NP-Completeness

~~To be~~ or ~~not~~ to ~~be~~ ....

Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

NP-Completeness

Standard reminder to set phones to silent/vibrate mode, please!

NP-Completeness

- During today's lesson:
  - Look at some important categorizations of algorithms based on their run times.
    - Including one important distinction: polynomial time vs. exponential time.
  - Look into the notion of decision problems, and their relationship to an associated computation or optimization problem.
  - Introduce the sets P and NP.
  - Introduce the notions of NP-hard and NP-complete problems.

NP-Completeness

- For the purpose of this discussion, we will be mostly interested in the distinction of polynomial time algorithms vs. non-polynomial time algorithms (exponential and above):

NP-Completeness

- For the purpose of this discussion, we will be mostly interested in the distinction of polynomial time algorithms vs. non-polynomial time algorithms (exponential and above):
  - Polynomial time: Algorithms with run time $\mathrm{O}\left(n^{\alpha}\right)$ for some $\alpha > 0$.

NP-Completeness

- For the purpose of this discussion, we will be mostly interested in the distinction of polynomial time algorithms vs. non-polynomial time algorithms (exponential and above):
  - Polynomial time: Algorithms with run time $\mathrm{O}(n^{\alpha})$ for some $\alpha > 0$.
  - Non-polynomial time: Algorithms with run time $\Omega(a^{n})$ for some $a > 1$.

NP-Completeness

- Examples of polynomial time:
  - Constant time
  - Logarithmic
  - Linear time
  - $n \log n$
  - Things like n^2, n^3, $n^3 \log n$
  - Etc.

NP-Completeness

- Examples of non-polynomial time:
  - Exponentials (of any base): 2^n, 10^n, etc.
  - $n\,2^n$
  - $n^2\,2^n$
  - $n!$
  - n^n
  - Etc.

NP-Completeness

- Tractable vs. intractable problems:
  - A problem for which a polynomial time algorithm exists that solves the problem is called a *tractable* problem.
  - Otherwise, we refer to them as *intractable* problems.

NP-Completeness

- Tractable vs. intractable problems:
  - A problem for which a polynomial time algorithm exists that solves the problem is called a *tractable* problem.
  - Otherwise, we refer to them as *intractable* problems.
  - Notice the subtlety — a polynomial time algorithm exists vs. a polynomial time algorithm is known.

NP-Completeness

- Tractable vs. intractable problems:
  - A problem for which a polynomial time algorithm exists that solves the problem is called a *tractable* problem.
  - Otherwise, we refer to them as *intractable* problems.
  - Notice the subtlety — a polynomial time algorithm exists vs. a polynomial time algorithm is known.
    - We often see statements such as "this problem is believed to be intractable" or "is considered intractable", meaning that no polynomial time algorithm is known, and it is not known for sure that none exist, but it is *believed* that none exist.

NP-Completeness

- An important notion related to these is that of *polynomial time reduction*.

NP-Completeness

- An important notion related to these is that of *polynomial time reduction*.
  - We talked about reductions — solving problem A using an algorithm that solves problem B.
    - In this case, we say that A reduces to B.

NP-Completeness

- An important notion related to these is that of *polynomial time reduction*.
  - We talked about reductions — solving problem A using an algorithm that solves problem B.
    - In this case, we say that A reduces to B.
    - Notice that we're applying the notion to the problems, and not necessarily to the algorithms — solving problem A reduces to being able to solve problem B.

NP-Completeness

- The reduction takes some time, since we need to transform an instance of a problem into another problem
  - This typically refers to transforming the input of algorithm A into a valid input for algorithm B, then invoking algorithm B, capture its output and derive (transform) the output for algorithm A.

NP-Completeness

- Relating to today's topic, since we're interested in the distinction between polynomial time vs. non-polynomial time algorithms, then the relevant reductions are those with polynomial run times:
  - They will allow us to prove statements about an algorithm having polynomial time or not.

NP-Completeness

- For example, given:

$$T_A(n) = T_R(n) + T_B(n)$$

- If we know that $T_A(n)$ is non-polynomial and we're trying to prove that $T_B(n)$ is also non-polynomial, then we need $T_R(n)$ to be polynomial — otherwise, the statement (the equation) would say absolutely nothing about $T_B(n)$ being or not polynomial.

NP-Completeness

- For example, given:

$$\mathrm{T}_{\mathrm{A}}(n) = \mathrm{T}_{\mathrm{R}}(n) + \mathrm{T}_{\mathrm{B}}(n)$$

- On the other hand, if $\mathrm{T}_\mathrm{B}(n)$ is polynomial and we're trying to show that we can find an algorithm A with polynomial time $\mathrm{T}_\mathrm{A}(n)$ by reducing A to B, then in this case we also need $\mathrm{T}_\mathrm{R}(n)$ to be polynomial — otherwise, the above equation says that $\mathrm{T}_\mathrm{A}(n)$ is non-polynomial, so we've failed to find such polynomial time algorithm A.

