최소 원형 스트링

위규범[†]·예홍진[†]

요 약

주어진 문자열을 환형 이동한 문자열들 중에서 사전적 순서로 가장 작은 것을 찾는 선형 시간 알고리즘을 제시한다. 이 문제는 등방성의 셀룰러 오토마타의 상태전이함수의 효율적인 구현에서 생긴다. 이 문제에 대한 단순한 알고리즘은 이차시간을 필요로 한다. 본 논문에서 제안하는 알고리즘은 부분스트링들의 비교 결과를 저장하고 나중에 같은 계산이 필요한 경우에 저장한 결과를 다시 사용함으로서 선형시간에 문제를 해결한다.

Minimal Circular Strings

Kyubum Wee[†]· Hong-Jin Yeh[†]

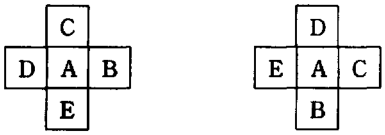
ABSTRACT

We present a linear time algorithm for finding a lexicographically minimal circular string in a given string. The problem was motivated by an effort to implement state transition functions in isotropic cellular automata. A naive algorithm for the problem would require quadratic time. The proposed algorithm runs in linear time by keeping the result of comparisons of substrings and reusing it afterwards when the same computation is needed.

1. Introduction

We consider the problem of finding a minimal circular string in a given string. More formally, given a string $s=a_1a_2\cdots a_n$ we want to find a position i such that the circular string $a_ia_{i+1}\cdots a_na_1a_2\cdots a_{i-1}$ is lexicographically smaller than or equal to any other circular string $a_ja_{j+1}\cdots a_na_1a_2\cdots a_{j-1}$ for $1\leq j\leq n$.

This problem was motivated by an effort to implement state transition functions of cellular automata [1]. A cell in cellular automata changes its state at the next time step based on its state and its neighbors' states at the current time step. Since cellular automata are isotropic, the neighbors' directions are irrelevant in det-ermining the next state. For example, consider the following figure.



(그림 1) 회전 대칭
(Fig. 1) rotational symmetry

Suppose that the cell at the center changes its state from the state A into the state D at the next

[†] 종신회원 : 아주대학교 정보 및 컴퓨터공학부 교수

논문접수 : 1998년 3월 25일, 심사완료 : 1998년 7월 13일

time step. Then there should be an entry (ACBED, D) in the table representing the state transition function of the celluar automata. Since CA is isotropic, the table needs not contain both (ACBED, D) and (ADCBE, D). Only one of them needs to be included in the table. If we decide to include (ACBED, D) in the table, then neither of (ADCBE, D), (ABEDC, D), and (AEDCB, D) needs to be included in the table. In other words, only one of the cyclic shifts of the sequence of neighbors′ states need go into the table.

Also when we simulate the behavior of the cellular automata, any of the above four configurations ACBED, ADCBE, ABEDC, and AEDCB should yield the next state D. So which one should be the representative of all the cyclic shifts? A natural choice would be the one that is smallest in the lexicographic ordering. In the above example, BEDC is the smallest among CBED, BEDC, EDCB, and DCBE.

A naive algorithm would compare n different sequences to find the smallest, where n is the length of the sequence. It is easy to see that such an algorithm needs quadratic time in the worst case. Here we present a linear time algorithm. Previous research results on this problem are by Booth[2] and by Shiloach[3]. Booth gave a linear time algorithm by generalizing the KMP string matching algorithm [4]. Shiloach's algorithm finds all the minimal circular strings. Both of these are complicated algorithms that are hard to understand. Our algorithm is not more efficient than the previous algorithms, but it uses a different approach, is simple and easy to understand, and fairly efficient.

2. Minimal Circular String Algorithm

Let us use the notation $u < v$ when the string u is lexicographically less than the string v, and $u \leq v$ when the string u is lexicographically less than or equal to the string v.

First we introduce a lemma. The lemma says that in a string that repeats a prefix, the circularly shifted string that starts with the second repetition of the prefix cannot be the minimal circular string unless it is identical to the original string.

