Department of Computer Science,
University of Otago

Technical Report OUCS-2004-19

On the Permutational Power of Token Passing Networks

Authors:
M. H. Albert
Department of Computer Science, University of Otago
N. Ruškuc,
School of Mathematics and Statistics, University of St Andrews
S. Linton
School of Computer Science, University of St Andrews

Status: Submitted for publication to the London Mathematical Society

---

Department of Computer Science,
University of Otago, PO Box 56, Dunedin, Otago, New Zealand

http://www.cs.otago.ac.nz/research/techreports.html

ON THE PERMUTATIONAL POWER OF TOKEN PASSING NETWORKS.

M. H. ALBERT, N. RUŠKUC, AND S. LINTON

Abstract. A token passing network is a directed graph with one or more specified input vertices and one or more specified output vertices. A vertex of the graph may be occupied by at most one token, and tokens are passed through the graph. The reorderings of tokens that can arise as a result of this process are called the language of the token passing network. It was known that these languages correspond through a natural encoding to certain regular languages. We show that the collection of such languages is relatively restricted, in particular that only finitely many occur over each fixed alphabet.

1. Introduction

The study of graphs whose vertices can be occupied by tokens, or pebbles, which are moved along the edges has ranged from recreational mathematics [8, 10] to motion planning and related topics [3, 4, 6]. In most of these papers the problem is restricted to moving a fixed set of pebbles within a given graph, generally aiming to obtain a specific configuration. On the other hand, early works such as [5, 7, 9] dealt in a similar way with moving tokens, now thought of as items of data, within a network (represented as a directed graph) with the aim of producing specified outputs from a fixed input, or sorted output from a variable input.

In the latter area the problem of identifying permutations which could be produced when the network was restricted to a fixed size was considered in [2]. They showed that, under a natural encoding scheme, the collection of permutations generated by a token passing network is always a regular language. In this paper we continue the analysis of these collections of permutations and establish in Theorem 1 and Theorem 2 that, in effect, for each alphabet size, there are only finitely many such languages.

In Section 6 we provide a complete catalog of these languages over the three letter alphabet, along with networks producing them. We also provide some examples to show that certain natural conjectures about the behaviour of these networks are not correct. The results of Section

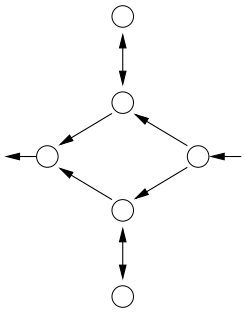

FIGURE 1. $S_{2,2}$, a network of two stacks of capacity two in parallel

6 are obtained by implementing some of the implicit computational methods introduced in [1] and [2],

We conclude this introduction with an informal, but illustrative example drawn from [2]. Consider the network $S_{2,2}$ shown in Figure 1. This network consists essentially of two stacks in parallel, each capable of containing up to two items. Input arrives at the rightmost node (shown by an inward pointing arrow without a source), and output occurs at the leftmost node (an outward pointing arrow without a target). We think of this network operating non-deterministically, with the following basic operations:

- If the input node is empty, then a new token may be added to the network at the input node. The input source is considered to be a (potentially infinite) queue containing tokens labelled 1, 2, ...
- If the source of an edge is occupied by a token, and its target is not, then the token may be moved along that edge.
- If the output node is occupied, then the token on it may be removed from the network.
- If the network is empty, operation may halt.

The output of a particular run of the network is the permutation $\pi = p_1 p_2 \cdots p_n$ where p_i is the label of the ith token removed from the network. The set of all such permutations will be denoted $\mathrm{Out}(S_{2,2})$.

Once four tokens are present in the network $S_{2,2}$ the next output symbol will be one of those four. Therefore, each element p_i in an output permutation π is one of the four smallest of the remaining elements. This suggests using the *rank encoding* of π to describe it, replacing each p_i by its rank among the remaining elements of π. Under this encoding $\mathrm{Out}(S_{2,2})$ forms a regular language. Of perhaps more interest are the minimal elements of the complement of $\mathrm{Out}(S_{2,2})$. These are permutations not in $\mathrm{Out}(S_{2,2})$ but with the property that if any single symbol is deleted from them, then the resulting permutation (that is

the output permutation of the remaining input items) is in $\mathrm{Out}(S_{2,2})$. It was noted in [2] that this set is infinite, containing at least the permutations:

$$4, 1, 6, 3, 8, 5, 10, 7, \ldots, 4n, 4n-3, 2, 4n-1$$

for any n. As a consequence of the methods of [1] and our explicit computations outlined below we can report that these permutations together with all the permutations of length 5 beginning with a 5 are the complete set of minimal elements in the complement of $\mathrm{Out}(S_{2,2})$.

