# Generating Binary Trees by Transpositions

Frank Ruskey * and Andrzej Proskurowski †

November 4, 1995

**Abstract**

Let $\mathbf{T}(n)$ denote the set of all bitstrings with $n$ 1's and $n$ 0's that satisfy the property that in every prefix the number of 0's does not exceed the number of 1's. This is a well known representation of binary trees. We consider algorithms for generating the elements of $\mathbf{T}(n)$ that satisfy one of the following constraints: (a) successive bitstrings differ by the transposition of two bits, or (b) successive bitstrings differ by the transposition of two adjacent bits. In case (a) a constant average time generation algorithm is presented. In case (b) we show that such generation is possible if and only if $n$ is even or $n < 5$. A constant average time algorithm is presented in this case as well.

## 1 Introduction

Binary Trees are of fundamental importance in computer science. In recent years there has been some interest in algorithms that generate all binary trees with a fixed number of nodes (for example, Ruskey and Hu [17], Proskurowski [10], Zaks [20], Pallo [9], Zerling [21]) or restricted classes of binary trees (for example, Lee, Lee, and Wong [6], Li [7]). Usually the trees are represented as integer sequences and those sequences are then generated. A natural question is whether the sequences can be generated so that successive sequences differ by small changes (e.g. a constant number of changes independent of the number of nodes in the tree). For general combinatorial objects the lists produced by such small change generation algorithms are sometimes called *combinatorial Gray codes* (see Joichi, White,

*Research supported by the Natural Sciences and Engineering Research Council of Canada under grant A3379.

†Partially supported by the Office of Naval research under contract N-00014-860419.

and Williamson [4]). For binary trees, Gray code type generation has been considered by Proskurowski and Ruskey [11], Lucas [8], and Roelants and Ruskey [19].

Let $\mathbf{T}(n)$ denote the set of all bitstrings with $n$ zeros and $n$ ones characterized by the *prefix property* (i.e., such that no prefix contains more 0's than 1's). Any such string can be interpreted as an encoding of an extended binary tree with $n$ internal nodes, ones representing the *internal* nodes, and zeros representing the *leaves* in the preorder traversal (the zero corresponding to the last leaf being implicit). This is perhaps the most natural of all sequence representations of binary trees. Algorithms for generating binary trees in this representation may be found in [20], [11], and elsewhere. The elements of $\mathbf{T}(n)$ can also be interpreted as well-formed parentheses strings where a one corresponds to a left parenthesis and a zero corresponds to a right parenthesis. Notice that they also have a symmetric *suffix property* that the number of 0's is at least as great as the number of 1's as the bitstring is read from right to left.

In this paper we consider bitstrings which have a fixed number of ones and zeroes. We call a list of such bitstrings a *Gray code* list if every two successive bitstrings in the list differ on exactly two positions holding 0 and 1 in one and 1 and 0, respectively, in the other; in other words, they differ by a *transposition* (or *interchange*) of two bits. A Gray code list is a *strong Gray code* list if the transposed bits are adjacent to each other.

Proskurowski and Ruskey [11] presented a Gray code algorithm for $\mathbf{T}(n)$. It is well-known that the cardinality of $\mathbf{T}(n)$ is given by a Catalan number. The Gray code algorithm of [11] was based on the classic convolution recurrence relation for Catalan numbers. The resulting algorithm generated each bitstring in time that was, on the average, $O(n)$. In this paper we develop a much simpler and more efficient algorithm; the algorithm is based on a simpler recurrence relation and runs in constant average time. The results of this paper should be construed as completely superceeding those of [11].

It was also shown in [11] that for odd $n \geq 5$ there is no strong Gray code for $\mathbf{T}(n)$. The non-existence proof made use of the *adjacent transposition graph*, $I(\mathbf{T}(n))$. For a set of bitstrings $S$ (in which each bitstring has a fixed number of ones and zeroes) the adjacent transposition graph $I(S)$ has vertex set $S$, with edges connecting those vertices that differ by an adjacent transposition. We say that two bitstrings are *adjacent* if they are adjacent in the adjacent transposition graph. A strong Gray code corresponds to a Hamilton path in the adjacent transposition graph. This graph is bipartite, and so no Hamilton path exists if the difference in the number of vertices in the two bipartitions differs by more than one.

The question of a strong Gray code for $\mathbf{T}(n)$ when $n$ is even was left unresolved in [11]. It was also given as a research problem by Proskurowski and Ruskey [12]. We will answer this question in the affirmative by presenting a constant average time algorithm to produce a Hamilton path in $I(\mathbf{T}(n))$ when $n$ is even.

The similar problem of a strong Gray code for combinations of $k$ objects out of $n$ objects as represented by bitstrings of length $n$ with exactly $k$ ones was recently resolved in three independent and different ways by Eades, Hickey, and Read [3], Buck and Wiedemann [2], and Ruskey [16]. We will imitate the approach of [16]. In order to do this we need to add an additional parameter $k$ to our problem. The addition of this parameter helps immensely; we can now follow the lead of [1] for the Gray code question, and of [16] for the strong Gray code question.

Let $\mathbf{T}(n,k)$ denote the set of all bitstrings in $\mathbf{T}(n)$ with prefix $1^k0$. These bitstrings correspond to binary trees with leftmost leaf at level $k$. If $1^k0s$ is an element of $\mathbf{T}(n,k)$ then the 1's in $s$ will be call the *free* 1's (since their position within the bitstrings can vary). By counting the sizes of the bipartitions in $\mathbf{T}(n,k)$ Ruskey and Miller [18] showed that there is no strong Gray code for $\mathbf{T}(n,k)$ for even $n$ or for odd $n > 5$ and odd $k$ (except when $k = 0, 1, n-1$, or $n$). We will present a strong Gray code algorithm for odd $n$ and even $k$ in a later section. A reason for considering a Gray code for $\mathbf{T}(n,k)$ is that a Gray code for $\mathbf{T}(n+1,1)$ can be trivially transformed into a Gray code for $\mathbf{T}(n)$ by ignoring the 10 prefix of every bitstring in $\mathbf{T}(n+1,1)$.

Let $T(n,k)$ be the number of bitstrings in $\mathbf{T}(n,k)$. It was shown in Ruskey and Hu [17] that the numbers $T(n,k)$ satisfy the following recurrence relation. (This recurrence, perhaps in a different form, certainly occurred earlier in connection with the study of Ballot problems, [14].)

$$T(n,k) = \begin{cases} T(n,2) & \text{if } k = 1 \\ T(n,k+1) + T(n-1,k-1) & \text{if } 1 < k < n \\ 1 & \text{if } k = n \end{cases} \tag{1}$$

Also, a non-recursive expression is known.

$$T(n,k) = \frac{k}{2n-k} \binom{2n-k}{n-k}$$

A constructive form of the proof of (1) is the basis of the combinatorial decompositions to follow, so we give it here after introducing a bit of notation.

For bitstrings $x$ and $y = 1^k0x$ let us define operations $flip$ and $insert$ as $flip(y) = 1^{k-1}01x$ and $insert(y) = 1^{k+1}00x$. We extend this definition to lists of bitstrings in the natural way.

To prove (1) we classify the elements of $\mathbf{T}(n,k)$ according to whether they are of the form $\alpha = 1^k01x$ or $\beta = 1^k00x$. In the former case $flip(1^{k+1}0x) = \alpha$ and in the latter $insert(1^{k-1}0x) = \beta$; elements of the argument sets are counted by $\mathbf{T}(n,k+1)$ and $\mathbf{T}(n-1,k-1)$, respectively.

The paper is organized as follows. Sections 2 develops a Gray code algorithm for $\mathbf{T}(n,k)$. Section 3 develops a strong Gray code algorithm for those cases where one exists. These two sections may be read independently of each other. Section 4 contains some concluding remarks and mentions a few open problems.

## 2 A Gray code for $\mathbf{T}(n,k)$

In this section we develop a constant average time algorithm that produces a Gray code list of $\mathbf{T}(n,k)$. 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 $L$. Clearly, the operations $first$ and $last$ commute with $flip$ and $insert$. Defining $\overline{L}$ as the reverse of the list $L$, we also have the obvious relation $first(\overline{L}) = last(L)$. Let $A \circ B$ denote the operation of concatenation of the two lists $A$ and $B$. By the proof of (1) given in the introduction, the following expression gives a recursive way of listing all the elements of $\mathbf{T}(n,k)$.

$$\mathbf{T}(n,k) = \begin{cases} flip(\mathbf{T}(n,2)) & \text{if } k=1 \\ flip(\overline{\mathbf{T}}(n,k+1)) \;\circ\; insert(\mathbf{T}(n-1,k-1)) & \text{if } 1<k<n \\ 1^n0^n & \text{if } k=n \end{cases} \tag{2}$$

In a slight abuse of notation, we let $\mathbf{T}(n,k)$ stand for both the set of bitstrings and the list of bitstrings produced by (2). By the remarks above, the corresponding recurrence with $\mathbf{T}$ replaced by $\overline{\mathbf{T}}$, and vice-versa, also holds. The reason for reversing the sublist in (2) is, of course, to produce a Gray code list as per the following theorem.

