BU CAS CS 332, Spring 2009: Section Notes

Andrei Lapets

May 4, 2009

This document contains notes corresponding to the material that has been or will be covered during the discussion sections of the spring 2009 iteration of the course BU CAS CS 332, taught by Professor Leonid Levin. These notes will contain intuitive exposition, as well as problems and examples that demonstrate how the definitions and theorems presented during lectures and in the official lecture notes can be instantiated and used. **These notes should not be used to determine what material is required for the course.** The required material for the course is limited to the material presented in the lectures, the lecture notes, the homework assignments, and anything explicitly assigned within the course textbooks. This document will be updated over time; the date on this title page corresponds to the day on which the latest changes were made.

The intention is for these notes to present the material in a form that is useful to you for understanding the lectures, solving problems, and studying for exams. You are encouraged to send questions, requests for improvements, actual improvements, clarifications, etc. for this document. All contributions can be explicitly acknowledged if desired, though they may eventually be edited or modified. Some of the sections in these notes are elaborations and extensions of the material found in the official lecture notes (`http://www.cs.bu.edu/fac/lnd/toc/`), due to Leonid A. Levin and his contributors.

1 Algorithms

We wish to study the collection of all algorithms[1] in a rigorous manner in order to make precise observations about their properties. What is an algorithm? Is

“find the solution to $x^3 = 64$ over $\mathbb{R}$”

an algorithm?

Our only real option is to try to define a collection of objects that seem to respect our intuitions about what an algorithm should be, and to study the properties of that collection of objects, instead. For example, the Church-Turing thesis:

“[the collection of algorithms is equivalent to the collection of Turing machines.]”

We can never prove this, yet we must somehow convince ourselves that this hypothesis is reasonable. We can improve our confidence by using our intuition to define *many* such collections of objects, and then showing that they are all equivalent. We will do so by looking at a variety of precise definitions corresponding to various intuitions, such as cellular automata, Turing machines, and pointer machines. All of these definitions turn out to be equivalent. Thus, in studying the properties of any of these collections of objects, we can be relatively confident that we are indeed studying what we call “algorithms.”

2 Turing Machines

In this section, we develop in detail the definition of a multi-head Turing machine.

2.1 A Kind of Cellular Automaton

Turing machines consist of an infinite tape of *cells* in which the left-most cell has one neighbor, and all other cells have exactly two neighbors:

						...

Each cell holds a piece of information represented by an element in an *alphabet* (a finite set). We usually use Σ to denote this set. So, for example, we might have $\Sigma = \{A, B, C\}$, or $\Sigma = \{0, 1\}$.

The tape is infinite, but many computations do not require an infinite tape. Thus, we typically require the existence of an element in the alphabet that denotes an “empty” cell. Suppose # is used to denote an empty cell. Then we might have an alphabet such as

$$\Sigma = \{0, 1, \#\},$$

and while operating, a Turing machine with this alphabet might look as follows:

1	0	1	1	0	#	#	#	...

Each cell of the Turing machine operates according to a transition function $\delta : \Sigma^3 \to \Sigma$ (here, we chose a transition function that can only see the two adjacent cells, but any constant number of adjacent cells would work as well, so long as it is used consistently). During each step of a global clock, all cells simultaneously

[1]Here, we speak only of all *deterministic* algorithms.

compute what contents they will hold during the next clock step. Each cell takes the contents of its neighbor cells, along with its own contents, and uses δ to compute its new contents:

$\ldots$	x_1	x_2	x_3	x_4	x_5	$\ldots$

$$\Downarrow$$

$\ldots$	$\delta(x_0, x_1, x_2)$	$\delta(x_1, x_2, x_3)$	$\delta(x_2, x_3, x_4)$	$\delta(x_3, x_4, x_5)$	$\delta(x_4, x_5, x_6)$	$\ldots$

The transition function δ could be represented using a simple lookup table. For example, the following table might represent δ:

$\delta(1,1,1)$	$\mapsto$	1
$\delta(1,1,0)$	$\mapsto$	0
$\delta(1,0,1)$	$\mapsto$	0
$\delta(1,0,0)$	$\mapsto$	0
$\delta(0,1,1)$	$\mapsto$	0
$\delta(0,1,0)$	$\mapsto$	0
$\delta(0,0,1)$	$\mapsto$	0
$\delta(0,0,0)$	$\mapsto$	0
$\delta(\#,\#,\#)$	$\mapsto$	#
$\delta(0,\#,\#)$	$\mapsto$	#
$\delta(1,\#,\#)$	$\mapsto$	#
$\delta(0,0,\#)$	$\mapsto$	#
$\delta(0,1,\#)$	$\mapsto$	#
$\delta(1,0,\#)$	$\mapsto$	#
$\delta(1,1,\#)$	$\mapsto$	#

The left-most cell can have a special version of the transition function that can operate on only two inputs, where the second input corresponds to the contents of the second cell in the tape:

$\delta(1,1)$	$\mapsto$	1
$\delta(1,0)$	$\mapsto$	0
$\delta(0,1)$	$\mapsto$	0
$\delta(0,0)$	$\mapsto$	0
$\delta(1,\#)$	$\mapsto$	1
$\delta(0,\#)$	$\mapsto$	0
$\delta(\#,\#)$	$\mapsto$	#

Question 2.1. Suppose that the initial tape of the Turing machine consists of a list of binary digits (a finite string of zeroes and ones), followed by empty cells. If the machine operates according to the transition function δ, the state of the machine will eventually correspond to the output of what familiar function?

Question 2.2. Recall the Σ is always finite. Will it always be possible to represent δ using a table of finite size?

2.2 Turing Machines with Tape Heads

The previous description is effectively that of a cellular automaton restricted to a tape that has a beginning. However, in the real world, tapes are usually used as passive storage, and each piece of tape cannot perform transition operations. Instead, "heads" are used to modify a particular location on the tape (this is precisely how most hard drives work).

We can add extensions and restrictions to the previous definition to capture this intuition. First, we can imagine that at each tape location, up to c heads can congregate, where c is a constant factor that will never change. We can then extend δ to take as input not only the contents of the tape cells, but also some flag(s) that indicate how many heads are currently assembled at that location. Because at most c heads can assemble at any location, we can represent these flags using a fixed number of bits. Let F be the space of possible flag values. For example, if $c = 4$ then at most 4 heads can be found at any single location, so we might have

$$F = \{0, 1, 2, 3, 4\}.$$

Obviously, F could also distinguish between different kinds of heads, and we might have

$$F = \{0, 1, 2\} \times \{0, 1, 2, 3\}.$$

where $(0, 3) \in F$ indicates that there are 0 heads of the first type, and 3 heads of the second type at a particular location.

Then, δ can be a function from $(F \times \Sigma)^3$ to $F \times \Sigma$. Effectively, δ decides not only what the contents of a cell should be, but also how many heads there are at a cell's location. For example, if $\Sigma = \{A, B, \#\}$ and $c = 2$, we might have

$$\begin{array}{rcl} & \vdots & \\ \delta((1,A),(0,B),(1,A)) & \mapsto & (2,B) \\ \delta((0,A),(1,A),(0,B)) & \mapsto & (0,A) \\ \delta((0,B),(1,A),(0,A)) & \mapsto & (0,A) \\ & \vdots & \end{array}$$

Restriction: Number of Heads Now, if we allow δ to change the number of heads at any location arbitrarily, we may as well define a new alphabet $\Sigma' = F \times \Sigma$ and we will have our previous definition. Intuitively, the number of heads should not change during operation. Thus, we can restrict the space of permissable transition functions by requiring that any δ preserve the number of heads on the tape.

Restriction: Passive Tape Furthermore, we are trying to simulate a passive tape. Thus, we make the restriction that for all δ, for any $x_1, x_2, x_3 \in \Sigma$,

$$\delta((0, x_1), (0, x_2), (0, x_3)) \mapsto (0, x_2),$$

so that if no heads are found at a tape location, the tape remains unchanged.

Finally, in order to allow for the creation and removal of tape heads, we can relax the last two restrictions for the first (or second) cell. This cell would be free to create a head and to consume a head at its location, based on its contents. However, this cell would still need to respect the constraint that limits the number of heads at any particular location to c. Furthermore, it could only decide whether or not to create a head based on its own contents and the contents of its immediate neighbor(s), so if the machine wanted to create a head based on the contents of cells further down the tape, it could only do so if another head moved from that location back to the special first (or second) cell.

2.3 Other Extensions and Restrictions

We can make additional extensions to our definition. All of the extensions in this section can be simulated by changing Σ in the above definition.

2.3.1 Input

For example, we can specify an “input alphabet” I that is used only for supplying the input to the machine, and create a new alphabet $\Sigma' = I \times \Sigma$. Thus, each cell would hold both an input symbol, and possibly a symbol corresponding to the operation of the machine, as before. In this case, we could restrict δ by saying that δ can only produce outputs (F, Σ), never using an input alphabet symbol in its output.

