Griffith University

3515ICT Theory of Computation
Computational Complexity

(Based loosely on slides by Harald Søndergaard of The University of Melbourne)

Big-Oh Notation

Recall that

$$g = O(f) \text{ iff } n > n_0 \Rightarrow g(n) < c \cdot f(n)$$

for some integer constants n_0 and c. *E.g.*,

$$\begin{aligned}
5n^3 + 2n^2 - 22n + 6 &= O(n^3) \\
5n^3 + 2n^2 - 22n + 6 &= O(n^4) \\
1 + 2 + \ldots + n &= O(n^2) \\
3n \log(n) &= O(n \log(n)) \\
3n \log(n) &= O(n^2)
\end{aligned}$$

In (time) complexity analysis we assess time consumption (for example in the worst case).

The idea is to give a bound for the asymptotic behaviour of running time, as a function of input size.

Function t is *polynomially bounded* if $t(n) = O(n^r)$ for some constant r.

Complexity Classes

Let M be a deterministic Turing machine. The *time complexity* of M is the function $t_M : \mathbb{N} \to \mathbb{N}$ defined by

$$t_M(n) = max \left\{ m \;\middle|\; \begin{array}{l} \exists w \in \Sigma^* \;.\; |w| = n \text{ and } M \\ \text{takes time } m \text{ to run on } w \end{array} \right\}$$

Let $t : \mathbb{N} \to \mathbb{N}$ be a function.

$$TIME(t(n)) = \left\{ L \;\middle|\; \begin{array}{l} L \text{ is decided by some} \\ \text{deterministic Turing} \\ \text{machine in } O(t(n)) \text{ time} \end{array} \right\}$$

Machine Model Affects Complexity

A one-tape Turing machine can decide the language

$$A = \{0^k 1^k \mid k \geq 0\}$$

in $O(n \log(n))$ time. (A naive machine takes $O(n^2)$ time.)

It works by repeatedly scanning across its input, crossing off every second 0 and every second 1, checking after each scan that the number of non-crossed symbols remains even.

Since it halves the number of 0s and 1s (integer division) in each scan, the time taken is $O(n \log(n))$ in the worst case (accept).

The machine cannot decide A in linear time.

Model Affects Complexity (cont.)

A 2-tape deterministic machine can do better.

It can copy all the 1s to its second tape and then match them against the 0s in linear time.

The two kinds of Turing machine have the same computational power, but they have different complexity properties.

Thus we lose the *robustness* of the Turing machine model when we move to studying complexity.

The Class P

Fortunately it is possible to recover much of that robustness.

We need to be less discerning, considering all the polynomial time deterministic deciders as one class.

We define: P is the class of languages decidable by a *deterministic* Turing machine in polynomial time:

$$P = \bigcup_k TIME(n^k)$$

Robustness of P

Recall that polynomial functions are closed under addition, multiplication, and composition.

P is robust in the sense that all the *deterministic* computational models are *polynomially invariant*: Changing to some other deterministic machine model will not take us outside P.

P roughly corresponds to the class of problems that are *tractable*.

Polynomial vs Exponential

Also note that it is possible for a function to be super-polynomial without being exponential.

For $t(n)$ to be polynomial means to be $O(n^r)$ for some integer r.

Here we take "exponential" to mean $\Omega(2^{cn})$ for some constant $c > 0$.

Then there are functions, such as $2^{\sqrt{n}}$ and $n^{\log n}$ which grow faster than any polynomial function, but more slowly than any exponential function.

Hence it is possible that $P \subset NP$, and at the same time, that NP-complete problems can be solved in less than exponential time.

Some Problems in P

$RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ relatively prime}\}$

is in P. A deterministic Turing machine can use Euclid's algorithm to find the greatest common divisor of x and y, and accept iff that is 1.

$$PATH = \left\{ \langle G, s, t\rangle \;\middle|\; \begin{array}{l} G \text{ is a directed graph} \\ \text{with a path from } s \text{ to } t \end{array} \right\}$$

is in P. We have various deterministic algorithms for this problem, all running in polynomial time.

Theorem: Every context-free language is in P.

We give an $O(n^3)$ dynamic programming algorithm to decide membership of a CFL L.

Some Problems in P (cont.)

For an input string $w = w_1 w_2 \cdots w_n$ we build an $n \times n$ table. Entry (i, j) will contain variable V iff V can generate $w_i \cdots w_j$.

For $i = 1$ to n

  For each rule $A \rightarrow w_i$

    Place A in $table(i, i)$

For $len = 2$ to n

  For $i = 1$ to $n - len + 1$

    Let $j = i + len - 1$

    For $k = i$ to $j - 1$

      For each rule $A \rightarrow BC$

        If $B \in table(i, k)$ and $C \in table(k + 1, j)$

          Place A in $table(i, j)$

Accept iff $S \in table(1, n)$

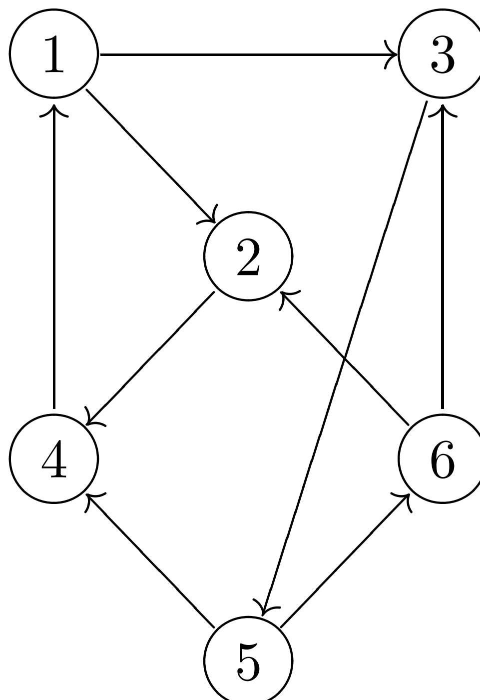
Nondeterminism Makes a Difference

The Hamiltonian path problem is $HAMPATH =$

$$\left\{ \langle G, s, t \rangle \;\middle|\; \begin{array}{l} G \text{ is a directed graph with a} \\ \text{Hamiltonian path from } s \text{ to } t \end{array} \right\}$$

where a Hamiltonian path visits each node in G exactly once.

Two instances: Is there a Hamiltonian path from 1 to 4? From 1 to 6?



HAMPATH with Nondeterminism

Let G have m nodes. Represent G as, say,

$$x_1 \# y_1 \#\# \cdots \#\# x_m \# y_m \#\#\# m$$

We can build a multitape nondeterministic Turing machine N to solve the problem.

One tape holds the graph (does not change).

N guesses a sequence of m nodes on a different tape, each node chosen between 1 and m:

$$\sqcup i_1 \sqcup i_2 \sqcup i_3 \cdots \sqcup i_m$$

(The following steps may require more tapes.)
If any node is repeated on the tape with the guess, N rejects. If $i_1 \neq s$ or $i_m \neq t$, N rejects. For each (i_k, i_{k+1}), if it is not an edge of G, N rejects. Otherwise N accepts.

. . . with Nondeterminism (cont.)

The guess is done in polynomial (linear) time.

Checking for repetition and verifying the start and end nodes is done in polynomial time.

Verifying an edge (by looking up the representation) is also done in polynomial time.

. . . and without Nondeterminism

"Guessing" in linear time is clearly very powerful.

