Can A Fly See A Checkerboard?

Erik Talvitie
Oberlin College
Santa Fe Institute

8/14/03

Abstract

Computational Mechanics provides techniques to discover structure and pattern both in data streams and in one-dimensional spatial systems. It is not clear, however, how these techniques might be extended to study two-dimensional systems. Many possible approaches to pattern discovery in spatial systems of more than one dimension will be discussed, especially a technique called "All-Paths Reconstruction," which illustrates some of the subtle issues that appear when attempting to uncover spatial structure.

Introduction

The ability to create models from observations lies at the core of scientific inquiry. Given some data, we would like to be able to construct a model of the system that created that data and also be able to use that model to describe aspects of the system and predict the future behavior of that system. Computational Mechanics has proved to be a successful technique in a wide variety of situations for doing just that. Given a series of measurements, we can use ϵ-machine reconstruction to obtain a model that describes the statistics of the data, that gives us information about the underlying process, and that is able to generate a data stream as a prediction of the future.

While Computational Mechanics is intrinsically temporal, using past and future as key concepts, reconstruction techniques have been successful at studying one-dimensional spatial systems by reading them in from left to right and assigning past to the left and the future to the right. However, in order to extend Computational Mechanics to higher dimensional spatial systems, we will need a new technique. There is no natural path through two-dimensional space as there is in one dimension and so if we would like to be able to discover pattern and structure in a two-dimensional system, Computational Mechanics will have to be extended to include a concept of space.

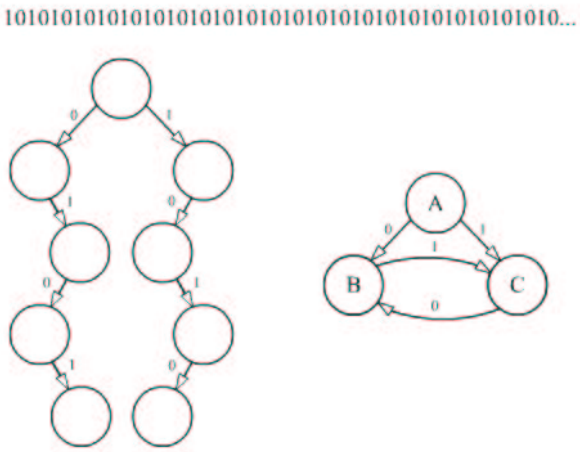

Figure 1: The parse tree and resulting ϵ-machine for a string of alternating 1s and 0s

Computational Mechanics

Computational Mechanics is fully described and formalized in [1]. Here we will simply introduce a few key concepts underpinning the rest of this paper.

Computational Mechanics is based on the idea of defining an equivalence relation on the space of "pasts" so that two pasts are equivalent if the probability distribution of the future conditioned on those two pasts is the same. In other words, two pasts are equivalent if after seeing either of them, we expect the same thing to happen. The equivalence classes based on this relation are called causal states and they are the minimal, maximally-predictive representation of the process, given the data. From these states, we can construct a finite state machine, called an ϵ-machine, which will act as our model for the process.

When actually performing an ϵ-machine reconstruction, these equivalence classes are found using a parse tree. Basically, we read each substring of a certain length (this length will determine what sort of processes we are capable of discovering but for practical purposes, it must be finite) and build a tree out of the symbols we see. Each node represents a "past" (a string of symbols seen) and the subtree under each node represents a "future" (symbols it is possible to see next). Therefore, we can build our equivalence classes by comparing the subtree under each node. Nodes with identical subtrees (call "morphs") are equivalent. See Figure 1 and 2 for examples.

Formal Language Theory

Formal Language Theory is a full and rich area of study and we will here only briefly introduce concepts necessary for later understanding. A good, general

