CS 181:
Introduction to Theoretical Computer Science

Einar Balan

Contents

1 | Computation and Representation

Major Idea: What can we compute? What does it mean to compute?

There are two aspects of computing:

1. Data (Representations of Objects)
    - We can represent numbers in many ways i.e. roman numerals or the place-value system we are familiar with today
    - Some representations are easier to work with than others – to convey the distance to the moon in Roman numerals would fill a small book
    - Clearly choosing the right representation can have a *dramatic* effect on computation
2. Algorithms (Operations on Data)
    - In general, there is more than one way to accomplish the same task; some better than others
    - To multiply, we can either perform repeated additions or the typical gradeschool multiplication algorithm
    - Gradeschool multiplication is far more efficient as an $O(n^2)$ algorithm compared the to the exponential nature of repeated addition
    - Choosing the right algorithm is also incredibly important to computation

Takeaway: It is important to utilize both a good data represntation and a good algorithm in all computing tasks.

Representations

- In general we can represent many objects as a sequence of 0's and 1's. Anything from images, text, video, audio, databases, etc. can be encoded in binary.
- BIG IDEA: We can compose representations in order to represent any object i.e. if you can represent objects of type T, then you can also represent lists of that object

Definition Representation, where E is one-to-one

$$E : O \rightarrow \{0,1\}^*$$

Example Represent natural numbers $E : N \rightarrow \{0,1\}^*$

Couple options:

- Unary: E(n) = 0000...0 (n + 1 zeroes)
- Binary: Standard binary encoding defined as follows

$$NtoB(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ NtoB(\lfloor \frac{n}{2} \rfloor) \circ (n\%2) & \text{else} \end{cases} \tag{1.1}$$

- We know an encoding E is valid iff there exists a decoding function s.t. $D : \{0,1\}^* \rightarrow O$ and $D(E(x)) = x$

Example $E : Z \rightarrow \{0,1\}^*$

$$ZtoB(n) = \begin{cases} 0 \circ NtoB(n) & \text{if } n >= 0 \\ 1 \circ NtoB(-n) & \text{else} \end{cases} \tag{1.2}$$

Example Represent rational numbers $E : Q \rightarrow \{0,1\}^*$

- We know we can represent any rational number with a fraction i.e. a pair of numbers
- So if we can find a way to encode two numbers in a one-to-one way, we can represent any rational number
- Suggestions
  1. Unary(p) ∘ 1 Unary(q)
  2. Encode length of numerator
  3. Add padding to shorter number to match lengths
  4. Utilize new $\overline{ZtoB} : Z \rightarrow \{0,1\}^*$ where each bit is duplicated and a 01 is added to the end
     - $QtoS(p/q) = \overline{ZtoB}(p) \circ \overline{ZtoB}(q)$

Big Idea: We can represent objects and lists of objects as compositions of representations.

2 | Prefix Free Encoding & Models of Computation

Nonformally, a prefix free encoding is one that is easy to decode if there are encodings of several objects concatenated together. These lists of objects are easy to decode because, as implied by the name, the prefix free encoding of an object will never be a prefix within the encoding of another distinct object using the same encoding function.

Definition Prefix Free Encoding

$$E : O \to \{0,1\}^*$$

E is prefix free if $\forall x \neq y \in O, E(x)$ is not a prefix of E(y)

Example $NtoB$ (binary encoding of natural numbers)

This encoding is not prefix free. Consider NtoB(2) = 10 and NtoB(5) = 101. 10 is a prefix of 101, so it does not satisfy the PFE property.

Example $\overline{ZtoB} : O \to \{0,1\}^*$

This is the same function as in the last lecture, in which bits are duplicated and 01 is added to indicate the end. It IS prefix free because the 01 end symbol will never be found in an encoding before the end of the encoding.

Theorem Suppose we have a prefix-free encoding $E : O \to \{0,1\}^*$.

Define $\overline{E}((x_1, x_2, ..., x_3)) = E(x_1) \circ E(x_2) \circ E(x_3) \circ ... \circ E(x_n)$

Then $\overline{E}$ is a valid encoding of O^*.

Proof Suppose someone gave us the binary sequence $E(x_1) \circ E(x_2) \circ E(x_3) \circ ... \circ E(x_n)$.

We can decode it as follows:

- Keep reading from left to right until the sequence matches an encoding
- Once we find it, chop it off to recover the first object and proceed

Thus, since our encoding has a decoder, it is a valid encoding.

A quick remark regarding effiency of PFE:

- length of PFE(x) is 2|E(x)| + 2, which leads to exponential growth in nested encodings (i.e. lists of lists)
- instead, we can get a conversion where the new encoding has length $|E(x)| + 2log_2(|E(x)|) + 2$ by encoding the length of the objects instead of the data itself

Additionally, concrete code of this encoding and decoding in action can be found here.

Summary We can view all inputs (images, videos, strings, graphs, etc.) as binary strings.

Algorithms

Informally, algorithms are a series of steps to solve some problem, or a way to transform inputs to a desired output. How can we formalize this?

Definition Specification

Function $f : \{0, 1\}^* \to \{0, 1\}^*$

i.e. $Mult : N \times N \to N$

Additionally, we can define steps as "some basic operations."

Boolean Circuits

A boolean circuit uses AND/OR/NOT as basic operations. For concision, we will omit their definitions. AND is often denoted $\wedge$, OR is $\vee$, and NOT is $\neg$.

Example MAJ3: $\{0, 1\}^3 \to \{0, 1\}$

$$MAJ3(a, b, c) = \begin{cases} 1 & \text{if } a + b + c \geq 2 \\ 0 & \text{else} \end{cases} \tag{2.1}$$

In terms of boolean operations, this can be defined as follows:

$$MAJ3(a, b, c) = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$$

Example

XOR2: $\{0,1\}^2 \to \{0,1\}$

$$XOR2(a,b) = (a \wedge \neg b) \vee (\neg a \wedge b)$$

XOR3: $\{0,1\}^3 \to \{0,1\}$

$$XOR3(a,b,c) = \begin{cases} 1 & \text{if odd number of a, b, c are 1} \\ 0 & \text{else} \end{cases} \tag{2.2}$$

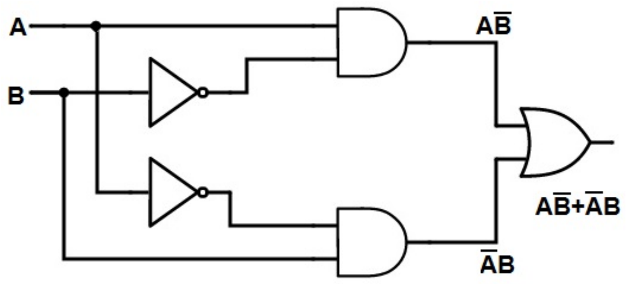
Boolean implementation of XOR3:

$$XOR3(a,b,c) = XOR2(XOR2(a,b),c)$$

$$XOR3(a,b,c) = a \oplus b \oplus c$$

In the case of a boolean circuit, "solving the problem" means computing the function and our "basic steps" are our boolean operations.

Boolean circuits can be represented using DAG's (Directed Acyclic Graphs) in circuit diagrams as follows:



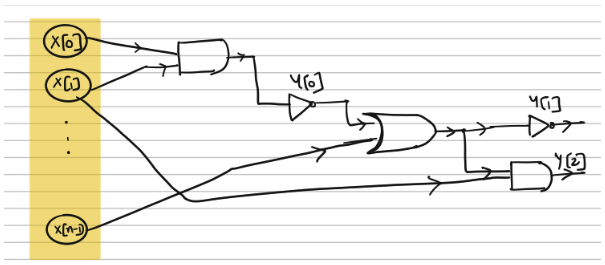
3 | Boolean Circuits

Continuing on from last time, Boolean Circuits use AND/OR/NOT as basic operations and can be represented as DAG's. Let's formally define them.

Definition Boolean Circuits

A (n, m, s)-Boolean Circuit is a DAG with $n + s$ vertices. n refers to the number of input variables, m is the number of output variables, and s is the size of the circuit.

- Exactly n of these vertices are labeled as inputs $x[0], x[1], ..., x[n-1]$
- The other s vertices are AND/OR/NOT gates
  - Each AND and OR gate has exactly two incoming edges
  - Each NOT gate has one incoming edges
- m of the gates are labeled as outputs $y[0], y[1], ..., y[m-1]$



We know that DAG's can be topologicallys sorted (i.e. there is a layering of the vertices such that for every edge (i, j), h(i) < h(j)), so Boolean Circuits can be topologically sorted too. This is important for computation of circuit outputs. How can we compute the outputs?

- layer the DAG via topological sort so that all input vertices are in layer 0
- we have computed in layers 0, 1, ..., k - 1
- now for each vertex in layer k, assign it the value of its operation on its wires (i.e. AND gate should be w1 $\wedge$ w2)
- output $y \in \{0, 1\}^m$ s.t. $y = y[0] \circ y[1] \circ ... \circ y[m-1]$

Choosing Basic Operations

We stated in our definition of a boolean circuit that the basic operations it uses are AND/OR/NOT. Why did we pick these operations in particular? Could we have picked some other operations?

