The class **NP**

K. Subramani[1]

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

March 9 and March 16, 2015

Outline

Outline

Outline

Outline

Outline

Reductions

Reductions

Main concept

Reductions

Main concept

Comparing problem difficulty through $A \leq B$.

Reductions

Main concept

Comparing problem difficulty through $A \leq B$.

When is problem B at least as hard as problem A?

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

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

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).

More on reductions

More on reductions

Definition

More on reductions

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 of L_2, such that

$$(\forall x \in \Sigma_1^*) \;\; x \in L_1 \leftrightarrow R(x) \in L_2.$$

More on reductions

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 of L_2, such that

$$(\forall x \in \Sigma_1^*) \;\; x \in L_1 \leftrightarrow R(x) \in L_2.$$

Furthermore, the function should be computable by an algorithm in $O(\log n)$ space, on strings of length n.

More on reductions

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 of L_2, such that

$$(\forall x \in \Sigma_1^*) \;\; x \in L_1 \leftrightarrow R(x) \in L_2.$$

Furthermore, the function should be computable by an algorithm in $O(\log n)$ space, on strings of length n.

Note

More on reductions

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 of L_2, such that

$$(\forall x \in \Sigma_1^*) \;\; x \in L_1 \leftrightarrow R(x) \in L_2.$$

Furthermore, the function should be computable by an algorithm in $O(\log n)$ space, on strings of length n.

Note

Good old days, we used poly-time reductions.

More on reductions

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 of L_2, such that

$$(\forall x \in \Sigma_1^*) \;\; x \in L_1 \leftrightarrow R(x) \in L_2.$$

Furthermore, the function should be computable by an algorithm in $O(\log n)$ space, on strings of length n.

Note

Good old days, we used poly-time reductions.

Proposition

More on reductions

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 of L_2, such that

$$(\forall x \in \Sigma_1^*) \;\; x \in L_1 \leftrightarrow R(x) \in L_2.$$

Furthermore, the function should be computable by an algorithm in $O(\log n)$ space, on strings of length n.

Note

Good old days, we used poly-time reductions.

Proposition

If R is a reduction computed by an algorithm A, then for all x, A halts after a polynomial number of steps.

Composition of Reductions

Composition of Reductions

Theorem

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.

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.

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.

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 $\log |x|$.

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:

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

Completeness

Definition

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.

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

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

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
$((L \in \mathcal{C}) \wedge (L' \leq L))$

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

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

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.*

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

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 $\mathcal{C} = \mathcal{C}'$.

The class **NP**

The class **NP**

Definition

The class **NP**

Definition

A decision problem is in **NP**, if, whenever the answer for a particular instance is “yes”, there is a simple proof of this fact.

The class **NP**

Definition

A decision problem is in **NP**, if, whenever the answer for a particular instance is “yes”, there is a simple proof of this fact.

Observations

The class **NP**

Definition

A decision problem is in **NP**, if, whenever the answer for a particular instance is “yes”, there is a simple proof of this fact.

Observations

1. How to solve the Hamilton path problem efficiently?

The class **NP**

Definition

A decision problem is in **NP**, if, whenever the answer for a particular instance is “yes”, there is a simple proof of this fact.

Observations

1. How to solve the Hamilton path problem efficiently? Don’t know.

The class **NP**

Definition

A decision problem is in **NP**, if, whenever the answer for a particular instance is “yes”, there is a simple proof of this fact.

Observations

1. How to solve the Hamilton path problem efficiently? Don’t know.
2. Suppose I give you a Hamilton path, can you verify its Hamiltonicity?

The class **NP**

Definition

A decision problem is in **NP**, if, whenever the answer for a particular instance is “yes”, there is a simple proof of this fact.

Observations

1. How to solve the Hamilton path problem efficiently? Don’t know.
2. Suppose I give you a Hamilton path, can you verify its Hamiltonicity?
3. Needle in a haystack analogy.

The class **NP**

Definition

A decision problem is in **NP**, if, whenever the answer for a particular instance is "yes", there is a simple proof of this fact.

Observations

1. How to solve the Hamilton path problem efficiently? Don't know.
2. Suppose I give you a Hamilton path, can you verify its Hamiltonicity?
3. Needle in a haystack analogy.
4. **NP** is profoundly asymmetric.

The class **NP**

Definition

A decision problem is in **NP**, if, whenever the answer for a particular instance is "yes", there is a simple proof of this fact.

Observations

1. How to solve the Hamilton path problem efficiently? Don't know.
2. Suppose I give you a Hamilton path, can you verify its Hamiltonicity?
3. Needle in a haystack analogy.
4. **NP** is profoundly asymmetric.
5. Is **P** $\subseteq$ **NP**?

The class **NP**

Definition

A decision problem is in **NP**, if, whenever the answer for a particular instance is “yes”, there is a simple proof of this fact.

Observations

1. How to solve the Hamilton path problem efficiently? Don’t know.
2. Suppose I give you a Hamilton path, can you verify its Hamiltonicity?
3. Needle in a haystack analogy.
4. **NP** is profoundly asymmetric.
5. Is **P** $\subseteq$ **NP**? What is a short proof for a problem in **P**?

Satisfiability

Satisfiability

SAT

Satisfiability

SAT

1. A boolean variable is a variable that assumes the values **true** or **false**.

Satisfiability

SAT

1. A boolean variable is a variable that assumes the values **true** or **false**.
2. The complement of a boolean variable x is denoted by $\bar{x}$ and assumes the value **true** if and only if the variable assumes **false**.

Satisfiability

SAT

1. A boolean variable is a variable that assumes the values **true** or **false**.
2. The complement of a boolean variable x is denoted by $\bar{x}$ and assumes the value **true** if and only if the variable assumes **false**.
3. A literal is a boolean variable or its complement.

Satisfiability

SAT

1. A boolean variable is a variable that assumes the values **true** or **false**.
2. The complement of a boolean variable x is denoted by $\bar{x}$ and assumes the value **true** if and only if the variable assumes **false**.
3. A literal is a boolean variable or its complement.
4. A clause is a disjunction of literals.

Satisfiability

SAT

1. A boolean variable is a variable that assumes the values **true** or **false**.
2. The complement of a boolean variable x is denoted by $\bar{x}$ and assumes the value **true** if and only if the variable assumes **false**.
3. A literal is a boolean variable or its complement.
4. A clause is a disjunction of literals.
5. A boolean formula is said to be in Conjunctive Normal Form (CNF), if it is a conjunction of clauses.

Satisfiability

SAT

1. A boolean variable is a variable that assumes the values **true** or **false**.
2. The complement of a boolean variable x is denoted by $\bar{x}$ and assumes the value **true** if and only if the variable assumes **false**.
3. A literal is a boolean variable or its complement.
4. A clause is a disjunction of literals.
5. A boolean formula is said to be in Conjunctive Normal Form (CNF), if it is a conjunction of clauses.
6. An assignment is a consistent mapping of the literals of a formula to **true**/**false**.

Satisfiability

SAT

1. A boolean variable is a variable that assumes the values **true** or **false**.
2. The complement of a boolean variable x is denoted by $\bar{x}$ and assumes the value **true** if and only if the variable assumes **false**.
3. A literal is a boolean variable or its complement.
4. A clause is a disjunction of literals.
5. A boolean formula is said to be in Conjunctive Normal Form (CNF), if it is a conjunction of clauses.
6. An assignment is a consistent mapping of the literals of a formula to **true**/**false**.
7. A formula is said to be satisfiable, if it has a satisfying assignment.

Satisfiability

SAT

1. A boolean variable is a variable that assumes the values **true** or **false**.
2. The complement of a boolean variable x is denoted by $\bar{x}$ and assumes the value **true** if and only if the variable assumes **false**.
3. A literal is a boolean variable or its complement.
4. A clause is a disjunction of literals.
5. A boolean formula is said to be in Conjunctive Normal Form (CNF), if it is a conjunction of clauses.
6. An assignment is a consistent mapping of the literals of a formula to **true**/**false**.
7. A formula is said to be satisfiable, if it has a satisfying assignment.

Definition

Satisfiability

SAT

1. A boolean variable is a variable that assumes the values **true** or **false**.
2. The complement of a boolean variable x is denoted by $\bar{x}$ and assumes the value **true** if and only if the variable assumes **false**.
3. A literal is a boolean variable or its complement.
4. A clause is a disjunction of literals.
5. A boolean formula is said to be in Conjunctive Normal Form (CNF), if it is a conjunction of clauses.
6. An assignment is a consistent mapping of the literals of a formula to **true/false**.
7. A formula is said to be satisfiable, if it has a satisfying assignment.

Definition

Given a CNF formula $\phi = C_1 \wedge C_2 \ldots C_m$ over the n boolean variables $\{x_1, x_2, \ldots x_n\}$ and their complements, the satisfiability problem (or SAT) asks if ϕ is satisfiable.

Satisfiability

SAT

1. A boolean variable is a variable that assumes the values **true** or **false**.
2. The complement of a boolean variable x is denoted by $\bar{x}$ and assumes the value **true** if and only if the variable assumes **false**.
3. A literal is a boolean variable or its complement.
4. A clause is a disjunction of literals.
5. A boolean formula is said to be in Conjunctive Normal Form (CNF), if it is a conjunction of clauses.
6. An assignment is a consistent mapping of the literals of a formula to **true/false**.
7. A formula is said to be satisfiable, if it has a satisfying assignment.

Definition

Given a CNF formula $\phi = C_1 \wedge C_2 \ldots C_m$ over the n boolean variables $\{x_1, x_2, \ldots x_n\}$ and their complements, the satisfiability problem (or SAT) asks if ϕ is satisfiable.

kSAT is the variant of SAT in which each clause has exactly k variables.

Variants of SAT

Variants of SAT

Exercise

Variants of SAT

Exercise

1. *Show that 1SAT is in* **P**.

Variants of SAT

Exercise

1. *Show that* 1*SAT is in* **P**.
2. *Show that the formula* $(p \vee \bar{q}) \wedge (\bar{p} \vee \bar{r}) \wedge (q \vee r) \wedge (p \vee q) \wedge (\bar{q} \vee r)$ *is unsatisfiable.*

Variants of SAT

Exercise

1. *Show that* 1*SAT is in* **P**.
2. *Show that the formula* $(p \vee \bar{q}) \wedge (\bar{p} \vee \bar{r}) \wedge (q \vee r) \wedge (p \vee q) \wedge (\bar{q} \vee r)$ *is unsatisfiable.*
3. *A CNF formula is said to be Horn, if each clause has at most one positive literal.*

Variants of SAT

Exercise

1. *Show that* 1*SAT is in* **P**.
2. *Show that the formula* $(p \vee \bar{q}) \wedge (\bar{p} \vee \bar{r}) \wedge (q \vee r) \wedge (p \vee q) \wedge (\bar{q} \vee r)$ *is unsatisfiable.*
3. *A CNF formula is said to be Horn, if each clause has at most one positive literal. Argue that HornSAT is in* **P**.

Variants of SAT

Exercise

1. *Show that* $1SAT$ *is in* **P**.
2. *Show that the formula* $(p \vee \bar{q}) \wedge (\bar{p} \vee \bar{r}) \wedge (q \vee r) \wedge (p \vee q) \wedge (\bar{q} \vee r)$ *is unsatisfiable.*
3. *A CNF formula is said to be Horn, if each clause has at most one positive literal. Argue that HornSAT is in* **P**.

Theorem

Variants of SAT

Exercise

1. *Show that* 1*SAT is in* **P**.
2. *Show that the formula* $(p \vee \bar{q}) \wedge (\bar{p} \vee \bar{r}) \wedge (q \vee r) \wedge (p \vee q) \wedge (\bar{q} \vee r)$ *is unsatisfiable.*
3. *A CNF formula is said to be Horn, if each clause has at most one positive literal. Argue that HornSAT is in* **P**.

Theorem

2*SAT is in* **P**.

Variants of SAT

Exercise

1. *Show that* 1*SAT is in* **P***.*
2. *Show that the formula* $(p \vee \bar{q}) \wedge (\bar{p} \vee \bar{r}) \wedge (q \vee r) \wedge (p \vee q) \wedge (\bar{q} \vee r)$ *is unsatisfiable.*
3. *A CNF formula is said to be Horn, if each clause has at most one positive literal. Argue that HornSAT is in* **P***.*

Theorem

2*SAT is in* **P***.*

Observation

Variants of SAT

Exercise

1. *Show that* 1*SAT is in* **P***.*
2. *Show that the formula* $(p \vee \bar{q}) \wedge (\bar{p} \vee \bar{r}) \wedge (q \vee r) \wedge (p \vee q) \wedge (\bar{q} \vee r)$ *is unsatisfiable.*
3. *A CNF formula is said to be Horn, if each clause has at most one positive literal. Argue that HornSAT is in* **P***.*

Theorem

2*SAT is in* **P***.*

Observation

Variants of SAT

Exercise

1. *Show that* 1*SAT is in* **P**.
2. *Show that the formula* $(p \vee \bar{q}) \wedge (\bar{p} \vee \bar{r}) \wedge (q \vee r) \wedge (p \vee q) \wedge (\bar{q} \vee r)$ *is unsatisfiable.*
3. *A CNF formula is said to be Horn, if each clause has at most one positive literal. Argue that HornSAT is in* **P**.

Theorem

2*SAT is in* **P**.

Observation

$$(a \vee b) \quad \Leftrightarrow \quad (\bar{a} \to b)$$

Variants of SAT

Exercise

1. *Show that* 1*SAT is in* **P**.
2. *Show that the formula* $(p \vee \bar{q}) \wedge (\bar{p} \vee \bar{r}) \wedge (q \vee r) \wedge (p \vee q) \wedge (\bar{q} \vee r)$ *is unsatisfiable.*
3. *A CNF formula is said to be Horn, if each clause has at most one positive literal. Argue that HornSAT is in* **P**.

Theorem

2*SAT is in* **P**.

Observation

$$(a \vee b) \quad \Leftrightarrow \quad (\bar{a} \rightarrow b) \wedge (\bar{b} \rightarrow a)$$

Implication Graph

Implication Graph

From constraints to Digraphs

Implication Graph

From constraints to Digraphs

The implication graph $G(\phi)$ corresponding to the formula ϕ is created as follows:

Implication Graph

From constraints to Digraphs

The implication graph $G(\phi)$ corresponding to the formula ϕ is created as follows:

1. Create one vertex for each literal; the vertex is labeled with the literal.

Implication Graph

From constraints to Digraphs

The implication graph $G(\phi)$ corresponding to the formula ϕ is created as follows:

1. Create one vertex for each literal; the vertex is labeled with the literal.
2. Corresponding to the clause $(x_i \vee x_j)$ draw a directed arc from $\bar{x}_i$ to x_j and another directed arc from $\bar{x}_j$ to x_i.

