Intractability I, II, III (Lecture 13-15 Notes)

We've seen many interesting algorithms for a variety of problems. We've tried to get algorithms with running time $O(n^k)$ for inputs of size n, where k is a constant independent of n: These are **polynomial time algorithms.** Examples:

- *s-t Shortest Path:* Given an unweighted graph $G = (V, E)$ and two nodes $s, t \in V$, find a shortest (simple) path from s to t. From 6.1210, we can use BFS, works in $O(|E|)$ time!
- *s-t Longest Path:* Given an unweighted graph $G = (V, E)$ and two nodes $s, t \in V$, find a longest path from s to t. The fastest algorithm is Bjorklund '2010 which is $O(1.66^{|V|})$ for undirected graphs. Exercise: Get an $O(2^{|V|} \cdot |E|)$ time algorithm for directed graphs. **This is exponential time! Why is this so hard?**
- *Maximum Matching:* Given an undirected bipartite graph $G = (V, E)$, find a largest set of edges that share no endpoints. Can be solved in polytime by using max-flow! The 2022 algorithm gives $O(|E|^{1+o(1)})$ time.
- *3D Matching:* Given a *tripartite* hypergraph $\mathcal{H} = (V, E)$ where $V = V_1 \sqcup V_2 \sqcup V_3$ are vertices and $E \subseteq V_1 \times V_2 \times V_3$ are *hyperedges* of 3 vertices each, find a largest set of hyperedges that share no endpoints. Can be solved in $O(3.52^{|V|})$ time, **again exponential!**
- *Linear Programming (LP):* Given an n-by-n real matrix A and vectors $\mathbf{b}$ and $\mathbf{c}$, find a real vector $\mathbf{x}$ maximizing $\mathbf{c}^\top \mathbf{x}$ subject to $A\mathbf{x} \leq \mathbf{b}$. This is solvable in $O(n^{2.372})$ time.
- *Integer Programming (IP):* Given an n-by-n integer matrix A and integer vectors $\mathbf{b}$ and $\mathbf{c}$, find an integer vector $\mathbf{x}$ maximizing $\mathbf{c}^\top \mathbf{x}$ subject to $A\mathbf{x} \leq \mathbf{b}$. This turns out to be **exponential time** even if all entries are constrained in the set $\{0, 1\}$!

Similar sounding problems can have extremely different running times!

Extremely interesting question: Can you get polytime algorithms for problems like IP, 3D-Matching and Longest Path? Today's answer: **Maybe not**, otherwise a huge open problem in theoretical computer science is resolved.

1 Decision, Search and Optimization Problems

- *Optimization Problem:* Given input I, find an object satisfying some property and having maximum/minimum weight.
- *Search Problem:* Given input I and a value K, find an object satisfying some property with weight $\geq K$ or $\leq K$.
- *Decision Problem:* Given input I and a value K, decide whether there exists an object satisfying some property with weight $\geq K$ or $\leq K$. It has a yes/no output.

Examples:

	Optimization	Search	Decision
s-t shortest	In: G, s, t Out: path with min weight	In: G, s, t, K path w. weight $\leq K$	In: G, s, t, K bool(exists path w. weight $\leq K$)
s-t longest	In: G, s, t Out: path with max weight	In: G, s, t, K path w. weight $\geq K$	In: G, s, t, K bool(exists path w. weight $\geq K$)
Maximum Matching	In: G Out: a max matching	In: G, K matching w. size $\geq K$	In: G, K bool(exists matching w. size $\geq K$)

Can solve OPTIMIZATION $\Rightarrow$ Can solve SEARCH $\Rightarrow$ Can solve DECISION. From now on, we focus on decision problems.

Therefore, if there is no efficient algorithm for DECISION, then there is no efficient algorithm for SEARCH nor OPTIMIZATION.

2 Intractability

2.1 Polynomial Time (P)

Definition. A decision problem π is *solvable in polynomial time* if there exists an algorithm $\mathcal{A}$ and a constant c such that for every input x of size n, $\mathcal{A}(x)$ runs in $O(n^c)$ time, and returns YES if and only if $\pi(x) = \text{YES}$. The class of all decision problems solvable in polynomial time is denoted as **P**.

Note: We only consider deterministic polynomial time algorithms (without randomization) when defining **P**. There are other classes **RP, BPP** etc that allow randomization.

Almost everything mentioned in 6.1210/6.1220 is in **P**.

2.2 Nondeterministic Polynomial Time (NP, "Nifty Proofs")

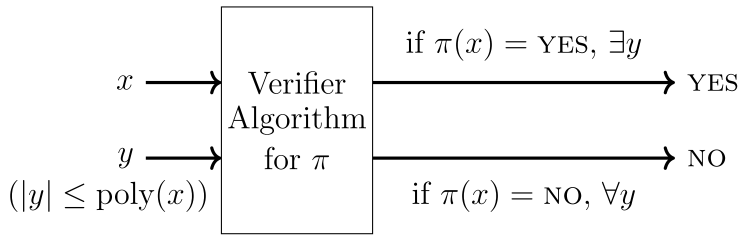
Informal Definition. A problem is in **NP** if for any instance whose answer is YES, we can provide a short proof that the answer is YES and this proof can be easily checked. (Note that we don't need proofs for NO-instances)

Definition. A decision problem π is in *nondeterministic polynomial time* if there exists an algorithm $\mathcal{V}_\pi$ and a constant c, c' such that

- $\mathcal{V}_\pi$ takes two inputs x, y, where $|y| \le n^{c'}$.
- $\mathcal{V}_\pi(x, y)$ runs in $O((|x| + |y|)^c)$ time.
- For every input x of size n, $\mathcal{A}(x)$ runs in $O(n^c)$ time, $\pi(x) = \text{YES}$ if and only if there exists a string y ($|y| \le n^{c'}$) such that $\mathcal{V}_\pi(x, y) = \text{YES}$.

