MA/CSSE 474
Theory of Computation

Computational Complexity

Announcements

- Don't forget the course evaluations on Banner Web.
  - If a 90%+ response rate for either section, everyone in that section gets a 5% bonus on the final exam
- Final Exam Monday 6-10PM. O 269
  - You can bring 3 double-sided sheets of paper
  - Covers whole course, but
  - Much more emphasis on later stuff
  - Includes several problems of the "which language class is this in?" flavor.

Accepting, Rejecting, Halting, and Looping

Consider :

L_1 = {<*M*, *w*>: *M* rejects *w*}.

L_2 = {<*M*, *w*>: *M* does not halt on *w*}.

L_3 = {<*M*, *w*>: *M* is a deciding TM and rejects *w*}.

What About These?

L_1 = {a}. **[in D]**

L_2 = {<*M*> : *M* accepts a}. **[in SD/D]**

L_3 = {<*M*> : *L*(*M*) = {a}}. **[HW 14]**

$L = \{<M_a, M_b> : \varepsilon \in L(M_a) - L(M_b)\}$

R is a reduction from ¬H. *R*(<*M*, *w*>) =
1. Construct the description of *M*#(*x*) that operates as follows:
   1.1. Erase the tape.
   1.2. Write *w*.
   1.3. Run *M* on *w*.
   1.4. Accept.
2. Construct the description of *M*?(*x*) that operates as follows:
   2.1. Accept.
3. Return <*M*?, *M*#>.

If *Oracle* exists and semidecides *L*, *C* = *Oracle*(*R*(<*M*, *w*>)) semidecides ¬H: *M*? accepts everything, including ε. So:

- <*M*, *w*> ∈ ¬H: *L*(*M*?) - *L*(*M*#) =

- <*M*, *w*> ∉ ¬H: *L*(*M*?) - *L*(*M*#) =

The Problem View	***The Language View***	***Status***
Does TM *M* have an even number of states?	{<*M*> : *M* has an even number of states}	D
Does TM *M* halt on *w*?	H = {<*M*, *w*> : *M* halts on *w*}	SD/D
Does TM *M* halt on the empty tape?	H_ε = {<*M*> : *M* halts on ε}	SD/D
Is there any string on which TM *M* halts?	H_{ANY} = {<*M*> : there exists at least one string on which TM *M* halts }	SD/D
Does TM *M* halt on all strings?	H_{ALL} = {<*M*> : *M* halts on Σ*}	¬SD
Does TM *M* accept *w*?	A = {<*M*, *w*> : *M* accepts *w*}	SD/D
Does TM *M* accept ε?	A_ε = {<*M*> : *M* accepts ε}	SD/D
Is there any string that TM *M* accepts?	A_{ANY} {<*M*> : there exists at least one string that TM *M* accepts }	SD/D

Does TM M accept all strings?	$A_{ALL} = \{<M> : L(M) = \Sigma^*\}$	¬SD
Do TMs M_a and M_b accept the same languages?	$EqTMs = \{<M_a, M_b> : L(M_a) = L(M_b)\}$	¬SD

Does TM M not halt on any string?	$H_{\neg ANY} = \{<M>$: there does not exist any string on which M halts$\}$	¬SD
Does TM M not halt on its own description?	$\{<M>$: TM M does not halt on input $<M>\}$	¬SD
Is TM M minimal?	$TM_{MIN} = \{<M>: M$ is minimal$\}$	¬SD
Is the language that TM M accepts regular?	$TMreg = \{<M> : L(M)$ is regular$\}$	¬SD
Does TM M accept the language A^nB^n?	$A_{anbn} = \{<M> : L(M) = A^nB^n\}$	¬SD

Undecidable Problems About CFLs

1. Given a CFL L and a string s, is $s \in L$? (decidable)
2. Given a CFL L, is $L = \varnothing$?
3. Given a CFL L, is $L = \Sigma^*$?
4. Given CFLs L_1 and L_2, is $L_1 = L_2$?
5. Given CFLs L_1 and L_2, is $L_1 \subseteq L_2$?
6. Given a CFL L, is $\neg L$ context-free?
7. Given a CFL L, is L regular?
8. Given two CFLs L_1 and L_2, is $L_1 \cap L_2 = \varnothing$?
9. Given a CFL L, is L inherently ambiguous?
10. Given PDAs M_1 and M_2, is M_2 a minimization of M_1?
11. Given a CFG G, is G ambiguous?

Complexity Classes

course.setOverviewMode(true);

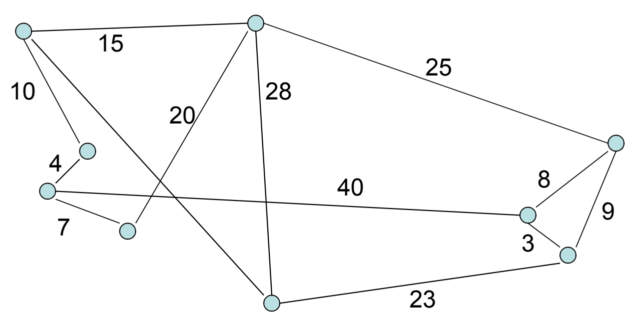
Asymptotic Analysis Review

in case it's been a while …

Are All Decidable Languages Equal?