To solve the same problem with a deterministic machine, it would seem that the best we can do is exhaustive search through all possible orderings of the n nodes.

A deterministic machine can do this (on tape 2) in exponential time (there are $(n-1)!$ sequences).

Nobody has come up with a polynomial-time solution.

Polynomial Verifiability

HAMPATH has a feature which it shares with a large number of important problems: It is polynomially verifiable.

On the one hand, *discovering* a Hamiltonian path seems difficult, and certainly we do not have a fast algorithm for it.

On the other hand, if somebody claims to have a path, *checking* their claim is easy: we can do that in polynomial time.

It is possible that *verifying* a Hamiltonian path is much easier than *determining its existence.*

Polynomial Verifiability (cont.)

A *verifier* for language A is an algorithm V with

$$A = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some } \underbrace{\text{string } c}_{\text{certificate}} \}$$

A polynomial time verifier runs in time that is polynomial in the length of w.

A is *polynomially verifiable* if it has a polynomial time verifier.

Polynomial Verifiability (cont.)

Notice that polynomial verifiability need not be closed under complement.

For example, consider $\overline{HAMPATH}$.

There is no obvious certificate of the *non-existence* of a path.

We don't know how to verify this non-existence without using the same exponential time method that was needed for *determining* non-existence in the first place.

The Class NP

NP is the class of languages that have polynomial time verifiers.

Theorem: A is in NP iff A is decided by some nondeterministic polynomial time Turing machine.

We can also phrase this as

$$NP = \bigcup_k NTIME(n^k)$$

$$NTIME(t(n)) = \left\{ L \;\middle|\; \begin{array}{l} L \text{ is decided by a non-} \\ \text{deterministic Turing} \\ \text{machine in } O(t(n)) \text{ time} \end{array} \right\}$$

This definition is robust, in the sense that NP doesn't change when we change the machine to any other nondeterministic model.

P vs NP

In summary:

For P, membership can be *decided* quickly.

For NP, membership can be *verified* quickly.

Clearly $P \subseteq NP$. Is

$$P = NP?$$

Many computer scientists consider this the

most important unanswered question

in computer science.

Some Problems in NP

Consider again the Hamiltonian path problem *HAMPATH*:

> Given a directed graph G, and nodes s and t, is there a path, starting in s, ending in t, and visiting each node exactly once?

This problem has the property of being *polynomially verifiable.*

Discovering a Hamiltonian path seems difficult, indeed we do not have a fast algorithm for it.

But if somebody claims to have a valid path, *checking* that claim is easy (in polynomial time): The path is a *certificate.*

It is possible that *verifying* a Hamiltonian path is much easier than *determining it existence.*

Some Problems in NP (cont.)

An *NP* problem of interest for cryptography is

$$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid \Sigma A = t \text{ for some } A \subseteq S\}$$

(Here A and S are considered multisets.)

We don't know of a polynomial-time decider, but the subset A provides an easy certificate.

A nondeterministic TM decides *SUBSET-SUM* in polynomial time:

1. Select a subset $A \subseteq S$, non-deterministically.
2. Accept if $\Sigma A = t$, otherwise reject.

No Closure under Complement?

Notice that polynomial verifiability need not be closed under complement.

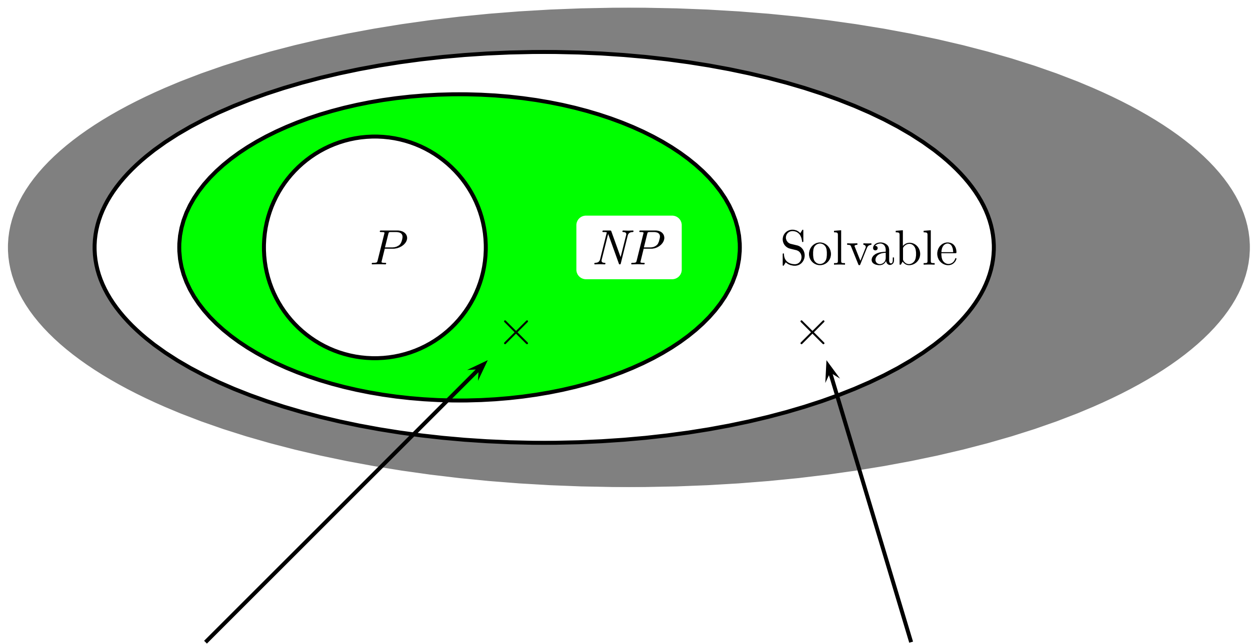
For example, consider $\overline{HAMPATH}$.

There is no obvious certificate of the *non-existence* of a path.

We don't know how to verify this non-existence without using the same exponential time method that was needed for *determining* non-existence in the first place.

Similarly $\overline{SUBSET\text{-}SUM}$ may not be in NP.

Namely, how could we easily verify that *no* subset adds to t?



Problem which *provably* can be solved by NDTM in polynomial time, though not by deterministic TM. **Exists?**

Solvable problem that *provably* cannot be solved in polynomial time by NDTM ("nondeterministically intractable"). **Exists!**

Polynomial-Time Reducibility

We can classify "hardness" of problems by refining the concept of reducibility.

The crucial point now is that when we talk about reducing problem A to problem B, the (mechanistic) reduction must take place in polynomial time.

A is *polynomial-time reducible* to B,

$$A \leq_P B$$

iff there is some polynomial-time computable function $f : \Sigma^* \to \Sigma^*$, such that for all w,

$$w \in A \text{ iff } f(w) \in B.$$

f is the *polynomial-time reduction* of A to B.

Theorem: If $A \leq_P B$ and $B \in P$ then $A \in P$.

NP-Completeness

B is *NP-hard* iff every $A \in NP$ is reducible to B in polynomial time.

B is *NP-complete* iff

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

We denote the class of NP-complete languages by NPC.