Implication Graph

From constraints to Digraphs

The implication graph $G(\phi)$ corresponding to the formula ϕ is created as follows:

1. Create one vertex for each literal; the vertex is labeled with the literal.
2. Corresponding to the clause $(x_i \vee x_j)$ draw a directed arc from $\bar{x}_i$ to x_j and another directed arc from $\bar{x}_j$ to x_i.
3. The resultant graph is called the implication graph corresponding to the given 2CNF formula.

Some observations

Some observations

Observations

Some observations

Observations

1. If there is a path from literal a to literal b in $G(\phi)$, then there is also a path from $\bar{b}$ to $\bar{a}$.

Some observations

Observations

1. If there is a path from literal a to literal b in $G(\phi)$, then there is also a path from $\bar{b}$ to $\bar{a}$.
2. Any assignment which leads to a path from **true** to **false** is **not** a satisfying assignment.

Some observations

Observations

1. If there is a path from literal a to literal b in $G(\phi)$, then there is also a path from $\bar{b}$ to $\bar{a}$.
2. Any assignment which leads to a path from **true** to **false** is **not** a satisfying assignment.
3. If there is a path from x_i to $\bar{x}_i$, then x_i cannot be assigned **true** in a satisfying assignment.

Some observations

Observations

1. If there is a path from literal a to literal b in $G(\phi)$, then there is also a path from $\bar{b}$ to $\bar{a}$.
2. Any assignment which leads to a path from **true** to **false** is **not** a satisfying assignment.
3. If there is a path from x_i to $\bar{x}_i$, then x_i cannot be assigned **true** in a satisfying assignment.
4. If there is a path from $\bar{x}_i$ to x_i, then x_i cannot be assigned **false** in a satisfying assignment.

Reachability Lemmata

Reachability Lemmata

Lemma

Reachability Lemmata

Lemma

If there is a variable x in $G(\phi)$ such that x is reachable from $\bar{x}$ and vice versa,

Reachability Lemmata

Lemma

If there is a variable x in $G(\phi)$ such that x is reachable from $\bar{x}$ and vice versa, then ϕ is unsatisfiable.

Reachability Lemmata

Lemma

If there is a variable x in $G(\phi)$ such that x is reachable from $\bar{x}$ and vice versa, then ϕ is unsatisfiable.

Lemma

If there is no variable x such that x is reachable from $\bar{x}$ and vice versa,

Reachability Lemmata

Lemma

If there is a variable x in $G(\phi)$ such that x is reachable from $\bar{x}$ and vice versa, then ϕ is unsatisfiable.

Lemma

If there is no variable x such that x is reachable from $\bar{x}$ and vice versa, then ϕ is satisfiable.

Reachability Lemmata

Lemma

If there is a variable x in $G(\phi)$ such that x is reachable from $\bar{x}$ and vice versa, then ϕ is unsatisfiable.

Lemma

If there is no variable x such that x is reachable from $\bar{x}$ and vice versa, then ϕ is satisfiable.

Proof.

Reachability Lemmata

Lemma

If there is a variable x in $G(\phi)$ such that x is reachable from $\bar{x}$ and vice versa, then ϕ is unsatisfiable.

Lemma

If there is no variable x such that x is reachable from $\bar{x}$ and vice versa, then ϕ is satisfiable.

Proof.

1. Assume that x is set to **true**, which means that there is no path from x to $\bar{x}$.

Reachability Lemmata

Lemma

If there is a variable x in $G(\phi)$ such that x is reachable from $\bar{x}$ and vice versa, then ϕ is unsatisfiable.

Lemma

If there is no variable x such that x is reachable from $\bar{x}$ and vice versa, then ϕ is satisfiable.

Proof.

1. Assume that x is set to **true**, which means that there is no path from x to $\bar{x}$.
2. A contradiction occurs only if $x \rightsquigarrow y$ and $x \rightsquigarrow \bar{y}$ for some variable y.

Reachability Lemmata

Lemma

If there is a variable x in $G(\phi)$ such that x is reachable from $\bar{x}$ and vice versa, then ϕ is unsatisfiable.

Lemma

If there is no variable x such that x is reachable from $\bar{x}$ and vice versa, then ϕ is satisfiable.

Proof.

1. Assume that x is set to **true**, which means that there is no path from x to $\bar{x}$.
2. A contradiction occurs only if $x \leadsto y$ and $x \leadsto \bar{y}$ for some variable y.
3. By the symmetry of $G(\phi)$, there must be paths $\bar{y} \leadsto \bar{x}$ and $y \leadsto \bar{x}$.

Reachability Lemmata

Lemma

If there is a variable x in $G(\phi)$ such that x is reachable from $\bar{x}$ and vice versa, then ϕ is unsatisfiable.

Lemma

If there is no variable x such that x is reachable from $\bar{x}$ and vice versa, then ϕ is satisfiable.

Proof.

1. Assume that x is set to **true**, which means that there is no path from x to $\bar{x}$.
2. A contradiction occurs only if $x \rightsquigarrow y$ and $x \rightsquigarrow \bar{y}$ for some variable y.
3. By the symmetry of $G(\phi)$, there must be paths $\bar{y} \rightsquigarrow \bar{x}$ and $y \rightsquigarrow \bar{x}$.
4. This means that there is a path $x \rightsquigarrow \bar{x}$,

Reachability Lemmata

Lemma

If there is a variable x in $G(\phi)$ such that x is reachable from $\bar{x}$ and vice versa, then ϕ is unsatisfiable.

Lemma

If there is no variable x such that x is reachable from $\bar{x}$ and vice versa, then ϕ is satisfiable.

Proof.

1. Assume that x is set to **true**, which means that there is no path from x to $\bar{x}$.
2. A contradiction occurs only if $x \rightsquigarrow y$ and $x \rightsquigarrow \bar{y}$ for some variable y.
3. By the symmetry of $G(\phi)$, there must be paths $\bar{y} \rightsquigarrow \bar{x}$ and $y \rightsquigarrow \bar{x}$.
4. This means that there is a path $x \rightsquigarrow \bar{x}$, i.e., a contradiction.

Reachability Lemmata

Lemma

If there is a variable x in $G(\phi)$ such that x is reachable from $\bar{x}$ and vice versa, then ϕ is unsatisfiable.

Lemma

If there is no variable x such that x is reachable from $\bar{x}$ and vice versa, then ϕ is satisfiable.

Proof.

1. Assume that x is set to **true**, which means that there is no path from x to $\bar{x}$.
2. A contradiction occurs only if $x \rightsquigarrow y$ and $x \rightsquigarrow \bar{y}$ for some variable y.
3. By the symmetry of $G(\phi)$, there must be paths $\bar{y} \rightsquigarrow \bar{x}$ and $y \rightsquigarrow \bar{x}$.
4. This means that there is a path $x \rightsquigarrow \bar{x}$, i.e., a contradiction.

The case where x is set to **false** can be handled similarly. □

The 2SAT Algorithm

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

1: **for** (each variable x) **do**

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

1: **for** (each variable x) **do**
2: **if** ($x \rightsquigarrow \bar{x}$)

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

for (each variable x) **do**
  if $(x \rightsquigarrow \bar{x})$ **and** $(\bar{x} \rightsquigarrow x)$ **then**

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

```
for (each variable x) do
    if (x ⇝ x̄) and (x̄ ⇝ x) then
        return(false).
```

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

for (each variable x) **do**
    if $(x \rightsquigarrow \bar{x})$ **and** $(\bar{x} \rightsquigarrow x)$ **then**
        return(**false**).
    end if

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

```
for (each variable x) do
    if (x ⇝ x̄) and (x̄ ⇝ x) then
        return(false).
    end if
end for
```

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

```
for (each variable x) do
    if (x ⇝ x̄) and (x̄ ⇝ x) then
        return(false).
    end if
end for
for (each variable x) do
```

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

for (each variable x) **do**
  if ($x \rightsquigarrow \bar{x}$) **and** ($\bar{x} \rightsquigarrow x$) **then**
    return(**false**).
  end if
end for
for (each variable x) **do**
  if ($x \rightsquigarrow \bar{x}$) **then**

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

for (each variable x) **do**
    if $(x \rightsquigarrow \bar{x})$ **and** $(\bar{x} \rightsquigarrow x)$ **then**
        return(**false**).
    end if
end for
for (each variable x) **do**
    if $(x \rightsquigarrow \bar{x})$ **then**
        $x =$ **false**.

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

for (each variable x) **do**
    if ($x \rightsquigarrow \bar{x}$) **and** ($\bar{x} \rightsquigarrow x$) **then**
        return(**false**).
    end if
end for
for (each variable x) **do**
    if ($x \rightsquigarrow \bar{x}$) **then**
        $x =$ **false**.
    else

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

```
for (each variable x) do
    if (x ⇝ x̄) and (x̄ ⇝ x) then
        return(false).
    end if
end for
for (each variable x) do
    if (x ⇝ x̄) then
        x = false.
    else
        if (x̄ ⇝ x) then
```

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

for (each variable x) **do**
    if $(x \rightsquigarrow \bar{x})$ **and** $(\bar{x} \rightsquigarrow x)$ **then**
        return(**false**).
    end if
end for
for (each variable x) **do**
    if $(x \rightsquigarrow \bar{x})$ **then**
        $x =$ **false**.
    else
        if $(\bar{x} \rightsquigarrow x)$ **then**
            $x =$ **true**.

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

for (each variable x) **do**
    if $(x \rightsquigarrow \bar{x})$ **and** $(\bar{x} \rightsquigarrow x)$ **then**
        return(**false**).
    end if
end for
for (each variable x) **do**
    if $(x \rightsquigarrow \bar{x})$ **then**
        $x =$ **false**.
    else
        if $(\bar{x} \rightsquigarrow x)$ **then**
            $x =$ **true**.
        else

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

for (each variable x) **do**
    if ($x \rightsquigarrow \bar{x}$) **and** ($\bar{x} \rightsquigarrow x$) **then**
        return(**false**).
    end if
end for
for (each variable x) **do**
    if ($x \rightsquigarrow \bar{x}$) **then**
        $x =$ **false**.
    else
        if ($\bar{x} \rightsquigarrow x$) **then**
            $x =$ **true**.
        else
            Set x to **true**

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

for (each variable x) **do**
  if $(x \rightsquigarrow \bar{x})$ **and** $(\bar{x} \rightsquigarrow x)$ **then**
    return(**false**).
  end if
end for
for (each variable x) **do**
  if $(x \rightsquigarrow \bar{x})$ **then**
    $x =$ **false**.
  else
    if $(\bar{x} \rightsquigarrow x)$ **then**
      $x =$ **true**.
    else
      Set x to **true** or **false**.

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

for (each variable x) **do**
    if $(x \rightsquigarrow \bar{x})$ **and** $(\bar{x} \rightsquigarrow x)$ **then**
        return(**false**).
    end if
end for
for (each variable x) **do**
    if $(x \rightsquigarrow \bar{x})$ **then**
        $x =$ **false**.
    else
        if $(\bar{x} \rightsquigarrow x)$ **then**
            $x =$ **true**.
        else
            Set x to **true** or **false**.
        end if

The 2SAT Algorithm

```
FUNCTION 2SAT-ALGORITHM(G(φ))
for (each variable x) do
    if (x ⇝ x̄) and (x̄ ⇝ x) then
        return(false).
    end if
end for
for (each variable x) do
    if (x ⇝ x̄) then
        x = false.
    else
        if (x̄ ⇝ x) then
            x = true.
        else
            Set x to true or false.
        end if
    end if
```

The 2SAT Algorithm

FUNCTION 2SAT-ALGORITHM($G(\phi)$)

for (each variable x) **do**
    if $(x \rightsquigarrow \bar{x})$ **and** $(\bar{x} \rightsquigarrow x)$ **then**
        return(**false**).
    end if
end for
for (each variable x) **do**
    if $(x \rightsquigarrow \bar{x})$ **then**
        $x =$ **false**.
    else
        if $(\bar{x} \rightsquigarrow x)$ **then**
            $x =$ **true**.
        else
            Set x to **true** or **false**.
        end if
    end if
end for

Algorithm 4.20: 2CNF satisfiability through Reachability

Analysis

Analysis

Exercise

What is the running time of the above algorithm?

Reducing Hamilton Path to SAT

Reducing Hamilton Path to SAT

Hamilton Path to SAT

Reducing Hamilton Path to SAT

Hamilton Path to SAT

Input instance: An unweighted, directed graph G.

Reducing Hamilton Path to SAT

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.

Reducing Hamilton Path to SAT

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.

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).

Reducing Hamilton Path to SAT

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.

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).
2. $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n. \quad [C_1]$.

Reducing Hamilton Path to SAT

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.

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

Reducing Hamilton Path to SAT

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.

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).
2. $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n.$ $[C_1]$.
3. $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i.$ $[C_2]$.

Reducing Hamilton Path to SAT

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.

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).
2. $(x_{1j} \vee x_{2j} \ldots x_{nj})$, $j = 1, 2, \ldots, n$. $[C_1]$.
3. $(\neg x_{ij} \vee \neg x_{kj})$, $j = 1, 2 \ldots n$, $i = 1, 2, \ldots, n$, $k = 1, 2, \ldots n$, $k \neq i$. $[C_2]$.
4. $(x_{i1} \vee x_{i2} \ldots \vee x_{in})$, $i = 1, 2 \ldots n$. $[C_3]$.

Reducing Hamilton Path to SAT

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.

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).
2. $(x_{1j} \vee x_{2j} \ldots x_{nj}), j = 1, 2, \ldots, n.$ $[C_1]$.
3. $(\neg x_{ij} \vee \neg x_{kj}), j = 1, 2 \ldots n, i = 1, 2, \ldots, n, k = 1, 2, \ldots n, k \neq i.$ $[C_2]$.
4. $(x_{i1} \vee x_{i2} \ldots \vee x_{in}), i = 1, 2 \ldots n.$ $[C_3]$.
5. $(\neg x_{ij} \vee \neg x_{ik}), i = 1, 2, \ldots, n, j, k = 1, 2, \ldots, n, j \neq k.$ $[C_4]$.

Reducing Hamilton Path to SAT

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.

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

Reducing Hamilton Path to SAT

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.

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

Completing the argument

Completing the argument

Satisfiability implies Hamilton Path

Completing the argument

Satisfiability implies Hamilton Path

Completing the argument

Satisfiability implies Hamilton Path

Let T denote a satisfying assignment to ϕ.

Completing the argument

Satisfiability implies Hamilton Path

Let T denote a satisfying assignment to ϕ.

We show that there must exist a Hamilton Path in G.

Completing the argument

Satisfiability implies Hamilton Path

Let T denote a satisfying assignment to ϕ.

We show that there must exist a Hamilton Path in G.

1. For each j, there is exactly one i, such that x_{ij} is **true** under T.

Completing the argument

Satisfiability implies Hamilton Path

Let T denote a satisfying assignment to ϕ.

We show that there must exist a Hamilton Path in G.

1. For each j, there is exactly one i, such that x_{ij} is **true** under T. (Why?)

Completing the argument

Satisfiability implies Hamilton Path

Let T denote a satisfying assignment to ϕ.

We show that there must exist a Hamilton Path in G.