THEOREM 1 *For any values of $n$ and $k, 1 \leq k \leq n$, $\mathbf{T}(n,k)$ is a Gray code list.*

PROOF: We will prove the theorem by mathematical induction on increasing values of $n$ and decreasing values of $k$ with the hypothesis $P(n,k)$ that

$\mathbf{T}(n,k)$ for $1 \le k \le n$ is a Gray code, and that (3) and (4) hold.

$$first(\mathbf{T}(n,k)) = \begin{cases} 101100(10)^{n-3} & \text{if } k = 1 \\ 1^k 010^k (10)^{n-k-1} & \text{if } 1 < k < n \\ 1^n 0^n & \text{if } k = n \end{cases} \tag{3}$$

$$last(\mathbf{T}(n,k)) = 1^k 0^k (10)^{n-k} \text{ for } 1 \le k \le n \tag{4}$$

By definition, $P(n,k)$ is true for $n = 1$ and for $k = n$. We have to show that if $P(m,l)$ holds for all $m < n$, and for all $l > k$ when $m = n$ then this implies $P(n,k)$.

To show (3) observe for $1 < k < n$ that

$$\begin{aligned} first(\mathbf{T}(n,k)) &= first(flip(\overline{\mathbf{T}}(n,k+1))) \\ &= flip(last(\mathbf{T}(n,k+1))) \\ &= flip(1^{k+1}0^{k+1}(10)^{n-k-1}) \\ &= 1^k 010^k (10)^{n-k-1} \end{aligned}$$

And for $k = 1$

$$\begin{aligned} first(\mathbf{T}(n,1)) &= flip(first(\mathbf{T}(n,2))) \\ &= flip(110100(10)^{n-3}) \\ &= 101100(10)^{n-3} \end{aligned}$$

To show (4) observe for $1 \le k \le n$ that

$$\begin{aligned} last(\mathbf{T}(n,k)) &= insert(last(\mathbf{T}(n-1,k-1))) \\ &= insert(1^{k-1}0^{k-1}(10)^{n-k}) \\ &= 1^k 0^k (10)^{n-k} \end{aligned}$$

That $\mathbf{T}(n,k)$ is a Gray code will follow from the inductive assumption and from the proper 'interface' between the last bitstring of the first sublist $last(flip(\overline{\mathbf{T}}(n,k+1)))$ and the first bitstring of the second sublist $first(insert(\mathbf{T}(n-1,k-1)))$. If $k = 1$ then there is no interface to consider, and for $2 < k < n$,

$$\begin{aligned} last(flip(\overline{\mathbf{T}}(n,k+1))) &= flip(first(\mathbf{T}(n,k+1))) \\ &= flip(1^{k+1}010^{k+1}(10)^{n-k-1-1}) \\ &= 1^k 0110^{k+1}(10)^{n-k-2} \\ &= 1^k 0\underline{1}10^{k-1}\underline{0}0(10)^{n-k-2} \end{aligned}$$

```
procedure T ( n, k, pos : integer;  dir : boolean );
begin
   if (k = 0) or (k = n) then PrintTree else
   if k = 1 then begin
      x[pos] := 1;  T( n, k+1, pos+1, dir );  x[pos] := 0;
   end else
      if dir then begin
         x[pos] := 1;  T( n, k+1, pos+1, not dir );  x[pos] := 0;
         T( n-1, k-1, pos+1, dir );
      end else begin
         T( n-1, k-1, pos+1, dir );
         x[pos] := 1;  T( n, k+1, pos+1, not dir );  x[pos] := 0;
      end;
end {of T};
```

Algorithm 1: The Gray code algorithm.

and

$$\begin{aligned} first(insert(\overline{\mathbf{T}}(n-1,k-1))) &= insert(first(\mathbf{T}(n-1,k-1))) \\ &= insert(1^{k-1}010^{k-1}(10)^{n-k-1}) \\ &= 1^k0010^{k-1}(10)^{n-k-1} \\ &= 1^k0\underline{0}10^{k-1}\underline{1}0(10)^{n-k-2} \end{aligned}$$

If $k=2$, $last(flip(\overline{\mathbf{T}}(n,k+1))) = 110\underline{1}1\underline{0}00(10)^{n-4}$ and $first(insert(\mathbf{T}(n-1,k-1))) = 110\underline{0}1\underline{1}00(10)^{n-4}$. As indicated by the underlined bits, the two bitstrings differ on two positions. ■

The recursive algorithm producing the list $T(n,k)$ is simple enough to be given explicitly as Algorithm 1. The global array `x` is initialized with $1^k0^{2n-k}$ and the initial call is `T(n,k,k+2,true)`. The parameter `pos` keeps track of where the free 1's are to be positioned.

Let $S(n,k)$ denote the number of calls to procedure `T` with parameters $n$ and $k$. Since on every level of recursive calls, procedure `T` requires constant time computation, the time complexity of the algorithm is proportional to $S(n,k)$. The numbers $S(n,k)$ satisfy the recurrence relation given below.

$$S(n,k) = \begin{cases} 1+S(n,2) & \text{if } k=1 \\ 1+S(n,k+1)+S(n-1,k-1) & \text{if } 1<k<n \\ 1 & \text{if } k=n \end{cases}$$

It can be shown by an easy induction that $S(n,k) \leq 3T(n,k) - 2$ for $1 < k \leq n$ and $S(n,1) \leq 3T(n,1) - 1$. This proves that $S(n,k)$ is proportional to $T(n,k)$ and thus there are constant (on the average) changes necessary to produce the next string in the list.

Note that the bitstrings $last(\mathbf{T}(n,k))$ and $first(\mathbf{T}(n,k-1))$ are adjacent, as are $last(\mathbf{T}(n,2))$ and $last(\mathbf{T}(n,1))$. Thus the following is a Gray code list for $\mathbf{T}(n)$.

$$\mathbf{T}(n) = \mathbf{T}(n,n) \circ \mathbf{T}(n,n-1) \circ \cdots \circ \mathbf{T}(n,2) \circ \overline{\mathbf{T}}(n,1)$$

This list starts with bitstring $1^n0^n$ and ends with $101100(10)^{n-3}$. Note that these bitstrings cannot be obtained from one another by a transposition.

As noted before, in the case of $k = 1$, the algorithm for $\mathbf{T}(n+1,1)$ gives rise to yet another Gray code listing of $\mathbf{T}(n)$ by ignoring the initial 10 prefix of the bitstrings in $\mathbf{T}(n+1,1)$. In this case the list starts with the bitstring $1100(10)^{n-2}$ and ends with $1010(10)^{n-2}$; these bitstrings are adjacent and hence a different list is produced from the one described in the preceeding paragraph.

Both of these lists are different from the list in [11]. Since the algorithm for listing $\mathbf{T}(n,k)$ runs in constant average time, the algorithms for listing $\mathbf{T}(n)$ also run in constant average time. These algorithms are clearly simpler and more efficient than the one in [11].

## 3 A strong Gray code for $\mathbf{T}(n,k)$

In this section we develop a recursive approach for finding a Hamilton path in the adjacent transposition graph when one exists.

As was noted in the introduction, the graph $I(\mathbf{T}(n,k))$ is bipartite. The vertices of the bipartition may be classified according to the parity of the length of the shortest path in $G(n,k)$ to the *canonical bitstring*, $1^k01^{n-k}0^{n-1}$. Let $D(n,k)$ denote the difference in the number of even distance vertices minus the number of odd distance vertices. It is interesting that this difference is always non-negative. There is a simple recurrence relation for the $D(n,k)$ numbers.

