Math 137B | Professor: Padraic Bartlett

Lecture 6: Sequential Dynamical Systems

Weeks 8-9 | *UCSB 2014*

(Relevant source material: the first four chapters of Mortveit and Reidys's "An Introduction to Sequential Dynamical Systems.)

Throughout this course, we have examined the intersections of graph theory with a number of other areas in mathematics and the sciences — namely, the fields of electrical engineering, algebra, and topology. In our last few weeks we will look at another such intersection: the field of **sequential dynamical systems,** in which we will use the language and tools of graph theory to examine discretized versions of dynamical systems and their applications.

We start with a definition:

Definition. A **sequential dynamical system**, abbreviated to SDS throughout this talk, consists of the following collection of objects:

1. A **base graph**, which is a graph G on n vertices $\{v_1, \dots v_n\}$. This graph can either be directed or undirected; if we do not specify which, we will assume that G is undirected.

2. A collection of **vertex states** K. Unless otherwise stated, K will be assumed to be finite. We will frequently work with $K = \mathbb{F}_p$; the field structure of $\mathbb{F}_p$ is useful for proving theorems. More specifically, $K = \mathbb{F}_2$ is an incredibly natural object to use to describe any sort of binary phenomena, and will be our most common choice of K.

3. A collection of **vertex** functions f_v, that take in the state of a vertex and the states of the neighbors of that vertex, and output a vertex state. Formally, if $n(v)$ denotes the number of neighbors of v, we think of these functions as maps $K^{n(v)+1} \to K$. We think of these as "local" update rules: i.e. these functions give us rules for what the vertex v will be at time $t+1$, if at time t we know its state and the states of its neighbors.

4. An **update order** $\pi = (v_{\pi(1)}, \dots v_{\pi(n)})$, that consists of some permutation of the vertices of G. (More generally, we can sometimes work with "word-update orders," where we let the update order merely be a list of vertices in G, instead of a permutation; this lets us consider lists that omit some vertices and repeat others. Unless explicitly stated, however, we will assume all of our update orders are permutations.)

Given such a set of objects, make the following definitions:

1. Call any element $\vec{x} \in K^n$ a **system state**, and associate it to the vertices of G by labeling vertex v_i with the state x_i.

2. Given any vertex state $\vec{x}$ and any vertex v, let $\vec{x}_v$ denote the array of $\vec{x}$'s coordinates that correspond to the vertex v and its neighbors.

3. For each vertex v and associated vertex function f_v, define the **local function** $F_v : K^n \to K^n$ as the following map:

$$F_v(\vec{x}) = (x_1, x_2, \ldots f_v(\vec{x}_v), \ldots x_{n-1}, x_n).$$

In other words, the local function F_v is simply the map that takes in any system state $\vec{x}$, and "updates" the coordinate corresponding to v with $f_v(\vec{x}_v)$.

4. Finally, we define the **SDS-map** $[\mathbf{F}_G, \pi]$ as the composition of these local functions in the order given by the update order: i.e.

$$[\mathbf{F}_G, \pi] = F_{v_{\pi(n)}} \circ F_{v_{\pi(n-1)}} \circ \ldots \circ F_{v_{\pi(2)}} \circ F_{v_{\pi(1)}}.$$

In contrast to how the maps F_v were "local" updates that changed the value at one vertex, we think of $[\mathbf{F}_G, \pi]$ as a "system update map:" given any **initial state** $\vec{x}$, it tells us how $\vec{x}$ changes after we run it through all of our local functions using the update order.

Conceptually, we think of applying $[\mathbf{F}_G, \pi]$ to a given initial state as "advancing" our SDS forward in time one step. We will often be interested in repeatedly applying $[\mathbf{F}_G, \pi]$ to various states and studying the long-term behavior of how these states move around: i.e. if we start from a fixed state and repeatedly apply our system update map, will we return to our original state? How long will it take? Where do we go along the way?

To understand the above definition, it is useful to look at a toy example. Consider the following fairly simple SDS on four vertices:

Example. Consider the following SDS:

1. Base graph: C_4, the unoriented cycle on four vertices, with vertex set v_1, v_2, v_3, v_4 and edges $v_1 \leftrightarrow v_2, v_2 \leftrightarrow v_3, v_3 \leftrightarrow v_4, v_4 \leftrightarrow v_1$.
2. Vertex state set: $\mathbb{F}_2$.
3. Vertex functions: $f_v = \mathbf{nor}_3(\vec{x}_v) : \mathbb{F}_2^3 \to \mathbb{F}_2$, that returns 1 if all of its arguments are 0, and 0 otherwise.
4. The update order (v_1, v_2, v_3, v_4).

Suppose that our SDS starts with the **initial state** $(0, 0, 0, 0)$. What happens when we apply our system update map to this initial state?

Well:

1. First, we apply the function F_{v_1} to our initial state $(0, 0, 0, 0)$. This takes in the vector $(0, 0, 0, 0)$, and replaces it with $(f_{v_1}(\vec{0}_v), 0, 0, 0)$. In particular, note that $\vec{0}_v$, the vector formed by the states of v and its neighbors is $(0, 0, 0)$, and all of our vertex update functions are **nor** functions; therefore, this expression is just $(\mathbf{nor}_3(0, 0, 0), 0, 0, 0) = (1, 0, 0, 0)$.

