后缀数组

MengChunlei

February 28, 2021

1 定义

对于一个长度为 n 的字符串 $s: s[0], s[1], ..., s[n-2], s[n-1]$, 它对应的后缀数组也是一个长度为 n 的数组 sa. 其中 $sa[i]$ 表示在所有 n 个后缀中, 排第 i 名的后缀是哪个 (第一个字母在 s 中的位置). 比如 s=aabaaaab, 那么对应的 sa 为:

$index$	0	1	2	3	4	5	6	7
s	a	a	b	a	a	a	a	b
sa	3	4	5	0	6	1	7	2

第 0 名: aaaab
第 1 名: aaab
第 2 名: aab
第 3 名: aabaaaab
第 4 名: ab
第 5 名: abaaaab
第 6 名: b
第 7 名: baaaab

另外还有一个数组称之为 rank 数组 rk, 它的长度也是 n,$rk[i]$ 表示以 $s[i]$ 开始的后缀的排名. 那么有以下关系: $sa[rk[i]] = rk[sa[i]] = i$

2 一般做法

很容易想到 $n^2log(n)$ 的做法. 代码如下:

Listing 1: Normal

```
std::vector<int> ComputeSA1(const std::string &s) {
  int n = static_cast<int>(s.size());
  std::vector<int> sa(n);
  for (int i = 0; i < n; ++i) {
    sa[i] = i;
  }
  std::sort(sa.begin(), sa.end(), [&](int p1, int p2) {
    for (; p1 < n && p2 < n && s[p1] == s[p2]; ++p1, ++p2);
    return p1 < n && p2 < n ? s[p1] < s[p2] : p1 == n;
  });
  return sa;
}
```

3 改进做法

优化的思路是这样的: 设 $rk_w[i]$ 表示以 i 开始的长度为 w 的子串的排名, 即 $s[i] \to s[i+w-1]$. 那么对于 rk_{2w} 来说, 它的计算可以借助于 rk_w 来进行. 简答来说, 就是以 $rk_w[i]$ 为第一关键字, 以 $rk_w[i+w]$ 为第二关键字进行排序, 就可以求出 rk_{2w} 数组.

Listing 2: Better

```
std::vector<int> UpdateRank(const std::vector<int> &sa,
                            const std::vector<int> &old_rank, int n, int w) {
  std::vector<int> rk(n);
  rk[sa[0]] = 0;
  for (int p = 0, i = 1; i < n; ++i) {
    if (old_rank[sa[i]] == old_rank[sa[i - 1]] &&
        ((sa[i] + w >= n && sa[i - 1] + w >= n) ||
         (sa[i] + w < n && sa[i - 1] + w < n &&
          old_rank[sa[i] + w] == old_rank[sa[i - 1] + w]))) {
      rk[sa[i]] = p;
    } else {
      rk[sa[i]] = ++p;
    }
  }
  return rk;
}
std::vector<int> ComputeSA2(const std::string &s) {
  int n = static_cast<int>(s.size());
  std::vector<int> sa(n);
  std::vector<int> rk(n);
  for (int i = 0; i < n; ++i) {
    sa[i] = i;
    rk[i] = s[i];
  }
  for (int w = 1; w < n; w <<= 1) {
    std::sort(sa.begin(), sa.end(), [&](int x, int y) {
      return rk[x] == rk[y]
                 ? (x + w < n && y + w < n ? rk[x + w] < rk[y + w] : x + w >= n)
                 : rk[x] < rk[y];
    });
    rk = UpdateRank(sa, rk, n, w);
  }
  return sa;
}
```

这个算法的复杂度为 $O(nlog^2(n))$. 接下来, 进一步的改进需要借助于计数排序和基数排序. 先了解下.

