COMPSCI 501

Formal Language Theory

Semilecture #15: Review for Final Exam

David Mix Barrington

17 May 2023

Review For Midterm

- Regular and Finite-State Languages (2)
- CFL's and PDA's (1)
- Turing Machines and Computability (3)
- Miscellaneous Topics (1)
- P, NP, and NP-Completeness (4)
- Space Complexity (4)
- Circuits and NC (2)

Sets, Strings, Languages

- We start with **sets**, an **alphabet**, sets of **strings** over that alphabet, and finally **languages**.
- The decision problem for a language L is to input a string w and return a boolean telling whether w is in L.
- Our object of study is *how difficult* a decision problem might be for some language.

Sections of the Course

- In Part I, we look at classes of languages that can be decided with particular constraints: the **regular** languages and the **context-free** languages.
- In Part II, we look at whether languages can be **decided** or **recognized** by **Turing machines**.
- In Part III (the rest of the course after this exam) we will measure languages by the **resources** needed to decide them.

Regular Languages

- We defined **regular expressions**, and defined the **regular languages** to be those that can be defined by regular expressions.
- We then defined various kinds of **finite-state machines**.
- Our two central results were **Kleene's Theorem** and the **Myhill-Nerode Theorem**.

Kleene's Theorem

- The class of languages **recognizable** by **DFA's** is equal to the class of regulars.
- From any regular expression, we can build an equivalent **NFA** from it, then in turn build an equivalent DFA by the **Subset Construction**.
- From any DFA or NFA, we can build an equivalent regular expression by using **State Elimination**.

The Myhill-Nerode Theorem

- Given *any* language L, we can define an equivalence relation such that u and v are equivalent if for all z, $uz \in L$ iff $vz \in L$.
- A language has **finitely many classes** in this relation iff the language is regular.
- Given any correct DFA for a language, we can **minimize** it to get an equivalent DFA with the smallest possible number of states.

Non-Regular Languages

- A consequence of the MN Theorem is that we can prove a language L to *not* be regular if there is an **infinite** set of strings that are **pairwise distinguishable** for L.
- The **Regular Language Pumping Lemma** is another way to prove languages to not be regular. It's less useful for this purpose, but we will need to understand it in order to understand the later CFL Pumping Lemma.

Closure Properties

- A natural question is whether, if we apply some operation to one or two regular languages, the result is regular.
- The definition of the regular languages tells us that they are closed under **union**, **concatenation** and **star**.
- From Kleene, we know that they are closed under **complement** and **intersection**, because the recognizable languages are.

Grammars and CFL's

- A **grammar** is a way to define a language, so that a string w is in L(G) if we can derive w from the **start symbol** of G using its **rules**. A **context-free language** is one that can be defined by a **context-free grammar**.
- Grammars can be put in **Chomsky Normal Form**, which is useful if you are starting with an arbitrary grammar.

Pushdown Automata

- A **pushdown automaton** is an NFA with a **stack**. On one step, a PDA might **read** an input, **push** or **pop** on the stack.
- The language of a PDA is the set of strings that it could read from the input and end in a **final state**.
- Our central result is that PDA's and CFG's define *the same class* of languages.

TD and BU Parsers

- Given a grammar G, we can define a **top-down parser**, a PDA P with the same language.
- P begins by putting S on the stack, **applying rules** to **non-terminals** on the stack, and reading **terminals** from the input to **match** terminals taken from the stack. It finishes with an **empty stack**, having read all of the input.

TD and BU Parsers

- The similar **bottom-up parser** works in a way that is reversed in both time and location.
- It reads input letters and **shifts** them onto the stack, then applies **reducing** rules of G backward, until it creates S onto the stack.
- It can only accept by removing this S from the stack after shifting all the letters.

Building a CFG from a PDA

- Although it's not of practical importance, we learned the proof that from any PDA, we can construct an equivalent CFG.
- We first place the PDA into a **normal form**, where it only accepts with an **empty stack** and *either* pushes or pops one letter at each step, *never* both.
- Our grammar has one non-terminal A_{pq} for every pair of states in the PDA.

Building a CFG from a PDA

- The non-terminal A_{pq} generates any strings that can be read by the PDA when it starts in state p with an empty stack and finishes in state q with an empty stack.
- Any computation of the PDA with more than one step can be broken down into smaller computations.
- We either return to the empty stack in the middle, or we push a letter that we pop at the end.

Proving Non-CFLness

- We can prove various languages to be non-CFL's by using the **CFL** Pumping **Lemma**.
- Any CFL L has a constant p such that any string $w \in L$, with $|w| \geq p$, can be divided into strings u, v, x, y, z such that $|vy| > 0$, $|vxy| \leq p$, and for all naturals i, uv^ixy^iz is in L.

Proving Non-CFLness

- We proved this lemma by using **surgery** on **parse trees**. Any correct parse tree for w can be altered to make a correct parse tree for these new strings.
- To prove that L is not a CFL, we choose a long string w in L, and show that any possible division of w into uvxyz must either fail to **pump up** or **pump down**.

The Chomsky Hierarchy

- A **linear bounded automaton** acts like a Turing machine, but does not have access to tape beyond the amount needed for its input.
- We also looked at **unrestricted grammars**, which can generate exactly the set of languages that are recognized by Turing machines.

Turing Machines

- We saw things that a Turing machine can do, starting with copying strings, comparing strings, counting, and so forth.
- We saw that our familiar programming techniques could (in principle) be adapted to this setting.
- We can describe Turing machines **formally**, at **implementation level**, or at **high level**. The latter amounts to a proof that a formal description *exists*.

Variants of TM's

- We proved the **Multitape Theorem**, that any Turing machine with multiple tapes can be simulated by a single-tape machine.
- With this we were able to simulate a **non-deterministic** machine with an ordinary one, though at enormous cost in time.
- The language of an NTM is the set of all strings that are found by an exhaustive search.

Variants of TM's

- We also defined **enumerators** — machines that take no input, but put a succession of strings onto an output tape, such that the strings that appear are **enumerated**.
- A language L is the language of some Turing machine iff it can be enumerated.
- A language can be *decided* by a Turing machine iff it can be **enumerated in order**.

TD and TR Languages

- A **Turing-recognizable** or **TR** language L is one where there exists a Turing machine that accepts any string w iff $w \in L$. If $w \notin L$, the machine may *either reject* or *fail to halt*.
- A **Turing-decidable** or **TD** language L is one where there exists a Turing machine that *accepts* all strings w such that $w \in L$ and *rejects* all strings such that $w \notin L$.

TD and TR Languages

- We proved the TD/TR Theorem, which says that a language is TD iff it and its complement are both TR.
- A **Turing-decidable** or **TD** language L is one where there exists a Turing machine that *accepts* all strings w such that $w \in L$ and *rejects* all strings such that $w \notin L$.

TD and TR Languages

- One direction is easy — any TD language is TR, and the complement of a TD language is also TR.
- For the other direction, we need to use dovetailing to run two processes in parallel. If we have a recognizer for language L and a different one for L-bar, we can run both in parallel and one will be sure to eventually halt.

Decidable Languages

- We've defined a number of languages *about* other computing systems. For example, the language A_{DFA} is the set of pairs (D, w) such that D is a DFA, w is a string over D's input alphabet, and $w \in L(D)$. Similarly we have A_{CFL} and A_{TM}.
- E_{TM} is the set of Turing machines with empty languages, and ALL_{TM} is the set of Turing machines whose language is Σ^*.

Decidable Languages

- Most languages concerning regular languages or DFA's are decidable.
- Sipser proves that A_{CFL} is decidable, and we prevented the more practical CKYLalgorithm to decide that.
- We proved that E_{CFL} is decidable, but it turns out that ALL_{CFL} is not.

Undecidability

- Using the **diagonal argument**, we are able to show that certain languages are not TD.
- For example, if there were a Turing machine M that decided A_{TM}, we could build a Turing machine that would accept any string that is the encoding of a machine that does not accept itself. This is a contradiction.

Undecidability

- So A_{TM} cannot be TD. Since we know that it is TR, it must fail to be co-TR.
- Other languages like E_{TM} and ALL_{TM} are easily adapted to show them to be undecidable.
- For example, given any machine M and string w, let R be the machine that erases its input and runs M on w. If we could tell whether R were in E_{TM} or ALL_{TM}, we could decide whether M accepts w.

Accepting Computation Histories

- The method of **accepting computation histories** lets us show some problems about other computation models to be undecidable.
- A **configuration** is a string that represents the state of a Turing machine at one step. A computation history is a sequence of configurations where each succeeding one follows from the previous by the rules.

Accepting Computation Histories

- Once we start the machine, the remainder of the computation history is determined, since the machine is deterministic.
- If the machine accepts its input, there will be an **accepting** computation history, proving that the machine accepts that string.
- But testing the *validity* of an alleged ACH is generally easier than finding *whether one exists*.

Accepting Computation Histories

- Recall the **LBA**, which is like a Turing machine but is limited to its original tape space. If we get an alleged ACH, we can run an LBA on it to see whether it is valid.
- The question of whether M accepts w is equivalent to whether some particular LBA has any string that accept.
- So if we could decide E_{LBA}, we could solve this undecidable problem, so E_{LBA} is undecidable itself.

Accepting Computation Histories

- Similarly, with some work, we can show that the set of strings that are not ACH's form a CFL, as long as we reformat the computation histories to alternate between *forwards* and *backwards*.
- To fail to be an ACH, it must either (1) fail to start with q_0w, (2) make an incorrect step from one configuration to the next, or (3) fail to finish with the accepting configuration. (1) and (3) are regular, and (2) is a CFL.

Post's Correspondence Problem

- **PCP** is a simple undecidable problem using ACH's.
- A PCP instance is a set of **dominoes**, each with a non-empty string on top and bottom.
- A **match** is a sequence of dominoes such that the concatenation of the top strings and the concatenation of the bottom strings are equal. The PCP problem is to determine whether a match exists in the instance.

Post's Correspondence Problem

- With some care, we can set up a given instance such that any match in it must be derived from an ACH of some machine M and some string w.
- At any given time, there is one configuration hanging over the two strings. The only way to continue toward a match is to copy the string and make a new hanging string that has to be the next configuration of M.

Mapping Reductions

- A useful way to organize undecidable results is with **mapping reductions**. If A and B are any two languages, $A \leq_m B$ is true if there exists a computable function f such that for any string w, $w \in A$ iff $f(w) \in B$.
- A computable function is one where there is a Turing machine that takes any input w and *always* halts with f(w) on its tape.

Mapping Reductions

- The relation A $\leq_m$ B is transitive and is **closed downward** for classes like TD, TR, and co-TR.
- Thus if we know that A is *not* TD, for example, and A $\leq_m$ B is true, then B cannot be TD. Similarly, if A is not TR, then B cannot be TR.
- But remember that the function f must take *true* instances of A to *true* ones of B, and vice versa.

Mapping Reductions

- Earlier we saw a function f that took an instance (M, w) of A_{TM} and mapped it to a machine R that erases its input and runs M on w. This function is computable.
- This gives us reductions $A_{TM} \leq_m E_{TM}$-bar and $A_{TM} \leq_m ALL_{TM}$.
- As with NP-completeness, we can define a language A to be **TR-complete** if it is TR and if B is any TR language, $B \leq_m A$.

The Arithmetic Hierarchy

- The **Arithmetic Hierarchy** is a class of languages where each one can be described using **first-order logic**.
- We are allowed to use **TD predicates**, **boolean operators**, and ∃ and ∀ **quantifiers** over strings.
- If the formula is in **prenex form**, we can classify it as being "Σ^i" or "$\prod^i$", where the "i" means how many levels of quantification we have and Σ means starting with ∃ and ∏ means starting with ∀.

