Complexity Theory

Lecture Notes 2016–2017

Johan Kwisthout
Donders Center for Cognition / Artificial Intelligence
Radboud University Nijmegen
Email: j.kwisthout@donders.ru.nl

last update: August, 2022

Many computational problems in artificial intelligence and computer science are *intractable*: They are difficult to solve given reasonable bounds on the available computation time and memory space available. This intractability is due to the *nature of these problems*; not because we haven't found smart algorithms to solve them yet. The field in theoretical computer science that aims to capture this notion of intractability is *computational complexity theory*. In these notes, that accompany the lecture slides, I will briefly introduce the basic concepts to you. In particular, I will cover:

- Decision problems
- Run-time complexity of algorithms
- Big-Oh notation, worst/best/average case complexity
- The classes P and NP
- Reductions
- NP-completeness

These lecture notes are partially based on the book *Cognition and Intractability: A Guide to Classical and Parametrized Complexity Analysis* currently being written by Iris van Rooij, Johan Kwisthout, Mark Blokpoel, and Todd Wareham; to appear with Cambridge University Press in 2016.

1 Decision problems

We formally define a computational problem D as a mapping from an input I to an output O. We distinguish between three different types of computational problems:

1. Search problems
2. Optimization problems
3. Decision problems

The intuitive meaning of each type may become clear from these examples:

Vertex Degree (optimization and search)
Input: A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$.
Output: A vertex $V \in \mathbf{V}$ such that the degree of V is maximum. (Here the *degree* of a vertex is the number of edges incident to that vertex).

Vertex Degree (search, but not optimization)
Input: A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ and an integer k.
Output: A vertex $V \in \mathbf{V}$ such that the degree of V is at least k if such a vertex exists (otherwise output that none exists).

Vertex Degree (optimization, but not search)
Input: A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$.
Output: A number k, representing the maximum degree of any vertex $V \in \mathbf{V}$.

Vertex Degree (decision)
Input: A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ and an integer k.
Question: Does there exist a vertex $V \in \mathbf{V}$ such that the degree of V is at least k?

The mapping in these four examples is respectively $D : \{\mathbf{G}\} \to \{V\}$, $D : \{\mathbf{G}, k\} \to \{V\}$, $D : \{\mathbf{G}\} \to \{k\}$, and $D : \{\mathbf{G}, k\} \to \{yes, no\}$. In complexity theory we are typically interested in decision problems; problems that pose a question that can be answered with either *yes* or *no*. The reason therefore is that if a decision problem is hard to *decide* (answer *yes* or *no*), the corresponding search and/or optimization problem is also hard to *solve*. The following algorithm decides **Vertex Degree**; it's easily adapted to return the degree of the graph (rather than *yes* or *no*).

Algorithm 1 Decide if the degree of a graph G with n vertices $\geq k$

VertexDegree$(\mathbf{G}, k)$

```
1:  d ← 0                                   # degree variable
2:  for i = 1 to n do
3:     d_i ← degree of vertex V_i
4:     if d_i > d then
5:        d ← d_i                           # d gets updated with new maximum value
6:     end if
7:  end for
8:  if d ≥ k then
9:     return yes                           # Yes, degree is at least k
10: else
11:    return no                            # No, degree is less than k
12: end if
```

As an aside: This particular implementation algorithm could be sped up a bit for some inputs by not keeping track of the current maximum degree, but just returning *yes* if the threshold k is reached, but in this way it is easy to see that if it is hard to decide **Vertex Degree**, it is impossible that is could be easy to actually *solve* the problem. Note that in the worst case the vertex with the maximum degree could be the n-th vertex we look at.

We distinguish between *yes*- and *no*-instances of these decision problems:

Definition 1.1. Let $D : I \to \{yes, no\}$ be a decision problem and let $i \in I$ be an instance of D. We say i is a *yes*-instance if $D(i) = yes$ and a *no*-instance if $D(i) = no$.

The instances I of a decision problem D can be partitioned into *yes*-instances I_{yes} and *no*-instances I_{no}, arbitrary instances in either class are referred to as $i_{yes} \in I_{yes}$, respectively $i_{no} \in I_{no}$. For example, a graph $\mathbf{G}$ where the maximum degree is 4 is a *yes*-instance for $k = 3$ and a *no*-instance for $k = 5$, so $D(\{\mathbf{G}, 3\}) = yes$ and $D(\{\mathbf{G}, 5\}) = no$. Similarly, for another graph $\mathbf{G}'$ with maximum degree 6 we have that $D(\{\mathbf{G}', 3\}) = yes$, $D(\{\mathbf{G}', 5\}) = yes$, and $D(\{\mathbf{G}', 7\}) = no$. We thus have that $\{\mathbf{G}, 3\} \in I_{yes}$, $\{\mathbf{G}', 3\} \in I_{yes}$, $\{\mathbf{G}', 5\} \in I_{yes}$, but $\{\mathbf{G}, 5\} \in I_{no}$ and $\{\mathbf{G}', 7\} \in I_{no}$.