1. For each j, there is exactly one i, such that x_{ij} is **true** under T. (Why?)
2. For each i, there is exactly one j, such that x_{ij} is **true** under T.

Completing the argument

Satisfiability implies Hamilton Path

Let T denote a satisfying assignment to ϕ.

We show that there must exist a Hamilton Path in G.

1. For each j, there is exactly one i, such that x_{ij} is **true** under T. (Why?)
2. For each i, there is exactly one j, such that x_{ij} is **true** under T. (Why?)

Completing the argument

Satisfiability implies Hamilton Path

Let T denote a satisfying assignment to ϕ.

We show that there must exist a Hamilton Path in G.

1. For each j, there is exactly one i, such that x_{ij} is **true** under T. (Why?)
2. For each i, there is exactly one j, such that x_{ij} is **true** under T. (Why?)
3. T is thus a permutation of the nodes $(\pi(1), \pi(2), \ldots, \pi(n))$, such that $\pi(i) = j$ if and only if x_{ij} is set to **true** under T.

Completing the argument

Satisfiability implies Hamilton Path

Let T denote a satisfying assignment to ϕ.

We show that there must exist a Hamilton Path in G.

1. For each j, there is exactly one i, such that x_{ij} is **true** under T. (Why?)
2. For each i, there is exactly one j, such that x_{ij} is **true** under T. (Why?)
3. T is thus a permutation of the nodes $(\pi(1), \pi(2), \ldots, \pi(n))$, such that $\pi(i) = j$ if and only if x_{ij} is set to **true** under T.
4. The clause system $[C_6]$ guarantees that adjacent elements on the permutation are connected by an edge in G.

Completing the argument

Satisfiability implies Hamilton Path

Let T denote a satisfying assignment to ϕ.

We show that there must exist a Hamilton Path in G.

1. For each j, there is exactly one i, such that x_{ij} is **true** under T. (Why?)
2. For each i, there is exactly one j, such that x_{ij} is **true** under T. (Why?)
3. T is thus a permutation of the nodes $(\pi(1), \pi(2), \ldots, \pi(n))$, such that $\pi(i) = j$ if and only if x_{ij} is set to **true** under T.
4. The clause system $[C_6]$ guarantees that adjacent elements on the permutation are connected by an edge in G.
5. It follows that G has a Hamilton path.

Completing the argument (contd.)

Completing the argument (contd.)

Hamilton Path implies Satisfiability

Assume that the graph G has a Hamilton path p.

Completing the argument (contd.)

Hamilton Path implies Satisfiability

Assume that the graph G has a Hamilton path p.

We show that ϕ is satisfiable.

Completing the argument (contd.)

Hamilton Path implies Satisfiability

Assume that the graph G has a Hamilton path p.

We show that ϕ is satisfiable. Observe that,

Completing the argument (contd.)

Hamilton Path implies Satisfiability

Assume that the graph G has a Hamilton path p.

We show that ϕ is satisfiable. Observe that,

1. Observe that p can be represented as a permutation $\pi = (\pi(1), \pi(2) \ldots \pi(n))$, where $\pi(i)$ represents the i^{th} vertex on the Hamilton path.

Completing the argument (contd.)

Hamilton Path implies Satisfiability

Assume that the graph G has a Hamilton path p.

We show that ϕ is satisfiable. Observe that,

1. Observe that p can be represented as a permutation $\pi = (\pi(1), \pi(2) \ldots \pi(n))$, where $\pi(i)$ represents the i^{th} vertex on the Hamilton path.
2. Consider the following assignment: $T(x_{ij}) =$ **true** if and only if $\pi(i) = j$.

Completing the argument (contd.)

Hamilton Path implies Satisfiability

Assume that the graph G has a Hamilton path p.

We show that ϕ is satisfiable. Observe that,

1. Observe that p can be represented as a permutation $\pi = (\pi(1), \pi(2) \ldots \pi(n))$, where $\pi(i)$ represents the i^{th} vertex on the Hamilton path.
2. Consider the following assignment: $T(x_{ij}) =$ **true** if and only if $\pi(i) = j$.
3. It is not hard to see that every clause in ϕ is satisfied.

Completing the argument (contd.)

Hamilton Path implies Satisfiability

Assume that the graph G has a Hamilton path p.

We show that ϕ is satisfiable. Observe that,

1. Observe that p can be represented as a permutation $\pi = (\pi(1), \pi(2) \ldots \pi(n))$, where $\pi(i)$ represents the i^{th} vertex on the Hamilton path.
2. Consider the following assignment: $T(x_{ij}) =$ **true** if and only if $\pi(i) = j$.
3. It is not hard to see that every clause in ϕ is satisfied.

Final Step

Completing the argument (contd.)

Hamilton Path implies Satisfiability

Assume that the graph G has a Hamilton path p.

We show that ϕ is satisfiable. Observe that,

1. Observe that p can be represented as a permutation $\pi = (\pi(1), \pi(2) \ldots \pi(n))$, where $\pi(i)$ represents the i^{th} vertex on the Hamilton path.
2. Consider the following assignment: $T(x_{ij}) =$ **true** if and only if $\pi(i) = j$.
3. It is not hard to see that every clause in ϕ is satisfied.

Final Step

Is the reduction polynomial in the size of the input?

Boolean Circuits (Syntax)

Boolean Circuits (Syntax)

Syntax

Boolean Circuits (Syntax)

Syntax

1. A boolean circuit C is a DAG $G = \langle V, E \rangle$.

Boolean Circuits (Syntax)

Syntax

1. A boolean circuit C is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots n\}$ are called the gates of C.

Boolean Circuits (Syntax)

Syntax

1. A boolean circuit C is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots n\}$ are called the gates of C.
3. We can assume without loss of generality that the edges are of the form (i, j), where $i < j$.

Boolean Circuits (Syntax)

Syntax

1. A boolean circuit C is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots n\}$ are called the gates of C.
3. We can assume without loss of generality that the edges are of the form (i, j), where $i < j$.
4. Each gate i has a sort $s(i)$ associated with it, where $s(i) \in \{\textbf{true}, \textbf{false}\} \cup \{x_1, x_2, \ldots\} \cup \{\vee, \wedge, \neg\}$.

Boolean Circuits (Syntax)

Syntax

1. A boolean circuit C is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots n\}$ are called the gates of C.
3. We can assume without loss of generality that the edges are of the form (i, j), where $i < j$.
4. Each gate i has a sort $s(i)$ associated with it, where $s(i) \in \{\textbf{true}, \textbf{false}\} \cup \{x_1, x_2, \ldots\} \cup \{\vee, \wedge, \neg\}$.
5. If $s(i) \in \{\textbf{true}, \textbf{false}\} \cup \{x_1, x_2, \ldots\}$, then its in-degree is 0.

Boolean Circuits (Syntax)

Syntax

1. A boolean circuit C is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots n\}$ are called the gates of C.
3. We can assume without loss of generality that the edges are of the form (i, j), where $i < j$.
4. Each gate i has a sort $s(i)$ associated with it, where $s(i) \in \{\textbf{true}, \ \textbf{false}\} \cup \{x_1, x_2, \ldots\} \cup \{\vee, \wedge, \neg\}$.
5. If $s(i) \in \{\textbf{true}, \ \textbf{false}\} \cup \{x_1, x_2, \ldots\}$, then its in-degree is 0.
6. If $s(i) \in \{\neg\}$, its in-degree is 1.

Boolean Circuits (Syntax)

Syntax

1. A boolean circuit C is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots n\}$ are called the gates of C.
3. We can assume without loss of generality that the edges are of the form (i, j), where $i < j$.
4. Each gate i has a sort $s(i)$ associated with it, where $s(i) \in \{\textbf{true}, \textbf{false}\} \cup \{x_1, x_2, \ldots\} \cup \{\vee, \wedge, \neg\}$.
5. If $s(i) \in \{\textbf{true}, \textbf{false}\} \cup \{x_1, x_2, \ldots\}$, then its in-degree is 0.
6. If $s(i) \in \{\neg\}$, its in-degree is 1.
7. All other gates have in-degree 2.

Boolean Circuits (Syntax)

Syntax

1. A boolean circuit C is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots n\}$ are called the gates of C.
3. We can assume without loss of generality that the edges are of the form (i, j), where $i < j$.
4. Each gate i has a sort $s(i)$ associated with it, where $s(i) \in \{\textbf{true}, \textbf{false}\} \cup \{x_1, x_2, \ldots\} \cup \{\vee, \wedge, \neg\}$.
5. If $s(i) \in \{\textbf{true}, \textbf{false}\} \cup \{x_1, x_2, \ldots\}$, then its in-degree is 0.
6. If $s(i) \in \{\neg\}$, its in-degree is 1.
7. All other gates have in-degree 2.
8. All gates except gate n have out-degree 1.

Boolean Circuits (Syntax)

Syntax

1. A boolean circuit C is a DAG $G = \langle V, E \rangle$.
2. The nodes $V = \{1, 2, \ldots n\}$ are called the gates of C.
3. We can assume without loss of generality that the edges are of the form (i, j), where $i < j$.
4. Each gate i has a sort $s(i)$ associated with it, where $s(i) \in \{\textbf{true}, \textbf{false}\} \cup \{x_1, x_2, \ldots\} \cup \{\vee, \wedge, \neg\}$.
5. If $s(i) \in \{\textbf{true}, \textbf{false}\} \cup \{x_1, x_2, \ldots\}$, then its in-degree is 0.
6. If $s(i) \in \{\neg\}$, its in-degree is 1.
7. All other gates have in-degree 2.
8. All gates except gate n have out-degree 1.
9. Gate n, is called the output gate and has out-degree 0.

Boolean Circuits (Semantics)

Boolean Circuits (Semantics)

Semantics

Boolean Circuits (Semantics)

Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

Boolean Circuits (Semantics)

Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

Boolean Circuits (Semantics)

Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is **true** or **false**, then it retains that value.

Boolean Circuits (Semantics)

Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is **true** or **false**, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.

Boolean Circuits (Semantics)

Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is **true** or **false**, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
3. If the gate has sort $\neg$, then its value is the complement of its input.

Boolean Circuits (Semantics)

Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is **true** or **false**, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
3. If the gate has sort $\neg$, then its value is the complement of its input.
4. If the gate has sort $\vee$, then its value is **true** if at least one of its two input gates has value **true** and is **false** otherwise.

Boolean Circuits (Semantics)

Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is **true** or **false**, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
3. If the gate has sort $\neg$, then its value is the complement of its input.
4. If the gate has sort $\vee$, then its value is **true** if at least one of its two input gates has value **true** and is **false** otherwise.
5. If the gate has sort $\wedge$, then its value is **true** if both its two input gates have value **true** and is **false** otherwise.

Boolean Circuits (Semantics)

Semantics

The semantics of circuits specifies a truth value for the circuit, corresponding to each appropriate assignment.

This value can be computed inductively as follows:

1. If the gate is **true** or **false**, then it retains that value.
2. If the gate is a variable, then its value is equal to its assignment.
3. If the gate has sort $\neg$, then its value is the complement of its input.
4. If the gate has sort $\vee$, then its value is **true** if at least one of its two input gates has value **true** and is **false** otherwise.
5. If the gate has sort $\wedge$, then its value is **true** if both its two input gates have value **true** and is **false** otherwise.
6. The value of the circuit is the value of the output gate.

CIRCUIT-SAT and CIRCUIT-VALUE

CIRCUIT-SAT and CIRCUIT-VALUE

Circuit-SAT

Circuit-SAT and Circuit-Value

Circuit-SAT

Given a circuit C, is there an assignment **true**/**false** to the variable gates, so that C evaluates to **true**?

CIRCUIT-SAT and CIRCUIT-VALUE

Circuit-SAT

Given a circuit C, is there an assignment **true**/**false** to the variable gates, so that C evaluates to **true**?

Circuit-Value

CIRCUIT-SAT and CIRCUIT-VALUE

Circuit-SAT

Given a circuit C, is there an assignment **true**/**false** to the variable gates, so that C evaluates to **true**?

Circuit-Value

Given a variable-free circuit C, does it evaluate to **true**?

CIRCUIT-SAT and CIRCUIT-VALUE

Circuit-SAT

Given a circuit C, is there an assignment **true**/**false** to the variable gates, so that C evaluates to **true**?

Circuit-Value

Given a variable-free circuit C, does it evaluate to **true**?

Exercise

Circuit-SAT and Circuit-Value

Circuit-SAT

Given a circuit C, is there an assignment **true**/**false** to the variable gates, so that C evaluates to **true**?

Circuit-Value

Given a variable-free circuit C, does it evaluate to **true**?

Exercise

Argue that Circuit-Value *is in* **P**.

Reduction from CIRCUIT-SAT to SAT

CIRCUIT-SAT to SAT

Reduction from Circuit-SAT to SAT

Circuit-SAT to SAT

Input instance: A circuit C.

Reduction from CIRCUIT-SAT to SAT

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.

Reduction from CIRCUIT-SAT to SAT

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.

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.

Reduction from CIRCUIT-SAT to SAT

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.

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.
2. If g is a variable gate, corresponding to variable x, add the clauses $(g \vee \neg x)$ and $(\neg g \vee x)$ to ϕ.

Reduction from CIRCUIT-SAT to SAT

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.

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.
2. If g is a variable gate, corresponding to variable x, add the clauses $(g \vee \neg x)$ and $(\neg g \vee x)$ to ϕ. $(g \Leftrightarrow x.)$

Reduction from CIRCUIT-SAT to SAT

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.

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.
2. If g is a variable gate, corresponding to variable x, add the clauses $(g \vee \neg x)$ and $(\neg g \vee x)$ to ϕ. ($g \Leftrightarrow x$.)
3. If g is a **true** gate, add (g) to ϕ; likewise, if it is a **false** gate, add $(\neg g)$.

Reduction from CIRCUIT-SAT to SAT

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.

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.
2. If g is a variable gate, corresponding to variable x, add the clauses $(g \vee \neg x)$ and $(\neg g \vee x)$ to ϕ. $(g \Leftrightarrow x.)$
3. If g is a **true** gate, add (g) to ϕ; likewise, if it is a **false** gate, add $(\neg g)$.
4. If g is a *NOT* gate with predecessor h, add the clauses $(g \vee h)$ and $(\neg g \vee \neg h)$ to ϕ.

Reduction from CIRCUIT-SAT to SAT

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.

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.
2. If g is a variable gate, corresponding to variable x, add the clauses $(g \vee \neg x)$ and $(\neg g \vee x)$ to ϕ. $(g \Leftrightarrow x.)$
3. If g is a **true** gate, add (g) to ϕ; likewise, if it is a **false** gate, add $(\neg g)$.
4. If g is a *NOT* gate with predecessor h, add the clauses $(g \vee h)$ and $(\neg g \vee \neg h)$ to ϕ.
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.

Reduction from CIRCUIT-SAT to SAT

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.

