算法基础 2022 春 Homework 4

任课教师：陈雪 due: Apr 8, 23:59

**（1）只准讨论思路，严禁抄袭**

**（2）只能阅读 bb 上的材料和教材算法导论。严禁网上搜寻任何材料，答案或者帮助**

**以下所有算法的设计请给出伪代码，证明算法正确性以及时间复杂度。**

**问题 1** (20 分). 为了将一个文本串 $x[1,\cdots,m]$ 转换为目标串 $y[1,\cdots,n]$，我们可以使用多种变换操作。我们的目标是，给定 $x$ 和 $y$，求将 $x$ 转换为 $y$ 的一个变换操作序列。我们使用一个数组 $z$ 保存中间结果，假定它足够大，可存下中间结果的所有字符。初始时，$z$ 是空的，结束时，应有 $z[j]=y[j], j=1,2,\cdots,n$。我们维护两个下标 $i$ 和 $j$，分别指向 $x$ 中位置和 $z$ 中位置，变换操作允许改变 $z$ 的内容和这两个下标。初始时，$i=j=1$。在转换过程中应处理 $x$ 的所有字符，这意味着在变换操作结束时，应有 $i=m+1$。

我们可以使用如下变换操作：

1. 复制：从 $x$ 复制一个字符到 $z$，即进行赋值 $z[j++]=x[i++]$
2. 替换：将 $x$ 中的一个字符替换为另一个字符 $c$，即 $z[j++]=c, i++$
3. 删除：删除 $x$ 中一个字符，即 $i++$
4. 插入：将字符 $c$ 插入中 $z$，即 $z[j++]=c$

每种变换操作每次执行都有一定的代价 $cost_{operate}$，$x$ 到 $y$ 的编辑距离是将 $x$ 转换为 $y$ 的最小的变换代价之和。设计动态规划算法，输入为文本串 $x$, 目标串 $y$ 和每种操作的代价 $cost$，求 $x[1,\cdots,m]$ 到 $y[1,\cdots,n]$ 的编辑距离并打印最优操作序列。

给出伪代码，证明算法正确性以及分析算法的时间和空间复杂度。

**Answer:**

If the edit distance of shorter strings have been calculated, we can calculate the edit distance of longer string by comparing the end of two strings. Time and space complexity are both $O(m \cdot n)$.

**Algorithm 1** edit distance algorithm

```
1: function EDITDISTANCE(x, y, m, n, cost)        // calculate edit distance between x and y
2:     for i = 0 to m do
```

3: $opt[i][0] = m \cdot cost_{delete}$
4: $operate[i][0] = delete$
5: **for** $j = 0$ to $n$ **do**
6: $opt[0][j] = n \cdot cost_{insert}$
7: $operate[i][0] = insert$
8: **for** $k = 2$ to $m + n$ **do**
9: **for** $i = \max\{1, k - m\}$ to $\min\{n, k - 1\}$ **do**
10: $j = k - i$
11: **if** $x[i] == y[j]$ **then**
12: $ed_{copy} = opt[i - 1][j - 1] + cost_{copy}$
13: **else**
14: $ed_{copy} = +\infty$
15: $ed_{replace} = opt[i - 1][j - 1] + cost_{replace}$
16: $ed_{delete} = opt[i - 1][j] + cost_{delete}$
17: $ed_{insert} = opt[i][j - 1] + cost_{insert}$
18: $opt[i][j] = \min\limits_{operate} ed_{operate}$
19: $operate[i][j] = \arg\min\limits_{operate} ed_{operate}$
20: $sequence = [\ ]$
21: **while** $i \geq 0$ and $j \geq 0$ **do**
22: $sequence.push(operate[i][j])$
23: $(i, j) = previousStep(operate[i][j])$
24: **return** $opt[m][n], sequence.inverse()$

---

**问题 2** (20 分). 给定一棵有 $n$ 个顶点的树 $T$，树上顶点集合 $V = \{v_1, v_2, \cdots, v_n\}$，每个顶点 $v_i \in V$ 被赋予了一个权重 $f_i \in \mathbb{R}$。如果一个顶点子集 $S \subset V$ 满足对于任意的 $v_i, v_j \in S$，$v_i$ 和 $v_j$ 都不相邻，我们称它为 $T$ 的一个顶点独立子集。我们希望找到树上的一个顶点独立子集 $S_{max}$，使得 $\sum\limits_{v_i \in S_{max}} f_i$ 能够最大化。

