NP Completeness and Cook-Levin Theorem

Lecture 23

April 21, 2015

23.1: NP

P and NP and Turing Machines

1. Polynomial vs. polynomial time verifiable...
   1. **P**: set of decision problems that have polynomial time algorithms.
   2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.
2. Question: What is an algorithm? Depends on the model of computation!
3. What is our model of computation?
4. Formally speaking our model of computation is Turing Machines.

P and NP and Turing Machines

1. Polynomial vs. polynomial time verifiable...
   1. **P**: set of decision problems that have polynomial time algorithms.
   2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.
2. Question: What is an algorithm? Depends on the model of computation!
3. What is our model of computation?
4. Formally speaking our model of computation is Turing Machines.

P and NP and Turing Machines

1. Polynomial vs. polynomial time verifiable...
   1. **P**: set of decision problems that have polynomial time algorithms.
   2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.
2. Question: What is an algorithm? Depends on the model of computation!
3. What is our model of computation?
4. Formally speaking our model of computation is Turing Machines.

P and NP and Turing Machines

1. Polynomial vs. polynomial time verifiable...
   1. **P**: set of decision problems that have polynomial time algorithms.
   2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.
2. Question: What is an algorithm? Depends on the model of computation!
3. What is our model of computation?
4. Formally speaking our model of computation is Turing Machines.

P and NP and Turing Machines

1. Polynomial vs. polynomial time verifiable...
   1. **P**: set of decision problems that have polynomial time algorithms.
   2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.
2. Question: What is an algorithm? Depends on the model of computation!
3. What is our model of computation?
4. Formally speaking our model of computation is Turing Machines.

P and **NP** and Turing Machines

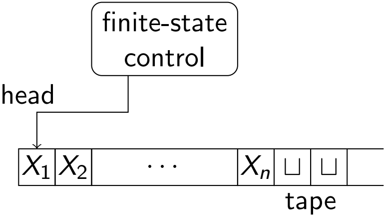
1. Polynomial vs. polynomial time verifiable...
   1. **P**: set of decision problems that have polynomial time algorithms.
   2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.
2. Question: What is an algorithm? Depends on the model of computation!
3. What is our model of computation?
4. Formally speaking our model of computation is Turing Machines.

P and NP and Turing Machines

1. Polynomial vs. polynomial time verifiable...
   1. **P**: set of decision problems that have polynomial time algorithms.
   2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.
2. Question: What is an algorithm? Depends on the model of computation!
3. What is our model of computation?
4. Formally speaking our model of computation is Turing Machines.

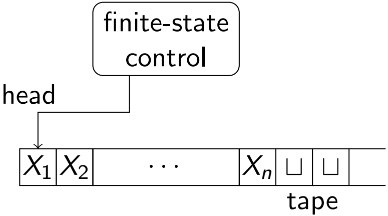
23.1.1: Turing machines

Turing Machines: Recap



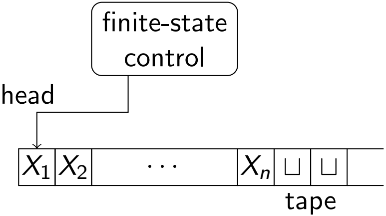
1. Infinite tape.
2. Finite state control.
3. Input at beginning of tape.
4. Special tape letter "blank" $\sqcup$.
5. Head can move only one cell to left or right.

Turing Machines: Recap



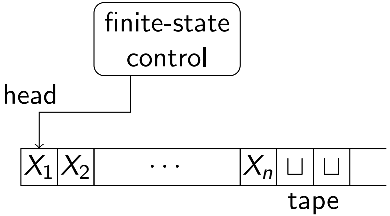
1. Infinite tape.
2. Finite state control.
3. Input at beginning of tape.
4. Special tape letter "blank" $\sqcup$.
5. Head can move only one cell to left or right.

Turing Machines: Recap



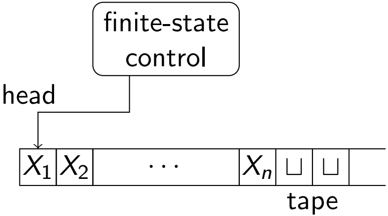
1. Infinite tape.
2. Finite state control.
3. Input at beginning of tape.
4. Special tape letter "blank" $\sqcup$.
5. Head can move only one cell to left or right.

Turing Machines: Recap



1. Infinite tape.
2. Finite state control.
3. Input at beginning of tape.
4. Special tape letter "blank" $\sqcup$.
5. Head can move only one cell to left or right.

Turing Machines: Recap



1. Infinite tape.
2. Finite state control.
3. Input at beginning of tape.
4. Special tape letter "blank" $\sqcup$.
5. Head can move only one cell to left or right.

Turing Machines: Formally

1. A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:
    1. Q is set of states in finite control
    2. q_0 start state, q_{accept} is accept state, q_{reject} is reject state
    3. Σ is input alphabet, Γ is tape alphabet (includes $\sqcup$)
    4. $\delta : Q \times \Gamma \to \{L, R\} \times \Gamma \times Q$ is transition function
        1. $\delta(q, a) = (q', b, L)$ means that M in state q and head seeing a on tape will move to state q' while replacing a on tape with b and head moves left.
2. $L(M)$: language accepted by M is set of all input strings s on which M accepts; that is:
    1. TM is started in state q_0.
    2. Initially, the tape head is located at the first cell.
    3. The tape contain s on the tape followed by blanks.
    4. The TM halts in the state q_{accept}.

Turing Machines: Formally

1. A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:
   1. Q is set of states in finite control
   2. q_0 start state, q_{accept} is accept state, q_{reject} is reject state
   3. Σ is input alphabet, Γ is tape alphabet (includes $\sqcup$)
   4. $\delta : Q \times \Gamma \to \{L, R\} \times \Gamma \times Q$ is transition function
      1. $\delta(q, a) = (q', b, L)$ means that M in state q and head seeing a on tape will move to state q' while replacing a on tape with b and head moves left.
2. $L(M)$: language accepted by M is set of all input strings s on which M accepts; that is:
   1. TM is started in state q_0.
   2. Initially, the tape head is located at the first cell.
   3. The tape contain s on the tape followed by blanks.
   4. The TM halts in the state q_{accept}.

Turing Machines: Formally

1. A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:
    1. Q is set of states in finite control
    2. q_0 start state, q_{accept} is accept state, q_{reject} is reject state
    3. Σ is input alphabet, Γ is tape alphabet (includes $\sqcup$)
    4. $\delta : Q \times \Gamma \rightarrow \{L, R\} \times \Gamma \times Q$ is transition function
        1. $\delta(q, a) = (q', b, L)$ means that M in state q and head seeing a on tape will move to state q' while replacing a on tape with b and head moves left.
2. $L(M)$: language accepted by M is set of all input strings s on which M accepts; that is:
    1. TM is started in state q_0.
    2. Initially, the tape head is located at the first cell.
    3. The tape contain s on the tape followed by blanks.
    4. The TM halts in the state q_{accept}.

Turing Machines: Formally

1. A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:
   1. Q is set of states in finite control
   2. q_0 start state, q_{accept} is accept state, q_{reject} is reject state
   3. Σ is input alphabet, Γ is tape alphabet (includes $\sqcup$)
   4. $\delta : Q \times \Gamma \rightarrow \{L, R\} \times \Gamma \times Q$ is transition function
      1. $\delta(q, a) = (q', b, L)$ means that M in state q and head seeing a on tape will move to state q' while replacing a on tape with b and head moves left.
2. $L(M)$: language accepted by M is set of all input strings s on which M accepts; that is:
   1. TM is started in state q_0.
   2. Initially, the tape head is located at the first cell.
   3. The tape contain s on the tape followed by blanks.
   4. The TM halts in the state q_{accept}.

Turing Machines: Formally

1. A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:
   1. Q is set of states in finite control
   2. q_0 start state, q_{accept} is accept state, q_{reject} is reject state
   3. Σ is input alphabet, Γ is tape alphabet (includes $\sqcup$)
   4. $\delta : Q \times \Gamma \to \{L, R\} \times \Gamma \times Q$ is transition function
      1. $\delta(q, a) = (q', b, L)$ means that M in state q and head seeing a on tape will move to state q' while replacing a on tape with b and head moves left.
2. $L(M)$: language accepted by M is set of all input strings s on which M accepts; that is:
   1. TM is started in state q_0.
   2. Initially, the tape head is located at the first cell.
   3. The tape contain s on the tape followed by blanks.
   4. The TM halts in the state q_{accept}.

Turing Machines: Formally

1. A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:
   1. Q is set of states in finite control
   2. q_0 start state, q_{accept} is accept state, q_{reject} is reject state
   3. Σ is input alphabet, Γ is tape alphabet (includes $\sqcup$)
   4. $\delta : Q \times \Gamma \rightarrow \{L, R\} \times \Gamma \times Q$ is transition function
      1. $\delta(q, a) = (q', b, L)$ means that M in state q and head seeing a on tape will move to state q' while replacing a on tape with b and head moves left.
2. $L(M)$: language accepted by M is set of all input strings s on which M accepts; that is:
   1. TM is started in state q_0.
   2. Initially, the tape head is located at the first cell.
   3. The tape contain s on the tape followed by blanks.
   4. The TM halts in the state q_{accept}.

Turing Machines: Formally

1. A TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:
   1. Q is set of states in finite control
   2. q_0 start state, q_{accept} is accept state, q_{reject} is reject state
   3. Σ is input alphabet, Γ is tape alphabet (includes $\sqcup$)
   4. $\delta : Q \times \Gamma \to \{L, R\} \times \Gamma \times Q$ is transition function
      1. $\delta(q, a) = (q', b, L)$ means that M in state q and head seeing a on tape will move to state q' while replacing a on tape with b and head moves left.
2. $L(M)$: language accepted by M is set of all input strings s on which M accepts; that is:
   1. TM is started in state q_0.
   2. Initially, the tape head is located at the first cell.
   3. The tape contain s on the tape followed by blanks.
   4. The TM halts in the state q_{accept}.

P via TMs

1. Polynomial time Turing machine.

Definition

M is a polynomial time TM if there is some polynomial $p(\cdot)$ such that on all inputs w, M halts in $p(|w|)$ steps.

2. Polynomial time language.

Definition

L is a language in **P** iff there is a polynomial time TM M such that $L = L(M)$.

P via TMs

1. Polynomial time Turing machine.

Definition

M is a polynomial time TM if there is some polynomial $p(\cdot)$ such that on all inputs w, M halts in $p(|w|)$ steps.

2. Polynomial time language.

Definition

L is a language in P iff there is a polynomial time TM M such that $L = L(M)$.

P via TMs

1. Polynomial time Turing machine.

Definition

M is a polynomial time TM if there is some polynomial $p(\cdot)$ such that on all inputs w, M halts in $p(|w|)$ steps.

2. Polynomial time language.

Definition

L is a language in **P** iff there is a polynomial time TM M such that $L = L(M)$.

NP via TMs

1. NP language...

Definition

L is an NP language iff there is a *non-deterministic* polynomial time TM M such that $L = L(M)$.

2. **Non-deterministic TM**: each step has a choice of moves
   1. $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$.
      1. Example: $\delta(q, a) = \{(q_1, b, L), (q_2, c, R), (q_3, a, R)\}$ means that M can non-deterministically choose one of the three possible moves from (q, a).
   2. $L(M)$: set of all strings s on which there *exists* some sequence of valid choices at each step that lead from q_0 to q_{accept}

NP via TMs

1. **NP** language...

Definition

L is an **NP** language iff there is a *non-deterministic* polynomial time TM M such that $L = L(M)$.

2. **Non-deterministic TM**: each step has a choice of moves
   1. $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$.
      1. Example: $\delta(q, a) = \{(q_1, b, L), (q_2, c, R), (q_3, a, R)\}$ means that M can non-deterministically choose one of the three possible moves from (q, a).
   2. $L(M)$: set of all strings s on which there *exists* some sequence of valid choices at each step that lead from q_0 to q_{accept}

NP via TMs

1. **NP** language...