The Arithmetic Hierarchy

- Here's an example. Let the language **MIN** consist of every Turing machine M such that there is no *shorter* machine that has the same language as M.
- We can write "$M \in MIN$" as $\forall C: (|C| < |M|) \rightarrow \exists w: (w \in M) \oplus (w \in C)$.
- But "$w \in M$" is not a TD predicate. We have to something like "$\exists h: ACH(M, w, h)$" using acceptation histories.
- Using this to decode, we can put MIN in the AH.

The Recursion Theorem

- Sipser shows a way to build (at a high level) a Turing machine that *prints its own description*. This is also called a **quine**, and in the slides we included a Java version.

- The **Recursion Theorem** says that if t is any two-argument function on strings defined by a Turing machine (which may or not be computable — it may be a partial function) there exists a machine R such that for any string w, $R(w) = t(R, w)$.

The Recursion Theorem

- The proof for quitting uses an indirect way to specify a string, so that the resulting string turns out to be the exact description of the string itself. The proof for the Recursion Theorem is also similar.
- To use the Theorem in practice, you can include instructions to “obtain your own description” in your code.
- This is a good way to create viruses.

Kolmogorov Complexity

- We can classify strings by how many bits we need to specify the string.
- One way to do it is to simply quote the string, so that if it was n bits long, we will need n + c bits to do it that way.
- But if there is a regularity in the string, you might be able to specify it with fewer bits.

Kolmogorov Complexity

- If we can define a machine M and a string u such that M, running on u, halts with w on its string, then we have a description for w which is (M, u).
- The **Kolmogorov complexity** of w is the length of its shortest description.
- Note that we have to account for the format of "(M, u)" when we compute its size. For example, we might use **double-letter** notation for M and then write u.

Kolmogorov Complexity

- By a **counting argument**, we proved that there exist strings of any length with $K(w) \geq |w|$. We also showed that most strings have $K(w) \geq |w| - c$.
- Finally, we observed that this K function is **not computable**, since to know a description of w was the shortest possible, we would have to rule out any shorter string's computation that later halted with w.

Polynomial Time

- We now turn to **time complexity**, where we count the number of Turing machine steps needed to decide membership in a language.
- We use **asymptotic notation** such as O(f(n)) for time bounds, as in COMPSCI 311. The class DSPACE(f) is the set of languages X such that there is a machine M such that for any string w of length n, M decides whether $w \in X$ using O(f(n)) steps in the worst case.

Polynomial Time

- The set P is the class of languages that is the union of the classes DSPACE(n^k) for all naturals k.
- That is, the worst-case time for the machine to decide whether $w \in X$, where $|w| = n$, is **polynomial** in n.
- This class is **robust** in that many other models have the same "polynomial time" class as for Turing machines.

Examples of Polynomial Time

- We saw that reachability and connectivity for graphs can be decided in P.
- We can carry out the Euclidean Algorithm, so that we can test relative primality for n-bit number in time polynomial in n.
- We can decide the language A_{CFL} in P using the **CYK algorithm**, which uses dynamic programming. We used a similar algorithm on HW#4 to decide whether an element is a possible product in a **groupoid.**

NP and Verifiers

- We turned to examples that we don't know how to solve in P, but which we could solve in exponential time with a brute force search.
- These included HAM-PATH, CLIQUE, and SUBSET-SUM.
- In each case, we could **verify** that a particular string was in the language, given a **witness** string, such as a Hamilton path for the an instance of the HAM-PATH problem.
- Deciding whether a witness is valid for an instance is a problem in P.

NP and Verifiers

- Going back to nondeterministic Turing machines, we can define the class NSPACE(f) to be the languages of NDTM's that take at most O(f(n)) steps on inputs of length n, no matter how they make their choices.
- Similarly NP is the union of NSPACE(n^k) for all naturals n.
- It's not hard to prove that a language is in NP if and only if it has a poly-time verifiers. So there are two alternative definitions of NP.

P Versus NP

- Clearly P is contained in NP, and we don't believe that P = NP, but for over 50 years the mathematical community has failed to prove either that they are equal or that they are not.
- As a practical matter, we can prove a large class of languages to be outside of P *unless P = NP*. These are the **NP-complete** languages.
- There are languages outside P that are not NP-complete, *unless P = NP*, but examples of them don't come up so often.

P-Reductions, Completeness

- Just as we used mapping reductions in computability theory to prove completeness, we define **poly-time reductions** to prove completeness in complexity theory.
- If X and Y are languages, $X \leq_p Y$ is defined to be true if there exists a poly-time function f such that for all strings w, $(w \in X) \leftrightarrow f(w) \in Y$.
- The classes P and NP are **closed downward** under $\leq_p$. This means that non-membership in P is closed upward under $\leq_p$.

P-Reductions, Completeness

- Just as A_{TM} is naturally TR-complete under $\leq_m$, we can define a language A_{NP} that is naturally complete under $\leq_p$.
- We define A_{NP} to be the set of pairs $(M, w, 1^t)$ such that M is an NDTM, w is an input to M, and that M can accept w in at most t steps.
- Once we know that NP-complete languages exist, we can find more interesting examples.
- The methodology to prove Y to be NP-complete is to show $Y \in NP$, find a language X that is known to be NP-complete, and show $X \leq_p Y$.

CNF, SAT, and 3-SAT

- The methodology to prove Y to be NP-complete is to show $Y \in NP$, find a language X that is known to be NP-complete, and show $X \leq_p Y$.
- To start this process, A_{NP} is less useful than some other languages we'll now define.
- Recall **conjunctive normal form** in boolean logic. A formula is in **CNF** if it is the AND of clauses, each of which is an OR of literals.
- If the clauses each have exactly three literals, the formula is said to be in **3-CNF**.

CNF, SAT, and 3-SAT

- A formula is **satisfiable** if there exists at least one setting of the variables making it true.
- The **SAT** formula is to input a formula and output whether it is satisfiable.
- If the formula is in CNF, we have the **CNF-SAT** language, and if it is in 3-CNF, we have **3-SAT**.
- We will show that all three of these languages are NP-complete, and these are the most common sources for reductions to others.

The Cook-Levin Theorem

- We prove CNF-SAT to be NP-complete by taking an arbitrary single-tape poly-time NDTM M and construct a CNF formula (for a given input w) that is satisfiable if and only if there is a way for M to accept the string w. This reduces A_{NP} to CNF-SAT.
- The main idea is to construct a **tableau**, which is a two-dimensional array of letters from the tape alphabet (or states of M) as in Turing machine configurations. Each row of the tableau is a configuration at one time step.

The Cook-Levin Theorem

- A tableau represents a computation of M on w, following the rules of M. If there is a valid tableau that ends with the accept state, then w is in L(M).
- We will represent each cell of the tableau by a set of boolean variables. Choosing values of these variables defines the letters of the cells.
- We will write a CNF formula that expresses that the tableau represents a valid accepting computation of M on w. If this formula is satisfiable, then $w \in L(M)$.

Conditions on a Tableau

- So we need to specify conditions on making the tableau valid and accepting.
- We first need each of the variables for $x_{i,j,c}$ to represent exactly one cell for $x_{i,j}$.
- Then we need the time step for x_i to represent the proper start configuration.
- Then step i+i needs to be a possible successor configuration from step i.
- Finally, step p(n) must be accepting.

Conditions on a Tableau

- Each of these conditions can be expressed as a CNF formula. The most complicated one says that each cell of the tableau can possibly come from the three cells below it, according to the rules of M.
- Every 2 by 3 **window** of the tableau must be one that can occur in a valid computation, and if all the windows are legal, then the entire computation must be valid.

Legal Windows

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

(a) head moves left, overwrites b with a c
(b) head moves right, overwrites b with an a
(c) head moves off right, overwrites hidden letter with a b
(d) no change at all
(e) head moves left from hidden position
(f) head was to the left, moved left, overwrote b with c

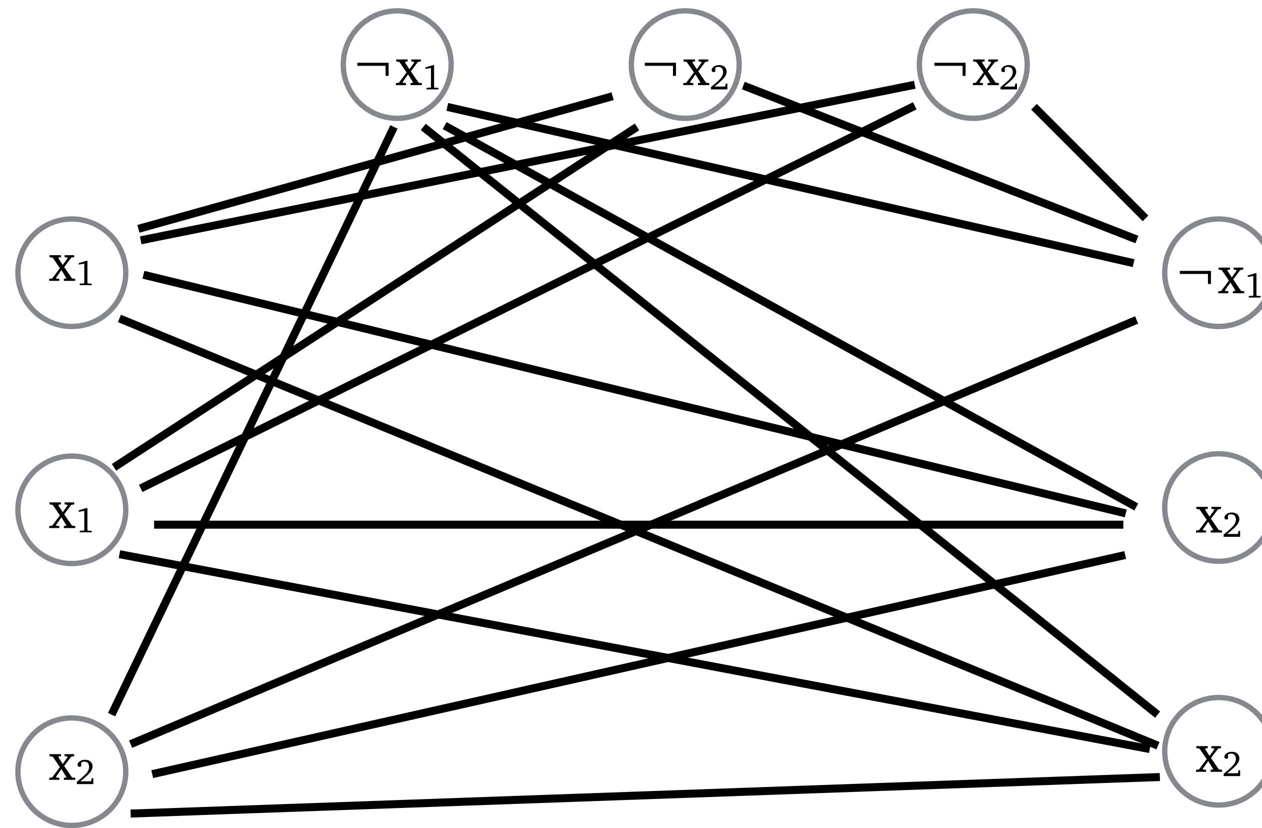
The Cook-Levin Theorem

- This argument proves that CNF-SAT is NP-complete.
- To show that 3-SAT is NP-complete, we need to show CNF-SAT $\leq_p$ 3-SAT.
- It's not possible to turn a CNF formula into an equivalent 3-CNF formula, but we can find a 3-CNF formula such that one is satisfiable if and only if the other is satisfiable.
- Each node of the CNF formula involves three variables, and we can use 3-CNF clauses to say that every node of the formula is correct.

