COMP3121 Lecture – Week 11

Computational Intractability

Serge Gaspers
LiC: Aleks Ignjatovic

[1]School of Computer Science and Engineering, UNSW Australia
[2]Optimisation Group, Decision Sciences, Data61, CSIRO

Outline

1. Overview
2. Turing Machines, P, and NP
3. Reductions and NP-completeness
4. NP-complete problems
5. Extended class 3821/9801

Resources

- Chapter 34, **NP-Completeness**, in the textbook: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms. The MIT Press, 3rd edition, 2009.
- Slides: `http://www.cse.unsw.edu.au/~sergeg/np.pdf`

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; $n!$

Time complexities

$n =$	10	100	1,000	10,000	100,000	1,000,000
n	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms	< 1 ms
$n^2 \log_2 n$	< 1 ms	< 1 ms	< 1 ms	13 ms	1.66 sec	3.3 min
n^3	< 1 ms	< 1 ms	10 ms	10 sec	2.78 hours	3.86 months
n^{20}	31.7 years	> 1 U	> 1 U	> 1 U	> 1 U	> 1 U
$2^{\sqrt{n}}$	< 1 ms	< 1 ms	33 ms	> 1 U	> 1 U	> 1 U
1.001^n	< 1 ms	< 1 ms	< 1 ms	< 1 ms	> 1 U	> 1 U
2^n	< 1 ms	> 1 U	> 1 U	> 1 U	> 1 U	> 1 U
$n!$	< 1 ms	> 1 U	> 1 U	> 1 U	> 1 U	> 1 U

Table: Processing speed for various time complexities, assuming 10^{11} instructions are processed per second (Intel Core i7). Here, U$= 13.798 \cdot 10^9$ years.

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
- Briefly: what to do when confronted with an NP-complete problem?

Outline

1 Overview

2 Turing Machines, P, and NP

3 Reductions and NP-completeness

4 NP-complete problems

5 Extended class 3821/9801

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.

Exercise on Decision Problems

Cluster into groups of 4-5 students. Answer the following questions.
Let $f : \mathbb{N} \to \mathbb{N}$ be a non-decreasing function.

1. Given an $O(f(n))$-time algorithm for Maximum Independent Set, design an algorithm for Independent Set with running time $O(f(n) \cdot \text{poly}(n))$.
2. Given an $O(f(n))$-time algorithm for Independent Set, design an algorithm for Maximum Independent Set with running time $O(f(n) \cdot \text{poly}(n))$.

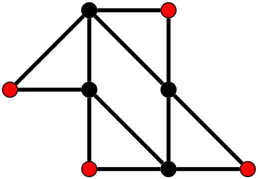
Independent Set

Input:	Graph G, integer k
Question:	Does G have an independent set of size at least k?

Maximum Independent Set

Input:	Graph G
Output:	A largest independent set of G

Def. An independent set of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that no two vertices of S are adjacent in G.



Formal-language framework

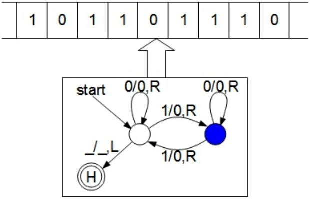
We can view decision problems as languages.

- 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.

Discussion: Non-deterministic Turing Machines

In groups, discuss whether you think that NTMs are realistic computation models

- Is this a good representation of how our computing devices work?
- What is different?

Video

The LEGO Turing Machine
https://www.youtube.com/watch?v=cYw2ewoO6c4

Accept and Decide 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.

Definition 4

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

- there is a constant k such that for any word $x \in L$, the NTM M accepts x in $O(|x|^k)$ computation steps, and
- there is a constant k' such that for any word $x \in \Sigma^* \setminus L$, on input x the NTM M halts in a non-accepting state ($Q \setminus A$) in $O(|x|^{k'})$ computation steps.

Deterministic Turing Machine

Definition 5

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
$\delta : Q \setminus A \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$.
$\Rightarrow$ For a given input word $x \in \Sigma^*$, there is exactly one sequence of computation steps starting in the start state.

Exercise: DTM

In groups:
Design a DTM $(Q, \Gamma, \Sigma = \{0,1\}, q_0, A, \delta)$ that accepts palindromes.
A palindrome is a word that is equal to its reverse; e.g., 011010110.
Recall:

- 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
- $\delta : Q \setminus A \times \Gamma \to Q \times \Gamma \times \{L, R\}$: transition function, where L stands for a move to the left and R for a move to the right.

