Simplified Game of Life: Algorithms and Complexity

Krishnendu Chatterjee
Institute of Science and Technology, Klosterneuburg, Austria

Rasmus Ibsen-Jensen
University of Liverpool, UK

Ismaël Jecker
Institute of Science and Technology, Klosterneuburg, Austria

Jakub Svoboda
Institute of Science and Technology, Klosterneuburg, Austria

Abstract

Game of Life is a simple and elegant model to study dynamical system over networks. The model consists of a graph where every vertex has one of two types, namely, dead or alive. A configuration is a mapping of the vertices to the types. An update rule describes how the type of a vertex is updated given the types of its neighbors. In every round, all vertices are updated synchronously, which leads to a configuration update. While in general, Game of Life allows a broad range of update rules, we focus on two simple families of update rules, namely, underpopulation and overpopulation, that model several interesting dynamics studied in the literature. In both settings, a dead vertex requires at least a desired number of live neighbors to become alive. For underpopulation (resp., overpopulation), a live vertex requires at least (resp. at most) a desired number of live neighbors to remain alive. We study the basic computation problems, e.g., configuration reachability, for these two families of rules. For underpopulation rules, we show that these problems can be solved in polynomial time, whereas for overpopulation rules they are PSPACE-complete.

2012 ACM Subject Classification Theory of computation

Keywords and phrases game of life, cellular automata, computational complexity, dynamical systems

Digital Object Identifier 10.4230/LIPIcs.MFCS.2020.22

Related Version A full version of the paper is available at `https://arxiv.org/abs/2007.02894`.

Funding *Krishnendu Chatterjee*: The research was partially supported by the Vienna Science and Technology Fund (WWTF) Project ICT15-003.
Ismaël Jecker: This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 754411.

1 Introduction

Game of Life is a well-known model to study dynamics over networks. We consider the classical model of Game of Life and study two simple update rules for which we establish algorithms and computational complexity. We start with a description of dynamical systems, then explain Game of Life and our simplified rules, and finally state our main results.

Dynamical systems. A dynamical system describes a set of rules updating the state of the system. The study of dynamical systems and computational questions related to them is a core problem in computer science. Some classic examples of dynamical systems are the following: (a) a set of matrices that determine the update of the state of the dynamical system [4, 14]; (b) a stochastic transition matrix that determines the state update (the classical model of Markov chains) [11]; (c) dynamical systems that allows stochastic and

non-deterministic behavior (aka Markov decision processes) [9, 15, 1]; and so on. To study dynamics over networks the two classical models are Game of Life [10, 3] and cellular automata [20].

Game of Life. Game of Life is a simple yet elegant model to study dynamical systems over networks. The network is modeled as a graph where every vertex has one of two types, namely, dead or alive. A configuration (or state of the system) is a mapping of the vertices to the types. An update rule describes how the type of a vertex is updated given the types of its neighbors. In every round, all vertices are updated synchronously, leading to a configuration update. In Game of Life, the update rules are deterministic, hence the configuration graph is deterministic. In other words, from each starting configuration, the updates lead to a finite path followed by a cycle in the configuration graph. While in Game of Life the successor of a state only depends on its number of neighbors of each type, in the more general model of cellular automata the update rule can also distinguish among the positions of the neighbors.

Simplified rules. While the update rules in Game of Life are quite general, in this work, we focus on two simplified rules, namely, *underpopulation rules* and *overpopulation rules*:

- *Underpopulation rule.* According to an underpopulation rule, a dead vertex becomes alive if it has at least i_0 live neighbors, and remains dead otherwise; a live vertex remains alive if it has at least i_1 live neighbors, and becomes dead otherwise.
- *Overpopulation rule.* According to an overpopulation rule, a dead vertex becomes alive if it has at least i_0 live neighbors, and remains dead otherwise; a live vertex remains alive if it has at most i_1 live neighbors, and becomes dead otherwise.

See Section 2 for the formal details of the definition, and a detailed comparison of our setting with cellular automata, and Conway's original Game of Life.

Motivation. While we consider simpler rules, we study their effect on any type of graph, contrary to cellular automata that focus on grids. This allows us to model several complex situations studied in the literature. For example, the underpopulation rule models the spread of ideas, where a person adopts a new idea only if sufficiently many neighbors adopt it, or study bandwagon effects where an item is considered useful if sufficiently many neighbors use it. In contrast, the overpopulation rule models anti-coordination effects where the goal is to avoid a popular pub, or model snob effects where individuals discard a fashion if too many neighbors have adopted it. See Section 3 for further details.

Basic computational problems. We study two basic computational problems for the underpopulation and overpopulation rule. The first computational problem is the *configuration reachability* question which asks, given an initial configuration and a target configuration, whether the target configuration is reachable from the initial configuration. The second computational problem is the *long-run average* question, which asks, given an initial configuration, what is the long-run average of the number of live vertices. Note that in the configuration graph, any initial configuration is the source of a finite path followed by a cycle. The long-run average question asks about the average number of live vertices in the cycle.

Our contributions. Our main contributions are algorithms and complexity results for the two families of rules. First, for the underpopulation rules, we present polynomial time algorithms for both computational problems for all graphs. Thus, we identify a simple update rule in Game of Life, that can model several interesting scenarios, for which we present

efficient algorithms. Second, for the overpopulation rules, we show that both computational problems are PSPACE-complete. Note that the PSPACE upper bound holds for general update rules for Game of Life, hence the main contribution is the PSPACE hardness proof. Moreover, we show that the PSPACE hardness even holds for regular graphs with a constant degree. Note that the difference between underpopulation and overpopulation is minimal (one inequality reversed), yet we show that while efficient algorithms exist for underpopulation rules, the computational problems for overpopulation rules are intractable.