2 Run-time complexity of algorithms

There are many ways to compute the time needed for an algorithm to solve a problem. When comparing or analysing algorithms, we are typically only interested in the *order of growth* of the running time as a function of the input size, and discard of constants, lower-range polynomials etcetera. We will use the so-called Big-Oh notation, $O(.)$, to express the relation between input size and complexity. The $O(.)$ notation is used to express an asymptotic upper bound. Informally, if we state that a function $f = O(g)$ we mean that f does not grow faster than g. The formal definition is a bit more tricky, as we allow for some additional 'start-up costs' that make f more costly than g in the beginning: we only require that g 'eventually outgrows' f.

Definition 2.1. A function $f(n)$ is $O(g(n))$ if there are constants $c \geq 0$, $n_0 \geq 1$ such that $f(n) \leq cg(n)$, for all $n \geq n_0$.

In other words, the $O(.)$ notation serves to ignore constants and lower order polynomials in the description of a function. For this reason $O(g(n))$ is also called the *order of magnitude* of $f(n)$.

Definition 2.2. An algorithm is said to be of time-complexity $O(f(n))$ if for every input with n elementary operations the number of steps it performs is $O(f(n))$.

We now will give a number of common examples. $O(1)$ or *constant time* algorithms have a running time that is upper-bounded by a constant, i.e., its running time is *independent* of the size of the input (at least from a particular input size n_0—see definition 2.1). A (rather trivial) example of such an algorithm might be the following:

Algorithm 2 Find the maximum element in a low-to-high sorted array with n values

MaximumSorted($A[1 \ldots n]$)

```
1: return A[n]
```

$O(n^c)$ or *polynomial time* algorithms have a running time that grows polynomially with the input size. For $c = 1$, $c = 2$ and $c = 3$ these running times are also known as *linear*, *quadratic* and *cubic*, respectively. For example, the following linear time algorithm finds the maximum number in an *unsorted* array by looking at all the elements one by one:

Algorithm 3 Find the maximum element in an unsorted array with n values

MaximumUnSorted($A[1 \ldots n]$)

```
1: m ← −∞
2: for i = 1 to n do
3:    if A[i] > m then
4:       m ← A[i]
5:    end if
6: end for
7: return m
```

This quadratic time algorithm sorts an array by repeatedly finding the minimum number and setting it in front of the array:

Algorithm 4 Sort an unsorted array with n values

SelectionSort($A[1 \dots n]$)

```
 1: for i = 1 to n do
 2:     st = i
 3:     for j = st + 1 to n do
 4:         if A[j] < A[st] then
 5:             st ← j
 6:         end if
 7:     end for
 8:     x = A[i]
 9:     A[i] = A[st]
10:     A[st] = x
11: end for
12: return A[1...n]
```

Note that there are two nested loops that range over n items, yielding an $O(n^2)$ time algorithm. Some algorithms need even more time: $O(c^n)$ or *exponential time* algorithms have a running time that grows exponentially with the input size. Here, c is a constant strictly larger than 1. The following $O(c^n)$ algorithm decides whether the nodes of a graph with n nodes can be coloured with c colours, such that no two adjacent nodes (i.e., nodes that share an edge) have the same colour. It tries to assign a colour to all of the n nodes, and checks after each assignment if the (partial) colouring still is valid. If not, it tries to colour that node using another colour; if all colours are used for that node, it tracks back and tries a different colour for the node before. Eventually, the algorithm will halt when all possibilities are exhausted or if all nodes have been assigned a (valid) colour. In the worst case, the algorithm needs to try all possible c^n colourings. The elementary operation here is the assignment in line (4).

