CSC373: Algorithm Design, Analysis and Complexity
Winter/Spring 2020

Allan Borodin and Sara Rahmati

Week of March 2-6, 2020

Announcements

- Requests for regrading are normally within one week of solutions being discussed or posted. For assignment 1, we will accept regrading requests up to March 2. Regrading requests for assignments must be made on Markus. All requests must have a one or two paragraph explanation as to why you think a question was not given the proper grade.
- Assignment 2 is due March 5 at 4:59. There is a correction for Q1 on A2. The desired time bound is $O(n^2 \cdot \max_i v_i)$.
- We expect to start listing questions for Assignment 3 by the end of this week.

This weeks agenda

- Going over two transformations: vextrex-cover $\leq_p$ set-cover, and 3SAT $\leq_p$ 3-COLOR.
- Proving 3SAT is NP complete.
- Reducing search and optimization problems to the corresponding decision problem.
- NP vs $co-NP$; Factoring in $NP \cap co-NP$
- Other aspects of complexity theory
- What is IP and LP?

Next week LP (only one week) and then 3 weeks for approximation algorithms and randomized algorithms.

The VertexCover $\leq_p$ SetCover transformation

Let $G = (V, E)$. A subset $V' \subset V$ is a vertex cover for G if every edge $e \in E$ is adjacent to at least one vertex in V'. In the VertexCover decision problem, we are given (G, k) and we need to determiine if G has a vertex cover of size k.

Let $U = \{u_1, \ldots, u_m\}$ be a set and let $\mathcal{C} = \{S_1, \ldots, S_n\}$ be a collection of subsets of U; that is, $S_i \subseteq U$. A subcollection $\mathcal{C}' \subset \mathcal{C}$ is a set cover for U if every u_j is in some $S_i \in \mathcal{C}'$. In the SetCover decision problem, we are given $(U, \mathcal{C}, k)$ and we need to determine if there is a subcollection $\mathcal{C}'$ of size k that covers U.

We showed last week that IndpendentSet $\leq_p$ VertexCover. Here we show that VertexCover $\leq_p$ SetCover.

For this transfiormation, given (G, k), let $U = \{e_1, \ldots, e_m\}$ be the edges of G and let $\mathcal{C} = \{V_1, \ldots, V_n\}$ where $V_i = \{e_j | e_j \text{ is adjacent to } v_i\}$.

VertexCover is a very special case of SetCover where every element (i.e,, an edge e) occurs in exactly two sets.

3CNF $\leq_p$ 3-COLOR: Outline of Transformation

Claim. 3-SAT $\leq_P$ 3-COLOR.

Pf. Given 3-SAT instance Φ, we construct an instance of 3-COLOR that is 3-colorable iff Φ is satisfiable.

Construction.

i. For each literal, create a node.
ii. Create 3 new nodes T, F, B; connect them in a triangle, and connect each literal to B.
iii. Connect each literal to its negation.
iv. For each clause, add gadget of 6 nodes and 13 edges.

$\uparrow$
to be described next

If ϕ is a 3CNF formula in n variables and m clauses, then $h(\phi) = G_\phi$ will have $2n + 6m + 3$ nodes and $3n + 13m + 3$ edges.

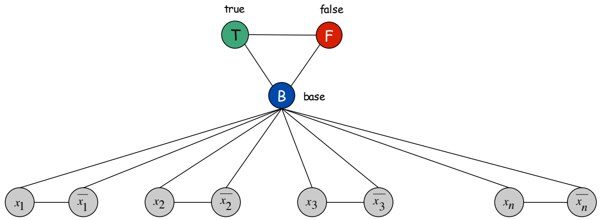
3CNF $\leq_p$ 3-COLOR: Consistent literals

This slide and the next two slides are perhaps better thought of as illustrating that if Φ is not satisfiable then then G is not 3 colorable.

Claim. Graph is 3-colorable iff Φ is satisfiable.

Pf. $\Rightarrow$ Suppose graph is 3-colorable.

- Consider assignment that sets all T literals to true.
- (ii) ensures each literal is T or F.
- (iii) ensures a literal and its negation are opposites.