Definition NAND

$$\text{NAND: } \{0,1\}^2 \to \{0,1\}$$

$$\text{NAND}(a,b) = \neg(a \wedge b)$$

NAND Circuits are defined similarly to Boolean Circuits, the only difference being the basic operations available. In NAND Circuits, only NAND is used. Is there a meaningful difference between the two? Is one more powerful than the other?

We can show that NAND circuits can be easily simulated using Boolean gates.

- simply convert NAND gates to and AND followed by a NOT
- NAND $\rightarrow$ AND + NOT

Is the opposite true? Can we express Boolean Circuits only in terms of NAND gates?

- NOT(a) = NAND(a, a)
- AND(a, b) = NOT(NAND(a, b)) = NAND(NAND(a, b), NAND(a, b))
- OR(a, b) = NAND(NOT(a), NOT(b)) = NAND(NAND(a, a), NAND(b, b))

So yes, we can simulate a Boolean Circuit with a NAND circuit.

Theorem Boolean circuits are **equivalent** to NAND circuits in computational power.

Definition Equivalent

f is computable by a Boolean Circuit $\iff$ f is computable by a NAND circuit.

Theorem

Every function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed by a Boolean circuit of size $O(n * m * 2^n)$.

Proof (for m = 1)

Given $f : \{0,1\}^n \to \{0,1\}^m$, an arbitrary function mapping boolean strings to binary, define the set

$$S = \{\alpha : f(\alpha) = 1\}$$

For each binary string $\alpha \in 0, 1^n$, define

$$E_\alpha : \{0,1\}^n \to \{0,1\}$$

$$E_\alpha = \begin{cases} 1 & \text{if } x = \alpha \\ 0 & else \end{cases} \tag{3.1}$$

Example So if $\alpha = (1, 1, ..., 1)$, we can construct a circuit using a chain of AND gates in order to compute E_α. The same is true if $\alpha = (1, 1, ..., 1, 0)$, except that we must add in a NOT gate after the final AND gate. If we extend this logic to the entire domain of $\{0,1\}^n$, it is clear that we can construct a circuit for arbitrary values of α in the domain.

We can use E_α to compute arbitary f.

Consider the set S from before, which is simply the set of binary strings in which f is 1. We have shown that it is possible to create a circuit for arbitary E_α (which is equal to 1 if the input x = α), so it is easy to create a circuit that computes f. It is simply

$$f(x) = \text{OR}(E_{\alpha_0}(x), E_{\alpha_1}(x), ..., E_{\alpha_{n-1}}(x)))$$

which follows from the logic that if x is equal to any one of the binary strings that cause f(x) to be 1, then the circuit should output 1. By chaining together all the circuits we constructed before with OR gates, we can construct the final circuit for arbitary f.

How many gates will we use?

$$\begin{aligned} \#\text{ gates used } &\le |S|(2n-1) + (|S|-1) \\ &= O(n * 2^n) \end{aligned}$$

We can extend this logic beyond the $m = 1$ by using the above to compute each bit of y.

Remark: We can do better with a circuit of size $O(\frac{2^n}{n})$, but this goes beyond the scope of this class.

Some functions, such as addition and multiplication, are frequently used. As a result, engineers work to make far more efficient circuits for them than guaranteed above. Addition can be computed in an $O(n)$ circuit while multiplication can be computed in an $O(n^2)$ circuit. However, some functions require an exponential number of gates to compute. We will show this.

Big Idea We can encode circuits as binary strings. Two corollaries to this idea: 1. Some functions need exponential size circuits and 2. Universal circuits exist (i.e. general computers)

Theorem Every (n, m, s) NAND Circuit can be represented by a binary string of length O((n + s)log(n + s)).

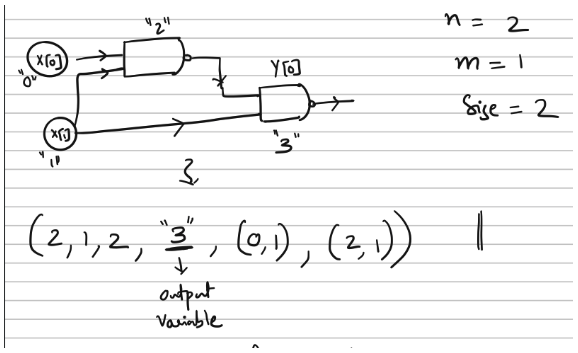
Proof We define size$_{n,m}$(s) to be all circuits on n inputs, m outputs, with at most s gates. The goal is to find an encoding E from this function to binary strings.

What do we need to specify in a NAND circuit?

- how many inputs? n
- how many outputs? m
- how many gates? $s_0 \leq s$
- which nodes correspond to output variables (list of indices corresponding to y[i])
- links between gates (pairs of nodes whose outputs are inputs to current index node)

We can store these features as a tuple and encode it in the standard nested PFE.

Example



How exactly to get valid encoding of list shown above?

- for n, m, and size we encode them with PFE
- same for node output numbers (we know there are m of them)
- same for pairs (simply read two of them at a time when decoding, we know there will be 2 * s_0 integers b/c 2 integer inputs for every gate)

How many bits are we using?

- recall that the PFE of an integer a takes $\leq 2log_2(a)$ bits
- $2log_2(n) + 2log_2(m) + 2log_2(s_0)$ for first 3 integers
- $m * 2log_2(n + s_0)$ for m integers in output list
- $2s_0 * 2log_2(n + s_0)$ for $2s_0$ integers in pairs
- adding these terms together we determine that this is $\leq 12(n+s)log_2(n+s)$ as stated in the theorem

Corollary 1. Some Functions Require Exponential Size Circuits

Theorem There exists functions $f : \{0,1\}^n \to \{0,1\}$ that require circuits of size $\frac{c*2^n}{n}$ for c > 0.

Proof

First some definitions.

$$\text{All}_n = \{f : \{0,1\}^n \to \{0,1\}\}$$

The above function is the class of all functions from n-length binary strings to 0 or 1.

$$SIZE_n(s) = \{\text{All functions with 1 bit output computable by circuits of size} \leq s\}$$

We want to show that $|All_n| > |SIZE_n(\frac{c*2^n}{n})|$.

- Counting All_n
  - we know there are 2^n possible inputs
  - each of these has 2 possible outputs
  - so we have 2^{2^n} possible functions (2 possible outputs over 2^n rows)
- Counting $SIZE_n(s)$
  - idea: count using encodings
  - # circuits $\leq$ number of binary strings of length $12(n+s)log_2(n+s)$
  - we know the number of strings of length $\leq l$ is $2^0+2^1+2^2+2^3+...+2^l = 2^{l+1}-1$ (geometric series)
  - so the number of circuits $\leq 2 * 2^{12(n+s)log_2(n+s)}$

To finish we need to compare $|ALL_n|$ to $|SIZE_n(s)|$. We will skip the algebraic manipulation (see lecture notes for that), but the conclusion is that $|ALL_n| > |SIZE_n(s)|$ where $s = \frac{2^n}{24n}$. Therefore $\exists$ functions on n bits that require $\frac{2^n}{24n}$ gates to compute.

Corollary 2: Universal Circuits

We know that we can encode a circuit as a binary string. Let's define a a function to take advantage of this fact.

Definition EVAL: $\{0,1\}^{S(n,m,s)} \times \{0,1\}^n \to \{0,1\}^m$

$$\text{EVAL}(C,x) = \begin{cases} C(x) & \text{if C is a valid circuit} \\ 0^m & \text{else} \end{cases} \tag{3.2}$$

Theorem There is a circuit for $\text{EVAL}_{n,m,s}$ of size $O(s^2 logs)$.

One last major idea: Circuits are efficient.

Theorem Physical Extended Church-Turing Thesis(PECTT)

If a function can be computed using s physical resources, then it can be computed by a circuit that uses roughly s gates. In other words, circuits are about as efficient as we can get in terms of computation.

Summary

- circuits can be implemented on physical devices
- every functin can be computed by circuits
- some functions require exponential size circuits
- universal circuits of size $cs^2 logs$ can simulations all size s circuits

4 | Deterministic Finite Automata

Circuits are a great model for bounded input lengths. But what if we want to compute on some unbounded input? We know we can determine a finite answer to an infinite class of questions using an algorithm. Can we use this to create a model of computation for unbounded inputs?

