On the computability of graph Turing machines

Nathanael Ackerman
Harvard University
Cambridge, MA 02138, USA
nate@math.harvard.edu

Cameron Freer
Remine
Falls Church, VA 22043, USA
cameron@remine.com

Abstract

We consider *graph Turing machines*, a model of parallel computation on a graph, in which each vertex is only capable of performing one of a finite number of operations. This model of computation is a natural generalization of several well-studied notions of computation, including ordinary Turing machines, cellular automata, and parallel graph dynamical systems. We analyze the power of computations that can take place in this model, both in terms of the degrees of computability of the functions that can be computed, and the time and space resources needed to carry out these computations. We further show that properties of the underlying graph have significant consequences for the power of computation thereby obtained. In particular, we show that every arithmetically definable set can be computed by a graph Turing machine in constant time, and that every computably enumerable Turing degree can be computed in constant time and linear space by a graph Turing machine whose underlying graph has finite degree.

1 Introduction

When studying large networks, it is important to understand what sorts of computations can be performed in a distributed way on a given network. In particular, it is natural to consider the setting where each node acts independently in parallel, and where the network is specified separately from the computation to be performed. In order to study networks whose size is considerably larger than can be held in memory by the computational unit at any single node, it is often useful to model the network as an infinite graph. (For a discussion of modeling large networks via infinite graphs, see, e.g., [Lov09].)

We define a notion of *graph Turing machine* that is meant to capture this setting. This notion generalizes several other well-known models of computation, including ordinary Turing machines, cellular automata, and parallel graph dynamical systems. Each of these models, in turn, occurs straightforwardly as a special case of a graph Turing machine, suggesting that graph Turing machines capture a natural concept of parallel computation on graphs.

A graph Turing machine (henceforth abbreviated as "graph machine") performs computation on a vertex-labeled edge-colored directed multigraph satisfying certain properties. This notion of computation is designed to capture the idea that in each timestep, every vertex performs a limited amount of computation (in parallel, independently of the other vertices), and can only distinguish vertices connected to it when they are connected by different colors of edges.

In this paper we study the functions that can be computed using graph machines, which we call *graph computable* functions. As we will see, this parallel notion of computation will yield significantly greater computational strength than ordinary Turing machines, even when we impose

March 28, 2017

constraints on the time and space resources allowed for the graph computation, or when we require the underlying graph to be efficiently computable. We will see that the computational strength of graph machines is exactly that of $\mathbf{0}^{(\omega)}$, the Turing degree of true arithmetic (thereby providing another natural construction of this degree). We also examine the relationship between various properties of the underlying graph (e.g., finiteness of degree) and the computational strength of the resulting graph machines.

1.1 Main results and overview of the paper

We begin by introducing the notions of colored graphs, graph machines, and graph computability (including resource-bounded variants) in Section 2.

Our main results fall into two classes: bounds on the computational power of arbitrary computable graph machines, and bounds among machines with an underlying graph every vertex of which has finite degree (which we say is of *finite degree*).

Theorem 3.6 states that every graph computable function is Turing reducible to $\mathbf{0}^{(\omega)}$. In the other direction, we show in Theorem 3.10 that this bound is attained by a single graph Turing machine.

Sitting below $\mathbf{0}^{(\omega)}$ are the arithmetical Turing degrees, i.e., those less than $\mathbf{0}^{(n)}$ for some $n \in \mathbb{N}$, where $\mathbf{0}^{(n)}$ denotes the n-fold iterate of the halting problem. We show in Corollary 3.9 that every arithmetical Turing degree contains a function that is graph Turing computable, moreover in constant time. (It remains open whether every degree below $\mathbf{0}^{(\omega)}$ can be achieved.)

We next show in Corollary 4.8 that functions determined by graph machines with underlying graph of finite degree are reducible to the halting problem, $\mathbf{0}'$. Further, we show in Corollary 4.9 that if we restrict to graph machines where every vertex has the same (finite) degree, then the resulting graph computable function is computable by an ordinary Turing machine.

We also show in Theorem 4.12 that every Turing degree below $\mathbf{0}'$ is the degree of some linear-space graph computable function with underlying graph of finite degree. When the Turing degree is k-computably enumerable (for some $k \in \mathbb{N}$) then we may further take the graph machine to run in constant time.

In Section 5, we sketch two properties of the graph machine or its underlying graph that do not restrict which functions are graph computable. We show, in Proposition 5.2, that we may take the graph machine to be itself efficiently computable, without changing the degree of functions thereby computed. We also show, in Proposition 5.4, that the requirement that the graph be directed adds no generality: any function which can be computed by a graph machine can be computed by a graph machine whose underlying graph is symmetric.

In Section 6, we examine how several other models of computation relate to graph machines. A graph Turing machine can be thought of as a generalization of an ordinary Turing machine, where the one-dimensional read-write tape is replaced by an arbitrary graph. In §6.1, we describe how to simulate an ordinary Turing machine via a graph Turing machine. One of the fundamental difficulties when generalizing ordinary Turing machines to graph Turing machines is to figure out how to determine the location of the head. We have taken an approach whereby there is no unique head, and hence each node processes its own data. The approach taken by [AAB+14] is to have a unique head, but to allow its location to be nondeterministic, in that it is allowed to move to any vertex connected to the current location that is displaying an appropriate symbol. (Note that they call their different notion a "graph Turing machine" as well.)

Our graph machines can also be viewed directly as dynamical systems, or as a sort of network computation. Cellular automata can be simulated by graph machines, as we show in §6.2, and parallel graph dynamical systems are essentially equivalent to the finite case of graph machines (see §6.3). Indeed, parallel graph dynamical systems include the above case of cellular automata, and also boolean networks and other notions of network computation, as described in [AMV15b, §2.2].

We conclude with Section 7 on possible extensions of graph machines, and Section 8 on open questions.

1.2 Notation

When $f\colon A \to B$ is a partial function and $a \in A$, we let $f(a)\uparrow$ signify that f is not defined at a, and $f(a)\downarrow$ signify that $f(a)$ is defined at a. Suppose $f, g\colon A \to B$ are partial functions. For $a \in A$ we say that $f(a) \cong g(a)$ if either ($f(a)\uparrow$ and $g(a)\uparrow$) or ($f(a)\downarrow$, $g(a)\downarrow$, and $f(a) = g(a)$). We say that $f \cong g$ when $(\forall a \in A)\ f(a) \cong g(a)$. If $f\colon A \to \prod_{i \leq n} B_i$ and $k \leq n \in \mathbb{N}$, then we let $f_{[k]}\colon A \to B_k$ be the composition of f with the projection map onto the k'th coordinate.

Fix an enumeration of computable partial functions, and for $e \in \mathbb{N}$, let $\{e\}$ be the e'th such function in this list. If X and Y are sets with $0 \in Y$, let $Y^{<X} = \{\eta\colon X \to Y \ :\ |\{a\ :\ \eta(a) \neq 0\}| < \omega\}$, i.e., the collection of functions from X to Y for which all but finitely many inputs yield 0. (Note that by this notation we do *not* mean partial functions from X to Y supported on fewer than $|X|$-many elements.) For a set X, let $\mathfrak{P}_{<\omega}(X)$ denote the collection of finite subsets of X. Note that the map which takes a subset of X to its characteristic function is a bijection between $\mathfrak{P}_{<\omega}(X)$ and $\{0,1\}^{<X}$.

When working with computable graphs, sometimes the underlying set of the graph will be a finite coproduct of finite sets and $\mathbb{N}^k$ for $k \in \mathbb{N}$. The standard notion of computability for $\mathbb{N}$ transfers naturally to such settings, making implicit use of the computable bijections between $\mathbb{N}^k$ and $\mathbb{N}$, and between $\coprod_{i \leq k} \mathbb{N}$ and $\mathbb{N}$, for $k \in \mathbb{N}$. We will sometimes say *computable set* to refer to some computable subset (with respect to these bijections) of such a finite coproduct X, and *computable function* to refer to a computable function having domain and codomain of that form or of the form $F^{<X}$ for some finite set F.

For sets $X, Y \subseteq \mathbb{N}$, we write $X \leq_{\mathrm{T}} Y$ when X is Turing reducible to Y (and similarly for functions and other computably presented countable objects).

In several places we make use of (ordinary) Turing machines, described in terms of their state space, transition function, and alphabet of symbols. For more details on results and notation in computability theory, see [Soa87].

2 Graph computing

In this section we will make precise what we mean by graph Turing machines as well as graph computable functions.

Definition 2.1. *A* **colored graph** *is a tuple $\mathcal{G}$ of the form $(G, (L, V), (C, E), \gamma)$ where*

- *G is a set, called the* **underlying set** *or the* **set of vertices***,*
- *L is a set called the* **set of labels** *and $V\colon G \to L$ is the* **labeling function***.*
- *C is a set called the* **set of colors** *and $E\colon G \times G \to \mathfrak{P}_{<\omega}(C)$ is called the* **edge coloring**
- *$\gamma\colon L \to \mathfrak{P}_{<\omega}(C)$ is called the* **allowable colors function** *and satisfies*

$$(\forall v, w \in G)\ E(v, w) \subseteq \gamma(V(v)) \cap \gamma(V(w)).$$

A **computable colored graph** *is a colored graph along with indices witnessing that G, L, and C are computable sets and that V,E, and γ are computable functions.*

The intuition is that a colored graph is an edge-colored directed multigraph where each vertex is assigned a label, and such that among the edges between any two vertices, there is at most one edge of each color, and only finitely many colors appear (which must be among those given by γ applied to the label). Eventually, we will allow each vertex to do some fixed finite amount of computation, and we will want vertices with the same label to perform the same computations.

For the rest of the paper by a *graph* we will always mean a colored graph, and will generally write the symbol $\mathcal{G}$ (possibly decorated) to refer to graphs.

Definition 2.2. *A graph $\mathcal{G}$ is* **finitary** *when its set of labels is finite.*

For a finitary graph, without loss of generality one may further assume that the set of colors is finite and that every color is allowed for every label.

Notice that graphs, as we have defined them, are allowed to have infinitely many labels and edge colors, so long as the edges connecting to any particular vertex are assigned a finite number of colors, depending only on the label. However, there is little harm in the reader assuming that the graph is finitary.

The main thing lost in such an assumption is the strength of various results providing upper bounds on the functions that can be computed using graph machines, as we take care to achieve our lower bounds (Corollary 3.9, Theorem 3.10, and Theorem 4.12) using finitary graphs.

Let $\mathcal{G}$ be a graph with underlying set G, and suppose $A \subseteq G$. Define $\mathcal{G}|_A$ to be the graph with underlying set A having the same set of labels and set of colors as $\mathcal{G}$, such that the labeling function, edge coloring function, and allowable colors function of $\mathcal{G}|_A$ are the respective restrictions to A.

Definition 2.3. *A* **graph Turing machine***, or simply* **graph machine***, is a tuple $\mathfrak{M} = (\mathcal{G}, (\mathfrak{A}, \{0,1\}), (S, s, \alpha), T)$ where*

- *$\mathcal{G} = (G, (L, V), (C, E), \gamma)$ is a graph, called the* **underlying graph***. We will speak of the components of the underlying graph as if they were components of the graph machine itself. For example, we will call G the* underlying set *of $\mathfrak{M}$ as well as of $\mathcal{G}$.*
- *$\mathfrak{A}$ is a finite set, called the* **alphabet***, having distinguished symbols 0 and 1.*
- *S is a countable set called the* **collection of states***.*
- *$\alpha \colon L \to \mathfrak{P}_{<\omega}(S)$ is the* **state assignment function***.*
- *s is a distinguished state, called the* **initial state***, such that $s \in \alpha(\ell)$ for all $\ell \in L$.*
- *$T \colon L \times \mathfrak{P}_{<\omega}(C) \times \mathfrak{A} \times S \to \mathfrak{P}_{<\omega}(C) \times \mathfrak{A} \times S$ is a function, called the* **lookup table***, such that for each $\ell \in L$ and each $z \in \mathfrak{A}$,*
    - *if $c \not\subseteq \gamma(\ell)$ or $t \notin \alpha(\ell)$, then $T(\ell, c, z, t) = (c, z, t)$, i.e., whenever the inputs to the lookup table are not compatible with the structure of the graph machine, the machine acts trivially, and*
    - *if $t \in \alpha(\ell)$ and $c \subseteq \gamma(\ell)$, then $T_{[1]}(\ell, c, z, t) \subseteq \gamma(\ell)$ and $T_{[3]}(\ell, c, z, t) \in \alpha(\ell)$, i.e., whenever the inputs are compatible with the structure, so are the outputs.*

  Further, for all $\ell \in L$, we have $T(\ell, \emptyset, 0, s) = (\emptyset, 0, s)$, i.e., if any vertex is in the initial state, currently displays 0, and has received no pulses, then that vertex doesn't do anything in the next step. This lookup table can be thought of as specifying a transition function.

