6.046: Design and Analysis of Algorithms

Rachel Wu

Spring 2017

These are my lecture notes from 6.046, Design and Analysis of Algorithms, at the Massachusetts Institute of Technology, taught this semester (Spring 2017) by Professors Debayan Gupta[1], Aleksander Madry[2], and Bruce Tidor[3].

I wrote these lecture notes in LaTeX in real time during lectures, so there may be errors and typos. I have lovingly pillaged Tony Zhang's[4] and Evan Chen's formatting commands and style. Should you encounter an error in the notes, wish to suggest improvements, or alert me to a failure on my part to keep the web notes updated, please contact me at rmwu@mit.edu.

This document was last modified 2017-05-16.

[1] debayan.edu
[2] madry.edu
[3] tidor@mit.edu
[4] txz@mit.edu

Contents

1 February 7, 2017 **1**
1.1 Administrivia . . . 1
1.2 Introduction . . . 1
1.3 Time complexity . . . 1
1.4 Interval scheduling . . . 2

2 February 9, 2017 **4**
2.1 Divide and conquer . . . 4

3 February 10, 2017 **6**
3.1 Proof writing . . . 6
3.2 Asymptotic notation . . . 6
3.3 Determining runtime . . . 7
3.3.1 Master theorem . . . 7
3.3.2 Guess and check (induction) . . . 7
3.3.3 Recursion tree . . . 7

4 February 14, 2017 **8**
4.1 Polynomials . . . 8
4.2 Fast Fourier transform . . . 9

5 February 16, 2017 **10**
5.1 Amortization . . . 10
5.2 Union-find . . . 10
5.2.1 Linked list representation . . . 10
5.2.2 Forest representation . . . 11

6 February 17, 2017 **13**
6.1 FFT review . . . 13

7 February 23, 2017 **14**
7.1 Competitive analysis . . . 14
7.1.1 Self-organizing lists . . . 14
7.2 How to learn in this class? . . . 16

8 February 24, 2017 **16**
8.1 Amortized analysis . . . 16
8.1.1 Aggregate analysis . . . 16
8.1.2 Accounting method . . . 17
8.1.3 Potential method . . . 17

9 February 28, 2017 **18**
9.1 Minimum spanning trees . . . 18
9.2 Kruskal's algorithm . . . 19
9.2.1 Implementation . . . 19

10 March 2, 2017 **20**
10.1 Administreview . . . 20
10.2 Maximum flow . . . 20
10.3 Minimum cut . . . 21

11 March 3, 2017 **22**
11.1 Minimum spanning trees (continued) . . . 22
11.2 Prim's algorithm . . . 23

12 March 7, 2017 **24**
12.1 Ford-Fulkerson algorithm . . . 24
12.2 Maximum bottleneck algorithm . . . 25
12.3 Edmonds-Karp algorithm . . . 25

13 March 9, 2017 **26**
13.1 Linear programming . . . 26
13.1.1 Geometric intuition . . . 27
13.2 LP algorithms . . . 27
13.2.1 Simplex algorithm . . . 27
13.2.2 Ellipsoid method . . . 27
13.2.3 Interior point method . . . 27
13.3 Duality . . . 27

14 March 10, 2017 (Review 1) **29**

15 March 11, 2017 (Review 2) **30**
15.1 Median finding . . . 30
15.2 FFT . . . 30
15.3 Union-find . . . 31

16 March 16, 2017 **32**
16.1 Game theory . . . 32
16.2 How to get rich . . . 33

17 March 17, 2017 **35**
17.1 Zero-sum games . . . 35
17.2 Learning from expert advice . . . 35

18 March 21, 2017 **37**
18.1 Randomized algorithms . . . 37
18.2 Monte Carlo algorithms . . . 37
18.3 Tail inequalities . . . 38

19 March 23, 2017 **39**
19.1 Random walks in graphs . . . 39

20 April 4, 2017 **41**
20.1 Universal Hashing . . . 41
20.1.1 Dot product hash family . . . 42
20.2 Perfect hashing . . . 42

21 April 6, 2017 **44**
21.1 Streaming . . . 44
21.2 Reservoir sampling . . . 44
21.3 Frequency moments . . . 45

22 April 7, 2017 **46**
22.1 Tail inequalities . . . 46
22.1.1 Markov's inequality . . . 46
22.1.2 Chebyshev's inequality . . . 46
22.1.3 Chernoff bound . . . 47
22.2 Streaming algorithms (graphs) . . . 47
22.2.1 Sparsifier . . . 48

23 April 11, 2017 **49**
23.1 Continuous optimization . . . 49

24 April 20, 2017 **49**
24.1 Gradient descent . . . 49
24.2 Linear regression . . . 50

25 April 21, 2017 **52**
25.1 Gradient descent . . . 52

26 April 27, 2017 **53**
26.1 Dynamic programming . . . 53
26.2 Variations on polynomial time . . . 54

27 May 2, 2017 **55**
27.1 Intractability . . . 55
27.2 P vs. NP . . . 55

28 May 4, 2017 **57**
28.1 Reductions 57

29 May 9, 2017 **60**
29.1 Approximation algorithms 60

30 May 11, 2017 **63**
30.1 Weakly NP-hard problems 63
30.2 Fixed-parameter tractability 63
30.3 Kernelization 64

31 May 16, 2017 **65**
31.1 Distributed algorithms 65
31.2 Leader election 66
31.3 Maximal independent set 67

1 February 7, 2017

1.1 Administrivia

- Lectures in 26-100, Mondays and Wednesdays at 11. Recitations on Fridays, based on registrar.
- Prerequisites: 6.006 and 6.042 (or proof experience).
- Two in-class quizzes (March 14, April 13), plus a final exam. 25% of grade, and final is worth 40%.
- There are 10 problem sets, with 10 grace days. Lowest two problem sets are half-weighted, to "reduce the variance in problem sets."
- Lectures are recorded, but attend class, or the professors will be sad.

1.2 Introduction

In 6.006, we learned about basic algorithms. This class is about the art and craft of algorithms. And if you really like the "art" side of this, take 6.854.

1.3 Time complexity

There are categories of time complexity, the simplest of which is linear time, an example of which is graph connectivity. The next is $O(n \log n)$ time, and then there's quadratic, polynomial time.

Definition 1.1. Problems in **Class P** are problems that can be solved in polynomial time.

Example 1.2

All-pairs-shortest-paths can be solved in polynomial time. However, not all problems fall into this class.

Definition 1.3. Problems in **Class NP** can be verified in polynomial time, but we are unsure if we can solve them in polynomial time.

Example 1.4

We can *check* if a graph has a Eulerian or Hamiltonian cycle in polynomial time (visiting every edge or vertex). We would like to know the relationship between P and NP, so the million dollar question is, P=NP?

Definition 1.5. If a problem is **NP-complete**, then it is as hard as any NP problem, and if that problem is in P, then P=NP.

Very similar-sounding problems can have very different complexities, so it's a fallacy to think "this problem sounds similar, it must be similar."

1.4 Interval scheduling

Imagine that you are the registrar, scheduling classes.

Problem 1.6 (Basic interval scheduling)
We have a list of requests r_i, for $1 \leq i \leq n$, with starting times a_i and ending times b_i. Two requests r_i and r_j are compatible if $a_i \geq b_j$ or $b_i \leq a_j$. The goal is to schedule the maximal compatible set.

Naive solution: check every single subset! But unfortunately, this takes exponential time, so on an exam you'd get no points.

> Come on, this is MIT. We want a braindead efficient algorithm, like when you're doing your pset at 1 or 2 am. —Madry

We begin with a greedy approach: schedule requests by selecting the most-profitable, remaining request at each step. What rule do we use for "profitability"?

1. Select the shortest request (lowest $b_i - a_i$).
2. Select request with fewest incompatibles.
3. Select the request that finishes earliest (lowest b_i).

Theorem 1.7
The greedy algorithm for interval scheduling with earliest finish time always returns the optimal answer.

Proof. Let $o(R)$ be the optimal solution, and $g(R)$ be the greedy solution. Let some r_i be the first request that differs in $o(r_i)$ and $g(r_i)$. Let r_i' denote r_i for the greedy solution. We claim that $a_i' > b_{i-1}$, else the requests differ at $i-1$. There are thus two cases.

Case 1. $b_i' < b_i$ The greedy solution would have chosen r_i.

Case 2. $b_i' \geq b_i$ The optimal solution is not optimal.

So the greedy and optimal solutions are equivalent. The runtime is $O(n \log n)$. Sort requests by lowest b_i in $O(n \log n)$ and take top m that are compatible in $O(n)$. □

Now imagine instead that we want to accommodate the most students, rather than scheduling the most classes.

Problem 1.8 (Weighted interval scheduling)
We have samples $r_i = (a_i, b_i, w_i)$, where we want to maximize $o(R) = \sum w_j$ for scheduled $r_j \in R$.

Solution. We will use dynamic programming. Our subproblem is to accept or reject r_i, given the optimal solution for the rest.

Case 1. $r_i \in R$ so $o(R) = w_i + o(R - \text{incompatibles})$

Case 2. $r_i \notin R$ so $o(R) = o(R - r_i)$

So

$$o(R) = \max \begin{cases} o(R) = w_i + o(R - \text{incompatibles}) \\ o(R - r_i) \end{cases}$$

Note that the set of incompatibles for r_i forms a prefix. Thus we have a subproblem for every suffix $r_j, r_{j+1}, \ldots, r_n$. It will take $O(n)$ time to find the right suffix, so this will take a total of $O(n^2)$ time. This time can be reduced to $O(n \log n)$ with binary search.

2 February 9, 2017

2.1 Divide and conquer

Lecture cancelled due to snow. Notes were taken from professor's recording.

We have an input of size n, which we divide into a pieces, recursively apply our algorithm, and combine the results. Let the runtime be $T(n) = aT(\frac{n}{a}) + f(n)$

Problem 2.1 (Median finding)
Given a set S of n numbers, we define $x \in S$, rank(x) as # of elements $\in S \leq x$. The upper median is rank$(x) = \lceil \frac{m}{2} \rceil$, and lower median is rank$(x) = \lfloor \frac{m}{2} \rfloor$. Given S, we want to find x subject to rank$(x) = i$.

Naive solution: we sort the list in $O(n \log n)$ and take the middle. This isn't bad, but can we do better? Yes. Algorithm designed in 1973 to do this.

1 Pick some $x \in S$.
2 Let $L = \{y \in S | y < x\}$, and $G = \{y \in S | y > x\}$. Then rank$(x) = |L| + 1$.
3 rank$(x) = \begin{cases} = i & \text{we are done} \\ > i & \text{want rank } i, \text{ recurse on } L \\ < i & \text{want rank } i - \text{rank}(x), \text{ recurse on } G \end{cases}$

How do we cleverly choose x? In the worst case, we always choose the minimum x, but then the runtime becomes $O(n^2)$. We would like to choose an x with middle-ish rank, so L and G and about the same size. x needs to be c-balanced, so $|L|$ and $|G|$ are within cn, for $c < 1$.

$$T(n) = T(cn) + O(n) \tag{2.1}$$

Since $c < 1$, the size decreases exponentially, so the algorithm runs in linear time.

1 Partition S into $\frac{n}{5}$ groups of size 5 each ($O(n)$).
2 Find a median of each group ($O(n)$).
3 Take x to be the median of medians, using the same algorithm.

Key observation: x is $\frac{3}{4}$-balanced. x is in the middle of the medians, so it is also smaller than all the "larger-than-larger-median" numbers and larger than all the "smaller-than-small-median" numbers.

> "At this point, I'd ask, 'are there any questions,' but well, are there any questions? I thought so."—madry (this is a video recording due to snow).

We didn't take into account the time to find x, which brings us to

$$T(n) = T(3n/4) + O(n) + T(n/5) \tag{2.2}$$

Proof left as an exercise to the reader that this is indeed $O(n)$.

Problem 2.2 (Integer multiplication)
Given two numbers a and b of n bits, compute ab.

Grade school multiplication takes $O(n^2)$. Can we do better?

Let $a = x \cdot 2^{n/2} + y$, so x and y are the left and right parts. $b = w \cdot 2^{w/2} + z$. So this gives us

$$ab = (wx) \cdot w^n + (xz + yw) \cdot w^{m/2} + yz \tag{2.3}$$

Unfortunately, this is still $O(n^2)$.

But we can use Karatsuba's algorithm. First compute xw and yz. Do not multiple out xz or yw. Instead, compute $(x + y)(z + w)$.

Key idea: $(x + y)(z + w) = (xz + yw) + xw + yz$

So now, runtime is

$$T(n) = 3T(\frac{n}{2}) + O(n). \tag{2.4}$$

By Master Theorem, $T(n) = \Theta(n^{\log_2 3})$. By repeatedly applying this, Schönage & Strassen gives $O(n \log n \log \log n)$, so we are approaching "linear" time.

3 February 10, 2017

3.1 Proof writing

Hi Devin.[5] Call him "young $." This class loves proofs. Proofs should be clear, concise, and rigorous.

Lemma 3.1
Use lemmas for clarity.

Here is a sample proof of merge sort (algorithm given below).

1. If a has length 1, return.
2. Split a in half into l and r.
3. Recursively both l and r.
4. Merge l and r by comparing their first element, and appending the smaller to the output.
5. Return when l and r are empty.

Proof. We prove its correctness by strong induction.

Lemma 3.2
Merge sort is correct on arrays of length 1.

This is a trivial base case. Merge sort returns on the first step for $n = 1$ and a list of length 1 is sorted. This is the base case.

Now assume that merge sort is correct on arrays from length 1 to $n - 1$. So the halves l and r will be correctly sorted. In the i^{th} iteration of step 4, the i^{th} smallest element in both arrays is appended to the output. Since this is true for all i, the output is correctly sorted. □

3.2 Asymptotic notation

The variations of O:

- $f(n) \in O(g(n))$ if $\exists c > 0, n_0$ subject to $\forall n > n_0$, $f(n) < cg(n)$. (similar to $\leq$)
- $f(n) \in o(g(n))$ if $\forall c > 0, \exists n_0$ subject to $f(n) < cg(n)$. (similar to $<$)
- $f(n) \in \Omega(g(n))$ if $\exists c > 0, n_0$ subject to $\forall n > n_0, cf(n) > g(n)$. (similar to $\geq$)
- $f(n) \in \Theta(g(n))$ if $\exists a > 0, b > 0, n_0$ such that $\forall n > n_0, af(n) < g(n)$ and $g(n) < bf(n)$. (similar to $=$)

So $f(n) = O(g(n))$ is equivalent to saying $g(n) = \Omega(f(n))$. And $f(n) = o(g(n))$ is equivalent to $g(n) = \omega(f(n))$.

