;
            // 将 cur 的相邻节点加入队列
            for (Node x : cur.adj())
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
        }
        // 划重点：更新步数在这里
        step++;
    }
}
```

队列q就不说了，BFS 的核心数据结构；**cur.adj()泛指cur相邻的节点，比如说二维数组中，cur上下左右四面的位置就是相邻节点**；**visited**的主要作用是防止走回头路，大部分时候都是必须的，但是像一般的二叉树结构，没有子节点到父节点的指针，不会走回头路就不需要visited。

二叉树的最小高度

先来个简单的问题实践一下 BFS 框架吧，判断一棵二叉树的最小高度，这也是 LeetCode 第 111 题，看一下题目：

怎么套到 BFS 的框架里呢？首先明确一下起点start和终点target是什么，怎么判断到达了终点？显然起点就是**root**根节点，终点就是最靠近根节点的那个「叶子节点嘛，叶子节点就是两个子节点都是null的节点：

```
if (cur.left == null && cur.right == null) // 到达叶子节点
```

那么，按照我们上述的框架稍加改造来写解法即可：

```
int minDepth(TreeNode root) {
    if (root == null) return 0;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);
    // root 本身就是一层，depth 初始化为 1
    int depth = 1;

    while (!q.isEmpty()) {
        int sz = q.size();
```

```
        // 将当前队列中的所有节点向四周扩散
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            // 判断是否到达终点
            if (cur.left == null && cur.right == null)
                return depth;
            // 将 cur 的相邻节点加入队列
            if (cur.left != null)
                q.offer(cur.left);
            if (cur.right != null)
                q.offer(cur.right);
        }
        // 这里增加步数
        depth++;
    }
    return depth;
}
```

二叉树是很简单的数据结构，我想上述代码你应该可以理解的吧，其实其他复杂问题都是这个框架的变形，在探讨复杂问题之前，我们解答两个问题：

1、为什么 BFS 可以找到最短距离，DFS 不行吗？

首先，你看 BFS 的逻辑，depth每增加一次，队列中的所有节点都向前迈一步，这保证了第一次到达终点的时候，走的步数是最少的。

DFS 不能找最短路径吗？其实也是可以的，但是时间复杂度相对高很多。

你想啊，**DFS 实际上是靠递归的堆栈记录走过的路径**，你要找到最短路径，肯定得把二叉树中所有树杈都探索完才能对比出最短的路径有多长对不对 ？

而 BFS 借助队列做到一次一步「齐头并进」，是可以在不遍历完整棵树的条件下找到最短距离的。

形象点说，**DFS** 是线，**BFS** 是面;DFS 是单打独斗，BFS 是集体行动。这个应该比较容易理解吧。

2、既然 BFS 那么好，为啥 DFS 还要存在？

BFS 可以找到最短距离，但是空间复杂度高，而 **DFS** 的空间复杂度较低。**dfs stack overflow**

还是拿刚才我们处理二叉树问题的例子，假设给你的这个二叉树是满二叉树，节点总数为N，对于 DFS 算法来说，空间复杂度无非就是递归堆栈，最坏情况下顶多就是树的高度，也就是满二叉树 O(logN)。

但是你想想 BFS 算法，队列中每次都会储存着二叉树一层的节点，这样的话最坏情况下空间复杂度应该是树的最底层节点的数量，也就是N/2，用 Big O 表示的话也就是O(N)。
由此观之，**BFS** 还是有代价的，一般来说在找最短路径的时候使用 **BFS**，其他时候还是 **DFS** 使用得多一些（主要是递归代码好写）。
好了，现在你对 BFS 了解得足够多了，下面来一道难一点的题目，深化一下框架的理解吧。

解开密码锁的最少次数

这道 LeetCode 题目是第 752 题，比较有意思：

```
You have a lock in front of you with 4 circular wheels. Each wheel has 10 slots:
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'. The wheels can rotate freely and
wrap around: for example we can turn '9' to be '0', or '0' to be '9'. Each move
consists of turning one wheel and one slot.
The lock initially starts at '0000', a string representing the state of the 4
wheels.
You are given a list of dead ends, meaning if the lock displays any of these codes,
the wheels of the lock will stop turning and you will be unable to open it.
Given a target representing the value of the wheels that will unlock the lock,
return the minimum total number of turns required to open the lock, or -1 if it is
impossible.

