An alternative method and implementation for the generation of a Hamiltonian path through the binary neighbour-swap graph

van Heck, I.

Award date:
2016

An alternative method and implementation for the generation of a Hamiltonian path through the binary neighbour-swap graph

2015-2016

Ivo van Heck
0775551

2015 - 2016

Contents

1 Introduction

In discrete mathematics, combinatorial generation is an often discussed topic. This branch of mathematics concerns itself with the generation of combinatorial objects. Within combinatorial generation there exists a subtopic known as minimal change generation. This subtopic concerns itself with generating all combinations of mathematical objects in such an order that consecutive combinations resemble each other in some way.

The most famous example of minimal change generation is known as the Gray code [3]. The Gray code is a way of ordering binary numbers in such a way that consecutive numbers differ in only bit. Another well-known example of minimal change generation are de Bruijn cycles [1]. These de Bruijn cycles provide an ordering of binary numbers such that any number can be created by removing the least-significant bit and adding a most-significant bit to the previous number.

The combinatorial objects central to this paper are permutations of multisets. In particular this paper concerns itself with a minimal change generation of such permutations. The specific case to which this paper gives a construction is the case where the multiset contains only two unique elements and the minimal change is the swapping of two consecutive elements in the permutation. This operation will be referred to as a neighbour swap.

Research into this, and similar, topics dates back quite far. In the seventeenth century the problem of minimal change generation using sets instead of multisets was solved to find a method for change ringing. Change ringing is a method of ringing a number of (church)bells in mathematical patterns. Perhaps the most famous paper written on this topic is [5] by D. H. Lehmer. The problem which this paper concerns was first solved in a paper published in 1984 [2]. Since then, both papers focussed on applications and implementations of suggested algorithms [4] and papers of more recreational [7] and mathematical nature [6] have been published.

The idea behind the algorithm described in this paper was first suggested by Cor Hurkens in personal correspondence with the project mentor, Tom Verhoeff. The primary advantage of this method of construction is the more transparent nature of the recursion. Hopefully this new solution strategy will lead to new insights in the generalised case of generating permutations of any multiset.

2 Problem definition

Before we can describe the problem accurately, we need some other definitions. The first definition we will discuss is a permutation, after which we will define a specific relation on these permutations. Then we will describe two ways of modelling the problem.

A **permutation** of a multiset S containing k_0 0's and k_1 1's is any finite sequence containing exactly k_0 0's and k_1 1's. The set of all permutations of S is denoted by $\sigma(S)$, however the shorthand $\sigma(k_0, k_1)$ will also be used. Note that $|\sigma(k_0, k_1)| = \binom{k_0+k_1}{k_0}$ since this is exactly the number of sequences with k_0 0's placed in $k_0 + k_1$ positions (and the rest of the position being 1).

Now that $\sigma(k_0, k_1)$ has been defined, we can also define the relation $\sim$ on it. This relation, which we will name the **neighbour swap relation**, holds if and only if the two permutations differ only by the order of two adjacent elements. More formally:

$$s \sim s' \Leftrightarrow \exists i : s_i = s'_{i+1} \wedge s_{i+1} = s'_i \wedge \forall j \neq i, i+1 : \; s_j = s'_j \tag{1}$$

Note that this relation is symmetric but not transitive. The symmetry is quite clear from the definition since if the $s \sim s'$ property holds for a fixed i, then the $s' \sim s$ property must hold for $i - 1$ as these properties are identical.

Now that we have these definitions, we can start discussing the problem. In the first way of modelling the problem the task is to find an ordering of $\sigma(k_0, k_1)$ such that any two consecutive permutations in the ordering satisfy the neighbour swap relation. Note that such an ordering isn't guaranteed to exist for every all parameters. In fact, it has been previously proven to exist if and only if k_0 and k_1 are both odd or either $k_0 = 1$ or $k_1 = 1$.[2]

The second way to formulate the problem is by using graphs. An advantage of this model is that it explicitly visualises the problem. First we construct an undirected unweighed graph by defining a set of vertices and a relation on these vertices. We choose the set of vertices to be $\sigma(k_0, k_1)$ and the neighbour swap relation as edges.

By defining the graph this way, our original problem translates to finding a path through the graph such that every vertex is visited exactly once. Such a path is called a Hamiltonian Path.

Note that these 2 models represent the same problem. We can prove this by showing that a solution to either model can be transformed into a solution to the other. However, as these two transformations are extremely similar, we will only describe one of them.

Clearly, if we have found an ordering on $\sigma(k_0, k_1)$ such that the neighbour swap relation holds on each consecutive two permutations, these permutations must also form a path through the graph. Since the ordering of $\sigma(k_0, k_1)$ must contain each element of exactly once, this path must also contain each vertex exactly once. Thus this path is a Hamiltonian Path.

3 Construction and proof of correctness

In this section we will discuss a possible way of constructing the solution to the problem. First we will look into the assumptions we make in order to solve this, followed by a description of the proposed construction. We will prove the correctness of the proposed construction along the way. Finally we will look at an example problem and how the solution would be constructed.

As a quick reminder, we are trying to find a way to construct a path p with the following two properties:

1. p contains every element from $\sigma(k_0, k_1)$ exactly once and nothing else.
2. Any two consecutive permutations in p are related (in the sense of $\sim$).

The input of the algorithm will be two integer parameters, k_0 and k_1. These are the parameters describing the problem as defined in §2. Our first assumption is that both k_0 and k_1 are odd, since the problem is unsolvable otherwise (as discussed earlier).

In order to solve this problem we have to construct a Hamiltonian path satisfying the conditions explained in §2. Finding such a path is fairly straightforward if either $k_0 = 1$ or $k_1 = 1$, so in that case we can directly output $(01^{k_1}, 101^{k_1-1}, ..., 1^{k_1-1}01, 1^{k_1}0)$ or the equivalent for $k_1 = 1$.

If however both $k_0, k_1 > 1$, we need a more complicated strategy. Before we can discuss this strategy though, we will to look at some families of subsets of $\sigma(k_0, k_1)$ that will help defining this strategy.

3.1 Defining subsets

Before we will look at some families of subsets of $\sigma(k_0, k_1)$, we have to introduce some operators.

In the definition as well as later in the paper we will use the $\uparrow$ (pronounced as "take") and the $\downarrow$ (pronounced as "drop) operators. Both these operators operate on lists. Since permutations are essentially lists of 0's and 1's, we can use these operators on permutations. For an integer i and a permutation s, $s \uparrow i$ is the list containing the first i elements of s. Thus $s \uparrow i$ is itself a permutation, however for (possibly) different k_0, k_1. $s \downarrow i$ is a list containing all elements of s except the first i. Because of this $(s \uparrow i).(s \downarrow i) = s$ for any i. Note that we will use a . for concatenation.