Consider XOR:

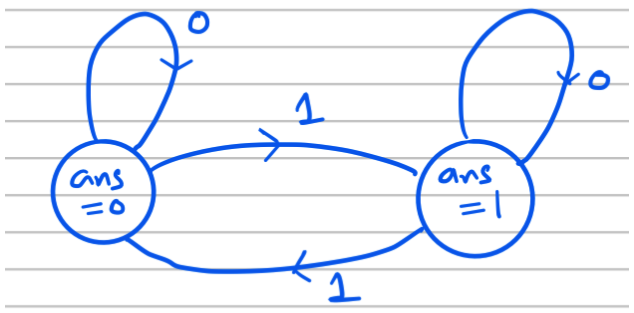
$$\text{XOR} : \{0,1\}^* \to \{0,1\}$$

$$\text{XOR}(x) = \begin{cases} 1 & \text{if number of inputs equal to 1 is odd} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

We can define an algorithm for XOR as follows:

- def XOR(x):
  - ans = 0
  - for i in range(len(x)):
    - ans = (ans + x[i]) % 2
  - return ans

This is an exmaple of aa "Single Pass Constant Memory Algorithm." We can create a diagram to represent it as follows:



Definition Deterministic Finite Automaton (DFA)

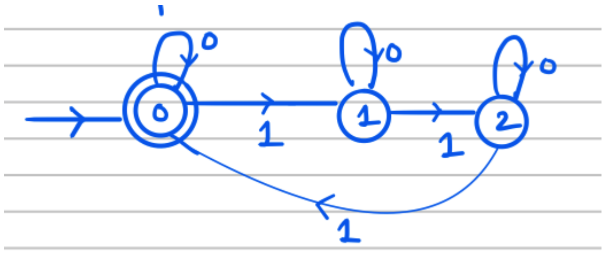
DFA with c states over {0, 1} is a pair $D \equiv (T, S)$, where $T : [c] \times \{0,1\} \to [c]$ and $S \subseteq [c]$. T is known as the transition function and defines the inputs that cause a transition in state. For example, if the current state is 0 and the input bit is 1, the transition function might output 1 to indicate a change in state from 0 to 1. S is the acceptor. 1 is output if the final state is a state $\in$ S.

Example

Consider the function below:

$$D(x) = \begin{cases} 1 & \text{if number of 1's in x is divisble by 3} \\ 0 & \text{else} \end{cases} \tag{4.2}$$

A DFA for this function is as follows



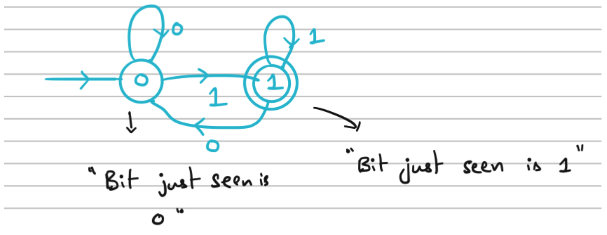
The transition function T is shown below and S, the set of accepting states is $\{0\}$. If we wanted to modify the function be 1 if number of 1's mod 3 was 1, we can simply modify S to $\{1\}$.

inital	bit	state
0	0	0
1	0	1
2	0	2
0	1	1
1	1	2
2	1	0

Example

$$D(x) = \begin{cases} 1 & \text{if x ends in1} \\ 0 & \text{else} \end{cases} \tag{4.3}$$

The DFA for the above function is as follows:

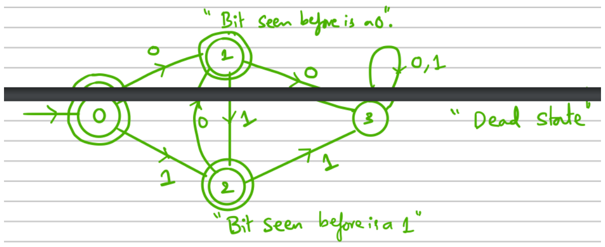


Example

$$D(x) = \begin{cases} 1 & \text{if x has alternating bits1} \\ 0 & \text{else} \end{cases} \tag{4.4}$$

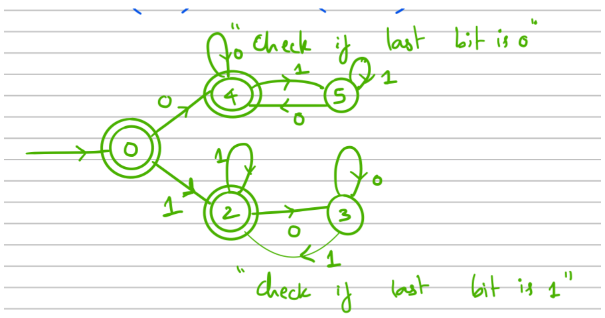
For example, f(01010) = f(10101) = 1 while f(01001) = 0.

The DFA for the above function is as follows:



Example Design a DFS that outputs 1 on strings with the same start and end bit. For example, f(011110) = 1 while f(0110101) = 0.

The DFA for the above function is as follows:



Anatomy of a DFA

Every DFA has *unbounded input length* and bounded

- number of states C
- transition function T
- set of accepting states S

Some more important symbols representing DFA parameters:

- set of states Q of cardinality C
- alphabet Σ (we will use binary)
- starting state s_0 (we will use 0)
- the language L_f, where f is the function computed by D and $L_f = \{x : f(x) = 1\}$

Composition of DFA's

It may be useful to put together several DFA's and connect them with an operation. For example, consider f_1 computed by DFA D_1 and f_2 computed by DFA D_2. We may want to find $f = f_1(x) \wedge f_2(x)$. Can we compute f using a single DFA?

It's important to once again note that DFA's can **only** be used to compute single pass, constant memory algorithms. Therefore, we must find a way to compute f_1 and f_2 within the same pass if we want to compute f.

This may be trivial at times, such as in the case where f_1 determines if there are an even number of 1's and f_2 determines if there are a multiple of 3 1's. In this case, we need only create a DFA that outputs 1 if there are a multiple of 6 1's. Let's try to generalize this beyond the special case.

Theorem DFA's are closed under "ANDs". f_1, f_2 are computable by DFA's $\iff$ $f_1 \wedge f_2$ computable by DFA's.

Put another way, if L_1, L_2 are recognized by DFA's, then there is a DFA that recognizes $L_1 \cap L_2$.

Proof We have $D_1 = (T_1, S_1)$ that computes f_1 with C_1 states and and $D_2 = (T_2, S_2)$ that computes f_2 with C_2 states. We want to compute $f_1 \wedge f_2$. The idea is to run D_1 and D_2 in parallel.

We will have $C_1 \times C_2$ states represented as (i, j) pairs, where $i \in \{0, 1, ..., C_1 - 1\}$ and $j \in \{0, 1, ..., C_2 - 1\}$.

We will define $T : (C_1 \times C_2) \times \{0, 1\} \to C_1 \times C_2$. In particular, $T((i, j), a) = (T_1(i, a), T_2(j, a))$. In other words, we simply use the transition functions from each DFA to find the next ordered pair to transition to. In this way, we are computing each function in parallel.

Finally, we define $S = \{(i, j) : i \in S_1 \text{ and } j \in S_2\}$. This completes the construction.

We can repeat the same idea for OR, with the only difference being how we define S. For OR, $S = \{(i, j) : i \in S_1 \text{ or } j \in S_2\}$. NOT is trivial as well, as we simply need to take the complement of the existing DFA's accepting set.

In practice, DFA's are often used for recognizing patterns. To discuss this, we introduce a new definition.

Definition Concatenation

f, g are functions from binary strings to binary. The concatenation of f, g $f \circ g = 1 \iff f(x_1) = 1$ and $g(x_2) = 1$ where x_1, x_2 are consecutive substrings of x.

Can we compute the concatenation of any two functions using a DFA?

5 | Non-Deterministic Finite Automata

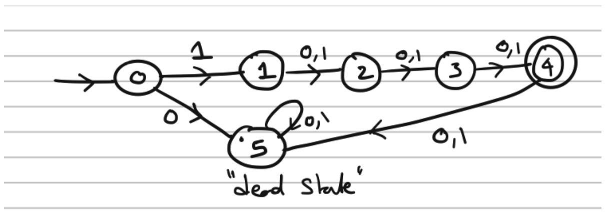
To begin discussion of NFA's, we will first discuss function concatenations. For convenience, the definition is repeated below:

Definition Concatenation

f, g are functions from binary strings to binary. The concatenation of f, g $f \circ g = 1 \iff f(x_1) = 1$ and $g(x_2) = 1$ where x_1, x_2 are consecutive substrings of x.

Example Function Concatenation

Consider f_1 and f_2. f_1 returns 1 for all x and f_2 returns 1 if x starts with 1 and has length exactly 4. Both of these are computable by DFA's as shown below (the constant one is pretty obvious so it has been omitted).



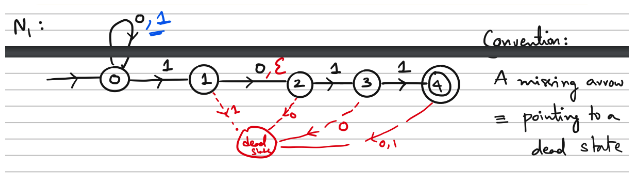
Can we compute the concatenation of these two functions using a DFA? In fact, we can. Using just 16 states, we can use a DFA to compute it. To clarify what exactly we're computing, $f_1 \circ f_2(001010) = 1$ since $f_1(00) = 1$ and $f_2(1010) = 1$.

