Complexity & Advanced Algorithms

Siddharth Bhat

Contents

Chapter 1

NLogSpace-completeness

1.1 Co-NLogSpace

$L \in$ Co-NLogSpace $\equiv L^c \in$ NLogSpace. That is, complement the language L. if L^c is in NLogSpace, then $L \in$ Co-NLogSpace.

We intuitively believe that NP $\neq$ Co-NP. However, we can show that NLogSpace = Co-NLogSpace.

$$\text{PATH} = \{\langle G, u, v\rangle \mid \textbf{exists} \text{ path between vertices } (u, v)\}$$
$$\overline{\text{PATH}} = \{\langle G, u, v\rangle \mid \textbf{no} \text{ path between vertices } (u, v)\}$$

We assume that $\overline{\text{PATH}}$is co-NL-Complete.

If we show that $\overline{\text{PATH}}$is in NLogSpace, then every problem in co-NLwill be in NL

1.1.1 Solving $\overline{\text{PATH}}$in NL

$$V_R \equiv \{\text{set of vertices reachable from } u\}$$
$$V_{NR} \equiv \{\text{set of vertices } \textbf{not} \text{ reachable from } u\}$$

Sid confusion, why can't we use PATH as a subroutine: When we have an NDTM, we cannot *observe that the NDTM returns a* 0. We can *observe if an NDTM succeeds*, but there are weird paths and exponential number of paths where the NDTM does not return a 0? But if this is true, then how is PATH NL-complete? I am very confused.

To represent V_R and V_{NR}, we use 1 bit per vertex (since V_R and V_{NR} are disjoint), so total space is V.

Assume we know $|V_R|$. In this case, we can check whether v is unreachable from u — Enumerate all vertices. If they are reachable from u, bump up a counter. If we don't hit v till the counter gets to $|V_R|$, then what we know that is v is unreachable.

However, if v were reachable from u, then as we enumerate, we would find v as we were going through all vertices (we would not hit V_R unless we visit v).

This is important, because in an NDTM, if *any* of the paths accept, then we accept.

$$V_R = \cup_i V_R(i)$$
$$V_R(0) = \{u\}$$

to compute $cur \in_? V_R(i+1)$, first **recompute** that $pred \in V_R(i)$, and then check that $(cur, pred) \in E(G)$. We cannot **store** $V_R(i)$, since we don't have enough space.

eventually we will reach $V_R(|V|)$, where we stop.

We can compute $|V_R| = \sum_i |V_R(i)|$. We compute $|V_R(i)|$ by checking over each vertex it's membership into $V_R(i)$. And if it does, we bump up our counter.

Reference: Read `Sipser-Chapter 8`

```
def belongs(G, i, startv, endv, curv):
    """Check if curv belongs to V_R(i)"""
    if i == 1:
        return startv == curv
    else:
        # log(V)
        for pred in G.vertices:
            # This can use a modified version of PATH that stores lengths?
            if small_belongs(G, i - 1, startv, endv, pred):
                if isneighbour(pred, curv):
                    return True

        return False

def countcard(G, startv, endv):
    """ Count the cardinality of V_R"""
    card = 0
    # log(V)
    for i in len(G.vertices):
        # this is also log(V)
        for curv in G.vertices:
            if small_belongs(G, i, startv, endv, curv):
                card += 1
    return 1
```

1.2 Oracles

For all inputs w of length $|w| = n$, there exists a **single** advice (a_n is allowed to be a single string that is polynomial in n). So, $a : \mathbb{N} \to \Sigma^*$, and the advice of a given input w is $a(|w|)$.

1.2.1 $\mathtt{P}^{\mathtt{poly}}$

$L \in \mathtt{P}^{\mathtt{poly}}$ if there is a polynomial time turing machine M which takes two inputs — a string $x \in \Sigma^*$, and an advice $a_n \in \Sigma^*$, such that for all inputs w such that $|w| = n$, then there exists a polynomial $p(n)$ with $|a_n| \leq p(|w|)$.

We force it to be polynomial in the word-length, because things like a lookup table take exponential space in the word-length (number of strings of length n is 2^n).

We can see that the advice is somewhat "hardwired" into the machine given the input length (since $a : \mathbb{N} \to \Sigma^*$). So, we have a sequence of machines $M_i : \mathbb{N} \to \{\text{Turing machines}\}$, and we instantiate the machine $M_{|w|}$ to check if $|w| \in L$.

NP is allowed to have a *varying witness*, while $\mathtt{P}^{\mathtt{poly}}$will have the *same* advice.

We don't even need to know if the advice string should be able to be found in polynomial time.

1.2.2 $\mathtt{P}^{\mathtt{poly}}$contains non-recursive languages

Chapter 2

Advice & Time Hierarchies

2.1 $\mathtt{P}^{\mathtt{poly}}$

This class could possibly be bigger than P.

In NP , witnesses are different for each string. In $\mathtt{P}^{\mathtt{poly}}$, witnesses are fixed for strings of a given length.

The advice string need to even be found in polynomial time!

Recursive language: Halts on all inputs with yes/no Recursively enumerable: Halts and returns yes on inputs which belong to the language. On inputs that do not halt, undefined behavior.

2.2 Unary language that is non-recursive

L is a unary language $\equiv L \subseteq 1^*$

Theorem 1 *Every unary language is decidable by* P^{poly}

Proof. let L be a unary language.

Since the only characteristic of a string in a unary language is its length, for any given length, there is *at most one string of that length* in L. So, we can index the set L by the string lengths! Hence, the advice function allows us to build up a lookup table for *any* unary language.

We construct the advice function $a_L : \mathbb{N} \to \{0, 1\}$ be such that $a_L(n) =$ does 1^n belong to L?. Now, let M decide L as follows: $M(str) = a(|str|)$. Since we don't need to build a (it's an oracle we take for granted, the proof is done).

Theorem 2 P^{poly}*contains non-recursive languages.*

Proof. Let $L_{nr} \subset \{0, 1\}^*$ be a nonrecursive language. We define $L_w = \{1^{\#w} \mid w \in L_{nr}\}$, which is a unary language. A string $1^k \in L_w$ acts as a witness for the existence of some string $w \in L_{nr}$ as the lex-ordering-position of the string w.

Example of # evaluated on some strings
$\#0 \to 0$
$\#1 \to 1$
$\#00 \to 3$
$\#01 \to 4$
$\#100 \to 5$
...

L_{nr} has now been reduced to L_u, since the mapping with # is a *bijection* . Also, L_u can be decided by $\mathtt{P}^{\mathtt{poly}}$. Hence, L_u can decide nonrecursive languages.

Question: Is the set $\{0,1\}^*$ countable? It doesn't feel like it is!

2.2.1 Sparse language

A **sparse language** is one where the number of strings of length n is bounded by a polynomial. $|L \cap \{0,1\}^n| \leq p(n)$.

Idle thought: Is there a classification theorem for sparse languages? "sparse-complete"

We study the relationship between NP and $\mathtt{P}^{\mathtt{poly}}$, using sparse languages.

2.2.2 Cook reduction

A language L_1 cook reduces to a language L_2 if there is a polynomial-time turing machine M_{L_1} that recognizes L_1 given oracle access to L_2.

The machine M_{L_1} Can query membership to L_2 multiple times (polynomial) before deciding if a string $w \in_? L_1$.

Lemma 1 *If L_1 Cook-reduces to L_2 and $L_2 \in P$, then $L_1 \in P$.*

Proof. L_1 is decided by a polynomial-time turing machine M_{L_1}, so it can make at most polynomial queries to L_2. Since $L_2 \in P$, There exists a polynomial-time turing machine M_{L_2} which solves the membership query.

The total running time for M_{L_1} is in P, so it can make at most polynomial queries to M_{L_2}. Hence, M_{L_1} can simulate M_{L_2} and solve the membership problem.

Theorem 3 *Every language $L \in \mathtt{NP}$ is Cook-reducible to a sparse language iff $\mathtt{NP} \subseteq \mathtt{P}^{\mathtt{poly}}$.*

This theorem is significant because we strongly believe that no NP -complete language is sparse! So, we believe that $\mathtt{NP} \not\subset \mathtt{P}^{\mathtt{poly}}$.

Since SAT is NP -complete, we simply need to show that SAT is cook-reducible to a sparse language iff $\mathtt{NP} \subseteq \mathtt{P}^{\mathtt{poly}}$.

We will exhibit polynomial-time advice string for all inputs of a given length, to use the power of $\mathtt{P}^{\mathtt{poly}}$.

Proof. **(Forward) SAT Cook-reducible to a sparse language $L \implies SAT \in \mathtt{P}^{\mathtt{poly}}$**

There is a polynomial-time machine M which can solve SAT given oracle access to sparse language L.

We want to show that SAT is in $P^p oly$.

Let M run in time $p(n)$ on inputs of length n

The advice string $a(n)$ we want to give is the oracle behaviour on sparse language L. Since the machine M can ask for string of length at most $p(n)$.

Since the language is sparse, the set of all strings of a given length in L is polynomial. So, $a(n) = concat(\{w \in L \mid |w| \leq p(n)\})$ where *concat* concatenates all the strings. $a(n)$ will be polynomial in length since the length of each string w is bounded by $p(n)$. Let $sparse(n)$ be the polynomial that controls the sparsity of L for any string n. That is, for any length i, the language L contains at most $sparse(i)$ strings.

The total number of strings in $a(n)$ will be $N = \sum_{i=0}^{p(n)} sparse(i)$, which is a polynomial in n. Hence, $a(n)$ is a legal advice string.

We're done here, we converted oracle access to a sparse language into a polynomial advice string.

(Backward) $SAT \in \texttt{Ppoly} \implies$ **SAT Cook-reducible to a sparse language** L

We are given a machine M_{sat} which seeks advice $a(n) : \mathbb{N} \to \{0,1\}^*$. The machine M_{sat} runs for polynomial time $p_sat(n)$.

We need to construct a sparse language L_{sparse}, such that given oracle access to L_{sparse}, we can solve SAT using a new machine M'.

Consider all strings that are queried by M_{sat} to M_{poly}. For an input of length n, the machine M_sat can query a $p_{sat}(n)$ times at maximum. Hence, we the language consisting of the subset of a that is sampled by M_{sat} is a sparse language. Given access to this language, we can substitute the function a with the sparse language which contains all advice accessed from a.

Chapter 3

Gaps in space and time

We wish to study what is not computable given some resource. If there resource is time, we want to understand what can be solved in $t(n)$ but not in smaller than $t(n)$ — in the sense of $o(t(n))$.

We can try to construct a hierarchy of problems that can be solved given increasing time.

$$f(n) \in o(g(n)) \equiv \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$
$$f(n) \in O(g(n)) \equiv \lim_{n\to\infty} \frac{f(n)}{g(n)} \in O(1)$$

3.1 Space Hierarchy

A function $f : \mathbb{N} \to \mathbb{N}$ is said to be **space constructible** if there exists a turing machine that on input 1^n, it computes $f(n)$ using space $O(f(n))$. So the output can be $1^{f(n)}$ say, since that uses space $O(f(n))$.

Most common functions such as polynomials, exponentials, and logarithms are all space constructible.