树 $T$ 采样算法导论 10.4 节介绍的左孩子右兄弟表示法描述，顶点的权重以每个顶点的属性的形式给出。算法输入为树 $T$ 的根 $root$，输出为最优的顶点独立子集 $S_{max}$。写出算法伪代码，证明算法正确性以及时间复杂度。

**Answer:**

If a solution of the problem contains a node, then the solution wouldn't contains it's children. And if the solution doesn't, the children could be in the solution. So with the knowledge of the optimal values in subtrees whose root is it's children in the two cases, we can compare the them to get the optimal values in tree whose root is the node in the two cases. Calculating in this way for all of tree by bottom-up.

**Algorithm 2** indepent vertex set algorithm

```
1: function RA(root)        // calculate maximum value when root is and isn't in the largest indepent vertex set
2:     child = root.left-child
3:     while child ≠ nil do        // Solve the problem in subtree recursively
4:         RA(child)
5:         child = child.right-sibling
6:     child = root.left-child
7:     root.in = root.f        // root.in is the maximum value if root is in the set
8:     root.out = 0        // root.out is the maximum value if root is out of the set
9:     while child ≠ nil do
10:        root.in+ = child.out        // choose root if we don't choose child
11:        root.out+ = max{child.in, child.out}        // otherwise
12:        child = child.right-sibling
13: function RS(root, isParentChosen)        // get the result
14:    if root.in > root.out and isParentChosen == False then        // if root need to be chosen
15:        result = [root]
16:        child = root.left-child
17:        isRootChosen = True
18:    else        // otherwise
19:        result = [ ]
20:        isRootChosen = False
21:    child = root.left-child
22:    while child ≠ nil do
```

23: $result+ =$RB$(child, isRootChosen)$

24: $child = child.right\text{-}sibling$

25: **function** IVS($T$) // find the largest indepent vertex set of T

26: RA($T.root$)

27: **return** RS($T.root, False$)

---

**问题 3** (20 分). 在很多网络问题中，我们需要合理地设置若干个通信中心，来调配各个节点间的流量。给定一棵有 $n$ 个节点的树状网络 $T$，其中节点分别是 $v_1, v_2, \cdots, v_n$，每对相邻节点 $v_i$ 和 $v_j$ 之间有正整数距离 $f_{ij}$。不相邻节点 $v_p$ 和 $v_q$ 之间的距离定义为 $dist(v_p, v_q) = \sum_{v_iv_j \in v_p \sim v_q} f_{ij}$，即不相邻节点的距离是两节点间的简单路径上的距离之和。

现在我们想在网络上设置 $k$ 个中心 $\{u_1, \cdots, u_k\} \subset \{v_1, \cdots, v_n\}$。每个节点 $v_i$（包括中心本身）会被分配到距离它最近的中心 $u_j$，该节点的通信代价 $cost(v_i) = \min_{1 \le j \le k} dist(v_i, u_j)$。设计算法，输出树上的 $k$ 个中心，使得所有节点通信代价中的最大值 $\max_{1 \le i \le n} cost(v_i)$ 最小化。

树 $T$ 采样算法导论 10.4 节介绍的左孩子右兄弟表示法描述，父节点到子节点的距离以子节点的属性的形式给出。算法输入为树 $T$ 的根 $root$ 和中心个数 $k$，输出为最优的 $k$ 个中心 $\{u_1, \cdots, u_k\}$。给出算法的伪代码，证明算法正确性以及时间复杂度。

**提示：**可以先将它转化为一个判定问题，即原问题是否存在不大于某个距离 $r$ 的解。该判定问题可以用贪心算法解决，然后使用二分搜索便能找到最优距离。

**Answer:**

As the hint says, we need know whether there exists a solution in a given radius $r$. For the deepest node, we can chosen it's furthest ancestor within radius as a center. Because there must exists a center within radius for the node, which infers that the center cann't be further. As the node is the deepest node, this ancestor can cover the area that any of it's alternative can cover. Therefore, the ancestor is the best choice. Then we delete all node that the best choice can cover, and do it until all of node in tree have been delete. If the number of centers is no more than $k$, it's a solution for the given radius $r$. And if it isn't, there doesn't exists any solution in $r$.