Definition

L is an **NP** language iff there is a *non-deterministic* polynomial time TM M such that $L = L(M)$.

2. **Non-deterministic TM**: each step has a choice of moves
   1. $\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$.
      1. Example: $\delta(q, a) = \{(q_1, b, L), (q_2, c, R), (q_3, a, R)\}$ means that M can non-deterministically choose one of the three possible moves from (q, a).
   2. $L(M)$: set of all strings s on which there *exists* some sequence of valid choices at each step that lead from q_0 to q_{accept}

NP via TMs

1. **NP** language...

Definition

L is an **NP** language iff there is a *non-deterministic* polynomial time TM M such that $L = L(M)$.

2. **Non-deterministic TM**: each step has a choice of moves
   1. $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$.
      1. Example: $\delta(q, a) = \{(q_1, b, L), (q_2, c, R), (q_3, a, R)\}$ means that M can non-deterministically choose one of the three possible moves from (q, a).
   2. $L(M)$: set of all strings s on which there *exists* some sequence of valid choices at each step that lead from q_0 to q_{accept}

NP via TMs

1. **NP** language...

Definition

L is an **NP** language iff there is a *non-deterministic* polynomial time TM M such that $L = L(M)$.

2. **Non-deterministic TM**: each step has a choice of moves
   1. $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$.
      1. Example: $\delta(q, a) = \{(q_1, b, L), (q_2, c, R), (q_3, a, R)\}$ means that M can non-deterministically choose one of the three possible moves from (q, a).
   2. $L(M)$: set of all strings s on which there *exists* some sequence of valid choices at each step that lead from q_0 to q_{accept}

Non-deterministic TMs vs certifiers

1. Two definition of **NP**:
   1. L is in **NP** iff L has a polynomial time certifier $C(\cdot, \cdot)$.
   2. L is in **NP** iff L is decided by a non-deterministic polynomial time TM M.
2. Equivalence...

Claim

Two definitions are equivalent.

3. Why?
4. Informal proof idea: the certificate t for C corresponds to non-deterministic choices of M and vice-versa.
5. In other words L is in **NP** iff L is accepted by a NTM which first guesses a proof t of length poly in input $|s|$ and then acts as a *deterministic* TM.

Non-deterministic TMs vs certifiers

1. Two definition of **NP**:
   1. L is in **NP** iff L has a polynomial time certifier $C(\cdot, \cdot)$.
   2. L is in **NP** iff L is decided by a non-deterministic polynomial time TM M.
2. Equivalence...

Claim

Two definitions are equivalent.

3. Why?
4. Informal proof idea: the certificate t for C corresponds to non-deterministic choices of M and vice-versa.
5. In other words L is in **NP** iff L is accepted by a NTM which first guesses a proof t of length poly in input $|s|$ and then acts as a *deterministic* TM.

Non-deterministic TMs vs certifiers

1. Two definition of **NP**:
   1. L is in **NP** iff L has a polynomial time certifier $C(\cdot, \cdot)$.
   2. L is in **NP** iff L is decided by a non-deterministic polynomial time TM M.
2. Equivalence...

Claim

Two definitions are equivalent.

3. Why?
4. Informal proof idea: the certificate t for C corresponds to non-deterministic choices of M and vice-versa.
5. In other words L is in **NP** iff L is accepted by a NTM which first guesses a proof t of length poly in input $|s|$ and then acts as a *deterministic* TM.

Non-deterministic TMs vs certifiers

1. Two definition of **NP**:
   1. L is in **NP** iff L has a polynomial time certifier $C(\cdot,\cdot)$.
   2. L is in **NP** iff L is decided by a non-deterministic polynomial time TM M.
2. Equivalence...

Claim

Two definitions are equivalent.

3. Why?
4. Informal proof idea: the certificate t for C corresponds to non-deterministic choices of M and vice-versa.
5. In other words L is in **NP** iff L is accepted by a NTM which first guesses a proof t of length poly in input $|s|$ and then acts as a *deterministic* TM.

Non-deterministic TMs vs certifiers

1. Two definition of **NP**:
   1. L is in **NP** iff L has a polynomial time certifier $C(\cdot, \cdot)$.
   2. L is in **NP** iff L is decided by a non-deterministic polynomial time TM M.
2. Equivalence...

Claim

Two definitions are equivalent.

3. Why?
4. Informal proof idea: the certificate t for C corresponds to non-deterministic choices of M and vice-versa.
5. In other words L is in **NP** iff L is accepted by a NTM which first guesses a proof t of length poly in input $|s|$ and then acts as a *deterministic* TM.

Non-deterministic TMs vs certifiers

1. Two definition of **NP**:
   1. L is in **NP** iff L has a polynomial time certifier $C(\cdot, \cdot)$.
   2. L is in **NP** iff L is decided by a non-deterministic polynomial time TM M.
2. Equivalence...

Claim

Two definitions are equivalent.

3. Why?
4. Informal proof idea: the certificate t for C corresponds to non-deterministic choices of M and vice-versa.
5. In other words L is in **NP** iff L is accepted by a NTM which first guesses a proof t of length poly in input $|s|$ and then acts as a *deterministic* TM.

Non-deterministic TMs vs certifiers

1. Two definition of **NP**:
   1. L is in **NP** iff L has a polynomial time certifier $C(\cdot, \cdot)$.
   2. L is in **NP** iff L is decided by a non-deterministic polynomial time TM M.
2. Equivalence...

Claim

Two definitions are equivalent.

3. Why?
4. Informal proof idea: the certificate t for C corresponds to non-deterministic choices of M and vice-versa.
5. In other words L is in **NP** iff L is accepted by a NTM which first guesses a proof t of length poly in input $|s|$ and then acts as a *deterministic* TM.

Non-deterministic TMs vs certifiers

1. Two definition of **NP**:
   1. L is in **NP** iff L has a polynomial time certifier $C(\cdot, \cdot)$.
   2. L is in **NP** iff L is decided by a non-deterministic polynomial time TM M.
2. Equivalence...

Claim

Two definitions are equivalent.

3. Why?
4. Informal proof idea: the certificate t for C corresponds to non-deterministic choices of M and vice-versa.
5. In other words L is in **NP** iff L is accepted by a NTM which first guesses a proof t of length poly in input $|s|$ and then acts as a *deterministic* TM.

Non-determinism, guessing and verification

1. A non-deterministic machine has choices at each step and accepts a string if there *exists* a set of choices which lead to a final state.
2. Equivalently the choices can be thought of as *guessing* a solution and then *verifying* that solution. In this view all the choices are made a priori and hence the verification can be deterministic. The "guess" is the "proof" and the "verifier" is the "certifier".
3. Note: Symmetry inherent in the definition of non-determinism. Strings in the language can be easily verified. No easy way to verify that a string is not in the language.

Non-determinism, guessing and verification

1. A non-deterministic machine has choices at each step and accepts a string if there *exists* a set of choices which lead to a final state.
2. Equivalently the choices can be thought of as *guessing* a solution and then *verifying* that solution. In this view all the choices are made a priori and hence the verification can be deterministic. The "guess" is the "proof" and the "verifier" is the "certifier".
3. Note: Symmetry inherent in the definition of non-determinism. Strings in the language can be easily verified. No easy way to verify that a string is not in the language.

Non-determinism, guessing and verification

1. A non-deterministic machine has choices at each step and accepts a string if there *exists* a set of choices which lead to a final state.
2. Equivalently the choices can be thought of as *guessing* a solution and then *verifying* that solution. In this view all the choices are made a priori and hence the verification can be deterministic. The "guess" is the "proof" and the "verifier" is the "certifier".
3. Note: Symmetry inherent in the definition of non-determinism. Strings in the language can be easily verified. No easy way to verify that a string is not in the language.

Algorithms: TMs vs RAM Model

1. Why do we use TMs some times and RAM Model other times?
2. TMs are very simple: no complicated instruction set, no jumps/pointers, no explicit loops etc.
   1. Simplicity is useful in proofs.
   2. The "right" formal bare-bones model when dealing with subtleties.
3. RAM model is a closer approximation to the running time/space usage of realistic computers for reasonable problem sizes
   1. Not appropriate for certain kinds of formal proofs when algorithms can take super-polynomial time and space

Algorithms: TMs vs RAM Model

1. Why do we use TMs some times and RAM Model other times?
2. TMs are very simple: no complicated instruction set, no jumps/pointers, no explicit loops etc.
   1. Simplicity is useful in proofs.
   2. The "right" formal bare-bones model when dealing with subtleties.
3. RAM model is a closer approximation to the running time/space usage of realistic computers for reasonable problem sizes
   1. Not appropriate for certain kinds of formal proofs when algorithms can take super-polynomial time and space

Algorithms: TMs vs RAM Model

1. Why do we use TMs some times and RAM Model other times?
2. TMs are very simple: no complicated instruction set, no jumps/pointers, no explicit loops etc.
   1. Simplicity is useful in proofs.
   2. The "right" formal bare-bones model when dealing with subtleties.
3. RAM model is a closer approximation to the running time/space usage of realistic computers for reasonable problem sizes
   1. Not appropriate for certain kinds of formal proofs when algorithms can take super-polynomial time and space

Algorithms: TMs vs RAM Model

1. Why do we use TMs some times and RAM Model other times?
2. TMs are very simple: no complicated instruction set, no jumps/pointers, no explicit loops etc.
   1. Simplicity is useful in proofs.
   2. The "right" formal bare-bones model when dealing with subtleties.
3. RAM model is a closer approximation to the running time/space usage of realistic computers for reasonable problem sizes
   1. Not appropriate for certain kinds of formal proofs when algorithms can take super-polynomial time and space

23.2: Cook-Levin Theorem

23.2.1: Completeness

"Hardest" Problems

1. **Question**

   What is the hardest problem in **NP**? How do we define it?

2. Towards a definition
   1. Hardest problem must be in **NP**.
   2. Hardest problem must be at least as "difficult" as every other problem in **NP**.

“Hardest” Problems

1. **Question**

   What is the hardest problem in **NP**? How do we define it?

2. Towards a definition
   1. Hardest problem must be in **NP**.
   2. Hardest problem must be at least as “difficult” as every other problem in **NP**.

“Hardest” Problems

1. **Question**

   What is the hardest problem in **NP**? How do we define it?

2. Towards a definition
   1. Hardest problem must be in **NP**.
   2. Hardest problem must be at least as “difficult” as every other problem in **NP**.

"Hardest" Problems

1. **Question**

   What is the hardest problem in **NP**? How do we define it?

2. Towards a definition
   1. Hardest problem must be in **NP**.
   2. Hardest problem must be at least as "difficult" as every other problem in **NP**.

"Hardest" Problems

1. **Question**

   What is the hardest problem in **NP**? How do we define it?

2. Towards a definition
   1. Hardest problem must be in **NP**.
   2. Hardest problem must be at least as "difficult" as every other problem in **NP**.

NP-Complete Problems

Definition

A problem X is said to be **NP-Complete** if

1. $X \in$ **NP**, and
2. (Hardness) For any $Y \in$ **NP**, $\mathbf{Y} \leq_P \mathbf{X}$.

Solving NP-Complete Problems

Proposition

*Suppose X is **NP-Complete**. Then X can be solved in polynomial time if and only if $\mathbf{P} = \mathbf{NP}$.*

Proof.

$\Rightarrow$ Suppose X can be solved in polynomial time

1. Let $Y \in \mathbf{NP}$. We know $Y \leq_P X$.
2. We showed that if $Y \leq_P X$ and X can be solved in polynomial time, then Y can be solved in polynomial time.
3. Thus, every problem $Y \in \mathbf{NP}$ is such that $Y \in P$; $NP \subseteq P$.
4. Since $\mathbf{P} \subseteq \mathbf{NP}$, we have $\mathbf{P} = \mathbf{NP}$.