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 6 (P)

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

Definition 7 (NP)

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

Definition 8 (coNP)

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

coP?

Theorem 9

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

coP?

Theorem 9

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

Proof sketch.

Need to show:
if L is accepted by a DTM M in polynomial time, then there is a DTM that decides 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.
(Note that this proof is nonconstructive: we might not know the running time of M.) □

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 $x \in L$.

Thus, $L \in$ NP iff for each $x \in L$ there exists a polynomial-length certificate c and a DTM M such that given x and a, M can verify in polynomial time that $x \in L$.

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 $\alpha : \text{var}(F) \rightarrow \{0, 1\}$ 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 10

CNF-SAT $\in$ NP.

Proof.

Exercise. □

Outline

1. Overview
2. Turing Machines, P, and NP
3. Reductions and NP-completeness
4. NP-complete problems
5. Extended class 3821/9801

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 $f : \Sigma^* \rightarrow \Sigma^*$ 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* $L_1 \leq_P L_2$*, then* $L_2 \in \mathsf{P}$ *implies* $L_1 \in \mathsf{P}$*.*

Proof.

Exercise. □

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 and then CNF-SAT (Cook–Levin 1971/1973 and 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

1 Overview

2 Turing Machines, P, and NP

3 Reductions and NP-completeness

4 NP-complete problems

5 Extended class 3821/9801

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 $y_1, \ldots, y_{k-3}$, and add the clauses $(\ell_1 \vee \ell_2 \vee y_1), (\neg y_1 \vee \ell_3 \vee y_2), (\neg y_2 \vee \ell_4 \vee y_3), \ldots, (\neg y_{k-3} \vee \ell_{k-1} \vee \ell_k)$.

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:

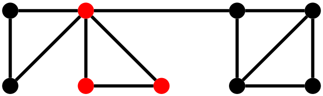
- If C has at most 3 literals, then copy C into F'.
- Otherwise, denote $C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$. Create $k-3$ new variables $y_1, \dots, y_{k-3}$, and add the clauses $(\ell_1 \vee \ell_2 \vee y_1), (\neg y_1 \vee \ell_3 \vee y_2), (\neg y_2 \vee \ell_4 \vee y_3), \dots, (\neg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

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

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.

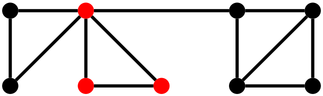
Groupwork.

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.

Groupwork.

Hint: Reduce from 3-CNF SAT.

Clique (2)

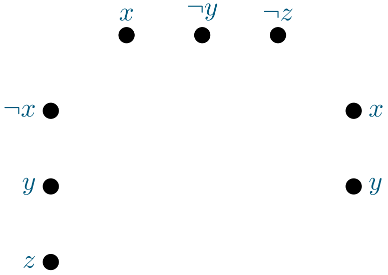
- Clique is in NP

Clique (2)

- 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

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

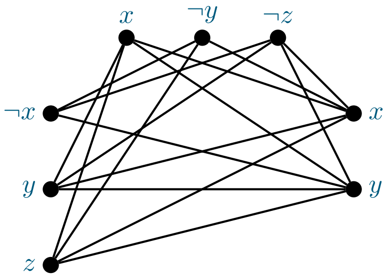
Clique (2)



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

- 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)



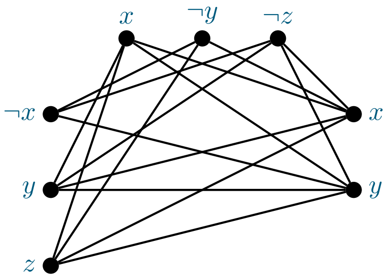
$(\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y)$

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

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

- Check correctness and polynomial running time

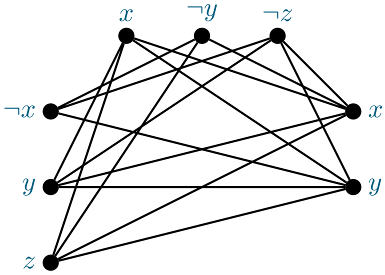
Clique (2)



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

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

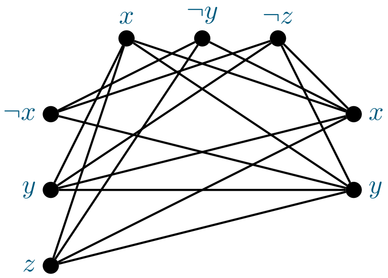
Clique (2)



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

- 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 $s_r \in \{v_1^r, \ldots, v_w^r\}$ for each $r \in \{1, \ldots, k\}$. 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 $r \in \{1, \ldots, k\}$ 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.

