Contents Lecture 9

- NP-completeness
- Polynomial time reductions
- Efficient certification and the definition of NP
- The circuit satisfiability problem
- The formula satisfiability problem (SAT)
- The Hamiltonian cycle problem
- The Traveling Salesperson problem
- The Graph coloring problem
- SAT solving

Hard problems

- 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!"

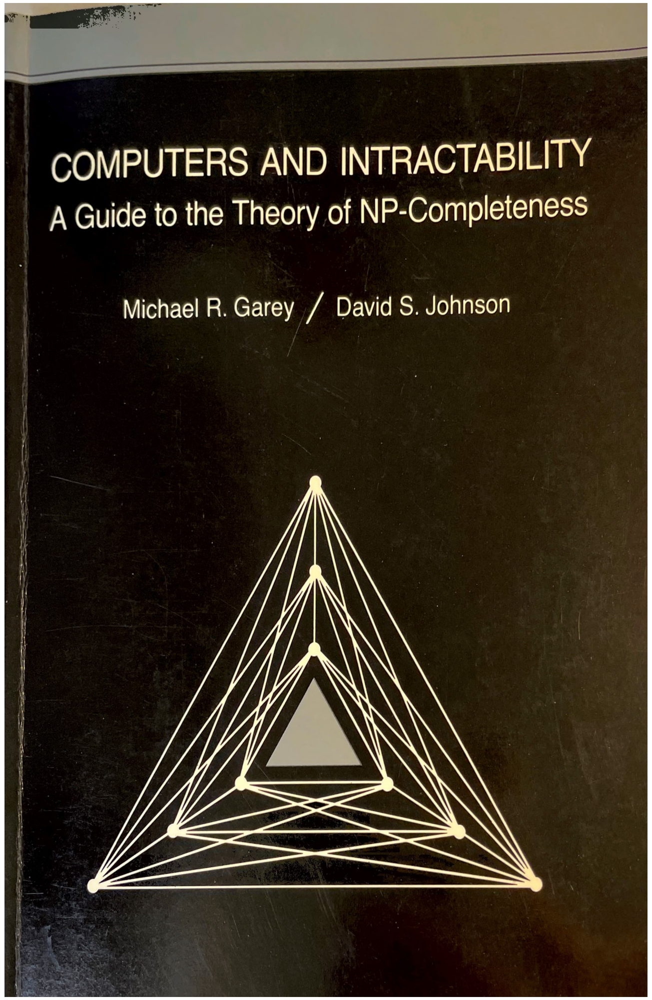
Complexity classes

- Complexity classes are used to categorize problems into how difficult they are to solve
- The easiest problems are solvable by polynomial time algorithms
- This complexity class is simply called P
- Another complexity class consists of the **NP-complete** problems, which most likely are hard to solve

Why is NP-completeness useful to know about?

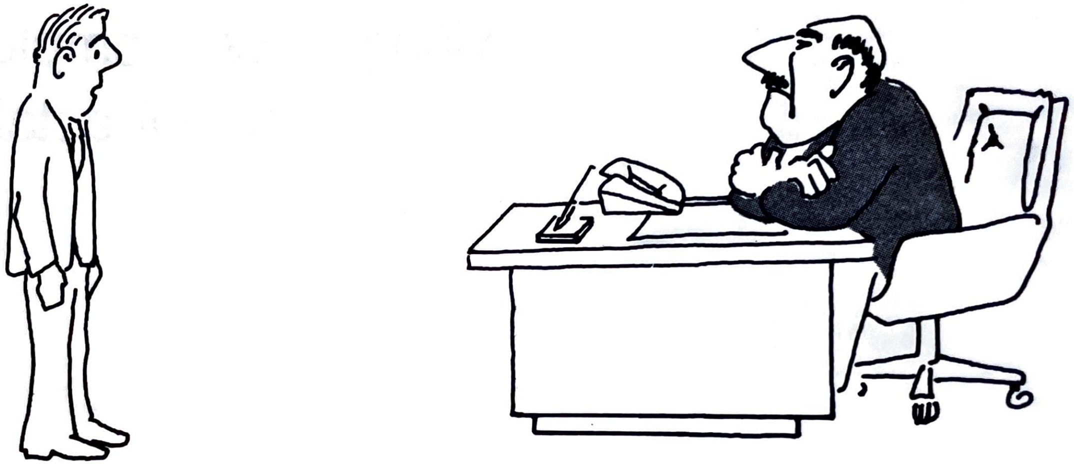
- Many think NP-completeness is "mysterious" but it is not...
- Except that nobody knows if $P = NP$
- If you need to solve a problem which you can prove is NP-complete, then you know that you most likely should not try to solve it, at least not in its most general form