Algorithm 5 Check whether a graph with n nodes can be coloured with c colours

GraphColour($\mathbf{G}, c$)

```
1: V[1...n] ← 0
2: i ← 1
3: while i > 0 do
4:    V[i] ← V[i] + 1
5:    if V[i] > c then
6:       V[i] = 0
7:       i ← i − 1
8:    end if
9:    if ValidColouring(V) then
10:      i ← i + 1
11:      if i > n then
12:         return true
13:      end if
14:   end if
15: end while
16: return false
```

Observe that we have the following series of increasing orders of magnitude:

$$O(1) \prec O(log(n)) \prec O(n) \prec O(n^2) \prec O(n^3) \prec \ldots \prec O(2^n) \prec O(n!) \prec \ldots$$

In addition to the $O(.)$ upper bound notation, there is a similarly defined *lower bound* notation:

Definition 2.3. A function $f(n)$ is $\Omega(g(n))$ if there are constants $c \geq 0$, $n_0 \geq 1$ such that $cg(n) \leq f(n)$, for all $n \geq n_0$.

When a function f is both upper-bounded and lower-bounded by g modulo some constants (i.e., f grows 'as hard' as g) we use the following notation:

Definition 2.4. A function $f(n)$ is $\Theta(g(n))$ if there are constants $c_1, c_2 \geq 0$, $n_0 \geq 1$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$, for all $n \geq n_0$.

These three notions are illustrated in Figure 1.

3 Worst, best, and average case running times

Assume you need to find the maximum number in an unsorted list of integers. Note that, *however the list looks like*, you need to examine *all* cards to make sure you have identified the highest number, as you cannot beforehand know whether the highest number is at the beginning of the list, somewhere in the middle, or at the end. Moreover, you do not know what the highest number actually is, so you cannot stop when you found it – you only *know*

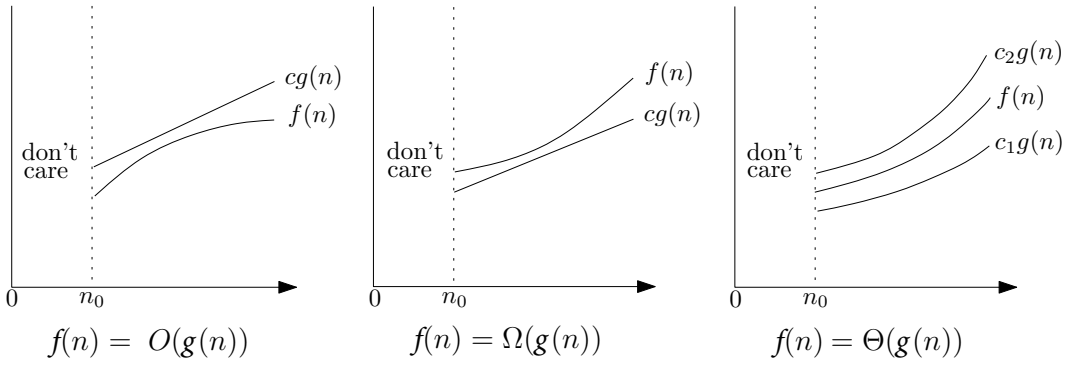


Figure 1: Illustrations of $O(.)$, $\Omega(.)$, and $\Theta(.)$

it was the highest number until you have seen all of them. In other words, in *each and every case*, to find the maximum of a list with n numbers you have to look at all n of them.

In the previous section, we studied the complexity of various algorithms on their 'worst scenario'. It can also be interesting to study how much work the algorithm needs to do in the 'best scenario', or on *average* when we look at all the possible inputs to the algorithm. We formalize these notions as follows:

Definition 3.1. We define the worst case, best case, and average case complexity of an algorithm as follows:

- The algorithm uses *at most* $f_w(n)$ elementary instructions for every possible input of size $n \leftrightarrow$ The algorithm has *worst case* complexity $f_w(n)$.
- The algorithm uses *at least* $f_b(n)$ elementary instructions for every possible input of size $n \leftrightarrow$ The algorithm has *best case* complexity $f_b(n)$.
- The algorithm uses *on average* $f_a(n)$ elementary instructions, averaged over all possible inputs of size $n \leftrightarrow$ The algorithm has *average case* complexity $f_a(n)$.

Recall the **Selection Sort** algorithm from the previous section and compare it to the following algorithm:

Algorithm 6 Sort an unsorted array with n values

InsertionSort($\mathbf{A}[1 \dots n]$)

```
1: for i = 1 to n do
2:     x ← A[i]
3:     j ← i − 1
4:     while j > 0 and A[j] > x do
5:         A[j + 1] ← A[j]
6:         j ← j − 1
7:     end while
8:     A[j + 1] ← x
9: end for
10: return A[1 . . . n]
```

Stop and think

Assume that the initial array is already sorted low-to-high. Count the number of array comparisons $\mathbf{A}[j] > x$ required. Now assume that the initial array is sorted high-to-low. Again count the number of comparisons required.