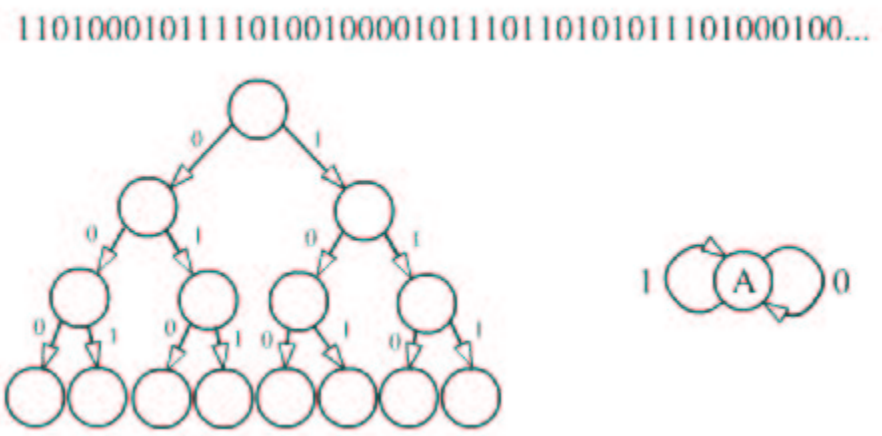

Figure 2: The parse tree and resulting ϵ-machine for a string of random coin flips (1 for heads and 0 for tails)

discussion of most important language theoretic concepts can be found in [2].

Define an alphabet to be a set of symbols and a word to be a finite collection of those symbols in some order. Then we define a language to be a set of words. What mainly interests formal language theory are the issues of deciding whether a word is in a particular language (called recognizing) or not and classifying languages according to their properties. The Chomsky Hierarchy of languages categorizes languages according to the difficulty of recognizing them. A brief overview of the main language classes follows:

•Regular Languages: Can be accepted by finite state machines (essentially machines that have finite memory)

•Context-Free Languages: Can be accepted by finite state machines that also have access to an infinite stack (machines with limited access to infinite memory)

•Context-Sensitive Languages: Can be accepted by finite state machines that can read and write to a tape whose length is proportional to the input size (machines with an arbitrary but bounded amount of memory)

•Recursively Enumerable Languages: Can be accepted by finite state machines that can read and write to an infinite tape (machines with infinite memory)

Two-Dimensional Formal Language Theory

Lindgren, Moore, and Nordahl [3] defined a two-dimensional extension to formal language theory. We define a patch to be a finite rectangle of symbols and a language to be a collection of patches. They found that the many equivalent

definitions of regular languages in one dimension, when extended to two dimensions, result in a hierarchy of distinct language classes. Briefly, this hierarchy of two-dimensional regular languages is (in order of increasing computational power):
•Local Lattice Languages (LLL): Can be described by a finite set of allowed sub-patches of some finite size
•DFA Languages: Can be recognized by a deterministic finite state machine which can move in one direction on every transition
•NFA Languages: Can be recognized by a non-deterministic finite state machine which can move in one direction on every transition
•Homomorphisms Of Local Lattice Languages (h(LLL)): Can be described by a LLL and a homomorphism on that LLL that sends the original alphabet to a smaller alphabet

Goals For Spatial Reconstruction

In order to adequately compare different reconstruction methods and settings, it will be useful to explicitely state our goals for reconstruction of a patch. We would like the result of our reconstruction to provide us with

•Description: We would like the result of our reconstruction to fully descibe the statistical properties of the patch as well as provide us with information about the effective memory and the unpredictability of the process.
•Recognition: We would like the result of our reconstruction to be able to recognize the output of the process as such.
•Generation: We would like the result of our reconstruction to be able to generate patches of arbitrary size that are consistent with the original process (this is the spatial equivalent to prediction).

Furthermore, we would like the result of our reconstruction to be an inherent property of process (or more accurately, of the language of patches created by that process) and not dependent on the particular path we choose through space.

One-Path ϵ-Machines

A naive setting would be to serialize our patch by reading it on a single path. However, in order to retain the spatial relationships between sites, we augment our alphabet and so record not only the symbol we see but also the direction our reader moved in order to see it. If we then treat the resulting string as a data stream and perform a traditional reconstruction on it, we get a machine like the one shown in Figure 3.

This one-path ϵ-machine is appealing mainly because of its close resemblance

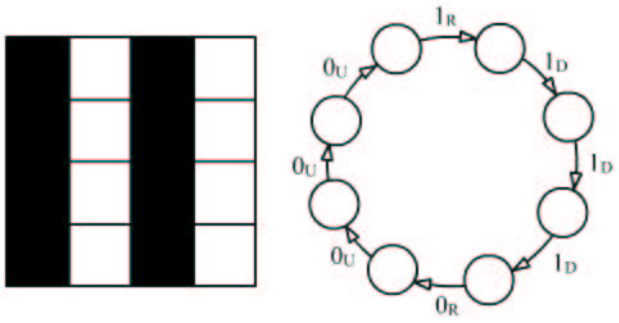

Figure 3: An example one-path machine (in this case for a vertical raster scan of an alternating stripes pattern

to the 4-way DFA described in [3]. Thus we are granted the recognition and generation benefits of this type of machine.