Classic book on NP theory



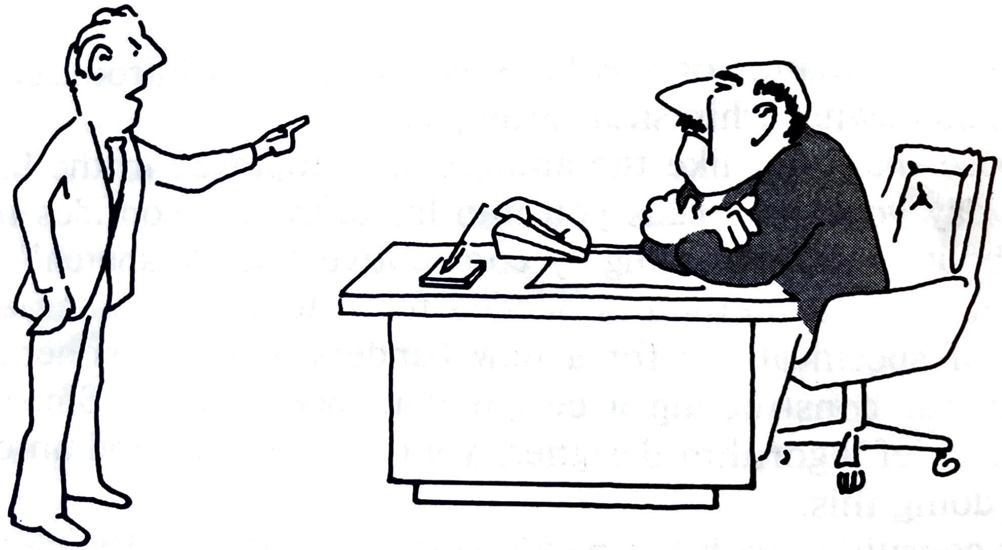
- Next three figures from this book

Wrong answer 1



"I can't find an efficient algorithm, I guess I'm just too dumb."

Wrong answer 2 (at least we think so)



"I can't find an efficient algorithm, because no such algorithm is possible!"

Correct answer

"I can't find an efficient algorithm, but neither can all these famous people."

Why is NP-completeness useful to know about?

- 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

An example of a hard problem: graph coloring

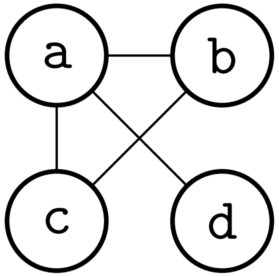
- Consider an undirected graph $G(V, E)$
- A k-coloring is an assignment of a color to each node using at most k colors and such that no neighbors are assigned the same color
- If you can invent a polynomial time algorithm for graph coloring you win a prize of USD 1,000,000 from the Clay Institute of 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

Optimization versus decision problems

- To make life simpler, we are happy with yes or no answer
- 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
- Decision problem: "does a k-coloring exist for G?"
- 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 correct
- For example, an example solution to the question "is G 3-colorable?" for the graph below can be (a=red,b=green,c=blue,d=blue)



- 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

- The complexity class NP consists of all problems for which there exists a polynomial time verification algorithm
- Note that each problem in P also is in NP:
$$P \subseteq NP$$
- Also e.g. sorting is in NP because it is easy to check that an array is sorted
- We will come to NP-completeness later but first some new concepts

Polynomial time reduction

- 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 P_1 is x
  - You have a function $f(x)$ which can map A_1 input to A_2 input
- If $A_2(f(x)) = A_1(x)$, 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 $P_1 \leq_{\mathrm{P}} P_2$

$Y \leq_P X$

- 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

$X \equiv_{\mathrm{P}} Y$

- If $Y \leq_{\mathrm{P}} X$ and $X \leq_{\mathrm{P}} Y$ then we write $X \equiv_{\mathrm{P}} Y$
- As expected it means we can solve X in polynomial time if and only if we can solve Y in polynomial time

Summary of the complexity classes P and NP

- 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
- Nobody knows if $P = NP$

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 $Y \leq_{\mathrm{P}} X$
- 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
- The difference between NP-complete and NP-hard is that it must be easy to verify a proposed solution to an NP-complete problem, which is not necessary for an NP-hard problem.

Summary so far and a what to do next

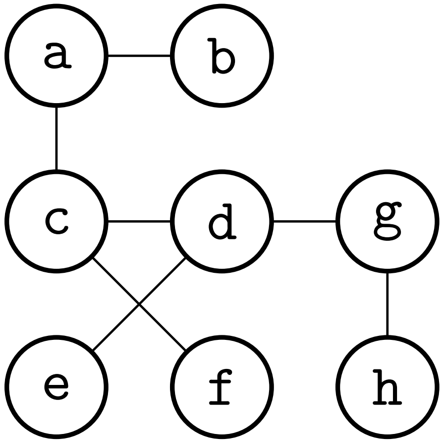
- Four complexity classes P, NP, NPC, and NP-hard:

$X \in P$	X can be solved in polynomial time
$X \in NP$	X can be verified in polynomial time
$X \in NPC$	$X \in NP$ and $Y \in NP \Rightarrow Y \leq_{\mathrm{P}} X$
X is NP-hard	$Y \in NP \Rightarrow Y \leq_{\mathrm{P}} X$

- $Y \leq_{\mathrm{P}} X$ 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

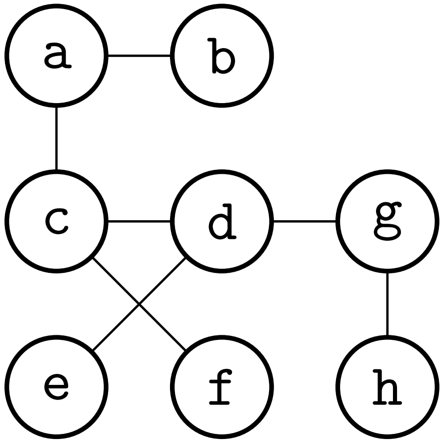
- Consider an undirected graph $G(V, E)$
- Let $S \subseteq V$ such that for no nodes $u, v \in S$ we have $(u, v) \in E$
- S is called an **independent set**
- Trivially $S = \{v\}$ for any $v \in V$
- The problem is to find an S with maximum size, $|S|$