2. Definitions and basic results

In a token passing network as defined informally above, we move tokens from vertex to vertex along directed edges, sometimes adding new tokens to the network at specified input vertices, and sometimes removing them from specified output vertices. Operation is to halt at any point when there are no tokens in the network. We now formalize this definition.

A *token passing network*, $\mathcal{T}(G, I, O)$, consists of a directed graph G together with non-empty subsets I and O of the vertices of G. When clear from context, we suppress the parameters. Of course I represents the set of input vertices, and O the set of output vertices.

Token passing networks operate as described in the introduction. Under this form of operation, in order to identify the output permutation, it suffices to know the *rank* of each output token, among all the remaining elements of the permutation. This rank cannot exceed the number of vertices in the underlying graph G of $\mathcal{T}$. The *rank encoding* of a permutation is obtained by replacing each symbol with its rank among the remaining symbols. By concentrating on the rank of each token, rather than its actual value, we can describe the operation of $\mathcal{T}$ quite simply.

A *state* of $\mathcal{T}$ consists of a sequence (possibly empty) of vertices of the network, not containing any duplicates. This represents the situation where the vertices in the sequence are occupied by numbered tokens, whose relative ordering agrees with the ordering of the sequence. That is, the smallest token occupies the first vertex of the sequence, the second smallest token the second vertex, and so on.

The *primitive transitions* in a token passing network in state s are of the following types:

Input: If $i \in I$ and i does not occur in s, then there is a transition $s \to si$.

Movement: If $s = avb$ and $v \to w$ is an edge of G and w does not occur in s then there is a transition $s \to awb$.

Output: If $s = aob$ with $o \in O$ then there is a transition $s \to ab$

A *run* of $\mathcal{T}$ begins with the empty state, and proceeds by some sequence of primitive transitions. At the end of this sequence the state of $\mathcal{T}$ must again be empty. The permutation produced by such a run is the decoded form of the sequence of tokens removed during output steps. For instance if this sequence were 23211 this would represent the permutation 24315.

The *output class*, $\mathrm{Out}(\mathcal{T})$ consists of all the permutations π that can be produced by some run of $\mathcal{T}$. This output class is closed under deletion. That is, if $\pi \in \mathrm{Out}(\mathcal{T})$ and we delete a symbol of π, then re-index the remaining symbols, preserving their relative order, to produce a permutation π' then it is also the case that $\pi' \in \mathrm{Out}(\mathcal{T})$. This is because we can produce the permutation π' by following the run of $\mathcal{T}$ which produced π but ignoring any operation on the token representing the element we wish to delete. A collection of permutations closed under this deletion operation is called a *pattern class*.

The style of argument of the preceding paragraph, will be repeated in a number of different contexts. When we use it, we will generally refer to the tokens we are ignoring in some modified run of $\mathcal{T}$ as *ghost tokens*.

The runs of $\mathcal{T}$ can be identified with the accepting computations of a non-deterministic finite state automaton by considering input and movement transitions as ϵ-transitions, and output transitions as transitions on the symbol k where k is the index of the output symbol in s. Under this identification the language accepted by $\mathcal{T}$, which we denote $L(\mathcal{T})$ is precisely the rank encoding of the collection of permutations $\mathrm{Out}(\mathcal{T})$. This is essentially the content of Theorem 1 of [2].

Let k be a positive integer. A *k-buffer* is the token passing network $\mathcal{B}_k = \mathcal{T}(K, K, K)$ where K is any directed graph on k vertices. From an operational standpoint any two k-buffers are completely equivalent, so the notation above is justified. The language $B(k) = L(\mathcal{B}_k)$ consists of the rank encodings of all permutations where the rank of any element is bounded by k. These permutations are referred to as *k-bounded*. Many other token passing networks produce this same language, for instance a cycle of k vertices with a single vertex serving as both input and output vertex.

The language accepted by a token passing network, $\mathcal{T}(G, I, O)$, is a sublanguage of $B(|G|)$ since at most $|G|$ tokens can occupy the graph. We define the *boundedness* of $\mathcal{T}$ to be the minimum k such that $L(\mathcal{T}) \subseteq B(k)$.

We will also consider token passing networks restricted to operate with a total of at most c tokens in the network at any one time. In terms of

the states introduced above, we restrict the operation of $\mathcal{T}$ to states represented by sequences of length at most c. We refer to such networks as *capacity restricted token passing networks* and denote the token passing network $\mathcal{T}$ restricted in this way by $\mathcal{T}_c$. Obviously $L(\mathcal{T}_c) \subseteq B(c) \cap L(\mathcal{T})$. We will see in Section 6 that, in general, this inclusion is proper. Specifically in Figure 6 we illustrate a token passing network which can produce certain 4-bounded permutations, but not without containing at least 5 tokens at some point in its operation.

The main results of this paper are the following:

Theorem 1. *Let c be a fixed positive integer. Then:*

$$\{L(\mathcal{T}) : \mathcal{T} \text{ a token passing network of boundedness } c\}$$

is finite.

