Algorithms and datastructures II

Lecture 9: Introduction to the complexity theory

Jan Hubička

Department of Applied Mathematics
Charles University
Prague

Nov 30 2020

Decision problems

Definition

A (decision) problem if a function from $\{0,1\}^*$ (the set of all possible inputs) to $\{0,1\}$.

Equivalently we can think of a decision problem to be a subset $A \subseteq \{0,1\}^*$

Decision problems

Definition

A (decision) problem if a function from $\{0,1\}^*$ (the set of all possible inputs) to $\{0,1\}$.

Equivalently we can think of a decision problem to be a subset $A \subseteq \{0,1\}^*$

Example (Bipartite matching)

Given a bipartite graph G and a number $k \in \mathbb{N}$. We want to decide if there exists matching in G with at least k edges.

Decision problems

Definition

A (decision) problem if a function from $\{0,1\}^*$ (the set of all possible inputs) to $\{0,1\}$.

Equivalently we can think of a decision problem to be a subset $A \subseteq \{0,1\}^*$

Example (Bipartite matching)

Given a bipartite graph G and a number $k \in \mathbb{N}$. We want to decide if there exists matching in G with at least k edges.

Lets satisfy the definition:

1. We need to encode input in a binary form that can describe every input size.

Decision problems

Definition

A (decision) problem if a function from $\{0,1\}^*$ (the set of all possible inputs) to $\{0,1\}$.

Equivalently we can think of a decision problem to be a subset $A \subseteq \{0,1\}^*$

Example (Bipartite matching)

Given a bipartite graph G and a number $k \in \mathbb{N}$. We want to decide if there exists matching in G with at least k edges.

Lets satisfy the definition:

1. We need to encode input in a binary form that can describe every input size.
2. Decision problem formally needs to produce result even for malformed inputs.

Decision problems

Definition

A (decision) problem if a function from $\{0,1\}^*$ (the set of all possible inputs) to $\{0,1\}$.

Equivalently we can think of a decision problem to be a subset $A \subseteq \{0,1\}^*$

Example (Bipartite matching)

Given a bipartite graph G and a number $k \in \mathbb{N}$. We want to decide if there exists matching in G with at least k edges.

Lets satisfy the definition:

1. We need to encode input in a binary form that can describe every input size.
2. Decision problem formally needs to produce result even for malformed inputs.
   We can just define answer as 0 in that case.

Decision problems

Definition

A (decision) problem if a function from $\{0,1\}^*$ (the set of all possible inputs) to $\{0,1\}$.

Equivalently we can think of a decision problem to be a subset $A \subseteq \{0,1\}^*$

Example (Bipartite matching)

Given a bipartite graph G and a number $k \in \mathbb{N}$. We want to decide if there exists matching in G with at least k edges.

Lets satisfy the definition:

1. We need to encode input in a binary form that can describe every input size.
2. Decision problem formally needs to produce result even for malformed inputs.
   We can just define answer as 0 in that case.

Can we solve the problem?

Decision problems

Definition

A (decision) problem if a function from $\{0,1\}^*$ (the set of all possible inputs) to $\{0,1\}$.

Equivalently we can think of a decision problem to be a subset $A \subseteq \{0,1\}^*$

Example (Bipartite matching)

Given a bipartite graph G and a number $k \in \mathbb{N}$. We want to decide if there exists matching in G with at least k edges.

Lets satisfy the definition:

1. We need to encode input in a binary form that can describe every input size.
2. Decision problem formally needs to produce result even for malformed inputs.
   We can just define answer as 0 in that case.

Can we solve the problem?
Yes: we can “reduce” it to a network flow problem.

Decision problems

Definition

A (decision) problem if a function from $\{0,1\}^*$ (the set of all possible inputs) to $\{0,1\}$.

Equivalently we can think of a decision problem to be a subset $A \subseteq \{0,1\}^*$

Example (Bipartite matching)

Given a bipartite graph G and a number $k \in \mathbb{N}$. We want to decide if there exists matching in G with at least k edges.

Lets satisfy the definition:

1. We need to encode input in a binary form that can describe every input size.
2. Decision problem formally needs to produce result even for malformed inputs. We can just define answer as 0 in that case.

Can we solve the problem?
Yes: we can "reduce" it to a network flow problem.