NP-Completeness

- Some times we refer to polynomial time reduction as an efficient reduction.
  - In this context, polynomial time is efficient, non-polynomial time is inefficient.

NP-Completeness

- Notation:

NP-Completeness

- Notation:
  - If problem A reduces to problem B, that means that solving A can not be harder than solving B.

NP-Completeness

- Notation:
  - If problem A reduces to problem B, that means that solving A can not be harder than solving B.
  - This is denoted by: $A \leqslant_P B$

NP-Completeness

- Notation:
  - If problem A reduces to problem B, that means that solving A can not be harder than solving B.
  - This is denoted by: $A \leqslant_P B$
  - We read it as: A polynomial-time-reduces-to B (or A reduces to B in polynomial time)

NP-Completeness

- Notice that if $A \leqslant_P B$:
  - We can not know whether A is easier (more efficient) than B or equally as hard — the only thing that we know is that A is, *at most*, as hard as B, since we can always use B to solve A.

NP-Completeness

- Notice that if $A \leqslant_P B$:
  - We can not know whether A is easier (more efficient) than B or equally as hard — the only thing that we know is that A is, *at most*, as hard as B, since we can always use B to solve A.
  - But there could always be some other way of solving A which is more efficient — the fact that we find a reduction from A to B says nothing about that.

NP-Completeness

- A fascinating class of questions on algorithmic theory arises when we consider *decision problems* and their complexity.

NP-Completeness

- A fascinating class of questions on algorithmic theory arises when we consider *decision problems* and their complexity.
- A *decision problem* is simply a problem with a yes/no answer
    - An algorithm that solves a decision problem outputs a boolean value.

NP-Completeness

- A fascinating class of questions on algorithmic theory arises when we consider *decision problems* and their complexity.
- A *decision problem* is simply a problem with a yes/no answer
  - An algorithm that solves a decision problem outputs a boolean value.
  - Notice the interesting detail: when reducing a decision problem to another decision problem, we only need to do a transformation for the input — the output for both algorithms is already compatible (at most, we may need to negate the output)

NP-Completeness

- Many (perhaps most) of the decision problems of interest are related to a computational or optimization problem.

NP-Completeness

- Many (perhaps most) of the decision problems of interest are related to a computational or optimization problem.
  - Examples:
  - Decision problem: Determine whether a given graph has a Hamiltonian cycle

NP-Completeness

- Many (perhaps most) of the decision problems of interest are related to a computational or optimization problem.
  - Examples:
  - Decision problem: Determine whether a given graph has a Hamiltonian cycle
    - Related computational problem: Find a Hamiltonian cycle (if there is one) in a given graph.

NP-Completeness

- Many (perhaps most) of the decision problems of interest are related to a computational or optimization problem.
  - Examples:
  - Decision problem: Determine whether a given graph has a Hamiltonian cycle
    - Related computational problem: Find a Hamiltonian cycle (if there is one) in a given graph.
  - Decision problem: Given two vertices in a given graph, is there a path between them shorter than a given value *k*?
    - Optimization problem: Find the shortest path.

NP-Completeness

- The decision problem always reduces to its associated optimization or computational problem

NP-Completeness

- The decision problem always reduces to its associated optimization or computational problem — examples:
  - We can solve the problem of determining whether there is a path shorter than *k* if we have an algorithm that finds the shortest path — simply invoke that algorithm, look at its output and check whether the shortest path's length is less than *k*.

NP-Completeness

- The decision problem always reduces to its associated optimization or computational problem — examples:
  - We can solve the problem of determining whether there is a path shorter than *k* if we have an algorithm that finds the shortest path — simply invoke that algorithm, look at its output and check whether the shortest path's length is less than *k*.
  - We can determine whether a graph has a Hamiltonian cycle if we have an algorithm that finds a Hamiltonian cycle in a graph.

NP-Completeness

- The converse is often true as well — the optimization or computational problem can often be reduced to the decision problem.

NP-Completeness

- The converse is often true as well — the optimization or computational problem can often be reduced to the decision problem. Example:
  - If we have an algorithm that determines whether there is a path shorter than a given value, we can determine the length of the shortest path by using that algorithm:

NP-Completeness

- The converse is often true as well — the optimization or computational problem can often be reduced to the decision problem. Example:
  - If we have an algorithm that determines whether there is a path shorter than a given value, we can determine the length of the shortest path by using that algorithm:
    - Call the decision algorithm many times, until we “narrow down” the length of the shortest path:
      - The algorithm outputs NO when asked whether there is a path shorter than 10, and outputs YES when asked whether there is a path shorter than 11 — we have our answer !

