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

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

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

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. $D : \{0,1\}^* \to O$ and $D(E(x)) = x$

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

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

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

- 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 $\overline{ZtoB} : Z \to \{0,1\}^*$ 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: $\{0,1\}^3 \rightarrow \{0,1\}$

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

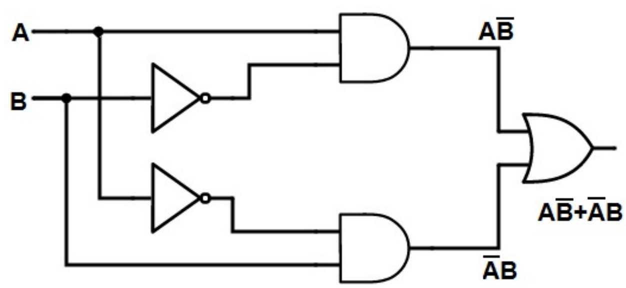
Boolean implementation of XOR3:

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

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

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:



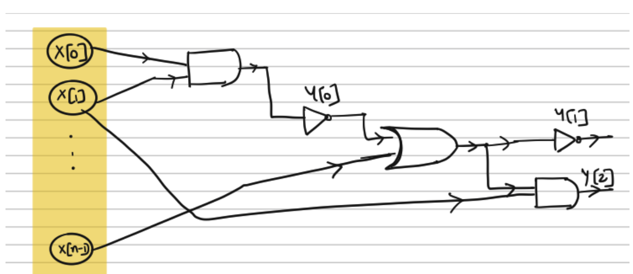
3 | Boolean Circuits

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 $\to$ 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?

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

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
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.

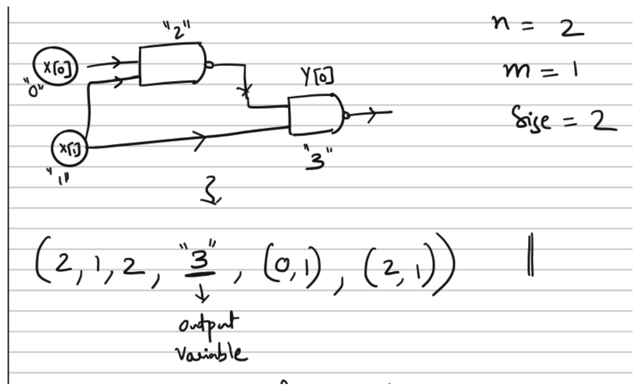
What do we need to specify in a NAND circuit?

- 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:

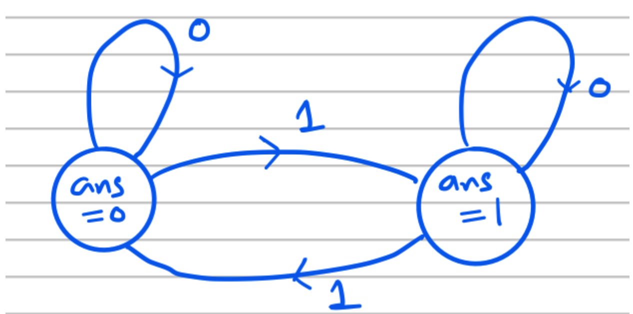
$$\text{XOR} : \{0,1\}^* \to \{0,1\}$$

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

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)

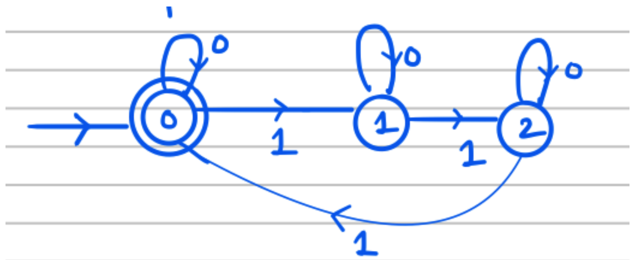
DFA with c states over {0, 1} is a pair $D \equiv (T, S)$, where $T : [c] \times \{0,1\} \to [c]$ and $S \subseteq [c]$. T is known as the transition function and defines the inputs that cause a transition in state. For example, if the current state is 0 and the input bit is 1, the transition function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



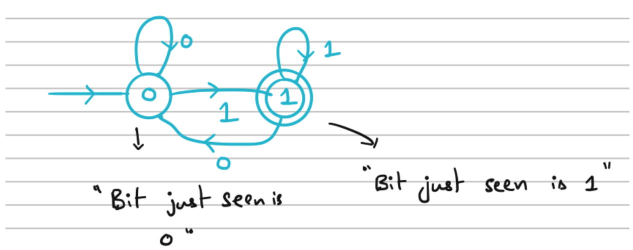
The transition function T is shown below and S, the set of accepting states is $\{0\}$. If we wanted to modify the function be 1 if number of 1's mod 3 was 1, we can simply modify S to $\{1\}$.

