Reductions and Completeness

K. Subramani

Lane Department of Computer Science and Electrical Engineering
West Virginia University

February 24, 2009
Outline

1. Reductions

2. Completeness
   - P-completeness
   - NP-completeness
1. Reductions

2. Completeness
   - $P$-completeness
   - $NP$-completeness
Reductions

Main concept

Comparing problem difficulty through $A \leq B$. When is problem $B$ at least as hard as problem $A$? When there is a transformation $R$, which for every input of $A$ produces an equivalent input $R(x)$ of $B$ such that $x \in A \iff R(x) \in B$.

Note

To be useful, $R$ should have limitations. (Hamilton Path to Reachability).

Definition

A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings computable by a DTM in space $O(\log n)$, such that for all inputs $x \in \Sigma^*$, $|x| = n$, $x \in L_1 \iff R(x) \in L_2$. 
Main concept

Comparing problem difficulty through \( A \leq B \). When is problem \( B \) at least as hard as problem \( A \)?

When there is a transformation \( R \), which for every input of \( A \) produces an equivalent input \( R(x) \) of \( B \) such that \( x \in A \iff R(x) \in B \).

Note

To be useful, \( R \) should have limitations. (Hamilton Path to Reachability).

Definition

A language \( L_1 \) is reducible to a language \( L_2 \) if there is a function \( R \) from strings of \( L_1 \) to strings computable by a DTM in space \( O(\log n) \), such that for all inputs \( x \in \Sigma^* \), \( |x| = n \), \( x \in L_1 \iff R(x) \in L_2 \).
Main concept

Comparing problem difficulty through $A \leq B$. When is problem $B$ at least as hard as problem $A$? 
When there is a transformation $R$, which for every input of $A$ produces an equivalent input $R(x)$ of $B$ such that $x \in A \iff R(x) \in B$.

Note

To be useful, $R$ should have limitations. (Hamilton Path to Reachability).

Definition

A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings computable by a DTM in space $O(\log n)$, such that for all inputs $x \in \Sigma^*$, $|x| = n$, 
$x \in L_1 \iff R(x) \in L_2$. 

Subramani

Complexity Classes
Main concept
Comparing problem difficulty through $A \leq B$. When is problem $B$ at least as hard as problem $A$? When there is a transformation $R$, which for every input of $A$ produces an equivalent input $R(x)$ of $B$ such that $x \in A \iff R(x) \in B$.

Note
To be useful, $R$ should have limitations. (Hamilton Path to Reachability).

Definition
A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings computable by a DTM in space $O(\log n)$, such that for all inputs $x \in \Sigma^*$, $|x| = n$, $x \in L_1 \iff R(x) \in L_2$. 

Subramani
Complexity Classes
Main concept

Comparing problem difficulty through $A \leq B$. When is problem $B$ at least as hard as problem $A$?

When there is a transformation $R$, which for every input of $A$ produces an equivalent input $R(x)$ of $B$ such that $x \in A \iff R(x) \in B$.

Note

To be useful, $R$ should have limitations. (Hamilton Path to Reachability).

Definition

A language $L_1$ is reducible to a language $L_2$ if there is a function $R$ from strings of $L_1$ to strings computable by a DTM in space $O(\log n)$, such that for all inputs $x \in \Sigma^*$, $|x| = n$,

$x \in L_1 \iff R(x) \in L_2$. 
Reductions (contd.)

Note

Good old days, we used poly-time reductions.

Proposition

If $R$ is a reduction computed by a DTM $M$, then for all $x$, $M$ halts after a polynomial number of steps.
Note

Good old days, we used poly-time reductions.

Proposition

If $R$ is a reduction computed by a DTM $M$, then for all $x$, $M$ halts after a polynomial number of steps.
Note

Good old days, we used poly-time reductions.

Proposition

If $R$ is a reduction computed by a DTM $M$, then for all $x$, $M$ halts after a polynomial number of steps.
Sample Reductions

Hamilton Path to SAT

Input instance: An unweighted, directed graph $G$.
Output instance: A CNF formula $\phi$, such that $G$ has a Hamilton path if and only if $\phi$ is satisfiable.
Step 1: Suppose $G$ has $n$ nodes; $\phi$ has $n^2$ variables of the form $x_{ij}$, where $x_{ij}$ represents the fact that node $j$ is the $i^{th}$ node in the Hamilton Path (may or may not be true).
Step 2: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. [$C_1$].
Step 3: $(\neg x_{ij} \lor \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. [$C_2$].
Step 4: $(x_{i1} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2 \ldots n$. [$C_3$].
Step 5: $(\neg x_{ij} \lor \neg x_{jk}), i = 1, 2, \ldots n, j, k = 1, 2, \ldots, n, j \neq k$. [$C_4$].
Step 6: $(\neg x_{ki} \lor \neg x_{(k+1)j}), k = 1, 2, \ldots, n - 1, (i, j) \notin G$. [$C_5$].
Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

Argument: Let $\tilde{x}$ denote a satisfying assignment to $\phi$. We show that there must exist a Hamilton Path in $G$.
Let $\pi = (\pi(1), \pi(2) \ldots \pi(n))$ denote a Hamilton path, where $\pi$ is a permutation. We show that $\phi$ is satisfiable.
Sample Reductions

Hamilton Path to SAT

Input instance: An unweighted, directed graph $G$.
Output instance: A CNF formula $\phi$, such that $G$ has a Hamilton path if and only if $\phi$ is satisfiable.
Step 1: Suppose $G$ has $n$ nodes; $\phi$ has $n^2$ variables of the form $x_{ij}$, where $x_{ij}$ represents the fact that node $j$ is the $i^{th}$ node in the Hamilton Path (may or may not be true).
Step 2: $(x_{1j} \lor x_{2j} \lor \ldots \lor x_{nj}), j = 1, 2, \ldots, n$. [$C_1$].
Step 3: $(\neg x_{ij} \lor \neg x_{kj}), j = 1, 2, \ldots n, i = 1, 2, \ldots n, k = 1, 2, \ldots n, k \neq i$. [$C_2$].
Step 4: $(x_{i1} \lor x_{i2} \lor \ldots \lor x_{in}), i = 1, 2, \ldots n$. [$C_3$].
Step 5: $(\neg x_{ij} \lor \neg x_{jk}), i = 1, 2, \ldots n, j, k = 1, 2, \ldots n, j \neq k$. [$C_4$].
Step 6: $(\neg x_{ki} \lor \neg x_{(k+1)i}), k = 1, 2, \ldots, n-1, (i, j) \notin G$. [$C_5$].
Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

Argument: Let $\bar{x}$ denote a satisfying assignment to $\phi$. We show that there must exist a Hamilton Path in $G$.
Let $\pi = (\pi(1), \pi(2) \ldots \pi(n))$ denote a Hamilton path, where $\pi$ is a permutation. We show that $\phi$ is satisfiable.
Sample Reductions

Hamilton Path to SAT

Input instance: An unweighted, directed graph $G$.
Output instance: A CNF formula $\phi$, such that $G$ has a Hamilton path if and only if $\phi$ is satisfiable.

