Polynomial-delay generation of functional digraphs up to isomorphism

Antonio E. Porreca[1] and Ekaterina Timofeeva[2]

[1] Université Publique
antonio.porreca@lis-lab.fr
[2] Aix-Marseille Université, CNRS, LIS, Marseille, France
ekatim239@gmail.com

Abstract. We describe a procedure for the generation of functional digraphs up to isomorphism; these are digraphs with uniform outdegree 1, also called mapping patterns, finite endofunctions, or finite discrete-time dynamical systems. This procedure is based on an algorithm for the generation of connected functional digraphs, which is then generalised to arbitrary ones. Both algorithms have an $O(n^3)$ delay between consecutive outputs. We also provide a proof-of-concept implementation of the algorithms.

1 Introduction and motivation

A finite, discrete-time dynamical system (A, f), called in the following just a dynamical system for brevity, is simply a finite set A of states together with a function $f: A \to A$ describing the evolution of the system in time. A dynamical system can be equivalently described by its transition digraph, which has $V = A$ as its set of vertices and $E = \{(a, f(a)) : a \in A\}$ as its set of arcs, that is, each state has an outgoing edge pointing to the next state. Since the dynamical systems we are dealing with are deterministic, their transition digraphs are all and only the digraphs having uniform outdegree 1, that is, *functional digraphs*.

The synchronous execution of two dynamical systems A and B gives a dynamical system $A \otimes B$, whose transition digraph is the *direct product* [8] of the transition digraphs of A and B. This product, together with a disjoint union operation of sum, gives a semiring structure over dynamical systems up to isomorphism [6] with some interesting algebraic properties, notably the lack of unique factorisation into irreducible digraphs. In order to develop the theory of the semiring of dynamical systems, it is useful to be able to find examples and counterexamples to our conjectures, and this often requires us to be able to efficiently generate all functional digraphs of a given number n of vertices up to isomorphism. Remark that the number of non-isomorphic functional digraphs over n vertices (sequence A001372 on the OEIS [13]) is exponential, asymptotically $c \times d^n/\sqrt{n}$ for some constants c and $d > 1$ [12]; as a consequence, the kind of efficiency we must look for is the ability to generate the first output in polynomial time, and the delay between consecutive outputs must be polynomial as well.

Since the outdegree of each vertex is exactly 1 and the number of vertices is finite, the general shape of a functional digraph is a disjoint union of connected components, each consisting of a limit cycle $\langle v_1, \ldots, v_k \rangle$, where the vertices $v_1, \ldots, v_k$ are the roots of k directed unordered rooted trees (simply referred to as *trees* in the following), with the arcs pointing towards the root.

Enumeration and generation problems for some classes of graphs have been analysed in the literature. For instance, efficient isomorphism-free generation algorithms for rooted, unordered trees are well known, even requiring only amortised constant time [1], and there exist polynomial delay algorithms for the isomorphism-free generation of *undirected* graphs [7]. More general techniques for the generation of combinatorial objects have been described by McKay [10]. For a

practical implementation of generators for several classes of graphs we refer the reader to software such as `nauty` and `Traces` [11].

However, the class of functional digraphs does not seem to have been considered yet from the point of view of efficient generation algorithms. Here we first propose a $O(n^3)$-delay algorithm for the generation of *connected* functional digraphs, based on an isomorphism code (which avoids generating multiple isomorphic digraphs) and on a novel approach where the successor of the current digraph is obtained by merging trees having adjacent roots along the limit cycle. This algorithm is then used as a subroutine in order to generate *all*, non necessarily connected functional digraphs with the same $O(n^3)$ delay.

2 Isomorphism codes for connected functional digraphs

In order to avoid generating multiple isomorphic functional digraphs, we first define a *canonical representation* based on an isomorphism code, which would also allow us to check in polynomial time whether two given functional digraphs are isomorphic when given by another representation (e.g., adjacency lists or matrices)[3].

Isomorphism codes for unordered rooted trees (which can be taken as directed with the arcs pointing towards the root, as is needed in our case) are well known in the literature; for instance, *level sequences* (sequences node depths given by a preorder traversal of the tree, arranged in lexicographic order) can be used for this purpose [1]. Here we adopt a solution described by Valiente [15], which has the additional property that the isomorphism code of a subtree is itself a valid tree isomorphism code.

Definition 1 (code of a tree). *Let $T = (V, E)$ be a tree. Then, the* isomorphism code of T *is the sequence of integers*

$$\operatorname{code} T = \langle |V| \rangle ^\frown \operatorname{code} T_1 {}^\frown \cdots {}^\frown \operatorname{code} T_k$$

where $T_1, \ldots, T_k$ are the immediate subtrees of T, i.e., the subtrees having as roots the predecessors of the root of T, arranged in lexicographically nondecreasing order of code, and $^\frown$ denotes sequence concatenation. In particular, if $|V| = 1$, i.e., if T only consists of the root, then $\operatorname{code} T = \langle 1 \rangle$.

For simplicity, in the rest of the paper, we identify a tree with its own code, i.e., we often write T instead of $\operatorname{code} T$. Furthermore, we denote the number of vertices of a tree T with the symbol $|T|$.

Since a connected functional digraph consists of a sequence of trees arranged along a cycle, and all rotations of the sequence are equivalent, we choose a canonical one as its isomorphism code.

Definition 2 (code of a connected functional digraph). *The* isomorphism code *of a* connected *functional digraph $C = (V, E)$ is the lexicographically minimal rotation*

$$\operatorname{code} C = \langle \operatorname{code} T_1, \ldots, \operatorname{code} T_k \rangle$$

of the sequence of isomorphism codes of its trees taken in order along the cycle, i.e., such that for all integer r we have

$$\langle \operatorname{code} T_1, \ldots, \operatorname{code} T_k \rangle \le \langle \operatorname{code} T_{1+(1+r) \bmod k}, \ldots, \operatorname{code} T_{1+(k+r) \bmod k} \rangle.$$

[3] Since functional digraphs are planar, they can actually be checked for isomorphism in linear time [4] or in logarithmic space [5].

For brevity, we refer to connected functional digraphs as *components* and, as with trees, we identify a component C with its own code. A valid code for a component C is also called a *canonical form* of C; unless otherwise specified, in the rest of the paper we consider all components to be in canonical form. By $|C|$ we denote the number of vertices of component C.

Isomorphism codes for arbitrary (i.e., non necessarily connected) functional digraphs will be defined later, in Section 4.

3 Generation of connected functional digraphs

The enumeration of *connected* functional digraphs is based on the following *tree merging* operation, to be applied to at least two trees (with the first being trivial, i.e., only having a root node) adjacent along the cycle.

Definition 3 (tree merging). *Let $\langle T_1, T_2, \ldots, T_k \rangle$ be a sequence of trees such that T_1 is trivial. Then* $\operatorname{merge} \langle T_1, T_2, \ldots, T_k \rangle$ *is the tree obtained by connecting $T_2, \ldots, T_k$ as immediate subtrees of T_1.*

This operation could, in principle, be applied even if T_1 has more than one node, by adding $T_2, \ldots, T_k$ as immediate subtrees to the root of T_1; however, as will be proved later, this is not needed for our purposes.

