A Fast Algorithm for Generating Non-Isomorphic Chord Diagrams

Joe Sawada*

January 2, 2001

Abstract

Using a new string representation, we develop two algorithms for generating non-isomorphic chord diagrams. Experimental evidence indicates that the latter of the two algorithms runs in constant amortized time. In addition, we use simple counting techniques to derive a formula for the number of non-isomorphic chord diagrams.

1 Introduction

Chord diagrams are the fundamental combinatorial objects underlying Vassiliev invariants, which have application in knot theory [1]. A *chord diagram* is a set of $2n$ points on an oriented circle (counter-clockwise) joined pairwise by n chords. Figure 1 illustrates a chord diagram with 4 chords. Two chord digrams are isomorphic if one can be obtained by some rotation of the other. Special instances of chord diagrams are shown to have application in stamp foldings by Koehler [7]. A related object called a linearized chord diagram is studied by Stoimenow in [11] and braided chord diagrams are discussed by Birman and Trapp in [2].

Two fundamental questions when dealing with any combinatorial object are:

*DIMATIA, Charles University, Prague, CZECH REPUBLIC. Research supported by NSERC and partial support of Czech Grant GAČR 201/99/0242 and ITI under project LN-00A 056.

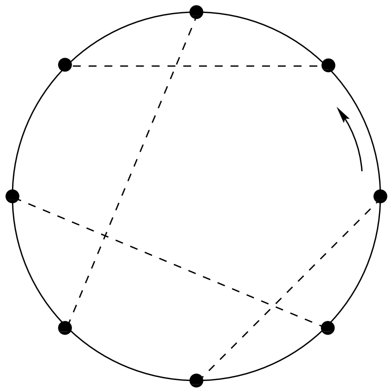

Figure 1: Chord diagram with 4 chords

1. How many instances of the object are there? (ie. How many non-isomorphic chord diagrams are there with n chords?)

2. How can we efficiently generate (list) all instances of the object? (ie. Can we develop a fast algorithm to generate all non-isomorphic chord diagrams with n chords.)

In answer to the first question, three independent papers by Li and Sun [8], Cori and Marcus [4], and Stoimenow [12], have derived enumeration formulas for the number of non-isomorphic chord diagrams. In each of these papers, the exact formula is the main result; however, in each case the derivation of the formula uses relatively complex methods. Cori and Marcus use Burnside's lemma (stated later in Section 3) along with liftings of quasi-diagrams; Li and Sun introduce a new object called a generalized m-configuration; Stoimenow uses Burnside's lemma along with two new objects: linerized chord diagrams and generalized linearized chord diagrams. As a secondary result in this paper, we derive an exact formula for the number of non-isomorphic chord diagrams with n chords using simple counting techniques.

The second question has not received as much attention as the first, or at least no significant results have been previously recorded. In response to this open problem, we develop two algorithms for generating non-isomorphic chord diagrams using a new string representation. A primary goal in any generation algorithm is for the amount of computation to be proportional to the number of objects generated. Such algorithms are said to be CAT for constant amortized time. The first algorithm we develop is very simple, but does not attain this time bound. The second algorithm requires more explanation, however experimental evidence gives a strong indication that it is CAT.

In the following section we give some basic number theory definitions, along with a background of a related object called a necklace. In Section 3 we derive an exact formula for enumerating non-isomorphic chord diagrams using simple techniques. In Section 4 we describe a new string representation for chord diagrams. Then, in Section 5, we outline a simple generation algorithm for non-isomorphic chord diagrams. In Section 6 we present another generation algorithm, with experimental results indicating that the algorithm is CAT. We conclude with a discussion of future work and open problems in Section 7.

2 Background

In the next section we derive an exact formula for the number of non-isomorphic chord diagrams with n chords. In the derivation, we encounter the following number theoretic functions.

The *Euler totient* function on an integer n, denoted $\phi(n)$, is the number of positive integers less than n that are relatively prime to n.

The bifactorial of an integer n, denoted $n!!$, is defined by the following:

$$n!! = \begin{cases} \displaystyle\prod_{j=0}^{\lfloor \frac{n-1}{2} \rfloor} n - 2j & \text{if } n > 0 \\ 1 & \text{if } n = 0 \text{ or } n = -1 \\ 0 & \text{if } n \leq -2 \ . \end{cases}$$

Using this notation, it is easy to see that the number of chord diagrams with n chords is $(2n-1)!!$.

2.1 Necklaces

An object closely related to a chord diagram is a necklace. A *necklace* is the lexicographically smallest element of an equivalence class of k-ary strings under rotation. For example, the set of all binary necklaces of length 4 is $\{0000, 0001, 0011, 0101, 0111, 1111\}$. We call an aperiodic necklace a *Lyndon*

```
procedure GenNecklaces ( t, p : integer );
local j : integer;
begin
    if t > n then
        if n mod p = 0 then PrintIt()
    else begin
        for j ∈ {a_{t-p}, ..., k-2, k-1} do begin
            a_t := j;
            if a_t = a_{t-p} then GenNecklaces( t+1, p );
            else GenNecklaces( t+1, t );
        end;
    end;
end;
```

Figure 2: The recursive necklace generation algorithm.

word and a string that is a prefix of a necklace a *pre-necklace*. We will reserve the term *periodic necklace* for a necklace that is not a Lyndon word.