2. Now, we apply F_{v_2} to this new state $(1,0,0,0)$: because the state of v_2 and its neighbors is currently $(1,0,0)$, this yields $(1, f_{v_2}(1,0,0), 0, 0) = (1,0,0,0)$.

3. Now, we apply F_{v_3} to this state $(1,0,0,0)$: as before, this yields $(1,0, f_{v_3}(0,0,0), 0) = (1,0,1,0)$.

4. Finally, we apply F_{v_4} to this state $(1,0,1,0)$:, this yields $(1,0,1, f_{v_4}(1,0,1)) = (1,0,1,0)$.

This tells us that $[\mathbf{Nor}_{C_4}, (v_1, v_2, v_3, v_4)](0,0,0,0) = (1,0,1,0)$.

The question we studied above — given a SDS, how does it interact with the various possible system states? — is a very commonly studied problem, and as such has a pair of useful visual interpretations.

In the general case where K is just some finite set, we can always form something called the **phase space** diagram for our SDS:

Definition. Given a SDS with system update map $[\mathbf{F}_G, \pi]$, we define the **phase space** diagram for this SDS as the following directed graph:

1. Vertices: the elements of K^n.

2. Edges: draw a directed edge from $\vec{x}$ to $\vec{y}$ whenever $[\mathbf{F}_G, \pi](\vec{x}) = \vec{y}$.

We illustrate this object for the SDS $[\mathbf{Nor}_{C_4}, (v_1, v_2, v_3, v_4)]$ that we studied earlier: this was created by applying the map $[\mathbf{Nor}_{C_4}, (v_1, v_2, v_3, v_4)]$ to each of the sixteen possible initial states in $\mathbb{F}_2^4$. If you are still not confident in your skills at applying update maps, double-check some of the calculations shown below!

Notice that this diagram is particularly sensitive to changes in the update order. For example, this is the phase space diagram for $[\mathbf{Nor}_{C_4}, (******)]$, which is the same SDS as before except with a different update order:

Notice that the structure of this phase diagram is different from our earlier phase diagram; this perhaps helps to illustrate that the update order for a given SDS is very important, and can materially change the behavior of a SDS as much as its update functions!

In the special case where a SDS's collection of vertex states is $\mathbb{F}_2$, there is another convenient way to visualize these state changes: a **space-time diagram**. We make the definitions needed to describe this object here:

Definition. Take any SDS with system update map $[\mathbf{F}_G, \pi]$, along with some initial state $\vec{x}$ for our system. We define the **forward orbit** of $\vec{x}$ under this map as the sequence

$$O^+(\vec{x}) = \big(\vec{x}, [\mathbf{F}_G, \pi](\vec{x}), [\mathbf{F}_G, \pi]^2(\vec{x}), [\mathbf{F}_G, \pi]^3(\vec{x}), \ldots\big)$$

This object is sometimes referred to as a "time series," for fairly natural reasons: if our SDS-map is representing how our system changes over one time step, this sequence tells us how our system with initial state $\vec{x}$ behaves over time!

In the event that our SDS-map $[\mathbf{F}_G, \pi]$ is bijective, then there is a well-defined notion of $[\mathbf{F}_G, \pi]^{-k}$ for any k. This lets us define the notion of the orbit of $\vec{x}$:

$$O(\vec{x}) = \left(\ldots, [\mathbf{F}_G, \pi]^{-2}(\vec{x}), [\mathbf{F}_G, \pi]^{-1}(\vec{x}), \vec{x}, [\mathbf{F}_G, \pi](\vec{x}), [\mathbf{F}_G, \pi]^2(\vec{x}), \ldots\right)$$

This is like the same object as above, except we can "rewind" time and see what states eventually lead to our initial state $\vec{x}$ as well.

Definition. Suppose that we have a SDS-map $[\mathbf{F}_G, \pi]$ where our vertex states come from $\mathbb{F}_2$. For any initial state $\vec{x}$, the **space-time diagram** corresponding to this initial state and SDS map is the following array: index the columns $\{1, \ldots n\}$ of our array by the vertices $\{v_1, \ldots v_n\}$ in G, and color the cells of row k black (1) or white (0) to correspond to the system state $[\mathbf{F}_G, \pi]^k(\vec{x})$.

We draw an example here, for the SDS we have been studying throughout this talk:

We typically draw these space-time diagrams until they begin to loop: i.e. until we have reached a state that we have reached earlier. Note that as illustrated in the example above, this looping does not necessarily need to return to the state $\vec{x}$ that our system started with: it is very possible that the state we start at is one we can never return to. However, because there are finitely many states and finitely many vertices, we are guaranteed that our system must eventually repeat a state that it has previously visited, and thereby enter a loop.

1 Examples of Sequential Dynamical Systems

One of the nice things about sequential dynamical systems is that their definition is remarkably broad; it is easy to describe many commonly-studied systems using the language of SDSs. We list a few examples here:

1. Suppose that you want to study the spread of some illness, say the flu, over a fixed population in a given day. A SDS (or more accurately, a word-SDS, where we let our update order potentially repeat vertices) is a natural way to model this phenomena. Here is one way that this could be set up:

    - Suppose that we have n people that meet and interact throughout the day. Give each of these people their own vertex v_i, and moreover give each meeting that occurs throughout the day its own vertex e_j. Assume that these event vertices e_j are indexed by the times at which they occur: i.e. event e_1 is the first event that happens, followed by event e_2, and so on/so forth. Connect each vertex v_i to each event e_j that person i is a part of throughout the day.
    - Our vertex state set, for its people, consists of two pieces of information. First, each person has some sort of health status (healthy, sick, beginning to become sick, vaccinated, recovering, etc.) Second, each person is currently at some event e_j. Events get some sort of overall score, that corresponds to the number of sick/healthy people at said event.