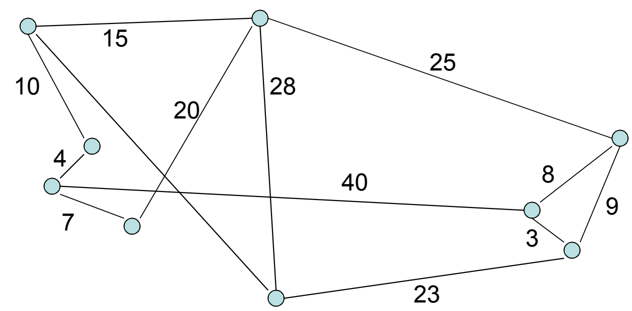
- (ab)*
- $WW^R = \{ww^R : w \in \{a, b\}^*\}$
- $WW = \{ww : w \in \{a, b\}^*\}$
- SAT = {*w* : *w* is a wff in Boolean logic and *w* is satisfiable}
- TSP (Traveling Salesman Problem). Next slides …

The Traveling Salesman Problem



Given *n* cities and the distances between each pair of them, find the shortest tour that returns to its starting point and visits each other city exactly once along the way.

The Traveling Salesman Problem



Given n cities:

Choose a first city	n
Choose a second	n-1
Choose a third	<u>n-2</u>
…	***n*!**

The Traveling Salesman Problem

Can we do better than n!

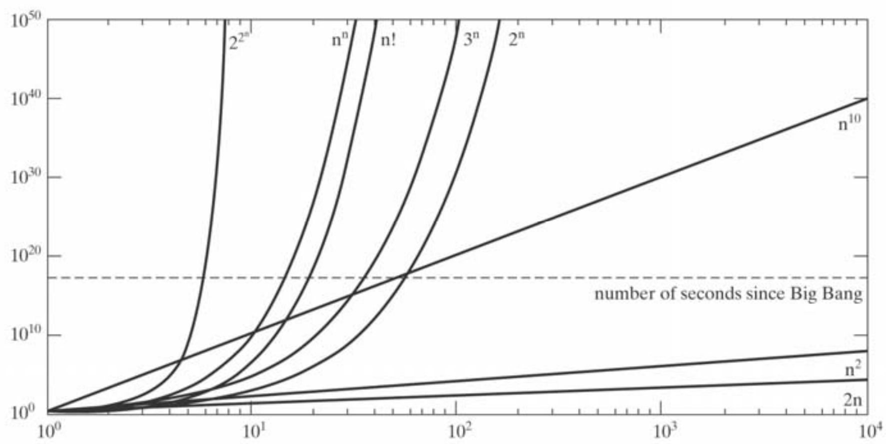
- First city doesn’t matter.
- Order doesn’t matter.

So we get (n-1!)/2.

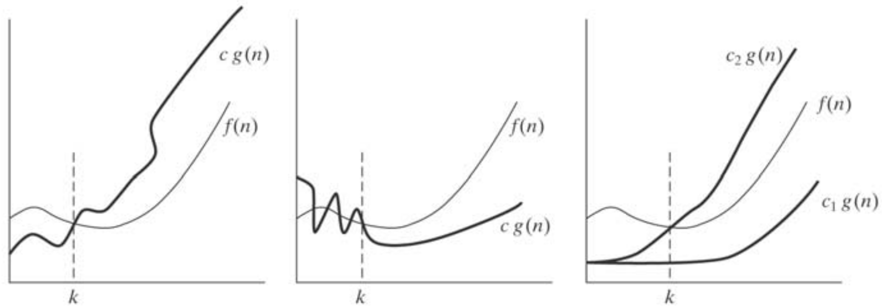
The Growth Rate of *n*!

2	2	11	479001600
3	6	12	6227020800
4	24	13	87178291200
5	120	14	1307674368000
6	720	15	20922789888000
7	5040	16	355687428096000
8	40320	17	6402373705728000
9	362880	18	121645100408832000
10	3628800	19	2432902008176640000
11	39916800	36	$3.6 \cdot 10^{41}$

Growth Rates of Functions



Asymptotic Dominance



$f(n) \in \mathcal{O}(g(n))$ $\quad$ $f(n) \in \Omega(g(n))$ $\quad$ $f(n) \in \Theta(g(n))$

Asymptotic Dominance - $\mathcal{O}$

$f(n) \in \mathcal{O}(g(n))$ iff there exists a positive integer k and a positive constant c such that:

$\forall n \geq k\ (f(n) \leq c\ g(n))$.

Alternatively, if the limit exists:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$$

Or, g grows at least as fast as f does.

Summarizing $\mathcal{O}$

$\mathcal{O}(c) \subseteq \mathcal{O}(\log_a n) \subseteq \mathcal{O}(n^b) \subseteq \mathcal{O}(d^n) \subseteq \mathcal{O}(n!) \subseteq \mathcal{O}(n^n)$

o (little oh)

Asymptotic strong upper bound: $f(n) \in o(g(n))$ iff, for every positive c, there exists a positive integer k such that:

$$\forall n \geq k\ (f(n) < c\, g(n)).$$

Alternatively, if the limit exists:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

In this case, we'll say that f is "little-oh" of g or that g grows strictly faster than f does.

Ω

- ***Asymptotic lower bound***: $f(n) \in \Omega(g(n))$ iff there exists a positive integer k and a positive constant c such that:

  $\forall n \geq k\ (f(n) \geq c\ g(n))$.

In other words, ignoring some number of small cases (all those of size less than k), and ignoring some constant factor c, $f(n)$ is bounded from below by $g(n)$.

Alternatively, if the limit exists:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

In this case, we'll say that *f* is "big-Omega" of *g* or that *g* grows no faster than *f*.

ω

- ***Asymptotic strong lower bound***: $f(n) \in \omega(g(n))$ iff, for every positive c, there exists a positive integer k such that:

  $\forall n \geq k\ (f(n) > c\ g(n))$.

Alternatively, if the required limit exists:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

In this case, we'll say that *f* is "little-omega" of *g* or that *g* grows strictly slower than *f* does.

Θ

$f(n) \in \Theta(g(n))$ iff there exists a positive integer k and positive constants c_1, and c_2 such that:

$$\forall n \geq k \ (c_1 \, g(n) \leq f(n) \leq c_2 \, g(n))$$

Or:

$f(n) \in \Theta(g(n))$ iff:
$f(n) \in \mathcal{O}(g(n))$, and
$g(n) \in \mathcal{O}(f(n))$.