[5] devneal@mit.edu

3.3 Determining runtime

3.3.1 Master theorem

Let $T(n) = aT(\frac{n}{b}) + f(n)$.

Case 1 $f(n) = O(n^{\log_b a - \epsilon})$, for $\epsilon > 0$, has solution $T(n) = \Theta(n^{\log_b a})$.

Case 2 $f(n) = \Theta(n^{\log_b a} \log^k n)$, for $k \geq 0$, has solution $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

Case 3 $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$, and $af(\frac{n}{b}) \leq kf(n)$ for $k < 1$ and sufficiently large n, has solution $T(n) = \Theta(f(n))$.

3.3.2 Guess and check (induction)

Sometimes we can work from a known runtime, and guess and check.

Example 3.3 ($T(n) = 2T(\frac{n}{4}) + n$)
We know that the runtime of merge sort is $2T(\frac{n}{4}) + n$, which is $O(n \log n)$. We can just guess $T(m) \leq cm, \forall m < n$. Then we derive that $c \geq 2$.

Example 3.4 ($T(n) = 2T(\frac{n}{2}) + 1$)
Guess $T(m) \leq cm$.

$$\begin{aligned} T(n) &\leq 2(c\frac{n}{2}) + 1 \\ &\leq cn + 1 \\ &\leq cn \end{aligned}$$

This is not true! There's a constant term, so let's try one too.

Now we guess $T(m) \leq cm - \alpha$. This is derived to work!

3.3.3 Recursion tree

Draw up the tree, count up the work, this is nice and visual. Not tikz-ing this.

4 February 14, 2017

4.1 Polynomials

A polynomial is a function that can be expressed as a sum of monomials.

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ &= \sum_{k=0}^{n-1} a_k x^k \Leftrightarrow (a_0, a_1, \ldots, a_{n-1}) \end{aligned} \tag{4.1}$$

The **degree** of A is $n-1$.

Operations on polynomials:

1. Evaluation: given $A(x)$ and x_0, evaluate $A(x_0)$. Naively, this can be done in $O(n^2)$ time, but using Homer's rule, $A(x_0) = a_0 + x_0(a_1 + x_0(\cdots + a_{n-1}x_0))$, so we can reduce it to $O(n)$ time.

2. Addition: given $A(x)$ and $B(x)$, compute $C(x) = A(x) + B(x)$. This takes $O(n)$.
$$c_k = a_k + b_k$$

3. Multiplication: given $A(x)$ and $B(x)$, find $C(x) = A(x)B(x)$. This results in a polynomial of degree $2(n-1)$. Naively, this takes $O(n^2)$.
$$c_k = \sum_{j=0}^{k} a_j b_{k-j}$$
However, this is a convolution, so using FFT, we can reduce this to $O(n \log n)$!

   "You should be ashamed if something is $O(n^2)$. There are lives at stake."—madry

How can we represent polynomials?

- List of coefficients (above).
- List of roots and scale. $A(x) = c(x - r_0)(x - r_1)\ldots(x - r_{n-1})$.
- Set of points on a curve. $y_j = A(x_j)$. Need $2(n-1)$ points to multiply.

Let us examine the costs of each operation, for the different representations.

	Evaluation	Addition	Multiplication
coef	$O(n)$	$O(n)$	$O(n^2)$
roots	$O(n)$	no solution $d > 4$	$O(n)$
points	$O(n^2)$ interpolation	$O(n)$	$O(n)$

4.2 Fast Fourier transform

What is an efficient way of switching between representations, so that we have good runtime for every task? This motivates the fast Fourier transform, which is one of the most important algorithms of the 20th century. It applies to signal processing, polynomial multiplication, and integer multiplication.

The entire polynomial multiplication using FFT looks like this.

1. Compute $\tilde{A}$ and $\tilde{B}$ in $O(n \log n)$.
2. Compute $\tilde{C} = \tilde{B} \cdot \tilde{A}$ in $O(n)$.
3. Compute C from $\tilde{C}$ in $O(n \log n)$.

FFT is a divide and conquer approach.

1. Divide coefficients into evens and odds.

$$A_{\text{even}} = \sum_{k=0}^{\lceil \frac{n}{2}-1 \rceil} a_{2k} x^k = (a_0, a_2, a_4, \dots) \tag{4.2}$$

$$A_{\text{odd}} = \sum_{k=0}^{\lfloor \frac{n}{2}-1 \rfloor} a_{2k+1} x^k = (a_1, a_3, a_5, \dots) \tag{4.3}$$

2. Compute $A_{\text{even}}(z)$ and $A_{\text{odd}}(z), \forall z \in X^2 = \{x^2 | x \in X\}$. We choose z to be the n^{th} roots of unity, such that X is **collapsible**, which means that
   (a) $|X| = 1$, or
   (b) $|X| = \frac{|X|}{2}$ and X^2 is collapsible too.
3. Combine step.

$$A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2), \forall x \in X \tag{4.4}$$

Let's analyze the running time.

$$T(n, |X|) = 2T(\frac{n}{2}, |X^2|) + O(n + |X|) \tag{4.5}$$

That is equivalent to $O(n \log n)$ with z_n. Everything works out! There is also the matrix view:

$$\begin{pmatrix} 1 & x_0 & \dots & x_0^{n-1} \\ 1 & x_1 & \dots & x_1^{n-1} \\ \vdots & & & \vdots \\ 1 & x_{n-1} & \dots & x_{n-1}^{n-1} \end{pmatrix} \quad A = y \tag{4.6}$$

$$V \qquad\qquad A = y$$

This is a full-rank matrix, so it has an inverse.

$$A = V^{-1} y \tag{4.7}$$

This is a hard problem, but we compute at nice x, so

$$(V^*)(\bar{V^*}) = nI \tag{4.8}$$

And this *is* faster!

5 February 16, 2017

5.1 Amortization

Definition 5.1. An operation has an **amortized cost** of $T(n)$ if *any* sequence of k operations (including worst case) has a cost $\leq kT(n)$.

We will discuss amortized analysis in three ways:

1. aggregate method: total divided by number of operations
2. accounting method: discussed in book and recitation
3. potential method: quantifies a sort of karma (?) to prepay credit for future costs (later in lecture)

5.2 Union-find

Union-find is a data structure for maintaining disjoint sets. We want the following operations:

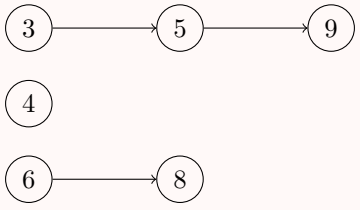
- MAKE-SET(x): add set $\{x\}$ to collection.
- FIND-SET(x): find the set with representative element x.
- UNION(x, y): replace sets with x and y with their union, and appoint a new arbitrary representative.

5.2.1 Linked list representation

Let us represent the data structure as a collection of linked lists.

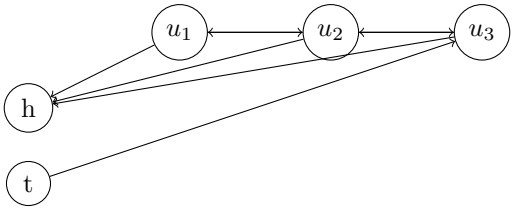
Problem 5.2

Suppose we have sets $\{3, 5, 9\}, \{4\}, \{6, 8\}$, with representative elements 3, 4, and 8.



MAKE-SET would take $O(1)$, but FIND-SET would take $O(n)$ and UNION would also take $O(n)$, since we need to traverse the lists.

So we try **candidate improvement**: we maintain head and tail pointers, for a doubly linked list.



Then, we could have $O(1)$ for FIND-SET. But in the worst case, MAKE-SET is still $O(n)$. The adversary would MAKE-SET(I), for $i \in \{0, 1, \ldots, n-1\}$, then UNION(I,0) for $i \in \{1, 2, \ldots, n-1\}$. Then the operations would take

$$O(n + \sum_{i=1}^{n-1} i) = O(n^2). \tag{5.1}$$

We could always concatenate small lists onto large lists, but then the adversary would make balanced lists... Worst case doesn't improve?

But this is not a common situation! Let n be the total number of elements (by MAKE-SET), and let m be the total number of operations ($m \geq n$).[6] We claim that the cost of all unions is $O(n \log n)$, and the total cost is $O(m+n \log n)$.

Proof. We track a single element u. When it is created, $|S(u)| = 1$. When it is UNIONed with another set $S(v)$, either

Case 1 If $|S(v)| \geq |S(u)|$, u's head pointer must be updated, and $|S(u)|$ at least doubles.

Case 2 Otherwise, the head pointer is not updated, and $|S(u)|$ at least increases.

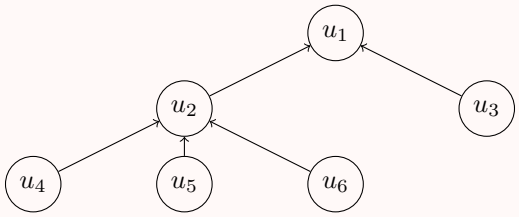
The head pointer to u is updated $\leq \log n$ times, since it is only updated when $|S(u)|$ doubles.[7] Per operation, the amortized cost is $\frac{O(m+n \log n)}{m}$, so each operation is an amortized $O(\log n)$. □

5.2.2 Forest representation

Now let us represent Union-find as a forest of trees.

Example 5.3

Suppose we have set u with representative element u_1.



MAKE-SET would take $O(1)$, just drop down a node. FIND-SET(X) would take $O(\text{height of } S(x))$, where you just traverse down. and UNION(U,V) would take $O(\text{height of } S(u) + \text{height of } S(v))$. This is worst case $O(n^2)$!

[6] Because we need to add all elements first.
[7] We only update head pointers of smaller sets.

First idea: let's merge shorter trees into taller trees. We would need to maintain the height of the tree.

```
Union(u,v)
1  ū ← Find-set(u), v̄ ← Find-set(v)
2  if rank(ū) = rank(ū)
3        rank(u) ← rank(ū) + 1, v̄.parent ← ū.
4  else if rank(ū) > rank(v̄)
5              v̄.parent ← ū
6        else ū.parent ← v̄.
```

Proof left as exercise to reader, but we want to show by induction that rank of tree is always $O(\log n)$, so worst case of each operation of $O(\log n)$.

Second idea: path compression, flatten the tree. Each Find-set traverses a path anyway, so we can just redirect parental pointer of each node visited to the root. We claim that the amortized cost of m operations, n of which are Make-set, is $O(\log n)$ each.

Proof. Let us define $\phi : \text{tree} \to \#$. Then

$$\sum_i \text{make-believe-cost}(\hat{c}_i) \equiv \sum_i \text{true-cost}(\hat{c}_i) + \sum_i \Delta\phi \tag{5.2}$$

And $\hat{c} = c + \phi_{\text{final}} - \phi_{\text{initial}}$. Read the lecture notes. □

Let $\phi(\text{data structure}) = \sum_u \log u$ size.

- Make-set: $\hat{c} = 1 + 0$
- Union: cost of 2 Find-set + link. $\hat{c}_{\text{link}} \leq \log rep(S(u)).size + rep(S(v)).size - \log rep(s(u)).size \leq O(\log n)$
- Read the notes.

Each idea produces an algorithm of $O(m \log n)$. What if we combined them? Well we find that any m operations, with n Make-sets, has a worst-case running time of $O(m\alpha(n))$, where α is the inverse-Ackerman function, which grows really slowly!

6 February 17, 2017

6.1 FFT review

We are given

$$A(x) = a_0 + a_1 x + \dots a_{n-1} x^{n-1} \tag{6.1}$$
$$B(x) = b_0 + b_1 x + \dots b_{n-1} x^{n-1}. \tag{6.2}$$

How quickly can we compute $C(x) = A(x)B(x)$? Naively, $O(n^2)$, but using FFT, $O(n \log n)$. We choose to evaluate $A(x)$ and $B(x)$ at the $2n$ roots of unity.

To multiple $A(x)$ and $B(x)$, for a degree of n:

1. find A_{even} and A_{odd}.
2. recursively evaluate A_{even} and A_{odd} at the square of the n^{th} roots of unity.
3. recursively multiply the n points
4. interpolate into $C(x)$

Again in matrix form,

$$\begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \vdots \\ \vdots & \omega_n^2 & \omega_n^4 & \vdots \\ 1 & \vdots & \vdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = y. \tag{6.3}$$

We can compute this by divide and conquer in $O(n \log n)$.

Problem 6.1 (Minkowski sum)

Let X and Y be sets of integers. The Minkowski sum $X + Y$ is set $\{x + y | (x, y) \in X \times Y\}$. (Also known as Cartesian sum).

How quickly can we compute the Minkowski sum, given two sets of size n? Let $f(x) = \sum_{i \in X} x^i$ and $g(x) = \sum_{j \in Y} x^j$. Then the Minkowski sum contains the exponents of $f(x)g(x)$. We can find that with FFT in $O(w \log w)$, where w is $\max(\max(x), \max(y))$, so numbers could be arbitrarily high.

7 February 23, 2017

7.1 Competitive analysis

We will compare our algorithm against theoretical lower bounds. An **online algorithm** is one in which we receive a sequence of inputs, where we must decide on the best possible outcome for the current input, before seeing more. Conversely, an **offline algorithm** is one in which we can see the entire input before deciding.

Example 7.1 (Tetris)

An online version would be seeing tetronimoes, one by one. An offline version would be to see all the blocks at the same time. We can't do that much better in the offline version.

Definition 7.2 (Competitive analysis)**.** An online algorithm A is α-competitive if there exists a constant k, such that for any sequence s of operations,

$$C_A(s) \leq \alpha C_{\text{best}}(s) + k, \tag{7.1}$$

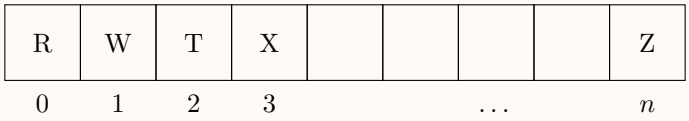
where C_{best} is the cost of the best offline algorithm.

7.1.1 Self-organizing lists

We will compare an online algorithm to the best possible offline version.

Problem 7.3 (Self-organizing lists)

Given a list L containing n elements, we have one operation, ACCESS(x), where x is a key to some element. Return the element with the given key.



The caveat is that we may permute the list after each ACCESS, and it takes $O(1)$ to transpose adjacent elements. The goal is to choose transpositions to minimize the cost of the algorithm, given an input sequence, for all sequences ($C_A(s)$).

Imagine that we are the adversary. Worst-case analysis: we always select the last element. In this case, $C_A(s) = \Omega(|S|^n)$.

Now consider the average case. Imagine that an element x is accessed with $\Pr\{x\}$. Then

$$\mathrm{E}[C_A(s)] = |s| \sum_{x \in L} \Pr\{x\} \operatorname{rank}(x). \tag{7.2}$$

This doesn't help the worst case, but we can improve the average case.

