Effective storage capacity of labeled graphs

Dana Angluin[a,1], James Aspnes[b,2,*], Rida A. Bazzi[c], Jiang Chen[d], David
Eisenstat[e], Goran Konjevod[f]

[a]*Department of Computer Science, Yale University*
[b]*Department of Computer Science, Yale University*
[c]*Computer Science and Engineering, SCIDSE, Arizona State University*
[d]*Google*
[e]*Department of Computer Science, Brown University*
[f]*Lawrence Livermore National Laboratory*

Abstract

We consider the question of how much information can be stored by labeling the
vertices of a connected undirected graph G using a constant-size set of labels,
when isomorphic labelings are not distinguishable. Specifically, we are interested
in the **effective capacity** of members of some class of graphs, the number of
states distinguishable by a Turing machine that uses the labeled graph itself in
place of the usual linear tape. We show that the effective capacity is related
to the **information-theoretic capacity** which we introduce in the paper. It
equals the information-theoretic capacity of the graph up to constant factors for
trees, random graphs with polynomial edge probabilities, and bounded-degree
graphs.

1. Introduction

We consider the question of how much information can be stored by la-
beling the vertices of a connected undirected graph G using a constant-size
set of labels, when isomorphic labelings are not distinguishable. An exact

---

information-theoretic bound is easily obtained by counting the number of isomorphism classes of labelings of G, which we call the **information-theoretic capacity** of the graph. More interesting is the **effective capacity** of members of some class of graphs, the number of states distinguishable by a Turing machine that uses the labeled graph itself in place of the usual linear tape.

The motivation for this model is self-organizing systems consisting of many communicating finite-state machines, where at any time, one machine (the location of the Turing machine head) takes a leadership role. Our main question is how much computing power such machines can cooperate to achieve. The answer depends on the inherent storage capacity of the communication graph, a function of its size (bigger gives more space) and symmetries (more symmetries makes the space harder to exploit).

In more detail, a **graph Turing machine** consists of an undirected connected graph G, each of whose nodes holds a symbol from some finite alphabet, together with a finite-state controller that can move around the graph and update the symbols written on nodes. Because there is no built-in sense of direction on an arbitrary graph, the left and right moves of a standard Turing machine controller are replaced by moves to adjacent graph nodes with a given symbol. If there is no such adjacent graph node, the move operation fails, which allows the controller to test its immediate neighborhood for the absence of particular symbols. If there is more than one such node, which node the controller moves to is chosen arbitrarily. (A more formal definition of the model is given in Section 3.)

The intent of this model is to represent what computations are feasible in various classes of simple distributed systems made up of a network of finite-state machines. Inclusion of an explicit head that can move nondeterministically to adjacent nodes (thus breaking at least local symmetries in the graph) makes the model slightly stronger than similar models from the self-stabilization literature (e.g., Dijkstra's original model in [1]) or population protocols [2]; we discuss the connection between our model and these other models in Section 2.

The main limitation on what a graph Turing machine can compute appears

to be the intrinsic **storage capacity** of its graph. For some graphs (paths, for example) the storage capacity is essentially equivalent to a Turing machine tape of the same size. For others (cliques, stars, some trees), the usable storage capacity may be much less, because symmetries within the graph make it difficult to distinguish different nodes with the same labeling. We define a notion of **information-theoretic capacity** of a graph (Section 4.1) that captures the number of distinguishable classes of labelings of the graph. Essentially this comes down to counting equivalence classes of labelings under automorphisms of the graph; it is related to the notion of the **distinguishing number** of a graph, which we discuss further in Section 2.3.

The information-theoretic capacity puts an upper bound on the **effective capacity** of the graph, the amount of storage that it provides to the graph Turing machine head (defined formally in Section 4.2). Extracting usable capacity requires not only that labelings of the graph are distinguishable in principle but that they are distinguishable to the finite-state controller in a way that allows it to simulate a classic Turing machine tape. We show, in Section 6.1, that an arbitrary graph with n nodes provides at least $\Omega(\log n)$ tape cells worth of effective capacity (which matches the information-theoretic upper bound for cliques and stars, up to constant factors). For specific classes of graphs, including trees (Section 7), bounded-degree graphs (Section 8), and random graphs with polynomial edge probabilities (Section 9), we show that the effective capacity similarly matches the information-theoretic capacity.

Notably, these classes of graphs are ones for which testing graph isomorphism is easy. Whether we can extract the full capacity of a general graph is open, and appears to be related to whether graph isomorphism for arbitrary graphs can be solved in **LOGSPACE**. We discuss this issue in Section 10.

2. Related work

2.1. Self-stabilizing models

A graph Turing machine bears a strong resemblance to a network of finite-state machines, which has been the basis for numerous models of distributed

3

computing, especially in the self-stabilization literature. Perhaps closest to the present work is the original self-stabilizing model of Dijkstra [1], where we have a collection of finite-state nodes organized as a finite connected undirected graph, and at each step some node may undergo a transition to a new state that depends on its previous state and the state of its immediate neighbors. The main difference between the graph Turing machine model and this is the existence of a unique head, and even more so, its ability to move to a single neighbor of the current node—these properties break symmetry in ways that are often difficult in classic self-stabilizing systems. A limitation of the graph Turing machine model is the restriction on what the head can sense of adjoining nodes: it cannot distinguish neighbors in the same state, or even detect whether one or many neighbors is in a particular state.

Itkis and Levin [3] give a general method for doing self-stabilizing computations in asynchronous general topology networks. Their model is stronger than ours, in that each node can maintain pointers to its neighbors (in particular, it can distinguish neighbors in the same state). Nonetheless, we have found some of the techniques in their paper useful in obtaining our current results.

2.2. Population protocols

There is also a close connection between our model and the **population protocol** model [2], in which a collection of finite-state agents interact pairwise, each member of the pair updating its state based on the prior states of both agents (see [4] for a recent survey on this and related models). This is especially true for work on population protocols with restricted communication graphs (for example, [5]). Indeed, it is *almost* possible to simulate a graph Turing machine in a population protocol, simply by moving the state of the head around as part of the state of the node it is placed on, and using interactions with neighbors to sense the local state. The missing piece in the population protocol model is that there is no mechanism for detecting the absence of a particular state in the immediate neighborhood. Although a fairness condition implies that every neighbor will make itself known eventually, the head node has no way to
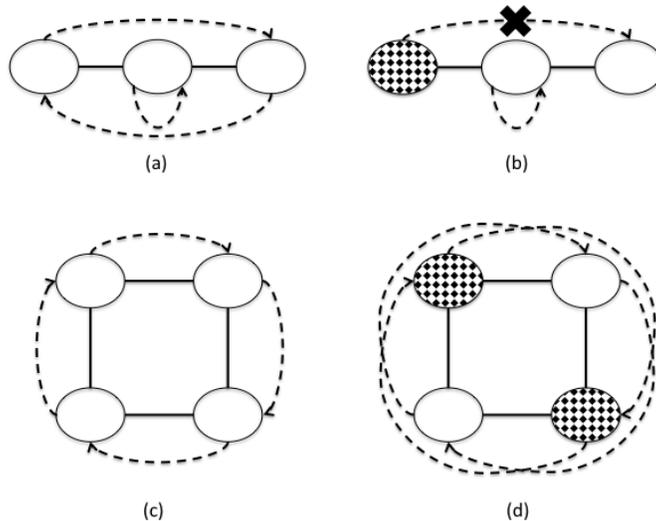
4

