TheoryPrep **(Due: 6/28/20)**

Assignment #5: Definitions and Proofs

Name: Student name(s)

We've taken some problems from the course texts and CS 121 material. Some problems are marked as (Challenge) or (Additional Problems). Do the rest first, and if you have time, return to these problems. It's **much more important** to have a rigorous understanding of the core problems and how to prove them than to simply finish all the additional problems.

This pset is different from previous weeks. In particular, there will be substantially more problems asking about intuition or interpretation. This is because as we've learned in the videos for this week, an important part of understanding definitions and proofs is understanding their spirit. A good definition is one that captures our intuition effectively and efficiently (ie with not too much complexity or additional constructions), which involves more aspects of design–this is one way in which math can be beautiful!

Problem 1: Understanding Definitions

Learning goal: Definitions are the fundamental building block of mathematics, and being able to develop intuitive understanding for definitions is critical. Especially in CS 121, the course contains many complex definitions, and building intuition for those definitions is integral to succeeding in the course.

(a) Let's make sure we understand the definition of a boolean circuit (and computing a boolean circuit). For each of these questions, you need only explain in English your reasoning–no need for a formal proof.

1. Why does it make sense for n of the vertices in a boolean circuit to have no in-neighbors?
2. The definition of boolean circuits requires $s \geq m$, or in English that the number of gates is at least as many as the number of outputs. Why do we need to require this?
3. Why is it important that a boolean circuit is a DAG?
4. Definition 3.6 computes a the output of a boolean circuit based on a topological sort of the vertices. Why is it important that we proceed in the order prescribed by topological sort? How does this relate to the condition in AON-CIRC programs that the variables on the righthand side of each line must be either inputs or from previous lines?
5. Note that there is no restriction on the number of outgoing edges from a gate. This means a gate could send its signal to two other gates. Recalling the physical implementations of the computers from the reading, provide an example of a physical model that agrees with our choice to allow a gate output to go to multiple gates and an example which would disagree with this choice.

,

Solution:

1. Those vertices represent the inputs to the boolean circuit. In a boolean circuit, in-neighbors represent inputs to a boolean gate, but the n vertices (the inputs) with no in-neighbors represent the inputs to the program, not boolean gates.
2. Every output is a gate (since an output comes from doing something on inputs–if you just returned an input you don't need an actual program). Therefore there must be as many outputs as gates.
3. A DAG is defined by two features: directed and acyclic. The graph is directed since gates are each sending information in a direction to the next gate. You can't run an AND gate in reverse! It is also important that the graph is acyclic, since

each gate is reliant on all gates preceeding it. If there was a cycle in the graph, we couldn't evaluate any of the gates in the cycle since they all depend on each other!

4. In a topological sort, each vertex is listed before its descendants. In Definition 3.6 when we are trying to evaluate a circuit, we need to make sure that when we evaluate a gate, we have evaluated all gates before it. Topological sort ensures that every gate that goes into a gate v is evaluated before it, since they all have v as a descendant. In an AON-CIRC program, the requirement that the variables on the righthand side must be from previous lines (or inputs) represents the exact same thing–the inputs to each boolean operation are calculated earlier in the algorithm!

5. A water-based transistor (figure 3.15 in the reading) would match our definition, since we could just split up the water flow and send a bit of water down each outgoing pipe. A marble-based computer would not match our definition since we only emit one outgoing marble.

   Note that the particular physical examples a student picks are not the important part (for example, a student could point out that. by adding additional marbles at the output of a gate, it matches our definition)–the important takeaway is that the decision to allow multiple outgoing edges from a gate is an important one that corresponds to certain ways to construct a computer but not others (echoing the larger takeaway from this week that definitions are created to formalize concepts and intuition).

(b) A good definition should be so intuitive/natural that you might have come up with by yourself! We'll make a definition for running a NAND-CIRC program. To review, in section 3.5.3, the textbook defines the NAND-CIRC Programming language similarly to the AON-CIRC programming language, so every line is of the form

$$foo = NAND(bar, blah) \tag{0.1}$$

Define what it means for a NAND-CIRC program P to compute some function $f : \{0,1\}^n \to \{0,1\}^m$

Note: After you do this problem, delete the first red box on the last page of the reading. Compare your definition to the one provided. How is it similar and/or different? Which do you prefer and why?