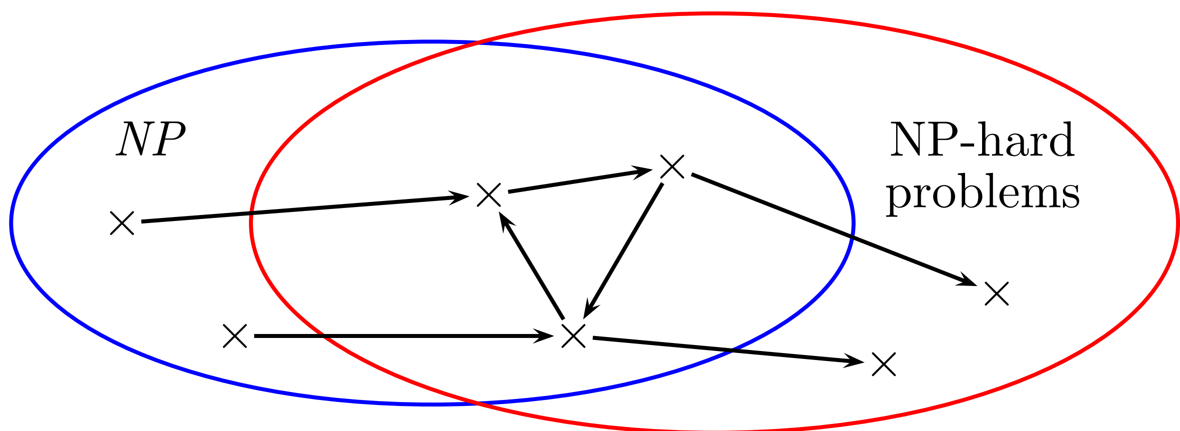
Our interest in this class is due to this result:

Theorem: If $B \in NPC$ and $B \in P$ then $P = NP$.

Theorem: If $B \in NPC$ and $B \leq_P C$ for $C \in NP$ then $C \in NPC$.

NPC Diagrammatically

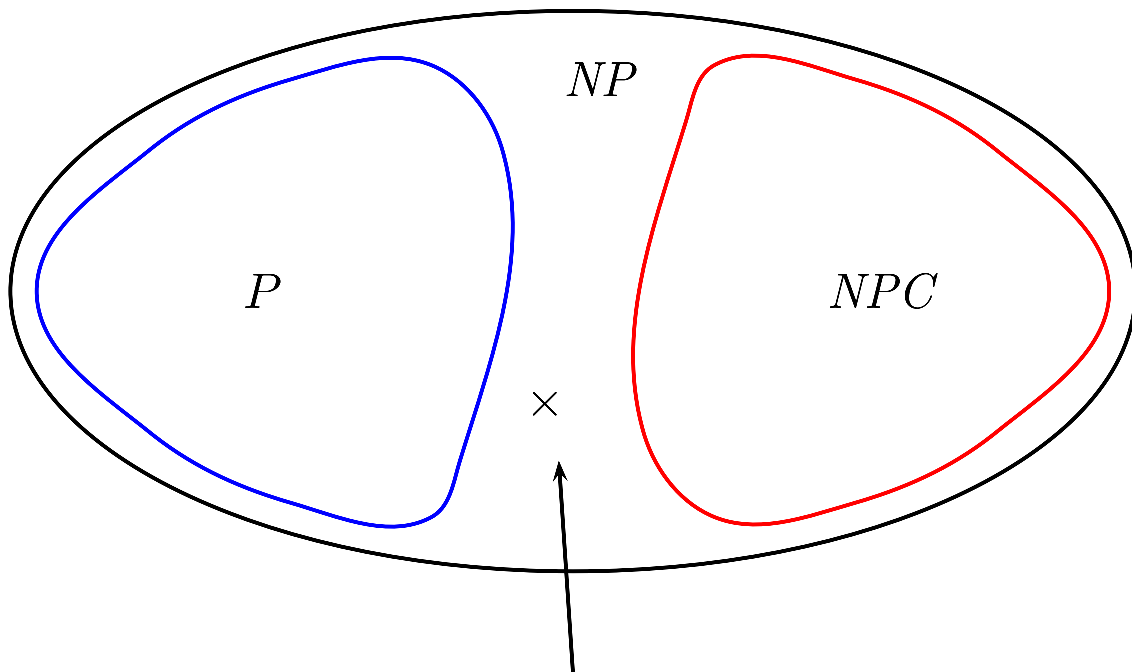
NP-complete problems are the "hardest" problems in NP.



Here arrows indicate polynomial time reduction (a transitive relation).

NPI

If $P \neq NP$ then we have this picture:



Moreover, if $P \neq NP$, this gap is known to be inhabited (Ladner's theorem).

Let $NPI = NP \setminus (P \cup NPC)$.

No specific problem is known to be in NPI. Difficult problems that have not (yet) been proven NP-complete, such as *graph isomorphism*, have been suggested as candidate members of NPI.

Proving NP-Completeness

The plan for demonstrating NP-completeness:

1. The mother of all NPC problems: SAT.
   (a) Show $SAT \in NP$.
   (b) Show that SAT is NP-hard.
2. For subsequent problems B:
   (a) Show $B \in NP$.
   (b) Show B NP-hard by polynomially reducing some known NP-complete problem (*e.g.*, SAT) to B.

Propositional Logic

Let the Boolean values be 0 and 1 and let V be a set of Boolean variables.

A *truth assignment* is a total function from V to $\{0, 1\}$.

Well-formed propositional formulas include 0, 1, and x for all $x \in V$.

Moreover, if ϕ and ψ are well-formed formulas, so are $\neg\phi$, $\phi \wedge \psi$ (conjunction), and $\phi \vee \psi$ (disjunction).

Propositional Logic (cont.)

The truth value of a formula ϕ is determined by the truth tables for $\neg$, $\wedge$, and $\vee$.

For $x \in V$, x and $\neg x$ are *literals.*

A *clause* is a disjunction of literals.

A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses.

Every formula has an equivalent formula in CNF.

ϕ is *satisfiable* if there is a truth assignment such that the truth value of ϕ is 1.

SAT is the problem: Given ϕ, is ϕ satisfiable?

Note that if ϕ has n variables then there are 2^n truth assignments.

SAT is in NP

To show that *SAT* is in *NP*, we build a 2-tape NTM over alphabet $\{0, 1, \wedge, \vee, \#\}$. Encode

$$\begin{array}{rcl} x_i & \text{by} & \bar{i}\#1 \\ \neg x_i & \text{by} & \bar{i}\#0 \end{array}$$

Here $\bar{i}$ is a string of i 1s.

For example, $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3)$ yields the input string

$$\underbrace{1\#11\#111}_{\text{variables}}\#\#\underbrace{1\#1\vee 11\#0\wedge 1\#0\vee 111\#1}_{\text{formula}}$$

The NTM will check that this is a valid *SAT* instance.

Then it will guess-and-check as usual, using tape 2 to store the guessed truth assignment.

SAT is in NP (cont.)

Tape 2: $1\#0\#\#11\#0\#\#111\#1\sqcup$

Tape 1: $1\#11\#111\#\#1\#1\vee 11\#0\wedge 1\#0\vee 111\#1\sqcup$

The latter checking happens in $O(n^2)$ time:

1. Scan to the formula on tape 1.
2. Look up variable's truth value on tape 2.
3. If it *differs* from the tape 1 value, look for next $\vee$, scan past it, and go to (2).
4. If $\wedge$ (or $\sqcup$) is found before $\vee$, reject.
5. If the variable's value *matches*, scan to after next $\wedge$ and go to (2).
6. If no $\wedge$ is found, accept.

Modelling NTMs

We still need to show that SAT is NP-hard.

Let M be an NTM, bounded by polynomial p (that is, given input of size n, M has halted by time $p(n)$.)