Figure 1: *Examples of distinguishability numbers. The line has distinguishability number 2, but the more symmetric square has distinguishability number 3.*

tell if this has happened yet. Urn automata [6], a precursor to the population protocol model in which a finite-state controller manages the population, also have some similarities to graph Turing machines, especially in the combination of a classical Turing-machine controller with an unusual data store.

The **community protocol** model of Guerraoui and Ruppert [7, 8] extends population protocols by allowing agents to store a constant number of pointers to other agents that can only be used in limited ways. Despite these restrictions, Guerraoui and Ruppert show that community protocols with n agents can simulate **storage modification machines** as defined by Schönhage [9], which consist of a dynamic graph on n nodes updated by a finite-state controller. Such machines can in turn simulate standard Turing machines with $O(n \log n)$ space. The community protocol and storage modification machine models are both stronger than our graph Turing machines because they allow for a dynamic graph, while our machines have to work with the graph they are given.

2.3. Distinguishing number

The **distinguishing number** [10] $d(G)$ of a graph G is the minimum number of colors (color is used instead of labels in the literature) needed to color

the vertices of G so that G has no non-trivial color-preserving automorphism. An automorphism is a permutation of the vertices of the graph that preserves adjacency so that two vertices that are adjacent are mapped to two vertices that are adjacent. A color-preserving automorphism is an automorphism that maps vertices to vertices of the same color. The concept is illustrated in Figure 1 in which the mapping is indicated with dashed arrows. In the example of a line, there is a non-trivial color-preserving automorphism if nodes are colored with only one color (a). If we color one of the two endpoints by introducing a second color, there is no non-trivial color preserving automorphism. In the case of the square, two colors are not sufficient. The figure shows the one-coloring (c) as well as one particular two-coloring (d) and the corresponding color-preserving automorphism, but it should be clear that no other two-colorings will eliminate all non-trivial color-preserving automorphsims..

If the distinguishing number of a class of graphs is bounded, then we can in principle color the nodes with a distinguishing coloring that uniquely identifies each node based on its position in the graph (though it still may require substantial work to identify a particular node). With a large enough alphabet, we can use a second component of the state to store the contents of a Turing machine tape cell. This would give an information-theoretic capacity for the graph of $\Theta(n)$.

Albertson and Collins [10] show that any graph has distinguishing number $O(\log(|\mathrm{Aut}(G)|))$, where $\mathrm{Aut}(G)$ is the set of automorphisms of G. This implies that the information-theoretic capacity of the class of graphs with constant-sized automorphism groups is $\Theta(n)$ (the effective capacity may be smaller in some cases). Thus graphs with low information-theoretic capacity will have large automorphism groups, i.e., lots of symmetry.

Computing distinguishing number exactly appears to be difficult. Some improved characterizations may be found in [11, 12].

3. Graph Turing machines

In this section, we define formally our notion of a graph Turing machine.

6

A **graph Turing machine** is specified by a 6-tuple $(\Sigma, Q, q_0, q_{\text{accept}}, q_{\text{reject}}, \delta)$, where Σ is a finite **alphabet** of **tape symbols**; Q is a finite set of **controller states**; $q_0 \in Q$ is the **initial controller state**; $q_{\text{accept}}, q_{\text{reject}} \in Q$ are **halting states**; and $\delta : (Q - \{q_{\text{accept}}, q_{\text{reject}}\}) \times \Sigma \times \mathcal{P}(\Sigma) \to Q \times \Sigma \times \Sigma$ is the **transition function**. We assume that the alphabet Σ contains the special **blank** symbol $-$. The graph G on which the machine runs and the initial position of the controller $v_0 \in V(G)$ are supplied separately.

The first argument of the transition function δ is the current state of the controller, the second argument is the symbol on the current node, and the third gives the set of symbols that appear on one or more of the neighbors of the current node. The output of δ gives the new state of the controller, the symbol to write to the current node, and the symbol indicating which adjacent node to move to.

The special states q_{accept} and q_{reject} are accepting and rejecting halting states, respectively; if the machine enters one of these two halting states, there is no move to a neighboring node. For transitions that do not enter a halting state, we require that the target symbol be present in the immediate neighborhood (i.e., that it is chosen from the set of neighboring symbols). Implicit in this rule is that, in the unusual event that G contains only a single node v_0 and the set of symbols on neighboring nodes is empty, the machine must halt immediately.

A **configuration** of a graph Turing machine $(\Sigma, Q, q_0, q_{\text{accept}}, q_{\text{reject}}, \delta)$ running on a graph G is a triple $(q, v, s) \in Q \times V(G) \times \Sigma^{V(G)}$ where q is the current state of the controller, v is its current position, and s specifies the current tape symbol s_v on each node v of G. A **halting configuration** is a configuration in which the controller state is either q_{accept} or q_{reject}; in the former case it is an **accepting configuration** and in the latter a **rejecting configuration**.

We consider (G, v_0) to be the input to the graph Turing machine, where G is a graph and $v_0 \in V(G)$ is the initial node. Given an input (G, v_0), the **initial configuration** of the machine is $(q_0, v_0, \{-\}^{V(G)})$, i.e., the configuration in which the controller starts on node v_0 in state q_0 and all nodes contain the

blank symbol. As with standard Turing machines, we write $M(G, v_0)$ for the machine M operating on input (G, v_0).

Given a non-halting configuration (q, v, s), let

$$(q', \sigma_1, \sigma_2) = \delta\left(q, s_v, \{s_u : (u, v) \in E(G)\}\right).$$

There is a **transition** from (q, v, s) to (q', v', s') if (a) $s'_v = \sigma_1$, (b) $s'_u = s_u$ for all $u \in V(G) - \{v\}$, and (c) $s'_{v'} = s_{v'} = \sigma_2$. Note that there may be more than one such transition if there is more than one neighbor v' with $s_{v'} = \sigma_2$, Note further that there are no transitions from a halting configuration.