Applying Cook-Levin

- Once we know that 3-SAT is NP-complete, we can show more languages to be NP-complete.
- In each case we first show that our new language X is in NP. Then we show 3-SAT $\leq_p$ X.
- On the next four slides we show some examples of four such reductions.

3-SAT $\leq_p$ CLIQUE

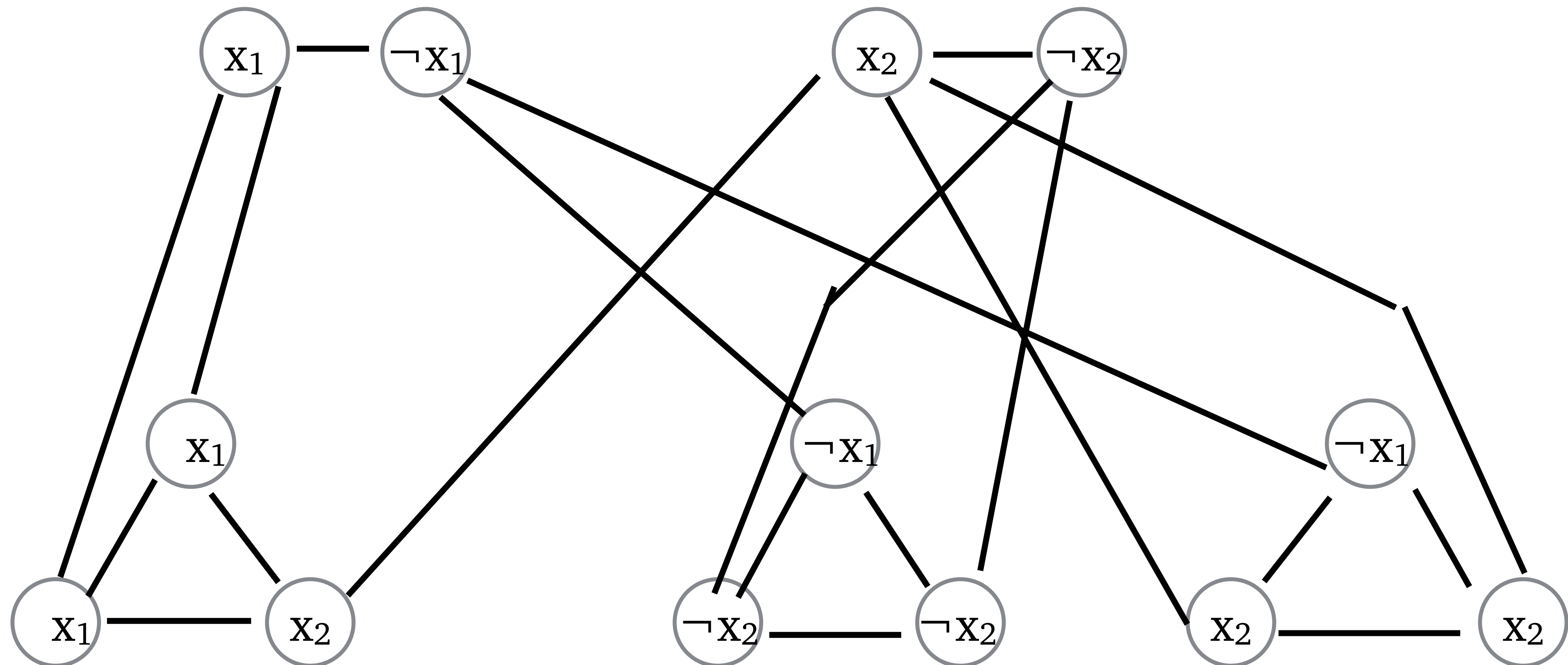


$$\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$$

We have edges if they are in *different* clauses, and the nodes *do not* have opposite labels like x_1 and $\neg x_1$.

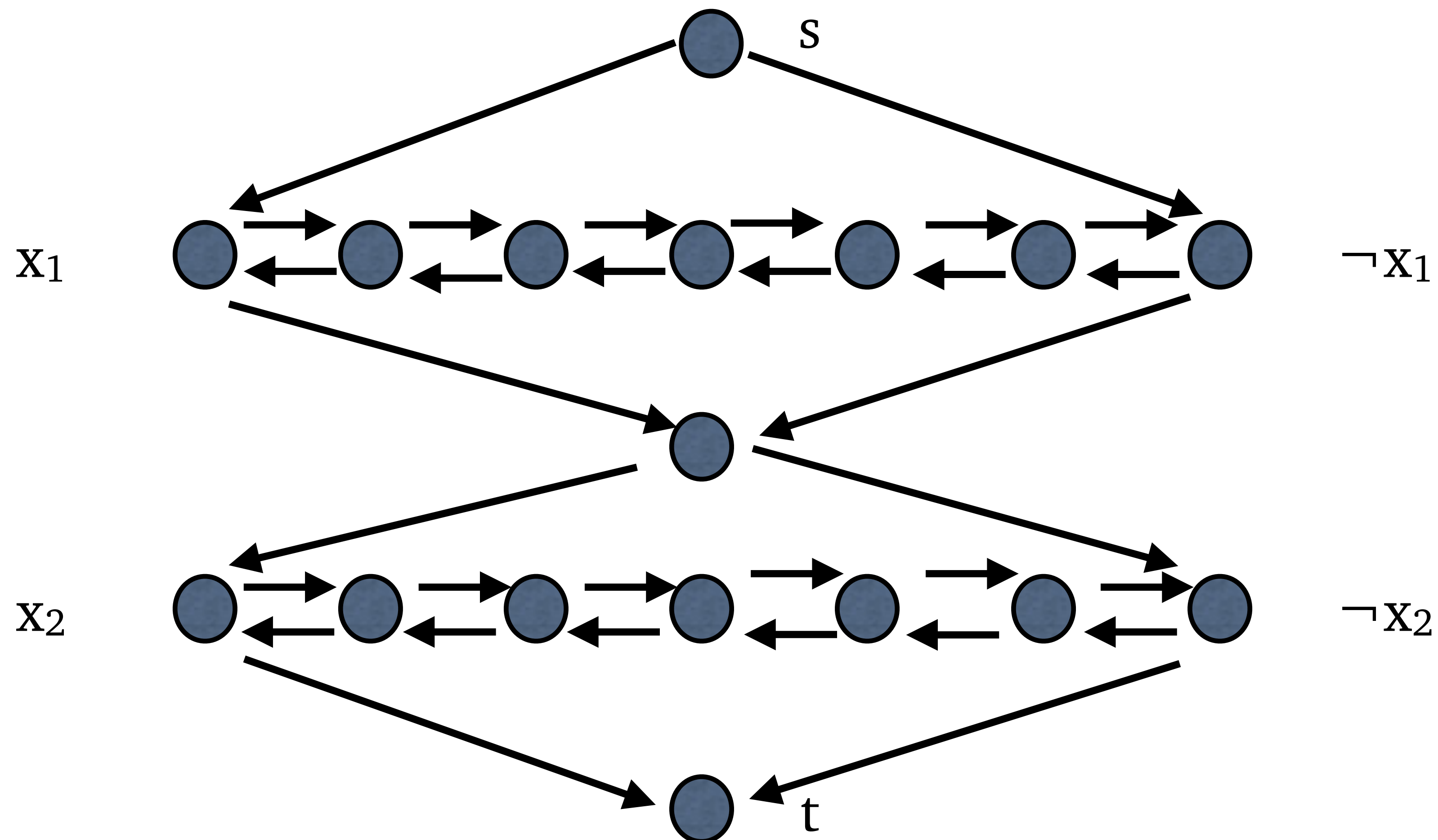
VERTEX-COVER is NP-C

To cover all the variable and edges, we need one at least one variable node and at least two of each clause node.



We also have to cover the transverse edges. But if we pick a good assignment for the variable nodes, we only need two nodes from each clause node. This covers with only m+2h.

Zigzags and Zagzigs



To touch all the nodes from s to t, we need to choose for each variable a **zigzag** from x_i to $\neg x_i$, or a **zagzig** from $\neg x_i$ to x_i. Choosing each zigzag or zagzig will correspond to making an assignment of the n boolean variables.

SUBSET-SUM is NP-complete

```
                           c1     ck
y1 | 1 0 0 0 … 0 | 1 0 … 0
z1 | 1 0 0 0 … 0 | 0 0 … 0
y2 |   1 0 0 … 0 | 0 1 … 0
z2 |   1 0 0 … 0 | 1 0 … 0
y3 |     1 0 … 0 | 1 1 … 0
z3 |     1 0 … 0 | 0 0 … 1
…
ym |           1 | 0 0 … 0
zm |           1 | 0 0 … 0
--------------------------
g1 |             | 1 0 … 0
h1 |             | 1 0 … 0
…
--------------------------
t  | 1 1 1 1 … 1 | 3 3 … 3
```

The 1's under the c's are for the literals.

If clause c_j has a positive literal x_i, we put a 1 in row y_i. If c_j has negative literal for x_i, we put 1 in z_i.

The g's and h's can fill from 1 to 3.

Defining the Hierarchy

- We earlier defined the **Arithmetic Hierarchy**, with alternating quantifiers with the base level as TD classes.
- So TR languages can be defined as Σ_1 languages, so that $A(w) = \exists x{:}R(w, x)$ where R is a TD predicate.
- With a slight change in the we can define the **Polynomial Hierarchy**, where the base level is the class P, and the quantifiers are bounded by polynomial length.

Defining the Hierarchy

- So a class in Σ_1 for the polynomial hierarchy is defined $A(w) \leftrightarrow \exists x{:}R(w, x)$ where R is a poly-time predicate and x is bounded in polynomial length in w.
- Thus the time to compute R(w, x) is bounded in poly time in the length of w.
- Similarly we can define a Π_1 class in the polynomial hierarchy by $B(w) \leftrightarrow \forall x{:}R(w, x)$, with the same conditions. R is poly-time, and the length of x is poly(w).

Examples of the Hierarchy

- NP is the class Σ_1 of the hierarchy.
- co-NP is the class $\prod_1$.
- The set of **tautologies** in boolean logic is complete for $\prod_1$.
- The set of **minimal formulas** in boolean logic, the ones that do not have shorter formulas equivalent to them, is complete for $\prod_2$.

ATM's and the Hierarchy

- An **alternating Turing machine** is a generalization of an NDTM. Along with nondeterministic choices by some entity that wants the machine to accept, there are also **universal choices** that are made by some other entity that wants the machine to reject.
- The **game semantics** is my preferred way to explain the meaning of alternating TM's. Every state of the machine is a choice for White, who wants it to accept, or for Black, who wants it to reject. One or the other must have a winning strategy.

ATM's and the Hierarchy

- In an ordinary NDTM, the input is in the language if and only if White, who makes all the choices, can make the machine accept.
- If a language is in the complement of the language of an NDTM, we can build an ATM where White is trying to make the original machine reject, but Black makes all the choices, and White can only win if there is no accepting path.

Defining PSPACE

- Let X be a language, and we have an algorithm that can decide, whether any string w of length n is in X, using at most O(f(n)) total tape cells.
- Then X is in the space complexity class **DSPACE(f)**.
- The class **PSPACE** is the set of languages that are in DSPACE(n^k) for any constant k.
- This is the simplest space complexity class, but it is a very powerful class.

Polynomial Space

- It's easy to observe that P is contained in PSPACE.
- But we can also decide a language in NP using polynomial space. We simply carry out the brute force simulation — the time is terrible, but we have enough space to keep track of which computation path we are testing.
- Similarly we can show by induction that every class Σ_k in the polynomial hierarchy is contained within PSPACE.