LEMMA 1

$$D(n,k) = \begin{cases} D(n,2) & \text{if } k = 1 \\ D(n,k+1) + (-1)^{n-k} D(n-1,k-1) & \text{if } 1 < k < n \\ 1 & \text{if } k = n \end{cases} \qquad (5)$$

PROOF: For a bitstring of the form $1^k01s$, the distance from the canonical bitstring $1^k01^{n-k}0^{n-1}$ is the same as the distance from $1^{k+1}0s$ to the canonical string $1^{k+1}01^{n-k-1}0^{n-1}$ of $\mathbf{T}(n,k+1)$. Thus, the parity difference contributed to $D(n,k)$ by all such bitstrings is $D(n,k+1)$. Similarly, the distance from a bitstring $1^k00s$ to $\alpha = 1^k001^{n-k}0^{n-2}$ is the same as the distance from $1^{k-1}0s$ to the canonical string of $\mathbf{T}(n-1,k-1)$. The distance from $\alpha$ to the canonical bitstring of $D(n,k)$ is $n-k$. Thus, the contribution to $D(n,k)$ from all strings of this form is $(-1)^{n-k}D(n-1,k-1)$. ■

LEMMA 2 *For* $1 \leq k \leq n$

$$D(n,k) = \begin{cases} 0 & \text{if } n \text{ odd, } k \text{ even, or } n \text{ odd, } k=1 \\ T(\lfloor n/2 \rfloor, \lceil k/2 \rceil) & \text{otherwise} \end{cases} \tag{6}$$

PROOF: An easy, if somewhat tedious, inductive proof using (5) can be given. An alternative proof is to use Theorem 2 of Ko and Ruskey [5], after noticing that $\mathbf{T}(n,k)$ is isomorphic to the set of ordered forests with $k$ trees and $2n-k$ nodes, where the number of vertices with (out)degree two is $n-k$ and the number of leaves (e.g., nodes of degree 0) is $n$. Yet another proof can be derived by using an involution similar to the ones used in Ruskey [15]. ■

The parity difference is one if and only if $k=n$, or $k=n-1$ and $k$ is odd, or $n=4$, or $n=5$ and $k=3$. The reader can easily check that $I(\mathbf{T}(n,k))$ has a Hamilton path in those cases. Thus we can now turn our attention to the cases where the parity difference is zero. If $k=n-1$ and $k$ is even then the adjacent transposition graph is a path and thus has a Hamilton path.

In the remainder this section we show how to construct a Hamilton path in $I(\mathbf{T}(n,k))$ if $n$ is odd and $k \geq n-3$ is even (note that the graphs $I(\mathbf{T}(n,1))$ and $I(\mathbf{T}(n,2))$ are isomorphic). The construction of the path is based on the identity

$$T(n,k) = \sum_{i=0}^{(k-2)/2} [T(n-2i, k-2i+1) + T(n-2i-1, k-2i)] \tag{7}$$

Identity (7) can be proven by iterating (1). Another way of proving (7) is by classifying the elements of $\mathbf{T}(n,k)$ according to the position of the first (leftmost) free 1 in the bitstring. The terms of (7) are paired together to form $k/2$ *blocks*. The blocks are indicated in (7) by the square brackets. We will show how to obtain a Hamilton path in the graph whose vertices are determined by each block and how to connect the paths together to form a Hamilton path in $I(\mathbf{T}(n,k))$.

The construction is recursive in (decreasing) values of $k$. The base case $k = n-3$ is easy to visualize since the vertices of the graph $I(\mathbf{T}(n,n-3))$ can be thought of as existing in three-space, with a dimension associated with each of the three rightmost 1's (e.g. the three free 1's) in the bitstring.

Figure 1 shows the graph $I(\mathbf{T}(5,2))$ with each vertex labeled by its bitstring. There is only one block. The figure also shows the coordinate system that is being used. Direction $i$ indicates movement of the $i$th free 1 in the bitstring, and the arrows direction indicates movement of that 1 to the right. In the discussion that follows these directions will be indicated with arrows such as $\uparrow, \swarrow$, etc.

Figure 2 shows a Hamilton path in the subgraphs that correspond to the smallest three blocks of (7) with $k = n-3$ and how they are connected. Only the edges on the Hamilton path are drawn. It should be clear what the vertices would be in succeeding blocks. From this figure it is evident how to construct a Hamilton path for any even $n$ and $k = n-3$. In fact, there are many Hamilton paths, but the one indicated is particularly useful when generalizing to arbitrary even $k$.

Define a graph $G_m$ as follows:

$$G_m = I(flip(\mathbf{T}(m+1, m-1)) \cup insert(flip(\mathbf{T}(m, m-2))))$$

The graph of the $j$th smallest size block for $k = n-3$ is isomorphic to $G_{2(j+1)}$. For example, the graph of Figure 1 is $G_4$, and Figure 2 shows Hamilton paths in $G_4, G_6$ and $G_8$.

We now start to extend the base case $k = n-3$ to the general case of odd $n$ and even $k \geq n-3$. The basic idea is to use the $G_m$ graphs of Figures 1 and 2 except that now each vertex is regarded as being the adjacent transposition graph of a set of bitstrings, rather than a single bitstring. A vertex in $I(\mathbf{T}(n,n-3))$ has the form $1^{n-3}0x10^j$ where $x$ contains two 1's. The corresponding set of bitstrings in the generalized vertex for $\mathbf{T}(n+i,n-3)$ is $1^{n-3}0x1\underline{1^i0^{i+j}}$. The notation $\underline{y}$ indicates the set of all bitstrings with as many 0's and 1's as $y$, and that satisfy the suffix condition. In other words, the generalized vertices classify the elements of $\mathbf{T}(n,k)$ according to the positions of the first (leftmost) three free 1's. A generalized vertex is called a *g-vertex.*

It is not always possible to find a Hamilton path in a g-vertex. For example, there is no Hamilton path in $I(111\underline{1100000})$ since the corresponding

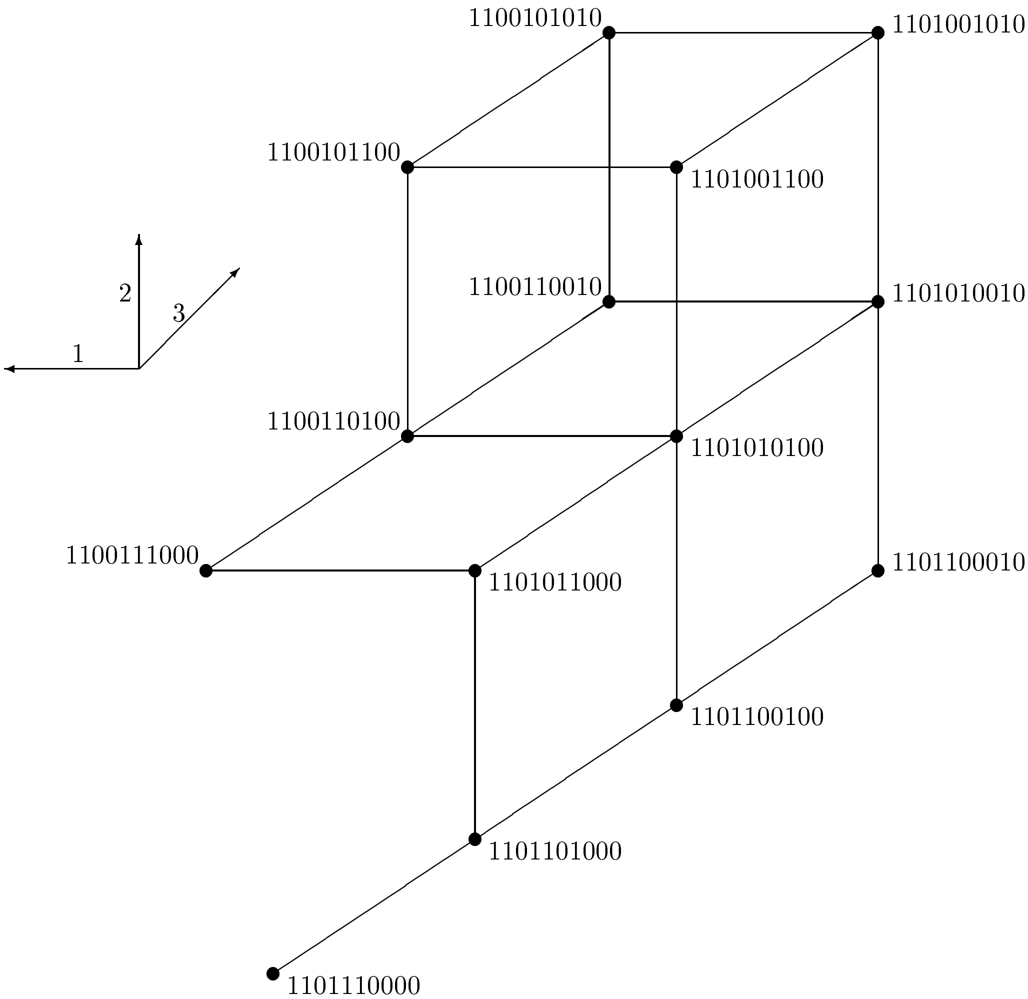

Figure 1: The graph $I(\mathbf{T}(5,2))$.