$Q = \{q_0, q_1, \ldots, q_{m-2}, q_r, q_a\}$

$\Gamma = \{\sqcup, a_1, \ldots, a_s, a_{s+1}, \ldots, a_t\}$

$\Sigma = \{a_{s+1}, \ldots, a_t\}$

Let $u \in \Sigma^*$ have length n.

We construct a formula $F(u)$ which mimics M's operation on u.

F and its construction depends on $p(n)$.

Modelling NTMs (cont.)

We use the following variables with the suggested meaning:

Q_{ik}	$0 \leq i \leq m$ $0 \leq k \leq p(n)$	M is in state q_i at time k
P_{jk}	$0 \leq j \leq p(n)$ $0 \leq k \leq p(n)$	M is scanning position j at time k
S_{jrk}	$0 \leq j \leq p(n)$ $0 \leq r \leq t$ $0 \leq k \leq p(n)$	Tape position j contains symbol a_r at time k

Modelling NTMs (cont.)

Capturing initial configuration ($u = a_{r_1} \cdots a_{r_n}$):

$$Q_{00}$$

$$P_{00}$$

$$S_{000}$$

$$S_{1r_10}$$

$$S_{2r_20}$$

$$\vdots$$

$$S_{nr_n0}$$

$$S_{(n+1)00}$$

$$\vdots$$

$$S_{p(n)00}$$

Capturing acceptance:

$$Q_{mp(n)}$$

Modelling NTMs (cont.)

Capturing state invariants (here $0 \leq i < i' \leq m$):

$\bigvee_{i=0}^{m} Q_{ik}$	M is in some state
$\neg Q_{ik} \vee \neg Q_{i'k}$	but no more than one

And tape head invariants ($0 \leq j < j' \leq p(n)$):

$\bigvee_{j=0}^{p(n)} P_{jk}$	Tape head is in some position
$\neg P_{jk} \vee \neg P_{j'k}$	but no more than one

Capturing symbol invariants (only one symbol in any position) is handled the same way.

Modelling NTMs (cont.)

Capturing tape consistency: Symbols not under the tape head are not changed ($0 \leq r \leq t$):

$$\neg S_{jrk} \vee P_{jk} \vee S_{jr(k+1)}$$

Capturing transitions:

$$\neg Q_{ik} \vee \neg P_{jk} \vee \neg S_{jrk} \vee Q_{i'(k+1)}$$
$$\neg Q_{ik} \vee \neg P_{jk} \vee \neg S_{jrk} \vee S_{jr'(k+1)}$$
$$\neg Q_{ik} \vee \neg P_{jk} \vee \neg S_{jrk} \vee P_{j'(k+1)}$$

Capturing halting:

$$\neg Q_{ik} \vee \neg P_{jk} \vee \neg S_{jrk} \vee Q_{i(k+1)}$$

for $i \in \{q_a, q_r\}$.

Similarly for the tape and the tape head.

Modelling NTMs (cont.)

The formula we have constructed is in CNF.

You should convince yourself that the construction ensures that

- if $u \in L(M)$ then $F(u)$ has a satisfying assignment, and
- if $u \notin L(M)$ then $F(u)$ has no satisfying assignment.

NP-Hard vs Exponential

Many interesting problems have been shown to be NP-hard, without anybody being able to show that they are in NP.

Such problems could well turn out to require exponential time, even if $P = NP$.

It is possible that some problem may require exponential time, and yet it is not NP-hard. Nothing in our definitions rules this out.

It is also possible that some NP-hard problem does not require exponential time (we could have $P = NP$ and the problem could require super-polynomial time).

SAT is NP-Complete

We established that every $A \in NP$ can be reduced to SAT in polynomial time.

Actually the construction we used reduced every NP problem to SAT_{CNF}, the satisfiability problem for formulas in Conjunctive Normal Form.

SAT is NP-Complete (cont.)

But then SAT (for arbitrary propositional formulas) is also NP-complete:

1. It is in NP (as for the CNF case), and
2. There is a trivial poly-time reduction of SAT_{CNF} to SAT.

$$SAT_{CNF} \rightleftarrows SAT$$

The reduction from right to left is covered by the general reduction we did in the last lecture.

(A direct reduction is not trivial, by the way: We know how to turn an arbitrary formula into CNF, but that technique can cause an exponential blow-up! However, in the translation we don't have to preserve equivalence, only satisfiability!)

3-SAT

A propositional formula is in *3-CNF* if it is in CNF and every clause has exactly three literals, as in

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee \neg x_4)$$

3-SAT is the problem whether a 3-CNF formula is satisfiable.

Let us show that the problem is NP-complete.

First, *3-SAT* is in *NP*, as the machine which solves SAT in polynomial time also works for *3-SAT*.

So it suffices to show $SAT_{CNF} \leq_P$ *3-SAT*.

3-SAT (cont.)

The reduction is as follows. Given an instance of SAT_{CNF}:

$$C_1 \wedge C_2 \wedge \ldots C_k$$

we turn every clause C into a bunch of 3-literal clauses as follows:

If C is a single literal ℓ, produce $(\ell \vee \ell \vee \ell)$.

If C is $(\ell_1 \vee \ell_2)$, produce $(\ell_1 \vee \ell_2 \vee \ell_2)$.

If C has three literals, leave it as it is.

Clearly satisfiability is preserved for these clauses.

3-SAT (cont.)

If C has $n > 3$ literals:

$$(\ell_1 \vee \ell_2 \vee \ell_3 \vee \cdots \vee \ell_n)$$

produce $n-2$ clauses using $n-3$ fresh variables $x_3 \ldots x_{n-1}$:

$$\begin{array}{ll} & (\ell_1 \vee \ell_2 \vee x_3) \\ \wedge & (\neg x_3 \vee \ell_3 \vee x_4) \\ & \vdots \\ \wedge & (\neg x_{n-2} \vee \ell_{n-2} \vee x_{n-1}) \\ \wedge & (\neg x_{n-1} \vee \ell_{n-1} \vee \ell_n) \end{array}$$

Why does this preserve satisfiability?

3-SAT (cont.)

$\boxed{\Rightarrow}$ Let t be a truth assignment that satisfies

$$(\ell_1 \vee \ell_2 \vee \ell_3 \vee \cdots \vee \ell_n)$$

and let ℓ_j be the first literal that comes out true.

We satisfy

$$\begin{array}{ll} & (\ell_1 \vee \ell_2 \vee x_3) \\ \wedge & (\neg x_3 \vee \ell_3 \vee x_4) \\ & \vdots \\ \wedge & (\neg x_{n-2} \vee \ell_{n-2} \vee x_{n-1}) \\ \wedge & (\neg x_{n-1} \vee \ell_{n-1} \vee \ell_n) \end{array}$$

by extending t so that $x_3 \ldots x_j$ are made true, and the remaining xs are false.

3-SAT (cont.)

$\boxed{\Leftarrow}$ Let t be a truth assignment that satisfies

$$\begin{array}{ll} & (\ell_1 \vee \ell_2 \vee x_3) \\ \wedge & (\neg x_3 \vee \ell_3 \vee x_4) \\ & \vdots \\ \wedge & (\neg x_{n-2} \vee \ell_{n-2} \vee x_{n-1}) \\ \wedge & (\neg x_{n-1} \vee \ell_{n-1} \vee \ell_n) \end{array}$$

Then t also satisfies $C =$

$$(\ell_1 \vee \ell_2 \vee \ell_3 \vee \cdots \vee \ell_n).$$

Namely, assume that it doesn't.