- Our vertex functions are different for a person or for an event:
  - The update function for an event e_j takes in the health statuses of all of the people at that event, and updates e_j to represent the aggregate "health" of the people at that event.
  - The update function for a person x_j looks up the current event this person is at using e_j along with x_j's current health, and updates their health. This then updates the person's location to the next place they vist during the day. (If you want, you could replace this location update function with a randomized update, in the event that you know people's overall behaviors but not their exact behavior for a given day.)
- Our update order is given by the events in a given day: in order, we process each event e_i followed by its participants.

Iterating this process over a series of days models the overall health of our population!

2. Task scheduling. Suppose that you have some sort of task that you can break down into a number of subtasks τ_i. Some of these tasks τ_i may "depend" on other tasks τ_j: i.e. there may be certain parts of our problem that cannot be started until we have completed other parts of our problem. We can represent this information by using a directed graph defined as follows:

   - Vertices: the tasks τ_j.
   - Edges: draw an edge from τ_i to τ_j if τ_i depends on τ_j.

   Assume that we have the ability to solve each of the individual tasks τ_j. What is the best "order" to try to complete our tasks in? In other words: suppose that we are trying to create a program (for a computer) or workflow (for a company) that orders our tasks in some list. What is the best ordering for us to consider?

   One way to answer this question is via a SDS! Consider the following extension of our graph above to a SDS:

   - Any task τ_i has two possible states: either not completed (0) or completed (1).
   - For any task τ_i, the update function for τ_i works as follows: look at all of the tasks with edges pointing to τ_i. If all of those tasks are completed (1), then all of the prerequisites for starting τ_i have been satisfied. Therefore, we can now complete τ_i: change τ_i to 1.

     Otherwise, τ_i is not yet workable. Do not change τ_i's status.
   - An update order is some ordering π of our tasks.

   Let $[T_\tau, \pi]$ denote the SDS described above. Starting from the initial state where each task is set to 0, we are interested in determining how many times we must apply $[T_\tau, \pi]$ until our SDS's system state is the all-1's vector (i.e. where each task is completed!) In particular, we are interested in determining what update order π requires the smallest number of consecutive applications of $[T_\tau, \pi]$ to get to this all-1's vector: this is in a sense the "most efficient" ordering for us to prioritize our tasks.

3. **Cellular automata**, as mentioned in class, are closely related to sequential dynamical systems in a number of ways. Formally, we define a **cellular automaton** over $\mathbb{Z}^k$ with states in $\mathbb{F}_2$ as the following set of objects:

Graph: The integer lattice $\mathbb{Z}^k$. Sometimes, we will work with $(\mathbb{Z}/n\mathbb{Z})^k$ instead, in cases where we do not want to work with an infinite grid.

States: Each vertex has a state in $\mathbb{F}_2$. Typically, we ask that only finitely many vertices correspond to the state 1, which we think of as denoting "life" or "activity;" conversely, we regard 0 as denoting death or quiescence.

Neighborhood: The **Moore** neighborhood[1] of any vertex $\vec{x}$ consists of all vertices $\vec{x'}$ such that $|x_i - x'_i| \leq 1$, for every coördinate i. So, for a cellular automaton on $\mathbb{Z}^2$, each vertex has eight vertices in its neighborhood.

Local update function: Some function $f : \mathbb{F}_2^{3^k} \to \mathbb{F}_2$, that takes in the state of a vertex and its neighbors and outputs some updated state for that vertex.

Initial state: Some element $\vec{x} \in (\mathbb{F}_2)^{\mathbb{Z}^k}$ that describes the initial state of every vertex in $\mathbb{Z}^k$.

Given a cellular automaton, we can define its **global update function** as the map Φ_f that applies the local update map f to the state of every vertex simultaneously. In this sense, a cellular automaton is not a SDS, as its update functions don't have a set order: they just all occur at once. However, they can be studied using similar techniques!

We think of a given cellular automaton as modeling some abstracted petri dish of bacteria, where a given grid either is alive or dead based on the local behavior of itself and its neighbors; in this sense, the update functions are representing various sorts of rules that determine

One thing that bears noting, given the example of cellular automata above, is the idea of **sequential** versus **simultaneous** in our dynamical systems. Throughout this class, we will typically assume that the update maps we apply are **sequential** in nature: i.e. that there is some set order π in which we apply our maps. This is a fairly natural setting for modeling processes with any sort of sense of causality: i.e. computer programs typically have a sequential ordering of cause and effect. However, there are many natural settings where one would want to study **simultaneous** update orders, where we apply all of our maps simultaneously: for example, our cellular automata above are naturally modeled by a simultaneous system, where we are advancing our entire petri dish one step in time at once, rather than having some cells arbitrarily "go first."

In practice, any phenomena that you capture with a simultaneous dynamical system can be modeled with a sequential dynamical system. Take any dynamical system in which we want to model all of our update maps occurring simultaneously, and consider the following construction:

[1] There are, of course, many notions of neighborhood one could take. For example, you could consider as the neighborhood of any vertex all of the vertices that are distance at most one from this vertex; this is the Von Neumann notion of neighborhood. We will work with Moore neighborhoods unless otherwise specified.