Theorem 4 *Let f be a space-constructible function. There exists a language L which can be decided in $O(f(n))$ space, but not in $o(f(n))$ space.*

Proof. The proof is to **construct** a language which can be decided on $O(f(n))$ space, but not in $o(f(n))$ space. Such a language tends to be artificial due to the construction having to work *for all f*.

We need two properties for this language L we create:

- It is **not decidable** in $o(f(n))$ space.
- It **is** decidable in $O(f(n))$ space.

We will use diagonalization to show an construct an L that **cannot be decided** in $o(f(n))$ space. List each TM that runs in $o(f(n))$ space. This collection of all TMs (viewed as strings) is written as:

$$ALLTM = \cup_{i=0}^{\infty}\{0,1\}^i$$

We will define a language L which cannot be decided by **any** TM on the above list.

We will create a matrix of the form $DECIDE(i,j) = M_i(\langle M_j \rangle)$. That is, we feed M_i the string of M_j.($\langle M_j \rangle$ interprets the machine M_j as a string).

Now, create a language L:

$$L \equiv \{M \mid M(\langle M \rangle) = 0\}$$

Note that L is **not decidable** in $o(f(n))$ space. Proof by contradtiction: Assume such a machine M_c (c for contradiction) exists. We now ask if $\langle M_c \rangle \in L$?

- If $\langle M_c \rangle \in L$, then $M_c(\langle M_c \rangle) = 0$ (by the definiton of L). But since M_c **decides** L, $M_c(\langle M_c \rangle) = 0 \implies \langle M_c \rangle \notin L$. **Contradiction**.
- On the other hand, say that $\langle M_c \rangle \notin L$, then $M_c(\langle M_c \rangle) = 1$ (by the definition of L). But since M_c **decides** L, $M_c(\langle M_c \rangle) = 1 \implies \langle M_c \rangle \in L$. **Contradiction**.

We now move to show that L **can be decided** in $O(f(n))$ space. Consider a machine `INTERPRET` that does this:

```
def INTERPRET(w):
    Mw = convert_to_TM(w)

    # Naive solution: Try to run Mw, see what happens.
    # flag = Mw.run(w)

    # Problem 1: How do we know it runs in o(f(n)) space?
    # flag = Mw.run_with_bounded_space(w, space_bound=f(n))

    # Problem 2: How do we know that Mw halts?
    # Count the size of the config. space, and reject if Mw
    # takes more steps than the configuration space size.
    flag = Mw.run_wth_bounded_space_and_steps(w, space_bound=f(n),
                                              steps_bound=Mw.config_space_size())


    return !flag
```

For more details, read Sipser chapter 9

Corollary 1 *For two functions $f1, f2 : \mathbb{N} \to \mathbb{N}$, if $f1 \in o(f2)$, then $DPSACE\,(f1) \subsetneq DPSACE\,(f2)$. (**Sid note: we do not need the condition that $f1 \neq f2$ thanks to the fact that in $o(n)$, the limit tends to** 0)*

3.2 Time Hierarchy

Theorem 5 *Let f be a time-constructible function. There exists a language L which can be decided in $O(f(n))$ time, but not in $o\left(\frac{f(n)}{\log(f(n))}\right)$ time.*

Proof. Proof is the same as that of space hierarchy (roughly).

We get the log factor for us to simulate a $f(n)$ time turing machine. We do not know how to perform the simulation with constant overhead.

Corollary 2 $P \subsetneq EXPTIME$

3.3 Polynomial Hierarchy

One interesting thing to study is the power of oracles (non-uniform computations). One can try to study the nature of languages, given oracle access.

Definition 1 *Let M be a turing machine, A be a language. **The language** $L(M^A)$ is the set of strings accepted by the machine M with oracle access to A.*

We can deneralize this by giving access to a *class of languages*!

Definition 2 *Let M be a turing machine, C be a class of languages. **The language** $L(M^C)$ is the set of strings accepted by the machine M with oracle access to any language in C.*

$$M^C = \{L(M^A) \mid A \in C\}$$

Definition 3 *Let C_1, C_2 be classes of languages. **The language** $L(C_1^{C_2})$ is the set of strings accepted by some machine in C_1 given oracle access to some machine in C_2.*

$$C_1^{C_2} = \{L(M_1^{M_2}) \mid M_1 \in C_1, M_2 \in C_2\}$$

We will use M^ϕ to denote oracle access to an "empty" oracle. Hence, $M^\phi \sim M$.

An example would be $\texttt{co-NP} \subset \texttt{P}^{\texttt{NP}}$, because the `P` oracle can flip the answer of the `NP` oracle.

Chapter 4

Polynomial Hierarchy

Chapter 5

Probabilistic proofs

5.1 IP — interactive proofs

Definition 4 *Completeness: For every true assertion, there is a valid proof.*

Definition 5 *Soundness: For every false assertion, no valid proof exists.*

A good proof system must also be such that the verifier is efficient (that is, polynomial time).

If we ask that a proof system must be sound and complete, there is no scope for error! Further, it is not clear if the verifier and the prover can "talk" to each other. If we choose to allow interactions, what are the implications?

We relax the assumptions this way — Relaxed compleness states that for every true assertion, there is a proof strategy that will convince the verifier with probability at least $> \frac{2}{3}$. Similarly, relaxed soundness states that for every false assertion, every proof strategy fails to convinve the verifier with probability at least $> \frac{2}{3}$.

The formalization is as follows:

Definition 6 *Interactive proof systems*

- *An interactive proof system for a language L consists of two entities: a prover P and a verifier V. P and V share common input, and work for $R \in \mathbb{N}$ rounds.*
- *In each round, the prover can send the verifier a message that is polynomial in the length of the input.*
- *The verifier can send a polynomial length reply to the prover.*
- *The verifier is a randomized polynomial time turing machine. Time is measured as a function of the length of the input.*
- ***Completeness**: $\forall x \in L$, there exists a prover strategy so that the verifier accepts with probability $> \frac{2}{3}$.*
- ***Soundness**: $\forall x \notin L$, any prover strategy will lead the verifier to accept with probability $< \frac{1}{3}$.*

Note that the power of the prover in unspecified in this definition — we are implicitly saying that finding a proof is generally much harder than verifying a proof. Hence, the prover has no real bounds on the power, while the verifier does.

We also have the value $R \in \mathbb{N}$, which lets us setup the number of rounds. This is a knob we can twiddle, that allows us to change the hardness of the problem.

Definition 7 *The IP hiearchy: Let $r : \mathbb{N} \to \mathbb{N}$ be the "number of rounds" function. Define $IP(r)$ to be the set of languages such that there exists an interactive proof system using at most $r(|x|)$ rounds on input x.*

For a class of functions $R \subset \{\mathbb{N} \to \mathbb{N}\}$, we can then define $\texttt{IP}(R) = \cup_{r \in R} \texttt{IP}(r)$.

Note that NP $\subset$ IP. Also, the number of rounds cannot be more than polynomial — the verifier is poly bounded in time, so the verifier cannot work more than poly rounds. So, we denote $\texttt{IP} \equiv \texttt{IP}(O(poly(n))$.

Both **randomness** and **interaction** are essential to the definition.

When randomness is removed but only interaction is present, this will be like NP . The prover can arrive by itself the set of messages the verifier would send to the prover.

When interaction is removed but randomness is remained, the verifier is similar to that of NP , but the verifier can now be **probabilistic**. This class of languages is likely beyond NP .

5.2 Graph non-isomorphism (GNI)

Two graphs G, H are isomorphic (denoted $G \sim H$), iff there exists a bijection such that $\forall x, y \in V_1, (x, y) \in E_1 \implies (f(x), f(y)) \in E_2$.

Using this, we define GNI , the problem of checking if two graphs are not isomorphic:

$$\texttt{GNI} \equiv \{\langle G, G' \rangle \mid G \nsim G'\}$$

Graph isomoprhism is in NP since the witness will just be the bijection. Hence, GNI is in co-NP, and it is not known whether GNI is in NP .

In an interactive proof system for GNI , the verifier asks the prover to distinguish between isomorphic graphs.

- G_1, G_2 are given to both prover, verifier.
- The verifier chooses a random $r \in \{1, 2\}$ uniformly at random.
- The verifier picks a random permutation π of the set $\{1, 2, \ldots, |V(G_1)|\}$
- the verifier constructs the graph H as the permutation of G_r under π. The graph H is sent to the prover. That is, $H \equiv \pi(G_r)$.
- the prover P replies with $r' \in \{12\}$. The reply r' is 1 if H is isomorphic to G_1, and 2 otherwise.
- The verifier accepts if $r = r'$.

Note that $H \sim G_r$. Now if $G_r \sim G_{other}$, then $H \sim G_r \sim G_{other}$, and so the prover has to literally guess between G_r and G_{other}, and at best it can simply guess. (Even though the prover has *unbounded computation*, it is unable to distinguish between G_r and G_{other}). In two rounds, the probability of the guesses of the prover being right is $\frac{1}{2}^2 = \frac{1}{4}$, which fulfils our soundness guarantee ($\frac{1}{4} < \frac{1}{3}$).

On the other hand if $G_r \nsim G_{other}$, then if the prover knows how to solve `GNI` , it can check between H, G_r, and G_{other} to consistently report G_r. In this case, the prover will *always* be correct, so this will pass compleness (since $1 > \frac{2}{3}$).

This is very interesting, because the verifier **does not know** whether $G_1 \sim_? G_2$. The verifier tries to engage with the prover, to understand whether $G_1 \sim_? G_2$.

Theorem 6 *GNI* $\subset$ *IP*(2)

Theorem 7 *co-NP* $\subset$ *IP*

Theorem 8 *IP* $=$ *PSPACE*

Chapter 6

Parallel Computing

Moore's law, blocking factors:

- Memory wall: memory latency was higher than compute.
- Power wall: Power leakage.
- ILP wall: ILP via branch prediction, out-of-order-execution, and speculative execution. Diminishing returns from instruction-level parallelism.

Interesting questions one can ask:

- How do we analyze parallel algorithms?
- Can every sequential algorithm be parallelized?
- What are the complexity classes related to parllel computning?
- Can sequential programs be automatically converted to parallel programs?

Concurrent data structures, course: Professor Govindarajulu.

6.1 `PRAM` model

Global shared memory, shared by n processors. Each processor has individual bidirectional buses into the memory.

Also, note that we have *random access* into the memory, which is different from a turing machine, which only offers sequential access.

(Sid question: what is a problem that can be solved efficiently given random access and not with sequential access?)

Access to shared memory costs the same as one unit of computation.

Different flavours provide different semantics to concurrent access of shared memory (EREW, CREW, CRCW).

- EREW - Exclusive Read, Exclusive Write. No scope for memory contention, so algorithm design is tough.

- CREW - Concurrent Read, Exclusive Write. Allow processors to read simeltaneously, writes are still exclusive. Is practically feasible.
- CRCW - Concurrent Read, Concurrent Write Processors can read/write simeltaneously. So here, we need to specify semantics of concurrent writes!

Flavours of concurrent write semantics:

- COMMON: Concurrent write is allowed as long as all the values being attempted are equal. Eg: boolean OR of n bits. Each processor p_i will read $a[i]$. if $a[i] = 1$, then p_i tries to write 1. it doesn't matter how many processors try to write 1, if any bit is 1, then the output will be 1. We need to make sure the cell is initialized to 0, so that if all bits are 0, the answer is 0.
- ARBITRARY: In the case of a concurrent write, *someone* wins and its write takes effect.
- PRIORITY: Assumes that processors have numbers that can be used to decide which write succeeds.