- 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 $|S| = k$?*

Continued: Independent set

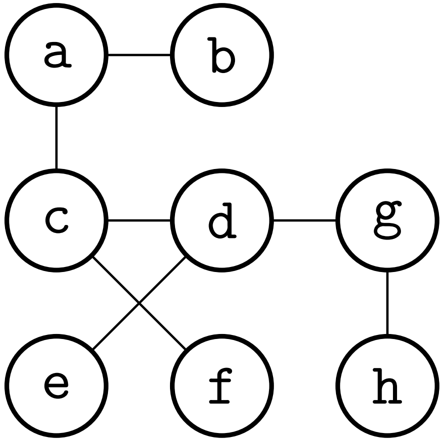
- Again:
  - Let $S \subseteq V$ such that for no nodes $u, v \in S$ we have $(u, v) \in E$
  - The problem is to find an S with maximum size, $|S|$



- Two independent sets of size four:
  - $S_1 = \{b, c, e, g\}$
  - $S_2 = \{a, e, f, g\}$

Problem: Vertex cover

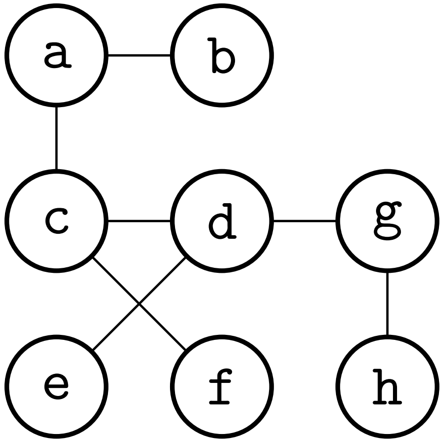
- Consider an undirected graph $G(V, E)$
- Let $S \subseteq V$ such that for every edge $(u, v) \in E$ we have $u \in S \vee v \in S$
- In other words: every edge $e \in E$ has at least one end in S
- S is called a **vertex cover**
- Note: it is the vertices that perform the covering of edges.
- Trivially $S = V$
- The problem is to find an S with minimum size, $|S|$



- Any suggestions?

Continued: Vertex cover

- Let $S \subseteq V$ such that for every edge $(u, v) \in E$ we have $u \in S \vee v \in S$
- 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?

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 $\leq_{\mathrm{P}}$ 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 $\Leftrightarrow$ $V - S$ 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

$\square$

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 = is there a vertex cover of size $|V| - x$?*
- And therefore: Independent set $\leq_{\mathrm{P}}$ Vertex cover

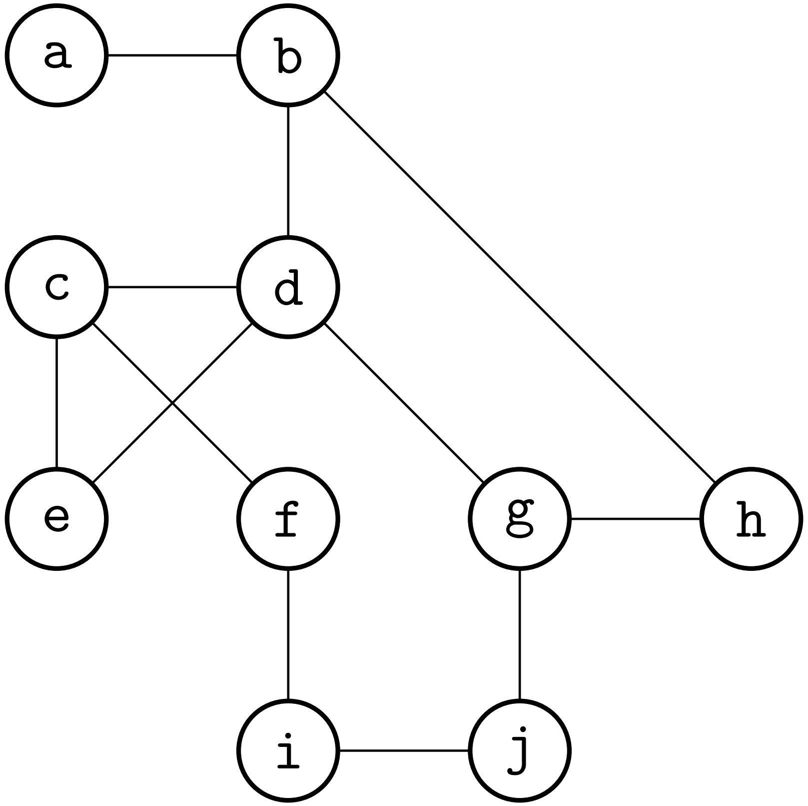
A reduction from Vertex cover to Independent set

- We can use 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

- We know 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 subsets $S_1, S_2, \ldots, S_m$ of S
- We want 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 $S_v = \{(v, w) \mid (v, w) \in E\}$

Continued

$$
\begin{aligned}
S_a &= \{(a,b)\} \\
S_b &= \{(a,b),(b,d),(b,h)\} \\
S_c &= \{(c,d),(c,e),(c,f)\} \\
&\vdots \\
S_j &= \{(i,j),(g,j)\}
\end{aligned}
$$

- We have now constructed an instance $f(x)$ of Set cover
- 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.

Conclusion and remark

- We have proved we can reduce Vertex cover to Set cover
- Therefore Vertex cover $\leq_P$ 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

Proving that a new problem is NP-complete

- 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? $X \leq_{\text{P}} Y$
- If that is the case, we have proved Y is NP-complete

Circuit satisfiability

- 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:

AND	$x \wedge y$	output $=1$ if all inputs are 1	at least two input signals
OR	$x \vee y$	output $=1$ if any input is 1	at least two input signals
NOT	$\neg x$	output $=$ negation of input	exactly one input signal

- 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

Algorithm versus circuit

- 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: Circuit satisfiability is NP-complete

- 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 $X \in NP$ we have $X \leq_{\mathrm{P}}$ Circuit satisfiability.

