搜索

大话搜索

搜索一般指在有限的状态空间中进行枚举，通过穷尽所有的可能来找到符合条件的解或者解的个数。根据搜索方式的不同，搜索算法可以分为 DFS，BFS，A*算法等。这里只介绍 DFS 和 BFS，以及发生在 DFS 上一种技巧-回溯。

搜索问题覆盖面非常广泛，并且在算法题中也占据了很高的比例。我甚至还在公开演讲中提到了 **前端算法面试中搜索类占据了很大的比重，尤其是国内公司**。

搜索专题中的子专题有很多，而大家所熟知的 BFS，DFS 只是其中特别基础的内容。除此之外，还有状态记录与维护，剪枝，联通分量，拓扑排序等等。这些内容，我会在这里一一给大家介绍。

另外即使仅仅考虑 DFS 和 BFS 两种基本算法，里面能玩的花样也非常多。比如 BFS 的双向搜索，比如 DFS 的前中后序，迭代加深等等。

关于搜索，其实在二叉树部分已经做了介绍了。而这里的搜索，其实就是进一步的泛化。数据结构不再局限于前面提到的数组，链表或者树。而扩展到了诸如二维数组，多叉树，图等。不过核心仍然是一样的，只不过数据结构发生了变化而已。

搜索的核心是什么？

实际上搜索题目**本质就是将题目中的状态映射为图中的点，将状态间的联系映射为图中的边。根据题目信息构建状态空间，然后对状态空间进行遍历，遍历过程需要记录和维护状态，并通过剪枝和数据结构等提高搜索效率。**

状态空间的数据结构不同会导致算法不同。比如对数组进行搜索，和对树，图进行搜索就不太一样。

再次强调一下，我这里讲的数组，树和图是**状态空间**的逻辑结构，而不是题目给的数据结构。比如题目给了一个数组，让你求数组的搜索子集。虽然题目给的线性的数据结构数组，然而实际上我们是对树这种非线性数据结构进行搜索。这是因为这道题对应的**状态空间是非线性的**。

对于搜索问题，我们核心关注的信息有哪些？又该如何计算呢？这也是搜索篇核心关注的。而市面上很多资料讲述的不是很详细。搜索的核心需要关注的指标有很多，比如树的深度，图的 DFS 序，图中两点间的距离等等。**这些指标都是完成高级算法必不可少的，而这些指标可以通过一些经典算法来实现**。这也是为什么我一直强调**一定要先学习好基础的数据结构与算法**的原因。

不过要讲这些讲述完整并非容易，以至于如果完整写完可能需要花很多的时间，因此一直没有动手去写。

另外由于其他数据结构都可以看做是图的特例。因此研究透图的基本思想，就很容易将其扩展到其他数据结构上，比如树。因此我打算围绕图进行讲解，并逐步具象化到其他特殊的数据结构，比如树。

状态空间

结论先行：**状态空间其实就是一个图结构，图中的节点表示状态，图中的边表示状态之前的联系，这种联系就是题目给出的各种关系**。

搜索题目的状态空间通常是非线性的。比如上面提到的例子：求一个数组的子集。这里的状态空间实际上就是数组的各种组合。

对于这道题来说，其状态空间的一种可行的划分方式为：

- 长度为 1 的子集
- 长度为 2 的子集
- 。。。
- 长度为 n 的子集（其中 n 为数组长度）

而如何确定上面所有的子集呢。

一种可行的方案是可以采取类似分治的方式逐一确定。

比如我们可以：

- 先确定某一种子集的第一个数是什么
- 再确定第二个数是什么
- 。。。

如何确定第一个数，第二个数。。。呢?

暴力枚举所有可能就可以了。

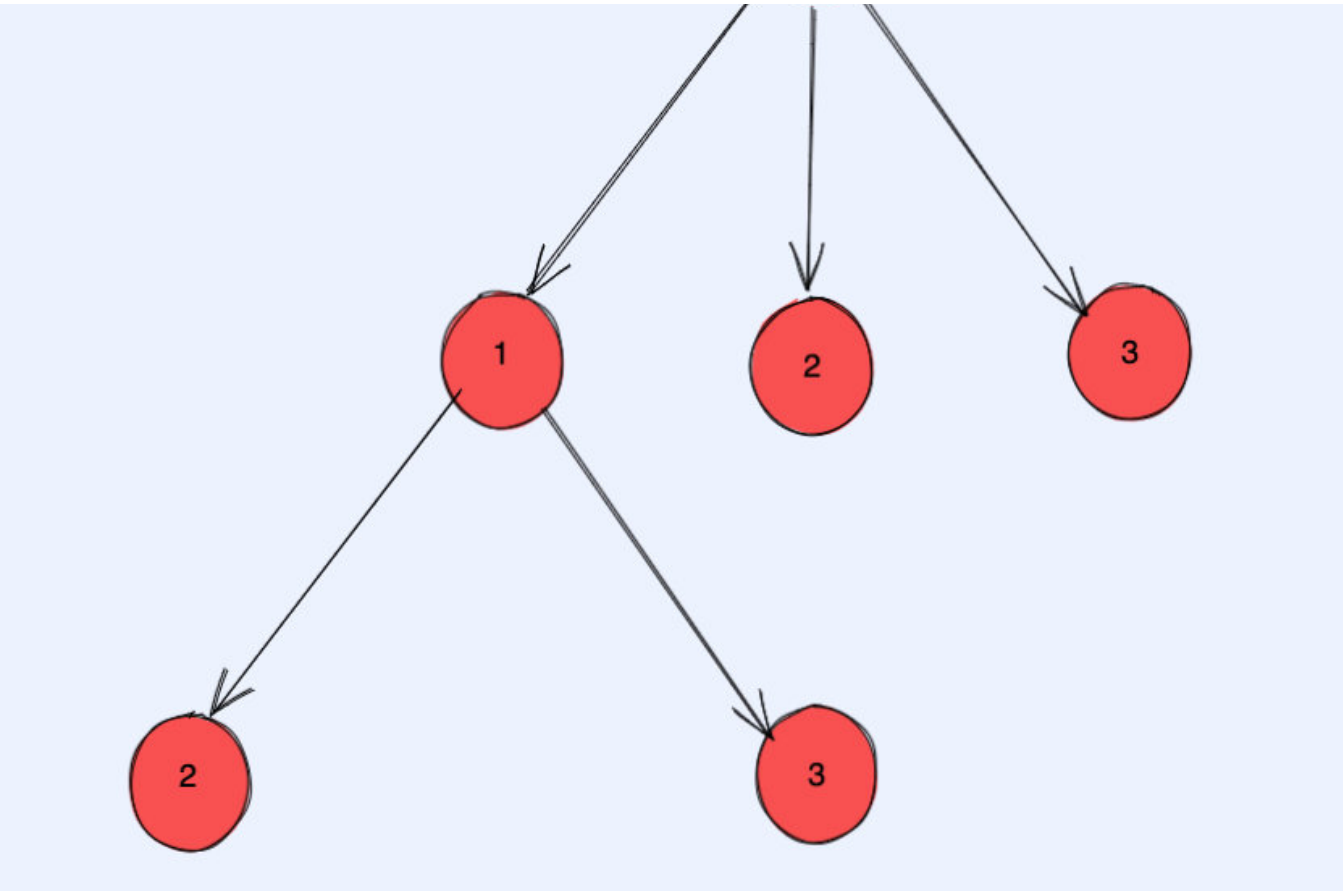
> 这就是搜索问题的核心，其他都是辅助，所以这句话请务必记住。

所谓的暴力枚举所有可能在这里就是尝试数组中所有可能的数字。

- 比如第一个数是什么？很明显可能是数组中任意一项。ok，我们就枚举 n 种情况。
- 第二个数呢？很明显可以是除了上面已经被选择的数之外的任意一个数。ok，我们就枚举 n - 1 种情况。

据此，你可以画出如下的决策树。

（下图描述的是对一个长度为 3 的数组进行决策的部分过程，树节点中的数字表示索引。即确定第一个数有三个选择，确定第二个数会根据上次的选择变为剩下的两个选择）



决策过程动图演示：

（电子书无法显示动图，大家可以手动复制到浏览器查看。动图地址：https://pic.stackoverflow.wiki/uploadImages/115/238/39/106/2021/05/27/18/33/b97ee92b-a516-48e1-83d9-b29c1eaf2eff.svg）

一些搜索算法就是基于这个朴素的思想，本质就是模拟这个决策树。这里面其实也有很多有趣的细节，后面我们会对其进行更加详细的讲解。而现在大家只需要对**解空间是什么以及如何对解空间进行遍历有一点概念就行了。** 后面我会继续对这个概念进行加深。

这里大家只要记住**状态空间就是图，构建状态空间就是构建图。如何构建呢？当然是根据题目描述了** 。

DFS 和 BFS

DFS 和 BFS 是搜索的核心，贯穿搜索篇的始终，因此有必要先对其进行讲解。

DFS

DFS 的概念来自于图论，但是搜索中 DFS 和图论中 DFS 还是有一些区别，搜索中 DFS 一般指的是通过递归函数实现暴力枚举。

> 如果不使用递归，也可以使用栈来实现。不过本质上是类似的。

首先将题目的**状态空间映射到一张图，状态就是图中的节点，状态之间的联系就是图中的边**，那么 DFS 就是在这种图上进行**深度优先**的遍历。而 BFS 也是类似，只不过遍历的策略变为了**广度优先**，一层层铺开而已。所以**BFS 和 DFS 只是遍历这个状态图的两种方式罢了，如何构建状态图才是关键**。

本质上，对上面的图进行遍历的话会生成一颗**搜索树**。为了**避免重复访问，我们需要记录已经访问过的节点**。这些是**所有的搜索算法共有的**，后面不再赘述。

如果你是在树上进行遍历，是不会有环的，也自然不需要为了**避免环的产生记录已经访问的节点**，这是因为树本质上是一个简单无环图。

算法流程

1. 首先将根节点放入**stack**中。
2. 从*stack*中取出第一个节点，并检验它是否为目标。如果找到目标，则结束搜寻并回传结果。否则将它某一个尚未检验过的直接子节点加入**stack**中。
3. 重复步骤 2。
4. 如果不存在未检测过的直接子节点。将上一级节点加入**stack**中。重复步骤 2。
5. 重复步骤 4。
6. 若**stack**为空，表示整张图都检查过了——亦即图中没有欲搜寻的目标。结束搜寻并回传“找不到目标”。

> 这里的 stack 可以理解为自实现的栈，也可以理解为调用栈

算法模板

下面我们借助递归来完成 DFS。

```
const visited = {}
function dfs(i) {
 if (满足特定条件) {
  // 返回结果 or 退出搜索空间
 }

 visited[i] = true // 将当前状态标为已搜索
 for (根据i能到达的下个状态j) {
  if (!visited[j]) { // 如果状态j没有被搜索过
   dfs(j)
  }
 }
}
```

