Lecture 21: NP-Completeness I

Michael Dinitz

November 14, 2024
601.433/633 Introduction to Algorithms

Introduction

Last few weeks: slower and slower algorithms for harder and harder problems

- From $O(m+n)$ time algorithms for BFS/DFS/topological sort/SCCs, to $O(m^2n)$ for max flow
- Today: start of two lectures on NP-completeness.
  - The (or at least a) line between tractability and intractability

Introduction

Last few weeks: slower and slower algorithms for harder and harder problems

- From $O(m+n)$ time algorithms for BFS/DFS/topological sort/SCCs, to $O(m^2 n)$ for max flow
- Today: start of two lectures on NP-completeness.
  - The (or at least a) line between tractability and intractability

Definition

An algorithm runs in *polynomial time* if its (worst-case) running time is $O(n^c)$ for some constant $c \geq 0$, where n is the size of the input.

Think of polynomial time as "fast", super-polynomial time as "slow"

Introduction

Last few weeks: slower and slower algorithms for harder and harder problems

- From $O(m+n)$ time algorithms for BFS/DFS/topological sort/SCCs, to $O(m^2 n)$ for max flow
- Today: start of two lectures on NP-completeness.
  - The (or at least a) line between tractability and intractability

Definition

An algorithm runs in *polynomial time* if its (worst-case) running time is $O(n^c)$ for some constant $c \geq 0$, where n is the size of the input.

Think of polynomial time as "fast", super-polynomial time as "slow"

Question: When do polynomial-time algorithms exist?

Decision Problems

Definition

A *decision problem* is a computational problem in which the output is either YES or NO.

Decision Problems

Definition

A *decision problem* is a computational problem in which the output is either YES or NO.

Examples:

- Max-Flow: Input is $G = (V, E), c : E \to \mathbb{R}_{\geq 0}, s, t \in V, k \in \mathbb{R}^+$. Output YES if there is an (s, t)-flow of value at least k, otherwise output NO.
- Shortest $s - t$ path: Input is $G = (V, E), \ell : E \to \mathbb{R}, s, t \in V, k \in \mathbb{R}$. Output YES if $d(s, t) \leq k$, otherwise output NO.

Decision Problems

Definition

A *decision problem* is a computational problem in which the output is either YES or NO.

Examples:

- Max-Flow: Input is $G = (V, E), c : E \to \mathbb{R}_{\geq 0}, s, t \in V, k \in \mathbb{R}^+$. Output YES if there is an (s, t)-flow of value at least k, otherwise output NO.
- Shortest $s - t$ path: Input is $G = (V, E), \ell : E \to \mathbb{R}, s, t \in V, k \in \mathbb{R}$. Output YES if $d(s, t) \leq k$, otherwise output NO.

Some problems naturally decision, others naturally optimization, but can turn any optimization problem into a decision problem.

- If can solve decision, can almost always solve optimization.

Decision Problems

Definition

A *decision problem* is a computational problem in which the output is either YES or NO.

Examples:

- Max-Flow: Input is $G = (V, E), c : E \to \mathbb{R}_{\geq 0}, s, t \in V, k \in \mathbb{R}^+$. Output YES if there is an (s, t)-flow of value at least k, otherwise output NO.
- Shortest $s - t$ path: Input is $G = (V, E), \ell : E \to \mathbb{R}, s, t \in V, k \in \mathbb{R}$. Output YES if $d(s, t) \leq k$, otherwise output NO.

Some problems naturally decision, others naturally optimization, but can turn any optimization problem into a decision problem.

- If can solve decision, can almost always solve optimization.

Note: Can divide instances (inputs) of any decision problem into YES-instances and NO-instances

P

Definition

P is the set of decision problems that can be solved in polynomial time.

Note: *problems* are in ***P***, not *algorithms*

P

Definition

P is the set of decision problems that can be solved in polynomial time.

Note: *problems* are in ***P***, not *algorithms*

Question: Are all decision problems in ***P***?

P

Definition

P is the set of decision problems that can be solved in polynomial time.

Note: *problems* are in ***P***, not *algorithms*

Question: Are all decision problems in ***P***?
Answer: No!

P

Definition

P is the set of decision problems that can be solved in polynomial time.

Note: *problems* are in ***P***, not *algorithms*

Question: Are all decision problems in ***P***?
Answer: No!

- By *time hierarchy theorem* there are problems that require super-polynomial time!
- Undecidability: there are problems which cannot be solved by *any* algorithm at all!

Verification

Different Setting: If *in addition* to the input we're given a purported solution, can we check that this solution is valid/feasible (in polynomial time)?

- Max-Flow: given $f : E \to \mathbb{R}_{\geq 0}$, check that value $\geq k$, flow conservation at all nodes other than s, t, and capacity constraints obeyed

Verification

Different Setting: If *in addition* to the input we're given a purported solution, can we check that this solution is valid/feasible (in polynomial time)?

- Max-Flow: given $f : E \to \mathbb{R}_{\geq 0}$, check that value $\geq k$, flow conservation at all nodes other than s, t, and capacity constraints obeyed

Definition (3-Coloring)

Input: Undirected graph $G = (V, E)$
Output: YES if $\exists$ *coloring* $f : V \to \{R, G, B\}$ such that $f(u) \neq f(v)$ for all $\{u, v\} \in E$. NO otherwise

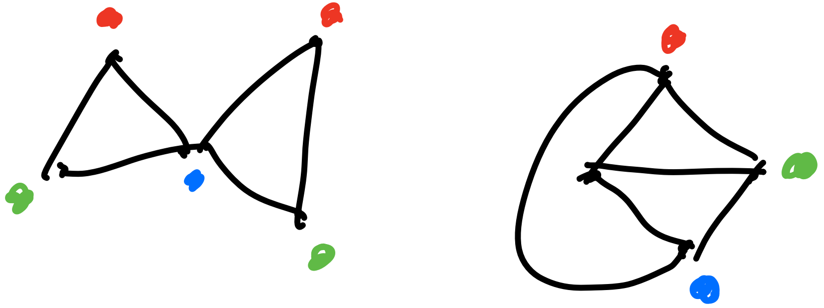
Verification

Different Setting: If *in addition* to the input we're given a purported solution, can we check that this solution is valid/feasible (in polynomial time)?