$\Leftarrow$ Since $\mathbf{P} = \mathbf{NP}$, and $X \in \mathbf{NP}$, we have a polynomial time algorithm for X. $\square$

Solving **NP-Complete** Problems

Proposition

Suppose X is **NP-Complete***. Then X can be solved in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

Proof.

$\Rightarrow$ Suppose X can be solved in polynomial time

1. Let $Y \in \mathbf{NP}$. We know $\mathbf{Y} \leq_P \mathbf{X}$.
2. We showed that if $\mathbf{Y} \leq_P \mathbf{X}$ and X can be solved in polynomial time, then Y can be solved in polynomial time.
3. Thus, every problem $Y \in \mathbf{NP}$ is such that $Y \in P$; $NP \subseteq P$.
4. Since $\mathbf{P} \subseteq \mathbf{NP}$, we have $\mathbf{P} = \mathbf{NP}$.

$\Leftarrow$ Since $\mathbf{P} = \mathbf{NP}$, and $X \in \mathbf{NP}$, we have a polynomial time algorithm for X. $\square$

Solving **NP-Complete** Problems

Proposition

Suppose X is **NP-Complete**. *Then X can be solved in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

Proof.

$\Rightarrow$ Suppose X can be solved in polynomial time

1. Let $Y \in \mathbf{NP}$. We know $\mathbf{Y} \leq_P \mathbf{X}$.
2. We showed that if $\mathbf{Y} \leq_P \mathbf{X}$ and X can be solved in polynomial time, then Y can be solved in polynomial time.
3. Thus, every problem $Y \in \mathbf{NP}$ is such that $Y \in P$; $NP \subseteq P$.
4. Since $\mathbf{P} \subseteq \mathbf{NP}$, we have $\mathbf{P} = \mathbf{NP}$.

$\Leftarrow$ Since $\mathbf{P} = \mathbf{NP}$, and $X \in \mathbf{NP}$, we have a polynomial time algorithm for X. $\square$

Solving **NP-Complete** Problems

Proposition

Suppose X is **NP-Complete**. *Then X can be solved in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

Proof.

$\Rightarrow$ Suppose X can be solved in polynomial time

1. Let $Y \in \mathbf{NP}$. We know $\mathbf{Y} \leq_P \mathbf{X}$.
2. We showed that if $\mathbf{Y} \leq_P \mathbf{X}$ and X can be solved in polynomial time, then Y can be solved in polynomial time.
3. Thus, every problem $Y \in \mathbf{NP}$ is such that $Y \in P$; $NP \subseteq P$.
4. Since $\mathbf{P} \subseteq \mathbf{NP}$, we have $\mathbf{P} = \mathbf{NP}$.

$\Leftarrow$ Since $\mathbf{P} = \mathbf{NP}$, and $X \in \mathbf{NP}$, we have a polynomial time algorithm for X. □

Solving NP-Complete Problems

Proposition

Suppose X is **NP-Complete***. Then X can be solved in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

Proof.

$\Rightarrow$ Suppose X can be solved in polynomial time

1. Let $Y \in \mathbf{NP}$. We know $\mathbf{Y} \leq_P \mathbf{X}$.
2. We showed that if $\mathbf{Y} \leq_P \mathbf{X}$ and X can be solved in polynomial time, then Y can be solved in polynomial time.
3. Thus, every problem $Y \in \mathbf{NP}$ is such that $Y \in P$; $NP \subseteq P$.
4. Since $\mathbf{P} \subseteq \mathbf{NP}$, we have $\mathbf{P} = \mathbf{NP}$.

$\Leftarrow$ Since $\mathbf{P} = \mathbf{NP}$, and $X \in \mathbf{NP}$, we have a polynomial time algorithm for X. $\square$

Solving **NP-Complete** Problems

Proposition

Suppose X is **NP-Complete**. *Then X can be solved in polynomial time if and only if* $\mathbf{P} = \mathbf{NP}$.

Proof.

$\Rightarrow$ Suppose X can be solved in polynomial time

1. Let $Y \in \mathbf{NP}$. We know $\mathbf{Y} \leq_P \mathbf{X}$.
2. We showed that if $\mathbf{Y} \leq_P \mathbf{X}$ and X can be solved in polynomial time, then Y can be solved in polynomial time.
3. Thus, every problem $Y \in \mathbf{NP}$ is such that $Y \in P$; $NP \subseteq P$.
4. Since $\mathbf{P} \subseteq \mathbf{NP}$, we have $\mathbf{P} = \mathbf{NP}$.

$\Leftarrow$ Since $\mathbf{P} = \mathbf{NP}$, and $X \in \mathbf{NP}$, we have a polynomial time algorithm for X. $\square$

NP-Hard Problems

1. **NP-Hard** problems:

Definition

A problem X is said to be **NP-Hard** if

1. (Hardness) For any $Y \in$ **NP**, we have that $\mathbf{Y} \leq_P \mathbf{X}$.

2. An **NP-Hard** problem need not be in **NP**!
3. Example: Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

NP-Hard Problems

1. **NP-Hard** problems:

Definition

A problem X is said to be **NP-Hard** if

1. (Hardness) For any $Y \in$ **NP**, we have that $\mathbf{Y} \leq_P \mathbf{X}$.

2. An **NP-Hard** problem need not be in **NP**!
3. Example: Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

NP-Hard Problems

1. **NP-Hard** problems:

Definition

A problem $\boldsymbol{X}$ is said to be **NP-Hard** if

1. (Hardness) For any $\boldsymbol{Y} \in \mathbf{NP}$, we have that $\mathbf{Y} \le_{P} \mathbf{X}$.

2. An **NP-Hard** problem need not be in **NP**!
3. Example: Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

NP-Hard Problems

1. **NP-Hard** problems:

Definition

A problem $\boldsymbol{X}$ is said to be **NP-Hard** if

1. (Hardness) For any $\boldsymbol{Y} \in \mathbf{NP}$, we have that $\mathbf{Y} \leq_P \mathbf{X}$.

2. An **NP-Hard** problem need not be in **NP**!
3. Example: Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

Consequences of proving NP-Completeness

1. If X is NP-Complete
   1. Since we believe $P \neq NP$,
   2. and solving X implies $P = NP$.
2. $\implies$ X is unlikely to be efficiently solvable.
3. $\implies$ At the very least, many smart people before you have failed to find an efficient algorithm for X.
4. (This is proof by mob opinion — take with a grain of salt.)

Consequences of proving **NP-Complete**ness

1. If X is **NP-Complete**
    1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
    2. and solving X implies $\mathbf{P} = \mathbf{NP}$.
2. $\implies$ X is unlikely to be efficiently solvable.
3. $\implies$ At the very least, many smart people before you have failed to find an efficient algorithm for X.
4. (This is proof by mob opinion — take with a grain of salt.)

Consequences of proving **NP-Complete**ness

1. If X is **NP-Complete**
   1. Since we believe **P** $\neq$ **NP**,
   2. and solving X implies **P** $=$ **NP**.
2. $\implies$ X is unlikely to be efficiently solvable.
3. $\implies$ At the very least, many smart people before you have failed to find an efficient algorithm for X.
4. (This is proof by mob opinion — take with a grain of salt.)

Consequences of proving NP-Completeness

1. If X is NP-Complete
   1. Since we believe $\mathsf{P} \neq \mathsf{NP}$,
   2. and solving X implies $\mathsf{P} = \mathsf{NP}$.
2. $\implies$ X is unlikely to be efficiently solvable.
3. $\implies$ At the very least, many smart people before you have failed to find an efficient algorithm for X.
4. (This is proof by mob opinion — take with a grain of salt.)

Consequences of proving NP-Completeness

1. If X is NP-Complete
   1. Since we believe $P \neq NP$,
   2. and solving X implies $P = NP$.
2. $\implies$ X is unlikely to be efficiently solvable.
3. $\implies$ At the very least, many smart people before you have failed to find an efficient algorithm for X.
4. (This is proof by mob opinion — take with a grain of salt.)

Consequences of proving NP-Completeness

1. If X is **NP-Complete**
   1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
   2. and solving X implies $\mathbf{P} = \mathbf{NP}$.
2. $\implies$ X is unlikely to be efficiently solvable.
3. $\implies$ At the very least, many smart people before you have failed to find an efficient algorithm for X.
4. (This is proof by mob opinion — take with a grain of salt.)

Consequences of proving **NP-Complete**ness

1. If X is **NP-Complete**
    1. Since we believe **P** $\neq$ **NP**,
    2. and solving X implies **P** $=$ **NP**.
2. $\implies$ X is unlikely to be efficiently solvable.
3. $\implies$ At the very least, many smart people before you have failed to find an efficient algorithm for X.
4. (This is proof by mob opinion — take with a grain of salt.)

23.2.2: Preliminaries

NP-Complete Problems

Question

Are there any problems that are **NP-Complete**?

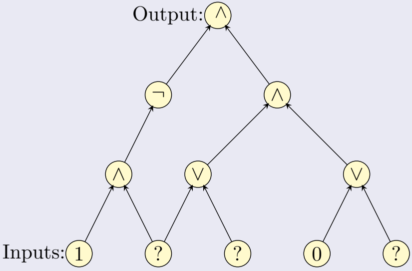
Answer

Yes! Many, many problems are **NP-Complete**.

Circuits

Definition

A circuit is a directed *acyclic* graph with



1. Input vertices (without incoming edges) labelled with **0**, **1** or a distinct variable.
2. Every other vertex is labelled $\vee$, $\wedge$ or $\neg$.
3. Single node output vertex with no outgoing edges.

23.2.3: Cook-Levin Theorem

Cook-Levin Theorem

Definition (Circuit Satisfaction (CSAT).)

Given a circuit as input, is there an assignment to the input variables that causes the output to get value $\mathbf{1}$?

Theorem (Cook-Levin)

CSAT *is* **NP-Complete***.*

Need to show

1. **CSAT** is in **NP**.
2. *every* **NP** problem X reduces to **CSAT**.

CSAT: Circuit Satisfaction

Claim

CSAT *is in* **NP**.

1. Certificate: Assignment to input variables.
2. Certifier: Evaluate the value of each gate in a topological sort of DAG and check the output gate value.

CSAT: Circuit Satisfaction

Claim

CSAT *is in* **NP**.

1. Certificate: Assignment to input variables.
2. Certifier: Evaluate the value of each gate in a topological sort of DAG and check the output gate value.

CSAT is NP-hard: Idea

1. Need to show that *every* NP problem X reduces to CSAT.
2. What does it mean that $X \in$ NP?
3. $X \in$ NP implies that there are polynomials $p()$ and $q()$ and certifier/verifier program C such that for every string s the following is true:
   1. If s is a YES instance ($s \in X$) then there is a *proof* t of length $p(|s|)$ such that $C(s, t)$ says YES.
   2. If s is a NO instance ($s \notin X$) then for every string t of length at $p(|s|)$, $C(s, t)$ says NO.
   3. $C(s, t)$ runs in time $q(|s| + |t|)$ time (hence polynomial time).

CSAT is NP-hard: Idea

1. Need to show that *every* **NP** problem X reduces to **CSAT**.
2. What does it mean that $X \in$ **NP**?
3. $X \in$ **NP** implies that there are polynomials $p()$ and $q()$ and certifier/verifier program C such that for every string s the following is true:
    1. If s is a YES instance ($s \in X$) then there is a *proof* t of length $p(|s|)$ such that $C(s, t)$ says YES.
    2. If s is a NO instance ($s \notin X$) then for every string t of length at $p(|s|)$, $C(s, t)$ says NO.
    3. $C(s, t)$ runs in time $q(|s| + |t|)$ time (hence polynomial time).