1. Base graph: let G denote the original simultaneous dynamical system's base graph. Create a copy v' of each vertex v in G, and connect that copied vertex to v and all of v's neighbors. Call this collection H, and think of it as the "temporary" update graph.

2. States: same stats as for the simultaneous system.

3. Update functions: Take any copied vertex v' in H, that corresponds to some vertex $v \in G$. The vertex $v \in G$ had some original update function f_v, that took in the states of v and its neighbors and output some state. Give v' that exact same update function: i.e. to update v', ignore whatever state v' has, and instead look at the states of v and its neighbors. Update v' to that state.

   As well, assign to each vertex $v \in G$ the update function that looks up the state of v', the cloned copy of v, and assigns that state to v itself.

4. Finally, our update order is any ordering of the vertices in H, followed by any ordering of the vertices in G.

If you run this process, the following things happen:

- First, we will assign each vertex v' in H the state that the corresponding vertex in v would get if we were running a simultaneous update system. Notice that because we only change values in H, this is indeed preserving the simultaneous nature that we are trying to simulate: i.e. after we update all of H, we have not yet changed any values in G.

- Now, we copy all of those H-states back over onto G. After doing this, we have effectively updated each vertex in G "simultaneously," in the sense that each vertex was updated using only the initial state that we had at the start of our update process.

You can think of the graph H above as the "memory" for our update process on G: i.e. we first calculate what all of G's states will be when we simultaneously update, and then we use this precalculated information to actually update G. We illustrate an example below:

2 Properties of Sequential Dynamical Systems: Fixed Points

One of the main virtues of sequential dynamical systems is that we can describe so very many things as a SDS of some sort. An unfortunate consequence is that it is correspondingly hard to deduce properties that hold for all sequential dynamical systems; if a system can describe lots of different kinds of phenomena, then we would expect it to be harder to deduce any sorts of "universal" properties for such systems!

However, if we limit the scope of our systems somewhat — i.e. to sequential dynamical systems with certain nice update function sets, or to systems with certain desirable properties — it turns out that we can actually make some mathematically useful observations! In this section, we make the first few such observations for this class. When reading these

results, don't worry so much about the overall utility of the specific claim being made — rather, think of these results as illustrating the kinds of statements and claims that one can study when working with a SDS!

The first concept we will work with is a fairly natural one from the dynamical-systems standpoint: the concept of **fixed points**.

Definition. Consider a SDS with system update map $[\mathbf{F}_G, \pi]$. A **fixed point** of this SDS is some initial state $\vec{x}$ such that $[\mathbf{F}_G, \pi](\vec{x}) = \vec{x}$; i.e. it is an initial state that is unchanged when we apply $[\mathbf{F}_G, \pi]$.

One quick and fairly useful observation we can make about fixed points is that they do not depend on the update order of our SDS:

Observation. Suppose that we have a SDS with system update map $[\mathbf{F}_G, \pi]$ that has a fixed point $\vec{x}$. Suppose we replace π with any other update order π, to get a SDS on the same base graph/vertex state/local function set, with system update map $[\mathbf{F}_G, \pi']$. Then $\vec{x}$ is also a fixed point of this new system update map.

Proof. This is actually remarkably easy to see. Notice that by definition, we have

$$[\mathbf{F}_G, \pi] = F_{v_{\pi(1)}} \circ \ldots \circ F_{v_{\pi(n)}},$$

where each F_{v_i} is the local function defined by

$$F_{v_i}(\vec{x}) = (x_1, \ldots x_{i-1}, f_{v_i}(\vec{x}_{v_i}), x_{i+1}, \ldots x_n)\,.$$

In particular, notice the following two things:

- Each F_{v_i} only changes the i-th coördinate of any input they receive, and leaves all of the other coordinates untouched.
- In particular, suppose for a moment that $f_{v_i}(\vec{x}_{v_i}) \neq x_i$. Then the i-th component of $F_{v_i}(\vec{x})$ will not be equal to x_i. Then, because no other F_{v_j} can change this i-th coördinate, the i-th coördinate of $[\mathbf{F}_G, \pi](\vec{x})$ cannot be equal to x_i, and therefore $\vec{x}$ cannot be a fixed point.
- The contrapositive of the above point then tells us that if $\vec{x}$ **is** a fixed point, then $f_{v_i}(\vec{x}_{v_i}) = x_i$, for all i. In particular, this tells us that $F_{v_i}(\vec{x}) = \vec{x}$ for every i, as these functions do not change the one coördinate that they can change!
- In other words, we have shown that each F_{v_i} sends $\vec{x}$ to $\vec{x}$. Therefore, it does not matter in what order we compose them; any string of F_{v_i}'s applied to $\vec{x}$ in any order will yield a function that sends $\vec{x}$ to $\vec{x}$.

This proves our claim: we have shown that if $\vec{x}$ is a fixed point of a SDS under one update order, then it is a fixed point of our SDS under **any** update order! □

This stands in sharp contrast to the existence of states of period k (i.e. a state $\vec{x}$ such that $[F_G, \pi]^k(\vec{x}) = \vec{x}$, and $[F_g, \pi]^l(\vec{x}) \neq \vec{x}$, for any positive natural number $l < k$.) As we saw in our earlier examples where we looked at $[\mathbf{Nor}_{C_4}, \pi]$ under different permutations π, the underlying phase state can change dramatically when we change π: i.e. in one case we got cycles of length ****, while in the other we had ****.