Observe that for every input, **Selection Sort** needed $O(n^2)$ array comparisons. However, while the number of array comparisons for **Insertion Sort** is also $O(n^2)$ in the worst case, it can be significantly lower in the best case.

4 The classes P and NP

For many problems we have polynomial-time algorithms. For many other problems we don't. Can we show that no such algorithm is possible (i.e., that it isn't just our ignorance that lies between the problem and an efficient solution)? Unfortunately, for most assumed-to-be-intractable problems an exponential lower bound is extremely difficult to prove. For example, if one would be able to prove that the **Constraint Satisfaction** problem does not have a polynomial-time algorithm would pay you $1,000,000 (it would settle a theoretic question for which the Clay Mathematics Institute offers this amount of money). So, a different approach seems warranted here. To get some intuition and a handle on the problem, we introduce the **Graph 3-Colourability** problem.

Graph 3-Colourability
Input: A graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$.
Question: Is there a three-colouring of the graph, i.e., and assignment $c : V \rightarrow \{c_1, c_2, c_3\}$ such that no vertices that share an edge have the same colour?

See below a positive and a negative instance of **Graph 3-Colourability**:

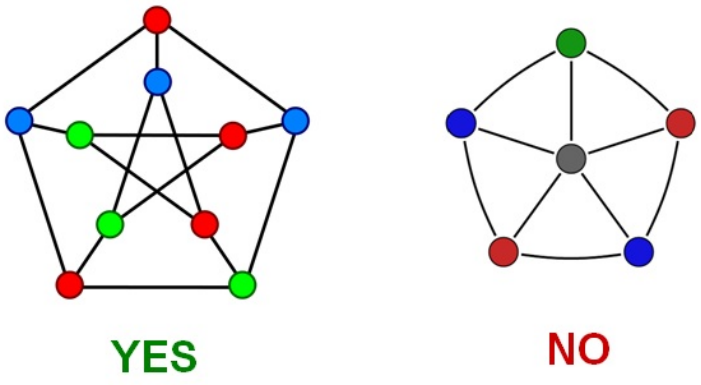
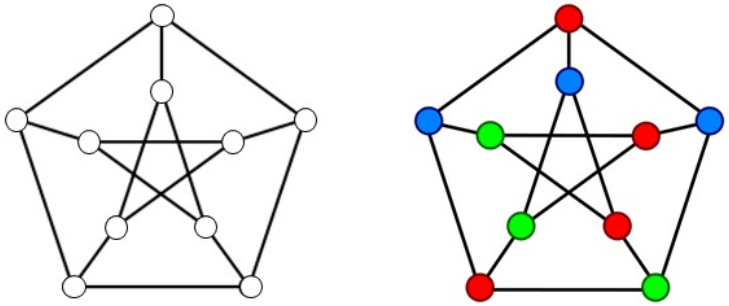


Figure 2: *yes* and *no*-instance of **Graph 3-Colourability**

Stop and think



Is it easy to decide this instance (left) of **Graph 3-Colourability**? Is it easy to verify that it is a *yes*-instance given a valid colouring (right)?

yes-instances of the **Graph 3-Colourability** problem are easily verifiable: you can check in polynomial time that a given colouring is correct. However, there is no straightforward way of *finding* such a correct colouring that is guaranteed to take only polynomial time (think of backtracking etc.). There is also no straightforward way of verifying that *no*-instances are indeed *not* colourable with three colours! This brings us to the following definitions:

Definition 4.1. P is the class of decision problems D that are solvable in polynomial-time. More formally, P is the class of decision problems $D : I \to \{yes, no\}$ for which there exists an algorithm with worst case time complexity $O(|i|^c)$ (for a constant c) that can decide whether $D(i) = yes$ or $D(i) = no$ for every instance $i \in I$.

Definition 4.2. NP is the class of decision problems D for which there exist a polynomial-time verifiable proof for *yes*-instances $D(i_{yes})$. More formally, NP is the class of decision problems $D : I \to \{yes, no\}$ for which there exists an algorithm with worst case time complexity $O(|i_{yes}|^c)$ (for a constant c) that can verify that $D(i_{yes}) = yes$ for every *yes*-instance $i_{yes} \in I_{yes}$.