Theorem 2. *Let c be a fixed positive integer. Then:*

$$\{L(\mathcal{T}_c) : \mathcal{T} \text{ a token passing network}\}$$

is finite.

These two results indicate that the permutational power of token passing networks is relatively restricted, in that there are infinitely many pattern classes represented by regular sublanguages of $B(C)$. Alternatively they can be viewed as compactness results for token passing networks. They say that, for a fixed boundedness or capacity bound, there is a finite test set, T, of permutations such that, if two token passing networks satisfying the boundedness conditions generate the same subset of T then they generate the same language. In Section 6 we will see that for boundedness (or capacity bound) 2, the set $T = \{21\}$ suffices, while for boundedness 3 we may take:

$$T = \{21, 321, 312, 31542, 324651\}.$$

3. Strongly cycle connected graphs

We say that a directed graph is *strongly cycle connected* if it has a strongly connected subgraph in which every edge belongs to a directed cycle of length at least three. For instance any directed graph on at least three vertices containing a directed Hamilton cycle is strongly cycle connected and the bi-directed orientation of an undirected graph is strongly cycle connected if and only if the graph contains no bridge edges.

In this section we prove a useful lemma which shows that strongly cycle connected graphs behave almost like buffers. In particular, if they appear as a subgraph within a token passing network, then each one can be used to store at least one token. Of course if a token passing

network $\mathcal{T}$ can store c elements, then $L(\mathcal{T}) \supseteq B(c)$, and the same holds for $L(\mathcal{T}_c)$.

Lemma 3. *Let C be a strongly cycle connected graph containing m vertices. The language of $\mathcal{T}(C, I, O)$ contains $B(m-2)$. If $m \in \{3, 4\}$ then it contains $B(m-1)$.*

Proof. We will show that if $m-2$ or fewer tokens are present in C, and if v is any vertex of C occupied by some token a, and $v \to w$ is an edge of C, then we can move a to w. In fact we will prove that this is possible after we have deleted any edges from C which do not belong to proper cycles.

This will suffice to prove the first part of the lemma. For, if fewer than $m-2$ tokens are in C then treating one of the unoccupied vertices as a "ghost token" and applying the above operation repeatedly, we can arrange for an input vertex to be unoccupied, allowing further input. Likewise, we can output any token from the network if it is occupied by $m-2$ or fewer tokens. So, any word in $B(m-2)$ can be produced by the network.

We refer to unoccupied vertices as *holes*. When we refer to the *distance from u to w* we mean the length of the shortest directed path from u to w. We will frequently make use of the fact that if a cycle C contains a hole, then we may advance the tokens in C along it by successively moving the hole backwards. When we wish to move a token t to a vertex v on the cycle by this method we will indicate this by the phrase *cycle t along C to v* or something similar.

Our assumption is that there are at least two holes. By moving holes backwards along paths we may assume that there are two holes at x and y either (Case 1) both at distance 1 from v, or (Case 2) x at distance 1, and y at distance 2 (along a path $v \to x \to y$). Choose a cycle A containing the edge $v \to w$. If A contains a hole we're done, after moving it backwards along the cycle to w. So, suppose otherwise. The critical situations in these two cases are illustrated in Figure 2 which may aid in visualising the details of the following arguments.

In case 1, choose a cycle A_x containing $v \to x$. If w belongs to A_x we can just use A_x in place of A in the preceding paragraph. So assume there is a vertex u of A not on A_x. Move the token a from v to x and move the hole from y backwards along the path from u to y, preserving the hole at v. Now cycle a back to v along A_x, reaching a state where A contains a hole.

Now consider case 2. Consider any cycle C_x containing $v \to x$. The either (i) $y \in C_x$ or (ii) $y \notin C_x$.

(sub-case i) Again if every vertex of A belongs to this cycle we're done. If not, move a hole from this cycle to v, and then move this hole back

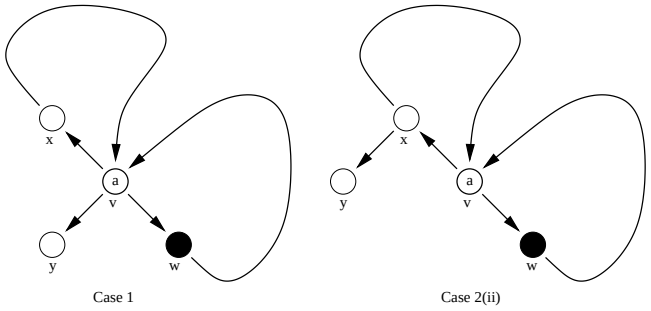

FIGURE 2. Critical cases for the argument concerning buffering capacity of strongly cycle connected graphs

along A to a vertex u not on C_x. Cycle C_x again until a returns to its original position, and A now contains a hole.