Groupwork.

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.

Groupwork.
Hint: Reduce from Clique.

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.

Groupwork.
Hint: Reduce from Clique.
Hint 2: The complement of $G = (V, E)$ is the graph $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{\{u, v\} : u, v \in V \text{ and } \{u, v\} \notin E\}$.

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 $s_1, \ldots, s_k$ to G' (*selector vertices*)

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*)
- 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).

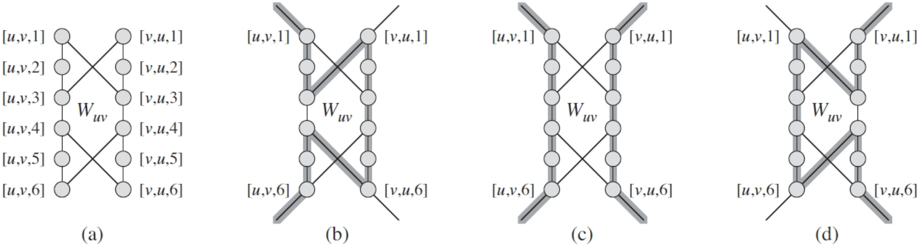
- Add k vertices $s_1, \ldots, s_k$ to G' (*selector vertices*)
- Each edge of G will be represented by a gadget (subgraph) of G'
- s.t. the set of edges covered by a vertex x in G corresponds to a partial cycle going through all gadgets of G' representing these edges.
- Attention: we need to allow for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

□

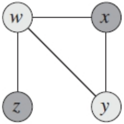
Hamiltonian Cycle (4)

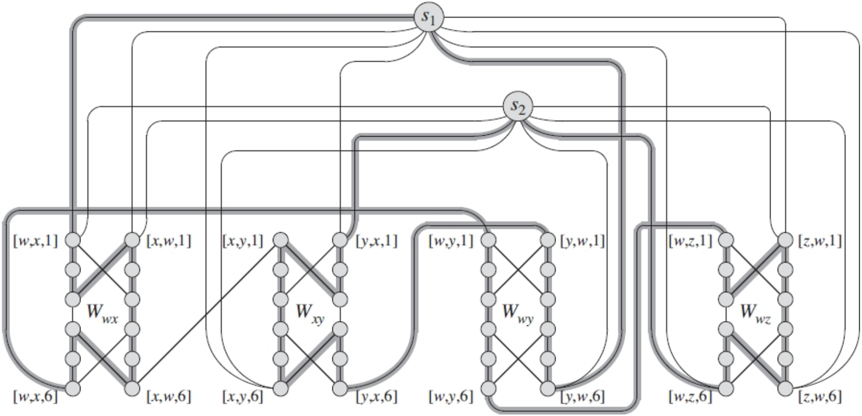
Gadget representing the edge $\{u, v\} \in E$

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



Hamiltonian Cycle (5)





Subset Sum

Subset Sum	
Input:	Set of positive integers S, target integer t
Question:	Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

On your own: read the NP-completeness proof of Subsection 34.5.5 in Chapter 34 of the textbook; stop at any time to see if you can finish it on your own.

Coping with NP-hardness

- Approximation algorithms
  - There is an algorithm, which, given an instance (G, k) for Vertex Cover, finds a vertex cover of size at most $2k$ or correctly determines that G has no vertex cover of size k.
- Exact exponential time algorithms
  - There is an algorithm solving Vertex Cover in time $O(1.2002^n)$, where $n = |V|$.
- Fixed parameter algorithms
  - There is an algorithm solving Vertex Cover in time $O(1.2738^k + kn)$.
- Heuristics
  - Heuristic A finds a smaller vertex cover than Heuristic B on benchmark instances $C_1, \dots, C_m$.
- Restricting the inputs
  - Vertex Cover can be solved in polynomial time on bipartite graphs, trees, interval graphs, etc.