4 计数排序

假设一个待排列的序列中所有的元素只有 C 种, 那么可以通过计算每种元素的个数进行排列. 假设排列的元素有 n 个. 这个的复杂度为 $O(n+C)$. 这是一个稳定排序算法. 示例代码如下:

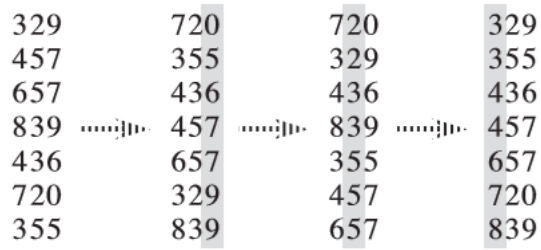
Listing 3: Counting Sort

```
template <typename Elem>
std::vector<Elem> CountingSort(const std::vector<Elem> &eles,
                               std::function<int(const Elem &)> functor,
                               int C) {
  int n = static_cast<int>(eles.size());
  std::vector<int> bucket(C, 0);
  for (int i = 0; i < n; ++i) {
    ++bucket[functor(eles[i])];
  }
  for (int i = 1; i < C; ++i) {
    bucket[i] += bucket[i - 1];
  }
  std::vector<Elem> result(n);
```

```
  for (int i = n - 1; i >= 0; --i) {
    result[--bucket[functor(eles[i])]] = eles[i];
  }
  return result;
}
```

5 基数排序

如果待排序的元素有 k 个关键字, 可以先对第 k 个关键字进行稳定排序, 然后再对第 $k-1$ 个元素进行稳定排序, 以此类推, 最后对第 1 个关键字进行稳定排序. 如下图所示:



下面是代码示例:

Listing 4: Radix Sort

```
template <typename Elem>
std::vector<Elem>
RadixSort(const std::vector<Elem> &eles, int k, const std::vector<int> &cs,
          std::function<int(const Elem &, int index)> functor) {
  std::vector<Elem> result = eles;
  for (int i = k - 1; i >= 0; --i) {
  result = CountingSort<Elem>(
      result, std::bind(functor, std::placeholders::_1, i), cs[i]);
  }
  return result;
}
```

6 最后的改进

对第三步中 $nlog^2(n)$ 的方法进行进行改进. 可以将循环内的每次排序看作是两个关键字的排序, 先对第二个关键字排序, 再对第一个关键字排序.

Listing 5: Last Improvement

```
std::vector<int> ComputeSA3(const std::string &s) {
  int n = static_cast<int>(s.size());
  int m = std::max(256, n) + 1;
  std::vector<int> sa(n);
  std::vector<int> rk(n);
  for (int i = 0; i < n; ++i) {
    sa[i] = i;
    rk[i] = s[i];
  }
  for (int w = 1; w < n; w <<= 1) {
    sa = RadixSort<int>(sa, 2, {m, m}, [&](const int &x, int index) {
      return index == 0 ? rk[x] : (x + w >= n ? 0 : rk[x + w] + 1);
    });
    rk = UpdateRank(sa, rk, n, w);
  }
  return sa;
}
```

这个算法的复杂度为 $O(nlog(n))$

7 一些优化

7.1 第二关键字排序

第二关键字的排序可以简化为如下实现:

Listing 6: Second Key Sort

```
int p = 0;
for (int i = n - 1; i >= n - w; --i) {
  id[p++] = i;
}
for (int i = 0; i < n; ++i) {
  if (sa[i] >= w) {
    id[p++] = sa[i] - w;
  }
}
```

7.2 记录 CountingSort 的结果

在 *CountingSort* 中记录 *functor* 的结果, 避免第二次调用

7.3 优化计数排序 C 的大小

每次 C 的大小就是可以设置为 *UpdateRank* 中 p 的大小, 而不是每次都是 m.
加上所有优化后的代码如下:

Listing 7: Best

```
std::vector<int> UpdateRankFiner(const std::vector<int> &sa,
                                 const std::vector<int> &old_rank, int n, int w,
                                 int *last_p) {
  std::vector<int> rk(n);
  rk[sa[0]] = 0;
  int p = 0;
  for (int i = 1; i < n; ++i) {
    if (old_rank[sa[i]] == old_rank[sa[i - 1]] &&
        ((sa[i] + w >= n && sa[i - 1] + w >= n) ||
         (sa[i] + w < n && sa[i - 1] + w < n &&
          old_rank[sa[i] + w] == old_rank[sa[i - 1] + w]))) {
      rk[sa[i]] = p;
    } else {
      rk[sa[i]] = ++p;
    }
  }
  *last_p = p + 1;
  return rk;
}

template <typename Elem>
std::vector<Elem> CountingSortFiner(const std::vector<Elem> &eles,
                                    std::function<int(const Elem &)> functor,
                                    int C) {
  int n = static_cast<int>(eles.size());
  std::vector<int> bucket(C, 0);
  std::vector<int> bucket_result(n);
  for (int i = 0; i < n; ++i) {
    ++bucket[bucket_result[i] = functor(eles[i])];
  }
  for (int i = 1; i < C; ++i) {
    bucket[i] += bucket[i - 1];
  }
  std::vector<Elem> result(n);
  for (int i = n - 1; i >= 0; --i) {
    result[--bucket[bucket_result[i]]] = eles[i];
```

```
  }
  return result;
}

std::vector<int> ComputeSA4(const std::string &s) {
  int n = static_cast<int>(s.size());
  int m = std::max(256, n) + 1;
  std::vector<int> sa(n);
  std::vector<int> rk(n);
  std::vector<int> cnt(m, 0);
  for (int i = 0; i < n; ++i) {
    ++cnt[rk[i] = s[i]];
  }
  for (int i = 1; i < m; ++i) {
    cnt[i] += cnt[i - 1];
  }
  for (int i = n - 1; i >= 0; --i) {
    sa[--cnt[rk[i]]] = i;
  }
  std::vector<int> id(n);
  for (int w = 1; w < n; w <<= 1) {
    int p = 0;
    for (int i = n - 1; i >= n - w; --i) {
      id[p++] = i;
    }
    for (int i = 0; i < n; ++i) {
      if (sa[i] >= w) {
        id[p++] = sa[i] - w;
      }
    }
    sa = CountingSortFiner<int>(id, [&](const int &x) { return rk[x]; }, m);
    rk = UpdateRankFiner(sa, rk, n, w, &m);
  }
  return sa;
}
```

8 height 数组

令 $suf[i]$ 表示后缀 $s[i] \to s[n-1]$
$height$ 数组是一个长度为 n 的数组, $height[i] = lcp(suf[sa[i]], suf[sa[i-1]])$, 即它表示排名第 i 的后缀和排名第 $i-1$ 的后缀的最长公共前缀. $height[0] = 0$
$height$ 数组有一个性质: $height[rk[i]] \geq height[rk[i-1]] - 1$. 证明如下:
设 $suf[i-1] = aAD$,(A, D 分别表示一个串, 可能为空, a 表示一个字符). 那么 $suf[i] = AD$.
另外假设 $suf[sa[rk[i-1]-1]] = aAB$,(注意 $B < D$), 所以有 $lcp(suf[i-1], suf[sa[rk[i-1]-1]]) = aA$. 也就是 $height[rk[i-1]]$ 的长度是 $|aA|$. 所以 $height[rk[i-1]] - 1$ 就是 A 的长度.
由于 $suf[i] = AD$, 且 $suf[sa[rk[i-1]-1]+1] = AB$, $AB < AD$, 所以 $suf[i]$ 跟 $suf[rk[i]-1]$ 的最长公共前缀至少是 A. 即 $height[rk[i]] \geq |A| = height[rk[i-1]] - 1$
借由此结论, 有 $O(n)$ 的方法计算 $height$ 数组:

Listing 8: ComputeHeight

```
std::vector<int> ComputeHeight(const std::string &s,
  const std::vector<int> &sa) {
  int n = static_cast<int>(s.size());
  std::vector<int> rk(n);
  for (int i = 0; i < n; ++i) {
    rk[sa[i]] = i;
  }
  std::vector<int> height(n, 0);
  for (int i = 0, k = 0; i < n; ++i) {
    if (rk[i] == 0) {
      continue;
    }
```

```
13      if (k > 0) {
14        --k;
15      }
16      while (i + k < n && sa[rk[i] - 1] + k < n &&
17             s[i + k] == s[sa[rk[i] - 1] + k]) {
18        ++k;
19      }
20      height[rk[i]] = k;
21    }
22    return height;
23  }
```