Portugal. Math. (N.S.)
Vol. 74, Fasc. 3, 2017, 173–200
DOI 10.4171/PM/2001

Portugaliae Mathematica
© European Mathematical Society

Two variants of the Froidure–Pin Algorithm for finite semigroups

Julius Jonušas, James D. Mitchell and Markus Pfeiffer

(Communicated by João Araújo and Peter J. Cameron)

Abstract. In this paper, we present two algorithms based on the Froidure–Pin Algorithm for computing the structure of a finite semigroup from a generating set. As was the case with the original algorithm of Froidure and Pin, the algorithms presented here produce the left and right Cayley graphs, a confluent terminating rewriting system, and a reduced word of the rewriting system for every element of the semigroup.

If U is any semigroup, and A is a subset of U, then we denote by $\langle A \rangle$ the least subsemigroup of U containing A. If B is any other subset of U, then, roughly speaking, the first algorithm we present describes how to use any information about $\langle A \rangle$, that has been found using the Froidure–Pin Algorithm, to compute the semigroup $\langle A \cup B \rangle$. More precisely, we describe the data structure for a finite semigroup S given by Froidure and Pin, and how to obtain such a data structure for $\langle A \cup B \rangle$ from that for $\langle A \rangle$. The second algorithm is a lock-free concurrent version of the Froidure–Pin Algorithm.

Mathematics Subject Classification: 20M10, 20M20, 20B40

Keywords: Semigroups, monoids, algorithms, Green's relations

1. Introduction

A *semigroup* is just a set U together with an associative binary operation. If A is a subset of a semigroup U, then we denote by $\langle A \rangle$ the smallest subsemigroup of U containing A, and refer to A as the *generators* of $\langle A \rangle$. If A and B are subsets of U, then we write $\langle A, B \rangle$ rather than $\langle A \cup B \rangle$. The question of determining the structure of the semigroup $\langle A \rangle$ given the set of generators A has a relatively long history; see the introductions of [2] or [4] for more details.

In [4] the authors present an algorithm for computing a finite semigroup; we refer to this as the *Froidure–Pin Algorithm*. More precisely, given a set A of generators belonging to a larger semigroup U, the Froidure–Pin Algorithm simultaneously enumerates the elements, produces the left and right Cayley graphs, a confluent terminating rewriting system, and a reduced word of the rewriting system for every element of $\langle A\rangle$. The Froidure–Pin Algorithm is perhaps the first algorithm for computing an arbitrary finite semigroup and is still one of the most powerful, at least for certain types of semigroup. Earlier algorithms, such as those in [8], [9], often only applied to specific semigroups, such as those of transformations or Boolean matrices. In contrast, the Froidure–Pin Algorithm can be applied to any semigroup where it is possible to multiply and test equality of the elements.

Green's relations are one of the most fundamental aspects of the structure of a semigroup, from both a theoretical and a practical perspective; see [7] or [18] for more details. The Green's structure of a semigroup underlies almost every other structural feature. As such determining the Green's relations of a finite semigroup is a necessary first step in most further algorithms. Computing Green's relations is equivalent to determining strongly connected components in the left and right Cayley graphs of a semigroup, for which there are several well-known algorithms, such as Tarjan's or Gabow's. Hence the problem of determining the Green's relations of a semigroup represented by a generating set can be reduced to the problem of finding the Cayley graphs. Thus the performance of algorithms for finding Cayley graphs has a critical influence on the majority of further algorithms for finite semigroups.

The Froidure–Pin Algorithm involves determining all of the elements of the semigroup $\langle A\rangle$ and storing them in the memory of the computer. In certain circumstances, it is possible to fully determine the structure of $\langle A\rangle$ without enumerating and storing all of its elements. One such example is the Schreier–Sims Algorithm for permutation groups; see [6], [15], [16]. In [2], based on [8], [9], [10], the Schreier–Sims Algorithm is utilised to compute any subsemigroup $\langle A\rangle$ of a regular semigroup U. Of course, this method is most efficient when trying to compute a semigroup containing relatively large, in some sense, subgroups. In other cases, it is not possible to avoid enumerating and storing all of the elements of $\langle A\rangle$. For example, if a semigroup S is $\mathscr{J}$-trivial, then the algorithms from [2] enumerate all of the elements of S by multiplying all the elements by all the generators, with the additional overheads that the approach in [2] entails. For semigroups of this type, the algorithms in [2] perform significantly worse than the Froidure–Pin Algorithm.

In this paper, we present two algorithms based on the Froidure–Pin Algorithm from [4]. The first algorithm (Algorithm 4.3 (Closure)) can be used to extend the output of the Froidure–Pin Algorithm for a given semigroup $\langle A\rangle$, to compute a supersemigroup $\langle A, B\rangle$ without recomputing $\langle A\rangle$. This algorithm might be useful for several purposes, such as for example: in combination with Tietze transformations to change generators or relations in a presentation for $\langle A\rangle$; finding small or irredundant generating sets for $\langle A\rangle$; computing the maximal subsemigroups of certain semigroups [1].

The second algorithm (Algorithm 5.6 (ConcurrentFroidurePin)) is a lock-free concurrent version of the Froidure–Pin Algorithm. Since computer processors are no longer getting faster, only more numerous, the latter provides a means for fully utilising contemporary machines for computing finite semigroups.

If S is a semigroup generated by a set A, then the time and space complexity of the Froduire–Pin Algorithm is $O(|S|\,|A|)$. Hence for example, in the case of transformation semigroups of degree n, the worst case complexity is at least $O(n^n)$. The algorithms presented here do not improve on the underlying complexity of the Froduire–Pin Algorithm but offer practical improvements on the runtime.

Both algorithms are implemented in the C++ library LIBSEMIGROUPS [11], which can be used in both GAP [12] and Python [3].

The paper is organised as follows. Some relevant background material relating to semigroups is given in Section 2. In Section 3 we describe the Froidure–Pin Algorithm and prove that it is valid. While there is much overlap between Section 3 and [4], this section is necessary to prove the validity of Algorithms 4.3 and 5.6, and because some details are omitted from [4]. Additionally, our approach is somewhat different to that of Froidure and Pin's in [4]. The first of our algorithms, for computing $\langle A, B\rangle$ given $\langle A\rangle$, is described in Section 4, and the lock-free concurrent version of the Froidure–Pin algorithm is described in Section 5. Sections 4 and 5 both contain empirical information about the performance of the implementation our algorithms in LIBSEMIGROUPS [11], and our implementation of Froidure and Pin's original algorithm in LIBSEMIGROUPS. The performance of LIBSEMIGROUPS is roughly the same as that of Pin's implementation in Semigroupe 2.01 [14].

2. Preliminaries

In this section, we recall some standard notions from the theory of semigroups; for further details see [7].

If $f : X \to Y$ is a function, then $\operatorname{dom}(f) = X$ and $\operatorname{im}(f) = f(X) = \{f(x) : x \in X\}$. If S is a semigroup and T is a subset of S, then T is a *subsemigroup* if $ab \in T$ for all $a, b \in T$. A semigroup S is a *monoid* if it has an identity element, i.e. an element $1_S \in S$ such that $1_S s = s1_S = s$ for all $s \in S$.

If A is a subset of a semigroup S, then we denote by $\langle A\rangle$ the smallest subsemigroup of S containing A, and refer to A as the *generators* or *generating set* of $\langle A\rangle$. If A is a generating set for a semigroup S, then the *right Cayley graph* of a semigroup S with respect to A is the digraph with vertex set S and an edge from s to sa for all $s \in S$ and $a \in A$. The *left Cayley graph* of S is defined analogously.

A *congruence* ρ on a semigroup S is an equivalence relation on S which is invariant under the multiplication of S. More precisely, if $(x, y) \in \rho$, then (xs, ys),

$(sx, sy) \in \rho$ for all $s \in S$. A *homomorphism* from a semigroup S to a semigroup T is just a function $f : S \to T$ such that $f(xy) = f(x)f(y)$ for all $x, y \in S$. If ρ is a congruence on a semigroup S, then the *quotient* S/ρ of S by ρ is the semigroup consisting of the equivalence classes $\{s/\rho : s \in S\}$ of ρ with the operation

$$x/\rho y/\rho = (xy)/\rho$$

for all $x, y \in S$. If $f : S \to T$ is a homomorphism of semigroups, then the *kernel* of f is

$$\ker(f) = \{(x, y) \in S \times S : f(x) = f(y)\},$$

and this is a congruence. There is a natural isomorphism between $S/\ker(f)$ and $\mathrm{im}(f)$ which is a subsemigroup of T.

An *alphabet* is just a finite set A whose elements we refer to as *letters*. A *word* is just a finite sequence $w = (a_1, \ldots, a_n)$ of letters $a_1, \ldots, a_n$ in an alphabet A. For convenience, we will write $a_1 \ldots a_n$ instead of $(a_1, \ldots, a_n)$. The *length* of a word $w = a_1 \ldots a_n \in A^*$ is n and is denoted $|w|$. A non-empty *subword* of a word $a_1 \ldots a_n$ is a word of the form $a_i a_{i+1} \ldots a_j$ where $1 \leq i \leq j \leq n$.

The *free semigroup* over the alphabet A is the set of all words over A with operation the concatenation of words; we denote the free semigroup on A by A^+. The free semigroup A^+ has the property that every function $f : A \to S$, where S is a semigroup, can be uniquely extended to a homomorphism $\nu : A^+ \to S$ defined by $\nu(a_1 \ldots a_n) = f(a_1)f(a_2) \ldots f(a_n)$ for all $a_1 \ldots a_n \in A^+$. We denote the monoid obtained from A^+ by adjoining the empty word by A^*; which is called the *free monoid*.

Suppose that $<$ is a well-ordering of A. Then $<$ induces a well-ordering on A^*, also denoted by $<$, where $u < v$ if $|u| < |v|$ or there exist $p, u', v' \in A^*$ and $a, b \in A$, $a < b$, such that $u = pau'$ and $v = pbv'$. We refer to $<$ as the *short-lex* order on A^*. Let S be a semigroup and let $A \subseteq S$. Then there exists a unique homomorphism $\nu : A^+ \to S$ that extends the inclusion function from A into S. The word $u \in A^+$ such that

$$u = \min\{v \in A^+ : \nu(u) = \nu(v)\},$$

with respect to the short-lex order on A^+, is referred to as a *reduced word* for S. For every word $w \in A^+$, there is a unique reduced word for S, which we denote by $\overline{w}$.

We require the following results relating to words and the short-lex order $<$.

Proposition 2.1 (cf. Proposition 2.1 in [4]). *Let A be any alphabet, let $u, v, x, y \in A^*$, and let $a, b \in A$. Then the following hold:*

(a) *if $u < v$, then $au < av$ and $ua < va$;*

(b) *if $ua \leq vb$, then $u \leq v$;*

(c) *if $u \leq v$, then $xuy \leq xvy$.*

Proposition 2.2 (cf. Proposition 2.1 in [17]). *If S is a semigroup, $A \subseteq S$, and $w \in A^+$ is reduced for S, then every non-empty subword of w is reduced.*

Throughout this paper, we refer to the semigroup U as the *universe*, and we let S be a subsemigroup of U represented by a finite set A of generators. The algorithms described herein require that we can:

- compute the product of two elements in U; and
- test the equality of elements in U.

