The Dynamics of Turing Machines

1 Abstract

We propose to study the behavior of Turing Machines in the context of dynamical systems. A *Turing Machine* (abbreviated "TM") is a model of computation involving a *transition function* on a space of *configurations*. We think of TMs as beginning with a finite *input string* and evolving according to the rules of the transition function; TMs can either run forever or halt in one of the *accepting* or *rejecting* state. Hence, a TM can be naturally understood as a discrete dynamical system with two fixed points. In this context the asymptotic behavior of TMs is of great interest in the field of Computability Theory; however, the problem has only recently begun being examined from the dynamical perspective. For our project we plan to examine the connection between these two disciplines, both in terms of how tools from abstract dynamical systems can help us interpret the behavior of TMs as well as how continuous dynamical systems can be approximated discretely to create analog models of computation. Particular topics of focus might include studying randomized approximation algorithms using the techniques of stability theory or investigating connections between the halting problem and limit cycles.

2 Introduction

We'll begin by describing some of common pieces of vocabulary used in computability theory.

2.1 Strings and Alphabets

First, we give the definitions for *alphabets* and *strings*. These will not be terribly important later on, it's just helpful to have the vocabulary.

Definition 1 (Alphabet)**.** An *alphabet* is a finite set, often denoted by Σ. The elements of Σ are called *characters*.

We make no assumptions about the characters in Σ; they will *exclusively* serve as formal symbols for use in strings.

Definition 2 (Strings)**.** Let Σ be an alphabet. We define an operation $\cdot$ on Σ as follows: Let $\sigma_1, \sigma_2 \in \Sigma$ be arbitrary (not necessarily distinct). Then define a *new* symbol $\sigma_1\sigma_2$, and let

$$\sigma_1 \cdot \sigma_2 = \sigma_1\sigma_2.$$

Also define a special symbol $\epsilon \notin \Sigma$ with the property that for all $\sigma \in \Sigma$,

$$\epsilon\sigma = \sigma = \sigma\epsilon.$$

Define $\Sigma^\star$ to be the collection of all formal symbols generated by $\Sigma \cup \{\epsilon\}$ under the operation $\cdot$ described above. Explicitly, if $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$, then we have

$$\Sigma^\star = \big\{\epsilon,\ \lceil\sigma_1\rfloor,\ \ldots,\ \lceil\sigma_n\rfloor,\ \lceil\sigma_1\sigma_1\rfloor,\ \ldots,\ \lceil\sigma_i\sigma_j\rfloor,\ \ldots,\ \lceil\sigma_n\sigma_n\rfloor,\ \ldots, \lceil\sigma_i\sigma_j\sigma_k\rfloor, \ldots\big\}$$

$$= \bigcup_{k=0}^{\infty} \left\{ \lceil \sigma_{i_1} \sigma_{i_2} \cdots \sigma_{i_k} \rfloor \;\middle|\; \sigma_{i_j} \in \Sigma \right\}$$

Note: we are using the tortoise shell brackets $\lceil \rfloor$ just to help visually separate the elements. Also, when $k = 0$, we take the convention that $\lceil \rfloor = \epsilon$. In any case, we call

(a) $\Sigma^\star$ the *Kleene star* of Σ,

(b) The $\cdot$ operation the *concatenation operation*, and

(c) The elements of $(\Sigma^\star, \cdot)$ *strings*.

Remark. This is the same as defining $(\Sigma^\star, \cdot)$ to be the *free monoid* over Σ.

Definition 3 (Length of a string)**.** Let Σ be an alphabet, and let $\sigma \in \Sigma^\star$. Suppose σ is of the form

$$\sigma = \sigma_{i_1} \sigma_{i_2} \cdots \sigma_{i_k}$$

where $\sigma_{i_j} \in \Sigma$ for each $j = 1, \ldots, k$. Then we define the *length* of σ to be

$$|\sigma| = k.$$

Remark. Note, since $\epsilon \notin \Sigma$, our assumption that each of the $\sigma_{i_j} \in \Sigma$ makes the length function well-defined.