However, as a description of the system, this machine is very poor. Notice in the figure that a pattern of alternating stripes requires only one bit of memory (it is necessary only to remember what type of stripe the read/write head is currently in to know what should come next) and yet the machine has eight states, and thus an extra two full bits of memory. This is a result of the fact that the machine is keeping track of the path, which is itself period eight. So it is clear that this type of reconstruction depends heavily on the selected path and therefore does not serve our needs.

Right-Down ϵ-Machines

A more refined setting was described in [4]. A right-down ϵ-machine has an augmented alphabet similar to that of a one-path machine but here only the directions right and down are allowed and every state has at least one transition for each of those directions. An example is shown in Figure 4. This type of machine can be reconstructed by following all paths that go only right and down from every site in the patch.

The most immediately obvious benefit of the right-down machine is that it provides us superficially with a good description of the process. For an alternating stripes pattern, we see that our machine says that we have one bit of memory and no uncertainty in any direction, closely matching our intuition.

The main drawback of the right-down ϵ-machine is that it is unclear what it means for such a machine to recognize a patch. In fact, it is even less clear how such a machine could generate a patch of arbitrary size. Since it can only go right and down, it couldn't possibly visit every site.

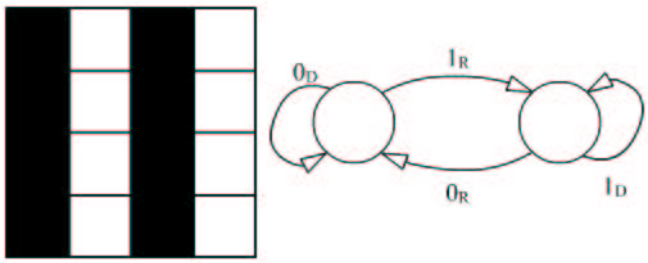

Figure 4: An example right-down machine (in this case for an alternating stripes pattern

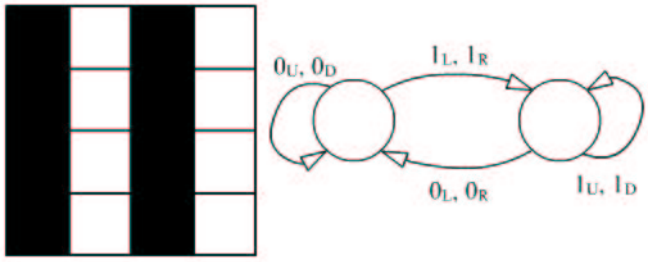

Figure 5: An example all-paths machine (in this case for an alternating stripes pattern

All-Paths ϵ-Machines

A slight extension to the right-down machine would be an all-paths ϵ machine. This type of machine is required to have at least one transition corresponding to all directions, rather than just right and down. See Figure 5 for an example.

An all-paths machine retains all the descriptive benefits of the right-down machine while also providing a natural definition of acceptance and procedure for generation as will be seen below. The formal definition is for a two-dimensional all-paths machine but can be easily extended down to one dimension or up to arbitrary dimensions.

Definition: All-Paths Machine An all-paths machine M is an pentuple, $M = (Q, \Sigma, \delta, q_0, F)$ where

Q is a finite set of states
Σ is the alphabet of the machine, namely the alphabet of the process, Σ_p, subscripted with direction (essentially $\Sigma_p \times \{R, L, U, D\}$).
$\delta : Q \times \Sigma \to Q$ is the transition function which sends a state and a symbol to a state
$q_0 \in Q$ is the start state of the machine
$F \subseteq Q$ is the set of final states of the machine

Note that because the transition function must send any state-symbol pair to a state, all states must have a transition in all directions. A move for an all-paths machine in state q can be described as follows:

1) Select a direction d (this can be done either randomly or by iterating along some path).
2) Move the head in the direction d.
3) Read the symbol a.
4) Change the state to $\delta(q, a_d)$.

Thus, an all-paths machine can be said to accept a patch if it reaches a final state at the end of *all* paths that visit every site on the patch.

An all-paths machine can generate in the same way a normal DFA would. One simply makes a series of moves but instead of reading symbols, allow the machine to write symbols.