In each of the main algorithms presented in this paper, rather than obtaining the actual elements of the subsemigroup S we obtain reduced words representing the elements of S. This is solely for the simplicity of the exposition in this paper, and is equivalent to using the actual elements of S by the following proposition.

Proposition 2.3 (cf. Proposition 2.3 in [4]). *If S is a semigroup, $A \subseteq S$, $v : A^+ \to S$ is the unique homomorphism extending the inclusion of A into S, then the set $R = \{\overline{w} : w \in A^*\}$ with the operation defined by $u \cdot v = \overline{uv}$ is a semigroup isomorphic to S.*

3. The Froidure–Pin Algorithm

In this section we describe a version of the Froidure–Pin Algorithm from [4] and prove that it is valid.

Throughout this section, let U be any semigroup, let S be a subsemigroup of U generated by $A \subseteq U$ where $1_U \in A$, and let $v : A^+ \to S$ be the unique homomorphism extending the inclusion function of A into S. Since $v(a) = a$ for all $a \in A$, we can compute $v(s)$ for any $s \in A^+$ by computing products of elements in S.

We require functions $f, l : A^+ \to A$ and $p, s : A^+ \to A^*$ defined as follows:

- if $w \in A^+$ and $w = au$ for some $a \in A$ and $u \in A^*$, then $f(w) = a$ and $s(w) = u$, i.e. $f(w)$ is the first letter of w and $s(w)$ is the suffix of w with length $|w| - 1$;
- if $w \in A^+$ and $w = vb$ for some $b \in A$ and $v \in A^*$, then $l(w) = b$ and $p(w) = v$, i.e. $l(w)$ is the last letter of w and $p(w)$ is the prefix of w with length $|w| - 1$.

Note that if $w \in A$, then both $p(w)$ and $s(w)$ equal the empty word.

Next, we give formal definition of the input and output of the version of the Froidure–Pin Algorithm described in this section. A less formal discussion of the definition can be found below.

Definition 3.1. The input of our version of the Froidure–Pin Algorithm is a *snapshot of* S which is a tuple (A, Y, K, B, ϕ) where:

(a) $A = \{a_1, \dots, a_r\}$ is a finite collection of generators for S where $a_1 < \dots < a_r$ for some $r \in \mathbb{N}$;

(b) $A \subseteq Y = \{y_1, \dots, y_N\}$ is a collection of reduced words for S, $y_1 < y_2 < \dots < y_N$, and if $x \in A^+$ is reduced, then either $x \in Y$ or $x > y_N$, for some $N \in \mathbb{N}$;

(c) $K \in \mathbb{N}$ and $1 \leq K \leq |Y| + 1$;

(d) either $B = \emptyset$ or $B = \{a_1, \dots, a_s\}$ for some $1 \leq s \leq r$; and

(e) if $K \leq |S|$, then ϕ is a function from

$$(A \times \{y_1, \dots, y_L\}) \cup (\{y_1, \dots, y_{K-1}\} \times A) \cup (\{y_K\} \times B)$$

to Y such that $v(\phi(u, v)) = v(uv)$ for all $u, v \in \mathrm{dom}(\phi)$ and either $L = K - 1$ or $L < K - 1$ and L is the largest value such that $|y_L| < |y_{K-1}|$.

Note that by part (e) of Definition 3.1, $\phi(u, v) \leq uv$ for all $u, v \in \mathrm{dom}(\phi)$ since $\phi(u, v) \in Y$ is reduced and both uv and $\phi(u, v)$ represent the same element of S. If in parts (c) and (e) of Definition 3.1, $L = K - 1$ and $B = A$, respectively, then the snapshot (A, Y, K, B, ϕ) could be written as the snapshot $(A, Y, K + 1, \emptyset, \phi)$ instead. We allow both formulations so that Algorithm 3.1 (Update) is more straightforward.

Roughly speaking, in Definition 3.1, part (b) say that Y consists of all the reduced words for S which are less than or equal to y_N. The set Y consists of those elements of S that have been found so far in our enumeration. The K in part (c) is the index of the next element in Y that will be multiplied on the right by some of the generators A. More precisely, y_K will be multiplied by the least element in $A \backslash B$ where B is given in part (d). The function ϕ in part (e) represents a subgraph of the right and left Cayley graphs of S with respect to A. More specifically, if $B \neq A$, then K is the index of the first element in Y where not all of the right multiples $y_K a$ where $a \in A$ are yet known. Similarly, L is the index of the last element in Y where all of the left multiples of y_L are known. Condition (e) indicates that if any right multiples of a reduced word w for S are known, then the left multiples of all reduced words of length at most $|w| - 1$ are also known. The condition also allows the left multiples of all reduced words of length $|w|$ to be known.

The output of our version of the Froidure–Pin Algorithm is another snapshot of the above type where: the output value of K is at least the input value; the output value of Y contains the input value as a subset; and the output function ϕ is an extension of the input function. In this way, we say that the output snapshot *extends* the input data structure. In other words, if $K \leq |S|$ or $Y \neq S$, then the output snapshot contains more edges in the right Cayley graph than the input snapshot, and may contain more elements of S, and more edges in the left Cayley graph.

The parameters $|Y|$ and K quantify the state of the Froidure–Pin Algorithm, in the sense that the minimum values are $|Y| = |A|$ and $K = 1$, and the snapshot is *complete* when $K = |S| + 1 = |Y| + 1$, which means that $Y = S$ and all of the edeges of the left and right Cayley graphs of S are known. We say that a snapshot of S is *incomplete* if it is not complete. If desirable the Froidure–Pin Algorithm can be halted before the output snapshot is complete (when $K \leq |S|$), and subsequently continued and halted, any number of times. Such an approach might be desirable, for example, when testing if $u \in U$ belongs to S, we need only run the Froidure–Pin Algorithm until u is found, and not until $K = |S| + 1$ unless $u \notin S$.

The minimal snapshot required by the Froidure–Pin Algorithm for $S = \langle A \rangle$ is:

$$(A, A, 1, \emptyset, \emptyset); \tag{3.2}$$

where the only known elements are the generators, and no left nor right multiples of any elements are known.

If (A, Y, K, B, ϕ) is a snapshot of the semigroup S, then $|Y|$ is referred to as its *size*, its *elements* are the elements of Y, and $x \in S$ *belongs* to the snapshot if the reduced word corresponding to x belongs to Y, i.e. $x = \nu(w)$ for some $w \in Y$.

Algorithm 3.1 (Update) is an essential step in the Algorithm 3.2 (Froidure–Pin) and we will reuse (Update) in Algorithm 4.3 (Closure). Roughly speaking, Update adds the first unknown right multiple to an incomplete snapshot. In Update a multiplication is only performed if it is not possible to deduce the value from existing products in the input snapshot. Deducing the value of a right multiple is a con-

Algorithm 3.1: Update: adds the first unknown right multiple to an incomplete snapshot

```
Input: An incomplete snapshot $(A, Y, K, B, \phi)$ for a semigroup $S$ where $B \neq A$
Output: A snapshot extending $(A, Y, K, B, \phi)$ that contains $y_K a$ where $a = \min(A \setminus B)$
1 if $\phi(s(y_K), a) < s(y_K)a$ then                 $s(y_K)a$ is not reduced, and so $y_K a$ is not reduced
2     $\phi(s(y_K), a) = y_i$ for some $y_i \in Y$
3     $\phi(y_K, a) := \phi(\phi(f(y_K), p(y_i)), l(y_i))$
4 else if $\nu(y_K a) = \nu(y_i)$ for some $i \leq |Y|$ then          $s(y_K)a$ is reduced but $y_K a$ is not
5     $\phi(y_K, a) := y_i$
6 else                                                  $y_K a$ is reduced
7     $y_{|Y|+1} := y_K a$ and $Y \leftarrow Y \cup \{y_{|Y|+1}\}$
8     $\phi(y_K, a) := y_{|Y|+1}$
9 return $(A, Y, K, B \cup \{a\}, \phi)$
```

stant time operation, whereas performing the multiplication almost always has higher complexity. For example, matrix multiplication is at least quadratic in the number of rows. This is the part of the Froidure–Pin Algorithm that is responsible for its relatively good performance.

Recall that since $\nu(a) = a$ for all $a \in A$ and ν is a homomorphism, the value of $\nu(w) \in S$ can be determined for any $w \in A^+$ by computing products in S. In practice, we perform the single multiplication $\nu(y_K)\nu(a) = \nu(y_K a)$.

Lemma 3.3. *If (A, Y, K, B, ϕ) is a snapshot of a semigroup S, $B \neq A$, and a is the least generator in $A \backslash B$, then Algorithm* 3.1 (Update) *returns a snapshot of S containing $y_K a$.*

Proof. There are three cases to consider:

(a) $s(y_K)a$ is not reduced

(b) $\nu(y_K a) = \nu(y_i)$ for some $i \leq N$

(c) neither (a) nor (b) holds.

If (a) or (b) holds, then the only component of the snapshot that is modified is ϕ, and so we must verify that ϕ is well-defined, and satisfies Definition 3.1(e). In the case when (c) holds, we also need to show that Definition 3.1(b) holds.

(a). In this case, $\phi(s(y_K), a) \in Y$ is defined because $s(y_K) \in Y$ and $s(y_K) < y_K$. Hence there exists $i \leq N$ such that $\phi(s(y_K), a) = y_i$. Since $p(y_i)l(y_i) = y_i < s(y_K)a$, by Proposition 2.1(b), $p(y_i) \leq s(y_K)$ and $s(y_K) < y_K$ since $|s(y_K)| < |y_K|$. Hence $\phi(f(y_K), p(y_i))$ is defined and

$$\phi(f(y_K), p(y_i)) \leq f(y_K)p(y_i) < f(y_K)s(y_K) = y_K.$$

By the definition of ϕ and since ν is a homomorphism,

$$\begin{aligned}\nu(y_K a) &= \nu(f(y_K)s(y_K)a) = \nu(f(y_K))\nu(s(y_K)a) = \nu(f(y_K))\nu(\phi(s(y_K), a)) \\ &= \nu(f(y_K))\nu(y_i) = \nu(f(y_K)y_i) = \nu(f(y_K)p(y_i))\nu(l(y_i)) \\ &= \nu(\phi(f(y_K), p(y_i)))\nu(l(y_i)) = \nu(\phi(f(y_K), p(y_i))l(y_i)) \\ &= \nu(\phi(\phi(f(y_K), p(y_i)), l(y_i))).\end{aligned}$$

Hence if $\phi(y_K, a) := \phi(\phi(f(y_K), p(y_i)), l(y_i))$, then ϕ continues to satisfy Definition 3.1(e).