- Max-Flow: given $f : E \to \mathbb{R}_{\geq 0}$, check that value $\geq k$, flow conservation at all nodes other than s, t, and capacity constraints obeyed

Definition (3-Coloring)

Input: Undirected graph $G = (V, E)$
Output: YES if $\exists$ *coloring* $f : V \to \{R, G, B\}$ such that $f(u) \neq f(v)$ for all $\{u, v\} \in E$. NO otherwise



Verification

Different Setting: If *in addition* to the input we're given a purported solution, can we check that this solution is valid/feasible (in polynomial time)?

- Max-Flow: given $f : E \to \mathbb{R}_{\geq 0}$, check that value $\geq k$, flow conservation at all nodes other than s, t, and capacity constraints obeyed

Definition (3-Coloring)

Input: Undirected graph $G = (V, E)$
Output: YES if $\exists$ *coloring* $f : V \to \{R, G, B\}$ such that $f(u) \neq f(v)$ for all $\{u, v\} \in E$. NO otherwise

Verification: Given f,

- Check that $f(u) \in \{R, G, B\}$ for all $u \in V$, and
- Check each edge $\{u, v\}$ to make sure that $f(u) \neq f(v)$

NP

NP: decision problems where solutions can be *verified* in polynomial time.

Definition

A decision problem Q is in NP (*nondeterministic polynomial time*) if there exists a polynomial time algorithm $V(I, X)$ (called the *verifier*) such that

1. If I is a YES-instance of Q, then there is some X (usually called the *witness*, *proof*, or *solution*) with size polynomial in $|I|$ so that $V(I, X) =$ YES.
2. If I is a NO-instance of Q, then $V(I, X) =$ NO for all X.

NP

NP: decision problems where solutions can be *verified* in polynomial time.

Definition

A decision problem Q is in NP (*nondeterministic polynomial time*) if there exists a polynomial time algorithm $V(I, X)$ (called the *verifier*) such that

1. If I is a YES-instance of Q, then there is some X (usually called the *witness*, *proof*, or *solution*) with size polynomial in $|I|$ so that $V(I, X) = \text{YES}$.
2. If I is a NO-instance of Q, then $V(I, X) = \text{NO}$ for all X.

Examples:

- 3-coloring: Witness X is a coloring $f : V \to \{R, B, G\}$, verifier checks each edge $\{u, v\}$ to make sure $f(u) \neq f(v)$
  - If I is a YES instance, then there is a coloring so verifier will return YES
  - If I is a NO instance, then no valid coloring exists. Whatever X is, verifier returns NO.

NP

$\boldsymbol{NP}$: decision problems where solutions can be *verified* in polynomial time.

Definition

A decision problem Q is in NP (*nondeterministic polynomial time*) if there exists a polynomial time algorithm $V(I, X)$ (called the *verifier*) such that

1. If I is a YES-instance of Q, then there is some X (usually called the *witness*, *proof*, or *solution*) with size polynomial in $|I|$ so that $V(I, X) = \text{YES}$.
2. If I is a NO-instance of Q, then $V(I, X) = \text{NO}$ for all X.

Examples:

- Max-Flow: Witness X is a flow $f : E \to \mathbb{R}_{\geq 0}$, verifier checks that it's feasible of value $\geq k$
  - If I is a YES instance, then there is a feasible flow of value at least k so verifier (on this flow) will return YES
  - If I a NO instance, then no feasible flow of value $\geq k$. Whatever X is, verifier returns NO.

NP

NP: decision problems where solutions can be *verified* in polynomial time.

Definition

A decision problem Q is in NP (*nondeterministic polynomial time*) if there exists a polynomial time algorithm $V(I, X)$ (called the *verifier*) such that

1. If I is a YES-instance of Q, then there is some X (usually called the *witness*, *proof*, or *solution*) with size polynomial in $|I|$ so that $V(I, X) = \text{YES}$.
2. If I is a NO-instance of Q, then $V(I, X) = \text{NO}$ for all X.

Examples:

- Factoring: Instance is pair of integers M, k. YES if M has as factor in $\{2, \ldots, k\}$, NO otherwise.
  - Witness: integer f in $\{2, 3, \ldots, k\}$. Verifier: returns YES if M/f is an integer and $f \in \{2, \ldots, k\}$, NO otherwise.
  - If YES instance, then an f does exist so verifier returns YES on that f. If NO, then no such f exists so verifier always returns NO.

NP

NP: decision problems where solutions can be *verified* in polynomial time.

Definition

A decision problem Q is in NP (*nondeterministic polynomial time*) if there exists a polynomial time algorithm $V(I, X)$ (called the *verifier*) such that

1. If I is a YES-instance of Q, then there is some X (usually called the *witness*, *proof*, or *solution*) with size polynomial in $|I|$ so that $V(I, X) = \text{YES}$.
2. If I is a NO-instance of Q, then $V(I, X) = \text{NO}$ for all X.

Examples:

- Traveling Salesman: Instance is weighted graph G and integer k. YES iff G has a tour (walk that touches very vertex at least once) of length $\leq k$.
  - Witness: tour P. Verifier checks that it is a tour, has length at most k
  - If YES instance, then such a tour exists $\implies$ verifier returns YES on that tour.
  - If NO, no such tour exists $\implies$ verifier always returns NO.

NP

NP: decision problems where solutions can be *verified* in polynomial time.

Definition

A decision problem $\boldsymbol{Q}$ is in $\boldsymbol{NP}$ (*nondeterministic polynomial time*) if there exists a polynomial time algorithm $\boldsymbol{V(I, X)}$ (called the *verifier*) such that

1. If $\boldsymbol{I}$ is a YES-instance of $\boldsymbol{Q}$, then there is some $\boldsymbol{X}$ (usually called the *witness*, *proof*, or *solution*) with size polynomial in $|\boldsymbol{I}|$ so that $\boldsymbol{V(I, X)}$ = YES.
2. If $\boldsymbol{I}$ is a NO-instance of $\boldsymbol{Q}$, then $\boldsymbol{V(I, X)}$ = NO for all $\boldsymbol{X}$.

