An Eades-McKay Algorithm for Well-Formed Parentheses Strings

Bette Bultena and Frank Ruskey

Department of Computer Science

University of Victoria

Victoria, B.C. V8W 3P6, Canada

August 10, 1998

Abstract

Let $\mathbf{T}(n)$ be the set of all well-formed parentheses strings of length $2n$. We show that the elements of $\mathbf{T}(n)$ can be listed so that successive strings differ by the transposition of a left and a right parenthesis. Furthermore, between the two parentheses that are transposed, only left parentheses occur.

Our listing is a modification of the well-known Eades-McKay [4] algorithm for generating combinations. Like that algorithm, ours generates strings from the lexicographically greatest string to the lexicographically least and can be implemented so that each string is generated in constant time, in an amortized sense.

1 Introduction

Among the classes of strings studied by mathematicians and computer scientists, perhaps none has been examined so intensely as the class of well-formed parentheses strings. There is a natural correspondence, right parentheses to internal nodes and left parentheses to leaves, between these strings and extended binary trees. As a consequence, one representation is often used to generate the other.

One natural investigation has been that of generating (i.e., listing) all parenthesis strings in some specified order; e.g., lexicographic order. Algorithms for listing parenthesis strings in lexicographic order may be found in Zaks [10] and Mäkinen [6]. Gray code algorithms for generating parentheses strings may be found in Ruskey and Proskurowski [9], and Lucas, Roelants van Baronaigien, and Ruskey [5]. A parallel algorithm for generating well-formed parentheses is given by Akl and Stojmenovic [1].

Listings of well-formed parentheses strings often use bitstring representations. Following Ruskey [8], we say that a transposition of a 0 and a 1 in a bitstring is *homogeneous* if only 0's occur between the 0 and 1 that are transposed. If the positions of the 1's are recorded in increasing order then only a single position has changed in the case of a homogeneous transposition, but not in the case of a non-homogeneous transposition. Define $\mathbf{B}(n,k)$ to be the set of all bitstrings of length n that contain exactly k 1's. Algorithms for generating the elements of $\mathbf{B}(n,k)$ by homogeneous transpositions have been developed by Eades and McKay [4], Chase [2], and Ruskey [7].

The objects being listed are strings over the binary alphabet $\{0,1\}$. If L is a list of strings and x is a symbol, then $L\cdot x$ denotes the list of strings obtained by appending an x to each string of L. For example if $L = \langle 01, 10\rangle$, then $L\cdot 1 = \langle 011, 101\rangle$. The notation $x\cdot L$ is defined similarly, except that x is prepended to each element of L. By x^R we denote the reversal of x and L^R denotes the list L, but with each string in the list reversed. For example, if $L = \langle 0111, 0011\rangle$, then $L^R = \langle 1110, 1100\rangle$. If L and L' are lists then $L \circ L'$ denotes the concatenation of the two lists. For example, if $L = \langle 01, 10\rangle$ and $L' = \langle 11, 00\rangle$, then $L \circ L' = \langle 01, 10, 11, 00\rangle$.

For a list L, let $first(L)$ denote the first element on the list and let $last(L)$ denote the last element on the list L. If L is a list $\langle l_1, l_2, \ldots, l_n\rangle$, then $\overline{L}$ denotes the list obtained by listing the elements of L in reverse order; i.e., $\overline{L} = \langle l_n, \ldots, l_2, l_1\rangle$. Note the obvious equations $first(\overline{L}) = last(L)$ and $last(\overline{L}) = first(L)$.

The following form of the Eades-McKay recurrence relation is simpler than that found in [4] and is the basis of our recurrence for parentheses strings [7]:

$$\mathbf{E}(n,k) = \begin{cases} 0^n & \text{if } k = 0 \\ 10^{n-1} \circ 010^{n-2} \circ \cdots \circ 0^{n-1}1 & \text{if } k = 1 \\ \mathbf{E}(n-1,k)\cdot 0 \circ \overline{\mathbf{E}(n-2,k-1)}\cdot 01 \circ \mathbf{E}(n-2,k-2)\cdot 11 & \text{if } 1 < k < n \\ 1^n & \text{if } k = n. \end{cases}$$

The first element in the list, $first(\mathbf{E}(n,k))$, is lexicographically largest, 1^k0^{n-k}. The last element, $last(\mathbf{E}(n,k))$, is lexicographically smallest, $0^{n-k}1^k$. The recursive algorithm derived from this definition produces a homogeneous listing for combinations and this is proven by induction in [3]. Note that the Gray code sequence produced is not cyclic if $1 < k \leq n$; that is, $first(\mathbf{E}(n,k))$ and $last(\mathbf{E}(n,k))$ do not differ by a homogeneous transposition.