(b). By the assumption of this case, $\nu(\phi(y_K, a)) = \nu(y_i) = \nu(y_K a)$, and Definition 3.1(e) continues to hold.

(c). In this case, $y_K a$ is reduced, and $y_K a \notin Y$. Hence $y_K a > \max Y$ by Definition 3.1(b), and so $Y \cup \{y_{|Y|+1}\}$ where $y_{|Y|+1} = y_K a$ satisfies the first part

Algorithm 3.2: FroidurePin: enumerate at least $\min\{M, |S|\}$ elements of a semigroup S.

```
Input: A snapshot $(A, Y, K, \varnothing, \phi)$ for a semigroup $S$ and a limit $M \in \mathbb{N}$
Output: A snapshot of $S$ which extends $(A, Y, K, \varnothing, \phi)$ and with size at least $\min\{M, |S|\}$
1  while $K \le |Y|$ and $|Y| < M$ do
2      $c := |y_K|$
3      while $K \le |Y|$ and $|y_K| = c$ and $|Y| < M$ do
4          $B := \varnothing$
5          while $B \setminus A \neq \varnothing$ do          loop over the generators $A$ in (short-lex) order
6              $(A, Y, K, B, \phi) \leftarrow$ Update$(A, Y, K, B, \phi)$
7          $K \leftarrow K + 1$
8      if $K > |Y|$ or $|y_K| > c$ then
9          $L = \max\{i \in \mathbb{N} : |y_i| < c\}$
10         for $i \in \{L+1, \ldots, K-1\}$ do          indices of words of length one less than $y_K$
11             for $a \in A$ do                          extend $\phi$ so that $A \times \{y_i\} \subseteq \mathrm{dom}(\phi)$
12                 $\phi(a, y_i) := \phi(\phi(a, p(y_i)), l(y_i))$
13 return $(A, Y, K, \varnothing, \phi)$
```

of Definition 3.1(b). If $w \in A^+$ is reduced and $w < y_K a = y_{|Y|+1}$, then $w = p(w)l(w) < y_K a$ and so either $p(w) < y_K$ or $l(w) < a$ by Proposition 2.1. In both cases, $\big(p(w), l(w)\big) \in \mathrm{dom}(\phi)$, and so $w = \phi\big(p(w), l(w)\big) \in Y$ and so Definition 3.1(b) holds.

By the assumption of this case $v(y_K a) \neq v(y_i)$ for any $y_i \in Y$, and so, in particular, $\overline{y_K a} \notin Y$. On the other hand, $\overline{y_K a} \le y_K a$ and $\overline{y_K a} < y_K a$ implies $\overline{y_K a} \in Y$ from the previous paragraph. Hence $y_K a = \overline{y_K a}$ is reduced. Finally, $v\big(\phi(y_K, a)\big) = v(y_{|Y|+1}) = v(y_K a)$ by definition and so Definition 3.1(e) holds. □

Next we state the Froidure–Pin Algorithm, a proof that the algorithm is valid follows from Lemmas 3.3 and 3.4.

Note that both the input, and output, of Algorithm 3.2 (FroidurePin) has fourth component equal to $\emptyset$, and as such it would appear to be unnecessary. However, it is used in the definition of a snapshot so that we can succinctly describe the output of Closure. FroidurePin could be modified to return a snapshot where the fourth component was not empty, but for the sake of relative simplicity we opted not to allow this.

Lemma 3.4. *If $(A, Y, K, \emptyset, \phi)$ is a snapshot of a semigroup S and $M \in \mathbb{N}$, then Algorithm 3.2 (FroidurePin) returns a snapshot for S with at least $\min\{M, |S|\}$ elements.*

Proof. We may suppose that $K \le |Y|$ and $|Y| < M$, since otherwise FroidurePin does nothing.

By Lemma 3.3, after applying Update to the input snapshot and every $a \in A$, in line 6, (A, Y, K, A, ϕ) is a snapshot of S containing $y_K A$. At this point, if $K \le |Y|$

and $|y_{K+1}| = |y_K|$, then we continue the while-loop starting in line 3. When we arrive in line 8 one of the following holds: $K > |Y|$, $|Y| \geq M$, or $|y_K| > c = |y_{K-1}|$.

If the condition in line 8 is not satisfied, then the condition in line 3 is false because $|Y| \geq M$. Hence the condition in line 3 is also false, and so (A, Y, K, A, ϕ) is returned in this case. Since Algorithmn 3.1 (Update) returns a snapshot, (A, Y, K, A, ϕ) could only fail to be a snapshot because the value of K was increased. In other words, the tuple (A, Y, K, A, ϕ) satisfies Definition 3.1 (a) to (d) and $v(\phi(u, v)) = v(uv)$ for all $(u, v) \in \operatorname{dom}(\phi)$. Since the condition in line 8 is not satisfied, $|y_K| = |y_{K-1}| = c$, and so (A, Y, K, A, ϕ) continues to satisfy Definition 3.1(e), which is hence a snapshot of S, as required.

If the condition in line 8 is satisfied, then either $K > |Y|$; or $K \leq |Y|$ and $|y_K| > |y_{K-1}| = c$. In either case, the tuple (A, Y, K, B, ϕ) satisfies Definition 3.1(a) to (d), but may fail to satisfy part (e), since ϕ may not be defined on $A \times \{y_{L+1}, \ldots, y_{K-1}\}$, where $L \in \mathbb{N}$ is the maximum value such that $|y_L| < |y_{K-1}|$ (as defined in line 9). The only component of (A, Y, K, B, ϕ) that is modified within the if-clause is ϕ. Hence by the end of the if-clause (A, Y, K, B, ϕ) is a snapshot for S, provided that $\phi(a, y_i)$ is well-defined for all $i \in \{L+1, \ldots, K-1\}$ and $v(\phi(a, y_i)) = v(ay_i)$ for all $a \in A$.

Let $i \in \{L+1, \ldots, K-1\}$. Since $|p(y_i)| = |y_i| - 1$ and $|y_i| = |y_{K-1}|$, $\phi(a, p(y_i))$ is defined and $|\phi(a, p(y_i))| \leq |ap(y_i)| = |y_{K-1}|$, because $\phi(a, p(y_i)) \leq ap(y_i)$. Since y_{K-1} is the largest reduced word of length $|y_{K-1}|$, there exists $j < K$ such that $\phi(a, p(y_i)) = y_j$. By definition, $(y_j, x) \in \operatorname{dom}(\phi)$ for all $x \in A$. In particular, $\phi(y_j, l(y_i)) = \phi(\phi(a, p(y_i)), l(y_i))$ is defined, and so the assignment in line 12 is valid.

Let $a \in A$ and $i \in \{L+1, \ldots, K-1\}$ be arbitrary. Then

$$\begin{aligned}
v(\phi(a, y_i)) &= v(\phi(\phi(a, p(y_i)), l(y_i))) = v(\phi(a, p(y_i))l(y_i)) \\
&= v(\phi(a, p(y_i)))v(l(y_i)) \\
&= v(ap(y_i))v(l(y_i)) = v(ap(y_i)l(y_i)) \\
&= v(ay_i),
\end{aligned}$$

as required.

Finally, the algorithm halts if $|K| > |Y|$ in which case the snapshot contains $|S|$ elements, or if $|Y| \geq M$. In either case, $|Y| \geq \min\{M, |S|\}$, as required. □

Corollary 3.5. *If $(A, Y, K, \emptyset, \phi)$ is a snapshot of a semigroup S and $M \in \mathbb{N}$ is such that $M \geq |S|$, then Algorithm 3.2 (FroidurePin) returns $(A, R, |S| + 1, \emptyset, \phi)$ where R is the set of all reduced words for elements of S and* $\operatorname{dom}(\phi) = (A \times R) \cup (R \times A)$.

Proof. Suppose that the snapshot returned by FroidurePin is $(A, Y, |S| + 1, \emptyset, \phi)$.

Under the assumptions of the statement, the last iteration of while-loop starting in line 3 terminates when $K = |Y| + 1$. Hence the condition of the if-clause in line 8 is satisfied, and so $\mathrm{dom}(\phi) = (A \times Y) \cup (Y \times A)$.

Assume that there exists a reduced word $w \in A^+$ such that $w \notin Y$. Then we may assume without loss of generality that w is the minimum such reduced word. Hence $p(w)$ is a reduced word and $p(w) < w$ and so $p(w) \in Y$. But then $\phi(p(w), l(w))$ is defined, and so $w = \phi(p(w), l(w)) \in Y$, a contradiction. Hence $Y = R$. □

4. The closure of a semigroup and some elements

In this section we give the first of the two new algorithms in this paper. Given a snapshot of a semigroup $S = \langle A \rangle \leq U$ and some additional generators $X \subseteq U$, this algorithm returns a snapshot for $T = \langle A, X \rangle$. If $v : (A \cup X)^+ \to T$ is the unique homomorphism extending the inclusion map from $A \cup X$ into T, then v restricted to A^+ is the unique homomorphism extending the inclusion map from A into S. Hence we only require the notation $v : (A \cup X)^+ \to T$.

The purpose of Algorithm 4.3 (Closure) is to avoid multiplying elements in the existing snapshot for S, by the generators A, in the creation of the snapshot for T wherever such products are already known. The principal complication is that the introduction of new generators can change the reduced word representing a given element s of S. The new generating set $A \cup X$ may allow s to be written as a shorter word than was previously possible with A. Suppose that $\{y_1, \dots, y_{K_S}\}$ is the set of those elements in the original snapshot for S whose right multiples by the old generating set were known. Closure terminates when the right multiplies by all the generators, old and new, of every element in $\{y_1, \dots, y_{K_S}\}$ are known. In short Closure is a version of FroidurePin where each of $K_S|A|$ products can be found in constant time. This works particularly well when the addition of new generators does not increase the size of the semigroups substantially, or when the complexity of multiplication is very high.

We prove that Closure is valid in Lemma 4.1.

Lemma 4.1. *If $(A, Y, K_S, \emptyset, \phi_S)$ is a snapshot of a semigroup $S \leq U$ and $X \subseteq U$, then Algorithm 4.3 (Closure) returns a snapshot of $T = \langle X, A \rangle$ that contains $\{y_1, \dots, y_{K_S}\}X$, and hence also contains Y.*

Proof. At the start of Closure, $(A \cup X, Z, K_T, B, \phi_T)$ is initialised as $(A \cup X, A \cup X, 1, \emptyset, \emptyset)$, which is the minimal snapshot of T by (3.2).