Later, when we outline two algorithms for generating non-isomorphic chord diagrams, we follow the methods used in Ruskey's recursive necklace generation algorithm GenNecklaces(t, p) shown in Figure 2 [3]. This algorithm has been the basis for generating many other objects with rotational equivalence. In particular, it has been used to develop CAT algorithms to generate bracelets [10], fixed density necklaces [9] and unlabeled necklaces [3]. The general idea of this backtracking algorithm is to generate a length t pre-necklace, stored in the array a, and then for each valid character, append it to the end of the pre-necklace to get a length $t+1$ pre-necklace. The parameter p maintains the length of the longest Lyndon prefix of the string. When the pre-necklace is of length n, a simple test determines whether or not it is a necklace. This algorithm can also generate Lyndon words by changing the condition from $n \bmod p = 0$ to $n = p$. The initial call is GenNecklaces(1,1) and a_0 is initially set to 0. A more detailed explanation and a proof showing the algorithm is CAT is found in [3].

3 Enumerating Non-Isomorphic Chord Diagrams

One of the most useful tools for enumerating combinatorial objects with equivalence under some group action is Burnside's Lemma.

BURNSIDE'S LEMMA *If a group G acts on a set S and $Fix(g) = \{s \in S | g(s) = s\}$, then the number of equivalence classes is given by*

$$\frac{1}{|G|} \sum_{g \in G} |Fix(g)|.$$

The set of all chord diagrams with n chords is partitioned into equivalence classes by the cyclic group $\mathbb{C}_{2n}$. Two chord diagrams are isomorphic if one can be obtained by some rotation of the other. If we let σ denote a single rotation, then the group elements of $\mathbb{C}_{2n}$ are σ^j for $j = 1, 2, \ldots, 2n$. To count the number of non-isomorphic chord diagrams with n chords, which we denote $C(n)$, we apply Burnside's lemma:

$$C(n) = \frac{1}{2n} \sum_{j=1}^{2n} Fix(\sigma^j).$$

The number of chord diagrams fixed by σ^j depends only on the order of σ^j. In other words, if two group elements σ^j and σ^k have the same order, then the set of chord diagrams fixed by each group element will be the same. The number of elements of $\mathbb{C}_{2n}$ with order p (where $p|2n$) is $\phi(p)$. Thus, if we let $T(2n, p)$ denote the number of chord diagrams with n chords fixed by a group element of order p (namely σ^q) then

$$C(n) = \frac{1}{2n} \sum_{pq=2n} \phi(p) T(2n, p).$$

We now derive a formula for $T(2n, p)$ by deriving recurrence equations for two cases: p odd and p even. We start by labeling the endpoints on a chord diagram from 1 to $2n$ in counter-clockwise order around the circle. With this labeling, we define the *length* of a chord starting from i and ending at j to be $j - i \bmod 2n$. We now consider the chords starting at $q, 2q, \ldots pq$, where $pq = 2n$. If a chord diagram is fixed by σ^q, then the length of the chords

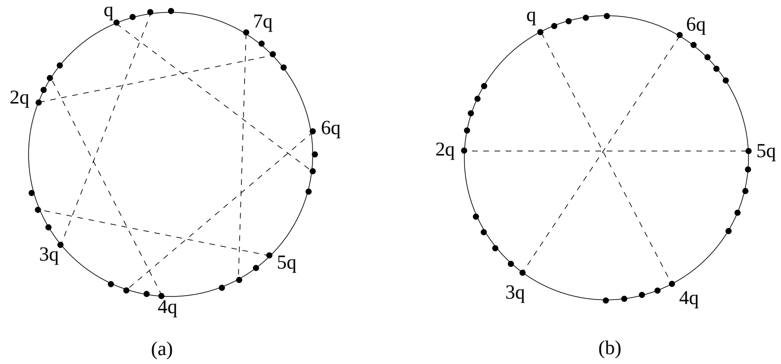

Figure 3: (a) One of the $2n-p$ possible lengths for the chords starting at $q, 2q, \ldots, pq$. (b) For p even, there is only one choice for the endpoint landing back in the list $q, 2q, \ldots, pq$.

starting at these positions must be the same. If p is odd then there are $2n-p$ possible lengths for the chords, since it is impossible for two endpoints in the list $q, 2q, \ldots pq$ to be joined together (see Figure 3(a)). If we now ignore these chords, we are reduced to the problem of counting $T(2n-2p, p)$. Thus, if p is odd:

$$T(2n, p) = (2n-p)T(2n-2p, p).$$

In the base case, $T(0, p) = 1$. If p is even, then there is the additional possibility that the chords have length n (see Figure 3(b)). Therefore if p is even:

$$T(2n, p) = (2n-p)T(2n-2p, p) + T(2n-p, p).$$

In the base cases, $T(p, p) = T(0, p) = 1$.