Then every ℓ_j is false.

So x_3 is true, hence x_4 is, and so on, up to x_{n-1}.

But then $(\neg x_{n-1} \vee \ell_{n-1} \vee \ell_n)$ is false.

We have a contradiction, so C is satisfiable.

NP-complete Graph Problems

Clique Given graph G and integer k, does G have a complete subgraph of size k?

Vertex Cover Given graph G and integer k, is there a set C of k nodes such that every edge has (at least) one end in C?

Edge Cover Given graph G and integer k, is there a set C of k edges such that every node is the end of (at least) one edge in C?

Dominating Set Given graph G and integer k, is there a set C of k nodes such that every node is either in C or adjacent to a node in C?

Independent Set Given graph G and integer k, is there a set C of k nodes such that no two nodes in C are adjacent?

Subgraph Isomorphism Given graphs G and H, is G isomorphic to some subgraph of H?

NP-complete Graph Problems (cont.)

Colouring Given graph G and integer k, can we colour the nodes of G with k colours so that no adjacent nodes have the same colour?

Hamiltonian Path Given graph G and nodes s and t, is there a Hamiltonian path in G from s to t? (A Hamiltonian path is a path that visits every node exactly once.)

Hamiltonian Cycle Given graph G, is there a Hamiltonian cycle in G?

Travelling Salesman Problem Given a weighted graph G and integer C, is there a Hamiltonian cycle whose cost (sum of edge weights) is less than C?

The Clique Problem

Clique Given graph G and integer k, does G have a complete subgraph of size k?

First, Clique is in NP, because given a set of k nodes of G, we can check whether each pair of nodes in C are adjacent in G in polynomial time.

Second, we show Clique is NP-hard by proving that 3SAT $\leq_P$ Clique.

Let $E = C_1 \wedge \cdots \wedge C_n$ be a boolean expression in 3CNF.

Let each literal occurrence in C be a node of G. The edges of G connect all pairs of nodes from different clauses *except* if the two nodes are complementary literals. Let k be n. Clearly (G, k) can be constructed in time polynomial in $|E|$.

It is straightforward to see that E is sastisfiable if and only if G has a clique of size k.

The Vertex Cover Problem

Let $G = (N, E)$ be an undirected graph.

$V \subseteq N$ is a *vertex cover* of G iff $n_1 \in V$ or $n_2 \in V$ for all edges $(n_1, n_2) \in E$.

We show that $\mathit{VERTEX}\text{-}\mathit{COVER} =$

$$\left\{ \langle G, k \rangle \;\middle|\; \begin{array}{l} G \text{ is a graph which has} \\ \text{a vertex cover of size } k \end{array} \right\}$$

is NP-complete.

The problem is in NP because a nondeterministic TM can guess a set of k nodes and check in polynomial time whether they cover G.

All it needs is a linear scan through all of G's edges, and for each edge, a linear scan of the guess.

Reducing 3-SAT to Vertex Cover

For a given 3-CNF formula ϕ we construct $\langle G, k\rangle$ such that G has a vertex cover of size k iff ϕ is satisfiable.

Let $\phi = (\ell_{11} \vee \ell_{12} \vee \ell_{13}) \wedge \cdots \wedge (\ell_{m1} \vee \ell_{m2} \vee \ell_{m3})$ have n variables $x_1 \ldots x_n$.

We produce $3m + 2n$ nodes, one per literal, and two per variable:

$$\{\ell_{kj} \mid 1 \leq k \leq m, 1 \leq j \leq 3\}$$
$$\{x_i, \neg x_i \mid 1 \leq i \leq n\}$$

and $6m + n$ edges:

$$\begin{aligned} T &= \{x_i, \neg x_i\} \\ C_k &= \{(\ell_{k1}, \ell_{k2}), (\ell_{k2}, \ell_{k3}), (\ell_{k3}, \ell_{k1})\} \\ L_k &= \{(\ell_{k1}, v_{k1}), (\ell_{k2}, v_{k2}), (\ell_{k3}, v_{k3})\} \end{aligned}$$

where v_{kj} is x_i or $\neg x_i$ depending on which ℓ_{kj} is.

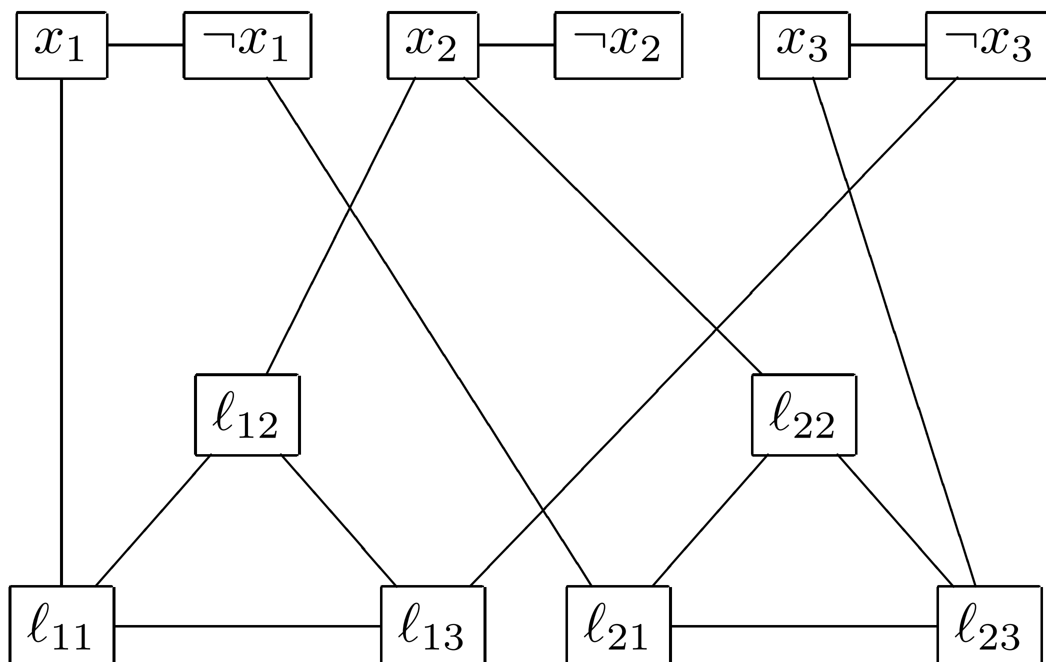
3-SAT to Vertex Cover (cont.)

The generated instance is $\langle G, 2m+n \rangle$.

For example,

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$$

is translated to the problem of whether this graph has a cover of size $2 \cdot 2 + 3 = 7$:



By this construction, a vertex cover must contain *at least* $2m+n$ nodes.

3-SAT to Vertex Cover (cont.)