For completeness, we also mention:

Definition 4.3. co−NP is the class of decision problems D for which there exist a polynomial-time verifiable proof for *no*-instances $D(i_{no})$. More formally, co−NP is the class of decision problems $D : I \to \{yes, no\}$ for which there exists an algorithm with worst case time complexity $O(|i_{no}|^c)$ (for a constant c) that can verify that $D(i_{no}) = no$ for every *no*-instance $i_{no} \in I_{no}$.

Stop and think

Consider the **Graph 3-ColourINability** problem: Given a graph, is it the case that there exists *no* colouring whatsoever that will colour the graph with at most three colours? It may not trivial to show for a given graph that no such colouring exists. However, it is easy to verify a *no*-instance to **Graph 3-ColourINability** (that is, a graph that *does* have such a three-colouring; be aware of the double negation here!): just give a counterexample, i.e., a valid colouring; you can easily show that this colouring (which must necessarily exist for *no*-instance) indeed colours the graph with three colours.

The co−NP stuff may dazzle you a bit. No worries - I only included it for completeness. We will focus on the remainder of this appendix on P and NP. We have seen a problem that is in NP, namely **Graph 3-Colourability**. We also saw a problem in P, for example **Selection Sort**. Here are a few more problems:

CNF-Sat
Input: A Boolean formula ϕ in conjunctive normal form.
Question: Is ϕ satisfiable?

Shortest Path
Input: A weighted graph **G** with two designated vertices S and F; integer k.
Question: Is there a path in **G**, starting with S and ending with F that has total weight at most k?

Stop and think

Try to answer the following questions: 1) Is the **CNF-Sat** problem in P? 2) Is the **CNF-Sat** problem in NP? 3) Is the **Shortest Path** problem in P? 4) Is the **Shortest Path** problem in NP?

And the answers:

1. We don't know, but we believe it is not (see Section 6).
2. Yes. It is easy to verify *yes*-instances once given a truth assignment to ϕ; one can evaluate this in time, linearly in the size of the formula.
3. Yes. Finding shortest paths is a classical polynomial-time solvable problem, using, e.g., Dijkstra's algorithm if all weights are positive. We can just solve the problem and output whether the shortest path has a length, smaller than or equal to k.
4. Yes. When given a **Shortest Path** instance and a tour (that happens to have length no more than k), it is easy to verify that this is indeed a *yes*-instance: just add up

the weights and check if the tour leads from S to F. If a problem is in P it is *also* in NP. Think about it. You can always throw away the solution and compute a solution yourself in polynomial time and check whether it was indeed a *yes*-instance you got...

Given a particular decision problem, how can one prove membership of one of these two classes P and NP? Or, maybe just as interesting, *disprove* membership? Actually, proving membership is at least conceptually easy: A decision problem is in P if there is an algorithm that can decide it in polynomial time; it is in NP if there is an algorithm that can verify *yes*-instances in polynomial time, given a so-called *certificate* (an actual solution, or other information that suffices as proof of being a *yes*-instance). So, we 'only' need to come up with a polynomial-time algorithm—that will suffice to show the *existence* of such an algorithm. We call such a proof a *constructive* proof: We show that there exists an algorithm by actually constructing such an algorithm. Disproving membership of P and NP, however, is much more complicated. To show that a decision problem is not in P (or NP) one needs to give a super-polynomial lower bound for deciding (or verifying) the problem at hand.

Stop and think

What if we can come up with a proof of membership in NP (i.e., a polynomial-time verifiable certificate), and a proof of non-membership in P (i.e., a super-polynomial lower bound)?

If one can show that a decision problem can be verified in polynomial time, but not decided in polynomial time, one has effectively proven that $\mathsf{P} \neq \mathsf{NP}$. The $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ question is one of the most important open problems in mathematics today. In fact, the question is so important that the Clay Mathematics Institute has offered a $ 1,000,000 price to anyone who can answer it. Why is solving this problem so hard? Almost the sole progress in the last forty-odd years on this question is about known mathematical techniques that provably cannot separate P from NP. For example, diagonalization (the technique used to show that the set of real numbers cannot be mapped to the set of natural numbers) cannot be used to show that there is at least one problem in NP that is not in P.

Thus, it is difficult to show that a problem P is in NP, but not in P. What we *can* do, is show that it is among the *hardest* problems in NP, that is, show that if P enjoys a polynomial-time algorithm, then *each and every* problem in NP does (effectively proving that $\mathsf{P} = \mathsf{NP}$). Given that we believe that $\mathsf{P} \neq \mathsf{NP}$, proving that P is "among the hardest problems in NP" (we will give a formal notion of that in Section 6) is fairly convincing evidence that P is intractable.

