Reductions and Completeness

K. Subramani¹

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

February 24, 2009

Outline



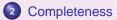
- 2 Completeness• P-completenes
 - NP-completeness

<ロ> <同> <同> < 同> < 三> < 三>

E.







- P-completeness
- NP-completeness

E

A D > A D > A D
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A
A

Main concept

Comparing problem difficulty through $A \le B$. 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, $x \in L_1 \leftrightarrow R(x) \in L_2$.

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, $x \in L_1 \leftrightarrow R(x) \in L_2$.

Main concept

Comparing problem difficulty through $A \le B$. 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, $x \in L_1 \leftrightarrow R(x) \in L_2$.

Main concept

Comparing problem difficulty through $A \le B$. 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, $x \in L_1 \leftrightarrow R(x) \in L_2$.

Main concept

Comparing problem difficulty through $A \le B$. 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, $x \in L_1 \leftrightarrow R(x) \in L_2$.

Completeness

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.

Completeness

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.

E

Completeness

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.

< ロ > < 同 > < 三 > < 三 > -

E

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 $i^{(0)}$ node in the Hamilton Path (may or may not be true). Step 2: $(x_{1j} \lor x_{2j} \dots x_{nj}), j = 1, 2, \dots, n.$ [C₁]. Step 3: $(-x_{ij} \lor \neg x_{kj}), j = 1, 2, \dots, n.$ [C₁]. Step 4: $(x_{1j} \lor x_{2j} \dots \lor x_{nj}), i = 1, 2, \dots, n.$ [C₃]. Step 5: $(-x_{ij} \lor \neg x_{k}), i = 1, 2, \dots, n. j, k = 1, 2, \dots, n. j \neq k.$ [C₄]. Step 6: $(-x_{k} \lor \neg x_{k+1}), j, k = 1, 2, \dots, n-1, (i, j) \notin G.$ [C₅]. Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

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

Let $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ is satisfiable.

(a)

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 *l*th node in the Hamilton Path (may or may not be true). Step 2: $(x_{1j} \lor x_{2j} \dots x_{nj}), j = 1, 2, \dots, n, [C_1]$. Step 3: $(-x_{ij} \lor \neg x_{ij}), j = 1, 2, \dots, n, i = 1, 2, \dots, n, k = 1, 2, \dots, n, k \neq i$. [*C*₂]. Step 4: $(x_{11} \lor x_{22} \dots \lor x_{nj}), i = 1, 2, \dots, n, [C_n]$. Step 5: $(\neg x_{ij} \lor \neg x_{ik}), i = 1, 2, \dots, n, j, k = 1, 2, \dots, n, j \neq k$. [*C*₄]. Step 6: $(\neg x_{ik} \lor \neg x_{ik}), i = 1, 2, \dots, n - 1, (i, j) \notin G$. [*C*₆]. Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

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

Let $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ is satisfiable.

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_n , where x_n represents the fact that node *i* is the i^m node in the Hamilton Path (may or may not be true). Step 2: $(x_1 \vee x_2 \dots x_n)$, $i = 1, 2, \dots, n$, $[G_1]$. Step 4: $(x_1 \vee x_2 \dots x_n)$, $i = 1, 2, \dots, n$, $[G_1]$. Step 5: $(x_n \vee x_n)$, $i = 1, 2, \dots, n$, $[G_1]$. Step 5: $(x_n \vee x_n)$, $i = 1, 2, \dots, n$, $k = 1, 2, \dots, n$, $j \neq k$, $[G_n]$. Step 7: $\phi = C_1 \wedge C_2 \wedge C_3 \wedge C_6$.

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

Let $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ is satisfiable.

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 3: $(\neg x_{ij} \lor \neg x_{kj})$, $j = 1, 2, ..., n, i = 1, 2, ..., n, k = 1, 2, ..., n, k \neq i$. $[C_2]$. Step 4: $(x_{i1} \lor \neg x_{kj})$, $j = 1, 2, ..., n, [C_3]$. Step 5: $(\neg x_{ij} \lor \neg x_{ik})$, $i = 1, 2, ..., n, j, k = 1, 2, ..., n, j \neq k$. $[C_4]$. Step 6: $(\neg x_{ki} \lor \neg x_{(k+1),j})$, k = 1, 2, ..., 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 $\tilde{\mathbf{x}}$ denote a satisfying assignment to ϕ . We show that there must exist a Hamilton Path in G.

