NP-completeness

Algorithms and Networks

Universiteit Utrecht

Today

- Complexity of computational problems
- Formal notion of computations
- NP-completeness
  - Why is it relevant?
  - What is it exactly?
  - How is it proven?
- P vs. NP
- Some animals from the complexity zoo

1

Introduction

Hard problems / Easy problems

- Finding the shortest simple path between vertices *v* and *w* in a given graph
- Determine if there is an Euler tour in a given graph
- Testing 2-colorability
- Satisfiability when each clause has two literals

- Finding the longest simple path between vertices *v* and *w* in a given graph
- Determine if there is a Hamiltonian circuit in a given graph
- Testing 3-colorability
- Satisfiability when each clause has three literals

Fast and slow

- Algorithms whose running time is *polynomial* in input size

- Algorithms whose running time is *exponential* in input size
- Or worse…

Or in between???

EXPTIME

- Many problems appear *not* to have a polynomial time algorithm
- For a few, we can proof that each algorithm needs exponential time:
  – EXPTIME hardness, in particular *generalized games* (Generalized Go, Generalized Chess)
- For most, including many important and interesting problems, we cannot.

NP-completeness

- Theory shows relations and explains behavior of many combinatorial problems
- From many fields:
  - Logic
  - Graphs, networks, logistics, scheduling
  - Databases
  - Compiler optimization
  - Graphics
  - …

We need formalisation!

- Formal notion of
  - *problem instance*
  - *decision problem*
  - *computation*
  - *running time*

2

Abstract and concrete problems

Universiteit Utrecht

Different versions of problems

Focus on decision problems

- Decision problems
  - Answer is yes or no
- Optimization problems
  - Answer is a number
- Construction problems
  - Answer is some object (set of vertices, function, …)

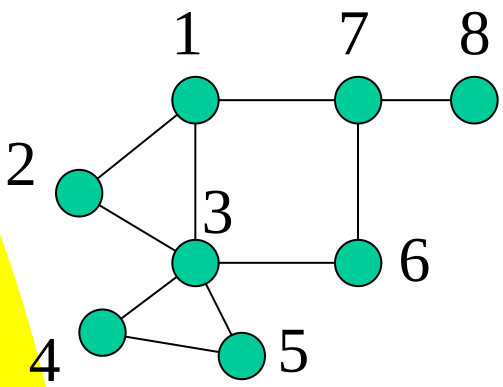
Abstract versus concrete problems

- Concrete:
  - Talk about graphs, logic formulas, applications, ...
- Abstract:
  - Sets of strings in finite alphabet

Abstract problem instances

- Computers work with bit strings
- Problems are described using objects:
  - *G* is a graph, …
  - Given a logic formula, …
  - Is there a clique of size at least *k*, ...
- We must map objects to *bit strings*
  (or another encoding like Gödel numbers...)

Formal problem instances



0	1	1	0	0	0	1	0
1	0	1	0	0	0	0	0
1	1	0	1	1	1	0	0
0	0	1	0	1	0	0	0
0	0	1	1	0	0	0	0
0	0	1	0	0	0	1	0
1	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0

Abstract decision problems

- Abstract decision problem:
  - Set of instances I
  - Subset of I: instances where the answer to the problem is YES.

Encoding / concrete problem

- Encoding: mapping of set of instances I to bitstrings in {0,1}*
- Concrete (decision) problem:
  - Subset of {0,1}*
- With encoding, an abstract decision problem maps to a concrete decision problem

3

P and NP

Complexity Class P

FORMAL

- Class of languages L, for which there exists a deterministic Turing Machine deciding whether $i \in L$, using running time $O(p(|i|))$ for some polynomial p

INFORMAL

- Class of decision problems that have polynomial time algorithms solving them

P

- An algorithm *solves* a problem
  - Decides if string in {0,1}* belongs to subset
- Time: deterministic, worst case
- Algorithm uses *polynomial time*, if there is a polynomial p such that on inputs of length *n* the algorithm uses at most p(*n*) time.
  - Size of input *x* is denoted |*x*|.
- P is the class of *concrete decision problems that have an algorithm that solves it, and that uses polynomial time*

Using P for abstract problems

- Abstract problem (with encoding) is in P, if the resulting problem is in P
- In practice: encoding and corresponding concrete problem is assumed *very implicitly*
- For polynomiality, encoding does not matter!
  - If we can transform encodings in polynomial time to each other
- Details: see e.g., chapter 34 of Introduction to Algorithms

Language of a problem

- Decision problem as a language:
  - Set of all yes-instances
- P is the set of all languages that have a polynomial time decision algorithm

