8. NP-completeness

So far: We have considered a wide range of problems that can be solved efficiently, i.e. where algorithm exists

- That solve the problem and
- Whose worst-case running time is polynomial

Motivation:
- What about problems that seem "difficult" to solve?
- In case we cannot come up with an efficient algorithm that is able to solve the problem in polynomial time, does this imply that no such algorithm exists?
- Moreover, is it possible to mathematically prove that no efficient algorithm for solving those problems exists?

Note: If we do not happen to know any efficient algorithm to solve a given "difficult" problem, this does not imply that no such algorithm exists.

Question: Given the above answer, is there any way to categorize "difficult" problems into different classes according to how difficult they are?

Answer: Yes (at least to some extent)

Definition: We can define the class of NP-complete problems as the set of those "difficult" problems for which

- there currently does not exist an efficient, i.e. polynomial-time algorithm to solve any of them,
- but which are of similar difficulty in the sense that if we know an efficient algorithm to solve one of those problems, this would allow us to derive efficient algorithms for all of those problems.

Remark: You can think of NP-complete problems as being of similar difficulty.

8.1 Polynomial time reductions

Goal: We want to rank computationally hard problems (i.e. those for which no efficient algorithm is known) via pairwise comparisons, e.g. "problem X is at least as hard as problem Y"

(main idea behind this reduction approach)

Strategy: In order to apply the above reasoning, we have to ensure that problems X and Y have a closely related degree of hardness. We thus make the following definition:

Definition: Given two problems X and Y, we say that problem Y is (polynomial-time) reducible to problem X and write "$Y \leq_p X$" if arbitrary instances of problem Y can be solved using a polynomial number of standard computational steps plus a polynomial number of calls to the algorithm that solves problem X.

We also say problem X is at least as hard as problem Y with respect to polynomial time.

Theorem 1: Given two problems X and Y such that $Y \leq_p X$. If X can be solved in polynomial time, than Y can be solved in polynomial time.

Proof: According to the above definition of $Y \leq_p X$, problem Y can be solved using a polynomial number of steps and a polynomial number of calls to the algorithm that solves X. So, if the algorithm to solve X is polynomial, we can therefore solve Y in polynomial time.

Strategy: Typically, we want to use $Y \leq_p X$ in order to prove properties of X given properties of Y (and not the other way around as in Theorem 1). More specifically, we often want to show that

Theorem 2: Given two problems X and Y such that $Y \overset{\longrightarrow}{\leq_p} X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.

($\longrightarrow$ indicates the direction of the logical flow)

Proof:

Group work (hint: how is Theorem 2 related to Theorem 1)

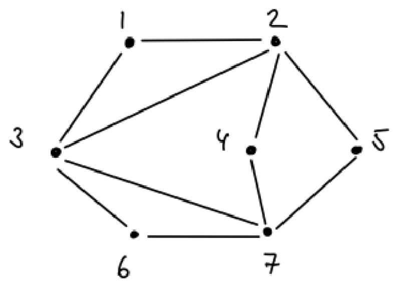
Answer: Theorem 2 is the contrapositive of Theorem 1, i.e. the two statements are equivalent. $\square$.

Example 1: Independent set and vertex cover

Problem 1: (independent set) Given an undirected graph G and a number $k \in N$, does G contain an independent set of size k or larger? Typically, we are interested in large independent sets rather than small ones.

Definition: Given and undirected graph $G = (V, E)$, we call a set S of nodes, $S \subseteq V$, independent if there is no pair of nodes in S that are connected by an edge.

Group work:



aim: find large independent set

Task: Find two independent sets in the above graph.

Answer: $S_1 = \{3, 4, 5\}$ is an independent set of size $k = |S_1| = 3$, because there is no edge beween 3 and $4, 3$ and 5, 4 and 5.

$S_2 = \{1, 4, 5, 6\}$ is another independent set of size $k = |S_2| = 4$ because there is no edge between 1 and 4, 1 and 5, 1 and 6, 4 and 6, 5 and 6.

Reminder: (see section 2.6.4) For a graph with n nodes, it requires $O(n^k)$ time to check if the graph contains an independent set of size $k \in N$.