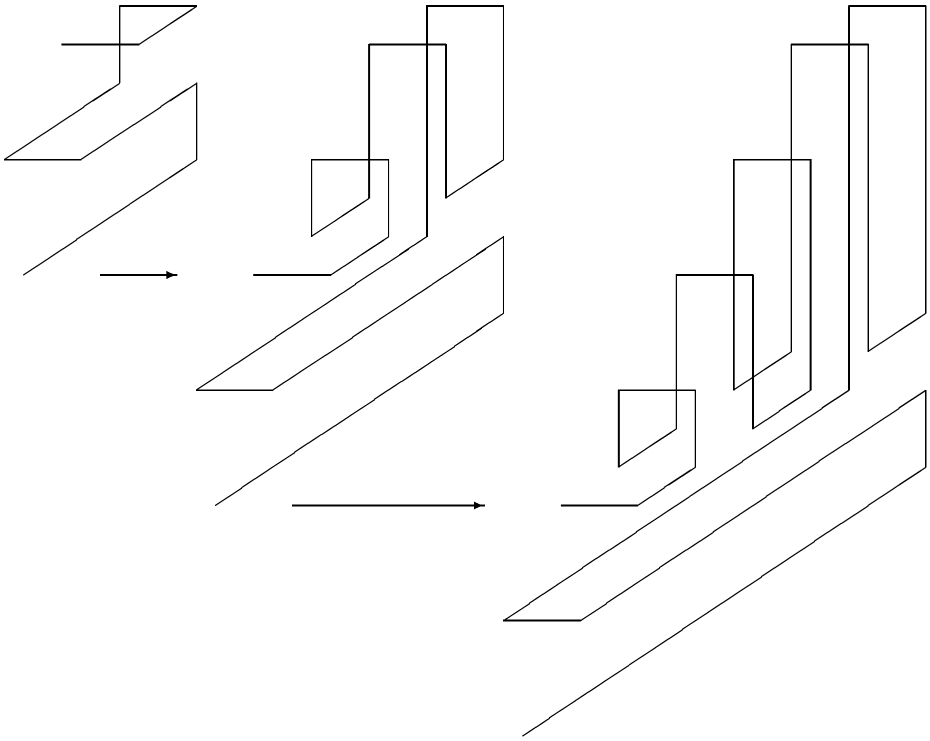

Figure 2: Hamilton path in three smallest blocks when $k = n - 3$.

adjacent transposition graph is isomorphic to $I(\mathbf{T}(6,4))$, for which $D(6,4) = 2$. We instead consider adjacent pairs of g-vertices, as in [16]. Two kinds of adjacency are possible. The g-vertices $x\underline{1^s0^t}$ and $y\underline{1^s0^t}$ are adjacent if $x$ and $y$ are adjacent. Also g-vertices $x1\underline{1^s0^{t+1}}$ and $x01\underline{1^s0^t}$ are adjacent. Our strategy is to find a Hamilton path by connecting up adjacent pairs of g-vertices. Within each pair of g-vertices we must find a Hamilton path that starts in one g-vertex of the pair and ends in the other g-vertex. Also, the Hamilton path must start and end with a g-vertex pairs. For this to occur there must be an even number of bitstrings in the union of the two g-vertex pairs.

In order to motivate the constructions that are to follow let us consider the specific case of $\mathbf{T}(7,2)$. There is one block of g-vertices as shown in Figure 3. The g-vertex pairs are shown connected with wavey lines. The vertex labels of Figure 3 have the prefix preceeding the first free 1 omitted for brevity (the prefix is either 110 or 1100). The g-vertex pair $\{110111\underline{11000000}, 1101101\underline{0110000}\}$ has adjacent transposition graph isomorphic to $G_6$, and the vertex pairs $\{11011001\underline{110000}, 110110001\underline{01100}\}$, $\{11001101\underline{110000}, 110011001\underline{01100}\}$,
and $\{11010101\underline{110000}, 110101001\underline{01100}\}$ have adjacent transposition graphs isomorphic to $G_4$. We are not allowed to pair, for example, the vertices $\{110111\underline{0110000}, 1101101\underline{110000}\}$ since this represents an odd number of bitstrings (namely $9 + 14 = 23$). In general this means that we never need to consider $G_m$ if $m$ is odd.

The other pairs in the Hamilton path, differing only by the position of the first free 1, $\{11001011\underline{110000}, 11010011\underline{001100}\}$, $\{110010101\underline{01100}, 110100101\underline{01100}\}$, and $\{1100111\underline{0110000}, 1101011\underline{0110000}\}$ have adjacent transposition graphs that are isomorphic to a different type of graph, $H_m$, as defined below.

$$H_m = I(10\mathbf{T}(m, m-2) \cup 01\mathbf{T}(m, m-2))$$

Note that $H$ graphs arise no matter what Hamilton path was taken through Figure 3. We are now faced with the problem of finding suitable Hamilton paths in $G$ graphs and in $H$ graphs. Figure 4 shows a Hamilton path in a $G$ graph. It is desirable to keep the number of Hamilton paths that need to be considered to a minimum. It turns out that one kind of path in the $G$ graphs, and three kinds of paths in the $H$ graphs are sufficient to allow the recursive construction of a Hamilton path in $\mathbf{T}(n,k)$ to be completed.

Since there are more free 1's to consider for $n - k$ greater than 5 we would have g-vertices in $H$ graphs to consider, just as we had g-vertices in

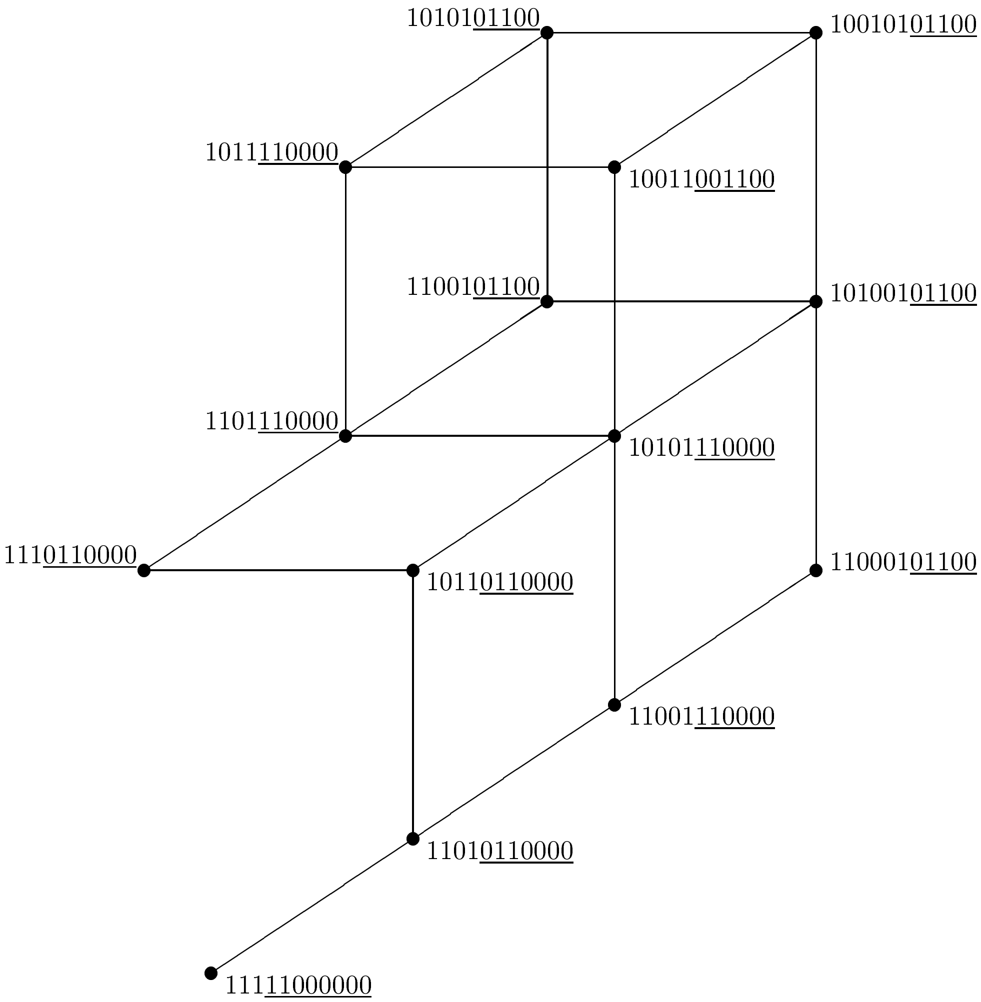


Figure 3: Generalized vertices in $G_4$ corresponding to $I(\mathbf{T}(7,2))$.