Verification algorithm

- Verification algorithm has two arguments:
  - Problem input
  - Certificate ("solution")
- Answers "yes" or "no"
- *Checks* if 2nd argument is certificate for first argument for studied problem
- The language *verified by the verification algorithm* A is
  - $\{i \mid$ there is an c with $A(i,c) =$ true$\}$

Complexity Class NP

Two *equivalent* definitions of NP

- Class of languages L, for which there exists a Non-Deterministic Turing Machine *deciding* whether $i \in L_+$, using running time $O(p(|i|))$
- Class of languages L, for which there exists a Deterministic Turing Machine *verifying* whether $i \in L_+$, using a polynomial sized certificate *c*, and using running time $O(p(|i|))$

NP

- Problems with polynomial time verification algorithm and polynomial size certificates
- Problem L belongs to the class NP, if there exists a 2-argument algorithm A, with
  - A runs in polynomial time
  - There is a constant *d* such that for each *x*, there is a certificate *y* with
    - A(i,c) = true
    - $|c| = O(|i|^d)$

Many problems are in NP

- Examples: Hamiltonian Path, Maximum Independent Set, Satisfiability, Vertex Cover, …
- Al of these have trivial certificates (set of vertices, truth assignment, …)
- In NP (not trivial): Integer Linear Program

$P \subseteq NP$

- If A decides L in polynomial time, then as verification algorithm, compute
  - $B(i,c) = A(i)$
  - "We do not need a certificate".
- Famous open problem: P = NP ?? Or not??

3

Reducibility

Reducibility

- Language L_1 is *polynomial time reducible* to language L_2 (or: $\mathbf{L_1 \leq_P L_2}$), if there exists a polynomial time computable function f : $\{0,1\}^* \rightarrow \{0,1\}^*$ such that
  - For all $x \in \{0,1\}^*$:
    - $x \in L_1$ if and only if $f(x) \in L_2$

Lemma

Lemma: If $L_1 \leq_P L_2$ then if $L_2 \in P$, then $L_1 \in P$.

Proof-idea: run an algorithm for L_2 on $f(i)$ for input i to problem L_1.

Also: If $L_1 \leq_P L_2$ then if $L_2 \in NP$, then $L_1 \in NP$.

4

NP-completeness and the Cook-Levin theorem

NP-completeness

A language L is **NP-complete**, if

1. $L \in NP$
2. For every $L' \in NP$: $L' \leq_P L$

A language L is **NP-hard**, if

1. For every $L' \in NP$: $L' \leq_P L$
   - NP-hardness sometimes also used as term for problems that are not a decision problem, and for problems that are '*harder than NP*'

What does it mean to be NP-complete?

- Evidence that it is (very probably) hard to find an algorithm that solves the problem
  - Always
  - Exact
  - In polynomial time

CNF-Satisfiability

- Given: expression over Boolean variables in conjunctive normal form
- Question: Is the expression satisfiable? (Can we give each variable a value true or false such that the expression becomes true).
  - CNF: “and” of clauses; each clause “or” of variables or negations (x_i or not(x_j))

Cook-Levin theorem

- Satisfiability is NP-complete
  - Most well known is Cook's proof, using Turing machine characterization of NP.

5

Proving that problems are NP-complete

Proving problems NP-complete

Lemma

1. Let L' $\leq_P$ L and let L' be NP-complete. Then L is NP-hard.
2. Let L' $\leq_P$ L and let L' be NP-complete, and L $\in$ NP. Then L is NP-complete.

3-Sat

- 3-Sat is CNF-Satisfiability, but each clause has exactly three literals
- Lemma: CNF-Satisfiability $\leq_P$ 3-Sat
  - Clauses with one or two literals:
    - Use two extra variables p and q
    - Replace 2-literal clause (x or y) by (x or y or p) and (x or y or not(p))
    - Similarly, replace 1-literal clause by 4 clauses
  - Clauses with more than three literals:
    - Repeat until no such clauses
      - For (l_1 or l_2 or … l_r) add new variable t and take as replacement clauses (l_1 or l_2 or t) and (not(t) or l_3 or … or l_r)

3-Sat is NP-complete

- Membership in NP
- Reduction
  - 3-Sat is important starting problem for many NP-completeness proofs

Clique

- Given: graph G=(V,E), integer *k*
- Question: does G have a clique with at least *k* vertices?

