A constant-time algorithm for middle levels Gray codes

Torsten Mütze
Institut für Mathematik
TU Berlin
10623 Berlin, Germany
muetze@math.tu-berlin.de

Jerri Nummenpalo
Department of Computer Science
ETH Zürich
8092 Zürich, Switzerland
njerri@inf.ethz.ch

Abstract. For any integer $n \geq 1$ a *middle levels Gray code* is a cyclic listing of all n-element and $(n+1)$-element subsets of $\{1, 2, \ldots, 2n+1\}$ such that any two consecutive subsets differ in adding or removing a single element. The question whether such a Gray code exists for any $n \geq 1$ has been the subject of intensive research during the last 30 years, and has been answered affirmatively only recently [T. Mütze. Proof of the middle levels conjecture. *Proc. London Math. Soc.*, 112(4):677–713, 2016]. In a follow-up paper [T. Mütze and J. Nummenpalo. An efficient algorithm for computing a middle levels Gray code. *Proc. ESA*, 2015] this existence proof was turned into an algorithm that computes each new set in the Gray code in time $\mathcal{O}(n)$ on average. In this work we complete this line of research by presenting an algorithm for computing a middle levels Gray code in optimal time and space: Each new set is generated in time $\mathcal{O}(1)$, and the required space is $\mathcal{O}(n)$.

Keywords: Middle levels conjecture, Gray code, Hamilton cycle

1. Introduction

Efficiently generating all objects in a particular combinatorial class such as permutations, subsets, combinations, partitions, trees, strings etc. is one of the oldest and most fundamental algorithmic problems, and such generation algorithms are used as building blocks in a wide range of practical applications (the survey [Sav97] lists numerous references). The 'holy grail' for all these problems is to come up with generation algorithms that generate each new object in constant time, entailing that consecutively generated objects may differ only in a constant amount (we will mention some concrete examples momentarily). In his influential paper [Ehr73], Ehrlich coined the term *loopless algorithms* for generation algorithms with the property that the time between any two consecutively generated objects is $\mathcal{O}(1)$. In subsequent years such algorithms with optimal running time have been discovered for a vast collection of combinatorial classes (many of these results are covered in the classical books [NW75, Wil89, Knu11]). To mention four concrete examples, loopless algorithms are known for the problems of (1) generating all permutations of the numbers $1, 2, \ldots, n$ by adjacent transpositions [Joh63, Tro62, Ehr73, Der75, Sed77], (2) generating all subsets of $[n] := \{1, 2, \ldots, n\}$ by adding or removing a single element [Gra53, Ehr73, BER76], (3) generating all k-element subsets of an $[n]$ by exchanging a single element [Ehr73, BER76, EHR84, EM84, Rus88], (4) generating all binary trees with n vertices by single rotation operations [Luc87, LvBR93].

In this paper we revisit the well-known problem of generating all n-element and $(n+1)$-element subsets of $[2n+1]$ by adding or removing a single element. In a computer these subsets are naturally represented by bitstrings of length $2n+1$, with 1-bits at the positions of the elements contained in the subset (and 0-bits at the remaining positions), so the problem is equivalent to generating all bitstrings of length $2n+1$ with weight n or $n+1$, where the *weight* of a bitstring is the number of 1-bits in it. We refer to such a Gray code as as *middle levels Gray code*. Clearly, a middle levels Gray code consists of $N := \binom{2n+1}{n} + \binom{2n+1}{n+1} = 2\binom{2n+1}{n} = 2^{\Theta(n)}$ many bitstrings in total, and the weight alternates between n and $n+1$ in each step. The existence of a middle levels Gray code for any $n \geq 1$ is asserted by the well-known *middle levels conjecture*, raised independently in the 80s by Havel [Hav83] and Buck and Wiedemann [BW84]. The conjecture has also been attributed to Dejter, Erdős, Trotter [KT88] and various others, and also appears in the popular books [Win04, Knu11, DG12]. The middle levels conjecture has attracted considerable attention over the last 30 years [Sav93, FT95, SW95, Joh04, DSW88, KT88, DKS94, HKRR05, GŠ10, MW12, SSS09, SA11], and a positive solution, i.e., an existence proof for a middle levels Gray code for any $n \geq 1$, has been announced only recently.

Theorem 1 ([Müt16]). *A middle levels Gray code exists for any $n \geq 1$.*

In a follow-up paper [MN15], this existence argument was turned into an efficient algorithm for computing a middle levels Gray code.

Theorem 2 ([MN15]). *There is an algorithm, which for a given bitstring of length $2n+1$, $n \geq 1$, with weight n or $n+1$ computes the next $\ell \geq 1$ bitstrings in a middle levels Gray code in time $\mathcal{O}(n\ell(1+\frac{n}{\ell}))$.*

Clearly, the running time of this algorithm is $\mathcal{O}(n)$ on average per generated bitstring for $\ell = \Omega(n)$. However, this falls short of the optimal $\mathcal{O}(1)$ time bound one could hope for, given that in each step only a single bit needs to be flipped (a constant amount of change). In fact, it was conjectured in [MN15] that coming up with a constant-time algorithm would require substantial new insights that would in particular yield a much simpler proof of Theorem 1.

1.1. Our results.

In this paper we claim the 'holy grail' for computing a middle levels Gray code efficiently by presenting an algorithm with optimal running time.

Theorem 3. *There is an algorithm, which for a given bitstring of length $2n+1$, $n \geq 1$, with weight n or $n+1$ computes the next $\ell \geq 1$ bitstrings in a middle levels Gray code in time $\mathcal{O}(\ell(1+\frac{n^2}{\ell}))$.*

Clearly, the running time of this algorithm is $\mathcal{O}(1)$ *on average* per generated bitstring for $\ell = \Omega(n^2)$ (recall that the total number $N = 2^{\Theta(n)}$ of bitstrings in the Gray code is exponential in n), and the required space is optimal $\mathcal{O}(n)$. The summand n^2 in the runtime bound in Theorem 3 comes from the initialization phase, which is performed only once in the beginning and is usually disregarded for these generation problems. Moreover, the initialization time can be reduced to $\mathcal{O}(n)$ if the algorithm starts from a prescribed bitstring. We shall see that most steps of our algorithm require only constant time $\mathcal{O}(1)$ in the worst case to generate the next bitstring, but after every sequence of $\Theta(n)$ of such 'fast' steps, a 'slow' step which requires time $\Theta(n)$ is encountered (yielding constant average time performance). Therefore, our algorithm could be easily transformed into a loopless algorithm, i.e., one with a $\mathcal{O}(1)$ *worst case* bound for each generated bitstring (and with the same space requirements), by introducing an additional FIFO queue of size $\Theta(n)$ and by simulating the original algorithm such that during every sequence of d 'fast' steps, $d-1$ results are stored in the queue and only one of them is returned, and during the 'slow' steps the queue is emptied at the same speed (the constant d must be chosen so that the queue is empty when the 'slow' steps are finished). Even though the resulting algorithm would indeed be loopless, it would still be slower than the original algorithm, because it produces every bitstring only after it was produced in the original algorithm (due to the delay caused by the queue and the additional instructions for queue management). In other words, the hidden constant in the $\mathcal{O}(1)$ bound for the modified algorithm is higher than for the original algorithm, so this modified loopless algorithm is only of theoretical interest, and we will not discuss it any further in this paper. A similar scenario where a loopless algorithm is slower than a corresponding constant average time algorithm is discussed in detail in [Sed77, Section 1].

We implemented our new middle levels Gray code algorithm (the one referred to in Theorem 3) in C++, and we invite the reader to experiment with this code, which can be found on the authors' websites [www]. As a benchmark, we used this code to compute a middle levels Gray code for $n = 19$ in 31 minutes on a standard desktop computer. This is by a factor of 46 faster than the 24 hours reported in [MN15] for the algorithm from Theorem 2, and by several orders of magnitude faster than the 164 days previously needed for a brute-force search [SA11] (note that a middle levels Gray code for $n = 19$ consists of $N = 137.846.528.820 \approx 10^{11}$ bitstrings). For comparison, a program that simply counts from $1, \ldots, N$ (and does nothing else) was only by a factor of 8 faster (4 minutes) than our middle levels Gray code computation for $n = 19$ on the same hardware. Roughly speaking, we need only about 8 instructions for producing the next bitstring in the Gray code. Given that we now have an optimal constant-time algorithm for generating middle levels Gray codes that performs extremely well in the benchmarks, the only really interesting remaining problem for future work is to come up with a much simpler algorithm, because our algorithm is admittedly rather complex.

This optimal algorithm now also opens the door for constant-time algorithms for a number of related Gray codes that have been constructed inductively using Theorem 1 as an induction basis. Essentially, these Gray codes consist of several suitably combined middle levels Gray codes of smaller dimensions. E.g., in [MS15] a Gray code was constructed that enumerates all k-element and $(n-k)$-element sets of $[n]$, $n \geq 2k+1$, by either adding or removing $n-2k$ elements in each step, settling a long-standing open problem [Sim91, Sim94, Hur94, Che00, SS04, Che03] (in graph-theoretic terms, these result shows that the so-called *bipartite Kneser graphs* have a Hamilton cycle). With the help of Theorem 3, this construction can now be turned into an efficient algorithm. As another example, in [GM16] Theorem 3 is used to derive constant-time algorithms for generating minimum-change orderings of all bitstrings of length n whose weight is in some interval $[k, l]$, $0 \leq k \leq l \leq n$, a considerable generalization of the classical problem (3) mentioned in the first paragraph of the introduction, improving upon the results from [SW14].

As in [MN15], in this work we restrict our attention to computing one particular 'canonical' middle levels Gray code for any $n \geq 1$, even though we know from [Müt16] that there are double-exponentially (in n) many different ones (recall [MN15, Remark 3]).

1.2. **New ingredients.** At this point let us briefly mention the main algorithmic improvements and differences between the algorithms from Theorem 2 and Theorem 3 which save us a factor of n in the running time. At the lowest level, the algorithm from Theorem 2 consists of a rather unwieldy recursion (the function Paths() in [MN15, Algorithm 1]) which for each bitstring computes the next bitstring in a middle levels Gray code. This recursion runs in time $\Theta(n)$, and therefore represents one of the bottlenecks in the running time. In addition, there are various higher-level functions (called by the function IsFlipVertex() in [MN15, Algorithm 4]), which are called only every $\Theta(n)$ many steps and run in time $\Theta(n^2)$, and which therefore also represent $\Theta(n)$ (on average) bottlenecks. These higher-level functions basically control which subsets of bitstrings are visited in which order, to make sure that each bitstring is visited exactly once.

To overcome those bottlenecks, in our new algorithm we replace the recursion at the lowest level by a new and simple combinatorial technique, first proposed in [MSW16], which allows us to compute for certain 'special' bitstrings that are encountered every $\Theta(n)$ many steps, a sequence of bit positions to be flipped during the next $\Theta(n)$ many steps. Computing such a bitflip sequence can be done in time $\Theta(n)$, and when this is accomplished each subsequent step takes only constant time: We simply flip the precomputed positions one after the other, until the next 'special' bitstring is encountered and the bitflip sequence has to be recomputed. Replacing the complicated inductive/recursive constructions from [Müt16, MN15] at the lowest level by a simple combinatorial bitflip rule indeed yields a considerable simplification of the proof of Theorem 1. However, the higher-level functions in the new algorithm essentially remain the same as in the old one. We cut down their running time by a factor of n (from quadratic to linear) by using more sohisticated data structures and by resorting to well-known standard algorithms such as Booth's linear-time algorithm [Boo80] for computing the lexicographically smallest rotation of a given string.

1.3. **Outline of this paper.** In Section 2 we present our new middle levels Gray code algorithm. In Section 3 we prove the correctness of the algorithm, and in Section 4 we discuss how to implement it to achieve the claimed runtime and space bounds.

2. The algorithm

As in [MN15], we describe our algorithm to compute a middle levels Gray code using the language of graph theory. To that end, we let Q_n denote the *n-dimensional cube*, the graph that has as vertices all bitstring of length n, with an edge between any two bitstrings that differ in exactly one bit. Moreover, for any $0 \leq k < n$ we let $Q_n(k, k+1)$ be the subgraph of Q_n induced by the vertices (=bitstrings) with weight k or $k+1$. Clearly, computing a middle levels Gray code is equivalent to computing a Hamilton cycle in the *middle levels graph* $Q_{2n+1}(n, n+1)$. Our algorithm to compute a Hamilton cycle in the middle levels graph consists of several nested functions, and in the following we explain these functions from bottom to top. The lowest level functions compute various sets of paths in the graph $Q_{2n}(n, n+1)$, and the higher-level functions combine these paths to a Hamilton cycle in the middle levels graph $Q_{2n+1}(n, n+1)$. As mentioned before, compared to the algorithm from [MN15], the lowest level functions of the new algorithm are completely different (they are described in Section 2.2 and 2.3 below), whereas the higher-level functions are essentially the same (they are described in Section 2.4 and 2.5 below). Consequently, to keep the paper concise we will often refer to [MN15] when describing the upper level functions.

2.1. **Definitions.** We begin by introducing some crucial definitions that will be used throughout the paper.

Lattice path interpretation of bitstrings. We let $B_n(k)$ denote the set of all bitstrings of length n with weight k. Any bitstring $x \in B_n(k)$ can be interpreted as a lattice path by reading x from left to right and drawing a path in the integer lattice $\mathbb{Z}_2$ that starts at the origin $(0, 0)$ and that consists of n many steps that change the current coordinate by $(1, 1)$ or $(1, -1)$, respectively, for every 1-bit or 0-bit encountered in x (note that the lattice path ends at the coordinate $(n, 2k-n)$). Figure 1 shows this correspondence/bijection for an example. Whenever we want to emphasize that a bitstring x should be considered as a lattice path, we refer to the 0-bits and 1-bits of x as *0-steps* or *1-steps*, respectively (then we are interested e.g. in the vertical location of these steps). We refer to the number of 0-steps of x below the horizontal line $y = 0$ as the number of *flaws* of x, and we let $D_{2n}^e \subseteq B_{2n}(n)$, $e \in \{0, 1, \ldots, n\}$, denote the set of all bitstrings (viewed as lattice paths) with exactly e flaws. We clearly have $B_{2n}(n) = D_{2n}^0 \cup D_{2n}^1 \cup \cdots \cup D_{2n}^n$ and the well-known Chung-Feller theorem asserts that $|D_{2n}^0| = |D_{2n}^1| = \cdots = |D_{2n}^n| = \frac{1}{n+1}\binom{2n}{n} = C_n$, where C_n denotes the n-th Catalan number (see [CF49]; in fact, the theorem was first proved in [Mac09, p. 168]). For any lattice path $x \in D_{2n}^0$, considering the first 0-step returning to the line $y = 0$ (after leaving the line $y = 0$ with the first 1-step), and denoting the corresponding index by a, the path $x = (x_1, x_2, \ldots, x_{2n})$ can be partitioned as $x = (1, u, 0, v)$ where $u := (x_2, \ldots, x_{a-1})$ and $v := (x_{a+1}, \ldots, x_{2n})$. Similarly, for any lattice path $x \in D_{2n}^1$, considering the unique 0-step below the line $y = 0$, and denoting the corresponding index by a, the path $x = (x_1, x_2, \ldots, x_{2n})$ can be partitioned as $x = (u, 0, 1, v)$ where $u := (x_1, \ldots, x_{a-1})$ and $v := (x_{a+2}, \ldots, x_{2n})$. We refer to these partitionings as *canonical decomposition of x*.