5 Reductions

Observe that sometimes you may have the intuition that a problem has no polynomial-time algorithm, but that you cannot prove it. What you may be able to prove, however, is that if one of the problems is polynomial-time computable then so is another. The strategy that we

use is to come up with an algorithm A that *translates* every instance of a particular decision problem D_1 into a corresponding instance of another decision problem D_2, and that takes polynomial time on every instance of D_1. This algorithm may be trivial or complex, but it must be such that every time the answer to D_1 is *yes*, the answer to D_2 is *yes*, and every time the answer to D_1 is *no*, the answer to D_2 is *no*. We call such an algorithm a *reduction* as it 'reduces' the burden to solve D_1 to 'merely' solving D_2: we can translate every instance i_1 of D_1 to a corresponding instance i_2 of D_2, decide i_2, and then output the answer to i_2—by definition, the answer to i_2 is the same as the answer to i_1.

We will start by giving the formal definition of a polynomial-time reduction. Then we will guide you through an easy reduction: from the **Clique** problem to the **Independent Set** (both defined below), and we will ask you to show that this is indeed a polynomial-time reduction.

Definition 5.1. For two decision problems D_1 and D_2 we say that D_1 *reduces to* D_2, if there exists an algorithm A that transforms any instance i_1 of D_1 into an instance $i_2 = A(i_1)$ of D_2 such that i_2 is a *yes*-instance for D_2 if and only if i_1 is a *yes*-instance for D_1. We furthermore say the reduction is a *polynomial-time* reduction if the algorithm A runs in polynomial time. The notation $D_1 \leq_m^P D_2$ means that D_1 is reducible in polynomial time to D_2. The 'm' in this notation can be ignored for now.

Stop and think

Imagine you are studying a problem P and you would like to know if it is decidable in polynomial-time or not. You try, possibly for a long time, to come up with a polynomial-time algorithm, but so far have failed to find one. You now have built up the intuition that there does not exist a polynomial-time algorithm solving P and, if this is indeed the case, you would like to be able to prove this. Imagine further, that there is a problem Q for which it has already been shown that there exists no polynomial-time algorithm. How could you use a polynomial-time reduction (as defined in Definition 5.1) to prove that P also has no polynomial-time algorithm?

Polynomial-time reductions are very useful in relating the time complexity of a problem of interest to other problems. Let us assume (as in Definition 5.1) that $D_1 \leq_m^P D_2$. Now the following statements hold:

- If D_2 can be decided in polynomial time, then so can D_1
- If D_1 cannot be decided in polynomial time, then D_2 cannot either

Let us examine both claims. Assume we have a polynomial-time algorithm for D_2. How can we use that algorithm to solve D_1 in polynomial time? Well, given that $D_1 \leq_m^P D_2$, we can first reduce D_1 to D_2. That can be done in polynomial time by definition. Then we decide

D_2. As both *yes*- and *no*-answers to D_2 map onto *yes*- and *no*-answers to D_1, respectively, the answer to D_2 equals the answer to D_1. The other claim is a direct consequence: if we would have proof that D_1 cannot be decided in polynomial time, then there cannot be any polynomial-time algorithm for D_2, otherwise we would arrive at a contradiction.

Take care not to swap D_1 and D_2 in the above statements! Even if D_1 can be solved in polynomial time and $D_1 \leq_m^P D_2$, that tells us *nothing* about the complexity of D_2! Now that we know the consequences of reducing D_1 to D_2, how can we come up with an algorithm doing the actual reduction, and how do we show that it is indeed a valid reduction? We will illustrate that process with a reduction from **Clique** to **Independent Set**. Below you will find the definitions of these decision problems:

Clique
Input: A graph $G = (V, E)$ and an integer k
Question: Does there exist a clique $V' \subseteq V$ such that $|V'| \leq k$? (Here a vertex set V' is called a *clique* if for all two vertices $u, v \in V'$ there is an edge $(u, v) \in E$).

Independent Set
Input: A graph $G = (V, E)$ and an integer k.
Question: Does there exist an independent set $V' \subseteq V$ such that $|V'| \leq k$? (Here a vertex set V' is called an *independent set* if there exist no two vertices $u, v \in V'$ such that $(u, v) \in E$).

Stop and think

Think of these two problems. What is related between them and what is different? Think of a generic transformation that we can apply to every instance to **Clique** such that it becomes an instance of **Independent Set**. Does that transformation map both *yes*-instances and *no*-instances of **Clique** to corresponding *yes*-instances and *no*-instances of **Independent Set**? Can that transformation be done in polynomial time?

