CS 466
F13

By: Shale Craig
Prof: Anna Lubiw

Adapted From Notes In Class and External Sources

February 25, 2014

Contents

Chapter 1

Introduction

In this course, we want to solve algorithmic problems, and compounding general knowledge with developments of the last 30 years. Though these notes are grouped by lecture, this course can be split into three sections:

1. Algorithmic Design
   - Assumed knowledge of greedy, divide & conquer, and dynamic programming techniques
   - Introduction to randomization, approximation, and online algorithmic techniques
2. Algorithmic Analysis
   - Assumed knowledge of big O, worst case asymptotic analysis techniques
   - Introduction to amortized analysis, probabilistic analysis, and approximation factors methods
3. Lower Bounds
   - Understanding of NP-COMPLETE-ness is assumed
   - Hardness of approximation is introduced

The class website is `https://www.student.cs.uwaterloo.ca/~cs466/`.

1.1 TSP

1.1.1 Problem Statement

Given a graph $G = (V, E)$ with weights on edges $w : E \to R$, find a TSP tour[1], which is a Hamiltonian Tour[2] C that visits each vertex exactly once and has minimum weight:

$$\sum_{e \in C} w(e)$$

[1] A tour is a traversal ordering of vertexes in a graph

[2] A Eulerian Tour is a tour that goes through each vertex exactly once and returns to the first vertex.

Since we can add infinite-weight edges to any non-complete graph, we assume that we have a complete graph[3].

TSP is a known NP-COMPLETE problem.

1.1.2 NP-COMPLETE

To show a problem R is NP-COMPLETE, we need to show that both:

1. Prove R is in NP.
2. Give a reduction (denoted "$\leq_p$") from a known NP-COMPLETE problem to an instance of the R problem.

Thus to prove that TSP is NP-COMPLETE, we need to show both:

- TSP is in NP
- Hamiltonian Cycle $\leq_p$ TSP.

Assuming that Hamiltonian Cycle in our reduction is a known NP-COMPLETE problem, we now know that TSP is NP-COMPLETE too.

Unless P = NP, we need to choose two of these three options:

- A *speedy* algorithm
- Solve the *problem precisely*
- Solve a *hard problem*

NP-COMPLETE problems are hard, so we must choose between *precision* and *speed* when solving them.

1.2 Metric TSP

We define Metric TSP as a weaker variant of TSP where the distance between two vertexes is the same in either order, and distances always follows the triangle inequality.

$$d(u, v) = d(v, u)$$
$$d(u, v) \leq d(u, \beta) + d(\beta, v)$$

There exists a fast approximation algorithm that exists for this problem:

- Find minimum-spanning tree (see Chapter 12 for more information) of the graph using Kruskal's Algorithm ($O(m \log n)$).

[3] A complete graph is one where there exists an edge between all vertexes in the graph.

- Take a tour walking around the tree, taking shortcuts to avoid re-visiting vertexes
- The distance added by the shortcuts is less than or equal to twice the distance of remaining in the tree.

Here's a quick proof to this lemma: In a given graph, call ℓ the length of the tour of our algorithm, and call ℓ_{TSP} the distance traveled in the minimal TSP tour.

We want to prove that $\ell \leq 2\ell_{TSP}$, so that this is a 2-approximation (For more information, see Chapter 13).

We know that $\ell \leq 2\ell_{MST} \leq 2\ell_{TSP}$ since deleting one edge of a minimum TSP tour gives us a spanning tree.

Sidenote: There exists a 1.5 approximation, but there were no specifics given in class. The apparently, the idea is to use a matching algorithm.

1.2.1 Implementation

- We must find a minimum spanning tree, which takes $O(m \log n)$ time using Prim and Kruskal's Algorithm.

  m is the number of edges.

  n is the number of vertexes.

- We will see improved heaps in this class that allow us to take the time down to $O(m + n \log n)$

This method doesn't work for General TSP, since if there exists a k-approximation, then P = NP[4].

1.3 Types of TSP

In general, the ordering is from hardest to easiest:

General TSP $\geq$ Metric TSP $\geq$ Euclidean TSP

General is the basic "general" TSP problem.

Metric is described above [5].

Euclidean is where vertexes are placed on a plane, and the weight of edges is the euclidean distance between vertexes [6].

[4] An approximation for this would solve the Hamiltonian-Cycle decision problem in P time. The Hamiltonian Cycle decision problem is known to be NP-COMPLETE.

[5] There is a (1.5)-approximation algorithm, but the minimum bound is 1.0045.

[6] There is an $(\epsilon + 1)$-solution for an $\forall \epsilon > 0$, but time grows as ϵ decreases. This is a Polynomial Time Approximation Scheme. See Chapter TODO for PTASs.

Part I

Data Structures

Every algorithm must store, access, or search data. We look at amortized analysis and more complicated data structures these data structures.

It's assumed knowledge of the following:

- Heap-based priority queue
- Dictionaries built using hashing, and balanced binary search.

Chapter 2

Binomial Heaps

2.1 Heaps

Heaps store are binary trees of elements, each with an numeric key. For a minimum heap, the minimum key is the root. Usually, we shape our heap as a near-perfect triangle, so we can store an array in level order and use indexing instead of pointers. The height of a heap is $\theta(\log n)$.

We have five main operations in priority queues:

Insert inserts at the bottom, and bubbles up - $\theta(\log n)$

Delete Minimum remove the root element, put the last item there, and bubble down. - $\theta(\log n)$

Decrement Key bubble the item up (or down) - $\theta(n)$

Build can be done faster than repeated insertion - $\theta(n)$

Merge merge two heaps into one heap - $\theta(n)$

2.2 Prim's Algorithm

Prim's Algorithm finds the **Minimum Spanning Tree** of a graph.

Start with one node $s \in V$.

```
remaining = V
while |remaining| > 0:
    e_connecting, v_connected = heap.popMin()
    answer.put(e_connecting)
    remaining.remove(v_connected)
    heap.updateWeights(v_connected)
return answer
```

We define n as the number of edges, and m is the number of edges. $m = O(n^2)$.

We can implement this two ways.

- If we store a heap of edges ordered by weight, this takes nDeleteMin + m(Insert + Delete) = $O((n + m) \log m)$ time.
- If we store a heap of vertexes, this takes nDeleteMin + mDecreaseKey = $O((n + m) \log n)$ time.

For now, these are the same[1]. Once we implement different heaps, this difference becomes relevant.

2.3 Binomial Heap

We can improve the merge speed of our "standard" heaps by using pointers (where each node has any number of children) instead of an array to implement the heaps.

We keep the heap order, but we go beyond the fact that the heaps are made of binary trees.

2.4 Binomial Tree

We define binomial trees as follows:

The root of B_0 is a single vertex with no children.

The root of B_1 is the root of a B_0 with an additional B_0 root vertex attached.

The root of B_2 is the root of a B_1 with an additional B_1 root vertex attached.

The root of B_k is the root of a B_{k-1} tree with an additional B_{k-1} root vertex as a child of the root vertex.

There are a few properties of Binomial Heaps we know about[2]:

Size of a B_k is 2^k.

Height of a B_k is k, defined in the number of edges from the root to a leaf vertex.

Width of level i of a B_k is $\binom{k}{i}$).

This is since $\binom{k}{i} = \binom{k-1}{i} + \binom{k-1}{i-1}$

2.5 Binomial Heaps

To create binomial trees of arbitrary heights, we need to start using forests of binomial trees.

We can represent $n = 13 = 0b1101 = 2^0 + 2^2 + 2^3$ elements as a B_0, B_2, and a B_3.

In general, for n elements, use $\log n$ trees.

Most of our operations on this will be through a series of merges.

[1] We know this, since $m = O(n^2)$ implies that $\log m = O(\log n)$, within a constant

[2] All properties in this list can be proved by induction.

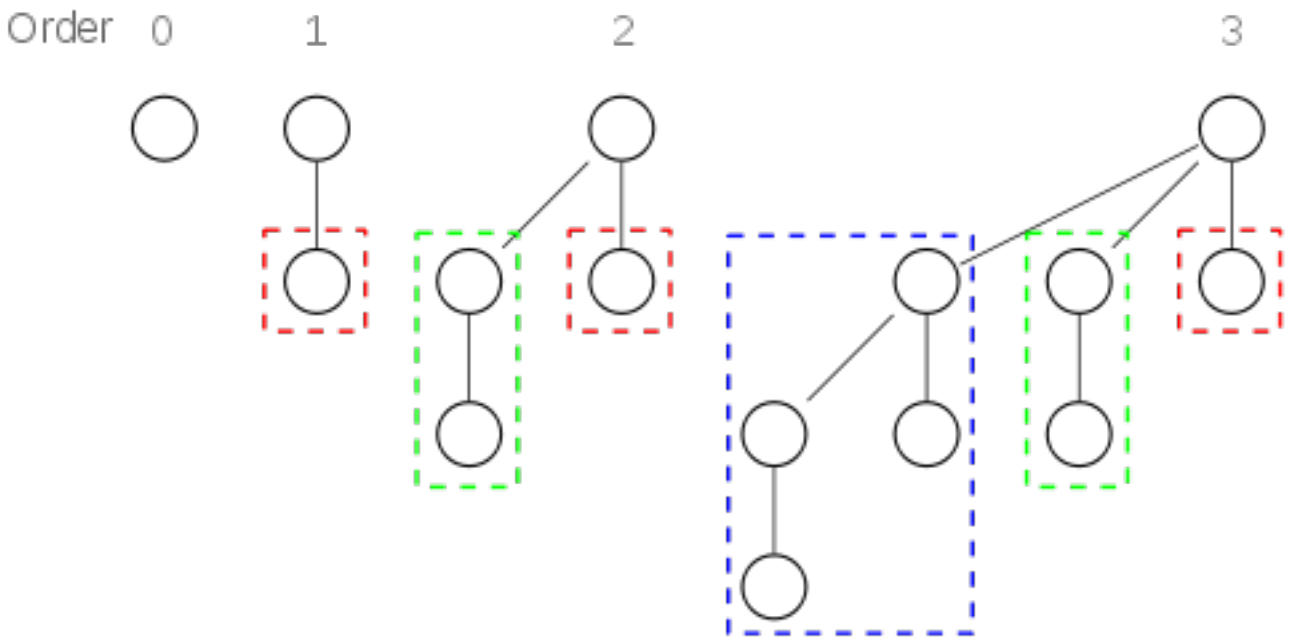

Figure 2.1: Sample Binomial Heaps

Merge works like binary addition ($\theta(1)$) across the trees, so the cost is the same as the bit cost of addition - $\theta(\log n)$

Insert is a merge of the pre-existing forest and a B_0 tree - $\theta(\log n)$ worst-case[3]

Delete Minimum is done by finding the smallest tree, breaking removing the root vertex, and merging those to the remaining untouched trees - $\theta(\log n)$

Decrease Key is done inside a binomial tree as a standard "bubble up", so the effect is limited to the height of the individual - $\theta(\log n)$

Build Binomial Heap can be done by repeated insertion in $O(n)$ time.

[3]TODO: is the amortized time different?

Chapter 3

Amortized Analysis

3.1 Example For Binomial Heaps

Binomial heaps take $O(\log n)$ time to merge. Let's prove that.

We want to determine the bit cost for incrementing a binary counter from 0 to n.

3.1.1 Worst-Case Analysis

The worst-case cost of one increment on a k-bit counter is k, so n increments cost $O(n \log n)$ ($\log n$ bits flipped n times).

3.1.2 Amortized Analysis

We can get a better bound.

- The 2^0 bit flips every time - n
- The 2^1 bit flips every other time - $\frac{n}{2}$
- The 2^2 bit flips every 4th time - $\frac{n}{4}$
- etc.

The total cost is:

$$\sum_{i=0}^{k} \frac{n}{2^i} \leq 2n$$

Thus, the average cost of incrementing a counter is $\frac{2n}{n} = 2$.

Since binomial heap appending is representable with a bit cost of a binary counter, the total cost for making a binomial heap is $O(n)$.

3.2 An Amortized Definition

Given a sequence of m operations with total cost $T(m)$, then the **amortized cost** per operation is $\frac{T(m)}{m}$.

3.3 Potential Method for Amortized Analysis

The idea for this method is that we are keeping an account of (time) cost.

Keeping track of a "potential-time" bank account, we keep track of the amortized difference between true cost of operations and a charge expected for all operations.

We call the bank balance after the ith operation Φ_i.

- Cost is true.
- Charge is artificial.

$$\Phi_i = \Phi_{i-1} + \text{charge}(i) - \text{cost}(i)$$

Since the potential (Φ_i) and charge are artificial, we define them to make analysis easy.

It's much simpler to define potential to get the charge:

$$\text{charge}(i) = \text{cost}(i) + \Phi_i - \Phi_{i-1}$$

If the final potential is $\geq$ than the initial potential, then the amortized cost is $\leq$ max charge.

$$\begin{aligned}
\sum_{i=1}^{m} \text{charge}(i) &= \sum_{i=1}^{m} \text{cost}(i) + \sum_{i=1}^{m} \Phi_i - \sum_{i=0}^{m-1} \Phi_i \\
&= \sum_{i=1}^{m} \text{cost}(i) + \Phi_m - \Phi_0 \\
\Phi_m - \Phi_0 \geq 0 \implies &\sum \text{charge}(i) \geq \sum \text{cost}(i) \\
\text{amortized cost} &= \sum \frac{\text{cost}(i)}{m} \\
&\leq \sum \frac{\text{charge}(i)}{m} \\
&\leq \text{max charge}
\end{aligned}$$

3.3.1 Potential Analysis in a Nutshell

We need to invent a potential(i) and a charge(i) and prove that $\Phi_m \geq \Phi_0$.

A goal when inventing potential and charge is to prove that max charge is small, since the amortized cost is less than or equal to the maximum charged.

$$\text{charge}(i) = \text{cost}(i) + \Phi_i - \Phi_{i-1}$$

3.3.2 Binary Counters Using the Potential Method

We know that only one bit will undergo $0 \to 1$ in a given increment.

The cost is high when $1 \to 0$ occurs many times. Let's pay for $0 \to 1$ and an extra \$1 for when this bit eventually flips $1 \to 0$.

Thus, $\text{charge}(i) = 2$.

By theorem, the amortized cost $\leq$ max charge $= 2$, so long as $\Phi_m \geq \Phi_0$.

Formally, we'd like to specify the relation between $\text{charge}(i)$ and Φ_i.

We make a jump here that Φ_i is the number of 1s in the counter after the ith operation.

Supposing the ith operation changes t_i bits $1 \to 0$, and 1 bit $0 \to 1$.

Then we have:

$$\begin{aligned} \text{cost}(i) &= t_i + 1 \\ \Phi_i &= \Phi_{i-1} - t_i + 1 \\ \text{charge}(i) &= \text{cost}(i) + \Phi_i - \Phi_{i-1} \\ &= t_i + 1 - t_i - 1 \\ &= 2 \end{aligned}$$

Thus, $\Phi_0 = 0$, and $\Phi_m \geq 0$, so the theorem applies.

3.4 Mergeable Heaps

There's a family of heaps who's main operation is a `merge`.

	Binomial Heap	Lazy Binomial Heap	Fibonacci Heap
insert	$O(\log n)$	$O(1)$	$O(1)$
delete min	$O(\log n)$	A $O(\log n)$	A $O(\log n)$
merge	$O(\log n)$	$O(1)$	$O(1)$
decrease key	$O(\log n)$	$O(\log n)$	A $O(1)$
build	$O(n)$	$O(n)$	$O(1)$

3.4.1 Lazy Binomial Heaps

We can improve merge an insert by lazily combining trees during `insert` and `merge` operations. We catch up on work when performing a `delete min` operation to have exactly one tree of each rank.

Implementing Delete-Min

- Look at all roots to find the min
- Delete that root, its children become separate
- Consolidate ranks from smallest to largest

The wost case cost of `delete min` is $\theta(n)$, with n singleton trees.

Amortized Analysis of Delete Min

We theorize that Lazy Binomial Heaps have A $O(\log n)$ cost for `delete min`.

By *magic*, we pick Φ to represent the number of trees. Thus, $\Phi_0 = 0$, and $\Phi_m \geq 0$, so $\Phi_m \geq \Phi_0$.

We know that $\text{charge}(i) = \text{cost}(i) + \Phi_i - \Phi_{i-1}$. Let's examine other operations costs first:

- Merge cost is $O(1)$, since the number of trees is the same.
- Decrease key cost is $O(\log n)$, and the number of trees is the same.
- Insert cost is $O(1)$, since the number of trees increase by one.

In the case of `delete min`, we have the degree $r \in O(\log n)$ of the node being deleted, and $t = \Phi_{i-1}$ as the number of trees being deleted.

Consolidate is called on $t - 1 + r$ trees.

Thus the total cost is $t - 1 + r + O(\log n)$.

After consolidation, we have $\Phi_i \in O(\log n)$.

$$\begin{aligned} \text{amortized cost} &\leq \text{max charge} \\ &\leq \text{cost}(i) + \Phi_i - \Phi_{i-1} \\ &= t - 1 + r + O(\log n) - t \\ &\leq r + O(\log n) \\ r \in O(\log n) &\implies \text{amortized cost} \in O(\log n) \end{aligned}$$

Thus `delete min` for lazy binomial heaps runs in $O(n)$ worst case, but $O(\log n)$ amortized.

3.4.2 Fibonacci Heaps

In these heaps, we want to improve the amortized cost of `decrease key`.

What if instead of bubbling up, we simply "cut off" the node being decreased (and its sub-tree) from its parents?

This is dangerous, since the number of trees increases, and the number of child nodes change (not just 2^i) for details.

TODO: the notes in lecture 3 reference assignment 2 for a practical alternative to Fibonacci Heaps. Dig this up.

Chapter 4

Splay Trees

In a nutshell, splay trees are self-adjusting data structures that alter data structure after each query. They're the tree equivalent to lists that use https://en.wikipedia.org/wiki/Move-to-front_transform to improve lookup times.

4.1 Requisite Knowledge

4.1.1 Dictionaries

These use keys from a totally ordered universe. Operations include:

- Insert
- Delete
- Search

Unbalanced Binary Search Trees

All operations take $O(h)$, where h is the height of the tree.

Balanced Binary Search Tree

We limit $h \in O(\log n)$. There are two (main) implementations: AVL and red-black trees. Both implementations must keep the balance information, and are re-balanced using rotations.

4.2 Regarding Splay Trees

Splay trees were invented (discovered?) by Sleator and Tarjan in '85. They offer A $\theta(\log n)$ cost per operation, with a $\theta(n)$ worst case running time. By not keeping balance information, they become easier

to implement than other conventional balanced trees.