Solution:
(from the book) Let $f : \{0,1\}^n \to \{0,1\}^m$ be a function and P and NAND-CIRC program. We say that P computes f if

1. P has n input variables $x[0], ..., x[n-1]$ and m output variables $y[0], ..., y[m-1]$.
2. If we assign each $x[0], ..., x[n-1]$ the values $x_0, ..., x_{n-1}$ and execute P in the same way, then we will get $y[0]|...|y[m-1] = f(x)$ (where | means concatenation of bits).

Possible insights students may have when comparing their definitions are that they were not quit formal/rigorous enough. An example of this would be referring to inputs and outputs in the abstract instead of defining them in terms of x and y.

(c) Let's make sure the definition of a Turing Machine makes sense as well. Again, for the below questions, explain your reasoning in English.

1. Why can't we define a Turing Machine as anything that python can compute? This is a real question–take some time to think about it!
2. Recalling that our motivation for a turing machine is a human computer, what does the transition function δ_M correspond to? What does the tape correspond to?
3. Your friend complains that we don't need an $\oslash$ since we can just use 0. Suppose that we replaced $\oslash$ with 0 in the definition. Would this still be a satisfactory definition? Why or why not?

4. Your friend notes that a human computer sometimes starts in the middle of the paper and therefore suggests that we should initialize $i = 10$ in the definition of a Turing Machine. Is this an equivalent definition? Does it make sense?

5. (Additional Problem) Your friend then observes that we are representing states as $[k]$, the set $\{0, 1, ..., k-1\}$, but when a human is doing computation it's more apt to store their memory as a string (wow this friend is really interrogating the definition–what a good theoretical computer scientist!). Therefore, your friend suggests representing states as A^m where $A = \{a, b, c, ..., z, \text{“ ”}\}$ (so alphabetical strings of length m). This means we'd change the definition to be below (note the only changes are $\delta_M :: A^m \times \Sigma \to A^m \times \Sigma \times \{L, R, S, H\}$ and $s =$ “”, but we copy the rest of the definition for convenience)

   A (one tape) Turing Machine with alphabet $\Sigma \supseteq \{0, 1, \triangleright, \oslash\}$ is represented by a transition function $\delta_M : A^m \times \Sigma \to A^m \times \Sigma \times \{L, R, S, H\}$.

   For every $x \in \{0,1\}^*$, the output of M on input x, denoted by $M(x)$ is the result of the following process

   - We initialize T to be the sequence $\triangleright, x_0, x_1, \ldots, x_{n-1}, \oslash, \ldots$ where $n = |x|$.
   - We initialize $i = 0, s =$ “” (so the initial state is the empty string)
   - We then repeat the following process
     (a) Let $(s', \sigma', D) = \delta_M(s, T[i])$.
     (b) Set $s \to s', T[i] \to \sigma'$.
     (c) If $D = R$ then set $i \to i + 1$ If $D = L$ then set $i \to \max(i - 1, 0)$. If $D = H$ then halt.
   - If the process above halts, then M's output, denoted by $M(x)$ is the string $y \in \{0,1\}^*$ obtained by concatenating all the symbols in $\{0, 1\}$ in positions $T[0], \ldots, T[i]$ where i is the final head position. If the process does not halt then we write $M(x) = \perp$

   Is this an equivalent definition?

Solution:

1. How do you define what python can do? This is actually an extremely complex question–even the reference for a lower level language like C is incredibly long. The definition of a python program would be substantially more complicated and difficult to work with. As you will learn in CS 121, python and Turing Machines are equivalent in power, but . There's not a singular correct answer to this question– it's more to get you thinking about why the definition of a Turing Machine is necessary.

2. The transition function mimics the human brain reading symbols, updating its memory, writing and moving around according to a set of internal rules. The tape corresponds to the scratch paper of the human computer–we assume it's infinite which is the same as assuming the human computer has infinite space to write things down.

