Contents Lecture 9

- NP-completeness
- Polynomial time reductions
- The satisfiability problem
- Efficient certification and the definition of NP
- The Hamiltonian cycle problem
- The Traveling Salesman problem
- The Graph coloring problem
We define an algorithm to be efficient if it has a polynomial running time complexity $O(n^k)$ for some $k$.

Informally, a well-known problem is hard if nobody knows an efficient algorithm to solve it.

Note: we do not say "a problem for which there cannot exist an efficient algorithm!"
Complexity classes are used to categorize problems into how difficult they are to solve.

- The easiest problems are solvable by a polynomial time algorithm.
- This complexity class is simply called P.
- Another complexity class consists of the **NP-complete** problems, which **most likely** are hard to solve.
Many think NP-completeness is "mysterious" but it is not...

If you need to solve a problem which you can prove is NP-complete, then you know that you most likely should not try to solve it, at least not in its most general form.

Then don’t tell your boss "I am sorry, I am too dumb to write a program for the problem you gave me"
Why is NP-completeness useful to know about?

Instead tell your boss ”The problem is NP-complete and none of these famous people can solve your problem efficiently either”

Two approaches when we need to solve an NP-complete problem

1. Solve a less general problem by exploiting some special knowledge about the input
2. Solve a simpler problem which approximates the optimal solution

Optimizing compilers and many other programs use both approaches
An example of a hard problem: graph coloring

- Consider an undirected graph $G(V, E)$
- A $k$-coloring is an assignment of a color to each node using at most $k$ colors and such that no neighbors are assigned the same color
- If you can invent a polynomial time algorithm for graph coloring you win a prize of USD 1,000,000 from the Clay Institute of Mathematics
- Actually, you win the prize even your answer simply is "impossible" plus a proof
- Graph coloring is one of thousands of NP-complete problems
To make life simpler, we are happy with one-bit output, namely for yes or no.

So we formulate our problems as decision problems instead of optimization problems.

We don’t ask for a mapping of node to color using the minimum number of colors.

Instead: our graph coloring question is: "does a $k$-coloring exist for $G$?" i.e. a decision-problem with yes/no answer.

The complexities of answering these two kinds of questions are expected to be similar, i.e. either both hard or both simple.
Solving a problem versus checking a solution

In the general case, for a sufficiently large graph $G$ it would take billions of years to find a coloring with current algorithms.

If somebody has guessed a solution, it is trivial to check if it is a valid coloring, i.e. a correct solution to our problem.

For example, an answer to the question "is $G$ 3-colorable?" for the graph below can be $(a=\text{red}, b=\text{green}, c=\text{blue}, d=\text{blue})$.

![Graph with vertices labeled a, b, c, d and edges connecting a to b, c, d, and b to c, d.]

It is then trivial to check in polynomial time that no neighbors have the same color.

We also say it is easy to verify whether a solution is a valid coloring — even for huge graphs.
The NP complexity class

1. The complexity class NP consists of all problems for which there exists a polynomial time verification algorithm.
2. Note that each problem in P also is in NP:
   \[ P \subseteq NP \]
3. Also e.g. sorting is in NP because it is easy to check that an array is sorted.
4. We will come to NP-completeness later but first some new concepts.
Consider two decision problems $P_1$ and $P_2$, and assume:

- You already know an algorithm $A_2$ for solving problem $P_2$
- You want to have an algorithm $A_1$ for problem $P_1$
- The input to $A_1$ (or, $P_1$), is $x$
- You have figured out a function $f(x)$ which can map $A_1$ input to $A_2$ input
- Assume that $A_2(f(x)) = A_1(x)$

This means that you have just created an algorithm $A_1$:

- when $A_1(x)$ should return 0, $A_2(f(x)) = 0$, and
- when $A_1(x)$ should return 1, $A_2(f(x)) = 1$

If $f$ is efficient, you have created a **polynomial time reduction** from $P_1$ to $P_2$, and we write $P_1 \leq_P P_2$
What can we do when we have reduced $Y$ to $X$ with a polynomial time function $f$?

We can compare the relative complexity of the problems $X$ and $Y$.

Which one is hardest to solve, $X$ or $Y$?