常用技巧

前序遍历与后序遍历

DFS 常见的形式有**前序和后序**。二者的使用场景也是截然不同的。

上面讲述了搜索本质就是在状态空间进行遍历，空间中的状态可以抽象为图中的点。那么如果搜索过程中，当前点的结果需要依赖其他节点（大多数情况都会有依赖），那么遍历顺序就变得重要。

比如当前节点需要依赖其子节点的计算信息，那么使用后序遍历自底向上递推就显得必要了。而如果当前节点需要依赖其父节点的信息，那么使用先序遍历进行自顶向下的递归就不难想到。

比如下文要讲的计算树的深度。由于树的深度的递归公式为：$f(x) = f(y) + 1$。其中 f(x) 表示节点 x 的深度，并且 x 是 y 的子节点。很明显这个递推公式的 base case 就是根节点深度为一，通过这个 base case 我们可以递推求出树中任意节点的深度。显然，使用先序遍历自顶向下的方式统计是简单而又直接的。

再比如下文要讲的计算树的子节点个数。由于树的子节点递归公式为：$f(x) = sum_{i=0}^{n} f(a_i)$ 其中 x 为树中的某一个节点，a_i 为树中节点的子节点。而 base case 则是没有任何子节点(也就是叶子节点)，此时 $f(x) = 1$。 因此我们可以利用后序遍历自底向上来完成子节点个数的统计。

关于从递推关系分析使用何种遍历方法， 我在《91 天学算法》中的基础篇中的《模拟，枚举与递推》子专题中对此进行了详细的描述。91 学员可以直接进行查看。关于树的各种遍历方法，我在树专题中进行了详细的介绍。

迭代加深

迭代加深本质上是一种可行性的剪枝。关于剪枝，我会在后面的《回溯与剪枝》部分做更多的介绍。

所谓迭代加深指的是**在递归树比较深的时候，通过设定递归深度阈值，超过阈值就退出的方式主动减少递归深度的优化手段。这种算法成立的前提是题目中告诉我们答案不超过 xxx**，这样我们可以将 xxx 作为递归深度阈值，这样不仅不会错过正确解，还能在极端情况下有效减少不必须的运算。

具体地，我们可以使用自顶向下的方式记录递归树的层次，和上面介绍如何计算树深度的方法是一样的。接下来在主逻辑前增加**当前层次是否超过阈值**的判断即可。

主代码：

```
MAX_LEVEL = 20
def dfs(root, level):
    if level > MAX_LEVEL: return
    # 主逻辑
dfs(root, 0)
```

这种技巧在实际使用中并不常见，不过在某些时候能发挥意想不到的作用。

双向搜索

有时候问题规模很大，直接搜索会超时。此时可以考虑从起点搜索到问题规模的一半。然后将此过程中产生的状态存起来。接下来目标转化为在存储的中间状态中寻找满足条件的状态。进而达到降低时间复杂度的效果。

上面的说法可能不太容易理解。 接下来通过一个例子帮助大家理解。

题目地址

https://leetcode-cn.com/problems/closest-subsequence-sum/

题目描述

```
给你一个整数数组 nums 和一个目标值 goal 。

你需要从 nums 中选出一个子序列，使子序列元素总和最接近 goal 。也就是说，如果子序列元素和为 sum ，你需要 最小化绝对差 abs(sum - goal) 。

返回 abs(sum - goal) 可能的 最小值 。

注意，数组的子序列是通过移除原始数组中的某些元素（可能全部或无）而形成的数组。



示例 1：

输入：nums = [5,-7,3,5], goal = 6
输出：0
解释：选择整个数组作为选出的子序列，元素和为 6 。
子序列和与目标值相等，所以绝对差为 0 。
示例 2：

输入：nums = [7,-9,15,-2], goal = -5
输出：1
解释：选出子序列 [7,-9,-2] ，元素和为 -4 。
绝对差为 abs(-4 - (-5)) = abs(1) = 1 ，是可能的最小值。
示例 3：

输入：nums = [1,2,3], goal = -7
输出：7


提示：

1 <= nums.length <= 40
-10^7 <= nums[i] <= 10^7
-10^9 <= goal <= 10^9
```

思路

从数据范围可以看出，这道题大概率是一个 $O(2^m)$ 时间复杂度的解法，其中 m 是 nums.length 的一半。

为什么？首先如果题目数组长度限制为小于等于 20，那么大概率是一个 $O(2^n)$ 的解法。

> 如果这个也不知道，建议看一下这篇文章 https://lucifer.ren/blog/2020/12/21/shuati-silu3/ 另外我的刷题插件 leetcode-cheatsheet 也给出了时间复杂度速查表供大家参考。

将 40 砍半恰好就可以 AC 了。实际上，40 这个数字就是一个强有力的信号。

回到题目中。我们可以用一个二进制位表示原数组 nums 的一个子集，这样用一个长度为 2^n 的数组就可以描述 nums 的所有子集了，这就是状态压缩。一般题目数据范围是 <= 20 都应该想到。

> 这里 40 折半就是 20 了。

> 如果不熟悉状态压缩，可以看下我的这篇文章 **状压 DP 是什么？这篇题解带你入门**

接下来，我们使用动态规划求出所有的子集和。

令 dp[i] 表示**选择 i 的二进制表示的数**的所有子集和。比如 i 的二进制位 1001，那么 dp[i] 表示选择数组第 1 项和数组第 4 项的所有子集和，其实就是 nums[0] + nums[3]。

具体地，我们可以枚举 1 << i 所有子集 j，并依次考虑**再**选择数组第 i 项。

那么转移方程为： `dp[(1 << i) + j] = dp[j] + A[i]` 。**dp[(1 << i) + j]表示选择数组第 i 项和选择情况如 j 二进制表示的子集和**。

动态规划求子集和代码如下：

```
def combine_sum(A):
    n = len(A)
    dp = [0] * (1 << n)
    for i in range(n):
        for j in range(1 << i):
            dp[(1 << i) + j] = dp[j] + A[i]
    return dp
```

接下来，我们将 nums 平分为两部分，分别计算子集和：

```
n = len(nums)
c1 = combine_sum(nums[: n // 2])
c2 = combine_sum(nums[n // 2 :])
```

其中 c1 就是前半部分数组的所有子集和，c2 就是后半部分的所有子集和。

接下来问题转化为： `在两个数组 c1 和 c2中找两个数，其和最接近 goal，其中 c1 和 c2 分别为数组 nums 前半部分的子集和与数组 nums 后半部分的子集和` 。而这是一个非常经典的双指针问题，逻辑类似两数和。

只不过两数和是一个数组挑两个数，这里是两个数组分别挑一个数罢了。

这里其实只需要一个指针指向一个数组的头，另外一个指向另外一个数组的尾即可。

代码不难写出：

```
def combine_closest(c1, c2, goal):
    # 先排序以便使用双指针
    c1.sort()
    c2.sort()
    ans = float("inf")
    i, j = 0, len(c2) - 1
    while i < len(c1) and j >= 0:
        _sum = c1[i] + c2[j]
        ans = min(ans, abs(_sum - goal))
        if _sum > goal:
            j -= 1
        elif _sum < goal:
            i += 1
        else:
            return 0
    return ans
```

上面这个代码不懂的多看看两数和。

代码

代码支持：Python3

Python3 Code:

```
class Solution:
    def minAbsDifference(self, nums: List[int], goal: int) -> int:
        def combine_sum(A):
            n = len(A)
            dp = [0] * (1 << n)
            for i in range(n):
                for j in range(1 << i):
                    dp[(1 << i) + j] = dp[j] + A[i]
            return dp

        def combine_closest(c1, c2):
            c1.sort()
            c2.sort()
            ans = float("inf")
            i, j = 0, len(c2) - 1
```

```
            while i < len(c1) and j >= 0:
                _sum = c1[i] + c2[j]
                ans = min(ans, abs(_sum - goal))
                if _sum > goal:
                    j -= 1
                elif _sum < goal:
                    i += 1
                else:
                    return 0
            return ans

        n = len(nums)
        return combine_closest(combine_sum(nums[: n // 2]), combine_sum(nums[n // 2 :]))
```

复杂度分析

令 n 为数组长度, m 为 $\frac{n}{2}$。

- 时间复杂度：$O(m * 2^m)$
- 空间复杂度：$O(2^m)$

相关题目推荐：

- **16. 最接近的三数之和**
- **805. 数组的均值分割** 强烈推荐 👍
- **1049. 最后一块石头的重量 II**
- **1774. 最接近目标价格的甜点成本**

这道题和双向搜索有什么关系呢?

回一下开头我的话： 有时候问题规模很大，直接搜索会超时。此时可以考虑从起点搜索到问题规模的一半。然后将此过程中产生的状态存起来。接下来目标转化为在存储的中间状态中寻找满足条件的状态。进而达到降低时间复杂度的效果。

对应这道题，我们如果直接暴力搜索。那就是枚举所有子集和，然后找到和 goal 最接近的，思路简单直接。可是这样会超时，那么就搜索到一半， 然后将状态存起来（对应这道题就是存到了 dp 数组）。接下来问题转化为两个 dp 数组的运算。**该算法，本质上是将位于指数位的常数项挪动到了系数位**。这是一种常见的双向搜索，我姑且称为 DFS 的双向搜索。目的是为了和后面的 BFS 双向搜索进行区分。

还有一种题目，让你找中间点。比如 **The Meeting Place** 这道题让你求一个点，使得所有人到这个点的距离和最短。 类似于双向搜索，我们的搜索是从很多方向的， 最终碰到一个**已经访问过的点**，我们可以将其看成是集合点备胎，并将距离和更新上去。这样最后扫描一次集合点备胎集合就可以找到距离和最短的集合点了。

BFS

BFS 也是图论中算法的一种。不同于 DFS， BFS 采用横向搜索的方式，从初始状态一层层展开直到目标状态，在数据结构上通常采用队列结构。

具体地，我们不断从队头取出状态，然后**将此状态对应的决策产生的所有新的状态推入队尾**，重复以上过程直至队列为空即可。

注意这里有两个关键点：

1. 将此状态对应的决策。 实际上这句话指的就是状态空间中的图的边，而不管是 DFS 和 BFS 边都是确定的。也就是说不管是 DFS 还是 BFS 这个决策都是一样的。不同的是什么？不同的是进行决策的方向不同。

2. 所有新的状态推入队尾。上面说 BFS 和 DFS 是进行决策的方向不同。这就可以通过这个动作体现出来。由于直接将所有**状态空间中的当前点的邻边**放到队尾。由队列的先进先出的特性，当前点的邻边访问完成之前是不会继续向外扩展的。这一点大家可以和 DFS 进行对比。

最简单的 BFS 每次扩展新的状态就增加一步，通过这样一步步逼近答案。其实也就等价于在一个权值为 1 的图上进行 BFS。由于队列的**单调性和二值性**，当第一次取出目标状态时就是最少的步数。基于这个特性，BFS 适合求解一些**最少操作**的题目。

