Time complexity

IIC3242

When is something efficiently computable?

Efficient = polynomial time

Does this make sense?

- We focus on distinction between e.g. n^3 and 2^n
- Exponential = brute-force search
- Polynomial = a bit smarter
  - Polynomial = realistically solvable
  - All deterministic models polynomially equivalent
  - Once we drop from exp to poly we gained some fundamental insight about the problem
  - All known poly = low degree poly

Complexity classes

How we define (any) complexity class?

Definition

Let $t : \mathbb{N} \to \mathbb{N}$ be a function. We define the (deterministic) **time complexity class** $\mathsf{DTIME}(t(n))$ as:

$$\mathsf{DTIME}(t(n)) = \{L \mid L \text{ is decided by an } O(t(n)) \text{ time deterministic Turing machine}\}.$$

For instance we have seen that the language of all palindromes belongs to $\mathsf{DTIME}(n^2)$.

Complexity class PTIME

PTIME= the class of all languages decidable in deterministic polynomial time on a single-tape TM

Definition

$$\text{PTIME} = \bigcup_k \text{DTIME}(n^k).$$

Some conventions

We will use high-level description of a TM and then illustrate how each step can be implemented in polynomial time on a TM

Recall that we always talk about languages

- To talk about graphs, numbers, etc. we will use encodings
- If G is a graph we use $\langle G \rangle$ for its string representation
- For a number x we use $\langle x \rangle$

Encoding is reasonable if:

- We can go from the natural representation to encoding in polynomial time
- For graphs this is the adjacency matrix, or list representation
- For numbers = base $b \geq 2$, but not unary

Examples of PTIME problems: PATH

$\mathsf{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a path from } s \text{ to } t\}$.

Brute-force = try all paths (exponential)

Poly-time algorithm:

1. Check that $\langle G, s, t\rangle$ is a directed graph with nodes s and t
2. Place mark on node s
3. Repeat the following until no new mark is produced:
4. Scan edges of G. If (a, b) is an edge with a marked and b not marked, then mark b.
5. If t is marked accept, otherwise reject.

Examples of PTIME problems: PATH

$\mathsf{PATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a path from } s \text{ to } t\}.$

Brute-force = try all paths (exponential)

Poly-time algorithm:

1. Check that $\langle G, s, t\rangle$ is a directed graph with nodes s and t
2. Place mark on node s
3. Repeat the following until no new mark is produced:
4. Scan edges of G. If (a, b) is an edge with a marked and b not marked, then mark b.
5. If t is marked accept, otherwise reject.

Another problem in PTIME: RELPRIME

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

Note that the length of the input is logarithmic in the size of the number, so trying out all the possible divisors is exponential

Poly-time algorithm (compute the g.c.d. using Euclid's algorithm):

R = On input $\langle x, y\rangle$ with x and y numbers in binary:

1. Repeat until $y = 0$
2. Assign $x \leftarrow x \bmod y$
3. Exchange x and y
4. If $x = 1$ accept, otherwise reject

Another problem in PTIME: RELPRIME

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

Note that the length of the input is logarithmic in the size of the number, so trying out all the possible divisors is exponential

Poly-time algorithm (compute the g.c.d. using Euclid's algorithm):

R = On input $\langle x, y\rangle$ with x and y numbers in binary:

1. Repeat until $y = 0$
2. Assign $x \leftarrow x \bmod y$
3. Exchange x and y
4. If $x = 1$ accept, otherwise reject

Another problem in PTIME: RELPRIME

To see we are in PTIME we need to show that steps 1–3 are executed logarithmic number of times in the size of the numbers

Observation: In stage 2 we cut x in half

- Apart maybe in the first time
- After this $x > y$ in stage 2 (we use mod in 2 and exchange them in 3)
- Then if $x/2 \geq y \Rightarrow x \bmod y < y \leq x/2$, so x halves
- If $x/2 < y \Rightarrow x \bmod y = x - y < x/2$, so x halves

Now, as x and y get exchanged in stage 3, each drops by half in stage 2. So we run steps 2–3 $min\{log_2 x, log_2 y\}$ times $=$ poly in the length of the input.

Another problem in PTIME: RELPRIME

To see we are in PTIME we need to show that steps 1–3 are executed logarithmic number of times in the size of the numbers

Observation: In stage 2 we cut x in half

- Apart maybe in the first time
- After this $x > y$ in stage 2 (we use mod in 2 and exchange them in 3)
- Then if $x/2 \geq y \Rightarrow x \bmod y < y \leq x/2$, so x halves
- If $x/2 < y \Rightarrow x \bmod y = x - y < x/2$, so x halves

Now, as x and y get exchanged in stage 3, each drops by half in stage 2. So we run steps 2–3 $min\{log_2 x, log_2 y\}$ times = poly in the length of the input.

Another problem in PTIME: RELPRIME

To see we are in PTIME we need to show that steps 1–3 are executed logarithmic number of times in the size of the numbers

Observation: In stage 2 we cut x in half

- Apart maybe in the first time
- After this $x > y$ in stage 2 (we use mod in 2 and exchange them in 3)
- Then if $x/2 \geq y \Rightarrow x \bmod y < y \leq x/2$, so x halves
- If $x/2 < y \Rightarrow x \bmod y = x - y < x/2$, so x halves

Now, as x and y get exchanged in stage 3, each drops by half in stage 2. So we run steps 2–3 $min\{log_2 x, log_2 y\}$ times $=$ poly in the length of the input.

More examples: now from logic

Consider the following restriction of SAT

More examples: now from logic

Consider the following restriction of SAT

Notation

- *A literal is a propositional variable or its negation:* p *and* $\neg q$

More examples: now from logic

Consider the following restriction of SAT

Notation

- *A literal is a propositional variable or its negation:* p *and* $\neg q$
- *A clause is a disjunction of literals:* $(p \vee q)$ *and* $(s \vee \neg q \vee \neg r)$

More examples: now from logic

Consider the following restriction of SAT

Notation

- *A literal is a propositional variable or its negation:* p *and* $\neg q$
- *A clause is a disjunction of literals:* $(p \vee q)$ *and* $(s \vee \neg q \vee \neg r)$
- *A Horn clause is a clause that has at most one positive literal (of the form* p*).*
  - $(s \vee \neg q \vee \neg r)$ *is a Horn clause and* $(p \vee q)$ *is not*

More examples: now from logic

Consider the following restriction of SAT

Notation

- *A literal is a propositional variable or its negation:* p *and* $\neg q$
- *A clause is a disjunction of literals:* $(p \vee q)$ *and* $(s \vee \neg q \vee \neg r)$
- *A Horn clause is a clause that has at most one positive literal (of the form* p*).*
  - $(s \vee \neg q \vee \neg r)$ *is a Horn clause and* $(p \vee q)$ *is not*
- *HORN-SAT* $= \{\varphi \mid \varphi$ *is a conjunction of Horn clauses and* φ *is satisfiable*$\}$

More examples: HORN-SAT

Theorem

HORN-SAT is in PTIME.

Proof: Follows by *unit-propagation*.

- Unit clause = consists of a single literal
- Repeat until no unit clauses are left:
- Pick a unit clause ℓ
- Remove every clause containing ℓ (apart the unit one)
- Remove $\neg\ell$ from any clause containing it
- If we get ℓ and $\neg\ell$ at some stage reject
- Accept otherwise

How do we prove that a problem is difficult?

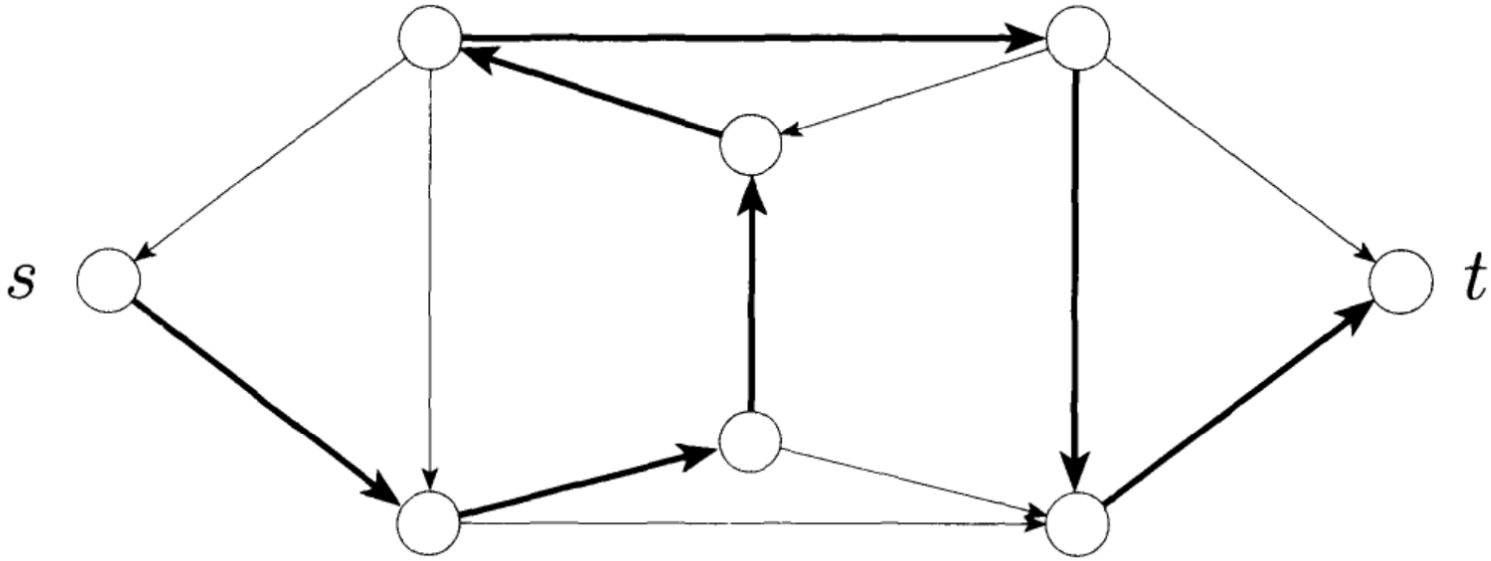
Poly-time is efficient, but what if a problem is intrinsically difficult (sometimes we know only an exponential algorithm).

Let us consider the following problem:

$$\text{HAMPATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}.$$

(Hamiltonian path = the one that passes each node precisely once)

Polynomial verifiability



An easy exponential algorithm: generate all paths of length n (number of nodes) and check if one is Hamiltonian

- Note that checking if a path is Hamiltonian = PTIME
- We don't know how to discover a HP quickly

This is called **polynomial verifiability**

Polynomial verifiability

Verifying if a path is Hamiltonian is much easier than determining if one exists.

Path = witness (certificate, proof) for the property

And this happens often:

$$\text{COMPOSITES} = \{\langle x \rangle \mid x = pq, \text{ for integers } p, q > 1\}.$$

Determining if a number is composite is difficult, but verifying if some other number is its divisor is easy

Polynomially verifiable problems

Definition

A **verifier** for a language L is a deterministic Turing machine V such that

$$L = \{w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c\}.$$

A **polynomial time verifier** V runs in time that is polynomial in w. A language is **polynomially verifiable** if it has a polynomial time verifier.