Since we know we can solve $Y$ using $X$ but we don’t know if we can solve $X$ using $Y$, it must be the case that $X$ is at least as hard to solve as $Y$ — possibly much harder.

This means we can use $\leq_p$ to compare the complexity of problems just as we can use $\leq$ to compare integers.

Consequences:

- If $X$ is easy to solve, then $Y$ must also be easy to solve.
- If $Y$ is hard to solve, then $X$ must also be hard to solve.

”Easy” above means polynomial time, and ”hard” not in polynomial time.
If $Y \leq_p X$ and $X \leq_p Y$ then we write $X \equiv_p Y$

As expected it means we can solve $X$ in polynomial time if and only if we can solve $Y$ in polynomial time.
P is the set of all problems which can be solved in polynomial time

NP is the set of all problems which can be verified in polynomial time (i.e. a proposed solution can be checked in polynomial time)
Definition of NP-completeness

- Consider a problem $X \in NP$
- Assume every problem $Y \in NP$ can be reduced to $X$
- Then $X$ is NP-complete
- That is, there are two conditions for a problem $X$ to be NP-complete:
  1. $X \in NP$
  2. For all $Y \in NP$ we have $Y \leq_P X$
- Therefore, NP-complete problems are the hardest problems in NP
- A valid question quickly becomes: are there any NP-complete problems? Yes, proved in 1971 by Cook
- NP-complete problems belong to the complexity class NPC
Definition of NP-hard

Consider a problem \( X \), which possibly is or is not in NP

Assume every problem \( Y \in NP \) can be reduced to \( X \)

Then \( X \) is NP-hard

Therefore, NP-hard problems are even harder than NP-complete problems
Summary so far and a what to do next

- Four complexity classes P, NP, NPC, and NP-hard:
  - \( X \in P \): \( X \) can be solved in polynomial time
  - \( X \in NP \): \( X \) can be verified in polynomial time
  - \( X \in NPC \): \( X \in NP \) and \( Y \in NP \Rightarrow Y \leq_P X \)
  - \( X \) is NP-hard: \( Y \in NP \Rightarrow Y \leq_P X \)

- \( Y \leq_P X \) can be used to show that \( Y \) is easy or \( X \) is hard

- Next we will demonstrate some reductions

- After that will prove that a problem called Circuit satisfiability is NP-complete

- Finally we will use reductions to prove that some other problems also are NP-complete — using reductions may make this relatively convenient
Problem: Independent set

- Consider an undirected graph $G(V, E)$
- Let $S \subseteq V$ such that for no nodes $u, v \in S$ we have $(u, v) \in E$
- $S$ is called an **Independent set**
- Trivially $S = \{v\}$ for any $v \in V$
- The problem is to find an $S$ with maximum size, $|S|$

Any suggestions?

Of course we want to find and print such an $S$

But our decision problem only is: *is there an independent set $S$ such that $|S| = k$?*
Problem: Independent set

- Let $S \subseteq V$ such that for no nodes $u, v \in S$ we have $(u, v) \in E$
- The problem is to find an $S$ with maximum size, $|S|$

Two independent sets of size four:
- $S_1 = \{b, c, e, g\}$
- $S_2 = \{a, e, f, g\}$
Consider an undirected graph $G(V, E)$
Let $S \subseteq V$ such that for every edge $(u, v) \in E$ we have $\{u, v\} \subseteq S$
In other words: every edge $e \in E$ has at least one end in $S$
$S$ is called a **Vertex cover**
Trivially $S = V$
The problem is to find an $S$ with minimum size, $|S|$

Any suggestions?
$S_1 = \{a, d\}$
Problem: Vertex cover

- Let $S \subseteq V$ such that for every edge $(u, v) \in E$ we have \( \{u, v\} \subseteq S \)
- In other words: every edge $e \in E$ has at least one end in $S$

\[ S = \{a, d, f, g\} \]

- With this $S$ every edge $e \in E$ has one end in $S$
- Is there a smaller vertex cover?
- Which problem is harder? Independent set or Vertex cover, or equally simple or hard?