Example

Consider a function $f_{reverse}(x)$, which simply returns the value of f when input with the reverse of x i.e. the bits are written right to left. Can we compute this with a DFA?

It seems to be exactly the opposite of what we can compute using a DFA. DFA's always implement one pass, constant memory algorithms and it would seem necessary to violate this in order to implement $f_reverse$. We will revisit this subject.

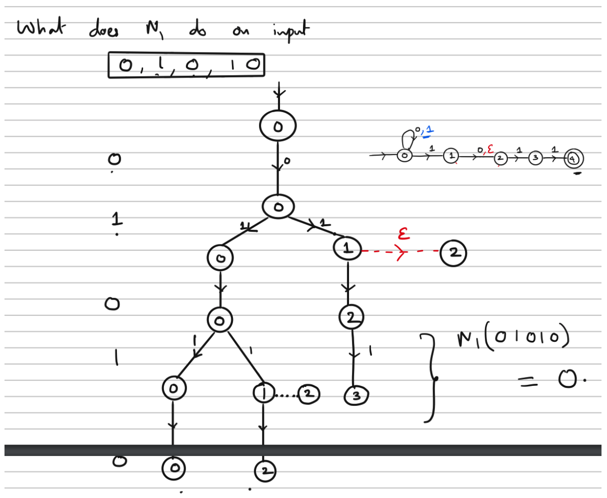
Non-Deterministic Finite Automata



NFA's are similar to DFA's, with some small variations:

- they can have mulitple outgoing edges with the same states
- some edges can be missing (by convention, this indicates they lead to dead states)
- some edges are labeled ε

Example



The above represents the branching diagram of the NFA. If any branch has a state within S after the last bit is read, then 1 is output.

Definition Non-Deterministic Finite Automata

An NFA, N is defined by a transition function and a set of accepting states. So N = (T, S).

$$T : [C] \times \{0, 1, \varepsilon\} \rightarrow \text{Power}([c]) \tag{5.1}$$

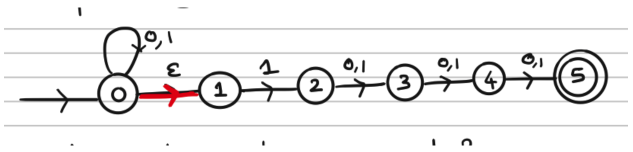
$$S \subseteq [c] \tag{5.2}$$

$$Power([c]) = \{I : I \subseteq [c]\} \tag{5.3}$$

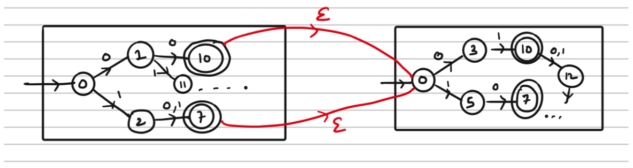
In other words, the transition function is mapping the current state and the current bit input to a subset of all states in the NFA.

Why NFA's

One might wonder, what are NFA's good for? Well, we can use them to compute the concatenation from earlier where f_1 is a constant and f_2 is 1 if the first bit is 0 and the input length is exactly 4. Logically, the concatenation of these two outputs 1 if the 4th bit from the end is 1 and 0 otherwise. The NFA is pictured below:



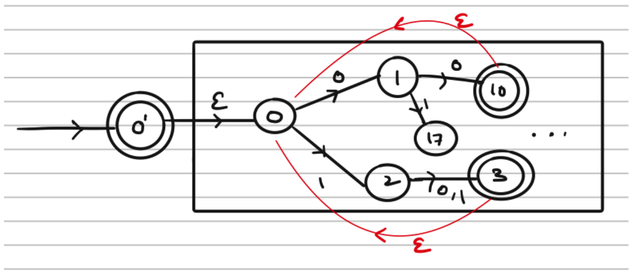
Important Takeaway: In general, if two functions are computable by DFA's, we can compute their concatentation using an NFA in which we simply add ε transitions pointing from accepting states of the first function to the start of the second.



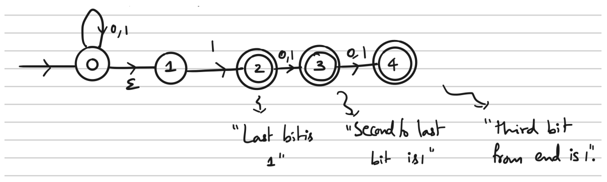
Kleen* Operation on Functions

We already discussed Kleen* as a set operation, but as a function operation it is defined as f*(x) = 1 if x can be broken up into $x_0, x_1, ..., x_n$ such that $f(x_i) = 1$ for all i.

Building upon the takeaway from earlier, if f is computable by a DFA, then f* is computable by an NFA. We simply add a dummy state leading into the previous DFA's initial state and add epsilon transitions pointing from all accepting states to the initial state.



Example Construct an NFA where L = { all strings that have a 1 in the last three positions }.



This works by branching off at bit and checking 1) if there are less than or equal to three bits remaining and 2) if one of those bits is a 1 (we only progress to an accepting state if 1 is seen and we go to dead state if more than three bits are read).

Recall the f_{reverse} example. $f_{\text{reverse}}(x)$ is one if f is 1 when the input string is the reverse of x. If f is computable by a DFA, what about f_{reverse}?

We can compute it with an NFA! The process is as follows:

1. add a new start state to the DFA for f
2. add ε transitions from new starting state to all previous accepting states
3. reverse the direction of all arrows
4. make old start state the new accept state

5.1 NFA's vs DFA's: Computability

One might wonder, are NFA's more powerful than DFA's? They seem to have more functionality with epsilon transitions and multiple possible outgoing edges from the same input bit, so one would expect that this is the case. However, a **mind boggling** theorem shows that this is not the case!

Theorem Every NFA has an equivalent DFA!

For every NFA N, $\exists$ a DFA D such that N(x) = D(x) $\forall$ x. As a result, the concatenation of two functions, the kleene star operation on a function, and

Proof The main idea is that for each level of our NFA input tree we need to know what states are reachab le at that level. We can merge redundant states at the same level into one.

First, we will assume the case where there are no ε transitions. Given an NFA $N = (T_N, S_N)$The goal is to find a DFA $D = (T_D, S_D)$ such that $N(x) = D(x) \forall x \in \{0,1\}^*$. If the NFA is on states [c], we will construct a DFA whose states corresond to subsets of Power([c]) i.e. if c = 3, the DFA states will be $\emptyset, \{0\}, \{1\}, \{2\}, \{0,1\}, \{1,2\}, \{0,2\}, \{0,1,2\}$. The number of states in our new DFA is 2^c.

We construct T_D with the following:

$$T_D : Power([c]) \times \{0,1\} \to Power([c])$$

$$T_D(I, a) = \bigcup_{i \in I} T_N(i, a)$$

And the accepting states:

$$S_D = \{R \subseteq [c] : R \cap S_T \neq \emptyset$$

The start state should just be $\{0\}$.

In words for each of these, for each element in I we group together the results from T_N removing duplicates. For S_D, we include all subsets that have a value in the original set of accepting states.

What if we add epsilon transitions?

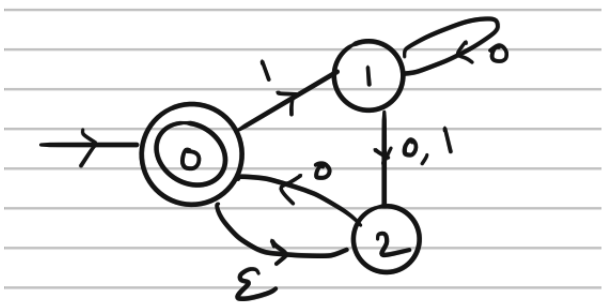
Definition Eps(I) = all states reachable by taking *varepsilon* edges from i + {i}

To finish, we simply wrap our transition function and new start state with Eps and we are done.

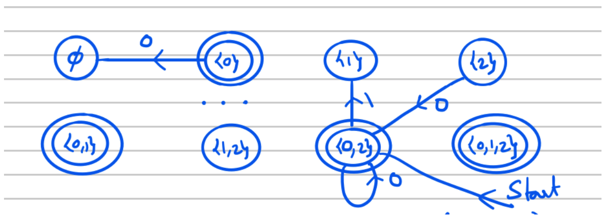
$$T_D(I, a) = Eps(\bigcup_{i \in I} T_N(i, a))$$

The start state should just be Eps($\{0\}$). ■

Example Given an NFA,



We can construct a DFA like the following (incomplete diagram),



Note that the ∅ state denotes the dead state. We create a state for each subset of states in the NFA and use the formula above to get arrows. The accepting states are simply all sets that contain the accepting states from the

5.2 Pattern Matching

In a standard pattern matching problem, we have some "text" and a "pattern," with the length of the pattern being much shorter than the length of the text. The question is, does the pattern occur within the text?

