CS 445
Analysis of algorithms 2
P, NP, etc.
Éric Schost
eschost@uwo.ca
Problems
What problems do we consider?

The classical approach consider decision problems.

Informal definition.

- A decision problem is a problem to which the answer is yes or no.

Examples.

- is this graph a tree?
- is this graph colorable with 3 colors?
- does this system of linear constraint admit an integer solution?
- etc
Usual notation

Problems are usually written **USING THIS FONT**.

Then, to indicate that an instance \( x \) solves the problem \( \text{PROBLEM} \), I will write \( x \in \text{PROBLEM} \).

Examples

- a graph \( G \) is 3 colorable \( \iff G \in 3\text{-COLOR} \).
- a set \( S \) of linear constraints has an integer solution \( \iff S \in \text{INTEGERLP} \).
- a set \( S \) of linear constraints has an integer solution with 0/1 coordinates \( \iff S \in 0/1\text{-INTEGERLP} \).
- a boolean formula \( B \) is satisfiable \( \iff B \in \text{SAT} \).
Optimization problems

Optimization problems (like the ones we saw) are not decision problems.

- find the maximal flow
- find the maximum spanning true
- optimize a linear function . . .

When entering the P vs. NP world, one usually considers the decision versions of optimization problems:

- given $K$, is there a flow of value $\geq K$?
- given $K$, is there a spanning tree of weight $\geq K$?
- etc.
Encoding of a problem

When dealing with complexity notions, we need to make things precise: the encoding of your problem matters.

Example 1: binary encoding of numbers.

- This is the natural way to write a number, by a sequence of 0’s and 1’s.
- The size of the encoding of $a$ is $\simeq \log(a)$.

Example 2: unary encoding of numbers.

- This is a not-so-natural way to write a number: you represent $n$ by $n$ 0’s and a 1.

\[
1 \rightarrow 01 \quad 10 \rightarrow 00000000001
\]

- The size of the encoding of $a$ is $\simeq a$.

In “real life” of algorithm design, nobody really cares too much about that, as usually, no real confusion should arise. But still, we want to play it safe.
Encoding of a problem

When dealing with complexity notions, we need to make things precise: the encoding of your problem matters.

Problem: factoring an integer $a$.

- In the **binary** encoding, the target would be a complexity polynomial in $\log(a)$.
  
  Nobody knows how to do that.

- In the **unary** encoding, it’s easy to get a polynomial time algorithm.
  
  – just try all possible factors: there are no more than $a$ factors;
  
  – a trial division takes polynomial time.

**Summary:** when it comes to estimate complexities, you should completely specify the encoding. Nobody *does*, but everybody *should*. 
Polynomial time

Def.

• A decision problem \textsc{Prob} (given in a specified encoding) can be solved in polynomial time if

  - there exists an algorithm \( A \) that takes as input an instance \( x \) and outputs “yes” or “no” in polynomial time

    \[
    O(\text{size}(x)^k).
    \]

  - for any instance \( x, x \in \text{Prob} \iff A(x) = \text{“yes”}.

Def.

• \( \text{P} \) is the set of all problems that can be solved in polynomial time.

Examples.

• Max flow, min cut, spanning trees, etc are in \( \text{P} \).
What is an algorithm?

A subtelty is in the specification of what an algorithm is.

Several models can be used.

- Up to now, we have used some sort of pseudo-code.
  The closest model would be the Random Access Machine (RAM).

- Classical models are Turing machines.
  - the input is written (in binary) on a tape
  - one or several extra tapes are available (workspace)
  - a cell reads / writes the information on the tapes, and shifts the tapes by one unit at a time.
What is an algorithm?

To some extent, the notion of “polynomial time” does not really depend on the computational model.

If a problem can be solved in polynomial time with one-tape Turing machines, it can be solved in polynomial time with

- multi-tape Turing machines,
- RAM machines,
- others . . .

and conversely. However, the exponents $k$ may vary.