$G$ graphs for $n - k = 5$. We will now describe the general construction. Many different Hamilton paths have been considered by the authors; the ones presented here are believed to lead to the simplest algorithm, given the approach of finding Hamilton paths in pairs of g-vertices.

For general even $k$ the construction of the Hamilton path in $I(\mathbf{T}(n,k))$ involves four graphs together with a Hamilton path in each graph. The Hamilton path is specified recursively by using g-vertex pairs in a $G$ graph or in an $H$ graph. Each g-vertex pair is represented by an edge that is labeled with a type of Hamilton path. These Hamilton paths (with the underlying $G$ or $H$ graph being understood) are denoted $A$, $X$, $Y$, and $Z$. Each instance of these graphs have two parameters $s$ and $t$ which define it up to graph isomorphism. An $A$ path consists of a $G$ graph and a labeled Hamilton path. The $X$,$Y$, and $Z$ paths consist of a $H$ graph and a Hamilton path for each.

In Figures 4, 5, 6, 7 certain edge labelings of g-vertex pairs along the Hamilton paths are implicit. In direction $\swarrow$ the label is always $A$, in direction $\nearrow$ (direction 3 in Figure 1) the label is always $A^{-1}$, and in directions $\leftrightarrow$ and $\updownarrow$ the label is $X$ unless indicated otherwise. Also, note that g-vertex pairs are shown with wavy lines, and the initial vertex is indicated with a square.

Table 1 shows the initial and final bitstrings in the four path types. It is assumed that $x$ and $y$ differ by an adjacent transposition, and that $t \geq s$. Note the symmetry of the initial and final vertices of $X$ and $Y$ graphs. In fact these paths differ only by the addition and removal of a single edge in their respective Hamilton paths. The paths $A$ and $Z$ are sometimes traversed in the opposite direction; that is to say from the final vertex to the initial vertex. These reversed paths are denoted by $A^{-1}$ and $Z^{-1}$. Because of symmetry we do not have to consider reversed paths in $X$ and $Y$ graphs. The underline notation is slightly abused in this table; it indicates both the set of bitstrings under consideration and the actual arrangements of bits in the initial and final bitstrings.

In order to justify the construction we need to examine the paths and how they interact in greater detail. Specifically, we must check that the claimed initial and final bitstrings are as stated in the table, and that the labels on the g-vertex pairs along the Hamilton path have the proper interface. By *the proper interface* we mean that in two succeeding g-vertex pairs the final bitstring of the first must differ from the first bitstring of the second by the appropriate adjacent transposition. Table 2 contains the labels of succeeding g-vertex pairs, and the intervening adjacent transposition, that occur in our

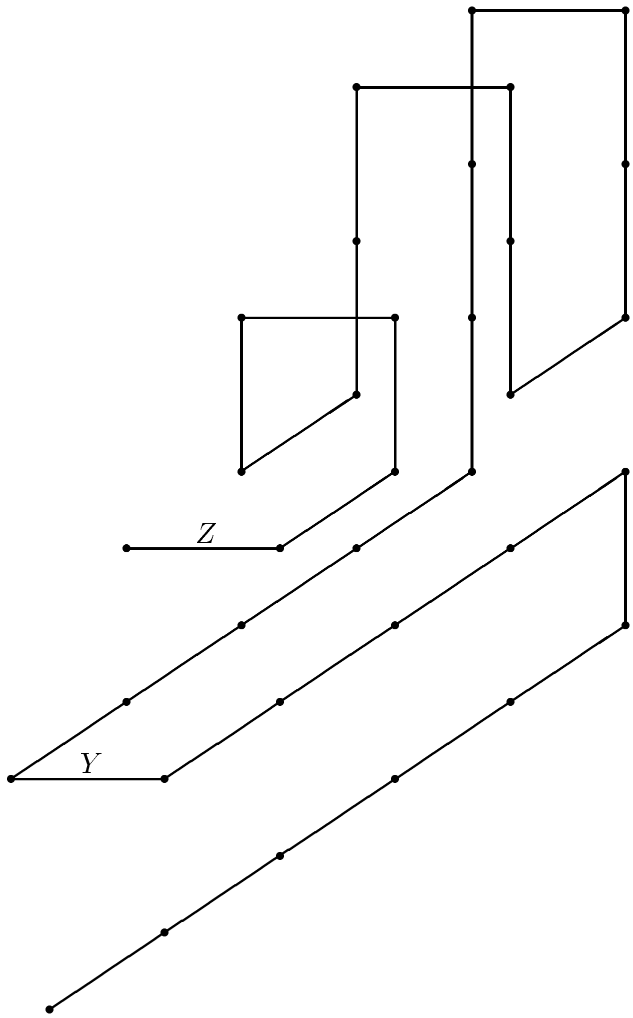


Figure 4: The graph $A$.

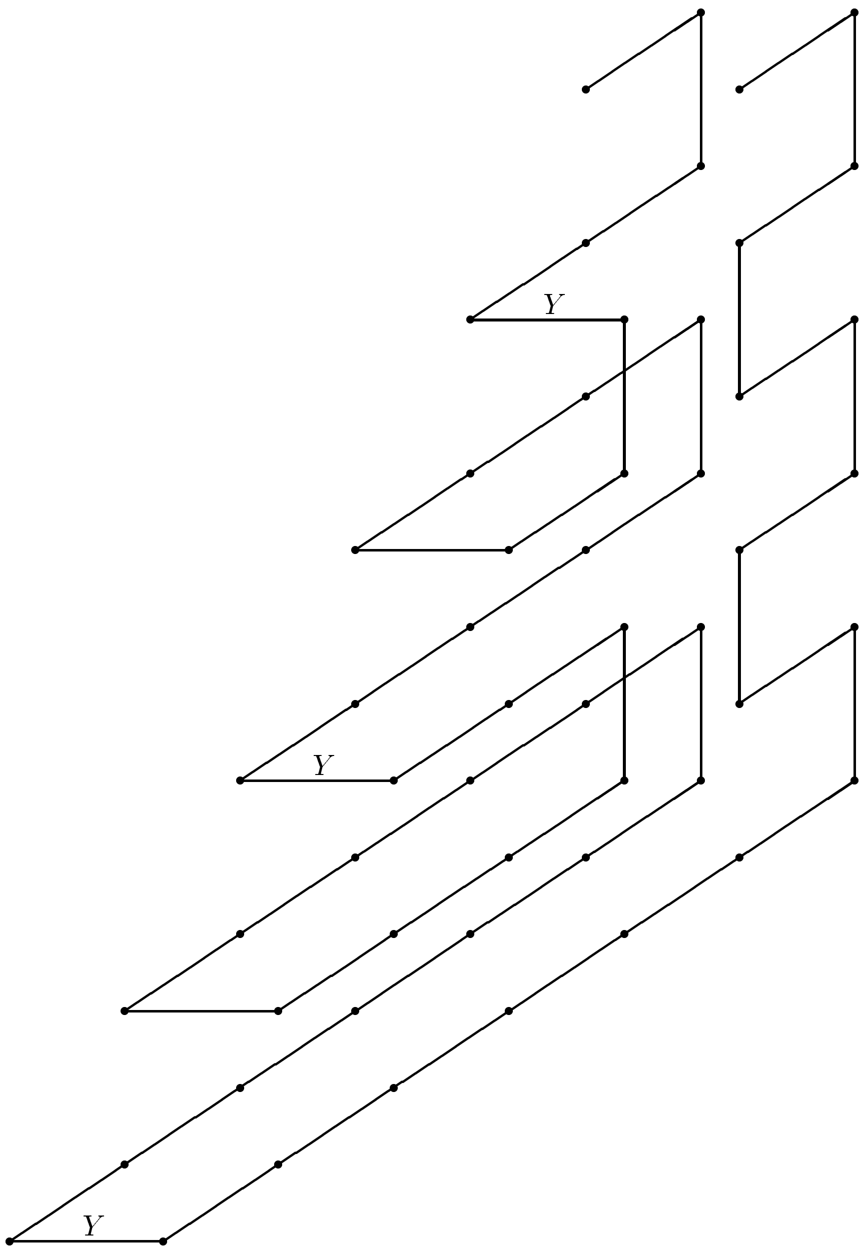


Figure 5: The graph $X$.