Since the all-paths machine fits all of our criteria, it seems reasonable that we can design reconstruction settings that will result in this type of machine.

Limitations of the All-Paths ϵ-Machine

Since we are going to be using all-paths machines to describe spatial systems, we should be aware of its limitations: what sorts of systems it can adequately describe. To that end,

Theorem: *Any two-dimensional language recognized by an all-paths machine is LLL.*
Proof: To show this, we will note that every final state in fact represents a set of possible neighborhoods. To see this, imagine that machine M is in state $q \in F$ at site $s_{i,j}$. Then to fill out the set of all possible von Neumann neighborhoods (namely all possible combinations of the values of the sites $s_{i+1,j}$, $s_{i-1,j}$, $s_{i,j+1}$, and $s_{i,j-1}$), one need only look at all possible combinations of transitions leading out from state q that lead to a final state. The possible values of $s_{i,j}$ can be determined by the incoming transitions to state q. To fill out the set of all possible neighborhoods of arbitrary shape and size, one can repeat the procedure for outer sites (in effect, causing the machine to follow all paths through the neighborhood). In any case, it is clear that because there are finitely many states, for any sized or shaped neighborhood, there can be only a finite number of possible neighborhoods for each final state.

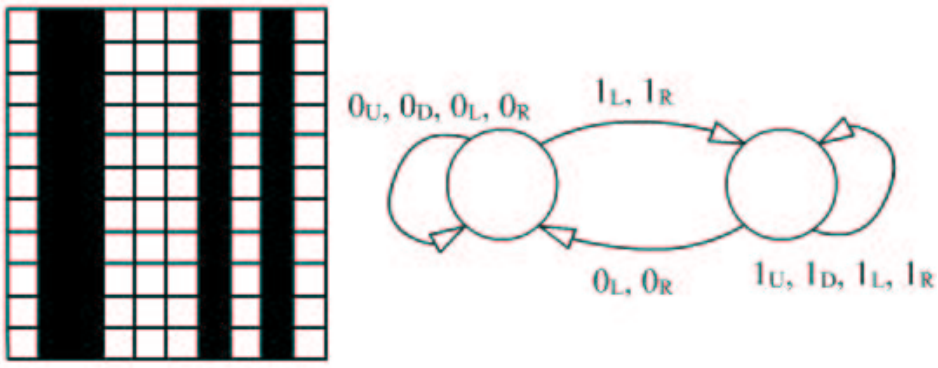

Figure 6: A random stripes pattern is made by flipping a coin across the top and then extending the ones and zeros down to fill the lattice

Now, in order to recognize a patch, the machine must reach a final state at the end of *every* path through the patch. Thus, it must be possible for the machine to be in at least one final state at any site. Then the (arbitrarily sized and shaped) neighborhood about every site must be one of the possible neighborhoods about the final state reached at that site. Since there are only finitely many final states and only finitely many possible neighborhoods for each final state, the patch can be described using a finite set of allowed neighborhoods of some size and shape. QED

It remains to be proven whether all LLL languages can be recognized by an all-paths machine and therefore whether all-paths machines are equivalent to LLLs.

All-paths machines' generation abilities are actually more limited than their recognition abilities. There exist languages that an all-paths machine can recognize but cannot generate. A simple example is that of random stripes (see Figure 6). The language is the set of all patches which consist of vertical black (1) and white (0) stripes in any configuration. The machine shown does, in fact, recognize the language of random stripes because, if on some path it encounters a 1 and a 0 vertically adjacent to each other, there will be some path for which the machine is not in a final state (trap states are implied but left out of the diagram). However, Figure 7 shows the result of having that machine generate over two different paths. While the first is clearly a random stripes pattern, the second looks like a field of coin flips.

The reason for this lies in the way the machine keeps track of which type of stripe it is in. For the alternating stripes example, there was a process to determine which type of stripe (black or white) the machine was currently reading. For the random stripes example, the process for determining this is simply to make a random decision. The problem is that if the machine has

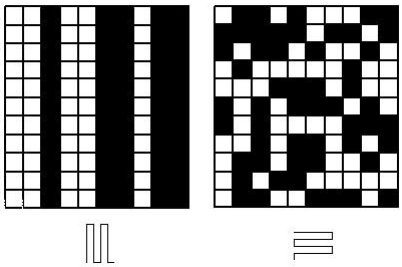

Figure 7: Two generating paths of the random stripes machine (the first is a vertical raster-scan and the second is a horizontal raster-scan)

already visited a stripe, that stripe has a value and therefore the machine should not make a random decision upon reentering that stripe but should instead print what it already printed. Hence, the machine must be able to remember the values of every stripe it has visited. Since the number of stripes is directly tied to the size of the patch, no one all-paths machine (which has finite memory) can possibly generate a random stripes pattern on all paths for any patch size.