3. This would not be a satisfactory definition. Consider if the turing machine is supposed to return 1 if and only if the length of the output is odd. Under our standard definition, we can certainly calculate this (think of a turing machine that reads the input symbols with internal state “odd” or “even”‘ and when it reads $\oslash$ writes a 1 or 0 respectively to the beginning of the tape and halts). However, if we replace all the $\oslash$ with 0 then a single 0 of input looks the same as 00, so the turing machine cannot distinguish between them. The student may provide any sufficient rationale and example for why the definition is unsatisfactory.

4. This is technically an identical definition, but it is much more inconvenient. Let's call our new construction that initializes $i = 10$ to be an “Offset Turing Machine.” To say these definitions are equivalent means that we can compute some function f with a Turing Machine if and only if we can compute it with an Offset Turing Machine. If we can compute f with a normal Turing Machine, we can turn that

into an Offset Turing Machine by adding to the transition rule that if it's in state 0 and not at $i = 0$ then it moves left until it sees $\triangleright$ and then runs the Offset Turnig Machine (if 0 was a relevant states, just add more states and change their meaning). If we can compute f with an Offset Turing Machine, we can turn that into a Turing Machine by having it move right and increment state until it gets to state 10 (and if states < 10 are used just add more states).

5. This is an equivalent definition. Having A^m as states is just like having $[27^m]$ as states; in order to transform a turing machine with alphabetical states into a turing machine with numerical states, just use $[27^m]$ states and map each of them to an alphabetical string–you can prepend and postpend this bijection on the original transition function (and to transform a turing machine with $[k]$ numerical states into alphabetical states similarly use $A^{\lceil \log_{27}(k) \rceil}$ as states and map each numerical state to an alphabetical string–you may have extra alphabetical states so just don't use those). The key insight is it doesn't matter how we label the states–numbering them or giving them alphabetical labels does not change their meaning.

Problem 2: Reading Proofs

Learning goal: You read a number of longer proofs for this week, and that's an important skill. This section will help you make sure you understood the proofs well by asking you to give intuition and evaluate if these proofs would work for slightly different situations.

(a) An important part of understanding proofs is understanding them intuitively. For each of the longer proofs from listed below, summarize the proof strategy in 1 to 2 English sentences.

- Equivalence of Boolean Circuit and AON-CIRC.
- Univerality of NAND
- (Additional Problem) Proof that there does not exist a surjection $f : \mathbb{N} \to \mathbb{P}(\mathbb{N})$ from week 3.
- (Additional Problem) Proof of Hall's Theorem (necessary and sufficient condition for matchings) from week 4.

Solution:

- We can replace gates with lines of code or vice versa, so AON-CIRC and boolean circuits are just two different ways of storing algorithms. We'll construct a substitution method to translate between the two mediums.
- Any finite function on n input bits is just looking up a bit in a 2^n length string, and we can do this by inductively searching each bit in a lookup.
- We can always find a binary string not in the image of a map from because we can make miss on the corresponding input

(b) Recall from the second video the proof of universality of NAND. Suppose we tried to use the same proof to prove the universality of IF. In particular, suppose we used the same proof (with some minor changes) to prove the below theorem (let IF-CIRC programs be composed of lines of the form $foo = IF(bar, blah, zoo)$ that are evaluated similarly to NAND-CIRC by evaluating each line in succession, but setting $foo = blah$ if $bar = 1$ and otherwise setting $foo = zoo$).

Theorem 1 *There exists some constant $c > 0$ such that $\forall n, m > 0$ and $f : \{0,1\}^n \to \{0,1\}^m$ there is an IF-CIRC program of at most $cm2^n$ lines that computes f.*

Would this work? If so, write the proof. If not, explain where the proof fails and then propose a small modification (ie added features) you could make to IF-CIRC to make it universal.

Solution:
This proof will not work for IF-CIRC. The proof of universality in the second video has two parts. First we prove that $LOOKUP_k$ is computable efficiently and then prove that we can use $LOOKUP_k$ with hardwired inputs (hardwired refers to the fact that instead of letting the user specify these inputs, we set their values in the program–so in this case we're seeting them to $f(x)$ if the program computes f) to compute any finite function. We can do the first since we can construct $LOOKUP_k$ with at most 2^k gates (following a similar inductive proof as given in the video). However, we cannot do the second step. Intuitively, this is because IF-CIRC cannot set a varaible to be 0 or 1 so it cannot hardwire the inputs (recall in NAND-CIRC, we can set $TEMP = NAND(X[0], X[0])$ and then $NAND(TEMP, X[0])$ gives us 1 and then $NAND(1, 1)$ gives us 0). Thus, this proof will not work. We could make this work by adding the "functions" $ZERO()$ and $ONE()$ to the IF-CIRC programming language that return 0 and 1 respectively.

