Linear-Time Online Algorithm Inferring the Shortest Path from a Walk

Shintaro Narisada, Diptarama Hendrian, Ryo Yoshinaka(✉), and Ayumi Shinohara

Graduate School of Information Sciences, Tohoku University, Sendai, Japan
shintaro_narisada@shino.ecei.tohoku.ac.jp,
{diptarama,ryoshinaka,ayumis}@tohoku.ac.jp

Abstract. We consider the problem of inferring an edge-labeled graph from the sequence of edge labels seen in a walk of that graph. It has been known that this problem is solvable in $O(n \log n)$ time when the targets are path or cycle graphs. This paper presents an online algorithm for the problem of this restricted case that runs in $O(n)$ time, based on Manacher's algorithm for computing all the maximal palindromes in a string.

Keywords: Graph inference · Walk · Palindrome

1 Introduction

Aslam and Rivest [2] proposed the problem of *minimum graph inference from a walk*. Let us consider an edge-labeled undirected (multi)graph G. A *walk* of G is a sequence of edges $e_1, \ldots, e_n$ such that each e_i connects v_{i-1} and v_i for some (not necessarily pairwise distinct) vertices $v_0, v_1, \ldots, v_n$. The *output* of the walk is the sequence of the labels of those edges. For a string w, *minimum graph inference from a walk* is the problem to compute a graph G with the smallest number of *vertices* such that w is the output of a walk of G. We give an example in Fig. 1. With no assumption on graphs to infer, trivially the graph with a single vertex with self-loops labeled with all output symbols is always minimum. The problem has been studied for different graph classes in the literature.

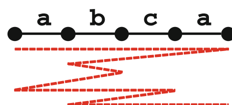


Fig. 1. Minimum path graph that has abcaacbbbaabccbbca as a walk output

S. Narisada—Currently affiliated with KDDI Corporation, Tokyo, Japan.

© Springer Nature Switzerland AG 2018
T. Gagie et al. (Eds.): SPIRE 2018, LNCS 11147, pp. 311–324, 2018.
https://doi.org/10.1007/978-3-030-00479-8_25

Aslam and Rivest [2] proposed polynomial time algorithms for the minimum graph inference problem for path graphs and cycle graphs, which include the variant of minimum path graph inference where a walk must start from an end of a path graph and end in the other end (Table 1), which we call an *end-to-end* walk. Raghavan [6] studied the problem further and showed that both minimum path and cycle graph inference from walk can be reduced to path graph inference from an end-to-end walk in $O(n)$ time. Moreover, he presented an $O(n \log n)$ time algorithm for inferring minimum path/cycle graph from a walk, while showing inferring minimum graph with bounded degree k is NP-hard for any $k \geq 3$. Maruyama and Miyano [4] strengthened Raghavan's result so that inferring minimum tree with bounded degree k is still NP-hard for any $k \geq 3$. On the other hand, Maruyama and Miyano [5] showed that it is solvable in linear time when trees have no degree bound. They also studied a variant of the problem where the input consists of multiple path labels rather than a single walk label, which was shown to be NP-hard. Akutsu and Fukagawa [1] considered another variant, where the input is the numbers of occurrences of vertex-labeled paths. They showed a polynomial time algorithm with respect to the size of output graph, when the graphs are trees of unbounded degree and the lengths of given paths are fixed. They also proved that the problem is strongly NP-hard even when the graphs are planar of unbounded degree.

Table 1. Time complexity of minimum graph inference bounded degree 2 from a walk

Algorithms	Connected graph bounded degree 2		
	Path		Cycle
	End-to-end walk	General walk	
Aslam and Rivest [2]	$O(n^3)$	$O(n^3)$	$O(n^5)$
Raghavan [6]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Proposed	$O(n)$	$O(n)$	$O(n)$

This paper focuses on the problem on graphs of bounded degree 2, i.e., path and cycle graphs. We propose a linear-time online algorithm that infers a minimum path graph from an end-to-end walk. Thanks to Raghavan's result [6], this entails that one can infer a minimum path/cycle graph in linear time from a walk, which is not necessarily end-to-end. Aslam and Rivest [2] showed that the minimum path graphs that have end-to-end walks xyy^Ryz and xyz coincide, where x, y, z are label strings and y^R is the reverse of y. Let us call a nonempty string of the form yy^Ry a *Z-shape*. Their result implies that to obtain the minimum path graph of a label string, one can repeatedly contract an arbitrary occurrence of a Z-shape yy^Ry to y until the sequence contains no such substring. Then the finally obtained string is just the sequence of labels of the edges of the minimum path graph. Raghavan [6] achieved an $O(n \log n)$ time algorithm by introducing a sophisticated order of rewriting, which always contract the smallest Z-shapes

in the sequence. We follow their approach of repetitive contraction of Z-shapes but with a different order. The order we take might appear more naive; We read letters of the input string one by one and always contract the firstly found Z-shape. This approach makes our algorithm online. Apparently finding Z-shapes is closely related to finding palindromes. Manacher [3] presented a linear-time "online" algorithm that finds all the maximal palindromes in a string. To realize linear-time Z-shape elimination, we modify Manacher's algorithm for Z-shape detection and elimination. Our experimental results show that our algorithm is faster than Raghavan's in practice, too.