Or:

$f(n) \in \Theta(g(n))$ iff:
$f(n) \in \mathcal{O}(g(n))$, and
$f(n) \in \Omega(g(n))$.

Is $n^3 \in \Theta(n^3)$?
Is $n^3 \in \Theta(n^4)$?
Is $n^3 \in \Theta(n^5)$?

Tackling Hard Problems

1. Use a technique that is guaranteed to find an optimal solution and likely to do so quickly. Linear programming:

The Concorde TSP Solver found an optimal route that visits 24,978 cities in Sweden.

http://www.tsp.gatech.edu/concorde.html

2. Use a technique that is guaranteed to run quickly and find a "good" solution, but not necessarily optimal.

http://en.wikipedia.org/wiki/Travelling_salesman_problem#Heuristic_and_approximation_algorithms

The Complexity Zoo

The attempt to characterize the decidable languages by their complexity:

http://qwiki.stanford.edu/wiki/Complexity_Zoo

See especially the Petting Zoo page.

All Problems Are Decision Problems

The Towers of Hanoi

Requires at least enough time to write the solution.

By restricting our attention to decision problems, the length of the answer is not a factor.

Encoding Types Other Than Strings

The length of the encoding matters.

Integers: use any base other than 1.

111111111111	vs	1100
111111111111111111111111111111	vs	11110

$\log_a x = \log_a b \log_b x$

- PRIMES = {w : w is the binary encoding of a prime number}

Encoding Types Other Than Strings

Graphs: use an adjacency matrix:

	1	2	3	4	5	6	7
1			•				
2				•			
3		•					
4					•		
5							
6							
7							

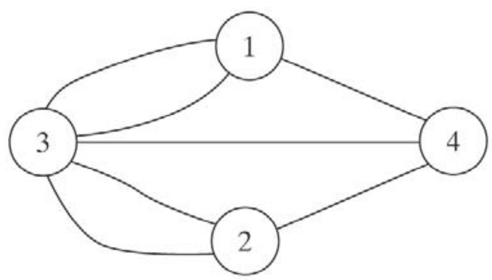
Or a list of edges:

```
101/1/11/11/10/10/100/100/101
```

Graph Languages

- CONNECTED = {<*G*> : *G* is an undirected graph and *G* is connected}.

- HAMILTONIANCIRCUIT = {<*G*> : *G* is an undirected graph that contains a ***Hamiltonian circuit***}.



Characterizing Optimization Problems as Languages

- TSP-DECIDE = {<*G*, *cost*> : <*G*> encodes an undirected graph with a positive distance attached to each of its edges and *G* contains a Hamiltonian circuit whose total cost is less than <*cost*>}.

Choosing A Model of Computation

We'll use Turing machines:

- Tape alphabet size?
- How many tapes?
- Deterministic vs. nondeterministic?

Measuring Time and Space Requirements

timereq(M) is a function of n:

- If M is a ***deterministic*** Turing machine that halts on all inputs, then:

  timereq(M) = $f(n)$ = the maximum number of steps that M executes on any input of length n.

Measuring Time and Space Requirements

- If *M* is a ***nondeterministic*** Turing machine all of whose computational paths halt on all inputs, then:

$s,\underline{\text{q}}\text{abab}$

$q_2,\text{\#}\underline{\text{a}}\text{bab}$ $\quad\quad$ $q_1,\text{q}\underline{\text{a}}\text{bab}$

$q_1,\text{qa}\underline{\text{b}}\text{ab}$ $\quad\quad$ $q_3,\underline{\text{q}}\text{bbab}$

timereq(*M*) = *f*(*n*) = the number of steps on the longest path that *M* executes on any input of length *n*.

Measuring Time and Space Requirements

spacereq(*M*) is a function of *n*:

- If *M* is a ***deterministic*** Turing machine that halts on all inputs, then:

  spacereq(*M*) = *f*(*n*) = the maximum number of tape squares that *M* reads on any input of length *n*.

- If *M* is a ***nondeterministic*** Turing machine all of whose computational paths halt on all inputs, then:

  spacereq(*M*) = *f*(*n*) = the maximum number of tape squares that *M* reads on any path that it executes on any input of length *n*.

Algorithmic Gaps

We'd like to show for a language L:

1. Upper bound: There exists an algorithm that decides *L* and that has complexity C_1.
2. Lower bound: Any algorithm that decides *L* must have complexity at least C_2.
3. $C_1 = C_2$.

If $C_1 = C_2$, we are done. Often, we're not done.

Algorithmic Gaps

Example: TSP

- Upper bound: *timereq* $\in \mathcal{O}(2^{(n^k)})$.
- Don't have a lower bound that says polynomial isn't possible.

We group languages by what we know. And then we ask: "Is class CL_1 equal to class CL_2?"

A Simple Example of Polynomial Speedup

Given a list of *n* numbers, find the minimum and the maximum elements in the list. Or, as a language recognition problem:

L = {<*list of numbers*, $number_1$, $number_2$>:
$number_1$ is the minimum element of the list and
$number_2$ is the maximum element}.

(23, 45, 73, 12, 45, 197; 12; 197) $\in L$.

A Simple Example of Polynomial Speedup

The straightforward approach:

simplecompare(*list*: list of numbers) =
max = *list*[1].
min = *list*[1].
For *i* = 2 to length(*list*) do:
If *list*[*i*] < *min* then *min* = *list*[*i*].
If *list*[*i*] > *max* then *max* = *list*[*i*].

Requires 2(*n*-1) comparisons. So *simplecompare* is $\mathcal{O}(n)$.

But we can solve this problem in (3/2)(*n*-1) comparisons.

How?