*A $\mathcal{G}$-***Turing machine** *(or simply a $\mathcal{G}$-***machine***) is a graph machine with underlying graph $\mathcal{G}$.*

A **computable graph machine** *is a graph machine along with indices witnessing that $\mathcal{G}$ is a computable graph, S is computable set, and α and T are computable functions.*

If A is a subset of the underlying set of $\mathfrak{M}$, then $\mathfrak{M}|_A$ is the graph machine with underlying graph $\mathcal{G}|_A$ having the same alphabet, states, and lookup table as $\mathfrak{M}$.

The intuition is that a graph machine should consist of a graph where at each timestep, every vertex is assigned a state and an element of the alphabet, which it displays. To figure out how these assignments are updated over time, we apply the transition function determined by the lookup table which tells us, given the label of a vertex, its current state, and its currently displayed symbol, along with the colored *pulses* the vertex has most recently received, what state to set the vertex to, what symbol to display next, and what colored pulses to send to its neighbors.

For the rest of the paper, $\mathfrak{M} = (\mathcal{G}, (\mathfrak{A}, \{0,1\}), (S, s, \alpha), T)$ will denote a *computable* graph machine whose underlying (computable) graph is $\mathcal{G} = (G, (L, V), (C, E), \gamma)$.

Definition 2.4. *A* **valid configuration** *of $\mathfrak{M}$ is a function $f \colon G \to \mathfrak{P}_{<\omega}(C) \times \mathfrak{A} \times S$ such that for all $v \in G$, we have*

- $f_{[1]}(v) \subseteq \gamma(V(v))$ *and*
- $f_{[3]}(v) \in \alpha(V(v))$.

In other words, a valid configuration is an assignment of colors, labels, and states that is consistent with the underlying structure of the graph machine, in the sense that the pulses received and the underlying state at each vertex are consistent with what its allowable colors function and state assignment function permit.

Definition 2.5. *A* **starting configuration** *of $\mathfrak{M}$ is a function $f \colon G \to \mathfrak{P}_{<\omega}(C) \times \mathfrak{A} \times S$ such that*

- $(\forall v \in G)\ f_{[1]}(v) = \emptyset$,
- $f_{[2]} \in \mathfrak{A}^{<G}$, *and*
- $(\forall v \in G)\ f_{[3]}(v) = s$.

We say that f is **supported** *on a finite set $A \subseteq G$ if $f_{[2]}(v) = 0$ for all $v \in G \setminus A$.*

Note that any starting configuration of $\mathfrak{M}$ is always a valid configuration of $\mathfrak{M}$.

In other words, a starting configuration is an assignment in which all vertices are in the initial state s, no pulses have been sent, and only finitely many vertices display a non-zero symbol.

Note that if A is a subset of the underlying set of $\mathfrak{M}$ and f is a valid configuration for $\mathfrak{M}$, then $f|_A$ is also a valid configuration for $\mathfrak{M}|_A$. Similarly, if f is a starting configuration for $\mathfrak{M}$, then $f|_A$ is a starting configuration for $\mathfrak{M}|_A$.

Definition 2.6. *Given a valid configuration f for $\mathfrak{M}$, the* **run** *of $\mathfrak{M}$ on f is the function $\langle \mathfrak{M}, f \rangle \colon G \times \mathbb{N} \to \mathfrak{P}_{<\omega}(C) \times \mathfrak{A} \times S$ satisfying, for all $v \in G$,*

- $\langle \mathfrak{M}, f \rangle(v, 0) = f(v)$ *and*
- $\langle \mathfrak{M}, f \rangle(v, n+1) = T(V(v), X, z, t)$ *for all $n \in \mathbb{N}$, where*
  - $z = \langle \mathfrak{M}, f \rangle_{[2]}(v, n)$ *and* $t = \langle \mathfrak{M}, f \rangle_{[3]}(v, n)$, *and*
  - $X = \bigcup_{w \in G} \big(E(w, v) \cap \langle \mathfrak{M}, f \rangle_{[1]}(w, n)\big)$.

We say that a run **halts at stage** *n if $\langle \mathfrak{M}, f \rangle(v, n) = \langle \mathfrak{M}, f \rangle(v, n+1)$ for all $v \in G$.*

A run of a graph machine is the function which takes a valid configuration for the graph machine and a natural number n, and returns the result of letting the graph machine process the valid configuration for n-many timesteps.

The following lemma is immediate from Definition 2.6.

Lemma 2.7. *Suppose f is a valid configuration for $\mathfrak{M}$. For $n \in \mathbb{N}$, define $f_n := \langle \mathfrak{M}, f \rangle(\,\cdot\,, n)$. Then the following hold.*

- *For all $n \in \mathbb{N}$, the function f_n is a valid configuration for $\mathfrak{M}$.*
- *For all $n, m \in \mathbb{N}$ and $v \in G$,*

$$f_{n+m}(v) = \langle \mathfrak{M}, f_n \rangle(v, m) = \langle \mathfrak{M}, f \rangle(v, n+m).$$

We now describe how a graph machine defines a function.

Definition 2.8. *For $x \in \mathfrak{A}^{<G}$, let $\widehat{x}$ be the valid configuration such that $\widehat{x}(v) = (\emptyset, x(v), s)$ for all $v \in G$. Define*

$$\{\mathfrak{M}\} \colon \mathfrak{A}^{<G} \to \mathfrak{A}^G$$

to be the partial function such that

- *$\{\mathfrak{M}\}(x)\uparrow$, i.e., is undefined, if the run $\langle \mathfrak{M}, \widehat{x} \rangle$ does not halt, and*
- *$\{\mathfrak{M}\}(x) = y$ if $\langle T, \widehat{x} \rangle$ halts at stage n and for all $v \in G$,*

$$y(v) = \langle \mathfrak{M}, \widehat{x} \rangle_{[2]}(v, n).$$

Note that $\{\mathfrak{M}\}(x)$ is well defined as $\widehat{x}$ is always a starting configuration for $\mathfrak{M}$.

While in general, the output of $\{\mathfrak{M}\}(x)$ might have infinitely many non-zero elements, for purposes of considering which Turing degrees are graph computable, we will mainly be interested in the case where $\{\mathfrak{M}\}(x) \in \mathfrak{A}^{<G}$, i.e., when all but finitely many elements of G take the value 0.

When defining a function using a graph machine, it will often be convenient to have extra vertices whose labels don't affect the function being defined, but whose presence allows for a simpler definition. These extra vertices can be thought as "scratch paper" and play the role of extra tapes (beyond the main input/output tape) in a multi-tape Turing machine. We now make this precise.

Definition 2.9. *Let X be an infinite computable subset of G. A function $\zeta \colon \mathfrak{A}^{<X} \to \mathfrak{A}^X$ is* **$\langle \mathcal{G}, X \rangle$-computable** *via $\mathfrak{M}$ if*

(a) $\{\mathfrak{M}\}$ is total,

(b) for $x, y \in \mathfrak{A}^{<G}$, if $x|_X = y|_X$ then $\{\mathfrak{M}\}(x) = \{\mathfrak{M}\}(y)$, and

(c) for all $x \in \mathfrak{A}^{<G}$, for all $v \in G \setminus X$, we have $\{\mathfrak{M}\}(x)(v) = 0$, i.e., when $\{\mathfrak{M}\}(x)$ halts, v displays 0, and

(d) for all $x \in \mathfrak{A}^{<G}$, we have $\{\mathfrak{M}\}(x)|_X = \zeta(x|_X)$.

A function is **$\mathcal{G}$-computable via $\mathfrak{M}$** *if it is $\langle \mathcal{G}, X \rangle$-computable via $\mathfrak{M}$ for some infinite computable $X \subseteq G$. A function is* **$\mathcal{G}$-computable** *if it is $\mathcal{G}$-computable via $\mathfrak{M}^\circ$ for some computable $\mathcal{G}$-machine $\mathfrak{M}^\circ$. A function is* **graph Turing computable**, *or simply* **graph computable**, *when it is $\mathcal{G}^\circ$-computable for some computable graph $\mathcal{G}^\circ$.*

The following lemma captures the sense in which functions that are $\langle \mathcal{G}, X \rangle$-computable via $\mathfrak{M}$ are determined by their restrictions to X.

Lemma 2.10. *Let X be an infinite computable subset of $\mathcal{G}$. There is at most one function $\zeta \colon \mathfrak{A}^{<X} \to \mathfrak{A}^X$ that is $\langle \mathcal{G}, X \rangle$-computable via $\mathfrak{M}$, and it must be Turing equivalent to $\{\mathfrak{M}\}$.*

Proof. Suppose there is some $\zeta\colon \mathfrak{A}^{<X} \to \mathfrak{A}^X$ that is $\langle \mathcal{G}, X\rangle$-computable via $\mathfrak{M}$. Then by Definition 2.9(a), $\{\mathfrak{M}\}$ is total. By Definition 2.9(b), for any $x \in \mathfrak{A}^{<G}$, the value of $\{\mathfrak{M}\}(x)$ only depends on $x|_X$, and so $\{\mathfrak{M}\}$ induces a function $\delta\colon \mathfrak{A}^{<X} \to \mathfrak{A}^G$. By Definition 2.9(d), the map $\mathfrak{A}^{<X} \to \mathfrak{A}^X$ given by $a \mapsto \delta(a)|_X$ is the same as ζ. Therefore there is at most one function $\mathfrak{A}^{<X} \to \mathfrak{A}^X$ that is $\langle \mathcal{G}, X\rangle$-computable via $\mathfrak{M}$.

By Definition 2.9(c), $\{\mathfrak{M}\}(x)|_{G\setminus X}$ is the constant 0 function for all x. Therefore $\{\mathfrak{M}\}$ is Turing equivalent to ζ. $\square$

2.1 Resource-bounded graph computation

Just as one may consider ordinary computability restricted by bounds on the time and space needed for the computation, one may devise and study complexity classes for graph computability. However, as we will see in §§3.2 and 4.2, unlike with ordinary computability, a great deal can be done with merely *constant time*, and in the finitary case, our key constructions can be carried out by machines that run in *linear space* — both of which define here.

Throughout this subsection, Q will be a collection of functions from $\mathbb{N}$ to $\mathbb{N}$.

Definition 2.11. *A function ζ is* **Q-time computable via $\mathfrak{M}$** *if*

(a) ζ is $\mathcal{G}$-computable via $\mathfrak{M}$, and

(b) there is a $q \in Q$ such that for all finite connected subgraphs $A \subseteq G$ and all starting configurations f of $\mathfrak{M}$ supported on A,

$$(\forall v \in G) \quad \langle \mathfrak{M}, f\rangle\big(v, q(|A|)\big) = \langle \mathfrak{M}, f\rangle\big(v, q(|A|) + 1\big)$$

i.e., $\mathfrak{M}$ halts in at most $q(|A|)$-many timesteps.

A function is **Q-time graph computable** *if it is Q-time computable via $\mathfrak{M}^\circ$ for some graph machine $\mathfrak{M}^\circ$.*

In this paper, we consider mainly time bounds where Q is the collection of constant functions $\{\lambda x.n \,:\, n \in \mathbb{N}\}$, in which case we speak of *constant-time graph computability.*

When bounding the space used by a computation, we will consider graph computations that depend only on a "small" neighborhood of the input.

Definition 2.12. *For each $A \subseteq G$ and $n \in \mathbb{N}$, the* **n-neighborhood** *of A, written $\mathbf{N}_n(A)$, is defined by induction as follows.*

Case 1: The 1-neighborhood of A is

$$\mathbf{N}_1(A) := A \cup \{v \in G \,:\, (\exists a \in A)\ E(v, a) \cup E(a, v) \neq \emptyset\}.$$

Case $k+1$: The $k+1$-neighborhood of A is

$$\mathbf{N}_{k+1}(A) = \mathbf{N}_1(\mathbf{N}_k(A)).$$

Definition 2.13. *A graph machine $\mathfrak{M}$* **runs in Q-space** *if $\{\mathfrak{M}\}$ is total and there are $p, q \in Q$ such that for any finite connected subgraph $A \subseteq G$ and any starting configuration f of $\mathfrak{M}$ that is supported on A, we have $|\mathbf{N}_{p(n)}(A)| \le q(n)$ and*

$$\langle \mathfrak{M}, f\rangle(v, \,\cdot\,) = \langle \mathfrak{M}|_{\mathbf{N}_{p(n)}(A)}, f|_{\mathbf{N}_{p(n)}(A)}\rangle(v, \,\cdot\,)$$

for all $v \in A$, where $n := |A|$.

