richardwu.ca

CS 466/666 Course Notes

Design and Analysis of Algorithms

Anna Lubiw • Fall 2018 • University of Waterloo

Last Revision: December 12, 2018

Table of Contents

Abstract

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. These notes are my interpretation and transcription of the content covered in lectures. The instructor has not verified or confirmed the accuracy of these notes, and any discrepancies, misunderstandings, typos, etc. as these notes relate to course's content is not the responsibility of the instructor. If you spot any errors or would like to contribute, please contact me directly.

1 September 10, 2018

1.1 Overview

How to design algorithms Assume: greedy, divide-and-conquer, dynamic programming

New: randomization, approximation, online algorithms

For one's basic repertoire, assume knowledge of basic data structures, graph algorithms, string algorithms.

Analyzing algorithms Assume: big Oh, worst case asymptotic analysis

New: amortized analysis, probabilistic analysis, analysis of approximation factors

Lower Bounds Assume: NP-completeness

New: hardness of approximation

1.2 Travelling Salesman Problem (TSP)

Given graph (V, E) with weights on edges $W : E \to \mathbb{R}^{\geq 0}$ find a *TSP tour* (i.e. a cycle that visits every vertex exactly once and has minimum weight or $\min \sum_{e \in C} w(e)$).

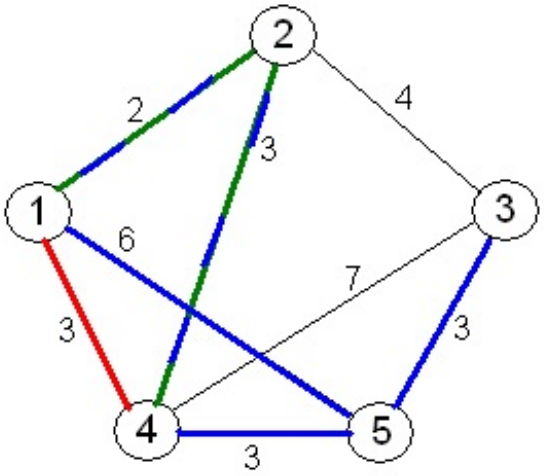


Figure 1.1: TSP tour is highlighted in blue.

Usually assume a **complete** graph (all possible $\binom{n}{2}$ edges exist). We can add missing edges with *high weight* to convert non-complete to complete.
Applications of TSP:

- School bus routes

- Delivery
- Tool path in manufacturing

To show the *decision version* of TSP (exist a tour of total weight $\leq k$) is NP-complete:

1. Show it is in NP (i.e. provide evidence (the tour itself) that there exists a TSP tour and show weights add up to $\leq k$ in polynomial time)
2. Show a known NP-complete problem reduces in polynomial time ($\leq_p$) to TSP (the Hamiltonian cycle problem can be reduced to TSP)

1.3 Approach to NP-complete problems

For NP-complete problems we want to:

- Find exact solutions
- Find fast algorithms
- Solve hard problems

We can in effect only choose two: for hard problems we give up on either *fastness* (exponential time algorithms) or *exactness* (approximation algorithms).

1.4 Metric TSP

An appproximation exists for the **metric TSP** version, where:

- $w(u, v) = w(v, u)$
- $w(u, v) \leq w(u, x) + w(x, v) \quad \forall x$

An algorithm (1977) was proposed for metric TSP:

1. Find a minimum spanning tree (MST) of the graph
2. Find a tour by walking *around* the tree.

   Think of doubling edges of MST to get Eulerian graph (i.e. every vetex has even degree), which lets us find an Eulerian tour traversing every edge once.

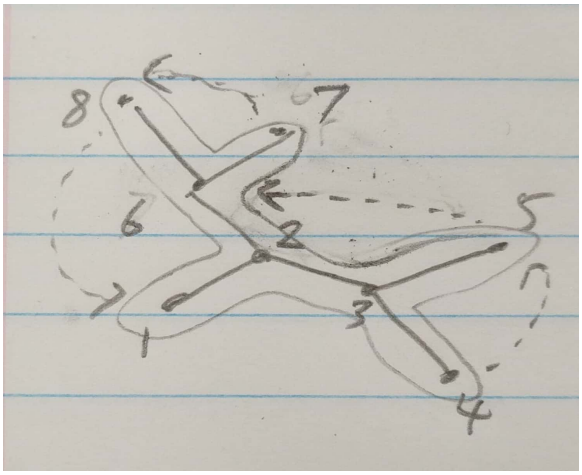


Figure 1.2: Eulerian tour is the solid line around the MST, the dotted lines show shortcuts taken, and the nodes are labelled in order.

3. Take shortcuts to avoid re-visiting vertices

   Instead of traversing a node twice (when walking around the MST), we take shortcuts and jump directly to the next unvisited node. By the triangle inequality our path should have a shorter path than if we actually traversed the MST edges twice, i.e.:

   $$l \leq 2l_{MST}$$

   where l is the length of our tour and l_{MST} is the length of the MST (remember we doubled every edge).

   Note the total path length we get will differ depending on which node we start with: thus one *could* attempt all paths to find the best path out of all approximated paths.

   Lemma 1.1. This algorithm is a **2-approximation**, i.e.:

   $$l \leq 2l_{TSP}$$

   where l_{TSP} is the minimum length of TSP.

   Proof. We need to show $l_{MST} \leq l_{TSP}$.

   Take the minimum TSP tour. Throw out an edge. This is a spanning tree T. Since

   $$l_{MST} \leq l_T \leq l_{TSP}$$

   the result follows. □

   Exercise 1.1. Show factor 2 can happen.

Analyzing/implementing this algorithm (let n # of vertices, m # of edges):
Steps 2 and 3 take $O(n + m)$.
Step 1 is our bottleneck: we've seen *Kruskal's* (sorted edges with union find to detect cycles) and *Prim's* (add shortest edge to un-visited vertex) MST algorithms.
Prim's took $O(m \log n)$ using a heap. An improvement is using a *Fibonnaci heap* (1987) which improves runtime for MST to $O(m + n \log n)$. A further improvement uses a randomized linear time algorithm for finding the MST (1995).

Theorem 1.1. For general TSP (no triangle inequality) if there is a polynomial time algorithm k-approximation for any constant k, then $P = NP$.

Proof. Exercise (hint: start with $k = 2$ and the Hamiltonian cycle problem. Show the 2-approximation can be used to solve the HC problem). □

Can we improve factor of 2 for metric case? Yes (Christofides 1996):

1. Compute MST
2. Look at vertices of odd degree **in MST** (there will be an even number). Find a minimum weight *perfect matching* of these vertices.

   The MST and perfect matching is Eulerian (since we added matchings between odd vertices): take an Eulerian tour and take shortcuts (as before).

Implementation: we need a matching algorithm - the best runtime (in this situation) is $O(n^{2.5}(\log n)^{1.5})$ (1991).

Lemma 1.2. We claim $l \leq 1.5 l_{TSP}$. Note that $l \leq l_{MST} + l_M$ (where l_M is the total length of the minimum weight perfect matching). We must show that $l_{MST} \leq l_{TSP}$ and $l_M \leq \frac{1}{2} l_{TSP}$.
Sketch: to show $l_M \leq \frac{1}{2} l_{TSP}$, we show the smallest matching is $\leq \frac{1}{2} l_{TSP}$.

Open question: do better than 1.5 for metric TSP. We know the lower bound is 1.0045 (if we could get 1.0045-approximation then $P = NP$).
There is also the **Euclidean TSP** version where $w(e) =$ Euclidean length. We can get ϵ-approximation $\forall \epsilon > 0$.

2 September 12, 2018

2.1 Data structures

Every algorithm needs data strutures. Assume knowledge of:

- Priority queue (heap)
- Dictionary (hashing, balanced binary search trees)

In this course, we look at fancier/better DSes and also amortized analysis.

2.2 Priority queue

Operations supported by a priority queue (PQ) are: insert, delete min (delete), decrease-key, build, merge.
We usually implement PQs with a **heap**: a binary tree of elements where the parent is $\leq$ than the left and right children (min-heap), therefore the min. is at the root.

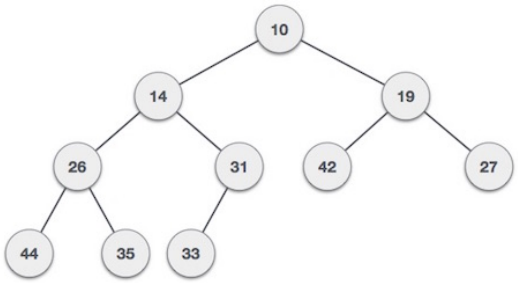


Figure 2.1: An example of a (min-)heap.

We assume that the shape is that of an almost perfect binary tree, where we always add a new element to the bottom right of the tree (if incomplete level) or the bottom left (start of a new level). We can store heaps in *level order in an array*, where for a given element indexed at i, acessing the parent via index $\lfloor \frac{i}{2} \rfloor$ and accessing the children via indexes $2i+1$ and $2i+2$.
The height of the tree is obviously $\theta(\log n)$. To implement each operation:

Insert Add new element in last position and bubble/sift up to recover ordering property. $\theta(\log n)$.

Delete min Remove root, it's the minimum. Move last position element to root and bubble/sift down (swap with smaller child). $\theta(\log n)$.

Derease-key Need only bubble/sift up (if $<$ parent). $\theta(\log n)$.

Build Repeated insertion is $\theta(n \log n)$.

Better approach: from bottom to top (after newly initialized heap in-place) bubble/sift down each element. $\theta(n)$.

2.3 Prim's algorithm

An application of heaps/PQs is for **Prim's MST algorithm**. Given graph with weights on edge, find spanning tree of minimum sum of edge weights.
For our given tree T so far, we find the minimum weight edge connecting to a new vertex. We begin with a PQ of all the edges and we will need to delete any newly added edge and any edges that lead to a newly added vertex.
Let $m = |E|$ number of edges and $n = |V|$ number of vertices. Every edge joins and leaves the heap once for a total of m times. We do delete min n times, so we have $\theta((m+n)\log m)$.
A better approach by using a heap of vertices and keeping track of shortest distance from T to vertex v gives us $\theta((m+n)\log n)$, which is only a constant time improvement since $\log m = \log n^2 = 2\log n$. We require the decrease-key operation here.
An even better approach for Prim's yields $\theta(m + n\log n)$ via *Fibonacci heaps*.

2.4 Binomial heaps

Binomial heaps improve the merge operation for heaps (which we can use for all other operations).
We use pointers to implement trees and each parent has an arbitrary number of k children (not necessarily binary tree) while maintaining heap order where parent is $\leq$ key of all children. We thus need to relax the shape; we also allow *multiple trees* for a given heap.
We define binomial trees B_k in terms of their rank k, which also coincides with the degree of the root.

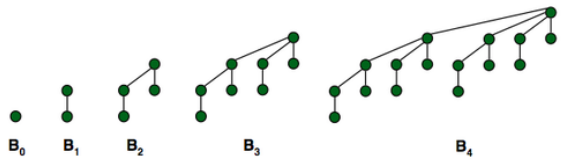


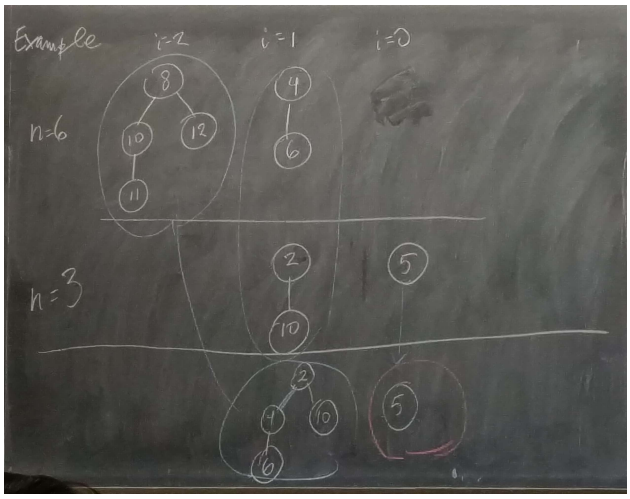
Figure 2.2: Example of five binomial trees with ranks $0, 1, \ldots, 4$.

In general, the number of node in B_k is $2^k = 2^{k-1} + 2^{k-1}$.
The height of B_k is k since, by induction, we have the recurrence $height(k) = 1 + height(k-1)$.

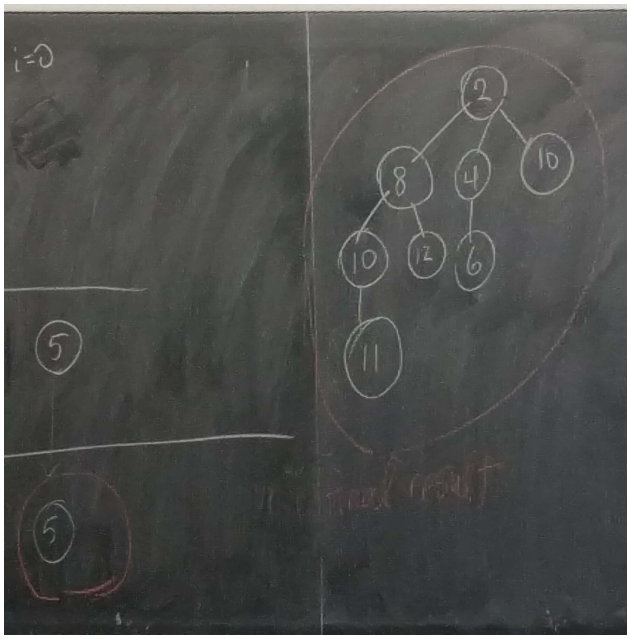
The number of nodes at depth i in B_k is

$$\binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$$

(show by induction).
Brief counting proof of binomial equivalence: given k items from which we want to choose i items, we can either picked the first item or not: if we did not pick the first element, then we need to pick i items from the remaining $k-1$ items; if we did pick the first element, we need to pick $i-1$ items from the remaining $k-1$ items.
Note that B_k's only permit powers of 2 number of elements: thus a **binomial heap** for n elements use a collection of B_is (heap ordered), at most one for each rank.
E.g. for $n = 13$, we have $13 = 2^3 + 2^2 + 2^0$ (from binary 1101 so we use B_3, B_2, B_0).
It *does not matter* which B_i's contain a particular element.
We use $\theta(\log n)$ trees for a given binomial heap with n elements (from binary number expansion).
For merging two binomial heaps, we follow addition of two binary numbers (which are our bitmasks of trees). E.g. for adding a heap with 6 elements to a heap with 3 elements, we add $110 + 11 = 1001$ resulting in a binomial heap with a B_3 and B_0 tree.
When merging two trees of the same rank k, we simply take the tree with the larger root and add it as a children of the root of the other tree.



(a) We carry through the B_0 from the second heap and merge the B_1 trees from two binomial heaps by making the tree with the larger root the children of the other root.



(b) After merging the newly B_2 with the B_2 tree from the first heap into a B_3 tree, we get our final resulting binomial heap with a B_0 and B_3 tree.

Analysis of operations:

Merge Joining two B_i's take $\theta(1)$. We join up to $\log n$ trees so we have $\theta(\log n)$.

Insert Merge binomial heap with single B_0 (one new element): again $\theta(\log n)$ (since we have the worst case when we insert into a heap with $2^m - 1$ elements).

Delete min Takes $\theta(\log n)$ to find the minimum by checking roots of all trees. Once removing the root from the

tree B_k, we end up with $k-1$ trees $(B_{k-1}, \ldots, B_0)$ which we need to merge with the other trees (merge operation is also $\theta(\log n)$) so we have $\theta(\log n)$ overall.

Decrease-key After decreasing key, we bubble/sift up as necessary like before, which takes $\theta(\log n)$ since each tree has height at most $\log n$.

Build Repeated insertion appears to be $O(n \log n)$, but in fact it is $\theta(n)$. This can be seen by repeated addition of 1 in binary to our cumulative total: we only do merges at certain times.

Proof. In general, the cost of incrementing a k-bit counter has a *worst-case cost* of $k+1$ ($\theta(k)$ bit operations) where you add 1 to $11\ldots1$.

We show that incrementing from 0 to n is $\theta(n)$:

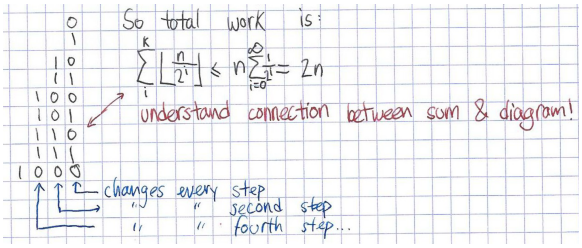


Figure 2.4: A sequence of incrementing from 0 to 1000_2 or 8_{10}. We observe that for binary bit i (rightmost is bit 0) the value of it changes every 2^i step. Thus for incrementing up to n, we sum up the number of times each bit i changes for k bits, which gives us $2n$ changes.

□

3 September 17, 2018

3.1 Amortization

Definition 3.1 (Amortized cost)**.** A sequence of m operations takes total cost $T(m)$: the **amortize cost** of one operation is thus $\frac{T(m)}{m}$.

3.2 Amortization "potential" method

Idea: use an "accounting trick" ("potential" in the physics sense)

Potential "savings" in bank account.

Cost the "true" cost.

Charge artificial: over/under-estimate cost at time of operation.

If charge > cost, we put excess in the bank (add to potential).

If cost > charge, extra has to come out of bank account.

Let Φ_i denote the potential after the ith operation, thus

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

Theorem 3.1. If the final potential $\geq$ initial potential (almost always 0) then the amortized cost per operation $\leq$ max charge.

Proof. The total charge we've introduced is

$$\begin{aligned}\sum_{i=1}^{m} \text{charge}(i) &= \sum_{i=1}^{m} \text{cost}(i) + \sum_{i=1}^{m} \Phi_i - \sum_{i=1}^{m} \Phi_{i-1} \\ &= \sum_{i=1}^{m} \text{cost}(i) + \Phi_m - \Phi_0\end{aligned}$$

Since $\Phi_m - \Phi_0 \geq 0$ then $\sum_{i=1}^{m} \text{charge}(i) \geq \sum_{i=1}^{m} \text{cost}(i)$.
Recall that we have for amortized cost

$$\frac{\sum \text{cost}(i)}{m} \leq \frac{\sum \text{charge}(i)}{m} \leq \text{ max charge}$$

□

To do potential analysis, devise potential/charge for each operation such that $\Phi_m \geq \Phi_0$ (the bank is never "in the red") and max charge is *small*.

Example 3.1. Applying the potential method to the binary counter example, add an extra "\$1" to each basic bit operation to compensate for when we roll over from string of all ones i.e. charge$(i) = 2$ (1 for the bit operation and "storing" the other 1 for the future).
By the theorem (assuming hypothesis holds), our amortized cost should be 2 per op.
Let's verify, initial potential $\Phi_0 = 0$

counter	cost	charge	potential
0 0 0 0	1		0
0 0 0 1	1	2	1
0 0 1 0	2	2	1
0 0 1 1	1	2	2
0 1 0 0	3	2	1
0 1 0 1	1	2	2

Intuition: potential is equal to the # of ones in counter, so final potential ≥ 0 (initial potential). Note that with cost of $\frac{3}{2}$ this fails.

Claim. Potential = # of ones in binary expansion of counter.
If this claim holds, final potential always ≥ 0 since we have at least one 1 in binary counter.

Proof. Proof by induction. Suppose current counter is

$$\begin{aligned}&01011\ldots011\ldots1 \\ &\quad\ldots100\ldots0\end{aligned}$$

where we have t_i ones at the end.

We thus have

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

Where we zeroed out our t_i ones (subtract) and added one 1. □

While potential method seems harder than previous sum argument, it is much more powerful in general.
This gives us $\theta(n)$ since if amortized cost is p (some constant), then n ops cost $pn \in \theta(n)$.

3.3 Summary of mergeable heaps

	binomial heap	lazy binomial heap	Fibonacci heaps
insert	$O(\log n)$	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$ (amortized)	$O(\log n)$ (amortized)
merge	$O(\log n)$	$O(1)$	$O(1)$
decrease-key	$O(\log n)$ (bubble-up)	$O(\log n)$	$O(1)$ (amortized; improves MST time)
build	$\theta(n)$	$O(n)$	$O(n)$

3.4 Lazy binomial heaps

Idea: be lazy on merge/insert i.e. allow *multiple trees of same size*. Catch up on delete-min: re-combine to form a proper binomial heap (i.e. when delete-min occurs).
For delete-min with lazy binomial heaps:

1. Look at all roots to find min, remove this root.
2. Consolidate trees:

```
1    for rank = 1 to max rank:
2      while there are >= 2 trees of this rank:
3        link them into one tree
```

   where max rank is $\theta(\log n)$

Recall that rank = degree of root = height of tree.
It seems the worst case cost is $\theta(n)$ (after inserting n singletons).