$\square$

Proving Circuit satisfiability is NP-complete

Proof.

- 1. Circuit satisfiability is in NP, and
  2. For all $X \in NP$ we have $X \leq_{\mathrm{P}}$ 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 $i_1, i_2, \ldots, i_n$.
- 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.

□

Proving Circuit satisfiability is NP-complete

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)

□

Proving Circuit satisfiability is NP-complete

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??

$\square$

Proving Circuit satisfiability is NP-complete

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

$\square$

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.

Proving Circuit satisfiability is NP-complete

Proof.

- More complicated code such as `x[rand()] = y[rand()] * 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.

Formula Satisfiability (called SAT)

- Instead of digital gates we use operators: $\vee$, $\wedge$ and $\neg$.
- In circuits the output of one gate can be input to multiple other gates.
- Can we easily translate a circuit into a formula?
- Can we reduce Circuit satifiability to SAT?
- First approach: translate the gates to their corresponding operators in an obvious way starting with the output.
- Obvious way: ”recursively copy gates from each input”
- Two problems:
  - We prefer a formula on the form of a conjunction of clauses, e.g.: $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$
  - Doing this in an obvious way is not efficient

Conjunctive normal form

- Conjunctive normal form is a conjunction of clauses.
- CNF such as: $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$
- We can achieve this by using DeMorgan's laws and the distributive laws to move $\neg$ and $\vee$ down
- So this is easily solved

Exponential size of the formula

- Recall the output of one gate T can be input of multiple other gates, $g_1, g_2, ..., g_n$
- Then when each of g_i translates their input from T they will create multiple copies of the same formula
- We will next see a way to avoid that.

Translating each gate

- Recall $p \rightarrow q$ means $\neg p \vee q$
- So $p \leftrightarrow q$ means $(\neg p \vee q) \wedge (\neg q \vee p)$
- We can give a name for each wire that is an output from a gate
- For an and-gate with inputs x_1 and x_2 we can call the output x_3
- The idea is that x_3 represents the value of the gate so only one "copy of the gate/formula" is needed
- $x_1 \wedge x_2 \leftrightarrow x_3$ for an and-gate
- $x_1 \vee x_2 \leftrightarrow x_3$ for an or-gate
- $\neg x_1 \leftrightarrow x_2$ for a not-gate
- These new variables that can be used in multiple expressions

The formula and an example

- For each gate in the circuit we make an equivalence operator
- Then all equivalence expressions must be true so there is one *and* with each as input
- In addition, the output should also be true so it is also an input to this *and*
- Say we have three inputs x_1, x_2, x_3, an and-gate with x_1 and x_2, the output of it and x_3 input to an or-gate and the output of that input to a not-gate.
- If the output is x_4 we have: $x_4 = \neg((x_1 \land x_2) \lor x_3)$
- Three equivalences:
  - $x_1 \land x_2 \leftrightarrow x_5$
  - $x_5 \lor x_3 \leftrightarrow x_6$
  - $\neg x_6 \leftrightarrow x_4$
- The formula: $x_4 \land (x_1 \land x_2 \leftrightarrow x_5) \land (x_5 \lor x_3 \leftrightarrow x_6) \land (\neg x_6 \leftrightarrow x_4)$
- Note we can create this formula in polynomial time

The formula and an example

- The formula: $x_4 \wedge (x_1 \wedge x_2 \leftrightarrow x_5) \wedge (x_5 \vee x_3 \leftrightarrow x_6) \wedge (\neg x_6 \leftrightarrow x_4)$
- Why are the circuit and the formula equivalent?
- Given values of the inputs x_1, x_2, x_3 in the circuit that lead to a one as output, i.e. to x_4, will the formula also be true?
- Yes, because the new variables have the same values as the wires of the circuit that resulted in the output one
- An example satisfying input to the circuit is $x_1 = 0, x_2 = 1, x_3 = 0$
- Using the same input in the formula, we will have $x_5 = 0$, $x_6 = 0$ and $x_4 = 1$ just as in the circuit
- With a satisfying input to the formula, the circuit will also have output one
- The next step is to make this a conjunction of clauses
- Instead of "moving" $\neg$ and $\vee$ down, we can write it in the desired form almost directly.

Recall distributive laws

- $p \vee (q \wedge r)$ can be written $(p \vee q) \wedge (p \vee r)$
- $p \wedge (q \vee r)$ can be written $(p \wedge q) \vee (p \wedge r)$

From and-gate to formula

- $x_1 \wedge x_2 \leftrightarrow x_3$
- Recall $p \to q$ means $\neg p \vee q$
- So $p \leftrightarrow q$ means $(\neg p \vee q) \wedge (\neg q \vee p)$
- Here p is $x_1 \wedge x_2$ and q is x_3
- So $x_1 \wedge x_2 \leftrightarrow x_3$ can be written $(\neg(x_1 \wedge x_2) \vee x_3) \wedge (\neg x_3 \vee (x_1 \wedge x_2))$
- DeMorgan's laws: $((\neg x_1 \vee \neg x_2) \vee x_3) \wedge (\neg x_3 \vee (x_1 \wedge x_2))$
- Simplified: $(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_3 \vee (x_1 \wedge x_2))$
- Distributive law: $(\neg x_1 \vee \neg x_2 \vee x_3) \wedge ((\neg x_3 \vee x_1) \wedge (\neg x_3 \vee x_2))$
- Simplified: $(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_1) \wedge (\neg x_3 \vee x_2)$
- So the and-gate can be translated to a conjunction of three clauses