2 Preliminaries

For a tuple $\boldsymbol{e} = (e_1, \ldots, e_m)$ of elements, we represent $(e_0, e_1, \ldots, e_m)$ by $e_0; \boldsymbol{e}$ or $(e_0; \boldsymbol{e})$. For two integers i, j, we define $[i : j] = \{ k \mid i \leq k \leq j \}$.

Let Σ be an alphabet. A sequence of elements of Σ is called a *string* and the set of strings is denoted by Σ^*. The empty string is denoted by ε and the set of nonempty strings is $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For a string $w = xyz$, x, y, and z are called a *prefix*, a *substring*, and a *suffix* of w, respectively. The length of w is denoted by $|w|$. The i-th letter of w is denoted by $w[i]$ for $1 \leq i \leq |w|$. For $1 \leq i \leq j \leq |w|$, $w[i : j]$ represents $w[i] \ldots w[j]$. The reversed string of w is denoted by $w^R = w[|w|] \cdots w[1]$. The string repeating w k times is w^k.

A string y is called an *even palindrome* if $y = xx^R$ for a string $x \in \Sigma^*$. The *radius* of y is $r = |x|$. We will call an even palindrome simply a palindrome, because we consider only even palindromes in this paper. When y occurs as a substring $w[i : j]$ of a string w, the position $c = i + r - 1$ is called the *center* (of the occurrence) of y. Especially, y is said to be the *maximal palindrome* centered at c iff either $i = 1$, $j = |w|$, or $w[i-1] \neq w[j+1]$. By $\rho_w(c)$ we denote the radius of the maximal palindrome centered at c in w. The sets $\{c - \rho_w(c) + 1, \ldots, c\}$ and $\{c + 1, \ldots, c + \rho_w(c)\}$ of positions are called the *left* and *right arms* of the maximal palindrome centered at c, respectively.

A string z is called a *Z-shape* if $z = xx^Rx$ for a non-empty string $x \in \Sigma^+$. The *tail* of z is the suffix x^Rx. When z occurs as a substring $z = w[i : j]$ of a string w, the positions $p_1 = i + s - 1$ and $p_2 = i + 2s - 1$ are called the *left* and *right pivots* (of the occurrence) of z. The occurrence of the Z-shape is represented by a pair $\langle p_1, p_2 \rangle$. Note that the left and right pivots are the centers of the constituent palindromes xx^R and x^Rx, respectively.

Example 1. For a string $w = \mathtt{ab\underline{abccbaabc}a}$, $\langle 5, 8 \rangle$ is an occurrence of Z-shape $w[3 : 11] = \mathtt{abccbaabc}$.

Minimum graph inference from a walk

Let us define a binary relation $\to$ over nonempty strings by $xyy^Ryz \to xyz$ for $x, z \in \Sigma^*$ and $y \in \Sigma^+$. We call a string w *irreducible* if there is no string w' such that $w \to w'$. Aslam and Rivest [2] proved that every string w admits a unique

irreducible string w' such that $w \to^* w'$, where $\to^*$ is the reflexive and transitive closure of $\to$. Let us call the string w' the *Z-normal form* of w and denote it by $\hat{w}$. Their result can be written as follows.

Theorem 1. ([2]) *The sequence of the labels of the edges of the minimum path graph with output T of an end-to-end walk is its Z-normal form $\hat{T}$.*

Therefore, to infer the minimum path graph from an end-to-end walk is to calculate its Z-normal form.

Example 2. The Z-normal form of T = `cbaaaabccbaabba` is $\hat{T}$ = `cba`, which is obtained by $\texttt{cb}\underline{\texttt{aaa}}\texttt{abccbaabba} \to \underline{\texttt{cbaabccba}}\texttt{abba} \to \texttt{c}\underline{\texttt{baabba}} \to \texttt{cba}$. Here, underlines show Z-shapes to contract. Another way to obtain $\hat{T}$ is $\texttt{cbaaaabcc}\underline{\texttt{baabba}} \to \texttt{cba}\underline{\texttt{aaa}}\texttt{bccba} \to \underline{\texttt{cbaabccba}} \to \texttt{cba}$.

3 Irreducible and Suffix-Reducible Strings

We call a string w *suffix-reducible* if every proper prefix of w is irreducible but w is reducible. Clearly a Z-shape occurs in a suffix-reducible string as a suffix. By deleting its tail, we obtain an irreducible string. A string w is said to be *pseudo-irreducible* if every proper prefix of w is irreducible.

Starting with $w = u_0 = \varepsilon$, our algorithm repeats the following procedure. We extend $w = u_{i-1}$ by reading letters from the input string T one by one until it becomes a suffix-reducible string $w = v_i$. Then we reduce v_i to $u_i = \hat{v}_i$ by deleting the tail of the Z-shape and resume reading letters of T. By repeatedly applying the procedure, we finally obtain the normal form $w = \hat{T}$.

Therefore, strings our algorithm handles are all pseudo-irreducible. We first study mathematical properties of such strings.

