Assignment One- Programming

1951976 李林飞

2021 年 4 月 11 日

1 BFS

1.1 核心思想

对于一个 N 位数，要求相邻位数上的数之差的绝对值为 K。采用 BFS 算法的思想，一层一层递进求解，每一层得到一个位上的可行解。其求解过程可先假设最高位数分别为 $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$, 设为 d(注意最高位不能为 0)。接着计算其次高位上的数字，由于次高位与最高位上两数之差的绝对值为 K，因此，如果 $d - K \geq 0$ 或者 $d + K \leq 9$, 则其结果即为此高位上的数字。依次类推，则可以得到最终结果。简单示例图如图 1(# 表示已无解)：

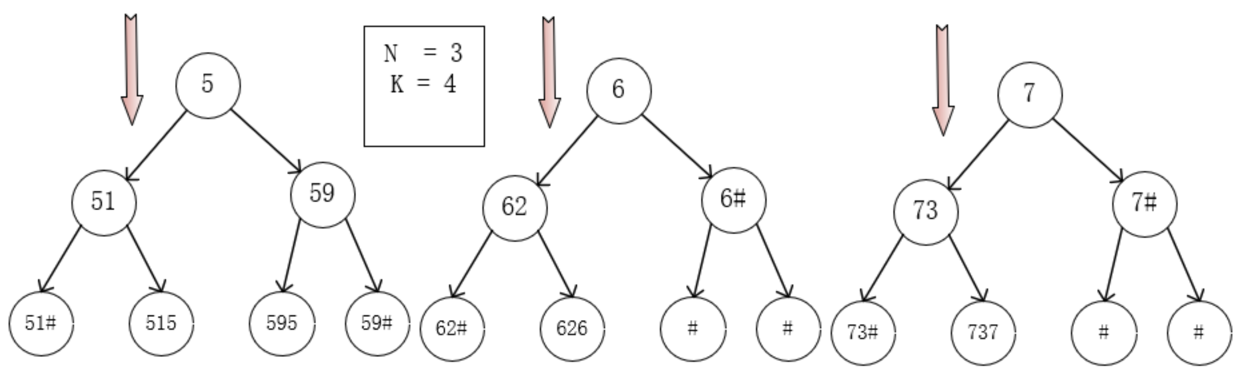

图 1: BFS 示例图

1.2 核心代码

```
vector<int> bfs(int N, int K){
// 初始化vector数组为{1,2,3,4,5,6,7,8,9}
vector<int> res{1,2,3,4,5,6,7,8,9};

// 预处理：如果N=1，则清空数组并返回
if (1 == N){
  res.clear();
  return res;
}

// BFS遍历求解
// 计数器：对位数遍历计数
int count = 1;

while(count < N) // 外层循环：位数遍历
{
  int resSize = res.size();  // 获取当前数组的长度

  // 内层循环：对当前数组中每一位数字遍历取个位数进行求解
  for (int i = 0; i < resSize; i++) {
    // 取末位数
    int unit_digit = res[i] % 10;

    //如果个位数和差值之差不小于0，则说明可以再加一位数字
    if (unit_digit - K >= 0)
    res.push_back(res[i] * 10 + unit_digit - K);
    // 如果个位数和差值之和不大于9，则说明可以再加一位数字
    //由于K=0的情况在上式已经考虑，为避免重复加入K!=0的条件
    if (unit_digit + K <=9 && K != 0)
    res.push_back(res[i] * 10 + unit_digit + K);
  }

  // 更新数组中的值：删除原先的基数
  res.erase(res.begin(), res.begin() + resSize);
  // 完成一轮位数的添加
  count++;
}

return res;
}
```

1.3 效率分析

假设位数为 N，绝对值差为 K。由于最高位上的数字有 9 中选择，剩余 N-1 位数字各有两种选择，根据排列组合中的乘法原理可知，总共的组合数为 $9 \cdot 2^{N-1}$。因此，其时间复杂度为：$C(n) = 9 \cdot 2^{N-1} = \frac{9}{2} \cdot 2^{N} \in \Theta(2^{N})$。

对于其空间复杂度，取决于 N 和 K 的取值。直观上理解是最后一层的可行解的个数，但事实并非如此。由于每一层可能淘汰一些结果，故在采用 vector 的实现过程中，其取决于相邻两层个数之和的最大值。在最坏的情况下，其空间复杂度也为 $\Theta(2^{N})$。

