Constant Time and Space Updates for the Sigma-Tau Problem

Zsuzsanna Lipták[1][0000−0002−3233−0691], Francesco Masillo[1][0000−0002−2078−6835], Gonzalo Navarro[2⋆][0000−0002−2286−741X], and Aaron Williams[3][0000−0001−6816−4368]

[1] Department of Computer Science, University of Verona, Italy
{zsuzsanna.liptak, francesco.masillo}@univr.it
[2] CeBiB & Department of Computer Science, University of Chile, Chile
gnavarro@dcc.uchile.cl
[3] Computer Science Department, Williams College, Williamstown, MA, USA
aaron.williams@williams.edu

Abstract. Sawada and Williams in [SODA 2018] and [ACM Trans. Alg. 2020] gave algorithms for constructing Hamiltonian paths and cycles in the Sigma-Tau graph, thereby solving a problem of Nijenhuis and Wilf that had been open for over 40 years. The Sigma-Tau graph is the directed graph whose vertex set consists of all permutations of n, and there is a directed edge from π to π' if π' can be obtained from π either by a cyclic left-shift (sigma) or by exchanging the first two entries (tau). We improve the existing algorithms from $\mathcal{O}(n)$ time per permutation to $\mathcal{O}(1)$ time per permutation. Moreover, our algorithms require only $\mathcal{O}(1)$ extra space. The result is the first combinatorial generation algorithm for n-permutations that is optimal in both time and space, and lists the objects in a Gray code order using only two types of changes. The simple C code (∼50 lines) can be found at https://github.com/fmasillo/sigma-tau.

Keywords: permutations · sigma-tau problem · dynamic data structures · combinatorial generation · combinatorial Gray codes

1 Introduction

The problem of efficiently generating all permutations of $[n] = \{1, 2, \ldots, n\}$ (in one-line notation) is one of the oldest in combinatorial generation. When surveying permutation generation algorithms in 1977, Sedgewick [37] remarked that "It was actually one of the first nontrivial nonnumeric problems to be attacked by computer." Updated surveys on generating combinatorial objects, including permutations, have been written by Savage [33], and more recently by Mütze [27].

Permutations are of fundamental importance in all areas of computer science. In string algorithms, they form the basis of compressed data structures such as compressed suffix arrays [19], compressed suffix trees [15,26,16], and BWT-based data structures, such as the FM-index [13], the RLFM-index [25], the r-index [17], or the extended r-index [5]. Permutations are also of central interest in

⋆ Funded in part by Basal Funds FB0001, ANID, Chile.

computational biology, where they have been used extensively to model genome rearrangements [2,20,3,1,21,12,14,7].

In this paper, we provide iterative permutation generation algorithms that update the current permutation in worst-case $\mathcal{O}(1)$ time (i.e., *loopless*) using $\mathcal{O}(1)$ (additional) space. (We use the transdichotomous RAM model, where a word has $\Theta(\log n)$ bits. So $\mathcal{O}(1)$ space is $\mathcal{O}(\log n)$ total bits; the current permutation's memory is not counted [39].) They create combinatorial Gray codes, where consecutive permutations differ by one of two operations (one type of swap or rotation). To the best of our knowledge, no existing permutation generation algorithm has this set of features, see [37,33,27].

Loopless algorithms for permutations rarely use $\mathcal{O}(1)$ space as it cannot support $n!$ different internal states: $\log(n!) = \Theta(n \log n)$. Thus, an $\mathcal{O}(1)$ space algorithm cannot count to $n!$ or compute natural sequences of length $n!$ like the factorial ruler sequence $\mathcal{L}_n$ (Oeis A055881 [28]). This discounts frameworks by Ganapathi and Chowdhurysee [18], which generalize 19 previous algorithms using $\mathcal{L}_n$ or a similar sequence $\mathcal{R}_n$; also see Knuth's framework [24]. (As a specific example, Zaks[4] uses two additional arrays.) Thus, an $\mathcal{O}(1)$ space algorithm must at times *read* from the current permutation. This is true of cool-lex order's simple successor rule [30], which can be generated by a loopless $\mathcal{O}(1)$ space algorithm for multiset permutations [39], but it uses $n-1$ different changes. Shorthand universal cycles [31,23,36] give simple Gray codes with two change types, but no existing loopless implementation uses $\mathcal{O}(1)$ space.