Additionally, $\lambda : A \to Z$ satisfies the following two conditions:

Algorithm 4.3: Closure: add additional generators to a snapshot for a semigroup

```
**Input**: A snapshot $(A, Y, K_S, \varnothing, \phi_S)$ for a semigroup $S \leq U$, and a collection of elements $X \subseteq U \setminus Y$
**Output**: A snapshot $(A \cup X, Z, K_T, \varnothing, \phi_T)$ for $T = \langle S, X \rangle$ that contains $\{y_1, \ldots, y_{K_S}\}X$.
Define $A = \{z_1 = y_1, \ldots, z_m = y_m\}$ and $X = \{z_{m+1}, \ldots, z_{m+n}\}$, $z_1 < z_2 < \cdots < z_{m+n}$
$Z := A \cup X$, $K_T := 1$, $\phi_T = \varnothing$          initialise the snapshot for $T$
Define $\lambda : A \longrightarrow Z$ to be the identity function on $A$          $\lambda$ maps a reduced word for $S$ to a reduced word for $T$ representing the same element, at this point $A$ is the set $\{y \in Y : \exists z \in Z,\ \nu(y) = \nu(z)\}$
while $\mathrm{dom}(\lambda) \neq Y$ do          there are reduced words in $Y$ not yet in the snapshot for $T$
    $c := |z_{K_T}|$
    while $\mathrm{dom}(\lambda) \neq Y$ and $|z_{K_T}| = c$ do
        $B := \varnothing$
        if $\exists y_i \in Y,\ \nu(z_{K_T}) = \nu(y_i)$ and $i < K_S$ then
            for $a \in A$ do          loop over the old generators in (short-lex) order
                if $\phi_S(y_i, a) \in \mathrm{dom}(\lambda)$ then
                    $\phi_T(z_{K_T}, a) := \lambda(\phi_S(y_i, a))$
                else
                    $\phi_T(z_{K_T}, a) := z_{K_T}a$ and $\lambda(\phi_S(y_i, a)) := z_{K_T}a$
                    $z_{|Z|+1} := z_K a$ and $Z \leftarrow Z \cup \{z_{K_T}a\}$
            $B \leftarrow A$
        while $(A \cup X) \setminus B \neq \varnothing$ do
            $a := \min\{(A \cup X) \setminus B\}$
            $(A \cup X, Z, K_T, B, \phi_T) \leftarrow$ Update$(A \cup X, Z, K_T, B, \phi_T)$
            if $\phi_T(z_{K_T}, a) = z_{K_T}a$ and $\nu(z_{K_T}a) = \nu(y_i)$ for some $y_i \in Y$ then
                $\lambda(y_i) := z_{K_T}a$

        $K_T \leftarrow K_T + 1$
    if $K_T > |Z|$ or $|z_{K_T}| > c$ then
        $L = \max\{i \in \mathbb{N} : |z_i| < c\}$
        for $i \in \{L + 1, \ldots, K_T - 1\}$ do
            for $a \in A \cup X$ do
                $\phi_T(a, z_i) := \phi_T(\phi_T(a, p(z_i)), l(z_i))$
```

(a) $\mathrm{dom}(\lambda) = \{y \in Y : \exists z \in Z, \nu(y) = \nu(z)\}$; and

(b) $\nu(\lambda(y)) = \nu(y)$ for all $y \in \mathrm{dom}(\lambda)$.

If $Y = \mathrm{dom}(\lambda)$, then the minimal snapshot of T is returned, and there nothing to prove. So, suppose that $Y \neq \mathrm{dom}(\lambda)$. We proved in Lemma 3.3, that Algorithm 3.1 (Update) returns a snapshot of T, given a snapshot of T. The data structure $(A \cup X, Z, K_T, B, \phi_T)$ is otherwise only modified within the while-loop starting on line 6, in the case that there exists $y_i \in Y$ such that $\nu(z_{K_T}) = \nu(y_i)$ and $i < K_S$. Hence it suffices to verify that after performing the steps in the if-clause starting in line 8 the tuple $(A \cup X, Z, K_T, B, \phi_T)$ is still a snapshot of T. In order to do this, we use the properties of λ given above. Hence we must also check that λ continues to satisfy conditions (a) and (b) whenever it is modified (i.e. in lines 13 and 20).

Suppose that λ satisfies conditions (a) and (b) above, and that there exists $y_i \in Y$ such that $v(z_{K_T}) = v(y_i)$ and $i < K_S$ and that $a \in A$. Since $i < K_S$, $\phi_S(y_i, a)$ is defined for all $a \in A$.

If $\phi_S(y_i, a) \in \mathrm{dom}(\lambda)$, then in line 11 we define $\phi_T(z_{K_T}, a) = \lambda(\phi_S(y_i, a))$. In this case, $(A \cup X, Z, K_T, B, \phi_T)$ satisfies Definition 3.1(a) to (d) trivially and

$$v(\phi_T(z_{K_T}, a)) = v(\lambda(\phi_S(y_i, a))) = v(\phi_S(y_i, a)) = v(y_i a) = v(z_{K_T} a).$$

and so Definition 3.1(e) holds.

If $\phi_S(y_i, a) \notin \mathrm{dom}(\lambda)$, then we define $\phi_T(z_{K_T}, a) = z_{K_T} a$. Since $v(\phi_T(z_{K_T}, a)) = v(z_{K_T} a)$ by definition, it suffices to show that $z_{K_T} a$ is a reduced word for T. Suppose, seeking a contradiction, that $u \in (A \cup X)^+$ is reduced, $v(u) = v(y_i a)$, and $u < z_{K_T} a$. Then either $p(u) < z_{K_T}$ or $p(u) = z_{K_T}$ and $l(u) < a$ by Propostion 2.1(b). In either case, $\phi_T(p(u), l(u)) = u$ is defined and so $u \in Z$. Thus $v(\phi_S(y_i, a)) = v(y_i a) = v(u)$ and so $\phi_S(y_i, a) \in \mathrm{dom}(\lambda)$ by (a), which contradicts the assumption of this case. Hence $z_{K_T} a$ is reduced. We must verify conditions (a) and (b) on λ after defining $\lambda(\phi_S(y_i, a)) = z_{K_T} a \in Z$ in line 13. Condition (a) holds, since we extended both Z and $\mathrm{dom}(\lambda)$ by a single value. Since

$$v(\lambda(\phi_S(y_i, a))) = v(z_{K_T} a) = v(z_{K_T}) v(a) = v(y_i) v(a) = v(y_i a) = v(\phi_S(y_i, a))$$

condition (b) also holds.

The only other part of the algorithm where λ is modified is line 20. Suppose that $a = \min((A \cup X) \backslash B)$ as defined in line 17. If $\phi_T(z_{K_T}, a) = z_K a$ and $v(z_{K_T}) = v(y_i)$ for some $y_i \in Y$, then $z_K a \in Z$ is reduced. In this case, we define $\lambda(y_i) = z_{K_T} a$. Conditions (a) and (b) hold by the above argument.

We have shown that within the while-loop starting on line 6, the tuple $(A \cup X, Z, K_T, B, \phi_T)$ satisfies Definition 3.1(a) to (d). Additionally, we have shown that conditions (a) and (b) hold for λ.

The remainder of the proof follows by the argument given in the proof of Lemma 3.4. □

4.1. Experimental results. In this section we compare the performance of Algorithm 3.2 (FroidurePin) and Algorithm 4.3 (Closure). Further details about the computations performed, including code that can be used to reproduce the data, in this section can be found in [13].

We note that in the implementation of Closure in LIBSEMIGROUPS [11], the snapshot of $S = \langle A \rangle$ is modified in-place to produce the data structure for $T = \langle A, X \rangle$, and that none of the elements of S need to be copied or moved in memory during Closure.

Figure 1, 2, 3, and 4 we plot a comparison of FroidurePin and Closure for 300 examples of randomly generated semigroups. For a particular choice of A and X, three computations were run:

(1) a snapshot for $\langle A\rangle$ was enumerated using FroidurePin until it contained $|\langle A\rangle|$ elements; we denote the time taken for this step by t_1.

(2) Closure was performed on the snapshot obtained from (1) and the generators X. We denote the size of the snapshot obtained in this way by M and the time taken for this step by t_2.

(3) A snapshot for $\langle A, X\rangle$ was enumerated using FroidurePin until it contained M (from (2)) elements. We denote the time taken for this step by t_3.

A point on the x-axis of any graph in this section correspond to a single choice of A and X. The points on the x-axes are sorted in increasing order according to the $M/\langle A\rangle$. In other words, $M/\langle A\rangle$ indicates the proportion of new elements generated in Closure. The y-axis corresponds to time divided by t_3 from (3). The triangles correspond to the mean value of t_2/t_3 taken over 3 separate runs, and the crosses similarly correspond to $(t_1 + t_2)/t_3$.

The values of A and X in Figure 1 were chosen as follows:

- the collection A was chosen to have size between 2 and 30 elements (uniformly at random) and to consist of 6×6 Boolean matrices, which were chosen uniformly at random from the space of all such Boolean matrices.

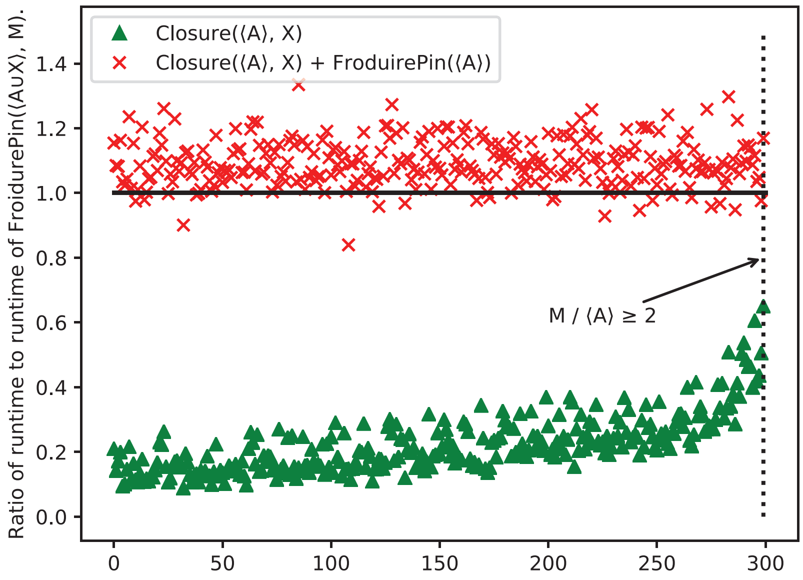

Figure 1. A and X consist of 6×6 Boolean matrices, $|A| \in \{2, \ldots, 30\}$, $|X| = 1$, M is the size of the snapshot for $\langle A, X\rangle$ returned by Algorithm 4.3 (Closure).