CSAT is NP-hard: Idea

1. Need to show that *every* **NP** problem X reduces to **CSAT**.
2. What does it mean that $X \in$ **NP**?
3. $X \in$ **NP** implies that there are polynomials $p()$ and $q()$ and certifier/verifier program C such that for every string s the following is true:
    1. If s is a YES instance ($s \in X$) then there is a *proof* t of length $p(|s|)$ such that $C(s, t)$ says YES.
    2. If s is a NO instance ($s \notin X$) then for every string t of length at $p(|s|)$, $C(s, t)$ says NO.
    3. $C(s, t)$ runs in time $q(|s| + |t|)$ time (hence polynomial time).

CSAT is NP-hard: Idea

1. Need to show that *every* **NP** problem X reduces to **CSAT**.
2. What does it mean that $X \in$ **NP**?
3. $X \in$ **NP** implies that there are polynomials $p()$ and $q()$ and certifier/verifier program C such that for every string s the following is true:
   1. If s is a YES instance ($s \in X$) then there is a *proof* t of length $p(|s|)$ such that $C(s, t)$ says YES.
   2. If s is a NO instance ($s \notin X$) then for every string t of length at $p(|s|)$, $C(s, t)$ says NO.
   3. $C(s, t)$ runs in time $q(|s| + |t|)$ time (hence polynomial time).

CSAT is NP-hard: Idea

1. Need to show that *every* **NP** problem X reduces to **CSAT**.
2. What does it mean that $X \in$ **NP**?
3. $X \in$ **NP** implies that there are polynomials $p()$ and $q()$ and certifier/verifier program C such that for every string s the following is true:
   1. If s is a YES instance ($s \in X$) then there is a *proof* t of length $p(|s|)$ such that $C(s, t)$ says YES.
   2. If s is a NO instance ($s \not\in X$) then for every string t of length at $p(|s|)$, $C(s, t)$ says NO.
   3. $C(s, t)$ runs in time $q(|s| + |t|)$ time (hence polynomial time).

CSAT is NP-hard: Idea

1. Need to show that *every* **NP** problem X reduces to **CSAT**.
2. What does it mean that $X \in$ **NP**?
3. $X \in$ **NP** implies that there are polynomials $p()$ and $q()$ and certifier/verifier program C such that for every string s the following is true:
    1. If s is a YES instance ($s \in X$) then there is a *proof* t of length $p(|s|)$ such that $C(s, t)$ says YES.
    2. If s is a NO instance ($s \notin X$) then for every string t of length at $p(|s|)$, $C(s, t)$ says NO.
    3. $C(s, t)$ runs in time $q(|s| + |t|)$ time (hence polynomial time).

CSAT is NP-hard: Idea

1. Need to show that *every* **NP** problem X reduces to **CSAT**.
2. What does it mean that $X \in$ **NP**?
3. $X \in$ **NP** implies that there are polynomials $p()$ and $q()$ and certifier/verifier program C such that for every string s the following is true:
    1. If s is a YES instance ($s \in X$) then there is a *proof* t of length $p(|s|)$ such that $C(s, t)$ says YES.
    2. If s is a NO instance ($s \notin X$) then for every string t of length at $p(|s|)$, $C(s, t)$ says NO.
    3. $C(s, t)$ runs in time $q(|s| + |t|)$ time (hence polynomial time).

Reducing X to CSAT

1. X is in **NP** means we have access to $p(), q(), C(\cdot,\cdot)$.
2. What is $C(\cdot,\cdot)$? It is a program or equivalently a Turing Machine!
3. How are $p()$ and $q()$ given? As numbers (coefficients and powers).
4. Example: if 3 is given then $p(n) = n^3$.
5. Thus an **NP** problem is essentially a three tuple $\langle p, q, C \rangle$ where C is either a program or a TM.

Reducing X to CSAT

1. X is in **NP** means we have access to $p(), q(), C(\cdot,\cdot)$.
2. What is $C(\cdot,\cdot)$? It is a program or equivalently a Turing Machine!
3. How are $p()$ and $q()$ given? As numbers (coefficients and powers).
4. Example: if 3 is given then $p(n) = n^3$.
5. Thus an **NP** problem is essentially a three tuple $\langle p, q, C \rangle$ where C is either a program or a TM.

Reducing X to CSAT

1. X is in **NP** means we have access to $p(), q(), C(\cdot,\cdot)$.
2. What is $C(\cdot,\cdot)$? It is a program or equivalently a Turing Machine!
3. How are $p()$ and $q()$ given? As numbers (coefficients and powers).
4. Example: if 3 is given then $p(n) = n^3$.
5. Thus an **NP** problem is essentially a three tuple $\langle p, q, C \rangle$ where C is either a program or a TM.

Reducing X to CSAT

1. X is in **NP** means we have access to $p(), q(), C(\cdot, \cdot)$.
2. What is $C(\cdot, \cdot)$? It is a program or equivalently a Turing Machine!
3. How are $p()$ and $q()$ given? As numbers (coefficients and powers).
4. Example: if 3 is given then $p(n) = n^3$.
5. Thus an **NP** problem is essentially a three tuple $\langle p, q, C \rangle$ where C is either a program or a TM.

Reducing X to CSAT

1. X is in **NP** means we have access to $p(), q(), C(\cdot, \cdot)$.
2. What is $C(\cdot, \cdot)$? It is a program or equivalently a Turing Machine!
3. How are $p()$ and $q()$ given? As numbers (coefficients and powers).
4. Example: if 3 is given then $p(n) = n^3$.
5. Thus an **NP** problem is essentially a three tuple $\langle p, q, C \rangle$ where C is either a program or a TM.

Reducing X to CSAT

1. X is in **NP** means we have access to $p(), q(), C(\cdot, \cdot)$.
2. What is $C(\cdot, \cdot)$? It is a program or equivalently a Turing Machine!
3. How are $p()$ and $q()$ given? As numbers (coefficients and powers).
4. Example: if $\mathbf{3}$ is given then $p(n) = n^3$.
5. Thus an **NP** problem is essentially a three tuple $\langle p, q, C \rangle$ where C is either a program or a TM.

Reducing X to CSAT

1. Thus an **NP** problem is essentially a three tuple $\langle p, q, C \rangle$ where C is either a program or TM.
2. Problem X: Given string s, is $s \in X$?
3. Same as the following: is there a proof t of length $p(|s|)$ such that $C(s, t)$ says YES.
4. How do we reduce X to **CSAT**? Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C \rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C \rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES.

Reducing X to CSAT

1. Thus an **NP** problem is essentially a three tuple $\langle p, q, C \rangle$ where C is either a program or TM.
2. Problem X: Given string s, is $s \in X$?
3. Same as the following: is there a proof t of length $p(|s|)$ such that $C(s, t)$ says YES.
4. How do we reduce X to **CSAT**? Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C \rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C \rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES.

Reducing X to CSAT

1. Thus an **NP** problem is essentially a three tuple $\langle p, q, C \rangle$ where C is either a program or TM.
2. Problem X: Given string s, is $s \in X$?
3. Same as the following: is there a proof t of length $p(|s|)$ such that $C(s, t)$ says YES.
4. How do we reduce X to **CSAT**? Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C \rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C \rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES.

Reducing X to CSAT

1. Thus an **NP** problem is essentially a three tuple $\langle p, q, C \rangle$ where C is either a program or TM.
2. Problem X: Given string s, is $s \in X$?
3. Same as the following: is there a proof t of length $p(|s|)$ such that $C(s, t)$ says YES.
4. How do we reduce X to **CSAT**? Need an algorithm $\mathcal{A}$ that
    1. takes s (and $\langle p, q, C \rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C \rangle$ are fixed).
    2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES.

Reducing X to CSAT

1. Thus an **NP** problem is essentially a three tuple $\langle p, q, C\rangle$ where C is either a program or TM.
2. Problem X: Given string s, is $s \in X$?
3. Same as the following: is there a proof t of length $p(|s|)$ such that $C(s, t)$ says YES.
4. How do we reduce X to **CSAT**? Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C\rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C\rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES.

Reducing X to CSAT

1. Thus an **NP** problem is essentially a three tuple $\langle p, q, C \rangle$ where C is either a program or TM.
2. Problem X: Given string s, is $s \in X$?
3. Same as the following: is there a proof t of length $p(|s|)$ such that $C(s, t)$ says YES.
4. How do we reduce X to **CSAT**? Need an algorithm $\mathcal{A}$ that
    1. takes s (and $\langle p, q, C \rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C \rangle$ are fixed).
    2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES.

Reducing X to CSAT

1. Thus an **NP** problem is essentially a three tuple $\langle p, q, C\rangle$ where C is either a program or TM.
2. Problem X: Given string s, is $s \in X$?
3. Same as the following: is there a proof t of length $p(|s|)$ such that $C(s, t)$ says YES.
4. How do we reduce X to **CSAT**? Need an algorithm $\mathcal{A}$ that
    1. takes s (and $\langle p, q, C\rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C\rangle$ are fixed).
    2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES.

Reducing X to CSAT

1. Thus an **NP** problem is essentially a three tuple $\langle p, q, C\rangle$ where C is either a program or TM.
2. Problem X: Given string s, is $s \in X$?
3. Same as the following: is there a proof t of length $p(|s|)$ such that $C(s, t)$ says YES.
4. How do we reduce X to **CSAT**? Need an algorithm $\mathcal{A}$ that
    1. takes s (and $\langle p, q, C\rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C\rangle$ are fixed).
    2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES.

Reducing X to CSAT

1. How do we reduce X to CSAT?
2. Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C \rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C \rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES
3. Simple but Big Idea: Programs are essentially the same as Circuits!
   1. Convert $C(s, t)$ into a circuit G with t as unknown inputs (rest is known including s)
   2. We know that $|t| = p(|s|)$ so express boolean string t as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
   3. Asking if there is a proof t that makes $C(s, t)$ say YES is same as whether there is an assignment of values to "unknown" variables $t_1, t_2, \ldots, t_k$ that will make G evaluate to true/YES.

Reducing X to CSAT

1. How do we reduce X to **CSAT**?
2. Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C \rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C \rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES
3. Simple but Big Idea: Programs are essentially the same as Circuits!
   1. Convert $C(s, t)$ into a circuit G with t as unknown inputs (rest is known including s)
   2. We know that $|t| = p(|s|)$ so express boolean string t as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
   3. Asking if there is a proof t that makes $C(s, t)$ say YES is same as whether there is an assignment of values to "unknown" variables $t_1, t_2, \ldots, t_k$ that will make G evaluate to true/YES.

Reducing X to CSAT

1. How do we reduce X to **CSAT**?
2. Need an algorithm $\mathcal{A}$ that
    1. takes s (and $\langle p, q, C\rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C\rangle$ are fixed).
    2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES
3. Simple but Big Idea: Programs are essentially the same as Circuits!
    1. Convert $C(s, t)$ into a circuit G with t as unknown inputs (rest is known including s)
    2. We know that $|t| = p(|s|)$ so express boolean string t as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
    3. Asking if there is a proof t that makes $C(s, t)$ say YES is same as whether there is an assignment of values to "unknown" variables $t_1, t_2, \ldots, t_k$ that will make G evaluate to true/YES.

Reducing X to CSAT

1. How do we reduce X to **CSAT**?
2. Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C\rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C\rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES
3. Simple but Big Idea: Programs are essentially the same as Circuits!
   1. Convert $C(s, t)$ into a circuit G with t as unknown inputs (rest is known including s)
   2. We know that $|t| = p(|s|)$ so express boolean string t as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
   3. Asking if there is a proof t that makes $C(s, t)$ say YES is same as whether there is an assignment of values to "unknown" variables $t_1, t_2, \ldots, t_k$ that will make G evaluate to true/YES.