Lemma 1. *Every suffix-reducible string has a unique nonempty suffix palindrome and thus has a unique Z-shape.*

There can be several suffix palindromes in an irreducible string. Lemma 1 implies that only one among those can become[1] the tail of the unique Z-shape in a suffix-reducible string (Lemma 1), in which moment the other ones that used to be suffix palindromes are not suffix palindromes any more. This lemma suggests us to keep watching just one (arbitrary) suffix palindrome when reading letters from the input in order to detect a Z-shape. When the palindrome we are watching has become a non-suffix palindrome, we look for another suffix palindrome to track. Suppose we are tracking a suffix palindrome centered at c of radius $r = \rho_w(c) = |w| - c$ in w. When appending a new letter t from the input to w, it

[1] To avoid lengthy expressions, we casually say that a palindrome centered at c in x *becomes* or *grows to* a bigger palindrome in xy when $\rho_x(c) < \rho_{xy}(c)$, without explicitly mentioning several involved mathematical objects that should be understood from the context or that are not important. Other similar phrases should be understood in an appropriate way.

is still a suffix palindrome in wt if and only if $wt[c-r] = wt[c+r+1] = t$. In that case, it is the tail of a Z-shape if and only if $\rho_w(c-r-1) \geq r+1$. Apparently we need to know the maximal radii at all positions to detect a Z-shape but appending a new letter or deleting the tail of a Z-shape disturbs those values even on positions that are not deleted. It takes more than linear time if we keep recalculating the maximal radius at every position. Therefore, we have to partly give up to maintain the exact values of maximal radii. However, there is a moment when maximal radii are stable.

Definition 1. Let w be an irreducible string and c a position in w. We say that c is *stable* in w, if for any string y, either

- there is a prefix x of y for which $|\widehat{wx}| < c$, or
- for any prefix x of y, $\rho_{\widehat{wx}}(c) = \rho_w(c)$.

Moreover, c is *strongly stable* if the former never happens.

That is, if c is stable, the maximum radius at c need not be recalculated when appending letters or deleting a Z-shape's tail at the end of the string, unless the position itself is deleted. In the remainder of this section, we present conditions for a position to be stable.

Let us write $c \sqsubset_w d$ if $c \leq d - \rho_w(d) < d \leq c + \rho_w(c) \leq d + \rho_w(d)$, which roughly means that the right arm of the palindrome centered at c includes the left arm of the one at d. Clearly $c \sqsubset_w d$ implies $\rho_w(c) \geq \rho_w(d)$. Moreover if $c \sqsubset_w d$ and $c = d - \rho_w(d)$ then $\langle c, d \rangle$ is a Z-shape in w. Note that the condition $c \leq d - \rho_w(d)$ in the above definition is redundant for a pseudo-irreducible string; one can see that if $c < d \leq c + \rho_w(c) \leq d + \rho_w(d)$ and $d - \rho_w(d) < c$, then $\langle c, d \rangle$ is a non-suffix Z-shape.

A *palindrome chain from* c_0 *in* w is a sequence $\boldsymbol{c} = (c_0, \ldots, c_k)$ of positions in w such that $c_{i-1} \sqsubset c_i$ for each $i = 1, \ldots, k$. The *frontier of the palindrome chain* $\boldsymbol{c}$ *in* w is the position $F_w(\boldsymbol{c}) = c_k + \rho_w(c_k)$, and the *maximum frontier from a position* c is

$$\mathscr{F}_w(c) = \max\{\, F_w(\boldsymbol{c}) \mid \boldsymbol{c} \text{ is a palindrome chain from } c \,\}\,.$$

The *originator* $\mathscr{A}(d)$ of a position d in w is the smallest position $\mathscr{A}(d) = c$ such that $c \leq d \leq \mathscr{F}_w(c)$. Figure 2 illustrates a palindrome chain in a string $w =$ `xabbcddeeddcbbaabbcddcddcy`.

The stability property can be rephrased in various ways.

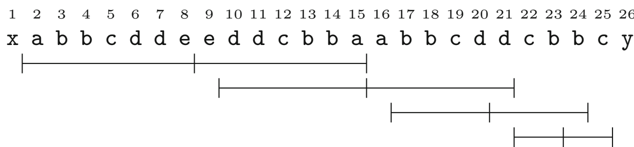


Fig. 2. In a string $w =$ `xabbcddeeddcbbaabbcddcbbcy`, $(8, 15, 20, 23)$ is a palindrome chain, whose frontier is 25. The originator of any position between 8 and 25 is 8

Algorithm 1. Z-detector

```
1  Let Pals be an empty array and w = ε;
2  Function ZDetect(T)
3      T := $T#;                                   // $ and # are sentinel symbols
4      w.append(T);                                // append a new letter from T to w
5      while there remains to read in T do
6          w.append(T);
7          ZDetectInChain(|w| − 1);
8      output "No Z-shape" and halt;
9  Function ZDetectInChain(c)
10     b = |w|;
11     Extend(c);
12     for d := b to c + Pals[c] do                // in increasing order
13         if d + Pals[2c − d] < c + Pals[c] then Pals[d] := Pals[2c − d] ;
14         else ZDetectInChain(d) and break;
15 Function Extend(c)
16     r := |w| − c − 1;
17     while w[c + r + 1] = w[c − r] do
18         r := r + 1;
19         if Pals[c − r] ≥ r then output ⟨c − r, c⟩ and halt;
20         w.append(T);
21     Pals[c] := r;
```