6.2 Matrix multiplication

- Recursively block the matrices.
- Multiply strips (cannon's algorithm).

6.2.1 Prefix computations

Given an array A of n elements and an associative operator $\circ$, we want to compute $P(i) = \circ_{k\in[0\ldots n]}A[k]$. $P(0) = A(0), P(1) = A(0) \circ A(1) \ldots$.

We can use the naive approach:

```
def scan(A, op):
    out[0] = A[0]
    for i in range(1, length(A)):
        out[i] = op(A[i], out[i - 1])
```

We have linear RAW dependences: $out[i] \to out[i-1]$.

So, we crete a complete binary tree with processors at the internal nodes. Input is at the leaf node. Each node performs the $\circ$ of its left and right subtree.

```
def sumfast(A, op, l, r):
    if (l == r):
        return A[l]
    else:
        mid = (l + r) / 2
        return op(scanfast(A, op, l, mid), scanfast(A, op, mid, r));

def sum(A, op):
    return sum(A, op, 0, length(A))
```

Note that this will not give us *prefix sums*. We can finish in $\log(n)$ time given 2^n processors.

For a *prefix sum*, we need a combination of upward and downward traversal. First send data from bottom to top. Next, send down data from top to bottom of the prefix sums towards the leaves.

Analysis of prefix computation

- Step 1 can use $\frac{n}{2}$ processors in parallel, each using 1 unit of time.
- Step 2 is a recursive calls and takes $T(\frac{n}{2})$ time.
- Step 3 uses $n$processors each of which take 1 unit of time.

Work done by the algorithm: $W(n) = W(n/2) + O(n)$ ($O(n)$ for the first and third step). $W(n) = O(n)$ is the solution.

Optimal parallel algorithm

A parallel algorithm that does the same amount of work as the best known sequential algorithm is called an *optimal algorithm*.

This makes sense, because if we set `num processors` $= 1$, we want the asymptotics to match the sequential algorithm.

Chapter 7

Design models of parallel algorithms

7.1 Partitioning

This is similar to divide-and-conquer, but we don't need to *combine* solutions! We can treat problems independently and solve it in parallel. Examples are parallel merging and searching.

We generate subproblems that are independent of each other. Example is quicksort. Once we partition the array into two subarrays, we sort the subarrays recursively.

7.1.1 Merging in parallel by partitioning

Two sorted arrays A and B are to be merged into an array C.

suboptimal algorithm — Time: $O(\log n)$**, work:** $O(n \log n)$

We define a function $Rank(x_0, X) = |\{x < x_0 \mid x \in X\}|$. Note that the position of x_0 in $sorted(X)$ is equal to $Rank(x_0, X)$. **Claim:** $Rank(x, C) = Rank(x, A) + Rank(x, B)$.

For $x \in A$, $Rank(x, A)$ is immediately available (since A is sorted). We need to find $Rank(x, B)$, but we can find this using binary search through B.

Time for each binary search is $O(\log n)$. Total time for merging is $O(\log n)$, since we are doing each binary search in parallel — we just need to read the array B, no need to update. The total work is $O(n \log n)$, since we are performing $O(\log n)$ work for n elements.

Note that this is **non optimal**. The sequential algorithm has a time complexity of $O(n)$.

We are going to try and reduce the work to $O(n)$.

Merging, take 2, optimal — time: $O(\log n)$**, work:** $O(n)$

General technique is to solve a smaller problem in parallel, and then extend the solution to the entire problem!

- The problem size to be solved is guided by the factor of non-optimality in the current algorithm. We need to reduce the total work to $O(n)$.

  For input size n, we do $O(n \log n)$ work. So, for input size $n/\log n$, we do $O(n/\log n \times \log(n/\log n) \sim O(n) + O(\log(\log(n)) \sim O(n)$.

- We pick every $\log n$th element of A. We merge the selected elements of A and B. However, we still perform binary search on the entireity of B.
- Pick elements $A[\log n], A[2\log n], \ldots, A[n - \log n], A[n]$, and rank the, in B (ie, find their corresponding positions in B.)
- Define $[B_{r(i)}, \ldots, B_{r(i+1)}] \equiv$ portion of B between $A[r \log i]$, $A[(r+1)\log i]$ in B.

```
A = (5) 6 9 12 (15) 18 19 (21) 23 26
B = 1 4 (..5..) 7 8 10 11 12 (..15..) 16 17 20 (..21..) 22

In the output array, we can merge
the array of B between the (..) elements of A
```

  The problem is that the size of $\log n$ per chunk in A does not mean that the size is $\log n$ in B.

```
A = (5) 6 9 12 (15) ... (...) ...
B = 6 6 6 6 6 6 6 ... 6

In this case, the entireity of B is between [5, 15]
```

  So, if we can somehow control the size of B, so, we can perform binary search in $O(\log n)$, with $n/\log n$ processors.

  We then need to perform the merge with $O(\log n)$, **under certain conditions**. There are again $n/\log n$ such merges.

  The work is $O(n)$.

  So now, the only thing we need to control is the size of partitions of B.

- If $[B_{r(i)}, \ldots, B_{r(i+1)}]$ is too large, then we can pick $\log n$ items of this section, and we can rank them in A! Each piece in A will be smaller than $\log n$, since the partition of A was already $\log n$.
- we can merge two sorted arrays of size n in time $O(\log n)$ with work $O(n)$. This algorithm works in `CREW`. We can improve this further, we will see this later.

7.1.2 Searching faster — time: $O(1)$, work: $O(\sqrt{n})$

Each binary search takes $O(\log n)$ time, and we have $O(n/\log n)$ subproblems, each of size $O(\log n)$.

Can we make search faster?

- Consider a sorted array A with n elements. We want to search for an element x. Given p processors, we can search at the indeces $1, n/p, 2n/p, \ldots, n$.
- Record the result of each comparison as 1 or 0. $cmp[i] = 1 \equiv A[i] < x$, $cmp[i] = 0 \equiv A[i] \geq x$. More succinctly, `cmp = map (\a -> a < x) A`.

- *cmp* will either have all 0s, all 1s, or a shift from 1s to 0s.
- If we have a shift from 1s to 0s, we know that x is likely in the n/p segment corresponding to the shift from 1 to 0.
- So now, we can recursively search that small segment.
- $T(n) = T(n/p) + O(p)$. ($O(p)$ since *cmp* has length p). Hence, $T(n) = T(n/p) + O(1)$. This gives us $O(\log n)$ when $p = 1$ (make sure this is correct, there is some **off by one here**.
- When $p = O(\sqrt{n})$, the time taken will be $O(\log n / \log(\sqrt{n})) = O(1)$ This looks useless from a work point of view, but we want to see what this is good for!

7.1.3 From parallel search to merge — time: $O(\log \log n)$, work: $O(???)$

- We have two sorted arrays A and B, which we want to merge.
- We want to rank some elements of A to create paritions of B.
- Let us take $\sqrt{n}$ elements of A in B.
- We have n processors, so each search can use $n/\sqrt{n} = \sqrt{n}$ processors.
- each search now finishes in $O(1)$ time.
- the problem is that the partitions of A are much larger now (they are $\sqrt{n}$ large).
- we have a $\sqrt{n}$ sized piece of A, and we have a size of B that is of size (?). Note that for each piece of A, we now choose to allocate $\sqrt{n}$ processors.
- So, we pick $n^{\frac{1}{4}}$ elements of A in B, each of which uses $n^{\frac{1}{4}}$ processors. Size of each piece is now $n^{\frac{1}{4}}$.
- So, we pick $n^{\frac{1}{8}}$ elements of A in B, each of which uses $n^{\frac{1}{8}}$ processors. Size of each piece is now $n^{\frac{1}{8}}$.
- We reduce the sequence $n \to \sqrt{n} \to n^{\frac{1}{4}} \to n^{\frac{1}{8}} \cdots \to O(1)$. This can be done in $\log \log n$ steps!

Chapter 8

Parallel algorithms, part 2

8.1 Pointer jumping

Pointer jumping is the technique of updating a successor with the successor's successor. As this is repeated, the sucessor gets closer to the root node. The distance between a node and its successor doubles in each round trip.

```
# F := Forest consisting of rooted, directed trees. F is specified using
# an array P

# P[i] := P[i] = j iff (i, j) is an edge in F. That is, j is a parent
# of i.

# P must contain self-loops at *each of the roots*. Each arc is
# specified by (i, P[i])

# output: a list S, containing the root of i at S[i]
def find_roots(P):
    for i in parallel([1, n]):
        S[i] = P[i]

        while S[i] != S[S[i]]:
            S[i] = S[S[i]]

    return S
```

8.2 List Ranking

We have a list L of n nodes. $S[i]$ contains a pointer to the node *following* node i on L, for $1 \leq i \leq n$. We assume that $S(i) = 0$ when i is the end of the list. The *List-ranking problem* is to determine the distance of each node i from the end of the list.

8.2.1 non-optimal list ranking using pointer jumping

```
def listrank(S):
    for i in parallel([1, n]):
        S[i] =  R[i] == 0 ? 0: 1


    for i in parallel([1, n]):
        Q[i] = S[i]
        while Q[i] != 0 && Q[Q[i]] != 0:
            R[i] = R[i] + R[Q[i]]
            Q[i] = Q[Q[i]]
```

this takes time $O(\log n)$, using $O(n \log n)$ operations.

8.2.2 Making our algorithm better

We want to make our algorithm better, we have a work complexity of $O(\log n)$ which we are trying to eliminate.

There are also some implementation issues. In the PRAM model, syncrhonous execution means that all n processors execute each step in parallel. So, we can have inconsistent results!

How do we pick a list of size $n/\log n$? Our input is in the form of an array of successor elements. So, we can't take equi-distant parts of the array, since it won't be a valid sub-list anymore.

What we can do is to pick *independent nodes*. Formally, we want to remove an independent set: vertices that share no edge amongst them.

```
1 -> (8) -> 5 -> 11 -> (2) -> 6 -> (10) -> 4 -> 3 -> (7) -> 12 -> 9
on removal:
1 -> 5 -> 11 -> 6 -> 4 -> 3 -> 12 -> 9
```

We can remove `8, 2, 10, 7` in parallel.

We want to go to a subset of size $n/\log n$, but by removing independent nodes, we can go smallest to $n/2$.

```
a -> (b) -> c -> (d) -> e -> (f) -> ...
```

There are no other elements in the above chain we can add to the independent set. So, we will need to repeat our process to reach $n/\log n$.

8.3 Detour: Independent sets

In a graph $G = (V, E)$, a subset of nodes $U \subseteq V$ is called an *independent set* if:

$$U \text{ is an independent set of G} \equiv \forall (u_1, u_2) \in U, u_1 \neq u_2 \implies (u_1, u_2) \notin E$$

.

Linked lists, when viewed as graphs, have large independent sets.

8.3.1 Technique: Symmetry breaking

The idea is to look at a symmetric setting, and then induce differences between them. Independent sets are symmetric, because given two nodes that are neighbours, they're both eligible to be in the independent set (modulo other obstructions). This algorithm is applicable for graph coloring.

