Lecture 1: Introduction

6.006 pre-requisite:

- Data structures such as heaps, trees, graphs
- Algorithms for sorting, shortest paths, graph search, dynamic programming

Course Overview

This course covers several modules:

1. Divide and Conquer - FFT, Randomized algorithms
2. Optimization - greedy and dynamic programming
3. Network Flow
4. Intractibility (and dealing with it)
5. Linear programming
6. Sublinear algorithms, approximation algorithms
7. Advanced topics

Theme of today's lecture

Very similar problems can have very different complexity. Recall:

- **P**: class of problems solvable in polynomial time. $O(n^k)$ for some constant k.

  Shortest paths in a graph can be found in $O(V^2)$ for example.

- **NP**: class of problems verifiable in polynomial time.

  Hamiltonian cycle in a directed graph $G(V, E)$ is a simple cycle that contains each vertex in V.

  Determining whether a graph has a hamiltonian cycle is NP-complete but verifying that a cycle is hamiltonian is easy.

- **NP-complete**: problem is in NP and is as hard as any problem in NP.

  If any NPC problem can be solved in polynomial time, then every problem in NP has a polynomial time solution.

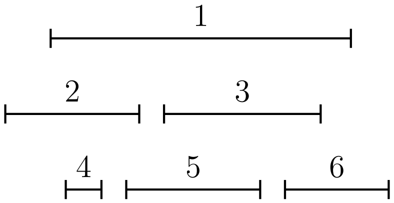
Interval Scheduling

Requests $1, 2, \ldots, n$, single resource

$s(i)$ start time, $f(i)$ finish time, $s(i) < f(i)$ (start time must be less than finish time for a request)

Two requests i and j are compatible if they don't overlap, i.e., $f(i) \leq s(j)$ or $f(j) \leq s(i)$.

In the figure below, requests 2 and 3 are compatible, and requests 4, 5 and 6 are compatible as well, but requests 2 and 4 are not compatible.



Goal: Select a compatible subset of requests of maximum size.

Claim: We can solve this using a greedy algorithm.

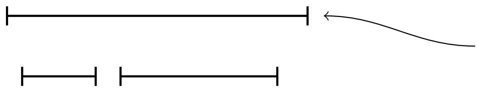
A greedy algorithm is a myopic algorithm that processes the input one piece at a time with no apparent look ahead.

Greedy Interval Scheduling

1. Use a simple rule to select a request i.
2. Reject all requests incompatible with i.
3. Repeat until all requests are processed.

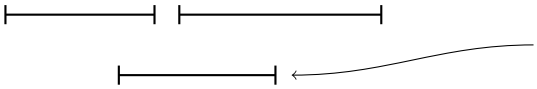
Possible rules?

1. Select request that starts earliest, i.e., minimum $s(i)$.



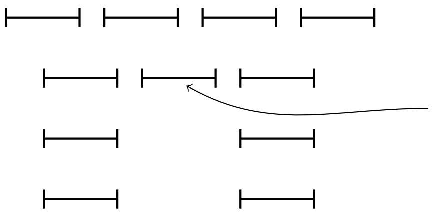
Long one is earliest. Bad :(

2. Select request that is smallest, i.e., minimum $f(i) - s(i)$.



Smallest. Bad :(

3. For each request, find number of incompatibles, and select request with minimum such number.



Least # of incompatibles. Bad :(

4. Select request with earliest finish time, i.e., minimum $f(i)$.

Claim 1. *Greedy algorithm outputs a list of intervals*

$$< s(i_1), f(i_1) >, < s(i_2), f(i_2) >, \ldots, < s(i_k), f(i_k) >$$

such that

$$s(i_1) < f(i_1) \leq s(i_2) < f(i_2) \leq \ldots \leq s(i_k) < f(i_k)$$

Proof. Simple proof by contradiction – if $f(i_j) > s(i_{j+1})$, interval j and $j+1$ intersect, which is a contradiction of Step 2 of the algorithm! □

Claim 2. *Given list of intervals L, greedy algorithm with earliest finish time produces k^* intervals, where k^* is optimal.*

Proof. Induction on k^*.

Base case: $k^* = 1$ – this case is easy, any interval works.

Inductive step: Suppose claim holds for k^* and we are given a list of intervals whose optimal schedule has $k^* + 1$ intervals, namely

$$S^*[1, 2, \ldots, k^* + 1] =< s(j_1), f(j_1) >, \ldots, < s(j_{k^*+1}), f(j_{k^*+1}) >$$

Say for some generic k, the greedy algorithm gives a list of intervals

$$S[1,2,\dots,k] =< s(i_1), f(i_1) >, \dots, < s(i_k), f(i_k) >$$

By construction, we know that $f(i_1) \leq f(j_1)$, since the greedy algorithm picks the earliest finish time.

Now we can create a schedule

$$S^{**} =< s(i_1), f(i_1) >, < s(j_2), f(j_2) >, \dots, < s(j_{k^*+1}), f(j_{k^*+1}) >$$

since the interval $< s(i_1), f(i_1) >$ does not overlap with the interval $< s(j_2), f(j_2) >$ and all intervals that come after that. Note that since the length of S^{**} is $k^* + 1$, this schedule is also optimal.

Now we proceed to define L' as the set of intervals with $s(i) \geq f(i_1)$.

Since S^{**} is optimal for L, $S^{**}[2,3,\dots,k^*+1]$ is optimal for L', which implies that the optimal schedule for L' has k^* size.

We now see by our initial inductive hypothesis that running the greedy algorithm on L' should produce a schedule of size k^*. Hence, by our construction, running the greedy algorithm on L' gives us $S[2,\dots,k]$.

This means $k - 1 = k^*$ or $k = k^* + 1$, which implies that $S[1,\dots,k]$ is indeed optimal, and we are done.

□

Weighted Interval Scheduling

Each request i has weight $w(i)$. Schedule subset of requests that are non-overlapping with maximum weight.

A key observation here is that the greedy algorithm no longer works.

Dynamic Programming

We can define our sub-problems as

$$R^x = \{j \in R | s(j) \geq x\}$$

Here, R is the set of all requests.

If we set $x = f(i)$, then R^x is the set of requests later than request i.

Total number of sub-problems $= n$ (one for each request)

Only need to solve each subproblem once and memoize.

We try each request i as a possible first. If we pick a request as the first, then the remaining requests are $R^{f(i)}$.

Note that even though there may be requests compatible with i that are not in $R^{f(i)}$, we are picking i as the first request, i.e., we are going in order.

$$opt(R) = \max_{1 \leq i \leq n} (w(i) + opt(R^{f(i)}))$$

Total running time is $O(n^2)$ since we need $O(n)$ time to solve each sub-problem.

Turns out that we can actually reduce the overall complexity to $O(n \log n)$. We leave this as an exercise.

Non-identical machines

As before, we have n requests $\{1, 2, \ldots, n\}$. Each request i is associated with a start time $s(i)$ and finish time $f(i)$, m different machine types as well $\tau = \{T_1, \ldots, T_m\}$. Each request i is associated with a set $Q(i) \subseteq \tau$ that represents the set of machines that request i can be serviced on.

Each request has a weight of 1. We want to maximize the number of jobs that can be scheduled on the m machines.

This problem is in NP, since we can clearly check that a given subset of jobs with machine assignments is legal.

Can $k \leq n$ requests be scheduled? This problem is NP-complete.

Maximum number of requests that should be scheduled? This problem is NP-hard.

Dealing with intractability

1. Approximation algorithms: Guarantee within some factor of optimal in polynomial time.

2. Pruning heuristics to reduce (possible exponential) runtime on "real-world" examples.

3. Greedy or other sub-optimal heuristics that work well in practice but provide no guarantees.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

Lecture 2: Divide and Conquer

- Paradigm
- Convex Hull
- Median finding

Paradigm

Given a problem of size n divide it into subproblems of size $\frac{n}{b}$, $a \geq 1$, $b > 1$. Solve each subproblem recursively. Combine solutions of subproblems to get overall solution.

$$T(n) = aT(\frac{n}{b}) + [\text{work for merge}]$$

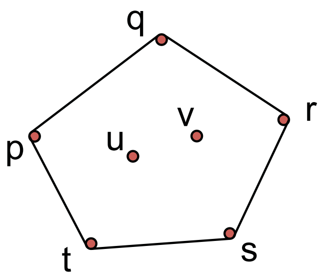
Convex Hull

Given n points in plane

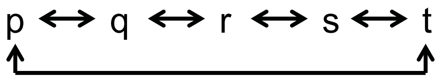
$$S = \{(x_i, y_i) | i = 1, 2, \ldots, n\}$$

assume no two have same x coordinate, no two have same y coordinate, and no three in a line for convenience.

Convex Hull (CH(S)): smallest polygon containing all points in S.



CH(S) represented by the sequence of points on the boundary in order clockwise as doubly linked list.



Brute force for Convex Hull

Test each line segment to see if it makes up an edge of the convex hull

- If the rest of the points are on one side of the segment, the segment is on the convex hull.
- else the segment is not.

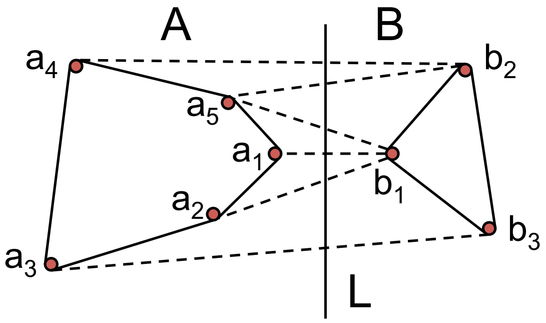
$O(n^2)$ edges, $O(n)$ tests $\Rightarrow$ $O(n^3)$ complexity
Can we do better?

Divide and Conquer Convex Hull

Sort points by x coord (once and for all, $O(n \log n)$)
For input set S of points:

- Divide into left half A and right half B by x coords
- Compute $CH(A)$ and $CH(B)$
- Combine CH's of two halves (merge step)

How to Merge?



- Find upper tangent (a_i, b_j). In example, (a_4, b_2) is U.T.
- Find lower tangent (a_k, b_m). In example, (a_3, b_3) is L.T.

- Cut and paste in time $\Theta(n)$.

First link a_i to b_j, go down b ilst till you see b_m and link b_m to a_k, continue along the a list until you return to a_i. In the example, this gives (a_4, b_2, b_3, a_3).

Finding Tangents

Assume a_i maximizes x within $CH(A)$ $(a_1, a_2, \ldots, a_p)$. b_1 minimizes x within $CH(B)$ $(b_1, b_2, \ldots, b_q)$

L is the vertical line separating A and B. Define $y(i, j)$ as y-coordinate of intersection between L and segment (a_i, b_j).

Claim: (a_i, b_j) is uppertangent iff it maximizes $y(i, j)$.

If $y(i, j)$ is not maximum, there will be points on both sides of (a_i, b_j) and it cannot be a tangent.

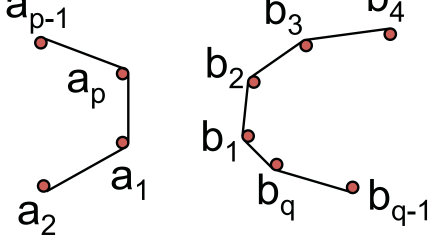
Algorithm: Obvious $O(n^2)$ algorithm looks at all a_i, b_j pairs. $T(n) = 2T(n/2) + \Theta(n^2) = \Theta(n^2)$.

```
i = 1
j = 1
while ($y(i, j+1) > y(i, j)$ or $y(i-1, j) > y(i, j)$)
    if ($y(i, j+1) > y(i, j)$) ▷ move right finger clockwise
        $j = j + 1 (\mod q)$
    else
        $i = i - 1 (\mod p)$ ▷ move left finger anti-clockwise
    return $(a_i, b_j)$ as upper tangent
```

Similarly for lower tangent.

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

Intuition for why Merge works



a_1, b_1 are right most and left most points. We move anti clockwise from a_1, clockwise from b_1. $a_1, a_2, \ldots, a_q$ is a convex hull, as is $b_1, b_2, \ldots, b_q$. If a_i, b_j is such that moving from either a_i or b_j decreases $y(i, j)$ there are no points above the (a_i, b_j) line.

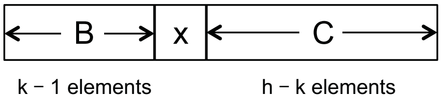
The formal proof is quite involved and won't be covered.

Median Finding

Given set of n numbers, define $rank(x)$ as number of numbers in the set that are $\leq x$. Find element of rank $\lfloor \frac{n+1}{2} \rfloor$ (lower median) and $\lceil \frac{n+1}{2} \rceil$ (upper median).

Clearly, sorting works in time $\Theta(n \log n)$.

Can we do better?

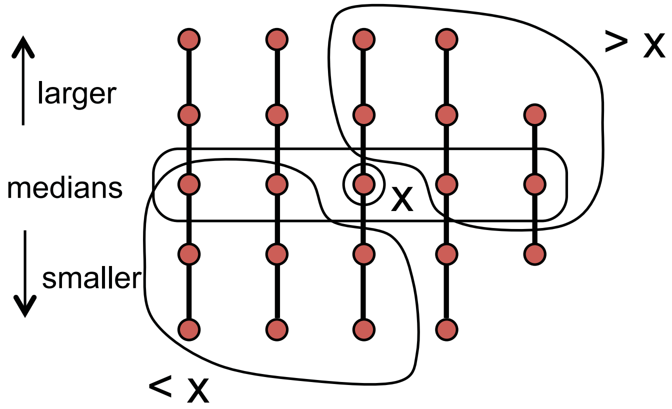


```
Select(S, i)
Pick $x \in S$ ▷ cleverly
Compute $k = rank(x)$
$B = \{y \in S | y < x\}$
$C = \{y \in S | y > x\}$
if $k = i$
    return x
else if $k > i$
    return Select$(B, i)$
else if $k < i$
    return Select$(C, i - k)$
```

Picking x Cleverly

Need to pick x so $rank(x)$ is not extreme.

- Arrange S into columns of size 5 ($\lceil \frac{n}{5} \rceil$ cols)
- Sort each column (bigger elements on top) (linear time)
- Find "median of medians" as x



How many elements are guaranteed to be $> x$?

Half of the $\lceil \frac{n}{5} \rceil$ groups contribute at least 3 elements $> x$ except for 1 group with less than 5 elements and 1 group that contains x.

At lease $3(\lceil \frac{n}{10} \rceil - 2)$ elements are $> x$, and at least $3(\lceil \frac{n}{10} \rceil - 2)$ elements are $< x$

Recurrence:

$$T(n) = \begin{cases} O(1), & \text{for } n \leq 140 \\ T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10} + 6), \Theta(n), & \text{for } n > 140 \end{cases} \tag{1}$$

Solving the Recurrence

Master theorem does not apply. Intuition $\frac{n}{5} + \frac{7n}{10} < n$.

Prove $T(n) \leq cn$ by induction, for some large enough c.

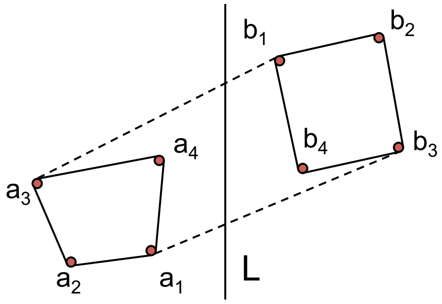
True for $n \leq 140$ by choosing large c

$$\begin{aligned} T(n) &\leq c\lceil \frac{n}{5} \rceil + c(\frac{7n}{10} + 6) + an & (2) \\ &\leq \frac{cn}{5} + c + \frac{7nc}{10} + 6c + an & (3) \\ &= cn + (-\frac{cn}{10} + 7c + an) & (4) \end{aligned}$$

If $c \geq \frac{70c}{n} + 10a$, we are done. This is true for $n \geq 140$ and $c \geq 20a$.

Appendix 1

Example



a_3, b_1 is upper tangent. $a_4 > a_3$, $b_2 > b_1$ in terms of Y coordinates.
a_1, b_3 is lower tangent, $a_2 < a_1$, $b_4 < b_3$ in terms of Y coordinates.

a_i, b_j is an upper tangent. Does not mean that a_i or b_j is the highest point. Similarly, for lower tangent.