The binary search needs $O(\log \sum_{i=1}^{n} f_{ij})$ ranges, as the diameter of the tree cann't be more than $\sum_{i=1}^{n} f_{ij}$. And each range costs $O(k \cdot n)$. So the time complexity is $O(k \cdot n \cdot \log \sum_{i=1}^{n} f_{ij})$.

**Algorithm 3** indepent vertex set algorithm

```
1: function FindCenter(T, k)        // find the largest indepent vertex set of T
2:     r = the longest distance among pairs of vertex
3:     l = 0
4:     while l < r do
5:         radius = ⌈(l + r)/2⌉
6:         unmark all nodes in T
7:         centers = ∅
8:         i = k
9:         while i > 0 and exists unmarked node in T do
10:            find the deepest and unmarked leaf d in T
11:            find d furthest ancestor c within radius
12:            centers = center ∪ {c}
13:            mark all node coverd by c
14:        if exists unmarked node in T then
15:            l = radius
16:        else
17:            solution = centers
18:            r = radius
19:    return solution
```

**问题 4** (25 分). 给定一个序列 $a_1, \cdots, a_n$，设计一个算法来找到最长递增子序列 $s_1, \cdots, s_m$。给出伪代码，证明算法正确性以及时间复杂度。

**Answer:**

Let the length of the longest subsequence that ends up with $a_i$ be $l_i$. So $l_j = \max\limits_{i<j, a_i<a_j}(l_i+1)$. However, it costs $\Theta(j)$ steps to calculate $l_j$ by traversing all of $i$.

Observing that for a fixed $k$ where $k \le \max\limits_{i<j} l_i$, $l_j \ge k+1$ iff $a_j > \min\limits_{i<j, l_i=k} a_i$. So we need to keep a supplementary list $b$, and $b_k = \min\limits_{l_i=k} a_i$. Then $l_j = \max\limits_{k \le \max\limits_{i<j} l_i, a_j > b_k}(k+1)$. But it still need $O(l_j)$ time for evrey $j$ as we need to find $\arg\max\limits_k a_j > b_k$.

What's more, $b$ is progressive increase as $b_{k+1} = \min_{l_i=k+1} a_i$ and $l_i = k+1$ iff $a_i > b_k$. Therefore, we can use binary search in $b$, which take $O(\log b_k) = O(\log n)$ steps.

As a consequence, we takes $O(n \log n)$ steps to find the longest subsequence.

**问题 5** (15 分)**.** 给定 $n$ 个物品及它们的重量 $w_1, \cdots, w_n$ 和价值 $v_1, \cdots, v_n$（$w_i, v_i \in \mathbb{N}, 1 \leq i \leq n$）。现在你有一个承重为 $W$ 的背包，如何才能带走总价值尽可能多的物品？设计一个动态规划算法来输出最优的选择。

(a) 给出伪代码，证明它的正确性以及时间复杂度。

(b) 它的时间复杂度是多项式级别的吗？

**Answer:**

(a) If the solution of $n-1$ object has been found out, we just need to decide whether we take away the $n$-th object or not.

(b) The time complexity of the knapsack's algorithm is $O(W \cdot n)$, which isn't polynomial. Because the input $W$ is a number, and it only take $O(\log W)$ bits.

**Algorithm 4** knapsack's algorithm

```
1: function KNAPSACK(n, w, v, W)
2:     for weight = 1 to W do        // bottom-up
3:         for obj = 1 to n do
4:             if obj == 1 then
5:                 if w[obj] > weight then
6:                     opt[weight][obj] = 0
7:                 else
8:                     opt[weight][obj] = opt[weight − w[obj]][obj − 1] + v[obj])
9:             else
10:                if w[obj] > weight then
11:                    opt[weight][obj] = opt[weight][obj − 1]
12:                else
13:                    opt[weight][obj] = max(opt[weight][obj − 1],
```

```
14:                                                  opt[weight − w[obj]][obj − 1] + v[obj])
15:     weight = W
16:     for obj = n to 2 do        // output selection
17:         if opt[weight][obj] == opt[weight][obj − 1] then
18:             take[obj] = False
19:         else
20:             take[obj] = True
21:             weight− = w[obj]
22:         if weight ≤ 0 then
23:             break
24:     if opt[weight][1] == 0 then
25:         take[1] = False
26:     else
27:         take[1] = True
28:     return take
```