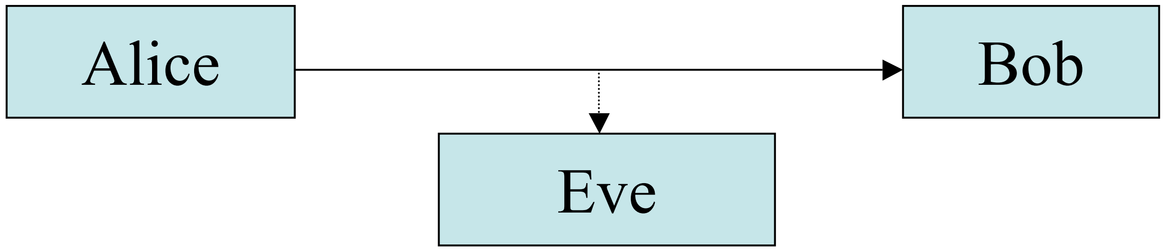
One Way Functions and Sparse Sets.

CS254
Chris Pollett
Nov. 27, 2006.

Outline

- Cryptography
- UP
- UP and One-way functions
- Sparse, Unary Languages, and the P=NP problem

Cryptography



- Alice and Bob want to communicate and have agreed on an encryption and decryption algorithms E and D.
- E takes a key string e and and a string x and outputs some encrypted text y. E(e,x) = y. **Here y is at most polynomially longer than x**.
- D takes a decryption key string d (often d=e, but in public key cryptography they are different) and acts so that D(d, y) = x.
- We want it to be difficult for an Eavesdropper Eve to be able to determine x if she knows the algorithms E and D as well as the encryption key e and the string y.
- In public key cryptography (PKC), e is well known and called the **public key** of Bob. Anyone wanting to send a secret message to Bob can use this key.
- On the other hand, only Bob might know d. d is called the **private key**.
- Since y is at most polynomially longer than x, there is an FNP algorithm to guess it. So PKC is possible only if FNP ≠ FP. I.e., P≠NP.

One way functions

- The functions in FNP-FP that are needed for cryptography are called **one-way functions**.

Defn. Let f be a map from strings to strings. We say that f is a **one-way function** if the following hold of f:

(1) it is one-to-one, and for all strings x, $|x|^{1/k} \leq |f(x)| \leq |x|^k$.

(2) f is in FP.

(3) f^{-1}, the inverse of f, is not in FP.

- For example, given $p < q$ primes, the function $f(p,q) = p*q$ is suspected to be one way.
- Public Key systems like RSA require this function to be one way to be secure.

UP

- We are now going to connect the existence of one-way functions to a complexity.
- It would be nice if one could build one-way functions based on NP-complete sets.
- This has been tried (Merkle-Hellmann). Nowadays, though, we have some evidence this approach is unlikely to work.

Defn. Call an NTM **unambiguous** if for any input x there is at most one accepting computation. Let **UP** be the class of languages accepted by unambiguous p-time NTMs.

- So $P \subseteq UP \subseteq NP$. We expect $P \neq UP$.

Thm. P=UP iff there are no one-way functions.

- As a promise class, UP seems unlikely to have complete problems, so is unlikely to equal NP.

Proof of Theorem

Suppose f is 1-way. Define the language:

$L_f = \{(x,y) \mid$ there is a z such that $f(z)=y$ and $z \leq x\}$.

Less equal of strings above is with respect to lex ordering. We claim that L_f is in UP-P. It is in UP since given (x,y) we can always guess a string z and check if it works. As 1-way implies f is 1-1, at most one string z will work. Suppose L_f were in P. Then we can invert f by binary search. Given y we'd like to invert, we first determine the length of x such that f(x)=y by asking queries of our L_f algorithm of the form: "Is $(1^{|y|\text{^}k}, y)$ in L_f ?", "Is $(1^{|y|\text{^}\{k-1\}}, y)$ in L_f ?", .., "Is $(1^{|y|\text{^}\{1/k\}}, y)$ in L_f ?". Until we get our first negative answer, this will be the first length which is lex smaller than x, and so will give us the length l of x. Using l, we can then make the query: "Is $(01^{l-1},y)$ in L_f?". If not, we know first digit of x is 1. If yes, we make the query "Is $(001^{l-2},y)$ in L_f?" and continue in the same way. The next query after that would be "Is $(101^{l-2},y)$ in L_f?" This would invert f in p-time contradicting f being 1-way. Therefore, L_f is not in P.