*A function ζ is Q-**space graph computable** via $\mathfrak{M}$ if ζ is $\mathcal{G}$-computable via $\mathfrak{M}$ where $\mathfrak{M}$ runs in Q-space. We say that ζ is Q-**space graph computable** if it is Q-space graph computable via $\mathfrak{M}^\circ$ for some graph machine $\mathfrak{M}^\circ$.*

In this paper, the main space bound we consider is where Q is the collection of linear polynomials, yielding *linear-space graph computability.* This definition generalizes the standard notion of linear-space computation, and reduces to it in the case of a graph machine that straightforwardly encodes an ordinary Turing machine (for details of the encoding see §6.1). For such encodings, the only starting configurations yielding nontrivial computations are those supported on a neighborhood containing the starting location of the Turing machine read/write head. In the case of arbitrary computable graph machines, computations can meaningfully occur from starting configurations supported on arbitrary connected subgraphs. This is why the bound on the size of the neighborhoods required to complete the computation is required to depend only on the size of a connected subgraph the starting configuration is supported on.

One way to view space-bounded graph computation is as computing functions that need only a finite amount of a graph to perform their computation, where this amount depends only on the "size" of the input (as measured by the size of a connected support of the starting configuration corresponding to the input). This perspective is especially natural if one thinks of the infinite graph as a finite graph that is larger than is needed for all the desired inputs.

3 Arbitrary graphs

In this section, we consider the possible Turing degrees of total graph computable functions. We begin with a bound for finite graphs.

Lemma 3.1. *Suppose G is finite. Let $\mathbf{h}$ be the map which takes a valid configuration f for $\mathfrak{M}$ and returns $n \in \mathbb{N}$ if $\langle \mathfrak{M}, f\rangle$ halts at stage n (and not earlier), and returns ∞ if $\langle \mathfrak{M}, f\rangle$ doesn't halt. Then*

- *$\langle \mathfrak{M}, f\rangle$ is computable and*
- *$\mathbf{h}$ is computable.*

Proof. Because G is finite, $\langle \mathfrak{M}, f\rangle$ is computable. Further, there are are only finitely many valid configurations of $\mathfrak{M}$. Hence there must be some $n, k \in \mathbb{N}$ such that for all vertices v in the underlying set of $\mathfrak{M}$, we have $\langle \mathfrak{M}, f\rangle(v, n) = \langle \mathfrak{M}, f\rangle(v, n + k)$, and the set of such pairs (n, k) is computable. Note that $\langle \mathfrak{M}, f\rangle$ halts if and only if there is some n, less than or equal to the number of valid configuration for $\mathfrak{M}$, for which this holds for $(n, 1)$. Hence $\mathbf{h}$, which searches for the least such n, is computable. □

We now investigate which Turing degrees are achieved by arbitrary computable graph machines.

3.1 Upper bound

We will now show that every graph computable function is computable from $\mathbf{0}^{(\omega)}$.

Definition 3.2. *Let f be a valid configuration for $\mathfrak{M}$, and let A be a finite subset of G. We say that $(B_i)_{i\leq n}$ is an n-**approximation of $\mathfrak{M}$ and f on A** if*

- *$A = B_0$,*
- *$B_i \subseteq B_{i+1} \subseteq G$ for all $i < n$, and*

- *if $B_{i+1} \subseteq B \subseteq G$ then for all $v \in B_{i+1}$,*

$$\langle \mathfrak{M}|_{B_{i+1}}, f_i|_{B_{i+1}} \rangle(v,1) = \langle \mathfrak{M}|_B, f_i|_B \rangle(v,1),$$

where again $f_i := \langle \mathfrak{M}, f \rangle(\,\cdot\,, i)$.

The following proposition (in the case where $\ell = n - n'$) states that if $(B_i)_{i \leq n}$ is an n-approximation of $\mathfrak{M}$ and f on A, then as long as we are only running $\mathfrak{M}$ with starting configuration f for ℓ-many steps, and are only considering the states of elements within $B_{n'}$, then it suffices to restrict $\mathfrak{M}$ to B_n.

Proposition 3.3. *The following claim holds for every $n \in \mathbb{N}$: For every valid configuration f for $\mathfrak{M}$, and finite $A \subseteq G$,*

- *there is an n-approximation of $\mathfrak{M}$ and f on A, and*
- *if $(B_i)_{i \leq n}$ is such an approximation, then*

$$\begin{aligned} &(\forall n' < n)(\forall \ell \leq n - n')(\forall v \in B_{n'}) \\ &\qquad \langle \mathfrak{M}|_{B_{n'+\ell}}, f|_{B_{n'+\ell}} \rangle(v,\ell) = \langle \mathfrak{M}, f \rangle(v,\ell). \quad (\Box_n) \end{aligned}$$

Proof. We will prove this by induction on n.

Base case: The claim is vacuous for $n = 0$, as there are no nonnegative values of n' to check.

Inductive case: Proof of the claim for $n = k+1$, assuming its truth for $n \leq k$.
To establish $(\Box_{k+1})$ consider

$$(\forall v \in B_{n'})\ \langle \mathfrak{M}|_{B_{n'+\ell}}, f|_{B_{n'+\ell}} \rangle(v,\ell) = \langle \mathfrak{M}, f \rangle(v,\ell) \qquad (\dagger)$$

where $n' < k+1$ and $\ell \leq (k+1) - n'$. If $\ell < (k+1) - n'$ and $k = 0$ then $\ell = 0$, and $(\dagger)$ holds trivially. If $\ell < (k+1) - n'$ and $k > 0$ then $\ell \leq k - n'$, and so $(\dagger)$ holds by the inductive hypothesis $(\Box_k)$. Hence we may restrict attention to the case where $\ell = (k+1) - n'$.

Let f be a valid configuration for $\mathfrak{M}$, let $A \subseteq G$ be finite, and let $(B_i)_{i \leq k}$ be a k-approximation of $\mathfrak{M}$ and f_1 on A. Let D_{k+1} be a subset of G such that for every $v \in B_k$ and every color c, if vertex v receives a pulse (in $\langle \mathfrak{M}, f \rangle$) of color c at the start of timestep 1, then there is some vertex $d \in D_{k+1}$ which sends a pulse of color c to v during timestep 1. Note that because there are only finitely many colors of pulses which elements of B_k can receive, and because B_k is finite, we can assume that D_{k+1} is finite as well.

Now let $B_{k+1} = B_k \cup D_{k+1}$. Because each vertex in B_k receives the same color pulses in $\mathfrak{M}$ as it does in $\mathfrak{M}|_B$ for any set B containing B_{k+1}, we have that $\langle \mathfrak{M}|_B, f|_B \rangle(b,1)$ agrees with $\langle \mathfrak{M}, f \rangle(b,1)$ whenever $b \in B_k$ and $B_{k+1} \subseteq B$. Therefore $(B_i)_{i \leq k+1}$ is a $(k+1)$-approximation of $\mathfrak{M}$ and f on A, and (B_k, B_{k+1}) is a 1-approximation of $\mathfrak{M}$ and f on B_k.

If $n' = k$, then $\ell = 1$, and so $(\dagger)$ holds by $(\Box_1)$ applied to the approximation (B_k, B_{k+1}).

If $n' < k$, then by induction, we may use $(\Box_k)$ where the bound variable f is instantiated by f_1, the bound variable n' by n', and the bound variable ℓ by $\ell - 1$, to deduce that

$$\langle \mathfrak{M}|_{B_{n'+\ell-1}}, f_1|_{B_{n'+\ell-1}} \rangle(v,\ell-1) = \langle \mathfrak{M}, f_1 \rangle(v,\ell-1)$$

for all $v \in B_{n'}$.

By Lemma 2.7, we have $\langle \mathfrak{M}, f_1 \rangle(v,\ell-1) = \langle \mathfrak{M}, f \rangle(v,\ell)$ and $\langle \mathfrak{M}|_{B_{n'+\ell-1}}, f_1|_{B_{n'+\ell-1}} \rangle(v,\ell-1) = \langle \mathfrak{M}|_{B_{n'+\ell-1}}, f|_{B_{n'+\ell-1}} \rangle(v,\ell)$ for all $v \in B_{n'}$. Because $(B_i)_{i \leq k}$ is a k-approximation of $\mathfrak{M}$ and f_1 on A, we know that $\langle \mathfrak{M}|_{B_{n'+\ell-1}}, f|_{B_{n'+\ell-1}} \rangle(v,\ell) = \langle \mathfrak{M}|_{B_{n'+\ell}}, f|_{B_{n'+\ell}} \rangle(v,\ell)$ for all $v \in B_{n'}$.

Therefore $(\dagger)$ holds, and we have established $(\Box_{k+1})$. $\Box$

We now analyze the computability of approximations and of runs.

Proposition 3.4. *Let $n \in \mathbb{N}$. For all computable graph machines $\mathfrak{M}$ and configurations f that are valid for $\mathfrak{M}$, the following are $\mathbf{f}^{(n)}$-computable, where $\mathbf{f}$ is the Turing degree of f.*

- *The collection $P_n(f) := \{(A, (B_i)_{i \leq n}) : A \subseteq G$ is finite and $(B_i)_{i \leq n}$ is an n-approximation of $\mathfrak{M}$ and f on $A\}$.*
- *The function $f_n := \langle \mathfrak{M}, f \rangle(\,\cdot\,, n)$.*

Further, these computability claims are uniform in n.

Proof. We will prove this by induction on n. The uniformity follows, since for all $n > 1$, we provide the same reduction to $\mathbf{f}^{(n)}$ and parametrized earlier quantities.

Base case (a): Proof of claim for $n = 0$.
Let $\mathfrak{M}$ be a graph machine and f a valid configuration for $\mathfrak{M}$. Given a finite $A \subseteq G$, the sequence (A) is the only 0-approximation of $\mathfrak{M}$ and f on A. Hence $P_0(f) = \{(A, (A)) : A \subseteq G \text{ finite}\}$ is computable. Further, $f_0 = f$ is computable from $\mathbf{f}^{(0)} = \mathbf{f}$.

Base case (b): Proof of claim for $n = 1$.
Let $\mathfrak{M}$ be a graph machine and f a valid configuration for $\mathfrak{M}$. For each finite $A \subseteq G$ and each finite B_0, B_1 containing A we can $\mathbf{f}$-compute whether $\langle \mathfrak{M}|_{B_0}, f|_{B_0}\rangle(v, 1) = \langle \mathfrak{M}|_{B_1}, f|_{B_1}\rangle(v, 1)$, and so we can $\mathbf{f}'$-compute $P_1(f)$. But by Proposition 3.3, we know that if $(A, (A, B)) \in P_1$ then for any $v \in A$ we have $\langle \mathfrak{M}, f\rangle(v, 1) = \langle \mathfrak{M}|_B, f|_B\rangle(v, 1)$. Hence we can compute f_1 from P_1, and so it is $\mathbf{f}'$-computable.

Inductive case: Proof of claim for $n = k + 1$ (where $k \geq 1$), assuming it for $n = k$.
Let $\mathfrak{M}$ be a graph machine and f a valid configuration for $\mathfrak{M}$. We know that $(B_i)_{i \leq k+1}$ is a $(k+1)$-approximation of $\mathfrak{M}$ and f on A if and only if both (i) the sequence $(B_i)_{i \leq k}$ is a k-approximation of $\mathfrak{M}$ and f_1 on A, and (ii) the sequence (B_k, B_{k+1}) is a 1-approximation of $\mathfrak{M}$ and f on B_k.

We can therefore compute $P_{k+1}(f)$ from $P_k(f_1)$ and $P_1(f)$. By the inductive hypothesis, $P_1(f)$ is $\mathbf{f}'$-computable. Hence we must show that $P_k(f_1)$ is $\mathbf{f}^{(k+1)}$-computable. Also by the inductive hypothesis, $P_k(f_1)$ is computable from the k'th Turing jump of f_1, and f_1 is $\mathbf{f}'$-computable. Hence $P_k(f_1)$ is $\mathbf{f}^{(k+1)}$-computable.