Definition 4 (Language)**.** Let Σ be an alphabet, and let $L \subseteq \Sigma^\star$. Then we call L a *language* over Σ.

On their own, alphabets and strings are not very interesting. Hence most of our questions center on *languages*. One might note that in our definition for a language, we made no requirements on L other than $L \subseteq \Sigma^\star$. Hence, L could be something extremely simple, such as

$$L_{\text{easy}} = \left\{ \sigma \in \Sigma \;\middle|\; |\sigma| \equiv_2 0 \right\},$$

or something fiendishly complex, like

$$L_{\text{hard}} = \left\{ \sigma \;\middle|\; \sigma \text{ is a base-2 encoding of a valid proof of the Hartman–Grobman theorem.}^{[1]} \right\}$$

We think of L_{easy} as having a much simpler "structure" when compared to L_{hard}. This is a byproduct of the differences in the predicates (the parts after the |'s) we've use to define our sets. In the first, the rule is very simple to state, and very simple to verify. We could imagine a program that determines membership in L_{easy} simply by scanning left-to-right and flipping a parity bit until reaching the end of the string. Then, it simply checks the final value of the bit to determine whether the length of the string is even or odd. See Fig. 1 for an illustration.

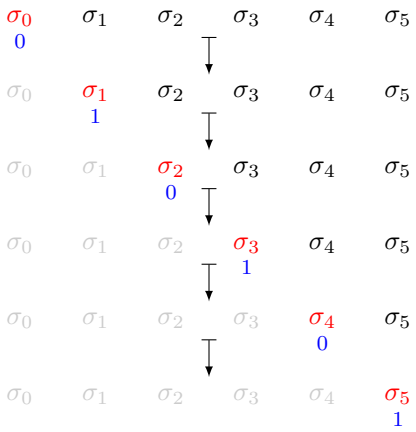

Figure 1: An illustration of the process. In each step, gray represents characters that've already been read, red symbols represent the current character, and blue symbol indicate the value of the parity bit.

We note that a computer built to execute this algorithm could be very simple: it would only need enough memory to store the program and keep track of the parity bit, and this would be sufficient to determine membership in L_{easy} for arbitrary input.

By contrast, L_{hard} seems a lot harder. Since proofs can be made arbitrarily long[2] while still remaining valid,[3] we'd need to keep track of claims that were made arbitrarily far back in the input. E.g., our string might encode a proof that's structured like the following:

- Axioms: A_0, A_1, …, A_n.
- Desired result: Z.
- Claim: B. Proof of claim: $A_0 \wedge A_1 \implies B_0$. Now, $B_0 \implies B_1$. Also $B_1 \wedge A_2 \implies \cdots \implies B$.
- Claim: C. Proof of claim:
  - $1 + 1 = 2$
  - $2 = 2$
  - $1 + 1 = 2$
  - …More vacuous filler claims, until finally
  - $A_0 \wedge B \implies C$.
- Similarly to the above for D through Y, until
- $\ldots \implies Z$.

So it's essentially impossible to create an algorithm that looks exclusively at local data to verify whether a given string σ is an element of L_{hard}. This difficulty is made formal by the *Chomsky Hierarchy*, which essentially gives an ordering to languages based on the

[2] See: this document
[3] See: maybe a different document

complexity of the "computers" needed to identify their members.[4] L_{easy} corresponds to one of the simplest possible models, which is known as a *Deterministic Finite Automaton*. By contrast, one of the most powerful models is known as a *Turing machine*, which we describe below. All of these can be unified into a single conceptual framework by viewing them as *dynamical systems*. The reader might consider how as we begin with our formal definitions.

2.2 Turing Machines

A *Turing machine* (often TM for short) is an abstract model of computation that allows us to encode computer programs as collections of sets with well-defined maps. There are many "equivalent" definitions for Turing machines;[5] we'll give the canonical one first and save the more dynamical-systems-flavored version for later.