The string c = witness, certificate of membership in L
(for poly-time verifiers c is clearly polynomial)

For HAMPATH the witness is the Hamiltonian path, for COMPOSITES it is a divisor, and for SAT it is the satisfying assignment

The class NP

Since poly-verifiability is such a fundamental property let us consider the class of all the problems having this property

Definition

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

Contains many (most) problems of practical importance

But NP actually comes from nondeterministic polynomial time

The class NP

Here is an NTM deciding HAMPATH:
$N =$ On input $\langle G, s, t \rangle$, where G is a directed graph with nodes s and t:

1. Nondeterministically select a list $p_1, \ldots, p_m$ of nodes from G, where m is the number of nodes in G.
2. Check if some node is repeated in the list. If yes, reject.
3. If $s \neq p_1$, or $t \neq p_m$ reject.
4. Check that (p_i, p_{i+1}) is an edge in G. If not reject. Otherwise accept.

This clearly runs in poly-time.

And this is the general trend.

The class NP: not a misnomer

Theorem

A language is in NP (has a poly-time verifier) if and only if it is decided by some nondeterministic Turing machine running in polynomial time.

Proof: First let $L \in \text{NP}$ and let V be its verifier (a TM). Let V run in time $d \cdot n^k$, with d and k natural numbers. An equivalent NTM works as follows:

$N =$ On input w of length n:

1. Nondeterministically select a string c of length $d \cdot n^k$.
2. Run V on input $\langle w, c \rangle$.
3. If V accepts accept, otherwise reject.

The class NP: not a misnomer

For the other direction, assume that a NTM N decides L in poly-time. The verifier V works as follows:

$V =$ On input $\langle w, c \rangle$, with w and c strings:

1. Simulate N on input w using c as the description of the nondeterministic branch we are using (recall the proof that TM = NTM).
2. If this branch accepts accept, otherwise reject.

Defining NP via NTMs

Often it is easier to work with the following definition of NP:

Definition

$$\mathsf{NP} = \bigcup_k \mathsf{NTIME}(n^k).$$

As with deterministic TMs, here:

$$\mathsf{NTIME}(t(n)) = \{L \mid L \text{ is a language decided by a } O(t(n)) \text{ time nondeterministic Turing machine } \}.$$

(Same comments about nondeterministic models as when dealing with deterministic ones)

More NP problems: CLIQUE

A clique in a graph is a subgraph where every pair of nodes is connected by an edge. A k-clique is a clique of size k.

$$\text{CLIQUE} = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k-\text{clique}\}.$$

A poly-time verifier for CLIQUE:

$V =$ On input $\langle\langle G, k\rangle, c\rangle$:

1. Check that c is a set of k nodes in G.
2. Check that G contains an edge connecting each pair of nodes in c.
3. If both pass accept, otherwise reject.

More NP problems: CLIQUE

Alternatively, this can be done by a NTM:

$V =$ On input $\langle G, k \rangle$:

1. Nondeterministically guess a subset c of k nodes in G.
2. Check that c is a clique.
3. If it is accept, otherwise reject.

Idea: Polynomial guess followed by a polynomial check

PTIME vs NP and all that good stuff

Informally:

- PTIME = languages where membership is *decided* quickly
- NP = languages where membership can be *verified* quickly

Million dollar question

So is PTIME = NP?

We have no idea. Note that if not the best way to solve difficult problems is by brute-force search.

So how do we solve NP problems on a real computer?

- We already showed this (TM = NTM): using exponential time

$$\text{NP} \subseteq \text{EXPTIME} = \bigcup_k \text{DTIME}(2^{n^k})$$

Reductions and hardness

If a problem is in NP this does still not rule out that there is a PTIME algorithm solving it (even if PTIME $\neq$ NP)

So when do we say that a problem is difficult?

Good attempt: when it is as hard to solve as **any** problem in NP

This is called NP-hardness

Completeness

Also interesting: when does a problem represent a complexity class?

When solving it allows us to solve the entire class

This is called completeness for the class (e.g. NP-completeness)

For this the problem has to be hard for the class and belong to it

Reductions

To define hardness and completeness we need to be able to transform (reduce) one problem to another

I.e. we need to make an algorithm transforming an instance of problem A to an instance of problem B and this needs to be efficient

Note that here we actually compute a function, not just decide if a word is inside a language

Reductions

Definition (Computable functions)

A function $f : \Sigma^* \to \Sigma^*$ is a **polynomial time computable function** if some polynomial time Turing machine exists that halts with just $f(w)$ on its tape, when started with the input $w \in \Sigma^*$.

Definition (Karp reductions)

Language A is **polynomial time Karp reducible** to language B, written $A \leq_P B$, if there is a polynomial time computable function $f : \Sigma^* \to \Sigma^*$ such that:

$$w \in A \iff f(w) \in B.$$

How do reductions work?

Consider the following two problems:

$\text{3SAT} = \{\langle \varphi \rangle \mid \varphi \text{ is a satisfiable propositional formula in 3CNF}\}.$

3CNF = three literals per clause (e.g. $x_1 \vee \overline{x_2} \vee x_7$)

$\text{CLIQUE} = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k-\text{clique}\}.$

Theorem

3SAT is polynomial time reducible to CLIQUE.

Converting formulas to graphs

Take

$$\varphi = (a_1 \vee b_1 \vee c_1) \wedge \cdots \wedge (a_k \vee b_k \vee c_k).$$

The reduction f produces $\langle G, k \rangle$, with G as follows:

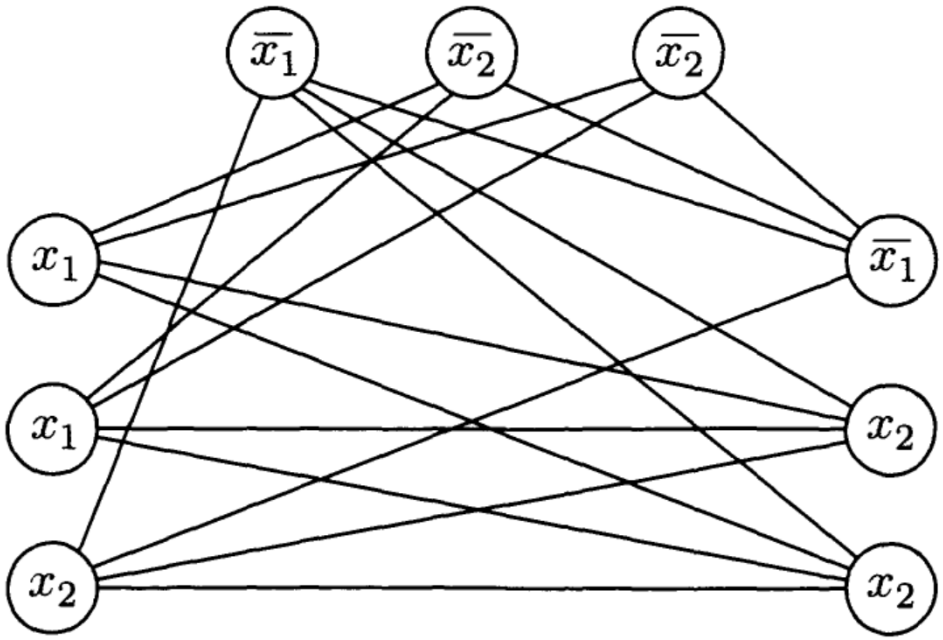
- G has k groups of 3 nodes
- Each triple corresponds to one clause of φ
- Each node is labelled as the literals in the corresponding clause
- No edges between nodes in one triple
- No edges between x and $\overline{x}$
- All other nodes have an edge

Converting formulas to graphs

If

$$\varphi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$$

Then we get:



Converting formulas to graphs

Recall that we need to show that $w \in 3SAT \iff f(w) \in CLIQUE$

If φ has a satisfying assignment, then we just select a single node in each triple that is true under this assignment.

If G has a k-clique, then this clique selects precisely one node from each triple. If we make this literal true we get a satisfying assignment.

Consequence: If we know how to solve CLIQUE we know how to solve 3SAT

Hardness and completeness

When is a problem as hard as any problem in NP?

Definition (Hardness)

A language B is **NP-hard** if for any $A \in \text{NP}$ we have $A \leq_P B$.

When does a problem represent NP?

Definition (Completeness)

A language B is **NP-complete** if B is NP-hard and $B \in \text{NP}$.

Why is this important?

Proposition

1. *(Transitivity) If* $A \leq_P B$ *and* $B \leq_P C$ *then* $A \leq_P C$.
2. *If* A *is NP-hard and* $A \in PTIME$ *then* $PTIME = NP$.
3. *If* A *is NP-complete, then* $A \in PTIME$ *if and only if* $PTIME = NP$.
4. *If* A *is NP-complete and* $A \leq_P B$, *for* $B \in NP$, *then* B *is NP-complete.*

Why is each of these items important?

The Cook-Levin theorem

So do we know of a problem that is NP-complete?

Theorem

SAT is NP-complete.

Recall:

$$\text{SAT} = \{\langle \varphi \rangle \mid \varphi \text{ is a satisfiable propositional formula.}\}$$

Proof: Computer circuitry uses AND, OR and NOT, just as boolean formulas.