Solving the two recurrence equations yields the following exact formula:

$$T(2n, p) = \begin{cases} (q-1)!!p^{\frac{q}{2}} & \text{if } p \text{ odd} \\ \displaystyle\sum_{j=0}^{\lfloor \frac{q}{2} \rfloor} \binom{q}{2j} \cdot (2j-1)!! \cdot p^j & \text{if } p \text{ even.} \end{cases}$$

The solution for odd p is easily obtained by substituting into the recurrence. Proof by induction on q will verify the solution when p is even.

4 Representing Chord Diagrams

There are numerous ways to represent chord diagrams. Several objects equivalent to our definition of chord diagrams have been studied by other authors, including polygons where the sides are identified pairwise [13], and one-vertex maps [6]. In this section we develop a new string representation.

Before we describe this new string representation for chord diagrams, we outline a very natural one. First, assign each chord a unique value from 1 to n, and then label the endpoints with the value of their incident chord. If we arbitrarily pick a starting point s, then we obtain a string representation by recording the endpoint values starting at s and moving counter-clockwise (by convention) around the circle. In this manner, any string with length $2n$ containing exactly two occurrences of the values 1 through n can be used to represent a chord diagram. An example of this string representation is shown in Figure 4(a). Such string representations have equivalence under string rotation and permutation of the alphabet symbols 1 through n. Thus, there may be up to $2n(n!)$ strings in each equivalence class. The lexicographically smallest strings in each equivalence class are more commonly known as *unlabeled necklaces* (where the number of each alphabet symbol is 2). Currently, there exists an efficient algorithm for generating binary unlabeled necklaces [3]; however no efficient algorithm exists for strings on an arbitrarily sized alphabet. There also exists an efficient algorithm to generate necklaces where the number of 0's is fixed [9]; but there is currently no efficient algorithm to generate necklaces if the number of each alphabet symbol is fixed.

Because no efficient generation algorithm currently exists using this natural string representation, we consider a new approach. This time we label each endpoint with its associated length (see Section 3). Note that the lengths are independent of the the starting point s; however, there is a dependency between each pair of endpoints joined by a chord - their values must sum to $2n$. If we again traverse counter-clockwise around the circle starting at s, recording the endpoint values, we obtain a new string representation. In this new string representation, we no longer have equivalence under permutation of the alphabet symbols, and the number of each alphabet symbol is no longer fixed; however, the size of the alphabet has increased from n to $2n-1$. An example of this string representation is given in Figure 4(b).

In each of the following two sections, we present an algorithm for generating non-isomorphic chord diagrams. Both algorithms use the new string representation outlined in this section; however, each algorithm defines a

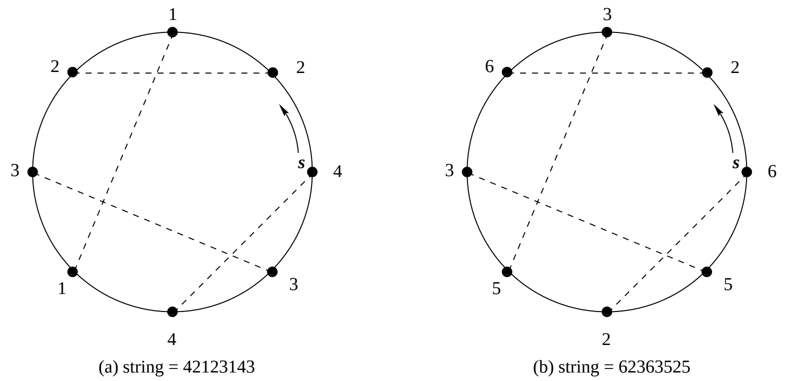

Figure 4: Two string representations: (a) label chords then endpoints (b) label endpoints by chord length.

different representative for each equivalence class.

5 A Simple Algorithm

In this section we develop a simple algorithm to list all non-isomorphic chord diagrams with n chords. To represent the chord diagrams, we use the new string representation described in the previous section. Using the lexicographically smallest string as the representative of each equivalence class, we arrive at a problem equivalent to generating length $2n$ necklaces on an alphabet of size $2n-1$, with the added restriction that each necklace corresponds to a valid chord diagram.

Recall that when generating necklaces, we build up a pre-necklace one character at a time. Applying this to chord diagrams, we instead add one chord or two characters at a time. Thus, if we are adding the value j to the tth position of the string, then we must also add the value $2n-j$ to the $t+j$th position. Of course, we must observe the condition that $t+j \leq 2n$. In addition, we must make sure that we do not overwrite values already assigned to positions t and $t+j$ in the pre-necklace. If we have already assigned a value to the tth position (ie. if $a_t \neq 0$), then we continue generation with position $t+1$ only if the string $a_1 \ldots a_t$ is a valid pre-necklace (ie. if $a_t \geq a_{t-p}$). By adding these simple modifications to $\mathsf{GenNecklaces}(t,p)$, we ensure that each necklace generated corresponds to a valid chord diagram.

```
procedure SimpleChords ( t, p : integer );
local j : integer;
begin
    if t > 2n then
        if 2n mod p = 0 then PrintIt()
    else begin
        if a_t = 0 and t + a_{t-p} ≤ 2n then begin
            for j ∈ {a_{t-p}, ..., 2n - t} do begin
                if a_{t+j} = 0 then begin
                    a_t := j;    a_{t+j} := 2n - j;
                    if a_t = a_{t-p} then SimpleChords( t + 1, p );
                    else SimpleChords( t + 1, t );
                    a_{t+j} := 0;
                end;
            end;
            a_t := 0;
        end;
        else if a_t = a_{t-p} then SimpleChords( t + 1, p );
        else if a_t > a_{t-p} then SimpleChords( t + 1, t );
    end;
end;
```

Figure 5: A simple algorithm for generating non-isomorphic chord diagrams with n chords.