A naive O(m * n) algorithm is the following (assuming p is lengthm and x is length n):

```
def patternmatch(x, p):
    l = len(p)
    for i in range(0, len(x) - l):
    if x[i, i + l] == p:
        return 1
    return 0
```

We can do better. Using the Knuth-Morris-Pratt (KMP) algorithm, we get an O(m + n) single-pass algorithm

- given P, first find a DFA D on m + 1 states O(m)
- for any string x, p occurs in x $\iff$ D(x) = 1
- now mimic behaviour of D on x O(n)

Regular Expressions

Regular expressions are a programming tool used to efficiently find some pattern within some text. We will formally define them here.

Definition

Base cases:

- "0" is a regex
- "1" is a regex
- $\emptyset$ is a regex
- *varepsilon* is a regex (empty string)

Compound cases (r_1, r_2 are regex):

- r_1r_2 (concatenation) is a valid regex
- r_1^* is a valid regex (repetition 0-n times)
- $(r_1|r_2)$ is a valid regex (r1 or r2)

Example

1. Does x = 0 match (0 | 1)? yes
2. x = 01 match (0 | 1)(0 | 1)? yes
3. x = 00 match (0 | 1)1(0 | 1)? no
4. What matches $(0|1)^*$? All strings
5. $(0|1)^*0$? strings ending in 0
6. $(10)^*$? repetitions of 10
7. $(0(10)^*|(10)^*1|(01)^*|(10)^*)$? all strings with alternating symbols
8. $0^*10^*10^*10^*$ strings with exactly three 1's
9. $((0^*10^*10^*10^*)^*|0^*)$? all strings with number of ones divisible by 3
10. $(0|1)^*1(0|1)(0|1)(0|1)$? all strings with 1 4th bit from the end

Theorem Regex is computationally equivalent to a DFA.

For every regex r, there is a DFA D such that the output of the regex is the same as the output of the DFA for all x.

For every DFA D, there is a regex r such that the output of the regex is the same as the output of the DFA for all x. A function is considered **regular** if there is a regex (or DFA) computing it.

We can compute regex pattern matching in O(n * m) time. The GREP algorithm is the following:

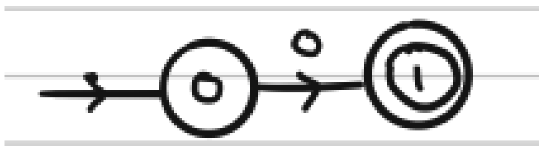
- convert the regex to an equivelant NFA N with O(m) states and transitions in O(m) time
- simulate NFA N on the input X in O(m * n) time

Proof By construction,

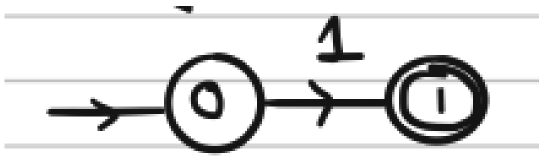
We defined regex inductively, so we will also build the NFA for a regex inductively.

Base cases:

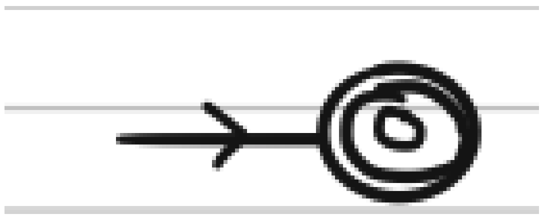
- "0"

  –
- "1"

  –
- "ε" (empty string)

  –

Compound cases:

- concatenation i.e. r = $r_1 r_2$
  - we saw that we can compute the concatenation of two NFA's by adding epsilon transitions from the accepting state of the first to the initial state of the second
- union i.e. r = $r_1 | r_2$
  - we can compute or of two NFA's by adding an epsilon transition from the start of one to the start of the other
- kleen* i.e. r = $r_1 *$
  - we saw that we can compute kleen* by adding a new (accepting) start state and epsilon transitions from accepting states to initial starting state

We have shown that there is an equivalent NFA for every regex. A regex can be converted into an NFA in linear time (as done in GREP).

How can an NFA be simulated efficiently? If an NFA has m states, we can simulate it on a string of length n in $O(nm^2)$ time by only keeping track of one copy for each reachable branch.

5.3 Can DFA's compute everything?

We know that DFA's are good for algorithms that require only constant memory and a single pass. Does this suffice for any function? Consider the following:

$$MAJ(x) = \begin{cases} 1 & \text{if at least half the input bits are 1} \\ 0 & \text{else} \end{cases}$$

MAJ is not regular, meaning there is no DFA that can compute MAJ. This is because we have to "count" the nuber of bits that are 1.

Example

$L_1 = \{x : \text{contains an equal number of 0's and 1's}\}$

This is also not computable by DFA's for the same reason.

Based on these two examples, one might thing that anytime "counting" is required for a function that it is not regular. However, this is not necessarily the case. In some cases, there might be shortcuts that yield the same result using a DFA. Consider the following

$$L_2 = \{x : \text{contains an equal number of (01)'s and (10)'s}\}$$

L_2 is actually regular! In this case, there are an equal number of 01 and 10 if and only if the start and end bits are the same. So we can easily create a DFA for this.

In order to prove functions are not regular, we make use of the "Pumping Lemma."

Pumping Lemma

If f is a regular function, there exists a number p such that every string x where f(x) $=$ 1 of length $\geq$ p, can be written as x $= a \circ b \circ c$.

- $f(a \circ b^i \circ c) = 1$ for all $i \geq 0$
- length of b > 0
- length of $a \circ b \leq p$

Intuitive description: If a language L is regular then every sufficiently long string $x \in L$ has a piece that can be repeated an arbitrary number of times while still being in L.

We can use this to prove that in the example above, MAJ is not regular.

Proof

Towards a contradiction, suppose MAJ is regular. This implies there must exist a number P such that the conditions of pumping lemma hold.

Consider the string x that has p 0's followed by p 1's. By pumping lemma, there should be a way to split x into three pieces such that the middle piece can be repeated an arbitrary number of times and the properties of pumping lemma still hold. However this is not the case due to the condition that the length of ab must be less than p.

Due to this restriction in length, the split between a and b will always be inside the 0's, meaning b will consist only of 0's. We cannot repeat b an arbitrary number of times because this will cause the output of the function to change to 0. This is a contradiction. Therefore, MAJ is not regular.

■

Example

Prove $L = \{0^k10^k | k \geq 1\}$ is not regular.

Suppose L was regular. Then there must exist p for which PL holds. Consider x = 0^p10^p. By PL, we must be able to split x into abc where length(ab) $\leq$ p. Following a similar argument to proof for MAJ, we can see that the split between a and b must fall within the first round of 0's, and therefore we cannot repeat b an arbitrary number of times or there will be an unequal number of 0's in the first section and the second section. abbc is not in L, but PL says it should be. We have reached a contradiction.

Example

Prove PALINDROME = { x: x = reverse(x) } is not regular.

We pick the same string as in the previous example and pump it to show PALINDROME is not regular.

Example

Prove $L = \{1^{n^2} : n \geq 1\}$ is not regular. i.e. all strings with only 1's and number of 1's is a square.

Suppose L is regular. There exists some number p such that PL holds. Consider X = 1^{p^2}. By PL, we should be able to split up x into a, b, and c such that length(ab) le p and length(b) is not 0. Is abbc in the language? We know that the length(abbc) $\leq p^2 + p$ in the case where length b = p and a = 0. This is less than the next square $(p+1)^2 = p^2 + 2p + 1$, which shows abbc is not in the language. This is a contradiction, so the language is not regular.

Note: There are languages that are not computable by DFAs, but you cannot prove this with Pumping Lemma (Beyond the scope of this course, but we could use Myhill-Nerode Theorem).

Proof Pumping Lemma

The idea is that we have a DFA with p states and a string x with length > p. By the pigeonhole principle, as we travel along the DFA there must be at least one repeated state that forms a loop. We will call the inputs along this loop b, the inputs before the loop a, and the inputs after the loop c. Since b is along a loop, no matter how many times we repeat it the string must be accepted. Since we have reached a state we have not seen, the length of ab must be <= p (since there are p states and one of them has been repeated before seeing the last states). Said another way, no more than p states must have been visited before we encounter a loop.

See a more rigorous proof by construction in the class notes for lecture 11.

6 | Turing Machines

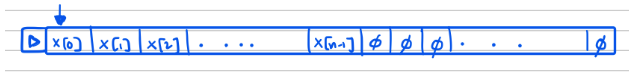
As we showed in the last chapter with our discussion of Pumping Lemma, DFA's are not able to compute all functions. For example, it is impossible to compute if a string is a palindrome using a DFA. This is due to the limitation of memory and only single pass algorithms. Can we create a model that can computes on arbitrary length inputs and is not subject to these limitations?

We will discuss Turing Machines, introduced by Alan Turing in 1936. This model of computation is as powerful as it gets. It will give us the ability to move the "head" both directions on input and read and write to a variable amount of memory. Essentially,

$$\text{Turing Machines} = \text{DFAs} + \text{left/right movement on input} + \text{read/write memory}$$

Anatomy of a Turing Machine

A turing machine consists of an infinitely long tape, on which the inputs are loaded into the first n spots. It can be thought of as an array. There are a finite number, k, states. The head can move left, right, or stay depending on the current state and the current symbol being read.