Important asymmetry: need a witness for YES, not a witness for NO.

P vs NP

Theorem

$\boldsymbol{P \subseteq NP}$

P vs NP

Theorem

$\boldsymbol{P} \subseteq \boldsymbol{NP}$

Proof.

Let $\boldsymbol{Q} \in \boldsymbol{P}$.

$\boldsymbol{V}(\boldsymbol{I}, \boldsymbol{X})$: Ignore $\boldsymbol{X}$, solve on instance $\boldsymbol{I}$. $\square$

P vs NP

Theorem

$\boldsymbol{P} \subseteq \boldsymbol{NP}$

Proof.

Let $\boldsymbol{Q} \in \boldsymbol{P}$.
$\boldsymbol{V}(\boldsymbol{I}, \boldsymbol{X})$: Ignore $\boldsymbol{X}$, solve on instance $\boldsymbol{I}$. □

Question: Does $\boldsymbol{P} = \boldsymbol{NP}$, i.e., is $\boldsymbol{NP} \subseteq \boldsymbol{P}$?

P vs NP

Theorem

$\boldsymbol{P} \subseteq \boldsymbol{NP}$

Proof.

Let $\boldsymbol{Q} \in \boldsymbol{P}$.

$\boldsymbol{V}(\boldsymbol{I}, \boldsymbol{X})$: Ignore $\boldsymbol{X}$, solve on instance $\boldsymbol{I}$. $\square$

Question: Does $\boldsymbol{P} = \boldsymbol{NP}$, i.e., is $\boldsymbol{NP} \subseteq \boldsymbol{P}$?

- *Almost* everyone thinks no, but we don't know for sure!
- Not even particularly close to a proof.
- Think about what $\boldsymbol{P} = \boldsymbol{NP}$ would mean...

Reductions

Question: How could we prove that $\boldsymbol{P} = \boldsymbol{NP}$ or $\boldsymbol{P} \neq \boldsymbol{NP}$?

Reductions

Question: How could we prove that $\boldsymbol{P} = \boldsymbol{NP}$ or $\boldsymbol{P} \neq \boldsymbol{NP}$?

- $\boldsymbol{P} = \boldsymbol{NP}$: Need to show that *every* problem in $\boldsymbol{NP}$ is also in $\boldsymbol{P}$!
- $\boldsymbol{P} \neq \boldsymbol{NP}$: Need to prove that *some* problem in $\boldsymbol{NP}$ not in $\boldsymbol{P}$.
  - What is the "hardest" problem in $\boldsymbol{NP}$?

Reductions

Question: How could we prove that $P = NP$ or $P \neq NP$?

- $P = NP$: Need to show that *every* problem in NP is also in P!
- $P \neq NP$: Need to prove that *some* problem in NP not in P.
  - What is the "hardest" problem in NP?

Definition

Problem A is *polytime reducible* to problem B (written $A \leq_p B$) if, given a polynomial-time algorithm for B, we can use it to produce a polynomial-time algorithm for A.

Reductions

Question: How could we prove that $\boldsymbol{P} = \boldsymbol{NP}$ or $\boldsymbol{P} \neq \boldsymbol{NP}$?

- $\boldsymbol{P} = \boldsymbol{NP}$: Need to show that *every* problem in $\boldsymbol{NP}$ is also in $\boldsymbol{P}$!
- $\boldsymbol{P} \neq \boldsymbol{NP}$: Need to prove that *some* problem in $\boldsymbol{NP}$ not in $\boldsymbol{P}$.
  - What is the "hardest" problem in $\boldsymbol{NP}$?

Definition

Problem $\boldsymbol{A}$ is *polytime reducible* to problem $\boldsymbol{B}$ (written $\boldsymbol{A} \leq_{\boldsymbol{p}} \boldsymbol{B}$) if, given a polynomial-time algorithm for $\boldsymbol{B}$, we can use it to produce a polynomial-time algorithm for $\boldsymbol{A}$.

Means that $\boldsymbol{B}$ is "at least as hard" as $\boldsymbol{A}$: if $\boldsymbol{B}$ is in $\boldsymbol{P}$, then so is $\boldsymbol{A}$.

- So "hardest" problems in $\boldsymbol{NP}$ are problems that many other problems reduce to.

Many-One (Karp) Reductions

Almost always (and always in this course), use a special type of reduction.

Definition

A *Many-one* or *Karp* reduction from A to B is a function f which takes arbitrary instances of A and transforms them into instances of B so that

1. If x is a YES-instance of A then $f(x)$ is a YES-instance of B.
2. If x is a NO-instance of A then $f(x)$ is a NO-instance B.
3. f can be computed in polynomial time.

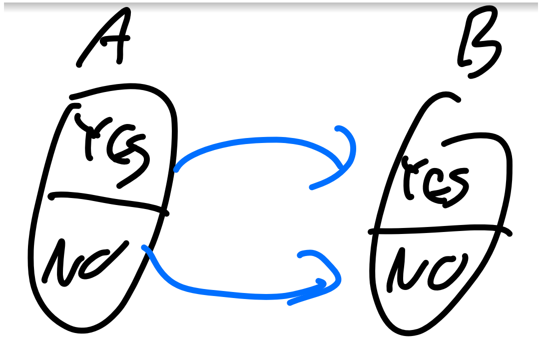
Many-One (Karp) Reductions

Almost always (and always in this course), use a special type of reduction.

Definition

A *Many-one* or *Karp* reduction from A to B is a function f which takes arbitrary instances of A and transforms them into instances of B so that

1. If x is a YES-instance of A then $f(x)$ is a YES-instance of B.
2. If x is a NO-instance of A then $f(x)$ is a NO-instance B.
3. f can be computed in polynomial time.