The resulting algorithm for generating non-isomorphic chord diagrams in lexicographic order, $\mathsf{SimpleChords}(t, p)$, is shown in Figure 5. The initial call is $\mathsf{SimpleChords}(1, 1)$ and a_0 is initially set to 1. Aperiodic chord diagrams can be generated by replacing the test $2n \bmod p = 0$ with $2n = p$, as was the case with necklaces.

Recall that our goal is to develop a generation algorithm which runs in constant amortized time. The goal does not look promising with this algorithm since the depth of the computation tree is $2n$ when we only require the assignment of n chords per diagram. To verify this conjecture we gather some experimental evidence. To calculate the amount of computation we sum the number of recursive calls plus the number of iterations of the for loop that did not produce a recursive call. The resulting ratio of this computation

Number of chords n	Non-isomorphic chord diagrams	Ratio of work done to chord diagrams generated
1	1	1.0
2	2	3.0
3	5	8.0
4	18	11.8
5	105	14.3
6	902	15.7
7	9749	16.9
8	127072	17.9
9	1915951	18.8
10	32743182	19.8
11	624999093	20.7

Table 1: Experimental results for $\mathsf{SimpleChords}(t, p)$

compared to the number of chord diagrams generated is given in Table 1 for $n \leq 11$. Notice that the ratios are steadily increasing as the number of chords increases. This is a strong indication that the algorithm is *not* CAT. For this reason, we attempt no mathematical analysis and focus on developing a faster algorithm.

6 A Fast Algorithm

In this section we develop an experimentally CAT algorithm for generating non-isomorphic chord diagrams. In this algorithm we use the same string representation for chord diagrams as in the previous algorithm, but this time we use a different representative for each equivalence class.

Let $\alpha = a_0 a_2 \cdots a_{2n-1}$ represent a chord diagram with n chords. Let pos_i be the increasing sequence composed of the positions (indexes) for all occurrences of the value i in α. Now consider the string $\beta = pos_1 pos_2 pos_3 \cdots pos_{2n-1}$. Using this construction, each string α yields a unique string β. We define the canonical form, or representative, of each equivalence class to be the string α with the lexicographically smallest string β. For example, in Table 2 we show the equivalence class of strings representing the chord diagram in Figure 4(b) along with their corresponding β strings.

α	β
62363525	16245703
23635256	05134627
36352562	47023516
63525623	36172405
35256236	25061347
52562363	14570236
25623635	*03461725*
56236352	27350614

Table 2: The canonical form for this equivalence class is 25623635.

Before we develop a generation algorithm using these representatives, we first outline a linear time verification algorithm for determining whether or not the string α (representing a chord diagram) is in canonical form.

6.1 A Verification Algorithm

A naïve method for determining if a chord diagram α is in canonical form is to compare its β string with the β string of all other strings in its equivalence class. Such an algorithm would take worst case time $O(n^2)$. We present an algorithm that runs in linear time.

By the definition of the canonical form, we see that the positions of the minimum value in the string $\alpha = a_0a_1 \cdots a_{2n-1}$ are the most critical. If v^* is the minimum value, then we consider the string $pos_{v^*} = p_1p_2 \cdots p_t$, where there are t occurrences of the value v^* in α. In order for α to be in canonical form then p_1 must equal 0 or equivalently $a_0 = v^*$. If p_1 had any other value, then there would exist a rotation of α such that $p_1 = 0$. This would yield a smaller pos_{v^*} string, and thus a smaller β string. Now consider the modified string $pos'_{v^*} = q_1q_2 \cdots q_t$, where $q_i = p_{i+1} - p_i$ for $i = 1, 2, \ldots, t-1$ and $q_t = 2n - p_t$. If the string pos'_{v^*} is a necklace, then it is easy to verify that the original string pos_{v^*} will be the lexicographically smallest string when compared to the corresponding pos_{v^*} strings from other strings in α's equivalence class. Furthermore, if pos'_{v^*} is a Lyndon word, then α will be the unique string in its equivalence class to yield the string pos_{v^*}, and thus it is in canonical form. If pos'_{v^*} is not a necklace, then we can find a rotation of the string α such that a smaller string pos_{v^*} can be obtained, implying that

α is *not* in canonical form. As an example to the above strategy, consider the string $\alpha = 363959463789$. Since the minimum value is 3, we consider $pos_3 = 028$ and $pos'_3 = 264$. Because pos'_3 is a Lyndon word, α is in canonical form.