Given input (G, v_0), a **computation path** is a sequence of configurations $C_0, C_1, \ldots$ where C_0 is the initial configuration and there is a transition from C_i to C_{i+1} for each i. A graph Turing machine **halts** on input (G, v_0) if every computation path is finite. A graph Turing machine **accepts** (**rejects**) input (G, v_0) if every computation path is finite and ends in an accepting (rejecting) configuration. The **running time** of a graph Turing machine with input (G, v_0) is the maximum length of any computation path, or ∞ if no such maximum exists.

Though most computations of graph Turing machines are inherently nondeterministic, we will call a graph Turing machine **deterministic** if for any input (G, v_0) it either accepts on all computation paths or rejects on all computation paths. The justification for this unusual usage is that for a deterministic graph Turing machine, the choice of which of several alternative nodes to move to can be made arbitrarily—possibly even according to some deterministic tie-breaking rule (whose inclusion would complicate the model.)

3.1. Simulations

To show that a graph Turing machine with a given input (G, v) has computation power that is no less than that of another machines, it is enough to show how the graph Turing machine can **simulate** the other machine. The other machine can be a standard Turing machine, a graph Turing machine with a different input, or a graph Turing machine with the same input but extended

in some way such as by adding more built-in storage (Section 6.1) or multiple heads (Section 6.2). The particular case of simulating a standard Turing machine will be used to define effective capacity in Section 4.2.

The idea of a simulation is to show how the configurations and transitions of the simulating machine can be mapped to configurations and transitions of the simulated machine so that the simulating machine clearly captures the behavior of the simulated machine. If a machine simulates another machine then it has at least as much computation power. Any computation of the simulated machine can be done by the simulating machine which can accept whenever the simulated configuration is an accepting configuration and reject whenever the simulated configuration is a rejecting configuration.

We say that a graph Turing machine M_1 with input (G_1, v_1) **simulates** a graph Turing machine M_2 with input (G_2, v_2) if there is a mapping from configurations of $M_1(G_1, v_1)$ to configurations of $M_2(G_2, v_2)$ such that two configurations that form a transition in $M_1(G_1, v_1)$ map to either two configurations that form a transition in $M_2(G_2, v_2)$ or to the same configuration of M_2. In other words, the images of two configurations that form a transition in M_1 are two configurations that form a transition in M_2 or the same configuration. Allowing for the same configuration to be repeated in the mapping is called *stuttering*. It is needed because intermediate steps of in the simulating machine do not necessarily reflect change of configuration in the simulated machine.

4. Storage capacity of graphs

In this section, we consider the question of how much information can be stored in a given graph. We first look at the information-theoretic capacity bound (Section 4.1), then consider how much of this potential capacity can actually be extracted (Section 4.2).

4.1. Information-theoretic capacity

The information-theoretic capacity of a graph is just the base 2 logarithm of the number of distinguishable labelings of its nodes, where two labelings

are **distinguishable** if there is no non-trivial automorphism of the graph that carries one to the other and **equivalent** otherwise. This quantity is in principle computable using Burnside's Lemma; the number of distinguishable labelings is

$$L(G) = |X/\mathrm{Aut}(G)| = \frac{1}{|\mathrm{Aut}(G)|} \sum_{g \in \mathrm{Aut}(G)} |X^g|,$$

where X is the set of all labelings, $X/\mathrm{Aut}(G)$ is the quotient set of equivalence classes of labelings under automorphisms in G, and X^g is the set of labelings preserved by a particular automorphism g. The information-theoretic capacity I_G of G is then the base 2 logarithm $\lg L(G)$ of this quantity.

In practice, computing the number of distinguishable labelings will be easiest for classes of graphs that have no non-trivial automorphisms, or for which the set of automorphisms has a particularly simple structure, such as cliques, stars, or trees. For example, any permutation of the nodes of a clique, or any permutation of the non-central nodes of a star, is an automorphism of the graph. We can map one labeling of the clique to another by a color-preserving permutation precisely when each labeling has the same number of nodes with each color. In the case of a star, we can map one labeling to another by color-preserving permutation if the central nodes have the same colors and the labeling has the same number of leaves with each color. It follows that an equivalence class can be specified by counting the number of nodes with each color (plus $O(1)$ bits for the central node for a star). In either case we get $\Theta(\log n)$ bits of information.

Graphs with constant distinguishing number (see Section 2.3), which are those for which a small number of carefully-colored nodes eliminate color-preserving automorphisms, will have information-theoretic capacity $\Theta(n)$. An example would be a path; by fixing distinct colors of the endpoints, no color-preserving automorphisms remain.

The information-theoretic capacity of general trees depends heavily on the structure of the tree: whether it looks more like a star, with many automorphisms, or a path, with few. We discuss this issue in detail in Section 7.

In general, we can bound the information-theoretic capacity of any graph with n nodes by $O(n)$; this is just the number of bits needed to represent all

possible labelings without considering equivalence.

The usefulness of the information-theoretic capacity is that it puts an upper bound on how much state can be stored in the graph. Call two configurations of a graph Turing machine **equivalent** if

1. The head is in the same state in both configurations.

2. There is a label-preserving automorphism of G that carries the position of the head in the first configuration to the position of the head in the second configuration.

It is not hard to see that equivalent states have equivalent successors, since the same automorphism can be used after a transition as long as we are careful to make the heads move to matching locations. It follows that for the purpose of simulating a graph Turing machine, we need only record its state up to equivalence.

Theorem 1. *Fix a graph Turing machine, and suppose it is used to simulate a standard Turing machine. When running on graph G with n nodes and information-theoretic capacity I_G, the simulation has at most $O(I_G)$ space.*

PROOF. We can describe a state of the graph Turing machine up to equivalence by specifying (a) some member of a class of equivalent graph labelings (I_G bits); (b) the state of the finite-state controller ($O(1)$ bits); and (c) the position of the finite-state controller ($\log n$ bits). Summing these quantities gives $O(1) + \log n + I_G$ bits, which translates into at most $O(\log n + I_G)$ tape cells for the simulated machine. But now observe that any graph has $I_G = \Omega(\log n)$ (provided the alphabet size is at least 2), since we can obtain at least $n + 1$ distinct automorphism classes by labeling k nodes with one symbol and $n - k$ with another, where k ranges from 0 to n. So $O(\log n + I_G) = O(I_G)$. $\qquad\square$

4.2. Effective capacity

Our intent is that the **effective capacity** of a graph is the size of the largest standard Turing machine tape that can be simulated using the graph. However, we can in principle make this size arbitrarily large for any fixed graph

by increasing the size of the alphabet and the number of states in the finite-state controller. To avoid this problem, we define effective capacity only for classes of graphs.

