NP Completeness

Tao Hou

Problems vs. Algorithms

- So far, our study has been focusing on ***algorithms***, which are used to solve ***problems***
  - Examples are merge sort, quick sort, and insertion sort, which are algorithms solving the ***sorting*** problem

Problems vs. Algorithms

- So far, our study has been focusing on ***algorithms***, which are used to solve ***problems***
  - Examples are merge sort, quick sort, and insertion sort, which are algorithms solving the ***sorting*** problem

- In this topic, we are turning our attention to ***problems*** themselves

Problems vs. Algorithms

- So far, our study has been focusing on ***algorithms***, which are used to solve ***problems***
  - Examples are merge sort, quick sort, and insertion sort, which are algorithms solving the ***sorting*** problem
- In this topic, we are turning our attention to ***problems*** themselves
- Previously, for different problems we consider, we were trying to show how we ***can*** design efficient algorithms for solving them

Problems vs. Algorithms

- So far, our study has been focusing on ***algorithms***, which are used to solve ***problems***
  - Examples are merge sort, quick sort, and insertion sort, which are algorithms solving the ***sorting*** problem
- In this topic, we are turning our attention to ***problems*** themselves
- Previously, for different problems we consider, we were trying to show how we ***can*** design efficient algorithms for solving them
- Now, we are going to show for certain problems, how we ***cannot*** design efficient algorithms

Hard vs. Easy Problems

- "Efficient" algorithms: ***polynomial-time*** algorithms, i.e., ones with worst-case time complexity being $O(n^k)$ for some constant k
- You might wonder whether all problems can be solved in polynomial time — the answer is ***no***:

Hard vs. Easy Problems

- "Efficient" algorithms: ***polynomial-time*** algorithms, i.e., ones with worst-case time complexity being $O(n^k)$ for some constant k
- You might wonder whether all problems can be solved in polynomial time — the answer is ***no***:
  - There are problems, such as Turing's famous "Halting Problem", that cannot be solved by any computer in ***finite*** steps

Hard vs. Easy Problems

- "Efficient" algorithms: ***polynomial-time*** algorithms, i.e., ones with worst-case time complexity being $O(n^k)$ for some constant k
- You might wonder whether all problems can be solved in polynomial time — the answer is ***no***:
  - There are problems, such as Turing's famous "Halting Problem", that cannot be solved by any computer in ***finite*** steps
  - There are also problems that can be solved in finite steps (e.g., in exponential time), but not in polynomial time

Hard vs. Easy Problems

- "Efficient" algorithms: ***polynomial-time*** algorithms, i.e., ones with worst-case time complexity being $O(n^k)$ for some constant k
- You might wonder whether all problems can be solved in polynomial time — the answer is ***no***:
  - There are problems, such as Turing's famous "Halting Problem", that cannot be solved by any computer in ***finite*** steps
  - There are also problems that can be solved in finite steps (e.g., in exponential time), but not in polynomial time
- Generally, we think of problems that are solvable by polynomial-time algorithms as being ***tractable***, or "easy", and problems that require superpolynomial time as being ***intractable***, or "hard"

NP-Complete problems

- The subject of this chapter, however, is an interesting class of problems called the "NP-complete" problems, whose status is unknown:
  - No polynomial-time algorithm have ever been discovered for an NP-complete problem, nor has anyone been able to prove that no polynomial-time algorithm can exist for any one of them

- The subject of this chapter, however, is an interesting class of problems called the "NP-complete" problems, whose status is unknown:
  - No polynomial-time algorithm have ever been discovered for an NP-complete problem, nor has anyone been able to prove that no polynomial-time algorithm can exist for any one of them
- This so-called $P \neq NP$ question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971

- The subject of this chapter, however, is an interesting class of problems called the "NP-complete" problems, whose status is unknown:
  - No polynomial-time algorithm have ever been discovered for an NP-complete problem, nor has anyone been able to prove that no polynomial-time algorithm can exist for any one of them
- This so-called $P \neq NP$ question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971
- Notice that NP-complete problems are still in general considered "hard" problems as most people believe you cannot find polynomial time algorithms for them

Similar problems could have completely different hardness

- Whether a problem is hard or easy can be very elusive
- Two problems could look very ‘similar’ on the surface, with one being polynomially-solvable and another being NP-complete

Similar problems could have completely different hardness

- Whether a problem is hard or easy can be very elusive
- Two problems could look very ‘similar’ on the surface, with one being polynomially-solvable and another being NP-complete