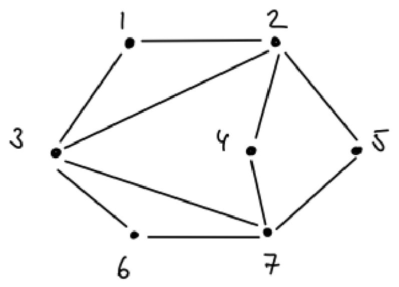
Problem 2: (vertex cover) Given an undirected graph $G = (V, E)$ and a number $k \in N$, does G contain a vertex cover of size $\leq k$?

Definition: Given an undirected graph $G = (V, E)$. A sub-set S of nodes, $S \subseteq V$ is a vectex cover if every edge $e \in E$ has at least one of its nodes in S

(vertex = node)

Remark: It is easy to find large vertex covers ($S = V$ will alsways do), but potentially difficult fto find small vertex covers. So, we are usually interested in finding small vertex covers rather than large ones.

Group work:



aim: find small vertex cover

Task: Find vertex cover fot the above graph

Answer: $S_1 = \{1, 2, 6, 7\}$ of size $k = |S_1| = 4$ is a vertex cover as any edge in G has one of its nodes in S_1.

Question: We know that problem 1 requires $O(n^k)$ time to be solved (which is exponential in k). For both problems, no polynomial time algorithm currenly exists. However,

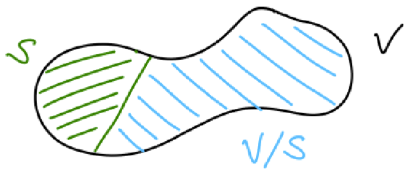
- What can we say about the relative difficulty of both problems?

Strategy: In the folloring, we will show that both problems are equally hard. In order to prove this, we will show that
(problem 1) $\leq_p$ (problem 2) and that
(problem 2) $\leq_p$ (problem 1).

Theorem: Given an undirected graph $G = (V, E)$ and a sub-set $S \subseteq V$. Then the following equivalence holds:

S is independent $\Leftrightarrow$ V/S is a vertex cover.

Reminder: If $S \subseteq V$, then $V/S := \{v \in V | v \notin S\}$, i.e. the set of elements in V which are not in set S.



Proof (of theorem):

"⇒" Suppose S is an independent set for graph G. consider an arbitrary edge $e = (u, v) \in E$. Since S is independent, u and v cannot both be part of S (Why?). This implies that either u or v (or both) are in V/S. As edge $e \in R$ was arbitrarily chosen, we can thus conclude that V/S is a vertex cover as all edges in E are covered by at least one node in V/S.

"⇐" Suppose that V/S is a vertex cover. Consider any two nodes in S, say u and v. If graph G would contain an edge $e = (u, v)$, neither node would be in V/S and V/S could thus not be a vertex cover. This would constitute a contradiction. It thus follows that any two nodes in S are not connected by an edge in G and that S is therefore an independent set.

Lemma 1: (Independent set) $\leq_p$ (vertex cover)

Proof: This follows immidiately from the previous theorem and the definition of $\leq_p$ as the independent set problem can be solved by using the algorithm to solve the vertex cover problem (i.e. to derive V/S as the vertex cover), i.e. by calling it exactly once. The previous theorem then allows us to convert V/S into the desired set S which solves the independent set problem in linear, i.e. in particular polynomial, time.

Lemma 2: (vertex cover) $\leq_p$ (independent set)

Proof: Analogous to the proof of lemma 1, i.e. simply reverse the roles of the two problems.

Example 2: vertex cover to set cover

Motivation: Replace previous problem 1 (independent set) by the more general problem of determining a so-called set cover.

Definition: Givan a set U of n elements, a collection of sub-sets $S_1, ..., S_m$ of U and a number $k \in N$.

A set cover of U of size $k \in N$ is a sub-set of k of the subsets $S_1, ..., S_m$ of U such that their union is equal to U.

Note:

- the sub-sets of the set cover may overlap each other, i.e. may have non-empty intersection
- the elements in set U need to be numbers.

Example:

- $U = N_{10} = \{1, 2, ..., 10\}$
- $S_1 = \{1, 2\}$
- $S_2 = \{3, 4, 5\}$
- $S_3 = \{6, 7, 8\}$
- $S_4 = \{9, 10\}$
- $S_5 = \{4, 7\}$
- $S_6 = \{5, 8\}$
- $S_7 = \{1, 3, 6, 9\}$
- $S_8 = \{2, 3, 4, 5\}$
- $S_9 = \{6, 7, 8, 10\}$

Group work: Is there a set cover of size 3?

