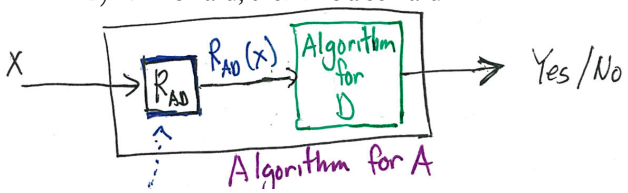
Oliver Song | 6.046 Final Exam Crib-sheet

NP Shit

Decision problems - Does an object of a certain quality exist?
Search problems - Find an object of a certain quality.
Optimization problems - Find an object of the best quality.
Not necessarily Decision < Search < Optimization- if decision is hard, search and optimization are too.
P - set of all problems solvable in polynomial time
NP - Verifiable, but not solvable in polynomial time
NP-Hard - Can be used to solve any problem in NP (all NP problems are reducible to NP-Hard problems)
NP-Complete - in NP and NP-Hard
Cook Reductions:
Useful when we have algorithm A in P and we want to show B in P
Karp Reductions:
Useful when A is NP-Complete and want to show B is NP-Complete
A is Karp-reducible to B if there exists a polytime algo such that

1) input x for problem A and R(x) is input for problem B
2) x is yes-input for A and R(x) is yes input for B

Uses: 1) If there is an algo that solves B, there is for A as well
2) If A is hard, then B is also hard



R_{AD} is Karp-reduction that transforms A inputs to D inputs in Ptime
To Show D is NP-complete:

1) Show D is in NP (prove there exists a polynomial time verifier for the problem)
2) Reduce some NP-complete problem to D
    1) Give polynomial time algorithm R such that if input x is an input for problem A, then R(x) is an input for problem D
    2) Show that if x ∈ A then R(x) ∈ B
    3) Show that if R(x) ∈ B then x ∈ A
3) The reduction must be polytime and map "yes" instances to "yes" instances and "no" instances to "no" instances

Good NP problems to know
3-SAT, or Boolean Satisfiability-
Problem of determining if variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. Conjunctive Normal Form: ORs ANDed. 3CNF = groups of 3.
Red-Blue Subsets
If you have a set of n elements, and a set of arbitrary subsets, can you color the set of elements such that every subset has at least one red and blue element?
Proof that Red-Blue Subsets is NP-complete
Use a reduction from 3-SAT. First, prove that RBS is in NP by giving polynomial time Verifier V(x, y). Need to check constraints: 1) R and B cover S 2) R and B are disjoint 3) Each subset has 1 red and 1 blue
Then, build 3-SAT set that is similar to the RBS situation, in which there are CNF statements that correspond to subsets.
Then, prove "there is a solution to the 3-SAT problem if and only if there is a solution to the RBS problem, by going from both sides; solution to 3-SAT if RBS, and solution to RBS if 3-SAT.
Therefore, Red-Blue Subsets is in NP-Hard, because it is at least as difficult as 3-SAT. Since it is NP and NP-Hard, it is NP-Complete.

Linear Programming

SIMPLEX(A, b, c)

```
1  (N, B, A, b, c, v) = INITIALIZE-SIMPLEX(A, b, c)
2  let Δ be a new vector of length n
3  while some index j ∈ N has c_j > 0
4      choose an index e ∈ N for which c_e > 0
5      for each index i ∈ B
6          if a_ie > 0
7              Δ_i = b_i / a_ie
8          else Δ_i = ∞
9      choose an index l ∈ B that minimizes Δ_i
10     if Δ_l == ∞
11         return "unbounded"
12     else (N, B, A, b, c, v) = PIVOT(N, B, A, b, c, v, l, e)
13 for i = 1 to n
14     if i ∈ B
15         x̄_i = b_i
16     else x̄_i = 0
17 return (x̄_1, x̄_2, ..., x̄_n)
```