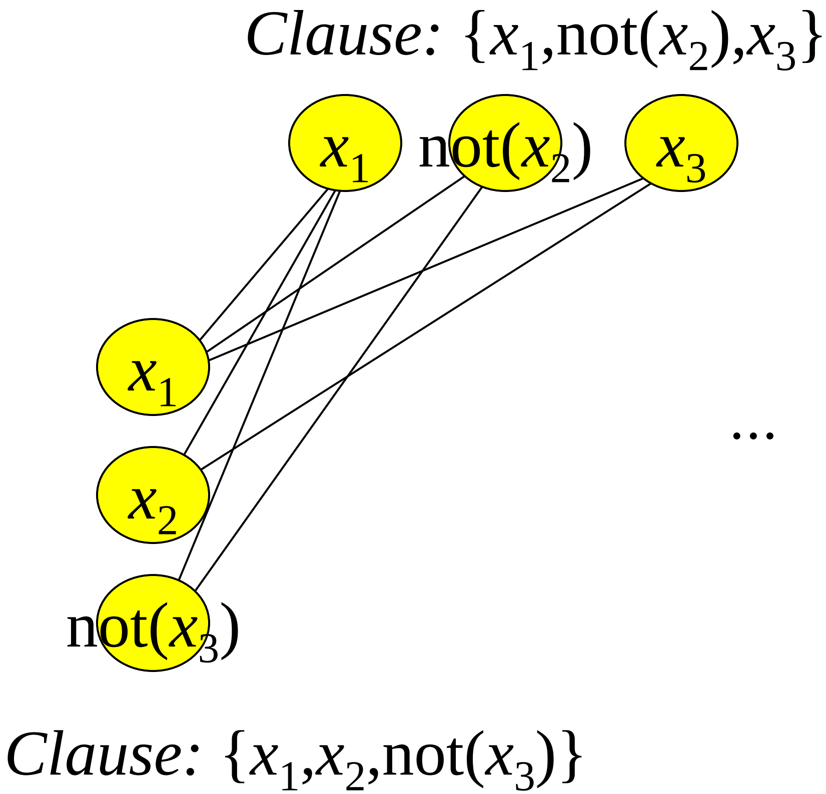
Clique is NP-complete.

In NP … easy!

NP-hardness: using 3-sat.

Reduction for Clique

- One vertex per literal per clause
- Edges between vertices in different clauses, except edges between x_i and not(x_i)
- If *m* clauses, look for clique of size *m*



Correctness

- There is a satisfying truth assignment, if and only if there is a clique with m vertices
- =>: Select from each clause the true literal. The corresponding vertices form a clique with m vertices.
- <=: Set variable x_i to true, if a vertex representing x_i is in the clique, otherwise set it to false. This is a satisfying truth assignment:
  - The clique must contain one vertex from each 3 vertices representing a clause.
  - It cannot contain a vertex representing x_i and a vertex representing not(x_i).

Independent set

- Independent set: set of vertices $W \subseteq V$, such that for all $v,w \in W$: $\{v,w\} \notin E$.
- Independent set problem:
  - Given: graph G, integer k
  - Question: Does G have an independent set of size at least k?
- Independent set is NP-complete

Independent set is NP-complete

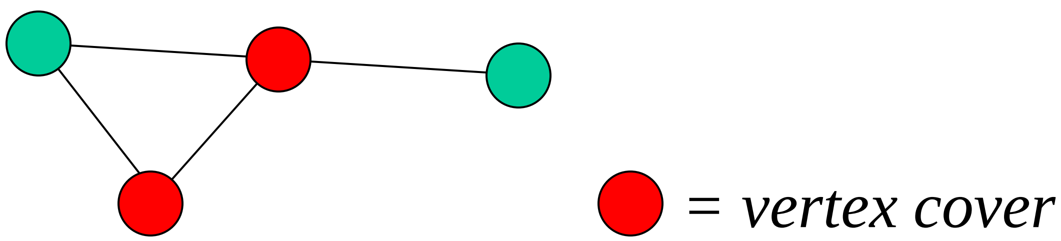
- In NP.
- NP-hard: transform from Clique.
- W is a clique in G, if and only if W is an independent set in the *complement* of G (there is an edge in G^c iff. there is no edge in G).

How do I write down this proof?

- Theorem. Independent Set is NP-complete.
- Proof: The problem belongs to NP: as certificates, we use sets of vertices; we can check in polynomial time for a set that it is a clique, and that its size is at least *k*.
  To show NP-hardness, we use a reduction from Clique. Let (G,*k*) be an input to the clique problem. Transform this to (G^c,*k*) with G^c the complement of G. As G has a clique with at least *k* vertices, if and only if G^c has an independent set with *k* vertices, this is a correct transformation. The transformation can clearly be carried out in polynomial time. QED

Vertex Cover

- Set of vertices $W \subseteq V$ with for all $\{x,y\} \in E$: $x \in W$ or $y \in W$.
- Vertex Cover problem:
  - Given G, find vertex cover of minimum size



Vertex cover is NP-complete

- In NP.
- NP-hard: transform from independent set.
- W is a vertex cover in G, if and only if V-W is an independent set in G.

