Analysis of Algorithms

V. Adamchik CSCI 570 Fall 2017

Lecture 10 University of Southern California

NP Hardness

Outline

Intro to Turing Machines
Halting Problem
Reduction
Complexity Classes
Independent Set
Vertex Cover
Set Cover

23 Problems of Hilbert

In 1900 Hilbert presented a list of 23 challenging (unsolved) problems in math

#1 The Continuum Hypothesis	impossible, 1963
#8 The Riemann Hypothesis	unproved yet
#10 On solving a Diophantine equations	impossible, 1970
#18 The Kepler Conjecture	proved,1998

Hilbert's 10th problem

Given a multivariate polynomial with integer coeffs, e.g. $4x^2y^3 - 2x^4z^5 + x^8$, "*devise a process according to which it can be determined in a finite number of operations*" whether it has an integer root.

Mathematicians: we should try to formalize what counts as a '*process*' and an '*operation*'.

Hilbert's 10th problem

In 1928 Hilbert rephrased it as follows:

Given a statement in first-order logic, devise an "*effectively calculable procedure*" for determining if it's provable.

Mathematicians: we should try to formalize what counts as an '*calculable procedure*' (aka algorithm) and an 'efficient *calculable procedure*'.

Hilbert's conjecture

Hilbert conjectured that any mathematical proposition can be decided (proved true or false) by mechanistic logical methods.

In 1931 it was unexpectedly disproven by Gödel.

Gödel showed that for any formal theory, there will always be undecidable propositions.

Mathematicians started to look for practical techniques (computation) for proving undecidability.

Gödel (1934):
Discusses some ideas for definitions of what functions/languages are "computable", but isn't confident what's a good definition.

Church (1936):
Invents lambda calculus, claims it should be the definition of "computable".

Gödel and Post (1936):
Argue that Church's definition isn't justified.

Meanwhile… a certain British grad. student in Princeton, unaware of all these debates…

Described a new model of computation, now known as the Turing Machine.

PH.D. student of A. Church

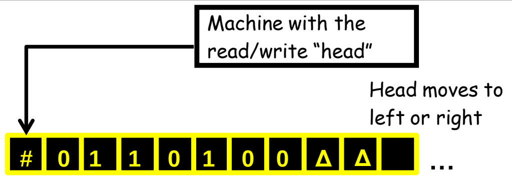
Alan Turing (1936, age 22)

Gödel, Kleene, and Church:
"Um, he nailed it. Game over, computation defined."

Turing Machines were adopted in the 1960's, years after his death.

Turing's Inspiration

Human writes symbols on paper
The paper is a sequence of squares
No upper bound on the number of squares
At most finitely many kinds of symbols
Human observes one square at a time
Human has only finitely many mental states
Human can change symbols and change
    focus to a neighboring square, but only
    based on its state and the symbol it observes
Human acts deterministically


Machine with the read/write "head"
Head moves to left or right
# 0 1 1 0 1 0 0 Δ Δ
…

input (the "tape"), indefinitely extensible to the right.

input consists of symbols from an alphabet Σ

# and Δ represent the start and end of the input

The machine never overwrites the leftmost symbol, but can overwrite the rightmost symbol.

When halt(accept/reject) state is reached, the machine halts. It might also never halt, in which case we say it loops.

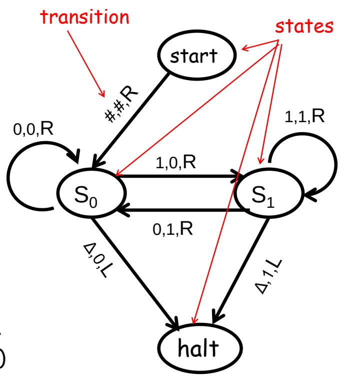
High Level Example of a Turing Machine

The machine that takes a binary string and appends 0 to the left side of the string.