> 关于单调性和二值性，我会在后面的 BFS 和 DFS 的对比那块进行讲解。

前面 DFS 部分提到了**不管是什么搜索都需要记录和维护状态，其中一个就是节点访问状态以防止环的产生**。而 BFS 中我们常常用来求点的最短距离。值得注意的是，有时候我们会使用一个哈希表 dist 来记录从源点到图中其他点的距离。这个 dist 也可以充当**防止环产生的功能**，这是因为第一次到达一个点后**再次到达此点的距离一定比第一次到达大**，利用这点就可知道是否是第一次访问了。

算法流程

1. 首先将根节点放入队列中。

2. 从队列中取出第一个节点，并检验它是否为目标。

   - 如果找到目标，则结束搜索并回传结果。
   - 否则将它所有尚未检验过的直接子节点加入队列中。

3. 若队列为空，表示整张图都检查过了——亦即图中没有欲搜索的目标。结束搜索并回传“找不到目标”。

4. 重复步骤 2。

算法模板

```
const visited = {}
function bfs() {
```

```
 let q = new Queue()
 q.push(初始状态)
 while(q.length) {
  let i = q.pop()
  if (visited[i]) continue
  for (i的可抵达状态j) {
   if (j 合法) {
    q.push(j)
   }
  }
 }
 // 找到所有合法解
}
```

常用技巧

双向搜索

题目地址(126. 单词接龙 II)

https://leetcode-cn.com/problems/word-ladder-ii/

题目描述

```
按字典 wordList 完成从单词 beginWord 到单词 endWord 转化，一个表示此过程的 转换序列 是形式上像 beginWord -> s1 -> s2 -> ... -> sk 

每对相邻的单词之间仅有单个字母不同。
转换过程中的每个单词 si（1 <= i <= k）必须是字典 wordList 中的单词。注意，beginWord 不必是字典 wordList 中的单词。
sk == endWord

给你两个单词 beginWord 和 endWord ，以及一个字典 wordList 。请你找出并返回所有从 beginWord 到 endWord 的 最短转换序列 ，如果不存在这样



示例 1：

输入：beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
输出：[["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]
解释：存在 2 种最短的转换序列：
"hit" -> "hot" -> "dot" -> "dog" -> "cog"
"hit" -> "hot" -> "lot" -> "log" -> "cog"


示例 2：

输入：beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
输出：[]
解释：endWord "cog" 不在字典 wordList 中，所以不存在符合要求的转换序列。
```

```
提示：

1 <= beginWord.length <= 7
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 中的所有单词 互不相同
```

思路

这道题就是我们日常玩的**成语接龙游戏**。即让你从 beginWord 开始， 接龙的 endWord。让你找到**最短**的接龙方式，如果有多个，则**全部返回**。

不同于成语接龙的字首接字尾。这种接龙需要的是**下一个单词和上一个单词**仅有一个单词不同。

我们可以对问题进行抽象：**即构建一个大小为 n 的图，图中的每一个点表示一个单词，我们的目标是找到一条从节点 beginWord 到节点 endWord 的一条最短路径。**

这是一个不折不扣的图上 BFS 的题目。套用上面的解题模板可以轻松解决。唯一需要注意的是**如何构建图**。更进一步说就是**如何构建边**。

由题目信息的转换规则：**每对相邻的单词之间仅有单个字母不同**。不难知道，如果两个单词的仅有单个字母不同 ，就**说明两者之间有一条边。**

明白了这一点，我们就可以构建邻接矩阵了。

核心代码：

```
neighbors = collections.defaultdict(list)
for word in wordList:
    for i in range(len(word)):
        neighbors[word[:i] + "*" + word[i + 1 :]].append(word)
```

构建好了图。 BFS 剩下要做的就是明确起点和终点就好了。对于这道题来说，起点是 beginWord，终点是 endWord。

那我们就可以将 beginWord 入队。不断在图上做 BFS，直到第一次遇到 endWord 就好了。

套用上面的 BFS 模板，不难写出如下代码:

> 这里我用了 cost 而不是 visitd，目的是为了让大家见识多种写法。下面的优化解法会使用 visited 来记录。

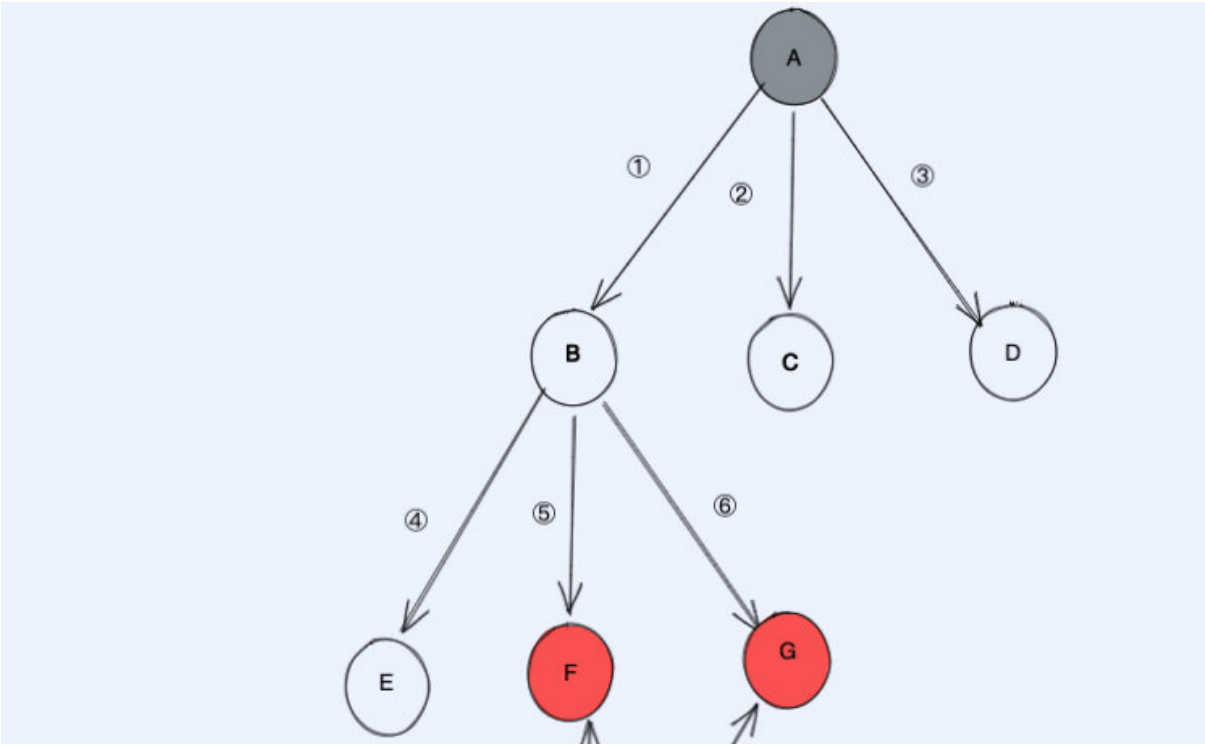
```
class Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) -> List[List[str]]:
```

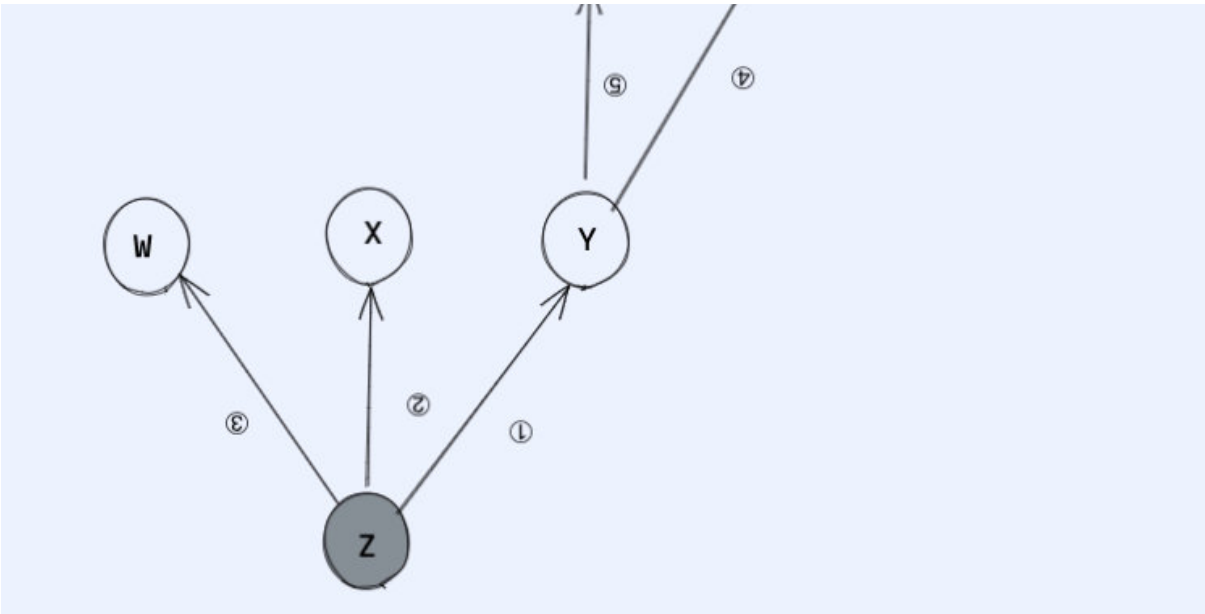
```
        cost = collections.defaultdict(lambda: float("inf"))
        cost[beginWord] = 0
        neighbors = collections.defaultdict(list)
        ans = []

        for word in wordList:
            for i in range(len(word)):
                neighbors[word[:i] + "*" + word[i + 1 :]].append(word)
        q = collections.deque([[beginWord]])

        while q:
            path = q.popleft()
            cur = path[-1]
            if cur == endWord:
                ans.append(path.copy())
            else:
                for i in range(len(cur)):
                    for neighbor in neighbors[cur[:i] + "*" + cur[i + 1 :]]:
                        if cost[cur] + 1 <= cost[neighbor]:
                            q.append(path + [neighbor])
                            cost[neighbor] = cost[cur] + 1
        return ans
```

当终点可以逆向搜索的时候，我们也可以尝试双向 BFS。更本质一点就是：**如果你构建的状态空间的边是双向的，那么就可以使用双向 BFS。**

和 DFS 的双向搜索思想是类似的。我们只需要使用两个队列分别存储中起点和终点进行扩展的节点（我称其为起点集与终点集）即可。当起点和终点在某一时刻交汇了，说明找到了一个从起点到终点的路径，其路径长度就是两个队列扩展的路径长度和。

以上就是双向搜索的大体思路。用图来表示就是这样的：





如上图，我们从起点和重点（A 和 Z）分别开始搜索，如果起点的扩展状态和终点的扩展状态重叠（本质上就是队列中的元素重叠了），那么我们就知道了一个从节点到终点的最短路径。