Example of restriction

- Weighted vertex cover
  - Given: Graph G=(V,E), for each vertex $v \in$ V, a positive integer weight $w(v)$, integer k.
  - Question: Does G have a vertex cover of total weight at most k?
- NP-complete
  - In NP.
  - NP-hardness: set all weights to 1 (VC).

Techniques for proving NP-hardness

- Local replacement
- Restriction
- Component Design

6

Local replacement proofs

Technique 1: Local replacement

- Form an instance of our problem by
  - Taking an instance of a known NP-complete problem
  - Making some change "everywhere"
  - Such that we get an equivalent instance, but now of the problem we want to show NP-hard

Examples of Local Replacement

- We saw or will see:
  - 3-Satisfiability
  - Independent Set
  - TSP
  - Vertex Cover

7

Restriction proofs

Technique 2: Restriction

- Take the problem.
- Add a restriction to *the set of instances*. **NOT** to the *problem definition*!
- Show that this is a known NP-complete problem

Restriction: Weighted Vertex Cover

- Weighted vertex cover
  - Given: Graph G=(V,E), for each vertex $v \in$ V, a positive integer weight $w(v)$, integer k.
  - Question: Does G have a vertex cover of total weight at most k?
- NP-complete
  - In NP.
  - NP-hardness: set all weights to 1 (VC).

Restriction: Knapsack

- Knapsack
  - Given: Set *S* of items, each with integer value *v* and integer weight *w*, integers W and V.
  - Question: is there a subset of *S* of weight no more than *W*, with total value at least *V*?
- NP-complete
  - In NP
  - NP-hardness: set all weights equal to their values (Subset sum)

9

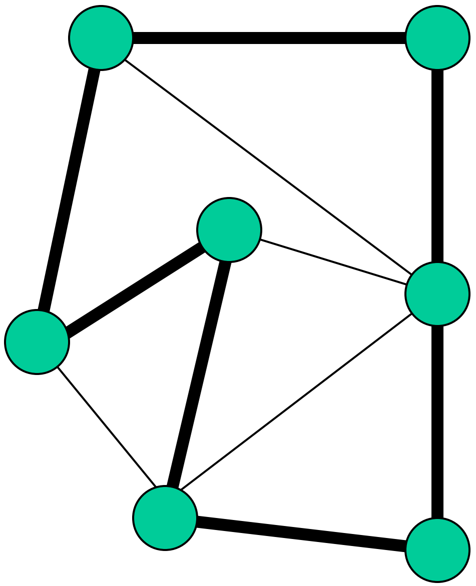
Component design proofs

Technique 3: Component design

- Build (often complicated) parts of an instance with certain properties
- Glue them together in such a way that the proof works
- Examples: Clique, Hamiltonian Circuit

Hamiltonian circuit

- Given: Graph G
- Question: does G have a simple cycle that contains all vertices?