Students may point out that just because this proof doesn't work doesn't imply that the theorem is false (and if they don't ask, you may want to bring this up as a salient point for discussion). That's a super valid point–the reason this question only concerns the proof presented in the video works for IF-CIRC is to help students make sure they really understood what was going on in the proof (in other words, understanding the proof is the competency that this question emphasizes, not trying to derive a new result about IF-CIRC). However, asking if IF-CIRC is universal is a good question to poset to students as well (you can see it's not because it cannot compute the zero function since if it gets all ones as inputs it can't output 0 since an IF gate can only output one of its inputs–to be rigorous we could apply this argument inductively).

(c) Note: Before you do this problem, make sure you've done Problem 1.b. You should delete the first red box on the last page of the reading to reveal the definition of how to compute the output of a NAND-CIRC program. You should use this definition.

The textbook provides a proof to demonstrate that a function is computable by a boolean circuit if and only if it's computable by an AON-CIRCUIT program of similar length (Theorem 3.9). Suppose we tried to use the same proof to prove that a function is computable by a NAND circuit if and only if it's computable by NAND-CIRC program of similar length. In particular, let's say we were trying to prove the below theorem.

Theorem 2 *Let $f : \{0,1\}^n \to \{0,1\}^m$ a function and $s \geq m$ some natural number. Then f is computable by a NAND circuit of s gates if and only if it's computable by an NAND-CIRC progam of s lines.*

Would this work? If so, write the proof. If not, explain where the proof fails.

Solution:
This proof will work for NAND circuits and NAND-CIRC programming language. Intuitively this is because all we require is that we're able to substitute in lines in one language for gates in the other and we can also do that with NAND. The full proof is below and is very similar to the AON-CIRC/boolean circuits proof.

We need to prove both directions. First we'll prove if a function is computable in NAND circuits of s gates then it is computable by a NAND-CIRC program of s lines. Fix a function f computable by a NAND circuit of s gates, and let C be a circuit that computes f. Take the vertices of C ain a topologically sorted order. In this order, for the ith gate, it will have two in-neighbors–add the line $i = NAND(a, b)$ where a, b are either the gate numbers of the in-neighbors or $x[j]$ if the in-neighbor is an input to the program. Note that the in-neighbors of gate i must be either inputs or previously calculated gates in the topological sort, so this is a valid program. Now check for each ith gate to see if it labeled with $y[j]$ and if so, replace all instances of i with $y[j]$. We see that the ith line of the program will agree with the ith gate by construction since it has the same inputs, so the constructed program and C both compute f (a rigorous proof can be given by induction, but it is clear by inspection in this case).

Now we'll prove that if a function is computable by a NAND-CIRC program of s lines then it is also computable by a NAND circuit of s gates. Fix f an arbitrary function

computable by a NAND-CIRC program of s lines, so we denote by P the program computing f. We will now construct a NAND circuit C of s gates that computes f. Create n vertices wiith no in-neighbors as the inputs for C. Now go down the lines in the program and for the ith line $foo = NAND(bar, blah)$, create a vertex called i. We know that *bar* and *blah* must either be input variables or assigned on previous lines of the program by the validity of P, so put ingoing edges from the respective inputs or numbered nodes for those lines into i (for concreteness, if *bar* is the fourth bit of the input and *blah* comes from the 8th line, then put edges $(8, i)$ and $(x[3], i)$. Now for each line j that assigns an output to $y[k]$, label node j with $y[k]$. We now see that by construction the gate labeled i must have the same value as the ith line of the program, and therefore they compute the same function (again, a rigorous proof can be given by induction, but it is clear by inspection in this case–also we're going to use a very similar inductive strategy in 3.b so no need to repeat it here).