The course notes allude to an example where single rotations do not give good average behavior, so we will do double rotations instead.

4.2.1 Splay Operation

The `splay(x)` operation moves x repeatedly to the root. This occurs through three cases. Refer to Figures 4.1, 4.2, and 4.3.

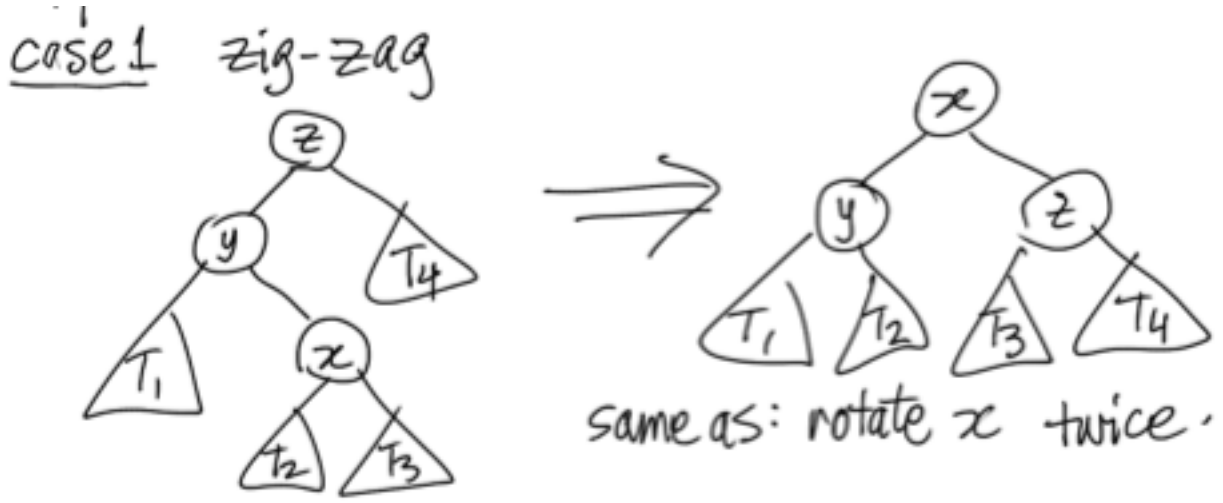

Figure 4.1: Splay Trees Case 1

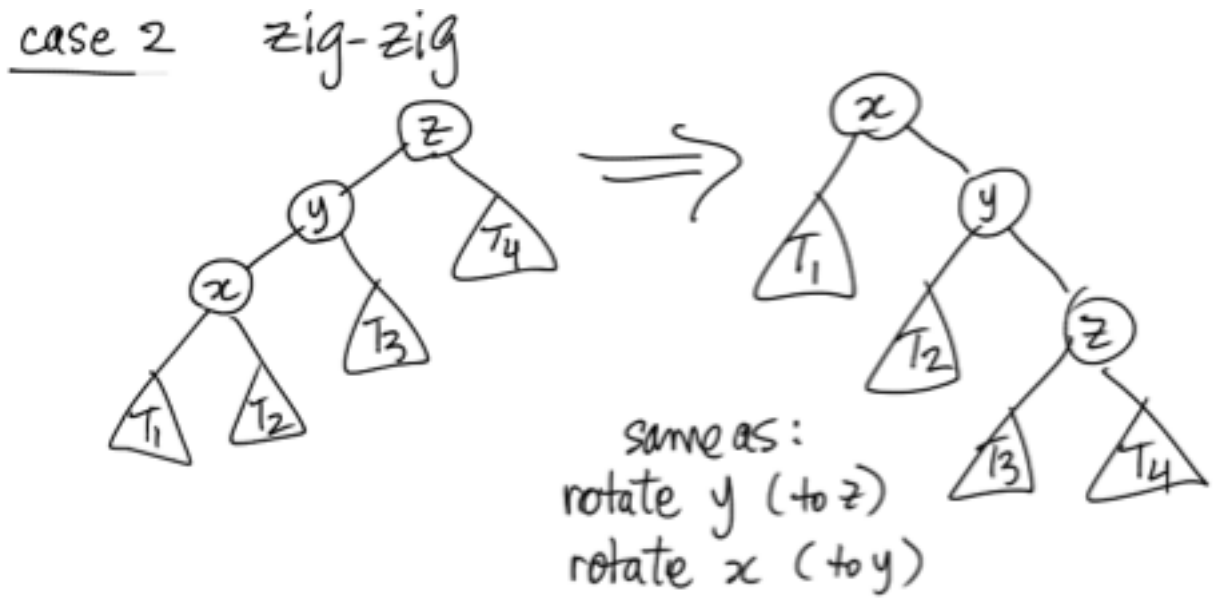

Figure 4.2: Splay Trees Case 2

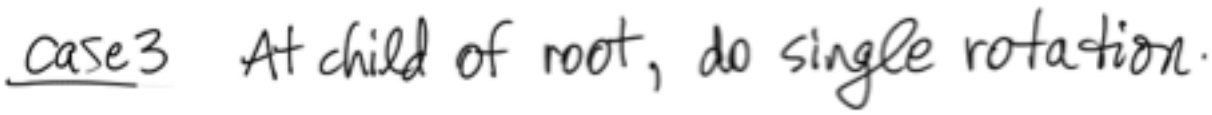

Figure 4.3: Splay Trees Case 3

4.2.2 Splay Tree Methods

Search - after finding x, calling `splay(x)`, even for unsuccessful searches.

Insert - usual binary search tree insert, then we `splay` the new node.

Delete - usual binary search tree delete, then splay the parent of the node being removed.

4.3 Amortized Analysis of Splay Trees

If the height h of a tree is large, then search is expensive, and we pay out of potential.

We define $D(x)$ as the number of descendants of x, including x, and $r(x) = \log(D(x))$. Finally, we define $\Phi(T) = \sum_x r(x)$. By *magic*, we have the max as $\Phi_{\max} = O(\log(n!)) = O(n \log(n))$, and the min as $\Phi_{\min} = O(n)$.

For a single node, we call $r(x)$ the current rank, and $r'(x)$ the rank after calling `splay(x)`.

We claim that the amortized cost of one step of `splay(x)` is:

$$O(\text{splay}(x)) \leq \begin{cases} 3(r'(x) - r(x)) : & \text{for cases 1 and 2} \\ 3(r'(x) - r(x)) + 1 : & \text{for case 3} \end{cases}$$

Note that $(r''(x) - r'(x)) + (r'(x) - r(x)) = r''(x) - r(x)$.

Since $\Phi_i \geq \Phi_0$, we now want to find the amortized cost.

- For case 3, refer to Figure 4.4.

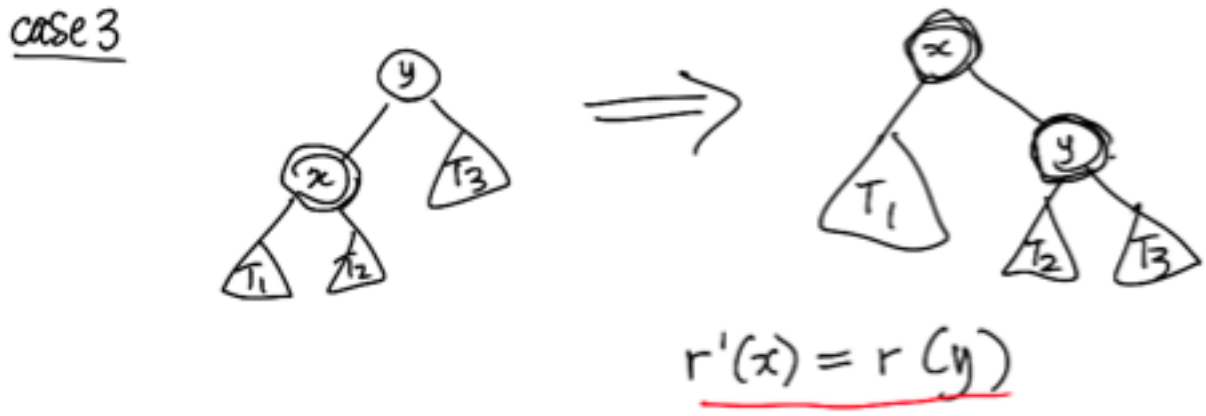

Figure 4.4: Amortized Splay Tree Analysis - Case 3

$$\begin{aligned}
\text{amortized cost} &\leq \text{charge} \\
&= \text{true cost} + \text{change in potential} \\
&= 1 + r'(x) + r'(y) - r(x) - r(y) \\
r'(x) = r(y) &\implies \text{amortized cost} = 1 + r'(y) - r(x) \\
&\leq 1 + r'(x) - r(x) \\
&\leq 1 + 3(r'(x) - r(x))
\end{aligned}$$

- For case 1, refer to Figure 4.5.

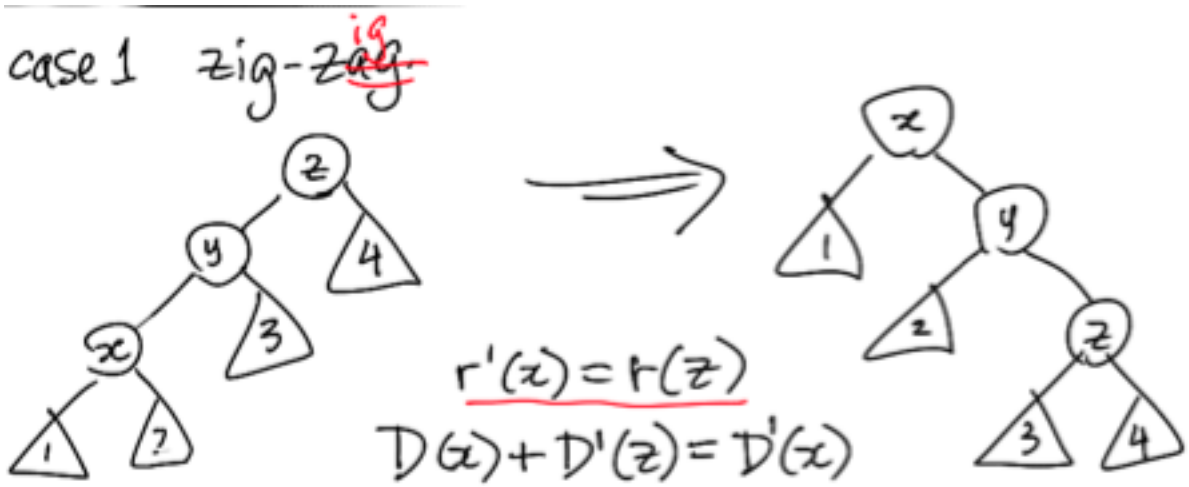

Figure 4.5: Amortized Splay Tree Analysis - Case 1

$$\begin{aligned}
\text{amortized cost} &\leq \text{true cost} + \text{change in potential} \\
&= 2 + (r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)) \\
r'(x) = r(z) &\implies \text{amortized cost} \leq 2 + r'(y) + r'(z) - r(x) - r(y) \\
r'(y) \leq r'(x) \wedge -r(y) \leq -r(x) &\implies \text{amortized cost} \leq 2 + r'(x) + r'(z) - 2r(x)
\end{aligned}$$

To show that $2+r'(x)+r'(z)-2r(x) \leq 3(r'(x)-r(x))$, it is enough to show that $2 \leq 2r'(x)-r(x)-r'(z)$.

$$\begin{aligned}
&\forall x, y > 0 \wedge x + y \leq 1 : \text{Range}(\log x + \log y) = (-\infty, -2] \\
&\implies \forall a + b \leq c \to \log\left(\frac{a}{c}\right) + \log\left(\frac{b}{c}\right) \leq -2 \\
D(x) + D'(z) \leq D'(x) &\implies \log(D(x) + \log(D'(z)) \leq 2\log(D'(x)) - 2 \\
r(x) + r'(z) &\leq 2r'(x) - 2 \\
2 &\leq 2r'(x) - r(x) - r'(z)
\end{aligned}$$

Thus the amortized cost of case 1 is $\leq 3(r'(x) - r(x))$.

- Case 2 is incredibly similar to case 1 with minor (ordering) modifications.

Without proof[1], we claim that a tree T root t and node x, the amortized cost of `splay(x)` is:

$$\begin{aligned} AO(\text{splay}) &\leq 3(r(t) - r(x)) + 1 \\ &\in O(\log \frac{D(t)}{D(x)}) \\ &= O(\log n) \end{aligned}$$

Let $r_i = r(x)$ after the ith step of the splay. So $r_0 = r(x)$, and $r_k = r(t)$ (where k is the final step). Thus the overall amortized cost of splay is:

$$\begin{aligned} AO(\text{splay}) &= 1 + \sum_{i=1}^{k} 3(r_i - r_{i-1}) \\ &= 3(r_k - r_0) + 1 \end{aligned}$$

We know that the cost of walking down the tree in each operation is $\leq$ the cost of the ensuing splay. Thus, we know the amortized cost of `insert`, `search`, and `delete` in a splay tree is $O(\log n)$.

We briefly touched in class that `insert` and `delete` both modify potential, but this is still covered by the $\log n$ work to walk to the inserted and deleted value.

[1] scraig postulates that this can be proved with a telescoping sum

Chapter 5

Union-Find Problem

Connected components in a graph are essentially the components where two can reach each other.

We want to find all connected components, and identify which component a given vertex is in. Let's make this efficient.

In general, we assume we are given a graph G with n vertexes and m edges. We then need to respond to two queries:

find are vertexes a and b in the same component?

union connect the components which vertexes c and d lie in.

Using depth-first search, it takes $O(n+m)$ time to perform `find`, and $O(1)$ time to perform `union`.

5.1 Dynamic Graph Connectivity

For many data structures, we can get much faster runtime by maintaining (and later updating) results as the underlying data changes.

Examples of where this is useful:

- Social networks as relationships are added and deleted.
- Minimum spanning tree[1]
- Kruskal's Algorithm[2]

5.2 Union-Find Data Structure

We want to maintain a collection of disjoint sets then evaluate:

[1] This is a special case of **Incremental Dynamic Connectivity**.

[2] Greedy algorithm that orders edges by weight, then adds them slowly into a tree (if an edge creates a cycle, don't add it). We end up with a minimum spanning tree.

Union(A, B) unites (modifies) the two sets A and B to be in the same set.

Find(e) which set contains e?

If we analyze Kruskal's algorithm using union-find data structure, we get:

$$\text{sort} + 2m\text{Finds} + n\text{Unions}$$

Sort takes $O(m \log m) = O(m \log n)$ time[3], so we want the finds and unions to work in $\leq O(m \log n)$ to have a speedy algorithm.

Define n as the number of elements, and m as the number of operations. For all implementations, the number of unions $\leq n - 1$.

5.2.1 Implementation With an Array

Using an array $S[1...n]$, where $S[i]$ contains the name of a set containing i.

Find $O(1)$

Union $O(n)$ worst case

To make this marginally faster, we can maintain a set for each set as well. Thus, `union(A, B)` will update S for the smaller set. Since each element changes its set name $\leq \log n$ times, the overall cost of all unions is $\leq O(n \log n)$.

The cost of m operations is thus $O(m + n \log n)$. With this implementation, this is the best possible if the number of finds is $\Omega(n \log n)$.

Thus in this case, we get $O((n + m) \log n)$ for Kruskal's algorithm.

5.2.2 A Better Implementation

In the case that the number of finds is small, the array-based union-find implementation is horrible.

When we represent each set as a tree, life becomes much better.

Union is implemented as merging the smaller tree as a child to the root of the larger tree. See Figure 5.1 for a pictorial visualization.

Find is implemented by traversing up the tree from the node, then returning the name of the root node. After traversing upwards, we perform path compression by setting the parent of all vertexes in the path to be the root of this tree. See Figure 5.2 for a pictorial visualization.

We determine the smaller tree by keeping track of the "rank" of a tree - the height if there was no path compression. When `union`-ing a smaller r_2 onto a larger r_1, the the new rank is $\max\{r_1, r_2 + 1\}$.

[3]Since $m \leq n^2$, we have $O(\log m) = O(\log n)$

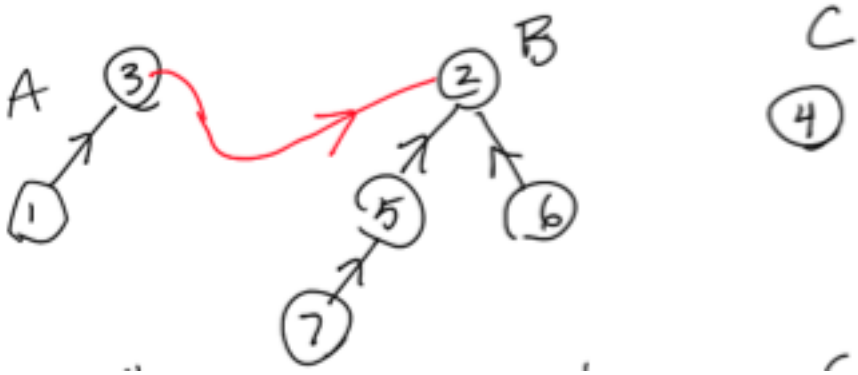

Figure 5.1: "Union" Operation in the Union-Find Data Structure

Figure 5.2: "Find" Operation in the Union-Find Data Structure

5.3 Analysis of the Union-Find Data Structure

The implementation is simple, but the analysis is hard. In '75, Tarjan proved that the cost of m operations is $\Theta(m\alpha(m,n))$ time[4].

We will prove the slightly higher bound of $O(m \log^* n)$ time for m operations[5].

We know that the cost of `find(v)` is the same as the distance from v to the root. In a nutshell, we will charge some to the `find`, and some to the nodes along the path from v to the root, then sum it up.

We claim (without proof) that:

1. $\text{rank}(v) < \text{rank}(\text{parent}(v))$.
2. The number of vertexes of rank r is $\leq \frac{n}{2^r}$ in size[6].

In our analysis, we divide vertexes into groups based on their rank. A vertex of rank r goes in a group

[4] $\alpha(m,n)$ is the inverse Ackerman function. It is very slow growing, and is ≤ 5 for all practical purposes.

[5] $\log^* n$ is essentially the minimum number of times $log(n)$ needs to be recursively called in order for $\log(\log(\ldots(n))) \leq 1$. The inverse of this is $2 \uparrow n$, which is 2 exponentiated with itself n times.

[6] This is because a vertex of rank v has $n \geq 2^r$ descendants, and vertexes of rank r have disjoint descendants.

number $\log^*(r)$. Thus a group g contain the ranks $2 \uparrow (g-1)+1, 2 \uparrow (g-1)+2, \ldots, 2 \uparrow g$. For group g, the number different ranks $c(g)+1$ in g is $\leq 2 \uparrow g$.