Usually, this technique requires randomization. However, there are special cases where fast, deterministic symmetry breaking is possible. Linked lists and directed cyclic graphs are examples where this is possible.

We first construct a symmetry-breaking based graph coloring solution, which is then used to find independent sets.

8.3.2 Coloring by Symmetry breaking

Considered a directed cycle of n nodes $0 \dots n-1$.

Assume we have 8 nodes, which are labeled using 3 bits. We may not have consecutive numbering of our nodes, so we assume that our nodes are randomly numbered, from 0 to 7 (3 bits).

- Initially, treat each number as a color for the vertex.
- We can reduce the number of colors to $\log n$ in one step:
  - Compare color with the parent. $Newcolor(u) = 2k + color(u)[k]$.
  - k is the index of the first bit position from LSB where $color(u)$ and $color(parent(u))$ differ.
  - So, $color(u)[k]$ is indexing the k-th bit of $color(u)$ starting from LSB.
  - note that $0 \leq k \leq \log n - 1$.
  - such a k will always exist, since we are guaranteed some unique labelling of the vertices when we start this process.

```
This table may not be fully accurate, re-check:

u   | v   | new color (mostly 2 bits)
110 | 000 | 11 (k = 1)
000 | 100 | 100 (k = 2)
100 | 111 | 00 (k = 0)
010 | 001 | 00 (k = 0)
001 | 011 | 10 (k = 1)
011 | 101 | 11 (k = 1)
111 | 010 | 01 (k = 0)
101 | 110 | 01 (k = 0)
```

Correctness proof

Proof by contradiction. Suppose we have an edge (u, v), where $newcolor(u) = newcolor(v)$. Let $newcolor(u) = 2k + color(u)[k]$, and $newcolor(v) = 2r + color(v)[r]$.

If $newcolor(u) = newcolor(v)$, then $2k + color(u)[k] = 2r + color(v)[r]$. Rearranging, we get that $2(r-k) = color(u)[k] - color(v)[k]$.

If $k = r$, then we get that $color(u)[k] = color(v)[k]$. But this cannot happen, because by definition, k is the bit where u, v first differ!

If $k \neq r$, then we get that $2(r-k) = color(u)[k] - color(v)[k]$. By comparing magnitudes, we see that $\left|color(u)[k] - color(v)[k]\right| \leq 1$ (since we're subtracting bit values), while $\left|2(r-k)\right| \geq 2$. This can't happen either for two equal values!

Analysing number of new colors

In one iteration, we can reduce the number of colors from n to $2\log n$. For the new colors, we only need $1 + ceil(\log\log n)$ bits.

Can we repeat this technique? Yes, we can. This technique reduces number of colors from t to $1 + ceil(\log t)$. At some point, $t < 1 + ceil(\log t)$, at which point we will be forced to stop.

This stopping point happens at $t = 3$. So, we repeat until only 8 colors are being used.

The total time is the solution to the recurrence $T(n) = T(\log n) + 1$. We define the function that solves the recurrence as $\log^* n$.

$$\log^* n = i \equiv \log(\log(\ldots i \text{ times} \ldots (n))) = 1$$

Reducing from 8 to 3 colors

for i in [8..3], If node u is colored i, then choose a color among $\{1, 2, 3\}$ that is not the same as its neighbours.

```
# color: map (vertex -> color)
# V: vertex set
for c in range(8, 3):
    for v in V:
        if color[v] == c:
            # we will always have one number here, since we have three
            # colors, and we are only removing two colors
            newcolor[v] = rand ({1, 2, 3} - color[pred(v)] - color[succ(v)])
    newcolor = color
```

This is always possible.

8.3.3 Finding Indepenent sets using the coloring

For bounded degree graphs colored with $O(1)$ colors, a coloring is equivalent to finding a large independent set.

Iterate on each color and count the number of nodes with a given color. Pick the subset of like colored nodes of the largest size. It is clearly an independent set, and has size of at least some fraction of n.

8.3.4 Algorithm outline

```
def rank(L):
    L1 = L

    for r in [2, R]:
        color the list with 3 colors
        pick an independent set U_i of nodes of size >= n /3
        L_i = remove nodes in U_i from L_{i - 1}

    Rank the List L_r using poiner jumping


    for i in [r, 1]:
        reinsert the nodes in U_i into L_i
```

We are removing $n/3$ nodes in each iteration, we want to stop at $n/\log n$ nodes. We need $O(\log\log n)$ iterations.

8.3.5 total time taken

Each iteration is $O(\log^* n)$. At $O(\log\log n)$ iterations, this takes $O(\log^* n \log\log n)$ time.

To rank the remaining list, we take $O(\log n)$ time.

To reintroduce the removed elements, we take $r = O(\log\log n)$ iterations, $O(\log\log n)$ time.

8.3.6 Slowing down re-introduction to make this optimal

We can reintroduce slower.

we can use only $n/\log n$ processor

8.3.7 Slowing down independent set

8.4 Back to list ranking

- Anderson-Miller is in JaJa's book
- Hellman-JaJa is another popular approach (read the paper)

Chapter 9

Tree processing

9.1 Traversal via an Euler tour

Definition 8 *an **Euler tour** is a cycle of a graph that includes every edge of the graph exactly once.*

Lemma 2 *A directed graph G **has an Euler tour** iff for every vertex,its in-degree equals its out-degree.*

For a tree $T = (V, E)$, to define an euler tour, we make it a directed graph. $T_e = (V_e, E_e)$, where $V_e = V$, and $E_e = \cup_{(u,v)\in V}\{(u, v), (v, u)\}$ That is, each (u, v) in E creates two edges (u, v), and (v, u) in E_e. T_e will have an Euler tour.

We have to define a successor function $s : E_e \to E_e$. Here, the successor for an edge. For a node u in T_e, order its **neighbours (both incoming and outgoing)** $v_1, v_2, \ldots v_d$. This can be done **independently at each node**. For $e = (v_i, u)$, set $s(e) = (u, v_{i+1 \bmod d})$. This choice of s is valid since we always have both edges (x, y) and (y, x), and we are therefore assured that (v_i, u) will be an incoming edge, and $(u, v_{i+1}$ will be an outgoing edge. Also, compute $i + 1$ modulo d, so that we eventually cycle.

TODO: relabel vertices to $[0..(d-1)]$ **so that modulo works properly TODO: add example**

Theorem 9 *s actually constructs a tour.*

Proof. Induction on number of vertices. If $n = 1$, obviously true. If $n = 2$, at most one edge present. We will go along the edge and come back, which is a valid tour.

- Let the tour be well defined for $n = k$. We will prove it for $n = k + 1$.
- Every tree has at least one leaf, call it l. Create a tree $T' = T/\{l\}$.
- Let u be a neighbour of l in T.
- Let $N(u) = \{v_0, v_1, \ldots v_i = l, v_{i+1}, \ldots v_d\}$.
- Set $s_{new}(u, v) \equiv (v, u)$. Set $s_{new}(v_{i-1}, u) \equiv (u, v)$.
- For all other vertices, $s_{new}(e) = s(e)$.

9.2 Using euler tours for traversal

Operations on a tree such a rooting, preorder, and postorder traversal can be converted to routines on an Euler tour.

9.2.1 Rooting a tree

Designate a node in a tree as the root. All edges in the tree are directed towards (or away) from the root.

- let $\{v_1, v_2, \ldots v_d\}$ be the neighbours of root node r.
- we mark the final edge of the tour as `NIL`, so we get an Euler path, and not an Euler tour.
- the edge (r, v_i) appears before (v_i, r).
- so the edge $parent \to child$ appears before $child \to parent$
- So, if uv precedes vu, then set $u = parent(v)$. Orient the edge uv as $v \to u$ (that is, $child \to parent$), since we want all edges towards the root.

9.2.2 Preorder traversal

We have a rooted tree with r as the root. In a preorder traversal, a node is listed before any of the nodes in its subtrees.

In an Euler tour, nodes in a subtree are visited by entering subtrees, and the exiting towards the parent.

If we can track the first occurence of a node in an euler path, this will tell us the preorder traversal. Note that edges in the euler tour occur first as $parent \to child$, and later as $child \to parent$. So, we can look at the sequence of edges in the euler tour, and find the preorder numbering.

9.2.3 Expression tree evauation of binary trees

Tree may not be balanced.

We use the `RAKE` technique to evaluate subexpressions. We rake the leaves from the expression tree — we remove the leaf node and its parent.

- $T = (V, E)$ is a tree rooted at root node r. $p : E \to E$ is the parent function.
- One step of the rake operation at a leaf l with $p(l) \neq r$ involves:
  - Remove node l, $p(l)$ from the tree
  - Make the sublings of l as the child of $p(p(l))$. That is, graft the siblings of l to the grandparent of l.

Why is this a good technique? Can this be applied in parallel to several leaf nodes? Yes, it can be applied to leaf nodes that don't share the same parent. In general, there is a richness of leaf nodes in a tree, since there are only $n - 1$ edges.

Each application of rake at all leaves reduces the number of leaves by half. Each application of `RAKE` is $O(1)$. So, total time is $O(\log n)$.

```
def shrinkTree(R):
    compute labels for leaf nodes, store in array A (exclude leftmost
    and rightmost nodes in this A)

    for _ in range(k):
        apply rake operation to all odd numbered leaves that are
        the *left* children of their parent

        apply rake operation to all odd numbered leaves that are
        the *right* children of their parent

        update A to be the remaining even leaves
```

Applying `Rake` means that we can process more than one leaf node at the same time.

Fo expression evaluation, this may mean that an internal node with only one operand gets raked.

```
          + g(u)

    + p(u)

Y     X (u)


--After raking--

   + g(u)
Y
```

- Transfer the impact of applying the operaot at p(u) to the sibling of u
- $R_u = a_u X_u + b_u$
- X_u is the result of the subexpression at node u – $X_u = f(left, right)$
- adjust a_u and b_u during any rake operation appropriately
- Initially, at each leaf node, $a_u = 1, b_u = 0$.

```
          + g(u)

    + p(u)
    X_w
    a_w
    b_w

v        (u) 5, 1, 0
```

```
X_v,
a_v,
b_v

--After raking--

     + g(u)
v
X_v'
a_v'
b_v'
```

- Before removing $p(u)$, the contribution of $p(u)$ to $g(u)$ will be $X_w a_w + b_w$.
- we want what $p(u)$ used to calculate to be what v calculates after.
- $X_w = (X_u a_u + b_u) + (X_v a_v + b_v) = (X_v a_v) + (X_u a_u + b_u + b_v)$
- What $p(u)$ used to calculate is: $a_w X_w + b_w = a_w(a_v X_v + a_u x_u + b_u + b_v) + b_w = a_w a_v x_v + a_w(a_u X_u + b_u + b_v) + b_w$
- what $p(v)$ should be: $a'_v = a_w a_v$, $b'_v = a_w(\dots)$

For other operators, proceed in a similar fashion (**TODO: do this and send to kiko, he seems interested!**)

Chapter 10

Tree processing, mach 2

Chapter 11

Parallel Graph algorithms

We now move to parallel graph algorithms. We will view recent work on 1-connectivity and 2-connectivity.