Step 1: Suppose $G$ has $n$ nodes; $\phi$ has $n^2$ variables of the form $x_{ij}$, where $x_{ij}$ represents the fact that node $j$ is the $i^{th}$ node in the Hamilton Path (may or may not be true).

Step 2: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. [$C_1$].

Step 3: $(\neg x_{ij} \lor \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. [$C_2$].

Step 4: $(x_{1i} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2 \ldots n$. [$C_3$].

Step 5: $(\neg x_{ij} \lor \neg x_{jk}), i = 1, 2, \ldots n, j, k = 1, 2, \ldots, n, j \neq k$. [$C_4$].

Step 6: $(\neg x_{ki} \lor \neg x_{(k+1)i}), k = 1, 2, \ldots, n - 1, (i, j) \notin G$. [$C_5$].

Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

Argument: Let $\tilde{x}$ denote a satisfying assignment to $\phi$. We show that there must exist a Hamilton Path in $G$.

Let $\pi = (\pi(1), \pi(2) \ldots \pi(n))$ denote a Hamilton path, where $\pi$ is a permutation. We show that $\phi$ is satisfiable.
Sample Reductions

Hamilton Path to SAT

Input instance: An unweighted, directed graph \( G \).
Output instance: A CNF formula \( \phi \), such that \( G \) has a Hamilton path if and only if \( \phi \) is satisfiable.

Step 1: Suppose \( G \) has \( n \) nodes; \( \phi \) has \( n^2 \) variables of the form \( x_{ij} \), where \( x_{ij} \) represents the fact that node \( j \) is the \( i^{th} \) node in the Hamilton Path (may or may not be true).

Step 2: \((x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n. [C_1]\). 

Step 3: \((-x_{ij} \lor -x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i. [C_2] \). 

Step 4: \((x_{i1} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2 \ldots n. [C_3] \). 

Step 5: \((-x_{ij} \lor -x_{jk}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k. [C_4] \). 

Step 6: \((-x_{ki} \lor -x_{(k+1)j}, k = 1, 2, \ldots, n - 1, (i, j) \notin G. [C_5] \). 

Step 7: \( \phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5 \). 

Argument: Let \( \tilde{x} \) denote a satisfying assignment to \( \phi \). We show that there must exist a Hamilton Path in \( G \).

Let \( \pi = (\pi(1), \pi(2) \ldots \pi(n)) \) denote a Hamilton path, where \( \pi \) is a permutation. We show that \( \phi \) is satisfiable.
Sample Reductions

Hamilton Path to SAT

Input instance: An unweighted, directed graph $G$.
Output instance: A CNF formula $\phi$, such that $G$ has a Hamilton path if and only if $\phi$ is satisfiable.

Step 1: Suppose $G$ has $n$ nodes; $\phi$ has $n^2$ variables of the form $x_{ij}$, where $x_{ij}$ represents the fact that node $j$ is the $i^{th}$ node in the Hamilton Path (may or may not be true).

Step 2: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. [$C_1$].

Step 3: $(\neg x_{ij} \lor \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. [$C_2$].

Step 4: $(x_{i1} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2 \ldots n$. [$C_3$].

Step 5: $(\neg x_{ij} \lor \neg x_{jk}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. [$C_4$].

Step 6: $(\neg x_{ki} \lor \neg x_{(k+1)j}, k = 1, 2, \ldots, n-1, (i,j) \notin G$. [$C_5$].

Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

Argument: Let $\tilde{x}$ denote a satisfying assignment to $\phi$. We show that there must exist a Hamilton Path in $G$.

Let $\pi = (\pi(1), \pi(2) \ldots \pi(n))$ denote a Hamilton path, where $\pi$ is a permutation. We show that $\phi$ is satisfiable.
Sample Reductions

Hamilton Path to SAT

Input instance: An unweighted, directed graph $G$.
Output instance: A CNF formula $\phi$, such that $G$ has a Hamilton path if and only if $\phi$ is satisfiable.

Step 1: Suppose $G$ has $n$ nodes; $\phi$ has $n^2$ variables of the form $x_{ij}$, where $x_{ij}$ represents the fact that node $j$ is the $i^{th}$ node in the Hamilton Path (may or may not be true).
Step 2: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. [$C_1$].
Step 3: $(-x_{ij} \lor -x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. [$C_2$].
Step 4: $(x_{i1} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2 \ldots n$. [$C_3$].
Step 5: $(-x_{ij} \lor -x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. [$C_4$].
Step 6: $(-x_{ki} \lor -x_{(k+1)i}), k = 1, 2, \ldots, n - 1, (i, j) \notin G$. [$C_5$].
Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

Argument: Let $\tilde{x}$ denote a satisfying assignment to $\phi$. We show that there must exist a Hamilton Path in $G$.

Let $\pi = (\pi(1), \pi(2) \ldots \pi(n))$ denote a Hamilton path, where $\pi$ is a permutation. We show that $\phi$ is satisfiable.
Sample Reductions

Hamilton Path to SAT

Input instance: An unweighted, directed graph G.
Output instance: A CNF formula φ, such that G has a Hamilton path if and only if φ is satisfiable.
Step 1: Suppose G has n nodes; φ has $n^2$ variables of the form $x_{ij}$, where $x_{ij}$ represents the fact that node j is the $i^{th}$ node in the Hamilton Path (may or may not be true).
Step 2: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. [C1].
Step 3: $(\neg x_{ij} \lor \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. [C2].
Step 4: $(x_{i1} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2 \ldots n$. [C3].
Step 5: $(\neg x_{ij} \lor \neg x_{jk}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. [C4].
Step 6: $(\neg x_{ki} \lor \neg x_{(k+1)i}, k = 1, 2, \ldots, n - 1, (i, j) \not\in G$. [C5].
Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

Argument: Let $\bar{x}$ denote a satisfying assignment to $\phi$. We show that there must exist a Hamilton Path in G.
Let $\pi = (\pi(1), \pi(2) \ldots \pi(n))$ denote a Hamilton path, where $\pi$ is a permutation. We show that $\phi$ is satisfiable.
Sample Reductions

Hamilton Path to SAT

Input instance: An unweighted, directed graph $G$.
Output instance: A CNF formula $\phi$, such that $G$ has a Hamilton path if and only if $\phi$ is satisfiable.

Step 1: Suppose $G$ has $n$ nodes; $\phi$ has $n^2$ variables of the form $x_{ij}$, where $x_{ij}$ represents the fact that node $j$ is the $i^{th}$ node in the Hamilton Path (may or may not be true).

Step 2: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. [$C_1$].

Step 3: $(\neg x_{ij} \lor \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. [$C_2$].

Step 4: $(x_{i1} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2 \ldots n$. [$C_3$].

