Palindrome Recognition Using a Multidimensional Tape

Therese Biedl[*], Jonathan F. Buss[*], Erik D. Demaine[†], Martin L. Demaine[†], Mohammadtaghi Hajiaghayi[†], Tomáš Vinař[*]
School of Computer Science
University of Waterloo

January 17, 2003

Abstract: The problem of palindrome recognition using a Turing machine with one multidimensional tape is proved to require $\Theta(n^2/\log n)$ time.

Introduction

A *palindrome* is a word that reads the same forward and backward. Hennie [1] showed that to test whether an input word is a palindrome requires $\Theta(n^2)$ time for a palindrome of length n on a standard one-tape Turing machine. A multitape Turing machine can test for palindromes in real time [2].

We consider the case of one two-dimensional tape. We extend Hennie's crossing-sequence argument to this case and prove that time $\Omega(n^2/\log n)$ is necessary. We also present an algorithm achieving $O(n^2/\log n)$ time. Both bounds assume that the input is presented linearly along the first row of the tape.

[*]School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1. Supported in part by grants from the Natural Sciences and Engineering Research Council (NSERC) of Canada.

[†]Work performed while at the School of Computer Science, University of Waterloo and supported in part by grants from NSERC. Current address: Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA.

The Lower Bound

The lower bound, like the bound for one-dimensional tapes, uses the concept of a *crossing sequence*. Assume that a Turing machine M to accept palindromes is fixed, and consider the movement of the tape head when the input is a palindrome w. To extend crossing sequences to two dimensions, we consider crossing a column boundary, and include in the specification of each crossing the row at which the head crossed the boundary.

Let w be an input word and let $i \geq 1$. The ith crossing sequence on word w, which we will denote $C_i(w)$, is a sequence $\{(q_1, r_1), (q_2, r_2), \ldots, (q_k, r_k)\}$ of states and row numbers such that

- At some time t_1, the tape head moves from cell (i, r_1) to cell $(i+1, r_1)$, and the next state is q_1.
- At some time $t_2 > t_1$, the tape head moves from cell $(i+1, r_2)$ to cell (i, r_2), and the next state is q_2.
- For all odd ℓ, $3 \leq \ell \leq k$, at some time $t_\ell > t_{\ell-1}$, the tape head moves from cell (i, r_i) to cell $(i+1, r_i)$, and the next state is q_ℓ.
- For all even ℓ, $3 \leq \ell \leq k$, at some time $t_\ell > t_{\ell-1}$, the tape head moves from cell $(i+1, r_i)$ to cell (i, r_i), and the next state is q_ℓ.
- Only at times $t_1, t_2, \ldots, t_k$ does the tape head move from column i to column $i+1$ or vice versa.

The state and row are the only information that the machine carries from one column to the next. This limitation leads to the following "splicing lemma" exactly as in the one-dimensional case [1].

Lemma 1 (Hennie) *Suppose that M accepts both xy and uv, with $|x| = i$ and $|u| = j$, and that $C_i(xy) = C_j(uv)$. Then M accepts both xv and uy.*

In the case where M accepts the language of palindromes, the lemma implies that two different palindromes must have different crossing sequences in most cases. (The exceptions arise when splicing two strings creates a new palindrome.) To obtain a lower bound, we concentrate on a subclass of palindromes and only some of the crossing sequences. For a word $x \in \{0,1\}^m$, define $w(x) = x0^m \operatorname{rev}(x)$. Let

$$L_m = \{w(x) : x \in \{0,1\}^*,\ |x| = m\}.$$

Words in L_m have the property that if we split and recombine any two such words at the middle part consisting entirely of 0s, then the resulting word is not a palindrome.