In many situations, a fixed point is a kind of phenomena that we may want to either avoid or insure exists. For example, suppose that we are modeling some sort of mixing process. Our graph could be a grid representing some sort of fluid, and the states of our vertices could correspond to the total flow of fluid through each cell. A **fixed point** here corresponds to some initial state that when we update our flow returns to itself: this could be either undesirable (if we want our process to "mix" our fluid, we may want to avoid fixed points) or desirable (we could want a way to sometimes stir our fluid predictably, i.e. in a way that comes back to itself when we finish.)

It turns out that if we restrict ourselves to sufficiently "nice" families of vertex update functions: i.e. vertex functions over the state set $\mathbb{F}_2$, that are **symmetric**[2], we can actually make some fairly strong claims:

Theorem. Suppose that $\{f_i\}_{i \in I}$ is some collection of symmetric functions from $\mathbb{F}_2^{k_i} \to \mathbb{F}_2$, with the following property:

- For any k, there is a f_i that takes in inputs from $\mathbb{F}_2^k$.
- Take any graph G and update order π. Build a SDS with G as our base graph, π as our update order, and $\mathbb{F}_2$ as our vertex state set, where our vertex functions all come from our collection $\{f_i\}_{i \in I}$. Then this SDS has no fixed point.

In a sense, you can think of $\{f_i\}_{i \in I}$ as some collection of functions that are "guaranteed" to generate fixed-point-free SDSs, no matter what graph or update order you pick.

There are two very surprising facts about these collections:

1. Such collections exist. In particular, take the collections $\{\mathbf{nor}_i\}_{i=1}^{\infty}$ and $\{\mathbf{nand}_i\}_{i=1}^{\infty}$: these both[3] satisfy the above property.
2. No other collections exist. In other words, any collection that satisfies the above properties is either the all-**nor**-set or the all-**nand** set.

The first part of this claim — that the all-**nor**-set or the all-**nand** set have the property claimed — we leave for the reader to verify on their own! It is not a hard thing to check, and is good practice to do.

The second part of this claim — that these are the **only** possible such collections — we prove here.

[2] We say that a function $f(x_1, \ldots x_n)$ is symmetric if the order in which we describe its inputs does not change the output: i.e. if $f(x_1, \ldots x_n) = f(x_{\pi(1)}, \ldots x_{\pi(n)})$, for any permutation π.

[3] In case you forgot: the functions $\mathbf{nor}_i$ take in i values from $\mathbb{F}_2$ and output 1 if they are all 0, and 0 otherwise. The functions $\mathbf{nand}_i$ are similar; they take in i inputs from $\mathbb{F}_2$ and output 0 if they are all 1, and 0 otherwise.

Proof. We proceed in two stages: we first prove that each individual function f_i is either a nor or a nand, and then proceed to show that our collection cannot contain both a nor *and* a nand.

To see the first claim: take any f_i from our collection, and assume without loss of generality that $f_i : \mathbb{F}_2^i \to \mathbb{F}$. If $i = 1$, then we can verify our claim by checking cases:

- First, notice that $f_1(0)$ must be equal to 1, as otherwise the graph K_1 has 0 as a fixed point.
- Similarly, we must have $f_1(1) = 0$, to avoid having 1 as a fixed point.
- This completely determines our function! In particular, we have shown that $f_1 = \mathbf{nor}_1 = \mathbf{nand}_1$. (Conveniently, we don't even have to worry about which of nor or nand this function is: in the one-input case, they are the same.)

For $i = 2$ we can proceed by cases in a similar fashion:

- Again, notice that $f_2(0, 0)$ must be equal to 1, as otherwise the graph K_2 has $(0, 0)$ as a fixed point.
- Similarly, we must have $f_2(1, 1) = 0$, to avoid having $(1, 1)$ as a fixed point.
- Because our function is symmetric, we know that $f_2(0, 1) = f_2(1, 0)$. So there are two cases:
  1. $f_2(0, 1) = f_2(1, 0) = 0$. In this case, we are $\mathbf{nor}_2$.
  2. $f_2(0, 1) = f_2(1, 0) = 1$. In this case, we are $\mathbf{nand}_2$.

Again, we have proven our claim.

We now proceed to prove our claim for any $i > 2$. Again, notice that as before our function must send the all-0 state to 1, and the all-1 state to 0.

We proceed by contradiction, and assume that f_i is neither $\mathbf{nor}_i$ nor $\mathbf{nand}_i$; we will seek to create a SDS using these functions that has a fixed point, which will contradict our initial assumptions that any SDS we can create must be fixed-point free. In this case, we must have a pair of system states $\vec{x}, \vec{y}$ such that

- $f_i(\vec{x}) = 1$, $f_i(\vec{y} = 0$, and
- neither $\vec{x}$ nor $\vec{y}$ are identically 0 or identically 1.

Such a pair of system states must exist if our function is neither $\mathbf{nor}_i$ nor $\mathbf{nand}_i$ (if this is not clear, check this for yourself!) Let l_x denote the number of 1's in the vector $\vec{x}$, and l_y denote the number of 1's in the vector $\vec{y}$.