Jonas Skeppstedt (jonasskeppstedt.net)  Lecture 9  2021  20 / 56
A reduction from Independent set to Vertex cover

- Is there an independent set $A$ of size $k$?
- Is there a Vertex cover $B$ of size $k$?
- We are not interested in $A$ or $B$ — only the 'yes' or 'no' answers
- Let us try to show: Independent set $\leq_P$ Vertex cover
- How are these problems related?
Independent set and Vertex cover are related

**Lemma**

In a graph $G = (V, E)$, $S$ is an independent set $\iff V - S$ is a vertex cover.

**Proof.**

- We first prove the $\Rightarrow$ direction, so assume $S$ is an independent set.
- Consider any edge $(u, v) \in E$.
- Since $S$ is an independent set, not both of $u$ and $v$ are in $S$.
- Therefore at least one of $u$ and $v$ are in $V - S$ which therefore is a vertex cover.
- To prove the $\Leftarrow$ direction, assume $V - S$ is a vertex cover.
- Consider any edge $(u, v) \in E$.
- Since $V - S$ is a vertex cover, at least one of $u$ and $v$ is in $V - S$.
- Therefore both $u$ and $v$ cannot be in $S$ which therefore is an independent set.
A reduction from Independent set to Vertex cover

- We now know that $S$ is an independent set if and only if $V - S$ is a vertex cover.
- Back to our decision problems:
  - Is there an independent set of size $k$?
  - Is there a vertex cover of size $k$?
- These questions refer to different $k$ so we can instead write:
  - Is there an independent set of size $x$?
  - Is there a vertex cover of size $y$?
- To reduce Independent set to Vertex cover, we can use the polynomial time reduction function $f(V, x) = |V| - x$.
- Our polynomial time reduction therefore becomes: is there an independent set of size $x = \text{is there a vertex cover of size } |V| - x$?
- And therefore: Independent set $\leq_P$ Vertex cover.
A reduction from Vertex cover to Independent set

- Of course, we can a similar reduction in the other direction
- Also, these two problems are equally hard — or easy — to solve
- Note: we only compare the relative complexity
- We do not know if there exists a polynomial time algorithm for these problems
Another reduction: from Vertex cover to Set cover

- Vertex cover selects a minimal number of vertices $S$ so that for all edges $(u, v) \in E$ at least one of $u$ and $v$ are in $S$
- In Set cover we have a set $S$ and a number of subsets $S_1, S_2, \ldots, S_m$ of $S$
- We want to select a minimal number of subsets such that their union is $S$
- So assume we have an algorithm $A$ for Set cover and want to use it to solve Vertex cover.
- We construct an instance $f(x)$ of Set cover from our instance $x$ of vertex cover.
- Then we use $A$ to determine if we can use only $k$ of the subsets?
- What should the reduction function $f$ be?
- Think about this one minute!
A reduction function $f$ from Vertex cover to Set cover

Our instance of vertex cover is called $x$ and has a graph $G(V, E)$

Since it is edges we want to cover (using nodes), let $S = E$

Define a subset $S_v = \{(v, w) \mid (v, w) \in E\}$
We have now constructed an instance $f(x)$ of Set cover:

- $S_a = \{(a, b)\}$
- $S_b = \{(a, b), (b, d), (b, h)\}$
- $S_c = \{(c, d), (c, e), (c, f)\}$
- $\ldots$
- $S_j = \{(i, j), (g, j)\}$

This looks reasonable and we can therefore try to prove this is a correct reduction.
Correctness of the reduction $f$

Lemma

There exists a vertex cover of $G(V, E)$ using at most $k$ nodes $\Leftrightarrow$ there exists a set cover of $S$ using at most $k$ subsets of $S$ created by $f$.

Proof.

- We prove the $\Leftarrow$ direction first.
  - Assume $A(f(x), k) = 1$. Then there is a set cover using subsets $S_{v_1}, S_{v_2}, \ldots \cup S_{v_i}$ such that $i \leq k$.
  - Therefore every edge $e \in E$ is incident to at least one of the nodes $\{v_1, v_2, \ldots, v_i\}$, which means the nodes $\{v_1, v_2, \ldots, v_i\}$, is a vertex cover of size at most $k$.
- To prove the $\Rightarrow$ direction, assume $\{v_1, v_2, \ldots, v_i\}$, is a vertex cover of size at most $k$.
  - Then the subsets $S_{v_1}, S_{v_2}, \ldots \cup S_{v_i}$ such that $i \leq k$ is a set cover of size at most $k$. 