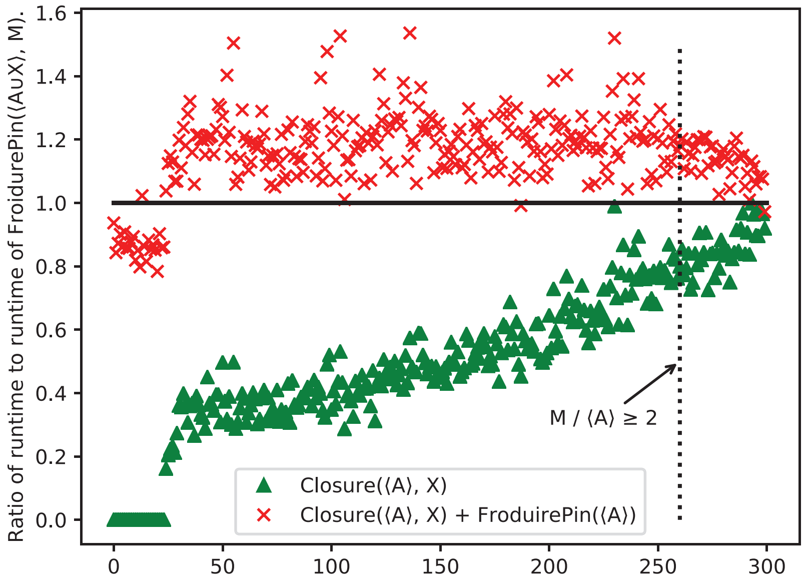

Figure 2. A and X consist of transformations of degree 7, $|A| \in \{2, \ldots, 8\}$, $|X| = 1$, M is the size of the snapshot for $\langle A, X\rangle$ returned by Algorithm 4.3 (Closure).

- the collection X was taken to consist of a single 6×6 Boolean matrices, again chosen uniformly at random.

The values of A and X in Figure 2 were chosen in a similar fashion from the space of all transformations of degree 7, where $|A| \in \{2, \ldots, 8\}$ and X was chosen so that $|X| = 1$. In Figures 3 and 4, A and X were chosen in the same way as collections of Boolean matrices, and transformations, with $|A|, |X| \in \{2, \ldots, 30\}$, and $|A|, |X| \in \{2, \ldots, 8\}$, respectively. The semigroups in Figures 1, 2, 3, and 4 range in size from 1767 to 681973, with time to run FroidurePin and Closure roughly in the range 10 to 1500 milliseconds, particular values chosen for A and X can be found in the table below. We rejected those choices of A and X where the time t_1 to enumerate $\langle A\rangle$ using FroidurePin was less than 10 milliseconds, to reduce the influence of random noise in the resulting data.

The range of values for $|A|$, the dimensions of these matrices, and degrees of these transformations were chosen so that the resulting semigroups were not too large or small, and it was possible to run FroidurePin on these semigroups in a reasonable amount of time.

It is not surprising to observe that when $M/\langle A\rangle$ is large that there is little benefit, or even a penalty, for running Closure. This due to the fact that the run time t_1 of FroidurePin on $\langle A\rangle$ is small in comparison to t_2 and t_3. In this case, in Closure

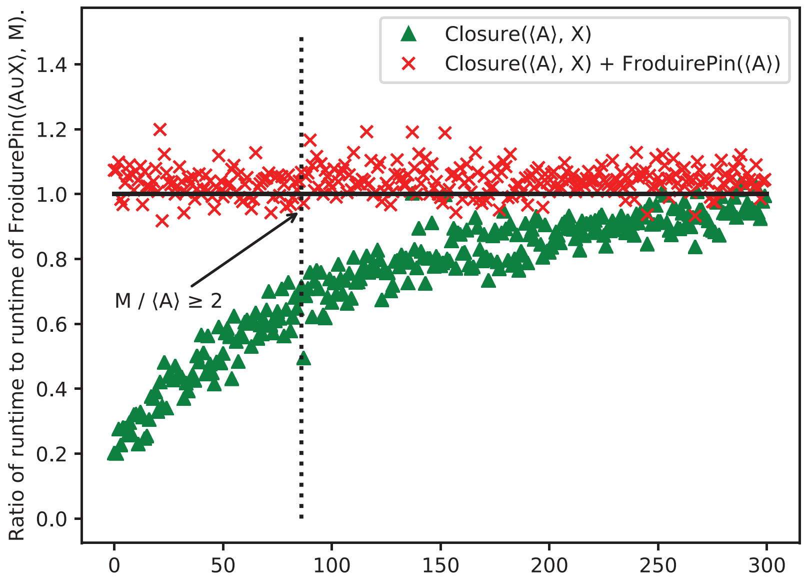

Figure 3. A and X consist of 6×6 Boolean matrices, $|A|, |X| \in \{2, \ldots, 30\}$, M is the size of the snapshot for $\langle A, X \rangle$ returned by Algorithm 4.3 (Closure).

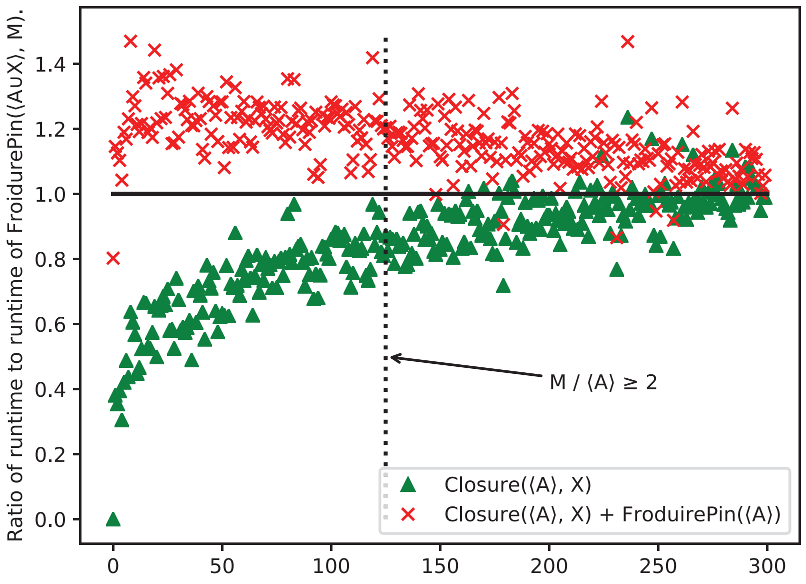

Figure 4. A and X consist of transformations of degree 7, $|A|, |X| \in \{2, \ldots, 8\}$, M is the size of the snapshot for $\langle A, X \rangle$ returned by Algorithm 4.3 (Closure).

applied to $\langle A \rangle$ and X, only a relatively small proportion of multiplications can be deduced, and so Closure is essentially doing the same computation as FroidurePin, but with an additional overhead. It does appear from these examples, that Closure is beneficial when $M/\langle A \rangle \leq 2$.

5. A lock-free concurrent version of the Froidure–Pin Algorithm

In this section we describe a version of the Froidure–Pin Algorithm that can be applied concurrently in multiple distinct processes. Roughly speaking, the idea is to split a snapshot of a semigroup S into fragments where an algorithm similar to Algorithm 3.2 (FroidurePin) can be applied to each fragment independently. Similar to the Froidure–Pin Algorithm the value of a right multiple of an element with a generator is deduced whenever possible. If such a right multiple cannot be deduced, then it is stored in a queue until all fragments are synchronised. The original Froidure–Pin Algorithm already offers a natural point for this synchronisation to occur, when the right multiples of all words of a given length have been determined. In the concurrent algorithm, the fragments are synchronised at exactly this point. In this way, the concurrent algorithm has two phases: the first when right multiples are determined, and the second when the fragments are synchronised. Some of the deductions of products that are made by the original Froidure–Pin Algorithm cannot be made in our concurrent version. This happens if the information required to make a deduction is, or will be, contained in a different fragment. In order to avoid locking, a fragment does not have access to any information in other fragments during the first phase. In the second phase, each fragment F can only access the words queued by all fragments for F. While this means that the concurrent algorithm potentially performs more actual multiplications of elements than the original, in Section 5.1 we will see that these numbers of multiplications are almost the same. Of course, we could have allowed the fragments to communicate during the first phase but in our experiments this was significantly slower. The evidence presented in Section 5.1 supports this, the time spent waiting for other fragments to provide information outweighs the benefit of being able to deduce a relatively small additional number of products.

Throughout this section, we again let U be any semigroup, let S be a subsemigroup of U generated by $A \subseteq U$, and let $v : A^+ \to S$ be the unique homomorphism extending the inclusion map. We also define $f, l : A^+ \to A$ and $p, s : A^+ \to A^*$ to be the functions defined at the start of Section 3.

The following is the precise definition of a fragment required for our concurrent algorithm.

Definition 5.1. A *fragment for S* is a tuple (A, Y, K, ϕ) where:

(a) $A = \{a_1, \dots, a_r\}$ is a finite collection of generators for S where $a_1 < \cdots < a_r$;

(b) $Y = \{y_1, \dots, y_N\}$ is a collection of reduced words for S and $y_1 < y_2 < \cdots < y_N$;

(c) $1 \leq K \leq |Y| + 1$ and if $K = 1$, then $Y = A$, or if $K > 1$, then $y_{K-1} = \max\{y_i \in Y : |y_i| = |y_{K-1}|\}$;

(d) if $R \subseteq A^+$ is the set of all reduced words for elements of S, then

$$\phi : (A \times \{y_1, \dots, y_L\}) \cup (\{y_1, \dots, y_{K-1}\} \times A) \to R$$

where $v(\phi(u, v)) = v(uv)$ for all $u, v \in \mathrm{dom}(\phi)$ and either:
 (i) $L = K - 1$; or
 (ii) $L < K - 1$ and L is the largest value such that $|y_L| < |y_{K-1}|$.

Note that in part (c) the condition that $y_{K-1} = \max\{y_i \in Y : |y_i| = |y_{K-1}|\}$ implies that either y_K does not exist or $|y_K| > |y_{K-1}|$. We refer to the size, elements, extension, and so on of a fragment for a semigroup S, in the same way as we did for the snapshots of S.

We suppose throughout this section that however ϕ in Definition 5.1 is actually implemented, it supports concurrent reads of any particular value $\phi(x, y)$ when $(x, y) \in \mathrm{dom}(\phi)$ and that it is possible to define $\phi(x, y)$ for any $(x, y) \notin \mathrm{dom}(\phi)$ concurrent with any read of $\phi(x', y')$ for $(x', y') \in \mathrm{dom}(\phi)$. Of course, the difficulty is that we do not necessarily know whether (x, y) belongs to $\mathrm{dom}(\phi)$ or not. If one process is defining $\phi(x, y)$ and another is trying to read $\phi(x, y)$, it is not possible to determine whether (x, y) belongs to $\mathrm{dom}(\phi)$ without locking, which we want to avoid. We avoid this in Algorithm 5.6 (ConcurrentFroidurePin) because ϕ is defined for those (w, a) and (a, w) where $a \in A$ and w is a reduced word whose length is strictly less than the minimum length of a word some of whose right multiples are not known.

The implementation in LIBSEMIGROUPS [11] represents ϕ as a pair of C++ Standard Template Library vectors, which support this behaviour provided that no reallocation occurs when we are defining $\phi(x, y)$ for $(x, y) \notin \mathrm{dom}(\phi)$. In ConcurrentFroidurePin, all of the reduced words $w \in A^+$ of a given length are produced before any value of $\phi(w, a)$ or $\phi(a, w)$ is defined, and so we can allocate enough memory to accommodate these definitions and thereby guarantee that it is safe to read and write values of $\phi(x, y)$ concurrently.