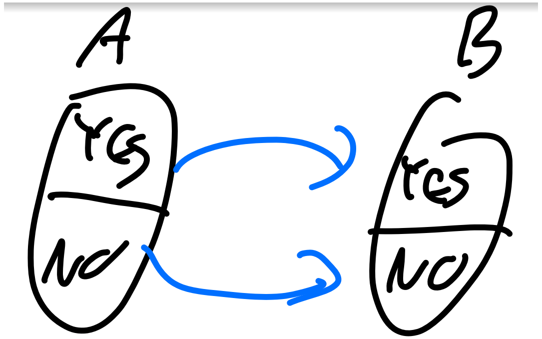
Many-One (Karp) Reductions

Almost always (and always in this course), use a special type of reduction.

Definition

A *Many-one* or *Karp* reduction from A to B is a function f which takes arbitrary instances of A and transforms them into instances of B so that

1. If x is a YES-instance of A then $f(x)$ is a YES-instance of B.
2. If x is a NO-instance of A then $f(x)$ is a NO-instance B.
3. f can be computed in polynomial time.



So given instance x of A, compute $f(x)$ and use polytime algorithm for B on $f(x)$

- Polytime, since f in polytime and algorithm for B in polytime
- Correct by first two properties of many-one reduction.

NP-Completeness

So what is "hardest problem" in $\boldsymbol{NP}$?

Definition

Problem $\boldsymbol{Q}$ is $\boldsymbol{NP}$-*hard* if $\boldsymbol{Q'} \leq_{\boldsymbol{p}} \boldsymbol{Q}$ for all problems $\boldsymbol{Q'}$ in $\boldsymbol{NP}$.

Definition

Problem $\boldsymbol{Q}$ is $\boldsymbol{NP}$-*complete* if it is $\boldsymbol{NP}$-hard and in $\boldsymbol{NP}$.

NP-Completeness

So what is "hardest problem" in $\boldsymbol{NP}$?

Definition

Problem $\boldsymbol{Q}$ is ***NP-hard*** if $\boldsymbol{Q}' \le_p \boldsymbol{Q}$ for all problems $\boldsymbol{Q}'$ in $\boldsymbol{NP}$.

Definition

Problem $\boldsymbol{Q}$ is ***NP-complete*** if it is $\boldsymbol{NP}$-hard and in $\boldsymbol{NP}$.

So suppose $\boldsymbol{Q}$ is $\boldsymbol{NP}$-complete.

- To prove $\boldsymbol{P} \neq \boldsymbol{NP}$: Hardest problem in $\boldsymbol{NP}$! If anything in $\boldsymbol{NP}$ is not in $\boldsymbol{P}$, then $\boldsymbol{Q}$ is not in $\boldsymbol{P}$
- To prove $\boldsymbol{P} = \boldsymbol{NP}$: Just need to prove that $\boldsymbol{Q} \in \boldsymbol{P}$.

NP-Completeness

So what is "hardest problem" in $\boldsymbol{NP}$?

Definition

Problem $\boldsymbol{Q}$ is *$\boldsymbol{NP}$-hard* if $\boldsymbol{Q}' \leq_{\boldsymbol{p}} \boldsymbol{Q}$ for all problems $\boldsymbol{Q}'$ in $\boldsymbol{NP}$.

Definition

Problem $\boldsymbol{Q}$ is *$\boldsymbol{NP}$-complete* if it is $\boldsymbol{NP}$-hard and in $\boldsymbol{NP}$.

So suppose $\boldsymbol{Q}$ is $\boldsymbol{NP}$-complete.

- To prove $\boldsymbol{P} \neq \boldsymbol{NP}$: Hardest problem in $\boldsymbol{NP}$! If anything in $\boldsymbol{NP}$ is not in $\boldsymbol{P}$, then $\boldsymbol{Q}$ is not in $\boldsymbol{P}$
- To prove $\boldsymbol{P} = \boldsymbol{NP}$: Just need to prove that $\boldsymbol{Q} \in \boldsymbol{P}$.

Is anything $\boldsymbol{NP}$-complete?

Circuit-SAT

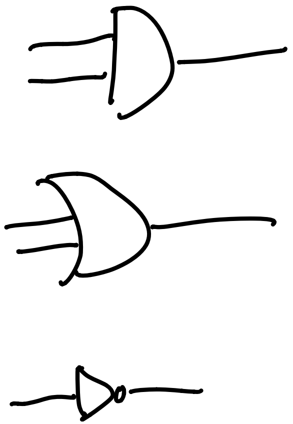
Definition

Circuit-SAT: Given a boolean circuit with a single output and no loops (some inputs might be hardwired), is there a way of setting the inputs so that the output of the circuit is **1**?

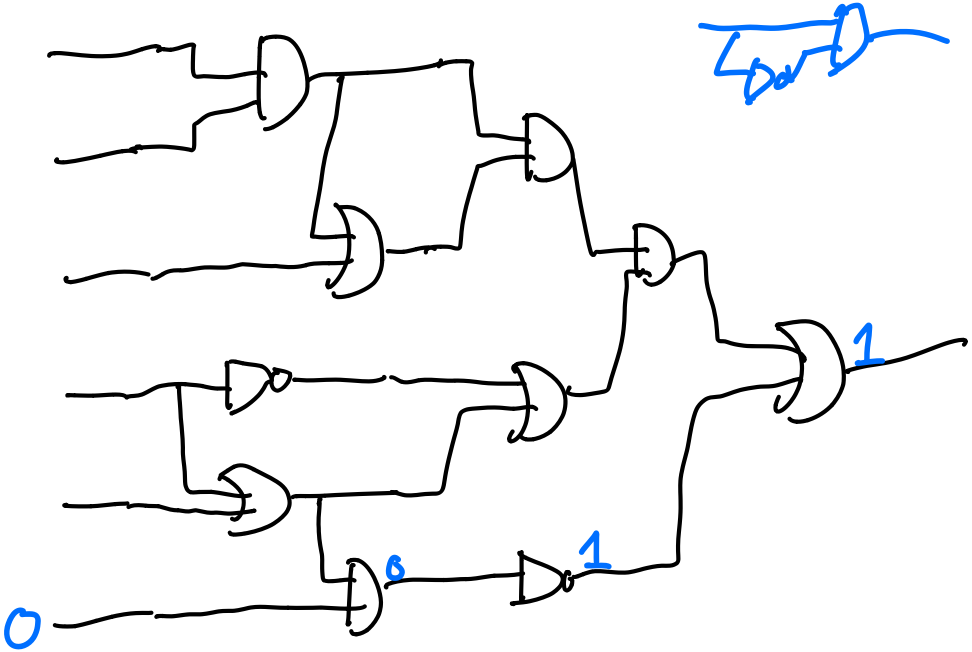
Gates: AND

OR

NOT



Arbitrary fan-out



Circuit-SAT

Theorem

*Circuit-SAT is **NP**-complete.*

Sketch of proof here. See book for details.