Let $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ is satisfiable.

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: $(x_{1j} \lor x_{2j} \ldots x_{nj})$, $j = 1, 2, \ldots, n$. [*C*₁]. Step 3: $(x_{nj} \lor x_{2j} \ldots x_{nj})$, $i = 1, 2, \ldots, n$. [*C*₁]. Step 4: $(x_{nj} \lor x_{2j} \ldots x_{nj})$, $i = 1, 2, \ldots, n$. [*C*₁]. Step 4: $(x_{nj} \lor x_{2j} \ldots x_{nj})$, $i = 1, 2, \ldots, n$. [*C*₁]. Step 5: $(x_{nj} \lor x_{2j} \ldots x_{nj})$, $i = 1, 2, \ldots, n$. [*C*₁]. Step 6: $(x_{nj} \lor x_{2j} \ldots x_{nj})$, $i = 1, 2, \ldots, n$. [*C*₁].

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

Let $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ is satisfiable.

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: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n.$ [*C*₁]. Step 3: $(\neg x_{ij} \lor \neg x_{kj}), j = 1, 2, \ldots, n.$ *i* [*C*₁]. Step 4: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n.$ *k* = 1, 2, ..., *n*, *k* = *i*. [*C*₂]. Step 5: $(\neg x_{ij} \lor \neg x_{kj}), i = 1, 2, \ldots, n.$ [*C*₁]. Step 6: $(\neg x_{ij} \lor \neg x_{kj}), i = 1, 2, \ldots, n.$ *j* = *k*. [*C*₄]. Step 7: $(\neg x_{ij} \lor \neg x_{kj}), i = 1, 2, \ldots, n.$ *k* = *i*. (*n*) $\neq K$. [*C*₄].

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

Let $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ is satisfiable.

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: $(x_{1j} \lor x_{2j} \ldots x_{nj})$, $j = 1, 2, \ldots, n$. [*C*₁]. Step 3: $(\neg x_{ij} \lor \neg x_{kj})$, $j = 1, 2, \ldots, n$. [*C*₁]. Step 4: $(x_{11} \lor x_{2j} \ldots \lor x_{in})$, $i = 1, 2, \ldots, n$, $k = 1, 2, \ldots, n$, $k \neq i$. [*C*₂]. Step 5: $(\neg x_{ij} \lor \neg x_{kj})$, $i = 1, 2, \ldots, n$, $i = 1, 2, \ldots, n$, $j \neq k$. [*C*₄]. Step 6: $(\neg x_{ij} \lor \neg x_{kj})$, $k = 1, 2, \ldots, n - 1$, $(i, j) \notin G$. [*C*₆]. Step 7: $\phi = C \land C_2 \land C_2 \land C_3 \land C_4$. Argument: Let *R* 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.

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: $(x_{1j} \lor x_{2j} \dots x_{nj}), j = 1, 2, \dots, n.$ [*C*₁]. Step 3: $(\neg x_{ij} \lor \neg x_{kj}), j = 1, 2, \dots, n.$ [*C*₁]. Step 4: $(x_{i1} \lor x_{i2} \dots \lor x_{in}), i = 1, 2, \dots, n.$ [*C*₃]. Step 5: $(\neg x_{ij} \lor \neg x_{ik}), i = 1, 2, \dots, n.$ [*C*₃]. Step 5: $(\neg x_{ij} \lor \neg x_{ik}), i = 1, 2, \dots, n.$ [*K* = 1, 2, ..., $n, j \neq k$. [*C*₄]. Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$

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

Let $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ is satisfiable.

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: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n. [C_1]$. Step 3: $(-x_{ij} \lor \neg x_{kj}), j = 1, 2, \ldots, n, i = 1, 2, \ldots, n, k = 1, 2, \ldots, n, k \neq i$. [*C*₂]. Step 4: $(x_{i1} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k$. [*C*₄]. Step 5: $(-x_{ij} \lor \neg x_{ik}), i = 1, 2, \ldots, n - 1, (i, j) \notin G$. [*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 $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ is satisfiable.