The following is an immediate consequence of the definition of the merging operation:

Proposition 4. *Let $\langle T_1, T_2, \ldots, T_k \rangle$ be a lexicographically nondecreasing sequence of trees with T_1 trivial. Then*

$$\operatorname{merge} \langle T_1, T_2, \ldots, T_k \rangle = \left\langle \sum_{i=1}^{k} |T_i| \right\rangle ^\frown T_2 ^\frown \cdots ^\frown T_k.$$

Notice that, even when starting from a valid component code and even when applying it to a lexicographically nondecreasing sequence of trees, this tree merging operation might produce invalid codes.

Example 5. Consider the following component:

$$\langle T_1, T_2, T_3 \rangle = \langle \langle 1 \rangle, \langle 1 \rangle, \langle 1 \rangle \rangle.$$

By merging T_1 and T_2 we obtain

$$\langle \operatorname{merge} \langle T_1, T_2 \rangle, T_3 \rangle = \langle \langle 2, 1 \rangle, \langle 1 \rangle \rangle$$

which is not a valid code, since the rotation $\langle \langle 1 \rangle, \langle 2, 1 \rangle \rangle$ is strictly inferior in lexicographic order.

It is thus necessary to check whether a code obtained by merging a nondecreasing sequence of adjacent trees of a component is indeed a valid code.

We can also define an inverse unmerging operation on trees.

Definition 6 (tree unmerging). *Given a tree T, we say that*

$$\operatorname{unmerge} T = \langle T_1, \ldots, T_k \rangle$$

if and only if $\operatorname{merge} \langle T_1, \ldots, T_k \rangle = T$.

Notice that there is only one way to unmerge a tree, i.e., detaching its immediate subtrees and putting them in a sequence after the trivial tree (i.e., the former root), and thus unmerge is indeed a well-defined function.

We can prove that unmerging trees in a component in canonical form in a left-to-right fashion always gives another canonical form.

Lemma 7. *Let $C = \langle T_1, \ldots, T_k \rangle$ be a component in canonical form, and let T_h be the leftmost nontrivial tree of C. Then*

$$C' = \langle T_1, \ldots, T_{h-1} \rangle ^\frown \operatorname{unmerge} T_h {}^\frown \langle T_{h+1}, \ldots, T_k \rangle$$

is also a canonical form, i.e., it is the minimum of its own rotations, and contains strictly more trees than C. We call the resulting component C' the component-unmerge *of C, in symbols* c-unmerge C.

Proof. By hypothesis, C begins with a sequence $\langle T_1 \ldots, T_{h-1} \rangle$ of trivial trees of length $h-1$. The unmerging of T_h has the form

$$\operatorname{unmerge} T_h = \langle S_1, \ldots, S_\ell \rangle$$

where S_1 is trivial. As a consequence, C' begins with a maximal sequence X of trivial trees of length at least h: the original ones $\langle T_1, \ldots, T_{h-1} \rangle$, together with S_1, obtained by detaching the root of T_h, and possibly other trivial trees following S_1.

A rotation of C' that is strictly inferior to C' would have to have another sequence Y of trivial trees with $|Y| \geq |X|$ and Y *separated from* X, i.e., with at least one intervening nontrivial tree. Such sequence Y did not already exist in C, because otherwise the rotation of C starting with Y would be strictly less than C, contradicting the hypothesis that C is a minimal rotation.

As a consequence, the supposed sequence Y of trivial trees would have to be generated by the unmerging of T_h. But all trivial trees in unmerge T_h occur at the beginning of the sequence, since the subtrees of a tree are always sorted lexicographically, and are thus adjacent to the trivial trees $\langle T_1 \ldots, T_{h-1}, S_1 \rangle$. As a consequence, C' is the minimal rotation and thus a canonical form.

Since the unmerging of any nontrivial tree produces at least two trees, the digraph C' has strictly more trees than C, which completes the proof. ☐

We can now define the set of merges of a given component.

Definition 8. *Let $C = \langle T_1, \ldots, T_k \rangle$ be a component and let*

$$C_{\ell,r} = \langle T_1, \ldots, T_{\ell-1}, \operatorname{merge} \langle T_\ell, \ldots, T_r \rangle, T_{r+1}, \ldots, T_k \rangle$$

for $1 \leq \ell < r \leq k$. Then, we denote the set of possible merges of C by

$$\operatorname{merges} C = \left\{ C_{\ell,r} \;\middle|\; \begin{array}{c} 1 \leq \ell < r \leq k, \textit{tree } T_\ell \textit{ is trivial,} \\ \langle T_\ell, \ldots, T_r \rangle \textit{ is lexicographically nondecreasing,} \\ C_{\ell,r} \textit{ is in canonical form, and } \operatorname{c-unmerge} C_{\ell,r} = C \end{array} \right\}.$$

Notice that, since c-unmerge is a function, for distinct components C_1 and C_2 we always have $\operatorname{merges} C_1 \cap \operatorname{merges} C_2 = \varnothing$.

Example 9. If we did not require $\operatorname{c-unmerge} C_{\ell,r} = C$ in Definition 8, the sets of merges of two distinct components would not be necessarily disjoint; for instance, both $\langle \langle 1 \rangle, \langle 1 \rangle, \langle 1 \rangle, \langle 3, 2, 1 \rangle \rangle$ and $\langle \langle 1 \rangle, \langle 2, 1 \rangle, \langle 1 \rangle, \langle 2, 1 \rangle \rangle$ would share an element:

$$\begin{aligned} \langle \langle 1 \rangle, \langle 2, 1 \rangle, \langle 3, 2, 1 \rangle \rangle &= \langle \langle 1 \rangle, \operatorname{merge} \langle \langle 1 \rangle, \langle 1 \rangle \rangle, \langle 3, 2, 1 \rangle \rangle \\ &= \langle \langle 1 \rangle, \langle 2, 1 \rangle, \operatorname{merge} \langle \langle 1 \rangle, \langle 2, 1 \rangle \rangle \rangle. \end{aligned}$$

3.1 Algorithm for the generation of connected functional digraphs

We can now describe an algorithm generating all components over n vertices by exploiting the notion of merging adjacent trees along the limit cycle.

Algorithm 1 (successor of a component)**.** *On input $C = \langle T_1, \ldots, T_k \rangle$ with $|C| = n$ vertices:*

1.1 If $\operatorname{merges} C \neq \varnothing$, then return $\min(\operatorname{merges} C)$. Otherwise go to step 1.2.
1.2 If all trees of C are trivial, then go to step 1.3. Otherwise let $U = \text{c-unmerge}\, C$ and, if the set $\{M \in \operatorname{merges} U : M > C\}$ is nonempty, then return the minimum of such set. Otherwise, set $C := U$ and repeat step 1.2.
1.3 Return $\underbrace{\langle \langle 1 \rangle, \ldots, \langle 1 \rangle \rangle}_{n+1 \text{ times}}$, the first component over $n+1$ vertices.

Let us show the execution of Algorithm 1 on an example.