动图演示：

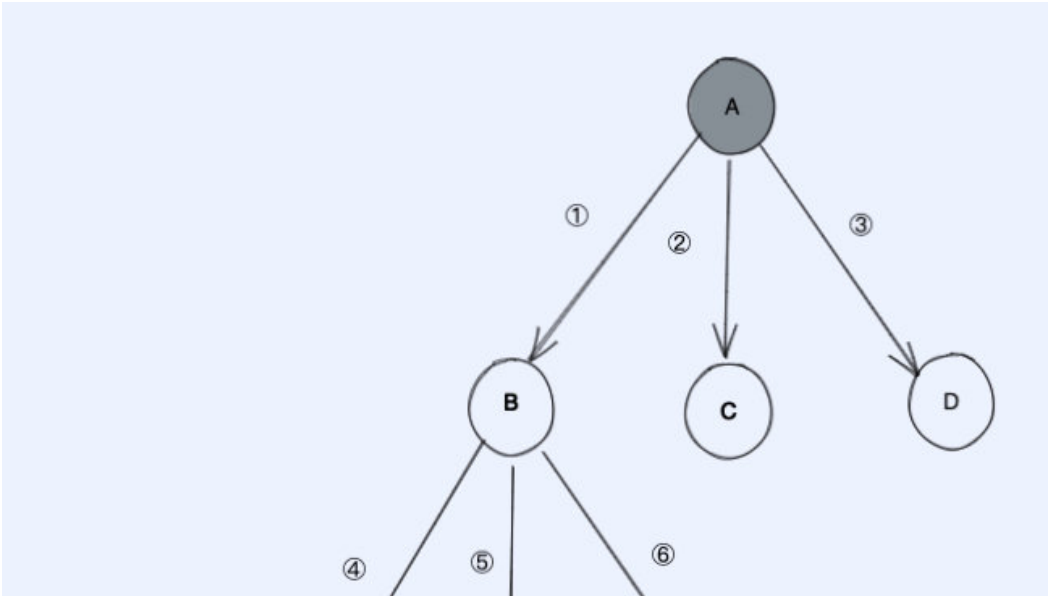
（电子书无法显示动图，大家可以手动复制到浏览器查看。动图地址：https://pic.stackoverflow.wiki/uploadImages/115/238/39/106/2021/05/31/17/41/ab3959a8-ebc2-4772-9f04-390f5cac675b.svg）

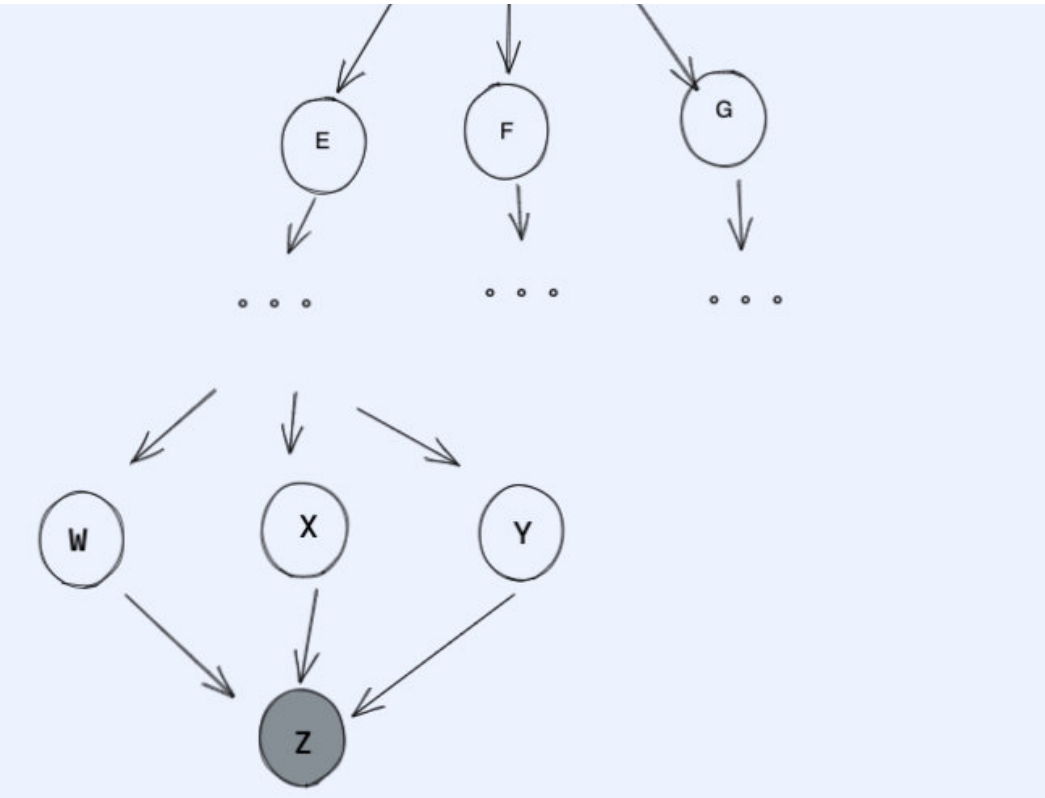
看到这里有必要暂停一下插几句话。

为什么双向搜索就快了？什么情况都会更快么？那为什么不都用双向搜索？有哪些使用条件？

我们一个个回答。

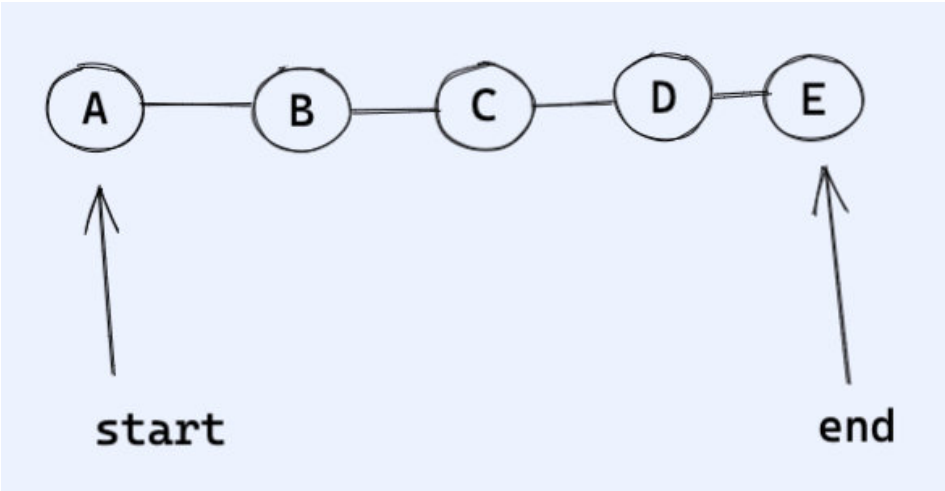
- 为什么双向搜索更快了？通过上面的图我们发现通常刚开始的时候边比较少，队列中的数据也比较少。而随着搜索的进行，**搜索树越来越大， 队列中的节点随之增多**。和上面双向搜索类似，这种增长速度很多情况下是指数级别的，而双向搜索**可以将指数的常系数移动到多项式系数**。如果不使用双向搜索那么搜索树大概是这样的：





可以看出搜索树大了很多，以至于很多点我都画不下，只好用 ”。。。“ 来表示。

- 什么情况下更快？相比于单向搜索，双向搜索通常更快。当然也有例外，举个极端的例子，假如从起点到终点只要一条路径，那么无论使用单向搜索还是双向搜索结果都是一样。



如图使用单向搜索还是双向搜索都是一样的。

- 为什么不都用双向搜索？实际上做题中，我建议大家尽量使用单向搜索，因为写起来更节点，并且大多数都可以通过所有的测试用例。除非你预估到可能会超时或者提交后发现超时的时候再尝试使用双向搜索。

- 有哪些使用条件？正如前面所说：”终点可以逆向搜索的时候，可以尝试双向 BFS。更本质一点就是：**如果你构建的状态空间的边是双向的，那么就可以使用双向 BFS。**“

让我们继续回到这道题。为了能够判断两者是否交汇，我们可以使用两个 hashSet 分别存储起点集合终点集。当一个节点既出现起点集又出现在终点集，那就说明出现了交汇。

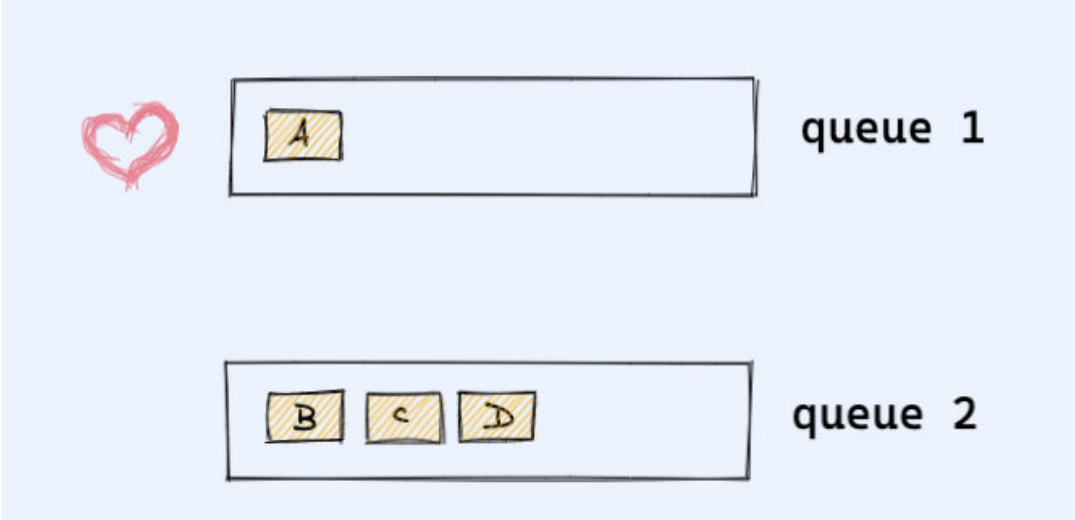
为了节省代码量以及空间消耗，我没有使用上面的队列，而是直接使用了哈希表来代替队列。这种做法可行的关键仍然是上面提到的**队列的二值性和单调性**。

由于**新一轮的出队列前**，队列中的权值都是相同的。因此从左到右遍历或者从右到左遍历，甚至是任意顺序遍历都是无所谓的。（很多题都无所谓）因此使用哈希表而不是队列也是可以的。这点需要引起大家的注意。希望大家对 BFS 的本质有更深的理解。