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: $(x_{1j} \lor x_{2j} \ldots x_{nj})$, $j = 1, 2, \ldots, n$. $[C_1]$. 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: $(x_{11} \lor x_{i2} \ldots \lor x_{in})$, $i = 1, 2, \ldots, n$, $j \in K$. $[C_4]$. Step 5: $(-x_{ki} \lor -x_{k(k)})$, $i = 1, 2, \ldots, n - 1$, $(i, j) \notin G$. $[C_5]$. Step 7: $\phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5$.

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

Let $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ is satisfiable.

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: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n. [C_1]$. Step 3: $(-x_{ij} \lor \neg x_{kj}), j = 1, 2, \ldots, n, i = 1, 2, \ldots, n, k = 1, 2, \ldots, n, k \neq i. [C_2]$. Step 4: $(x_{i1} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k. [C_4]$. Step 5: $(-x_{kj} \lor \neg x_{k}), i = 1, 2, \ldots, n, n - 1, (i, j) \notin G$. [C₅]. Step 7: $\phi = C_1 \land C_2 \land C_1 \land C_4 \land C_5$.

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

Let $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ is satisfiable.

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: $(x_{1j} \lor x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n. [C_1]$. Step 3: $(-x_{ij} \lor \neg x_{kj}), j = 1, 2, \ldots, n, i = 1, 2, \ldots, n, k = 1, 2, \ldots, n, k \neq i. [C_2]$. Step 4: $(x_{i1} \lor x_{i2} \ldots \lor x_{in}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k. [C_4]$. Step 5: $(-x_{kj} \lor \neg x_{k}), i = 1, 2, \ldots, n, n - 1, (i, j) \notin G$. [C₅]. Step 7: $\phi = C_1 \land C_2 \land C_1 \land C_4 \land C_5$.

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

Let $\pi = (\pi(1), \pi(2) \dots \pi(n))$ denote a Hamilton path, where π is a permutation. We show that ϕ is satisfiable.

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.

<ロ> < 四> < 四> < 回> < 回> < 回> <

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.

<ロ> < 四> < 四> < 回> < 回> < 回> <

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 .

Argument: If C is satisfiable, then ϕ is satisfiable.

If ϕ is satisfiable, then C is satisfiable.

<ロ> < 四> < 四> < 回> < 回> < 回> <

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.

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.

f ϕ is satisfiable, then C is satisfiable.

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.

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.

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.

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

f ϕ is satisfiable, then C is satisfiable.

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).

If ϕ is satisfiable, then C is satisfiable.

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.

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.

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.

Reductions

Completeness

Sample Reductions (contd.)

Reduction by generalization

CIRCUIT VALUE to CIRCUIT SAT. R is the identity function!

E

Reductions

Completeness

Sample Reductions (contd.)

Reduction by generalization

CIRCUIT VALUE to CIRCUIT SAT. R is the identity function!

< ロ > < 同 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ >

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.

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'.

E

P-completeness NP-completeness

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'.

◆□ > ◆□ > ◆ Ξ > ◆ Ξ > ・ Ξ · ���

P-completeness NP-completeness

Completeness

Definition

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

reduced to L.

Definition

A complexity class $\ensuremath{\mathcal{C}}$ is closed under reductions, if

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

Proposition

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

Corollary

If two classes 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'.

P-completeness NP-completeness

Completeness

Definition

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

reduced to L.

Definition

A complexity class $\ensuremath{\mathcal{C}}$ is closed under reductions, if

 $((L \in \mathcal{C}) \land (L' \leq L)) \to (L' \in \mathcal{C}).$

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'.

< ロ > < 同 > < 臣 > < 臣 > -

P-completeness NP-completeness

Completeness

Definition

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

reduced to L.

Definition

A complexity class $\ensuremath{\mathcal{C}}$ 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'.

P-completeness NP-completeness

Completeness

Definition

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

reduced to L.

Definition

A complexity class $\ensuremath{\mathcal{C}}$ 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'.

P-completeness NP-completeness

Completeness

Definition

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

reduced to L.

Definition

A complexity class $\ensuremath{\mathcal{C}}$ 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.

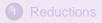
Corollary

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'.