Answer: Yes. Take S_7, S_8 and S_9

Problem 1: **(set cover)** Given a set U of n elements and a collection of sub-sets of $U, S_1, S_2, ..., S_m$, does a set cover of size $\leq k, k \in N$, exist?

Note: Typically, the goal is to find a set cover of minimal size.

Problem 2: vertex cover (see example 1)

Theorem: (vertex cover) $\leq_p$ (set cover)

Concern: The vertex cover problem deals with an undirected graph $G = (V, E)$, whereas the set cover problem deals (more generally) with a set of n elements (which could mean anything).

Does the theorem compare apples and branches?

Yes, it does (to some extent), but this is ok, provided it makes sense in the context of the definition of what $X \leq_p Y$ means.

We will see that this is case in the following...

Proof (of the theorem):

(Def.) Reminder: In order to show that $X \leq_p Y$, we need to show that

(1) arbitrary (!) instances of Y (not X!) can be solved by
(2) using a polynomial number of standard computational steps and by
(3) making a polynomial number of calls to the algorithm that solves X (not Y).

In this case, i.e. for Y = (vertex cover) and X = (set cover), suppose we have an algorithm to solve (set cover). Let us consider an arbitrary (!) instance of (vertex cover), specified by graph $G = (V, E)$ and a number $k \in N$.

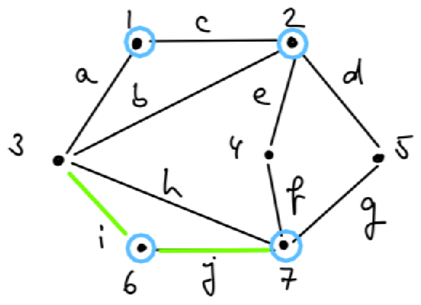
key step in proof (getting the mapping right) The goal in (vertex cover) is to find a sub-set of V that covers the edges E in G. In order to establish a meaningful mapping to a corresponding instance of (set cover), we pick a specific (!) instance of (set cover) and identify set U with set E, i.e. the edges in graph G of (vertex cover).

Whenever we pick a vertex (i.e. node) in (vertex cover), we cover all the edges in G incident on it. In (set cover), this corresponds to adding subset S_i containing all edges in G that are incident to node i to the emerging set cover.

Claim: U can be covered with a set cover of size $\leq k \Leftrightarrow$ the corresponding graph G has a vertex cover of size $\leq k$

Need to consider an arbitrary (!) graph $G = (V, E)$

Example:



- set $U = E = \{a, ..., j\}$.
- take vertex cover $S \subseteq V$
- for each node $v \in S$, define corresponding sub-set S_v of U that contains edges incident on v
  e.g. $v = 6 \Rightarrow S_6 = \{i, j\}$

vertex cover of size 4: $\{1, 2, 6, 7\}$

Proof of claim:

"$\Rightarrow$": Denote $S_{i_1}, S_{i_2}, \ldots, S_{i_l}$ the $l \leq k$ sub-sets of U that correspond to the set cover of U.

This means that all edges of graph G are covered, i.e. that every edge $e = (u, v)$ in G is incident to one of the vertices in sub-sets $S_{i,1}, \ldots S_{i,l}$. This implies that $i_1, ..., i_l, l \leq k$, is a vertex cover of G of size $l \leq k$.

"$\Leftarrow$": Conversely, if $\{i_1, \ldots, i_l\}, l \leq k$, is a vertex cover of G, the corresponding sets $S_{i_1}, \ldots, S_{i_l}$, are a set cover of size $l \leq k$ of U.

8.2 Reductions via so-called "gadgets": the satisfiability problem

Motivation Introduce more abstract kinds of problems to be used for $\leq_p$-comparisons to other problems.