Using this strategy, we can determine whether or not a string α is in canonical form unless the string pos'_{v^*} is a periodic necklace. If pos'_{v^*} has length t and period p, and assigning $p' = 2n(\frac{p}{t})$, then the rotations of the string α starting at positions $p', 2p', \ldots, 2n - p'$ will all yield the same string pos_{v^*}. In this case, we must continue examining α's corresponding β string. We update the value v to the next smallest value found in α, and focus on the new string pos_v. Observe that we can no longer employ the same strategy as before, since the starting points for the other rotations of α that may be the canonical form have been restricted. Of these remaining strings, for α to be in canonical form, it must have the lexicographically smallest string pos_v. To determine this efficiently, we modify the string pos_v in the following manner. First, the values $p', 2p', \ldots, 2n - p', 2n$ are inserted into pos_v so the string is still in sorted order. Then each value j is replaced with $j \bmod p'$. Finally, we replace all 0's, which were originally the values $p', 2p', \ldots, 2n$ with p'. We denote this modified string by pos'_v. Notice that such a construction implies that the string pos'_v for $\sigma^j(\alpha)$ where j in $p', 2p', \ldots, 2n - p'$ is a rotation of the string pos'_v for α. Thus, as before, if the resulting string pos'_v is a Lyndon word, then α is in canonical form. If pos'_v is a periodic necklace with period p and length t , then we repeat this procedure with the next largest v, updating p' to $2n(\frac{p}{t})$. If pos'_v is not a necklace, then α is not in canonical form. Due to the dependencies on the string α, if v ever exceeds n, then the chord diagram is in canonical form and has period equal to the last updated value for p'.

To get a better understanding of this verification algorithm, we go through two examples.

Example 1

Consider a chord diagram represented by $\alpha = 1925819258$. We want to determine if α is in canonical form. First we consider $pos_1 = 05$ and $pos'_1 = 55$. Since pos'_1 is a periodic necklace we must consider $pos_2 = 27$, with $p' = 5$. To modify pos_2, we insert the value 5 and 10 to get the string 2 5 7 10. Next, we replace each value j with $j \bmod 5$ to get 2020. Finally, we replace the 0's with 5 to get the new string $pos'_2 = 2525$. Since this is a periodic necklace, we must repeat this procedure for the string pos_5 updating $p' = 5$. We now consider $pos_5 = 38$, and perform the modifications to get $pos'_5 = 3535$. Again we have a periodic necklace, and update $p' = 5$. Since the next value exceeds n, we conclude that α is in canonical form (with period 5). $\square$

EXAMPLE 2

Consider the string $\alpha = 3\ 6\ 10\ 13\ 11\ 4\ 7\ 10\ 3\ 12\ 4\ 13\ 6\ 9\ 12\ 5$ representing a chord diagram with 8 chords. To determine if it is in canonical form we first consider $pos_3 = 08$ with $pos'_3 = 88$. Since the latter string is a periodic necklace we must consider $pos_4 = 5\ 10$ with $pos'_4 = 5828$. Now since 5828 is not a necklace, the string α is *not* in canonical form. $\square$

In the worst case, this verification algorithm must analyze each string pos'_v for $v = 1, 2, \ldots n$. Using Duval's algorithm for factoring a string into Lyndon words [5], we can determine if pos'_v is a necklace or a Lyndon word in linear time. Therefore, an upper bound for the running time of the algorithm is proportional to $\sum_{v=1}^{n} |pos'_v|$. Observe that length of each string pos'_v is at most $|pos_v| + |pos_{v-1}|$. Thus, since $\sum_{v=1}^{n} |pos_v| < 2n$, the verification algorithm runs in time $O(n)$.

6.2 The Generation Algorithm

In this subsection we describe a fast algorithm for generating chord diagrams. The method behind the generation algorithm follows directly from the verification algorithm described in the previous subsection.

Following the verification algorithm, the placement of the minimum value v^* is the most important. Specifically, the value v^* must occur in the position a_0 and the string pos'_{v^*} must be a necklace. Thus, the first step in the generation algorithm is to generate all strings pos_{v^*} (the placing of the values v^* in α) so that the corresponding string pos'_{v^*} is a necklace. For each string pos'_{v^*} that is a Lyndon word, it doesn't matter how the rest of the string α is filled as long as each position has value at least $v^* + 1$. In addition, we must observe the dependencies of the string representation which imply that whenever the value v is added to position s, the value $2n - v$ must be added to position $(s + v) \bmod 2n$. If the string pos'_{v^*} is a periodic necklace, then we repeat the process by attempting to place the next largest value v in such a way that pos'_v is a necklace. The result of this approach is the generation of all strings α which represent unique chord diagrams.

This algorithm is naturally divided into three separate recursive routines: the first routine $\mathsf{Gen}(t, p, s, v^*, last, B)$ generates the necklaces pos'_{v^*}; the second routine $\mathsf{Gen2}(t, p, s, v, p', part)$ generates the necklaces pos'_v for all $v > v^*$; and the third routine $\mathsf{GenRest}(s, e, v)$ fills the remaining positions with values that are at least v. The routine $\mathsf{FastChords}()$ drives these routines to generate all non-isomorphic chord diagrams with n chords.