Example 10. Let us compute all 9 components over $n = 4$ vertices. We begin with the cycle of length n:

$$C_1 = \langle \langle 1 \rangle, \langle 1 \rangle, \langle 1 \rangle, \langle 1 \rangle \rangle$$

The set of merges of C_1 is

$$\operatorname{merges} C_1 = \big\{ \langle \langle 4, 1, 1, 1 \rangle \rangle, \langle \langle 1 \rangle, \langle 3, 1, 1 \rangle \rangle, \langle \langle 1 \rangle, \langle 1 \rangle, \langle 2, 1 \rangle \rangle \big\},$$

The sequences $\langle \langle 2, 1 \rangle, \langle 1 \rangle, \langle 1 \rangle \rangle, \langle \langle 3, 1, 1 \rangle, \langle 1 \rangle \rangle$, and $\langle \langle 1 \rangle, \langle 2, 1 \rangle, \langle 1 \rangle \rangle$, while obtained by merging adjacent trees of C_1, are not valid codes of components, since they are not in canonical form (i.e, they are not their own minimal rotation).

In order to compute the next component, we take the lexicographically minimal $C_2 \in \operatorname{merges} C_1$, namely

$$C_2 = \langle \langle 1 \rangle, \langle 1 \rangle, \langle 2, 1 \rangle \rangle$$

We repeat this procedure twice, taking the minimum C_3 of $\operatorname{merges} C_2$ and C_4 of $\operatorname{merges} C_3$:

$$C_3 = \langle \langle 1 \rangle, \langle 3, 2, 1 \rangle \rangle$$
$$C_4 = \langle \langle 4, 3, 2, 1 \rangle \rangle$$

We can not repeat these steps any further, since $\operatorname{merges} C_4 = \varnothing$. Therefore, we compute $C_3 = \text{c-unmerge}\, C_4$ and try to obtain the lexicographically minimal C_5 such that $C_5 \in \operatorname{merges} C_3$ and $C_5 > C_4$. Such C_5 does not exist, since C_4 is the only element in $\operatorname{merges} C_3$, therefore we repeat the same process with $C_2 = \text{c-unmerge}\, C_3$. We try to obtain the lexicographically minimal C_5, such that $C_5 \in \operatorname{merges} C_2$ and $C_5 > C_3$. Indeed, such C_5 exists:

$$C_5 = \langle \langle 2, 1 \rangle, \langle 2, 1 \rangle \rangle$$

Since $\operatorname{merges} C_5 = \varnothing$, we try to obtain the lexicographically minimal component C_6 such that $C_6 \in \operatorname{merges}\, (\text{c-unmerge}\, C_5) = \operatorname{merges} C_2$ and $C_6 > C_5$. We obtain:

$$C_6 = \langle \langle 4, 1, 2, 1 \rangle \rangle$$

We have $\text{merges}\, C_6 = \varnothing$ and there does not exist C_7 such that $C_7 > C_6$ and $C_7 \in \text{merges}\, C_2$. Then, we look for C_7 as the next component such that $C_7 \in \text{merges}\,(\text{c-unmerge}\, C_2) = \text{merges}\, C_1$ and $C_7 > C_2$. Such C_7 exists:

$$C_7 = \langle\langle 1\rangle, \langle 3, 1, 1\rangle\rangle$$

We compute the minimal of its merges:

$$C_8 = \langle\langle 4, 3, 1, 1\rangle\rangle$$

Since C_8 is the only element of $\text{merges}\, C_7$, we compute $C_9 \in \text{merges}\,(\text{c-unmerge}\, C_7) = \text{merges}\, C_1$ and $C_9 > C_7$. Such C_9 exists:

$$C_9 = \langle\langle 4, 1, 1, 1\rangle\rangle$$

Here the generation for $n = 4$ vertices halts, since $\text{merges}\, C_9 = \varnothing$ and C_9 is the lexicographic maximum of $\text{merges}\, C_1$.

3.2 Proof of correctness

In this section we prove the correctness of Algorithm 1.

Lemma 11. *For each n we have*

$$\max(\text{merges}\, \underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n \text{ times}}) = \langle\langle n, \underbrace{1, \ldots, 1}_{n-1 \text{ times}} \rangle\rangle.$$

Proof. For $n = 1$ we have $\text{merges}\, \langle\langle 1\rangle\rangle = \varnothing$, i.e, there are no possible merges. For $n > 1$

$$\text{merges}\, \underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n \text{ times}} = \left\{ \begin{array}{c} \langle \underbrace{\langle 1\rangle, \ldots, \langle 1\rangle}_{n-2 \text{ times}}, \langle 2, 1\rangle\rangle, \langle \underbrace{\langle 1\rangle, \ldots, \langle 1\rangle}_{n-3 \text{ times}}, \langle 3, 1, 1\rangle\rangle, \\ \ldots, \\ \langle\langle 1\rangle, \langle n-1, \underbrace{1, \ldots, 1}_{n-2 \text{ times}} \rangle\rangle, \langle\langle n, \underbrace{1, \ldots, 1}_{n-1 \text{ times}} \rangle\rangle \end{array} \right\}$$

All the other tree merges do not give valid codes, since they are not lexicographically minimal rotations. Indeed, the largest element in lexicographical order in this set is the only one that starts with n and not $\langle 1\rangle$. □

Lemma 12. *Let C be a component generated by Algorithm 1. Then, all components in $\text{merges}\, C$ are also generated by Algorithm 1.*

Proof. By induction, we prove that if the statement is true for components C having less than k trees (hypothesis 1), then it is true for components having k trees.

Suppose that C has k trees. If $\text{merges}\, C$ is empty, there is nothing to prove; suppose then that $\text{merges}\, C = \{C_1, \ldots, C_m\}$ in increasing lexicographic order. Then, on input C Algorithm 1 outputs $\min(\text{merges}\, C) = C_1$ in step 1.1. Let us prove that, assuming that C_i has been generated (hypothesis 2), then C_{i+1} is also generated.

Let $D_0 = C_i$ and let $D_{j+1} = \max(\operatorname{merges} D_j)$ for all j such that $\operatorname{merges} D_j \neq \varnothing$. Let D be the unique D_j with $\operatorname{merges} D_j = \varnothing$. Then $D_0 = C_i$ is generated by hypothesis 2, and each D_{j+1} by hypothesis 1, since all the merges of a component D_j have strictly less trees than D_j itself (by Lemma 7 and Definition 8), and $D_0 = C_i$ has strictly less than k trees, since $C_i \in \operatorname{merges} C$. In particular D is generated.

It only remains to show that Algorithm 1 outputs C_{i+1} on input D. This is indeed the case in step 1.2, since each D_{j+1} is $\max(\operatorname{merges} D_j)$ and thus $\{M \in \operatorname{merges} D_j : M > D_{j+1}\} = \varnothing$. The only D_j for which this set is not empty is $D_0 = C_i$, and $\min\{M \in \operatorname{merges} C : M > C_i\} = C_{i+1}$ by our hypothesis on the lexicographic ordering of $\operatorname{merges} C$.