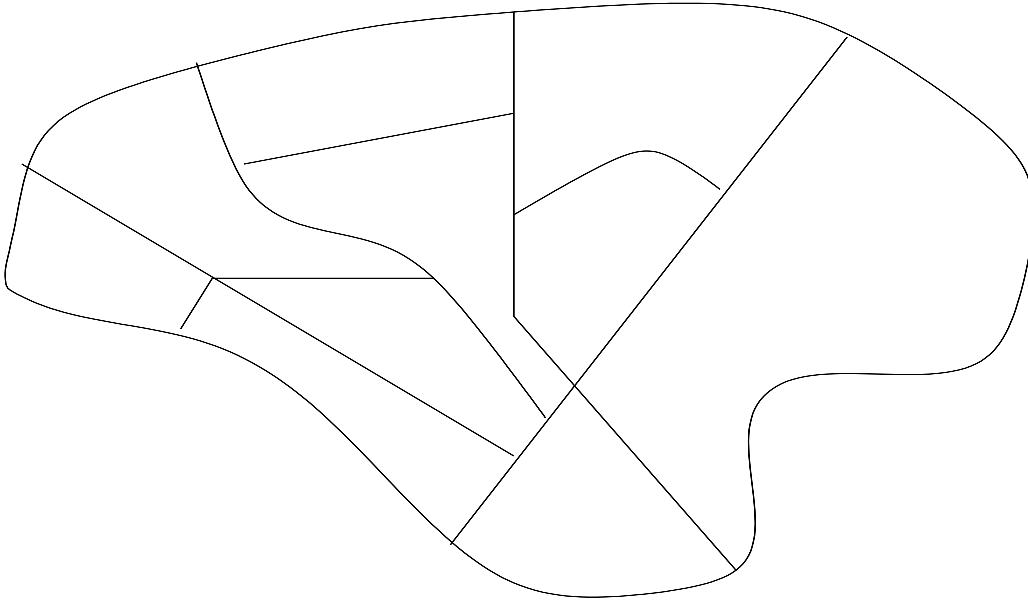
Clearly the graph is constructed in polynomial time.

We still need to show that G has a vertex cover of size $2m + n$ iff ϕ is satisfiable.

$\boxed{\Rightarrow}$ The cover defines a truth assignment t, as it includes exactly one of x_i and $\neg x_i$. This t makes each clause true: The ℓ_{kj} that is not covered is true.

$\boxed{\Leftarrow}$ Let t satisfy ϕ. That gives us n nodes (x_i or $\neg x_i$) of the cover. For each clause C_k, if ℓ_{kj} is satisfied, add the two nodes corresponding to the other two literals in C_k.

Map and Graph Colouring



Four colours suffice in the plane!

Tractability of n-colouring:

- 1-colouring is trivial.
- 2-colouring is easy.
- 3-colouring is NP-complete.
- 4-colouring is trivial.

NP and co-NP

With deterministic machines, reversing the yes/no outcome gives an equivalent problem.

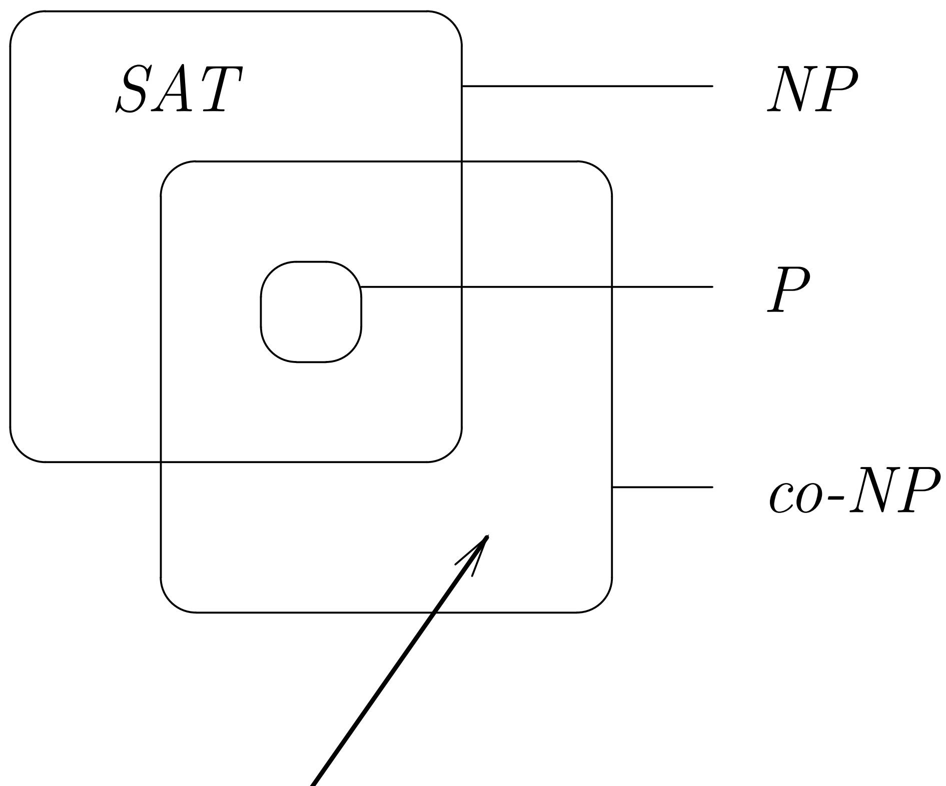
Using a non-deterministic machine to guess-and-check, it is not clear that this property should hold!

For example, how does one guess-and-check that a propositional formula is unsatisfiable?

It is not known whether NP is closed under complement. (Of course, if $P = NP$ then it is.)

The set of languages that are complements of the NP languages is called *co-NP*.

NP and co-NP (cont.)



The *dual* of SAT, that is, the *validity* problem for propositional formulas in CNF, is in *co-NP*.

Yes/No Problems Easier?

We have established that $VERTEX\text{-}COVER =$

$$\left\{ \langle G, k \rangle \;\middle|\; \begin{array}{l} G \text{ is a graph which has} \\ \text{a vertex cover of size } k \end{array} \right\}$$

is NP-complete.

However, the natural question to ask when we have a graph G is "what is G's smallest vertex cover?"

Are these "optimization" problems more difficult than the yes/no version?

No: For all the common problems in NPC, the complexity is the same, to within a polynomial.

Yes/No Problems Easier? (cont.)

Suppose we have a polynomial-time algorithm for finding the smallest vertex cover. We can immediately use that to answer the yes/no question.

Conversely, if we have an algorithm for the yes/no question, we can run that for values k in $1 \ldots n$, given a graph with n nodes.

In fact, with binary search the cost in running time is not a factor n, just a factor $\log(n)$.

Hamiltonian Path is NP-Complete

Recall the Hamiltonian path problem
HAMPATH =

$$\left\{ \langle G, s, t \rangle \;\middle|\; \begin{array}{l} G \text{ is a directed graph with a} \\ \text{Hamiltonian path from } s \text{ to } t \end{array} \right\}$$

where a Hamiltonian path visits each node in G exactly once.

HAMPATH is in *NP* since a Hamiltonian path can be guessed and checked in polynomial time.

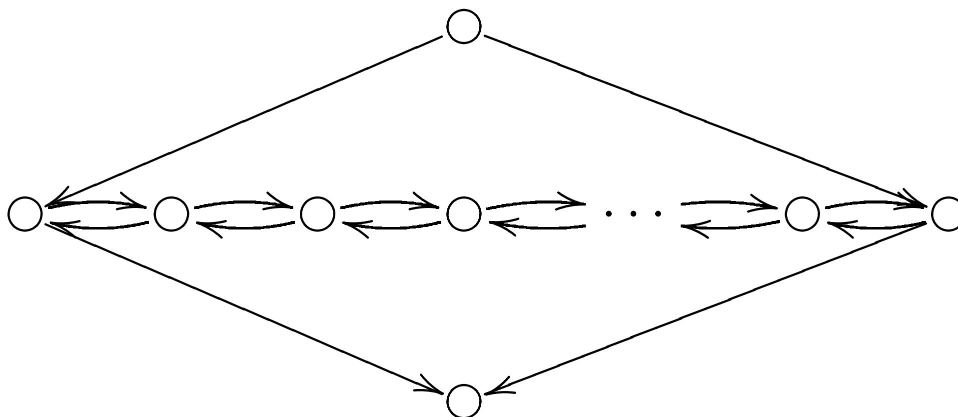
We now establish that *HAMPATH* is NP-complete by reduction from *3SAT*.

