Efficient Recognition of Abelian Palindromic Factors and Associated Results

Costas S. Iliopoulos, Steven Watts

Efficient Recognition of Abelian Palindromic Factors and Associated Results

Costas S. Iliopoulos and Steven Watts (✉)

Department of Informatics, King's College London, UK
[costas.iliopoulos, steven.watts]@kcl.ac.uk

Abstract. A string is called a *palindrome* if it reads the same from left to right. In this paper we define the new concept of an *abelian palindrome* which satisfies the property of being abelian equivalent to some palindrome of the same length. The identification of abelian palindromes presents a novel combinatorial problem, with potential applications in filtering strings for palindromic factors. We present an algorithm to efficiently identify abelian palindromes, and additionally generate an *abelian palindromic array*, indicating the longest abelian palindrome at each location. Specifically, for an alphabet of size $|\Sigma| \leq \log_2(n)$ and after $\mathcal{O}(n)$ time preprocessing using $\mathcal{O}(n + |\Sigma|)$ space, we may determine if any factor is abelian palindromic in $\mathcal{O}(1)$ time. Additionally, we may determine the abelian palindromic array in $\mathcal{O}(|\Sigma|n)$ time. We further specify the algorithmic complexity when this condition on alphabet size $|\Sigma|$ is relaxed.

1 Introduction

The identification of palindromic factors in strings, has been a much studied area of stringology, due to the interesting combinatorial aspects and the strong ties with genetic analysis, where palindromes often correspond to significant structures in DNA [3].

Variations of the palindrome identification problem have been frequently introduced, for example Karhumäki et al. presented results on k-abelian palindromes on rich and poor words [2]. Holub et al. considered the problem as applied to binary words, investigating the properties of palindromic factors of binary strings [4].

We introduce our own simple modification to the problem, yet to be explored, namely abelian palindromes. Though an interesting combinatorial problem in it itself, an efficient method of detecting abelian palindromes can potentially provide a filter by which ordinary palindromic factors may be deduced. This follows from the fact that an ordinary palindrome must necessarily also be an abelian palindrome, and therefore the search space may be reduced if non abelian palindromes can be efficiently dismissed.

Likewise, the abelian palindromic array may potentially be used to assist in the calculation of the ordinary palindromic array, for the purpose of performing a greedy factorisation of a string into ordinary palindromes. This follows from the fact that the abelian palindromic array provides an upper bound for the equivalent value in the ordinary palindromic array.

The rest of this paper is organised as follows. In Section 2, we present basic definitions and notation on strings as well as definitions and results on abelian palindromes.

Section 3 presents various data structures and algorithmic tools used in our final algorithms. In Section 4, a detailed implementation of our algorithms are presented, the first being identification of abelian palindromes, the second being the generation of the abelian palindromic array. Our concluding remarks are noted in Section 5. Additionally, the pseudocode of our implementation may be found in Section 6.

2 Preliminaries

2.1 Basic Terminology

We begin with basic definitions and notation from [1]. Let $x = x[0]x[1]\dots x[n-1]$ be a *string* of length $|x| = n$ over a finite ordered alphabet. We consider the case of strings over an *integer alphabet* Σ: each character may be replaced by its lexicographical rank in such a way that the resulting string consists of integers in the range $\{0, \dots, n-1\}$. We use $\text{ORD}(x[i])$ to refer to the lexicographical rank of the character $x[i]$. We use $\Sigma[i]$ to refer to the ith character of Σ, i.e. $\Sigma[\text{ORD}(x[i])] = x[i]$. For example, for the alphabet $\Sigma = \{\texttt{a}, \texttt{c}, \texttt{g}, \texttt{t}\}$ we have $\text{ORD}(\texttt{g}) = 2$ and $\Sigma[2] = \texttt{g}$.

For two positions i and j on x, we denote by $x[i\,..\,j] = x[i]\dots x[j]$ the *factor* (sometimes called *substring*) of x that starts at position i and ends at position j (it is of length 0 if $j < i$), and by ε the *empty string* of length 0. We recall that a *prefix* of x is a factor that starts at position 0 ($x[0\,..\,j]$) and a *suffix* of x is a factor that ends at position $n-1$ ($x[i\,..\,n-1]$).

Let y be a string of length m with $0 < m \leq n$. We say that there exists an *occurrence* of y in x, or, more simply, that y *occurs in* x, when y is a factor of x. Every occurrence of y can be characterised by a starting position in x. Thus we say that y occurs at the *starting position* i in x when $y = x[i\,..\,i+m-1]$.