Finally, by Proposition 3.3, if $(B_i)_{i \leq k+1}$ is an approximation of $\mathfrak{M}$ for f and A up to $k+1$ then for any $v \in A$ we have $\langle \mathfrak{M}|_{B_{k+1}}, f|_{B_{k+1}}\rangle(v, k+1) = \langle \mathfrak{M}, f\rangle(v, k+1)$. We can therefore compute f_{k+1} from $P_{k+1}(f)$ (which will find such an approximation). Hence f_{k+1} is $\mathbf{f}^{(k+1)}$-computable. □

We then obtain the following two results.

Corollary 3.5. *If f is a valid configuration for $\mathfrak{M}$, then f_n is $\mathbf{f}^{(n)}$-computable and so $\langle \mathfrak{M}, f\rangle$ is $\mathbf{f}^{(\omega)}$-computable, where $\mathbf{f}$ is the Turing degree of f.*

Proof. By Proposition 3.3, for each $v \in G$ (the underlying set of $\mathfrak{M}$) and each $n \in \mathbb{N}$, there is an approximation of $\mathfrak{M}$ for f and $\{v\}$ up to n. Further, by Proposition 3.4 we can $\mathbf{f}^{(n)}$-compute such an approximation, uniformly in v and n. But if $(B_i^v)_{i \leq n}$ is an approximation of $\mathfrak{M}$ for f and $\{v\}$ up to n then $\langle \mathfrak{M}|_{B_n^v}, f|_{B_n^v}\rangle(v, n) = \langle \mathfrak{M}, f\rangle(v, n)$. So $f_n = \langle \mathfrak{M}, f\rangle(\,\cdot\,, n)$ is $\mathbf{f}^{(n)}$-computable, uniformly in n. Hence $\langle \mathfrak{M}, f\rangle$ is $\mathbf{f}^{(\omega)}$-computable. □

Theorem 3.6. *Suppose that $\{\mathfrak{M}\} \colon \mathfrak{A}^{<G} \to \mathfrak{A}^G$ is a total function. Then $\{\mathfrak{M}\}$ is computable from $\mathbf{0}^{(\omega)}$.*

Proof. Let f be any starting configuration of $\mathfrak{M}$. Then f is computable. Hence by Corollary 3.5, $\langle \mathfrak{M}, f\rangle(v, n+1)$ is $\mathbf{0}^{(n+1)}$-computable. This then implies that the function determining whether or not $\{\mathfrak{M}\}(x)$ halts after n steps is $\mathbf{0}^{(n+2)}$-computable.

But by assumption, $\{\mathfrak{M}\}(x)$ halts for every $x \in \mathfrak{A}^{<G}$, and so $\{\mathfrak{M}\}$ is $\mathbf{0}^{(\omega)}$-computable. □

3.2 Lower bound

We have seen that every graph computable function is computable from $\mathbf{0}^{(\omega)}$. In this subsection, we will see that this bound can be obtained. We begin by showing that every arithmetical Turing degree has an element that is graph computable in constant time. From this we then deduce that there is a graph computable function Turing equivalent to $\mathbf{0}^{(\omega)}$.

We first recall the following standard result from computability theory (see [Soa87, III.3.3]).

Lemma 3.7. *Suppose $n \in \mathbb{N}$ and $X \subseteq \mathbb{N}$. Then the following are equivalent.*

- $X \leq_{\mathrm{T}} \mathbf{0}^{(n)}$.
- *There is a computable function $g\colon \mathbb{N}^{n+1} \to \mathbb{N}$ such that*
  - $h(\,\cdot\,) := \lim_{x_0 \to \infty} \cdots \lim_{x_{n-1} \to \infty} g(x_0, \ldots, x_{n-1}, \,\cdot\,)$ *is total.*
  - $h \equiv_{\mathrm{T}} X$.

We now give the following construction.

Proposition 3.8. *Let $n \in \mathbb{N}$ and suppose $g\colon \mathbb{N}^{n+1} \to \mathbb{N}$ is computable such that*

$$h(\,\cdot\,) := \lim_{x_0 \to \infty} \cdots \lim_{x_{n-1} \to \infty} g(x_0, \ldots, x_{n-1}, \,\cdot\,)$$

is total. Then h is graph computable in constant time $5n + 5$, via a graph machine whose labels, colors, alphabets, states, and lookup table are all finite and do not depend on n or g.

Proof. The first step in the construction is to define a graph machine which can take the limit of a sequence. We will think of this a subroutine which we can (and will) call several times. Let $\mathcal{G}_{\mathfrak{L}}$ be the following graph.

- The underlying set is $\mathbb{N} \cup \{*\}$, where $*$ is some new element.
- There is only one label, p, which all vertices have.
- The colors of $\mathcal{G}_{\mathfrak{L}}$ are $\{\mathbf{B}_0, \mathbf{B}_1, \mathbf{SB}, \mathbf{SF}_0, \mathbf{SF}_1, \mathbf{A}\}$.
- The edge coloring is $E_{\mathfrak{L}}$, satisfying the following for all $m, m' \in \mathbb{N}$.
  - $E_{\mathfrak{L}}(m, m') = \emptyset$ and $E_{\mathfrak{L}}(m', m) = \{\mathbf{B}_0, \mathbf{B}_1\}$ when $m < m'$.
  - $E_{\mathfrak{L}}(m, m) = \{\mathbf{B}_0, \mathbf{B}_1\}$.
  - $E_{\mathfrak{L}}(*, m) = \{\mathbf{SB}\}$ and $E(m, *) = \{\mathbf{SF}_0, \mathbf{SF}_1\}$.
  - $E_{\mathfrak{L}}(*, *) = \emptyset$.

Let $\mathfrak{M}_{\mathfrak{L}}$ be the following graph machine.

- The underlying graph is $\mathcal{G}_{\mathfrak{L}}$.
- The alphabet is $\{0, 1\}$.
- The states of $\mathfrak{M}_{\mathfrak{L}}$ are $\{s, a_0, a_1, a_2, u, b\}$.
- The lookup table $T_{\mathfrak{L}}$ satisfies the following, for all z in the alphabet, states x, and collections X of colors.
  - (i) $T_{\mathfrak{L}}(p, \emptyset, z, s) = T_{\mathfrak{L}}(\emptyset, 0, s)$.
  - (ii) $T_{\mathfrak{L}}(p, X \cup \{\mathbf{A}\}, z, x) = (\emptyset, 0, a_0)$.

(iii) $T_{\mathfrak{L}}(p, X, z, a_0) = (\{\mathbf{SB}\}, 0, a_1)$.

(iv) $T_{\mathfrak{L}}(p, X, z, a_1) = (\emptyset, 0, a_2)$.

(v) $T_{\mathfrak{L}}(p, X, z, a_2) = (\emptyset, k, u)$ if $\mathbf{SF}_k \in X$ and $\mathbf{SF}_{1-k} \notin X$ for some $k \in \{0, 1\}$.

(vi) $T_{\mathfrak{L}}(p, X \cup \{\mathbf{SB}\}, z, x) = (\{\mathbf{B}_z\}, 0, b)$ if $x \notin \{a_0, a_1, a_2\}$.

(vii) $T_{\mathfrak{L}}(p, X, z, b) = (\{\mathbf{SF}_k\}, 0, u)$ if $\mathbf{B}_k \in X$ and $\mathbf{B}_{1-k} \notin X$ for some $k \in \{0, 1\}$.

(viii) $T_{\mathfrak{L}}(p, X, z, b) = (\emptyset, 0, u)$ if $\{\mathbf{B}_0, \mathbf{B}_1\} \subseteq X$.

(ix) $T_{\mathfrak{L}}(p, X, z, u) = (\emptyset, z, u)$ if $\mathbf{A} \notin X$.

(x) $T_{\mathfrak{L}}(p, X, z, x) = (\emptyset, 0, u)$ in all other cases.

We now describe what this graph machine does, beginning from a starting configuration. First, condition (i) sets everything to a clean slate, i.e., makes sure that at the beginning of the second timestep, every vertex will display 0. This ensures that the outcome won't depend on the values initially displayed on any vertex.

Next, by condition (ii), if a vertex receives a pulse of type $\mathbf{A}$ then at the next timestep it enters state a_0. We can think of this as signaling that this subroutine has been "activated". This will only ever be sent to the element $*$, which we call the "activation vertex".

Then, by conditions (iii) and (iv), once the activation vertex is in state a_0 it will send an $\mathbf{SB}$-pulse to every other vertex. This signals to them to start calculating the limit. The activation vertex will then pause and enter state a_1 and then a_2. This pause will give the other vertices an opportunity to calculate their limiting value.

The way the vertices calculate the limiting values is as follows. Once a vertex receives an $\mathbf{SB}$-pulse, it sends a pulse to its predecessors (in $\mathbb{N}$) announcing its currently displayed symbol (using the encoding $0 \mapsto \mathbf{B}_0$ and $1 \mapsto \mathbf{B}_1$). This is described in condition (vi).

Once a vertex has received the collection of those symbols displayed on vertices greater than it, it asks whether both symbols 0 and 1 occur in this collection. If so, then it knows that its displayed symbol is not the limit, and so it enters state u and does nothing (i.e., u signifies termination for the subroutine). This is described in condition (viii).

On the other hand, if a vertex sees that there is only one symbol displayed among those vertices larger than itself, then it knows that that symbol is the limiting value of the subroutine. The vertex then passes this information along to the activation vertex, via a pulse of type $\mathbf{SF}_0$ or $\mathbf{SF}_1$ (depending on the limiting value) and enters the subroutine termination state. This is described in condition (vii).

Finally, if the activation vertex is in a_2 and receives exactly one pulse among $\mathbf{SF}_0$ and $\mathbf{SF}_1$, then it knows that this is the limiting value, and sets its display symbol to that value and enters the subroutine termination state. This is described in condition (v).

Of course, once a vertex is in the termination state, it will stay there, displaying the same symbol, unless and until it receives a pulse of type $\mathbf{A}$. This is described in condition (ix).

Condition (x) was added to complete the description of the lookup table, but because h is total, this will never occur in any actual run that begins at a starting configuration, even when this graph machine is embedded as a subroutine into a larger graph machine, as we subsequently describe. Note that this subroutine will always complete its computation within 4 timesteps.

We need one more graph machine to operate as a subroutine. The purpose of this subroutine will be to send pulses, in sequence, that activate other vertices.

Let $\mathcal{G}^n_{\mathfrak{B}}$ be the graph satisfying the following.

- The underlying set is $\{\star_{-5n}, \dots, \star_0\}$.
- There is only one label q, which all elements have.
- The colors are $\{\mathbf{S}, \mathbf{R}, \mathbf{Q}, \mathbf{A}, \mathbf{SF}_0, \mathbf{SF}_1\}$

- The edge coloring is $E^n_{\mathfrak{B}}$.
- The only vertex pairs having non-empty sets of edge colors are the following.
  - $E^n_{\mathfrak{B}}(\star_i, \star_{i+1}) = \{\mathbf{R}\}$ for $-5n \leq i < 0$.
  - $E^n_{\mathfrak{B}}(\star_0, \star_0) = \{\mathbf{S}\}$.
  - $E^n_{\mathfrak{B}}(\star_0, \star_{-5n}) = \{\mathbf{Q}\}$.

We define the graph machine $\mathfrak{M}^n_{\mathfrak{B}}$ as follows.

- The underlying graph is $\mathcal{G}^n_{\mathfrak{B}}$.
- The alphabet is $\{0, 1\}$.
- The states are $\{s, d, r, u\}$.
- The lookup table $T^n_{\mathfrak{B}}$ satisfies the following for all z in the alphabet, states x, and collections X of colors.
  - (i) $T^n_{\mathfrak{B}}(q, X, 0, s) = (\emptyset, 0, s)$ if $\{\mathbf{R}, \mathbf{S}\} \cap X = \emptyset$.
  - (ii) $T^n_{\mathfrak{B}}(q, X, 1, s) = (\{\mathbf{S}\}, 0, s)$.
  - (iii) $T^n_{\mathfrak{B}}(q, X \cup \{\mathbf{S}\}, z, s) = (\{\mathbf{Q}\}, 0, s)$ if $\mathbf{R} \notin X$.
  - (iv) $T^n_{\mathfrak{B}}(q, X, z, s) = (\{\mathbf{A}\}, 0, d)$ if $\{\mathbf{Q}, \mathbf{R}\} \cap X \neq \emptyset$.
  - (v) $T^n_{\mathfrak{B}}(q, X, z, d) = (\{\mathbf{R}\}, 0, r)$.
  - (vi) $T^n_{\mathfrak{B}}(q, X, z, r) = (\emptyset, k, d)$ if $\mathbf{SF}_k \in X$ and $\mathbf{SF}_{1-k} \notin X$ for some $k \in \{0, 1\}$, and otherwise $T^n_{\mathfrak{B}}(q, X, z, r) = (\emptyset, 0, r)$.
  - (vii) $T^n_{\mathfrak{B}}(q, X, z, u) = (X, z, u)$.
  - (viii) $T^n_{\mathfrak{B}}(q, X, z, x) = (\emptyset, 0, u)$ in all other cases.