The class of all decision problems in nondeterministic polynomial time is denoted as **NP**.

Here y is a **certificate** proving that x is a YES-instance of π.


x
y
$(|y| \le \text{poly}(x))$
Verifier Algorithm for π
if $\pi(x) = \text{YES}, \exists y$
YES
NO
if $\pi(x) = \text{NO}, \forall y$

Examples:

- *s-t Longest Path* $\in$ **NP**: We can let $\mathcal{V}_\pi((G, s, t, K), y)$ check that y is a simple path in G from s to t of length $\ge K$. This runs in poly$(|V|)$ time and $|y| \le |V|$. Whenever there is a simple path P of length $\ge K$, then $y = P$ triggers YES, otherwise any y triggers NO.

- *Linear Programming* $\in$ **NP**: We can let $\mathcal{V}_\pi((\mathbf{c}, \mathbf{b}, A, K), y)$ check that y encodes a vector $\mathbf{y}$ such that $\mathbf{c}^\top \mathbf{y} \geq K$ and $A\mathbf{y} \leq \mathbf{b}$.

Often the certificate is hinted in the NP problem itself! This is not always the case though.

Are any of these in **NP**?

- *Graph Isomorphism:* Given two graphs G_1 and G_2, is there a permutation of the vertices of G_1 to those in G_2 that makes G_1 into G_2?

  Answer. **NP**. Verifier takes in y as a permutation $G_1 \to G_2$.

- *Shortest Path:* Given a graph G and nodes s, t, return a shortest path from s to t.

  Answer. Not in **NP**. It is not a decision problem.

- *No-Long Path:* Given a graph G, nodes s, t and a number of K, is it true that there is no simple s-t path of length $\geq K$?

  Answer. This is the *complement* of an NP-problem: It is in **coNP**. Whether it is **NP** is unknown!

- *No-Short Path:* Given a graph G, nodes s, t and a number of K, is it true that every s-t path has length $> K$?

  Answer. This is in **P** (use BFS). We'll show that $\mathbf{P} \subseteq \mathbf{NP}$.

Theorem. $\mathbf{P} \subseteq \mathbf{NP}$.

Proof. Let $\pi \in \mathbf{P}$ be a problem, with an algorithm $\mathcal{A}_\pi$ that runs in poly($|x|$) time and returns yes if and only if $\pi(x) = \text{YES}$. Then we let $\mathcal{V}_\pi(x, y)$ ignore y and run $\mathcal{A}_\pi(x)$. This is in polynomial time and returns the correct answer anyway. ■

Definition. Denote **EXP** as the class of decision problems solvable in $2^{\text{poly}(n)}$ time.

Theorem. $\mathbf{NP} \subseteq \mathbf{EXP}$.

Proof Sketch. We run the verifier for all possible inputs y. Since $|y| \leq n^c$, this will take $\leq 2^{n^c}$ time. Then we return YES if some call says YES. ■

3 Believed State of the World

Known:

- $\mathbf{P} \subsetneq \mathbf{EXP} \subsetneq \mathbf{R}$ where $\mathbf{R}$ is the class of *decidable* problems.
- There are *undecidable* problems such as the HALTING Problem.

Big Open Problem: Is $\mathbf{P} = \mathbf{NP}$?

How do we address this?

3.1 Success of NP-Completeness

There exists concrete problems in **NP** called *NP-complete* such that if you ever develop a polynomial time algorithm for any one of these problems, then **P** = **NP**!

Turns out Longest Path, IP, Circuit SAT (in Lec 15) and CNF SAT (Lec 14) are NP-Complete Problems.

To prove that a problem π is NP-complete,

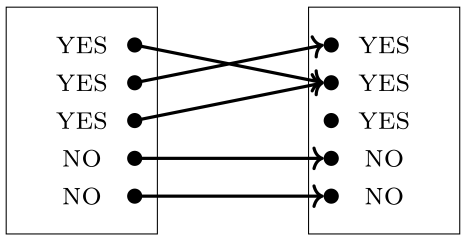
1. Show that $\pi \in \mathbf{NP}$
2. Show that π is *NP-hard*

where a problem π is NP-hard if a polynomial time for π can be used to solve any NP problem in polynomial time. We do this by exhibiting "many-one polynomial time reductions".