Step 5: $(\neg x_{ij} \lor \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. [$C_4$].

Step 6: $(\neg x_{ki} \lor \neg x_{(k+1)i}, k = 1, 2, \ldots, n - 1, (i, j) \notin G$. [$C_5$].

Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

Argument: Let $\tilde{x}$ denote a satisfying assignment to $\phi$. We show that there must exist a Hamilton Path in $G$.

Let $\pi = (\pi(1), \pi(2) \ldots \pi(n))$ denote a Hamilton path, where $\pi$ is a permutation. We show that $\phi$ is satisfiable.
Sample Reductions

Hamilton Path to SAT

Input instance: An unweighted, directed graph $G$.
Output instance: A CNF formula $\phi$, such that $G$ has a Hamilton path if and only if $\phi$ is satisfiable.
Step 1: Suppose $G$ has $n$ nodes; $\phi$ has $n^2$ variables of the form $x_{ij}$, where $x_{ij}$ represents the fact that node $j$ is the $i^{th}$ node in the Hamilton Path (may or may not be true).
Step 2: $(x_{1j} \lor x_{2j} \ldots x_{nj})$, $j = 1, 2, \ldots, n$. [$C_1$].
Step 3: $(\neg x_{ij} \lor \neg x_{kj})$, $j = 1, 2 \ldots n$, $i = 1, 2, \ldots, n$, $k = 1, 2, \ldots n$, $k \neq i$. [$C_2$].
Step 4: $(x_{i1} \lor x_{i2} \ldots \lor x_{in})$, $i = 1, 2 \ldots n$. [$C_3$].
Step 5: $(\neg x_{ij} \lor \neg x_{jk})$, $i = 1, 2, \ldots, n$, $j, k = 1, 2, \ldots, n$, $j \neq k$. [$C_4$].
Step 6: $(\neg x_{ki} \lor \neg x_{(k+1)i})$, $k = 1, 2, \ldots, n - 1$, $(i, j) \notin G$. [$C_5$].
Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

Argument: Let $\tilde{x}$ denote a satisfying assignment to $\phi$. We show that there must exist a Hamilton Path in $G$.

Let $\pi = (\pi(1), \pi(2) \ldots \pi(n))$ denote a Hamilton path, where $\pi$ is a permutation. We show that $\phi$ is satisfiable.
Sample Reductions

### Hamilton Path to SAT

**Input instance:** An unweighted, directed graph $G$.

**Output instance:** A CNF formula $\phi$, such that $G$ has a Hamilton path if and only if $\phi$ is satisfiable.

**Step 1:** Suppose $G$ has $n$ nodes; $\phi$ has $n^2$ variables of the form $x_{ij}$, where $x_{ij}$ represents the fact that node $j$ is the $i^{th}$ node in the Hamilton Path (may or may not be true).

**Step 2:** $(x_{i1} \lor x_{i2} \ldots x_{ijn})$, $j = 1, 2, \ldots, n$. $[C_1]$.

**Step 3:** $(\neg x_{ij} \lor \neg x_{kj})$, $j = 1, 2 \ldots n$, $i = 1, 2, \ldots, n$, $k = 1, 2, \ldots n$, $k \neq i$. $[C_2]$.

**Step 4:** $(x_{i1} \lor x_{i2} \ldots \lor x_{in})$, $i = 1, 2 \ldots n$. $[C_3]$.

**Step 5:** $(\neg x_{ij} \lor \neg x_{jk})$, $i = 1, 2, \ldots, n$, $j, k = 1, 2, \ldots, n$, $j \neq k$. $[C_4]$.

**Step 6:** $(\neg x_{ki} \lor \neg x_{(k+1)i})$, $k = 1, 2, \ldots, n - 1$, $(i, j) \not\in G$. $[C_5]$.

**Step 7:** $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

**Argument:** Let $\bar{x}$ denote a satisfying assignment to $\phi$. We show that there must exist a Hamilton Path in $G$.

Let $\pi = (\pi(1), \pi(2) \ldots \pi(n))$ denote a Hamilton path, where $\pi$ is a permutation. We show that $\phi$ is satisfiable.
Sample Reductions

Hamilton Path to SAT

Input instance: An unweighted, directed graph $G$.
Output instance: A CNF formula $\phi$, such that $G$ has a Hamilton path if and only if $\phi$ is satisfiable.

Step 1: Suppose $G$ has $n$ nodes; $\phi$ has $n^2$ variables of the form $x_{ij}$, where $x_{ij}$ represents the fact that node $j$ is the $i^{th}$ node in the Hamilton Path (may or may not be true).
Step 2: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots , n$. [C$_1$].
Step 3: $(-x_{ij} \lor -x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots , n, k = 1, 2, \ldots n, k \neq i$. [C$_2$].
Step 4: $(x_{i1} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2 \ldots n$. [C$_3$].
Step 5: $(-x_{ij} \lor -x_{jk}), i = 1, 2, \ldots , n, j, k = 1, 2, \ldots , n, j \neq k$. [C$_4$].
Step 6: $(-x_{ki} \lor -x_{(k+1)i}, k = 1, 2, \ldots , n - 1, (i, j) \not\in G$. [C$_5$].
Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

Argument: Let $\bar{x}$ denote a satisfying assignment to $\phi$. We show that there must exist a Hamilton Path in $G$.

Let $\pi = (\pi(1), \pi(2) \ldots \pi(n))$ denote a Hamilton path, where $\pi$ is a permutation. We show that $\phi$ is satisfiable.
Sample Reductions

Hamilton Path to SAT

Input instance: An unweighted, directed graph $G$.
Output instance: A CNF formula $\phi$, such that $G$ has a Hamilton path if and only if $\phi$ is satisfiable.
Step 1: Suppose $G$ has $n$ nodes; $\phi$ has $n^2$ variables of the form $x_{ij}$, where $x_{ij}$ represents the fact that node $j$ is the $i^{th}$ node in the Hamilton Path (may or may not be true).
Step 2: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. [$C_1$].
Step 3: $(\neg x_{ij} \lor \neg x_{kj}), j = 1, 2 \ldots n$, $i = 1, 2, \ldots, n$, $k = 1, 2, \ldots n$, $k \neq i$. [$C_2$].
Step 4: $(x_{i1} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2 \ldots n$. [$C_3$].
Step 5: $(\neg x_{ij} \lor \neg x_{jk}), i = 1, 2, \ldots, n$, $j, k = 1, 2, \ldots, n$, $j \neq k$. [$C_4$].
Step 6: $(\neg x_{ki} \lor \neg x_{(k+1)i}, k = 1, 2, \ldots, n-1, (i, j) \not\in G$. [$C_5$].
Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

Argument: Let $\bar{x}$ denote a satisfying assignment to $\phi$. We show that there must exist a Hamilton Path in $G$.
Let $\pi = (\pi(1), \pi(2) \ldots \pi(n))$ denote a Hamilton path, where $\pi$ is a permutation. We show that $\phi$ is satisfiable.
Sample Reductions (contd.)