Jonas Skeppstedt (jonasskeppstedt.net)
Conclusion and remark

- We have proved we can reduce Vertex cover to Set cover
- Therefore $\text{Vertex cover} \leq_P \text{Set cover}$
- Our reduction only needed to construct one instance of Set cover
- We are allowed to make a polynomial number of calls to $A$ but very often we can construct an instance which only needs one call
- Since both problems are NP-complete we can also reduce from Set cover to Vertex cover — however, that is much more complicated and we will not do that
Consider a new problem $Y$ such that:
- We cannot come up with an efficient algorithm for $Y$
- We suspect it is NP-complete

How can we prove it is?
- Firstly, does it have a polynomial time verifier so $Y \in NP$?
- Can we make a reduction from a problem $X$ which is known to be NP-complete?
- That is: can we solve $X$ using a reduction to $Y$? $X \leq_P Y$
- If that is the case, we have proved $Y$ is NP-complete
The first problem that was shown to be NP-complete is Circuit satisfiability.

A boolean circuit consists of input signals, wires, gates, and output signals.

A gate is one of:

<table>
<thead>
<tr>
<th>Gate</th>
<th>Symbol</th>
<th>Output</th>
<th>Input Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>$x \land y$</td>
<td>$1$ if all inputs are $1$</td>
<td>at least two</td>
</tr>
<tr>
<td>OR</td>
<td>$x \lor y$</td>
<td>$1$ if any input is $1$</td>
<td>at least two</td>
</tr>
<tr>
<td>NOT</td>
<td>$\neg x$</td>
<td>Negation of input</td>
<td>exactly one</td>
</tr>
</tbody>
</table>

All digital circuits can be implemented with these.

To build a computer, we also need storage elements, and a clock signal.

A digital circuit is an extremely general and powerful concept.
In theory we can implement any algorithm using only circuits — the disadvantage is that it will become too big to be practical for non-trivial algorithms.

And it is nice to be able to run different apps on a computer/phone and not only one so we prefer using memories so we can put a different app there and run it instead.

What can be computed is the same, however.

Another practical difference is that a circuit has a fixed number of input bits while an algorithm can process any number of input bits.
A simple circuit

- Let $i_1, i_2, \ldots, i_n$ be the $n$ input bits to a circuit.
- Assume we only have one output bit.
- Thus our circuit is a function $f(i_1, i_2, \ldots, i_n)$ with output 0 or 1.
- With $n = 3$ we can for example have $f(i_1, i_2, i_3) = (i_1 \land i_2) \lor \neg i_3$.
- Since $\land$ has higher precedence than $\lor$ we write this as:
  
  $$f(i_1, i_2, i_3) = i_1 \land i_2 \lor \neg i_3$$

- Circuit satisfiability is the following problem: given a circuit with $n$ inputs, can we select the values of each input bit $i_1, i_2, \ldots, i_n$ so that the output becomes 1?
- If we can, then we have satisfied the circuit.
- In our example, $f$ becomes 1 if both $i_1$ and $i_2$ are 1, or $i_3$ is 0, and it becomes 0 otherwise.
- Therefore this circuit is satisfiable.
The Cook theorem was published in 1971.

1973 Levin published a similar result in Russian.

The theorem is sometimes called The Cook-Levin theorem but I prefer the Cook theorem since Cook was first.

Theorem

Circuit satisfiability is NP-complete.
The Cook theorem: Circuit satisfiability is NP-complete

Theorem

Circuit satisfiability is NP-complete.

Proof.

We will only sketch a proof because some of the details are too tedious.

We need to show two things:

1. Circuits satisfiability is in NP, and
2. For all $X \in \text{NP}$ we have $X \leq_p \text{Circuit satisfiability}$. 
Proof.