Theorem 3.2. Lazy binomial heaps have $O(\log n)$ amortized cost for delete-min and $O(1)$ for insert/merge.

Proof. Let the potential be the # of trees and $\Phi_0 = 0$.
Clearly $\Phi_m \geq 0$ (we never have negative # of trees) so our previous result for the potential method applies.
We need to determine the charge per operation. Recall

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

Merge cost is 1 (not doing anything), and # of trees is the same (we combine the potentials of the two trees, no additional trees). So we have charge of 1.

Insert cost is 1, # of trees increases by 1, so we have charge of 2.

Delete-min Let r be the degree of the min node ($O(\log n)$), t be the number of trees before delete-min (Φ_{i-1}).

Thus consolidation will be invoked on $t - 1 + r$ number of trees.

Thus the cost will be $\leq t - 1 + r + O(\log n)$, where we have to merge/link at most $t - 1 + r$ times.

$O(\log n)$ is our loop from rank 1 to max rank ($O(\log n)$) and also keeping track of the # of trees of each rank.

Ultimately $\Phi_i \in O(\log n)$ since we end up with a Binomial heap with $O(\log n)$ trees.

We have

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

since $r \in O(\log n)$.

Thus delete-min has $O(n)$ worst case but $O(\log n)$ amortized.

□

3.5 Fibonacci heaps

Improvement on lazy binomial heaps by reducing `decrease-key` to $O(1)$ amortized time (vs $O(\log n)$).
Recall `decrease-key` bubbles-up O(height of tree): instead cut the tree. However we must be careful of the # of resulting trees and the nodes per tree changes (not 2^i anymore).
Instead, the number of nodes in a Fiobnacci tree of rank k is $\geq (k+2)$th Fibonacci number (f_{k+2}), where the Fibonacci numbers are $0, 1, 1, 2, 3, 5, \ldots$ with $f_0 = 0$.
This gives rank $\in O(\log n)$ because $f_k \in \theta(\phi^k)$.

4 September 19, 2018

4.1 Splay trees

Recall: a dictionary has keys from *totally ordered* universe and supports the operations **insert**, **delete**, and **search** (by key).
They can be implemented via hashing or balanced binary search trees. Recall for a **binary search tree** we have:

Search follow search tree invariant (left subtree $<$ root, right subtree $>$ root)

Insert insert where search fails

Delete Replace node to be deleted by either in-order successor (left-most child in right sub-tree), OR in-order predecessor (right-most child in left sub-tree).

Recursively delete chosen successor or predecessor, respectively.

Obviously if node to be deleted has 0 or 1 child, one can simply attach the child to the parent of the deleted node.

All operations for a binary search tree take $O(\text{height of tree})$; if balanced then height $\in O(\log n)$.
Some balance search trees variants include the **AVL tree** and **red-black** tree, which both employs rotation to balance the tree.

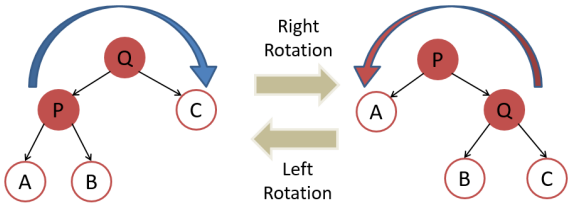


Figure 4.1: Right and left tree rotations. Note that the shorter paths that do not go through both P and Q become longer after a rotation. This helps balances out the height whenever a violation of the search tree invariant (AVL vs red-black) is violated.

Splay trees (Sleator & Tarjan, 1985) are a variant of balanced binary search trees

- $O(\log n)$ amortized cost per operation
- Easier to implement than AVL and red-black trees
- Do not need to keep balance information
- Careful: tree may become unbalanced
- Danger: repeated search for deep nodes.
  Fix: adjust tree whenever node is "touched".

The operations for a splay tree are

Splay(x) repeat until some target x node is root. We have up to 3 cases for where x is relative to its parent and grandparent:

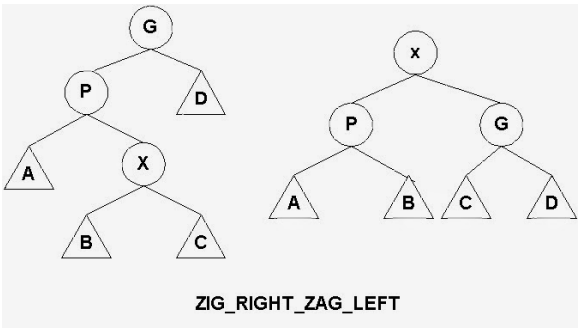


Figure 4.2: Zig-zag case: We want to lift x to the root where there is a zig-zag pattern up though parent P and grandparent G. We perform a left-rotation on x first, then perform a right rotation on x.

Case 1: zig-zag Equivalent to two rotations on x.

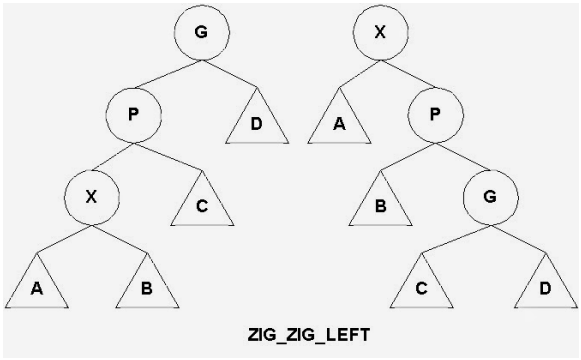


Figure 4.3: Zig-zig case: We want to lift x to the root where there is a straight pattern up though parent P and grandparent G. We perform a right rotation on P first then a right-rotation on x.

Case 2: zig-zig Equivalent to one rotation on y then one rotation on x.

Case 3: no grandparent or "zig" Single rotation on x.

Search After finding x (or place where search fails) using the usual search algorithm, we splay on x.

Insert Do usual insert on x, then splay(x).

Delete Do usual delete then splay parent of removed node.

4.2 Amortized analysis of splay trees

Goal: $O(\log n)$ amortized cost per operation. Recall that

$$\text{charge} = \text{cost} + \Delta\Phi$$

wher $\Delta\Phi = \Phi_i - \Phi_{i-1}$ or the change in potential.
Denote $D(x)$ as the # of descendants of x (including x itself).
Denote $r(x) = \log D(x)$, which is the best height possible for subtree rooted at x given $D(x)$.
Define our potential $\Phi(\text{tree}) = \sum_{x \text{ a node}} r(x)$.
For a degenerate tree with one single path of nodes down (height n), we have

$$\Phi = \sum_{i=1}^{n} \log i \in O(n \log n)$$

For a perfectly balanced tree, note that for a given node at height h, it has 2^h descendants and thus $r(x_h) = \log 2^h = h$.
So we have

$$\begin{aligned}
\Phi &= \sum_{\text{all nodes}} \text{height} \\
&= \sum_{h=1}^{\text{height of tree}} h \cdot \frac{n}{2^h} \qquad\qquad \frac{n}{2^h} = \text{ \# of nodes at height } h \\
&= n \sum_{h=1}^{\text{height of tree}} \frac{h}{2^h} \\
&\in O(n)
\end{aligned}$$

where we use the identity $S = \sum \frac{i}{2^i} \in O(1)$ (proof: take $2S - S$ and cancel out individual terms of the expanded series).
We need to analyze each of

1. zig, zig-zag and zig-zig
2. splay(x)
3. insert, delete and search

Denote r' and D' as the new rank and new # of descendents, respectively.

Claim. Amortized cost of one operation (i.e. our charge per operation) on a node x is

$$\text{charge} \leq \begin{cases} 3(r'(x) - r(x)) & \text{for zig-zag and zig-zig} \\ 3(r'(x) - r(x)) + 1 & \text{for zig} \end{cases}$$

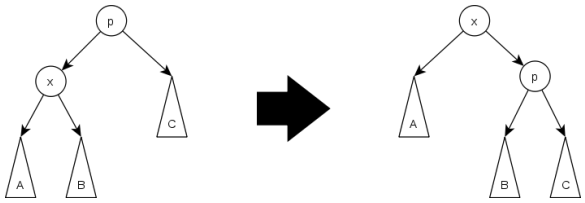


Figure 4.4: Zig (case 3) from before.

Proof. **Zig** Notice that $r'(x) = r(p)$ (i.e. x in the new tree has the same # of descendants or rank as old p).

Thus we have

$$\begin{aligned} \text{charge} &= \text{cost} + \Delta\Phi \\ &= 1 + r'(x) + r'(p) - r(x) - r(p) \\ &= 1 + r'(p) - r(x) && r'(x) = r(p) \\ &\leq 1 + r'(x) - r(x) && r'(p) \leq r'(x) \text{ since p child or x now} \\ &\leq 1 + 3(r'(x) - r(x)) && r'(x) - r(x) \geq 0 \text{ since x has at least the same subtree heights} \end{aligned}$$

where cost = 1 since we do 1 rotation.

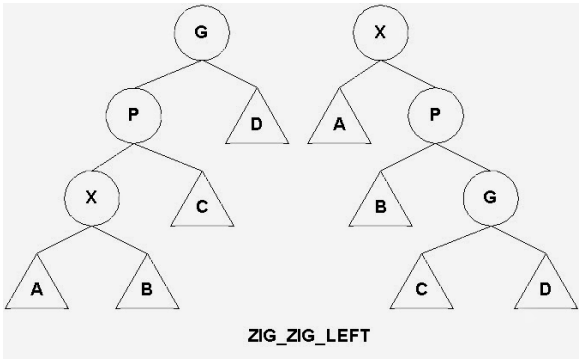


Figure 4.5: Zig-zig (case 1) from before.

Zig-zig Notice that $r'(x) = r(g)$ (same argument as before). Furthermore, $D(x) + D'(z) \leq D'(x)$.

Thus we have

$$\begin{aligned}
\text{charge} &= \text{cost } + \Delta\Phi \\
&= 2 + r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g) \\
&= 2 + r'(p) + r'(g) - r(x) - r(p) && r'(x) = r(g) \\
&\leq 2 + r'(x) + r'(g) - r(x) - r(p) && r'(p) \leq r'(x) \\
&\leq 2 + r'(x) + r'(g) - 2r(x) && -r(p) \leq -r(x)
\end{aligned}$$

where cost $= 2$ since we do 2 rotations.

If we show $2 + r'(x) + r'(g) - 2r(x) \leq 3(r'(x) - r(x))$ then we are done, i.e.

$$\begin{aligned}
& 2 + r(x) + r'(g) \leq 2r'(x) \\
\iff & \log D(x) + \log D'(g) \leq 2 \log D'(x) - 2
\end{aligned}$$

Note that $D(x) + D'(g) \leq D'(x)$ (from diagram), thus we essentially need to show

$$\log a + \log b \leq 2 \log c - 2$$

if $a + b \leq c$, which holds (proof left as exercise).

Zig-zag Similary proof as zig-zig.

Therefore we have charge $\leq 3(r'(x) - r(x)) + 1$ for all of zig, zig-zig, and zig-zag.

Splay(x)

Claim. Charge of splay(x) is $O(\log n)$.

If we add up $3(r'(x) - r(x)) + 1$ (charge for each individual zig, zig-zig, or zig-zag) as x goes up the tree, we get a telescoping sum $3(r(\text{root}) - r(\text{original x})) + O(\log n) \leq O(r(\text{root})) + O(\log n) = O(\log n)$.

Search, insert, delete

Theorem 4.1. The amortized cost of search, insert and delete are $O(\log n)$.

Proof. **Search** Note that

$$\text{charge } = \text{charge(splay}(x) + \text{ cost } + \Delta\Phi$$

where cost $+\Delta\Phi$ is the cost of other work (i.e. walking the path from root to x), which is $\leq$ cost of splay ($O(\log n)$), thus charge is $O(\log n)$.

Delete One node disappears, Φ goes down which is okay.

Insert Increase D for all nodes on path root to x.

We can prove this is $O(\log n)$.

□

□

4.3 Optimality conjecture for splay trees (open problem)

Splay trees are (within big Oh) as good as we can get with binary search trees, even by looking ahead at sequence of operations and planning rotations for any binary search tree variant.

5 September 25, 2018

5.1 Union find

Also known as **disjoint sets**: data structure for representing disjoint sets that supports efficient lookup of elements (and which set they belong to) and insertions/merges (between multiple disjoint sets).
Motivation: find all connected components of a graph. Note that depth first search would take $O(n+m)$ where n, m represents the number of node and edges, respectively.
Dynamic graph connectivity: able to maintain connected components as the graph changes. It allows us to answer queries such as "given vertices u, v are they connected"?
Some application examples include:

1. Social networks: relationships added/deleted
2. Minimum spanning tree: recall Kruskal's algorithm involves ordering the edges by weight $e_1, \ldots, e_m$ where we add e_i to our collection of components T iff e_i joins two different components.

   This is a special case of **incremental dynamic connectivity**: we add edges but don't delete any.

Supports two operations:

1. `Union(A,B)` - unite two sets A and B (destroys A, B)
2. `Find(e)` - which set contains element e

Analysis of Kruskal's using union find:

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

where sort takes $O(m \log m) = O(m \log n)$. We want the `Find`s and `Union`s to take $\leq O(m \log n)$.
In this analysis, we care about the sequence of `Union` and `Find` so amortized analysis is relevant.
From herein, let n denote the number of elements and m the number of operations where the # of unions $\leq n-1$.
There are multiple ways to implement union find.
Using an **array** $S[1 \ldots n]$ where $S[i]$ is the name of set containing element i. This means that `Find` is $O(1)$ but `Union` has worst case $O(n)$ (since we need to iterate through array and update all elements to new set name).
Tiny improvement: for `Union(A,B)` we update $S[i]$ for i in the smaller set of A, B. Thus if

$$A : 1, 3$$
$$B : 2, 7, 6, 5$$
$$C : 4$$

and we perform a union between A and B we only update $S[1], S[3]$ to B.
Note that the cost of all possible unions for n elements is $\leq O(n \log n)$ since each element changes its set $\leq \log n$ time (tree with leaves as each individual element; height is number of times an element changes set). Thus the cost of m operations is $O(m + n \log n)$ if # of finds is $\Omega(n \log n)$.
Thus for Kruskal's we have $O((m+n) \log n)$.
Abstractly, union find can be represented as a forest of n-ary trees where each tree is a disjoint set. The root of the tree is the "representative member". Second method: we use **pointers** to construct our forest of elements. Let *rank* of a node i be the length of the longest path from any element to i.

Find Walk up the tree from element e to get set name from root.

Union On union, add pointer from root of "smaller" tree to root of "larger" tree.

We can introduce an optimization: **path compression**. On `Find`, update pointer of every element in the path to point directly to the root. Note we could have done this during `Union`, but this is pre-emptive since we may not perform many `Find`s after. While this doubles the work of `Find`, we have the same $O(\cdot)$.
How do we define "smaller" and "larger"?
We can keep track of a tree's rank r where $r(\text{single node}) = 0$ and the union of two trees T_1 and T_2 with rank $r_1 \geq r_2$ would create a tree of rank $r = \max\{r_1, r_2 + 1\}$. Without path compression rank is equivalent to tree height.

Exercise 5.1. Show that an element of rank r has $\geq 2^r$ descendants ($r \geq 1$).
Proof by induction.

Analysis is a bit harder:

Theorem 5.1. (Tarjan 1975). The cost of m operations on the above union find data struture is $O(m \cdot \alpha(m, n))$ i.e. the amortized cost is $O(\alpha(m, n))$, where $\alpha(m, n)$ is the inverse Ackermann function, which is ≤ 5 for all practical purposes, so we have effectively $O(1)$ amortized cost.
This bound is tight (infinite examples where algorithm takes this runtime).

There is an easier bound to prove with path compression using a charging scheme with $O(m \log^* n)$. Note that $\log^* n$ is defined as

$$\log^* n = \min_i \{\log(\log(\ldots \log n)) \leq 1\}$$

where $\log^* n = i$ is the number of logs required such that the above expression is ≤ 1. How quickly does $\log^* n$ grow?
Note that the tower function is defined as $2 \uparrow n = 2^{2^{2^{\cdots}}}$. Thus we have $\log^*(2 \uparrow n) = n$, and note that

n	0	1	2	3	4	5
$2 \uparrow n$	1	2	4	16	65536	big number

So this bound is very good.
Note the cost of `Find(e)` is the distance from e to the root. We charge some of this cost to `Find` and some to the nodes along the path.

Claim. We claim $rank(e) < rank(parent(e))$.

Claim. The # of vertices of rank r is $\leq \frac{n}{2^r}$.

Proof. Using our previous claim that rank r has $\geq 2^r$ descendants and that vertices of rank r have disjoint descendants. □

Proof of runtime: for a given vertex of rank r, assign to group $\log^* r$. The number of groups is $\log^* n$. Note that group g contains ranks $2 \uparrow (g-1) + 1, 2 \uparrow (g-1) + 2, \ldots, 2 \uparrow g$ which has $\leq 2 \uparrow g$ ranks.
Then for `Find(e)`, for each vertex u on path from e to root, if u has parent and grandparent and $group(u) = group(parent(u))$, then charge 1 to u. Otherwise charge 1 to `Find(e)`.
Note that this covers the cost of the actual operation itself `Find(e)` (we've allocated enough charge).
The total times we charge to `Find(e)` (instead of a vertex u) is $\leq \log^* n + 1$ since group changes at most $\log^* n - 1$ times (and we add 2 for the root and child of root).
The total charge to each vertex u in group g: if u is charged then path compression will give it a new parent of higher rank than the old parent by claim 1.
So u in group g is charged

$$c(g) = (\#\text{ of ranks in group g}) - 1$$

times before it acquires a parent in a higher group and after then it is not charged, thus $c(g) \leq 2 \uparrow g$.

Total charge of all vertices in group g is $c(g) \cdot N(g)$ where $N(g)$ is the # of vertices in group g. Note that

$$\begin{aligned} N(g) &\leq \sum_{r=2\uparrow(g-1)+1}^{2\uparrow g} \frac{n}{2^r} \\ &\leq \frac{n}{2^{2\uparrow(g-1)+1}} \sum_{0}^{\infty} \frac{1}{2^i} \\ &= \frac{n}{2 \uparrow g} \end{aligned}$$

Thus $c(g) \cdot N(g) \leq n$.
Thus the charge to all vertices is $n \cdot \log^* n$ where n is the charge to 1 group and $\log^* n$ is the # of groups, thus we have the total charge for m `Find`s that are allocated to `Find`s and vertices

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

6 October 1, 2018

6.1 Geometric data

There are multiple problems associated with geometric data (i.e. multi-dimensional data in $\mathbb{R}^n$):

Range searching Given points in space, query a region R to find points in R.

In 2D, given a set of points pre-process them to handle range query for rectangle R.

Let us denote 3 measures:

1. P - preprocessing time S - space Q - query time ($\geq$ output size)

We may also consider a measure U for the update time to add/delete points.

Note that in $\mathbb{R}^1$, to find points in a given interval $[x_1, x_2]$, we can sort the points and find all points in between via binary searching for x_1, x_2. Thus we have

$$\begin{aligned} P &= O(n \log n) \\ S &= O(n) \\ Q &= O(\log n + t) \end{aligned}$$

is the output size.

To handle updates, we could instead use a balanced binary search tree, where P, S, Q remain the same. Update time $U = O(\log n)$.

In $\mathbb{R}^2$: in the static case (no updates) we have quad trees, kd trees, and range trees.

Quad trees Divide square into 4 subsquares recursively until each square has 0 or 1 points

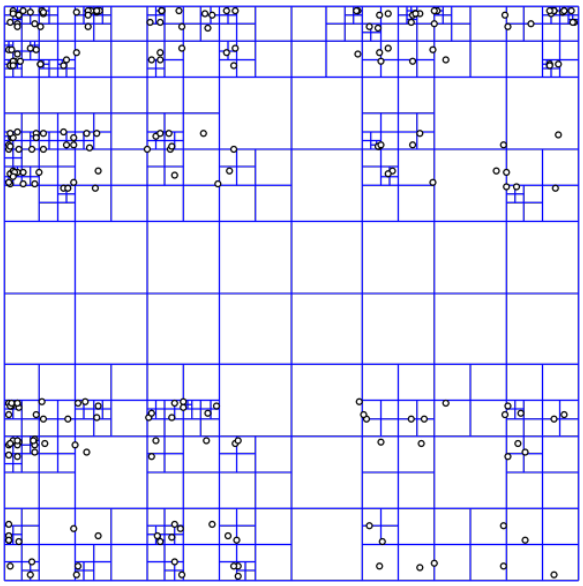


Figure 6.1: Example of quad tree partitioning space into quadrants/subsquares.

For our runtimes/space we have

$$P = O(n \log n)$$
$$S = O(n)$$
$$Q = \theta(\sqrt{n} + t)$$

(intuition for $\sqrt{n}$: it is equivalent to $2^{\log \sqrt{n}} = 2^{\log n/2}$ where we may need to check up to $\log n/2$ levels of nodes, and our branch factor is 2).

kd trees Alternately divide points in half vertically and horizontally.