Elementary operations on sequences/bitstrings. For any sequence $s = (s_1, s_2, \ldots, s_n)$ and any integer a, we define $s + a = a + s := (s_1 + a, s_2 + a, \ldots, s_n + a)$ and $\text{rev}(s) := (s_n, \ldots, s_1)$. Moreover, we let

$|s| := n$ denote the length of the sequence. For any bitstring $x = (x_1, x_2, \ldots, x_n)$, we define $\mathrm{rev}(x) := (x_n, x_{n-1}, \ldots, x_1)$ and $\overline{x} := (\overline{x_1}, \overline{x_2}, \ldots, \overline{x_n})$, where $\overline{x_i}$ denotes the complement of x_i. We define $\overline{\mathrm{rev}}(x) := \mathrm{rev}(\overline{x}) = \overline{\mathrm{rev}(x)}$. For even n we also define $\pi(x) := (x_1, x_3, x_2, x_5, x_4, \ldots, x_{n-1}, x_{n-2}, x_n)$.

2.2. **Computing paths in $Q_{2n}(n, n+1)$.** In this section we provide a simple combinatorial description of a set of disjoint paths $\mathcal{P}_{2n}(n, n+1)$ that together visit all vertices the graph $Q_{2n}(n, n+1)$. The starting vertices of these paths are the vertices $x \in D_{2n}^0$, and in the following we describe a rule $\sigma(x)$ which specifies the sequence of bit positions to be flipped along the path starting at x. To compute the bitflip sequence $\sigma(x)$, we interpret x as a lattice path, and we alternatingly flip 0-steps and 1-steps of this lattice path (corresponding to 0-bits and 1-bits in x, respectively). Specifically, for $x = (x_1, x_2, \ldots, x_{2n}) \in D_{2n}^0$, $n \geq 1$, we consider the canonical decomposition $x = (1, u, 0, v)$ and we define

$$\sigma(x) := \big(|u| + 2,\ 1,\ 1 + \sigma'(u)\big) \ , \tag{1a}$$

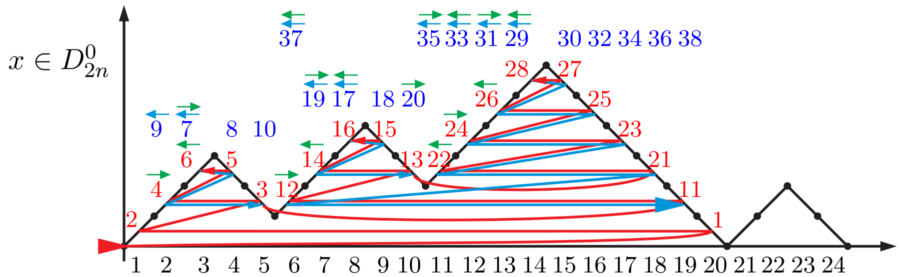
where $\sigma'(x')$ is defined for any subpath $x' = (x_i, x_{i+1}, \ldots, x_{i+2n'-1}) \in D_{2n'}^0$ of x by considering the canonical decomposition $x' = (1, u', 0, v')$ and by recursively computing

$$\sigma'(x') := \begin{cases} () & \text{if } |x'| = 0 \ , \\ \big(|u'| + 2,\ 1 + a,\ 1 + \sigma'(u'),\ 0 + b,\ |u'| + 2 + c,\ |u'| + 2 + \sigma'(v')\big) & \text{otherwise} \ , \end{cases} \tag{1b}$$

where

$$a := \begin{cases} 1 & \text{if } i \text{ even and } |u'| > 0 \\ -1 & \text{if } i \text{ odd} \\ 0 & \text{otherwise} \end{cases} , \quad b := \begin{cases} -1 & \text{if } i \text{ even and } i > 2 \\ 1 & \text{if } i \text{ odd} \\ 0 & \text{otherwise} \end{cases} , \quad c := \begin{cases} 1 & \text{if } i \text{ odd and } |v'| > 0 \\ 0 & \text{otherwise} \end{cases} . \tag{1c}$$

In words, the sequence $\sigma(x)$ defined in (1a) first flips the 0-step immediately after the subpath u (at position $|u| + 2$ of x), then the 1-step immediately before the subpath u (at position 1 in x), and then recursively steps of u (no steps of v are flipped at all). Assuming for a moment that $a = b = c = 0$, then the sequence $\sigma'(x')$ defined in (1b) first flips the 0-step immediately after the subpath u', then the 1-step immediately before the subpath u', then recursively steps of u', then again the step immediately to the left of x' (which is not part of x', hence the index 0; this will be a 1-step), then again the step immediately to the right of u' (this will be a 0-step), and finally recursively all steps of v'. The variables a, b, c have the additional effect of shifting the positions of certain flips by at most 1. The overall effect is that $\sigma(x)$ flips all steps of x at least once, some twice, and some even three times. Consider e.g. the lattice path x in Figure 1: The 1-step at position 14 is flipped once (in step 28, i.e., $\sigma(28) = 14$), the 0-step at position 5 is flipped twice (in step 3 and 10, i.e., $\sigma(3) = \sigma(10) = 5$), and the 0-step at position 4 three times (in steps 5, 8 and 37, i.e., $\sigma(5) = \sigma(8) = \sigma(37) = 4$). It becomes apparent from this figure that the recursion $\sigma(x)$ has a straightforward combinatorial interpretation: We consider the 'hills' of the lattice path x with increasing height levels and from left to right on each level, and flip the steps of each hill in phase 1 alternatingly between the rightmost and leftmost step (going up the hill, these are the red indices in the figure), and in phase 2 alternatingly between the leftmost and rightmost step (going down the hill, these are the blue indices in the figure). Moreover, all positions determined in phase 2 that are located on a 1-step of x are shifted to the left by 1 (this is indicated by the blue arrows in the figure), and the additional effect of the variables a, b, c is another shift of all positions (encountered in phase 1 or 2) that are located on a 1-step of x by either $+1$ or -1 depending on the parity of the index of the 1-step (this is indicated by the green arrows in the figure). We emphasize here that during the recursive computation of $\sigma(x)$, no steps of x are ever actually flipped, but we always consider the same lattice path (and its subpaths) as a function argument.



$x = (1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0)$

$\sigma(x) = (20, 1, 5, 3, 4, 2, 3, 4, 1, 5, 19, 7, 10, 6, 9, 8, 6, 9, 7, 11, 18, 10, 17, 13, 16, 12, 15, 14, 12, 15, 13, 16, 10, 17, 11, 18, 4, 19)$

FIGURE 1. Illustration of the recursive computation of the bitflip sequence $\sigma(x)$, $x \in D^0_{2n}$. The red and blue numbers $i = 1, 2, \ldots, 38$ indicate the position of x to be flipped in step i, i.e., the i-th entry of $\sigma(x)$, where the green and blue arrows act as modifiers that change the position drawn in the figure by $+1$ (right-arrow) or -1 (left-arrow). E.g., the 11-th entry of $\sigma(x)$ is the position of the 0-step of x marked with a red 11, so $\sigma(11) = 19$; the 12-th entry of $\sigma(x)$ is the position of the 1-step of x marked with a red 12 plus 1 (because of the green right-arrow), so $\sigma(12) = 6+1 = 7$; the 7-th entry of $\sigma(x)$ is the position of the 1-step of x marked with a blue 7 minus 1 plus 1 (because of the blue and green arrow in opposite directions), so $\sigma(7) = 3 - 1 + 1 = 3$; the 37-th entry of $\sigma(x)$ is the position of the 1-step of x marked with a blue 37 minus 1 minus 1 (because of the blue and green left-arrow), so $\sigma(37) = 6 - 1 - 1 = 4$.

Note that the total number of bitflips of $\sigma(x)$ for any $x \in D^0_{2n}$ is $2|u| + 2$, where $x = (1, u, 0, v)$ is the canonical decomposition of x. For any $x \in D^0_{2n}$, we let $P_\sigma(x)$ denote the sequence of vertices obtained by flipping the bits of x one after the other in the order given by $\sigma(x)$. Later we will formally prove the following properties:

(i) For any $x \in D^0_{2n}$, flipping the bits at the positions indicated by $\sigma(x)$ one after the other yields a sequence of distinct vertices in the graph $Q_{2n}(n, n+1)$, i.e., a path $P_\sigma(x)$ in this graph. Moreover, all paths in $\mathcal{P}_{2n}(n, n+1) := \{P_\sigma(x) \mid x \in D^0_{2n}\}$ are disjoint, and together they visit all vertices of $Q_{2n}(n, n+1)$.

(ii) For any first vertex $x \in D^0_{2n}$, considering the canonical decomposition $x = (1, u, 0, v)$, the last vertex of $P_\sigma(x)$ is given by $(\pi(u), 0, 1, v) \in D^1_{2n}$. Consequently, the sets of first and last vertices of the paths $\mathcal{P}_{2n}(n, n+1)$ are given by D^0_{2n} and D^1_{2n}, respectively.

Table 1 illustrates properties (i) and (ii) by showing the start and end vertices of all the paths $\mathcal{P}_{2n}(n, n+1)$ in the graph $Q_{2n}(n, n+1)$ for $n = 3$ and the corresponding bitflip sequences.

The paths $\mathcal{P}_{2n}(n, n+1)$ are exactly the ones constructed in [MW12] and [Müt16] for the all-one parameter sequence. However, the key advantage of our new combinatorial description via bitflip sequences is that it allows for a constant-time algorithm: We simply compute the bitflip sequence at a first vertex $x \in D^0_{2n}$ using the recursive rule (1) (this can be done in time linear in $|u|$, where $(1, u, 0, v)$ is the canonical decomposition of x), and then execute each of the bitflips in $\sigma(x)$ in constant time along the path $P_\sigma(x)$ (which has length $2|u| + 2$). In contrast to that, the inductive description of the same paths given in [MW12] and [Müt16] does not allow for an efficient algorithm at all, and the recursive description given in [MN15] only allows for a linear-time algorithm (at *each* vertex along the path, a linear-time recursion has to be performed).

2.3. **Flippable pairs.** It turns out that the set of paths $\mathcal{P}_{2n}(n, n+1)$ constructed in the previous section using the recursion σ is not sufficient for computing a Hamilton cycle in the middle levels

First vertex $x \in D_{2n}^0$	Bitflip sequence $\sigma(x)$	Last vertex $x \in D_{2n}^1$
(1, 1,1,0,0 ,0)	$(6,1,5,3,4,2,3,4,1,5)$	(1,0,1,0 ,0,1)
(1, 1,0,1,0 ,0)	$(6,1,3,2,1,3,5,4,2,5)$	(1,1,0,0 ,0,1)
(1, 1,0 ,0, 1,0)	$(4,1,3,2,1,3)$	(1,0 ,0,1, 1,0)
(1,0, 1,1,0,0)	$(2,1)$	(0,1, 1,1,0,0)
(1,0, 1,0,1,0)	$(2,1)$	(0,1, 1,0,1,0)

Table 1. Paths $\mathcal{P}_{2n}(n,n+1)$ in $Q_{2n}(n,n+1)$ for $n=3$ obtained from the bitflip sequences $\sigma(x)$, $x \in D_{2n}^0$. The dark-grey and light-grey boxes highlight the substrings of x corresponding to the subpaths u or v, respectively, in the canonical decomposition $x=(1,u,0,v)$ (the same colors are given for the corresponding substrings $\pi(u)$ and v of the canonical decomposition $(\pi(u),0,1,v)$ of the last vertex of the path $P_\sigma(x)$).

graph. As in [MN15], we will need another set of paths $\widetilde{\mathcal{P}}_{2n}(n,n+1)$, which is obtained from the paths $\mathcal{P}_{2n}(n,n+1)$ by *small local modifications*. The modifications are done in pairs, i.e., we change the bitflip sequence for some pairs of paths $P,P' \in \mathcal{P}_{2n}(n,n+1)$ such that the resulting modified paths $R,R' \in \widetilde{\mathcal{P}}_{2n}(n,n+1)$ are disjoint and satisfy the following conditions: $F(P)=F(R)$, $F(P')=F(R')$, $L(P)=L(R')$, $L(P')=L(R)$, $V(P)\cup V(P')=V(R)\cup V(R')$, where $V(G)$ denotes the vertex set of any graph G, and for any path P constructed via a bitflip sequence $F(P)$ and $L(P)$ denote the first and last vertex of the path, respectively. In words, the paths R,R' visit the same set of vertices as P,P', but the two pairs of end vertices are connected the other way. We refer to a pair (P,P') of such paths as a *flippable pair of paths*, and to the corresponding modified paths (R,R') as *flipped paths*. By construction, each path from $\mathcal{P}_{2n}(n,n+1)$ will be part of at most one flippable pair (i.e., all flippable pairs are disjoint) and the set $\widetilde{\mathcal{P}}_{2n}(n,n+1)$ of flipped paths will contain fewer paths than the set $\mathcal{P}_{2n}(n,n+1)$.

For certain vertices $x \in D_{2n}^0$ and the corresponding bitflip sequence $\sigma(x)$, we now define a modified bitflip sequence $\widetilde{\sigma}(x)$ which defines the flipped paths $\widetilde{\mathcal{P}}_{2n}(n,n+1)$.

We refer to a pair of vertices (x,x'), $x,x' \in D_{2n}^0$, as *type 1*, if they satisfy the relations $x=(1,1,0,0,v)$ and $x'=(1,0,1,0,v)$ for some $v \in D_{2n-4}^0$, and to pair of vertices (x,x'), $x,x' \in D_{2n}^0$, as *type 2*, if they satisfy the relations $x=(1,v_1,1,1,0,0,0,v_0)$ and $x'=(1,v_1,1,0,1,0,0,v_0)$ for some $v_i \in D_{2n_i}^0$ for $i \in \{0,1\}$ with $n_0+n_1=n-3$ and $n_1 \geq 1$.