If $(A, A, 1, \emptyset, \emptyset)$ is a minimal snapshot for S, we refer to the collection of fragments

$$(A, \{a \in A : b(a) = j\}, 1, \emptyset) \qquad j \in \{1, \dots, k\}$$

as a *minimal collection of fragments* for S.

The next lemma describes when some fragments for S can be assembled into a snapshot of S.

Lemma 5.2. *Suppose that* $(A, Y_1, K_1, \phi_1), \dots, (A, Y_k, K_k, \phi_k)$ *is a collection of fragments for* S *which is either minimal or* $Y_i = \{y_{i,1}, \dots, y_{i,N_i}\}$ *and the following hold:*

(i) $Y_i \cap Y_j = \emptyset$ *for all* i, j, $i \neq j$;

(ii) *for all reduced* $w \in A^+$ *either*

$$w \in \bigcup_{i=1}^{k} Y_i \quad \text{or} \quad w > \max \bigcup_{i=1}^{k} Y_i;$$

(iii) *if* $K_j > 1$, *then* $|y_{j,K_j-1}| = \max\{|y| : y \in Y_j, |y| \leq c\}$ *where* $c = \max\{|y_{i,K_i-1}| : 1 \leq i \leq k, K_i > 1\}$

(iv) $\operatorname{dom}(\phi_i) = (A \times \{y_{i,1}, \dots, y_{i,K_i-1}\}) \cup (\{y_{i,1}, \dots, y_{i,K_i-1}\} \times A)$ *for all* i; *and*

(v) $\operatorname{im}(\phi_i) \subseteq \bigcup_{j=1}^{k} Y_j$ *for all* i.

If $Y = \bigcup_{i=1}^{k} Y_i = \{y_1, \dots, y_N\}$, $y_1 < \cdots < y_N$, $K \in \mathbb{N}$ *is such that* $y_{K-1} = \max\{y \in Y : |y| = c\}$, *and* $\phi = \bigcup_{i=1}^{k} \phi_i$, *then* $(A, Y, K, \emptyset, \phi)$ *is a snapshot of* S.

Proof. If the collection of fragments is minimal, then clearly the union, as defined in the statement, is a minimal snapshot for S.

Suppose that the collection of fragments is not minimal. It follows that there exists $i \in \{1, \dots, k\}$ such that $K_i > 1$, and so the value c in part (iii) is well-defined. It is clear that Definition 3.1(a) to (d) are satisfied. It therefore suffices to show that part (e) of Definition 3.1 holds. Part (i) of the assumption of this lemma implies that ϕ is well-defined. It suffices to verify that

$$\begin{aligned} &\operatorname{dom}(\phi) = (A \times \{y_1, \dots, y_{K-1}\}) \cup (\{y_1, \dots, y_{K-1}\} \times A), \\ &\operatorname{im}(\phi) \subseteq Y, \qquad \text{and} \qquad v\big(\phi(u, v)\big) = v(uv) \end{aligned}$$

for all $u, v \in \operatorname{dom}(\phi)$. That $\operatorname{im}(\phi) \subseteq Y$ follows from (v). If $(u, v) \in \operatorname{dom}(\phi)$, then $(u, v) \in \operatorname{dom}(\phi_i)$ for some i, and so $v\big(\phi(u, v)\big) = v\big(\phi_i(u, v)\big) = v(uv)$.

Let $(a, y_i) \in A \times \{y_1, \dots, y_{K-1}\}$. Then $y_i \in Y_j$ for some j and so $y_i = y_{j,t}$ for some t. But $|y_{j,t}| = |y_i| \leq |y_{j,K_j-1}|$ by the definition of K and part (iii) of the assumption of this lemma. Hence either $|y_{j,t}| < |y_{j,K_j-1}|$ and so $y_{j,t} < y_{j,K_j-1}$, or $|y_{j,t}| = |y_{j,K_j-1}|$ and, by Definition 5.1(c), $y_{j,t} < y_{j,K_j-1}$. In either case, $t \leq K_j - 1$, and so $(a, y_i) = (a, y_{j,t}) \in \operatorname{dom}(\phi_j) \subseteq \operatorname{dom}(\phi)$. If $(y_i, a) \in \{y_1, \dots, y_{K-1}\} \times A$, then $(y_i, a) \in \operatorname{dom}(\phi)$ by a similar argument.

If $(a, y_i) \in \operatorname{dom}(\phi)$, then $(a, y_i) \in \operatorname{dom}(\phi_j)$ for some j. Hence $y_i \in \{y_{j,1}, \dots, y_{j,K_j-1}\}$. It follows that, since $y_{K-1} = \max\{y \in Y : |y| = c\}$, $y_{K-1} \geq y_{j,K_j-1} \geq y_i$, and so $(a, y_i) \in A \times \{y_1, \dots, y_{K-1}\}$. If $(y_i, a) \in \operatorname{dom}(\phi)$, then $(y_i, a) \in \{y_{i,1}, \dots, y_{i,K_i-1}\} \times A$ by a similar argument. Therefore $\operatorname{dom}(\phi) = (A \times \{y_1, \dots, y_{K-1}\}) \cup (\{y_1, \dots, y_{K-1}\} \times A)$, as required. □

Algorithm 5.4: ApplyGenerators

```
Input: A collection of fragments $(A, Y_1, K_1, \phi_1), \ldots, (A, Y_k, K_k, \phi_k)$ for a semigroup $S$ satisfying the
       hypothesis of Lemma 5.2 and $j \in \{1, \ldots, k\}$.
Output: A set $Q_j = \{(b(w), w) : |w| = c + 1\} \subseteq \{1, \ldots, k\} \times A^+ \setminus (Y_1 \cup \cdots \cup Y_k)$ where
        $c = \max\{|y_{i,K_i - 1}| : 1 \leq i \leq k\}$ and the tuple $(A, Y_j, K_j, \phi_j)$.
1  $Q_j := \varnothing$                                          $Q_j$ is a container for new words
2  while $K_j \leq |Y_j|$ and $|y_{j,K_j}| = c$ do
3      for $a \in A$ do                     loop over the generators in (short-lex) order
4          if $\phi_j(s(y_{j,K_j}), a) < s(y_{j,K_j})a$ then              $s(y_{j,K_j})a$ is not reduced
5              $y := \phi_i(s(y_{j,K_j}), a)$ where $s(y_{j,K_j}) \in Y_i$
6              $w := \phi_m(f(y_{j,K_j}), p(y)) \in Y_n$ where $p(y) \in Y_m$
7              if $n = j$ or $|w| < c$ then
8                  $\phi_j(y_{j,K_j}, a) := \phi_n(w, l(y))$
9                  continue
10         if $\nu(y_{j,K_j}a) = \nu(y)$ for some $y \in Y_1 \cup \cdots \cup Y_k$ then
11             $\phi_j(y_{j,K_j}, a) := y$
12         else
13             $Q_j \leftarrow Q_j \cup \{(b(y_{j,K_j}a), y_{j,K_j}a)\}$
14     $K_j \leftarrow K_j + 1$
15 return $Q_j$ and $(A, Y_j, K_j, \phi_j)$
```

In Algorithm 5.4 (ApplyGenerators), and more generally in our concurrent version of the Froidure–Pin Algorithm, we require a method for assigning reduced words $w \in A^+$ that do not belong to any existing fragment for S, to a particular fragment for S. If we want to distribute S into k fragments, then we let $b : R := \{w \in A^+ : w \text{ is reduced for } S\} \to \{1, \ldots, k\}$ be any function. Preferably so that our algorithms are more efficient, b should have the property that $|b^{-1}(i)|$ is approximately equal to $|R|/k$ for all i. For example, we might take a hash function for $\nu(w)$ modulo k, as the value of $b(w)$. If the number of fragments $k = 1$ or $b(w)$ is constant for all reduced words w for S, then ConcurrentFroidurePin is just FroidurePin with some extra overheads.

Lemma 5.3. *Algorithm 5.4 (ApplyGenerators) can be performed concurrently on each fragment $(A, Y_j, K_j, \emptyset, \phi_j)$ of its input.*

Proof. Every value assigned to ϕ_j in ApplyGenerators equals a value for ϕ defined in Update. It is possible that some assignments made in Update for ϕ cannot be made for ϕ_j in ApplyGenerators. In particular, in Update if $s(y_{j,K_j})a$ is not reduced, then $\phi_j(s(y_{j,K_j}), a)$ is always defined in Update but is only defined in some cases in ApplyGenerators. Hence that ϕ_j is well-defined follows by the proof of Lemma 3.3.

The values $\phi_i(s(y_{j,K_j}), a)$, $\phi_m(f(y_{j,K_j}), p(y))$, and $\phi_n(w, l(y))$ are read in ApplyGenerators and may belong to other fragments. But $|s(y_{j,K_j})|, |p(y)| < c$ and $\phi_n(w, l(y))$ is only used if $n = j$ or $|w| < c$. The only value which is written in ApplyGenerators is $\phi_j(y_{j,K_j}, a)$, and $|y_{j,K_j}| = c$. It follows that ApplyGenerators

Algorithm 5.5: ProcessQueues

Input: The output of Algorithm 5.4 (ApplyGenerators) for all $m \in \{1, \dots, k\}$ and $j \in \{1, \dots, k\}$.
Output: A fragment for S extending (A, Y_j, K_j, ϕ_j), containing every reduced word wa such that $(j, wa) \in Q_1 \cup \dots \cup Q_k$.

```
for $(b(wa), wa) \in Q_1 \cup \dots \cup Q_k$ do        loop over $Q_1 \cup \dots \cup Q_k$ in short-lex order on $wa$
    if $b(wa) = j$ then
        if $\nu(wa) = \nu(y)$ for some $y \in Y_j$ then
            $\phi_j(w, a) = y$
        else
            $Y_j \leftarrow Y_j \cup \{wa\}$
            $\phi_j(w, a) := wa$
return $(A, Y_j, K_j, \phi_j)$
```

only reads values of $\phi_d(u, a)$ or $\phi_d(a, u)$ when $d \neq j$ and $|u| < c$, while the algorithm only writes to values of $\phi_j(u, a)$ when $|u| = c$. Therefore there are no concurrent reads and writes in ApplyGenerators. □

Note that after applying ApplyGenerators, the tuple (A, Y_j, K_j, ϕ_j) is no longer a fragment because Definition 5.1(d) may not be satisfied.

The next algorithm, Algorithm 5.5 (ProcessQueues), performs the synchronisation step alluded to at the start of this section. We prove the validity of ProcessQueues in Lemma 5.4.

Lemma 5.4. *Algorithm* 5.5 (*ProcessQueues*) *returns a fragment for S satisfying Definition* 5.1(d)(ii) *and it can be performed concurrently on each* $j \in \{1, \dots, k\}$.

Proof. If ProcessQueues is run concurrently in k processes, for distinct values of j, then each process only writes to ϕ_j and only reads from $Q_1 \cup \dots \cup Q_k$. Hence ProcessQueues can be performed concurrently.

Parts (a) of Definition 5.1 holds trivially. For part (b), it suffices to note that wa added in line 6, is a reduced word, since we loop over the elements in $Q_1 \cup \dots \cup Q_k$ in short-lex order.