Example 1: The satisfiability problem or SAT problem

Definitions • $x_i \in \{0, 1\}$, a Boolean variable, where $x_i = 1 =$ "true" and $x_i = 0 =$ "false"

- $X = \{x_1, x_2, \ldots, x_n\}$ a set of Boolean variables
- term over X = one $x_i \in X$ or its negation, i.e. $\bar{x}_i$
- a clause of length $l = t_1 \vee t_2 \vee \ldots \vee t_l$, where sign "$\vee$" means "OR" and where $t_i \in \{x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n\}$
- a truth assignment $\mathcal{V}$ = a function $\mathcal{V} : X \rightarrow \{0, 1\}$, where $\mathcal{V}(\bar{x_i}) = v(\bar{x}_i)$
- $\mathcal{V}$ satisfies a clause $C = t_1 \vee \ldots \vee t_l$ if $\mathcal{V}$ causes C to evaluate to 1 ("true"), i.e. where

$$\mathcal{V}(C) = \mathcal{V}(t_1) \vee \ldots\ldots \mathcal{V}(t_l) = 1$$

- $\mathcal{V}$ satisfies clauses C_1 to C_k if $\mathcal{V}$ causes *every* clause C_i, $i \in \{1, \ldots, k\}$, to evaluate to 1, i.e. where

$$\mathcal{V}(C_1) \wedge \ldots \wedge (C_k) = 1$$

  and where sign "$\wedge$" means "and".

  We then say that "$\mathcal{V}$ satisfies an assignment w.r.t. C_1 to C_k" or we say that "the set of clauses C_1 to C_k is satisfying under assignment $\mathcal{V}$"

Special Cases

- truth assignment $\mathcal{V} : X \rightarrow \{1\}$, i.e. this assignment sets all variables in X to 1, i.e. "true"
- truth assignment $\mathcal{V}' : X \rightarrow \{0\}$, i.e. this assignment sets all variables in X to 0, i.e. "false".

Example

$$\begin{aligned} C_1 &= (x_1 \vee \bar{x}_2) \\ C_2 &= (\bar{x}_1 \vee \bar{x}_3) \\ C_3 &= (x_2 \vee \bar{x}_3) \end{aligned}$$

Question • Can we find a truth assignment $\mathcal{V}$ that satisfies C_1, C_2 and C_3?

Group Work

1. Does the truth assignment $\mathcal{V} : X \rightarrow \{1\}$ satisfy C_1 to C_3
2. What about truth assignment $\mathcal{V}' : X \rightarrow \{0\}$?

Answer 1.

$$\begin{aligned} \mathcal{V}(C_1) &= (1 \vee \bar{1}) = (1 \vee 0) = 1 && \surd \\ \mathcal{V}(C_2) &= (\bar{1} \vee \bar{1}) = (0 \vee 0) = 0 && (no!) \\ \mathcal{V}(C_3) &= (1 \vee \bar{1}) = (1 \vee 0) = 1 && \surd \end{aligned}$$

$\Rightarrow$ no, $\mathcal{V}$ doesn't satisfy C_1 to C_3, because of C_2

2.
$$\begin{aligned}
\mathcal{V}'(C_1) &= (0 \vee \bar{0}) = (0 \vee 1) = 1 && \surd \\
\mathcal{V}'(C_2) &= (\bar{0} \vee \bar{0}) = (1 \vee 1) = 1 && \surd \\
\mathcal{V}'(C_3) &= (0 \vee \bar{0}) = (0 \vee 1) = 1 && \surd
\end{aligned}$$

$\Rightarrow$ yes, $\mathcal{V}'$ satisfies C_1 to C_3

Goal in the following (Definition: Satisfiability problem or SAT)

Given a finite set of clauses C_1 to C_k over a finite set X of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$, is it possible to find a truth assignment that satisfies C_1 to C_k?

Special Case (Definition: 3-Satisfiability problem or 3-SAT) Given a finite set of clauses C_1 to C_k *of length 3 each* over a set of Boolean variables $X = \{x_1, x_2, \ldots, x_n\}$ is it possible to find a truth assignment that satisfies C_1 to C_k?

Comment The 3-SAT problem can be shown to be of equivalent difficulty as the more general SAT problem.

Key Observation (Why are 3-SAT and SAT computationally hard to solve?)

- for a set X of n Boolean variables, we have to make only n 0/1 decisions when defining any possible truth assignment $\mathcal{V} : X \to \{0, 1\}$
- however, there are typically several possible ways of satisfying each of the k constraints C_i in isolation and
- we have to arrange our decisions so *all* k constraints C_1 to C_k are *simultaneously* satisfied

Goal We want to show that

(3-SAT) $\leq_p$ (independent set) $\qquad (*)$

Concern Any instance of (3-SAT) deals with Boolean variables for a given set of k constraints C_1 to C_k, whereas any instance (independent set) deals with an undirected $G = (V, E)$.