So polynomial time is good because you don’t really have to worry about the specifics of your model, and still say something meaningful.
Polynomial time certification

Certifying answers could be easier than finding them.

Def.

- Consider a decision problem PROB. It can be certified in polynomial time if
  - there exists an algorithm $B$ that takes as input an instance $x$ and an extra input $y$ (a certificate) and outputs “yes” or “no” in polynomial time;
  - for any instance $x$, $x \in$ PROB if and only if there exists $y$ of size $O(\text{size}(x)^k)$, such that $B(x, y) =$“yes”.

Def.

- NP is the set of all problems that can be certified in polynomial time.

Why NP? This stands for Nondeterministic Polynomial.
Prop. $P \subset \text{NP}$. What does this mean? That if you can solve a problem in polynomial time, you can certify it in polynomial time.

Proof.

• Suppose that $A(x)$ is an algorithm that solve a problem PROB in polynomial time.

• Then we can design an algorithm $B(x, y)$ that just outputs $A(x)$.

• $B$ runs in polynomial time.

• Then $x \in \text{PROB}$ if and only if there exists $y$ of size 1 (for instance) such that $B(x, y) = \text{“yes”}$. 
The big question

Do we have $P=NP$ or $P\neq NP$?

- Tons of hard problems are in $NP$ (examples upcoming), for which we have no clue for a polynomial-time algorithm.

- So reasonable people believe $P\neq NP$.

- But nobody can prove it.
  
  Better, there are strong restrictions on how a proof of $P\neq NP$ should be like.

- Make $1000000$ by solving the problem.
Some examples

Graph coloring: 3-COLOR.

- Instance $x$: a graph.
- Problem: is the graph $x$ 3-colorable?
- Certificate $y$: an assignment of one color per vertex.
- Algorithm $B(x, y)$: test if this is a 3-coloring.
- NP? yes! $B$ runs in polynomial time, and if $x$ is 3-colorable, there exists a certificate of size $\leq \text{size}(x)$.
Some examples

0/1 Linear programming: 0/1-\textsc{IntegerLP}.

- Instance $x$: a set of constraints in $n$ variables.
- Problem: are there 0/1 values of the variables that satisfy $x$?
- Certificate $y$: a $n$-uple of 0, 1.
- Algorithm $B(x, y)$: test if this satisfies the constraints.
- \textbf{NP}? yes! $B$ runs in polynomial time, and if $x$ is satisfiable, there exists a certificate of size $\leq \text{size}(x)$. 
Some examples

Circuit satisfiability: \textsc{CircuitSat}.

- Instance $x$: a circuit (directed graph, no cycle).
  - The inputs are labelled by boolean variables $v_1, \ldots, v_n$.
  - The internal vertices are labelled by AND/OR/NOT.
  - There is a marked vertex $v$.

Example.
Some examples

Circuit satisfiability: CircuitSAT.

- Instance $x$: a circuit (directed graph, no cycle).
  - The inputs are labelled by boolean variables $v_1, \ldots, v_n$.
  - The internal vertices are labelled by AND/OR/NOT.
  - There is a marked vertex $v$.

- Problem: is there a choice of boolean values $v_i$ such that $v$ takes the value 1?

- Certificate $y$: a $n$-uple of 0, 1.

- Algorithm $B(x, y)$: evaluate $x$ at $y$ and test the value at $v$.

- NP? yes! $B$ runs in polynomial time, and if $x$ is satisfiable, there exists a certificate of size $\leq \text{size}(x)$.
Boolean formula satisfiability: SAT.

- Instance $x$: a boolean formula (with AND, OR, NOT) in $n$ variables.
- Problem: is there a choice of the variables that makes it true?
- Certificate $y$: a $n$-uple of 0, 1.
- Algorithm $B(x, y)$: test if $x(y)$ is true.
- \textbf{NP}? yes! $B$ runs in polynomial time, and if $x$ is satisfiable, there exists a certificate of size $\leq \text{size}(x)$. 
Some examples

3-terms conjonctive formula satisfiability: 3SAT.