Proposition 1. *The following four are equivalent:*

(1) c is stable in w,
(2) $\mathscr{F}_w(c) < |w|$,
(3) for any string y, either
  - *there is a prefix x of y for which $|\widehat{wx}| < c$, or*
  - *for any prefix x of y, $|\widehat{wx}| > \mathscr{F}_w(c)$,*

(4) $c \sqsubset_w d$ implies that d is stable in w for all d.

Here is another rephrasing.

Corollary 1. *Suppose that positions $c+1, \ldots, |w|-1$ are all stable in an irreducible string w. Then a position c is stable if and only if $c + \rho_w(c) < |w|$.*

It is not hard to see that a position c is strongly stable if and only if all the positions $d \le c$ are stable. If a position c is strongly stable in w, then $|\widehat{wx}| > c$ for any x.

4 Algorithm

Our algorithm is based on Manacher's [3] for calculating the radius of the maximal palindrome at every position in an input. Algorithm 1 detects the first

occurrence of a Z-shape in the input string T. Commenting out Line 19 gives his original algorithm with slightly different appearance. The algorithm reads letters from the input one by one, while focusing on the left-most suffix palindrome among possibly many others. The algorithm computes the maximum radius at each position from left to right and stores those values in the array *Pals*. The function $\mathsf{Extend}(c)$ calculates $Pals[c]$ naively comparing letters on the left and right in the same distance from c, knowing that the radius is at least $|w| - c - 1$. Due to the symmetry, the maximum radii at positions in the right arm of a big palindrome coincide those at the corresponding positions in the left arm, unless those palindromes in the right arm may go beyond the right end of the big palindrome. The function $\mathsf{ZDetectInChain}(c)$ copies the value of $Pals[c-r]$ to $Pals[c+r]$ ($d = c+r$ in the algorithm) for $r \leq Pals[c]$ as long as $c+r+Pals[c-r] < c+Pals[c]$. If $c+r+Pals[c-r] \geq c+Pals[c]$, which means $c \sqsubset c+r$, $\mathsf{ZDetectInChain}(c)$ recursively calls $\mathsf{ZDetectInChain}(c+r)$. If there is no such r, it means that we have reached the frontier of the originator of c. By the correctness of his algorithm and Lemma 1, we see that Algorithm 1 outputs the Z-shape occurrence of the shortest suffix-reducible prefix of the input. If the input has no Z-shape, it halts with the array *Pals* such that $Pals[c] = \rho_w(c)$ for all the positions c. One may think of using this algorithm to compute the normal form by deleting the tail of the found Z-shape. However, deleting a Z-shape tail alters the maximal radii, which have been calculated before, and maintaining those values is not a trivial issue. As we have discussed earlier, to keep recalculating the exact values of the maximal radii takes more than linear time.

4.1 Outline of Our Algorithm

Our online algorithm for calculating the Z-normal form of an input string T is shown as Algorithm 2. Throughout the algorithm, the string w in the working space is kept pseudo-irreducible. That is, $w^{\lhd} = w[1 : |w|-1]$ is irreducible and we would like to know if w itself is still irreducible. Algorithm 2 consists of functions $\mathsf{Stabilize}$, $\mathsf{SlowExtend}$ and $\mathsf{FastExtend}$ in addition to the main function $\mathsf{ZReduce}$. Among those, $\mathsf{Stabilize}$ plays the central role. The data structures we use are very simple: a working string w, an array *Pals* for the maximal radius at each position of w, and a stack of positions. Those are all global variables in Algorithm 2. At the beginning, we add extra fresh symbols $\mathtt{\$}$ and $\mathtt{\#}$ to the left and right ends of the input, respectively. Those work as sentinel symbols so that we never try to access the working string beyond the ends when extending a suffix palindrome.

The working string is initialized to be the empty string and is expanded by appending letters from T one by one by append. Suppose that we have read u from the input and $w = \hat{u}$ is in the working space. When the function $\mathsf{Stabilize}(c)$ is called, we know that $c + \rho_{w^{\lhd}}(c) = |w^{\lhd}|$ but not yet sure if $c + \rho_w(c) = |w|$ holds. Then $\mathsf{Stabilize}(c)$ processes the shortest prefix v of the unprocessed suffix of T such that c is stable in the resultant string $w' = \widehat{wv}$, i.e., $\mathscr{F}_{w'}(c) = |w'| - 1$. In an extreme case, we have $w' = w$ and just confirm $\mathscr{F}_w(c) = |w| - 1$. After the execution of $\mathsf{Stabilize}(c)$, unless it returns true, it is guaranteed that all positions $d \in [c : \mathscr{F}_{w'}(c)]$ are stable in w' and satisfy $Pals[d] = \rho_{w'}(d)$. This is why we

