# NP-Problems

Let's start looking at two graph problems:

Euler walks: An Euler walk is a walk trough a graph which contains each edge exactly once.

EULER WALKS

Input: A connected, undirected graph G.
Goal: Is there an Euler walk in G or not?

Euler presented a simple solution: There is an Euler walk if and only if every node has even degree. This problem can be solved in polynomial time.

Hamiltonian cycle: A hamiltonian cycle is a cycle containing ever node exactly once.

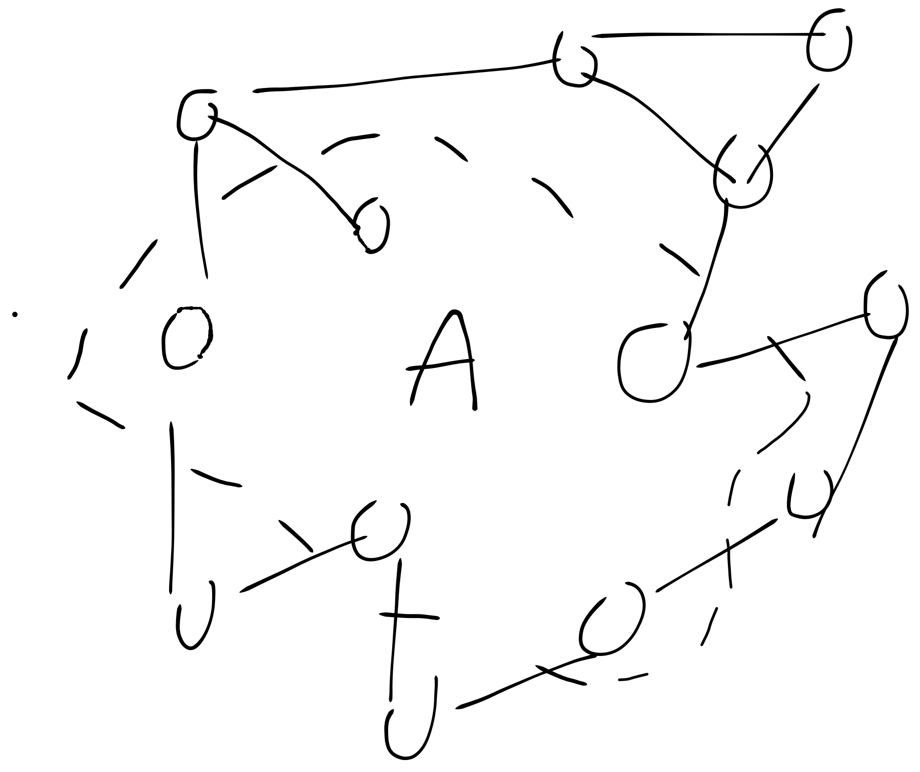
HAMILTONIAN CYCLE

Input: A connected, undirected graph G.
Goal: Is there a hamiltonian cycle in G or not?

This problem has no known efficient solution.

Another difficult graph problem is Independent Set.

Independent set: If G is a graph and $A \subseteq V$ then A is independent if and only if there are no edges going between nodes in A.



INDEPENDENT SET

Input: A graph G. An integer K.
Goal: Is there an independent set A of size K in G?

It might seem reasonable to ask for the largest independent set in G. This, however, is not quite the same problem. There are, in fact, three groups of problems to consider.

## Three types of problems

Problems can be classified according to what type of goal/output we want.

### Decision problems

In a decision problem we just want an answer yes/no.

Ex: Given a graph G and nodes s,t, is there a shortest path of length L between s and t?
Answer: Yes/ No.

### Optimization problems

In these problems the answer is an integer that measures the size of an optimal solution.

Ex: Given a graph G and nodes s,t, what is the length of a shortest path between s and t?
Answer: A number.

### Construction problems

In these problems we want to actually construct a solution.

Ex: Given a graph G and nodes s,t, find a shortest path between s and t.
Answer: Description of a path.

For technical reasons we will be most interested in decision problems.

Corresponding optimization problem:

MAX-IS

Input: A graph $G$.
Goal: What is the size of a maximal IS in $G$?

Corresponding construction problem:

CONSTRUCT-MAX-IS

Input: A graph $G$.
Goal: Find a largest IS in $G$.

