Finding palindromes: variants and algorithms

Johan Jeuring

Technical Report UU-CS-2013-016

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Finding palindromes: variants and algorithms

Johan Jeuring[1,2]

[1] Department of Information and Computing Sciences, Universiteit Utrecht
[2] School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands
J.T.Jeuring@uu.nl

Abstract. The problem of finding palindromes in strings appears in many variants: find exact palindromes, ignore punctuation in palindromes, require space around palindromes, etc. This paper introduces several predicates that represent variants of the problem of finding palindromes in strings. It also introduces properties for palindrome predicates, and shows which predicates satisfy which properties. The paper connects the properties for palindrome predicates to two algorithms for finding palindromes in strings, and shows how we can extend some of the predicates to satisfy the properties that allow us to use an algorithm for finding palindromes.

1 Introduction

Rinus Plasmeijer was born on 26-10-52, which makes him 60 on the date I start writing this paper, an excellent occasion to celebrate a productive career in functional programming!

If I turn Rinus' date of birth around, I get 25-01-62, which is close to its original, but not exactly equal. This birthdate is an example of an *approximate palindrome*, a sequence of symbols which is a *palindrome* if you are allowed a minor number of edit operations on the reverse of the original.

Palindromes have long been considered interesting curiosities used in wordplays. We now know that palindromes play an important role in DNA. If I search for the keyword palindrome in the electronic publications available at the library of Utrecht University, I get more than 500 hits. The first ten of these hits are all about palindromes in DNA. My guess is that at least 90% of these 500 publications are about palindromes in DNA. DNA stores information in palindromes amongst others to repair genes. For example, the male DNA contains huge approximate palindromes with gaps in the middle [5]. Some of these palindromes are more than a million base-pairs long. Essential genes, such as the genes for male testes, are encoded on these palindromes.

We need software to find palindromes in large pieces of text, or approximate palindromes with gaps in DNA. Algorithms for determining whether or not a string is a palindrome, and finding palindromes in strings have a long history in computer science, longer than Rinus' career. In an earlier paper [3] I describe the history of finding palindromes. The current paper discusses some of the

variants of the problem of finding palindromes, describes their properties, and gives two algorithms for finding palindromes. The main contributions of this paper are the description of the variants of palindrome finding, their properties, and the relation between these properties and algorithms for finding palindromes. The algorithms themselves are not new. The corresponding software has been implemented in Haskell, and can be found on hackage[1].

2 What is a palindrome?

Palindromes. How can I determine whether or not a string (a list of characters) is a palindrome? The simplest method is to reverse the string and to compare it with itself. So the string xs is a palindrome ($palindrome\ xs$), if xs is equal to its reverse: $xs == reverse\ xs$, where $xs == ys$ is $True$ only when the strings xs and ys are exactly equal. In Haskell I write:

$$\begin{aligned} &palindrome &&:: String \to Bool \\ &palindrome\ xs &&= xs == reverse\ xs \end{aligned}$$

where $reverse$ is defined in the $Prelude$. Without the type declaration, this definition would also work on lists $[a]$ instead of strings, provided we have an equality operator on the type a.

The $palindrome$ predicate satisfies several properties. First, the empty list is a palindrome:

$$palindrome\ [\,] \qquad \text{(EMPTY)}$$

A singleton list is a palindrome, since under standard character equality, $c == c$ for all characters c.

$$\forall\ c.palindrome\ [c] \qquad \text{(SINGLE)}$$

This property doesn't hold for all kinds of palindromes, since in some cases the comparison operator used is not a real equality, and is for example not reflexive. A third property allows me to extend a palindrome at the front with a string and at the back with the reverse of this string to obtain a palindrome. This property is an equivalence: if I remove a string from the front of a palindrome, and remove its reverse from the back, I also obtain a palindrome.

$$\forall\ xs\ ys.palindrome\ ys \Leftrightarrow palindrome\ (xs +\!\!+\ ys +\!\!+\ reverse\ xs) \qquad \text{(EXTEND)}$$

A consequence of this property is that once a string is not a palindrome, I cannot extend it on both the front and the back to become a palindrome. The final property I introduce is the 'palindromes in palindromes' (PALINPAL) property. This property says that if a large palindrome contains a smaller palindrome that does not appear exactly in the middle, the large palindrome contains a second