2.3.2 Empty Cell Symbol

We can also make the restriction that the symbol for any empty cell, #, can only be used for the part of the tape that has never been visited by a head. Then, we would need to add additional special symbols to represent empty cells that *have* been visited by a head. Typically, this is a restriction that ensures that it is possible for a cell to recognize that it is at the “border” between the current computation and the “rest” of the tape. If we make such a restriction, it can make the definition of δ simpler. For instance, in the example above, we did not consider cases for δ in which blank spaces appeared to the left of binary digits.

2.3.3 General Extensions of the Alphabet

As long as an alphabet is finite, it is perfectly acceptable to create a finite number of “tracks” of a tape by subdividing the cells into smaller components. For example, we might have

$$\Sigma = (\{A, B, \emptyset\} \times \{X, Y, \emptyset\} \times \{0, 1, \emptyset\}) \cup \{\#\},$$

and an operating machine might look as follows:

number of heads	0	1	0	2	0	0	0	
input	A	B	A	A	A			
work	X	Y	Y	$\emptyset$	$\emptyset$	#	#	...
output	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$			

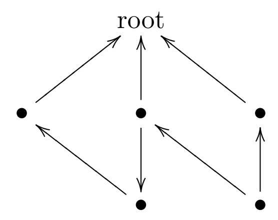
Each track can have its own alphabet, and the alphabet of the overall machine would simply be the cartesian product of these alphabets. However, note that the general symbol for an empty cell, #, is distinct from symbols for empty components of a cell, as it is still needed for indicating the part of the tape that has never been visited by a head.

3 Pointer Machines

We develop in detail the definition of a pointer machine. The structure and transition behaviors of pointer machines are subject to restrictions that are similar to those we encountered when considering cellular automata and Turing machines.

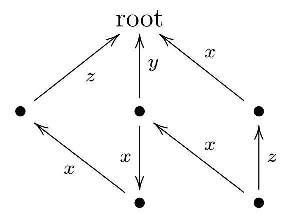
3.1 Pointer Machine Structure

A pointer machine is a finite connected graph with a distinguishing *root* node and directed edges. For example:



Intuitively, a directed edge $N \to M$ represents the fact that N can see M. We call the nodes that N can see its *children*, and all nodes that are on a directed path of length 2 from N its *grandchildren*. That is, if we have $N \to M \to O$, then O is the grandchild of N.

In this model, edges are used for holding pieces of information, represented by one of a finite set of *colors*. Thus, edges could correspond to cells in the previous model, and the set of colors could correspond to the alphabet. For example, if the set of colors is $\{x, y, z\}$, then we might have a graph as follows:



Pointer machines are meant to represent physically conceivable systems, so just as we did in our definitions of cellular automata and Turing machines, we introduce some restrictions. The outgoing edges of a node can be considered its contents or data, and the number of *other* nodes that can look and see the contents of a node is not limited. However, each individual node can only look at a finite number of other nodes at any given time.[2] We accomplish this by stating that for every node in the graph, the *outgoing* edges must have different colors.

Question 3.1. Suppose we have a pointer machine with the set of colors $\{a, b, c, d\}$ that satisfies the restrictions presented so far. Consider any node in the graph. How many outgoing edges (edges pointing away from the node) can exist? How many incoming edges can exist?

[2]One scenario that illustrates such a system might be a collection of radio stations on distinct frequencies: every radio station can broadcast its programming on its own frequency to anyone that wants to listen. However, only a finite number of radio receivers fit inside each radio station. Thus, each radio station can only listen to a finite number of other radio stations.

3.2 Pointer Machine Operation

3.2.1 Input/Output and Working Colors

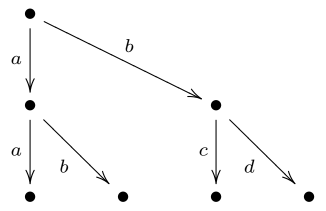
As with Turing machines, it is possible to separate the set of colors into distinct collections: input/output colors, and "working" colors. We can then make the restriction that when the graph is in its initial state, all edge colors must be from the set of input/output colors, and that when the computation stops, we consider only the edges with input/output colors as the "output" of the computation. Thus, we might have the following as a set of colors:

$$\{\underbrace{x, y, z}_{\text{i/o}}, \underbrace{a, u, v, w}_{\text{working}}\}$$

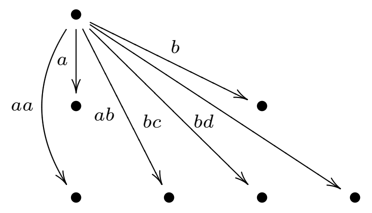
3.2.2 Transition Function

Each node in a pointer machine operates according to a transition function. During each step of a global clock, all nodes simultaneously do the following:

"Pulling Stage." The node acquires copies of all the edges of its children, and uses color composition to distinguish these edges. For example, suppose the following is a subgraph corresponding to the children and children's children of a node.



Then, after the pulling stage, the node has the following edges:



Notice that if the set of colors is C, the node can have up to $|C|^2 + |C|$ distinct edges during this stage.

Node Creation. The node can then create new nodes based on the colors of all its outgoing edges (pre-existing and composite). **In the model presented in lecture, the node is allowed to create edges (directed either to or from itself) only between itself and these new nodes. It cannot create any edges between new nodes, or between any of its pre-existing children.** [3] The usual restrictions apply to the newly-created nodes: the edges from the node must all have different colors.

Transformation. The node can recolor each edge based on the colors of all its outgoing edges (pre-existing and composite).

Drop. The node must then drop all edges that are of a composite color.

[3]Note that the newly created node can, in the next time step, obtain an edge to any of its creator's children, and in two time steps, an edge to any of its creator's grandchildren, so our examples from sections can be simulated under this restriction.

3.2.3 Active Nodes

In our definition of a pointer machine, we can introduce a restriction that corresponds to the notion of “head” in a Turing machine by allowing nodes to be *active*. At any given time step, a node can perform a transition operation as described above only if it is active. All other nodes act as passive storage, just like the passive tape in our definition of a Turing machine with an unbounded number of heads.

We make this restriction precise by designating one of the working colors as the *active color* (in this text, we will denote this color using a). A node is then considered to be active when it sees an edge that carries the active color. We require that any edge with an active color must have an inverse. Also, we make an exception for convenience: the root node can have two active color edges: the self-edge, and one other edge with an active color.

It is not clear from the lecture notes whether a node is active when an active edge is directed *towards* it,[4] or *away* from it. However, because active edges must have inverses, this is not very significant. For example, suppose that a node is active when an active edge is directed *towards* it. Then, the root node that has already been activated with a self-edge can activate the second node in the following example by changing the color of the edge labelled y to a:

$$\begin{tikzcd}
\text{root} \arrow[loop left, "a"] \arrow[r, bend left, "y"] & \bullet \arrow[l, bend left, "x"] \arrow[r, dotted, Rightarrow] & \text{root} \arrow[loop left, "a"] \arrow[r, bend left, "a"] & \bullet \arrow[l, bend left, "x"]
\end{tikzcd}$$

On the other hand, if we adopt a definition in which a node becomes active when an active edge points away from it, it is necessary to state that a node becomes active when it sees an active edge of one of its children. For example, in the following diagram, the non-root node will see that the root node has an active edge, and can then create its own active edge by modifying the color of the edge labelled x:

$$\begin{tikzcd}
\text{root} \arrow[loop left, "a"] \arrow[r, bend left, "y"] & \bullet \arrow[l, bend left, "x"] \arrow[r, dotted, Rightarrow] & \text{root} \arrow[loop left, "a"] \arrow[r, bend left, "y"] & \bullet \arrow[l, bend left, "a"]
\end{tikzcd}$$

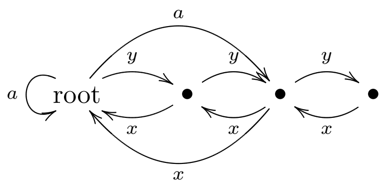
Now, anyone that directly sees the second node can also become active by seeing this new active edge. Thus, the two definitions of an active node are equivalent.

In either case, we make the additional restriction that active edges must form a tree to the root. Thus, an active node can never remove an active edge that leads to the root unless it sees no incoming active edges. You should convince yourself that this restriction is plausible because all active edges have inverses, and all active edges are always oriented in the same direction with respect to the root (either always away from it, or always towards it, as in the illustrations above). You should also convince yourself that maintaining this consistent orientation is not problematic: you would simply need to place additional constraints on the re-coloring portion of the transition operation. With this restriction, we ensure that the subgraph of active nodes is always connected.

However, notice that under these restrictions, it is still acceptable for non-active nodes to exist on paths between an active root node and some other active node. For example, below, the second node from the left

[4] I believe this is in fact the definition used in lecture, though it is the opposite of the one we used in section. Please let me know if this is not the case.

is not active, but the third one is:

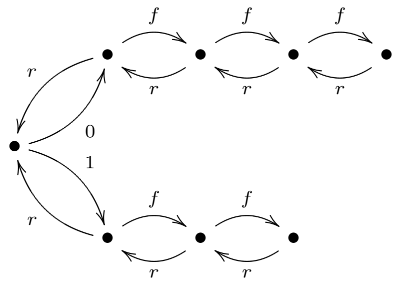


3.2.4 Sequential vs. Parallel Pointer Machines

Without additional restrictions, our definition corresponds to that of a parallel pointer machine. In a sequential pointer machine, we make the additional restriction that only nodes that have direct and inverse edges to the root (at some particular time step) can be active (during that time step). Thus, under this restriction, only a finite, bounded number of nodes can be active at any time step.