The Cook-Levin theorem: proof

Take arbitrary $A \in \text{NP}$

Let N be a NTM deciding A in time n^k, for some k

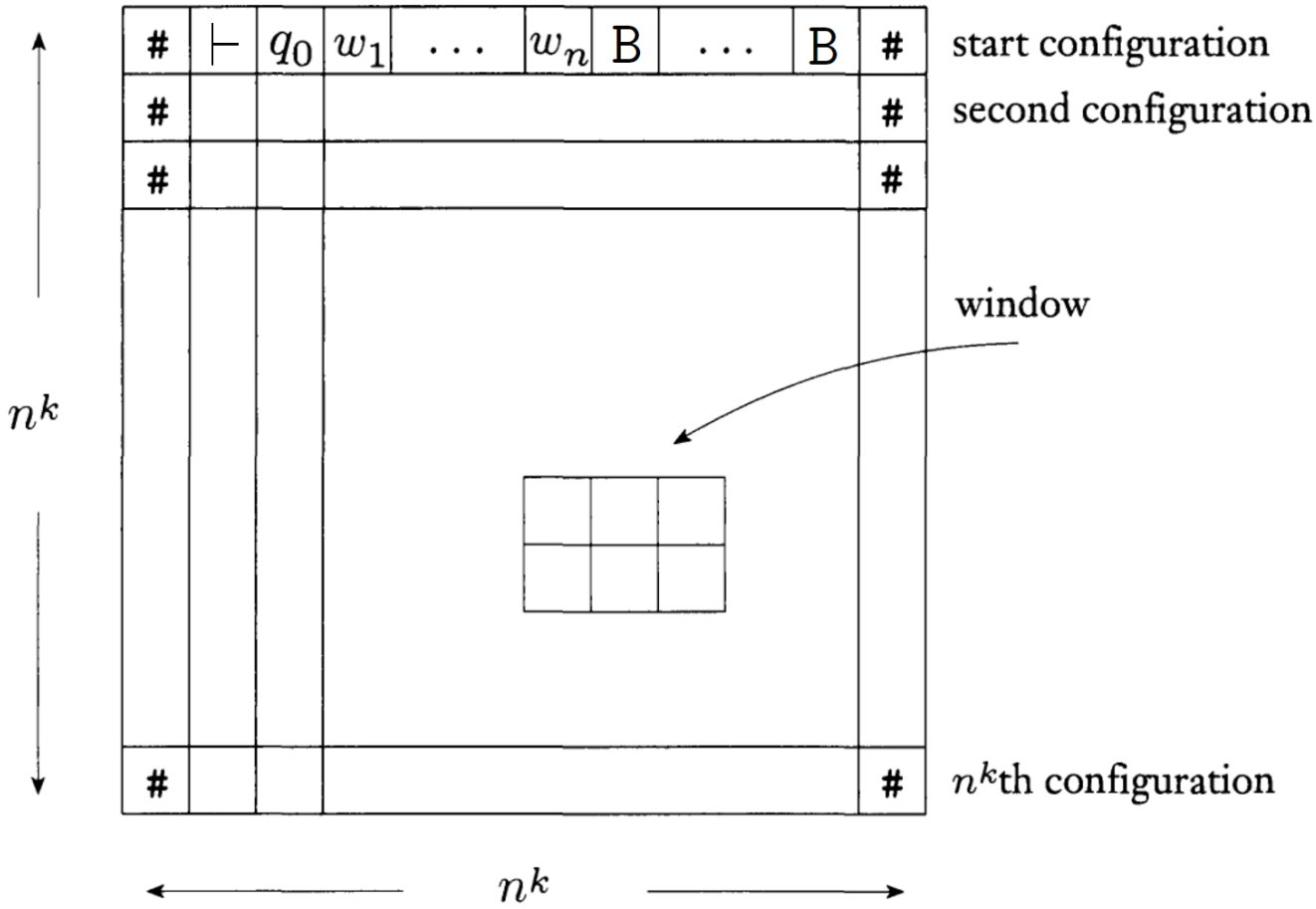
Wlog we assume that N works in time $n^k - 3$

- If you want to be very precise replace this by $t_M(n)$
- And show that for every NTM M there is an equivalent one that takes *precisely* $t_M(n)$ steps on every branch on every input of length n
- The same holds for deterministic machines

We describe a branch of computation of N using a **tableau**

The Cook-Levin theorem: proof

A tableau:



The Cook-Levin theorem: proof

A tableau is an $n^k \times n^k$ table:

- Rows are configurations of N on input w
- Rows start and end with $\#$
- First row is the initial configuration
- Each row yields the next one
- A tableau is **accepting** if some row is an accepting configuration

N accepts w iff there is an accepting tableau of N on w

The Cook-Levin theorem: proof

For a word w we construct a formula φ_w such that:

There is an accepting tableau of N on w iff φ_w is satisfiable

Let $C = Q \cup \Gamma \cup \{\#\}$, with Q, Γ from N

φ_w uses variables $x_{i,j,s}$, where $i, j \in \{1, \ldots, n^k\}$ and $s \in C$

Each entry of a tableau is called a **cell**

- $cell[i, j]$ is the cell at position (i, j)

Our formula will be:

$$\varphi_w = \varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{move} \wedge \varphi_{accept}$$

The Cook-Levin theorem: proof

$x_{i,j,s} = 1$ means that $cell[i,j] = s$

φ_{cell} makes sure that each cell has precisely one symbol in it:

$$\varphi_{cell} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{\substack{s,t \in C \\ s \neq t}} \left(\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}} \right) \right) \right]$$

The Cook-Levin theorem: proof

φ_{start} codes the initial configuration in the first row:

$$\begin{aligned}\varphi_{start} \quad = \quad & x_{1,1,\#} \wedge x_{1,2,\vdash} \wedge x_{1,3,q_0} \wedge \\ & x_{1,4,w_1} \wedge x_{1,5,w_2} \wedge \cdots \wedge x_{1,n+3,w_n} \wedge \\ & x_{1,n+4,\mathtt{B}} \wedge \cdots \wedge x_{1,n^k-1,\mathtt{B}} \wedge x_{1,n^k,\#}\end{aligned}$$

Here we assume $w = w_1 \cdots w_n$

The Cook-Levin theorem: proof

φ_{accept} makes sure there is an accepting configuration:

$$\varphi_{accept} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{accept}}$$

The Cook-Levin theorem: proof

φ_{move} codes how the machine N works

Namely, it makes sure that each row yields the next one

This is done by making sure that each 2×3 window of cells is **legal**

A window is legal if it does not violate N's transition function

The Cook-Levin theorem: proof

Say that in N we have:

- $\delta(q_1, a) = \{(q_1, b, \rightarrow)\}$ and
- $\delta(q_1, b) = \{(q_2, c, \leftarrow), (q_2, a, \rightarrow)\}$

These are legal windows:

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	$\vdash$	a
#	$\vdash$	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

Why?

The Cook-Levin theorem: proof

Say that in N we have:

- $\delta(q_1, a) = \{(q_1, b, \rightarrow)\}$ and
- $\delta(q_1, b) = \{(q_2, c, \leftarrow), (q_2, a, \rightarrow)\}$

These windows are not legal:

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_1	a	a

(c)

b	q_1	b
q_2	b	q_2

Why?

The Cook-Levin theorem: proof

Claim

If the top row of the tableau is the initial configuration and **every** *window in the tableau is legal, then each row of the tableau is a configuration that legally follows from the preceding one.*

Hint: Look at the central cell of the top row in a window.

The Cook-Levin theorem: proof

Now we can define φ_{move} easily:

$$\varphi_{move} = \bigwedge_{1<i\leq n^k, 1<j<n^k} (\text{ the } (i,j) \text{ window is legal })$$

The formula the (i,j) window is legal simply codes any legal configuration:

$$\bigvee_{\substack{a_1,\ldots,a_6 \\ \text{is a legal window}}} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

Note: the six cells of each window can be only set in a predefined way (i.e. this is polynomial)

The Cook-Levin theorem: proof

Clearly φ_w is satisfiable iff N accepts w

To finish the proof we need to make sure that φ_w can be computed in polynomial time

But this is easy: homework

The Cook-Levin theorem: consequences

Now we have a real problem that is NP-complete

To show another problem in NP to be NP-complete we can now just give a reduction from SAT to his problem (no need to code Turing machines)

Instead of SAT we will often use 3SAT

Proposition

3SAT is NP-complete.

Prove this!

(Note that this also means that CLIQUE is NP-complete)

More NP-complete problems: HAMPATH

Recall:

$$\text{HAMPATH} = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}.$$

Theorem

HAMPATH is NP-complete.

Proof: We already know that HAMPATH is in NP

For NP-hardness we do a reduction from 3SAT to HAMPATH

More NP-complete problems: HAMPATH

For a 3CNF formula φ we construct a directed graph G with nodes s, t such that:

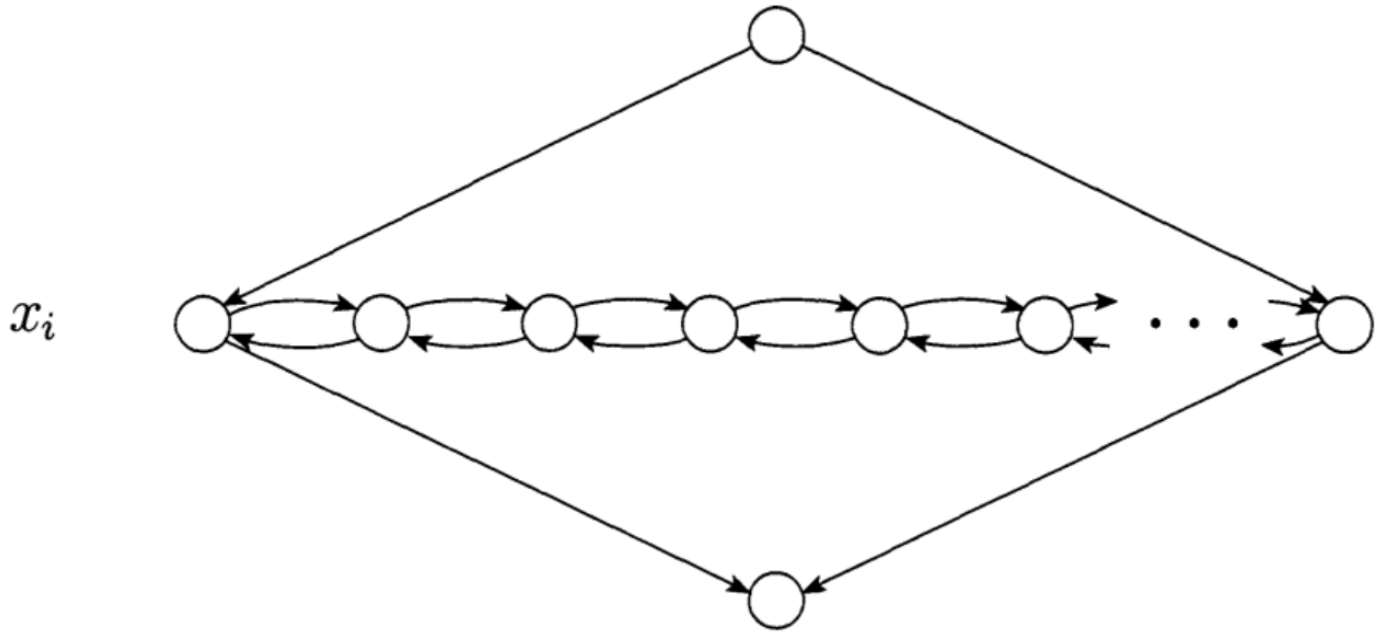
φ is satisfiable iff G has a Hamiltonian path fro s to t

$$\varphi = (u_1 \vee v_1 \vee w_1) \wedge (u_2 \vee v_2 \vee w_2) \wedge \cdots \wedge (u_k \vee v_k \vee w_k)$$

- u_i, v_i, w_i are variables x_i of $\overline{x_i}$
- $x_1, \ldots, x_\ell$ are all the variables in φ
- $c_1, \ldots, c_k$ are the k clauses of φ