We now proceed by cases, depending on whether $f_2(0, 1) = f_2(1, 0) = 0$ or $f_2(0, 1) = f_2(1, 0) = 1$:

1. Suppose that $f_2(0, 1) = f_2(1, 0) = 0$. In this case, create the following graph:
   - Take a copy of K_{l_x}.

- For each vertex $v \in K_{l_x}$, create $i - l_x$ new vertices, and connect all of those vertices to v.

Now, consider the SDS on this graph where we assign f_i to all of the vertices in the K_{l_x} part, and f_2 to all of the degree-1 vertices we attached to this complete graph. If we look at the system state where we initialize each vertex in K_{l_x} to 1 and the rest to 0, we can see that

- Each f_i will be applied to a vector with l_x-many 1's, and therefore yield 1, because $f_i(\vec{x}) = 1$ and our functions are symmetric.
- Each f_2 will be applied to a vector with one 1 and one 0, and therefore yield 0.

In other words: this state is a fixed point! This contradicts our assumption that any SDS we could build would be fixed-point free, as desired.

2. Now, suppose instead that $f_2(0,1) = f_2(1,0) = 1$. In this case, create the following graph:

   - Take a copy of K_{i-l_y}.
   - For each vertex $v \in K_{i-l_y}$, create l_y new vertices, and connect all of those vertices to v.

   Now, consider the SDS on this graph where we assign f_i to all of the vertices in the K_{i-l_y} part, and f_2 to all of the degree-1 vertices we attached to this complete graph. If we look at the system state where we initialize each vertex in K_{i-l_y} to 0 and the rest to 1, we can see that

   - Each f_i will be applied to a vector with l_y-many 1's, and therefore yield 0, because $f_i(\vec{y}) = 0$ and our functions are symmetric.
   - Each f_2 will be applied to a vector with one 1 and one 0, and therefore yield 1.

   In other words: this state is also a fixed point! This again contradicts our assumption that any SDS we could build would be fixed-point free.

This proves our first claim: that each individual function f_i is either a nor or a nand. To see that our collection cannot contain both a nor *and* a nand, we again proceed by contradiction: assume that we have both a $\mathbf{nor}_i$ and $\mathbf{nand}_j$ in our collection. Consider the complete bipartite graph $K_{i-1,j-1}$, where we associate $\mathbf{nor}_i$ to all of the vertices in one part and $\mathbf{nand}_j$ to all of the vertices in the other part of our bipartite graph. Initialize every vertex in the degree-$i-1$/**nor** part to 0, and initialize and every vertex in the degree-$j-1$/**nand** part to 1. Then, notice that

- each $\mathbf{nor}_i$ function will see 1's in its neighbors and therefore remain in its current state 0, while
- each $\mathbf{nand}_i$ function will see 0's in its neighbors and therefore remain in its current state 1.

In other words: we have created a SDS with a fixed point! This gives us a contradiction, and completes the last part of our proof, as desired. □

3 Properties of Sequential Dynamical Systems: Invertibility

A second property that is often desirable in a SDS is the concept of **invertibility**. We indirectly mentioned this earlier when defining the **time series** for a given initial state, but formally define this concept here:

Definition. Consider a SDS on a graph G with n vertices, vertex states from some set K, and system update map $[\mathbf{F}_G, \pi]$. We say that this SDS is **invertible** if its corresponding system update map, thought of as a function $K^n \to K^n$, is invertible; i.e. a bijection. (Notice that because K^n is a finite set, it suffices to show that $[\mathbf{F}_G, \pi]$ is either injective or surjective to get that it is a bijection; this is a fairly elementary consequence of basic set theory, and is one that we will use without further justification in our proofs.)

Given a specific SDS, how can we tell if it is invertible? We answer this in the following observation:

Observation. Take any SDS on a graph G with n vertices, vertex states from some set K, local functions $\{f_v\}_{v \in V(G)}$, and system update map $[\mathbf{F}_G, \pi] = F_{v_{\pi(1)}} \circ \ldots \circ F_{v_{\pi(n)}}$. This system update map is invertible if and only if the following holds: for any vertex v and set of states for the neighbors of v $\vec{x}_{n(v)} \in K^{n(v)}$, the map

$$g_{v,\vec{x}_{n(v)}} : K \to K, g_{v,\vec{x}_{n(v)}}(k) = f_v(\overbrace{x_1, \ldots k, \ldots x_{n(v)}}^{\substack{\text{the vector } \vec{x}_{n(v)}, \\ \text{with } k \text{ inserted in} \\ v\text{-th position}}})$$

is a bijection.

Proof. We first briefly explain what these maps $g_{v,\vec{x}_{n(v)}}$ are: they are simply the functions that we get by taking each f_v and fixing all of their inputs except the one corresponding to v itself. For example, suppose that we were working with the SDS on C_3 where each vertex is assigned the function $\mathbf{nor}_3$. Then, for example, we would have

$$f_{v_1} = \mathbf{nor}_3(x_3, x_1, x_2).$$

The map $g_{v_1,\vec{x}_{n(v)}}$ would then correspond to any way of fixing the values of v_1's neighbors ahead of time, to make this just a function of vertex v_1's state: i.e.

$$g_{v_1,(0,0)}(x_1) = \mathbf{nor}_3(0, x_1, 0), \qquad g_{v_1,(0,1)}(x_1) = \mathbf{nor}_3(0, x_1, 1),$$
$$g_{v_1,(1,0)}(x_1) = \mathbf{nor}_3(1, x_1, 0), \qquad g_{v_1,(1,1)}(x_1) = \mathbf{nor}_3(1, x_1, 1).$$

With this stated, the proof is actually really easy: it's just a matter of wrapping your head around the notation. First, notice that $[\mathbf{F}_G, \pi]$ is invertible if and only if each of the functions $F_{v_{\pi(1)}}, \ldots \circ F_{v_{\pi(n)}}$ are invertible. (Again, this is a consequence of some elementary set theory; prove this if you do not see why it is true!)