NP-Completeness

- This justifies the use of decision problems in the theoretical study of algorithms and complexities:

NP-Completeness

- This justifies the use of decision problems in the theoretical study of algorithms and complexities:
  - If we show that a decision problem is hard (intractable), then that automatically shows that the associated optimization or computational problem is also hard (since the decision problem reduces to the other one)

NP-Completeness

- Next, let's look at the sets, or *complexity classes*, P and NP

NP-Completeness

- The formal definitions are quite involved, so we'll skip those and go for the *somewhat* practical definitions...

NP-Completeness

- The formal definitions are quite involved, so we'll skip those and go for the *somewhat* practical definitions...
  - The complexity class P is the set of (decision) problems that can be decided in polynomial time.

NP-Completeness

- The formal definitions are quite involved, so we'll skip those and go for the *somewhat* practical definitions...
  - The complexity class P is the set of (decision) problems that can be decided in polynomial time.
    - That is, the problems for which a polynomial time algorithm exists that solves (decides) the problem.

NP-Completeness

- The formal definitions are quite involved, so we'll skip those and go for the *somewhat* practical definitions...
  - The complexity class P is the set of (decision) problems that can be decided in polynomial time.
    - That is, the problems for which a polynomial time algorithm exists that solves (decides) the problem.
  - Straightforward enough: P stands for Polynomial time (so, we're talking about the set of Polynomial time decidable problems)

NP-Completeness

- The complexity class NP is a bit trickier:
  - NP stands for Nondeterministic Polynomial time

NP-Completeness

- The complexity class NP is a bit trickier:
  - NP stands for Nondeterministic Polynomial time
    - The set of (decision) problems that can be decided in polynomial time using a *nondeterministic* machine!

NP-Completeness

- The complexity class NP is a bit trickier:
  - NP stands for Nondeterministic Polynomial time
    - The set of (decision) problems that can be decided in polynomial time using a *nondeterministic* machine!
  - Now, we don't want to deal with nondeterministic (as in, random) machines, so here is the tricky part:

NP-Completeness

- The complexity class NP is a bit trickier:
  - NP stands for Nondeterministic Polynomial time
    - The set of (decision) problems that can be decided in polynomial time using a *nondeterministic* machine!
  - Now, we don't want to deal with nondeterministic (as in, random) machines, so here is the tricky part:
  - It turns out that the above definition is equivalent to the following, more convenient one:
    - NP is the set of decision problems that can be decided in polynomial time if provided with a certificate, typically corresponding to a solution to the associated optimization or computational problem.

NP-Completeness

- The key detail here is counting on an efficient (i.e., polynomial time) *verification* algorithm.

NP-Completeness

- The key detail here is counting on an efficient (i.e., polynomial time) *verification* algorithm.
- For example: Determining whether a given graph has a Hamiltonian cycle is an NP problem:
  - It can be efficiently *verified* — that is, if we're given a certificate claiming to be a Hamiltonian cycle for the graph, we can verify that claim in polynomial time.

NP-Completeness

- The key detail here is counting on an efficient (i.e., polynomial time) *verification* algorithm.
- For example: Determining whether a given graph has a Hamiltonian cycle is an NP problem:
  - It can be efficiently *verified* — that is, if we're given a certificate claiming to be a Hamiltonian cycle for the graph, we can verify that claim in polynomial time (we only need to verify that all the vertices are visited, that all edges correspond to existing edges, and that no vertex is visited more than once).

NP-Completeness

- The key detail here is counting on an efficient (i.e., polynomial time) *verification* algorithm.
- The subset-sum problem is in NP as well:
  - Given a certificate corresponding to the subset of values that add to 0 (or the given parameter, in the more general form of the problem), we can verify in linear time that they indeed add to 0.

NP-Completeness

- Here's another tricky detail: The set P $\subseteq$ NP. That is, if a problem can be decided in polynomial time, then it can be verified in polynomial time.

NP-Completeness

- Here's another tricky detail: The set P $\subseteq$ NP. That is, if a problem can be decided in polynomial time, then it can be verified in polynomial time.
  - Just ignore the certificate — solve (decide) the problem in polynomial time, which can be done without the certificate, since the problem is in the set P.

NP-Completeness

- Here's a big (tricky?) question: Is NP $\subseteq$ P?

NP-Completeness

- Here's a big (tricky?) question: Is NP $\subseteq$ P?
  - That is: is every problem that can be efficiently verified also efficiently solved?

