NP-completeness

Serge Gaspers

UNSW

Outline

Outline

Polynomial time

Polynomial-time algorithm

Polynomial-time algorithm:
There exists a constant $c \in \mathbb{N}$ such that the algorithm has (worst-case) running-time $O(n^c)$, where n is the size of the input.

Polynomial time

Polynomial-time algorithm

Polynomial-time algorithm:
There exists a constant $c \in \mathbb{N}$ such that the algorithm has (worst-case) running-time $O(n^c)$, where n is the size of the input.

Example

Polynomial: n; $n^2 \log_2 n$; n^3; n^{20}
Super-polynomial: $n^{\log_2 n}$; $2^{\sqrt{n}}$; 1.001^n; 2^n; $n!$

Tractable problems

Central Question

Which computational problems have polynomial-time algorithms?

Million-dollar question

Intriguing class of problems: NP-complete problems.

NP-complete problems

It is unknown whether NP-complete problems have polynomial-time algorithms.

- A polynomial-time algorithm for one NP-complete problem would imply polynomial-time algorithms for all problems in NP.

Gerhard Woeginger's P vs NP page:
http://www.win.tue.nl/~gwoegi/P-versus-NP.htm

Polynomial vs. NP-complete

Polynomial

- Shortest Path: Given a graph G, two vertices a and b of G, and an integer k, does G have a simple a–b-path of length at most k?
- Euler Tour: Given a graph G, does G have a cycle that traverses each edge of G exactly once?
- 2-CNF SAT: Given a propositional formula F in 2-CNF, is F satisfiable?
  A k-CNF formula is a conjunction (AND) of clauses, and each clause is a disjunction (OR) of at most k literals, which are negated or unnegated Boolean variables.

NP-complete

- Longest Path: Given a graph G and an integer k, does G have a simple path of length at least k?
- Hamiltonian Cycle: Given a graph G, does G have a simple cycle that visits each vertex of G?
- 3-CNF SAT: Given a propositional formula F in 3-CNF, is F satisfiable?
  Example:
  $(x \vee \neg y \vee z) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$.

Overview

What's next?

- Formally define P, NP, and NP-complete (NPC)
- (New) skill: show that a problem is NP-complete

Outline

Decision problems and Encodings

<Name of Decision Problem>

Input:	<What constitutes an instance>
Question:	<Yes/No question>

Decision problems and Encodings

<Name of Decision Problem>
Input: <What constitutes an instance>
Question: <Yes/No question>

We want to know which decision problems can be solved in polynomial time – polynomial in the size of the input n.

- Assume a “reasonable” encoding of the input
- Many encodings are polynomial-time equivalent; i.e., one encoding can be computed from another in polynomial time.
- Important exception: unary versus binary encoding of integers.
  - An integer x takes $\lceil \log_2 x \rceil$ bits in binary and $x = 2^{\log_2 x}$ bits in unary.

Formal-language framework

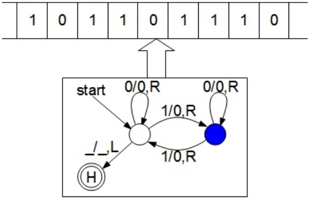
We can view decision problems as languages.

- Alphabet Σ: finite set of symbols. W.l.o.g., $\Sigma = \{0, 1\}$
- Language L over Σ: set of strings made with symbols from Σ: $L \subseteq \Sigma^*$
- Fix an encoding of instances of a decision problem Π into Σ
- Define the language $L_\Pi \subseteq \Sigma^*$ such that

$$x \in L_\Pi \Leftrightarrow x \text{ is a Yes-instance for } \Pi$$

Non-deterministic Turing Machine (NTM)

- input word $x \in \Sigma^*$ placed on an infinite tape (memory)
- read-write head initially placed on the first symbol of x
- computation step: if the machine is in state s and reads a, it can move into state s', writing b, and moving the head into direction $D \in \{L, R\}$ if $((s,a),(s',b,D)) \in \delta$.



- Q: finite, non-empty set of states
- Γ: finite, non-empty set of tape symbols
- $\_ \in \Gamma$: blank symbol (the only symbol allowed to occur on the tape infinitely often)
- $\Sigma \subseteq \Gamma \setminus \{b\}$: set of input symbols
- $q_0 \in Q$: start state
- $A \subseteq Q$: set of accepting (final) states
- $\delta \subseteq (Q \setminus A \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$: transition relation, where L stands for a move to the left and R for a move to the right.