3.3 Example: Concatenating Lists

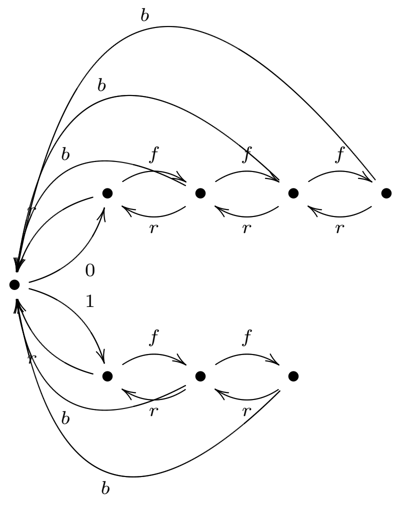
Let us try defining a sequential pointer machine that consists of two doubly-linked lists (seen by the root), and must produce the concatenation of these two lists. For this example, we will use the first variant of the definition of an active node: a node is active when an active edge points *towards* it.[5] The input might look as follows:



However, this input graph is not enough for us to be able to perform the concatenation under the existing constraints. In particular, it is impossible to create an active edge from the root to any of the non-head nodes of the lists. This is because they do not have an inverse to the root, so the requirement that all active edges must have an inverse would not be satisfied if the root were ever to create an active edge to those nodes.

[5]This is the opposite of the convention we used in our section discussion, but I believe that this is the convention used in the definition presented during lecture.

Thus, the actual input would need to look as follows, with edges from every node back to the root:



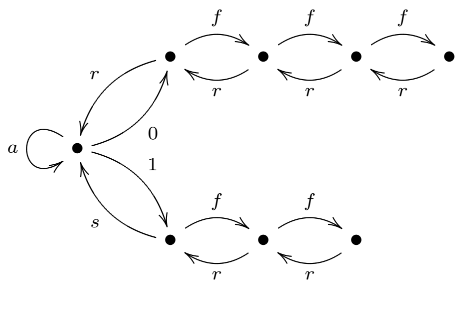
Homework Digression. This transformation from the first graph to the second one is an example of the transformation you will define in your homework assignment. That is, in your assignment, you start with a graph that *does not* have edges from all nodes back to the root, and you must create a copy of the graph where all nodes do have such an edge. In your homework, this will be possible because the only active nodes you will need will be the *new* ones your root node creates, and those can always be created with such edges back to the root.

To reduce clutter, we will omit these backward edges from our diagrams henceforward, but keep in mind that they exist, and that they are what allow us to create active edges from the root to any non-head node.

Our set of colors will consist of forward and reverse pointers, a label for each list (either 0 or 1), some additional labels for special pointers, and the active color:

$$\{\underbrace{0, 1, f, r, b}_{\text{i/o}}, \underbrace{a, p, q}_{\text{working}}\}$$

We could have additional edges and colors for representing the contents of the lists, but we leave this to the imagination. Now, all that remains is to define the transition operation that each active node will perform. The computation begins once the root is activated with the special self-edge:



We need to somehow create an edge from the last node of the top linked list to the first node of the bottom one. In order to accomplish this, we must make active the last node of the top list. We can do so by having the root node move an active pointer along the forward pointers of the top list.

Thus, the first step would be for the root node to relabel the 0 edge with a. That is, during the pulling stage, the root will have the following regular and composite edges (excluding the self-edge):

$$0, 1, 0r, 0f, 1r, 1f.$$

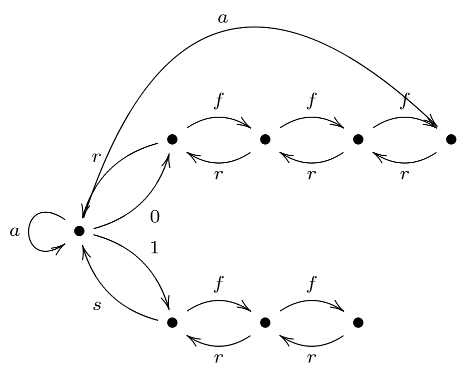
It will leave the 1 edge, and relabel the 0 edge with a. The other composite edges will be dropped during the final stage of the transition for the first time step. Next, the root node would follow the following procedure at any given time step:

- If after the pulling stage, the root node sees an outgoing composite edge labelled af and no composite edge as, it will drop the edge labelled a and relabel the edge labelled af to a (this will facilitate the traversal of the active edge all the way to the end of the top list).
- If it does not see such an edge, then the root must have an active edge pointing to the end of the top list unless one of the remaining cases below applies, so it will do nothing and wait (for the last node in the top list to move its forward pointer to the head of the bottom list).
- If it sees a composite edge as, it will do nothing and wait (for the head of the bottom list to get its reverse pointer to the end of the top list).
- If it sees an edge labelled $1q$, it will drop its 1 edge, and create a dummy node with an edge q pointing towards it (to give the head of the bottom list a time step to drop its own q node).
- If it sees a q edge, it will drop the q edge along with its active edge (the one pointing to the head of the bottom list), stopping the computation.

Thus, in the second time step, the pulling stage at the root will produce:

$$a, 1, ar, af, 1r, 1f.$$

The a edge will be removed and the edge af will be relabelled a. This process will continue at each time step until the graph looks as follows (once again, recall that we are omitting edges back to the root to avoid clutter):



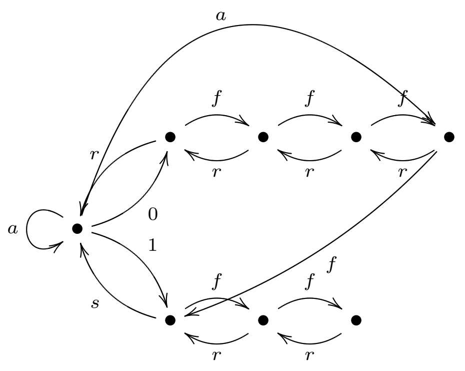
The root will then detect that it has reached the end of the list because it will not see an af edge in its composite edge list during the pulling stage.

Up to this point, we did not specify the behavior of each of the non-root nodes when they become activated. We must take into account that only the last node in the list can become activated. Thus, the non-root nodes should follow the following procedure:

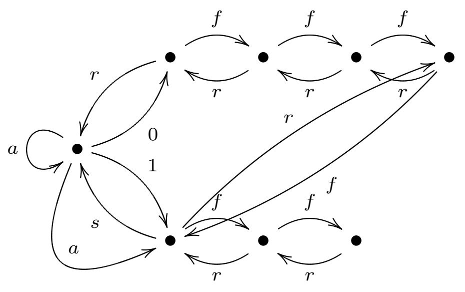
- If there is an edge f, do nothing unless one of the remaining rules apply.

- If there is no edge f, then this is a node that is at the end of the top list (the one at the end of the bottom list is never activated). Thus, there must be a composite edge $b1$ as well (back to the root, and to the head of the bottom list). Relabel this edge f to create the forward pointer to the head of the bottom list.
- If there is an edge $s0$, relabel this p.
- If there is an edge pf, drop the edge p and relabel the edge pf to p (this allows the bottom list's head to traverse the top list until it reaches the end).
- If there is an edge p but no edge pf, relabel pf to r and create a dummy node with an edge q to it.
- If there is an edge q, remove it, and remove the edge s if it still exists.

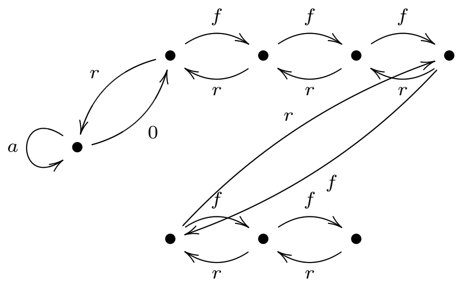
Suppose we are at the time step during which the root creates an active edge pointing to the last node of the top list. After the next time step, the graph should look as follows:



Now, during the next time step after this, the root node will again see an edge labelled af, and so will move the active edge to the head of the bottom list. Then, the root will see an edge as, and this will let it know that the head of the bottom list is now active, so it should wait. Meanwhile, the head of the bottom list will create a pointer to the head of the top list by relabelling the composite edge $s0$ with the label p. It will then traverse the list just as the root did earlier, until it obtains an edge labelled p pointing to the last node in the top list and relabels it to r:



At this point, the head of the bottom list will also create a dummy node with an edge labelled q to it. Then, the root node will see this. The root node will then create its own q edge. The head of the bottom list will then have a time step to remove its q node as well as the node s that returns to the root. Finally, the root will remove its q node, its 1 node, and its active edge pointing to the head of the bottom list, leaving the output:



Notice that when defining a pointer machine, it is okay to have a finite number of different procedures (in our case, one for the root node, and one for the non-root nodes). When defining them in such a manner in general, be sure to specify how a node will know which procedure to run (e.g. by looking at its surroundings). In our case, a node could determine whether or not it was the root by, for example, looking to see if it had an outgoing 0 edge.