Definition 1. A class of graphs $\mathcal{G}$ has effective capacity f, $f : \mathcal{G} \mapsto \mathbb{N}$, if: for any standard Turing machine M, there is a graph Turing machine M' which, for any G in $\mathcal{G}$, and any vertex v_0 of G, $M'(G, v_0)$ simulates M running on an initially blank tape with $f(G)$ tape cells.

Note that because we have not specified alphabet sizes, effective capacity is defined only up to constants. Furthermore, any particular construction can only demonstrate a lower bound on effective capacity. For example, we show in Section 6.1 that the class of all graphs has effective capacity $\Omega(\log n)$, where n is the number of nodes in the graph. The reason for this is that we can use the nodes in the graph as a unary counter, and then use a standard construction [13] to simulate a $\log(n)$-space Turing machine. However, this does not exclude the possibility that some subclass of the class of all graphs has higher effective capacity, or that there might be a construction that obtains $\Omega(\log n)$ space on general graphs while doing better on some specific graphs.

An example of a class of graphs with high effective capacity are paths. The essential idea is that we can use a path directly to simulate a standard Turing machine tape, with each node in the path representing one cell in the tape. A minor complication is that a standard Turing machine can tell its left from its right while a graph Turing machine can only do so if the neighbors of the current cell have different labels. But we can handle this by adding an extra field in each node that holds repeating values $\{0, 1, 2\}$ (this is the slope mechanism from [3]), with the left neighbor of a node with value x being the one with $(x - 1) \bmod 3$ and the right being the one with $(x + 1) \bmod 3$. This extra information triples the size of the alphabet, but that is permitted by the definition.

On the other hand, we can't do any better than $\Theta(n)$. It is immediate from Definition 1 and Theorem 1 that no class of graphs has an effective capacity that exceeds the information-theoretic capacity by more than a constant factor.

Definition 1 also does not include a time bound. A natural restriction would

12

be to consider **polynomially-bounded effective capacity**, where the simulation can use at most a polynomial number of steps for each step of the simulated machine. In our constructions, we are more interested in showing possibility rather than specific time bounds, but we will state time bounds when we can.

5. Graph traversal

A fundamental tool for doing computation with a graph Turing machine is the ability to traverse every node in the graph. In this section, we show how this can be done regardless of the structure of the graph.

We adapt depth-first search to our needs. Depth-first search requires a stack, which we can represent by marking the nodes that are on the stack. Unfortunately, because the graph may contain cycles, a simple mark does not suffice to indicate unambiguously a stack node's parent. If each node were labeled with its distance modulo three from the root, then the correct parent node would be evident from the labels. So, we need a way to label nodes with their distance modulo three from the root. This essentially requires visiting the nodes and labeling them in breadth-first order. Visiting and labeling nodes in breadth-first order can be done in phases. In phase k, we label nodes that are not yet labeled and are adjacent to nodes labeled in phase $k-1$. The expansion of the labeled *frontier* can be achieved if we have a way to visit all nodes in the graph, as in depth-first search. This seems like a circular dependency, so we have to parametrize the traversals. In phase k, we only need to do depth-first traversal of all nodes that were labeled in phase $k-1$ and the nodes that are to be labeled in phase k without visiting any other nodes. The required depth-first traversal in phase k can be done using only the labeling of phase $k-1$, thereby breaking the seeming circularity.

For the depth-first search method, we assume that every node has the fields color (which may be black, white or gray) and depth (which may be ∞, 0, 1 or 2). The depth field is calculated modulo 3 to keep the size of the state finite; this approach is similar to the "centered slope" technique used in [3]. The variable head refers to the node that is the current position of the graph Turing machine.

In the initial configuration in which every node contains the blank symbol, we implicitly have v.depth $= \infty$ and v.color $=$ white for every node v. In order to initiate the process of assigning the depth labels, we explicitly set head.color to white and head.depth to 0 (not shown in the code).

In the depth-first search algorithm the head is initially at the root whose color is white. The color is changed to gray (line 3) and then each of the nodes adjacent to the root is depth-search recursively (line 6). The search at a given node v is **finished** if v.color $=$ black or v.depth $= \infty$. At that point, the color of the node, which must be gray, is changed to black (line 8) and the depth-first search backtracks to the parent node (line 9). If the node that is finished does not have a parent in the search, it must be the root node and the search is terminated (line 10). When the search is terminated, the head is at the root node and all nodes who depth is not infinity have color black. By exchanging the roles of white and black nodes in the last step of the search (line 13) , we effectively reset all nodes from white to black without requiring another traversal of the graph, and ensures that the postcondition of the procedure matches the precondition.

Because depth-first search will be used in the process of establishing the correct depth labels, its invariant refers to C, the portion of the graph that has been successfully depth labeled so far. A node v is in C if and only if v.depth $\neq \infty$. The depth-first search algorithm is a framework to allow the performance of some action at every node of C. The action can occur in pre-order or post-order with respect to the search.

During the depth-first search, a node v is **finished** if v.color $=$ black or v.depth $= \infty$. A per-node action is **safe** if it respects the invariant concerning the depth field, leaves the head where it found it, and alters the color or depth fields of a node only when that node is finished and remains finished after the alteration.

In each phase of the breadth-first algorithm to establish the depth labels we use depth-first search to visit every node v in C and perform the safe per-node operation of changing the depth field of every neighbor w of v with w.depth $= \infty$

14

Invariant: letting $C := \{v \mid v.\mathsf{depth} \neq \infty\}$, the induced subgraph on C is connected, $\mathsf{root} \in C$, and all nodes $v \in C$ have $v.\mathsf{depth} = d(\mathsf{root}, v) \bmod 3$

Precondition: $\mathsf{head} = \mathsf{root}$, and all nodes v with $v.\mathsf{depth} \neq \infty$ have $v.\mathsf{color} = \mathsf{white}$

```
 1  done := false;
 2  repeat
 3      if head.color = white then
 4          head.color := gray;
            // perform a safe per-node action (preorder)
 5      end
 6      if head has a neighbor w with w.color = white and
         w.depth = (head.depth + 1) mod 3 then   head := w;
 7      ;
 8      else
 9          head.color := black;
            // perform a safe per-node action (postorder)
10          if head has a neighbor v with v.color = gray and
             v.depth = (head.depth − 1) mod 3 then   head := v;
11          ;
12          else   done := true;
13          ;
14      end
15  until done;
16  exchange the roles of white and black;
```

Postcondition: the precondition holds

Algorithm 1: Depth-first traversal of a distance-labeled subgraph

to $(v.\mathsf{depth} + 1) \bmod 3$. If at least one node has its depth field modified, then C has expanded, and the next phase of the breadth-first expansion ensues. Otherwise, the process of assigning depth labels to all nodes accessible from the root is complete. We should note here that the breadth first search is written with the assumption that all nodes whose depth is not infinity when the search begins have color white. The depth first search exchanges the black and white colors at the end of the search.