Lemma In a string u of the form $u = xxw$, where x and w are substrings, let v be the circularly shifted string $v = xwx$. If $u \neq v$, then v cannot be the minimal circular string.

proof Since $u \neq v$, either $u < v$ or $v < u$. In the case $u < v$, v clearly is not a minimal circular string. In the case $v < u$, $xwx < xxw$. Hence $wx < xw$ by eliminating the prefix x from both sides of the inequality. Then by appending the string x to both sides, we get $wxx < xwx = v$. But wxx is also a circularly shifted string of u. Therefore v cannot be a minimal circular string in this case, either. □

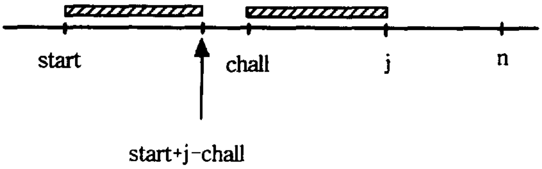
Notation The notation A[k:m] is used to represent the contiguous part of the array from A[k] to A[m], and A[k:*] the circular string starting from the position k and ending at the position k-1. □

Now we present an algorithm for finding a minimal circular string in a given string. The algorithm takes the input string in an array A[1..n] and returns the starting position of the minimal circular string.

Note that the problem of finding the minimal circular string of w is the same as the problem of finding the minimal substring of length $|w|$ in the string ww. So first we double the array A[1..n] into A[1..2n] in such a way that A[1..n] = A[n+1..2n]. Now the problem is finding the lexicographically minimal string of length n in A[1..2n].

We use the variable *j* to scan the array A[1..2n]. The variable start is used to keep track of the starting position of the minimal sequence in A[1:*j*]. The variable *chall* is used to maintain the starting position of the challenger string. That is, A[*start* :

start+j-chall] = A[*chall* : *j*]. See figure 2. For example, if A[1:*j*] = (3 4 2 4 3 6 5 2 4 3), then *start* = 3 and *chall* = 8. Here note that A[*start* : *start+j-chall*] = A[*chall* : *j*] = (2 4 3). If there is no challenger in A[1:*j*], *chall* is set to 0. The challenger has the possibility of turning out to be smaller than the currently minimal sequence.



(그림 2)
(Fig. 2)

The array P[1..2n] remembers the starting position of the challenger when the champion(*start*) and the challenger(*chall*) are compared. In more detail, P[*j*] is set to *chall* when A[*start+j-chall*] and A[*j*] are compared. P is used for not having to backtrack the scanning varible *j*, when the challenger becomes the champion and the new challenger is looked for.

start is initialized to 1, *chall* to 0, and *j* to 2. As the scanning variable *j* is incremented, *start* and *chall* need to be adjusted. We consider the following five cases:

(case 1) There was no challenger in A[1:*j*-1].
(case 2) There was a challenger in A[1:*j*-1] and A[*start*] > A[*j*].
(case 3) There was a challenger in A[1:*j*-1] and A[*start*] ≤ A[*j*] and A[*start+j-chall*] < A[*j*]
(case 4) There was a challenger in A[1:*j*-1] and A[*start*] ≤ A[*j*] and A[*start+j-chall*] > A[*j*]
(case 5) There was a challenger in A[1:*j*-1] and A[*start*] ≤ A[*j*] and A[*start+j-chall*] = A[*j*]

Before considering each of the five cases, let us mention that the algorithm sees to it that the intervals A[*start* : *start+j-chall*] and A[*chall* : *j*] never overlap, by resetting *chall* to 0 when the two intervals have grown to meet. Two intervals meet when *start+j-chall* = *chall*, that is 2 * *chall* = *start* + *j*.

When the two intervals meet, we know that A[*start* : *start+j-chall*] = A[*chall* : *j*] and that they are adjacent. Hence, by the above lemma, it is safe to give up A[*chall*:*] as a candidate for the minimal circular string. So resetting *chall* to 0 is justfied.

Now we will consider each of the five cases.

(case 1) There was no challenger in A[1:*j*-1].

If A[*start*] > A[*j*], then set *start* to *j*. If A[*start*] = A[*j*], then set *chall* to *j*. If A[*start*] < A[*j*], then there is nothing to do. □

(case 2) There was a challenger in A[1:*j*-1] and A[*start*] > A[*j*].