[1] `http://hackage.haskell.org/package/palindromes`

copy of the smaller palindrome at the other arm of the large palindrome. Figure 1 gives an example: suppose I have a palindrome p (say "abadaba", with center b), which contains a palindrome q (say "aba", with center a). Then the string q' I get by mirroring q in p with respect to p's center is a palindrome again ("aba", with center a'). This property is essentially a consequence of the symmetry of

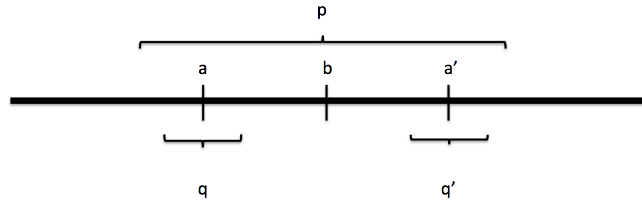

Fig. 1. The palindromes in palindromes property (PALINPAL)

equality: for all a, and b: $a == b \Leftrightarrow b == a$.

Text palindromes. The standard example 'A man, a plan, a canal, Panama!' is not a palindrome according to the *palindrome* definition. Reversing it gives '!amanaP ,lanac a ,nalp a ,nam A', in which it is hard to recognize the original. For this string to also pass the palindrome test, I slightly adapt the definition of what is a palindrome. I call a string a text palindrome if it is equal to its reverse after throwing away all punctuation symbols such as spaces, comma's, periods, etc, and after turning all characters into lower case characters.

$$\begin{aligned}
&\mathit{textPalindrome} :: \mathit{String} \to \mathit{Bool} \\
&\mathit{textPalindrome} = \mathit{palindrome} \circ \mathit{lowerLetter} \\
&\mathit{lowerLetter} :: \mathit{String} \to \mathit{String} \\
&\mathit{lowerLetter} = \mathit{map}\ \mathit{toLower} \circ \mathit{filter}\ \mathit{isLetter}
\end{aligned}$$

where *isLetter* and *toLower* are functions from the module *Data.Char*. The predicate *textPalindrome* satisfies all palindromic properties.

Word palindromes. When looking for palindromes in a text, I often only want palindromes that start and end in complete words. For example, the longest text palindrome in the King James Bible is the string: "no man; even amon", from Isaiah 41:28. The complete verse reads

```
For I beheld, and there was no man; even among them,
and there was no counsellor, that, when I asked of them,
could answer a word.
```

Since Amon is also a biblical name, it is probably slightly confusing to list ”no man; even amon” as the longest palindrome in the Bible. If I only consider palindromes that start and end in words, I get the string ”war draw” in Joel 3:9 as the longest palindrome. A word palindrome is a text palindrome that is preceded and followed by non-letter symbols. To determine whether or not a string is a word palindrome, I also need the context of the input string. The type *CString* describes three tuples of strings, modelling a string (the second component) with its context before (the first) and after (the third).

```
type CString = (String, String, String)
wordPalindrome :: CString → Bool
wordPalindrome input@(before, string, after) =
      textPalindrome string
   ∧ surroundedByPunctuation input
surroundedByPunctuation (before, _, after)   =
      (null before ∨ ¬ (isLetter (last before)))
   ∧ (null after  ∨ ¬ (isLetter (head after)))
```

Since the predicate *wordPalindrome* fundamentally depends on its context, it doesn't satisfy the palindromic properties. Even the PALINPAL property is not satisfied, since the punctuation around a word might differ for two occurrences of a palindrome in a palindrome.

Palindromes in DNA. A sequence of DNA symbols 'A', 'T', 'C' or 'G' is a palindrome if its reverse is the *complement* of the original, where 'T' is the complement of 'A' and vice versa, and similarly for 'C' and 'G'. It follows that we cannot use the == operator anymore in the definition of what it means to be a palindrome in DNA. We define the DNA symbol comparison function =:= by

```
(=:=)         :: Char → Char → Bool
'A' =:= 'T' = True
'T' =:= 'A' = True
'C' =:= 'G' = True
'G' =:= 'C' = True
_   =:= _   = False
```

This operator is symmetric but not reflexive. We use the new equality operator in a definition of *dnaPalindrome* for sequences of DNA symbols. We pairwise combine the elements of an input sequence *xs* and *reverse xs* with the equality operator =:= using the *PreludeList* function *zipWith*, and fold the list we obtain to a single result using the *PreludeList* function *and*.

```
dnaPalindrome      :: String → Bool
dnaPalindrome      = palindromeEq (=:=)
type CharEq        = Char → Char → Bool
palindromeEq       :: CharEq → String → Bool
palindromeEq eq xs = and (zipWith eq xs (reverse xs))
```

Note that the predicate *palindrome* can be defined in terms of *palindromeEq* by $palindrome = palindromeEq\ (==)$. Since a DNA symbol is not its own complement the SINGLE property does not hold, and all palindromes in DNA have even length. *dnaPalindrome* satisfies the EMPTY, EXTEND, and PALINPAL properties.