A Simple Example of Polynomial Speedup

efficientcompare(*list*: list of numbers) =
 max = *list*[1].
 min = *list*[1].
 For *i* = 3 to length(*list*) by 2 do:
  If *list*[*i*] < list[*i*-1] then:
   If *list*[*i*] < *min* then *min* = *list*[*i*].
   If *list*[*i*-1] > *max* then *max* = *list*[*i*-1].
  Else:
   If *list*[*i*-1] < min then min = *list*[*i*-1].
   If *list*[*i*] > max then max = *list*[*i*].
 If length(*list*) is even then check the last element.

Requires 3/2(*n*-1) comparisons.

String Search

```
t:   a b c a b a b c a b d

p:   a b c d

       a b c d

         a b c d

             . . .
```

String Search

simple-string-search(t, p: strings) =
    i = 0.
    j = 0.
    While $i \leq |t| - |p|$ do:
        While $j < |p|$ do:
            If $t[i+j] = p[j]$ then $j = j + 1$.
                Else exit this loop.
            If $j = |p|$ then halt and accept.
                Else:
                    $i = i + 1$.
                    $j = 0$.
    Halt and reject.

Let n be $|t|$ and let m be $|p|$. In the worst case (in which it doesn't find an early match), *simple-string-search* will go through its outer loop almost n times and, for each of those iterations, it will go through its inner loop m times.

So *timereq*(*simple-string-search*) $\in \mathcal{O}(nm)$.

K-M-P algorithm is $\mathcal{O}(n+m)$

Replacing an Exponential Algorithm with a Polynomial One

- Context-free parsing can be done in $\mathcal{O}(n^3)$ time instead of $\mathcal{O}(2^n)$ time. (CYK algorithm)
- Finding the greatest common divisor of two integers can be done in $\mathcal{O}(\log_2(max(n, m)))$ time instead of exponential time.

The Language Class P

$L \in$ P iff

- there exists some deterministic Turing machine M that decides L, and
- *timereq*(M) $\in \mathcal{O}(n^k)$ for some k.

We'll say that L is ***tractable*** iff it is in P.

Closure under Complement

Theorem: The class P is closed under complement.

Proof: If M accepts L in polynomial time, swap accepting and non accepting states to accept $\neg L$ in polynomial time.

Defining Complement

- CONNECTED = {<G> : G is an undirected graph and G is connected} is in P.
- NOTCONNECTED = {<G> : G is an undirected graph and G is not connected}.
- $\neg$CONNECTED = NOTCONNECTED $\cup$ {strings that are not syntactically legal descriptions of undirected graphs}.

$\neg$CONNECTED is in P by the closure theorem. What about NOTCONNECTED?

If we can check for legal syntax in polynomial time, then we can consider the universe of strings whose syntax is legal. Then we can conclude that NOTCONNECTED is in P if CONNECTED is.

Languages That Are in P

- Every regular language.
- Every context-free language since there exist context-free parsing algorithms that run in $\mathcal{O}(n^3)$ time.
- Others:
  - $A^nB^nC^n$
  - Nim

To Show That a Language Is In P

- Describe a one-tape, deterministic Turing machine.
- It may use multiple tapes. Price:
- State an algorithm that runs on a conventional computer.
  Price:

How long does it take to compare two strings?

q	a	a	a	;	a	a	a	q	…

Bottom line: If ignoring polynomial factors, then just describe a deterministic algorithm.

Regular Languages

Theorem: Every regular language can be decided in linear time. So every regular language is in P.

Proof: If *L* is regular, there exists some DFSM *M* that decides it. Construct a deterministic TM *M′* that simulates *M*, moving its read/write head one square to the right at each step. When *M′* reads a q, it halts. If it is in an accepting state, it accepts; otherwise it rejects.

On any input of length *n*, *M′* will execute *n* + 2 steps.

So $timereq(M') \in \mathcal{O}(n)$.

Context-Free Languages

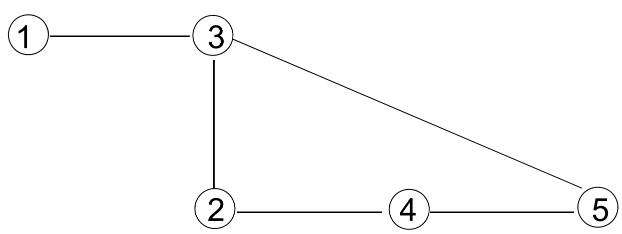
Theorem: Every context-free language can be decided in $\mathcal{O}(n^{18})$ time. So every context-free language is in P.

Proof: The Cocke-Kasami-Younger (CKY) algorithm can parse any context-free language in time that is $\mathcal{O}(n^3)$ if we count operations on a conventional computer. That algorithm can be simulated on a standard, one-tape Turing machine in $\mathcal{O}(n^{18})$ steps.

WE could get bogged down in the details of this, but w ewon't!

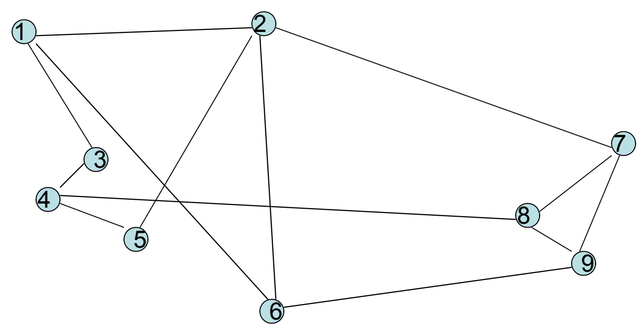
Graph Languages

Represent a graph $G = (V, E)$ as a list of edges:



`101/1/11/11/10/10/100/100/101/11/101`

Graph Languages



CONNECTED =
{<*G*> : *G* is an undirected graph and
G is connected}.