2 Well-Formed Parentheses

The set of all well-formed parentheses strings of length $2n$ is denoted by $\mathbf{T}(n)$. The subset of $\mathbf{T}(n)$ consisting of all well-formed parentheses strings with exactly k leading left parentheses is denoted by $\mathbf{T}(n,k)$. For notational convenience we regard the elements of these sets as bitstrings, where the left parentheses are represented by 1's and the right parentheses by 0's. Using this representation, $\mathbf{T}(n,k)$ was generated by non-homogeneous transpositions in [9].

Our goal is to find a homogeneous Gray code listing of $\mathbf{T}(n)$. It is sufficient to generate $\mathbf{T}(n,k)$ since $10\cdot\mathbf{T}(n) = \mathbf{T}(n+1,1)$. A well-known characterization of the 0-1 representation of well-formed parentheses strings is that every prefix contains at least as many 1's as 0's,

0 1 0 1 0 1 0 0 0 1 1 1	0 0 0 0 1 1 0 1 0 1 1 1
0 0 1 1 0 1 0 0 0 1 1 1	0 1 0 0 0 1 0 1 0 1 1 1
0 0 1 0 1 1 0 0 0 1 1 1	0 0 1 0 0 1 0 1 0 1 1 1
0 1 0 0 1 1 0 0 0 1 1 1	0 0 0 1 0 1 0 1 0 1 1 1
0 0 0 1 1 1 0 0 0 1 1 1	0 0 0 1 1 0 0 1 0 1 1 1
0 0 0 1 1 0 1 0 0 1 1 1	0 1 0 0 1 0 0 1 0 1 1 1
0 1 0 0 1 0 1 0 0 1 1 1	0 0 1 0 1 0 0 1 0 1 1 1
0 0 1 0 1 0 1 0 0 1 1 1	0 0 1 1 0 0 0 1 0 1 1 1
0 0 1 1 0 0 1 0 0 1 1 1	0 1 0 1 0 0 0 1 0 1 1 1
0 1 0 1 0 0 1 0 0 1 1 1	0 1 0 0 0 0 1 1 0 1 1 1
0 1 0 0 0 1 1 0 0 1 1 1	0 0 1 0 0 0 1 1 0 1 1 1
0 0 1 0 0 1 1 0 0 1 1 1	0 0 0 1 0 0 1 1 0 1 1 1
0 0 0 1 0 1 1 0 0 1 1 1	0 0 0 0 1 0 1 1 0 1 1 1
0 0 0 0 1 1 1 0 0 1 1 1	0 0 0 0 0 1 1 1 0 1 1 1

Table 1: A homogeneous transposition listing of $\mathbf{T}^R(6,3) = \mathbf{G}(8,3)\cdot 0111$.

and that the number of 0's equals the number of 1's. Reversing a bitstring maps the 0's to the left parentheses and the 1's to the right parentheses.

Define $\mathbf{G}(n,k)$ to be the set of all bitstrings in $\mathbf{B}(n,k)$ whose complements (0's and 1's interchanged) are prefixes of some well-formed parentheses string. Note that every prefix of an element of $\mathbf{G}(n,k)$ has at least as many 0's as 1's. Formally,

$$\mathbf{G}(n,k) \;=\; \{\mathbf{b} \in \mathbf{B}(n,k) \;\mid\; 2\sum_{i=1}^{j} b_i \le j \text{ for } 1 \le j \le n\}.$$

Note that $\mathbf{T}^R(n,k) = \mathbf{G}(2n{-}k{-}1, n{-}k)\cdot 01^k$. Denote by $g(n,k) = |\mathbf{G}(n,k)|$ the cardinality of $\mathbf{G}(n,k)$. It follows from the fact [8] that $\frac{k}{2n-k}\binom{2n-k}{n-k}$ is the number of well-formed parentheses strings of length $2n$ with prefix 1^k0, that

$$g(n,k) = \frac{n-2k+1}{n+1}\binom{n+1}{k}.$$

Table 1 shows the elements of $\mathbf{G}(8,3)\cdot 0111$.