Lecture 3: Divide and Conquer: Fast Fourier Transform

- Polynomial Operations vs. Representations
- Divide and Conquer Algorithm
- Collapsing Samples / Roots of Unity
- FFT, IFFT, and Polynomial Multiplication

Polynomial operations and representation

A polynomial $A(x)$ can be written in the following forms:

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \\ &= \sum_{k=0}^{n_1} a_k x^k \\ &= \langle a_0, a_1, a_2, \ldots, a_{n-1} \rangle \quad \text{(coefficient vector)} \end{aligned}$$

The degree of A is $n-1$.

Operations on polynomials

There are three primary operations for polynomials.

1. **Evaluation**: Given a polynomial $A(x)$ and a number x_0, compute $A(x_0)$. This can be done in $O(n)$ time using $O(n)$ arithmetic operations via Horner's rule.

   - **Horner's Rule:** $A(x) = a_0 + x(a_1 + x(a_2 + \cdots x(a_{n-1}) \cdots))$. At each step, a sum is evaluated, then multiplied by x, before beginning the next step. Thus $O(n)$ multiplications and $O(n)$ additions are required.

2. **Addition:** Given two polynomials $A(x)$ and $B(x)$, compute $C(x) = A(x) + B(x)$ $(\forall x)$. This takes $O(n)$ time using basic arithmetic, because $c_k = a_k + b_k$.

3. **Multiplication:** Given two polynomials $A(x)$ and $B(x)$, compute $C(x) = A(x) \cdot B(x)$ $(\forall x)$. Then $c_k = \sum_{j=0}^{k} a_j b_{k-j}$ for $0 \leq k \leq 2(n-1)$, because the degree of the resulting polynomial is twice that of A or B. This multiplication is then equivalent to a convolution of the vectors A and reverse(B). The *convolution* is the inner product of all relative shifts, an operation also useful for smoothing etc. in digital signal processing.

   - Naive polynomial multiplication takes $O(n^2)$.
   - $O(n^{\lg 3})$ or even $O(n^{1+\varepsilon})$ $(\forall \varepsilon > 0)$ is possible via Strassen-like divide-and-conquer tricks.
   - Today, we will compute the product in $O(n \lg n)$ time via Fast Fourier Transform!

Representations of polynomials

First, consider the different representations of polynomials, and the time necessary to complete operations based on the representation.

There are 3 main representations to consider.

1. Coefficient vector with a monomial basis

2. Roots and a scale term

   - $A(x) = (x - r_0) \cdot (x - r_1) \cdot \dots \cdot (x - r_{n-1}) \cdot c$
   - However, it is impossible to find exact roots with only basic arithmetic operations and kth root operations. Furthermore, addition is extremely hard with this representation, or even impossible. Multiplication simply requires roots to be concatenated, and evaluation can be completed in $O(n)$.

3. Samples: (x_0, y_0), (x_1, y_1), $\dots$, (x_{n-1}, y_{n-1}) with $A(x_i) = y_i$ $(\forall i)$ and each x_i is distinct. These samples uniquely determine a degree $n-1$ polynomial A, according to the Lagrange and Fundamental Theorem of Algebra. Addition and multiplication can be computed by adding and multiplying the y_i terms, assuming that the x_i's match. However, evaluation requires interpolation.

The runtimes for the representations and the operations is described in the table below, with algorithms for the operations versus the representations.

Algorithms vs.	**Representations**		
	Coefficients	**Roots**	**Samples**
Evaluation	$O(n)$	$O(n)$	$O(n^2)$
Addition	$O(n)$	∞	$O(n)$
Multiplication	$O(n^2)$	$O(n)$	$O(n)$

We combine the best of each representation by converting between coefficients and samples in $O(n \lg n)$ time.

How? Consider the polynomial in matrix form.

$$V \cdot A = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

where V is the Vandermonde matrix with entries $v_{jk} = x_j^k$.

Then we can convert between coefficients and samples using the matrix vector product $V \cdot A$, which is equivalent to evaluation. This takes $O(n^2)$.

Similarly, we can samples to coefficients by solving $V \backslash Y$ (in MATLAB®notation). This takes $O(n^3)$ via Gaussian elimination, or $O(n^2)$ to compute $A = V^{-1} \cdot Y$, if V^{-1} is precomputed.

To do better than $\Theta(n^2)$ when converting between coefficients and samples, and vice versa, we need to choose special values for $x_0, x_1, \ldots, x_{n-1}$. Thus far, we have only made the assumption that the x_i values are distinct.

Divide and Conquer Algorithm

We can formulate polynomial multiplication as a divide and conquer algorithm with the following steps for a polynomial $A(x)$ ∀ x ∈ X.

1. Divide the polynomial A into its even and odd coefficients:

$$A_{even}(x) = \sum_{k=0}^{\lceil \frac{n}{2} - 1 \rceil} a_{2k} x^k = \langle a_0, a_2, a_4, \ldots \rangle$$

$$A_{odd}(x) = \sum_{k=0}^{\lfloor \frac{n}{2} - 1 \rfloor} a_{2k+1} x^k = \langle a_1, a_3, a_5, \ldots \rangle$$

2. Recursively conquer $A_{even}(y)$ for $y \in X^2$ and $A_{odd}(y)$ for $y \in X^2$, where $X^2 = \{x^2 \mid x \in X\}$.

3. Combine the terms. $A(x) = A_{even}(x^2) + x \cdot A_{odd}(x^2)$ for $x \in$ X.

However, the recurrences for this algorithm is

$$\begin{aligned} T(n, |X|) &= 2 \cdot T\left(\frac{n}{2}, |X|\right) + O(n + |X|) \\ &= O(n^2) \end{aligned}$$

which is no better than before.

We can do better if X is *collapsing*: either $|X| = 1$ (base case), or $|X^2| = \frac{|X|}{2}$ and X^2 is (recursively) collapsing. Then the recurrence is of the form

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) = O(n \lg n).$$

Roots of Unity

Collapsing sets can be constructed via square roots. Each of the following collapsing sets is computing by taking all square roots of the previous set.

1. $\{1\}$

2. $\{1, -1\}$

3. $\{1, -1, i, -i\}$

4. $\{1, -1, \pm\frac{\sqrt{2}}{2}(1+i), \pm\frac{\sqrt{2}}{2}(-1+i)\}$, which lie on a unit circle

We can repeat this process and make our set larger and larger by finding more and more points on this circle. These points are called the nth roots of unity. Formally, the nth roots of unity are n x's such that $x^n = 1$. These points are uniformly spaced around the unit circle in the complex plane (including 1). These points are of the form $(\cos\theta, \sin\theta) = \cos\theta + i\sin\theta = e^{i\theta}$ by Euler's Formula, for $\theta = 0, \frac{1}{n}\tau, \frac{2}{n}\tau, \ldots, \frac{n-1}{n}\tau$ (where $\tau = 2\pi$).

The nth roots of unity where $n = 2^\ell$ form a collapsing set, because $(e^{i\theta})^2 = e^{i(2\theta)} = e^{i(2\theta \bmod \tau)}$. Therefore the even nth roots of unity are equivalent to the $\frac{n}{2}$nd roots of unity.

FFT, IFFT, and Polynomial Multiplication

We can take advantage of the nth roots of unity to improve the runtime of our polynomial multiplication algorithm. The basis for the algorithm is called the Discrete Fourier Transform (DFT).

The DFT allows the transformation between coefficients and samples, computing $A \to A^* = V \cdot A$ for $x_k = e^{i\tau k/n}$ where $n = 2^\ell$, where A is the set of coefficients and A^* is the resulting samples. The individual terms $a_j^* = \sum_{j=0}^{n-1} e^{i\tau jk/n} \cdot a_j$.

Fast Fourier Transform (FFT)

The FFT algorithm is an $O(n \lg n)$ divide and conquer algorithm for DFT, used by Gauss circa 1805, and popularized by Cooley and Turkey and 1965. Gauss used the algorithm to determine periodic asteroid orbits, while Cooley and Turkey used it to detect Soviet nuclear tests from offshore readings.

A practical implementation of FFT is FFTW, which was described by Frigo and Johnson at MIT. The algorithm is often implemented directly in hardware, for fixed n.

Inverse Discrete Fourier Transform

The Inverse Discrete Fourier Transform is an algorithm to return the coefficients of a polynomial from the multiplied samples. The transformation is of the form $A^* \to V^{-1} \cdot A^* = A$.

In order to compute this, we need to find V^{-1}, which in fact has a very nice structure.

Claim 1. *$V^{-1} = \frac{1}{n}\bar{V}$, where $\bar{V}$ is the complex conjugate of V.*[1]

[1] Recall the complex conjugate of $p + qi$ is $p - qi$.

Proof. We claim that $P = V \cdot \bar{V} = nI$:

$$\begin{aligned} p_{jk} &= (\text{row } j \text{ of } V) \cdot (\text{col. } k \text{ of } \bar{V}) \\ &= \sum_{m=0}^{n-1} e^{ij\tau m/n} \overline{e^{ik\tau m/n}} \\ &= \sum_{m=0}^{n-1} e^{ij\tau m/n} e^{-ik\tau m/n} \\ &= \sum_{m=0}^{n-1} e^{i(j-k)\tau m/n} \end{aligned}$$

Now if $j = k$, $p_{jk} = \sum_{m=0}^{n-1} = n$. Otherwise it forms a geometric series.

$$\begin{aligned} p_{jk} &= = \sum_{m=0}^{n-1} (e^{i(j-k)\tau/n})^m \\ &= \frac{(e^{i\tau(j-k)/n})^n - 1}{e^{i\tau(j-k)/n} - 1} \\ &= 0 \end{aligned}$$

because $e^{i\tau} = 1$. Thus $V^{-1} = \frac{1}{n}\bar{V}$, because $V \cdot \bar{V} = nI$. □

This claim says that the Inverse Discrete Fourier Transform is equivalent to the Discrete Fourier Transform, but changing x_k from $e^{ik\tau/n}$ to its complex conjugate $e^{-ik\tau/n}$, and dividing the resulting vector by n. The algorithm for IFFT is analogous to that for FFT, and the result is an $O(n \lg n)$ algorithm for IDFT.

Fast Polynomial Multiplication

In order to compute the product of two polynomials A and B, we can perform the following steps.

1. Compute $A^* = FFT(A)$ and $B^* = FFT(B)$, which converts both A and B from coefficient vectors to a sample representation.

2. Compute $C^* = A^* \cdot B^*$ in sample representation in linear time by calculating $C_k^* = A_k^* \cdot B_k^*$ $(\forall k)$.

3. Compute $C = \text{IFFT}(C^*)$, which is a vector representation of our final solution.

Applications

Fourier (frequency) space many applications. The polynomial $A^* = FFT(A)$ is complex, and the amplitude $|a_k^*|$ represents the amplitude of the frequency-k signal, while $\arg(a_k^*)$ (the angle of the 2D vector) represents the phase shift of that signal. For example, this perspective is particularly useful for audio processing, as used by Adobe Audition, Audacity, etc.:

- High-pass filters zero out high frequencies
- Low-pass filters zero out low frequencies
- Pitch shifts shift the frequency vector
- Used in MP3 compression, etc.

Lecture 4: Divide and Conquer: van Emde Boas Trees

- Series of Improved Data Structures
- Insert, Successor
- Delete
- Space

This lecture is based on personal communication with Michael Bender, 2001.

Goal

We want to maintain n elements in the range $\{0, 1, 2, \ldots, u-1\}$ and perform Insert, Delete and Successor operations in $\mathcal{O}(\log\log u)$ time.

- If $n = n^{\mathcal{O}(1)}$ or $n^{(\log n)^{\mathcal{O}(1)}}$, then we have $\mathcal{O}(\log\log n)$ time operations
  - Exponentially faster than Balanced Binary Search Trees
  - Cooler queries than hashing
- Application: Network Routing Tables
  - u = Range of IP Addresses → port to send ($u = 2^{32}$ in IPv4)

Where might the $\mathcal{O}(\log\log u)$ bound arise ?

- Binary search over $\mathcal{O}(\log u)$ elements
- Recurrences
  - $T(\log u) = T\left(\frac{\log u}{2}\right) + \mathcal{O}(1)$
  - $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$

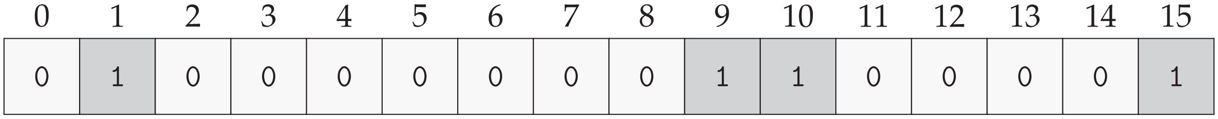
Improvements

We will develop the van Emde Boas data structure by a series of improvements on a very simple data structure.

Bit Vector

We maintain a vector V of size u such that $V[x] = 1$ if and only if x is in the set. Now, inserts and deletes can be performed by just flipping the corresponding bit in the vector. However, successor/predecessor requires us to traverse through the vector to find the next 1-bit.

- Insert/Delete: $\mathcal{O}(1)$
- Successor/Predecessor: $\mathcal{O}(u)$



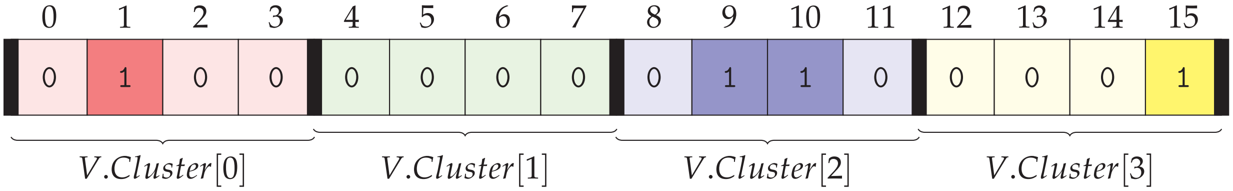
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1

Figure 1: Bit vector for $u = 16$. THe current set is $\{1, 9, 10, 15\}$.

Split Universe into Clusters

We can improve performance by splitting up the range $\{0, 1, 2, \ldots, u-1\}$ into $\sqrt{u}$ clusters of size $\sqrt{u}$. If $x = i\sqrt{u} + j$, then $V[x] = V.Cluster[i][j]$.

$$low(x) = x \bmod \sqrt{u} = j$$
$$high(x) = \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor = i$$
$$index(i, j) = i\sqrt{u} + j$$



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1

Figure 2: Bit vector ($u = 16$) split into $\sqrt{16} = 4$ clusters of size 4.

- Insert:
  - Set $V.cluster[high(x)][low(x)] = 1$ $\mathcal{O}(1)$

  - Mark cluster $high(x)$ as non-empty $\mathcal{O}(1)$

- Successor:

  - Look within cluster $high(x)$ $\mathcal{O}(\sqrt{u})$
  - Else, find next non-empty cluster i $\mathcal{O}(\sqrt{u})$
  - Find minimum entry j in that cluster $\underline{\mathcal{O}(\sqrt{u})}$
  - Return $index(i, j)$ Total $= \mathcal{O}(\sqrt{u})$

Recurse

The three operations in Successor are also Successor calls to vectors of size $\sqrt{u}$. We can use recursion to speed things up.

- $V.cluster[i]$ is a size-$\sqrt{u}$ van Emde Boas structure ($\forall\ 0 \leq i < \sqrt{u}$)
- $V.summary$ is a size-$\sqrt{u}$ van Emde Boas structure
- $V.summary[i]$ indicates whether $V.cluster[i]$ is nonempty

```
INSERT($V, x$)
Insert($V.cluster[high(x)], low[x]$)
Insert($V.summary, high[x]$)
```

So, we get the recurrence:

$$\begin{aligned} T(u) &= 2T(\sqrt{u}) + \mathcal{O}(1) \\ T'(\log u) &= 2T'\left(\frac{\log u}{2}\right) + \mathcal{O}(1) \\ \implies T(u) &= T'(\log u) = \mathcal{O}(\log u) \end{aligned}$$

```
SUCCESSOR($V, x$)
$i = high(x)$
$j = Successor(V.cluster[i], j)$
if $j == \infty$
    $i = Successor(V.summary, i)$
    $j = Successor(V.cluster[i], -\infty)$
return $index(i, j)$
```

$$\begin{aligned} T(u) &= 3T(\sqrt{u}) + \mathcal{O}(1) \\ T'(\log u) &= 3T'\left(\frac{\log u}{2}\right) + \mathcal{O}(1) \\ \implies T(u) &= T'(\log u) = \mathcal{O}((\log u)^{\log 3}) \approx \mathcal{O}((\log u)^{1.585}) \end{aligned}$$

To obtain the $\mathcal{O}(\log\log u)$ running time, we need to reduce the number of recursions to one.