When testing if the above statement $(*)$ holds, do we need to compare apples to bananas?

Response Yes, at least to some extent, but this is ok and correct, as long as we can find a valid mapping between any arbitrary (!) instance of (3-SAT) and a corresponding, i.e. particular (!) instance of (independent set).

In order to prove statement $(*)$, we need to refer back to the definition of "$\leq_p$".

One example on how to successfully establish a valid mapping between "apples and bananas" was shown when proving that

(vertex cover) $\leq_p$ (set cover) $\qquad$ (see 8.1)

Reminder In order to show that (3-SAT) $\leq_p$ (independent set), we need to (see definition of "$\leq_p$") show that

1. an *arbitrary* instance of (3-SAT) can be solved by
2. using a polynomial number of standard computational steps and by
3. making a polynomial number of calls to the algorithm that solves the corresponding instance of (independent set)

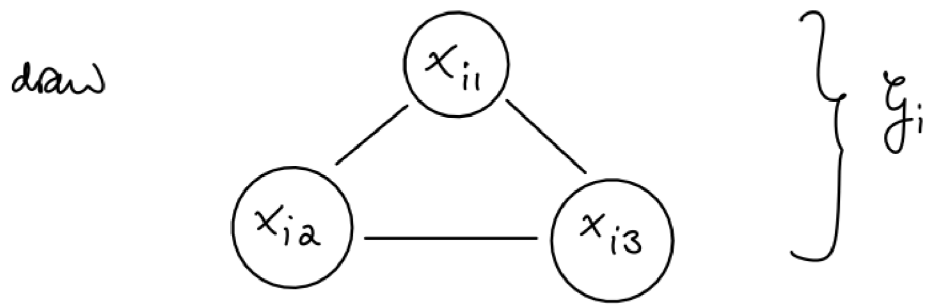
Challenges in this case (A) Assuming that we have an algorithm for solving (independent set) which operates on a graph $G = (V, E)$ and its nodes and edges, find a way to encode the Boolean constraints of any *arbitrary* instance of (3-SAT) in nodes and edges of a corresponding (i.e. particular) graph, i.e. a particular instance of (independent set)

(B) The encoding or mapping from (*) must be done *in such a way* that satisfiability of (3-SAT) corresponds to the existence of an independent set in the corresponding graph.

Theorem (3-SAT) $\leq_p$ (independent set)

Ideas for proof: Rules for converting 3-SAT problem into a corresponding graph G

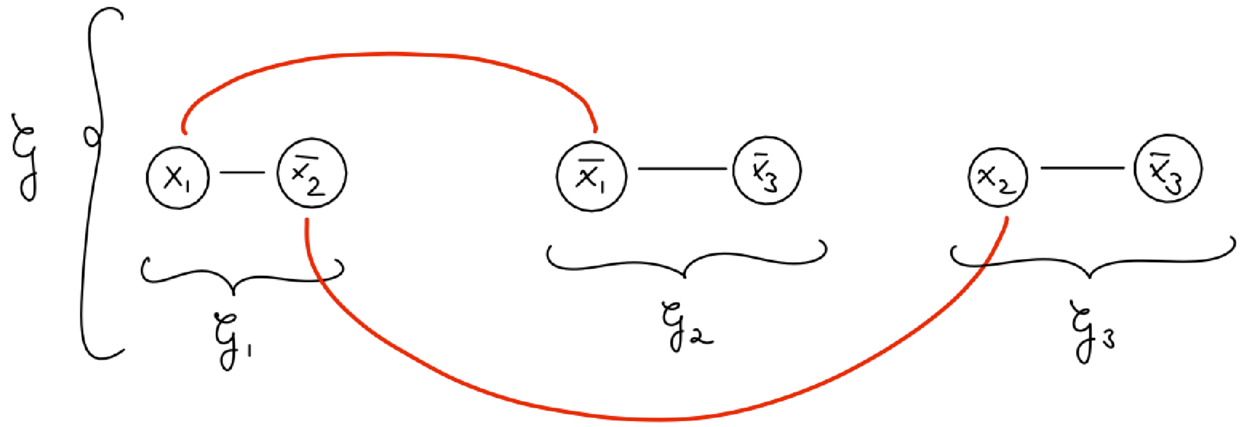
- for every conflict C_i of an arbitrary instance of (3-SAT), draw a corresponding fully connected sub-graph $G_i = (V_i, E_i)$ which consists of 3 notes that each correspond to the three Boolean variables in conflict C_i
  $\Rightarrow$ for $C_i = (x_{i1} \vee x_{i2} \vee x_{i3})$



