排列式 DFS

主讲人 令狐冲

什么是排列式搜索?

问题的模型是求出一个集合中所有元素的满足某个条件的排列

排列和组合的区别是排列是有顺序的

[1,2,3] 和 [3,2,1] 是同一个组合但不是同一个排列

排列的搜索树

画一画

全排列问题

https://www.lintcode.com/problem/permutations/

求出给定没有重复的输入集的所有排列

[1,2,3] 有 6 个排列

代码

```java
public List<List<Integer>> permute(int[] nums) {
    List<List<Integer>> results = new ArrayList<>();
    if (nums == null) {
        return results;
    }

    dfs(nums, new boolean[nums.length], new ArrayList<Integer>(), results);

    return results;
}

private void dfs(int[] nums,
                 boolean[] visited,
                 List<Integer> permutation,
                 List<List<Integer>> results) {
    if (nums.length == permutation.size()) {
        results.add(new ArrayList<Integer>(permutation));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        if (visited[i]) {
            continue;
        }

        permutation.add(nums[i]);
        visited[i] = true;
        dfs(nums, visited, permutation, results);
        visited[i] = false;
        permutation.remove(permutation.size() - 1);
    }
}
```

```python
def permute(self, nums):
    if not nums:
        return [[]]

    permutations = []
    self.dfs(nums, [], set(), permutations)
    return permutations

def dfs(self, nums, permutation, visited, permutations):
    if len(nums) == len(permutation):
        permutations.append(list(permutation))
        return

    for num in nums:
        if num in visited:
            continue
        permutation.append(num)
        visited.add(num)
        self.dfs(nums, permutation, visited, permutations)
        visited.remove(num)
        permutation.pop()
```

算法的时间复杂度是多少呢?

时间复杂度

O(N! * N)

DFS时间复杂度通用公式——

O(方案总数 * 构造每个方案的时间)

```java
public List<List<Integer>> permuteUnique(int[] nums) {
    List<List<Integer>> results = new ArrayList<>();
    if (nums == null) {
        return results;
    }

    Arrays.sort(nums);
    dfs(nums, new boolean[nums.length], new ArrayList<Integer>(), results);

    return results;
}
```

https://www.lintcode.com/problem/permutations-ii/description

```java
// 1. 递归的定义
//  找到所有 permutation 开头的排列, 加到 results 里
private void dfs(int[] nums,
                 boolean[] visited,
                 List<Integer> permutation,
                 List<List<Integer>> results) {
    // 2. 递归的出口
    if (nums.length == permutation.size()) {
        results.add(new ArrayList<Integer>(permutation));
        return;
    }

    // 3. 递归的拆解
    for (int i = 0; i < nums.length; i++) {
        if (visited[i]) {
            continue;
        }
        // 当前的数和前面的数一样, 但前面数, 没用过
        if (i > 0 && nums[i] == nums[i - 1] && !visited[i - 1]) {
            continue;
        }

        // [] => [1]
        permutation.add(nums[i]);
        visited[i] = true;
        dfs(nums, visited, permutation, results);
        // [1] => []
        visited[i] = false;
        permutation.remove(permutation.size() - 1);
    }
```

多重循环的方式实现的全排列

九章算法
www.jiuzhang.com

```python
def permute(self, nums):
    if not nums:
        return [[]]

    results = []
    if len(nums) == 1:
        results.append(nums)
    elif len(nums) == 2:
        for a in nums:
            for b in nums:
                if a != b:
                    results.append([a, b])
    elif len(nums) == 3:
        for a in nums:
            for b in nums:
                if a == b: continue
                for c in nums:
                    if c in [a, b]: continue
                    results.append([a, b, c])
    elif len(nums) == 4:
        for a in nums:
            for b in nums:
                if a == b: continue
                for c in nums:
                    if c in [a, b]: continue
                    for d in nums:
                        if d in [a, b, c]: continue
                        results.append([a, b, c, d])
    elif len(nums) == 5:
        for a in nums:
```