More NP-complete problems: HAMPATH

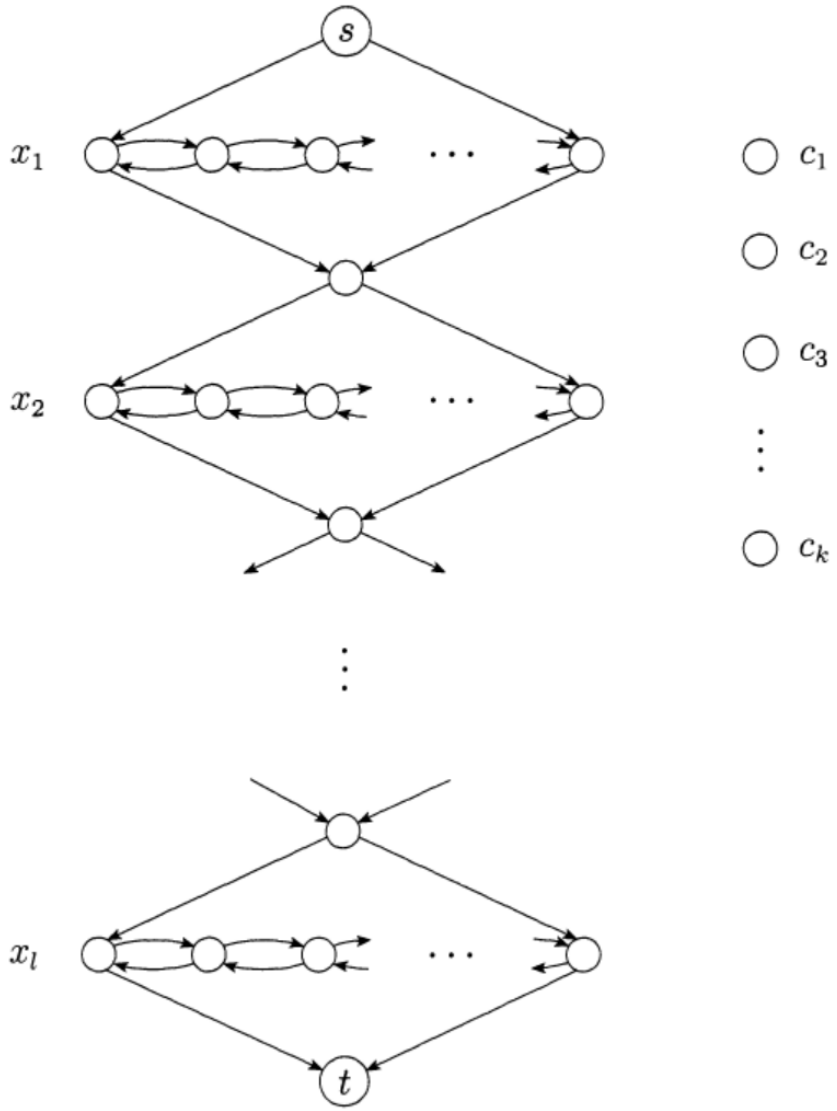
Variable x_i is represented by a diamond structure:



and a clause c_j by a single node:

More NP-complete problems: HAMPATH

The structure of the graph G is as follows:

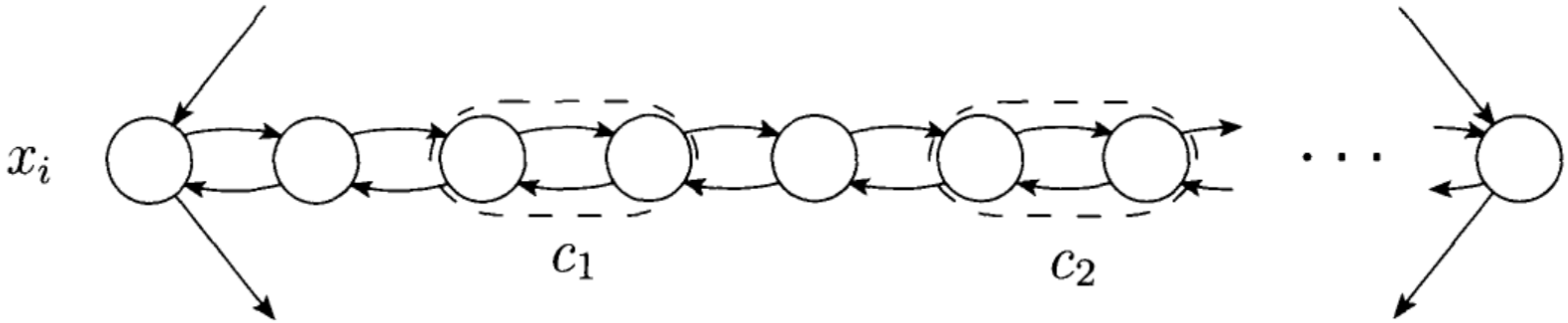


More NP-complete problems: HAMPATH

We need to connect diamonds (variables) with the clauses

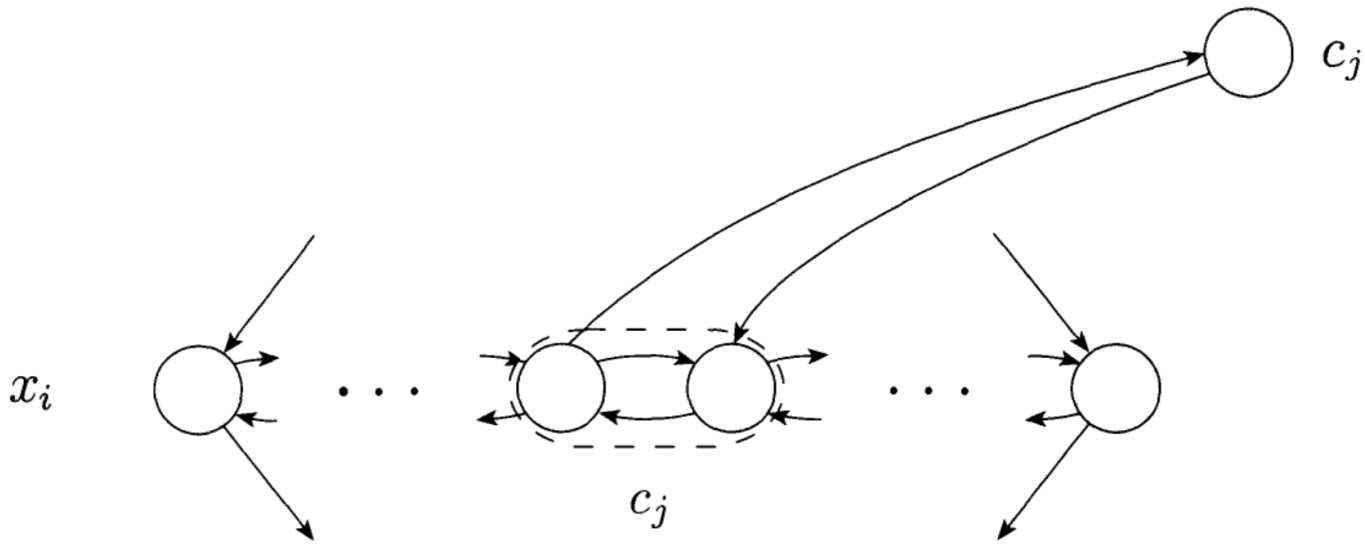
For this we use the horizontal rows in each diamond:

- They consist of $3k + 3$ nodes
- Two outer ones and $3k + 1$ inside
- One pair for each clause, plus one separating each pair



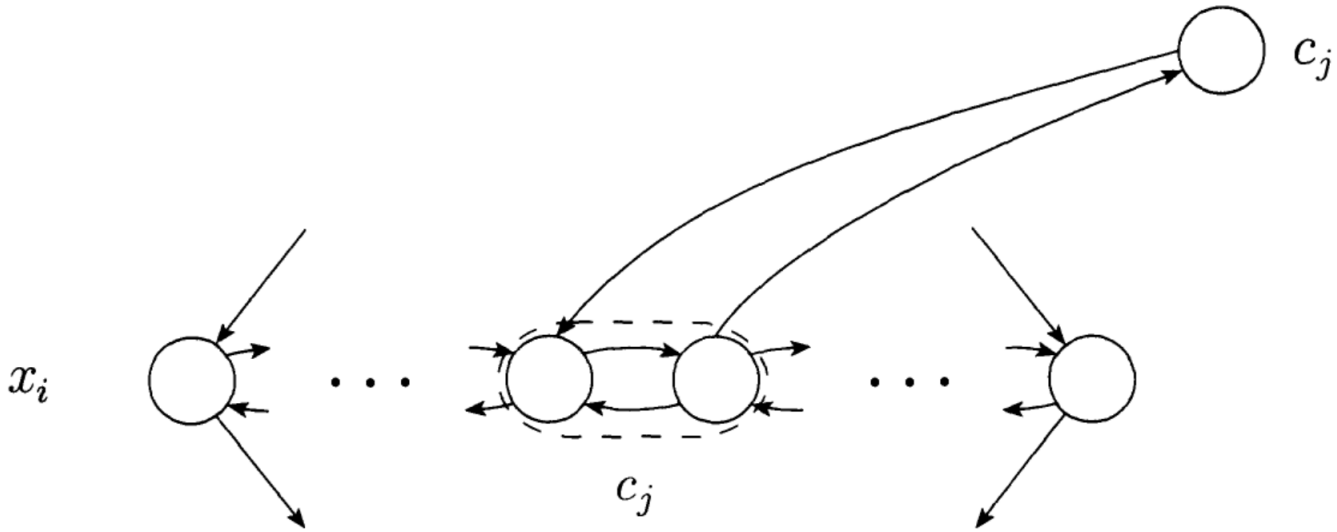
More NP-complete problems: HAMPATH

If x_i appears in the clause c_j we do the following:



More NP-complete problems: HAMPATH

If $\overline{x_i}$ appears in the clause c_j we do the following:



More NP-complete problems: HAMPATH

G can obviously be computed in polynomial time (it terms of φ), so we only need to prove that the reduction works.