In each step of computation, we are in a state i and read the bit at the head of the tape. We can take several actions including

- changing the state
- write something new at the head
- move the head (left, right, or stay)

Definition Turing Machines

- k states
- $\Sigma \supseteq \{0, 1, \triangleright, \varnothing\}$ (there is a finite alphabet)
  - $\triangleright$ denotes the start of the tape
  - $\varnothing$ denotes nothing is at that position in the tape
- Transition function $\delta : \{0, 1, ..., k-1\} \times \Sigma \times \{L, R, S, H\}$
  - $\delta(\text{state}_i, a) = (\text{state}_j, b, L)$
  - a is the symbol being read, state_j is the new state to go to, b is the symbol to write at the head, and L is the direction to go in
  - H stands for halt

Computation:

```
Start with head at x[0]
State: "0" (starting state)
Repeat:
    (new_state, new_symbol, A) = delta(current_state, Tape[Head])
    cur_state = new_state
    Tape[Head] = new_symbol
    if A == L: Head = max(0, Head - 1)
    if A == R: Head += 1
    if A == S: Head = Head
    if A == H: exit()
```

By convention if a Turing Machine M halts on an input, then the output is the contents of the tape up to and including the head. If M does not halt, M(x) = "$\perp$". We say a function is computed by a TM M if f(x) = M(x) for all x. A language L is recognized by M if $x \in L \implies M(x) = 1$ and $x \notin L \implies M(x) = 0$.

Example

$k = 1, \Sigma = \{0, 1, \triangleright, \varnothing\}$

$$\delta_M(0, 0) = (0, 1, R)$$
$$\delta_M(0, 1) = (0, 0, R)$$
$$\delta_M(0, \varnothing) = (0, \varnothing, H)$$

This Turing Machine simply computes the complement of the input. That is if we load in 1001, we will get 0110 as output.

Example Now we show that Turing Machines are more powerful than DFA's by showing we can compute a function that DFA's cannot: MAJ.

$$MAJ : \{0,1\}^* \to \{0,1\}$$
$$MAJ(x) = \begin{cases} 1 & \text{if there are at least as many 1's as 0's} \\ 0 & \text{else} \end{cases}$$

The idea here is to try to match every 0 we see to a 1. If we are unable to match a 0, then there are more 0's than 1's and we output 0.

Pseudocode:

```
1. Scan to to the right until a 0 is found
2. If no 0 is found:
     "clean up" the tape and return 1
3. If a 0 is found:
     Mark the 0 as seen
     Go to the start of the tape
     Scan the input to find a 1
     If 1 found:
         Mark 1 as seen
         Go to start
     If 1 not found:
         clean up and return 0
```

Click here for a visualization.

Example Another function that DFA's could not compute was palindrome. Recall that palindrome returned 1 if the input string was the same forwards as it was backwards.

Here the idea is to look at the first symbol in the string, then proceed to the end and look at the last symbol. If they are not the same, we return 0. If they are the same, we mark both symbols as seen and proceed to the first unseen symbol and repeat.

Pseudocode:

```
1. Enter state that remembers the first bit x: GoEndx and mark first bit as seen
(overwrite with a)
2. Go to the end
3. If the last bit matches x
    Replace it with nothing and proceed to first unseen bit
    Repeat
4. If the last bit does not match x
    Clean up the tape and return 0
5. If there are no bits remaining that have not been seen:
    Clean up the tape and return 1
```

I will omit the precise transition function definitions, but that along with a visualization can be found here.

From these two examples, we can see that Turing Machines are more powerful than DFA's as they can compute functions that DFA's cannot. We can take this a step further and say that Turing Machines are essentially as powerful as it gets. They are functionally equivalent to modern day programming languages and can simulate things such as random access.

Theorem

For every python program P, $\exists$ a TM M such that $\forall$ x P(x) = M(x). In other words, Turing machines are computationally equivalent to Python. If P takes T time to compute, M will take T^2 time.

Theorem

A function f is computable if there exists a TM M for which f(x) = M(x) for all x.

Thesis Church-Turing Thesis

Every function that is computable by physical means is computable by a Turing Machine.

The fact that every python program (and in fact every implementation of a function) can be simulated using a Turing Machine buys us some very nice abilities. We call them, "Having Our Cake and Eating it Too."

Definition HOCAEIT Principle

HOC: To show something is computable you can use a high level programming language.
EIT: To show something is uncomputable, we only have to show that turing machines cannot compute it.

6.1 Universality

Just like for circuits, we can create a universal turing machine that can simulate all turing machines. As before, we use the idea of code as data to implement this. If we can encode a Turing Machine as a binary string then we can pass them as input to a Turing Machine (which essentially compiles & executes it). First, let us think about how we might encode any arbitrary Turing Machine.

Encoding

Recall the definition of a Turing Machine, which can be completely described by a transition function $\delta : [k] \times \Sigma \to [k] \times \Sigma \times \{L, R, S, H\}$, where k is the number of states and Σ is the alphabet. We can represent every possible set of inputs and outputs to δ as a 5-tuple as in the following table:

State	**Input**	**New State**	**Write Bit**	**Action**
0	a_0	17	a_{10}	R
0	a_1	10	a_4	L
...	...	...	...	...
k - 1	a_{l-1}	4	a_0	S

We use this fact to determine the following encoding form where l is the number of symbols in the alphabet and k is the number of states:

$$(k, l, ((0, a_0, 17, a_10, R), (0, a_1, 10, a_4, L), ..., (k-1), a_{l-1}, 4, a_0, S))$$

We then encode every integer as a binary string using $ZtoB$ and every tuple using some prefix free encoding. In total the length of the encoding will $O(kl(logk + logl))$.

- we have kl tuples
- each tuple has 5 integers, which can be represented in $O(logk + logl)$ (since each integer is at most $\max(k, l)$)

Theorem

A turing machine M can be represented as a binary string of length $O(kl(logk + logl))$ denoted $\langle M \rangle$.

Note: This is a one-to-one relationship, so there are strings that are not valid turing machines. In this case, we pair these strings with the trivial TM that always outputs 0.

Evaluation

Now that we have successfully encoded our TM, we want to define a function that can determine the output of any encoded TM passed into it.

Definition $EVAL : \{0,1\}^* \to \{0,1\}^* \cup \bot$

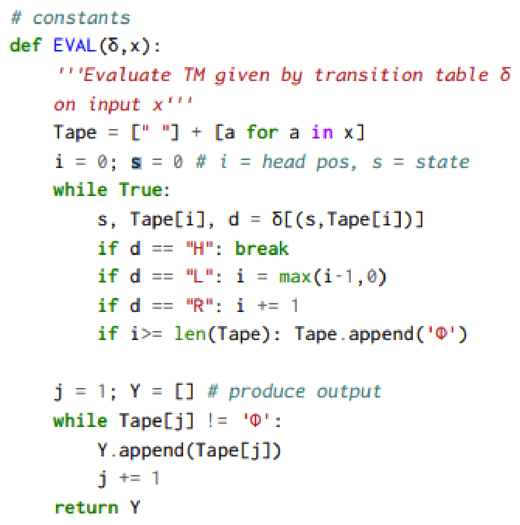
$$EVAL(M,x) = \begin{cases} M(x) & \text{if M is a valid TM} \\ 0 & \text{otherwise} \end{cases}$$

Theorem

EVAL is computable (Turing 1936). There exists a TM U such that $U(M,x) = EVAL(M,x)$ $\forall$ inputs.

Proof

We can use the HOC principle to show that EVAL is computable by a TM by showing it is computable with a Python program. The specific program is as follows (notice it is very similar to the psuedocode we used to define Turing Machines)



```
# constants
def EVAL(δ,x):
    '''Evaluate TM given by transition table δ
    on input x'''
    Tape = [" "] + [a for a in x]
    i = 0; s = 0 # i = head pos, s = state
    while True:
        s, Tape[i], d = δ[(s,Tape[i])]
        if d == "H": break
        if d == "L": i = max(i-1,0)
        if d == "R": i += 1
        if i>= len(Tape): Tape.append('Φ')

    j = 1; Y = [] # produce output
    while Tape[j] != 'Φ':
        Y.append(Tape[j])
        j += 1
    return Y
```

We know that since every program has a corresponding TM, there must be a TM to compute EVAL.

Implications of Universality

A universal TM is essentially just a general purpose computer. Any function can be computed using it. We have relatively small universal TM's using just 25 states and the standard alphabet. This implies that we can compleley describe a model to compute any function using just 500 numbers (25 states x 4 symbols x 5-tuple).

Some more notes:

- this brings up the concept of metacircular evaluators; for example the C compiler is written in C
- universality transcends the specific model
  - there exists a python program that runs all python programs

– there exists a python program that compiles and executes all java programs

Definition Turing Complete Languages

Anything that can be used to simulate a universal turing machine is considered turing complete. The implications of this are quite interesting: any turing complete language can be used to implement any programming language or compute any function that is computable. For example, theoretically we can use LaTeX to write a C compiler or HTML+CSS to write a Python interpreter (both of these are Turing Complete)!