We now describe what the graph machine $\mathfrak{M}^n_{\mathfrak{B}}$ does. First notice that the only way for a vertex to get out of the initial state s is for it to receive an **S**-pulse, a **Q**-pulse or an **R**-pulse. Only **S**-pulses can be sent from a vertex that is in the initial state and which hasn't received any other pulses. Also, there is only one **S**-edge, namely, a self loop at $\star_0$. Hence the first timestep of the graph machine's behavior is determined by what the vertex $\star_0$ initially displays.

If the vertex $\star_0$ initially displays 0, then all vertices display 0 in the next step, and the subroutine does nothing else. This is described in condition (i). If, however, vertex $\star_0$ initially displays 1, then an **S**-pulse is sent by $\star_0$ to itself. This is described in condition (ii). Once $\star_0$ receives the **S**-pulse, it reverts back to displaying 0, and sends a **Q**-pulse to $\star_{-5n}$. This is described in condition (iii).

Once vertex $\star_{-5n}$ receives a **Q**-pulse, the main loop begins. In the main loop, first vertex $\star_{-5n}$ sends out an **A**-pulse and moves to a state d, as described in condition (iv). The purpose of the **A**-pulse is to tell the vertex that receives it to *activate* and start calculating a limit. While there are no vertices in $\mathfrak{M}^n_{\mathfrak{B}}$ with **A**-colored edge, we will combine $\mathfrak{M}^n_{\mathfrak{B}}$ with copies of $\mathfrak{M}_{\mathfrak{L}}$, connecting the two graphs using **A**-colored edges. Note that only vertices of the form $\star_{-5k}$ with $k \leq n$ will be connected to copies of $\mathfrak{M}_{\mathfrak{L}}$ via **A**-colored edges. The other vertices are there to provide additional timesteps in between **A**-pulses to allow the activated copies of $\mathfrak{M}_{\mathfrak{L}}$ time to complete their computations.

Once a vertex is in state d, it sends an **R**-pulse to its "neighbor to the right" (the vertex with least index greater than it), and moves to state r, where it will stay unless it is $\star_0$, as described in conditions (v) and (vi). Every vertex acts the same way upon arrival of an **R**-pulse as on an **Q**-pulse. Hence each vertex in the sequence sends, in succession, an **A**-pulse, and then enters the state r.

Condition (vii) ensures that when a vertex enters the subroutine termination state (which will only ever happens to $\star_0$ in the course of this subroutine's use by the larger program), the displayed symbol remains constant. Condition (viii) also describes a circumstance that happens only when the subroutine is used by the larger program, as does the first clause of condition (vi) (which agrees with condition (v) of $\mathfrak{M}_{\mathfrak{L}}$).

When we connect up $\mathfrak{M}^n_{\mathfrak{B}}$ with copies of $\mathfrak{M}_{\mathfrak{L}}$, vertex $\star_0$ participates in calculating the final limit. Hence, in addition to having edges colored by $\mathbf{A}$, vertex $\star_0$ also has $\mathbf{SF}_0$ and $\mathbf{SF}_1$-edges. If $\star_0$ receives a pulse of one of those colors, it then displays the corresponding value and moves to a state that keeps this value constant.

We now combine the graph machines for subroutines, to get a graph machine that calculates h.

For $e \in \mathbb{N}$, define the graph $\mathcal{G}_{g,e}$ as follows.

- The underlying set is $\mathbb{N}^n \cup \mathbb{N}^{n-1} \cup \dots \cup \mathbb{N}^2 \cup \mathbb{N} \cup \{\star_{-5n}, \dots, \star_0\}$.
- There are two labels, p and q. Vertex $\star_i$ is labeled by q (for $-5n \leq i \leq 0$), and every other vertex is labeled by p.
- The colors are $\{\mathbf{S}, \mathbf{C}_0, \mathbf{C}_1, \mathbf{B}_0, \mathbf{B}_1, \mathbf{SB}, \mathbf{SF}_0, \mathbf{SF}_1, \mathbf{A}, \mathbf{Q}, \mathbf{R}\}$.
- The edge coloring is E, satisfying the following.
    - For each $\bar{c} \in \mathbb{N}^{n-1}$ and $k, m \in \mathbb{N}$ with $k \neq m$ we have the following.
        - $E(\bar{c}k, \bar{c}m) = E_{\mathfrak{L}}(k, m)$.
        - $E(\bar{c}k, \bar{c}k) = \{\mathbf{C}_{g(\bar{c}ke)}, \mathbf{B}_0, \mathbf{B}_1\}$.
        - $E(\bar{c}, \bar{c}k) = E_{\mathfrak{L}}(*, k)$ and $E(\bar{c}k, \bar{c}) = E_{\mathfrak{L}}(k, *)$.
        - $E(\star_{-5n}, \bar{c}k) = \{\mathbf{A}\}$.
    - For each $\bar{c} \in \mathbb{N}^i$ for $0 < i < n-1$ and $k, m \in \mathbb{N}$ we have the following.
        - $E(\bar{c}k, \bar{c}m) = E_{\mathfrak{L}}(k, m)$.
        - $E(\bar{c}, \bar{c}k) = E_{\mathfrak{L}}(*, k)$ and $E(\bar{c}k, \bar{c}) = E_{\mathfrak{L}}(k, *)$.
        - $E(\star_{-5(n-i+1)}, \bar{c}k) = \{\mathbf{A}\}$.
    - For each $k, m \in \mathbb{N}$ we have the following.
        - $E(k, m) = E_{\mathfrak{L}}(k, m)$.
        - $E(\star_0, k) = E_{\mathfrak{L}}(*, k)$ and $E(k, \star_0) = E_{\mathfrak{L}}(k, *)$.
        - $E(\star_0, \star_0) = \{\mathbf{S}, \mathbf{A}\}$.
    - For each $-5n \leq k, m < 0$ we have the following.
        - $E(\star_k, \star_m) = E^n_{\mathfrak{B}}(\star_k, \star_m)$.
        - $E(\star_k, \star_0) = E^n_{\mathfrak{B}}(\star_0, \star_0)$.

The graph $\mathcal{G}_{g,e}$ is such that for any tuple $\bar{c} \in \mathbb{N}^{\leq k}$, the set $\{\bar{c}\} \cup \{\bar{c}k\}_{k \in \mathbb{N}}$ is isomorphic to $\mathcal{G}_{\mathfrak{L}}$ (after ignoring the edges $\{\mathbf{C}_0, \mathbf{C}_1\}$). This allows us to iteratively take limits. Further, each $\bar{c} \in \mathbb{N}^n$ has a self-loop which encodes the value of $g(\bar{c}e)$. This will be used to initialize the displayed symbols of vertices in the matrix that we will later use to take the limits.

We define the graph machine $\mathfrak{M}_{g,e}$ as follows.

- The underlying graph is $\mathcal{G}_{g,e}$.
- The states are $\{d, s, a_0, a_1, a_2, u, b\}$.
- The lookup table T is such that the following hold, for all z in the alphabet, states t, and collections X of colors.

(i) $T(q, X, z, t) = T^n_{\mathfrak{B}}(q, X, z, t)$.

(ii) $T(p, X, z, t) = T_{\mathfrak{L}}(p, X, z, t)$ if ($t \neq s$ and $\mathbf{A} \notin X$) or ($t = a_0$ and $\mathbf{C}_k \in X$ and $\mathbf{C}_{1-k} \notin X$) for some $k \in \{0, 1\}$.

(iii) $T(p, X \cup \mathbf{A}, z, s) = (\{\mathbf{C}_0, \mathbf{C}_1\}, 0, a_0)$.

(iv) $T(p, X, z, a_0) = (\emptyset, k, u)$ if $\mathbf{C}_k \in X$ and $\mathbf{C}_{1-k} \notin X$ for some $k \in \{0, 1\}$.

(v) $T(p, X, z, t) = (\emptyset, 0, u)$ in all other cases.

We now describe a run of $\mathfrak{M}_{g,e}$ on a starting configuration. First, just as with $\mathfrak{M}^n_{\mathfrak{B}}$, the computation begins by observing the behavior or $\star_0$. If $\star_0$ initially displays 0, then all vertices stay in the initial state and display 0 on the next timestep. If $\star_0$ initially displays 1, then the computation proceeds as in $\mathfrak{M}^n_{\mathfrak{B}}$, and vertex $\star_{-5n}$ sends an $\mathbf{A}$-pulse, which *activates* all vertices of the form $\overline{c} \in \mathbb{N}^n$.

Vertices of the form $\overline{c} \in \mathbb{N}^n$ are not designed to compute the limits of anything, and so upon activation their values must be initialized. This is done by each such vertex attempting to send itself an $\mathbf{C}_0$-pulse and $\mathbf{C}_1$-pulse. In other words, each such vertex sends a $\mathbf{C}_0$-pulse and $\mathbf{C}_1$-pulse along all $\mathbf{C}_0$-edges and $\mathbf{C}_1$-edges connected to it, respectively, if any exist (and all $\mathbf{C}_0$-edges and $\mathbf{C}_1$-edges connected to such a vertex are self-loops). Because of how the graph $\mathcal{G}_{g,e}$ was constructed, each such $\overline{c}$ will only receive the pulse $\mathbf{C}_{g(\overline{c}e)}$. Hence after the pulse is received, vertex $\overline{c}$ completes its initialization by setting its displayed symbol to the index of whichever pulse it receives.

Meanwhile, vertices $\{\star_{-5n}, \dots, \star_0\}$ are sending $\mathbf{R}$-pulses in succession along the sequence, causing each such vertex, in order, to attempt to send an activation $\mathbf{A}$-pulse (i.e., a pulse along all $\mathbf{A}$-edges connected to it, of which there will be at most one). However, because only every fifth vertex in the sequence $\{\star_{-5n}, \dots, \star_0\}$ is connected via an $\mathbf{A}$-colored edge, by the time the next $\mathbf{A}$-pulse is sent, i.e., along the edge attached to $\star_{-5n+5}$, the initialization procedure has finished.

The next activation pulse is then sent from $\star_{-5n+5}$ to all vertices of the form $\overline{c} \in \mathbb{N}^{n-1}$. It causes these vertices to begin the process of calculating the limit of the sequence currently displayed by the vertices $\{\overline{c}0, \overline{c}1, \dots\}$. While this limit is being calculated, the vertices in $\{\star_{-5n+5}, \dots, \star_0\}$ are attempting to send out activation pulses, in sequence. By the time the next $\mathbf{A}$-pulse is sent out, at $\star_{-5n+10}$, each vertex $\overline{c} \in \mathbb{N}^{n-1}$ is displaying the limit of the values displayed by $\{\overline{c}0, \overline{c}1, \dots\}$.

This process then repeats until we get to $\star_0$, which plays a double role. First, once $\star_0$ has received an $\mathbf{R}$-pulse, it sends out an $\mathbf{A}$-pulse to itself. This signals $\star_0$ to begin the process of calculating the limit of the symbols displayed at $\{0, 1, \dots\}$. Secondly, when this calculations finishes, $\star_0$ displays

$$\lim_{x_0 \to \infty} \cdots \lim_{x_{n-1} \to \infty} g(x_0, \dots, x_{n-1}, e),$$

which is the desired value, $h(e)$.