(d) (Additional Problem) You're trying to implement NAND circuits in real life with marble computers (see page 21 of the reading for a refresher), but because each gate only shoots out one marble, you realize you can only have one output coming out of each gate. You decide to model this by creating a type of circuits called "marble NAND circuits" that are the same as NAND circuits, except each gate may only have one outgoing edge (the input vertices may have any number of outgoing edges). This captures the constraint that a marble gate can only output one marble. Your friend remarks that this is totally fine since you can be just as efficient even if each gate in your circuit has a singular output. In particular, they claim the following theorem holds by a slightly modified proof of Theorem 3.9.

Theorem 3 *Let $f : \{0,1\}^n \to \{0,1\}^m$ a function and $s \geq m$ some natural number. Then f is computable by a NAND circuit of s gates if and only if it's computable by a marble NAND circuit of at most s gates.*

Would modifying Theorem 3.9 work to prove this? If so, give the proof. If not, explain where the proof fails, and give a weaker result that would hold (recall that a weaker result is something that would be implied by Theorem 3). If you give a weaker result try to either prove it or explain how you would prove it.

Solution:
Modifying the proof of theorem 3.9 will not work. The proof of Theorem 3.9 relies on replaceing gates a NAND circuit with gates in a marble NAND circuit, but if the gates in a NAND circuit have multiple outgoing edges, this does not lead to a valid marble NAND circuit.

The curious student may point out again that the failure of this proof technique does not imply the statement is false (again, we choose to focus on asking question of the form "would this proof work for this other construct" primarily to test our understanding of the proof not to develop new theorems). The theorem is false, and below is an example (if time permits this may be instructive to discuss with students). Below is a diagram of a NAND circuit to calculate the AND function

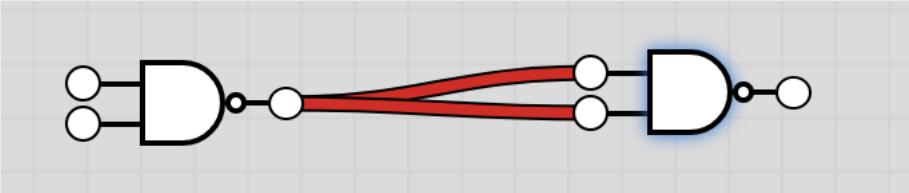

Figure 1: A NAND Circuit to calculate AND

I will prove by exhausition that there is no 2 gate NAND marble circuit to calculate AND.

1. Case 1: The first gate takes in $x[0], x[0]$ (without loss of generality this also covers the case where it takes in $x[1], x[1]$). The second gate can either take in $x[0], x[1]$ (in which case it calculated NAND not AND), or the output of the first gate and $x[0]$ in which case it disagrees with AND on input 01, or the output of the first

gate and $x[1]$ in which case it disagrees with AND on input 10.

2. Case 2: The first gate takes in $x[0], x[1]$. Then the second gate can take in either $x[0], x[1]$ (in which case it calculates NAND not AND), or the output of the first gate and $x[0]$ (or $x[1]$ which is the same without loss of generality) in which case it disagrees with AND on input 01.

Thus we see that a NAND circuit can calculate AND in 2 gates but a NAND marble circuit cannot.

The weaker theorem below will hold

Theorem 4 *Let $f : \{0,1\}^n \to \{0,1\}^m$ a fuinction and $s \geq m$ some natural number. Then f is computable by a NAND circuit if and only if it's computable by a marble NAND circuit.*

The cause for alarm in the original theorem is that it required that both methods of computation would be equally efficient, but if you can reuse the outputs from gates, you will necessarily be more efficient. To prove the above theorem, you need to prove that something computable by a marble NAND circuit is computable by a NAND circuit and vice versa. The former follows because any marble NAND circuit is also a NAND circuit.