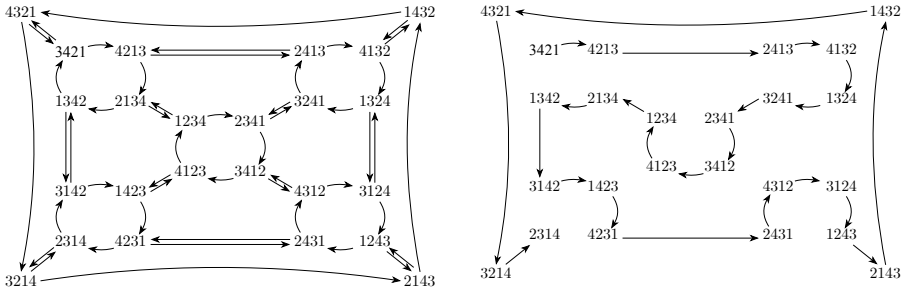
Our algorithms generate (σ, τ)-Gray codes by Sawada and Williams [34,35]. Here τ swaps the first two values, and σ rotates the full permutation one position to the left. Hamilton paths are given for all n [34,35] and Hamilton cycles for odd n [35] in the underlying directed Cayley graph $\mathcal{G}_n$. Figure 1 shows $\mathcal{G}_4$ and a Hamilton path; Hamilton cycles do not exist for even n [29,38]. Both papers give successor rules and worst-case $\mathcal{O}(n)$ time array-based C programs. Egan created length $n!+(n-1)!+(n-2)!+(n-3)!+n-3$ *superpermutations* [9,11] using (σ, τ)-Gray codes from an earlier manuscript [41]. Prior work had found Hamilton cycles in the *undirected shuffle exchange network* (i.e., $\mathcal{G}_n$ plus σ^{-1} edges) [8,4].

In his pioneering work on loopless algorithms, Ehrlich [10] differentiates between (a) changing the current object into its successor, and (b) deciding which change to apply in (a). Note that both types of computation must be completed in worst-case $\mathcal{O}(1)$ time to obtain a loopless algorithm. Our loopless σ-τ algorithms use circular data structures to address (a) since the σ operation requires $\Theta(n)$ time in a conventional array. To address (b), we must carefully introduce additional variables that can be updated in worst-case $\mathcal{O}(1)$ time. The output of one of our algorithms for $n = 4$ (see Section 4) is visualized in Figure 2.

Our contribution is summarized in Theorem 1 (subsuming Lemmas 3, 6, and 7). Full C code can be found at https://github.com/fmasillo/sigma-tau.

Theorem 1. *There is a data structure implementing the Hamilton path successor rule of [34], as well as the Hamilton path and Hamilton cycle successor rules of [35], in worst-case $\mathcal{O}(1)$ time per permutation, using $\mathcal{O}(1)$ additional space.*

[4] His *pancake flip order* dates to the 1700s [22] (see [6]) and is loopless in a BLL [40].



(a) The Sigma-Tau graph $\mathcal{G}_4$ [34]. (b) Hamilton path HP from 3421 to 2314.

Fig. 1: Our loopless algorithms traverse Hamilton paths and cycles in the Sigma-Tau graph $\mathcal{G}_n$ in worst-case $\mathcal{O}(1)$ time per vertex. The path in (b) follows [35].

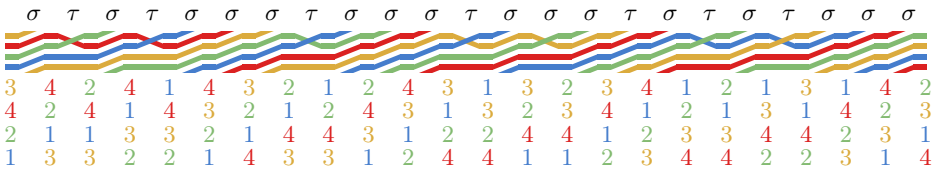


Fig. 2: An alternate order of permutations HP' from [34]. Each τ transition swaps the first (i.e., topmost) pair of elements. Each σ left-rotates all elements one position, with the leftmost visualized as wrapping around from top to bottom.

2 Constant time successor rule for Hamilton paths

Sawada and Williams in [34] provided a successor rule for Hamilton paths, which they later modified slightly in [35] to harmonize with the Hamilton cycle successor rule in the same paper. We first look at the latter rule, and will discuss the original rule [34] in Section 4.