Reducing X to CSAT

1. How do we reduce X to **CSAT**?
2. Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C \rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C \rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES
3. Simple but Big Idea: Programs are essentially the same as Circuits!
   1. Convert $C(s, t)$ into a circuit G with t as unknown inputs (rest is known including s)
   2. We know that $|t| = p(|s|)$ so express boolean string t as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
   3. Asking if there is a proof t that makes $C(s, t)$ say YES is same as whether there is an assignment of values to "unknown" variables $t_1, t_2, \ldots, t_k$ that will make G evaluate to true/YES.

Reducing X to CSAT

1. How do we reduce X to **CSAT**?
2. Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C \rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C \rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES
3. Simple but Big Idea: Programs are essentially the same as Circuits!
   1. Convert $C(s, t)$ into a circuit G with t as unknown inputs (rest is known including s)
   2. We know that $|t| = p(|s|)$ so express boolean string t as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
   3. Asking if there is a proof t that makes $C(s, t)$ say YES is same as whether there is an assignment of values to "unknown" variables $t_1, t_2, \ldots, t_k$ that will make G evaluate to true/YES.

Reducing X to CSAT

1. How do we reduce X to **CSAT**?
2. Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C\rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C\rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES
3. Simple but Big Idea: Programs are essentially the same as Circuits!
   1. Convert $C(s, t)$ into a circuit G with t as unknown inputs (rest is known including s)
   2. We know that $|t| = p(|s|)$ so express boolean string t as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
   3. Asking if there is a proof t that makes $C(s, t)$ say YES is same as whether there is an assignment of values to "unknown" variables $t_1, t_2, \ldots, t_k$ that will make G evaluate to true/YES.

Reducing X to CSAT

1. How do we reduce X to **CSAT**?
2. Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C\rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C\rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES
3. Simple but Big Idea: Programs are essentially the same as Circuits!
   1. Convert $C(s, t)$ into a circuit G with t as unknown inputs (rest is known including s)
   2. We know that $|t| = p(|s|)$ so express boolean string t as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
   3. Asking if there is a proof t that makes $C(s, t)$ say YES is same as whether there is an assignment of values to "unknown" variables $t_1, t_2, \ldots, t_k$ that will make G evaluate to true/YES.

Reducing X to CSAT

1. How do we reduce X to **CSAT**?
2. Need an algorithm $\mathcal{A}$ that
   1. takes s (and $\langle p, q, C \rangle$) and creates a circuit G in polynomial time in $|s|$ (note that $\langle p, q, C \rangle$ are fixed).
   2. G is satisfiable if and only if there is a proof t such that $C(s, t)$ says YES
3. Simple but Big Idea: Programs are essentially the same as Circuits!
   1. Convert $C(s, t)$ into a circuit G with t as unknown inputs (rest is known including s)
   2. We know that $|t| = p(|s|)$ so express boolean string t as $p(|s|)$ variables $t_1, t_2, \ldots, t_k$ where $k = p(|s|)$.
   3. Asking if there is a proof t that makes $C(s, t)$ say YES is same as whether there is an assignment of values to "unknown" variables $t_1, t_2, \ldots, t_k$ that will make G evaluate to true/YES.

Example: Independent Set

1. Problem: Does $G = (V, E)$ have an **Independent Set** of size $\geq k$?
   1. Certificate: Set $S \subseteq V$.
   2. Certifier: Check $|S| \geq k$ and no pair of vertices in S is connected by an edge.
2. Formally, why is **Independent Set** in **NP**?

Example: Independent Set

1. Problem: Does $G = (V, E)$ have an **Independent Set** of size $\geq k$?
   1. Certificate: Set $S \subseteq V$.
   2. Certifier: Check $|S| \geq k$ and no pair of vertices in S is connected by an edge.
2. Formally, why is **Independent Set** in **NP**?

Example: Independent Set

Formally why is **Independent Set** in **NP**?

1. Input: $< n, y_{1,1}, y_{1,2}, \ldots, y_{1,n}, y_{2,1}, \ldots, y_{2,n}, \ldots, y_{n,1}, \ldots, y_{n,n}, k >$ encodes $< G, k >$.
   1. n is number of vertices in G
   2. $y_{i,j}$ is a bit which is 1 if edge (i, j) is in G and 0 otherwise (adjacency matrix representation)
   3. k is size of independent set.
2. Certificate: $t = t_1 t_2 \ldots t_n$. Interpretation is that t_i is 1 if vertex i is in the independent set, 0 otherwise.

Example: Independent Set

Formally why is **Independent Set** in **NP**?

1. Input: $< n, y_{1,1}, y_{1,2}, \ldots, y_{1,n}, y_{2,1}, \ldots, y_{2,n}, \ldots, y_{n,1}, \ldots, y_{n,n}, k >$ encodes $< G, k >$.
    1. n is number of vertices in G
    2. $y_{i,j}$ is a bit which is 1 if edge (i, j) is in G and 0 otherwise (adjacency matrix representation)
    3. k is size of independent set.
2. Certificate: $t = t_1 t_2 \ldots t_n$. Interpretation is that t_i is 1 if vertex i is in the independent set, 0 otherwise.

Example: Independent Set

Formally why is **Independent Set** in **NP**?

1. Input: $<$ $n, y_{1,1}, y_{1,2}, \ldots, y_{1,n}, y_{2,1}, \ldots, y_{2,n}, \ldots, y_{n,1}, \ldots, y_{n,n}, k >$ encodes $< G, k >$.
   1. n is number of vertices in G
   2. $y_{i,j}$ is a bit which is 1 if edge (i, j) is in G and 0 otherwise (adjacency matrix representation)
   3. k is size of independent set.
2. Certificate: $t = t_1 t_2 \ldots t_n$. Interpretation is that t_i is 1 if vertex i is in the independent set, 0 otherwise.

Example: Independent Set

Formally why is **Independent Set** in **NP**?

1. Input: $<$ $n, y_{1,1}, y_{1,2}, \ldots, y_{1,n}, y_{2,1}, \ldots, y_{2,n}, \ldots, y_{n,1}, \ldots, y_{n,n}, k >$ encodes $< G, k >$.
    1. n is number of vertices in G
    2. $y_{i,j}$ is a bit which is 1 if edge (i, j) is in G and 0 otherwise (adjacency matrix representation)
    3. k is size of independent set.
2. Certificate: $t = t_1 t_2 \ldots t_n$. Interpretation is that t_i is 1 if vertex i is in the independent set, 0 otherwise.

Example: Independent Set

Formally why is **Independent Set** in **NP**?

1. Input: $<$ $n, y_{1,1}, y_{1,2}, \ldots, y_{1,n}, y_{2,1}, \ldots, y_{2,n}, \ldots, y_{n,1}, \ldots, y_{n,n}, k >$ encodes $< G, k >$.
   1. n is number of vertices in G
   2. $y_{i,j}$ is a bit which is 1 if edge (i, j) is in G and 0 otherwise (adjacency matrix representation)
   3. k is size of independent set.
2. Certificate: $t = t_1 t_2 \ldots t_n$. Interpretation is that t_i is 1 if vertex i is in the independent set, 0 otherwise.

Example: Independent Set

Formally why is **Independent Set** in **NP**?

1. Input: $< n, y_{1,1}, y_{1,2}, \ldots, y_{1,n}, y_{2,1}, \ldots, y_{2,n}, \ldots, y_{n,1}, \ldots, y_{n,n}, k >$ encodes $< G, k >$.
   1. n is number of vertices in G
   2. $y_{i,j}$ is a bit which is 1 if edge (i, j) is in G and 0 otherwise (adjacency matrix representation)
   3. k is size of independent set.
2. Certificate: $t = t_1 t_2 \ldots t_n$. Interpretation is that t_i is 1 if vertex i is in the independent set, 0 otherwise.

Certifier for **Independent Set**

Certifier $C(s, t)$ for **Independent Set**:

```
if (t_1 + t_2 + ... + t_n < k) then
    return NO
else
    for each (i, j) do
        if (t_i ∧ t_j ∧ y_{i,j}) then
            return NO

return YES
```

Example: Independent Set

A certifier circuit for Independent Set

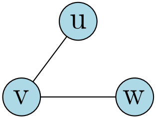
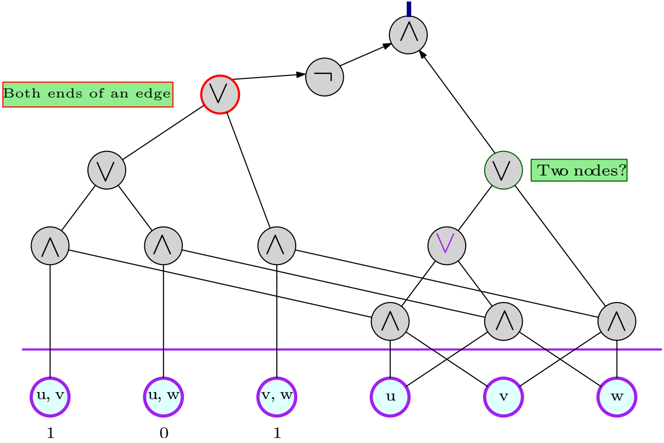


Figure: Graph G with $k = 2$



Example: Independent Set

A certifier circuit for Independent Set

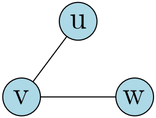
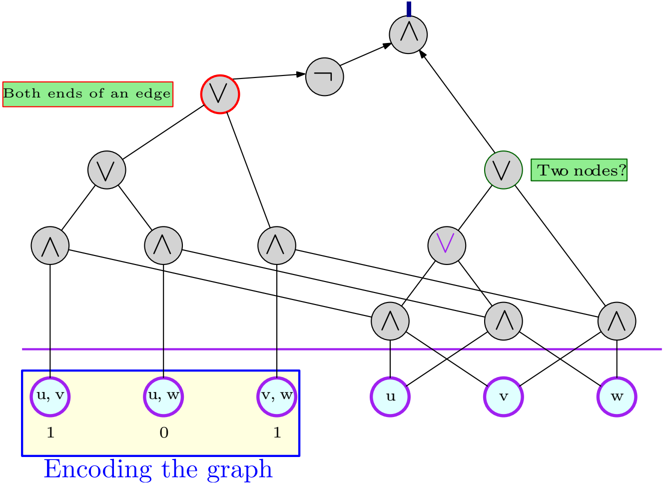


Figure: Graph G with $k = 2$



Example: Independent Set

A certifier circuit for Independent Set

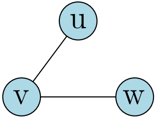
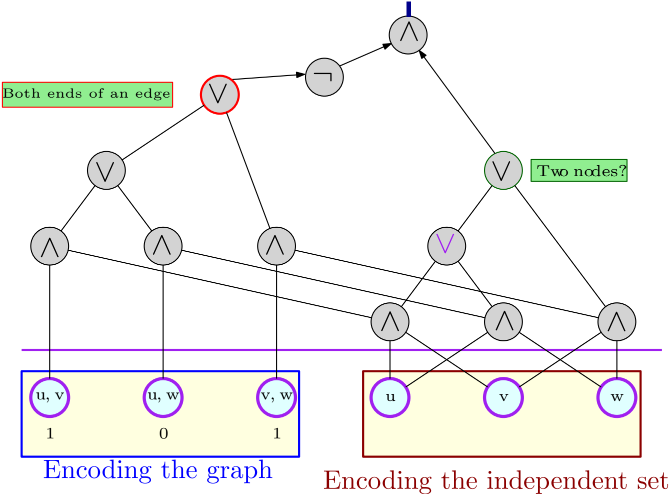


Figure: Graph G with $k = 2$



Programs, Turing Machines and Circuits

1. Consider "program" A that takes $f(|s|)$ steps on input string s.
2. Question: What computer is the program running on and what does *step* mean?
3. Real computers difficult to reason with mathematically because
   1. instruction set is too rich
   2. pointers and control flow jumps in one step
   3. assumption that pointer to code fits in one word
4. Turing Machines
   1. simpler model of computation to reason with
   2. can simulate real computers with *polynomial* slow down
   3. all moves are *local* (head moves only one cell)