Within the algorithm a global linked list is used to keep track of the available positions of α, in increasing order. The variable $head$ is the value of the first available position and the value $2n$ represents the end of the list. If s is an available position in the list then $s.next$ will give the value of the next available position in the list. If the list is implemented using an array with next and previous pointers then the functions $\mathsf{Add}(s)$, $\mathsf{Remove}(s)$, and $\mathsf{Avail}(s)$ can be implemented in constant time. The routine $\mathsf{InitList}()$ initializes the list to contain every position from 0 to $2n - 1$. The function $\mathsf{Print}()$ prints out the contents of the string α.

The various details of the functions $\mathsf{FastChords}()$, $\mathsf{Gen}(t, p, s, v^*, last, B)$, $\mathsf{Gen2}(t, p, s, v, p', part)$ and $\mathsf{GenRest}(s, e, v)$ are described in the following subsections. Many of the details correspond directly with comments made in the verification algorithm.

6.2.1 FastChords()

The routine $\mathsf{FastChords}()$ drives the algorithm by calling $\mathsf{Gen}(1, 1, head, v^*, 0, \text{TRUE})$ for each value v^* ranging from 1 to $n-1$. Before making the call, it makes the first assignment of the value v^* to the position a_0 as well as the assignment of the value $2n - v^*$ to the position a_{v^*}. The only string with minimum value n is $\alpha = n^{2n}$. This string is listed separately at the end of this function. Pseudocode for $\mathsf{FastChords}()$ is shown in Figure 6.

6.2.2 Gen($t, p, s, v^*, last, B$)

This function generates all necklaces pos'_{v^*} by recursively going through each available position s in α and attempting to place the value v^*. The function maintains the following parameters (the first two are from the necklace generation algorithm):

- t: maintains the length of the pre-necklace pos'_{v^*}
- p: maintains the length of the longest Lyndon prefix of pos'_{v^*}

```
procedure FastChords ();
local i, v* : integer;
begin
    InitList();
    for v* ∈ {1, 2, ..., n − 1} do pos_{v*,0} := 0;
    for v* ∈ {1, 2, ..., n − 1} do begin
        a_0 := v*;    a_{v*} := 2n − v*;
        Remove(0); Remove(v*);
        Gen(1, 1, head, v*, 0,TRUE);
        Add(v*); Add(0);
    end;
    for i ∈ {0, 1, 2, ..., 2n − 1} do a_i := n;
    Print();
end;
```

Figure 6: FastChords()

- s: the position of α to be filled
- v^*: the value to be placed into position s
- $last$: the position of the last inserted value v^* in α
- B: boolean value indicating if it the first time the pre-necklace pos'_{v^*} has been encountered

At each call to Gen($t, p, s, v^*, last, B$), the string $pos'_{v^*} = q_1 q_2 \cdots q_{t-1}$ is a pre-necklace. To extend this string to a length t pre-necklace, the next value q_t must be at least q_{t-p}. If we set the value $min = last - q_{t-p}$ then if $s \geq min$, then the new value $s - last$ can be appended to the pre-necklace of length $t-1$ (as long as the associated position $(s+v^*) \bmod 2n$ is available) to obtain a new pre-necklace of length t.

Before we attempt to extend the pre-necklace pos'_{v^*} we must consider its status with the value $2n - last$ appended to the end. If $min < 2n$, then the string with the appended value is a Lyndon word and for each Lyndon word we call GenRest($head, head.next, v^* + 1$) to fill the remainder of the string α . If $min = 2n$ and $t \bmod p = 0$, then the modified string is a periodic necklace and for each periodic necklace we attempt to place the value $v^* + 1$

by calling $\mathsf{Gen2}(1, 1, head, v^* + 1, 2n\frac{p}{t}, 0)$, unless α has been completely filled ($t = n$), in which case we simply print the string. For these tests we must be careful not to consider the same pre-necklace pos'_{v^*} twice. The boolean value B indicates whether or not the pre-necklace has been encountered before. If B is TRUE, then it is the first time the pre-necklace has been encountered.

Once we have checked if pos'_{v^*} with the appended value $2n - last$ is a necklace, we proceed by attempting to add the value v^* to the position s. If we can, then we remove the values v^* and $2n - v^*$ from the avail list, make the appropriate assignments to α and pos'_{v^*}, and make a recursive call with appropriate updates to the parameters. Finally, regardless of whether or not a value has been placed, we make a recursive call for the next available position $s.next$, but here we must set the boolean value B to FALSE, since the same pre-necklace is used in the resulting recursive call.

This function assumes that the first position in the string α has been assigned the value v^*. Pseudocode for $\mathsf{Gen}(t, p, s, v^*, last, B)$ is shown in Figure 7.