Polynomial Space

- PSPACE is contained within exponential time. If it runs more that the number of configurations it has, we can shut it off.
- Sipser applies this idea to show that the language ALL_{NFA} is in NPSPACE. By reasoning about the corresponding DFA of our NFA, we know that if it doesn't accept everything, there must be a string of exponential length that is not in L(N).
- With polynomial space, we can guess this string if it exists, using the memory to verify that each configuration follows from the next.

Logarithmic Space

- We now turn to smaller space classes, starting with **L**, the languages that can be decided in O(log n) tape cells on input of size n.
- This only makes sense if we get our input on a **read-only input tape**, and only count our read-write memory toward our space bound.
- A two-way DFA defines the class DSPACE(1), where there is only constant memory other than the read-only input tape.

Logarithmic Space

- We use "L" and "NL" to mean DSPACE(log n) and NSPACE(log n) respectively.
- We can define $\{a^n b^n : n \geq 0\}$, or even $\{a^n b^n c^n : n \geq 0\}$, in L by counting the letters in binary.
- In NL, we can solve **reachability** in a directed graph. In O(log n) space, we can remember one node number, verify that another node has an edge to it, and jump to the new node, forgetting the old one. If we can get to the target node, then there was a path, and conversely.

Logarithmic Space

- Of course we know that reachability is in P, using DFS or BFS, but these use linear space and so the NL algorithm is quite different.
- Any language in L or NL is also in P.
- Given a space-bounded machine, we can make its **configuration graph**, with a node for each configuration and an edge to any node for a configuration we could to go in one step. In a log-space machine, there are only a polynomial number of configurations.

Logarithmic Space

- In a machine for L, we can start at the node for the start configuration, follow the unique edge from every node, and see whether we reach an accepting configuration.
- In a machine for NL, we want to know whether there is any path from the start configuration to the accepting one. In P, we can answer that using DFS or BFS. So NL $\subseteq$ P.

Savitch's Theorem

- We think that NP is a much larger class than P, as we don't know how to decide an NP language deterministically with less than exponential time.
- But NL seems to be closer to L. **Savitch's Theorem** says that NL is contained in DSPACE($\log^2 n$). This is an important use of dynamic programming.
- How well can we solve reachability if our goal is to minimize *deterministic space*?

Savitch's Theorem

- How well can we solve reachability if our goal is to minimize *deterministic space*?
- Let PATH(c, d, t) be a predicate saying that there is a path of at most t steps from c to d.
- We can easily find PATH(c, d, 1) as a base case.
- We can find PATH(c, d, 2) by testing any node e to see whether both PATH(c, e, 1) and PATH(e, d, 1) are both true.

Savitch’s Theorem

- Similarly, we can attack the problem of PATH(c, d, t) by *recursively* testing, for each node e, whether PATH(c, e, t/2) and PATH(e, d, t/2) are both true.
- Our recursion will need a recursion depth O(log t). In an NL machine, the size of the graph for which we want reachability is polynomial, so the recursion depth is O(log n).
- In the recursion, our stack frame will need O(log n) bits for each recursive call, since we must remember the node e, which is O(log n) bits.

Savitch's Theorem

- The time for this algorithm is terrible, $O(n^{\log n})$ in general, but it does test membership for NL in DSPACE($\log^2$).
- This generalizes, as long as $f \geq \log n$, to say that NSPACE(f) $\subseteq$ DSPACE(f^2).
- A special case of this is that NSPACE(n^k) $\subseteq$ DSPACE(n^{2k}), from which it follows that NPSPACE = PSPACE.

Completeness for NL

- For it to make sense for a problem to be complete for NL, we can't use p-reductions because every nontrivial language in P is $\leq_p$ reducible to any other.
- Within P, we'll define $\leq_L$ reductions, defined just as with $\leq_p$ except that the function making $w \in X \leftrightarrow f(w) \in Y$ is a **log-space transducer**.
- This is a machine with w on a read-only input tape, O(log n) bits of read-write memory, and a **write-only output tape** where it puts f(w).

Completeness for NL

- L-transducers are transitive! It's not immediate that if $X \leq_L Y$ and $Y \leq_L Z$, then $X \leq_L Z$, because you *a priori* need too much read-write space to hold the string in Y.
- The trick is, whenever you need a bit of the string in Y, is to recompute it from w, going back to the start of the computation that defines it. We use $O(\log n)$ space for each of the two computations, and this is still $O(\log n)$ space.
- In practice, if there are only $O(1)$ compositions, you can think of the intermediate steps being in read-write memory.

Completeness for NL

- Once we have the definitions, it's pretty obvious that the language REACH is NL-complete. We know that it's in NL. If we have an arbitrary NL language, there is a log-space NDTM for it, and it has a configuration graph with polynomially many nodes. All we need to know is whether there is a path from the start to final configuration.
- Why is this log space? We just have to construct the configuration graph, which is just cycling through all the possible configurations.

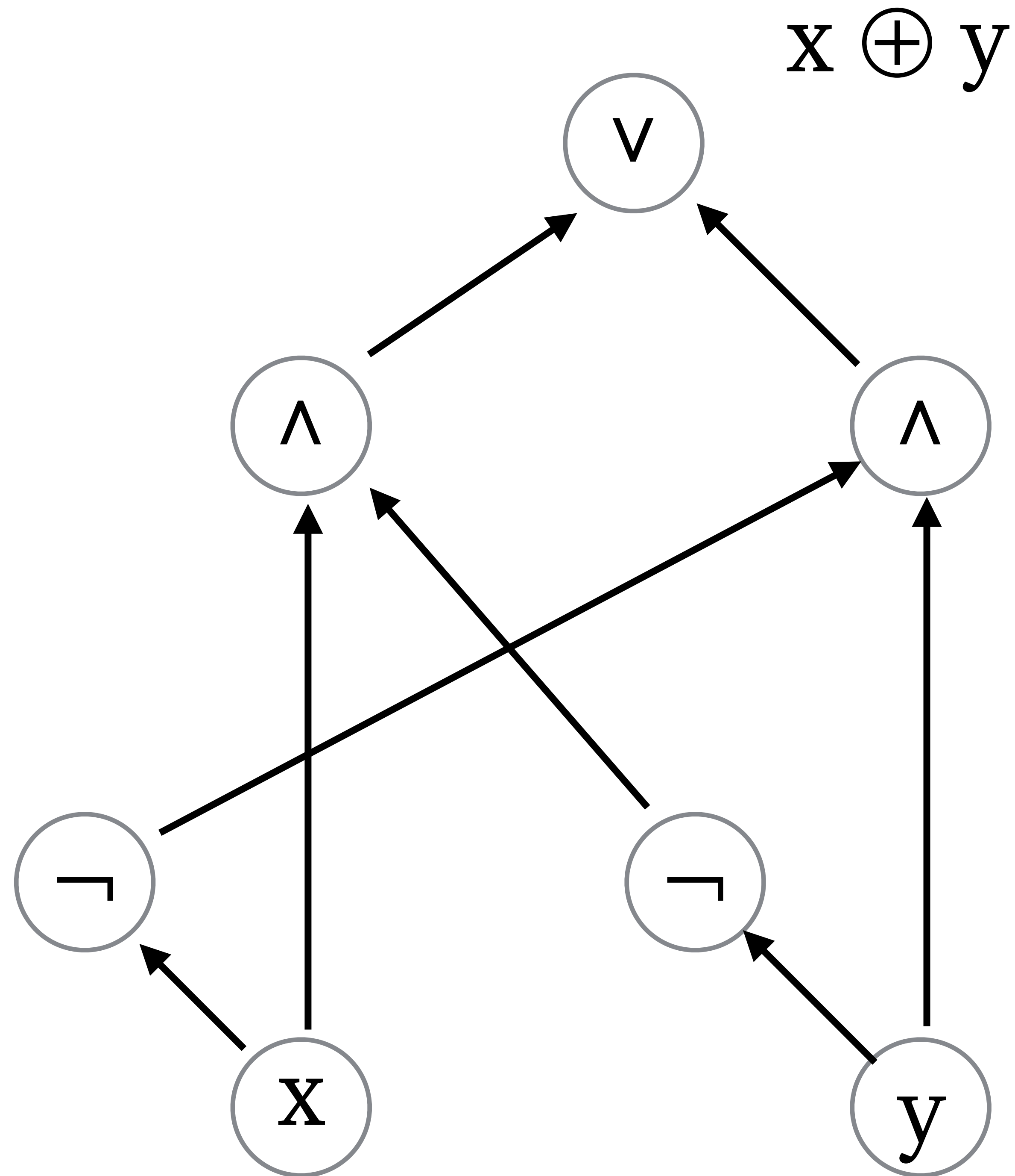
Completeness for P

- Having $\leq_L$ reductions lets us talk about completeness for the class P.
- We could define a language A_P of the tuples (M, w, 1^p) such that M accepts w in at most p steps. It's pretty clear that any language in P can be reduced to A_P using $\leq_L$ reductions.
- But a more interesting P-complete language brings us to **circuit complexity**.

Circuit Complexity

- A **boolean circuit** is similar to a boolean formula, but gates are allowed to **fan out** to multiple gates.
- The **size** of the circuit is the number of nodes, and its **depth** is the length of the longest path from the **output node** to any **input node**. An input node gets the value of one bit of the input string.
- Every node, except for an input node, is computed from a binary AND gate, a binary OR gate, or a unary NOT gate. The directed circuit of the graph must be **acyclic** — we cannot have later gates feed into earlier ones.

Some Circuit Examples



Some Circuit Examples

- We can construct a poly-size circuit for any regular language. We can just simulate a DFA, using a constant number of gates for each time step, using height O(n) and size O(n). (And this is contained in L and NC^1.)
- We can construct a poly-size circuit for any CFL, using the CYK algorithm. In a result we will probably not get to COMPSCI 501, we can show that the class of CFL's is bounded within a class that is (probably) not a strict subset of P.

The Circuit Value Problem

- Let C be a circuit with inputs $x_1,\ldots,x_n$ and with p(n) total nodes.
- The **Circuit Value Problem** is to take a boolean assignment of the input values and find the value of the input node.
- It should be pretty clear that CVP is contained in the class P. If we have a Turing machine, with polynomial space available, all we need to do is evaluate each node in the right order until we have the value of the input node.

Cook-Levin Again

- In order to prove that CVP is complete for P, we need to take an arbitrary language X in the class P, and prove $X \leq_L$ CVP.
- So given an input w and a Turing machine M and a polynomial time bound p(n), we need to create a circuit of polynomial nodes, with $w_1,\ldots,w_n$ as the inputs, such that $w \in X$ iff the output of the the circuit on the input gives 1.
- And we have to construct the circuit from w using a log-space transducer.

Cook-Levin Again

- Fortunately we have already done this work.
- We can even simplify the Cook-Levin **tableau** construction, because our poly-time single-tape computation is *deterministic*.
- Lay out a square tableau of configurations for each row of p(n) cells and states.
- Then we have one configuration for each of the p(n) time steps. To compute a cell for position x and time step t, we can do this with a circuit.

Cook-Levin Again

- Then we have one configuration for each of the p(n) time steps. To compute a cell for position x and time step t, we can do this with a circuit.
- If we have cells (x-1,t-1), (x, t-1), and (x+1, t-1), each encoded as enough boolean nodes, we can take this input and feed into into a circuit to get the result of cell (x, t).
- Note that we will need multiple fan-out, since for example (x-1,t-1) must also feed (x-1,t-1).