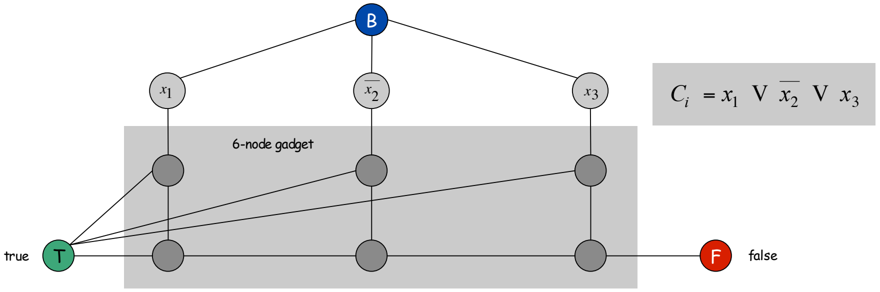


3CNF $\leq_p$ 3-COLOR: The clause gadget

Claim. Graph is 3-colorable iff Φ is satisfiable.

Pf. $\Rightarrow$ Suppose graph is 3-colorable.

- Consider assignment that sets all T literals to true.
- (ii) ensures each literal is T or F.
- (iii) ensures a literal and its negation are opposites.
- (iv) ensures at least one literal in each clause is T.

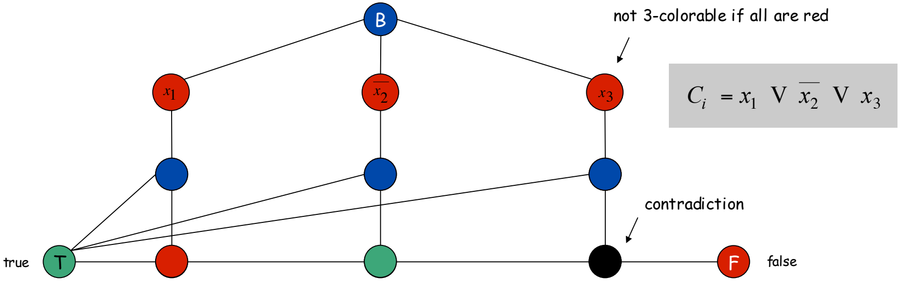


G_ϕ is 3-colourable $\Rightarrow$ ϕ satisfiable

Claim. Graph is 3-colorable iff Φ is satisfiable.

Pf. $\Rightarrow$ Suppose graph is 3-colorable.

- Consider assignment that sets all T literals to true.
- (ii) ensures each literal is T or F.
- (iii) ensures a literal and its negation are opposites.
- (iv) ensures at least one literal in each clause is T.



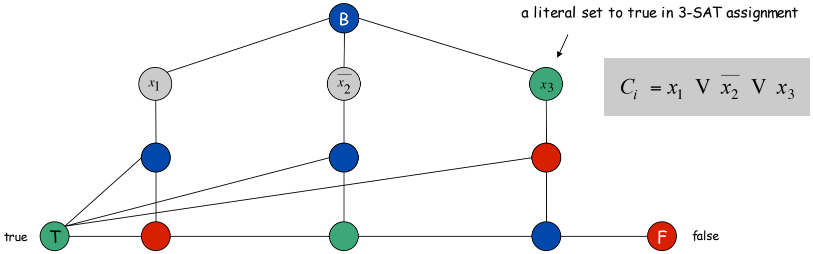
B
x_1
$\overline{x_2}$
x_3
not 3-colorable if all are red
$C_i = x_1 \vee \overline{x_2} \vee x_3$
contradiction
true
T
F
false

ϕ satisfiable $\Rightarrow$ G_ϕ is 3-colourable

Claim. Graph is 3-colorable iff Φ is satisfiable.

Pf. $\Leftarrow$ Suppose 3-SAT formula Φ is satisfiable.

- Color all true literals T.
- Color node below green node F, and node below that B.
- Color remaining middle row nodes B.
- Color remaining bottom nodes T or F as forced. ▪



a literal set to true in 3-SAT assignment
B
x_1
$\overline{x_2}$
x_3
$C_i = x_1 \vee \overline{x_2} \vee x_3$
true
T
F
false

- Note we are choosing precisely one green node in each clause to force a proper colouring.