Lemma 2 *For any two distinct words $w_1, w_2 \in L_m$ and any $i, j \in \{m, \ldots, 2m\}$, the ith crossing sequence of w_1 and the jth crossing sequence w_2 must be different.*

Proof: Assume to the contrary that there exist $i, j \in \{m, \ldots, 2m\}$ for which the ith crossing sequence of $w_1 = x_1 0^m \operatorname{rev}(x_1)$ and the jth crossing sequence of $w_2 = x_2 0^m \operatorname{rev}(x_2)$ are the same, but $w_1 \neq w_2$. By the previous lemma, M accepts the word $x_1 0^{m+i-j} \operatorname{rev}(x_2)$, which is not a palindrome. $\Box$

The number of possible crossing sequences of length less than l in a computation using at most n^2 time is less than $(sn^2)^l$, where s is the number of states of M. Since L_m has 2^m members with mutually disjoint sets of crossing sequences, we must have $(sn^2)^l \geq 2^m$, which yields $l \geq \log_{(sn^2)} 2^m = m/(2\log n + \log s)$. Therefore some word $w \in L_m$ has m crossing sequences of length $\Omega(m/\log n)$. The time used by M is at least the sum of the lengths of its crossing sequences, which is $\Omega(n^2/\log n)$.

We have proved the following.

Theorem 1 *A one-tape two-dimensional Turing machine that accepts the language of palindromes requires $\Omega(n^2/\log n)$ steps to accept some palindrome of length n.*

The proof extends immediately to k-dimensional tapes using crossing sequences across a $(k-1)$-dimensional hyperplane.

The Upper Bound

Now we show that the lower bound is tight, by giving an algorithm for a Turing machine that accepts palindromes in $O(n^2/\log n)$ time.

The outline of the algorithm is as follows. Let the input alphabet be $\{0, 1, \ldots, a-1\}$. Break the input string into blocks of some length y. Interpret each block as an a-ary number N with value between 0 and $a^y - 1$. Now move down to row N, thus using the row to encode the value of this block. Similarly, we study the matching block, interpret its reverse as the binary encoding of a number and go to the corresponding row. By comparing

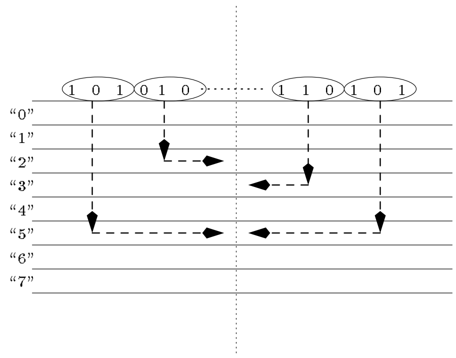

Figure 1: Matching by rows. In this example, the outermost blocks match, but the next two do not.

whether we marked the same row both times, we can discover whether the two blocks were the reverse of each other.

In this way, by crossing from one end of the string to the other just twice, we can compare two blocks of length y. Hence, only n/y passes will be needed to compare the whole string. By choosing y suitably, we obtain the desired running time.

The precise algorithm is as follows. We assume a left endmarker; the blank at the end of the input serves as a right endmarker.

1. Initialization:

   (a) Assume that a string of length n is initially in the first row of the tape.

   (b) Compute $\log n$, and write it in unary, using 0s, into the second row. (To compute $\log n$, make repeated scans of the input, marking every second unmarked symbol in the input, until all symbols are marked. The number of scans is $\lfloor \log n \rfloor + 1$.)

   (c) Compute $\log \log n$ from $\log n$, and write it in unary into the third row.

   (d) Subtract the third row from the second row, so that the second row now contains $\log n - \log \log n$ in unary (using 0s). Erase the third row.

   As we will see, $\log n - \log \log n$ is the value that we will use for the length y of the blocks. Hence we have now computed y.

2. Repeatedly compare blocks as follows:

   (a) Deal with the leftmost block:

      i. Fill the space underneath the leftmost block:

         A. Start in the second row (which contains 0^y).

         B. Repeatedly copy the contents of the current row to the next row, adding 1 (as an a-ary number) each time.

         C. Stop when all as are written.

         The space underneath the leftmost block now contains all possible strings with y characters, sorted by their numerical value.

      ii. Mark the space underneath the leftmost block as matching or non-matching:

         A. Go to the first column of the leftmost block.

         B. Memorize the character c in the input in a state.

         C. Go down that column (as long as it is filled). For every entry that matches c, replace the entry by $\uparrow$. For every entry that doesn't match c, replace the entry by $\downarrow$.