Input: #10010Δ
Output: #**0**10010Δ

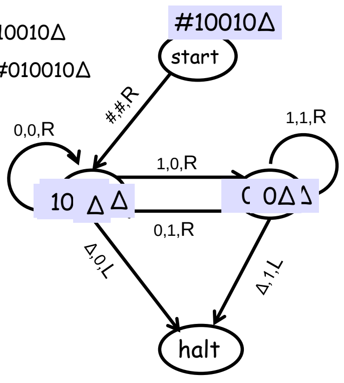
# - leftmost char
Δ – rightmost char

Transition on each edge
read,write,move (L or R)



Deterministic Turing Machine

Input: #10010Δ
Output: #010010Δ



Runtime Complexity

Let M be a Turing machine that halts on all inputs.

Assume we compute the running time purely as a function of the length of the input string.

Definition: The running complexity is the function $f: N \rightarrow N$ such that f(n) is the maximum number of steps that M uses on any input of length n.

Algorithms

The set Σ^* is the set of all finite sequences of elements of alphabet Σ.

A language $L \subseteq \Sigma^*$ is decidable if there is a Turing Machine M which halts on every input $x \in L$.

A problem P is *decidable* if it can be solved by a Turing machine T that always halt.

We say that P has an algorithm.

The Church-Turing Thesis:

"Any natural / reasonable notion of computation can be simulated by a TM."

This is not a theorem.

Is it… *…an observation?*

…a definition?

…a hypothesis?

…a law of nature?

…a philosophical statement?

Everyone believes it. No counterexample yet.

Hypercomputation

In 1995 Hava Siegelman proposed Artificial Recurrent Neural Networks (ARNN). She claims that ARNNs can "compute" non-computable functions.

In 2006 Martin Davis criticized the idea in a paper "The Myth of Hypercomputation."

Some people were disagree with Martin saying that it's a myth that hypercomputation is a myth.

Complexity Classes

A fundamental complexity class P (or PTIME) is a class of decision problems that can be solved by a deterministic Turing machine in polynomial time.

A fundamental complexity class EXPTIME is a class of decision problems that can be solved by a deterministic Turing machine in $O(2^{p(n)})$ time, where p(n) is a polynomial.

Nondeterministic Turing Machine

The deterministic Turing machine means that there is only one valid computation starting from any given input. A computation path is like a linked list.

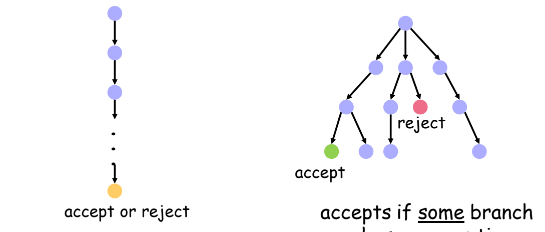
Nondeterministic TM defined in the same way as deterministic, except that a computation is like a tree, where at any state, it's allowed to have a number of choices.

The big advantage: it is able to try out many possible computations in parallel and to accept its input if any one of these computations accepts it.

Computations

deterministic computation

nondeterministic computation



accepts if <u>some</u> branch reaches an accepting configuration

Complexity Class: NP

A fundamental complexity class NP is a class of decision problems that can be solved by a <u>nondeterministic</u> Turing machine in polynomial time.

<u>Equivalently</u>, the NP decision problem has a certificate that can be checked by a polynomial time deterministic Turing machine.

We will be using the last definition for proving NP completeness.

P versus NP

It has been proven that Nondeterministic TM can be simulated by Deterministic TM.

But how <u>fast</u> we can do that simulation?

The famous P ≠ NP conjecture, would answer that we cannot hope to simulate nondeterministic Turing machines very fast (in polynomial time).

Is every language in decidable? Is every function computable?

Answer: No

Write a program to output "HELLO WORLD" on the screen and then terminate (halt).

The TA grading script G must be able to take any program P and grade it.

What kind of program could a student hand in?

```
while (P == NP)
   print "HELLO WORLD";
```

Despite the simplicity of the HELLO assignment, there is no program to correctly grade it!