Finally, how does one show that 3SAT is *NP*-complete

We now know how to generate more and more *NP*-compplete problems **assumming** that 3SAT is *NP*-complete. But how do we know that 3SAT (or any problem if we don't use 3SAT) is *NP*-complete. Without at least one *NP*-hard problem to begin, we cannot generate other *NP*-hard problems.

Following Cook's original proof, we can show how we can encode Turing machine computations in the language of propositional logic. This will yield the desired fresult that 3SAT is *NP*-hard (and therefore *NP*-complete).

CLRS uses a circuit value problem relying on our understanding of Boolean circuits. Although perhaps more intuitive, there is some hand-waving in that approach whereas the TM encoding shows how we can make everything precise. Note: We may not be proving all the claims but argue that there is enough detail to be convinced that this encoding of TM computations is sufficient to obtain the dsesired result.

Turing machines

- We are using the classical one tape TM. This is the simplest TM variant to formalize which will enable the proof for the NP completeness of SAT. In the proof, we are assuming (without loss of generality) that all time bounds $T(n)$ are computable in polynomial time.
- Claim Any reasonable (classical) computing model algorithm running in time $T(n)$, can be simulated by a TM in time $T(n)^k$ for some k. Hence we can use the TM model in the definition of P and NP.
- Since we are only considering decision problems we will view TMs that are defined for decision problems and hence do not need an output other than a reject and accept state.
- Following standard notation, formally, a specific TM is defined by a tuple $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$
- We will briefly explain (using the board) the model and notation. Note that Q, Σ, Γ are all finite sets.

Satisfiability is NP-Complete

- SAT = { < ϕ > | ϕ is a satisfiable Boolean formula }
- Theorem: SAT is NP-complete.
- Lemma 1: SAT $\in$ NP.
- Lemma 2: SAT is NP-hard.
- Proof of Lemma 1:
  - Recall: L $\in$ NP if and only if ($\exists$ V, poly-time verifier) ($\exists$ p, poly)
    x $\in$ L iff ($\exists$ c, $|c| \leq p(|x|)$) [V(x, c) accepts]
  - So, to show SAT $\in$ NP, it's enough to show ($\exists$ V) ($\exists$ p)
    ϕ $\in$ SAT iff ($\exists$ c, $|c| \leq p(|x|)$) [V(ϕ, c) accepts]
  - We know: ϕ $\in$ SAT iff there is an assignment to the variables such that ϕ with this assignment evaluates to 1.
  - So, let certificate c be the assignment.
  - Let verifier V take a formula ϕ and an assignment c and accept exactly if ϕ with c evaluates to true.
  - Evaluate ϕ bottom-up, takes poly time.

Satisfiability is NP-Complete

- Lemma 2: SAT is NP-hard.
- Proof of Lemma 2:
  - Need to show that, for any $A \in NP$, $A \leq_p SAT$.
  - Fix $A \in NP$.
  - Construct a poly-time f such that

    $$w \in A \text{ if and only if } \overset{\text{A formula, write it as } \phi_w\text{.}}{f(w)} \in SAT.$$

  - By definition, since $A \in NP$, there is a nondeterministic TM M that decides A in polynomial time.
  - Fix polynomial p such that M on input w always halts, on all branches, in time $\leq p(|w|)$; assume $p(|w|) \geq |w|$.
  - $w \in A$ if and only if there is an accepting computation history (CH) of M on w.

Satisfiability is NP-Complete

- Lemma 2: SAT is NP-hard.
- Proof, cont'd:
  - Need $w \in A$ if and only if $f(w)$ ($= \phi_w$) $\in$ SAT.
  - $w \in A$ if and only if there is an accepting CH of M on w.
  - So we must construct formula ϕ_w to be satisfiable iff there is an accepting CH of M on w.
  - Recall definitions of computation history and accepting computation history from Post Correspondence Problem: $\# \; C_0 \; \# \; C_1 \; \# \; C_2$
    - Configurations include tape contents, state, head position.
  - We construct ϕ_w to describe an accepting CH.
  - Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ as usual.
  - Instead of lining up configs in a row as before, arrange in $(p(|w|) + 1)$ row $\times$ $(p(|w|) + 3)$ column matrix:

Proof that SAT is NP-hard

- ϕ_w will be satisfiable iff there is an accepting CH of M on w.
- Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$.
- Arrange configs in $(p(|w|) + 1) \times (p(|w|) + 3)$ matrix:

$$\begin{array}{cccccccccccc}
\# & q_0 & w_1 & w_2 & w_3 & & w_n & -- & -- & & -- & \# \\
\# & & & & & & & & & & & \# \\
\# & & & & & & & & & & & \# \\
\vdots & & & & & & & & & & & \vdots \\
\# & & & & & & & & & & & \#
\end{array}$$

- Successive configs, ending with accepting config.
- Assume WLOG that each computation takes exactly p(|w|) steps, so we use p(|w|) + 1 rows.
- p(|w|) + 3 columns: p(|w|) for the interesting portion of the tape, one for head and state, two for endmarkers.

Proof that SAT is NP-hard

- ϕ_w is satisfiable iff there is an accepting CH of M on w.
- Entries in the matrix are represented by Boolean variables:
  - Define $C = Q \cup \Gamma \cup \{\#\}$, alphabet of possible matrix entries.
  - Variable $x_{i,j,c}$ represents “the entry in position (i, j) is c”.
- Define ϕ_w as a formula over these $x_{i,j,c}$ variables, satisfiable if and only if there is an accepting computation history for w (in matrix form).
- Moreover, an assignment of values to the $x_{i,j,c}$ variables that satisfies ϕ_w will correspond to an encoding of an accepting computation.
- Specifically, $\phi_w = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$, where:
  - ϕ_{cell} : There is exactly one value in each matrix location.
  - ϕ_{start} : The first row represents the starting configuration.
  - ϕ_{accept} : The last row is an accepting configuration.
  - ϕ_{move} : Successive rows represent allowable moves of M.

ϕ_{cell}

- For each position (i,j), write the conjunction of two formulas:

  $\bigvee_{c \in C} x_{i,j,c}$: Some value appears in position (i,j).

  $\bigwedge_{c, d \in C, c \neq d} (\neg x_{i,j,c} \vee \neg x_{i,j,d})$: Position (i,j) doesn't contain two values.

- ϕ_{cell}: Conjoin formulas for all positions (i,j).

- Easy to construct the entire formula ϕ_{cell} given w input.
- Construct it in polynomial time.
- Sanity check: Length of formula is polynomial in |w|:
  - $O((p(|w|)^2)$ subformulas, one for each (i,j).
  - Length of each subformula depends on C, $O(|C|^2)$.

ϕ_{start}

- The right symbols appear in the first row:

  $\#\quad q_0\quad w_1\quad w_2\quad w_3\qquad w_n\quad --\quad --\qquad --\quad \#$

ϕ_{start}: $x_{1,1,\#} \wedge x_{1,2,q0} \wedge x_{1,3,w1} \wedge x_{1,4,w2} \wedge$

$\wedge\ x_{1,n+2,wn} \wedge x_{1,n+3,--} \wedge$

$\wedge\ x_{1,p(n)+2,--} \wedge x_{1,p(n)+3,\#}$

ϕ_{accept}

- For each j, $2 \leq j \leq p(|w|) + 2$, write the formula:

$$x_{p(|w|)+1,j,qacc}$$

- q_{acc} appears in position j of the last row.
- ϕ_{accept}: Take disjunction (or) of all formulas for all j.
- That is, q_{acc} appears in some position of the last row.