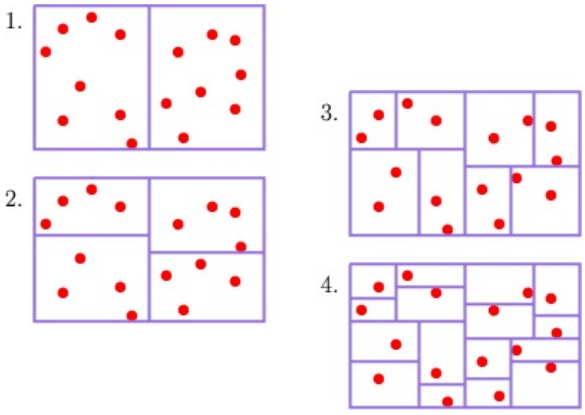


Figure 6.2: Example of kd tree dividing in half the points vertically first, then horizontally, then vertically, and finally horizontally.

We can first sort all points (each by both dimensions) and find our median/mid-point dividing lines for

each iteration. We then construct an binary search tree of our dividing lines. Thus we have

$$P = O(n \log n)$$
$$S = O(n)$$
$$Q = \theta(\sqrt{n} + t)$$

To do the actual query, we check if our rectangle endpoints if they belong in either side of each split-point. We then recurse into the side(s) that our rectangle is contained in.

Note that our query time with $\sqrt{n}$ is much worse than $\log n$.

Range trees Improve Q at the expense of S.

We construct a balanced BST on x-coordinates where the **leaves** are the points sorted by x-coordinates.

For a given internal node v, its descendants $D(v)$ is associated with a **slab**: that is we store at v a list $A(v)$ of points in $D(v)$ sorted by their y-coordinates.

Note an upper bound on space is $O(n^2)$: each internal node may store up to n nodes and we have $\frac{n}{2}$ internal nodes. However, a tighter bound is $O(n \log n)$ where we notice each leaf node can be a part of at most $O(\log n)$ ancestor nodes.

For processing: we first sort by x-coordinates. We maintain a y-coordinate sorted list as well. For each internal node we can simply extract the corresponding nodes in the slab from the sorted y-coordinate list. Thus $P = O(n \log n)$.

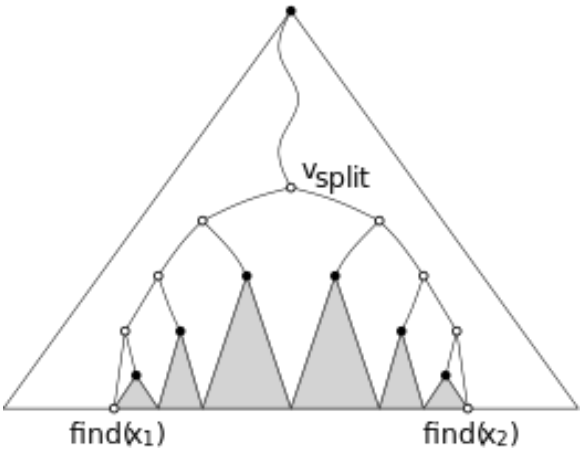


Figure 6.3: Example of a range tree where we are querying for points in between x_1 and x_2.

For searching: we search for x_1, x_2. We want the subsets of leaves between, which we can then filter by y-coordinate.

To find the leaves in between, we look at the internal nodes z, which are the **right children** of nodes on search path to x_1 and the **left children** of nodes on search path to x_2 (after paths split). Thus z corresponds to slabs with union $[x_1, x_2]$.

Remark 6.1. Why couldn't we just use $A(v_{split})$, where v_{split} is the common ancestor of x_1, x_2? Note that while v_{split} is the common ancestor, x_1 may actually be in the right children of a node in the left path further down (and similarly x_2 in the left children), so we don't want to include nodes outside this range (see figure).

For each slab z in $[x_1, x_2]$, we perform binary search on $A(z)$ to get points between $[y_1, y_2]$. Note that we have query time $O(\log n + t)$ per slab thus we have total query time $Q = O(\log^2 n + t)$ where t is the

output size (we have $O(\log n)$ slabs and since slabs are disjoint and each output is counted only once). How can we reduce to $Q = O(\log n + t)$? We can save work on repeated searches for y_1, y_2 using **fractional cascading**. For a given node z and child node w (of which we have sorted lists $A(z)$ and $A(w)$ by y-coordinate), we keep pointers from each element in $A(z)$ to the same (or next higher) element in $A(w)$. We perform binary search on the parent $A(z)$, then when we search in $A(w)$ we continue searching from the pointers we left off at. We thus only binary search on at most n elements so we have $O(\log n)$.

Point location Query a point p to find which region contains p.

The plane is generally divided into disjoint regions and we want to query for a point p and its region.

Remark 6.2. A special case of these regions are the regions generated by the "closest to center": given several points of interest (e.g. Tim Horton's locations) and a query point p, which is the closest point of interest?

We can split the plane at the midpoint between two or more arbitrary POI which ends up creating a plane of disjoint regions: this is the **Voronoi diagram**.

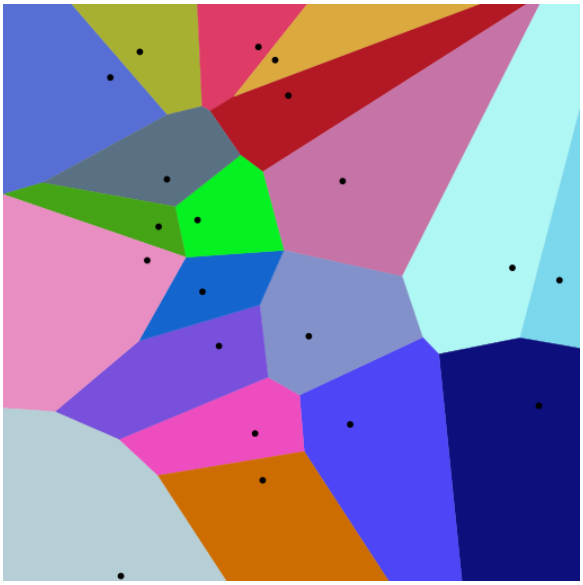


Figure 6.4: Voronoi diagram where each shaded region is mapped to a POI.

In $\mathbb{R}^1$, we keep track of the endpoints of the regions and binary search on our query point.

In $\mathbb{R}^2$, we divide our disjoint regions into slabs with vertical lines at every vertex of the regions. Let n denote the number of resulting slabs.

To search for a query point, we find the slab containing the x-coordinate $O(\log n)$. We then do a binary search inside the slab for the region containing the y-coordinate (this works for arbitrary line segments for a fixed x for our point since we can solve for $y = mx + b$), which is also $O(\log n)$. Thus we have in total $Q = O(\log n)$.

Exercise 6.1. Find a solution where we have space $S = O(n^2)$.

Hint: we have up to $n + 1$ vertical slabs. In each slab, we have up to $n + 1$ dividing line segments, thus we require $O(n^2)$ space.

Improvement to space: changes from one slab to the next are few (at each boundary vertex, some line segments end and some begin per each slab, while some remain the same).

We can thus think of our slabs as a **sequence of binary search trees** that have few changes in between them.

We can initially construct search tree containing line segments in the leftmost slab, then sweep left to right to introduce/take away line segments in each subsequent slab. A *persistent* tree will let us use the same sub-tree from previous steps instead of constructing an entirely new tree. We thus end up storing at most as many elements as there are line segments or $O(n)$ space.

Remark 6.3. Data structures that can be updated and queried in the past are called **persistent data strucutres** (e.g. querying for Facebook friends in the past).

Partial persistence only allows updates in the present while **full persistence** allows updates to the past.

We require partial persistence for range trees.

With the improvement, we end up with

$$P = O(n \log n)$$
$$S = O(n)$$
$$Q = O(\log n)$$

7 October 3, 2018

7.1 Randomized algorithms

Randomized alorithms are algorithms that use *random numbers*. The output and/or run time depend on random numbers. We must do **expected case analysis** for analyzing run time.
Advantages of randomized algorithms:

1. practical: faster/simpler algorithms in general
2. theoretical: can we even prove randomness helps? e.g. can randomness give poly-time for NP-hard problems? The evidence is slight.

Examples of randomized algorithms include hashing, quicksort, and quickselect.

Example 7.1 (Quicksort)**.** Let

```
1   Input: S = {s_1, ..., s_n}
2
3   if n = 0,1 return S
4   else
5     i = random[1..n]              // s_i is our pivot
6     L = {s_j : s_j < s_i}         // size l
7     M = {s_j : s_j = s_i}         // size m
8     B = {s_j : s_j > s_i}
9
10    return (Quicksort(L), M, Quicksort(B))
```

The worst case run time is $O(n^2)$ i.e. when $l = n - 1$ (and $n_i - 1$ on subsequent iterations i).
Intuition: we "expect" s_i to be in the middle hence

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

thus we have expected case $O(n \log n)$.

Remark 7.1. There is a slight difference between *average* and *expected* case:

average case analysis No random numbers, assume all inputs equally likely

expected case analysis Algorithms use random numbers, NO assumption on input

More formally, we have the model that includes the random number generation

$$x = rand[1, \ldots, n]$$
$$x = rand[0, 1]$$

which both have $O(1)$ cost each.
Some terminology:

sample space all possible "runs" of algorithms for fixed input

random variable maps sample space to run time (integer)

expected value Expectation of random variable X where

$$E[X] = \sum_x xP(X = x)$$

Example 7.2. Biased coin where $P(H) = \frac{1}{3}$. The expected number of coin tosses to get a head is:

$$\sum iP(\text{i tosses}) = 1 \cdot \frac{1}{3} + 2 \cdot \frac{2}{3}\frac{1}{3} + 3 \cdot \left(\frac{2}{3}\right)^2\frac{1}{3} + \ldots$$

Properties of expected values include:

$E(X+Y) = E(X) + E(Y)$	linearity
$E(cX) = cE(X)$	constant multiplication
$E(XY) = E(X)E(Y)$	X and Y are independent $P(X = x, Y = y) = P(X = x)P(Y = y)$
$E(X) < E(Y)$	if $X < Y$
$\max\{E(X), E(Y)\} \leq E(\max\{X, Y\})$	

The run time depends on the input *and* random numbers. For a randomized algorithm, our run time can be represented as $T(I, R)$ where I is a fixed input and R is a sequence of results of $rand[\ldots]$. We eventually want T as a function of the input size n.
Thus the **worse case** run time is the *max* over all inputs I where $|I| = n$, and the **expected case** run time is the *average* over all random operations R.
The expected case is formally $E(T(I, R)) = \sum_R P(R)T(I, R)$ and the worst case can be expressed as

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

Example 7.3. We can perform analysis on quicksort without using recurrences using expected case analysis.
We want to find $E(X)$ where X is the # of comparisons and $X(u, v)$ is the number of comparison between u and v.

Thus

$$\begin{aligned} E(X) &= E(\sum_{u,v \in S} X(u,v)) \\ &= \sum_{u,v \in S} E(X(u,v)) \end{aligned}$$

Note that for any pair u, v, we compare them either 0 or 1 times since given parts L, M, B as above:

- u, v are initially in the same part
- they are in different parts after partitioning
- they are never compared after they go in different parts

For $E(X(u,v))$ we can look at the step where u, v are separated. It is obvious that we only compare u, v if the pivot is either u or v. WLOG if $u < v$ and we sort all our number such that u has rank r and v has rank $r + k$ (rank is the order of an element when sorted):

$$\begin{aligned} E(X(u,v)) &= 1 \cdot P(\text{compare u,v}) + 0 \cdot P(\text{no comparison}) \\ &= \frac{2}{k+1} \end{aligned}$$

where $k + 1$ is the number of choices we have when u is separated from v because we chose something between u, v, inclusively. Thus we have

$$\begin{aligned} E(X) &= \sum_{r=1}^{n-1} \sum_{k=1}^{n-r} \frac{2}{k+1} \\ &\leq 2 \sum_{r=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \\ &\leq 2 \sum_{r=1}^{n} O(\log n) && \text{harmonic series} \\ &= O(n \log n) \end{aligned}$$

7.2 Selection and Quickselect

The selection problem is as follows: given $S = \{s_1, \ldots, s_n\}$ numbers and $k \in [1, \ldots, n]$, return s_i of rank k i.e. $k = 1$ (min), $k = n$ (max), $k = \lfloor \frac{n}{2} \rfloor$ (median).
The Quickselect algorithm is as follows

```
1  if n <= constant
2    sort and return kth item
3  else
4    i = rand[1, ..., n]            // s_i pivot
5    partition S into
6      L: smaller than s_i          // size l
7      M: equal to s_i              // size m
8      B: bigger than s_i           // size b
9    recurse on appropriate set
```

The worst case is $O(n^2)$, but expected case is $O(n)$ using recurrence relations.
Open question: can we do expected case analysis for quickselect like we did for quicksort?
History for selection problem:

1960: Hoare Quickselect has $E(\#\text{ of comparison}) = 3n + o(n)$.

1973: Blum A *non-randomized* algorithm with expected case $O(n)$ where $E(\#\text{ of comparison}) = 5.43n + o(n)$.

1975: Floyd-Rivest Another randomized algorithm with $E(\#\text{ of comparison}) = 1.5n + o(n)$.

1985 Proved lower bound is $2n$ for deterministic/non-randomized algorithm.

1989: Munro & Cunto Proved *any* randomized algorithm has lower bound $E(\#\text{ of comparison}) \geq 1.5n + o(n)$, so Rivest's algorithm is tight.

1999: current determinisitic upper bound Deterministic algorithm with $2.95n$ comparisons

2001: current deterministic lower bound Proved lower bound is $(2 + \epsilon)n$ where $\epsilon = 2^{-80}$.

Conclusion: randomness provably helps.

7.3 Lower bound on median selection

Theorem 7.1. Proposed by Blum et al. in 1975, finding the median ($k = \lfloor \frac{n}{2} \rfloor$) requires $\geq 1.5n$ comparisons in worst case.

Proof. The proof uses an **adversarial argument**: i.e. what is the worst possible case for our algorithm?
Let m be the median, L be elements $< m$ and H be elements $> m$, each with $\frac{n-1}{2}$ elements.

Claim. We claim the number of comparisons between elements within L (#LL) and elements within H (#HH) is $\geq n - 1$ i.e. $\#LL + \#HH \geq n - 1$.
Note each element in L must "lose" a comparison (i.e. $<$) to an element in L or m itself.
Similarly each element in H must "win" a comparison (i.e. $>$) to an element in H or m itself.
This forms a tree of comparisons (each edge is a comparison) with at least $n - 1$ edges.

Claim. The worst case number of comparisons between elements in L and in H (#LH) is $\geq \frac{n-1}{2}$.
As the algorithm is computing comparisons (i.e. whether $s_i < s_j$), the adversary would construe/manufacture the worst possible case for these comparisons. The adversary maliciously tries to maximize the # of comparisons required by putting elements in L and H such that the number of #LH comparisons is maximized.
The adversary algorithm is as follows

```
1  on comparison x,y:
2    if x and y are set (in L/H)
3      continue
4    if x is set, y not set
5      if x in L, put y in H
6      if x in H, put y in L
7    if x,y are unset
8      put one in L, one in H
```

But we still require $\frac{n-1}{2}$ elements in each of L and H, thus the adversary stops when either $|L|$ or $|H|$ is $\geq \frac{n-1}{2}$.
Thus the adversary forces at least $\frac{n-1}{2}$ comparisons.

□

8 October 12, 2018

8.1 Las Vegas vs Monte Carlo

There are a few distinctions between randomized algorithms:

Las Vegas algorithm Always produces the correct output, regardless of random numbers generated. Expected polynomial runtime.

Quicksort is one such example.

Monte Carlo algorithm Produces the correct output with high probability (that can be bounded by number of trials). The runtime should always be polynomial.

How are these related?

Las Vegas to Monte Carlo Stopping algorithm after some time and outputting junk.

Monte Carlo to Las Vegas Given a correctness test (with good run time), we test the output of the Monte Carlo and if incorrect, repeat until we get the correct answer.

8.2 Primality test

Given an odd number n, is n composite (i.e. not prime)?
We phrase it this way so we have a problem in NP - verify YES answers with "proof" i.e. the factors of the composite number.
Note that if the input is n, the input size is $\log n$ (# of bits), so trial division up to $\sqrt{n}$ takes $O(\sqrt{n})$ which is **not** polynomial time wrt to the input size.
There does exists a polynomial time (non-randomized) algorithm to test primality (Agrawal, Kayal, and Saxena, 2002: AKS primality test).

Theorem 8.1 (Fermat's Little Theorem)**.** If p is prime then $a^{p-1} \equiv 1 \mod p$ for all $0 < a < p$.

Remember that the contrapositive states that $A \Rightarrow B$ is equivalent to $\neg B \Rightarrow \neg A$, thus FLT restated says that if there exists $0 < a < n$ and $a^{n-1} \not\equiv 1$ then n is composite. We call such an a a **Fermat witness** to n's compositeness.
Idea: to test if n is composite:

- Generate random a in $[1, \ldots, n-1]$
- Test if a is a Fermat witness (this can be done efficiently)
- If it is, output YES n is a composite
- otherwise, MAYBE n is prime

For this to work (efficiently), we require that if n is composite then there are many Fermat witnesses.
However, there are composite numbers with *no Fermat witnesses*: the **Carmichael numbers** e.g. 561, 1105, 1729, etc.
We thus need **strong witnesses**.

Definition 8.1 (Strong witness)**.** Let $n - 1 = 2^t \cdot u$ (n is even) where u is odd. Then $a \in [1, \ldots, n-1]$ is a **strong witness** if for some $0 \leq i < t$ we have $k = 2^i \cdot u$ and

$$a^k \not\equiv +1, -1 \mod n$$
$$a^{2k} \equiv 1 \mod n$$

That is a^k is a non-trivial square root of $1 \mod n$. Note that $-1 \equiv n - 1 \mod n$.

Theorem 8.2. If n is prime then there are no strong witnesses.
Idea: integers mod prime p form a finite field in which 1 has exactly two square roots, 1 and -1.

Theorem 8.3. If n is composite then there are $\geq \frac{n-1}{2}$ strong witnesses.
That is the probability that $a \in [1, \ldots, n-1]$ is a strong witness is $\geq \frac{1}{2}$.

Refer to CLRS for proofs of the two theorems above.
We thus define the pseudocode for our procedure `witness(a,n)` that tests if a is a strong witness of n

```
1   witness(a,n):
2     compute t,u where n - 1 = 2^t u, u odd
3     x_0 = a^u mod n
4
5     for i = 1...t
6       x_i = (x_{i-1})^2 mod n
7       if x_i = 1 and x_i-1 != 1 and x_{i-1} != n-1
8         return TRUE                                  // a is a strong witness
9     if x_t != 1 return TRUE                          // a is a Fermat witness
10
11    return FALSE
```

The runtime of `witness` is polynomial in $\log n$ since $t \leq \log n$ so the loop is executed $\log n$ times and squaring in mod n takes $\log n$ time.
Thus we have the **Miller-Rabin algorithm** which applies `witness` in a Monte Carlo approach

```
1   repeat s times
2     x = rand[1, ..., n-1]
3     if witness(x,n) then return YES: n is composite
4
5   return NO n is not composite                       // this is really a "MAYBE"
```

The runtime is therefore $O(s \log^k n)$ ($k \approx 2$) which is polynomial in $\log n$.
If n is prime then the algorithm is *always correct*.
If n is composite then

$$P(\text{alg outputs NO}) \leq P(\bigcap_{j=1}^{s} \text{at trial j x is not a strong witness}) \leq \frac{1}{2^s}$$

This is a Monte Carlo algorithm with a **one-sided error** where:

- If the algorithm outputs YES (n is composite) it is correct
- If the algorithm outputs NO (n is prime) the probability of error is $\leq \frac{1}{2^s}$

8.3 Complexity classes

Recall we have the P and NP classes

P decision problems solvable in polynomial time

NP non-deterministic polynomial time: decision problems where YES answers can be verified in polynomial time given a certificate or proof

co-NP complement of problem is in NP i.e. decision problems where NO answers can be verified in polynomial time given a certificate

Some open questions include $NP = co - NP$, $P = NP$ and $P = NP \cap co - NP$.

Definition 8.2 (RP complexity class)**.** The RP or **randomized polynomial time class for one-sided Monte Carlo algorithms** are decision problems that have a randomized algorithm A running in worst-case polynomial time such that for any input x

$$x \text{ IS YES} \Rightarrow P(A(x) \text{ outputs YES}) \geq \frac{1}{2}$$
$$x \text{ IS NO} \Rightarrow P(A(x) \text{ outputs YES}) = 0$$

Thus YES is always correct and NO is wrong with probability $\leq \frac{1}{2}$.