Determining a specification or a test for the class of languages that cannot be generated by an all-paths machine remains an open problem.

Despite these limitations, all-paths machines remain as good descriptions, recognizers, and generators for a wide variety of spatial systems and so we will move on to developing ways to reconstruct them.

All-Paths Reconstruction

The most natural setting in which to reconstruct all-paths machines is a simple extension to the technique used to reconstruct right-down machines. Instead of following all paths that go right and down from each site on the patch, we now follow all paths from each site, augmenting the alphabet as we did before.

For the sake of simplicity and the ease of understanding, for the remainder of this paper we will consider one-dimensional all-paths reconstruction. All previous definitions and proofs can be easily extended to the one-dimensional case.

The Hidden Structure Of Space

An ϵ-machine describes a process and when reconstructing, we strive to capture the process and not just the specific example presented to us. However, it is inevitable that the lattice does in fact contain a specific example. This actually imposes a sort of implicit spatial memory, which adds structure to our system. The nature of space puts restrictions on the strings it is possible to see. Consider

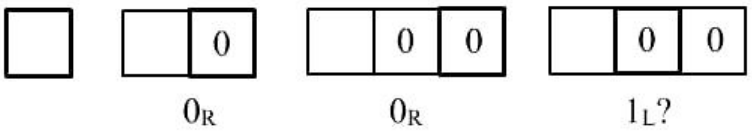

Figure 8: The string $0_R0_R1_L$ is self-contradictory because it claims to have seen a 0 and a 1 in the same site.

Figure 8. From it, we can see quite clearly that the string $0_R0_R1_L$ can never be read because it claims to have seen both a 1 and a 0 at the same site, which is impossible.

These restrictions manifest themselves in all-paths reconstruction in added structure to the resulting machine. In Figure 9, we see an ideal parse tree if we were going to reconstruct a string of coin flips. Since all morphs are identical, we have only one equivalence class and hence one state that goes to itself on any symbol. This parse tree would occur if the language we were reconstructing were Σ^*, the language of all possible strings made from the alphabet. However, because of spatial memory, there are some strings we cannot see and therefore the language we are examining is a proper subset of Σ^*. We will call this language LPS, the language of Legal Path Strings. Figure 10 shows the actual parse tree that results from an all-paths reconstruction at path depth 4 on a string of coin flips and the resulting machine. Note that the machine has no recurrent component. In fact, as path depth increases, so do the number of states and no recurrent component ever forms.

The Complexity Of LPS

In order to understand more fully why these restrictions on our language cause so much extra complexity in our machine (in fact, effectively an infinite amount of complexity), it is helpful to place LPS in the Chomsky Hierarchy of languages. Without loss of generality, we will consider the specific case where the process alphabet is {0, 1}.

Proposition: LPS is not regular
Proof: We will assume LPS is regular and seek a contradiction of the pumping lemma. Let n be the pumping lemma constant and consider the string $z = {0_R}^n 1_R {0_L}^n 1_L$. By the pumping lemma, then, we can write $z = uvw$, where $|uv| \leq n$ and $|v| \geq 1$ and such that $uv^iw \in LPS$. However, consider the case when $i = 2$. Since $|uv| \leq n$, we know that $v = {0_R}^m$ for some $m \leq n$. Then when $i = 2$, we have the string ${0_R}^{n+m} 1_R {0_L}^n 1_L$. Since $n + m \geq n$, this path string claims to have seen both a 1 and a 0 m steps to the right of where it started. Hence, this path string is illegal and we have found a contradiction. QED