Namely, that:

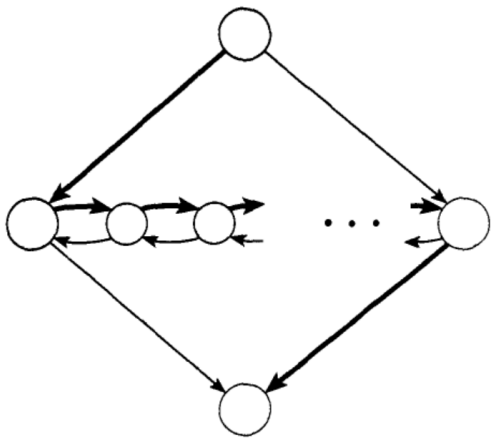
φ is satisfiable iff G has a Hamiltonian path from s to t

More NP-complete problems: HAMPATH

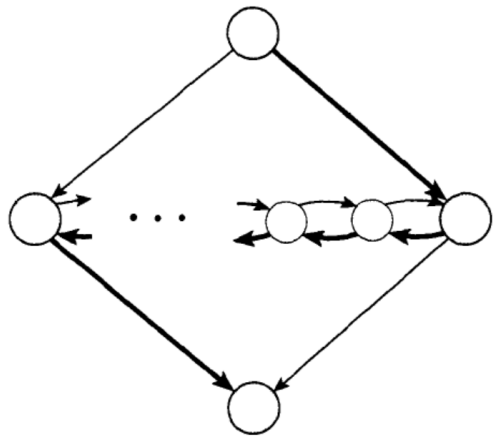
Suppose that φ is satisfiable

Hamiltonian path: start at s and:

- zig-zag if x_i is true
- zag-zig if it is false



zig-zag



zag-zig

To go to clauses pick one literal that is true and go to the clause

More NP-complete problems: HAMPATH

Assume that there is a Hamiltonian path from s to t

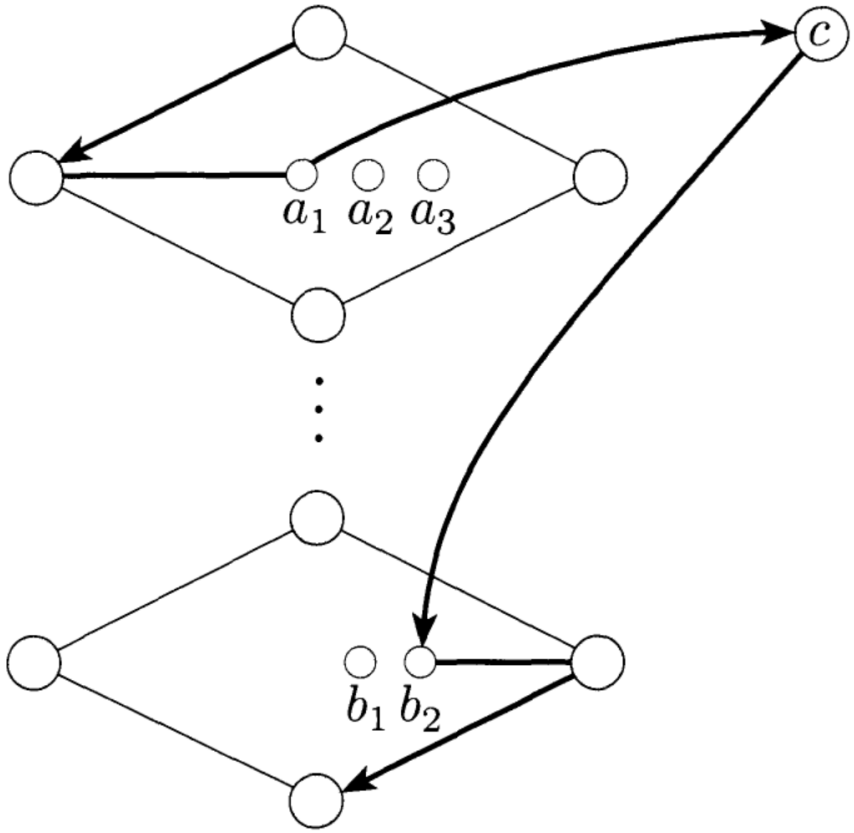
If the path is like in the previous direction of the proof we can get an assignment making φ true

But all Hamiltonian paths in G are like this

We prove this by contradiction

More NP-complete problems: HAMPATH

That is, a path not of that shape does the following:



Show that such path can not be Hamiltonian □

More NP-complete problems: SUBSETSUM

Not just formulas and graphs are NP-hard:

$$\text{SUBSETSUM} = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\} \text{ and there is a subset } \{y_1, \ldots, y_\ell\} \text{ such that } \Sigma_{i=1}^{\ell} y_i = t\}.$$

Here we are basically asking if a set of numbers has a subset summing up to a value t

Theorem

SUBSETSUM is NP-complete.

Proof: The upper bound is trivial (guess and check)

More NP-complete problems: SUBSETSUM

To show NP-hardness we reduce 3SAT to SUBSETSUM

Let φ be a 3CNF formula with:

- $x_1, \ldots x_\ell$ variables of φ
- $c_1, \ldots, c_k$ clauses of φ

Our set S will have the numbers:

- $y_1, z_1, \ldots y_\ell, z_\ell$ to code the variables
- $g_1, h_1, \ldots, g_k, h_k$ to code the clauses

The following table has numbers in S in **decimal notation**

(for $\varphi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \ldots) \wedge \cdots \wedge (\overline{x_3} \vee \ldots \vee \ldots)$)

More NP-complete problems: SUBSETSUM

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

More NP-complete problems: SUBSETSUM

So in the table y_i, z_i have two parts (representing x_i):

- On the left they have 1 followed by $\ell - i$ 0s
- On the right y_i has 1 in position j if x_i appears in clause j, 0 otherwise
- On the right z_i has 1 in position j if $\overline{x_i}$ appears in clause j, 0 otherwise

The numbers g_j, h_j represent clauses and are equal:

- One 1 followed by $k - j$ 0s

The target t is ℓ 1s followed by k 0s

More NP-complete problems: SUBSETSUM

Clearly polynomial, so we need to show:

$$\varphi \text{ is satisfiable iff } S \text{ has a subset summing up to } t$$

Assume that φ is satisfiable:

- If x_i is true select y_i, otherwise select z_i
- This gives us 1 in first ℓ positions
- In last k we always have between 1 and 3 (each clause has at least one and at most 3 true literals)
- Select as many gs and hs as needed to get 3 in each position
- We have a solution to SUBSETSUM

More NP-complete problems: SUBSETSUM

Conversely, assume that a subset of S sums to t.

- Then we always select either y_i or z_i
- No carry when summing digits (decimal) occurs
- x_i is true if we selected y_i, false otherwise
- In the final k columns we always get 3
- Here at most 2 can come from gs and hs
- But this corresponds to the literal being made to true in a clause
- So φ is satisfiable

□

What about PTIME-completeness?

Let's try!

A problem A is PTIME-complete if:

- A belongs to PTIME; and
- For every $B \in \text{PTIME}$ we have that $B \leq_P A$ (i.e. B polynomially reduces to A)

Will this work?

What about PTIME-completeness?

Consider the following problem:

$$\text{ZERO} = \{\langle n \rangle \mid n \text{ is a natural number and } n = 0\}.$$

Under the previous definition ZERO is PTIME-complete:

- It is clearly inside PTIME
- If B is in PTIME, then B polytime reduces to ZERO:
  - Take M a PTIME machine for B:
  - On input w compute $M(w)$
  - If M accepts map to zero
  - Otherwise map to 1

This can be shown for any language $A \neq \emptyset, \Sigma^*$

What about PTIME-completeness?

The issue: *our reduction is too hard*

To talk about reductions inside a complexity class, the reduction should be easier than the problems it reduces from and to

I.e. complete problem: the most difficult one in a sense that we can get from any problem in the class to the complete problem *easily* and then use it to solve the original problem

Solution: LOGSPACE reductions

So our reductions need to be easier than PTIME

We will use LOGSPACE reductions

When is a function LOGSPACE computable?

Enters space complexity

LOGSPACE reductions

If we define space complexity as for time we get:

- A machine uses space $f(n)$ if it scans only $O(f(n))$ cells when processing an input of length n

So LOGSPACE reductions cannot even read the input

Which does not make that much sense (i.e. then what is the input for?)

What does it mean to work with some space? (Think of palindromes)

Turning machines with input and output

Definition

A (deterministic) k-**tape Turing machine with input and output** is an ordinary k-string Turing machine where for each $\delta(q, a_1, \ldots, a_k) = (q', b_1, \ldots, b_k, D_1, \ldots, D_k)$ we have:

1. $a_1 = b_1$ (the first tape is read only)
2. $D_k \neq \leftarrow$; and $b_k \neq \mathtt{B} \Rightarrow a_k = \mathtt{B}$ (the last tape is write only)
3. If $a_1 = \mathtt{B}$ then $D_1 = \leftarrow$ (don't wonder off the input).

We can read the input and provide the output

- First tape = read-only input tape
- Last tape = write-only output tape
- Tapes 2 to $k - 1$ are work tapes

Turning machines with input and output

So we can define space complexity without charging for reading the input/writing the output:

Definition

If M is a k-tape Turing machine with input and output, then the space used by M on input w is the total number of cells accessed by the heads of M on its work tapes (tapes 2 through $k-1$).

Turning machines with input and output

When talking about space complexity we will be using machines with input/output

(Deterministic) Space complexity classes:

Definition

The space complexity class $\text{DSPACE}(f(n))$ consists all languages L that are decided by some k-tape Turing machine with input and output that uses at most $O(f(n))$ space.

Note that we allow any $k(\geq 3)$. Why?

When do we not need the output tape? (Deciders)

LOGSPACE reductions

Definition

A function $f : \Sigma^* \to \Sigma^*$ is **LOGSPACE-computable** if there is a k-tape Turing machine with input and output which, when started with w on the input tape terminates with $f(w)$ on its output tape and the space used by the machine is $O(\log n)$, for $n = |w|$.

Definition

A language A is **LOGSPACE-reducible** to the language B, written $A \leq_L B$, if there is a LOGSPACE-computable function f such that:

$$w \in A \iff f(w) \in B.$$

PTIME-completeness

Now it will make sense:

Definition

A problem B is **PTIME-complete** if $B \in \text{PTIME}$ and for every other $A \in \text{PTIME}$ we have that $A \leq_L B$.

Now we can not overshoot with the reduction.

PTIME-completeness

The usual properties of reductions still hold.

Proposition

1. *If* $A \leq_L B$ *and* $B \in PTIME$, *then* $A \in PTIME$.
2. *If* $A \leq_L B$ *and* $B \leq_L C$, *then* $A \leq_L C$.

Proof: It is enough to show transitivity. Note that the composition of reductions is a reduction. But just computing the result of the first reduction and feeding it to the second one might use too much space.

- *Question:* In LOGSPACE reductions how big can the output string be?

Reductions compose: continued

- f reduction from A to B and M_f the machine computing it
- g reduction from B to C and M_g the machine computing it

The idea is to simulate M_g on $M_f(w)$ without writing $M_f(w)$:

- We write the cursor position in $M_f(w)$ that M_g needs
- If the cursor goes right: produce the next bit of $M_f(w)$ (i.e. run M_f until it does so)
- If the cursor moves left: run M_f from the start until the needed bit is produced
- Key observation: $|f(w)|$ is poly in $|w|$, so the cursor is of size $O(log|w|)$

(Picture on the whiteboard) □

Fun fact

All the Karp reductions we used are/can be made LOGSPACE

In fact, the theory of NP-completeness works the same with LOGSPACE reductions (Why?)

For classes above NP polynomial-time reductions are fine

For PTIME and below we use LOGSPACE

Please be confused by this (at least a bit)!

A PTIME-complete problem

Recall:

Notation

- *A literal is a propositional variable or its negation:* p *and* $\neg q$
- *A clause is a disjunction of literals:* $(p \vee q)$ *and* $(s \vee \neg q \vee \neg r)$
- *A Horn clause is a clause that has at most one positive literal (of the form* p*).*
  - $(s \vee \neg q \vee \neg r)$ *is a Horn clause and* $(p \vee q)$ *is not*
- *HORN-SAT* $= \{\varphi \mid \varphi$ *is a conjunction of Horn clauses and* φ *is satisfiable*$\}$

PTIME-completeness: HORN-SAT

Theorem

HORN-SAT is PTIME-complete.

