Lecture 14

Real Computers are Finite

Unbounded “memory” is critical to most undecidability pfs
Real computers are finite: \( n \) bits of state (registers, cache, RAM, HD, ...) \( \Rightarrow \leq 2^n \) configs – it's a DFA!

“Does M accept w” is decidable: run M on w; if it runs more that \( 2^n \) steps, it’s looping. (Recall LBA pfs.)

BUT:

\( 2^n \) is astronomical: a modest laptop has \( n = 100's \) of gigabits of state; # atoms in the universe \( \sim 2^{262} \)

Are “real” computer problems undecidable?

Options:

1. 100 G is so much >> 262, let’s say it’s approximately unbounded \( \Rightarrow \) undecidable
2. Explore/quantify the “computational difficulty” of solving the (decidable) “bounded memory” problem

1st is somewhat crude, but easy, and not crazy, given that we really don’t have methods that are fundamentally better for 100Gb memories than for arbitrary algorithms

2nd is more refined but harder; goal of next few weeks is to develop theory supporting such aims

Measuring “Compute Time”

TM: simple, just count steps
Defn: If M is a TM deciding \( L \), the time complexity of \( M \) is the function \( T(n) \) such that \( T(n) \) is the max number of steps taken by \( M \) on any input \( w \in \Sigma^n \) of length \( n \).

Why as a function of \( n \)? Mainly to smooth and summarize
Loosely, the time complexity of \( L \) is the least such \( T \) over all \( M \) deciding \( L \).

(I say “loosely” because it may be that no one \( M \) is fastest on all inputs, but nevertheless we may be able to bound it.)
Example: \( L = \{ a^n b^n \mid n \geq 0 \} \)
(on a One-Tape TM)

A simple algorithm (zig-zag, cross off letters): \( T(n) = n^2 \)
Somewhat trickier: cross off 5 letters at a time: \( T(n) = n^2/5 \)
A more complex algorithm:

- On a “two-track” tape, drag along a binary counter: \( T(n) = n \log_2 n \)

Slightly more work:
- As above, but a decimal counter: \( T(n) = n \log_{10} n \)

More work still:
- As above, but use lots of states to count off first 10 million a’s & b’s:
  \( T(n) = \text{if } (n < 10^7) \text{ then } n \text{ else } n \log_{10} n \)

One conclusion:
- Focus on growth rate, not const or small \( n \). I.e., big-O


Complexity Classes

Defn:
\( \text{TIME}(T(n)) = \text{the set of languages decidable by single-tape TMs in time } \mathcal{O}(T(n)) \)

E.g. \( \{ a^n b^n \mid n \geq 0 \} \in \text{TIME}(n \log n) \)

Example: \( L = \{ a^n b^n \mid n \geq 0 \} \)
(on a Two-Tape TM)

Counter on tape 2; +1 for every a; -1 for every b
Time: \( \mathcal{O}(n) \) – faster than best 1-tape TM for \( L \)

(Analysis is a bit subtle. “+1/-1” take \( \log n \) steps in worst case, but “carries/borrows” usually don’t propagate very far.
Can prove amortized cost of +1/-1 is only \( \mathcal{O}(1) \) per operation.)

One Conclusion: “Time” is somewhat technology-sensitive
(In fact, gap between 1 tape and 2 is quadratic: \( \{ w w \mid w \in \Sigma^* \} \))

“Tapes are Lame”

Obviously, “real” computers have essentially constant-time access to any bit of memory, not sequential access as on a tape
Fast “random access” will allow faster algorithms for many problems, so time on a TM may seem a poor surrogate for time on real computers

How poor?
A Model of a “Real Computer”

“Random Access Machines” (RAMs)
Memory is an array
Unit time access to any word
Basic, unit time ops like +, -, *, /, test-if-zero,…
Programs

For comparison to TMs, perhaps have read-only “input tape” or other string-oriented input convention and special “accept/reject” operations. Program typically not in memory (but could be)

Proof: look at your homework #1 and see how long your simulations took.

TM by RAM is quick

RAM by TM is slower, but cubic is conservative. In time T, the RAM can touch at most T memory words, each word holds at most T bits, it takes time at most $T^2$ to slog through tape to fetch/store a word, etc.

A Church-Turing thesis for “time”?

Church-Turing thesis: all “reasonable” models of computation are equivalent – i.e. all give the same set of decidable problems

“Extended” Church Turing thesis: All “reasonable” models of computation are equivalent up to a polynomial difference in time complexity

E.g. from above, this is true of deterministic singe- and multi-tape TMs and RAMs

More on what “reasonable” means later…

The class P

Definition:
$$P = \bigcup_{k=1}^{\infty} \text{TIME}(n^k)$$

I.e., the set of (decision) problems solvable by computers in polynomial time. I.e., $L \in P$ iff there is an algorithm deciding L in time $T(n) = O(n^k)$ for some fixed k (i.e., k is independent of the input).

Examples: sorting, shortest path, MST, connectivity, …
Why “Polynomial”?  

Point is not that $n^{2000}$ is a nice time bound, or that the differences among $n$ and $2n$ and $n^2$ are negligible.

Rather, simple theoretical tools may not easily capture such differences, whereas exponentials are qualitatively different from polynomials and may be amenable to theoretical analysis.

"My problem is in P" is a starting point for a more detailed analysis  
"My problem is not in P" may suggest that you need to shift to a more tractable variant

Another view of Poly vs Exp

Next year’s computer will be 2x faster. If I can solve problem of size $n_0$ today, how large a problem can I solve in the same time next year?

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Increase</th>
<th>E.g. $T=10^{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>$n_0 \rightarrow 2n_0$</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$n_0 \rightarrow \sqrt{2}n_0$</td>
<td>$10^6$</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>$n_0 \rightarrow \sqrt[3]{2}n_0$</td>
<td>$10^4$</td>
</tr>
<tr>
<td>$2^{n/10}$</td>
<td>$n_0 \rightarrow n_0 + 10$</td>
<td>400</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$n_0 \rightarrow n_0 + 1$</td>
<td>40</td>
</tr>
</tbody>
</table>
Some notes on HW #4

\[ \exists \text{ decidable } D \]
\[ \forall \text{ decidable } L \]
\[ L \leq_m D \]
\[ \overline{D} \rightarrow \overline{L} \]

\[ x_0 \notin D \]
\[ x_1 \in D \]
\[ \text{ runs } \text{ decider } \text{ for } L \]
\[ f(w) : \text{ reject } \text{ decider } L \]
\[ \text{ if } y \text{ output } x_1 \]
\[ \text{ else } x_0 \]

\[ A \preceq_m B \iff \overline{A} \preceq_m \overline{B} \]

2. A not T. dom.

Then \( B = \epsilon \).

Every TM meets time \( n + O(1) \).
More on P vs NP

Lecture 16

\[ P = \bigcup_k \text{TIME}(n^k) \]

\[ n \log n = O(n^2) \]

Given \( G, a, b \), 3-Path \( a \rightarrow b \)

Sorting is \( \text{TIME}(n \log n) \) complex.

\[ \text{Given nodes } A, B \]
\[ (A, B)_i \text{ is } c \]

\[ \text{Shortest Path} \]

\[ G, a, b, K \text{ shortest path of length } \leq K \]

CFL recognition \( O(n^3) \)
Complexity Classes

Defn:
TIME(T(n)) = the set of languages decidable by single-tape TMs in time O(T(n))

E.g. \{ a^n b^n | n \geq 0 \} \in TIME(n \log n)

The class P

Definition:
P = \bigcup_{k \geq 1} TIME(n^k)

I.e., the set of (decision) problems solvable by computers in polynomial time. I.e., \( L \in P \) iff there is an algorithm deciding \( L \) in time \( T(n) = O(n^k) \) for some fixed \( k \) (i.e., \( k \) is independent of the input).

Examples: sorting, shortest path, MST, connectivity, ...

A Church-Turing thesis for “time”?