Assume that there is a solution algorithm $A(G, k)$ such that $A(G, k) =$ Yes if and only if $G$ has an IS of size $k$.

MAX-IS has a solution algorithm $B(G)$:

```
(1)  for $k \leftarrow$ to $n$
(2)     if $A(G, k) =$ Yes
(3)        $m \leftarrow k$
(4)  return $m$
```

CONSTRUCT-MAX-IS has a solution algorithm $C(G)$:

```
(1)  $m \leftarrow B(G)$
(2)  $S \leftarrow V$
(3)  foreach $v \in V$
(4)     if $B(G(S - \{v\})) = m$
(5)        $S \leftarrow S - \{v\}$
(6)     return $S$
```

## Why is IS "hard"?

We can see that IS can be solved by checking all subsets of V(G) of size k. If k is a small number, this is possible. But if k is large, like n/2, this will be an exponential time algorithm. So this is no good way of solving IS in the general case.

In order to get a polynomial time algorithm we would have to "zoom in" on the possible candidates for independent sets. The problem seems to be that the graph structure is so general that it is hard to use any partial information to tell if there is an IS in the graph or not. You have to "check everything". But no-one has been able to prove that ut really is so.

## Another problem: Vertex Cover

Given and undirected graph G we say that a vertex cover is a set S of nodes such that each edge in the graph contains at least on node in S. The set V(G) is if course a vertex cover, but we are interested in small vertex cover. The problem VERTEX COVER takes G (graph) and K (integer) as input. The problem is then to decide if G has a vertex cover of size K or not. This is also known to be a hard problem. There is a close connection between VERTEX COVER and INDEPENDENT SET.

Observation: A set S is a vertex cover in G if and only if V(G)-S is an independent set in G.

This shows that VERTEX COVER with input (G, K) has yes-answer if and only if INDEPENDENT SET with input (G, |V(G)| - K) has yes-answer.

## Reductions (informally)

This was a first example of a technique called reductions. We could "translate" VERTEX COVER to INDEPENDENT SET. This means that if we have an efficient algorithm for solving INDEPENDENT SET we would also have an efficient method for solving VERTEX COVER. (And vice versa.) This somehow means that the two problems are "equally hard".

Long experience has shown something extraordinary: There is a large class of hard problems that seems to comprise almost all known hard problems, and this class is such that

All problems in this class are equally hard!

Furthermore it can be shown that if any of these problems could be solved efficiently, then (almost) all problems can be solved efficiently!

This class is the class of NP-Complete problems.

The natural guess is that none of these problems can be solved efficiently ( or ... ?)

We will look at one more example of a reduction:

Reduction from SUBSET SUM to PARTITIONING

Given an instance $a_1, a_2, ..., a_n$ and $M$ of SUBSET SUM we create the following instance of PARTITIONING:

Set $S = \sum a_i$. Define the input to partitioning as $\{a_1, a_2, ..., a_n, S - 2M\}$ if $M \leqslant S/2$ and $\{a_1, a_2, ..., a_n, 2M - S\}$ if $M > S/2$.

The we can show that there is a solution to this instance of PARTITIONING if and only if there is a subset sum $M$.

## Some other hard problems

(No known polynomial time algorithms solving the problem.)

### GRAPH COLORING

Input: A graph G. An integer K.
Goal: Is there a coloring of G with K colors?

A graph coloring is a coloring of the nodes so that no adjacent nodes have equal colors.

### SET COVERING

Input: A family F of subsets of a set V. An integer K.
Goal: Is there a set of K subsets taken from F such that their union is V?

### SUBSET SUM

Input: A set A of integers. An integer M.
Goal: Is there a subset of A with sum M?

## Traveling Salesman Problem:

A traveling salesman want to visit all cities in a country and then return to his home town. In order to save costs he wants to do this as economically as possible.