- Instance $x$: a boolean formula in $n$ variables $x_1, \ldots, x_n$ of the form

$$
(y_{1,1} \text{ or } y_{1,2} \text{ or } y_{1,3}) \text{ and } \cdots \text{ and } (y_{k,1} \text{ or } y_{k,2} \text{ or } y_{k,3})
$$

with $y_{i,j}$ of the form $x_\ell$ or not($x_\ell$).

Example

$$(x_1 \text{ or } x_2 \text{ or not}(x_1)) \text{ and } (x_2 \text{ or } x_3 \text{ or not}(x_1)) \text{ and } (\text{not}(x_1) \text{ or } x_3 \text{ or not}(x_2))$$.
3-terms conjunctive formula satisfiability: 3SAT.

- Instance $x$: a boolean formula in $n$ variables $x_1, \ldots, x_n$ of the form
  \[(y_{1,1} \text{ or } y_{1,2} \text{ or } y_{1,3}) \text{ and } \cdots \text{ and } (y_{k,1} \text{ or } y_{k,2} \text{ or } y_{k,3})\]
  with $y_{i,j}$ of the form $x_\ell$ or not($x_\ell$).

- Problem: is there a choice of the variables that makes it true?

- Certificate $y$: a $n$-uple of 0, 1.

- Algorithm $B(x, y)$: test if $x(y)$ is true.

- **NP?** yes! $B$ runs in polynomial time, and if $x$ is satisfiable, there exists a certificate of size $\leq \text{size}(x)$. 
**Exponential time**

Def.

- **EXP** is the set of all problems that can be solved in exponential time \( O(2^{\text{size}(x)^k}) \) for some \( k \).

Prop.

- **NP \subset EXP**
  
  Problems in **NP** cannot be extraordinarily bad.

Idea of the proof: try all possible certificates.

- For a given \( x \), we look for a certificate of size \( \text{size}(x)^k \), for some constant \( k \).
- Supposing we work with binary symbols. Then, there are \( 2^{\text{size}(x)^k} \) certificates
- Each of them takes polynomial time.
Reduction
Reduction formulizes and extends the idea that you can use subroutines to solve new problems.

**Key idea:** if you can solve a problem PROB2 in polynomial time, you can use it to solve PROB1 in polynomial time.

**Def.**

- A problem PROB1 can be reduced to a problem PROB2 if
  - there exists an algorithm $C(x)$ that runs in polynomial time,
  - such that $x \in \text{PROB1}$ if and only if $C(x) \in \text{PROB2}$.

**Notation:** PROB1 $\leq$ PROB2.
Example

Prop. \textsc{CircuitSat} \leq \textsc{Sat}.

What do we have to do:

- We want to transform an instance of \textsc{CircuitSat} into an instance of \textsc{Sat}.
- The transformation should take polynomial time.

becomes

\[(y_1 = (x_1 \text{ and } x_2)) \text{ and } (y_2 = (x_2 \text{ and } x_3)) \text{ and } (y_3 = (y_2 \text{ or } y_3)) \text{ and } (y_3)\]

where \(y = z\) stands for \((y \text{ or not}(z)) \text{ and } (z \text{ or not}(y))\)
Example

Prop. SAT ≤ 3SAT.

What do we have to do:

- We want to transform an instance of SAT into an instance of 3SAT.
- The transformation should take polynomial time.

Step 1. A formula can easily be put in disjunctive form

\[ Y = (x_1 \text{ and } x_2 \text{ and } (x_3 \text{ or } (x_4))) \text{ or not}(x_1 \text{ or } x_3) \]

can become

\[
\begin{align*}
y_1 &= x_1 \text{ and } x_2 \\
&\text{and } y_2 = x_3 \text{ or } x_4 \\
&\text{and } y_3 = y_1 \text{ and } y_2 \\
&\text{and } y_4 = x_1 \text{ or } x_3 \\
&\text{and } y_5 = \text{not}(y_4) \\
&\text{and } Y = y_3 \text{ or } y_5
\end{align*}
\]
Example

Prop. SAT \leq 3SAT.