4 Reductions

Definition. Let Q and π be decision problems. A *(many-one) polynomial time reduction* from Q to π is an algorithm $\mathcal{R}$ that takes as input an instance x of Q, runs in poly($|x|$) time, and returns an instance y of π such that $Q(x) = \pi(y) \in \{\text{YES}, \text{NO}\}$. If such a reduction exists, we write $Q \leq_P \pi$.

Note that $y = \mathcal{R}(x)$ has poly size in $|x|$ since $\mathcal{R}$ runs in poly time.



Instances of Q Instances of π

Claim. If $Q \leq_P \pi$ and $\pi \in \mathbf{P}$. then $Q \in \mathbf{P}$.

Proof. Let $\mathcal{A}$ be a poly-time algorithm for π. Let $\mathcal{R}$ be a many-one poly-time reduction from Q to π. Here is an algorithm for Q:

1. Given instance x of Q, run $\mathcal{R}$ on x to get an instance $y = \mathcal{R}(x)$ of π.
2. Solve y using $\mathcal{A}$ and return the answer.

The runtime is polynomial since $\mathcal{R}$ runs in poly($|x|$) time, $|y|$ is poly($|x|$) and hence $\mathcal{A}$ runs in poly(poly($|x|$)) time. ■

Definition. A problem π is NP-hard if every problem Q in **NP** can be many-one poly-time reduced to π. A decision problem is NP-complete if it is in **NP** and is NP-hard.

By the above claim, if an NP-hard problem has a poly time algorithm, then **P** = **NP**!

4.1 Transitivity

Once we have one NP-hard problem, to show its NP-hardness it will suffice to reduce it to other problems. We won't have to reduce from every NP-problem ever again! To see why this is the case, we will show a nice property of polynomial time reductions.

Claim. If $A \leq_P B$ and $B \leq_P C$ then $A \leq_P C$.

Proof Sketch. Successively run the reduction algorithms, the runtime will be poly(poly($|x|$)) which is still polynomial. ∎

5 SAT

Today we **assume that 3-SAT is NP-Complete** and reduce it to other problems.

5.1 CNF-SAT (Boolean Satisfiability in Conjunctive Normal Form)

Input: A CNF-Formula F on variables $x_1, \cdots, x_n$.

Output: YES if there is a Boolean assignment to $x_1, \cdots, x_n$ such that $F = 1$; NO otherwise.

A **CNF-Formula** on $x_1, \cdots, x_n$ (Boolean variables) is of the form

$$C_1 \wedge C_2 \wedge \cdots \wedge C_m$$

where each C_i (called a clause) is of the form

$$C_i = \ell_{i1} \vee \ell_{i2} \vee \cdots \vee \ell_{ip_i}$$

and each ℓ_{ij} (called a literal) is some x_k or $\neg x_k$ (Boolean negation)

Example. $(x_1 \vee \neg x_9 \vee x_7) \wedge (x_3 \vee x_5 \vee \neg x_7 \vee x_2)$ is a CNF-Formula.

Here, $\wedge$ is the Boolean AND operation and $\vee$ is the Boolean OR operation.

Example. $F = (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$ on the assignment $(x_1, x_2, x_3) = (1, 0, 1)$ evaluates to

$$(1 \vee \neg 0) \wedge (\neg 1 \vee 1) = 1 \wedge 1 = 1.$$

We see that F evaluates to 1 if and only if for every clause C_j some literal evaluates to 1 on the assignment.

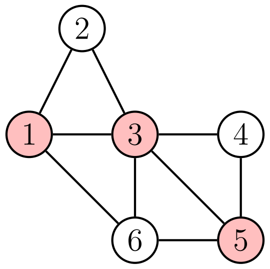
If k is an integer, the k-SAT problem is the CNF-SAT problem restricted to CNFs with $\leq k$ literals in every clause.

- 1-SAT is in **P** (exercise).
- 2-SAT is in **P** (more interesting exercise).
- 3-SAT is NP-Complete (next time!).
- k-SAT ($k \geq 3$) is NP-Complete.

6 Vertex Cover

A *vertex cover* of a graph $G = (V, E)$ is a subset $S \subseteq V$ such that every edge has at least one endpoint in S. Here $|S|$ is the *size* of the vertex cover.

For example, $S = \{1, 3, 5\}$ is a vertex cover of size 3 in this diagram:



6.1 VC (Vertex Cover Problem)

Input: $G = (V, E)$ and integer K.

Output: YES if there is a vertex cover of size $\leq K$; NO otherwise.

Note that VC $\in$ **NP**: Use a subset $S \subseteq V$ as the certificate, and check that every edge has at least one endpoint in S.

Theorem. 3-SAT $\leq_P$ VC (implies VC is NP-Complete)

Proof. Take a 3CNF-Formula

$$\varphi = (\ell_{11} \vee \ell_{12} \vee \ell_{13}) \wedge \cdots \wedge (\ell_{m1} \vee \ell_{m2} \vee \ell_{m3})$$

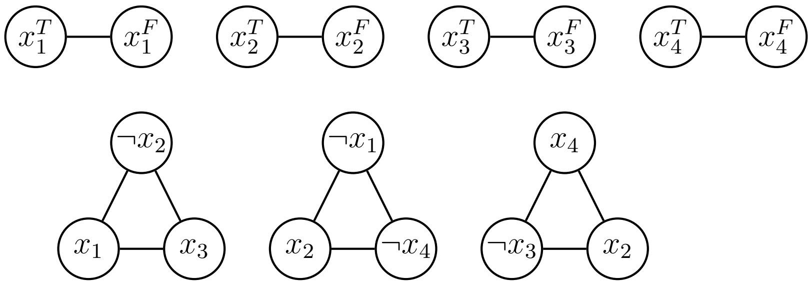
where ℓ_{ij} is some x_k or $\neg x_k$ and $(x_1, \cdots, x_n)$ are the inputs.