With all the operators we need defined, we can define some families of subsets on $\sigma(k_0, k_1)$. The first of these is the following family of subsets:

$$Q_i := \{s \in \sigma(k_0, k_1) \mid s \uparrow i \text{ contains at least two 0's}\} \tag{2}$$

Since Q_i contains only elements from $\sigma(k_0, k_1)$, it is also implicitly dependent on k_0 and k_1.

Note that Q_i has some interesting properties, some of which are worth pointing out. One of these properties is the fact that Q_i is an increasing[1] sequence of sets. Another is the fact that $Q_{2+k_1} = \sigma(k_0, k_1)$, since the first $k_1 + 2$ elements

[1] The definition of increasing sequences we use is in the sense of set containment. Thus a sequence X of sets is increasing if and only if $\forall i < j : X_i \subseteq X_j$

of any permutation must contain at least two 0's. From these two properties we can deduce that $Q_{2+k_1} = Q_{3+k_1} = ... = Q_{k_0+k_1} = \sigma(k_0, k_1)$.

A final property that's worth looking at is the nature of Q_0 and Q_1. Since any prefix of length zero or one cannot contain two 0's, these must always be empty. Thus the first non-empty set in this family is Q_2.

Now with these properties in mind, we can take a look at an example. Let us consider the Q_i's for $\sigma(3, 3)$. Q_2 contains all the permutations starting with two 0's (and no other permutations), so:

$$Q_2 = \{000111, 001011, 001101, 001110\}$$

Since Q_i is an increasing sequence, we can write Q_3 :

$$Q_3 = Q_2 \cup \{010001, 010010, 010100, 100011, 100101, 100110\}$$

Here you can see how each permutation in Q_3 that is not contained in Q_2 has its second 0 exactly at position 3. You can see this pattern continues to hold:

$$Q_4 = Q_3 \cup \{011001, 011010, 101001, 101010, 110001, 110010\}$$
$$Q_5 = Q_4 \cup \{011100, 101100, 110100, 111000\}$$
$$Q_6 = Q_5$$

Since Q_i is an increasing sequence, we create a partition by defining $P_i := Q_i \setminus Q_{i-1}$ for all i such that Q_i exists and is non-empty. Note that we do no require Q_{i-1} to be non-empty. As such, P_2 is included in this definition and is always equal to Q_2 (since, as mentioned earlier, Q_1 is always empty). In de example given above the P_i's are the sets given explicitly by their elements. As mentioned in the example, each P_i contains exactly those permutations with their second 0 exactly at position i. Since $Q_{2+k_1} = Q_{3+k_1} = ... = Q_{k_0+k_1}$ the last non-empty part is P_{2+k_1}.

With these definitions, we are ready to discuss the general strategy.

3.2 Solution strategy

Since we have seen that $Q_{k_0+k_1} = \sigma(k_0, k_1)$, finding a path on an arbitrary Q_i is sufficient to solve the original problem. The advantage of formulating a solution in terms of Q_i is that we can use mathematical induction. That is, if we have a path on Q_i and we can find a path on P_{i+1} such that it can be connected to the original path, we have found a path on Q_{i+1}.

However, rather than using the bipartition of Q_{i+1} as Q_i and P_{i+1}, we will bipartition it as Q_{i-1} and $P_i \cup P_{i+1}$. This is done because the construction of the path through P_i will differ significantly for even and odd i's respectively. By bundling P_i and P_{i+1} we can formulate the construction of the path based on having both an even and odd part. i will always be even, since the first part is P_2 and we use increments of two.

In order to ensure the paths on Q_{i-1} and the path to be constructed on $P_i \cup P_{i+1}$ can be connected, we have to impose a constraint on the path on Q_{i-1}. The constraint we will use is that the end of the path on Q_{i-1} must be neighbouring the lexicographically smallest element of P_i. With this constraint we can start

our path with the lexicographical smallest element and be ensured that it can be connected to the other path. Since we are constructing a path on $P_i \cup P_{i+1}$, this constraint means the path we construct must end on a permutation neighbouring the lexicographically smallest permutation of Q_{i+2}, if it is non-empty. A more formal definition can be found in §3.4.

3.3 Constructing a path on $P_i \cup P_{i+1}$

Since the first P_i will be P_2 for any problem, we will assume that i is even. We will first construct the path on P_i and then the path on P_{i+1}.

3.3.1 Constructing a path on P_i

Since we are constructing a path on P_i, we know each permutation will have its second 0 at position i. Since i is even, we can say that for any $s \in P_i$, $s \uparrow i$ contains an even number of 0's (since it contains exactly two 0's) and an even number of 1's. Because the total number of 0's and 1's are both odd, $s \downarrow i$ must also contain an odd number of 0's and 1's.

We will now generate a path on the permutations of $s \uparrow (i-1)$ and $s \downarrow i$ separately and combine these into a path on P_i. Since $s \uparrow (i-1)$ contains only one 0, finding a path on these permutations is trivial since it is the solution to the problem with $k_0 = 1$ and $k_1 = i - 2$. Let us denote this sequence by h and its length by n. Note that $n = \binom{i-1}{1} = i - 1$. Thus n is odd. We choose the order of this path such that the first permutation is the lexicographically smallest permutation. We can always do this since the path will always start and end in the lexicographically smallest and largest element.

Generating a path on the permutations of $s \downarrow i$ can be done through recursion since $s \downarrow i$ also contains an odd number of 0's and 1's. The number of 0's is strictly lower and the number of 1's is at most the same. A formal proof of this using induction will be given later. For now let us assume this gives us a path t of all permutations of $s \downarrow i$ such that it satisfies $\sim$. Let m be the length of t. We can deduce that m must be even, because every permutation has a distinct reversed permutation[2] (that is, no permutation is its own reverse). Similar to h, we choose the start and end point of t to be the lexicographically smallest and largest permutation respectively.

Now we need to combine h and t in such a way that we can find a path through P_i. We do this by creating a matrix:

$$M_i = \begin{pmatrix} h_1.0.t_1 & h_1.0.t_2 & \cdots & h_1.0.t_m \\ h_2.0.t_1 & h_2.0.t_2 & \cdots & h_2.0.t_m \\ \vdots & \vdots & \ddots & \vdots \\ h_n.0.t_1 & h_n.0.t_2 & \cdots & h_n.0.t_m \end{pmatrix}$$

Because both t and h satisfy $\sim$, any horizontally or vertically adjacent permutations in M_i also satisfy $\sim$. Now we can traverse the matrix column-first, which

[2]Since any such permutation s has an odd number of both 0's and 1's, it cannot contain the same number of 0's and 1's in its two halves. Thus, when reversed it cannot be the same permutation since second half of s is the first of its reversed and thus has a different number of 0's and 1's

gives us the path:

$$\{h_1.0.t_1, h_2.0.t_1, ..., h_n.0.t1, h_n.0.t_2, h_{n-1}.0.t_2, ..., h_1.0.t_2, h_1.0.t_3, ..., h_1.0.t_m\}$$

Note that we know that the path will end at $h_1.0.t_m$ because m is even. Thus, when constructing a path on P_{i+1} we will need to ensure that its starting permutation is neighbouring $h_1.0.t_m$. Note that because we chose the start and end point of h and t respectively, we know this permutation to be $01^{i-2}01^{k_1-i+2}0^{k_0-2}$.