Now, notice that each F_{v_i} is simply the local function defined by

$$F_{v_i}(\vec{x}) = (x_1, \ldots x_{i-1}, f_{v_i}(\vec{x}_{v_i}), x_{i+1}, \ldots x_n)\,.$$

What does it mean for this function to be a bijection? Well: consider the functions $g_{v_i,\vec{x}_{n(v_i)}}$. If there is some set of values $\vec{x}_{n(v_i)}$ such that one of these maps fails to be injective, then by definition there are two values k, k' such that $g_{v_i,\vec{x}_{n(v_i)}}(k) = g_{v,\vec{x}_{n(v_i)}}(k')$. Take $\vec{x}_{n(v_i)}$ and extend it to a vector in $K^{n(v_i)+1}$ by putting k or k' in the v_i-coordinate; call these two vectors $\vec{x}_{v_i}, \vec{x}'_{v_i}$ respectively.

By construction, we know that $f_v(\vec{x}_{v_i}) = f_v(\vec{x}'_{v_i})$. Therefore, if we extend these vectors to a pair of system states $\vec{x}, \vec{x}' \in K^n$, we actually have

$$F_{v_i}(\vec{x}) = (x_1, \ldots x_{i-1}, f_{v_i}(\vec{x}_{v_i}), x_{i+1}, \ldots x_n) = \big(x_1, \ldots x_{i-1}, f_{v_i}(\vec{x}'_{v_i}), x_{i+1}, \ldots x_n\big) = F_{v_i}(\vec{x}').$$

In other words, F_{v_i} is not injective!

Therefore, we have shown that it is clearly necessary for the functions $g_{v_i,\vec{x}_{n(v_i)}}$ to be bijections if the F_{v_i} functions are bijections. The same logic shows that it is **sufficient** for the $g_{v_i,\vec{x}_{n(v_i)}}$-functions to be bijections; if we have any two vectors $\vec{x}, \vec{x}'$ such that

$$F_{v_i}(\vec{x}) = (x_1, \ldots x_{i-1}, f_{v_i}(\vec{x}_{v_i}), x_{i+1}, \ldots x_n) = \big(x_1, \ldots x_{i-1}, f_{v_i}(\vec{x}'_{v_i}), x_{i+1}, \ldots x_n\big) = F_{v_i}(\vec{x}'),$$

then we know in particular that $f_{v_i}(\vec{x}_{v_i}) = f_{v_i}(\vec{x}'_{v_i})$, and moreover that all of the coordinates $x_1, \ldots x_n \neq x_{v_i}$ are identical. Therefore, these two values are equal if and only if $g_{v_i,\vec{x}_{n(v_i)}}(x_{v_i}) = g_{v_i,\vec{x}_{n(v_i)}}(x'_{v_i})$. If the functions $g_{v_i,\vec{x}_{n(v_i)}}$ are all bijections, this forces $x_{v_i} = x'_{v_i}$, and therefore tells us that our F_{v_i} functions are bijections! □

This might initially look like a fairly weak observation: we've effectively just said that a function is invertible if and only if it is made out of invertible things. Yet, this actually is a fairly useful observation to have made: it lets us evaluate whether a large-scale phenomena like $[\mathbf{F}_G, \pi]$ is invertible based on only local information like how these $g_{v_1,\vec{x}_{n(v)}}$ maps work!

To illustrate the power of this observation, consider the following two examples:

Example. Take any graph G. Turn this into a SDS with vertex states from $\mathbb{Z}/k\mathbb{Z}$, where all of the vertex functions are of the form

$$f_v(x_1, \ldots x_n) = (\mathbf{mod\ k})_n(x_1, \ldots x_n) = x_1 + \ldots + x_n \mod k.$$

This SDS is invertible; to see why, simply apply the observation from above, and note that each $g_{v,\vec{x}_{n(v)}}$ has the form

$$g_{v,\vec{x}_{n(v)}}(x_v) = x_v + (x_1 + \ldots x_{n(v)}) \mod k = x_v + c \mod k,$$

for some constant $c \in \mathbb{Z}/k\mathbb{Z}$. In particular, this means that the maps $g_{v,\vec{x}_{n(v)}}$ are all of the form $x \mapsto x + c \mod k$ for various constants c, which are all bijections on $\mathbb{Z}/k\mathbb{Z}$.

Example. Consider the cellular automata rule $f_{73} : \mathbb{F}_2^3 \to \mathbb{F}_2$, that takes in a triple (x_{i-1}, x_i, x_{i+1}) and outputs a new value for x_i as defined by the table below:

(x_{i-1}, x_i, x_{i+1})	111	110	101	100	011	010	001	000
f_{73}	0	1	0	0	1	0	0	1

No SDS that uses this vertex update function — say, for example, the SDS on C_n where we use this function on all of the vertices of our cycle — can be invertible. To see why, simply notice that $f_{73}(101) = f_{73}(111) = 0$, and therefore that the corresponding g-function $g_{v,(11)} : \mathbb{F}_2 \to \mathbb{F}_2, g_{v,(11)} = f_{73}(1, x_v, 1)$ is not a bijection. Therefore, by our observation above, the SDS that is built by using this function must also fail to be a bijection.