In the new version [35], the following successor rule was given for constructing a Hamilton path in $\mathcal{G}_n$, for any $n > 1$:

> **Hamilton path successor rule for $\mathcal{G}_n$** ([35]) Let $\pi = \pi(1)\pi(2)\cdots\pi(n)$ be a permutation and let r be the symbol to the right of n when π is considered cyclically and skipping over $\pi(2)$. Define the successor rule HP on $\mathcal{G}_n$ as follows:

$$HP(\pi) = \begin{cases} \tau(\pi) & \text{if } (r, \pi(2)) \in \{(1,2),(2,3),\ldots,(n-2,n-1),(n-1,2)\} \\ & \quad \text{and } \pi \neq n(n-1)(n-2)\cdots 1; \\ \sigma(\pi) & \text{otherwise.} \end{cases}$$

Let $p = \pi^{-1}(n)$, then the definition of r in the successor rule above is: $r = \pi(3)$ if $p = 1$, $r = \pi(1)$ if $p = n$, and $r = \pi(p+1)$ otherwise.

The authors of [35] gave a simple array-based implementation (see their Appendix), which, as they state, results in $\mathcal{O}(n)$ worst-case time per permutation. The code runs in $\Omega(n)$ time for three reasons: (1) the sigma-operation, (2) identifying the position of n in π, and (3) deciding if π is the *special (decreasing) permutation* $\pi_{sp} = n(n-1)(n-2)\cdots 321$. The programs in [34,35] also count to $n!$, thus requiring $\Omega(n \log n)$ bits of memory, which is not $O(1)$ space.

Our implementation uses an array and three integer variables. Given a permutation π, an *up-step*[5] is a position where π, taken circularly, increases, that is, a position i such that $\pi(i) < \pi(1 + (i \bmod n))$. Our data structure consists of the following components:

1. an array $C[1, n]$ containing a rotation of π,
2. a pointer b to the position of $\pi(1)$, $C[b] = \pi(1)$,
3. a pointer p to the position of n, $C[p] = n$, and
4. a counter u, giving the number of up-steps of π.

Example 1. Let $n = 7$ and $\pi = 5624137$. Then the following are two possible implementations: $C_1 = [4, 1, 3, 7, 5, 6, 2], b_1 = 5, p_1 = 4, u_1 = 4$, or $C_2 = [5, 6, 2, 4, 1, 3, 7], b_2 = 1, p_2 = 7, u_2 = 4$. Note that u remains invariant.

Note that any value $\pi(i)$ can be accessed in constant time, since $\pi(i) = C[1 + (b + i - 2 \bmod n)]$. In particular, permutation π can be listed as $C[b], C[b+1], \ldots, C[n], C[1], \ldots, C[b-1]$, and can thus be returned in $\mathcal{O}(n)$ time, if required.

We can check the conditions whether to apply τ or σ in constant time:

Lemma 1. *Let π be a permutation and C, b, p, u as defined. We can test in $\mathcal{O}(1)$ time if (1) $(r, \pi(2)) \in \{(1, 2), (2, 3), \ldots, (n-2, n-1), (n-1, 2)\}$, and (2) $\pi = \pi_{sp}$.*

Proof. 1. Recall that $\pi(i) = C[1 + (b + i - 2 \bmod n)]$ is computed in constant time. In particular $\pi(2) = C[1 + (b \bmod n)]$. On the other hand, we compute r in constant time as $C[1 + (p \bmod n)]$ if $p \neq b$ and $C[1 + (b+1 \bmod n)]$ otherwise. So we test whether $r < n - 1$ and $\pi(2) = r + 1$, or $r = n - 1$ and $\pi(2) = 2$.

2. It is easy to see that $\pi = \pi_{sp}$ if and only if $u = 1$ and $p = b$. □

We next show how to implement σ and τ using our data structure:

Lemma 2. *Both operations σ and τ can be executed in constant time using the data structure C, b, p, u.*

Proof. A σ-operation is implemented in constant time by simply incrementing b circularly, $b = 1 + (b \bmod n)$. A τ-operation, which exchanges $\pi(1)$ with $\pi(2)$, is implemented, again in constant time, as follows:

1. decrement u once if $\pi(n) < \pi(1)$, once if $\pi(1) < \pi(2)$, and once if $\pi(2) < \pi(3)$;
2. increment u once if $\pi(n) < \pi(2)$, once if $\pi(2) < \pi(1)$, and once if $\pi(1) < \pi(3)$;
3. set $p = 1 + (b \bmod n)$ if $p = b$, or set $p = b$ if $p = 1 + (b \bmod n)$; and
4. exchange $C[b]$ with $C[1 + (b \bmod n)]$. □

[5] Note that this definition differs from *ascent*, which is not taken circularly.