Church-Turing thesis: all “reasonable” models of computation are equivalent – i.e. all give the same set of decidable problems

“Extended” Church Turing thesis: All “reasonable” models of computation are equivalent up to a polynomial difference in time complexity

E.g. from above, this is true of deterministic single- and multi-tape TMs and RAMs

More on what “reasonable” means later…

Polynomial vs Exponential Growth
Nondeterministic Time

Given a nondeterministic TM M that always halts, its run time $T(n)$ is the length of the longest computation path (accepting or rejecting) on any input of length $n$.

(In fact, the theory doesn’t change much if you make it “shortest accepting path”, but that’s just a detail.)

The class NP

Definition:

$NP = \bigcup_{k=1}^{\infty} \text{Nondeterministic-TIME}(n^k)$

I.e., the set of (decision) problems solvable by computers in Nondeterministic polynomial time. I.e., $L \in NP$ iff there is a nondeterministic algorithm deciding L in time $T(n) = O(n^k)$ for some fixed $k$ (i.e., $k$ is independent of the input).

Ex: sorting, shortest path, …, and (probably) more!

Theorem: Every problem solvable in nondeterministic time $T(n)$ can be solved deterministically in time exponential in $T(n)$.

Proof:
As before, do breadth first simulation. (Depth-first works too.)

The Clique Problem

Given: a graph $G=(V,E)$ and an integer $k$

Question: is there a subset $U$ of $V$ with $|U| \geq k$ such that every pair of vertices in $U$ is joined by an edge.
"Problem" – the general case  
Ex: The Clique Problem: Given a graph G and an integer k, does G contain a k-clique?

"Problem Instance" – the specific cases  
Ex: Does \( G \) contain a 4-clique? (no)  
Ex: Does \( G \) contain a 3-clique? (yes)

Decision Problems – Just Yes/No answer
Problems as Sets of "Yes" Instances  
Ex: CLIQUE = \( \{ (G,k) \mid G \text{ contains a k-clique} \} \)  
E.g., (\( \begin{array}{cc}
\circ & \circ \\
\circ & \c
\end{array} \), 4) \( \notin \) CLIQUE  
E.g., (\( \begin{array}{cc}
\circ & \circ \\
\circ & \c
\end{array} \), 3) \( \in \) CLIQUE

Satisfiability

Boolean variables \( x_1, \ldots, x_n \)  
taking values in \{0,1\}. 0=false, 1=true

Literals  
\( x_i \) or \( \neg x_i \) for \( i = 1, \ldots, n \)

Clause  
a logical OR of one or more literals  
e.g. \( (x_1 \lor \neg x_3 \lor x_7 \lor x_{12}) \)

CNF formula (“conjunctive normal form”)  
a logical AND of a bunch of clauses

Satisfiability

CNF formula example  
\((x_1 \lor \neg x_3 \lor x_7) \land (\neg x_1 \lor \neg x_4 \lor x_5 \lor \neg x_7)\)  
If there is some assignment of 0’s and 1’s to the variables that makes it true then we say the formula is satisfiable  
the one above is, the following isn’t  
\( x_1 \land (\neg x_1 \lor x_2) \land (\neg x_2 \lor x_3) \land \neg x_3 \)

Satisfiability: Given a CNF formula F, is it satisfiable?
Common property of these problems:
Discrete Exponential Search
Loosely–find a needle in a haystack

“Answer” is literally just yes/no, but there’s always a somewhat more elaborate “solution” (aka “hint” or “certificate”) that transparently justifies each “yes” instance (and only those) – but it’s buried in an exponentially large search space of potential solutions.

‡Transparently = verifiable in polynomial time
The class NP

Definition:

NP = \bigcup_{k \geq 1} \text{Nondeterministic-TIME}(n^k)

I.e., the set of (decision) problems solvable by computers in Nondeterministic polynomial time. I.e.,

L \in \text{NP} iff there is a nondeterministic algorithm deciding L in time T(n) = O(n^k) for some fixed k (i.e., k is independent of the input).

Alternate Views of Nondeterminism

NTM – there is a path…

Parallel – make the tree

Search – look for a path (or sat-ing assignment or clique or…)

Guess and Check

Polynomial Verifier

Alternate Way To Define NP

A language L is polynomially verifiable iff there is a polynomial time procedure v(-,-), (the “verifier”) and an integer k such that

for every x \in L there is a “hint” h with |h| \leq |x|^k such that v(x,h) = YES and

for every x \notin L there is no hint h with |h| \leq |x|^k such that v(x,h) = YES

(“Hints,” sometimes called “certificates,” or “witnesses”, are just strings.)

Equivalently:

There is some integer k and language L_v in P s.t.:

L = \{ x | \exists y, |y| \leq |x|^k \land \langle x,y \rangle \in L_v \}

Example: Clique

"Is there a k-clique in this graph?"

any subset of k vertices might be a clique

there are many such subsets, but I only need to find one

if I knew where it was, I could describe it succinctly, e.g.

"look at vertices 2,3,17,42,..."

I'd know one if I saw one: "yes, there are edges between 2 & 3, 2 & 17,... so it's a k-clique"

this can be quickly checked

And if there is not a k-clique, I wouldn't be fooled by a statement like "look at vertices 2,3,17,42,..."
More Formally: CLIQUE is in NP

procedure v(x,h)
  if
    x is a well-formed representation of a graph
    G = (V, E) and an integer k,
  then
    h is a well-formed representation of a k-vertex
    subset U of V,
    U is a clique in G,
  then output "YES"
  else output "I'm unconvinced"

Is it correct?

For every x = (G,k) such that G contains a k-clique,
there is a hint h that will cause v(x,h) to say YES,
namely h = a list of the vertices in such a k-clique
and
No hint can fool v into saying yes if either x isn’t
well-formed (the uninteresting case) or if x = (G,k)
but G does not have any cliques of size k (the
interesting case)

The 2 defns are equivalent (cont.)

Theorem: L in NP iff L is polynomially verifiable
Pf: ⇔ Suppose L is poly time verifiable, V is a time n^d-time
   TM implementing the verifier, and k is the exponent in the
   hint length bound. Consider this TM:

M: on input x, nondeterministically choose a string y of
   length at most |x|^k, then run V on ⟨x,y⟩; accept iff it does.

Then M is an NTM accepting L: By defn of poly verifier
x ∈ L iff ∃y, |y| ≤ |x|^k ∧ V accepts ⟨x,y⟩, and M tries
(nondeterministically) all such y’s, accepting iff it finds one
that V accepts.

Time bound for M: (|x|+|x|^k+3)^d = O(n^d) = n^{O(1)}
Example: SAT

“Is there a satisfying assignment for this Boolean formula?”

any assignment might work
there are lots of them
I only need one
if I had one I could describe it succinctly, e.g., “x₁=T, x₂=F, ..., xₙ=T”
I’d know one if I saw one: “yes, plugging that in, I see formula = T...”
this can be quickly checked
And if the formula is unsatisfiable, I wouldn’t be fooled by, “x₁=T, x₂=F, ..., xₙ=F”

More Formally: SAT ∈ NP

Hint: the satisfying assignment A
Verifier: v(F,A) = syntax(F,A) && satisfies(F,A)

Syntax: True iff F is a well-formed formula & A is a truth-assignment to its variables
Satisfies: plug A into F and evaluate

Correctness:
If F is satisfiable, it has some satisfying assignment A, and we’ll recognize it
If F is unsatisfiable, it doesn’t, and we won’t be fooled

Alternate Views of Nondeterminism

NTM – there is a path…

Parallel – make the tree

Search – look for a path (or sat-ing assignment or clique or…)

Guess and Check

Polynomial Verifier

The complexity class NP

NP consists of all decision problems where

You can verify the YES answers efficiently (in polynomial time) given a short (polynomial-size) hint

And

No hint can fool your polynomial time verifier into saying YES for a NO instance

(one among exponentially many; know it when you see it)

(implausible for all exponential time problems)
**Keys to showing that a problem is in NP**

What’s the output? (must be YES/NO)
What’s the input? Which are YES?
For every given YES input, is there a hint that would help? Is it polynomial length?
  OK if some inputs need no hint
For any given NO input, is there a hint that would trick you?

