Reductions and Completeness

K. Subramani[1]

[1]Lane Department of Computer Science and Electrical Engineering
West Virginia University

February 24, 2009

Outline

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

Outline

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 $x \in A \Leftrightarrow R(x) \in B$.

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 $x \in \Sigma^*$, $|x| = n$, $x \in L_1 \leftrightarrow R(x) \in L_2$.

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 $x \in A \Leftrightarrow R(x) \in B$.

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 $x \in \Sigma^*$, $|x| = n$, $x \in L_1 \leftrightarrow R(x) \in L_2$.

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 $x \in A \Leftrightarrow R(x) \in B$.

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 $x \in \Sigma^*$, $|x| = n$, $x \in L_1 \leftrightarrow R(x) \in L_2$.

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 $x \in A \Leftrightarrow R(x) \in B$.

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 $x \in \Sigma^*$, $|x| = n$, $x \in L_1 \leftrightarrow R(x) \in L_2$.

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 $x \in A \Leftrightarrow R(x) \in B$.

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 $x \in \Sigma^*$, $|x| = n$, $x \in L_1 \leftrightarrow R(x) \in L_2$.

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.

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.

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.

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: $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. $[C_1]$.
Step 3: $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. $[C_2]$.
Step 4: $(x_{i1} \vee x_{i2} \ldots \vee x_{in}), i = 1, 2 \ldots n$. $[C_3]$.
Step 5: $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. $[C_4]$.
Step 6: $(\neg x_{ki} \vee \neg x_{(k+1).j}, k = 1, 2, \ldots, n-1, (i, j) \notin G$. $[C_5]$.
Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

Argument: Let $\tilde{\mathbf{x}}$ denote a satisfying assignment to ϕ. 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 π is a permutation. We show that ϕ 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: $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. $[C_1]$.

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

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

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

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

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

Argument: Let $\tilde{\mathbf{x}}$ denote a satisfying assignment to ϕ. 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 π is a permutation. We show that ϕ 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: $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. $[C_1]$.
Step 3: $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. $[C_2]$.
Step 4: $(x_{i1} \vee x_{i2} \ldots \vee x_{in}), i = 1, 2 \ldots n$. $[C_3]$.
Step 5: $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. $[C_4]$.
Step 6: $(\neg x_{ki} \vee \neg x_{(k+1),j}, k = 1, 2, \ldots, n-1, (i,j) \notin G$. $[C_5]$.
Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

Argument: Let $\bar{x}$ denote a satisfying assignment to ϕ. 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 π is a permutation. We show that ϕ 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: $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. $[C_1]$.
Step 3: $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. $[C_2]$.
Step 4: $(x_{i1} \vee x_{i2} \ldots \vee x_{in}), i = 1, 2 \ldots n$. $[C_3]$.
Step 5: $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. $[C_4]$.
Step 6: $(\neg x_{ki} \vee \neg x_{(k+1).j}, k = 1, 2, \ldots, n-1, (i, j) \notin G$. $[C_5]$.
Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

Argument: Let $\vec{x}$ denote a satisfying assignment to ϕ. 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 π is a permutation. We show that ϕ 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: $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. $[C_1]$.
Step 3: $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. $[C_2]$.
Step 4: $(x_{i1} \vee x_{i2} \ldots \vee x_{in}), i = 1, 2 \ldots n$. $[C_3]$.
Step 5: $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. $[C_4]$.
Step 6: $(\neg x_{ki} \vee \neg x_{(k+1).j}, k = 1, 2, \ldots, n-1, (i, j) \notin G$. $[C_5]$.
Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

Argument: Let $\vec{x}$ denote a satisfying assignment to ϕ. 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 π is a permutation. We show that ϕ 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: $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. $[C_1]$.
Step 3: $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. $[C_2]$.
Step 4: $(x_{i1} \vee x_{i2} \ldots \vee x_{in}), i = 1, 2 \ldots n$. $[C_3]$.
Step 5: $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. $[C_4]$.
Step 6: $(\neg x_{ki} \vee \neg x_{(k+1),j}, k = 1, 2, \ldots, n-1, (i, j) \notin G$. $[C_5]$.
Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