Since the largest rank in a structures can be n, the number of groups must be $\leq \log^* n$.

We want to find the charge for `find(v)`: For each vertex u on the path from v to the root:

- if u has a parent and grandparent, and group(u) = group(parent(u)), then charge 1 to u.
- Otherwise, charge 1 to `find(v)`.

Thus the total charge to find(v) $\leq \log^* n + 1$, since the group changes $\leq \log^* n - 1$ times, and 2 more for the root and it's child.

We now need to determine the charge to individual nodes.

If a vertex u in group g is charged, then path compression will give it a new parent of higher rank. Therefore a u in group g is charged $c(g)$ times until its parent is in a higher group. We know that $c(g) \leq 2 \uparrow g$.

The total charge to all nodes in a group g is:

$$\begin{aligned}
(\text{number of ranks in } g)(\text{number of nodes in } g) &= c(g)N(g) \\
N(g) &\leq \sum_{r=2\uparrow(g-1)+1}^{2\uparrow g} \frac{n}{2^r} \\
&\leq \frac{n}{2^{2\uparrow(g-1)+1}} \sum_{i=0}^{\infty} \frac{1}{2^i} \\
&\leq \frac{n}{2 \uparrow g} \\
\implies c(g)N(g) &\leq n
\end{aligned}$$

Thus the total charge to all nodes is $n \log^* n$.

For Kruskal's algorithm, we find the total charge to `finds` and `nodes` as:

$$O(m(\log^* n + 1) + n \log^* n) = O(m \log^* n)$$

Not bad.

Chapter 6

Geometric Data Structures

So far data structures have been implemented with comparable keys.

When working in higher dimensions, we have two main problem types:

- Find points inside a region
- Find regions containing a point

6.1 Range Search

By preprocessing n points in k dimensions, so we can handle range queries. In 2D, this would be querying for points contained within a rectangle.

We have 3 main measures for range search methods:

P the preprocessing time

S the space taken for preprocessing

Q the query time

U the update time (only some algorithms can have updated data)

6.1.1 Range Queries for $k = 1$

When $k = 1$, we sort data and use binary searches. Thus we have:

P $O(n \log n)$

S $O(n)$

Q $O(\log n + t)$, where t is the output size.

U $O(\log n)$

6.1.2 Range Queries for $k = 2$

We have a few cool implementations, most of which are covered in CS240.

Quad Tree

Divide squares into four subsquares, repeat until each square has $(0, 1)$ points.

kd-Tree

Divide points in half vertically then horizontally (then recurse).

P $O(n \log n)$

S $O(n)$

Q $\Theta(\sqrt{n} + t)$, where t is the output size.

Range Trees

See the subsection on Range trees below.

6.1.3 Range Trees for $k = 2$

A nth dimension range tree improves Q at the expense of S. It uses a binary search tree across one dimension, where each internal node has an additional $n - 1$-dimension range tree.

A $k = 1$-dimension range tree is a sorted list.

P sort by x, then sort by y and do some work - $O(n \log n)$

S each point occurs in $\log n$ of the sorted-by-y lists - $O(n \log n)$

Q search for the x_l and x_r in $O(\log n)$ time. For all children of paths to x_l and x_r, we search the y list - $O(\log^2 n + t)$

Fractional Cascading

We can improve Q to $O(\log n + t)$ by using a technique called fractional cascading.

Generally, we keep a pointer from each element in the x's list to the corresponding element in y's list. This gives us $Q = (\log n + t)$, since we binary search once for y_U, y_L in the list of root and follow pointers.

6.2 Point Location

Given a set of disjoint regions in a k-dimensional space, we want to quickly respond to queries that query the location they are in. This can help with queries like: which city is coordinate (a, b) in? Where is the nearest Tim Hortons? etc.

6.2.1 Point Location for $k = 1$ Dimension

In 1d, we use a balanced binary search tree.

P $O(n \log n)$

S $O(n)$

Q $O(\log n)$

6.2.2 Point Location for $k = 2$ dimensions

We can divide the entire space into slabs by adding a vertical line at every point.

Then given a query point x, find the correct slab ($O(\log n)$) then binary search by y ($O(\log n)$).

Q $O(\log n)$

S $\Theta(n^2)$ (ew)

Less Space Through Persistent Data Structures

Given that in one slab to the next, very few changes, we only need to make a BST for the leftmost slab and update for subsequent slabs.

The total number of updates to the BST is $O(n)$, since every segment is inserted and deleted exactly once.

If we update a BST and search it in the past, this idea is called a "persistent data structure".

Partial persistence allows queries in the past and only the present be changed.

Full persistence allows queries and changes at any point in time.

Using Driscoll, . . . , Tarjan '89, we can add partial persistence to any data structure.

This gives us a planar point location of:

P $O(n \log n)$

S $O(n)$

Q $O(\log n)$

In an awesome way, this runs in the same time as the initial 1D problem.

Chapter 7

Randomized Algorithms

Algorithms that use random numbers have their output and/or their runtime depend on random numbers. This forces us to use amortized (expected) analysis.

Practicaly speaking, it gets us easier and faster algorithms. Theoretically speaking, it's OPEN whether randomization helps for P vs NP, but we'll see an example where it'll help a tiny bit.

In previous classes, we've seen QuickSort[1] and SkipLists.

We define randomized algorithms as ones that execute either method `rand[1, ..., n]` or `rand[0, 1]`, both of which run in $O(1)$ time[2].

Thus, the running time for fixed input depends on random numbers - i.e. a random variable.

A few definitions are necessary:

Sample Space is the space of all possible outcomes (for fixed input).

Random Variables map the sample space to real numbers (at runtime).

We need to rely on some stats for the upcoming parts. See Section B.1 for expected knowledge.

We set the function $T(I)$ as the time it takes depending on the random variable I. Obviously, we set $E(T(I))$ as the expected runtime across all possible values of I.

We then say that the function[3] $T(n)$ is the maximum of $E(T(I))$ across all I's.

$$T(n) = \max_{|I|=n} E(T(I))$$

7.1 Selection

Given a set of n numbers S, we'd like to return the k-th smallest element of S.

[1] See the QuickSort details in Section A.1

[2] We can get the first method using the second.

[3] I and n are different types, and polymorphism works in pseudocode.

For example:

- $k = 1$ is the min
- $k = 2$ is the 2nd min
- $k = n$ is the max
- $k = \lfloor \frac{n}{2} \rfloor$ is the median

Let's implement QuickSelect:

```
def QuickSelect(S, k):
    n = |S|
    if n < constant
        Sort(S)
        return kth element
    i = rand(1...n)
    partion S into:
    L = {s : s < S[i]}
    M = {s : s == S[i]}
    R = {s : s > S[i]}
    if k < |L| return QuickSelect(L, k)
    if k <= |L| + m return s[i]
    return QuickSelect(B, k - (|L| + |M|))
```

This is worst-case $O(n^2)$ when pivot is always the min or the max, but it often isn't the worst-case.

We can do more detailed analysis to find the expected time of finding it on a set S of size n.

In other words, we want $E(T(n))$, where $T(n)$ is a random variable runtime of QuickSelect on a set of size n.

We have recursive calls on sets of size ℓ or $n - \ell$. For an upper bound, assume that k lies in the larger (worse) half of the recursion. In other words, we assume that $k \leq \frac{n}{4}$ or $k \geq \frac{3n}{4}$. Thus the recursion is $\leq T(\frac{3n}{4})$.

Assuming that $T(i) \leq T(j)$ for $i \leq j$, we get:

$$E(T(n)) \leq \frac{1}{2}E\left(T\left(\frac{3n}{4}\right)\right) + \frac{1}{2}E\left(T\left(n-1\right)\right) + O(n)$$

$$f\left(n\right) \leq \frac{1}{2}f\left(\frac{3n}{4}\right) + \frac{1}{2}f\left(n-1\right) + O(n)$$

We can prove by induction that $f(n) = O(n)$.

7.2 Random V.S. Non-Randomized Algorithms

1960 Hoare QuickSelect has $3n + o(n)$ expected comparisons
1973 BFPRT created a non-randomized selection in $O(n)$ time, with $5.43n + o(n)$ comparisons. This is the

same with respect to $O(n)$, but different constant than randomized algorithms.
1975 Floyd Rivest created a randomized algorithm that takes $1.5n + o(n)$ expected comparisons.
1989 Munro & Cunto proved that any ralgorithm takes at least $1.5n$ expected comparisons.
1985 [???] proved a lower bound of $2n$ comparisons for non-randomized algorithms. Randomization probably helps.

Currently, our best non-randomized bounds are:

Bound	Year	# comparisons
Upper Bound	1999	2.95n
Lower Bound	2001	$(2+\varepsilon)n$, $\varepsilon = 2^{-80}$

7.3 Lower Bound on Median

Theorem:[4] Finding the median of n elements takes $\geq 1.5n$ comparisons in the worst case.

Proof:[5]

Let $L = \{\text{elements } < m\}$, and $M = \{\text{elements } > m\}$. So that each set has $\frac{n-1}{2}$ elements.

We claim that the number of L vs H comparisons must be $\geq \frac{n-1}{2}$ in the worst case.

We set it up so the adversary answers the comparisons that our algorithm queries. Our adversary consistently answers by "setting" elements to L and H at all times.[6] We can now create an adversary strategy:

```
def compare(x, y):
    if x and y have been seen before:
        return result of comparison
    if one of (x, y) have been seen:
        put the unseen one in the other set
    if neither are set:
        put x in L, y in H
```

An adversary must stop when $\max(|L|, |H|) = \frac{n-1}{2}$, so they can force at most $\frac{n-1}{2}$ comparisons.

Since there are always $\geq n-1$ L vs H comparisons, and the adversary can force an additional $\geq \frac{n-1}{2}$ L vs H comparisons, the overall algorithm must make $\geq 1.5n$ comparisons in the worst case.

7.3.1 $O(n)$ Non-Randomized Selection Algorithm

The idea here is that we divide sets of n elements into groups of 5, then find the median of each group. We then execute a recursive call to find a median of medians P. This guarantees P between $\frac{3n}{10}$ and $\frac{7n}{10}$ in

[4] Blum et al. '75
[5] This proof type is an adversary proof.
[6] We're effectively trying to find the worst case scenario, played out by an adversary. Stick to the plot, foo!

rank. We get the recurrence:

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

We can prove that $T(n) = O(n)$.[7]

For more information, look up the "median of medians" algorithm online.

[7]This can probably be done using induction, but it isn't noted.

Chapter 8

Primality Testing

8.1 Randomized Algorithm Types

There are two kinds of randomized algorithms:

8.1.1 Las Vegas Type Algorithms

Las Vegas algorithms always return the correct output, and have good expected runtime. An example of this type of algorithm is quicksort.

We can convert Las Vegas to Monte Carlo algorithms by stopping after some time and outputting a junk answer.

8.1.2 Monte Carlo Type Algorithms

Monte Carlo algorithms are quick with a high probability of success, and have a good guaranteed runtime.

If we have a fast correctness test we can convert Monte Carlo algorithms to Las Vegas algorithms, repeating the algorithm if output isn't correct.

8.2 Primality Testing Using a Monte Carlo Algorithm

Given an odd number n, is n composite?[1] Phrased this way, we have a decision problem in NP, which is verifying `YES` answers.

It is important to know that while the input is n, the input size is $\log(n)$ – the number of bits used expressing n. Thus trial division ($O(\sqrt{n})$ time) is not poly-time.[2]

[1] In other words, is n not prime?

[2] In 2002, Agrawal, Kayal, and Saxena published a poly-time non-randomized algorithm to test primality known as the AKS Primality Test.

We use the following theorem to help us with our solutions:

8.2.1 Fermat's Little Theorem

If p is prime, then $\forall 0 < a < p$: $a^{p-1} \equiv 1 \mod p$.

We can prove this by showing:

$$\begin{aligned} a^p(p-1)! &\equiv (p-1)! \mod p \\ a^p &\equiv 1 \mod p \end{aligned}$$

The contrapositive[3] states that whenever $a^{n-1} \not\equiv 1 \mod n$ doesn't hold for $0 < a < n$, then a is a *Fermat Witness* to n being composite.

8.2.2 Prime-Testing

The idea is to test n being composite using randomly-generated a in $[1, \ldots, n-1]$ for being a Fermat witness.

If it is, then YES n is composite. If it isn't, then MAYBE n is prime.

The bad news is that there are composite numbers without any Fermat witnesses[4].

Where $n - 1 = 2^t u$ (for an odd u), we need a Strong Witness that n is prime. We define $a \in [1, n-1]$ as a strong witness of n being composite if for some $0 \leq i < t$, $k = 2^i u$:

$$\begin{aligned} a^k &\not\equiv 1, -1 \mod n \\ a^{2k} &\equiv 1 \mod n \end{aligned}$$

In CLRS, they prove that if n is prime, there are no strong witnesses; they also prove that if n is composite, there are $\geq \frac{n-1}{2}$ strong witnesses.

8.2.3 Implementation

```
witness (a , n):
u = n - 1 % 2
t = log((n - 1) / u) // base 2
x[0] = a ^ u mod n
for i = 1 ... t:
    x[i] = x[i-1]^2 mod n
    if (x[i] == 1 and x[i-1] != 1 and x[i-1] != n - 1):
        return true // a is a strong witness to n being composite
return x[t] != 1 // a is a Fermat witness to n being composite
```

[3] Contrapositive notes can be found at B.3.1.

[4] The numbers without Fermat witnesses are called Carmichael numbers. The first 3 are 561, 1105, 1729.

The runtime of this algorithm is polynomial in $\log n$.

8.2.4 Miller-Rabin Algorithm

The idea of this algorithm is to test s times that random numbers aren't witnesses to n being composite.

```
isComposite(n):
    for i = 1 ... s:
        x = rand(1...n-1)
        if (witness(x, n)):
            return YES // n is composite
    return MAYBE // n is prime
```

If n is prime, the algorithm is always correct. IF n is composite however, we can tabulate the probability it is unsure:

$$\begin{aligned} Pr\{\text{Alg outputs MAYBE}\} &= Pr\left\{\cap_{j=1}^{s}\{\text{at trial } j,\ x \text{ is not a strong witness }\}\right\} \\ &= \frac{1}{2^s} \end{aligned}$$

This is a Monte-Carlo algorithm with a one-sided error[5].

8.3 Complexity Classes

We can define a number of decision classes:

P are the decision problems solvable in polynomial time. These are also known as the class of languages L accepted in polynomial time.

NP are the class of languages L accepted in non-deterministic polynomial time. These are also known as the decision problems that can be verified in polynomial time[6].

There are a few[7] OPEN problems about this[8]:

$$\begin{aligned} \text{NP} &\stackrel{?}{=} \text{CO-NP} \\ \text{P} &\stackrel{?}{=} \text{NP} \\ \text{P} &\stackrel{?}{=} \text{NP} \cup \text{CO-NP} \\ \text{P} &\stackrel{?}{=} \text{RP} \\ \text{RP} &\stackrel{?}{=} \text{NP} \end{aligned}$$

[5] This means that for a decision problem, only one of YES or NO can be wrong.
[6] i.e. the YES answers can be verified in poly-time.
[7] okay, maybe "many"
[8] The classes RP and CO-NP are defined later.

8.3.1 Randomized Polynomial Time, One Sided Monte-Carlo

The RP class of problems is the class of languages that have a randomized algorithm A running in worst-case polynomial time such that for any input x:

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

In other words, the algorithm always returns no for input x that don't match, and *sometimes* returns yes for x that match [9].

We know that P $\subseteq$ RP, since the probabilities that P problems will accept and decline are 0 and 1 respectively.

Supposing language L is in RP, i.e. there is a randomized algorithm A that fits the definitions of RP. A depends on x and random choices. If we think of the random choices as a string y of random bits, we write $A(x, y)$ as applying A on x with random bits y. Since A runs in polynomial with respect to $|x|$ ($A \in p(|x|)$), we know that the string $y \in p(|x|)$. Using A as the verification algorithm and y as the certificate, we can show that L is in NP.

8.3.2 Zero Error Probabilistic Polynomial Time

ZPP is the class of languages accepted by Las Vegas algorithms with an expected polynomial runtime.

Note that P $\subseteq$ ZPP $\subseteq$ RP.

An in-class quiz consisted in proving that ZPP = RP $\cap$ co-RP is true.

See here for more details on the co-RP complexity class.

8.3.3 Open Questions

It is OPEN if these containments are proper, or if they can be made more precise:

$$\text{P} \subseteq \text{RP} \subseteq \text{NP}$$

[9] Though the specification is that the probability must be ≥ 0.5, repeated random tests can increase the probability for lower values. A better constraint is that the probability that $A(x)$ accepts must be non-zero.

Chapter 9

Finger-Printing - Pattern Matching and Polynomial Identities

9.1 String Equality

It's pretty expensive to compare strings, especially if they're long, stored in separate locations, etc. We compare a smaller fingerprint x where x is an n-bit binary number ($x < 2^n$). For a randomly chosen $p \in \{1 \dots M\}$[1], we can set:

$$H_p(x) = x \mod p$$

While $x = y$ implies $H_p(x) = H_p(y)$, this contrapositive doesn't hold true if p divides $|x - y|$.

With repeated (in)equality testing of $H_p(x)$ to $H_p(y)$, we can build confidence about $x \stackrel{?}{=} y$. Our algorithm will know for sure when $x \neq y$, but it can't be sure they are equal. Thus this is a Monte-Carlo Algorithm.

To better analyze our algorithm, we want to define $Pr\{\text{failure}\}$. If we define $\pi(n)$ as the number of primes less than n, then $\pi(n) \approx \frac{n}{\ln n}$[2]. Another result from number theory dictates that the number of prime divisors of $A < 2^n$ is $\pi(n)$.

$$\begin{aligned} Pr\{\text{failure}\} &= \frac{\text{number of primes } p < M \text{ and } p \text{ divides } |x - y| < 2^n}{\pi(M)} \\ &= \frac{\pi(n)}{\pi(M)} \end{aligned}$$

If we pick $M = n^2$, then we have $Pr\{\text{failure}\}$:

$$\begin{aligned} Pr\{\text{failure}\} &= \frac{n}{\ln n} \frac{\ln n^2}{n^2} \\ &= \frac{2}{n} \end{aligned}$$

[1] M is chosen later

[2] This is a prime number theorem.

9.2 Pattern Matching

We can use a similar idea as string matching for pattern matching:

Given a test string T and a pattern string P (where $|T| = n$, $|P| = m$), does P appear as a substring of T?

There's a straightforward $O(nm)$ solution[3].

9.2.1 Rabin-Karp Algorithm

Rabin-Karp supplies a simple and efficient randomized algorithm.