Circuit-SAT

Theorem

*Circuit-SAT is **NP**-complete.*

Sketch of proof here. See book for details.

Lemma

*Circuit-SAT is in **NP**.*

Circuit-SAT

Theorem

Circuit-SAT is ***NP****-complete.*

Sketch of proof here. See book for details.

Lemma

Circuit-SAT is in ***NP****.*

Proof.

Witness is a T/F (or 1/0) assignment to inputs. Verifier simulates circuit on assignment, checks that it outputs **1**.

Circuit-SAT

Theorem

*Circuit-SAT is **NP**-complete.*

Sketch of proof here. See book for details.

Lemma

*Circuit-SAT is in **NP**.*

Proof.

Witness is a T/F (or 1/0) assignment to inputs. Verifier simulates circuit on assignment, checks that it outputs $\mathbf{1}$.

- If input is a YES instance then there is some assignment so circuit outputs $\mathbf{1}$. When verifier run on that assignment, returns YES.
- In input is a NO instance then in every assignment circuit outputs $\mathbf{0}$. So verifier returns NO on every witness.

□

Circuit-SAT is $\boldsymbol{NP}$-hard

Let $\boldsymbol{A} \in \boldsymbol{NP}$. Want to show $\boldsymbol{A} \leq_{\boldsymbol{p}}$ Circuit-SAT (construct a many-one reduction).

Circuit-SAT is $\boldsymbol{NP}$-hard

Let $\boldsymbol{A} \in \boldsymbol{NP}$. Want to show $\boldsymbol{A} \leq_{\boldsymbol{p}}$ Circuit-SAT (construct a many-one reduction).

Where to start? What do we know about $\boldsymbol{A}$?

Circuit-SAT is NP-hard

Let $A \in NP$. Want to show $A \leq_p$ Circuit-SAT (construct a many-one reduction).

Where to start? What do we know about A?

- In NP, so has verifier algorithm V
- V algorithm runs on a computer (or Turing machine)!

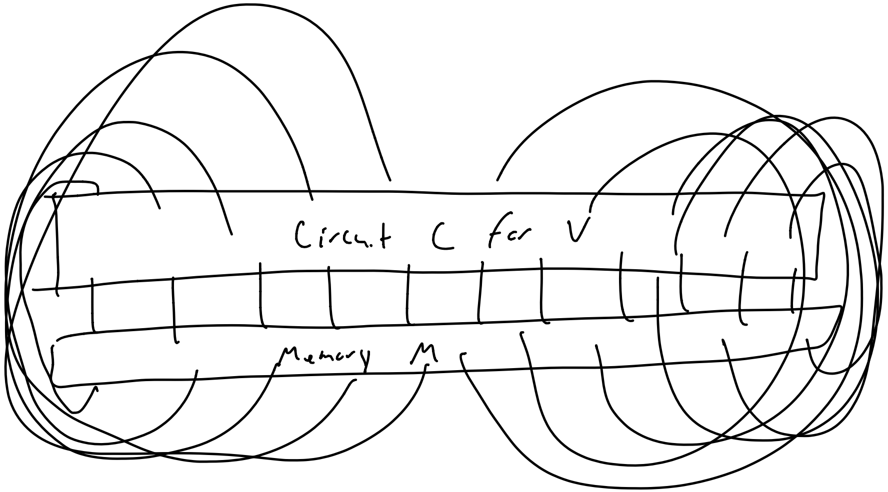
Circuit-SAT is *NP*-hard

Let $A \in NP$. Want to show $A \leq_p$ Circuit-SAT (construct a many-one reduction).

Where to start? What do we know about A?

- In NP, so has verifier algorithm V
- V algorithm runs on a computer (or Turing machine)!

Computer: memory + circuit for modifying memory!



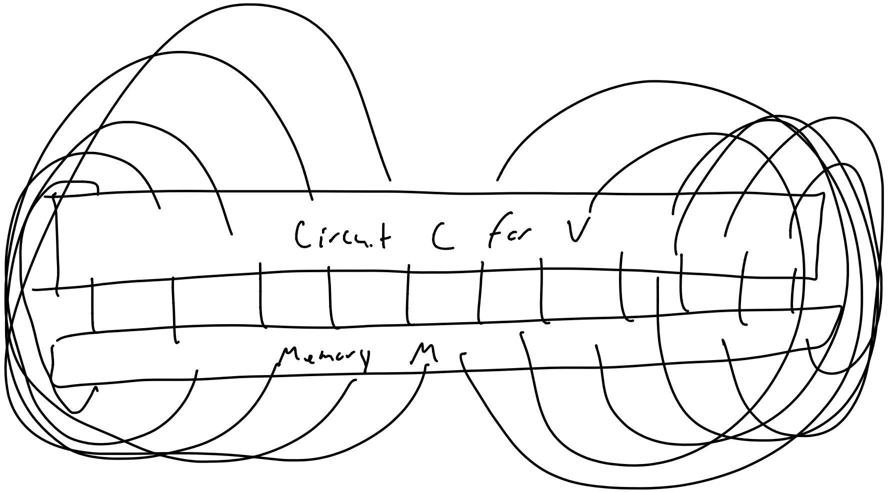
Circuit-SAT is NP-hard

Let $A \in NP$. Want to show $A \leq_p$ Circuit-SAT (construct a many-one reduction).

Where to start? What do we know about A?

- In NP, so has verifier algorithm V
- V algorithm runs on a computer (or Turing machine)!

Computer: memory + circuit for modifying memory!