Ex.: ***Shortest*** vs. ***longest*** simple paths

- Finding shortest paths: polynomially-solvable
- Finding longest paths: NP-complete

Similar problems could have completely different hardness

- Whether a problem is hard or easy can be very elusive
- Two problems could look very 'similar' on the surface, with one being polynomially-solvable and another being NP-complete

Ex.: ***Shortest*** vs. ***longest*** simple paths

- Finding shortest paths: polynomially-solvable
- Finding longest paths: NP-complete

Euler tour vs. Hamiltonian cycle:

- An ***Euler tour*** of an undirected graph is a cycle that traverses each edge of the graph exactly once (it is allowed to visit vertices more than once)
- A ***Hamiltonian cycle*** of an undirected graph is a simple cycle that traverses each vertex exactly once
- We can determine whether a graph has an Euler tour in only $O(|E|)$ time
- Determining whether an undirected graph has a Hamiltonian cycle is NP-complete

Throughout this topic, we shall consider three classes of problems:

- ***P***: Problems that are solvable in polynomial time
- ***NP***: Problems that are “verifiable” in polynomial time
  - We have $P \subseteq NP$
- ***NPC***: NP-complete problems, those problems that are as hard as ***any*** problems in the class *NP* (you can also say is the hardest problem in *NP*)

Why do we study it?

Understanding NP-completeness theory is critical to algorithm designers:

- If you can find that a problem is NP-complete, you would then do better to spend your time finding an approximation algorithm or solving a tractable special case
- Many natural and interesting problems that on the surface seem no harder than sorting, graph searching, or network flow are in fact NP-complete

Why do we care polynomial time?

We focus on polynomial time algorithms for certain reasons:

- Although a polynomial running time of $\Theta(n^{100})$ is completely disastrous in practice, the polynomial-time algorithms we actually encountered ***typically require much less time***
- Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, ***more efficient algorithms often follow***

Why do we care polynomial time?

We focus on polynomial time algorithms for certain reasons:

- Although a polynomial running time of $\Theta(n^{100})$ is completely disastrous in practice, the polynomial-time algorithms we actually encountered ***typically require much less time***
- Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, ***more efficient algorithms often follow***
- For many ***different machine models***, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another
  - E.g., a "serial random-access machine" model we typically assume, or an "abstract Turing machine" model

Why do we care polynomial time?

We focus on polynomial time algorithms for certain reasons:

- Although a polynomial running time of $\Theta(n^{100})$ is completely disastrous in practice, the polynomial-time algorithms we actually encountered ***typically require much less time***
- Experience has shown that once the first polynomial-time algorithm for a problem has been discovered, ***more efficient algorithms often follow***
- For many ***different machine models***, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another
  - E.g., a "serial random-access machine" model we typically assume, or an "abstract Turing machine" model
- Most importantly, the class of polynomial-time solvable problems has nice ***closure properties***, since polynomials are closed under addition, multiplication, and composition
  - E.g., if we apply a polynomial-time algorithm for polynomially many times, we still have a polynomial-time algorithm

Decision Problems vs. Optimization Problems

- Many problems of interest are ***optimization*** problems: each legal solution has an associated value, and we wish to find a legal solution with the best value
- (Example) SHORTEST-PATH-OPTMZ: given an undirected graph G and vertices u and v, we wish to find a path from u to v with the fewest edges

Decision Problems vs. Optimization Problems

- Many problems of interest are ***optimization*** problems: each legal solution has an associated value, and we wish to find a legal solution with the best value
- (Example) SHORTEST-PATH-OPTMZ: given an undirected graph G and vertices u and v, we wish to find a path from u to v with the fewest edges
- However, the theory of NP-Completeness focuses only on ***decision*** problems: given an input, a program should produce "yes" or "no"
- (Example) SHORTEST-PATH: given an undirected graph G, two vertices u and v, and ***an integer*** k, is there a path from u to v with $\leq k$ edges?

Why We Can Only Consider Decision Problems?

- For an optimization problem (which people usually care about in real life), there is typically a ***corresponding version of decision problem*** (e.g., SHORTEST-PATH-OPTMZ vs. SHORTEST-PATH)

Why We Can Only Consider Decision Problems?