How to take the Dual

1) Rewrite the objective as minimization (maximize the negative)
2) Rewrite each inequality constraint as a less than or equal and rearrange each constraint so that the right hand side is 0
3) Define a non-negative dual variable for each inequality constraint, and an unrestricted dual variable for each equality constraint.
    $\lambda_1 \geq 0$ for inequality, λ_2 for equality
4) For each constraint, eliminate the constraint and add the term (dual variable)*(left hand side of constraint) to the objective. Maximize the result over the dual variables.
5) Rewrite the objective so that it consists of several terms of the form (primal variable)*(expression with dual variables), plus remaining terms involving only dual variables
6) Remove each term of the form (primal variable)*(expression with dual variables) and replace with a constraint of the form:
    1) Expression ≥ 0 if the primal is non-negative
    2) Expression ≤ 0 if the primal is negative
    3) Expression = 0 if the primal is unrestricted
7) If the linear program in step 1 was rewritten as minimization, rewrite as minimization.

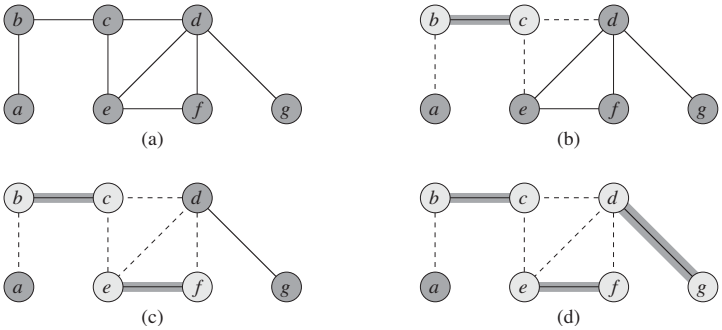
Hashing

Universal Hashing- a single hash function can get fucked if someone purposely chooses all keys that hash to the same slot. Solution- have a family of hash functions and randomly choose one of them to hash any given key. That way you or the attack have no idea what will hash to what, but since it's perfectly random you can guarantee that most slots will be hit evenly. It's called **universal** if for each pair of distinct keys k, l in U, the number of hash functions h in H for which h(k) = h(l) is at most $|H|/m$. m = number of slots.
Perfect Hashing- Hashing is perfect if O(1) memory accesses are required to perform a search in the worst case. We use two levels of hashing, with universal hashing at each level.

Approximation Algorithms

A lot of problems are NP-complete but we can approximate some shit and still get reasonable answers in reasonable amounts of time. We say that an algorithm for a problem has an approximation ratio of p(n) if for any input size n, the cost C of the solution produced by the algorithm is within a factor of p(n) of the cost C* of an optimal solution: $\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$

General question for appx algorithms will be: prove X algorithm is a Y-approximation. For example, APPROX-VERTEX-COVER:



Is a 2-approximation of VERTEX-COVER (NP-Complete) and runs in polynomial time. Whenever you figure out a Y-approximation algorithm, the proof is generally pretty non-intuitive.

Uncertainty

Amortized Analysis

1) Aggregate Analysis

Fucking count everything. Add up everything and divide by the number of shits you counted. All operations have same "cost".

2) Accounting method

This is where we "pay it forward" with certain operations to "cash them in" later for different operations. Logically leads to.....

3) Potential method

This is where the **Φ** shit comes in.

Competitive Analysis

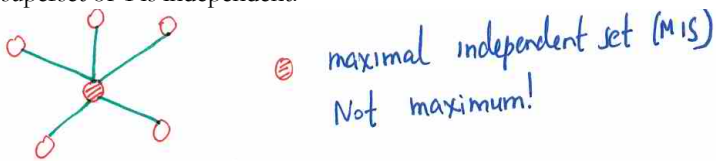
The idea is that for an online algorithm, you don't know what data you're working with. So it is possible that the person sending data can sabotage your algorithm by sending the worst possible inputs. So, given some arbitrarily bad input giver, how does your algorithm perform? So you end up with a ratio of online to offline. For the ski-rental example, we make an algorithm that chooses optimally until T days, then pays 2T, making it 2-competitive (worst case 2x cost).