Example 1:
Input: deadends = ["0201","0101","0102","1212","2002"], target = "0202"
Output: 6
Explanation:
A sequence of valid moves would be "0000" -> "1000" -> "1100" -> "1200" -> "1201"
-> "1202" -> "0202".
Note that a sequence like "0000" -> "0001" -> "0002" -> "0102" -> "0202" would be
invalid,
because the wheels of the lock become stuck after the display becomes the dead end
"0102".
```

题目中描述的就是我们生活中常见的那种密码锁，若果没有任何约束，最少的拨动次数很好算，就像我们平时开密码锁那样直奔密码拨就行了。
但现在的难点就在于，不能出现deadends，应该如何计算出最少的转动次数呢？
第一步，我们不管所有的限制条件，不管deadends和target的限制，就思考一个问题：如果让你设计一个算法，穷举所有可能的密码组合，你怎么做？
穷举呗，再简单一点，如果你只转一下锁，有几种可能 总共有 **4** 个位置，每个位置可以向上转，也可以向下转，也就是有 **8** 种可能对吧。
比如说从**"0000"**开始，转一次 可以穷举出"1000", "9000", "0100", "0900"...共 8 种密码。然后，再以这 8 种密码作为基础，对每个密码再转一下，穷举出所有可能...
仔细想想，这就可以抽象成一幅图，每个节点有 **8** 个相邻的节点，又让你求最短距离，这不就是典型的 BFS 嘛，框架就可以派上用场了，先写出一个「简陋」的 BFS 框架代码再说别的：

```java
// 将 s[j] 向上拨动一次
String plusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '9')
        ch[j] = '0';
    else
        ch[j] += 1;
    return new String(ch);
}
// 将 s[i] 向下拨动一次
String minusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '0')
        ch[j] = '9';
    else
        ch[j] -= 1;
    return new String(ch);
}
// BFS 框架，打印出所有可能的密码
void BFS(String target) {
    Queue<String> q = new LinkedList<>();
    q.offer("0000"); // start point

    while (!q.isEmpty()) {
        int sz = q.size();
        // 将当前队列中的所有节点向周围扩散
        for (int i = 0; i < sz; i++) {
            String cur = q.poll();
            // 判断是否到达终点
            System.out.println(cur);
            // 将一个节点的相邻节点加入队列
            for (int j = 0; j < 4; j++) {
                String up = plusOne(cur, j);
                String down = minusOne(cur, j);
                q.offer(up); // up string
                q.offer(down); // down string
            }
        }
        // 在这里增加步数
    }
    return;
}
```

PS:这段代码当然有很多问题，但是我们做算法题肯定不是一蹴而就的，而是从简陋到完美的。不要完美主义，咱要慢慢来，好不。
这段 BFS 代码已经能够穷举所有可能的密码组合了，但是显然不能完成题目，有如下问题需要解决：
1、会走回头路。比如说我们从"0000"拨到"1000"，但是等从队列拿出"1000"时，还会拨出一个"0000"，这样的话会产生死循环。
2、没有终止条件，按照题目要求，我们找到target就应该结束并返回拨动的次数。
3、没有对**deadends**的处理，按道理这些「死亡密码」是不能出现的，也就是说你遇到这些密码的时候需要跳过。

只要按照 BFS 框架在对应的位置稍作修改即可修复这些问题：

```
int openLock(String[] deadends, String target) {
    // 记录需要跳过的死亡密码
    Set<String> deads = new HashSet<>();
    for (String s : deadends) deads.add(s);
    // 记录已经穷举过的密码，防止走回头路
    Set<String> visited = new HashSet<>();
    Queue<String> q = new LinkedList<>();
    // 从起点开始启动广度优先搜索
    int step = 0;
    q.offer("0000");
    visited.add("0000");

    while (!q.isEmpty()) {
        int sz = q.size();
        // 将当前队列中的所有节点向周围扩散
        for (int i = 0; i < sz; i++) {
            String cur = q.poll();
            // 判断是否到达终点
            if (deads.contains(cur))
                continue;
            if (cur.equals(target))
                return step;

            // 将一个节点的未遍历相邻节点加入队列
            for (int j = 0; j < 4; j++) {
                String up = plusOne(cur, j);
                if (!visited.contains(up)) {
                    q.offer(up);
                    visited.add(up);
                }
                String down = minusOne(cur, j);
```