2 Preliminaries

Given a finite alphabet A, we denote by A^* the set of finite sequences of elements of A, and by A^ω the set of infinite sequences of elements of A. The elements of A^* and A^ω are called *words* over A. The *length* of a word $w = a_1a_2a_3 \ldots \in A^* \cup A^\omega$ is its number of letters, denoted $|w| \in \mathbb{N} \cup \{+\infty\}$. A *factor* of w is a sequence of consecutive letters of w. For every $0 \leq i \leq j \leq |w|$, we denote by $w[i,j]$ the factor $a_{i+1}a_{i+2} \ldots a_j$ of w.

A (finite) *graph* is a pair $G = (V, E)$ composed of a finite set of *vertices* V and a set of *edges* $E \subseteq V \times V$ that are pairs of vertices. A *walk* of G is a sequence $\rho = s_1, s_2, s_3 \ldots \in V^* \cup V^\omega$ such that each pair of consecutive vertices is an edge: $(s_i, s_{i+1}) \in E$ for every $1 \leq i < |\rho|$. A (simple) *path* is a walk whose vertices are all distinct. A (simple) *cycle* is a walk in which the first and last vertices are identical, and all the other vertices are distinct. A graph is called *undirected* if its set of edges is symmetric: $(s,t) \in E \Leftrightarrow (t,s) \in E$. Two vertices of an undirected graph are called *neighbors* if they are linked by an edge.

2.1 Configurations and update rules

A *configuration* of a graph is a mapping of its vertices into the set of states $\{0,1\}$. We say that a vertex is *dead* if it is in state 0, and *alive* if it is in state 1. An *update rule* $\mathcal{R}$ is a set of *deterministic*, *time-independant*, and *local* constraints determining the evolution of configurations of a graph: the successor state of each vertex is determined by its current state and the states of its neighbors. We define an update rule formally as a pair of functions: for each $q \in \{0,1\}$, the *state update function* ϕ_q maps any possible neighborhood configuration to a state in $\{0,1\}$. The successor state of a vertex in state q with neighborhood in configuration c_n is then defined as $\phi_q(c_n) \in \{0,1\}$.

In this work, we study the effect on undirected graphs of update rules definable by state update functions that are *symmetric* and *monotonic* (configurations are partially ordered by comparing through inclusion their subsets of live vertices). In this setting, a vertex is not able to differentiate its neighbors, and has to determine its successor state by comparing the number of its live neighbors with a threshold. These restrictions give rise to four families of rules, depending on whether ϕ_0 and ϕ_1 are monotonic increasing or decreasing. We study the two families corresponding to increasing ϕ_0, the two others can be dealt with by using symmetric arguments.

Underpopulation. An *underpopulation (update) rule* $\mathcal{R}^+(i_0, i_1)$ is defined by two thresholds: $i_0 \in \mathbb{N}$ is the minimal number of live neighbors needed for the birth of a dead vertex, and $i_1 \in \mathbb{N}$ is the minimal number of live neighbors needed for a live vertex to stay alive. Formally, the successor $\phi_q(m)$ of a vertex currently in state $q \in \{0,1\}$ with $m \in \mathbb{N}$ live neighbors is

$$\phi_0(m) = \begin{cases} 0 \text{ if } m < i_0; \\ 1 \text{ if } m \geq i_0. \end{cases} \qquad \phi_1(m) = \begin{cases} 0 \text{ if } m < i_1; \\ 1 \text{ if } m \geq i_1. \end{cases}$$

This update rule is symmetric and monotonic.

Overpopulation. An *overpopulation (update) rule* $\mathcal{R}^-(i_0, i_1)$ is defined by two thresholds: $i_0 \in \mathbb{N}$ is the minimal number of live neighbors needed for the birth of a dead vertex, and $i_1 \in \mathbb{N}$ is the maximal number of live neighbors allowing a live vertex to stay alive. Formally, the successor $\phi_q(m)$ of a vertex currently in state $q \in \{0, 1\}$ with $m \in \mathbb{N}$ live neighbors is

$$\phi_0(m) = \begin{cases} 0 \text{ if } m < i_0; \\ 1 \text{ if } m \geq i_0. \end{cases} \quad \phi_1(m) = \begin{cases} 0 \text{ if } m > i_1; \\ 1 \text{ if } m \leq i_1. \end{cases}$$

This update rule is symmetric and monotonic.

Basic computational problems. To gauge the complexity of an update rule, we study two corresponding computational problems. Formally, given an update rule $\mathcal{R}$ and a graph G, the *configuration graph* $\mathrm{C}(G, \mathcal{R})$ is the (directed) graph whose vertices are the configurations of G, and whose edges are the pairs (c, c') such that the configuration c' is successor of c according to the update rule $\mathcal{R}$. Note that $\mathrm{C}(G, \mathcal{R})$ is finite since G is finite. Moreover, since the update rule $\mathcal{R}$ is deterministic, every vertex of the configuration graph is the source of a single infinite walk composed of a finite path followed by a cycle.