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.
2. If g is a variable gate, corresponding to variable x, add the clauses $(g \vee \neg x)$ and $(\neg g \vee x)$ to ϕ. ($g \Leftrightarrow x$.)
3. If g is a **true** gate, add (g) to ϕ; likewise, if it is a **false** gate, add $(\neg g)$.
4. If g is a *NOT* gate with predecessor h, add the clauses $(g \vee h)$ and $(\neg g \vee \neg h)$ to ϕ.
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. ($g \Leftrightarrow (h \vee h')$.)

Reduction from CIRCUIT-SAT to SAT

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.

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.
2. If g is a variable gate, corresponding to variable x, add the clauses $(g \vee \neg x)$ and $(\neg g \vee x)$ to ϕ. $(g \Leftrightarrow x.)$
3. If g is a **true** gate, add (g) to ϕ; likewise, if it is a **false** gate, add $(\neg g)$.
4. If g is a *NOT* gate with predecessor h, add the clauses $(g \vee h)$ and $(\neg g \vee \neg h)$ to ϕ.
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. $(g \Leftrightarrow (h \vee h').)$
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.

Reduction from CIRCUIT-SAT to SAT

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.

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.
2. If g is a variable gate, corresponding to variable x, add the clauses $(g \vee \neg x)$ and $(\neg g \vee x)$ to ϕ. $(g \Leftrightarrow x.)$
3. If g is a **true** gate, add (g) to ϕ; likewise, if it is a **false** gate, add $(\neg g)$.
4. If g is a *NOT* gate with predecessor h, add the clauses $(g \vee h)$ and $(\neg g \vee \neg h)$ to ϕ.
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. $(g \Leftrightarrow (h \vee h').)$
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. $(g \Leftrightarrow (h \wedge h').)$

Reduction from CIRCUIT-SAT to SAT

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.

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.
2. If g is a variable gate, corresponding to variable x, add the clauses $(g \vee \neg x)$ and $(\neg g \vee x)$ to ϕ. ($g \Leftrightarrow x$.)
3. If g is a **true** gate, add (g) to ϕ; likewise, if it is a **false** gate, add $(\neg g)$.
4. If g is a *NOT* gate with predecessor h, add the clauses $(g \vee h)$ and $(\neg g \vee \neg h)$ to ϕ.
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. ($g \Leftrightarrow (h \vee h')$.)
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. ($g \Leftrightarrow (h \wedge h')$.)
7. If g is an output gate, add the clause (g).

Argument

Argument

Argument

Argument

Argument

1. If C is satisfiable, then ϕ is satisfiable.

Argument

Argument

1. If C is satisfiable, then ϕ is satisfiable.
2. If ϕ is satisfiable, then C is satisfiable.

Graph coloring

Graph coloring

The Graph coloring problem

Graph coloring

The Graph coloring problem

A coloring of an undirected graph $G = \langle V, E \rangle$ is an assignment $V \rightarrow \{1, 2, \ldots, k\}$.

Graph coloring

The Graph coloring problem

A coloring of an undirected graph $G = \langle V, E \rangle$ is an assignment $V \rightarrow \{1, 2, \ldots, k\}$.

The coloring is said to be **valid** if no two adjacent vertices have the same color.

Graph coloring

The Graph coloring problem

A coloring of an undirected graph $G = \langle V, E \rangle$ is an assignment $V \rightarrow \{1, 2, \ldots, k\}$.

The coloring is said to be **valid** if no two adjacent vertices have the same color.

In the GRAPH k-COLORING problem, you are given a number k and asked if G can be colored using k colors.

Graph coloring

The Graph coloring problem

A coloring of an undirected graph $G = \langle V, E \rangle$ is an assignment $V \to \{1, 2, \ldots, k\}$.

The coloring is said to be **valid** if no two adjacent vertices have the same color.

In the GRAPH k-COLORING problem, you are given a number k and asked if G can be colored using k colors.

Exercise

Graph coloring

The Graph coloring problem

A coloring of an undirected graph $G = \langle V, E \rangle$ is an assignment $V \rightarrow \{1, 2, \ldots, k\}$.

The coloring is said to be **valid** if no two adjacent vertices have the same color.

In the GRAPH k-COLORING problem, you are given a number k and asked if G can be colored using k colors.

Exercise

1. *Argue that* GRAPH 2-COLORING *is in* **P**.

Graph coloring

The Graph coloring problem

A coloring of an undirected graph $G = \langle V, E \rangle$ is an assignment $V \to \{1, 2, \ldots, k\}$.

The coloring is said to be **valid** if no two adjacent vertices have the same color.

In the GRAPH k-COLORING problem, you are given a number k and asked if G can be colored using k colors.

Exercise

1. *Argue that* GRAPH 2-COLORING *is in* **P**.
2. *Argue that* GRAPH 3-COLORING *can be reduced to* 3*SAT*.

3-coloring to 3-SAT

3-coloring to 3-SAT

Reduction

3-coloring to 3-SAT

Reduction

Input: An undirected graph $G = \langle V, E \rangle$.

3-coloring to 3-SAT

Reduction

Input: An undirected graph $G = \langle V, E \rangle$.

Output: A CNF formula ϕ, such that ϕ is satisfiable if and only if G has a valid 3-coloring.

3-coloring to 3-SAT

Reduction

Input: An undirected graph $G = \langle V, E \rangle$.

Output: A CNF formula ϕ, such that ϕ is satisfiable if and only if G has a valid 3-coloring.

1. Let x_{ij}, $i = 1, 2, \ldots, n$, $j = 1, 2, 3$ be the boolean variable that is **true** if vertex i gets color j, and **false** otherwise.

3-coloring to 3-SAT

Reduction

Input: An undirected graph $G = \langle V, E \rangle$.

Output: A CNF formula ϕ, such that ϕ is satisfiable if and only if G has a valid 3-coloring.

1. Let x_{ij}, $i = 1, 2, \ldots, n$, $j = 1, 2, 3$ be the boolean variable that is **true** if vertex i gets color j, and **false** otherwise.
2. Every vertex should get at least one color.

3-coloring to 3-SAT

Reduction

Input: An undirected graph $G = \langle V, E \rangle$.

Output: A CNF formula ϕ, such that ϕ is satisfiable if and only if G has a valid 3-coloring.

1. Let x_{ij}, $i = 1, 2, \ldots, n$, $j = 1, 2, 3$ be the boolean variable that is **true** if vertex i gets color j, and **false** otherwise.
2. Every vertex should get at least one color.

$$(x_{i1} \vee x_{i2} \vee x_{i3}), \quad i = 1, 2, \ldots, n$$

3-coloring to 3-SAT

Reduction

Input: An undirected graph $G = \langle V, E \rangle$.

Output: A CNF formula ϕ, such that ϕ is satisfiable if and only if G has a valid 3-coloring.

1. Let x_{ij}, $i = 1, 2, \ldots, n$, $j = 1, 2, 3$ be the boolean variable that is **true** if vertex i gets color j, and **false** otherwise.
2. Every vertex should get at least one color.

$$(x_{i1} \vee x_{i2} \vee x_{i3}), \quad i = 1, 2, \ldots, n$$

3. Every vertex should get at most one color.

3-coloring to 3-SAT

Reduction

Input: An undirected graph $G = \langle V, E \rangle$.

Output: A CNF formula ϕ, such that ϕ is satisfiable if and only if G has a valid 3-coloring.

1. Let x_{ij}, $i = 1, 2, \ldots, n$, $j = 1, 2, 3$ be the boolean variable that is **true** if vertex i gets color j, and **false** otherwise.
2. Every vertex should get at least one color.

$$(x_{i1} \vee x_{i2} \vee x_{i3}), \quad i = 1, 2, \ldots, n$$

3. Every vertex should get at most one color.

3-coloring to 3-SAT

Reduction

Input: An undirected graph $G = \langle V, E \rangle$.

Output: A CNF formula ϕ, such that ϕ is satisfiable if and only if G has a valid 3-coloring.

1. Let x_{ij}, $i = 1, 2, \ldots, n$, $j = 1, 2, 3$ be the boolean variable that is **true** if vertex i gets color j, and **false** otherwise.
2. Every vertex should get at least one color.

$$(x_{i1} \vee x_{i2} \vee x_{i3}), \quad i = 1, 2, \ldots, n$$

3. Every vertex should get at most one color.

$$\neg(x_{i1} \quad \wedge \quad x_{i2})$$

3-coloring to 3-SAT

Reduction

Input: An undirected graph $G = \langle V, E \rangle$.

Output: A CNF formula ϕ, such that ϕ is satisfiable if and only if G has a valid 3-coloring.

1. Let x_{ij}, $i = 1, 2, \ldots, n$, $j = 1, 2, 3$ be the boolean variable that is **true** if vertex i gets color j, and **false** otherwise.
2. Every vertex should get at least one color.

$$(x_{i1} \vee x_{i2} \vee x_{i3}), \quad i = 1, 2, \ldots, n$$

3. Every vertex should get at most one color.

$$\begin{array}{ccc} \neg(x_{i1} & \wedge & x_{i2}) \\ \neg(x_{i1} & \wedge & x_{i3}) \end{array}$$

3-coloring to 3-SAT

Reduction

Input: An undirected graph $G = \langle V, E \rangle$.

Output: A CNF formula ϕ, such that ϕ is satisfiable if and only if G has a valid 3-coloring.

1. Let x_{ij}, $i = 1, 2, \ldots, n$, $j = 1, 2, 3$ be the boolean variable that is **true** if vertex i gets color j, and **false** otherwise.
2. Every vertex should get at least one color.

$$(x_{i1} \vee x_{i2} \vee x_{i3}), \quad i = 1, 2, \ldots, n$$

3. Every vertex should get at most one color.

$$\begin{array}{ccc} \neg(x_{i1} & \wedge & x_{i2}) \\ \neg(x_{i1} & \wedge & x_{i3}) \\ \neg(x_{i2} & \wedge & x_{i3}), \end{array}$$

3-coloring to 3-SAT

Reduction

Input: An undirected graph $G = \langle V, E \rangle$.

Output: A CNF formula ϕ, such that ϕ is satisfiable if and only if G has a valid 3-coloring.

1. Let x_{ij}, $i = 1, 2, \ldots, n$, $j = 1, 2, 3$ be the boolean variable that is **true** if vertex i gets color j, and **false** otherwise.
2. Every vertex should get at least one color.

$$(x_{i1} \vee x_{i2} \vee x_{i3}), \quad i = 1, 2, \ldots, n$$

3. Every vertex should get at most one color.

$$\begin{array}{lcl} \neg(x_{i1} & \wedge & x_{i2}) \\ \neg(x_{i1} & \wedge & x_{i3}) \\ \neg(x_{i2} & \wedge & x_{i3}), i = 1, 2, \ldots, n \end{array}$$

Completing the reduction

Completing the reduction

Connectivity requirements

Completing the reduction

Connectivity requirements

If $(u, v) \in E$, then u and v should get different colors.

Completing the reduction

Connectivity requirements

If $(u, v) \in E$, then u and v should get different colors.

Completing the reduction

Connectivity requirements

If $(u, v) \in E$, then u and v should get different colors.

$$\neg(x_{u1} \quad \wedge \quad x_{v1})$$

Completing the reduction

Connectivity requirements

If $(u, v) \in E$, then u and v should get different colors.

$$\begin{array}{ccc} \neg(x_{u1} & \wedge & x_{v1}) \\ \neg(x_{u2} & \wedge & x_{v2}) \end{array}$$

Completing the reduction

Connectivity requirements

If $(u, v) \in E$, then u and v should get different colors.

$$\begin{array}{lcl} \neg(x_{u1} & \wedge & x_{v1}) \\ \neg(x_{u2} & \wedge & x_{v2}) \\ \neg(x_{u3} & \wedge & x_{v3}) \end{array}$$

Completing the reduction

Connectivity requirements

If $(u, v) \in E$, then u and v should get different colors.

$$\begin{array}{ccc} \neg(x_{u1} & \wedge & x_{v1}) \\ \neg(x_{u2} & \wedge & x_{v2}) \\ \neg(x_{u3} & \wedge & x_{v3}) \\ \forall(u, v) & \in & E \end{array}$$

Integer Partitioning and Subset Sum

Integer Partitioning and Subset Sum

Integer Partitioning

Integer Partitioning and Subset Sum

Integer Partitioning

Given a list $S = \{x_1, x_2, \ldots, x_n\}$ of integers, is there a set $A \subseteq S$, such that $\sum_{x_i \in A} x_i = \sum_{x_i \notin A} x_i$?

Integer Partitioning and Subset Sum

Integer Partitioning

Given a list $S = \{x_1, x_2, \ldots, x_n\}$ of integers, is there a set $A \subseteq S$, such that $\sum_{x_i \in A} x_i = \sum_{x_i \notin A} x_i$?

Subset Sum

Integer Partitioning and Subset Sum

Integer Partitioning

Given a list $S = \{x_1, x_2, \ldots, x_n\}$ of integers, is there a set $A \subseteq S$, such that $\sum_{x_i \in A} x_i = \sum_{x_i \notin A} x_i$?

Subset Sum

Given a list $S = \{x_1, x_2, \ldots, x_n\}$ of integers and a target t, is there a set $A \subseteq S$, such that $\sum_{x_i \in A} x_i = t$?

Integer Partitioning and Subset Sum

Integer Partitioning

Given a list $S = \{x_1, x_2, \ldots, x_n\}$ of integers, is there a set $A \subseteq S$, such that $\sum_{x_i \in A} x_i = \sum_{x_i \notin A} x_i$?

Subset Sum

Given a list $S = \{x_1, x_2, \ldots, x_n\}$ of integers and a target t, is there a set $A \subseteq S$, such that $\sum_{x_i \in A} x_i = t$?

Exercise

Integer Partitioning and Subset Sum

Integer Partitioning

Given a list $S = \{x_1, x_2, \ldots, x_n\}$ of integers, is there a set $A \subseteq S$, such that $\sum_{x_i \in A} x_i = \sum_{x_i \notin A} x_i$?

Subset Sum

Given a list $S = \{x_1, x_2, \ldots, x_n\}$ of integers and a target t, is there a set $A \subseteq S$, such that $\sum_{x_i \in A} x_i = t$?

Exercise

Reduce INTEGER PARTITIONING *to* SUBSET SUM

Binary Knapsack

Binary Knapsack

Binary Knapsack

Binary Knapsack

Binary Knapsack

1. You are given n objects $O = \{o_1, o_2, \ldots, o_n\}$.

Binary Knapsack

Binary Knapsack

1. You are given n objects $O = \{o_1, o_2, \ldots, o_n\}$.
2. Object o_i has weight w_i and profit p_i.

Binary Knapsack

Binary Knapsack

1. You are given n objects $O = \{o_1, o_2, \ldots, o_n\}$.
2. Object o_i has weight w_i and profit p_i.
3. You are also given a knapsack of weight capacity W.

Binary Knapsack

Binary Knapsack

1. You are given n objects $O = \{o_1, o_2, \ldots, o_n\}$.
2. Object o_i has weight w_i and profit p_i.
3. You are also given a knapsack of weight capacity W.
4. The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.