From these definitions it follows that a vertex set is a clique if *every* pair of two vertices in the set is connected; the set is an independent set if *no* pair of two vertices in the set is connected. Now how can we transpose the input of **Clique** to match this difference in the problem? The idea is easy, once you have thought of it: we transform each input to **Clique** to an input to **Independent Set** by "negating the edges": if there is an edge in the **Clique** instance, we'll have the absence of an edge in the **Independent Set** and vice versa. See Figure 3 for an example.

Note the duality of both graphs: for each edge in the **Clique** instance to the left, there is the absence of an edge in the **Independent Set** instance to the right; for each missing edge in the **Clique** instance, there is an edge in the **Independent Set** instance. Observe that $\{A, B, C\}$ forms a clique in the left graph *and* an independent set in the right graph.

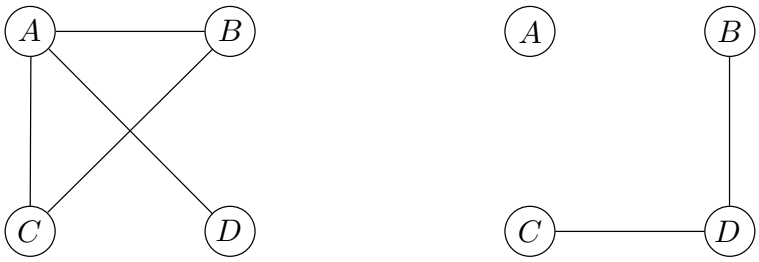


Figure 3: Example of transforming the input for a **Clique** instance to the input for an **Independent Set** instance

Stop and think

- Argue why the above observation also holds in general: let G be the graph of the **Clique** instance and G' the graph that is obtained by "negating the edges" of G. Show that there is an independent set of size k in G', if and only if there is a clique of size k in G.
- Argue that this transformation takes time, polynomial in the size of G and thus is a polynomial-time reduction from **Clique** to **Independent Set**.
- Show that the reduction is symmetrical: using the same reduction, we can prove **Independent Set** $\leq_m^P$ **Clique** as well as **Clique** $\leq_m^P$ **Independent Set**.

In general, to show that a problem D_1 polynomial-time reduces to a problem D_2 by mathematical proof, all of the next proof steps have to be explicated:

1. Describe an algorithm A that transforms instances for D_1 into instances for D_2, possibly using an example (as we did in Figure 3).
2. Consider an *arbitrary* instance i_1 for D_1. Prove that $i_2 = A(i_1)$ is a *yes*-instance for D_2 *if and only if* i_1 is a *yes*-instance for D_1. The 'if and only if' proof has two sub-steps:
   (a) Assume i_1 is a *yes*-instance for D_1. Show that then also i_2 is a *yes*-instance for D_2 (i.e., give a convincing argument).
   (b) Assume i_2 is a *yes*-instance for D_2. Show that then also i_1 is a *yes*-instance for D_1. (i.e., give a convincing argument). Or as an alternative to this sub-step: Assume i_1 is a *no*-instance for D_1. Show that then also i_2 is a *no*-instance for D_2.
3. Show that A runs in polynomial-time.

Observe that we map all instances of D_1 to corresponding instances of D_2 by the transformation algorithm A. This need not necessarily be symmetrical: there may be instances of D_2 that do not map back to instances of D_1. However, if the reduction works both ways, then

it is *symmetrical* and there is indeed a mapping from all instances of D_2 to D_1, as well as the other way around. The reduction we saw above, from **Clique** to **Independent Set**, is indeed symmetrical.

6 NP-completeness

Having covered the complexity classes P and NP, and polynomial-time reductions in previous sections, we can now give the definition of NP-hardness. NP-hard problems derive their name from the fact that they *are at least as hard* to compute as *any* problem in the class NP, which is formalized in the definitions below.

Definition 6.1. A problem P is NP-hard if for *all* decision problems $D \in \mathsf{NP}$ there exists a polynomial-time reduction to P.

Definition 6.2. A decision problem D that is both NP-hard and in NP is called NP-complete.

Stop and think

Explain why the following holds: If an NP-complete problem were to be in P then P = NP.