Traveling Salesman Problem (TSP):
**Given a graph $G = (V, E)$ with edge weights, is there a walk of length at most $L$ that visits all nodes exactly once and then returns to the start node?**

It can be seen that this problem is related to HAMILTONIAN CYCLE

**The Knapsack Problem:**

A tourist want to pack her knapsack but she doesn't want to carry more than $W$ kg. There are lot of things she want to bring along and they all have a known weights and utilities:

| Thing | Weight | Utility |
|---|---|---|
| Tent | 10 | 100 |
| Sleeping bag | 7 | 80 |
| Pillow | 0.5 | 10 |
| Extra sweater | 1 | 25 |
| Toothbrush | 0.01 | 5 |
| Book | 0.1 | 2 |
| etc | | |

**Is it possible to chose a set of things with combined weight at most $W$ kg and combined utility at least $U$?**

## The class NP

There is a common pattern in all these problems. In all of them we are looking for something, call it C. This something is supposed to fulfill some condition F(C). Informally this means that C is a solution to the problem. In cases the following is true:

1. C is not a complicated structure.

2. Given C, it is easy to tell if C fulfills F(C).

The condition F is: being an independent set of size K, being a vertex cover of size K and so on. The structure C is called a certificate. We can informally define NP problems as problems such that the answer to the problem is yes if and only if there exists a certificate.

> You are looking for something. It is hard to find, but when you have found it you can easily see that it is what you have been looking for.

We will now give a more formal definition. In the previous lecture we gave another definition of NP problems. We will see later that the definitions are equivalent.

## Formal definition of P

A *formal language* $L$ is a set of strings.

Example:

{“abc”, “qwerty”, “xyzzy”}
{binary strings of odd lenght}
{binary strings that represents prime numbers }
{syntactically correct C-programs}

A language can be describe in different ways:

- An enumeration of the strings in the language.
- A set of rules defining the language.
- An algorithm which recognize the strings in the language.

To every decision problem there is a corresponding language:
The language of all yes-instances.

We say that the algorithm $A$ decides $L$ if

$$A(x) = \text{Yes if } x \in L,$$
$$A(x) = \text{No if } x \notin L.$$

$A$ runs in *polynomial time* if $A(x)$ runs in time $O(|x|^k)$ for all $x$ and some integer $k$.

$P = \{L : \exists A$ that decides $L$ i polynomial time$\}$

## A formal definition of NP

$A$ *verifies* the instance $x$ of the problem $L$ if there is a certificate $y$ such that $|y| \in O(|x|^s)$ and

$$A(x,y) = \text{Yes} \quad \Leftrightarrow \quad x \in L$$

This means that $A$ decides the language
$L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^* : A(x,y) = \text{Ja}\}$

$NP = \{L : \exists A$ that verifies $L$ in polynomial time$\}$

$P \subseteq$ NP since all problem that can be decided in polynomial time also can be verified in polynomial time.

Given that it to exist hard problems, what should we do about it?

Two different approaches:

1. We could try to solve them. ( Efficiently.)

2. We could try to understand why they are hard.

( Obs: There are exponential time algorithms for solving the problems. )

The first approach has been unsuccessful. The second approach has had some success even if the success is of an unexpected kind.

The second approach has led to the theory of NP-Complete problems. We will describe this theory in this and the next lecture. The theory starts with two insights:

1. The recognition of the problem SAT as an especially important hard problem.

2. The extreme usefulness of the concept of reductions between problems.

**Satisfiability (SAT):**

Let's say that we describe a system with a Propositional Logic formula. We want to find certain situations that correspond to this formula being true. We want to know of there are values for the variables making the formula true, i.e. the formula is satisfied.

Ex.

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

**Are there values for the variables making the formula true?**

The formula is satisfied if $x$ and $z$ are true and $y$ and $w$ are false.

## Reductions

Let us assume that we have a problem A. We want to find an algorithm which solves the problems for all instances.