那我们是不是不需要队列，就用哈希表，哈希集合啥的存就行了？非也！我会在双端队列部分为大家揭晓。

这道题的具体算法：

- 定义两个队列：q1 和 q2 ，分别从起点和终点进行搜索。
- 构建邻接矩阵
- 每次都尝试从 q1 和 q2 中的较小的进行扩展。这样可以达到剪枝的效果。



- 如果 q1 和 q2 交汇了，则将两者的路径拼接起来即可。

代码

- 语言支持：Python3

Python3 Code:

```python
class Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: list) -> list:
        #  剪枝 1
        if endWord not in wordList: return []
        ans = []
        visited = set()
        q1, q2 = {beginWord: [[beginWord]]}, {endWord: [[endWord]]}
        steps = 0
        # 预处理，空间换时间
```

```
        neighbors = collections.defaultdict(list)
        for word in wordList:
            for i in range(len(word)):
                neighbors[word[:i] + "*" + word[i + 1 :]].append(word)
        while q1:
            # 剪枝 2
            if len(q1) > len(q2):
                q1, q2 = q2, q1
            nxt = collections.defaultdict(list)
            for _ in range(len(q1)):
                word, paths = q1.popitem()
                visited.add(word)
                for i in range(len(word)):
                    for neighbor in neighbors[word[:i] + "*" + word[i + 1 :]]:
                        if neighbor in q2:
                            # 从 beginWord 扩展过来的
                            if paths[0][0] == beginWord:
                                ans += [path1 + path2[::-1] for path1 in paths for path2 in q2[neighbor]]
                            # 从 endWord 扩展过来的
                            else:
                                ans += [path2 + path1[::-1] for path1 in paths for path2 in q2[neighbor]]
                        if neighbor in wordList and neighbor not in visited:
                            nxt[neighbor] += [path + [neighbor] for path in paths]
            steps += 1
            # 剪枝 3
            if ans and steps + 2 > len(ans[0]):
                break
            q1 = nxt
        return ans
```

总结

我想通过这道题给大家传递的知识点很多。分别是：

- 队列不一定非得是常规的队列，也可以是哈希表等。不过某些情况必须是双端队列，这个等会讲双端队列给大家讲。
- 双向 BFS 是只适合双向图。也就是说从终点也往前推。
- 双向 BFS 从较少状态的一端进行扩展可以起到剪枝的效果
- visitd 和 dist/cost 都可以起到记录点访问情况以防止环的产生的作用。不过 dist 的作用更多，相应空间占用也更大。

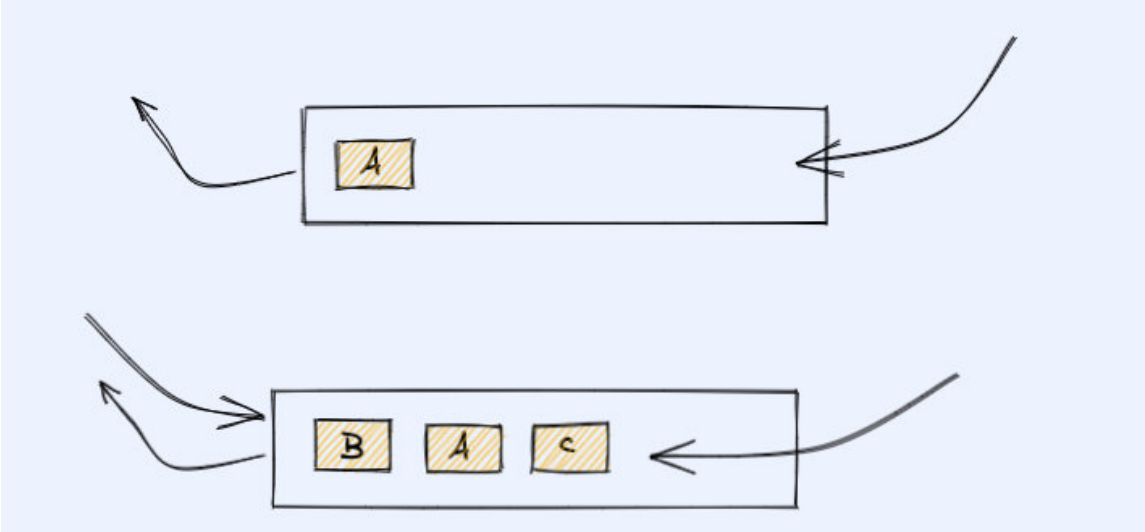
双端队列(01 BFS)

上面提到了 BFS 本质上可以看做是在一个边权值为 1 的图上进行遍历。实际上，我们可以进行一个简单的扩展。如果图中边权值不全是 1，而是 0 和 1 呢？这样其实我们用到双端队列。

双端队列可以在头部和尾部同时进行插入和删除，而普通的队列仅允许在头部删除，在尾部插入。

使用双端队列，当每次取出一个状态的时候。如果我们可以**无代价**的进行转移，那么就可以将其直接放在队头，否则放在队尾。由**前面讲的队列的单调性和二值性**不难得出算法的正确性。而如果状态转移是有代价的，那么就将其放到队尾即可。这也是很多语言提供的内置数据结构是双端队列，而不是队列的原因之一。

如下图：



上面的队列是普通的队列。 而下面的双端队列，可以看出我们在队头插队了一个 B。

动图演示：

（电子书无法显示动图，大家可以手动复制到浏览器查看。动图地址：
https://pic.stackoverflow.wiki/uploadImages/115/238/39/106/2021/05/31/17/07/5d905ba0-c4c4-4bb4-91e9-d2ccb79d435b.svg）

> 思考：如果图对应的权值不出 0 和 1，而是任意正整数呢?

前面我们提到了**是不是不需要队列，就用哈希表，哈希集合啥的存就行了？** 这里为大家揭秘。不可以的。因为哈希表无法处理这里的权值为 0 的情况。

由于这里的 BFS 处理的常见是 0 和 1 的，因此也被称为 01 BFS。

题目推荐：

- 6081. 到达角落需要移除障碍物的最小数目 强烈推荐 👍

会了 01 BFS 那么 AC 这道力扣双周赛的 T4 hard 题就不在话下了！

最后留给大家一个题目当作业 最便宜的公交路线。

DFS 和 BFS 的对比

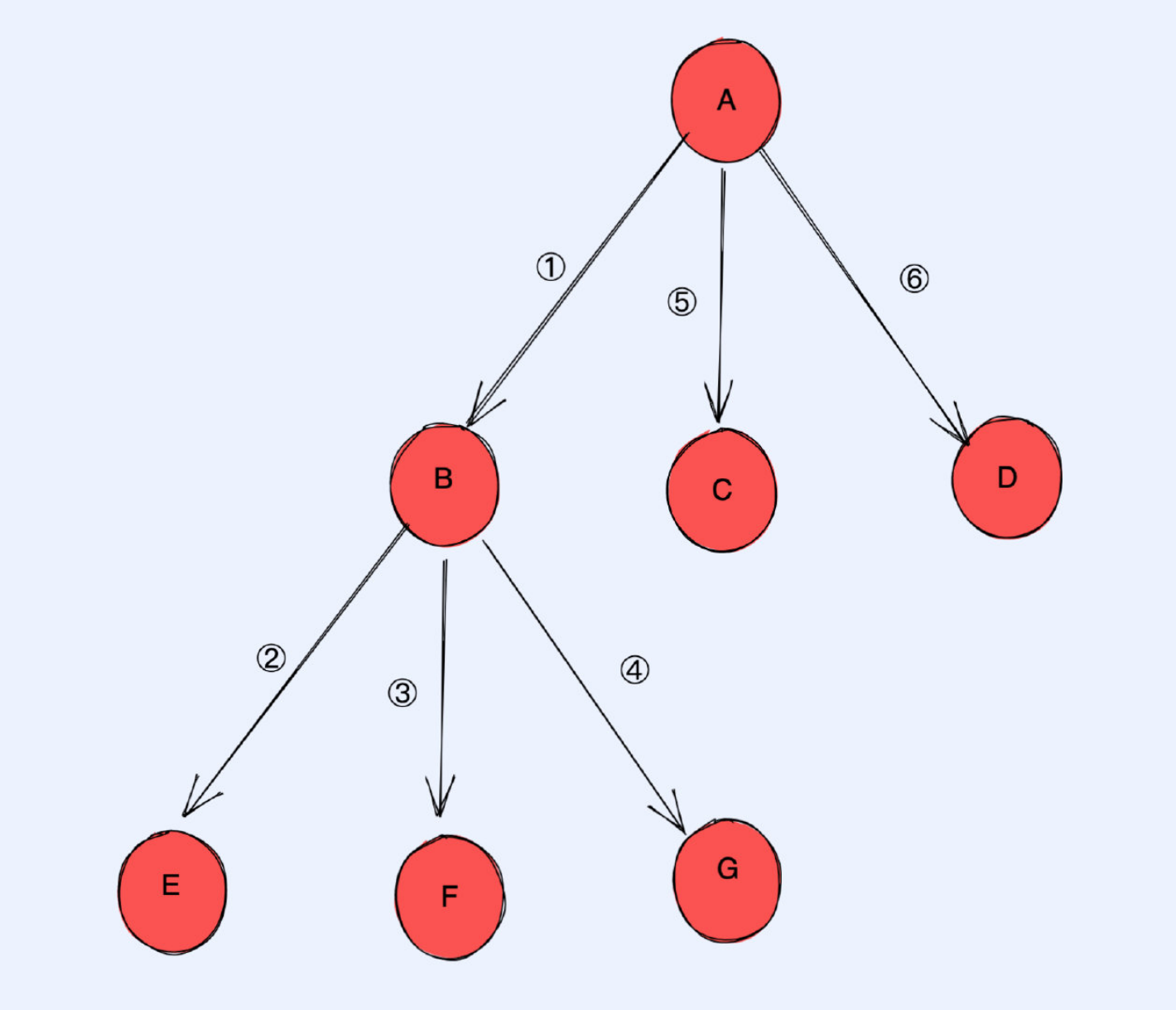
BFS 和 DFS 分别处理什么样的问题？两者究竟有什么样的区别？这些都值得我们认真研究。

简单来说，不管是 DFS 还是 BFS 都是对**题目对应的状态空间进行搜索**。

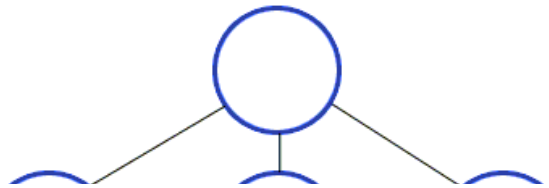
具体来说，二者区别在于：

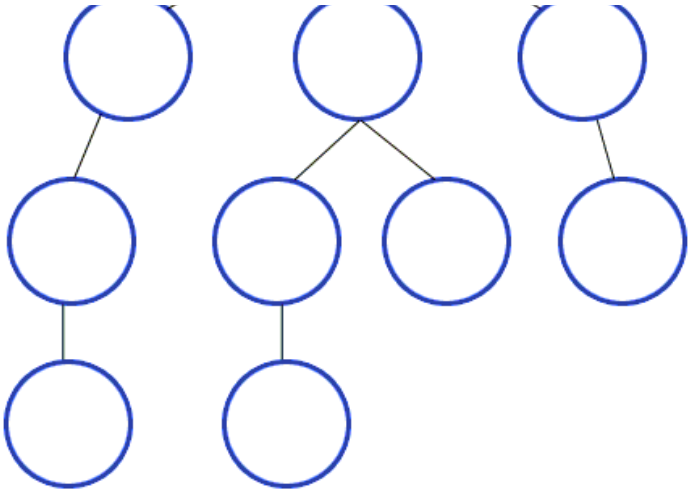
- DFS 在分叉点会任选一条深入进入，遇到终点则返回，再次返回到分叉口后尝试下一个选择。基于此，我们可以在路径上记录一些数据。**由此也可以衍生出很多有趣的东西。**