PTIME-completeness: HORN-SAT

Theorem

HORN-SAT is PTIME-complete.

Proof: We already showed that HORN-SAT $\in$ PTIME.

PTIME-completeness: HORN-SAT

Theorem

HORN-SAT is PTIME-complete.

Proof: We already showed that HORN-SAT $\in$ PTIME.

To show that HORN-SAT is PTIME-hard.

PTIME-completeness: HORN-SAT

Theorem

HORN-SAT is PTIME-complete.

Proof: We already showed that HORN-SAT $\in$ PTIME.

To show that HORN-SAT is PTIME-hard.

- We will show that $\overline{\text{HORN-SAT}}$, the complement of HORN-SAT is PTIME-hard

PTIME-completeness: HORN-SAT

Theorem

HORN-SAT is PTIME-complete.

Proof: We already showed that HORN-SAT $\in$ PTIME.

To show that HORN-SAT is PTIME-hard.

- We will show that $\overline{\text{HORN-SAT}}$, the complement of HORN-SAT is PTIME-hard
- Why is this enough?

HORN-SAT is PTIME-hard

Take $L \in \text{PTIME}$. We have to show that L is logspace reducible to $\overline{\text{HORN-SAT}}$.

That is: there is a function $f : \Sigma^* \to \Sigma^*$ such that

- f is computable in LOGSPACE and;
- for every $w \in \Sigma^*$: $w \in L$ if and only if $f(w) \in \overline{\text{HORN-SAT}}$

Notation

$f(w) = \varphi_w$

HORN-SAT is PTIME-hard

Since $L \in \text{PTIME}$, there is a deterministic TM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ such that:

- M halts on every input
- $L = L(M)$
- $t_M(n)$ is $O(n^c)$, with $c > 0$ a natural number

We we simulate the run of M on input w using the formula φ_w: M accepts w if and only if φ_w **is not satisfiable**.

HORN-SAT is PTIME-hard

To simulate M we use the following variables:

$$\begin{array}{lcl} s_{t,p,a} & : & t \in [0, t_M(n)],\ p \in [0, t_M(n)] \text{ and } a \in \Gamma \\ c_{t,p} & : & t \in [0, t_M(n)] \text{ and } p \in [0, t_M(n)] \\ e_{t,q} & : & t \in [0, t_M(n)] \text{ and } q \in Q \end{array}$$

- $s_{t,p,a}$ is 1 if the symbol as time t in position p is a
- $c_{t,p}$ is 1 if the head position at time t is p
- $e_{t,q}$ is 1 if the state at time t is q

φ_w is defined as $\varphi_I \wedge \varphi_N \wedge \varphi_\delta$

HORN-SAT is PTIME-hard

φ_I: The initial state.

We assume that the input $w = a_1 \cdots a_n$

$$e_{0,q_0} \wedge c_{0,1} \wedge s_{0,0,\vdash} \wedge \left(\bigwedge_{p=1}^{n} s_{0,p,a_p} \right) \wedge \left(\bigwedge_{p=n+1}^{t_M(n)} s_{0,p,\mathrm{B}} \right)$$

HORN-SAT is PTIME-hard

φ_N: The machine does not accept w

$$\bigvee_{t=0}^{t_M(n)} e_{t,q_{reject}}$$

HORN-SAT is PTIME-hard

But this is not a Horn-formula!!!

φ_N: The machine does not accept w

$$\bigwedge_{t=0}^{t_M(n)} \neg e_{t,q_{accept}}$$

HORN-SAT is PTIME-hard

φ_δ defines how the machine works (we represent $\leftarrow$ as -1, $\square$ as 0 and $\rightarrow$ as 1):

$$\bigwedge_{t=0}^{t_M(n)-1} \bigwedge_{p=0}^{t_M(n)-1} \bigwedge_{(q,a,q',b,\ell)\in\delta} \Big[\Big(\neg e_{t,q} \vee \neg c_{t,p} \vee \neg s_{t,p,a} \vee e_{t+1,q'} \Big) \wedge \Big(\neg e_{t,q} \vee \neg c_{t,p} \vee \neg s_{t,p,a} \vee c_{t+1,p+\ell} \Big) \wedge \Big(\neg e_{t,q} \vee \neg c_{t,p} \vee \neg s_{t,p,a} \vee s_{t+1,p,b} \Big) \wedge \bigwedge_{p'\in([0,t_M(n)]\setminus\{p\})} \bigwedge_{d\in\Gamma} \Big(\neg e_{t,q} \vee \neg c_{t,p} \vee \neg s_{t,p,a} \vee \neg s_{t,p',d} \vee s_{t+1,p',d} \Big) \Big]$$

HORN-SAT is PTIME-hard

We need to prove that $w \in L$ if and only if φ_w **is not** satisfiable.

- Do this as an exercise!

We also need the following:

- φ_w is a conjunction of Horn clauses
- φ_w can be constructed in LOGSPACE($\log t_M(n)$ is $O(\log n)$)

□

Practice time

Exercise

Modify the previous proof to show that SAT is NP-complete.

Another PTIME-complete problem

A system of integer linear inequalities is of the form:

$$A\vec{x} \quad \leq \quad \vec{b}$$

where:

- A is an integer matrix of size $m \times n$
- $\vec{x}$ is a vector of variables of size $n \times 1$
- $\vec{b}$ is a vector of integers of size $m \times 1$

A vector $\vec{c}$ of *real numbers* of size $n \times 1$ is a solution to the system if $A\vec{c} \leq \vec{b}$.

PTIME-completeness: Linear programming

A fundamental problem in engineering:

$$\text{PROG-LIN} = \{(A, \vec{b}) \mid A\vec{x} \leq \vec{b} \text{ is a system of integer linear equations that have a solution}\}$$

Theorem

PROG-LIN is PTIME-complete.

Proof

The difficult part: PROG-LIN $\in$ PTIME.

Proof

The difficult part: PROG-LIN $\in$ PTIME.

- First algorithm: Khachiyan (1979)

  Not used in practice due to the high degree polynomial

Proof

The difficult part: PROG-LIN $\in$ PTIME.

- First algorithm: Khachiyan (1979)

  Not used in practice due to the high degree polynomial

- A better algorithm: Karmarkar (1984)

Proof

The difficult part: PROG-LIN $\in$ PTIME.

- First algorithm: Khachiyan (1979)

  Not used in practice due to the high degree polynomial

- A better algorithm: Karmarkar (1984)

Easy direction: PROG-LIN is PTIME-hard.

Proof

The difficult part: PROG-LIN $\in$ PTIME.

- First algorithm: Khachiyan (1979)

  Not used in practice due to the high degree polynomial

- A better algorithm: Karmarkar (1984)

Easy direction: PROG-LIN is PTIME-hard.

- We will show that HORN-SAT $\leq_L$ PROG-LIN

Proof

Let φ be a conjunction of Horn clauses.

We will construct a system of linear equations $A\vec{x} \leq \vec{b}$ such that:

φ is satisfiable iff $A\vec{x} \leq \vec{b}$ has a solution.

Each variable in φ is a variable in the equation system.

- For each variable p we include the equations:

$$\begin{aligned} -p &\leq 0 \\ p &\leq 1 \end{aligned}$$

Proof

Furthermore, they system has the following equations:

- If p is a conjunction in φ:

$$-p \leq -1$$

- If $(\neg p_1 \vee \cdots \vee \neg p_n \vee q)$ is a conjunction in φ:

$$\left(\sum_{i=1}^{n} p_i\right) - q \leq n - 1$$

- If $(\neg p_1 \vee \cdots \vee \neg p_n)$ is a conjunction in φ:

$$\left(\sum_{i=1}^{n} p_i\right) \leq n - 1$$

Small changes for NP-completeness

If we require that the solution to consist only of integers, the problem becomes NP-complete.

In this case the problem is called integer linear programming.

More PTIME-complete problems: Boolean circuits

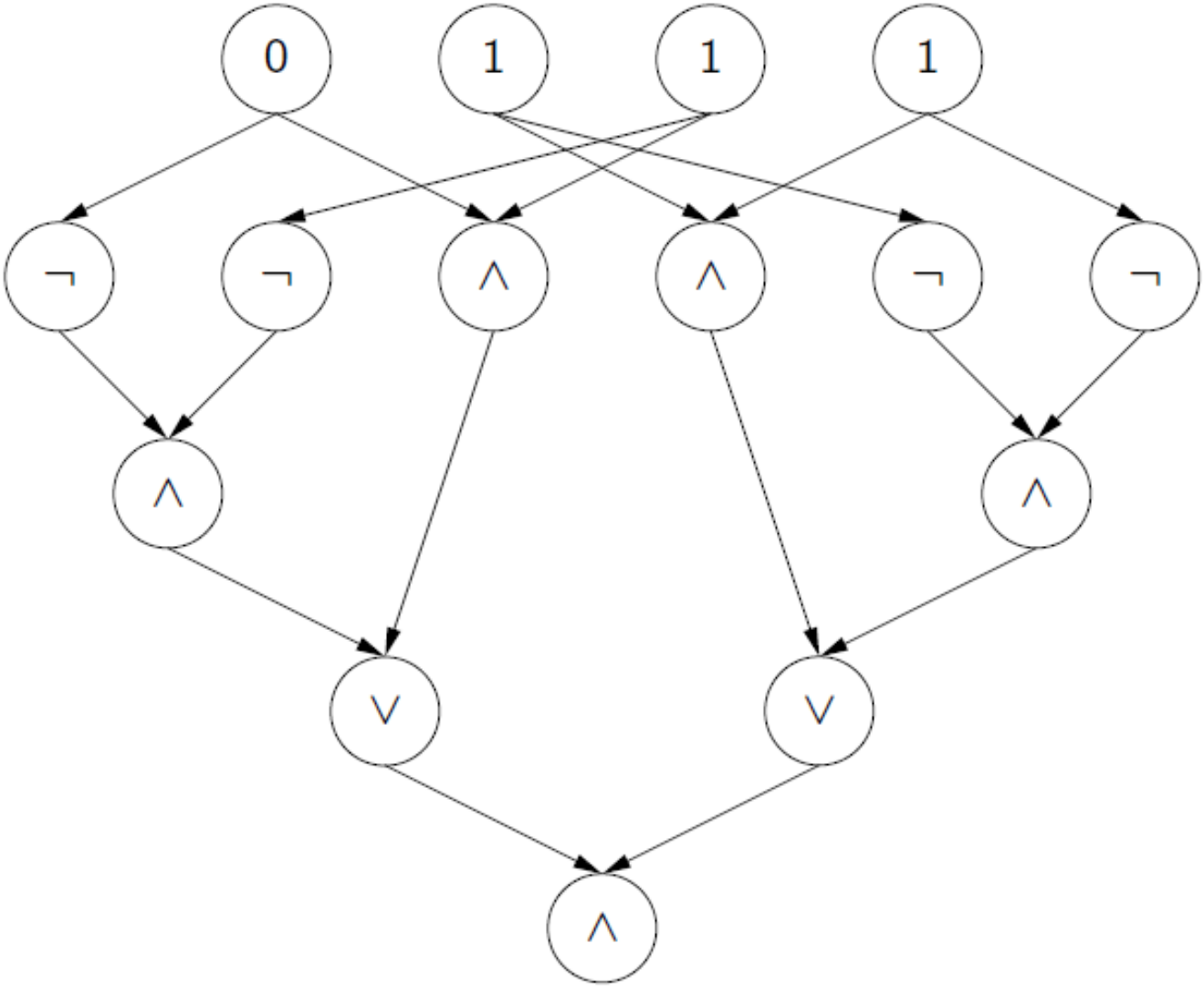
A Boolean circuit is a directed acyclic graph where:

- Every node without in-edges has the label 0 or 1
- Every node with in-edges has the label $\neg$, $\wedge$ o $\vee$
  - A node labelled $\neg$ has just one in-edge

Notation

- *Input node: No in-edges*
- *Interior node: With in-edges*
- *Output node: No out-edges*

Boolean circuits: example



Boolean circuits as functions

We can view a Boolean circuit as a function:

Boolean circuits as functions

We can view a Boolean circuit as a function:

- The input are the values of input nodes

Boolean circuits as functions

We can view a Boolean circuit as a function:

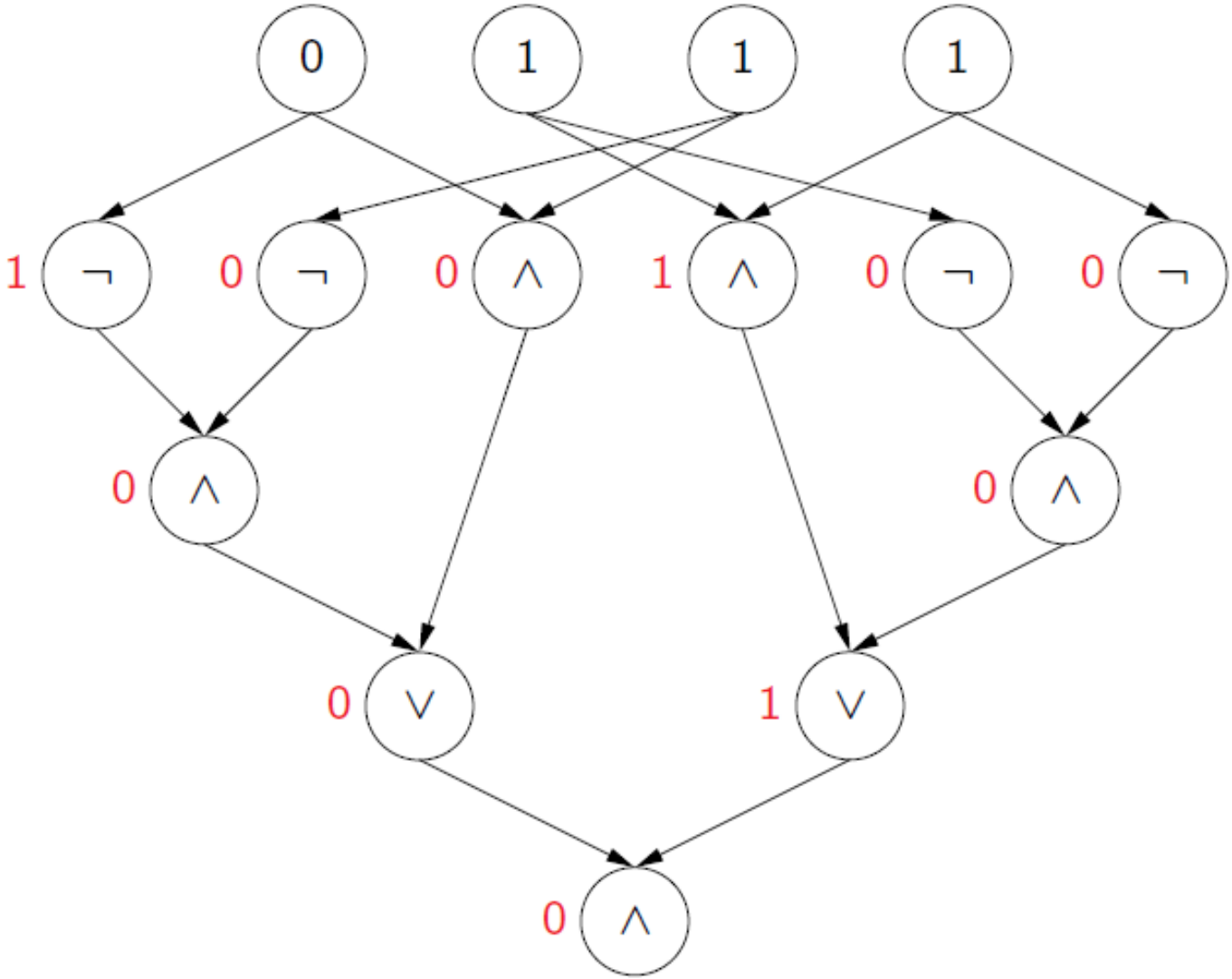
- The input are the values of input nodes
- Assign to each interior node N the result of evaluating the label of N with the values associated to nodes that have an edge entering N

Boolean circuits as functions

We can view a Boolean circuit as a function:

- The input are the values of input nodes
- Assign to each interior node N the result of evaluating the label of N with the values associated to nodes that have an edge entering N
- The result are the values of output nodes

Circuits as functions: an example



Circuits as functions

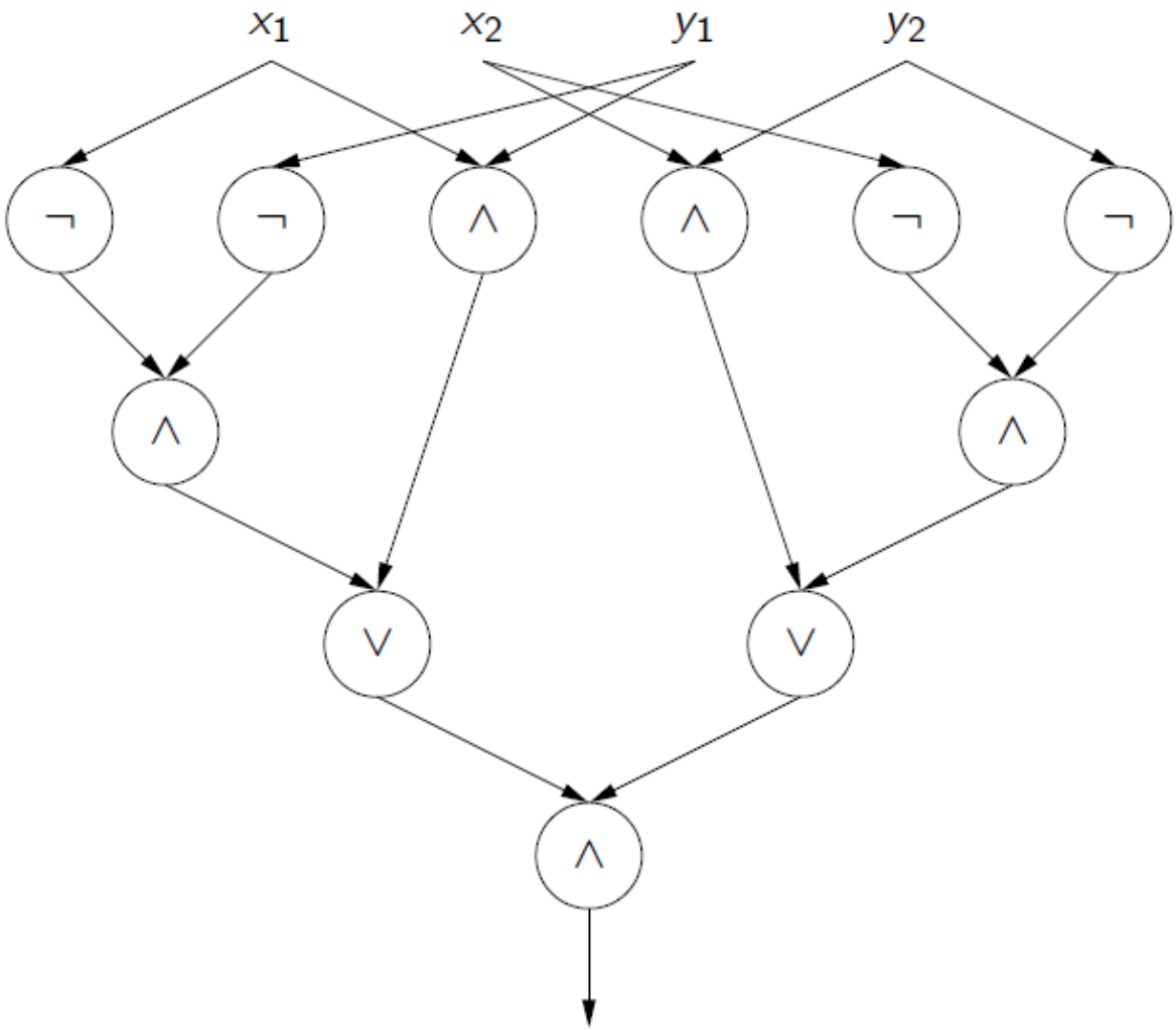
The previous circuit represents a function

$$f(x_1, x_2, y_1, y_2) = \begin{cases} 1 & x_1 = y_1 \text{ and } x_2 = y_2 \\ 0 & \text{otherwise} \end{cases}$$

To define a function associated to a circuit

- replace the values 0/1 with variables
- To indicate output we use an edge that does not enter any node

Circuits as functions



Circuit problems

Boolean circuit defines a value

Boolean circuit with variables defines a function

Theorem

For any function $f : \{0,1\}^n \to \{0,1\}^m$ there is a boolean circuit with n inputs and m outputs that computes f.

Prove this for homework (start small)

Circuit problems

Two natural problems associated to circuits:

$$\text{CIRCUIT VALUE} = \{\langle C \rangle \mid C \text{ is a Boolean circuit that evaluates to } 1\}$$

$$\text{CIRCUIT SAT} = \{\langle C \rangle \mid C \text{ is a Boolean circuit with variables and there is an assignment of variables such that } C \text{ evaluates to } 1\}$$

PTIME-completeness: circuits

Theorem

CIRCUIT VALUE is PTIME-complete.

Proof. The upper bound is easy.

For PTIME-hardness take any $L \in \text{PTIME}$ and let M be the machine deciding L in PTIME.