We'll create a graph G_φ and an integer K such that G_φ has a vertex cover of size $\leq K$ if and only if φ is satisfiable.

1. For every variable x_i create two vertices connected by an edge (x_i^T, x_i^F).
   (This will mean that either x_i^T or x_i^F needs to be in any vertex cover)

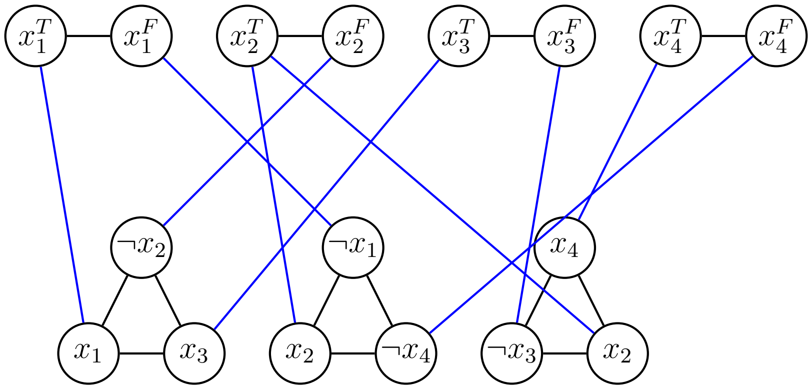
2. For every clause $(\ell_{j1}, \ell_{j2}, \ell_{j3})$ create 3 vertices connected in a triangle.
   (This means for every clause at most one of its literals is not in the VC)

   Example: $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_4) \wedge (x_4 \vee \neg x_3 \vee x_2)$ means



3. For each ℓ_{jk} in clause j,

- If $\ell_{jk} = x_p$, connect node ℓ_{jk} to x_p^T
- If $\ell_{jk} = \neg x_p$, connect node ℓ_{jk} to x_p^F



4. And set $K = n + 2m$ (recall n is the number of inputs; m is the number of clauses).

A few properties to note:

1. For each (x_i^T, x_i^F), one of the endpoints must be in VC to cover this edge.
2. For each $(\ell_{i1}, \ell_{i2}, \ell_{i3})$, at least two of the nodes must be in VC, so at most one is *not* in the VC.
3. $|V| = 3m + 2n$ and $|E| = n + 3m + 3m = \text{poly}(m, n)$ and it takes $\text{poly}(m, n)$ time to build graph.
4. Size of any VC is at least $n + 2m$ (due to properties 1 and 2).

By setting $K = n + 2m$, we have that *any VC of size $\leq K$ has size exactly K (and will have exactly 2 nodes from each triangle and exactly one node from each x^T, x^F)*

We now prove the equivalence of φ and G_φ.

- Suppose φ has a satisfying assignment $(x_1, \cdots, x_n) = (\overline{x_1}, \cdots, \overline{x_n})$.

  1. For each $\overline{x_i} = 1$, we put $x_i^T \in S$, otherwise $x_i^F \in S$.
  2. For each clause $(\ell_{j1}, \ell_{j2}, \ell_{k3})$, (at least) one of the ℓ_{jk} evaluated satisfies the clause. We put the two nodes corresponding to the other two literals in S.

  Claim: S is a vertex cover with size K.

  Proof. First, $|S| = 2m + n = K$. All the black edges are covered by S by the above procedure. For the each blue edge $(\ell_{jk}, x_p^\bullet)$, we have $\ell_{jk} \notin S \Rightarrow x_p^\bullet \in S$ from step 2, so we are done. ■

- Suppose G_φ has a VC T of size $\leq K$. Then T contains exactly one of each x_i^T, x_i^F and exactly 2 from each clause triangle. We define $x_i = 0, 1$ according to whether x_i^F or $x_i^T \in S$. We will show that this satisfies φ: For each clause, exactly one of its literals is not in T; that means the $x_i^\bullet$ connected to that literal must be in T, hence satisfying that literal and hence the clause. ■

Therefore VC is NP-Complete!

6.2 Subset-Sum

SUBSET-SUM: Given a set S of n positive integers and an integer t (written in binary), does there exist $A \subseteq S$ such that $\sum_{a \in A} a = t$?

This is clearly in **NP**: Certificate being $y = A$, verifier then checks the sum.

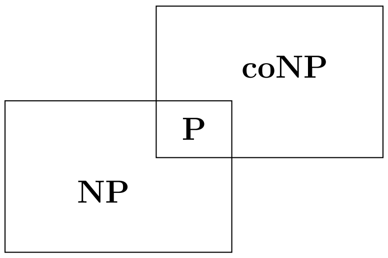
(Optional) We will prove VC $\leq_P$ SUBSET-SUM at the end of the notes.