4 Equivalence of Models of Computation

We are interested in arguing that various models of computation are equivalent. For the purposes of our discussion, we will use *model of computation* and a *collection of machines* interchangeably.[6] A machine is any object m that takes any binary input $x \in \{0,1\}^*$ and either diverges or produces an output in $\{0,1\}^*$.

We must first define precisely what equivalence between models of computation means. Because this is a definition, there is no correct way to do this; it is up to us to find a definition that corresponds to our intuition. One way we might characterize the fact that two models are actually the same is to use a notion of simulation.

Definition 4.1. A machine (in any model) m *simulates* a machine m' if for all inputs x, either $m(x) = m'(x)$ or both $m(x)$ and $m'(x)$ diverge.

We can now define equivalence between models in terms of simulation.

Definition 4.2. Two models of computation S_1 and S_2 are *equivalent* if for any $m_1 \in S_1$, there exists $m_2 \in S_2$ such that m_2 simulates m_1, and for any $m'_2 \in S_2$, there exists $m'_1 \in S_1$ such that m'_1 simulates m'_2.

For example, we can show that one direction of this definition is satisfied for pointer machines and Turing machines: for every pointer machine, the algorithm it represents can be simulated using a particular Turing machine.

Lemma 4.3. *For every pointer machine P, there exists a Turing machine M such that for every input x, $P(x) = M(x)$, or the computations $P(x)$ and $M(x)$ both diverge.*

Notice that this argument does not yet show that the two models are equivalent.

Question 4.4. Assuming that Lemma 4.3 holds, what other statement do we need to prove in order to show that the two models of computation are equivalent?

We will subdivide the proof of equivalence further by using an intermediate model of computation, the Kolmogorov machine. We will first show that the pointer machine model is equivalent to the Kolmogorov machine model, then argue that the Kolmogorov machine model is equivalent to the Turing machine model. Thus, this proof will take advantage of the fact that the definition of equivalence over models of computation is transitive.

Exercise 4.5. Using only the definitions introduced so far along with the transitivity of equality for binary numbers, write a formal proof that our definition of equivalence for models of computation is transitive.

4.1 Kolmogorov Machines

The definition of a Kolmogorov machines is the same as that of a pointer machine, with the additional restriction that the number of incoming edges to a node must also be bounded by a constant. Equivalently, we could require that at every time step all edges in the graph of the machine must have inverses.

4.2 Proof of Lemma 4.3 (First Half)

We are given an arbitrary pointer machine, and must construct a Kolmogorov machine that has the same input/output behavior (as required by our definition of simulation). We call this Kolmogorov machine the simulator.

[6]For example, we might have the collection of all multi-head Turing machines, and the collection of all pointer machines.

4.2.1 Structure of Simulator

We first describe how the structure of the Kolmogorov machine will simulate the structure of the pointer machine at each time step. In particular, a node in the pointer machine may have arbitrarily many incoming edges at some time step. Thus, the Kolmogorov machine will simulate a single pointer machine node with a special tree structure that consists of:

- a node, corresponding to the original pointer machine node;
- a balanced binary tree in which each leaf has exactly one incoming edge of the original pointer machine node.

Thus, by representing a node using an arbitrarily large balanced binary tree, we can satisfy the constraint of the Kolmogorov machine definition while still maintaining an arbitrary number of incoming edges to each simulated "node."

4.2.2 Transition of Simulator

We have defined how the simulator will represent the pointer machine at each time step. All that remains is to show how it will simulate the transition performed by the pointer machine at each time step. The Kolmogorov machine will not be able to perform the simulation in a single time step, however, because the balanced binary trees that separate the nodes will not be of a constant size. Thus, the simulator will perform each stage of the pointer machine transition by using many time steps (each stage of the transition will take polylogarithmic time).

"Pulling Stage." The simulator will start the pulling stage by first having each node create a *separate* balanced binary tree for all incoming paths of length 2 in the original pointer machine. There are more distinct possible paths of length 2 than length 1 (as before), so this tree will be twice as deep. Once this is done, all nodes will have edges leading to these special balanced binary trees of their *grandchildren.*

Node Creation. The only difference here is that for any newly-created nodes, it is necessary to also create the balanced binary trees for handling incoming edges.

Recoloring. The node can now recolor each outgoing edge based on the colors of all of its outgoing edges (pre-existing and composite). However, recoloring could now take an amount of time that is comparable to the size of the balanced binary trees, as any recoloring may require traversal of the balanced binary trees.

The definition of a pointer machine allows a node to know that two of its outgoing non-composite edges actually point to the same node (as this is possible in the normal pointer machine definition). This can be simulated by providing every node in the graph a unique identifier. Since we can assume all nodes have edges to the root, this identifier could be exactly the path of colored edges the node would take through the root's balanced binary tree. The root's balanced binary tree is at most depth $\log n$ (if n is the number of nodes in the graph), so each node could obtain its own identity in the form of a small chain of $\log n$ nodes. Each node could then check the identifier chains of any two of its children to see if they are in fact the same node.

Drop. Once the edges are recolored, the edges with composite colors must be dropped. Then, the unused paths in the balanced binary trees (both the original and the ones that are twice as deep) must be dropped, as well. Once this is done, it is necessary to merge the two trees of incoming edges that each node has, and then rearrange them into a balanced state. This is possible because only edges with non-composite colors are left over, so the number of outgoing edges on the entire graph is still subject to the same bounds.

4.2.3 Analysis of Simulation

One of the important things to understand about the simulation is its complexity. For example, we might ask how many additional nodes the simulator will need to introduce in order to create the balanced binary trees for all each of the pointer machine nodes being modelled. In other words, we are asking what the difference in size (in terms of the number of nodes) will be between the pointer machine graph and the Kolmogorov machine graph at each time step.

In order to determine this, we need to observe two facts. First, how large must the balanced binary tree be for an individual node be if there are k incoming edges? The tree must have a leaf for every edge, and we know that the number of nodes in a balanced binary tree with k leaves is always less than $2k$. We know this because each level of a binary tree decreases by a factor of 2, so the number of nodes in the tree is

$$k + \frac{k}{2} + \frac{k}{4} + \frac{k}{8} + \ldots + 1 = k + k \cdot \sum_{n=1}^{\log k} \frac{1}{2^n} \leq k + k \cdot 1 \leq 2k.$$

Thus, we know that if there are k incoming edges for a particular node, the number of additional nodes that the simulator will need will be at most $2k$. Next, we determine how many nodes will be needed for all the trees together. Because the number of outgoing edges for each node in a pointer machine is bounded to some constant $|C|$ (where C is the set of colors for the machine), we know that the total number of outgoing edges in a pointer machine with n nodes is at most

$$|C| \cdot n.$$

In other words, if V is the set of nodes in the graph and $out(v)$ is the number of outgoing edges from a node v, then

$$|C| \cdot n \geq \sum_{v \in V} out(v).$$

But all edges have a source node and a destination node. Thus, if $in(v)$ is the number of incoming edges to a node, and we must also have

$$|C| \cdot n \geq \sum_{v \in V} in(v).$$

We know that for each node, the number of tree leaves that will be needed is $in(v)$, so the number of nodes needed for the tree will be at most $2 \cdot in(v)$, so

$$2 \cdot |C| \cdot n \geq \sum_{v \in V} 2 \cdot in(v).$$

Thus, the number of tree nodes needed will be within a constant factor (in particular, $2 \cdot |C|$) of the size of the graph.

5 Computability

We switch our attention to functions. There are many more functions than machines, and later, we will see that machines are just special kinds of functions (computable functions).

For convenience, we make the assumption that all functions have the same domain, $\{0, 1, \odot\}^*$, and have the output range $\{0, 1\}$. The special symbol $\odot$ (called the *mark*) will be used to separate bit strings into components. If x and y are in $\{0, 1, \odot\}^*$, then $x \odot y$ is a string of length $|x| + 1 + |y|$. We define two simple functions on strings, Prefix_k and Suffix_k, where for a string x, $\text{Prefix}_k(x)$ is the prefix of the string up to the kth occurrence of $\odot$, and $\text{Suffix}_k(x)$ is the suffix of the string after the kth occurrence of $\odot$.

In order to more succinctly discuss functions that are undefined, we introduce a more general notion of "equality."

Definition 5.1. For any two functions f, g, we say that $f(x) \doteq g(x)$ if either $f(x) = g(x)$, or $f(x)$ and $g(x)$ are both undefined. [7]

5.1 Classes of Functions

We define several classes of functions. The first two are based on whether a function is defined on all inputs.

Definition 5.2. For any function f, f is *total* if for all $x \in \{0, 1, \odot\}^*$, $f(x)$ is defined. If f is not total, then it is *partial.*

Furthermore, we define the class of functions we have been studying.

Definition 5.3. For any function f, f is *computable* if there exists a Turing machine m such that for all $x \in \{0, 1, \odot\}^*$, $f(x) \mathrel{\mathring{=}} m(x)$.

Exercise 5.4. Find examples of the following kinds of functions:

- A total function that is not computable?
- A partial function that is not computable?
- A total computable function?
- A partial computable function?