We first consider a pair of vertices (x,x'), $x,x' \in D_{2n}^0$, of type 1. Using the definition (1), for such vertices we have $\sigma(x)=(4,1,3,2,1,3)$ and $\sigma(x')=(2,1)$, and we define

$$\widetilde{\sigma}(x) := (3,1) \ , \tag{2a}$$

$$\widetilde{\sigma}(x') := (4,1,2,3,1,2) \ . \tag{2b}$$

We now consider a pair of vertices (x,x'), $x,x' \in D_{2n}^0$, of type 2 as defined above. Using again the definition (1), for such vertices we have $\sigma(x)=(s,|v_1|+1+(4,2,3,1,2,3,-1,4))$ and $\sigma(x')=(s,|v_1|+1+(2,1,-1,2,4,3,1,4))$, where s is a prefix of length $2|v_1|+2$, and we define

$$\widetilde{\sigma}(x) := (s,|v_1|+1+(4,1,3,2,1,3,-1,4)) \ , \tag{2c}$$

$$\widetilde{\sigma}(x') := (s,|v_1|+1+(2,1,-1,2,4,3,2,4)) \ . \tag{2d}$$

We let $P_{\widetilde{\sigma}}(x)$ denote the path obtained by flipping in x the bits in $\widetilde{\sigma}(x)$ one after the other. Moreover, we let $\widetilde{\mathcal{P}}_{2n}(n,n+1)$ denote the set of all paths $P_{\widetilde{\sigma}}(x)$ for which $x \in D_{2n}^0$ is part of a type 1 or type 2 pair of vertices. It is readily checked that for pairs of vertices (x,x') of type 1 or type 2, the paths $(P_\sigma(x),P_\sigma(x'))$ and $(P_{\widetilde{\sigma}}(x),P_{\widetilde{\sigma}}(x'))$ obtained from $\widetilde{\sigma}$ as defined in (2) indeed satisfy the conditions for flippable and flipped pairs of paths mentioned before, respectively. The straightforward direct definition of flippable pairs given before is indeed a considerable simplification in the proof of Theorem 1 compared to the complicated inductive definition from [Müt16] and the unwieldy recursive definition from [MN15] (the function PATHS() called with parameter $\texttt{flip} = \texttt{true}$).

2.4. **The Hamilton cycle algorithm.** In this section we present our algorithm to compute a Hamilton cycle in the middle levels graph $Q_{2n+1}(n, n+1)$. The Hamilton cycle is obtained by combining the paths $\mathcal{P}_{2n}(n, n+1)$ and $\widetilde{\mathcal{P}}_{2n}(n, n+1)$ in the graph $Q_{2n}(n, n+1)$ that can be computed via the bitflip sequences σ and $\widetilde{\sigma}$, respectively. Specifically, the algorithm is based on the following decomposition of the middle levels graph $Q_{2n+1}(n, n+1)$ (see Figure 2). For any graph G whose vertices are bitstrings and any bitstring x, we write $G \circ x$ for the graph obtained by attaching x to every vertex (=bitstring) of G. Partitioning the vertices of the middle levels graph $Q_{2n+1}(n, n+1)$ according to the value of the last bit, we observe that it consists of a copy of $Q_{2n}(n, n+1) \circ 0$ and a copy of $Q_{2n}(n-1, n) \circ 1$, plus the set of edges $M_{2n+1} = \{((x, 0), (x, 1)) \mid x \in B_{2n}(n)\}$ (these edges form a matching between the vertex sets $B_{2n}(n) \circ 0$ and $B_{2n}(n) \circ 1$). Observe that the graphs $Q_{2n}(n, n+1)$ and $Q_{2n}(n-1, n)$ are isomorphic and that the mapping $\overline{\text{rev}}$ is an isomorphism between them. By property (ii) from Section 2.2 the sets of first and last vertices of the paths $\mathcal{P}_{2n}(n, n+1)$ defined in Section 2.2 are given by D_{2n}^0 and D_{2n}^1, respectively. It is easy to check that these two sets are preserved under the mapping $\overline{\text{rev}}$, from which we conclude with the help of property (i) from Section 2.2 that

$$\mathcal{C}_{2n+1} := \mathcal{P}_{2n}(n, n+1) \circ 0 \ \cup \ \overline{\text{rev}}(\mathcal{P}_{2n}(n, n+1)) \circ 1 \ \cup \ M'_{2n+1} \tag{3}$$

with $M'_{2n+1} := \{((x, 0), (x, 1)) \mid x \in D_{2n}^0 \cup D_{2n}^1\} \subseteq M_{2n+1}$ is a *2-factor* in the middle levels graph, i.e., a set of disjoint cycles that together visit all vertices of the graph. Note that along each of the cycles in the 2-factor, the paths from $\mathcal{P}_{2n}(n, n+1) \circ 0$ are traversed in forward direction, and the paths from $\overline{\text{rev}}(\mathcal{P}_{2n}(n, n+1)) \circ 1$ in backward direction. Observe also that the definition of flippable and flipped pairs of paths given in Section 2.3 allows us to replace in the definition (3) any pair of flippable paths from $\mathcal{P}_{2n}(n, n+1)$ by the corresponding flipped paths from $\widetilde{\mathcal{P}}_{2n}(n, n+1)$, and the resulting subgraph of $Q_{2n+1}(n, n+1)$ will again be a 2-factor, albeit with a different cycle structure. Specifically, if two paths that form a flippable pair lie on two different cycles, then replacing them with the corresponding flipped paths will join the two cycles to one cycle. Our final algorithm makes all those choices such that the resulting 2-factor consists only of a single cycle, i.e., a Hamilton cycle.

To call the algorithm HAMCYCLE() described in Algorithm 1, we need to provide three input parameters: n determines the dimension of the cube (we want a Hamilton cycle in $Q_{2n+1}(n, n+1)$), x is the starting vertex of the cycle (it is a bitstring of length $2n + 1$ with weight n or $n + 1$), and ℓ is the number of vertices to visit along the Hamilton cycle. The local variable y is the current vertex along the Hamilton cycle, and the local variable i counts the number of vertices that have already been visited. The calls VISIT(y) in lines H9, H12, H18 and H21 indicate where a function using our Hamilton cycle algorithm could perform further operations on the current vertex y. Each time a vertex along the Hamilton cycle is visited, we increment i and check whether the desired number ℓ of vertices has been visited (lines H10, H13, H19 and H22). Let us postpone the definition of the functions INIT() and FLIPTREE() called in lines H1 and H4 a bit, and let us assume for a moment that the input vertex x of the middle levels graph $Q_{2n+1}(n, n+1)$ has the form $x = (x', 0)$ with $x' \in D_{2n}^0$. In this case the variables y and i will be initialized to $y := x$ and $i := 1$ in line H1. Let us also neglect for the moment all operations involving the variable T, and let us assume that the return value of FLIPTREE() called in line H4 is always `false`. With these simplifications, the algorithm HAMCYCLE() computes exactly the 2-factor $\mathcal{C}_{2n+1}$ defined in (3) in the middle levels graph $Q_{2n+1}(n, n+1)$. Indeed, one complete execution of the first for-loop corresponds to following one path from the set $\mathcal{P}_{2n}(n, n+1) \circ 0$ (the first set on the right hand side of (3)) in the graph $Q_{2n}(n, n+1) \circ 0$ starting at its first vertex and ending at its last vertex, and one complete execution of the second for-loop corresponds to following one path from the set $\overline{\text{rev}}(\mathcal{P}_{2n}(n, n+1)) \circ 1$ (the

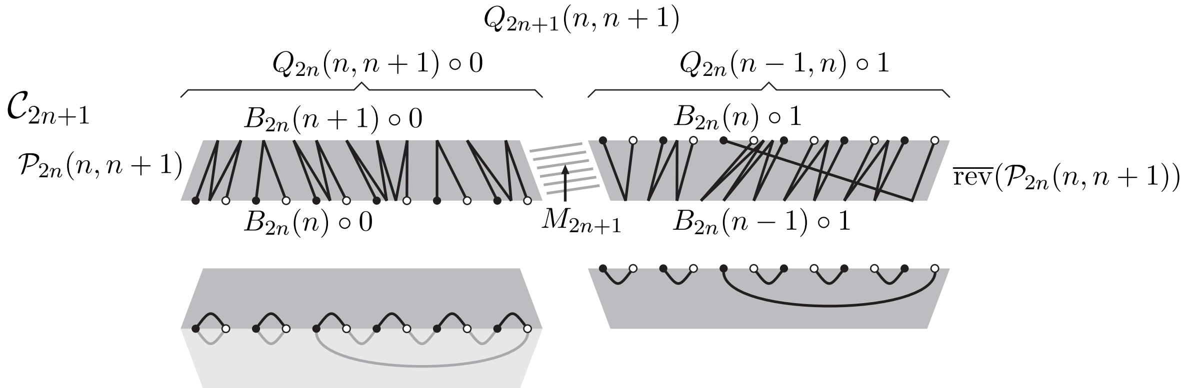


FIGURE 2. The top part of the figure shows the decomposition of the middle levels graph and the definition (3). In this figure the 2-factor $\mathcal{C}_{2n+1}$ consists of three disjoint cycles that together visit all vertices of the graph. The first and last vertices of the paths (belonging to $\mathcal{P}_{2n}(n,n+1)$ and to $\overline{\mathrm{rev}}(\mathcal{P}_{2n}(n,n+1))$) are drawn in black and white, respectively. The bottom part of the figure shows a simplified drawing that helps analyzing the cycle structure of the 2-factor (it has two short cycles and one long cycle).

Algorithm 1: HAMCYCLE(n,x,ℓ)

Input: An integer $n\geq 1$, a vertex $x\in Q_{2n+1}(n,n+1)$, an integer $\ell\geq 1$
Result: Starting from the vertex x, the algorithm visits the next ℓ vertices on a Hamilton cycle in $Q_{2n+1}(n,n+1)$

$(T,y,i):=$ INIT(n,x,ℓ)
while `true` **do**
  $y^-:=(y_1,y_2,\ldots,y_{2n})$ `/* ignore last bit of` y `*/`
  if FLIPTREE$(T)=$ `false` **then** $s:=\sigma(y^-)$ `/* compute bitflip sequence` σ `... */`
  else $s:=\widetilde{\sigma}(y^-)$ `/* ... or` $\widetilde{\sigma}$ `*/`
  $T:=\mathrm{rot}(T)$ `/* rotate state tree` T `*/`
  for $j:=1$ **to** $|s|$ **do** `/* flip bits according to sequence` s `*/`
    $y_{s_j}:=1-y_{s_j}$ `/* flip bit at position` s_j `*/`
    VISIT(y)
    if $(i:=i+1)=\ell$ **return**
  $y_{2n+1}:=1$ `/* flip last bit to 1 */`
  VISIT(y)
  if $(i:=i+1)=\ell$ **return**
  $(u,0,1,v):=(y_1,y_2,\ldots,y_{2n})$ `/* canonical decomposition of` $(y_1,\ldots,y_{2n})\in D_{2n}^1$ `*/`
  $s:=\sigma((1,\overline{\mathrm{rev}}(\pi(v)),0,\overline{\mathrm{rev}}(u)))$ `/* compute bitflip sequence` σ `*/`
  for $j:=|s|$ **downto** 1 **do** `/* flip bits according to reverse sequence` s `*/`
    $y_{2n+1-s_j}:=1-y_{2n+1-s_j}$ `/* flip bit at position` $2n+1-s_j$ `*/`
    VISIT(y)
    if $(i:=i+1)=\ell$ **return**
  $y_{2n+1}:=0$ `/* flip last bit to 0 */`
  VISIT(y)
  if $(i:=i+1)=\ell$ **return**

---

second set on the right hand side of (3)) in the graph $\overline{\text{rev}}(Q_{2n}(n,n+1))\circ 1$ starting at its last vertex and ending at its first vertex. At the intermediate steps in lines H11 and H20, the last bit is flipped. These flips correspond to traversing an edge from the matching M'_{2n+1} which constitutes the third set on the right hand side of (3). The paths $\mathcal{P}_{2n}(n,n+1)$ are computed in lines H4 and H15 using the recursion σ, and the resulting bitflip sequences are applied in the two inner for-loops (specifically, in line H8 and H17). Note that if a path P from the set $\mathcal{P}_{2n}(n,n+1)$ has $y\in D_{2n}^1$ as last vertex and if $(u,0,1,v)$ is the canonical decomposition of y, then $\overline{\text{rev}}$ maps the last vertex of the path $P'\in\mathcal{P}_{2n}(n,n+1)$ that has $(1,\overline{\text{rev}}(\pi(v)),0,\overline{\text{rev}}(u))$ as first vertex onto y. This is a consequence of property (ii) from Section 2.2, from which we obtain that the last vertex of P' is $(\overline{\text{rev}}(v),0,1,\overline{\text{rev}}(u))$ (note that $\pi(\overline{\text{rev}}(\pi(v)))=\overline{\text{rev}}(\pi(\pi(v)))=\overline{\text{rev}}(v)$), and applying $\overline{\text{rev}}$ to this vertex indeed yields y. From these observations and the definitions in lines H14, H15 and H17 it follows that the paths in the second set on the right hand side of (3) are indeed traversed in reverse order by our algorithm (starting at the last vertex and ending at the first vertex).

We proceed to explain the significance of the function FlipTree() called in line H4 and the variable T in our algorithm. The variable T is an ordered rooted tree with n edges (we give the precise definition of these trees in the next section), and captures high-level state information of our algorithm. The state tree T is possibly modified in the call FlipTree(T), and whenever the boolean return value of this function is `true`, then instead of using the bitflip sequence σ to compute a path from $\mathcal{P}_{2n}(n,n+1)$ in line H4, the bitflip sequence $\widetilde{\sigma}$ is used to compute a path from $\widetilde{\mathcal{P}}_{2n}(n,n+1)$ in line H5. Consequently, the function FlipTree() controls which flippable pairs of paths from $\mathcal{P}_{2n}(n,n+1)$ are replaced by the corresponding flipped paths in $\widetilde{\mathcal{P}}_{2n}(n,n+1)$ in the first set on the right hand side of (3) so that the resulting 2-factor in the middle levels graph $Q_{2n+1}(n,n+1)$ is a Hamilton cycle. Observe that these modifications do not apply to the paths in the second set on the right hand side of (3): For the corresponding instructions in the second for-loop, only the paths in $\mathcal{P}_{2n}(n,n+1)$ computed via σ in line H15 are relevant. Note that the variable T is also modified in line H6 in each iteration of the while-loop (the definition of the rotation operation rot() is given in the next section).