By induction, all elements of $\operatorname{merges} C$ are then generated, which proves the statement for components C of k trees; once again by induction, this shows that the statement holds for all k and concludes the proof. □

Lemma 13. *Suppose that cycle $\underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n \text{ times}}$ is generated by Algorithm 1, and let $C = \langle T_1, \ldots, T_k\rangle$ be a component with $|C| = n$ and $k < n$ trees. Then C is also generated by Algorithm 1.*

Proof. By reverse induction on k. Since $k < n$, component C has at least one nontrivial tree, and thus $C' = \operatorname{c-unmerge} C$ exists. We still have $|C'| = n$ and its number of trees k' is strictly larger than k. Then C' is generated, either because $k' = n$ (and thus $C' = \underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n \text{ times}}$), or because $k' < n$ and thus the induction hypothesis applies. But then C is also generated by Lemma 12, since $C \in \operatorname{merges} C'$. □

Theorem 14. *If Algorithm 1 is applied repeatedly, zero or more times, starting from $\langle\langle 1\rangle\rangle$, then all components are generated in increasing order of number of vertices.*

In particular, by repeatedly applying Algorithm 1 starting from the component $C = \underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n \text{ times}}$, i.e., the cycle of length n, and stopping before reaching $\underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n+1 \text{ times}}$, then we obtain all components over n vertices.

Proof. It suffices to prove that all cycles $\underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n \text{ times}}$ are generated, and the result will then follow from Lemma 13. We prove that by induction on n. The cycle $\langle\langle 1\rangle\rangle$ is the initial component, and thus it is generated even before applying Algorithm 1.

Let $n > 1$ and $C = \langle\langle n-1, \underbrace{1, \ldots, 1}_{n-2 \text{ times}}\rangle\rangle$. Then $|C| = n - 1$; by induction hypothesis the cycle $\underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n-1 \text{ times}}$ is generated, and thus C is also generated by Lemma 13. On input C, Algorithm 1 behaves as follows. No merge of C at all can be performed in step 1.1, since there is only one tree. The algorithm in step 1.2 unmerges this tree, giving $U = \underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n-1 \text{ times}}$. Since $C = \max(\operatorname{merges} U)$ by Lemma 11, the algorithm sets $C := U$ and repeats step 1.2. Since all trees are now trivial, the algorithm returns $\underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n \text{ times}}$ in step 1.3.

In order to show that all components over n vertices are obtained by repeated application of Algorithm 1 starting from $\underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n \text{ times}}$, it suffices to notice that all components over n vertices

are generated after all components over $n-1$ vertices (if any) and before all components over $n+1$ vertices. Indeed, in steps 1.1 and 1.2 the number of vertices never changes, since the only operations are tree merging and unmerging; the only operation increasing the number of vertices happens in step 1.3, when we start generating larger components. □

The following lemma shows that the maximum number of iterations of step 1.2 is actually independent of the size of the component.

Lemma 15. *On input a component $C = \langle T_1, \dots, T_k \rangle$ such that $\operatorname{merges} C = \varnothing$, its successor is obtained by executing step 1.2 of Algorithm 1 at most twice.*

Proof. Since $\operatorname{merges} C = \varnothing$, trees $T_1, \dots, T_k$ are nontrivial, therefore the successor of C is obtained either in step 1.2 or in step 1.3. We first compute $U = \text{c-unmerge}\, C = \langle \langle 1 \rangle, T'_2, \dots, T'_{k'} \rangle$ by unmerging the leftmost nontrivial tree T_1 of C, which gives a trivial tree in the first position. If all other trees of U are also trivial, then the successor is returned in step 1.3.

Otherwise, if the set $\{M \in \operatorname{merges} U : M > C\}$ is nonempty, then we return its minimum, and the successor of C is thus obtained by executing step 1.2 of Algorithm 1 only once.

Otherwise, we set $C := U = \langle \langle 1 \rangle, T'_2, \dots, T'_{k'} \rangle$ and repeat step 1.2 a second time. Now, by unmerging the leftmost nontrivial tree of C we recompute $U = \langle \langle 1 \rangle, \langle 1 \rangle, T''_3, \dots, T''_{k''} \rangle$, which begins with at least two trivial trees (the second trivial tree is either T'_2, or obtained by unmerging it). Then, there exists at least one component M obtained from U by merging adjacent trees, namely $M = \langle T \rangle$ with $T = \operatorname{merge} \langle \langle 1 \rangle, \langle 1 \rangle, T''_3, \dots, T''_{k''} \rangle$; furthermore, we have $M > C$ since C begins with a trivial tree and M with a nontrivial one. Then we return M as the output of Algorithm 1. □

We still need to prove that no component is generated multiple times by Algorithm 1.

Lemma 16. *Algorithm 1 never outputs the same cycle $C = \underbrace{\langle \langle 1 \rangle, \dots, \langle 1 \rangle \rangle}_{n \text{ times}}$ twice.*

Proof. The component $C = \langle \langle 1 \rangle \rangle$ is never output by Algorithm 1, since it is not the merge of any component (and thus it is not generated in step 1.1 or 1.2) and it is too small to be generated in step 1.3.

All cycles of length $n > 1$ are generated in step 1.3 from a smaller component: they can not be a result of merges in steps 1.1 or 1.2, since all their trees are trivial. The component $C = \langle \langle 1 \rangle, \langle 1 \rangle \rangle$ is thus generated in step 1.3 from a component of size 1. But there is only one such component, namely $\langle \langle 1 \rangle \rangle \langle \langle 1 \rangle \rangle$.

Now let $n \geq 2$ and suppose that the cycle $C = \underbrace{\langle \langle 1 \rangle, \dots, \langle 1 \rangle \rangle}_{n+1 \text{ times}}$ is generated by Algorithm 1 in step 1.3 from an input A. Then $\operatorname{merges} A = \varnothing$ (otherwise C would be output in step 1.1), and thus A cannot be a cycle, i.e., it has a nontrivial tree. Hence, Algorithm 1 does not immediately skip step 1.2. Then it must be the case that $A = \max(\operatorname{merges}(\text{c-unmerge}\, A))$, since C is not output in this step.

If step 1.2 is only executed once, then $\text{c-unmerge}\, A$ only has trivial trees, that is, $\text{c-unmerge}\, A$ is the cycle $\underbrace{\langle \langle 1 \rangle, \dots, \langle 1 \rangle \rangle}_{n \text{ times}}$, and then $A = \langle \langle n, \underbrace{1, \dots, 1}_{n-1 \text{ times}} \rangle \rangle$ by Lemma 11.

If step 1.2 were executed exactly twice, that would imply that $\text{c-unmerge}\, A$ has nontrivial trees, that $\text{c-unmerge}\, A = \max(\operatorname{merges}(\text{c-unmerge}(\text{c-unmerge}\, A)))$, otherwise the algorithm would halt in step 1.2, and that $\text{c-unmerge}(\text{c-unmerge}\, A)$ only has trivial trees, that is, it is the cycle $\underbrace{\langle \langle 1 \rangle, \dots, \langle 1 \rangle \rangle}_{n \text{ times}}$. Since $\text{c-unmerge}\, A = \max(\operatorname{merges}(\underbrace{\langle \langle 1 \rangle, \dots, \langle 1 \rangle \rangle}_{n \text{ times}}))$, then by Lemma 11

c-unmerge $A = \langle\langle n, \underbrace{1, \dots, 1}_{n-1 \text{ times}} \rangle\rangle$. But then $\text{merges}(\text{c-unmerge}\, A) = \varnothing$, which is impossible because at least $A \in \text{merges}(\text{c-unmerge}\, A)$.