In a slight abuse of notation we use $\mathbf{G}$ for both the set and as a list of bitstrings. The following formula is our recursive construction of the list $\mathbf{G}(n,k)$:

$$\mathbf{G}(n,k) = \begin{cases} 0^n & \text{if } k = 0 \\ 010^{n-2} \circ \cdots \circ 0^{n-1}1 & \text{if } k = 1 \\ \mathbf{G}(n-1,k)\cdot 0 \circ \overline{\mathbf{G}(n-2,k-1)}\cdot 01 \circ \mathbf{G}(n-2,k-2)\cdot 11 & \text{if } 1 < k < n/2 \\ \mathbf{G}(n-1,k-1)\cdot 1 & \text{if } k = n/2. \end{cases}$$

Note the slight differences from the definition of $\mathbf{E}(n,k)$: (a) The first element when $k = 1$, is different because of the prefix constraint on $\mathbf{G}(n,k)$, and (b) For $\mathbf{G}(n, n/2)$, the rightmost bit must be a 1, so we can redefine the list as $\mathbf{G}(n,n/2) = \mathbf{G}(n-1, n/2-1)\cdot 1$.

Lemma 2.1 *The list* $\mathbf{G}(n,k)$ *satisfies the following properties:*

1. *Successive strings differ by a transposition of two bits and all bits between those that are transposed are zeroes; i.e., the transpositions are homogeneous.*
2. $first(\mathbf{G}(n,k)) = (01)^k 0^{n-2k}$.
3. $last(\mathbf{G}(n,k)) = 0^{n-k}1^k$.

Proof: The proof is by induction on n. For $k = 1$, the list clearly has the stated properties.

For $k = n/2$, note that, *as sets*, $\mathbf{G}(2k,k) = \mathbf{G}(2k-1,k-1)\cdot 1$. Observe, furthermore, that $first(\mathbf{G}(2k,k)) = first(\mathbf{G}(2k-1,k-1))\cdot 1 = (01)^{k-1}0\cdot 1 = (01)^k$ and $last(\mathbf{G}(2k,k)) = last(\mathbf{G}(2k-1,k-1)\cdot 1 = 0^{2k-1-(k-1)}1^{k-1}\cdot 1 = 0^k 1^k$. Since a transposition that is homogeneous in $\mathbf{G}(2k-1,k-1)$ is also homogeneous in $\mathbf{G}(2k-1,k-1)\cdot 1$, it follows that all transpositions in $\mathbf{G}(2k,k)$ are homogeneous. Thus all three conditions are met in this case.

Consider now the general case. We need only consider what happens at the two interfaces between the three sublists. For $1 < k < n/2$, the list for $\mathbf{G}(n,k)$ appears as below:

$$
\begin{array}{rl}
\left.\begin{array}{r}
(01)^k 0^{n-2k-1}\mathbf{0} \\
\vdots \\
0^{n-k-1}1^{k-1}\underline{1}\;\underline{\mathbf{0}}
\end{array}\right\} & \mathbf{G}(n-1,k) \\
\left.\begin{array}{r}
0^{n-k-1}1^{k-1}\mathbf{01} \\
\vdots \\
(01)^{k-2}0\underline{1}0^{n-2k}\underline{\mathbf{0}}\mathbf{1}
\end{array}\right\} & \overline{\mathbf{G}(n-2,k-1)} \\
\left.\begin{array}{r}
(01)^{k-2}000^{n-2k}\mathbf{11} \\
\vdots \\
0^{n-k}1^{k-2}\mathbf{11}
\end{array}\right\} & \mathbf{G}(n-2,k-2)
\end{array}
$$

The positions which are underlined are the ones that change, and have no 1's between them, and so conform to the homogeneous requirement. Again all three conditions are satisfied. $\square$

3 Implementing the recursion

A *direct* algorithm [8] is one that explicitly identifies the positions of the two bits that are swapped. These positions were determined in the proof of Lemma 2.1, namely, at the first interface, positions $n-1$ and n, and at the second interface, positions $2k-2$ and $n-1$. Each time a swap occurs, the bitstring is output.

The work of the recursion is done by the procedure GEN, shown in Figure 1, which closely follows the recursive definition of the $\mathbf{G}(n,k)$ listing. The reverse listing procedure (not shown) is called NEG; it generates $\overline{\mathbf{G}(n,k)}$, and essentially executes the statements of GEN in reverse order.

```
procedure GEN(n, k)
    if k = 0 then return;
    if k = 1 then begin
        for i = 2, 3, ..., n − 1 do swap(i, i + 1);
        return;
    end;
    if 2k = n then begin
        GEN(n − 1, k − 1);
        return;
    end;
    GEN(n − 1, k);
    swap(n − 1, n);
    NEG(n − 2, k − 1);
    swap(2k − 2, n − 1);
    GEN(n − 2, k − 2);
end {of GEN};
```

Figure 1: The procedure GEN that generates the list $\mathbf{G}(n, k)$.