> "In research, you can look at this and say, 'I'll just pick another problem!' "—tidor

> "If the best algorithm isn't very good, then maybe a braindead algorithm won't be much worse!"—tidor

We could just swap every element to the front, after every access. (LRU cache!) P Q R S X, access X. We transpose to X P Q R S in $\text{rank}(x) - 1$ transposes. The "move-to-front" or MTF heuristic performs "well." The total cost of an access is $2 \cdot \text{rank}(x) - 1$.

We show that MTF is 4-competitive for self-organizing lists.

Proof. Let L_i and L_i^* be the list after i^{th} access using MTF and the optimal algorithm, respectively. Let c_i and c_i^* be the respective costs.

$$c_i = 2\text{rank}(x_i)_{L_{i-1}} - 1 \tag{7.3}$$
$$c_i^* = \text{rank}(x_i)_{L_{i-1}} + t_i, \tag{7.4}$$

where t_i is the number of transpositions. We perform amortized analysis with a potential function. Let

$$\begin{aligned}\phi &= 2 \times \text{the number of inversions between } L_i \text{ and } L_i^* \\ &= 2\left|\left\{(x,y) : (x <_{L_i} y \text{ and } y <_{L_i^*} x\right\}\right|,\end{aligned}$$

where the weird notation means $\text{rank}(x)_{L_i} < \text{rank}(y)_{L_i}$.

Note that $\phi(L_0) = 0$ since we start at the same list, and $\phi(L_i) \geq 0$ since the minimum number of inversions is 0.

At each step, we could represent the two lists with a heat map.

After accessing element i, we cross out i^{th} column and row. There there are 4 sets.

1. A is third quadrant,
2. Draw this diagram from notes later

Each move by opt creates at most one inversion, so

$$\phi(L_i) - \phi(L_{i-1}) \leq 2(|A| - |B| + t_i) \tag{7.5}$$

The amortized cost of a single access is

$$\begin{aligned}\hat{c_i} &= c_i + \phi(L_i) - \phi(L_{i-1}) \\ &\leq (2r - 1) + 2(|A| + |B| + t_i) \\ &= 2r - 1 + 4\,|A| - 2r + 2 + 2t_i \\ &\leq 4(r^* + t_i) = 4c_i^*\end{aligned}$$

Now consider the total cost of $|s|$ operations. I'll type it up later. □

There are several things we should note.

1. There's no need to find the actual optimal algorithm.
2. If $L_0 \neq L_0^*$, then the cost to make them the same could be $\Theta(n^2)$, but we treat n as a constant; the size we care about is s.

7.2 How to learn in this class?

Take stuff apart, ask it questions! Figure out how things actually work!

8 February 24, 2017

8.1 Amortized analysis

A data structure has amortized cost $T(n)$ if *any* sequence of k operations (including worst case) takes at most $kT(n)$ time. We will focus on two examples.

Example 8.1 (Queue)
Imagine a data structure, a queue from 2 stacks.

- ENQUEUE: we push onto S_1.
- DEQUEUE: if S_2 is not empty, pop from S_2. If it is empty, pop all elements from S_1 to S_2, and pop from S_2.

Example 8.2 (Binary counter)
We maintain a binary counter, $[0, 0, \ldots, 0]$. We have one operation, INCREMENT: starting form the right, flip all 1s to 0s, upon finding a 0. Flip that to 1, and return.

8.1.1 Aggregate analysis

Definition 8.3 (Aggregate analysis)**.** Add up total work for n operations, and divide that total by n.

For the queue from 2 stacks, focus on a single element x_0 during a sequence of operations. The most that x_0 contributes to is

1. a push onto S_1,
2. a pop from S_1, push onto S_2,
3. and a pop from S_2.

x_0 only contributes $O(1)$ work over $O(n)$ operations, so all n elements contribute at most $O(n)$ total work.

For the binary counter, consider a sequence of n calls to INCREMENT. The first bit is flipped once for each call, so n bit flips. The second bit is flipped once every other call, so $n/2$ bit flips. The third bit is flipped once every four calls, etc.

$$\text{total} = n \sum_{k=0}^{\log_2 n} \frac{1}{2^k} \leq 2n = O(n) \tag{8.1}$$

8.1.2 Accounting method

Definition 8.4 (Accounting method)**.** Operations can deposit "coins" in a "bank" so later operations can draw from the bank.[8]

For the queue with two stacks, ENQUEUE does 1 work pushing x, deposits 3 "coins" into the bank (4 coin cost). DEQUEUE uses 1 coin if S_2 not empty. If S_2 is empty, we cover the pop and push with 2 coins. The last pop costs the last coin. So total cost is for n coins is $4n$, which is $O(n)$.

For the binary counter, deposit one coin when flipping a bit $0 \to 1$ (This is because expensive operations are when we have to scan a bunch of 1s and flip a 0.) When we INCREMENT, assume there are k 1s before the first 0. This means there are at least k coins in the bank. Therefore, the cost of INCREMENT is only the cost to flip the final $0 \to 1$. So there is $O(1)$ amortized cost.

8.1.3 Potential method

Definition 8.5 (Potential method)**.** Define a potential function ϕ which maps the state of the data structure to a number. The amortized cost of any operation is then $\hat{c}_i = c_i + \Delta\phi = c_i + \phi_i - \phi_{i-1}$.[9]

> "Coming up with a ϕ is black magic"—devneal

For the queue, let us define $\phi = 2\left|S_1^i\right|$. ENQUEUE, $\hat{c}_i = 1 + 2(\left|S_1^i\right| - \left|S_1^{i-1}\right|)$. DEQUEUE,

$$\begin{aligned}\hat{c}_i &= 2\left|S_1^{i-1}\right| + 1 + 2(\left|S_1^i\right| - \left|S_1^{i-1}\right|) \\ &= 2\left|S_1^{i-1}\right| + 1 + 2(0 - \left|S_1^{i-1}\right|) = 1 \text{ or} \\ &= 1 \text{ for unexpensive.}\end{aligned}$$

For the binary counter, INCREMENT is expensive when we flip a lot of 1s, so let us define $\phi = \#$ of 1s. Assume there are k 1s on the right and INCREMENT is called.

$$\hat{c}_i = k + 1 + \Delta\phi = k + 1 + (-k + 1) = 2 = O(1) \tag{8.2}$$

> "this pset is making up for the easy last pset, so the amortized hardness of the pset is traditional"–devneal

[8] If there are cheap operations, each contributing a little credit, the little operations will be fine.

[9] ϕ should decrease a lot during an expensive operation.

9 February 28, 2017

9.1 Minimum spanning trees

Definition 9.1 (Spanning tree)**.** A tree is a connective graph with no cycles, and a **spanning tree** is a subset of the graph that spans all vertices.

We pose the following problem.

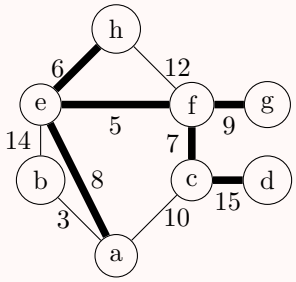
Problem 9.2 (Minimum spanning tree)

Given a weighted graph $G = (V, E)$, find a spanning tree of minimum weight, $w(T) = \sum_{e \in T} w(e)$.

For example, we could be trying to lay power cables through the city and want to find the minimum cost. There are also applications in many AI fields like computer vision and NLP.

Example 9.3

Supposed we have the following graph, with edges in the MST bolded. Here, we see that we would add the $a \to b$ edge to complete the MST.



Theorem 9.4 (MST property)

Given connected weighted graph $G = (V, E)$, let $U \subset V, U \neq \emptyset$. If (u, v) is a light edge (lowest cost) with the property that $u \in U$ and $v \in V - U$, then there exists a MST with the edge (u, v) in it.

Proof by cut and paste. Assume for contradiction that (u, v) is not in any MST. Then $\exists (u', v')$ connecting U and $V - U$ in T. Now say we replace (u', v') with (u, v) in T'. So $w(T') > w(T)$ by assumption, which contradicts $w(T') \leq w(T)$ from the fact that T is a MST. Therefore, (u, v) must be in some MST. □

We realize that this is a cut in the graph. If edges do not cross the cut, then the cut respects that edge set.

Proposition 9.5

If (u, v) is a unique-weight light edge crossing the cut, then it is in *every* MST.

9.2 Kruskal's algorithm

Algorithm 9.6 (Kruskal's algorithm)

Initialize $T = (V, \emptyset)$. Examine edges by increasing weight. For each edge, we add if it joins previously unconnected components, otherwise reject. The algorithm terminates when all components are connected or all edges have been examined (and there is no spanning tree).

Proof of correctness. The loop invariant is that prior to each iteration, $T \subseteq MST$. At initialization, T is empty so this is trivially true. Each edge added to T is a light edge crossing a cut of V. By the MST property, it is in a MST. At termination, we've added all edges that belong in a MST, so we must "have" a MST.

For distinct edge weights, the MST is unique, so "a MST" = "the MST." For non-distinct edge weights, there may be multiple MSTs. □

Question 9.7. How do we ensure that there's always a cut? We only add edges between disconnected subsets.

Question 9.8. How do we deal with identical weights? Break ties arbitrarily.

9.2.1 Implementation

We can implement this with the Union-find data structure.

```
1  Initialize T ← ∅.                                   // Θ(1)
2  for v ∈ V, MAKE-SET(v).                             // Θ(|V|)
3  Sort E by weight.                                   // Θ(|E| log |E|)
4  for e = (u, v) ∈ E                                  // Θ(T_findset + T_union)
5      if FIND-SET(u) ≠ FIND-SET(v)                    // not in same tree
6          add e to T, UNION(u, v).
```

We analyze the total running time.

$$\begin{aligned} T &= O(E \log E) + O(V) + O(E\alpha(V)) \\ &= O(E \log V) \end{aligned} \tag{9.1}$$

10 March 2, 2017

10.1 Administreview

Quiz 1 is coming! Unfortunately it's on π-day, from 7:30-9:30p, for lectures 1-9. There are two review sessions on March 10 (meta-review) and 11.

10.2 Maximum flow

Yay more graph theory—makes me happy.

A **flow network** is a directed graph $G = (V, E)$, a source s and sink t, edge capacities $c : E \to \mathbb{R}_+$. G can also be represented as a complete graph, with some capacities 0 (for edges that "didn't" exist. We define gross flow

$$g : E \to \mathbb{R}_+, \tag{10.1}$$

feasibility as

$$g(u, v) \leq c(u, v), \forall (u, v) \in E, \tag{10.2}$$

and flow conservation as

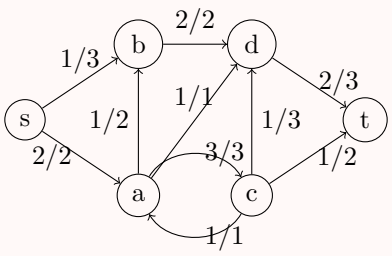
$$\forall_{v \neq s,t} \sum_u g(u, v) - \sum_u g(v, u) = 0. \tag{10.3}$$

Gross flow is hard, so we have the notion of **net flow**, $f : V \times V \to \mathbb{R}$, where positive flow is along the arrow, and negative flow is against. This satisfies the properties that

- $\forall_{(u,v)}, f(u, v) \leq c(u, v)$
- $\sum_u f(u, v) = 0$
- $\forall_{(u,v)}, f(u, v) = -f(v, u)$ (flow symmetry)
- $f(u, u) = 0$

Example 10.1

We have a sample graph and its flows.



Problem 10.2 (Maximum flow problem)

Given $G = (V, E, s, t, c)$, find a flow of max value $|f| = \sum_v f(s, v)$

There is always the trivial empty flow. There are also simple flows: flow cycles ($|f| = 0$) and paths from $s \to t$ ($|f| > 0$).

Proposition 10.3 (Flow decomposition)

Any flow f can be decomposed into a union of flow cycles and flow $s \to t$ paths.

Sketch of proof. We can decompose G into a subgraph (support graph) with edges $f(u,v) > 0$. Let $e = (u,v) \in S(G)$ and $v \neq s,t$. Then there exists at least one u', such that $(v,u') \in S(G)$, since for each flow in, there must be a flow out. At some point, we eventually either find t, or we find a cycle. If P is a $s \to t$ path or cycle, we can reduce f on P by the **bottleneck capacity**, $\min_{e \in P} f(e)$. Now at least one edge is no longer in $S(G)$, since its flow was reduced to 0. We can induct on this. □

10.3 Minimum cut

A dual relationship between $s \to t$ paths and $s \to t$ cuts motivates **minimum cuts**. Let f^* be a maximum flow, and $F^* = |f^*|$. How do we determine if there even exists a flow? If there is a path from $s \to t$ with edges of positive capacity, there is a flow.

Now the contrapositive: if $F^* = 0$, how could we verify it? Suppose for contradiction that there's a non-zero flow. Then the flow always decomposes from a union of flow cycles and $s \to$ flow paths, but there aren't any $s \to t$ paths, so there are only flow cycles, which push 0 flow.

Definition 10.4 ($s \to t$ cut)**.** An *$s \to t$* **cut** is a cut $(S, V - S)$ such that $s \in S$ and $t \in V - S$. Then the **capacity** of S is defined as $c(S) = \sum_{u \in S} \sum_{v \in V - S} c(u,v)$.

$F^* = 0$ if and only if $\nexists s \to t$ path $\in S(G)$, or $c(S^*) = 0$, where

$$S^* = \arg\min c(S), \tag{10.4}$$

the minimum $s \to t$ cut. This property shows that $s \to t$ paths are dual to $s \to t$ cuts.

For a given $s \to t$ path S, we define flow

$$f(S) = \sum_{u \in S} \sum_{v \in V - S} f(u,v). \tag{10.5}$$

Claim 10.5. For any flow f and any $s \to t$ cuts S and S',

$$f(S) = f(S'). \tag{10.6}$$

Proof. For any $s \to t$ cut, $F^* = |f^*| = f(\{\ldots\}) = f(S^*) \leq c(S^*)$. □

So we have shown weak duality of max flow and min cut.

Definition 10.6 (Residual graph)**.** The **residual graph** G_f given G and current flow f, has capacities $c_f(u,v) \leftarrow c(u,v) - f(u,v)$.

11 March 3, 2017

11.1 Minimum spanning trees (continued)

Given $G = (V, E, w)$ where $w : E \to \mathbb{R}$, the MST of G is a set of edges $T \subseteq E$ that is a tree connecting all $v \in V$ and has minimum weight $\sum_{e \in T} w(e)$.

Definition 11.1. A **safe edge** is an edge, which when added to a set T' of edges of some MST, produces another set of edges which are a subset of some MST.

Algorithm 11.2

We have the following algorithm.