But **WAIT!** Using dynamic programming we can solve SUBSET-SUM in $O(nt)$ time. Isn't this polynomial? Is **P** = **NP**? Well, $O(nt)$ is actually *pseudopolynomial*. The integers are specified in binary, so the input representation of t is of size $\log t$ and hence $O(nt)$ is actually *exponential* in the input size!

7 A bit on coNP

coNP is the class of decision problems whose complements are in **NP** (the YES and NO are flipped)

We see that **P** $\subseteq$ **coNP** by using the same proof but flipping the answers. So



It is an OPEN question that **NP** = **coNP**! Believed no. It is also OPEN that **P** = **NP** $\cap$ **coNP**.

7.1 Factoring

FACTORING: Given two integers n, k written in binary, $1 < k < n$, is there some prime p such that $p \mid n$ and $k \leq p < n$?

If FACTORING is in **P**, we can factor numbers in polytime: RSA Cryptography will be broken!

7.2 Primes

PRIMES: Given n in binary, is n a prime?

Turns out PRIMES $\in$ **P**. (Agrawal, Kayal, Saxena 2004)

Theorem. FACTORING $\in$ **NP** $\cap$ **coNP**.

Proof.

- In **NP**: Certificate y being a prime; check y is prime, $k \leq y < n$ and $y \mid n$.
- In **coNP**: Certificate y being a prime factorization; check that y is in the prime factorization of n (multiply to see if we get n), then verify each prime in y is $< k$. We used the uniqueness of prime factorization! ■

Note: How large is y? Each prime in y has $\leq \log n$ bits, and the number of primes in the prime factorization is $\leq \log n$, so $|y| \leq \log^2 n$. Therefore the procedure is poly($\log n$) time!

The believed state of the world is that FACTORING $\notin$ **P**, though.

8 Optional: VC $\leq_P$ Subset-Sum

Given $G = (V, E)$ and integer k, want to create an instance of SUBSET-SUM. Denote $n = |V|$ and $m = |E|$. We assume $E = \{0, \cdots, m-1\}$, otherwise just relabel.

Here is how we will construct an instance (S, t) of SUBSET-SUM:

1. S starts off as $\varnothing$.
2. For every $e \in E$, add $\mathfrak{b}_e = 4^e$ to S. These will be called *edge numbers*.
3. For every $v \in V$, add $\mathfrak{b}_v = 4^m + \sum_{e:v\in e} 4^e$ to S. These will be called *vertex numbers*.
4. Set $t = k \cdot 4^m + 2\sum_{e\in E} 4^e$.

All numbers have $O(m)$ bits. Let's consider them in base-4 representation (in reverse order):

- $\mathfrak{b}_e$: an $(m+1)$-length vector with a 1 in position e (and 0 everywhere else).

$$\underbrace{00\cdots00}_{e}1\underbrace{00\cdots00}_{m-e}$$

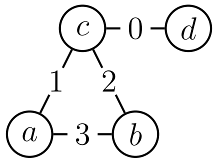
- $\mathfrak{b}_v$: an $(m+1)$-length vector with a 1 in position m and all positions e such that $v \in e$.

$$00\cdots00100\cdots00100\cdots001$$

- t: an $(m+\log_4 k)$-length vector with a 2 in all positions $e \in E$ and such that substring from positions m to $m + \log_4 k$ is exactly the base-4 representation of k.

$$\underbrace{22\cdots22}_{m}\underbrace{k_1k_2\cdots}_{\log_4 k}$$

For example, the following graph with $k = 3$ reduces to $S = \{\text{edge numbers}\} \cup \{\text{vertex numbers}\} = \{4^0, 4^1, 4^2, 4^3\} \cup \{4^4 + 4^1 + 4^3, 4^4 + 4^3 + 4^2, 4^4 + 4^0 + 4^1 + 4^2, 4^4 + 4^0\}$ and $t = 3 \cdot 4^4 + 2(4^0 + 4^1 + 4^2 + 4^3)$.



In base-4 representation: $S = \{10000, 01000, 00100, 00010\} \cup \{01011, 00111, 11101, 10001\}$ and $t = 22223$.

Claim 1. If C is a VC of size k for G, then there exists a SUBSET-SUM solution.

Proof. Construct $A = \{\mathfrak{b}_v\}_{v\in C} \cup \{\mathfrak{b}_e \mid e \text{ covered by } C \text{ exactly once}\}$ in poly time. (Note that every edge is covered by C either once or twice) Then

$$\begin{aligned}\sum A &= \left[\sum_{v\in C} \mathfrak{b}_v\right] + \left[\sum_{e \text{ covered once}} \mathfrak{b}_e\right] \\ &= \left[|C|4^m + \sum_{e \text{ covered twice}} 2\cdot 4^e + \sum_{e \text{ covered once}} 4^e\right] + \left[\sum_{e \text{ covered once}} 4^e\right] \\ &= k \cdot 4^m + 2\sum_{e} 4^e = t\end{aligned}$$

and hence A is a solution to Subset-Sum.