5.2 Simulation in General

We can now introduce the notions of simulation and intersection. These are similar to the notion of simulation used when dealing with universal machines, but are more general because they apply to functions.

Definition 5.5. For any natural number k, for any function u, for any function f, u k-simulates f if there exists p such that for all $s \in \{0, 1, \odot\}^*$, $u(p \odot s) \doteq f(s)$, where $\text{Prefix}_k(p \odot s) = p\odot$ and $\text{Suffix}_k(p \odot s) = s$.[8]

Intuitively, in the case of machines, p could be viewed as representing the program that describes a machine.

Definition 5.6. For any natural number k, for any function u, for any function f, u k-intersects f if there exists p such that for all $s \in \{0, 1, \odot\}^*$, $u(p \odot s) \doteq f(p \odot s)$, where $\text{Prefix}_k(p \odot s) = p\odot$ and $\text{Suffix}_k(p \odot s) = s$.

[7] A machine diverges on an input iff the function it computes is undefined on that input.

[8] This merely means that p contains $k - 1$ instances of $\odot$.

We now introduce definitions for isolating special functions for a set of functions. Once again, these are generalizations of the universal machine for an arbitrary collection of functions.

Definition 5.7. For any collection of functions F, for any u, u is *universal* for F if there exists k such that for any $f \in F$, u k-simulates f.

Definition 5.8. For any collection of functions F, for any u, u is *complete* for F if there exists k such that for any $f \in F$, u k-intersects f.

5.3 Applying the Definitions

We can practice applying these definitions by introducing two operators on functions, $+$ and $-$.

Definition 5.9. For any k, for any function f, for any $x \in \{0, 1, \odot\}$, we define f^+ and f^- in the following way:

$$f^+(x) = f(\text{Prefix}_k(x) \odot x) \text{ and } f^-(x) = f(\text{Suffix}_k(x)).$$

Now, we can prove some theorems about collections of functions closed under these operators.

Theorem 5.10. *Suppose that there exists a collection of functions F such that for every $f \in F$, $f^+ \in F$ (in other words, F is closed under $+$). Suppose also that there exists $u \in F$ such that u is universal for F. Then u is complete for F.*

Proof. We begin by recognizing that we can assume that there exists $u \in F$ that is universal for F. By expanding the definition for universality, we see that this implies that

$$\forall f \in F, \exists p \text{ such that for all } s, u(p \odot s) = f(s).$$

Thus, we can assume this fact. We want to show that u is complete for F. Take any $f \in F$. We know that $f^+ \in F$, and since u is universal, there exists p such that

$$\forall s, u(p \odot s) = f^+(s).$$

In the above statement, we can choose any s. Given any s', let us choose $s = p \odot s'$. Then, we have

$$\forall s', u(p \odot (p \odot s')) = f^+(p \odot s').$$

But by the definition of f^+ and the associativity of concatenation, this means

$$\forall s', u((p \odot p) \odot s') = f((p \odot p) \odot s').$$

But this means that for any $f \in F$, there exists a prefix $p \odot p$ such that u $2k$-simulates f.[9] Thus, u satisfies the definition for completeness. □

Exercise 5.11. Suppose that there exists a collection of functions F such that for every $f \in F$, $f^- \in F$ (in other words, F is closed under $-$). Suppose also that there exists $u \in F$ such that u is complete for F. Prove that u is universal for F.

[9] Notice that we use $2k$ because $p \odot p \odot$ contains twice as many mark symbols as p.

5.4 Negation

We define a version of negation that works for both partial and total functions. This definition of negation is closely related to universality and completeness.

Definition 5.12. For any function f (partial or total), on any input $x \in \{0, 1, \odot\}$, $\neg f$ is a total function defined as

$$\neg f(x) = \begin{cases} 1 & \text{if } f(x) = 0 \\ 0 & \text{otherwise} \end{cases}$$

Theorem 5.13. *For any class of functions F, if for all $f \in F$, $\neg f \in F$ (in other words, F is closed under negation), then there does not exist a function inside F that is complete for F.*

Proof. We will prove this by contradiction. Suppose there exists $u \in F$ such that u is complete for F. Since F is closed under negation, $\neg u \in F$. Since u is complete, there exist k and p such that for all s,

$$u(p \odot s) = \neg u(p \odot s).$$

But by definition of negation, for all $x \in \{0, 1, \odot\}^*$, $u(x) \neq \neg u(x)$, a contradiction. Thus, no such $u \in F$ can exist. □

We can now use this theorem to easily determine whether certain classes of functions contain a universal or complete function.

Exercise 5.14. For each of the following collections of functions, determine whether or not they contain a function that is complete for the collection.

- The set of all functions?
- The set of all total functions?
- The set of all computable functions?
- The set of all total computable functions?

Exercise 5.15. Let u be the universal Turing machine. We know that u is computable. Is $\neg\neg u$ computable?

Exercise 5.16. Find an explicit function u that is complete for the set of linear functions $\{f(x) = cx \mid c \in \mathbb{N}\}$.

Note that it is possible to define such an explicit u, and you are expected to do so. In particular, you should define an equation

$$u(z) = \ldots$$

where the body of the function denoted by $\ldots$ above is an explicit formula that can contain the following:

- arithmetic operations (+,-,·,/,log, pow,mod) and summations/products ($\sum_{i=0}^{n} f(i)$, $\prod_{i=0}^{n} f(i)$);
- string operations (Prefix$_k$, Suffix$_k$, string concatenation, string length);[10]
- constants and variables.

You are obviously allowed to make your solution modular by defining helper functions, as long as the helper functions you employ also respect these restrictions.

Remember that from the perspective of f, the inputs to f *cannot contain an explicitly discernible "mark" character*. Thus, s can be *any* string/integer, and the functions f will always treat, for example, $p \odot s$ as nothing more than an integer in some representation (e.g. binary, with $\{0,1\}^*$, or ternary (base three), with $\{0,1,\odot\}^*$). Any mark symbols are treated just like the digits of the integer. However, your definition of u is allowed to treat mark symbols in a different manner. But be careful: if u interprets strings or parts of strings in a manner different from f, u must perform the appropriate conversions explicitly (no hand-waving).

You get to decide how the prefix p is explicitly constructed given a certain f, and you also get to decide how u manipulates that p in its body, but that p must be in the same space as the input to f. Thus, for example,

$$p \in \{0,1\}^*, \ s \in \{0,1\}^*, \ f \in \{0,1\}^* \to \{0,1\}^*, \ u \in \{0,1\}^* \to \{0,1\}^*.$$

You must respect these restrictions in order to get full credit.

Once you provide an explicit equation defining u, you must use algebraic reasoning to show that u indeed satisfies the definition for completeness. That is, show that for any f, you can construct an explicit p such that for any integer s, $u(p \odot s) = f(p \odot s)$.

Other hints: Here are a few other details and comments:

- the function u must be independent of all of the quantified variables in the definition of completeness; it can only depend on its input; if you want u to be able to see one of these variables, you'll need to encode it somehow in the prefix you construct, and u will need to decode it;
- the prefix p is allowed to depend on the function f, so take advantage of this fact;
- you may assume that all integers in this problem are positive.

[10]These can obviously be simulated using arithmetic operations.

Example 5.17. Consider the set consisting of two boolean functions:

$$B = \{and, or\}.$$

Each function takes a string of bits, and returns a single bit that is either the conjunction of all the bits, or the disjunction of all the bits. Thus, for example,

$$and(1111111) = 1, \quad and(11001) = 0.$$

Thus, $and \in \{0,1\}^* \to \{0,1\}$ and $or \in \{0,1\}^* \to \{0,1\}$.

Suppose we want to find a function u that is complete for the class B. Thus, we need to define a single u such that for any $f \in B$ (f can either be *and* or *or* in this case), there exists p such that on p concatenated with any s, u and f will have the same output.

First, our domain has no "mark" character, so we must define one. In our case, we let "mark" be represented by 0. This allows us to use the Prefix_1 function. For example,

$$\text{Prefix}_1(11101010100) = 111.$$

Thus, we are now able to represent an arbitrary integer using the *length* of a prefix. For example, suppose we want to represent the integer 5 in a prefix for any s. If $p = 11111$ and $s = 11010101010$, then the concatenation of p and s is

$$\underbrace{11111}_{p}0\underbrace{11010101010}_{s}.$$

Now, it's possible to extract the value 5 from the prefix simply by computing

$$\lceil \log(\text{Prefix}_1(11111011010101010)) \rceil = \lceil \log(11111) \rceil = 5.$$

For our problem, we can choose two special prefixes: $p_{or} = 111$ and $p_{and} = 11$. We can also define the following explicit helper function:

$$\ell(z) = \lceil \log(\text{Prefix}_1(z)) \rceil - 2.$$

You should be able to convince yourself that for any s,

$$\ell(p_{or}0s) = \ell(1110s) = 1, \quad \text{and } \ell(p_{and}0s) = 0.$$

Thus, we have successfully used the prefix to encode information about which function u should simulate. We can now define u in the following explicit manner:

$$u(z) = \ell(z) \cdot or(z) + (1 - \ell(z)) \cdot and(z).$$

Thus, for *and*, there exists the prefix p_{and} such that for any s,

$$\begin{aligned}
u(p_{\text{and}}0s) &= \ell(p_{and}0s) \cdot or(p_{and}0s) + (1 - \ell(p_{and}0s)) \cdot and(p_{and}0s) \\
&= 0 \cdot or(p_{and}0s) + (1 - 0) \cdot and(p_{and}0s) \\
&= 0 + and(p_{and}0s) \\
&= and(p_{and}0s).
\end{aligned}$$

The proof for the *or* case is analogous. Thus, u is complete for B.

