Great Ideas in Computing

University of Toronto CSC196
Fall 2025

Class 14: October 22 (2025)

Announcements and Agenda

Announcements

- Next week is reading week.
- Our second guest lecture will be given this Froday, October 24 by Professor Chris Maddison on the subject of quantum computers.
- I have posted the first two questions for Assignment 3. Hopefully by the end of today's class , these will be understandable questions.

Today's Agenda

- Polynomial time Turing reductions and polynomial time transformations. NP complete decision problems. The $P \neq NP$ conjecture and its importance. Examples of NP complete sets. The Karp tree of polynomial time transformations. The transformation 3-SAT $\leq^{poly}_{trans}$ Independent Set.

Two types of reductions

The general version of reduction $A \leq_T^{poly} B$ means that there is a poly time algorithm ALG that can call a subroutine for B (as often as it likes) and ALG computes A. Here we count each call to the subroutine as 1 step. It is not difficult to see that if $A \leq_T^{poly} B$ and B is computable in polynomial time, then A is computable in polynomial time.

The $\leq_T^{poly}$ reduction is what Cook used in his seminal 1971 paper.

The more restricted transformation (which we call a polynomial time transformation) $A \leq_{trans}^{poly} B$ means that there is a polynomial time function h (transforming an input instance of A to an input instance of B) such that $x \in A$ if and only if $h(x) \in B$. Note that $|h(x)| \leq p(|x|)$ for some polynomial p. Why?

Two types of reductions

The general version of reduction $A \leq_T^{poly} B$ means that there is a poly time algorithm ALG that can call a subroutine for B (as often as it likes) and ALG computes A. Here we count each call to the subroutine as 1 step. It is not difficult to see that if $A \leq_T^{poly} B$ and B is computable in polynomial time, then A is computable in polynomial time.

The $\leq_T^{poly}$ reduction is what Cook used in his seminal 1971 paper.

The more restricted transformation (which we call a polynomial time transformation) $A \leq_{trans}^{poly} B$ means that there is a polynomial time function h (transforming an input instance of A to an input instance of B) such that $x \in A$ if and only if $h(x) \in B$. Note that $|h(x)| \leq p(|x|)$ for some polynomial p. Why?
It is easy to see that $A \leq_{trans}^{poly} B$ and $B \in P$ implies $A \in P$.

Following Cook's paper, Karp provided a list of 21 combinatorial and graph theoretical problems that are NP complete. Karp used the more restrictive $\leq_{trans}^{poly}$. If you like names associated with these reductions then we can denote $\leq_T^{poly}$ as $\leq_{Cook}$ and $\leq_{trans}^{poly}$ as $\leq_{Karp}$.

NP-completeness wrt reductions $\leq_T^{poly}$ and $\leq_{trans}^{poly}$

Let's first explicitly give the definition NP-complete.

Definition: A language (or decision problem) L is NP complete if

1. $L \in NP$.
2. L is NP-hard *with respect to some polynomial time reduction*, for example with respect to either $\leq_T^{poly}$, or $\leq_{trans}$ *poly*. That is, if we are using $\leq_{trans}^{poly}$, then L is NP-hard if for every $A \in NP$, there is a polynomial time computable function h such that $w \in A$ if and only if $h(w) \in L$.

It is not difficult to show :

Fact: $B \in NP$ and $A \leq_{trans}^{poly} B$ implies $A \in NP$.

However, we do not believe that $B \in NP$ and $A \leq_T^{poly} B$ implies $A \in NP$. That is, we do not believe that the class NP is closed under $\leq_T^{poly}$ but is provably closed under $\leq_{trans}^{poly}$.

If I do not say otherwise, when only considering decision problems, I will use the more restrictive $\leq_{trans}^{poly}$.

The importance of NP-completeness

To personalize the notation, we can sometimes use $\leq_{Cook}$ (respectively, $\leq_{Karp}$) for general poly time reductions (respectively, polynomial time transformations).

Basic Fact:

One thing we want in any science (or engineering) is for concepts to be related and organized so as to understand things better. We also wants concepts to have impact and insight.

This is what NP completeness gives us. At some level of generality, we can understand an important class of problems (almost all combinatorial decision and optimization problems, many problems in number theory, and much more) in terms of one NP complete problem. We have:

If L is NP-complete (wrt to either $\leq_{Cook}$ or $\leq_{Karp}$), then $L \in P$ if and only if $P = NP$.