3.3.2 Constructing a path on P_{i+1}

To construct a path on P_{i+1} we will try to use a strategy similar to the one used to find a path on P_i. However, since $i+1$ is odd, we will have some problems using that strategy. That is, if we define similar t and h, we cannot find h recursively since the permutations will not contain an odd number of 0's and 1's.

To avoid this problem, we will generate the permutations with a 0 in the $(i+2)$nd position separately from the permutations with a 1 in that position. Let us first consider the permutations with a 1 at this position. In either case, we have an odd number of 1's and a single 0 in $s \uparrow (i-1)$. This means that $s \downarrow i$ contains both an even number of 0's and 1's. This combined with $s_{i+1} = 0$ and $s_{i+2} = 1$ gives us that the number of 0's and 1's in $s \downarrow (i+2)$ must both be odd.

This gives us the same properties as we used in the construction for the path on P_i, and as such we can create similar h and t in the same way. This time however, we choose the start and end points of t to be reversed, thus the start point being the lexicographically largest permutation and the end point the lexicographically smallest permutation. This again gives us a similar matrix:

$$M_{i+1} = \begin{pmatrix} h_1.01.t_1 & h_1.01.t_2 & \cdots & h_1.01.t_m \\ h_2.01.t_1 & h_2.01.t_2 & \cdots & h_2.01.t_m \\ \vdots & \vdots & \ddots & \vdots \\ h_n.01.t_1 & h_n.01.t_2 & \cdots & h_n.01.t_m \end{pmatrix}$$

We can construct a path through M_{i+1} in the same way as we did before through M_i. This gives us the path:

$$\{h_1.01.t_1, h_2.01.t_1, ..., h_n.01.t1, h_n.01.t_2, h_{n-1}.01.t_2, ..., h_1.01.t_2, h_1.01.t_3, ..., h_1.01.t_m\}$$

Now let us consider the permutations in P_{i+1} with a 0 in the $(i+2)$nd position. In this case, we will again look at two cases separately. The first case being the case in which $i = k_1 + 1$ and the second where $i < k_1 + 1$. Note that if $i > k_1 + 1$, P_{i+1} must be empty thus we don't need to concern ourselves with this case.

If $i = k_1 + 1$, all the 1's are contained in the first $i+1$ elements. Thus, there are no permutations in the path defined above (since there exist no permutations with all the 1's in the first $i+1$ positions and a 1 in the $i+2$nd position). However, we can still use the h as defined above. Since for any permutation s

in this category $s \downarrow i+1$ is the same (namely the sequence only containing 0's), we can use the path:

$$\{h_1.0^{k_0-1}, h_2.0^{k_0-1}, h_3.0^{k_0-1}, ..., h_n.0^{k_0-1}\}$$

It might seem problematic that the end of this sequence does not satisfy the extra constraint we need to ensure it can be connected to the next part of the path, however in this case the path is finished since P_{k_1+3} must be empty.

Now let us consider the second case, in which $i < k_1 + 1$. In this case, a permutation s in this category must have have at least one 1 in $(s \downarrow i+2)$. Thus, every such permutation must also have a leftmost 1 in $(s \downarrow i+2)$. Since the leftmost 1 can be swapped to position $i+1$ with some number of swaps, we have a path connecting these permutations to the path we found through M_{i+1}. Since these same permutations exist for each h_j, we can glue these into the path by replacing the edge $(h_j.01.t_k, h_{j=1}.01.t_k)$ with the path:

$$\{h_j.01.t_k, h_j.001.(t_k \downarrow 1), h_j.0001.(t_k \downarrow 2), ..., h_j.0^{m+1}1.(t_k \downarrow m),$$
$$h_{j+1}.0^{m+1}1.(t_k \downarrow m), ..., h_{j+1}.0001.(t_k \downarrow 2), h_{j+1}.001.(t_k \downarrow 1), h_{j+1}.01.t_k\}$$

Here m is chosen such that $m+1$ is the index of the second 1 in t_k. By doing this for each such edge, we have also included all permutations in this category. Since this was the final category, we have now found a path on P_{i+1}.

Now in order to construct the path on $P_i \cup P_{i+1}$, we have to combine these two paths. Because we chose to reverse the order of the sequence t in the path on P_{i+1} however, we have the property that the last permutation on our path on P_i is neighbouring our first permutation on the path of P_{i+1}. We can see this more clearly by writing out these permutation explicitly. As described above, the last permutation of the path on P_i is $01^{i-2}01^{k_1-i+2}0^{k_0-2}$. By similar logic, we can deduce that the first permutation of our path on P_{i+1} is $01^{i-1}01^{k_1-i+1}0^{k_0-2}$. These two permutation are clearly neighbours, thus we can connect our paths.

The last thing we have to check is whether this path satisfies the constraint imposed on the endpoint of the path. Again, we can see this more clearly by explicitly writing out the final permutation of this path. This permutation is $01^{i-1}010^{k_0-2}1^{k_1-i+1}$. Assuming P_{i+2} is non-empty, this permutation is indeed neighbouring the lexicographically smallest permutation of this part, $01^i0^{k_0-1}1^{k_1-i+1}$.

3.4 Formal Proof

In order to prove the construction described above more formally, we first have to define more precisely what it is we will be proving. The following theorem contains exactly what we will prove:

For all k_0, k_1, i such that k_0, k_1 odd and $1 < i < k_0 + k_1$, there exists a path p through Q_i of $\sigma(k_0, k_1)$ such that the following properties hold:

1. p contains all elements of Q_i exactly once.
2. $p_j \sim p_{j+1}$ holds for any j (such that both p_j and p_{j+1} exist).
3. If i is odd and P_{i+1} is non-empty with lexicographically smallest permutation s, $p_n \sim s$ holds (with n the length of p).

In order to prove this, we will use mathematical induction. Thus, we have to prove the validity of this theorem in certain base cases and the validity of this theorem for values of k_0, k_1 and i assuming the theorem holds for smaller values of these parameters. More precisely, the theorem is assumed to hold for the values all k_0', k_1' and i' such that one of the following three properties holds:

- $k_0' < k_0$ and $k_1' \leq k_1$ for any i'
- $k_0' \leq k_0$ and $k_1' < k_1$ for any i'
- $k_0' = k_0$, $k_1' = k_1$ and $i' < i$.