Note that the map $e \mapsto \mathfrak{M}_{g,e}$ is computable, and that the lookup table T is independent of e. Further, the time it takes for $\mathfrak{M}_{g,e}$ to run is independent of e. Therefore h is graph computable in constant time. $\square$

In summary, we define a "subroutine" graph machine that, on its own, computes the limit of a computable binary sequence. We then embed n repetitions of this subroutine into a single graph machine that computes the n-fold limit of the $(n+1)$-dimensional array given by g. The subroutine graph machine has a countably infinite sequence (with one special vertex) as its underlying graph, in that every vertex is connected to all previous vertices (and all are connected to the special vertex). Each vertex first activates itself, setting its displayed symbol to the appropriate term in the sequence whose limit is being computed. Each vertex sends a pulse to every previous vertex, signaling its displayed state. Any vertex which receives both 0 and 1 from vertices later in the sequence knows that the sequence alternates at some later index. Finally, any vertex which only receives a 0 or 1 pulse, but not both, sends a pulse corresponding to the one it receives to the special vertex. This special vertex then knows the limiting value of the sequence.

This technical construction allows us to conclude the following.

Corollary 3.9. *Suppose $X \subseteq \mathbb{N}$ is such that $X \leq_{\mathrm{T}} \mathbf{0}^{(n)}$. Then X is Turing-equivalent to some function that is graph computable in constant time by a machine whose underlying graph is finitary.*

Proof. By Lemma 3.7, X is Turing equivalent to the n-fold limit of some computable function. By Proposition 3.8, this n-fold limit is graph computable in constant time by a machine whose underlying graph is finitary. $\square$

Not only are all graph computable functions Turing reducible to $\mathbf{0}^{(\omega)}$, but this bound can be achieved.

Theorem 3.10. *There is a graph computable function (via a machine whose underlying graph is finitary) that is Turing equivalent to $\mathbf{0}^{(\omega)}$.*

Proof. For each $n \in \mathbb{N}$, uniformly choose a computable function $g_n : \mathbb{N}^{n+1} \to \mathbb{N}$ such that its n-fold limit h_n satisfies $h_n \equiv_{\mathrm{T}} \mathbf{0}^{(n)}$. For $e, n \in \mathbb{N}$, let the graph machines $\mathcal{G}_{g_n,e}$ and $\mathfrak{M}_{g_n,e}$ be the graphs and graph machines described in the proof of Proposition 3.8.

Let $\mathcal{G}_\omega$ be the graph which is the (computable) disjoint union of the uniformly computable graphs $\{\mathcal{G}_{g_n,e} : e, n \in \mathbb{N}\}$. Note that $\mathcal{G}_\omega$ is finitary as the (finite) number of labels of $\mathcal{G}_{g_n,e}$ does not depend on n or e.

Further note that for all $e, n \in \mathbb{N}$, the graph machines $\mathfrak{M}_{g_n,e}$ have the same lookup table. We can therefore let $\mathfrak{M}_\omega$ be the graph machine with underlying graph $\mathcal{G}_\omega$ having this lookup table. The function $\{\mathfrak{M}_\omega\}$ is total because each $\{\mathfrak{M}_{g_n,e}\}$ is (and the corresponding submachines of the disjoint union do not interact).

By the construction of $\mathfrak{M}_\omega$, we have $\mathbf{0}^{(\omega)} \leq_{\mathrm{T}} \{\mathfrak{M}_\omega\}$. On the other hand, $\{\mathfrak{M}_\omega\} \leq_{\mathrm{T}} \mathbf{0}^{(\omega)}$ holds by Theorem 3.6. Hence $\{\mathfrak{M}_\omega\} \equiv_{\mathrm{T}} \mathbf{0}^{(\omega)}$. $\square$

4 Finite degree graphs

We have seen that every arithmetical function is graph computable. However, as we will see in this section, if we instead limit ourselves to graphs where each vertex has finite degree, then not only is every graph computable function computable from $\mathbf{0}'$, but also we can obtain more fine-grained control over the Turing degree of the function by studying the degree structure of the graph.

4.1 Upper bound

Before we move to the specific case of graphs of finite degree (defined below), there is an important general result concerning bounds on graph computability and approximations to computations.

Definition 4.1. *Let $\Theta : \mathfrak{P}_{<\omega}(G) \to \mathfrak{P}_{<\omega}(G)$. We say that Θ is a* **uniform approximation** *of $\mathfrak{M}$ if for all finite subsets $A \subseteq G$,*

- *$A \subseteq \Theta(A)$, and*
- *for any valid configuration f for $\mathfrak{M}$, the pair $(A, \Theta(A))$ is a 1-approximation of $\mathfrak{M}$ and f on A.*

Lemma 4.2. *Let $\Theta(A)$ be a uniform approximation of $\mathfrak{M}$. Then for any finite subset A of G, any valid configuration f for $\mathfrak{M}$, and any $n \in \mathbb{N}$, the tuple $(A, \Theta(A), \Theta^2(A), \ldots, \Theta^n(A))$ is an n-approximation of $\mathfrak{M}$ and f on A.*

Proof. We prove this by induction on n.

Base case: $n = 1$
We know by hypothesis that $(A, \Theta(A))$ is a 1-approximation of $\mathfrak{M}$ and f on A.

Inductive case: $n = k + 1$
We know that $(A, \Theta(A), \dots, \Theta^k(A))$ is a k-approximation of $\mathfrak{M}$ and f_1 on A. It therefore suffices to show that $(\Theta^k(A), \Theta^{k+1}(A))$ is a 1-approximation of $\mathfrak{M}$ and f on A. But this holds by our assumption on Θ. ☐

Note that while we will be able to get even better bounds in the case of finite degree graphs, we do have the following bound on computability.

Lemma 4.3. *Let Θ be a uniform approximation of $\mathfrak{M}$. Then for any valid configuration f,*

(a) $\langle \mathfrak{M}, f \rangle$ is computable from Θ and f (uniformly in f), and

(b) if $\{\mathfrak{M}\}$ is total, then $\{\mathfrak{M}\} \leq_{\mathrm{T}} \Theta'$.

Proof. Clause (a) follows from Lemma 4.2 and the definition of an approximation.

If $\{\mathfrak{M}\}$ is total, then for each starting configuration f of $\mathfrak{M}$, there is an n such that $f_n = f_{n+1} = \{\mathfrak{M}\}(f)$. Hence $\{\mathfrak{M}\}$ is computable from the Turing jump of $\langle \mathfrak{M}, \cdot \rangle$, and so it is computable from Θ'. Therefore clause (b) holds. ☐

We now introduce the *degree function* of a graph.

Definition 4.4. *For $v \in G$, define the* **degree** *of v to be the number of vertices incident with it, i.e.,*

$$\deg_{\mathcal{G}}(v) := |\{w \,:\, E(v, w) \cup E(w, v) \neq \emptyset\}|,$$

and call $\deg_{\mathcal{G}}(\,\cdot\,)\colon G \to \mathbb{N} \cup \{\infty\}$ the **degree function** *of $\mathcal{G}$.*

We say that $\mathcal{G}$ has **finite degree** *when $\mathrm{rng}(\deg_{\mathcal{G}}) \subseteq \mathbb{N}$, and say that $\mathcal{G}$ has* **constant degree** *when $\deg_{\mathcal{G}}$ is constant.*

We will see that for a graph $\mathcal{G}$ of finite degree, its degree function bounds the computability of $\mathcal{G}$-computable functions.

The following easy lemma will allow us to provide a computation bound on graph Turing machines all vertices of whose underlying graph have finite degree.

Lemma 4.5. *Suppose that $\mathcal{G}$ has finite degree. Then $\deg_{\mathcal{G}} \leq_{\mathrm{T}} \mathbf{0}'$.*

Proof. Let G be the underlying set of $\mathcal{G}$. Because $\mathcal{G}$ is computable, for any vertex $v \in G$, the set of neighbors of v is computably enumerable, uniformly in v. The size of this set is therefore $\mathbf{0}'$-computable, uniformly in v, and so $\deg_{\mathcal{G}} \leq_{\mathrm{T}} \mathbf{0}'$. ☐

Recall the definition of n-neighborhood (Definition 2.12).

Lemma 4.6. *Suppose that $\mathcal{G}$ has finite degree. Then*

(a) the 1-neighborhood map $\mathbf{N}_1$ is computable from $\deg_{\mathcal{G}}$, and

(b) for any $\mathcal{G}$-machine $\mathfrak{M}$, the map $\mathbf{N}_1$ is a uniform approximation to $\mathfrak{M}$.

Proof. Clause (a) follows from the fact that given the degree of a vertex one can search for all of its neighbors, as this set is computably enumerable (uniformly in the vertex) and of a known finite size.

Clause (b) follows from the fact that if a vertex receives a pulse, it must have come from some element of its 1-neighborhood. ☐

We now obtain the following more precise upper bound on complexity for finite degree graphs.

Theorem 4.7. *Suppose that $\mathcal{G}$ has finite degree and $\{\mathfrak{M}\}$ is total. Then $\{\mathfrak{M}\}$ is computable from $\deg_{\mathcal{G}}$ and its range is contained in $\mathfrak{A}^{<G}$.*

Proof. First note that if f is a starting configuration of $\mathfrak{M}$ and for all $k' > k$ we have $f(k') = 0$, then for any $m \in \mathbb{N}$ and any $v \in G \setminus \mathbf{N}_m(\{0, \dots, k\})$, we have that $\langle \mathfrak{M}, f \rangle(v, m) = \langle \mathfrak{M}, f \rangle(v, 0)$. Therefore $\{\mathfrak{M}\}(f)$ halts in m steps if and only if $\{\mathfrak{M}|_{\mathbf{N}_m(\{0,\dots,k\})}\}(f|_{\mathbf{N}_m(\{0,\dots,k\})})$ halts in m steps. Therefore we can determine whether or not $\{\mathfrak{M}\}$ halts in m steps by examining $\{\mathfrak{M}|_{\mathbf{N}_m(\{0,\dots,k\})}\}(f|_{\mathbf{N}_m(\{0,\dots,k\})})$, which is uniformly computable from $\mathbf{N}_m$ by Lemma 3.1, since $\mathbf{N}_m(\{0, \dots, k\})$ is finite.

But $\mathbf{N}_m$ is just $\mathbf{N}_1^m$, and by Lemma 4.6(a), $\mathbf{N}_1$ is computable from $\deg_{\mathcal{G}}$. For each m we can uniformly $\deg_{\mathcal{G}}$-computably check whether $\{\mathfrak{M}\}(f)$ halts at stage m. Hence $\{\mathfrak{M}\}$ is $\deg_{\mathcal{G}}$-computable as $\{\mathfrak{M}\}$ halts on all starting configurations. □

We then obtain the following important corollaries.

Corollary 4.8. *Suppose that $\mathcal{G}$ has finite degree. Then any $\mathcal{G}$-computable function is $\mathbf{0}'$-computable.*

Proof. This follows from Theorem 4.7 and Lemma 4.5. □

Corollary 4.9. *Suppose that $\mathcal{G}$ has constant degree. Then any $\mathcal{G}$-computable function is computable (in the ordinary sense).*

Proof. This follows from Theorem 4.7 and the fact that $\deg_{\mathcal{G}}$ is constant (and hence computable). □

4.2 Lower bound

In this subsection, we consider the possible Turing degrees of graph computable functions where the underlying graph has finite degree. In particular, we show that every Turing degree below $\mathbf{0}'$ is the degree of some total graph computable function where the underlying graph has finite degree.

Recall from Lemma 3.7 (for $n = 1$) that a set $X \subseteq \mathbb{N}$ satisfies $X \leq_{\mathrm{T}} \mathbf{0}'$ when the characteristic function of X is the limit of a 2-parameter computable function. The following standard definition (see [Soa87, III.3.8]) describes a hierarchy among sets $X \leq_{\mathrm{T}} \mathbf{0}'$.

Definition 4.10. *Let $A \subseteq \mathbb{N}$ be such that $A \leq_{\mathrm{T}} \mathbf{0}'$, and let h be the characteristic function of A. For $k \in \mathbb{N}$, the set A is* ***k-computably enumerable****, or k-c.e., when there is a computable function $g \colon \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ such that*

- *$(\forall m \in \mathbb{N})\ g(0, m) = 0$,*
- *$(\forall m \in \mathbb{N})\ h(m) = \lim_{n\to\infty} g(n, m)$, and*
- *for each $m \in \mathbb{N}$, $|\{\ell \in \mathbb{N} : g(\ell, m) \neq g(\ell + 1, m)\}| \leq k$, i.e., the uniformly computable approximation to the limit alternates at most k-many times for each input.*

In particular, the 1-c.e. sets are precisely the c.e. sets.

We next construct a collection of graph machines $\{\mathcal{M}_e : e \in \mathbb{N}\}$ that we will use in Theorem 4.12.