Some examples of *NP* complete decision problems

In our examples we always assume some natural way to represent the inputs as strings over some finite alphabet. In particular, integers are represented in say binary or decimal. Polynomial time means time bounded by a polynomial $p(n)$ where n is the length of the input string.

I will explain each of the following decision problems as we introduce them. Some problems are naturally decision problems. Others are decision variants of optimization problems and other relations or functions. Each of these decision problems are easily seen to be in *NP* (i.e. it is easy to provide a verification proedicate and succinct certificate). We will soon define completeness and indicate why each of these problems is *NP* complete.

- L_{HC} as defined previously; i.e., the set of graphs that have a Hamiltonian cycle.
- $SAT = \{F | F$ is a propositional formula that is *satisfiable*$\}$
- $PARTITION = \{(a_1, a_2, \ldots, a_n) | \exists S : \sum_{a_i \in S} a_i = \frac{1}{2} \sum_{i=1}^{n} a_i\}$
- $VERTEX\text{-}COLOUR = \{(G, k) | G$ can be vertex coloured with k colours$\}$

A example of a language in NP language that is believed to not be NP complete and believed to not be in P

$FACTOR = \{(N, k) | N \text{ is an integer that has a proper factor } m \leq k\}$

It is not difficult to see that $FACTOR$ is in NP.

Suppose $FACTOR \in P$. Can you then see how to factor a number N (i.e. provide the prime factorization) in polynomial time? That is, in time $p(n)$ for some polynomial P where $n = \log N$ is the length of the encoding.

We may have mentioned before that it is widely believed that we cannot factor integers in polynomial time and we use that assumption for some cryptographic applications.

The problem of efficiently factoring goes back at least two centuries to Gauss. We will soon see some evidence that $FACTOR$ is not complete.
Note: To determine if an interger is prime or composite is computable in polynomial time while we believe FACTOR is not computable in polynomial time.

NP hardness

A decison problem (or any problem, like say an optimization problem) F is NP-hard if *every* problem $L' \in NP$ (equivalently, any NP complete decision problem) can be "efficicently reduced" to F. While for decidsion problems $\leq_{trans}^{poly}$ is usually sufficent, we will need $\leq_{T}^{poly}$ when L is not a decision problem.

A decision problem L is NP-complete if it is both in NP and NP-hard. **Here again are the immediate consequences of a problem being NP-complete.**

- If L is NP complete, and $L \in P$, then every $L' \in NP$ is in P
- Equivalently, if any $L' \in NP$ is not in P, then every NP-complete problem is not in P.
- There are hundreds (and really thousands) of problems that are NP-complete and since we *"religously"* believe $P \neq NP$, we believe that none of these complete problems can be decided in polynomial time. I emphasize that this is in terms of *worst case* complexity and optmality for optimization problems.

Why the religious belief

Why do we believe so strongly that $P \neq NP$. It is simply that many very talented people over literally centuries have tried to efficiently solve problems that are in NP (and believed to not be in P and especially those that are NP-complete) and failed to do so.

Even so, there have been surprises in complexity theory and one still has to keep in mind that $P \neq NP$ is still a conjecture and not a proven result.

Our confidence in this conjecture is strong enough that modern day cryptography makes this assumption and indeed makes even stronger assumptions. For example, cryptographic protocols usually assume that there exist *one-way functions* f for which it is easy (i.e. poly time) to compute $f(x)$ for any x but given y, it is difficult to find an x such that $f(x) = y$.

For cryptography we also need assurance that a problem is not only hard in a worst case sense but also hard in some "average case" sense.

What would happen if someone solves the P vs NP question?

A frequent question that is asked is the following: What would be the consequences if someone resolves the P vs NP question

What would happen if someone solves the P vs NP question?

A frequent question that is asked is the following: What would be the consequences if someone resolves the P vs NP question

While the mathematical and scientfiic impact will be enormous, mathematics and science will not end.

If someone proves that (as we do not believe) $P = NP$, then the "practical impact" will depend on how efficiently we can solve NP complete problems; that is, what are the polynomial time bounds.