Suppose T and P are binary strings. We want to compare the fingerprint of P to fingerprints of successive substrings of T.

Using a "rolling hash", these fingerprints in T can be computed very efficiently[4].

```
def hasMatch(text T, text P):
    p = randomPrime(1 ... m)
    compute Hp(P)
    compute Hp(T[1 ... m])
    for i in range(1 ... n-m+1):
        if Hp(P) == Hp(T[i ... i+m-1])
            return PROBABLE_MATCH
    output NO_MATCH
```

We have the runtime of $O(n + m)$ arithmetic operations. We are more concerned about the failure rate - the probability that we output PROBABLE_MATCH without there being a real match. Iff p divides[5] $|P - T[i \dots i + m - 1]|$ for some i, then p divides $\Pi_i |P - T[i \dots i + m - 1] \leq 2^{nm}$.

Thus, the following of failure is: (and to recap...)

$$\begin{aligned} & p \text{ divides } |P - T[i \dots i + m - 1] \text{ for some } i \\ \implies & p \text{ divides } \Pi_i |P - T[i \dots i + m - 1]| \leq 2^{nm} \\ \implies & Pr\{\text{failure}\} \leq \frac{\pi(nm)}{\pi(M)} \end{aligned}$$

Where M is some number. We can choose $M = n^2 m$, then we have:

$$\begin{aligned} Pr\{\text{failure}\} &\leq \frac{nm}{\ln(nm)} \frac{\ln(n^2 m)}{n^2 m} \\ &< \frac{2}{n} \end{aligned}$$

i.e. if $n = 4000 < 2^{12}$ and $m = 250 < 2^8$, then $M = n^2 m < 2^{32}$. We can use a 32-bit fingerprint prime, and the $Pr\{\text{error}\} < 10^{-3}$.

[3] We have a few other methods:
Using finite automata, we have $O(m|\Sigma| + n)$ non-randomized algorithms, where Σ is the size of the alphabet.
Using Knuth-Morris-Pratt or Boyer-Moore algorithms, we have $O(n + m)$ non-randomized algorithms.

[4] $H_p(T[i + 1 \dots i + m]) = (2H_p(T[i \dots i + m - 1]) - T[i]2^m + T[i + m]) \mod p$

[5] Where P and $T[\dots]$ are viewed as binary numbers

In practice this is slower than Boyer-Moore, but it's better when you need to test multiple patterns in one string.

9.2.2 Verifying Polynomial Identities

Given a Vandermonde matrix M:

$$M = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \dots & x_n^{n-1} \end{bmatrix}$$

There is the Vandermonde identity: $\det(M) = \Pi_{j<i}(x_i - x_j)$. We can verify this by substituting random values for variables[6][7][8].

9.2.3 Verifying Polynomial Identities

Theorem: let $f(x_1 \dots x_n)$ be a multivariate polynomial of total degree[9] d. If f is not identically 0 and if we choose values $a_1 \dots a_n$ for $x_1 \dots x_n$ independently and uniformly from a finite set S, then we claim $Pr\{f(a_1 \dots a_n) = 0\} \leq \frac{d}{|S|}$.

For example, if $S = \{0, \pm 1, \pm 2 \dots \pm d\}$, then $Pr\{f(a_1, \dots a_n) = 0\} \leq \frac{1}{2}$.

Proof: We can do this by induction on n:

The basic case is when $n = 1$ single variable of degree d implies that there are $\leq d$ roots and in general, we can substitute and evaluate..

$$f(x_1 \dots x_n) = \sum_{t=0} x_1^t g(x_2 \dots x_n)$$

9.2.4 Verifying Matrix Multiplication

Given three matrices A, B, and C that are all $n \times n$ in size. We want to verify that $AB = C$.

While the naive matrix multiplication is $O(n^3)$, one of the faster multiplication algorithms is $O(n^{2.376})$ by Coppersmith and Winograd in 1990[10]. These are complicated to implement, and the chance of implementing buggy programs is very high.

The idea is that by choosing a vector $x = [x_1 \dots x_n]$, we can quickly verify that $ABx = Cx$ is correct[11]

```
choose each x[i] = rand(0, 1)
if A(Bx) == C(x):
```

[6] There exists a efficient algorithm for computing determinants.
[7] Can compute modulo prime
[8] There's a theorem for equality that's useful, but that type of test comes up in symbolic math programs
[9] e.g. $f(x_1, x_2, x_3) = x_1 x_2^3 + x_3^2 + x_1 x_2$ has total degree 4.
[10] There are slight improvements since, notably Williams finding $O(n^{2.3727})$.
[11] By the previous part, this is a degree 1 ($d = 1$) multivariate polynomial.

```
        return MAYBE
    return NO
```

We can set the probability of error $Pr\{\text{error}\} \leq \frac{d}{|S|} = \frac{1}{2}$ (since $S = \{0, 1\} \rightarrow |S| = 2$).

This runs in $O(n^2)$ time, and we can repeat it to reduce error.

Chapter 10

Linear Programming in Low Dimension

Linear programming is a math (and computational) method for achieving the best outcome given a model expressed as a series of linear relationships.

In other words, given a $d \times 1$-vector $\vec{x}$, an $n \times d$ matrix A, a $1 \times d$ vector c, and a $n \times 1$ vector b, maximize cx while satisfying the constraint $Ax \leq b$.

Expressed differently, we have d inequalities we need to satisfy, and n variables $x_i : i \in \{0 \dots n\}$ while we're trying to maximize $\sum c_i x_i$.

More in this section can be found on [MR section 9.10.1], or see Chapter 4 of the book Computational Geometry by de Berg, van Kreveld, Overmars and Schwarzkopf, Springer 2000.

10.1 Naive Algorithm

In 2D, each constraint $a_1 x_1 + a_2 x_2 \leq b$ is a half-space. As long as the feasible region is non-empty and is bounded by an inequality[1], an optimal solution is at a meeting point of at two lines - a vertex[2].

This gives us a stupid algorithm: try all $\binom{n}{d}$ sets of vertexes, eliminate infeasible vertexes, then find the maximum objective value. This gives an $O(\binom{n}{d}) = O(n^d)$ algorithm.

10.2 Applications of Linear Programming

We can use this to plan menus. With n nutrients, we need b_i of nutrient i. With d foods, each food j has a cost c_j and an amount $a_{i,j}$ of nutrient i.

Defining x_j as the volume of food j purchased, we want to minimize cx while maintaining that $Ax \geq b$.

[1] If there is no inequality bounding this, then the optimal solution occurs at $\pm\infty$.
[2] It's important to note that the optimal solution may not be unique.

10.3 History of Linear Programming

10.3.1 Simplex Method

Dantzig introduced the simplex method in the 1940s, spurring the development of computers. Geometrically, it walks from one vertex of a feasible region to an adjacent one according to a simplex pivot rule that dictates which inequality to remove and which to add. For almost all simplex pivot rules, we know examples taking exponential time.

The *Hirsch Conjecture* conjects that the diameter of a convex d-dimension polyhedron with n inequalities is $\leq n - d$. Sadly, it was disproved in 2012.

OPEN: This doesn't mean that there is no polynomial (or even linear) bound.

In general though, the simplex method is very good in practice.

10.3.2 Other Algorithms

There have been some polynomial-time algorithms for linear programming:

Katchian discovered the ellipsoid method in 1980.

Karkarkar discovered the interior point method in 1984 (it operates on bit representations of numbers).

OPEN: Is there an algorithm that uses the number of arithmetic operations polynomial in both n and d?

The 1970s and 1980s saw linear programming being used in small ($d = 2, 3$) dimensions.

Uses of this were finding the best line fitting points, and whether a cast can be removed from a mold[3].

Finally, *Megiddo* found an algorithm that runs in $O(n)$ when d is fixed[4].

10.4 Randomized Incremental Linear Programming Algorithm

We're going to examine Seidel's Randomized Incremental Linear Programming Algorithm.

The idea is that we want to add half-planes h_i one by one, updating the optimal solution vertex v every time.

When we add h_i, there are two cases:

1. In the case that $v \in h_i$, we have no work to do.
2. In the case that $v \notin h_i$, we need to find a new optimum. We know that the new optimum will line on ℓ_i, a line the h_i plane. So we solve the 1-dimensional LP problem along line ℓ_i.

   The 1D LP (LP1) algorithm runs as follows: (Where L is a set of rays in 1D)

[3] de Berg et al. used 3D linear programming to achieve this

[4] Actually, this algorithm runs in $O(2^{2^d} n)$, but the "linear" growth of n is the object

```
LP_1(L):
    find and return lowest upper bound on x
```

LP_1 runs in $O(|L|)$.

Then, we can implement $LP_2(H)$, $H = \{h_1 \dots h_n\}$ as follows:

```
LP_2(H):
    shuffle H
    v = point at infinity
    for i = 1 ... n: // add H[i]
        if v is not in H[i]:
            v = LP_1(intersect(H[1 ... i-1]), L[i])
```

Since LP_1$= O(i)$ in this implementation, then it runs in worst-case $O(n^2)$.

We can calculate the expected runtime using **backwards analysis**[5]:

After adding h_i, suppose the new optimum is vertex v' is at the intersection of h', and h''.

Given that we have i lines, halfplane h_i is equally likely to be any one of them.

We did work for h_i when we call LP_1, but only if $h_i = h'$ or $h_i = h''$. Since h_i is equally likely to be any of them, we know:

$$Pr\{h_i \in \{h', h''\}\} = \frac{2}{i}$$

Thus we know that the expected total work when calling LP_1 is:

$$\sum_{i=1}^{n} \frac{2}{i} O(i) = O(n)$$

In higher dimensions, the $\frac{2}{i}$ becomes $\frac{d}{i}$, since it takes d hyperplanes to specify a vertex. Thus, we have the recurrence relation:

$$T_d(n) = T_d(n-1) + \frac{d}{n} O\left(T_{d-1}(n)\right)$$
$$T_d(n) = O(d!n)$$

I think she mentioned in class that we can solve this recurrence by proving with $T_2(n)$ by induction, then proving $T_d(n)$ by induction.

10.5 Randomized Incremental Disc Fitting

We can use a similar approach to find the smallest enclosing disk for a set of points:

Given points $p_1 \dots p_n \in \mathbf{R}^d$, find the smallest radius disc enclosing all points.

This is not linear programming (since it involves quadratics), but Megiddo's approach still works, so there is an $O(n)$ non-randomized algorithm[6].

We can create a *randomized-incremental* approach as follows:

[5] The idea of this kind of analysis is to consider the situation after an element has been added, and note that it is a random element among the set added so far.

[6] That of course depends badly on d.

Given a disc S_{i-1} for a solution to $i-1$ points, add a new point p_i.

If p_i is contained in S_{i-1}, $S_i = S_{i-1}$.

If not, we know that S_i goes through p_i.

Thus, we have a *easier* (or *smaller*) problem: given some points and a special point p_i, find the smallest disc containing all points and with p_i on a boundary. The trick for this question is realizing that S_i goes through both p_i and $p_{\text{previous max}}$. Once we have three fixed points on a disc, we have a unique solution. Using this principle leads to an expected[7] runtime of $O(n)$.

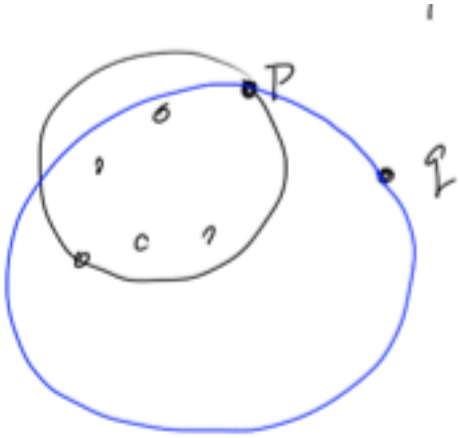

Figure 10.1: Smallest Disc

[7] TODO: Why isn't this guaranteed?

Chapter 11

Randomized Algorithms for Satisfiability (SAT)

Generally, satisfiability is the question that asks that given a boolean formula with n variables and m clauses when expressed in CNF, can we assign TRUTH or FALSE values to satisfy the formula. In the example below with E, assigning x_1 and x_2 satisfies the formula:

$$E = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_3)$$
$$x_1 = \text{TRUE}$$
$$x_3 = \text{FALSE}$$

The 3-SAT algorithm is an NP-variant where all clauses have 3 distinct literals. The 2-SAT algorithm can be solved in polynomial time (in fact, $O(n)$ time).

We can apply this to everything, as it helps with quantified boolean formulae. SAT is a case of one (implicit) $\exists$ quantifier.

11.1 Techniques for SAT

There are heuristics that help us "resolve" different clauses. In fact, we can solve 3-SAT (in a non-obvious way) in $O(1.5^n)$ time using deterministic algorithms instead of the obvious $O(2^n \text{poly}(n, m))$.

Using randomized algorithms, we're going to get better than $O(1.5^n)$ for 3-SAT[1].

We are unlikely to get randomized polynomial time algorithms, since this implies randomized poly-time for all problems in NP. (eek)

11.2 Randomized SAT Solving

The idea is that we're going to be given an input E in CNF, then we're going to "hill climb" to better values. This algorithm is called Papadimitrion's algorithm ('91).

[1] By concentrating on 2-SAT then broadening our scope to 3-SAT.

```
randomly assign T/F assignment A
for i = 1...t:
    if A satisfies E return YES
    pick a random unsatisfied clause C
    randomly pick a literal x in C
    flip x's value
return NO (maybe)
```

We want to choose t and determine the error probability.

Errors occur when E is satisfiable and we return NO.

Suppose A^* is a truth value assignment that satisfies E. For i as the number of variables with same value in A and A^*, we can say that if i reaches n, then $A = A^*$ and the algorithm outputs YES.

When we re-assign the value of the variable, we know that i goes up or down by 1.

11.2.1 Randomized Walk on a Line

To do this analysis, we need to know about random walks on lines.

Start at a randomly chosen i, each step moves right (`i++`) with probability $\frac{1}{2}$ and left (`i--`) with probability $\frac{1}{2}$. When $i = 0$, we always go right. When $i = n$, we terminate.

The question can now be phrased as: What are the expected number of steps to get to n?

Alternatively, we can analyze this as a Markov chain, or a finite automaton with probabilistic state movements.

We're now looking for the expected number of steps to get from i to n - denoted t_i.

$$\begin{aligned} t_n &= 0 \\ t_0 &= 1 + t_1 \\ t_i &= 1 + t_{i-1}\frac{1}{2} + t_{i+1}\frac{1}{2} \end{aligned}$$

This is awkward for induction, but if we rearrange it, it's pretty smooth:

$$\begin{aligned} t_i + t_{i+1} &= 2(t_{i-1} - t_i) \\ d_i &= t_i + t_{i+1} \\ d_0 &= t_0 - t_1 \\ &= 1 \\ d_i &= 2 + d_{i-1} \\ &= 1 + 2i \end{aligned}$$

If we substitute this for t_i, we get:

$$\begin{aligned} t_i &= d_i + t_{i+1} \\ t_n &= 0 \\ t_i &= \sum_{j=1}^{n-1} d_J \\ &= \sum_{j=1}^{n-1} (1 + 2j) \\ &= n - 1 + \sum_{j=1}^{n-1} j \\ &= (n - i) + n(n-1) - i(i-1) \\ &= n^2 - i^2 \end{aligned}$$

The maximum is $t_0 = n^2$, and $t_i \leq n^2$.

11.2.2 Finding Error in our Approximation

For Papadimitrion's solution to 2-SAT, we can model the number of steps as a random walk on a line. In this representation, we say t_i is a state where i variables are "set correctly", and assume the worst case scenario of only one assignment being correct.

In a clause $C = (\alpha \vee \beta)$ being modified was not satisfied, then one of α or β must be true in the optimal solution. If only one needs to be inverted, we pick the correct one $\frac{1}{2}$ the time. If both need to be inverted, we pick the correct one every time. So we can say that the probability that i increases is $\geq \frac{1}{2}$.

By strategically picking the value of t, we can easily determine the expected number of repeats. Using Markov's inequality (which can be found in Section B.2):

Supposing $X \geq 0$ and $E(X) = \mu$, then $\Pr\{X \geq c\mu\} \leq \frac{1}{c}$ for a constant c. In our case, $\mu = n^2$, so we choose $c = 2$.

$$\Pr\{\#\text{ steps} > 2n^2\} < \frac{1}{2}$$

So set $t = 2n^2$, then $\Pr\{\text{error}\} < \frac{1}{2}$.

From this we know that the runtime is not $O(n^2)$, but it actually is $O(n^2 \cdot poly(n, m))$ time.

11.2.3 Papadimitrion's Algorithm in Higher Dimensions

For a given clause $C = (\alpha \vee \beta \vee \gamma)$, if A does not satisfy C, but A^* does with $\alpha = T$.

$$\Pr(\text{Algorithm flips } \alpha) = \frac{1}{3}$$

$$\Pr(i \text{ increases}) \geq \frac{1}{3}$$

So we analyze a random walk on a line:

$$\Pr(\text{i goes to } i+1) = \frac{1}{3}$$
$$\Pr(\text{i goes to } i-1) = \frac{1}{3}$$

Using Markov's inequality as before, we are expected to take $\approx 2^n$ steps to get to n, the final value[2].

11.2.4 Schöning's Algorithm

Schöning ('99) gives two improvements to the algorithm:

- Start with a random assignment A
- An increasing number of trials is not helpful if we've taken many steps without reaching A^*. We're probably stuck near or at 0, so lets pick a "new" random A.

```
schoning(E):
    for i = 1...s:
        randomly pick A
        repeat t = 1...3n:
            if A satisfies E output YES
            else
                pick unsatisfied clause C
                randomly flip a variable in C
    output NO-MAYBE
```

In the inner loop, the probability of error is $\Pr(\text{error}) \lesssim 1 - \left(\frac{3}{4}\right)^n$.

When we set $S = c\left(\frac{3}{4}\right)^n$, the probability of error $\Pr(\text{error}) \lesssim (1 - \left(\frac{3}{4}\right)^n)^{c\left(\frac{4}{3}\right)^n}$.

From calculus, we know that $\left(1 - \frac{1}{a}\right)^a \leq \frac{1}{e}$, so the probability of error is $\Pr(\text{error}) \lesssim \frac{1}{e^c}$.

The bottom line is that we get $\Pr(\text{error}) \leq \frac{1}{2}$, with runtime $O\left(\left(\frac{4}{3}\right)^n n\right)$. While this is exponential, it does beat the best known non-randomized algorithm that we know.

[2] Analysis omitted from course slides.