Circuit SAT to SAT

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.

Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.

Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg x)$ and $(\neg g \lor x)$ to $\phi$.

Step 3: If $g$ is a true gate, add $(g)$ to $\phi$; likewise, if it is a false gate, add $(\neg g)$.

Step 4: If $g$ is a NOT gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.

Step 5: If $g$ is an OR gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.

Step 6: If $g$ is an AND gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.

Step 7: If $g$ is an output gate, add the clause $(g)$.

Argument: If $C$ is satisfiable, then $\phi$ is satisfiable.

If $\phi$ is satisfiable, then $C$ is satisfiable.
**CIRCUIT SAT to SAT**

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.

Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.

Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg x)$ and $(\neg g \lor x)$ to $\phi$.

Step 3: If $g$ is a true gate, add $(g)$ to $\phi$; likewise, if it is a false gate, add $(\neg g)$.

Step 4: If $g$ is a NOT gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.

Step 5: If $g$ is an OR gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.

Step 6: If $g$ is an AND gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.

Step 7: If $g$ is an output gate, add the clause $(g)$.

Argument: If $C$ is satisfiable, then $\phi$ is satisfiable.

If $\phi$ is satisfiable, then $C$ is satisfiable.
Sample Reductions (contd.)

**CIRCUIT SAT to SAT**

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.

Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.

Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg x)$ and $(\neg g \lor x)$ to $\phi$.

Step 3: If $g$ is a true gate, add $(g)$ to $\phi$; likewise, if it is a false gate, add $(\neg g)$.

Step 4: If $g$ is a NOT gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.

Step 5: If $g$ is an OR gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.

Step 6: If $g$ is an AND gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.

Step 7: If $g$ is an output gate, add the clause $(g)$.

Argument: If $C$ is satisfiable, then $\phi$ is satisfiable.

If $\phi$ is satisfiable, then $C$ is satisfiable.
Circuit SAT to SAT

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.

Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.

Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg x)$ and $(\neg g \lor x)$ to $\phi$.

Step 3: If $g$ is a true gate, add $(g)$ to $\phi$; likewise, if it is a false gate, add $(\neg g)$.

Step 4: If $g$ is a NOT gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.

Step 5: If $g$ is an OR gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.

Step 6: If $g$ is an AND gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.

Step 7: If $g$ is an output gate, add the clause $(g)$.

Argument: If $C$ is satisfiable, then $\phi$ is satisfiable.
If $\phi$ is satisfiable, then $C$ is satisfiable.
Sample Reductions (contd.)

**CIRCUIT SAT to SAT**

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.

Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.

Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg)$ and $(\neg g \lor x)$ to $\phi$.

Step 3: If $g$ is a true gate, add $(g)$ to $\phi$; likewise, if it is a false gate, add $(\neg g)$.

Step 4: If $g$ is a NOT gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.

Step 5: If $g$ is an OR gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.

Step 6: If $g$ is an AND gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.

Step 7: If $g$ is an output gate, add the clause $(g)$.

Argument: If $C$ is satisfiable, then $\phi$ is satisfiable.
If $\phi$ is satisfiable, then $C$ is satisfiable.
Circuit SAT to SAT

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.

Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.
Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg x)$ and $(\neg g \lor x)$ to $\phi$.
Step 3: If $g$ is a true gate, add $(g)$ to $\phi$; likewise, if it is a false gate, add $(\neg g)$.
Step 4: If $g$ is a NOT gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.
Step 5: If $g$ is an OR gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.
Step 6: If $g$ is an AND gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.
Step 7: If $g$ is an output gate, add the clause $(g)$.

Argument: If $C$ is satisfiable, then $\phi$ is satisfiable.
If $\phi$ is satisfiable, then $C$ is satisfiable.
**Circuit SAT to SAT**

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.

Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.

Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg x)$ and $(\neg g \lor x)$ to $\phi$.

Step 3: If $g$ is a true gate, add $(g)$ to $\phi$; likewise, if it is a false gate, add $(\neg g)$.

Step 4: If $g$ is a NOT gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.

Step 5: If $g$ is an OR gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.

Step 6: If $g$ is an AND gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.

Step 7: If $g$ is an output gate, add the clause $(g)$.

Argument: If $C$ is satisfiable, then $\phi$ is satisfiable.

If $\phi$ is satisfiable, then $C$ is satisfiable.
CIRCUIT SAT to SAT

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.
Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.
Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg x)$ and $(\neg g \lor x)$ to $\phi$.
Step 3: If $g$ is a true gate, add $(g)$ to $\phi$; likewise, if it is a false gate, add $(\neg g)$.
Step 4: If $g$ is a NOT gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.
Step 5: If $g$ is an OR gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.
Step 6: If $g$ is an AND gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.
Step 7: If $g$ is an output gate, add the clause $(g)$.
Argument: If $C$ is satisfiable, then $\phi$ is satisfiable.
If $\phi$ is satisfiable, then $C$ is satisfiable.
CIRCUIT SAT to SAT

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.
Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.
Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg x)$ and $(\neg g \lor x)$ to $\phi$.
Step 3: If $g$ is a true gate, add $(g)$ to $\phi$; likewise, if it is a false gate, add $(\neg g)$.
Step 4: If $g$ is a NOT gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.
Step 5: If $g$ is an OR gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.
Step 6: If $g$ is an AND gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.
Step 7: If $g$ is an output gate, add the clause $(g)$.
Argument: If $C$ is satisfiable, then $\phi$ is satisfiable.
If $\phi$ is satisfiable, then $C$ is satisfiable.
**CIRCUIT SAT to SAT**

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.

Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.

Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg)$ and $(\neg g \lor x)$ to $\phi$.

Step 3: If $g$ is a **true** gate, add $(g)$ to $\phi$; likewise, if it is a **false** gate, add $(\neg g)$.

Step 4: If $g$ is a **NOT** gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.

Step 5: If $g$ is an **OR** gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.

Step 6: If $g$ is an **AND** gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.

Step 7: If $g$ is an output gate, add the clause $(g)$.

Argument: If $C$ is satisfiable, then $\phi$ is satisfiable. If $\phi$ is satisfiable, then $C$ is satisfiable.
CIRCUIT SAT to SAT

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.

Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.

Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg x)$ and $(\neg g \lor x)$ to $\phi$.

Step 3: If $g$ is a true gate, add $(g)$ to $\phi$; likewise, if it is a false gate, add $(\neg g)$.

Step 4: If $g$ is a NOT gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.

Step 5: If $g$ is an OR gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.

Step 6: If $g$ is an AND gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.

Step 7: If $g$ is an output gate, add the clause $(g)$.

Argument: If $C$ is satisfiable, then $\phi$ is satisfiable.

If $\phi$ is satisfiable, then $C$ is satisfiable.
**Circuit SAT to SAT**

Input instance: A circuit $C$.
Output instance: A CNF formula $\phi$ such that $\phi$ is satisfiable if and only if $C$ is.