Programs, Turing Machines and Circuits

1. Consider "program" A that takes $f(|s|)$ steps on input string s.
2. Question: What computer is the program running on and what does *step* mean?
3. Real computers difficult to reason with mathematically because
   1. instruction set is too rich
   2. pointers and control flow jumps in one step
   3. assumption that pointer to code fits in one word
4. Turing Machines
   1. simpler model of computation to reason with
   2. can simulate real computers with *polynomial* slow down
   3. all moves are *local* (head moves only one cell)

Programs, Turing Machines and Circuits

1. Consider "program" A that takes $f(|s|)$ steps on input string s.
2. Question: What computer is the program running on and what does *step* mean?
3. Real computers difficult to reason with mathematically because
   1. instruction set is too rich
   2. pointers and control flow jumps in one step
   3. assumption that pointer to code fits in one word
4. Turing Machines
   1. simpler model of computation to reason with
   2. can simulate real computers with *polynomial* slow down
   3. all moves are *local* (head moves only one cell)

Programs, Turing Machines and Circuits

1. Consider "program" A that takes $f(|s|)$ steps on input string s.
2. Question: What computer is the program running on and what does *step* mean?
3. Real computers difficult to reason with mathematically because
   1. instruction set is too rich
   2. pointers and control flow jumps in one step
   3. assumption that pointer to code fits in one word
4. Turing Machines
   1. simpler model of computation to reason with
   2. can simulate real computers with *polynomial* slow down
   3. all moves are *local* (head moves only one cell)

Programs, Turing Machines and Circuits

1. Consider "program" A that takes $f(|s|)$ steps on input string s.
2. Question: What computer is the program running on and what does *step* mean?
3. Real computers difficult to reason with mathematically because
   1. instruction set is too rich
   2. pointers and control flow jumps in one step
   3. assumption that pointer to code fits in one word
4. Turing Machines
   1. simpler model of computation to reason with
   2. can simulate real computers with *polynomial* slow down
   3. all moves are *local* (head moves only one cell)

Programs, Turing Machines and Circuits

1. Consider "program" A that takes $f(|s|)$ steps on input string s.
2. Question: What computer is the program running on and what does *step* mean?
3. Real computers difficult to reason with mathematically because
   1. instruction set is too rich
   2. pointers and control flow jumps in one step
   3. assumption that pointer to code fits in one word
4. Turing Machines
   1. simpler model of computation to reason with
   2. can simulate real computers with *polynomial* slow down
   3. all moves are *local* (head moves only one cell)

Programs, Turing Machines and Circuits

1. Consider "program" A that takes $f(|s|)$ steps on input string s.
2. Question: What computer is the program running on and what does *step* mean?
3. Real computers difficult to reason with mathematically because
   1. instruction set is too rich
   2. pointers and control flow jumps in one step
   3. assumption that pointer to code fits in one word
4. Turing Machines
   1. simpler model of computation to reason with
   2. can simulate real computers with *polynomial* slow down
   3. all moves are *local* (head moves only one cell)

Programs, Turing Machines and Circuits

1. Consider "program" A that takes $f(|s|)$ steps on input string s.
2. Question: What computer is the program running on and what does *step* mean?
3. Real computers difficult to reason with mathematically because
   1. instruction set is too rich
   2. pointers and control flow jumps in one step
   3. assumption that pointer to code fits in one word
4. Turing Machines
   1. simpler model of computation to reason with
   2. can simulate real computers with *polynomial* slow down
   3. all moves are *local* (head moves only one cell)

Certifiers that at TMs

1. Assume $C(\cdot, \cdot)$ is a (deterministic) Turing Machine M
2. Problem: Given M, input s, p, q decide if there is a proof t of length $p(|s|)$ such that M on s, t will halt in $q(|s|)$ time and say YES.
3. There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.
    1. $\mathcal{A}$ first computes $p(|s|)$ and $q(|s|)$.
    2. Knows that M can use at most $q(|s|)$ memory/tape cells
    3. Knows that M can run for at most $q(|s|)$ time
    4. Simulates the evolution of the state of M and memory over time using a big circuit.

Certifiers that at TMs

1. Assume $C(\cdot, \cdot)$ is a (deterministic) Turing Machine M
2. Problem: Given M, input s, p, q decide if there is a proof t of length $p(|s|)$ such that M on s, t will halt in $q(|s|)$ time and say YES.
3. There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.
    1. $\mathcal{A}$ first computes $p(|s|)$ and $q(|s|)$.
    2. Knows that M can use at most $q(|s|)$ memory/tape cells
    3. Knows that M can run for at most $q(|s|)$ time
    4. Simulates the evolution of the state of M and memory over time using a big circuit.

Certifiers that at TMs

1. Assume $C(\cdot,\cdot)$ is a (deterministic) Turing Machine M
2. Problem: Given M, input s, p, q decide if there is a proof t of length $p(|s|)$ such that M on s, t will halt in $q(|s|)$ time and say YES.
3. There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.
    1. $\mathcal{A}$ first computes $p(|s|)$ and $q(|s|)$.
    2. Knows that M can use at most $q(|s|)$ memory/tape cells
    3. Knows that M can run for at most $q(|s|)$ time
    4. Simulates the evolution of the state of M and memory over time using a big circuit.

Certifiers that at TMs

1. Assume $C(\cdot, \cdot)$ is a (deterministic) Turing Machine M
2. Problem: Given M, input s, p, q decide if there is a proof t of length $p(|s|)$ such that M on s, t will halt in $q(|s|)$ time and say YES.
3. There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.
    1. $\mathcal{A}$ first computes $p(|s|)$ and $q(|s|)$.
    2. Knows that M can use at most $q(|s|)$ memory/tape cells
    3. Knows that M can run for at most $q(|s|)$ time
    4. Simulates the evolution of the state of M and memory over time using a big circuit.

Certifiers that at TMs

1. Assume $C(\cdot, \cdot)$ is a (deterministic) Turing Machine M
2. Problem: Given M, input s, p, q decide if there is a proof t of length $p(|s|)$ such that M on s, t will halt in $q(|s|)$ time and say YES.
3. There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.
    1. $\mathcal{A}$ first computes $p(|s|)$ and $q(|s|)$.
    2. Knows that M can use at most $q(|s|)$ memory/tape cells
    3. Knows that M can run for at most $q(|s|)$ time
    4. Simulates the evolution of the state of M and memory over time using a big circuit.

Certifiers that at TMs

1. Assume $C(\cdot, \cdot)$ is a (deterministic) Turing Machine M
2. Problem: Given M, input s, p, q decide if there is a proof t of length $p(|s|)$ such that M on s, t will halt in $q(|s|)$ time and say YES.
3. There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.
    1. $\mathcal{A}$ first computes $p(|s|)$ and $q(|s|)$.
    2. Knows that M can use at most $q(|s|)$ memory/tape cells
    3. Knows that M can run for at most $q(|s|)$ time
    4. Simulates the evolution of the state of M and memory over time using a big circuit.

Certifiers that at TMs

1. Assume $C(\cdot, \cdot)$ is a (deterministic) Turing Machine M
2. Problem: Given M, input s, p, q decide if there is a proof t of length $p(|s|)$ such that M on s, t will halt in $q(|s|)$ time and say YES.
3. There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.
   1. $\mathcal{A}$ first computes $p(|s|)$ and $q(|s|)$.
   2. Knows that M can use at most $q(|s|)$ memory/tape cells
   3. Knows that M can run for at most $q(|s|)$ time
   4. Simulates the evolution of the state of M and memory over time using a big circuit.

Certifiers that at TMs

1. Assume $C(\cdot,\cdot)$ is a (deterministic) Turing Machine M
2. Problem: Given M, input s, p, q decide if there is a proof t of length $p(|s|)$ such that M on s, t will halt in $q(|s|)$ time and say YES.
3. There is an algorithm $\mathcal{A}$ that can reduce above problem to **CSAT** mechanically as follows.
   1. $\mathcal{A}$ first computes $p(|s|)$ and $q(|s|)$.
   2. Knows that M can use at most $q(|s|)$ memory/tape cells
   3. Knows that M can run for at most $q(|s|)$ time
   4. Simulates the evolution of the state of M and memory over time using a big circuit.

Simulation of Computation via Circuit

1. Think of M's state at time ℓ as a string $x^{\ell} = x_1 x_2 \ldots x_k$ where each $x_i \in \{0, 1, B\} \times Q \cup \{q_{-1}\}$.
2. At time 0 the state of M consists of input string s a guess t (unknown variables) of length $p(|s|)$ and rest $q(|s|)$ blank symbols.
3. At time $q(|s|)$ we wish to know if M stops in q_{accept} with say all blanks on the tape.
4. We write a circuit C_{ℓ} which captures the transition of M from time ℓ to time $\ell + 1$.
5. Composition of the circuits for all times 0 to $q(|s|)$ gives a big (still poly) sized circuit $\mathcal{C}$
6. The final output of $\mathcal{C}$ should be true if and only if the entire state of M at the end leads to an accept state.

Simulation of Computation via Circuit

1. Think of M's state at time ℓ as a string $x^\ell = x_1 x_2 \ldots x_k$ where each $x_i \in \{0, 1, B\} \times Q \cup \{q_{-1}\}$.
2. At time 0 the state of M consists of input string s a guess t (unknown variables) of length $p(|s|)$ and rest $q(|s|)$ blank symbols.
3. At time $q(|s|)$ we wish to know if M stops in q_{accept} with say all blanks on the tape.
4. We write a circuit C_ℓ which captures the transition of M from time ℓ to time $\ell + 1$.
5. Composition of the circuits for all times 0 to $q(|s|)$ gives a big (still poly) sized circuit $\mathcal{C}$
6. The final output of $\mathcal{C}$ should be true if and only if the entire state of M at the end leads to an accept state.

Simulation of Computation via Circuit

1. Think of M's state at time ℓ as a string $x^{\ell} = x_1 x_2 \ldots x_k$ where each $x_i \in \{0, 1, B\} \times Q \cup \{q_{-1}\}$.
2. At time 0 the state of M consists of input string s a guess t (unknown variables) of length $p(|s|)$ and rest $q(|s|)$ blank symbols.
3. At time $q(|s|)$ we wish to know if M stops in q_{accept} with say all blanks on the tape.
4. We write a circuit C_{ℓ} which captures the transition of M from time ℓ to time $\ell + 1$.
5. Composition of the circuits for all times 0 to $q(|s|)$ gives a big (still poly) sized circuit $\mathcal{C}$
6. The final output of $\mathcal{C}$ should be true if and only if the entire state of M at the end leads to an accept state.

Simulation of Computation via Circuit

1. Think of M's state at time ℓ as a string $x^{\ell} = x_1 x_2 \ldots x_k$ where each $x_i \in \{0, 1, B\} \times Q \cup \{q_{-1}\}$.
2. At time 0 the state of M consists of input string s a guess t (unknown variables) of length $p(|s|)$ and rest $q(|s|)$ blank symbols.
3. At time $q(|s|)$ we wish to know if M stops in q_{accept} with say all blanks on the tape.
4. We write a circuit C_{ℓ} which captures the transition of M from time ℓ to time $\ell + 1$.
5. Composition of the circuits for all times 0 to $q(|s|)$ gives a big (still poly) sized circuit $\mathcal{C}$
6. The final output of $\mathcal{C}$ should be true if and only if the entire state of M at the end leads to an accept state.