Argument: Let $\tilde{\mathbf{x}}$ denote a satisfying assignment to ϕ. 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 π is a permutation. We show that ϕ 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: $(x_{1j} \vee x_{2j} \dots x_{nj}), j = 1, 2, \dots, n$. $[C_1]$.
Step 3: $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \dots n, i = 1, 2, \dots, n, k = 1, 2, \dots n, k \neq i$. $[C_2]$.
Step 4: $(x_{i1} \vee x_{i2} \dots \vee x_{in}), i = 1, 2 \dots n$. $[C_3]$.
Step 5: $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \dots, n, j, k = 1, 2, \dots, n, j \neq k$. $[C_4]$.
Step 6: $(\neg x_{ki} \vee \neg x_{(k+1),j}, k = 1, 2, \dots, n-1, (i, j) \notin G$. $[C_5]$.
Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

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

Let $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ 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: $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. $[C_1]$.
Step 3: $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. $[C_2]$.
Step 4: $(x_{i1} \vee x_{i2} \ldots \vee x_{in}), i = 1, 2 \ldots n$. $[C_3]$.
Step 5: $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. $[C_4]$.
Step 6: $(\neg x_{ki} \vee \neg x_{(k+1),j}, k = 1, 2, \ldots, n-1, (i, j) \notin G$. $[C_5]$.
Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

Argument: Let $\vec{x}$ denote a satisfying assignment to ϕ. 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 π is a permutation. We show that ϕ 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: $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. $[C_1]$.
Step 3: $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. $[C_2]$.
Step 4: $(x_{i1} \vee x_{i2} \ldots \vee x_{in}), i = 1, 2 \ldots n$. $[C_3]$.
Step 5: $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. $[C_4]$.
Step 6: $(\neg x_{ki} \vee \neg x_{(k+1),j}, k = 1, 2, \ldots, n-1, (i,j) \notin G$. $[C_5]$.
Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

Argument: Let $\tilde{\mathbf{x}}$ denote a satisfying assignment to ϕ. 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 π is a permutation. We show that ϕ 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: $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. $[C_1]$.
Step 3: $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. $[C_2]$.
Step 4: $(x_{i1} \vee x_{i2} \ldots \vee x_{in}), i = 1, 2 \ldots n$. $[C_3]$.
Step 5: $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. $[C_4]$.
Step 6: $(\neg x_{ki} \vee \neg x_{(k+1),j}, k = 1, 2, \ldots, n-1, (i, j) \notin G$. $[C_5]$.
Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

Argument: Let $\tilde{x}$ denote a satisfying assignment to ϕ. 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 π is a permutation. We show that ϕ 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: $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. $[C_1]$.
Step 3: $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. $[C_2]$.
Step 4: $(x_{i1} \vee x_{i2} \ldots \vee x_{in}), i = 1, 2 \ldots n$. $[C_3]$.
Step 5: $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. $[C_4]$.
Step 6: $(\neg x_{ki} \vee \neg x_{(k+1),j}, k = 1, 2, \ldots, n-1, (i, j) \notin G$. $[C_5]$.
Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

Argument: Let $\tilde{\mathbf{x}}$ denote a satisfying assignment to ϕ. 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 π is a permutation. We show that ϕ 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: $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n$. $[C_1]$.
Step 3: $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i$. $[C_2]$.
Step 4: $(x_{i1} \vee x_{i2} \ldots \vee x_{in}), i = 1, 2 \ldots n$. $[C_3]$.
Step 5: $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. $[C_4]$.
Step 6: $(\neg x_{ki} \vee \neg x_{(k+1),j}, k = 1, 2, \ldots, n-1, (i,j) \notin G$. $[C_5]$.
Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

Argument: Let $\tilde{\mathbf{x}}$ denote a satisfying assignment to ϕ. 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 π is a permutation. We show that ϕ is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

Input instance: A circuit C.
Output instance: A CNF formula ϕ such that ϕ is satisfiable if and only if C is.
Step 1: The variables of ϕ will contain all the variables of C. Additionally, for each gate g in C, we create a new variable in ϕ, 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 ϕ.
Step 3: If g is a **true** gate, add (g) to ϕ; 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 ϕ.
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 ϕ.
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 ϕ.
Step 7: If g is an output gate, add the clause (g).
Argument: If C is satisfiable, then ϕ is satisfiable.