- For an optimization problem (which people usually care about in real life), there is typically a ***corresponding version of decision problem*** (e.g., SHORTEST-PATH-OPTMZ vs. SHORTEST-PATH)

- The decision problem is typically "easier" than the optimization problem: we can ***simply invoke the algorithm for the optimization problem*** and have an algorithms for the decision problem
  - E.g., given an instance (G, u, v, k) of SHORTEST-PATH, we can find the length ℓ of the shortest path of (G, u, v) using an algorithm for the optimization problem, and then check if $k \geq \ell$

Why We Can Only Consider Decision Problems?

- For an optimization problem (which people usually care about in real life), there is typically a ***corresponding version of decision problem*** (e.g., SHORTEST-PATH-OPTMZ vs. SHORTEST-PATH)

- The decision problem is typically "easier" than the optimization problem: we can ***simply invoke the algorithm for the optimization problem*** and have an algorithms for the decision problem
  - E.g., given an instance (G, u, v, k) of SHORTEST-PATH, we can find the length ℓ of the shortest path of (G, u, v) using an algorithm for the optimization problem, and then check if $k \geq \ell$
  - Remark: This is indeed a first instance of "***reduction***" which we will study more in-depth later

Why We Can Only Consider Decision Problems?

- For an optimization problem (which people usually care about in real life), there is typically a ***corresponding version of decision problem*** (e.g., SHORTEST-PATH-OPTMZ vs. SHORTEST-PATH)

- The decision problem is typically "easier" than the optimization problem: we can ***simply invoke the algorithm for the optimization problem*** and have an algorithms for the decision problem
  - E.g., given an instance (G, u, v, k) of SHORTEST-PATH, we can find the length ℓ of the shortest path of (G, u, v) using an algorithm for the optimization problem, and then check if $k \geq \ell$
  - Remark: This is indeed a first instance of "***reduction***" which we will study more in-depth later

- Implication: if we show that a decision problem is "hard", we also show that ***its related optimization problem is hard***
  - If we can find a polynomial-time algorithm for the optimization problem, we can definitely find a polynomial-time algorithm for the decision problem

Why We Can Only Consider Decision Problems?

- For an optimization problem (which people usually care about in real life), there is typically a ***corresponding version of decision problem*** (e.g., SHORTEST-PATH-OPTMZ vs. SHORTEST-PATH)

- The decision problem is typically "easier" than the optimization problem: we can ***simply invoke the algorithm for the optimization problem*** and have an algorithms for the decision problem
  - E.g., given an instance (G, u, v, k) of SHORTEST-PATH, we can find the length ℓ of the shortest path of (G, u, v) using an algorithm for the optimization problem, and then check if $k \geq \ell$
  - Remark: This is indeed a first instance of "***reduction***" which we will study more in-depth later

- Implication: if we show that a decision problem is "hard", we also show that ***its related optimization problem is hard***
  - If we can find a polynomial-time algorithm for the optimization problem, we can definitely find a polynomial-time algorithm for the decision problem
  - Equivalently, if we ***cannot*** find a polynomial-time algorithm for the decision problem, we ***also cannot*** find a polynomial-time algorithm for the optimization problem

- Thus, though NP-completeness theory restricts attention to decision problems, it ***often has implications for optimization problems***

Formal Definition of *Problems*

Problem

A ***problem*** is an association of problem instances (inputs) and to their solutions

Problem

A ***problem*** is an association of problem instances (inputs) and to their solutions

(Example) SHORTEST-PATH-OPTMZ:

- An instance for the problem is a triple (G, u, v)
- A solution is a sequence of vertices in G encoding the shortest path from u to v, with perhaps the empty sequence denoting that no path exists
- Notice: a given problem instance may be associated with more than one solution

Formal Definition of *Problems*

Problem

A ***problem*** is an association of problem instances (inputs) and to their solutions

(Example) SHORTEST-PATH-OPTMZ:

- An instance for the problem is a triple (G, u, v)
- A solution is a sequence of vertices in G encoding the shortest path from u to v, with perhaps the empty sequence denoting that no path exists
- Notice: a given problem instance may be associated with more than one solution

Decision Problem

A ***decision problem*** is a problem where the associated solution of each instance is either "yes" or "no"

Problem

A ***problem*** is an association of problem instances (inputs) and to their solutions

(Example) SHORTEST-PATH-OPTMZ:

- An instance for the problem is a triple (G, u, v)
- A solution is a sequence of vertices in G encoding the shortest path from u to v, with perhaps the empty sequence denoting that no path exists
- Notice: a given problem instance may be associated with more than one solution

Decision Problem

A ***decision problem*** is a problem where the associated solution of each instance is either "yes" or "no"

(Example) SHORTEST-PATH:

- An instance for the problem is a triple (G, u, v, k)
- Return ...