If someone proves $P \neq NP$, then the "practical impact" will depend on whether or not a given problem can be solved efficiently "in practice" (i.e. for most inputs or for "the inputs we care about"). More about worst case vs "practical application" later.

But how do we prove that a decision problem is *NP*-complete?

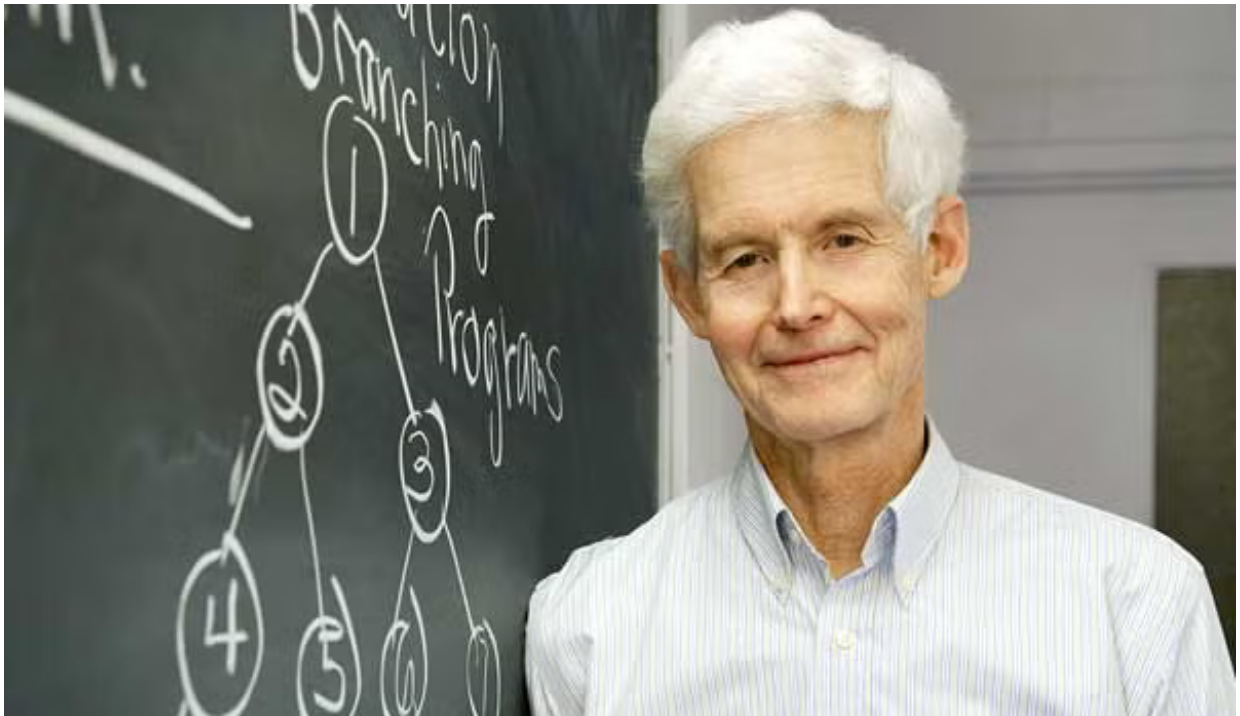
Suppose we know that some problem (for example, *SAT*) is *NP*-complete. Then if we can show *SAT* can be poly time reduced or transformed to (for example) *VC* = *vertex-cover*, then *VC* must also be *NP*-complete.

Fact: Polytime reductions and polytime transformations are transitive relations. That is, for example, $A \leq_{Karp} B$ and $B \leq_{Karp} C$ implies $A \leq_{Karp} C$.

In this way, thousands of decision problems *L* have been created by a tree of polynomial time transformations. We will show a portion of Karp's tree of *NP* complete problems. But we have to start the tree with some *NP* problem that we prove is *NP*-complete. This "root problem" will be the *SAT* problem which was the original problem Cook that showed was *NP*-complete. He also showed that the *IndependentSet* (equivalently, the *Clique* problem) was *NP* complete by reducing *SAT* to *IndependentSet*.