Complete Languages for PSPACE

- It's pretty easy to prove that *some* language exists that is complete for PSPACE. We need X to be in PSPACE, and that $Y \leq_p X$ for every Y in SPACE.
- Let A_{PS} be the language $(M, w, 1^t)$ such that M accepts w using at most t tape cells.
- If Y is in PSPACE, let $Y = L(M_Y)$, where M_Y is bounded by p(n) space.
- Then we map w to $(M_Y, w, 1^{p(n)})$. So $w \in Y$ iff $(M_Y, w, 1^{p(n)}) \in A_{PS}$.

Quantified Boolean Formulas

- Our more interesting natural PSPACE complete language is defined with **quantified boolean formulas**.
- We start with n boolean variables, combine them with boolean operators, and **quantify** boolean variables using ∃ and ∀ quantifiers.
- One example of this is satisfiability, when we write a boolean formula and then bind each variable with ∃ quantifiers.

Solving TQBF in PSPACE

- So if the formula begins as $\exists x_1 : \varphi(x_2, \ldots, x_n)$, we can decide whether the sentence is true.
- We try setting x_1 to 1, accept if the result becomes 1, then otherwise check $x_1 = 0$.
- Checking these two results are each recursive. The space we need for the computation is one bit for x_1, then whatever space we need for the rest of either of the two computions.
- We only need linear space to finish it.

Reducing PSPACE to TQBF

- So now we are left with the harder half of the proof, where we have a language A in PSPACE and we need to prove $A \leq_p$ TQBF.
- Here's an approach that doesn't work.
- Design a a tableau as for Cook-Levin, with width $O(n^k)$, assigning boolean variables and expressing that we have an accepting tableau. The problem is that the height of the tableau is exponential in n.

Reducing PSPACE to TQBF

- We can represent the configuration at any time step by $O(n^k) =$ bits.
- Let any two configurations and let t be any number up to the exponential time bound.
- We can compute, as for Cook-Levin, whether the machine can take it from configuration c to configuration c' in one step.
- How can we determine whether M can take c to d in at most $2^{p(n)}$ steps?

Expressing Paths by Savitch

- Suppose we express the statement $\varphi_{c,\ d,\ t}$ that says we can go from c to d in at most t steps.
- We can write $\varphi_{c,d,t} = \exists e{:}[\varphi_{c,e,t/2} \wedge \varphi_{e,d,t/2}]$.
- Of course e is not a boolean, so we are writing $\exists e_1 \ldots e_m{:}[\varphi_{c,e,t/2} \wedge \varphi_{e,d,t/2}]$.
- That's ok, but there's a problem with when we double the boolean formula poly-many times. We don't have a poly-length TQBF formula, so we can't map $\leq_p$ to it.

Saving Formula Space

- So far, though, we haven't used both types of quantifiers.
- To say that both $\varphi_{c,e,t/2}$ and $\varphi_{e,d,t/2}$ are true, we can write $\forall c_1,c_2:\{(c,e), (e,d)\}\varphi_{c1,c2,t/2}$.
- This can be written in boolean operations as $[((c_1 \leftrightarrow c) \wedge (c_2 \leftrightarrow e)) \vee ((c_1 \leftrightarrow e) \wedge (c_2 \leftrightarrow d))] \rightarrow \varphi_{c1,c2,t/2}$.
- However we pick c_1 and c_2, these conditions make those two φ statements both true.

Saving Formula Space

- Our TQBF formula reduces the $2^{p(n)}$-step formula to a $2^{p(n)}/2$-step formula by adding $\exists e_1 \ldots e_m : \forall c_1 c_2$:(booleans), which is polynomial length.
- This makes the entire TQBF formula still in polynomial length, since we have p(n) rounds.
- More precisely, the formula length is $O(p^2(n))$, since we have $O(p(n))$ $\exists$ quantifiers, and $O(p(n))$ rounds in all.

The Formula Game for TQBF

- We've proved that the language TQBF, of the set of quantified sentences formulas that evaluate to true, is complete for PSPACE.
- Here is a natural interpretation of TQBF as a **formula game**.
- Given the quantified formula in prenex form, the game proceeds by the players **choosing variables.**

Generalized Geography

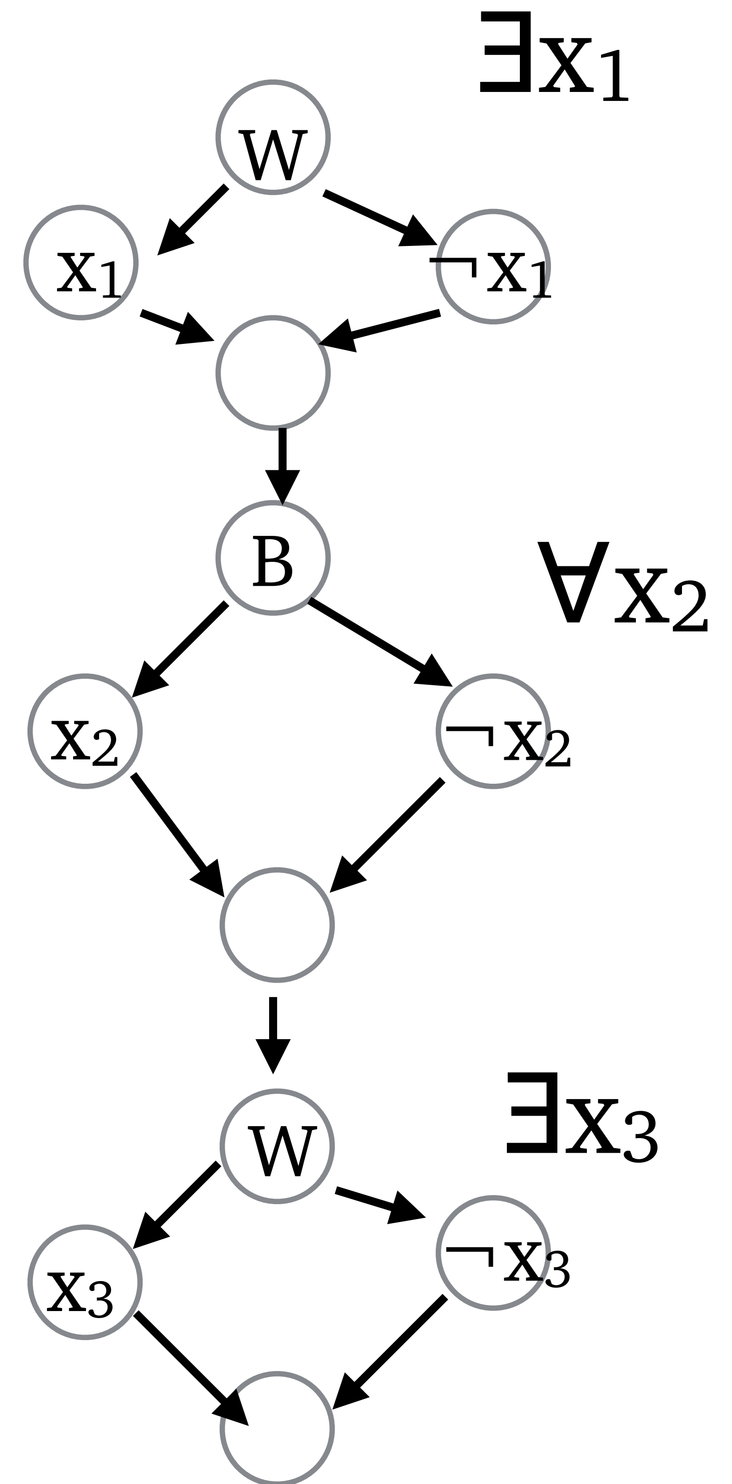
- Here is another game that turns out to be PSPACE-complete as well.
- Sipser first describes it as **generalized geography**, where the game board is a set of cities.
- The first player chooses a city. Each succeeding city must begin with the same letter as the last letter of the previous city.
- So if you start with AMHERST, I could move TURNERSFALLS, and I could follow with SALEM.

Formula Game $\leq_p$ GG

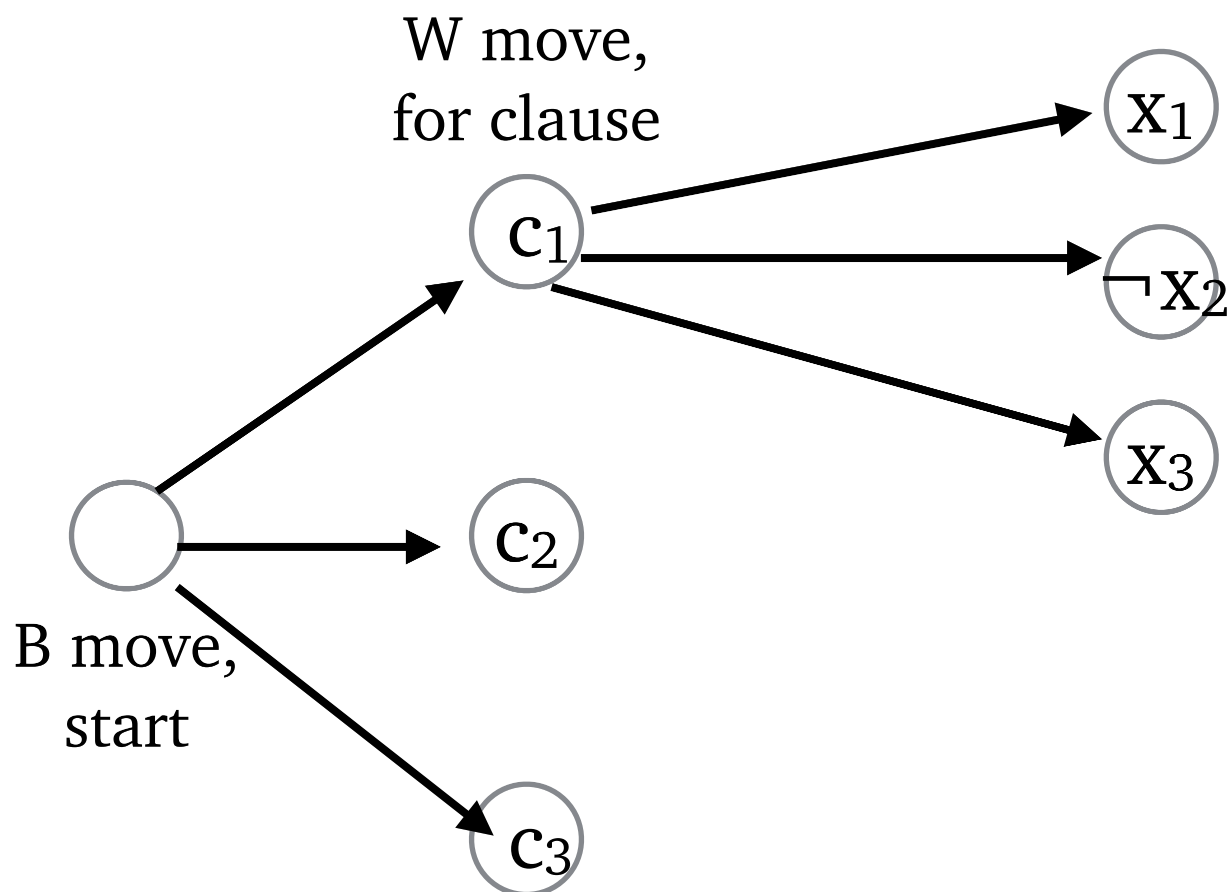
- We now have the tools to show that the GG game is also PSPACE-complete.
- We've shown that GG is in PSPACE, and we will show that the Formula Game reduces to GG.
- Given an arbitrary QBF formula, we'll assume that the part after the quantified parts are in CNF form. (The argument for TQBF completeness works under this assumption, as we can check.)