A graph is 1-connected if every pair of vertices has a path between them.

In a sequential model, DFS/BFS can be used. In the parallel setting, we know that DFS cannot be parallelized. BFS can be, but it is inefficient to do so (We will see this later). So, we need new approaches to this problem.

A connected component is a subset of vertices V_i such that every pair of vertices in V_i have a path between them.

The algorithm we study has some resemblance with the union-find algorithm.

11.1 The algorithm for 1-connectivity

The algorithm is by **Chandra, Sarwate, Hirschberg**.

11.1.1 Intuition

- Consider an initial set of rooted trees where each tree contains a single vertex.
- Eventually, each rooted tree will correspond to a connected component.
- Two trees can be combined into a bigger tree if these trees contain vertices u and v which belong to different trees, and the edge $(u, v) \in E(G)$. The next iteration proceeds with trees merged from the previous iterations.
- The parallelism is in merging the trees

Instead of calling it a tree, we call it a *super-vertex*.

We define the *graph for an iteration* as the graph of super-vertices and edges between the super-vertices.

- $G_0 = G$
- G_i is the graph with super-vertices at iteration i. We construct G_{i+1} from G_i.

Important questions:

- How to represent and arrange the super-vertices?
- How do we build the graph for the next iteration?
- How many iterations do we need?
- What is the time and work complexity of the algorithm?

11.1.2 How to represent the matrix?

We will use an adjaceny matrix to start with. Initially, the matrix is of size $n \times n$ where $n = |V(G)|$.

If the graph at the start of the kth iteration has n_k vertices, then the matrix is of size $n_k \times n_k$. We will refer to this matrix as A_k.

$A_k[u, v] = 0$ means that the super-vertices do not share an edge. $A_k[u, v] = 1$ if u and v do share an edge.

11.1.3 How do we build the graph for the next iteration?

We will make use of *concurrent writes* to create the matrix A_{k+1} from the graph G_k.

In G_k, if there exist two distinct super-vertices u_s and v_s such that a vertex u in u_s and a vertex v in v_s and the edge (u, v) is in $E(G)$.

11.1.4 How do we arrange the super-vertices?

Each vertex of G is given a label ($label : V(G) \to \mathbb{N}$, $label$ is injective) so that if $label(u) = label(v)$, then u and v are part of the same super-vertex.

The common label used for all vertices in the super-vertex will be the label of the smallest numbered vertex in the super-vertex.

- We set this up such that **the root of every tree is the node with the smallest id**.
- As we combine two trees to make a bigger tree, we will make the tree with the lower root id as the parent.
- We use *pointer jumping*f to adjust the labels.

11.1.5 The merging algorithm

We define a function:

$$C : V \to V$$
$$C(v) = min\{label(w) \mid A[v, w] = 1\}$$

In the first iteration, starting with an initial set of n trees, we merge trees as follows:

- C creates a forest of trees on V and with $E = \{(v, C(v)) \mid v \in V(G)\}$.

- C partitions V such that all vertices in the same connected component are in the same partition.
- Each cycle in the forest is either a self-loop or of length 2.
- We now use pointer jumping to make everyone in a tree agree on a representative.

11.1.6 The algorithm

```
A_0 = A
n_0 = n
k = 0
while not done:
        k = k + 1
        for v in V pardo:
                C(v) = min {w | A[k - 1][v][w] == 1}

                Shrink each tree in the forest

        pass
```

11.1.7 Analysis

- Number of iterations: We can show that in each iteration till the ened, the number of super vertices decreases by a factor of two. So, the total number of iterations is $O(\log n)$.
- Time spent in each iteration: In each iteration, we need to do a pointer jumping across the forest, This takes $O(\log n)$time and $O(n)$ work.
- Total time: $O(\log^2 n)$.
- Work: $O(n+m)$ $(n = |V|, m = |E|)$

There are better algorithm that reduce the time to $O(\log n)$ by **Shiloach and Vishkin** — Don't perform aggressive pointer jumping every round, but perform one step of the pointer jumping each round.

11.2 k-Connectivity

Famous result by **Cherian and Thirumella**:

A graph is k-connected iff the subgraph H is k-connected, where H is:

$$T_1 = BFS(G)$$
$$T_2 = BFS(G/T_1)$$
$$T_3 = BFS(G/(T_1 \cup T_2))$$
$$T_k = \dots$$
$$H = T_1 \cup T_2 \dots T_k$$

Note that each of the T_i are disjoint, and each T_i may have n edges, so H has only kn vertices. This is drastically better that $|E| = O(n^2)$.

The bottleneck in practice for this is BFS.

Current work tries to replace the BFS with other structures, and then we repair the damage later on.

Chapter 12

Randomized Algorithms

Reference — randomized algorithms, Motwani and Raghavan.

12.1 Randomized Quicksort

```
def randquicksort(l):
        pivot = uniform_random_pick(S)
        S1 = [x for x in l if x > pivot]
        S2 = [x for x in l if x < pivot]
        return randquicksort(S1) + [x] + randquicksort(S2)
```

We assume time to partition is $O(n)$.

Maximum value of $T(n)$ occurs when the pivot element x_i is the largest element of the remaining set. So, each iteration, we take i time to partition, and then recurse.

$$T_{worst}(n) = n + (n-1) + \cdots = O(n^2)$$

$$P_{worst}(n) = \frac{1}{n} \cdot \frac{1}{n-1} \cdots \frac{1}{2} \cdot 1 = 1/n!$$

Best case is when the pivot splits the set S into two subsets. Then, $T(n) = O(n \log n)$.

When we pick pivots, as long as we can guarantee that a *constant* fraction is inside one of the pivot sets, we will still get log, since it will reduce by $constant^k$ for k steps.

So, $O(n \log n) \leq T(n) \leq O(n^2)$. We now derive the *expected value* of $T(n)$.

- If the ith smallest element is choice as the pivot, then $|S_1| = i-1, |S_2| = n-(i-1)-1 = n-i$. This choice has probability $\frac{1}{n}$, since we are picking an element uniformly. i is the *rank* we would like to pick.
- $T(n) = x + T(X) + T(n-X)$ is the recurrence relation, where X is a random variable, $X \in [0, n]$, and X denotes the *rank of the element* we would like to pick in the array.
- now, $T(n)$ is also a random variable.
- $\Pr[X = i] = \frac{1}{n} = Pr[n-1-X = i]$, since the pivot is chosen *uniformly at random.*

$$
\begin{aligned}
&\mathbb{E}\big[T(n)\big] \\
&= \mathbb{E}\big[n + T(X) + T(n-X)\big] \\
&= n + \mathbb{E}\big[T(X)\big] + \mathbb{E}\big[T(n-X)\big] \\
&= n + \sum_{i=1}^{n-1} \mathbb{E}\big[T(i)\big] \cdot \Pr[X = i] + \sum_{i=1}^{n-1} \mathbb{E}\big[T(n-i)\big] \cdot \Pr[n - X = i] \\
&= n + \sum_{i=1}^{n-1} \mathbb{E}\big[T(i)\big] \cdot \frac{1}{n} + \sum_{i=1}^{n-1} \mathbb{E}\big[T(n-i)\big] \cdot \frac{1}{n} \\
&= n + \frac{2}{n} \sum_{i=1}^{n-1} \mathbb{E}\big[T(i)\big]
\end{aligned}
$$

Let $f(n) \equiv \mathbb{E}\big[T(n)\big]$ for simplicity

$$f(n) = n + \frac{2}{n} \sum_{i=1}^{n-1} f(i)$$

$$f(n) = n + \frac{2}{n}\big(f(1) + f(2) + \ldots f(n-1)\big)$$

$$nf(n) = n^2 + 2\big(f(1) + f(2) + \ldots f(n-1)\big)$$

$$(n-1)f(n-1) = (n-1)^2 + 2\big(f(1) + f(2) + \ldots f(n-2)\big)$$

subtracting $(n-1)f(n-1)$ from $nf(n)$,

$$nf(n) - (n-1)f(n-1) = n^2 - (n-1)^2 + 2f(n-1)$$

$$nf(n) = (2n-1) + 2f(n-1) + (n-1)f(n-1)$$

$$nf(n) = (2n-1) + (n+1)f(n-1)$$

$$f(n) = \frac{2n-1}{n} + \frac{n+1}{n} f(n-1)$$

We now need to guess $f(n)$. We guess $f(n) \leq 2n \log n$:

$$
\begin{aligned}
f(n) &= \frac{2n-1}{n} + \frac{n+1}{n} 2(n-1)\log(n-1) \\
&= \frac{2n-1}{n} + 2(n+1)\log(n-1) - 2\frac{(n+1)}{n}\log(n-1) \\
&\leq 2n\log n + \left[\frac{2n-1}{n} + 2\log(n-1) - 2\frac{(n+1)}{n}\log(n-1)\right] \\
&\leq 2n\log n + \left[\text{less than 0 (look this up)}\right] \leq 2n\log n
\end{aligned}
$$

So, the randomized quicksort algorithm has *expected* runtime $O(n \log n)$. However, we don't know the spread *around* the expected value. To answer these questions, we study **tail inequalities**.

12.2 Tail inequalities

The more information we know about the random variable, the better the estimate we can derive about a given tail probability.

12.2.1 Markov Inequality

If X is a non-negative valued random variable with an expectation of μ, then

$$\Pr[X \geq c\mu] \leq \frac{1}{c}$$

Proof

$$
\begin{aligned}
&\mu = \sum_a a \Pr[X = a] \\
&\mu = \sum_{a<c\mu} a \cdot \Pr[X = a] + \sum_{a \geq c\mu} a \cdot \Pr[X = a] \\
&\text{since } X \text{ is non-negative, } a \geq 0 \\
&\mu \geq 0 + \sum_{a \geq c\mu} a \cdot \Pr[X = a] \\
&\mu \geq c\mu \sum_{a \geq c\mu} \Pr[X = a] \\
&\mu \geq c\mu \cdot \Pr[X \geq c\mu] \\
&\Pr[X \geq c\mu] \leq \frac{1}{c}
\end{aligned}
$$

Bounding quicksort using Markov inequality

$$\Pr[T(n) \geq 12n\log n] = \Pr\left[T(n) \geq 4\mathbb{E}\big[T(X)\big]\right] \leq \frac{1}{6}$$

12.2.2 Chebyshev Inequality

Let X be a radom variable with mean μ. We define the variance $var(X) \equiv \mathbb{E}\big[(X-\mu)^2\big]$. The standard deviation, $\sigma_x \equiv \sqrt{var(x)}$.

- Let X be a random variable with mean μ and standard deviation σ.
- Then, $\Pr\big[|X-\mu| \geq c\sigma\big] \leq \frac{1}{c^2}$

Let us define $Y = var(x) = (X-\mu)^2$

$$\mathbb{E}\big[Y\big] = \mathbb{E}\big[var(X)\big] = \sigma^2$$

$$\Pr\big[|X-\mu| \geq c\sigma\big] = \Pr\big[(X-\mu)^2 \geq (c\sigma)^2\big] = \Pr\big[Y \geq (c\sigma)^2\big]$$

We can now apply markov inequality:

$$\Pr\big[Y \geq (c\sigma_x)^2\big] =$$
(**TODO:** why is $\sigma_x^2 = \mu_y$?)
$$\Pr\big[Y \geq c^2\mu_y\big] \leq \frac{1}{c^2}$$

Chapter 13

Tail Inequalities

Better inequalities can be ontained by a powerful technique called Chernoff bounds. However, they are harder to use.