5.5 Properties of Programs

The $\overset{\circ}{=}$ relation gives us a notion of equivalence on individual outputs of programs. We can extend this to a definition of equivalence over entire programs (or, in this case, program descriptions).

Definition 5.18. Let u be the universal Turing machine. For any two program descriptions $p, q \in \{0, 1, \odot\}$, we say that

$$p \sim q,$$

(that is, p and q are equivalent) if for all $s \in \{0, 1, \odot\}$,

$$u(p \odot s) \overset{\circ}{=} u(q \odot s).$$

We can now state Kleene's fixed point theorem about programs.

Theorem 5.19. *Let $\mathcal{M}$ be the space of all programs. For any total computable function $A \in \mathcal{M} \to \mathcal{M}$, there exists some $p \in \mathcal{M}$ such that $A(p) \sim p$.*

Note that because all programs can be represented as a bit string, $\mathcal{M} = \{0, 1, \odot\}$. Intuitively, this theorem states that whenever we define a program that transforms other programs but never diverges, such a program will always act like the identity function on at least one program.

Now, we can consider a specific kind of computable function on programs: a computable predicate. A computable predicate on programs is just a function

$$T \in \mathcal{M} \to \{0, 1\}$$

that corresponds to some property of programs.

In other words, it's a function that always returns 1 if the property of interest holds, and 0 otherwise. One example of such a predicate is T_{length}, which corresponds to a property related to a program's description length. It is defined as follows:

$$T_{\text{length}}(p) = \begin{cases} 0 & \text{if the length of } p \text{ is over 10} \\ 1 & \text{otherwise} \end{cases}$$

We can ask about an interesting property of such a predicate: is it the same on equivalent programs? Obviously, there are many cases in which such a property of a predicate is valuable: if two programs behave exactly the same way on all inputs, we could want a predicate that will treat these two programs as if they were the same.

Definition 5.20. For any $T \in \mathcal{M} \to \{0, 1\}$, we say that T is *robust* if for all $p, q \in \mathcal{M}$,

$$p \sim q \text{ implies } T(p) = T(q).$$

Another way to view robust predicates is to divide the space of all programs $\mathcal{M}$ into equivalence classes under $\sim$, so that all programs that have equivalent behavior are treated as a single object. Call this space $\mathcal{M}/\sim$. In this scenario, robust predicates are just predicates that can be expressed as predicates over equivalence classes of programs, of the form $T \in \mathcal{M}/\sim \to \{0, 1\}$.

Definition 5.21. For any $T \in \mathcal{M} \to \{0, 1\}$, we say that T is *non-constant* if there exist $p, q \in \mathcal{M}$ such that

$$T(p) \neq T(q).$$

We can also say that T is *constant* if for all $p, q \in \mathcal{M}$,

$$T(p) = T(q).$$

We can now apply these definitions and theorems in the following exercise.

Exercise 5.22. (Homework #6) Prove that for any computable, *robust* predicate $T \in \mathcal{M} \to \{0,1\}$, for all $p, q \in \mathcal{M}$,

$$T(p) = T(q).$$

In other words, show that a predicate T cannot be all of the following simultaneously: (1) computable, (2) robust, and (3) non-constant.

Notice that the statement is *independent* of whether or not p and q are equivalent! That means the proof must work regardless of whether or not $p \sim q$. Also notice that T is computable, so it is also a program.

5.6 Recursion

We now introduce the notion of recursion. First, we specify a special kind of infinite set: one whose elements can be listed simply by counting up.[11]

Definition 5.23. Any infinite set Y is *recursively enumerable* if there exists a total computable function $I : \mathbb{N} \to Y$ such that for all $y \in Y$, there exists $n \in \mathbb{N}$ such that $I(n) = y$.

It should be clear that $\mathbb{N}$ and $\mathbb{Z}$ are recursively enumerable. It turns out that the set of inputs on which a Turing machine converges is also recursively enumerable. In order to show this, we must construct a total computable function $I : \mathbb{N} \to \{0,1\}^*$.

5.6.1 More on Convergence

It is worth examining exactly what it means for an algorithm to converge on an input.

Remark 5.24. *Let u be the universal Turing machine. For an input z, we say that $u(z)$ converges if* for that particular z, *there exists a natural number $k \in \mathbb{N}$ such that $u(z)$ will halt and produce an output in k time steps.*

Notice that all that is required for u to converge on an input is that there exist some finite number of steps after which it will converge. Also, notice that this does not contradict our previous issues with detecting divergence: if we do not know ahead of time whether a k exists for some input z, we would still be unable to differentiate divergence from convergence at some distant time step in the future.

Another way to view this is to observe that humans can detect that certain kinds of programs converge on certain inputs. This ability is based on the analysis of the particular algorithm in question. Thus, for a particular algorithm, and a particular input z, it is possible that there exists a formal argument that $u(z)$ will converge.

Because we used u in our definition, it is trivial to extend this definition to any Turing machine: instead of z, we consider any $p \in \mathcal{M}$ and the behavior of $u(p \odot x)$.

5.6.2 Predicates Regarding Halting

We define the halting and bounded halting predicates.

[11] We focus on infinite sets, as finite sets can be handled much more trivially, as we describe further below.

Definition 5.25. For any Turing machine $p \in \mathcal{M}$, and any input x,

$$H(p,x) = \begin{cases} 1 & \text{if } u(p \odot x) \text{ converges} \\ 0 & \text{otherwise} \end{cases}$$

We know that H would allow us to make a total extension of the universal Turing machine, so it cannot be computable.

Definition 5.26. For any Turing machine $p \in \mathcal{M}$, and any input x, and any $t \in \mathbb{N}$,

$$B(p,x,t) = \begin{cases} 1 & \text{if } u(p \odot x) \text{ finishes in } t \text{ time steps} \\ 0 & \text{otherwise} \end{cases}$$

The predicate B is total and computable, because it will always terminate in t time steps, and it is easy to check if a Turing machine has finished (simply count the number of active heads).

5.6.3 Enumerating Converging Inputs

We can now describe a process that enumerates all of the inputs on which a particular Turing machine converges:

```
build a set $E_p$:
    for $n$ from 0 to $\infty$:
        for all inputs $x$ and $t$ s.t. $|x| < n$ and $t < n$:
            if $B(p,x,t) = 1$ then add $x$ to the set (avoiding duplicates)
            otherwise, continue
```

There is an E_p for every program $p \in \mathcal{M}$. If E_p treated as a list, then we can say $E_p(i)$ is the ith element of this list. We can then view $E_p(i)$ as a computation that either diverges (if $i > |E_p|$) or converges to a particular input.

For some programs (e.g. those that always diverge), the set is empty, so E_p will always diverge. However, note that as long as there are at least i elements in E_p, $E_p(i)$ will converge according to our above definition for convergence. We assumed that the sets we consider are infinite, so $E_p(i)$ will always converge. We specify how to handle finite sets further below.

Next, recall that the space of all inputs could be represented as $\{0,1\}^*$. Thus, for any E_p, we could also consider the complement of E_p, which is the set of all inputs on which the program diverges:

$$\overline{E}_p = \{0,1\}^* - E_p.$$

Definition 5.27. For any set $Y \subseteq \{0,1\}^*$, Y is *decidable* (a.k.a. computable, a.k.a. recursive) if both Y and $\overline{Y} = \{0,1\}^* - Y$ are recursively enumerable.

In particular, a decidable set must necessarily have a total computable function that corresponds to it (returning 1 for elements within the decidable set, and 0 on elements outside of it). We can use this idea to show that the halting problem is not decidable.

Exercise 5.28. Is $\overline{E}_p$ recursively enumerable for any p? No, because if it is, then we can construct an algorithm for the halting predicate:

```
$H(p, x) =$
    for $i$ from 0 to $\infty$:
        if $E_p(i) = x$ then 1
        if $\overline{E}_p(i) = x$ then 0
```

5.6.4 Finite Sets

Notice that if $Y = \emptyset$, or Y is any set of finite size, then Y is trivially decidable: we simply build a Turing machine that encodes the entire set inside inside its transition function. This machine can then check any input against this encoded set. The function this Turing machine represents is total computable, so such a set Y is trivially decidable.

5.7 Gödel's Theorem

Suppose we have an alphabet of symbols, such as

$$\Sigma = \{x, y, \Rightarrow, \neg\}.$$

We can then define a grammar over the set Σ^*. For example, the following might be a grammar (using BNF notation):

$$G ::= \text{true} \mid \text{false} \mid G \Rightarrow G \mid G \wedge G \mid \neg G.$$

A grammatical string in this system might be

$$((\text{true} \Rightarrow \text{true}) \wedge \text{true}) \Rightarrow \text{true}.$$

Let $G \subseteq \Sigma^*$ be the set of grammatically well-formed strings.