We denote the *reverse* string of x by x^R as the string obtained when reading x from right to left, i.e. $x^R = x[n-1]x[n-2]\dots x[1]x[0]$. We say a string x is a *palindrome* when $x = x^R$.

We make use of the bit-wise *exclusive or* (XOR) operation between two binary strings x and y of the same length $|x| = |y|$, denoted $x \oplus y$. This adheres to the standard definition of XOR on two binary strings, i.e. $z[i] = (x[i] + y[i]) \pmod 2$ where $z = x \oplus y$. We may similarly apply the XOR operation to integers, $x, y \in \mathbb{Z}$ by converting x and y to their respective binary equivalents, performing the XOR operation, and converting the binary result into an integer. For example, given $x = 5, y = 11$ we have $x \oplus y = 5 \oplus 11 = \texttt{0101} \oplus \texttt{1011} = \texttt{1110} = 14$.

2.2 Abelian Palindromes

The concept of abelian strings relates to the idea of disregarding the order of appearance of characters in a string, and concerning ourselves only with the number of occurrences of each character within the string. With this in mind, we wish to define the concept of an *abelian palindrome*. To facilitate this, we must first recall the definition of a *Parikh Vector*.

Definition 1. *The* Parikh vector $\mathcal{P}(T)$ *of a string* T *over the alphabet* Σ*, is a vector of size* $|\Sigma|$ *which enumerates the number of occurrences of each character of the alphabet in* T*. If the character* $c \in \Sigma$ *has ordinality* $i = \text{ORD}(c)$ *in the lexicographical ordering of the alphabet* Σ*, then* $\mathcal{P}(T)[i]$ *stores the number of occurrences of* c *in* T*.*

We say that two strings T_1 *and* T_2 *are* abelian equivalent *denoted* $T_1 \approx_p T_2$ *if and only if they have the same Parikh vector, i.e. are permutations of each other.*

For example, the string $T_1 = \texttt{accgta}$ has the Parikh vector $\mathcal{P}(T_1) = (2, 2, 1, 1)$. The string $T_2 = \texttt{gactcac}$ has the same Parikh vector and thus $T_1 \approx_p T_2$. We may now define the concept of an abelian palindrome.

Definition 2. *A string* T *is an* abelian palindrome *if and only if there exists some palindrome* P *such that* $P \approx_p T$*.*

Note that in general, a string T will more easily satisfy the abelian palindromic property over the palindromic property. This comes as a direct result of Lemma 1, which follows clearly from Definition 2.

Lemma 1.

$$T \text{ palindromic} \implies T \text{ abelian palindromic} \tag{1}$$

$$T \text{ not abelian palindromic} \implies T \text{ not palindromic} \tag{2}$$

Proof. Assume T is palindromic, choose $P = T$. Therefore we have $P = T \approx_p T$. Thus T is abelian palindromic and Statement 1 is proven. Statement 2 follows as the contrapositive of Statement 1. □

3 Tools

3.1 Initial Observations

We wish to efficiently identify abelian palindromic factors within a string. To enable this, we define some further concepts and auxiliary data structures.

From Definition 2, it is clear that whether a string T is an abelian palindrome is dependant on the values in its Parikh vector $\mathcal{P}(T)$, specifically the number of values that are odd or even. We use $|\mathcal{P}(T)|$ to refer to the total number of values in $\mathcal{P}(T)$, and further use $|(\mathcal{P}(T))|_{\texttt{odd}}$ and $|(\mathcal{P}(T))|_{\texttt{even}}$ to refer to the number of odd and even values in $\mathcal{P}(T)$ respectively. This notation allows us to succinctly describe the defining quality of an abelian palindrome in Lemma 2.

Lemma 2. T *abelian palindromic* $\iff 0 \leq |(\mathcal{P}(T))|_{\texttt{odd}} \leq 1$.

Proof. We refer to the length of T as n. We call $T_l = T[0\,..\,\lfloor \frac{n-1}{2} - 1 \rfloor]$ the *left half* of T and $T_r = T[\lceil \frac{n-1}{2} + 1 \rceil\,..\,n-1]$ the *right half* of T. Note that if n is even, $T = T_l\, T_r$. If n is odd, $T = T_l\, c\, T_r$ where $c = T[\frac{n-1}{2}]$. For an ordinary palindrome P, it is clear that if a character s occurs m times in P_l it must correspondingly occur m times in P_r, to preserve the palindromic property of P.