**FALSE Example**

A$_{TM}$ is in NP
Input: a pair $<M,w>$
Output: yes/no does $M$ accept $w$
Hint: $y$, an accepting computation history of $M$ on $w$
Clearly, such a $y$ exists for all accepted $x$ and only accepted $x$, so we accept the right $x$'s and reject the rest.
And it’s fast – checking successive configs in the history is at worst quadratic in the length of the history, so the verifier for $<x,y>$ runs in time $|<x,y>|^O(1)$.

**P and NP**

Definition:

$P = \bigcup_{k \in \mathbb{N}} \text{TIME}(n^k)$
I.e., the set of (decision) problems solvable by computers in polynomial time.

$NP = \bigcup_{k \in \mathbb{N}} \text{Nondeterministic-TIME}(n^k)$
I.e., the set of (decision) problems solvable by computers in Nondeterministic polynomial time.
Alternate Definition of NP

A language L is **polynomially verifiable** iff there is a polynomial time procedure \( v(-,-) \), (the “verifier”) and an integer \( k \) such that

for every \( x \in L \) there is a “hint” \( h \) with \( |h| \leq |x|^k \) such that \( v(x,h) = \text{YES} \)

and

for every \( x \not\in L \) there is no hint \( h \) with \( |h| \leq |x|^k \) such that \( v(x,h) = \text{YES} \)

("Hints,” sometimes called “certificates,” or “witnesses”, are just strings.)

Equivalently:

There is some integer \( k \) and language \( L_v \) in P s.t.:

\[ L = \{ x \mid \exists y, |y| \leq |x|^k \land \langle x,y \rangle \in L_v \} \]

FALSE Example

A\(_{TM} \) is in NP
Input: a pair \(<M,w>\)
Output: yes/no does \( M \) accept \( w \)
Hint: \( y \), an accepting computation history of \( M \) on \( w \)
Clearly, such a \( y \) exists for all accepted \( x \) and only accepted \( x \), so we accept the right \( x \)’s and reject the rest.
And it’s fast – checking successive configs in the history is at worst quadratic in the length of the history, so the verifier for \(<x,y>\) runs in time \(|<x,y>|^{O(1)}\).

P vs NP vs Exponential Time

Theorem: Every problem in \( NP \) can be solved deterministically in exponential time

Proof: “hints” are only \( n^k \) long; try all \( 2^{n^k} \) possibilities, say by backtracking. If any succeed, say YES; if all fail, say NO.
**P and NP**

Every problem in P is in NP
one doesn’t even need a hint for problems in P so just ignore any hint you are given

Every problem in NP is in exponential time

I.e., $P \subseteq NP \subseteq Exp$

We know $P \neq Exp$, so either $P \neq NP$, or $NP \neq Exp$ (most likely both)

---

**Problems**

**Short Path:**

4-tuples $\langle G, s, t, k \rangle$, where $G=(V,E)$ is a digraph with vertices $s$, $t$, and an integer $k$, for which there is a path from $s$ to $t$ of length $\leq k$

**Long Path:**

4-tuples $\langle G, s, t, k \rangle$, where $G=(V,E)$ is a digraph with vertices $s$, $t$, and an integer $k$, for which there is an acyclic path from $s$ to $t$ of length $\geq k$

---

**Mostly Long Paths**

“Are the *majority* of paths from A to B long ($>k$)?”

Any path might work.

Yes! $\Rightarrow$ There are lots of them.

If I knew one I’d know a succinct way to describe it.

I’d know one: “yes, I see an edge from $A$ to $2$ and from $2$ to $42$ and... total length $> k$”

And if there isn’t a long path, I wouldn’t be fooled …

**More Problems**

**Independent-Set:**

Pairs $\langle G, k \rangle$, where $G=(V,E)$ is a graph and $k$ is an integer, for which there is a subset $U$ of $V$ with $|U| \geq k$ such that no two vertices in $U$ are joined by an edge.

**Clique:**

Pairs $\langle G, k \rangle$, where $G=(V,E)$ is a graph and $k$ is an integer $k$, for which there is a subset $U$ of $V$ with $|U| \geq k$ such that every pair of vertices in $U$ is joined by an edge.
More Problems

Euler Tour:
Graphs $G=(V,E)$ for which there is a cycle traversing each edge once.

Hamilton Tour:
Graphs $G=(V,E)$ for which there is a simple cycle of length $|V|$, i.e., traversing each vertex once.

TSP:
Pairs $\langle G,k \rangle$, where $G=(V,E,w)$ is a weighted graph and $k$ is an integer, such that there is a Hamilton tour of $G$ with total weight $\leq k$.

Some similar patterns that suggest problems not in NP

Rather than “there is a...” maybe it’s “no...” or “for all...”

E.g.
UNSAT: “no assignment satisfies formula,” or “for all assignments, formula is false”

Or
NOCLIQUE: “every subset of $k$ vertices is not a $k$-clique”

These examples are in co-NP: complements of problems in NP. (Quantifier characterization:
$... L = \{ x | \forall y, |y| \leq |x|^k \land \langle x,y \rangle \in L \} \ldots \}
NP == co-NP? Unknown, but seems unlikely.

Generic Pattern in These Examples

Set of all $x$ for which there is a $y$ with some property $P$, and
1) $y$ isn’t too big ($|y| = |x|^O(1)$), and
2) the property is easy (poly time) to check (given $x$ & $y$; perhaps not easy at all given only $x$)

“There is a” is a reflection of the quantifier characterization of NP:
$L$ is in NP iff there is some integer $k$ and language $L_v$ in P s.t.:
$L = \{ x | \exists y, |y| \leq |x|^k \land \langle x,y \rangle \in L_v \}$

Some similar patterns that suggest problems not in NP

Rather than “there is a...” maybe it’s “...is the min (or max)...”

E.g.
MAXCLIQUE: $k$ is the size of the largest clique in $G$

Or
MINTSP: $k$ is the cost of the cheapest Ham cycle in $G$

Again, they seem NP-like, but are probably “harder.” E.g.,
not only do you need to prove existence of $k$-clique (a problem in NP) you also need to prove absence of a
$(k+1)$-clique (a co-NP question)
Quantifier structure often: “… $\exists y_1 \forall y_2 (y_1 < y_2 \Rightarrow \ldots)$”
Some similar patterns that suggest problems not in NP

Rather than “there is a…” maybe it’s something even more complicated, like

• the “mostly long paths” example above,
• “there is an exponentially long string y with property P”,
• some quantifier structure other than just \( \exists \), such as
  \[ \exists x_1 \land x_2 \exists x_3 \land x_4 \exists x_5 \land x_6 \ldots \text{formula}(x_1 \ldots x_n) = \text{True} \]
• or many other things

Bottom line:
NP is a common, but not universal, problem pattern

2 Final Points About “Hints”

1. Hints/verifiers aren’t unique. The “…there is a…” framework often suggests their form, but many possibilities

   “is there a clique” could be verified from its vertices, or its edges, or all but 3 of each, or all non-vertices, or… Details of the hint string and the verifier and its time bound shift, but same bottom line

2. In NP doesn’t prove its hard

   “Short Path” or “Small spanning tree” can be formulated as “…there is a…”, but, due to very special structure of these problems, we can quickly find the solution even without a hint. The mystery is whether that’s possible for the other problems, too.

Review from previous lecture

P \( \subseteq \) NP \( \subseteq \) Exp; at least one containment is proper
Examples in NP:
SAT, short/long paths, Euler/Ham tours, clique, indp set…
Common feature:
 “…there is a…”
(and some related problems do not appear to share this feature: UnSAT, maxClique, MostlyLongPaths, …)
Some Problem Pairs

- Euler Tour
- 2-SAT
- 2-Coloring
- Min Cut
- Shortest Path
- Hamilton Tour
- 3-SAT
- 3-Coloring
- Max Cut
- Longest Path

Superficially different, similar computationally

Solving NP problems without hints

The most obvious algorithm for most of these problems is brute force:
- try all possible hints; check each one to see if it works.
- Exponential time:
  - \(2^n\) truth assignments for \(n\) variables
  - \(n!\) possible TSP tours of \(n\) vertices
  - \(\binom{n}{k}\) possible \(k\) element subsets of \(n\) vertices
  - etc.