Algorithm 2. Z-reducer

```
1  Let Stack be an empty stack, Pals an empty array and w = ε;
2  Function ZReduce(T)
3      T := $T#;                                          // $ and # are sentinel symbols
4      w.append(T);
5      while there remains to read in T do
6          w.append(T); Stack.clear();
7          Stabilize(|w| − 1);
8      return w[2 : |w| − 1];                             // strip the sentinel symbols
9  Function Stabilize(c)
10     b = |w|; unstable := true;
11     while unstable do
12         unstable := false;
13         if SlowExtend(c) then return true;
14         for d := c + Pals[c] downto b do              // in decreasing order
15             if d + Pals[d] ≥ c + Pals[c] then
16                 if FastExtend(d) then
17                     if Stabilize(d) then
18                         if c = |w| then return true;
19                         if d = |w| then Pals[d] := Pals[2c − d];
20                         w.append(T); unstable := true;
21                         break;
22                 Stack.push(d);
23     return false;
24 Function SlowExtend(c)
25     r := |w| − c − 1;
26     while w[c + r + 1] = w[c − r] do
27         r := r + 1;
28         if Pals[c − r] ≥ r then                        // detect a suffix Z-shape ⟨c − r, c⟩
29             w := w[1 : c − r];                         // contract the suffix Z-shape
30             Pals := Pals[1 : c − r];                   //        same as above
31             return true;
32         Pals[c + r] := Pals[c − r];                    // transfer the value
33         w.append(T);
34     Pals[c] := r;                                      // Pals[c] = ρ_w(c)
35     return false;
36 Function FastExtend(d)
37     while Stack is not empty do
38         r := Stack.top() − d;
39         if Pals[d − r] ≥ Pals[d + r] then Stack.pop();
40         else Pals[d] := r + Pals[d − r]; return false;   // Pals[d] = ρ_w(d)
41     return true;
```

name the function Stabilize. Moreover if the call of Stabilize(c) was from the main function ZReduce(T), $c = \mathscr{A}_{w'}(c)$ and those positions d are all strongly stable.

To stabilize all the positions up to the (future) frontier of c, Stabilize(c) recursively calls Stabilize(d) for positions d such that $c \sqsubset d$. This accords with the definition of the frontier. To determine positions d on which we should recursively call Stabilize(d), we need to know the value of $\rho_w(c)$ first of all. The function Stabilize(c) first calls SlowExtend(c). When the function SlowExtend(c) is called, we are sure $c + \rho_{w^\lhd}(c) \geq |w^\lhd|$. By reading more letters from the input, it does three tasks. One is to calculate the maximal radius at c exactly, taking the unread part of the input into account. One is to detect and contract a Z-shape whose right pivot is c. The last one is to transfer the values of *Pals* on the left arm to the right arm. We extend the palindrome at c by comparing values of $w[c-r]$ and $w[c+r+1]$. When it happens that $Pals[c-r] \geq r$, this means that we find a Z-shape occurrence $\langle c-r, c\rangle$. In this case, the suffix palindrome shall be deleted, and the function returns true. When the palindrome has become non-suffix, it returns false. During the extension of the palindrome at c, it copies the value of $Pals[c-r]$ to $Pals[c+r]$. This transfer might appear nonsense, since it might be the case that $\rho_w(c-r) \neq \rho_w(c+r)$. However, this "sloppy calculation" of radii is advantageous over the exactly correctly calculated values. Those copied values are "adaptive" in extensions and deletions of succeeding part of the working string (and thus the maximal radius at c), in the sense that they can always be used to certainly detect a Z-shape occurrence where the concerned position is the left-pivot. The exactly correct values are too rigid to have this property. Those values will be fixed in the recursive calls of Stabilize.

On Line 14 of Algorithm 2, we recursively call Stabilize(d) in decreasing order for positions on the right arm of the palindrome at c. This "reversed" order might appear unnatural, but this is also related to the adaptability of values in *Pals*. To stabilize positions, anyway we have to calculate the maximal radius at some positions, though they are not yet stable. If we calculate $\rho_w(d)$ in increasing order, they are not adaptive any more. In this case, once some suffix of the working string is deleted and then extended, those exact values would become useless. Contrarily, we calculate $\rho_w(d)$ in the opposite order. Then the previously copied values of *Pals* on the left are adaptive and remain useful, unless they are deleted.

Palindromes overlap a lot even in an irreducible string and we must avoid scanning the same position multiple times. The function FastExtend(d) tells us whether the palindrome at d is a suffix of $w^\lhd$ without looking at letters in the working space. The computation is quickly done by using a stack which stores positions $c_0, \ldots, c_k$ forming a palindrome chain with d such that $F_{w^\lhd}(d, c_0, \ldots, c_k) = |w^\lhd|$. Moreover, it is guaranteed that those positions c_i are stable and $Pals[c_i] = \rho_w(c_i)$. If the right arm of the palindrome centered at d can reach $|w^\lhd|$, the left arm of it must have the structure that can be seen as the "reversed" palindrome chain symmetric to the one in *Stack*. By examining whether $Pals[2d - c_i] = Pals[c_i]$ for each i, one can tell whether the right arm of the maximal palindrome at d can reach the position $|w| - 1$. If it is the

case, $\mathsf{FastExtend}(d)$ returns true and lets SlowExtend extend the palindrome by investigating further. Otherwise, $\mathsf{FastExtend}(d)$ lets $Pals[d] = \rho_w(d)$ and returns false.