Lemma 4.11. *There is a finite lookup table $\mathcal{F}$ and a collection, definable uniformly in e, of graph machines $\mathcal{M}_e$ having edge coloring E_e and common lookup table $\mathcal{F}$ such that whenever $e \in \mathbb{N}$ satisfies*

- *$\{e\} \colon \mathbb{N} \to \{0, 1\}$ is total, and*
- *$\lim_{n\to\infty} \{e\}(n) = m_e$ exists,*

then the following hold.

- *$\mathcal{G}_e$, the underlying graph of $\mathcal{M}_e$, has only finitely many edges and each vertex is of degree at most 3.*
- *If f is a valid configuration of $\mathcal{M}_e$ where all vertices are in the initial state, then*
  - *if $f_{[1]}(0) = 0$, i.e., vertex 0 displays 0 in the configuration f, then $\langle \mathcal{M}_e, f\rangle$ halts with every vertex displaying 0, and*
  - *if $f_{[1]}(1) = 0$, i.e., vertex 0 displays 1 in the configuration f, then $\langle \mathcal{M}_e, f\rangle$ halts with vertex 0 displaying $\lim_{n\to\infty}\{e\}(n) = m_e$ (and every other vertex displaying 0).*
- *If $|\{n : \{e\}(n) \neq \{e\}(n+1)\}| < k$, then $\mathcal{M}_e$ halts on any starting configuration in at most $(2k+4)$-many timesteps.*

Proof. First we describe the graphs $\mathcal{G}_e$. For notational convenience, write $\{e\}(-1) := 1 - \{e\}(0)$. The edges are determined by the following, for $n \geq 0$.

- $E_e(0, 1) = \{r\}$, and $E_e(2, 0) = \{b\}$, and $E_e(1, 2) = \{g\}$.
- If $\{e\}(n) = \{e\}(n + 1)$, then vertices $2n + 1$ and $2n + 2$ have degree 0, i.e., there are no edges connecting them to any other vertices.
- If $\{e\}(n) \neq \{e\}(n+1)$, then $E_e(2k+3, 2n+3) = E_e(2n+3, 2n+4) = E_e(2n+4, 2k+4) = \{g\}$, where k is the "most recent alternation", i.e., the largest k such that $-1 \leq k < n$ and $\{e\}(k) \neq \{e\}(k + 1)$.

We now define the lookup table $\mathcal{F}$.

- There are two states: s (the initial state) and a.
- If a vertex displays 1, then it sends an r-pulse and sets its display to 0.
- If a vertex receives an r-pulse or a g-pulse, then it sends a b-pulse and a g-pulse.
- If a vertex receives a b-pulse, then it sets its state to a and alternates its displayed symbol.

When $\mathcal{M}_e$ is run, on the first step every vertex sets its displayed symbol to 0. If vertex 0 had initially displayed 0, then this is all that happens. However, if vertex 0 had initially displayed 1, then vertex 0 will also send out a r-pulse to vertex 1. This is the only r-pulse that will ever be sent (as 0 is the only vertex which is the source of an r-colored edge, and it will not send any other r-pulses).

At the second stage, this r-pulse can be thought of as being converted to a g-pulse. Further, g-pulses have the property that (1) they propagate forward along directed edges, splitting whenever possible, and (2) they turn into a b-pulse when being sent from vertex 2 to vertex 0. As such, the number of b-pulses that vertex 0 receives is the number of vertices in $\mathcal{M}_e$ with two out edges (i.e., the number of times a g-pulse *splits* in two). However, by construction, the number of such vertices is the number of times that $\{e\}$ alternates values.

Hence the number of b-pulses that vertex 0 receives is equal to the number of times that $\{e\}$ alternates values. But vertex 0 alternates the symbols it displays exactly when it receives a b-pulse, and so when the graph reaches a halting configuration (in the sense that the subsequent configuration one timestep later is the same), vertex 0 will display m_e. ☐

Theorem 4.12. *For every $X : \mathbb{N} \to \{0, 1\}$ such that $X \leq_{\mathrm{T}} \mathbf{0}'$ there is a graph machine $\mathcal{N}_X$ with lookup table $\mathcal{F}$ and finitary underlying graph $\mathcal{H}_X$ such that*

- *every vertex of $\mathcal{H}_X$ has degree at most* 3*,*
- $\{\mathcal{N}_X\}$ *is total and Turing equivalent to X, and*
- *if X is k-c.e. then $\{\mathcal{N}_X\}$ halts in $(2k+4)$-many steps on any input.*

In particular, $\mathcal{N}_X$ runs in linear space.

Proof. Let Y be a computable function such that $X(m) = \lim_{n\to\infty} Y(n,m)$ for all $m \in \mathbb{N}$, and such that if X is k-c.e., then Y witnesses this fact. Let y_m be a code such $\{y_m\}(n) = Y(n,m)$ for all $n \in \mathbb{N}$.

Recall the graphs $\mathcal{G}_e$ (for $e \in \mathbb{N}$) defined in Lemma 4.11, and let $\mathcal{H}_X$ be the disjoint union of $\{\mathcal{G}_{y_m} : m \in \mathbb{N}\}$ in the following sense:

- $\mathcal{H}_X$ is a graph with underlying set $\mathbb{N} \times \mathbb{N}$.
- For each $m \in \mathbb{N}$, let $\mathcal{H}_{X,m}$ be the subgraph of $\mathcal{H}_X$ consisting of elements whose second coordinate is m. Then the map defined by $(n,m) \mapsto n$ for $n \in \mathbb{N}$ is an isomorphism from $\mathcal{H}_{X,m}$ to $\mathcal{G}_{y_m}$.
- There are no edges between any elements of $\mathcal{H}_{X,m}$ and $\mathcal{H}_{X,m'}$ for distinct $m, m' \in \mathbb{N}$.

Note that $\mathcal{H}_X$ is finitary (and in fact has just one label) by the construction of the graphs $\mathcal{G}_e$, which have common lookup table $\mathcal{F}$.

Suppose that $x \in \{0,1\}^{\mathbb{N}\times\mathbb{N}}$ has only finitely many 0's. As earlier, let $\widehat{x}$ be the valid configuration satisfying $\widehat{x}(\ell) = (\emptyset, x(\ell), s)$ for all $\ell \in \mathbb{N} \times \mathbb{N}$. Then by Lemma 4.11, $\langle \mathcal{N}_X, \widehat{x} \rangle$ halts and satisfies the following.

- For any $n, m \in \mathbb{N}$, the vertex $(n+1, m)$ displays 0.
- For any $m \in \mathbb{N}$, if $x(0,m) = 0$ then $\langle \mathcal{N}_X, \widehat{x} \rangle$ halts with 0 displayed at vertex $(0,m)$.
- For any $m \in \mathbb{N}$, if $x(0,m) = 1$ then $\langle \mathcal{N}_X, \widehat{x} \rangle$ halts with $X(m)$ displayed at vertex $(0,m)$.

In particular, $\{\mathcal{N}_X\}$ is a total function that is Turing equivalent to X. Finally note that given any connected subgraph A on which the starting configuration is supported, the computation is completely determined by the $(2k+4)$-neighborhood of that subgraph. Further, each vertex has degree at most 3, and so the size of each such neighborhood is at most $3^{2k+4} \cdot |A|$. Hence $\mathcal{N}_X$ runs in linear space. ☐

5 Graph-theoretic properties that do not affect computability

We now show that the graph machines themselves can be taken to be efficiently computable without affecting the Turing degrees of the functions that can be computed via them. We also show that, while we have made use of directed edges throughout the above constructions, from a computability perspective this was just a notational convenience. The arguments in this section will be sketched, and we refer the reader to a forthcoming extended version of this paper for detailed discussion.

5.1 Efficiently computable graphs

So far we have considered how resource bounds interact with graph computability, though the graph machines themselves have remained arbitrary computable structures. Here we show that requiring the graph machines to be efficiently computable has no effect on the complexity functions that can be graph computed.

Cenzer and Remmel [CR91] showed that an arbitrary computable relational structure is computably isomorphic to some polynomial-time structure. We base our result here on their key ideas.

Definition 5.1. *A graph machine $\mathfrak{M}$ is* **polynomial-time computable** *if (a) its underlying graph $\mathcal{G}$ is polynomial-time computable, in the sense that set G is a polynomial-time computable subset of $\mathbb{N}$, the labels L and colors C are polynomial-time computable subsets of $\mathbb{N}$, and the functions V, E, and γ are polynomial-time computable, and (b) the functions S, α, and T are polynomial-time computable, all with respect to a standard encoding of finite powersets.*

Proposition 5.2. *For every computable graph machine $\mathfrak{M}$, there is a computably isomorphic graph machine $\mathfrak{M}^\circ$ that polynomial-time computable.*

Proof sketch. We will build the polynomial-time computable graph machine $\mathfrak{M}^\circ$ along with its computable isomorphism to $\mathfrak{M}$ simultaneously. Suppose $\{a_i : i \in \mathbb{N}\}$ is a computable increasing enumeration of G. We will define $\{b_i : i \in \mathbb{N}\}$ by induction so that the map $a_i \mapsto b_i$ is the desired isomorphism.

Suppose we have defined $\{b_i : i < n\}$. First consider if any new labels, colors, or states are needed to describe the relationship a_n and $\{a_i : i < n\}$. If there are such labels, colors, or states, we look at the transition table T and consider the length of a minimal computations needed to calculate T on these new labels, colors, and states, along with the ones previously identified. Note that there can be at most finitely many new labels, colors, and states introduced at this stage, and so there is an upper bound on the length of these computations. We then add corresponding new labels, colors, or states to the structure $\mathfrak{M}^\circ$ where the natural numbers associated to these labels, colors, or states are large enough to ensure that the computation of T° remains polynomial-time computable.

Similarly, we consider the length of the computation needed to determine $V(a_n)$, $E(a_i, a_n)$, $E(a_n, a_i)$, and $\gamma(a_n)$ for $i < n$ and choose a natural number to assign to b_n which is large enough to ensure that the corresponding $V^\circ(b_n)$, $E^\circ(b_i, b_n)$, $E^\circ(b_n, b_i)$, and $\gamma^\circ(b_n)$ for $i < n$ are all polynomial-time computable. One can verify that this yields the desired polynomial-time computable graph machine and computable isomorphism. □

In particular, because computably isomorphic graph machines compute functions of the same Turing degrees, in Corollary 3.9 and Theorem 3.10 we may obtain the stated graph computable functions via graph machines whose underlying graphs are polynomial-time computable. Likewise, in Theorem 4.12, we may take the underlying graph $\mathcal{H}_X$ to be polynomial-time computable.

5.2 Symmetric graphs

Definition 5.3. *The graph $\mathcal{G}$ is* **symmetric** *if $E(v, w) = E(w, v)$ for all $v, w \in G$.*

Proposition 5.4. *Every graph computable function is $\mathcal{G}_S$-computable for some symmetric graph $\mathcal{G}_S$.*

Proof sketch. Given a computable graph $\mathcal{G}$ and a $\mathcal{G}$-machine $\mathfrak{M}$, define $\mathcal{G}_S$ to be the *symmetric* graph with underlying set $G_S = G \cup \{e_{(v,w)}, e^*_{(v,w)} : v, w \in G\}$, set of colors C, and edge coloring E_S defined such that

$$\begin{aligned} E(v,w) &= E_S(v, e_{(v,w)}) = E_S(e_{(v,w)}, e^*_{(v,w)}) \\ &= E_S(e^*_{(v,w)}, w) = E_S(v, e^*_{(v,w)}) \end{aligned}$$

for all $v, w \in G$.

In particular, whenever $c \in E(v,w)$ we have that (i) $e^*_{(v,w)}$ is connected to w via an edge of color c, and (ii) there are paths of length 1 and length 2 (of color c) connecting v and $e^*_{(v,w)}$.

We now define $\mathfrak{M}_S$ to be the $\mathcal{G}_S$-machine that simulates $\mathfrak{M}$ as follows. Every three timesteps of $\mathfrak{M}_S$ corresponds to one timestep of $\mathfrak{M}$. State transitions in $\mathfrak{M}_S$ between elements of G are the same as in $\mathfrak{M}$ (except that each transition takes three timesteps to be processed). However, any time that a vertex v sends a c-pulse to w within $\mathfrak{M}$, the corresponding pulse in $\mathfrak{M}_S$ instead goes from v to both $e_{(v,w)}$ and $e^*_{(v,w)}$. When $e_{(v,w)}$ receives a c-pulse, it sends a c-pulse to $e^*_{(v,w)}$. However, $e^*_{(v,w)}$ only sends a c-pulse to w after it has itself received c-pulses in *two* successive timesteps (i.e., first one from v, and then one from $e_{(v,w)}$).