Proof. In the cases of $k_0 = 1$ or $k_1 = 1$, we have already shown a solution at the start of §3. In the case of $i = 2$, the construction described in §3.3.1 works to find a path on Q_2 of any $\sigma(k_0, k_1)$ that satisfies the theorem. In case of $i = 3$ we can use the construction described in §3.3 to find a path on Q_3 that satisfies all conditions.

Now let us assume this theorem holds for all values smaller than k_0, k_1 and i. With this assumption, the recursion described in §3.3 can be replaced with the path assumed to exist because of the induction hypothesis. Thus, if i is even we can use the construction of §3.3.1 to find a path on P_i and concatenate it to the path we know exists for Q_{i-1} to find a path on Q_i. If i is odd, we use the construction of §3.3 to find a path on $P_{i-1} \cup P_i$ and concatenate it to a path on Q_{i-2}. Note that we know the second condition holds at the point of concatenation because of the third condition. Thus we have a path on Q_i of $\sigma(k_0, k_1)$ satisfying the theorem. □

3.5 Example

In this section we will look at an example of how this construction works. The problem we will be using as an example is $\sigma(3, 3)$. This is the same as we used as an example in §3.1.

The first step is finding a path on $P_2 \cup P_3$ as described in §3.3. By using the strategy described in §3.3.1, we find $h = \{0\}$. Now we will find t through recursion with parameters $k_0 = 1, k_1 = 3$. Since this is the base case of the recursion, we know $t = \{0111, 1011, 1101, 1110\}$. This gives us:

$$M_2 = \begin{pmatrix} 0.0.0111 & 0.0.1011 & 0.0.1101 & 0.0.1110 \end{pmatrix}$$

Since this matrix is 1 by 4, the path through it is fairly straightforward:

$$\{000111, 001011, 001101, 001110\}$$

Now we can construct M_3 as described in §3.3.2. Here we find $h = \{01, 10\}$ and (through recursion to the case $k_0 = 1, k_1 = 1$) $t = \{10, 01\}$. Notice how the order of t is such that it starts with the lexicographically largest element. From these sequences h and t we construct M_3:

$$M_3 = \begin{pmatrix} 01.01.10 & 01.01.01 \\ 10.01.10 & 10.01.01 \end{pmatrix}$$

This would give us the path $\{010110, 100110, 100101, 010101\}$, however as discussed in §3.3.2 we still need to insert the permutations of the form $h.00.r$. After inserting these permutations at all relevant positions of the path we get:

$$\{010110, 100110, 100101, 100011, 010011, 010101\}$$

Thus our path on $Q_3 = P_2 \cup P_3$ is:

$$\{000111, 001011, 001101, 001110, 010110, 100110, 100101, 100011, 010011, 010101\}$$

Now for the second step we use the same strategy to find a path on $P_4 \cup P_5$. Again, we first construct the matrix M_4 by construction h and t. In this case we find $h = \{011, 101, 110\}$ and $t = \{01, 10\}$. This gives us:

$$M_4 = \begin{pmatrix} 011.0.01 & 011.0.10 \\ 101.0.01 & 101.0.10 \\ 110.0.01 & 110.0.10 \end{pmatrix}$$

From this matrix we find the path:

$$\{011001, 101001, 110001, 110010, 101010, 011010\}$$

For P_5 we find that the matrix M_5 is empty, since there are no permutations of the form $h.01.t$. However, we still use $h = \{0111, 1011, 1101, 1110\}$ to construct the path through P_5 as described in §3.3.2:

$$\{011100, 101100, 110100, 111000\}$$

Now by appending all the paths we find the path p on $\sigma(3,3) = Q_5$:

$$\begin{aligned} p = \{&000111, 001011, 001101, 001110, 010110, 100110, 100101, \\ &100011, 010011, 010101, 011001, 101001, 110001, 110010, \\ &101010, 011010, 011100, 101100, 110100, 111000\} \end{aligned}$$

4 Implementation

In this section we will discuss an implementation of the algorithm described. This implementation is written in Mathematica and is focussed on code clarity rather than efficiency. The complete code can be found in appendix A, however most of the code can be found in parts in this section along with the explanation. Non-essential code such as tests will not be discussed here though.

The code can be split into three general categories, each with their own subsection here. The categories are:

- Elementary operations
- Queries
- Generation of the Hamiltonian path

Before we discuss these in their respective sections, however, we will discuss how we represent the concepts used in the algorithm in Mathematica data-types. The first mathematical object which we will discuss is the permutation.

In our implementation a permutation is a list[3] of integers. More specifically, permutations in $\sigma(k_0, k_1)$ will be lists containing k_0 0's and k_1 1's. Paths are represented as lists of permutations. Therefore a path is essentially a list of lists of integers.

Since the algorithm is essentially concerned with paths on sets like $\sigma(k_0, k_1)$ and Q_i's and never the sets themselves, they do not have a representation in the code.

4.1 Elementary operations

The functions defined here will mostly be generic functions that are not specific to our algorithm. These functions can generally be understood even without knowledge of the problem, the rest of the algorithm or the strategy it uses. The order in which we discuss these functions is the order in which they are defined in the code. The first function we will discuss is `swap`.

4.1.1 swap

The purpose of the function `swap` is to swap two elements of a list. As such, it requires a list and the indices of the elements which should be swapped as parameters. It is assumed the index parameters are valid for the given list. If the two indices are the same, the list that is returned is the same as the input list. This is the function definition:

```
swap[list_List, {index1_Integer, index2_Integer}] :=
 ReplacePart[list,
   {index1 -> list[[index2]], index2 -> list[[index1]]}]
```

[3] For those unfamiliar with the Mathematica syntax, a list is shown as a sequence of elements between the { } symbols and separated by comma's. Thus, the list with the integers 1, 2 and 3 in that order would be $\{1, 2, 3\}$.