Maintain Min and Max

We store the minimum and maximum entry in each structure. This gives an $\mathcal{O}(1)$ time overhead for each *Insert* operation.

```
SUCCESSOR($V, x$)
$i = high(x)$
if $low(x) < V.cluster[i].max$
        $j = Successor(V.cluster[i], low(x))$
else $i = Successor(V.summary, high(x))$
        $j = V.cluster[i].min$
return $index(i, j)$
```

$$\begin{aligned} T(u) &= T(\sqrt{u}) + \mathcal{O}(1) \\ \implies T(u) &= \mathcal{O}(\log\log u) \end{aligned}$$

Don't store Min recursively

The *Successor* call now needs to check for the min separately.

$$\textbf{if } x < V.min : \textbf{return } V.min \tag{1}$$

```
INSERT($V, x$)
if $V.min == None$
    $V.min = V.max = x$    ▷ $\mathcal{O}(1)$ time
    return
if $x < V.min$
    $swap(x \leftrightarrow V.min)$
if $x > V.max$
    $V.max = x)$
if $V.cluster[high(x) == None$
    $Insert(V.summary, high(x))$    ▷ First Call
$Insert(V.cluster[high(x)], low(x))$    ▷ Second Call
```

If the **first call** is executed, the **second call** only takes $\mathcal{O}(1)$ time. So

$$\begin{aligned} & T(u) = T(\sqrt{u}) + \mathcal{O}(1) \\ \Longrightarrow\ & T(u) = \mathcal{O}(\log\log u) \end{aligned}$$

```
DELETE($V, x$)
if $x == V.min$    ▷ Find new min
    $i = V.summary.min$
    if $i = None$
        $V.min = V.max = None$    ▷ $\mathcal{O}(1)$ time
        return
    $V.min = index(i, V.cluster[i].min)$    ▷ Unstore new min
$Delete(V.cluster[high(x)], low(x))$    ▷ First Call
if $V.cluster[high(x)].min == None$
    $Delete(V.summary, high(x))$    ▷ Second Call
▷ Now we update $V.max$
if $x == V.max$
if $V.summary.max = None$
else
    $i = V.summary.max$
    $V.max = index(i, V.cluster[i].max)$
```

If the **second call** is executed, the **first call** only takes $\mathcal{O}(1)$ time. So

$$\begin{aligned} & T(u) = T(\sqrt{u}) + \mathcal{O}(1) \\ \Longrightarrow\ & T(u) = \mathcal{O}(\log\log u) \end{aligned}$$

Lower Bound [Patrascu & Thorup 2007]

Even for static queries (no Insert/Delete)

- $\Omega(\log\log u)$ time per query for $u = n^{(\log n)^{\mathcal{O}(1)}}$
- $\mathcal{O}(n \cdot poly(\log n))$ space

Space Improvements

We can improve from $\Theta(u)$ to $\mathcal{O}(n \log\log u)$.

- Only create nonempty clusters
  - If $V.min$ becomes $None$, deallocate V
- Store $V.cluster$ as a hashtable of nonempty clusters
- Each insert may create a new structure $\Theta(\log\log u)$ times (each empty insert)
  - Can actually happen [Vladimir Čunát]
- Charge pointer to structure (and associated hash table entry) to the structure

This gives us $\mathcal{O}(n \log\log u)$ space (but randomized).

Indirection

We can further reduce to $\mathcal{O}(n)$ space.

- Store vEB structure with $n = \mathcal{O}(\log\log u)$ using BST or even an array

  $\implies \mathcal{O}(\log\log n)$ time once in base case
- We use $\mathcal{O}(n/\log\log u)$ such structures (disjoint)

  $\implies \mathcal{O}(\frac{n}{\log\log u} \cdot \log\log u) = \mathcal{O}(n)$ space for small
- Larger structures "store" pointers to them

  $\implies \mathcal{O}(\frac{n}{\log\log u} \cdot \log\log u) = \mathcal{O}(n)$ space for large
- Details: Split/Merge small structures

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

Lecture 5: Amortization

Amortized analysis is a powerful technique for data structure analysis, involving the total runtime of a sequence of operations, which is often what we really care about. This lecture covers:

- Different techniques of amortized analysis
  - aggregate method
  - accounting method
  - charging method
  - potential method
- Examples of amortized analysis
  - table doubling
  - binary counter
  - 2-3 tree and 2-5 tree

Table doubling

(Recall from 6.006) We want to store n elements in a table of size $m = \Theta(n)$. One idea is to double m whenever n becomes larger than m (due to insertions). The cost to double a table of size m is clearly $\Theta(m) = \Theta(n)$, which is also the worse case cost of an insertion.

But what is the total cost of n insertions? It is at most

$$2^0 + 2^1 + 2^2 + \cdots + 2^{\lceil \lg n \rceil} = \Theta(n).$$

In this case, we say each insertion has $\Theta(n)/n = \Theta(1)$ *amortized cost.*

Aggregate Method

The method we used in the above analysis is the aggregate method: just add up the cost of all the operations and then divide by the number of operations.

$$\text{amortized cost per operation } = \frac{\text{total cost of } k \text{ operations}}{k}$$

Aggregate method is the simplest method. Because it's simple, it may not be able to analyze more complicated algorithms.

Amortized Bound Definition

Amortized cost can be, but does not have to be, average cost. We can assign any amortized cost to each operation, as long as they "preserve the total cost", i.e., for any sequence of operations,

$$\sum \text{amortized cost } \geq \sum \text{ actual cost}$$

where the sum is taken over all operations.

For example, we can say a 2-3 tree achieves $O(1)$ amortized cost per create, $O(\lg n^*)$ amortized cost per insert, and 0 amortized cost per delete, where n^* is the maximum size of the 2-3 tree during the entire sequence of operations. The reason we can claim this is that for any sequence of operations, suppose there are c creations, i insertions and $d \leq i$ deletions (cannot delete from an empty tree), the total amortized cost is asymptotically the same as the total actual cost:

$$O(c + i \lg n^* + 0d) = O(c + i \lg n^* + d \lg n^*)$$

Later, we will tighten the amortized cost per insert to $O(\lg n)$ where n is the *current* size.

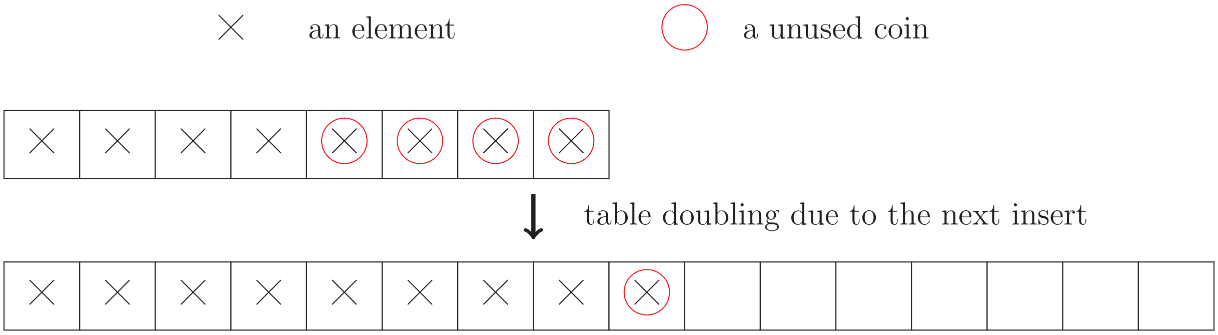
Accounting Method

This method allows an operation to store credit into a bank for future use, if its assigned amortized cost > its actual cost; it also allows an operation to pay for its extra actual cost using existing credit, if its assigned amortized cost < its actual cost.

Table doubling

For example, in table doubling:

- if an insertion does not trigger table doubling, store a coin represnting $c = O(1)$ work for future use.
- if an insertion does trigger table doubling, there must be $n/2$ elements that are inserted after the previous table doubling, whose coins have not been consumed. Use up these $n/2$ coins to pay for the $O(n)$ table doubling. See figure below.
- amortized cost for table doubling: $O(n) - c \cdot n/2 = 0$ for large enough c.
- amortized cost per insertion: $1 + c = O(1)$.



2-3 trees

Now let's try the accounting method on 2-3 trees. Our goal is to show that insert has $O(\lg n)$ amortized cost and delete has 0 amortized cost. Let's try a natural approach: save a $O(\lg n)$ coin for inserting an element, and use this coin when we delete this element later. However, we will run into a problem: by the time we delete the element, the size of the tree may have got bigger $n' > n$, and the coin we saved is not enough to pay for the $\lg n'$ actual cost of that delete operation! This problem can be solved using the charging method in the next section.

Charging Method

The charging method allows operations to charge cost retroactively to *past* operations.

$$\begin{aligned}\text{amortized cost of an operation} &= \text{actual cost of this operation}\\ &\quad- \text{total cost charged to past operations}\\ &\quad+ \text{total cost charged by future operations}\end{aligned}$$

Table doubling and halving

For example, in table doubling, when the table doubles from m to $2m$, we can charge $\Theta(m)$ cost to the $m/2$ insert operations since the last doubling. Each insert is charged by $\Theta(1)$, and will not be charged again. So the amortized cost per insert is $\Theta(1)$.

Now let's extend the above example with table halving. The motivation is to save space when with deletes. If the table is down to $1/4$ full, $n = m/4$, we shrink the table size from m to $m/2$ at $\Theta(m)$ cost. This way, the table is half full again after any resize (doubling or shrinking). Now each table doubling still has $\geq m/2$ insert operations to charge to, and each table halving has $\geq m/4$ delete operations to charge to. So the amortized cost per insert or delete is still $\Theta(1)$.

Free deletion in 2-3 trees

For another example, let's consider insertion and deletion in 2-3 trees. Again, our goal is to show that insert has $O(\lg n)$ amortized cost, where n is the size of the tree when that insert happens, and delete has 0 amortized cost.

Insert does not need to charge anything.

Delete will charge an insert operation. But we will not charge the insert of the element to be deleted, because we will run into the same problem as the accounting method. Instead, each delete operation will charge the insert operation that brought the tree to its current size n. Each insert is still charged at most once, because for the tree size to reach n again, another insert must happen.

Potential Method

This method defines a potential function Φ that maps a data structure (DS) configuration to a value. This function Φ is equivalent to the total unused credits stored up by all past operations (the bank account balance). Now

$$\text{amortized cost of an operation} \;=\; \text{actual cost of this operation} \;+\Delta\Phi$$

and

$$\sum \text{amortized cost} \;=\sum \text{actual cost} \;+\Phi(\text{final DS})-\Phi(\text{initial DS}).$$

In order for the amortized bound to hold, Φ should never go below Φ(initial DS) at any point. If $\Phi(\text{initial DS}) = 0$, which is usually the case, then Φ should never go negative (intuitively, we cannot "owe the bank").

Relation to accounting method

In accounting method, we specify $\Delta\Phi$, while in potential method, we specify Φ. One determines the other, so the two methods are equivalent. But sometimes one is more intuitive than the other.

Binary counter

Our first example of potential method is incrementing a binary counter. E.g.,

$$\begin{array}{rc} & 0011010111 \\ \text{increment} & \downarrow \\ & 0011011000 \end{array}$$

Cost of increment is $\Theta(1 + \#1)$, where $\#1$ represents the number of trailing 1 bits. So the intuition is that 1 bits are bad.

Define $\Phi = c \cdot \#1$. Then for large enough c,

$$\begin{aligned}\text{amortized cost} &= \text{actual cost } + \Delta\Phi \\ &= \Theta(1 + \#1) + c(-\#1 + 1) \\ &= \Theta(1)\end{aligned}$$

$\Phi(\text{initial DS}) = 0$ if the counter starts at $000 \cdots 0$. This is necessary for the above amortized analysis. Otherwise, Φ may become smaller than $\Phi(\text{initial DS})$.

Insert in 2-3 trees

Insert can cause $O(\lg n)$ splits in the worst case, but we can show it causes only $O(1)$ amortized splits. First consider what causes a split: insertion into a 3-node (a node with 3 children). In that case, the 3-node needs to split into two 2-nodes.

So 3-nodes are bad. We define Φ = the number of 3-nodes. Then $\Delta\Phi \leq 1 -$ the number of splits. Amortized number of splits = actual number of splits + $\Delta\Phi$ = 1. $\Phi(\text{initial DS}) = 0$ if the tree is empty initially.

The above analysis holds for any (a, b)-tree, if we define Φ to be the number of b-nodes.

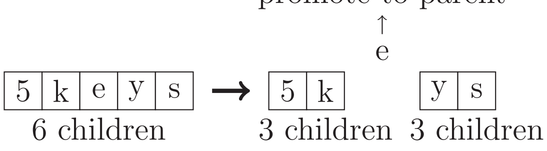
If we consider both insertion and deletion in 2-3 trees, can we claim both $O(1)$ splits for insert, and $O(1)$ merges for delete? The answer is no, because a split creates two 2-nodes, which are bad for merge. In the worse case, they may be merged by the next delete, and then need split again on the next insert, and so on.

What do we solve this problem? We need to prevent split and merge from creating 'bad' nodes.

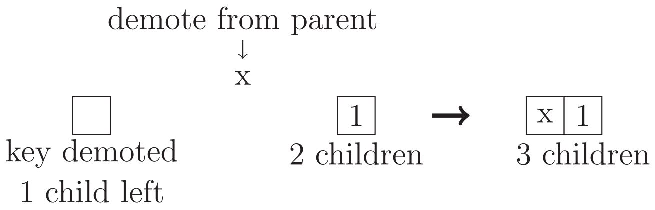
Insert and delete in 2-5 trees

We can claim $O(1)$ splits for insert, and $O(1)$ merges for delete in 2-5 trees.

In 2-5 trees, insertion into a 5-node (a node with 5 children) causes it to split into two 3-nodes.



Deletion from a 2-node causes it to merge with another 2-node to form a 3-node.



5-nodes and 2-nodes are bad. We define $\Phi = \#$ of 5-nodes $+ \#$ of 2-nodes. Amortized splits and merges = 1. Φ(initial DS) = 0 if the tree is empty initially.

The above analysis holds for any (a, b)-tree where $b > 2a$, because splits and merges do not produce bad nodes. We define Φ to be the number of b-nodes plus the number of a nodes.

Note: The potential examples could also be done with the accounting method by placing coins on 1s (binary counter) or 2/5-nodes ($(2, 5)$-trees).

Lecture 6: Randomized Algorithms

- Check matrix multiplication
- Quicksort

Randomized or Probablistic Algorithms

What is a randomized algorithm?

- Algorithm that generates a random number $r \in \{1, ..., R\}$ and makes decisions based on r's value.
- On the same input on different executions, a randomized algorithm may
  - Run a different number of steps
  - Produce a different output

Randomized algorithms can be broadly classified into two types- Monte Carlo and Las Vegas.

Monte Carlo	Las Vegas
runs in polynomial time always	runs in expected polynomial time
output is correct with high probability	output always correct

Matrix Product

$$C = A \times B$$

Simple algorithm: $O(n^3)$ multiplications.
Strassen: multiply two 2×2 matrices in 7 multiplications: $O(n^{\log_2 7}) = O(n^{2.81})$
Coppersmith-Winograd: $O(n^{2.376})$

Matrix Product Checker

Given $n \times n$ matrices A, B, C, the goal is to check if $A \times B = C$ or not.

Question. Can we do better than carrying out the full multiplication?

We will see an $O(n^2)$ algorithm that:

- if $A \times B = C$, then $Pr[\text{output=YES}] = 1$.
- if $A \times B \neq C$, then $Pr[\text{output=YES}] \leq \frac{1}{2}$.

We will assume entries in matrices $\in \{0, 1\}$ and also that the arithmetic is mod 2.

Frievald's Algorithm

Choose a random binary vector $r[1...n]$ such that $Pr[r_i = 1] = 1/2$ independently for $r = 1, ..., n$. The algorithm will output 'YES' if $A(Br) = Cr$ and 'NO' otherwise.

Observation

The algorithm will take $O(n^2)$ time, since there are 3 matrix multiplications Br, $A(Br)$ and Cr of a $n \times n$ matrix by a $n \times 1$ matrix.

Analysis of Correctness if $AB \neq C$

Claim. If $AB \neq C$, then $Pr[ABr \neq Cr] \geq 1/2$.

Let $D = AB - C$. Our hypothesis is thus that $D \neq 0$. Clearly, there exists r such that $Dr \neq 0$. Our goal is to show that there are many r such that $Dr \neq 0$. Specifically, $Pr[Dr \neq 0] \geq 1/2$ for randomly chosen r.