(sub-case ii) Cycle along C_x until there is a hole at v, but a does not occupy x. Move the token at x to y. This creates two holes at v and x. Move the hole at x onto A. Move a back onto A along C_x. Now A has a hole and we're done.

The final sentence of the lemma follows simply by considering the possible cases. A strongly cycle connected graph with 3 vertices contains a directed triangle, and trivially can produce any element of $B(2)$. If there are 4 vertices then either there is a directed 4-cycle, or two 3-cycles containing a common edge. In either case it is easy to check that all elements of $B(3)$ can be produced. □

The 5 vertex strongly connected graph consisting of two directed 3-cycles sharing a common vertex does not produce all of $B(4)$. So in general, the bound $m - 2$ in the lemma above cannot be improved. However, it is easy to see that, if G is a graph which has the property that for every edge $v \to w$, there is a path in $G \setminus v$ from w to every other vertex, then the language generated by a token passing network based on G contains $B(|G| - 1)$ since we can guarantee the movement of a token along any edge $v \to w$ provided there is at least one hole in the graph.

4. Token passing networks with fixed boundedness

The aim of this section is to prove Theorem 1. So, let a fixed positive real number c be given, and consider a language $L \subseteq B(c)$ produced by at least one token passing network of boundedness c. Among the networks which produce this language choose one, $\mathcal{T}$, which has the following properties:

- There is a single input vertex i and a single output vertex o.

- Among all such token passing networks, $\mathcal{T}$ is one with the smallest possible number of vertices.

We will establish a bound, independent of L, on the number of vertices of $\mathcal{T}$. Of course this suffices to prove the theorem since there are only finitely many token passing networks of such sizes.

First note that every vertex v of G has the property that there is a directed path from i to v and also one from v to o. For if there were a vertex for which this failed, then v could never be occupied in any run of $\mathcal{T}$. Thus v could be deleted without affecting the output language, and so the second condition on $\mathcal{T}$ would be contradicted.

Now choose a shortest directed path S:

$$i = v_0 \to v_1 \to \cdots \to v_m = o$$

from the input to the output vertex. We will call this the *spine* of $\mathcal{T}$. The complement of the spine will be called the *body*, its set of vertices will be denoted B, and the number of vertices in the body will be denoted b.

Since the boundedness of $\mathcal{T}$ is c, it follows that $b < c$. For, we can fill all the vertices of the body with tokens, working in reverse order of distance from i to avoid any possible blockages and then move the next token along the spine from input to output. If $b \geq c$ this would contradict the c-boundedness of $\mathcal{T}$.

We say that a vertex $v \in S$ *is spanned* if there is some vertex $w \in B$, and vertices $v_1 \in S$ not following v and $v_2 \in S$ not preceding v such that there is a directed path from v_1 to w not meeting S except at v_1 and one from w to v_2 not meeting S except at v_2. Note that either v_1 or v_2 (or even both) might equal v.

For each vertex $w \in B$, there is an earliest vertex $v_1 \in S$ which allows a directed path from v_1 to w not meeting S except at v_1, and a latest vertex $v_2 \in S$ allowing a directed path from w to v_2 not meeting S except at v_2. Since the sum of the lengths of these paths is at most $2b$ the distance on S between v_1 and v_2 must be at most $2b$. In particular the number of elements of S which are spanned is less than $2c^2$.

Furthermore by lemma 3, at most $3c$ of the vertices of the spine can lie on cycles of length 3 or more, or else we would be able to store at least $c + 1$ items, again violating the boundedness condition of $\mathcal{T}$.

Consider any vertex $v \in S$ which is neither spanned, nor belongs to any cycle. We call such a vertex *queuelike* because the only possible edges having v as an endpoint are of the form:

$$u \leftrightarrow v \leftrightarrow w$$

where $u, w \in S$ are the predecessor and successor of v in S.

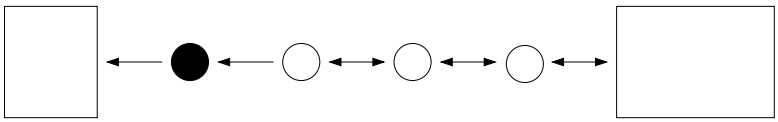

FIGURE 3. Schematic of a token passing network containing two internal queues. The shaded node is an internal queue of length one. The boxes may contain additional structure.

A maximal interval v_i through v_j of S consisting entirely of queuelike vertices with the additional property that for $i \leq k < j$ both the edges $v_i \to v_{i+1}$ and $v_{i+1} \to v_i$ are present in $\mathcal{T}$ will be called an *internal queue*. The structure of a token passing network containing an internal queue is illustrated in Figure 3 which also illustrates the fact that it is possible for an internal queue to consist of a single vertex. Note that every queuelike vertex belongs to a unique internal queue.