Definition (Reduction)

Given problems A and B, we say that A is (polynomial time) reducible to B (and write $A \longrightarrow B$) if there exists function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$ it holds $A(x) = B(f(x))$ and f can be computed in polynomial time relative to $|x|$. Function f is also called (polynomial time) reduction.

Properties of reduction

Definition (Reduction)

Given problems A and B, we say that A is (polynomial time) reducible to B (and write $A \longrightarrow B$) if there exists function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$ it holds $A(x) = B(f(x))$ and f can be computed in polynomial time relative to $|x|$. Function f is also called (polynomial time) reduction.

Intuition: $A \longrightarrow B$ implies that B is “at least as hard to solve as A”.

Properties of reduction

Definition (Reduction)

Given problems A and B, we say that A is (polynomial time) reducible to B (and write $A \longrightarrow B$) if there exists function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$ it holds $A(x) = B(f(x))$ and f can be computed in polynomial time relative to $|x|$. Function f is also called (polynomial time) reduction.

Intuition: $A \longrightarrow B$ implies that B is “at least as hard to solve as A”.

Lemma

If $A \longrightarrow B$ and B can be solved in polynomial time, then A can be solved in polynomial time too.

Properties of reduction

Definition (Reduction)

Given problems A and B, we say that A is (polynomial time) reducible to B (and write $A \longrightarrow B$) if there exists function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$ it holds $A(x) = B(f(x))$ and f can be computed in polynomial time relative to $|x|$. Function f is also called (polynomial time) reduction.

Intuition: $A \longrightarrow B$ implies that B is "at least as hard to solve as A".

Lemma

If $A \longrightarrow B$ and B can be solved in polynomial time, then A can be solved in polynomial time too.

Proof.

1. Assume that there is an algorithm solving B in $O(b^k)$ where b is length of the input and k is some constant.

Properties of reduction

Definition (Reduction)

Given problems A and B, we say that A is (polynomial time) reducible to B (and write $A \longrightarrow B$) if there exists function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$ it holds $A(x) = B(f(x))$ and f can be computed in polynomial time relative to $|x|$. Function f is also called (polynomial time) reduction.

Intuition: $A \longrightarrow B$ implies that B is “at least as hard to solve as A”.

Lemma

If $A \longrightarrow B$ and B can be solved in polynomial time, then A can be solved in polynomial time too.

Proof.

1. Assume that there is an algorithm solving B in $O(b^k)$ where b is length of the input and k is some constant.
2. Let f be a reduction $A \longrightarrow B$ computable in time $O(a^\ell)$ for input of length a.

Properties of reduction

Definition (Reduction)

Given problems A and B, we say that A is (polynomial time) reducible to B (and write $A \longrightarrow B$) if there exists function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$ it holds $A(x) = B(f(x))$ and f can be computed in polynomial time relative to $|x|$. Function f is also called (polynomial time) reduction.

Intuition: $A \longrightarrow B$ implies that B is "at least as hard to solve as A".

Lemma

If $A \longrightarrow B$ and B can be solved in polynomial time, then A can be solved in polynomial time too.

Proof.

1. Assume that there is an algorithm solving B in $O(b^k)$ where b is length of the input and k is some constant.
2. Let f be a reduction $A \longrightarrow B$ computable in time $O(a^\ell)$ for input of length a.
3. To solve $A(x)$ of length a we first compute $g(x)$ in time $O(a^\ell)$ and we produce output of length $O(a^\ell)$. Next we compute $B(f(x))$ in time $O((a^k)^\ell) = O(a^{k\ell})$.

Properties of reduction

Definition (Reduction)

Given problems A and B, we say that A is (polynomial time) reducible to B (and write $A \longrightarrow B$) if there exists function $f : \{0,1\}^* \to \{0,1\}^*$ such that for every $x \in \{0,1\}^*$ it holds $A(x) = B(f(x))$ and f can be computed in polynomial time relative to $|x|$. Function f is also called (polynomial time) reduction.

Intuition: $A \longrightarrow B$ implies that B is "at least as hard to solve as A".

Lemma

If $A \longrightarrow B$ and B can be solved in polynomial time, then A can be solved in polynomial time too.

Proof.