Since A[*start*] > A[*j*], *j* becomes the new starting position. There is no challenger at the moment. □

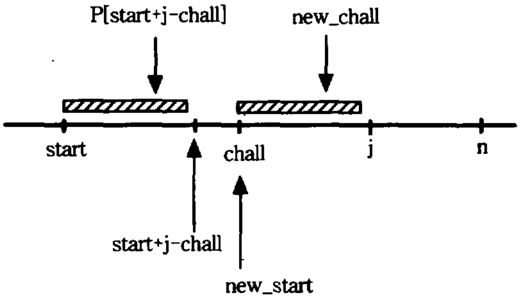
(case 3) There was a challenger in A[1:*j*-1] and A[*start*] ≤ A[*j*] and A[*start+j-chall*] < A[*j*].

Since there was a challenger in A[1:*j*-1], we know that A[*start* : *start+j*-1-*chall*] = A[*chall* : *j*-1]. Since A[*start+j-chall*] < A[*j*], A[*start* : *start+j-chall*] < A[*chall* : *j*]. Hence A[*chall* : *j*] cannot be a challenger. So *chall* is set to 0. P[*j*] is set to *chall*. P[*j*] remembers that the current challenger lost at position *j*. □

(case 4) There was a challenger in A[1:*j*-1] and A[*start*] ≤ A[*j*] and A[*start+j-chall*] > A[*j*].

Since there was a challenger in A[1:*j*-1], we know that A[*start* : *start+j*-1-*chall*] = A[*chall* : *j*-1]. Since A[*start+j-chall*] > A[*j*], A[*start* : *start+j-chall*] > A[*chall* : *j*]. Hence the challenger wins and becomes *start*.

Now who is the new challenger? The new challenger is the first position in A[*chall*+1 : *j*] whose value is the same as A[*chall*], if there is any. In order to find the position of the new challenger, we do not have to compare each element in A[*chall*+1 : *j*] to A[*chall*]. Instead we take advantage of the fact that A[*start* : *start*+*j*-1-*chall*] = A[*chall* : *j*-1]. We check the value of P[*start*+*j*-*chall*]. If P[*start*+*j*-*chall*] = 0, then it means that there was no challenger whe'n we were scanning A[*start*+*j*-*chall*]. Hence there is no new challenger, either. If P[*start*+*j*-*chall*] ≠ 0, then it means that there was a challenger when we were scanning A[*start*+*j*-*chall*] and that the starting position of the challenger was P[*start*+*j*-*chall*]. Hence the position of the new challenger is P[*start*+*j*-*chall*] + (*chall*-*start*). See figure 3.



(그림 3)
(Fig. 3)

In this case, note that the scanning variable *j* should not be incremented. We know that A[new*start* : new*start*+*j*-1-new*chall*] = A[new*chall* : *j*-1]. So we should start comparing from A[new*start*+*j*-new*chall*] and A[j]. □

(case 5) There was a challenger in A[1:*j*-1] and A[*start*] ≤ A[*j*] and A[*start*+*j*-*chall*] = A[*j*].

Since there was a challenger in A[1:*j*-1], we know that A[*start* : *start*+*j*-1-*chall*] = A[*chall* : *j*-1]. Since A[*start*+*j*-*chall*] = A[*j*], A[*start* : *start*+*j*-*chall*] = A[*chall* : *j*]. Hence the competition between *start* and *chall* has not been resolved yet. So they should be again compared at the next repetition of the loop with *j* incremented. The algorithm only has to remember the starting position of the current challenger by setting P[*j*] to *chall*.

But there is one thing to worry about : It could be that A[*start*] = A[*j*]. In such a case, should we keep the position *j* as another challenger? Fortunately, that is not necessary. The reason is as follows : First, note that A[*start*] = A[*j*] = A[*start*+*j*-*chall*]. Hence the position *start*+*j*-*chall* must have been a challenger bef-ore, but is not a challenger any more. Now let us consider two cases : (i) $A[start+j-chall : *] \le A[j : *]$ and (ii) $A[start+j-chall : *] > A[j : *]$.

(case i) $A[start+j-chall : *] \le A[j : *]$.

The position *start*+*j*-*chall* was a challenger before, but is not any more, which means it has been determined that $A[start+j-chall : *]$ cannot be a minimal circular string. But $A[j : *]$ is even greater than or equal to $A[start+j-chall : *]$. Hence $A[j : *]$ cannot be a minimal circularstring, either. So it is safe not to keep the position *j* as another challenger.

(case ii) $A[start+j-chall : *] > A[j : *]$.