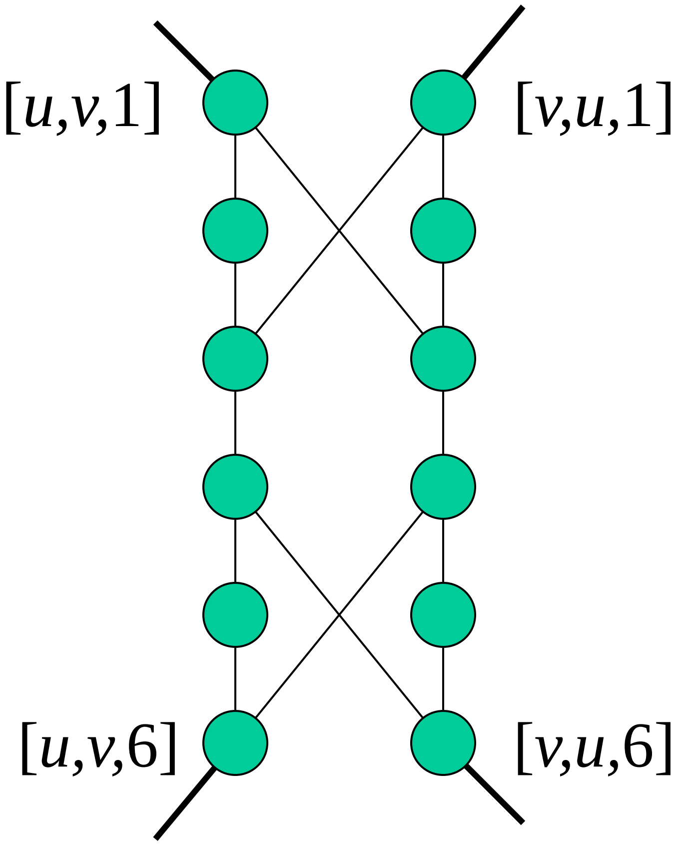


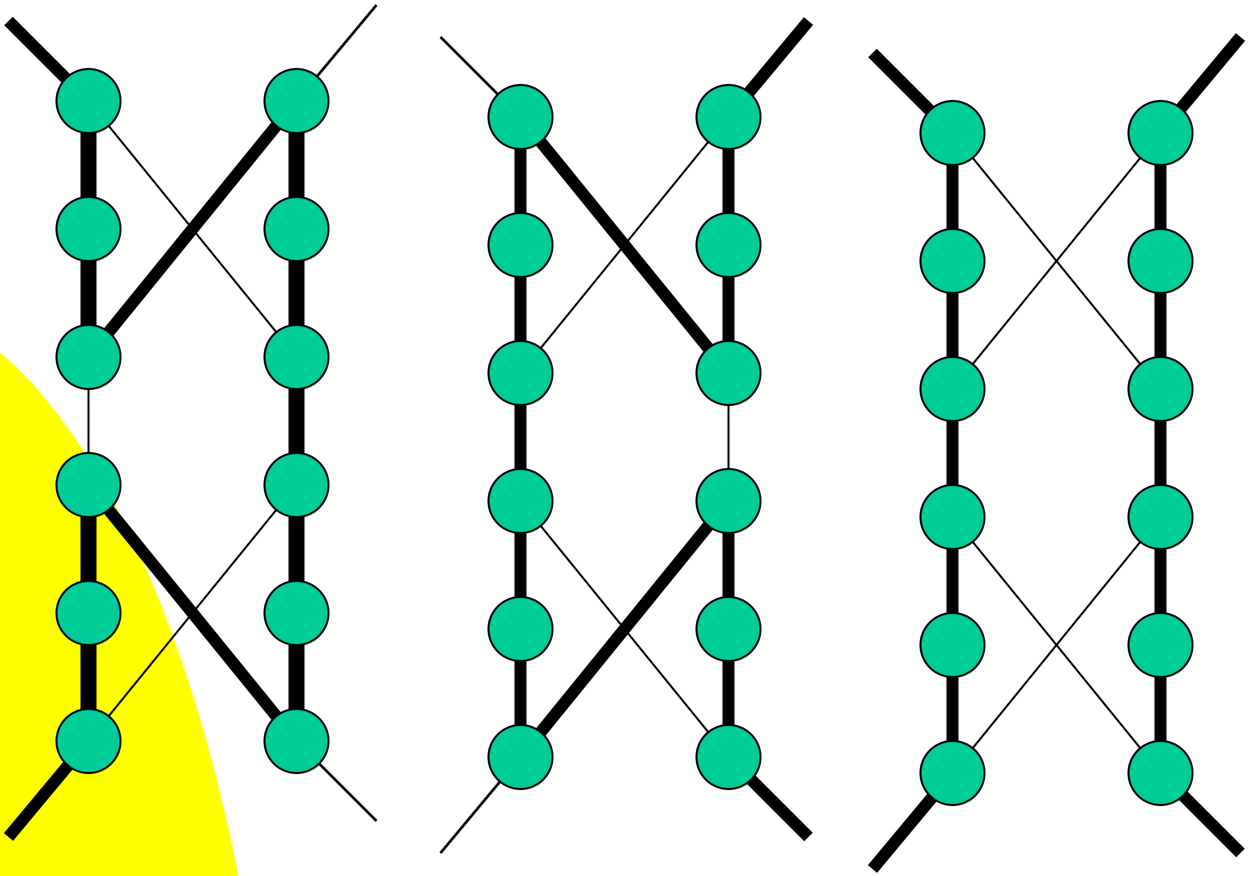
NP-completeness of Hamiltonian Circuit

- HC is in NP.
- Vertex Cover $\leq_P$ Hamiltonain Circuit: complicated proof (*component design*)
  - Widgets
  - Selector vertices
  - Given a graph *G* and an integer *k*, we construct a graph H, such that H has a HC, if and only if G has a VC of size *k*.

