Manacher 算法

描述

给定一个长度为 n 的字符串 s，请找到所有对 (i,j) 使得子串 $s[i\ldots j]$ 为一个回文串。当 $t=t_{rev}$ 时，字符串 t 是一个回文串（t_{rev} 是 t 的反转字符串）。

更进一步的描述

显然在最坏情况下可能有 $O(n^2)$ 个回文串，因此似乎一眼看过去该问题并没有线性算法。

但是关于回文串的信息可用 **一种更紧凑的方式** 表达：对于每个位置 $i=0\ldots n-1$，我们找出值 $d_1[i]$ 和 $d_2[i]$（二者分别表示以位置 i 为中心的长度为奇数和长度为偶数的回文串个数。

举例来说，字符串 $s=\mathtt{abababc}$ 以 $s[3]=b$ 为中心有三个奇数长度的回文串，也即 $d_1[3]=3$： $$ a\ \overbrace{b\ a\ \underset{s_3}{b}\ a\ b}^{d_1[3]=3}\ c $$ 字符串 $s=\mathtt{cbaabd}$ 以 $s[3]=a$ 为中心有两个偶数长度的回文串，也即 $d_2[3]=2$： $$ c\ \overbrace{b\ a\ \underset{s_3}{a}\ b}^{d_2[3]=2}\ d $$ 因此关键思路是，如果以某个位置 i 为中心，我们有一个长度为 l 的回文串，那么我们有以 i 为中心的长度为 $l-2$，$l-4$ 等等的回文串。所以 $d_1[i]$ 和 $d_2[i]$ 两个数组已经足够表示字符串中所有子回文串的信息。

一个令人惊讶的事实是，存在一个复杂度为线性并且足够简单的算法计算上述两个“回文性质数组” $d_1[]$ 和 $d_2[]$。在这篇文章中我们将详细地描述该算法。

解法

总的来说，该问题具有多种解法：应用字符串哈希，该问题可在 $O(n\log n)$ 时间内解决，而使用后缀数组和快速 LCA 该问题可在 $O(n)$ 时间内解决。

但是这里描述的算法 **压倒性** 地简单，并且在时间和空间复杂度上具有更小的常数。该算法由 **Glenn K. Manacher** 在 1975 年提出。

朴素算法

为了避免在之后的叙述中出现歧义，这里我们指出什么是“朴素算法”。

该算法通过下述方式工作，对每个中心位置 i，在比较一对对应字符后，只要可能，该算法便尝试将答案加 1。

该算法是比较慢的：它只能在 $O(n^2)$ 的时间内计算答案。

该朴素算法的实现如下：

```
vector<int> d1(n), d2(n);
```

```
for (int i = 0; i < n; i++) {
  d1[i] = 1;
  while (0 <= i - d1[i] && i + d1[i] < n && s[i - d1[i]] == s[i + d1[i]]) {
    d1[i]++;
  }

  d2[i] = 0;
  while (0 <= i - d2[i] - 1 && i + d2[i] < n &&
         s[i - d2[i] - 1] == s[i + d2[i]]) {
    d2[i]++;
  }
}
```

Manacher 算法

这里我们将只描述算法中寻找所有奇数长度子回文串的情况，即只计算 $d_1[]$寻找所有偶数长度子回文串的算法（即计算数组 $d_2[]$将只需对奇数情况下的算法进行一些小修改。

为了快速计算，我们维护已找到的最靠右的子回文串的 **边界** (l,r)即具有最大 r 值的回文串，其中 l 和 r 分别为该回文串左右边界的位置）。初始时，我们置 $l=0$ 和 $r=-1$

现在假设我们要对下一个 i 计算 $d_1[i]$而之前所有 $d_1[]$ 中的值已计算完毕。我们将通过下列方式计算：

- 如果 i 位于当前子回文串之外，即 $i>r$那么我们调用朴素算法。因此我们将连续地增加 $d_1[i]$同时在每一步中检查当前的子串 $[i-d_1[i]\ldots i+d_1[i]]$ 是否为一个回文串。如果我们找到了第一处对应字符不同，又或者碰到了 s 的边界，则算法停止。在两种情况下我们均已计算完 $d_1[i]$此后，仍需记得更新 (l,r)
- 现在考虑 $i\le r$ 的情况。我们将尝试从已计算过的 $d_1[]$ 的值中获取一些信息。首先在子回文串 (l,r) 中反转位置 i即我们得到 $j=l+(r-i)$现在来考察值 $d_1[j]$因为位置 j 同位置 i 对称，我们 **几乎总是** 可以置 $d_1[i]=d_1[j]$该想法的图示如下（可认为以 j 为中心的回文串被“ 拷贝 ”至以 i 为中心的位置上）：

$$ \ldots\ \overbrace{ s_l\ \ldots\ \underbrace{ s_{j-d_1[j]+1}\ \ldots\ s_j\ \ldots\ s_{j+d_1[j]-1} }_\text{palindrome}\ \ldots\ \underbrace{ s_{i-d_1[j]+1}\ \ldots\ s_i\ \ldots\ s_{i+d_1[j]-1} }_\text{palindrome}\ \ldots\ s_r }^\text{palindrome}\ \ldots $$

然而有一个 **棘手的情况** 需要被正确处理：当“ 内部 ”的回文串到达“ 外部 ”回文串的边界时，即 $j-d_1[j]+1\le l$或者等价地说$i+d_1[j]-1\ge r$因为在“ 外部 ”回文串范围以外的对称性没有保证，因此直接置 $d_1[i]=d_1[j]$ 将是不正确的：我们没有足够的信息来断言在位置 i 的回文串具有同样的长度。

实际上，为了正确处理这种情况，我们应该“ 截断 ”回文串的长度，即置 $d_1[i]=r-i$之后我们将运行朴素算法以尝试尽可能增加 $d_1[i]$ 的值。

该种情况的图示如下（以 j 为中心的回文串已经被截断以落在“ 外部 ”回文串内 $$ \ldots\ \overbrace{ \underbrace{ s_l\ \ldots\ s_j\ \ldots\ s_{j+(j-l)} }_\text{palindrome}\ \ldots\ \underbrace{ s_{i-(r-i)}\ \ldots\ s_i\ \ldots\ s_r }_\text{palindrome} }^\text{palindrome}\ \underbrace{ \ldots \ldots \ldots \ldots \ldots }_\text{try moving here} $$

该图示显示出，尽管以 j 为中心的回文串可能更长，以至于超出“ 外部 ”回文串，但在位置 i我们只能利用其完全落在“ 外部 ”回文串内的部分。然而位置 i 的答案可能比这个值更大，因此接下来我们将运行朴素算法来尝试将其扩展至“ 外部 ”回文串之外，也即标识为 “try moving here” 的区域。

最后，仍有必要提醒的是，我们应当记得在计算完每个 $d_1[i]$ 后更新值 (l,r)

同时，再让我们重复一遍：计算偶数长度回文串数组 $d_2[]$ 的算法同上述计算奇数长度回文串数组 $d_1[]$ 的算法十分类似。

From:
https://wiki.cvbbacm.com/ - **CVBB ACM Team**

Permanent link:
https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal_string:lgwza:manacher_%E7%AE%97%E6%B3%95&rev=1601618694

Last update: **2020/10/02 14:04**