From or-gate to formula

- $x_1 \lor x_2 \leftrightarrow x_3$
- Again: $p \leftrightarrow q$ means $(\neg p \lor q) \land (\neg q \lor p)$
- So $x_1 \lor x_2 \leftrightarrow x_3$ can be written $(\neg(x_1 \lor x_2) \lor x_3) \land (\neg x_3 \lor (x_1 \lor x_2))$
- Simplified: $(\neg(x_1 \lor x_2) \lor x_3) \land (\neg x_3 \lor x_1 \lor x_2)$
- DeMorgan's law: $((\neg x_1 \land \neg x_2) \lor x_3) \land (\neg x_3 \lor x_1 \lor x_2)$
- Distributive law: $((\neg x_1 \lor x_3) \land (\neg x_2 \lor x_3)) \land (\neg x_3 \lor x_1 \lor x_2)$
- Simplified: $(\neg x_1 \lor x_3) \land (\neg x_2 \lor x_3) \land (\neg x_3 \lor x_1 \lor x_2)$
- So the or-gate can also be translated to a conjunction of three clauses

From not-gate to formula

- $\neg x_1 \leftrightarrow x_2$
- Again: $p \leftrightarrow q$ means $(\neg p \vee q) \wedge (\neg q \vee p)$
- So $\neg x_1 \leftrightarrow x_2$ can be written $(\neg\neg x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_1)$
- Simplified: $(x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_1)$
- The last is true (as expected) when either $x_1 = 0$ and $x_2 = 1$, or when $x_1 = 1$ and $x_2 = 0$
- So the not-gate can be translated to a conjunction of two clauses

Conjunction of clauses

- We started with: $x_4 \wedge (x_1 \wedge x_2 \leftrightarrow x_5) \wedge (x_5 \vee x_3 \leftrightarrow x_6) \wedge (\neg x_6 \leftrightarrow x_4)$
- Since each gate can be translated to a conjunction of clauses, we can make one big conjunction:

  x_4
  $\wedge (\neg x_1 \vee \neg x_2 \vee x_5)$
  $\wedge (\neg x_5 \vee x_1)$
  $\wedge (\neg x_5 \vee x_2)$
  $\wedge (\neg x_5 \vee x_6)$
  $\wedge (\neg x_3 \vee x_6)$
  $\wedge (\neg x_6 \vee x_5 \vee x_3)$
  $\wedge (x_6 \vee x_4)$
  $\wedge (\neg x_4 \vee \neg x_6)$
- Above is more complicated than $x_4 = \neg((x_1 \wedge x_2) \vee x_3)$ but always possible to create in polynomial time plus conjunction of clauses

3-Satisfiability

- 3-Satisfiability (3-SAT) is a problem very similar to Satisfiability (SAT)
- A clause in 3-SAT always contains three terms, e.g. $x_1 \lor \overline{x_3} \lor x_4$
- It is easy to translate a SAT instance into a 3-SAT instance, i.e. reducing SAT to 3-SAT
- 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

- 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.