Is CONNECTED in P?

CONNECTED is in P

connected(<*G* = (*V*, *E*>) =

1. Set all vertices to be unmarked.
2. Mark vertex 1.
3. Initialize *L* to {1}.
4. Initialize *marked-vertices-counter* to 1.
5. Until *L* is empty do:
    - 5.1. Remove the first element from *L*. Call it *current-vertex*.
    - 5.2. For each edge *e* that has *current-vertex* as an endpoint do:
        Call the other endpoint of *e* *next-vertex*. If *next-vertex* is not already marked then do:
        - Mark *next-vertex*.
        - Add *next-vertex* to *L*.
        - Increment *marked-vertices-counter* by 1.
6. If *marked-vertices-counter* = |*V*| accept. Else reject.

Analyzing *connected*

- Step 1 takes time that is $\mathcal{O}(|V|)$.
- Steps 2, 3, and 4 each take constant time.
- The loop of step 5 can be executed at most $|V|$ times.
  - Step 5.1 takes constant time.
  - Step 5.2 can be executed at most $|E|$ times. Each time, it requires at most $\mathcal{O}(|V|)$ time.
- Step 6 takes constant time.

So *timereq*(*connected*) is:

$|V|\cdot\mathcal{O}(|E|)\cdot\mathcal{O}(|V|) = \mathcal{O}(|V|^2|E|)$.

But $|E| \leq |V|^2$. So *timereq*(*connected*) is:

$\mathcal{O}(|V|^4)$.

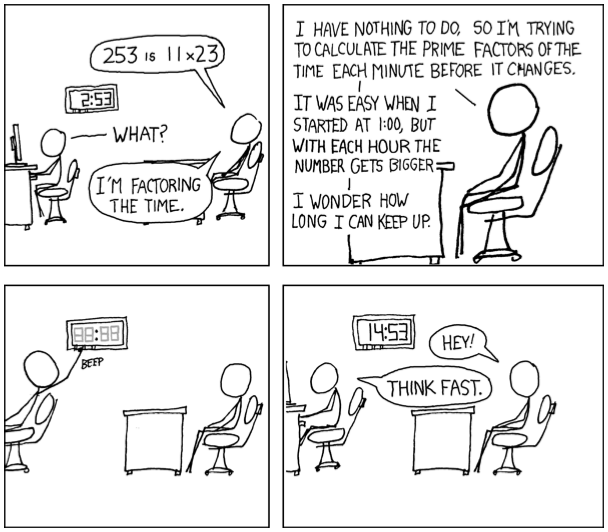
Primality Testing

RELATIVELY-PRIME =
{<*n*, *m*> : *n* and *m* are integers that are relatively prime}.

PRIMES =
{*w* : *w* is the binary encoding of a prime number}

COMPOSITES =
{*w* : *w* is the binary encoding of a nonprime number}

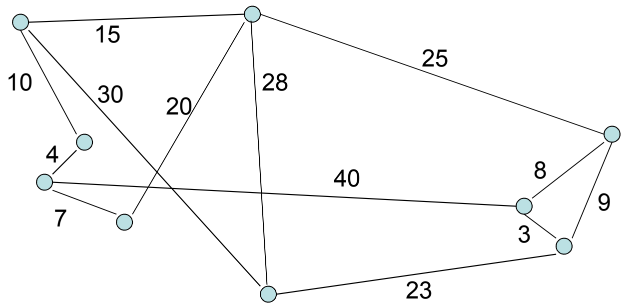
But Finding Factors Remains Hard



http://xkcd.com/247/

Returning to TSP

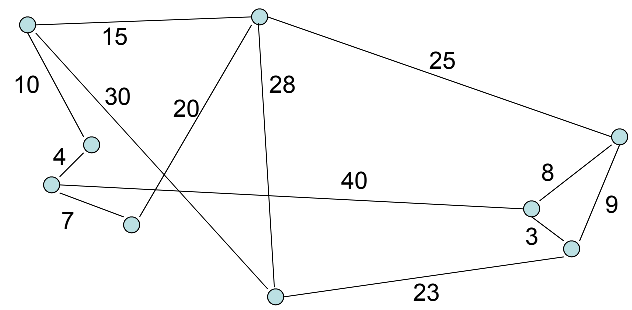
TSP-DECIDE = {<*G*, *cost*> : <*G*> encodes an undirected graph with a positive distance attached to each of its edges and *G* contains a Hamiltonian circuit whose total cost is less than <*cost*>}.



An NDTM to decide TSP-DECIDE:

Returning to TSP

An NDTM to decide TSP-DECIDE:



1. For i = 1 to |V| do:
   Choose a vertex that hasn't yet been chosen.
2. Check that the path defined by the chosen sequence of vertices is a Hamiltonian circuit through G with distance less than cost.

TSP and Other Problems Like It

TSP-DECIDE, and other problems like it, share three properties:

1. The problem can be solved by searching through a space of partial solutions (such as routes). The size of this space grows exponentially with the size of the problem.

2. No better (i.e., not based on search) technique for finding an exact solution is known.

3. But, if a proposed solution were suddenly to appear, it could be checked for correctness very efficiently.

The Language Class NP

Nondeterministic deciding:

$L \in$ NP iff:

- there is some NDTM M that decides L, and
- $timereq(M) \in \mathcal{O}(n^k)$ for some k.

NDTM deciders:

$s,\underline{q}abab$

$q_2,\#\underline{a}bab$ $q_1,q\underline{a}bab$

$q_1,qa\underline{b}ab$ $q_3,\underline{q}bbab$

TSP Again

TSP-DECIDE = {<*G*, *cost*> : <*G*> encodes an undirected graph with a positive distance attached to each of its edges and *G* contains a Hamiltonian circuit whose total cost is less than <*cost*>}.