```
                if (!visited.contains(down)) {
                    q.offer(down);
                    visited.add(down);
                }
            }
        }
        // 在这里增加步数
        step++;
    }
    // 如果穷举完都没找到目标密码，那就是找不到了
    return -1;
}
```

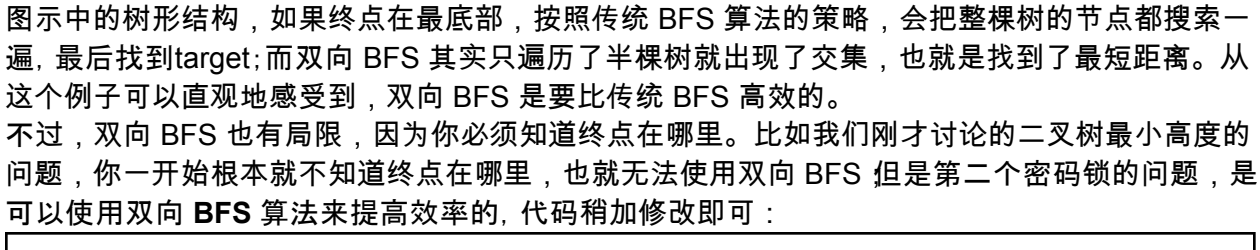
至此，我们就解决这道题目了。有一个比较小的优化：可以不需要dead这个哈希集合，可以直接将这些元素初始化到visited集合中，效果是一样的，可能更加优雅一些。

四、双向 BFS 优化

你以为到这里 BFS 算法就结束了？恰恰相反。BFS 算法还有一种稍微高级一点的优化思路：双向 BFS，可以进一步提高算法的效率。

篇幅所限，这里就提一下区别：传统的 BFS 框架就是从起点开始向四周扩散，遇到终点时停止；**而双向 BFS 则是从起点和终点同时开始扩散，当两边有交集的时候停止。**

为什么这样能够能够提升效率呢？其实从 Big O 表示法分析算法复杂度的话，它俩的最坏复杂度都是O(N)，但是实际上双向 BFS 确实会快一些，我给你画两张图看一眼就明白了：



图示中的树形结构，如果终点在最底部，按照传统 BFS 算法的策略，会把整棵树的节点都搜索一遍，最后找到target；而双向 BFS 其实只遍历了半棵树就出现了交集，也就是找到了最短距离。从这个例子可以直观地感受到，双向 BFS 是要比传统 BFS 高效的。

不过，双向 BFS 也有局限，因为你必须知道终点在哪里。比如我们刚才讨论的二叉树最小高度的问题，你一开始根本就不知道终点在哪里，也就无法使用双向 BFS；但是第二个密码锁的问题，是可以使用双向 **BFS** 算法来提高效率的，代码稍加修改即可：

```
int openLock(String[] deadends, String target) {
    Set<String> deads = new HashSet<>();
    for (String s : deadends) deads.add(s);
    // 用集合不用队列，可以快速判断元素是否存在
    Set<String> q1 = new HashSet<>();
    Set<String> q2 = new HashSet<>();
    Set<String> visited = new HashSet<>();

    int step = 0;
    q1.add("0000");
    q2.add(target);
```

```
    while (!q1.isEmpty() && !q2.isEmpty()) {
        // 哈希集合在遍历的过程中不能修改，用 temp 存储扩散结果
        Set<String> temp = new HashSet<>();
        // 将 q1 中的所有节点向周围扩散
        for (String cur : q1) {
            // 判断是否到达终点
            if (deads.contains(cur))
                continue;
            if (q2.contains(cur))
                return step;
            visited.add(cur);

            // 将一个节点的未遍历相邻节点加入集合
            for (int j = 0; j < 4; j++) {
                String up = plusOne(cur, j);
                if (!visited.contains(up))
                    temp.add(up);
                String down = minusOne(cur, j);
                if (!visited.contains(down))
                    temp.add(down);
            }
        }
        // 在这里增加步数
        step++;
        // temp 相当于 q1
        // 这里交换 q1 q2，下一轮 while 就是扩散 q2
        q1 = q2;
        q2 = temp;
    }
    return -1;
}
```

双向 **BFS** 还是遵循 **BFS** 算法框架的，只是不再使用队列，而是使用 **HashSet** 方便快速判断两个集合是否有交集。

另外的一个技巧点就是 **while** 循环的最后交换**q1**和**q2**的内容，所以只要默认扩散**q1**就相当于轮流扩散**q1**和**q2**。

其实双向 **BFS** 还有一个优化，就是在 **while** 循环开始时做一个判断：

```
// ...
while (!q1.isEmpty() && !q2.isEmpty()) {
    if (q1.size() > q2.size()) {
        // 交换 q1 和 q2
```

```
        temp = q1;
        q1 = q2;
        q2 = temp;
    }
    // ...
```

为什么这是一个优化呢？
因为按照 BFS 的逻辑，队列（集合）中的元素越多，扩散之后新的队列（集合）中的元素就越多；在双向 BFS 算法中，如果我们每次都选择一个较小的集合进行扩散，那么占用的空间增长速度就会慢一些，效率就会高一些。

不过话说回来，无论传统 **BFS** 还是双向 **BFS**，无论做不做优化，从 **Big O** 衡量标准来看，空间复杂度都是一样的，只能说双向 **BFS** 是一种 **trick** 吧，掌握不掌握其实都无所谓。最关键的是把 BFS 通用框架记下来，反正所有 BFS 算法都可以用它套出解法