...and to date, every alg, even much less-obvious ones, are slow, too

P vs NP

Theory
- \(P = NP?\)
- Open Problem!
- I bet against it

Practice
- Many interesting, useful, natural, well-studied problems known to be NP-complete
- With rare exceptions, no one routinely succeeds in finding exact solutions to large, arbitrary instances

Another NP problem: Vertex Cover

Input: Undirected graph \(G = (V, E)\), integer \(k\).
Output: True iff there is a subset \(C\) of \(V\) of size \(\leq k\) such that every edge in \(E\) is incident to at least one vertex in \(C\).

Example: Vertex cover of size \(\leq 2\).

In NP? Exercise
$3SAT \leq_p VertexCover$

$3SAT \leq_p VertexCover$

$k=6$

$3SAT \leq_p VertexCover$
### 3SAT \leq_p VertexCover

\[(x_1 \lor x_2 \lor \neg x_3) \land (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3)\]

### Correctness of “3SAT \leq_p VertexCover”

**Summary of reduction function f**: Given formula, make graph G with one group per clause, one node per literal. Connect each to all nodes in same group, plus complementary literals \((x, \neg x)\). Output graph G plus integer \(k = 2^q\) number of clauses. Note: f does not know whether formula is satisfiable or not; does not know if G has k-cover; does not try to find satisfying assignment or cover.

**Correctness**:

- Show f poly time computable: A key point is that graph size is polynomial in formula size; mapping basically straightforward.
- Show c in 3-SAT iff f(c)=(G,k) in VertexCover:
  - \((\Rightarrow)\) Given an assignment satisfying c, pick one true literal per clause. Add other 2 nodes of each triangle to cover. Show it is a cover: 2 per triangle cover triangle edges; only true literals (but perhaps not all true literals) uncovered, so at least one end of every \((x, \neg x)\) edge is covered.
  - \((\Rightarrow)\) Given a k-vertex cover in G, uncovered labels define a valid (perhaps partial) truth assignment since no \((x, \neg x)\) pair uncovered. It satisfies c since there is one uncovered node in each clause triangle (else some other clause triangle has > 1 uncovered node, hence an uncovered edge.)
3SAT $\leq_p$ VertexCover

**3-SAT Instance:**
- Variables: $x_1, x_2, \ldots$
- Literals: $y_{ij}, 1 \leq i \leq q, 1 \leq j \leq 3$
- Clauses: $c_i = y_{i1} \lor y_{i2} \lor y_{i3}, 1 \leq i \leq q$
- Formula: $c = c_1 \land c_2 \land \ldots \land c_q$

**VertexCover Instance:**
- $k = 2q$
- $G = (V, E)$
- $V = \{ [i,j] \mid 1 \leq i \leq q, 1 \leq j \leq 3 \}$
- $E = \{ ([i,j], [k,l]) \mid i = k \text{ or } y_{ij} = \neg y_{kl} \}$

Correctness of “3SAT $\leq_p$ VertexCover”

*Summary of reduction function $f$: Given formula, make graph $G$ with one node per clause, one node per literal. Connect each to all nodes in same group, plus complementary literals ($x, \neg x$). Output graph $G$ plus integer $k = 2 \times$ number of clauses. Note: $f$ does not know whether formula is satisfiable or not; does not know if $G$ has $k$-cover; does not try to find satisfying assignment or cover.*

**Correctness:**
- Show $f$ poly time computable: A key point is that graph size is polynomial in formula size; mapping basically straightforward.
- Show $c$ in 3-SAT iff $f(c)=(G,k)$ in VertexCover:
  \( \Rightarrow \) Given an assignment satisfying $c$, pick one true literal per clause. Add other 2 nodes of each triangle to cover. Show it is a cover: 2 per triangle cover triangle edges; only true literals (but perhaps not all true literals) uncovered, so at least one end of every ($x, \neg x$) edge is covered.
  \( \Rightarrow \) Given a $k$-vertex cover in $G$, uncovered labels define a valid (perhaps partial) truth assignment since no ($x, \neg x$) pair uncovered. It satisfies $c$ since there is one uncovered node in each clause triangle (else some other clause triangle has $>1$ uncovered node, hence an uncovered edge.)
Utility of “3SAT ≤_p VertexCover”

Suppose we had a fast algorithm for VertexCover, then we could get a fast algorithm for 3SAT:

Given 3-CNF formula w, build Vertex Cover instance y = f(w) as above, run the fast VC alg on y; say “YES, w is satisfiable” iff VC alg says “YES, y has a vertex cover of the given size”

On the other hand, suppose no fast alg is possible for 3SAT, then we know none is possible for VertexCover either.

“3SAT ≤_p VertexCover” Retrospective

Previous slide: two suppositions
Somewhat clumsy to have to state things that way.
Alternative: abstract out the key elements, give it a name (“polynomial time mapping reduction”), then properties like the above always hold.

Polynomial-Time Reductions

Definition: Let A and B be two problems.
We say that A is polynomially (mapping) reducible to B (A ≤_p B) if there exists a polynomial-time algorithm f that converts each instance x of problem A to an instance f(x) of B such that:

x is a YES instance of A iff f(x) is a YES instance of B

x ∈ A ⇔ f(x) ∈ B

Polynomial-Time Reductions (cont.)

Define: A ≤_p B “A is polynomial-time reducible to B”, iff there is a polynomial-time computable function f such that:

x ∈ A ⇔ f(x) ∈ B

“complexity of A” ≤ “complexity of B” + “complexity of f”

(1) A ≤_p B and B ∈ P ⇒ A ∈ P
(2) A ≤_p B and A ∉ P ⇒ B ∉ P
(3) A ≤_p B and B ≤_p C ⇒ A ≤_p C (transitivity)
Two definitions of “A \leq_p B”

Some books use more general defn: “could solve A in poly time, if I had a poly time subroutine for B.”

Defn on previous slides is special case where you only get to call the subroutine once, and must report its answer.

This special case is used in ~98% of all reductions (And is the only one used in Ch 7, I think.)

NP-Completeness

Definition: Problem B is NP-hard if every problem in NP is polynomially reducible to B.

Definition: Problem B is NP-complete if:
1. B belongs to NP, and
2. B is NP-hard.

Polynomial-Time Reductions (cont.)

Define: A \leq_p B “A is polynomial-time reducible to B”, iff there is a polynomial-time computable function f such that: x \in A \iff f(x) \in B

“complexity of A” \leq “complexity of B” + “complexity of f”

(1) A \leq_p B and B \in P \Rightarrow A \in P
(2) A \leq_p B and A \notin P \Rightarrow B \notin P
(3) A \leq_p B and B \leq_p C \Rightarrow A \leq_p C (transitivity)
**NP-Completeness**

Definition: Problem B is **NP-hard** if every problem in NP is polynomially reducible to B.

Definition: Problem B is **NP-complete** if:
1. B belongs to NP, and
2. B is NP-hard.

---

**Why is SAT NP-complete?**

Cook’s proof is somewhat involved; details later. But its essence is not so hard to grasp:

<table>
<thead>
<tr>
<th>Generic “NP” problem:</th>
<th>“SAT”:</th>
</tr>
</thead>
<tbody>
<tr>
<td>is there a poly size “solution,” verifiable by computer in poly time</td>
<td>is there a (poly size) assignment satisfying the formula</td>
</tr>
</tbody>
</table>

Encode “solution” using Boolean variables. SAT mimics “is there a solution” via “is there an assignment”. Digital computers just do Boolean logic, and “SAT” can mimic that, too, hence can verify that the assignment actually encodes a solution.

---

**Proving a problem is NP-complete**

Technically, for condition (2) we have to show that every problem in NP is reducible to B. (Yikes! Sounds like a lot of work.)