Chapter 12

Minimum Spanning Trees

The problem MST can be expressed as follows:

> Given an undirected graph $G = (V, E)$ with edge weights $w : E \to \mathbf{R}^+$, find a minimum-weight spanning tree.

In other words, find the tree on the graph that reaches all vertexes such that has the minimal total of edge weight in the tree.

Let's assume that edge weights are distinct[1].

We can generate this problem to a spanning forest of disconnected graph[2]

There are two basic rules:

Inclusion Rule The inclusion rule dictates that for a given vertex v, if v's minimum weight incident edge[3] is $e = vu$, then $vu \in \text{MST}(G)$. Since we know this, we can contract the vertexes v and u into each other creating a vertex v', and continue the process with a smaller graph. For every vertex r which has an edge to both v and u, we just add the smaller of the two edges to v'.

Exclusion Rule The exclusion rule dictates that for a given cycle C with maximum weight edge e, then $e \notin \text{MST}(G)$. We may delete e and continue.

While basically all MST algorithms work under these rules, we can't get the MST without contraction.

In this analysis, n is the number of vertexes, and m is the number of edges.

12.1 Kruskal's Algorithm ('56)

Kruskal's Algorithm uses the inclusion rule.

[1] If they aren't, we only need to break ties consistently.
[2] I have no clue what this means.
[3] An incident edge to v is an edge between v and another vertex.

```
mst(G)
    repeat:
        e = (u, v) = minimumWeightEdge(G)
        T += uv
        contract(G, e)
    end
```

By sorting edges by weight, then using union-find to find the new vertexes connected to edges after contraction, this algorithm takes $O(m \log n)$ time.

12.2 Prim's Algorithm ('57)

```
mst(G):
    S = randomVertexFrom(G.V)
    repeat:
        e = minimumWeightEdgeFrom(S)
        T += e
        contract(G, e)
        S.put(e.from)
        S.put(e.to)
```

Implementing with a heap takes $O(m \log n)$ time. Implementing with a Fibonacci Heap takes $O(n \log n + m)$ time, which is linear when $m > n \log n$.

12.3 Borůvka's Algorithm ('26)

The idea is that we want to apply the inclusion rule to all vertexes at once. We'll actually just apply it until every vertex is a contracted one, and the resultant number of vertexes is $\leq \frac{n}{2}$.

12.3.1 Borůvka Step

The is that we want to ensure every vertex is part of at least one merge.

```
baruvka(G):
    unmark all vertexes
    for each v in V:
        if v is unmarked:
            find minimum weight edge e=vu
            add e to T, contract v to u
            mark u
    return T
```

For each vertex v, the Borůvka Step checks v's minimum weight edge, and contracts $v \to u$, which takes $O(\deg(v))$ time. Thus, the entire step takes:

$$O\left(\sum_v \deg(v)\right) = O(m)$$

The step reduces the graph to $\leq \frac{n}{2}$ vertexes.

12.3.2 Borůvka's Algorithm

The idea is to repeat the Borůvka Step until only one vertex is left.

Since there are going to be $O(\log n)$ reductions, the total time is $O(m \log n)$. This isn't as fast in practice as Prim, but it's much simpler to implement.

12.4 History of MST Algorithms

- In '75, Yao, Cheriton, and Tarjan found a $O(m \log\log n)$ algorithm.
- In '85, Fredman and Tarjan found a $O(m \log^* n)$ algorithm.
- In '87, Chazelle found an $O(m\alpha(n))$ algorithm.

OPEN: Is there a linear time ($O(n+m)$) algorithm?

12.5 Karger's Algorithm ('93)

Karger gave a Las-Vegas MST algorithm with linear expected run time. The idea of his algorithm is to use random sampling, the exclusion rule, and recursion.

We want the algorithm MST(E) to return the MST of each connected component of $G = (V, E)$.

```
MST(E):
    take a random subset R <= E of size |R| = r // chosen later: r=2n
    T = MST(R)
    for each edge uv in E, do:
        if uv is not in T and uv is heavier than all edges in the uv path in T:
            classify uv as heavy
        else:
            classify uv as light and replace uv with e in T
    E = E - heavy edges
    return MST(E)
```

This is correct by the exclusion rule. If we added a new edge $e = uv$ to the sample R:

- If e is heavy, T does not change.
- If e is light, we would've updated T to use uv.

Additionally T is the MST of the entire graph iff all edges not in T are heavy.

We can classify edges and verify if T is a MST of the entire graph in $O(m)$ time[4].

12.5.1 Sampling Lemma

We propose that the number of light edges $E(\#\text{ light edges}) \leq \frac{mn}{r}$. Since there are m edges, this is enough to show that $\Pr(\text{e is light}) \leq \frac{n}{r}$.

We can prove this by working backwards: Consider $R' = R \cup \{e\}$, where e is a random element of R'. By the notes above, e is light with respect to R if and only if e is in the $\text{MST}(R')$ (which has $\leq n-1$ edges). So $\Pr(e \text{ is light}) \leq \frac{n-1}{|R'|} < \frac{n}{r}$.

12.5.2 Analysis of Expected Runtime

We have the following recursion:

$$T(m,n) = \text{recursive call on } R + \text{time to classify} + \text{recursive call to find the MST of } E\text{-heavy}$$
$$T(m,n) = T(r,n) + O(m+n) + T\left(\frac{mn}{r}, n\right)$$

With $r = 2n$, this becomes:

$$T(m,n) = T(2n,n) + O(m+n) + T\left(\frac{mn}{2n}, n\right)$$
$$T(m,n) = T(2n,n) + T\left(\frac{m}{2}, n\right) + O(m+n)$$

The final idea, (attributed to Karger, Klein, and Tarjan in '94) is on each recursive call to do 3 Borůvka steps first. This reduces # vertexes $\leq \frac{n}{8}$ with $O(n+m)$ work. For some constant d, we have:

$$T(m,n) \leq T\left(\frac{n}{4}, \frac{n}{8}\right) + T\left(\frac{m}{2}, \frac{n}{8}\right) + d(m+n)$$

Proving that $T(m,n) \leq c(m+n)$ for some constant c is sufficient to prove $T(m,n) \in O(n+m)$. We implicitly assume that the base case has been proven, and prove the inductive step:

$$\begin{aligned} T(m,n) &\leq c\left(\frac{n}{4} + \frac{n}{8}\right) + c\left(\frac{m}{2} + \frac{n}{8}\right) + d\,(m+n) \\ &= \left(\frac{c}{2} + d\right) n + \left(\frac{c}{2} + d\right) m \\ &\leq c(n+m) \text{ as long as } \frac{c}{2} + d \leq c \end{aligned}$$

So the expected runtime is $O(m+n)$.

[4] In class, she alluded that this is complicated, and didn't go any further. At the bottom of her notes though, there is a link to A CMU grad course's (link) explanation of the matter.

Part II

Approximating Hard Things

Chapter 13

Approximation Algorithms

Recall P $\stackrel{?}{=}$ NP.

We have a set P $\subseteq$ NP, and another set NP-COMPLETE $\subseteq$ NP, where NP-COMPLETE are the hardest problems in NP. There are a few problems in NP that are not in P and not known to be in NP-COMPLETE (factoring, graph isomorphism, etc).

It is OPEN if there exist poly-time correct algorithms to solve NP-COMPLETE problems.

Ladner proved that:

> If P $\neq$ NP, then there are infinitely many problems in the space between P and NP-COMPLETE.

It seems like we must either give up correctness, or speed.

And thus, approximation algorithms are born. For optimization problems, these algorithms guarantee that their result is *close to* the optimal solution.

13.1 Concerning Approximation Algorithms

An approximation algorithm finds in polynomial-time a solution that is close to the optimal, either in terms of ratio or in constant difference.

Edge-Coloring in a Graph

Given a graph G, color the edges such that if two edges are incident[1], they have different colors.

A variant of this problem is NP-COMPLETE:

> Given G and $k \in \mathbf{N}$, can you edge color G with k colors?

[1] Two edges are incident if they share a vertex.

Vizing's Theorem states that for the maximum degree across all vertexes in a graph Δ, $\Delta \leq$ minimum number of colors $\leq$ $\Delta + 1$.

Furthermore, there exists a polynomial-time algorithm to color any graph with $\Delta + 1$ colors.

Since the algorithm exists, we can approximate within $+1$ of the OPT solution.

This type of approximation (constant additive) is rare, since we usually get a good ratio of APPROX to OPT.

Vertex Cover

Given a graph $G = (V, E)$, find a minimum-size vertex cover - a set $U \subseteq V$ such that every edge has at least one endpoint in U.

We can use this kind of algorithm to monitor all links in a network.

The decision version of Vertex-Cover is NP-COMPLETE. Where Independent set is the question to find the maximum set of vertexes where no two are joined by an edge, there is a reduction this way:

$$3\text{SAT} \leq_p \text{independent set} \leq_p \text{vertex cover}$$

The argument for reduction between vertex cover and independent set is that $U \subseteq V$ is a minimum vertex cover if and only if $V - U$ is a maximum independent set.

The existence of an approximation algorithm for vertex cover that's good within an additive constant (as for edge coloring) implies P = NP.

13.2 Greedy Algorithm for Max Vertex Cover

```
maxCover(V):
    C = []
    while true:
        if no edges remain: break;
        C.append(vertex of max degree)
        remove covered edges
```

We will show that $|C| \leq O(\log n)|\text{OPT}|$.

Exercise for reader (not in notes): Show that the greedy algorithm can give $\frac{|C|}{|\text{OPT}|} \in \Omega(\log n)$.

13.3 Set Cover Problem

The max vertex problem is a subset of the Set Cover problem.

Given a collection of sets $S_1, S_2, \ldots, S_k$ where $S_i \subseteq [1, n]$. Find a minimum sized set $C \subseteq [1, n]$ such that for all $i \in [1, n]$, $i \in S_j$ for some $j \in C$.

In the real world, this works as follows:

Where sets S_i are a type of pizza, and set elements e are individual people. An element $e \in S_i$ means that the person eats that type of pizza. We want to find the minimum number of pizza types to feed all people.

13.3.1 Vertex vs Set Cover

We can show that vertex cover is a special case of Set Cover: Elements e of our sets are edges in the graph, and sets correspond to vertexes in our graph.

Since every element in our vertex cover is in exactly two of our sets, a A Set Cover that every element is in exactly two of our sets allows us to transform our Set Cover problem into a vertex cover.

13.3.2 Greedy Approximation Algorithm for Set Cover

The idea is to iteratively choose the set that has the most yet uncovered elements.

```
setCover(S[] s):
    C = []
    while there are uncovered elements:
        S[i] = a set that covers the max number of uncovered elements
        C.append(i)
```

We claim that $|C| = \text{APPROX} \leq O(\log n)\text{OPT}$, where OPT is the minimum number of sets to cover all elements.

This proof is taken from Vazirani's book[2], which is a simpler proof than the one presented in CLRS.

We distribute the cost (1) of choosing a set S_i over the newly covered elements. Let $c(e)$ represent the cost of adding element e.

We define S as the maximum size set, since it is the first one chosen. For a defined element $e \in S$, we define $c(e) = \frac{1}{|S|}$. We know that $|S| >$ the average number of elements per set in the OPT solution. We also know that the average number of elements n per set in the OPT solution is $\geq \frac{n}{\text{OPT}}$. This implies that for the first set we have:

$$c(e) \leq \frac{\text{OPT}}{n}$$

More generally, let the ordering $e_1, e_2, \ldots, e_i, \ldots e_n$ be an ordering of elements as they are covered (we expect many ties).

We define that the number elements newly covered by S' as k. For a given $e_i \in S'$, we know that the number of elements uncovered prior to S' being chosen must be $\geq n - i + 1$.

Since the set picked is the one with the maximal k, we know that it must be $\geq$ AVG on the range $j \geq i$ covered by OPT. We know that $\text{AVG} \geq \frac{n-i+1}{\text{OPT}}$, since any lower would mean that there are more than OPT sets chosen in the OPT solution. Thus, $k \geq \frac{n-i+1}{\text{OPT}}$, which implies that $c(e_i) \leq \frac{\text{OPT}}{n-i+1}$.

[2]This seems to be the only reference to Vazirani in her notes, but she mentions in class that he visited UWaterloo a while ago and showed her a misprint in the subtitle of an old version of his book.

$$\begin{aligned}\text{number of sets chosen by greedy} &= \sum_{i=1}^{n} c(e_i) \\ &\in O(\log n)\text{OPT}\end{aligned}$$

Thus, the number of sets chosen by the greedy is within a factor of $O(\log n)\text{OPT}$.

Chapter 14

Linear Programs and Randomization

14.1 Vertex Cover

Please recall the definition of the vertex cover problem on page 53, and the approximation within $O(\log n)\text{OPT}$ presented on page 53.

14.1.1 Constant-Factor Approximation for Vertex Cover

There's a "stupid" approximation algorithm for vertex cover that's better than the greedy algorithm.

```
vertex_cover(E, V):
    C = {}
    while E != {}:
        (u, v) = E.remove_random()
        e = {u, v}
        C.append(e)
        E.remove_connected_to([u, v])
    return C
```

Since we pick the set of edges is a matching[1]. Where M is the set of edges we pick, we know that $|C| = 2|M|$. We also know that $|M| \leq \text{OPT}$, since every matching edge needs its own vertex in the OPT vertex cover.

Thus, our "stupid" greedy algorithm has $|C| = 2|M| \leq 2\text{OPT}$, and thus gives us an approximation factor of 2.

Best Approximation Factor Known for Vertex Cover

The best approximation factor known for the Vertex Cover problem is 2.

[1] A matching M is a set of edges such that no two share a common vertex.

Weighted Vertex Cover

Given weights on vertices $w : V \to \mathbf{R}^+$, find the vertex cover C of minimum weight $\sum_{v \in C} w(v)$.

In fact, we can express this as an integer linear program.

Create a variable $x(v)$ for each $v \in V$.

$$x(v) = \begin{cases} 1 & : v \in C \\ 0 & : v \notin C \end{cases}$$

Now we're trying to find the solution to the following linear program:

$$\min \sum_{v \in V} w(v)x(v)$$

Given the constraints that $\forall e \in E : e = (u, v) : x(u) + x(v) \geq 1$, and bounding the values of $x()$ to be $x(v) \in \{0, 1\}$.

The solutions to this integer linear program are exactly minimum-weight vertex covers. While Integer Linear Programs (ILP) are NP-COMPLETE[2], relaxing the constraints to be non-integer Linear Program (LP) allows us to use the simplex method (Refex to Subsection 10.3.1). To use simplex, we can relax $x(v) \in \{0, 1\}$ to $x(v) \in [0, 1]$.

Suppose that $\bar{x}$ is an optional solution to the linear program. Let $C_{\text{LP}} = \{v \in V : \bar{x}(v) \geq \frac{1}{2}\}$. Our linear program guarantees that $\bar{x}(u) + \bar{x}(v) \geq 1$, so at least one of u, v has $\bar{x}(-) \geq \frac{1}{2}$, and that vertex is definitely in C_{LP}. Then C_{LP} is a vertex cover.

Starting with the premise that OPT = OPT to ILP $\geq$ OPT to LP, since the LP allows more solutions, so the OPT decreases. Given that $\bar{x}(-)$ is the OPT solution to the LP, we have:

$$\begin{aligned} \text{OPT} &= \sum w(v)\bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq 0.5} w(v)\bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq 0.5} w(v)\frac{1}{2} \\ &= \frac{1}{2} \sum_{v \in V : \bar{x}(v) \geq 0.5} w(v) \\ &= \frac{1}{2} w(C_{\text{LP}}) \end{aligned}$$

So OPT $\geq \frac{1}{2} w(C_{\text{LP}})$ or $w(C_{\text{LP}}) \leq 2\text{OPT}$.

14.2 Set Cover Problem

Given a collection of sets $S_1, S_2, \ldots S_k$, where $S_i \subseteq \{1, \ldots, n\}$, find a minimum size set of sets such that every element is covered.

[2]There is no proof given, and Chapter 10 doesn't deal with this type of problem.

In other words, find $C \subseteq \{1, \ldots, k\}$ such that $\forall i \in \{1, \ldots, n\}$, $i \in S_j$ for some $j \in C$.

For a Set Cover, the greedy algorithm gives $\leq O(\log n)\text{OPT}$ (refer to Subsection 13.3.2).

There's something that's marginally better. Define f as the maximum number of sets any single element e is contained in. For vertex cover, the elements are edges, and the sets are edges incident to a vertex v. f for vertex cover is 2, so it is a special case of Set Cover.

There is a polynomial time approximation algorithm using linear programming and duality for Set Cover that gives a solution of $\leq f\text{OPT}$, which is an approximation factor f.

Depending on the difference between $\log n$ and f, one may be factor.

Later, we will prove that Set Cover has no constant factor approximation in polynomial time unless P = NP.

Approximation For Minimization and Maximization Problems

A ρ-approximization algorithm for a minimization problem guarantees that $A(I) \leq \rho\text{OPT}(I)$, so $\rho \geq 1$.

A ρ-approximization algorithm for a maximization problem guarantees that $A(I) \geq \rho\text{OPT}(I)$, so $\rho \leq 1$.

Some texts use $\frac{1}{\rho}$ for maximization problems. CLRS doesn't do max, and Vazirani uses the convention here.

Max-Cut Problem

The Max Cut problem is a sub-type of the Set Cover problem.

> Given a graph $G = (V, E)$, partition V into S, $V - S$ to maximize the number of edges with one end in S and another end in S.

We define $c(S)$ as the size of cut[3] for S, $V - S$.

Without proof, the (decision version of the) Max-Cut problem is NP-complete, but the minimum cut can be found in polynomial time.

An approximation for Max Cut is as follows:

```
max_cut(G):
    S = V.any_subset()
    while vertexes can be moved to increse c(S):
        move a vertex to increase c(S)
```

Given that we can increase $c(S)$ at most m times[4], we must have a polynomial time algorithm.

Define the number of edges inside S as $e(s)$, and the number of edges from v to S as $d_S(v)$, and the number of edges from v to $V - S$ as $d_{V-S}(v)$. At the end of the algorithm, $\forall v \in S$, $d_S(v) \leq d_{V-S}(v)$, otherwise we

[3] That is, the number of edges cut.

[4] As usual, m is the number of edges.

would move it.

$$\sum_{v \in S} d_{V-S}(v) \geq \sum_{v \in S} d_S(v)$$
$$\implies 2e(S) \leq c(S)$$

Similarly $\forall v \in V - S$: Thus:

$$d_{V-S}(v) \leq d_S(v)$$
$$\sum_{v \in V-S} d_{V-S}(v) \leq \sum_{v \in V-S} d_S(v)$$
$$\implies 2e(V-S) \leq c(S)$$

Since $m = e(S) + e(V-S) + c(S)$, we can say:

$$\begin{aligned} m &= e(S) + e(V-S) + c(S) \\ 2m &= 2e(S) + 2e(V-S) + 2c(S) \\ &\leq c(S) + c(S) + 2c(S) \\ &= 4c(S) \end{aligned}$$

Thus $m \leq 2c(S)$, which means we have an approximation factor of $\frac{1}{2}$.