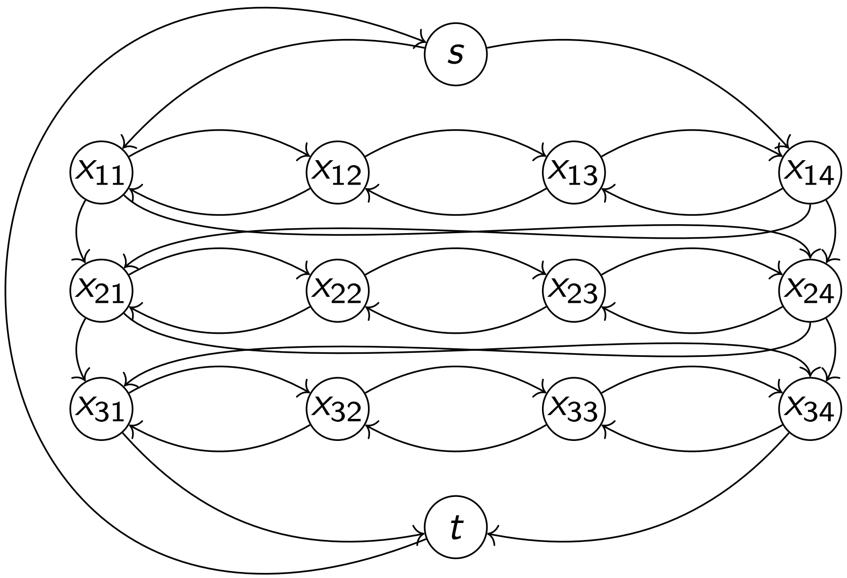
The Hamiltonian Cycle Problem

- 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

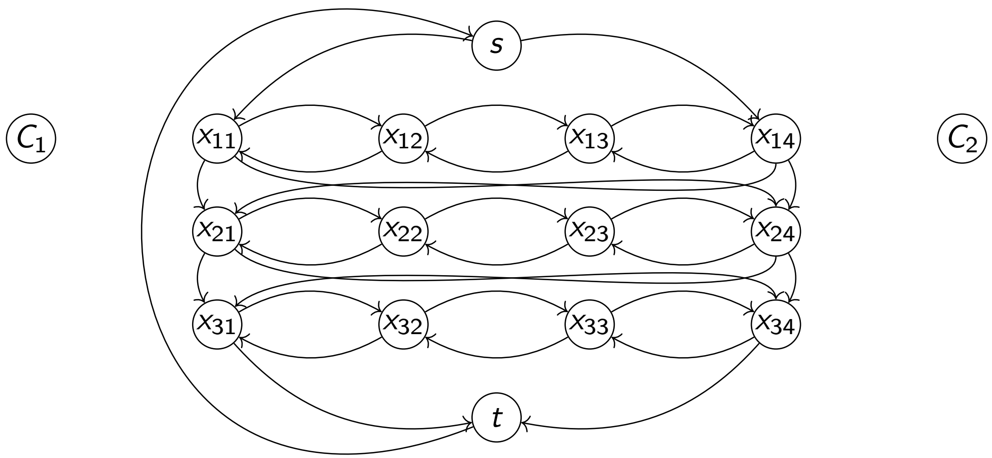
- Assume we have n variables x_i and k clauses C_j in the 3-SAT instance
- $\Phi = C_1 \wedge C_2 \wedge \ldots C_k$
- $C_j = t_{j1} \vee t_{j2} \vee t_{j3}$
- Each t is a term, or literal, which is either a variable or the negation of a variable
- For example: $\Phi = (x_1 \vee x_2 \vee \overline{x_3})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

The Hamiltonian Cycle Problem



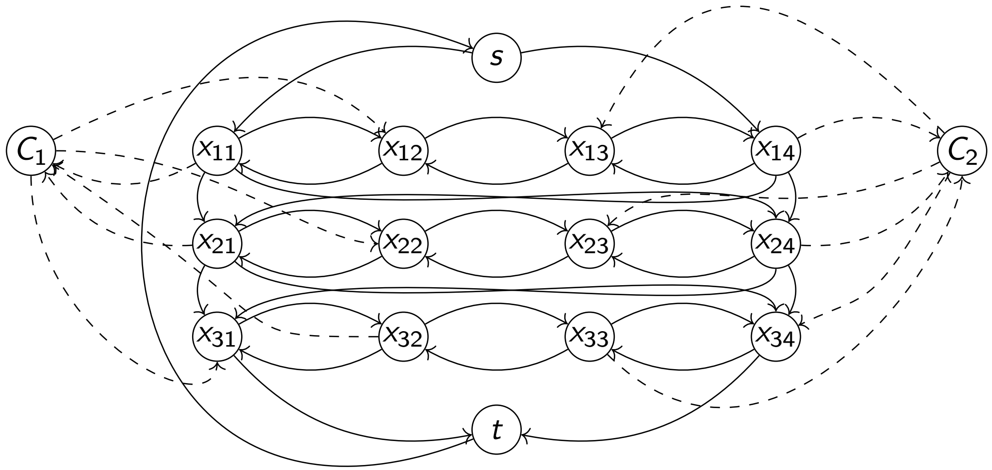
- $\Phi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$
- There is one "row" in the graph for each 3-SAT input variable x_i
- Every Hamiltonian cycle must go from s to either x_{11} or x_{14}
- A row can be passed either in left or right direction
- As the graph 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



- $\Phi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$
- 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



- $\Phi = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$
- 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

The Traveling Salesperson Problem (TSP)

- Another problem in which a sequence of all nodes of a graph is requested is the Traveling Salesperson 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 $d(u, v)$
- A **tour** visits all cities and returns to the originating city
- The Traveling Salesperson 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 Salesperson Problem

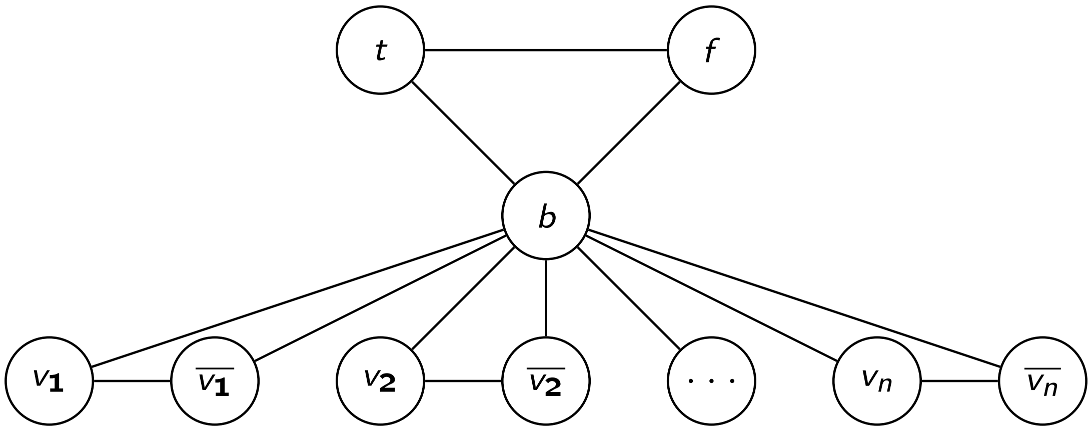
- Given a directed graph $G(V, E)$ for the Hamilton Cycle problem, we construct an instance of TSP as follows
- For each each $(u, v) \in E$ we assign a distance $d(u, v) = 1$ and for all pairs such that $(u, v) \notin E$ we assign a distance $d(u, v) = 2$
- If and only if there is a solution to TSP 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