Example 3. We show a running example of Algorithm 2. Consider an input string T = abbaa1221aabbaa11aabbb. Assume that $\mathsf{ZReduce}(T)$ has read w = \$abbaa12 and computed $Pals[1:7]$, where the red part has been stabilized. Then, $\mathsf{Stabilize}(8)$ extends the palindrome at 8 by $\mathsf{SlowExtend}(8)$ up to w = \$abbaa1221aabbaa. Next, $\mathsf{Stabilize}(15)$, called by $\mathsf{Stabilize}(8)$, finds a maximal palindrome aa in w = \$abbaa1221aabbaa1. Then 15 is pushed to the stack. After that, $\mathsf{FastExtend}(13)$, called via $\mathsf{Stabilize}(13)$ by $\mathsf{Stabilize}(8)$, reveals that $\rho_w(13) \geq 3$ in w = \$abbaa1221aabbaa1 using $Stack = (15)$. Then $\mathsf{SlowExtend}(13)$ extends the radius of the palindrome at 13 by one as w = \$abbaa1221aabbaa11. Then $\mathsf{Stabilize}(13)$ calls $\mathsf{Stabilize}(17)$. By reading further letters from T, the working string becomes w = \$abbaa1221aabbaa11aab, where a Z-shape occurrence $\langle 13, 17\rangle$ = 1aabbaa11aab is found. By deleting the tail of it, we have w = \$abbaa1221aab, on which $\mathsf{Stabilize}(8)$ resumes calculation. Now the palindrome is extended as w = \$abbaa1221aabbb. Then $\mathsf{Stabilize}(18)$ calls $\mathsf{Stabilize}(14)$, which detects and contracts $\langle 13, 14\rangle$ = bbb. We now have w = \$abbaa1221aa. After that, $\mathsf{Stabilize}(8)$ continues extending the palindrome and obtains w = \$abbaa1221aab#. Finally, ZReduce halts with $\hat{T} = w[2 : |w| - 1]$ = abbaa1221aab.

4.2 Correctness and Complexity of the Algorithm

To prove the correctness of our algorithm, we first introduce some technical definitions, which characterize "adaptive" values.

Definition 2. Let us write $i \sim_k j$ if $\min\{i, k\} = \min\{j, k\}$. We say that $Pals$ on w is *accurate enough between* c *and* d if for any $e \in [c : d]$, it holds that $Pals[e] \sim_{d-e} \rho_w(e)$. We denote this property by $Æ_w(c, d)$ with implicit understanding of $Pals$.

Let $\nu_w(c)$ denote the largest e such that $e \sqsubset c$. If there is no such e, let $\nu_w(c) = 1$. We say that c is *left-good* in w if $Æ_w(\nu_w(c), c)$ holds. We say that c is *right-good* in w if $Æ_w(c, c + \rho_w(c))$ holds.

Clearly $Æ_w(c_1, d_1)$ implies $Æ_w(c_2, d_2)$ if $[c_2 : d_2] \subseteq [c_1 : d_1]$.

Lemma 2. *Suppose that c is left-good and $\rho_w(c) = |w| - c$ for a pseudo-irreducible string w. Then w has a Z-shape occurrence $\langle c - \rho_w(c), c\rangle$ if and only if $Pals[c - \rho_w(c)] \geq \rho_w(c)$. Suppose in addition $Pals[c - r] = Pals[c + r]$ for all $r = 1, \ldots, \rho_w(c)$. Then, c is right-good.*

If $Æ_w(d, c)$ holds, then one can correctly determine whether w has a Z-shape with right pivot c. Namely, $\langle d, c\rangle$ is a Z-shape if and only if $Pals[d] \geq c - d$. We detect a suffix Z-shape whose right pivot is c extending a suffix palindrome at c in $\mathsf{SlowExtend}(c)$. Lemma 2 means that this indeed works well when c is left-good

and values on the left arm are copied to the corresponding positions on the right arm. Note that the left-goodness depends on $w[1:c]$ only. This means that this property is robust against deletion and extension of the right arm.

We will show that the function Stabilize satisfies the following precondition and postcondition, where w and w' are the working strings before and after a call, respectively.

Condition 1 (Precondition of Stabilize(c)).

- *Stack* is empty,
- $c + \rho_w(c) \geq |w| - 1$,
- c is left-good,
- For all positions $d \in [1 : \mathscr{A}(c) - 1] \cup [c + 1 : |w| - 1]$, d is stable in w and $Pals[d] = \rho_w(d)$.

Condition 2 (Postcondition of Stabilize(c)).