We now define the *backwards mobility* of an internal queue Q. This is the largest number k such that if tokens 1 through k are placed in order in Q (with 1 closest to the output vertex) then it is possible to rearrange them in Q in such a way that 1 is no longer the first element, using only the vertices of $\mathcal{T}$ between i and the end of Q.

Observation 4. *The backwards mobility of an internal queue cannot exceed c.*

Proof. Suppose that we had an internal queue Q of backwards mobility at least $c + 1$. Then certainly (by using virtual or ghost tokens if necessary) we could rearrange the initial contents 1 through $c+1$ of Q into a sequence:

$$a, \ldots, 1, \ldots$$

where 1 does not occur in the first position. By leaving the elements before 1 fixed we could now move 1 further down the sequence. So, after a series of such movements we could rearrange the contents of Q into some permutation π ending with 1. By repeating this same series of movements we could generate all the powers of π, including π^{-1}. However, in π^{-1}, token $c+1$ has been moved to the front of the queue. It could now be output from $\mathcal{T}$, contradicting c-boundedness. □

There is an analogous notion of *forwards mobility*, the largest number k such that if tokens 1 through k are placed in order in Q then it is possible to rearrange them in Q in such a way that k is no longer the last element, using only the vertices of $\mathcal{T}$ between the end of Q and o. By similar reasoning to the above, the forwards mobility of an internal queue is also at most c.

Observation 5. *If the length of an internal queue is greater than the sum of its forwards and backwards mobilities, then it is possible to produce a token passing network generating the same language as $\mathcal{T}$ but having fewer vertices.*

Proof. The key idea in proving this observation is that in such an internal queue, we can safely delete a vertex from the middle. It is instructive to begin with an extreme case of this observation, namely an internal queue whose forward and backward mobilities are both 0. Then, as soon as an object is added to the queue, because the backward mobility is 0, it will necessarily be the first to eventually leave the queue. So, we can reschedule any action involving addition of further elements to the queue until this element has finally left the queue. In fact, we can postpone its addition to the queue until the situation following the queue has been adjusted to the state at which this element leaves the queue for the final time. This then allows the element to simply be pushed directly through the queue. Thus the queue actually performs no useful function and can be short-circuited.

In the general case, suppose that we are about to add a token, a, to Q which already contains as many tokens as the sum of its forward and backward mobilities. This addition would create a token, k, in the middle of the queue whose order, relative to its successors and predecessors in the queue would have to remain fixed until some token had been removed permanently from the queue. The token k functions as a barrier between operations of $\mathcal{T}$ between i and it, and operations between it and o. So any operations of the latter type that precede the permanent removal of some element from Q can be advanced to occur before the addition of a to Q ensuring that at no point does Q ever contain more tokens than the sum of its forward and backward mobilities. Now the observation follows since vertices of Q in excess of this number are now superfluous to its effective operation. □

It follows from these two observations that, under the conditions imposed on $\mathcal{T}$, no internal queue has length at most $2c$. In fact, no internal queue can be bounded at both ends by one way edges, for such a queue has zero mobility in either direction. Thus there can be at most two internal queues in any block of queuelike elements. Since there are at most $2c^2$ non-queuelike vertices on S and at most two internal queues each of size at most $2c$ between non-queuelike vertices, there can be at most $8c^3$ queuelike vertices in $\mathcal{T}$, establishing Theorem 1.

5. Capacity restricted token passing networks

We now turn to the proof of Theorem 2. In this case the proof relies on a decomposition of the directed graph G underlying $\mathcal{T}$. If we can

identify part of the graph which can be operated in such a way as to simulate the effect of a buffer of size c, then $L(\mathcal{T}_c)$ (respectively $L(\mathcal{T})$) must equal $B(c)$. Conversely, we will argue that if we cannot find such parts of the graph, then the structure of the graph as a whole must be very simple, and, if the graph is sufficiently large, parts could be pruned away without affecting its permutational power.

Our graph decomposition occurs on two levels. The first is fairly standard. Let a directed graph G be given. Then G has a maximal acyclic quotient A, and the equivalence classes of the quotient map are precisely the strongly connected components of G. A strongly connected graph S has a maximal quotient T which is a bi-directed tree (the only type of strongly connected graph which has no cycles of length at least 3). The equivalence classes of this quotient map are strongly cycle connected, and we will call them the strongly cycle connected components of S.

The proof of the theorem now follows a standard inductive approach as in the previous section. We begin with a language of the form $L(\mathcal{T}_c)$. Among all the token passing networks whose capacity c restrictions generate this language and which have singleton input and output sets we choose a smallest one $\mathcal{T}(G, \{i\}, \{o\}))$. We then argue that the sizes of its strongly cycle connected components, and the two quotients A and T can be bounded, establishing a bound on $|G|$. The result then follows.