We first show that if T is an abelian palindrome, the number of odd values in the Parikh vector $|(\mathcal{P}(T))|_{\texttt{odd}}$ can not exceed 1 by contradiction. Let us assume that $|(\mathcal{P}(T))|_{\texttt{odd}} > 1$. In this case, we have at least 2 different characters $s_1, s_2 \in \Sigma$ with an odd number of occurrences in T. For any permutation of the characters in T, at least one of these two characters must have all its occurrences contained entirely within T_l and T_r, and we call this character s. The character s therefore occurs $2m$ times in T where m is the number of occurrences of s in T_l. Therefore s has an even number of occurrences, which leads us to a contradiction. Therefore we conclude that $0 \leq |(\mathcal{P}(T))|_{\texttt{odd}} \leq 1$.

We now show that we can always form a palindrome from a permutation of T when $0 \leq |(\mathcal{P}(T))|_{\texttt{odd}} \leq 1$. In the notation below we use P_l^R to represent the reversal of P_l.

Let us assume $|(\mathcal{P}(T))|_{\texttt{odd}} = 0$. In this case, $\mathcal{P}(T)$ contains only even values. We distribute the characters evenly to form an even length palindrome P such that $P \approx_p T$ as follows (with braces under characters indicating the number of repetitions of that character):

$$P_l = \underbrace{\Sigma[0]}_{\frac{1}{2}\mathcal{P}(T)[0]} \underbrace{\Sigma[1]}_{\frac{1}{2}\mathcal{P}(T)[1]} \ \ldots \ \underbrace{\Sigma[|\Sigma|-1]}_{\frac{1}{2}\mathcal{P}(T)[|\Sigma|-1]}$$

$$P = P_l \; P_l^R$$

Now let us assume $|(\mathcal{P}(T))|_{\texttt{odd}} = 1$. In this case, $\mathcal{P}(T)$ contains a single odd entry corresponding to some character $c = \Sigma[i]$. We distribute the characters evenly, placing the character c at the centre, to form an odd length palindrome P such that $P \approx_p T$ as follows:

$$P_l = \underbrace{\Sigma[0]}_{\frac{1}{2}\mathcal{P}(T)[0]} \ \ldots \ \underbrace{\Sigma[i-1]}_{\frac{1}{2}\mathcal{P}(T)[i-1]} \underbrace{\Sigma[i+1]}_{\frac{1}{2}\mathcal{P}(T)[i+1]} \ \ldots \ \underbrace{\Sigma[|\Sigma|-1]}_{\frac{1}{2}\mathcal{P}(T)[|\Sigma|-1]} \underbrace{\Sigma[i]}_{\frac{1}{2}(\mathcal{P}(T)[i]-1)}$$

$$P = P_l \; \Sigma[i] \; P_l^R$$

Thus we have shown that $0 \leq |(\mathcal{P}(T))|_{\texttt{odd}} \leq 1$ is both a necessary and sufficient condition for T to be an abelian palindrome. Therefore Lemma 2 follows. □

3.2 Prefix Parity Integer Array

We aim to describe a new data structure that will prove useful in recognising palindromic factors, beginning with some new definitions.

Lemma 2 provides us with a useful criterion by which we can seek longest abelian palindromes. We first provide some additional definitions which will prove useful.

Definition 3. *A* prefix Parikh vector $\mathcal{P}_i(T)$ *of a string* T *is the Parikh vector of the* i*th prefix of* T*:*

$$\mathcal{P}_i(T) = \mathcal{P}(T[0\,..\,i]) \textit{ for } 0 \leq i \leq n-1$$

Definition 4. *A* parity vector $\mathbb{P}(T)$ *of a string* T *over the alphabet* Σ *is a bit vector of length* Σ *which indicates the parity (even or odd) of the number of occurrences of each character of* Σ *in* T *(0 indicates* `even`*, 1 indicates* `odd`*):*

$$\mathbb{P}(T)[i] = \mathcal{P}(T)[i] \pmod 2$$

Definition 5. *A* prefix parity vector $\mathbb{P}_i(T)$ *of a string* T *is the parity vector of the* i*th prefix of* T*:*

$$\mathbb{P}_i(T) = \mathbb{P}(T[0\,..\,i]) \text{ for } 0 \leq i \leq n-1$$

Definition 6. *A* parity integer $\hat{\mathbb{P}}(T)$ *of a string* T *over the alphabet* Σ *is a decimal integer representing the value of the parity vector* $\mathbb{P}(T)$ *when interpreted as a binary number, with the order of magnitude of each bit determined by the lexicographical order of the alphabet* Σ*:*