Not a boolean circuit in Circuit-SAT sense: loops (feedback)

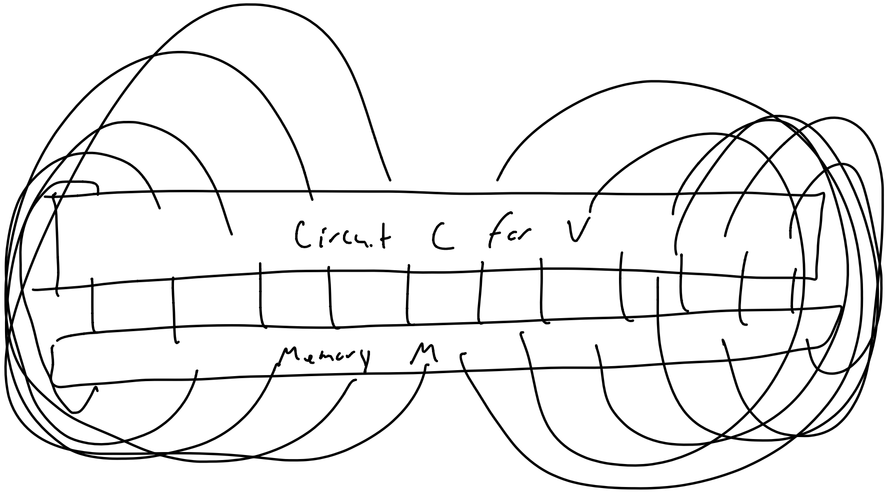
Circuit-SAT is *NP*-hard

Let $A \in NP$. Want to show $A \leq_p$ Circuit-SAT (construct a many-one reduction).

Where to start? What do we know about A?

- In NP, so has verifier algorithm V
- V algorithm runs on a computer (or Turing machine)!

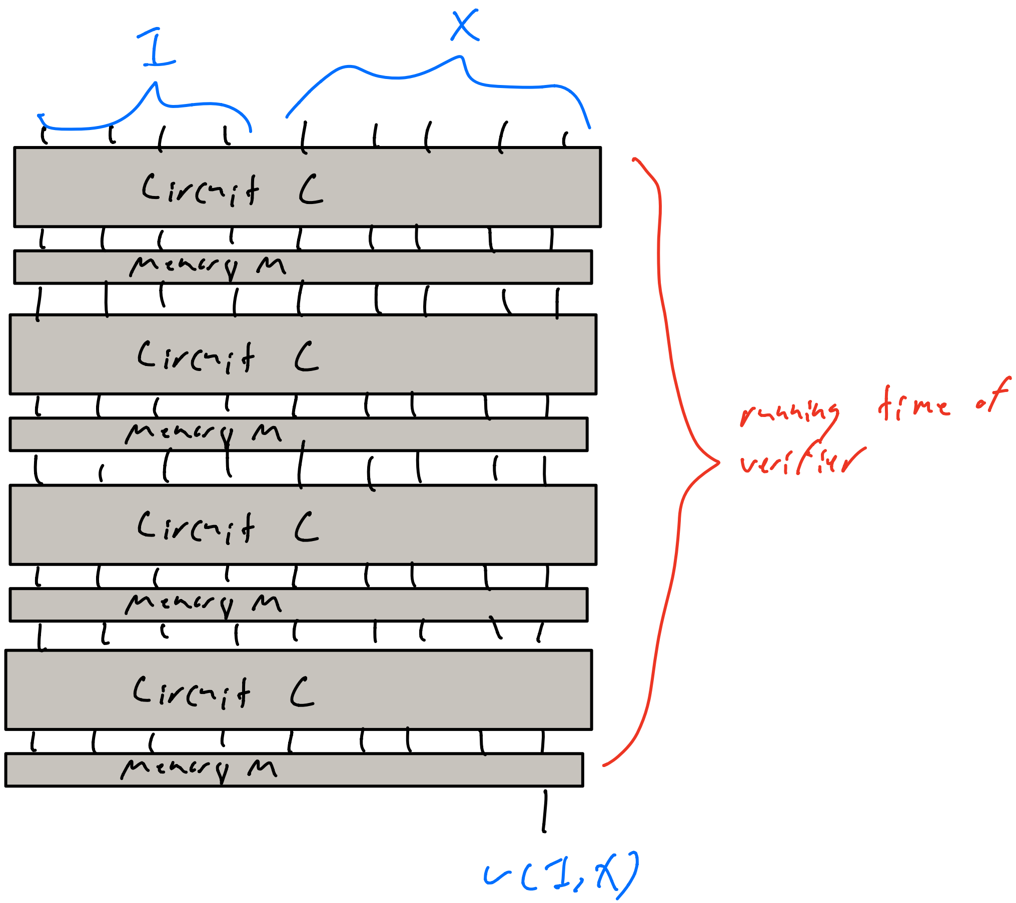
Computer: memory + circuit for modifying memory!