Precondition: all nodes v have $v.\mathsf{depth} = \infty$
1   head.color := white;
2   head.depth := 0;
3   **repeat**
4     $C := \{v \mid v.\mathsf{depth} \neq \infty\}$;
     // C consists of all nodes within distance k of the
        starting node, where k is the number of completed
        iterations
5   **foreach** $v \in C$ **do**
6   **foreach** w *in v's neighbors with* $w.\mathsf{depth} = \infty$ **do**
7         $w.\mathsf{color} := \mathsf{black}$;
8         $w.\mathsf{depth} := (v.\mathsf{depth} + 1) \bmod 3$;
9   **end**
10   **end**
11   **until** *no node w with* $w.\mathsf{depth} = \infty$ *is encountered*;

Algorithm 2: Breadth-first assignment of distance labels for Algorithm 1

To establish the correctness of the depth-first search, we observe that at any time, the induced subgraph on the nodes that are white or gray is connected, and the gray nodes are the nodes along a shortest path from the root to the location of the head or a predecessor of it. Because the head never moves to a black node, it is clear that the only way a node fails to have a white or gray neighbor is if it is the root and it is the last node to be finished. This is the safety property; the liveness property is that we continue to make progress. If we continue to go down, eventually there must be a node with no white or gray children, which will be made black. This in turn establishes the correctness of the breadth-first depth labeling process: at each phase all the currently labeled

16

nodes are visited, and any unlabeled neighbors of them are properly labeled. All accessible nodes are properly depth labeled if and only if no neighbors of labeled nodes are unlabeled.

If the graph G has n nodes and m edges accessible from the root, the depth labeling process completes in $O(n^2)$ steps. Once all accessible nodes are depth labeled, then the depth-first search process can visit every node in $O(m)$ steps.

6. Variants of the model

In this section, we show that the graph Turing machine model is robust against minor changes, including:

- Expanding the store on the controller to $O(\log n)$ bits. (Section 6.1.)

- Replacing the single head with k heads. (Section 6.2.)

- Removing the controller's ability to see adjacent nodes. (Section 6.3.)

These are analogous to classic Turing machine results showing that small changes in the definition do not affect what we can compute.

6.1. Extracting logarithmic space

Using the graph traversal algorithm from Section 5, we can extract $\Theta(\log n)$ space from any graph with n nodes, with a slowdown that is quadratic in n. This simplifies further constructions by letting us replace the finite-state controller in a graph Turing machine with a **LOGSPACE** controller.

All graphs have at least $\Theta(\log n)$ bits of storage available, by ignoring the edge structure and treating the nodes as a unary counter. The following operations can be performed in linear time, with one traversal.

- **Halve the counter value** Traverse the graph, changing every other 1 to a 0.

- **Subtract one** Find a node with bit 1 and change it to 0.

- **Add one** Find a node with bit 0 and change it to 1.

17

- **Test for nonzero** Traverse the graph looking for a 1.

- **Test parity** Traverse the graph, and sum up the values mod 2.

- **Doubling** Repeatedly look for an unmarked node with bit 1, mark it and *add one*. When done, erase all marks.

Following a classic construction of Minksy [13], with halving, doubling, and parity testing, we can implement a pushdown stack of bits (the least-significant bit is the top of the stack, and halving and doubling correspond to pop and push, respectively). Two such stacks provide a bidirectional tape that stores $\Theta(\log n)$ bits. The cost is that we double the size of the alphabet (to store the extra bit on each node) and increase the time for a step by $\Theta(n^2)$.

6.2. Multiple heads

To simulate multiple heads, we include in each node a component indicating which heads are present at that node. A step of the k-head Turing machine is simulated by a single head using two traversals: the first, to collect the states of all the heads and their neighbors; the second, to move all the heads according to the transition function for the k-head machine. This expands the size of the alphabet and requires linear time overhead.

6.3. Limited vision

Our standard graph Turing machine model assumes that the controller can see the states of nodes adjacent to its current position. This simplifies programming and avoids the issue of what happens if the controller attempts to move to a symbol that is not found in the neighborhood. An alternative is to remove the power to see neighbors and have attempts to move to nonexistent symbols fail, leaving the head in the same position.[3]

In this failed-move model, we can nonetheless gather the set of adjacent symbols with a little effort. First mark the current position of the head with a

---

[3]Moving to an arbitrary neighbor may also be a reasonable choice; this can be handled using essentially the same methods.

unique symbol (necessary to allow it to get back without getting lost). Then, for each possible neighboring symbol σ, attempt to move to σ. If the move succeeds, we can add σ to the list of neighbors and return to the head's original position. If it fails, we move on to the next σ. After restoring the original symbol at the head's position, we can then make a move based on the set of neighbors as in the standard model. This adds one symbol to the alphabet and increases the cost of each step by $O(|\Sigma|)$.

7. Trees

The question of whether the information-theoretic capacity of a general family of graphs is necessarily achievable effectively seems to be tied up with the problem of graph isomorphism. For trees, the isomorphism problem is simpler and can be solved in polynomial time [14]. Moreover, it is possible to place a total order on classes of isomorphic trees, which can in turn be used to drive a counter simulation that extracts the full storage capacity of the graph, albeit at the expense of an exponential slowdown introduced by the embedded counter machine simulation.

The canonization algorithm of [14] runs in **LOGSPACE**, so it is tempting to use the **LOGSPACE** simulation of Section 6.1 to execute it directly. Unfortunately, the algorithm assumes that the tree is presented on a read-only work tape using unique identifiers for each node. We don't have this in our model. So instead we describe a new mechanism for comparing labeled trees that works despite this restriction, while allowing us to compute the next labeling of a tree in place. Both the isomorphism and increment procedures run in polynomial time.

Before presenting our mechanism, we consider the information-theoretic capacity of trees.

7.1. Rooted vs unrooted trees

The trees we consider are rooted; this assumption does not affect the asymptotic storage capacity of the tree. Using the same alphabet, a tree with a fixed

root has at most $\lg n$ bits of information more than its unrooted equivalent. The proof is the following.