Approximate palindromes. Sometimes I not only want to find perfect palindromes, but also palindromes that contain a limited number of errors. A palindrome with a limited number of errors is often called an *approximate palindrome.* For example, in the book Judges in the King James Bible, verse 19:9 reads:

```
And when the man rose up to depart, he, and his
concubine, and his servant, his father in law, the
damsel's father, said unto him, Behold, now the day
draweth toward evening, I pray you tarry all night:
behold, the day groweth to an end, lodge here, that
thine heart may be merry; and to morrow get you early
on your way, that thou mayest go home.
```

The substring "draweth toward" is a text palindrome with one error: the 'e' and the 'o' don't match. This is an example of an error that is resolved by substituting one symbol by another symbol. Other errors may be resolved by inserting or deleting a symbol. The substitution, insertion, and deletion operations are the operations used in calculating the Levenshtein distance between two strings.

A string s is an approximate palindrome with k errors, if at most k substitution, deletion, or insertion operations are needed to convert the reverse of s into s. Note that this number of operations will generally be twice the number of operations necessary for turning a string into a palindrome. It follows that "draweth toward" is an approximate palindrome with two errors, substituting 'e' for 'o' and 'o' for 'e'. In the following definition we abstract from the equality operator (==), because we also want to determine approximate palindromes in DNA, for example.

```
approximatePalindrome     :: Int → String → Bool
approximatePalindrome k s = levenshteinDistance (==) s (reverse s) ⩽ k
levenshteinDistance :: CharEq → String → String → Int
levenshteinDistance eq (x : xs) (y : ys) =
        ((if x == y then 0 else 1) + levenshteinDistance eq xs       ys      )
   `min` (1                        + levenshteinDistance eq (x : xs) ys      )
   `min` (1                        + levenshteinDistance eq xs       (y : ys))
levenshteinDistance eq xs     ys       = max (length xs) (length ys)
```

As a program, this predicate is terribly inefficient. The *approximatePalindrome* predicate satisfies the EMPTY, SINGLE, and EXTEND properties. Since it takes an integer argument, the PALINPAL property has to be slightly reformulated. Suppose I have a palindrome p satisfying *approximatePalindrome* k, which contains a palindrome q satisfying *approximatePalindrome* k'. Then the string q' I get by mirroring q in p with respect to p's center satisfies *approximatePalindrome* k'.

Unfortunately, this property doesn't hold for *approximatePalindrome*. The errors in q need not appear in q', and vice versa, so I cannot make a statement about whether or not q' satisfies *approximatePalindrome* k' given that q satisfies *approximatePalindrome* k'.

Gapped palindromes. A palindrome with a gap is a palindrome in which a gap of a particular size in the middle is ignored. An example of a palindrome with a gap is found in Revelations, where verses 20:7-8 read:

```
And when the thousand years are expired, Satan shall
be loosed out of his prison, And shall go out to
deceive the nations which are in the four quarters of
the earth, Gog, and Magog, to gather them together to
battle: the number of whom is as the sand of the sea.
```

Here "Gog, and Magog" is a text palindrome with a gap of length three in the middle: the 'n' and the 'M' around the central 'd' don't match. A gapped palindrome is a special case of an approximate palindrome, where the errors occur in the middle of the palindrome, but one that occurs so often in DNA that it deserves a special category. Since the gap appears in the middle of the string, the length of the gap is odd if the length of the palindrome is odd, and even if the length of the palindrome is even. To be precise, a string s is a palindrome with a gap of length g in the middle, if it satisfies the predicate *gappedPalindrome* g s:

```
gappedPalindrome     :: Int → String → Bool
gappedPalindrome g s = palindrome (rmCenter g s)

rmCenter     :: Int → String → String
rmCenter g s = let ls                 = length s
                   armLength          = div (ls − g) 2
                   (before, rest)     = splitAt armLength s
                   (gap, after)       = splitAt g rest
                   sameParity m n     = even m == even n
               in if   g ⩽ ls ∧ sameParity g ls
                  then before ++ after
                  else error "removeCenter"
```