The Graph Coloring Problem

- 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

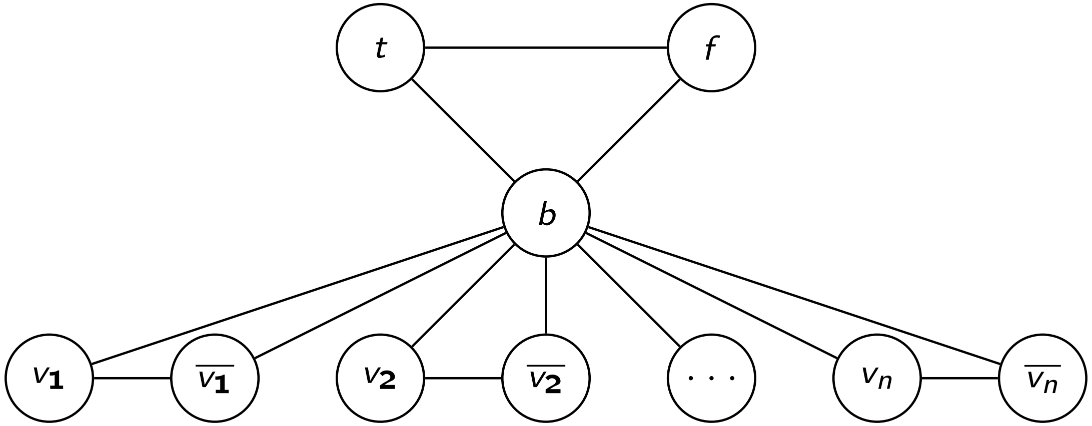
Reduction from 3-SAT to 3-coloring

- 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



- 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



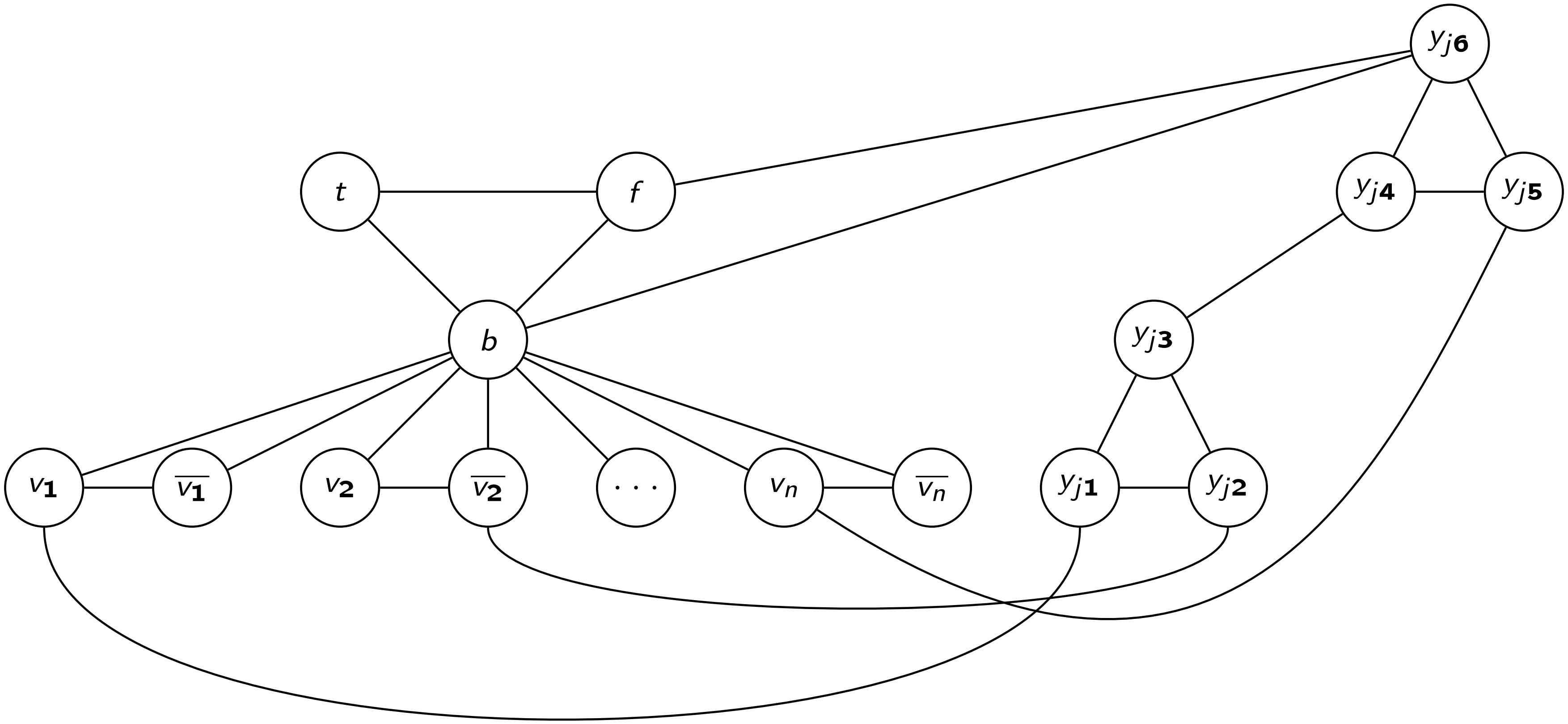
- Since each v_i and $\overline{v_i}$ is a neighbor of b, a 3-coloring must select the color of either t or f for them
- We will denote the color of t by T, the color of f by F and the color of b by B below

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 $C_j = x_1 \vee \overline{x_2} \vee x_n$, 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