Check here for more examples of Turing Complete languages (including some very unexpected ones like Minecraft).

6.2 Uncomputability

The statement "A function is computable if there is a Turing Machine that computes it" implies the existence of uncomputable functions and indeed this is true.

Theorem There exist uncomputable functions.

A function f is uncomputable if it cannot be computed by a TM.

Consider a new function, the "Toddler" function, that we define below:

$$\text{Todd} : \{0,1\}^* \to \{0,1\}$$
$$\text{Todd}(\langle M\rangle) = \begin{cases} 1 & \text{if } M(\langle M\rangle) \text{ halts and the first bit of output is 0} \\ 0 & \text{otherwise} \end{cases}$$

In other words, TODD does the opposite of what M does, much like a toddler. We will see that TODD is uncomputable. Another way to think of TODD is as follows:

$$\text{Todd} : \{0,1\}^* \to \{0,1\}$$
$$\text{Todd}(\langle M\rangle) = \begin{cases} 1 & \text{if } M(\langle M\rangle) \text{ halts and the first bit of output is 0} \\ 0 & \text{if } M(\langle M\rangle) \text{ halts and the first bit of output is 1} \\ 0 & \text{if } M(\langle M\rangle) \text{ does not halt} \end{cases}$$

Proof

Suppose there was a machine, N, that computed Toddler. Consider what should happen if we try to compute TODD using N given the description of N as input. That is, what is $N(\langle N\rangle)$? It should be equal to Todd($\langle N\rangle$) if N computes Todd. However, we will see that by definition this is not the case.

Case 1: If N halts on $\langle N\rangle$ and the output is 1

- then Todd($\langle N\rangle$) should be 0
- clearly N does not compute Todd since the outputs are not the same
- we have reached a contradiction

Case 2: If $N(\langle N\rangle)$ halts and the output is 0

- then Todd($\langle N\rangle$) should be 1
- once again, N does not compute Todd since the output differs
- we have reached a contradiction

Case 3: If If $N(\langle N\rangle)$ does not halt

- we will never know what to output
- Toddler is nicely defined in this case (it should equal 0)
- N cannot compute Toddler since it does not have the same output as Toddler would have in this case

In every case, we see that N returns the wrong value so it cannot be that it computes Todd. (Reminder that a TM M is said to compute a function f when for all inputs x, $M(x) = f(x)$). So we conclude by contradiction that Todd is uncomputable.

■

An alternative vizualization once again starts with the assumption that N computes TODD. If this is true then following must hold:

$$N(\langle N\rangle) = \text{TODD}(\langle N\rangle) = \begin{cases} 1 & \text{if } N(\langle N\rangle) \text{ halts and first bit is 0} \\ 0 & \text{otherwise} \end{cases}$$

But this is clearly a contradiction because it implies that $N(\langle N\rangle) = \neg N(\langle N\rangle)$. It's like saying $0 = 1$, which is clearly not true. So TODD must be uncomputable.

■

Intuitively, the main issue with the machine N that "computes" Toddler is that N might not halt. The fact that Toddler is uncomputable implies that many other more practical problems are uncomputable. Consider this: can we create a program to determine if another program will terminate? This would be very useful in applications such as OS or designing programming languages or debuggers, etc.

We can define HALT as follows:

$$\text{HALT} : \{0,1\}^* \to \{0,1\}$$
$$\text{HALT}(\langle M \rangle, x) = \begin{cases} 1 & \text{if } M \text{ halts when run on } x \\ 0 & \text{else} \end{cases}$$

Unfortunately, halt is uncomputable.

Proof

We can use a concept known as reduction to show this. If HALT was computable, then we could use it to compute TODD as follows.

Solve Todd using Halt:

```
1. If Halt(<M>, <M>) = 1
    Run M on <M>
    Output the opposite of the first bit.
2. Else
    Output 0
```

If you're paying close attention, an important question arises: why doesn't the same contradiction from before hold when using HALT to compute TODD? Specifically, why is it that we can safely determine the output of <M> and return the opposite in a way that does not contradict the original function? This is mostly because we are not using N (the turing machine assumed to compute TODD) to directly compute TODD. We first check if N halts, and then output the opposite of its result if so. The problem with using N directly was that the machine could end up in a loop, and in this case we don't know what the opposite of our output was (there will never be any). Instead, we are using a different Turing Machine that uses HALT as a blackbox in order to, via its own definition, compute TODD. So if Halt was computable, then it would also be possible to compute TODD (and x would equal $\neg x$). However, clearly this cannot be the case since we proved TODD was uncomputable. It must be that HALT is uncomputable as well.

■

We can also prove Halt is uncomputable using a more direct approach, rather than a reduction from TODD.

Proof

Suppose there was a machine H that computed HALT. Consider the following program:

```
def CANTSOLVEME(<M>):
    if H(<M>, <M>) == 1:
        while true: continue
    else:
        return 0
```

What does CANTSOLVEME do if we input a TM that computes CANTSOLVEME?

- If CANTSOLVEME(<CANTSOLVEME>) halts and returns some output, then we should enter the infinite loop and not halt! This is a contradiction.
- If CANTSOLVEME(<CANTSOLVME>) doesn't halt then we should return 0 and halt. Another contradiction.

No matter what happens, we get a contradiction. So HALT must be uncomputable.

∎

Is Halt really a difficult problem? The short answer is yes, even for a relatively simple programs. Consider how you might write a program that determines if the Goldback Conjecture was true. The Goldbach Conjecture says that every even number can be written as a sum of two primes.

We can write a program as follows:

```
def isPrime(n):
    for a in range(2, n):
        if n % a == 0:
            return 0
    return 1
def GOLDBACH():
    n = 4
    while True:
        goldbachN = false
        for p in range(2, n):
            if isPrime(p) and isPrime(n - p):
                goldbachN = True
                break
        if goldbachN: n += 2
        else: return False
```

It is difficult to tell if this program will halt just by looking at it. It will only halt if the Goldbach conjecture is false, otherwise it will proceed forever (since there are an infinite number of even numbers to consider). So yeah. Halt is a hard problem. If HALT was computable, we could use it to determine the answers to many other problems (such as the Goldbach conjecture, which to this day is unproven).

6.3 Reduction

Put simply, a reduction from A to B means that we use B to solve A. We are reducing the task of computing A to the task of computing B.

We showed before how to solve Toddler using Halt. This proved to us that Halt was uncomputable since we already knew Toddler was uncomputable. We can generalize this same approach to work for other problems.

If we have two problems: A and B. We know that B is uncomputable. If we can reduce problem A to problem B, then this is proof that B is also uncomputable.

This works because if we can use problem B to solve problem A, then it must be that A is computable if B is computable. However, if we know that A is uncomputable then we know that B is uncomputable as well. This is because the second statement is the contrapositive of the first. If we could solve B, then we must also be able to solve A. This is a contradiction since we know A is uncomputable.

Turing/Cook Reductions

We can use the blackbox for B multiple times. For example, if f is uncomputable then it must be that $NOT f$ is uncomputable as well. We can prove this by contradiction. If $NOT f$ was computable, we could use it to solve f by taking its opposite. However, this is a contradiction because we know f is uncomputable. Therefore, $NOT f$ must uncomputable as well.

KARP Reductions

In KARP reductions, we reduce problem A to problem B by transforming the inputs for A to inputs for B. Specifically, we use function $R : \{\text{inputs for A}\} \to \{\text{inputs for B}\}$.

There are three parts for KARP Reduction.

1. Specification (What): R
2. Utility (Why): $\forall x A(x) = B(R(x))$
    - Therefore, if we can solve B, then we must be able to solve A by giving B the transformed input to A. This leads to a contradiction when we know that A is uncomputable. So in general, to prove if a problem B is uncomputable we reduce from an uncomputable problem A to problem B.
3. Implementation (How): Need to make sure R is computable

Consider a new function:

$$\text{HALTONZERO} : \{0,1\}^* \to \{0,1\}$$
$$\text{HALTONZERO}(\langle M \rangle) = \begin{cases} 1 & \text{if M halts on input 0} \\ 0 & \text{if M does not halt on input 0} \end{cases}$$

We can create a reduction function that maps inputs (M, x) to a turing machine described by the following pseudocode:

```
def N(z):
    res = EVAL(<M>, x)
    return res
```

This reduction function R returns a turing machine that computes the output of the turing machine that is given to it as input.

We can easily show that $\forall x \text{HALT}(M, x) = \text{HALTONZERO}(R(M, x))$. Note that the input to TM N is ignored.

- If M halts on x, then N also halts on all inputs (including 0) so HALT(M, x) = HALTONZERO(N) = 1
- If M does not halt on x, then N does not halt on all inputs (including 0) so HALT(M, x) = HALTONZERO(N) = 0

We have succesfully shown the reduction holds. So since, HALT is uncomputable we know HALTONZERO must also be uncomputable.