And we will <u>prove</u> this.

Undecidable Problems

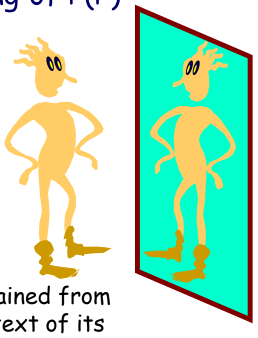
Undecidable means that there is no computer program that always gives the correct answer: it may give the wrong answer or run forever without giving any answer.

<u>The halting problem</u> is the problem of deciding whether a given Turing machine halts when presented with a given input.

Turing's Theorem:
The Halting Problem is not decidable.

The meaning of P(P)

P(x) means the output that arises from running program P on input x, assuming that P eventually halts.



P(P) means the output obtained from running program P on the text of its own source code.

The Halting Set K

Definition:

K is the set of all programs P such that P(P) halts.

K = { program P | P(P) halts }

Question:
Is there a program HALT such that:

HALT(P) = yes, if $P \in K$, so P(P) halts.
HALT(P) = no, if $P \notin K$, so P(P) doesn't halt.

The Halting Problem

Suppose a program HALT that solves the halting problem is indeed exist.

We will call HALT as a subroutine in a new program called CONFUSE.

```
bool CONFUSE(P) {
   if (HALT(P) == True)
         then loop forever;
   else return True;
}
```

Does CONFUSE(CONFUSE) halt?

Does CONFUSE(CONFUSE) halt?

```
bool CONFUSE(P) {
  if (HALT(P) == True) then loop forever;
  else return True;
}
```

Consider two cases:
1. assume CONFUSE(CONFUSE) does halt.

   by definition of HALT, we have that HALT(CONFUSE) is True.

   then by definition of CONFUSE, we have that CONFUSE(CONFUSE) loops forever.

Does CONFUSE(CONFUSE) halt?

```
bool CONFUSE(P) {
  if (HALT(P) == True) then loop forever;
  else return True;
}
```

Second case:
2. CONFUSE(CONFUSE) does not halt.

   by definition of HALT, we have that HALT(CONFUSE) is False.

   Then by definition of CONFUSE, we have that CONFUSE (CONFUSE) returns True.

Why is the Halting Problem so important?

Because a lot of really practical problems are the halting problem in disguise.

Reduction

Denoted by

$Y \leq_p X$

or

$Y \longrightarrow X$

Polynomial Reduction

To reduce problem Y to problem X (we write $Y \leq_p X$) we want a function f that maps Y to X such that:

1) f is a polynomial time computable
2) $y \in I_y$ (instance of Y) is solvable if and only if $f(y) \in I_x$ is solvable.

In plain form (use X to solve Y):

reduce an input of Y into an input of X,
solve X,
reduce the solution back to Y.

$Y \leq_p X$

If we can solve X in polynomial time, we can solve Y in polynomial time.

Examples:

Image Segmentation $\leq_p$ Min-Cut

Survey Design $\leq_p$ Max-Flow

$Y \leq_p X$

If we can solve X, we can solve Y.

Negate this statement.

If we cannot solve Y, we cannot solve X.

We use this to prove NP completeness: knowing that Y is hard, we prove that X harder.

In plain form: X is at least as hard as Y.

Note, we reduce TO the problem we want to show is the harder problem.

Two ways of using $Y \leq_p X$

1) X is easy
If we can solve X in polynomial time,
we can solve Y in polynomial time.

2) Y is hard
Then X is at least as hard as Y

P and NP

P = set of problems that can be solved in polynomial time by a deterministic TM.

NP = set of problems that can be solved in polynomial time by a nondeterministic TM.

NP = set of problems for which solution can be verified in polynomial time by a deterministic TM.