如下图，我们遍历到 A，有三个选择。此时我们可以任意选择一条，比如选择了 B，程序会继续往下进行选择分支 2，3 。。。



如下动图演示了一个典型的 DFS 流程。后面的章节，我们会给大家带来更复杂的图上 DFS。

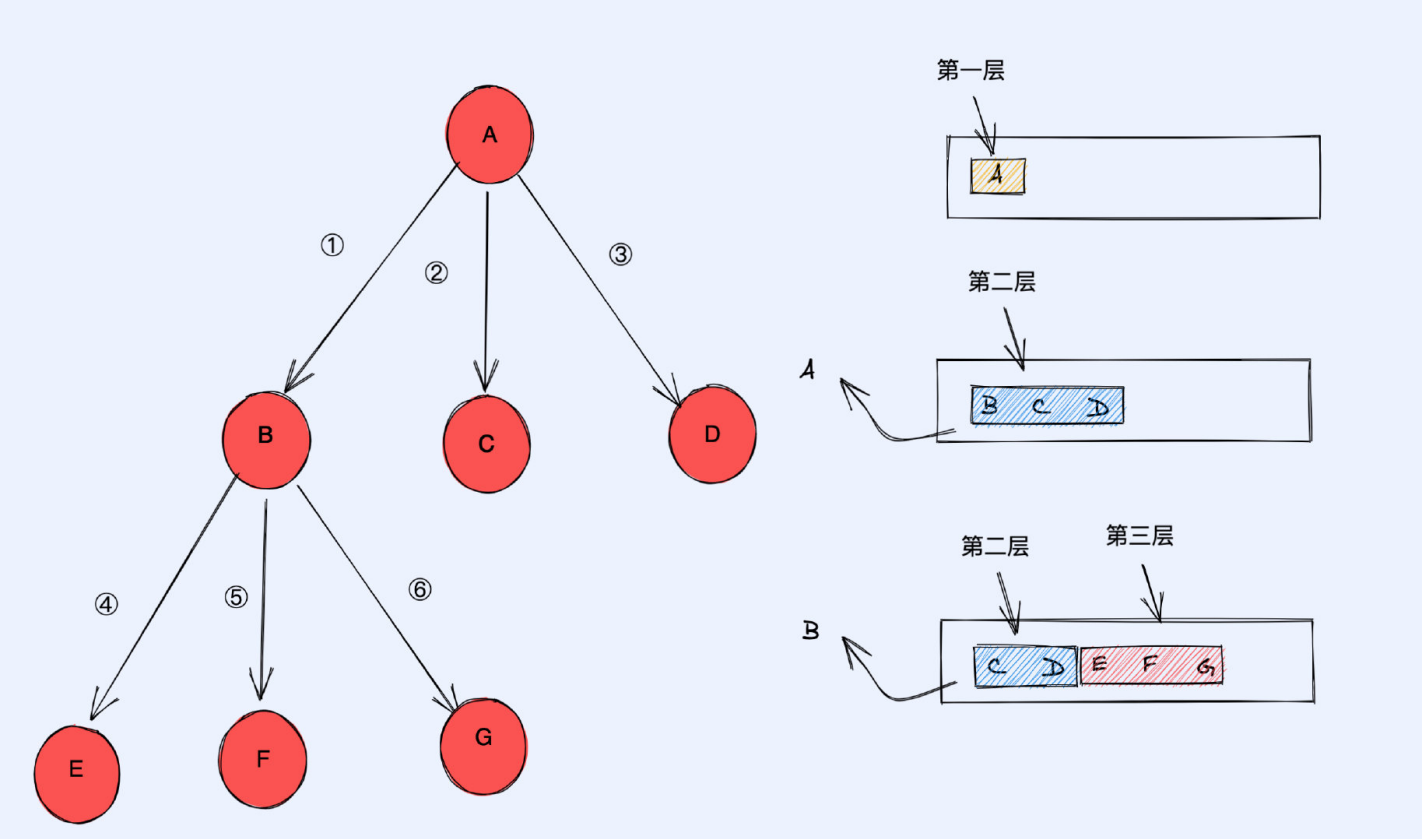




binary-tree-traversal-dfs

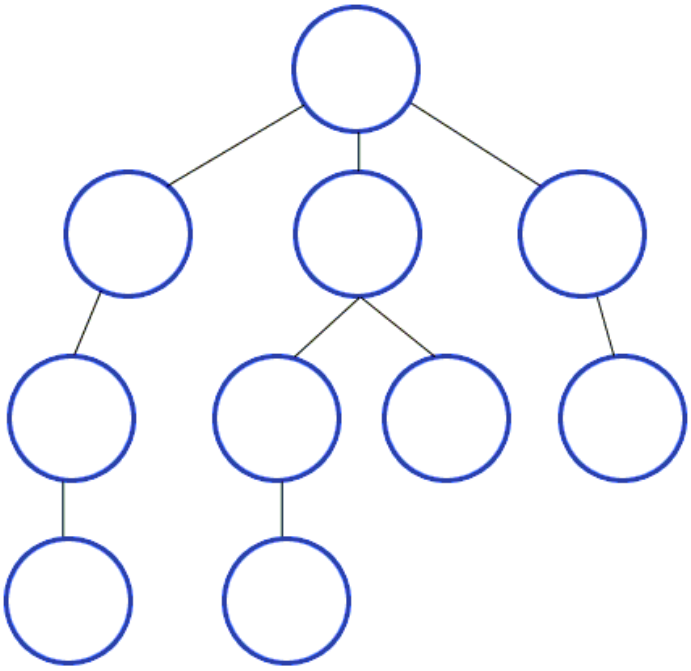
- BFS 在分叉点会选择搜索的路径各尝试一次。使用队列来存储待处理的元素时，队列中**最多**只会有两层的元素，这是**队列的二值性**。另外队列还满足单调性，即相同层的元素在一起。**基于这个特点有很多有趣的优化。**

如下图，广度优先遍历会将搜索的选择全部选择一遍会才会进入到下一层。和上面一样，我给大家标注了程序执行的一种可能的顺序。



可以发现，和我上面说的一样。右侧的队列始终最多有两层的节点，并且相同层的总在一起，换句话说队列的元素在层上**满足单调性**。

如下动图演示了一个典型的 BFS 流程。后面的章节，我们会给大家带来更复杂的图上 BFS。



binary-tree-traversal-bfs

回溯

什么是回溯

回溯就是通俗意义上的暴力枚举所有可能，思维难度不大，难点在于回溯伴随了一点点代码编写技巧。 因此我们要**掌握的仅仅是回溯的编码技巧**。

回溯是 DFS 中的一种技巧。回溯法采用 **试错** 的思想，它尝试分步的去解决一个问题。在分步解决问题的过程中，当它通过尝试发现现有的分步答案不能得到有效的正确的解答的时候，它将 **取消上一步甚至是上几步的计算，再通过其它的可能的分步答再次尝试寻找问题的答案**。

通俗上讲，回溯是一种走不通就回头的算法。

比如题目让你求将 1 - n 这 n 个数字如何排才能满足奇数索引位置的值大于相邻的偶数索引值?

暴力的思路就是：

- 第一个先放 1
- 第二个放 2
- 第三个放 3。发现不对了，2 小于 3。于是我们看能不能放比 2 小的了？不能了，那就退回到第二步放 2 的地方，我们不妨放 3
- 。。。

这其实就是回溯。当然这道题有更好的解法，完全不需要这样暴力回溯。只不过我想告诉你，这就是回溯。

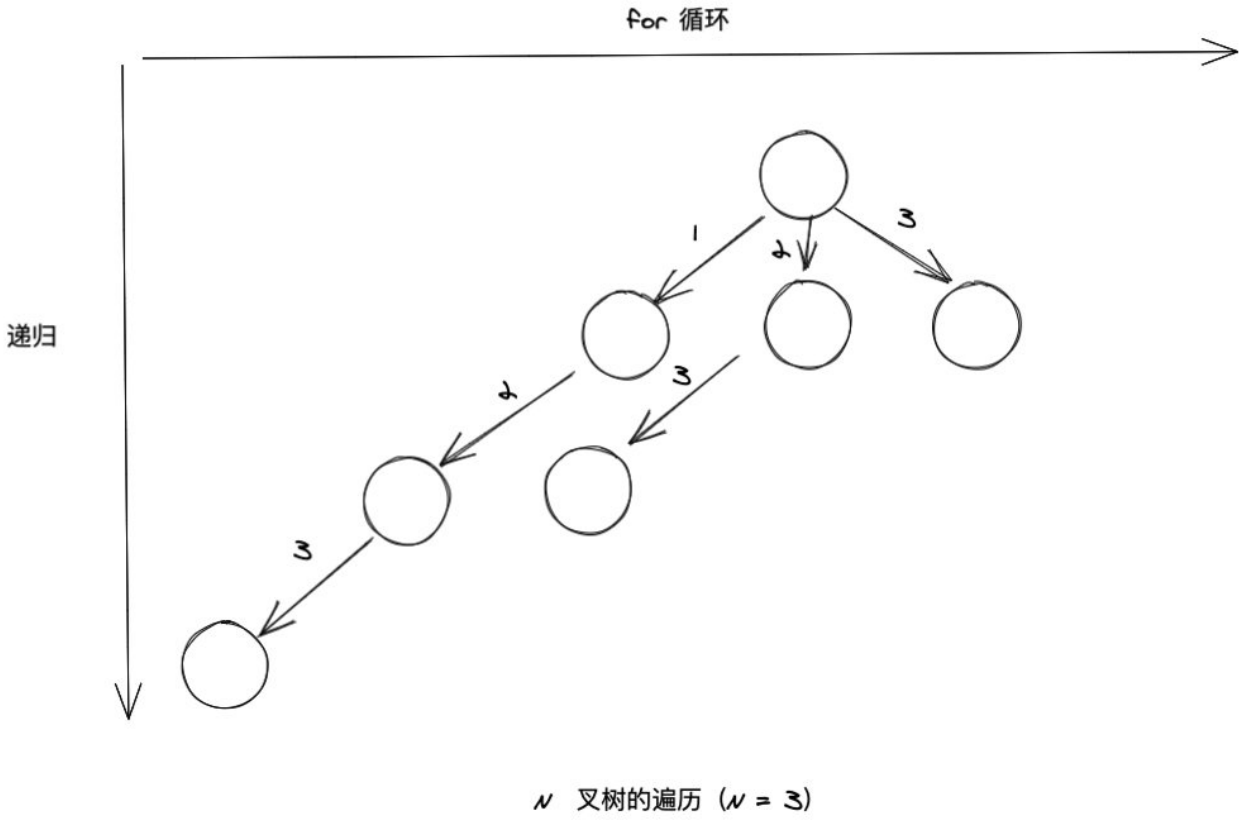
回溯的本质是穷举所有可能，尽管有时候可以通过剪枝去除一些根本不可能是答案的分支，但是从本质上讲，仍然是一种极其暴力的算法。