$$\hat{\mathbb{P}}(T) = \sum_{i=0}^{|\Sigma|-1} 2^i \times \mathbb{P}(T)[i]$$

Definition 7. *A* prefix parity integer $\hat{\mathbb{P}}_i(T)$ *of a string* T *is the parity integer of the* i*th prefix of* T*:*

$$\hat{\mathbb{P}}_i(T) = \hat{\mathbb{P}}(T[0\,..\,i]) \text{ for } 0 \leq i \leq n-1$$

Definition 8. *The* prefix parity integer array $\hat{\mathbb{P}}_A(T)$ *of a string* T *of length* n *is an integer array of length* n*, which contains the value of* $\hat{\mathbb{P}}_i(T)$ *at each position* i*:*

$$\hat{\mathbb{P}}_A(T)[i] = \hat{\mathbb{P}}_i(T)$$

The prefix parity integer array $\hat{\mathbb{P}}_A(T)$ (example: see bottom of Figure 1) is the key to identifying longest abelian palindromes in a string T. To observe this, we note that the Parikh vector of a factor of T can be determined by evaluating the difference between the two prefix Parikh vectors at the start and end indexes of the factor. The parity vector and parity integer of a factor can also be determined in a similar way, by employing the bit-wise exclusive or (XOR) operation. We summarise these observations in Lemma 3.

Lemma 3. *Given a string* T*:*

$$\mathcal{P}(T[i\,..\,j]) = \mathcal{P}_j(T) - \mathcal{P}_{i-1}(T) \tag{1}$$

$$\mathbb{P}(T[i\,..\,j]) = \mathbb{P}_j(T) \oplus \mathbb{P}_{i-1}(T) \tag{2}$$

$$\hat{\mathbb{P}}(T[i\,..\,j]) = \hat{\mathbb{P}}_j(T) \oplus \hat{\mathbb{P}}_{i-1}(T) \tag{3}$$

Proof. Given a factor $F = T[i\,..\,j]$ we have $T[0\,..\,j] = T[0\,..\,i-1]\,F$. Therefore it follows that $\mathcal{P}(T[0\,..\,j]) = \mathcal{P}(T[0\,..\,i-1]) + \mathcal{P}(F) \implies \mathcal{P}(F) = \mathcal{P}(T[0\,..\,j]) - \mathcal{P}(T[0\,..\,i-1])$. Thus Statement 1 is proven.

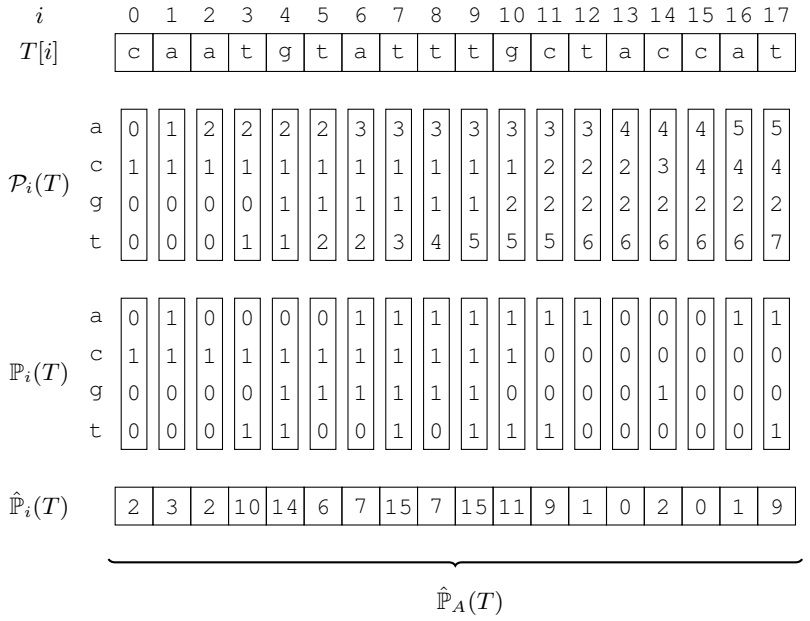

Fig. 1. Example of prefix Parikh vectors, prefix parity vectors and prefix parity integers.