Advertisements

- Algorithms @ UNSW
  http://www.cse.unsw.edu.au/~algo/
- COMP6741 - Parameterized and Exact Computation
  http://www.cse.unsw.edu.au/~cs6741/

Outline

1. Overview
2. Turing Machines, P, and NP
3. Reductions and NP-completeness
4. NP-complete problems
5. Extended class 3821/9801

Algorithm for Subset Sum

Subset Sum	
Input:	Set of positive integers S, target integer t
Question:	Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

Algorithm for Subset Sum

Subset Sum	
Input:	Set of positive integers S, target integer t
Question:	Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

- Dynamic Programming algorithm
- Denote $S = \{s_1, \ldots, s_n\}$
- Table $T[0..n, 0..t]$

Algorithm for Subset Sum

Subset Sum	
Input:	Set of positive integers S, target integer t
Question:	Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

- Dynamic Programming algorithm
- Denote $S = \{s_1, \dots, s_n\}$
- Table $T[0..n, 0..t]$

$$T[i,r] = \begin{cases} \text{true} & \text{if } \exists X \subseteq \{s_1, \dots, s_i\} : \sum_{x \in X} x = r \\ \text{false} & \text{otherwise} \end{cases}$$

Algorithm for Subset Sum

Subset Sum

Input:	Set of positive integers S, target integer t
Question:	Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

- Dynamic Programming algorithm
- Denote $S = \{s_1, \ldots, s_n\}$
- Table $T[0..n, 0..t]$

$$T[i,r] = \begin{cases} \text{true} & \text{if } \exists X \subseteq \{s_1, \ldots, s_i\} : \sum_{x \in X} x = r \\ \text{false} & \text{otherwise} \end{cases}$$

- bases cases... DP recurrence... running time

Algorithm for Subset Sum

Subset Sum

Input:	Set of positive integers S, target integer t
Question:	Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

- Dynamic Programming algorithm
- Denote $S = \{s_1, \dots, s_n\}$
- Table $T[0..n, 0..t]$

$$T[i,r] = \begin{cases} \text{true} & \text{if } \exists X \subseteq \{s_1, \dots, s_i\} : \sum_{x \in X} x = r \\ \text{false} & \text{otherwise} \end{cases}$$

- bases cases... DP recurrence... running time

Subset Sum can be solved in time $O(n \cdot t)$

Algorithm for Subset Sum

Subset Sum
Input: Set of positive integers S, target integer t
Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

- Dynamic Programming algorithm
- Denote $S = \{s_1, \dots, s_n\}$
- Table $T[0..n, 0..t]$

$$T[i,r] = \begin{cases} \text{true} & \text{if } \exists X \subseteq \{s_1,\dots,s_i\} : \sum_{x\in X} x = r \\ \text{false} & \text{otherwise} \end{cases}$$

- bases cases... DP recurrence... running time

Subset Sum can be solved in time $O(n \cdot t)$ (pseudo-polynomial algorithm).

Weak vs Strong NP-completeness

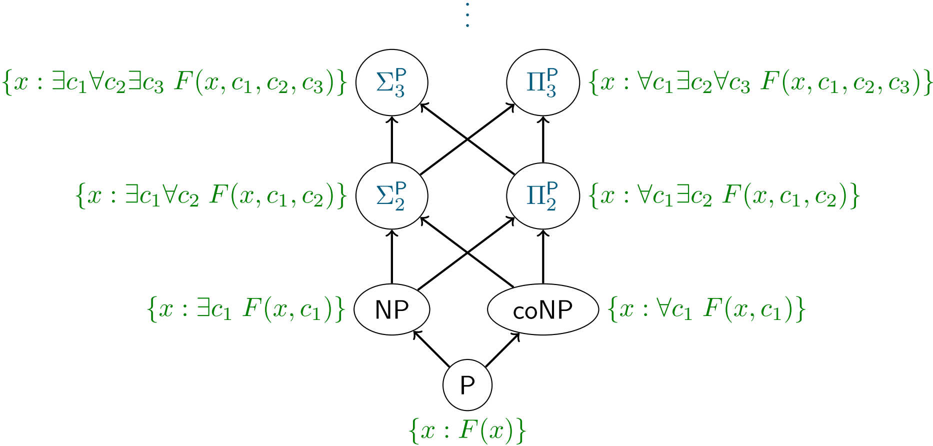
For problems whose input contains integers:

- Weakly NP-hard = NP-hard
- Strongly NP-hard = NP-hard, even if the integers in the input are represented in unary