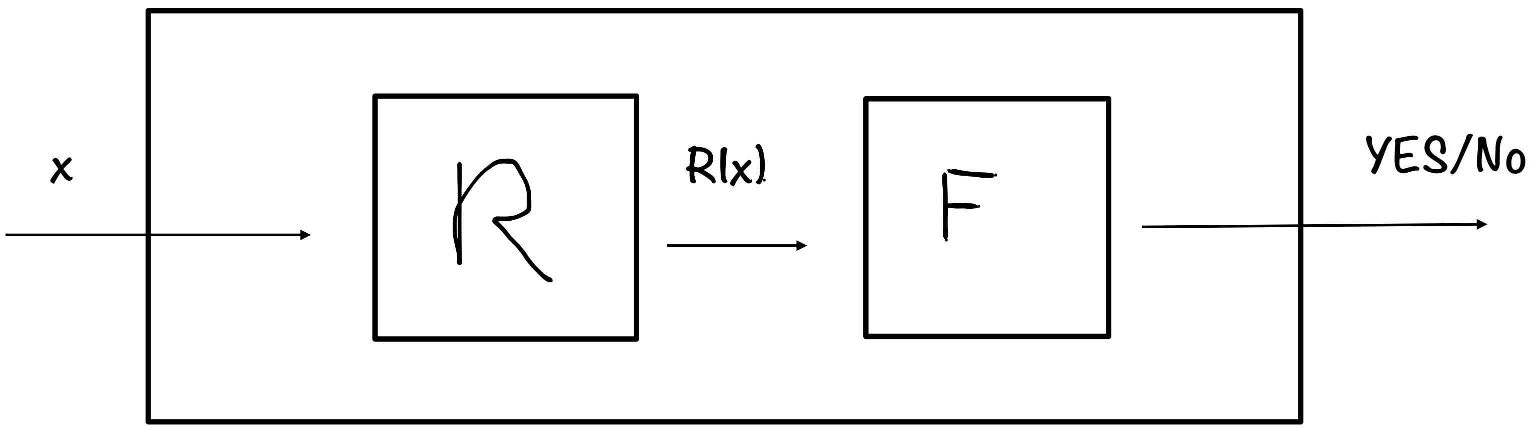
Let us assume that we have another problem B and that there is an algorithm F that solves the problem. This means that if y is an instance of B then the computation F(y) halts with yes or no as output and

The true answer is yes => F(y) = yes
The true answer is no => F(y) = no

Then a reduction of A to B is an algorithm R which takes inputs x to A and transforms them to inputs y = R(x) to B such that

The true answer to x in problem A is yes => F(R(x)) = yes
The true answer to x in problem A is no => F(R(x)) = no

New algorithm. Solves the A-problem



## Karp - Reductions

If a reduction should be useful it cannot be too complicated. We will usually demand that they are polynomial in the size of the input x. These polynomial time algorithms are called Karp - Reductions.

If A can be reduced to B by a Karp - Reduction we express this fact by writing

$$A \leq_P B$$

The subscript P stands for polynomial. Often, we will drop the P and assume that it is understood that the reduction is polynomial.

Two important consequences of the definition is:

1. If $A \leq B$ and $B \in P$ then $A \in P$.

2. If $A \leq B$ and $A \notin P$ then $B \notin P$.

This means that, potentially, reductions could be used to prove that a problem B cannot be solver efficiently, given that we know that another problem cannot.

If we can solve SAT efficiently, then there are many other problems that also can be solved efficiently.

But probably we cannot solve SAT efficiently?

The brilliant idea: Turn the reductions in the other direction!

If we have a problem A such that $SAT \leq A$, we have good reason to believe that A cannot be solved efficiently.

We will look at some "simplifications of SAT and see that they, in a sense are as hard to solve as SAT.

In fact, some authors use SAT as a name for CNF-SAT. In that case, then reduction below is meaningless.

## CNF-SAT

A formula on Conjunctive Normal Form is a formula that can be written as a disjunction of clauses which, in turn, are conjunctions of negated and un-negated variables.

Ex: $(x \vee y \vee z \vee w) \wedge (y \vee z) \wedge (x \vee y \vee w)$

CNF-SAT is the problem to decide if a CNF-formula is satisfiable or not.

It can be shown that $SAT \leq CNF\text{-}SAT$.

## The reduction CNF-SAT $\leq$ 3-CNF-SAT

We want to reduce SAT to 3-SAT:
Given a SAT-formula $\Phi = c_1 \wedge \cdots \wedge c_k$ we construct an equivalent 3-SAT-formel $\Phi_3$ be replacing each clause in $\Phi$ with one or more 3-SAT- clauses.