- The *configuration reachability problem*, denoted Reach, asks, given a graph G, an initial configuration c_I, and a final configuration c_F, whether the walk in $\mathrm{C}(G, \mathcal{R})$ starting from c_I eventually visits c_F.
- The *long-run average problem*, denoted Avg, asks, given a threshold $\delta \in [0, 1]$, a graph G, and an initial configuration c_I, whether δ is strictly smaller than the average ratio of live vertices in the configurations that are part of the cycle in $\mathrm{C}(G, \mathcal{R})$ reached from c_I.

2.2 Comparison to other models

We show similarities and differences between our update rules and similar models.

Cellular automata. Cellular automata study update rules defined on (usually infinite) grid graphs [20]. Compared to the setting studied in this paper, more rules are allowed since neither symmetry nor monotonicity is required, yet underpopulation and overpopulation rules are not subcases of cellular automata, as they are defined for any type of graph, not only grids. To provide an easy comparison between the update rules studied in this paper and some well-studied cellular automata, we now define Rule 54 and Rule 110 (according to the numbering scheme presented in [19]) using the formalism of this paper.

1. Rule 54 [5, 13] coincides with the overpopulation rule $\mathcal{R}^-(1, 0)$ applied to the infinite unidimensional linear graph. A dead vertex becomes alive if one of its neighbors is alive, and a live vertex stays alive only if both its neighbors are dead. Formally, the successor $\phi_q(m)$ of a vertex currently in state $q \in \{0, 1\}$ with $m \in \{0, 1, 2\}$ live neighbors is

   $$\phi_0(m) = \begin{cases} 0 \text{ if } m = 0; \\ 1 \text{ if } m \geq 1. \end{cases} \quad \phi_1(m) = \begin{cases} 0 \text{ if } m \geq 1; \\ 1 \text{ if } m = 0. \end{cases}$$

   This update rule is symmetric and monotonic. It can be used to model logical gates [13], and is conjectured to be Turing complete.
2. Rule 110 [8] is defined over the infinite unidimensional linear graph. A dead vertex copies the state of its right neighbor, and a live vertex stays alive as long as at least one of its neighbors is alive. Formally, the successor $\phi_q(\ell, r)$ of a vertex currently in state $q \in \{0, 1\}$ with left neighbor in state $\ell \in \{0, 1\}$ and right neighbor in state $r \in \{0, 1\}$ is

   $$\phi_0(\ell, r) = r; \quad \phi_1(\ell, r) = \begin{cases} 0 \text{ if } \ell = r = 0; \\ 1 \text{ otherwise.} \end{cases}$$

   This update rule is monotonic, but not symmetric. It is known to be Turing complete.

Game of Life. Game of Life requires update rules that are symmetric, but not necessarily monotonic. The most well known example is Conway's Game of Life [10, 3], that has been adapted in various ways, for example as 3-D Life [2], or the beehive rule [21]. Conway's game of life studies the evolution of the infinite two-dimensional square grid according to the following update rule: a dead vertex becomes alive if it has exactly three live neighbors, and a live vertex stays alive if it has two or three live neighbors. Formally, the successor $\phi_q(m)$ of a vertex currently in state $q \in \{0,1\}$ with $m \in \mathbb{N}$ live neighbors is

$$\phi_0(m) = \begin{cases} 0 \text{ if } m \neq 3; \\ 1 \text{ if } m = 3. \end{cases} \qquad \phi_1(m) = \begin{cases} 0 \text{ if } m \notin \{2,3\}; \\ 1 \text{ if } m \in \{2,3\}. \end{cases}$$

This update rule is symmetric, but not monotonic. It is known to be Turing complete [3].

3 Motivating Examples

Our dynamics can represent situations where an individual (a vertex) adopts a behavior (or a strategy) only if the behavior is shared by sufficiently many neighboring individuals. Then, the underpopulation setting corresponds to behaviors that are abandoned if not enough neighbors keep on using it, while the overpopulation setting models behaviors that are dropped once they are adopted by too many. We present several examples.

3.1 Underpopulation

Innovation. The problem of spreading innovation is considered in [16, 17]. Initially, a small group of people starts using a new product and if others see it used, they adopt the innovation. In our setting this corresponds to the underpopulation model. The question of determining whether the innovation gets to some key people can be formalised as the configuration reachability problem Reach, and predicting how many people will eventually be using the innovation amounts to solve the long-run average problem Avg. Similar questions are asked in [18], where the authors study how opinions form.

Positive feedback. In the paper [12], the bandwagon and Veblen effects are described. These consider a fact that the demand for an item increases with the number of people using that item. Under this hypothesis, determining the demand corresponds to solve Avg for an underpopulation rule. Many more examples, for example, how people behave depending on what their friends do, can be found in [7]. Anything from emotions to obesity can spread through a network, usually following small modifications of the underpopulation rule.

3.2 Overpopulation

Anticoordination. Imagine that you want to go mushroom hunting. You enjoy the peaceful walk in the forest and love the taste of fried wild mushrooms, or mushroom soup. Mushrooming is a solitary activity and if too many of your neighbors decide to go mushrooming too, they annoy you, and you find fewer mushrooms in already searched forest. So, if you were not mushrooming the day before you can get convinced that the mushrooms are growing by some neighbors that show you baskets full of delicious mushrooms. However, if you decide to go and see too many people there, you get discouraged and do not go the next day.

This behavior is called anticoordination and was described in [6], it more generally describes optimal exploitation of resources. The questions here are: does some set of people go mushroom hunting, how many people will be mushroom hunting? The overpopulation closely corresponds to this with Reach and Avg answering the questions.