Simulation of Computation via Circuit

1. Think of M's state at time ℓ as a string $x^\ell = x_1 x_2 \ldots x_k$ where each $x_i \in \{0, 1, B\} \times Q \cup \{q_{-1}\}$.
2. At time 0 the state of M consists of input string s a guess t (unknown variables) of length $p(|s|)$ and rest $q(|s|)$ blank symbols.
3. At time $q(|s|)$ we wish to know if M stops in q_{accept} with say all blanks on the tape.
4. We write a circuit C_ℓ which captures the transition of M from time ℓ to time $\ell + 1$.
5. Composition of the circuits for all times 0 to $q(|s|)$ gives a big (still poly) sized circuit $\mathcal{C}$
6. The final output of $\mathcal{C}$ should be true if and only if the entire state of M at the end leads to an accept state.

Simulation of Computation via Circuit

1. Think of M's state at time ℓ as a string $x^\ell = x_1 x_2 \ldots x_k$ where each $x_i \in \{0, 1, B\} \times Q \cup \{q_{-1}\}$.
2. At time 0 the state of M consists of input string s a guess t (unknown variables) of length $p(|s|)$ and rest $q(|s|)$ blank symbols.
3. At time $q(|s|)$ we wish to know if M stops in q_{accept} with say all blanks on the tape.
4. We write a circuit C_ℓ which captures the transition of M from time ℓ to time $\ell + 1$.
5. Composition of the circuits for all times 0 to $q(|s|)$ gives a big (still poly) sized circuit $\mathcal{C}$
6. The final output of $\mathcal{C}$ should be true if and only if the entire state of M at the end leads to an accept state.

NP-Hardness of Circuit Satisfaction

1. Key Ideas in reduction:
   1. Use TMs as the code for certifier for simplicity
   2. Since $p()$ and $q()$ are known to $\mathcal{A}$, it can set up all required memory and time steps in advance
   3. Simulate computation of the TM from one time to the next as a circuit that only looks at three adjacent cells at a time
2. Note: Above reduction can be done to **SAT** as well. Reduction to **SAT** was the original proof of Steve Cook.

NP-Hardness of Circuit Satisfaction

1. Key Ideas in reduction:
   1. Use TMs as the code for certifier for simplicity
   2. Since $p()$ and $q()$ are known to $\mathcal{A}$, it can set up all required memory and time steps in advance
   3. Simulate computation of the TM from one time to the next as a circuit that only looks at three adjacent cells at a time
2. Note: Above reduction can be done to **SAT** as well. Reduction to **SAT** was the original proof of Steve Cook.

NP-Hardness of Circuit Satisfaction

1. Key Ideas in reduction:
   1. Use TMs as the code for certifier for simplicity
   2. Since $p()$ and $q()$ are known to $\mathcal{A}$, it can set up all required memory and time steps in advance
   3. Simulate computation of the TM from one time to the next as a circuit that only looks at three adjacent cells at a time
2. Note: Above reduction can be done to **SAT** as well. Reduction to **SAT** was the original proof of Steve Cook.

NP-Hardness of Circuit Satisfaction

1. Key Ideas in reduction:
   1. Use TMs as the code for certifier for simplicity
   2. Since $p()$ and $q()$ are known to $\mathcal{A}$, it can set up all required memory and time steps in advance
   3. Simulate computation of the TM from one time to the next as a circuit that only looks at three adjacent cells at a time
2. Note: Above reduction can be done to **SAT** as well. Reduction to **SAT** was the original proof of Steve Cook.

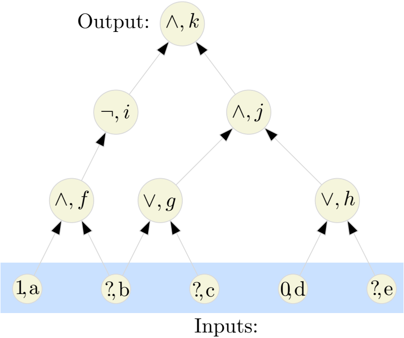
NP-Hardness of Circuit Satisfaction

1. Key Ideas in reduction:
   1. Use TMs as the code for certifier for simplicity
   2. Since $p()$ and $q()$ are known to $\mathcal{A}$, it can set up all required memory and time steps in advance
   3. Simulate computation of the TM from one time to the next as a circuit that only looks at three adjacent cells at a time
2. Note: Above reduction can be done to **SAT** as well. Reduction to **SAT** was the original proof of Steve Cook.

23.2.4: Other NP Complete Problems

SAT is NP-Complete

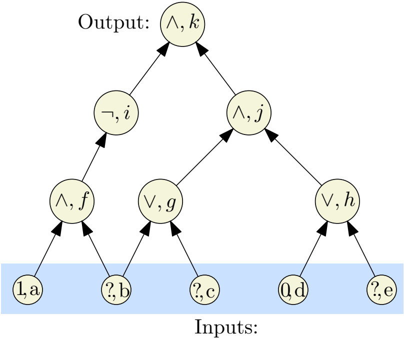
1. We have seen that **SAT** $\in$ **NP**
2. To show **NP-Hardness**, we will reduce Circuit Satisfiability (**CSAT**) to **SAT**
   Instance of **CSAT** (we label each node):



SAT is NP-Complete

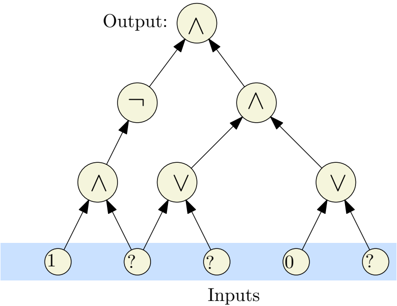
1. We have seen that **SAT** $\in$ **NP**
2. To show **NP-Hardness**, we will reduce Circuit Satisfiability (**CSAT**) to **SAT**

   Instance of **CSAT** (we label each node):

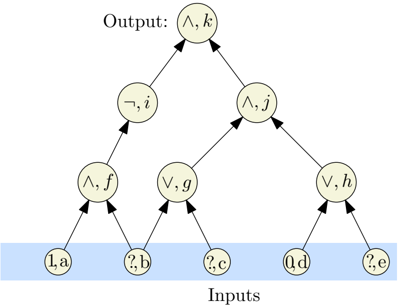


Converting a circuit into a CNF formula

Label the nodes



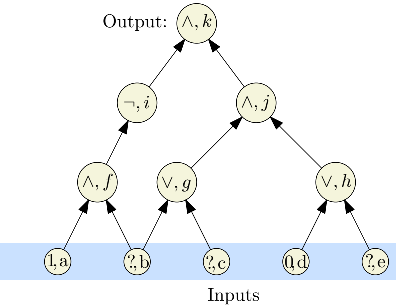
(A) Input circuit



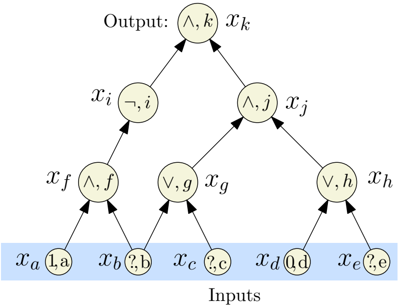
(B) Label the nodes.

Converting a circuit into a CNF formula

Introduce a variable for each node



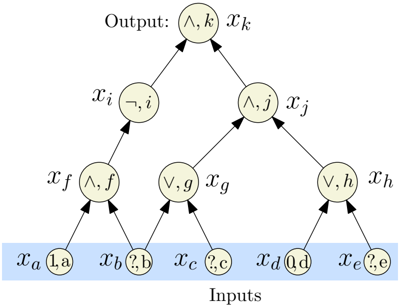
(B) Label the nodes.



(C) Introduce var for each node.

Converting a circuit into a CNF formula

Write a sub-formula for each variable that is true if the var is computed correctly.



(C) Introduce var for each node.

x_k (Demand a sat' assignment!)

$x_k = x_i \wedge x_k$

$x_j = x_g \wedge x_h$

$x_i = \neg x_f$

$x_h = x_d \vee x_e$

$x_g = x_b \vee x_c$

$x_f = x_a \wedge x_b$

$x_d = 0$

$x_a = 1$

(D) Write a sub-formula for each variable that is true if the var is computed correctly.

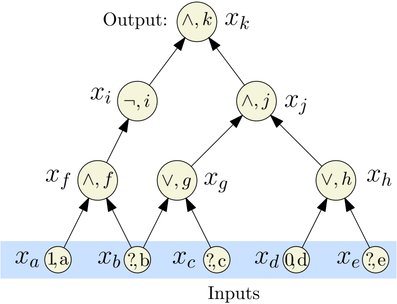
Converting a circuit into a CNF formula

Convert each sub-formula to an equivalent CNF formula

x_k	x_k
$x_k = x_i \wedge x_j$	$(\neg x_k \vee x_i) \wedge (\neg x_k \vee x_j) \wedge (x_k \vee \neg x_i \vee \neg x_j)$
$x_j = x_g \wedge x_h$	$(\neg x_j \vee x_g) \wedge (\neg x_j \vee x_h) \wedge (x_j \vee \neg x_g \vee \neg x_h)$
$x_i = \neg x_f$	$(x_i \vee x_f) \wedge (\neg x_i \vee x_f)$
$x_h = x_d \vee x_e$	$(x_h \vee \neg x_d) \wedge (x_h \vee \neg x_e) \wedge (\neg x_h \vee x_d \vee x_e)$
$x_g = x_b \vee x_c$	$(x_g \vee \neg x_b) \wedge (x_g \vee \neg x_c) \wedge (\neg x_g \vee x_b \vee x_c)$
$x_f = x_a \wedge x_b$	$(\neg x_f \vee x_a) \wedge (\neg x_f \vee x_b) \wedge (x_f \vee \neg x_a \vee \neg x_b)$
$x_d = 0$	$\neg x_d$
$x_a = 1$	x_a

Converting a circuit into a CNF formula

Take the conjunction of all the CNF sub-formulas



$$
\begin{aligned}
& x_k \wedge (\neg x_k \vee x_i) \wedge (\neg x_k \vee x_j) \\
& \wedge (x_k \vee \neg x_i \vee \neg x_j) \wedge (\neg x_j \vee x_g) \\
& \wedge (\neg x_j \vee x_h) \wedge (x_j \vee \neg x_g \vee \neg x_h) \\
& \wedge (x_i \vee x_f) \wedge (\neg x_i \vee x_f) \\
& \wedge (x_h \vee \neg x_d) \wedge (x_h \vee \neg x_e) \\
& \wedge (\neg x_h \vee x_d \vee x_e) \wedge (x_g \vee \neg x_b) \\
& \wedge (x_g \vee \neg x_c) \wedge (\neg x_g \vee x_b \vee x_c) \\
& \wedge (\neg x_f \vee x_a) \wedge (\neg x_f \vee x_b) \\
& \wedge (x_f \vee \neg x_a \vee \neg x_b) \wedge ([)] \neg x_d \wedge x_a
\end{aligned}
$$

We got a CNF formula that is satisfiable if and only if the original circuit is satisfiable.

Reduction: CSAT $\leq_P$ SAT

1. For each gate (vertex) v in the circuit, create a variable x_v
2. Case $\neg$: v is labeled $\neg$ and has one incoming edge from u (so $x_v = \neg x_u$). In **SAT** formula generate, add clauses $(x_u \vee x_v)$, $(\neg x_u \vee \neg x_v)$. Observe that

$$x_v = \neg x_u \text{ is true} \quad \iff \quad \begin{array}{l} (x_u \vee x_v) \\ (\neg x_u \vee \neg x_v) \end{array} \text{ both true.}$$

Reduction: CSAT $\leq_P$ SAT

Continued...