$D = AB - C \neq 0 \implies \exists\ i, j$ s.t. $d_{ij} \neq 0$. Fix vector v which is 0 in all coordinates except for $v_j = 1$. $(Dv)_i = d_{ij} \neq 0$ implying $Dv \neq 0$. Take any r that can be chosen by our algorithm. We are looking at the case where $Dr = 0$. Let

$$r' = r + v$$

Since v is 0 everywhere except v_j, r' is the same as r exept $r'_j = (r_j + v_j) \mod 2$. Thus, $Dr' = D(r + v) = 0 + Dv \neq 0$. We see that there is a 1 to 1 correspondence between r and r', as if $r' = r + V = r'' + V$ then $r = r''$. This implies that

$$\text{number of } r' \text{ for which } Dr' \neq 0 \geq \text{number of } r \text{ for which } Dr = 0$$

From this we conclude that $Pr[Dr \neq 0] \geq 1/2$

Quicksort

Divide and conquer algorithm but work mostly in the divide step rather than combine. Sorts “in place” like insertion sort and unlike mergesort (which requires $O(n)$ auxiliary space).

Different variants:

- Basic: good in average case
- Median-based pivoting: uses median finding
- Random: good for all inputs in expectation (Las Vegas algorithm)

Steps of quicksort:

- Divide: pick a pivot element x in A, partition the array into sub-arrays L, consisting of all elements $< x$, G consisting of all elements $> x$ and E consisting of all elements $= x$.
- Conquer: recursively sort subarrays L and G
- Combine: trivial

Basic Quicksort

Pivot around $x = A[1]$ or $A[n]$ (first or last element)

- Remove, in turn, each element y from A
- Insert y into L, E or G depending on the comparison with pivot x
- Each insertion and removal takes $O(1)$ time
- Partition step takes $O(n)$ time
- To do this in place: see CLRS p. 171

Basic Quicksort Analysis

If input is sorted or reverse sorted, we are partitioning around the min or max element each time. This means one of L or G has $n-1$ elements, and the other 0. This gives:

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

However, this algorithm does well on random inputs in practice.

Pivot Selection Using Median Finding

Can guarantee balanced L and G using rank/median selection algorithm that runs in $\Theta(n)$ time. The first $\Theta(n)$ below is for the pivot selection and the second for the partition step.

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(n)$$
$$T(n) = \Theta(n \log n)$$

This algorithm is slow in practice and loses to mergesort.

Randomized Quicksort

x is chosen at random from array A (at each recursion, a random choice is made). Expected time is $O(n \log n)$ for all input arrays A. See CLRS p.181-184 for the analysis of this algorithm; we will analyze a variant of this.

"Paranoid" Quicksort

```
Repeat
    choose pivot to be random element of $A$
    perform Partition
Until
    resulting partition is such that
    $|L| \leq \frac{3}{4}|A|$ and $|G| \leq \frac{3}{4}|A|$
Recurse on $L$ and $G$
```

“Paranoid” Quicksort Analysis

Let’s define a ”good pivot” and a ”bad pivot”-

Good pivot: sizes of L and $G \leq \frac{3}{4}n$ each

Bad pivot: one of L and G is $\leq \frac{3}{4}n$ each

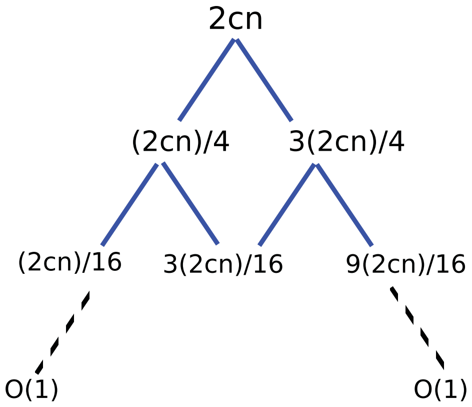
bad pivots	good pivots	bad pivots
$\frac{n}{4}$	$\frac{n}{2}$	$\frac{n}{4}$

We see that a pivot is good with probability $> 1/2$.

Let $T(n)$ be an upper bound on the expected running time on any array of n size. T(n) comprises:

- time needed to sort left subarray
- time needed to sort right subarray
- the number of iterations to get a good call. Denote as $c \cdot n$ the cost of the partition step

Expectations



$$T(n) \leq max_{n/4 \leq i \leq 3n/4}(T(i) + T(n-i)) + E(\#\text{iterations}) \cdot cn$$

Now, since probability of good pivot $> \frac{1}{2}$,

$$E(\#iterations) \leq 2$$

$$T(n) \leq T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + 2cn$$

We see in the figure that the height of the tree can be at most $\log_{\frac{4}{3}}(2cn)$ no matter what branch we follow to the bottom. At each level, we do a total of $2cn$ work. Thus, expected runtime is $T(n) = \Theta(n \log n)$

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

Lecture 8: Hashing

Course Overview

This course covers several modules:

1. Review: dictionaries, chaining, simple uniform
2. Universal hashing
3. Perfect hashing

Review

Dictionary Problem

A dictionary is an Abstract Data Type (ADT) that maintains a set of items. Each item has a key. The dictionary supports the following operations:

- **insert(item)**: add item to set
- **delete(item)**: remove item from set
- **search(key)**: return item with key if it exists

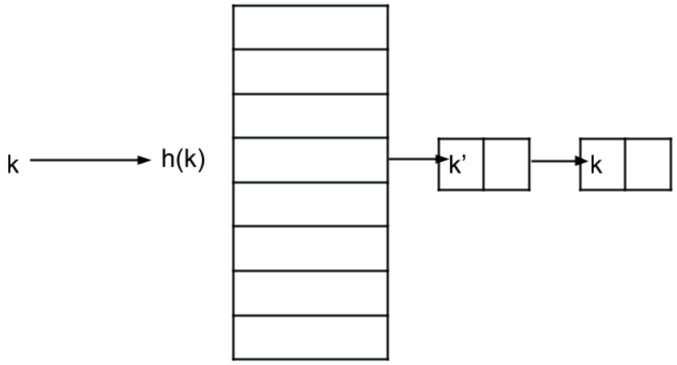
We assume that items have distinct keys (or that inserting new ones clobbers old ones).

This problem is easier than predecessor/successor problems solved in previous lecture (by van Emde Boas trees, or by AVL/2-3 trees/skip lists).

Hashing from 6.006

Goal: $O(1)$ time per operation and $O(n)$ space complexity.
Definitions:

- u = number of keys over all possible items
- n = number of keys/items currently in the table
- m = number of slots in the table



Solution: hashing with chaining

Assuming simple uniform hashing,

$$\Pr_{k_1 \neq k_2}\{h(k_1) = h(k_2)\} = \frac{1}{m}$$

we achieve $\Theta(1 + \alpha)$ time per operation, where $\alpha = \frac{n}{m}$ is called load factor. The downside of the algorithm is that it requires assuming input keys are random, and it only works in average case, like basic quicksort. Today we are going to remove the unreasonable simple uniform hashing assumption.

Etymology

The English 'hash' (1650s) means "cut into small pieces", which comes from the French 'hacher' which means "chop up", which comes from the Old French 'hache' which means "axe" (cf. English 'hatchet'). Alternatively, perhaps they come from Vulcan 'la'ash', which means "axe". (R.I.P. Leonard Nimoy.)

Universal Hashing

The idea of universal hashing is listed as following:

- choose a random hash function h from $\mathcal{H}$
- require $\mathcal{H}$ to be a *universal hashing family* such that

$$\Pr_{h \in \mathcal{H}}\{h(k) = h(k')\} \leq \frac{1}{m} \quad \text{for all} \quad k \neq k'$$

.

- now we just assume h is random, and make no assumption about input keys. (like Randomized Quicksort)

Theorem: For n arbitrary distinct keys and random $h \in \mathcal{H}$, where $\mathcal{H}$ is a universal hashing family,

$$E[\text{ number of keys colliding in a slot }] \leq 1 + \alpha \quad \text{where} \quad \alpha = \frac{n}{m}$$

Proof: Consider keys $k_1, k_2, \ldots, k_n$. Let $I_{i,j} = \begin{cases} 1 & \text{if } h(k_i) = h(k_j) \\ 0 & \text{otherwise} \end{cases}$. Then we have

$$\begin{aligned} E[I_{i,j}] &= \Pr\{I_{i,j} = 1\} \\ &= \Pr\{h(k_i) = h(k_j)\} \\ &\leq \frac{1}{m} \text{ for any } j \neq i \end{aligned} \tag{1}$$

$$\begin{aligned} E[\#\text{ keys hashing to the same slot as } k_i] &= E\left[\sum_{j=1}^{n} I_{i,j}\right] \\ &= \sum_{j=1}^{n} E[I_{i,j}] \text{ (linearity of expectation)} \\ &= \sum_{j \neq i} E[I_{i,j}] + E[I_{i,i}] \\ &\leq \frac{n}{m} + 1 \end{aligned} \tag{2}$$

□

From the above theorem, we know that Insert, Delete, and Search all take $O(1+\alpha)$ expected time. Here we give some examples of universal hash functions.

All hash functions: $\mathcal{H} = \{\text{all hash functions } h : \{0, 1, \ldots, u-1\} \to \{0, 1, \ldots, m-1\}\}$. Apparently, $\mathcal{H}$ is universal, but it is useless. On one hand, storing a single hashing function h takes $\log(m^u) = u \log(m)$ bits $\gg n$ bits. On the other hand, we would need to precompute u values, which takes $\Omega(u)$ time.

Dot-product hash family:
Assumptions

- m is a prime

- $u = m^r$ where r is an integer

In real cases, we can always round up m and u to satisfy the above assumptions. Now let's view keys in base m: $k = \langle k_0, k_1, \ldots, k_{r-1} \rangle$. For key $a = \langle a_0, a_1, a_2, \ldots, a_{r-1} \rangle$, define

$$\begin{aligned} h_a(k) &= a \cdot k \bmod m \text{ (dot product)} \\ &= \sum_{i=0}^{r-1} a_i k_i \bmod m \end{aligned} \tag{3}$$

Then our hash family is $\mathcal{H} = \{h_a \mid a \in \{0, 1, \ldots, u-1\}\}$

Storing $h_a \in \mathcal{H}$ requires just storing one key, which is a. In the **word RAM model**, manipulating $O(1)$ machine words takes $O(1)$ time and "objects of interest" (here, keys) fit into a machine word. Thus computing $h_a(k)$ takes $O(1)$ time.

Theorem: Dot-product hash family $\mathcal{H}$ is universal.

Proof: Take any two keys $k \neq k'$. They must differ in some digits. Say $k_d \neq k_{d'}$. Define *not* $d = \{0, 1, \ldots, r-1\} \setminus \{d\}$. Now we have

$$\begin{aligned} \Pr_a\{h_a(k) = h_a(k')\} &= \Pr_a\left\{\sum_{i=0}^{r-1} a_i k_i = \sum_{i=0}^{r-1} a_i k'_i \pmod m\right\} \\ &= \Pr_a\left\{\sum_{i \neq d} a_i k_i + a_d k_d = \sum_{i \neq d} a_i k'_i + a_d k'_d \pmod m\right\} \\ &= \Pr_a\left\{\sum_{i \neq d} a_i (k_i - k'_i) + a_d (k_d - k'_d) = 0 \pmod m\right\} \\ &= \Pr_a\left\{a_d = -(k_d - k'_d)^{-1} \sum_{i \neq d} a_i (k_i - k'_i) \pmod m\right\} \\ &\quad (m \text{ is prime} \Rightarrow \mathbb{Z}_m \text{ has multiplicative inverses}) \\ &= \mathop{E}_{a_{not\ d}} [\Pr_{a_d}\{a_d = f(k, k', a_{not\ d})\}] \\ &\quad (= \sum_x \Pr\{a_{not\ d} = x\} \Pr_{a_d}\{a_d = f(k, k', x)\}) \\ &= \mathop{E}_{a_{not\ d}} \left[\frac{1}{m}\right] \\ &= \frac{1}{m} \end{aligned} \tag{4}$$

□

Another universal hash family from CLRS: Choose prime $p \geq u$ (once). Define $h_{ab}(k) = [(ak + b) \bmod p)] \bmod m$. Let $\mathcal{H} = \{h_{ab} \mid a, b \in \{0, 1, \ldots, u - 1\}\}$.

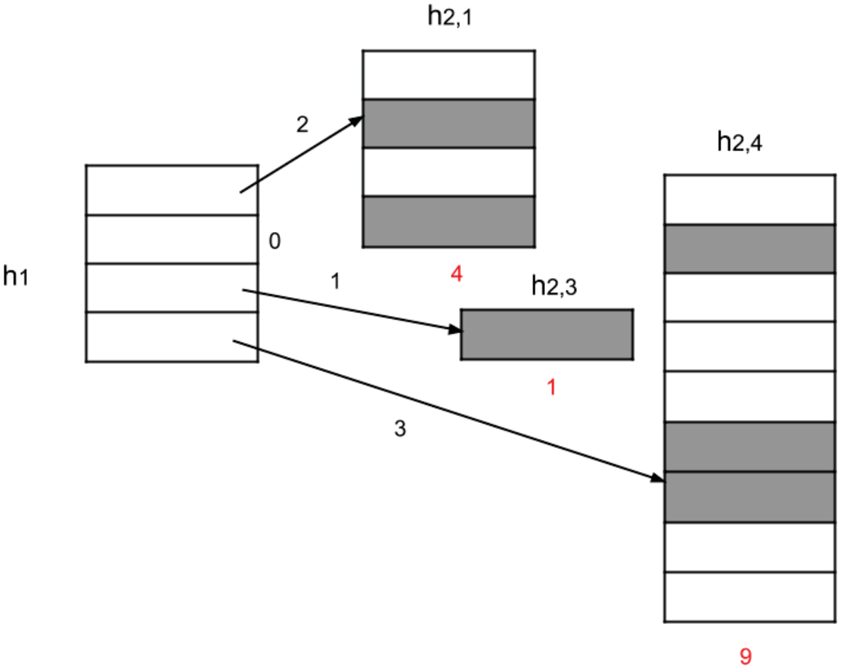
Perfect Hashing

Static dictionary problem: Given n keys to store in table, only need to support search(k). No insertion or deletion will happen.

Perfect hashing: [Fredman, Komlós, Szemerédi 1984]

- polynomial build time with high probability (w.h.p.)
- $O(1)$ time for search in worst case
- $O(n)$ space in worst case

Idea: 2-level hashing



The algorithm contains the following two major steps:

Step 1: Pick $h_1 : \{0, 1, \ldots, u - 1\} \rightarrow \{0, 1, \ldots, m - 1\}$ from a universal hash family for $m = \Theta(n)$ (e.g., nearby prime). Hash all items with chaining using h_1.

Step 2: For each slot $j \in \{0, 1, \ldots, m-1\}$, let l_j be the number of items in slot j. $l_j = |\{i \mid h(k_i) = j\}|$. Pick $h_{2,j} : \{0, 1, \ldots, u-1\} \to \{0, 1, \ldots, m_j\}$ from a universal hash family for $l_j^2 \leq m_j \leq O(l_j^2)$ (e.g., nearby prime). Replace chain in slot j with hashing-with-chaining using $h_{2,j}$.

The space complexity is $O(n + \sum_{j=0}^{m-1} l_j^2)$. In order to reduce it to $O(n)$, we need to add two more steps:

Step 1.5: If $\sum_{j=0}^{m-1} l_j^2 > cn$ where c is a chose constant, then redo Step 1.

Step 2.5: While $h_{2,j}(k_i) = h_{2,j}(k_i')$ for any $i \neq i', j$, repick $h_{2,j}$ and rehash those l_j.

The above two steps guarantee that there are no collisions at second level, and the space complexity is $O(n)$. As a result, search time is $O(1)$. Now let's look at the build time of the algorithm. Both Step 1 and Step 2 are $O(n)$. How about Step 1.5 and Step 2.5?

For Step 2.5,

$$\begin{aligned}
\Pr_{h_{2,j}}\{h_{2,j}(k_i) = h_{2,j}(k_i') \text{ for some } i \neq i'\} &\leq \sum_{i \neq i'} \Pr_{h_{2,j}}\{h_{2,j}(k_i) = h_{2,j}(k_i')\} \text{ (union bound)} \\
&\leq \binom{l_j}{2} \cdot \frac{1}{l_j^2} \\
&< \frac{1}{2}
\end{aligned} \tag{5}$$

As a result, each trial is like a coin flip. If the outcome is "tail", we move to the next step. By Lecture 7, we have $E[\#\text{trials}] \leq 2$ and $\#\text{trials} = O(\log n)$ w.h.p. By a Chernoff bound, $l_j = O(\log n)$ w.h.p., so each trial takes $O(\log n)$ time. Because we have to do this for each j, the total time complexity is $O(\log n) \cdot O(\log n) \cdot O(n) = O(n \log^2 n)$ w.h.p.