For a SDS with vertex states taken from $\mathbb{F}_2$, this gives us a fairly nice corollary:

Corollary. Take any SDS with vertex states from $\mathbb{F}_2$ and system update map $[\mathbf{F}_G, \pi] = F_{v_{\pi(1)}} \circ \ldots \circ F_{v_{\pi(n)}}$. If this system update map is invertible, then its inverse is just $[\mathbf{F}_G, \pi^*]$, where π^* is π in reverse order: i.e. if $\pi = (v_1, \ldots v_n)$, then $\pi^* = (v_n, v_{n-1} \ldots v_1)$

Proof. First, notice that the only two invertible functions from $\mathbb{F}_2$ to $\mathbb{F}_2$ are the identity map and the map that switches 0 and 1; in either case, repeating either map twice yields the identity.

Therefore, when we look at the composition

$$\begin{aligned}[\mathbf{F}_G, \pi] \circ [\mathbf{F}_G, \pi*] &= (F_{v_{\pi(1)}} \circ \ldots \circ F_{v_{\pi(n-1)}} \circ F_{v_{\pi(n)}}) \circ (F_{v_{\pi(n)}} \circ F_{v_{\pi(n-1)}} \circ \ldots \circ F_{v_{\pi(1)}}) \\ &= (F_{v_{\pi(1)}} \circ \ldots \circ (F_{v_{\pi(n-1)}} \circ (F_{v_{\pi(n)}} \circ F_{v_{\pi(n)}}) \circ F_{v_{\pi(n-1)}}) \circ \ldots \circ F_{v_{\pi(1)}})\end{aligned}$$

we can see that each pair $F_{v_{\pi(k)}} \circ F_{v_{\pi(k)}} = F^2_{v_{\pi(k)}}$ is just the map that sends $(x_1, \ldots x_n)$ to $(x_1, \ldots g^2_{v_{\pi(k)}, \vec{x}_{n(v_{\pi(k)})}}(x_{v_{\pi(k)}}), \ldots x_n)$. In particular, because g is an invertible map from $\mathbb{F}_2$ to $\mathbb{F}_2$, we know that g^2 is the identity, and therefore that this composition is the identity as well!

Repeatedly applying this observation above gives us that the entire composition $[\mathbf{F}_G, \pi] \circ [\mathbf{F}_G, \pi*]$ is the identity, as claimed. □

Just like with fixed points, we can actually make some very strong claims about any SDS whose vertex functions are symmetric and take values in $\mathbb{F}_2$:

Theorem. Suppose that we have an invertible SDS with vertex values from the set $\mathbb{F}_2$, in which all of our local vertex functions f_v are symmetric. Then each of our vertex functions are either $\mathbf{parity}_n$ or $\mathbf{1} + \mathbf{parity}_n$, where these functions are defined as follows:

$$\begin{aligned}\mathbf{parity}_n(x_1, \ldots x_n) &= x_1 + \ldots + x_n \mod 2, \\ \mathbf{1} + \mathbf{parity}_n(x_1, \ldots x_n) &= x_1 + \ldots + x_n + 1 \mod 2.\end{aligned}$$

Proof. Take any vertex function $f_v : \mathbb{F}_2^i \to \mathbb{F}_2$, and look at $f_v(0, \ldots 0)$. I claim that if this function outputs 0 on this input, then it is $\mathbf{parity}_i$, and otherwise it is $\mathbf{1} + \mathbf{parity}_i$.

To see why, simply induct on the number of 1's in any input to f_v. It is clear that for all inputs with no 1's, our function f_v agrees with our claim that it is one of $\mathbf{parity}_i$ or $\mathbf{1} + \mathbf{parity}_i$.

Now, assume inductively that for all inputs with k or fewer 1's, our function f_v agrees with one of $\mathbf{parity}_i$ or $\mathbf{1} + \mathbf{parity}_i$. Take any input $\vec{x}$ with $k+1$ 1's in it. By rearranging, insure that this input has a 1 in the v-th position; because our function is symmetric, we

do not lose any generality by assuming that our vector has this form. Consider the vector $\vec{x}'$ formed by taking $\vec{x}$ and replacing its v-th coördinate with a 0; this is now a vector with k 1's in it.

By our earlier observation, we know that $f_v(\vec{x}) \neq f_v(\vec{x}')$, as they agree at everywhere but their v-th coördinate, and we switched the value in our v-th coördinate! Therefore, when we increased the number of 1's in an input by one, the output of our function flipped. In particular, if our function gave us the parity of k on an input of k 1's, it now gives us the parity of $k+1$ on an input of $k+1$ 1's — in other words, it is still the parity function! Similarly, if it used to be 1+ the parity function on inputs of size k, it is still that function on inputs of size $k+1$.

By induction and the observation that our function is symmetric, we have established our claim for all possible inputs; i.e. each of our vertex functions are either $\mathbf{parity}_i$ or $\mathbf{1} + \mathbf{parity}_i$, as claimed. □