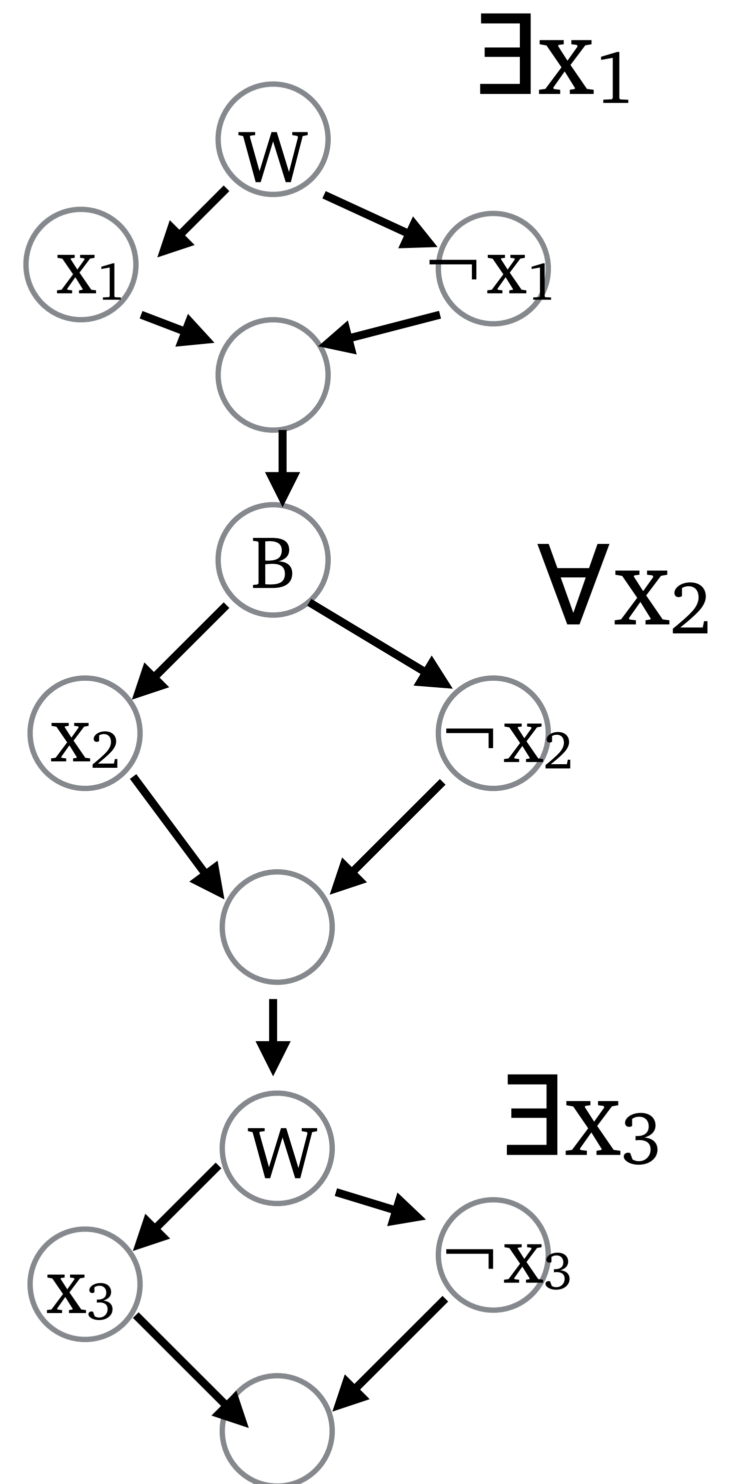
Formula Game $\leq_p$ GG

- We set up our GG graph with a start node, where White will make the first move, and the second move is third moves are forced.
- Then Black will make a free choice in the next move and the next two are forced.
- They will alternate, each choosing one node.



Formula Game $\leq_p$ GG





If the boolean part is true, then every clause is satisfied. We let Black pick which clause, and White tries to pick a true literal.

Poly-Time Games in PSPACE

- In general, when we play a game with a polynomial time bound, the winning strategy can be determined in PSPACE.
- You recursively find who has the strategy, with the base case being the final position of the game, and each step recursing from time t to time t+1.
- Because the time bound is polynomial, you only need room in the method stack for polynomial space. (Each stack frame uses only polynomial space.)

Log-Space Games in P

- But if we play for polynomial time on a game board that remains fixed, where we keep track of a only constant number of positions on the board, we can find winning strategies in P (poly-time).
- A position in the game is recorded by O(log n) bits, if the fixed board is stored in read-only memory. So there are only polynomially many configurations in the game.

A P-Complete Game

- The **circuit game** starts with the position in the output node of the circuit.
- When the current node is an OR gate, White moves to an input of the current node. If the current node is an AND gate, Black move to an input of the current node.
- When we reach an input literal, White wins the game if it is 1 and Black wins if it is 0.

A P-Complete Game

- Now remember that the players have infinite computational power available. So each one of them knows exactly which node will evaluate to, given the input values and the fixed circuit.

- So if the input node will evaluate to 1, White should have a winning strategy, and she does. If it is her move, there must be an input that evaluates to 1. If it is Black's move, then every move must evaluate to 1.

Non-Reachability in NL

- Using our knowledge about completeness, we'll prove the result by showing that **non-reachability** in directed graphs is in NL.
- We need a nondeterministic procedure, with input (G, s, t), that can accept its input if and only if there *isn't* a directed path from s to t.
- The key method is to **count**, for any natural d, the number of nodes that can be reached from s with at most d edges.

Inductive Counting for NL

- Let N_d be the number of nodes that have paths of length of d or fewer edges, starting from s.
- We know that $N_0 = 1$, since only s can be reached from itself with no edges. What about N_1?
- We can find it deterministically in L by counting the edges out of s. But getting N_2 gets harder. If we cycle through all the two-step paths from s, we have to make sure we deal with multiple paths to the same place.
- Here's where we can use nondeterminism.

Inductive Counting for NL

- If there is a path of length at most d from s to x, our nondeterministic machine can verify this by taking a path from s to x, remembering how many edges it has taken. As in our NL algorithm for reachability, it only needs to remember the current node, and the number of edges, in its log space memory.
- Suppose I *know* the value of N_d. I can cycle through all the nodes, and for each one I *guess* whether it has a path of length at most d from s. If it is, I verify this fact.

Inductive Counting for NL

- Suppose I *know* the value of N_d. I can cycle through all the nodes, and for each one I *guess* whether it has a path of length at most d from s. If it is, I verify this fact.
- If, when I've gone through all the nodes, unless I've guessed and verified N_d of those nodes, my procedure fails. There *exists* a sequence of correct guesses, for the correct value of N_d, and we can consider only runs of the procedure when we find it.

Inductive Counting for NL

- Suppose I want to know whether a particular node z has a path of length at most d+1 from s.
- If I go through this process to guess and verify the N_d nodes at distance d, *at the same time* I can decide whether there is a (d+1)-path to z.
- If z itself is one of my guessed nodes, then it does have a path. Also, if any of my guessed nodes has an edge to z, then z has a path. If neither of these things happen, and the guessing was correct, then z does not have a (d+1)-path.

Inductive Counting for NL

- I don't have space to remember the nodes I've seen, but I can remember how many nodes with (d+1)-paths there are. (I'm doing the guess and verify process again for each node I'm checking.)
- *If* all the guesses and verifications are correct, we have used the value of N_d to compute N_{d+1}.
- By repeating this process, I will *either* get the correct value of N_{n-1} *or* my procedure fails.

NLBA's and Context-Sensitive

- When Immerman and Szelepcsenyi proved their result (and won the Godel Prize in 1995), part of their fame was due to solving a problem older than the formalization of NL and co-NL.
- We saw earlier that the **context-sensitive grammars** define the class of languages we've called NLBA's, nondeterministic **linear bounded automata**. These are Turing machines that are limited to the read/write space used to store the original input.

NSPACE(f) = co-NSPACE(f)

- If f is any space bound with $f(n) \geq \log n$, we can apply the same argument to show that NSPACE(f) is equal to co-NSPACE(f).
- (There are subtleties about whether f is **space constructible**, but for convenience we will assume that it is.)
- If M is an NSPACE(f) machine, then it has $2^{O(f)}$ nodes in its configuration graph. Let X be a co-NSPACE(f) language, so that X-bar = L(M). All we need to show $w \in X$ is that there is no accepting path on M with input w.

NSPACE(f) = co-NSPACE(f)

- If M is an NSPACE(f) machine, then it has $2^{O(f)}$ nodes in its configuration graph. Let X be a co-NSPACE(f) language, so that X-bar = L(M). All we need to show $w \in X$ is that there is no accepting path on M with input w.
- By the algorithm we've demonstrated, we can nondeterministically test whether there *isn't* such a path using space $\log(2^{O(f)}) = O(f)$. So membership in X is in NSPACE(f) as well as X-bar.

ATIME(f) $\subseteq$ SPACE(f)

- So in these two lectures we'll present the **Meyer-Stockmeyer Alternation Theorem**, following Sipser's argument in Chapter 10.3.
- In this lecture we'll characterize a time-bounded alternating TM (called an ATM, potentially confused with an Automated Teller Machine or Asynchronous Transfer Mode).
- Let O(f(n)) be a time bound, with $f(n) \geq n$. Our first result will simulate ATIME(f) with deterministic space O(f).

Recursion Again

- We've essentially seen this argument already with the Formula Game of Lecture 12.2.
- We first alter the machine to add a clock, so that every configuration has a time step.
- We want to compute the acceptance (or the winning strategy) of every configuration c for every time step t.
- If we know the result for of every node c reachable on time step t+1, we can compute the result for c.

SPACE(f) $\subseteq$ ATIME(f^2)

- For the other result in this lecture, we see how well we can attack a deterministic space-bounded problem using alternating time.
- We don't actually prove that ATIME(f) = SPACE(f), though we can come close.
- Remember that a computation with O(f) space uses $2^{O(f)}$ configurations. For either SPACE(f) or NSPACE(f), we need to determine whether there is a path from the start configuration to an accepting one (or *the* accepting one if we prefer).

Savitch Again

- But we can solve this path problem using Savitch.
- We need to express the argument a bit differently when we have alternation available.
- In game semantics, we will define the **Savitch game**, where White will have a winning strategy if the desired path exists, and Black wins if not.
- We begin with White claiming the path exists, for a number of edges at most t. White's move is to **claim** that node a to b exists because there is a paths of length at most t/2 from a to c and c to b.

ASPACE(f) $\subseteq$ DTIME($2^{O(f)}$)

- Let's first consider the class ASPACE(f), again where $f(n) \geq \log n$.
- In the game semantics, we have a game where White wants to make the machine accept and Black wants it to reject, and each state of the machine is assigned to one of the two players to choose the next move.
- The first question, as always with space bounded classes, is the number of configurations that the machine has.

Configuration Graphs Again

- There are $c^{f(n)}$ tape cells in the read-write memory, n locations in the read-only input, O(f(n)) possible head positions in the read-write memory, and O(1) possible states.
- Multiplying this out, again since $f(n) \geq n$, there are $2^{O(f)}$ total configurations.
- As we've seen we will use a **marking algorithm** to determine the winning strategy.
- We can also put a clock on the machine, so that there are no cycles in the directed graph of moves.

DTIME($2^{O(f)}$) $\subseteq$ ASPACE(f)

- So now we turn to the remaining piece of the Alternation Theorem, using alternating space to determine the result of a DTIME($2^{O(f)}$) computation.
- We can view the computation as a tableau, with $2^{O(f)}$ cells in each row, and $2^{O(f)}$ total rows.
- But we are now given only f(n) total space.
- We don't have room to write the tableau or even a row if the tableau, but we can write a **pointer** into the tableau.