1. Circuit satisfiability is in NP, and
2. For all $X \in \text{NP}$ we have $X \leq_P \text{Circuit satisfiability}$.

Assume we have a circuit $C$ and found an assignment of values to all input variables $v_i$ which results in an output of 1 from $C$.

That is: $C$ is a concrete circuit with some particular gates — and not any ”abstract” circuit

So we are given a sequence $v_1, v_2, \ldots, v_n$.

How can we check if this is a solution to Circuit satisfiability for $C$?

We can just evaluate $C$ with this sequence as input and check that the output is 1.

And this is of course trivial to do in polynomial time.
Proof.

1. Circuit satisfiability is in NP, and
2. For all $X \in NP$ we have $X \leq_P$ Circuit satisfiability.

We now need to prove that every problem $X$ in NP can be solved by reducing $X$ to Circuit satisfiability.

What is needed for that?

For a given problem $X$ and for any input to $X$ we must be able to solve $X$ using Circuit satisfiability, i.e. determine if $X$ for that input should be a "yes" or a "no" (or, 1 or 0)
Proof.

- For $X$ to be in NP, it must have a polynomial time verification algorithm $A$
- $A$ takes two inputs:
  - the input $I$ to $X$, and
  - the proposed solution $S$ to $X$.
- So $A(I, S)$ should in polynomial time determine if $S$ is a solution to $X$ when the input is $I$
- $I$ is a string of $n$ bits and $S$ is a string of $p(n)$ bits
- How can we use Circuit satisfiability for this??
Proof.

- $A(I, S)$ determines in polynomial time if $S$ is a solution to $X$.
- We can now create a circuit which implements $A(I, S)$.
- With all $n + p(n)$ bits from $I$ and $S$ this circuit $C$ will output 0 or 1 depending on if $S$ was the solution.
- There are $n + p(n)$ boolean input variables to $C$.
- To use $C$ to solve $X$ we will let $C$ find $S$ for us!
- We do this as follows: let the first $n$ bits to $C$ be $I$, and the remaining boolean variables $v_1, v_2, \ldots, v_{p(n)}$ be the unknown variables for which Circuit satisfiability should find an assignment.
Proving Circuit satisfiability is NP-complete

Proof.

- We have just shown the key idea of how we can use Circuit satisfiability to solve any problem in NP.
- The critical sentence I did not explain is: 
  \[ We \ can \ now \ create \ a \ circuit \ which \ implements \ A(I, S) \]
- This proof of Circuit satisfiability being NP-complete relies on that we actually can take a polynomial time algorithm \( A \) and create a circuit \( C \) so that \( A(I, S) = C(I, S) \)
- Why should that \( C \) exist and why should we be able to create it?
- That is the tedious part. We need to translate every step in \( A \) down to gates.
- For example: \( x = a > b ? c * d : e / f \) will become gates for evaluating \( a > b \), the multiplication, and the division, and then a multiplexer which has the outcome \( > \) as control input and of the arithmetic operations as data inputs.
Proof.

- More complicated code such as $x\text{[rand(0)]} = y\text{[rand(0)]} \times 2$ with arrays and pseudo random numbers can also be translated to gates but it is not as straightforward as for simple expressions.

- The main reason we can handle any algorithm is that we can view the state of a computer as a state in a finite state machine which in itself can be translated to gates, although a huge number of gates.

- Cook proved his theorem using Turing machines, which are equivalent to computers.
3-Satisfiability

- 3-Satisfiability (3-SAT) is a problem very similar to Circuit satisfiability.
- In 3-SAT we also have gates and variables which should be assigned values so that the only output boolean value becomes a 1.
- In 3-SAT a term is either a boolean variable $x_i$ or its negation $\neg x_i$.
- For simplicity we write $\overline{x_i}$ instead of $\neg x_i$.
- A disjunction in 3-SAT always contains three terms, e.g. $x_1 \lor \overline{x_3} \lor x_4$.
- Such a disjunction is also called a clause.
- Negation is only permitted with a variable, and thus only in a clause.
- There can be any number of clauses and each must evaluate to 1.
- Therefore, at the highest level, 3-SAT consists of a conjunction of clauses.
- An example instance of 3-SAT is:

$$((x_1 \lor \overline{x_3} \lor x_4) \land (\overline{x_1} \lor \overline{x_2} \lor x_3) \land (\overline{x_2} \lor \overline{x_4} \lor x_5) \land (\overline{x_2} \lor x_3 \lor x_4))$$
The Hamiltonian Cycle Problem asks whether there exists a simple cycle with all nodes of a directed graph.