Distributed Algorithms

General idea is that there are nodes that can know their neighbors and can communicate with their neighbors by sending messages. Using this fact, the nodes have to come to some decision or make some computation. Examples:

Leader election- send your Unique ID to your neighbors and when you receive one, compare it to yours and send out the bigger one. Eventually, only one will remain - O(n). Advanced: Start with local leaders and round robin eliminate. O(log n).

Max independent set- Independent set I is maximal if no strict superset of I is independent.



Simplified Luby's Algorithm

Initialize: All processes added to "LIVE" set. MIS = ∅

1) Each live node/process "marks" itself with probability 1/2d
2) Each marked node v checks neighbors. If any marked, v unmarks itself. NOTE: Two adjacent marked nodes can unmark each other
3) Each remaining marked node adds itself to MIS (win!) and removes itself and all neighbors from LIVE set.
4) Terminate when LIVE = ∅.

P(# of rounds > 8d*ln n) = at most 1/n

Expected # of rounds O(d * ln n)

Cryptography (yeah!)

Diffie-Hellman Key Exchange

Publics are g and p where g is 2<=g<=p-2

Alice selects a, computes g^a and sends to Bob, who selects b (1<= a, b <= p-2) computes g^b and sends g^b to Alice.

Alice can compute $(g^b)^a \bmod p = K$

Bob can compute $(g^a)^b \bmod p = K$

Can be thwarted by man-in-the-middle attack

RSA (uses hard prime factorability)

Alice picks two large secret primes p and q, computes N = p*q

Chooses encryption exponent e which satisfies gcd(e, (p-1)(q-1))=1

Alice's public key is then (N, e)

and the decryption exponent is obtained using Extended Euclidean:

$e*d \equiv 1 \bmod (p-1)(q-1)$

Alice's private key is then (d, p, q)

Encryption: $c = m^e \bmod N$

Decryption: $m = c^d \bmod N$

Extended-Euclid(a, b)

```
1 if b == 0
2         return (a, 1, 0)
3 else (d', x', y') = Extended-Euclid(b,a mod b)
4         (d, x, y) = ( d', y', x'- FLOOR(a/b)*y' )
5         return (d, x, y)
```

Then take the result x, plug in to a(x+b)+b(y-a)=1, where x+b = d.

Sub-linear Algorithms

The general idea is to make a nearly accurate approximation for what you're trying to find: sortedness of list/connectedness of graph. First you need to define "close" to sorted or connected, which involves defining some variable **ϵ** which is the acceptable error rate (1/10, maybe). For graphs, **ϵ**-close to connected if can add **ϵ**dn and transform to connected. For lists, **ϵ**-close to sorted if can remove **ϵ**n items to have a sorted list.

The general solution algorithm is a combination of:

1) Do it 1/**ϵ**, 1/**ϵ**d, 1/**ϵ**something times
2) Each step pick random element, do a search for or from that element (Binary Search for arrays, BFS for graphs)

Compression (yeah!)

Nothing special here, standard 6.02. You can't really remove bits; all you can do is *represent* them. Three types of compression mentioned:

Lossless: Run-length encoding, huffman-coding, lempel-ziv

Lossy: .jpg/.mpg/.mp3, wavelets, bloom filters

Run-length- based on how many in a row there are

Huffman- purely based on frequency of occurrence (build tree)

Lempel-ziv- use pointers to previous places where saw the substring

.jpg/.mpg...- store with fourier representation, and cut undetectable

Bloom filter- data struct for answering membership queries (+ save lots of space, - false positive)

Key Algorithms

van Emde Boas (vEB) - faster than quicksort- O(n lg lg n) running time. You can only run it if the input is wihtin a universe of integers size 1 through n.

In general, if you have an adversary choosing difficult inputs to an algorithm, introduce randomness. This will guarantee limited worst case behavior.