A Circuit-Like Game

- Also, unlike the tableau in the Cook-Levin Theorem, each cell is *deterministically* defined by the three cells next to it in the prior row.
- We'll define our game where the position at any time is one cell of the tableau. We'll use the O(f) space given to us to keep track of where we are.
- The start of the game is the final cell of the tableau, where White claims that the contents of this cell is the accepting state. At any given time, White is maintaining a claim of the current cell.

A Circuit-Like Game

- So at a move at some generic row and time step for a cell, White first moves by naming the three adjacent cells at the prior step. Unless these three cells verify White's claim about the original cell, White loses the game immediately.
- Black then picks one of the three cells, and the game continues with White claiming the contents of that new cell based on her prior moves.
- Let's prove that the player has the right strategy.

A Circuit-Like Game

- What if the DTIME machine accepts? In this case White can make a correct claim at the start of the game. White can just *tell the truth*, claiming the three adjacent cells have the contents that they really have in the actual computation. At the end, White's final claim matches the input.
- What if the DTIME does not accept? Then White is *lying* about the final cell. At every new move, White must lie *again* about the three new cells, since they have to agree with her false claim .

A Circuit-Like Game

- So at least one of White's claimed three cells must disagree with the actual result of that cell in the original computation. All Black needs is to *challenge the lie*.
- At each step, then, White must maintain an incorrect claim about the current cell.
- Once we reach the input level, White must now be claiming an incorrect $t = 0$ cell, and will lose.

Circuit Classes Again

- If we allow ourselves a polynomial number of computing elements, we can put one on each of the nodes of a polynomial-size circuit.
- The **parallel time** of the resulting circuit depends on the **depth** of the circuit.
- The output value of any given gate cannot be computed until its input values have already been computed.

Circuit Classes Again

- A given circuit is parametrized by both its **size**, the number of gates, and its **depth**, which is the longest path (in its directed graph) from any input node to the output node.
- The depth will in general be proportional to the time to compute the circuit, given a processor for each gate.
- So we can study what size and depth bounds we can find for different problems.

Uniformity for Circuits

- We can define complexity classes for languages, in terms of size or depth, just as we do this for time and space for Turing machine computation.
- But there are some fundamental complications.
- If we design a boolean circuit, we have to specify its number of inputs. (Unlike a Turing machine, for example, a single computer can operate on any number of inputs.)
- We thus define a **circuit family** where C_n is a circuit for each number n of inputs for each one.

Uniformity for Circuits

- We could then define the class PSIZE, for example, so that a language X is in PSIZE if there exists a circuit family such that for each circuit C_n, for every bit string w in Σ^n, w on C_n returns 1 if and only if w is in X.
- But there's a problem with this definition.
- Consider the language, say, $\{1^n$: n is in A_{TM} in binary$\}$. This is clearly not decidable.
- But the *size complexity* of this language is O(1)!

Uniformity for Circuits

- If we want to define an "easy to compute" language in terms of circuit size or depth, we also need a new condition called **uniformity**.
- For any particular circuit C_n, there must be a way to *specify* the circuit.
- For example, we define the class U_P-PSIZE, for **poly-size uniform** poly-size circuits, to be the language X such that (1) C_n has size $n^{O(1)}$ for each n, (2) there is a poly-time algorithm that outputs the circuit C_n, and (3) $\forall w$: $C_n(w) \leftrightarrow w \in X$.

U_L=PSIZE = U_P-PSIZE = P

- Let's prove that both these versions of **uniform circuit complexity** for poly-size are the same class, because both equal P, for poly-time.
- Let X be in U_P-PSIZE, and if I want to know whether $w \in X$, with $|w| = n$, how can I tell?
- I can construct the circuit C_n, then evaluate the result $C_n(w)$, using the Circuit Value Problem we showed to be in poly-time.
- These two poly-time operations are in P. And since $L \subseteq P$, U_L-PSIZE is also contained in P.

The Class NC

- So now that we can define uniform circuit classes, we are ready to define our class of languages that are highly parallelizable.
- The class NC is the subset of U_L-PSIZE that has circuits of only *poly-log* depth, still with poly-size.
- Remember that poly-log(n) means $(\log n)^{O(1)}$, any constant power raised to O(log n). Asymptotically, any poly-log function is $o(n^{\varepsilon})$ for any $\varepsilon > 0$.
- It's perhaps surprising that lots of interesting languages are in this seemingly small class.

The Class NC^0

- The class NC^0 consists of fan-in circuit families of depth O(1).
- These are exactly the functions which depend on only O(1) of the input values.
- It's easy to rule out languages from NC^0, as long as we can show that the output depends on an unbounded number of the n input values as n increases.

The Class AC^0

- In the class AC^0, any output bit may depend on any input bit.
- Here's an example of an AC^0 function — take two $n \times n$ boolean matrices A and B, and output the product AB. (In a decision language, we could return one of the bits of the product.)
- Since $(AB)_{i,j}$ is the OR of $A_{i,k} \wedge B_{k,j}$ for all k from 1 to n, the computation uses only O(1) depth using unbounded fan-in, one for OR, one for $\wedge$.

The Class NC^1, included in L

- The next class in the hierarchy is NC^1, with $O(\log n)$ depth, poly-size, and fan-in $O(1)$.
- Here we start getting some interesting upper bound results in this class.
- In the following lecture we'll prove that any regular language is in NC^1.
- We can also do some interesting integer arithmetic in NC^1, like MAJORITY and binary multiplication.

The Class AC^1, containing NL

- Now we connect L and NL to the class AC^1. We'll show that $NL \subseteq AC^1$, with the circuit family to be U_L-uniform. (The circuit family can be shown to be even more uniform than that, given the right definitions.)

- We remember that the PATH language (G, s, t), where G a directed graph and s and t nodes such that there is a path from s to t, is complete for NL under L reductions.

PARITY in NC^1

- The PARITY problem is to input n bits and determine whether there are an even or odd bits set to 1. The language PARITY is $\{w: |w|_1 \text{ is odd}\}$, or $\{w: \text{XOR of } w_i = 1\}$.
- As we mentioned, by a harder theorem, this problem cannot be solved in AC^0, even if the circuit family can be totally non-uniform. But it's not hard to show that PARITY is in NC^1.

Regular Languages in NC^1

- So it's a natural question to compare the "easy" languages, from our study of regular languages, to other classes of languages.
- What if D is a DFA, and X is the language L(D). Can we build a circuit family with polynomial size and O(log n) depth with fan-in 2? It turns out that we can.

Integer Arithmetic in NC^1

- As we showed earlier, we can add two binary numbers in AC^0, but not in NC^0.
- What we'd like to do is **iterated addition**, where say we are given n n-bit numbers and want to compute their sum (in n+logn bits, as it happens).
- This would be useful for two other problems we'd like to solve: MAJORITY and MULTIPLICATION.

MAJORITY and MULTIPICATION

- The MAJORITY problem is to take n bits and ask whether there are more 1 bits than there are 0's in the string. (Ties are not a majority of 1's.)
- This can be done with iterated addition. Just treat each bit as a number equal to 0 or 1, and compute the sum of these n numbers.

A Trick for Iterated Addition

- But if we tried to add n numbers together with the method we used to add two of them, we would make a tree of depth O(log n), which would get us depth O(log n) *if we used unbounded fan-in*. The tree of AC^0 computations would require AC^1.
- Using fan-in 2 to add two numbers, we'd get a tree of NC^1 operations to get NC^2.

A Trick for Iterated Addition

- So here's a trick. We change the representation for numbers to **signed digit base four (SDB4)**. One consequence is that there are different strings to represent the same number.
- For example, $23'2'13$ represents the number $2{\cdot}4^4$ - $3{\cdot}4^3$ - $2{\cdot}4^2$ + $1{\cdot}4^1$ · $3{\cdot}4^0$ = 512 - 192 - 32 + 4 + 3 = 295. The same number could be $112'21'$ = 256 + 64 - 32 + 8 - 1 = 295.

Boolean Circuits and Formulas

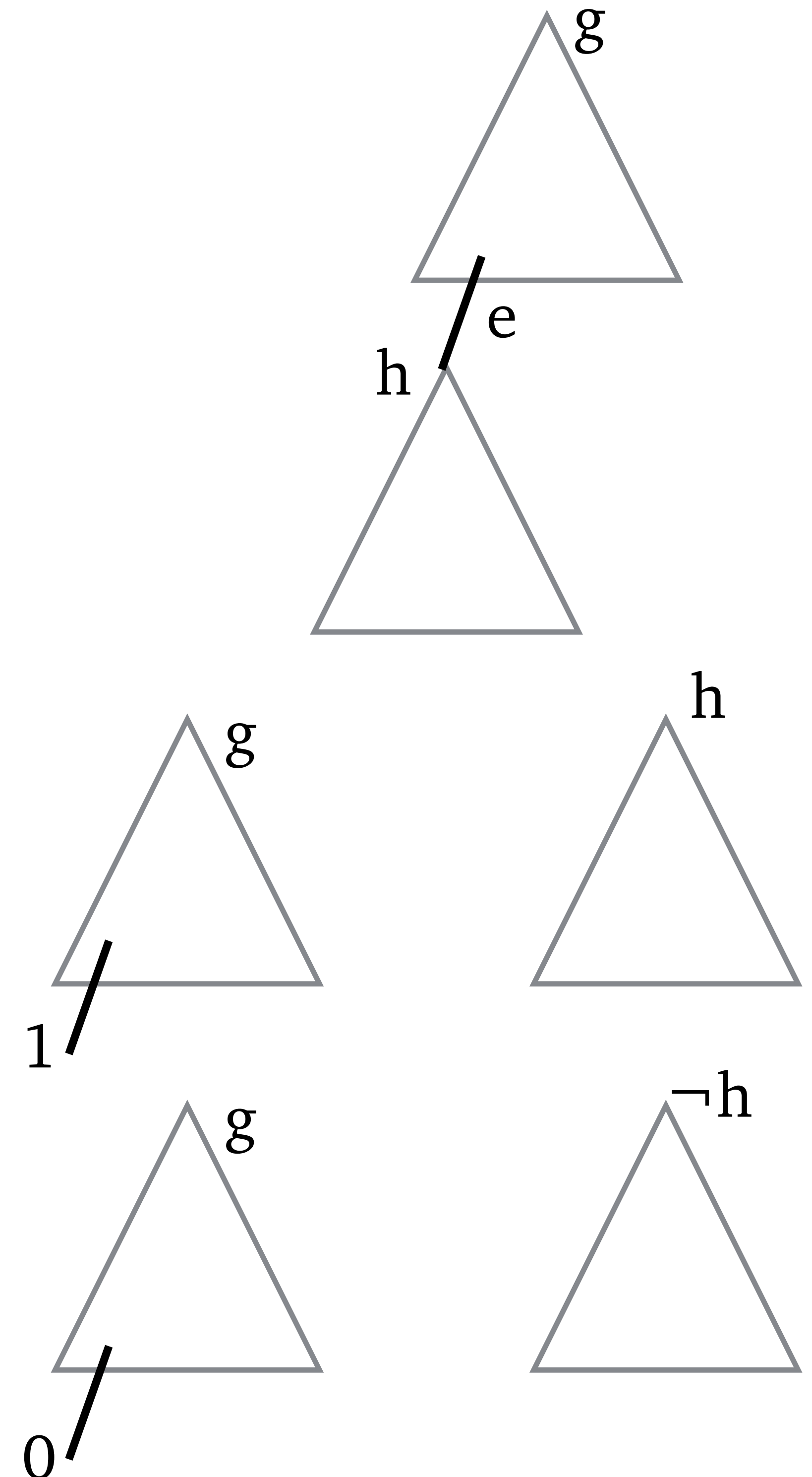
- Remember that each node of a boolean circuit is either an input gate or is computed by a gate from one or more earlier computed gates. Its **fan-in** is the number of gates whose values are used.
- It is possible that every circuit has **fan-out** of 1, so that the circuit is a **tree** or a **formula** instead of just a **directed acyclic graph**.
- It's a natural question whether circuits become weaker if they must be formulas.