What do we have to do:

- We want to transform an instance of SAT into an instance of 3SAT.
- The transformation should take polynomial time.

Step 2. Any formula of the form \( y = x_1 \) and \( x_2, \ldots \) can be turned into a 3-term “or” form.

\[
\begin{align*}
y = x_1 \text{ and } x_2 & \iff (y \text{ or } \neg(x_1 \text{ and } x_2)) \text{ and } (\neg(y) \text{ or } (x_1 \text{ and } x_2)) \\
& \iff (y \text{ or } \neg(x_1 \text{ and } x_2)) \text{ and } (\neg(y) \text{ or } x_1) \text{ and } (\neg(y) \text{ or } x_2) \\
& \text{ and } \\
(\neg(y) \text{ or } x_1) & \iff (\neg(y) \text{ or } x_1 \text{ or } x_1) \\
(\neg(y) \text{ or } x_2) & \iff (\neg(y) \text{ or } x_2 \text{ or } x_2)
\end{align*}
\]

Etc ... The transformation takes polynomial time.
NP completeness

Def.

- A problem \( \text{Prob} \) is **NP-hard** if for every problem \( \text{Prob}' \) in **NP**, we have

  \[
  \text{Prob}' \leq \text{Prob}
  \]

  If you can solve \( \text{Prob} \) in polynomial time, you can solve all problems in **NP** in polynomial time.

- A problem \( \text{Prob} \) is **NP-complete** if it is NP-hard and in **NP**.

Prop.

- If \( \text{Prob}_1 \leq \text{Prob}_2 \) and \( \text{Prob}_1 \) is NP-hard, then \( \text{Prob}_2 \) is NP-hard too.
A hard result

**Theorem.** \textsc{CircuitSAT} is \textbf{NP}-complete.

**Idea of the proof.**

We have to show that any problem \textsc{Prob} in \textbf{NP} can be reduced to \textsc{CircuitSAT}. Since \textsc{Prob} is \textbf{NP}, it has a certifying algorithm $B(x, y)$.

To do the reduction, given $x$, we need to find a way to test if some $y$ gives $B(x, y) = \text{“yes”}$.

We transform the algorithm $y \mapsto B(x, y)$ into a huge (but polynomial-size) circuit, having one vertex for each memory cell and for each time step.

**Theorem.** \textsc{3SAT} and \textsc{SAT} are \textbf{NP}-complete.

**Proof:** \textsc{CircuitSAT} $\leq \textsc{SAT} \leq \textsc{3SAT}$. 
Coverings and independents
Independent sets

An independent set in a non-oriented graph is a subset $S$ of the vertices such that no vertices in $S$ are connected.
Independent sets

An independent set in a non-oriented graph is a subset $S$ of the vertices such that no vertices in $S$ are connected.

The vertices in red are an independent set.
Independent sets

An independent set in a non-oriented graph is a subset $S$ of the vertices such that no vertices in $S$ are connected.

The vertices in red are not an independent set.
Vertex covers

A vertex cover in a non-oriented graph is a subset $S$ of the vertices such that each edge has an extremity in $S$. 
Vertex covers

A vertex cover in a non-oriented graph is a subset $S$ of the vertices such that each edge has an extremity in $S$.

The vertices in blue form a vertex cover.
Vertex covers

A vertex cover in a non-oriented graph is a subset $S$ of the vertices such that each edge has an extremity in $S$.

The vertices in blue do not form a vertex cover.
NP-completeness

Prop.

- $S$ is a vertex cover if and only if $\text{vertices}(G) - S$ is an independent set.

$S$ is a vertex cover if and only if each edge has at least one extremity in $S$ if and only if each edge has at most one extremity in $\text{vertices}(G) - S$.