NP-Hard and NP-Complete

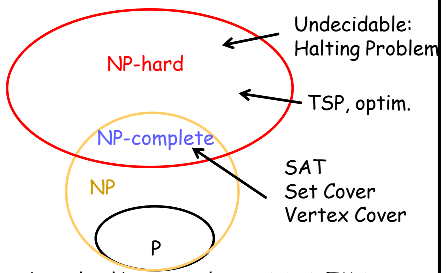
X is *NP-Hard*, if $\forall Y \in NP$ and $Y \leq_p X$.

X is *NP-Complete*, if X is NP-Hard and $X \in NP$.

Venn Diagram ($P \neq NP$)

NPH problems do not have to be in NP.

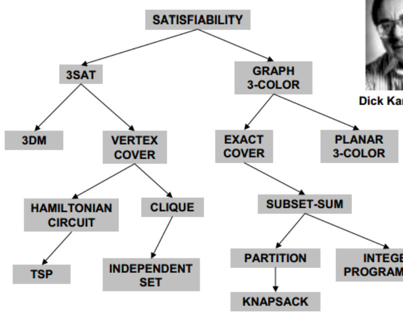
NPC problems are the most difficult NP problems.



NPC problems can be solved by a *nondeterministic* TM in polynomial time.

It's not known if NPC problems can be solved by a *deterministic* TM in polynomial time.

Reduction



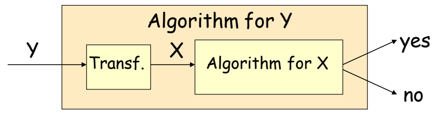
Dick Karp (1972)

Karp introduced the now standard methodology for proving problems to be NP-Complete.
He received a Turing Award for his work (1985).

NP-Completeness Proof Method

To show that X is NP-Complete:

1) Show that X is in NP
2) Pick a problem Y, known to be an NP-Complete (for example, Y is SAT)
3) Prove $Y \leq_p X$ (reduce Y to X)



Boolean Satisfiability Problem

A propositional logic formula is built from variables, operators AND (conjunction, $\wedge$), OR (disjunction, $\vee$), NOT (negation, $\neg$), and parentheses:

$$(X_1 \vee \neg X_3) \wedge (X_1 \vee \neg X_2 \vee X_4 \vee X_5) \wedge \ldots$$

A formula is said to be satisfiable if it can be made TRUE by assigning appropriate logical values (TRUE, FALSE) to its variables.

A formula is in conjunctive normal form (CNF) if it is a conjunction of clauses.
A literal is a variable or its negation.
A clause is a disjunction of literals.

Cook-Levin Theorem (1971)

Theorem. CNF SAT is NP-complete.

No proof…

Cook received a Turing Award for his work.

You are not responsible for knowing the proof.

Disjunctive Normal Form

A propositional logic formula that is a disjunction of conjunctions of literals:

$$(X_1 \wedge \neg X_3 \wedge X_4) \vee (\neg X_1 \wedge X_2 \wedge \neg X_5) \vee \ldots$$

Such a formula is satisfiable if and only if at least one of its clauses is satisfiable.

And a conjunction is satisfiable if and only if it does not contain both X and ¬ X for some variable X.

This can be checked in linear time.

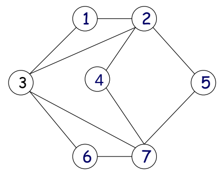
Independent Set

Given a graph, we say that a subset of vertices is "independent" if no two of them are joined by an edge.

{1,4}

{3,4,5}

{1,4,5,6}



Independent Set

Optimization Version:

Given a graph, find the largest independent set.

Decision Version:

Given a graph and a number k, does the graph contains an independent set of size at least k?

Optimization vs. Decision

Optimization vs. Decision

If one can solve an optimization problem (in polynomial time), then one can answer the decision version (in polynomial time)

Conversely, by doing binary search on the bound b, one can transform a polynomial time answer to a decision version into a polynomial time algorithm for the corresponding optimization problem

In that sense, these are essentially equivalent.

However, they belong to two different complexity classes.

Independent Set is NP Complete

Given a graph and a number k, does the graph contains an independent set of size at least k?

Is it in NP?