GENERIC-MST(V, E, w):

```
1  T ← ∅
2  while T is not spanning
3        find safe edge e for T
4        T ← T ∪ {e}
5  return T
```

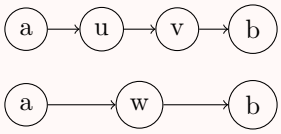
We can find safe edges with the greedy-choice property: for any cut $S, V - S$, for $S, V - S \neq \emptyset$, any least-weight edge (u, v) with $u \in S, v \in V - S$ must belong to some MST.

Sketch of proof. Consider a cut $(S, V - S)$ with minimum crossing edge e, and an MST T with $e \notin T$. we can create a new spanning tree by replacing the edge $e' \in T$ which crosses $S, V - S$ with e. □

Contraction of an edge $e = \{u, v\}$ produces a graph $G - e$ where u and v are merged into a single vertex. Edge e is destroyed, and all edges incident to u or v are instead incident to the merged vertex.

Example 11.3 (Edge contraction)

For example, we merge u and v into w.



Proposition 11.4 (Optimal substructure property)

If an edge e belongs to some MST T of G, and T' is some MST of $G' = G - e$, then $T = T' \cup \{e\}$ is an MST of G.

Proof. T is spanning because it connects the components split by contracting e. T is minimum because

$$w(T) = w(T') + w(e) \leq w(T^* - e) + w(e) = w(T^*). \tag{11.1}$$

□

11.2 Prim's algorithm

Algorithm 11.5 (Prim's algorithm)

Select a vertex to start. For each iteration, find the lightest edge connecting T to an isolate vertex.

We analyze the running time. Naively, $O(|V| + |E|\,|V|)$, which is cubic in $|V|$, but we can do better.

Algorithm 11.6

Dijkstra similar modification...

Now the runtime depends on the data structure, $O(|E| \cdot \text{relax} + |V| \cdot \text{extract})$. but using Fibonacci heaps, we can have $O(|E| + |V| \log |V|)$. We can have a similar running time as Dijkstra's if we start from the same vertex.

For the residual graph, if we currently have flow at capacity, we can reverse the edge. It's a matter of how much deltas we can add to the current graph. If there's a path in the residual, then we can push more flow through.

Ford-Fulkerson continually finds augmenting paths, looping until done.

12 March 7, 2017

12.1 Ford-Fulkerson algorithm

The professor admits that we went over stuff too quickly near the end, last class.

Example 12.1

During sad times (1956), the USSR wanted to send the max-flow worth of supplies between cities. On the other hand, the US wanted to figure out which supply lines to interrupt to disconnect the graph. Together, this was essentially max-flow-min-cut.

Algorithm 12.2 (Ford-Fulkerson algorithm)

We can keep finding augmenting paths to find the max-flow.

```
1  f(u, v) = 0, ∀u, v ∈ V
2  while an augmenting path exists in G_f with DFS
3        push G_f(p) units of flow
4  return f
```

> "At other institutions, they only care if it's fast, but we're at MIT so we have to show that it's correct first!"—madry

Theorem 12.3 (Max-flow-min-cut theorem)

The following statements are equivalent.

1. $\exists s \to t$ cut S with $|f| = c(S)$.
2. f is a max flow (i.e. $|f| = F^*$).
3. f admits no augmenting paths.

Statements 1 and 2 imply strong duality, $F^* = c(S^*)$. Statements 2 and 3 prove that the Ford-Fulkerson algorithm is correct.

Proof. We will show that $1 \implies 2$, $2 \implies 3$, and $3 \implies 1$.

1. By weak duality, $|f| \leq F^* \leq c(S^*)$. However, $c(S^*) \leq c(S)$ since that's the minimum cut, and $c(S) = |f|$, so $c(S^*) = |f|$.
2. If there is an augmenting path, then we are not done since we can push even more flow.
3. The set of vertices reachable from s does not contain t, so this forms a $s \to t$ cut. The the flow across the cut is 0 on the residual graph, so each of the edges is fully saturated, so the net flow is equal to the capacity of the cut.

□

> "Now we continue from where other schools start and ask 'how fast does it run?' "—madry

Each iteration takes $O(m)$ time, where m is the number of edges (since the graph is connected). Assume that $\forall c, c \in \mathbb{Z}$ and $c \in [0, C]$. Each iteration, the capacity increases by at least 1 (augmenting path), since the residual graph also has all integral capacities. Therefore, the number of iterations is bounded by $F^* \leq nC$, since there are at most n edges leading from the source.

So the total runtime is $O(mnC)$, which is pseudo-polynomial since the value of C is exponential in its size.

We can show that for $\mathbb{Q}$, the running time is still pseudo-polynomial. However, if capacities are $\mathbb{R}$, then the algorithm could never terminate.

Lemma 12.4 (Flow-integrality lemma)
If $c \in \mathbb{Z}$, then $\exists f^*$ such that f^* is a max flow of integral value.

12.2 Maximum bottleneck algorithm

We don't like that Ford-Fulkerson is exponential in size of C. We can greedily choose any path that maximizes increase in $c_f(p)$.

Claim 12.5. We can find the maximum bottleneck path in $O(m \log n)$ time.

We can use that to show that the running time is $O(m^2 \log n \log nC)$, which is actually polynomial.

12.3 Edmonds-Karp algorithm

Choose an augmenting path with minimum number of edges. This runs in $O(m^2 n)$, which is strongly polynomial.

Applications include the stable marriage problem.

13 March 9, 2017

13.1 Linear programming

Linear programming is a very common and versatile tool.

Example 13.1 (Elections)
Imagine that there are three possible policies: build roads, gun control, farm subsidies, and gas tax. There are also three demographics: urban, suburban, and rural. The votes gained matrix is the following.

$$A = \begin{pmatrix} -2 & 5 & 3 \\ 8 & 2 & -5 \\ 0 & 0 & 10 \\ 10 & 0 & 0 \end{pmatrix} \tag{13.1}$$

> "Suburban guys don't care. They drive a lot, but also have a lot of money."—madry

We have the constraint that the populations are $(100, 200, 30)$ and we require at least $(50, 100, 25)$, a majority in each voting district. How can we achieve these with minimal cost?

Solution. Let $y = (y_1, y_2, y_3, y_4)$ be the amount spent per policy, $c = (1, 1, 1, 1)$, and $b = (50, 100, 25)$. We want to

$$\begin{aligned} \text{maximize } & z = c^T y \\ \text{subject to } & Ay \geq b, \\ & y \geq 0. \end{aligned}$$

From our professor's notes,

$$y^* = \frac{1}{111}(2050000, 425000, 625000) \approx (18468, 3829, 5631).$$

But how do we get this answer? A **linear program** is some linear optimization problem subject to linear constraints.

Definition 13.2 (Standard form)**.** Let $x \in \mathbb{R}^n, c \in \mathbb{R}^n, b =\in \mathbb{R}^m$, and $A \in \mathbb{R}^{mn}$.

$$\begin{aligned} \text{maximize } & z = c^T x \\ \text{subject to } & Ax \leq b, \\ & x \geq 0. \end{aligned}$$

Question 13.3. Why is the standard form universal? Just stick in factors of 1 and -1, or flip around $\geq$ and $\leq$ for equality. For equality, just copy each equation and introduce the other version. If $x \in \mathbb{R}$, then introduce x_i^+ and $x_i^- \geq 0$, and transform $x_i \to x_i^+ - x_i^-$. 18.200 has a great table.

13.1.1 Geometric intuition

Let c be the direction of optimization, and each entry in $Ax \leq b$ forms a half-space bounded by some hyperplane. All the constraints specify a polytope, which is a subspace of $\mathbb{R}^n$, where each face corresponds to some hyperplane constraint. In 2D, we would have lines and an enclosed polygon.

> "A polytope." "A poly—what?" "A polytope. p-o-l-y-t-o-p-e."

TODO draw in the graphs, since I think they're interesting.

13.2 LP algorithms

13.2.1 Simplex algorithm

Start at one of the vertices. Does moving to a neighbor vertex improve the objective? If so, move. Keep pivoting. This works since polytopes are convex. Running time very practical, but worst case can be exponential.

13.2.2 Ellipsoid method

Invariant maintains that optimum is within an ellipsoid. After some number of iterations, space is small enough to just find the solution. Polynomial time worst case, but not practical. There are large constant factors.

13.2.3 Interior point method

Start at the middle of the polytope and keep moving vaguely towards c and will eventually converge. This method is both polynomial time *and* practical. Most packages switch between simplex and interior point method.

13.3 Duality

We have the primal and dual linear programs.

$$\begin{array}{rr} \text{primal} & \\ \text{maximize } z = c^T x & \\ \text{subject to } Ax \leq b, & \\ x \geq 0. & \end{array} \qquad \begin{array}{rr} \text{dual} & \\ \text{minimize} & w = b^T y \\ \text{subject to} & A^T y \geq c, \\ & y \geq 0. \end{array} \tag{13.2}$$

Theorem 13.4 (Weak duality)
If x is feasible for the primal and y is feasible for the dual, then $c^T x \leq b^T y$.

Proof. Let x and y be feasible solutions for the primal and dual. Then by multiplying both sides by y,

$$\sum_i y_i \sum_j A_{i,j} x_j \leq \sum_i y_i b_i$$
$$yAx \leq b^T y$$

We do the same for the primal.

$$\sum_j x_j \sum_i A_{i,j} x_j \leq \sum_j x_j b_j$$
$$xA^T y \geq c^T x$$

We see that $yAx = xA^T y$, so $c^T x \leq b^T y$. □

14 March 10, 2017 (Review 1)

"Do we want them to stare at the pies for the next 2 hours?"—madry

Fall 2014, Quiz 1, Problem 2.

1. *Prove that an edge is not in any MST.* If an edge is the heaviest edge in any cycle, then it is not in any MST.
2. *Prove that an edge is in every MST.* The edge in question is the minimum edge in the tree. Kruskal's will always add this edge.
3. *Suppose we have an MST and add an edge. The new edge introduces a cycle, so we delete the heaviest edge. Prove this is still an MST.* Let $G' = (V, E \cup e)$. Then all edges originally deleted are still deleted, and all the edges not in the cycle are still in the MST. The heaviest edge in the cycle is replaced with e.
4. *Now we have a new vertex v' and edges connecting it E'. Find the MST of $G' = (V \cup V', E \cup E')$.* Iteratively apply part b.

15 March 11, 2017 (Review 2)

Today was the second review session, which went over relevant content for quiz 1. These notes were derived from chalkboard photos by the amazing Tracy Cheng.[10] The topics include:

1. asymptotics + recurrences
2. divide and conquer: median finding and FFT (polynomial multiplication)
3. amortized analysis: Union-find, competitive analysis
4. minimum spanning trees
5. max-flow (Ford-Fulkerson, bipartite matching)
6. linear programming

15.1 Median finding

The algorithm for median finding is the following.

1. split array into $n/5$ groups of size 5
2. sort each group in $O(1)$ time, since they are small
3. recursively find the median of medians
4. use the median of medians as pivot p

The key takeaway is that the pivots are approximately balanced (in the middle), so the recursion is

$$T(n) = O(n) + T(\frac{n}{5}) + T(\frac{7n}{10}) = O(n).$$

15.2 FFT

Let $A(x) = a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$. The coefficient representation is a vector $(a_0, a_1, \ldots, a_{n-1})$, and the point-value representation contains d points evaluated by the polynomial. If we were to multiply $A(x)$ and $B(x)$, defined similarly with constants b_i, we would get $C(x) = A(x)B(x)$, which is a polynomial of max degree $2n - 2$.

$$\begin{aligned} c_0 &= a_0b_0 \\ c_1 &= a_0b_1 + a_1b_0 \\ &\cdots \\ c_k &= \sum_{i+j=k} a_ib_j = \sum_{i=0}^{k} a_ib_{k-i}, \end{aligned}$$

which is the convolution of a_i and b_j. We can use FFT to find this quickly.

1. evaluate A and B at $2n - 1$ points.
2. multiply the points in $O(n)$.

[10] tcheng17@mit.edu

3. interpolate to reconstruct C, through DFT.

FFT runs in $O(n \log n)$ because at each step, we evaluate 2 polynomials of degree $n/2$ at $n/2$ points (since roots of unity are collapsible).

15.3 Union-find

Union-find is a data structure that maintains disjoint sets. It has three operations:

1. Make-set adds $\{x\}$ to the data structure.
2. Find-set finds the set (representative element) containing x
3. Union combines two sets

We can achieve amortized $O(\log n)$ cost.

16 March 16, 2017

16.1 Game theory

Example 16.1 (Prisoner's dilemma)

Two people are caught. if neither tells, they're okay. If one tells, the other is screwed. If both tell, both are kinda screwed.

The maximum-welfare solution is silence, but the stable solution is that both tell. A game is a thought experiment to reason about what rational agents do in situations of conflict. However, these do not account for factors like a criminal's gang might punish him if he tells, or two players could never face each other again if the other tells (on a game show).

> "If you want someone to be selfish, just tell them to be rational."—madry

Problem 16.2 (Two-person zero-sum game)

Let A be the utility matrix of player a and B be the utility matrix of player b. Element $a_{i,j}$ is the gain for player a when he plays i and b plays j.[a] In addition,

$$a_{i,j} = -b_{i,j}, \forall i, j$$

[a] Sometimes player a is known as the row player and player b is the column player.

Example 16.3 (Rock-paper-scissors)

Let the rows and columns be in order of R-P-S.

$$\begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix}$$

There is no stable single outcome, but the stable distribution is $(1/3, 1/3, 1/3)$, randomized for both.

Definition 16.4. A Nash equilibrium is a pair of strategies such that no player can improve his/her expected utility by deviating.

These strategies are stable even as common knowledge.

Proposition 16.5

Every game with a finite umber of players ad finite number of actions has a Nash equilibrium.

Theorem 16.6 (Min-max)

For any matrix A and x, y probability distributions, let $V_R = \min_x \max_y xAy$ and $V_C = \max_y \min_x xAy$. Then $V_C = V_R$.

We observer that $V_C \geq V_R$ since R has an advantage.[11] This theorem proves the existence of a Nash equilibrium for a two-person zero-sum game. Let $x^* = \arg\max_x \min_y xAy$ and $y^* = \arg\min_y \max_x xAy$. Then (x^*, y^*) is the Nash equilibrium.

> "Most people have thinking caps but I have a thinking T-shirt." *takes off jacket* "Algorithmic powers activate!" *class cheers* "You can borrow it for quizzes at a fee!"