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:

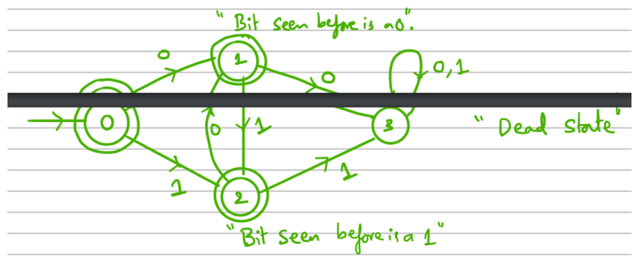


Example

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

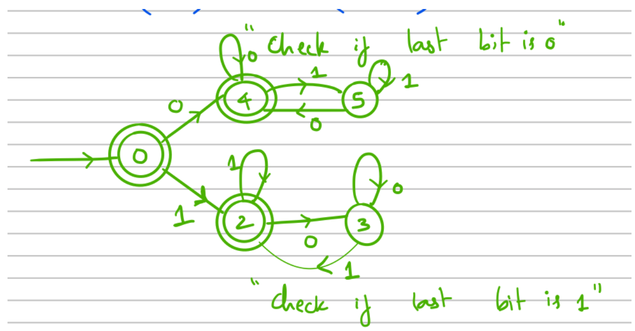
For example, f(01010) = f(10101) = 1 while f(01001) = 0.

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

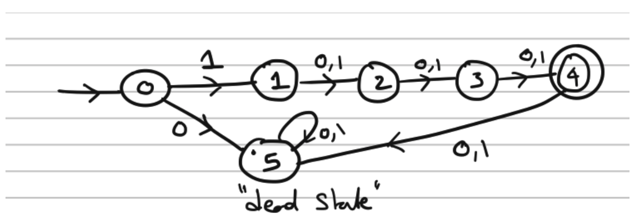
To begin discussion of NFA's, we will first discuss function concatenations. For convenience, the definition is repeated below:

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).



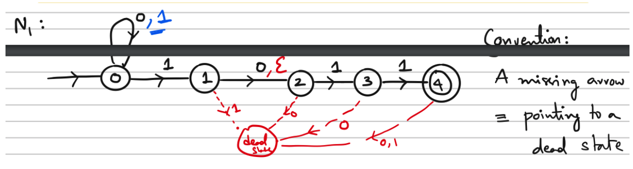
Can we compute the concatenation of these two functions using a DFA? In fact, we can. Using just 16 states, we can use a DFA to compute it. To clarify what exactly we're computing, $f_1 \circ f_2(001010) = 1$ since $f_1(00) = 1$ and $f_2(1010) = 1$.

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.

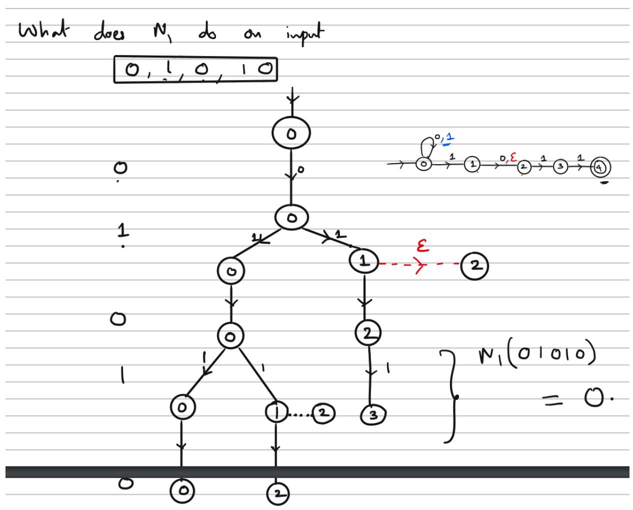
Non-Deterministic Finite Automata



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).