The latter is a bit more complex. Suppose that we have a NAND circuit C of s gates that computes some function f. We will start with the graph for C and make it so that every gate has at most one outgoing edge, which will make it a valid marble NAND circuit. Organize the vertices in topological sort order $v_0, ..., v_s$ (note that technically a topological sort contains all vertices, but we just drop the input vertices). Starting with v_s and moving up the list, if the vertex v_i has multiple outgoing edges, we will add another vertex for each outgoing edge after the first, label all of these new vertices v'_i, put edges from each parent of v_i to each new v'_i, and split up the outgoing edges from v_i such that each v'_i now takes one of the additional outgoing edges from v_i (so v_i and all the v'_i all have one outgoing edges). Importantly, since we are preceding in reverse topological sort order, each new copy v'_i we generate will only generate outgoing edges on vertices v_j for $j < i$. Thus, by the end of this process, each v_i or v'_i will only have one outgoing edge. The last thing to verify is that this marble NAND circuit calculates the same thing. We can proceed by induction. I claim that upon evaluation on some x, $P(k)$, which is the predicate that the v_k and v'_k in the new circuit are equal to v_k in the original circuit (which we will call u_k for clairty), holds. The base case for $k = 0$ is true since by construction both v_k and v'_k are just the NAND of the two inputs of u_k. The inductive step follows because for some $k + 1$ (assuming $P(l)$ for all $l \leq k$), if u_{k+1} has ingoing edges from inputs, so do v_{k+1} and v'_{k+1}. If u_{k+1} has ingoing edges from some v_l for $k \leq k$ (and this is the only possibility since we topologically sorted the edges, so u_{k+1} cannot have an ingoing edge from some u_l for $l \geq k + 1$), then v_{k+1} and v'_{k+1} have ingoing edges from v_l or v'_l, which by induction have the same value as u_l. Thus, v_{k+1}, v'_{k+1} have the same value as u_{k+1}. Thus, the outputs are the same so the new circuit provides a marble NAND circuit computing f.

because you can start with the NAND circuit and copy any gates with multiple outgoing edges such that each copy only has one outgoing edge (the key is that when you copy a gate, it only generates more outgoing edges on gates earlier in the topological sort, so this process will terminate).

(e) (Additional Problem) Sometimes proofs will be very concise and you may have to fill in the details for yourself. Below is a concise proof taken from American Mathematical Monthly, v.116, issue 1, January 2009, page 69. It is proving that $\sqrt{2}$ is irrational (and is slightly different from the commonly-presented proof by contradiction).

Assume $\sqrt{2}$ is rational and choose the least integer $q > 0$ such that $(\sqrt{2} - 1)q$ is a nonnegative integer. Let $q' = (\sqrt{2}-1)q$. Clearly $0 < q' < q$, but we can easily show that $(\sqrt{2}-1)q'$ is a nonnegative integer, contradicting the minimality of q.

Write out a more detailed version of this proof that explains and justifies each step. Comment on which proof you think is better–does this depend on the scenario?

Solution:
Suppose towards a contradiction that $\sqrt{2}$ is rational, so $\sqrt{2}-1$ is rational so we can write $\sqrt{2}-1=\frac{p}{p'}$ for $p,p'\in\mathbb{Z}$ (and to be super rigorous we can ensure p,p' both positive since $\sqrt{2}-1>0$). Clearly $(\sqrt{2}-1)p'=p$ is a nonnegative integer, so there must be a least integer $q>0$ such that $(\sqrt{2}-1)q$ is a nonnegative integer.

Let $q'=(\sqrt{2}-1)q$, so q' is a nonnegative integer less than q (since $\sqrt{2}-1<1$). Then we have

$$\begin{aligned} q'(\sqrt{2}-1) &= q(\sqrt{2}-1)^2 \\ &= q(2-2\sqrt{2}+1) \\ &= q(3-2\sqrt{2}) \\ &= q-2q(\sqrt{2}-1) \end{aligned} \tag{0.2}$$

$2<\frac{9}{4}\Rightarrow\sqrt{2}<\frac{3}{2}\Rightarrow\sqrt{2}-1<\frac{1}{2}$ so we know that $q=q2\frac{1}{2}>q2(\sqrt{2}-1)$ and therefore the fourth line must be >0–it must also be an integer since q is an integer and so is $q(\sqrt{2}-1)$. Therefore q' is both strictly less than q, and $q'(\sqrt{2}-1)$ is a nonnegative integer. This contradicts that q is the smallest positive integer such that $(\sqrt{2}-1)q$ is a nonnegative integer. Hence a contradiction and we see that $\sqrt{2}$ is irrational.