Step 1.2 is never executed more than twice by Lemma 15, which implies that there is only one input A for Algorithm 1 giving output C. □

Lemma 17. *Let C be a component with at least one nontrivial tree. Then, it is impossible for Algorithm 1 to generate C twice, once in step 1.1 and once in step 1.2.*

Proof. By contradiction, suppose that C is generated on input A in step 1.1. Then we have $C = \min(\text{merges}\, A)$, and thus in particular $C \in \text{merges}\, A$.

Suppose that C is also generated on input B in step 1.2. Then $C \in \text{merges}(\text{c-unmerge}\, D)$ for some D, but $C > D$ in lexicographic order. Since $D \in \text{merges}(\text{c-unmerge}\, D)$, we have $C \neq \min(\text{merges}(\text{c-unmerge}\, D))$.

Since the sets of merges of distinct components are disjoint, necessarily $A = \text{c-unmerge}\, D$. But then C is simultaneously the minimum and not the minimum of $\text{merges}\, A$, which is a contradiction. □

Lemma 18. *Let C be a component with at least one nontrivial tree. Then, it is impossible for Algorithm 1 to generate C twice in step 1.1.*

Proof. Suppose that C is generated from both inputs A and B in step 1.1. Then $C = \min(\text{merges}\, A)$ and $C = \min(\text{merges}\, B)$. But then $C \in \text{merges}\, A \cap \text{merges}\, B$, which implies $A = B$. □

Lemma 19. *Let C be a component with at least one nontrivial tree. Then, it is impossible for Algorithm 1 to generate C twice in step 1.2.*

Proof. By Lemma 15, the output C is generated either after one iteration of step 1.2, or after two.

If C is generated from an input A after only *one* iteration, then

$$C = \min\{M \in \text{merges}\, U : M > A\}$$

with $U = \text{c-unmerge}\, A$. Since $C \in \text{merges}\, U$, we also have $U = \text{c-unmerge}\, C$, thus

$$C = \min\{M \in \text{merges}(\text{c-unmerge}\, C) : M > A\}$$

which implies

$$A = \max\{M \in \text{merges}(\text{c-unmerge}\, C) : M < C\}$$

which proves that A only depends on C and is thus unique.

On the other hand, if C is generated from A after *two* iterations, then

$$C = \min\{M \in \text{merges}\, U : M > \text{c-unmerge}\, A\}$$

with $U = \text{c-unmerge}(\text{c-unmerge}\, A)$. Since $C \in \text{merges}\, U$, we also have $U = \text{c-unmerge}\, C$, thus

$$C = \min\{M \in \text{merges}(\text{c-unmerge}\, C) : M > \text{c-unmerge}\, A\}$$

which implies

$$\text{c-unmerge}\, A = \max\{M \in \text{merges}(\text{c-unmerge}\, C) : M < C\}.$$

Furthermore, we have $A = \max(\text{merges}(\text{c-unmerge}\,A))$, otherwise there would exist a component $M \in \text{merges}(\text{c-unmerge}\,A)$ with $M > A$, and Algorithm 1 would halt after only one iteration of step 1.2, a contradiction.

As a consequence, we have

$$A = \max(\text{merges}(\max\{M \in \text{merges}(\text{c-unmerge}\,C) : M < C\}))$$

which only depends on C and is thus unique. □

From Lemmas 16, 17, 18, and 19 we can finally prove that no duplicates whatsoever are generated.

Theorem 20. *Algorithm 1 never generates the same component multiple times (i.e., it computes an injective function).* □

3.3 Complexity analysis

We can now prove that the proposed generation procedure has an $O(n^3)$ delay between outputs.

Lemma 21. *Given a sequence of trees $\langle T_1, \ldots, T_k \rangle$ with $n = |T_1| + \cdots + |T_k|$ total vertices, it is possible to check in $O(n)$ time if the sequence is its own minimal rotation, and thus the valid code of a component.*

Proof. Let $P = \langle 0 \rangle ^\frown T_1 ^\frown \langle 0 \rangle ^\frown T_2 ^\frown \cdots ^\frown \langle 0 \rangle ^\frown T_k$ be the concatenation of the codes of the given trees, each prefixed with an extra $\langle 0 \rangle$; remark that 0 is strictly less than any integer appearing in the code of a tree. Let P' be the minimal rotation of P; then P' must begin with 0, since it is the minimum element of the sequence, and thus $P' = \langle 0 \rangle ^\frown U_1 ^\frown \langle 0 \rangle ^\frown U_2 ^\frown \cdots ^\frown \langle 0 \rangle ^\frown U_k$, where each U_i is one of the original trees T_j and $\langle U_1, \ldots, U_k \rangle$ is a rotation of $\langle T_1, \ldots, T_k \rangle$.

We claim that $\langle U_1, \ldots, U_k \rangle$ is, more specifically, the *minimal* rotation of $\langle T_1, \ldots, T_k \rangle$. Otherwise, by contradiction, there would exist another rotation $\langle V_1, \ldots, V_k \rangle < \langle U_1, \ldots, U_k \rangle$. Let V_i and U_i be the leftmost trees such that $V_i \neq U_i$. Then

$$\underbrace{\langle 0 \rangle ^\frown V_1 ^\frown \cdots ^\frown \langle 0 \rangle}_{V} {}^\frown V_i ^\frown \cdots ^\frown \langle 0 \rangle ^\frown V_k < \underbrace{\langle 0 \rangle ^\frown U_1 ^\frown \cdots ^\frown \langle 0 \rangle}_{U} {}^\frown U_i ^\frown \cdots ^\frown \langle 0 \rangle ^\frown U_k = P'$$

since the two prefixes U and V are identical and $V_i < U_i$. But this contradicts the assumption that P' is the minimal rotation of P.

Then P is its own minimal rotation if and only if $\langle T_1, \ldots, T_k \rangle$ is a minimal rotation. Since P is a sequence of integers, the former property can be checked in linear time by using a lexicographically minimal rotation algorithm, such as Booth's [2,3]. □

Theorem 22. *Algorithm 1 takes $O(n^3)$ time, which is thus the delay for the generation of the successor of any components over n vertices.*

Proof. Let $C = \langle T_1, \ldots, T_k \rangle$ the input component.

- If the successor of C generated by Algorithm 1 is obtained in step 1.1, then:
  - Generation of $\text{merges}\,C$ takes $O(n^3)$ operations, since we go through all ℓ and r such that $1 \leq \ell < r \leq k$, where $k = n$ in the worst case, which takes $O(n^2)$ operations. For each ℓ and r we check if T_ℓ is trivial ($O(1)$ operations), if $\langle T_\ell, \ldots, T_r \rangle$ is lexicographically nondecreasing ($O(n)$ operations), if $C_{\ell,r}$ is in canonical form, which takes $O(n)$ operations by Lemma 21, and if $\text{c-unmerge}\,C_{\ell,r} = C$, which takes $O(n)$ operations. Therefore, overall this generation takes $O(n \cdot n^2) = O(n^3)$ operations, since $\text{merges}\,C$ has $O(n^2)$ elements ($k \leq n$ in Definition 8).