Reducing 3SAT to HAMPATH

Assume we have a *3SAT* formula ϕ.

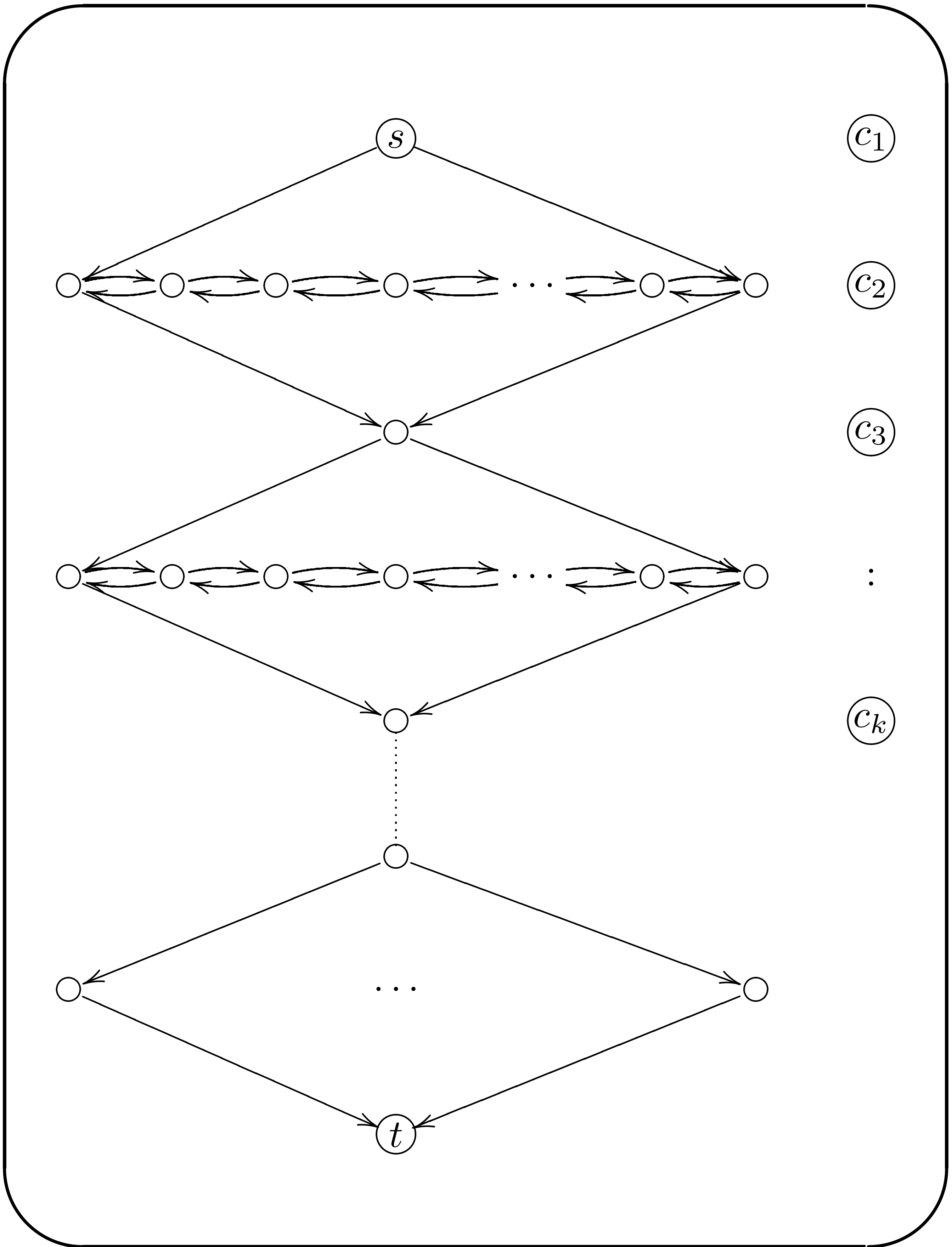
We need to construct (in polynomial time) a graph G such that a Hamiltonian path from s to t exists iff ϕ is satisfiable.

For each variable in ϕ we produce a diamond like this:



Apart from the four nodes that form the diamond, there are $2k$ nodes, two per clause.

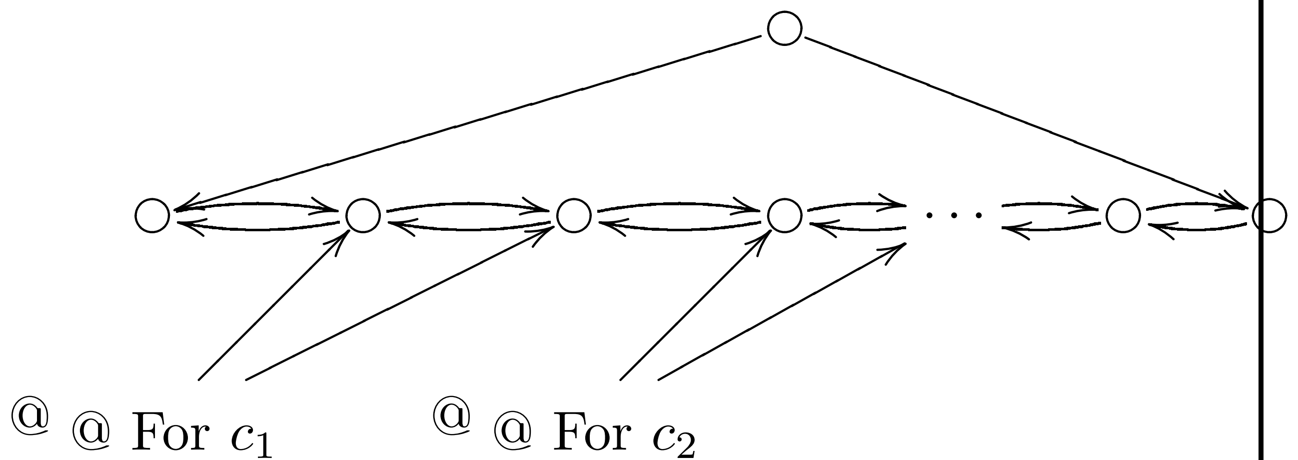
We also construct k separate nodes, one per clause.



s
c1
c2
c3
⋮
ck
t

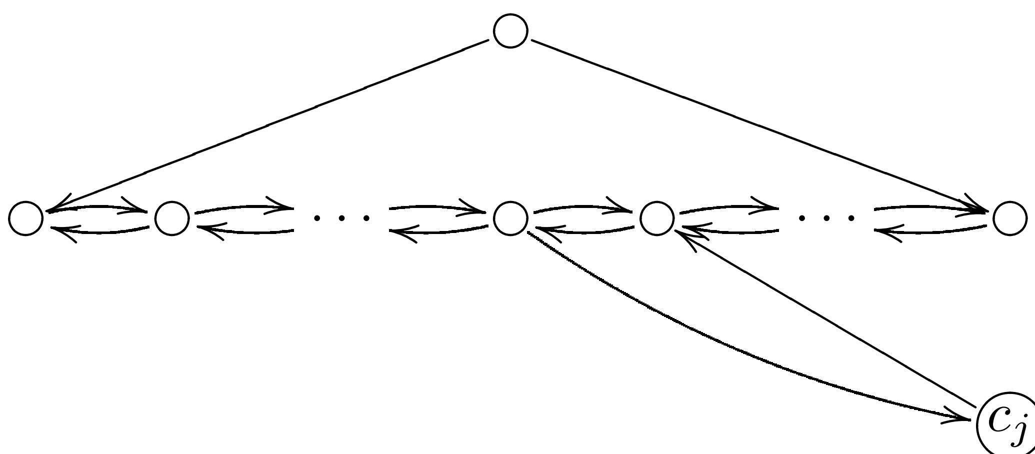
Reducing 3SAT to HAMPATH (cont.)