Finally, we can identify a set of axioms Φ that corresponds to grammatical strings that represent something that is "true." For example, the set of axioms might be

$$\Phi = \{\text{true}, \neg\text{false}\}.$$

We could interpret strings as both statements, and proofs of those statements. For example, the statement $s = \text{true}$ is trivially true, and its proof is simply $P = \text{true}$, an axiom. We can now define a formal system and some properties that a formal system might have.

Definition 5.29. We say that A is a *formal system* if A is a computable predicate in $\Sigma^* \times \Sigma^* \Rightarrow \{0, 1\}$. In effect, for any statement $s \in G$, and any proof $P \in G$, we have

$$A(s, P) = 1$$

if P is a proof of the statement s. In any other case, A can diverge or output something arbitrary (unless further restrictions are placed upon it independently).

Definition 5.30. We say that A is a *rich* system if for every possible $x \in \{0, 1\}^*$, there is an $s_x \in G$ that can be computed from x such that

$$\begin{aligned} \exists P \text{ s.t. } A(P, s_x) &= 1 \text{ if } u(x) = 1 \\ \exists P \text{ s.t. } A(P, \neg s_x) &= 1 \text{ if } u(x) = 0 \end{aligned}$$

Definition 5.31. For a system A, consider the following two possibilities:

- there exists P such that $A(P, s_x) = 1$;
- there either exists P such that $A(P, \neg s_x) = 1$.

We say A is *consistent* if at most one of these holds. We say that A is *complete* if at least one of these holds.

We can now state and prove the theorem.

Theorem 5.32. *There exists no system A which is formal, rich, consistent, and complete.*

Proof. Assume A is formal, rich, consistent, and complete. Let u be the universal Turing machine. Consider the following function $\tilde{u}$:

```
$\tilde{u}(x)$ = for all $P \in G$ (enumerated by increasing size):
    if $A(P, s_x) = 1$ then return 1;
    if $A(P, \neg s_x) = 1$ then 0.
```

Because A is rich, there is always an appropriate computable s_x for every x. Because A is consistent, exactly one of $A(P, s_x)$ or $A(P, \neg s_x)$ will converge. Thus, $\tilde{u}$ will behave like u on those inputs for which u converges. Because A is complete, at least one of these will converge, so $\tilde{u}$ is total and computable. Thus, $\tilde{u}$ is a total extension of u. This means that $\tilde{u}$ is complete for the class of total computable functions, and is also within that class. Because the class of total computable functions is closed under negation, this is impossible, so we have a contradiction. Thus, no such A could exist. □

5.8 More Decidability Examples: Decidability and Concatenation

5.8.1 Counterexample

For any two sets $X, Y \subset \{0,1\}^*$, define the *concatenation* of X and Y to be

$$XY = \{xy \mid x \in X, y \in Y\}.$$

Suppose we want to construct a set A such that A is undecidable, but AA is decidable. Consider the following function:

$$a(x) = \begin{cases} 1x1 & \text{if } |x| \text{ is odd and } u(x) \text{ halts} \\ 1x0 & \text{if } |x| \text{ is odd and } u(x) \text{ diverges} \\ 0x11 & \text{if } |x| \text{ is even and } u(x) \text{ halts} \\ 0x00 & \text{if } |x| \text{ is even and } u(x) \text{ diverges} \end{cases}$$

The first bit of the output represents whether the input string x is of odd or even length, and the last one or two bits represent whether the universal Turing machine halts on x.

Let us define the set

$$A = \{a(x) \mid x \in \{0,1\}^*\} \cup \{x \mid x \in \{0,1\}^* \text{ and } |x| \text{ is even }\} \cup \{0, 1\}.$$

Essentially, all the strings in A that are of *odd* length and longer than 1 bit represent solutions to the halting problem. All other strings are either 0, 1, or of even length.

First, we show that AA is decidable. Note that because A contains all strings of even length as well as the strings 1 and 0, we have that $AA = \{0,1\}^*$, because each element in $\{0,1\}^*$ is either of even length, or made up of a string of even length with a 0 or 1 concatenated onto one of its ends. The set $\{0,1\}^*$ is trivially decidable.

Now, let us show that A is undecidable. Suppose we have a total computable function f that decides whether an element $x \in \{0,1\}^*$ is in A (in other words, $f(x) = 1$ iff $x \in A$). Then, we can construct a total computable function that is equivalent to the halting predicate by passing an appropriate string to the function f:

$$H(p,x) = \begin{cases} 1 & \text{if } |px| \text{ is odd and } f(1px1) = 1 \\ 0 & \text{if } |px| \text{ is odd and } f(1px0) = 1 \\ 1 & \text{if } |px| \text{ is even and } f(0px11) = 1 \\ 0 & \text{if } |px| \text{ is even and } f(0px00) = 1 \end{cases}$$

The first bit of the input to f ensures that there is no confusion about whether the output will represent a solution to the halting problem for a string of even length, or a string of odd length. Because f is total and computable, so is this implementation of H. This is a contradiction because we know that $H(p,x)$ is not total and computable, so A cannot be decidable.

Exercise 5.33. We could simplify the above definitions by removing two cases from our definition of $a(x)$ while still keeping A undecidable and AA decidable. Perform this simplification.

5.8.2 Enumeration of Complement

Assume that A is decidable. Suppose we want to show that AA is decidable by using enumeration (we saw in section the strategy that checks all possible splits of a string using the total computable function that decides A, and we know this works). Since A is decidable, we have an enumerator E_A for A, so we can construct an enumerator for AA:

```
$E_{AA} =$
    for $n$ from 0 to $\infty$:
        for all $i, j < n$:
            add $E_A(i)E_A(j)$ to the list
```

However, for AA to be decidable, we would still need an enumerator for $\overline{AA}$. Unfortunately, we know that $\overline{A}\ \overline{A} \neq \overline{AA}$, and in fact, an enumerator for $E_{\overline{A}}$ is of no use to us. In particular, suppose

$$A = \{111, 10\}.$$

Notice that $11 \in \overline{A}$ and $110 \in \overline{A}$, which means $11110 \in \overline{A}\ \overline{A}$. But we also have $11110 \in AA$, so it cannot be that $11110 \in \overline{AA}$. Thus, we need to be more careful. If we still want to enumerate $\overline{AA}$, the easiest approach is to use the total computable function f_A that decides A and employ the aforementioned splitting approach:

```
$E_{\overline{AA}} =$ for $n$ from 0 to $\infty$:
    for all $x \in \{0,1\}^*$ s.t. $|x| < n$:
        for all possible splits $x'x''$ of $x$:
            if $f_A(x') = 1$ and $f_A(x'') = 1$ then break out and try next $x$
            otherwise, continue
        if no split caused the loop to end, add $x$ to the list
```

5.9 Kolmogorov Complexity

When dealing with string compression, one natural question to ask is how much information is contained within a string, or how complex it is. [12] One way to measure a string's complexity is to take the length of the *shortest* possible compressed version of this string. But what is a compressed version of a string? Is 1 a compressed version of 11111? Intuitively, in order for a string s^* to be a compressed version of s, it should be possible to recover s with s^* alone, and no additional information that is *unique to that string.* However, it is still fair to establish some sort of standard that applies to *all* strings, as then, no individual string is getting any kind of special treatment.

Thus, we can use the universal multi-headed Turing machine to establish this standard: fix some input $x \in \{0,1\}^*$; the shortest representation for a string i is the shortest $p \in \mathcal{M}$ (in terms of its description length) such that $u(p \odot x)$ halts and outputs i.

Definition 5.34. Let B_V be the bounded volume predicate (this is a generalization of the bounded halting predicate we defined earlier):

$$B_V(p,x,t) = \begin{cases} 1 & \text{if } u(p \odot x) \text{ finishes in volume } t \\ 0 & \text{otherwise} \end{cases}$$

We can now define the function for Kolmogorov complexity:

$$K_t(i/x) = |p| \text{ for the shortest } p \in \mathcal{M} \text{ s.t. } B_V(p,x,t) = 1 \text{ and } u(p \odot x) = i.$$

Exercise 5.35. Prove that for any string $i \in \{0,1\}^*$, there exists $c \in \mathbb{N}$ and $t \in \mathbb{N}$ such that

$$K_t(i/x) \leq |i| + c \text{ and } t \leq |i|.$$

Exercise 5.36. Prove that for any string $i \in \{0,1\}^*$, for any fixed inputs $x, x' \in \{0,1\}^*$, there exists $c \in \mathbb{Z}$ such that for any $t \in \mathbb{N}$, if $K_t(i/x)$ and $K_t(i/x')$ are defined, then

$$K_t(i/x) \leq K_t(i/x') + c.$$

Definition 5.37. In Exercise 5.35 you showed that for any $i \in \{0,1\}^*$, there exists $c \in \mathbb{N}$ and $t \in \mathbb{N}$ such that $K_t(i/x) \leq |i| + c$ and $t \leq |i|$. Given these c and t, a string $i \in \{0,1\}^*$ is *incompressible* if for all $x \in \{0,1\}^*$,

$$K_t(i/x) \geq |i| + c \text{ and } t \geq |i|.$$

Exercise 5.38. Prove that there exists $i \in \{0,1\}^*$ such that i is incompressible.