- for all k conflicts C_1 to C_k combine their corresponding sub-graphs G_i into a large graph $G = (V, E)$, where
  $V = V_1 \cup V_2 \cup \ldots V_k =$ i.e., the set of nodes in G is the union of nodes from all k subgraphs G_i
  $E = E_1 \cup E_2 \cup \ldots E_k \cup \tilde{E} =$ i.e., the edges in G is the union of edges from all k subgraphs G_i and a new set of edges $\tilde{E}$
- set $\tilde{E}$ comprises an edge for any pair of nodes that (a) belong to two different sub-graphs G_i and G_j, $i \neq j$, **and** (b) correspond to Boolean variables that are *not simultaneously satisfiable* by *any* truth assignment $\mathcal{V}$

Example (revisit previous example which is 2-SAT, really, but which gives you an idea of how to do this for any 3-SAT problem as well)

$$C_1 = (x_1 \vee \bar{x}_2), C_2 = (\bar{x}_1 \vee \bar{x}_3), C_3 = (x_2 \vee \bar{x}_3)$$



- the red edges are those in set $\tilde{E}$ and are called conflict edges

Group Work Why do we add edges between all nodes corresponding to the same clause C_i?

Answer These edges ensure that any independent set derived for graph G comprises at most *one* of the Boolean variables in C_i to evaluate to "true" under truth assignment $\mathcal{V}$.

These edges therefore prevent us from considering independent sets of size k where part of the size k is due to several nodes in a sub-graph G_i begin selected.

By making the sub-graphs G_i fully connected we ensure that any independent set of size k for graph G corresponds automatically to satisfying all k constraints separately, as we then know that *exactly* one node from each of the k sub-graphs G_i, is included in the independent set of G

Group Work By making the graph G as described above, can we find independent sets of size $> k$ when dealing with k clauses C_i to C_k?

Answer No, we can only find independent sets of size $\leq k$ because every sub-graph G_i is fully connected and we can thus select only 0 or 1 node from each G_i in any independent set.

!!! The case of |ind. set| $= k$ thus corresponds to the satisfiability of clauses C_1 to C_k.

Proof of Theorem (3-sat) $\leq_p$ (ind. set)

We assume that we have an algorithm for solving (independent set) and want to solve an arbitrary instance of (3-SAT), where $X = \{x_1, x_2, \ldots, x_n\}$ is the set of Boolean variables and where we have k clauses C_1 to C_k. In order to convert this arbitrary instance of (3-SAT) into a corresponding graph G, we apply the rules that we just introduced above.

We know claim that:

a given 3-SAT problem with k clauses is satisfiable $\leftrightarrow$ the corresponding graph G has an independent set of size k

Proof "$\Rightarrow$"

If 3-SAT is satisfiable, then every sub-graph G_i contains exactly one chosen node (and not more, as the sub-graph is fully connected thereby ensuring that no pair of nodes in the sub-graph can ever be part of the same independent set).

(Why not fewer? This would imply that one sub-graph G_i has no node in the independent set which would contradict the fact that the 3-SAT problem is satisfiable)

We thus have an independent set of size k in graph G

Proof "$\Leftarrow$"

We know that graph G has an independent set of size k. Given the way that graph G was constructed, we can conclude that each sub-graph G_i has exactly one node x_i in the independent set.

In order to find a truth assignment $\mathcal{V}$ that satisfies all clauses C_1 to C_k, we assign:

$$\mathcal{V}(X_i) = \begin{cases} 1 & \text{if } x_i \in \text{ (independent set)} \\ 0 & \text{else} \end{cases}$$

This truth assignment $\mathcal{V}$ satisfies clauses C_1 to C_k as $\mathcal{V}(X_i) = 1$ for one Boolean variable X_i in every C_i

Remember: $\mathcal{V}(C_i) = 1$ if $v(x_i) = 1$ for at least one x_i in clause C_i □

Lemma (Transitivity of reduction)

If $X \leq_p Y$ and $Y \leq_p Z \Rightarrow X \leq_p Z$

Proof (Omitted here. The strategy employed by the proof is to apply the definition of "$\leq_p$" twice.)