ϕ_{move}

- As for PCP, correct moves depend on correct changes to local portions of configurations.
- It's enough to consider 2 $\times$ 3 rectangles:
- If every 2 $\times$ 3 rectangle is "good", i.e., consistent with the transitions, then the entire matrix represents an accepting CH.
- For each position (i,j), $1 \le i \le p(|w|)$, $1 \le j \le p(|w|)+1$, write a formula saying that the rectangle with upper left at (i,j) is "good".
- Then conjoin all of these, $O(p(|w|)^2)$ clauses.
- Good tiles for (i,j), for a, b, c in Γ:

a	b	c
a	b	c

#	a	b
#	a	b

a	b	#
a	b	#

ϕ_{move}

- Other good tiles are defined in terms of the nondeterministic transition function δ.
- E.g., if $\delta(q_1, a)$ includes tuple (q_2, b, L), then the following are good:
  - Represents the move directly; for any c:
  - Head moves left out of the rectangle; for any c, d:
  - Head is just to the left of the rectangle; for any c, d:
  - Head at right; for any c, d, e:
  - And more, for #, etc.
- Analogously if $\delta(q_1, a)$ includes (q_2, b, R).
- Since M is nondeterministic, $\delta(q_1, a)$ may contain several moves, so include all the tiles.

c	q_1	a
q_2	c	b

q_1	a	c
d	b	c

a	c	d
b	c	d

d	c	q_1
d	q_2	c

e	d	c
e	d	q_2

ϕ_{move}

- The good tiles give partial constraints on the computation.
- When taken together, they give enough constraints so that only a correct CH can satisfy them all.
- The part (conjunct) of ϕ_{move} for (i,j) should say that the rectangle with upper left at (i,j) is good:
- It is simply the disjunction (or), over all allowable tiles, of the subformula:

a1	a2	a3
b1	b2	b3

$$x_{i,j,a1} \wedge x_{i,j+1,a2} \wedge x_{i,j+2,a3} \wedge x_{i+1,j,b1} \wedge x_{i+1,j+1,b2} \wedge x_{i+1,j+2,b3}$$

- Thus, ϕ_{move} is the conjunction over all (i,j), of the disjunction over all good tiles, of the formula just above.

ϕ_{move}

- ϕ_{move} is the conjunction over all (i,j), of the disjunction over all good tiles, of the given six-term conjunctive formula.
- Q: How big is the formula ϕ_{move}?
- $O(p(|w|)^2)$ clauses, one for each (i,j) pair.
- Each clause is only constant length, O(1).
  - Because machine M yields only a constant number of good tiles.
  - And there are only 6 terms for each tile.
- Thus, length of ϕ_{move} is polynomial in |w|.
- $\phi_w = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$, length also poly in |w|.

ϕ_{move}

- $\phi_w = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$, length poly in |w|.
- More importantly, can produce ϕ_w from w in time that is polynomial in |w|.
- w $\in$ A if and only if M has an accepting CH for w if and only if ϕ_w is satisfiable.
- Thus, A $\leq_p$ SAT.
- Since A was any language in NP, this proves that SAT is NP-hard.
- Since SAT is in NP and is NP-hard, SAT is NP-complete.

Reducing search and optimizatian problems to the corresponding decision problem

We earlier made a claim that we can polynomial time reduce an NP-hard search or optimization problemn to the corresponding decision problem. This is usually easy to do for any specific problem.
Note: Now we are considering the general form of polynomial time reductions and not the more restricted polyynomial time transformation.

Let's consider the following search problem: Given a CNF formula F, find a satisfying assignment if F is satisfiable, and otherwise say that F is not satisfiable.

So let SAT be a subroutine that solves the decision problem for CNF formulas (i.e., determnies whether or not a given CNF formula F) is satisfiable. After one call to the subroutine we know whether or not the formula is satisfiable. So we might as well assume that F is satisfiable.

Finishing the reduction of the SAT search problem to the SAT decision problem

The idea now is to create a satsifying assignment τ one vartiable at a time. Say the propositional variables are $x_1, x_2, \ldots, x_n$.

Set $\tau(x_1) = \textit{true}$ and consider the formual $F' = F|_{x_1=\textit{true}}$. Call *SAT* to decide whether or not F' is satisfiable. If so, then set $\tau(x_1) = \textit{true}$; otherwise, set $\tau(x_1) = \textit{false}$. In either case, we go on to consider $F' = F|_{x_1=\tau(x_1)}$ and as we did for x_1, we can now determine a truth value $\tau(x_2)$ for x_2.

We continue to do this one variable at a time and thereby compute a satisfying assignment τ.

Note that there could be many satisfyling assignments τ and we have just found one such satisfying τ.