1. Case $\vee$: So $x_v = x_u \vee x_w$. In **SAT** formula generated, add clauses $(x_v \vee \neg x_u)$, $(x_v \vee \neg x_w)$, and $(\neg x_v \vee x_u \vee x_w)$. Again, observe that

$$\Big(x_v = x_u \vee x_w\Big) \text{ is true} \quad \Longleftrightarrow \quad \begin{array}{l} (x_v \vee \neg x_u), \\ (x_v \vee \neg x_w), \\ (\neg x_v \vee x_u \vee x_w) \end{array} \quad \text{all true.}$$

1. Case $\wedge$: So $x_v = x_u \wedge x_w$. In **SAT** formula generated, add clauses $(\neg x_v \vee x_u)$, $(\neg x_v \vee x_w)$, and $(x_v \vee \neg x_u \vee \neg x_w)$. Again observe that

$$x_v = x_u \wedge x_w \text{ is true} \iff \begin{array}{l} (\neg x_v \vee x_u), \\ (\neg x_v \vee x_w), \\ (x_v \vee \neg x_u \vee \neg x_w) \end{array} \text{ all true.}$$

Reduction: CSAT $\leq_P$ SAT

Continued...

1. If v is an input gate with a fixed value then we do the following. If $x_v = 1$ add clause x_v. If $x_v = 0$ add clause $\neg x_v$
2. Add the clause x_v where v is the variable for the output gate

Correctness of Reduction

Need to show circuit C is satisfiable iff φ_C is satisfiable

$\Rightarrow$ Consider a satisfying assignment a for C

1. Find values of all gates in C under a
2. Give value of gate v to variable x_v; call this assignment a'
3. a' satisfies φ_C (exercise)

$\Leftarrow$ Consider a satisfying assignment a for φ_C

1. Let a' be the restriction of a to only the input variables
2. Value of gate v under a' is the same as value of x_v in a
3. Thus, a' satisfies C

Correctness of Reduction

Need to show circuit C is satisfiable iff φ_C is satisfiable

$\Rightarrow$ Consider a satisfying assignment a for C

1. Find values of all gates in C under a
2. Give value of gate v to variable x_v; call this assignment a'
3. a' satisfies φ_C (exercise)

$\Leftarrow$ Consider a satisfying assignment a for φ_C

1. Let a' be the restriction of a to only the input variables
2. Value of gate v under a' is the same as value of x_v in a
3. Thus, a' satisfies C

Correctness of Reduction

Need to show circuit C is satisfiable iff φ_C is satisfiable

$\Rightarrow$ Consider a satisfying assignment a for C

1. Find values of all gates in C under a
2. Give value of gate v to variable x_v; call this assignment a'
3. a' satisfies φ_C (exercise)

$\Leftarrow$ Consider a satisfying assignment a for φ_C

1. Let a' be the restriction of a to only the input variables
2. Value of gate v under a' is the same as value of x_v in a
3. Thus, a' satisfies C

Correctness of Reduction

Need to show circuit C is satisfiable iff φ_C is satisfiable

$\Rightarrow$ Consider a satisfying assignment a for C

1. Find values of all gates in C under a
2. Give value of gate v to variable x_v; call this assignment a'
3. a' satisfies φ_C (exercise)

$\Leftarrow$ Consider a satisfying assignment a for φ_C

1. Let a' be the restriction of a to only the input variables
2. Value of gate v under a' is the same as value of x_v in a
3. Thus, a' satisfies C

Correctness of Reduction

Need to show circuit C is satisfiable iff φ_C is satisfiable

$\Rightarrow$ Consider a satisfying assignment a for C

1. Find values of all gates in C under a
2. Give value of gate v to variable x_v; call this assignment a'
3. a' satisfies φ_C (exercise)

$\Leftarrow$ Consider a satisfying assignment a for φ_C

1. Let a' be the restriction of a to only the input variables
2. Value of gate v under a' is the same as value of x_v in a
3. Thus, a' satisfies C

Correctness of Reduction

Need to show circuit C is satisfiable iff φ_C is satisfiable

$\Rightarrow$ Consider a satisfying assignment a for C

1. Find values of all gates in C under a
2. Give value of gate v to variable x_v; call this assignment a'
3. a' satisfies φ_C (exercise)

$\Leftarrow$ Consider a satisfying assignment a for φ_C

1. Let a' be the restriction of a to only the input variables
2. Value of gate v under a' is the same as value of x_v in a
3. Thus, a' satisfies C

Correctness of Reduction

Need to show circuit C is satisfiable iff φ_C is satisfiable

$\Rightarrow$ Consider a satisfying assignment a for C

1. Find values of all gates in C under a
2. Give value of gate v to variable x_v; call this assignment a'
3. a' satisfies φ_C (exercise)

$\Leftarrow$ Consider a satisfying assignment a for φ_C

1. Let a' be the restriction of a to only the input variables
2. Value of gate v under a' is the same as value of x_v in a
3. Thus, a' satisfies C

Correctness of Reduction

Need to show circuit C is satisfiable iff φ_C is satisfiable

$\Rightarrow$ Consider a satisfying assignment a for C

1. Find values of all gates in C under a
2. Give value of gate v to variable x_v; call this assignment a'
3. a' satisfies φ_C (exercise)

$\Leftarrow$ Consider a satisfying assignment a for φ_C

1. Let a' be the restriction of a to only the input variables
2. Value of gate v under a' is the same as value of x_v in a
3. Thus, a' satisfies C

Correctness of Reduction

Need to show circuit C is satisfiable iff φ_C is satisfiable

$\Rightarrow$ Consider a satisfying assignment a for C

1. Find values of all gates in C under a
2. Give value of gate v to variable x_v; call this assignment a'
3. a' satisfies φ_C (exercise)

$\Leftarrow$ Consider a satisfying assignment a for φ_C

1. Let a' be the restriction of a to only the input variables
2. Value of gate v under a' is the same as value of x_v in a
3. Thus, a' satisfies C

Showed that...

Theorem

SAT *is* **NP-Complete**.

Proving that a problem X is NP-Complete

1. To prove X is **NP-Complete**, show
   1. Show X is in **NP**.
      1. certificate/proof of polynomial size in input
      2. polynomial time certifier $C(s, t)$
   2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X
2. SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?
3. Transitivity of reductions:
4. $Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

Proving that a problem X is NP-Complete

1. To prove X is **NP-Complete**, show
   1. Show X is in **NP**.
      1. certificate/proof of polynomial size in input
      2. polynomial time certifier $C(s, t)$
   2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X
2. SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?
3. Transitivity of reductions:
4. $Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

Proving that a problem X is NP-Complete

1. To prove X is **NP-Complete**, show
    1. Show X is in **NP**.
        1. certificate/proof of polynomial size in input
        2. polynomial time certifier $C(s, t)$
    2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X
2. SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?
3. Transitivity of reductions:
4. $Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

Proving that a problem X is NP-Complete

1. To prove X is **NP-Complete**, show
   1. Show X is in **NP**.
      1. certificate/proof of polynomial size in input
      2. polynomial time certifier $C(s, t)$
   2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X
2. SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?
3. Transitivity of reductions:
4. $Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

Proving that a problem X is NP-Complete

1. To prove X is **NP-Complete**, show
   1. Show X is in **NP**.
      1. certificate/proof of polynomial size in input
      2. polynomial time certifier $C(s,t)$
   2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X
2. SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?
3. Transitivity of reductions:
4. $Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

Proving that a problem X is NP-Complete

1. To prove X is **NP-Complete**, show
   1. Show X is in **NP**.
      1. certificate/proof of polynomial size in input
      2. polynomial time certifier $C(s, t)$
   2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X
2. SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?
3. Transitivity of reductions:
4. $Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

Proving that a problem **X** is **NP-Complete**

1. To prove X is **NP-Complete**, show
   1. Show X is in **NP**.
      1. certificate/proof of polynomial size in input
      2. polynomial time certifier $C(s, t)$
   2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X
2. SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?
3. Transitivity of reductions:
4. $Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

Proving that a problem **X** is **NP-Complete**

1. To prove X is **NP-Complete**, show
    1. Show X is in **NP**.
        1. certificate/proof of polynomial size in input
        2. polynomial time certifier $C(s,t)$
    2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X
2. SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?
3. Transitivity of reductions:
4. $Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

Proving that a problem **X** is **NP-Complete**

1. To prove X is **NP-Complete**, show
   1. Show X is in **NP**.
      1. certificate/proof of polynomial size in input
      2. polynomial time certifier $C(s, t)$
   2. Reduction from a known **NP-Complete** problem such as **CSAT** or **SAT** to X
2. SAT $\leq_P$ X implies that every **NP** problem $Y \leq_P X$. Why?
3. Transitivity of reductions:
4. $Y \leq_P SAT$ and $SAT \leq_P X$ and hence $Y \leq_P X$.

NP-Completeness via Reductions

1. What we know so far:
   1. **CSAT** is **NP-Complete**.
   2. **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**.
   3. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
   4. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
   5. **Vertex Cover** is **NP-Complete**.
   6. **Clique** is **NP-Complete**.
2. Gazillion of different problems from many areas of science and engineering have been shown to be **NP-Complete**.
3. A surprisingly frequent phenomenon!

NP-Completeness via Reductions

1. What we know so far:
   1. **CSAT** is **NP-Complete**.
   2. **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**.
   3. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
   4. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
   5. **Vertex Cover** is **NP-Complete**.
   6. **Clique** is **NP-Complete**.
2. Gazillion of different problems from many areas of science and engineering have been shown to be **NP-Complete**.
3. A surprisingly frequent phenomenon!

NP-Completeness via Reductions

1. What we know so far:
   1. **CSAT** is **NP-Complete**.
   2. **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**.
   3. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
   4. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
   5. **Vertex Cover** is **NP-Complete**.
   6. **Clique** is **NP-Complete**.
2. Gazillion of different problems from many areas of science and engineering have been shown to be **NP-Complete**.
3. A surprisingly frequent phenomenon!

NP-Completeness via Reductions

1. What we know so far:
   1. **CSAT** is **NP-Complete**.
   2. **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**.
   3. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
   4. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
   5. **Vertex Cover** is **NP-Complete**.
   6. **Clique** is **NP-Complete**.
2. Gazillion of different problems from many areas of science and engineering have been shown to be **NP-Complete**.
3. A surprisingly frequent phenomenon!

NP-Completeness via Reductions

1. What we know so far:
   1. **CSAT** is **NP-Complete**.
   2. **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**.
   3. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
   4. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
   5. **Vertex Cover** is **NP-Complete**.
   6. **Clique** is **NP-Complete**.
2. Gazillion of different problems from many areas of science and engineering have been shown to be **NP-Complete**.
3. A surprisingly frequent phenomenon!

NP-Completeness via Reductions

1. What we know so far:
   1. **CSAT** is **NP-Complete**.
   2. **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**.
   3. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
   4. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
   5. **Vertex Cover** is **NP-Complete**.
   6. **Clique** is **NP-Complete**.
2. Gazillion of different problems from many areas of science and engineering have been shown to be **NP-Complete**.
3. A surprisingly frequent phenomenon!

NP-Completeness via Reductions

1. What we know so far:
    1. **CSAT** is **NP-Complete**.
    2. **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**.
    3. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
    4. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
    5. **Vertex Cover** is **NP-Complete**.
    6. **Clique** is **NP-Complete**.
2. Gazillion of different problems from many areas of science and engineering have been shown to be **NP-Complete**.
3. A surprisingly frequent phenomenon!

NP-Completeness via Reductions

1. What we know so far:
    1. **CSAT** is **NP-Complete**.
    2. **CSAT** $\leq_P$ **SAT** and **SAT** is in **NP** and hence **SAT** is **NP-Complete**.
    3. **SAT** $\leq_P$ **3-SAT** and hence 3-SAT is **NP-Complete**.
    4. **3-SAT** $\leq_P$ **Independent Set** (which is in **NP**) and hence **Independent Set** is **NP-Complete**.
    5. **Vertex Cover** is **NP-Complete**.
    6. **Clique** is **NP-Complete**.
2. Gazillion of different problems from many areas of science and engineering have been shown to be **NP-Complete**.
3. A surprisingly frequent phenomenon!

Notes

Notes

Notes

Notes

S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM J. Comput.*, 5(4):691–703, 1976.