- In the end, we choose $\min(\text{merges}\, C)$, which takes $O(n^3)$ operations.

  Therefore, the successor of C obtained in step 1.1 is obtained with a $O(n^3)$ delay.

- If the successor of C generated by Algorithm 1 is obtained in step 1.2, then:
  - Computing $U = \text{c-unmerge}\, C$ takes $O(n)$ operations.
  - Then, the generation of $\text{merges}\, U$ takes $O(n^3)$ operations, as previously discussed.
  - Then, for each element M of $\text{merges}\, U$ we have to check if $M > C$, which takes $O(n)$ operations. Therefore, overall this verification takes $O(n \cdot n^2) = O(n^3)$ operations, since $\text{merges}\, U$ has $O(n^2)$ elements.
  - In the end, we choose the minimum M among the components remaining from the previous steps, if any exist, which takes $O(n^3)$ operations.
  - If such M does not exist, we set $C := U$ and repeat step 1.2.

  By Lemma 15 step 1.2 is executed at most twice. Therefore, the successor of C obtained in step 1.1 is obtained with a $O(n^3)$ delay, which includes the operations of step 1.1.

- If the successor of C generated by Algorithm 1 is obtained in step 1.3, then the delay is $O(n^3)$ due to the execution of steps 1.1 and 1.2, since we simply need to create $\underbrace{\langle\langle 1\rangle, \ldots, \langle 1\rangle\rangle}_{n+1 \text{ times}}$, the first component over $n + 1$ vertices, which takes $O(n)$ extra time. □

4 Generation of arbitrary functional digraphs

We can now exploit Algorithm 1 as a subroutine in order to devise an efficient algorithm for the generation of arbitrary (non necessarily connected) functional digraphs. More precisely, like Algorithm 1, we will be able to compute the successor of a given functional digraph under some ordering. As a subroutine, we exploit an algorithm for generating partitions of an integer n [9].

An arbitrary functional digraph can be represented by a sequence of its connected components. Since there is no intrinsic order among components, any permutation of the sequence would represent the same functional digraph; as a consequence, once again we choose a canonical permutation.

Definition 23 (code of a functional digraph). *Let $G = (V, E)$ be an arbitrary functional digraph having m connected components $C_1, \ldots, C_m$. Then, the* isomorphism code *of G is the sequence*

$$\text{code}\, G = \langle \text{code}\, C_1, \ldots, \text{code}\, C_m \rangle.$$

where the connected components $C_1, \ldots, C_m$ are in nondecreasing order of generation by means of Algorithm 1.

As usual, we identify a functional digraph G with its own code and refer to it as a canonical form for G, and we denote its number of vertices by $|G|$.

Notice how the isomorphism code for a functional digraph resembles a PQ-tree, a data structure representing permutations of a given set of elements which, incidentally, is used to efficiently check isomorphic graphs of certain classes [4]. However, for our application we need to represent the equivalence of all permutations of the components $C_1, \ldots, C_m$, as well as the equivalence of all rotations of the trees $T_1, \ldots, T_k$ of a component C_i, and this latter condition is not represented directly in a PQ-tree.

For the generation of arbitrary functional digraphs, we modify the partition generation algorithm by Kelleher and O'Sullivan [9] in such a way that, if no further partition of n exists, then the lexicographically minimum partition of $n + 1$ is output.

Proposition 24 (Kelleher, O'Sullivan [9]). *There exists an algorithm that takes as input a partition of an integer n, represented by a sequence of nondecreasing integers, and returns the next partition of n in lexicographic order (if there exists a next partition).* □

Algorithm 2 (successor of an integer partition)**.** *On input a partition P of the integer n, return the next partition of n in lexicographic order, if any; if P was the last partition of n, then return the first partition of $n+1$ in lexicographic order, i.e., $\underbrace{(1, 1, \dots, 1)}_{n+1 \text{ times}}$.*

This allows us to formalise the generation of arbitrary functional digraphs.

Algorithm 3 (successor of an arbitrary functional digraph)**.** *On input $G = \langle C_1, \dots, C_m \rangle$, a functional digraph over n vertices having components $C_1, \dots, C_m$:*

3.1 If there exists a component such that its successor given by Algorithm 1 has the same size, then let C_h be the rightmost such component, and return

$$\langle C_1, \dots, C_{h-1}, C'_h, C'_{h+1}, \dots, C'_m \rangle$$

where C'_h is the aforementioned successor of C_h and, for all $i > h$, the component C'_i is a copy of C'_h if $|C_i| = |C'_h|$; otherwise $|C_i| > |C'_h|$, and then we let C_i be the first component over $|C_i|$ vertices, i.e., the cycle of length $|C_i|$.

3.2 If no such component exists, let $P = (|C_1|, \dots, |C_m|)$ the partition of the integer n corresponding to the sizes of the components of G, and let $Q = (q_1, \dots, q_{m'})$ be the next partition according to Algorithm 2, then return $\langle C'_1, \dots, C'_{m'} \rangle$, where each C'_i is the first component over q_i vertices, i.e., the cycle of length q_i.

As an example, let us compute all 19 functional digraphs over 4 vertices.

Example 25. We begin with $n = 4$ self loops:

$$G_1 = \langle\langle\langle 1 \rangle\rangle, \langle\langle 1 \rangle\rangle, \langle\langle 1 \rangle\rangle, \langle\langle 1 \rangle\rangle\rangle$$

In order to compute the next functional digraph, we try finding a component such that its successor given by Algorithm 1 has the same size. G_1 does not have such a component, since all the components are self loops, thus its successor G_2 is obtained by computing the successor of its partition via Algorithm 2. Let $P_1 = (1, 1, 1, 1)$ be the partition of G_1, then the partition of G_2 is $P_2 = (p_1, p_2, p_3) = (1, 1, 2)$. Each component C_i of G_2 is the first component over p_i vertices, i.e., the cycle of length p_i, thus

$$G_2 = \langle\langle\langle 1 \rangle\rangle, \langle\langle 1 \rangle\rangle, \langle\langle 1 \rangle, \langle 1 \rangle\rangle\rangle$$

Since $\langle\langle 1 \rangle, \langle 1 \rangle\rangle$ has a successor of the same size according to the Algorithm 1, the next generated digraph is

$$G_3 = \langle\langle\langle 1 \rangle\rangle, \langle\langle 1 \rangle\rangle, \langle\langle 2, 1 \rangle\rangle\rangle$$

There is no component of G_3 such that its successor given by Algorithm 1 has the same size, thus its successor G_4 must be obtained with the next partition $P_3 = (1, 3)$ where each component is the first generated component over corresponding number of vertices:

$$G_4 = \langle\langle\langle 1 \rangle\rangle, \langle\langle 1 \rangle, \langle 1 \rangle, \langle 1 \rangle\rangle\rangle$$

Since $\langle\langle 1\rangle, \langle 1\rangle, \langle 1\rangle\rangle$ has a successor of the same size according to the Algorithm 1, the next generated digraph is

$$G_5 = \langle\langle\langle 1\rangle\rangle, \langle\langle 1\rangle, \langle 2, 1\rangle\rangle\rangle$$