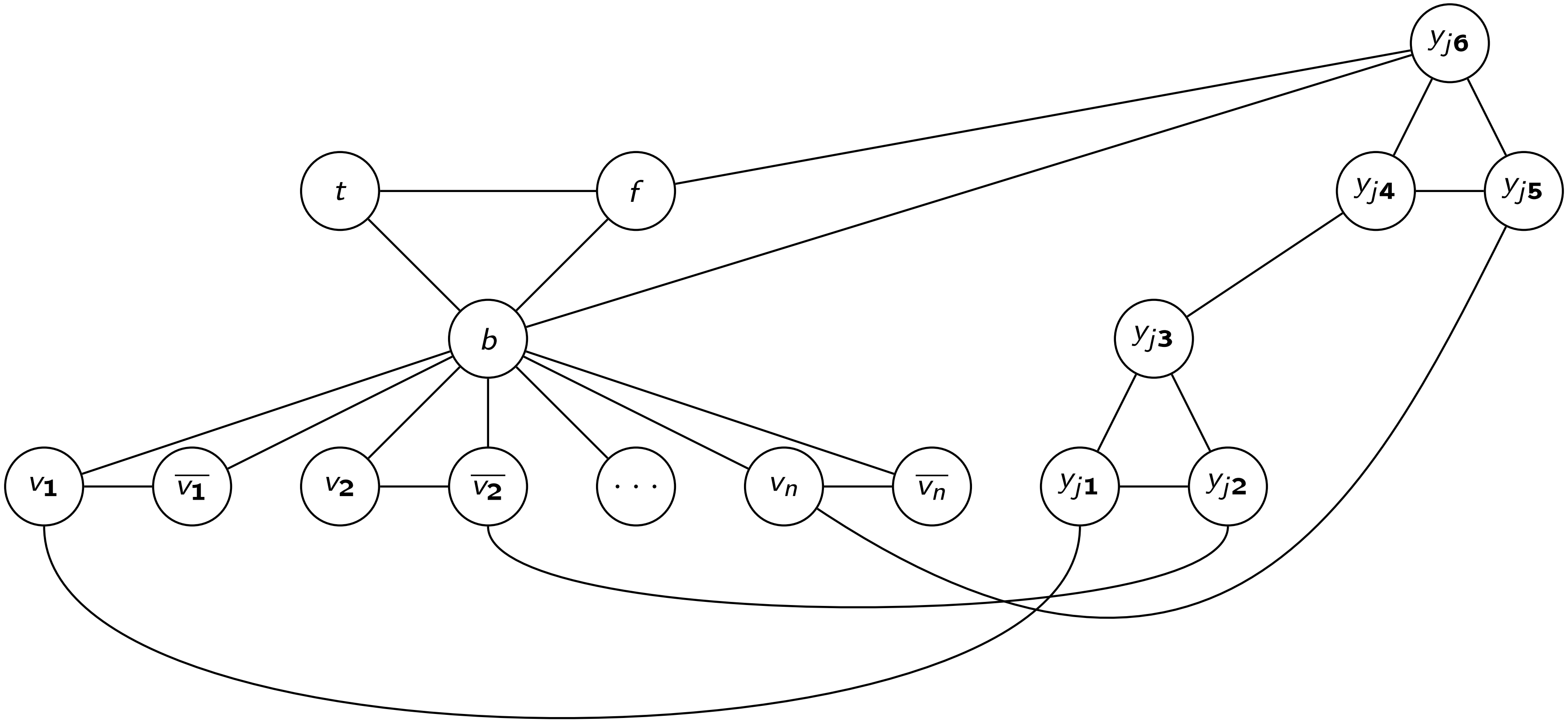
- Assume $C_j = x_1 \vee \overline{x_2} \vee x_n$, and $p_j = v_1$, $q_j = \overline{v_2}$, and $r_j = v_n$



- 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 $(x_1, x_2, x_3) = (0, 1, 0)$



SAT solving

- No efficient algorithm for SAT solving is know in the general case
- In practice, there are numerous SAT instances that can be solved even with millions of variables
- We have a set F of clauses in CNF form, using a set V of variables and $|V| = n$.
- A variable is **free** when it has not been assigned a value yet
- In an **assignment** no variable is free
- In a **partial assignment** some variables are free
- It can be possible to satisfy F with a partial assignment: in $C = x_1 \vee x_2 \vee \overline{x_4}$ is satisfied if either $x_1 = 1$, $x_2 = 1$ or $x_4 = 0$
- It is too slow to enumerate and check all 2^n possible assignments

SAT solving with backtracking

- Much better than enumerating all assignments

function *basic_sat*(F)
begin
    if any clause C in F cannot be satisfied **then**
        /* all variables in C are assigned a value and all literals in C are 0 */
        return 0
    else if all clauses in F are satisfied **then**
        /* every clause contains a literal with value 1 */
        return 1
    select a variable x_j marked as free
    if *basic_sat*$(F$ with $x_j = 0)$ **then**
        return 1
    else
        $s \leftarrow$ *basic_sat*$(F$ with $x_j = 1)$
        mark x_j as free
        return s
end

Unit propagation

- Try to discover early that a partial assignment cannot satisfy F
- The algorithm then can try a different partial assignment
- A **unit clause** is a clause with only one literal that has a free variable
- Assume we have $C = x_1 \vee x_2 \vee \overline{x_4}$
- And partial assignments: $x_1 = 0$ and $x_2 = 0$ have been made.
- Next try $x_4 = 0$
- This approach is called **unit propagation**.
- It is trivial to add it in the select step in the basic SAT solver
- When we check if the partial assignment either satisfies F or cannot satisfy F, we can also collect candidate variables that may be used in unit propagation
- More about SAT: `https://jakobnordstrom.github.io`