1. Assume that there is an algorithm solving B in $O(b^k)$ where b is length of the input and k is some constant.
2. Let f be a reduction $A \longrightarrow B$ computable in time $O(a^\ell)$ for input of length a.
3. To solve $A(x)$ of length a we first compute $g(x)$ in time $O(a^\ell)$ and we produce output of length $O(a^\ell)$. Next we compute $B(f(x))$ in time $O((a^k)^\ell) = O(a^{k\ell})$.
4. Overall time is $O(a^\ell + a^{k\ell})$

□

Properties of reduction II

"$\longrightarrow$" can be seen as a binary relation on the set of all problems. It satisfies:

1. "$\longrightarrow$" is reflexive: $A \longrightarrow A$.

Properties of reduction II

"$\longrightarrow$" can be seen as a binary relation on the set of all problems. It satisfies:

1. "$\longrightarrow$" is reflexive: $A \longrightarrow A$.
2. "$\longrightarrow$" is transitive: $A \longrightarrow B$ and $B \longrightarrow C \implies A \longrightarrow C$.

Properties of reduction II

"$\longrightarrow$" can be seen as a binary relation on the set of all problems. It satisfies:

1. "$\longrightarrow$" is reflexive: $A \longrightarrow A$.
2. "$\longrightarrow$" is transitive: $A \longrightarrow B$ and $B \longrightarrow C \implies A \longrightarrow C$.
3. "$\longrightarrow$" is not antisymmetric.

Properties of reduction II

"$\longrightarrow$" can be seen as a binary relation on the set of all problems. It satisfies:

1. "$\longrightarrow$" is reflexive: $A \longrightarrow A$.
2. "$\longrightarrow$" is transitive: $A \longrightarrow B$ and $B \longrightarrow C \implies A \longrightarrow C$.
3. "$\longrightarrow$" is not antisymmetric.
4. There exists problems such that $A \not\longrightarrow B$ and $B \not\longrightarrow A$.

Properties of reduction II

"$\longrightarrow$" can be seen as a binary relation on the set of all problems. It satisfies:

1. "$\longrightarrow$" is reflexive: $A \longrightarrow A$.
2. "$\longrightarrow$" is transitive: $A \longrightarrow B$ and $B \longrightarrow C \implies A \longrightarrow C$.
3. "$\longrightarrow$" is not antisymmetric.
4. There exists problems such that $A \not\longrightarrow B$ and $B \not\longrightarrow A$.

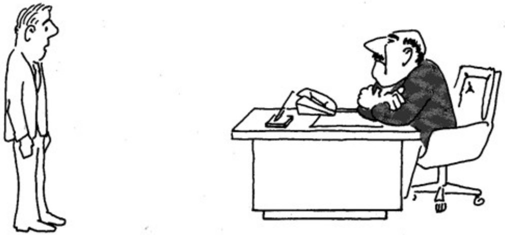
It follows that $\longrightarrow$ is a preorder (or quasiorder).

Properties of reduction II

"$\longrightarrow$" can be seen as a binary relation on the set of all problems. It satisfies:

1. "$\longrightarrow$" is reflexive: $A \longrightarrow A$.
2. "$\longrightarrow$" is transitive: $A \longrightarrow B$ and $B \longrightarrow C \implies A \longrightarrow C$.
3. "$\longrightarrow$" is not antisymmetric.
4. There exists problems such that $A \not\longrightarrow B$ and $B \not\longrightarrow A$.

It follows that $\longrightarrow$ is a preorder (or quasiorder).



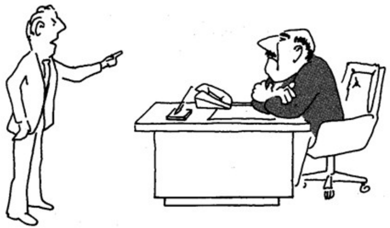
"I can't find an efficient algorithm, I guess I'm just too dumb."

Properties of reduction II

"$\longrightarrow$" can be seen as a binary relation on the set of all problems. It satisfies:

1. "$\longrightarrow$" is reflexive: $A \longrightarrow A$.
2. "$\longrightarrow$" is transitive: $A \longrightarrow B$ and $B \longrightarrow C \implies A \longrightarrow C$.
3. "$\longrightarrow$" is not antisymmetric.
4. There exists problems such that $A \not\longrightarrow B$ and $B \not\longrightarrow A$.