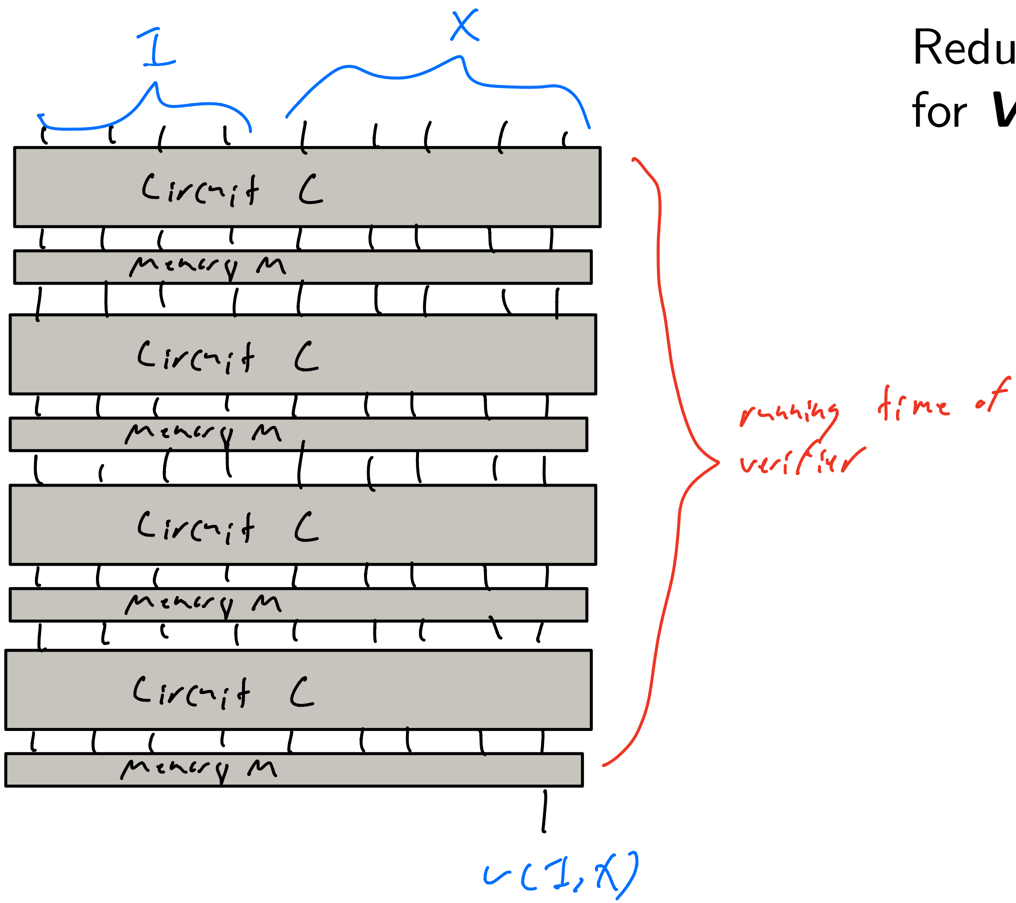
Not a boolean circuit in Circuit-SAT sense: loops (feedback)

Fix: "Unroll" circuit using fact that V runs in polynomial time

Reduction

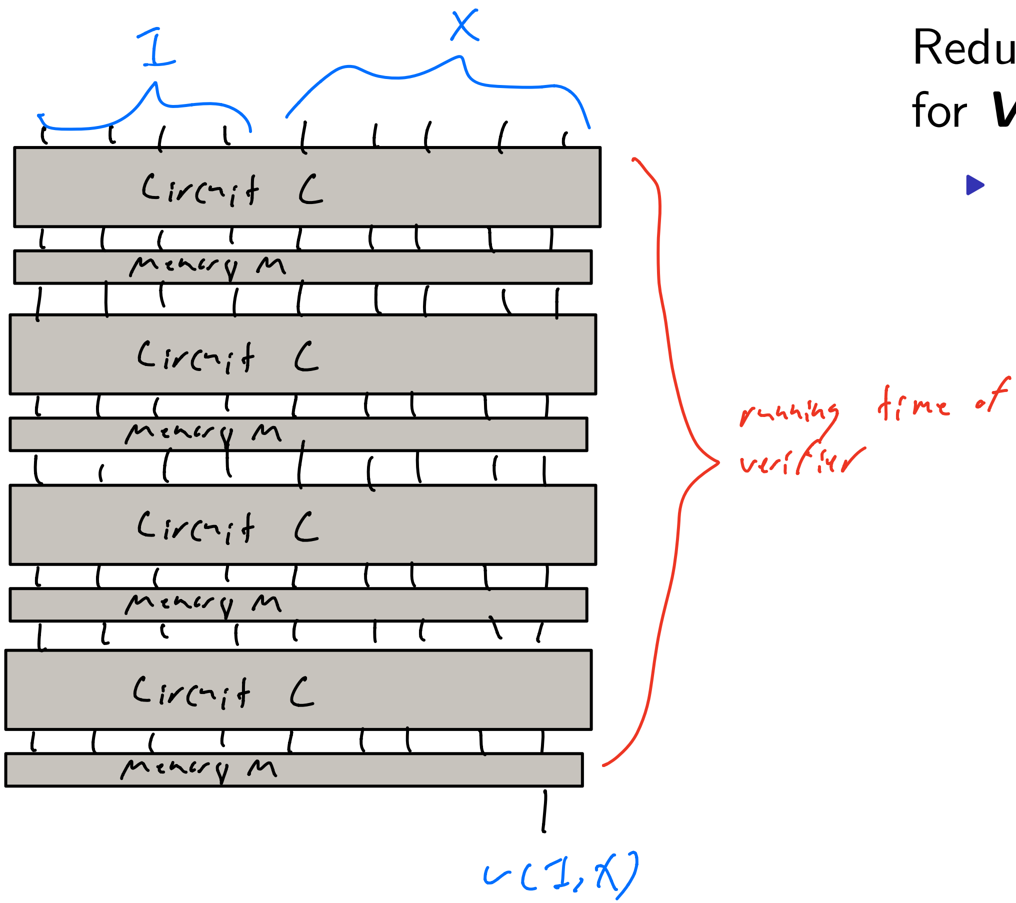

I
X
Circuit C
Memory M
Circuit C
Memory M
Circuit C
Memory M
Circuit C
Memory M
running time of verifier
V(I, X)

Reduction



Reduction: given instance $\boldsymbol{I}$ of $\boldsymbol{A}$, construct this circuit for $\boldsymbol{V}$, hardwire $\boldsymbol{I}$. Combined circuit $\boldsymbol{f(I)}$