Step 1: The variables of $\phi$ will contain all the variables of $C$. Additionally, for each gate $g$ in $C$, we create a new variable in $\phi$, also denoted by $g$.

Step 2: If $g$ is a variable gate, corresponding to variable $x$, add the clauses $(g \lor \neg)$ and $(\neg g \lor x)$ to $\phi$.

Step 3: If $g$ is a **true** gate, add $(g)$ to $\phi$; likewise, if it is a **false** gate, add $(\neg g)$.

Step 4: If $g$ is a $NOT$ gate with predecessor $h$, add the clauses $(g \lor h)$ and $(\neg g \lor \neg h)$ to $\phi$.

Step 5: If $g$ is an $OR$ gate with predecessors $h$ and $h'$, add the clauses $(\neg h \lor g)$, $(\neg h' \lor g)$ and $(h \lor h' \lor \neg g)$ to $\phi$.

Step 6: If $g$ is an $AND$ gate with predecessors $h$ and $h'$, add the clauses $(\neg g \lor h)$, $(\neg g \lor h')$ and $(\neg h \lor \neg h' \lor g)$ to $\phi$.

Step 7: If $g$ is an output gate, add the clause $(g)$.

Argument: If $C$ is satisfiable, then $\phi$ is satisfiable.

If $\phi$ is satisfiable, then $C$ is satisfiable.
CIRCUIT SAT to SAT

Input instance: A circuit \( C \).
Output instance: A CNF formula \( \phi \) such that \( \phi \) is satisfiable if and only if \( C \) is.
Step 1: The variables of \( \phi \) will contain all the variables of \( C \). Additionally, for each gate \( g \) in \( C \), we create a new variable in \( \phi \), also denoted by \( g \).
Step 2: If \( g \) is a variable gate, corresponding to variable \( x \), add the clauses \((g \lor \neg \ )\) and \((\neg g \lor x)\) to \( \phi \).
Step 3: If \( g \) is a true gate, add \((g)\) to \( \phi \); likewise, if it is a false gate, add \((\neg g)\).
Step 4: If \( g \) is a NOT gate with predecessor \( h \), add the clauses \((g \lor h)\) and \((\neg g \lor \neg h)\) to \( \phi \).
Step 5: If \( g \) is an OR gate with predecessors \( h \) and \( h' \), add the clauses \((\neg h \lor g)\), \((\neg h' \lor g)\) and \((h \lor h' \lor \neg g)\) to \( \phi \).
Step 6: If \( g \) is an AND gate with predecessors \( h \) and \( h' \), add the clauses \((\neg g \lor h)\), \((\neg g \lor h')\) and \((\neg h \lor \neg h' \lor g)\) to \( \phi \).
Step 7: If \( g \) is an output gate, add the clause \((g)\).
Argument: If \( C \) is satisfiable, then \( \phi \) is satisfiable.
If \( \phi \) is satisfiable, then \( C \) is satisfiable.
Reduction by generalization

CIRCUIT VALUE to CIRCUIT SAT. $R$ is the identity function!
Sample Reductions (contd.)

**Reduction by generalization**

CIRCUIT VALUE to CIRCUIT SAT. \( R \) is the identity function!
Composition of Reductions

**Theorem**

If $R$ is a reduction from $L_1$ to $L_2$ and $R'$ is a reduction from $L_2$ to $L_3$, then $R' \circ R$ is a reduction from $L_1$ to $L_3$.

**Proof.**

Trivial for poly-time reductions. Not so obvious for log-space reductions, since output of $R(x)$ could be larger than $\log |x|$.

Main idea: Dovetail simulations.
Composition of Reductions

Theorem

If $R$ is a reduction from $L_1$ to $L_2$ and $R'$ is a reduction from $L_2$ to $L_3$, then $R' \circ R$ is a reduction from $L_1$ to $L_3$.

Proof.

Trivial for poly-time reductions. Not so obvious for log-space reductions, since output of $R(x)$ could be larger than $\log |x|$.

Main idea: Dovetail simulations.
Composition of Reductions

Theorem

If $R$ is a reduction from $L_1$ to $L_2$ and $R'$ is a reduction from $L_2$ to $L_3$, then $R' \circ R$ is a reduction from $L_1$ to $L_3$.

Proof.

Trivial for poly-time reductions. Not so obvious for log-space reductions, since output of $R(x)$ could be larger than $\log |x|$.

Main idea: Dovetail simulations.
Theorem

If \( R \) is a reduction from \( L_1 \) to \( L_2 \) and \( R' \) is a reduction from \( L_2 \) to \( L_3 \), then \( R' \circ R \) is a reduction from \( L_1 \) to \( L_3 \).

Proof.

Trivial for poly-time reductions. Not so obvious for log-space reductions, since output of \( R(x) \) could be larger than \( \log |x| \).

Main idea: Dovetail simulations.
Composition of Reductions

Theorem

If $R$ is a reduction from $L_1$ to $L_2$ and $R'$ is a reduction from $L_2$ to $L_3$, then $R' \circ R$ is a reduction from $L_1$ to $L_3$.

Proof.

Trivial for poly-time reductions. Not so obvious for log-space reductions, since output of $R(x)$ could be larger than $\log |x|$.

Main idea: Dovetail simulations.
Completeness

**Definition**
A language $L$ in a complexity class $C$ is said to be $C$-complete, if any language $L' \in C$ can be reduced to $L$.

**Definition**
A complexity class $C$ is closed under reductions, if

$$((L \in C) \land (L' \leq L)) \rightarrow (L' \in C).$$

**Proposition**
P, NP, coNP, L, NL, PSPACE and EXP are all closed under reductions.

**Corollary**
If two classes $C$ and $C'$ are both closed under reductions and there exists a language $L$ that is complete for both $C$ and $C'$, then $C = C'$. 
Completeness

Definition

A language $L$ in a complexity class $C$ is said to be $C$-complete, if any language $L' \in C$ can be reduced to $L$.

Definition

A complexity class $C$ is closed under reductions, if

$$((L \in C) \land (L' \leq L)) \rightarrow (L' \in C).$$

Proposition

P, NP, coNP, L, NL, PSPACE and EXP are all closed under reductions.

Corollary

If two classes $C$ and $C'$ are both closed under reductions and there exists a language $L$ that is complete for both $C$ and $C'$, then $C = C'$. 
Completeness

**Definition**

A language \( L \) in a complexity class \( C \) is said to be \( C \)-complete, if any language \( L' \in C \) can be reduced to \( L \).

**Definition**

A complexity class \( C \) is closed under reductions, if

\[
((L \in C) \land (L' \leq L)) \rightarrow (L' \in C).
\]

**Proposition**

\( P, NP, \text{coNP}, L, NL, \text{PSPACE} \) and \( EXP \) are all closed under reductions.