Proof of min-max. We express V_R and V_C as dual linear programs.

$$\begin{array}{cc} \text{primal} & \\ \text{maximize} & z \\ \text{subject to} & \sum_{i,j} A_{i,j} x_i - z \geq 0, \\ & \sum_i x_i = 1 \\ & x \geq 0. \end{array} \qquad \begin{array}{cc} \text{dual} & \\ \text{minimize} & u \\ \text{subject to} & \sum_{i,j} A_{i,j} y_j - u \leq 0, \\ & \sum_j y_j = 1 \\ & y \geq 0. \end{array} \tag{16.1}$$

Let V_R and V_C be the optimums for the primal and dual, respectively. By strong duality, $V_R = V_C$.

□

16.2 How to get rich

> "If that's your goal, I think you chose the wrong university. Let me know if you want to transfer."—madry

We can model the stock market as a simple random walk, where you go up or down. Let X_t be the market index at the end of day t. Each day, you gain a million if you guess right and lose a million if you guess wrong.

There are m advisors who each supply their own prediction. We want to minimize regret,

$$\#\text{ of our mis-predictions - }\#\text{ of mistakes by } \textit{best} \text{ expert.}$$

A naive approach is to choose the majority vote of the experts. This is bad, because everyone except for one is always telling you the wrong answer.

Now assume that the best expert is perfect.

Algorithm 16.7 (Halving algorithm)
Maintain a set S of trustworthy experts. Initially, $S = \{1, 2, \ldots, m\}$. At day t, we take the majority of experts in S. At the end of each day, remove everyone who predicted wrong.

Claim 16.8. Using the halving algorithm, regret $\leq \log_2 m$.

Whenever we make a mistake, size of S halves (majority wrong). We assumed that one expert is perfect, so we won't get rid of everyone. If S ever becomes empty, re-initialize and continue. We can show that the regret $\leq m^* \log_2 m$, where m^* is the optimum.

[11] R and C denote row and column.

However, replenishing S loses data. So instead of categorizing experts in a binary fashion, we maintain a weighted trustworthiness for each expert. At day t, we predict by a weighted majority. At the end of each day, $\forall_i$ that made a mistake, $w_i \leftarrow \frac{1}{2} w_i$. We can show that for this, regret $\leq 1.4m^* + 2.4 \log_2 m$.

17 March 17, 2017

17.1 Zero-sum games

We are given players R, C, where R has a set of m possible strategies and C has a set of n possible strategies. There is a payoff matrix A which is a m by n matrix, where $a_{i,j}$ is the payoff to player R for taking actions i and j. There are pure strategies (exact) and mixed strategies (probability distribution).

Given strategies p^*, q^* respectively, neither R nor C can gain by unilaterally switching strategies. The **min-max theorem** says that every zero-sum game has *at least* one Nash equilibrium, and we can find a Nash equilibrium of the game when both players seek to maximize their worst-case payoff.

Let player R play $p = (p_1, p_2, \ldots, p_m)$. For fixed p, player C will respond with

$$q = \arg\min_q \sum_{i,j} p_i q_j A_{i,j}.$$

So player R will optimize to play the

$$p^* = \max_p \min_q \sum_{i,j} p_i q_j A_{i,j}. \tag{17.1}$$

The same applies for player S, who plays the

$$q^* = -\max_q \min_p \sum_{i,j} p_i q_j A_{i,j}. \tag{17.2}$$

We claim that there is strong duality, and $p^* = q^*$. Refer to equation 16.1 for the linear program.

17.2 Learning from expert advice

We have a set of n experts, who each provide a 0-1 decision on each time t. We vote based on these experts to minimize the number of mistakes. We compare the number of mistakes we make to the number of mistakes made by the best expert.

Algorithm 17.1 (Randomized halving)

Maintain a pool of trustworthy experts S. In each round, randomly select one expert and follow his/her prediction. Remove all experts who were wrong from S.

We can show that the expected number of mistakes for this algorithm is bounded by $O(n)$, under the strong assumption that at least one expert is always right. Let F_t be the fraction of experts who were wrong on round t, who are still in S. Then the expected number of mistakes is

$$\mathrm{E}[m] = \sum_t F_t.$$

Let P_t be the fraction of experts who are still in S. Then

$$P_t = \Pi_t (1 - F_t) \geq \frac{1}{n},$$

and linearizing,

$$\sum_t 1 - F_t \geq -\log n,$$

and

$$\mathrm{E}[m] = \sum_t F_t \leq \log n. \tag{17.3}$$

Algorithm 17.2 (Randomized weighted majority)

Initialize every expert to have equal weight 1. At each round t, choose a random expert with probability proportional to weight. Decrease

$$w_i \leftarrow w_i - \epsilon$$

of all experts who were wrong.

We will prove that

$$\mathrm{E}[m] \leq (1+\epsilon)m^* + \frac{\log n}{\epsilon}.$$

Let F_t now be the weighted fraction of all experts. $W_T \leq \Pi_{t\leq T} 1 - \epsilon F_t) = n\Pi(1-\epsilon)F_t$ If $w^*_{i,j}$ is the weight of the best expert at time t, then $w^*_{i,j} \geq (q-\epsilon)^{m^*}$, related to the number of total number of mistakes by the best expert. At every time step, $w^*_{i,j} \leq W_t$, which implies that

$$\log 1 - \epsilon m^* \leq \log n + \sum \log 1 - \epsilon F_t \leq -\epsilon F_t. \tag{17.4}$$

Using Taylor expansions again,

$$\epsilon \mathrm{E}[m] \leq \log n - m^* \log 1 - \epsilon \leq \log n + m^*(1+\epsilon)\epsilon.$$

We divide by ϵ and we are done.

We see that there is a tradeoff between best experts and how much we down weight.

18 March 21, 2017

18.1 Randomized algorithms

A **randomized algorithm** uses coin flips to make decisions! There are three flavors of randomized algorithms. **Monte Carlo** algorithms run fast and produce the correct output with high probability. **Las Vegas** algorithms run fast with high probability, but correct output. **Atlantic City** algorithms run slow and may produce wrong results, but people use them sometimes for primality testing (we won't use these).

> "We get UUUUUGEEE speedups!"—debayan

18.2 Monte Carlo algorithms

These run fast and produce correct output with high probability.

Problem 18.1 (Matrix product verification)
Given two matrices $A, B, C \in \mathbb{P}_2^{n\times n}$, we want to verify if $C = AB$.

> "I don't remember what I did in high school. I do remember buying Age of Empires."—debayan

Naively, this takes $O(n^3)$. Strassens and other methods reduce the 3 slightly.

> "There are many disgusting methods. I recommend you look at some of these late at night if watching scary movies isn't enough."—debayan

Note that we only count multiplications here, since additions take much less time, so it's essentially free. We are looking for a $O(n^2)$ algorithm. If $AB = C$, then $\Pr\{\text{yes}\} = 1$. However, if $AB \neq C$, then $\Pr\{\text{yes}\} \leq 1/2$ (false positive).

Algorithm 18.2 (Frievalds' algorithm)
Select a random binary vector $r \in \mathbb{P}_2^n$. If $A(Br) = Cr$, output yes. Otherwise, output no.

Proof of correctness. True positives are trivially correct. Now let $D = AB - C \neq 0$. Non-zero terms are witnesses. For Dr, there exists a "good" r for every "bad" r, i.e. there exists a bijection from bad to good witnesses.

> **Question 18.3.** What are good and bad witnesses? Good witnesses preserve the non-zero values, and bad witnesses report 0s (when they should report 1s).

Now given a different $v \in \mathbb{P}_2^n$, we can construct a good witness $r' = r + v$ such that r' preserves all errors. Therefore, there is a 1:1 between right and wrong. □

Recall our median finding algorithm for finding ranks.

Algorithm 18.4 (Randomized select)

Given a list A and target rank i, find rank i. Select $x \in A$ as a pivot. Partition A into L and R at x, and x has rank k. If $i = k$, we are done. Otherwise, recurse on L or R.

In the worst case, we always pick the max or min, and this takes $O(n^2)$ time. However, if our guesses were simply 9/10-good, then our recursion tree has depth $\log_{10/9} n$. We can show that the expected length works. This is a Las Vegas algorithm.

Algorithm 18.5 (Quicksort)

Select a pivot $x \in A$. Partition around x. Sort L and R.

People use quicksort because it *works*. It's very fast in practice, though it's a Las Vegas algorithm.

18.3 Tail inequalities

> "Markov's inequality was invented by his teacher Chebyshev, and Chebyshev's inequality was invented by some French guy"—debayan

Theorem 18.6 (Markov's inequality)

If X is a non-negative random variable with a positive expected value, then for $c > 0$,

$$\Pr\{X \geq c\mathrm{E}[X]\} \leq 1/c.$$

Theorem 18.7 (Chernoff bounds)

Let Y be a sum of n a random Bernoulli variables. Then for $r \geq \epsilon$,

$$\Pr\{Y \geq \mathrm{E}[Y] + \epsilon\} \leq e^{-2\epsilon/n}$$

19 March 23, 2017

19.1 Random walks in graphs

Given a graph $G = (V, E)$, we start at some vertex s and at each time step, we randomly select an edge to traverse. We define P_v^t as the probability at step t of being at vertex v. For $t = 0$, we see that

$$P_v^0 = \begin{cases} 1 & v = s \\ 0 & \text{otherwise.} \end{cases} \tag{19.1}$$

For an arbitrary time t and vertex v,

$$P_v^t = \sum_{(u,v)\in E} \frac{P_u^t}{d(u)} \tag{19.2}$$

We let $p^t = (p_u^t), \forall u \in V$. We can also have a lazy random walk, such that

$$\hat{P}_v^{t+1} = P_{\text{lazy}} \hat{P}_v^t + (1 - P_{\text{lazy}}) \sum_{(u,v)\in E} \frac{\hat{P}_u^t}{d(u)} \tag{19.3}$$

Conceptually, we are introducing self-loops. Now we introduce pretty notation. Let A be an adjacency matrix where

$$A_{u,v} = \begin{cases} 1 & (v, u) \in E \\ 0 & \text{otherwise,} \end{cases}$$

and let D be a degree matrix where

$$D_{u,v} = \begin{cases} d(u) & u = v \\ 0 & \text{otherwise.} \end{cases}$$

Then we can define a walk matrix $W = AD^{-1}$, where

$$W_{u,v} = \begin{cases} \frac{1}{d(v)} & (v, u) \in E \\ 0 & \text{otherwise.} \end{cases} \tag{19.4}$$

This is the transition matrix of the Markov chain representing this graph.[12] Therefore, we can say that

$$p^{t+1} = Wp^t = W^{t+1}p^0 \tag{19.5}$$

For the lazy walk case, we can modify W as

$$\hat{W} = P_{\text{lazy}} I + (1 - P_{\text{lazy}})W. \tag{19.6}$$

A random walk has a stationary distribution if $W\pi = \pi$. For an undirected graph,

$$\pi_v = \frac{d(v)}{\sum_{u\in V} d(u)}. \tag{19.7}$$

[12] Note that we assume a uniform distribution on the transition probabilities to each edge, and that the graph is undirected.

Proposition 19.1

An undirected graph has a unique, stationary distribution if

1. the graph is connected, and
2. the graph is non-bipartite.

For directed graphs,

1. the graph is strongly-connected (irreducible), and
2. the graph is aperiodic.

Example 19.2 (Diffusion)

We have a drop of dye in water.

Depending on where the drop starts, the droplet will spread out, but eventually it will reach a stationary uniform distribution.

Example 19.3 (Cards)

We can shuffle a deck of cards in one of two ways: riffle shuffling (interleave half and half) or top-to-insert cards. For riffle shuffling, it takes 8-10 iterations to approach the stationary distribution, but for top-to-insert, it takes around 200. This is derived from eigen-decomposition.

Example 19.4 (Page rank)

In the past, search was a huge issue. How do we increase the relevancy of search results?

Google ignored the semantics and focused on the web graph.

1. $\textsc{CountRank} = \sum_{v \in V} A_{u,v}$ is the in-degree of v, or $A \cdot \mathbb{1}$, where $\mathbb{1}$ is a vector of 1s.
2. $\textsc{WeightRank} = \sum_{v \in V} A_{u,v} \cdot \frac{1}{d(v)}$, weighted by out-degree, or $W \cdot \mathbb{1}$, where $\mathbb{1}$ is a vector of 1s.
3. $\textsc{RecRank}(u) = \sum_{v \in V} A_{u,v} \cdot \frac{1}{d(v)} \textsc{RecRank}(v)$, which is the stationary distribution of W.
4. $\textsc{PageRank} = (1-\alpha) W \textsc{RecRank} + \alpha(\frac{1}{n}\mathbb{1})$, where $n = |V|$. We either RecRank or with some probability, jump to a random page. This step ensures a stationary distribution.

Google originally simulated this on a smaller toy graph, and used that (since the network graph is enormous).

20 April 4, 2017

> "Just go to a VC and scream 'blockchain' at them until they give you money"—debayan

20.1 Universal Hashing

Constant expected time is different from constant time with high probability. Today, we are dealing with the former. Consider an abstract data type, the dictionary problem.

Problem 20.1 (Dictionary problem)

There are three operations:

1. INSERT(ITEM) inserts a key-value pair, like a word and its definition.
2. DELETE(KEY) deletes a key from the dictionary.
3. SEARCH(KEY) finds a key and its value.

These can be done in expected $O(1)$ time, with $O(n)$ space.

We have a universe u of all possible keys of size m, n of which are present in the dictionary. We also have a hashing function $h : u \to k$. For a given dictionary, the load factor is $1 + \alpha = 1 + n/m$.

Definition 20.2 (Simple uniform hashing assumption)**.** For any two distinct keys k, k', $\Pr\{h(k) = h(k')\} = \frac{1}{m}$

Algorithm 20.3 (Universal hashing)

Instead of assuming that the input is random, we can hash randomly. Choose a random h from $\mathcal{H}$, a universal hash function family. In particular,

$$\Pr\{h(k) = h(k')\}_{h\in\mathcal{H}} \leq \frac{1}{m}, \forall k \neq k'$$

Then there are no assumptions of randomness for the keys.

Theorem 20.4

For n arbitrary keys and a random $h \in \mathcal{H}$,

$$\mathrm{E}[\#\text{ keys in slot}] \leq 1 + \frac{n}{m}.$$

Proof. Let $\mathbb{I}_{h(k_i)=h(k_j)}$ be an indicator variable. Then

$$\mathrm{E}[\#\text{ keys in slot}] = \mathrm{E}\left[\sum_{j=1}^{n} \mathbb{I}_{i,j}\right].$$

By linearity of expectation, this equals

$$\mathrm{E}\left[\mathbb{I}_{i,i}\right] + \sum_{i \neq j} \mathrm{E}\left[\mathbb{I}_{i,j}\right] \leq 1 + \frac{n}{m}$$

□

But do such families exist? Yes! A trivial example is the set of all hash functions. But that's quite useless in practice.