This predicate specifies perfect palindromes with gaps. If I want to find other kinds of palindromes with gaps, I have to replace *palindrome* with the required predicate. Provided g is at most the length of the input list, and the parity of the input list is the same as the parity of g, *gappedPalindrome* g satisfies the EMPTY, SINGLE, and EXTEND properties. Since gapped palindromes only have a gap at their center, I need to adapt the formulation of the PALINPAL property to apply it to gapped palindromes. Suppose I have a palindrome p satisfying *gappedPalindrome* g, which contains a palindrome q satisfying *palindrome*. Then the string q' I get by mirroring q in p with respect to p's center satisfies *palindrome*. This property holds for gapped palindromes.

The palindrome predicate. I have introduced six predicates for determining whether or not a string is a palindrome: besides the basic *palindrome* predicate, these are the predicates *textPalindrome*, *wordPalindrome*, *dnaPalindrome*, *approximatePalindrome,* and *gappedPalindrome*. It doesn't stop here, of course. The examples in this section show gapped text palindromes, and approximate text palindromes. The example of palindromes in male DNA requires finding gapped approximate DNA palindromes. Some DNA files use both capital and underscore letters for DNA symbols, and it follows that I have to find gapped approximate DNA text palindromes. The number of possible variants is substantial.

I redefine the *palindrome* predicate to accommodate all of the palindromic variants. The predicate now takes six arguments: two booleans denoting whether or not I want to find text or word palindromes, two integers denoting the length of the gap and the allowed number of errors, an equality operator, and a string in context.

$$
\begin{array}{l}
palindrome :: Bool \to Bool \to Int \to Int \to CharEq \to CString \to Bool \\
palindrome\ text\ word\ g\ k\ eq\ (before, s, after) \\
\quad \begin{array}{ll}
\mid text & = palindrome\ False\ False\ g\ k\ eq\ (before, lowerLetter\ s, after) \\
\mid word & = surroundedByPunctuation\ (before, s, after) \\
 & \wedge\ palindrome\ False\ False\ g\ k\ eq\ (before, lowerLetter\ s, after) \\
\mid g > 0 & = palindrome\ False\ False\ 0\ k\ eq\ (before, rmCenter\ g\ s, after) \\
\mid k > 0 & = levenshteinDistance\ eq\ s\ (reverse\ s) \leqslant k \\
\mid otherwise & = palindromeEq\ eq\ s
\end{array}
\end{array}
$$

Predicate *palindrome* combines the previous predicates in a single predicate, and also deals with combinations of palindromic aspects. The properties satisfied by *palindrome* are obtained by combining the properties for its components.

3 Finding palindromes

Both versions of the *palindrome* predicate defined in the previous section can be used to determine whether or not a string is a palindrome. The first version takes a number of steps linear in the length of the input string to do so. These predicates can be used to verify that a given string is a palindrome, but they are not very useful for finding the largest palindrome in the Bible, or for finding the gapped approximate text palindromes in DNA. This section discusses first which kind of palindromes we want to find, and then gives two algorithms for finding such palindromes.

3.1 Finding which palindromes?

Software for finding palindromes is particularly useful for finding palindromes in large documents. For example, I analysed the human Y chromosome, consisting of almost 25 million DNA symbols, and chromosome 18, consisting of

almost 75 million symbols. The typical questions about palindromes asked by geneticists are: "what are the longest palindromes occurring in this string", or "how many palindromes of length in between m and n occur in this string?" The question of where a particular palindromic string appears inside DNA is more a pattern-matching problem than a palindrome finding problem. Almost all of the palindrome-related questions can be answered relatively fast if I know the length of the longest palindrome around each position of the input string. A string of length n has $2n + 1$ *positions* (sometimes also called center position, or just center): the position before the first character, the positions of the characters, the positions in between two characters, and the position after the last character. For example, the list of the longest palindromes around each position in the string "abb" is `["","a","","b","bb","b",""]`. The EXTEND property says that if palindrome q is the longest palindrome around its center in a string s, then all strings obtained by removing equally many symbols from the front and the back of q are also palindromes, and none of its extensions is a palindrome. I call the longest palindrome around center a in the string s the *maximal palindrome* around center a in s. The list of all maximal palindromes in a string is a concise description of all palindromes that occur in the string. For a list consisting of n copies of the same symbol, the total length of the list of maximal palindromes is quadratic in n. An even more concise description of all palindromes that occur in a string is obtained by returning the list of *lengths* of maximal palindromes in a string. Given a center position and the length of the maximal palindrome around it, I can easily reconstruct all palindromes around that center. The resulting list of lengths of maximal palindromes has length $2n + 1$ for an input list of length n. In the following sections I will develop algorithms for finding the lengths of all maximal palindromes in a string.