回溯的思维技巧

回溯的思维技巧就是用决策树（多叉树）的角度思考。

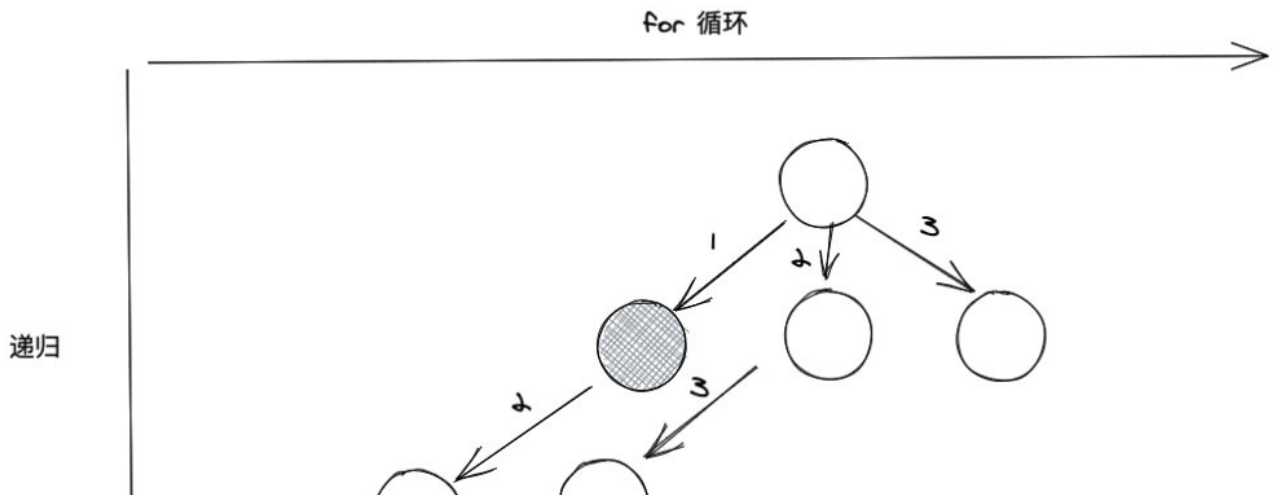
回溯法可以抽象为树形结构，并且是是一颗高度有限的树（N 叉树）。回溯法解决的都是在集合中查找子集，集合的大小就是树的叉树。

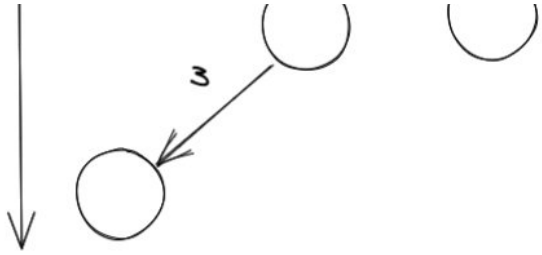
以求数组 [1,2,3] 的子集为例：



N 叉树的遍历（N = 3）

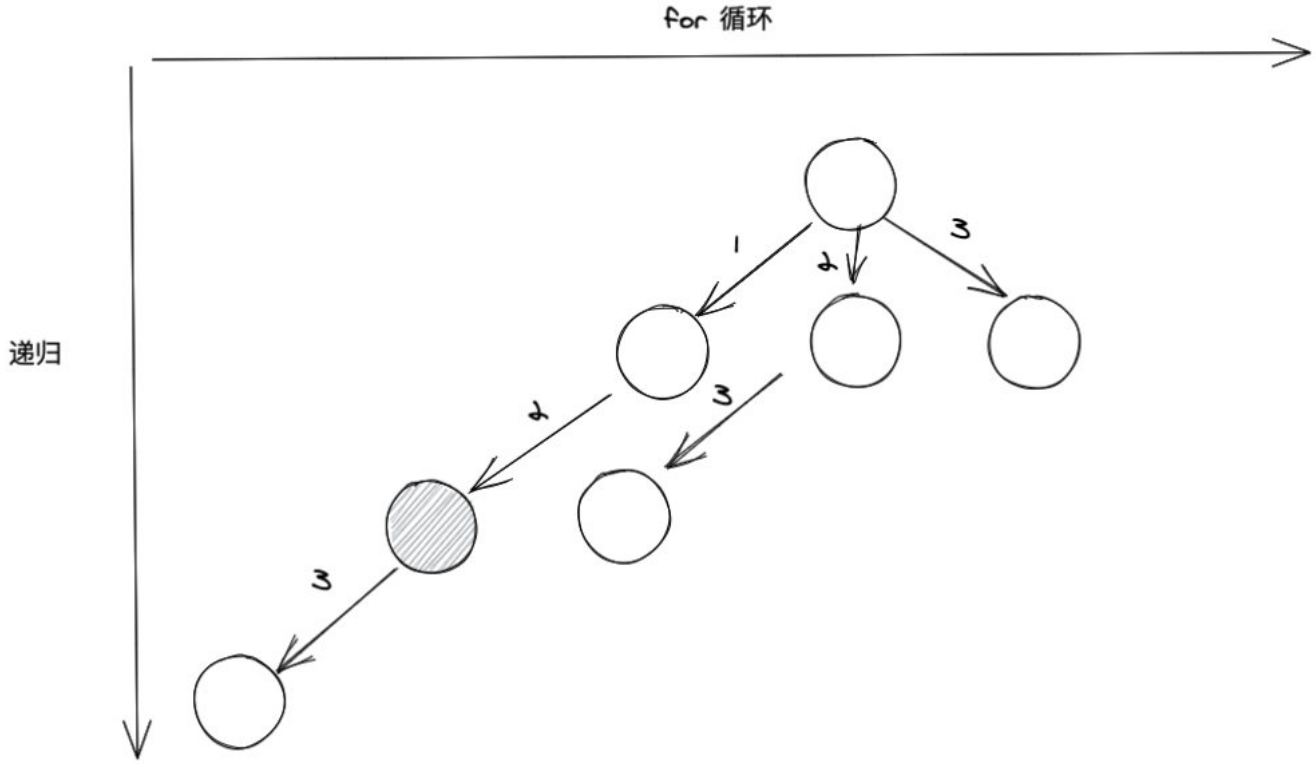
以上图来说， 我们会在每一个节点进行加入到结果集这一次操作。





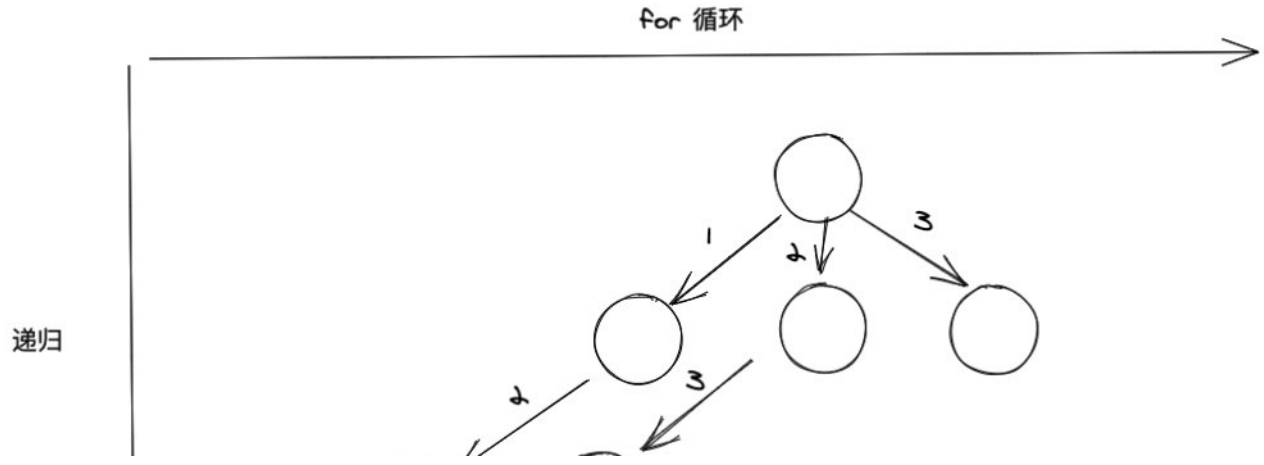
N 叉树的遍历（N = 3）

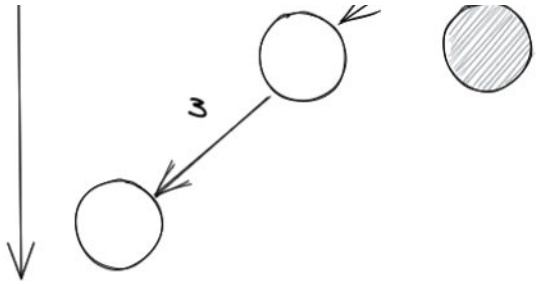
对于上面的灰色节点， 加入结果集就是 [1]。



N 叉树的遍历（N = 3）

这个加入结果集就是 [1,2]。





N 叉树的遍历（$N = 3$）

这个加入结果集就是 [2,3]，以此类推。一共有六个子集，分别是 [1], [1,2], [1,2,3], [2], [2,3] 和 [3]。

而对于全排列问题则会在叶子节点加入到结果集，不过这都是细节问题。掌握了思想之后，大家再去学习细节就会事半功倍。

再比如解数独问题。我们就可以构造一颗 9 叉树，每个叉都是一个决策（放数字 几）。如果遇到不合法的状态就回退状态（可以借助递归的函数返回，回溯状态完善），直到全部放完毕即可。

下面我们来看下回溯问题具体代码怎么写。

算法流程

1. 构造空间树。即每一步我们面临的决策是什么。
2. 进行遍历（递归+遍历）。
3. 如遇到边界条件（特判），即不再向下搜索，转而搜索另一条链。
4. 达到目标条件，输出结果。（终止条件）

算法模板

伪代码：

```
const visited = {}
function dfs(i) {
 if (满足特定条件) {
  // 返回结果 or 退出搜索空间
 }

 visited[i] = true // 将当前状态标为已搜索
 dosomething(i) // 对i做一些操作
 for (根据i能到达的下个状态j) {
  if (!visited[j]) { // 如果状态j没有被搜索过
   dfs(j)
  }
 }
```

```
  undo(i) // 恢复i
}
```