For Step 1.5, we define $I_{i,i'} = \begin{cases} 1 & \text{if } h(k_i) = h(k_i') \\ 0 & \text{otherwise} \end{cases}$. Then we have

$$\begin{aligned} E\left[\sum_{j=0}^{m-1} l_j^2\right] &= E\left[\sum_{i=1}^{n}\sum_{i'=1}^{n} I_{i,i'}\right] \\ &= \sum_{i=1}^{n}\sum_{i'=1}^{n} E[I_{i,i'}] \text{ (linearity of expectation)} \\ &\leq n + 2\binom{n}{2}\cdot\frac{1}{m} \\ &= O(n) \text{ because } m = \Theta(n) \end{aligned} \tag{6}$$

By Markov inequality, we have

$$\Pr_{h_1}\left\{\sum_{j=0}^{m-1} l_j^2 \leq cn\right\} \leq \frac{\sum_{j=0}^{m-1} l_j^2}{cn} \leq \frac{1}{2}$$

for a sufficiently large constant c. By Lecture 7, we have $E[\#\text{trials}] \leq 2$ and $\#\text{trials} = O(\log n)$ w.h.p. As a result, Step 1 and Step 1.5 combined takes $O(n \log n)$ time w.h.p.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

Lecture 9: Augmentation

This lecture covers augmentation of data structures, including

- easy tree augmentation
- order-statistics trees
- finger search trees, and
- range trees

The main idea is to modify "off-the-shelf" common data structures to store (and update) additional information.

Easy Tree Augmentation

The goal here is to store $x.f$ at each node x, which is a function of the node, namely f(subtree rooted at x). Suppose $x.f$ can be computed (updated) in $O(1)$ time from x, *children* and *children.f*. Then, modification a set S of nodes costs $O(\#$ of ancestors of $S)$ to update $x.f$, because we need to walk up the tree to the root. Two examples of $O(\lg n)$ updates are

- AVL trees: after rotating two nodes, first update the new bottom node and then update the new top node
- 2-3 trees: after splitting a node, update the two new nodes.
- In both cases, then update up the tree.

Order-Statistics Trees (from 6.006)

The goal of order-statistics trees is to design an Abstract Data Type (ADT) interface that supports the following operations

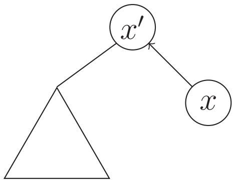
- insert(x), delete(x), successor(x),
- rank(x): find x's index in the sorted order, i.e., $\#$ of elements $< x$,
- select(i): find the element with rank i.

We can implement the above ADT using easy tree augmentation on AVL trees (or 2-3 trees) to store subtree size: $f(\text{subtree}) = \#$ of nodes in it. Then we also have $x.size = 1 + \sum c.size$ for c in $x.children$.

As a comparison, we cannot store the rank for each node. In that case, insert$(-\infty)$ will change the ranks for *all* nodes.

rank(x) can be computed as follows:

- initialize rank $= x.left.size + 1$ [1]
- walk up from x to root, whenever taking a left move $(x \to x')$, rank += $x'.left.size + 1$



select(i) can be implemented as follows:

- $x =$ root
- rank $= x.left.size + 1$ [2]
- if $i =$ rank: return x
- if $i <$ rank: $x = x.left$
- if $i >$ rank: $x = x.right$, i $-=$ rank

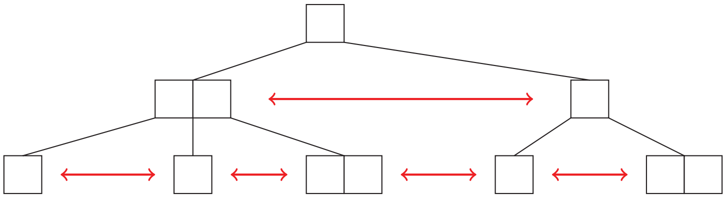
Finger Search Trees

The goal of finger search trees [Brown and Tarjan, 1980] is that, if we already have node y, we want to search x from y in $O(\lg |\text{rank}(y) - \text{rank}(x)|)$ time. Intuitively, we would like the search of x to be fast if we already have a node y that is close to x.

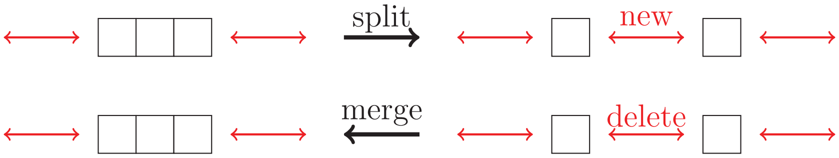
One idea is to use **level-linked** 2-3 trees, where each node has pointers to its next and previous node on the same level.

[1] omit the +1 term if indices start at 0.

[2] same as above.



The level links can be maintained during split and merge.



We store all keys in the leaves. Non-leaf nodes do not store keys; instead, they store the min and max key of the subtree (via easy tree augmentation).

Then the original top-down search(x) (without being given y) can be implemented as follows:

- start from the root, look at min & max of each child c_i
- if $c_i.\min \leq x \leq c_i.\max$, go down to c_i
- if $c_i.\max \leq x \leq c_{i+1}.\min$, return $c_i.\max$ (as predecessor) or $c_{i+1}.\min$ (as successor)

search(x) from y can be implemented as follows. Initialize v to the leaf node containing y (given), and then in a loop do

- if $v.\min \leq x \leq v.\max$ (this means x is in the subtree rooted at v), do top-down search for x from v and return
- elif $x < v.\min$: $v = v.prev$ (the previous node in this level)
- elif $x > v.\max$: $v = v.next$ (the next node in this level)
- $v = v.parent$

Analysis. We start at the leaf level, and go up by 1 level in each iteration. At step i, level link at height i skips roughly c^i keys (ranks), where $c \in [2, 3]$. Therefore, if $|\text{rank}(y) - \text{rank}(x)| = k$, we will reach the subtree containing x in $O(\lg k)$ steps, and the top-down search that follows is also $O(\lg k)$.

Orthogonal Range Searching and Range Trees

Suppose we have n points in a d-dimension space. We would like a data structure that supports **range query** on these points: find all the points in a give **axis-aligned box**. An axis-aligned box is simply an interval in 1D, a rectangle in 2D, and a cube in 3D.

To be more precise, each point x_i (for i from 1 to n) is a d-dimension vector $x_i = (x_{i1}, x_{i2}, \ldots, x_{id})$. Range-query$(a, b)$ takes two points $a = (a_1, a_2, \ldots, a_d)$ and $b = (b_1, b_2, \ldots, b_d)$ as input, and should return a set of indices $\{i \mid \forall j, a_j \leq x_{ij} \leq b_j\}$.

1D case

We start with the simple case of 1D points, i.e., all x_i's and a and b are scalars.

Then, we can simply use a sorted array. To do range-query(a, b), we simply perform two binary searches for a and b, respectively, and then return all the points in between (say there are k of them). The complexity is $O(\lg n + k)$.

Sorted arrays are inefficient for insertion and deletion. For a dynamic data structure that supports range queries, we can use finger search tree from the previous section. Finger search trees support efficient insertion and deletion. To do range-query(a, b), we first search for a, and then keep doing finger search to the right by 1 until we exceed b. Each finger search by 1 takes $O(1)$, so the total complexity is also $O(\lg n + k)$.

However, neither of the above approaches generalizes to high dimensions. That's why we now introduce range trees.

1D range trees

A 1D range tree is a complete binary search tree (for dynamic, use an AVL tree). Range-query(a, b) can be implemented as follows:

- search(a)
- search(b)
- find the least common ancestor (LCA) of a and b, v_{split}
- return the nodes and subtrees "in between". There are $O(\lg n)$ nodes and $O(\lg n)$ subtrees "in between".

Analysis. $O(\lg n)$ to implicitly represent the answer. $O(\lg n + k)$ to output all k answers. $O(\lg n)$ to report k via subtree size augmentation.

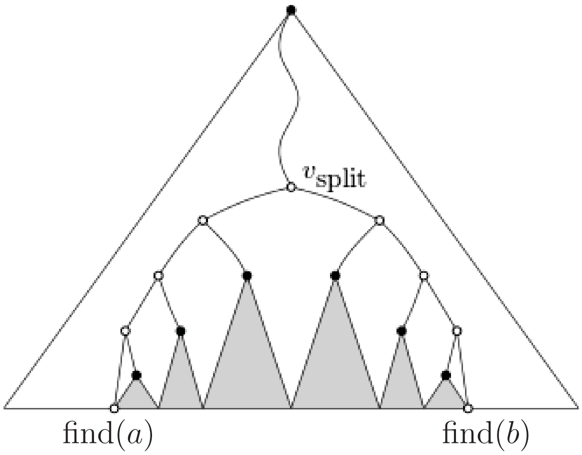


Figure 1: 1D range tree. Range-query(a, b) returns all hollow nodes and shaded subtrees. Image from Wikipedia http://en.wikipedia.org/wiki/Range_tree

2D range trees

A 2D range tree consists of a primary 1D range tree and many secondary 1D range trees. The primary range stores all points, keyed on the first coordinate. Every node v in the primary range tree stores all points in v's subtree in a secondary range tree, keyed on the second coordinate.

Range-query(a, b) can be implemented as follows:

- use the primary range tree to find all points with the correct range on the first coordinate. Only implicitly represent the answer, so this takes $O(\lg n)$.
- for the $O(\lg n)$ nodes, manually check whether their second coordinate lie in the correct range.
- for the $O(\lg n)$ subtrees, use their secondary range tree to find all points with the correct range on the second coordinate.

Analysis. $O(\lg^2 n)$ to implicitly represent the answer, because we will find $O(\lg^2 n)$ nodes and subtrees in secondary range trees. $O(\lg^2 n + k)$ to output all k answers. $O(\lg^2 n)$ to report k via subtree size augmentation.

Space complexity is $O(n \lg n)$. The primary subtree is $O(n)$. Each point is duplicated up to $O(\lg n)$ times in secondary subtrees, one per ancestor.

d-D range trees

Just recurse: primary 1D range tree → secondary 1D range trees → tertiary 1D range trees → $\cdots$

Range-query complexity: $O(\lg^d n + k)$.
Space complexity: $O(n \lg^{d-1} n)$.

See 6.851 for Chazelle's improved results: $O(\lg^{d-1} n + k)$ range-query complexity and $O\left(n \left(\frac{\lg n}{\lg \lg n}\right)^{d-1}\right)$ space complexity.

Lecture 10: Dynamic Programming

- Longest palindromic sequence
- Optimal binary search tree
- Alternating coin game

DP notions

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution based on optimal solutions of subproblems
3. Compute the value of an optimal solution in bottom-up fashion (recursion & memoization)
4. Construct an optimal solution from the computed information

Longest Palindromic Sequence

Definition: A palindrome is a string that is unchanged when reversed.
Examples: radar, civic, t, bb, redder
Given: A string $X[1 \cdots n]$, $n \geq 1$
To find: Longest palindrome that is a subsequence
Example: Given "c h a r a c t e r"
output "c a r a c"
Answer will be ≥ 1 in length

Strategy

$L(i, j)$: length of longest palindromic subsequence of $X[i \cdots j]$ for $i \leq j$.

```
def L(i, j) :
if i == j: return 1
if X[i] == X[j]:
        if i + 1 == j: return 2
        else : return 2 + L(i + 1, j - 1)
else :
        return max(L(i + 1, j), L(i, j - 1))
```

Exercise: compute the actual solution

Analysis

As written, program can run in exponential time: suppose all symbols $X[i]$ are distinct.

$T(n)$ = running time on input of length n

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) & n > 1 \end{cases}$$
$$= 2^{n-1}$$

Subproblems

But there are only $\begin{pmatrix} n \\ 2 \end{pmatrix} = \theta(n^2)$ distinct subproblems: each is an (i, j) pair with $i < j$. By solving each subproblem only once. running time reduces to

$$\theta(n^2) \cdot \theta(1) = \theta(n^2)$$

where $\theta(n^2)$ is the number of subproblems and $\theta(1)$ is the time to solve each subproblem, given that smaller ones are solved.

Memoize $L(i, j)$, hash inputs to get output value, and lookup hash table to see if the subproblem is already solved, else recurse.

Memoizing Vs. Iterating

1. Memoizing uses a dictionary for $L(i, j)$ where value of L is looked up by using i, j as a key. Could just use a 2-D array here where null entries signify that the problem has not yet been solved.

2. Can solve subproblems in order of increasing $j - i$ so smaller ones are solved first.

Optimal Binary Search Trees: CLRS 15.5

Given: keys $K_1, K_2, \cdots, K_n, K_1 < K_2 < \cdots < K_n$, WLOG $K_i = i$

weights $W_1, W_2, \cdots, W_n$

Find: BST T that minimizes:

$$\sum_{i=1}^{n} W_i \cdot (depth_T(K_i) + 1)$$

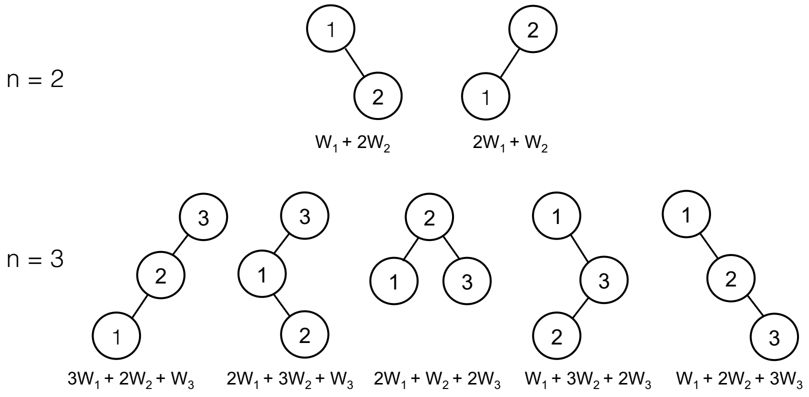
Example: $W_i = p_i =$ probability of searching for K_i

Then, we are minimizing expected search cost.

(say we are representing an English → French dictionary and common words should have greater weight)

Enumeration

Exponentially many trees



Strategy

$W(i, j) = W_i + W_{i+1} + \cdots + W_j$

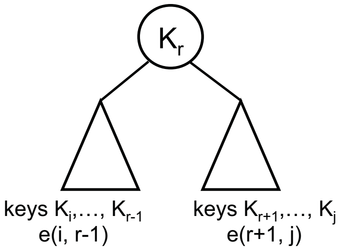
$e(i, j) =$ cost of optimal BST on $K_i, K_{i+1}, \cdots, K_j$

Want $e(1, n)$

Greedy solution?

Pick K_r in some greedy fashion, e.g., W_r is maximum.

greedy doesn't work, see example at the end of the notes.



DP Strategy: Guess all roots

$$e(i,j) = \begin{cases} W_i & \text{if } i = j \\ \min_{i \le r \le j} (e(i, r-1) + e(r+1, j) + W(i,j)) & \text{else} \end{cases}$$

$+W(i,j)$ accounts for W_r of root K_r as well as the increase in depth by 1 of all the other keys in the subtrees of K_r (DP tries all ways of making local choices and takes advantage of overlapping subproblems)

Complexity: $\theta(n^2) \cdot \theta(n) = \theta(n^3)$

where $\theta(n^2)$ is the number of subproblems and $\theta(n)$ is the time per subproblem.

Alternating Coin Game

Row of n coins of values $V_1, \cdots, V_n$, n is even. In each turn, a player selects either the first or last coin from the row, removes it permanently, and receives the value of the coin.

Question

Can the first player always win?

Try: 4 42 39 17 25 6

Strategy

$V_1, V_2, \cdots, V_{n-1}, V_n$

1. Compare $V_1 + V_3 + \cdots + V_{n-1}$ against $V_2 + V_4 + \cdots + V_n$ and pick whichever is greater.
2. During the game only pick from the chosen subset (you will always be able to!)

How to maximize the amount of money won assuming you move first?

Optimal Strategy

$V(i,j)$: max value we can definitely win if it is our turn and only coins $V_i, \cdots, V_j$ remain.

$V(i,i)$: just pick i.

$V(i,i+1)$: pick the maximum of the two.