It follows that $\longrightarrow$ is a preorder (or quasiorder).



"I can't find an efficient algorithm, because no such algorithm is possible!"

Properties of reduction II

"$\longrightarrow$" can be seen as a binary relation on the set of all problems. It satisfies:

1. "$\longrightarrow$" is reflexive: $A \longrightarrow A$.
2. "$\longrightarrow$" is transitive: $A \longrightarrow B$ and $B \longrightarrow C \implies A \longrightarrow C$.
3. "$\longrightarrow$" is not antisymmetric.
4. There exists problems such that $A \not\longrightarrow B$ and $B \not\longrightarrow A$.

It follows that $\longrightarrow$ is a preorder (or quasiorder).

"I can't find an efficient algorithm, but neither can all these famous people."

Michael Garey and David S. Johnson "Computers and Intractability: A Guide to the Theory of NP-Completeness" (1979)

Properties of reduction II

"$\longrightarrow$" can be seen as a binary relation on the set of all problems. It satisfies:

1. "$\longrightarrow$" is reflexive: $A \longrightarrow A$.
2. "$\longrightarrow$" is transitive: $A \longrightarrow B$ and $B \longrightarrow C \implies A \longrightarrow C$.
3. "$\longrightarrow$" is not antisymmetric.
4. There exists problems such that $A \not\longrightarrow B$ and $B \not\longrightarrow A$.

It follows that $\longrightarrow$ is a preorder (or quasiorder).

"I can't find an efficient algorithm, I guess I'm just too dumb."

Properties of reduction II

"$\longrightarrow$" can be seen as a binary relation on the set of all problems. It satisfies:

1. "$\longrightarrow$" is reflexive: $A \longrightarrow A$.
2. "$\longrightarrow$" is transitive: $A \longrightarrow B$ and $B \longrightarrow C \implies A \longrightarrow C$.
3. "$\longrightarrow$" is not antisymmetric.
4. There exists problems such that $A \not\longrightarrow B$ and $B \not\longrightarrow A$.

It follows that $\longrightarrow$ is a preorder (or quasiorder).

"I can't find an efficient algorithm, because no such algorithm is possible!"

Properties of reduction II

"$\longrightarrow$" can be seen as a binary relation on the set of all problems. It satisfies:

1. "$\longrightarrow$" is reflexive: $A \longrightarrow A$.
2. "$\longrightarrow$" is transitive: $A \longrightarrow B$ and $B \longrightarrow C \implies A \longrightarrow C$.
3. "$\longrightarrow$" is not antisymmetric.
4. There exists problems such that $A \not\longrightarrow B$ and $B \not\longrightarrow A$.

It follows that $\longrightarrow$ is a preorder (or quasiorder).

"I can't find an efficient algorithm, but neither can all these famous people."

SAT: satisfiability of formulas in CNF

Definition (Conjuctive normal form)

Formula φ in conjunctive normal form (CNF) consists of

1. clauses separated by $\vee$ (“and”),
2. every clause consist of literals separated by $\wedge$ (“or”),
3. every literal is either variable or its negation.

SAT: satisfiability of formulas in CNF

Definition (Conjuctive normal form)

Formula φ in conjunctive normal form (CNF) consists of

1. clauses separated by $\vee$ (“and”),
2. every clause consist of literals separated by $\wedge$ (“or”),
3. every literal is either variable or its negation.

SAT

1. Input: formula φ in CNF
2. Output: 1 if and only if there exists an $\{0, 1\}$-assignment to variables such that $\varphi(\ldots) = 1$

SAT: satisfiability of formulas in CNF

Definition (Conjuctive normal form)

Formula φ in conjunctive normal form (CNF) consists of

1. clauses separated by $\vee$ (“and”),
2. every clause consist of literals separated by $\wedge$ (“or”),
3. every literal is either variable or its negation.

SAT

1. Input: formula φ in CNF
2. Output: 1 if and only if there exists an $\{0, 1\}$-assignment to variables such that $\varphi(\ldots) = 1$

Examples:

1. $(x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z)$

SAT: satisfiability of formulas in CNF

Definition (Conjuctive normal form)

Formula φ in conjunctive normal form (CNF) consists of

1. clauses separated by $\vee$ ("and"),
2. every clause consist of literals separated by $\wedge$ ("or"),
3. every literal is either variable or its negation.