Similarly, co-RP ("complement") are decision problems with randomized algorithms where NO is always correct and YES is wrong with probability $\leq \frac{1}{2}$.

Definition 8.3 (ZPP complexity class)**.** The ZPP or **zero error probability polynomial time** class are decision problems that have Las Vegas algorithms with expected polynomial run time.

Lemma 8.1. We claim $P \subseteq ZPP \subseteq RP \subseteq NP$.

Proof. $P \subseteq ZPP$ A polynomial time algorithm is a Las Vegas algorithm with no use of randomness.

$ZPP \subseteq RP$ We require

Theorem 8.4 (Markov's inequality)**.** If X is a random variable $X \geq 0$ with $E(X) = \mu$ then $P(X \geq c\mu) \leq \frac{1}{c}$.

Proof. Note that

$$\begin{aligned} \mu = E(X) &\geq \sum_{x \geq c\mu} xP(X = x) \\ &\geq c\mu \sum_{x \geq c\mu} P(X = x) \\ &= c\mu P(X \geq c\mu) \end{aligned}$$

Thus we have $\frac{1}{c} \geq P(X \geq c\mu)$. □

Suppose we had a ZPP decision algorithm A with expected run time $T(n)$ bounded by a polynomial in n. Define a Monte-Carlo algorithm A' as follows:

- on input x of length n, run $A(x)$ for time $2T(n)$
- if $A(x)$ produces YES/NO answer in that time, output it
- else output NO

Then A' runs in polynomial time (always), and

- if A' outputs YES this is correct
- if A' outputs no then

$$\begin{aligned} P(\text{error}) &\leq P(A(x) \text{ takes more than } 2T(n) \text{ time}) \\ &\leq \frac{1}{2} \qquad \text{Markov's inequality} \end{aligned}$$

$RP \subseteq NP$ Suppose we have a decision problem and an RP algorithm A for it.

An execution of A depends on the input x and random numbers y, which we can denote as $A(x, y)$ where $A(x, y)$ runs in time polynomial in $|x|$.

From the definition of RP, if x is a YES input then there is a y with $|y|$ bounded by polynomial in $|x|$ such that $A(x, y)$ outputs YES (in fact many y's).

If x is a NO input then there is no y such that $A(x, y)$ outputs YES.

Thus y acts as the certificate to verify a YES input (by running $A(x, y)$) in polynomial time.

□

It remains an open problem whether the containments are proper.

Lemma 8.2. We claim $ZPP = RP \cap co - RP$.

Proof. From above, $ZPP \subseteq RP$ and similarly $ZPP \subseteq co - RP$, thus $ZPP \subseteq RP \cap co - RP$.
It remains to prove that $RP \cap co - RP \subseteq ZPP$ (assignment 4). □

9 October 15, 2018

9.1 More Monte Carlo primality

Recall from last day: a randomized Monte Carlo algorithm to test if n is prime:

- Polynomial time
- One-sided error: if alg. claims n is composite it must be correct. If alg. claims n is prime, prob(error) $\leq \frac{1}{2}$ (larger fraction would be okay).
- Can improve with repeated trials (such that error is then bounded by $\frac{1}{2^n}$ for n trials).

Note: there is a non-randomized polynomial time algorithm to test primality (from last day).
Follow-up:

Question how do we generate a large random t-bit prime?

Answer generate a random t-bit number and test if it's prime. If not, generate a new number.

For deriving the expected runtime, we need to know the distribution of primes.

Application: RSA cryptosystem.

- Depends on hardness of factoring $n = pq$ where p, q are primes
- Factoring: given a number n, find prime factorization
- No known polynomial time non-randomized nor randomized algorithm, not known to be NP-hard
- Decision version: given n, m does n have a (prime) factor $\geq m$?

  Not known to be in NP-complete, but it is in NP.

9.2 Fingerprinting

Example 9.1. Suppose we wanted to test equality of strings and it is too expensive to send/compare the entire string (e.g. two databases in different locations).

Solution. Send/compare a smaller "fingerprint".

Let x be an n-bit binary number ($< 2^n$).
Compute $H_p(x) \equiv x \mod p$ where p is a prime chosen at random in $1, \ldots, M$ (we choose M), thus $H_p(x)$ has size $\log M$.
Note that if $x = y$ for some y, then $H_p(x) = H_p(y)$ must be true.
But we can have $x \neq y$ but $H_p(x) = H_p(y)$ which happens if $|x - y|$ is divisible by p, which is our "failure". What is error bound on prob(failure)?
Need two results from number theory:

1. Prime Number Theorem: Let $\pi(N)$ denote the # of primes $< N$.
   Then $\pi(N) \sim \frac{N}{\ln N}$.
2. # of primes dividing $A < 2^n$ is $\pi(n)$.

Thus the error rate is

$$\begin{aligned} prob(failure) &\leq \frac{\text{\# of primes } p < M, \text{ p divides } |x - y < 2^n|}{\text{\# of primes } < M} \\ &\sim \frac{\pi(n)}{\pi(M)} \end{aligned}$$

If we choose $M = n^2$, then

$$prob(failure) = \frac{n}{\ln n} \cdot \frac{\ln n^2}{n^2} = \frac{2}{n}$$

So by comparing fingerprints of length $O(\log n)$ (since we chose $M = n^2$, our resulting fingerprints $< n^2$ thus will have $O(\log n^2) = O(\log n)$ length), we get a good randomized test with prob(error) $= \frac{2}{n}$ (one-sided error).
Note by choosing p at random every time, we generate good behaviour for *all* x, y (as opposed to fixing p and have it fail with certainty for some x, y).

9.3 Verifying polynomial identities

Example 9.2. The **Vandermonde matrix** is given by

$$M = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \ldots & x_n^{n-1} \end{bmatrix}$$

Theorem 9.1.

$$\det(M) = \prod_{i,j \quad j<i} (x_i - x_j)$$

We could verify this theorem by plugging in arbitrary values for $x_1, \ldots, x_n$ and computing $\det(M)$ as fast as matrix multiplication (obvious runtime is $O(n^3)$ but in reality $O(n^\omega)$ where ω is the matrix multiplication constant, currently 2.373, a slight improvement to the former best **Coppersmith-Winograd** algorithm).

Can compute $\det(M) \mod p$ for some prime p to do comparison(?)

In the above Vandermonde example, there was a theorem, but in general this arbitrary testing has other applications such as in *symbolic math* and *automatic theorem proving.*
General problem: given multivariate polynomial, test if $f(x_1, \ldots, x_n) \equiv 0$ (identically 0 i.e. all coefficients of terms are 0; we could multiply out the polynomial and check the coefficients but this is exponential time).

Example 9.3. The polynomial $x_1x_2^3 + x_3^2 + x_1x_2$ is clearly not identically 0.

Definition 9.1. The **degree of a term** $x_1^{i_1}x_2^{i_2}\ldots x_n^{i_n}$ is $\sum_{j=1}^{n} i_j$.
The **degree of a polynomial** is the max over all its terms.

We can thus use a randomized algorithm that plug in random values $x_1, \ldots, x_n$ to test if it's 0. To calculate the error probability, we will require the following theorem:

Theorem 9.2 (Schwartz-Zippel)**.** Let $f(x_1, \ldots, x_n)$ be a multi-variate polynomial of total degree d, $f \not\equiv 0$ (f not identically 0).
If we choose values $a_1, \ldots, a_n$ for $x_1, \ldots, x_n$ independently and uniformly from finite set $S \subseteq \mathbb{F}$ then

$$prob(f(a_1, \ldots, a_n) = 0) \leq \frac{d}{|S|}$$

If we choose $S = \{\pm 1, \pm 2, \ldots \pm d\}$ then our probability of error is $\leq \frac{1}{2}$.

Proof. By induction on # of variables n.
Base case when $n = 1$: note that the # of roots of $f(x) \leq d$ where d is our degree, thus we have probability of $\frac{d}{|S|}$ of choosing one of those d roots.
Induction case: we can write

$$f(x_1, \ldots, x_n) = \sum_{i=0}^{d} x_1^i f_i(x_2, \ldots, x_n)$$

where f_i has at most degree $d - i$ (we rewrite f as a sum of terms with x_1 for all possible powers of x_1).
Since $f \not\equiv 0$, at least one term for $i > 0$ is $\not\equiv 0$. Take $k = \max i$ over these i's. Then

$$f_k(x_2, \ldots, x_n) \not\equiv 0$$

where f has total degree $\leq d - k$. By induction

$$P(f_k(a_2, \ldots, a_n) = 0) \leq \frac{d-k}{|S|}$$

If $f(a_2, \ldots, a_n) \neq 0$ then $f(x_1, a_2, \ldots, a_n)$ is a degree k polynomial in terms of x_1 (we plug in constants for all other $x_2, \ldots, x_n$).
Thus

$$P(f(a_1, \ldots, a_n) = 0 \mid f_k(a_2, \ldots, a_n) \neq 0) \leq \frac{k}{|S|}$$

from our base case when $d = 1$.
Finally, we use the identity

$$P(A) \leq P(B) + P(A \mid B^c)$$

thus we have

$$\begin{aligned} P(f(a_1,\ldots,a_n)=0) &\leq P(f_k(a_2,\ldots,a_n)=0) + P(f(a_1,\ldots,a_n)=0 \mid f_k(a_2,\ldots,a_n)\neq 0) \\ &\leq \frac{d-k}{|S|} + \frac{k}{|S|} \\ &\leq \frac{d}{|S|} \end{aligned}$$

□

We thus have a Monte Carlo algorithm to test polynomial identities (i.e. identically 0 polynomials) where we have a one-sided error: if algorithm claims NO (not identically 0) then it is correct. If it claims YES (identically zero) then prob(error) can be arbitrarily small (depending on the # of trials and our set S).
Open problem: Testing polynomial identities in polynomial time deterministically (i.e. no randomness).
Applications:

Verifying matrix multiplication We are given matrices A, B and we compute C as the product of the two matrices (e.g. C was computed with fast matrix mulplication that is complicated and prone to implementation error).

We'd like to verify if $C = AB$.

Suppose $A, B \in \mathbb{R}^{n\times n}$ for simplicity. Let $x = (x_1,\ldots,x_n)$ thus we can verify

$$A(Bx) = Cx$$

where we end up with n multivariate polynomials with (up to) n variables of total degree 1. For example

$$Cx = \begin{bmatrix} 5 & 15 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5x_1 + 15x_2 & 2x_1 - x_2 \end{bmatrix}$$

Since $d = 1$, we can pick a small $S = \{0, 1\}$, thus the probability of error is $\leq \frac{d}{|S|} = \frac{1}{2}$.

10 October 17, 2018

10.1 Linear programming

Linear programming is when we have an optimization problem with linear inequalities. Given variables $x_1,\ldots,x_d$ in d-dimensions we may have

$$\max c_1x_1 + c_2x_2 + \ldots + c_dx_d$$

subject to

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \ldots + a_{1d}x_d \leq b_1 \\ &\vdots \\ &a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nd}x_d \leq b_n \end{aligned}$$

or simply

$$\max c^t x$$

subject to

$$Ax \leq b$$

where $c \in \mathbb{R}^{d\times 1}$, $x \in \mathbb{R}^{d\times 1}$, $A \in \mathbb{R}^{n\times d}$ and $b \in \mathbb{R}^{n\times 1}$.
Each constraint $a_1x_1 + a_2x_2 \leq b$ forms a half-space. The intersection of all half-spaces is our feasible region (i.e. region that satifies our constraints which is a convex polyhedron or a convex hull).
Note that we have a few cases:

- Optimum occurs at a **vertex**: the intersection of two (or more) lines of constraints.
- We can have multiple optima.
- We can have an unbounded solution.
- The problem may also be infeasible.

Lemma 10.1. In any dimension d if the feasible region is non-empty and bounded then there is an optimum solution at a vertex.

This lemma implies a finite algorithm where we test all vertices see which gives the optimal value (i.e. maximum or minimum value): that is we test all sets of $\binom{n}{d}$ constraints where we set them to equality to find the vertices.
For each vertex, we test if x is feasible. If it is then we find c^tx and compare all such x.
This takes $O(n^d)$ time. Linear programming (e.g. the Simplex algorithm) attempts to make this more efficient.
Some applications of linear programming include:

- Planning diets: where x_i is the amount and c_i is the cost of food type i. We also need to meet minimum dietary requirements such as
$$a_{j1}x_1 + a_{j2}x_2 + \ldots + a_{jd}x_d \geq b_j$$
for various nutrients $j = 1, \ldots, n$.

The Simplex method (Dantzig 1940s) is a more efficient algorithm for solving linear programs (which subsequently spurred the development of computers).
At a high-level It starts at some vertex that is the intersection of two constraints. It chooses and removes one of the equality constraints and adds one new equality constraint, which effectively causes us to "walk along" a constraint edge to find our next vertex.
It however requires a rule (our "pivot rules") for which constraint to remove and which constraint to add.
Simplex is very good in practice but we do not know a pivot rule that guarantees polynomial time.
The run time is related to the diameter (minimum # of edges between two vertices) of our feasible convex polyhedron: intuitively we need to walk along an order of our diameter to reach our optimum solution.
The **Hirsch conjecture** states that the diameter is $\leq n - d$ where n is the number of constraints and d is the number of dimensions. This conjecture was however disproved in 2012.
There does exists polynomial time algorithm for linear programming:

Khachiyan 1980 ellipsoid method

Karmarkan 1984 interior point method

At a high level the algorithms operate on the bit representations of numbers.
Thus there remains the open problem if there exists a polynomial time algorithm that uses only arithmetic steps.
In the 1970s and 1980s, linear programming was applied to many small dimensional problems. In 2D, we can apply linear programming to find the best fit line for some points. In 3D, we can determine whether a cast can be removed from a mold.
An algorithm by Megiddo in 1983 takes $O(n)$ for fixed d: it is actually $O(2^{2^d} \cdot n)$.
We will look at **Seidel's randomized incremental linear programming algorithm**: at a high level, we add half-plane constraints one-by-one and update our (current) optimum solution v: how exactly do we update our v?

Note if we add our half-planes in random order then the expected runtime is $O(n)$ (# of constraints).
To update, we add a half-plane h_i. There are multiple cases:

1. If $v \in h_i$ (v is still in the half-space of our half-plane), then no update is necessary.
2. If $v \notin h_i$, then we claim the optimum solution lies on line l_i of h_i.

   We essentially have a 1-D linear programming problem where we have for example constraints $x_1 \leq 2$, $x_1 \leq 5$ and $1 \leq x_1$ and we find the optimal x that maximizes $c^T x$ of this 1D problem.

The algorithm $LP_2(H)$ where $H = \{h_1, \ldots, h_n\}$ or the set of half-planes is thus as follows:

```
1  LP_2(H):
2    Let h_1 ... h_n be in random order
3    v = point at infinity                          // init to unbounded optimum
4    for i = 1 ... n
5      if v not in h_i:
6        v = LP_1({h_1, ..., h_i-1} intersect l_i})  // l_i is the line at h_i
```

Note the 1D linear programming with i constraints runs in $O(i)$, thus our worst case is $\sum_{i=1}^{n} O(i) = O(n^2)$.
To find our expected run time: the idea is to note that case 1 (where $v \in h_i$) happens often.
Consider the situation after adding h_i: did we encounter case 1 or case 2? (this technique of analysis is called **backwards analysis**).
After adding h_i we have i half-planes. Note that the probability that the h_i we just added is any particular h is equally likely thus $P(h_i = h) = \frac{1}{i}$.
Case 2 where $v' \notin h_i$ happens if we had to update our v because our h_i made our previous v' infeasible. The probability that case 2 happens is choosing one of the two h's that form our new vertex v i.e. v is determined by two lines l, l' where $P(h_i = l) = P(h_i = l') = \frac{1}{i}$. Thus $P(\text{case two}) \leq \frac{2}{i}$.
The expected work across all our LP_1 is

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

where $\frac{2}{i}$ is the probability of having to invoke LP_1 and $O(i)$ is the runtime of our LP_1 for i constraints.
In higher dimensions, $\frac{2}{i}$ becomes $\frac{d}{i}$ since d constraints determine a vertex v.
Thus we have the recurrence

$$T_d(n) = T_d(n-1) + \frac{d}{n}(T_{d-1}(n))^n$$

which solves to $T_d(n) = O(d!n)$.

11 October 22, 2018

11.1 SAT

Satisfiability or SAT: Given a CNF (conjunctive normal form) formula with n variables, m clauses can we assign true/false to variables to satisfy the formula.

3-SAT all clauses have 3 literals (variable or negation). This is NP-complete.

2-SAT 2 literals per clause. There exist polynomial time algorithms.

Some applications of SAT:

- AI

- All problems with quantified Boolean formulas e.g. $\forall x \exists y \forall z F(x, \ldots)$.

  SAT is the case with one existential quantifier for all variables.

Some techniques for solving SAT problems:

- heuristics or "resolution"
- brute force worst case run time with $O(1.5^n)$ (best algorithm known thus far); better than the obvious $O(2^n)$ brute force

Question 11.1. Can we get randomized polynomial time for 3-SAT?
This would give randomized polynomial time for all NP-complete problems.
We will thus show a randomized algorithm that can beat the best known deterministic (non-randomized) algorithm.

A randomized SAT algorithm (Papadimitriou 1991) gives us a one-sided error approach:

```
1   input: Boolean formula E in CNF
2   idea: local improvement (hill climbing)
3
4   start with any T/F assignment A
5   repeat t times:                    // t to be chosen
6     if A satisfies E:
7       return YES
8     pick an unsatisfied clause C
9     randomly pick a literal a in C
10    flip a's value
11  return NO
```

Note that YES is always correct and NO might be an error: a satisfiable formula but we failed to find an assignment A that works. What is this error rate?
Let A^* be a valid (i.e. makes the formula true) assignment that satisfies E. Let i denote the # of variables with the same value in current A and A^*. If i reaches n then $A = A^*$ and algorithm outputs YES.
How does i change? In each iteration, $i = i + 1$ or $i = i - 1$ (it either becomes the right assignment or the wrong assignment).
We analyze the probability of either case for 2-SAT and 3-SAT separately. We require the notion of **random walks**. As a Markov chain: suppose we are currently at state i and we can move to either $i - 1$ and $i + 1$. The probability of moving to $i + 1$ and $i - 1$ is $\frac{1}{2}$ each, and at state 0 the probability of moving to state 1 is 1 (wall at 0).
What is the expected # of steps to n? Let t_i denote the expected number of steps to get from i to n. We can derive recurrence relations in terms of t_i by conditioning on the first step (first step analysis):

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

Note that from the recurrence we have

$$\begin{aligned} t_i - t_{i+1} &= 2 + t_{i-1} - t_i \\ \Rightarrow d_i &= 2 + d_{i-1} \end{aligned}$$

where $d_i = t_i - t_{i+1}$, and thus $d_0 = t_0 - t_1 = 1$. Solving the recurrent we get $d_i = 1 + 2i$ so plugging this back into $t_i = d_i + t_{i+1}$ and $t_n = 0$ we get

$$\begin{aligned} t_i &= \sum_{j=i}^{n-1} d_j \\ &= \sum_{j=i}^{n-1} (1 + 2j) \\ &= (n - i) + n(n - 1) - i(i - 1) \\ &= n^2 - i^2 \end{aligned}$$

where max of t_i is n^2.
Back to 2-SAT where we have $(x_i \vee x_j)$ clauses. What is the probability our i # of matches increases $i = i + 1$ or decreases $i = i - 1$? Note the algorithm picks an unsatisfied clause $C = (\alpha \vee \beta)$ where in A (our current assignment) $\alpha = F$ and $\beta = F$. In A^*, at least one is T. Suppose $\alpha = T$. We pick to flip α, β to true with probability $\frac{1}{2}$ each. If we pick α then $i = i + 1$ and if we pick β then i goes up or down.
Applying the Markov chain results $E(\text{number of steps to reach } i = n) \leq n^2$.
Note the algorithm might succeed earlier with $A \neq A^*$ but A might still satisfy E. Furthermore, the probability of the $i = i + 1$ case is higher than $\frac{1}{2}$ since the choice of flipping β might improve our i. Therefore our analysis is not tight (upper bound).
What value of t should we pick such that $P(\text{not reaching } i = n \text{ after } t \text{ steps}) \leq \frac{1}{2}$?
By Markov's inequality we have $P(X \geq c\mu) \leq \frac{1}{c}$. In our case we have $\mu = n^2$ (our expected value) so we choose $c = 2$ such that $P(t > 2n^2) \leq \frac{1}{2}$. That is we set $t = n^2$ to have a probability of error $\leq \frac{1}{2}$.
The # of steps we take is $O(n^2)$ thus our runtime is $O(n^2 \text{poly}(n, m))$ where $\text{poly}(n, m)$ is the time for testing A, etc.
Note that this is not a breakthrough for 2-SAT: there are better deterministic algorithms.
For 3-SAT, our unsatisfied clause becomes $C = (\alpha \vee \beta \vee \gamma)$. Since our current A does not satisfy C, we have $\alpha = \beta = \gamma$.
Since A^* does satisfy C, at least one of the variables is T. Suppose $\alpha = T$, then $P(\text{alg. flips } \alpha) = \frac{1}{3}$ thus $P(i \text{ increases}) \geq \frac{1}{3}$.
Thus we have a random walk with $p_{i,i+1} = \frac{1}{3}$ and $p_{i,i-1} = \frac{2}{3}$. The expected # of steps to reach n is approx. 2^n, which is as bad as our brute force algorithm.
A different randomized algorithm (Schoning 1999) introduces two new ideas

1. start with random truth-assignment of A
2. increasing # of trials (t) is not helpful: we're likely to get stuck at 0, so we pick a new random assignment A

The algorithm follows

```
repeat s times:
  pick random A
  repeat t = 3n times:
    if A satisfies E:
      return YES
    pick unsatisfiable clause C
      same as Papadimitriou
return NO
```

Fact: in the inner loop ($t = 3n$ times), we have $P(\text{error}) \leq 1 - \left(\frac{3}{4}\right)^n$. If we set $s = c(\left(\frac{4}{3}\right)^n)$ (our outer loop iteration count), we get an overall probability of error

$$\begin{aligned} P(\text{error}) &\leq \left(1 - \left(\frac{3}{4}\right)^n\right)^{c(4/3)^n} \\ &\leq \frac{1}{e^c} \end{aligned}$$

where the last line follows since $(1 - \frac{1}{a})^a \leq \frac{1}{e}$, which follows from $\frac{1}{e^x} \geq 1 - x$.
So ultimately we can get a probability of error $\leq \frac{1}{2}$ with # of steps $O(\left(\frac{4}{3}\right)^n \cdot n)$ so the runtime is $\approx O(1.3^n \text{poly}(m, n))$ vs. the best known deterministic algorithm with $O(1.465^n)$ runtime.