The total space occupied by our data structure is the permutation itself (array C), and in addition the three integer variables, each taking $\Theta(\log n)$ bits. Our algorithm needs $\mathcal{O}(n)$ time to write the initial permutation $\pi_{sp} \cdot \tau$ to C and $O(1)$ time to initialize the variables b, p, u. (Alternatively, we can view the initial permutation as the input, in which case we refer to the input array as C.) Thus:

Lemma 3. *Using the data structure consisting of array $C[1, n]$ and the variables b, p, u, which are initialized in $\mathcal{O}(n)$ time, we can construct a Hamilton path in $\mathcal{G}_n$, starting from the permutation $\pi_{sp} \cdot \tau = (n-1)n(n-2)\cdots 321$ and implementing the HP successor rule of [35], in $\mathcal{O}(1)$ worst-case time per permutation, using $\mathcal{O}(1)$ extra words.*

3 Constant time successor rule for Hamilton cycles

In [35], a successor rule for Hamiltonian cycles for odd n is provided. The authors define the *special set* R_n, included in the conditions for applying τ rather than σ. For a permutation π, we define $\pi_{\setminus 2}$ the $(n-1)$-length string obtained from π by removing the element in position 2 (which is also an $(n-1)$-permutation in case $\pi(2) = n$). Then the special set is defined as $R_n = \{\pi \mid \pi(2) = n \text{ and } \pi_{\setminus 2} \text{ is a rotation of } id_{n-1}\}$. E.g., $R_5 = \{15234, 25341, 35412, 45123\}$.

> **Hamilton cycle successor rule for $\mathcal{G}_n$, where n is odd** ([35]) Let $\pi = \pi(1)\pi(2)\cdots\pi(n)$ be a permutation and let r be the symbol to the right of n when π is considered cyclically and skipping over $\pi(2)$. Define:
>
> $$HC(\pi) = \begin{cases} \tau(\pi) & \text{if } (r, \pi(2)) \in \{(1,2),(2,3),\ldots,(n-2,n-1),(n-1,2)\} \\ & \text{or } \pi \in R_n \\ \sigma(\pi) & \text{otherwise.} \end{cases}$$

Again, the array based implementation given in [35] results in $\mathcal{O}(n)$ time per permutation in the worst case. In order to achieve constant time, we slightly modify our data structure, replacing counter u by counter u', the number of up-steps of $\pi_{\setminus 2}$, that is, we count up-steps skipping over position 2.

Lemma 4. *It can be checked in constant time whether $\pi \in R_n$.*

Proof. It is clear that an $(n-1)$-permutation is a rotation of the identity $123\cdots(n-2)(n-1)$ if and only if the number of its up-steps is $n-2$. The fact that n is inserted in position 2 is equivalent to $p = 1 + (b \bmod n)$. If $\pi(2) = n$ then $\pi_{\setminus 2}$ is an $(n-1)$-permutation, and therefore, $\pi_{\setminus 2}$ is a rotation of $123\cdots(n-2)(n-1)$ if and only if $u' = n-2$. Both checks can be done in constant time. □

Lemma 5. *Both operations σ and τ can be executed in constant time, using the modified data structure C, b, p, u'.*

Proof. For the σ-operation, before setting $b = 1 + (b \bmod n)$ we need to update u' in constant time as follows:

1. decrement u' once if $\pi(1) < \pi(3)$, and once if $\pi(3) < \pi(4)$;
2. increment u' once if $\pi(1) < \pi(2)$, and once if $\pi(2) < \pi(4)$.

For the τ-operation we do as follows, also in constant time:

1. decrement u' once if $\pi(n) < \pi(1)$, and once if $\pi(1) < \pi(3)$;
2. increment u' once if $\pi(n) < \pi(2)$, and once if $\pi(2) < \pi(3)$;
3. set $p = 1 + (b \bmod n)$ if $p = b$, or set $p = b$ if $p = 1 + (b \bmod n)$;
4. exchange $C[b]$ with $C[1 + (b \bmod n)]$. □

Similarly to the Hamilton path data structure, we use additional $\mathcal{O}(1)$ words. Note that the Hamilton cycle can be started at any permutation. From this discussion and Lemmas 1, 4, and 5, we have:

Lemma 6. *Using the data structure consisting of array $C[1, n]$ and the variables b, p, u', which are initialized in $\mathcal{O}(n)$ time, we can construct a Hamilton cycle in $\mathcal{G}_n$, starting from the identity permutation and implementing the HC successor rule of [35], in $\mathcal{O}(1)$ worst-case time per permutation, using $\mathcal{O}(1)$ extra words.*

4 Simpler rule for Hamilton paths and termination

The original Hamilton path successor rule HP' given in [34] differs in only one detail from the one in [35], namely that in the condition for τ, $(n-1, 2)$ is replaced by $(n-1, 1)$. The resulting Hamilton paths in $\mathcal{G}_4$ is visualized in Figure 2.