3.2 A naive algorithm for finding palindromes

In this subsection I will describe the obvious algorithm for finding the length of all maximal palindromes in a string.

Given a string as input, I want to find the list of lengths of maximal palindromes around all centers of the string. I use the function *maximalPalindromes* for this purpose.

$$maximalPalindromes :: String \to [Int]$$

I want to find the length of the maximal palindrome around each center in a string. I will do this by trying to extend the trivial palindromes consisting of either a single letter (for odd centers, starting counting centers with 0) or of the empty string (for even centers) around each center. This only works for *palindrome* predicates satisfying the EXTEND and SINGLE property. If the predicate doesn't satisfy the SINGLE predicate, I only look at the even centers. To extend a palindrome, I have to compare the characters before and after the current palindrome. It would be helpful if I had random access into the string, so that looking up the character at a particular position in a string can be done

in constant time. Since an array allows for constant time lookup, I change the input type of *maximalPalindromes* to an array.

$$maximalPalindromes :: Array\ Int\ Char \rightarrow [Int]$$

If I change my input type from strings to arrays, I have to convert an input string into an array, for which I use the function *listArray* from the module *Data.Array*. Function *maximalPalindromes* calculates the length of maximal palindromes by first calculating all center positions of an input array, and then the length of the maximal palindrome around each of these centers.

```
maximalPalindromes a = let (first, last) = bounds a
                           centers       = [0 .. 2 * (last - first + 1)]
                       in  map (lengthPalindromeAround a) centers
```

Function *lengthPalindromeAround* takes an array and a center position, and calculates the length of the longest palindrome around that position.

```
lengthPalindromeAround :: Array Int Char -> Int -> Int
lengthPalindromeAround a center
  | even center  = lengthPalindrome (first + c - 1) (first + c)
  | odd  center  = lengthPalindrome (first + c - 1) (first + c + 1)
  where c                             = div center 2
        (first, last)                 = bounds a
        lengthPalindrome start end =
          if   start < 0 ∨ end > last - first ∨ a ! start ≢ a ! end
          then end - start - 1
          else lengthPalindrome (start - 1) (end + 1)
```

For each position, this function may take an amount of steps linear in the length of the array, so this is a worst-case quadratic-time algorithm. A more precise analysis shows that this algorithm is linear in the sum of the lengths of the palindromes found. The sum of the lengths of the palindromes in the King James Bible is less than twice the length of the Bible, so for this example this function behaves like a linear-time program. For determining palindromes in DNA, the situation is similar. The Y chromosome contains huge palindromes, but they hardly overlap. Chromosome 18 contains quite a few "ATAT"-sequences, but the longest of these has length 66, and almost all are much shorter.

3.3 Efficient algorithms for finding palindromes

Using the PALINPAL property, I now develop an algorithm for finding palindromes that requires a number of steps approximately equal to the length of its input. This linear-time algorithm can be used to find palindromes in documents of any size, and any content, even in very long strings consisting of the same symbol. Finding palindromes in a string of length $5,000,000$ using this algorithm requires a number of seconds on a modern laptop. It is impossible to find palindromes

substantially faster, unless you have a machine with many cores, and use a parallel algorithm.

The program for efficiently finding palindromes is only about 25 lines long. Although the program is short, it is rather intricate. I guess that you need to experiment a bit with to find out how and why it works.

The reason why the algorithm for finding palindromes from the previous subsection is naive is that *lengthPalindromeAround* calculates the maximal palindrome around a center independently of the palindromes calculated previously. I now change this by calculating the maximal palindromes from left to right around the centers of a string. In this calculation I either extend a palindrome around a center, or I move the center around which I determine the maximal palindrome rightwards because I have found a maximal palindrome around a center. So I replace the definition of *maximalPalindromes* by

$$\begin{array}{ll} \mathit{maximalPalindromes} & :: \mathit{Array\ Int\ Char} \rightarrow [\mathit{Int}] \\ \mathit{maximalPalindromes}\ a = & \mathbf{let}\ (\mathit{first}, \mathit{last}) = \mathit{bounds}\ a \\ & \mathbf{in}\ \ \mathit{reverse}\ (\mathit{extendPalindrome}\ a\ \mathit{first}\ 0\ [\,]) \end{array}$$

Before I introduce and explain function *extendPalindrome*, I give an example of how the algorithm works.