Claim 2. Say $A \subseteq V, E' \subseteq E$ such that $\sum_{a \in A} \mathfrak{b}_a + \sum_{e \in E'} \mathfrak{b}_e = t$ (a solution to Subset-Sum). Then we claim that A is a vertex cover of size k of G.

Proof. Denote $\mathbf{1}_{\text{event}} \in \{0,1\}$ as the indicator variable of an event. $\sum_{a \in A} \mathfrak{b}_a + \sum_{e \in E'} \mathfrak{b}_e = t$ expands to

$$|A|4^m + \sum_{e \in E} \begin{pmatrix} \text{\# of times } e \\ \text{covered by } A \end{pmatrix} \cdot 4^e + \sum_{e \in E} \mathbf{1}_{e \in E'} \cdot 4^e = k \cdot 4^m + 2 \sum_{e \in E} 4^e$$
$$|A|4^m + \sum_{e \in E} (\leq 2) \cdot 4^e + \sum_{e \in E} (\leq 1) \cdot 4^e = k \cdot 4^m + 2 \sum_{e \in E} 4^e$$

Now consider what is happening in base-4 representation. On the LHS, in positions 0 to $m-1$, we are adding digits ≤ 2 to digits ≤ 1, whereas in positions m onwards only $|A|$ (in base-4) is a contribution. So there are **no carry-overs** for the addition in the LHS. Hence we can compare the sum digits-by-digits:

$$\begin{pmatrix} \text{\# of times } e \\ \text{covered by } A \end{pmatrix} + \mathbf{1}_{e \in E'} = 2 \qquad (\forall e \in E)$$
$$|A| \cdot 4^m = k \cdot 4^m$$

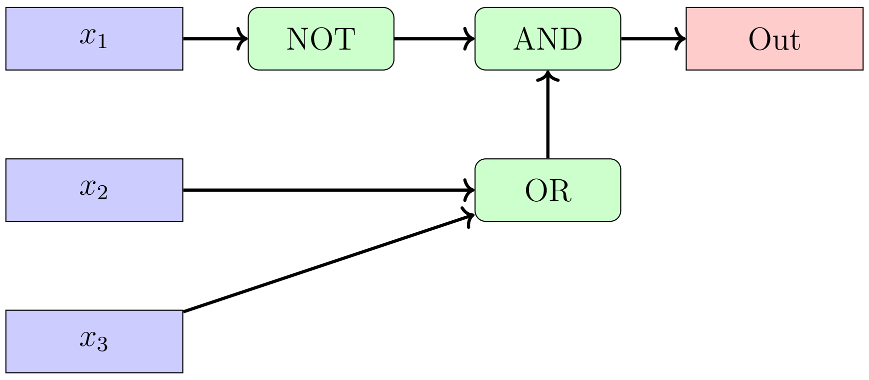
The first equalities tell us that # of times e covered by A is always ≥ 1, for all $e \in E$. The second equality tells us that $|A| = k$. Therefore A is a vertex cover of size k.

Hence, by Claim 1 and Claim 2, the instance of Subset-Sum that reduction algorithm gives is yes-instance if and only if the input (an instance of VC) is a yes-instance. Hence VC $\leq_P$ Subset-Sum and therefore Subset-Sum is NP-Complete. ■

9 Boolean Circuits

Boolean gates: AND, OR, NOT.

A **circuit** is a directed acyclic graph (no feedback) where the nodes are gates and the n inputs, and there is one output gate. The edges are wires from inputs to gates and wires between gates.



Circuit Evaluation: Given Boolean values $(0,1)$ to the variables $x_1, \cdots, x_n$, plug them in and propagate to see the output. This takes **linear time** as we can use a topological order and dynamic programming.

Circuit-SAT: Given a Boolean circuit $\mathcal{C}$ on n inputs, does there exist an assignment of inputs so that $\mathcal{C}$ evaluates to 1?

Claim: Circuit-SAT $\in$ **NP**.

Proof. The verifier just evaluates the circuit with y being the input assignment. ■

Today we will prove

Cook-Levin Theorem. Circuit-SAT Is NP-complete.

We have to exhibit a many-one poly-time reduction from *every problem in* **NP** to Circuit-SAT.

We will do this implicitly (not by going through all problems in **NP**) by using the *definition* of **NP**, namely the existence of a verifier. We haven't formally specified a machine model (Turing Machine, RAM etc.). Fortunately, the Cook-Levin Theorem holds for *any reasonable machine model.*

9.1 Assumptions on the Machine Model

- A computer **program** is stored in memory as a sequence of instructions encoding an **operation** to be performed, **addresses** of operands in memory, and an **address** where the result is stored.
- A program **counter** keeps track of which instruction is next. This is automatically incremented after each instruction. Therefore execution is usually sequential, except that an execution can also write to the program counter (allowing *for loops* and *conditional branches*).
- At any point of the execution, memory holds the entire state of the computation (program, counter, working storage, and various bits of state for bookkeeping).
- If an algorithm has running time $O(p(n))$, we only need $O(p(n)\underbrace{\log p(n)}_{\substack{\text{pointer size}\\ \text{in RAM}}})$ space.
- **Main Assumption of the Model:**
  Let L_i be the state in memory at step i of a run of the algorithm. Between state L_i and L_{i+1}, a simple operation is performed, transforming L_i to L_{i+1}. For a fixed input size n (and hence runtime $p(n)$), there is a *fixed Boolean circuit* M that transforms L_i to L_{i+1} for each i, and M can be constructed in poly$(p(n))$ time. (Think of M as being *embedded in hardware*)