4.1.2 neighbourSwap

This function is a specific case of the previous function and simply swaps an element from a given list with the next element in the list. This connection to swap is also visible in its function definition:

```
neighbourSwap[list_List, index_Integer] :=
 swap[list, {index, index + 1}]
```

4.1.3 generateMultiset

The function `generateMultiset` generates a list of elements each repeated a specified number of times. This function could be considered the inverse of the Mathematica function `Tally`, with loss of order. The input of this function is a list of element-integer pairs, where each element is to be repeated the integer number of times. The function definition is:

```
generateMultiset[tally_List] :=
 Flatten[Table[
   ConstantArray[First[tally[[i]]], Last[tally[[i]]]],
   {i, Length[tally]}], 1]
```

Since the input and output format of this function might be confusing, we will also consider an example of this function. If we would want to create a list of three `a`'s, one `b` and four `c`'s, we could use the following:

```
generateMultiset[{{a, 3}, {b, 1}, {c, 4}}]
```

The output this would generate is:

```
{a, a, a, b, c, c, c, c}
```

4.1.4 generateIntegerMultiset

This function is a specific case of the the previous function, `generateMultiset`, where the elements to be repeated are simply the integers $0, 1, 2, ..., n$. This could be used to generate a permutation in $\sigma(k_0, k_1)$ from the parameters k_0, k_1. This function simply uses the `generateMultiset` function, as can be seen in the function definition:

```
generateIntegerMultiset[count_List] :=
 generateMultiset[Table[{i - 1, count[[i]]},
  {i, 1, Length[count]}]]
```

4.2 Queries

The queries are functions which determine whether some condition is met. All functions in this section will return a boolean. These functions are mostly used for testing of the other functions. The names of these functions follow the Mathematica naming convention of ending in a capital Q.

4.2.1 permutationPathQ

This function determines whether a path contains all permutations for a given k_0 and k_1. Note that this implementation is quite inefficient, but since it's only used in testing this isn't particularly important.

```
permutationPathQ[path_List, count_List ] :=
 Sort[path] ===
   Sort[Permutations[generateIntegerMultiset[count]]]
```

4.2.2 neighbourQ

This function determines whether the neighbour-swap relation holds for two given permutations. It calculates the positions where the two lists differ, and checks whether there are only 2 positions where this is the case. If so, it also checks whether these positions are consecutive.

```
neighbourQ[{v1_List, v2_List}] :=
  With[{diff = Flatten[Position[v1 - v2, _?(# != 0 &)], 1]},
   Length[diff] == 2 && diff[[1]] + 1 == diff[[2]] ];
```

4.2.3 neighbourPathQ

This function determines whether the neighbour-swap relation holds for any two consecutive permutations.

```
neighbourPathQ[path_List] :=
 And @@ Map[neighbourQ[#1] &, Partition[path, 2, 1]]
```

4.2.4 hamiltonianNeighbourPathQ

This function determines whether a given path represents a Hamiltonian path in the neighbour-swap graph. It is equivalent to both being a neighbourPath and a permutationPath. Any path which satisfies this query is a solution to the initial problem.

```
hamiltonianNeighbourPathQ[path_,
  d : {numberOfZeroes_, numberOfOnes_}] :=
 neighbourPathQ[path] && permutationPathQ[path, d]
```

4.2.5 listStartsQ

This function determines whether a given list is a prefix of another. That is, whether there exist an i such that $s \uparrow i = t$ for parameters s, t.

```
listStartsQ[list_List, prefix_List] :=
 prefix === Take[list, Length[prefix]]
```

4.2.6 permutationPathGivenPrefixQ

This function determines whether a path contains all permutations starting with a given part.

```
permutationPathGivenPrefixQ[path_List, head_List,
 {numberOfZeroes_Integer, numberOfOnes_Integer} ] :=
  With[{perms = Permutations[
   generateIntegerMultiset[{numberOfZeroes, numberOfOnes}]]}
   ,
   Sort[Pick[perms, listStartsQ[#, head] & /@ perms]]
```

```
    === Sort[path]
  ]
```

4.3 Generation of the Hamiltonian path

This section contains the primary function, `neighbourPath`, which takes the parameters k_0 and k_1, and returns a path which satisfies the `hamiltonianNeighbourPathQ` query, and a number of auxiliary functions. The `neighbourPath` definition is split into two parts, namely the base case and a recursive definition.

4.3.1 Base case

This function definition handles the solution to the case where $k_0 = 1$ or $k_1 = 1$. This code simply neighbour-swaps at each index in a row, so as to move the singleton through the rest of the list step by step.

```
neighbourPath[count_List /;
 Length[count] === 2 && Or @@ Map[# == 1 &, count]] :=
 Block[
  {index = First[Flatten[Position[count, 1]]],
   v = generateIntegerMultiset[count]
   },
  If[index == 2,
   Reverse[{v = Reverse[v]}~Join~
     Table[v = neighbourSwap[v, i], {i, Length[v] - 1}]],
   {v}~Join~
     Table[v = neighbourSwap[v, i], {i, Length[v] - 1}]
   ]
  ]
```

4.3.2 Recursive case

The recursive case will be using some auxiliary functions which will be defined after this. However, the titles of these functions should be enough to understand how this code works. The general strategy is the same as in §3, however rather than solve only on P_i and P_{i+1} and solve the rest through recursion it solves it all at once.

```
neighbourPath[count_List /; Length[count] === 2]
 := Module[
  {vertex = generateIntegerMultiset[count],
   prefix, suffix, prefixPath, suffixPath,
   path = {}, i, cut},
  For[i = 2, i < Last[count] + 2, i++,
   cut = If[OddQ[i], i + 1, i];
   prefix = Take[vertex, cut];
   suffix = Drop[vertex, cut];
   prefixPath = generatePrefixPath[prefix];
   suffixPath =
    If[OddQ[i],
     Reverse[neighbourPath[
```

```
      {Count[suffix, 0], Count[suffix, 1]}]],
     neighbourPath[
      {Count[suffix, 0], Count[suffix, 1]}]];
    path =
    Join[path,
     combinePrefixAndSuffixPaths[prefixPath,
      suffixPath, i]];
   vertex = neighbourSwap[path[[-1]], i];
   ];
  Join[path, generatePrefixPath[vertex]]
  ]
```

4.3.3 generatePrefixPath

This function generates the path on $s \uparrow i$.

```
generatePrefixPath[prefix_List] := Block[
  {list = prefix,
   end = Flatten[Position[prefix, 0]][[2]] - 2},
  Join[{prefix},
   Table[list = neighbourSwap[list, i], {i, end}]]
  ]
```

4.3.4 combinePrefixAndSuffixPaths

This function combines the paths on $s \uparrow i$ and $s \downarrow i$ using the matrix argument from §3.3. It uses an auxiliary function for the even and odd cases respectively.

```
combinePrefixAndSuffixPaths[
 prefixPath_, suffixPath_, i_] :=
 If[OddQ[i],
  combinePrefixAndSuffixPathsOdd[prefixPath, suffixPath],
  combinePrefixAndSuffixPathsEven[prefixPath, suffixPath]
 ]
```

4.3.5 cominePrefixAndSuffixPathsEven

This function combines paths on $s \uparrow i$ and $s \downarrow i$ in the case of i even. This is done as described in §3.3.1.

```
combinePrefixAndSuffixPathsEven[
 prefixPath_List, suffixPath_List] :=
 Module[
  {headPathReverse = Reverse[prefixPath]},
  Flatten[
   Table[
    Map[Join[#1, suffixPath[[i]]] &,
     If[OddQ[i],
      prefixPath,
      headPathReverse
     ]
```

```
    ],
     {i, Length[suffixPath]}], 1]
  ]
```