**Corollary**

If two classes \( C \) and \( C' \) are both closed under reductions and there exists a language \( L \) that is complete for both \( C \) and \( C' \), then \( C = C' \).
**Completeness**

**Definition**

A language $L$ in a complexity class $C$ is said to be $C$-complete, if any language $L' \in C$ can be reduced to $L$.

**Definition**

A complexity class $C$ is closed under reductions, if

$$(L \in C) \land (L' \leq L) \rightarrow (L' \in C).$$

**Proposition**

$P$, $NP$, $coNP$, $L$, $NL$, $PSPACE$ and $EXP$ are all closed under reductions.

**Corollary**

If two classes $C$ and $C'$ are both closed under reductions and there exists a language $L$ that is complete for both $C$ and $C'$, then $C = C'$. 

**Subramani Complexity Classes**
Completeness

Definition

A language $L$ in a complexity class $C$ is said to be $C$-complete, if any language $L' \in C$ can be reduced to $L$.

Definition

A complexity class $C$ is closed under reductions, if

$$((L \in C) \land (L' \leq L)) \rightarrow (L' \in C).$$

Proposition

$P$, $NP$, $coNP$, $L$, $NL$, $PSPACE$ and $EXP$ are all closed under reductions.

Corollary

If two classes $C$ and $C'$ are both closed under reductions and there exists a language $L$ that is complete for both $C$ and $C'$ then $C = C'$. 
Completeness

Definition
A language $L$ in a complexity class $C$ is said to be $C$-complete, if any language $L' \in C$ can be reduced to $L$.

Definition
A complexity class $C$ is closed under reductions, if

$((L \in C) \land (L' \leq L)) \rightarrow (L' \in C)$.

Proposition
$P, NP, coNP, L, NL, PSPACE$ and $EXP$ are all closed under reductions.

Corollary
If two classes $C$ and $C'$ are both closed under reductions and there exists a language $L$ that is complete for both $C$ and $C'$, then $C = C'$. 
Completeness

**Definition**

A language $L$ in a complexity class $C$ is said to be $C$-complete, if any language $L' \in C$ can be reduced to $L$.

**Definition**

A complexity class $C$ is closed under reductions, if

$$((L \in C) \land (L' \leq L)) \rightarrow (L' \in C).$$

**Proposition**

$P$, $NP$, $coNP$, $L$, $NL$, $PSPACE$ and $EXP$ are all closed under reductions.

**Corollary**

*If two classes $C$ and $C'$ are both closed under reductions and there exists a language $L$ that is complete for both $C$ and $C'$ then $C = C'$.***
Outline

1. Reductions

2. Completeness
   - P-completeness
   - NP-completeness
**P-completeness of Circuit Value**

**Theorem**

*Circuit Value* is **P-complete**.

**Proof.**

Let $L$ be some language in $P$.

$\Rightarrow$ There exists a Turing machine $M = (K, \Sigma, \delta, s)$, which halts on any string in $x \in \Sigma^*$ in time at most $|x|^k$, for a fixed constant $k$.

$\Rightarrow$ There exists a computation table $T$ for $M(x)$ of dimensions $|x|^k \times |x|^k$, where $T_{ij}$ represents the contents of position $j$ at time $i$ (after $i$ steps have been completed).

We assume that the machine is standardized as follows:

(i) It has only one string.

(ii) It halts within $|x|^k - 2$ steps.

(iii) The computation pads the string with a sufficient number of $\sqcup$s, so that the length of the string is exactly $|x|^k$.

(iv) The tape alphabet ($\Gamma$) is standardized to include symbols for (state, symbol) pairs. For instance $0_s$ represents the fact that we are currently in state $s$ scanning symbol 0.

(v) States “yes” and “no” are recorded as is.

(vi) Computation is accepting if $T_{|x|^k-1,j} = \text{“yes”}$ for $j = 2$. 
P-completeness of \textsc{Circuit Value}

**Theorem**

\textsc{Circuit Value} is \textbf{P-complete}.

**Proof.**

Let $L$ be some language in $\textbf{P}$.

\Rightarrow \text{There exists a Turing machine } M = (K, \Sigma, \delta, s), \text{ which halts on any string in } x \in \Sigma^* \text{ in time at most } |x|^k, \text{ for a fixed constant } k.

\Rightarrow \text{There exists a computation table } T \text{ for } M(x) \text{ of dimensions } |x|^k \times |x|^k, \text{ where } T_{ij} \text{ represents the contents of position } j \text{ at time } i \text{ (after } i \text{ steps have been completed).}

We assume that the machine is standardized as follows:

(i) It has only one string.

(ii) It halts within $|x|^k - 2$ steps.

(iii) The computation pads the string with a sufficient number of $\sqcup$s, so that the length of the string is exactly $|x|^k$.

(iv) The tape alphabet ($\Gamma$) is standardized to include symbols for (state, symbol) pairs. For instance $0_s$ represents the fact that we are currently in state $s$ scanning symbol 0.

(v) States “yes” and “no” are recorded as is.

(vi) Computation is accepting if $T_{|x|^k - 1, j} = \text{“yes”}$ for $j = 2$. 

\[ \square \]
**Theorem**

*Circuit Value* is *P*-complete.

**Proof.**

Let $L$ be some language in $P$.

⇒ There exists a Turing machine $M = (K, \Sigma, \delta, s)$, which halts on any string in $x \in \Sigma^*$ in time at most $|x|^k$, for a fixed constant $k$.

⇒ There exists a computation table $T$ for $M(x)$ of dimensions $|x|^k \times |x|^k$, where $T_{ij}$ represents the contents of position $j$ at time $i$ (after $i$ steps have been completed).

We assume that the machine is standardized as follows:

(i) It has only one string.

(ii) It halts within $|x|^k - 2$ steps.

(iii) The computation pads the string with a sufficient number of $\sqcup$s, so that the length of the string is exactly $|x|^k$.

(iv) The tape alphabet ($\Gamma$) is standardized to include symbols for (state, symbol) pairs. For instance $0_s$ represents the fact that we are currently in state $s$ scanning symbol 0.

(v) States “yes” and “no” are recorded as is.

(vi) Computation is accepting if $T_{|x|^k-1, j} = \text{“yes”}$ for $j = 2$. 

\[\square\]
Theorem

CIRCUIT VALUE is P-complete.

Proof.

Let $L$ be some language in P.

$\Rightarrow$ There exists a Turing machine $M = (K, \Sigma, \delta, s)$, which halts on any string in $x \in \Sigma^*$ in time at most $|x|^k$, for a fixed constant $k$.

$\Rightarrow$ There exists a computation table $T$ for $M(x)$ of dimensions $|x|^k \times |x|^k$, where $T_{ij}$ represents the contents of position $j$ at time $i$ (after $i$ steps have been completed).

We assume that the machine is standardized as follows:

(i) It has only one string.

(ii) It halts within $|x|^k - 2$ steps.