This change can be easily accommodated using our data structure, by a simple change in the condition for applying τ. Alternatively, insights from [32] on this Hamilton path can be used for a further simplification: The *syntactic sequence* of a Hamilton path in $\mathcal{G}_n$ is a string over the alphabet $\{\tau, \sigma\}$ which specifies the sequence of operations applied. Rytter and Zuba [32] showed that for the Hamilton path resulting from successor rule HP', this sequence is highly compressible.

Lemma 7. *Using the data structure consisting of array $C[1, n]$ and variables b, p, u, which are initialized in $\mathcal{O}(n)$ time, we can construct a Hamilton path in $\mathcal{G}_n$, starting from the permutation $\pi_{sp} \cdot \tau = (n-1)n(n-2)\cdots 321$ and implementing the HP' successor rule of [34], in $\mathcal{O}(1)$ worst-case time per permutation, using $\mathcal{O}(1)$ extra words.*

Termination. To terminate our algorithms, we cannot resort to a counter maintaining the number of permutations, as is done in [34,35], since this would exceed the $\mathcal{O}(1)$ space restriction. Instead, we apply termination conditions identifying the final permutation. For example, as we start the *HC* algorithm at the identity $id = 123\cdots n$, the final permutation is $n123\cdots(n-1)$. This is the unique permutation with $\pi(1) = n$, $\pi(2) = 1$, and $u' = n-2$ up-steps (skipping over $\pi(2)$). Similar tests terminate *HP* and HP': starting from $\pi_{sp} \cdot \tau$, we have to detect when the last permutation $(n-2)(n-1)(n-3)(n-4)\cdots 21n$ occurs. This can be done again in constant time and space by checking whether $u = 2, \pi(1) = n-2$, $\pi(2) = n-1, \pi(n-1) = 1$, and $\pi(n) = n$: the only permutation with those extreme values fixed and with no further up-steps is the one containing the descending sequence $(n-3)\cdots 2$ in between.

References

1. Bader, D.A., Moret, B.M.E., Yan, M.: A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. J. Comput. Biol. **8**(5), 483–491 (2001)
2. Bafna, V., Pevzner, P.A.: Genome rearrangements and sorting by reversals. In: Proc. of the 34th Annual Symposium on Foundations of Computer Science (FOCS 1993). pp. 148–157. IEEE Computer Society (1993)
3. Bafna, V., Pevzner, P.A.: Sorting by transpositions. SIAM J. Discret. Math. **11**(2), 224–240 (1998)
4. Bass, D.W., Sudborough, I.H.: On the shuffle-exchange permutation network. In: Proc. of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN'97). pp. 165–171. IEEE (1997)
5. Boucher, C., Cenzato, D., Lipták, Zs., Rossi, M., Sciortino, M.: r-Indexing the eBWT. In: Proc. of the 28th International Symposium on String Processing and Information Retrieval (SPIRE 2021). Lecture Notes in Computer Science, vol. 12944, pp. 3–12. Springer (2021)
6. Cameron, B., Sawada, J., Therese, W., Williams, A.: Hamiltonicity of k-sided pancake networks with fixed-spin: Efficient generation, ranking, and optimality. Algorithmica **85**(3), 717–744 (2023)
7. Cerbai, G., Ferrari, L.S.: Permutation patterns in genome rearrangement problems: The reversal model. Discret. Appl. Math. **279**, 34–48 (2020)
8. Compton, R.C., Gill Williamson, S.: Doubly adjacent gray codes for the symmetric group. Linear and Multilinear Algebra **35**(3-4), 237–293 (1993)
9. Egan, G.: Superpermutations. `http://www.gregegan.net/SCIENCE/Superpermutations/Superpermutations.html` (2018)
10. Ehrlich, G.: Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. J. ACM **20**(3), 500–513 (1973)
11. Engen, M., Vatter, V.: Containing all permutations. The American Mathematical Monthly **128**(1), 4–24 (2020)
12. Feng, J., Zhu, D.: Faster algorithms for sorting by transpositions and sorting by block interchanges. ACM Trans. Algorithms **3**(3), 25 (2007)
13. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM **52**, 552–581 (2005)
14. Fertin, G., Labarre, A., Rusu, I., Tannier, E., Vialette, S.: Combinatorics of Genome Rearrangements. Computational molecular biology, MIT Press (2009)
15. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theoretical Computer Science **410**(51), 5354–5364 (2009)
16. Gagie, T., Navarro, G., Prezza, N.: Fully-functional suffix trees and optimal text searching in BWT-runs bounded space. Journal of the ACM **67**(1), article 2 (2020)
17. Gagie, T., Navarro, G., Prezza, N.: Optimal-time text indexing in BWT-runs bounded space. In: Proc. of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018). pp. 1459–1477 (2018)
18. Ganapathi, P., Chowdhury, R.: A unified framework to discover permutation generation algorithms. The Computer Journal **66**(3), 603–614 (2023)
19. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2), 378–407 (2005)
20. Hannenhalli, S., Pevzner, P.A.: Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. In: Proc. of the 27th Annual ACM Symposium on Theory of Computing (STOC 1995). pp. 178–189. ACM (1995)