Let a language $L(\mathcal{T}_c)$ be given, and suppose that among all the token passing networks whose c-capacity restriction generates this language, $\mathcal{T}$ is one whose underlying graph, G, is as small as possible. As in the previous section this implies that for every vertex, v, of G there is at least one directed path from i to v and at least one directed path from v to o.

For convenience suppose that $L(\mathcal{T}_c) \neq B(c)$. The following lemma is directed towards limiting the size of the directed acyclic quotient A of G.

Lemma 6. *Let D be a directed acyclic graph, and let i and o be specified vertices of D. Suppose that for every vertex v of D there is a directed path from i to v and also one from v to o. If D has a directed spanning tree from i which has c or more leaves, then $L(\mathcal{T}_c(D, \{i\}, \{o\})) = B(c)$.*

Proof. We prove this result by describing an algorithm for actually running this network to produce any given word $\omega \in B(c)$. Take an inward directed spanning tree T leading to the output vertex o, and identify in it c vertices which are the leaves of some outward directed spanning tree, S, from i. We will use these vertices as storage, and

we shall arrange matters so that the following invariant properties are maintained:

- If a token is stored at v in T and w is a storage vertex such that the path in T from w to o passes through v then a token is also stored at w.
- If tokens are stored at v and w in T, and the path in T from v to o passes through w, then the token stored at w occurs earlier in ω than the one stored at v.

These properties certainly hold at the beginning of the run, since no tokens are stored. Suppose that we have reached some intermediate step of the run, and the next token of ω which we have not yet produced is ω_i. If ω_i is in storage, then by the second invariant property, the path from it to o is unblocked by other stored elements, and so we can move it to o, remove it from the network, and maintain both invariant properties.

Suppose now that ω_i has not yet been placed in storage. Consider the next input token, α, (which may not represent ω_i). Choose a currently unoccupied storage vertex, v, at the maximum available distance from o in T. If placing α at v maintains both invariants, then do so. If not, then consider the subtree of T rooted at v. All the storage vertices nearest the root are occupied. Among the occupants is one which occurs earliest in ω. Move this token from its current location v_1 to v. Now, if placing α at v_1 maintains both invariants then do so. Otherwise, consider the subtree of T rooted at v_1. Proceed as in the preceding step. Eventually, we must free a storage location at which α can be placed, since at worst, when we eventually must arrive at a leaf of T then we can place α there. Thus α can be added to storage while maintaining the invariants.

Continue this procedure for as long as necessary. In each phase we either store a new element, or output an element required next in ω. The invariants ensure that we never get blocked in any way, so eventually we will have succeeded in producing all of ω. □

Note that a minor modification of the proof establishes the same result when D has an inward directed spanning tree to o having c or more leaves.

Consider the graph A which is the directed acyclic quotient of G. By the lemma above and the remark following it we may assume the in-degree and out-degree of any vertex of A is at most c.

Choose an outward directed spanning tree T from i in A with the maximum possible number of leaves. By Lemma 6 this tree has at most $c - 1$ leaves. Thus it also has at most $c - 2$ branch vertices (vertices which do not have degree 2 consisting of an in-edge and an

out-edge). Consider any segment, S, between a branch vertex and a leaf, or between two branch vertices. There cannot be incoming edges to this segment from off the segment at as many as c vertices. If there were, we could use these c vertices as a form of storage to produce all of $B(c)$ (the argument is much the same as, but simpler than, that for Lemma 6). Moreover, the in-degree of any vertex in A is at most c, thus there are fewer than c^2 incoming edges to S from off S. However, there cannot be any edges internal to S other than the ones belonging to S, for the existence of such an edge would create either a cycle (impossible as A is acyclic), or the possibility of creating a spanning tree with more leaves.

The total number of segments is at most $2c-4$, hence the total number of edges of A which are not edges of T is bounded above by $(2c-4)c^2$. Suppose that there were a segment longer than $(2c-4)c^2M$ (for a value of M to be chosen later). Then this segment would contain a sequence of M consecutive vertices, each of degree exactly two in A. These vertices represent strongly connected components of G, so their actual structure may be somewhat more complex. However, each such component which is anything other than a simple 2-way path, connected only at its two ends within G is capable of storing at least one element. Thus, there cannot be as many as c vertices of this segment which represent components not of this type. Choosing $M > c^2$ we find more than c consecutive vertices of the segment which represent either individual vertices of G, or two way paths connected only at their endpoints. Any manipulation of c or fewer tokens along such a path can be carried out equally well when one of the vertices of the path is deleted, yielding a contradiction.

Now it remains only to show that the strongly connected components of G which are not simple two way paths connected only at their endpoints, can also contribute only a bounded amount to the size of G.