Statement 2 follows from Statement 1 by observing that the truth table for the XOR operator is analogous to the parity table for the subtraction operator (when we interpret 0 as `even`, 1 as `odd`):

$-$	`even`	`odd`
`even`	`even`	`odd`
`odd`	`odd`	`even`

$\oplus$	0	1
0	0	1
1	1	0

Note that subtraction $\pmod 2$ and XOR are both commutative operations, and therefore the order of operations is unimportant for both tables.

Statement 3 is simply an alternative formulation of Statement 2 in the form of parity integers instead of parity vectors. □

Given $\hat{\mathbb{P}}_A(T)$, it now becomes simple to verify whether a factor $T[i\,..\,j]$ is an abelian palindrome, i.e. $0 \leq |\mathcal{P}(T[i\,..\,j])|_{\texttt{odd}} \leq 1$.

Lemma 4. *Given a text T over the alphabet Σ, the following holds:*

$$T[i\,..\,j] \textit{ abelian palindromic} \iff \hat{\mathbb{P}}_j(T) \oplus \hat{\mathbb{P}}_{i-1}(T) \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\}$$

Proof. The lemma follows from the application of previously defined lemmas. We use brackets under runs of characters to indicate the length of that run of characters:

$$T[i\,..\,j] \text{ is abelian palindromic}$$

$$\underset{\text{Lem. 2}}{\iff} 0 \leq |\mathcal{P}(T[i\,..\,j])|_{\texttt{odd}} \leq 1$$

$$\iff |\mathcal{P}(T[i\,..\,j])|_{\texttt{odd}} = 0 \ \vee \ |\mathcal{P}(T[i\,..\,j])|_{\texttt{odd}} = 1$$

$$\underset{\text{Def. 4}}{\iff} \mathbb{P}(T[i\,..\,j]) = \underbrace{0\ldots0}_{|\Sigma|} \vee \mathbb{P}(T) \in \{\underbrace{0\ldots0}_{|\Sigma|-1}1\ ,\ \underbrace{0\ldots0}_{|\Sigma|-2}10\ ,\ \ldots\ ,\ 1\underbrace{0\ldots0}_{|\Sigma|-1}\}$$

$$\underset{\text{Def. 6}}{\iff} \mathbb{P}(T[i\,..\,j]) = 0 \ \vee \ \hat{\mathbb{P}}(T) \in \{2^0, 2^1, 2^2, \ldots, 2^{|\Sigma|-1}\}$$

$$\underset{\text{Lem. 3}}{\iff} \hat{\mathbb{P}}_j(T) \oplus \hat{\mathbb{P}}_{i-1}(T) \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\}$$

□

Lemma 4 immediately leads us to Lemma 5, which allows us to identify the longest factor of a string T starting at i which is abelian palindromic.

Lemma 5. *Given a text T over the alphabet Σ, the longest abelian palindromic factor of T occurring at position i is $T[i\,..\,j]$ where j satisfies the following:*

$$j = \max\{j' : \hat{\mathbb{P}}_{j'}(T) \in M(T,i)\}$$
$$M(T,i) = \{\hat{\mathbb{P}}_{i-1}(T) \oplus k : k \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\}\}$$

For a given string T and position i, we call $M(T,i)$ the match set.

Proof. For a fixed i, the longest $T[i\,..\,j]$ which is abelian palindromic is found by determining the largest j, such that i and j satisfy the condition in Lemma 4. We may derive the match set $M(T,i)$ from this condition by employing the fact that XOR is commutative:

$$\hat{\mathbb{P}}_{j'}(T) \oplus \hat{\mathbb{P}}_{i-1}(T) \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\}$$

$$\iff \exists k \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\} \text{ such that } \hat{\mathbb{P}}_{j'}(T) \oplus \hat{\mathbb{P}}_{i-1}(T) = k$$

$$\iff \exists k \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\} \text{ such that } \hat{\mathbb{P}}_{j'}(T) = \hat{\mathbb{P}}_{i-1}(T) \oplus k$$

$$\iff \mathbb{P}_{j'}(T) \in \{\hat{\mathbb{P}}_{i-1}(T) \oplus k : k \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\}\} = M(T,i)$$

Thus for a given i, the largest j' satisfying the above condition gives us the j corresponding to the largest abelian palindromic factor $T[i\,..\,j]$. □

3.3 Rightmost Array

We describe a simple data structure that will prove useful for identifying longest abelian palindromes.