20.1.1 Dot product hash family

Let m be prime, and $u = m^r$, for $r \in \mathbb{Z}$. We have key $k = \langle k_0, k_1, \ldots, k_{r-1} \rangle$, with digits in base m, and a hash specification key $a = \langle a_0, a_1, \ldots, a_{r-1} \rangle$. Then

$$h_a(k) = a \cdot k \mod m.$$

We assume this may be computed in constant time, since we treat r as a constant.[13]

> "How many computer programmers does it take to change a light-bulb? None; it's a hardware problem."—debayan

Proof of universality. Without loss of generality, if $k \neq k'$, then $k_d \neq k'_d$ for some d.

$$\begin{aligned}
&\Pr\left\{h(k) = h(k')\right\}_a \\
&= \Pr\left\{\sum_{i=0}^{r-1} a_i k_i = \sum_{i=0}^{r-1} a_i k'_i\right\} \\
&= \Pr\left\{\sum i \neq d a_i(k_i - k'_i) + a_d(k_d - k'_d) = 0\right\} \\
&= \Pr\left\{a_d(k_d - k'_d)\right\} \cdot -\sum_{i \neq d} a_i(k_i - k'_i) \mod m
\end{aligned}$$

So we can multiply through by inverses and the above is equal to

$$\Pr\left\{a_d = f(k, k', a_{\neq d}) \mod m.\right\}$$

□

20.2 Perfect hashing

We can have polynomial build time, $O(1)$ worst case search time, and $O(n)$ space. Note that this is a *static* dictionary. Instead of using a linked list for the buckets, we can just use another hash table.

1. Pick h_1 from $\mathcal{H}$. This is the initial function to hash into large slots. Recall that m is prime and $m = \Theta(n)$. For each slot $j \in \{0, 1, \ldots, m-1\}$, l_j is the number of items in j (we know this in advance).

[13] Another (textbook) universal hashing function is $(ak + b) \mod p \mod m$, but we're not going over it.

2. If $\sum l_j^2 > cn$, that is, if some slots are too big, redo the previous step.
3. For h_2, reserve l_j^2 slots. There are *so many* slots that the probability of collision is at most $1/2$.[14]
4. If there's a collision in the second layer table, redo.

Using the union bound, we see that

$$\begin{aligned}
&\Pr\{h_{2,j}(k_i) = h_{2,j}(k_{i'})\} \\
&\leq \sum_{i \neq i'} \Pr\{h_{2,j}(k_i) = h_{2,j}(k_{i'})\} \\
&\leq \binom{l_j}{2} \frac{1}{l_j^2} \leq \frac{1}{2}
\end{aligned}$$

We can use Chernoff to show that none of the tables will be bigger than $\log n$. The expected number of redo operations is 2. With high probability, we will redo $\log n$ times, so creating the table should take around $O(n \log^2 n)$.

[14] This is similar to the birthday paradox!

21 April 6, 2017

21.1 Streaming

So far, we have not considered the space required for algorithms. Sometimes, we cannot fit the entire data on a disk. There may only be $\log n$ space, or even constant space.

Note, these are different from **online algorithms**, which need to produce partial output, and there are no space restrictions. In contrast, **streaming algorithms** have access to the entire input, but space is limited. Both algorithms may relax correctness.

Example 21.1 (Mean value)

We have a stream of inputs and want to find the mean. We can just keep a running total, which requires $2 \log n$ space, one register for the sum, and one for the number of elements seen.

Example 21.2 (Majority element)

There is a single element that is the majority. We keep a counter and an item. If counter is 0, we stick in the current element and increment the counter. If we see the same element, we increment the counter. If we see a different element, we decrement the counter.

"Thank you for that laugh. That is kindness, my friends."—debayan

21.2 Reservoir sampling

Suppose we have a stream, and we want a random sample of the stream.

Problem 21.3

We are given a stream $X = \{x_1, x_2, \dots, x_n\}$. We want to export a single x_i with probability $1/n$.

While seeing the inputs, we don't know n. For the first k elements, our probabilities for each element update as follows.

$$\begin{array}{lllll} k=1 & 1 & & & \\ k=2 & 1/2 & 1/2 & & \\ k=3 & 1/3 & 1/3 & 1/3 & \\ & \vdots & & & \\ k=k & 1/k & 1/k & \dots & 1/k \end{array}$$

We realize that to go from $1/2$ to $1/3$, we want to keep the $1/2$ with probability $2/3$. We can induct on this idea.

Problem 21.4 (k-sample)

Instead of exporting one element, we export k elements.

We keep the first k elements. For each element beyond k, keep the $i+1^{\text{th}}$ element with probability $k/(i+1)$ and discard a random element of the k.

21.3 Frequency moments

We have a stream of x's, $x_i \in \{1, 2, \ldots, m\}$. Let f_i be the number of times i appears.

Definition 21.5 (Frequency moment)**.** The p^{th} frequency moment

$$F_p = \sum_{i=1}^{m} f_i^p.$$

Assuming that $0^0 = 0$, then F_0 is the number of distinct elements, $F_1 = n$, and $F_2 = \sum f_i^2$. Calculating an exact F_0 is very hard, so given positive ϵ, δ, we will make a bounded guess

$$(1-\epsilon)F_0 \leq \hat{F_0} \leq (1+\epsilon)F_0$$

within $1-\delta$ of the true value.

Definition 21.6 (Pairwise-independent hash family)**.** Given $h \in \mathcal{H} : X \to Y$, then $\forall x_1, x_2 \in X, y_1, y_2 \in Y$,

$$\Pr\{h(x_1) = y_1 \wedge h(x_2) = y_2\} = \frac{1}{|Y|^2}.$$

Algorithm 21.7

Initialize $z = 0$. For each item j, if the number of trailing zeroes in the binary form of $h(j)$ is $> z$, then $z \leftarrow \text{zeroes}(h(j))$. Return 2^z.

22 April 7, 2017

Midterm 2 is coming up, next Thursday!

22.1 Tail inequalities

A **tail inequality** for a random variable bounds the probability that the variable will be far from its expectation. We apply tail inequalities to hash tables.

We have a hash table with n keys and n bins. Hash keys are assigned independently.

$$\mathrm{E}\left[\#\text{keys in bin}\right] = 1$$

However, this tells us nothing about the worst case. We would like the bound the spread. Let $X_b \in \{1, 2, \ldots, n\}$ be the number of balls that land in bin i.

22.1.1 Markov's inequality

We first use Markov's inequality. Recall that

$$\Pr\left\{X \geq a\right\} \leq \frac{\mathrm{E}\left[x\right] \to 1}{a} \leq \frac{1}{a}.$$

The union bound gives that

$$\Pr\left\{\max_b X_b \geq a\right\} \leq \sum_{b=1}^{n} \Pr\left\{X_b \geq a\right\} = \frac{n}{a}.$$

This is unhelpful. If all balls were in one bin, then the expectation would still be 1.

22.1.2 Chebyshev's inequality

We will now move onto Chebyshev's inequality. If X is a discrete random variable with expected value $\mathrm{E}\left[X\right]$ and variance $\operatorname{Var} x$, then

$$\Pr\left\{|X - \mathrm{E}\left[X\right]| \geq a\right\} \leq \frac{\operatorname{Var} X}{a^2}.$$

Let X_{b_i} be an indicator variable for whether ball b falls into bin i. These are distributed as Bernoulli random variables with probability $1/n$, so

$$\operatorname{Var} X_b = \operatorname{Var} \sum X_{b_i} = \sum \operatorname{Var} X_{b_i}.$$

Applied to our problem, $\operatorname{Var} X_b = 1 - 1/n \leq 1$. Therefore,

$$\Pr\left\{|X - 1| \geq a\right\} \leq \frac{1}{a^2}.$$

With the union bound,

$$\Pr\left\{\max_b X_b \geq a + 1\right\} \leq \frac{n}{a^2}.$$

We set $a = c\sqrt{n}$,

$$\Pr\left\{\max_b X_b \geq a + 1\right\} \leq \frac{n}{nc^2} = \frac{1}{c^2}$$

so with high probability, the most balls in each bin is $O(\sqrt{n})$.

22.1.3 Chernoff bound

Let $X_1, \ldots, X_n$ be mutually independent random variables, and let X_i be a Bernoulli random variable. Let $\mu = \mathrm{E}\left[\sum_i X_i\right]$. Then

$$\begin{cases} \Pr\left\{\sum_i X_i > (1+\beta)\mu\right\} < e^{-\beta^2\mu/3} & \text{if } 0 < \beta < 1 \\ \Pr\left\{\sum_i X_i > (1+\beta)\mu\right\} < e^{-\beta\mu/3} & \text{if } \beta > 1 \\ \Pr\left\{\sum_i X_i < (1-\beta)\mu\right\} < e^{-\beta^2\mu/2} & \text{if } 0 < \beta < 1 \end{cases}$$

We apply this bound to hash tables.

$$\Pr\left\{X_b > (1+\beta)\mathrm{E}\left[X_b\right]\right\} \le e^{-\beta\mu/3}$$

$$\Pr\left\{X_b > 1 + 3c\log n\right\} \le e^{-3c\log n/3} \le \frac{1}{n^2}$$

$$\Pr\left\{\max_b X_b > 1 + 3c\log n\right\} \le \frac{1}{n}$$

We set $\beta = 3c\log n$.

22.2 Streaming algorithms (graphs)

Let $G = (V, E)$, and set $n = |V|$. We generally always need $O(n)$ space. The stream is E. We define a **spanner** as the subgraph $H = (V, E_H)$ such that $E_H \subset E$ and

$$d_G(u,v) \le d_H(u,v) \le \alpha d_G(u,v),$$

where $d_G(u,v)$ is the shortest distance between u and v in G.

SPANNER
1 initialize $E_H \leftarrow \emptyset$
2 **for** $(u,v) \in E$
3 **if** $d_H(u,v) > \alpha$
4 add (u,v) to E_H
5 **return** E_H

Proof of correctness. We can decompose any path $u \rightsquigarrow v$ into

$$P_{u,v} = \{(u,p_1),(p_1,p_2),\ldots,(p_{n-1},v)\}.$$

In H, we then have

$$d_H(u,v) \le \sum_{(l,h)\in P} d_H(l,h).$$

If $(l,h) \in E_H$, then $d_H(l,h) = 1$, but if $(l,h) \notin E_H$, then $d_H(l,h) \le \alpha$. Thus,

$$d_H(u,v) \le \sum_{(l,h)\in P} \alpha = k\alpha \le \alpha d_G(u,v).$$

□

This takes $O(n)$ memory for the vertices and $O(|E_H|)$ for the edges. By combinatorial fun,

$$|E_H| \le 1 + n^{1+\frac{2}{\alpha+1}}.$$

By setting $\alpha > 2$, we can have $o(n^2)$ edges.

22.2.1 Sparsifier

We would like to preserve capacity across every cut. We blackbox a non-streaming algorithm that constructs a $1+\epsilon$ sparsifier (capacity within) with $\epsilon^2 n$ edges in $O(E)$ memory.

Claim 22.1. If we find a sparsifier H_1, H_2 for G_1, G_2, then $H_1 \cup H_2$ is also a sparsifier for $G_1 \cup G_2$.

Claim 22.2. If H_2 is a α-sparsifier of H_1 and H_1 is a β-sparsifier of G, then H_2 is a $\alpha\beta$-sparsifier of G.

We are given a dense graph, for which we create $t = O(n)$ chunks of $O(n)$ edges for E. We create a sparsifier for each of the t chunks. Since we cannot union all of them together, we can pairwise sparsify each of the H_i, divide and conquer style.

That results in a $\epsilon^{\log n}$-sparsifier for G. If we set $\gamma = 2 \log t$, then we have a $1+\gamma$-sparsifier that uses $\epsilon^2 n \log^2 n$ edges and memory.

23 April 11, 2017

23.1 Continuous optimization

Today we begin a 2-lecture series about continuous optimization.

Given objective function $f : \mathbb{R}^n \to \mathbb{R}$, find $x^* = \arg\min f(x)$ and $f(x^*) = \min f(x)$. If we need to find the maximum instead, we can minimize $-f$. To include constraints, we minimize the augmented function $g(x) = f(x) + \psi(x)$, where $\psi \to \infty$ as x approaches the constraints.

We assume that f is continuous and infinitely differentiable. Gradient descent is a locally greedy approach that can guarantee that the $f(x^{i+1}) \leq f(x^i)$.

This lecture bored me...a lot, so I left.

24 April 20, 2017

24.1 Gradient descent

Gradient descent is unconstrained minimization, given objective function $\rho : \mathbb{R}^n \to \mathbb{R}$. We want to find $x^* = \arg\min_{x \in \mathbb{R}^n} \rho(x)$, if x^* exists. The update step is

$$x^{(j+1)} = x^{(j)} - \eta \vec{\nabla} f(x^{(i)}).$$

We make a locally linear approximation for change in objective function,

$$\rho(x + \delta) = f(x) + \vec{\nabla} f(x)^T \delta + \frac{1}{2} \delta^T \vec{\nabla}^2 f(x) \delta + \dots$$

where higher order terms vanish. There are two outcomes. We either converge to critical points $\hat{x}$ or diverge to ∞. From basic calculus, $\hat{x}$ may either be a minima, maxima, or saddle point. However, if f is convex, that is

$$\forall_{x,y,\lambda \in \mathbb{R}} f(\lambda x + (1 - \lambda)) \leq 2f(x) + (1 - \lambda) f(y),$$

then our lives are easier.

Claim 24.1. If f is convex, then every critical point is a global minimum.

Definition 24.2. f is β-smooth iff

$$\forall_{x,\delta} \delta^T \vec{\nabla}^2 f(x) \delta \leq \beta \left| \delta \right|^2.$$

That is, the largest eigenvalue of the Hessian $\mathcal{H}$ is $\leq \beta$.

Definition 24.3. f is α-strongly convex iff

$$\forall_{x,\delta} \delta^T \vec{\nabla}^2 f(x) \delta \geq \alpha \left| \delta \right|^2, \alpha > 0.$$

That is, the smallest eigenvalue of the $\mathcal{H}$ is $\geq \alpha$. This means that we can upper and lower bound f with parabolas. If $h_{\tilde{x}} = f(\tilde{x}) + \vec{\nabla} f(\tilde{x})^T \delta + \alpha \left| \delta \right|^2$, then $h_{\tilde{x}} \leq f(x), \forall x, \tilde{x}$.

Theorem 24.4

If f is β-smooth and α-strongly convex, then

$$T = \Omega\left(\frac{\beta}{\alpha}\log\frac{f(x^0) - f(x^*)}{\epsilon}\right)$$

and $f(x^{(i)}) - f(x^*) \leq \epsilon$.

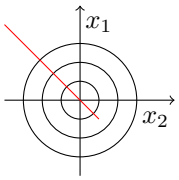
Example 24.5

Let $f(x) = \frac{1}{2}\alpha x_1^2 + \frac{1}{2}x_2^2$. How does it converge?