4.3.6 combinePrefixAndSuffixPathsOdd

This function combines paths on $s \uparrow i$ and $s \downarrow i$ in the case of i odd. This code is much more complex than its even counterpart because the problem is harder in the odd case, as is explained in §3.3.2. Because the trick of adding in the parts with middle 00 is something not very natural to a functional programming language, this function definition is programmed much more imperatively. As a result, the code is quite long and complex in Mathematica.

```
combinePrefixAndSuffixPathsOdd[
 headPath_, tailPath_] := Module[
  {path = {},
   subPath = {},
   vertex,
   forward = True,
   headLength,
   tailIterator,
   headIterator,
   i},
  headLength = Length[headPath[[1]]];
  For[tailIterator = 1,
   tailIterator <= Length[tailPath],
   tailIterator++,
   For[
    headIterator =
     If[OddQ[tailIterator], 1, Length[headPath]],
    If[OddQ[tailIterator],
     headIterator <= Length[headPath],
     headIterator >= 1],
     If[OddQ[tailIterator], headIterator++,
      headIterator--],
     (*Iterate over the heads forwards if i is odd,
      and backwards otherwise*)
     vertex = headPath[[headIterator]]
              ~Join~
              tailPath[[tailIterator]];
     If[vertex[[headLength]] == 1 &&
      vertex[[headLength + 1]] == 0,
      subPath = {vertex};
      If[forward,
       For[i = headLength,
        vertex[[i]] != vertex[[i + 1]],
        i++,
         vertex = neighbourSwap[vertex, i];
         AppendTo[subPath, vertex];
        ];
```

```
      path = path~Join~subPath;
      forward = False;
      ,(*else*)
      For[i = headLength,
       vertex[[i]] != vertex[[i + 1]],
       i++,
        vertex = neighbourSwap[vertex, i];
        AppendTo[subPath, vertex];
       ];
      path = path~Join~Reverse[subPath];
      forward = True;
      ];
     ,(*else*)
     AppendTo[path, vertex];
     ];
    ];
 ];
path
]
```

5 Conclusion

In this paper we have shown an alternative construction for a Hamiltonian path through the neighbour-swap graph of $\sigma(k_0, k_1)$ for the case k_0, k_1 both odd. In this construction, we have also ensured that the path can be broken down into pieces with conceptual meaning. That is, the path is ensured to contain all elements of Q_i before the first element of P_{i+1} for all i. Furthermore, if the user is only interested in a path on Q_i rather than $\sigma(k_0, k_1)$, this construction can be used to find such a path even without having to generate a path on the entire graph.

Hopefully this alternative construction of a solution will offer new insights in more general cases. We will give a small introduction of two of these generalised cases.

The first generalisation is the problem of finding Hamiltionian paths on $\sigma(k_0, k_1)$ with the exception of certain permutations for k_0, k_1 not necessarily odd. Interesting to this problem is deciding which permutations not to include and proving that this is the smallest set of permutations to be excluded such that a Hamiltonian path exists.

The second generalisation concerns not permutations of $\sigma(k_0, k_1)$, but the general case $\sigma(k_0, k_1, ..., k_n)$. A solution to this problem has been published already, however it uses another way to model the problem [6]. Hopefully this problem can also be solved by generalising the algorithm described in this paper, to yield an alternative solution to that problem as well.

A Mathematica code

In this appendix the complete Mathematica code is included.

Generating a hamiltionian path through a NeighbourSwapGraph

Elementary Operations

swap

Swaps 2 elements of a list at given positions.

In[1]:=
```
swap[list_List, {index1_Integer, index2_Integer}] :=
 ReplacePart[list, {index1 → list[[index2]], index2 → list[[index1]]}]
```

In[21]:=
```
swap[{1, 3, 17}, {2, 3}] (* Expected: {1,17,3} *)
swap[{1, 2, 3, 4, 5}, {4, 1}] (* Expected: {4,2,3,1,5} *)
swap[{1, 2, 3}, {2, 2}] (* Expected {1,2,3} *)
```

Out[21]=
```
{1, 17, 3}
```

Out[22]=
```
{4, 2, 3, 1, 5}
```

Out[23]=
```
{1, 2, 3}
```

neighbourSwap

Swaps an element of a list at a given position with the next element of that list.

In[2]:=
```
neighbourSwap[list_List, index_Integer] := swap[list, {index, index + 1}]

neighbourSwap[{1, 2, 3}, 2] (* Expected: {1,3,2} *)
{1, 3, 2}
```

generateMultiset

Given a list of pairs. Creates a list of every first element a pair repeated a number of times defined by the second element of that pair with the same order as the pairs are given in.

In[3]:=
```
generateMultiset[tally_List] := Flatten[Table[
   ConstantArray[First[tally[[i]]], Last[tally[[i]]], {i, Length[tally]}], 1]


generateMultiset[{{a, 3}, {b, 3}, {c, 3}}]
generateMultiset[{{4, 1}, {3, 2}, {2, 3}, {1, 4}}]
generateMultiset[{{{}, 3}, {{1, 2, 3}, 1}, {a, 7}}]
{a, a, a, b, b, b, c, c, c}

{4, 3, 3, 2, 2, 2, 1, 1, 1, 1}

{{}, {}, {}, {1, 2, 3}, a, a, a, a, a, a, a}
```

generateIntegerMultiset

Given a list of integers. Generates a list of the first n integers repeated count[[n+1]] times where n is the length of the given list.

```
generateIntegerMultiset[count_List] :=
 generateMultiset[Table[{i - 1, count[[i]]}, {i, 1, Length[count]}]]
```

In[47]:=
```
generateIntegerMultiset[{1, 2}] (* Expected: {0,1,1} *)
generateIntegerMultiset[{3, 3, 2, 2}] (* Expected: {0,0,0,1,1,1,2,2,3,3} *)
```

Out[47]= {0, 1, 1}

Out[48]= {0, 0, 0, 1, 1, 1, 2, 2, 3, 3}

Queries

permutationPathQ

Determines whether the given path contains each vertex of the neighbourgraph, defined by the second argument, exactly once and nothing else.

In[5]:=
```
permutationPathQ[path_List, count_List ] :=
 Sort[path] === Sort[Permutations[generateIntegerMultiset[count]]]

permutationPathQ[{{1, 1, 0}, {0, 1, 1}, {1, 0, 1}}, {1, 2}] (* Expected: True *)
permutationPathQ[{{1, 1, 0, 0}, {1, 0, 1, 0}, {1, 0, 0, 1},
  {0, 1, 1, 0}, {0, 1, 0, 1}, {0, 0, 1, 1}}, {2, 2}] (* Expected: True *)
permutationPathQ[{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}, {0, 0, 1}}, {2, 1}]
(* Expected: False *)
permutationPathQ[{{1, 0, 0}, {0, 0, 1}}, {2, 1}] (* Expected: False *)
permutationPathQ[{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}, {1, 2}] (* Expected: False *)
```

True

True

False

False

False

neighbourQ

Determines for 2 given lists of 0's and 1's, whether they can be obtained from each other through a neighbourswap. This is equivalent to determining whether they are adjacent in the neighbourswap graph.