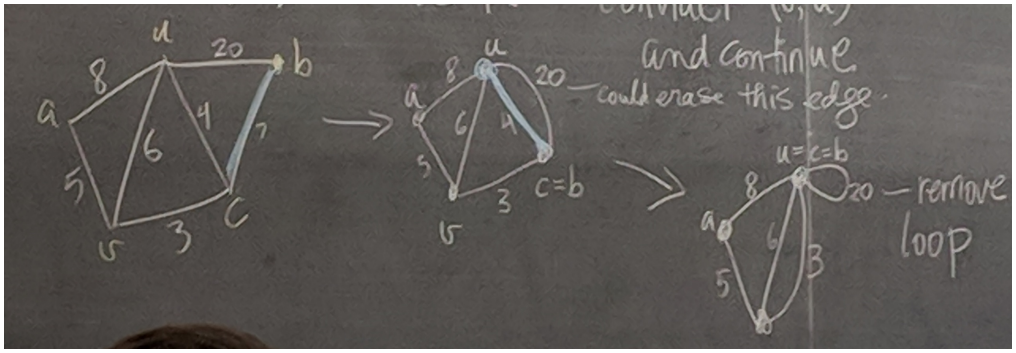
12 October 24, 2018

12.1 Minimum spanning tree

MST: Given undirected graph $G = (V, E)$ where $|V| = n$ and $|E| = m$ with weights on edges $w : E \to \mathbb{R}^+$ (assume distinct), find the minimum (smallest sum of weights) spanning (reaches all vertices) tree (no cycles).
One can also extend this to *minimum spanning forest* for disconnected graph.
We follow two basic rules:

Exclusion rule If cycle c has max weight edge e, then $e \notin$ MST: delete e and continue.

Inclusion rule If vertex v's minimum weight incident edge is (v, u), then $(v, u) \in$ MST: contract (v, u) and continue. If there is a loop (self-edge), remove it.



There are a number of MST algorithms:

Kruskal's (1956) Uses the inclusion rule (and checking the exclusion rule) repeatedly:

```
repeat:
  e = (u,v) = minimum weight edge
  put uv in T and contract uv
  (usual test: does (u,v) make cycle with T which is equivalent to
  saying if (u,v) is a loop in contracted G)
```

Implementation: sort edges then use union-find to detect cycles: $O(m \log n)$ time.

Prim's (1957) Also uses the inclusion rule repeatedly:

```
pick start vertex s
repeat:
  find minimum weight edge e = (s,v)
  put e in T, contract e
  s now has incident edges from v
```

Implementation: with a regular heap we have $O(m \log n)$; with a Fibonacci heap we have $O(n \log n + m)$.

Boruvka's (1926) Good for parallel algorithms. Idea: in one step, apply the inclusion rule at all vertices.

Since we choose n minimum incident edges each time and each minimum incident edge can be shared by at most two vertices, we end up with at most $\leq \frac{n}{2}$ vertices after contraction.

We will not use *all* vertices: just ensure every vertex gets contracted eventually. One Boruvka step is:

```
unmark all vertices
for each v in V:
  if v is unmarked:
    find min weight edge e = (v,u)
    add e to T
    contract v to u
    mark u and v
```

Once all vertices and contracted vertices have been marked (# of vertices after one Boruvka step $\leq \frac{n}{2}$), we must repeat until one vertex remain.

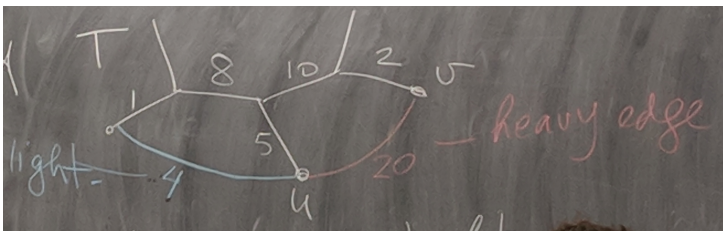
Implementation: to find minimum weight edge $e = (v, u)$ and to contract takes $O(deg(v))$ so the total work per step is $O(n) + O(\sum deg(v)) = O(m + n)$. Since we take at most $\log n$ steps we have $O((m + n) \log n)$ runtime.

Chazelle 1997 $O(m\alpha(m, n))$ where $\alpha(m, n)$ is the inverse Ackermann function.

It is a difficult algorithm to implement.

Open question: is there an $O(m + n)$ time algorithm?
We will look at a $O(m + n)$ randomized algorrithm for MST (Karger 1993). Idea: use random sample and apply the exclusion rule.

```
// return MST of each connected component G = (V,E)
MST(E):
  // size r to be determined
  take random sample R subset of E of size r
  T = MST(R)
  for each edge (u,v) in E:
    classify (u,v) as heavy or light
  E = E - {heavy edges}
  return MST(E)                // implicitly terminates once E is a tree
```



We define an edge (u, v) to be **heavy** (relative to T) if $(u, v) \notin T$ and (u, v) is heavier than all edges in $u \to v$ path in T. Otherwise (u, v) is **light**. Note: edges of T are light. If there is no other path $u \to v$ (e.g. if (u, v) connects two components) then (u, v) is light.
Correctness follows from the exclusion rule (which says we should always throw away heavy edges).
A light edge can be used to improve T: add light edge and remove heavier edge of cycle.
How does algorithm terminate? When all edges outside T are heavy.
Fact: we can classify heavy/light for edges in $O(m)$ time. This is hard so it won't be covered.
For runtime complexity, we require the **Sampling Lemma**:

Lemma 12.1 (Sampling lemma)**.** We claim $E(\#\text{ of light edges}) \leq \frac{m \cdot n}{r}$.

Proof. (T.Chan version). Consider random edge e. It is enough to show $P(e \text{ is light}) \leq \frac{n}{r}$.
We use backwards analysis: consider $R' = R \cup \{e\}$ (after we sample an edge e to our current R of size r edges). e is a random element of R' and e is light in R iff e is in $\text{MST}(R')$. $\text{MST}(R')$ has $\leq n - 1$ edges, thus we have

$$P(e \text{ is light}) \leq \frac{n-1}{r+1} < \frac{n}{r}$$

So the expected run time is

$$T(m, n) = T(r, n) + O(m + n) + T(\frac{mn}{r}, n)$$

where each term corresponds to $\text{MST}(R)$, the heavy/light classifying step, and the recursive call on E'.
We choose $r = 2n$, thus we have $T(m, n) = T(2n, n) + O(m + n) + T(\frac{m}{2}, n)$.
One last refinement: we can reduce the # of vertices. At each recursive call, first run 3 steps of Boruvka's ($O(m+n)$ time), such that # of vertices $\leq \frac{n}{8}$. we can rewrite our recurrence as

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

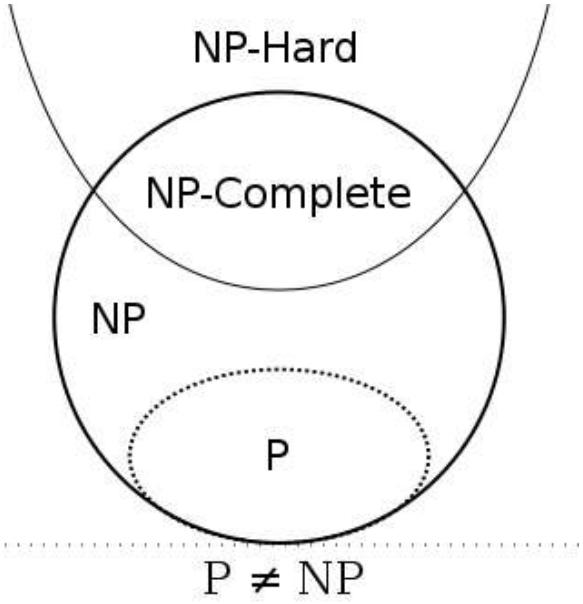
where d is some constant.
We prove by induction (and by guessing) that $T(m, n) \leq c(m + n)$ for some $c > 0$.

$$\begin{aligned} T(m, n) &\leq c(\frac{n}{4} + \frac{n}{8}) + c(\frac{m}{2} + \frac{n}{8}) + d(m + n) \\ &\leq (\frac{c}{2} + d)n + (\frac{c}{2} + d)m \end{aligned}$$

so we solve $\frac{c}{2} + d \leq c \Rightarrow c \geq 2d > 0$ as desired. □

13 October 29, 2018

13.1 Approximation algorithms



Ladner proved if $P \neq NP$, then there are problems in between P and NP-complete that are in NP (e.g. graph isomorphism, factoring; not too many other natural candidates).
The major open question is $P = NP$? Similarly, does randomization help is an open question. Regardless, randomization is still a good tool for faster and algorithms.
What can we do with NP-complete problems?

- heuristics
- exponential algorithm (but as fast as possible e.g. randomize SAT)
- other models (quantum, natural)
- approximation algorithms
- parameterized complexity

For **approximation algorithms**, we give up on exact algorithms for optimization problems but want a guarantee on quality of solution. Formally:

Definition 13.1 (Approximation algorithm)**.** An **approximation algorithm** for optimization problem finds in polynomial time a solution "close to" optimum, where "close to" means the difference is small (rare) OR the ratio of solution to optimum is good.

An example where the difference is small:

Example 13.1. Edge colouring in graph: colour edges such that if two edges are incident, they have different colours.
Note that the minimum # of colours $\geq$ max degree of anyy vertex.

Theorem 13.1 (Vizing's theorem)**.** Let Δ be the maximum degree of any vertex. Then

$$\Delta \leq \#\text{ edge colours} \leq \Delta + 1$$

Furthermore there is a polynomial time algorithm (we won't study it) to colour any graph with $\Delta + 1$ colours.

BUT edge colouring is NP-hard even though we can approximate to within +1.
This is rare: usually approximation algorithms are bounded by a ratio e.g. TSP $\leq 2OPT$ for metric case.

13.2 Vertex and set cover

Example 13.2. Vertex cover: given 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.
Decision version is NP-complete. Relationship to independent set (set of vertices where no two are adjacent): U is a minimum vertex cover iff $V - U$ is the maximum independent set.

Exercise 13.1. Show that if there is an approximation algorithm (polynomial time) for vertex cover good within an additive constant then $P = NP$.

A greedy algorithm for vertex cover could be:

```
1  C = []
2  repeat until no edges remain:
3    C = C U {one vertex of max degree}
4    remove covered edges
```

We will show $|C| \leq O(\log n)|OPT|$.

Exercise 13.2. Find examples to show the algorithm can give $\frac{|C|}{|OPT|} \geq \Omega(\log n)$.
(start with example where greedy $\neq$ OPT).

Example 13.3. More general problem with **set cover**: given collection of sets $S_1, \ldots, S_k$ where $S_i \subseteq \{1, \ldots, n\}$ find a minimum subcollection of S_i's such that every element $1, \ldots, n$ is covered i.e. find $C \subseteq \{1, \ldots, k\}$ such that $\forall i \in \{1, \ldots, n\}$ $i \in S_j$ for some $j \in C$.
For example, let $n = 4$ and

$$S_1 = \{1, 2\}$$
$$S_2 = \{3\}$$
$$S_3 = \{2, 3, 4\}$$
$$S_4 = \{1, 3, 4\}$$

We only require two sets to cover e.g. S_1, S_4 or S_1, S_3.
Vertex cover is a special case: elements are edges of graph and sets are edges incident to one vertex. Note every element is in two sets.

Question 13.1. Is every set cover problem a vertex problem? No, only if every element is in 2 sets.

A greedy algorithm for set follow is as follows:

```
1  C = []
2  while there are uncovered elements:
3    S_i = set that covers max # of uncovered elements
4    C = C U {i}
```

Theorem 13.2. The greedy algorithm for set cover is polynomial time that gives $|C| \leq O(\log n)|OPT|$ where OPT is the minimum cover.

Proof. (from Varzirani's book, shorter than CLRS). When we choose some S_i (cost 1) we will distribute the cost to newly covered elements and then sum over elements (let $c(e)$ denote cost we assign to element e).
Let S be the first chosen set (S has maximum size). For all $e \in S$, we have $c(e) = \frac{1}{|S|}$ (note $\sum c(e) = 1$).
Note that we have the following:

$$|S| \geq \text{ max size of } S_i \geq \text{ avg size of } S_i's \geq \frac{n}{|OPT|}$$

More generally, let $e_1, e_2, \ldots, e_i, \ldots, e_n$ where we order the elements as they are covered (lots of ties of course) e.g. $e_1, \ldots, e_{i-1}$ covered by S, $e_i, \ldots$ covered by S_i, etc.
Consider element e_i: first covered by S'. Suppose S' covers t new elements: so $c(e_i) = \frac{1}{t}$.
The number of uncovered elements remaining when we choose S' is $\geq n - i + 1$. Note that S' covers t elements, which was maximum size amongst all S_i's thus

$$\max \geq \text{ avg } \geq \frac{n-i+1}{|OPT|}$$

that is $t \geq \frac{n-i+1}{|OPT|}$, so $c(e_i) = \frac{1}{t} \leq \frac{|OPT|}{n-i+1}$.
Thus $|C|$, the number of chosen S_i's is

$$\begin{aligned} |C| &= \sum_{i=1}^{n} c(e_i) \\ &\leq \sum_{i=1}^{n} \frac{|OPT|}{n-i+1} \\ &= |OPT|(1 + \frac{1}{2} + \ldots + \frac{1}{n}) \end{aligned}$$

where the summation is the nth Harmonic number so $|C| \leq O(\log n)|OPT|$. □

14 October 31, 2018

14.1 Randomized vertex vcover

Recall we had a greedy algorithm with cover $|C| \leq O(\log n)|OPT|$.
Here is a seemingly trivial randomized algorithm:

```
1  V = []
2  while E is not empty:
3    choose e = (u,v) at random
4    add u,v to V
5    remove all edges incident to u,v
```

Lemma 14.1. This algorithm finds a vertex cover C with $|C| \leq 2|OPT|$.

Proof. The edges we choose in the first step of the loop form a matching M where no edges are adjacent (we cover edges of M with different vertex for each edge).

We know that $|OPT| \geq |M|$ since each edge needs at least one vertex in the optimal cover, and since each edge in the matching adds two vertices we have

$$|C| = 2|M| \leq 2|OPT|$$

□

The best known approximation factor (in polynomial time) for vertex cover is 2.

Exercise 14.1. Recall: U is a vertex cover iff $V - U$ is an independent set. Does 2 approximation for vertex cover give any good approximation for the independent set problem?

14.2 Weighted vertex cover

Every vertex v has weight $w(v)$. Find the minimum weight vertex cover i.e. find a vertex cover $U \subseteq V$ to minimize $\sum_{v \in U} w(v)$.
An approximation algorithm for weighted vertex cover: use general technique where we transform the problem into a linear program.
Precisely, we express it as an integer linear program.
Let $x(v)$ be a variable for each vertex where $x(v) = 1$ if we choose v and 0 otherwise.
Thus we want to

$$\min_{v \in V} w(v)x(v)$$

subject to

$$x(u) + x(v) \geq 1 \qquad \forall (u, v) \in E$$

where the constraint implies we choose at least one of u and v and possibly both.
Note that integer LP is NP-hard, but regular LP is in P (ellipsoid method or Simplex for practical but not provable P time). Idea: relax integer condition and solve for $0 \leq x(v) \leq 1$, which we can do in polynomial time. We then round the solution. Suppose $\bar{x}$ is our LP solution, then we let $C = \{v \mid \bar{x}(v) \geq \frac{1}{2}\}$ our vertex cover solution.

Claim. We claim C is a vertex cover and $w(C) \leq 2OPT$ where OPT is the minimum weight of any vertex cover.

Proof. C is a vertex cover. Consider edge (u, v) where we have $\bar{x}(u) + \bar{x}(v) \geq 1$. At least one vertex (say u) has $\bar{x}(u) \geq \frac{1}{2}$ so $u \in C$.
We prove $w(C) \leq 2OPT$. Note that

$$\begin{aligned}
\text{OPT} &= \text{OPT to Int LP} \\
&\geq \text{OPT to LP} \\
&= \sum_{v} w(v)\bar{x}(v) \\
&\geq \sum_{v, \bar{x}(v) \geq \frac{1}{2}} w(v)\bar{x}(v) \\
&\geq \sum_{v, \bar{x}(v) \geq \frac{1}{2}} \frac{1}{2} w(v) \\
&= \sum_{v \in C} \frac{1}{2} w(v)
\end{aligned}$$

where line 2 follows since our solution to the LP can be equivalent or better than the integer LP. So $OPT \geq \frac{1}{2}w(C)$ thus $w(C) \leq 2OPT$. □

To recap for vertex cover:

Greedy $O(\log n)$ approximation factor

Randomized pick edges 2 approximation

Weighted vertices Integer LP with LP relaxation gives 2 approximation

14.3 Set cover revisited

Recall set cover: given elements $1, \ldots, n$ and sets $S_1, \ldots, S_k$ where $S_i \subseteq \{1, \ldots, n\}$, find the minimum # of sets to cover all elements.
Greedy still works with $O(\log n)$ approximation.

Exercise 14.2. For integer LP version: what is the approximation factor?
Define $f = \max_{\text{element } i}\{\#\text{ of sets containing } i\}$. We get an approximation $\leq f \cdot OPT$.
For the vertex cover version of set cover, we have $f = 2$ (each edge can be covered by at most 2 verties or 2 sets).
Which is better (f or $O(\log n)$) depends on the problem.

Is there a constant factor approximation for set cover (in polynomial time)? No, unless $P = NP$ (later in course).

14.4 Approximation factors

Definition 14.1 (Approximation factor)**.** A ρ-approximation algorithm A for a **minimization** problem guarantees for all input I we have $A(I) \leq \rho OPT(I)$ where $A(I)$ and $OPT(I)$ are the value of the objective functions given by algorithm A and the optimal value, respectively. Note $\rho \geq 1$ and we want ρ close to 1.
Similarly a ρ-approximation for a **maximization** problem guarantees $A(I) \geq \rho OPT(I)$ where $\rho \leq 1$ (some sources use $\frac{1}{\rho}$ where $\rho \geq 1$).

Example 14.1. Maximization problem: finding the max cut in a graph i.e. given graph $G = (V, E)$ find $S \subseteq V$ such that the **cut** edges that go from S to $V - S$ are maximized (in quantity).
Note that minimum cut (e.g. for network flows) is in polynomial time but maximum cut is NP-hard.
An approximation algorithm for max cut uses the idea of local improvement:

```
1  S = any subset of V
2  repeat until no further improvements:
3    for each vertex v:
4      if moving v to other set improves cut, do it
```

Lemma 14.2. We claim $c(S) \geq \frac{1}{2}m$ where $c(S)$ is the # of edges in our cut and m is the # of edges.
Since $m \geq OPT$ ($m = OPT$ if graph is bipartite) then $c(S) \geq \frac{1}{2}OPT$.