P, NP, and certificates

- In the following, F represents poly-time computable predicates (function returning true or false)
- P: class of languages $\{x : F(x)\}$
- NP: class of languages $\{x : \exists c_1 \; F(x, c_1)\}$
- coNP: class of languages $\{x : \forall c_1 \; F(x, c_1)\}$
- where $|c_1| \leq \text{poly}(|x|)$

Polynomial Hierarchy



Oracles

- Oracle for a complexity class Π: solves any problem in Π in one computation step

Oracles

- Oracle for a complexity class Π: solves any problem in Π in one computation step
- NP^{Π}: class of languages accepted in polynomial time by an NTM with access to an oracle for Π

Oracles

- Oracle for a complexity class Π: solves any problem in Π in one computation step
- NP$^{\Pi}$: class of languages accepted in polynomial time by an NTM with access to an oracle for Π
- Alternatively NP$^{\Pi}$: class of languages of the form $\{x : \exists c_1 \; F^{\Pi}(x, c_1)\}$ where F^{Π} is a poly-time computable predicate with access to an oracle for Π

Oracles

- Oracle for a complexity class Π: solves any problem in Π in one computation step
- NP$^{\Pi}$: class of languages accepted in polynomial time by an NTM with access to an oracle for Π
- Alternatively NP$^{\Pi}$: class of languages of the form $\{x : \exists c_1 \ F^{\Pi}(x, c_1)\}$ where F^{Π} is a poly-time computable predicate with access to an oracle for Π
- coNP$^{\Pi}$: class of languages of the form $\{x : \forall c_1 \ F^{\Pi}(x, c_1)\}$

Oracles

- Oracle for a complexity class Π: solves any problem in Π in one computation step
- NP$^{\Pi}$: class of languages accepted in polynomial time by an NTM with access to an oracle for Π
- Alternatively NP$^{\Pi}$: class of languages of the form $\{x : \exists c_1 \; F^{\Pi}(x, c_1)\}$ where F^{Π} is a poly-time computable predicate with access to an oracle for Π
- coNP$^{\Pi}$: class of languages of the form $\{x : \forall c_1 \; F^{\Pi}(x, c_1)\}$

$$\Sigma_0^P = \mathsf{P} \qquad \Pi_0^P = \mathsf{P}$$

$$\Sigma_{k+1}^P = \mathsf{NP}^{\Sigma_k^P} \qquad \Pi_{k+1}^P = \mathsf{coNP}^{\Sigma_k^P}$$

Oracles

- Oracle for a complexity class Π: solves any problem in Π in one computation step
- NP^{Π}: class of languages accepted in polynomial time by an NTM with access to an oracle for Π
- Alternatively NP^{Π}: class of languages of the form $\{x : \exists c_1 \ F^{\Pi}(x, c_1)\}$ where F^{Π} is a poly-time computable predicate with access to an oracle for Π
- coNP^{Π}: class of languages of the form $\{x : \forall c_1 \ F^{\Pi}(x, c_1)\}$

$$\Sigma_0^P = \mathsf{P} \qquad \Pi_0^P = \mathsf{P}$$

$$\Sigma_{k+1}^P = \mathsf{NP}^{\Sigma_k^P} \qquad \Pi_{k+1}^P = \mathsf{coNP}^{\Sigma_k^P}$$

All complexity classes in the polynomial hierarchy are closed under $\leq_P$ reductions.

Oracles

- Oracle for a complexity class Π: solves any problem in Π in one computation step
- NP^{Π}: class of languages accepted in polynomial time by an NTM with access to an oracle for Π
- Alternatively NP^{Π}: class of languages of the form $\{x : \exists c_1 \; F^{\Pi}(x, c_1)\}$ where F^{Π} is a poly-time computable predicate with access to an oracle for Π
- coNP^{Π}: class of languages of the form $\{x : \forall c_1 \; F^{\Pi}(x, c_1)\}$