Widget

- For each edge $\{u,v\}$ we have a widget W_{uv}





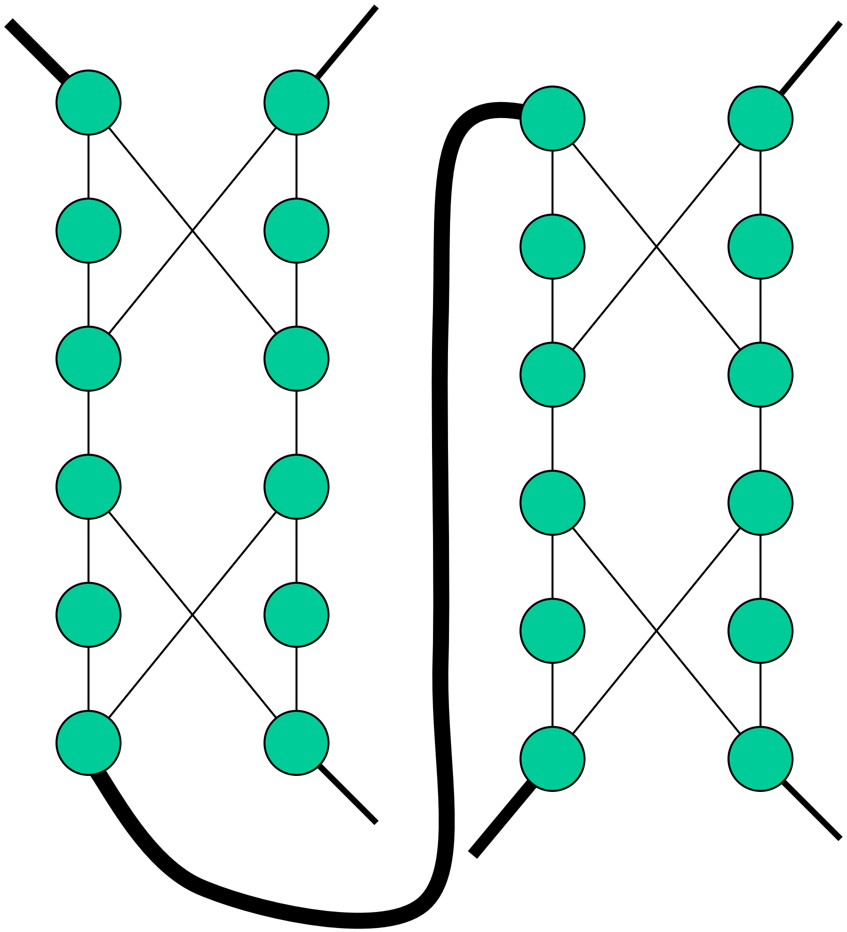
Only possible ways to visit all vertices in widget

Selector vertices

- We have k selector vertices $s_1, \ldots, s_k$
- These will represent the vertices selected for the vertex cover