Recall that A[*start* : *start*+*j*-*chall*] = A[*chall* : *j*]. By concatenating strings we can see that $A[start : start+j-chall]A[start+j-chall : *] > A[chall : j]A[j : *]$. Hence $A[start : *] > A[chall : *]$. Therefore the current challenger will beat the champion(*start*) eventually. Then *chall* will be the new *start*, and *j* will be the new challenger. So we don't have to keep *j* as another challenger this time. It will be considered later. □

Now the algorithm follows:

Input : an array A[1..*n*] of symbols { $n \ge 2$ }

Output : the starting position *start* of a minimal circular string.

```
(1) for i := 1 to 2n do P[i] := 0
(2) for i := 1 to n do A[n+i] := A[i]
(3) start := 1; chall := 0; j := 2
(4) while (j≤n) or ((chall≠0) and (chall≤n))
(5)     if chall = 0 then                    {case 1}
```

```
      if A[start] > A[j] then start := j
      else if A[start] = A[j] then chall := j
      j := j + 1
      else if A[start] > A[j] then        {case 2}
         start := j; chall := 0; j := j + 1
      else if A[start+j-chall)] < A[j] then {case 3}
         P[j] := chall; chall := 0; j := j + 1
      else if A[start+j-chall] > A[j] then {case 4}
         dist := chall - start
         start := chall
         if P[j-dist] = 0 then
            chall := 0
         else
            chall := P[j-dist]+dist
            if chall = start then chall := 0
      else                                {case 5}
         P[j] := chall; j := j + 1
      if 2 * chall = start + j then chall := 0
                                     {by the lemma}
return start
```

3. The Efficiency of the Algorithm

Now we will show that the loop terminates, and does so in no later than $2n$ repetitions. First, note that *start* + j always increases : At each repetition of the loop, j always increases except one case, that is, case 4. In this case, j is stationary, while *start* increases by advancing to the position that used to be the challenger. Hence *start* + j always increases at least by one at each repetition of the loop. Since j is always ahead of *start*, j is greater than (*start* +j)/2. Therefore after $2n$ repetitions of the loop, j gets greater than n.

Observe that *chall* is also greater than (*start*+j)/2 unless *chall* is zero. It is because, as we observed above, when the interval A[*start* : *start*+j-*chall*] has grown up to meet the interval A[*chall* : j], *chall* is reset to 0. Hence *chall* is always to the right of the middle point of *start* and j unless it is 0. Hence after $2n$ repetitions of the loop, *chall* either gets greater than n or is zero. Therefore, the loop does terminate no later than $2n$ steps. Thus the proposed algorithm's time complexity is $O(n)$.

4. Conclusions

We presented an efficient algorithm for finding the lexicograhically smallest sequence among all the cyclically shifted sequences of a given sequence. This algorithm was motivated by an effort to efficiently implement the state transition functions of isotropic cellular automata, but we expect other applications of this algorithm in the problems involving sequences.

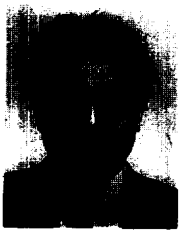
References

[1] C. C. Langton, "Studying artificial life with cellular automata", Physica 22D, pp.120-140, 1986.

[2] K. S. Booth, "Lexicographically least circular substrings", Inform. Process. Lett. 10(4), pp.240-242, 1980.

[3] Y. Shiloach, "Fast canonization of circular strings", J. of Algorithms 2, pp.107-121, 1981.

[4] D. Knuth, J. Morris, and V. Pratt, "Fast pattern matching in strings", SIAM J. Comput. 6(2), pp. 323-350, 1977.



위 규 범

1978년 서울대학교 수학과(학사)

1984년 University of Wisconsin -.Madison 전산학과(이학석사)

1992년 Indiana University - Bloomington 전산학과(이학박사)

1993년~현재 아주대학교 정보및컴퓨터공학부 조교수

관심분야 : 컴퓨터 이론

예 홍 진

1986년 서울대학교 수학교육과(이학사)

1988년 아주대학교 전자계산학과(공학석사)

1990년 UJF - Grenoble 1 대학교 응용수학과(D.E.A.)

1993년 UCB - Lyon 1 대학교 전자계산학과(공학박사)

1993년~현재 아주대학교 정보통신대학 조교수

관심분야 : 컴퓨터 계산, 병렬 알고리즘, 병렬 VLSI 알고리즘