In other words, each node must be on this path exactly once, and we must return to the node where we started.

We will next prove that this problem is NP-complete.

How can we do that?

The usual start is:

- Prove the problem is in NP, i.e. has a polynomial-time verification.
- Find a suitable problem $Q$ known to be NP-complete
- Solve $Q$ using the new problem, i.e. reduce from $Q$

A polynomial time verification of a proposed solution $C$ simply checks that $C$ is a cycle and that each node is in $C$ exactly once. So the problem is in NP.
The Hamiltonian Cycle Problem

- It turns out it often is practical to reduce from 3-SAT
- Given an instance of 3-SAT we should create a graph $G$
- We then solve the Hamiltonian Cycle problem for $G$ to prove that this problem is at least as hard as 3-SAT, i.e. NP-complete
- Of course, $G$ must be created so that Hamiltonian Cycle has a solution if and only if the 3-SAT has a solution
The Hamiltonian Cycle Problem

- Assume we have \( n \) variables \( x_i \) and \( k \) clauses \( C_j \) in the 3-SAT instance
- \( \Phi = C_1 \land C_2 \land \ldots \land C_k \)
- \( C_j = t_{j1} \lor t_{j2} \lor t_{j3} \)
- Each \( t \) is a term, or literal, which is either a variable or the negation of a variable
- For example: \( \Phi = (x_1 \lor x_2 \lor \overline{x_3}) \land (\overline{x_1} \lor \overline{x_2} \lor x_3) \)
- \( n = 3 \) and \( k = 2 \)
- We will next create a graph from \( \Phi \) in steps
The Hamiltonian Cycle Problem

- $\Phi = (x_1 \lor x_2 \lor \overline{x_3}) \land (\overline{x_1} \lor \overline{x_2} \lor x_3)$
- There is one "row" in the graph for each 3-SAT input variable $x_i$
- Every Hamiltonian cycle must go from $s$ to either $x_{11}$ or $x_{14}$
- A row can be passed either in left or right direction
- As the graph looks now, there are $2^3$ Hamiltonian cycles since we can select either left or right direction in each of the three rows
- The number of nodes in each row is twice the number of clauses, $k$
The Hamiltonian Cycle Problem

- $\Phi = (x_1 \lor x_2 \lor \overline{x_3}) \land (\overline{x_1} \lor \overline{x_2} \lor x_3)$
- A Hamiltonian cycle going right in row $i$ means $x_i = 1$, and going left means $x_i = 0$
- If clause $C_j$ contains $x_i$ we should add an edge from row $i$ to $C_j$, and from $C_j$ to row $i$
- Since we have $x_i$, these edges should be in the right direction
- For $\overline{x_i}$, there should be edges in the left direction instead
The Hamiltonian Cycle Problem

\[ \Phi = (x_1 \lor x_2 \lor \overline{x}_3) \land (\overline{x}_1 \lor \overline{x}_2 \lor x_3) \]

- A Hamiltonian cycle going right in row \( i \) means \( x_i = 1 \), and going left means \( x_i = 0 \)
- If clause \( C_j \) contains \( x_i \) we should add an edge from row \( i \) to \( C_j \), and from \( C_j \) to row \( i \)
- Since we have \( x_i \), these edges should be in the right direction
- For \( \overline{x}_i \), there should be edges in the left direction instead
- Edges incident to a clause node are dashed only for visibility and are not special in any way
Another problem in which a sequence of all nodes of a graph is requested is the Traveling Salesman Problem (TSP).

Consider a set of cities with distances between every pair of cities.

We denote the distance between two cities \( u \) and \( v \) by \( d(u, v) \).

It is not required that \( d(u, v) = d(v, u) \).

Furthermore, it is also not required that \( d(u, v) + d(v, w) \geq d(u, w) \), i.e. the triangle inequality is not required to be satisfied.