The key takeaway from this exercise is that your audience determines how detailed your proof needs to be. This was published in a journal where most readers will be very experienced mathematicians and therefore it's okay to leave out these details. However, if this was presented in a discrete math class (like this one), we'd certainly think it appropriate to provide more details!

Problem 3: Longer Proofs

Learning goal: This week we will focus on doing one longer proof that combines understanding complex definitions and a couple of proof techniques. This will likely be one of the longest proofs you've done in the course, and it will hopefully help you grow your complex proof-writing skills.

(a) Let XOR-CIRC be the programming language where each line is of the form $foo = XOR(bar, blah)$, $foo = 1$, or $foo = 0$ (so each line can take one of those three forms). Compare the power of XOR-CIRC and NAND-CIRC. In particular, you should prove one of the following three things: NAND-CIRC and XOR-CIRC are equally powerful, so any function computable in one is computable in the other; NAND-CIRC is more powerful than XOR-CIRC so everything computable in XOR-CIRC is computable in NAND-CIRC, but there exists a function computable in NAND-CIRC not computable in XOR-CIR; XOR-CIRC is more powerful than NAND-CIRC so everything computable in NAND-CIRC is computable in XOR-CIRC, but there exists a function computable in XOR-CIRC not computable in XOR-CIR; or neither is more powerful, so there exist functions computable in each programming language not computable in the other.

It may be helpful to write a proof outline for your own proof first.

Solution:
I will prove that NAND-CIRC is more powerful than XOR-CIRC. First I will show that any function computable by XOR-CIRC is computable by NAND-CIRC. I will give a way to calculate $foo = XOR(bar, blah)$, $foo = 1$, and $foo = 0$ in NAND, and this implies that if function f is computable by a program P in XOR-CIRC, the program Q in NAND-CIRC that result from substituting each line of the XOR-CIRC program calculates the same thing and therefore f is computable in NAND-CIRC.

- $foo = XOR(bar, blah)$ can be calculated by the following lines in NAND-CIRC.

$$\begin{aligned} temp &= NAND(bar, blah) \\ temp2 &= NAND(bar, temp) \\ temp3 &= NAND(blah, temp) \\ foo &= NAND(temp2, temp3) \end{aligned} \tag{0.3}$$

- $foo = 1$ can be calulated by the following two lines of NAND-CIRC

$$\begin{aligned} temp &= NAND(x[0], x[0]) \\ foo &= NAND(temp, x[0]) \end{aligned} \tag{0.4}$$

- $foo = 0$ can be calulated by the following three lines of NAND-CIRC

$$\begin{aligned} temp &= NAND(x[0], x[0]) \\ temp2 &= NAND(temp, x[0]) \\ foo &= NAND(temp2, temp2) \end{aligned} \tag{0.5}$$

Now to prove that NAND-CIRC can calculate something that XOR-CIRC cannot. We know we can calculate $MAJ : \{0,1\}^4 \to \{0,1\}$ (which returns 1 if and only if at least 3 of its inputs are 1) with NAND-CIRC since NAND-CIRC is universal, so I will show that MAJ is not computable by XOR. I will prove the following lemma

Lemma 5 *For any XOR-CIRC program P with inputs of length 4 and one bit of output, there exits some $a_i, b \in \{0,1\}$ such that on any input x,*

$$P(x) = \sum_{i=0}^{3} a_i x_i + b \mod 2 \tag{0.6}$$

This lemma immediately implies that MAJ is not calculable in XOR-CIRC. Suppose towards a contradiction that P calculates MAJ. Fix a_i, b such that $P(x) = \sum_{i=0}^{3} a_i x_i + b \mod 2$. We know that all of $a_1 = 1$ since otherwise $P(1110) = a_1 + a_2 + a_3 + b \mod 2 = a_2 + a_3 + b \mod 2 = P(0110)$ (which can't be true because $MAJ(1110) = 1$ but $MAJ(0110) = 0$). Similar arguments imply that $a_2, a_3, a_4 = 1$. However, this implies that

$$\begin{aligned} 1 = MAJ(1111) = P(1111) = a_1 + a_2 + a_3 + a_4 + b \mod 2 \\ = a_3 + a_4 + b \mod 2 = P(0011) = MAJ(0011) \end{aligned} \tag{0.7}$$

Since $MAJ(0011) \neq 1$, this is a contradiction. Thus, MAJ is not computable by XOR-CIRC.