Suppose some Oracle presented a candidate path *c*:

<*G*, *cost*, v_1, v_7, v_4, v_3, v_8, v_5, v_2, v_6, v_1>

How long would it take to verify that *c* proves that:

<*G*, *cost*> is in TSP-DECIDE?

Deterministic Verifying

A Turing machine V is a ***verifier*** for a language L iff:

$w \in L$ iff $\exists c$ ($<w, c> \in L(V)$).

We'll call c a ***certificate***.

Deterministic Verifying

An alternative definition for the class NP:

$L \in$ NP iff there exists a deterministic TM V such that:

- V is a verifier for L, and
- $timereq(V) \in \mathcal{O}(n^k)$ for some k.

ND Deciding and D Verifying

Theorem: These two definitions are equivalent:

(1) $L \in$ NP iff there exists a nondeterministic, polynomial-time TM that decides it.
(2) $L \in$ NP iff there exists a deterministic, polynomial-time verifier for it.

Proof: WE skip it

Proving That a Language is in NP

- Exhibit an NDTM to decide it.
- Exhibit a DTM to verify it.

Example

- SAT = {w : w is a Boolean wff and w is satisfiable} is in NP.

$F_1 = P \wedge Q \wedge \neg R$?
$F_2 = P \wedge Q \wedge R$?
$F_3 = P \wedge \neg P$?
$F_4 = P \wedge (Q \vee \neg R) \wedge \neg Q$?

SAT-decide(F_4) =

SAT-verify (<F_4, (P = *True*, Q = *False*, R = *False*)>) =

3-SAT

- A ***literal*** is either a variable or a variable preceded by a single negation symbol.
- A ***clause*** is either a single literal or the disjunction of two or more literals.
- A wff is in ***conjunctive normal form*** (or CNF) iff it is either a single clause or the conjunction of two or more clauses.
- A wff is in ***3-conjunctive normal form*** (or 3-CNF) iff it is in conjunctive normal form and each clause contains exactly three literals.

3-SAT

	3-CNF	CNF
$(P \vee \neg Q \vee R)$	•	•
$(P \vee \neg Q \vee R) \wedge (\neg P \vee Q \vee \neg R)$	•	•
P		•
$(P \vee \neg Q \vee R \vee S) \wedge (\neg P \vee \neg R)$		•
$P \rightarrow Q$		
$(P \wedge \neg Q \wedge R \wedge S) \vee (\neg P \wedge \neg R)$		
$\neg(P \vee Q \vee R)$		

Every wff can be converted to an equivalent wff in CNF.

- 3-SAT = { *w* : *w* is a wff in Boolean logic,
  w is in 3-conjunctive normal form, and
  w is satisfiable}.

Is 3-SAT in NP?

The Relationship Between P and NP

Is P = NP?

Here are some things we know:

$P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$

$P \neq EXPTIME$

The Millenium Prize

Using Reduction in Complexity Proofs

A ***mapping reduction*** R from L_1 to L_2 is a

- Turing machine that
- implements some computable function f with the property that:

$$\forall x\ (x \in L_1 \leftrightarrow f(x) \in L_2).$$

If $L_1 \leq L_2$ and M decides L_2, then:

$$C(x) = M(R(x)) \text{ will decide } L_1.$$

Using Reduction in Complexity Proofs

If R is deterministic polynomial then:

$$L_1 \leq_P L_2.$$

And, whenever such an R exists:

- L_1 must be in P if L_2 is: if L_2 is in P then there exists some deterministic, polynomial-time Turing machine M that decides it. So $M(R(x))$ is also a deterministic, polynomial-time Turing machine and it decides L_1.

- L_1 must be in NP if L_2 is: if L_2 is in NP then there exists some nondeterministic, polynomial-time Turing machine M that decides it. So $M(R(x))$ is also a nondeterministic, polynomial-time Turing machine and it decides L_1.

Why Use Reduction?

Given $L_1 \leq_P L_2$, we can use reduction to:

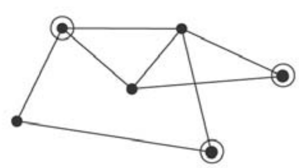
- Prove that L_1 is in P or in NP because we ***already know*** that L_2 is.

- Prove that L_1 would be in P or in NP if we ***could somehow show*** that L_2 is. When we do this, we cluster languages of similar complexity (even if we're not yet sure what that complexity is). In other words, L_1 is no harder than L_2 is.

INDEPENDENT-SET

- INDEPENDENT-SET = {<*G*, *k*> : *G* is an undirected graph and *G* contains an independent set of at least *k* vertices}.

An ***independent set*** is a set of vertices no two of which are adjacent (i.e., connected by a single edge. So, in the following graph, the circled vertices form an independent set:



In a scheduling program the vertices represent tasks and are connected by an edge if their corresponding tasks conflict. We can find the largest number of tasks that can be scheduled at the same time by finding the largest independent set in the task graph.

3-SAT and INDEPENDENT-SET

3-SAT $\leq_P$ INDEPENDENT-SET.

Strings in 3-SAT describe formulas that contain literals and clauses.

$(P \vee Q \vee \neg R) \wedge (R \vee \neg S \vee Q)$

Strings in INDEPENDENT-SET describe graphs that contain vertices and edges.

`101/1/11/11/10/10/100/100/101/11/101`

Gadgets

A ***gadget*** is a structure in the target language that mimics the role of a corresponding structure in the source language.