Showing *SAT* is *NP* complete

Cook showed that *SAT* is *NP* complete by showing how to efficiently encode any polynomial time Turing machine computation of a verification predicate $R(x, y)$ and a "guess" for a certificate y within propositional logic. (We usually discuss this in more detail in CSC373.) Recall what we said about validity in predicate calculus (1st order logic).
Steve Cook won the Turing Award in 1982.



A simple polynomial time transformation

Let $G = (V, E)$ be a graph. An *independent set* of nodes in G is a subset $I \subseteq V$ if for all $u, v \in I$, $(u, v) \notin E$.

A *clique* in G is a subset $K \subseteq V$ if for all $u, v \in K$, $(u, v) \in E$.

We will let $\bar{S}$ for a set of edges $S \subseteq E$ denote the complement of the set. That is $(u, v) \in S$ if and only if $(u, v) \notin S$.

NOTE: I is an independent set in G if and only if $\bar{I}$ is a clique in G.

The Independent Set and Clique problems

The Independent Set (IS) problem is the decision problem for the language
$L = \{(G, k)\}$: there exists an independent set I in G such that $|I| = k\}$

The Clique problem is the decision problem for the language
$L = \{(G, k)\}$: there exists a clique K in G such that $|K| = k\}$
Claim: IS $\leq^{poly}_{trans}$ Clique. Does anyone see why?

The Independent Set and Clique problems

The Independent Set (IS) problem is the decision problem for the language $L = \{(G, k)\}$: there exists an independent set I in G such that $|I| = k\}$

The Clique problem is the decision problem for the language $L = \{(G, k)\}$: there exists a clique K in G such that $|K| = k\}$

Claim: IS $\leq^{poly}_{trans}$ Clique. Does anyone see why?

For a graph $G = (V, E)$, we let $\bar{G}$ denote the complement graph $\bar{G} = (V, \bar{E})$. That is, $(u, v) \in E$ is an edge in G if and only if $(u, v) \notin \bar{G}$.

Then IS $\leq^{poly}_{trans}$ Clique by the transform h that converts (G, k) to $(\bar{G}, k)$. .

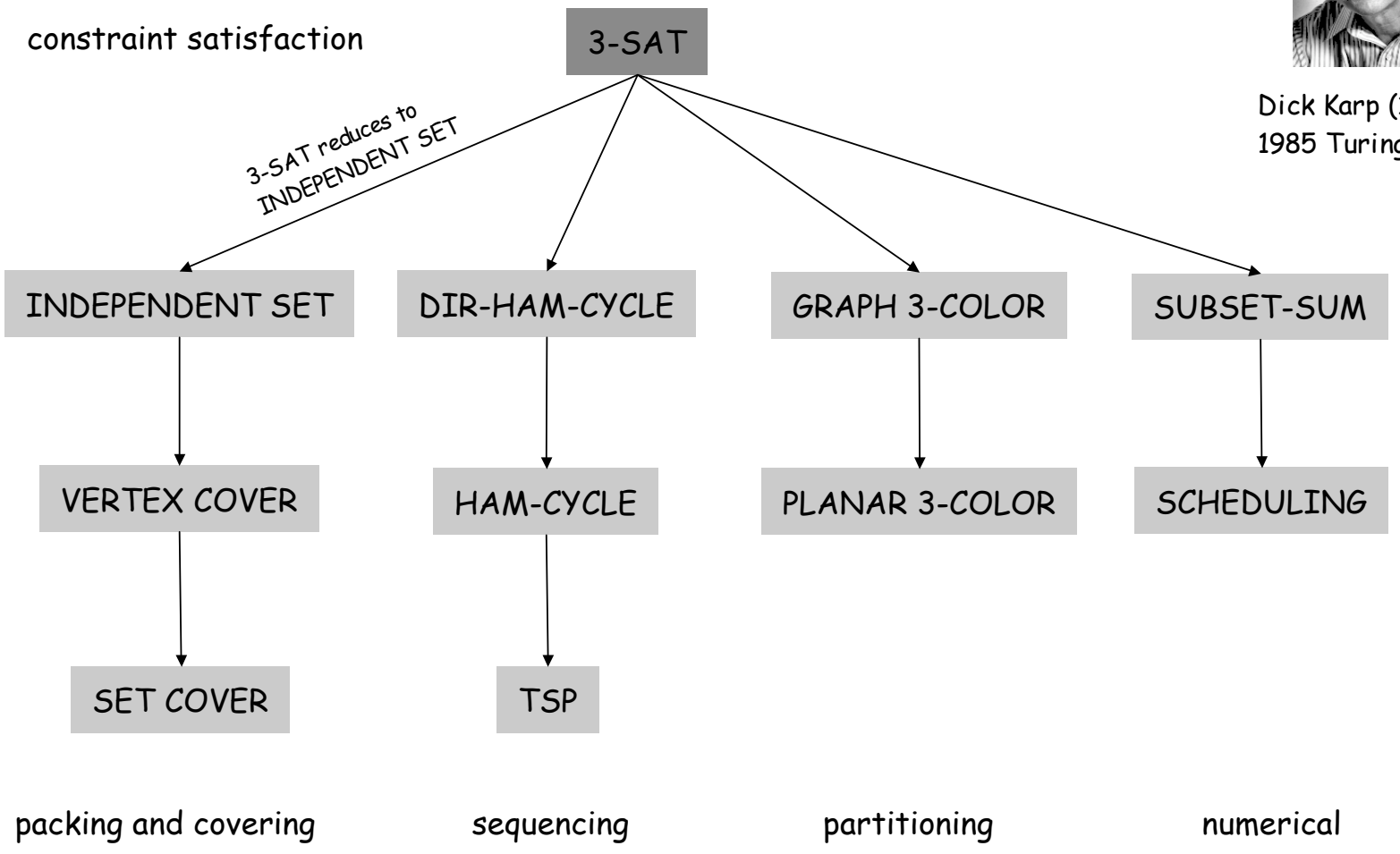
Clique $\leq^{poly}_{trans}$ IS by the same transformation (i.e., taking the complement of the graph.