< ロ > < 同 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < 回 > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ >

P-completeness NP-completeness

Outline





E.

《口》《聞》《臣》《臣》

P-completeness NP-completeness

P-completeness of CIRCUIT VALUE

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.

 \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 NP-completeness

P-completeness of CIRCUIT VALUE

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.

 \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 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 ⊔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 NP-completeness

P-completeness of CIRCUIT VALUE

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.

 \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 NP-completeness

P-completeness of CIRCUIT VALUE

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.

 \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 NP-completeness

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 3*m* 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.

The reduction can be accomplished in log lxl space

P-completeness NP-completeness

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 $m = \lceil \log |f| \rceil$. 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} , $0 \le i \le |x|^k - 0 \le j \le |x|^k - 1$ and $1 \le i \le m$. Each binary entry S_i depends only on the 3m entries S_{ij} , $0 \le i \le |x|^k - 0$, S_{ij} ,

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

P-completeness NP-completeness

P-completeness of CIRCUIT VALUE (contd.)

Proof.

When i = 0 or j = 0 or $j = |\mathbf{x}|^k$, 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 $\mathbf{s} = (\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_m)$, 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} of $\mathbf{s}_{ij} = 0 \le j \le |\mathbf{x}|^k - 1$ 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, ... m. But these are boolean functions and hence can be contined through dates

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

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

P-completeness NP-completeness

P-completeness of CIRCUIT VALUE (contd.)

Proof.

When i = 0 or j = 0 or $j = |\mathbf{x}|^k$, 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 $\mathbf{s} = (\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_m)$, 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} , $0 \le i \le |\mathbf{x}|^k - 0 \le j \le |\mathbf{x}|^k - 1$ and $1 \le I \le m$. Each binary entry S_{ij} depends only on the 3m entries $S_{i-1,j-1,i'}$, $S_{i-1,j,i'}$, $S_{i-1,j+1,i'}$, where I' ranges over $1, 2, \ldots, m$. But these are boolean functions and hence can be captured through gates. Create $(|\mathbf{x}|^n - 1) \ge (|\mathbf{x}|^n - 2)$ gates, one for each entry T_{ij} .

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

< 口 > < 同 > < 三 > < 三 > -

P-completeness NP-completeness

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 $s = (s_1, s_2, ..., s_m)$, 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, ..., 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 $T_{l,l}$.

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

< ロ > < 同 > < 三 > < 三 > -

P-completeness NP-completeness

P-completeness of CIRCUIT VALUE (contd.)

Proof.

When i = 0 or j = 0 or $j = |\mathbf{x}|^k$, 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 $\mathbf{s} = (\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_m)$, 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} , $0 \le i \le |\mathbf{x}|^k - 0 \le j \le |\mathbf{x}|^k - 1$ 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 $1, 2, \dots, m$. But these are boolean functions and hence can be captured through gates.

Create $(|x|^{\kappa} - 1) \times (|x|^{\kappa} - 2)$ gates, one for each entry $T_i j$.

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

< 口 > < 同 > < 三 > < 三 > -

P-completeness NP-completeness

P-completeness of CIRCUIT VALUE (contd.)

Proof.

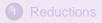
When i = 0 or j = 0 or $j = |x|^k$, the contents of T_{ij} are known apriori. Crucial observation: T_{ij} depends only on the entries $T_{i-1,j-1}$, $T_{i-1,j}$ and $T_{i-1,j+1}$. Why? Encode each tape symbol as a binary vector $s = (s_1, s_2, \ldots, s_m)$, where $m = \lceil \log |\Gamma| \rceil$. The 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.

< 口 > < 同 > < 三 > < 三 > -

P-completeness NP-completeness

Outline





• NP-completeness

E.

《口》《聞》《臣》《臣》

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!

(二)、(同)、(三)、(三)、

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 $L \in \mathbf{NP}$; 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

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

(a)

э.

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 $L \in \mathbf{NP}$; 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!

<ロ> <同> <同> < 回> < 回> < => < => <

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 $L \in \mathbf{NP}$; 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_1 is now a variable at row / of

the table!

< ロ > < 同 > < 三 > < 三 > -

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 $L \in \mathbf{NP}$; 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!

< ロ > < 同 > < 三 > < 三 > -