13.1 Chernoff bounds

Let $X_1, X_2, \ldots X_n$ be**independent, identically distributed** (IID) random variables.

Let $X = X_1 + X_2 + \ldots X_n$.

Let each X_i be a *Bernoulli* random variable - that is, the values the random variable can take is $\{0, 1\}$.

Let $\Pr(X_i = 1) = p$. Hence, $\Pr(X_i = 0) = 1 - p$.

Finally, let us compute $\mathbb{E}\big[X\big] \equiv \mu$. We exploit the fact that they are IID to derive:

$$\mathbb{E}\big[X\big] = \sum_i \mathbb{E}\big[X_i\big] = n(1 \cdot p + (1-p) \cdot 0) = np$$

13.1.1 Physical interpretation

Consider throwing a biased coin over n trials. Each trial, the prob. of heads is p. So, each X_i corresponds to the fact that the ith trial is heads. X counts the number of heads over the n trials.

13.1.2 The bound

$$\Pr(X \geq \mu(1+\delta)) \leq \left[\frac{e^\delta}{(1+\delta)^{1+\delta}}\right]^\mu$$

13.1.3 Proof technique

We use higher order moments ($\mathbb{E}\big[x^k\big]$ is the kth moment). We will use *exponential moments* - That is, $\mathbb{E}\big[e^x\big]$ style moments.

13.1.4 The proof

For each X_i, we define $Y_i \equiv e^{tX_i}$, for a real number t that will be **chosen later**.

Since the distribution is IID, we will use p and $1-p$ for the random variables X_i. We do not need to consider a separate p_i for each X_i.

$$\mathbb{E}\big[Y_i\big] = \mathbb{E}\big[e^{tX_i}\big] = p \cdot e^{t \cdot 1} + (1-p) \cdot e^0 = p \cdot e^t + 1 - p$$

Now, we define another random variable $Y \equiv Y_1 \cdot Y_2 \ldots Y_n$.

$$\mathbb{E}\big[Y\big] = \mathbb{E}\big[Y_1 \cdot Y_2 \ldots Y_n\big]$$

Since Y_i is independent, expectation of product = product of expectation

$$\mathbb{E}\big[Y\big] = \prod_{i=1}^{n} \mathbb{E}\big[Y_i\big] = (pe^t + 1 - p)^n$$

First, notice that $Y = e^{tX}$:

$$Y = Y_1 \cdot Y_2 \cdot Y_n = e^{tX_1} e^{tX_2} \ldots e^{eX_n} = e^{t(X_1 + X_2 + \ldots X_n)} = e^{tX}$$

Further, Notice that we can transfer the bound of X to Y:

$$X \geq \mu(1+\delta) \implies e^{tX} \geq e^{t\mu(1+\delta)})$$
$$Y \geq e^{t\mu(1+\delta)}$$

Also notice that Y is a *positive valued* random variable, so we can now use the Markov inequality.

$$\Pr[Y \geq e^{t\mu(1+\delta)}] \leq \frac{\mathbb{E}\big[Y\big]}{e^{t\mu(1+\delta)}} = \frac{\prod(1-p+pe^t)}{e^{t\mu(1+\delta)}}$$
$$\leq \frac{\prod_{i=1}^{n}(1-p+pe^t)}{e^{t\mu(1+\delta)}}$$
since $1 + x \leq e^x$, (intuition: taylor series of e^x,
$$\leq \frac{\prod_{i=1}^{n} e^{(-p+pe^t)}}{e^{t\mu(1+\delta)}}$$
$$= \frac{e^{-\mu(1-e^t)}}{e^{t\mu(1+\delta)}}$$
$$= e^{-\mu(1-e^t)-t\mu(1+\delta)}$$

We wish to minimise this right hand side, do we differentiate wrt t and set it to 0. Rather than minimise the function, we can minimise the log of the function.

$$
\begin{aligned}
f(t) &\equiv \log(RHS) \\
f(t) &= -\mu(1-e^t) - t\mu(1+\delta) \\
f'(t) &= \mu e^t - \mu(1+\delta) \\
f'(t) = 0 &\implies t = \ln(1+\delta)
\end{aligned}
$$

TODO: We need to check that it is the minima, but that's up to us!
Now, we get:

$$\Pr[Y \geq e^{t\mu(1+\delta)}] \leq e^{-\mu(1-e^t)-t\mu(1+\delta)}$$

TODO: finish by substituting t

13.1.5 Simplification of the right hand size

We can simplify the RHS to give:

$$\Pr(X \geq \mu(1+\delta)) \leq \begin{cases} e^{-\mu\delta^2/4} & \text{if } \delta \leq 1 \\ e^{-\mu\delta \ln \delta} & \text{if } \delta > 1 \end{cases}$$

13.2 Example of use of Chernoff bounds

Say we have n balls and n bins. We throw each ball to a bin chosen independently and uniformly at random. $X_i \equiv$ number of balls in bin i We wish to compute $\mathbb{E}\big[X_i\big]$. For this, we define a new random variable

$$Y_{i,j} = \begin{cases} 1 & \text{ball } j \text{ goes to bin } i \\ 0 & \text{otherwise} \end{cases}$$

.

Note that $X_i = \sum_j Y_{i,j}$

$$\mathbb{E}\big[X_i\big] = \mathbb{E}\big[\sum_j Y_{i,j}\big] = \sum_j \mathbb{E}\big[Y_{i,j}\big] = \sum_{j=1}^{n} \frac{1}{n} = 1$$

Now, we want to ask $\Pr(X_i \geq 4)$?
By **Markov inequality**, we can answer:

$$\Pr(X_i \geq 4) = \Pr(X \geq 4 \cdot \mathbb{E}\big[X_i\big]) \leq \frac{1}{4}$$

By **Chebyshev inequality**, we first compute:

$$\begin{aligned}\mathbb{E}\big[Y_{i,j}\big] &= \frac{1}{n} \\ Var(Y_{i,j}) = \mathbb{E}\big[Y_{i,j}^2\big] - (EY_{i,j})^2 \\ &= (1^2 \cdot \frac{1}{n} + 0) - (\frac{1}{n})^2 \\ &= \frac{1}{n} - \frac{1}{n^2}\end{aligned}$$

$X_i = sum_j Y_{i,j}$, Y_{ij} independent. Therefore,

$$Var(X_i) = \sum_j Var(Y_{ij}) = n(\frac{1}{n} - \frac{1}{n^2}) = 1 - \frac{1}{n}$$

We now try to write the inequality we care about in the form of chebyshev:

$$Pr(|X - EX| >= c.sigma_x) <= (chebyshev)1/c^2$$

$$Pr(X_i >= 4) <= (chebyshev)Pr(|X_i - 1| >= 3) \leq \frac{\sigma_{x_i}^2}{9}$$

$$Pr(X_i >= 4) = Pr(X_i >= 1.(1+3)) <= (chernoff)e^{-13ln3} = 3^{-3}$$

Chapter 14

Applications of Tail Inequalities

14.1 Set balancing problem

Consider trying to divide a data set into two parts: a test set and a training set. To make sure that both are roughly similar, we want to divide it so that both data sets have the same number of data items for any given feature. Similar requirelements also occur for drug trials.

- We will think of n data items with n features (we can generalize this to m features), arranged in a matrix A. A has entries from $\{0, 1\}$. Rows are features, columns are data items
- The goal is to find a vector x of size n with entries from $\{-1, 1\}$ such that Ax has the has the smallest possible maximum absolute entry.
- Rows with $+1$ in x belong to one class, and those with -1 belong to another class.
- The maximum absolute value of each entry in Ax tells us how many data items differ at feature i according to the division by x.

TODO: setup example

14.1.1 Solution

Brute force is to take all possible choices of 2^n vectors.

Simple randomlized algorithm: We can consider choosing each element of x uniformly at random from $\{1, -1\}$. We will show that the maximum absolute entry of Ax in such an x is bounded by $O((n \log n)^{\frac{1}{2}})$ with high probability.

- Consider choosing each element of x uniformly at random from $\{-1, 1\}$.
- Let the product $Ax = y$.
- Consider any y_i, say y_1. By the definition of matrix multiplication, $y_1 = A_{11}x_1 + A_{12}x_2 + \cdots + A_{1n}x_n$.
- Note that $\mathbb{E}[x_i] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot -1 = 0$, and therefore, by linearity of expectation, $\mathbb{E}[y_i] = 0$.

- Since x_i is chosen independently, we can apply the Chernoff bound on y_i.
- We had derived a Chernoff bound for bernoulli $\{0,1\}$. But we use $\{1,-1\}$, so we need to either derive a new Chernoff bound, or we shift them to convert them to $\{0,1\}$.
- We want to know $\Pr[X \geq k]$ for some k.
- Define $Y_i = (1+X_i)/2$. Note that Y_i is $\{0,1\}$ valued.
- Define Y as the sum of Y_is.
- $\mathbb{E}\big[Y\big] = n/2$.
- Also note that $X \geq k$ iff $Y \geq n/2 + k/2$.
- Now, $\Pr[X \geq k] = \Pr[Y \geq n/2 + k/2]$
- $\Pr[Y \geq n/2(1+k/n)] = \Pr[Y \geq \mathbb{E}\big[Y\big](1+k/n)]$
- This is a Chernoff bound with $\delta = k/n < 1$.
- Applying Chernoff bounds with the delta, we get that

$$\Pr[Y \geq \mathbb{E}\big[Y\big](1+\delta)] \leq e^{-\mathbb{E}\left[Y\right]\frac{\delta^2}{4}} = e^{-\frac{n}{2}\frac{k^2}{4n^2}} = e^{\frac{-k^2}{8n}}$$

- We now get that

$$\Pr(Y_1 \geq 8\sqrt{n\log n}) \leq e^{\frac{-k^2}{8n}} = e^{\frac{-64n\log n}{8n}} = e^{-8\log n} = n^{-8}$$

- The probability of the first element being large ($\geq \sqrt{n\log n}$) is upper bounded by n^{-8}.
- By symmetry, (since Y_1 can be negative), we get

$$\Pr[Y_1 \leq d] \leq n^{-8}$$

- Combining both of these, we recieve

$$\Pr[|Y_1| \geq \sqrt{8n\log n}] \leq 2n^{-8}$$

- $E_i \equiv |Y_i| \geq 8\sqrt{n\log n}$. What we want is to bound the event $E \equiv \cup_{i=1}^{n} E_i$
- We use Boole's inequality to simplify the union computation:

$$\Pr[E_1 \cup E_2 \cdots \cup E_n] \leq \Pr(E_1) + \Pr(E_2) + \ldots \Pr(E_n)$$

  Apply the aboce to get that with probability

$$n \cdot \frac{2}{n^8} = \frac{2}{n^7}$$

  we will exceed the required bounds.

- So, the probability of **not exceeding** the bound will be

$$1 - \frac{2}{n^7}$$

- Note that this is considered as **with high probability** ($w.h.p$):

$$1 - \frac{1}{poly(n)}$$

14.2 Randomization plus algebra — Fingerprinting

Let U be a universe of objects, and $x, y \in U$. We want to ask $x =_? y$. One can answer this using $\log |U|$ bits deterministically.