SAT

1. Input: formula φ in CNF
2. Output: 1 if and only if there exists an $\{0, 1\}$-assignment to variables such that $\varphi(\ldots) = 1$

Examples:

1. $(x \vee y \vee z) \wedge (\neg x \vee y \vee z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z)$
2. $(x \vee y) \wedge (x \vee \neg y) \wedge \neg x$

SAT: satisfiability of formulas in CNF

3-SAT

Same as SAT but every clause has at most 3 literals

Observation

3-SAT $\longrightarrow$ SAT

Proof.

1. Assume that there is some clause with $k > 3$ literals. Write it as $(\alpha \vee \beta)$ where α has 2 literals and β has k literals.

SAT: satisfiability of formulas in CNF

3-SAT

Same as SAT but every clause has at most 3 literals

Observation

3-SAT $\longrightarrow$ SAT

Proof.

1. Assume that there is some clause with $k > 3$ literals. Write it as $(\alpha \vee \beta)$ where α has 2 literals and β has k literals.
2. Replace this clause by two caluses: $(\alpha \vee x)$ and $(\neg x \vee \beta)$.

SAT: satisfiability of formulas in CNF

3-SAT

Same as SAT but every clause has at most 3 literals

Observation

3-SAT $\longrightarrow$ SAT

Proof.

1. Assume that there is some clause with $k > 3$ literals. Write it as $(\alpha \vee \beta)$ where α has 2 literals and β has k literals.
2. Replace this clause by two caluses: $(\alpha \vee x)$ and $(\neg x \vee \beta)$.
3. Repeating steps 1 and 2 decomposes long clause to short in $k - 3$ steps. This can be done for all long clauses extending input polynomially.

SAT: satisfiability of formulas in CNF

3-SAT

Same as SAT but every clause has at most 3 literals

Observation

3-SAT $\longrightarrow$ SAT

Proof.

1. Assume that there is some clause with $k > 3$ literals. Write it as $(\alpha \vee \beta)$ where α has 2 literals and β has k literals.
2. Replace this clause by two caluses: $(\alpha \vee x)$ and $(\neg x \vee \beta)$.
3. Repeating steps 1 and 2 decomposes long clause to short in $k - 3$ steps. This can be done for all long clauses extending input polynomially.
4. Apply $3 - SAT$ to solve expanded formula. Original formula is satisfiable $\iff$ expanded formula is.

□

IndSet: Independent set in graph

Subset A of vertices of G is independent if there is no edge connecting two vertices in A.

IndSet

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an independent set of size k.

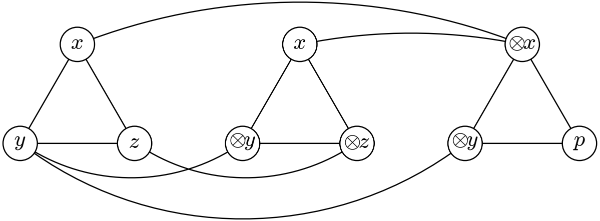
IndSet: Independent set in graph

Subset A of vertices of G is independent if there is no edge connecting two vertices in A.

IndSet

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an independent set of size k.

3-SAT $\longrightarrow$ IndSet



$$(x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee \neg y \vee p)$$

IndSet: Independent set in graph

Subset A of vertices of G is independent if there is no edge connecting two vertices in A.

IndSet

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an independent set of size k.

IndSet $\longrightarrow$ SAT

IndSet: Independent set in graph

Subset A of vertices of G is independent if there is no edge connecting two vertices in A.

IndSet

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an independent set of size k.

IndSet $\longrightarrow$ SAT

Given G and k we want to write formula that is satisfied iff G has independent set of size k.

1. Enumerate vertices as $1, \ldots, n$ and create variables $v_1, \ldots v_n$.
   $v_i = 1$ if v_i is in the independent set.

IndSet: Independent set in graph

Subset A of vertices of G is independent if there is no edge connecting two vertices in A.

IndSet

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an independent set of size k.

IndSet $\longrightarrow$ SAT

Given G and k we want to write formula that is satisfied iff G has independent set of size k.