For the very first NP-complete problem (SAT) this had to be proved directly.

However, once we have one NP-complete problem, then we don’t have to do this every time. Why? Transitivity.

---

**“NP-completeness”**

Cool concept, but are there any such problems?

Yes!

Cook’s theorem: SAT is NP-complete
Alt way to prove NP-completeness

Lemma: Problem B is NP-complete if:
(1) B belongs to NP, and
(2') A is polynomial-time reducible to B, for some problem A that is NP-complete.

That is, to show (2') given a new problem B, it is sufficient to show that SAT or any other NP-complete problem is polynomial-time reducible to B.

Ex: VertexCover is NP-complete

3-SAT is NP-complete (shown by S. Cook)
3-SAT ≤p VertexCover
VertexCover is in NP (we showed this earlier)
Therefore VertexCover is also NP-complete
So, poly-time algorithm for VertexCover would give poly-time algs for everything in NP

NP-complete problem: Clique

Input: Undirected graph G = (V, E), integer k.
Output: True iff there is a subset C of V of size ≥ k such that all vertices in C are connected to all other vertices in C.

Example: Clique of size ≥ 4

In NP? Exercise

3SAT ≤p Clique

k=3
3SAT ≤_p Clique

3SAT ≤_p Clique

(x₁ ∨ x₂ ∨ ¬x₃) ∧ (x₁ ∨ ¬x₂ ∨ ¬x₃) ∧ (¬x₁ ∨ x₃)
3-SAT \leq_p Clique

3-SAT Instance:
- Variables: \( x_1, x_2, \ldots \)
- Literals: \( y_{ij}, 1 \leq i \leq q, 1 \leq j \leq 3 \)
- Clauses: \( c_i = y_{i1} \lor y_{i2} \lor y_{i3}, 1 \leq i \leq q \)
- Formula: \( c = c_1 \land c_2 \land \ldots \land c_q \)

Clique Instance:
- \( K = q \)
- \( G = (V, E) \)
- \( V = \{ [i,j] \mid 1 \leq i \leq q, 1 \leq j \leq 3 \} \)
- \( E = \{ ([i,j], [k,l]) \mid i \neq k \text{ and } y_{ij} = \neg y_{kl} \} \)

Correctness of “3-SAT \leq_p Clique”

Summary of reduction function \( f \):
- Given formula, make graph \( G \) with column of nodes per clause, one node per literal. Connect each to all nodes in other columns, except complementary literals (\( x, \neg x \)). Output graph \( G \) plus integer \( k \) = number of clauses. Note: \( f \) does not know whether formula is satisfiable or not; does not know if \( G \) has \( k \)-clique; does not try to find satisfying assignment or clique.
- Correctness:
  - Show \( f \) poly time computable: A key point is that graph size is polynomial in formula size; mapping basically straightforward.
  - Show \( c \) in 3-SAT iff \( f(c) = (G,k) \) in Clique:
    - \((\Rightarrow)\) Given an assignment satisfying \( c \), pick one true literal per clause. Show corresponding nodes in \( G \) are \( k \)-clique.
    - \((\Leftarrow)\) Given a \( k \)-clique in \( G \), clique labels define a truth assignment; show it satisfies \( c \). Note: literals in a clique are a valid truth assignment [no ”\((x, \neg x)\)” edges] & \( k \) nodes must be 1 per column, [no edges within columns].

3-SAT \leq_p UndirectedHamPath

Example: \( (x \lor y) \land (\neg x \lor y) \land (\neg x \lor \neg y) \)

Ham Path Gadget

Many copies of this 12-node gadget, each with one or more edges connecting each of the 4 corners to other nodes or gadgets (but no other edges to the 8 “internal” nodes).
- Claim: There are only 2 Ham paths – one entering at 1, exiting at 1’ (as shown); the other (by symmetry) 0 \rightarrow 0’
- Pf: Note #: at 1\textsuperscript{st} visit to any column, must next go to middle node in column, else it will subsequently become an untraversable “dead end.”
- WLOG, suppose enter at 1. By #, must then go down to 0. 2 cases:
  - Case a: (top left) If next move is to right, then # forces path up, left is blocked, so right again, # forces down, etc; out at 1’.
  - Case b: (top rt) If exit at 0, then path must eventually reenter at 0’ or 1’. # forces next move to be up/down to the other of 0/1’. Must then go left to reach the 2 middle columns, but there’s no exit from them. So case b is impossible.
Lecture 24

3-SAT \leq_p UndirectedHamPath

Example:

\((x \lor y) \land (\neg x \lor y) \land (\neg x \lor \neg y)\)

\(s\)
\(t\)

(Note: this is not the same as the reduction given in the book.)

3-SAT \leq_p UndirectedHamPath

Time for the reduction: to be computable in poly time it is necessary (but not sufficient) that G's size is polynomial in \(n\), the length of the formula. Easy to see this is true, since G has \(q + 12(p + m) + 1 = O(n)\) vertices, where \(q\) is the number of clauses, \(p\) is the number of instances of literals, and \(m\) is the number of variables. Furthermore, the structure is simple and regular, given the formula, so easily / quickly computable, but details are omitted. (More detail expected in your homeworks, e.g.)

Ham Path Gadget

Many copies of this 12-node gadget, each with one or more edges connecting each of the 4 corners to other nodes or gadgets (but no other edges to the 8 "internal" nodes).

Claim: There are only 2 Ham paths – one entering at 1, exiting at 1' (as shown); the other (by symmetry) 0→0'.

Pf: Note *: at 1\textsuperscript{st} visit to any column, must next go to middle node in column, else it will subsequently become an untraversable "dead end."

WLOG, suppose enter at 1. By *, must then go down to 0. 2 cases:

Case a: (top left) If next move is to right, then * forces path up, left is blocked, so right again, * forces down, etc; out at 1'.

Case b: (top rt) if exit at 0, then path must eventually reenter at 0' or 1'. * forces next move to be up/down to the other of 0'/1'. Must then go left to reach the 2 middle columns, but there's no exit from them. So case b is impossible.
Correctness, I

Ignoring the clause nodes, there are $2^m$ s-t paths along the "main chain," one for each of $2^m$ assignments to $m$ variables. If $f$ is satisfiable, pick a satisfying assignment, and pick a true literal in each clause. Take the corresponding "main chain" path; add a detour to/from $c_i$ for the true literal chosen from clause $i$. Result is a Hamilton path.

Correctness, II

Conversely, suppose $G$ has a Ham path. Obviously, the path must detour from the main chain to each clause node $c_i$. If it does not return immediately to the next gadget on main chain, then (by gadget properties on earlier slide), that gadget cannot be traversed. Thus, the Ham path must consistently use "top chain" or consistently "bottom chain" exits to clause nodes from each variable gadget. If top chain, set that variable True; else set it False. Result is a satisfying assignment, since each clause is visited from a "true" literal.

Subset-Sum, AKA Knapsack

$\text{KNAP} = \{ (w_1, w_2, \ldots, w_n, C) \mid \text{a subset of the } w_i \text{ sums to } C \}$

$w_i$'s and $C$ encoded in radix $r \geq 2$. (Decimal used in following example.)

Theorem: $3\text{-SAT} \leq_p \text{KNAP}$

Pf: given formula with $p$ variables & $q$ clauses, build KNAP instance with $2(p+q)$ $w_i$'s, each with $(p+q)$ decimal digits. For the $2p$ "literal" weights, H.O. $p$ digits mark which variable; L.O. $q$ digits show which clauses contain it. Two "slack" weights per clause mark that clause. See example below.

3-SAT $\leq_p$ KNAP

Formula: $(x \lor y) \land (\neg x \lor y) \land (\neg x \lor \neg y)$

<table>
<thead>
<tr>
<th>Variables</th>
<th>$x$</th>
<th>$y$</th>
<th>$(x \lor y)$</th>
<th>$\neg x \lor y$</th>
<th>$\neg x \lor \neg y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1$ ($x$)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$w_2$ ($\neg x$)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$w_3$ ($y$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$w_4$ ($\neg y$)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$w_5$ ($t_{i1}$)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$w_6$ ($t_{i2}$)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$w_7$ ($t_{j1}$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$w_8$ ($t_{j2}$)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$w_9$ ($t_{k1}$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$w_{10}$ ($t_{k2}$)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| Slack | 1 | 1 | 3 | 3 | 3 |

C
Correctness

Poly time for reduction is routine; details omitted.