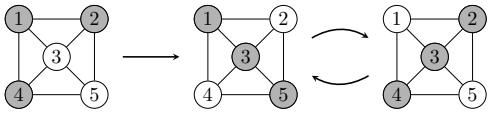

Figure 1 Evolution of a graph under the underpopulation rule $\mathcal{R}^+(2,2)$. Live vertices are gray.

Snob effect. Many items are desirable because they are expensive, or unique. This behavior was observed in [12]. People start doing something, but if too many people do it, it loses appeal. For instance fashion works this way for all of us: People get inspired by what they see, but if too many people wear the same outfit, they change it.

4 Underpopulation: PTIME Algorithm

In this section, we study underpopulation rules: a vertex comes to life if it has sufficiently many living neighbours, and then still requires enough living neighbours to stay alive. Our main result is as follows:

▶ **Theorem 1.** *For every underpopulation rule, the reachability and long-run average problems are decidable in polynomial time.*

The above result depends on two key propositions. Let us start by having a look at an example. Figure 1 presents successive configurations of a graph where each vertex requires at least 2 living neighbours in order to be alive in the next step. The resulting behaviour is quite simple: after the initial step, the graph keeps on oscillating between two configurations: the middle vertex has reached a stable state, and the other vertices alternate between being dead and alive. We show that the behaviour of graphs following underpopulation rules can actually never be much more complicated than this. First, no huge cycle of configurations can happen.

▶ **Proposition 2.** *For every $i_0, i_1 \in \mathbb{N}$ and every undirected graph G, the configuration graph $\mathrm{C}(G, \mathcal{R}^+(i_0, i_1))$ admits no simple cycle of length bigger than two.*

Moreover, a cycle is always reached early in the process.

▶ **Proposition 3.** *For every $i_0, i_1 \in \mathbb{N}$ and every undirected graph G, the configuration graph $\mathrm{C}(G, \mathcal{R}^+(i_0, i_1))$ admits no simple path of length $2|E| + 2(i_0 + i_1 + 1)|V| + 4$ or more.*

We now present the proof of Theorem 1, by showing that Proposition 3 yields a polynomial time algorithm for both Reach and Avg.

Proof of Theorem 1. Since underpopulation rules are deterministic, once a vertex is repeated in a walk in $\mathrm{C}(G, \mathcal{R}^+(i_0, i_1))$, no new configuration can be visited. Therefore, Proposition 3 bounds polynomially the number of configurations reachable from an initial configuration. Since computing the successor of a configuration and checking the equality of configurations can both be done in polynomial time, we obtain the following polynomial time algorithms solving Reach and Avg: first, we list all the configurations reachable from the initial configuration, then we check if the final configuration is part of it, respectively if the rate of live vertices in the reached loop is higher than the required threshold. ◀

The remainder of this section is devoted to the proof of Propositions 2 and 3. Let us fix an underpopulation rule $\mathcal{R}^+(i_0, i_1)$, a graph $G = (V, E)$, and an initial configuration of G. Our proofs rely on a key lemma that sets a bound on the number of times a vertex of

G switches its state between two configurations separated by two time steps. We begin by introducing some technical concepts and notations, then we state our key lemma (Subsection 4.1). Afterwards, we proceed with the formal proofs of Propositions 2 and 3 (Subsection 4.2).

4.1 Key Lemma

We begin by introducing some auxiliary notation, and then we state our key lemma.

Histories. The *history* of a vertex $s \in V$ is the infinite word $\tau_s \in \{0,1\}^\omega$ whose letters are the successive states of s. For instance, in the setting depicted in Figure 1, the histories are:

$$\tau_1 = 1(10)^\omega \quad \tau_2 = (10)^\omega \quad \tau_3 = 01^\omega \quad \tau_4 = (10)^\omega \quad \tau_5 = (01)^\omega$$

The state of vertex 3 stabilises after the first step, and the other four vertices end up oscillating between two states. The proofs of this section rely on counting, in the histories of G, the number of factors (sequence of consecutive letters) matching some basic regular expressions. In order to easily refer to these numbers, we introduce the following notations. Let us consider the alphabet $\{0,1,?\}$, where ? is a wildcard symbol matching both 0 and 1. Given an integer $m \in \mathbb{N}$ and a word $y \in \{0,1,?\}^*$ of size $n \leq m$, we denote by $[y]_m \in \mathbb{N}$ the number of factors of the prefixes $\tau_s[0,m]$, $s \in V$, that match the expression y. Formally,

$$[y]_m = \big|\{(s,i) \in V \times \mathbb{N} | i+n \leq m, \tau_s[i,i+n] = y\}\big|.$$

Key Lemma. We show that we can bound the number of state switches of the vertices of G between two configurations separated by two time steps.

▶ **Lemma 4.** *For every $m \geq 3$, the equation $[1?0]_m + [0?1]_m \leq 2|E| + 2(i_0 + i_1 + 1)|V|$ holds.*

Proof sketch. The basic idea is that the current state of a vertex $s \in V$ indirectly contributes to s having the same state two steps in the future: let us suppose that s is alive at time $i \in \mathbb{N}$. Then s contributes towards making its neighbours alive at time $i+1$, which in turn contribute towards making their own neighbours, including s, alive at time $i+2$. This idea can be formalised by studying in details the number of occurrences of diverse factors in the histories of G.