Consider a bi-directed tree T which is the quotient of such a strongly connected component of G by identifying vertices belonging to a common strongly cycle connected components. If T contains $c+1$ or more leaves, then it can store c items, and so produces $B(c)$. On the other hand if it contains at most c leaves, then it can contain at most $c-1$ branch vertices. Consider once more, a segment S of T.

As in the previous argument, there cannot be as many as c vertices of S which have an incoming edge from off S. Finally, by Lemma 3 there can not be as many as c vertices of S that represent non-trivial strongly cycle connected components. So, if S has more than $2c^2$ elements, then there is a block of more than c consecutive elements of S which are actually vertices of G and whose only adjacencies in G are the edges involving them in S. One of these can be deleted without affecting the language produced by $\mathcal{T}_c$, giving a contradiction.

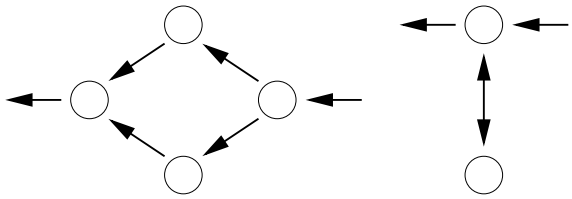

FIGURE 4. Simple token passing networks producing all 2-bounded permutations.

Since the total size of the strongly cycle connected components of G is already known to be bounded, we have succeeded in establishing Theorem 2.

6. Examples

We illustrate some of the preceding results with some example networks. In all cases we will adhere to the convention that our networks have a single input and a single output vertex.

The 2-bounded (but not 1-bounded) permutation classes produced by token passing networks are not very interesting. There is only one, which is the class of all 2-bounded permutations. Two simple networks producing it are shown in Figure 4.

Following the proofs of Theorems 1 and 2 for the case of 3-bounded classes shows that the underlying networks cannot be very complex. There are in fact precisely five 3-bounded classes that can be produced by token passing networks (and there is no distinction between the inherently bounded, or capacity bounded framework in this case). The five classes are:

- **(A)** All 3-bounded permutations. These can be produced by adding an extra vertex in the middle section of the left hand network in Figure 4, or by combining two copies of the right hand network in series, as well as by many other networks.
- **(B)** The 3-bounded permutations avoiding the pattern 321. These can be produced by two queues in parallel.
- **(C)** The 3-bounded permutations avoiding the pattern 312. These can be produced by a stack.
- **(D)** The 3-bounded permutations avoiding both the pattern 31542 and 32541.
- **(E)** A class whose set of minimal avoided patterns is infinite, given in the language of the rank encoding by:

$$322321,\ 3213(31)^*321.$$

Networks producing the latter four classes are shown in Figure 5.

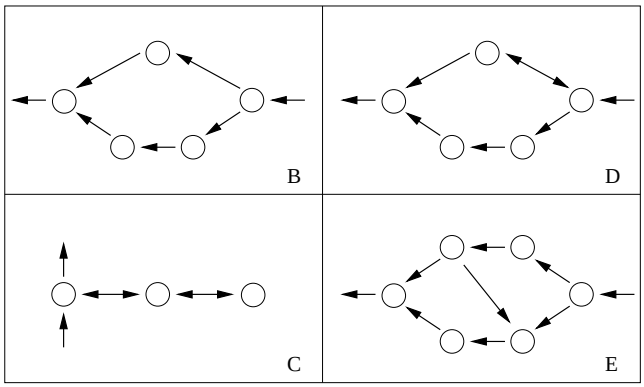

FIGURE 5. Examples of networks producing each possible non-universal 3-bounded class.

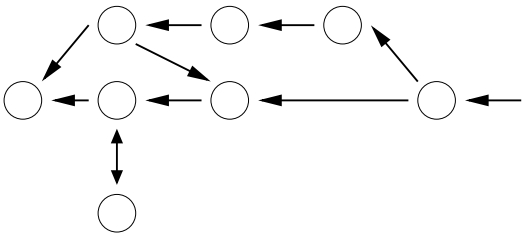

FIGURE 6. A network which produces the 4-bounded permutation with encoding 4222434111 but not without holding at least five tokens at some point.

A notable feature of the bottom right network in Figure 5 is that it produces the permutation 32541, but in doing so, the element 2 cannot be output before the element 4 is added to the network. This establishes that one type of greedy protocol for operating token passing networks is not effective.

Our final example is a token passing network $\mathcal{T}$ which has the property that:

$$L(\mathcal{T}_4) \neq L(\mathcal{T}) \bigcap B(4).$$

The network shown in Figure 6 can produce the permutation

$$4\,2\,3\,5\,8\,7\,10\,1\,6\,9$$

whose rank encoding is:

$$4\,2\,2\,2\,4\,3\,4\,1\,1\,1.$$

However, it cannot do so without at some point having five tokens in the network, namely token 10 must be added before token 7 is output. As this network has an inherent capacity of 5 this still leaves open the question of whether it is ever necessary to add more tokens to the network than its inherent capacity.