An example. I want to find the maximal palindromes in the string "yabadabadoo". The algorithm starts by finding the maximal palindrome around the position in front of the string, which cannot be anything else than the empty string. It moves the position around which to find the maximal palindrome one step to point to the 'y'. The maximal palindrome around this position is "y", since there is no character in front of it. It again moves the position around which to find palindromes one step to point to the position in between 'y' and 'a'. Since 'y' and 'a' are different, the maximal palindrome around this position is the empty string. Moving the center to 'a', it finds that "a" is the maximal palindrome around this center, since 'y' and 'b' are different. The maximal palindrome around the next center in between 'a' and 'b' is again the empty string. Moving the center to 'b', it can extend the current longest palindrome "b" around this center, since both before and after 'b' it finds an 'a'. It cannot further extend the palindrome "aba", since 'y' and 'd' are different. To determine the maximal palindrome around the center in between 'b' and 'a', the next center position, it uses the fact that "aba" is a palindrome, and that it already knows that the maximal palindrome around the center in between 'a' and 'b' is the empty string. Using the PALINPAL property, it finds that the maximal palindrome around the position in between 'b' and 'a' is also the empty string, without having to look at the 'b' and the 'a'. To determine the maximal palindrome around the next center position on the last 'a' of "aba", it has to determine if 'd' equals 'b', which it doesn't of course. Also here it uses the PALINPAL property. Since "a" is the maximal palindrome around the center of the first 'a' in "aba", and it reaches until the start of the palindrome "aba", I have to determine if the palindrome "a" around the second 'a' can be extended. I won't describe all steps

extendPalindrome takes in detail, but only give one more detail I already described above: the second occurrence of the palindrome "aba" in "yabadabadoo" is not found by extending the palindrome around its center, but by using the PALINPAL property to find "aba" a second time in "abadaba".

Function extendPalindrome. Function *extendPalindrome* takes four arguments. The first argument is the array a in which we are determining maximal palindromes. The second argument is the position in the array directly after the longest palindrome around the current center (the longest palindrome around the center before the first symbol has length 0, so the position directly after the empty palindrome around the first center is the first position in the array). I will call this the current *rightmost* position. The third argument is the length of the current longest palindrome around that center (starting with 0), and the fourth and final argument is a list of lengths of longest palindromes around positions before the center of the current longest tail palindrome, in reverse order (starting with the empty list [ ]). It returns the list of lengths of maximal palindromes around the centers of the array, in reverse order. Applying function reverse to the result gives the maximal palindromes in the right order. The function *extendPalindrome* maintains the invariant that the current palindrome is the longest palindrome that reaches until the current rightmost position.

There are three cases to be considered in function *extendPalindrome*. If the current position is after the end of the array, so *rightmost* is greater than *last*, I cannot extend the current palindrome anymore, and it follows that it is maximal. It only remains to find the maximal palindromes around the centers between the current center and the end of the array, for which I use the function *finalPalindromes*. If the current palindrome extends to the start of the array, or it cannot be extended, it is also maximal, and I add it to the list of maximal palindromes found. I then determine the maximal palindrome around the following center by means of the function *moveCenter*. If the element at the current rightmost position in the array equals the element before the current palindrome I extend the current palindrome.

```
extendPalindrome a rightmost curPal curMaxPals
  | rightmost > last =
      -- reached the end of the array
    finalPalindromes curPal curMaxPals (curPal : curMaxPals)
  | rightmost − curPal == first ∨ a ! rightmost ≢ a ! (rightmost − curPal − 1) =
      -- the current palindrome extends to the start
      -- of the array, or it cannot be extended
    moveCenter a rightmost (curPal : curMaxPals) curMaxPals curPal
  | otherwise =
      -- the current palindrome can be extended
    extendPalindrome a (rightmost + 1) (curPal + 2) curMaxPals
  where (first, last) = bounds a
```

In two of the three cases, function *extendPalindrome* finds a maximal palindrome, and goes on to the next center by means of function *finalPalindromes* or *move-*

Center. In the other case it extends the current palindrome, and moves the rightmost position one further to the right.

Function moveCenter. Function *moveCenter* moves the center around which the algorithm determines the maximal palindrome. In this function I make essential use of the PALINPAL property. It takes the array as argument, the current rightmost position in the array, the list of maximal palindromes to be extended, the list of palindromes around centers before the center of the current palindrome, and the number of centers in between the center of the current palindrome and the rightmost position. It uses the PALINPAL property to calculate the longest palindrome around the center after the center of the current palindrome.