1. Enumerate vertices as $1, \ldots, n$ and create variables $v_1, \ldots v_n$.
   $v_i = 1$ if v_i is in the independent set.
2. For every edge $\{i, j\}$ add clause $\neg v_i \vee \neg v_j$.

IndSet: Independent set in graph

Subset A of vertices of G is independent if there is no edge connecting two vertices in A.

IndSet

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an independent set of size k.

IndSet $\longrightarrow$ SAT

Given G and k we want to write formula that is satisfied iff G has independent set of size k.

1. Enumerate vertices as $1, \ldots, n$ and create variables $v_1, \ldots v_n$.
   $v_i = 1$ if v_i is in the independent set.
2. For every edge $\{i, j\}$ add clause $\neg v_i \vee \neg v_j$.
3. We need to test if set is large enough.

IndSet: Independent set in graph

Subset A of vertices of G is independent if there is no edge connecting two vertices in A.

IndSet

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an independent set of size k.

IndSet $\longrightarrow$ SAT

Given G and k we want to write formula that is satisfied iff G has independent set of size k.

1. Enumerate vertices as $1, \ldots, n$ and create variables $v_1, \ldots v_n$.
   $v_i = 1$ if v_i is in the independent set.
2. For every edge $\{i, j\}$ add clause $\neg v_i \vee \neg v_j$.
3. We need to test if set is large enough.
   Create $k \times n$ matrix X. $x_{i,j}$ will say that i-th element of the independent set is vertex j.

IndSet: Independent set in graph

Subset A of vertices of G is independent if there is no edge connecting two vertices in A.

IndSet

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an independent set of size k.

IndSet $\longrightarrow$ SAT

Given G and k we want to write formula that is satisfied iff G has independent set of size k.

1. Enumerate vertices as $1, \ldots, n$ and create variables $v_1, \ldots v_n$.
   $v_i = 1$ if v_i is in the independent set.
2. For every edge $\{i, j\}$ add clause $\neg v_i \vee \neg v_j$.
3. We need to test if set is large enough.
   Create $k \times n$ matrix X. $x_{i,j}$ will say that i-th element of the independent set is vertex j.
   Add following clauses:
   1. Every column has at most one 1. Add clauses $x_{i,j} \implies \neg x_{i',j}$ for every $i' \neq i$.

IndSet: Independent set in graph

Subset A of vertices of G is independent if there is no edge connecting two vertices in A.

IndSet

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an independent set of size k.

IndSet $\longrightarrow$ SAT

Given G and k we want to write formula that is satisfied iff G has independent set of size k.

1. Enumerate vertices as $1, \ldots, n$ and create variables $v_1, \ldots v_n$.
   $v_i = 1$ if v_i is in the independent set.
2. For every edge $\{i, j\}$ add clause $\neg v_i \vee \neg v_j$.
3. We need to test if set is large enough.
   Create $k \times n$ matrix X. $x_{i,j}$ will say that i-th element of the independent set is vertex j.
   Add following clauses:
   1. Every column has at most one 1. Add clauses $x_{i,j} \implies \neg x_{i',j}$ for every $i' \neq i$.
   2. Every row has at most one 1. Add clauses $x_{i,j} \implies \neg x_{i,j'}$ for every $j' \neq j$.

IndSet: Independent set in graph

Subset A of vertices of G is independent if there is no edge connecting two vertices in A.

IndSet

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an independent set of size k.

IndSet $\longrightarrow$ SAT

Given G and k we want to write formula that is satisfied iff G has independent set of size k.

1. Enumerate vertices as $1, \dots, n$ and create variables $v_1, \dots v_n$.
   $v_i = 1$ if v_i is in the independent set.
2. For every edge $\{i, j\}$ add clause $\neg v_i \vee \neg v_j$.
3. We need to test if set is large enough.
   Create $k \times n$ matrix X. $x_{i,j}$ will say that i-th element of the independent set is vertex j.
   Add following clauses:
   1. Every column has at most one 1. Add clauses $x_{i,j} \implies \neg x_{i',j}$ for every $i' \neq i$.
   2. Every row has at most one 1. Add clauses $x_{i,j} \implies \neg x_{i,j'}$ for every $j' \neq j$.
   3. Every row has at least one 1. Add clauses $(x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,n})$.

IndSet: Independent set in graph

Subset A of vertices of G is independent if there is no edge connecting two vertices in A.