Binary Knapsack

Binary Knapsack

1. You are given n objects $O = \{o_1, o_2, \ldots, o_n\}$.
2. Object o_i has weight w_i and profit p_i.
3. You are also given a knapsack of weight capacity W.
4. The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.
5. Profits are additive.

Binary Knapsack

Binary Knapsack

1. You are given n objects $O = \{o_1, o_2, \ldots, o_n\}$.
2. Object o_i has weight w_i and profit p_i.
3. You are also given a knapsack of weight capacity W.
4. The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.
5. Profits are additive.
6. The integer programming formulation is:

Binary Knapsack

Binary Knapsack

1. You are given n objects $O = \{o_1, o_2, \ldots, o_n\}$.
2. Object o_i has weight w_i and profit p_i.
3. You are also given a knapsack of weight capacity W.
4. The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.
5. Profits are additive.
6. The integer programming formulation is:

$$\max \qquad \sum_{i=1}^{n} p_i \cdot x_i$$

Binary Knapsack

Binary Knapsack

1. You are given n objects $O = \{o_1, o_2, \ldots, o_n\}$.
2. Object o_i has weight w_i and profit p_i.
3. You are also given a knapsack of weight capacity W.
4. The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.
5. Profits are additive.
6. The integer programming formulation is:

$$\begin{array}{lll} \max & \sum_{i=1}^{n} p_i \cdot x_i & \\ & \sum_{i=1}^{n} w_i \cdot x_i & \leq W \end{array}$$

Binary Knapsack

Binary Knapsack

1. You are given n objects $O = \{o_1, o_2, \ldots, o_n\}$.
2. Object o_i has weight w_i and profit p_i.
3. You are also given a knapsack of weight capacity W.
4. The goal is to select a subset of the objects which does not violate the capacity constraint of the knapsack while maximizing the profit of the objects selected.
5. Profits are additive.
6. The integer programming formulation is:

$$\begin{array}{lll} \max & \sum_{i=1}^{n} p_i \cdot x_i & \\ & \sum_{i=1}^{n} w_i \cdot x_i & \leq W \\ & x_i = \{0, 1\} \;\; \forall i = 1, 2, \ldots, n & \end{array}$$

Binary Knapsack (contd.)

Binary Knapsack (contd.)

Exercise

Binary Knapsack (contd.)

Exercise

Demonstrate through a counterexample that the greedy strategy used for fractional knapsack does not work in the binary knapsack case.

Binary Knapsack (contd.)

Exercise

Demonstrate through a counterexample that the greedy strategy used for fractional knapsack does not work in the binary knapsack case.

Solution

Binary Knapsack (contd.)

Exercise

Demonstrate through a counterexample that the greedy strategy used for fractional knapsack does not work in the binary knapsack case.

Solution

1. Consider three objects o_1, o_2 and o_3 with weights 10 units, 20 units and 30 units respectively and profits \$60, \$100 and \$120 respectively.

Binary Knapsack (contd.)

Exercise

Demonstrate through a counterexample that the greedy strategy used for fractional knapsack does not work in the binary knapsack case.

Solution

1. Consider three objects o_1, o_2 and o_3 with weights 10 units, 20 units and 30 units respectively and profits \$60, \$100 and \$120 respectively.
2. Let the knapsack have weight capacity 50 units.

Binary Knapsack (contd.)

Exercise

Demonstrate through a counterexample that the greedy strategy used for fractional knapsack does not work in the binary knapsack case.

Solution

1. Consider three objects o_1, o_2 and o_3 with weights 10 units, 20 units and 30 units respectively and profits \$60, \$100 and \$120 respectively.
2. Let the knapsack have weight capacity 50 units.
3. The greedy solution is

Binary Knapsack (contd.)

Exercise

Demonstrate through a counterexample that the greedy strategy used for fractional knapsack does not work in the binary knapsack case.

Solution

1. Consider three objects o_1, o_2 and o_3 with weights 10 units, 20 units and 30 units respectively and profits \$60, \$100 and \$120 respectively.
2. Let the knapsack have weight capacity 50 units.
3. The greedy solution is $\{o_1, o_2\}$.

Binary Knapsack (contd.)

Exercise

Demonstrate through a counterexample that the greedy strategy used for fractional knapsack does not work in the binary knapsack case.

Solution

1. Consider three objects o_1, o_2 and o_3 with weights 10 units, 20 units and 30 units respectively and profits \$60, \$100 and \$120 respectively.
2. Let the knapsack have weight capacity 50 units.
3. The greedy solution is $\{o_1, o_2\}$.
4. The optimal solution is

Binary Knapsack (contd.)

Exercise

Demonstrate through a counterexample that the greedy strategy used for fractional knapsack does not work in the binary knapsack case.

Solution

1. Consider three objects o_1, o_2 and o_3 with weights 10 units, 20 units and 30 units respectively and profits \$60, \$100 and \$120 respectively.
2. Let the knapsack have weight capacity 50 units.
3. The greedy solution is $\{o_1, o_2\}$.
4. The optimal solution is $\{o_2, o_3\}$.

A DP-based algorithm for binary knapsack

A DP-based algorithm for binary knapsack

Principle of optimality

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let KNAP(n, W) denote the given instance of the problem.

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let KNAP(n, W) denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let KNAP(n, W) denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object o_n.

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let KNAP(n, W) denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object o_n.
4. Either $o_n \in S$ or $o_n \notin S$.

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let KNAP(n, W) denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object o_n.
4. Either $o_n \in S$ or $o_n \notin S$.
5. If $o_n \in S$, then $S - \{o_n\}$ **must** constitute an optimal solution for

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let KNAP(n, W) denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object o_n.
4. Either $o_n \in S$ or $o_n \notin S$.
5. If $o_n \in S$, then $S - \{o_n\}$ **must** constitute an optimal solution for KNAP($n-1, W-w_n$).

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let KNAP(n, W) denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object o_n.
4. Either $o_n \in S$ or $o_n \notin S$.
5. If $o_n \in S$, then $S - \{o_n\}$ **must** constitute an optimal solution for KNAP$(n-1, W-w_n)$. (Why?)

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let KNAP(n, W) denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object o_n.
4. Either $o_n \in S$ or $o_n \notin S$.
5. If $o_n \in S$, then $S - \{o_n\}$ **must** constitute an optimal solution for KNAP$(n-1, W-w_n)$. (Why?)
6. If $o_n \notin S$, then S **must** be an optimal solution for

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\text{KNAP}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object o_n.
4. Either $o_n \in S$ or $o_n \notin S$.
5. If $o_n \in S$, then $S - \{o_n\}$ **must** constitute an optimal solution for $\text{KNAP}(n-1, W-w_n)$. (Why?)
6. If $o_n \notin S$, then S **must** be an optimal solution for $\text{KNAP}(n-1,$

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let KNAP(n, W) denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object o_n.
4. Either $o_n \in S$ or $o_n \notin S$.
5. If $o_n \in S$, then $S - \{o_n\}$ **must** constitute an optimal solution for KNAP$(n-1, W-w_n)$. (Why?)
6. If $o_n \notin S$, then S **must** be an optimal solution for KNAP$(n-1, W)$.

A DP-based algorithm for binary knapsack

Principle of optimality

1. Let $\text{KNAP}(n, W)$ denote the given instance of the problem.
2. Let $S \subseteq O$ denote the optimal solution.
3. Focus on object o_n.
4. Either $o_n \in S$ or $o_n \notin S$.
5. If $o_n \in S$, then $S - \{o_n\}$ **must** constitute an optimal solution for $\text{KNAP}(n-1, W-w_n)$. (Why?)
6. If $o_n \notin S$, then S **must** be an optimal solution for $\text{KNAP}(n-1, W)$. (Why?)

Formulating the recurrence

Formulating the recurrence

The Recurrence

Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity w.

Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity w.
2. Which entry of the table are we interested in?

Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity w.
2. Which entry of the table are we interested in? Clearly, $V[n, W]$.

Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity w.
2. Which entry of the table are we interested in? Clearly, $V[n, W]$.
3. As per the discussion above,

Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity w.
2. Which entry of the table are we interested in? Clearly, $V[n, W]$.
3. As per the discussion above,

$$V[i, w] \quad = \quad \max \left\{ \right.$$

Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity w.
2. Which entry of the table are we interested in? Clearly, $V[n, W]$.
3. As per the discussion above,

$$V[i, w] \quad = \quad \max \begin{cases} V[i-1, w-w_i] + p_i \\ \end{cases}$$

Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity w.
2. Which entry of the table are we interested in? Clearly, $V[n, W]$.
3. As per the discussion above,

$$V[i, w] \quad = \quad \max \begin{cases} V[i-1, w-w_i] + p_i & (o_i \text{ is included}) \\ \end{cases}$$

Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity w.
2. Which entry of the table are we interested in? Clearly, $V[n, W]$.
3. As per the discussion above,

$$V[i, w] \quad = \quad \max \begin{cases} V[i-1, w-w_i] + p_i & (o_i \text{ is included}) \\ V[i-1, w] \end{cases}$$

Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity w.
2. Which entry of the table are we interested in? Clearly, $V[n, W]$.
3. As per the discussion above,

$$V[i, w] \quad = \quad \max \begin{cases} V[i-1, w-w_i] + p_i & (o_i \text{ is included}) \\ V[i-1, w] & (o_i \text{ is excluded}) \end{cases}$$

Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity w.
2. Which entry of the table are we interested in? Clearly, $V[n, W]$.
3. As per the discussion above,

$$V[i, w] \quad = \quad \max \begin{cases} V[i-1, w-w_i] + p_i & (o_i \text{ is included}) \\ V[i-1, w] & (o_i \text{ is excluded}) \end{cases}$$

4. Initial conditions:

Formulating the recurrence

The Recurrence

1. Let $V[i, w]$ denote the optimal solution for the subset $\{o_1, o_2, \ldots, o_i\}$, assuming that the Knapsack has a capacity w.
2. Which entry of the table are we interested in? Clearly, $V[n, W]$.
3. As per the discussion above,

$$V[i, w] \quad = \quad \max \begin{cases} V[i-1, w-w_i] + p_i & (o_i \text{ is included}) \\ V[i-1, w] & (o_i \text{ is excluded}) \end{cases}$$

4. Initial conditions:

$$\begin{aligned} V[0, w] &= 0, \quad 0 \leq w \leq W \\ pause V[i, w] &= -\infty, \quad w < 0 \end{aligned}$$

Example

Example

Example

Example

Example

Solve the following instance of Knapsack:
$n = 4$,

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$,

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$,

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$											

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1											

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0						

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2											

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0							

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3											

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0							

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4											

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0								

Example

Example

Solve the following instance of Knapsack:
$n = 4$, $\mathbf{w} = \langle 5, 4, 6, 3 \rangle$, $W = 10$, $\mathbf{p} = \langle 10, 40, 30, 50 \rangle$.
Solution:

$V[i, w]$	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Final observations

Final observations

Observation

Final observations

Observation

1. *The running time of the DP-based algorithm for binary knapsack is*

Final observations

Observation

1. *The running time of the DP-based algorithm for binary knapsack is* $O(n \cdot W)$.

Final observations

Observation

1. *The running time of the DP-based algorithm for binary knapsack is* $O(n \cdot W)$.
2. *Is the running time polynomial?*

Final observations

Observation

1. *The running time of the DP-based algorithm for binary knapsack is $O(n \cdot W)$.*
2. *Is the running time polynomial?*
3. *The Subset Sum problem can be easily reduced to binary knapsack.*

Final observations

Observation

1. *The running time of the DP-based algorithm for binary knapsack is $O(n \cdot W)$.*
2. *Is the running time polynomial?*
3. *The Subset Sum problem can be easily reduced to binary knapsack. How?*

Final observations

Observation

1. *The running time of the DP-based algorithm for binary knapsack is $O(n \cdot W)$.*
2. *Is the running time polynomial?*
3. *The Subset Sum problem can be easily reduced to binary knapsack. How?*
4. *We thus have,* INTEGER PARTITION $\leq$ SUBSET SUM $\leq$ BINARY KNAPSACK.

Three related graph problems

Three related graph problems

Vertex Cover (VC)

Three related graph problems

Vertex Cover (VC)

Given a graph $G = \langle V, E \rangle$ and a number K, is there a set $V' \subseteq V$, $|V'| \leq K$, such that for every edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$?

Three related graph problems

Vertex Cover (VC)

Given a graph $G = \langle V, E \rangle$ and a number K, is there a set $V' \subseteq V$, $|V'| \leq K$, such that for every edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$?

Independent Set (IS)

Three related graph problems

Vertex Cover (VC)

Given a graph $G = \langle V, E \rangle$ and a number K, is there a set $V' \subseteq V$, $|V'| \leq K$, such that for every edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$?

Independent Set (IS)

Given a graph $G = \langle V, E \rangle$ and a number K, is there a set $V' \subseteq V$, $|V'| \geq K$, such that for every pair of vertices $(u, v) \in V'$, $(u, v) \notin E$.

Three related graph problems

Vertex Cover (VC)

Given a graph $G = \langle V, E \rangle$ and a number K, is there a set $V' \subseteq V$, $|V'| \leq K$, such that for every edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$?

Independent Set (IS)

Given a graph $G = \langle V, E \rangle$ and a number K, is there a set $V' \subseteq V$, $|V'| \geq K$, such that for every pair of vertices $(u, v) \in V'$, $(u, v) \notin E$.

Clique (CQ)

Three related graph problems

Vertex Cover (VC)

Given a graph $G = \langle V, E \rangle$ and a number K, is there a set $V' \subseteq V$, $|V'| \leq K$, such that for every edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$?

Independent Set (IS)

Given a graph $G = \langle V, E \rangle$ and a number K, is there a set $V' \subseteq V$, $|V'| \geq K$, such that for every pair of vertices $(u, v) \in V'$, $(u, v) \notin E$.

Clique (CQ)

Given a graph $G = \langle V, E \rangle$ and a number K, is there a set $V' \subseteq V$, $|V'| \leq K$, such that for pair of vertices $(u, v) \in V'$, $(u, v) \in E$.

Observation relating the three problems

Observation relating the three problems

Theorem

Observation relating the three problems

Theorem

Let $G = \langle V, E \rangle$ denote a graph and let $S \subseteq V$.

Observation relating the three problems

Theorem

Let $G = \langle V, E \rangle$ denote a graph and let $S \subseteq V$.

The following statements are equivalent:

Observation relating the three problems

Theorem

Let $G = \langle V, E \rangle$ denote a graph and let $S \subseteq V$.

The following statements are equivalent:

1. *S is a vertex cover.*

Observation relating the three problems

Theorem

Let $G = \langle V, E \rangle$ denote a graph and let $S \subseteq V$.

The following statements are equivalent:

1. *S is a vertex cover.*
2. *$V - S$ is an independent set.*

Observation relating the three problems

Theorem

Let $G = \langle V, E \rangle$ denote a graph and let $S \subseteq V$.