Example

Consider the following function:

$$\text{NOTEMPTY} : \{0,1\} \to \{0,1\}$$
$$\text{NOTEMPTY}(\langle M\rangle) = \begin{cases} 1 & \text{if there is an x such that M(x) = 1} \\ 0 & \text{else} \end{cases}$$

Essentially, NOTEMPTY is 1 if the language is not empty and 0 otherwise. We want to prove that it is not computable via a reduction. We will reduce from HALTONZERO. Consider an R that returns the following program:

```
def N(z):
    EVAL(<M>, 0)
    return 1
```

Let's consider the cases:

1. If HALTONZERO($\langle M\rangle$) = 1, then N will return 1 on all inputs so the language is not empty and NOTEMPTY($\langle N\rangle$) = 1
2. If HALTONZERO($\langle M\rangle$) = 0, then N will return 1 on no inputs so the language is empty and NOTEMPTY($\langle N\rangle$) = 0

6.4 Semantics

A very important concept in Computer Science is software verification. We want to ensure we have created a program that works as expected in all cases. Another way to think of this is to check if a program A is equivalent to another program B for all possible inputs. This would be an amazing tool to have for software testing, computer security, and programming languages. To investigate if this is possible, we introduce several new ideas

Definition Semantic Functions/Properties

For an F which takes TM descriptions as input, F is semantic if for all equivalent programs M and M' (i.e. programs that are equal on all inputs), $F(\langle M\rangle) = F(\langle M'\rangle)$.

Another way to think about it is that a function is semantic if it only depends on the input/output behaviour of the TM, rather than implementation details.

Example

Semantic functions:

- HALTONZERO
- ISMAJORITY (1 if M is equivalent to MAJORITY)

Non-semantic functions:

- F(<M>) = 1 iff <M> ends with a 1
- F(<M>) = 1 iff <M> halts in in 100 steps on 0 (this depends on implementation)
- TODD(<M>)

Theorem Rice's Theorem (1951)

Every non-trivial semantic function/property is uncomputable. Trivial functions are the constant 0 or 1 function on all input.

Proof

Consider F, a nontrivial semantic property, and the program INF(z) which just runs an infinite loop. There are two cases: $F(\langle INF\rangle) = 0$ and $F(\langle INF\rangle) = 1$. We will consider the first case here.

We want to reduce from HALTONZERO to F. We use an R which returns the following program:

```
def N(z):
    EVAL(<M>, 0)
    EVAL(<M_1>, z)
```

Two cases:

- If HALTONZERO($\langle M\rangle$) = 1, then N is equivalent to M so F($\langle N\rangle$) = F($\langle M_1\rangle$) = 1.
- If HALTONZERO($\langle M\rangle$) = 0, then N is equivalent to INF so F($\langle N\rangle$) = F($\langle M_1\rangle$) = 0.

We have shown HALTONZERO reduces to F, so F (a general semantic property) is uncomputable.

We follow as similar idea when $F(\langle INF\rangle) = 1$ except we just use NOTF.

■

7 | Context Free Grammars

We saw that Turing Machines are able to compute any computable function. However, they have a significant limiitation in that they cannot reason about themselves in meaningful ways. Even further, it can sometimes be considered a bug to be Turing Complete, as seen in the case of the Ethereum Contract language Solidity. Hackers can take advantage of Turing Completeness! For these reasons, sometimes a more limited model of computation is desirable. Context Free Grammars are one such model. Compared to DFA's and TM's, CFG compututational power ranks as follows:

$$\text{DFA} < \text{CFG} < \text{TM}$$

Definition

We define a grammar G, with variables V, terminal symbols Σ, rules R, and start variables S. V is disjoint from Σ.

$$G = (V, \Sigma, R, S)$$

Rules are defined as follows. They go from variables to terminal symbols and other variables.

$$R : V \to \Sigma \cup V$$

Consider a grammar, $G_1 : (\{A\}, \{0, 1\}, R, A)$ with the following rules

$$A \to 0A1$$
$$A \to \varepsilon$$

We can **derive** a string based on our rules. Consider $\alpha \in (\Sigma \cup V)^*, \beta \in (\Sigma \cup V)^*$. We say $\alpha \underset{G}{\Longrightarrow} \beta$ if we can get β by replacing a variable in α using a rule from R.

The following are valid derivations for G_1.

- $A \underset{G_1}{\Longrightarrow} 0A1$
- $0A \underset{G_1}{\Longrightarrow} 00A1$
- $00A1 \underset{G_1}{\Longrightarrow} 000A1$
- $00A1 \underset{G_1}{\Longrightarrow} 001$

- $AA0 \underset{G_1}{\Longrightarrow} 0A1A0$
- $AA0 \underset{G_1}{\Longrightarrow} A0A10$

We say $\alpha \underset{G_1}{\Longrightarrow}^* \beta$ if there exists a **chain of derivations** to get β from α. In otherwords, there exists some $\alpha_1, \alpha_2, ..., \alpha_{R-1}$ such that $\alpha \underset{G}{\Longrightarrow} \alpha_1 \underset{G}{\Longrightarrow} \alpha_2 \underset{G}{\Longrightarrow} \alpha_3 \underset{G}{\Longrightarrow} ... \underset{G}{\Longrightarrow} \alpha_{R-1} \underset{G}{\Longrightarrow} \beta$.

Definition

If G is a grammar, $x \in \Sigma^*$ is matched to G if $S \underset{G}{\Longrightarrow}^* x$. We say $L_G = \{x \in \Sigma^* : \text{x matches G}\}$ and $F_G : \Sigma^* \to \{0,1\}, F_G(x) = 1$ iff x matches G.

We say a function $f : \Sigma^* \to \{0,1\}^*$ is **context free** if there exists a grammar G such that $f(x) = F_G(x) \forall x$.

Example

$G_1 : (\{A\}, \{0,1\}, R, A)$ with the following rules

$$A \to 0A1$$
$$A \to \varepsilon$$

What is the language L_{G_1}? Answer: It's $0^n 1^n$ for $n \geq 0$. Since the starting variable is A, we can only ever add two bits at time (a 0 and a 1). So there must be an equal number of 0's and 1's.

Recall, we saw earlier that this is not a regular language. So context free grammars must be more powerful than DFA's.

Example $G_2 : (\{A\}, \{0,1\}, R, A)$ with the following rules

$$A \to \text{0A0} \mid \text{1A1} \mid 0 \mid 1 \mid \varepsilon$$

G_2 generates the language that accepts all palindromes.

Example $G_3 : (\{S\}, \{\text{“(”}, \text{“)”}\}, R, S)$

$$S \rightarrow \text{(S)} \mid \text{SS} \mid \varepsilon$$

This generates the language with all well matched parenthesis expressions.

Why CFG's?

All programming languages use CFG's for syntax checking. For every programming language there exists a grammar that generates syntactically valid programs.

We saw before that non regular languages can be context free. However, we can take this a step further and claim that CFG's are *strictly* more powerful than DFA's.

Theorem

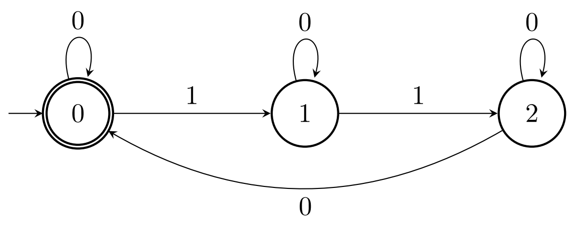
Every regular language is also context free. For every DFA D, there exists a CFG G such that $D(x) = F_G(x) \forall x$.

Proof

We can prove this by construction. The goal is a grammar that emulates the behaviour of the DFA i.e. with the same input, the function output is the same.

- Create a variable for each state
- The starting variable is V_0, the first state
- For each transition $T(i, a) = j$, add a rule $V_i \rightarrow aV_j$
- For each accepting state i, add $V_i \rightarrow \varepsilon$

Example



For the equivalent grammar, $\Sigma = \{0, 1\}, V = \{V_0, V_1, V_2\}, S = V_0$.

Rules:

$$V_0 \rightarrow 0V_0|1V_1|\varepsilon$$
$$V_1 \rightarrow 0V_1|1V_2$$
$$V_2 \rightarrow 0V_2|1V_0$$

To go from NFA's to CFG's we perform a similar construction. Every transition has a rule with the seen bit preceeding the next state (including epsilon transitions, where the seen bit is the empty string).

7.1 Designing CFG's

When thinking about how to design a CFG that matches a language the following tips can be helpful

1. Break up the language into simpler pieces and take the union
   - Union can be performed by adding a new start symbol and new rule $S \rightarrow S_1|S_2$
2. CFG's can link strings
3. We can convert regular expressions to CFG's with the construction discussed before

Example

Consider the following languages. Design CFG's that match them.

- ADD $= 0^m10^n10^{m+n}$
- L $= \{x : x$ is of even length, middle two symbols are the same$\}$
- L $= \{x : x$ is of odd length and first, middle, last symbols are the same$\}$

Solutions in lecture 16 class notes.