Let (T, v) be a rooted tree with n nodes. Let X be the set of labelings of (unrooted) T with n nodes. Let $\mathrm{Aut}(T)$ be the automorphism group of T and $\mathrm{Aut}(T)_v$ be the subgroup of automorphisms that fix v. Let X be the set of labelings of T. Then $X/\mathrm{Aut}(T)$ is the set of inequivalent labelings of unrooted T, and $X/\mathrm{Aut}(T)_v$ is the set of inequivalent labelings of T with root v. Since there are at most n nodes to which an automorphism may map v, $\mathrm{Aut}(T)_v$ has at most n cosets in $\mathrm{Aut}(T)$. It follows that $|X/\mathrm{Aut}(T)_v|$ is at most n times as large as $|X/\mathrm{Aut}(T)|$.

7.2. Information-theoretic capacity of a tree

Let T be a tree and let $T_1, T_2, \ldots, T_d$ be the subtrees rooted at the children of the root of T. Let f_k map a rooted tree to the number of inequivalent labelings over an alphabet of size $k \geq 1$; this gives the information-theoretic capacity of the tree. Then we have the recurrence

$$f_k(T) = k \prod_i \binom{c_i + f_k(U_i) - 1}{f_k(U_i) - 1}$$

where $U_1, U_2, \ldots, U_\ell$ are the non-isomorphic classes of the T_js, and c_i is the multiplicity of U_i among the T_js. Here the k counts the number of possible states in the root, and each factor in the product counts the number of ways of choosing a multiset of c_i values out of $f_k(U_i)$ possibilities; this is the number of distinguishable labelings of the i-th equivalence class. The base for the recurrence is provided by a tree with one node, whose root has no children; in this case, the product is empty, leaving just k.

We will show that it is possible to extract exactly $f_k(T)$ states from a rooted tree T, using only a constant amount of additional space at each node beyond the labels of size k. To do so, we show that a graph Turing machine can compute a total ordering on tree labelings and use this ordering build a counter with values $0 \ldots f_k(T) - 1$, by implementing increment and decrement as the successor and predecessor operations of these labelings.

7.3. Comparing trees

We define an ordering on (isomorphism classes of) labeled rooted trees by induction. Let T be a tree with root label ℓ and immediate subtrees $T_1, \ldots, T_c$; let T' be a tree with root label ℓ' and immediate subtrees $T'_1, \ldots, T'_d$. If $\ell < \ell'$, then $T < T'$. If $\ell > \ell'$, then $T > T'$. Otherwise, let $C = \{T_1, \ldots, T_c\}$ and $D = \{T'_1, \ldots, T'_d\}$ be multisets. If $C = D$, then $T = T'$, where subtree equality is defined inductively. If $\max(C - D) > \max(D - C)$, then $T > T'$. Otherwise, $\max(D - C) > \max(C - D)$, and $T < T'$. It is straightforward to verify that this relation $\leq$ on labeled rooted trees is reflexive, transitive, and antisymmetric up to isomorphism.

Algorithm 3 computes this total ordering. We present it in the context of a graph Turing machine with two heads, one on tree T and one on tree T'. Each node has a "removed" bit, which is initially unset. The main invariant is that removing all subtrees whose removed bits are set does not affect the order.

The algorithm begins by comparing the root labels. Assuming that they are equal, for each of the immediate subtrees T_i of T, we attempt to find a immediate subtree of T' isomorphic to T_i. If there is such a subtree T'_j, we set the removed bit for both T_i and T'_j; these subtrees offset one another and will not be considered again. If no match can be made, then we set the removed bit for each T'_j that was determined to be less than T_i, as these subtrees cannot be the maximum in the symmetric difference of the two multisets of subtrees. At the end, if all immediate subtrees have been removed, then $T = T'$. Otherwise, if all immediate subtrees of T' have been removed, then $T > T'$, else $T < T'$.

When testing isomorphism between two top-level trees T and T', each pair of subtrees at the same level are compared at most once; in Algorithm 3, this corresponds to at most a constant amount of work being done for each distinct pair of values of head_1 and head_2. It follows that the total number of steps done by the algorithm is proportional to the product of the sizes of the two trees.

Precondition: all nodes of both trees are distance-labeled and all state values are null

```
 1  done := false;
 2  repeat
 3  │   if head₁.label < head₂.label then  result := LT;
 4  │   ;
 5  │   else if head₁.label > head₂.label then  result := GT;
 6  │   ;
 7  │   else if no child of head₂ has state = null then
    │   │   // all children of head₂ have state = removed
 8  │   │   if all children of head₁ have state = removed then  result := EQ;
 9  │   │   ;
10  │   │   else  result := GT;
11  │   │   ;
12  │   else if no child of head₁ has state = null then  result := LT;
13  │   ;
14  │   else  result := ⊥;
15  │   ;
16  │   if result = ⊥ then
17  │   │   set head₁ to a child of head₁ with state = null;
18  │   │   set head₂ to a child of head₂ with state = null;
19  │   else
20  │   │   set the state of all children of head₁ and head₂ to null;
21  │   │   if head₁ is the root (of tree T) then  done := true;
22  │   │   ;
23  │   │   else if result = EQ then
24  │   │   │   head₁.state = removed;
25  │   │   │   head₁ = head₁.parent;
26  │   │   │   head₂.state = removed;
27  │   │   │   head₂ = head₂.parent;
28  │   │   │   foreach child c of head₂ with c.state ∈ {considered, notMax}
    │   │   │     do
29  │   │   │   │   c.state := null;
30  │   │   │   end
31  │   │   else
32  │   │   │   if result = GT then  head₂.state := notMax;
33  │   │   │   ;
34  │   │   │   else  head₂.state := considered;
35  │   │   │   ;
36  │   │   │   head₂ = head₂.parent;
37  │   │   │   if there exists a child c of head₂ with c.state = null then
38  │   │   │   │   head₂ := c;
39  │   │   │   else
40  │   │   │   │   head₁.state = considered;
41  │   │   │   │   head₁ = head₁.parent;
42  │   │   │   │   foreach child c of head₂ with c.state = considered do
43  │   │   │   │   │   c.state := null;
44  │   │   │   │   end                    22
45  │   │   │   │   foreach child c of head₂ with c.state = notMax do
46  │   │   │   │   │   c.state := removed;
47  │   │   │   │   end
48  │   │   │   end
49  │   │   end
50  │   end
51  until done;
```

7.4. Implementing counters with labeled trees

In this section, we simulate counter machines with operations of clear, increment, and compare counters for equality, which can implement a standard Turing machine, albeit with exponential slowdown [13].

Given a tree T whose nodes can be labeled $0, \ldots, k-1$, we represent counter values $0, \ldots, f_k(T) - 1$ by the labelings they index in the order defined above, where f_k is the count of isomorphism classes of labelings defined in Section 7.2. Zero corresponds to the all-zeros labeling, so clearing a register can be accomplished with one traversal. Two counters whose underlying unlabeled trees are isomorphic can be compared using the isomorphism algorithm. The increment operation requires a new algorithm, which appears as Algorithm 4.