Random Algorithm for Max Cut Problem

```
max_cut(G):
    pick S at random
```

We can do this analysis:

$$E[c(S)] = E\left[\sum_{e \in E} \sigma(e)\right]$$
$$\sigma(e) = \begin{cases} 1: & \text{if } e \text{ in cut} \\ 0: & \text{otherwise} \end{cases}$$
$$\begin{aligned} E[c(S)] &= \sum_{e \in E} E[\sigma(e)] \\ &= \sum_{e \in E} \frac{1}{2} \\ &= \frac{m}{2} \end{aligned}$$

By doing this repeatedly, we can get a $O(n)$-time algorithm with a $\frac{1}{2}$ approximation factor.

State of the Art in Max Cut

The best known approximation factor for Max Cut is 0.878 (unattributed).

Chapter 15

Max SAT

Given a set of m clauses in CNF in n boolean variables $X = \{x_1, \ldots, x_n\}$, find a boolean assignment to variables to make a maximum number of clauses true.

The decision version of this problem is NP-COMPLETE, since it's just the usual SAT problem. For clauses of size ≤ 2, it is still NP-COMPLETE. For problems with clauses are size 2, deciding if all can be satisfied is in P[1].

15.1 Algorithm for Max-SAT

The algorithm is to pick the truth assignment at random.

We know that for any clause c containing t different variables, we want to determine the probability that c is satisfied:

$$\begin{aligned}
\Pr(c \text{ is satisfied}) &= 1 - \Pr c \text{ is not satisfied} \\
&= 1 - \frac{1}{2^t} \\
&\geq (1 - \frac{1}{2}) \\
&= \frac{1}{2}
\end{aligned}$$

Thus the expected number of clauses satisfied is $\geq \frac{1}{2}m$.

We know that OPT $\leq m$, so the expected APPROX factor is $\frac{1}{2}$.

15.2 Facts about the Max-SAT problem

By the probabilistic method[2], there always exists a truth value assignment that satisfies at least half the clauses.

[1] To prove that it is NP-COMPLETE, we only need to reduce 3-SAT to Max 2-SAT, just turn $(x \vee y \vee z)$ into about 10 2-SAT clauses such that $(x \vee y \vee z)$ is satisfied if about 7 of the 2-SAT clauses are satisfied.

[2] Not explained in class, but apparently there are many powerful consequences.

The proof is essentially that since the expected value of a random variable is α, then there is at least one value for the variable that has value $\geq \alpha$.

15.3 Improved Algorithm for Max-SAT

By formulating this as an ILP problem, we can solve this using a LP relaxation (Similar to the Weighted Vertex Cover solution on Page 57) and "randomized rounding" techniques.

Make variables x_i, $i \in [1, n]$ for each boolean variable a_i, and y_i, $i \in [1, m]$ for each clause.

We want to maximize $\sum_i y_i$.

We have one constraint per clause:

$$c_1 = (\bar{a_1} \vee a_2) \rightarrow y_1 \leq (1 - x_1) + x_2$$

And we have the LP relaxation constraints:

$$\begin{aligned} 0 \leq & x_i \leq 1 : i \in [1, n] \\ 0 \leq & y_i \leq 1 : i \in [1, m] \end{aligned}$$

Using the P LP approximation, we solve for x's.

Then, we set our truth values $A = \{a_0, \ldots, a_n\}$ as follows:

$$a_i = \begin{cases} 1: & \text{with probability } x_i \\ 0: & \text{otherwise} \end{cases}$$

Analysis of Randomized Max 2-SAT

We have a given $c_2 = (a_1 \vee a_2)$. In a LP solution, $y \leq x_1 + x_2$. Then we have:

$$\begin{aligned} \Pr(c_2 \text{ is satisfied}) &= \Pr(a_1 = 1 \vee a_2 = 1) \\ &= x_1 + x_2 - x_1 x_2 \\ &\geq x_1 + x_2 - \left(\frac{x_1 + x_2}{2}\right)^2 \\ &\geq y_2 - \frac{y_2^2}{4} \\ &\geq y_2 - \frac{y_2}{4} \\ &= \frac{3}{4} y_2 \\ \Pr(c_2 \text{ is satisfied}) &\geq \frac{3}{4} y_2 \\ E[\text{NUM OF CLAUSES SATISFIED}] &\geq \frac{3}{4} \sum_i y_i \\ &= \frac{3}{4} \text{OPT}_{\text{LP}} \geq \frac{3}{4} \text{OPT}_{\text{ILP}} \geq \frac{3}{4} \text{OPT}_{\text{MAX SAT}} \end{aligned}$$

Thus, we expect the number of clauses satisfied to be $\geq \frac{3}{4}\text{OPT}$.

Analysis of Randomized Max SAT

We won't do the analysis, but it shows that this method is better for small clauses, and the algorithm described in Section 15.1 is better for large clauses.

15.4 Max-Sat Retrospective

The best-known max-SAT algorithm is by Goemans & Williamson '94, and is a 0.878 approximation factor[3].

We also know that there is no approximation factor ≥ 0.942, unless P = NP.

So far, we've seen:

- metricTSP - Saw a 2-approximation, but know of a 1.5-approximation
- Vertex Cover - 2-approximation
- Set Cover - $O(\log n)$
- Max cut - $\frac{1}{2}$
- Max SAT - $\frac{3}{2}$

15.5 Polynomial-Time Approximation Scheme

These types of algorithms are ones where we can get arbitrarily close to 1.

Though most of the details on these are in the next lesson (Chapter 16), we will go into a few proofs now.

Without proof, it is presented that:

> Polynomial-time α-approximation for the Independent set problem implies a polynomial-time α-approximation for Max 3-SAT.

15.5.1 Reduction Preserving Constant Factor Approximation

In '92, this breakthrough result was published:

> If Max 3-SAT has a PTAS, then P = NP.

Joined with the previous theorem, then there is no PTAS for the independent set unless P = NP.

Proving this:

[3]Yes, this is the same factor as Max-Cut, as denoted on page 14.2

Given a Max 3-SAT formula F, we want to transform it into an independent set problem as a polynomial-time reduction Max $3-$ SAT $\leq_p$ Independent Set. While doing this, we want to preserve the approximation factor.

A clause $c = (x_1 \vee \bar{x_2} \vee x_3)$ becomes a triangle, and an edge between $(x_i, \bar{x_i})$ for all occurences of the inversion in the SAT problem. This can be seen in Figure 15.1.

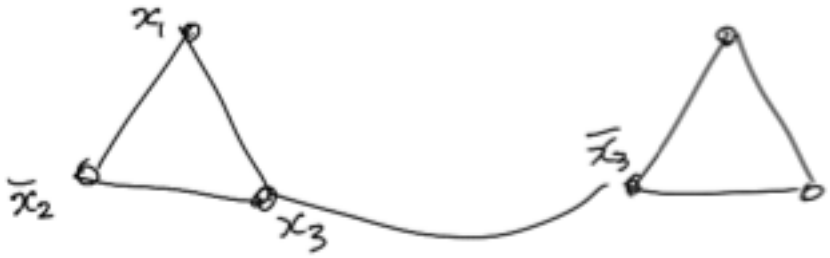

Figure 15.1: Independent Set from a Triangle

There truth value assignment that satisfies all k clauses iff there is an independent set of k vertices.

In particular, $\text{OPT}_{\text{Max 3-SAT}}(F) = \text{OPT}_{\text{Ind. Set}}(G)$, and a polynomial time approximation for algorithm for Ind. Set that gives $A_{\text{Ind. Set}}(G) \geq \alpha \text{OPT}_{\text{Ind. Set}}(G)$ implies a polynomial-time approximation algorithm for the max-SAT that gives $A_{\text{Max 3-SAT}}(F) \geq \alpha \text{OPT}_{\text{Max 3-SAT}}(F)$.

15.5.2 Reduction Preserving Constant Factor Approximation, With a Different Constant

We give an example where a reduction provides constant-factor approximation, but not the same constant.

We allude that the example here is a "gap-preserving reduction", but we won't learn more about it in this course.

We'd like to show how a polynomial-time $(1+\varepsilon)$-approximation for Vertex Cover implies a polynomial-time $(1-5\varepsilon)$-approximation for Max 3-SAT.

From this, we make the observations:

- With the breakthrough result from earlier, we now know that there's no PTAS for vertex cover (unless P = NP).
- We have a 2-approximation for Vertex Cover, but this promises us a $(1-5)$-approximation, which is vacuous (and therefor it doesn't apply).

To prove our theorem, we need a reduction Max 3-SAT $\leq$ vertex cover with a good approximation factor preservation.

Using the above, we can reduce Max 3-SAT $\leq$ Ind. Set, plus the idea that U is an Ind. Set iff $V - U$ is a vertex cover.

While this idea doesn't give a good approximation factor in general, but we only need it for graphs that come from Max 3-SAT. From earlier, picking random truth values for Max 3-SAT, so $\text{OPT}_{\text{SAT}} \geq \frac{1}{2}m$.

$A_{\text{VC}} \leq (1+\varepsilon)\text{OPT}_{\text{VC}}$ by assumption.

By reduction, we get the approximate polynomial-time for Max 3-SAT "A_{SAT}".

$$\begin{aligned} A_{\text{SAT}} &= 3m - A_{\text{VC}} \\ \text{OPT}_{\text{SAT}} &= 3m - \text{OPT}_{\text{VC}} \end{aligned}$$

We want to prove that $A_{\text{SAT}} \geq (1-5\varepsilon)\text{OPT}_{\text{SAT}}$.

From the above, we can reduce:

$$\begin{aligned} A_{\text{SAT}} &= 3m - A_{\text{VC}} \\ &\geq 3m - (1+\varepsilon)\text{OPT}_{\text{VC}} \\ &= 3m - (1+\varepsilon)(3m - \text{OPT}_{\text{SAT}}) \\ &= \text{OPT}_{\text{SAT}} - \varepsilon m + \varepsilon\text{OPT}_{\text{SAT}} \\ &\geq \text{OPT}_{\text{SAT}} - \varepsilon 6\text{OPT}_{\text{SAT}} + \varepsilon\text{OPT}_{\text{SAT}} \\ &= \text{OPT}_{\text{SAT}}(1-5\varepsilon) \end{aligned}$$

Chapter 16

Geometric Packing PTAS

We're going to look at packing problems.

16.1 Set Packing

We can express set packing through the following statement:

> Given elements $\{1, \ldots, n\}$ and sets $\{S_1, \ldots, S_k\}$, with $S_i \subseteq [1, n]$. Find the maximum set of S_i such that no two sets intersect.

There's a graph version of the same problem:

> Given a graph $G = (V, E)$, find a maximum size subset $U \subseteq V$ such that no edge $e = (u, v)$ has both endpoints in U.

The largest Indep. Set (See the definition of this problem inside Section 14.1) is a special case of set packing, where all edges incident to vertex v_i is the set S_i.

The independent set problem is NP-COMPLETE, and we can turn general set packing into the independent set problem, therefore set packing is NP-COMPLETE.

In fact, there is no $n^{1-\varepsilon}$-approximation ratio for independent set, unless P = NP.

16.2 Geometric Set Packing With Squares

> Given n unit squares in the plane. Find the maximum number of squares such that no two intersect.

16.2.1 Simple Constant Factor Approximation For Packing Unit Squares

```
pick_non_intersecting(S[] squares):
    ans = []
    while (|S| != 0):
        sq = s.random()
        s.remove_intersecting(sq)
        ans.append(sq)
    return ans
```

This is a $\frac{1}{4}$-approximation algorithm because at one square chosen by A intersects at most four squares in OPT.

Thus, each square of A intersects ≤ 4 squares of OPT. Therefore OPT $\leq 4A$.

16.2.2 Grid Approximation Algorithm

If we represent squares by their center points, and put down a unit grid with the "even" squares, we can get the set of points from each of the shaded squares.

Let R_0 be the set of shaded squares. In fact, we can define:

$$R_0 = \{(x, y) : \lfloor x \rfloor \text{ is even}, \lfloor y \rfloor \text{ is even}\}$$
$$R_1 = \{(x, y) : \lfloor x \rfloor \text{ is odd}, \lfloor y \rfloor \text{ is even}\}$$
$$R_2 = \{(x, y) : \lfloor x \rfloor \text{ is even}, \lfloor y \rfloor \text{ is odd}\}$$
$$R_3 = \{(x, y) : \lfloor x \rfloor \text{ is odd}, \lfloor y \rfloor \text{ is odd}\}$$

The best way to show this is through a figure. Refer to Figure 16.1 for a pictorial example.

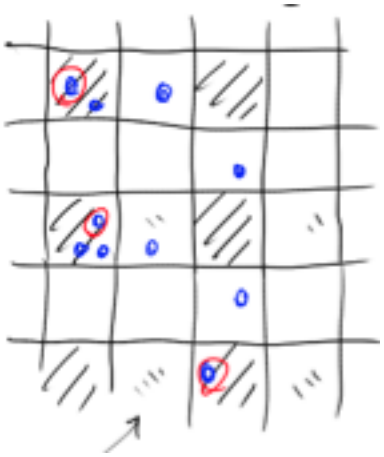

Figure 16.1: Grid Approximation Algorithm

The algorithm is to take one point from each shaded square $R_0 \cap P$, if there is one.

```
grid_approx_alg(P Point[]):
    for i = 0...3:
        Q[i] = opt solution R_i.intersect(P)
    return max(Q)
```

We know that $|Q| \geq \frac{1}{4} \sum |Q_i|$, since we have max $\geq$ avg.

$$\begin{aligned}\text{OPT} &= \sum_{i=0}^{3} |\text{OPT} \cap R_i| \\ &\leq \sum_{i=0}^{3} |Q_i| \\ &\leq 4|Q|\end{aligned}$$

Thus, $|Q| \geq \frac{1}{4}\text{OPT}$.

16.2.3 Arbitrary Grid Approximation Algorithm

It turns out that we can get arbitrarily close to an approximation factor of 1, using a "shifting grid" approach[1]

We pick a $k \geq 2$, and let $R_{i,j}$ be defined as:

$$R_{i,j} = \{(x, y) : \lfloor x \rfloor \,\%k \neq i, \lfloor y \rfloor \,\%k \neq j\}$$

Over $i \in [0, k-1]$ and $j \in [0, k-1]$. Refer to Figure **??** for $k = 3$.

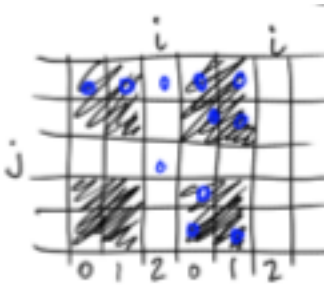

Figure 16.2: Arbitrary Grid Approximation Algorithm

Note that our points lie in a $n \cdot n$ square without loss of generality.

For a given k, the number of black squares is $\leq \frac{n}{k} \cdot \frac{n}{k} = \frac{n^2}{k^2}$.

Also, the number of points we can choose in a single black square is $\leq (k-1)^2$.

Since each set of black squares is independent, we can solve the problem optimally for $R_{i,j} \cap P$ in polynomial time by trying all possible subsets of $\leq (k-1)^2$ points. There are $O(n^{(k-1)^2})$ possible subsets per square.

The algorithm is as follows:

```
arbitrary_grid_approximation_algorithm(P, k):
    for i=[0, k-1]:
        for j=[0, k-1]:
            Q[i][j] = R[i][j].intersect(P)
    return max(Q)
```

[1] Discovered by Hochbaum & Maas, '85.

$$\max \geq \text{avg}$$
$$\implies |Q| \geq \frac{1}{k^2} \sum_{i,j=0}^{k-1} Q_{i,j}$$

Since each point is in $(k-1)^2$ of the $R_{i,j}$'s, then we have:

$$\begin{aligned}(k-1)^2 \text{OPT} &= \sum_{i,j} |\text{OPT} \cap R_{i,j}| \\ &\leq \sum_{i,j} |Q_{i,j}| \\ &\leq k^2 |Q|\end{aligned}$$

So we have $|Q| \geq \frac{(k-1)^2}{k^2}\text{OPT}$ with a runtime $O(k^2 n^{(k-1)^2})$.

16.3 PTAS-like Definitions

You'd think that using these for so long, we would've defined this stuff already:

Approximation Scheme an algorithm A, input I and parameter ε.

- For min problem: $A(I, \varepsilon) \leq (1+\varepsilon)\text{OPT}(I)$
- For max problem: $A(I, \varepsilon) \geq (1-\varepsilon)\text{OPT}(I)$

Polynomial Time Approximation Scheme (PTAS) an algorithm A that for each fixed ε, A runs in P time in $n = |I|$. e.g. $O(n^{\frac{1}{\varepsilon}})$.

Fully Polynomial Time Approximation Scheme (FPTAS) an algorithm A that runs in P time in $n = |I|$ and $\frac{1}{\varepsilon}$. e.g. $O(\frac{1}{\varepsilon}^2 n^3)$.

For the algorithm described in Subsection 16.2.3, the approximation factor $\frac{(k-1)^2}{k^2} = 1 - \frac{2k-1}{k^2}$, so $\varepsilon = \frac{2k-1}{k^2}$.

The runtime is $O(k^2 n^{(k-1)^2})$. We have $\varepsilon < \frac{2k}{k} = \frac{2}{k}$, so $k < \frac{2}{\varepsilon}$. Also we have $(k-1)^2 < k^2 < \frac{4}{\varepsilon^2}$. So we have the overall runtime in $O(\frac{1}{\varepsilon^2} n^{\frac{4}{\varepsilon^2}})$. Thus this is a PTAS but not a FPTAS.

Chapter 17

Bin Packing PTAS

Bin packing is a cool problem.

17.1 Bin Packing Description and Variants

Given n numbers[1] $S = \{s_1, \ldots, s_n\}$, and $s_i \in [0, 1]$. Pack S into the minimal number of unit bins possible.

This problem is NP-HARD, since it generalizes the PARTITION problem:

Given S as above, we can we split S into two bins U and $S - U$ such that:

$$\sum_{s \in U} s = \sum_{s \in S - U} s$$

There are two variants to this problem:

Online numbers in S arrive one at a time and must be inserted to bins immediately.

Offline all numbers in S arrive simultaneously.