For part (c), we must show that $1 \leq K_j \leq |Y_j| + 1$ and that y_{j,K_j-1} is the maximum word in Y_j of length $|y_{j,K_j-1}|$. Since K_j is not modified by ProcessQueues the first condition holds. ApplyGenerators must have returned to obtain the input for ProcessQueues, which implies that y_{j,K_j-1} is the maximum word in Y_j of length $|y_{j,K_j-1}|$ before ProcessQueues is called. ProcessQueues only adds words to Y_j of length $c + 1 > |y_{j,K_j-1}|$ and so y_{j,K_j-1} is the maximum word in Y_j, and part (c) holds.

For part (d), suppose that $w \in Y_1 \cup \dots \cup Y_k$ and $a \in A$ are such that $\phi_j(w, a)$ is defined in ApplyGenerators or ProcessQueues. In either case, $\phi_j(w, a) \in Y$ and since we have shown that every element in Y is reduced, $\phi_j(w, a)$ must be also.

The input of ApplyGenerators satisfies Lemma 5.3, in particular part (iv), which states that before applying ApplyGenerators

$$\operatorname{dom}(\phi_j) = (A \times \{y \in Y_j : |y| < c\}) \cup (\{y \in Y_j : |y| < c\} \times A).$$

If $a \in A$ and $w \in Y_n$ such that $|w| = c$ and $b(wa) = j$, then either ApplyGenerators defines some $\phi_j(w, a)$ or (j, wa) is placed in Q_n. In the latter case, $\phi_j(w, a)$ is defined in ProcessQueues. Hence when ProcessQueues returns

$$\operatorname{dom}(\phi_j) = (A \times \{y_{j,1}, \dots, y_{j,L}\}) \cup (\{y_{j,1}, \dots, y_{j,K_j-1}\} \times A)$$

where $L < K_j - 1$ and L is the largest value such that $|y_L| < |y_{K_j-1}|$. That $\nu\big(\phi(u, v)\big) = \nu(uv)$ for all $u, v \in \operatorname{dom}(\phi)$ follows by the argument in the proof of Lemma 3.3. □

We have all of the ingredients to state the concurrent version of Froidure–Pin, the validity of which is proven in Lemma 5.5.

Lemma 5.5. *If $(A, Y, K, \emptyset, \phi)$ is a snapshot for a semigroup S and $M \in \mathbb{N}$, then Algorithm* 5.6 (*ConcurrentFroidurePin*) *returns a snapshot of S that extends $(A, Y, K, \emptyset, \phi)$ has at least* $\min\{M, |S|\}$ *elements.*

Proof. By Lemma 5.3 and 5.4, when line 8 is reached, every tuple (A, Y_j, K_j, ϕ_j) is a fragment for S, and the loops applying ApplyGenerators and ProcessQueues can be performed concurrently.

Algorithm 5.6: ConcurrentFroidurePin

```
Input: A collection of fragments $(A,Y_1,K_1,\phi_1),\dots,(A,Y_k,K_k,\phi_k)$ for a semigroup $S$ satisfying the
       hypothesis of Lemma 5.2 and $M \in \mathbb{N}$.
Output: A collection of fragments $(A,Y_1,K_1,\phi_1),\dots,(A,Y_k,K_k,\phi_k)$ for a semigroup $S$ satisfying
        the hypothesis of Lemma 5.2 and with size at least $\min\{M,|S|\}$.
1  $c := \max\{|y_{i,K_i-1}| : 1 \le i \le k,\ K_i > 1\} \cup \{1\}$
2  while $\exists j\ K_j \le |Y_j|$ and $|Y_1 \cup \dots \cup Y_k| < M$ do
3      $Q_1 := \varnothing, \dots, Q_k = \varnothing$
4      for $j \in \{1,\dots,k\}$ do                                  This can be done concurrently
5          $Q_j, (A,Y_j,K_j,\phi_j) \leftarrow$ ApplyGenerators$(A,Y_j,K_j,\phi_j)$
6      for $j \in \{1,\dots,k\}$ do                                  This can be done concurrently
7          $(A,Y_j,K_j,\phi_j) \leftarrow$ProcessQueues$(Q,j)$
8      for $j \in \{1,\dots,k\}$ do                                  This can be done concurrently
9          $L_j = \max\{i \in \mathbb{N} : |y_{j,i}| < c,\ y_{j,i} \in Y_j\}$
10         for $i \in \{L_j+1,\dots,K_j-1\}$ do          if $\{y \in Y_j : |y| = c\} = \varnothing$, then $K_j - 1 < L_j + 1$
11             for $a \in A$ do
12                 $\phi_j(a,y_{j,i}) := \phi_n(\phi_m(a,p(y_{j,i})),l(y_{j,i}))$ where $p(y_{j,i}) \in Y_m$ and $\phi_m(a,p(y_{j,i})) \in Y_n$
13     $c \leftarrow c+1$                         $c \leftarrow \max\{|y_{i,K_i-1}| : 1 \le i \le k,\ K_i > 1\}$
14 return $(A,Y_1,K_1,\phi_1),\dots,(A,Y_k,K_k,\phi_k)$
```

We will show that by the end of the for-loop started in line 8, $(A, Y_1, K_1, \phi_1), \ldots, (A, Y_k, K_k, \phi_k)$ is a collection of fragments satisfying the conditions of Lemma 5.2, and that the steps within the for-loop can be executed concurrently for each fragment. That the values assigned to ϕ_j are valid follows by the argument in the proof of Lemma 3.4. Suppose that $j \in \{1, \ldots, k\}$ is given. When the for-loop started in line 8 is complete, (A, Y_j, K_j, ϕ_j) clearly satisfies Definition 5.1(a) to (c), since ϕ_j is the only component which is modified inside the for-loop. Hence it suffices to verify Definition 5.1(d). That $v(\phi_j(u, v)) = v(uv)$ for all $u, v \in \mathrm{dom}(\phi_j)$ follows again by the same argument as in Lemma 3.4.

By Lemma 5.4, before ϕ_j is modified in this loop it satisfies Definition 5.1(d)(ii), i.e.

$$\mathrm{dom}(\phi_j) = (A \times \{y_{j,1}, \ldots, y_{j,L}\}) \cup (\{y_{j,1}, \ldots, y_{j,K_j-1}\} \times A)$$

where $L < K_j - 1$ is the largest value such that $|y_L| < |y_{K_j-1}|$. After the for-loop starting in line 10,

$$\mathrm{dom}(\phi_j) = (A \times \{y_{j,1}, \ldots, y_{j,K_j-1}\}) \cup (\{y_{j,1}, \ldots, y_{j,K_j-1}\} \times A). \tag{5.6}$$

In other words, Definition 5.1(d)(i) holds, and so (A, Y_j, K_j, ϕ_j) is a fragment.

It is clear that the collection of fragments contains at least $\min\{M, |S|\}$ elements. Next, we show that the for-loop started in line 8 can be executed concurrently. Since $|p(y_{j,i})| < |y_{j,i}| = c$ and so $\phi_m(a, p(y_{j,i}))$ is defined before line 8. Furthermore, $|\phi_m(a, p(y_{j,i}))| \le |y_{j,i}| = c$ and by, (5.6), $\phi_n(w, b)$ is defined for all reduced w of length at most c and for all $b \in A$. In other words, $\phi_n(\phi_m(a, p(y_{j,i})), l(y_{j,i}))$ is defined before line 8.

It remains to show that the collection of fragments satisfies the conditions in Lemma 5.2. For Lemma 5.2(i), a reduced word y belongs to Y_j if and only if $b(y) = j$, and hence $Y_i \cap Y_j = \emptyset$ if $i \neq j$. To show that Lemma 5.2(ii) holds, it suffices to show that $\bigcup_{i=1}^{k} Y_i = \{w \in A^+ : |w| \le |y_N|, w \text{ is reduced}\}$. Suppose that $w \in A^+$ is reduced and $|w| \le |y_N|$. Then $|p(w)| < |y_N|$ and so $p(w) \in \{y_{j,1}, \ldots, y_{j,K_j-1}\}$ for some j. Hence $(p(w), l(w)) \in \mathrm{dom}(\phi_j)$ and so $w = \phi_j(p(w), l(w)) \in \bigcup_{i=1}^{k} Y_i$. Within ConcurrentFroidurePin the values of K_j are only modified in the calls to ApplyGenerators. Every call to ApplyGenerators increases K_j so that either $|y_{j,K_j-1}| = c + 1$, or there are no words of length $c + 1$ in the jth fragment. In other words, Lemma 5.3(iii) holds. We showed in (5.6) that Lemma 5.2(iv) holds and Lemma 5.2(v) holds trivially. □

5.1. Experimental results. In this section we compare the original version of Algorithm 3.2 (FroidurePin) as implemented in LIBSEMIGROUPS [11], and the concurrent version in Algorithm 5.6 (ConcurrentFroidurePin). The implementation of ConcurrentFroidurePin in LIBSEMIGROUPS [11] uses thread based parallelism using C++11 Standard Template Library thread objects.

We start by comparing the number of products of elements in S that are actually computed in FroidurePin and ConcurrentFroidurePin. In [4], Theorem 3.2, it is shown that the number of such products in FroidurePin is $|S| + |R| - |A| - 1$ where R is the set of relations for S generated by FroidurePin. One of the main advantages of the Froidure–Pin Algorithm, concurrent or not, is that it avoids multiplying elements of S as far as possible by reusing information learned about S at an earlier stage of the algorithm. This is particularly important when the complexity of multiplying elements in S is high. ConcurrentFroidurePin also avoids multiplying elements of S, but is more limited in its reuse of previously obtained information. The number of products of elements S depends on the number of fragments k used by ConcurrentFroidurePin and the function $b : A^* \to \{1, \ldots, k\}$. The *full transformation monoid* T_n of degree $n \in \mathbb{N}$ consists of all functions from $\{1, \ldots, n\}$ to $\{1, \ldots, n\}$ under composition of functions. It is generated by the following transformations:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n-1 & n \\ 2 & 3 & 4 & \cdots & n & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 2 & 3 & \cdots & n-1 & n \\ 2 & 1 & 3 & \cdots & n-1 & n \end{pmatrix},$$
$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n-1 & n \\ 1 & 2 & 3 & \cdots & n-1 & 1 \end{pmatrix}.$$

We compare the number of products of elements of S in FroidurePin and ConcurrentFroidurePin for each of $k = 1, 2, 4, \ldots, 32$ fragments and for the full transformation monoid of degree $n = 3, \ldots, 8$; see Figure 1. The number of products in FroidurePin is a lower bound for the number in ConcurrentFroidurePin, and we would not expect ConcurrentFroidurePin to achieve this bound. However, from

n	3	4	5	6	7	8
$\lvert T_n \rvert = n^n$	27	256	3125	46656	823543	16777216
FroidurePin	40	340	3877	54592	926136	18285899
ConcurrentFroidurePin (1 fragments)	45	405	4535	66293	1048758	20235231
ConcurrentFroidurePin (2 fragments)	45	415	4586	67835	1106562	22763829
ConcurrentFroidurePin (4 fragments)	47	406	4587	67682	1153668	23093948
ConcurrentFroidurePin (8 fragments)	46	405	4589	67433	1155484	23411798
ConcurrentFroidurePin (16 fragments)	46	402	4596	67578	1153832	23616000
ConcurrentFroidurePin (32 fragments)	46	404	4563	67755	1152818	23566915

Table 1. Comparison of the number of products of elements in Algorithms 3.2 (FroidurePin) and 5.6 (ConcurrentFroidurePin) applied to the full transformation monoid T_n.