Proof. Let $e(S)$ be the number of edges inside S currently. For some v, let $d_S(v)$ and $d_{V-S}(v)$ be the number of edges from v to S and $V - S$, respectively.
For all $v \in S$ we must have $d_S(v) \leq d_{V-S}(v)$ since if we moved v to $V - S$ this would decrease the number of our cut edges. Therefore

$$\sum_{v \in S} d_S(v) \leq \sum_{v \in S} d_{V-S}(v)$$

$$\Rightarrow 2e(S) \leq c(S)$$

Similarly for $v \in V - S$ we must have $d_{V-S}(v) \leq d_S(v)$ so

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

Thus we have

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

so $c(S) \geq \frac{1}{2}m$. □

15 November 5, 2018

15.1 Max SAT

The Max SAT problem: given a set of m clauses (CNF) in n Boolean variables, find true/false assignments to variables to make maximum number of clauses true.

Example 15.1. Given the clauses $(\bar{x}_1 \vee \bar{x}_2), (x_1 \vee x_3), (x_2 \vee x_3)$ and $\bar{x}_3$ note that $x_1 = T, x_2 = T, x_3 = F$ gives 3 clauses satisfied.

The decision version of this asks whether $\geq k$ of these clauses can be satisfied. Even for small cases where size of clauses is ≤ 2 terms (Max 2-SAT) is **NP-hard** (however, determining if **all clauses** for 2-SAT is in P).
Idea: reduce 3-SAT to Max 2-SAT. For example, given 3-SAT $(l_1 \vee l_2 \vee l_3)$ becomes 10 2-SAT clauses, such that the 3-SAT clause is satisfied iff 7 of the 2-SAT clauses are satisfied.
We argue a **randomized algorithm** for Max SAT where we can get 0.878-approximation. This algorithm is complicated but we will show some of the key ideas.

1st algorithm (simple) Pick truth assignments at random i.e. x_i is true with probability $\frac{1}{2}$. For any clause C

$$P(C \text{ is satisfied}) = 1 - P(C \text{ is NOT satisfied})$$

Suppose C has t variables, then we have $1 - \frac{1}{2^t} \geq \frac{1}{2}$. Thus

$$\begin{aligned} E(\#\text{ clauses satisfied}) &= E(\sum_{\text{clause } C} \sigma_C) && \sigma_C = 1 \text{ if } C \text{ is satisfied, 0 otherwise} \\ &= \sum_C E(\sigma_c) && \text{linearity of expectation} \\ &\geq m\frac{1}{2} \\ &\geq \frac{1}{2}OPT \end{aligned}$$

where OPT is the maximum # of clauses we could possibly satisfy in the optimal solution.

One consequence of the above:

Theorem 15.1. There always exists a truth value assignment that satisfies $\geq \frac{1}{2}$ of the clauses.

Proof. By "probabilistic method" (powerful technique): if expected value of a random variable is α then there exists a value for the variable at least α (i.e. the expectation requires there to be some value with non-zero probability greater than or equal). □

2nd algorithm (integer LP and randomized rounding) We create one variable x_i for each Boolean variable a_i (n x_i's) and one variable y_i for each clause c_j (m y_i's).

Our LP is thus

$$\max \sum y_i$$

where we have one constraint per clause and one constraint per Boolean variable where $y_j = 1$ iff x_i's satisfy clause C_j, and $x_i = 0 \iff a_i = F$ and $x_i = 1 \iff a_i = T$.

For the clauses, suppose we had $C_1 = (\bar{a}_1 \vee a_2)$. Then we have

$$y_1 \leq (1 - x_1) + x_2$$

where $y_1 = 1 \Rightarrow x_2 = 1$ or $x_1 = 0$.

We need to add an addition constraint (for integer LP) where we require $x_i = 0$ or 1, $y_i = 0$ or 1 (we can require $x_i, y_i \leq 1$ and apply integer constraint).

Claim. This integer LP exactly solves the integer linear program for Max SAT.

We now relax the ILP to a linear program:

$$\begin{aligned} 0 \leq x_i \leq 1 \qquad & \forall i = 1, \ldots, n \\ 0 \leq y_j \leq 1 \qquad & \forall j = 1, \ldots, m \end{aligned}$$

We can solve this LP in polynomial time (then round, which basically hard sets x_i to either true or false).

We use **randomized rounding**: we set a_i to true with probability x_i and 0 otherwise.

What is the expected # of clauses satisfied?

Example 15.2. Suppose we had $C_2 = (a_1 \vee a_2)$ and $y_2 \leq x_1 + x_2$. Note that

$$\begin{aligned} P(C_2 \text{ is satisfied}) &= P(a_1 = 1 \text{ or } a_2 = 1) \\ &= x_1 + x_2 - x_1 x_2 \\ &\geq x_1 + x_2 - \left(\frac{x_1 + x_2}{2}\right)^2 && \text{geo mean} \leq \text{arith mean} \\ &\geq y_2 - \frac{y_2^2}{4} && x - \frac{x^2}{4} \text{ is increasing for } x \in [0,2] \text{ and } x_1 + x_2 \geq y_2 \\ &\geq y_2 - \frac{y_2}{4} && 0 \leq y_2 \leq 1 \\ &= \frac{3}{4} y_2 \end{aligned}$$

So the expected # of clauses satisfied is

$$\geq \frac{3}{4} \sum y_i = \frac{3}{4} OPT_{LP} \geq \frac{3}{4} OPT_{ILP} = \frac{3}{4} OPT_{maxSAT}$$

3rd algorithm (de-randomization) De-randomization is hard in general but easy for this algorithm.

We attempt to find a deterministic algorithm to find truth values that satisfy $\geq \frac{1}{2}$ of the clauses.

Example 15.3. Given the following clauses, their expectation of satisfaction are

$$\begin{aligned}
&\bar{x}_1 : \frac{1}{2} \\
&(x_1 \vee \bar{x}_2) : \frac{3}{4} \\
&(x_1 \vee x_2 \vee \bar{x}_3) : \frac{7}{8} \\
&(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) : \frac{7}{8} \\
&(x_1 \vee x_2 \vee x_3) : \frac{7}{8}
\end{aligned}$$

where the sum of the expectations is $3\frac{7}{8}$.

if we set/fix $x_1 = F$ we get expectations

$$\begin{aligned}
&\bar{x}_1 : 1 \\
&(x_1 \vee \bar{x}_2) : \frac{1}{2} \\
&(x_1 \vee x_2 \vee \bar{x}_3) : \frac{3}{4} \\
&(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) : 1 \\
&(x_1 \vee x_2 \vee x_3) : \frac{3}{4}
\end{aligned}$$

where a clause is 1 if $\bar{x}_1$ is in clause (since $\bar{x}_1$ becomes true) and $(1 - \frac{1}{2^t})$ otherwise where t are the number of variables not containing x_1. The sum of expectations is now 4.

If we had set $x_1 = T$ we'd get $0 + 1 + 1 + \frac{3}{4} + 1 = 3\frac{3}{4}$.

Remark 15.1. Without fixing x_1, our expectation E_{SAT} can be computed by simply conditioning on x_1 i.e.

$$\begin{aligned}
E_{SAT} &= \frac{1}{2} E_{SAT}(x_1 = F) + \frac{1}{2} E_{SAT}(x_1 = T) \\
\Rightarrow 3\frac{7}{8} &= \frac{1}{2}(4) + \frac{1}{2}(3\frac{3}{4})
\end{aligned}$$

So *at least one choice* gives $E \geq$ original expectation: in this case choose $x_1 = F$.

Similarly $E_{SAT}(x_2 = F) = 1 + 1 + \frac{1}{2} + 1 + \frac{1}{2} = 4$ and $E_{SAT}(x_2 = T) = 1 + 0 + 1 + 1 + 1 = 4$, where when $x_2 = T$ we see all clauses are decided where 4 of the 5 clauses are satisfied i.e. $k \geq 4$ is YES.

We can show that we always satisfy at least half the clauses (in general without randomness in polynomial time).

We can apply the best result from the 1st and 2nd algorithm, and the de-randomization algorithm separately to get $\frac{3}{4}$-approximation.

15.2 Summary of approximation problems and algorithms covered

We have looked at minimization problems where

Metric TSP 1.5-approximation (we saw 2-approximation)

Vertex cover 2-approximation

Set cover $O(\log n)$-approximation

as well as maximization problems

Max cut $\frac{1}{2}$-approximation

Max SAT $\frac{3}{4}$-approximation

Some open questions that remain:

1. Which problems have constant-factor approximations?
2. How close to 1 can the approximation factors be?

Note that approximation algorithms are only interesting if $P \neq NP$ (otherwise we can solve all with approximation factor 1).

16 November 7, 2018

16.1 Set packing

Set packing: given elements $1, \ldots, n$ and sets $S_1, \ldots, S_k$ such that $S_i \subseteq \{1, \ldots, n\}$, find maximum number of S_i's such that no two intersect.

Exercise 16.1. The graph version is independent set: given graph $G = (V, E)$ find maximum subset $U \subseteq V$ such that no edge (u, v) has both endpoints in U.
Show that any set packing problem can be turned into an independent set problem (different from set cover vs. vertex cover).

Set packing is NP-hard. Fact: There is no $n^{1-\epsilon}$ factor approximation for independent set unless $P = NP$.
We will look at a geometric version of set packing with PTAS (polynomial time approximation scheme; good solution). Given n unit squares in $\mathbb{R}^2$, find maximum pairwise disjoint subset of squares. Note if two squares lie on the same edge they intersect.
Application: map labelling. Points are cities and each city can have 1 of 4 possible touching labels (our unit squares). Label the maximum number of cities.
A **simple approximation algorithm:** Pick a square. Throw away all squares that intersect.

Claim. This is a $\frac{1}{4}$-approximation.

Proof. Every square in OPT that is NOT in A, the set of squares our algorithm chooses, intersects a square in A. Otherwise our algorithm would eventually pick the square and it would be in A.
Every square in A intersects ≤ 4 squares of OPT because squares of OPT are disjoint (and unit size).

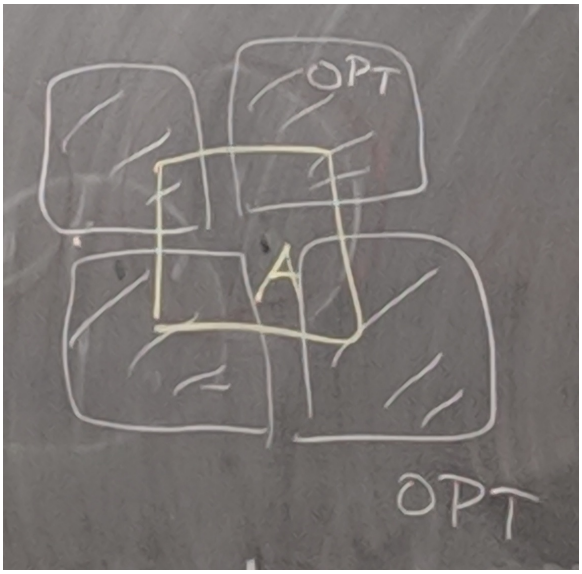


Figure 16.1: At most 4 squares from OPT can intersect a chosen square in A.

Thus $A \geq \frac{1}{4}OPT$ so $OPT \leq 4A$. □

An equivalent formuation to set packing of unit squares: given n points in $\mathbb{R}^2$ choose a maximum number of points such that for any p, q chosen $|p_x - q_x| > 1$ or $|p_y - q_y| > 1$. This is a formulation where p and q are the coordinates of the centre of our unit squares.

16.2 Grid approximation for geometric packing

A harder $\frac{1}{4}$-approximation algorithm: **grid approximation algorithm**. We lay down a unit grid across our points. Note that we can always work with a grid of at most size $n \times n$: if a cluster of points span farther than n unit squares then we can separate them as disjoint subproblems since we are guaranteed to have empty strips of unit squares in between.
We find a set of shaded squares that are separated by 1 unit square. Let $R_0 = \{(x, y) \mid \lfloor x \rfloor \text{ is even}, \lfloor y \rfloor \text{ is even}\}$ be our first set of shaded squares. We then pick one point from each of the shaded squares if non-empty: this is our optimal solution for R_0. Observe these points satisfy our constraint and are a solution.
We do this also for

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

where R_i partitions our plane. Let Q_i be the optimal solution for $R_i \cap P$ (P is our set of points).
Let Q be the largest of Q_i's.

Claim. We claim $|Q| \geq \frac{1}{4}OPT$.

Proof. Note that

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

where the second inequality follows from the fact that Q_i gives the maximum number of independent points in R_i, which our OPT solution choose an equivalent or fewer number of independent points.
Therefore $|Q| \geq \frac{1}{4} OPT$. □

16.3 Larger grid approximation

Idea: use larger shaded blocks. Fix a $k \geq 2$.
That is our set of shaded squares for a given i, j is

$$R_{ij} = \{(x, y) \mid \lfloor x \rfloor \mod k \neq i, \lfloor y \rfloor \mod k \neq j\}$$

where $O \leq i \leq k-1$ and $0 \leq j \leq k-1$.
For $k = 3$, this correspond to size 2 shaded quares separated by unit square strips.

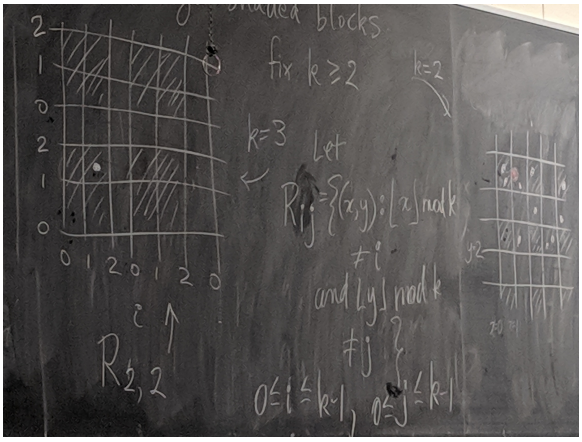


Figure 16.2: Using larger blocks of shaded squares: in this example we see $R_{2,2}$ for $k = 3$.

In our previous algorithm with unit shaded squares, $k = 2$.
How many R_{ij}'s are there? k^2 (we can shift our shaded square up to k times horizontally and vertically).
Every point is in how many R_{ij}? $(k-1)^2$: every point lives in some unit square and so there are $k-1$ of the R_{ij}s where the columns cover that unit square and $k-1$ of the R_{ij}'s where the rows cover that unit square (every R_{ij} has a unit strip of non-shaded row/columns).
The number of independent points (i.e. disjoint squares) we can choose in one shaded block is $\leq (k-1)^2$ (we can see this easily for $k = 3$ where we can choose up to 4 points that are outside the unit square centred at the intersection of the four shaded squares in a given block).
We use brute force for every shaded block: try all subsets of size $(k-1)^2$ for $\left(\frac{n}{k}\right)^2$ shaded blocks.
Note that the number of subsets in each shaded block is $O(n^{(k-1)^2})$ (think $\binom{n}{(k-1)^2}$).

Thus the time to find an optimal solution Q_{ij} in R_{ij} is $O(\text{poly}(n,k) \cdot n^{(k-1)^2})$ (since $n^{(k-1)^2}$ dominates our other terms).
Take $Q = \max Q_{ij}$ where $|Q| \geq \frac{1}{k^2} \sum_{i,j=0}^{k-1} |Q_{ij}|$ (max $\geq$ avg).
We thus have

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

where the first line follows because every point is in $(k-1)^2$ of the R_{ij}'s. Therefore

$$|Q| \geq \frac{(k-1)^2}{k^2} OPT$$

k	2	3	4	5	
approx. factor	$\frac{1}{4}$	$\frac{4}{9}$	0.64	0.81	0.9

As k gets larger, approximation factor goes to 1 but run time goes to exponential (as $k \to n$ we are essentially brute forcing the entire problem).

16.4 Approximation scheme definition

Definition 16.1. An **approximation scheme** is an algorithm A, input I, and parameter ϵ for minimization problem where $A(I,\epsilon) \leq (1+\epsilon)OPT$.
For a maximization problem, we have $A(I,\epsilon) \geq (1-\epsilon)OPT$.
Note that $A(I,\epsilon)$ is the solution returned by A.

Definition 16.2. A **polynomial time approximation scheme (PTAS)**: for each fixed ϵ, A runs in polynomial time in terms of $n = |I|$ (e.g. $O(n^{1/2})$).

Definition 16.3. A **full polynomial time approximation scheme (FPTAS)**: A runs in polynomial time in terms of $n = |I|$ and $\frac{1}{\epsilon}$ e.g. $O(\left(\frac{1}{\epsilon}\right)^2 n^3)$.
We will see an example of a FPTAS next week.

Note for our shifting grid algorithm above, we had an approximation factor of

$$\begin{aligned}\frac{(k-1)^2}{k^2} &= \frac{k^2 - 2k + 1}{k^2} \\ &= 1 - \frac{2k-1}{k^2}\end{aligned}$$

so $\epsilon = \frac{2k-1}{k^2}$ i.e. $\epsilon = \theta(\frac{1}{k})$ so $k = \theta(\frac{1}{\epsilon})$.
The runtime as before is thus $O(\text{poly}(n,k) \cdot n^{\left(\frac{1}{\epsilon}\right)^2})$ which is not polynomial time in $\frac{1}{\epsilon}$ thus it is not a FPTAS.

17 November 12, 2018

17.1 Bin packing

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

For example when $S = \{\frac{1}{3}, \frac{1}{2}, \frac{1}{2}\}$, we require a minimum of two bins.
There are two versions:

Online version Numbers in S arrive one-by-one and must be placed in bins immediately.

The **First Fit** algorithm inserts element into the first bin that fits it.

Given $S = (3, 8, 1, 2, 5, 1)$ with size 10 bins, First Fit requires 3 bins while the optimal offline solution requires 2 bins.

Claim. First Fit has approximation ratio 2 (compared to OPT packing).

Proof. Let m be the number of bins used by First Fit.

We cannot have 2 bins that is $\leq \frac{1}{2}$ full (otherwise the elements in one would have been placed in the other bin).

Therefore there are $m-1$ bins that are $> \frac{1}{2}$ full so

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

In fact the approximation ratio is better where First Fit $\leq 1.7OPT + 1$ bins: its *asymptotic approximation ratio* is 1.7 i.e. as $OPT \to \infty$ then the $+1$ is irrelevant. □

Offline version We know all of S ahead of time.

We can first sort S from largest to smallest and then apply First Fit.

Fact: The asymptotic approximation ratio is $\frac{11}{9} = 1.22$.

There exists a **PTAS for offline bin packing**:

Theorem 17.1 (de la Vega and Lueker 1981)**.** For any constant $\epsilon > 0$ there is a polynomial algorithm that uses $\leq (1+\epsilon)OPT + 1$ bins (asymptotic approximation factor $(1+\epsilon)$). Note the $+1$ is crucial.

Exercise 17.1. Prove for 2 bins there is no polynomial approximation algorithm with approximation factor < 1.5 unless $P = NP$.

Hint: use fact that the Partition problem (given a set of integers is there a way to partition the integers into two sets such that the sum of both sets are equal) is NP-complete.

The PTAS algorithm builds up from special cases:

Case 1 For all i, $s_i \geq \delta$ for some constant δ and there are only k possible values for s_i's, k a constant.

We will solve exactly using brute force. There are $\leq \frac{1}{\delta}$ elements in each bin (since bin size is 1 and all our elements are at least size δ) so $\leq k^{\frac{1}{\delta}}$ ways to fill each bin (denote these as "patterns").

There are $\leq n$ bins and for each bin we have $k^{\frac{1}{\delta}}$ patterns so we have $O(n^{k^{\frac{1}{\delta}}})$. This is polynomial time for fixed k, δ.

We can do better using integer linear programming.

Case 2 For all i, $s_i \geq \delta$, δ constant (no k constant possible values).

Idea: round s_i's so we have only k values and apply case 1. We should round up to prevent underestimation.

Remark 17.1. What if we had $S = \{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\}$? If we rounded up to 0.34, we would up end with 2 bins instead of 1 bin.

We sort $s_1 \leq s_2 \leq \ldots \leq s_n$ then partition the numbers into k segments. For the ith segment where $i \in \mathbb{Z}^+$ we round all values $S_{(i-1)\lceil \frac{n}{k} \rceil + 1}, \ldots, S_{i\lceil \frac{n}{k} \rceil - 1}$ up to the value of $S_{i\lceil \frac{n}{k} \rceil}$ (i.e. we round each value to the maximum value in its segment).

Our modified input S^+ would look like

$$S^+ = \{S_{\lceil \frac{n}{k} \rceil}, \ldots, S_{\lceil \frac{n}{k} \rceil}, S_{2\lceil \frac{n}{k} \rceil}, \ldots, S_{2\lceil \frac{n}{k} \rceil}, \ldots, S_n, \ldots, S_n\}$$

where we have $\frac{n}{k}$ copies of each of the k possible values and $|S^+| = n$.