4.2 Proof of Proposition 2 and Proposition 3

Using Lemma 4, we are finally able to demonstrate the two results left unproven at the beginning of this section. First, we prove Proposition 2. Note that the proof only uses the fact that $[1?0]_m + [0?1]_m$ is bounded, and not the precise bound.

▶ **Proposition 2.** *For every $i_0, i_1 \in \mathbb{N}$ and every undirected graph G, the configuration graph $\mathrm{C}(G, \mathcal{R}^+(i_0,i_1))$ admits no simple cycle of length bigger than two.*

Proof. Since Lemma 4 bounds the value of $[1?0]_m + [0?1]_m$ for every $m \in \mathbb{N}$, then for every vertex $s \in V$, factors of the form 1?0 or 0?1 only appear in a bounded prefix of τ_s. Therefore, the corresponding infinite suffix only contains factors of the form 1?1 or 0?0, which immediately yields that the periodic part of τ_s is of size either 1 or 2. Since this is verified by every vertex, this shows that under the underpopulation rule $\mathcal{R}^+(i_0,i_1)$, the graph G either reaches a stable configuration, or ends up alternating between two configurations. ◀

Finally, we prove Proposition 3. This time, we actually need the precise bound exposed by Lemma 4.

▶ **Proposition 3.** *For every $i_0, i_1 \in \mathbb{N}$ and every undirected graph G, the configuration graph $\mathrm{C}(G, \mathcal{R}^+(i_0, i_1))$ admits no simple path of length $2|E| + 2(i_0 + i_1 + 1)|V| + 4$ or more.*

Proof. We prove that if no cycle is completed in the first $2i$ steps of the process for some $0 < i \in \mathbb{N}$, then the histories of G admit at least $2i - 2$ factors of the form 1?0 or 0?1. Since $[1?0]_{2i+2} + [0?1]_{2i+2}$ is smaller than $2|E| + 2(i_0 + i_1 + 1)|V|$ by Lemma 4, this implies that $2i \leq 2|E| + 2(i_0 + i_1 + 1)|V| + 2$, which proves the lemma.

Let $i \in \mathbb{N}$ be a strictly positive integer, and let us suppose that no configuration is repeated amongst the first $2i$ steps of the process. Let us first focus on the sequence of i odd configurations $c_1, c_3, c_5, \ldots, c_{2i-1}$ of G. By supposition, no configuration is repeated, hence for every $1 \leq j \leq i - 1$, at least one vertex has distinct states in the configurations c_{2j-1} and c_{2j+1}. These $i - 1$ changes either consist in the death of a live vertex, counting towards the value $[1?0]_{2i+2}$, or in the birth of a dead vertex, counting towards the value $[0?1]_{2i+2}$. Similarly, focusing on the sequence of i even configurations $c_2, c_4, c_6, \ldots, c_{2i}$ yields $i-1$ distinct occurrences of vertices changing state between two successive positions of even parity, counting towards the value $[1?0]_{2i+2} + [0?1]_{2i+2}$. As a consequence, $2i - 2 \leq [1?0]_{2i+2} + [0?1]_{2i+2}$, hence, by Lemma 4, $2i \leq 2|E| + 2(i_0 + i_1 + 1)|V| + 2$, which concludes the proof. ◀

5 Overpopulation: PSPACE completeness

In this section, we study overpopulation rules: a vertex comes to life if it has sufficiently many living neighbors, and dies if it has too many living neighbors. Our result is in opposition to the result of the previous section:

▶ **Theorem 5.** *The following assertions hold:*

- *For every overpopulation rule, the reachability and long-run average problems are in PSPACE.*
- *For the specific case $\mathcal{R}^-(2, 1)$, the reachability and long-run average are PSPACE-hard.*

▶ Remark 6. The first item of Theorem 5 (the PSPACE upper bound) is straightforward. Our main contribution is item 2 (PSPACE hardness). We present a graph construction simulating a Turing machine. In addition, our basic construction can be modified to ensure that we obtain a regular graph of degree 10.

5.1 General idea for hardness

We create a graph and an initial position simulating a polynomial-space Turing machine. The graph is mostly composed of dead vertices, with some live vertices that carry signals and store data. The graph is regular and consists of **blobs** of vertices. One blob corresponds to one cell of the tape and stores a tape alphabet symbol. Blobs are connected in a row, and at most a single blob is active at every point in time. The active blob receives a signal that corresponds to the state of the Turing machine. It computes the transition function using the received signal and its stored value. The result of the transition function is then used to (1) modify the content of the blob and (2) send the resulting state to the neighboring blob, activating it.

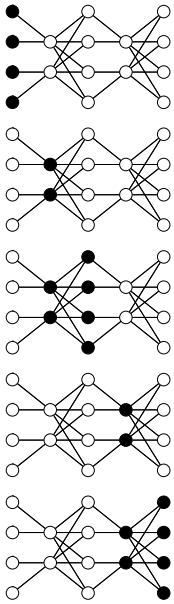

Figure 2 Signal going throught two connected wires (We suppose its left end is output of some gadget).

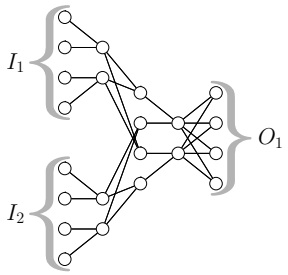

Figure 3 Gate computing AND.