If formula is satisfiable, select the literal weights corresponding to the true literals in a satisfying assignment. If that assignment satisfies k literals in a clause, also select \((3 - k)\) of the “slack” weights for that clause. Total will equal \(C\).

Conversely, suppose KNAP instance has a solution. Note \(\leq 5\) one’s per column, so no “carries” in sum (recall – weights are decimal); i.e., columns are decoupled. Since H.O. \(p\) digits of \(C\) are 1, exactly one of each pair of literal weights included in the subset, so it defines a valid assignment. Since L.O. \(q\) digits of \(C\) are 3, but at most 2 “slack” weights contribute to it, at least one of the selected literal weights must be 1 in that clause, hence the assignment satisfies the formula.

As a supplement to Paul Beame’s guest lecture, here are a few slides of mine on roughly the same topics. Again, this won’t be exactly the same as what he did or as what’s in the book, but hopefully another perspective will help clarify it all.

Lecture 25

Boolean Circuits

Directed acyclic graph
Vertices = Boolean logic gates \((\land, \lor, \neg, \ldots)\)
Multiple input bits \((x_1, x_2, \ldots)\)
Single output bit \((w)\)
Gate values as expected (e.g. by induction on depth to \(x_1\)’s)
Boolean Circuits

Two Problems:
Circuit Value: given a circuit and an assignment of values to its inputs, is its output = 1?
Circuit SAT: given a circuit, is there an assignment of values to its inputs such that output = 1?

Complexity:
- Circuit Value Problem is in P
- Circuit SAT Problem is in NP

Given implementation of computers via Boolean circuits, it may be unsurprising that they are complete in P/NP, resp.

Some Details

For \( q \in Q, a \in \Gamma, 1 \leq i, j \leq T \), let

\[
\text{state}(q, i, j) = 1 \text{ if } M \text{ is in state } q \text{ at time } i \text{ w/ head in tape cell } j, \text{ and letter}(a, i, j) = 1 \text{ if tape cell } j \text{ holds letter } a \text{ at time } i.
\]

write cell \( i \) at step \( j \), run

\[
\begin{align*}
\text{writes}(i, j) &= \bigvee_{q \in Q} \text{state}(q, i, j) \\
\text{letter}(b, i, j) &= (\neg \text{writes}(i, j) \land b_{i-1, j}) \lor
\bigvee_{(q, a)} (\text{writes}(i, j) \land \text{state}(q, i, j) \land \text{letter}(a, i, j))
\end{align*}
\]

where the "or" is over \( \{(q, a) \mid \delta(q, a) = \langle \delta(q, a) \rangle\} \), \( d = \pm 1 \)

state \( (p, i, j) = \bigvee_{(q, a, d)} \text{state}(q, i, j, d) \land \text{letter}(a, i, j, d), \)
where the "or" is over \( \{(q, a, d) \mid \langle q, a, d \rangle = \delta(q, a) \} \)

Row 0: initial config; columns -1, T+1: all false
Output: state(\( q_{\text{accept}}, T, 1 \))

Again, not exactly the version in the book, but close in spirit...
Some Details

Additionally, assume NTM has only 2 nondet choices at each step. For \( q \in Q, a \in \Gamma, 1 \leq i,j \leq T, \) state\((q,i,j)\), letter\((a,i,j)\) as before. Let choice\((i) = 0/1\) define which ND choice M makes at step i. Then, letter\()()\) and state\()()\) circuits change to incl choice, e.g.:

\[
\begin{align*}
\text{state}(p,i,j) &= \neg \text{choice}(i-1) \land \bigvee_{(q,a,d)} \text{state}(q,i-1,j-d) \land \text{letter}(a,i-1,j-d) \lor \\
\text{choice}(i-1) \land \bigvee_{(q',a',d')} \text{state}(q',i-1,j-d') \land \text{letter}(a',i-1,j-d') ,
\end{align*}
\]

where the "ors" are over
\[
\{(q,a,d) \mid (p,-,d) = \delta(q, a, \text{choice}=0)\} , \quad \{(q',a',d') \mid (p,-,d') = \delta(q', a', \text{choice}=1)\} , \quad d = \pm 1
\]

AND

\( TM \) input \( \rightarrow \) circuit constants; circuit inputs are the choice bits; circuit is satisfiable iff \( \exists \) seq of choices s.t. NTM accepts

Correctness

Poly time reduction:

Given \( \delta \), key subcircuit is fixed, size \( O(1) \). Calculate \( n = \text{input length}, T = n^k \). Circuit has \( O(T^2) = O(n^{2k}) \) copies of that subcircuit, (plus some small tweaks at boundaries). Circuit exactly reflects M’s computation, given the choice sequence. So, if \( M \) accepts input \( x \), then there is a choice sequence s.t. circuit will output \( 1 \), i.e., the circuit is satisfiable. Conversely, if the circuit is satisfiable, then any satisfying input constitutes a choice sequence leading \( M \) to accept \( x \).

Thus, Circuit-SAT is NP-complete.
Correctness of “Circuit-SAT \leq_p 3-SAT”

Summary of reduction: Given circuit, add variable for every gate’s value, build clause for each gate, satisfiable iff gate value variable is appropriate logical function of its input variables, convert each to CNF via standard truth-table construction. Output conjunction of all, plus output variable. Note: as usual, does not know whether circuit or formula are satisfiable or not; does not try to find satisfying assignment.

Correctness:
Show it’s poly time computable: A key point is that formula size is linear in circuit size; mapping basically straightforward; details omitted.
Show c in Circuit-SAT iff f(c) in SAT:
(\Rightarrow) Given an assignment to x_i’s satisfying c, extend it to w_i’s by evaluating the circuit on x_i’s gate by gate. Show this satisfies f(c).
(\Leftarrow) Given an assignment to x_i’s & w_i’s satisfying f(c), show x_i’s satisfy c (with gate values given by w_i’s).
Thus, 3-SAT is NP-complete.

Common Errors in NP-completeness Proofs

Backwards reductions
Bipartiteness \leq_p SAT is true, but not so useful.
(\text{XYZ} \leq_p SAT shows XYZ in NP, doesn’t show it’s hard.)

Sloooow Reductions
“Find a satisfying assignment, then output…”

Half Reductions
Delete clause nodes in HAM reduction. It’s still true that “satisfiable \Rightarrow G has a Ham path”, but path doesn’t necessarily give a satisfying assignment.
Coping with NP-Completeness

Is your real problem a special subcase?
E.g. 3-SAT is NP-complete, but 2-SAT is not; ditto 3- vs 2-coloring
E.g. you only need planar graphs, or degree 3 graphs, …?

Guaranteed approximation good enough?
E.g. Euclidean TSP within $2 \times \text{Opt}$ in poly time

Fast enough in practice (esp. if $n$ is small),
E.g. clever exhaustive search like backtrack, branch & bound, pruning

Heuristics – usually a good approximation and/or usually fast

TSP - Nearest Neighbor Heuristic

NN Heuristic – go to nearest unvisited vertex

Fact: NN tour can be about $(\log n) \times \text{opt}$, i.e.

$$\lim_{n \to \infty} \frac{NN}{OPT} \to \infty$$

(above example is not that bad)

NP-complete problem: TSP

Input: An undirected graph $G=(V,E)$ with integer edge weights, and an integer $b$.

Output: YES iff there is a simple cycle in $G$ passing through all vertices (once), with total cost $\leq b$.

Example: $b = 34$

2x Approximation to Euclidean TSP

A TSP tour visits all vertices, so contains a spanning tree, so TSP cost is $> \text{cost of min spanning tree}$.