Polynomial-time solvable problem

A problem is said to be ***polynomial-time solvable*** if there exist an algorithm such that:

- The algorithm returns a correct associated solution to each instance of the problem
- The algorithm finishes in $O(n^k)$ time for any instance of size k

Polynomial-time solvable problem

A problem is said to be ***polynomial-time solvable*** if there exist an algorithm such that:

- The algorithm returns a correct associated solution to each instance of the problem
- The algorithm finishes in $O(n^k)$ time for any instance of size k

Complexity class *P*

The complexity class ***P*** is the set of decision problems that are polynomial-time solvable

Consider two problems:

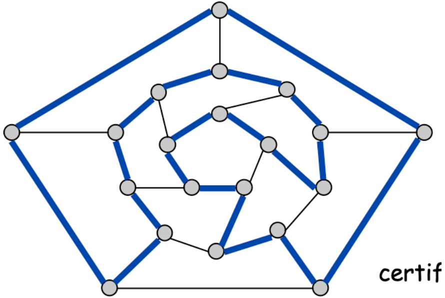
Hamiltonian cycle problem (HAM-CYCLE)

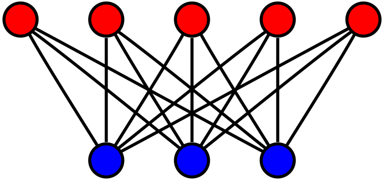
Given an undirected graph $G = (V, E)$, does G contain a ***Hamiltonian*** cycle, i.e., a simple cycle containing each vertex in V?

Consider two problems:

Hamiltonian cycle problem (HAM-CYCLE)

Given an undirected graph $G = (V, E)$, does G contain a ***Hamiltonian*** cycle, i.e., a simple cycle containing each vertex in V?





Left dodecahedron graph (taken from [K&T] slides) has a Hamiltonian cycle while the right bipartite graph (taken from Wikipedia) does not have one

Satisfiability problem (SAT)

Given a boolean formula with n boolean variables $x_1, \dots, x_n$, is there an assignment (of true/false values) to $x_1, \dots, x_n$ making the whole formula true?

Satisfiability problem (SAT)

Given a boolean formula with n boolean variables $x_1, \ldots, x_n$, is there an assignment (of true/false values) to $x_1, \ldots, x_n$ making the whole formula true?

A boolean formula is made up of the boolean variables $x_1, \ldots, x_n$, operators including $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\longrightarrow$ (implication), $\leftrightarrow$ (if and only if), and composite (combinations) of them possibly with parenthesis. E.g.,:

$$((x_1 \longrightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2.$$

- The two problems are “hard” to solve ***directly*** in the sense that no one on this earth has come up with a polynomial-time algorithm for them (although no one ever has proven otherwise)

Certificates and Verifiers

- The two problems are “hard” to solve ***directly*** in the sense that no one on this earth has come up with a polynomial-time algorithm for them (although no one ever has proven otherwise)
- Let’s consider an easier version for such a problem by solving it “***indirectly***” (HAM-CYCLE as an example)
- Suppose that someone tells you a given graph G is Hamiltonian and offers to prove it ***by giving you a sequence of vertices*** which this person claims to be a Hamiltonian cycle
- It would then be easy to ***verify*** this: simply verify whether the sequence contain all the vertices and whether each two consecutive vertices form an edge

Certificates and Verifiers

- The two problems are "hard" to solve ***directly*** in the sense that no one on this earth has come up with a polynomial-time algorithm for them (although no one ever has proven otherwise)
- Let's consider an easier version for such a problem by solving it "***indirectly***" (HAM-CYCLE as an example)
- Suppose that someone tells you a given graph G is Hamiltonian and offers to prove it ***by giving you a sequence of vertices*** which this person claims to be a Hamiltonian cycle
- It would then be easy to ***verify*** this: simply verify whether the sequence contain all the vertices and whether each two consecutive vertices form an edge
- The "verification" process can definitely be done in polynomial time in terms of the size of G
- Formally speaking, the algorithm used for the "verification" is termed as a ***verification algorithm***, and the sequence of vertices you used for verification is called a ***certificate***