17.2 First-Fit Bin Packing

The first-fit algorithm is really simple:

Insert the element into the first bin that fits it.

Due to the algorithm's simplicity, this can be done online.

[1] I don't know why she used S, given that it usually denotes a set. Oh well.

Let m be the number of bins used by first fit. Since no two bins are $\leq \frac{1}{2}$ full, then at least $m-1$ bins are $> \frac{1}{2}$.

$$\begin{aligned}\frac{1}{2}(m-1) < \sum s_i & \\ &\leq \text{OPT} \\ m &< 2\text{OPT} + 1\end{aligned}$$

So $m \leq 2\text{OPT}$.

We state without proof that the first fit algorithm uses $\leq 1.7\text{OPT} + 1$ bins, and this is tight[2].

17.3 First Fit Decreasing Bin Packing

An off-line algorithm allows us to sort our input in a decreasing order.

First-fit decreasing sorts S descending, then applies the first fit algorithm.

We have the asymptotic ration of $\frac{11}{9} = 1.22$, which is tight.

17.4 PTAS for offline Bin Packing

For any constant $\varepsilon > 0$, there is a polynomial-time algorithm A'that uses $\leq (1+\varepsilon)\text{OPT} + 1$ bins.

The "+1 bins" is crucial:

> Prove that 2 bins there is no P time approximation algorithm with an approximation factor of < 1.5 (unless P = NP).

The algorithm A' is built from two cases:

1. $\forall i : s_i \geq \delta$, and there are only k possible values of s_i for constant k and δ.

   We can solve exactly through brute force enumeration.

   There are $\leq \frac{1}{\delta}$ items in each bin, so there are $k^{\frac{1}{\delta}}$ ways to fill each bin. Given that there are $\leq n$ bins, where each bin has $k^{\frac{1}{\delta}}$ choices. Overall, we have $O(n^{k^{\frac{1}{\delta}}})$, which is polynomial for fixed k and δ, but huge.

   Apparently we can do better with ILP algorithms.

2. $\forall i : s_i \geq \delta$ for some constant δ.

   We convert this to the first case by rounding.

   Sort values of s_i ascending, and chop every k values of s_i (k is chosen later).

[2]Being tight means that we've proved that it's ≤ 1.7, and have infinitely many examples where the algorithm gives this ratio, effectively proving the approximation uses $= 1.7\text{OPT} + 1$ bins.

Create a set with modified input rounded up to the nearest $s_{\frac{ni}{k}}$:

$$S^+ = \{s_{\frac{n}{k}}, \dots s_{\frac{n}{k}}, s_{\frac{2n}{k}}, \dots s_{\frac{2n}{k}}, \dots s_n, \dots s_n\}$$

In other words, there are $\frac{n}{k}$ of $s_{\frac{n}{k}}$.

We then apply part 1 to S^+.

If we define S^- similarly to S^+ but instead rounding down instead of up, then we have:

$$\begin{aligned} \text{OPT}(S^-) &\leq \text{OPT}(S) \\ \text{OPT}(S^+) &\leq \text{OPT}(S^-) + \frac{n}{k} \\ &\leq \text{OPT}(S) + \frac{n}{k} \end{aligned}$$

Now $n\delta \leq \sum s_i \leq \text{OPT}(S)$, so $n \leq \frac{\text{OPT}(S)}{\delta}$.

$$\begin{aligned} \text{OPT}(S^+) &\leq \text{OPT}(S) + \frac{\text{OPT}(S)}{k\delta} \\ &= \text{OPT}(S) + \left(1 + \frac{1}{k\delta}\right) \end{aligned}$$

If we set $k = \frac{1}{\delta^2}$, then $\text{OPT}(S^+) \leq opt(S)(1+\delta)$.

The final algorithm uses case 2 to pack all s_i's where $s_i \geq \delta$, then use first fit to pack the remaining s_i's in the empty spaces of all bins.

17.4.1 Analysis of the PTAS

Our goal is to prove that $A \leq (1+\varepsilon)\text{OPT} + 1$.

If the second part does not use new bins, then it's ok to use $\delta = \varepsilon$.

Otherwise, we have use m bins in our algorithm.

Only the last bin can have size $\leq 1 - \delta$, otherwise first fit wouldn't have filled the last bin. Then:

$$\begin{aligned} \sum s_i &\geq (m-1)(1-\delta) \\ m &\leq \frac{\sum s_i}{1-\delta} + 1 \leq \frac{\text{OPT}(S)}{1-\delta} + 1 \end{aligned}$$

If we choose $\frac{1}{1-\delta} = 1 + \varepsilon$, then we have $\delta = \frac{\varepsilon}{1+\varepsilon}$ and $m \leq (1+\varepsilon)\text{OPT}(S) + 1$ as desired.

The runtime is as follows:

$$O(n^{k^{\frac{1}{\delta}}}) = O\left(n^{\left(\frac{(1+\varepsilon)^2}{\varepsilon}\right)^{\frac{1+\varepsilon}{\varepsilon}}}\right)$$

17.5 Improvements for Bin Packing Algorithms

Karmarkar and Karp in '82 created an asymptotic $(1+\varepsilon)$-approximation that runs in $O\left(\left(\frac{1}{\varepsilon}\right)^8 n \log n\right)$ time. This is an example of a FPTAS.

It is OPEN if we can get $n_{\text{bins}} \leq \text{OPT}(S) + O(1)$ in P time.

Chapter 18

Knapsack FPTAS

18.1 Problem Background

Give objects $1 \dots n$, each with size $s_i \in \mathbf{N}$ and profit $p_i \in \mathbf{N}$, given knapsack capacity B find $K \subseteq \{1, \dots, n\}$ such that $\sum_{i \in K} s_i \leq B$ while maximizing $\sum_{i \in K} p_i$.

The decision version of this problem is NP-COMPLETE[1].

18.2 Pseudo-Polynomial Time Algorithm for Knapsack with DP

We want to find a solution to the knapsack problem using a DP solution.

A subproblem $S(i, p)$ is defined as the minimum size of the subset of items in $[1, i]$ with profit of exactly p.

Our algorithm is as follows:

```
knapsack_dp(S[] s, P[] p, B):
    p_max = max(P)
    sums = [n][p_max];
    for p = [1, p_max]:
        if (p == p[1]):
            sums[1][p] = p[1]
        else:
            sums[1][p] = infinity
    for i = [2, n]:
        for p = [1, p_max]:
            sums[i][p] = min(sums[i-1][p], sums[i-1][p-p[i]] + s[i])
    return max p such that sums[n][p] <= B
```

The runtime of this is $O(n\texttt{p\_max})$, which is $O(n^2 \max_i(p_i))$.

[1] This can be proved by reducing subset sum to knapsack or partition.

This algorithm is pseudo-polynomial time, since the runtime depends on the largest p_i, not on the size of p_i's, which is $\log(p_i)$ bits.

Some NP-COMPLETE problems don't have pseudo-polynomial time algorithms (unless P = NP). For example, TSP is still NP-COMPLETE with $0-1$ weights, since it still can be reduced to the Hamiltonian cycle problem.

18.3 FPTAS for the Knapsack Problem

The idea is that if p_i is few bits, then the pseudo-polynomial time algorithm runs in polynomial time $O(n^2 \max_i(p_i))$. So let's round p_i's to have few bits.

Run the following algorithm for a value of t:

```
knapsack_fptas(epsilon, S[] s, P[] p_initial, B):
    t = ...
    P[] p = new P[p_initial.length]
    for i = [0, p_initial.length]:
        p[i] = floor(p_initial[i]/t)
    return knapsack_dp(s, p, B)
```

Suppose the result returns $K(t) \subseteq [1, n]$. $K(t)$ is feasible, since $\sum_{i \in K(t)} s_i \leq B$.

We need to analyze $P(K(t)) = \sum_{i \in K(t)} p_i$ compared to $P(K^*) = \text{OPT}$.

We know that $p_i - t < tp'_i \leq p_i$. Then we have:

$$\begin{aligned}
\sum_{i \in K(t)} p_i &\geq \sum_{i \in K(t)} tp'_i \\
&\geq \sum_{i \in K^*} tp'_i \\
&\geq \sum_{i \in K^*} (p_i - t) \\
&= \sum_{i \in K^*} p_i - t|K^*| \\
&\geq \text{OPT} - t|K^*| \\
&\geq \text{OPT} - tn \\
&= \text{OPT}\left(1 - \frac{tn}{\text{OPT}}\right) \\
&\geq \text{OPT}\left(1 - \frac{tn}{\max_i(p_i)}\right)
\end{aligned}$$

We want $\geq \text{OPT}(1-\varepsilon)$, so we choose t such that $\varepsilon = \frac{tn}{\max_i(p_i)}$, and $t = \frac{\varepsilon \max_i p_i}{n}$.

The runtime is in $O(n^2 \max_i(p'_i))$. We know that:

$$\begin{aligned} p'_i &= \left\lfloor \frac{p_i}{t} \right\rfloor \\ &= \left\lfloor \frac{np_i}{\varepsilon \max_i(p_i)} \right\rfloor \\ &\leq \frac{n}{\varepsilon} \end{aligned}$$

Thus, the runtime is $O(n^3 \frac{1}{\varepsilon})$, which means it's a FPTAS.

18.3.1 Comments on the State-of-the-Art

The best known algorithm has the runtime $O(n \log \frac{1}{\varepsilon} + \left(\frac{1}{\varepsilon}\right)^4)$. The general idea is to separate it into large profit items and small profit items, then to use the large ones first.

18.3.2 FPTAS and Pseudo-Polynomial Time Algorithms

Though we've shown that for the knapsack problem, having a pseudo-polynomial time algorithm $\rightarrow$ FPTAS, it's not known (i.e. it's OPEN) that this is true in general.

Garey & Johnson in '78 proved that if a problem has a FPTAS[2], then there must exist a pseudo-polynomial time algorithm for the problem.

The idea is that with a minimization problem and a obj function that's integer-valued (this is the technical assumption, I'm not sure what it means), and a bound of $\text{OPT} < q(|I_{\text{unary}}|)$, where I_{unary} is the input with numbers expressed in unary and q is some polynomial.

Suppose our algorithm A is a FPTAS with runtime $p(|I|, \frac{1}{\varepsilon})$ and $A(I) \leq (1+\varepsilon)\text{OPT}(I)$.

Picking that ε small enough to get the OPT, we can argue that this is pseudo-polynomial.

$$\begin{aligned} \varepsilon &= \frac{1}{q(|I_{\text{unary}}|)} \\ A(I) &\leq (1+\varepsilon)\text{OPT}(I) \\ &< \text{OPT}(I) + 1 \end{aligned}$$

Since our obj function is integer valued, we can make the argument that A must give the OPT solution.

Thus, the runtime is $p(|I|, \frac{1}{\varepsilon}) = p(|I|, q(|I_{\text{unary}}|))$, which is pseudo-polynomial time.

[2] Apparently they made some technical assumptions as well.

Chapter 19

Hardness of Approximation

In general, some NP-COMPLETE problems can be approximated in different manners.

Listed from hardest to easiest, we have:

- $O(n^{\varepsilon})$ - factor
- $O(n \log n)$ - factor (Set Cover)
- Constant factor (Vertex Cover, Euclidean TSP)
- PTAS (Packing Unit Squares, Bin Packing)
- FPTAS (knapsack)

If the positive results give an approximation algorithm, then negative results give that it's hardest.

We can show that a PTAS for vertex cover existing implies that P = NP.

For the Max-3SAT problem, we know through reductions that a PTAS for either vertex cover or Independent Set implies that there must be a PTAS for Max-3SAT, since there are reductions preserving good approximations.

19.1 A New Definition of NP

Earlier, we defined NP as a set of decision problems verifiable in P given a certificate of polynomial size.

If we think about this as a game between the prover P (who finds the certificate) and the verifier V (verifies the problem), then we can get some edge by talking about how much V asks P and how much V guesses randomly.

An interactive proof system is essentially what we're building. V asks P about parts of the proof, and guesses[1] some others.

[1] P does not infer, but guesses randomly.

19.1.1 Graph Isomorphism

> Given 2 graphs G_1 and G_2, can you relabel G_1 to get G_2?

Graph isomorphism is in NP, but it is OPEN if it is in P or in NP-COMPLETE.

For the V, we can pick one of G_1 or G_2 at random, and ask P about which one we relabeled.

- If $G_1 \neq G_2$, then the prover can answer correctly.
- If $G_1 = G_2$, then the prover can't do better than 50% right.

Thus V runs many trials to verify $G_1 \neq G_2$ with high probability.

This is a restricted instance, since there are no rounds and only one prover[2].

19.1.2 Probabilistically Checkable Proofs

Given a result to a decision problem, we have a game between the Prover and Verifier.

- Prover writes the "proof".
- The Verifier is a randomized algorithm that "checks" the proof and answers YES or NO.

If the statement is true, then there is (always?) a proof that makes V (always?) answer YES.

If the statement is false, then V must answer NO with $\Pr \geq \frac{3}{4}$, no matter the proof given.

If we limit V to $O(f(n))$ random bits, and $O(g(n))$, then we define PCP as follows:

> PCP$[f, g]$ is the class of decision problems with Probabilistically Checkable Proof where V uses $O(f(n))$ random bits, and $O(g(n))$ bits of the proof.

We have PCP$[0, \text{poly}(n)] = \text{NP}$, and PCP$[0, 0] = \text{P}$.

19.1.3 PCP Theorem

The "PCP Theorem" is by (Aurora, Lund, Motwani, Sudan, and Szegedy in '92). It states that:

$$\text{NP} = \text{PCP}[\log n, 1].$$

Essentially, V uses random bits to choose where to look at the proof.

Proving PCP$[\log n, 1] \subseteq \text{NP} = \text{PCP}[0, \text{poly}(n)]$ is not hard, since the verifier only needs to eliminate randomness and tries all possible random strings of $O(\log n)$ bits. It then looks at n^k bits of the proof.

The other direction is hard.

[2] Apparently there are instances where more than one prover is necessary, but I can't think of any.

19.1.4 Implications of the PCP Theorem to Hardness of Approximation

From the PCP theorem, we know that a PTAS for Max 3-SAT implies that P = NP.

Using NP = PCP$[\log n, 1]$, we can take any problem in NP and the PCP$[\log n, 1]$ verifier's algorithm for it.

The algorithm depends on the input bits $x_1, \ldots, x_n$, the proof bits $y_1, \ldots, y_t$ (for $t \in O(\text{poly}(n))$), and the random bits $r_1, \ldots, r_k$ (for $k \in O(\log n)$).

We can reduce any[3] algorithm to a Boolean 3-SAT formula.

Using variables for $y_1, \ldots, y_t$, we can choose the formula $F(x, y, r)$ to capture the verifier's algorithm.

Let $F(x, y) = \wedge_r F(x, y, r)$.

Since r is polynomial size, we can say the following:

- if x is YES input, then there exists a y such that all $F(x, y, x)$ are satisfied and thus $F(x, y)$ is satisfied.
- if x is NO input, then for any y, at most $\frac{1}{4}$ of the $F(x, y, x)$ are satisfied, and at most a fraction of the clauses of $F(x, y)$ can be satisfied.

By having a gap when x is a NO input between OPT and "all", we can detect this gap with a good approximation algorithm.

[3] Maybe any NP-COMPLETE?

Chapter 20

Online Algorithms

Given a sequence of requests, our algorithm must handle each request as it comes. This is the usual scenario for data structures, but we will study situations where it makes sense to compare with complete solutions.

- Bin packing: (first vs best fit)
- Splay Trees - dynamic optimality conjecture
- Paging (LRU and LFU)
- Ski rental - rental is \$30, but purchasing skiis is \$300.

20.1 Robots Finding Doors

Suppose we have a robot an unknown distance d away from a door in an unknown direction.

In an OPT route, we can provide a solution in length d.

20.1.1 Algorithm 0

```
find_door_0():
    i = 0
    while (true):
        i++
        go i steps right
        go 2i steps left
        go i steps right
        if seen_door():
            goto door
```

This takes $4(\sum_i^{d-1}) + d$ steps if the door is on the right, and $4(\sum_i^{d-1}) + 3d$ steps if the door is on the left, so this runs in $\theta(d^2)$ time.

20.1.2 Algorithm 1

```
find_door_1():
    i=0
    while (true):
        i++
        if i is odd:
            go 2^i steps right
            go 2^i steps left
        else: // (i is even)
            go 2^i steps left
            go 2^i steps right
```

If $2^i \leq d \leq 2^{i+1}$, then the distance travelled is $\leq 2\sum_j^{i+1} j^2 + d$. Thus the distance travelled $\leq 2(2^{i+2}-1)+d \leq 2(4d+1)+d \leq 9d$.

Since we know the distance travelled is $9d$, then we have what we call a competitive ratio of 9.

20.1.3 Algorithm 2

This is a randomized version of the algorithm.

Flip a coin to choose th initial direction - $f \in \{0, 1\}$.

Then we do the algorithm described in Section 20.1.2. The odd/even test becomes $i \mod 2 = f$, and we now get to see the expected distance travelled:

$$
\begin{aligned}
2(\sum_j 2^j) + \frac{1}{2}2(2^{i+1}) + d & \\
&= 2(2^{i+1}-1) + 2(2^{i+1}) + d \\
&\leq 2(2d-1) + 2d + d \\
&\leq 7d
\end{aligned}
$$

We have a competitive ratio of 7 in this case.

20.1.4 Algorithm 3

The best randomized algorithm has a competitive ratio of 4.592. Let's take a look at it.

For a value of r chosen below, do the following algorithm:

```
find_door_3():
    f = random_bit()
    x = random_float(0, 1)
    i = 0
    while (true):
```

```
if i %2 = f:
    walk r^{i+x} right
    walk r^{i-x} left
else:
    walk r^{i+x} left
    walk r^{i-x} right
++i
```

It can be shown that the expected distance travelled $\leq \frac{r+1}{\ln r} + 1$, which is minimized when $r = 3.59$, giving the distance $4.59d$.

This is the best known approximation factor.

20.1.5 Further Expansion

This becomes harder in 2D, as a robot is trying to find an unknown shape in a plane.

20.2 Auction Strategies

There is an item of value B, and the auction bids occur one at a time. These bids are more like offers, since the algorithm must accept or reject bids immediately as they arrive. All bids are positive nonzero integers.

We want $A(\sigma) \geq c\text{OPT}(\sigma) + b$, where $c \leq 1$.

If the number of bids is unknown, then the algorithm must accept the first bid, or else $\frac{A}{\text{OPT}} = 0$.

Supposing the number of bids is known, the algorithm accepts the first bid above some threshold T, otherwise it accepts the last bid.

Supposing the maximum bid is $M \leq B$, then $\text{OPT} = M$.