Accepted Language

Definition 1

A NTM accepts a word $x \in \Sigma^*$ if there exists a sequence of computation steps starting in the start state and ending in an accept state.

Definition 2

The language accepted by an NTM is the set of words it accepts.

Acceptance in polynomial time

Definition 3

A language L is accepted in polynomial time by an NTM M if

- L is accepted by M, and
- there is a constant k such that for any word $x \in L$, the NTM M accepts x in $O(|x|^k)$ computation steps.

Deterministic Turing Machine

Definition 4

A Deterministic Turing Machine (DTM) is a Non-deterministic Turing Machine where the transition relation contains at most one tuple $((s,a),(\cdot,\cdot,\cdot))$ for each $s \in Q \setminus A$ and $a \in \Gamma$.

The transition relation δ can be viewed as a function
$\delta : Q \setminus A \times \Gamma \to Q \times \Gamma \times \{L,R\}$.
$\Rightarrow$ For a given input word $x \in \Sigma^*$, there is exactly one sequence of computation steps starting in the start state.

DTM equivalents

Many computational models are polynomial-time equivalent to DTMs:

- Random Access Machine (RAM, used for algorithms in the textbook)
- variants of Turing machines (multiple tapes, infinite only in one direction, ...)
- ...

P and NP

Definition 5 (P)

P $= \{L \subseteq \Sigma^* :$ there is a DTM accepting L in polynomial time$\}$

Definition 6 (NP)

NP $= \{L \subseteq \Sigma^* :$ there is a NTM accepting L in polynomial time$\}$

Definition 7 (coNP)

coNP $= \{L \subseteq \Sigma^* : \Sigma^* \setminus L \in \text{NP}\}$

coP?

Theorem 8

If $L \in \mathsf{P}$, then there is a polynomial-time DTM that halts in an accepting state on every word in L and it halts in a non-accepting state on every word not in L.

coP?

Theorem 8

If $L \in \mathsf{P}$, then there is a polynomial-time DTM that halts in an accepting state on every word in L and it halts in a non-accepting state on every word not in L.

Proof sketch.

Suppose $L \in \mathsf{P}$. By the definition of P, there is a DTM M that accepts L in polynomial time.
Idea: design a DTM M' that simulates M for $c \cdot n^k$ steps, where $c \cdot n^k$ is the running time of M and transitions to a non-accepting state if M does not halt in an accepting state.
(Note that this proof is nonconstructive: we might not know the running time of M.) $\square$

NP and certificates

Non-deterministic choices

A NTM for an NP-language L makes a polynomial number of non-deterministic choices on input $x \in L$.
We can encode these non-deterministic choices into a certificate c, which is a polynomial-length word.
Now, there exists a DTM, which, given x and c, verifies that $x \in L$ in polynomial time.

Thus, $L \in$ NP iff there is a DTM V and for each $x \in L$ there exists a polynomial-length certificate c such that $V(x, c) = 1$, but $V(y, \cdot) = 0$ for each $y \notin L$.

CNF-SAT is in NP

- A CNF formula is a propositional formula in conjunctive normal form: a conjunction (AND) of clauses; each clause is a disjunction (OR) of literals; each literal is a negated or unnegated Boolean variable.
- An assignment $\alpha : \text{var}(F) \to \{0, 1\}$ satisfies a clause C if it sets a literal of C to true, and it satisfies F if it satisfies all clauses in F.

CNF-SAT

Input:	CNF formula F
Question:	Does F have a satisfying assignment?

Example: $(x \vee \neg y \vee z) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$.

Lemma 9

CNF-SAT $\in$ NP.

CNF-SAT is in NP

- A CNF formula is a propositional formula in conjunctive normal form: a conjunction (AND) of clauses; each clause is a disjunction (OR) of literals; each literal is a negated or unnegated Boolean variable.
- An assignment $\alpha : \mathrm{var}(F) \to \{0, 1\}$ satisfies a clause C if it sets a literal of C to true, and it satisfies F if it satisfies all clauses in F.

CNF-SAT	
Input:	CNF formula F
Question:	Does F have a satisfying assignment?

Example: $(x \vee \neg y \vee z) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$.

Lemma 9

CNF-SAT $\in$ NP.

Proof.

Certificate: assignment α to the variables.
Given a certificate, it can be checked in polynomial time whether all clauses are satisfied. $\square$

Brute-force algorithms for problems in NP

Theorem 10

Every problem in NP can be solved in exponential time.

