

回溯算法 one pager

MLE 算法指北

2025 年 2 月 5 日

1 回溯算法概述

回溯算法 (Backtracking) 是一种暴力搜索算法，适用于组合、排列、子集、路径搜索等问题，其核心思想是尝试 (选择一个可能的解)，回溯 (如果发现不可行则撤回)，然后继续尝试其他可能性。

1.1 回溯算法的代码框架

回溯算法一般包含：

- **状态变量**：记录搜索进度 (如当前路径、可选集合等)
- **终止条件**：满足条件时，将当前路径加入解集
- **递归搜索**：枚举当前状态所有可能的选择
- **回溯 (撤销选择)**：恢复到上一步状态，继续尝试其他可能解

通用代码框架：

Algorithm 1 Backtracking 通用框架

```
1: function BACKTRACK(状态变量)
2:   if 终止条件满足 then
3:     记录当前解 return
4:   end if
5:   for 选择 in 可选集合 do
6:     做选择
7:     Backtrack(更新后的状态变量)
8:     撤销选择 (回溯)
9:   end for
10: end function
```

2 经典回溯问题

我们选择 5 道最经典的 Backtracking 题目，并总结解法。

2.1 组合问题 (Combinations)

题目描述：给定两个整数 n 和 k ，返回所有可能的 k 个数的组合。

代码实现：

Listing 1: Combinations

```
def combine(n, k):
    res = []

    def backtrack(start, path):
        if len(path) == k:
            res.append(path[:]) # 记录解
            return

        for i in range(start, n + 1):
            path.append(i)
            backtrack(i + 1, path)
            path.pop() # 回溯
```

```

    backtrack(1, [])
    return res

```

时间复杂度: $O(\binom{n}{k}) = O(n!/(k!(n-k)!))$

2.2 全排列问题 (Permutations)

题目描述: 给定一个不含重复数字的数组, 返回所有可能的全排列。

代码实现:

Listing 2: Permutations

```

def permute(nums):
    res = []

    def backtrack(path, used):
        if len(path) == len(nums):
            res.append(path[:])
            return

        for i in range(len(nums)):
            if used[i]: # 避免重复选择
                continue

            used[i] = True
            path.append(nums[i])
            backtrack(path, used)
            path.pop() # 回溯
            used[i] = False

    backtrack([], [False] * len(nums))
    return res

```

时间复杂度: $O(n!)$

2.3 子集问题 (Subsets)

题目描述: 给定一个整数数组, 返回所有可能的子集 (幂集)。

代码实现:

Listing 3: Subsets

```

def subsets(nums):
    res = []

    def backtrack(start, path):
        res.append(path[:]) # 记录当前路径

        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop() # 回溯

    backtrack(0, [])
    return res

```

时间复杂度: $O(2^n)$

2.4 N 皇后问题 (N-Queens)

题目描述: 在 $N \times N$ 的棋盘上放置 N 个皇后, 使得它们不能相互攻击, 求所有解法。

代码实现:

Listing 4: N-Queens

```

def solveNQueens(n):
    res = []
    board = [["."] * n for _ in range(n)]

```

```

def is_valid(row, col):
    for i in range(row):
        if board[i][col] == "Q" or \
            (col - (row - i) >= 0 and board[i][col - (row - i)] == "Q") or \
            (col + (row - i) < n and board[i][col + (row - i)] == "Q"):
            return False
    return True

def backtrack(row):
    if row == n:
        res.append(["".join(r) for r in board])
        return

    for col in range(n):
        if is_valid(row, col):
            board[row][col] = "Q"
            backtrack(row + 1)
            board[row][col] = "."

    backtrack(0)
    return res

```

时间复杂度: $O(n!)$

2.5 单词搜索 (Word Search)

题目描述: 给定一个 $m \times n$ 的字母网格和一个单词, 判断该单词是否存在于网格中, 可以上下左右移动, 但不能重复使用某个单元格。

代码实现:

Listing 5: Word Search

```

def exist(board, word):
    rows, cols = len(board), len(board[0])

    def backtrack(r, c, index):
        if index == len(word):
            return True

        if r < 0 or r >= rows or c < 0 or c >= cols or board[r][c] != word[index]:
            return False

        temp, board[r][c] = board[r][c], "#" # 标记访问
        found = (backtrack(r + 1, c, index + 1) or
                 backtrack(r - 1, c, index + 1) or
                 backtrack(r, c + 1, index + 1) or
                 backtrack(r, c - 1, index + 1))
        board[r][c] = temp # 恢复原状态
        return found

    return any(backtrack(r, c, 0) for r in range(rows) for c in range(cols))