Example As we already know that

1. (3-SAT) $\leq_p$ (independent set)
2. (independent set) $\leq_p$ (vertex cover)
3. (vertex cover $\leq_p$ (set cover)

We can conclude that

(3-SAT) $\leq_p$ (Set cover)

8.3 Efficient certification and the definition of NP

Observation/Motivation For many algorithms that are computationally *hard to solve*, we can fairly *efficiently check* if a proposed solution is correct or not.

Definitions

- s = binary string of finite length that encodes the input to a problem. i.e. describes a particular instance of that problem.
  $|s|$ = length of the above string, i.e. $|S| \in \mathbb{N}_0$
- X = a decision problem which we identify with the set of strings s for which the answer to the decision problem is "yes". (e.g. set of strings describing all possible 3-SAT problems)
- A = algorithm to solve decision problem X, it takes as input a string s and returns as output $A(s)$, i.e. as answer, either "yes" or "no".
- we say that algorithm A solves decision problem X if $A(s) = \text{yes} \Leftrightarrow s \in X$
- A has polynomial running time if there exists a polynomial function $p(n), p : \mathbb{N} \to \mathbb{R}+$, so that for every input string s, algorithm A terminates in at most $O(p(|s|)$ steps.
- P = the set of all problems for which a *known* algorithm exists that solves the problem in polynomial time.

Group Work Given a 3-SAT problem of k clauses C_1 to C_k given n Boolean variables $\tilde{X}$ and a proposed solution, i.e. a truth assignment $\mathcal{V} : \tilde{X} \to \{0, 1\}$ that is supposed to satisfy that 3-SAT problem. Can you think of an efficient algorithm to check if $\mathcal{V}$ satisfies the given 3-SAT problem?

Answer We need to compute $\mathcal{V}(C_1)$ to $\mathcal{V}(C_k)$ and check of $\mathcal{V}(C_i) = 1$ for $\forall i \in \{1, \ldots, k\}$. We know that this requires $O(k)$ time. We thus have an efficient algorithm to check if a proposed solution is correct or not. (We call this kind of algorithm a checking algorithm in the following).

Definitions

- an algorithm B is called an efficient certifier or efficient checking algorithm for a problem X if
  1. B is a polynomial-time algorithm that takes as input a string s (i.e. an input string s describing a particular instance of problem X) and a certificate string t that describes a proposed answer to problem X for input string s, and
  2. there exists a polynomial function $p(n)$, $n \in \mathbb{N}$, so that for every string s, we have the equivalence:
     $s \in X \Leftrightarrow$ there exists a certificate string t with $|t| \leq p(|s|)$ and $B(s, t) = \text{yes}$.
- NP = set of all problems for which an efficient certifier exists.

Example

- X = 3-SAT
- s = a particular 3-SAT problem, e.g. a set of k particular clauses C_1 to C_k given n Boolean variables. e.g. for $k = 3$ and $n = J$ $(x_1 \vee x_2 \vee \bar{x}_5)$, $(x_3 \vee \bar{x}_4 \vee \bar{x}_5)$, $(x_1 \vee x_2 \vee x_3)$

- t = a certificate string describing a potential proposed solution to problem X and input string s i.e. a solution to a practical instance of a 3-SAT problem described by s, e.g.
  $x_1 = 1$, $x_2 = 1$, $x_3 = 1$, $x_4 = 0$, $x_5 = 0$
  For this example $A(s)$ = yes, the particular 3-SAT problem described by s is satisfiable and there exists an efficient certifier B which we can use to show that $B(s,t)$ = yes. In particular, $s \in X$ in the example.

Group Work Suppose we are dealing with X = independent set. Can we devise an efficient certifier B? Suppose you are given an input string s and a corresponding certificate string t, show you B would operate to derive $B(s,t)$ = yes and derive an asymptotic bound for the worst case running time of B

Answer For any proposed answer t to a particular problem of X that is specified by s, the efficient certifier B needs to check that all possible pairs of nodes of the proposed independent set described by string t are not connected by an edge. If the proposed independent set contains m nodes, we need to check $\binom{m}{2}$ different pairs of nodes to see if they are connected by an edge. This requires $O(m^2)$ time, i.e. $X \in$ NP as we have found an efficient certifier B that can check a proposed solution to any instance of X in polynomial time.