IndSet

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an independent set of size k.

IndSet $\longrightarrow$ SAT

Given G and k we want to write formula that is satisfied iff G has independent set of size k.

1. Enumerate vertices as $1, \ldots, n$ and create variables $v_1, \ldots v_n$.
   $v_i = 1$ if v_i is in the independent set.
2. For every edge $\{i, j\}$ add clause $\neg v_i \vee \neg v_j$.
3. We need to test if set is large enough.
   Create $k \times n$ matrix X. $x_{i,j}$ will say that i-th element of the independent set is vertex j.
   Add following clauses:
    1. Every column has at most one 1. Add clauses $x_{i,j} \implies \neg x_{i',j}$ for every $i' \neq i$.
    2. Every row has at most one 1. Add clauses $x_{i,j} \implies \neg x_{i,j'}$ for every $j' \neq j$.
    3. Every row has at least one 1. Add clauses $(x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,n})$.
    4. Tie matrix and the independent set: Add clauses $x_{i,j} \implies v_j$.

Clique: Clique in a graph

Clique

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an clique of size k.

Clique: Clique in a graph

Clique

1. Input: Unoriented graph G and number k.
2. 1 if and only if G contains an clique of size k.

IndSet $\longrightarrow$ Clique
Clique $\longrightarrow$ IndSet

3,3-SAT

3-SAT

Same as 3-SAT but every variable appears in at most 3 literals.

3,3-SAT

3-SAT

Same as 3-SAT but every variable appears in at most 3 literals.

3-SAT $\longrightarrow$ 3,3-SAT

3,3-SAT

3-SAT

Same as 3-SAT but every variable appears in at most 3 literals.

3-SAT $\longrightarrow$ 3,3-SAT

if variable x appears more than 3 times, replace it by multiple variables $x_1, x_2, \ldots x_k$ and add clauses requiring them to have same values $(x_1 \implies x_2)$, $(x_2 \implies x_3)$, $\ldots$, $(x_{k-1} \implies x_k)$, $(x_k \implies x_1)$.

Definition (P)

P is the class of all (decision) problems that can be solved by a polynomial time algorithm.

$L \in P$ if and only if there exists algorithm A and polynomial f such that for every input x running $A(x)$ will finish in time at most $f(|x|)$ and $A(x) = L(x)$.

Definition (P)

P is the class of all (decision) problems that can be solved by a polynomial time algorithm.

$L \in P$ if and only if there exists algorithm A and polynomial f such that for every input x running $A(x)$ will finish in time at most $f(|x|)$ and $A(x) = L(x)$.

Definition (NP)

NP is the class of all (decision) problems L such that there exists some problem $K \in P$ and a polynomial g such that for every input x it holds that $L(x) = 1$ iff there exists $y \in \{0,1\}^*$ of lenght at most $g(|x|)$ such that $K(x,y) = 1$.

You can think of y as a certificate that $L(x) = 1$.

Definition (P)

P is the class of all (decision) problems that can be solved by a polynomial time algorithm.

$L \in P$ if and only if there exists algorithm A and polynomial f such that for every input x running $A(x)$ will finish in time at most $f(|x|)$ and $A(x) = L(x)$.

Definition (NP)

NP is the class of all (decision) problems L such that there exists some problem $K \in P$ and a polynomial g such that for every input x it holds that $L(x) = 1$ iff there exists $y \in \{0, 1\}^*$ of lenght at most $g(|x|)$ such that $K(x, y) = 1$.

You can think of y as a certificate that $L(x) = 1$.

Observation

SAT $\in NP$.

Definition (P)

P is the class of all (decision) problems that can be solved by a polynomial time algorithm.

$L \in P$ if and only if there exists algorithm A and polynomial f such that for every input x running $A(x)$ will finish in time at most $f(|x|)$ and $A(x) = L(x)$.

Definition (NP)

NP is the class of all (decision) problems L such that there exists some problem $K \in P$ and a polynomial g such that for every input x it holds that $L(x) = 1$ iff there exists $y \in \{0, 1\}^*$ of lenght at most $g(|x|)$ such that $K(x, y) = 1$.

You can think of y as a certificate that $L(x) = 1$.

Observation

SAT $\in NP$.

Observation

$P \subseteq NP$.