```

时间复杂度: $O(m \times n \times 3^k)$

3 经典回溯问题及 LeetCode 题目整理

回溯算法广泛应用于组合、排列、子集、棋盘问题和路径搜索。本章整理了每个类别最具代表性的 LeetCode 题目, 帮助系统学习和练习回溯算法。

3.1 组合问题 (Combinations)

组合问题涉及从给定集合中选取 k 个元素的所有可能组合, 不考虑顺序。常用递归 + 剪枝来优化搜索。

- 77. Combinations
- 39. Combination Sum

- 40. Combination Sum II
- 216. Combination Sum III
- 377. Combination Sum IV
- 254. Factor Combinations
- 17. Letter Combinations of a Phone Number
- 401. Binary Watch
- 1079. Letter Tile Possibilities
- 1268. Search Suggestions System

方法论：

- 递归搜索，并通过 *start* 控制范围，防止重复选取。
- 组合问题一般不能重复选择同一元素，避免无效搜索。

3.2 排列问题 (Permutations)

排列问题要求输出所有可能的排列，需要记录已使用的元素，防止重复。

- 46. Permutations
- 47. Permutations II
- 60. Permutation Sequence
- 784. Letter Case Permutation
- 996. Number of Squareful Arrays
- 526. Beautiful Arrangement
- 267. Palindrome Permutation II
- 996. Number of Squareful Arrays
- 996. Next Closest Time
- 996. Stamping the Sequence

方法论：

- 递归过程中维护 *used* 数组，防止重复使用元素。
- 当所有元素都被选取时，保存当前排列。

3.3 子集问题 (Subsets)

子集问题的目标是找到所有可能的子集（幂集），可通过递归 + 回溯构造所有可能的解。

- 78. Subsets
- 90. Subsets II
- 79. Word Search
- 491. Increasing Subsequences
- 1239. Maximum Length of a Concatenated String with Unique Characters
- 131. Palindrome Partitioning
- 332. Reconstruct Itinerary
- 473. Matchsticks to Square
- 967. Numbers With Same Consecutive Differences

- **1079. Letter Tile Possibilities**

方法论：

- 递归时，每次都加入当前路径，从而生成所有可能的子集。
- 处理重复元素时，可先排序，然后跳过重复选项。

—

3.4 棋盘问题 (N-Queens & Sudoku)

这类问题通常涉及在棋盘上放置元素（如皇后、数独），需要合法性检查。

- **51. N-Queens**
- **52. N-Queens II**
- **37. Sudoku Solver**
- **36. Valid Sudoku**
- **130. Surrounded Regions**
- **994. Rotting Oranges**
- **529. Minesweeper**
- **489. Robot Room Cleaner**
- **302. Smallest Rectangle Enclosing Black Pixels**
- **1274. Number of Ships in a Rectangle**

方法论：

- N 皇后问题中，每行只能放置一个皇后，递归按行搜索，并通过合法性检查进行剪枝。
- 数独求解中，使用回溯 + 递归搜索空位，并逐步填充数字。

—

3.5 路径搜索 (Word Search & DFS)

路径搜索问题通常涉及 ** 在网格或图中搜索特定路径 **，DFS 是主要手段。

- **79. Word Search**
- **212. Word Search II**
- **130. Surrounded Regions**
- **200. Number of Islands**
- **1254. Number of Closed Islands**
- **417. Pacific Atlantic Water Flow**
- **529. Minesweeper**
- **980. Unique Paths III**
- **1219. Path with Maximum Gold**
- **847. Shortest Path Visiting All Nodes**

方法论：

- 采用 DFS + 回溯搜索所有可能路径。
- 使用 visited 标记已访问的单元格，避免重复搜索。
- 剪枝优化：若当前搜索方向无解，则及时返回。

—

3.6 总结

- ** 组合问题 **: 控制递归范围，避免重复计算。
- ** 排列问题 **: 使用 'used' 数组标记已选元素。
- ** 子集问题 **: 每次递归都记录当前路径。
- ** 棋盘问题 **: 加入合法性检查，减少搜索空间。
- ** 路径搜索 **: 使用 DFS 结合回溯，剪枝优化。