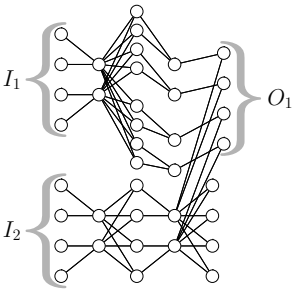

Figure 4 Storage Unit, signal at I_1 toggles state of four vertices to the left of O_1. Signal at I_2 gets to O_1 only if these four vertices are alive.

5.2 Basic gadgets

We describe the gadgets used in the construction. Each gadget g has a constant number of inputs $I_1, I_2, \ldots, I_c$ and outputs $O_1, O_2, \ldots, O_d$ that receive, respectively send, signals. Each input I_i (output O_i) is composed of four vertices that always share the same state. We view live cells as true and dead cells as false, and denote by $I_{i,t} \in \{0,1\}$ ($O_{i,t} \in \{0,1\}$) the value of the input I_i (output O_i) at time t. Each one of our basic gadgets g has an evaluation time t_g, and is determined to realise a function $f_g : \{0,1\}^c \to \{0,1\}^d$. Starting from the *inert* state (i.e, all the vertices are dead), if the c inputs receive some signal (and no new parasite signal is received during the next t_g steps of the process), it computes f_g in t_g steps, broadcast the result through the d outputs, and then goes back to the inert state. We say that g *computes* the function

$$\begin{array}{rccc} f_g : & \{0,1\}^c & \to & \{0,1\}^d, \\ & (I_{1,t}, I_{2,t}, \ldots, I_{c,t}) & \mapsto & (O_{1,t+t_g}, O_{2,t+t_g}, \ldots, O_{d,t+t_g}). \end{array}$$

Moreover, for each gadget we suppose that the input is erased after one step, and in turn the gadget is responsible for erasing its output after one step. Here are the **basic gadgets**:

- The wire transmits a signal. It is evaluated in 2 time steps, has one input I_1, and one output O_1 satisfying $O_{1,t+2} = I_{1,t}$. Several wires can be connected to create a longer wire. Figure 2 illustrates the inner workings of the wire.
- The splitter duplicates a signal. It is evaluated in 2 time steps, has one input I_1, and two outputs O_1, O_2 satisfying $O_{1,t+2} = I_{1,t}$ and $O_{2,t+2} = I_{1,t}$.
- The OR gate computes the logical disjunction. It is evaluated in 4 time steps, has two inputs I_1, I_2, and one output O_1 satisfying $O_{1,t+4} = I_{1,i} \vee I_{2,i}$.

- The AND gate (Figure 3) computes the logical conjunction. It is evaluated in 4 time steps, has two inputs I_1, I_2, and one output O_1 satisfying $O_{1,t+4} = I_{1,t} \wedge I_{2,t}$.
- The NOT gate computes the logical negation. It is evaluated in 4 time steps, has two inputs (a clock signal is required to activate the gate), and one output O_1 satisfying $O_{1,t+4} = \neg I_{2,t} \wedge I_{1,t}$.

To create a Turing machine, we use one more gadget: the storage unit (Figure 4). Contrary to the previous gadgets, it does not necessarily erase itself after use, and can store one bit of information that can be sent upon request. Formally, a storage unit has a state $S \in \{0,1\}$, two inputs I_1, I_2, and one output O_1. The first input is used to modify the current state: if $I_{1,t}$ is true, then the storage unit changes its state in four steps. The second input is used to make the storage unit broadcast its current state: $O_{1,t+4} = S \wedge I_{2,t}$.

Note that every gadget has a fixed number of vertices.

5.3 Functions

Our basic gadgets compute the basic logical operators. We show that combining them yields bigger gadgets that can compute any binary function, with a small restriction: it is not possible to produce a positive signal out of a negative signal. For example, our NOT gate needs a clock signal to be activated. Therefore, we only consider binary functions that map to 0 all the tuples starting with a 0.

▶ **Lemma 7.** *Let $c \in \mathbb{N}$. For every function $f : \{0,1\}^c \to \{0,1\}$ mapping to 0 every tuple whose first component is 0, we can construct a gadget computing f that is composed of $\mathcal{O}(2^c)$ basic gadgets, and is evaluated in $\mathcal{O}(c)$ steps.*

5.4 Simulating the Turing machine

We now show how to simulate a Turing machine with a graph following $\mathcal{R}^-(2,1)$.

▶ **Lemma 8.** *Let T be a Turing machine. For every input u evaluated by T using $C \in \mathbb{N}$ cells of the tape, there exists a bounded degree graph G on $\mathcal{O}(C)$ vertices and an initial configuration c_0 of G such that T stops over the input u if and only if updating c_0 with the overpopulation rule $\mathcal{R}^-(2,1)$ eventually yields the configuration with only dead vertices*

Proof. We suppose that the Turing machine T has a single final state, which can only be accessed after clearing the tape. We present the construction of the graph G simulating T through the following steps. First, we encode the states of T, the tape alphabet, and the transition function in binary. Then, we introduce the notion of blob, the building blocks of G, and we show that blobs are able to accurately simulate the transition function of T. Afterwards, we approximate the size of a blob, and finally we define G.