Reducing MAXSAT to the corresponding decision problem

In the MAXSAT problem, we are given a CNF formula $F = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ and the objective is to find a truth assignment τ that will maximize the number of clauses that are satisfied.

In the corresponding decision problem $MSAT_k$, given (F, k), we need to decide if F has a truth assignment satifying k clauses. **Note:** This is a different problem that kSAT where we restrict clauses to have at most k literals.

Note that SAT is a special case of $MSAT_k$. Why? It follows that $MSAT_k$ is NP-hard and it is easy to see that $MSAT_k$ is in NP so that the problem is NP-complete. We can then test $MSAT_k$ for each possible k ($k \leq m$, the number of clauses) to find the optimum value k using $MSAT_k$ as a subroutine. Then as in the search problem we can construct a truth assignment that acheieves this value, by setting the propositional values, one at a time.

Another reduction of an optimization problem to the corresponding decision problem

Consider the Weighted Maximum Independent Set (WMIS) optimization problem. Namely, given $G = (V, E, w)$, with $w : V \to \mathbb{N}$, find an independent set $V' \subseteq V$ so as to maximize $\sum_{v \in V'} w(v)$. The weights are non-negative integers, represented in binary so lets say that the largest integer is 2^ℓ and hence the length of the encoding will be $O(|E| + |V| \cdot \ell)$.

To compute an optimal solution, we need to use the IS decision procedure to find the optimum value $k \leq 2^\ell$ such that G has an independent set with total weight k. We cannot test every $k \leq 2^\ell$ (if ℓ is large). So we test using binary search to locate the optimum k. Now we need to find an optimum solution of weight k.

We find a solution one vertex at a time. Namely, we can ask if we select v_1 to be in the IS, will the graph $G' = G$ with v_1 (and all its adjacent edges) removed have an IS of weight $k - w(v_1)$. If yes then we add v_1 to the IS and remove it. Otherwise we just remove v_1. We continue this way, one vertex at a time, and compute an optimal solution.

kSAT and exact *kSAT*

As mentioned we sometimes use *kSAT* to mean at most k literals per clause and sometimes to mean exactly k literals per clause. Clearly in either interpretation, *kSAT* is in *NP*. If we interpret *kSAT* as at most k literals per clause then again 3*SAT* is a special case of *kSAT* for all $k \geq 4$.

What about exact *kSAT*. How do we show that it is *NP* hard? Here is the simple idea. Lets just do exact 3*SAT* Consider a clause $C = (x \vee y)$ that has just 2 literals. We transform that clause to $(x \vee y \vee z) \wedge (x \vee y \vee \bar{z})$. For a unit clause $C = (x)$ we can first transform to $(x \vee y) \wedge (x \vee \bar{y}$.

Clay Math Institute Millenium Problems: $1,000,000 each

1. Birch and Swinnerton-Dyer Conjecture
2. Hodge Conjecture
3. Navier-Stokes Equations
4. $P = NP$?
5. Poincaré Conjecture **(Solved)**[1]
6. Riemann Hypothesis
7. Yang-Mills Theory

[1] Solved by Grigori Perelman 2003: Prize unclaimed

How important is the P vs NP question

Lance Fortnow has an article on P and NP in the September 2009 Communications of the ACM, in which he says

> *"The P versus NP problem has gone from an interesting problem related to logic to perhaps the most fundamental and important mathematical question of our time, whose importance only grows as computers become more powerful and widespread."*

Claim: It is worth well over the $1,000,000

Other long standing and fundamental open problems in complexity theory

In the following disucssions about closure unbder complement, we are considering $\leq_p$ to mean tramnsformation and not the general reduction.

For any language L (i.e., decision problem), we define $\bar{L} = \{x : x \notin L\}$. Similarly, for any class $\mathcal{C}$ of languages the class $co - \mathcal{C} = \{\bar{L} : L \in \mathcal{C}\}$. An open problem related to the P vs NP issue is the whether or not $NP = co - NP$.

Conjecture: $\overline{SAT} \notin NP$

We have the following fact: $NP = co - NP$ iff $\bar{L} \in NP$ for some (any) NP-complete language L.