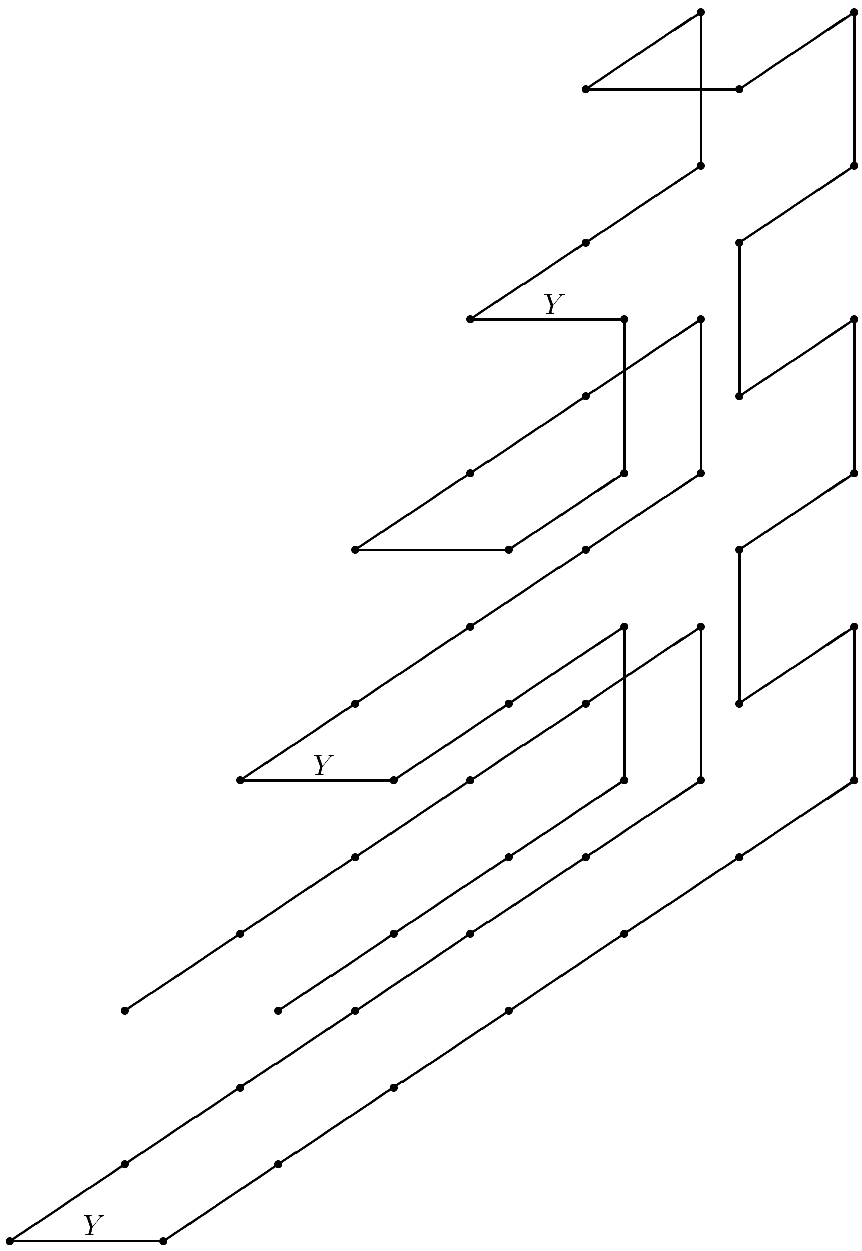


Figure 6: The graph $Y$.

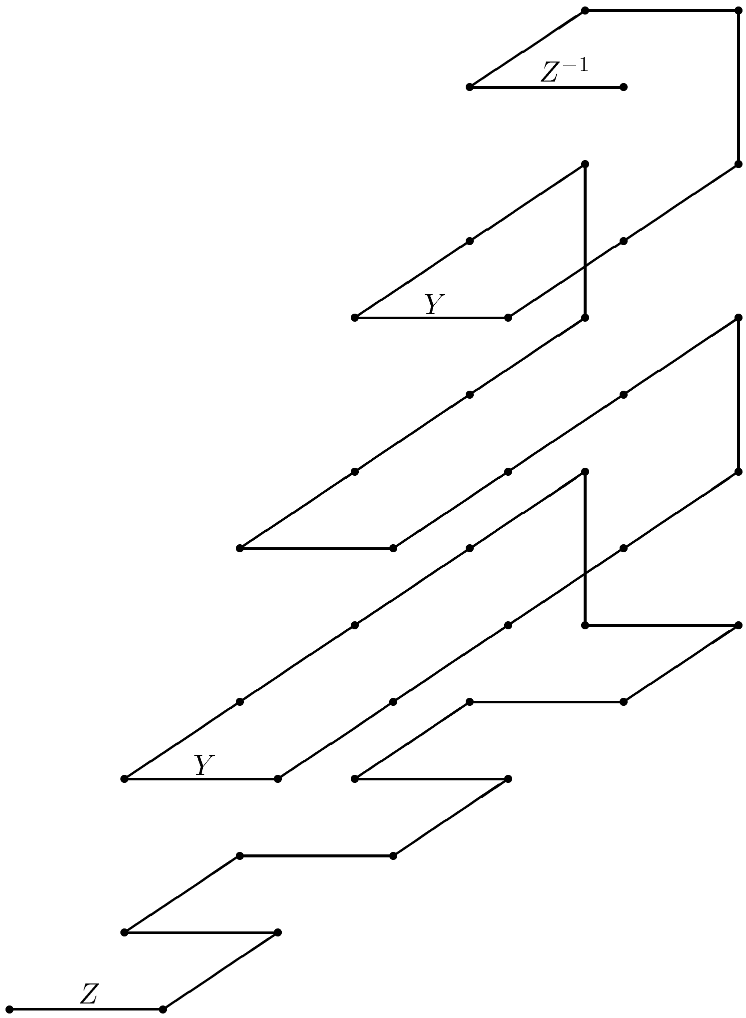


Figure 7: The graph $Z$.

| type | initial | final | conditions | direction |
|---|---|---|---|---|
| $A(s,t)$ | $x01\underline{01^s0^t}$ | $x1\underline{1^s0^{t+2}}$ | $s,t$ even | $\swarrow$ |
| $X(s,t)$ | $x1\underline{0^{t-s}1^s0^s}$ | $y1\underline{0^{t-s}1^s0^s}$ | $s$ even | $\leftrightarrow, \updownarrow$ |
| $Y(s,t)$ | $x1\underline{01^s0^t}$ | $y1\underline{01^s0^t}$ | $s,t$ even | $\leftrightarrow$ |
| $Z(s,t)$ | $x1\underline{1^s0^t}$ | $y1\underline{0^{t-s}1^s0^s}$ | $s,t$ even | $\leftrightarrow$ |

Table 1: The path types.

graphs. The reader can check, using Table 1, that these sequencings are valid. We also leave it to the reader to check that the appropriate intial and final bitstring are produced by the labelings used in Figures 4, 5, 6, and 7.

In order to generate the elements of $\mathbf{T}(n)$ as a strong Gray code list when $n$ is even we observe that the graphs $I(\mathbf{T}(n+1,2))$, $I(\mathbf{T}(n+1,1))$, and $I(\mathbf{T}(n))$ are all isomorphic. Thus, the algorithm for $\mathbf{T}(n+1,2)$ will suffice except that the prefix 110 is replaced by the prefix 1. Using this algorithm the first bitstring of the list is $10101^{n-2}0^{n-2}$ and the last bitstring is $1^n0^n$.

The results of this section are summarized by the following statement.

Theorem 2 *The graph $I(\mathbf{T}(n,k))$ has a Hamilton path if and only if $D(n,k)$ is at most one.*

## 4 Concluding Remarks

Several algorithms have been given for generating the elements of $\mathbf{T}(n)$, including the two presented here and those of [11]. Some of these algorithms run in constant average time but none of them run in constant worst case time. It now seems possible to produce such an algorithm by a more in-depth analysis of the transposition algorithm presented in this paper. This is the same route followed in [1] (see also [13]) for bitstrings representing combinations: A careful analysis of the recursive generation algorithm produced a constant average time algorithm.