Verification algorithm

For a problem Q, a ***verification algorithm*** (or simply ***verifier***), denoted $C(x, y)$, is an algorithm satisfying:

- $C(x, y)$ returns "yer"/"no"
- input x is an instance of Q
- input $y \in \{0, 1\}^*$ (a binary string) is a ***certificate***
- x is an "yes"-instance of $Q \Leftrightarrow$ there exist a certificate y making $C(x, y)$ return "yes"

Verification algorithm

For a problem Q, a ***verification algorithm*** (or simply ***verifier***), denoted $C(x, y)$, is an algorithm satisfying:

- $C(x, y)$ returns "yer"/"no"
- input x is an instance of Q
- input $y \in \{0, 1\}^*$ (a binary string) is a ***certificate***
- x is an "yes"-instance of $Q \Leftrightarrow$ there exist a certificate y making $C(x, y)$ return "yes"

Notice:

- For a "yes"-instance x, it's okay that $C(x, y)$ returns "no" given some certificate y
- As long as ***there is one certificate*** y making $C(x, y)$ return "yes", it is fine
- But if x is a "no"-instance, then $C(x, y)$ should return "no" ***for all certificates***

How to Write a Verifier

To write a verification algorithm $C(x, y)$ for a problem:

- You first "***decode***" the certificate y (which is a binary string) into something "meaningful"
- You then use it to verify whether y is a "yes"-instance

How to Write a Verifier

To write a verification algorithm $C(x, y)$ for a problem:

- You first "***decode***" the certificate y (which is a binary string) into something "meaningful"
- You then use it to verify whether y is a "yes"-instance
- Notice that in order to "decode" it, you have to design an "***encoding***" for the certificate
- When you try to decode y, if you cannot decode it into something meaningful, the verifier simply returns "no"

How to Write a Verifier

To write a verification algorithm $C(x, y)$ for a problem:

- You first "***decode***" the certificate y (which is a binary string) into something "meaningful"
- You then use it to verify whether y is a "yes"-instance
- Notice that in order to "decode" it, you have to design an "***encoding***" for the certificate
- When you try to decode y, if you cannot decode it into something meaningful, the verifier simply returns "no"

(Example) A verification algorithm $C(x, y)$ for SAT:

- Notice that here x is a boolean formula with boolean variables $x_1, \ldots, x_n$
- Given y as a bit string, decode y into a T/F assignment to $x_1, \ldots, x_n$
  - The simplest thing to do is to take the first n bits in y and take them as the T/F assignment to $x_1, \ldots, x_n$
  - If y has less than n bits, return "no"
- Then use the T/F assignment of $x_1, \ldots, x_n$ from the certificate y to verify whether the boolean formula evaluate to true; If true, return "yes"; otherwise, return "no"

(Example) A verification algorithm $C(x, y)$ for HAM-CYCLE:

- Given y as a bit string, decode y into a sequence of n vertices where n is the number of vertices in the input graph $G := x$
  - If y cannot be decoded into n vertices, return "no"
  - Ignore the remaining bits
- Then verify whether the n vertices form a valid Hamiltonian cycle

The Complexity Class ***NP***

The complexity class ***NP*** is a set of decision problems such that a problem $Q \in$ ***NP*** if and only if there is a verification algorithm $C(x, y)$ for Q running in polynomial time in term of the size of the Q's instance x

The Complexity Class ***NP***

The complexity class ***NP*** is a set of decision problems such that a problem $Q \in$ ***NP*** if and only if there is a verification algorithm $C(x, y)$ for Q running in polynomial time in term of the size of the Q's instance x

Example: SAT, HAM-CYCLE $\in$ ***NP***

Polynomial reduction

- There is one complexity class ***NPC*** which is yet to be defined
- Roughly saying, ***NPC*** is the set of “hardest” problems in ***NP***
- But to do that, we need a way to compare the “difficulty” of problems
- For that, we introduce the notion of “***reducibility***”, which is probably the single most important notion in the topic

Polynomial reduction

A decision problem Q_1 is said to ***polynomially reduces*** to (or simply ***reduces*** to) another decision problem Q_2 if there is a polynomial time ($O(|x_1|^k)$) algorithm $\mathcal{F}$ taking an instance x_1 for Q_1 and computing an instance x_2 for Q_2 such that:

- x_1 is a “yes”-instance for Q_1 iff x_2 is a “yes”-instance for Q_2

Polynomial reduction

A decision problem Q_1 is said to ***polynomially reduces*** to (or simply ***reduces*** to) another decision problem Q_2 if there is a polynomial time ($O(|x_1|^k)$) algorithm $\mathcal{F}$ taking an instance x_1 for Q_1 and computing an instance x_2 for Q_2 such that:

- x_1 is a "yes"-instance for Q_1 iff x_2 is a "yes"-instance for Q_2

(Implications) If Q_1 polynomially reduces to Q_2:

- Q_1 is "no harder" than Q_2 in the sense that
- Any polynomial-time algorithm $\mathcal{A}$ for Q_2 can be used to solve Q_1 in polynomial-time, by doing:
  - Given an instance x_1 of Q_1, use $\mathcal{F}$ to compute an instance x_2 of Q_2
  - Then use $\mathcal{A}$ to decide whether x_2 is a "yes"-instance for Q_2
  - Return "yes" if $\mathcal{A}$ returns "yes", and return "no" if $\mathcal{A}$ returns "no",
- So if Q_2 can be solved in polynomial time, then Q_1 also can

So whether a problem Q_1 reduces to another problem Q_2 completely relies on whether you can find a "reduction algorithm" $\mathcal{F}$

Example:

- Q_1: Given a string x_1, does x_1 contain the letter "a"?
- Q_2: Given a string x_2, does x_2 contain the letter "b"?

So whether a problem Q_1 reduces to another problem Q_2 completely relies on whether you can find a “reduction algorithm” $\mathcal{F}$

Example:

- Q_1: Given a string x_1, does x_1 contain the letter “a”?
- Q_2: Given a string x_2, does x_2 contain the letter “b”?
- Q_1 reduces to Q_2 with the reduction algorithm: given an instance x_1 of Q_2, replace each occurrence of “a” in x_1 with “b” and each occurrence of “b” in x_1 with “a” and produce an instance x_2 of Q_2

The Complexity Class ***NPC***

- A problem $Q \in$ ***NP*** is called ***NP-Complete*** if *all* problems in ***NP*** reduce to Q
- The complexity class ***NPC*** $\subseteq$ ***NP*** is the set of all NP-Complete problems

The Complexity Class ***NPC***

- A problem $Q \in$ ***NP*** is called ***NP-Complete*** if *all* problems in ***NP*** reduce to Q
- The complexity class ***NPC*** $\subseteq$ ***NP*** is the set of all NP-Complete problems

Proposition

P $\subseteq$ ***NP***

The Complexity Class ***NPC***

- A problem $Q \in$ ***NP*** is called ***NP-Complete*** if *all* problems in ***NP*** reduce to Q
- The complexity class ***NPC*** $\subseteq$ ***NP*** is the set of all NP-Complete problems

Proposition

P $\subseteq$ ***NP***

proof:

- Let Q be a problem in ***P***
- Then there is an algorithm $\mathcal{A}$ solving Q in polynomial time

The Complexity Class ***NPC***

- A problem $Q \in$ ***NP*** is called ***NP-Complete*** if *all* problems in ***NP*** reduce to Q
- The complexity class ***NPC*** $\subseteq$ ***NP*** is the set of all NP-Complete problems

Proposition

P $\subseteq$ ***NP***

proof:

- Let Q be a problem in ***P***
- Then there is an algorithm $\mathcal{A}$ solving Q in polynomial time
- To show that $Q \in$ ***NP***, we only need to design a polynomial-time verifier $\mathcal{C}(x, y)$ for Q
- To do this, in $\mathcal{C}$, we only need to invoke $\mathcal{A}$ on x, and return the answer of $\mathcal{A}$ (certificate y is completely ignored)

Is $P = NP$?

- The biggest question in CS: Is $P = NP$?
- This question was raised in the 1970's, and there is not an answer till this day
- The common belief is that $P \, ! = NP$
- The key lies in those NP-Complete problems, because if you can find an algorithm for a single NP-Complete problem, then all problems in NP, including all the other NP-Complete problems, can be solved in polynomial time (so $P = NP$)
- However, there are tons of NP-Complete problems out there, and ***no one*** has ever found a polynomial-time algorithm ***for any of them*** till this day

Proposition

If an NP-Complete problem can be solve in polynomial time, then all problems in NP can be solved in polynomial time (so $P = NP$)