The function FlipTree() therefore nicely encapsulates the core 'intelligence' of our Hamilton cycle algorithm. The definition of this function is rather technical, and in this informal description of the algorithm we make no attempt to justify its correctness, but restrict ourselves to giving the definition in the next section (the correctness proof is provided in Section 3 below). The function Init() will be defined and discussed in Section 2.6 below.

2.5. **The function FlipTree().** The function FlipTree() is essentially the same as the function IsFlipVertex() from [MN15]. The crucial difference is that the function FlipTree() modifies and returns the state tree T that is provided as a function argument, and also returns a boolean value from $\{\texttt{true},\texttt{false}\}$ (whereas IsFlipVertex() recomputes the state tree on each call from the vertex that is provided as an argument). To define the function FlipTree(), we begin by introducing a few definitions.

Ordered rooted trees. An *ordered rooted tree* is a rooted tree where the children of each vertex have a specified left-to-right ordering. We think of an ordered rooted tree as a tree embedded in the plane with the root on top, with downward edges leading from any vertex to its children, and the children appear in the specified left-to-right ordering (see the right hand side of Figure 3, the root vertex is drawn boldly). We denote by $\mathcal{T}_n^*$ the set of all ordered rooted trees with n edges.

Bijection between lattice paths and ordered rooted trees. We identify each bitstring/lattice path $x=(x_1,\ldots,x_{2n})\in D_{2n}^0$ with an ordered rooted tree from the set $\mathcal{T}_n^*$ as follows (see the right hand side of Figure 3 and [Sta99, Exercise 6.19 (e)]): Starting with a tree that has only a root vertex, we read the bits of x from left to right, and for every 1-step we add a new rightmost child to the current

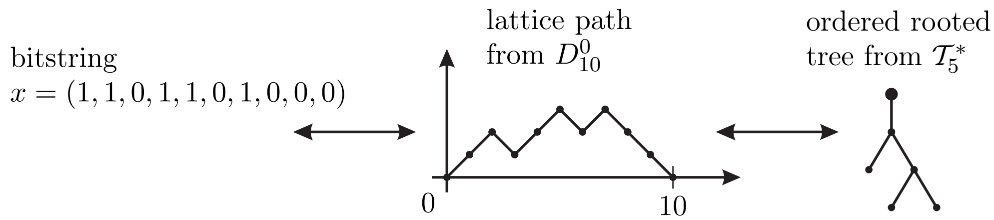


Figure 3. Bijections between bitstrings and lattice paths (left hand side), and between lattice paths from D_{2n}^0 and ordered rooted trees from $\mathcal{T}_n^*$ (right hand side).

vertex and move to this child, for every 0-step we add no edge but simply move back to the parent of the current vertex (we clearly end up back at the root). This construction defines a bijection between the bitstrings/lattice paths D_{2n}^0 and the ordered rooted trees $\mathcal{T}_n^*$, and in the following we will repeatedly switch between these two representations.

Operations on ordered rooted trees. In the following we need several more definitions and concepts introduced in [MN15]. To save space and to avoid duplication, we refer to Section 2.5 in that paper for the definitions. Specifically, we will need the definitions of *thin/thick leaves*, *clockwise/counterclockwise-next leaves*, of certain sets of trees $\mathcal{T}_{n,1}^*, \mathcal{T}_{n,2}^* \subseteq \mathcal{T}_n^*$, and of tree transformations τ_1 and τ_2 on the sets $\mathcal{T}_{n,1}^*$, $\mathcal{T}_{n,2}^*$, respectively. Moreover, we need the definitions of the bijection h on the set D_{2n}^0 (which can be interpreted as a bijection on $\mathcal{T}_n^*$ using the correspondence between lattice paths and ordered rooted trees mentioned before), and the operation rot() of rotating an ordered rooted tree.

Algorithm 2: FlipTree(T)

Input: An ordered rooted tree $T \in T_n^*$
Output: {`true`, `false`}, the tree T may be modified (passed by reference)

T1 $r :=$ `false` `/* initialize return value */`
T2 **if** ($T \in \mathcal{T}_{n,1}^*$ and IsFlipTree$_1(T) =$ `true`) **then** $T := \tau_1(T)$ and $r :=$ `true`
T3 **else if** ($T \in \tau_1(\mathcal{T}_{n,1}^*)$ and IsFlipTree$_1(\tau_1^{-1}(T)) =$ `true`) **then** $T := \tau_1^{-1}(T)$ and $r :=$ `true`
T4 **else if** ($T \in \mathcal{T}_{n,2}^*$ and IsFlipTree$_2(T) =$ `true`) **then** $T := \tau_2(T)$ and $r :=$ `true`
T5 **else if** ($T \in \tau_2(\mathcal{T}_{n,2}^*)$ and IsFlipTree$_2(\tau_2^{-1}(T)) =$ `true`) **then** $T := \tau_2^{-1}(T)$ and $r :=$ `true`
T6 **return** r

With these definitions at hand, consider now the definition of the function FlipTree(T). We emphasize again that the state tree T passed as a function argument is passed by reference, so the state tree may be modified by this function. On lines T2–T3 the function checks whether T is a preimage or image under τ_1 and calls the auxiliary function IsFlipTree$_1$() (defined below) with T or $\tau_1^{-1}(T)$ as an argument, respectively. A similar check is performed in lines T4–T5 with respect to τ_2, using the auxiliary function IsFlipTree$_2$() (defined below). Whenever one of these conditions is satisfied, then the state tree T is modified accordingly and the function FlipTree(T) returns `true`. Note that the sets of trees $\mathcal{T}_{n,1}^*, \tau_1(\mathcal{T}_{n,1}^*), \mathcal{T}_{n,2}^*$ and $\tau_2(\mathcal{T}_{n,2}^*)$ are all disjoint, so T is contained in at most one of them. Furthermore, observe that the state tree T is also modified (unconditionally) in each iteration of the while-loop of the algorithm HamCycle(), by applying one rotation operation to it (line H6).

The auxiliary functions IsFlipTree$_1$() and IsFlipTree$_2$() achieve exactly the same functionality as the corresponding two functions with the same names in [MN15]. However, to speed up their running time from quadratic to linear in n (recall the discussion from Section 1.2), we use slightly more sohisticated data structures and algorithms to store and manipulate ordered rooted trees (the data structures are explained in detail in Section 4).

To define these two functions, we need to define another auxiliary function $\textsc{Root}()$ which is called from within $\textsc{IsFlipTree}_1()$ and $\textsc{IsFlipTree}_2()$ and which has the property that for any tree $T \in \mathcal{T}_n^*$, for all integers $i \geq 0$ the return value of $\textsc{Root}(\mathrm{rot}^i(T))$ is the same rotated version of T (no matter which rotated version of T is provided as an input). In other words, the function $\textsc{Root}(T)$ computes a canonically rooted version of T.

The function $\textsc{Root}(T)$*:* Given a tree $T \in \mathcal{T}_n^*$, first compute the center vertex/vertices c_1, c_2 of T ($c_1 = c_2$ if the center is unique), i.e., the vertex/vertices that minimizes the maximum distance to any other vertex. If there are two centers ($c_1 \neq c_2$), then compute T_1' as the tree obtained by rooting T so that c_1 is the root and c_2 its leftmost child, compute T_2' as the tree obtained by rooting T so that c_2 is the root and c_1 its leftmost child, and return the tree from $\{T_1', T_2'\}$ with the lexicographically smaller bitstring representation. If the center is unique ($c_1 = c_2$), then let $T_1, T_2, \ldots, T_k$ be the subtrees of T rooted at $c_1 = c_2$. Compute the bitstring representations $x_1, x_2, \ldots, x_k$ for each of them, and compute the lexicographically smallest string rotation of $(-1, x_1, -1, x_2, -1, \ldots, -1, x_k)$ using Booth's algorithm [Boo80] (-1 is an additional symbol that is lexicographically smaller than 0 and 1, ensuring that the lexicographically smallest string rotation starts at a tree boundary). Let $\widehat{T}$ be the tree obtained by rooting T at c_1 such that the subtrees $T_1, \ldots, T_k$ appear exactly in this lexicographically smallest ordering, and return $\widehat{T}$.

The function $\textsc{IsFlipTree}_1(T)$*:* Given a tree $T \in \mathcal{T}_{n,1}^*$, compute $\widehat{T} := \textsc{Root}(T)$ and let T' be the tree obtained by rotating $\widehat{T}$ until it is a preimage of τ_1 for the first time. Return `true` if $T = T'$ (compare the bitstring representations of T and T'), and `false` otherwise.

The function $\textsc{IsFlipTree}_2(T)$*:* Given a tree $T \in \mathcal{T}_{n,2}^*$, if the bitstring representation of T has the form $T = (1^{n-1}, 0^{n-2}, 1, 0, 0)$ (the exponents indicate the number of repetitions of the bits 0 or 1) or if T has more than one thin leaf, return `false`. Otherwise, let v be the unique thin leaf of T, v' the parent of v, v'' the parent of v', and w the clockwise-next leaf of v, and let T' be the tree obtained from T by replacing the edge (v, v') by (v, v'') so that v becomes a child of v'' to the left of v' (so T' has only thick leaves), and by rotating it such that the leaf w becomes the root. Let d be the distance between the root w and the leftmost leaf v of T'. Compute all other rotated versions of T' for which the root is a leaf and the parent of the leftmost leaf x has another leaf as its child to the right of x (initially, v and v' are these two leaves). For each of them compute the distance d' between the root and the leftmost leaf, and return `false` if $d' > d$ for one of them. Otherwise, compute $\widehat{T} := \textsc{Root}(T')$, and let T'' be the tree obtained by rotating $\widehat{T}$ until for the first time the root is a leaf, the parent of the leftmost leaf x has another leaf as its child to the right of x, and the distance from the root to x is d. Return `true` if $T' = T''$ (compare the bitstring representations of T' and T''), and `false` otherwise.

2.6. **The function** $\textsc{Init}()$**.** The function $\textsc{Init}(n, x, \ell)$ called in line H1 does the following: First check whether the input vertex x of the middle levels graph $Q_{2n+1}(n, n+1)$ has the form $x = (x', 0)$ with $x' \in D_{2n}^0$. If this is the case then initialize the state tree as $T := h^{-1}(x')$ (with h as defined in the previous section), the current vertex as $y := x$ and the vertex counter as $i := 1$. Otherwise, move backwards along the Hamilton cycle to the previous vertex which has this special form using the $\textsc{Paths}()$ algorithm from [MN15] (which is an equivalent recursive description of the paths $\mathcal{P}_{2n}(n, n+1)$). Then move forward along the Hamilton cycle by essentially executing one iteration of the while-loop of the algorithm $\textsc{HamCycle}()$ until x is encountered again (with the crucial difference that the vertices on the cyle before x are *not* visited by $\textsc{Visit}()$), and further until the next vertex z which has the form $z = (z', 0)$ with $z' \in D_{2n}^0$ is encountered. Then initialize the state tree as $T := h^{-1}(z')$, the current vertex as $y := z$ and set the vertex counter to the number of vertices visited between x and z. For details how this is implemented, we refer to the comments of our C++ implementation [www]. By what we said before, we could completely get rid of calling the recursive Paths() algorithm from [MN15] by imposing the additional restriction that the starting vertex x of our Hamilton cycle has the form $x = (x', 0)$ with $x' \in D_{2n}^0$, e.g., by always starting at $x = (1^n, 0^{n+1})$. Given that the graph $Q_{2n+1}(n, n+1)$ is vertex-transitive, this is a rather weak restriction.

3. Correctness of the algorithm

In this section we prove the correctness of our algorithm to compute a Hamilton cycle in the middle levels graph. We begin by proving that the paths computed by the recursions σ and $\widetilde{\sigma}$ satisfy all the properties claimed in Section 2.2 and 2.3. The correctness proof of the higher-level functions FlipTree() and HamCycle() can be taken almost verbatim from the proof provided in [MN15, Section 3].

3.1. **Correctness of the paths computed by σ.**

This section is devoted to proving properties (i) and (ii) claimed in Section 2.2. The main result of this section, Lemma 6 below, shows that the set of paths $\mathcal{P}_{2n}(n, n+1)$ defined in Section 2.2 via the recursive bitflip rule σ is exactly the same as the paths defined inductively in [Müt16] (and, as was shown in [MN15, Lemma 4], these are the same paths as computed by the Paths() algorithm from that paper).

We begin by extending some terminology concerning lattice paths. Given a lattice path $x = (x_1, x_2, \ldots, x_{2n}) \in D_{2n}^0$ and its canonical decomposition $x = (1, u, 0, v)$, we say that the pair of integers (i, j), $i, j \in \{2, 3, \ldots, |u|+1\}$, forms a *base pair of* x, if $x_i = 1$, $x_j = 0$ and the subpath $(x_{i+1}, x_{i+2}, \ldots, x_{j-1})$ of x is contained in the set D_{j-i-1}^0. In words, the i-th step of x is a 1-step, the j-th step a 0-step, and the subpath in between these two steps forms a hill (x_i and x_j are the bases of the hill). Figure 4 shows exemplarily three base pairs in red, green and blue, respectively. Clearly, the base pairs for x form a partition of all indices $\{2, 3, \ldots, |u|+1\}$ into pairs and the number of these pairs equals the number of recursive calls to σ' when computing $\sigma(x)$ as defined in (1). (Note that $(1, |u|+2)$ is *not* a base pair, so the first call $\sigma(x)$ is not counted.) Specifically, the base pair (i, j) of x corresponds to the canonical decomposition of the subpath $x' \in D_{2n'}^0$ of x that starts at index i and has the property $x_{i+2n'} = 0$ when computing $\sigma'(x')$ via (1b). Consequently, there is a one-to-one correspondence between the calls to σ' and the base pairs of x.

The next lemma is an alternative definition of the bitflip sequence $\sigma(x)$, $x \in D_{2n}^0$, introduced in Section 2.2. It follows immediately from the definition (1).

Lemma 4. *Let $x \in D_{2n}^0$ and consider the bitflip sequence $s := \sigma(x)$ defined in* (1)*. For any base pair (i, j) of x we have*

$$s_{2i-1} = j \ , \tag{4a}$$
$$s_{2i} = i + a \ , \tag{4b}$$
$$s_{2j-1} = i - 1 + b \ , \tag{4c}$$
$$s_{2j} = j + c \ , \tag{4d}$$

where a, b, c are as defined in (1c).