Balancing Trees in NC^1

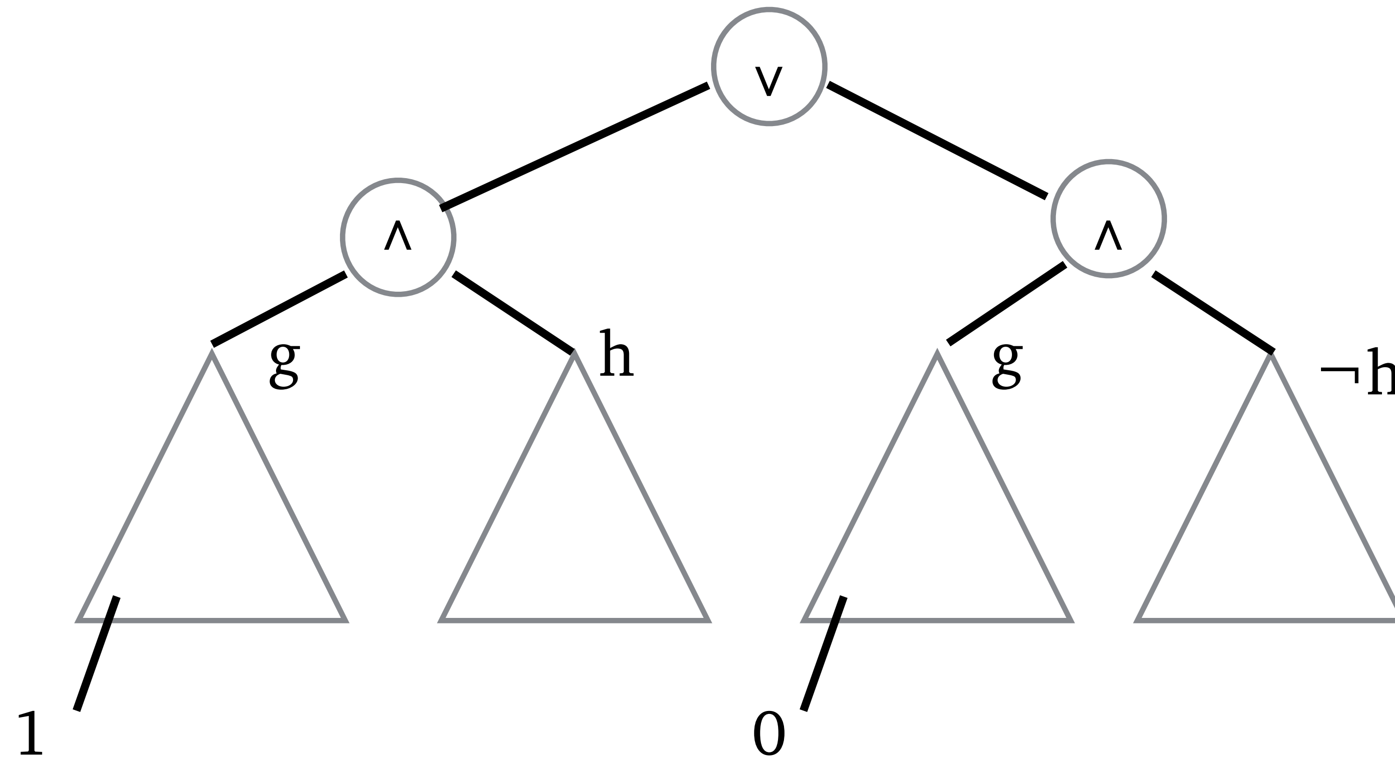
- An arbitrary poly-size circuit could have very large depth, almost as large as its depth. So we'll alter it somehow in this case to reduce its depth while maintaining an equivalent function.
- We'll also simplify the circuit to have only AND and OR gates, except for input gates for literals (variables and negated variables). You'll consider on HW#6 how any poly-size circuit can be put in this form.

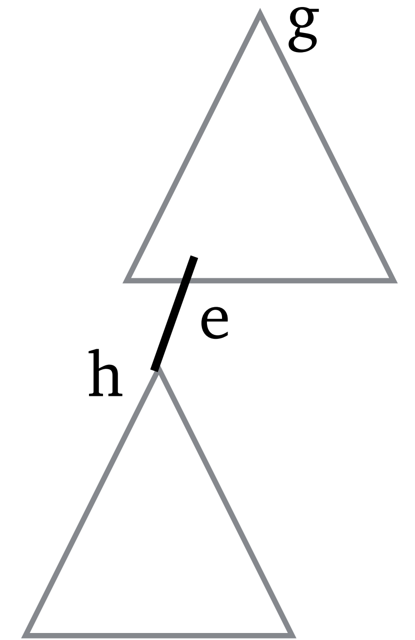
Working Toward Balance

- We can now draw the tree with h as g's ancestor, with $w/4 \leq w(h) \leq 3w/4$.
- We'll now build a more balanced tree, with slightly greater size, but with the same function as the original tree.
- Here are some pieces.



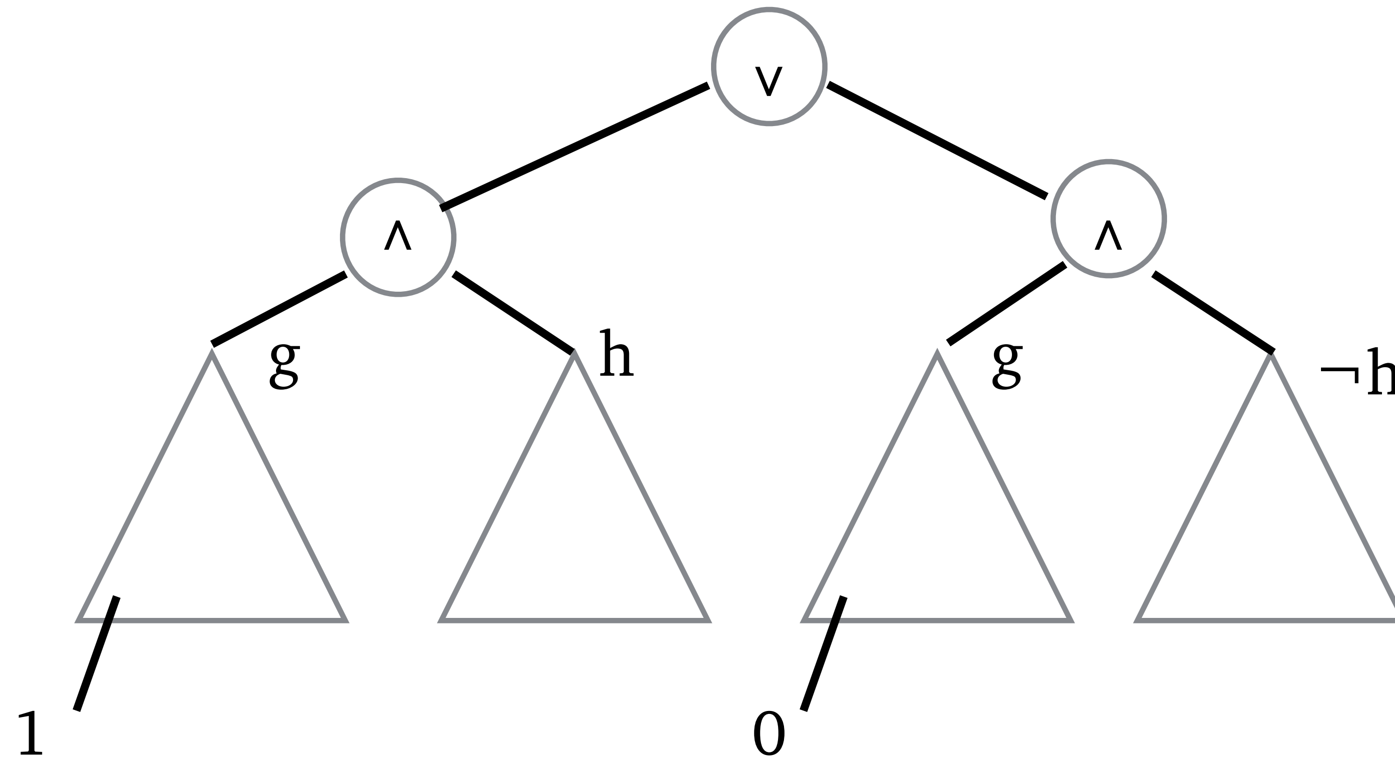
Working Toward Balance

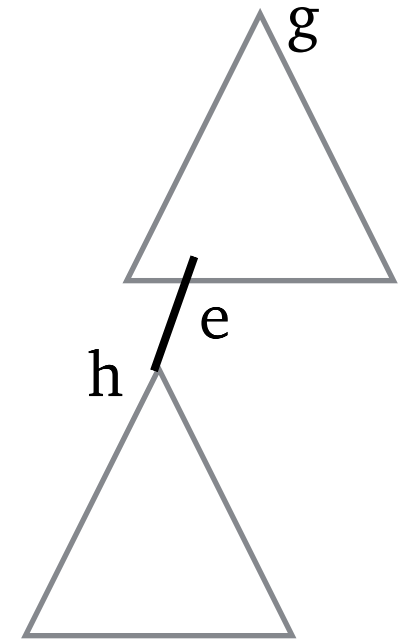




- Let's convince ourselves that the new tree has the same function as the original tree.
- There are two cases when h is true and when h is false.

Working Toward Balance





- Each of these four pieces has weight $\leq 3w/4$.
- The total weight is at most $4(3w/4)+5 = 3w+5$.
- The new depth has added 3.

Alternating Logspace in P

- We saw that in the Alternation Theorem, languages with unlimited alternation with log space are exactly the languages in P.
- If we have no alternation in a log space TM, we get the language L.
- What we'll see today is what happens when we add *some* alternation to a log space TM.
- This turns out to give us a spectrum of classes from L through P, ranging across NC.

Constraints on AL Machines

- So how can we parametrize AL machines to limit their amount of alternation?
- One natural method is to count the **number of alternations** in the ATM computation.
- In the game semantics, we count every time that a White move is followed by a Black move, or vice versa. In the version with existential states and universal states, we count an alternation every time we change from one state type to the other.

The General Result

- The two main theorems are:
- ATM's with log space and $O(\log^i n)$ alternations are equivalent to AC^i, for $i \geq 1$
- ATM's with log space and $O(\log^i n)$ time are equivalent to NC^i, for $i \geq 2$.
- We're not going to prove this theorem here, but we'll tell you about some special cases, and give an idea about how the proof works.

Solving Circuits with ATM's

- But without the technical details, the main idea of simulating circuits with alternation is pretty much what we saw in the Alternation Theorem, with the Circuit Game.
- We start the game at the output node. When the current node is an $\vee$, White picks a child of that node. When it is $\wedge$, Black picks a child. The game ends when at an input, when White wins if the value is true and Black if it is false.

Solving ATM's with Circuits

- What if we have an ATM, with the time or alternations constraints, and we want to design a circuit with the proper size and depth?
- Essentially we make a gate for each state of the ATM game. There are only polynomially of them because it's log space.
- We assign edges for the circuit based on the moves that White and Black have available.