An NP-complete problem D is a problem in NP such that *each and every* (other) problem D' in NP can be reduced, in polynomial time, to D. So, that gives us *in principle* a polynomial-time algorithm for all problems in NP: for every D' in NP, first reduce D' to D, then apply the polynomial-time algorithm that decides D (we know there is such an algorithm, as we assumed that D is in P). Note that we don't actually need to *have* that algorithm readily available—it suffices to proof that such an algorithm exists, and thus in principle can be applied to decide D. The challenge is, it seems, in proving that D is NP-hard. Surely, we may be able to come up with a reduction from *a particular* decision problem D_1 in NP to D, but what is required is a proof that *every* decision problem D' reduces to D in polynomial time!

However, our job has been made considerably easier by Stephen Cook in 1971. He showed that *every problem in* NP can be reduced in polynomial-time to **CNF-Sat**. In Section 4 we claimed, that even if we cannot directly prove that problems like **CNF-Sat** cannot be decided in polynomial time, we *can* prove that if there exists a polynomial-time algorithm for *one* of them, there exists one for *all* of them. That is because all these problems can be reduced, directly or indirectly, from **CNF-Sat** using a polynomial-time reduction. Because **CNF-Sat** is NP-hard, so is *any* problem that can be reduced (in polynomial time) from **CNF-Sat**. It suffices to give such a reduction from any existing NP-hard problem to D, our problem of interest, to show that D is NP-hard as well.

I will not go into details about Cook's proof as it requires much more technical details than can be provide here. The general idea, however, is that Cook showed that the computation

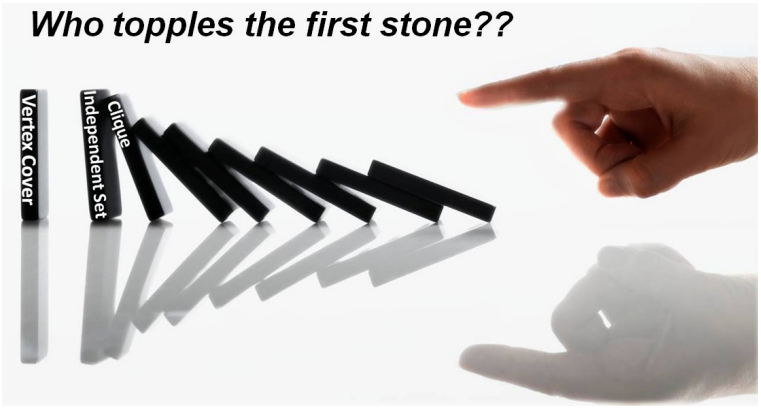


Figure 4: We can show that if the first stone falls (i.e., a problem from which all the others can be reduced is shown to be NP-hard) then the chain of reductions will prove that all these stones will fall (all these problems are NP-hard). But who topples the first stone?

that leads to the acceptance of every yes-instance i_{yes} of an arbitrary problem D' in NP can be represented by a (huge, but polynomially sized) **CNF-Sat** formula ϕ, such that this computation ends with 'accept' if and only if ϕ is satisfiable. That means: if we can decide that ϕ is satisfiable in polynomial time, we can confirm that i_{yes} is indeed a *yes*-instance of D', for *each and every* problem D' in NP.

A common mistake when trying to prove NP-hardness is to reverse the direction of a reduction when trying to prove NP-hardness. To recap: to show that your problem P is NP-hard, you must provide a polynomial-time reduction *from* an known NP-hard problem Q *to* P. It is easy to go wrong here (and it occasionally happens to all of us), so take care and try to fully understand what you are doing and why. A reduction in the wrong direction does not prove that your problem isn't NP-hard–it just doesn't prove anything.

The strategy that I suggest to prove that your problem P is NP-hard is to first select a suitable NP-complete problem Q, then to come up with a reduction from Q to P, and finally to show that this reduction takes polynomial time. In principle, if P is indeed NP-hard, there exists a reduction from every problem in NP to P. But, to paraphrase George Orwell's *Animal Farm*, "all polynomial-time reductions are equal, but some are more equal than others". A smart choice for Q will make your life considerably easier. The following heuristics may be helpful.

- Can you show that there is a Q that is a special case of P?
- Can you show that there is a Q that is sufficiently similar to P, so that you can reduce Q tot P by 'local replacement'?
- If the above fail, the best heuristic may be reduce from **CNF-Sat** given the generality of this problem.

There is much more to say about reductions and NP-hardness. I advise you to go to the library and look up a copy of the canonical book *Computers and Intractability: A Guide to the Theory of* NP*-completeness* by Garey and Johnson. Although it is 35 years old (from

1979) it is still the authoritative textbook on this subject. Most prominent is its impressive list of known NP-hard problems, which may act as a starting point for reductions.