Note that Lemma 4 does not mention the first two entries of the sequence $s = \sigma(x)$, which are given by $s_1 = |u| + 2$ and $s_2 = 1$ by the definition (1a).

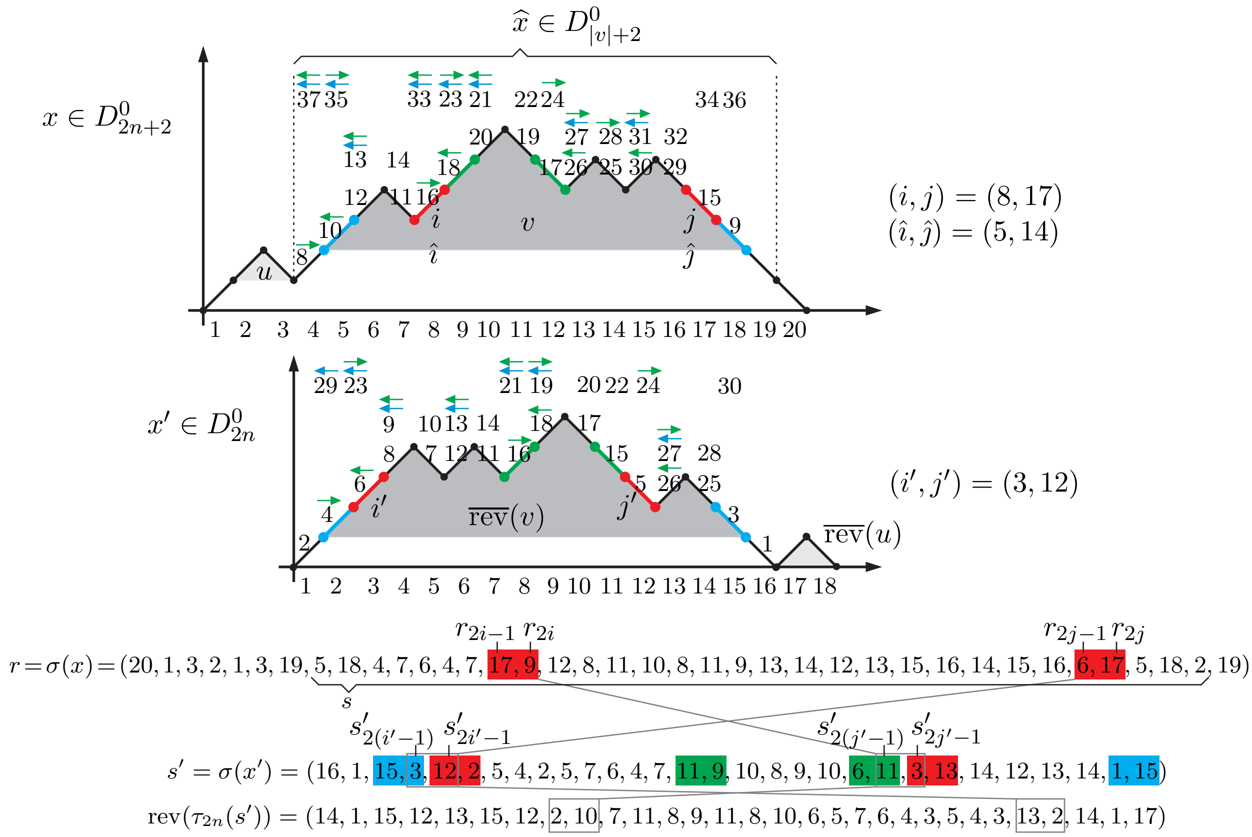


FIGURE 4. Illustration of the notations used in the statement and proof of Lemma 5. The small green and blue arrows have the same meaning as in Figure 1. The two red, green and blue pairs of steps form base pairs in the lattice paths x and x'. The corresponding four entries in s and s' are highlighted in the respective colors. In this example, we have $|u| = 2$, $|v| = 14$ and $n = 9$, and the sum of any two corresponding entries of s and $\mathrm{rev}(\tau_{2n}(s'))$ is $2n + 1 = 19$.

For integers $n \geq 1$ and $i \in \{1, 2, \ldots, 2n\}$ we define

$$\tau_{2n}(i) := \begin{cases} i+1 & \text{if } i < 2n \text{ is even }, \\ i-1 & \text{if } i > 1 \text{ is odd }, \\ i & \text{otherwise }. \end{cases} \tag{5}$$

and for an integer sequence $s = (s_1, \ldots, s_k)$ we define $\tau_{2n}(s) := (\tau_{2n}(s_i))_{i=1,\ldots,k}$.

Lemma 5. *Let $n \geq 1$ and $u \in D^0_{|u|}$ and $v \in D^0_{|v|}$ be such that $|u|+|v|+2 = 2n$. Moreover, consider the lattice paths $\widehat{x} := (1, v, 0) \in D^0_{|v|+2}$, $x := (1, u, \widehat{x}, 0) \in D^0_{2n+2}$ and $x' := (1, \overline{\mathrm{rev}}(v), 0, \overline{\mathrm{rev}}(u)) \in D^0_{2n}$. Then the sequence s obtained from the subsequence $1 + |u| + \sigma'(\widehat{x})$ of $\sigma(x)$ by removing the first and last entry and the sequence $s' := \sigma(x')$ satisfy the relation $s = 2n + 1 - \mathrm{rev}(\tau_{2n}(s'))$.*

For the reader's convenience, Lemma 5 is illustrated in Figure 4.

The proof of Lemma 5 is rather technical due to the parameters a, b and c in the definition (1c) which entail many case distinctions.

Proof. Throughout the proof, we will repeatedly use that $|u|$ and $|v|$ are even numbers. First note that both sequences s and s' have the same length $2|v| + 2$. We begin proving that the first and last elements of the sequences s and s' satisfy the claimed relation. The subpath $\widehat{x}$ starts at an odd index in x, so by the definition (1a) the first and last entry of s are given by

$$s_1 = (|u|+1) + 1 + \mathbf{1}_{|v|>0} = |u| + 2 + \mathbf{1}_{|v|>0} \ , \tag{6}$$

$$s_{2|v|+2} = (|u|+1) - \mathbf{1}_{|u|>0} \ , \tag{7}$$

where $\mathbf{1}_{\{|v|>0\}}, \mathbf{1}_{\{|u|>0\}} \in \{0,1\}$ are indicator functions that evaluate to 1 if and only if $|v|>0$ or $|u|>0$, respectively. Also by the definition (1a), the first and last entry of s' are

$$s'_1 = |v| + 2 \ , \tag{8}$$

$$s'_{2|v|+2} = |v| + 1 \ , \tag{9}$$

respectively. Using the assumption

$$|u| + |v| + 2 = 2n \ , \tag{10}$$

we therefore obtain that the first and last entry of $2n+1-\mathrm{rev}(\tau_{2n}(s'))$ are

$$2n+1-\tau_{2n}(s'_{2|v|+2}) \overset{(9),(5)}{=} 2n+1-(|v|+1-\mathbf{1}_{|v|>0}) \overset{(10),(6)}{=} s_1 \ ,$$

$$2n+1-\tau_{2n}(s'_1) \overset{(8),(5)}{=} 2n+1-(|v|+2+\mathbf{1}_{|u|>0}) \overset{(10),(7)}{=} s_{2|v|+2} \ ,$$

respectively, as claimed.

For the rest of the proof we consider the remaining entries of s and s'. We fix a base pair (i,j) in x that belongs to the subpath v (which is a subpath of $\widehat{x}$), and we consider the corresponding base pair

$$(\hat{\imath},\hat{\jmath}) := (i,j) - (|u|+1) \ , \tag{11}$$

$\hat{\imath},\hat{\jmath} \in \{2,3,\ldots,|v|+1\}$, normalized to indices starting at 1 in the subpath $\widehat{x}$ (this base pair is shown in red at the top of Figure 4). Defining $r := \sigma(x)$ we clearly have $s_k = r_{k+2|u|+3}$. To see this recall that the sequence s is obtained from r by removing the first two entries corresponding to the first 1-step and the last 0-step of x, by removing the next $2|u|$ entries corresponding to the subpath u of x, and by removing one more entry corresponding to the last 0-step of $\widehat{x}$ (in total $2|u|+3$ entries are removed from the beginning), plus removing the last entry (which is irrelevant here). Using this relation and the definition (11) we obtain that $s_{2\hat{\imath}+k} = r_{2i+k+1}$ and $s_{2\hat{\jmath}+k} = r_{2j+k+1}$, which together with Lemma 4 yields

$$s_{2\hat{\imath}-2} = r_{2i-1} = j \ , \tag{12a}$$

$$s_{2\hat{\imath}-1} = r_{2i} \quad = i + a \ , \tag{12b}$$

$$s_{2\hat{\jmath}-2} = r_{2j-1} = i - 1 + b \ , \tag{12c}$$

$$s_{2\hat{\jmath}-1} = r_{2j} \quad = j + c \ , \tag{12d}$$

where a, b and c are defined in (1c) with respect to the lattice path x (these four entries of s are highlighted in red in Figure 4). We proceed to show that the four entries of s referred to in (12) satisfy the claimed relation with the corresponding four entries of the sequence s'. Since for each entry s_k, $k \in \{2,3,\ldots,2|v|+1\}$, there is a unique base pair (i,j) in x such that s_k appears on the left hand side of one of the equations in (12), this will prove the lemma. Note that $s = 2n+1-\mathrm{rev}(\tau_{2n}(s'))$ if and only if $s + \tau_{2n}(\mathrm{rev}(s')) = 2n+1$ (the mappings τ_{2n} and rev commute), so to complete the proof it suffices to show that for any $\ell \in \{2\hat{\imath}-2, 2\hat{\imath}-1, 2\hat{\jmath}-2, 2\hat{\jmath}-1\}$ we have $s_\ell + \tau_{2n}(s'_{2|v|+3-\ell}) = 2n+1$.

The first thing to observe is that since $(\hat{\imath},\hat{\jmath})$ is a base pair in $\widehat{x} = (1,v,0)$, the pair

$$(i',j') := |v| + 3 - (\hat{\jmath},\hat{\imath}) \tag{13}$$

is a base pair in $x' = (1, \overline{\mathrm{rev}}(v), 0, \overline{\mathrm{rev}}(u))$ (the base pair (i',j') is shown in red in the middle of Figure 4). We now distinguish the four above-mentioned cases for the variable ℓ. For the following computations the reader should keep in mind that when applying τ_{2n}, we always have $2 \le \hat{\imath} \le |v|$

and $3 \leq \hat{\jmath} \leq |v|+1$, and moreover $\hat{\imath} \geq 3$ for odd $\hat{\imath}$, and $\hat{\jmath} \leq |v|$ for even $\hat{\jmath}$. For the following we let a', b' and c' be the values of the parameters defined in (1c) with respect to the lattice path x'.

Case 1: $\ell = 2\hat{\imath}-2$. In Figure 4, the ℓ-th entry of s is the first red entry of s. The relevant entry of the sequence s' in this case is $s'_{2|v|+3-\ell} = s'_{2(|v|+3-\hat{\imath})-1}$. As (i',j') is a base pair in x', we obtain from Lemma 4 that

$$s'_{2(|v|+3-\hat{\imath})-1} \overset{(13)}{=} s'_{2j'-1} \overset{(4c)}{=} i'-1+b' \overset{(13)}{=} |v|+2-\hat{\jmath}+b' \ . \tag{14}$$

If $\hat{\jmath} = |v|+1$, then

$$s_{2\hat{\imath}-2} \overset{(12a),(11)}{=} \hat{\jmath}+|u|+1 = |u|+|v|+2 \overset{(10)}{=} 2n \ , \tag{15}$$

and by (13) we are considering the base pair $(i',j') = (2, |v|+3-\hat{\imath})$ in x', implying that we have $b'=0$ in (14) and therefore

$$\tau_{2n}(s'_{2(|v|+3-\hat{\imath})-1}) \overset{(14)}{=} \tau_{2n}(1) \overset{(5)}{=} 1 \ . \tag{16}$$

Summing the right hand sides of (15) and (16) yields $2n+1$, as desired.

On the other hand, if $\hat{\jmath} < |v|+1$ then $\hat{\jmath}$ and $i' = |v|+3-\hat{\jmath}$ have opposite parity and therefore $b'=-1$ if $\hat{\jmath}$ is odd and $b'=1$ if $\hat{\jmath}$ is even. In both cases we obtain (independently of the parity of $\hat{\jmath}$)

$$\tau_{2n}(s'_{2(|v|+3-\hat{\imath})-1}) \overset{(14),(5)}{=} |v|+2-\hat{\jmath} \tag{17}$$

and therefore

$$s_{2\hat{\imath}-2} + \tau_{2n}(s'_{2(|v|+3-\hat{\imath})-1}) \overset{(12a),(17)}{=} j-\hat{\jmath}+|v|+2 \overset{(11),(10)}{=} 2n+1 \ ,$$

as claimed.

Case 2: $\ell = 2\hat{\jmath}-2$. In Figure 4, the ℓ-th entry of s is the third red entry of s. The relevant entry of the sequence s' in this case is $s'_{2|v|+3-\ell} = s'_{2(|v|+3-\hat{\jmath})-1}$. As (i',j') is a base pair in x', we obtain from Lemma 4 that

$$s'_{2(|v|+3-\hat{\jmath})-1} \overset{(13)}{=} s'_{2i'-1} \overset{(4a)}{=} j' \overset{(13)}{=} |v|+3-\hat{\imath} \ . \tag{18}$$

Note that the value of b in (12c) depends on the base pair (i,j), specifically, on the parity of i, and since i and $\hat{\imath}$ have opposite parity (recall (11)), we obtain from (1c) that $b=1$ if $\hat{\imath}$ is even and $b=-1$ if $\hat{\imath}$ is odd (the case $b=0$ does not occur as $i \geq 3$). Consequently, if $\hat{\imath}$ is even we have

$$s_{2\hat{\jmath}-2} + \tau_{2n}(s'_{2(|v|+3-\hat{\jmath})-1}) \overset{(12c),(18),(5)}{=} (i-1+1)+(|v|+3-\hat{\imath}-1) = i-\hat{\imath}+|v|+2 \overset{(11),(10)}{=} 2n+1 \ .$$

If $\hat{\imath}$ is odd then the middle term of the previous calculation changes to $(i-1-1)+(|v|+3-\hat{\imath}+1)$, yielding the same result. This proves the claim in this case.