The following statements are equivalent:

1. *S is a vertex cover.*
2. *$V - S$ is an independent set.*
3. *$V - S$ is a clique in $G^c = \langle V, E^c \rangle$, where two vertices are adjacent in G^c if and only if they are non-adjacent in G.*

Observation relating the three problems

Theorem

Let $G = \langle V, E \rangle$ denote a graph and let $S \subseteq V$.

The following statements are equivalent:

1. *S is a vertex cover.*
2. *$V - S$ is an independent set.*
3. *$V - S$ is a clique in $G^c = \langle V, E^c \rangle$, where two vertices are adjacent in G^c if and only if they are non-adjacent in G.*

Exercise

Observation relating the three problems

Theorem

Let $G = \langle V, E \rangle$ denote a graph and let $S \subseteq V$.

The following statements are equivalent:

1. *S is a vertex cover.*
2. *$V - S$ is an independent set.*
3. *$V - S$ is a clique in $G^c = \langle V, E^c \rangle$, where two vertices are adjacent in G^c if and only if they are non-adjacent in G.*

Exercise

1. *Argue that $VC \leq IS \leq CQ$.*

Observation relating the three problems

Theorem

Let $G = \langle V, E \rangle$ denote a graph and let $S \subseteq V$.

The following statements are equivalent:

1. *S is a vertex cover.*
2. *$V - S$ is an independent set.*
3. *$V - S$ is a clique in $G^c = \langle V, E^c \rangle$, where two vertices are adjacent in G^c if and only if they are non-adjacent in G.*

Exercise

1. *Argue that $VC \leq IS \leq CQ$.*
2. *Show that if a graph is k-colorable, then it has an independent set of size at least $\frac{n}{k}$.*

Observation relating the three problems

Theorem

Let $G = \langle V, E \rangle$ denote a graph and let $S \subseteq V$.

The following statements are equivalent:

1. *S is a vertex cover.*
2. *$V - S$ is an independent set.*
3. *$V - S$ is a clique in $G^c = \langle V, E^c \rangle$, where two vertices are adjacent in G^c if and only if they are non-adjacent in G.*

Exercise

1. *Argue that $VC \leq IS \leq CQ$.*
2. *Show that if a graph is k-colorable, then it has an independent set of size at least $\frac{n}{k}$. Is the converse true.*

First Formal Definition

First Formal Definition

Definition

First Formal Definition

Definition

NP is the class of problems *A* of the following form:

First Formal Definition

Definition

NP is the class of problems A of the following form:

x is a yes-instance of A if and only if there exists a w,

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **P** regarding pairs (x, w) and $|w| = poly(|x|)$.

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **P** regarding pairs (x, w) and $|w| = poly(|x|)$.

Observations

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **P** regarding pairs (x, w) and $|w| = poly(|x|)$.

Observations

1. *w is a witness of the fact that x is a yes-instance.*

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **P** regarding pairs (x, w) and $|w| = poly(|x|)$.

Observations

1. *w is a witness of the fact that x is a yes-instance. It is called a* certificate.

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **P** regarding pairs (x, w) and $|w| = poly(|x|)$.

Observations

1. *w is a witness of the fact that x is a yes-instance. It is called a* certificate.
2. *B is the problem of checking whether x is a genuine needle.*

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **P** regarding pairs (x, w) and $|w| = poly(|x|)$.

Observations

1. *w is a witness of the fact that x is a yes-instance. It is called a* certificate.
2. *B is the problem of checking whether x is a genuine needle. For instance, if A is* HAMILTON-PATH*, then x is a graph,*

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **P** regarding pairs (x, w) and $|w| = poly(|x|)$.

Observations

1. *w is a witness of the fact that x is a yes-instance. It is called a* certificate.
2. *B is the problem of checking whether x is a genuine needle. For instance, if A is* HAMILTON-PATH*, then x is a graph, w is a path,*

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **P** regarding pairs (x, w) and $|w| = poly(|x|)$.

Observations

1. *w is a witness of the fact that x is a yes-instance. It is called a* certificate.
2. *B is the problem of checking whether x is a genuine needle. For instance, if A is* HAMILTON-PATH*, then x is a graph, w is a path, and B is the problem of checking whether w is a valid Hamilton path for x.*

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **P** regarding pairs (x, w) and $|w| = poly(|x|)$.

Observations

1. *w is a witness of the fact that x is a yes-instance. It is called a* certificate.
2. *B is the problem of checking whether x is a genuine needle. For instance, if A is* HAMILTON-PATH*, then x is a graph, w is a path, and B is the problem of checking whether w is a valid Hamilton path for x.*
3. *w is required to be polynomially balanced.*

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **P** regarding pairs (x, w) and $|w| = poly(|x|)$.

Observations

1. *w is a witness of the fact that x is a yes-instance. It is called a* certificate.
2. *B is the problem of checking whether x is a genuine needle. For instance, if A is* HAMILTON-PATH*, then x is a graph, w is a path, and B is the problem of checking whether w is a valid Hamilton path for x.*
3. *w is required to be polynomially balanced. This ensures that B runs in time polynomial in $|x|$.*

First Formal Definition

Definition

NP is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **P** regarding pairs (x, w) and $|w| = poly(|x|)$.

Observations

1. *w is a witness of the fact that x is a yes-instance. It is called a* certificate.
2. *B is the problem of checking whether x is a genuine needle. For instance, if A is* HAMILTON-PATH*, then x is a graph, w is a path, and B is the problem of checking whether w is a valid Hamilton path for x.*
3. *w is required to be polynomially balanced. This ensures that B runs in time polynomial in $|x|$.*
4. **NP** $\subseteq$ **EXP**, *where* **EXP**=**TIME**$(2^{poly(n)})$.

Generalizing **NP**

Generalizing **NP**

Definition

Generalizing **NP**

Definition

NTIME$(f(n))$ is the class of problems A of the following form:

Generalizing **NP**

Definition

NTIME$(f(n))$ is the class of problems A of the following form:

x is a yes-instance of A if and only if there exists a w,

Generalizing **NP**

Definition

NTIME$(f(n))$ is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

Generalizing **NP**

Definition

NTIME$(f(n))$ is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **TIME**$(f(n)$ regarding pairs (x, w), $|x| = n$ and $|w| = O(f(n))$.

Generalizing **NP**

Definition

NTIME$(f(n))$ is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **TIME**$(f(n)$ regarding pairs (x, w), $|x| = n$ and $|w| = O(f(n))$.

As argued previously,

Generalizing **NP**

Definition

NTIME$(f(n))$ is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **TIME**$(f(n)$ regarding pairs (x, w), $|x| = n$ and $|w| = O(f(n))$.

As argued previously,

$$\textbf{NTIME}(f(n)) \subseteq \textbf{TIME}(2^{f(n)})$$

Generalizing **NP**

Definition

NTIME$(f(n))$ is the class of problems A of the following form:

> x is a yes-instance of A if and only if there exists a w, such that (x, w) is is a yes-instance of B,

where B is a decision problem in **TIME**$(f(n)$ regarding pairs (x, w), $|x| = n$ and $|w| = O(f(n))$.

As argued previously,

$$\textbf{NTIME}(f(n)) \subseteq \textbf{TIME}(2^{f(n)})$$

Another definition for **NP**

Another definition for **NP**

Definition

Another definition for **NP**

Definition

NP is the class of properties *A* of the form

Another definition for **NP**

Definition

NP is the class of properties A of the form

$$A(x) = \exists w : B(x, w)$$

Another definition for **NP**

Definition

NP is the class of properties A of the form

$$A(x) = \exists w : B(x, w)$$

where B is in **P** and where $|w| = poly(|x|)$.

Some observations

Some observations

Observations

Some observations

Observations

1. We have associated with the decision problem A, the property $A(x)$, where $A(x)$ is **true** if and only if x is a yes-instance of A.

Some observations

Observations

1. We have associated with the decision problem A, the property $A(x)$, where $A(x)$ is **true** if and only if x is a yes-instance of A.

   For instance, if x is a graph and $A(x)$ is the property that x has a Hamilton path, then $B(x, w)$ is the polynomial time property that w is a Hamilton path for x.

Some observations

Observations

1. We have associated with the decision problem A, the property $A(x)$, where $A(x)$ is **true** if and only if x is a yes-instance of A.

   For instance, if x is a graph and $A(x)$ is the property that x has a Hamilton path, then $B(x, w)$ is the polynomial time property that w is a Hamilton path for x.

2. Algorithmically, the quantifier $\exists$ represents the process of searching for the witness w.

Some observations

Observations

1. We have associated with the decision problem A, the property $A(x)$, where $A(x)$ is **true** if and only if x is a yes-instance of A.

   For instance, if x is a graph and $A(x)$ is the property that x has a Hamilton path, then $B(x, w)$ is the polynomial time property that w is a Hamilton path for x.
2. Algorithmically, the quantifier $\exists$ represents the process of searching for the witness w.
3. Prover-Verifier conversation.

Some observations

Observations

1. We have associated with the decision problem A, the property $A(x)$, where $A(x)$ is **true** if and only if x is a yes-instance of A.

   For instance, if x is a graph and $A(x)$ is the property that x has a Hamilton path, then $B(x, w)$ is the polynomial time property that w is a Hamilton path for x.
2. Algorithmically, the quantifier $\exists$ represents the process of searching for the witness w.
3. Prover-Verifier conversation.
4. Are the complements of **P** properties in **P**?

Some observations

Observations

1. We have associated with the decision problem A, the property $A(x)$, where $A(x)$ is **true** if and only if x is a yes-instance of A.

   For instance, if x is a graph and $A(x)$ is the property that x has a Hamilton path, then $B(x, w)$ is the polynomial time property that w is a Hamilton path for x.
2. Algorithmically, the quantifier $\exists$ represents the process of searching for the witness w.
3. Prover-Verifier conversation.
4. Are the complements of **P** properties in **P**?
5. How about complements of **NP** properties? These properties belong to the class **coNP**; they have easy to check no instances, but no known method of verifying yes-instances in polynomial time.

Exercise

Exercise

Exercise

Exercise

Exercise

1. *Is* **coNP** *the complement of* **NP***?*

Exercise

Exercise

1. *Is* **coNP** *the complement of* **NP***?*
2. *Is* **NP** $\cap$ **coNP** *identical to* **P***?*

Exercise

Exercise

1. *Is* **coNP** *the complement of* **NP***?*
2. *Is* $\mathbf{NP} \cap \mathbf{coNP}$ *identical to* **P***?*
3. *Show that if* $\mathbf{P} = \mathbf{NP}$ *then* $\mathbf{NP} = \mathbf{coNP}$.

Exercise

Exercise

1. *Is* **coNP** *the complement of* **NP***?*
2. *Is* **NP** $\cap$ **coNP** *identical to* **P***?*
3. *Show that if* **P** $=$ **NP** *then* **NP** $=$ **coNP**. *Is the converse true?*

Nondeterministic Computation

Nondeterministic Computation

Fundamentals

Nondeterministic Computation

Fundamentals

1. A computer program is deterministic in that given the initial state and input, the execution trace is fixed, i.e., there are no choices for the program to make.

Nondeterministic Computation

Fundamentals

1. A computer program is deterministic in that given the initial state and input, the execution trace is fixed, i.e., there are no choices for the program to make.
2. A nondeterministic program can make several possible choices at each step.

Nondeterministic Computation

Fundamentals

1. A computer program is deterministic in that given the initial state and input, the execution trace is fixed, i.e., there are no choices for the program to make.
2. A nondeterministic program can make several possible choices at each step. For instance, consider the instruction:

Nondeterministic Computation

Fundamentals

1. A computer program is deterministic in that given the initial state and input, the execution trace is fixed, i.e., there are no choices for the program to make.
2. A nondeterministic program can make several possible choices at each step. For instance, consider the instruction:

   goto both line_1, line_2.

Nondeterministic Computation

Fundamentals

1. A computer program is deterministic in that given the initial state and input, the execution trace is fixed, i.e., there are no choices for the program to make.
2. A nondeterministic program can make several possible choices at each step. For instance, consider the instruction:

   goto both line_1, line_2.
3. The computation then becomes a tree instead of a straight line.

Nondeterministic Computation

Fundamentals

1. A computer program is deterministic in that given the initial state and input, the execution trace is fixed, i.e., there are no choices for the program to make.
2. A nondeterministic program can make several possible choices at each step. For instance, consider the instruction:

   goto both line_1, line_2.
3. The computation then becomes a tree instead of a straight line.
4. The output of a nondeterministic program is “yes”, if any of the computations in the tree leads to a an accepting state and “no” otherwise.

Nondeterministic Computation

Fundamentals

1. A computer program is deterministic in that given the initial state and input, the execution trace is fixed, i.e., there are no choices for the program to make.
2. A nondeterministic program can make several possible choices at each step. For instance, consider the instruction:

   goto both line_1, line_2.
3. The computation then becomes a tree instead of a straight line.
4. The output of a nondeterministic program is “yes”, if any of the computations in the tree leads to a an accepting state and “no” otherwise.
5. The running time of a nondeterministic program is the height of its computation tree.

Nondeterministic Computation

Fundamentals

1. A computer program is deterministic in that given the initial state and input, the execution trace is fixed, i.e., there are no choices for the program to make.
2. A nondeterministic program can make several possible choices at each step. For instance, consider the instruction:

   goto both line_1, line_2.
3. The computation then becomes a tree instead of a straight line.
4. The output of a nondeterministic program is "yes", if any of the computations in the tree leads to a an accepting state and "no" otherwise.
5. The running time of a nondeterministic program is the height of its computation tree.

Exercise

Write a nondeterministic program for 3*SAT.*

Final definition of **NP**

Final definition of **NP**

Definition

Final definition of **NP**

Definition

NP is the class of problems for which a nondeterministic program exists that runs in time $poly(n)$, on instances of length n,

Final definition of **NP**

Definition

NP is the class of problems for which a nondeterministic program exists that runs in time $poly(n)$, on instances of length n, such that the input is a yes-instance if and only if there exists a computation path that returns "yes."

Final definition of **NP**

Definition

NP is the class of problems for which a nondeterministic program exists that runs in time *poly*(n), on instances of length n, such that the input is a yes-instance if and only if there exists a computation path that returns "yes."

Definition

Final definition of **NP**

Definition

NP is the class of problems for which a nondeterministic program exists that runs in time $poly(n)$, on instances of length n, such that the input is a yes-instance if and only if there exists a computation path that returns "yes."

Definition

NTIME$(f(n))$ is the class of problems for which a nondeterministic program exists that runs in time $O(f(n))$, on instances of length n,

Final definition of **NP**

Definition

NP is the class of problems for which a nondeterministic program exists that runs in time $poly(n)$, on instances of length n, such that the input is a yes-instance if and only if there exists a computation path that returns "yes."

Definition

NTIME$(f(n))$ is the class of problems for which a nondeterministic program exists that runs in time $O(f(n))$, on instances of length n, such that the input is a yes-instance if and only if there exists a computation path that returns "yes."