5.10 Conversions Involving Space and Time Efficiency

5.10.1 From Small Space to Small Time

Suppose we have a machine $m \in \mathcal{M}$ and $c \in \mathbb{R}$ such that for all $x \in \{0,1\}^*$, $m(x)$ halts in time $O(c^{|x|})$ but never uses more than $O(|x|)$ space on its tape. There exists an equivalent machine m' that uses $O(c^{|x|})$ space but only $O(|x|)$ time.

First, let us consider all possible states of the machine m. The state of a machine is the size of the working tape, and this is $O(|x|)$. Since each position in the tape can have one of a number of constant symbols (the size of the tape alphabet is constant), this implies that the number of possible states is $O(c^{|x|})$.

[12] "Information" in this context refers primarily to complexity or entropy that is analogous to the same concepts in the physical world, and not to the usual notion of semantic content that relies heavily on an accompanying context and interpretation.

Suppose we have a graph that contains all $O(c^{|x|})$ possible states as nodes, and $O(c^{2|x|})$ edges representing legal transitions between nodes. If we treat this graph as a pointer machine m', m' can compute the same function as m in time $O(|x|)$. It accomplishes this by having every node perform the following procedure in parallel at each time step:

- obtain edges to all its grandchildren;
- remove all edges to its children.

Notice that after k iterations of the above algorithm, every node has an edge to all nodes

5.10.2 From Small Time to Small Space

6 Non-determinism

6.1 Non-deterministic Turing Machines

It is possible to generalize our model of the single-head deterministic Turing machine by allowing it the option of duplicating its entire state (including head position) and tape at *every* time step during its operation, at *no cost.* Obviously, if a Turing machine is allowed only to duplicate its state and tape exactly, no additional power is introduced. Thus, we allow duplication only of the the two copies make *different* transitions at that time step.

Another way to model this power is to assume that a deterministic Turing machine is given a string that "drives" it: at every time step, the machine can consult a bit (or symbol) in this string when making its transition decision. Then, we quantify the machine's behavior over all possible strings.

Definition 6.1. For any Turing machine $m \in \mathcal{M}$, let the *language* L_m for m be the set

$$L_m = \{x \in \{0,1\} \mid m(x) = 1\}.$$

6.2 P and NP

Definition 6.2. For a deterministic Turing machine $m \in \mathcal{M}$, we say that m represents a *polynomial-time* algorithm if there exists some constant $k \in \mathbb{R}$ such that for all $x \in \{0,1\}^*$, $m(x)$ terminates in less than $|x|^k$ time steps.

Definition 6.3. We say that a language L is in P if there exists a polynomial-time Turing machine m such that L is the language for m. In other words, if $\mathcal{M}_{\text{poly}}$ is the set of all polynomial-time deterministic Turing machines, then

$$P = \{L_m \mid m \in \mathcal{M}_{\text{poly}}\}$$

Exercise 6.4. Suppose that m is a deterministic Turing machine and there exist $c, k \in \mathbb{R}$ such that for all $x \in \{0,1\}^*$, $m(x)$ is computed in time $c^{|x|^k}$. Does there *necessarily* exist a polynomial-time non-deterministic Turing machine m' such that for all $x \in \{0,1\}^*$, $m(x) = m'(x)$ and $m'(x)$ is computed in time $|x|^k$? If so, prove this. If not, provide a sketch of a counterexample.

Theorem 6.5. *For a non-deterministic Turing machine $m \in \mathcal{M}$ and the language L for m, if $L \in NP$ then there exists some constant $c \in \mathbb{R}$ and a* deterministic *Turing machine m' such that for all $x \in \{0,1\}^*$, $m'(x) = m(x)$ and $m'(x)$ terminates in less than $O(c^{|x|})$ time steps.*

Question 6.6. Suppose both a deterministic Turing machine m and a non-deterministic Turing machine m' are both limited to using a fixed length of tape ℓ. Can m compute all functions that m' can compute?

6.3 Polynomial-time Reductions

Definition 6.7. An algorithm f is *polynomial-time reducible* to an algorithm g if there exists a polynomial-time algorithm r, called the *reduction*, such that for all $x \in \{0,1\}^*$, $f(x) = g(r(x))$.

$$\begin{tikzcd}
x \arrow[r, "r"] \arrow[d, "f"] & r(x) \arrow[dl, "g"] \\
f(x) &
\end{tikzcd}$$

Sometimes, it is convenient to split the reduction into two parts, r and r', such that $f(x) = r'(g(r(x)))$:

$$\begin{tikzcd}
x \arrow[r, "r"] \arrow[d, "f"] & r(x) \arrow[d, "g"] \\
f(x) & g(r(x)) \arrow[l, "r'"]
\end{tikzcd}$$

6.4 Relationship between Verification, Inversion, and Search Problems

We illustrate the correspondence between search, inversion, and decision problems using an example. Consider the problem of finding a Hamiltonian path p given a graph G. Suppose we have a verification problem v of the following form: given a graph G and a path p, decide whether the path p is a Hamiltonian path in G:

$$v : \{(G,p) \mid \ G \text{ is a graph and } p \text{ is a path in } G\ \} \to \{0,1\}.$$

Clearly, there exists a polynomial-time algorithm computing v. Suppose we modify this function slightly (without changing its complexity) by having it return G as well:

$$w : \{(G,p) \mid \ G \text{ is a graph and } p \text{ is a path in } G\ \} \to \{(0,G),(1,G)\}.$$

Clearly, w is polynomial, since $w(G,p) = (v(G,p),G)$. Now, suppose we consider the inverse of w, w^{-1}:

$$w^{-1} : \{(0,G),(1,G)\} \to \{(G,p) \mid \ G \text{ is a graph and } p \text{ is a path in } G\ \}.$$

In this case, w^{-1} is the search problem corresponding to the verification problem v. In this example, the problem of finding a Hamiltonian path for an input graph is NP-complete.

6.5 NP-Completeness

Definition 6.8. A language L_m for a machine m is NP*-hard* if for all m' such that $L_{m'} \in NP$, there exists a polynomial-time reduction r such that for all $x \in \{0,1\}^*$,

$$m'(x) = m(r(x))$$

$$\begin{tikzcd}
\mathrm{m}' \arrow[dr, "r"'] & \cdots & \mathrm{m}'' \arrow[dl, "r'"] \\
& m &
\end{tikzcd}$$

We can diagram how one would prove that an arbitrary m represents an NP-hard language: by demonstrating a polynomial-time r^* such that

$$\begin{tikzcd}
\mathrm{m}' \arrow[dr, "r"'] & \cdots & \mathrm{m}'' \arrow[dl, "r'"] \\
& m \arrow[d, "r^*"] & \\
& m_{NP-\text{hard}} &
\end{tikzcd}$$

Definition 6.9. A language L_m is *NP-complete* if $L_m \in NP$ and L_m is NP-hard.

6.5.1 NP-complete Problems

We introduce two NP-complete problems.

Example 6.10. The *graph coloring* problem is as follows: given an arbitrary graph G and number of distinct colors k, is it possible to color all the vertices in the graph G such that no two vertices connected with an edge have the same color?

The Turing machine corresponding to this problem takes G, k as input, and produces an output in $\{0, 1\}$. The language corresponding to this machine will be a subset of all pairs G, k.

Example 6.11. The *set cover* problem is as follows: given a collection of sets $C_1, \ldots, C_n$ (possibly overlapping) and a value k where $k < n$, find a subcollection of these sets $C_{j_1}, \ldots, C_{j_k}$ such that

$$\bigcup_{i=1}^{n} C_i = \bigcup_{i=1}^{k} C_{j_i}.$$

6.5.2 Register Allocation

Suppose we are given as input a program P written in a language that has two kinds of statements: variable allocation,

`var X`,

and variable hiding/destruction,

`free X`.

A variable `X` is "in scope" at any point after a statement `var X`, and not "in scope" at any time after a `free X` statement. We want to compile this program onto a machine architecture with exactly k registers. At every point in a program, each register can only represent the data in exactly one variable. This means that if at any point in the program more than k variables are in scope, we must allocate memory to store some of the variables.

The problem is stated as follows: given P and k, is it absolutely necessary to store at least one variable's data in memory? We make the simplifying assumption that any two variables declared using the same variable name must be stored in the same register.

Exercise 6.12. Prove that this problem is NP-complete.

6.5.3 Radio Towers

Suppose we have just enough building materials to construct exactly k radio towers, and each radio tower has a range d. Now, suppose we are given an arbitrary collection of islands on a grid. The problem is to decide whether it is possible to position the k towers on the islands in a way that ensures that every island will receive a signal from at least one tower.

Exercise 6.13. Prove that the radio towers problem is NP-complete.[13]

[13]Hint: look up the "vertex cover" problem.