If $T = 2$ and $M \geq 2$, $\frac{2}{M} \geq \frac{2}{B}$. If $T = 1$ and $M = 1$, then $\frac{1}{1} = 1$. It's best to have the highest T possible.

20.2.1 Deterministic T Threshold

If we set $T = \left\lfloor \sqrt{B} \right\rfloor$, we claim that the competitive ratio is $\frac{1}{\sqrt{B}}$.

If $M > \left\lfloor \sqrt{B} \right\rfloor$, then in the worst case A accepts $\left\lfloor \sqrt{B} \right\rfloor + 1$. The competitive ratio is $\frac{\lfloor \sqrt{B} \rfloor + 1}{M} \geq \frac{\lfloor \sqrt{B} \rfloor + 1}{B} \geq \frac{1}{\sqrt{B}}$.

If $M \leq \left\lfloor \sqrt{B} \right\rfloor$, then the algorithm accepts the last bid. In the worst case, that bid is 1, so the competive ratio is $\frac{1}{\sqrt{B}}$.

In either case, the competitive ratio is $\frac{1}{\sqrt{B}}$.

20.2.2 Random T Threshold

Choose a random i threshold from $i \in [i, \log B]$, then set $T = 2^i$.

The worst case is that no bid is occurs $\geq 2^i$, so A gets 0.

We want to prove the expected competitive ratio is $\geq O\left(\frac{1}{\log b}\right)$.

Suppose that M is the max bid, so $\text{OPT} = M$.

Suppose that $2^k \leq M < 2^{k+1}$. The probability that we choose $i = k$ is $\frac{1}{\log B}$.

For $i = k$, A gets $\geq 2^k \geq \frac{M}{2}$.

Thus the expected value for the algorithm is $\geq \frac{M}{2 \log B}$. Then, $\frac{A}{\text{OPT}} \geq \frac{1}{2 \log B}$.

Chapter 21

Paging

An online algorithm is c-competitive if there exists a constant b such that $A(\sigma) \leq c\text{OPT}(\sigma) + b$.

We define the paging problem as follows:

> We are given a "cache" of fast memory that holds k elements, and a "disk" of slow memory that holds $n >> k$ pages.
>
> When a page is requested, if it's in the cache, we have no problem.
>
> Otherwise, we "page fault" and read the page into the cache at a cost of 1, evicting at least one page to do it.
>
> We'd like to choose a page eviction strategy with a minimal cost.

21.1 Optimum Offline Strategy

The optimum strategy is to evict the page with the next request furthest in the future. The proof is done from the observation that we can modify any optimum solution to this one, decision by decision.

21.2 Online Cache Strategies

FIFO - first in first out

LRU - least recently used

LFU - least frequently used

21.2.1 LRU vs FIFO

Despite LRU being better than FIFO in practice, both LRU and FIFO have competitive ratio k.

We can prove this by dividing a request sequence into phases. A phase stops just before the $k+1$th different page is requested.

The algorithms use $\leq k$ swaps per page, since LRU and FIFO will not evict a page used in that single phase. We know that OPT must evict ≥ 1 page per phase, because there are $k+1$ distinct pages involved. Thus, we have $\frac{A}{\text{OPT}} \leq k$, plus an additive constant for a partial phase at an end of a request sequence.

21.2.2 Limitations of Deterministic Selection

We'd like to prove that any deterministic algorithm has competitive ratio $\geq k$.

If k is the cache size, and the number of pages is $k+1$, and the adversary can always supply a sequence of length n asking for the page known not to be in the cache.

Since there are n swaps in a sequence of length n, and we know that a perfect solution subjected to this adversary would use $\frac{n}{k}$ swaps, because each time it evicts, it must be good for the next k requests.

Then we have $\frac{A}{\text{OPT}} \geq k$ for all deterministic algorithms.

21.2.3 Randomized Page Swapping Algorithm

This randomized algorithm is attributed to Fiat in '91.

```
serve(p):
    p.makeRecent()
    if !cache.contains(p):
        if all pages are recent
            pages.all.makeNotRecent()
        evict a random non-recent page
```

Without proof, the expected competitive ratio is $\Theta(\log k)$, which is the best possible for randomized algorithms, assuming the adversary does not see the random choices.

21.3 k-Server Problem

The problem formulation is as follows:

> There are k servers to service requests in metric space at points $\{p_1, \ldots, p_n\}$.
>
> When a request for p_i occurs, if a server is at p_i, fine.
>
> Otherwise, move a server from its location (p_j) to p_i at cost $d(p_j, p_i)$.
>
> We'd like to serve requests in a given order while minimizing the total distance.

Paging is a special case of this algorithm where $d(p_i, p_j) = 1$ for all i, j.

Without proof, we state that the offline algorithm can solve the problem in poly-time.

21.3.1 Greedy Online Algorithm

Given that all points are in 2D, our heuristic is to move the closest server to the point.

This unboundedly bad, as a sequence of $(p_1, p_2, p_1, p_2, \ldots)$ will take more time than necessary.

It is OPEN[1] if there is a k-competitive algorithm that solves the problem, in any dimension.

The best known algorithm is $(2k - 1)$-competitive ('94).

21.3.2 k-Competitive Algorithm for Points on a Line

For points on a line, we split up requests r_i into three types:

1. If r_i is to the right of all servers, move the rightmost server right.
2. If r_i is to the left of all servers, move the leftmost server left.
3. If r_i is between two servers s_a, s_b, move them both towards r_i, stopping both when one reaches the request.

If multiple servers arrive at the same location, break ties arbitrarily.

We present that (without proof) this algorithm is k-competitive.

[1] Since 1988, the time is apparently nigh

Chapter 22

Fixed Parameter Tractable Algorithms I

We know that NP-COMPLETE problems seem to only have exponential time algorithms, but we'd like to classify "how exponential" these problems are.

22.1 Completing Problems with Fixed Parameters

When solving the Vertex Cover problem to find $C \subseteq V$ such that every edge has at least one end point in C.

Say that k is the minimum size vertex cover, and k is known.

We try to find all subsets of k vertices $\binom{n}{k}O(n^k) = O(\binom{n}{k}n^k) = O(kn^{k+1})$, which is polynomial time for constant k.

The same idea works for both clique and independent set but not for graph coloring[1].

22.2 A feel for Fixed Parameter Tractable Algorithms

While $O(n^k)$ is not exponential, it's still pretty bad.

We'd much prefer $O(f(k)n^c)$ for some polynomial $f(k)$ independent of n and some constant c independent of k. Even better would be $O(f(k) + n^c)$.

22.2.1 FPTA for Vertex Cover

We want to build a FPTA for Vertex Cover by branching on all possibilities that for $e = (u, v)$, either u or v is in C (where C is the cover).

At each tree, we pick an uncovered edge to branch on.

```
vertex_cover(E, V, k):
```

[1] Given a graph, is it 3-colorable is NP-COMPLETE, so we still run into that issue.

```
if E == {} return true
if k == 0 return false
pick e = E.random()
(u, v) = e
return vertex_cover(E.without_incident(u), V–u, k–1)
    || vertex_cover(E.without_incident(v), V–v, k–1)
```

The tree has depth k, so we take time $O(2^k n)$.

By modifying the algorithm, we can trivially find the vertex cover itself.

22.2.2 Kernelization

If we wanted to, we can improve the Vertex Cover technique to $O(f(k) + n^c)$ using kernelization.

Essentially, if there exists a vertex v with $\deg(v) > k$, then v must be in the cover C, otherwise we'd need all neighbors of v (of which there are $> k$) to be in C.

```
vertex_cover_kernelized(E, V, k):
    c_2 = all vertexes with deg(v) > k
    k_2 = k - |C_2|
    V_2 = V - C_2
    E_2 = E.without_any_incident(C_2).remove_isolated()
    if |V_2| > 2k^2
        return false
    return VC(E_2, V_2, k_2)
```

The actual vertex cover is $C' \cup \text{VC}(E', V', k')$ We know that the maximum degree in V' is $\leq k$. If (V', E') has a vertex cover of size $\leq k$, then V' has $\leq k^2$ edges. So $|E'| \leq 2k^2$, which is not very big.

The idea of kernalization is due to Prof. J. Buss in '93.

The call to VC takes $O(2^k|E'|) = O(2^k 2k^2)$, and finding C' takes $O(m + n)$ time. Thus, the total run time is $O(2^k k^2 + m + n)$.

22.3 Defining Fixed Parameter Tractable Algorithms

A problem is fixed parameter tractable (FPT) in parameter k if it has an algorithm[2] with runtime $O(f(k)n^c)$, wher n is the input size, $f(k)$ is independent of n, and c is a constant independent of k.

Given a FPT problem, we can get an algorithm with runtime $O(f'(k)+n^{c'})$. This is neither deep, indicative on how to construct this, nor useful, but is interesting.

22.3.1 Common Parameter Examples

We can pick a few types of examples of parameters:

[2]The algorithm is a Fixed-Parameter Tractable Algorithm (FPTA).

- Value of the OPT solution
- Maximum degree of the graph
- Dimension for geometric problems
- Genus of a graph[3]

Refer to Chapter 23 for many FPTA examples.

22.4 Randomized FPT Algorithm for k-Path

> Given graph G, $k \in \mathbf{N}$, and a starting vertex s and an end vertex t, find a simple[4] $s \to t$ path with k internal vertexes.

In general, this is a NP-HARD problem, since if $k = n - 2$, we're asking for a Hamiltonian Path $s \to t$.

With the power of randomness, we can save the day!

Randomly color all vertexes into k different colors. Then look only for paths that use all k colors.

The algorithm will always say "NO" correctly, since it will never find a simple k-path within the coloring.

If there does exist a simple path P, we can say the following:

$$\begin{aligned}\Pr(\text{P is colorful}) &= \frac{k!}{k^k} \\ &\geq \frac{\left(\frac{k}{e}\right)^k}{k^k} \\ &= \frac{1}{e^k}\end{aligned}$$

So the algorithm is correct when it says "YES" with a probability $\geq \frac{1}{e^k}$.

Defining $p = \frac{1}{e^k}$, then we can say:

Since A is correct with Probability $\geq p$, then after $\frac{1}{p}$ repetitions, the probability of failure is

$$\begin{aligned}(1-p)^{\frac{1}{p}} &< (e^{-p})^{\frac{1}{p}} \\ &= \frac{1}{e}\end{aligned}$$

In our case, repeating e^k times gives us the probability of error $\leq \frac{1}{e}$.

Finding a colorful path for a given ordering is easy, so we find all of them.

The runtime of setting up and searching orderings is $O(k!m)$, so the final runtime is $O(e^k 2^k m)$, with a probability of error of $\leq \frac{1}{e}$.

[3] Apparently, this means 0 for planar graphs, and 1 for embeddable on a torus without crossing edges.
[4] Don't repeat vertexes.

Chapter 23

Fixed Parameter Tractable Algorithms II

The Independent set problem is expressed as follows:

> Given a graph G, does it have an independent set[1] $\geq k$?

The brute-force time is $O(n^k(n+m))$, where there are k subsets of n vertexes. The brute-force solution not FPT.

In fact, it is OPEN if there is a FPTA for Independent Set (and this parameter), and it's also OPEN if existence of a FPTA implies P = NP.

23.1 FPTA for Independent Set

The general idea is to use a parameter for a FPTA that measures the "tree-ness" of our graph.

23.1.1 Independent Set on a Tree

We want to find the independent set on a tree.

We define the recursive functions $IS(v)$ and $IS'(V)$, which are the maximum weight of the independent set of the subtree rooted at v that do ($IS(v)$) or do not ($IS'(v)$) include v.

We can even put weights on vertexes as $w(v)$.

```
ind_set_tree(V):
    for v = V.leaves:
        IS_2[v] = 0
        IS[v] = w(v)
    for v in V.vertexes-in-leaf-to-root-order:
        // where v_i are the children of v...
        IS_2[v] = sum_i(IS[v_i])
        IS[v]   = max(w(v) + sum_i(IS_2[v_i]), IS_2[v])
```

[1] of vertexes that share no edges

```
return IS[root]
```

23.1.2 Independent Set on Graphs that are "Almost" Trees

Series Parallel Graphs

A Series-Parallel graph (SP) is defined recursively as:

- A single edge connecting two vertexes.
- Two parallel SP graphs sharing the same start and end vertexes.
- Two SP graphs connected in series.

Refer to Figure 23.1 for an example. We can find the Independent set in series parallel graph by dynamic

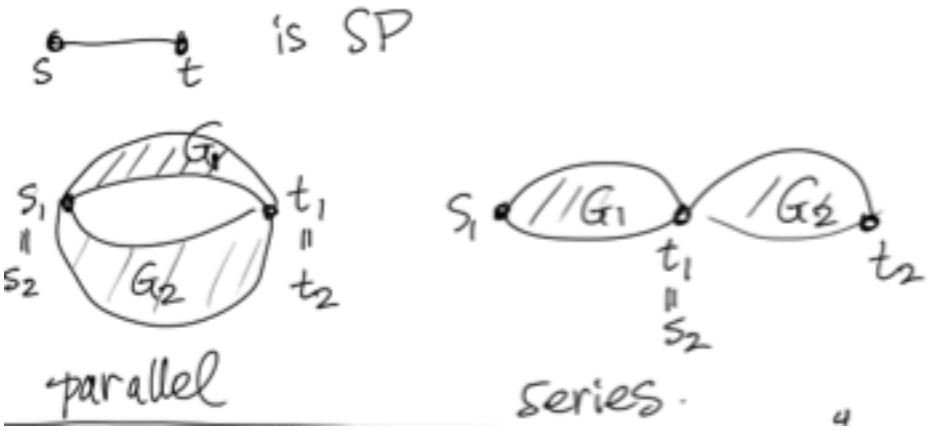

Figure 23.1: Composition of SP Graphs

programming based on the maximum independent set for all permutations containing or not containing s or t.

23.1.3 Decomposing Series Parallel Graphs

We can model the decomposition of a SP as a tree.

Vertexes that represent a parallel decomposition are a 2-tuple of (s, t). Vertexes that represent a series decomposition are a 3-tuple of $(s, v_{\text{middle}}, t)$. Edges in the graph appear as leaf vertexes.

We have two properties:

1. If $e = (u, v)$ is an edge of G then u and v trivially appear together in a tree node.
2. Every vertex v of G corresponds to a sub-tree T.

23.1.4 Generalization to General Graphs

The concept of Tree-Width was created by Robertson & Seymour as part of the "Graph Minors Project", the result of 20 papers, totaling around 500 pages.

We want to represent a graph as a tree, and have properties 1 and 2 from above.

I believe that we define bags as a 2-or-3-tuple of a vertex.

The width of a decomposition is the size of the largest bag in the tree - 1.

The tree width of G is the minimum width of any tree decomposition, which is $\leq n - 1$.

And we only need $\leq n$ bags in any tree decomposition.

Finding the tree-width of a graph is NP-HARD, but there is a FPT algorithm $O(2^{O(k^3}n)$.

23.1.5 Graphs of Tree Width

The maximum weight of the independent set in a graph of a tree-width k can be found in time $O(2^k n)$.

The idea of the proof is to use DP to the tree upwards.

For each bag B of size $\leq k+1$ we find for each subset $A \subseteq B$ (there are $O(2^k)$ of them), a maximum weight independent set including A (excluding $B - a$) in the subtree rooted at B.

23.1.6 Other Problems FPT in Tree-Width

- 3-Coloring
- Minimum coloring
- Hamiltonian cycle (apparently it's even more complicated)

23.1.7 Hardness Results of FPT Problems

All relative results are of the form: A FPT algorithm existing for X implies there is a FPT algorithm for Y, and is proved through a reduction that preserves the parameter and the FPT.

Appendix A

Sample Algorithms

A.1 QuickSort

We have data $S = \{s_1 \dots s_n\}$.

```
def QuickSort(S):
    if n == 0 or n == 1:
        return S
    i = random(1, n)
    L = {s_j : s_j < s_i}
    M = {s_j : s_j = s_i}
    R = {s_j : s_j > s_i}
    return QuickSort(L).append(M).append(QuickSort(R))
```

In the worst case, this runs in $T(n) = T(n-1) + O(n) = O(n^2)$ time.

We "expect" the pivot to be in the middle, so $|L| = |B| = \frac{n}{2}$ and $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.[1]

More formally, we'll analyze randomized QuickSort with recursive calls of $T(\ell) + T(n - \ell - 1)$.

By random choice of pivot, ℓ is equally likely to be each of $\{0 \dots n-1\}$, each with a $\Pr(\frac{1}{n})$. Then, we get:

$$\begin{aligned} T(n) &= \frac{1}{n}\left(\sum_{\ell=0}^{n-1} T(\ell) + T(n-\ell-1)\right) + O(n) \\ &= \frac{2}{n}\left(\sum_{\ell=0}^{n-1} T(\ell)\right) + O(n) \end{aligned}$$

Using a proof by induction, we arrive at $T(n) = O(n \log n)$, which means that we can expect quicksort to take $O(n \log n)$ time[2].

[1] In this case, we use "expectations" that the input is "average", which is not amortized analysis. Better analysis is below.
[2] In CLRS, there is nice analysis without recurrence relations.

Appendix B

Math Review

B.1 Expected Values - Statistics

The expected value of the random variable X is $E(X)$. For discrete X, we know:

$$E(X) = \sum_x xPr\{X = x\}$$

For any X and Y, we know that:

$$E(X + Y) = E(X) + E(Y)$$

For a constant c, we know:

$$E(cX) = cE(X)$$

If X and Y are independent random variables, we know:

$$Pr\{X = x \text{ and } Y = y\} = Pr\{X = x\}Pr\{Y = y\}$$
$$E(XY) = E(X)E(Y)$$

More details are in CLRS.

B.2 Markov's Inequality

If the random variable $X \geq 0$ and $E(X) = \mu$, then $\Pr\{X \geq c\mu\} \leq \frac{1}{c}$ for a constant c.

$$\begin{aligned}
\mu &= E(X) \\
&= \sum x \Pr(X = x) \\
&\geq \sum_{x \geq c\mu} x \Pr(X = x) \\
&\geq c\mu \sum_{x \geq c\mu} \Pr(X = x) \\
&= \Pr(X \geq c\mu) \\
\Pr(X \geq c\mu) &\leq \frac{\mu}{c\mu} \\
&= \frac{1}{c}
\end{aligned}$$

This proof can be found on page 5 of the Lecture 11 notes.

B.3 Logic

B.3.1 Contrapositive

The contrapositive is defined as $(A \implies B) \implies (\neg B \implies \neg A)$.

B.3.2 Conjunctive Normal Form (CNF)

Given a boolean formula, CNF is the form:

$$E = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_3)$$