递归 vs 循环

递归实现搜索的本质
是实现了按照给定参数来决定循环层数
的一个多重循环
递归实现的搜索=n重循环，n由输入决定

著名的NP问题：TSP问题

https://www.lintcode.com/problem/traveling-salesman-problem

排列式搜索的典型代表

Traveling Salesman Problem

又称中国邮路问题

5种解法

暴力 DFS

暴力 DFS + 最优性剪枝(prunning)

状态压缩动态规划

随机化算法 - 使用交换调整策略

随机化算法 - 使用反转调整策略

一个题掌握四种算法:

1. 排列式搜索 Permutaition Style DFS
2. 最优性剪枝算法 Optimal Prunning Algorithm
3. 状态压缩动态规划 State Compression Dynamic Programming
4. 随机化算法 Randomlization Algorithm
    a. 又称为遗传算法 Genetic Algorithm，模拟退火算法 Simulated Annealing

暴力搜索

```python
class Result:
    def __init__(self):
        self.min_cost = float('inf')

class Solution:
    """
    @param n: an integer,denote the number of cities
    @param roads: a list of three-tuples,denote the road between cities
    @return: return the minimum cost to travel all cities
    """
    def minCost(self, n, roads):
        graph = self.construct_graph(roads, n)
        result = Result()
        self.dfs(1, n, set([1]), 0, graph, result)
        return result.min_cost

    def dfs(self, city, n, visited, cost, graph, result):
        if len(visited) == n:
            result.min_cost = min(result.min_cost, cost)
            return

        for next_city in graph[city]:
            if next_city in visited:
                continue
            visited.add(next_city)
            self.dfs(next_city, n, visited, cost + graph[city][next_city], graph, result)
            visited.remove(next_city)

    def construct_graph(self, roads, n):
        graph = {
            i: {j: float('inf') for j in range(1, n + 1)}
            for i in range(1, n + 1)
        }
        for a, b, c in roads:
            graph[a][b] = min(graph[a][b], c)
            graph[b][a] = min(graph[b][a], c)
        return graph
```

加入剪枝(prunning)优化

```python
def minCost(self, n, roads):
    graph = self.construct_graph(roads, n)
    result = Result()
    self.dfs(1, n, [1], set([1]), 0, graph, result)
    return result.min_cost

def dfs(self, city, n, path, visited, cost, graph, result):
    if len(visited) == n:
        result.min_cost = min(result.min_cost, cost)
        return

    for next_city in graph[city]:
        if next_city in visited:
            continue
        if self.has_better_path(graph, path, next_city):
            continue
        visited.add(next_city)
        path.append(next_city)
        self.dfs(
            next_city,
            n,
            path,
            visited,
            cost + graph[city][next_city],
            graph,
            result,
        )
        path.pop()
        visited.remove(next_city)
```

```python
def construct_graph(self, roads, n):
    graph = {
        i: {j: float('inf') for j in range(1, n + 1)}
        for i in range(1, n + 1)
    }
    for a, b, c in roads:
        graph[a][b] = min(graph[a][b], c)
        graph[b][a] = min(graph[b][a], c)
    return graph

def has_better_path(self, graph, path, city):
    for i in range(1, len(path)):
        if graph[path[i - 1]][path[i]] + graph[path[-1]][city] >\
              graph[path[i - 1]][path[-1]] + graph[path[i]][city]:
            return True
    return False
```

状态压缩动态规划

```
def minCost(self, n, roads):
    graph = self.construct_graph(roads, n)
    state_size = 1 << n
    f = [
        [float('inf')] * (n + 1)
        for _ in range(state_size)
    ]
    f[1][1] = 0
    for state in range(state_size):
        for i in range(2, n + 1):
            if state & (1 << (i - 1)) == 0:
                continue
            prev_state = state ^ (1 << (i - 1))
            for j in range(1, n + 1):
                if prev_state & (1 << (j - 1)) == 0:
                    continue
                f[state][i] = min(f[state][i], f[prev_state][j] + graph[j][i])
    return min(f[state_size - 1])

def construct_graph(self, roads, n):
    graph = {
        i: {j: float('inf') for j in range(1, n + 1)}
        for i in range(1, n + 1)
    }
    for a, b, c in roads:
        graph[a][b] = min(graph[a][b], c)
        graph[b][a] = min(graph[b][a], c)
    return graph
```