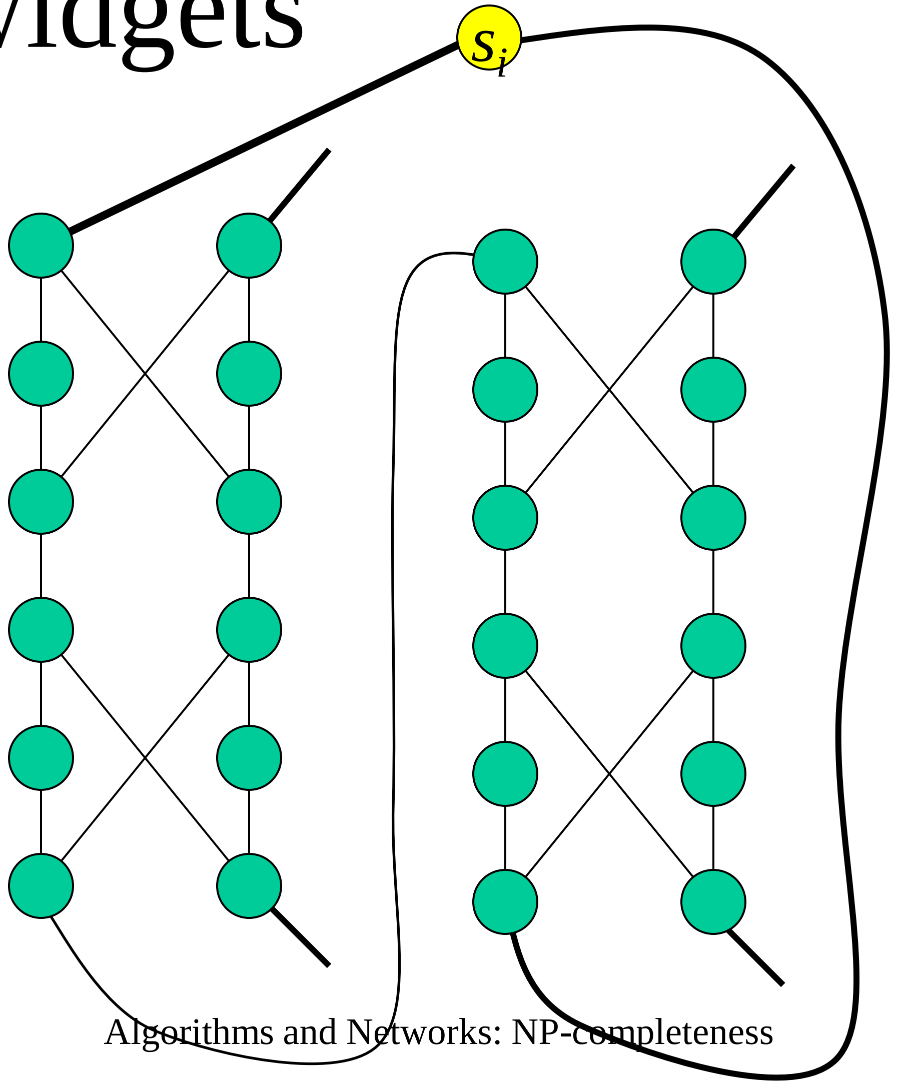
Connecting the widgets

- For each vertex v we connect the widgets of the edges $\{v,w\}$. Suppose v has neighbors $x_1, \ldots, x_r$: add edges $\{[v,x_1,6],[v,x_2,1]\}$, $\{[v,x_2,6],[v,x_3,1]\}$, …, $\{[v,x_{r-1},6],[v,x_r,1]\}$.



Connecting the selector vertices to the widgets

- Each selector vertex is attached to the first neighbor widget of each vertex, i.e. to vertex [v,x_1,1] and to the last neighbor widget [v,x_r,6]



Vertex in example has degree 2

Correctness of reduction

- Lemma: G has a vertex cover of size (at most) k, if and only if H has a Hamiltonian circuit.

Finally

- The reduction takes polynomial time.
- So, we can conclude that Hamiltonian Circuit is NP-complete.

TSP

- NP-completeness of TSP by *local replacement*:
  - In NP.
  - Reduction from Hamiltonian Circuit:
    - Take city for each vertex
    - Take cost(i,j) = 1 if $\{i,j\} \notin E$
    - Take cost(i,j) = 0, if $\{i,j\} \in E$
    - G has HC, if and only if there is a TSP-tour of length 0.
- Remark: variant with triangle inequality: use weights 2, 1 and n

10

Weak and strong NP-completeness

Problems with numbers

- Strong NP-complete:
  - Problem is NP-complete if numbers are given in unary
- Weak NP-complete:
  - Problem is NP-complete if numbers are given in binary, *but* polynomial time solvable when numbers are given in unary

Examples

- **Subset-sum**
  - Given: set of positive integers S, integer *t*.
  - Question: Is there a subset of S with total sum *t*?
    - Weak NP-complete. (Solvable in *pseudo-polynomial time* using dynamic programming: O(nt) time…)
- **3-Partition**
  - Given: set of positive integers S, (integer *t*).
  - Question: can we partition S into sets of exactly 3 elements each, such that each has the same sum (*t*)*?*
    - Strong NP-hard.
    - *t* must be the sum of S divided by |S|/3 = number of groups
    - Starting point for many reductions

Remark

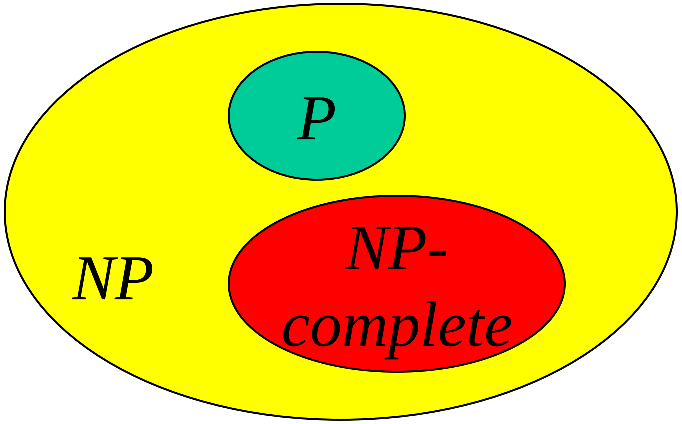
- Easily made mistake: reductions from subset sum that create exponentially large instances
- Subgraph Isomorphism for degree 2 graphs
  - Given: Graphs G and H, such that each vertex in G and H has degree at most 2
  - Question: Is G a subgraph of H?
    - NP-hardness proof can be done with 3-PARTITION