         D. Repeat this for all other columns of the block. (The block ends when there is a blank in the second row.)

      iii. Mark the appropriate row:

         A. Scan all rows underneath the block down to the first blank row.

         B. If a row contains a $\downarrow$ somewhere, replace all entries in the row by #.

         C. Only one row will not contain a $\downarrow$ (namely, the row that exactly matched the content of the block initially).

   (b) Deal with the rightmost block:

      i. Copy the unary encoding of y from the beginning of the second row to the end of the second row (located by searching for the blank in the first row).

      ii. Fill the space underneath the rightmost block as before, except write the strings in reverse (least significant bit at the left).

$$
\begin{array}{ccccccccccc}
0 & 1 & 0 & & 0 & 1 & 0 & & 0 & 1 & 0 \\
0 & 0 & 0 & & \uparrow & \downarrow & \uparrow & & \# & \# & \# \\
0 & 0 & 1 & & \uparrow & \downarrow & \downarrow & & \# & \# & \# \\
0 & 1 & 0 & & \uparrow & \uparrow & \uparrow & & \uparrow & \uparrow & \uparrow \\
0 & 1 & 1 & \Longrightarrow & \uparrow & \uparrow & \downarrow & \Longrightarrow & \# & \# & \# \\
1 & 0 & 0 & & \downarrow & \downarrow & \uparrow & & \# & \# & \# \\
1 & 0 & 1 & & \downarrow & \downarrow & \downarrow & & \# & \# & \# \\
1 & 1 & 0 & & \downarrow & \uparrow & \uparrow & & \# & \# & \# \\
1 & 1 & 1 & & \downarrow & \uparrow & \downarrow & & \# & \# & \#
\end{array}
$$

Figure 2: Filling the space beneath a block.

iii. Mark the space underneath the rightmost block as matching or non-matching as before.

iv. Mark the correct row underneath the rightmost block as before.

(c) Go to the correct row underneath the rightmost block and scan left. If the first non-blank seen is not $\uparrow$, then there was a mismatch and the word is not a palindrome, so crash.

(d) Cleanup:

i. Copy the unary encoding of y to underneath the second block.

ii. Overwrite the marked rows with # as well.

iii. Overwrite the checked blocks of the input with #.

(e) Repeat the matching procedure until the leftmost and the rightmost block overlap. When this happens, use a brute-force approach to test whether the remaining word (which has length less than $2y \in O(\log n)$) is a palindrome.

Analysis

Now we analyze the time complexity. The initialization (computing y) uses $O(n \log n)$ time. The final round (testing the last $2y$ characters to be a palindrome) takes $O(y^2) \subseteq O(\log^2 n)$ time. Thus the dominant factor of the computation time is the product of the number of rounds and the time it takes to process any one block.

In each round, the machine checks $2y$ input symbols, hence the total number of rounds is $O(n/y)$. During each round, filling the space underneath the block involves an $y \times z$ rectangle, for $z = a^y$. Each cell in the rectangle is only visited a constant number of times; hence filling the rectangle takes $O(yz)$ time. Finally, to test whether the two marked rows are the same takes $O(n)$ time. Thus each round takes $O(yz + n)$ time.

The time for all rounds is therefore proportional to $nz + n^2/y = nz + n^2/\log z$. Taking $z = n/\log n$ and thus $y = \log n - \log\log n$ gives a time bound of $O(n^2/\log n)$.

The above yields

Theorem 2 *A one-tape two-dimensional Turing machine can test whether a word of length n is a palindrome in time $O(n^2/\log n)$.*

References

[1] F. C. Hennie, One-Tape Off-Line Turing Machine Complexity, *Information and Control* 8 (1965) 553–578.

[2] Z. Galil, Palindrome Recognition in Real Time by a Multitape Turing Machine, *J. Computer and System Sciences* 16 (1978) 140–157.