Binary encoding. Let $T_s \in \mathbb{N}$ be the number of states of T, and $T_a \in \mathbb{N}$ be the size of its tape alphabet. We pick two small integers s and n satisfying $T_s \leq 2^{s-1}$ and $T_a \leq 2^{n-1}$. We encode the states of T as elements of $\{0,1\}^s$, and the alphabet symbols as elements of $\{0,1\}^n$, while respecting the following three conditions: the blank symbol is mapped to 0^n, the final state of T is mapped to 0^s, and all the other states are mapped to strings starting with 1. Then, with respect to these mappings, we modify the transition function of T to:

$$F : \{0,1\}^s \times \{0,1\}^n \to \{0,1\}^s \times \{0,1\}^s \times \{0,1\}^n.$$

Instead of using one bit to denote the movement, we use $2s$ bits to store the state and signify the movement: if the first s bits are zero, the head is moving right; if the second s bits are zero, the head is moving left; if the first $2s$ bits are zero, the computation ended. Moreover, the last n bits of the image of F do not encode the new symbol, but the symmetric difference between the previous and the next symbol: if the i-th bit of the tape symbol goes from y_i to z_i, then F outputs $d_i = y_i \oplus z_i$ (XOR of these two).

Constructing blobs. As we said at the beginning, the graph G is obtained by simulating each cell of the tape with a blob, which is a gadget storing a tape symbol, and that is able, when receiving a signal corresponding to a state of T, to compute the corresponding result of the transition function. The main components of a blob are as follows.

- Memory: n storage units $(s_1, s_2, \ldots, s_n)$ are used to keep in memory a tape symbol $a \in \{0,1\}^n$ of T.
- Receptor: $2s$ inputs $(I_1, I_2, \ldots, I_{2s})$ are used to receive states $q \in \{0,1\}^s$ of T either from the left or from the right.
- Transmitter: $2s$ outputs $(O_1, O_2, \ldots, O_{2s})$ are used to send states $q \in \{0,1\}^s$ of T either to the right or to the left.
- Transition gadget: using Lemma 7, we create a gadget computing each of the $2s + n$ output bits of F. These gadgets are then combined into a bigger gadget g_F that evaluates them all separately in parallel, and computes the transition function F. Note that g_F is composed of $\mathcal{O}((n+s)2^{n+s})$ basic gadgets, and its evaluation time is $\mathcal{O}(n+s)$.

Blobs are connected in a row to act as a tape: for every $1 \leq i \leq s$, the output O_i of each blob is connected to the input I_i of the blob to its right, and the output O_{s+i} of each blob is connected to the input I_{s+i} of the blob to its left. When receiving a signal, the blob transmits the received state and the tape symbol stored in memory to the transition gadget g_F, which computes the corresponding transition, and then apply its results. We now detail this inner behavior. Note that when a gadget is supposed to receive simultaneously a set of signals coming from different sources, it is always possible to add wires of adapted length to ensure that all the signals end up synchronized.

Simulating the transition function. In order to simulate the transition function of T, a blob acts according to the three following steps:

1. Transmission of the state. A blob can receive a state either from the left (through inputs $I_1, I_2, \ldots, I_s$) or from the right (through inputs $I_{s+1}, I_{s+2}, \ldots, I_{2s}$), but not from both sides at the same time, since at every point in time there is at most one active state. Therefore, if for every $1 \leq i \leq s$ we denote by x_i the disjunction of the signals received by I_i and I_{s+i}, then the resulting tuple $(x_1, x_2, \ldots, x_s)$ is equal to the state received as signal (either from the left or the right), which can be fed to the gadget g_F. Formally, the blob connects, for all $1 \leq i \leq s$, the pair I_i, I_{s+i} to an OR gate whose output is linked to the input I_i of g_F.
2. Transmission of the tape symbol. Since the first component of any state apart from the final state is always 1, whenever a blob receives a state, the component x_1 defined in the previous paragraph has value 1. The tape symbol $(y_1, y_2, \ldots, y_n)$ currently stored in the blob can be obtained by sending, for every $1 \leq i \leq n$, a copy of x_1 to the input I_2 of the storage unit s_i, causing it to broadcast its stored state y_i. The tuple can then be fed to the gadget g_F. Formally, the blob uses n splitters to transmit the result of the OR gate between I_1 and I_{s+1} to the input I_1 of each storage unit. Then, for every $1 \leq i \leq n$, the output O_1 of the storage unit s_i is connected to the input I_{s+i} of g_F.

3. Application of the transition. Upon receiving a state and a tape symbol, g_F computes the result of the transition function, yielding a tuple $(r_1, r_2, \ldots, r_{s+n})$. The blob now needs to do two things: send a state to the successor blob, and update the element of the tape.
   Connecting the output O_i of g_F to the output O_i of the blob for every $1 \leq i \leq 2s$ ensures that the state is sent to the correct neighbor: the values $(r_1, r_2, \ldots, r_s)$ are nonzero if the head is supposed to move to the right, and the outputs $O_1, O_2, \ldots, O_s$ of the blob are connected to the right. Conversely, $(r_{s+1}, r_{s+2}, \ldots, r_{2s})$ is nonzero if the head is supposed to move to the left, and the outputs $O_{s+1}, O_{s+2}, \ldots, O_{2s}$ of the blob are connected to the left.
   Finally, connecting the output O_{2s+i} of g_F to the input I_1 of s_i for all $1 \leq i \leq n$ ensures that the state is correctly updated: this sends the signal d_i to the input I_1 of the storage unit s_i. Since d_i is the difference between the current bit and the next, the state of s_i will change only if it has to.