If the last center is on the last element, there is no center in between the rightmost position and the center of the current palindrome. I call *extendPalindrome* with rightmost position one more than the previous position, and a current palindrome of length 1.

If the previous element in the list of maximal palindromes reaches exactly to the left end of the current palindrome, I use the PALINPAL property of palindromes to find the next current palindrome using *extendPalindrome*.

In the other case, I have found the longest palindrome around a center, add that to the list of maximal palindromes, and proceed by moving the center one position, and calling *moveCenter* again. I only know that the previous element in the list of maximal palindromes does not reach exactly to the left end of the current palindrome, so it might be either shorter or longer. If it is longer, I need to cut off the new maximal palindrome found, so that it reaches exactly to the current rightmost position.

```
moveCenter a rightmost curMaxPals prevMaxPals nrOfCenters
  | nrOfCenters == 0 =
      -- the last center is on the last element:
      -- try to extend the palindrome of length 1
    extendPalindrome a (rightmost + 1) 1 curMaxPals
  | nrOfCenters − 1 == head prevMaxPals =
      -- the previous maximal palindrome reaches
      -- exactly to the end of the last current
      -- palindrome. Use the palindromes in palindromes
      -- property to extend the current palindrome
    extendPalindrome a rightmost (head prevMaxPals) curMaxPals
  | otherwise =
      -- move the center one step. Add the length of
      -- the longest palindrome to the maximal
      -- palindromes
    moveCenter a
      rightmost
      (min (head prevMaxPals) (nrOfCenters − 1) : curMaxPals)
      (tail prevMaxPals)
      (nrOfCenters − 1)
```

In the first case, function *moveCenter* moves the rightmost position one to the right. Here we use the SINGLE property of *palindrome*. In the second case it calls *extendPalindrome* to find the maximal palindrome around the next center, and in the third case it adds a maximal palindrome to the list of maximal palindromes, and moves the center of the current palindromes one position to the right.

Function finalPalindromes. Function *finalPalindromes* calculates the lengths of the longest palindromes around the centers that come after the center of the current palindrome of the array. These palindromes are again obtained by using the palindromes in palindromes property. Function *finalPalindromes* is called when we have reached the end of the array, so it is impossible to extend a palindrome. We iterate over the list of maximal palindromes, and use the palindromes in palindromes property to find the maximal palindrome at the final centers. As in the function *moveCenter*, if the previous element in the list of maximal palindromes reaches before the left end of the current palindrome, I need to cut off the new maximal palindrome found, so that it reaches exactly to the end of the array.

```
finalPalindromes nrOfCenters prevMaxPals curMaxPals
  | nrOfCenters == 0 = curMaxPals
  | otherwise        =
    finalPalindromes
      (nrOfCenters - 1)
      (tail prevMaxPals)
      (min (head prevMaxPals) (nrOfCenters - 1) : curMaxPals)
```

In each step, function *finalPalindromes* adds a maximal palindrome to the list of maximal palindromes, and moves on to the next center.

I have discussed the number of steps this algorithm takes for each function. At a global level, this algorithm either extends the current palindrome, and moves the rightmost position in the array, or it extends the list of lengths of maximal palindromes, and moves the center around which we determine the maximal palindrome. If the length of the input array is n, the number of steps the algorithm is n for the number of moves of the rightmost position, plus $2n+1$ for the number of center positions. This is a linear-time algorithm.

3.4 Variants

The algorithm for finding palindromes given in the Section 3.2 applies to palindrome predicates satisfying the EXTEND property, and the algorithm in the Section 3.3 additionally requires the PALINPAL property. So the first algorithm can be used to find approximate palindromes, and neither can be used to find word palindromes.

Finding approximate palindromes. Approximate palindromes can be found using the algorithm in Section 3.2. If I only allow substitutions as edit operation, this

algorithm is linear in the sum of the lengths of the palindromes found, which might be quadratic in the length of the input string in the worst case, but is linear in almost all real-world applications. This raises two questions:

- How can I also deal with insertions and deletions as edit operations?
- Can I somehow extend the *approximatePalindrome* predicate or the linear-time algorithm for finding palindromes from Section 3.3 to also find approximate palindromes?