Decisions problems

- Given $k$, is there an independent set of size $\geq k$?
- Given $k$, is there a vertex cover of size $\leq k$?

Prop.

- Both problems are $\textbf{NP}$-complete.

The fact that their are $\textbf{NP}$ is easy!
Reduction from 3-SAT

Prop. $3\text{SAT} \leq \text{Max-Independent}$. 

We need to transform an instance of $3\text{SAT}$, with $k$ clauses, into an independent set problem. Consider for instance the formula with $k = 3$

$$(x_1 \text{ or } x_2 \text{ or } \neg(x_1)) \text{ and } (x_2 \text{ or } x_3 \text{ or } \neg(x_1)) \text{ and } (\neg(x_1) \text{ or } x_3 \text{ or } \neg(x_2)).$$

Build a graph that has $3 \times \text{(number of clauses)}$ vertices:

For the moment, finding an independent set of size $\geq k$ amounts to pick one vertex in each group ($= \text{find which term in the clause will be true}$).
Reduction from 3-SAT

We add extra edges that restrict the independents: we connect all instances of \((x_i, \text{not}(x_i))\) across groups.

It is easy to check that:

- the formula is satisfiable if and only if there is an independent set of size \(\geq k\);
- the reduction takes polynomial time.
A more general covering problem

Def.

- Given a finite set $S$ and some subsets $S_1, \ldots, S_m$ that cover $S$ ($S = S_1 \cup \cdots \cup S_m$), a set cover of $S$ is the choice of $S_{i_1}, \ldots, S_{i_r}$ that still cover $S$.

Vertex cover is a particular case of set cover ($S$ is the set of edges; the $S_i$ are indexed by the vertices).

We showed that \textsc{VertexCover} is \textbf{NP}-complete, so \textsc{SetCover} is \textbf{NP}-complete too (easy to check that it is in \textbf{NP}).
2SAT
2SAT

2-terms conjonctive formula satisfiability: 2SAT.

- Instance $x$: a boolean formula in $n$ variables $x_1, \ldots, x_n$ of the form

$$ (y_{1,1} \text{ or } y_{1,2}) \text{ and } \cdots \text{ and } (y_{k,1} \text{ or } y_{k,2}) $$

with $y_{i,j}$ of the form $x_\ell$ or $\text{not}(x_\ell)$.

Example.

$$ (x_1 \text{ or } x_2) \text{ and } (x_2 \text{ or } \text{not}(x_1)) \text{ and } (x_3 \text{ or } \text{not}(x_2)). $$
2SAT

2-terms conjonctive formula satisfiability: 2SAT.

• Instance $x$: a boolean formula in $n$ variables $x_1, \ldots, x_n$ of the form

  $$(y_{1,1} \text{ or } y_{1,2}) \text{ and } \cdots \text{ and } (y_{k,1} \text{ or } y_{k,2})$$

  with $y_{i,j}$ of the form $x_\ell$ or not$(x_\ell)$.

• Problem: is there a choice of the variables that makes it true?

• Certificate $y$: a $n$-uple of 0, 1.

• Algorithm $B(x, y)$: test if $x(y)$ is true.

• NP? yes!

• Better: 2SAT can be solved in polynomial time!
MAX2SAT

Optimal 2-terms conjonctive formula satisfiability: MAX2SAT.

• Instance $x$: a boolean formula in $n$ variables $x_1, \ldots, x_n$ of the form

$$(y_{1,1} \text{ or } y_{1,2}) \text{ and } \cdots \text{ and } (y_{k,1} \text{ or } y_{k,2})$$

with $y_{i,j}$ of the form $x_\ell$ or $\text{not}(x_\ell)$.

• Problem: given an integer $k$, is there an assignment of the variables that satisfies at least $k$ clauses?

• Certificate $y$: a $n$-uple of 0, 1.

• Algorithm $B(x, y)$: test how many clauses are true.