经典题目

老样子，没时间的同学先做强烈推荐的题目。

- **36. 有效的数独**
- **37. 解数独** 与 36 类似，还需要点回溯的思想 强烈推荐 👍
- **39. 组合总和**
- **40. 组合总和 II**
- **46. 全排列**
- **47. 全排列 II** 强烈推荐 👍
- **52. N 皇后 II**
- **78. 子集**
- **90. 子集 II** 强烈推荐 👍
- **113. 路径总和 II**
- **131. 分割回文串**
- **1255. 得分最高的单词集合**

剪枝

回溯题目的另外一个考点是剪枝， 通过恰当地剪枝，可以有效减少时间，比如我通过剪枝操作将**石子游戏 V**的时间从 900 多 ms 优化到了 500 多 ms。

剪枝在每道题的技巧都是不一样的， 不过一个简单的原则就是**避免根本不可能是答案的递归**。

笛卡尔积

一些回溯的题目，我们仍然也可以采用笛卡尔积的方式，将结果保存在返回值而不是路径中，这样就避免了回溯状态，并且由于结果在返回值中，因此可以使用记忆化递归， 进而优化为动态规划形式。

参考题目：

- **140. 单词拆分 II**
- **816. 模糊坐标**

这类问题不同于子集和全排列，其组合是有规律的，我们可以使用笛卡尔积公式，将两个或更多子集联合起来。

常用的指标与统计方法

搜索本质是对状态空间进行遍历求解。而状态空间可以抽象为图，而对于图有很多有趣的指标。那么对于搜索来说，常用的指标有哪些呢？这里给大家介绍。

这些指标是基本的算法。当你实际应用（或者做题）的时候，往往是将几种基础算法进行组合或者稍微改动，因此掌握这些基础算法显得尤为重要。

1. 树的深度与子树大小

计算树的深度和子树大小其实我在前面《前序遍历与后序遍历》部分已经讲过了，只不过当时只是讲了递推公式而已。

这里我直接给出代码。

```
visited = set()
d = collections.defaultdict(int)
def dfs(x):
 visited.add(x) # 将 x 标记为已访问
 # 枚举 x 的所有邻居，并尝试进入
 for neibor in neibors[x]:
  if neibor in visited: continue
  d[neibor] = d[x] + 1 # 递推公式
  dfs(neibor)
}
```

经过上面的处理，d 存储的就是所有的节点对应的深度映射。

很多困难的题都是基于这个简单的算法完成的。比如让你求解一个点到所有**其他点**的距离和如何求？再比如求树的重心如何求？

> 关于树的重心是什么，大家可以搜索一下

2. 图的 DFS 序

树的 dfs 序指的是**dfs 过程中访问节点的顺序**，包括进入和离开，也就是说一个节点会在 dfs 序中出现两次。

为了方便描述，dfs 序用 D 表示。

基于此，有一个有意思的点就是：**假设一个 x 节点在 dfs 序中出现位置为 l 和 r，那么 dfs 序 D 的子序列[l,r]就是以 x 为根的子序列**。

这有什么用？其实基于这样，我们就将逻辑上的非线性结构（树）映射到了线性结构（数组）上，这样我们就可以在数组上对子树进行一些统计。

3. 图的拓扑序

图的拓扑序其实是 BFS 的一个功能。

给一个**无环图**，如果有一个序列满足以下条件，那么 A 就是这个他的拓扑序。

- A 包含图中所有节点
- 对于图中的每一个边 (x, y)，x 在 A 中的位置都在 y 之前（当然也可以反过来）

求解 A 的过程就是拓扑排序。

本质上，图的拓扑序就是使用队列来存储图中入度为 0 的点，**并将其剔除**。由于这些点被剔除了，因此可能会使得图中其他的点入度减少。如果其他的点入度被减少到 0，那么就继续入队重复上面的过程。

对应前文讲的队列的二值性和单调性，这里的队列中只会有度为 0 的节点。因此这个队列其实是单值性，是一种更特殊的 BFS。

代码：

```
# items 为图的所有点
# indegree 为图的入度信息
# neighbors 则是每个点的邻居信息
def tp_sort(self, items, indegree, neighbors):
 q = collections.deque([])
 ans = []
 for item in items:
  if indegree[item] == 0:
   q.append(item)
 while q:
  cur = q.popleft()
  ans.append(cur)

  for neighbor in neighbors[cur]:
   indegree[neighbor] -= 1
   if indegree[neighbor] == 0:
    q.append(neighbor)

 return ans
```

正如前面的算法描述，这里的代码结构是非常简单清晰的。

拓扑排序经常被处理一些有依赖关系的问题。比如排课系统，而课程之前有一定的前置课程，你必须修完前置课程才能学习本可能，问你如何安排才能学完所有的课程等等。

题目推荐：

- **1203. 项目管理**
- **1494. 并行课程 II**

4. 图的联通分量

从图中任意一点出发，我们可以访问到的所有的点可以构成一个联通分量。因此对**图中每一个点进行一次这样的遍历就可以得到若干个联通分量（也可能只有一个，比如说树）**

形象地来说，你可以把图看成是几个分离的水域，图中的点水域的一部分。那么你要如何求水域的个数（不是点的个数）呢?

- 我们可以往某一个点倒墨水，让墨水进行扩散。 如果此时所有水域都被染色了，那么水域个数（联通分量个数）就是 1。
- 否则继续往下一个点倒墨水，重复上面过程直到所有点都被倒了墨水或者所有的水域都被染色。

同时为了防止一个点被多次（因为这是没有意义的），我们需要记录一下每个点被倒墨水的情况。

代码：

```
visited = set()
count = 0
def dfs(x):
 visited.add(x)
 for neibor in neibors:
  if neibor in visited: continue
  dfs(x)
for x in range(n):
 if x not in visited:
  dfs(x)
  count += 1
```

如上代码中的 count 就记录了图中联通分量的个数。力扣中很多题都用到了这个技巧，比如**小岛专题**[1]。

并查集也特别适合处理联通分量个数的计算，大家不妨结合起来理解。

5. 环的检测

环的检测最简单的就是利用 visited 来记录已经访问的节点信息。但是如果要求环的大小等统计信息这就不够了。

> 后面的并查集也可以用来检测是否有环，具体可以参考进阶篇《并查集》的讲义

我们可以用时间戳法计算得出。用一个 dist 表来记录每个节点第一次访问的时间戳 dist[i] = j 表示第一次访问节点 i 的时间戳为 j。初始化时间戳为 0 ， 每次 访问一个节点时间戳增加 1。这样访问到一个已经访问过的节点就可以根据当前的时间戳 t2 和 第一次访问其的时间戳就可得到环大小为 t2 - t1。

总结

以上就是《搜索篇（上）》的所有内容了。总结一下搜索篇的解题思路：

- 根据题目信息构建状态空间（图）。
- 对图进行遍历（BFS 或者 DFS）
- 记录和维护状态。（比如 visited 维护访问情况， 队列和栈维护状态的决策方向等等）

BFS 是面，每一层的节点同时进行搜索。而 DFS 是线，纵向一个一个解决。一般来说找最短路径的时候使用 BFS，其他时候还是 DFS 写起来比较方便。

BFS 一般需要借助于队列这种数据结构，而 BFS 则可以借助递归或者栈来进行。 如果需要求最短距离，推荐大家使用 BFS，这样可大大剪枝，当然最差情况还是和 DFS 复杂度一致。对于不是求最短距离的情况大家随意，使用 BFS 和 DFS 均可。。

战略上，我们可以将二叉树遍历，多叉树遍历都看成图的特殊情况，因此建议大家从图的遍历入手来学习。战术上，我推荐大家从数组，链表遍历开始，进而到二叉树，多叉树遍历，最后学习图的遍历。

我们花了大量的篇幅对 BFS 和 DFS 进行了详细的讲解，包括两个的对比。

核心点需要大家注意：

- DFS 通常都是有递推关系的，而递归关系就是图的边。根据递归关系大家可以选择使用前序遍历或者后序遍历。
- BFS 由于其单调性，因此适合求解最短距离问题。
- 。。。

双向搜索的本质是将复杂度的常数项从一个影响较大的位置（比如指数位）移到了影响较小的位置（比如系数位）。

回溯的本质就是暴力枚举所有可能。要注意的是，由于回溯通常结果集都记录在回溯树的路径上，因此如果不进行撤销操作，则可能在回溯后状态不正确导致结果有差异， 因此需要在递归到底部往上冒泡的时候进行撤销状态。**回溯我建议大家画决策图图进行学习。**

上面提到的搜索都是单向搜索。还有一种比较少见的搜索方式是双向搜索。即分别从起点和中点进行搜索，这样时间复杂度大概可以大概会降低一半左右。有时候暴力搜索刚好超时的时候，可以考虑使用双向搜索。比如如下题目：

```
Given a set of n integers where n <= 40. Each of them is at most 1012, determine the maximum sum subset having sum less

Example:

Input  : set[] = {45, 34, 4, 12, 5, 2} and S = 42
Output : 41
Maximum possible subset sum is 41 which can be
```

```
obtained by summing 34, 5 and 2.

Input  : Set[] = {3, 34, 4, 12, 5, 2} and S = 10
Output : 10
Maximum possible subset sum is 10 which can be
obtained by summing 2, 3 and 5.
```

使用暴力回溯的时间复杂度为 $O(2^n)$。我们也可以：

- 将数组一份为二，并对每一部分分别进行暴力回溯，时间复杂度为 $O(2^{n/2})$
- 不妨设两个暴力回溯的解集为 S1 和 S2。迭代 S1 ，对于 S1 的每一项 x 在 S2 中寻找 y，使得 x + y <= S。
- 为了使上一步更有效率，可以对 S2 进行一次排序，进而使用二分找到 y。

这种算法的时间复杂度是 $O(2^{n/2} * n)$

这种算法面试的考察频率相对比较低，大家根据自己的实际情况选择性掌握即可。

搜索篇知识点比较密集，希望大家多多总结复习。另外搜索篇往往有一些小技巧。这种技巧非常多，不方便罗列。我的建议大家先掌握搜索的基本思想和核心框架，然后通过做题去**消化**黑科技小技巧。比如 **LCS 03. 主题空间** 这道题就用了一点小技巧，类似的有很多，大家边做边体会。

另外一个很有用的小技巧是看到题目中有最小字样就可以考虑使用 bfs，因为 bfs 特别适合处理这种问题，比如最近周赛的一道题 **2059. 转化数字的最小运算数**。题目要求你返回**返回将 x = start 转化为 goal 的最小操作数**，你应该立马想到 bfs，先不管能不能做出来，这个敏感度需要有！

扩展阅读

- 小岛问题
- 深度优先遍历
- 回溯算法

参考资料

[1] 小岛专题: *https://github.com/azl397985856/leetcode/blob/master/thinkings/island.md*

上一页 下一页