(iii) The computation pads the string with a sufficient number of $\sqcup$’s, so that the length of the string is exactly $|x|^k$.

(iv) The tape alphabet ($\Gamma$) is standardized to include symbols for (state, symbol) pairs. For instance $0_s$ represents the fact that we are currently in state $s$ scanning symbol 0.

(v) States “yes” and “no” are recorded as is.

(vi) Computation is accepting if $T_{|x|^k-1,j} = \text{“yes”}$ for $j = 2$. 

Subramani Complexity Classes
**Theorem**

**CIRCUIT VALUE is P-complete.**

**Proof.**

Let $L$ be some language in $P$.

1. There exists a Turing machine $M = (K, \Sigma, \delta, s)$, which halts on any string in $x \in \Sigma^*$ in time at most $|x|^k$, for a fixed constant $k$.

2. There exists a computation table $T$ for $M(x)$ of dimensions $|x|^k \times |x|^k$, where $T_{ij}$ represents the contents of position $j$ at time $i$ (after $i$ steps have been completed).

We assume that the machine is standardized as follows:

(i) It has only one string.

(ii) It halts within $|x|^k - 2$ steps.

(iii) The computation pads the string with a sufficient number of $\sqcup$s, so that the length of the string is exactly $|x|^k$.

(iv) The tape alphabet ($\Gamma$) is standardized to include symbols for (state, symbol) pairs. For instance $0_s$ represents the fact that we are currently in state $s$ scanning symbol 0.

(v) States “yes” and “no” are recorded as is.

(vi) Computation is accepting if $T_{|x|^k - 1, j} = “yes”$ for $j = 2$. 

$\Box$
Proof.

When \( i = 0 \) or \( j = 0 \) or \( j = |x|^k \), the contents of \( T_{ij} \) are known apriori.

Crucial observation: \( T_{ij} \) depends only on the entries \( T_{i-1,j-1}, T_{i-1,j} \) and \( T_{i-1,j+1} \). Why?

Encode each tape symbol as a binary vector \( s = (s_1, s_2, \ldots, s_m) \), where \( m = \lceil \log |\Gamma| \rceil \). The encoding of “yes” begins with 1 and the encoding of “no” begins with 0.

The computation table is now a table of binary entries \( S_{ijl}, 0 \leq i \leq |x|^k - 1, 0 \leq j \leq |x|^k - 1 \text{ and } 1 \leq l \leq m \).

Each binary entry \( S_{ij} \) depends only on the \( 3m \) entries \( S_{i-1,j-1,l'}, S_{i-1,j+1,l'}, S_{i-1,j,l'} \), where \( l' \) ranges over \( 1, 2, \ldots m \).

But these are boolean functions and hence can be captured through gates.

Create \((|x|^k - 1) \times (|x|^k - 2)\) gates, one for each entry \( T_{ij} \).

The reduction can be accomplished in \( \log |x| \) space.
Proof.

When \( i = 0 \) or \( j = 0 \) or \( j = |x|^k \), the contents of \( T_{ij} \) are known apriori.

Crucial observation: \( T_{ij} \) depends only on the entries \( T_{i-1,j-1}, T_{i-1,j} \) and \( T_{i-1,j+1} \). Why?

Encode each tape symbol as a binary vector \( s = (s_1, s_2, \ldots, s_m) \), where \( m = \lceil \log |\Gamma| \rceil \). The encoding of “yes” begins with 1 and the encoding of “no” begins with 0.

The computation table is now a table of binary entries \( S_{ij} \), \( 0 \leq i \leq |x|^k - 1 \), \( 0 \leq j \leq |x|^k - 1 \) and \( 1 \leq l \leq m \).

Each binary entry \( S_{ij} \) depends only on the \( 3m \) entries \( S_{i-1,j-1,l'}, S_{i-1,j,l'}, S_{i-1,j+1,l'} \), where \( l' \) ranges over \( 1, 2, \ldots, m \).

But these are boolean functions and hence can be captured through gates.

Create \( (|x|^k - 1) \times (|x|^k - 2) \) gates, one for each entry \( T_{ij} \).

The reduction can be accomplished in \( \log |x| \) space.
Proof.

When \( i = 0 \) or \( j = 0 \) or \( j = |x|^k \), the contents of \( T_{ij} \) are known apriori.

Crucial observation: \( T_{ij} \) depends only on the entries \( T_{i-1,j-1}, T_{i-1,j} \) and \( T_{i-1,j+1} \). Why?

Encode each tape symbol as a binary vector \( s = (s_1, s_2, \ldots, s_m) \), where \( m = \lceil \log |\Gamma| \rceil \). The encoding of “yes” begins with 1 and the encoding of “no” begins with 0.

The computation table is now a table of binary entries \( S_{ijl}, 0 \leq i \leq |x|^k - 1, 0 \leq j \leq |x|^k - 1 \) and \( 1 \leq l \leq m \).

Each binary entry \( S_{ij} \) depends only on the \( 3m \) entries \( S_{i-1,j-1,l'}, S_{i-1,j,l'}, S_{i-1,j+1,l'} \), where \( l' \) ranges over \( 1, 2, \ldots m \).

But these are boolean functions and hence can be captured through gates.

Create \( (|x|^k - 1) \times (|x|^k - 2) \) gates, one for each entry \( T_{ij} \).

The reduction can be accomplished in \( \log |x| \) space.
Proof.

When $i = 0$ or $j = 0$ or $j = |x|^k$, the contents of $T_{ij}$ are known apriori.

Crucial observation: $T_{ij}$ depends only on the entries $T_{i-1,j-1}$, $T_{i-1,j}$ and $T_{i-1,j+1}$. Why?

Encode each tape symbol as a binary vector $s = (s_1, s_2, \ldots, s_m)$, where $m = \lceil \log |\Gamma| \rceil$. The encoding of “yes” begins with 1 and the encoding of “no” begins with 0.

The computation table is now a table of binary entries $S_{ijl}$, $0 \leq i \leq |x|^k - 1$, $0 \leq j \leq |x|^k - 1$ and $1 \leq l \leq m$.

Each binary entry $S_{ij}$ depends only on the $3m$ entries $S_{i-1,j-1,l'}$, $S_{i-1,j,l'}$, $S_{i-1,j+1,l'}$, where $l'$ ranges over $1, 2, \ldots m$.

But these are boolean functions and hence can be captured through gates.

Create $(|x|^k - 1) \times (|x|^k - 2)$ gates, one for each entry $T_{ij}$.

The reduction can be accomplished in $\log |x|$ space.
Proof.

When \( i = 0 \) or \( j = 0 \) or \( j = |x|^k \), the contents of \( T_{ij} \) are known apriori.

Crucial observation: \( T_{ij} \) depends only on the entries \( T_{i-1,j-1} \), \( T_{i-1,j} \) and \( T_{i-1,j+1} \). Why?