Linear Programming

Linear Programming

The Problem (LP)

Linear Programming

The Problem (LP)

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &\leq \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Linear Programming

The Problem (LP)

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &\leq \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Observation

Linear Programming

The Problem (LP)

$$\begin{aligned} \exists \mathbf{x} \ \ \mathbf{A} \cdot \mathbf{x} &\leq \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Observation

1. *Is* LP *in* **NP***?*

Linear Programming

The Problem (LP)

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &\leq \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Observation

1. *Is* LP *in* **NP** *? Does Guess and Verify work?*

Linear Programming

The Problem (LP)

$$\begin{aligned} \exists \mathbf{x}\ \ \mathbf{A} \cdot \mathbf{x} &\leq \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Observation

1. *Is* LP *in* **NP**? *Does Guess and Verify work?*
2. *Is* LP *in* **coNP**?

Complexity

Complexity

Fundamentals

Complexity

Fundamentals

1. Assume that **A** has m rows and n columns.

Complexity

Fundamentals

1. Assume that **A** has m rows and n columns.
2. Observe that with the introduction of slack variables, we can rewrite the Linear programming problem as:

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

where $m \leq n$

Complexity

Fundamentals

1. Assume that **A** has m rows and n columns.
2. Observe that with the introduction of slack variables, we can rewrite the Linear programming problem as:

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

   where $m \leq n$
3. A basis of the above system is a collection of m linearly independent columns.

Complexity

Fundamentals

1. Assume that $\mathbf{A}$ has m rows and n columns.
2. Observe that with the introduction of slack variables, we can rewrite the Linear programming problem as:

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

   where $m \leq n$
3. A basis of the above system is a collection of m linearly independent columns.
4. A basic solution is obtained by solving the system $\mathbf{B} \cdot \mathbf{x_B} + \mathbf{N} \cdot \mathbf{x_N} = \mathbf{b}$, $\mathbf{x_N} = \mathbf{0}$.

Complexity

Fundamentals

1. Assume that **A** has *m* rows and *n* columns.
2. Observe that with the introduction of slack variables, we can rewrite the Linear programming problem as:

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

   where $m \leq n$
3. A basis of the above system is a collection of *m* linearly independent columns.
4. A basic solution is obtained by solving the system $\mathbf{B} \cdot \mathbf{x_B} + \mathbf{N} \cdot \mathbf{x_N} = \mathbf{b}$, $\mathbf{x_N} = \mathbf{0}$.
5. The basic solution is feasible if every element of $\mathbf{x_B}$ is non-negative.

Complexity

Fundamentals

1. Assume that **A** has *m* rows and *n* columns.
2. Observe that with the introduction of slack variables, we can rewrite the Linear programming problem as:

$$\begin{aligned} \exists \mathbf{x} \ \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

   where $m \leq n$
3. A basis of the above system is a collection of *m* linearly independent columns.
4. A basic solution is obtained by solving the system $\mathbf{B} \cdot \mathbf{x_B} + \mathbf{N} \cdot \mathbf{x_N} = \mathbf{b}$, $\mathbf{x_N} = \mathbf{0}$.
5. The basic solution is feasible if every element of $\mathbf{x_B}$ is non-negative.
6. The above system is feasible if and only if it has a basic feasible solution.

Complexity

Fundamentals

1. Assume that $\mathbf{A}$ has m rows and n columns.
2. Observe that with the introduction of slack variables, we can rewrite the Linear programming problem as:

$$\begin{aligned} \exists \mathbf{x}\ \ \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

   where $m \leq n$
3. A basis of the above system is a collection of m linearly independent columns.
4. A basic solution is obtained by solving the system $\mathbf{B} \cdot \mathbf{x_B} + \mathbf{N} \cdot \mathbf{x_N} = \mathbf{b}$, $\mathbf{x_N} = \mathbf{0}$.
5. The basic solution is feasible if every element of $\mathbf{x_B}$ is non-negative.
6. The above system is feasible if and only if it has a basic feasible solution.
7. So all that we have to do now is to show that the basic solutions are polynomial in the size of the input.

Linear Programming theorem

Linear Programming theorem

Theorem

Linear Programming theorem

Theorem

Let $\mathbf{x} = [x_1, x_2, \ldots, x_m, 0, 0, \ldots, 0]^T$ *be a basic solution of the system*

Linear Programming theorem

Theorem

Let $\mathbf{x} = [x_1, x_2, \ldots, x_m, 0, 0, \ldots, 0]^T$ *be a basic solution of the system*

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Linear Programming theorem

Theorem

Let $\mathbf{x} = [x_1, x_2, \ldots, x_m, 0, 0, \ldots, 0]^T$ *be a basic solution of the system*

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Then,

$$|x_j| \leq m! \cdot \alpha^{m-1} \cdot \beta$$

where,

Linear Programming theorem

Theorem

Let $\mathbf{x} = [x_1, x_2, \ldots, x_m, 0, 0, \ldots, 0]^T$ *be a basic solution of the system*

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &= \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Then,

$$|x_j| \leq m! \cdot \alpha^{m-1} \cdot \beta$$

where,

$$\begin{aligned} \alpha &= \max_{i,j} |a_{ij}| \\ \beta &= \max_j |b_j| \end{aligned}$$

Farkas' Lemma

Farkas' Lemma

Lemma

Farkas' Lemma

Lemma

Either,

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &\leq \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

Farkas' Lemma

Lemma

Either,

$$\begin{aligned} \exists \mathbf{x} \ \ \mathbf{A} \cdot \mathbf{x} &\leq \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

or (mutually exclusively)

Farkas' Lemma

Lemma

Either,

$$\begin{aligned} \exists \mathbf{x} \;\; \mathbf{A} \cdot \mathbf{x} &\leq \mathbf{b} \\ \mathbf{x} &\geq \mathbf{0} \end{aligned}$$

or (mutually exclusively)

$$\begin{aligned} \exists \mathbf{y} \;\; \mathbf{y} \cdot \mathbf{A} &\geq \mathbf{0} \\ \mathbf{y} &\geq \mathbf{0} \\ \mathbf{y} \cdot \mathbf{b} &< 0 \end{aligned}$$

Primality testing

Primality testing

PRIMES

Primality testing

PRIMES

Given a number N, determine whether it is a prime number, i.e., divisible only by one and itself.

Primality testing

PRIMES

Given a number N, determine whether it is a prime number, i.e., divisible only by one and itself.

Exercise

Primality testing

PRIMES

Given a number N, determine whether it is a prime number, i.e., divisible only by one and itself.

Exercise

1. *Show that* PRIMES *is in* **coNP**.

Primality testing

PRIMES

Given a number N, determine whether it is a prime number, i.e., divisible only by one and itself.

Exercise

1. *Show that* PRIMES *is in* **coNP**.
2. *Show that* PRIMES *is in* **NP**.

Notations

Notations

Logarithms and natural numbers

Notations

Logarithms and natural numbers

Normally, when taking a logarithm, we get a real number.

Notations

Logarithms and natural numbers

Normally, when taking a logarithm, we get a real number. In order to work with natural numbers, we adopt the following convention:

Notations

Logarithms and natural numbers

Normally, when taking a logarithm, we get a real number. In order to work with natural numbers, we adopt the following convention:

$$\log x = \lceil \log_2 x \rceil.$$

The Lucas test for primality

The Lucas test for primality

Theorem

The Lucas test for primality

Theorem

A number $p > 1$ is prime if and only if and only if there exists a number r,

The Lucas test for primality

Theorem

A number $p > 1$ is prime if and only if and only if there exists a number r, $1 < r < p$,

The Lucas test for primality

Theorem

A number $p > 1$ is prime if and only if and only if there exists a number r, $1 < r < p$, such that $r^{p-1} \equiv 1 \bmod p$

The Lucas test for primality

Theorem

A number $p > 1$ is prime if and only if and only if there exists a number r, $1 < r < p$, such that $r^{p-1} \equiv 1 \mod p$ and furthermore,

The Lucas test for primality

Theorem

A number $p > 1$ is prime if and only if and only if there exists a number r, $1 < r < p$, such that $r^{p-1} \equiv 1 \mod p$ and furthermore, $r^{\frac{p-1}{q}} \not\equiv 1 \mod p$ for all prime divisors q of $(p-1)$.

The Lucas test for primality

Theorem

A number $p > 1$ is prime if and only if and only if there exists a number r, $1 < r < p$, such that $r^{p-1} \equiv 1 \mod p$ and furthermore, $r^{\frac{p-1}{q}} \not\equiv 1 \mod p$ for all prime divisors q of $(p-1)$.

Exercise

The Lucas test for primality

Theorem

A number $p > 1$ is prime if and only if and only if there exists a number r, $1 < r < p$, such that $r^{p-1} \equiv 1 \mod p$ and furthermore, $r^{\frac{p-1}{q}} \not\equiv 1 \mod p$ for all prime divisors q of $(p-1)$.

Exercise

Can you design a nondeterministic algorithm for PRIMES*?*

The Lucas test for primality

Theorem

A number $p > 1$ is prime if and only if and only if there exists a number r, $1 < r < p$, such that $r^{p-1} \equiv 1 \mod p$ and furthermore, $r^{\frac{p-1}{q}} \not\equiv 1 \mod p$ for all prime divisors q of $(p-1)$.

Exercise

Can you design a nondeterministic algorithm for PRIMES*?*

We have to bound the number of prime divisors.

The Lucas test for primality

Theorem

A number $p > 1$ is prime if and only if and only if there exists a number r, $1 < r < p$, such that $r^{p-1} \equiv 1 \mod p$ and furthermore, $r^{\frac{p-1}{q}} \not\equiv 1 \mod p$ for all prime divisors q of $(p-1)$.

Exercise

Can you design a nondeterministic algorithm for PRIMES*?*

We have to bound the number of prime divisors.

How many prime divisors can p have?

The Lucas test for primality

Theorem

A number $p > 1$ is prime if and only if and only if there exists a number r, $1 < r < p$, such that $r^{p-1} \equiv 1 \mod p$ and furthermore, $r^{\frac{p-1}{q}} \not\equiv 1 \mod p$ for all prime divisors q of $(p-1)$.

Exercise

Can you design a nondeterministic algorithm for PRIMES*?*

We have to bound the number of prime divisors.

How many prime divisors can p have? At most $\log p$.

FUNCTION PRIMALITY CHECKING(p)

FUNCTION PRIMALITY CHECKING(p)

1: Guess r.

FUNCTION PRIMALITY CHECKING(p)

1: Guess r.
2: **if** ($r^{p-1} \not\equiv 1 \mod p$) **then**

```
FUNCTION PRIMALITY CHECKING(p)
Guess r.
if (r^{p-1} ≢ 1 mod p) then
    return("no").
```

FUNCTION PRIMALITY CHECKING(p)

1: Guess r.
2: **if** ($r^{p-1} \not\equiv 1 \mod p$) **then**
3: **return**("no").
4: **else**

FUNCTION PRIMALITY CHECKING(p)

Guess r.
if ($r^{p-1} \not\equiv 1 \mod p$) **then**
  return("no").
else
  Guess $q_1, q_2, \ldots q_k$ as the prime divisors of $(p-1)$.

FUNCTION PRIMALITY CHECKING(p)

```
Guess r.
if (r^{p-1} ≢ 1 mod p) then
    return("no").
else
    Guess q_1, q_2, ... q_k as the prime divisors of (p − 1).
    if (any q_i is not a prime divisor of (p − 1)) then
```

```
FUNCTION PRIMALITY CHECKING(p)
Guess r.
if (r^(p-1) ≢ 1 mod p) then
    return("no").
else
    Guess q_1, q_2, ... q_k as the prime divisors of (p − 1).
    if (any q_i is not a prime divisor of (p − 1)) then
        return("no").
```

FUNCTION PRIMALITY CHECKING(p)

```
Guess r.
if (r^(p-1) ≢ 1 mod p) then
    return("no").
else
    Guess q_1, q_2, ... q_k as the prime divisors of (p − 1).
    if (any q_i is not a prime divisor of (p − 1)) then
        return("no").
    end if
```

FUNCTION PRIMALITY CHECKING(p)

Guess r.
if $(r^{p-1} \not\equiv 1 \mod p)$ **then**
    return("no").
else
    Guess $q_1, q_2, \ldots q_k$ as the prime divisors of $(p-1)$.
    if (any q_i is not a prime divisor of $(p-1)$) **then**
        return("no").
    end if
end if

FUNCTION PRIMALITY CHECKING(p)

Guess r.
if $(r^{p-1} \not\equiv 1 \bmod p)$ **then**
    return("no").
else
    Guess $q_1, q_2, \ldots q_k$ as the prime divisors of $(p-1)$.
    if (any q_i is not a prime divisor of $(p-1)$) **then**
        return("no").
    end if
end if
for $(i = 1$ **to** $k)$ **do**

FUNCTION PRIMALITY CHECKING(p)

Guess r.
if ($r^{p-1} \not\equiv 1 \mod p$) **then**
  return("no").
else
  Guess $q_1, q_2, \ldots q_k$ as the prime divisors of $(p-1)$.
  if (any q_i is not a prime divisor of $(p-1)$) **then**
    return("no").
  end if
end if
for ($i = 1$ **to** k) **do**
  if ($r^{\frac{p-1}{q}} \equiv 1 \mod p$) **then**

FUNCTION PRIMALITY CHECKING(p)

Guess r.
if $(r^{p-1} \not\equiv 1 \mod p)$ **then**
    return("no").
else
    Guess $q_1, q_2, \ldots q_k$ as the prime divisors of $(p-1)$.
    if (any q_i is not a prime divisor of $(p-1)$) **then**
        return("no").
    end if
end if
for $(i = 1$ **to** $k)$ **do**
    if $(r^{\frac{p-1}{q}} \equiv 1 \mod p)$ **then**
        return("no").

FUNCTION PRIMALITY CHECKING(p)

Guess r.
if $(r^{p-1} \not\equiv 1 \mod p)$ **then**
    return("no").
else
    Guess $q_1, q_2, \ldots q_k$ as the prime divisors of $(p-1)$.
    if (any q_i is not a prime divisor of $(p-1)$) **then**
        return("no").
    end if
end if
for $(i = 1$ **to** $k)$ **do**
    if $(r^{\frac{p-1}{q}} \equiv 1 \mod p)$ **then**
        return("no").
    end if

FUNCTION PRIMALITY CHECKING(p)

Guess r.
if $(r^{p-1} \not\equiv 1 \bmod p)$ **then**
  return("no").
else
  Guess $q_1, q_2, \ldots q_k$ as the prime divisors of $(p-1)$.
  if (any q_i is not a prime divisor of $(p-1)$) **then**
    return("no").
  end if
end if
for $(i = 1$ **to** $k)$ **do**
  if $(r^{\frac{p-1}{q}} \equiv 1 \bmod p)$ **then**
    return("no").
  end if
end for

FUNCTION PRIMALITY CHECKING(p)

Guess r.
if $(r^{p-1} \not\equiv 1 \bmod p)$ **then**
  return("no").
else
  Guess $q_1, q_2, \ldots q_k$ as the prime divisors of $(p-1)$.
  if (any q_i is not a prime divisor of $(p-1)$) **then**
    return("no").
  end if
end if
for $(i = 1$ **to** $k)$ **do**
  if $(r^{\frac{p-1}{q}} \equiv 1 \bmod p)$ **then**
    return("no").
  end if
end for
return("yes").