Assume that $c_i$ contains $j$ literals $l_1, \ldots l_j$. We bouild new clauses in $\Phi_3$:

$$\begin{array}{ll}
j = 3 & l_1 \vee l_2 \vee l_3 \\
j = 2 & (l_1 \vee l_2 \vee y_i) \wedge (l_1 \vee l_2 \vee \neg y_i) \\
j = 1 & (l_1 \vee y_i \vee z_i) \wedge (l_1 \vee y_i \vee \neg z_i) \wedge \\
 & (l_1 \vee \neg y_i \vee z_i) \wedge (l_1 \vee \neg y_i \vee \neg z_i) \\
j > 3 & (l_1 \vee l_2 \vee y_i^1) \wedge (\neg y_i^1 \vee l_3 \vee y_i^2) \wedge \\
 & (\neg y_i^2 \vee l_4 \vee y_i^3) \wedge \cdots \wedge (\neg y_i^{j-3} \vee l_{j-1} \vee l_j)
\end{array}$$

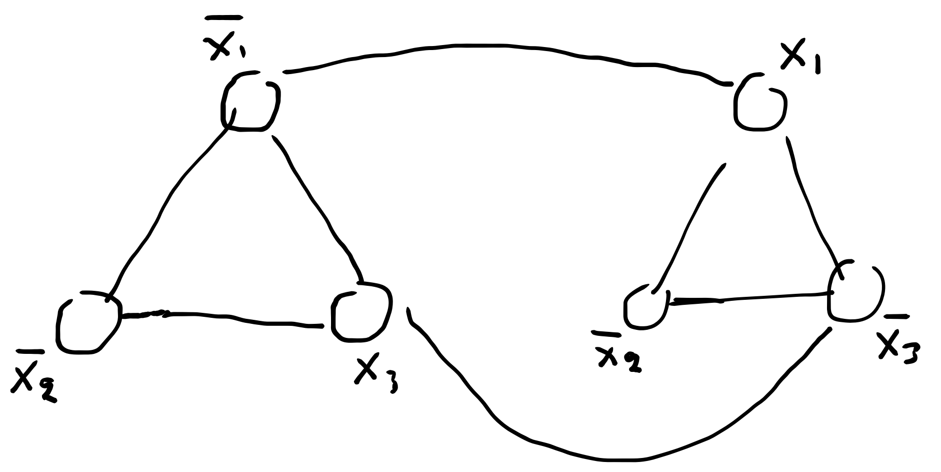
$\Phi_3$ is satisfiable exactly when $\Phi$ is.

## 3-CNF-SAT $\leq$ IS

We show the technique by looking at an example

$$\varphi = (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

We construct a graph



$K = 2$

First: Let us assume that the graph contains an IS of size 2. I must contain exactly one node from each triangle. For instance, we could choose the two $x_2$:s. This correspond to setting $x_2$ to TRUE.
Second: Let us assume that it is possible to satisfy . Then there is at least one true literal in each triangle. Chose corresponding nodes. The will form an IS of size 2.

## THE BIG QUESTION

It follows from the definition that $P \subseteq NP$.

$$\boxed{\text{IS } N = NP?}$$

Since 1971 this is the most famous open problem in computer science.

Most people believe that the answer is no. Then there must be problems in NP - P. SAT would be a plausible candidate.

It seems as if hard NP-Problems can be reduced to each other. This observation leads us to the following definition.

NP-Completeness: A problem Q is NP-Complete if

1. Q is in NP.
2. For each A in NP, there is a reduction from A to Q, i.e. all NP problems can be reduced to Q.

Are there any NP-Complete problems? Well, there are:

Cook's Theorem: SAT is NP-Complete

## Other NP-Complete problems

It is ease to see that reductions are transitive, i.e.

$A \leq B$ and $B \leq C \Rightarrow A \leq C$

We know that SAT $\leq$ INDEPENDENT SET. We also know that for each A in NP we have $A \leq$ SAT. But this means that for all A in NP we have $A \leq$ INDEPENDENT SET

So INDEPENDENT SET is an NP-Complete problem.

We realize that the NP-Complete problems must be the hardest problems in NP. If any NP-Complete problem can be solved efficiently then all can!

So we wouldn't expect to be able to find efficient solutions to NP-Complete problems.

The best way to "show" that a problem is impossible to solve efficiently is to show that it is NP-Complete.

This is the core of applied Complexity Theory.

But how do we show that a problem is NP-Complete?