We assume that each node of the tree has space to save its old label, and the increment routine will save the previous counter value. To increment a subtree T with overflow, first save the root label and then increment its immediate subtrees (saving their respective values). If every subtree overflowed (i.e., is zero), then increment the root label mod k and overflow if it becomes zero. Otherwise, use the isomorphism checker to find and mark the minimum nonzero immediate subtree M. Restore every tree other than M to its original value and then zero those that are less than M. The call stack for this recursive algorithm is represented using a state label at each node of the tree.

We prove the correctness of the increment algorithm by induction. Suppose we are incrementing a labeled tree T with immediate subtrees $T_1 \geq \ldots \geq T_c$. Let T' be the resulting tree. In case $T_1, \ldots, T_c$ are already at their respective maximums, it is straightforward to verify that T' is the successor of T. Otherwise, let U be any relabeling of T such that $U > T$. We show that $T < T' \leq U$ and thus that T' is the successor of T.

Let $T'_1 \geq \cdots \geq T'_c$ be the immediate subtrees of T' and let $U_1 \geq \cdots \geq U_c$ be the immediate subtrees of U. Note that, on account of the labelings, T'_i (respectively U_i) may not correspond to T_i. Given that some T_i is not maximum, then the root labels of T and T' are identical, and the algorithm is able to find a minimum incremented tree T'_j, where we choose j to be as large as possible

```
 1  procedure increment(T)
 2  │   T.oldLabel := T.label
 3  │   foreach child T' of T do
 4  │   │   increment(T')
 5  │   end
 6  │   if every child T' of T returned overflow then
 7  │   │   T.label := T.label + 1 (mod k) if T.label = 0 then
 8  │   │   │   return overflow
 9  │   │   end
10  │   else
11  │   │   Using Algorithm 3, find minimum nonzero M among children of
    │   │      T.
12  │   │   foreach child T' of T do
    │   │   │   // restore old labels
13  │   │   │   foreach i in T' do
14  │   │   │   │   i.label := i.oldLabel
15  │   │   │   end
    │   │   │   // compare using Algorithm 3
16  │   │   │   if T' < M then
    │   │   │   │   // set tree value to 0
17  │   │   │   │   foreach i in T' do
18  │   │   │   │   │   i.label := 0
19  │   │   │   │   end
20  │   │   │   end
21  │   │   end
22  │   end
23  end
```

Algorithm 4: Counter increment

in case of ties. We have $T_i = T_i'$ if $i < j$, and $T_j < T_j'$ (count the number of trees greater than or equal to T_j'). For all $i > j$, the tree T_i' is zero. If the root label of U is not equal to the root label of T then $T' < U$. Otherwise, let ℓ be the least index for which $T_\ell < U_\ell$. For all $i < \ell$, we have $U_i = T_i$. If $\ell < j$, then $U > T'$. Otherwise, U_ℓ has the same shape (disregarding labels) as some tree $T_i < T_j'$. Since T_j' was the minimum increment, it follows that $T_j' \leq U_\ell$ and thus that $T' \leq U$.

8. Capacity of a bounded-degree graph

For a bounded-degree graph, we can use the mechanism in [5] (which itself derives much of its structure from the previous construction in [3]) with only a few small modifications.

In a graph with degree bound Δ, it is possible to assign each node a label in $\{1, \ldots, \Delta^2 + 1\}$ so that each node's label is unique within a ball of radius 2. This is a **distance two labeling**, and it gives each node the ability to identify its neighbors uniquely. Angluin *et al.* [5] construct a distance two labeling non-deterministically, by having each node adopt a new label if it detects a second-order neighbor with the same label. In our model, we can construct the labeling deterministically, by iteratively assigning each node a label that does not conflict with a second-order neighbor (it is easy to see that each time we do this, we cannot create any new conflicts, so we converge after $O(n)$ iterations to a correct labeling).

Using such a labeling, it is straightforward to adapt the traversal routine to build a spanning tree, which in turn can simulate a Turing machine tape to provide $\Theta(n)$ bits of effective capacity.

9. Random graphs

Suppose we consider random graphs G drawn from $G(n, p)$, i.e., a graph on n nodes where each edge uv appears with probability p.[4] Suppose further that

---

[4]See [15, 16] for an overview of random graphs.

p scales as $\Theta(n^{-c})$ for some fixed $0 < c < 1$. Then it is possible to achieve an effective capacity of $\Theta(n)$ with high probability[5] from graphs in this class. Note that graphs in this class are connected with high probability.

The basic idea is that if we can compute a total order on nodes, we can use each node to hold one Turing machine cell, with left and right movements corresponding to moving down or up in the ordering. We compute this ordering by assigning a signature to each node, based on random values stored in its neighborhood. For simplicity, we assume that the Turing machine simulator can generate random values.

We start with a basic lemma:

Lemma 2. *Let u and v be nodes in $G(n,p)$, where $p = \Theta(n^{-c})$ for fixed $0 < c < 1$. Then with high probability, for any two nodes u and v, the set $\delta(u) - \delta(v)$ of nodes w that are adjacent to u but not v has has size $\Omega\left(n^{1-c}\right)$.*

PROOF. The probability that w is adjacent to u but not v is exactly $p(1-p) = \Theta(n^{-c}) \geq \alpha n^{-c}$ for some α. Summing over all possible w to obtain a total count X gives $\mathrm{E}[X] \geq \alpha(n-2)n^{-c}$. These events are independent, so Chernoff bounds (e.g., [16, Theorem 2.1]) give

$$\Pr\left[X \leq \frac{1}{2}\,\mathrm{E}[X]\right] \leq \exp\left(-\frac{\left(\frac{1}{2}\,\mathrm{E}[X]\right))^2}{2\,\mathrm{E}[X]}\right)$$
$$= \exp\left(-\,\mathrm{E}[X]/8\right)$$
$$= \exp\left(-\Theta\left(n^{1-c}\right)\right).$$

Since this holds with high probability for each individual pair u, v, summing over all u and v shows that it holds with high probability for all u and v. $\qquad\square$

Suppose now that each node w holds a random bit r_w that is 0 or 1 with equal probability, and we count the number of one bits in $\delta(u)$; to avoid ugly summations, we will write this as $r_{\delta(u)}$. Then

---

[5]We use **with high probability** to mean that the probability that the event occurs is $1 - O(n^{-c})$ for any fixed c; this implies the weaker condition that the probability that the even occurs goes to 1 in the limit as n grows.