11

Some discussion

Discussion

- Is P ≠ NP? (who thinks so?)
- www.claymath.org/prizeproblems/pvsnp.htm : one of the millennium problems
- Why so hard to prove?
- What to do with problems that are NP-complete?
- Other complexity notions…



P vs NP is hard to prove

- P = NP? Hard to design poly algorithm...
- Current mathematical knowledge does not suffice to prove P != NP:
  - "Natural Proofs" can not separate P from NP (Razborov & Rudich, 1993)
  - $P^A = NP^A$, but $P^B \neq NP^B$ for some *oracles* A and B, so *diagonalisation* can not separate P from NP (Baker, Gill, & Solovay, 1975)

12

A Few Animals from
The Complexity Zoo

Much more classes

- In Theoretical Computer Science, a large number of other complexity classes have been defined
- Here, we give an informal introduction to a few of the more important ones
- There is much, much, much more…

coNP

- Complement of a class: switch “yes” and “no”
- coNP: complement of problems in NP, e.g.:

NOT-HAMILTONIAN

- Given: Graph G
- Question: Does G NOT have a Hamiltonian circuit

UNSATISFIABLE

- Given: Boolean formula in CNF
- Question: Do all truth assignments to the variable make the formula false?

PSPACE

- All decision problems solvable in polynomial space
- Unknown: is P=PSPACE?
- Savitch, 1970: PSPACE = NPSPACE
  - NPSPACE: solvable with non-deterministic program in polynomial time
- PSPACE-complete, e.g.,
  - generalized Tic-Tac-Toe, generalized Reversi,
  - Quantified Boolean formula's (QBF):

$$\forall x_1 \exists x_2 \forall x_3 \exists x_4 \left(x_1 \vee \neg x_2 \vee x_3 \right) \wedge \left(x_2 \vee \neg x_3 \right)$$

EXPTIME

- Decision problems that can be solved in exponential time
- P is unequal EXPTIME (Stearns, Hartmanis, 1965)
- EXPTIME complete problems:
  - Generalized chess, generalized checkers, generalized go (Japanese drawing rule)
  - Given a Turing Machine M and integer *k*, does M halt after *k* steps?

And a few more

- NEXPTIME: non-deterministic exponential time
- EXPSPACE = NEXPSPACE: exponential space

Graph Isomorphism

- Discussed in another lecture
- Given two graphs, are they *isomorphic*?
- In NP, not known to be NP-complete; not known to be in P
- Several problems are *equaly hard*: Isomorphism-complete

NC

- NC: “Nicks class”, after Nick Pippinger
- Talks about the time to solve a problem with a **parallel** machine
- Model: we have a polynomial number of processors, that use the same memory
  - Variants depending on what happens when processors try to read or write the same memory location simultaneously

NC – the definition

- NC: decision problems that can be solved with a PRAM (Parallel Random Access Machine) with polynomial number of processors in *polylogarithmic* time
  - O((log n)^d) for some constant d
- Unknown: P=NC?
- P-complete problems are expected not to be in NC. An example is
  - Linear Programming (formulated as decision problem)

Counting

- #P: ("Sharp-P")
- Problems that outputs a number
- The precise definition will not be given here. Think as: "what is the number of certificates for this instance", with polynomial checking of certificates
- #P-complete e.g.:
  - Number of satisfying truth assignments of 3SAT-formula
  - Number of Hamiltonian circuits in a graph
  - Number of perfect matchings in a given graph
- PP is a related class (vaguely: "decide if the number of solutions is at most given number *k*")

On PP and #P

- Inference:
  - Given: probabilistic network, observations O, variable X, value x, value p in [0,1]
  - Question: $\Pr(X = x \mid O) <= p$?
- Decision variant of problem from course Probabilistic Reasoning
- Is PP-complete; variants are #P-complete
- PP-hard and #P-hard problems are probably not polynomial...

LSPACE or L

- Problems can be solved with only logarithmic extra space:
  - You can read the input as often as you want
  - You may use only O(log n) extra memory
    - E.g.: $\Theta(1)$ pointers to your input
- NL: non-deterministic logspace...

More

- The Polynomial Time Hierarchy
- Oracles
- UP: unique solutions