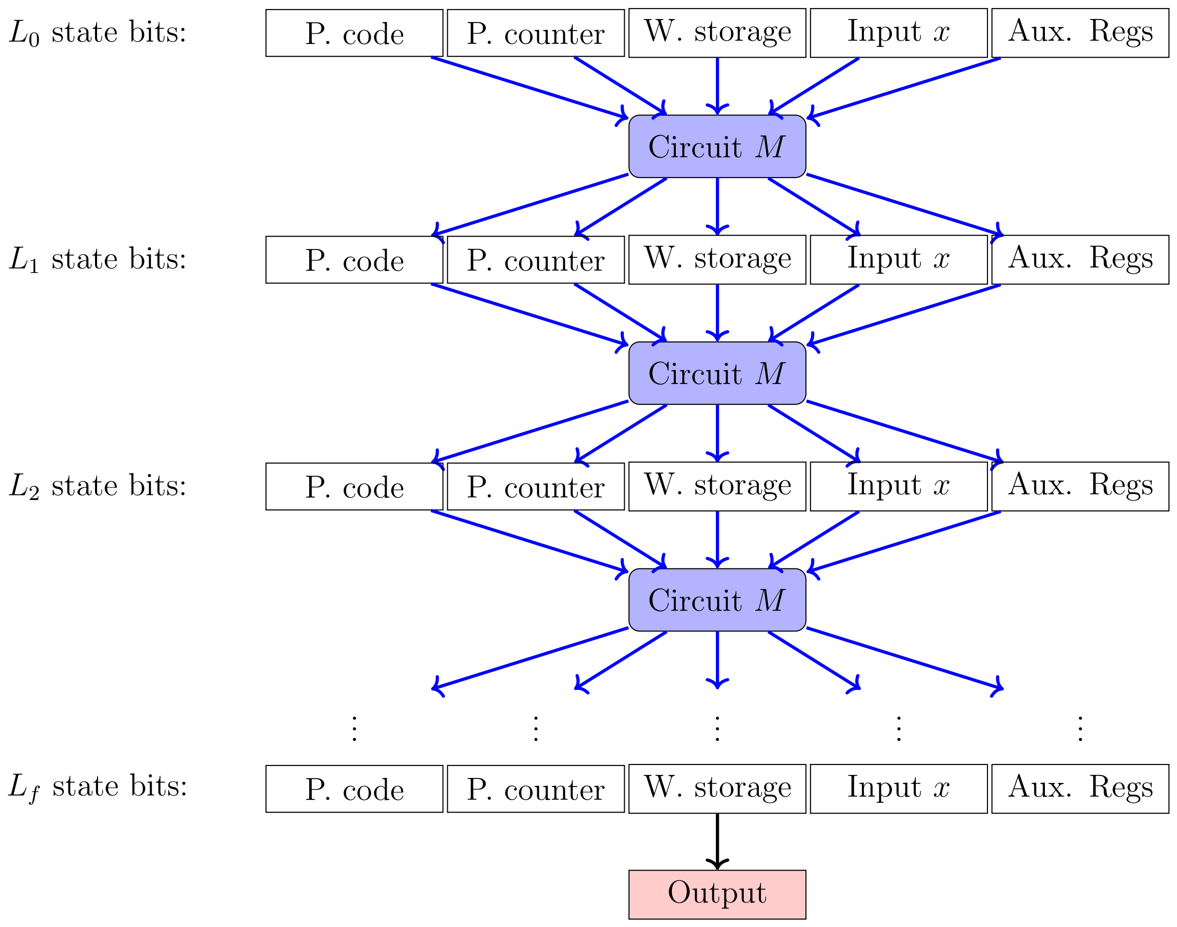
Given the Main Assumption, we can prove that an entire program can be converted into a circuit of polynomial size given a fixed input size:

Theorem. Suppose Q is a decision problem that can be solved by a program P in $p(n)$ time on inputs of size n (in binary). Then for every fixed n there is a Boolean Circuit C_n that can be constructed in $O(\text{poly}(p(n)))$ time such that for every input $x_1, \cdots, x_n$ of Q,

$$Q(x_1, \cdots, x_n) = C_n(x_1, \cdots, x_n).$$

Note: Programs work on arbitrary size inputs but are of constant size, whereas circuits work on fixed size inputs and their size is polynomial in the program's runtime on size n inputs.

Proof Sketch. Any run of P on size n inputs and runtime $p(n)$ looks like this:



This is combines to an entire Boolean circuit constructible in $O(|M| \cdot p(n)) = O(\text{poly}(p(n)))$ time. ■

9.2 Proof of Cook-Levin

Let A be any problem in **NP**. Let $\mathcal{V}_A$ be its verifier algorithm. Recall that if x is a YES-instance to A, then there exists a proof y with $|y| = \text{poly}(|x|)$ and $\mathcal{V}_A(x, y) = \text{YES}$; and if x is a NO-instance then $\mathcal{V}_A(x, y) = \text{NO}$ for all y.