Definition (P)

P is the class of all (decision) problems that can be solved by a polynomial time algorithm.

$L \in P$ if and only if there exists algorithm A and polynomial f such that for every input x running $A(x)$ will finish in time at most $f(|x|)$ and $A(x) = L(x)$.

Definition (NP)

NP is the class of all (decision) problems L such that there exists some problem $K \in P$ and a polynomial g such that for every input x it holds that $L(x) = 1$ iff there exists $y \in \{0, 1\}^*$ of lenght at most $g(|x|)$ such that $K(x, y) = 1$.

You can think of y as a certificate that $L(x) = 1$.

Observation

SAT $\in NP$.

Observation

$P \subseteq NP$.

$P = NP$ is open since 1970's.

NP-completeness

Definition (*NP*-hardness)

Problem *L* is *NP*-hard if every problem from *NP* can be reduced to *L*.

NP-completeness

Definition (*NP*-hardness)

Problem L is *NP*-hard if every problem from *NP* can be reduced to L.

Lemma

If some NP-hard problem L is in P then $P = NP$.

NP-completeness

Definition (*NP*-hardness)

Problem L is *NP*-hard if every problem from *NP* can be reduced to L.

Lemma

If some NP-hard problem L is in P then $P = NP$.

Definition (*NP*-completeness)

NP-hard problem L is called *NP*-complete iff it is in *P*.

Theorem (Cook's theorem or Cook-Levin theorem)

SAT is NP-complete.

NP-completeness

Definition (NP-hardness)

Problem L is NP-hard if every problem from NP can be reduced to L.

Lemma

If some NP-hard problem L is in P then $P = NP$.

Definition (NP-completeness)

NP-hard problem L is called NP-complete iff it is in P.

Theorem (Cook's theorem or Cook-Levin theorem)

SAT is NP-complete.

Lemma

Given two problems $L, M \in NP$. If L is NP-complete and $L \longrightarrow M$, then M is NP-complete.

NP-completeness

Definition (NP-hardness)

Problem L is NP-hard if every problem from NP can be reduced to L.

Lemma

If some NP-hard problem L is in P then $P = NP$.

Definition (NP-completeness)

NP-hard problem L is called NP-complete iff it is in P.

Theorem (Cook's theorem or Cook-Levin theorem)

SAT is NP-complete.

Lemma

Given two problems $L, M \in NP$. If L is NP-complete and $L \longrightarrow M$, then M is NP-complete.

Examples of NP-complete problems:

1. Logical problems: SAT, 3-SAT, 3,3-SAT, SAT for general formulas (not in CNF), Boolean circuit SAT, . . .

NP-completeness

Definition (NP-hardness)

Problem L is NP-hard if every problem from NP can be reduced to L.

Lemma

If some NP-hard problem L is in P then $P = NP$.

Definition (NP-completeness)

NP-hard problem L is called NP-complete iff it is in P.

Theorem (Cook's theorem or Cook-Levin theorem)

SAT is NP-complete.

Lemma

Given two problems $L, M \in NP$. If L is NP-complete and $L \longrightarrow M$, then M is NP-complete.

Examples of NP-complete problems:

1. Logical problems: SAT, 3-SAT, 3,3-SAT, SAT for general formulas (not in CNF), Boolean circuit SAT, ...
2. Graph problems: IndSet, Clique, graph coloring, Hamiltonian path, Hamiltonian cycle, ...

NP-completeness

Definition (NP-hardness)

Problem L is NP-hard if every problem from NP can be reduced to L.

Lemma

If some NP-hard problem L is in P then $P = NP$.

Definition (NP-completeness)

NP-hard problem L is called NP-complete iff it is in P.

Theorem (Cook's theorem or Cook-Levin theorem)

SAT is NP-complete.

Lemma

Given two problems $L, M \in NP$. If L is NP-complete and $L \longrightarrow M$, then M is NP-complete.

Examples of NP-complete problems:

1. Logical problems: SAT, 3-SAT, 3,3-SAT, SAT for general formulas (not in CNF), Boolean circuit SAT, . . .
2. Graph problems: IndSet, Clique, graph coloring, Hamiltonian path, Hamiltonian cycle, . . .
3. Numerical problems: Finding subset of a given sum, Knapsack, $Ax = 1$, . . .