Definition 9. *The* rightmost array $\mathcal{R}(A)$ *of an integer array* A *of length* n *over the alphabet* $\{0, \ldots, n-1\}$ *stores at position* i *the index of the rightmost occurrence of the integer* i *in* A*. If there is no occurrence of* i *in* A *then* $A[i] = -1$*. Formally stated:*

$$\mathcal{R}(A)[i] = k \iff A[k] = i \ \wedge \ A[k'] \neq i \ \ \forall k' > k$$
$$\mathcal{R}(A)[i] = -1 \iff A[k] \neq i \ \ \forall k$$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$A[i]$	2	3	2	10	14	6	7	15	7	15	11	9	1	0	2	0	1	9
$\mathcal{R}(A)[i]$	15	16	14	1	-1	-1	5	8	-1	17	3	10	-1	-1	4	9	-1	-1

Fig. 2. Example of rightmost array.

4 Algorithms

4.1 Abelian Palindromic Factor Recognition

ABELIAN PALINDROMIC FACTOR RECOGNITION
Input: A string T of length n.
Output: A function F : $\{0\,..\,n-1\} \times \{0\,..\,n-1\} \to \{\texttt{true}, \texttt{false}\}$ where $F(i, j)$ returns `true` if $T[i\,..\,j]$ is abelian palindromic and `false` if $T[i\,..\,j]$ is not abelian palindromic, in $\mathcal{O}(1)$ time.

Our algorithm to generate a function recognising abelian palindromic factors, relies on the construction of the prefix parity integer array $\hat{\mathbb{P}}_A(T)$. As shown in Lemma 4, we are able to determine if $T[i\,..\,j]$ is abelian palindromic by evaluating the truthfulness of the expression $\hat{\mathbb{P}}_j(T) \oplus \hat{\mathbb{P}}_{i-1}(T) \in \{0\} \cup \{2^0, 2^1, \ldots, 2^{|\Sigma|-1}\}$.

Given $\hat{\mathbb{P}}_A(T)$, this expression may be evaluated in $\mathcal{O}(1)$ time for a given i and j. This follows from the fact that XOR is a constant time operation. Additionally, we may check if an integer is a power of 2 in constant time by employing the $(\mathrm{mod}\ 2)$ operation.

It is important to note, that these operations are constant time under the assumption that their arguments do not exceed the maximum word size w of the computer implementation used. If we assume that the alphabet size is bounded by the logarithm of n, then this assumption holds, i.e. $|\Sigma| \leq \log_2(n)$. Alternatively, the limitation on $|\Sigma|$ need not depend on n, and may instead be expressed in terms of the word size w of a machine. If the word size is w, we may assume these operations are constant for an alphabet size $|\Sigma| \leq w$. For a larger $|\Sigma|$, the expression in Lemma 4 may be evaluated in $\mathcal{O}(\frac{|\Sigma|}{w})$ time.

We now consider the construction of $\hat{\mathbb{P}}_A(T)$. It is possible to construct the array directly while maintaining a single instance of $\hat{\mathbb{P}}_i(T)$, by Lemma 6.

Lemma 6.

$$\hat{\mathbb{P}}_A(T)[0] \quad = 2^{\text{ORD}(T[0])}$$

$$\hat{\mathbb{P}}_A(T)[i] \quad = \hat{\mathbb{P}}_A(T)[i-1] + (2\,\mathbb{P}_i(T)[\text{ORD}(T[i])] - 1) \times 2^{\text{ORD}(T[i])} \quad 0 < i \leq n-1$$

Proof. The case for $\hat{\mathbb{P}}_A(T)[0]$ is trivially true. We note that $\hat{\mathbb{P}}_A(T)[i] = \hat{\mathbb{P}}_i(T)$ is an integer representation of $\mathbb{P}_i(T)$ interpreted as a binary string. $\mathbb{P}_i(T)$ and $\mathbb{P}_{i-1}(T)$ differ by a single bit flip, corresponding to the character encountered at $T[i]$. Therefore by Definition 6 and 7, $\hat{\mathbb{P}}_i(T)$ and $\hat{\mathbb{P}}_{i-1}(T)$ will accordingly differ by a single power of 2, specifically $2^{\text{ORD}(T[i])}$.

Whether $2^{\text{ORD}(T[i])}$ should be added or subtracted is dependant on the current parity of the character $T[i]$. This is determined by $\mathbb{P}_i(T)[\text{ORD}(T[i])]$, with 1 corresponding to addition $(+1)$ and 0 corresponding to subtraction (-1).