Definition 5 (Turing Machine)**.** A *Turing machine* is a 5-tuple

$$M = (Q, F, \Gamma, \Sigma, \delta)$$

such that the following conditions hold:

1) Q, F, Γ, Σ and F are finite sets. Note, we choose the following naming conventions:
   i) Q is called the set of *states* for M,
   ii) $F = \{\checkmark, \text{✗}\}$ is called the set of *final states* (read "accept" and "reject" respectively),
   iii) Γ is called the *work alphabet*, and
   iv) Σ is called the *input alphabet*;
2) $F \subseteq Q$ and $\Sigma \subseteq \Gamma$;
3) Q contains a distinguished element q_0, called the *start state* or *initial state*;
4) Γ contains a distinguished element $\textvisiblespace$ called the *blank character* (we require $\textvisiblespace \notin \Sigma$);
5) δ (called the *transition function*) has the following properties:
   i) δ is a partial function $\delta : (Q \setminus F) \times \Gamma \nrightarrow Q \times \Gamma \times \{L, R\}$. Note the inclusion of the word *partial* — in general we do not require δ to be defined on all of $(Q \setminus F) \times \Gamma$.
   ii) L, R are understood as "shift" directions, as explained below.

We think of Turing machines as operating on an infinite *work tape*, where the cells of the tape each represent a symbol from the alphabet Γ.[6] One can think of this tape as representing the "memory" of the TM.

The tape is initialized with a finite word $\sigma = \sigma_1\sigma_2\cdots\sigma_n$ representing the input to our program, while the rest of the tape is filled with $\textvisiblespace$ characters. The TM initializes in the *start state*, with the *tape head* (drawn with an $\uparrow$ in the below) on σ_1. We think of the tape head as "pointing" to whatever character the TM is reading currently.

[4] Technically "computer" should be replaced with "computational model," but the distinction isn't important for us today

[5] Here, "equivalent" means that the models of computation they define are equally powerful. That is, given two distinct definitions D_0, D_1 for Turing machines, the behavior of any machine defined with D_0 can be simulated using a machine defined by D_1 and vice versa.

[6] Formally, the addition of the tape requires using an infinite tuple of elements from $\prod_{i=-\infty}^{\infty} \Gamma$, with all but finitely-many entries non-blank.

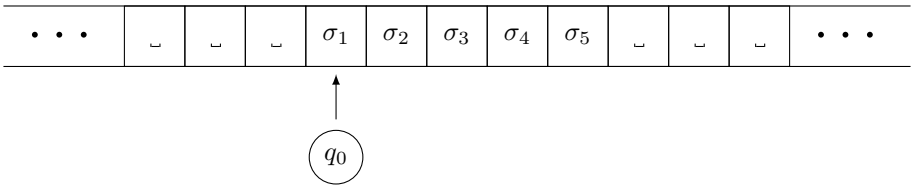

Figure 2: Turing machine initialized with the input string $\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$.

The "program" is then executed by the δ function. For instance, suppose $\delta(q_0, \sigma_1) = (q_i, \gamma_1, R)$. Then we interpret this as the Turing machine writing a γ_1 on the current tape, updating its state to q_1, and moving one space to the right on the tape.

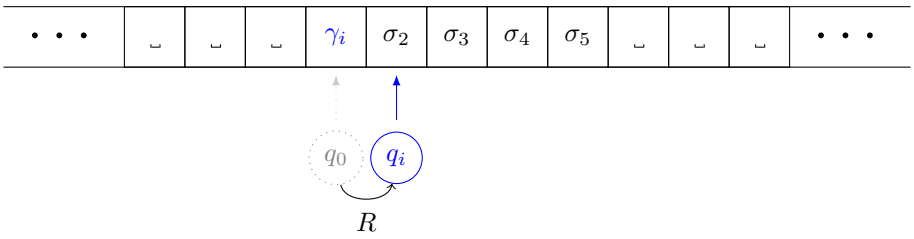

Figure 3: Turing machine after performing a single step. Differences shown in blue.