Brute-force algorithms for problems in NP

Theorem 10

Every problem in NP can be solved in exponential time.

Proof.

Let Π be an arbitrary problem in NP. [Use certificate-based definition of NP]
We know that $\exists$ a polynomial p and a polynomial-time verification algorithm V such that:

- for every $x \in \Pi$ (i.e., every YES-instance for Π) $\exists$ string $c \in \{0,1\}^*$, $|c| \le p(|x|)$, such that $V(x,c) = 1$, and
- for every $x \notin \Pi$ (i.e., every NO-instance for Π) and every string $c \in \{0,1\}^*$, $V(x,c) = 0$.

Brute-force algorithms for problems in NP

Theorem 10

Every problem in NP can be solved in exponential time.

Proof.

Let Π be an arbitrary problem in NP. [Use certificate-based definition of NP]
We know that $\exists$ a polynomial p and a polynomial-time verification algorithm V such that:

- for every $x \in \Pi$ (i.e., every YES-instance for Π) $\exists$ string $c \in \{0,1\}^*$, $|c| \le p(|x|)$, such that $V(x,c) = 1$, and
- for every $x \notin \Pi$ (i.e., every NO-instance for Π) and every string $c \in \{0,1\}^*$, $V(x,c) = 0$.

Now, we can prove there exists an exponential-time algorithm for Π with input x:

- For each string $c \in \{0,1\}^*$ with $|c| \le p(|x|)$, evaluate $V(x,c)$ and return YES if $V(x,c) = 1$.
- Return NO.

Running time: $2^{p(|x|)} \cdot n^{O(1)} \subseteq 2^{O(2 \cdot p(|x|))} = 2^{O(p(|x|))}$, but non-constructive. $\square$

Outline

Polynomial-time reduction

Definition 11

A language L_1 is polynomial-time reducible to a language L_2, written $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ such that for all $x \in \Sigma^*$,

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$

A polynomial time algorithm computing f is a reduction algorithm.

New polynomial-time algorithms via reductions

Lemma 12

If $L_1, L_2 \in \Sigma^*$ *are languages such that* $L_1 \le_P L_2$*, then* $L_2 \in \mathsf{P}$ *implies* $L_1 \in \mathsf{P}$*.*

NP-completeness

Definition 13 (NP-hard)

A language $L \subseteq \Sigma^*$ is NP-hard if

$$L' \leq_P L \text{ for every } L' \in \text{NP}.$$

Definition 14 (NP-complete)

A language $L \subseteq \Sigma^*$ is NP-complete (in NPC) if

1. $L \in$ NP, and
2. L is NP-hard.

A first NP-complete problem

Theorem 15

CNF-SAT is NP-complete.

Proved by encoding NTMs into SAT (Cook, 1971; Levin, 1973) and then CNF-SAT (Karp, 1972).

Proving NP-completeness

Lemma 16

If L *is a language such that* $L' \leq_P L$ *for some* $L' \in$ NPC*, then* L *is* NP*-hard.*
If, in addition, $L \in$ NP*, then* $L \in$ NPC*.*

Proving NP-completeness

Lemma 16

If L is a language such that $L' \leq_P L$ for some $L' \in$ NPC, then L is NP-hard. If, in addition, $L \in$ NP, then $L \in$ NPC.

Proof.

For all $L'' \in$ NP, we have $L'' \leq_P L' \leq_P L$.
By transitivity, we have $L'' \leq_P L$.
Thus, L is NP-hard. □

Proving NP-completeness (2)

Method to prove that a language L is NP-complete:

1. Prove $L \in$ NP
2. Prove L is NP-hard.
   - Select a known NP-complete language L'.
   - Describe an algorithm that computes a function f mapping every instance $x \in \Sigma^*$ of L' to an instance $f(x)$ of L.
   - Prove that $x \in L' \Leftrightarrow f(x) \in L$ for all $x \in \Sigma^*$.
   - Prove that the algorithm computing f runs in polynomial time.

Outline

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is in NP, since it is a special case of CNF-SAT.

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is in NP, since it is a special case of CNF-SAT.
To show that 3-CNF SAT is NP-hard, we give a polynomial reduction from CNF-SAT.

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is in NP, since it is a special case of CNF-SAT.
To show that 3-CNF SAT is NP-hard, we give a polynomial reduction from CNF-SAT.
Let F be a CNF formula. The reduction algorithm constructs a 3-CNF formula F' as follows. For each clause C in F:

- If C has at most 3 literals, then copy C into F'.
- Otherwise, denote $C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$.