Algorithm 6.17: A nondeterministic algorithm for PRIMES

Details

Details

Hidden details

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p?

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime?

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively!

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r;$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1;$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1);$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2;$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \ldots q_k;$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \ldots q_k; C(q_k))$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \ldots q_k; C(q_k))$$

4. Unless $p = 2$, p will be odd and hence $q_1 = 2$.

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \ldots q_k; C(q_k))$$

4. Unless $p = 2$, p will be odd and hence $q_1 = 2$. So without loss of generality, the certificate for p will have the following form:

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \ldots q_k; C(q_k))$$

4. Unless $p = 2$, p will be odd and hence $q_1 = 2$. So without loss of generality, the certificate for p will have the following form:

$$(r;$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \ldots q_k; C(q_k))$$

4. Unless $p = 2$, p will be odd and hence $q_1 = 2$. So without loss of generality, the certificate for p will have the following form:

$$(r; 2;$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \ldots q_k; C(q_k))$$

4. Unless $p = 2$, p will be odd and hence $q_1 = 2$. So without loss of generality, the certificate for p will have the following form:

$$(r; 2; (1);$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \ldots q_k; C(q_k))$$

4. Unless $p = 2$, p will be odd and hence $q_1 = 2$. So without loss of generality, the certificate for p will have the following form:

$$(r; 2; (1); q_2;$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \ldots q_k; C(q_k))$$

4. Unless $p = 2$, p will be odd and hence $q_1 = 2$. So without loss of generality, the certificate for p will have the following form:

$$(r; 2; (1); q_2; C(q_2) \ldots q_k;$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \ldots q_k; C(q_k))$$

4. Unless $p = 2$, p will be odd and hence $q_1 = 2$. So without loss of generality, the certificate for p will have the following form:

$$(r; 2; (1); q_2; C(q_2) \ldots q_k; C(q_k))$$

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \ldots q_k; C(q_k))$$

4. Unless $p = 2$, p will be odd and hence $q_1 = 2$. So without loss of generality, the certificate for p will have the following form:

$$(r; 2; (1); q_2; C(q_2) \ldots q_k; C(q_k))$$

For instance, the certificate for 67 is:

Details

Hidden details

1. How do we check that the q_i represent all the divisors of p? Repeated division.
2. How do we check that the q_is are prime? Recursively! Guess their certificates as well.
3. Accordingly, the certificate for p, will have the following form:

$$(r; q_1; C(q_1); q_2; C(q_2) \dots q_k; C(q_k))$$

4. Unless $p = 2$, p will be odd and hence $q_1 = 2$. So without loss of generality, the certificate for p will have the following form:

$$(r; 2; (1); q_2; C(q_2) \dots q_k; C(q_k))$$

For instance, the certificate for 67 is:

$(2; 2; (1); 3; (2; 2; (1)); 11; (8; 2; (1); 5; (3; 2; (1)))$.

Theorem

Theorem

Let $\Sigma = \{(,), 0, 1, ;\}$.

Theorem

Let $\Sigma = \{(,),0,1,;\}$*. The size of p's certificate in* Σ *is at most* $4 \cdot \log^2 p$*.*

Theorem

Let $\Sigma = \{(,),0,1,;\}$. *The size of p's certificate in* Σ *is at most* $4 \cdot \log^2 p$.

Proof

Theorem

Let $\Sigma = \{(,),0,1,;\}$*. The size of p's certificate in* Σ *is at most* $4 \cdot \log^2 p$*.*

Proof

1. Clearly true for $p = 2$ and $p = 3$.

Theorem

Let $\Sigma = \{(,),0,1,;\}$*. The size of p's certificate in* Σ *is at most* $4 \cdot \log^2 p$*.*

Proof

1. Clearly true for $p = 2$ and $p = 3$.
2. $q_1, q_2, q_3, \ldots, q_k$ are prime divisors of $(p-1)$ ($k \leq \log p$.).

Theorem

Let $\Sigma = \{(,),0,1,;\}$. *The size of p's certificate in* Σ *is at most* $4 \cdot \log^2 p$.

Proof

1. Clearly true for $p = 2$ and $p = 3$.
2. $q_1, q_2, q_3, \ldots, q_k$ are prime divisors of $(p-1)$ ($k \leq \log p$.). Hence, $q_2 \cdot q_3 \ldots q_k \leq \frac{p-1}{2}$.

Theorem

Let $\Sigma = \{(,),0,1,;\}$*. The size of p's certificate in* Σ *is at most* $4 \cdot \log^2 p$*.*

Proof

1. Clearly true for $p = 2$ and $p = 3$.
2. $q_1, q_2, q_3, \ldots, q_k$ are prime divisors of $(p-1)$ ($k \leq \log p$.). Hence, $q_2 \cdot q_3 \ldots q_k \leq \frac{p-1}{2}$.
3. Total number of symbols needed to represent r is at most $\log p$.

Theorem

Let $\Sigma = \{(,),0,1,;\}$. *The size of p's certificate in* Σ *is at most* $4 \cdot \log^2 p$.

Proof

1. Clearly true for $p = 2$ and $p = 3$.
2. $q_1, q_2, q_3, \ldots, q_k$ are prime divisors of $(p-1)$ ($k \leq \log p$.). Hence, $q_2 \cdot q_3 \ldots q_k \leq \frac{p-1}{2}$.
3. Total number of symbols needed to represent r is at most $\log p$.
4. Total number of symbols needed to represent 2 and its certificate (1) is 5.

Theorem

Let $\Sigma = \{(,), 0, 1, ;\}$. *The size of p's certificate in* Σ *is at most* $4 \cdot \log^2 p$.

Proof

1. Clearly true for $p = 2$ and $p = 3$.
2. $q_1, q_2, q_3, \ldots, q_k$ are prime divisors of $(p-1)$ ($k \leq \log p$.). Hence, $q_2 \cdot q_3 \ldots q_k \leq \frac{p-1}{2}$.
3. Total number of symbols needed to represent r is at most $\log p$.
4. Total number of symbols needed to represent 2 and its certificate (1) is 5.
5. Total number of symbols needed to represent all the q_is, $i = 2, 3, \ldots p$ is at most $2 \cdot (\log(\frac{p-1}{2})) \leq 2 \cdot (\log p - 1)$.

Theorem

Let $\Sigma = \{(,),0,1,;\}$. *The size of p's certificate in* Σ *is at most* $4 \cdot \log^2 p$.

Proof

1. Clearly true for $p = 2$ and $p = 3$.
2. $q_1, q_2, q_3, \ldots, q_k$ are prime divisors of $(p-1)$ ($k \le \log p$.). Hence, $q_2 \cdot q_3 \ldots q_k \le \frac{p-1}{2}$.
3. Total number of symbols needed to represent r is at most $\log p$.
4. Total number of symbols needed to represent 2 and its certificate (1) is 5.
5. Total number of symbols needed to represent all the q_is, $i = 2, 3, \ldots p$ is at most $2 \cdot (\log(\frac{p-1}{2})) \le 2 \cdot (\log p - 1)$.
6. Total number of symbols needed to represent all the delimiters is $2 \cdot k \le 2 \cdot \log p$.

Theorem

Let $\Sigma = \{(,),0,1,;\}$*. The size of p's certificate in* Σ *is at most* $4 \cdot \log^2 p$.

Proof

1. Clearly true for $p = 2$ and $p = 3$.
2. $q_1, q_2, q_3, \ldots, q_k$ are prime divisors of $(p-1)$ ($k \leq \log p$.). Hence, $q_2 \cdot q_3 \ldots q_k \leq \frac{p-1}{2}$.
3. Total number of symbols needed to represent r is at most $\log p$.
4. Total number of symbols needed to represent 2 and its certificate (1) is 5.
5. Total number of symbols needed to represent all the q_is, $i = 2, 3, \ldots p$ is at most $2 \cdot (\log(\frac{p-1}{2})) \leq 2 \cdot (\log p - 1)$.
6. Total number of symbols needed to represent all the delimiters is $2 \cdot k \leq 2 \cdot \log p$.
7. Total number of parentheses is 2.

Theorem

Let $\Sigma = \{(,),0,1,;\}$*. The size of p's certificate in* Σ *is at most* $4 \cdot \log^2 p$*.*

Proof

1. Clearly true for $p = 2$ and $p = 3$.
2. $q_1, q_2, q_3, \ldots, q_k$ are prime divisors of $(p-1)$ ($k \leq \log p$.). Hence, $q_2 \cdot q_3 \ldots q_k \leq \frac{p-1}{2}$.
3. Total number of symbols needed to represent r is at most $\log p$.
4. Total number of symbols needed to represent 2 and its certificate (1) is 5.
5. Total number of symbols needed to represent all the q_is, $i = 2, 3, \ldots p$ is at most $2 \cdot (\log(\frac{p-1}{2})) \leq 2 \cdot (\log p - 1)$.
6. Total number of symbols needed to represent all the delimiters is $2 \cdot k \leq 2 \cdot \log p$.
7. Total number of parentheses is 2.
8. By induction $|C(q_i)| \leq 4 \cdot \log^2 q_i$.

Proof

Proof

It follows that:

$$|C(p)| \leq \log p + 5 + 2 \cdot (\log p - 1) + 2 \cdot \log p + 2 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i$$

Proof

It follows that:

$$\begin{aligned} |C(p)| &\leq \log p + 5 + 2 \cdot (\log p - 1) + 2 \cdot \log p + 2 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\ &\leq 5 \cdot \log p + 5 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \end{aligned}$$

Proof

It follows that:

$$
\begin{aligned}
|C(p)| &\leq \log p + 5 + 2 \cdot (\log p - 1) + 2 \cdot \log p + 2 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\sum_{i=2}^{k} \log q_i)^2
\end{aligned}
$$

Proof

It follows that:

$$
\begin{aligned}
|C(p)| &\leq \log p + 5 + 2 \cdot (\log p - 1) + 2 \cdot \log p + 2 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\sum_{i=2}^{k} \log q_i)^2 \\
&= 5 \cdot \log p + 5 + 4 \cdot \log^2(q_2 \cdot \ldots \cdot q_k)
\end{aligned}
$$

Proof

It follows that:

$$\begin{aligned}
|C(p)| &\leq \log p + 5 + 2 \cdot (\log p - 1) + 2 \cdot \log p + 2 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\sum_{i=2}^{k} \log q_i)^2 \\
&= 5 \cdot \log p + 5 + 4 \cdot \log^2 (q_2 \cdot \ldots \cdot q_k) \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\log \frac{p-1}{2})^2
\end{aligned}$$

Proof

It follows that:

$$
\begin{aligned}
|C(p)| &\leq \log p + 5 + 2 \cdot (\log p - 1) + 2 \cdot \log p + 2 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\sum_{i=2}^{k} \log q_i)^2 \\
&= 5 \cdot \log p + 5 + 4 \cdot \log^2 (q_2 \cdot \ldots \cdot q_k) \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\log \frac{p-1}{2})^2 \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\log p - 1)^2
\end{aligned}
$$

Proof

It follows that:

$$
\begin{aligned}
|C(p)| &\leq \log p + 5 + 2 \cdot (\log p - 1) + 2 \cdot \log p + 2 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\sum_{i=2}^{k} \log q_i)^2 \\
&= 5 \cdot \log p + 5 + 4 \cdot \log^2 (q_2 \cdot \ldots \cdot q_k) \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\log \frac{p-1}{2})^2 \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\log p - 1)^2 \\
&\leq 4 \log^2 p + 9 - 3 \cdot \log p
\end{aligned}
$$

Proof

It follows that:

$$
\begin{aligned}
|C(p)| &\leq \log p + 5 + 2 \cdot (\log p - 1) + 2 \cdot \log p + 2 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\sum_{i=2}^{k} \log q_i)^2 \\
&= 5 \cdot \log p + 5 + 4 \cdot \log^2 (q_2 \cdot \ldots \cdot q_k) \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\log \frac{p-1}{2})^2 \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\log p - 1)^2 \\
&\leq 4 \log^2 p + 9 - 3 \cdot \log p \\
&\leq 4 \log^2 p,
\end{aligned}
$$

Proof

It follows that:

$$
\begin{aligned}
|C(p)| &\leq \log p + 5 + 2 \cdot (\log p - 1) + 2 \cdot \log p + 2 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot \sum_{i=2}^{k} \log^2 q_i \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\sum_{i=2}^{k} \log q_i)^2 \\
&= 5 \cdot \log p + 5 + 4 \cdot \log^2 (q_2 \cdot \ldots \cdot q_k) \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\log \frac{p-1}{2})^2 \\
&\leq 5 \cdot \log p + 5 + 4 \cdot (\log p - 1)^2 \\
&\leq 4 \log^2 p + 9 - 3 \cdot \log p \\
&\leq 4 \log^2 p, \text{ when } p \geq 5.
\end{aligned}
$$

Binary alphabet

Binary alphabet

How many bits one needs in order to represent p's certificate?

Binary alphabet

How many bits one needs in order to represent p's certificate?

Theorem

Let $\Sigma' = \{\sigma_1, ..., \sigma_t\}$ be any alphabet with $|\Sigma'| \geq 2$, and let x be a string in Σ'.

Binary alphabet

How many bits one needs in order to represent p's certificate?

Theorem

Let $\Sigma' = \{\sigma_1, ..., \sigma_t\}$ be any alphabet with $|\Sigma'| \geq 2$, and let x be a string in Σ'. Then x can be represented using $|x| \cdot \log |\Sigma'|$ bits,

Binary alphabet

How many bits one needs in order to represent *p*'s certificate?

Theorem

Let $\Sigma' = \{\sigma_1, ..., \sigma_t\}$ *be any alphabet with* $|\Sigma'| \geq 2$*, and let* x *be a string in* Σ'*. Then* x *can be represented using* $|x| \cdot \log|\Sigma'|$ *bits, where* $|x|$ *is the number of symbols from* Σ' *present in* x*.*

Binary alphabet

How many bits one needs in order to represent p's certificate?

Theorem

Let $\Sigma' = \{\sigma_1, ..., \sigma_t\}$ be any alphabet with $|\Sigma'| \geq 2$, and let x be a string in Σ'. Then x can be represented using $|x| \cdot \log |\Sigma'|$ bits, where $|x|$ is the number of symbols from Σ' present in x.

Corollary

Binary alphabet

How many bits one needs in order to represent p's certificate?

Theorem

Let $\Sigma' = \{\sigma_1, \ldots, \sigma_t\}$ be any alphabet with $|\Sigma'| \geq 2$, and let x be a string in Σ'. Then x can be represented using $|x| \cdot \log |\Sigma'|$ bits, where $|x|$ is the number of symbols from Σ' present in x.

Corollary

p's certificate requires at most $12 \cdot \log^2 p$ bits.