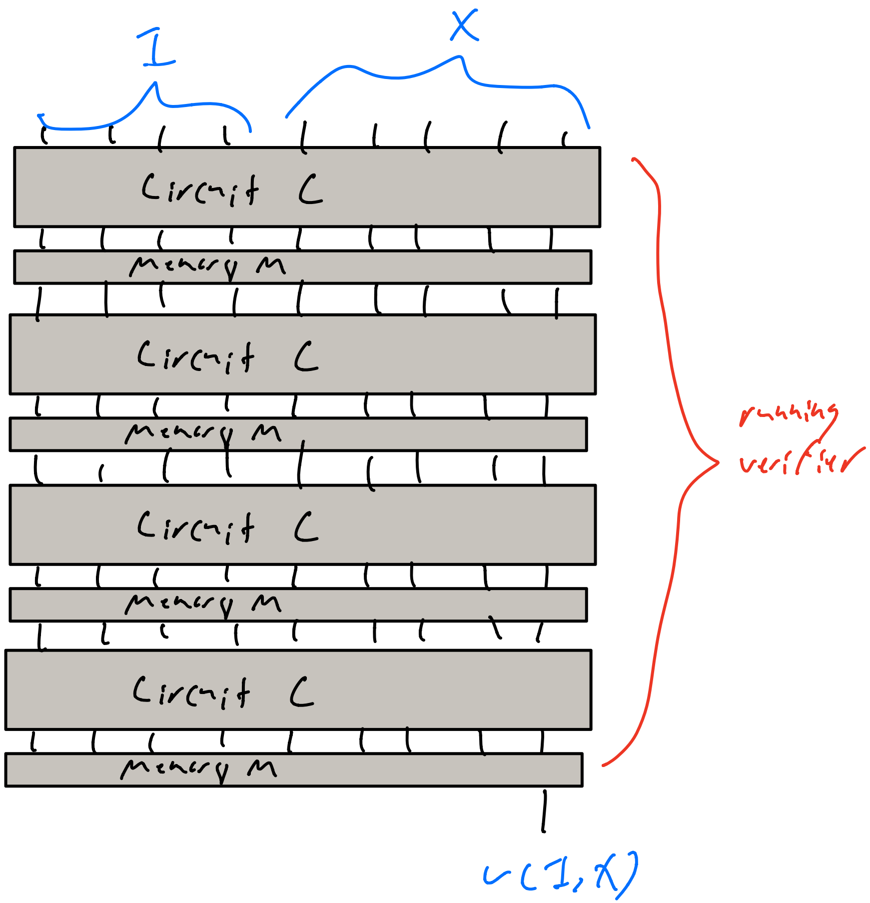
Reduction



Reduction: given instance $\boldsymbol{I}$ of $\boldsymbol{A}$, construct this circuit for $\boldsymbol{V}$, hardwire $\boldsymbol{I}$. Combined circuit $\boldsymbol{f(I)}$

- Polytime since $\boldsymbol{V}$ runs in polytime

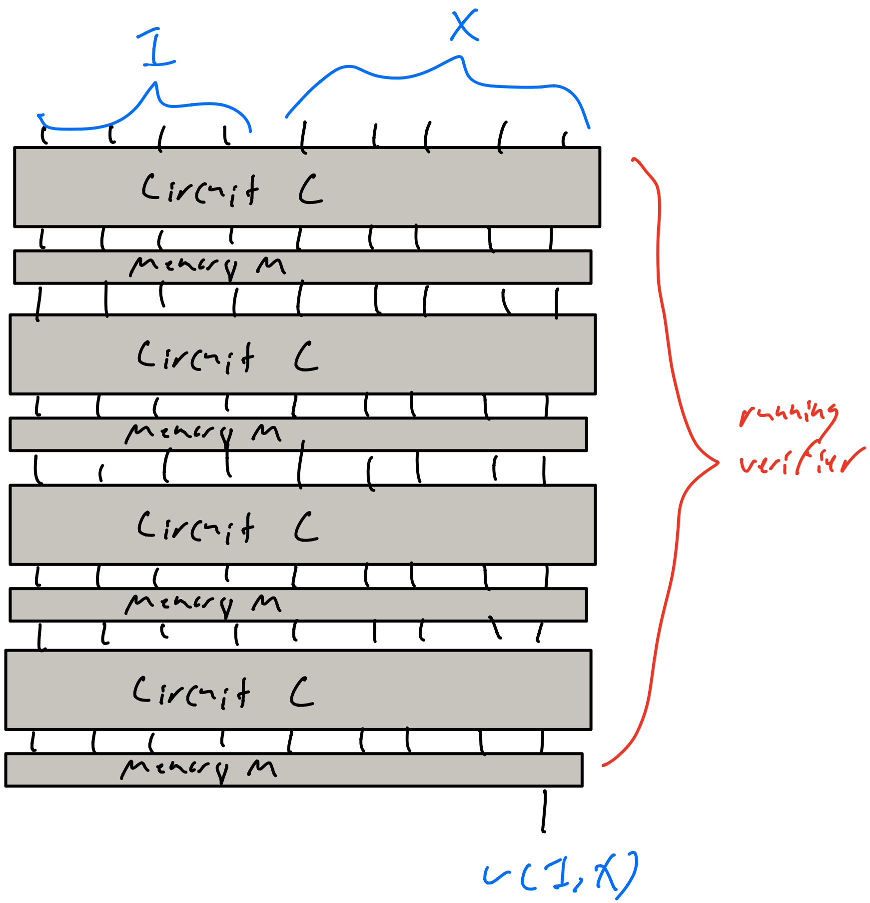
Reduction



Reduction: given instance $\boldsymbol{I}$ of $\boldsymbol{A}$, construct this circuit for $\boldsymbol{V}$, hardwire $\boldsymbol{I}$. Combined circuit $\boldsymbol{f(I)}$

- Polytime since $\boldsymbol{V}$ runs in polytime
- If $\boldsymbol{I}$ YES of $\boldsymbol{A}$: there is some $\boldsymbol{X}$ so that $\boldsymbol{V(I, X)}$ $=$ YES
  $\implies$ some $\boldsymbol{X}$ so that when $\boldsymbol{X}$ input to $\boldsymbol{f(I)}$, outputs $\mathbf{1}$
  $\implies$ $\boldsymbol{f(I)}$ YES instance of Circuit-SAT.

Reduction



Reduction: given instance I of A, construct this circuit for V, hardwire I. Combined circuit $f(I)$

- Polytime since V runs in polytime
- If I YES of A: there is some X so that $V(I, X)$ = YES
  $\implies$ some X so that when X input to $f(I)$, outputs $\mathbf{1}$
  $\implies$ $f(I)$ YES instance of Circuit-SAT.
- If I NO of A: For every X, know that $V(I, X) =$ NO
  $\implies$ for every X, when X input to $f(I)$, outputs $\mathbf{0}$
  $\implies$ $f(I)$ NO instance of Circuit-SAT