As before, by taking a component with the successor of the same size twice we obtain two following digraphs:

$$G_6 = \langle\langle\langle 1\rangle\rangle, \langle\langle 3, 2, 1\rangle\rangle\rangle$$

$$G_7 = \langle\langle\langle 1\rangle\rangle, \langle\langle 3, 1, 1\rangle\rangle\rangle$$

There is no component of G_7 such that its successor given by Algorithm 1 has the same size, thus its successor G_8 must be obtained with the next partition $P_4 = (2, 2)$ where each component is the first generated component over corresponding number of vertices:

$$G_8 = \langle\langle\langle 1\rangle, \langle 1\rangle\rangle, \langle\langle 1\rangle, \langle 1\rangle\rangle\rangle$$

We take the rightmost component of G_8 that has a successor of the same size and we obtain the next generated digraph:

$$G_9 = \langle\langle\langle 1\rangle, \langle 1\rangle\rangle, \langle\langle 2, 1\rangle\rangle\rangle$$

By repeating the same step but putting a copy of the first component so that the obtained digraph is in canonical form, we generate

$$G_{10} = \langle\langle\langle 2, 1\rangle\rangle, \langle\langle 2, 1\rangle\rangle\rangle$$

There exists no component of G_{10} such that its successor has the same size, therefore we move to the next partition $P_5 = (4)$ and we return the digraph with the first component over 4 vertices, i.e., the cycle of length 4.

$$G_{11} = \langle\langle\langle 1\rangle, \langle 1\rangle, \langle 1\rangle, \langle 1\rangle\rangle\rangle$$

By repeatedly taking the successor of the only component $\langle\langle 1\rangle, \langle 1\rangle, \langle 1\rangle, \langle 1\rangle\rangle$ we obtain all remaining functional digraphs over 4 vertices, namely:

$$G_{12} = \langle\langle\langle 1\rangle, \langle 1\rangle, \langle 2, 1\rangle\rangle\rangle$$

$$G_{13} = \langle\langle\langle 1\rangle, \langle 3, 2, 1\rangle\rangle\rangle$$

$$G_{14} = \langle\langle\langle 4, 3, 2, 1\rangle\rangle\rangle$$

$$G_{15} = \langle\langle\langle 2, 1\rangle, \langle 2, 1\rangle\rangle\rangle$$

$$G_{16} = \langle\langle\langle 4, 1, 2, 1\rangle\rangle\rangle$$

$$G_{17} = \langle\langle\langle 1\rangle, \langle 3, 1, 1\rangle\rangle\rangle$$

$$G_{18} = \langle\langle\langle 4, 3, 1, 1\rangle\rangle\rangle$$

$$G_{19} = \langle\langle\langle 4, 1, 1, 1\rangle\rangle\rangle$$

Here the generation for $n = 4$ vertices halts, since the successor of $\langle\langle 4, 1, 1, 1\rangle\rangle$ has more than 4 vertices, and there is no further partition of size 4.

4.1 Proof of correctness

In order to prove the correctness of Algorithm 3, it is useful to first formalise the enumeration ordering it provides.

Definition 26. *Given two functional digraphs $G_1 = \langle C_1, \dots, C_m\rangle$ and $G_2 = \langle C'_1, \dots, C'_{m'}\rangle$, their order is defined as follows. Let $P_1 = (|C_1|, \dots, |C_m|)$ and $P_2 = (|C'_1|, \dots, |C'_m|)$ be the partitions of $|G_1|$ and $|G_2|$ corresponding to the two digraphs.*

- *If P_1 precedes P_2 (resp., P_2 precedes P_1) by order of generation by Algorithm 2, then G_1 precedes G_2 (resp., G_2 precedes G_1).*
- *If $P_1 = P_2$, then G_1 precedes G_2 (resp., G_2 precedes G_1) if it precedes it in the lexicographic order induced by the ordering of components given by Algorithm 1.*

Lemma 27. *The order given by Definition 26 is a well-order on the set of functional digraphs.*

Proof. The order is total since, given functional digraphs G_1 and G_2, either they have different partitions, and thus are comparable according to the order of generation by Algorithm 2, or they have the same partition, and then they are comparable by the lexicographic order induced by Algorithm 1, which is total by Theorems 14 and 20.

This is a well-order since, given a set S of functional digraphs, we can take the subset containing all digraphs having the minimal partition according to Algorithm 2 (this is also a well-order) and compare them by the lexicographic order induced by Algorithm 1 (another well-order) in order to find the minimum of S. □

Lemma 28. *On input a functional digraph $G = \langle C_1, \dots, C_m\rangle$ in canonical form, the output of Algorithm 3 is also in canonical form.*

Proof. Let G' be the output of Algorithm 3 on input G.

- If G' is obtained in step 3.1, then it has the form

$$G' = \langle C_1, \dots, C_{h-1}, C'_h, C'_{h+1}, \dots, C'_m\rangle.$$

  The prefix $\langle C_1, \dots, C_{h-1}\rangle$ shared with G is nondecreasing since G is in canonical form. Since C'_h is the successor of C_h, this holds also for the prefix $\langle C_1, \dots, C_{h-1}, C'_h\rangle$. Immediately after we have zero or more copies of C'_h, followed by a suffix consisting of larger minimal components (i.e., cycles) of nondecreasing sizes (since the partitions are nondecreasing sequences of integers), and this keeps the sequence in nondecreasing order of generation by Algorithm 1.
- If G' is generated in step 3.2, then it consists of cycles of nondecreasing length, which is a canonical form. □

Lemma 29. *On input a functional digraph, Algorithm 3 outputs its immediate successor in the order of Definition 26.*

Proof. Let $G = \langle C_1, \dots, C_m\rangle$ be a functional digraph.

- If by applying Algorithm 3 on G we obtain its successor G' in step 3.1, then it has the form

$$G' = \langle C_1, \dots, C_{h-1}, C'_h, C'_{h+1}, \dots, C'_m \rangle$$

and it follows G in the order of Definition 26 because the two digraphs share the same partition and G' is larger by lexicographic order, since the two digraphs share the same prefix $\langle C_1, \dots, C_{h-1} \rangle$ and C'_h is the successor of C_h in the order given by Algorithm 1. Suppose that the functional digraph H follows G and precedes G' according to the order of Definition 26; then it also shares the same prefix $\langle C_1, \dots, C_{h-1} \rangle$; since no component is strictly between C_h and C'_h, the next component of H is one of the two. If it is C_h, then G and H share the prefix $\langle C_1, \dots, C_h \rangle$, and since each component of the suffix $\langle C_{h+1}, \dots, C_m \rangle$ of G is maximal among the components of the same size (otherwise C_h would not be the rightmost component with a successor of the same size), then H must share the same suffix, and thus $H = G$. If, on the other hand, the h-th component of H is C'_h, then it shares a prefix $\langle C_1, \dots, C_{h-1}, C'_h \rangle$ with G'. Since each component $\langle C'_{h+1}, \dots, C'_m \rangle$ of the suffix of G' is minimal among the components of the same size, then H must share the same suffix, and thus be identical to G'. This shows that there exists no functional digraph H strictly between G and G', i.e., G' is the immediate successor of G in the order of Definition 26.
- If G' is obtained in step 3.2, then its partition P' immediately follows the partition P of G in the order given by Algorithm 2, thus G' follows G in the order of Definition 26. There is no digraph H strictly between G and G' since G is the last digraph having partition P, and G' is the first having partition P'. □

Theorem 30. *All functional digraphs are obtained, in nondecreasing order by number of vertices, by applying Algorithm 3 zero or more times to the empty digraph. Algorithm 3 never generates two isomorphic functional digraphs (i.e., it computes an injective function).*

Proof. Since the order of Definition 26 is a well-order, and thus in particular a total order, by Lemma 27, and since Algorithm 3 outputs immediate successors by Lemma 29 according to that order, each functional digraph is reachable by consecutive applications of Algorithm 3. Once again by totality of the order and by Lemma 29, distinct inputs will produce distinct outputs, and since every output is in canonical form by Lemma 28, there will be no isomorphic duplicates. □

4.2 Complexity analysis

It is straightforward to prove the efficiency of Algorithm 3.