Example: 3-SAT $\leq_P$ INDEPENDENT-SET.

$$(P \vee Q \vee \neg R) \wedge (R \vee \neg S \vee Q)$$

$$\Big\Downarrow \quad \text{(approximately)}$$

`101/1/11/11/10/10/100/100/101/11/101`

So we need:

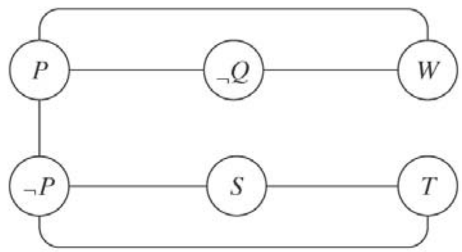
- a gadget that looks like a graph but that mimics a literal, and
- a gadget that looks like a graph but that mimics a clause.

3-SAT $\leq_P$ INDEPENDENT-SET

R(<*f*: Boolean formula with *k clauses*>) =

1. Build a graph *G* by doing the following:
    1.1. Create one vertex for each instance of each literal in *f*.
    1.2. Create an edge between each pair of vertices for symbols in the same clause.
    1.3. Create an edge between each pair of vertices for complementary literals.
2. Return <*G*, *k*>.

$(P \vee \neg Q \vee W) \wedge (\neg P \vee S \vee T)$:

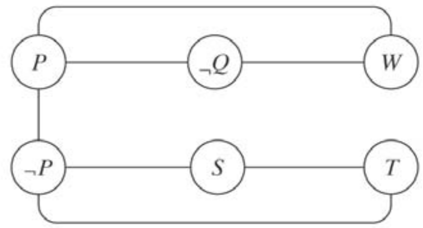


R is Correct

Show: $f \in$ 3-SAT iff *R*(<*f*>) $\in$ INDEPENDENT-SET by showing:

- $f \in$ 3-SAT $\rightarrow$ *R*(<*f*>) $\in$ INDEPENDENT-SET
- *R*(<*f*>) $\in$ INDEPENDENT-SET $\rightarrow f \in$ 3-SAT

One Direction



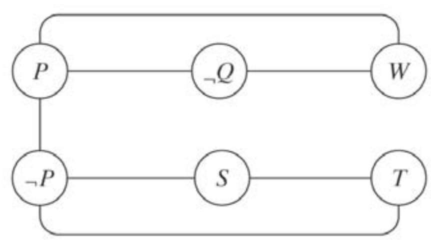
$f \in$ 3-SAT $\rightarrow R(\langle f \rangle) \in$ INDEPENDENT-SET:
There is a satisfying assignment *A* to the symbols in *f*.
So *G* contains an independent set *S* of size *k*, built by:

1. From each clause gadget choose one literal that is made positive by *A*.
2. Add the vertex corresponding to that literal to *S*.

S will contain exactly *k* vertices and is an independent set:

- No two vertices come from the same clause so step 1.2 could not have created an edge between them.
- No two vertices correspond to complimentary literals so step 1.3 could not have created an edge between them.

The Other Direction



- $R(\langle f \rangle) \in$ INDEPENDENT-SET.
- So the graph *G* that *R* builds contains an independent set *S* of size *k*.
- We prove that there is some satisfying assignment *A* for *f*:

No two vertices in *S* come from the same clause gadget. Since *S* contains at least *k* vertices, no two are from the same clause, and *f* contains *k* clauses, *S* must contain one vertex from each clause.

Build *A* as follows:

1. Assign *True* to each literal that corresponds to a vertex in *S*.
2. Assign arbitrary values to all other literals.

Since each clause will contain at least one literal whose value is *True*, the value of *f* will be *True*.

NP-Completeness

A language *L* might have these properties:

1. *L* is in NP.
2. Every language in NP is deterministic, polynomial-time reducible to *L*.

- *L* is ***NP-hard*** iff it possesses property 2.

  An NP-hard language is at least as hard as any other language in NP.

- *L* is NP-***complete*** iff it possesses *both* property 1 and property 2.

  All NP-complete languages can be viewed as being equivalently hard.

NP-Hard vs. NP-Complete

An example: puzzles vs. games (Appendix N).

To use this theory to analyze a game like chess, we must generalize it so that we can talk about solution time as a function of problem size:

CHESS = {<*b*>: *b* is a configuration of an $n \times n$ chess board and there is a guaranteed win for the current player}.

Sudoku

- SUDOKU = {<*b*>: *b* is a configuration of an $n \times n$ grid and *b* has a solution under the rules of Sudoku}.

	5			9	4	2	1	
4							8	
	3		7					
				2				4
2			4		6			8
6				3				
					8		6	
	7							3
	6	8	9	5			4	

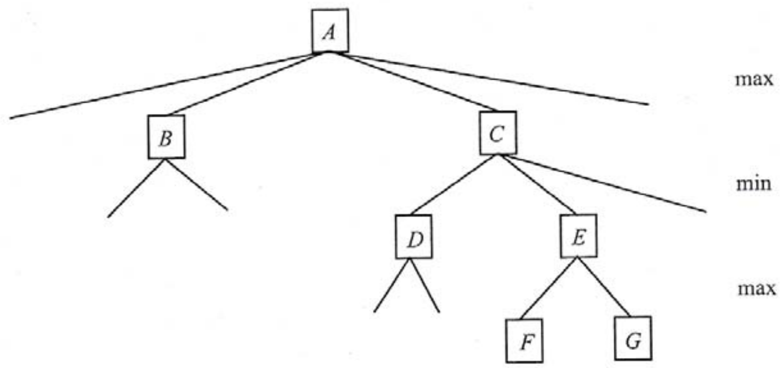
Sudoku

- SUDOKU = {<*b*>: *b* is a configuration of an $n \times n$ grid and *b* has a solution under the rules of Sudoku}.

A deterministic, polynomial time verifier for SUDOKU, on input:

<*b*, (1,1,1), (1,2,5), (1,3,4), …>

Chess



A deterministic polynomial time verifier for CHESS?

NP-Hard vs. NP-Complete

SUDOKU = {<*b*>: *b* is a configuration of an $n \times n$ grid and *b* has a solution under the rules of Sudoku}.

NP-complete.