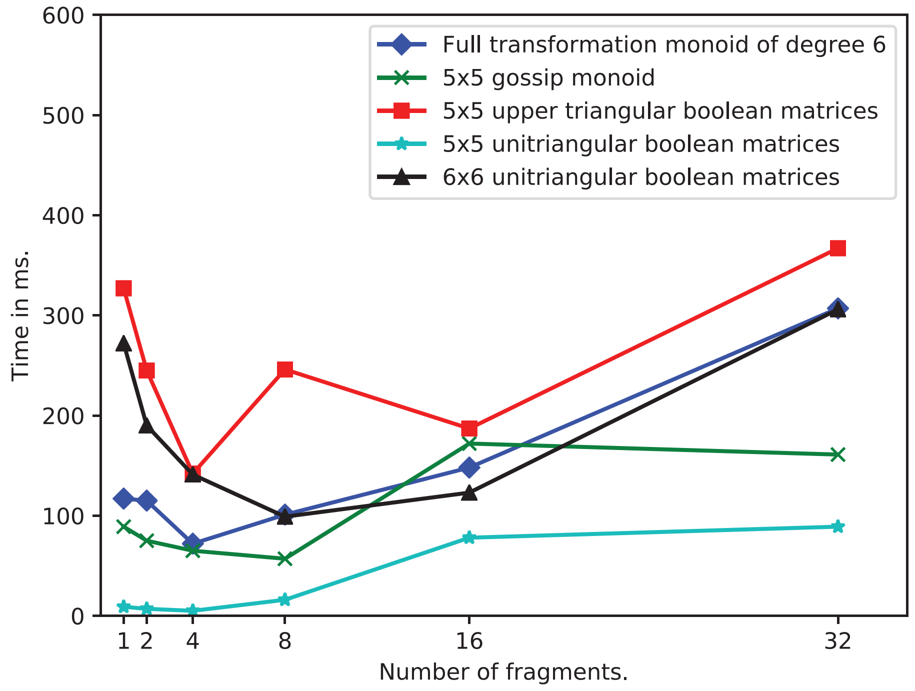

Figure 5. Run times for Algorithm 5.6 (ConcurrentFroidurePin) against number of fragments.

the table in Figure 1 it can be observed that the number of products in ConcurrentFroidurePin is of the same order of magnitude as that in ConcurrentFroidurePin. In [4], it was noted that there are 678223072849 entries in the multiplication table for T_7 but only slightly less than a million products are required in FroidurePin; we note that only slightly more than a million products are required in ConcurrentFroidurePin.

In Figures 5 and 6 we plot the performance of ConcurrentFroidurePin against the number of fragments it uses for a variety of examples of semigroups S. As would be expected, if the semigroup S is relatively small, then there is no advantage to using ConcurrentFroidurePin; see Figure 5. However, if the semigroup S is relatively large, then we see an improvement in the runtime of ConcurrentFroidurePin against FroidurePin; see Figure 6 and 7. Note that the monoid of reflexive 5×5 Boolean matrices has 1414 generators.

All of the computations in this section were run on a Intel Xeon CPU E5-2640 v4 2.40GHz, 20 physical cores, and 128GB of DDR4 memory.

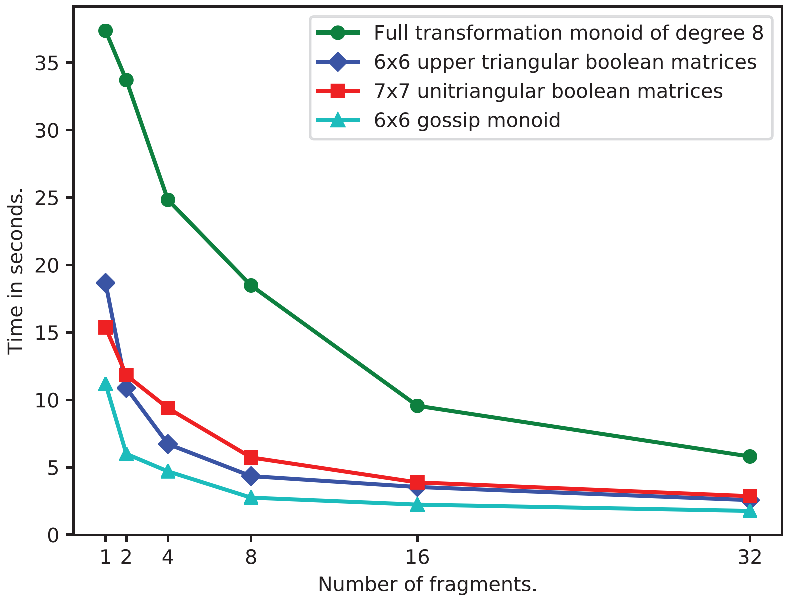

Figure 6. Run times for Algorithm 5.6 (ConcurrentFroidurePin) against number of fragments.

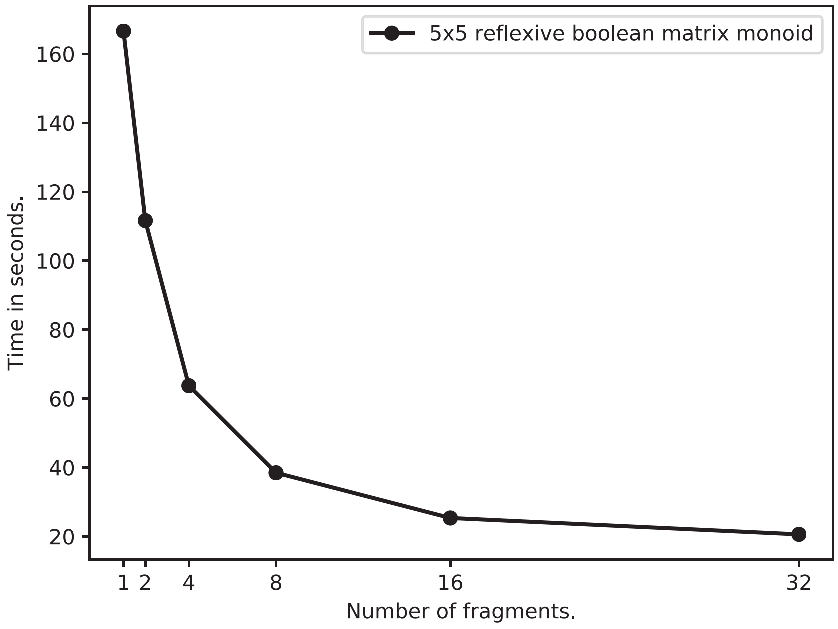

Figure 7. Run times for Algorithm 5.6 against number of fragments.

References

[1] C. Donoven, J. D. Mitchell, and W. Wilson, Computing maximal subsemigroups of a finite semigroup, submitted https://arxiv.org/abs/1606.05583

[2] J. East, A. Egri-Nagy, J. D. Mitchell, and Y. Péresse, Computing finite semigroups, to appear in *Journal of Symbolic Computation*, http://arxiv.org/abs/1510.01868

[3] Luke Elliot, Alex Levine, James D. Mitchell, and Nicolas M. Thiéry, libsemigroups python bindings, https://github.com/james-d-mitchell/libsemigroups-python-bindings

[4] Véronique Froidure and Jean-Eric Pin, Algorithms for computing finite semigroups, In *Foundations of computational mathematics* (*Rio de Janeiro*, 1997), pages 112–126. Springer, Berlin, 1997.

[5] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.8.7*; 2017. http://www.gap-system.org

[6] D. Holt with B. Eick and E. O'Brien, *Handbook of computational group theory*, CRC Press, Boca Raton, Ann Arbor, London, Tokyo, 2004.

[7] John M. Howie, *Fundamentals of semigroup theory*, volume 12 of *Londo, Mathematical Society Monographs. New Series*, The Clarendon Press Oxford University Press, New York, 1995, Oxford Science Publications.

[8] Janusz Konieczny, Green's equivalences in finite semigroups of binary relations, *Semigroup Forum*, 48(2):235–252, 1994.

[9] Gerard Lallement and Robert McFadden, On the determination of Green's relations in finite transformation semigroups, *J. Symbolic Comput.*, 10(5):481–498, 1990.

[10] S. A. Linton, G. Pfeiffer, E. F. Robertson, and N. Ruškuc, Groups and actions in transformation semigroups, *Math. Z.*, 228(3):435–450, 1998.

[11] J. D. Mitchell et al, *libsemigroups – C++ library – version* 0.3.1, May 2017, https://james-d-mitchell.github.io/libsemigroups/

[12] J. D. Mitchell et al, *Semigroups – GAP package, Version* 3.0.1, June 2017, http://gap-packages.github.io/Semigroups/

[13] J. Jonušas and J. D. Mitchell, *Benchmarking libsemigroups*, https://james-d-mitchell.github.io/2017-06-09-benchmarks/

[14] Jean-Eric Pin, Semigroupe 2.01: a software for computing finite semigroups, April 2009, https://www.irif.fr/~jep/Logiciels/Semigroupe2.0/semigroupe2.html

[15] Ákos Seress, *Permutation group algorithms*, volume 152 of *Cambridge Tracts in Mathematics*, Cambridge University Press, Cambridge, 2003.

[16] Charles C. Sims, Computational methods in the study of permutation groups, In *Computational Problems in Abstract Algebra* (*Proc. Conf., Oxford*, 1967), pages 169–183. Pergamon, Oxford, 1970.

[17] Charles C. Sims, *Computation with finitely presented groups*, Encyclopedia of mathematics and its applications, Cambridge University Press, Cambridge, England, New York, 1994.

[18] J. Rhodes and B. Steinberg, *The q-theory of finite semigroups*, Springer Monographs in Mathematics, New York, 2009.

Received March 16, 2017; revision received June 19, 2017

J. Jonušas, TU Wien, DA 05 F22, Wiedner Hauptstraße 8-10, 1040 Wien, Austria
E-mail: julius.jonusas@tuwien.ac.at

J. D. Mitchell, University of St Andrews, Mathematical Institute, North Haugh,
St Andrews, Fife KY16 9SS, Scotland
E-mail: jdm3@st-andrews.ac.uk

M. Pfeiffer, University of St Andrews, Jack Cole Building, North Haugh, St Andrews,
Fife KY16 9SX, Scotland
E-mail: markus.pfeiffer@st-andrews.ac.uk