However, consider mapping elements of U into a sparse universe V, such that

$$x =_? y \iff V(x) =_? V(y) \text{ (with high probability)}$$

.

These images in V are called *fingerprints*.

14.2.1 A concrete use: Marix product verification

Let F be a field, let A, B be two matrices with entries from F. Suppose that it is claimed that $C =_? A \cdot B$. The fastest known matrix multiplication is slower that $O(n^2)$. The algorithms are difficult to implement.

There is a simpler *randomized algorithm*. Let r be a vector whose entries are chosen uniformly at random from $\{0, 1\} \subset F$.

We now check

$$Cr =_? ABr$$

And we claim that this is a good fingerprint for $AB =_? C$.

Suppose $AB \neq C$. In that case, we will show that

$$\Pr(ABr = Cr \leq \frac{1}{2})$$

Consider the matrix $D \equiv AB - C$. Since $AB \neq C$, matrix D is not the zero matrix. We are interested in the event that $Dr = 0$.

Assume without loss of generality that the first row of D has a nonzero entry, and all nonzero entries in that row are before any zero entry. (This is WLOG since we can always move the rows up and down. Similarly, we can move the columns up and down. This will not change the null space **TODO: prove this**).

Consider the first row of D, and the scalar obtained by multiplying the first row of D with r. The result is 0 iff

$$v_1 = \sum_{i=1}^{n} D_{1i} r_i$$

k is the index of the rightmost non-zero entry, $k \leq n$

$$v_1 = \sum_{i=1}^{k} D_{1i} r_i$$

$$v_1 = 0 \iff r_1 = -\frac{\sum_{i=2}^{k} D_1 i r_i}{D_{11}}$$

The probability of $Dr = 0$ is upper bounded by this event ($v_1 = 0$).

To compute the above probability, imagine that $r_2, \ldots r_k$ have been frozen. now, check the probability of picking an r_1 such that $r_1 = \ldots$. If we did, then we infer that $C = AB$, while this is actually not true! So, our **error** is the probability of us picking $r_1 = \ldots$.

This proof technique is called as the **principle of deferred choices**.

Note that the right hand size is a scalar from the field F. The left hand side(r_1) is uniformly chosen amongst two values in F. The required probability therefore **cannot exceed 1/2**.

So, in t trials, the probability that all t trials fail will be $\frac{1}{2}^t$.

For $t = O(\log n)$, the failure probability is polynomially small.

Chapter 15

Applications of Tail Inequalities - 2

15.1 Polynomial verification

check that $P_1(x)P_2(x) =_? P_3(x)$

If both polynomials have degree n, we can make it work in $n \log n$ using FFT. We will design an algorithm faster than this.

- Let $S \subset F$ be a subset of size at least $2n + 1$.
- We evaluate $P_1(s)P_2(s)$, and $P_3(s)$ for $s \in S$, s chosen uniformly at random (using Horner's method, this is $O(n)$ per point). The evaluations are the fingerprintts.
- Clearly, if $P_3(x) = P_1(x)P_2(x)$, this item will not make a mistake. This algorithm *makes a mistake* if $P_3(x) \neq P_1(x)P_2(x)$, but the points we have in S fail to catch this.
- The probability that this makes a mistake: We create a new polynomial

  $$Q(x) \equiv P_3(x) - P_1(x)P_2(x)$$

  It's degree is at most $2n$. If $P_3(x) \neq P_1(x)P_2(x)$, then $Q(x)$ is a nonzero polynomial.
- The polynomial $Q(x)$ has at most $2n$ roots. So, The probability that $Q(r) = 0$ has probability $2n/|S|$, which is the probability of the error.
- We can make the error rate polynomially small in n by using repeated trials, or by picking a larger S.

This technique is useful when we don't have the polynomial directly available. For example, maybe we are only given oracle access to evaluation. For example, **permenant of a matrix**, apparently.

15.2 Definitions to classify the kinds of error

For both the algorithms considered, when the inputs are identical, the algorithm does not make an error.

In inputs that are not identical, we make an error that is bounded by a constant.

15.2.1 The class RP

RP is the class of languages L such that there exists a randomized algorithm A running in **worst case polynomial time**, such that for any input x:

$$x \in L \implies \Pr(A \text{ accepts } x) \geq \frac{1}{2}$$
$$x \notin L \implies \Pr(A \text{ accepts } x) = 0$$

The example was the IP proof for graph non-isomorphism.

15.2.2 The class co-RP

co-RP is the class of languages L such that there exists a randomized algorithm A running in **worst case polynomial time**, such that for any input x:

$$x \in L \implies \Pr(A \text{ accepts } x) = 0$$
$$x \notin L \implies \Pr(A \text{ accepts } x) \leq \frac{1}{2}$$

15.2.3 Reflection on RP and co-RP

The algorithms that we studed are the complement. We make no error on strings in the language, but we can have an error on strings that are not in the language. So, the algorithms we studied are co-RP .

These are considered Monte-Carlo algorithms.

15.2.4 ZP / Las Vegas algorithms

ZP contains languages L such that there is a randomized algorithm A that always outputs the correct answer in **expected polynomial time**. These are also called as Las Vegas algorithms.

15.3 Proof by existence / Probabilistic method

(Refer to Chapter 5, Motwani and Raghavan)

Many a times, we want to show that a particular combinatorial object exists. It maybe very inefficient to build, because of a huge space and a small target of interest, like finding a needle in a haystack. Randomization can come to the rescue here:

- If a random variable X has an expected value $\mathbb{E}\big[X\big] = a$, then there exists a realisation of X with a value $\geq a$ and a realisation with value $\leq a$.
- If arandom objects from some universe of objects has some property P with nonzero probability, then there must exist an object with that property P in this universe.

15.3.1 Example 1

Consider a graph G. We want to find a subgraph G' of G such that it is bipartite, and has the largest number of edges of G (largest bipartite subgraph of G).

We will show the existence of a G' with $|E(G')| \geq |E(G)|/2$.

(We can use this technique to recursively break the graph into $\log n$ bipartite subgraphs, and many algorithms work well on bipartite graphs)

The randomized algorithm to produce this subgraph G' is easy. We assign a bit $b(v)$ to each $v \in V(G)$. Put all vertices in G'. An edge $(u, v) \in G' \iff b(u) \neq b(v)$. The resulting graph looks bipartite with the two bipartite regions consisting of all vertices $b(v) = 0, b(v) = 1$.

Notice that

$$
\begin{aligned}
V_0 &\equiv \{v \in V \mid b(v) = 0\} \\
V_1 &\equiv \{v \in V \mid b(v) = 1\} \\
G' &\equiv (V_0 \cup V_1, (V_0 \times V_1) \cap E(G))
\end{aligned}
$$

$$
\begin{aligned}
X_{uv} &\equiv uv \in E'(G) \\
\mathbb{E}\big[X_{uv}\big] &= \Pr(uv \in E'(G)) = \frac{1}{2}
\end{aligned}
$$

$$
\begin{aligned}
X &\equiv \sum_{u,v \in E(G)} X_{uv} \\
\mathbb{E}\big[X\big] &= \mathbb{E}\big[\sum X_{uv}\big] = \sum \mathbb{E}\big[X_{uv}\big] = |E(G)|/2
\end{aligned}
$$

So, there must exist as assignment b that constructs a bipartite graph with the required properties.

15.3.2 Expander graphs

We start defining an (α, β, n, d) expander.

A bipartite graph $G = (V_1 \cup V_2, E)$ on n nodes is an (α, β, n, d) expander iff:

- Every vertex in V_1 has degree *at most* d.
- For any subset S of vertices from V_1, such that $|S| \leq \alpha n$, then there are *at least* $\beta|S|$ neighbours in V_2.

(Sid: Ideally, d should be very small, and β should be as large as possible. Max number of neighbours will be $d|S|$. That is $\beta \leq d$)

To build such graphs in a deterministic manner is not easy. We can construct these randomized — For example, consider $d = 18, \alpha = 1/3, \beta = 2$

Construction

Let each vertex in V_1 choose d neighbours in V_2 by sampling independently and uniformly at random. We can **sample with replacement** (can pick the same thing repeatedly — each vertex is independent). However, we will consider only one copy of any multiple choice.

Consider any subset S of V_1 with $|S| = s$. Let T be **some fixed subset** of V_2 of size $< \beta|S|$. Consider the event that *all neighbours of* S are in T.

This has probability:

probability of all neighbours lying in T
We simplify this by assuming that $|T| = \beta|S|$
$= (\text{Prob. of picking an element in T}^{\text{number of neighbours}})^{\text{number of vertices}}$
$= (\beta|S|/n)^{d|S|}$

$E_{ij} \equiv \Pr(\text{There exists some } S_i \text{ and some } T_j \text{ such that all neighbours of } S_i \text{ are in } T_j)$
$\Pr(\cup E_{ij}) \leq \sum \Pr(E_{ij})$

There are nCs ways to choose S and $nC\beta s$ ways to chose T.
The probability that for some S, all of its neighbours are in T is upper bounded by

$$nCs \; nC\beta s \; (\beta s/n)^{ds}$$

Stirling's approximation: $nCk \leq (en/k)^k$
Simplifying, what we get is:

$$(en/s)^s \cdot (en/\beta s)^{\beta s} \cdot (\beta s/n)^{ds}$$

Plug in and simplify. What we will eventually get is that it is upper bounded by $\frac{1}{2}^s$.
Next, we should range over all sizes of $s = 1 \ldots \alpha n$, which will give us

$$Total = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots \leq \frac{1}{2}$$

Proof of stirling's approximation

TODO: sid, should fill up

Chapter 16

Proofs by existence

We continue to observe proofs by existence.

16.1 Example 2 of expanders

We have a bipartite graph $G \equiv (L, R, E)$ such that:

- $|L| = n$
- $|R| = 2^{\log^2 n} \equiv n^{\log n}$
- Every susbet of $\frac{n}{2}$ vertices of L has at least $2^{\log^2 n} - n$
- no vertex of R has more than $12 \log^2 n$ neighbours

We want to show the existence of such a bipartite graph by existence.

Let every vertex of L choose d neighbours in R independently, uniformly at random. Choices are made with replacement. Repeat edges are merged into a single edge.

We want every vertex in L to pick d vertices of R. To aid this computation, let us imagine $L' = nd$, such that each vertex in L' has d copies of vertices of L. Now, each vertex in L' picks one vertex in R. Expected number of edges for each vertex in R will be $\frac{nd}{r}$

$$\Pr(deg(v) \geq 12 \log^2 n) = \Pr(deg(v) \geq (1+5)12 \log^2 n) \leq e^{-2 \log^2 n}$$

Derivation in Motwani and raghavan

We want to compute size of neighbour set for a given subset of L:
$S \subset L$, $|S| = \frac{n}{2}$, $T \subset R$

$$\Pr(N(S) \subset T) \leq \frac{|T|}{r}^{|S|d} = (1 - \frac{n}{2^{\log^2 n}}^{\frac{nd}{2}}$$

$$\Pr(\exists S \exists T N(S) \subset T) \leq \binom{n}{|S|}\binom{r}{|T|}(1 - \frac{n}{r})^{\frac{nd}{2}}$$

..