We need to show we can verify a solution in polynomial time.

Given a set of vertices, we can easily count them and then verify that any two of them are not joined by an edge.

Independent Set is NP Complete

Given a graph and a number k, does the graph contains an independent set of size at least k?

Is it in NP-hard?

We need to pick Y such that $Y \leq_p$ IndSet for $\forall Y \in$ NP

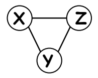
Reduce from 3-SAT.

3-SAT is SAT where each clause has at most 3 literals.

3SAT $\leq_p$ IndSet

We construct a graph G that will have an independent set of size k iff the 3-SAT instance with k clauses is satisfiable.

For each clause (X ∨ Y ∨ Z) we will be using a special gadget:



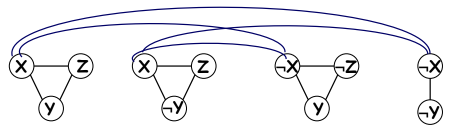
Next, we need to connect gadgets.

As an example, consider the following instance:

(X ∨ Y ∨ Z) ∧ (X ∨ ¬Y ∨ Z) ∧ (¬X ∨ Y ∨ ¬ Z) ∧ (¬X ∨ ¬Y)

3SAT $\leq_p$ IndSet

(X ∨ Y ∨ Z) ∧ (X ∨ ¬Y ∨ Z) ∧ (¬X ∨ Y ∨ ¬Z) ∧ (¬X ∨ ¬Y)



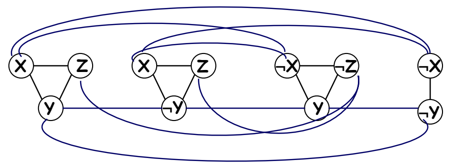
How do we connect gadgets?

X in one gadgets with ¬X in all other gadgets.

Do the same for Y and Z.

3SAT $\leq_p$ IndSet

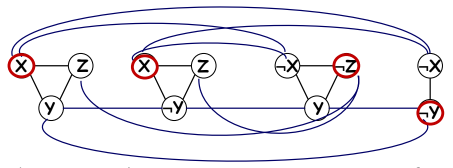
(X ∨ Y ∨ Z) ∧ (X ∨ ¬Y ∨ Z) ∧ (¬X ∨ Y ∨ ¬Z) ∧ (¬X ∨ ¬Y)



Claim: the 3-SAT instance with k clauses is satisfiable iff the graph G has an independent set of size k.

3SAT $\leq_p$ IndSet

(X ∨ Y ∨ Z) ∧ (X ∨ ¬Y ∨ Z) ∧ (¬X ∨ Y ∨ ¬Z) ∧ (¬X ∨ ¬Y)



⟹)Assume the 3-SAT instance is satisfiable

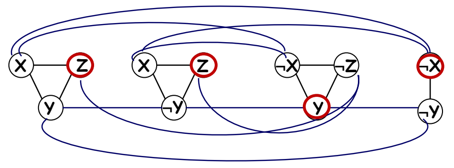
Then each triangle has a true literal.

For example, X, ¬Y and ¬Z are true.

Those true literals (one per gadget) make an independent set.

3SAT $\leq_p$ IndSet

(X ∨ Y ∨ Z) ∧ (X ∨ ¬Y ∨ Z) ∧ (¬X ∨ Y ∨ ¬Z) ∧ (¬X ∨ ¬Y)



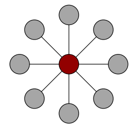
⟸)Assume G has an Ind. Set of size k.

If a vertex belongs to Ind. Set make it true.

Otherwise, it's false.

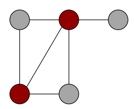
Vertex Cover

Given G=(V,E), find the smallest S⊆V s.t. every edge is incident to vertices in S.



Decision Version:

Given a graph and a number k, does the graph contains a vertex cover of size at most k?

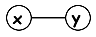


Vertex Cover

Theorem: *for a graph G=(V,E), S is a independent set if and only if V-S is a vertex cover*

$\Longrightarrow$)Suppose S is a independent set.