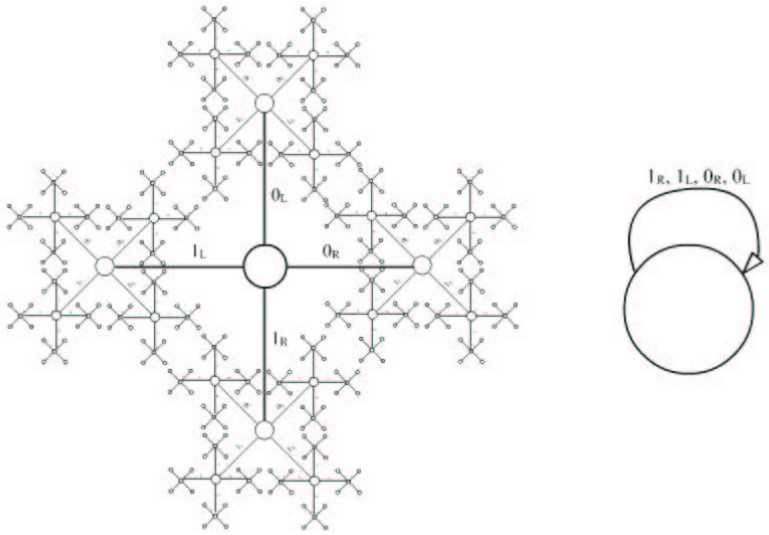

Figure 9: The ideal all-paths parse tree for reconstructing a string of coin flips. Note that all possible strings are seen and so all morphs are identical, giving the resulting machine a single state.

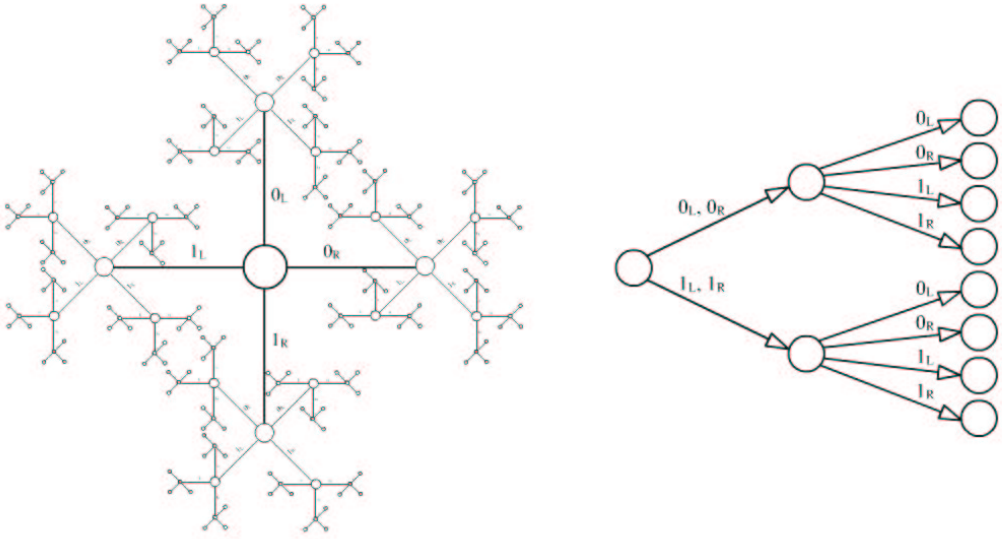

Figure 10: The actual parse tree generated by an all-paths reconstruction with path-depth of four on a string of coin flips. Note that not all possible strings are seen and hence the resulting machine has added structure.

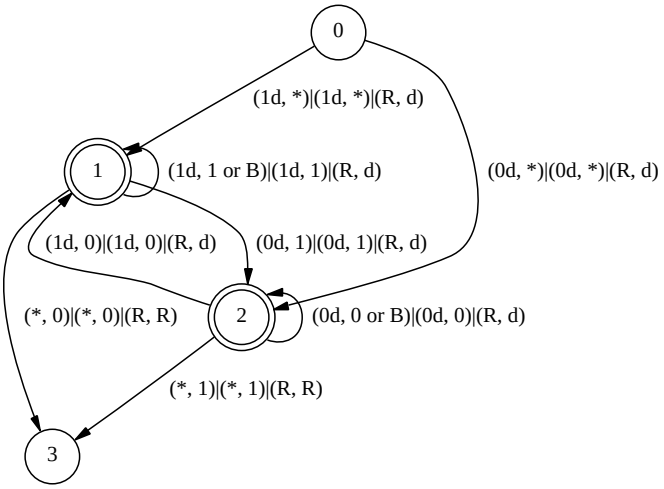

Figure 11: A graphical representation of the linearly bounded turing machine that accepts LPS. Transition labels can be interpreted as (Head 1 Reads, Head 2 Reads)|(Head 1 Writes, Head 2 Writes)|(Head 1 Moves, Head 2 Moves). In order to reduce the number of transitions on the diagram, wildcards have been used. "*" means any symbol. A "d" should be interpreted as either L or R. If a wildcard is used twice in one transition label, the intent is that both instances are the same.