Case 3: $\ell = 2\hat{\imath}-1$. In Figure 4, the ℓ-th entry of s is the second red entry of s. The relevant entry of the sequence s' in this case is $s'_{2|v|+3-\ell} = s'_{2(|v|+2-\hat{\imath})}$. Here we distinguish the subcases (i) $\hat{\jmath} = \hat{\imath}+1$ and (ii) $\hat{\jmath} > \hat{\imath}+1$.

Case 3 (i): $\hat{\jmath} = \hat{\imath}+1$. As (i',j') is a base pair in x', we obtain from Lemma 4 that

$$s'_{2(|v|+2-\hat{\imath})} = s'_{2(|v|+3-\hat{\jmath})} \overset{(13)}{=} s'_{2i'} \overset{(4b)}{=} i'+a' \overset{(13)}{=} |v|+3-\hat{\jmath}+a' = |v|+2-\hat{\imath}+a' \ . \tag{19}$$

Using that i' and $\hat{\imath}$ have the same parity ($i' = |v|+2-\hat{\imath}$) and the definition (1c), we have $a'=0$ if $\hat{\imath}$ is even and $a'=-1$ if $\hat{\imath}$ is odd (the case $a'=1$ does not occur, since the subpath of x' strictly between steps i' and $j'=i'+1$ has length 0). Similarly, the value of a in (12b) depends on the parity of i, and since i and $\hat{\imath}$ have opposite parity (recall (11)), we obtain from (1c) that $a=-1$ if $\hat{\imath}$ is even and $a=0$ if $\hat{\imath}$ is odd (again the case $a=1$ does not occur, since the subpath of x strictly between steps i and $j=i+1$ has length 0). Consequently, if $\hat{\imath}$ is even we have

$$s_{2\hat{\imath}-1} + \tau_{2n}(s'_{2(|v|+2-\hat{\imath})}) \overset{(12b),(19),(5)}{=} (i-1)+(|v|+2-\hat{\imath}+1) = i-\hat{\imath}+|v|+2 \overset{(11),(10)}{=} 2n+1 \ . \tag{20}$$

If $\hat\imath$ is odd then the middle term of the previous calculation changes to $i+(|v|+2-\hat\imath-1+1)$, yielding the same result.

Case 3 (ii): $\hat\jmath > \hat\imath+1$. (This is the subcase shown in Figure 4.) By the condition $\hat\jmath > \hat\imath+1$ there is a (unique) base pair $(\hat\imath+1,k)$ with $\hat\imath+2 \le k \le \hat\jmath-1$ in $\widehat{x}$. It follows that $|v|+3-(k,\hat\imath+1) = (|v|+3-k, j'-1)$ (recall (13)) is a base pair in x', so from Lemma 4 we obtain that

$$s'_{2(|v|+2-\hat\imath)} \overset{(13)}{=} s'_{2(j'-1)} \overset{(4d)}{=} = j'-1+c' \overset{(13)}{=} |v|+2-\hat\imath+c' \ . \tag{21}$$

In this case we have $c'=0$ as the subpath of x' strictly between steps $j'-1$ and j' has length 0. Moreover, for the value of a in (12b) we have $a=-1$ if $\hat\imath$ is even and $a=1$ if $\hat\imath$ is odd (the case $a=0$ does not occur, since the subpath of x strictly between steps i and j is nonempty). Consequently, if $\hat\imath$ is even, then adding up $s_{2\hat\imath-1}$ and $\tau_{2n}(s'_{2(|v|+2-\hat\imath)})$ and using (12b), (21) and (5) yields the same two terms as in the middle of (20), proving the claim in this case. If $\hat\imath$ is odd, then the middle term of this calculation changes to $(i+1)+(|v|+2-\hat\imath-1)$, yielding the same result.

Case 4: $\ell = 2\hat\jmath-1$. In Figure 4, the ℓ-th entry of s is the fourth red entry of s. The relevant entry of the sequence s' in this case is $s'_{2|v|+3-\ell} = s'_{2(|v|+2-\hat\jmath)}$. Here we distinguish the subcases (i) $\widehat{x}_{\hat\jmath+1}=0$ and (ii) $\widehat{x}_{\hat\jmath+1}=1$.

Case 4 (i): $\widehat{x}_{\hat\jmath+1}=0$. (This is the subcase shown in Figure 4.) In this case there is (unique) base pair $(k,\hat\jmath+1)$ in $\widehat{x}$ with $1 \le k \le \hat\imath-1$. It follows that $|v|+3-(\hat\jmath+1,k) = (i'-1,|v|+3-k)$ (recall (13)) is a base pair in x', so from Lemma 4 we obtain in the case $\hat\jmath < |v|+1$ that

$$s'_{2(|v|+2-\hat\jmath)} \overset{(13)}{=} s'_{2(i'-1)} \overset{(4b)}{=} i'-1+a' \overset{(13)}{=} |v|+2-\hat\jmath+a' \ . \tag{22}$$

Using that $\hat\jmath$ and $i'-1$ have the same parity ($i'-1 = |v|+2-\hat\jmath$) and the definition (1c), we have $a'=1$ if $\hat\jmath$ is even and $a'=-1$ if $\hat\jmath$ is odd (the case $a'=0$ does not occur, since the subpath of x' strictly between steps $i'-1$ and $|v|+3-k$ is nonempty). If on the other hand $\hat\jmath = |v|+1$, then

$$s'_{2(|v|+2-\hat\jmath)} = s'_2 \overset{(1a)}{=} 1 = |v|+2-\hat\jmath \ . \tag{23}$$

Moreover, for the value of c in (12d) we have $c=0$, as the subpath of x strictly between steps j and $j+1$ has length 0. Consequently, if $\hat\jmath$ is even we have

$$s_{2\hat\jmath-1} + \tau_{2n}(s'_{2(|v|+2-\hat\jmath)}) \overset{(12d),(22),(5)}{=} j+(|v|+2-\hat\jmath+1-1) = j-\hat\jmath+|v|+2 \overset{(11),(10)}{=} 2n+1 \ .$$

If $\hat\jmath$ is odd and $\hat\jmath = |v|+1$, then the middle term of the previous calculation changes to $j+1 = j+(|v|+2-\hat\jmath)$ (recall (23)), yielding the same result. If $\hat\jmath$ is odd and $\hat\jmath < |v|+1$, then the middle term of this calculation changes to $j+(|v|+2-\hat\jmath-1+1)$, again yielding the same result.

Case 4 (ii): $\widehat{x}_{\hat\jmath+1}=1$. In this case there is a (unique) base pair $(\hat\jmath+1,k)$ in $\widehat{x}$ with $\hat\jmath+2 \le k \le |v|+1$. As $|v|+3-(k,\hat\jmath+1) = (|v|+3-k, i'-1)$ (recall (13)) is a base pair in x', we obtain from Lemma 4 that

$$s'_{2(|v|+2-\hat\jmath)} \overset{(13)}{=} s'_{2(i'-1)} \overset{(4d)}{=} i'-1+c' \overset{(13)}{=} |v|+2-\hat\jmath+c' \ . \tag{24}$$

Using that $\hat\jmath$ and $|v|+3-k$ have opposite parity ($|v|+3-k$ has the opposite parity as $i'-1$, and $i'-1 = |v|+2-\hat\jmath$) and the definition (1c), we have $c'=1$ if $\hat\jmath$ is even and $c'=0$ if $\hat\jmath$ is odd (note that the subpath of x' from step i' to j' is nonempty). Moreover, for the value of c in (12d) we have $c=1$ if $\hat\jmath$ is odd and $c=0$ if $\hat\jmath$ is even ($\hat\jmath$ and i have the same parity, also the subpath of x from step $\hat\jmath+1+(|u|+1)$ to step $k+(|u|+1)$ is nonempty). Consequently, if $\hat\jmath$ is even we have

$$s_{2\hat\jmath-1} + \tau_{2n}(s'_{2(|v|+2-\hat\jmath)}) \overset{(12d),(24)(5)}{=} j+(|v|+2-\hat\jmath+1-1) = j-\hat\jmath+|v|+2 \overset{(11),(10)}{=} 2n+1 \ .$$

If $\hat\jmath$ is odd then the middle term of the previous calculation changes to $(j+1)+(|v|+2-\hat\jmath-1)$, yielding the same result.

This completes the proof of the last case and thus the proof of the lemma. ☐

Extending the definitions from Section 2.1, we let $D_n^e(k) \subseteq B_n(k)$ denote the set of all bitstrings (viewed as lattice paths) that consist of n steps in total, k of them 1-steps (and $n-k$ many 0-steps), with exactly e many flaws. Clearly, we have $D_{2n}^e(n) = D_{2n}^e$. Furthermore, we partition $D_n^0(k)$ into two sets $D_n^{=0}(k)$ and $D_n^{>0}(k)$, respectively, depending on whether the lattice path returns to the line $y=0$ or not (after leaving this line with the first 1-step). Note that we have $D_{2n}^0(n) = D_{2n}^{=0}(n)$ and $D_{2n}^{>0}(n) = \emptyset$. We also extend the concept of canonical decomposition in a straightforward way to all bitstrings $D_n^{=0}(k)$ and $D_n^1(k)$ (decompose according to the first point to the right of the origin where the lattice path touches the line $y=0$ or $y=-1$, respectively). Accordingly, the definition (1) of the recursion σ extends straightforwardly to any bitstrings $x \in D_{2n}^{=0}(k)$, yielding a path $P_\sigma(x)$ in the graph $Q_{2n}(k,k+1)$ (the fact that the resulting sequence of vertices actually forms a path in $Q_{2n}(k,k+1)$ follows from the next lemma).

For the statement and proof of the next lemma we assume that the reader is familiar with the inductive construction of the paths $\mathcal{P}_{2n}(k,k+1)$, $k \in \{n,n+1,\ldots,2n-1\}$, in the graph $Q_{2n}(k,k+1)$ described in [Müt16, Section 2.2]. Each path $P \in \mathcal{P}_{2n}(k,k+1)$, $P =: (v_1,v_2,\ldots,v_\ell)$, is oriented, i.e., we distinguish its first vertex $F(P) := v_1$ from its last vertex $L(P) := v_\ell$. The second vertex is given by $S(P) := v_2$. The paths $\mathcal{P}_{2n}(k,k+1)$ are (vertex-)disjoint by construction.

Lemma 6. *Let $n \geq 1$ and $k \in \{n,n+1,\ldots,2n-1\}$ be fixed, and let $\mathcal{P}_{2n}(k,k+1)$ be the set of paths in $Q_{2n}(k,k+1)$ defined in* [Müt16, Section 2.2] *for the parameter sequence $\alpha_{2i} = (1,1,\ldots,1) \in \{0,1\}^{i-1}$, $i=1,\ldots,n-1$. For any path $P \in \mathcal{P}_{2n}(k,k+1)$ and its first vertex $x := F(P) \in D_{2n}^{=0}(k)$ we have $P_\sigma(x) = P$.*

In the following we will repeatedly use that by [Müt16, Lemma 11] the sets of first, second and last vertices of the paths in $\mathcal{P}_{2n}(k,k+1)$ satisfy the relations

$$F(\mathcal{P}_{2n}(k,k+1)) = D_{2n}^{=0}(k) \ , \tag{25a}$$

$$S(\mathcal{P}_{2n}(k,k+1)) = D_{2n}^{>0}(k+1) \ , \tag{25b}$$

$$L(\mathcal{P}_{2n}(k,k+1)) = D_{2n}^{1}(k) \ . \tag{25c}$$

More specifically, by [Müt16, Lemma 14], considering for any $P \in \mathcal{P}_{2n}(k,k+1)$ the canonical decomposition of $F(P) = (1,u,0,v) \in D_{2n}^{=0}(k)$, then the second vertex and last vertex of P are given by

$$S(P) = (1,u,1,v) \in D_{2n}^{>0}(k+1) \ , \tag{26a}$$

$$L(P) = (\pi(u),0,1,v) \in D_{2n}^{1}(k) \ , \tag{26b}$$

respectively.

To avoid duplication, in proving Lemma 6 we will repeatedly refer to several relations in [Müt16, Section 2.2] that describe the construction of the paths $\mathcal{P}_{2n}(k,k+1)$ in detail.

Proof. We prove the lemma by induction on n. To settle the base case $n=1$ observe that the set of paths $\mathcal{P}_2(1,2)$ defined in [Müt16, Eq. (2)] contains only a single path $((1,0),(1,1),(0,1)) =: P$, and for $x := F(P) = (1,0)$ we have $\sigma(x) = (2,1)$, so indeed $P_\sigma(x) = P$.

For the induction step $n \to n+1$ we assume that the lemma holds for $n \geq 1$ and all $k = n, n+1,\ldots,2n-1$, and prove it for $n+1$ and all $k = n+1,n+2,\ldots,2n+1$. We distinguish the cases $k = n+1$ and $k \in \{n+2,\ldots,2n+1\}$.

For the case $k \in \{n+2,\ldots,2n+1\}$ let $\mathcal{P}_{2n+2}(k,k+1)$ be the set of paths defined in [Müt16, Eq. (8)] and let P be a path from this set. Then P is of the form $P' \circ (\alpha,\beta)$, $\alpha,\beta \in \{0,1\}$, with $P' \in \mathcal{P}_{2n}(k-\alpha-\beta,k+1-\alpha-\beta)$ (i.e., P' is contained in one of the four sets on the right hand side of [Müt16, Eq. (8)]). Defining $x' := F(P')$ we have $P_\sigma(x') = P'$ by induction, and since $x := F(P) \in D_{2n+2}^{=0}(k)$ and $k \geq n+2$, the two sequences $\sigma(x)$ and $\sigma(x')$ are the same (in particular the two last steps of x are never flipped by $\sigma(x)$). Therefore, we have $P_\sigma(x) = P_\sigma(x') \circ (\alpha, \beta) = P' \circ (\alpha, \beta) = P$, as claimed.

For the case $k = n+1$ let $\mathcal{P}_{2n+2}(n+1, n+2)$ be the set of paths defined in [Müt16, Eq. (11)] and let P be a path from this set. We distinguish two cases depending on which of the three sets on the right hand side of [Müt16, Eq. (11)] the path P is contained in.

Case 1: If P is contained in the first or in the second set on the right hand side of [Müt16, Eq. (11)], i.e., in the set $\mathcal{P}_{2n}(n+1, n+2) \circ (0,0)$ or the set $\mathcal{P}_{2n}(n, n+1) \circ (1,0)$, then the claim follows by induction with the same argument as before when $k \geq n+2$.