Theorem $P \subseteq NP$

Proof We need to show that for any $X \in P \Rightarrow x \in NP$. So, suppose we have a problem $X \in P$, i.e. a problem for which an efficient algorithm A exists to solve it. We need to show that X then also has an efficient certifier B. We can devise an efficient certifier B as follows:

For a given pair of strings (s,t), B simply calls algorithm A and returns $A(s)$, (i.e. we run the known poly-time algorithm A that *solves* X as part of the checking algorithm B) and then checks in polynomial time if solution $A(s)$ (yes/no) corresponds to the solution to a problem s as specified by the certificate string t. Overall, B runs in polynomial time because it consists of two sets that require polynomial time. □

Visually (figure)

$$\boxed{B} = 1 \times \boxed{A} + \text{pol. time for comparing } A(s) \text{ to } t$$

Big Question Is P = NP?

i.e. is there a problem $X \in NP$ which has an efficient certifier but has *no* efficient algorithm to solve it, i.e. is there an $X \in NP$, $X \notin P$?

Answer We don't know. Most people *believe* that $P \subsetneq NP$, i.e. that there are problems $X \in NP$ solutions to which can be efficiently checked, but which cannot be efficiently solved.

Remark If you think you can prove or disprove $P = NP$, you can win 10^6 \$US. To find out more, go to: `www.claymath.org/millennium/P_vs_NP/`

8.4 More about NP-Complete Problems

Motivation As we currently do not know if $P = NP$, we focus instead on finding out what the computationally most difficult problems in NP are.

Definition A problem X is called NP-complete if

1. $X \in NP$, AND
2. $y \leq_p X$ for all $y \in NP$

Comments
- $X \in NP$ does *not* imply that X is NP-complete.
- an NP-complete problem X is a problem that is at least as hard to solve as every other problem $Y] \in NP$

Concern The definition of NP-complete requires us to go through *all* problems $Y \in NP$. Is this feasible?

Theorem *Suppose X is an NP-complete problem.* Then the following equivalence holds.

$$X \in P \Leftrightarrow P = NP$$

Proof We know that X is NP-complete, i.e. we know that $X \in NP$ and $Y \leq_p X$ for all $Y \in NP$ (see definition)

"$\Rightarrow$" If $X \in P$, i.e. if X can be solved in polynomial time, and $Y \leq_p X$ for all $Y \in NP$ (because we know that X is NP-complete), it follows directly that all $Y \in NP$ can be solved in polynomial time. ie. $P = NP$

"$\Leftarrow$" If $P = NP$, every $X \in NP$ is automatically $X \in P$

Conclusion If there is an NP-Complete problem $X \in NP$ that cannot be solved in polynomial time

Warning $X \in NP$ is *not* equivalent to X being NP-complete (see definition of NP-complete)

Theorem If Y is an NP-complete problem $X \in NP$ and $Y \leq_p X$ then X is also NP-complete.

Proof To show that $X \in NP$ is NP-Complete, we need to show that for all $Z \in NP$, $Z \leq_p X$. As Y is known to be NP-complete, we know that $Y \in NP$ and $Z \leq PY$ for all $Z \in NP$. As we already know that $Y \leq_p X$ and because of the transitivity of $\leq_p$ and $Z \leq_p Y$ for all $Z \in NP$, we can thus conclude that $Z \leq_p X$ for all $Z \in NP$. Hence, X is also NP-complete.

Comment It is possible to show that the following problems are NP-complete: (independent set), (3-SAT), (set cover), and (vertex cover).

Idea Behind Proof (example: 3-SAT)

We already know that (3-SAT) $\in NP$. We thus still need to show that $Z \leq_p$ (3-SAT) for all $Z \in NP$. For this, we need to show that $Z \leq_p$ (3-SAT) for all $Z \in NP$ can be mapped to a corresponding instance of a 3-SAT problem in such a way that solving any arbitrary instance of problem Z is equivalent to solving the corresponding, particular instance of the 3-SAT problem.