Proposition: LPS is context-sensitive
Proof: We will construct a linearly bounded Turing machine that accepts LPS. Consider a two-headed, two-tape Turing machine where at each move, each head moves, reads, and writes independently and must stay on its own tape. The input is written on Tape 1 (with Head 1 starting on the left-most symbol) and Tape 2 starts off blank. For clarity, we will say that each symbol has a symbolic component (1 or 0) and a directional component (the subscript). At each move, Head 1 reads a symbol, prints nothing, and moves right. Head 2 reads a symbol, prints the symbolic component of the symbol read by Head 1 on the *previous* move, and then moves in the direction of the directional component of the symbol read by Head 1 on *this* move. If at any point, Head 2 reads a non-blank symbol other than the one it is going to print, the string is inconsistent and the machine can reject. A graphical example of such a machine is shown in Figure 11. Since Head 1 never writes anything, Tape 1 needs only the amount of memory necessary to store the input. Head 2 essentially follows instructions from the input and can therefore write at most a number of symbols equal to the number of symbols in the input. So this machine requires a tape size proportional to the size of its input and it is therefore linearlly bounded. Therefore, FPS is context-sensitive. QED

The immediate question is of course, is LPS context-free? It has yet to be

proven one way or the other, however, intuitively we can guess that the answer is no. In order to be context-free, LPS would need to be recognized by a PDA, a finite-state machine with a single, infinite stack. It seems as though in order to recognize this language, the machine must remember an arbitrary number of both left and right moves (because if it comes back to the site where it started, it should not forget everything it's already read). This would require two stacks (a two-stack machine is equivalent to a Turing Machine). At the moment, however, this remains only an intuition.

Settings In Which All-Paths Reconstruction Is Useful

Because the inherent structue of space is non-regular, All-Paths Reconstruction does not produce the desired machine for a wide variety of systems. However, there are situations in which spatial memory does not pose a problem.

Where there is uncertainty, there exist sites on the lattice that, as far as the process is concerned, could have either a 1 or a 0 (for a binary alphabet) but as far as the lattice is concerned, have a specific value. This difference is the crux of the problem, since, in order to reconstruct the process, we want, in essence, to see both a 1 and a 0 in the same site, which the lattice will not allow.

One situation in which this does not pose an issue is when reconstructing processes that do not have any uncertainty. Since the values of all sites are completely determined, in order to reconstruct the process, we do not ever expect to see both a 1 and a 0 in the same site, and therefore the restrictions space places on us are not restrictions at all.

Another situation one could imagine where all-paths reconstruction could be useful is one where the patch it is reconstructing is active. If as the reconstruction is taking place, every once in a while, the patch is redrawn, with all random decisions made randomly once again, spatial memory would essentially be eliminated. In this situation, the reconstruction would identify regularities and invariants in the system, though it is unclear whether it would be able to learn temporal patterns.

A similar situation to the above would be if the pattern on the lattice were being laid down by an agent. If, while the reconstructor were reading the patch, the writing agent were allowed to continue writing around it (perhaps at a much faster time scale), again the spatial structure would be destroyed and all-paths reconstruction could infer a model of the writer agent's internal process.

Other Possible Reconstruction Settings

In all-paths reconstruction, we essentially forced all of the spatial knowledge into our alphabet and therefore our language. This has turned out to make the results of our reconstructions undesirable for many patterns due to the fact that our now spatial language is not regular. Below are several ideas for future directions that have yet to be explored deeply. Each of them attempts to move

the knowledge of space to different parts of our reconstruction in the hope that we will be able to "factor out" the added structure.

"Beefing Up"

This idea was suggested by Carl McTague. The method is simply to avoid spatial reconstruction altogether. In essence, we place the knowledge that this is a spatial system in the human observer performing the reconstruction. In a one-dimensional system, we can do a traditional time-series reconstruction both "forwards" (left to right) and "backwards" (right to left). If we do both of these, we will get two machines, which, using a technique called subset reconstruction, we can combine into one machine that has both the right-going information and the left-going information. In other words, we construct an all-paths machine out of two one-path machines.