In[72]:=
```
neighbourQ[{v1_List, v2_List}] :=
  With[{diff = Flatten[Position[v1 - v2, _?(# ≠ 0 &)], 1]},
   Length[diff] == 2 && diff[[1]] + 1 == diff[[2]]];
```

In[73]:=
```
neighbourQ[{{0, 0, 0, 1, 1}, {0, 0, 1, 0, 1}}] (* Expected: True *)
neighbourQ[{{0, 0, 1, 0, 0}, {1, 0, 0, 0, 0}}] (* Expected: False *)
neighbourQ[{{1, 1, 0, 0, 0}, {1, 0, 1, 0, 1}}] (* Expected: False *)
```

Out[73]= True

Out[74]= False

Out[75]= False

neighbourPathQ

Determines whether a given list represents a path through a neighbourswap graph. This is equivalent to saying that for every element of the path (except the last), the next element of the path can be obtained from it through a neighbourswap.

In[7]:=
```
neighbourPathQ[path_List] := And @@ Map[neighbourQ[#1] &, Partition[path, 2, 1]]
```

```
neighbourPathQ[{{1, 0, 0, 0}, {0, 1, 0, 0}, {1, 0, 0, 0}, {0, 1, 0, 0}}]
(* Expected: True *)
neighbourPathQ[{{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}]
(* Expected: True *)
neighbourPathQ[{{1, 0, 0}, {0, 0, 1}, {0, 1, 0}}](* Expected: False *)
```

True

True

False

hamiltonianNeighbourPathQ

Determines whether a given list represents a hamiltonian path through the neighbourswap graph described by the second parameter.

In[8]:=
```
hamiltonianNeighbourPathQ[path_, d : {numberOfZeroes_, numberOfOnes_}] :=
 neighbourPathQ[path] && permutationPathQ[path, d]
```

```
hamiltonianNeighbourPathQ[{{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}, {2, 1}]
(* Expected: True *)
hamiltonianNeighbourPathQ[
 {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}, {0, 0, 1, 0}},
 {3, 1}] (* Expected: False *)
hamiltonianNeighbourPathQ[{{0, 0, 1}, {1, 0, 0}, {0, 1, 0}}, {2, 1}]
(* Excpected: False *)
```

True

False

False

listStartsQ

Determines whether a given list starts with the elements of another given list.

In[9]:=
```
listStartsQ[list_List, prefix_List] := prefix === Take[list, Length[prefix]]
```

In[10]:=
```
listStartsQ[{1, {1, 2, 3}, {2}, b}, {1, {1, 2, 3}, {2}, b}] (*expected: True*)
listStartsQ[{1, {2, 3}}, {1, {2}}](* Expected: False *)
```

Out[10]= True

Out[11]= False

permutationPathGivenPrefixQ

Determines whether a given path contains each vertex of the neighbourgraph that starts with the given head, defined by the second argument, exactly once and nothing else.

```
permutationPathGivenPrefixQ[path_List, head_List,
  {numberOfZeroes_Integer, numberOfOnes_Integer} ] :=
 With[{perms = Permutations[
     generateIntegerMultiset[{numberOfZeroes, numberOfOnes}]]},
  Sort[Pick[perms, listStartsQ[#, head] & /@ perms]] === Sort[path]
 ]


permutationPathGivenPrefixQ[{{0, 0, 1}, {0, 1, 0}}, {0}, {2, 1}]
permutationPathGivenPrefixQ[{{0, 0, 1}, {1, 0, 0}}, {1}, {2, 1}]
```

True

False

Generation of the hamiltonian path in the binary case

Trivial case

Finds the path through the neighbourswap graph if the amount of either 0's or 1's is exactly one.

In[13]:=
```
neighbourPath[
  count_List /; Length[count] === 2 && Or @@ Map[# == 1 &, count]] := Block[
  {index = First[Flatten[Position[count, 1]]],
   v = generateIntegerMultiset[count]
  },
  If[index == 2, Reverse[
    {v = Reverse[v]} ~Join~ Table[v = neighbourSwap[v, i], {i, Length[v] - 1}]],
   {v} ~Join~ Table[v = neighbourSwap[v, i], {i, Length[v] - 1}]
  ]
 ]

neighbourPath[{1, 1}]
hamiltonianNeighbourPathQ[%, {1, 1}]
```

```
{{0, 1}, {1, 0}}
```

```
True
```

```
neighbourPath[{1, 3}]
hamiltonianNeighbourPathQ[%, {1, 3}]
```

```
{{0, 1, 1, 1}, {1, 0, 1, 1}, {1, 1, 0, 1}, {1, 1, 1, 0}}
```

```
True
```

```
neighbourPath[{3, 1}]
hamiltonianNeighbourPathQ[%, {3, 1}]
```

```
{{0, 0, 0, 1}, {0, 0, 1, 0}, {0, 1, 0, 0}, {1, 0, 0, 0}}
```

```
True
```

Recursive case

Finds the path through the neighbourswap graph in other cases with use of recursion.

In[40]:=
```
neighbourPath[count_List /; Length[count] === 2] := Module[
  {vertex = generateIntegerMultiset[count],
   prefix,
   (*The part of vertex of even length containing the first 2 ones. *)
   suffix, (*The part of vertex not contained in vertexHead *)
   prefixPath,
   (*Variable used to store the path through the subgraph of prefix *)
   suffixPath, (*Variable used to store the path
    through the subgraph of suffix *)
   path = {}, (*The eventual awnser to be returned *)
   i,
   cut},
  For[i = 2, i < Last[count] + 2, i++,
   cut = If[OddQ[i], i + 1, i];
   prefix = Take[vertex, cut];
   suffix = Drop[vertex, cut];
   prefixPath = generatePrefixPath[prefix];
   suffixPath =
    If[OddQ[i], Reverse[neighbourPath[{Count[suffix, 0], Count[suffix, 1]}]],
     neighbourPath[{Count[suffix, 0], Count[suffix, 1]}]];
   path = Join[path, combinePrefixAndSuffixPaths[prefixPath, suffixPath, i]];
   vertex = neighbourSwap[path[[-1]], i];
  ];
  Join[path, generatePrefixPath[vertex]]
 ]
```

In[41]:=
```
neighbourPath[{3, 3}];
hamiltonianNeighbourPathQ[%, {3, 3}]
```

Out[42]=
```
True
```

In[34]:=
```
neighbourPath[{3, 5}];
hamiltonianNeighbourPathQ[%, {3, 5}]
```

Out[35]=
```
True
```

In[36]:=
```
neighbourPath[{5, 3}];
hamiltonianNeighbourPathQ[%, {5, 3}]
```

Out[37]=
```
True
```

In[38]:=
```
neighbourPath[{7, 13}];
hamiltonianNeighbourPathQ[%, {7, 13}]
```

Out[39]=
```
True
```

Auxillary functions

Combines the prefix and suffix paths into a path on their concationation in the odd case.