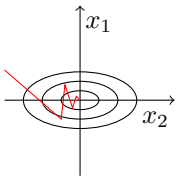
The gradient is $\vec{\nabla} f(x) = (a, 1)$ and the Hessian is

$$\vec{\nabla}^2 f(x) = \begin{pmatrix} a & 0 \\ 0 & 1 \end{pmatrix}.$$

When $\alpha = 1$, we have level curves.



When $\alpha \gg 1$, we have smushed ellipses.



24.2 Linear regression

Given points $x^{(1)} \dots x^{(m)} \in \mathbb{R}^n$ and labels $y^{(1)} \dots y^{(m)} \in \mathbb{R}$, we want to find for $\omega \in \mathbb{R}^n$,

$$h_\omega(x^{(j)}) = \sum_i \omega_i x_i^{(j)} = \omega^T x^{(j)} = y^{(i)}.$$

To find the best fit, we can use mean squared error,

$$L(\omega) = \frac{1}{n}\sum_{j=1}^{n} E_j(\omega)^2$$

MSE has a very nice gradient, which we can gradient descend on.

> "Clearly I have expensive tastes when it comes to blackboards. I want all of them"—madry

If $\mathcal{H}$ is positive semidefinite, then $L(\omega)$ is convex. In fact, if for $m \geq n$, $x^{(i)} \dots x^{(n)}$ are linearly independent, then $\mathcal{H}$ is positive definite, and L is α-strongly convex.

At w^*, $\vec{\nabla} L(w^*) = 0$ (normal equations for L), so $(X^T X)w^* = X^T y$ and we can find

$$w^* = (X^T X)^{-1} X^T y.$$

Analytically, this is beautiful! But in practice, finding inverses is $O(n^3)$, so it's better to gradient descend and find an approximation.

Example 24.6

Find $Ax^* = b$.

In linear algebra, they tell you to find inverses. In practice, that's awful. We want to find $f_A(x) = \frac{1}{2}x^T Ax - bx$, since $\vec{\nabla} f_A(s) = Ax - b$ When we set $\vec{\nabla} f_A = 0$, we recover the original linear system.

25 April 21, 2017

25.1 Gradient descent

Today, we review gradient descent—yet again! We're reviewing the lecture I walked out of.

Recall that the Hessian $\mathcal{H}$ is

$$\begin{pmatrix} \frac{\partial f^2(x)}{\partial x_1^2} & \cdots & \frac{\partial f^2(x)}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f^2(x)}{\partial x_n \partial x_1} & \cdots & \frac{\partial f^2(x)}{\partial x_n \partial x_n} \end{pmatrix}$$

At the bare minimum, $f(x)$ must be convex. There are a few definitions of convexity.

- $\forall x, y \in \mathbb{R}^n, 0 \leq \lambda \leq 1$, $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$. That is, if we draw a line between two points, the function always lies below.
- $\forall x, \delta \in \mathbb{R}^n$, $f(\lambda + \delta) \leq f(x) + \vec{\nabla} f(x)^T \delta$
- Finally, $\vec{\nabla}^2 f(x) = \mathcal{H}$ is positive semidefinite, $\forall \vec{x} \in \mathbb{R}^n$.

Convex functions can always be upper bounded by some parabola. Let $\delta^T \vec{\nabla}^2 f(x) \delta \leq \beta \left|\delta\right|^2$. Then we can pick the stereotypically beautiful step size of

$$\eta = \frac{1}{\beta}.$$

If $f(x)$ is strongly convex, then it will converge within a log bound.

26 April 27, 2017

26.1 Dynamic programming

Missed a lecture since I was sad and tired. Will fill in once the video's out. We continue dynamic programming.

1. Define subproblems. Think recursively.
2. Guess part of the answer, with relation to the problem. If there is only one step, then this is a greedy algorithm.
3. Combine optimal solutions to subproblems into the optimal solution for the current problem.
4. Recurse and memoize.
5. Solve the original problem.

Now consider the complexities for these problems. Prefix problems are $x[:i]$, suffix problems are $x[i:]$, both of which are $O(n)$. Substring problems are $x[i:j]$, which is $O(n^2)$, which is still polynomial.

Problem 26.1 (Edit distance)

Edit distance is a common problem, with applications from auto-correct to CRISPR. We are given two strings, x, y, and three operations:

- $\textsc{Insert}(c)$ inserts a character into a certain location,
- $\textsc{Delete}(c)$ deletes a character, and
- $\textsc{Replace}(c, c')$ replaces c with c'.

We are also provided with a list of costs per operation.

The subproblems here are suffixes, where

$$c(i,j) = \textsc{Edit-dist}(x[i:], y[j:]).$$

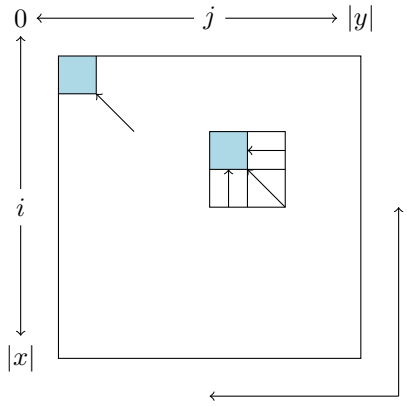
At any given time, the possible options are:

$$\textsc{Insert}(y[j]) \quad \textsc{Delete}(x[i]) \quad \textsc{Replace}(x[i], y[j])$$

Given these operations, the running time is

$$c(i,j) = \min \begin{cases} c(\textsc{Delete}(x[i]) + c(i+1,j)) \\ c(\textsc{Insert}(y[j]) + c(i,j+1)) \\ c(\textsc{Replace}(x[i], y[j]) + c(i+1,j+1)) \end{cases}$$

We can view these subproblems graphically.



As we see, this is a DAG towards the upper left corner.

Problem 26.2 (Knapsack)

We have a set of items, each with size s_i and value v_i. We are limited to a total size S. The goals is to choose a subset V as to maximize $\sum_{i \in V} v_i$ subject to $\sum_{i \in V} s_i \leq S$. This is actually NP-complete.

Let X represent the current size of the selected items.

$$DP([i, X) = \max \begin{cases} DP[i+1, X - s_i] + v_i \\ DP[i+1, X] \end{cases}$$

There are $S \cdot n$ possible subproblems, since there are S possible values for X and n items. This is $O(nS)$. However, the size of the input is $O(n \log S + n \log V + \log S)$, so the running time is exponential in terms of the input size.

26.2 Variations on polynomial time

Definition 26.3. If a problem's running time depends on the values of the input, then the problem is pseudo-polynomial.

Definition 26.4. If a problem has a running time like $2^{O(\log n)^c}$, then the running time will depend on c, and the problem is quasi-polynomial.

These generally occur when we are dealing with smaller versions of NP-complete or NP-hard problems. An example of quasi-polynomial is the Steiner tree.

Problem 26.5 (Steiner tree)

We are given an undirected graph $G = (V, E)$ with edge weights. Given a subset $A \subset T \subset V$, find a minimal tree spanning A.

27 May 2, 2017

27.1 Intractability

Today we'll start talking about intractability!

Example 27.1

- Shortest path in weighted graph has been beaten to death in poly-time. Longest path is NP-complete.
- Minimum spanning tree is fine. Traveling salesman is sad.
- Linear programming is okay. Integer programming is sad.
- 2d stable marriage is fine. 3d matching is sad.

If students haven't caught on, seemingly little changes in the problem cause seemingly drastic changes in tractability. There are generally two questions we ask of problems.

1. Are there poly-time algorithms for NP-hard problems?
2. Given a problem, can we tell if it's NP-hard?

Consider the MST problem.

- The MST-optimization problem exports either the minimum spanning tree or a statement that the graph is not connected.
- The MST-search problem is to find a spanning tree of weight $\leq k$. The output should be a tree, or k is too low, or the graph is not connected.
- The MST-decision problem is to determine whether such a tree exists.

If we can solve MST-search, then we can solve MST-decision, and if we can solve MST-optimization, then we can solve MST-search. If there is no poly-time solution to MST-decision, then there is no poly-time solution to MST-search or MST-optimization.

The solution of a poly-time search problem can lead to a poly-time optimization problem by binary search. If there is no such poly-time solution to search, then we're screwed for optimization too.

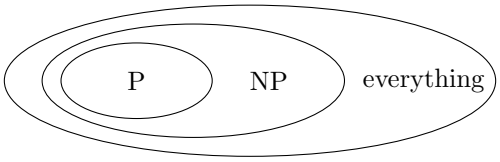
27.2 P vs. NP

Definition 27.2. A decision problem π is solvable in polynomial time if there exists a poly-time algorithm A such that $\forall x$, x is a "yes" or "no" to π implies that $A(x)$ outputs "yes" or "no," and vice-versa.

Definition 27.3. Formally, every "yes" instance exists a certificate of polynomial length that can be verified in polynomial time that it is indeed a "yes" instance.

There are three common misconceptions about P and NP.

1. NP doesn't stand for non-polynomial. Everyone thinks it does.
2. P and NP are not disjoint. Everyone thinks they are. P $\subseteq$ NP, since P problems can be verified in poly-time.
3. P $\subseteq$ NP $\subseteq$ all problems. NP isn't everything.



We can say $\pi_1 \leq_P \pi_2$ if π_2 is at least as hard as π_1.

Definition 27.4. A problem π is NP-hard if $\forall \pi' \in NP, \pi' \leq_P \pi$.

Definition 27.5. A problem π is NP-complete if π is NP-hard and $\pi \in NP$.

People joke that NP-completeness was the greatest export from computer science to normal people engineering.

Theorem 27.6 (Cook's theorem)

Imagine a circuit with three types of logic gates, `AND`, `OR`, and `NOT`. The circuit has binary inputs $\{x_1, x_2, \ldots\}$. cSAT is NP-complete.

First we show that cSAT is in NP. Given an assignment, we can plug in and check the circuit, taking $O(n)$ time total, $O(1)$ for each gate.

Now we show that cSAT is NP-hard. If we pick any problem $\pi \in$ NP, then $\pi \leq_p$ cSAT, so we reduce from a generic π to cSAT.

1. Let x be an input to π.
2. The reduction builds a circuit c_x that is satisfiable if and only if $\pi(x) =$ "yes." So $c_x(y)$ is an implementation of

No more notes for today. I'm bored af.

28 May 4, 2017

Final exam will be comprehensive but focus on the last third of material.

28.1 Reductions

We can reduce π_1 from π_2 if there is a special case of π_1 that is equivalent to π_2 (so formally, $\pi_1 \leq_P \pi_2$). Last time, we learned that circuit SAT is NP-complete. Subsequently, Karp also reduced lots of stuff from cSAT, including SAT.

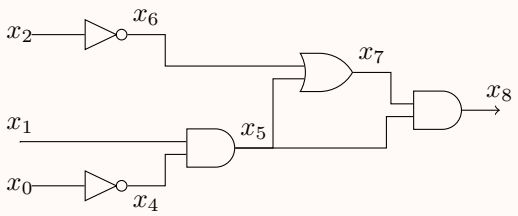
Problem 28.1 (SAT)

Given a boolean formula, is it satisfiable? There are n boolean variables and m boolean connections $(\vee, \wedge, \neg)$. Parentheses are taken into account.

Reducing SAT from cSAT initially poses some difficulties, since the circuit may have large fan-outs. Therefore, we require that each gate correctly evaluates its inputs. We show this reduction by example.

Example 28.2 (SAT from cSAT)

Consider the following circuit, for which the output $x_8 = 1$.



For this circuit to be satisfied, the following boolean expression must be satisfied.

$$\begin{aligned}\phi = x_8 \wedge (x_8 \leftrightarrow (x_5 \wedge x_7)) \wedge (x_7 \leftrightarrow (x_5 \vee x_6) \\ \wedge (x_6 \leftrightarrow \neg x_3) \wedge (x_5 \leftrightarrow (x_4 \wedge x_2)) \\ \wedge (x_4 \leftrightarrow \neg x_1).\end{aligned}$$

Thus, we can reduce any circuit to its appropriate formula in poly-time. If the circuit is satisfiable, then the formula is satisfiable, and vice versa.

Problem 28.3 (3-SAT)

Given a formula ϕ in conjunctive-normal form (CNF),[a] with 3 literals per clause, is ϕ satisfiable?

[a] CNF is an `AND` of `OR`s.

It is easy to show that 3-SAT $\leq_P$ SAT, since 3-SAT is a special case of SAT. However, we want to go the other way, and Karp showed that (refer to CLRS). So far, we have claimed that cSAT $\leq_p$ SAT $\leq_p$ 3-SAT.

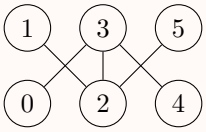
Problem 28.4 (Vertex cover (VC))

Given a graph $G = (V, E)$, find a subset $S \subseteq V$ such that $|S| \leq k$ and every edge $e \in E$ is incident on at least one vertex in S.

Vertex cover is NP—we just take the solution and check off all edges.

Example 28.5

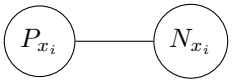
For example, consider the following graph.



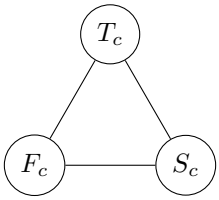
A possible vertex cover is $\{2, 3\}$.

We claim that VC is NP-complete.

Proof. We reduce vertex cover from 3-SAT by gadget construction. Each variable x_i corresponds to an edge of P_{x_i} and N_{x_i}, from which only one vertex is sufficient to cover the edge.



Each clause c corresponds to a triangle of F_c, S_c, T_c, or the first, second, and third variables. Two vertices are required to cover this triangle, and we soon realize that these two literals need not be satisfied, since only one literals needs be true for three `OR`s.



Now we relate variables to clauses. If the i^{th} literal of clause c corresponds to variable x_j, then connect F_{c_i} and x_j. If we find a VC for this graph, then we have satisfied the equations, and vice versa.

If a clause is satisfied, then at least one of its "relation" incident edges is covered by a variable. The remaining two incident edges are covered by selecting clause literal vertices. Covering an incident edge with a variable is equivalent to selecting that variable. □

This construction is clearer by example. [TODO]

Example 28.6

Let $\phi = xxx$

However, because cSAT is NP-complete,

$$\text{cSAT} \leq_P \text{SAT} \leq_P \text{3-SAT} \leq_P \text{VC} \leq_P \text{cSAT}$$

since cSAT is as hard as any of the other problems. Therefore, they are all equally hard!