In this way, in $\mathfrak{M}_S$ when v sends out a c-pulse, this causes (three timesteps later) a c-pulse to reach all vertices $w \in G$ for which $c \in E(v,w)$. However, even though in $\mathcal{G}_S$ the relation E_S is symmetric, when v sends out a c pulse it will not reach any $w \in G$ such that $E(w,v)$ holds. Hence the behavior of $\mathfrak{M}_S$ within G simulates the (slowed-down) behavior of $\mathfrak{M}$. $\square$

6 Representations of other computational models via graph machines

We now describe several other models of computation, mainly on graphs, and describe how they can be viewed as special cases of graph machines. These examples provide further evidence for graph machines being a universal model of computation on graphs.

6.1 Ordinary Turing machines

We begin by showing how to simulate an ordinary Turing machine by a graph Turing machine.

Let M be a Turing machine (with a finite set of states and finite transition function, as usual). Then there is an equivalent graph machine whose underlying graph represents a Turing machine tape, as we now describe. The underlying graph is a one-sided chain with underlying set $\{-1\} \cup \mathbb{N}$. Every vertex in $\mathbb{N}$ has degree 2 (with edges both to and from its predecessor on the left and successor on the right), while -1 is connected only to 0. There are two labels, one of which holds of all of $\mathbb{N}$ and other of which holds at -1.

Any vertex of $\mathbb{N}$ which is in the initial state and has not received any pulse does nothing. When the vertex -1 is in the initial state and displays 1, it sends a pulse to 0 to signal the creation of the read/write head "above 0". The presence of a head above a vertex is encoded via the state of that vertex. The lookup table of the graph machine will ensure that there is a unique head, above precisely one vertex, at each timestep following any starting configuration with 1 displayed at -1.

At any subsequent timestep, only the vertex with the head will send out pulses. Between two any adjacent vertices in $\mathbb{N}$, there are sufficiently many edge colors to transmit the current state of M, and so at each timestep, the vertex with the head uses the transition function of M to determine the location of the head at the next time step, and (if the head needs to move according to M) transmits the appropriate signal (containing the current state of M as well as signaling that the head is above it now), to its neighbor, and accordingly adjusts its own state. If the head does not need to move, the vertex with the head merely sets its own state and displayed symbol according to the transition function of M.

One can show that not only does this embedding yield a graph machine that computes functions of the same Turing degree as the original Turing machine M, but that the output of this graph machine produces (on $\mathbb{N}$) the exact same function as M, moreover via a weak bisimulation (in which the function is computed with merely a small linear time overhead).

The doubly-infinite one-dimensional read/write tape of an ordinary Turing machine has cells indexed by $\mathbb{Z}$, the free group on one generator, and in each timestep the head moves according to the generator or its inverse. This interpretation of a Turing machine as a $\mathbb{Z}$-machine has been generalized to H-machines for arbitrary finitely generated groups H by [ABS17], and our simulation above extends straightforwardly to this setting as well.

One might next consider how cleanly one might embed various extensions of Turing machines where the tape is replaced by a graph, such as Kolmogorov–Uspensky machines [KU58], Knuth's pointer machines [Knu68, pp. 462–463], and Schönhage's storage modification machines [Sch80]. For a further discussion of these and their relation to sequential abstract state machines, see [Gur93] and [Gur00].

6.2 Cellular automata

We now consider cellular automata; for background, see, e.g., the book [TM87]. Cellular automata (which we take to be always finite-dimensional, finite-radius and with finitely-many states) can be naturally simulated by graph machines, moreover of constant degree. In particular, Corollary 4.9 applies to this embedding.

Not only is the evolution of every such cellular automata computable (moreover via this embedding as a graph machine), but there are particular automata whose evolution encodes the behavior of a universal Turing machine ([Coo04] and [WN09]). Several researchers have also considered the possibility of expressing intermediate Turing degrees via this evolution ([Bal04], [Coh02], and [Sut03]). Analogously, one might ask which Turing degrees can be expressed in the evolution of graph machines.

We now describe this embedding. Cells of the automata are taken to be the vertices of the graph, and cells are connected to its "neighbors" (other cells within the given radius) by a collection of edges (of the graph) whose labels encode their relative position (e.g., "1 to the left of") and all possible cellular automaton states. (In particular, every vertex has the same finite degree.) The displayed symbol of each vertex encodes the cellular automaton state of that cell.

The rule of the cellular automaton is encoded in the lookup table so as to achieve the following: At the beginning of each timestep, each cell announces its state by virtue of its vertex sending out a pulse to the vertices of the neighboring cells, along edges whose labels encode this state. (If during some timestep a vertex does not receive a pulse from the direction corresponding to a neighboring cell, then it assumes that the vertex corresponding to the neighboring cell is in its initial state and is displaying 0.) Each vertex then updates its displayed symbol based on how the corresponding cell should update its state, based on the states of its neighbors, according to the rule of the original cellular automaton.

Note that this encoding produces a bisimulation between the original cellular automaton and the graph machine built based on it.

6.3 Parallel graph dynamical systems

Parallel graph dynamical systems [AMV15b] can be viewed as essentially equivalent to the finite case of graph Turing machines, as we now describe. Finite cellular automata can also be viewed as a special case of parallel graph dynamical systems, as can finite boolean networks, as noted in [AMV15b, §2.2]. For more on parallel graph dynamical systems, see [AMV15a], [AMV15b], and [BCZ04].

A parallel graph dynamical system specifies a finite (possibly directed) graph and an evolution operator (determining how a given tuple of states for the vertices of the graph transitions into a new tuple). The evolution operator is required to be local, in the sense of determining the new state of a vertex given only its state and those of its adjacencies.

We may embed a parallel graph dynamical system as a graph machine similarly to how we embedded cellular automata above. However, because different vertices have non-isomorphic neighborhoods, we can no longer label edges connecting a vertex to a given neighbor based on the "direction" of this neighbor. We therefore require a different collection of label types for every vertex of the graph, which are used to signal the source of the edge.

This embedding produces a bisimulation between the given parallel graph dynamical system and the graph machine built from it. Note that this embedding also works for the natural infinite extension of parallel graph dynamical systems, where each vertex is required to have finite in-degree. This restriction on in-degree ensures that each vertex of the corresponding graph machine has only finitely many edge colors, even though the set of all edge colors may be infinite.

In contrast, it is not immediately clear how best to encode an arbitrary (parallel) abstract state machine [BG03] as a graph Turing machine (due to the higher arity relations of the ASM).

7 Possible extensions

We now describe several possible extensions of graph Turing machines that may be worth studying.

It would be interesting to develop a non-deterministic version of graph Turing machines, where certain pulses are allowed, but not required, to be sent. We expect that such a framework would naturally encompass an infinitary version of Petri nets [Pet77], as well as the machines described by [AAB$^+$14].

Another interesting extension would be a randomized version of graph Turing machines, where the pulses fire independently according to a specified probability distribution. In this setting, one could then study the joint distribution of overall behavior. Randomized graph Turing machines might encompass dynamical Bayesian networks [Mur02] and other notions of probabilistic computation on a graph.

Our framework involves underlying graphs that are specified before the computation occurs. Is there a natural way to extend our framework to allow for new nodes to be added, destroyed, duplicated, or merged? Such an extension might naturally encompass infinitary generalizations of models of concurrency and parallelism based on graph rewrite rules, such as interaction nets [Laf90] and bigraphs [Mil09].

8 Open questions

We conclude with several open questions.

- Does every Turing degree below $\mathbf{0}^{(\omega)}$ contain a graph computable function? So far, we merely know that every arithmetical degree contains a graph computable function — but there are Turing degrees below $\mathbf{0}^{(\omega)}$ that are not below $\mathbf{0}^{(n)}$ for any $n \in \mathbb{N}$.

- Are there graph computable functions that are not graph computable in constant time? In particular, can $\mathbf{0}^{(\omega)}$ be graph computed in constant time? (The construction in Theorem 3.10 is linear-time.)

- Are there graph computable functions that are not graph computable by a graph machine with finitary underlying graph?

Acknowledgements

The authors would like to thank Tomislav Petrović, Linda Brown Westrick, and the anonymous referees of earlier versions for helpful comments.

References

[AAB$^+$14] D. Angluin, J. Aspnes, R. A. Bazzi, J. Chen, D. Eisenstat, and G. Konjevod, *Effective storage capacity of labeled graphs*, Inform. and Comput. **234** (2014), 44–56.

[ABS17] N. Aubrun, S. Barbieri, and M. Sablik, *A notion of effectiveness for subshifts on finitely generated groups*, Theoretical Computer Science (2017).

[AMV15a] J. A. Aledo, S. Martinez, and J. C. Valverde, *Graph dynamical systems with general Boolean states*, Appl. Math. Inf. Sci. **9** (2015), no. 4, 1803–1808.

[AMV15b] ———, *Parallel dynamical systems over graphs and related topics: a survey*, J. Appl. Math. (2015), no. 594294.

[Bal04] J. Baldwin, *Review of* A New Kind of Science *by Stephen Wolfram*, Bull. Symbolic Logic **10** (2004), no. 1, 112–114.

[BCZ04] C. L. Barrett, W. Y. C. Chen, and M. J. Zheng, *Discrete dynamical systems on graphs and Boolean functions*, Math. Comput. Simulation **66** (2004), no. 6, 487–497.

[BG03] A. Blass and Y. Gurevich, *Abstract state machines capture parallel algorithms*, ACM Trans. Comput. Log. **4** (2003), no. 4, 578–651.

[Coh02] H. Cohn, *Review of* A New Kind of Science *by Stephen Wolfram*, MAA Reviews (2002).

[Coo04] M. Cook, *Universality in elementary cellular automata*, Complex Systems **15** (2004), no. 1, 1–40.

[CR91] D. Cenzer and J. Remmel, *Polynomial-time versus recursive models*, Ann. Pure Appl. Logic **54** (1991), no. 1, 17–58.

[Gur93] Y. Gurevich, *Kolmogorov machines and related issues*, Current Trends in Theoretical Computer Science, World Scientific Series in Computer Science, vol. 40, World Scientific, 1993, pp. 225–234.

[Gur00] ———, *Sequential abstract-state machines capture sequential algorithms*, ACM Trans. Comput. Log. **1** (2000), no. 1, 77–111.

[Knu68] D. E. Knuth, *The art of computer programming. Vol. 1: Fundamental algorithms*, Addison-Wesley, 1968.

[KU58] A. N. Kolmogorov and V. A. Uspensky, *On the definition of an algorithm*, Uspekhi Mat. Nauk **13** (1958), no. 4, 3–28.

[Laf90] Y. Lafont, *Interaction nets*, 17th Ann. ACM Symp. Principles of Programming Languages (POPL) (F. E. Allen, ed.), ACM Press, 1990, pp. 95–108.

[Lov09] L. Lovász, *Very large graphs*, Current developments in mathematics, 2008, Int. Press, Somerville, MA, 2009, pp. 67–128.

[Mil09] R. Milner, *The space and motion of communicating agents*, Cambridge University Press, 2009.

[Mur02] K. P. Murphy, *Dynamic Bayesian networks: Representation, inference and learning*, Ph.D. thesis, University of California, Berkeley, 2002, p. 268.

[Pet77] J. L. Peterson, *Petri nets*, ACM Comput. Surv. **9** (1977), no. 3, 223–252.

[Sch80] A. Schönhage, *Storage modification machines*, SIAM J. Comput. **9** (1980), no. 3, 490–508.

[Soa87] R. I. Soare, *Recursively enumerable sets and degrees*, Perspectives in Mathematical Logic, Springer-Verlag, Berlin, 1987.

[Sut03] K. Sutner, *Cellular automata and intermediate degrees*, Theoret. Comput. Sci. **296** (2003), no. 2, 365–375.

[TM87] T. Toffoli and N. Margolus, *Cellular automata machines: a new environment for modeling*, MIT Press, Cambridge, MA, 1987.

[WN09] D. Woods and T. Neary, *The complexity of small universal Turing machines: a survey*, Theoret. Comput. Sci. **410** (2009), no. 4-5, 443–450.