Theorem 31. *Algorithm 3 takes $O(n^3)$ time, which is thus the delay for the generation of the successor of arbitrary functional digraphs over n vertices.*

Proof. Let $G = \langle C_1, \cdots, C_m \rangle$ be a functional digraph over n vertices.

- If the successor of G generated by Algorithm 3 in step 3.1, then let $n_1 = |C_1|, \dots, n_m = |C_m|$ be the number of vertices of each component. We have $n = n_1 + \cdots + n_m$. In the worst case we have to try to obtain a successor generated by Algorithm 1 for each component, but we always obtain a component of larger size except for C_1. This sequence of operations takes $O(n_1^3) + \cdots + O(n_m^3)$ time. Since $n_1^3 + \cdots + n_m^3 \leq (n_1 + \cdots + n_m)^3 = n^3$, the output is generated in $O(n^3)$ time.
- If the successor of G generated by Algorithm 3 is obtained in step 3.2, then the computation of the partition of G takes linear time. The generation of the next partition according to Algorithm 2 and the construction of the cycles of the corresponding lengths also take linear time. Therefore, the output is generated in $O(n)$ time, plus the $O(n^3)$ time spent in step 3.1. □

5 Implementation

We provide a proof-of-concept implementation of the algorithms by means of the `funkdigen` command-line tool [14]. In its current version, it not yet meant to be an example of optimised software, but rather a straightforward and readable translation in Python 3 of the algorithms described in this paper, hopefully useful as a basis upon which a more efficient version may be developed. The `funkdigen` tools outputs connected or arbitrary functional digraph codes as described in Definitions 2 and 23.

6 Conclusions

We have described the first $O(n^3)$-delay generation algorithm for the class of functional digraphs, both connected and arbitrary, which proves that these classes of graphs can be generated with a polynomial delay for each size n.

It is, of course, an open problem to establish if functional digraphs can be generated with a smaller delay. That would require us to somehow avoid testing $O(n^2)$ possible merges in order to construct the next candidate digraph, and to avoid checking if it is its own minimal rotation at each iteration. Is an amortised constant time delay even possible?

Another line of research would be to describe a variation of this algorithm (based on merging adjacent trees) suitable for more general classes of graphs, without the uniform outdegree 1 constraint. More generally, we hope that the techniques described in this paper can find application to other combinatorial problems.

Acknowledgements Antonio E. Porreca was funded by his salary as a French State agent, and Ekaterina Timofeeva by the undergraduate internship program "Incubateurs de jeunes scientifiques" of Aix-Marseille Université and by the Agence National de la Recherche (ANR) project ANR-18-CE40-0002 FANs. Both authors are affiliated to Aix-Marseille Université, CNRS, LIS, Marseille, France.

We would like to thank Kellogg S. Booth, Jerome Kelleher, Brendan D. McKay, and Kévin Perrot for their useful suggestions and for providing some bibliographic references.

References

1. Beyer, T., Mitchell Hedetniemi, S.: Constant time generation of rooted trees. SIAM Journal on Computing **9**(4), 706–712 (1980), https://doi.org/10.1137/0209055
2. Booth, K.S.: Lexicographically least circular substrings. Information Processing Letters **10**(4–5), 240–242 (1980), https://doi.org/10.1016/0020-0190(80)90149-0
3. Booth, K.S.: Lexicographically least circular substrings. https://www.cs.ubc.ca/~ksbooth/PUB/LCS.shtml (2019), accessed: 2022-10-31, archived at http://web.archive.org/web/20220725150720/https://www.cs.ubc.ca/~ksbooth/PUB/LCS.shtml
4. Booth, K.S., Lueker, G.S.: Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. Journal of Computer and System Sciences **13**(3), 335–379 (1976), https://doi.org/10.1016/S0022-0000(76)80045-1
5. Datta, S., Limaye, N., Nimbhorkar, P., Thierauf, T., Wagner, F.: Planar graph isomorphism is in log-space. In: 2009 24th Annual IEEE Conference on Computational Complexity. pp. 203–214 (2009), https://doi.org/10.1109/CCC.2009.16
6. Dennunzio, A., Dorigatti, V., Formenti, E., Manzoni, L., Porreca, A.E.: Polynomial equations over finite, discrete-time dynamical systems. In: Mauri, G., El Yacoubi, S., Dennunzio, A., Nishinari, K., Manzoni, L. (eds.) Cellular Automata, 13th International Conference on Cellular Automata for Research and Industry, ACRI 2018. Lecture Notes in Computer Science, vol. 11115, pp. 298–306. Springer (2018), https://doi.org/10.1007/978-3-319-99813-8_27

7. Goldberg, L.A.: Efficient algorithms for listing unlabeled graphs. Journal of Algorithms **13**(1), 128–143 (1992), https://doi.org/10.1016/0196-6774(92)90009-2
8. Hammack, R., Imrich, W., Klavžar, S.: Handbook of Product Graphs. Discrete Mathematics and Its Applications, CRC Press, second edn. (2011), https://doi.org/10.1201/b10959
9. Kelleher, J., O'Sullivan, B.: Generating all partitions: A comparison of two encodings. arXiv:0909.2331v2 [cs.DS] (2009), https://doi.org/10.48550/arXiv.0909.2331
10. McKay, B.D.: Isomorph-free exhaustive generation. Journal of Algorithms **26**(2), 306–324 (1998), https://doi.org/10.1006/jagm.1997.0898
11. McKay, B.D., Piperno, A.: Practical graph isomorphism, II. Journal of Symbolic Computation **60**, 94–112 (2014), https://doi.org/10.1016/j.jsc.2013.09.003
12. Meir, A., Moon, J.W.: On random mapping patterns. Combinatorica **4**, 61–70 (1984), https://doi.org/10.1007/BF02579158
13. OEIS Foundation Inc.: Number of mappings (or mapping patterns) from n points to themselves; number of endofunctions. The On-Line Encyclopedia of Integer Sequences (2023), published electronically at https://oeis.org/A001372
14. Porreca, A.E., Timofeeva, E.: `funkdigen`: A generator of functional digraphs up to isomorphism. https://github.com/aeporreca/funkdigen (2023)
15. Valiente, G.: Algorithms on Trees and Graphs. Texts in Computer Science, Springer, 2nd edn. (2021), https://doi.org/10.1007/978-3-030-81885-2