16.2 CNF and MAXSAT

In CNF, each clause is a disjunction of literals. The formula is a conjunction of clauses. CNF $\equiv$ product of sums

We show that for m clauses, there is a truth assignment that satisfies at least $\frac{m}{2}$ clauses.

- Consider a random assignment of truth values to variables
- consider a clause C_i of k variables
- C_i is not satisfied with probability 2^{-k}
- Define the random variable $Z_i \equiv C_i$ is sat
- $\mathbb{E}\big[Z_i\big] = \Pr C_i \text{ is sat} = 1 - 2^{-k}$
- Let $Z \equiv$ number of clauses satisfied, $Z = \sum_i Z_i$
- $\mathbb{E}\big[Z\big] = \sum_i \mathbb{E}\big[Z_i\big] = m(1 - 2^{-k}) \geq m/2$ as $k \geq 1$

The problem to maximise the satisfying clauses is called as MAXSAT .

For any problem instance I, define $m^*(I)$ to be the maximum number of clauses to be satisfied. Let $m(I)$ be the expected number oc clauses that can be satisfied by a randomized algorithm A

the ratio $\frac{m^A(I)}{m^*(I)}$ is the performance of the algorithm A.

We seek algorithms for whom this ratio is close to 1. The previous approach establishes that $\frac{1}{2}$ can be the ratio.

We will establish an algorithm with ratio $\frac{3}{4}$.

16.2.1 ILP formulation

We write the problem as an ILP problem, we then relax the ILP to an LP. We round the solution from LP to get integrality constraints. We will loose some quality at this step.

Consider a clause C_i. An indicator variable $Z_i \in \{0, 1\}$ is used to define whether C_i is satisfied or not. We need to maximise $\sum_i Z_i$

For each variable $x_j \in \{T, F\}$, create an indicator variable $y_j \in \{0, 1\}$

Since each variable can appear in pure or complemented form, we reason about these separately.

$$C_{i+} \equiv \text{indeces of variables that appear in pure form in } C_i$$

$$C_{i-} \equiv \text{Indeces of variables that appear in complemented form in } C_i$$

$$\begin{aligned} &\text{Maximize } \sum_i Z_i \\ &\text{Subject to } \sum_{j \in C_{i+}} y_j + \sum_{j \in C_{i-}} (1 - y_j) \geq z_i \\ &y_i, z_i \in \{0, 1\} \end{aligned}$$

Next, in the ILP relaxation, we allow $y_i, z_i \in [0, 1]$.

We will use $u_i \sim y_i$, where u_i is the LP relaxation, and $v_i \sim z_i$, where v_i is the LP relaxation.

Notice that $\sum_i v_i$ is an upper bound on the number of clauses to be satisfier. But u_i is not integral, so they don't correspond to truth assignments.

Key Idea: We set y_i to 1 with probability u_i.

16.2.2 Algebra to show that we did good, kid

We now estimate the probability that C_i is satisfied: We will show that a clause C_i with k literas is satisfied with probability $1 - (1 - 1/k)^k v_i$

Let us assume that wlog all variables in C_i appear in the pure form:

$$C_i = x_1 \vee x_2 \vee \ldots x_k \sim \text{LP: } (u_1 + u_2 + \ldots u_k \geq v_i)$$

Note that C_i is unsat if $x_1, x_2, \ldots x_k = 0$. So, C_i is unsat with probability $\prod_j (1 - u_j)$. Hence, C_i is sat with probability $1 - \prod_j (1 - u_j)$

We claim that the above problem for C_i is minimized when $u_j = v_i/k$, for each j. (take exponent of sat probability and calculus)

So, the probability of interest is $p = 1 - (1 - v_i/k)^k$. We now claim that

$$p(k) \geq 1(1 - 1/k)^k z \quad \forall z \in [0, 1]$$

. We need to use convexity.

16.2.3 Next steps

We have algo 1 that guarantees an approximation ratio of at least $\frac{1}{2}$. algo 2 guarantees an approximation ratio of $1 - (1 - 1/k)^k$.

We can now combine them, by either running both algorithms and then taking the max, or deciding which algorithm to use by using a random fair coin.

In both cases, we can show that the expected performance raio to be at least 3/4

Chapter 17

Approximate Counting

The idea is to see how many objects satisfy a given condition.

- Count the number of spanning trees of a graph (matrix-tree theorem)
- Count the number of matchings of a given graph (permenant of a graph)
- Count the number of truth assignments that satisfy a formula in DNF (sum of products)

17.1 Counting truth assignments in DNF

DNF syntax is $DNF \equiv C_1 \vee C_2 \ldots C_n$ where each $C_i \equiv L_1 \wedge L_2 \ldots L_j$. $L \equiv$ `literal`, each literal is either a variable X_k, or its negation X_k'. Problem has uses in network design and reliability.

A truth assignment is said to satisfy a DNF boolean formula F, if some clause in F is true in the assignment. We are interested in $\#F$ – the number of distinct satisfying assignments for F.

17.2 DNF counting — Problem formalization

Let U be a finite set of truth assignments, and a boolean function $f : U \mapsto \{0, 1\}$. Define $G \equiv \{u \mid f(u) = 1\}$. We assume that given $u \in U$, $f(u)$ is easy to compute. We also assume that it is possible to sample uniformly at random from U. We want to estimate the size of G.

Note that the requirement that we can uniformly sample from U is quite important! We need an explicit encoding of the object to be able to sample efficiently and uniformly at random.

17.2.1 Solution — Monte Carlo

Sketch

Choose N independent samples from U, say $u_1, u_2, \ldots u_N$. Evaluate f at each one of these samples, and count the number of satisfying instances. Use this count to estimate G.

Details

Let

$$
\begin{aligned}
&Y_i \equiv \text{random variable, indicates if } u_i \in G \\
&Y_i = \begin{cases} 1 & f(u_i) = 1 \\ 0 & f(u_i) = 0 \end{cases} \\
&\mathbb{E}\big[Y_i\big] = \Pr\left[Y_i = 1\right] = \frac{|G|}{|U|}
\end{aligned}
$$

Define another random variable:

$$
\begin{aligned}
&Z \equiv \text{Size that we guess based on } Y_i \\
&Z = |U| \cdot \frac{\sum_i Y_i}{N} \\
&\mathbb{E}\big[Z\big] = \frac{|U|}{N}\mathbb{E}\big[\sum_i Y_i\big] = \frac{|U|}{N}\sum_i \frac{|G|}{|U|} = |G|
\end{aligned}
$$

We want the value of Z to be close to $|G|$. Once again, we can use Chernoff bounds to approximate the tail inequality:

$$
\begin{aligned}
&\epsilon \equiv \text{confidence of the estimate} \\
&Y \equiv \sum_i Y_i \\
&r \equiv \frac{|G|}{|U|},\ N \equiv |U|,\ Nr = |G| \\
&\Pr\left[(1-\epsilon)|G| \le Z \le (1+\epsilon)|G|)\right] \\
&= \Pr\left[(1-\epsilon)|G| \le \frac{|U|Y}{N} \le (1+\epsilon)|G|\right] \\
&= \Pr\left[(1-\epsilon)Nr \le Y \le (1+\epsilon)Nr\right] \\
&= 1 - \Pr\left[Y \ge (1+\epsilon)Nr\right] - \Pr\left[Y \le (1-\epsilon)Nr\right] \\
&\ge 1 - 2e^{-\epsilon^2 Nr/4}
\end{aligned}
$$

Let $\delta \equiv 2e^{-\epsilon^2 Nr/4}$. We want δ to be small. So, we need N and r to be large for δ to be small. For this to work, since $r \equiv |G|/|U|$, we need $|G|$ to be large. This means that we need a large number of truth assignments for this to work.

17.2.2 Importance sampling

Do not rely on uniform sampling, so we use skewed sampling. Skew towards useful samples. We want to increase $r \equiv |G|/|U|$.

Let V be a finite universe. We are given m subsets $H_1, H_2, \ldots H_m$ such that:

- $\forall i,\ |H_i|$ can be computed in polynomial time.
- It is possible to sample uniformly at random from any H_i.
- For all $v \in V$, we can determine if $v \in H_i$ in polynomial time.

The goal is to estimate $H \equiv \bigcup_i H_i$.

In the equivalence to DNF, V is all possible assignments, each H_i is the assignments that satisfy C_i. $|H_i| = 2^{n-\text{number of clauses not in } C_i}$.

Set for sampling during importance sampling

Define a multiset U of the union of H_is where the multiset keeps multiple copies of each element. Alternatively, we can define it as:

$$U \equiv \{(v,i) \mid v \in H_i\}$$
$$|U| = \sum_i |H_i| \geq |H|$$

We also define for every $v \in V$, the coerage set of v as $cov(v,U) \equiv \{(v,i) \mid (v,i) in U$. The size of $cov(v)$ is the number of H_is that contain v_i.

Properties of *cov*

- The number of coverage sets is exactly $|H|$
- The coverage sets partition U, ie, $U = \bigcup_{v \in H} cov(v)$
- $|U| = \sum_{v \in H} |cov(v)|$
- for all $v \in H$, $|cov(v) \leq m$

Deifne a function $f : U \mapsto \{0,1\}$ as $f((v,i)) = 1$ if $i = \min\{j \mid v \in H_j\}$, and 0 otherwise.
Define $G \equiv \{(v,j) \in U \mid f((v,j)) = 1\}$. Notice that $|G| = |H|$.

Sid viewpoint

We create the set U which is a set of $(value, pos)$. We then quotient by an equivalence relation $(v_1,p_1)\ (v_2,p_2) \iff v_1 = v_2$. Then, we pick a distinguished/canonical element per equivalence class, called $(v_i, smallest)$, where *smallest* is the smallest index among all the v_i.

Also notice that $r = \frac{|G|}{|U} \geq \frac{1}{m}$, since:

$$|U| = \sum_{v \in H} |cov(v)| \leq \sum v \in Hm = m|H| \leq m|G|$$

Now, we use monte-carlo to sample U. Same calculations hold now. Now, we need to sample uniformly at random from U.

- Pick an index j with probability $\Pr[i \text{ is chosen}] \equiv |H_i|/U$.
- Now, pick an element uniformly at random from H_i.
- **(Is there some geometric intuition for this picking process?)**

Now, this pair (v, i) is now uniformly at random over U **(TODO: finish proof)**

17.2.3 Some definitions

Polynomially randomized approximation scheme (PRAS)

Given a counting problem Π, A is a randomized algorithm which runs in polytime such that on any instance I of P, and a real number $\epsilon > 0$, it produces an output $A(I)$ such that

$$\Pr\left[(1-\epsilon)\#I \leq A(I) \leq (1+\epsilon)\#I\right] \geq \frac{3}{4}$$

Fully randomized approximation scheme (FPRAS)

Runtime s bounded by a polynomial in both n and $\frac{1}{\epsilon}$

P

A problem is in $\#P$ if the corresponding decision problem is in `NP` .

A problem is $\#P$ if there is an NDMP such that for any problem instance I, it has a number of accepting computations that is equal to the number of distinct solutions to instance I.

$\#P$-complete problems are those which can be reduced in polytime. DNF counting is $\#P$ complete.