Thus the mapping $2b - 1$ where $b \in \{0, 1\}$ is the most recently flipped bit $b = \mathbb{P}_i(T)[\text{ORD}(T[i])]$, indicates the appropriate addition $(+1)$ or subtraction (-1). □

With this iterative equation for $\hat{\mathbb{P}}_A(T)$, we now have all the tools necessary to efficiently determine abelian palindromic factors and solve the problem as stated. We formalise the result in Theorem 1.

Theorem 1. *Given a string T of length n over the alphabet Σ, after $\mathcal{O}(n)$ time preprocessing and $\mathcal{O}(n + |\Sigma|)$ space, we may perform queries to determine if $T[i\,..\,j]$ is abelian palindromic in $\mathcal{O}(1)$ time when $|\Sigma| \leq \log_2(n)$.*

Additionally with no constraint on the size of Σ, with $\mathcal{O}(\frac{|\Sigma|}{w}n)$ time preprocessing we may perform such queries in $\mathcal{O}(\frac{|\Sigma|}{w})$ time, where w is the computer word size.

Proof. By using Lemma 6 we may iteratively construct $\hat{\mathbb{P}}_A(T)[i]$ from $\hat{\mathbb{P}}_A(T)[i-1]$ in $\mathcal{O}(1)$ time at each step, while maintaining the Σ-sized data structure $\mathbb{P}_i(T)$, resulting in a total time complexity $\mathcal{O}(n)$ and space complexity $\mathcal{O}(n + |\Sigma|)$ to construct $\hat{\mathbb{P}}_A(T)$.

By evaluating the expression on $\hat{\mathbb{P}}_A(T)$ in Lemma 4, we may then determine if $T[i\,..\,j]$ is abelian palindromic. Evaluating this expression may be performed in $\mathcal{O}(1)$ time when the number of bits required to store $\hat{\mathbb{P}}_i(T)$ is no larger than a single computer word w, i.e. when $|\Sigma| \leq \log_2(n) \leq w$. In general, $\hat{\mathbb{P}}_i(T)$ may be stored in $|\Sigma|$ bits, requiring $\lceil \frac{|\Sigma|}{w} \rceil$ words to store, and thus a multiplying factor of $\mathcal{O}(\frac{|\Sigma|}{w})$ time is required for all operations involving $\hat{\mathbb{P}}_i(T)$, both when constructing $\hat{\mathbb{P}}_A(T)$ and when evaluating the expression in Lemma 4, corresponding to a single query. □

4.2 Abelian Palindromic Array Algorithm

ABELIAN PALINDROMIC ARRAY
Input: A string T of length n.
Output: An array P of size n such that $A[i]$ stores the length of the longest abelian factor of T occurring at position i, i.e. as a prefix of $T[i \,..\, n-1]$.

Our algorithm to generate the abelian palindromic array makes use of Theorem 1 and the rightmost array described in Definition 9. We also make use of Lemma 7.

Lemma 7. *The abelian palindromic array P of a string T satisfies:*

$$P[i] = \max\{\mathcal{R}(j) : j \in M(T, \hat{\mathbb{P}}_A(T)[i])\}$$

Where M is the match set as described in Lemma 5.

Proof. Lemma 5 indicates that the longest abelian palindromic factor occurring at i is $T[i \,..\, j]$ where j is the index of the rightmost prefix parity integer with a value contained in the match set $M(T, i)$.

By Definition 9, this rightmost j can be found by taking the largest value obtained when querying the rightmost array with every member of the match set. □

Theorem 2. *Given a string T of length n over the alphabet Σ, we may determine the abelian palindromic array of T in $\mathcal{O}(|\Sigma|n)$ time and $\mathcal{O}(n + |\Sigma|)$ space, when $|\Sigma| \leq \log_2(n)$.*

Proof. Via the proof in Theorem 1 we are able to calculate the prefix parity integer array $\hat{\mathbb{P}}_A(T)$ in $\mathcal{O}(n)$ time and with $\mathcal{O}(n + |\Sigma|)$ space.

Since $|\Sigma| \leq \log_2(n)$, we know all values of $\mathcal{R}(A)[i] \in \{-1, 0, \ldots, n-1\}$. Therefore the rightmost array $\mathcal{R}(\hat{\mathbb{P}}_A(T))$ may be calculated in $\mathcal{O}(n)$ time, by parsing $\hat{\mathbb{P}}_A(T)$ from right to left and storing any new values encountered. Full details are available in the pseudocode in Section 6.