For the other direction…

Proof cont´d

Suppose L is in UP - P. Let U be the unambiguous NTM for L. If x is the accepting computation of U on input y, define $f_U(x) = 1y$; otherwise, if x is not an accepting computation of U, then define $f_U(x) = 0x$. As one can verify if a string is a legal computation of an NTM in p-time, f_U is p-time. The lengths of the inputs and outputs of the above function are polynomially related. It is also 1-1. Finally, if we could invert f_U in deterministic p-time we could tell if y was in L in deterministic p-time, contradicting L in UP-P.

Density

- The UP languages are one class of languages in NP which are unlikely to all be in P.
- We now investigate another class of languages which are unlikely to be hard for NP: the sparse languages.

Defn Let L be a language.The density of L is the function $dens_L(n)=|\{x \text{ in } L \mid |x|\leq n\}|$. A language is said to be **sparse** if there is a polynomial p(n) such that $dens_L(n) \leq p(n)$ for all n.

- For example, a **unary language** L is a language which is a subset of $\{0\}^*$. So $dens_L(n) \leq n$, so such an L is sparse.
- Mahaney has shown that there are no sparse NP-complete sets unless P=NP. Today, we will show a weaker result…

Unary languages and NP-completeness

Thm. Suppose that a unary language U is NP-complete. Then P=NP.

Proof. Let U be an NP-complete unary language. Let R reduce SAT to U. We can assume the output of R is always in $\{0\}^*$, as if the output is not a unary string we immediately know it is not in U.

Given a formula F in $x_1,..,x_n$. Our algorithm for SAT considers partial truth assignments to the first j variables. We represent such assignments as strings t in $\{0,1\}^j$. Let F[t] be the formula resulting from substituting the values of the first j variables in F according to t. (cont´d next page)

Proof cont´d

In our computation, we will make use of a hash table which associates strings t with values v of the formula under than assignment. I.e., The table has entries (H(t),v), where H is a hash function to be specified. Our algorithm for SAT is:

Initialize t = empty string. Call SAT-COMPUTE(F,t) where

SAT-COMPUTE(F,t):

If |t| = n then return "yes" if F[t] has no clauses, else return "no".

Otherwise, look up H(t) in the table if there is an entry (H(t),v) return v.

Otherwise, compute SAT-COMPUTE(F,t0) or SAT-COMPUTE(F, t1). Based on this update the table with (H(t),v) and return v where v is "yes" if either of the above said "yes" and is no otherwise.

That is the algorithm. Now need to show that is a choice of H which makes it p-time…

Proof cont´d some more

- We want an H so that:
    1. If H(t)=H(t´) then either F[t] and F[t´] are both satisfiable or both not.
    2. We want the range of H to be small so that it can be searched efficiently and many values succeed.
- Let H(t) = R(F[t]).
- Notice if H(t) = H(t´) then R(F[t]) = R(F([t´]) and this in turn means they are either both in U or not -- and hence, both satisfiable or not. Thus, (1) hold of this H.
- All length of all values of H(t) can be bounded by p(n), so (2) will also hold.
- To see this let's estimate the run-time of algorithm…

Even more proof

- The time to look up a value in the table is O(p(n)), so the runtime is O(M*p(n)) where M is the number of invocations of the algorithm.
- The invocations form a binary tree of depth at most n.

Claim There is a set $T=\{t_1,t_2..\}$ of invocations of the algorithm such that (a) $|T| \geq M/2n$, (b) all invocations in T are recursive (not leaves), (c) none of the elements in T is a prefix of the other.

Proof. First we delete the leaves from the tree, leaving M/2 nodes. Select any bottom not yet deleted node and add it to T. Delete it and all its parents from the tree. Notice the ancestors are all prefixes so couldn't be in T. Repeat the above on the remaining tree. Notice at each step we delete at most n nodes. Hence, T will have size at least M/2n. Notice if $t_i \neq t_j$ then $H(t_i) \neq H(t_j)$ since if they did, then the one that occurred second would have been able to look up the value in the hash table and thus not have been recursive.

- We have shown there are M/2n different values in the table.
- But we also know the table has size at most p(n).
- Hence, $M \leq 2np(n)$ so the whole algorithm is polynomial.