Lemma 3.

$$\Pr\left[r_{\delta(u)} = r_{\delta(v)}\right] = O\left(n^{(c-1)/2}\right).$$

PROOF. Observe that

$$\Pr\left[r_{\delta(u)} = r_{\delta(v)}\right] = \Pr\left[r_{\delta(u)} - r_{\delta(u)\cap\delta(v)} = r_{\delta(v)} - r_{\delta(u)\cap\delta(v)}\right]$$

$$= \Pr\left[r_{\delta(u)-\delta(v)} = r_{\delta(v)-\delta(u)}\right].$$

These last two quantities are independent binomial random variables obtained by summing $\Omega\left(n^{1-c}\right)$ bits each (from Lemma 2). For any fixed value of $r_{\delta(u)-\delta(v)}$, the probability that $r_{\delta(v)-\delta(u)}$ takes the same value is $O\left(1/\sqrt{n^{1-c}}\right) = O\left(n^{(c-1)/2}\right)$. $\square$

Now repeat the construction in parallel k times using k independent random bits per node to obtain a probability of $O\left(n^{k(c-1)/2}\right)$. For any fixed $c < 1$, there is a constant k such that this probability for any particular pair is bounded by n^{-3} and the probability that we get identical values summed over all pairs is bounded by $\binom{n}{2}n^{-3} \leq n^{-1}$. So assuming the condition in Lemma 2 holds, most random k-bit labelings $r^1, r^2, \ldots, r^k$ will give a unique signature $r^1_{\delta(u)}, \ldots r^k_{\delta(u)}$ for each node u.

It is easy to see that a **LOGSPACE** controller can compute these signatures and order nodes by signature. This implies that a **LOGSPACE** controller can also detect if two nodes have the same signatures, and generate new random bit values until this condition no longer holds. It follows that we can obtain $\Theta(n)$ effective capacity from any family of graphs that satisfies the condition of Lemma 2, and thus have:

Theorem 4. *A member of the family of random graphs $G(n,p)$ where $p = \Theta(n^{-c})$ for any fixed $0 < c < 1$ has effective capacity $\Theta(n)$ with high probability.*

This result depends on being able to assign random values to each node, which requires extending the basic graph Turing machine model to include randomization in the obvious way. We suspect that this is not necessary to obtain

the result, and that local structural properties of the graph (for example, the degrees of each node) could substitute for the random bits used in the construction. The difficulty that arises is that such structural properties will not in general be independent, and as our graph Turing machine will always be working in some fixed graph, if the properties happen to lead to many duplicate signatures there is not much the machine can do about it. We leave the question of the deterministic effective capacity of random graphs to future work.

10. Conclusion

We have defined a new class of graph-based Turing machines, motivated by potential applications in self-organizing systems of finite-state automata. We have shown that this class is robust under natural changes to the model, and that its power is primarily characterized by the effective capacity of the underlying graph, which is the amount of usable storage obtained by writing symbols from a finite alphabet on its nodes. This is at least $\Omega(\log n)$ bits of space for an arbitrary n-node graph, and rises to $\Theta(n)$ bits for bounded-degree graphs and almost all random graphs with polynomial edge probabilities. For trees, the effective capacity ranges from $\Theta(\log n)$ for trees with many symmetries (stars) to $\Theta(n)$ for trees with few (binary trees, paths). In intermediate cases we have shown that we can always get within a constant factor of the full information-theoretic capacity corresponding to the number of non-isomorphic states, although the time complexity of our algorithm could be significantly improved.

The main open problem remaining is whether it is possible to extract the full information-theoretic capacity from an arbitrary graph. This seems closely tied to the problem of computing graph isomorphism, which is not known to be hard, even for **LOGSPACE**. The reason is that distinguishing two different labelings of a graph appears to depend on being able to distinguish between non-isomorphic subgraphs (since this gives a weak form of orientation to the graph). However, the problem is not exactly the same, because we have the ability to supplemental isomorphism testing by using some of our labels as signposts and

we do not need a perfect isomorphism tester as long as we can group subgraphs into small equivalence classes. So it may be that extracting the full capacity of an arbitrary graph is possible without solving graph isomorphism in general.

11. Acknowledgments

References

[1] E. W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM 17 (11) (1974) 643–644.

[2] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, R. Peralta, Computation in networks of passively mobile finite-state sensors, Distributed Computing 18 (4) (2006) 235–253.

[3] G. Itkis, L. A. Levin, Fast and lean self-stabilizing asynchronous protocols, in: Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 226–239.

[4] J. Aspnes, E. Ruppert, An introduction to population protocols, in: B. Garbinato, H. Miranda, L. Rodrigues (Eds.), Middleware for Network Eccentric and Mobile Applications, Springer-Verlag, 2009, pp. 97–120.

[5] D. Angluin, J. Aspnes, M. Chan, M. J. Fischer, H. Jiang, R. Peralta, Stably computable properties of network graphs, in: Proc. Distributed Computing in Sensor Systems: 1st IEEE International Conference, 2005, pp. 63–74.

[6] D. Angluin, J. Aspnes, Z. Diamadi, M. J. Fischer, R. Peralta, Urn automata, Tech. Rep. YALEU/DCS/TR-1280, Yale University Department of Computer Science (Nov. 2003).

[7] R. Guerraoui, E. Ruppert, Even small birds are unique: Population protocols with identifiers, Tech. Rep. CSE-2007-04, Department of Computer Science and Engineering, York University (2007).

[8] R. Guerraoui, E. Ruppert, Names trump malice: Tiny mobile agents can tolerate byzantine failures, in: S. Albers, A. Marchetti-Spaccamela, Y. Matias, S. E. Nikoletseas, W. Thomas (Eds.), ICALP (2), Vol. 5556 of Lecture Notes in Computer Science, Springer, 2009, pp. 484–495.

[9] A. Schönhage, Storage modification machines, SIAM J. Comput. 9 (3) (1980) 490–508.

[10] M. O. Albertson, K. L. Collins, Symmetry breaking in graphs, Electronic Journal of Combinatorics 3 (1) (1996) R18.

[11] M. O. Albertson, Distinguishing Cartesian powers of graphs, Electronic Journal of Combinatorics 12 (2005) N17.

[12] A. Russell, R. Sundaram, A note on the asymptotics and computational complexity of graph distinguishability, Electronic Journal of Combinatorics 5 (1) (1998) R23.

[13] M. L. Minsky, Computation: Finite and infinite machines, Prentice-Hall series in automatic computation, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1967.

[14] S. Lindell, A logspace algorithm for tree canonization (extended abstract), in: STOC, ACM, 1992, pp. 400–404.

[15] B. Bollobás, Random Graphs, 2nd Edition, Cambridge University Press, 2001.

[16] S. Janson, T. Łuczak, A. Ruciński, Random Graphs, John Wiley & Sons, 2000.