This technique may prove to be useful in reconstructing one-dimensional all-paths machines. However, it is unclear how to extend it into the second dimension. How many one-path machines would we need and which ones?

Also, as we have shown, the set of languages accepted by an all-paths machine is a subset of the set of the local lattice languages, which is itself a proper subset of the regular languages. In other words, not all one-path machines can be "beefed up." Determining when it is possible to "beef up" a machine may prove to be difficult as well.

Left-Right Tree

In this setting, suggested by Jim Crutchfield, we attempt to make our parsing data structure more inherently spatial. Imagine that, instead of the traditional parse tree, we have essentially two parse trees coming out of one node: one we will label "left" and the other we will label "right". Here we treat the subscripts on our symbols, not as part of the symbol, but as movement commands on the tree. If we see a 0_R, we will move to the right (so if we are in the "left" tree, we move towards the root and if we are in the "right" tree, we move out from the root) and add a 0 to our parse tree, if the node does not already exist. Because this tree captures spatial position and adjacency, restrictions on our language will not cause missing branches in the tree.

This setting, like the one above, is well defined in one dimension but not clearly extendable to higher dimensions. What would the data structure look like in two-dimensions?

A more serious issue is a sort of "amnesia affect." Unlike in traditional reconstruction, there is a path from any node back to the root. Since the root of the tree symbolizes the state in which we have seen nothing, this means it is possible to forget information, an undesirable trait in our reconstruction. This forgetfullness property may be another indication that LPS is not context-free, since the data structure is essentially emulating a single stack, which seems not to be sufficient.

All-Simple-Paths

Since spatial restrictions only apply to paths that loop on themselves, it is logical to imagine following all paths that do not intersect themselves. A non-self-intersecting path is called a simple path. A simple path cannot contradict itself and so spatial memory becomes a moot point. Here we are allowing our reading mechanism to be structured in hopes that it can take spatial memory into account.

This idea has not yet been well explored, however one immediately obvious problem exists. In one dimension, the language of all simple path strings is certainly regular. However, in two dimensions, and presumably in higher dimensions, it is easy to show that this language is not regular. So, choosing this subset of LPS may not in fact create more correct reconstructions.

Local-Lattice Reconstruction

As we have noticed, each state in an all-paths machine represents a set of possible neighborhoods. It may be possible to somehow reconstruct causal states from neighborhoods. This would require the reading apparatus to be inherently spatial, possibly reading in neighborhoods instead of individual symbols.

This possibility is still very vague and it is not clear what sort of reconstruction algorithm would be appropriate. Indeed, since it has not been proven that the languages accepted by all-paths machines are exactly equivalent to the local lattice languages, it may be that there exist some languages for which it is impossible to reconstruct causal states from local lattices.

Conclusions

To directly address the question posed by the title of this paper, yes, a fly can see a checkerboard. Spatial structure does not pose a problem in the specific case of zero-entropy patterns. However, in exploring this, we have seen that the inherent memory in space is a subtle issue that must be considered when attempting to discover patterns in spatial systems. It is unclear what is the best way to "factor" out this unwanted structure or in fact if there is a best way. It may be the case that there is no one way to study a spatial system but instead a variety of techniques, each dealing with spatial memory in a different way, that are appropriate for different situations.

It should also be noted that we have been assuming throughout this study that our ultimate goal for a reconstruction should be a finite state machine of some sort. These machines have served us well in the study of formal languages and so seem a natural choice. However, finite state machines are serial in nature, even if space is built in to the definition, as with the all-paths machine. It may be possible to abstract the idea of causal states to something that is more inherently spatial and for which, therefore, the existence of spatial memory will not be a problem. Exactly what the nature of this structure would be or how

one would go about reconstructing it are deep questions that go to the heart of the nature of causality and pattern.

References

[1] C. R. Shalizi and J. P. Crutchfield. Computational mechanics: Pattern and prediction, structure and simplicity. *Journal of Statistical Physics*, 104:819–881, 2001.

[2] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[3] Kristian Lindgren, Christopher Moore, and Mats Nordahl. Complexity of two-dimensional patterns. *Journal of Statistical Physics*, 91:909–951, 1998.

[4] D. P. Feldman. Computational mechanics of classical spin systems. *Ph.D. Dissertation, Physics Department, University of California, Davis*, 1998.