1.4 运行结果

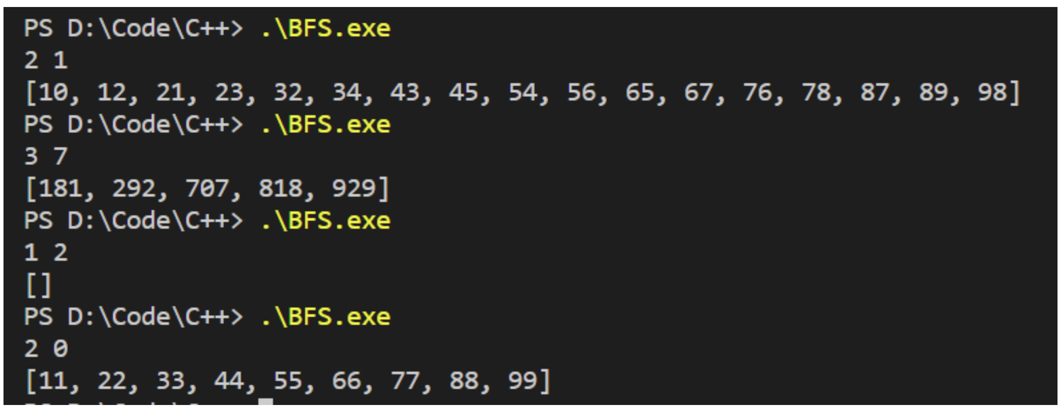

图 2: BFS 运行结果

2 DFS

2.1 核心思想

DFS 算法与 BFS 算法的求解过程差不多，唯一不同的是 DFS 是先从一个位数开始，直到求得以该数开头的所有可行解，才会继续计算以下一个数开头的可行解。简单示例图如图 3。对比图 1 和图 3 可知，BFS 算法是先找出最高位上的所有可行数，再计算次高位，依次类推；而 DFS 是先计算以某数开头的所有可行解，再计算下一数开头的所有可行解，依次类推。

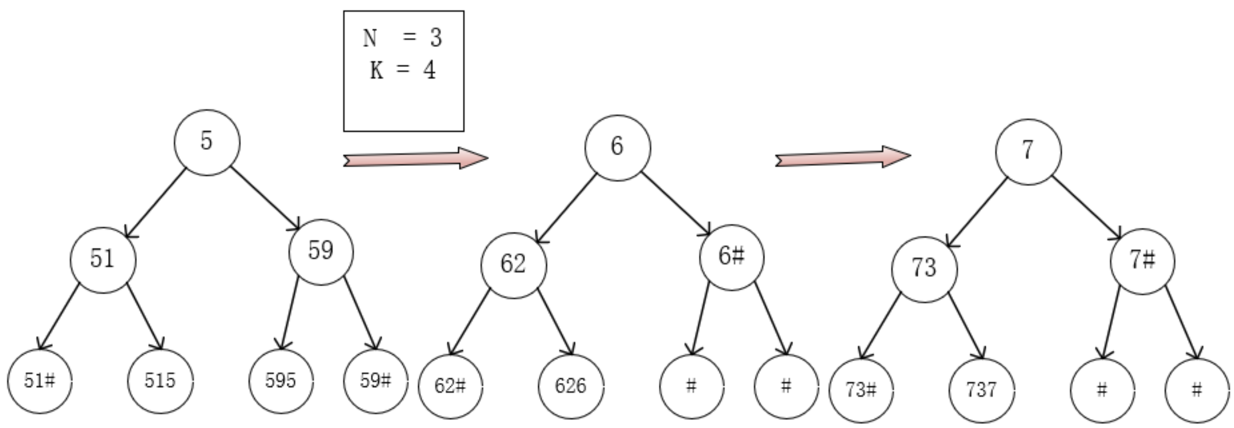

图 3: DFS 示例图

2.2 核心代码

```cpp
vector<int>res; // 全局数组

// 递归求解以first_digit数字开头的所有可行解
void Recursion(int first_digit, int n, int K)
{
  if (1 == n) // N位数已取得
  {
    res.push_back(first_digit);
    return;
  }

  // 当前数值的末位上的数字
  int unit_digit = first_digit % 10;

  //如果个位数和差值之差不小于0，则添加一位数，位数减1
  if (unit_digit - K >= 0)
  Recursion(10 * first_digit + unit_digit - K, n - 1, K);
  // 如果个位数和差值之和不大于9，则添加一位数，位数减1
  if (K!=0 && unit_digit + K <= 9) //由于K=0的情况在上式已经考虑，为避免重复加入K! =0的条件
  Recursion(10 * first_digit + unit_digit + K, n - 1 , K);
}

// 遍历递归求解
vector<int> dfs(int N, int K) {
  if (1 == N)  // 若N=1，则返回空数组
  return vector<int>{};

  // 遍历递归求解分别以{1,2,3,4,5,6,7,8,9}开头的可行解
  for (int i = 1; i <= 9; i++)
  Recursion(i, N, K);
  return res;
}
```

2.3 效率分析

实际上，DFS 和 BFS 算法解决这个问题的方法都是在遍历一片森林，其中包含了 9 棵二叉树。树的最长深度为就是已知的位数 N，而结点数取决于 N 和 K 的取值，并不容易确定，在最坏的情况下为 $9 \cdot 2^{N-1}$。因此，DFS 算法的复杂度也为 $\Theta(2^N)$。

2.4 运行结果

```
PS D:\Code\C++> .\DFS.exe
2 1
[10, 12, 21, 23, 32, 34, 43, 45, 54, 56, 65, 67, 76, 78, 87, 89, 98]
PS D:\Code\C++> .\DFS.exe
3 7
[181, 292, 707, 818, 929]
PS D:\Code\C++> .\DFS.exe
1 0
[]
PS D:\Code\C++> .\DFS.exe
2 0
[11, 22, 33, 44, 55, 66, 77, 88, 99]
```

图 4: DFS 运行结果

3 综合分析

虽然 DFS 和 BFS 算法的时间复杂度都是指数级的，但其空间复杂度可以通过调整存储的数据结构进行优化，相比于蛮力法（即从 N 位最小数到 N 位最大数遍历寻找每一个满足条件的数字）的时间复杂度 $9 \cdot 10^{N-1} \in \Theta(10^N)$,DFS 算法和 BFS 算法的效率提高了不少。