$V(i,i+2)$, $V(i,i+3), \cdots$

$$V(i,j) = max\{\langle \text{range becomes } (i+1,j)\rangle + V_i, \langle \text{range becomes } (i,j-1)\rangle + V_j\}$$

Solution

$V(i+1,j)$ subproblem with opponent picking

we are guaranteed $min\{V(i+1,j-1), V(i+2,j)\}$

Where $V(i+1,j-1)$ corresponds to opponent picking V_j and $V(i+2,j)$ corresponds to opponent picking V_{i+1}

We have

$$V(i,j) = max\left\{ min\left\{ \begin{array}{c} V(i+1,j-1), \\ V(i+2,j) \end{array} \right\} + V_i, min\left\{ \begin{array}{c} V(i,j-2), \\ V(i+1,j-1) \end{array} \right\} + V_j \right\}$$

Complexity?

$$\Theta(n^2)\cdot\Theta(1) = \Theta(n^2)$$

Example of Greedy Failing for Optimal BST problem

Thanks to Nick Davis!

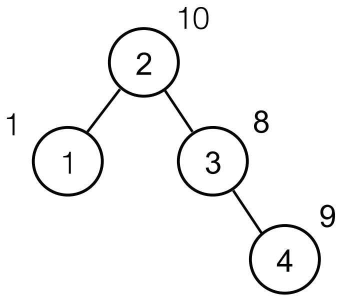


Figure 1: cost $= 1 \times 2 + 10 \times 1 + 8 \times 2 + 9 \times 3 = 55$

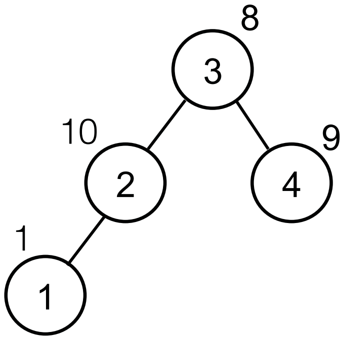


Figure 2: cost $= 1 \times 3 + 10 \times 2 + 8 \times 1 + 9 \times 2 = 49$

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

Lecture 11: All-Pairs Shortest Paths

Introduction

Different types of algorithms can be used to solve the all-pairs shortest paths problem:

- Dynamic programming
- Matrix multiplication
- Floyd-Warshall algorithm
- Johnson's algorithm
- Difference constraints

Single-source shortest paths

- given directed graph $G = (V, E)$, vertex $s \in V$ and edge weights $w : E \to \mathbb{R}$
- find $\delta(s, v)$, equal to the shortest-path weight $s- > v, \forall v \in V$ (or $-\infty$ if negative weight cycle along the way, or ∞ if no path)

Situtation	**Algorithm**	**Time**
unweighted ($w = 1$)	BFS	$O(V + E)$
non-negative edge weights	Dijkstra	$O(E + V \lg V)$
general	Bellman-Ford	$O(VE)$
acyclic graph (DAG)	Topological sort + one pass of B-F	$O(V + E)$

All of the above results are the best known. We achieve a $O(E + V \lg V)$ bound on Dijkstra's algorithm using Fibonacci heaps.

All-pairs shortest paths

- given edge-weighted graph, $G = (V, E, w)$
- find $\delta(u, v)$ for all $u, v \in V$

A simple way of solving All-Pairs Shortest Paths (APSP) problems is by running a single-source shortest path algorithm from each of the V vertices in the graph.

Situtation	**Algorithm**	**Time**	$E = \Theta(V^2)$
unweighted ($w = 1$)	$\lvert V\rvert\times$ BFS	$O(VE)$	$O(V^3)$
non-negative edge weights	$\lvert V\rvert\times$ Dijkstra	$O(VE + V^2 \lg V)$	$O(V^3)$
general	$\lvert V\rvert\times$ Bellman-Ford	$O(V^2E)$	$O(V^4)$
general	Johnson's	$O(VE + V^2 \lg V)$	$O(V^3)$

These results (apart from the third) are also best known — don't know how to beat $|V|\times$ Dijkstra

Algorithms to solve APSP

Note that for all the algorithms described below, we assume that $w(u, v) = \infty$ if $(u, v) \notin E$.

Dynamic Programming, attempt 1

1. **Sub-problems:** $d_{uv}^{(m)}$ = weight of shortest path $u \to v$ using $\leq m$ edges

2. **Guessing:** What's the last edge (x, v)?

3. **Recurrence:**

$$d_{uv}^{(m)} = \min(d_{ux}^{(m-1)} + w(x, v) \text{ for } x \in V)$$

$$d_{uv}^{(0)} = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{otherwise} \end{cases}$$

4. **Topological ordering:** for $m = 0, 1, 2, \ldots, n-1$: for u and v in V:

5. **Original problem:**

   If graph contains no negative-weight cycles (by Bellman-Ford analysis), then shortest path is simple $\Rightarrow \delta(u, v) = d_{uv}^{(n-1)} = d_{uv}^{(n)} = \cdots$

Time complexity

In this Dynamic Program, we have $O(V^3)$ total sub-problems.

Each sub-problem takes $O(V)$ time to solve, since we need to consider V possible choices. This gives a total runtime complexity of $O(V^4)$.

Note that this is no better than $|V|\times$ Bellman-Ford

Bottom-up via relaxation steps

```
for $m = 1$ to $n$ by 1
    for $u$ in $V$
        for $v$ in $V$
            for $x$ in $V$
                if $d_{uv} > d_{ux} + d_{xv}$
                    $d_{uv} = d_{ux} + d_{xv}$
```

In the above pseudocode, we omit superscripts because more relaxation can never hurt.

Note that we can change our relaxation step to $d_{uv}^{(m)} = \min(d_{ux}^{\lceil m/2 \rceil} + d_{xv}^{\lceil m/2 \rceil} \text{ for } x \in V)$. This change would produce an overall running time of $O(n^3 \lg n)$ time. (student suggestion)

Matrix multiplication

Recall the task of standard matrix multiplication,

Given $n \times n$ matrices A and B, compute $C = A \cdot B$, such that $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$.

- $O(n^3)$ using standard algorithm
- $O(n^{2.807})$ using Strassen's algorithm
- $O(n^{2.376})$ using Coppersmith-Winograd algorithm
- $O(n^{2.3728})$ using Vassilevska Williams algorithm

Connection to shortest paths

- Define $\oplus = \min$ and $\odot = +$
- Then, C $= A \odot B$ produces $c_{ij} = \min_k(a_{ik} + b_{kj})$
- Define $D^{(m)} = (d_{ij}^{(m)})$, $W = (w(i,j))$, $V = \{1, 2, \ldots, n\}$

With the above definitions, we see that $D^{(m)}$ can be expressed as $D^{(m-1)} \odot W$. In other words, $D^{(m)}$ can be expressed as the circle-multiplication of W with itself m times.

Matrix multiplication algorithm

- $n-2$ multiplications $\Rightarrow O(n^4)$ time (stil no better)
- Repeated squaring: $((W^2)^2)^{2\cdots} = W^{2^{\lg n}} = W^{n-1} = (\delta(i,j))$ if no negative-weight cycles. Time complexity of this algorithm is now $O(n^3 \lg n)$.

We can't use Strassen, etc. since our new multiplication and addition operations don't support negation.

Floyd-Warshall: Dynamic Programming, attempt 2

1. **Sub-problems:** $c_{uv}^{(k)}$ = weight of shortest path $u \to v$ whose intermediate vertices $\in \{1, 2, \ldots, k\}$
2. **Guessing:** Does shortest path use vertex k?
3. **Recurrence:**
$$c_{uv}^{(k)} = \min(c_{uv}^{(k-1)}, c_{uk}^{(k-1)} + c_{kv}^{(k-1)})$$
$$c_{uv}^{(0)} = w(u,v)$$
4. **Topological ordering:** for k: for u and v in V:
5. **Original problem:** $\delta(u,v) = c_{uv}^{(n)}$. Negative weight cycle $\Leftrightarrow$ negative $c_{uu}^{(n)}$

Time complexity

This Dynamic Program contains $O(V^3)$ problems as well. However, in this case, it takes only $O(1)$ time to solve each sub-problem, which means that the total runtime of this algorithm is $O(V^3)$.

Bottom up via relaxation

```
C = (w(u,v))
for k = 1 to n by 1
    for u in V
        for v in V
            if c_uv > c_uk + c_kv
                c_uv = c_uk + c_kv
```

As before, we choose to ignore subscripts.

Johnson's algorithm

1. Find function $h : V \to \mathbb{R}$ such that $w_h(u,v) = w(u,v) + h(u) - h(v) \geq 0$ for all $u, v \in V$ or determine that a negative-weight cycle exists.

2. Run Dijkstra's algorithm on (V, E, w_h) from every source vertex $s \in V \Rightarrow$ get $\delta_h(u,v)$ for all $u, v \in V$

3. Given $\delta_h(u,v)$, it is easy to compute $\delta(u,v)$

   Claim. $\delta(u,v) = \delta_h(u,v) - h(u) + h(v)$

   Proof. Look at any $u \to v$ path p in the graph G

   - Say p is $v_0 \to v_1 \to v_2 \to \cdots \to v_k$, where $v_0 = u$ and $v_k = v$.

   $$\begin{aligned} w_h(p) &= \sum_{i=1}^{k} w_h(v_{i-1}, v_i) \\ &= \sum_{i=1}^{k} [w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)] \\ &= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\ &= w(p) + h(u) - h(v) \end{aligned}$$

   - Hence all $u \to v$ paths change in weight by the same offset $h(u) - h(v)$, which implies that the shortest path is preserved (but offset).

   □

How to find h?

We know that

$$w_h(u,v) = w(u,v) + h(u) - h(v) \geq 0$$

This is equivalent to,

$$h(v) - h(u) \leq w(u,v)$$

for all $(u,v) \in V$. This is called a **system of difference constraints**.

Theorem. *If (V, E, w) has a negative-weight cycle, then there exists no solution to the above system of difference constraints.*

Proof. Say $v_0 \to v_1 \to \cdots \to v_k \to v_0$ is a negative weight cycle.

Let us assume to the contrary that the system of difference constraints has a solution; let's call it h.

This gives us the following system of equations,

$$\begin{aligned}
h(v_1) - h(v_0) &\le w(v_0, v_1) \\
h(v_2) - h(v_1) &\le w(v_1, v_2) \\
&\vdots \\
h(v_k) - h(v_{k-1}) &\le w(v_{k-1}, v_k) \\
h(v_0) - h(v_k) &\le w(v_k, v_0)
\end{aligned}$$

Summing all these equations gives us

$$0 \le w(\text{cycle}) < 0$$

which is obviously not possible.

From this, we can conclude that no solution to the above system of difference constraints exists if the graph (V, E, w) has a negative weight cycle.

□

Theorem. *If (V, E, w) has no negative-weight cycle, then we can find a solution to the difference constraints.*

Proof. Add a new vertex s to G, and add edges (s, v) of weight 0 for all $v \in V$.

- Clearly, these new edges do not introduce any new negative weight cycles to the graph
- Adding these new edges ensures that there now exists at least one path from s to v. This implies that $\delta(s, v)$ is finite for all $v \in V$
- We now claim that $h(v) = \delta(s, v)$. This is obvious from the triangle inequality: $\delta(s, u) + w(u, v) \ge \delta(s, v) \Leftrightarrow \delta(s, v) - \delta(s, u) \le w(u, v) \Leftrightarrow h(v) - h(u) \le w(u, v)$

□

Time complexity

1. The first step involves running Bellman-Ford from s, which takes $O(VE)$ time. We also pay a pre-processing cost to reweight all the edges ($O(E)$)

2. We then run Dijkstra's algorithm from each of the V vertices in the graph; the total time complexity of this step is $O(VE + V^2 \lg V)$

3. We then need to reweight the shortest paths for each pair; this takes $O(V^2)$ time.

The total running time of this algorithm is $O(VE + V^2 \lg V)$.

Applications

Bellman-Ford consult any system of difference constraints (or report that it is unsolvable) in $O(VE)$ time where V = variables and E = constraints.

An exercise is to prove the Bellman-Ford minimizes $\max_i x_i - \min_i x_i$.

This has applications to

- Real-time programming
- Multimedia scheduling
- Temporal reasoning

For example, you can bound the duration of an event via difference constraint $LB \leq t_{end} - t_{start} \leq UB$, or bound a gap between events via $0 \leq t_{start2} - t_{end1} \leq \varepsilon$, or synchronize events via $|t_{start1} - t_{start2}| \leq \varepsilon$ or 0.

Lecture 12: Greedy Algorithms and Minimum Spanning Tree

Introduction

- Optimal Substructure
- Greedy Choice Property
- Prim's algorithm
- Kruskal's algorithm

Definitions

Recall that a *greedy algorithm* repeatedly makes a locally best choice or decision, but ignores the effects of the future.

A *tree* is a connected, acyclic graph.

A *spanning tree* of a graph G is a subset of the edges of G that form a tree and include all vertices of G.

Finally, the *Minimum Spanning Tree* problem: Given an undirected graph $G = (V, E)$ and edge weights $W : E \to \mathbb{R}$, find a spanning tree T of minimum weight $\sum_{e \in T} w(e)$.

A naive algorithm

The obvious MST algorithm is to compute the weight of every tree, and return the tree of minimum weight. Unfortunately, this can take exponential time in the worst case. Consider the following example:

If we take the top two edges of the graph, the minimum spanning tree can consist of any combination of the left and right edges that connect the middle vertices to the left and right vertices. Thus in the worst case, there can be an exponential number of spanning trees.

Instead, we consider greedy algorithms and dynamic programming algorithms to solve MST. We will see that greedy algorithms can solve MST in nearly linear time.

Properties of Greedy Algorithms

Problems that can be solved by greedy algorithms have two main properties:

- Optimal Substructure: the optimal solution to a problem incorporates the optimal solution to subproblem(s)
- Greedy choice property: locally optimal choices lead to a globally optimal solution

We can see how these properties can be applied to the MST problem

Optimal substructure for MST

Consider an edge $e = \{u, v\}$, which is an edge of some MST. Then we can contract e by merging the vertices u and v to create a new vertex. Then any edge adjacent to vertex u or v is adjacent to the newly created vertex, and the process could result in a multiedge if u and v have a mutual neighbor. We resolve the multiedge problem by creating a single edge with the minimum weight between the two edges.

This leads us to the following lemma:

Lemma 1. *If T' is a minimum spanning tree of G/e, then $T' \cup \{e\}$ is an MST of G.*

Proof. Let T^* be an MST of G containing the edge e. Then T^*/e is a spanning tree of G' = G/$\{e\}$. By definition, T' is an MST of G'. Thus the total weight of T' is less than or equal to that of T^*/e, or w(T') $\leq w(T^*$/e). Then

$$w(T) = w(T') + w(e) \leq w(T^*/e) + w(e) = w(T')$$

. □

The statement can be used as the basis for a dynamic programming algorithm, in which we guess an edge that belongs to the MST, retract the edge, and recurse. At the end, we decontract the edge and add e to the MST.

The lemma guarantees that this algorithm is correct. However, this algorithm is requires exponential time, because there are an exponential number of edges that we can guess to form our MST.

We make the algorithm polynomial time by removing the guessing process.

Greedy Choice Property

The MST problem can be solved by a greedy algorithm because the the locally optimal solution is also the globally optimal solution. This fact is described by the Greedy-Choice Property for MSTs, and its proof of correctness is given via a "cut and paste" argument common for greedy proofs.

Lemma 2 (Greedy-Choice Property for MST)**.** *For any cut* $(S, V \setminus S)$ *in a graph* $G = (V, E, w)$*, any least-weight crossing edge* $e = \{u, v\}$ *with* $u \in S$ *and* $v \notin S$ *is in some MST of* G*.*

Proof. First, consider an MST T of G. Then T has a path from u to v. Because $u \in S$ and $v \notin S$, the path has some edge $e' = \{u', v'\}$ which also crosses the cut. Then $T' = T \setminus \{e'\} \cup \{e\}$ is a spanning tree of G and $w(T') = w(T) - w(e') + w(e)$, but e is a least-weight edge crossing $(S, V \setminus S)$. Thus $w(e) \leq w(e')$, so $w(T') \leq w(T)$. Therefore T' is an MST too. □

Prim's Algorithm

Now, we can apply the insights from the optimal structure and greedy choice property to build a polynomial-time, greedy algorithm to solve the minimum spanning tree problem.