The chain of $2k$ nodes is a pair per clause:

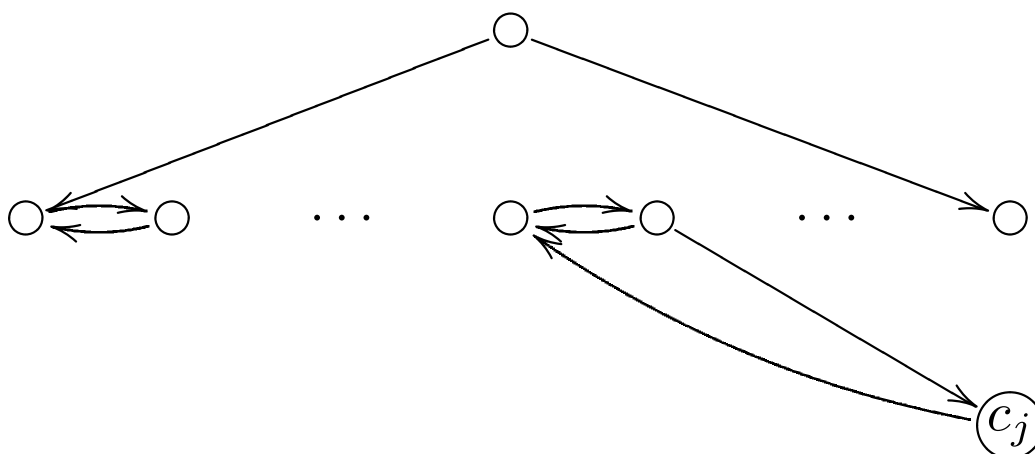


Reducing 3SAT to HAMPATH (cont.)

If this is x_i's chain, and x_i occurs in c_j, we add edges



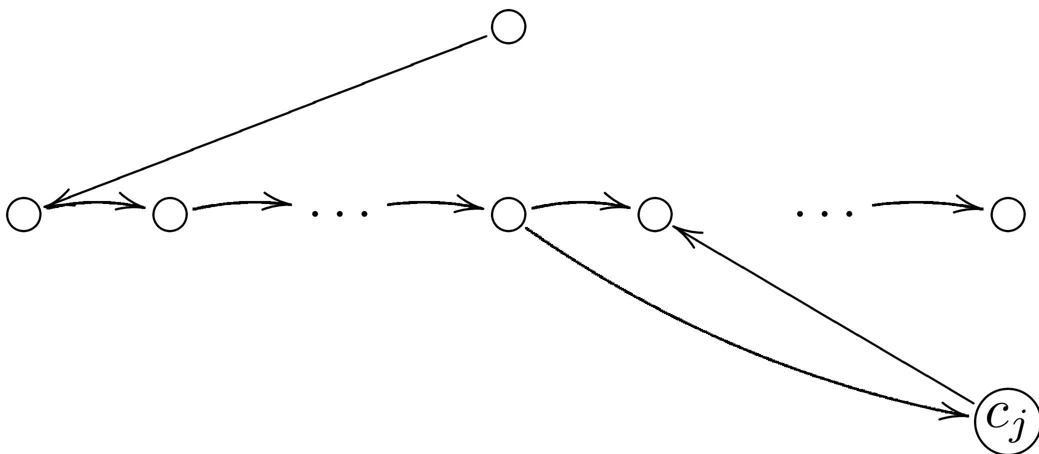
If $\neg x_i$ occurs in c_j, we add the edges



Reducing 3SAT to HAMPATH (cont.)

The idea is that a traversal down through the diamonds corresponds to a truth assignment.

Walking left-to-right through the ith chain corresponds to setting x_i to true.



Setting the variable to false corresponds to walking in the opposite direction.

Reducing 3SAT to HAMPATH (cont.)

Given a satisfying truth assignment, there is clearly a Hamiltonian path through the graph.

Note that all clause nodes c_j can be reached, but going from any particular chain node to c_j is optional.

Conversely, every Hamiltonian path through the graph from s to t determines a truth assignment.

Note that if a path takes us from a chain node to a clause node, we have to return to the neighbouring node in the chain, since that node could only be visited that way.

The graph is produced in polynomial time.

Reducing 3SAT to SUBSET-SUM

We already saw that

$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid \Sigma A = t \text{ for some } A \subseteq S\}$

is in NP.

Let us reduce *3SAT* to *SUBSET-SUM*.

Let ϕ have n variables and k clauses.

We construct a set of $2n + 2k$ numbers and a "target" sum

$$t = \underbrace{1\ 1\ 1 \cdots 1}_{n}\underbrace{3\ 3\ 3 \cdots 3}_{k}$$

The numbers in the set are chosen in a clever way, which we show by example.

3SAT to SUBSET-SUM (cont.)

$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$ yields

	1	2	3	c_1	c_2
x_1	1	0	0	1	0
$\neg x_1$	1	0	0	0	1
x_2	0	1	0	0	1
$\neg x_2$	0	1	0	1	0
x_3	0	0	1	1	1
$\neg x_3$	0	0	1	0	0
c_1	0	0	0	1	0
	0	0	0	1	0
c_2	0	0	0	0	1
	0	0	0	0	1
t	1	1	1	3	3

This set can be produced in polynomial time.

3SAT to SUBSET-SUM (cont.)

Note that each clause column (upper right quarter of the table) must have 0, 1, 2, or 3 1s (since this came from a formula in 3-CNF).

Zero 1s corresponds to the clause not being satisfied, anything else means satisfied.

The clause rows (bottom half of the table) are simply fillers that allow any clause row with 1 or more 1s to sum to 3.

Clearly a subset summing to t then determines a satisfying truth assignment for the formula, and vice versa.

Pseudo-Polynomial Algorithms

It is tempting to suggest that *SUBSET-SUM* can be solved in polynomial time by a table-filling method.

Pseudo-Polynomial Algorithms (cont.)

Given $\{2, 5, 8, 9\}$ and target sum 13, we create

	1	2	3	4
0	T	T	T	T
1	F	F		
2	T	T		
3	F	F		
4	F	F		
5	F	T		
6	F	F		
7	F	T		
$\vdots$				
13	F	F		

The ith column gives the sums that could be constructed using the first i elements of the set.

Pseudo-Polynomial Algorithms (cont.)

However, we require that numerical methods use a "reasonable" representation of numbers, such as decimal representation.

The table above may be polynomial in the size n of the set, but each element of the set, as well as the desired sum, are integers, and in the input, each is described by a string of length $O(\log(n))$.

So the table is in fact exponential in the size of input.

Nevertheless, intractability of $SUBSET\text{-}SUM$ depends strongly on the fact that large numbers are allowed. For sets of "small" numbers, the problem is tractable.