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

CIRCUIT SAT to SAT

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

If ϕ is satisfiable, then C is satisfiable.

Sample Reductions (contd.)

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.

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. $\square$

Completeness

Definition

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

Definition

A complexity class $\mathcal{C}$ is closed under reductions, if $((L \in \mathcal{C}) \wedge (L' \leq L)) \rightarrow (L' \in \mathcal{C})$.

Proposition

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

Corollary

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

Completeness

Definition

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

Definition

A complexity class $\mathcal{C}$ is closed under reductions, if $((L \in \mathcal{C}) \wedge (L' \leq L)) \rightarrow (L' \in \mathcal{C})$.

Proposition

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

Corollary

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

Completeness

Definition

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

Definition

A complexity class $\mathcal{C}$ is closed under reductions, if $((L \in \mathcal{C}) \wedge (L' \leq L)) \rightarrow (L' \in \mathcal{C})$.

Proposition

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

Corollary

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

Completeness

Definition

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

Definition

A complexity class $\mathcal{C}$ is closed under reductions, if $((L \in \mathcal{C}) \wedge (L' \leq L)) \rightarrow (L' \in \mathcal{C})$.

Proposition

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

Corollary

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

Completeness

Definition

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

Definition

A complexity class $\mathcal{C}$ is closed under reductions, if $((L \in \mathcal{C}) \wedge (L' \leq L)) \rightarrow (L' \in \mathcal{C})$.

Proposition

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

Corollary

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

Completeness

Definition

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

Definition

A complexity class $\mathcal{C}$ is closed under reductions, if $((L \in \mathcal{C}) \wedge (L' \leq L)) \rightarrow (L' \in \mathcal{C})$.

Proposition

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

Corollary

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

Completeness

Definition

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

Definition

A complexity class $\mathcal{C}$ is closed under reductions, if $((L \in \mathcal{C}) \wedge (L' \leq L)) \rightarrow (L' \in \mathcal{C})$.

Proposition

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

Corollary

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

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 (Γ) 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 $j = 2$.

□

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 ⊔s, so that the length of the string is exactly $|x|^k$.

(iv) The tape alphabet (Γ) 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 $j = 2$.

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 (Γ) 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 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 (Γ) 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 $j = 2$.

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 (Γ) 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 $j = 2$.

□

P-completeness of CIRCUIT VALUE (contd.)

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}, $0 \leq i \leq |x|^k -$, $0 \leq j \leq |x|^k - 1$ 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. □

P-completeness of CIRCUIT VALUE (contd.)

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}, $0 \leq i \leq |x|^k -$, $0 \leq j \leq |x|^k - 1$ 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. □

P-completeness of CIRCUIT VALUE (contd.)

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}, $0 \le i \le |x|^k -$, $0 \le j \le |x|^k - 1$ and $1 \le l \le m$.
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. □

P-completeness of CIRCUIT VALUE (contd.)

Proof.

When $i = 0$ or $j = 0$ or $j = |x|^k$, the contents of T_{ij} are known apriori.
Crucial observation: $T_i j$ 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}, $0 \leq i \leq |x|^k -$, $0 \leq j \leq |x|^k - 1$ 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 $T_i j$.

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

P-completeness of CIRCUIT VALUE (contd.)

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}, $0 \leq i \leq |x|^k -$, $0 \leq j \leq |x|^k - 1$ 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. □

P-completeness of CIRCUIT VALUE (contd.)

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}, $0 \leq i \leq |x|^k -$, $0 \leq j \leq |x|^k - 1$ 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. □

P-completeness of CIRCUIT VALUE (contd.)

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}, $0 \leq i \leq |x|^k -$, $0 \leq j \leq |x|^k - 1$ 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 $L \in$ **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 $L \in$ **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 $L \in$ **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 $L \in$ **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 $L \in$ **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! □