NP-Completeness

- Here's a big (tricky?) question: Is NP $\subseteq$ P?
  - That is: is every problem that can be efficiently verified also efficiently solved?
  - Basic intuition obviously says NO:

NP-Completeness

- Here's a big (tricky?) question: Is NP $\subseteq$ P?
  - That is: is every problem that can be efficiently verified also efficiently solved?
  - Basic intuition obviously says NO:
    - Think of finding a needle in a haystack vs. verifying that there is a needle at a location that we're told.

NP-Completeness

- Here's a big (tricky?) question: Is NP $\subseteq$ P?
  - That is: is every problem that can be efficiently verified also efficiently solved?
  - Basic intuition obviously says NO:
    - Think of finding a needle in a haystack vs. verifying that there is a needle at a location that we're told.
    - Find the factorization of a number vs. verify that a set of numbers is the factorization of a given number.

NP-Completeness

- We're used to this asymmetry between solving a problem and verifying a given solution.
- Some technologies even *rely* on this!

NP-Completeness

- We're used to this asymmetry between solving a problem and verifying a given solution.
- Some technologies even *rely* on this! Modern cryptography, for example, relies on the asymmetry of discrete logarithm problem:

NP-Completeness

- We're used to this asymmetry between solving a problem and verifying a given solution.
- Some technologies even *rely* on this! Modern cryptography, for example, relies on the asymmetry of discrete logarithm problem:
  given a^x mod m, determine the value of x — the asymmetry being: given x, it is trivial to obtain a^x mod m, but given a^x mod m, it is *believed* that finding x is an intractable problem)

NP-Completeness

- Notice, BTW, that NP $\subseteq$ P would imply that P = NP (i.e., that the two sets are the same), since P $\subseteq$ NP

NP-Completeness

- Despite this intuition, as obvious as it may seem, whether P = NP is an unanswered question; not only that: it is considered the single most important open problem in theoretical computer science!

NP-Completeness

- Despite this intuition, as obvious as it may seem, whether P = NP is an unanswered question; not only that: it is considered the single most important open problem in theoretical computer science!
  - Perhaps the puzzling aspect is: if it seems so obvious that P ≠ NP, why has no-one been able to mathematically prove it?

NP-Completeness

- Despite this intuition, as obvious as it may seem, whether P = NP is an unanswered question; not only that: it is considered the single most important open problem in theoretical computer science!
  - Perhaps the puzzling aspect is: if it seems so obvious that P ≠ NP, why has no-one been able to mathematically prove it?
    - Corollary: if no-one, despite a lot of brilliant people trying really hard, has been able to prove it (that P ≠ NP), could it be because it is not true that P ≠ NP??

NP-Completeness

- NP-complete problems:
  - In short, a set of problems in NP that all reduce to each other!

NP-Completeness

- NP-complete problems:
  - In short, a set of problems in NP that all reduce to each other!
  - Huh?? How could this happen???

NP-Completeness

- NP-complete problems:
    - In short, a set of problems in NP that all reduce to each other!
    - Huh?? How could this happen??? (back to this in a minute)

NP-Completeness

- NP-complete problems:
  - In short, a set of problems in NP that all reduce to each other!
  - One important implication is that these problems are all equivalent:
    - If someone ever finds a polynomial time solution to one of them, then *all* of them have polynomial time solutions.
    - Conversely, if someone ever proves that one of these problems is intractable, then we know that *all* of them are intractable.
      - Corollary: this would prove that P ≠ NP !!!

NP-Completeness

- NP-complete problems:
  - How do we prove that a given problem is NP-complete?

NP-Completeness

- NP-complete problems:
  - How do we prove that a given problem is NP-complete?
    - If we can reduce any other problem X_{NPC} known to be NP-complete, then we're done — every other NP-complete problem reduces in polynomial time to X_{NPC}, and therefore to our given problem

NP-Completeness

- NP-complete problems:
  - How do we prove that a given problem is NP-complete?
    - If we can reduce any other problem X_{NPC} known to be NP-complete, then we're done — every other NP-complete problem reduces in polynomial time to X_{NPC}, and therefore to our given problem
    - Wait ... That shows that every other NP-complete problem reduces to ours — how do we know that ours reduce to either one of the other NP-complete problems?

NP-Completeness

- NP-complete problems:
  - How do we prove that a given problem is NP-complete?
    - Turns out that this detail has already been taken care of (that's how we know about this class of NP-complete problems to begin with!)
    - So, we really only need to show that some problem known to be NP-complete reduces to ours in polynomial time, and that proves that ours is also NP-complete!

NP-Completeness

- NP-complete problems:
  - We'll complete this discussion next class — we'll go over the circuit satisfiability problem, which was the first one to be discovered to be NP-complete, and the one that "closes the loop" in our argument of "every one reduces to every other one"
  - We'll also look at a few of the "classic" NP-complete problems and a couple of the reductions that prove their NP-completeness.