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is in NP, since it is a special case of CNF-SAT.
To show that 3-CNF SAT is NP-hard, we give a polynomial reduction from CNF-SAT.
Let F be a CNF formula. The reduction algorithm constructs a 3-CNF formula F' as follows. For each clause C in F:

- If C has at most 3 literals, then copy C into F'.
- Otherwise, denote $C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$. Create $k-3$ new variables $y_1, \dots, y_{k-3}$, and add the clauses $(\ell_1 \vee \ell_2 \vee y_1), (\neg y_1 \vee \ell_3 \vee y_2), (\neg y_2 \vee \ell_4 \vee y_3), \dots, (\neg y_{k-3} \vee \ell_{k-1} \vee \ell_k)$.

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is in NP, since it is a special case of CNF-SAT.
To show that 3-CNF SAT is NP-hard, we give a polynomial reduction from CNF-SAT.
Let F be a CNF formula. The reduction algorithm constructs a 3-CNF formula F' as follows. For each clause C in F:

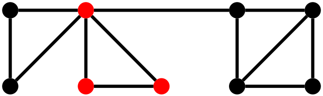
- If C has at most 3 literals, then copy C into F'.
- Otherwise, denote $C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$. Create $k-3$ new variables $y_1, \dots, y_{k-3}$, and add the clauses $(\ell_1 \vee \ell_2 \vee y_1), (\neg y_1 \vee \ell_3 \vee y_2), (\neg y_2 \vee \ell_4 \vee y_3), \dots, (\neg y_{k-3} \vee \ell_{k-1} \vee \ell_k)$.

Show that F is satisfiable $\Leftrightarrow$ F' is satisfiable.
Show that F' can be computed in polynomial time (trivial; use a RAM). $\square$

Clique

A clique in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that every two vertices of S are adjacent in G.

CLIQUE	
Input:	Graph G, integer k
Question:	Does G have a clique of size k?



Theorem 18

CLIQUE *is* NP*-complete.*

Clique (2)

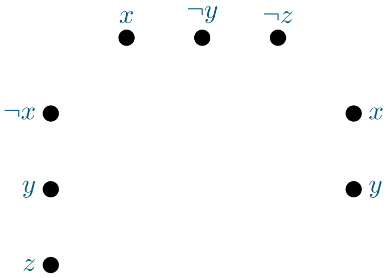
- CLIQUE is in NP

Clique (2)

- Clique is in NP
- Let $F = C_1 \wedge C_2 \wedge \ldots C_k$ be a 3-CNF formula
- Construct a graph G that has a clique of size k iff F is satisfiable

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

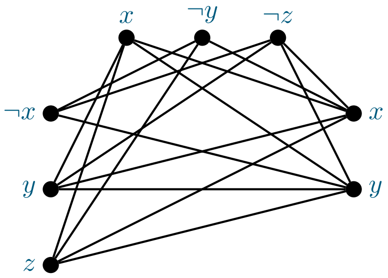
Clique (2)



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

- CLIQUE is in NP
- Let $F = C_1 \wedge C_2 \wedge \ldots C_k$ be a 3-CNF formula
- Construct a graph G that has a clique of size k iff F is satisfiable
- For each clause $C_r = (\ell_1^r \vee \cdots \vee \ell_w^r)$, $1 \leq r \leq k$, create w new vertices $v_1^r, \ldots, v_w^r$

Clique (2)



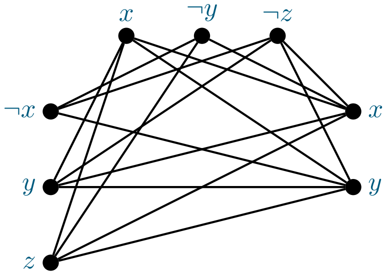
$(\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y)$

- Clique is in NP
- Let $F = C_1 \wedge C_2 \wedge \dots C_k$ be a 3-CNF formula
- Construct a graph G that has a clique of size k iff F is satisfiable
- For each clause $C_r = (\ell_1^r \vee \dots \vee \ell_w^r)$, $1 \le r \le k$, create w new vertices $v_1^r, \dots, v_w^r$
- Add an edge between v_i^r and v_j^s if

$$r \neq s \quad \text{and}$$
$$\ell_i^r \neq \neg \ell_j^s \quad \text{where } \neg\neg x = x.$$

- Check correctness and polynomial running time

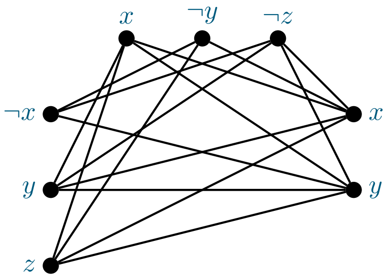
Clique (2)



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

- Correctness: F has a satisfying assignment iff G has a clique of size k.