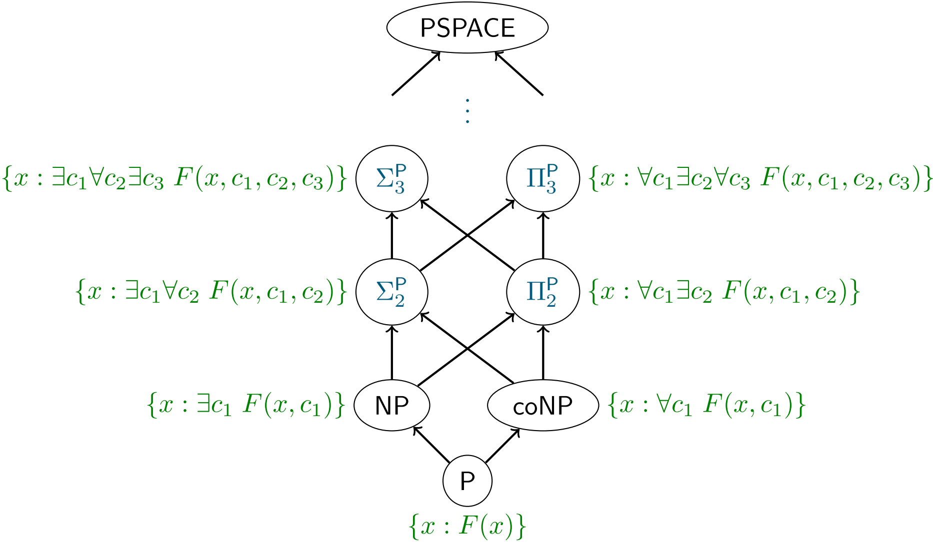
$$\Sigma_0^P = \mathsf{P} \qquad \Pi_0^P = \mathsf{P}$$

$$\Sigma_{k+1}^P = \mathsf{NP}^{\Sigma_k^P} \qquad \Pi_{k+1}^P = \mathsf{coNP}^{\Sigma_k^P}$$

All complexity classes in the polynomial hierarchy are closed under $\leq_P$ reductions.

$$NP^{NP} = NP^{SAT}$$

PSPACE


PSPACE
⋮
$\{x : \exists c_1 \forall c_2 \exists c_3\ F(x, c_1, c_2, c_3)\}$
Σ_3^P
Π_3^P
$\{x : \forall c_1 \exists c_2 \forall c_3\ F(x, c_1, c_2, c_3)\}$
$\{x : \exists c_1 \forall c_2\ F(x, c_1, c_2)\}$
Σ_2^P
Π_2^P
$\{x : \forall c_1 \exists c_2\ F(x, c_1, c_2)\}$
$\{x : \exists c_1\ F(x, c_1)\}$
NP
coNP
$\{x : \forall c_1\ F(x, c_1)\}$
P
$\{x : F(x)\}$

Counting Problems

<Name of Counting Problem>	
Input:	<What constitutes an instance>
Question:	<Number of Yes-instances>

- FP: class of polynomial-time solvable counting problems
- #P: class of counting problems whose solution is the number of accept paths of a polynomial-time Non-deterministic Turing Machine
- Alternatively: a counting problem Π is in #P if there exists a polynomial-time computable function F such that $\Pi(x) = |\{c : F(x, c)\}|$

#P-completeness

- Turing reduction: $\Pi_1 \leq_T \Pi_2$ if there is an algorithm that solves P_1 in polynomial time using an oracle for Π_2
- Π is #P-hard if every problem in #P can be Turing reduced to Π
- Π is #P-complete if Π is in #P and Π is #P-hard.

#P-completeness

- Turing reduction: $\Pi_1 \leq_T \Pi_2$ if there is an algorithm that solves P_1 in polynomial time using an oracle for Π_2
- Π is #P-hard if every problem in #P can be Turing reduced to Π
- Π is #P-complete if Π is in #P and Π is #P-hard.

#CNF-SAT is #P-complete.
#Bipartite-Perfect-Matchings is #P-complete.

#P-completeness

- Turing reduction: $\Pi_1 \leq_T \Pi_2$ if there is an algorithm that solves P_1 in polynomial time using an oracle for Π_2
- Π is #P-hard if every problem in #P can be Turing reduced to Π
- Π is #P-complete if Π is in #P and Π is #P-hard.

#CNF-SAT is #P-complete.
#Bipartite-Perfect-Matchings is #P-complete.

Exercise: Show that #3-CNF-SAT is #P-complete.

#P-completeness

- Turing reduction: $\Pi_1 \leq_T \Pi_2$ if there is an algorithm that solves P_1 in polynomial time using an oracle for Π_2
- Π is #P-hard if every problem in #P can be Turing reduced to Π
- Π is #P-complete if Π is in #P and Π is #P-hard.

#CNF-SAT is #P-complete.
#Bipartite-Perfect-Matchings is #P-complete.

Exercise: Show that #3-CNF-SAT is #P-complete.
Hint: What goes wrong when using our reduction CNF-SAT $\leq_P$ 3-CNF-SAT? How to fix it?