Case 2: If P is contained in the third set on the right hand side of [Müt16, Eq. (11)], i.e., in the set $\mathcal{P}^*_{2n+2}$, then by the definitions in [Müt16, Eqs. (5),(9)] and the fact that for $\alpha_{2n} = (1, 1, \ldots, 1) \in \{0,1\}^{n-1}$ the mapping $f_{\alpha_{2n}}$ defined in [Müt16, Eq. (3)] satisfies $f_{\alpha_{2n}}(x) = \overline{\mathrm{rev}}(\pi(x))$ for all $x \in \{0,1\}^{2n}$ (recall the definitions of $\overline{\mathrm{rev}}$ and π given in Section 2.1). The sequence of edges of the path P when traversing it from its first to its last vertex has the form $(e_1, E_2, e_3, E_4, e_5)$, where e_1, e_3 and e_5 are single edges and E_2 and E_4 are sequences of edges that satisfy the following conditions: there are two unique paths $P', P'' \in \mathcal{P}_{2n}(n, n+1)$ such that

$$\begin{aligned}
e_1 &= \big((S(P'), 0, 0), (S(P'), 0, 1)\big) \ , \\
e_3 &= \big((L(P'), 0, 1), (L(P'), 1, 1)\big) = \big((L(P'), 0, 1), (\overline{\mathrm{rev}}(\pi(L(P''))), 1, 1)\big) \ , \\
e_5 &= \big((\overline{\mathrm{rev}}(\pi(F(P''))), 1, 1), (\overline{\mathrm{rev}}(\pi(F(P''))), 0, 1)\big) \ ,
\end{aligned}$$

E_2 is given by traversing the edges of $P' \circ (0,1)$ starting at the vertex $(S(P'), 0, 1)$ and ending at the vertex $(L(P'), 0, 1)$, and E_4 is given by the traversing all the edges of $\overline{\mathrm{rev}}(\pi(P'')) \circ (1,1)$ in reverse order starting at the vertex $(\overline{\mathrm{rev}}(\pi(L(P''))), 1, 1)$ and ending at the vertex $(\overline{\mathrm{rev}}(\pi(F(P''))), 1, 1)$.

We proceed to show that for $x := F(P)$ we have $P_\sigma(x) = P = (e_1, E_2, e_3, E_4, e_5)$ by determining the bitflip sequences that define P' and P'' and comparing those sequences with $\sigma(x)$.

By what we said before, the first vertex of P is $x = (S(P'), 0, 0)$, so using (25b) we can write $S(P') = (1, u, 1, v) \in D_{2n}^{>0}(n+1)$ for some $u \in D_{|u|}^0$ and $v \in D_{|v|}^0$ with $|u| + |v| + 2 = 2n$ and therefore $x = (1, u, \widehat{x}, 0) \in D_{2n+2}^0$ with $\widehat{x} := (1, v, 0) \in D_{|v|+2}^0$. From (26a) and (26b) we obtain that $F(P') = (1, u, 0, v) =: x' \in D_{2n}^0$ and $L(P') = (\pi(u), 0, 1, v) \in D_{2n}^1$. Moreover, since $\overline{\mathrm{rev}}(\pi(L(P''))) = L(P')$ (recall the definition of e_3) we have $L(P'') = (\overline{\mathrm{rev}}(\pi(v)), 0, 1, \overline{\mathrm{rev}}(u)) \in D_{2n}^1$ and $F(P'') = (1, \overline{\mathrm{rev}}(v), 0, \overline{\mathrm{rev}}(u)) =: x'' \in D_{2n}^0$ (note here that $\overline{\mathrm{rev}}$ and π commute and that each of these operations is idempotent). By induction we also know that $P_\sigma(x') = P'$ and $P_\sigma(x'') = P''$.

Using the definition (1a) we obtain

$$\sigma(x) = \big(|x|, 1, 1 + \sigma'(u), 1 + |u| + \sigma'(\widehat{x})\big) \ , \tag{27}$$

$$\sigma(x') = \big(|u| + 2, 1, 1 + \sigma'(u)\big) \ , \tag{28}$$

where the subsequence u starts at an even index in both x and x'.

Recall that the first edge of P, the edge e_1, is given by flipping the last bit of x (this is the $(2n+2)$-th bit), and by (27) the first entry of $\sigma(x)$ is $|x| = 2n+2$, so this is also the first edge of $P_{\sigma(x)}$.

By (27) and (28), the sequence obtained by removing the first element of $\sigma(x')$ appears as a subsequence at exactly the same positions in $\sigma(x)$. Using that $P_\sigma(x') = P'$, if follows that the sequence of edges E_2 traversed by P after the edge e_1 is exactly the same sequence of edges as traversed by $P_\sigma(x)$ after the edge e_1. We conclude that $P_\sigma(x)$ and P agree on the edges (e_1, E_2) until both paths reach the vertex $(L(P'), 0, 1)$.

Recall that the next edge of P, the edge e_3, is given by flipping the second-to-last bit of this vertex (this is the $(2n+1)$-th bit), and by the relations $\widehat{x} = (1, v, 0)$, (27) and (1b), the next relevant entry of $\sigma(x)$ is indeed $1 + |u| + |v| + 2 = 2n + 1$, so this is also the next edge of $P_{\sigma(x)}$. We conclude that $P_\sigma(x)$ and P agree on the edges (e_1, E_2, e_3) until both paths reach the vertex $(L(P'), 1, 1) = (\overline{\mathrm{rev}}(\pi(L(P''))), 1, 1)$ (recall the definition of e_3).

By the definition of E_4 and the relation $P_\sigma(x'') = P''$, the bit positions flipped when following this sequence of edges is given by $2n+1-\mathrm{rev}(\tau_{2n}(\sigma(x'')))$ (the operation $\overline{\mathrm{rev}}$ on the vertices of path P'' results in the transformation $g(s) = 2n-1-s$ of the bitflip sequence, the permutation π results in the transformation τ_{2n}, and traversing the path backwards results in the transformation rev). Applying Lemma 5 (using that $x'' = (1, \overline{\mathrm{rev}}(v), 0, \overline{\mathrm{rev}}(u)))$, this sequence is equal to the sequence obtained by removing the first and last entry from $1 + |u| + \sigma'(\widehat{x})$. Using (27) again we conclude that $P_\sigma(x)$ and P agree on the edges (e_1, E_2, e_3, E_4) until both paths reach the vertex $(\overline{\mathrm{rev}}(\pi(F(P''))), 1, 1)$.

Recall that the next edge of P (which is the last edge of P), the edge e_5, is given by flipping the second-to-last bit of this vertex (the $(2n+1)$-th bit), and by the same reasoning as for e_3, the last entry of $\sigma(x)$ is indeed $2n+1$. We conclude that $P_\sigma(x)$ and P agree on all edges $(e_1, E_2, e_3, E_4, e_5)$.

This completes the analysis in case 2 of the induction step, and hence the proof of the lemma. □

Given Lemma 6, we may from now on use the notation $\mathcal{P}_{2n}(n, n+1)$ interchangeably for the set of paths defined inductively in [Müt16, Section 2.2] and the set of paths defined by the bitflip sequence σ introduced in Section 2.2. By combining Lemma 6, property (i) from [Müt16, Section 2.2] and the relations (25a) and (25c) (which are part of [Müt16, Lemma 11]), we obtain that the paths $\mathcal{P}_{2n}(n, n+1)$ computed by σ indeed satisfy properties (i) and (ii) claimed in Section 2.2.

3.2. **Correctness of the paths computed by $\widetilde{\sigma}$.** It follows directly from the definitions (2) that for pairs of vertices (x, x'), $x, x' \in D_{2n}^0$, of type 1 or type 2, the paths $(P_{\widetilde{\sigma}}(x), P_{\widetilde{\sigma}}(x'))$ form a flipped pair of paths with respect to the flippable pair of paths $(P_\sigma(x), P_\sigma(x'))$ (one can easily verify that these pairs of paths visit the same set of vertices, and that the end vertices are connected the other way).

3.3. **Correctness of the function Init().** The correctness of this function follows since by Lemma 6 and [MN15, Lemma 4], the paths $\mathcal{P}_{2n}(n, n+1)$ defined by our recursion σ, by the recursive Paths() algorithm from [MN15], and inductively in [Müt16] are all the same (so moving along a path can be done using either of the three methods).

3.4. **Correctness of the algorithm HamCycle().** Having proved that the paths $\mathcal{P}_{2n}(n, n+1)$ computed by σ indeed satisfy properties (i) and (ii) claimed in Section 2.2, then under the assumption that the auxiliary functions IsFlipTree$_1$() and IsFlipTree$_2$() always return `false` (i.e., FlipTree() always returns `false`), the arguments given in Section 2.4 show that the algorithm HamCycle() correctly computes the 2-factor $\mathcal{C}_{2n+1}$ defined in (3) in the middle levels graph $Q_{2n+1}(n, n+1)$. We proceed to show that the algorithm HamCycle() computes a different 2-factor (but still a 2-factor) for each possible choice of boolean functions IsFlipTree$_1$() and IsFlipTree$_2$() that are called from within FlipTree() (later we prove that the functions IsFlipTree$_1$() and IsFlipTree$_2$() specified in Section 2.5 yield a 2-factor that consists only of a single cycle, i.e., a Hamilton cycle). As already mentioned in Section 2.4, the different 2-factors are obtained by replacing some flippable pairs of paths from $\mathcal{P}_{2n}(n, n+1)$ in the first set on the right hand side of (3) by the corresponding flipped pairs of paths from $\widetilde{\mathcal{P}}_{2n}(n, n+1)$. There are two potential problems that the function FlipTree() could cause in this approach, and the next lemma shows that none of these problems occurs. First, if a call FlipTree(T) in line H4 returns `true` then the corresponding vertex y^- defined in line H3 might neither be part of a type 1 or type 2 pair, so

$\widetilde{\sigma}(y^-)$ called in line H5 is undefined. Second, given a flippable pair of paths $P, P' \in \mathcal{P}_{2n}(n, n+1)$, the results of the two corresponding calls to FlipTree() might be different (inconsistent), so our algorithm would not compute a valid 2-factor.

Lemma 7. *For $n \geq 1$ let* IsFlipTree$_1$() *and* IsFlipTree$_2$() *be arbitrary boolean functions on the sets of ordered rooted trees $\mathcal{T}^*_{n,1}$ and $\mathcal{T}^*_{n,2}$ defined in Section* 2.5*, respectively, and let* FlipTree() *be as defined in Algorithm* 2*. Let T and y^- be the values of the corresponding variables after the assignment in line* H3*. If* FlipTree$(T) =$ `true`*, then y^- is part of a type 1 or type 2 pair of vertices $y^-, x' \in D^0_{2n}$ (either the pair (y^-, x') or (x', y^-)). Moreover, denoting by T' be the value of the state tree when $y^- = x'$, then* FlipTree$(T') =$ `true`*.*

Proof. To prove the lemma we first establish the following invariant of the algorithm HamCycle(): Immediately after the assignment in line H3 the variables T and y^- satisfy the relation $y^- = h(T)$ with h as defined in Section 2.5.

We prove that this invariant holds by induction over the number of iterations of the while-loop in the algorithm HamCycle(). For $k = 1, 2, \ldots$ we let T_k and y^-_k denote the values of the variables T and y^- during the k-th iteration of the while-loop immediately after the assignment in line H3. To settle the base case $k = 1$, note that $y^-_1 = h(T_1)$ by the definition of the function Init() given in Section 2.6. For the induction step let $k \geq 1$ and consider the k-th iteration of the while-loop, assuming that

$$h^{-1}(y^-_k) = T_k \ . \tag{29}$$

We first consider the case that FlipTree(T_k) returns `false`. In this case we clearly have

$$T_{k+1} = \mathrm{rot}(T_k) \tag{30}$$

(note that FlipTree() does not modify the variable T, but only the rotation operation is applied in line H6 of the algorithm HamCycle()). Applying [Müt16, Lemma 20] we obtain

$$h^{-1}(y^-_{k+1}) = \mathrm{rot}(h^{-1}(y^-_k)) \overset{(29),(30)}{=} T_{k+1}$$

(note here that the 2-factor $\mathcal{C}^1_{2n+1}$ referred to in [Müt16, Lemma 20] and defined in [Müt16, Eqs. (5),(7)] is exactly the 2-factor $\mathcal{C}_{2n+1}$ defined in (3): to see this recall Lemma 6 and that the mapping $f_{\alpha_{2n}}$ defined in [Müt16, Eq. (3)] satisfies $f_{\alpha_{2n}} = \overline{\mathrm{rev}}$ for $\alpha_{2n} = (0, 0, \ldots, 0) \in \{0, 1\}^{n-1}$). This proves the invariant in this case.

We now consider the case that FlipTree(T_k) returns `true`, assuming that the condition in line T2 is satisfied. In this case we clearly have

$$T_{k+1} = \mathrm{rot}(\tau_1(T_k)) \ . \tag{31}$$

As $T_k \in \mathcal{T}^*_{n,1}$, we obtain from the definition of the set $\mathcal{T}^*_{n,1}$ and from [Müt16, Lemma 21] that $y^-_k = h(T_k)$ is the first component of a type 1 pair (y^-_k, x') of vertices $y^-_k, x' \in D^0_{2n}$ (recall the definitions from Section 2.3). Using (29), the definition of τ_1 and [Müt16, Lemma 21], we obtain

$$h^{-1}(x') = \tau_1(T_k) \ . \tag{32}$$

Furthermore, applying the definition of flippable and flipped pairs of paths and [Müt16, Lemma 20] yields

$$h^{-1}(y^-_{k+1}) = \mathrm{rot}(h^{-1}(x')) \overset{(32),(31)}{=} T_{k+1} \ ,$$

proving the invariant also in this case. The case that FlipTree(T_k) returns `true` as a consequence of the condition in line T3 being satisfied can be treated analogously. Also the remaining two cases corresponding to lines T4 and T5 follow along similar lines, using [Müt16, Lemma 22] instead of [Müt16, Lemma 21].

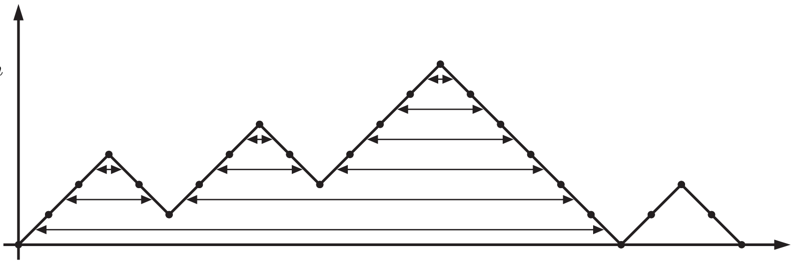


FIGURE 5. Auxililiary pointers to compute the bitflip sequence $\sigma(x)$ for any $x \in D_{2n}^n$ in time $\mathcal{O}(n)$. The lattice path x is the same as in Figure 1, and the resulting sequence $\sigma(x)$ is shown in that figure.