All that is left is then to prove the lemma. We will prove by induction the predicate $Q(n) =$ for any XOR-CIRC program $P$of n lines with input length 4, there exists $a_i, b \in \{0,1\}$ such that $\forall x \in \{0,1\}^4 . P(x) = \sum_{i=0}^{3} a_i x_i + b \mod 2$.

Base Case: For $n = 1$, there is only one line of the program, so it must assign $y[0]$. If the line is $y[0] = 1$ or $y[0] = 0$, then we choose $a_i = 0$ (where by this I mean set $a_i = 0$ for all i) and $b = 1$ or $b = 0$ respectively. The only other possible program is the line $y[0] = XOR(x[j], x[k])$. If $j = k$ then this is just the zero function for which we can choose $a_i = 0, b = 0$, and if $j \neq k$, then we can choose $a_j, a_k = 1$ and all other a_i and b as 0, giving us $P(x) = x_j + x_k \mod 2$, which is exactly what XOR does.

Inductive Step: Suppose $Q(i)$ is true for any $i \leq n$. We prove $Q(n+1)$, so fix a program P of $n+1$ lines. Consider the $n+1$th line of $Q(n+1)$. If this line is not of the form $y[0] = ...$, then the program is identical to a program with at most n lines (since the last line doesn't change the output), so by the inductive hypothesis, we can write $P(x) = \sum_{i=0}^{3} a_i x_i + b \mod 2$. Otherwise, if the last line is of the form $y[0] = 1$ or $y[0] = 0$ then we can set $a_i = 0$ and $b = 1$ or $b = 0$ respectively.

The most complicated case is if the last line is of the form $y[0] = NAND(c_1, c_2)$ where c_1, c_2 are either input variables or set on previous lines (by the validity condition). If c_k is an input variable $x[j]$, then we can write them of the form $c_k = \sum_{i=0}^{3} a_i x_i + b \mod 2$ by setting $b = 0$, $a_j = 1$, and the rest of the $a_i = 0$. Otherwise, it is set on a previous line, say the mth line. In that case, we can think of c_k as the output of an m-line program with $m \leq n$, so by the inductive hypothesis, we can write $c_k = \sum_{i=0}^{3} a_i x_i + b$. Thus, in either case, we can write $c_1 = \sum_{i=0}^{3} a_i x_i + b \mod 2$ and $c_2 = \sum_{i=0}^{3} a'_i x_i + b' \mod 2$. Thus, since XOR adds together its input modulus two, we have (applying modulus rules)

$$\begin{aligned} P(x) &= \sum_{i=0}^{3} a_i x_i + b + \sum_{i=0}^{3} a'_i x_i + b' \mod 2 \\ &= \sum_{i=0}^{3} (a_i + a'_i) x_i + (b + b') \mod 2 \\ &= (\sum_{i=0}^{3} (a_i + a'_i) x_i \mod 2) + (b + b' \mod 2) \mod 2 \\ &= \sum_{i=0}^{3} (a_i + a'_i \mod 2) x_i + (b + b' \mod 2) \mod 2 \end{aligned} \tag{0.8}$$

Since $b + b' \mod 2 \in \{0, 1\}$ and $a_i + a'_i \mod 2 \in \{0, 1\}$, this proves $Q(n + 1)$.

Problem 4: Review

Learning goal: The goal for this section is to reflect on how you can improve your proof skills!

(a) Select the problem from a previous week that you think you would learn the most from redoing. Copy the problem and your instructor's feedback below, rewrite the proof, and add a short reflection on the changes you made.

(b) (Additional Question) Choose another problem you received feedback on and copy the problem and feedback you received, rewrite the proof, and write a sentence about why you made your changes (so do Part A again) OR solve a problem you left blank from a previous week (try to pick from a week you felt least secure on).

Problem 5: Logistics

Purpose: This helps us make sure the course is going at the right speed!

(a) How long did you spend on the videos and readings this week?

(b) How long (including time in problem sessions) did you spend on this problem set?

(c) Do you have any feedback about the course in general (did the videos and readings sufficiently prepare you for the problem set)?