We can solve S^+ exactly using case 1 and this provides a solution (same # of bins) for S.

For example if $k = 2$, half the elements will round to the median and the other half will round to the maximum.

Analysis: bracket S by S^+ and S^- where

$$S^- = \{0, \ldots, 0, S_{\lceil \frac{n}{k} \rceil}, \ldots, S_{\lceil \frac{n}{k} \rceil}, \ldots, S_{(k-1)\lceil \frac{n}{k} \rceil}, \ldots, S_{(k-1)\lceil \frac{n}{k} \rceil}\}$$

i.e. drop the maximum segment from S^+, i.e. we round down.

Note that

$$OPT(S^-) \leq OPT(S) \leq OPT(S^+)$$

and

$$OPT(S^+) \leq OPT(S^-) + \frac{n}{k}$$

where we put each of the $\frac{n}{k}$ copies of S_n in S^+ in separate bins on top of our solution for S^-.

Therefore $OPT(S^+) \leq OPT(S) + \frac{n}{k}$.

Also $n\delta \leq \sum s_i \leq OPT(S)$ because $s_i \geq \delta$ for every i so $n \leq \frac{OPT(S)}{\delta}$.

Thus we have

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

if we choose $k = \frac{1}{\delta^2}$ then $OPT(S^+) \leq OPT(S)(1 + \delta)$, where case 1 solves $OPT(S^+)$ exactly.

Case 3 We specify two steps

A Use case 2 to pack all $s_i \geq \delta$ (choose δ later).

B Use First Fit to pack remaining s_i's ($< \delta$) into empty space left in all bins.

Goal: given $\epsilon > 0$ we want the algorithm to produce $\leq (1 + \epsilon)OPT + 1$.

Analysis: if all $s_i \geq \epsilon$ then use **A** with $\delta = \epsilon$ to get $(1 + \epsilon)OPT$. Otherwise apply **B**. Let m be the number of bins used and only the last bin has size $\leq 1 - \delta$ (First Fit would fill all other bins with $< \delta$ size elements).

Note that

$$\sum s_i \geq (m-1)(1-\delta)$$
$$\Rightarrow m \leq \frac{\sum s_i}{1-\delta} + 1$$
$$\Rightarrow m \leq \frac{OPT(S)}{1-\delta} + 1$$

Set $\frac{1}{1-\delta} = 1+\epsilon$ or $\delta = 1 - \frac{1}{1+\epsilon} = \frac{\epsilon}{1+\epsilon}$ to get our goal.
Then $m \leq (1+\epsilon)OPT(S) + 1$.

We solve for the runtime in terms of ϵ: note from before the dominant term (from case 1) is $O(n^{k^{\frac{1}{\delta}}})$.
We chose $k = \frac{1}{\delta^2}$ in case 2 where $\delta = \frac{\epsilon}{1+\epsilon}$ in case 3 so we have

$$O(n^{\left(\frac{1+\epsilon}{\epsilon}\right)^{2\left(\frac{1+\epsilon}{\epsilon}\right)}})$$

For $\epsilon = \frac{1}{2}$, we get $n^{91.1}$.

Improvementes: (Karmarkar and Karp 1982) there is an asymptotic $(1+\epsilon)$ approximation that runs in $O(\left(\frac{1}{\epsilon}\right)^8 n \log n)$ time which is an FPTAS.

Open question: is there an algorithm where we have $\leq OPT(S) + 1$ in polynomial time?

18 November 14, 2018

18.1 Knapsack problem

Given objects $1, \ldots, n$ each with size $s_i \in \mathbb{N}$ and **profit** $p_i \in \mathbb{N}$ (require integral profit values) and given a knapsack capacity B, find $K \subseteq \{1, \ldots, n\}$, $\sum_{i \in K} s_i \leq B$ and $\max \sum_{i \in K} p_i$.
Recall a pseudo-polynomial time algorithm for knapsack utilizing dynamic programming: For each profit sum, find the minimum size subset. The subproblem $S(i,p)$ is the minimum size subset of $\{1, \ldots, i\}$ of profit p.
The pseudocode is (where $P = n \max\{p_i\}$)

```
1  Initialize for all p = 1,...,P
2    S(1,p) = s_1 if p = p_1, infinity otherwise
3  For i = 2,...,n
4    For p = 1,...,P
5      S(i,p) = min( S(i-1, p - p_i) + s_i, S(i-1, p) )
6  Return max P such that S(n,p) <= B
```

The run-time $O(nP) = O(n^2 \max\{p_i\})$ which is pseudo polynomial time because the runtime depends on values of p_i (which is not a polynomial of their number of bits $\log p_i$).
Note the Knapsack problem (as a decision problem) is NP-complete. Partition reduces to Knapsack (let B be half the total sum of the partition elements, set profits to 1 for each element, see if any solution exactly fills up B capacity).

Remark 18.1. Some (a few) NP-complete problems have pseudo polynomial time algorithms. Most do not (unless $P = NP$) e.g. Travelling Salesman Problem (weights on edges). When weights are 1 then this is the Hamiltonian cycle.

From above we have two notions of "easier" NP-complete problems

1. Pseudo polynomial time algorithm (e.g. Knapsack)
2. FPTAS

They are in fact related.
For example, we can come up with an FPTAS algorithm for Knapsack. The idea is to round the p_i's to have fewer bits so that our approximation is within some ϵ.
The algorithm is as follows:

1. Given $\epsilon > 0$ (want approximation $\geq (1-\epsilon)OPT$) and given $s_1, \ldots, s_n, p_1, \ldots, p_n, B$ for all i

$$p_i' = \lfloor \frac{p_i}{t} \rfloor$$

   where we will derive and choose t based on ϵ.

2. Run dynamic programming algorithm on p_i''s (same s_i's and same B)

Suppose algorithm gives set $K(t) \subseteq \{1, \ldots, n\}$. Then $K(t)$ is **feasible** if $\sum_{i \in K(t)} s_i \leq B$. We must analyze $P(K(t)) = \sum_{i \in K(t)} p_i$ compared to $P(K^*) = OPT$ where K^* is the optimal solution.
We know that

$$p_i - t \overset{(**)}{<} tp_i' \overset{(*)}{\leq} p_i \qquad \text{properties of floors}$$

Now we want to relate our $P(K(t))$ to $P(K^*)$

$$\begin{aligned}
P(K(t)) &= \sum_{i \in K(t)} p_i \\
&\overset{(*)}{\geq} \sum_{i \in K(t)} tp_i' \\
&\geq \sum_{i \in K^*} tp_i' && \text{our DP algorithm must have given us the optimal } K(t) \text{ with max profit} \\
&\overset{(**)}{\geq} \sum_{i \in K^*} p_i - t \\
&= \sum_{i \in K^*} p_i - t|K^*| \\
&= OPT - t|K^*| \\
&\geq OPT - tn \\
&= OPT(1 - \frac{tn}{OPT})
\end{aligned}$$

We want $\geq OPT(1-\epsilon)$ so set $\epsilon = \frac{tn}{OPT}$. Note that $OPT \geq \max_i p_i$ (this assuems $s_i \leq B$ for all i, otherwise we discard that item since it will never fit).
Thus $OPT(1 - \frac{tn}{OPT}) \geq OPT(1 - \frac{tn}{\max\{p_i\}})$ thus we choose t such that

$$t = \epsilon \frac{\max\{p_i\}}{n}$$

Note from before our run time is $O(n^2 \max\{p_i'\})$, and since

$$p_i' = \lfloor \frac{p_i}{t} \rfloor = \lfloor \frac{p_i n}{\epsilon \max\{p_i\}} \rfloor \leq \frac{n}{\epsilon}$$

so the runtime is $O(n^3 \frac{1}{\epsilon})$ thus we have an FPTAS algorithm.

18.2 Pseudo polynomial time and FPTAS

Question 18.1. Can every pseudo-polynomial time algorithm be converted into an FPTAS?
This is an **open question**.

The converse holds:

Theorem 18.1 (Garey and Johnson 1978)**.** If a problem has an FPTAS (and some other technical assumptions) then there is a pseudo polynomial time algorithm.

Proof. Idea: suppose we have a minimization problem and technical assumptions:

1. objective function is integer-valued
2. bound $OPT < q(|I_{unary}|)$ i.e. the optimal solution is bounded by some polynomial q of the unary representation of the input (the unary representation with length n of a number as opposed to binary where we have the # of bits or $\log n$)

Let A be an FPTAS with run time poly$(|I|, \frac{1}{\epsilon})$ i.e. $A(I, \epsilon) \leq (1+\epsilon)OPT(I)$.
Idea: pick ϵ so small we get the true OPT solution, then we analyze its run time.
Set $\epsilon = \frac{1}{q(|I_{unary}|)}$, then

$$A(I, \epsilon) \leq (1 + \frac{1}{q(|I_{unary}|)})OPT < OPT + 1$$

So $A(I, \epsilon) = OPT$ since our objective value is integer valued.
Note the runtime is poly$(|I|, \frac{1}{\epsilon})$ = poly$(|I|, q(|I_{unary}|))$ which is polynomial in terms of the unary representation i.e. pseudo polynomial time. □

19 November 19, 2018

19.1 NP-complete and approximation factors

The current status of NP-complete problems with respect to approximation, listed from harder to easier:

- Set Cover: $(\log n)$-approximation
- Vertex Cover: constant-factor
- Bin packing, packing squares: PTAS
- Knapsack problem: FPTAS

Examples we have seen:

Lemma 19.1. A 2-approximation for general TSP results in $P = NP$.

Proof. Assume we have an algorithm polynomial time that retains 2-approximation for TSP.
Use it to solve Hamiltonian cycle in polynomial time given $G = (V, E)$ input to Ham cycle.
Construct $G' = (V, E')$ where E' is all possible edges where $w(e) = 1$ if $e \in E$ and $w(e) = n + 2$ if $e \notin E$ i.e. $e \in E' \setminus E$.
By assumption we can find T a TSP tour of G' where $w(T) \leq 2OPT$ (OPT is the optimal TSP tour of G').
We observe there to be a gap between our solutions for a TSP tour T: either $w(t) = n$ if $T \subseteq E$ or $w(T) \geq (n-1) + n + 2 > 2n$ if $T \not\subseteq E$ (since T uses at least one edge of weight $n + 2$).

Claim. G has Hamiltonian cycle iff $w(T) = n$.

Proof. Backwards direction is easy.
Forwards: Assume G has Ham cycle. Then $OPT = n$. so $w(T) \leq 2OPT \leq 2n$ thus $w(T) = n$ by our gap. □

□

Other examples of negative results:

- Polynomial-time k-approximation for Independent Set gives us k-approximation for Clique
- Constant factor approximation for Clique gives us PTAS for Clique
- **Theorem 19.1.** PTAS for independent set gives us PTAS for Max 3-SAT

  Proof. Use standard reduction: 3-SAT reduces to Independent Set. See notes. □

19.2 Max 3-SAT

Given 3-SAT formula find the truth-value assignment that satisfies a maximum number of clauses.

Theorem 19.2 (1992)**.** A PTAS for Max 3-SAT implies $P = NP$.

Proof. Exercise. Look at implications for Independent Set and Clique. □

19.3 Interactive proof systems

We need a new chracterization of NP to handle difficult certificates. Recall NP for decision problems mean the problem can be **verified** in polynomial time given a **certificate** of polynomial size (e.g. Hamiltonian cycle, Max 3-SAT).
Instead we think of this gs a game between

Prover P all powerful, finds certificate

Verifier V computationally limited, runs in polynomial time, checks certificate

we generalize our notion of NP such that the verifier V can use randomness and can interact with prover P. This is called an **interactive proof system**.

Example 19.1 (Graph isomorphism)**.** Given 2 graphs, to verify for graph isomorphism, can you re-label one graph to get the same graphs?

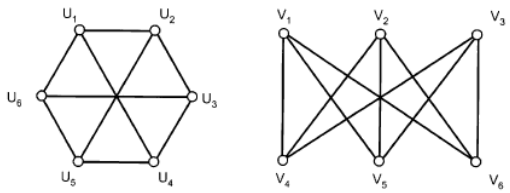


Figure 19.1: Two isomorphic graphs where we can have an isomorphic (1-to-1/onto) mapping between vertices (and edges).

Open question: is graph isomorphism in NP? In P? In NP-complete? In Co-NP (given two graphs can you verify if $G_1 \not\approx G_2$ i.e. not isomorphic in polynomial time)?
We will show an interactive proof protocol to verify $G_1 \not\approx G_2$:

Verifier
- Pick either G_1 or G_2 at random
- Randomly re-label the picked graph
- Ask prover: was it G_1 or G_2?
  If $G_1 \not\approx G_2$, then prover will always answer correctly.
  If $G_1 \approx G_2$, then prover will answer correctly 50% of the time.

The verifier runs many trials to verify (with high probability) that $G_1 \not\approx G_2$.

More restricted proof systems do not use multiple rounds.

19.4 Probabilistically checkable proofs

Given statement e.g. G has a Ham cycle:

- The Prover P writes a "proof"
- The Verifier V is a randomized algorithm that "checks" the proof and outputs YES or NO

Conditions for correctness of a "probabilitistically checkable proof":

1. If statement is TRUE, there must exist a "proof" that makes V answer YES always
2. If statement is FALSE, then V answers NO with probability $\geq \frac{3}{4}$ regardless of the "proof" given

Some limits we place on V:

- Polynomial time
- For input size of n: $O(f(n))$ random bits used and $O(g(n))$ bits of the proof it can look at

We denote $PCP[f, g]$ as the class of decision problems with probabilitistically checkable proof where our limits are satisfied with the corresponding f and g.
For example, $NP = PCP[0, \text{poly}(n)]$ and $P = PCP[0, 0]$ (polynomial time with no prover so no proof to check).

Theorem 19.3 (PCP theorem)**.** We claim $NP = PCP[\log n, 1]$.
That is: V looks at only $O(1)$ bits of the proof and V uses *random bits* (of $O(\log n)$) to choose where to look at the proof (random bits as addresses into proof string).

Proof. The easy direction is showing $PCP[\log n, 1] \subseteq NP = PCP[0, \text{poly}(n)]$.
To do this, we eliminate randomness by checking all possible random strings of the proof which have $O(\log n) \leq k \log n$ bits.
We thus have $2^{k \log n} = n^k$ strings to look at. The verifier looks at all these n^k strings of the proof thus we have $PCP[0, \text{poly}(n)]$.
The other direction is much harder to show. □

Claim. PCP theorem implies that a PTAS for Max 3-SAT implies $P = NP$.

Proof. Use $NP = PCP[\log n, 1]$.
Idea: take any problem in NP and make a polynomial time algorithm for it using approximation algorithm for Max 3-SAT.
Consider the Verifer's $PCP[\log n, 1]$ algorithm. It depends on

- $x = x_1 \dots x_n$: the bits of the input
- $y = y_1 \dots y_t$: the bits of "proof" given by Prover where $t \in O(\text{poly}(n))$
- $r = r_1 \dots r_k$: the random bits where $k \in O(\log n)$

Note that any algortihm can be turned into a Boolean 3-SAT formula (as in first NP-completness proof).
Let $F(x, y, r)$ be the formula for a given x, y, r that captures the Verifier's algorithm.
Let $F(x, y) = \bigwedge_{\text{all choices for } r} F(x, y, r)$ which is polynomial sized since we have a polynomial number of random strings r.
If x is a YES input, then $\exists y$ such that all $F(x, y, r)$ is satisfied and thus $F(x, y)$ is satisfied.
If x is a NO input, then at most $\frac{1}{4}$ of the $F(x, y, r)$ are satisfied i.e. at most $\frac{1}{4}$ of the clauses of $F(x, y)$ are satisfied.
This gives us a "gap" ("all" or $\frac{1}{4}$ clauses are satisfied) that we can distinguish with a good approximation algorithm for MAX 3-SAT i.e. it'll tell us whether x is a YES or NO with just an approximation algorithm for Max 3-SAT in polynomial time, so $P = NP$. □

20 November 21, 2018

20.1 Online algorithms

Suppose we are given a sequence of requests: an **online algorithm** must handle each request as it comes. An **offline algorithm** on the other hand gets to look at the whole sequence beforehand.
Some examples of online algorithms:

- Linked list accessing (Move to Front heuristic)
- Paging (LRU, LFU)
- Splay trees (dynamic optimality conjecture)
- Bin packing (first fit, best fit)

A real world example: suppose you are renting skis: it is \$30 to rent or \$300 to buy. In hindsight (i.e. offline) if the number of times you ski is ≤ 10 then it is more worthwhile to rent. Otherwise it is more worthwhile to buy the skis.
An online algorithm would be to rent 10 times, then purchase the ski.
We claim this is bounded by a factor of 2 of OPT: if the # of times is ≤ 10, then our algorithm gives us the OPT solution. If the # of times is > 10, then we pay $2\times$ OPT.

20.2 Competitive analysis

The goal of competitive analysis is to compare an online algorithm with the optimal offline solution (even if OPT is hard to find).

Definition 20.1 (Competitive ratio)**.** Algorithm A is c-competitive if $\exists$ constant b such that (for a minimization problem)

$$A(\sigma) \leq cOPT(\sigma) + b$$

where σ is the sequence of events and for a maximization problem

$$A(\sigma) \geq cOPT(\sigma) - b$$

Note we always allow additive term b (unlike approximation algorithms).

20.3 Paging

We have fast memory (the cache which holds k pages) and slow memory.
When a page is requested, if it is in the cache then our cost is de facto 0. Otherwise we have a **page fault** where we must bring the page into cache.
We must **evict** some page from cache to swap in our page with a cost of 1. Which page should we evict?
The goal is to minimize the overall cost in an online setting.
The optimal offline strategy is to evict the page whose next request is the furthest in the future.
For example if we had access

$$ABCBDCEABE$$

where $k = 3$, we have ABC in our cache up until D when we evict B for D (cache ADC), then we evict C for E (cache ADE), and then we evict A for B (cache BDE) for a total cost of 3.
It is difficult to prove that this is optimal.
Some online strategies include:

1. FIFO
2. LRU: least recently used.

   For example if we had access

   $$ABCBDCEABE$$

   where $k = 3$, we have ABC in our cache up until D when we evict A (least recently used) for D (cache DBC), then we evict B for E (cache DEC), then we evict D for A (cache AEC), and then we evict C for B (cache AEB), for a total cost of 4.
3. LFU: least frequently used

Theorem 20.1 (Sleator and Tarjan 1985)**.** LRU and FIFO have competitive ratio k, but LRU is better in practice.

Proof. Divide request sequence into **phases**. A phase stops just before we see $k + 1$ different pages.
For example, given access sequence

$$ABACBACADADEDB$$

for $k = 3$ we have 3 phases: phase 1 is $ABACBACA$ (all uses ABC), phase 2 is $DADED$ (all uses ADE) and phase 3 is B.
Both LRU and FIFO cost $\leq k$ per phase because once we put a page of phase i into the cache, we won't evict it during phase i.
Note that OPT must evict ≥ 1 request for a given phase and the first request of the next phase because we would request $k + 1$ different pages.
So $\frac{ALG}{OPT} \leq k +$ additive term for last phase. □

Theorem 20.2. Any online deterministic algorithm has competitive ratio $\geq k$.

Proof. We use an adversary argument here. Let k be the cache size and we use $k + 1$ pages.
The adversary would always ask for the page not in the current cache. The algorithm costs n the length of the sequence (assuming the cache is initially populated).

Claim. An offline solution that evicts the page with the maximum next request time costs $\frac{n}{k}$.

Proof. Note that each time the offline algorithm evicts, we are good for the next k requests (since we evict the page used furthest in the future so we must be at least k pages until then). □

Note that $OPT \leq \frac{n}{k}$ our offline algorithm so we have

$$\frac{ALG}{OPT} \geq k$$

□

A **randomized** online "marking" algorithm for paging: to serve a request for page p:

```
1  If p is not in the cache:
2    If all pages in cache are marked:
3      Unmark them
4    Choose a random unmarked page to evict and move p in
5  Mark p
```

Theorem 20.3. The expected competitive ratio of our randomized algorithm is $\theta(\log k)$.

Proof. As before, divide into phases. In phase i, k different pages are requested.
At the beginning on a phase, all pages in the cache are unmmarked.
Let S_i be the pages in the cache at the start of phase i.
A request for page p in phase i is **old** if p is in S_i (the pages at the *start of the phase*); otherwise it is **new**.

Example 20.1. For example, suppose we have a cache with size $k = 4$. At the start of phase i, we have $ABCD$ (all unmarked) in our cache i.e. $S_i = \{A, B, C, D\}$.
Suppose we get a request for E (new): we randomly choose an unmarked page to evict e.g. C then swap E in and mark it, incurring cost of 1 (cache $ABED$).
Suppose we get a request for B (old): it's already in the cache so we do nothing and mark B (cache $ABED$).
Suppose we get a request for F (new): we randomly choose an ummarked page (either A or D), e.g. we choose A to swap out with F. We then mark F, incurring cost of 1 (cache $FBED$).
Suppose we then get a request for C (old, was in S_i initially). Since C is not in our current cache we need to swap it out with an unmmarked page (only D) in this case incurring a cost of 1.