At each step of the computation, the Turing machine performs a similar process: Read the tape, rewrite and move as dictated by δ, and then repeat. If the Turing machine reaches the state ✓, then it halts and the input is said to be "accepted." If it reaches ✗, the state is "rejected." In the case that the machine never halts, we treat this as a form of rejection.

3 Computational Universality

A Universal Turing Machine is a Turing machine which can simulate any arbitrary Turing Machine. It takes as its Turing Machine code, and an input word, and then simulates the Turing Machine on that input. Intuitively, any modern computer is a (finite memory) Universal Turing Machine: it can store arbitrary computer programs as data, and then run them on other data it stores or with user input.

In a similar vein, a system is *computationally universal* (or just *universal*) if it has the ability to simulate an arbitrary Turing machine. This is a loose definition because it is not immediately clear what it means for a dynamical system to simulate a Turing machine. There are many possible ways to create a more precise definition; we will follow the method of Delvenne et al. [CITATION]. Before we do, though, we note a few reasonable expectations about what simulating a Turing machine should mean:

- If a dynamical system can simulate an arbitrary Turing machine, we should be able to take any combination of Turing machine and input, and convert it into some question about the system. The question should be "equivalent" to the question of whether the Turing machine would accept its input, in the sense that it should have a "yes" answer if the Turing machine accepts, and a "no" answer if the Turing machine rejects or never halts.

- The Universal Turing Machine is a specific Turing Machine that exists, and therefore it is a dynamical system on the space of its possible configurations. We would definitely expect this dynamical system to be considered universal.
- There are a number of other computational models which have ability to simulate Turing Machines, and which can also be considered dynamical systems. One good example is cellular automata, which are like Turing machine tapes where every cell has its own state and updates itself based on its neighbors' states. If a given instance of a computation model is capable of simulating arbitrary Turing machines, we would expect its associated dynamical system to be universal.

Formally defining universality

We will follow the lead of Delvenne et al. [CITATION] in defining what it means for a dynamical system to be universal. Like them, we will consider only *symbolic* dynamical systems, that is, dynamical systems on spaces consisting of infinite words made out of symbols drawn from a finite alphabet. We will need to do some spadework before we're ready to give the full definition.

Spadework Part I - R.E.-completeness

Our ultimate goal is, given the code for a Turing machine M, and the input w, to ask a question about a particular dynamical system which has a "yes" answer exactly when M accepts w. In the terms used in the field of computability, we want to be able to reduce an arbitrary instance of the question "Will Turing machine M accept the input w?" to a question about the dynamical system.

Fortunately, much is already known about the question "Will Turing machine M accept the input w?". In particular, it is equivalent to the question "Will Turing machine M halt on the input w?" in the sense that if we had a way to answer either of the questions, we could easily answer the other as well. This second question is called the Halting Problem, and computer scientists say that it is "complete for the class of recursively enumerable problems."

So, depending on your background, you may be asking two questions: "What is the class of recursively enumberable problems?" and "What does it mean to be complete for it?" A problem is called recursively enumerable if it can be solved by a Turing machine that may not ever halt, that is, if the question has a "yes" answer, the Turing machine will accept in finite time, but if the question has a "no" answer, the machine may run indefinitely. They are called "recursively enumerable" because it would be possible for a Turing machine (a computation model capable of *recursion*) to *enumerate* every input that would have a "yes" answer, in such a way that every such input would eventually be listed in finite time. (The machine could do this by listing all inputs lexicographically, and spending half its time simulating the computation on the first input, a quarter of its time on the second input, and so on, printing out any input when it is accepted.)

A given problem, call it P, is complete for a class of problems, C, if two conditions hold. First, P must be an element of C. Second, all problems in C must be reducible to P, meaning that if $P' \in C$, we can convert any instance of P' to an instance of P that will have the same answer, without doing an unreasonable amount of work in the conversion (for our purposes here, any finite computation time is reasonable). The Halting Problem is known to be complete for the recursively enumerable problems, or r.e.-complete.