29 May 9, 2017

29.1 Approximation algorithms

For the past 2 lectures, we've discussed NP-hard problems, and this semester, we've fallen in love with polynomial time solutions, which are also correct. Unfortunately, there are no exactly polynomial time algorithms for general NP-hard problems. Instead, we turn to approximation algorithms. An approximation algorithm runs in polynomial time and allows suboptimal solutions, but bounds sub-optimality.

When proving NP-hardness and NP-completeness, we often focused on decision problems. Does there exist an assignment such that...? Here, we focus on *optimization* versions instead.

Definition 29.1 (Approximation algorithm)**.** Say we have an optimization problem of size n, where c^* is the cost of the optimal solution and c is the cost of the approximation. Then we have a $\rho(n)$ approximation algorithm, where the ratio bound $\rho(n)$ is

$$\max\left(\frac{c}{c^*}, \frac{c^*}{c}\right) \leq \rho(n), \forall n.^{15}$$

An **approximation scheme** takes input $\epsilon > 0$ and provides a $1+\epsilon$ approximation. There are two flavors.

1. A polynomial time approximation scheme (PTAS) is polynomial in n, but not necessarily in $1/\epsilon$. For example, $O(n^{2/\epsilon})$.
2. A fully polynomial time algorithm (FPTAS), is polynomial in n and ϵ. For example, $O(n/\epsilon^2)$.

Problem 29.2 (Minimum vertex cover)

A vertex cover takes as input $G = (V, E)$ and outputs $S \subseteq V$ such that $\forall e = (u, v), S \wedge e \neq \emptyset$. The objective is to minimize $|S|$.

The class sugggests heuristics like highest-degree vertices, random vertices.

Algorithm 29.3 (2-approx. of vertex cover)

The professor suggests the following algorithm.

1. Randomly select an edge $(u, v) \in E$ and add both of its vertices to S.
2. Remove all edges incident to u or v from E.
3. Repeat until $E = \emptyset$.

The running time analysis depends on how we represent the graph in memory, but it is $O(|V| + |E|)$ using adjacency lists. This algorithm is non-deterministic;

[15] The first term represents minimization and the second represents maximization.

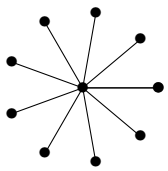
that is, the output depends on the input order. However, the output S is always a valid vertex cover, since edges are only removed when they are covered by a vertex in S. We claim that this is a 2-approximation.

Proof. Let A be the set of *edges* picked by the algorithm. No edges in A share an endpoint, since we removed all edges incident to u and v. Then

$$|S_A| = 2\,|A| \leq 2\,|S^*|$$

since the optimal vertex cover S^* must include at least one endpoint for each edge in A. □

Note that picking only on endpoint is not an improvement. Consider the following graph.



We could accidentally select all 9 radial nodes, where the optimal is obviously the single center node. Greedy selection (always pick edge with highest degree) does not improve the worst-case bound and it further complicates analysis, so we randomly select. Finally, post-processing also helps in practical settings, even though it doesn't affect the worst-case theoretical bounds.

Problem 29.4 (Set cover)

Given a set X of n points and m subsets such that

$$S_1 \cup S_2 \cup \cdots \cup S_m = X,$$

find a cover $C \subseteq \{1, \ldots, m\}$ such that $\cup_{i \in C} S_i = X$.

Example 29.5 (Vaccination)

Professor Tidor likes biology things. We want to cover as many epitopes (disease flavors) as possible, with a set of viruses, but can't stick every single one into the vaccine. Flu vaccines are manufactured far in advance, so selecting the specific strains of viruses is an important problem. There are good years and bad years for immunologists.

Algorithm 29.6

We use a greedy approach.

1. Choose a new set S_i with maximum number of uncovered points and add i to the cover C.
2. Mark elements from S_i as covered.
3. Repeat until done.

This is a valid cover since we don't end until every element has been marked, and we don't mark until we add a set that covers each element. However, the cover may not be optimal. This algorithm takes $O(\min(n, m))$ iterations, since at least one element and its set is removed per iteration. Each iteration takes $O(mn)$, so the running time is $O(mn \min(m, n))$. We show that this is a $1+\log n$ approximation.

Proof. Let $t = |C^*|$ be the size of the optimal cover. By picking the set with the most uncovered elements, we will remove at least a large fraction $(1/t)$ elements each round.

Let X_i be the set of uncovered elements at round i. Then X_i can always be covered by t or fewer sets, since t sets can cover everything (in theory). For that to be true, then there exists a set that covers at least $|X_i|/t$ elements. Then

$$\begin{aligned}
\forall i, |X_{i+1}| &\leq (1 - \frac{1}{t}) |X_i| \\
|X_i| &\leq (1 - \frac{1}{t})^i |X| \\
&= (1 - \frac{1}{t})^{t \log n} n \leq e^{-\log n} n \equiv 1
\end{aligned}$$

That is, the algorithm terminates at step $t \log nn$, so X_i is the trivial case. Since $|C^*| = t$ and $|C| \leq t \log n + 1$, the algorithm is a $1 + \log n$ approximation. □

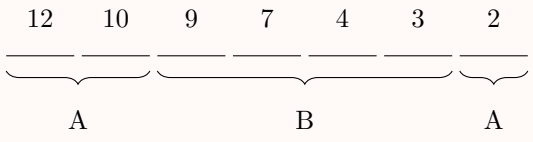
Problem 29.7 (Partition)

Given a sorted list S of n positive numbers, find a partition of indices into sets A, B such that A, B cover S and $\max(\sum_{i \in A} S_i, \sum_{i \in B} s_i)$ is minimized.

This is an optimization problem seeking the most balanced partition (equal sets, or closest to).

Example 29.8

Consider the following set of numbers, $\{12, 10, 9, 7, 4, 3, 2\}$. The optimal partition would be $A = \{12, 10, 2\}$ and $B = \{9, 7, 4, 3\}$.



30 May 11, 2017

30.1 Weakly NP-hard problems

Recall from last lecture that we had 3 goals: solve hard problems, using a fast (poly-time) algorithm, which find exact answers. For most of this class, we focused on the latter two goals.

> "It's like activities, academics, and sleep at MIT. You pick two. But that's wrong. You don't need sleep anyways."—madry

Today, we focus on **weakly NP-hard problems**, which is polynomial in n, but depends on the size of inputs. In particular, consider a problem parametrized by (x, k), for which there exist algorithms that are polynomial in n, but depend on some unconstrained $f(k)$.

Problem 30.1 (k-vertex cover)

Given a graph $G = (V, E), k \geq 0$, is there a set S of at most size $\leq k$ that "covers" all edges?

Let k be the parameter and $n = |V| + |E|$, where $k \ll n$ Naively, we could use a brute-force approach: enumerate all subsets. There are

$$\binom{|V|}{k} + \binom{|V|}{k-1} + \cdots + |V|$$

subsets, which is approximately $|V|^k$. Each step takes $O(|E|)$, so the total running time takes $O(|E|\,|V|^k)$.

30.2 Fixed-parameter tractability

A problem is **fixed-parameter tractable (FPT)** if an algorithm can solve it with running time $f(k)n^{O(1)}$.

Algorithm 30.2 (Bounded search-tree)

We propose the following algorithm for solving k-vertex cover instead.

1. Pick an edge $(u, v) \in E$.
2. Try to add u to S. Delete u and incident edges to u from G. Recurse with $k' = k - 1$.
3. Try v and repeat.

The base case is $k = 0$. There is a k-vertex cover if $E = \emptyset$.

There are 2^k nodes and each call takes $O(|V|)$ time. The running time of this algorithm is $2^k O(|V|)$.

Note 30.3. Why didn't we define FPT as $f(k) + n^{O(1)}$? These two definitions are actually equivalent. For example, $2^k n$ is at most $O(4^k + n^2)$. The intuition behind this equivalence is to preprocess the graph into a smaller, but harder problem.

30.3 Kernelization

A problem is **kernelizable** if we have a poly-time algorithm that converts an input instance (x, k) into a small and equivalent input (x', k'). Here, we say "small" means $|x'| \leq f(k)$, and "equivalent" means that the answer to (x, k) is equal to the answer to (x', k').

Theorem 30.4

A problem is fixed-parameter tractable if and only if there is a kernelization.

Proof. Kernelization trivially leads to a FPT algorithm. Kernelize and solve.

The converse is more interesting. Let A be a hypothetical algorithm that runs in $f(k)n^c$. If $n \leq f(k)$, then the instance (x, k) is a kernel and we are done. Otherwise, if $n > f(k)$, then we can solve the instance using A and the trivial instance is the kernel. □

The real goal is to find a kernel that's polynomial in k. We show that this is indeed possible for k-vertex cover.

Algorithm 30.5 (Quadratic kernel for k-vertex cover)

There are several observations.

1. If there is a self loop, then that vertex must be in the cover, so we add the vertex and delete the loop.
2. If there are many duplicate edge, we collapse them into a single edge.
3. If a vertex has degree $> k$, then it has to be in the cover. Otherwise, we'd need all its k neighbors at least.
4. Remove isolated vertices.

If there are over k^2 remaining edges, then there is no k-vertex cover. We end up with a graph that is simple and has all degrees at most k. If G^* has more than $k^2 n$ edges, then G^* does not have a k-vertex cover, because each vertex covers at most k edges, and we can cover at most $k^2 n$ edges.

To reach this kernel, we require a running time of $O(|V|\,|E|)$ trivially, and $O(|V| + |E|)$ with some work. Now, the brute-force solution costs $O(|V| + |E| + 2^k k^{2k+2})$ time, and the bounded search-tree solution costs $O(|V| + |E| + 2^k k^2)$ time. As some extra juicy knowledge, there's even a linear kernel! Currently, the best parametrized vertex cover is $O(kV + (1.274)^k)$.

Theorem 30.6

If we have an efficient poly-time approximation scheme (EPTAS) for a problem, then it is FPT in $O(f(1/\epsilon)n^{O(1)})$.

We can relate kernels to approximation algorithms, which are parametrized by ϵ. The contrapositive is often used to rule out the existence of an EPTAS for some problems. That is, if it is not a FPT, then there is no EPTAS.

31 May 16, 2017

31.1 Distributed algorithms

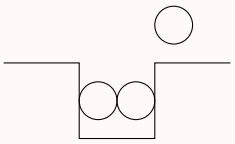
The final exam is on Monday!

> "Three hour exam! We want to go for the full experience. I was actually thinking of four."—madry

So far, the class has "lied" to us. We assumed that everything ran on a single computational unit. In the real world, this is rarely the case. There could be many players, but we may not always receive n times the gain for n times the players.

Example 31.1

We're digging a deep hole, and more people won't help.

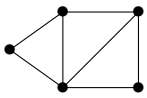


But if we dig a shallow trench, more people are good!

One model is **parallel computing**, in which there are multiple processors working at the same time.

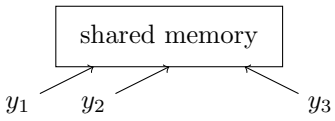
Another model is **distributed computing**, in which we have many processors over a network. The setup is as follows: we have n processors (players), each of which has input x_I and needs to compute $y_i = f_i(x_1, \ldots, x_n)$. There are two prevailing models.

1. The **message passing** model has a network of nodes, connected somehow.

   Each node can communicate with its neighbors via messages, and labels are all relative.

2. The **shared memory** model allows nodes to communicate via a shared piece of memory, which can be written to and read from.

31.2 Leader election

A common issue in distributed computing is leader election.

Problem 31.2 (Leader election)
There are n processors, but exactly one outputs "I am the leader."

This is impossible if we require a deterministic result and the processors are truly identical.

Proof. Imagine that we have two processors, exactly identical. They send identical messages forever, and one of them has to say "I am the leader," but they always send the same messages, so there cannot be a unique leader. □

The first solution is to assume that each processor i has a unique ID (for example, MAC addresses).

Algorithm 31.3

1. Each processor i has a local variable $\max_i$, initialized to its own ID.
2. At each round, it sends $\max_i$ to each neighbor and upon receiving a message, updates its $\max_i$
3. If at the end, a processor's ID matches the max, output "I am the leader."

The second solution is to use randomness, and we do not require uniqueness.

Algorithm 31.4

1. Try to "manufacture" unique IDs for each processor by producing $\text{ID}_i \leftarrow \{1, \dots, k\}$.
2. Run previous algorithm.

For fixed $i \neq j, \Pr\{\text{ID}_i = \text{ID}_j\} = 1/k$. We use the union bound over all $\binom{n}{2}$ pairs, so

$$\Pr\{\text{collisions}\} \leq \sum_{i \neq j} \Pr\{\text{ID}_i = \text{ID}_j\} \leq \binom{n}{2}\frac{1}{k} \leq \epsilon$$

where we require $k \geq \frac{1}{\epsilon}\binom{n}{2}$. This is a Monte Carlo algorithm, correct with some probability $O(\delta)$. But for some critical applications, we'd like an exact algorithm. We can detect and repeat if repeats. In expectation, we require $1/(1-\epsilon)$ repeats. The expectation time to succeed, given probability p of success, is

$$\sum_i^\infty (1-p)^i p = \frac{1}{p}.$$

31.3 Maximal independent set

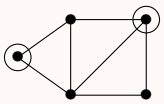
"Maximal" means we cannot increase the size, while "maximum" means the largest. We are looking for the former.

Problem 31.5 (Maximal independent set (MIS))

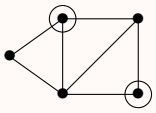
Find a protocol such that each processor outputs `YES` or `NO`, where the `YES` processors (optimists) form a maximal independent set.

Example 31.6

We find a maximal independent set on the following graph.



We find another MIS on the same graph.



We use leader election to solve this problem.

Algorithm 31.7

1. Until done, elect a leader among active processors.
2. Leader outputs `YES` and tells its neighbors to output `NO`.
3. All processors that have decided become inactive.

However, the running complexity is sad. We could have a worst case of a path, in which the algorithm takes $O(\delta n)$.

Algorithm 31.8 (Luby's randomized MIS algorithm)

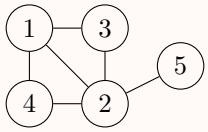
We do not require unique IDs, but require randomization. The protocol proceeds in phases, each of which has two rounds.

1. Assume that $n \in \{1, \dots, k\}$ and broadcast. If received values $< n$ (you are uniquely maximum), then join the MIS and output `YES`
2. If joined MIS, announce to neighbors. Each processor that receives this message outputs `NO`.

This algorithm takes advantage of local maxima.

Example 31.9

Consider the following graph.



In the first round, $\{3, 4, 5\}$ join the MIS and output `YES`. Then all its neighbors output `NO`.

We claim that this algorithm finishes in $O(4 \log n)$ phases with high probability. The proof in the notes only applies to an annelid.[16] In class we prove by picture.

The key claim is that $\Pr\{\text{becoming inactive}\} \geq 1/2$. Left as exercise to reader.

[16] line graph lol