A **tour** visits all cities and returns to the originating city.

The Traveling Salesman problem asks if there is a tour using a total distance of at most \( x \).

We will next prove that TSP is NP-complete by reduction from Hamiltonian cycle.

If we can solve Hamiltonian cycle using TSP, TSP is at least as hard as Hamiltonian cycle.

It is clear the TSP is in NP.
Reducing Hamilton Cycle to Traveling Salesman

- Given a directed graph $G(V, E)$ for the Hamilton Cycle problem, we construct an instance of TSP as follows:

  - For each $(u, v) \in E$ we assign a distance $d(u, v) = 1$ and for all pairs such that $(u, v) \notin E$ we assign a distance $d(u, v) = 2$.

- If and only if there is a solution to TSP for this graph with a total distance of $n$, there exists a Hamiltonian cycle for $G$.

- The proof of this claim is trivial. If there is such a TSP tour, this tour constitutes a Hamiltonian cycle, and if $G$ has a Hamiltonian cycle, the TSP tour must have length $n$. 

Jonas Skeppstedt (jonasskeppstedt.net)
Recall the graph coloring problem: for an undirected graph $G(V, E)$. Is there a mapping from node to colors so that neighboring nodes are assigned different colors and at most $k$ colors are used?

We have already seen that for $k = 2$ the decision problem is in P.

We will next show that for $k = 3$ the decision problem is NP-complete.

Firstly, it is clear the problem is in NP.
Reduction from 3-SAT to 3-coloring

- Given a 3-SAT instance $I$ with $n$ variables and $k$ clauses, we will create a graph which is 3-colorable if and only if $I$ is satisfiable.
- We start with a triangle consisting of the nodes $t$, $f$, and $b$.
- Nodes $t$ and $f$ correspond to true and false, or 1 and 0 respectively.
- Node $b$ is a node, often called base in the literature, which is used to force nodes corresponding to variables and their negation to be colored with the same color as $t$ or as $f$.

![Graph representation of 3-SAT to 3-coloring reduction]

- For each variable $x_i$, and $\overline{x_i}$, there are nodes $v_i$ and $\overline{v_i}$. 
Reduction from 3-SAT to 3-coloring

Since each $v_i$ and $\overline{v_i}$ is a neighbor of $b$, a 3-coloring must select the color of either $t$ or $f$ for them.

We will denote the color of $t$ by $T$, the color of $f$ by $F$ and the color of $b$ by $B$ below.
Representing clauses in $G$

- We denote the three terms, or literals, in clause $C_j$ by $p_j$, $q_j$ and $r_j$.
- Thus, if $C_j = x_1 \lor \overline{x_2} \lor x_n$, then $p_j = v_1$, $q_j = \overline{v_2}$, and $r_j = v_n$.
- We need to create a subgraph for each clause which will be colorable if and only if at least one term is colored with $T$.
- Such a subgraph needs to have a certain node which is neighbor to both $f$ and $b$ so that it can be colored with $T$ (if at least one term also is colored with $T$, of course).
- Essentially, we want to create the equivalence of an OR-gate, or disjunction.
Representing an OR-gate

- Assume $C_j = x_1 \lor \overline{x_2} \lor x_n$, and $p_j = v_1$, $q_j = \overline{v_2}$, and $r_j = v_n$

A subgraph with these six nodes $y_{jk}$, $1 \leq k \leq 6$ is created for each $C_j$

- As can be easily verified node $y_{j6}$ can be colored with $T$ if at least one of $p_j$, $q_j$ and $r_j$ is colored with $T$
An example

- Let $c(v)$ denote the color of $v$ and assume $c(p_j) = c(q_j) = c(r_j) = F$
- So $(c(v_1), c(v_2), c(v_3)) = (F, T, F)$ and $(x_1, x_2, x_3) = (0, 1, 0)$

- $c(y_{j1}) \in \{B, T\}$, and $c(y_{j2}) \in \{B, T\}$ so $c(y_{j1}) = T$
- $c(y_{j1}) \in \{B, T\}$, and $c(y_{j2}) \in \{B, T\}$ so $c(y_{j1}) = T$