Let n_i be the # of *new* requests in phase i. New requests are always cost 1 (need to swap something out from our S_i).
What is the expected cost of *old* requests? Note we incur a cost of 1 if we are unlucky and we evicted a page of S_i and then requested it later in the phase, e.g. the request for C in our example.
Consider our *first old request*, say for page p. Before p there have been $\leq n_i$ new requests i.e. $\leq n_i$ pages evicted (at random). Therefore

$$P(p \text{ was evicted}) \leq \frac{n_i}{k}$$

More generally let $p_1, p_2, \ldots$ be distinct old page requests (in order). When p_{j+1} is requested, there are $k - j$ yet un-requested pages from S_i.
We have marked $j + (\leq n_i)$ pages (j old pages and $\leq n_i$ new pages). Therefore the number of unmarked pages in the cache is $\geq k - j - n_i$ so

$$P(p_{j+1} \text{ in cache}) \geq \frac{k - j - n_i}{k - j}$$

therefore

$$P(p_{j+1} \text{ was evicted}) \leq \frac{n_i}{k-j}$$

Summing over j from $j = 0, \ldots, k - n_i - 1$ then the expected cost of old requests is

$$\leq n_i(\frac{1}{k} + \frac{1}{k-1} + \ldots + \frac{1}{k-(k-n_i-1)})$$

$$\leq n_i(H_k - 1) \qquad H_k = 1 + \frac{1}{2} + \ldots + \frac{1}{k} \text{ kth Harmonic number}$$

adding in our n_i new requests, we have an expected cost $n_i \cdot H_k$.

Claim. OPT costs $\geq \frac{1}{2}\sum n_i$. Proof in notes.

So the competitive ratio is

$$H_k \sum n_i \leq 2H_k OPT = \theta(\log k)OPT$$

as desired. □

21 November 26, 2018

21.1 k-server problem

We have k servers to service requests in a *metric space* of points $p_1, \ldots, p_t$. If there is a request for p_i:

- if a server is already at p_i: this is fine
- otherwise move a server from its location, say p_j, to p_i at cost $d(p_j, p_i)$ (distance)

The goal is to server all requests and minimize sum of cost/distances.
Paging is a special case: "points" are all apges in slow memory and "serving" a request is putting a page in the cache. The k servers correspond to the cache of size k where distances are all 1.
A local greedy algorithm(for general k-server): to meet the next request, say at p_i, move server from p_j that minimizes $d(p_j, p_i)$ i.e. use the closest server.

Claim. This algorithm is not c-competitive for any constant c.

Proof. Consider the counterexample where we have p_1, p_2 and q where $d(p_1, p_2) < d(p_2, q) < d(p_1, q)$. Initially our servers are at p_2 and q. If a stream of requests come in for $p_1, p_2, p_1, p_2, \ldots$ then we will incur $nd(p_1, p_2)$ cost for n potentially infinite requests.
However the optimal solution is to relocate q to p_1 initially to only incur $d(p_1, q)$ total cost. □

Conjecture 21.1. (open since 1988): there is k-competitive algorithm for k-server problem.

Some special cases with k-competitive algorithms:

- uniform distances (e.g. paging: use LRU)

  Exercise 21.1. Make k-server example (not necessarily with uniform distances) where LRU is not k-competitive.

- points on a line with distance on the line (i.e. $p_i \in \mathbb{R}$)

For the special case where $p_i \in \mathbb{R}$: we have the **double coverage algorithm**: we have two cases:

1. If a request is to the right (or left) of all servers, move closest server
2. If a request is in between 2 servers q and r, move **both** q and r towards the request. Stop when one reaches the request.

Note that this algorithm works well for our previous example with p_1, p_2, q. Also note that we can use "phantom" positions: we do not actually move the 2nd server to decide which server (so we do not incur the cost until that server actually services a request).
Proof that this algorithm is k-competitive:

Proof. Idea: use amortized analysis with potential function Φ. Think of ALG having k servers and OPT having k servers. Φ measures the "difference".
Analyze change in Φ as: first OPT serves request then ALG servers request.

Aside. We won't use this but there is a polynomial time algorithm to find OPT: uses dynamic programming.

We will ensure the following properties of Φ:

1. $\Phi_i \geq 0$
2. If ith move of OPT costs s_i, then increase in Φ is $\leq ks_i$
3. If ith move of ALG cost t_i, then increase in Φ is $\leq -t_i$ i.e. Φ decreases by t_i

Suppose we have the above guarantees, then

$$\Phi_{i+1} - \Phi_i \leq ks_i - t_i$$

Summing up the telescoping series

$$\Phi_f - \Phi_0 \leq k \sum s_i - \sum t_i$$

where $\sum s_i$ is the cost of OPT and $\sum t_i$ is the cost of ALG, so

$$ALG \leq kOPT - \Phi_f + \Phi_0 \leq kOPT + \Phi_0$$

What is Φ? At any time step i we let

$$\Phi_i = kM + D$$

where

D sum of all $\binom{k}{2}$ distances between pairs of ALG's servers

M minimum weight matching between OPT's servers and ALG's servers

For example, when $k = 1$, we have only 1 server each for ALG and OPT separated by some distance d so we have one unique matching $M = d$.
When $k = 2$, we have two servers for each of ALG (1 and 2) and OPT ($1'$ and $2'$). A minimum matching might be $d(1, 1') + d(2, 2')$ (but there exists another matching $d(1, 2') + d(2, 1')$).
We verify our three properties for Φ hold with our definition

- $\Phi_i \geq 0$: always holds since $M, D \geq 0$
- Suppose OPT moves a server distance s_i. D does not change and M increases by $\leq s_i$
- Suppose ALG moves with cost t_i.

Case 1: far right or left Suppose a new request comes at the far right (or left): suppose OPT has already put a server at the new request. Then our matching from the corresponding OPT server to the corresponding ALG server (both at the new request) have 0 distance betwen them: our matching decreases by t_i.

So *decrease* of Φ is $\geq kt_i$.

What about D? Distance from our moved ALG server to all other $k-1$ servers increases by t_i, so the net change in Φ is a decrease of $\geq t_i$.

Case 2: in between two servers Suppose the two servers q and r each move by distance d (although one of them is a phantom move, we must account for it). Let $t_i = 2d$. We must show Φ decreases by $\geq 2d = t_i$.

We first show that M does not increase (suppose a OPT server is already at request: the server that reaches the request point forms a matching with the OPT server (so change decrease in M is t_i), and the other phantom server can increase M by at most t_i) and then show D decreases by $2d$ because $d(q, r)$ decreases by $2d$ (hint: separate vertices into left and right side of request point).

□

22 November 28, 2018

22.1 Fixed parameter tractable algorithms

NP-Complete problems seem to have exponential-time algorithms: but how exponential are they?
For example, pseudo-polynomial time algorithms are exponential in terms of the input's number of bits.
Try other "parameters": small parameters may result in polynomial time.
Example: Vertex Cover. Given graph G, $k \in \mathbb{N}$. Is there a vertex cover of size $\leq k$?
Try k as a parameter: we try all subsets of size k i.e. all $\binom{n}{k}$ subsets which is $O(n^k)$. Since verifying if a subset of size k is a vertex cover takes $O(nk)$ we the overall runtime is therefore $O(kn^{k+1})$.
The same idea works for the Clique and Indepenedent Set problems. But note that for some problems e.g. the k-colouring vertices is still NP-Complete for $k = 3$.
We observe that $O(n^k)$ is still really bad: idealy we want $O(f(k)n^c)$ or even better $O(f(k) + n^c)$ for some constant c and $f(k)$ which is independent of n for a tractable fixed parameter.
We can achieve this for Vertex Cover (but not Independent Set). The algorithm branches on all choices of vertices for each edge (u, v): that is either u or v (or both) must be in the Vertex Cover.

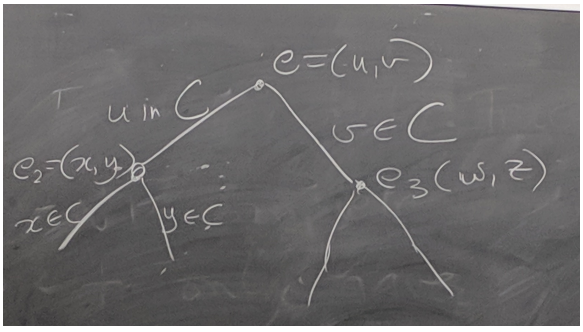


Figure 22.1: We branch on each choice of either vertex u or v for an edge $e = (u, v)$, building a binary tree.

The resulting binary tree where we fix some $e = (u, v)$ to the root of the tree has height k with 2^k leaves. The algorithm is:

```
1  VC2(G, k):
2    if E = {} return True
3    if k = 0 return False
4    Pick e = (u,v) in E
5    Return VC2(G \ u, k-1) or VC2(G \ v, k-1)      // remove u (or v) and all ↩
       incident edges from G, respectively
```

Exercise 22.1. Try the above algorithm with a small example and find a Vertex Cover.

The runtime for the above algorithm is $O(2^k n)$ where we have 2^k internal tree nodes and we do $O(n)$ at each node. Next we'd like to improve this to $O(f(k) + n^c)$ using a technique called **kernelization**.

Claim. If $deg(v) > k$ then v must be in any vertex cover C with $|C| \leq k$.

Proof. If $v \notin C$ then all neighbours ($> k$) must be in C to cover the edges incident to v. □

A better algorithm using the above claim:

```
1  VC3(G, k):
2    C' = all vertices of degree > k
3    k' = k - |C'|
4    G' = G \ (C' and all incident edges)      // also remove isolated vertices (↩
       degree 0)
5    if G' has > 2k^2 vertices:
6      return False
7    else:
8      return VC2(G', k')
```

Claim. Claim about size of G': since all vertices in G' have degree $\leq k$, if G' has vertex cover $\leq k$ then G' must have $\leq k^2$ edges.
Therefore G' has $\leq 2k^2$ vertices (since every edge can have at most two distinct vertices).

Rune time analysis: note we invoke $VC2$ on G' and k', thus we have $O(2^{k'}2k^2) \leq O(2^{k+1}k^2)$ since we have $\leq 2k^2$ vertices in G'.
Finding C' and G' takes $O(n+m)$ (where m is the number of edges) thus we have a total runtime of $O(n+m+2^{k+1}k^2)$.
This idea of "kernelization" is by Jonathan Buss (CS faculty). The above algorithm is *sort of* practical: for $n = 10^4$ and $k = 10$

Brute force $kn^{k+1} = 10^{45}$

VC2 $2^k n = 10^7$

VC3 $2^{k+1}k^2 + n + m = 2 \cdot 10^5 + m$

Definition 22.1 (Fixed parameter tractable (FPT))**.** A problem is **fixed parameter tractable (FPT)** in parameter k if it has an algorithm with runtime $O(f(k)n^c)$ where n is the input size, $f(k)$ is a function of only k, and c is a constant (independent of k).

Some examples of parameters:

1. Value of OPT: e.g. Vertex Cover or Independent Set of size k
2. Max degree in graphs
3. Dimension of space for geometric problems

4. Genus of graph (where planar graphs have genus 0 graphs (no crossing edges on plane), graphs with no crossing edges on a torus have genus 1, genus 2 for torus with two holes, etc.).

Some examples of FPT problems:

1. Vertex cover of size k
2. Simple path of length k
3. Finding k disjoint triangles in a graph
4. Drawing a graph in plane with k edges crossing

Hardness: some parameterized problems have no FPT algorithm unless $P = NP$ e.g. Independent Set of size k.

Theorem 22.1 (Kernelization)**.** If a problem is FPT then there exists an algorithm with runtime $O(f'(k) + n^{c'})$.

Remark 22.1. Proof is non-constructive i.e. it does not show us how to find such an algorithm for an arbitrary FPT problem.

22.2 Simple path of length k

We define a simple path has a path with no repeated vertices. Note a shortest path is always simple.
The problem is to find a simple path from vertex s to t of exactly length k (i.e. exactly k edges).

Exercise 22.2. Show this is NP-hard: reduce Hamiltonian path to this with $k = n - 1$.

A randomized FPT algorithm:

```
1  Randomly colour vertices with k colours (allow same colour at both ends of an edge↩
       )
2  Look for a colourful s-t path       // each colour appears exactly once on the ↩
       path
```

Note that

Correctness

Probability of error If there is no simple s-t path of length k, then we will never find a colourful s-t path regardless of colouring so algorithm answers NO.

Suppose there exists a simple s-t path P of length $k-1$ (therefore k vertices from s to t including s and t, makes math easier), then

$$P(P \text{ is colourful}) = \frac{k!}{k^k}$$

where $k!$ is the number of permutations such that our path is colourful and k^k are all possible colourings of the k vertices with k colours.

By Sterling's approximation $k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}$ thus

$$P(P \text{ is colourful}) \geq \frac{\left(\frac{k}{e}\right)^k}{k^k} = \frac{1}{e^k}$$

therefore the algorithm outputs YES with probability $\geq \frac{1}{e^k}$.

Claim. If the probability of success is p, then the probability of failure after $\frac{1}{p}$ trials is

$$\leq (1-p)^{\frac{1}{p}} < (e^{-p})^{\frac{1}{p}} = \frac{1}{e}$$

since $e^{-p} = 1 - p + \frac{p^2}{2} - \ldots$ by Taylor expansion.

So we perform e^k trials such that the probability of failure is $\leq \frac{1}{e}$

Runtime We want the runtime for checking if there is a colourful path.

Basic approach: try all $k!$ orders of colours starting from s. The runtime is $O(k!m)$.

Fact 22.1. We can indeed improve to $O(2^k m)$.

So we have an FPT algorithm with error probability $\leq \frac{1}{e}$ and runtime $O(e^k 2^k m)$.

Fact 22.2. We can actually de-randomize this algorithm.

23 December 3, 2018

23.1 FPT for general Independent Set

Recall Fixed Parameter Tractable algorithms can be expressed in runtime as $O(f(k)n^c)$ for some constant c independent of k for some parameter k.
For the Independent Set problem (given graph G, does it have an independent set of size $\geq k$), the brute force method takes $O(n^k(n+m))$ we check every possible subset of vertices of size k (out of n vertices). Note this is not FPT.
In fact, the IS problem is not FPT: no one has found an FPT algorithm for IS (for the specific size k parameter). Open: would such an FPT algorithm imply $P = NP$?
Recall for Vertex Cover where k is the size of the minimum Vertex Cover we do have FPT algorithms: $O(2^k n)$ using the branching algorithm and $O(2^k k^2 + m + n)$ using kernelization.

23.2 FPT for tree-like graph Independent Set

Suppose we want to find an independent set on a tree with weights on vertices $w(v)$ (and we want to maximize the total weight). Do we start at the leaves? We can't simply just take levels.
A dynamic programming algorithm: where given a child tree rooted at some vertex v, we have two subproblems: $IS(v)$ is the maximum weight of the independent set rooted at v; $IS^o(v)$ is the maximum weight of IS that *does not include* v.
Our algorithm follows:

```
1   Initialize for every leaf v
2     IS^o(v) = 0
3     IS(v) = w(v)
4   For all nodes v in leaf-root order
5     // v has children u_1,..., u_t
6     IS^o(v) = \sum_{i=1}^t IS(u_i)
7     IS(v) = max{w(v) + \sum_{i=1}^t IS^o(u_i), IS^o(v)}
8   Return IS(root)
```

We can use the same idea for graphs that are "close to" trees.
We require the idea of **Series-parallel graphs (SP graphs)** which is defined recursively: it is a graph with two terminal nodes s and t that satisfy one of the following patterns:

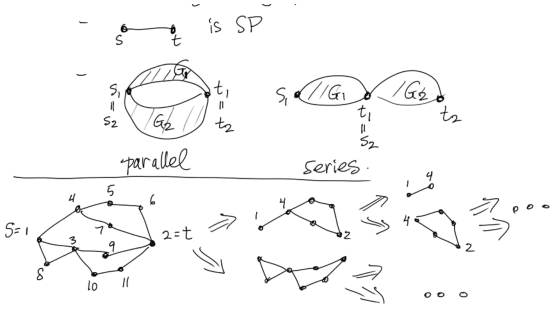


Figure 23.1: Top: definition of Series-parallel (SP) graphs. Bottom: example of deconstruction an SP graph into its subgraphs.

Base case s connected to t by one edge

Parallel $s_1 = t_1$ and $s_2 = t_2$ connected in parallel by two SP graphs G_1, G_2

Series s_1 connected to $s_2 = t_1$ connected to t_2 in series by two SP graphs G_1, G_2

IS for SP graphs: we can do dynamic programming based on the following subproblems:

- max IS including s and t
- max IS including s, not t
- max IS including not s, t
- max IS including not s, not t

We can actually transform the IS problem for SP graphs into an IS problem for trees:

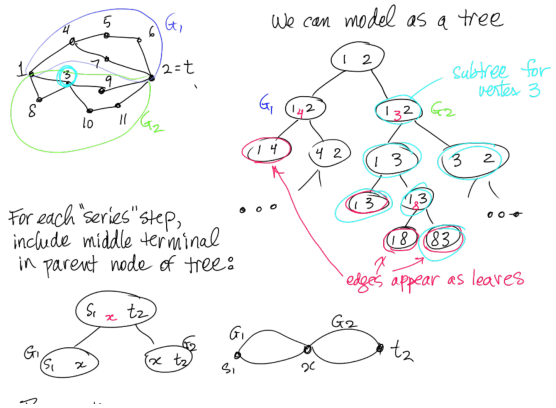


Figure 23.2: Transformation of an SP graph to a tree structure for the Independent Set problem.

- A subtree for graph G is rooted at a node (s, t) where s and t are the two terminal nodes of the entire SP graph. If G is a series configuration, include the middle terminal node x in the root i.e. (s, x, t)

- The left and right subtree corresponds to G_1 and G_2 in both the parallel and series cases
- Leaves correspond to individual edges (this is the base case of our SP graph definition)

We note a few additional properties of our tree transformation:

1. If $e = (u, v)$ is an edge of G then u and v appear together in a tree node, in fact leaves (see diagram for red nodes)
2. Every vertex v of G corresponds to a subtree of T (see diagram for vertex 3 corresponding to the green subtree)

There is a generalization of this tree transformation (Robertson & Seymour) that introduces the idea of **tree-widths**: we can arbitrarily represent a graph G as a tree T whereby the following conditions are met:

- for every vertex v of G, the "bags" (a node) containing it form a subtree
- for every edge $e = (u, v)$ of G, there is a bag containing u and v

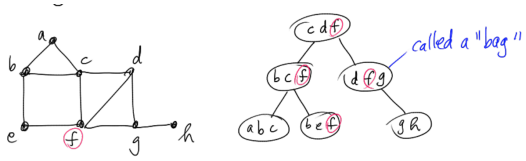


Figure 23.3: Generalization of SP graphs: any graph G can be transformed into some tree representation T.

The **width of the decomposition** is the (size of the largest bag) -1. The **tree-width** of a graph is the *minimum* width of any tree decomposition ($\leq n - 1$; a single bag).
Note that graphs of tree-width 1 correspond to forests (subgraphs of tree), and graphs of tree-width 2 correspond to subgraphs of SP graphs.

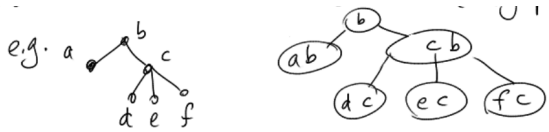


Figure 23.4: Graphs of tree-width 1 correspond to forests or subgraphs of trees.

Theorem 23.1. The maximum weight Independent Set in a graph of tree-width k can be found in time $O(2^k n)$.
Idea: use dynamic programming. For each bag B (size $\leq k + 1$), we find for each subset $A \subseteq B$ (there are $O(2^k)$ of them) the maximum weight Independent Set in subtree rooted at B that includes A and excludes $B \setminus A$.

How do we find the tree-width of a graph? This is NP-hard but FPT.

Theorem 23.2. There is an algorithm with running time $O(2^{O(k^3)} \cdot n)$.

There are more efficient algorithms if we are okay with *approximate* tree-width.
Note that many problems are FPT in terms of tree-width:

- 3-colouring
- min-colouring
- Hamiltonian cycle

23.3 Hardness of FPT algorithms

To prove hardness for FPT algorithms we must use reductions that *preserve* fixed parameter tractibility. We can then prove results like: an FPT algorithm for problem X implies an FPT algorithm for problem Y (which we can specify some $W[i]$ class of equivalently hard FPT algorithms).
Recall that PCP (probabilistically-checkable proofs) theorem gave us that if a PTAS exists for certain problems (e.g. Max 3-SAT) then $P = NP$.
Open: we have no such strong results for FPT.