The question "Will Turing machine M accept input w?" (call it Q) is also r.e.-complete. Recall that our goal was to find a question QDS about a dynamical system, which we could

reduce Q to. One way to express this constraint is that QDS should be r.e.-complete: then both QDS and Q could be reduced to each other, so they would be equivalent questions, which is what we are looking for. In fact, we will see that the r.e.-completeness of QDS is exactly what we will require.

Spadework Part II - Effective Symbolic Systems

A symbolic set can be thought of as a set of words made from a finite alphabet A. We could express such a set as $(A \cup \{B\})^{\mathbf{n}}$, meaning the set of one-sided infinite words with characters which are either drawn from A or are the "blank" symbol B. Finite words would then end be expressed as infinite words ending with an infinite tail of Bs. But we could encode each element of $A \cup \{B\}$ with a binary sequence (with all encodings the same length), so any symbolic set can be encoded as a subset of the set $\{0,1\}^{\mathbf{n}}$ of one-sided infinite binary words.

We give $\{0,1\}^{\mathbf{n}}$ the following distance metric d: $d(x,y) = 0$ if x and y agree at all indices, and otherwise $d(x,y) = 2^{-n}$ where n is the smallest index at which x and y differ. Under this metric, the set of all sequences beginning with a finite binary word w is a both a closed and open set (a "clopen" set) under this metric. Call sets like this, generated by a common prefix, cylinders. It can be shown that the clopen sets of $\{0,1\}^{\mathbf{n}}$ are precisely the finite unions of cylinders. What is special about this is that we can associate with each clopen subset of $\{0,1\}^{\mathbf{n}}$ a unique finite set of finite words, which are the generating prefixes for the cylinders that make up that set. So we can express every clopen set of $\{0,1\}^{\mathbf{n}}$ in a finite way, which means the we can list off all the clopen sets in some lexicographic order (in other words, the set is countable). We'll be exploiting this fact later.

Now we're ready to define the main dynamical systems we'll be working with. If $X \subseteq \{0,1\}^{\mathbf{n}}$ is a symbolic set and $f : X \to X$ is a continuous map on X, then (X, f) is an *effective symbolic system* if:

(i) X is closed.

(ii) Checking whether some clopen set $Y \subseteq \{0,1\}^{\mathbf{n}}$ has a non-zero intersection with X is decidable by a Turing machine in finite time.

(iii) The inverse map f^{-1} on clopen sets of X can be computed in finite time.

These requirements may seem a bit arbitrary, but they are made for a good reason. We will shortly be performing some operations on the clopen sets of X, and it turns out that, since X is closed, these are exactly the intersections of X with the clopen sets of $\{0,1\}^{\mathbf{n}}$, so we know this is a countable set. Requirement (ii) helps ensure that a Turing machine could list out these clopen sets without ever getting stuck checking if a particular clopen set of $\{0,1\}^{\mathbf{n}}$ has a nonempty intersection with X. And requirement (iii) foreshadows operations we will be doing on clopen sets.

Spadework Part III - Temporal Logic

Recall that we are looking for a way to "ask questions" about dynamical systems, whose answers will tell us something about the behavior of Turing machines. In order to be able to construct such questions, we will use subsets of state space to encode logical statements. Since we want to be able to make statements about the *eventual* behavior of the system, we will use a logical framework called temporal logic. Temporal logic includes all of the standard logical operators:

- $\top$: always true
- $\bot$: always false

- $\vee$: or
- $\neg$: not
- $\wedge$: and (which is actually redundant because it is constructible from $\vee$ and $\neg$)

It adds two additional operators to the list:

- $\circ$: next, a unary operator, with $\circ\phi$ meaning that ϕ will be true after one time step.
- $\mathcal{U}$: until, a binary operator, with $\phi\mathcal{U}\psi$ meaning that ψ will be true within finite time, and for all time between the present and one step before the time when ψ is true (inclusive), ϕ will be true

The developers of temporal logic, Arthur Prior and Hans Kamp, have described a set of rules for incorporating these operators into classical logic in a way that is consistent with our interpretation of them [CITATION]. For our purposes, suffice it to say that it works.