Encode each tape symbol as a binary vector \( s = (s_1, s_2, \ldots, s_m) \), where \( m = \lceil \log |\Gamma| \rceil \). The encoding of “yes” begins with 1 and the encoding of “no” begins with 0.

The computation table is now a table of binary entries \( S_{ij} \), \( 0 \leq i \leq |x|^k - 1 \), \( 0 \leq j \leq |x|^k - 1 \) and \( 1 \leq l \leq m \).

Each binary entry \( S_{ij} \) depends only on the \( 3m \) entries \( S_{i-1,j-1,l'} \), \( S_{i-1,j,l'} \), \( S_{i-1,j+1,l'} \), where \( l' \) ranges over \( 1, 2, \ldots m \).

But these are boolean functions and hence can be captured through gates.

Create \((|x|^k - 1) \times (|x|^k - 2)\) gates, one for each entry \( T_{ij} \).

The reduction can be accomplished in \( \log |x| \) space.
Proof.

When \( i = 0 \) or \( j = 0 \) or \( j = |x|^k \), the contents of \( T_{ij} \) are known apriori.

Crucial observation: \( T_{ij} \) depends only on the entries \( T_{i-1,j-1} \), \( T_{i-1,j} \) and \( T_{i-1,j+1} \). Why?

Encode each tape symbol as a binary vector \( s = (s_1, s_2, \ldots, s_m) \), where \( m = \lceil \log |\Gamma| \rceil \). The encoding of “yes” begins with 1 and the encoding of “no” begins with 0.

The computation table is now a table of binary entries \( S_{ijl}, 0 \leq i \leq |x|^k - 1, 0 \leq j \leq |x|^k - 1 \) and \( 1 \leq l \leq m \).

Each binary entry \( S_{ij} \) depends only on the \( 3m \) entries \( S_{i-1,j-1,l'}, S_{i-1,j,l'}, S_{i-1,j+1,l'} \), where \( l' \) ranges over \( 1, 2, \ldots, m \).

But these are boolean functions and hence can be captured through gates.

Create \( (|x|^k - 1) \times (|x|^k - 2) \) gates, one for each entry \( T_{ij} \).

The reduction can be accomplished in \( \log |x| \) space.
Proof.

When $i = 0$ or $j = 0$ or $j = |x|^k$, the contents of $T_{ij}$ are known apriori.

Crucial observation: $T_{ij}$ depends only on the entries $T_{i-1,j-1}$, $T_{i-1,j}$ and $T_{i-1,j+1}$. Why?

Encode each tape symbol as a binary vector $s = (s_1, s_2, \ldots, s_m)$, where $m = \lceil \log |\Gamma| \rceil$. The encoding of “yes” begins with 1 and the encoding of “no” begins with 0.

The computation table is now a table of binary entries $S_{ijl}$, $0 \leq i \leq |x|^k - 1$, $0 \leq j \leq |x|^k - 1$ and $1 \leq l \leq m$.

Each binary entry $S_{ij}$ depends only on the $3m$ entries $S_{i-1,j-1,l'}$, $S_{i-1,j,l'}$, $S_{i-1,j+1,l'}$, where $l'$ ranges over $1, 2, \ldots, m$.

But these are boolean functions and hence can be captured through gates.

Create $(|x|^k - 1) \times (|x|^k - 2)$ gates, one for each entry $T_{ij}$.

The reduction can be accomplished in $\log |x|$ space.
Outline

1. Reductions
2. Completeness
   - P-completeness
   - NP-completeness
Theorem (Cook)

SAT is NP-complete.

Proof.

We will show that CIRCUIT SAT is NP-complete. Cook’s theorem follows. Let $L \in \text{NP}$; this means that $L$ is decided by a NDTM $M = (K, \Sigma, \delta, s)$, which halts with a “yes” or “no” on all strings $x \in \Sigma^*$ in at most $|x|^k$ time. Standardize the Turing Machine so that degree of non-determinism is exactly 2. It follows that a sequence of non-deterministic choices is a bit-string $(c_0, c_1, \ldots, c_{|x|^k-1})$.

Use same reduction as CIRCUIT VALUE; the only difference is that $c_i$ is now a variable at row $i$ of the table!
Theorem (Cook)

**SAT is NP-complete.**

Proof.

We will show that CIRCUIT SAT is NP-complete. Cook's theorem follows.

Let \( L \in \text{NP} \); this means that \( L \) is decided by a NDTM \( M = (K, \Sigma, \delta, s) \), which halts with a “yes” or “no” on all strings \( x \in \Sigma^* \) in at most \(|x|^k\) time.

Standardize the Turing Machine so that degree of non-determinism is exactly 2. It follows that a sequence of non-deterministic choices is a bit-string \((c_0, c_1, \ldots, c_{|x|^k - 1})\).

Use same reduction as CIRCUIT VALUE; the only difference is that \( c_i \) is now a variable at row \( i \) of the table!
Theorem (Cook)

SAT is NP-complete.

Proof.

We will show that CIRCUIT SAT is NP-complete. Cook's theorem follows.
Let $L \in \textbf{NP}$; this means that $L$ is decided by a NDTM $M = (K, \Sigma, \delta, s)$, which halts with a “yes” or “no” on all strings $x \in \Sigma^*$ in at most $|x|^k$ time.
Standardize the Turing Machine so that degree of non-determinism is exactly 2. It follows that a sequence of non-deterministic choices is a bit-string $(c_0, c_1, \ldots, c_{|x|^k-1})$.
Use same reduction as CIRCUIT VALUE; the only difference is that $c_i$ is now a variable at row $i$ of the table!
Theorem (Cook)

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

Proof.

We will show that **CIRCUIT SAT** is **NP-complete**. Cook's theorem follows.

Let $L \in \text{NP}$; this means that $L$ is decided by a NDTM $M = (K, \Sigma, \delta, s)$, which halts with a “yes” or “no” on all strings $x \in \Sigma^*$ in at most $|x|^k$ time.

Standardize the Turing Machine so that degree of non-determinism is exactly 2. It follows that a sequence of non-deterministic choices is a bit-string $(c_0, c_1, \ldots, c_{|x|^k-1})$.

Use same reduction as **CIRCUIT VALUE**; the only difference is that $c_i$ is now a variable at row $i$ of the table!
Theorem (Cook)

SAT is NP-complete.

Proof.

We will show that CIRCUIT SAT is NP-complete. Cook’s theorem follows. Let $L \in \textbf{NP}$; this means that $L$ is decided by a NDTM $M = (K, \Sigma, \delta, s)$, which halts with a “yes” or “no” on all strings $x \in \Sigma^*$ in at most $|x|^k$ time. Standardize the Turing Machine so that degree of non-determinism is exactly 2. It follows that a sequence of non-deterministic choices is a bit-string $(c_0, c_1, \ldots, c_{|x|^k-1})$.

Use same reduction as CIRCUIT VALUE; the only difference is that $c_i$ is now a variable at row $i$ of the table!