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

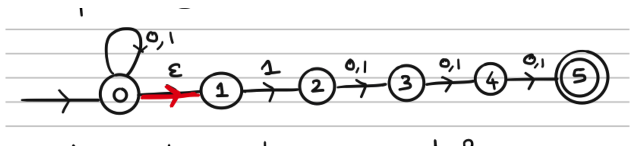
$$S \subseteq [c] \tag{5.2}$$

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

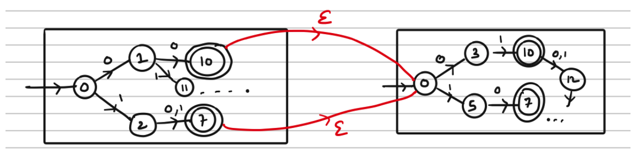
In other words, the transition function is mapping the current state and the current bit input to a subset of all states in the NFA.

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:



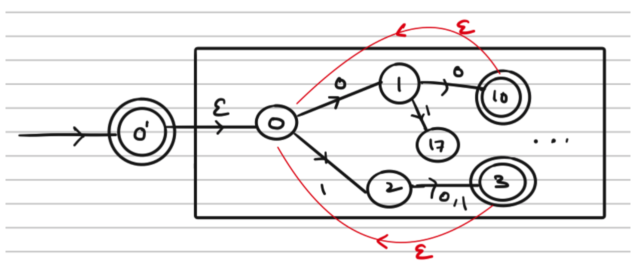
Important Takeaway: In general, if two functions are computable by DFA's, we can compute their concatentation using an NFA in which we simply add *varepsilon* transitions pointing from accepting states of the first function to the start of the second.



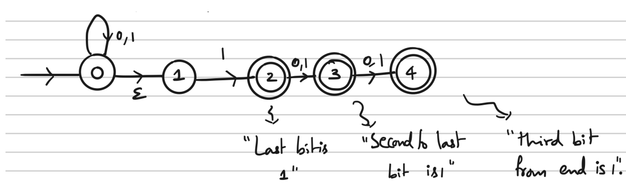
Kleen* Operation on Functions

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?

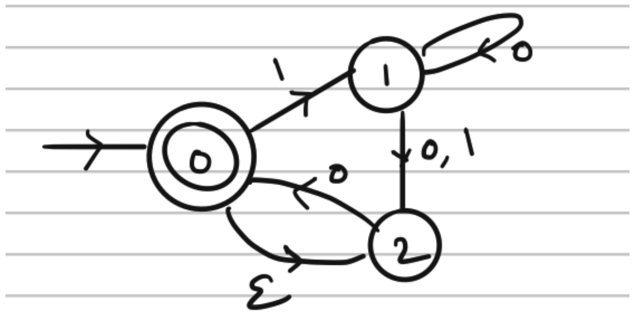
Definition Eps(I) = all states reachable by taking *varepsilon* edges from i + {i}

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

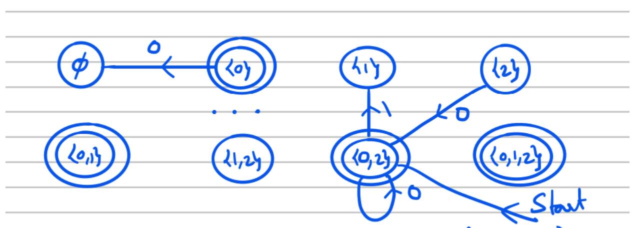
$$T_D(I, a) = Eps(\bigcup_{i \in I} T_N(i, a))$$

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):

- $r_1 r_2$ (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:

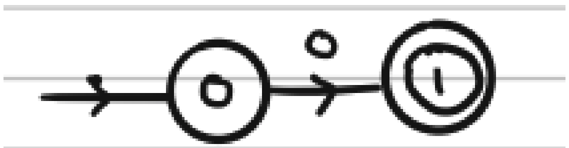
- convert the regex to an equivelant NFA N with O(m) states and transitions in O(m) time
- simulate NFA N on the input X in O(m * n) time

Proof By construction,

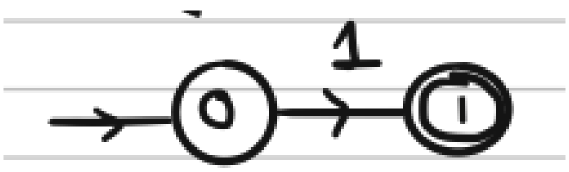
We defined regex inductively, so we will alos build the NFA for a regex inductively.

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

Suppose L was regular. Then there must exist p for which PL holds. Consider x = $0^p 1 0^p$. 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,

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