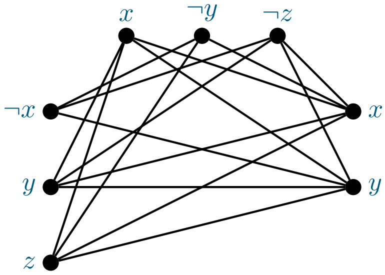
Clique (2)



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

- Correctness: F has a satisfying assignment iff G has a clique of size k.
- $(\Rightarrow)$: Let α be a sat. assignment for F. For each clause C_r, choose a literal ℓ_i^r with $\alpha(\ell_i^r) = 1$, and denote by s^r the corresponding vertex in G. Now, $\{s^r : 1 \leq r \leq k\}$ is a clique of size k in G since $\alpha(x) \neq \alpha(\neg x)$.

Clique (2)



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

- Correctness: F has a satisfying assignment iff G has a clique of size k.
- $(\Rightarrow)$: Let α be a sat. assignment for F. For each clause C_r, choose a literal ℓ_i^r with $\alpha(\ell_i^r) = 1$, and denote by s^r the corresponding vertex in G. Now, $\{s^r : 1 \leq r \leq k\}$ is a clique of size k in G since $\alpha(x) \neq \alpha(\neg x)$.
- $(\Leftarrow)$: Let S be a clique of size k in G. Then, S contains exactly one vertex $s_r \in \{v_1^r, \dots, v_w^r\}$ for each $r \in \{1, \dots, k\}$. Denote by l^r the corresponding literal. Now, for any r, r', it is not the case that $l_r = \neg l_{r'}$. Therefore, there is an assignment α to $\text{var}(F)$ such that $\alpha(l_r) = 1$ for each $r \in \{1, \dots, k\}$ and α satisfies F.

Vertex Cover

A vertex cover in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that every edge of G has an endpoint in S.

Vertex Cover	
Input:	Graph G, integer k
Question:	Does G have a vertex cover of size k?

Theorem 19

Vertex Cover *is* NP*-complete.*

The proof is left as an exercise.

Hamiltonian Cycle

A Hamiltonian Cycle in a graph $G = (V, E)$ is a cycle visiting each vertex exactly once.
(Alternatively, a permutation of V such that every two consecutive vertices are adjacent and the first and last vertex in the permutation are adjacent.)

HAMILTONIAN CYCLE
Input: Graph G
Question: Does G have a Hamiltonian Cycle?

Theorem 20

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch.

Hamiltonian Cycle

A Hamiltonian Cycle in a graph $G = (V, E)$ is a cycle visiting each vertex exactly once.
(Alternatively, a permutation of V such that every two consecutive vertices are adjacent and the first and last vertex in the permutation are adjacent.)

HAMILTONIAN CYCLE
Input: Graph G
Question: Does G have a Hamiltonian Cycle?

Theorem 20

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch.

- HAMILTONIAN CYCLE is in NP: the certificate is a Hamiltonian Cycle of G.

Hamiltonian Cycle

A Hamiltonian Cycle in a graph $G = (V, E)$ is a cycle visiting each vertex exactly once.
(Alternatively, a permutation of V such that every two consecutive vertices are adjacent and the first and last vertex in the permutation are adjacent.)

HAMILTONIAN CYCLE	
Input:	Graph G
Question:	Does G have a Hamiltonian Cycle?

Theorem 20

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch.

- HAMILTONIAN CYCLE is in NP: the certificate is a Hamiltonian Cycle of G.
- Let us show: VERTEX COVER $\leq_P$ HAMILTONIAN CYCLE

. . . □

Hamiltonian Cycle (2)

Theorem 21

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch (continued).

- Let us show: VERTEX COVER $\leq_P$ HAMILTONIAN CYCLE

Hamiltonian Cycle (2)

Theorem 21

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch (continued).

- Let us show: VERTEX COVER $\leq_P$ HAMILTONIAN CYCLE
- Let $(G = (V, E), k)$ be an instance for VERTEX COVER (VC).
- We will construct an equivalent instance G' for HAMILTONIAN CYCLE (HC).