- If it returns true, then
  - $w' = \widehat{wu}$ for the shortest string u appended from the input such that $|w'| \leq c$,
  - *Stack* is empty.
- If it returns false, then $w' = \widehat{wu}$ for the shortest string u appended from the input such that
  - $(c; Stack)$ is a palindrome chain such that $\mathscr{F}_{w'}(c) = F_{w'}(c; Stack) = |w'| - 1$,
  - d is stable in w' and $Pals[d] = \rho_w(d)$ for all $d \in [c : |w'| - 1]$.

Lemma 3 (Stabilize)**.** *Suppose that c satisfies Condition 1. Then after executing* Stabilize(c)*, Condition 2 is satisfied.*

Assuming that Lemma 3 is true, we establish the following proposition.

Proposition 2. *Algorithm 2 calculates the normal form of the input.*

When Stablize(c) tries to fix the value $Pals[c]$ to be $\rho_w(c)$, the right arm of the palindrome at c may be cut in the middle after finding the end of the right arm in a string, unless it has been stabilized. Then we need to extend it again. The **while** loop is repeated until c becomes stable.

Condition 3 (Precondition of the while loop). In addition to Condition 1,

- for all positions $d \in [b : |w| - 1]$, $Pals[d] = Pals[2c - d]$.

In what follows we give some lemmas that explain the behavior of our algorithm in a more formal way.

Lemma 4 (SlowExtend)**.** *Suppose that at the beginning of an iteration of the **while** loop of* Stabilize(c)*, Condition 3 holds. Let w and w' be the working strings before and after execution of* SlowExtend(c)*, respectively. Then either*

– $\mathsf{SlowExtend}(c)$ *returns* true,
– $w' = \widehat{wu}$ *for* u *appended from the input such that* wu *is suffix-reducible and the right pivot of the Z-shape is* c,

or

– $\mathsf{SlowExtend}(c)$ *returns* false,
– $w' = wu$ *for* u *appended from the input such that* $c + \rho_{w'}(c) = |w'| - 1$ *and* w' *is pseudo-irreducible,*
– $\mathit{Pals}[c] = \rho_{w'}(c)$,
– *for all* $r \in [1 : \mathit{Pals}[c]]$, $\mathit{Pals}[c + r] = \mathit{Pals}[c - r]$.

Lemma 5 ($\mathsf{FastExtend}$)**.** *Suppose that* $\mathsf{FastExtend}(d)$ *is called from* $\mathsf{Stabilize}(c)$ *satisfying that*

– $(c; \mathit{Stack})$ *is a palindrome chain from some* $e > d$ *such that* $F_w(c; \mathit{Stack}) = \max\{ \mathscr{F}_w(e) \mid d < e \leq c + \mathit{Pals}[c] \} = |w| - 1$,
– c *is left-good and right-good,*
– *for all* $e \in [d + 1 : |w| - 1]$, e *is stable and* $\mathit{Pals}[e] = \rho_w(e)$,

Then after the execution,

– *if it returns* true, *then* $d + \rho_w(d) \geq |w| - 1$ *and Stack is empty,*
– *if it returns* false, *then*
  - $(d; \mathit{Stack})$ *is a palindrome chain such that* $F_w(d; \mathit{Stack}) = \max\{ \mathscr{F}_w(e) \mid d \leq e \leq c + \mathit{Pals}[c] \} = |w| - 1$,
  - *for all* $e \in [d : |w| - 1]$, e *is stable and* $\mathit{Pals}[e] = \rho_w(e)$.

Lemma 6. *Suppose that* c *is right-good and* $c \sqsubset_w d$ *in a pseudo-irreducible string* w*. Then* d *is left-good in* w.

Hence, when $\mathsf{FastExtend}(d)$ returns true, Condition 1 for d is satisfied.

Now we have prepared enough for analyzing the function $\mathsf{Stabilize}(c)$. Our goals is to show that Condition 2 holds for $\mathsf{Stabilize}(c)$ provided that Condition 1 holds. The function $\mathsf{Stabilize}(c)$ calls $\mathsf{Stabilize}(d)$ recursively. For now we assume that Condition 1 implies Condition 2 for those d. Then this inductive argument completes a proof of Lemma 3.

Suppose that Condition 1 holds for $\mathsf{Stabilize}(c)$. If $\mathsf{SlowExtend}(c)$ returns true, clearly Condition 2 holds by Lemma 4. Hereafter we suppose that $\mathsf{SlowExtend}(c)$ returns false.

Lemma 7 (for loop). *Suppose that Condition 3 is satisfied at the beginning of every iteration of the* ***while*** *loop. Then, at the beginning of each iteration of the* ***for*** *loop of* $\mathsf{Stabilize}(c)$, *the following holds.*

– $(c; \mathit{Stack})$ *is a palindrome chain such that*

$$F_w(c; \mathit{Stack}) = \max(\{ \mathscr{F}_w(e) \mid d < e \leq c + \mathit{Pals}[c] \} \cup \{c + \mathit{Pals}[c]\}) = |w| - 1,$$

– c *is left-good,*

– *for all* $e \in [c+1:d]$, $Pals[d] = Pals[2c-d]$,
– *for all* $e \in [d+1:|w|-1]$, e *is stable and* $Pals[e] = \rho_w(e)$.