As mentioned before It is widely believed that $NP \neq co - NP$
How would you verify that a CNF formula F is *not* satisfiable?

Why we believe integer factoring is not NP-hard

As mentioned, some cryptographic schemes (i.e. for decoding a cryptographically encoded message) rely on the assumption that factoring can *not* be done efficiently (even in some average case sense).

However, it is believed that factoring is not NP-hard. To make this comment precise, consider the following decision problem which can be used to factor an integer: $FACTOR = \{(x, m) | x \text{ has a proper factor} \leq m\}$. It should be clear that $FACTOR$ is in NP.

Claim: $co - FACTOR = \{(x, m) | x \notin FACTOR\}$ is in NP. Hence we do not believe $FACTOR$ can be NP complete.

To see this, we first note that that $PRIME = \{x | x \text{ is a prime number}\}$ is now known to be in P. (It is sufficient to know that $PRIME$ is in NP which was known since the early 1970's.) Then a certificate y for $(x.m)$ being in $co - FACTOR$ is the prime factorization $y = (p_1, e_1, p_2, e_2, \ldots p_r, e_r)$ of x which can be verified by checking that each p_i is a prime and that $x = (p_1^{e_1} \cdot p_2^{e_2} \ldots \cdot p_r^{e_r})$

More complexity theory issues

After sequential time, the most studied complexity measure is *space*. If one maintains the input so that it is "read-only", it is meaningful to consisder sublinear (e.g. $\log n$) space bounds.

Analogous to deteriministic (resp. non-deterministic) time bounded classes $DTIME(T)$ (resp. $NTIME(T)$ we can define space bounded classes.

n contrast to what is widely believed about sequential time bounded classes, we have the following two Theorems for "reasonable" space bounds $S(n) \geq \log n$:

- $NSPACE(S) \subseteq DSPACE(S^2)$
- $NSPACE(S) = co - NSPACE(S)$

It is an open problem as to whether or not $NSPACE(S) = DSPACE(S)$.

It is also clear (for the models of computation we consider) that both $DSPACE(S)$ and $NSPACE(S)$ are contained in $\cup_c DTIME(c^{S(n)})$. Why? Hence $DSPACE(\log n) \subseteq P$.

Conjecture: $DSPACE \neq P$

And more complexity issues

Later in the course we will devote a lecture to randomized algorithms. For a number of computational problems, the use of randmization will provide the best known time bounds. And in some computational settings (e,g, cryptography, sublinear time algorithms), randomization is **necessary**.

With regard to complexity theory, analgous to the class P of polynomial time decisions, we have three different classes of decisions problems solvable in *randomized polynomial time*. These classes are ZPP (expected polynomial time, no error), RP (polynomial time, one sided error) and BPP (polynomial time, two sided error).

While it is clear that $RP \cap co - RP = ZPP \subseteq RP \subseteq BPP$, it is not known if any of these inclusions are proper.

Moreover, it is not known if $RP = P$ or $BPP = P$. Lately, some prominent complexity theorists conjecture tht $BPP = P$ in which case, ignoring polynomial factors, randomization can be avoided.

One final complexity comment

Although the P vs NP issue is not resolved, there are “natural” problems which *provably* require (say) exponential time. (In fact, there are some problems which require “much more” than exponential time.

Without defining things carefully, one such problem is the decision problem for the first order theory of the reals. That is, we want to decide when a fully quantified first order arithmetic formual is true or false when the variables are real numbers.

For example, the statement $\forall x > 0, \exists y[y^2 = x]$ is true for real numbers but false for say the integers or rational numbers. Similarly, $\exists y[y^2 = -1]$ is true for the complex numbers but false for the real numbers.
While it is known that this decision problem is computable (in time $\approx 2^{2^n}$) where n is the length of the 1st order statment, it is also proven that the decision problem cannot be done in time say 2^n.

Integer Programming and Linear Programming

Our next major topic is Linear Programming (LP) . One could easily teach an entire course on LP theory and still there would be much more to discuss. Our interest will be mainly in using LP relaxations of Integer Programs (IP) to obtain approximation algorithms. But still, it is important to know a little about LP theory.