We now apply Lemma 7, which enables us to determine the longest abelian palindromic factor occurring at i by performing $|\Sigma|$ constant time queries. Thus a total of $\mathcal{O}(|\Sigma|n)$ constant time queries are required, and the total time complexity to generate the abelian palindromic array is $\mathcal{O}(|\Sigma|n)$. □

5 Conclusion

We have presented two algorithms, the first for recognising whether or not a factor is abelian palindromic, and the second for generating an array which provides the length of the longest abelian palindromic factor at each position in a string.

The proposed algorithms are both dependant on a new data structure called the prefix parity integer array, requiring $\mathcal{O}(n)$ time to compute for a string with an alphabet size $|\Sigma| \leq \log_2(n)$. Additional complexity is required to determine the longest abelian palindromic factor for each position, namely $\mathcal{O}(|\Sigma|n)$ time.

The main improvement in this work, would be to remove the need for the current requirement that $|\Sigma| \leq \log_2(n)$, in order to obtain our current best complexity time. This appears to be a reasonable goal.

6 Pseudocode

Algorithm Abelian Palindromes

```
1: function GETPREFIXPARITYINTEGERARRAY(T, Σ)
2:     n = |T|
3:     σ = |Σ|
4:     A = integer array of length n filled with 0                ▷ stores final result
5:     B = integer array of length σ filled with 0                ▷ stores powers of 2
6:     B[0] = 1
7:
8:     for i = 1 to σ − 1 do
9:         B[i] = 2 × B[i − 1]
10:    end for
11:
12:    ℙ = boolean array of length σ filled with 0
13:    prev = 0
14:
15:    for i = 0 to n − 1 do
16:        if ℙ[ORD(T[i])] == 1 then          ▷ ORD is the 0-indexed lexicographical order
17:            A[i] = prev − B[ORD(T[i])]
18:        else
19:            A[i] = prev + B[ORD(T[i])]
20:        end if
21:
22:        ℙ[ORD(T[i])] = not ℙ[ORD(T[i])]          ▷ not 1 = 0, not 0 = 1
23:        prev = A[i]
24:    end for
25:
26:    return A
27: end function
```

Algorithm Abelian Palindromes

```
1: function GETRIGHTMOSTARRAY(A)
2:     n = |A|
3:     R = integer array of length n filled with -1               ▷ stores final result
4:
5:     for i = n − 1 to 0 do                                      ▷ parses A from right to left
6:         if R[A[i]] == −1 then
7:             R[A[i]] = i
8:         end if
9:     end for
10:
11:    return R
12: end function
```

Algorithm Abelian Palindromes

```
1: function GETMATCHSET(x, n)
2:     M = integer array of length n + 1 filled with 0                ▷ stores final result
3:
4:     for i = 0 to n − 1 do
5:         M[i] = x ⊕ 2^i                                             ▷ XOR operation
6:     end for
7:
8:     M[n] = x
9:
10:    return M
11: end function
```

Algorithm Abelian Palindromes

```
1: function GETABELIANPALINDROMICARRAY(T, Σ)
2:     n = |T|
3:     σ = |Σ|
4:     A = GETPREFIXPARITYINTEGERARRAY(T, Σ)
5:     R = GETRIGHTMOSTARRAY(A)
6:     P = integer array of length n filled with 0                    ▷ stores final result
7:
8:     for i = 0 to n − 1 do
9:         M = GETMATCHSET(A[i − 1], σ)                               ▷ A[−1] defined as 0
10:
11:        rightmostMatch = −1
12:
13:        for each match in M do
14:            if R[match] > rightmostMatch then
15:                rightmostMatch = R[match]
16:            end if
17:        end for
18:
19:        if rightmostMatch > i − 1 then
20:            P[i] = rightmostMatch − i + 1
21:        else
22:            P[i] = 0
23:        end if
24:    end for
25:
26:    return P
27: end function
```

References

1. Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.
2. Juhani Karhumäki and Svetlana Puzynina. *On k-Abelian Palindromic Rich and Poor Words*, pages 191–202. Springer International Publishing, Cham, 2014.
3. Sandeep Subramanian, Srilakshmi Chaparala, Viji Avali, and Madhavi K Ganapathiraju. A pilot study on the prevalence of dna palindromes in breast cancer genomes. *BMC medical genomics*, 9(3):73, 2016.
4. Štěpán Holub and Kalle Saari. On highly palindromic words. *Discrete Applied Mathematics*, 157(5):953 – 959, 2009.