Using this invariant we now argue as follows: Suppose FLIPTREE(T) returns `true` because the condition in line T2 is satisfied, i.e., $T \in \mathcal{T}_{n,1}^*$ and ISFLIPTREE$_1$(T) = `true`. Applying the same reasoning as before, it follows that $y^- = h(T)$ is the first component of a type 1 pair (y^-, x') of vertices $y^-, x' \in D_{2n}^0$ and $h^{-1}(x') = \tau_1(T)$ (recall (32)). Using the invariant $x' = h(T')$, we obtain that $T' = \tau_1(T) \in \tau_1(\mathcal{T}_{n,1}^*)$. Consequently, FLIPTREE($\tau_1^{-1}(T')$) = ISFLIPTREE$_1$(T) = `true`, i.e., in the call FLIPTREE(T') the condition in line T3 will be satisfied, and the function will return `true` as well. The other three cases corresponding to the conditions in line T3–T5 can again be proven along very similar lines. This completes the proof of the lemma. □

All the remaining arguments required for completing the proof of correctness of the algorithm HAMCYCLE() can be taken almost verbatim from [MN15], starting with Lemma 7 until the end of Section 3 in that paper. We briefly mention the few changes that are necessary to transfer the arguments to our algorithm. The most important remark is that due the changes to the functions ISFLIPTREE$_1$() and ISFLIPTREE$_2$() compared to the definitions in [MN15], the spanning tree $\mathcal{H}_n$ of the graph $\mathcal{G}_n$ defined in [MN15, Section 3.4.3] is in general different. Consequently, the resulting Hamilton cycle computed by our algorithm HAMCYCLE() is different. Nonetheless, the proof of [MN15, Lemma 9] still goes through with minimal adjustments: The lexicographic comparisons that are used as a tie-breaking criterion in the definitions of ISFLIPTREE$_1$() and ISFLIPTREE$_2$() are in our new algorithm replaced by calls to the function ROOT() defined in Section 2.5 to compute a canonically rooted version of the given tree. In essence, these changes do not affect the correctness proof at all, but are only required to speed up the running time of ISFLIPTREE$_1$() and ISFLIPTREE$_2$() from quadratic to linear in n.

4. Running time and space requirements of the algorithm

4.1. **Running time.** For any bitstring/lattice path $x \in D_{2n}^0$, the bitflip sequence $\sigma(x)$ can be computed in time $\mathcal{O}(n)$. To achieve this, we precompute in time $\mathcal{O}(n)$ an array of bidirectional pointers below the 'hills' of x between corresponding pairs of steps on the same height level (see Figure 5; this idea is borrowed from [MSW16]). Using these pointers, each canonical decomposition operation encountered in the recursion (1) can be done in constant time, so that the overall running time of the recursion is $\mathcal{O}(n)$. Clearly, the sequence $\widetilde{\sigma}(x)$ can also be computed in time $\mathcal{O}(n)$ by modifying the sequence $\sigma(x)$ as described in (2) in constantly many positions.

In our algorithm HAMCYCLE(), the state tree T with n edges (an ordered rooted tree) is stored using adjacency lists, rather than a bitstring of length $2n$ as in [MN15]. The conversion between the two representations can clearly be done in time $\mathcal{O}(n)$ (recall the correspondence between bitstrings D_{2n}^0 and trees $\mathcal{T}_n^*$ from Figure 3). The advantage of the adjacency list representation is that it allows to do one rotation operation rot() in time $\mathcal{O}(1)$, so a full tree rotation until we are back at the starting vertex takes time $\mathcal{O}(n)$. Using this representation, we can compute the function ROOT() defined in Section 2.5 in time $\mathcal{O}(n)$: The center vertex/vertices can be computed in linear time by removing leaves in rounds until only a single vertex or a single edge is left (this vertex/vertices form the center). Booth's algorithm to compute the lexicographically smallest string rotation also runs in linear time. The same time bound $\mathcal{O}(n)$ therefore also holds for the functions ISFLIPTREE$_1$() and ISFLIPTREE$_2$() (both functions essentially rotate a tree for one or two full rotations).

It was shown in [MW12, Theorem 10] that the distance between any two neighboring vertices of the form $(x, 0)$, $(x', 0)$ with $x, x' \in D_{2n}^0$ (x and x' are first vertices of two paths $P, P' \in \mathcal{P}_{2n}(n, n+1)$) on a cycle in (3) is exactly $4n+2$. Moreover, by the definition (2), replacing in the first set on the right hand side of (3) a path $P \in \mathcal{P}_{2n}(n, n+1)$ that is contained in a flippable pair by the corresponding flipped path $P' \in \widetilde{P}_{2n}(n, n+1)$ changes this distance by $c \in \{-4, 0, +4\}$. It follows that in each iteration of the while-loop of our algorithm HAMCYCLE(), exactly $4n + 2 + c$ vertices are visited. Combining this with the time bounds $\mathcal{O}(n)$ derived for the functions $\sigma()$, $\widetilde{\sigma}()$ and FLIPTREE() that are called once or twice during each iteration of the while-loop, we conclude that the while-loop takes time $\mathcal{O}(n + \ell)$ to visit ℓ vertices of the Hamilton cycle.

The function INIT() takes time $\mathcal{O}(n^2)$, as each call to the PATHS() recursion takes time $\mathcal{O}(n)$ (see [MN15, Section 4.1]), and only $\mathcal{O}(n)$ such calls are necessary (every path in $\mathcal{P}_{2n}(n, n+1)$ has only length $\mathcal{O}(n)$). Computing h^{-1} can also be achieved in time $\mathcal{O}(n^2)$ by implementing its recursive definition directly (see [MN15, Section 2.5]).

Combining the time bounds $\mathcal{O}(n^2)$ for the initialization phase and the time $\mathcal{O}(n + \ell)$ spent in the while-loop we obtain the claimed overall time bound $\mathcal{O}(n^2 + \ell) = \mathcal{O}(\ell(1 + \frac{n^2}{\ell}))$ for the algorithm HAMCYCLE().

4.2. **Space requirements.** Throughout our algorithm, we only store constantly many bitstrings of length $2n$ and ordered rooted trees with n edges, proving that the entire space needed is $\mathcal{O}(n)$.

Acknowledgements

The authors thank Darius Wuttke for his assistance in writing the C++ code accompanying this paper.

References

[BER76] J. Bitner, G. Ehrlich, and E. Reingold. Efficient generation of the binary reflected Gray code and its applications. *Comm. ACM*, 19(9):517–521, 1976.

[Boo80] K. Booth. Lexicographically least circular substrings. *Inform. Process. Lett.*, 10(4-5):240–242, 1980.

[BW84] M. Buck and D. Wiedemann. Gray codes with restricted density. *Discrete Math.*, 48(2-3):163–171, 1984.

[CF49] K. Chung and W. Feller. On fluctuations in coin-tossing. *Proceedings of the National Academy of Sciences of the United States of America*, 35(10):605–608, 1949.

[Che00] Y. Chen. Kneser graphs are Hamiltonian for $n \geq 3k$. *J. Combin. Theory Ser. B*, 80(1):69–79, 2000.

[Che03] Y. Chen. Triangle-free Hamiltonian Kneser graphs. *J. Combin. Theory Ser. B*, 89(1):1–16, 2003.

[Der75] N. Dershowitz. A simplified loop-free algorithm for generating permutations. *Nordisk Tidskr. Informationsbehandling (BIT)*, 15(2):158–164, 1975.

[DG12] P. Diaconis and R. Graham. *Magical mathematics*. Princeton University Press, Princeton, NJ, 2012. The mathematical ideas that animate great magic tricks, With a foreword by Martin Gardner.

[DKS94] D. Duffus, H. Kierstead, and H. Snevily. An explicit 1-factorization in the middle of the Boolean lattice. *J. Combin. Theory Ser. A*, 65(2):334–342, 1994.

[DSW88] D. Duffus, B. Sands, and R. Woodrow. Lexicographic matchings cannot form Hamiltonian cycles. *Order*, 5(2):149–161, 1988.

[Ehr73] G. Ehrlich. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. Assoc. Comput. Mach.*, 20:500–513, 1973.

[EHR84] P. Eades, M. Hickey, and R. Read. Some Hamilton paths and a minimal change algorithm. *J. ACM*, 31(1):19–29, January 1984.

[EM84] P. Eades and B. McKay. An algorithm for generating subsets of fixed size with a strong minimal change property. *Information Processing Letters*, 19(3):131–133, 1984.

[FT95] S. Felsner and W. Trotter. Colorings of diagrams of interval orders and α-sequences of sets. *Discrete Math.*, 144(1-3):23–31, 1995. Combinatorics of ordered sets (Oberwolfach, 1991).

[GM16] P. Gregor and T. Mütze. Trimming and gluing Gray codes. *arXiv:1607.08806*, Jul 2016.

[Gra53] F. Gray. Pulse code communication, 1953. March 17, (filed Nov. 1947). U.S. Patent 2,632,058.

[GŠ10] P. Gregor and R. Škrekovski. On generalized middle-level problem. *Inform. Sci.*, 180(12):2448–2457, 2010.

[Hav83] I. Havel. Semipaths in directed cubes. In *Graphs and other combinatorial topics (Prague, 1982)*, volume 59 of *Teubner-Texte Math.*, pages 101–108. Teubner, Leipzig, 1983.

[HKRR05] P. Horák, T. Kaiser, M. Rosenfeld, and Z. Ryjácek. The prism over the middle-levels graph is Hamiltonian. *Order*, 22(1):73–81, 2005.

[Hur94] G. Hurlbert. The antipodal layers problem. *Discrete Math.*, 128(1-3):237–245, 1994.

[Joh63] S. Johnson. Generation of permutations by adjacent transposition. *Math. Comp.*, 17:282–285, 1963.

[Joh04] R. Johnson. Long cycles in the middle two layers of the discrete cube. *J. Combin. Theory Ser. A*, 105(2):255–271, 2004.

[Knu11] D. Knuth. *The Art of Computer Programming, Volume 4A*. Addison-Wesley, 2011.

[KT88] H. Kierstead and W. Trotter. Explicit matchings in the middle levels of the Boolean lattice. *Order*, 5(2):163–171, 1988.

[Luc87] J. Lucas. The rotation graph of binary trees is Hamiltonian. *Journal of Algorithms*, 8(4):503–535, 1987.

[LvBR93] J. Lucas, D. R. van Baronaigien, and F. Ruskey. On rotations and the generation of binary trees. *Journal of Algorithms*, 15(3):343–366, 1993.

[Mac09] P. MacMahon. Memoir on the theory of the partitions of numbers. Part iv. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 209(441-458):153–175, 1909.

[MN15] T. Mütze and J. Nummenpalo. Efficient computation of middle levels Gray codes. In N. Bansal and I. Finocchi, editors, *Algorithms - ESA 2015*, volume 9294 of *Lecture Notes in Computer Science*, pages 915–927. Springer Berlin Heidelberg, 2015. All references are to the unabridged version at *arXiv:1506.07898*.

[MS15] T. Mütze and P. Su. Bipartite Kneser graphs are Hamiltonian. *arXiv:1503.09175*. To appear in *Combinatorica*, Mar 2015.

[MSW16] T. Mütze, C. Standke, and V. Wiechert. A minimum-change version of the Chung-Feller theorem for Dyck paths. *arXiv:1603.02525*, Feb 2016.

[Müt16] T. Mütze. Proof of the middle levels conjecture. *Proc. London Math. Soc.*, 112(4):677–713, 2016. All references are to the unabridged version at *arXiv:1404.4442*.

[MW12] T. Mütze and F. Weber. Construction of 2-factors in the middle layer of the discrete cube. *J. Combin. Theory Ser. A*, 119(8):1832–1855, 2012.

[NW75] A. Nijenhuis and H. Wilf. *Combinatorial algorithms*. Academic Press, New York-London, 1975. Computer Science and Applied Mathematics.

[Rus88] F. Ruskey. Adjacent interchange generation of combinations. *Journal of Algorithms*, 9(2):162–180, 1988.

[SA11] M. Shimada and K. Amano. A note on the middle levels conjecture. *arXiv:0912.4564*, September 2011.

[Sav93] C. Savage. Long cycles in the middle two levels of the Boolean lattice. *Ars Combin.*, 35-A:97–108, 1993.

[Sav97] C. Savage. A survey of combinatorial Gray codes. *SIAM Rev.*, 39(4):605–629, 1997.

[Sed77] R. Sedgewick. Permutation generation methods. *ACM Comput. Surv.*, 9(2):137–164, June 1977.

[Sim91] J. Simpson. Hamiltonian bipartite graphs. In *Proceedings of the Twenty-second Southeastern Conference on Combinatorics, Graph Theory, and Computing (Baton Rouge, LA, 1991)*, volume 85, pages 97–110, 1991.

[Sim94] J. Simpson. On uniform subset graphs. *Ars Combin.*, 37:309–318, 1994.

[SS04] I. Shields and C. Savage. A note on Hamilton cycles in Kneser graphs. *Bull. Inst. Combin. Appl.*, 40:13–22, 2004.

[SSS09] I. Shields, B. Shields, and C. Savage. An update on the middle levels problem. *Discrete Math.*, 309(17):5271–5277, 2009.

[Sta99] R. Stanley. *Enumerative combinatorics. Vol. 2*, volume 62 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1999. With a foreword by Gian-Carlo Rota and appendix 1 by Sergey Fomin.

[SW95] C. Savage and P. Winkler. Monotone Gray codes and the middle levels problem. *J. Combin. Theory Ser. A*, 70(2):230–248, 1995.

[SW14] B. Stevens and A. Williams. The coolest way to generate binary strings. *Theory Comput. Syst.*, 54(4):551–577, 2014.

[Tro62] H. Trotter. Algorithm 115: Perm. *Commun. ACM*, 5(8):434–435, August 1962.

[Wil89] H. Wilf. *Combinatorial algorithms: an update*, volume 55 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1989.

[Win04] P. Winkler. *Mathematical puzzles: a connoisseur's collection*. A K Peters, Ltd., Natick, MA, 2004.

[www] currently `http://www.math.tu-berlin.de/~muetze` and `http://people.inf.ethz.ch/njerri`.