7. Summary and conclusions

We have shown that the permutational power of token passing networks is relatively limited, at least in the variety of classes of permutations that they can produce. For the sake of simplicity, the arguments we used in proving Theorems 1 and 2 were extremely conservative in the numerical bounds derived. In many cases multiplicative bounds could have been replaced by additive ones. So the actual size of the smallest network producing any given c-bounded language (if such a network exists) is quite small. This allowed us to determine the complete catalogue of such languages for $c = 3$. A similar catalogue for $c = 4$ would probably be feasible, though the collection of minimal avoided patterns is already much richer in this case.

We have skirted the issue of the complexity of determining $L(\mathcal{T})$ given $\mathcal{T}$. The underlying non deterministic automaton has, potentially, $|G|!$ states, although in practice many of these are unreachable and so need never be considered. The equivalent minimal deterministic automaton often exhibits a very straightforward structure. Determining the minimal avoided patterns using the methods of [1] requires several applications of non-deterministic transducers, complementation, and redeterminization and so is of exponential complexity in the worst case, and empirically also in practice. However, the final automaton produced by this procedure is generally quite small. It has been possible, by various ad hoc methods, to extend the practical range in which these computations can be carried out. Still, a general theoretical understanding of why the deterministic automata for the language and its minimal avoided patterns are so simple is lacking as is a method to exploit this apparent phenomenon in constructing one from the other.

The proof of Lemma 6 contains an algorithm for the efficient solution of the following problem:

> **Given:** A directed acyclic graph G with a specified input vertex i, and output vertex o and such that for any vertex v there is a directed path from i to v and from v to o, together with a positive integer c, not greater than the number of leaves of some directed spanning tree of G rooted at i.
> **Problem:** Use this graph to sort an incoming sequence of packets provided only with the guarantee that no packet will be preceded by c or more packets which should follow it.

This problem is the inverse view of the problem addressed in Lemma 6 and the algorithm provided in the proof of the lemma provide an online linear time solution of it in which no packet ever moves more than $2|G|$ times.

We have seen that the relationship between $L(\mathcal{T}_c)$ and $L(\mathcal{T}) \cap B(c)$ is not entirely simple. However, a range of questions of a similar character remain open. For example, given a token passing network of inherent boundedness c what is the minimum number of tokens we must allow in the network to guarantee producing every permutation in its language?

References

[1] M. H. Albert, M. D. Atkinson, and N. Ruškuc. Regular closed sets of permutations. *Theoret. Comput. Sci.*, 306(1-3):85–100, 2003.

[2] M. D. Atkinson, M. J. Livesey, and D. Tulley. Permutations generated by token passing in graphs. *Theoret. Comput. Sci.*, 178(1-2):103–118, 1997.

[3] V. Auletta, A. Monti, M. Parente, and P. Persiano. A linear-time algorithm for the feasibility of pebble motion on trees. *Algorithmica*, 23(3):223–245, 1999.

[4] Vincenzo Auletta and Pino Persiano. Optimal pebble motion on a tree. *Inform. and Comput.*, 165(1):42–68, 2001.

[5] Donald E. Knuth. *The art of computer programming*. Addison-Wesley Publishing Co., Reading, Mass.-London-Amsterdam, second edition, 1975. Volume 1: Fundamental algorithms, Addison-Wesley Series in Computer Science and Information Processing.

[6] C. H. Papadimitriou, P. Raghavan, M. Sudan, and H. Tamaki. Motion planning on a graph. In *Proc. of 35-th IEEE Symp. on Found. of Comp. Sc.*, pages 511–520, 1994.

[7] Vaughan R. Pratt. Computing permutations with double-ended queues. Parallel stacks and parallel queues. In *Fifth Annual ACM Symposium on Theory of Computing (Austin, Tex., 1973)*, pages 268–277. Assoc. Comput. Mach., New York, 1973.

[8] Daniel Ratner and Manfred Warmuth. The $(n^2 - 1)$-puzzle and related relocation problems. *J. Symbolic Comput.*, 10(2):111–137, 1990.

[9] Robert Tarjan. Sorting using networks of queues and stacks. *J. Assoc. Comput. Mach.*, 19:341–346, 1972.

[10] Richard M. Wilson. Graph puzzles, homotopy, and the alternating group. *J. Combinatorial Theory Ser. B*, 16:86–96, 1974.

Department of Computer Science, University of Otago, Dunedin, New Zealand

E-mail address: malbert@cs.otago.ac.nz

School of Mathematics and Statistics, University of St. Andrews, St. Andrews, United Kingdom

E-mail address: nik@mcs.st-and.ac.uk

School of Computer Science, University of St. Andrews, St. Andrews, United Kingdom

E-mail address: sal@dcs.st-and.ac.uk