Prim's Algorithm Psuedocode

```
Maintain priority queue $Q$ on $V \setminus S$, where $v.key = \min\{w(u, v) \mid u \in S\}$
$Q = V$
Choose arbitrary start vertex $s \in V$, $s.key = \emptyset$
for $v$ in $V \setminus \{s\}$
    $v.key = \infty$
while $Q$ is not empty
    $u$ = Extract-Min($Q$), add $u$ to $S$
    for $v \in$ Adj[$u$]
        if $v \in Q$ and $v \notin S$ and $w(u, v) < v.key$:
        $v.key = w(u, v)$ (via a Decrease-Key operation)
        $v.parent = u$
return $\{\{v, v.parent\} \mid v \in V \setminus \{s\}\}$
```

In the above pseudocode, we choose an arbitrary start vertex, and attempt to sequentially reduce the distance to all vertices. After attempting to find the lowest weight edge to connect all vertices, we return our MST

Correctness

We prove the correctness of Prim's Algorithm with the following invariants.

1. $v \notin S \implies v.key = \min\{w(u, v) \mid u \in S\}$
2. Tree T_S within $S \subseteq$ MST of G.

The first invariant is follows from Step 8 of the algorithm above. A proof of the second invariant follows:

Proof. Assume by induction that $T_S \subseteq$ MST T^*. Then $S \rightarrow S' \cup \{e\}$, where e is a least-weight edge crossing the cut $(S, V \setminus S)$. Then we can greedily cut and paste e, which implies that we can modify T^* to include e without removing T_S, since the edges of T_S do not cross the cut. Therefore $T_S \cup \{e\} = T'_S \subseteq T^*$. □

Thus Prim's Algorithm always adds edges that have the lowest weight and gradually builds a tree that is always a subset of some MST, and returns a correct answer.

Runtime

Prim's algorithm runs in

$$O(V) \cdot T_{\text{Extract-Min}} + O(E) \cdot T_{\text{Decrease-Key}}$$

The $O(E)$ term results from the fact that Step 8 is repeated a number of times equal to the sum of the number of adjacent vertices in the graph, which is equal to $2|E|$, by the handshaking lemma.

Then the effective runtime of the algorithm varies with the data structures used to implement the algorithm. The table below describes the runtime with the different implementations of the priority queue.

Priority Queue	$T_{\text{Extract-Min}}$	$T_{\text{Decrease-Key}}$	Total
Array	$O(V)$	$O(1)$	$O(V^2)$
Binary heap	$O(\lg V)$	$O(\lg V)$	$O(E \lg V)$
Fibonacci heap [CLRS ch. 19]	$O(\lg V)$ (amortized)	$O(1)$ (amortized)	$O(E + V \lg V)$

Kruskal's Algorithm

Kruskal's Algorithm is another algorithm to solve the MST problem. It constructs an MST by taking the globally lowest-weight edge and contracting it.

Kruskal's Algorithm Pseudocode

```
Maintain connected components that have been added to the
MST so far $T$, in a Union-Find structure
Initialize $T = \emptyset$
**for** $v$ in $V$
    Make-Set($v$)
Sort $E$ by weight
For $e = (u, v) \in E$ (in increasing-weight order):
    **if** Find-Set($u$) $\neq$ Find-Set($v$):
        Add $e$ to $T$
        Union($u, v$)
```

Correctness

We use the following invariant to prove the correctness of Kruskal's Algorithm.

Claim 3. *The tree-so-far $T \subseteq$ MST T^*.*

Proof. We give an induction proof. We begin by assuming that the tree-so-far $T \subseteq T^*$, via the inductive hypothesis. When we add an edge e between some components C_1 and C_2, we use the greedy-choice property on the cut $(C_1, V \setminus C_2)$. Thus we have added the edge without removing T, and our new tree-so-far remains a subset of the MST T^*. □

Runtime

Kruskal's algorithm has an overall runtime of

$$T_{sort}(E) + O(V) \cdot T_{\text{Make-Set}} + O(E)(T_{\text{Find}} + T_{\text{Union}}) = O(E \lg E + E\alpha(V))$$

We note that $T_{\text{Make-Set}}$ is $O(1)$ and $T_{\text{find}} + T_{\text{Union}}$ is amortized $O(\alpha(V))$ for Union-Find data structures.

Furthermore, if all weights are integer weights, or all weights are in the range $[0, E^{O(1)}]$, then the runtime of the sorting step is $O(E)$, using Counting Sort or a similar algorithm, and the runtime of Kruskal's Algorithm will be better than that of Prim's Algorithm.

Other MST Algorithms

Currently, the fastest MST algorithm is a randomized algorithm with an expected runtime of $O(V + E)$. The algorithm was proposed by Karger, Klein, and Tarjan in 1993.

Lecture 14: Baseball Elimination

In this lecture, we will be studying how to use max-flow algorithms to compute elimination of sports teams in a league. Concretely, let us consider the standings of the AL Eastern Division of the Major League Baseball on August 30, 1996[1], without Detroit's record.

Table 1: Standings of AL East on August 30, 1996.

Team	Wins (w_i)	Losses (ℓ_i)	To Play (r_i)	Games against each other				
NY	75	59	28	–	5	7	4	3
Baltimore	71	63	28	5	–	2	4	4
Boston	69	65	28	7	2	–	4	0
Toronto	63	71	28	4	4	4	–	0
Detroit	?	?	28	3	4	0	0	–
				NY	Ba	Bo	T	D

From this chart, how can we figure out if a team is eliminated? A naive sports writer can only compute that Team i is eliminated if $w_i + r_i < w_j$ for some other j. For example, if Detroit's record was $w_5 = 46$, then Detroit is certainly eliminated since $w_5 + r_5 = 46 + 28 < 75 = w_1$.

This condition, however, is sufficient, but not necessary. For instance, consider $w_5 = 47$. Though $w_5 + r_5 = 75$, either NY or Baltimore will reach 76 wins since they have 5 games left against each other. How can we determine if Detroit is eliminated for arbitrary values of w_5?

To answer this question, we can use max-flow. Consider the Figure 1, where capacity between s and node $i - j$ is the number of games left to be played between team i and j, between node $i - j$ and node $k = 1, 2, 3, 4$ is infinity, and node k and t is $w_5 + r_5 - w_k$. The intuition for the construction of the graph is that we will assume Detroit win all r_5 games, and try to keep the number of wins per team to be less than or equal to the total possible wins of Detroit ($\leq w_5 + r_5$).

Theorem 1. *Team 5 (Detroit) is eliminated if and only if max-flow does not saturate all edges leaving the source, i.e., max flow value* < 26.

Proof. Saturation of the edge capacity corresponds to playing all the remaining games. If all the games *cannot* be played, while keeping the total number of wins of a team to be less than or equal to $w_5 + r_5$, then Team 5 is eliminated. □

[1] roughly speaking!

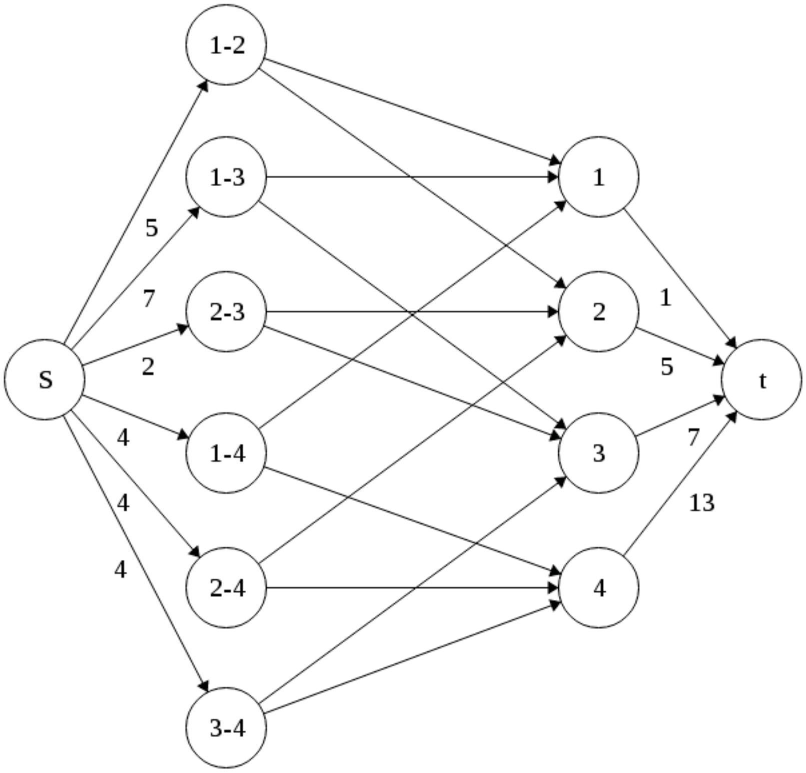


Figure 1: Flow network to determine if Team 5 is eliminated

In Figure 1, the min-cut (S, T) is $S = \{s, 1-2, 1-3, 2-3, 1, 2, 3\}$ and $T = \{1-4, 2-4, 3-4, 4, t\}$. The capacity of the min-cut $c(S, T) = 4+4+4+1+5+7$ $= 25 < 26$. Therefore, Team 5 Detroit is eliminated.

Alternate explanation: Note that the max-flow will find the subset of teams that eliminates Team 5. In this example, consider subset of teams 1,2, and 3. The total number of wins among the 3 teams is 215 wins, and they have 14 games left to play with each other. Then there will be 229 total wins at the end of the regular season. This implies that there exists at least one team that wins $\lceil \frac{229}{3} \rceil = 77$ games. Therefore, if Detroit only has 48 wins, then it is certainly eliminated $(48+28 = 76 < 77)$. We can find such set using max-flow and max-flow min-cut theorem.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

Lecture 15: Linear Programming

Linear programming (LP) is a method to achieve the optimum outcome under some requirements represented by linear relationships. More precisely, LP can solve the problem of maximizing or minimizing a linear objective function subject to some linear constraints.

In general, the standard form of LP consists of

- Variables: $\boldsymbol{x} = (x_1, x_2, \ldots, x_d)^\top$
- Objective function: $\boldsymbol{c} \cdot \boldsymbol{x}$
- Inequalities (constraints): $A\boldsymbol{x} \leq \boldsymbol{b}$, where A is a $n \times d$ matrix

and we maximize the objective function subject to the constraints and $\boldsymbol{x} \geq 0$.

LP has many different applications, such as flow, shortest paths, and even politics. In this lecture, we will be covering different examples of LP, and present an algorithm for solving them. We will also learn how to convert any LP to the standard form in this lecture.

1 Examples of Linear Programming: Politics

In this example, we will be studying how to campaign to win an election. In general, there are n demographics, each with p_i people, and m issues that the voters are interested in. Given the information on how many votes can be obtained per dollar spent advertising in support of an issue, how can we guarantee victory by ensuring a majority vote in all demographics?[1]

In particular, consider the example with 3 demographics and 4 issues shown in Table 1. What is the minimum amount of money we can spend to guarantee majority in all demographics?

Let x_1, x_2, x_3, x_4 denote the dollars spent per issue. We can now formulate this problem as an LP problem:

$$\begin{aligned}
\text{Minimize } & x_1 + x_2 + x_3 + x_4 \\
\text{Subject to } & -2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50,000 \text{ (Urban Majority)} && (1)\\
& 5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100,000 \text{ (Suburban Majority)} && (2)\\
& 3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25,000 \text{ (Rural Majority)} && (3)\\
& x_1, x_2, x_3, x_4 \geq 0 \text{ (Can't unadvertise)} && (4)
\end{aligned}$$

[1] We will assume that the votes obtained by advertising different issues are disjoint.

Policy	Demographic		
	Urban	Suburban	Rural
Building roads	-2	5	3
Gun Control	8	2	-5
Farm Subsidies	0	0	10
Gasoline Tax	10	0	2
Population	100,000	200,000	50,000

Table 1: Votes per dollar spent on advertising, and population.

1.1 Certificate of Optimality

Though we have not presented an algorithm for solving this problem, given a solution, we can verify that the solution is optimal with a proper certificate. For example,

$$\begin{aligned}
x_1 &= \frac{2050000}{111}\\
x_2 &= \frac{425000}{111}\\
x_3 &= 0\\
x_4 &= \frac{625000}{111}\\
x_1 + x_2 + x_3 + x_4 &= \frac{3100000}{111}
\end{aligned}$$

is a solution to this problem. Now consider the following equation (certificate):

$$\begin{aligned}
\frac{25}{222}\cdot(1) + \frac{46}{222}\cdot(2) + \frac{14}{222}\cdot(3) &= x_1 + x_2 + \frac{140}{222}x_3 + x_4\\
&\geq \frac{25}{222}\cdot 50000 + \frac{46}{222}\cdot 100000 + \frac{14}{222}\cdot 25000\\
&= \frac{3100000}{111}\\
\Rightarrow x_1 + x_2 + \frac{140}{222}x_3 + x_4 &\geq \frac{3100000}{111}
\end{aligned}$$

We also know that $x_1 + x_2 + x_3 + x_4 \geq x_1 + x_2 + \frac{140}{222}x_3 + x_4$. Therefore, the given solution is an optimal solution to the problem.

2 Linear Programming Duality

The short certificate provided in the last section is not a coincidence, but a consequence of ***duality*** of LP problems. For every primal LP problem in the form of

$$\begin{gathered}\text{Maximize } \boldsymbol{c}\cdot\boldsymbol{x}\\ \text{Subject to } A\boldsymbol{x}\le\boldsymbol{b},\boldsymbol{x}\ge 0,\end{gathered}$$

there exists an equivalent dual LP problem

$$\begin{gathered}\text{Minimize } \boldsymbol{b}\cdot\boldsymbol{y}\\ \text{Subject to } A^{\top}\boldsymbol{y}\ge\boldsymbol{c},\boldsymbol{y}\ge 0.\end{gathered}$$

This property of LP can be used show many important theorems. For instance, the max-flow min-cut theorem can be proven by formulating the max-flow problem as the primal LP problem.

3 Converting to Standard Form

The natural LP formulation of a problem may not result in the standard LP form. In these cases, we can convert the problem to standard LP form without affecting the answers by using the following rules.

- **Minimize an objective function**: Negate the coefficients and maximize.
- **Variable x_j does not have a non-negativity constraint**: Replace x_j with $x'_j - x''_j$, and $x'_j, x''_j \ge 0$.
- **Equality constraints**: Split into two different constraints; $x = b \Rightarrow x \le b, x \ge b$.
- **Greater than or equal to constraints**: Negate the coefficients, and translate to less than or equal to constraint.

4 Formulating LP Problems

In this section, we will give brief descriptions of how to formulate some problems seen previously in this class as LP problems. Once we have a LP formulation, we can convert the problem into the standard form as described in Section 3.

4.1 Maximum Flow

We can model the max flow problem as maximization of sum of flows, under some constraints which will model different properties of the flow. Given $G(V, E)$, the capacity $c(e)$ for each $e \in E$, the source s, and the sink t,

$$\begin{aligned} \text{Maximize } & \sum_{v \in V} f(s, v) \\ \text{Subject to } & f(u, v) = -f(v, u) \ \forall u, v \in V \ \text{ skew symmetry} \\ & \sum_{v \in V} f(u, v) = 0 \ \forall u \in V - \{s, t\} \ \text{ conservation} \\ & f(u, v) \leq c(u, v) \ \forall u, v \in V \ \text{ capacity.} \end{aligned}$$

4.2 Shortest Paths

We can model the shortest paths problem as minimization of the sum of all distances from a node. Note that this sum is minimized only when all distances are minimized. Given $G(V, E)$, the weight $w(e)$ for each $e \in E$, and the source s,

$$\begin{aligned} \text{Maximize } & \sum_{v \in V} d(v) \\ \text{Subject to } & d(v) - d(u) \leq w(u, v) \forall u, v \in V \text{ triangular inequality} \\ & \sum_{v \in V} d(s) = 0. \end{aligned}$$

Note the maximization above, so all distances don't end up being zero. There is no solution to this LP if and only if there exists a negative weight cycle reachable from s.

5 Algorithms for LP

There are many algorithms for solving LP problems:

- **Simplex algorithm**: In the feasible region, $\boldsymbol{x}$ moves from vertex to vertex in direction of $\boldsymbol{c}$. The algorithm is simple, but runs in exponential time in the worst case.
- **Ellipsoid algorithm**: It starts with an ellipsoid that includes the optimal solution, and keeps shrinking the ellipsoid until the optimal solution is found. This was the first poly-time algorithm, and was a theoretical breakthrough. However, the algorithm is impractical in practice.

- **Interior Point Method**: $\boldsymbol{x}$ moves inside the polytope following $\boldsymbol{c}$. This algorithm runs in poly-time, and is practical.

In this lecture, we will study only the simplex algorithm.