Consider an edge in G.



If $x \in S$, then $y \notin S$.	It follows, $y \in V-S$
If $y \in S$, then $x \notin S$.	It follows, $x \in V-S$
If $x \notin S$ and $y \notin S$.	It follows, both are in a vertex cover

Vertex Cover

Theorem: *for a graph G=(V,E), S is a independent set if and only if V-S is a vertex cover*

$\Longleftarrow$)Suppose V-S is a vertex cover.

Pick any x, y $\in S$

Then cannot be an edge between x and y.

Because otherwise, that edge wouldn't covered in V-S.

So, S is an independent set.

Vertex Cover in NP-Complete

Ind. Set $\le_p$ Vertex Cover

Claim: *a graph G=(V,E) has an independent set of size at least k if and only if G has a vertex cover of size at most V-k.*

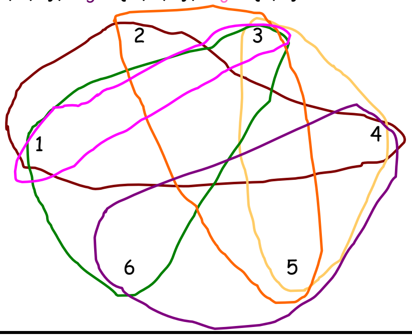
Proof is by the previous theorem.

Set Cover

Given a set U of elements and a collection $S_1, . . . , S_m$ of subsets of U, is there a collection of at most k of these sets whose union equals U?

Visualizing Set Cover

U = {1, …, 6}, S_1 = {1,2,4}, S_2 = {3,4,5}, S_3 = {1,3,6}, S_4 = {2,3,5}, S_5 = {4,5,6}, S_6 = {1,3}



Set Cover is NP complete

Is it in NP?

We need to show we can verify a solution in polynomial time.

Given a list of k sets from the given collection. We can check in polynomial time whether they cover all of U.

Is it in NP hard?

Vertex Cover $\le_p$ Set Cover

Given a vertex cover, create an instance of a set cover.

Define U as a set of all edges, U = E.

Define S_v as a set that contains all edges incident to vertex v.

This construction can be done in polynomial time.

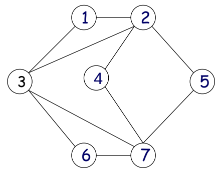
Vertex Cover $\le_p$ Set Cover

Define S_v as a set that contains all edges incident on vertex v.

$S_1=\{(1,2),(1,3)\}$

$S_2=\{(2,1),(2,3),(2,4),(2,5)\}$

and so on…



Claim: *a graph G=(V,E) has a vertex cover of size at most k if and only if U has a set cover of size at most k.*

Vertex Cover $\le_p$ Set Cover

Claim: *a graph G=(V,E) has a vertex cover of size at most k if and only if G has a set cover of size at most k.*

$\Longrightarrow$)Given a vertex cover of size at most k

In our construction consider sets S_v where v from the vertex cover. There are at most k of them.

We need to show that those S_v cover U=E.

Take any edge, at least one of its vertices must be in the vertex cover.

Thus, it will be covered by at least one set S_v.

Vertex Cover $\le_p$ Set Cover

Claim: *a graph G=(V,E) has a vertex cover of size at most k if and only if G has a set cover of size at most k.*

$\Longrightarrow$)Given a set cover of size at most k

In our construction, sets are naturally corresponds to vertices. Let a vertex cover be that set.

We need to show that those vertices cover E.

Take any edge, at least one of its vertices must be in the set cover.

Thus, the vertex cover will contain at least one of its endpoints.

Transitivity

If $Z \le_p Y$ and $Y \le_p X$, then $Z \le_p X$

We have showed today:

3SAT $\le_p$ IndSet $\le_p$ Vertex Cover $\le_p$ Set Cover

Set Cover is at least as hard as Vertex Cover which is at least as hard as Independent Set, which is at least as hard as SAT.

More reductions next time!