• NP? yes!

• This one is NP-complete.
Polynomial time algorithm for 2SAT

Idea: \((x_i \text{ or } x_j)\) is equivalent to

\[
\neg(x_i) \Rightarrow x_j \quad \text{and to} \quad \neg(x_j) \Rightarrow x_i.
\]

- We can chain these implications to eventually find out a satisfiable solution.
- So we put them in a graph.

Example.

\((x_1 \text{ or } x_2)\) and \((x_2 \text{ or } \neg(x_1))\) and \((x_3 \text{ or } \neg(x_2))\).
How to use the graph

Suppose that we have an assignment of the variables that makes the formula true.

- Each vertex in the graph is assigned either true or false.
- If a vertex $v$ is true and there is an edge $v \rightarrow w$, then $w$ is true too.

So if there is a cycle $x_i \rightarrow \text{not}(x_i) \rightarrow x_i$, for some $i$, both $x_i$ and not($x_i$) should be true, which is impossible. The formula cannot be satisfied.
How to use the graph

To exploit this idea, we find a topological order on the graph.

**Def.** A topological order is a numbering $f$ of the vertices such that

- $f(v) = f(w)$ if and only if there is a loop $v \to w \to v$.
- if there is a path $v \to w$ but no path $w \to v$, then $f(v) < f(w)$.

How to compute such an $f$? Later.

How to use it?

- There is a cycle $x_i \to \text{not}(x_i) \to x_i$ if and only if $f(x_i) = f(\text{not}(x_i))$.

So knowing $f$, we can easily test this property. If this is not the case, we will construct an assignment that makes the formula true.
Preparing an assignment

Suppose that for each variable \( i \), \( f(x_i) \neq f(\text{not}(x_i)) \).

- if \( f(x_i) < f(\text{not}(x_i)) \), set \( x_i = \text{false} \).
- if \( f(\text{not}(x_i)) < f(x_i) \), set \( x_i = \text{true} \).

Prop. This assignment makes the formula true.

By contradiction: suppose that the formula is false.

- So, there is one clause which is false. Say this clause is \((x_i \text{ or } x_j)\), so the assignment has \( x_i = x_j = \text{false} \).
- So \( f(x_i) < f(\text{not}(x_i)) \) and \( f(x_j) < f(\text{not}(x_j)) \)
- There are edges \( \text{not}(x_i) \rightarrow x_j \) and \( \text{not}(x_j) \rightarrow x_i \), so
  \[
  f(\text{not}(x_i)) \leq f(x_j) \quad \text{and} \quad f(\text{not}(x_j)) \leq f(x_i).
  \]
- We deduce \( f(x_i) < f(x_j) \) and \( f(x_j) < f(x_i) \), a contradiction.
Reducing 3SAT to MAX2SAT

Preliminaries. Consider a clause like \((x \lor y \lor z)\).

We introduce a new variable \(t\), and consider the clauses

\[
\begin{align*}
&x, \; y, \; z, \; t, \; \neg(x) \lor \neg(y), \; \neg(y) \lor \neg(z), \; \neg(z) \lor \neg(x), \\
&\quad \quad x \lor \neg(t), \; y \lor \neg(t), \; z \lor \neg(t).
\end{align*}
\]

Then:

- you cannot satisfy more than 7 of these new clauses;
- if you satisfy 7 of them, \((x \lor y \lor z)\) is true;
- if \((x \lor y \lor z)\), you can satisfy 7 of them.
Reducing 3SAT to MAX2SAT

Reduction. Given a family of clauses $k$ that form a 3SAT problem, we introduce

- one new variable $t_i$ per clause
- and the 10 clauses as seen before.

Then:

- you cannot satisfy more than $7k$ of these new clauses;
- you can satisfy $7k$ of them simultaneously if and only if you can satisfy all $k$ input clauses simultaneously.

Conclusion: MAX2SAT is \textbf{NP}-complete.