We first present Integer Programming. As an example, consider the weighted Vertex Cover problem. Recall that in the weighted VC problem, the input is $G = (V, E, w)$ with $w : V \to \mathbb{R}$ and say $V = \{v_1, v_2, \ldots, v_n\}$. We can formulate the weighted VC problem as an IP (in fact, as $\{0, 1\}$-IP) as follows:

Minimize $\sum_i w_i x_i$
Subject to $x_i + x_i \geq 1$ for all $(v_i, v_j) \in E$.
$x_i \in \{0, 1\}$. for all i.

The interpretation of the IP variables is that $x_i = 1$ iff v_i is included in the vertex cover. Clearly the constraints force any setting of the IP variables $(x_1, x_2, \ldots, x_n)$ to be a valid vertex cover and the objective is precisely what we are trying to optimize.

More generally, an IP (for a minimization problem) aims to minimize a linear objective function subject to a set of linear and integral constraints. That is:

Minimize $\sum_i c_i x_i$ Subject to m linear constraints of the form:
$\sum_i a_{j,i} x_i \; R \; b_j$ where R is $\leq, \geq, or =$ for $j = 1, 2, \ldots, m$.
and integral constraints $x_i \in \mathbb{Z}$.

It turns out that one can always convert an IP to a *standard form* where all the inequalities are $\geq$ and all the variables are in $\mathbb{N}$. For a maximization problem we want to maximize a linear function subject to a set of linear constaints and in standard form all the linear constraints are $\leq$.

In a $\{0, 1\}$ IP, the integrality constraints are replaced by $x_i \in \{0, 1\}$.

The $\{0,1\}$ IP decision problem is *NP* complete

The IP decision problem is to determine if a set of linear constraints can be satisfied by an integral solution (or a $\{0,1\}$ solution for a $\{0,1\}$ IP). Note that we can absorb the objective function into the linear constraints so as to determine if there is a solution at most some value.

Theorem

The $\{0,1\}$ IP decision problem is NP complete.

This should not be a surprise since we already showed how to represent the Vertex Cover optimization problem as an IP.

The usual proof transforms a CNF formula into an IP where each clause is represented by a linear inequality. For every propositonal variable y_i we have an $\{0,1\}$ IP variable x_i where we interpret $x_i = 1$ as $y_i = \mathit{true}$ and $x_i = 0$ as $y_i = \mathit{false}$.

For example, the clause $(y_1 \vee \bar{y}_2 \vee y_3)$ would be represented by the inequality $x_1 + (1 - x_2) + x_3 \geq 1$ which forces the clause to be true for a particular settng of the variables.

The decision problem is *NP* complete

Since the $\{0,1\}$ IP problem is a special case of the IP decision problem, the IP problem is *NP* hard. But here is one of the rare cases that I mentioned where it is not obvious why the IP decision problem is in *NP*. The issue here is that the IP may have a solution BUT what if it is too big and therefore cannot be a certificate.

However, using some linear algebra, it can be shown that if the IP has a solution then it has a solution whose length is polynomial in the length of the IP formulation.

Aside: Usually we assume that the coefficients in an IP formulation are either integral or rationals so that the problem can be finitely represented.

IP vs LP

What then is LP? LP is just like IP except now we allow rational solutions. Assuming the coefficients are integral or rational, if there is a solution then there is a rational solution. Moreover, if there is a solution to the LP then there is a solution whose variable values have lenghts bounded by a polynomial is the length of the LP formulation and hence LP (like IP) is in *NP*.

Is LP easier or harder than IP?

IP vs LP

What then is LP? LP is just like IP except now we allow rational solutions. Assuming the coefficients are integral or rational, if there is a solution then there is a rational solution. Moreover, if there is a solution to the LP then there is a solution whose variable values have lenghts bounded by a polynomial is the length of the LP formulation and hence LP (like IP) is in *NP*.

Is LP easier or harder than IP?

It turns out that any LP can be solved in polynomial time (theoretically and in pratcice efficiently). We probably won't give a polynomial time algorithm but we will discuss the history and complexity of solving LP problems. As I said, we will mainly be interested in using LP as a big "big hammer" using the terminology of the DPV text.