The first question is answered by applying standard dynamic programming techniques, as also used to determine the edit-distance between two strings. As for the second question: I have spent many hours on designing algorithms for finding approximate palindromes using the palindromes in palindromes concept, but failed. Anyone?

Finding word palindromes. Since the *wordPalindromes* predicate doesn't satisfy the various palindromic properties, none of the algorithms for finding palindromes can be used to find word palindromes. It is relatively easy to change the *wordPalindrome* property such that it satisfies an adapted EXTEND property. Instead of three, I now split a list into five components and I add a boolean *word*, $((before, (before', s, after'), after), word)$, such that the *string* consisting of $before' +\!\!+\ s +\!\!+\ after'$ is a text palindrome, and s is the longest word palindrome with the same center contained in *string* if *word* holds. If *word* doesn't hold, then there is no word palindromic substring with the same center.

```
type CString' = ((String, (String, String, String), String), Bool)
wordPalindrome' :: CString' → Bool
wordPalindrome' ((before, (before', s, after'), after), word) =
  let string = before' ++ s ++ after'
  in   textPalindrome string
     ∧ (¬ word
        ∨ wordPalindrome (before ++ before', s, after' ++ after)
            ∧ ((null before' ∧ null after')
                ∨ ( and
                    ∘ map (¬ ∘ surroundedByPunctuation)
                    ∘ sameCenterSubstrings
                    $ (before, init before' ++ tail after', after)
                    )
                )
        )
sameCenterSubstrings                        :: CString → [CString]
sameCenterSubstrings (before, []  , after) = [(before, []  , after)]
sameCenterSubstrings (before, [a], after) = [(before, [a], after)]
sameCenterSubstrings (before, xs , after) =
    (before, xs, after)
  : sameCenterSubstrings (before ++ [head xs], tail (init xs), last xs : after)
```

I adapt the EXTEND property by requiring the concatenation of the three strings in the middle to be a text palindrome, and by calculating from the text palindrome the contained word palindrome, if such a word palindrome exists. Using this property, I can now develop a quadratic-time algorithm for finding word palindromes.

4 Conclusions

I have introduced several variants of the palindrome problem, the palindromic properties satisfied by these variants, and two algorithms that can be used to find palindromic substrings, depending on the properties satisfied by the particular palindromic variant sought. The description of the variants and their properties is new to my knowledge; the algorithms for finding palindromes have already been described in the last century by Galil, Manacher, myself, and others [1,4,2].

Acknowledgements. I discussed many aspects of finding palindromes in DNA with Anjana Ramnath of the Indian Institute of Science on Bioinformatics and Computational Biology. Jennifer Hughes of the Whitehead Institute of the MIT Department of Biology helped me finding approximate palindromes with gaps in the male DNA. Bastiaan Heeren commented on a previous version of this paper.

References

1. Zvi Galil and Joel Seiferas. A linear-time on-line recognition algorithm for "palstar". *Journal of the ACM*, 25:102–111, January 1978.
2. Johan Jeuring. The derivation of on-line algorithms, with an application to finding palindromes. *Algorithmica*, 11:146–184, 1994.
3. Johan Jeuring. The history of finding palindromes. In *Liber Amicorum Doaitse Swierstra*. Department of Information and Computing Sciences, Utrecht University, 2012.
4. Glenn Manacher. A new linear-time 'on-line' algorithm for finding the smallest initial palindrome of a string. *Journal of the ACM*, 22:346–351, 1975.
5. Helen Skaletsky, Tomoko Kuroda-Kawaguchi, Patrick J. Minx, Holland S. Cordum, LaDeana Hillier, Laura G. Brown, Sjoerd Repping, Tatyana Pyntikova, Johar Ali, Tamberlyn Bieri, Asif Chinwalla, Andrew Delehaunty, Kim Delehaunty, Hui Du, Ginger Fewell, Lucinda Fulton, Robert Fulton, Tina Graves, Shun-Fang Hou, Philip Latrielle, Shawn Leonard, Elaine Mardis, Rachel Maupin, John McPherson, Tracie Miner, William Nash, Christine Nguyen, Philip Ozersky, Kymberlie Pepin, Susan Rock, Tracy Rohlfing, Kelsi Scott, Brian Schultz, Cindy Strong, Aye Tin-Wollam, Shiaw-Pyng Yang, Robert H. Waterston, Richard K. Wilson, Steve Rozen, and David C. Page. The male-specific region of the human y chromosome is a mosaic of discrete sequence classes. *Nature*, 423(6942):825–837, 2003.