Size of a blob. To prepare the signal for the transition function and to send the signal to another blob, only $\mathcal{O}(n+s)$ basic gadgets and $\mathcal{O}(n)$ steps are needed. As a consequence, the size of a blob is mainly determined by the size of the transition gadget g_F: one blob is composed of $\mathcal{O}((n+s)2^{n+s})$ basic gadgets of constant size, and evaluating a transition requires $\mathcal{O}(n+s)$ steps. Since n and s are constants (they depend on T, and not on the input u), the blob has constant size. Moreover, all the basic gadgets used in the construction have bounded degree, so the blob also has bounded degree.

Constructing G. Now that we have blobs that accurately simulate the transition function of T, constructing the graph G simulating the behavior of T over the input u is easy: we simply take a row of C blobs (remember that $C \in \mathbb{N}$ is the number of tape cells used by T to process u). Since the size of a blob is constant, G is polynomial in C. We define the initial configuration of G by setting the states of the $|u|$ blobs on the left of the row to the letters of u, and setting the inputs I_1 to I_s of the leftmost blob to the signal corresponding to the initial state of T as if it was already in the process. As explained earlier, the blobs then evolve by following the run of T. If the Turing machine stops, then its tape is empty and the final state is sent. Since in G the final state is encoded by 0^s and the blank symbol is encoded by 0^n, this results in G reaching the configuration where all the vertices are dead. Conversely, if T runs forever starting from the input u, there will always be some live vertices in G to transmit the signal corresponding to the state of T. ◀

Proof of Theorem 5. By Lemma 8, we can reduce any problem solvable by a polynomially bounded Turing machine into Reach, asking whether the configuration with only dead vertices is reached, or into Avg, asking whether the long-run average is strictly above 0. ◀

6 Conclusion

In this work, we identify two simple update rules for Game of Life. We show (in Section 3) that these simple rules can model several well-studied dynamics in the literature. While we show that efficient algorithms exist for the underpopulation rule, the computational problems are PSPACE-hard for the overpopulation rule. An interesting direction for future work would be to consider whether for certain special classes of graphs (e.g., grids) efficient algorithms can be obtained for the overpopulation rule.

References

1 C. Baier and J-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
2 Carter Bays. Candidates for the game of life in three dimensions. *Complex Systems*, 1(3):373–400, 1987.
3 Elwyn R Berlekamp, John H Conway, and Richard K Guy. *Winning Ways for Your Mathematical Plays, Volume 4*. AK Peters/CRC Press, 2004.
4 Vincent D Blondel, Olivier Bournez, Pascal Koiran, and John N Tsitsiklis. The stability of saturated linear dynamical systems is undecidable. *Journal of Computer and System Sciences*, 62(3):442–462, 2001.
5 Nino Boccara, Jamil Nasser, and Michel Roger. Particlelike structures and their interactions in spatiotemporal patterns generated by one-dimensional deterministic cellular-automaton rules. *Physical Review A*, 44(2):866, 1991.
6 Arthur W. Brian. Inductive reasoning and bounded rationality. *American Economic Review*, 84(2):406–11, 1994. doi:10.1109/4235.771167.
7 Nicholas A. Christakis and James H. Fowler. *Connected: The Surprising Power of Our Social Networks and How They Shape Our Lives – How Your Friends' Friends' Friends Affect Everything You Feel, Think, and Do*. New York: Little, Brown Spark, 2009.
8 Matthew Cook. Universality in elementary cellular automata. *Complex systems*, 15(1):1–40, 2004.
9 J. Filar and K. Vrieze. *Competitive Markov Decision Processes*. Springer-Verlag, 1997.
10 Martin Gardener. Mathematical games: The fantastic combinations of John Conway's new solitaire game" life,". *Scientific American*, 223:120–123, 1970.
11 J.G. Kemeny, J.L. Snell, and A.W. Knapp. *Denumerable Markov Chains*. D. Van Nostrand Company, 1966.
12 Harvey Leibenstein. Bandwagon, snob, and Veblen effects in the theory of consumers' demand. *The Quarterly Journal of Economics*, 64(2):183–207, 1950. doi:10.2307/1882692.
13 Genaro Juárez Martínez, Andrew Adamatzky, and Harold V McIntosh. Phenomenology of glider collisions in cellular automaton rule 54 and associated logical gates. *Chaos, Solitons & Fractals*, 28(1):100–111, 2006.
14 Joël Ouaknine, Amaury Pouly, João Sousa-Pinto, and James Worrell. On the decidability of membership in matrix-exponential semigroups. *Journal of the ACM (JACM)*, 66(3):1–24, 2019.
15 M.L. Puterman. *Markov Decision Processes*. John Wiley and Sons, 1994.
16 Everett M. Rogers. *Diffusion of innovations*. Free Press of Glencoe, 1962.
17 Thomas Valente. Network models of the diffusion of innovations. *Computational & Mathematical Organization Theory*, 2:163–164, January 1995. doi:10.1007/BF00240425.
18 Duncan Watts and Peter Dodds. Influentials, networks, and public opinion formation. *Journal of Consumer Research*, 34:441–458, February 2007. doi:10.1086/518527.
19 Stephen Wolfram. Statistical mechanics of cellular automata. *Reviews of modern physics*, 55(3):601, 1983.
20 Stephen Wolfram. *Cellular automata and complexity: collected papers*. CRC Press, 2018.
21 Andrew Wuensche. Self-reproduction by glider collisions: the beehive rule. *Alife9 proceedings*, pages 286–291, 2004.