Find MST

Find “DFS” Tour

Shortcut

$\text{TSP} \leq \text{shortcut} < \text{DFST} = 2 \times \text{MST} < 2 \times \text{TSP}$
Summary

Big-O – good
P – good
Exp – bad
Exp, but hints help? NP
NP-hard, NP-complete – bad (I bet)
To show NP-complete – reductions
NP-complete = hopeless? – no, but you need to lower your expectations: heuristics & approximations.

Beyond NP

Many complexity classes are worse, e.g. time $2^2^n$, $2^{2^2^n}$, …
Others seem to be “worse” in a different sense, e.g., not in NP, but still exponential time. E.g., let $L_p = \text{“assignment } y \text{ satisfies formula } x\text{”, } \in P$
Then:
 SAT = { $x \mid \exists y \ freshness\langle x, y \rangle \in L_p$ }
 UNSAT = { $x \mid \forall y \ freshness\langle x, y \rangle \in L_p$ }
 $QBF_k = \{ x \mid \exists y_1, \forall y_2, \exists y_3 \ldots \Diamond_k \langle x, y_1 \ldots y_k \rangle \in L_p \}$
 $QBF_\omega = \{ x \mid \exists y_1, \forall y_2, \exists y_3 \ldots \langle x, y_1 \ldots \rangle \in L_p \}$

“I can’t find an efficient algorithm, but neither can all these famous people.”

[Lecture 27]
Beyond NP

Many complexity classes are worse, e.g. time $2^n$, $2^{2^n}$, ...
Others seem to be “worse” in a different sense, e.g., not in NP, but still exponential time. E.g., let $L_p$ = “assignment $y$ satisfies formula $x$”, $\in P$
Then:

$\text{SAT} = \{ x \mid \exists y \langle x,y \rangle \in L_p \}$
$\text{ UNSAT} = \{ x \mid \forall y \langle x,y \rangle \notin L_p \}$
$\text{QBF}_k = \{ x \mid \exists y_1 \forall y_2 \exists y_3 \ldots \forall y_k \langle x,y_1 \ldots y_k \rangle \in L_p \}$
$\text{QBF}_\omega = \{ x \mid \exists y_1 \forall y_2 \exists y_3 \ldots \langle x,y_1 \ldots \rangle \in L_p \}$

The “Polynomial Hierarchy”

$\Sigma^p_0: \text{ } \{ x \mid \exists y \forall z \langle x,y,z \rangle \in L_p \}$
$\Pi^p_0: \text{ } \{ x \mid \forall y \exists z \langle x,y,z \rangle \in L_p \}$

$\Pi^p_1: \text{P time given SAT}$

$\Pi^p_i (\text{co-NP}): \text{ } \{ x \mid \forall y \langle x,y \rangle \in L_p \}$

$\text{UNSAT, ...}$

$\Sigma^p_i: \text{ } \{ x \mid \exists y \forall z \langle x,y,z \rangle \in L_p \}$

$\Delta^p_i: \text{P}$

Potential Utility: It is often easy to give such a quantifier-based characterization of a language; doing so suggests (but doesn’t prove) whether it is in P, NP, etc. and suggests candidates for reducing to it.

Examples

$\text{QBF}_k$ in $\Sigma^p_k$

Given graph $G$, integers $j$ & $k$, is there a set $U$ of $\leq j$ vertices in $G$ such that every $k$-clique contains a vertex in $U$?

Given graph $G$, integers $j$ & $k$, is there a set $U$ of $\geq j$ vertices in $G$ such that removal of any $k$ edges leaves a Hamilton path in $U$?

Space Complexity

DTM $M$ has space complexity $S(n)$ if it halts on all inputs, and never visits more than $S(n)$ tape cells on any input of length $n$.

NTM … on any input of length $n$ on any computation path.

$\text{DSPACE}(S(n)) = \{ L \mid L \text{ acc by some DTM in space } O(S(n)) \}$

$\text{NSPACE}(S(n)) = \{ L \mid L \text{ acc by some NTM in space } O(S(n)) \}$
**Model-independence**

As with Time complexity, model doesn’t matter much. E.g.:

$$\text{SPACE}(n) \text{ on DTM} = O(n) \text{ bytes on your laptop}$$

Why? Simulate each by the other.

**Space vs Time**

Time T \subseteq \text{Space T}

Pf: no time to use more space

Space T \subseteq \text{Time } 2^T

Pf: if run longer, looping

---

**Space seems more powerful**

Intuitively, space is reusable, time isn’t

Ex.: SAT \in \text{DSPACE}(n)

Pf: try all possible assignments, one after the other

Even more:

$$\text{QBF}_k = \{ \exists y_1 \forall y_2 \exists y_3 \ldots \exists y_k \langle x, y_1 \ldots y_k \rangle \in L_p \} \in \text{DSPACE}(n)$$

$$\text{QBF}_x = \{ \exists y_1 \forall y_2 \exists y_3 \ldots \langle x, y_1 \ldots \rangle \in L_p \} \in \text{DSPACE}(n)$$

PSPACE = Space(n^{O(1)})

NP \subseteq \text{PSPACE}

pf: depth-first search of NTM computation tree
Games

2 player “board” games
E.g., checkers, chess, tic-tac-toe, nim, go, …
A finite, discrete “game board”
Some pieces placed and/or moved on it
“Perfect information”: no hidden data, no randomness
Player I/Player II alternate turns
Defined win/lose configurations (3-in-a-row; checkmate; …)

Winning strategy:
\[ \exists \text{move by player I} \ \forall \text{moves by II} \ \exists \text{a move by I} \ \forall \ldots \ \text{I wins.} \]

Game Tree

Config:
Where are pieces
Relevant history
Who goes next
Play:
All moves
Win/lose:

Config:
Where are pieces
Relevant history
Who goes next
Play:
All moves
Win/lose:

Winning Strategy
Complexity of 2 person, perfect information games

From above, IF
config (incl. history, etc.) is poly size
only poly many successors of one config
each computable in poly time
win/lose configs recognizable in poly time, and
game lasts poly # moves
THEN
in PSPACE!
Pf. depth-first search of tree, calc node values as you go.
Complexity of 2 person, perfect information games

From above, if
config (incl. history, etc.) is poly size
only poly many successors of one config
each computable in poly time
win/lose configs recognizable in poly time, and
game lasts poly # moves
then
in PSPACE!
Pf: depth-first search of tree, calc node values as you go.

A Game About Paths:
Which Player Has A Winning Strategy?

Given: digraph G with $2^n + 1$ vertices, movable markers s, t on two vertices
Outline:
Player I: “I have a path (from s to t)”
Player II: “I doubt it”
Play alternates, starting with player I:
Player I: places marker m on some node (“path goes thru m”)
Player II: $(s,t) \leftarrow (s,m)$ or $(m,t)$ (“I doubt this half”)
Ends after n rounds; Player I wins if s = t, or s -> t is an edge
Winning The Path Game

Player I has a winning strategy if there is an s-t path:
- Path has $2^n$ edges; choosing middle vertex of that path for “m” in each round halves the remaining path length, so after n rounds, path length is $\leq 1$, which is the “win” condition for Player I.

Player II has a winning strategy if there is no s-t path:
- If there is no s-t path, for every m, either there is no s-m path or no m-t path (or both). In the former case, choose (s, m), else (m, t). At termination, $s \not= t$ and $s \not\rightarrow t$ isn’t an edge.

Game Tree/Strategy

2n levels
Player I ($\exists$) chooses among many possible “m” nodes
Player II ($\forall$) chooses left/right half

Complexity & The Path Game

M: a space $S(n)$ NTM. WLOG, before accepting, M:
- erases tape
- goes to left end of tape
So, there are unique init & accept configs, $C_0$, $C_a$.