Hamiltonian Cycle (2)

Theorem 21

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch (continued).

- Let us show: VERTEX COVER $\leq_P$ HAMILTONIAN CYCLE
- Let $(G = (V, E), k)$ be an instance for VERTEX COVER (VC).
- We will construct an equivalent instance G' for HAMILTONIAN CYCLE (HC).
- Intuition: Non-deterministic choices
  - for VC: which vertices to select in the vertex cover
  - for HC: which route the cycle takes

...

□

Hamiltonian Cycle (3)

Theorem 22

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch (continued).

- Add k vertices $s_1, \dots, s_k$ to G' (*selector vertices*)

Hamiltonian Cycle (3)

Theorem 22

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch (continued).

- Add k vertices $s_1, \ldots, s_k$ to G' (*selector vertices*)
- Each edge of G will be represented by a gadget (subgraph) of G'
- s.t. the set of edges covered by a vertex x in G corresponds to a partial cycle going through all gadgets of G' representing these edges.

Hamiltonian Cycle (3)

Theorem 22

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch (continued).

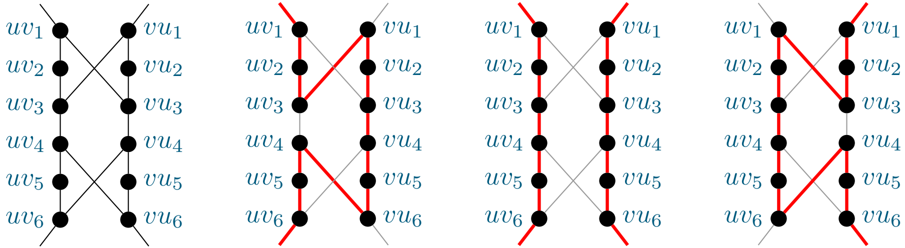
- Add k vertices $s_1, \ldots, s_k$ to G' (*selector vertices*)
- Each edge of G will be represented by a gadget (subgraph) of G'
- s.t. the set of edges covered by a vertex x in G corresponds to a partial cycle going through all gadgets of G' representing these edges.
- Attention: we need to allow for an edge to be covered by both endpoints

. . . □

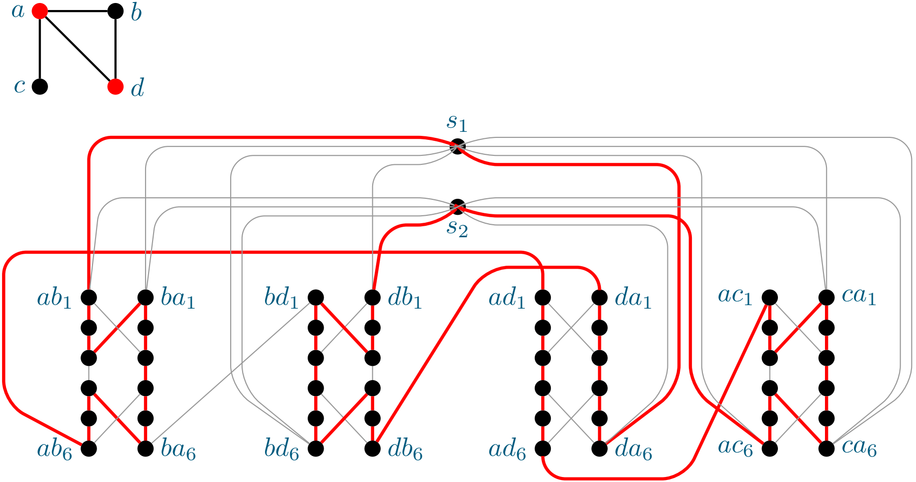
Hamiltonian Cycle (4)

Gadget representing the edge $\{u, v\} \in E$

Its states: 'covered by u', 'covered by u and v', 'covered by v'



Hamiltonian Cycle (5)



Outline

Further Reading

- Chapter 34, **NP-Completeness**, in (Cormen et al., 2009)
- Garey and Johnson's influential reference book (Garey and Johnson, 1979)

References I

Stephen A. Cook (1971). "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971)*, pp. 151–158.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2009). *Introduction to Algorithms*. 3rd ed. The MIT Press.

Michael R. Garey and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

Richard M. Karp (1972). "Reducibility among combinatorial problems". In: *Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972)*. New York: Plenum, pp. 85–103.

Leonid Levin (1973). "Universal sequential search problems". In: *Problems of Information Transmission* 9.3, pp. 265–266.