Moreover if we ***break*** *the loop, still Condition 3 holds. If we return* true *on Line 18, Condition 2 holds for* c.

Lemma 8 (while loop). *At the beginning of an iteration of the* ***while*** *loop in* Stabilize(c)*, Condition 3 holds. Moreover if it returns* true*, Condition 2 holds.*

Theorem 2. *Algorithm 2 calculates the normal form of the input in linear time.*

Proof. The function Stabilize is called from ZReduce or Stabilize itself. In both cases, Stabilize(c) is called right after a new letter is appended at position $c+1$. More precisely, in the latter case, Stabilize(d) is called just after SlowExtend(c) or Stabilize(c) appended a new letter at position $d+1$. Note that when **while** loop repeats, the letters on the positions $d+1$ for which Stabilize(d) was called are deleted. Therefore, the number of calls of Stabilize is bounded by $|T|$.

This explanation about the number of calls of Stabilize also shows that the total number of the execution of the **while** loop is bounded by $|T|$ and this implies the number of calls of SlowExtend is also bounded by $|T|$. The total running time of SlowExtend is bounded by the number of its calls and the times of appending letters from T, which is bounded by $O(|T|)$ in total. The same argument on the number of calls of Stabilize applies to that of executions of the **for** loop. This implies that the total number of positions that is pushed onto the stack is bounded by $|T|$, which implies that total running time of FastExtend is bounded by $O(|T|)$.

All in all, Algorithm 2 runs in linear time. ☐

By using Algorithm 2 and Raghavan's algorithm [6], the smallest path and cycle can be inferred from walks in linear time.

Corollary 2. *Given a string* w *of length* n*, the smallest path and cycle on which* w *is the output of a walk can be inferred in* $O(n)$ *time.*

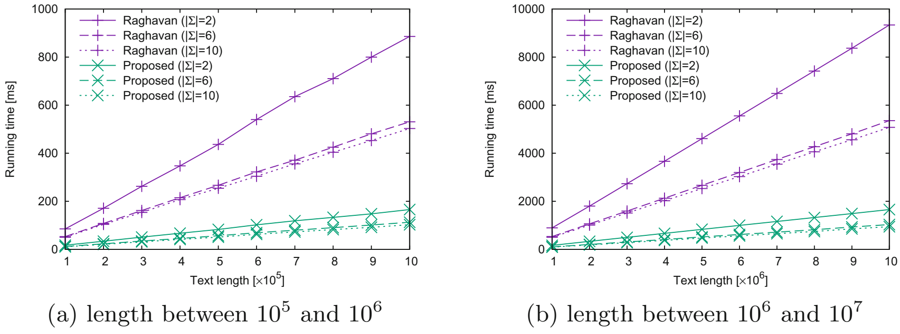
5 Experiments

This section presents experimental performance of our algorithm comparing with Raghavan's $O(n \log n)$ time algorithm [6].

We implemented these algorithms in C++ and compiled with Visual C++ 12.0 (2013) compiler. The experiments were conducted on Windows 7 PC with Xeon W3565 and 12GB RAM. In the whole experiments, we got the average running time for 10 times of attempts.

First, for randomly generated strings of length between 10^5 and 10^6 over Σ of size $|\Sigma| = 2, 6, 10$, we compared the running time of the algorithms (Fig. 3 (a)). For any alphabet size, our proposed algorithm ran faster.

Furthermore, we conducted experiments for strings of length between 10^6 and 10^7 with the same alphabets, and got a similar result (Fig. 3 (b)). Here,



(a) length between 10^5 and 10^6 (b) length between 10^6 and 10^7

Fig. 3. Running time for the random strings with $|\Sigma| = 2, 6, 10$

the slope of Raghavan's algorithm's performance increases slightly as the string length increases. On the other hand, our proposed algorithm keeps the same slope. This shows the proposed algorithm runs in linear time in practice.

Acknowledgments. The research is supported by JSPS KAKENHI Grant Numbers JP15H05706, JP26330013 and JP18K11150, and ImPACT Program of Council for Science, Technology and Innovation (Cabinet Office, Government of Japan).

References

1. Akutsu, T., Fukagawa, D.: Inferring a graph from path frequency. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 371–382. Springer, Heidelberg (2005). https://doi.org/10.1007/11496656_32
2. Aslam, J.A., Rivest, R.L.: Inferring graphs from walks. In: Computational Learning Theory, pp. 359–370 (1990)
3. Manacher, G.K.: A new linear-time on-line algorithm for finding the smallest initial palindrome of a string. J. ACM **22**(3), 346–351 (1975)
4. Maruyama, O., Miyano, S.: Graph inference from a walk for trees of bounded degree 3 is NP-complete. In: Wiedermann, J., Hájek, P. (eds.) MFCS 1995. LNCS, vol. 969, pp. 257–266. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60246-1_132
5. Maruyama, O., Miyano, S.: Inferring a tree from walks. Theor. Comput. Sci. **161**(1), 289–300 (1996)
6. Raghavan, V.: Bounded degree graph inference from walks. J. Comput. Syst. Sci. **49**(1), 108–132 (1994)