In[15]:=
```
combinePrefixAndSuffixPathsOdd[headPath_, tailPath_] := Module[
  {path = {},
   subPath = {},
   vertex,
   forward = True,
   headLength,
   tailIterator,
    headIterator,
   i},
  headLength = Length[headPath[[1]]];
  For[tailIterator = 1, tailIterator ≤ Length[tailPath],
   tailIterator++, (*Iterate over the tails *)
   For[headIterator = If[OddQ[tailIterator], 1, Length[headPath]],
     If[OddQ[tailIterator], headIterator ≤ Length[headPath], headIterator ≥ 1],
     If[OddQ[tailIterator], headIterator++, headIterator--], (*Iterate
      over the heads forwards if i is odd, and backwards otherwise*)
     vertex = headPath[[headIterator]]~Join~tailPath[[tailIterator]];
     If[vertex[[headLength]] == 1 && vertex[[headLength + 1]] == 0,
      subPath = {vertex};
      If[forward,
       For[i = headLength, vertex[[i]] ≠ vertex[[i + 1]], i++,
        vertex = neighbourSwap[vertex, i];
        AppendTo[subPath, vertex];
       ];
       path = path~Join~subPath;
       forward = False;
       , (*else*)
       For[i = headLength, vertex[[i]] ≠ vertex[[i + 1]], i++,
        vertex = neighbourSwap[vertex, i];
        AppendTo[subPath, vertex];
       ];
       path = path~Join~Reverse[subPath];
       forward = True;
      ];
      , (*else*)
      AppendTo[path, vertex];
     ];
    ];
  ];
  path
 ]
```

Given a list of heads such that isNeighbourPath[headPath] holds and a list of tails such that isHamiltonianNeighbourPath[tailPath, {Count[tailPath,0], Count[tailPath,1]}] holds. Returns a list such that isNeighbourPath holds and for all elements h in headPath it holds that all vertices whose label starts with h are contained.

In[16]:=
```
combinePrefixAndSuffixPathsEven[prefixPath_List, suffixPath_List] := Module[
  {headPathReverse = Reverse[prefixPath]},
  Flatten[Table[Map[Join[#1, suffixPath[[i]]] &,
     If[OddQ[i], prefixPath, headPathReverse]], {i, Length[suffixPath]}], 1]
 ]

combinePrefixAndSuffixPathsEven[{{1, 0, 0, 1}, {0, 1, 0, 1}, {0, 0, 1, 1}},
 {{1, 0}, {0, 1}}](*Expected: {{1,0,0,1,1,0},{0,1,0,1,1,0},
  {0,0,1,1,1,0},{0,0,1,1,0,1},{0,1,0,1,0,1},{1,0,0,1,0,1}}*)
combinePrefixAndSuffixPathsEven[{{1, 1}},
 {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 1, 0}, {0, 0, 0, 1}}]
 (*Expected: {{1,1,1,0,0,0},{1,1,0,1,0,0},{1,1,0,0,1,0},{1,1,0,0,0,1}}*)
```

```
{{1, 0, 0, 1, 1, 0}, {0, 1, 0, 1, 1, 0}, {0, 0, 1, 1, 1, 0},
 {0, 0, 1, 1, 0, 1}, {0, 1, 0, 1, 0, 1}, {1, 0, 0, 1, 0, 1}}
```

```
{{1, 1, 1, 0, 0, 0}, {1, 1, 0, 1, 0, 0}, {1, 1, 0, 0, 1, 0}, {1, 1, 0, 0, 0, 1}}
```

Given a list of 0's and 1's of even length with exactly two 0's, with one of them being the first element of the list. Generates a list permutations of this given list, such that for each permutation (except the last) the following permutation is the same except for the the first 0 being switched with the 1 after the first 0.

In[17]:=
```
generatePrefixPath[prefix_List] := Block[
  {list = prefix,
   end = Flatten[Position[prefix, 0]][[2]] - 2},
  Join[{prefix}, Table[list = neighbourSwap[list, i], {i, end}]]
 ]

generatePrefixPath[{0, 1, 1, 0}]
neighbourPathQ[%]
```

```
{{0, 1, 1, 0}, {1, 0, 1, 0}, {1, 1, 0, 0}}
```

```
True
```

```
generatePrefixPath[{0, 1, 1, 1, 0, 1}]
neighbourPathQ[%]
```

```
{{0, 1, 1, 1, 0, 1}, {1, 0, 1, 1, 0, 1}, {1, 1, 0, 1, 0, 1}, {1, 1, 1, 0, 0, 1}}
```

```
True
```

In[24]:=
```
combinePrefixAndSuffixPaths[prefixPath_, suffixPath_, i_] :=
 If[OddQ[i], combinePrefixAndSuffixPathsOdd[prefixPath, suffixPath],
  combinePrefixAndSuffixPathsEven[prefixPath, suffixPath]]
```

References

[1] de NG Bruijn. A combinatorial problem. *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen. Series A*, 49(7):758, 1946.

[2] Peter Eades, Michael Hickey, and Ronald C Read. Some hamilton paths and a minimal change algorithm. *Journal of the ACM (JACM)*, 31(1):19–29, 1984.

[3] Frank Gray. Pulse code communication, March 17 1953. US Patent 2,632,058.

[4] Tim Hough and Frank Ruskey. *An efficient implementation of the Eades, Hickey, Read adjacent interchange combination generation algorithm*. University of Victoria, Department of Computer Science, 1987.

[5] DH Lehmer. Permutation by adjacent interchanges. *American Mathematical Monthly*, pages 36–46, 1965.

[6] Grzegorz Stachowiak. Hamilton paths in graphs of linear extensions for unions of posets. *SIAM Journal on Discrete Mathematics*, 5(2):199–206, 1992.

[7] Tom Verhoeff. Combinatorial choreography. In Douglas McKenna Robert Bosch and Reza Sarhangi, editors, *Proceedings of Bridges 2012: Mathematics, Music, Art, Architecture, Culture*, pages 607–612, Phoenix, Arizona, 2012. Tessellations Publishing. Available online at `http://archive.bridgesmathart.org/2012/bridges2012-607.html`.