As in the Cook-Levin theorem we use a tableau to describe the computation of M (note that M is deterministic, so for each input there is only one tableau)

PTIME-completeness: circuits

For convenience we modify the tableau representation as follows:

- First and last column are #
- The alphabet of the tableau has the machine alphabet plus the symbols a_q, for each a in the alphabet of M and state q
- Head position is represented by a_q
- The last row always has $1_{q_{accept}}$ or $0_{q_{reject}}$ in the third position which signals that the machine accepts/rejects
- The machine always does precisely $t_M(n)$ steps for an input of length n
- Tableau accepts iff the machine does

PTIME-completeness: circuits

Tableau selfie:

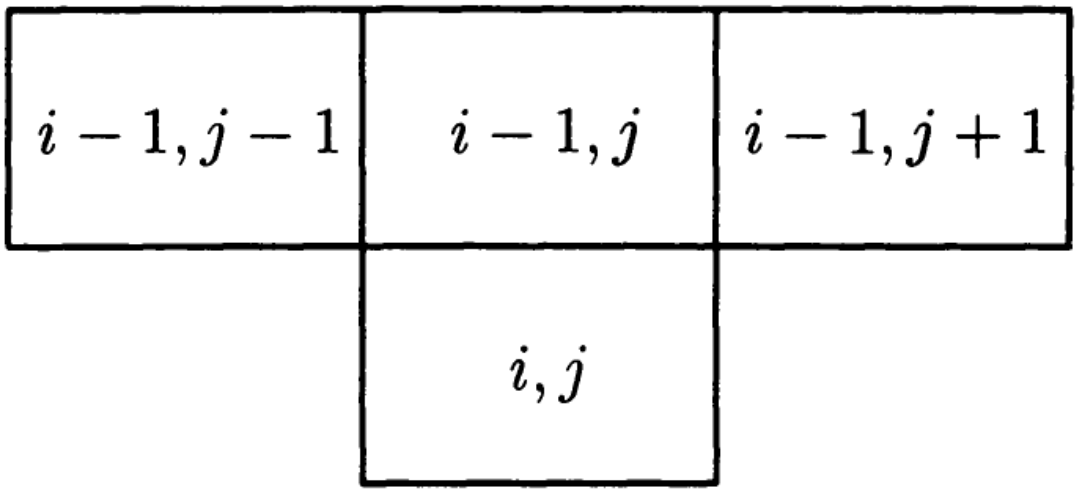
$$
\begin{array}{ccccccccc}
\# & \vdash & 0_{q_0} & 1 & 1 & 1 & \mathrm{B} & \mathrm{B} & \# \\
\# & \vdash & 0 & 1_{q_1} & 1 & 1 & \mathrm{B} & \mathrm{B} & \# \\
\# & \vdash & 0 & 1 & 1_{q_1} & 1 & \mathrm{B} & \mathrm{B} & \# \\
\# & \vdash & 0 & 1 & 1 & 1_{q_1} & \mathrm{B} & \mathrm{B} & \# \\
\# & \vdash & 0 & 1 & 1 & 1 & \mathrm{B}_{q_1} & \mathrm{B} & \# \\
\# & \vdash & 0 & 1 & 1 & 1_{q_3} & \mathrm{B} & \mathrm{B} & \# \\
\# & \vdash & 0 & 1 & 1_{q_3} & 1 & \mathrm{B} & \mathrm{B} & \# \\
\# & \vdash & 0 & 1_{q_3} & 1 & 1 & \mathrm{B} & \mathrm{B} & \# \\
\# & \vdash & 0_{q_3} & 1 & 1 & 1 & \mathrm{B} & \mathrm{B} & \# \\
\# & \vdash_{q_3} & 0 & 1 & 1 & 1 & \mathrm{B} & \mathrm{B} & \# \\
\# & \vdash & 0_{q_{reject}} & 1 & 1 & 1 & \mathrm{B} & \mathrm{B} & \#
\end{array}
$$

PTIME-completeness: circuits

For a string w we will construct a circuit $R(w)$ such that M accepts w iff $R(w)$ evaluates to 1

$T_{i,j}$ the cell in the tableau at position i, j

- The value of $T_{i,j}$ depends only on:
- The values of $T_{i-1,j-1}$, $T_{i-1,j}$ and $T_{i-1,j+1}$
- Some formalities with border cases



PTIME-completeness: circuits

Let A contain all the symbols appearing on the tableau:

- Each $a \in A$ is represented using a vector $(s_1, \ldots, s_m) \in \{0,1\}^m$
- Here $m : \lceil log_2 |A| \rceil$

Our tableau in position $T_{i,j}$ actually has entries $S_{i,j,\ell}$, with $\ell = 1 \ldots m$

Each $S_{i,j,\ell}$ depends on $3m$ binary entries:

- $S_{i-1,j-1,1}, \ldots, S_{i-1,j-1,m}$
- $S_{i-1,j,1}, \ldots, S_{i-1,j,m}$
- $S_{i-1,j+1,1}, \ldots, S_{i-1,j+1,m}$

PTIME-completeness: circuits

That is, there exist binary functions $F_1, \ldots F_m$ such that for each i, j

$$S_{i,j,\ell} = F_\ell(S_{i-1,j-1,1}, \ldots, S_{i-1,j-1,m}, S_{i-1,j,1}, \ldots, S_{i-1,j+1,m})$$

Every boolean function can be computed by a circuit:

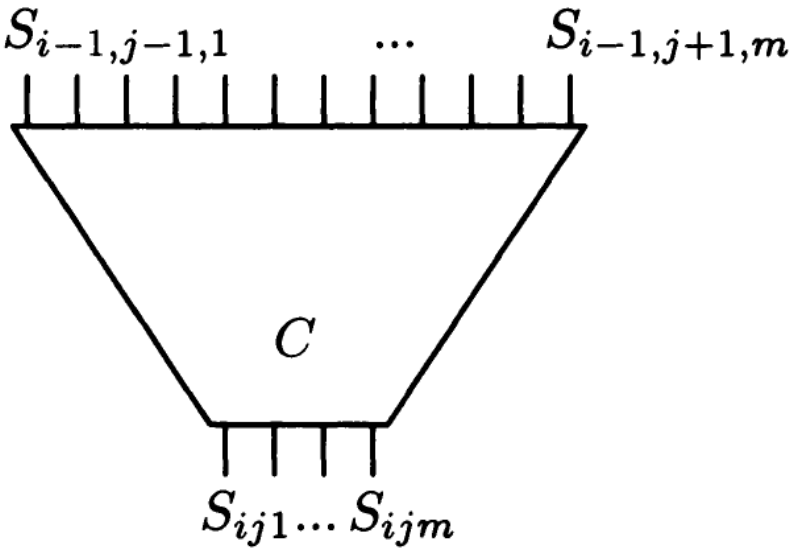
- There is C with $3m$ entries
- C computes (the binary encoding of) $T_{i,j}$ (for **any** i, j)
- When its input are encodings of $T_{i-1,j-1}$, $T_{i-1,j}$, $T_{i-1,j+1}$

C depends only on M and not on the input string

Remember valid windows from Cook-Levin

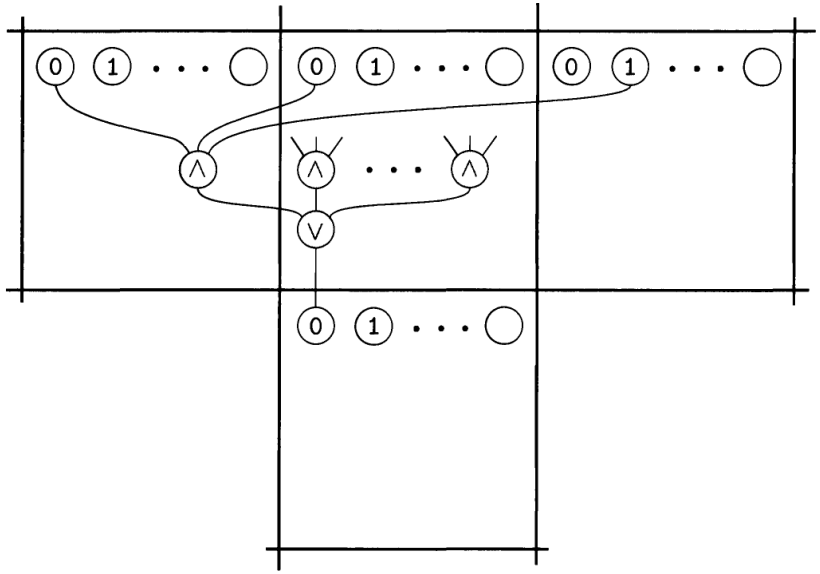
PTIME-completeness: circuits

Visually:



PTIME-completeness: circuits

Visually in a bit more detail:



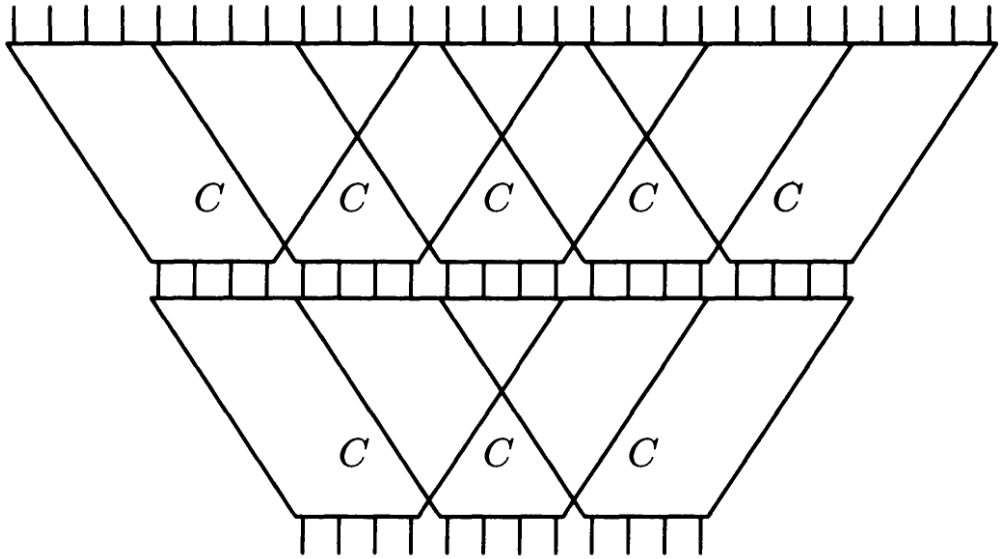
PTIME-completeness: circuits

For input word w we define the circuit $R(w)$:

- We have $t_M(|w|)^2$ copies of C
- Each copy is for one $T_{i,j}$
- The number is a bit smaller, as first row/column are not there
- $C_{i,j}$ the (i,j)-th copy of C
- The input gates of $C_{i,j}$ are the output gates of $C_{i-1,j-1}, C_{i-1,j}, C_{i-1,j+1}$
- The input gates of the circuit is the first row and first/last column
- The output gate is the first output of the circuit $C_{t_M(|w|),2}$ (recall the initial assumption)

PTIME-completeness: circuits

The whole circuit:



PTIME-completeness: circuits

To finish:

- Show that M accepts W iff $R(w)$ is true
- Show that $R(w)$ is computable in LOGSPACE
- For latter recall that C is fixed
- And recall that to construct the input gates you only need to count up to $t_M(|w|)$

More cool stuff with circuits

Exercise

Modify the previous proof to show that CIRCUIT SAT is NP-complete.