It is remarkable that Theorem 2 is almost exactly the same as theorems in [2], [3], and [16] except that the set $\mathbf{T}(n,k)$ is replaced by $\mathbf{C}(n,k)$, the set of bitstrings representing combinations of $k$ items out of $n$ items. One is

| g-vertex | connection | g-vertex |
|---|---|---|
| $A^{-1}(s,t)$ | $\leftrightarrow, \updownarrow$ | $A(s,t)$ |
| $A(s,t)$ | $\swarrow$ | $A(s,t+2)$ |
| $A(s,t)$ | $\swarrow$ | $Y(s,t+2)$ |
| $X(s,t)$ | $\leftrightarrow, \updownarrow$ | $X(s,t)$ |
| $X(s,t)$ | $\swarrow$ | $X(s,t+1)$ |
| $Z(s,t)$ | $\nearrow$ | $X(s,t-1)$ |
| $X(s,s+1)$ | $\leftrightarrow, \updownarrow$ | $A(s,s)$ |
| $Y(s,t)$ | $\downarrow$ | $A(s,t)$ |

Table 2: Possible sequences of g-vertex pairs.

tempted to try to find a generalization that encompasses all these theorems. This is the subject of future investigations.

The strong Gray code algorithm for binary trees and $\mathbf{T}(n,k)$ has been implemented as a Pascal program and can be obtained from the authors. The program is similar to the one published in [16].

## References


[1] J.R. Bitner, G. Ehrlich, and E.M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Comm. ACM*, 19:517–521, 1976.

[2] M. Buck and D. Wiedemann. Gray codes with restricted density. *Discrete Math.*, 48:163–171, 1984.

[3] P. Eades, M. Hickey, and R.C. Read. Some Hamilton paths and a minimal change algorithm. *JACM*, 31:19–29, 1984.

[4] J.T. Joichi, D.E. White, and S.G. Williamson. Combinatorial gray codes. *SIAM J. Computing*, 9:130–141, 1980.

[5] C.W. Ko and F. Ruskey. Solution of some multi-dimensional lattice path parity difference recurrence relations. *Discrete Math.*, to appear.

[6] C.C. Lee, D.T. Lee, and C.K. Wong. Generating binary trees of bounded height. *Acta Informatica*, 23:529–544, 1986.

[7] L. Li. Ranking and unranking of AVL trees. *SIAM J. Computing*, 15:1025–1035, 1986.

[8] J. Lucas. The rotation graph of binary trees is Hamiltonian. *J. Algorithms*, 8:503–535, 1987.

[9] J. Pallo. Enumerating, ranking, and unranking binary trees. *Computer J.*, 29:171–175, 1986.

[10] A. Proskurowski. On the generation of binary trees. *JACM*, 27:1–2, 1980.

[11] A. Proskurowski and F. Ruskey. Binary tree gray codes. *J. Algorithms*, 6:225–238, 1985.

[12] A. Proskurowski and F. Ruskey. Research problem 88. *Discrete Math.*, 35:321–322, 1987.

[13] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[14] D. Rotem and Y. Varol. Generation of binary trees from ballot sequences. *JACM*, 25:396–404, 1978.

[15] F. Ruskey. Solution of some lattice path parity difference recurrence relations. *Congressus Numerantium*, 59:257–266, 1987.

[16] F. Ruskey. Adjacent interchange generation of combinations. *J. Algorithms*, 9:162–180, 1988.

[17] F. Ruskey and T.C. Hu. Generating binary trees lexicographically. *SIAM J. Computing*, 6:745–758, 1977.

[18] F. Ruskey and D.J. Miller. Adjacent interchange generation of combinations and trees. Technical Report DCS-44-IR, U. Victoria, 1984.

[19] D. Roelants van Baronaigien and F. Ruskey. A Hamilton path in the rotation lattice of binary trees. *Congressus Numerantium*, 59:313–319, 1987.

[20] S. Zaks. Lexicographic generation of ordered trees. *Theoretical Computer Science*, 10:63–82, 1980.

[21] D. Zerling. Generating binary trees using rotations. *JACM*, 32:694–701, 1985.