Digraph G:
- Nodes: configs of M on fixed input $x$,
- Edges: $C \rightarrow C'$ iff M can move from config C to C' in 1 step.
M accepts x iff there is a path from $C_0$ to $C_a$ in G

Savitch’s Theorem

Theorem:
$\text{NSPACE}(S(n)) \subseteq \text{DSPACE}(S^2(n))$

Pf:
Accept iff Player I wins path game
Game tree has height $\log(\#\text{configs}) = O(S(n))$
Each node needs $O(S(n))$ bits to describe 2-3 configs (s,m,t)
Can evaluate win/lose at each leaf by examining 2 configs
So, evaluate tree in $O(S^2(n))$ space.
**Corollary:**

\[ \text{DetPSPACE} = \text{NondetPSPACE} \] (So we just say “PSPACE”)

Analogous result for P-TIME is of course the famous \( P = NP \) question.

**TQBF**

“True Quantified Boolean Formulas”

\[ \text{TQBF} = \{ \exists y_1 \forall x_1 \exists y_2 \ldots \text{f} | \text{assignment } x,y \text{ satisfies formula } f \} \]

(each \( x_i, y_i \) may be one or many bits; doesn’t matter.)

TQBF in PSPACE: think of it as a game between \( \exists, \forall; \exists \) wins if formula satisfied. Do DFS of game tree as in examples above, evaluating nodes (\( \land, \lor \)) as you backtrack.

**TQBF is PSPACE-complete**

**“TQBF is to PSPACE as SAT is to NP”**

\[ \text{TQBF} = \{ \exists y_1 \forall x_1 \exists y_2 \ldots \text{f} | \text{assignment } x,y \text{ satisfies formula } f \} \]

Theorem: TQBF is PSPACE-complete

Pf Idea:

TQBF in PSPACE: above

\( M \) an arbitrary \( n^k \) space TM, show \( L(M) \leq_p \text{TQBF} \) below

\( y_k \): the \( n^k \)-bit config “m” picked by \( \exists \)-player in round \( k \)

\( x_k \): 1 bit; \( \forall \)-player chooses which half-path is challenged

Formula f: x’s select the appropriate pair of y configs; check that 1st moves to 2nd in one step (à la Cook’s Thm)

**More Detail**

For “x selects a pair of y’s”, use the following trick:

\[ f_1(s_1, t_1) = \exists y_1 \forall x_1 \ g(s_1, t_1, y_1, x_1) \]

becomes

\[ \exists y_1 \forall x_1 \exists s_2, t_2 \ [ ( x_1 \rightarrow (s_2 = s_1 \land t_2 = y_1)) \land \]

\[ (\neg x_1 \rightarrow (s_2 = y_1 \land t_2 = t_1)) \land f_2(s_2, t_2) ] \]

Here, \( x_1 \) is a single bit; others represent \( n^k \)-bit configs, and “=” means the \( \land \) of bitwise \( \leftrightarrow \) across all bits of a config

The final piece of the formula becomes \( \exists z \ g(s_k, t_k, z) \), where \( g(s_k, t_k, z) \), as in Cook’s Thm, is true if config \( s_k \) equals \( t_k \) or moves to \( t_k \) in 1 step according to \( M \)’s nondet choice \( z \).

A key point: formula is poly computable (e.g., poly length)
And so GGEO is PSPACE-complete

TQBF $\leq_p$ Generalized Geography
**SPACE: Summary**

Defined on TMs (as usual) but largely model-independent

Time $T \subseteq$ Space $T \subseteq Time 2^{cT}$

Cor: $NP \subseteq PSPACE$

Savitch: $Nspace(S) \subseteq Dspace(S^2)$

Cor: $Pspace = NPspace (!)$

TQBF is PSPACE-complete (analog: SAT is NP-complete)

PSPACE and games (and games have serious purposes: auctions, allocation of shared resources, hacker vs firewall,...)

---

**An Analogy**

*NP is to PSPACE as Solitaire is to Chess*

I.e., NP probs involve finding a solution to a fixed, static puzzle with no adversary other than the structure of the puzzle itself. PSPACE problems, of course, just plain use poly space. But they often involve, or can be viewed as, games where an interactive adversary dynamically thwarts your progress towards a solution.

The former, tho hard, seems much easier than the later--part of the reason for the (unproven) supposition that $NP \nsubseteq PSPACE$

---

**Computability Theory**

See Midterm Review Slides

---

**Lecture 30**

Review & Wrapup
Real Computers are Finite

Unbounded “memory” is critical to most undecidability pfs
Real computers are finite: n bits of state (registers, cache, RAM, HD, …) ⇒ ≤ 2^n configs – it’s a DFA!
“Does M accept w” is decidable: run M on w; if it runs more that 2^n steps, it’s looping. (Recall LBA pfs.)
BUT:
2^n is astronomical: a modest laptop has n = 100’s of gigabits of state; # atoms in the universe ~ 2^{262}

Time & Space Complexity

Defined on TM’s but largely model-independent
(1-tape, multi-tape, RAMs, …)
Esp. if we focus on asymptotic complexity, up to polynomials
E.g. P, PSPACE
For space, model-independence even extends to nondeterministic models
For time, this is a major open problem
E.g., does P = NP?

Are “real” computer problems undecidable?

Options:
100 G is so much >> 262, let’s say it’s approximately unbounded ⇒ undecidable
Explore/quantify the “computational difficulty” of solving the (decidable) “bounded memory” problem
1st is somewhat crude, but easy, and not crazy, given that we really don’t have methods that are fundamentally better for 100Gb memories than for arbitrary algorithms
2nd is more refined but harder; goal of next few weeks is to develop theory supporting such aims

P

Many important problems are in P: solvable in deterministic polynomial time
Details are more the fodder of algorithms courses, but we’ve seen a few examples here, plus many other examples in other courses
Few problems not in P are routinely solved;
For those that are, practice is usually restricted to small instances, or we’re forced to settle for approximate, suboptimal, or heuristic “solutions”
A major goal of complexity theory is to delineate the boundaries of what we can feasibly solve
NP

The tip-of-the-iceberg in terms of problems conjectured not to be in P, but a very important tip, because
a) they’re very commonly encountered, probably because
b) they arise naturally from basic “search” and “optimization” questions.

Definition: poly time NTM
Equivalent views: poly time verifiable, “guess and check”, “is there a…” – all useful

NP-completeness

Defn & Properties of $\leq_P$
A is NP-hard: everything in NP reducible to A
A is NP-complete: NP-hard and in NP
"the hardest problems in NP"
"All alike under the skin"
Most known natural problems in NP are complete
#1: 3CNF-SAT
Many others: Clique, VertexCover, HamPath, Circuit-SAT,…

Beyond NP

“Polynomial Hierarchy”:
Quantified Boolean formulas with fixed number of alternations of $\exists$, $\forall$
Collapses if NP = co-NP
Important in helping recognize variants of NP problems
PSPACE
Exponential Time
Double-Exponential Time
…

Complexity class relationships

P $\subseteq$ NP $\cap$ co-NP $\subseteq$ NP $\cup$ co-NP $\subseteq$ PSPACE $\subseteq$ ExpTime

NP $\neq$ co-NP?
All containments above proper?
A taste of things we didn’t get to

Resource-bounded Hierarchy Theorems:

If $t(n) \ll T(n)$ (e.g., $\lim_{n \to \infty} t(n)/T(n) = 0$), then

$\text{DSPACE}(t(n)) \subseteq \text{DSPACE}(T(n))$

Similar for $\text{DTIME}$, (but fussier about "$\ll$"

E.g.: $\text{TIME}(n) \subseteq \text{TIME}(n^2) \subseteq \text{TIME}(n^3) \ldots$

$\text{P} \subseteq \text{TIME}(2^n) \subseteq \text{TIME}(3^n) \subseteq \ldots \text{TIME}(2^{2^n}) \subseteq \text{TIME}(2^{2^{2^n}})$

Method: diagonalization again

$\text{NSPACE}$ is closed under complementation

Is there an s-t path in $G$?

Is there no s-t path in $G$?

Final Exam

Monday, 2:30

In this Classroom

Two pages of notes allowed; otherwise closed book.

Coverage: comprehensive

Sipser, Chapters 3, 4, 5; 7, 8.1-8.3

Lectures

Homework

Some bias (~ 60/40) towards topics since midterm

Thanks, and Good Luck!