21. Hartman, T., Shamir, R.: A simpler and faster 1.5-approximation algorithm for sorting by transpositions. Inf. Comput. **204**(2), 275–290 (2006)
22. Hindenburg, C.F.: Sammlung combinatorisch-analytischer Abhandlungen, vol. 1. ben Gerhard Fleischer dem Jungern (1796)
23. Holroyd, A.E., Ruskey, F., Williams, A.: Shorthand universal cycles for permutations. Algorithmica **64**, 215–245 (2012)
24. Knuth, D.E.: The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations (Art of Computer Programming). Addison-Wesley Professional (2005)
25. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nord. J. Comput. **12**(1), 40–66 (2005)
26. Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Succinct representations of permutations and functions. Theoretical Computer Science **438**, 74–88 (2012)
27. Mütze, T.: Combinatorial Gray codes—an updated survey. Electronic Journal of Combinatorics **30**(3-DS26) (2023)
28. OEIS Foundation Inc.: Sequence A055881 in the On-line Encyclopedia of Integer Sequences. `https://oeis.org/A055881`, accessed on June 2 2023
29. Rankin, R.A.: A campanological problem in group theory. In: Mathematical Proceedings of the Cambridge Philosophical Society. vol. 44, pp. 17–25. Cambridge University Press (1948)
30. Ruskey, F., Williams, A.: The coolest way to generate combinations. Discrete Mathematics **309**(17), 5305–5320 (2009)
31. Ruskey, F., Williams, A.: An explicit universal cycle for the $(n-1)$-permutations of an n-set. ACM Transactions on Algorithms (TALG) **6**(3), 1–12 (2010)
32. Rytter, W., Zuba, W.: Syntactic view of sigma-tau generation of permutations. Theor. Comput. Sci. **882**, 49–62 (2021)
33. Savage, C.D.: A survey of combinatorial Gray codes. SIAM Rev. **39**(4), 605–629 (1997)
34. Sawada, J., Williams, A.: A Hamilton path for the Sigma-Tau problem. In: Proc. of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2018). pp. 568–575. SIAM (2018)
35. Sawada, J., Williams, A.: Solving the Sigma-Tau problem. ACM Trans. Algorithms **16**(1), 11:1–11:17 (2020)
36. Sawada, J., Williams, A.: Constructing the first (and coolest) fixed-content universal cycle. Algorithmica pp. 1–32 (2022)
37. Sedgewick, R.: Permutation generation methods. ACM Computing Surveys (CSUR) **9**(2), 137–164 (1977)
38. Swan, R.G.: A simple proof of Rankin's campanological theorem. The American mathematical monthly **106**(2), 159–161 (1999)
39. Williams, A.: Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In: Proc. of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2009). pp. 987–996. SIAM (2009)
40. Williams, A.: $O(1)$-time unsorting by prefix-reversals in a boustrophedon linked list. In: Proc. of the 5th International Conference on Fun with Algorithms (FUN 2010). pp. 368–379. Springer (2010)
41. Williams, A.: Hamiltonicity of the Cayley digraph on the symmetric group generated by $\sigma = (1\ 2 \dots\ n)$ and $\tau = (1\ 2)$. CoRR **abs/1307.2549** (2013)