How do we propose to use temporal logic to construct statements about effective symbolic systems? Well, as we have suggested above, we are interested in operating on the clopen sets of symbolic systems, and here is where that will come into play. Let's say we have some effective dynamical system (X, f). We will use the fact that the clopen subsets of X are countable, and let $P_0, P_1, P_2, ...$ be a listing of all the clopen subsets of X (including $\emptyset$ and X in some positions).

Then let $\mathcal{P}_0, \mathcal{P}_1, \mathcal{P}_2, ...$ be a set of propositional symbols, which are the logical "units" that along with the operators above are the building blocks of temporal logic formulas. The listing will contain $\top$ and $\bot$ in some positions. We're going to define an "interpretation" operator $|\cdot|$ that takes logical formulas to subsets of X. The intuition behind it will be that if ϕ is a logical formula that expresses something about the "current configuration of the system," $|\phi|$ is the set of possible system configurations (points of X) for which that statement is true. Formally, the operator will behave like this:

- If ϕ is just the symbol $\mathcal{P}_n$, $|\phi| = P_n$, and we stipulate that the orderings should align such that $|\top| = X$ and $|\bot| = \emptyset$. This means that the symbol $\mathcal{P}_n$ represents the statement "The current system configuration is in the set P_n," which is of course always true if $P_n = X$ and never true if $P_n = \emptyset$.
- $|\phi_1 \vee \phi_2| = |\phi_1| \cup |\phi_2|$, because for either ϕ_1 or ϕ_2 to be true, the system can be in any state in which either is true.
- $|\neg\phi| = X \setminus |\phi|$, because the set of states for which ϕ is not true is the complement of the set of states for which it is.
- $|\circ\phi| = f^{-1}(|\phi|)$, which means that $\circ\phi$ represents the statement, ϕ will be true after one application of f, meaning that each application of f corresponds to one "time step" within the temporal logic system. Note that we know that this is computable, by requirement (iii) of effective symbolic systems.
- $|\phi_1 \mathcal{U} \phi_2| = \bigcup_{n \in \mathbf{n}} A_n$ with $A_0 = |\phi_2|$ and the other sets defined by the recurrence relation $A_{n+1} = f^{-1}(A_n) \cap |\phi_1|$. Here A_n can be interpreted as, the set of system states for which ϕ_2 will be true on the n^{th} time step from the current state, and ϕ_1 will be true on the $0, ...n-1$ time steps. The union of all of these represents all states for which "ϕ_1 is true until ϕ_2 is eventually true."

  So, in short, we have used temporal logic to construct a set of formula that express statements about the state of the system. We say a given formula is satisfiable if

its interpretation is non-empty, meaning that there is some nonempty set of X that satisfies it.

Universality

Our spadework concluded, we're ready to say what it means for an effective dynamical system to be universal. We will use Delvenne et. al's precise definition here:

"*An effective dynamical system is* universal *if there is a recursive family of temporal formulae such that knowing whether a given formula of the family is satisfiable is an r.e.-complete problem.*"

A "recursive" family of temporal formulae is a set of temporal formula for which membership in the set can be decided by a turing machine in finite time; this is essentially a regularity condition which prevents the family from being outlandishly defined.

To get a feel for what this definition means, and ensure that it is reasonable, let's start by confirming that the Universal Turing Machine dynamical system is, as we expected from the start, universal.

A configuration of the Universal Turing Machine, as with all Turing machines, is defined by finite control state and its tape contents, both of which can be encoded in a binary strings. So the configuration space of a UTM is a subset X of $\{0,1\}^{\mathbf{n}}$, and the code for the UTM is some specific transition function on this space, f.