Proposition

If an NP-Complete problem can be solve in polynomial time, then all problems in ***NP*** can be solved in polynomial time (so $\boldsymbol{P} = \boldsymbol{NP}$)

proof:

- Let Q be an NP-Complete problem and let $\mathcal{A}$ be a polynomial-time algorithm for Q
- Let Q' be an arbitrary problem in ***NP***
- Since Q' reduces to Q, we have a polynomial-time reduction algorithm $\mathcal{F}$ from Q' to Q
- Then we can have a polynomial-time algorithm for Q': given an instance x' of Q', compute an instance x of Q using the algorithm $\mathcal{F}$, then you just return whatever $\mathcal{A}$ returns on x

The complexity class *EXP*

- ***P***: Decision problems for which there is a polynomial-time algorithm
- ***NP***: Decision problems for which there is a polynomial-time verifier

The complexity class *EXP*

- ***P***: Decision problems for which there is a polynomial-time algorithm
- ***NP***: Decision problems for which there is a polynomial-time verifier
- ***EXP***: Decision problems for which there is an ***exponential-time*** algorithm

The complexity class *EXP*

- ***P***: Decision problems for which there is a polynomial-time algorithm
- ***NP***: Decision problems for which there is a polynomial-time verifier
- ***EXP***: Decision problems for which there is an ***exponential-time*** algorithm

Proposition

NP $\subseteq$ ***EXP***

The complexity class *EXP*

- ***P***: Decision problems for which there is a polynomial-time algorithm
- ***NP***: Decision problems for which there is a polynomial-time verifier
- ***EXP***: Decision problems for which there is an ***exponential-time*** algorithm

Proposition

NP $\subseteq$ ***EXP***

proof:

- Let Q be a problem in ***NP*** with a verifier $C(x, y)$
- Given an instance x of Q, we enumerate all possible certificates for x and see whether there is one certificate y making $C(x, y)$ return "yes"
- If there is such a certificate, we return "yes" for x, otherwise, we return "no"
- Notice that we only need to enumerate certificates of up to a size $m = poly(|x|)$, because any certificate beyond size m will not be helpful to us (see the "decoding" process for certificates)
- Total time would be $2^{poly(|x|)} f(|x|)$, where $f(|x|)$ is the time complexity of $C(x, y)$

The million-dollar question

(Figure from [K&T] slides)

Some remarks

- The term "NP" does not stand for "non-polynomial time" (we don't know whether these problems can or cannot be solved in polynomial time)
- It stands for "non-deterministic polynomial-time solvable" (the verification algorithm we write is indeed a "non-deterministic algorithm")
- As mentioned, if you have shown that a problem is NP-complete, then you should do something else rather trying to find a polynomial-time algorithm for it
- But people on this planet ***haven't proved*** that an NP-Complete problem does not have a polynomial-time algorithm
- Only that people believe so because there are tons of NP-Complete problems and no one has ever found a polynomial-time algorithm for ***any*** of them

How to show that a problem is NP-Complete?

How to show that a problem Q is NP-Complete?

1. Show that $Q \in$ ***NP*** by providing a polynomial-time verifier
2. Show that Q is "***NP-hard***", i.e., all problems in ***NP*** reduce to Q

How to show that a problem is NP-Complete?

How to show that a problem Q is NP-Complete?

1. Show that $Q \in$ ***NP*** by providing a polynomial-time verifier
2. Show that Q is "***NP-hard***", i.e., all problems in ***NP*** reduce to Q

- To do that, you need a "first" NP-hard problem Q^*
- Then you only need to show that Q^* reduces to Q
  - Due to the *transitivity* of reducibilities (Q_1 reduces to Q_2, Q_2 reduces to $Q_3 \Rightarrow Q_1$ reduces to Q_3)

How to show that a problem is NP-Complete?

How to show that a problem Q is NP-Complete?

1. Show that $Q \in$ ***NP*** by providing a polynomial-time verifier
2. Show that Q is "***NP-hard***", i.e., all problems in ***NP*** reduce to Q

- To do that, you need a "first" NP-hard problem Q^*
- Then you only need to show that Q^* reduces to Q
  - Due to the *transitivity* of reducibilities (Q_1 reduces to Q_2, Q_2 reduces to $Q_3 \Rightarrow Q_1$ reduces to Q_3)

- So the general strategy for showing a problem Q to be NP-hard is to first find a problem Q^* known to be NP-hard, and then reduce Q^* to Q