6.2.3 Gen2$(t, p, s, v, p', part)$

This function generates all necklaces pos'_v for values $v > v^*$, given the remaining available positions in the string α. It maintains the following parameters:

- t: maintains the length of the pre-necklace pos'_v
- p: maintains the length of the longest Lyndon prefix of pos'_v
- s: the position of α to be filled
- v: the value to be placed into position s
- p': the value as described in the verification algorithm
- $part$: maintains the number of times p' has been inserted

According to the verification algorithm, we must make two modifications to the string pos_v. First we convert all positions s in the string to $s \bmod p'$. Second we must insert the values p' at particular locations in the string. Thus, if we generate the pre-necklaces pos_v by converting each position s to $s \bmod p'$ and adding the values p' where necessary, we are in fact generating the pre-necklaces pos'_v.

```
procedure Gen ( t, p, s, v*, last: integer; B: boolean );
local s',e,min : integer;
begin
    min := last + pos'_{v*,t-p};
    if min < 2n and B = TRUE then GenRest(head, head.next, v* + 1);
    if min = 2n and t mod p = 0 and B = TRUE then begin
        if t = n then Print();
        else Gen2(1, 1, head, v* + 1, 2n p/t, 0);
    end;
    if min < 2n and s < 2n then begin
        e := (s + v*) mod 2n;
        if s ≥ min and Avail(e) then begin
            s' := s.next;
            if s' = e then s' := e.next;
            a_s := v*;   a_e := 2n − v*;
            Remove(s);   Remove(e);
            pos'_{v*,t} := s − last;
            if s = min then Gen(t + 1, p, s', v*, s,TRUE);
            else Gen(t + 1, t, s', v*, s,TRUE);
            Add(e);   Add(s);
        end;
        Gen(t, p, s.next, v*, last,FALSE);
end; end;
```

Figure 7: Gen$(t, p, s, v^*, last, B)$

The extension of the pre-necklaces $pos'_v = q1q2 \cdots q_{t-1}$ is similar to the previous function, but in this case the value min is simply q_{t-p} and the value we wish to add is $s \bmod p'$. Thus if $s \bmod p' \geq min$ and the associated position $(s+v) \bmod 2n$ is available, then we can extend the pre-necklace pos'_v in a similar fashion to $\mathsf{Gen}(t, p, s, v^*, last, B)$.

Once we have considered all available positions s in α, the parameter s will equal $2n$. If $head = 2n$, then α is full and by construction it is in canonical form. In this case the string α is printed. Otherwise, we analyze the string pos'_v to see if it is a Lyndon word or a periodic necklace. If min is strictly less than p' then the string is a Lyndon word and $\mathsf{GenRest}(head, head.next, v+1)$ is called to fill the remaining available positions in α. If $t \bmod p = 0$ then the string is a periodic necklace. In this case we call $\mathsf{Gen2}(1, 1, head, v+1, 2n\frac{p}{t}, 0)$ to generate the necklaces pos'_{v+1}. Before we make the initial test of $s = 2n$ however, we must consider three special cases.

Case 1. When $v = n$ we do not want to place the value v in any position s greater than n. This is because the resulting string is equivalent to placing the value in the position $n - s$. Thus as soon as we reach such a state we terminate generation from this node, unless α is full ($head = 2n$) in which case we print the string.

Case 2. We must consider the case when the value v is not placed in the string α. This state occurs when $t = 1$ and $s > p'$. In this case we continue with the placement of $v+1$ by calling $\mathsf{Gen2}(1, 1, head, v+1, p', 0)$. Before making this call, we must make sure that v is less than n. Otherwise we will end up trying to place a value greater than n which will result in the value $2n - v$, which will be less than n, being added to α.

Case 3. The final case to consider is the placement of the values p' in the string pos'_v. These values are placed the first time the position s exceeds the value $p'(part+1)$. Once the value is added, then the generation is continued by incrementing the parameter $part$ by 1, and updating the values t and p as usual.

Pseudocode for $\mathsf{Gen2}(t, p, s, v, p', part)$ is shown in Figure 8.

6.2.4 GenRest(s, e, v)

The routine $\mathsf{GenRest}(s, e, v)$ is a simple recursive procedure that fills the remaining available positions in α with values greater than or equal to v. It takes the following parameters as input:

- s: the first available position in α

```
procedure Gen2 ( t, p, s, v, p', part: integer );
local s',e,min: integer;
begin
    min := pos'_{v,t-p};
    if v = n and s > n then begin
        if head = 2n then Print();
    end;
    else if t = 1 and s > p' then begin
        if v < n then Gen2(1, 1, head, v + 1, p', 0);
    end;
    else if s > p'(part + 1) then begin
        pos'_{v,t} := p';
        if min = p' then Gen2(t + 1, p, s, v, p', part + 1);
        else Gen2(t + 1, t, s, v, p', part + 1);
    end;
    else if s = 2n then begin
        if head = 2n then Print();
        else if min < p' then GenRest(head, head.next, v + 1);
        else if t mod p = 0 then Gen2(1, 1, head, v + 1, 2n p/t, 0);
    end;
    else begin
        e := (s + v) mod 2n;
        if s mod p' ≥ min and Avail(e) then begin
            s' := s.next;
            if s' = e then s' := e.next;
            a_s := v;    a_e := 2n − v;
            Remove(s);    Remove(e);
            pos'_{v,t} := s mod p';
            if s mod p' = min then Gen2(t + 1, p, s', v, p', part);
            else Gen2(t + 1, t, s', v, p', part);
            Add(e);    Add(s);
        end;
        Gen2(t, p, s.next, v, p', part);
end; end;
```