什么是随机化算法

随机化一个初始方案
通过一个调整策略调整到局部最优值
在时间限制内重复上述过程直到快要超时

随机化算法 - 使用交换调整策略

```python
def minCost(self, n, roads):
    graph = self.construct_graph(roads, n)
    min_cost = float('inf')
    for _ in range(RANDOM_TIMES):
        path = self.get_random_path(n)
        cost = self.adjust_path(path, graph)
        min_cost = min(min_cost, cost)
    return min_cost

def construct_graph(self, roads, n):
    graph = {
        i: {j: float('inf') for j in range(1, n + 1)}
        for i in range(1, n + 1)
    }
    for a, b, c in roads:
        graph[a][b] = min(graph[a][b], c)
        graph[b][a] = min(graph[b][a], c)
    return graph

def get_random_path(self, n):
    import random

    path = [i for i in range(1, n + 1)]
    for i in range(2, n):
        j = random.randint(1, i)
        path[i], path[j] = path[j], path[i]
    return path
```

```python
def adjust_path(self, path, graph):
    n = len(graph)
    adjusted = True
    while adjusted:
        adjusted = False
        for i in range(1, n):
            for j in range(i + 1, n):
                if self.can_swap(path, i, j, graph):
                    path[i], path[j] = path[j], path[i]
                    adjusted = True
    cost = 0
    for i in range(1, n):
        cost += graph[path[i - 1]][path[i]]
    return cost

def can_swap(self, path, i, j, graph):
    before = self.adjcent_cost(path, i, path[i], graph)
    before += self.adjcent_cost(path, j, path[j], graph)
    after = self.adjcent_cost(path, i, path[j], graph)
    after += self.adjcent_cost(path, j, path[i], graph)
    return before > after

def adjcent_cost(self, path, i, city, graph):
    cost = graph[path[i - 1]][city]
    if i + 1 < len(path):
        cost += graph[city][path[i + 1]]
    return cost
```

随机化算法 - 使用反转调整策略

```python
def minCost(self, n, roads):
    graph = self.construct_graph(roads, n)
    min_cost = float('inf')
    for _ in range(RANDOM_TIMES):
        path = self.get_random_path(n)
        cost = self.adjust_path(path, graph)
        min_cost = min(min_cost, cost)
    return min_cost

def construct_graph(self, roads, n):
    graph = {
        i: {j: float('inf') for j in range(1, n + 1)}
        for i in range(1, n + 1)
    }
    for a, b, c in roads:
        graph[a][b] = min(graph[a][b], c)
        graph[b][a] = min(graph[b][a], c)
    return graph

def get_random_path(self, n):
    import random

    path = [i for i in range(1, n + 1)]
    for i in range(2, n):
        j = random.randint(1, i)
        path[i], path[j] = path[j], path[i]
    return path
```

```python
def adjust_path(self, path, graph):
    n = len(graph)
    adjusted = True
    while adjusted:
        adjusted = False
        for i in range(1, n):
            for j in range(i + 1, n):
                if self.can_reverse(path, i, j, graph):
                    self.reverse(path, i, j)
                    adjusted = True
    cost = 0
    for i in range(1, n):
        cost += graph[path[i - 1]][path[i]]
    return cost

def can_reverse(self, path, i, j, graph):
    before = graph[path[i - 1]][path[i]]
    if j + 1 < len(path):
        before += graph[path[j]][path[j + 1]]
    after = graph[path[i - 1]][path[j]]
    if j + 1 < len(path):
        after += graph[path[i]][path[j + 1]]
    return before > after

def reverse(self, path, i, j):
    while i < j:
        path[i], path[j] = path[j], path[i]
        i += 1
        j -= 1
```