It is very unusual that thret same transformation would work for both transformations. Some A $\leq^{poly}_{trans}$ B might be relatively easy while B $\leq^{poly}_{trans}$ A is a difficult tranformation (even when we know that such a transformation must exist). This is the case for some of the transformations in the Karp tree.

A tree of reductions/transformations

Polynomial-Time Reductions

Dick Karp (1972)
1985 Turing Award



Defining 3-SAT

I now have to define the problems listed in Karp's tree. I will just define a few of these problems. Today, I will just define 3-SAT and Independent Set.

I should then also define the transformations indicated by a $\rightarrow$ in the tree. Some of these transformations are relatively easy but some are complicated. Each transformation is given by a polynomial time transformation h that is tranasforming one problem into another.

Let's first define the 3-SAT and Independent Set problems.

3-SAT: A propositional formual F is in conjective normal form if F is presented as $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ where each C_j is called a *clause*. Each clause is a disjunction of literals where a literal is a propositional variable x or its complement $\bar{x}$.

Example: $F = (\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee \bar{x}_1)$.
This is equivalent to $(x_1 \rightarrow x_2) \wedge (x_2 \rightarrow x_3) \wedge (x_3 \rightarrow \bar{x}_1)$.

Continuing the definition of 3-SAT

A truth assignment is a function $\tau : \{x_i\} \to \{TRUE, FALSE\}$.
That is, eaach propositional variable is given a truth value.

Given a truth assignment τ, we can evaluate the formula for this truth assignment to obtain a value in $\{TRUE, FALSE\}$. We denote this by $F|_\tau$.

Consider the previous example: $F = (\bar{x}_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee \bar{x}_1)$.

If we set all propositional variables to $TRUE$, then the first and second clauses are $TRUE$ but the third clause is $FALSE$ and thus the formula is $FALSE$ when evaluated at the truth assignment that sets all variables to $TRUE$.

On the other hand, if we set $\tau(x_1) = FALSE, \tau(x_2) = \tau(x_3) = TRUE$ then $F|_\tau = TRUE$.

Finishing the definition of 3-SAT

In the previous example, every clause has exactly 2 literals. We call this an exact 2-SAT formula.

A 3-SAT formula is a formula in CNF in which every clause has at most 3 literal.

A formula F is satisfiable if there is at least one truth assignment for which $F|\tau = TRUE$.

The language 3-SAT $= \{$3-SAT satisfiable formulas.$\}$

The 3-SAT language (decision problem) is NP-complete.

However, the 2-SAT problem is computable in polynomial time.