算法第五次作业

李雨轩

计算机2205

2204112913

一. 子集和问题

1. 题目描述

给定一个整数集合 $S = \{a_1, a_2, \ldots, a_n\}$ 和一个目标整数 c，子集和问题是要确定是否存在一个 S 的子集，其元素之和等于 T。形式化地，我们希望找到一个子集 $S' \subseteq S$，使得 $\sum_{a_i \in S'} a_i = c$。

2. 算法设计

1. **backtrack 函数**：用来进行回溯搜索的主要函数。它采用递归方式，在每一步尝试添加或不添加当前元素到路径中，并继续递归搜索。
2. **递归终止条件**：当 `index` 超过了集合的长度时，递归终止，这意味着已经处理完了所有元素。
3. **条件判断**：在每一步递归中，首先检查当前的和是否等于目标和 `target_sum`，如果是，则输出路径并返回。这是一种剪枝策略，避免继续向下搜索，从而提高效率。如果当前和加上当前元素仍然小于等于目标和，则将当前元素添加到路径中，并递归调用 `backtrack` 函数，同时更新当前的和和下一个元素的索引。如果不添加当前元素也无法满足条件，则直接递归调用 `backtrack` 函数，不添加当前元素，继续搜索。
4. **路径回退**：在递归返回之前，需要回退当前已经添加到路径中的元素，以便进行下一步的尝试。这是通过从路径中弹出最后一个元素，并将当前元素对应的值从当前和中减去实现的。
5. **全局变量 sum**：使用了一个静态的全局变量 `sum` 来保存当前路径的和。
6. **路径打印**：在找到符合条件的子集和时，调用 `printPath` 函数输出路径。

3. 算法描述

```
function subsetSum(set, target_sum):
```

```
    backtrack(set, length(set), [], target_sum, 0)

function backtrack(set, length, path, target_sum, index):
    static sum = 0

    // 终止条件
    if index >= length:
        return

    // 如果当前和等于目标和，则输出路径并返回
    if sum == target_sum:
        printPath(path)
        return

    // 尝试添加当前元素到路径中
    if sum + set[index] <= target_sum:
        sum += set[index]
        path.push(set[index])
        backtrack(set, length, path, target_sum, index+1)

        // 回溯，从路径中移除当前元素
        sum -= set[index]
        path.pop()

    // 尝试不添加当前元素，继续向后搜索
    backtrack(set, length, path, target_sum, index+1)

```

二. 最小费用问题

1. 题目描述

设有n件工作分配给n个人。将工作i分配给第j个人所需要的费用为c_{ij}。试设计一个算法，为每个人都分配一件不同的工作，使得总费用达到最小。

2. 问题分析与算法设计

使用回溯算法来解决每个人分配不同工作的问题。问题的描述是将n件工作分配给n个人，使得每个人都分配到一件不同的工作，并且总费用达到最小。

让我解释一下代码的执行流程：

1. `backtrack` 函数是实现回溯算法的核心部分。它采用了递归的方式，在每一步尝试所有可能的工作分配，并更新最小费用。其中：
   - `c` 是一个二维向量，存储每个人分配每个工作的费用。
   - `n` 是工作和人的数量。
   - `index` 表示当前正在考虑的人的索引。

- `choise` 存储当前的工作分配情况。
- `min_cost` 保存当前找到的最小总费用。
- `now_cost` 保存当前分配方案的总费用。

2. `main` 函数读取输入，并初始化调用 `backtrack` 函数所需的参数。
3. `std::swap(choise[i], choise[index])` 用于尝试不同的工作分配。
4. `if(now_cost<min_cost)` 检查当前分配方案的总费用是否小于当前最小总费用，如果是，则继续搜索该方案。
5. 当 `index` 达到 `n-1` 时，意味着已经分配完所有人的工作，此时更新最小总费用并返回。

这个程序通过递归地尝试所有可能的工作分配方案，以找到总费用最小的方案

3. 算法描述

```
procedure backtrack(cost, n, index, choice, min_cost, now_cost)
    if index >= n - 1 then
        if now_cost < min_cost then
            min_cost = now_cost
        end if
        return
    end if

    for i from index to n do
        swap(choice[i], choice[index])
        if now_cost < min_cost then
            now_cost = now_cost + cost[i][choice[i]]
            backtrack(cost, n, index + 1, choice, min_cost, now_cost)
            now_cost = now_cost - cost[i][choice[i]]
        end if
        swap(choice[i], choice[index])
    end for
end procedure
```

三. 最小长度电路板排列问题

1. 题目描述

最小长度电路板排列问题是大规模电子系统设计中提出的实际问题。该问题的提法是，将 n 块电路板以最佳排列方案插入带有 n 个插槽的机箱中。n 块电路板的不同的排列方式对应于不同的电路板插入方案。

设$B=1，2，\ldots，n$是 n 块电路板的集合。集合$L=N_1，N_2,\cdots，N_m$是 n 块电路板的 m 个连接块。其中每个连接块N_i是 B 的一个子集，且 N_i 中的电路板用同一根导线连接在一起。 在最小长度电路板排列问题中，连接块的长度是指该连接块中第l快电路板到最后一块电路板之间的距离。

试设计一个回溯法，找出所给n个电路板的排列问题，使得m个连接块中最大长度达到最小。

2. 问题分析与算法设计

设计一个回溯法来解决这个最小长度电路板排列问题

2.1 定义问题的表示

- **电路板排列**：使用一个数组 `P[1..n]` 来表示电路板的排列，其中 `P[i]` 是排在第 `i` 个位置的电路板编号。
- **连接块的长度**：对于每个连接块 `Ni`，它的长度是该块中最左边和最右边电路板在 `P` 中位置的差的绝对值加1（即 `max_position - min_position + 1`）。

2.2 状态空间的定义

- **初始状态**：一个空的排列。
- **状态转移**：每一步在当前排列的尾部添加一个还没有被排列的电路板。
- **目标状态**：所有电路板都被排列完毕，并且所有连接块的最大长度达到可能的最小值。

2.3 回溯的搜索过程

- 使用递归来实现回溯搜索。在每一步尝试将每一个尚未排列的电路板加到当前排列的尾部，然后递归地继续处理。
- 对于每一步的排列尝试，计算当前所有连接块的最大长度，更新当前找到的最小的最大长度。

2.4 剪枝优化

- 在搜索过程中，如果当前的连接块的最大长度已经超过了已经找到的最小的最大长度，则可以停止进一步搜索当前分支（剪枝）。

3. 代码实现

```python
def backtrack(P, used, current_max, best_max):
    n = len(P)
    if len(P) == n:
        # 所有电路板都排列完毕
        best_max = min(best_max, current_max)
        return best_max

    for i in range(1, n + 1):
        if not used[i]:
            # 尝试添加电路板i
            P.append(i)
            used[i] = True

            # 更新连接块的最大长度
            local_max = 0
```

```
            for block in connection_blocks:
                min_pos = min(P.index(x) for x in block if x in P)
                max_pos = max(P.index(x) for x in block if x in P)
                local_max = max(local_max, max_pos - min_pos + 1)

            if local_max < best_max:
                # 仅当当前的最大长度小于已找到的最佳最大长度时继续
                best_max = backtrack(P, used, max(current_max, local_max), best_max)

            # 回溯
            used[i] = False
            P.pop()

    return best_max

# 初始化
n = len(boards)
P = []
used = [False] * (n + 1)
best_max = float('inf')

# 运行回溯算法
best_max = backtrack([], used, 0, best_max)
```