## 5 Appendix

This appendix contains an implementation of the strong Gray code algorithm for generating $\mathbf{T}(n,k)$ and $\mathbf{T}(n)$. It is not intended to appear in the published version of the paper.

```
(****************************************************************)
(*                                                              *)
(*   This program implements the adjacent interchange binary tree  *)
(*   generation algorithm of Ruskey and Proskurowski.           *)
(*                                                              *)
(****************************************************************)

program GenerateBinaryTrees (input,output);

const
   MAX = 100;
   forw = +1;  back = -1;
   riht = +1;  left = -1;

type
   range = 0..MAX;
   direction = -1..1;

var
   n,k : integer;
   x : array [range] of 0..1;
   y : array [range] of range;

procedure A ( s, t : range;  dir : direction;
              p : range;  dirp : direction ); forward;
procedure Z ( s, t : range;  dir : direction;
              p : range;  dirp : direction ); forward;
procedure X ( s, t : range;
              p : range;  dirp : direction ); forward;
procedure Y ( s, t : range;
              p : range;  dirp : direction ); forward;


procedure WriteBinaryTree;
var i : integer;
begin
   for i := 1 to 2*n do write( x[i]:1 );
```

```
   writeln;
end {of WriteCombination};


procedure Move ( p : range;  dir : direction );
(* Moves p-th 1 in direction dir where -1 = left, 1 = right *)
begin
   WriteBinaryTree;
   x[y[p]] := 0;
   y[p] := y[p] + dir;
   x[y[p]] := 1;
end {of Move};


procedure MA ( s, t : range;  dir : direction );
begin
   Move( s-1, -dir );
   A( s-2, t, +dir, s-1, -dir );
end {of MA};

procedure AM ( s, t : range;  dir : direction );
begin
   A( s-2, t, +dir, s-1, -dir );
   Move( s-1, -dir );
end {of MA};


procedure XM ( s, t : range;  dir : direction );
begin
   X( s-2, t, s, dir );
   Move( s, dir );
end {of MX};

procedure MX ( s, t : range;  dir : direction );
begin
   Move( s, dir );
   X( s-2, t, s, dir );
end {of MX};


procedure A;
(* Generates from 0101...10...0 to 11...100...0 *)
var i, j : range;
begin
```

```
if s = 0 then Move( p, dirp ) else
if dir = forw then begin
  Z( s-2, t, forw, p, +dirp );
  Move( s-1, riht );
  for i := 1 to (t-s) div 2 do begin
    X( s-2, t-2*i+1, s, riht );
    for j := 1 to i-1 do MX( s, t-2*i+1, riht );
    Move( p, -dirp );
    for j := i-1 downto 1 do XM( s, t-2*i+1, left );
    X( s-2, t-2*i+1, s, left );
    Move( s-1, riht );
    for j := 1 to i do XM( s, t-2*i, riht );
    X( s-2, t-2*i, p, +dirp );
    for j := i downto 1 do MX( s, t-2*i, left );
    Move( s-1, riht );
  end;
  for j := 1 to (t-s) div 2 do XM( s, s-1, riht );
  X( s-2, s-1, p, -dirp );
  for j := 1 to (t-s) div 2 do MX( s, s-1, left );
  Move( s, left );
  for i := 0 to (t-s) div 2 do AM( s, s-2+2*i, forw );
  Y( s-2, t, p, +dirp );
  for i := (t-s) div 2 downto 0 do MA( s, s-2+2*i, back );
  Move( s, left );
  for i := 0 to (t-s) div 2 do AM( s, s-2+2*i, forw );
  A( s-2, t, forw, s-1, left );
end else {dir = back} begin
  A( s-2, t, back, s-1, riht );
  for i := (t-s) div 2 downto 0 do MA( s, s-2+2*i, back );
  Move( s, riht );
  for i := 0 to (t-s) div 2 do AM( s, s-2+2*i, forw );
  Y( s-2, t, p, +dirp );
  for i := (t-s) div 2 downto 0 do MA( s, s-2+2*i, back );
  Move( s, riht );
  for j := (t-s) div 2 downto 1 do XM( s, s-1, riht );
  X( s-2, s-1, p, -dirp );
  for j := (t-s) div 2 downto 1 do MX( s, s-1, left );
  for i := (t-s) div 2 downto 1 do begin
     Move( s-1, left );
     for j := 1 to i do XM( s, t-2*i, riht );
     X( s-2, t-2*i, p, +dirp );
     for j := i downto 1 do MX( s, t-2*i, left );
     Move( s-1, left );
     X( s-2, t-2*i+1, s, riht );
```

```
      for j := 1 to i-1 do MX( s, t-2*i+1, riht );
      Move( p, -dirp );
      for j := i-1 downto 1 do XM( s, t-2*i+1, left );
      X( s-2, t-2*i+1, s, left );
    end;
    Move( s-1, left );
    Z( s-2, t, back, p, +dirp );
  end;
end {of A};


procedure Z;
var i,j : integer;
begin
  if s = 0 then Move( p, dirp ) else
  if dir = forw then begin
    {Bottom plane of graph}
    Z( s-2, t, forw, p, +dirp );
    for i := 0 to (t-s) div 2 - 1 do begin
      Move( s-1, riht );
      X( s-2, t-1-2*i, p, -dirp );
      Move( s-1, riht );
      X( s-2, t-2-2*i, p, +dirp );
    end;
    Move( s-1, riht );
    X( s-2, s-1, p, -dirp );
    Move( s, riht );
    {Middle portion of graph}
    for i := (t-s) div 2 downto 2 do begin
      for j := 0 to i-1 do AM( s, s-2+2*j, forw );
      Y( s-2, s-2+2*i, p, +dirp );
      for j := i-1 downto 0 do MA( s, s-2+2*j, back );
      Move( s, riht );
      A( s-2, s-2, forw, s-1, left );
      for j := 1 to i-1 do MA( s, s-2+2*j, forw );
      Move( p, -dirp );
      for j := i-1 downto 1 do AM( s, s-2+2*j, back );
      A( s-2, s-2, back, s-1, riht );
      Move( s, riht );
    end;
    {Final portion of graph}
    A( s-2, s-2, forw, s-1, left );
    Move( s-1, left );
    Y( s-2, s, p, +dirp );
```

```
    Move( s-1, riht );
    A( s-2, s-2, back, s-1, riht );
    Move( s, riht );
    X( s-2, s-1, p, -dirp );
    Move( s-1, left );
    Z( s-2, s, back, p, +dirp );
  end else {dir = back} begin
    Z( s-2, s, forw, p, +dirp );
    Move( s-1, riht );
    X( s-2, s-1, p, -dirp );
    Move( s, left );
    A( s-2, s-2, forw, s-1, left );
    Move( s-1, left );
    Y( s-2, s, p, +dirp );
    Move( s-1, riht );
    A( s-2, s-2, back, s-1, riht );
    for i := 2 to (t-s) div 2 do begin
      Move( s, left );
      A( s-2, s-2, forw, s-1, left );
      for j := 1 to i-1 do MA( s, s-2+2*j, forw );
      Move( p, -dirp );
      for j := i-1 downto 1 do AM( s, s-2+2*j, back );
      A( s-2, s-2, back, s-1, riht );
      Move( s, left );
      for j := 0 to i-1 do AM( s, s-2+2*j, forw );
      Y( s-2, s-2+2*i, p, +dirp );
      for j := i-1 downto 0 do MA( s, s-2+2*j, back );
    end;
    Move( s, left );
    X( s-2, s-1, p, -dirp );
    Move( s-1, left );
    for i := (t-s) div 2 - 1 downto 0 do begin
      X( s-2, t-2-2*i, p, +dirp );
      Move( s-1, left );
      X( s-2, t-1-2*i, p, -dirp );
      Move( s-1, left );
    end;
    Z( s-2, t, back, p, +dirp );
  end;
end {of Z};


procedure X;
var i,j : integer;
```

```
begin
   if s = 0 then Move( p, dirp ) else
   begin
      A( s-2, s-2, back, s-1, riht );
      for i := 1 to (t-s) div 2 do begin
         Move( s, left );
         A( s-2, s-2, forw, s-1, left );
         Move( s, left );
         A( s-2, s-2, back, s-1, riht );
      end;
      if (t-s) mod 2 = 1 then begin
         Move( s, left );
         for i := 0 to (t-s) div 2 do AM( s, s-2+2*i, forw );
         Y( s-2, t-1, p, +dirp );
         for i := (t-s) div 2 downto 0 do MA( s, s-2+2*i, back );
         Move( s, riht );
      end else begin
         Move( p, +dirp );
      end;
      for i := (t-s) div 2 downto 1 do begin
         A( s-2, s-2, forw, s-1, left );
         for j := 1 to i do MA( s, s-2+2*j, forw );
         Move( p, -dirp );
         for j := i downto 2 do AM( s, s-2+2*j, back );
         A( s-2, s, back, s-1, riht );
         Move( s, riht );
         for j := 1 to i-1 do AM( s, s-2+2*j, forw );
         Y( s-2, s-2+2*i, p, +dirp );
         for j := i-1 downto 0 do MA( s, s-2+2*j, back );
         Move( s, riht );
      end;
      A( s-2, s-2, forw, s-1, left );
   end;
end {of X};


procedure Y;
var i,j : integer;
begin
   if s = 0 then Move( p, dirp ) else
   begin
      A( s-2, t-2, back, s-1, riht );
      for i := 1 to (t-s) div 2 do MA( s, t-2-2*i, back );
      Move( s, left );
```

```
      for i := 0 to (t-s) div 2 do AM( s, s-2+2*i, forw );
      Y( s-2, t, p, +dirp );
      for i := (t-s) div 2 downto 0 do MA( s, s-2+2*i, back );
      Move( s, riht );
      for i := 1 to (t-s) div 2 do begin
         A( s-2, s-2, forw, s-1, left );
         Move( s, riht );
         A( s-2, s-2, back, s-1, riht );
         Move( s, riht );
      end;
      A( s-2, s-2, forw, s-1, left );
      Move( p, -dirp );
      A( s-2, s-2, back, s-1, riht );
      Move( s, left );
      A( s-2, s-2, forw, s-1, left );
      Move( s-1, left );
      Y( s-2, s, p, +dirp );
      Move( s, left );
      A( s-2, s, forw, s-1, left );
      for i := 2 to (t-s) div 2 do begin
         Move( p, -dirp );
         for j := i-1 downto 1 do AM( s, s-2+2*j, back );
         A( s-2, s-2, back, s-1, riht );
         Move( s, left );
         for j := 0 to i-1 do AM( s, s-2+2*j, forw );
         Y( s-2, s-2+2*i, p, +dirp );
         for j := i-1 downto 1 do MA( s, s-2+2*j, back );
         Move( s, left );
         A( s-2, s, forw, s-1, left );
         for j := 1 to i-1 do MA( s, s+2*j, forw );
      end;
   end;
end {of Y};


procedure LeftBinaryTrees( n, k : integer );
var i : integer;
begin
  {Initialize}
  for i := 1 to 2*n do x[i] := 0;
  for i := 1 to k do x[i] := 1;
  x[1+2*k] := 1;  y[n-k] := 1+2*k;
  for i := 1 to n-k-1 do begin
      x[2+i+2*k] := 1;  y[n-k-i] := 2+i+2*k;
```

```
  end;
  {Connect together "A" type graphs}
  A( n-k-1, n-k-1, forw, n-k, left );
  for i := 2 to k div 2 do begin
      Move( n-k, left );
      A( n-k-1, n-k+(i-1)*2-1, forw, n-k, left );
  end;
  WriteBinaryTree;
end {of LeftBinaryTrees};


procedure BinaryTrees( n : integer );
var i : integer;
begin
  {Initialize}
  for i := 1 to 2*n do x[i] := 0;
  x[1] := 1;  y[n]   := 1;
  x[3] := 1;  y[n-1] := 3;
  for i := 1 to n-2 do begin
      x[4+i] := 1;  y[n-1-i] := 4+i;
  end;
  {Connect together "A" type graphs}
  A( n-2, n-2, forw, n-1, left );
  WriteBinaryTree;
end {of BinaryTrees};


begin
   write( 'Enter n and k (k=0 for all trees): ' );
   readln( n, k );
   if k <> 0 then begin
     if n mod 2 = 0 then writeln( 'n must be odd' ) else
     if k mod 2 = 1 then writeln( 'k must be even' ) else
     LeftBinaryTrees( n, k );
   end else begin
     if n mod 2 = 1 then writeln( 'n must be even' ) else
     BinaryTrees( n );
   end;
end {of GenerateBinaryTrees}.
```