Because we know exactly which array element is the 1, the function $\mathsf{swap}(i, j)$ simply sets $b[i]$ to 0, and $b[j]$ to 1. After swapping, it also calls a print procedure. To generate $\mathbf{G}(n, k)$ initialize $\mathbf{b}[1 \ldots n]$ to $(01)^k 0^{n-2k}$, call print, and then call GEN(n, k). To generate $\mathbf{T}(n, k)$, also set $\mathbf{b}[2n - k \ldots 2n]$ to be 01^k, call print, which now outputs strings in reverse order, and then call GEN$(2n - k - 1, n - k)$.

4 Analysis

Consider the computation tree for this algorithm; it has internal nodes for every call to GEN or NEG. Each internal node of this tree has a single child when $n = 2k$, or has 3 children when $1 < k < n/2$. Leaves occur if $k = 0$ or $k = 1$. A constant amount of computation is done at each node of the tree, per string output. See Figure 2 for an example of a computation tree for $\mathbf{G}(8, 4)$. The boxes indicate the parameters in the calls to swap.

Theorem 4.1 *The amount of computation per string is the algorithm is constant.*

Proof: Consider two things: (a) the amount of computation used by the for-loop, and (b) all other computations. We first consider (a): Since a recursive call with $n = 2$ is impossible, the for-loop always produces at least one string and each time through the loop, a string is produced. Thus the total amount of computation used in the for-loop is $O(g(n, k))$.

The remaining computation, (b), is proportional to the number of nodes in the computation tree, since constant computation is done on each recursive call, aside from the for-loop.

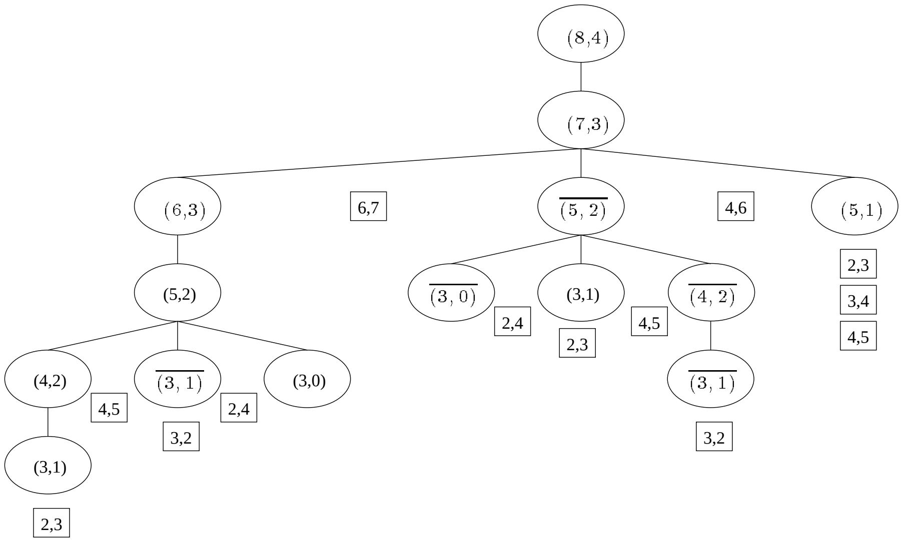

Figure 2: The computation tree of $\mathbf{G}(8,4)$.

Let n_i be the number of nodes in the computation tree with i children. Unless $i = 0, 1, 3$, the value of n_i is zero. Every node counted by n_1 has parameters $(2k, k)$. Observe that every call with parameter $(2k, k)$ results in a call with parameters $(2k-2, k-1)$, and that a call with parameters $(2k-2, k-1)$ is a leaf if $k = 2$ or has three children if $k > 2$. Thus $n_1 \leq g(n, k)$.

Since every leaf corresponds to at least one string, $n_0 \leq g(n, k)$. Also $2n_3 \leq g(n, k)$, since every node with 3 children involves 2 swaps and each swap produces a string. Finally, the total number of nodes $n_0 + n_1 + n_2$ satisfies $n_0 + n_1 + n_2 \leq 2n_0 + 2n_3 \leq g(n, k)$. Thus, the algorithm does constant computation per string.

$\square$

References

[1] S. Akl and I. Stojmenovic, "Generating t-ary Trees in Parallel," *Nordic J. of Computing*, 3 (1996) 63-71.

[2] P.J. Chase, "Combination generation and Graylex ordering," *Congressus Numerantium*, 69 (1989)215-242.