CHESS = {<*b*>: *b* is a configuration of an $n \times n$ chess board and there is a guaranteed win for the current player}.

NP-hard, not thought to be in NP.
If fixed number of pieces: PSPACE-complete.
If varialbe number of pieces: EXPTIME-complete.

Showing that *L* is NP-Complete

How about: Take a list of known NP languages and crank out the reductions?

$NPL_1 \geq L$

$NPL_2 \geq L$

$NPL_3 \geq L$

…

Showing that *L* is NP-Complete

Suppose we had one NP-complete language L':

$$\underbrace{NPL_1 \quad NPL_2 \quad NPL_3 \quad NPL_4 \quad NPL\ldots}_{L'}$$

$$\downarrow$$

$$L$$

Finding an *L*′

- The key property that every NP language has is that it can be decided by a polynomial time NDTM.
- So we need a language in which we can describe computations of NDTMs.

The Cook-Levin Theorem

Define: SAT = {*w* : *w* is a wff in Boolean logic and *w* is satisfiable}

Theorem: SAT is NP-complete.

Proof:

- SAT is in NP.
- SAT is NP-hard.

NP-Complete Languages

- SUBSET-SUM = {<*S*, *k*> : *S* is a multiset of integers, *k* is an integer, and there exists some subset of *S* whose elements sum to *k*}.
- SET-PARTITION = {<*S*> : *S* is a multiset of objects each of which has an associated cost and there exists a way to divide *S* into two subsets, *A* and *S* - *A,* such that the sum of the costs of the elements in *A* equals the sum of the costs of the elements in *S* - *A*}.
- KNAPSACK = {<*S*, *v*, *c*> : *S* is a set of objects each of which has an associated cost and an associated value, *v* and *c* are integers, and there exists some way of choosing elements of *S* (duplicates allowed) such that the total cost of the chosen objects is at most *c* and their total value is at least *v*}.

NP-Complete Languages

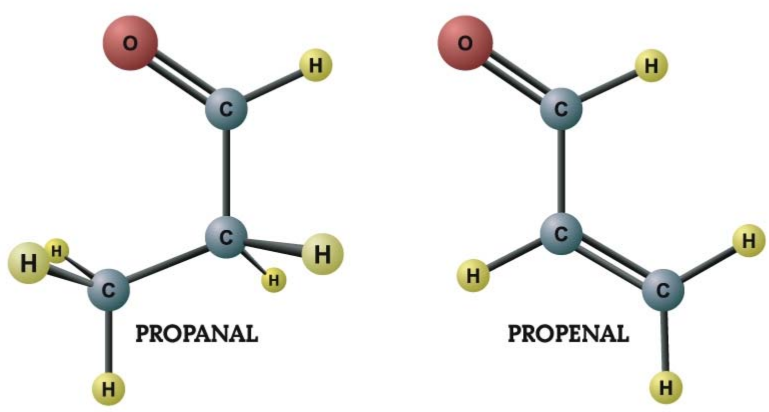
- TSP-DECIDE.
- HAMILTONIAN-PATH = {<*G*> : *G* is an undirected graph and *G* contains a Hamiltonian path}.
- HAMILTONIAN-CIRCUIT = {<*G*> : *G* is an undirected graph and *G* contains a Hamiltonian circuit}.
- CLIQUE = {<*G*, *k*> : *G* is an undirected graph with vertices *V* and edges *E*, *k* is an integer, $1 \leq k \leq |V|$, and *G* contains a *k*-clique}.
- INDEPENDENT-SET = {<*G*, *k*> : *G* is an undirected graph and *G* contains an independent set of at least *k* vertices}.

NP-Complete Languages

• SUBGRAPH-ISOMORPHISM = {<G_1, G_2> :
G_1 is isomorphic to some subgraph of G_2}.

Two graphs *G* and *H* are ***isomorphic*** to each other iff there exists a way to rename the vertices of *G* so that the result is equal to *H*. Another way to think about isomorphism is that two graphs are isomorphic iff their drawings are identical except for the labels on the vertices.

SUBGRAPH-ISOMORPHISM



NP-Complete Languages

- BIN-PACKING = {<*S*, *c*, *k*> : *S* is a set of objects each of which has an associated size and it is possible to divide the objects so that they fit into *k* bins, each of which has size *c*}.

BIN-PACKING

In three dimensions:

NP-Complete Languages

- SHORTEST-SUPERSTRING = {<*S*, *k*> : *S* is a set of strings and there exists some superstring *T* such that every element of *S* is a substring of *T* and *T* has length less than or equal to *k*}.

SHORTEST-SUPERSTRING

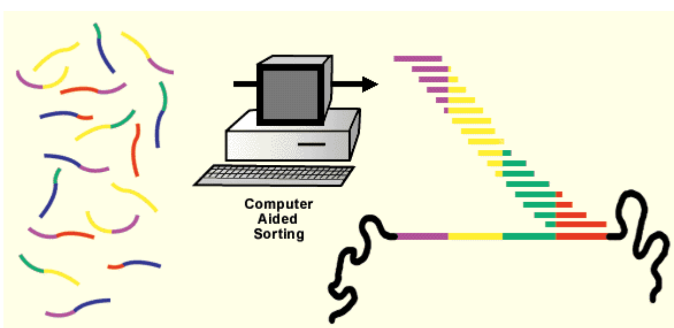


Fig 2: Short fragments of DNA sequence are ordered by overlapping data to recreate the whole genome sequence

Source: Wiley: Interactive Concepts in Biology

Proving that *L* is NP-Complete

NPL_1 NPL_2 NPL_3 NPL_4 NPL...

L_1

$\downarrow$

L_2

Theorem:

If:

L_1 is NP-complete,

$L_1 \leq_P L_2$, and

L_2 is in NP,

Then L_2 is also NP-complete.