Since $\mathcal{V}_A$ is a poly-time algorithm, for every N there exists a circuit C_N such that C_N simulates $\mathcal{V}_A$ on all inputs (x, y) of size N and C_N can be computed in poly(N) time.

Consider the following reduction:

```
$\mathcal{R}(x)$:                                    ▷ $x$ is an instance of $A$ of size $n$
  (1) Let $\mathcal{V}_A(x, y)$ be the verifier algorithm taking input $x$ of length $n$ and $y$ of length $p(n)$.
  (2) Construct the corresponding Boolean circuit $C_N(x_1, \cdots, x_n, y_1, \cdots, y_{p(n)})$ where $N = n + p(n)$.
  (3) Hardcode[1] $x = \overline{x_1 \cdots x_n}$, and return the circuit $C(y_1, \cdots, y_{p(n)}) = C_N(\overline{x_1}, \cdots, \overline{x_n}, y_1, \cdots, y_{p(n)})$.
```

Then x is a YES-instance of $A \Leftrightarrow$ there exists y for $\mathcal{V}_A \Leftrightarrow$ there exists a satisfying input to C. ■

[1]means to replace each x_i with its Boolean value

10 3-SAT (and thus CNF-SAT) is NP-Complete

Assume C is an instance of Circuit-SAT with n variables, t gates and m wires. We can safely assume that every gate and every variable has a path to the output gate, so $m \geq n + t - 1$ (C is connected) and the size of C is $O(m)$. We will reduce Circuit-SAT to 3-SAT in two steps:

1. Convert C into a simpler circuit C_1 that outputs a conjunction ($\wedge$) of $O(m)$ "clauses" of the form $(x), (x \Leftrightarrow \neg y), (x \Leftrightarrow (y \vee z)), (x \Leftrightarrow (y \wedge z))$.

   C_1 will have input variables $x_1, \cdots, x_n, g_1, \cdots, g_m$ where g_i is variable encoding the truth value of gate i of C. We will encode each g_i such that $g_i = 1$ if and only if the output of gate i of C is 1.

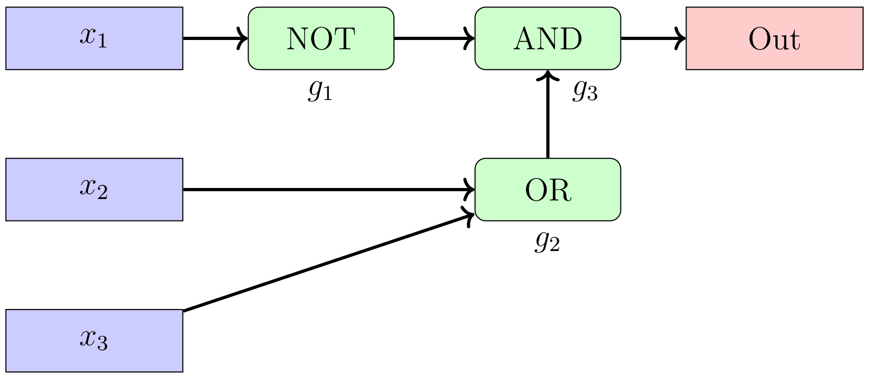
   For a gate with corresponding variable g_i and inputs from gates/input variables a and b, we write the following "clauses":

   - $g_i \Leftrightarrow (a \wedge b)$ if the gate is an AND
   - $g_i \Leftrightarrow (a \vee b)$ if the gate is an OR
   - $g_i \Leftrightarrow \neg a$ if the gate is a NOT

   To ensure all variables encode their gates correctly, we then take the conjunction (AND) of all of these "clauses". Finally, add a $\wedge g_{\text{output}}$ to the end of the expression, where g_{output} is the variable for the output gate so that the output of C_1 should be the same as the output of C. Constructing C_1 only takes $O(m)$ time by using a topological order.

   For example, for this circuit, C_1 is

$$(g_1 \Leftrightarrow \neg x_1) \wedge (g_2 \Leftrightarrow (x_2 \wedge x_3)) \wedge (g_3 \Leftrightarrow (g_1 \wedge g_2)) \wedge g_3$$



2. Replace each "clause" with real clauses:

   - $g_i \Leftrightarrow (a \wedge b)$ is equivalent to $(g_i \vee \neg a \vee \neg b) \wedge (\neg g_i \vee a) \wedge (\neg g_i \vee b)$.
   - $g_i \Leftrightarrow (a \vee b)$ is equivalent to $(\neg g_i \vee a \vee b) \wedge (g_i \vee \neg a) \wedge (g_i \vee \neg b)$.
   - $g_i \Leftrightarrow \neg a$ is equivalent to $(\neg g_i \vee \neg a) \wedge (g_i \vee a)$.

   Exercise: Verify these equivalences.

Therefore, we can convert an instance of Circuit-SAT to an equivalent instance of 3-SAT! Thus

$$\text{Circuit-SAT} \leq_P \text{3-SAT} \qquad \Rightarrow \qquad \text{3-SAT is NP-Complete.} \qquad \blacksquare$$