[3] P. Eades, M. Hickey and R. Read, "Some Hamilton Paths and a Minimal Change Algorithm," *Journal of the ACM*, 31 (1984) 19-29.

[4] P. Eades and B. McKay, An Algorithm for Generating Subsets of Fixed Size with a Strong Minimal Change Property," *Information Processing Letters*, 19 (1984) 131-133.

[5] J. Lucas, D. Roelants van Baronaigien and F. Ruskey, "On Rotations and the Generation of Binary Trees", *J. Algorithms*, 15 (1993) 343-366.

[6] E. Mäkinen, "A Survey of Binary Tree Codings," The Computer Journal, 34 (1991) 438-443.

[7] F. Ruskey, "Simple Combinatorial Gray Codes Constructed by Reversing Sublists," *ISAAC Conference, Lecture Notes in Computer Science*, Springer-Verlag, 762 (1993) 201-208.

[8] F. Ruskey, *Combinatorial Generation*, book in preparation.

[9] F. Ruskey and A. Proskurowski, "Generating Binary Trees by Transpositons," *J. Algorithms*, 11 (1990) 68-84.

[10] S. Zaks, "Lexicographic Generation of Ordered Trees," *Theoretical Computer Science*, 10 (1980) 63-82.

5 Appendix A: The Code Using the Direct Implementation

This appendix is not intended for publication, but just as a record of our implementation and to assist the referees of the paper.

```
/***Given the values n and k, this program generates
all the bitstrings of length 2n with exactly k leftmost 1's and
no suffix having more 1's than 0's
This is the representation for generating T(n,k)
The output is in homogeneous Gray code order
This algorithm is the DIRECT VERSION
*******/

#include <stdio.h>
#define MAX 100

/**********globals***********************/
int N; /*the total number of bits in the string*/
int K; /*the number of non-fixed 1's in the string*/
int A[MAX];  /*the holding array for the the object strings*/

/**********local functions*****************/

/*prints out the contents of A in reverse order
*/
void printIt()
{
  int i;

  for (i=N-1; i>=0; i--) {
    printf("%d ",A[i]);
  }
  printf("\n");
}

/*this will enter '0' and '1' into A[i] and A[j]
and call the printIt function
*/
swap(int i,int j)
{
  A[i] = 0;
  A[j] = 1;
  printIt();
```

```
}

/*this generates the bitstrings of length n, with only 1 '1',
  moving it from second position to last position in A.
*/
singleGen(int n)
{
  int i;

  for (i=1;i<n-1;i++) {
    swap(i,i+1);
  }
}


/*this generates the reverse of singleGen
*/
singleNeg(int n)
{
  int i;

  for (i=n-2;i>0;i--) {
    swap(i+1,i);
  }
}


/*this generates all bitstrings with k
  1's, with no prefix having more 1's than 0's.
*/
void gen(int n,int k)
{
  int i;

  if (k==0)
  return;

  if (k==1) {
    singleGen(n);
    return;
  }

  if ((2*k)==n) {
    gen(n-1,k-1);
  return; }
```

```
  gen(n-1,k);
  swap(n-2,n-1);
  neg(n-2,k-1);
  swap(2*k-3,n-2);
  gen(n-2,k-2);
}

/*this generates the reverse listing of gen
*/
neg(int n,int k)
{
  int i;

  if (k==0) {
  return; }

  if (k==1) {
    singleNeg(n);
    return;
  }

  if ((2*k)==n) {
    neg(n-1,k-1);
    return;
  }

  neg(n-2,k-2);
  swap(n-2,2*k-3);
  gen(n-2,k-1);
  swap(n-1,n-2);
  neg(n-1,k);
}

/***********main program*************/
/*receives the inputs n and k of T(n,k) from the user,
  initializes the array A and calls the generating function
*/

int main(int argc, char**argv)
{
  int n, k;
  int i;

  if (argc < 3) {
```

```
    printf("you must submit both n and k of T(n,k)\n");
    return;
  }

  n = atoi(argv[1]);
printf("the value for n is %d\n",n);
  k = atoi(argv[2]);
printf("the value for k is %d\n",k);

  N = 2*n;
  K = n-k;
  for (i=0;i<2*K;i=i+2) {
    A[i] = 0;
    A[i+1] = 1;
  }
  for (i=2*K;i<N-k;i++)
    A[i] = 0;

  for (i=N-k;i<N;i++)
    A[i] = 1;

  printIt();
  gen(N-k-1,K);
  printf("that's the end\n");
}
```