Figure 8: Gen2$(t, p, s, v, p', part)$

```
procedure GenRest ( s, e, v : integer );
begin
    if s = 2n then Print();
    else if e ≠ 2n then begin
        if e − s ≥ v and 2n − e + s ≥ v then begin
            a_s := e − s;   a_e := 2n − a_s;
            Remove(s);  Remove(e);
            GenRest(head,head.next,v);
            Add(e);   Add(s);
        end;
        if 2n − e.next + s ≥ v then GenRest(s,e.next,v);
end; end;
```

Figure 9: GenRest(s, e, v)

- e: another available position in α
- v: the minimum value to be placed

The idea is to attempt to place a chord joining positions s and e. Such an assignment is valid as long as the values $e - s$ and $2n - e + s$ are both greater than or equal to v. If the assignment is valid then a recursive call is made with the next two available positions. Regardless if the assignment is valid, we make a recursive call to check the next possible position for e which is $e.next$. If $2n - e.next + s < v$ then clearly no empty positions past e will provide valid assignments. Once all the positions are filled ($s = 2n$) then the string is printed. Pseudocode for GenRest(s, e, v) is shown in Figure 9.

6.2.5 Analysis

As with the previous algorithm, we obtain experimental results for the amount of work done compared to the number of chord diagrams generated. Since the work done for each recursive call is constant, we count the amount of work done by summing the number of recursive calls. The resulting ratios are shown in Table 3 for $n \leq 12$. Notice that the ratios are decreasing (after $n = 5$) as the number of chords increases. This gives a very strong indication that the algorithm runs in constant amortized time.

Number of chords n	**Non-isomorphic chord diagrams**	**Ratio of work done to chord diagrams generated**
1	1	1.0
2	2	3.0
3	5	9.2
4	18	13.6
5	105	14.2
6	902	12.4
7	9749	11.0
8	127072	10.0
9	1915951	9.4
10	32743182	8.9
11	624999093	8.5
12	13176573910	8.2

Table 3: Experimental results for FastChords()

CONJECTURE 1 *The algorithm for generating non-isomorphic chord diagrams, FastChords(), is CAT.*

A mathematical proof for the conjecture is not given in this paper, leaving a challenging open problem.

7 Future Work

In this paper we have outlined a fast algorithm for generating non-isomorphic chord diagrams. However, we have not found a mathematical proof to show that the algorithm is CAT. We have also mentioned two other open problems in this paper:

- The development of an efficient algorithm to generate k-ary unlabeled necklaces.
- The development of an efficient algorithm to generate k-ary necklaces where the number of occurrences of each alphabet symbol is fixed.

The canonical form used in the algorithm FastChords() has recently been used to develop a CAT algorithm for the latter problem if the number of occurences of some value v is relatively prime to n [10].

References

[1] D. Bar-Natan, On the Vassiliev Knot Invariants, Topology 34 (1995) 423-472.

[2] J. Birman and R. Trapp, Braided chord diagrams, Journal of Knot Theory and its Ramifications, Vol. 7 No. 1 (1998) 1-22.

[3] K. Cattell, F. Ruskey, J. Sawada, C.R. Miers, M. Serra, Fast algorithms to generate necklaces, unlabeled necklaces, and irreducible polynomials over GF(2), to appear in Journal of Algorithms.

[4] R. Cori and M. Marcus, Counting non-isomorphic chord diagrams, Theoretical Computer Science, 204 (1998) 55-73.

[5] J-P. Duval, Factoring words over an ordered alphabet, Journal of Algorithms, 4 (1983), 363-381.

[6] J. Harer and D. Zagier, The Euler characteristic of the moduli space of curves, Inventiones Mathematicae, 85 (1986) 457-485.

[7] J.E. Koehler, S.J., Folding a strip of stamps, Journal of Combinatorial Theory, 5 (1968), 135-152.

[8] B. Li and H. Sun, Exact number of chord diagrams and an estimation of the number of spine diagrams of order n, Chinese Science Bulletin, Vol. 42 No. 9 (1997), 705-720.

[9] F. Ruskey, J. Sawada, An efficient algorithm for generating necklaces of fixed density, SIAM Journal on Computing, 29 (1999) 671-684.

[10] J. Sawada, *Fast Algorithms to Generate Restricted Classes of Strings Under Rotation*, Dissertation, University of Victoria (2000).

[11] A. Stoimenow, Enumeration of chord diagrams and an upper bound for Vassiliev invariants, Journal of Knot Theory and its Ramifications, Vol. 7, No. 1 (1998), 93-114.

[12] A. Stoimenow, On the number of chord diagrams, manuscript, www.informatik.hu-berlin.de/ stoimenow.

[13] T. Walsh and A. Lehman, Counting rooted maps by genus I,II, Journal of Combinatorial Theory (B), 13 (1972), 192-218, 122-141.