5.1 Simplex Algorithm

As mentioned before, the simplex algorithm works well in practice, but runs in exponential time in the worst case. At a high level, the algorithm works as Gaussian elimination on the inequalities or constraints. The simplex algorithm works as follows:

- Represent LP in "slack" form.
- Convert one slack form into an equivalent slack form, while likely increasing the value of the objective function, and ensuring that the value does not decrease.
- Repeat until the optimal solution becomes apparent.

5.1.1 Simplex Example

Consider the following example:

$$\begin{aligned} \text{Minimize } & 3x_1 + x_2 + x_3 \\ \text{Subject to } & x_1 + x_2 + 3x_2 \leq 30 \\ & 2x_1 + 2x_2 + 5x_3 \leq 24 \\ & 4x_1 + x_2 + 2x_3 \leq 36 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Change the given LP problem to slack form, consisting of the original variables called *nonbasic* variables, and new variables representing slack called *basic* variables.

$$\begin{aligned} z &= 3x_1 + x_2 + 2x_3 \\ x_4 &= 30 - x_1 - x_2 - 3x_3 \\ x_5 &= 24 - 2x_1 - 2x_2 - 5x_3 \\ x_6 &= 36 - 4x_1 - x_2 - 2x_3 \end{aligned}$$

We start with a *basic solution*: we set all nonbasic variables on the right hand side to some feasible value, and compute the values of the basic variables. For instance, we can set $x_1 = x_2 = x_3 = 0$. Note that the all 0 solution satisfies all constraints in this problem, but may not do so in the general case.

We now perform the pivoting step:

- Select a nonbasic variable x_e whose coefficient in the objective function is positive.
- Increase the value of x_e as much as possible without violating any constraints.
- Set x_e to be basic, while some other basic variable becomes nonbasic.

In this example, we can increase the value of x_1. The third constraint will limit the value of x_1 to 9. We then get

$$x_1 = 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4}.$$

Now rewrite the other constraints with x_6 on the right hand side.

$$\begin{aligned}
z &= 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \\
x_1 &= 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \\
x_4 &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \\
x_5 &= 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}
\end{aligned}$$

We note the equivalence of the solutions. That is, the original basic solution $(0, 0, 0, 30, 24, 36)$ satisfies the rewritten constraints, and has the objective value of 0. The second basic solution $(9, 0, 0, 21, 6, 0)$ has the objective value of 27.

At this point, pivoting on x_6 will actually cause the objective value to decrease (though the computation is not shown here). Thus let us pick x_3 as the next pivot to get

$$\begin{aligned}
z &= \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\
x_1 &= \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \\
x_2 &= \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + -\frac{x_6}{8} \\
x_4 &= \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}
\end{aligned}$$

which results in basic solution $(\frac{33}{4}, 0, \frac{3}{2}, \frac{69}{4}, 0, 0)$ with objective value of $\frac{111}{4}$.

Finally, pivoting on x_2 yields

$$\begin{aligned}
z &= 28 - \frac{x_3}{6} - \frac{x_5}{6} - \frac{2x_6}{3} \\
x_1 &= 8 + \frac{x_3}{6} + \frac{x_5}{6} - \frac{x_6}{3} \\
x_2 &= 4 - \frac{8x_3}{3} - \frac{2x_5}{3} + \frac{x_6}{3} \\
x_4 &= 18 - \frac{x_3}{2} + \frac{x_5}{2}
\end{aligned}$$

Though we will not prove the correctness of this algorithm in this lecture, when all coefficients of all nonbasic variables are negative, the simplex algorithm has found the optimal solution. In general, simplex algorithm is guaranteed to converge in $\binom{n+m}{n}$ iterations where n is the number of variables, and $n+m$ is the number of constraints. This for general n and m can be exponential.

6 More Topics of LP

There are several important questions regarding LP that were not discussed in this lecture:

- How do we determine if LP is feasible?
- What if LP is feasible, but the initial basic solution is infeasible?
- How do we determine if the LP is unbounded?
- How do we choose the pivot?

These questions are answered in the textbook and other LP literature.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

Lecture 16: NP-Completeness

Introduction

- NP-hardness and NP-completeness
- 3SAT
  - ↳ Super Mario Brothers
  - ↳ 3 Dimensional Matching
    - ↳ Subset Sum *(weak)*
      - ↳ Partition *(weak)*
        - ↳ Rectangle Packing *(weak)*
    - ↳ 4-Partition *(strong)*
      - ↳ Rectangle Packing *(strong)*
        - ↳ Jigsaw Puzzles

NP-Hard and NP-Complete problems

Today, we discuss NP-Completeness.

Recall from 6.006:

- **P** = the set of problems that are solvable in polynomial time. If the problem has size n, the problem should be solved in $n^{O(1)}$.
- **NP** = the set of decision problems solvable in nondeterministic polynomial time. The output of these problems is a YES or NO answer. Nondeterministic refers to the fact that a solution can be guessed out of polynomially many options in $O(1)$ time. If any guess is a YES instance, then the nondeterministic algorithm will make that guess.

In this model of nondeterminism, we can assume that all guessing is done first. This is equivalent to finding a polynomial-time verifier of polynomial-size certificates for YES answers.

Note that there is an asymmetry between YES and NO inputs

- A problem X is **NP-complete** if $X \in$NP and X is NP-hard.

- A problem X is NP-hard if every problem $Y \in$NP reduces to X.
    - If P $\neq$ NP, then $X \notin$ P.
- A **reduction** from problem A to problem B is a polynomial-time algorithm that converts inputs to problem A into equivalent inputs to problem B. Equivalent means that both problem A and problem B must output the same YES or NO answer for the input and converted input.
    - If $B \in$ P, then $A \in$ NP
    - If $B \in$ NP, then $A \in$ NP
    - If A is NP-hard, then B is NP-hard.

We can show that problems are NP-complete via the following steps.

1. **Show $X \in$ NP**. Show that $X \in$ NP by finding a nondeterministic algorithm, or giving a valid verifier for a certificate.
2. **Show X is NP-hard**. Reduce from a known NP-complete problem Y to X. This is sufficient because all problems $Z \in$ NP can be reduced to Y, and the reduction demonstrates inputs to Y can be modified to become inputs to X, which implies X is NP-hard. We must demonstrate the following properties for a complete reduction.
    (a) Give an polynomial-time conversion from Y inputs to X inputs.
    (b) If Y's answer is YES, then X's answer is YES.
    (c) If X's answer is YES, then Y's answer is YES.

Finally, a **gadget** transforms features in an input problem to a feature in an output problem.

3SAT

The 3SAT was discovered to be NP-complete by Cook in 1971.

Definition 1. ***3SAT:*** *Given a boolean formula of the form:*

$$(x_1 \vee x_3 \vee \bar{x_6}) \wedge (\bar{x_2} \vee x_3 \vee \bar{x_7}) \wedge \dots$$

is there an assignment of variables to True and False, such that the entire formula evaluates to True?

We note that a literal is of the form $\{x_i, \bar{x_i}\}$, and both forms of the literal correspond to the variable x_i. A clause is made up of the OR of 3 literals, and a formal is the AND of clauses.

3SAT $\in$ NP because we can create a verifier for a certificate. For a given instance of 3SAT, a certificate corresponds to a list of assignments for each variable, and a verifier can compute whether the instances is satisfied, or can be evaluated to true. Thus the verifier is polynomial time, and the certificate has polynomial length.

It is important to note that this verifier only guarantees that a 3SAT instance is verifiable. To ensure that a 3SAT instance is not verifiable, the algorithm would have to check every variable assignment, which cannot be done in polynomial time.

3SAT is also NP-hard. We give some intuition for this result. Consider any problem in NP. Because it belongs in NP, a nondeterministic polynomial time algorithm exists to solve this problem, or a verifier to check a solution. The verifier is an algorithm that can be implemented as a circuit. Now, the circuit consists of AND, OR and NOT gates, which can be represented as a formula. This formula can be converted to 3SAT form, where each clause has 3 literals, which is equivalent to the original formula. Thus all problems in NP can be converted to 3SAT, and the inputs to the original problem are equivalent to the converted inputs to 3SAT, thus 3SAT is NP-complete.

Super Mario Brothers [Aloupis, Demaine, Guo, Viglietta 2014]

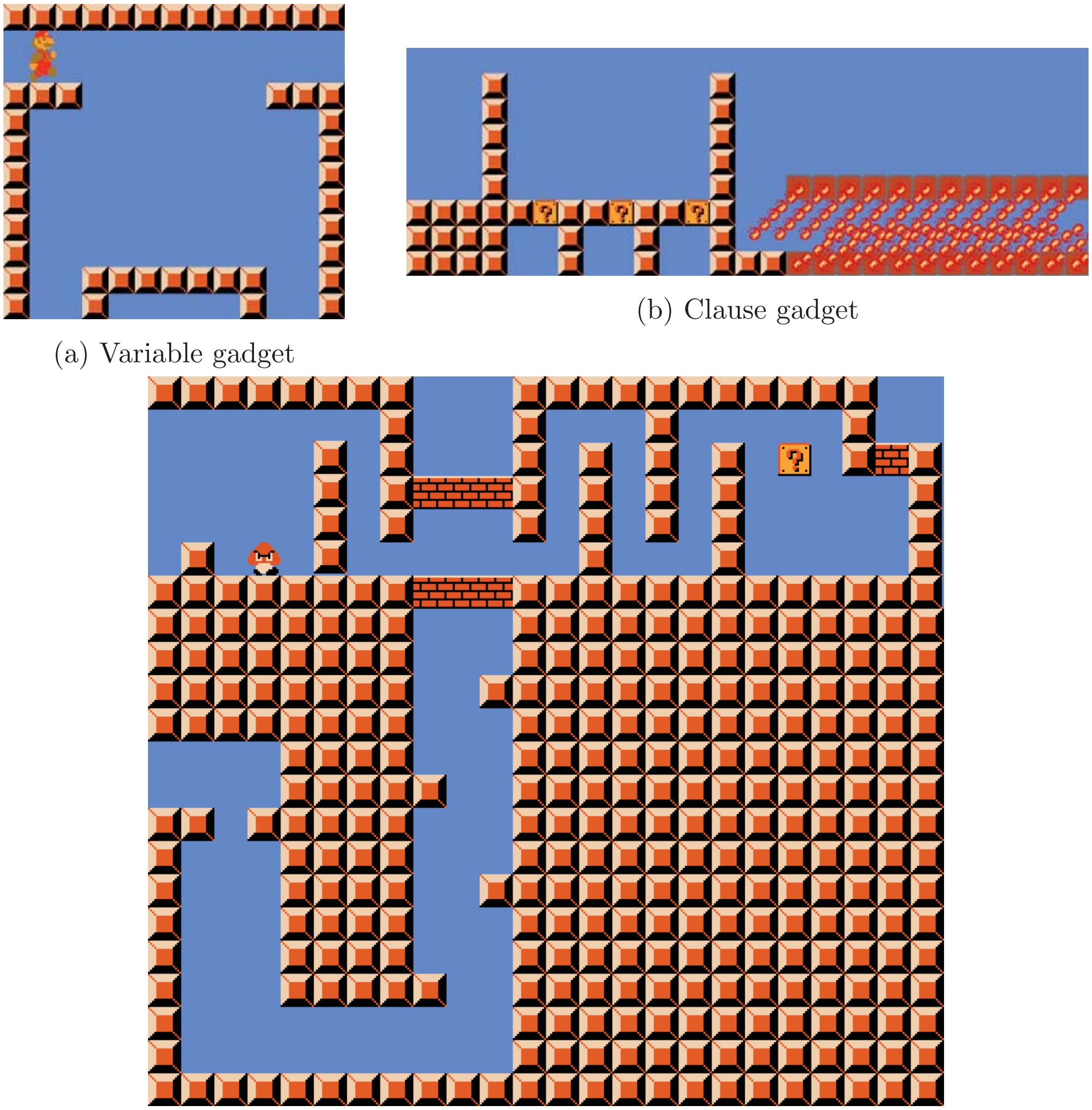
We show that Super Mario Brothers is NP-hard by giving a reduction from 3SAT. This version of Super Mario Brothers is generalized to an arbitrary screen size of $n \times n$, so we remove limits on the number of items on screen. We have the following problem definition

Definition 2. ***Super Mario Brothers:*** *Given a level of Super Mario Brothers, can we advance to the next level?*

Because we reduce from 3SAT, we are given a 3SAT instance, and we must generate a level of Super Mario Brothers that corresponds to that 3SAT instance.

We construct the level by constructing gadgets for each of the variables in the 3SAT formula, as pictured in Figure 1. Mario jumps down from the ledge, and cannot jump back up. He can fall to the left or right, corresponding to assigning the variable to True or False. The remainder of the level is set up such that this choice cannot be reversed.

We also create the following gadget for clauses. After choosing the assignment for a given variable, Mario visits all of the clause gadgets with the same literal value,



(a) Variable gadget

(b) Clause gadget

(c) Crossover gadget

Figure 1: Gadgets for Super Mario Brothers.

then moves to the next variable gadget. By visiting a clause, Mario can release a star.

Finally, after visiting all of the variable gadgets, Mario must re-traverse the clause gadgets. If the clause gadget was previously visited, a star is available, and he can pass through the fire. Otherwise, he will not be able to traverse the clause gadget and die.

Thus, winning the level is equivalent to passing through all the clause gadgets on the second pass through. Mario can only pass all the clause gadgets if they have all been satisfied by a variable assignment. The actions throughout the variable gadgets correspond to the solution to the 3SAT formula, so if Mario can pass the level, we have a solution to the 3SAT problem.

The final gadget needed is the crossover gadget. It ensures that Mario does not switch between variable and clause gadgets when it is not allowed. The total size of all these gadgets within the polynomial size required by the reduction.

Thus, Mario can win the level if and only if the original 3SAT formula could be satisfied. Therefore have a reduction from 3SAT, and Super Mario Brothers is NP-hard.

3 Dimensional Matching (3DM)

Definition 3. ***3DM:*** *Given disjoint sets X, Y, and Z, each of n elements and triples $T \subseteq X \times Y \times Z$ is there a subset $S \subseteq T$ such that each element $\in X \cup Y \cup Z$ is in exactly one $s \in S$?*

3DM is NP. Given a certificate, which lists a candidate list of triples, a verifier can check that each triple belongs to T and every element of $X \cup Y \cup Z$ is in one triple.

3DM is also NP-complete, via a reduction from 3SAT. We build gadgets for the variables and clauses.

The variable gadget for variable x_i is displayed in the picture. Only red or blue triangles can be chosen, where the red triangles correspond to the true literal, while the blue triangles correspond to the false literal. Otherwise, there will be overlap, or some inner elements will not be covered. There is an $2n_{x_i}$ "wheel" for each variable gadget, where n_{x_i} corresponds to the number of occurrences of x_i in the formula.

The clause gadget for the clause $x_i \wedge \bar{x}_j \wedge x_k$ is displayed in the picture. The dot that is unshared in each variable's triangle within the clause gadget is also the single dot within the variable gadget.

Then, if we set x_i to true, we take all of the red false triangles in the variable gadget, leaving a blue true triangle available to cover the clause gadget. However, this still potentially leaves $\bar{x}_j$ and x_k uncovered, so we need a garbage collection gadget, pictured below. There are $\sum_x n_x$ clauses of these gadgets, because there are n_x unnecessary elements of the variable gadget that won't be covered. However, of the remaining elements, one of these per clause will be covered, so the remaining need to be covered by the garbage collection clause.

Thus, if a solution exists to the 3SAT formula, we can find the solution to the 3DM problem by choosing the points in 3DM corresponding to the variable values. If we have a 3DM solution, we can map this back to the satisfying assignment for 3SAT. Thus, our reduction is complete and 3DM is NP-hard.

Subset Sum

Definition 4. ***Subset Sum*** *Given n integers $A = \{a_1, a_2, \ldots, a_n\}$ and a target sum t, is there a subset $S \subseteq A$ stuch that*

$$\sum S = \sum_{a_i \in S} a_i = t$$

The Subset Sum problem is NP-complete. It is in NP, because a verifier can simply check that the given subset is a subset of A and that its sum is equivalent to the target in polynomial time.

It is NP-hard via a reduction from 3DM. View the numbers in base $b = 1 + \max_i n_{x_i}$. Then the triple (x_i, x_j, x_k) can be written in base b in the form $000100100001000 = b^i + b^j + b^k$, where the first 1 corresponds to i, the second to j, and the third to k. The target sum $t = 111111111111111 = \sum_i b^i$.

This prevents any 1s from colliding, which ensures that each element is used exactly once, because multiple 1s correspond to reusing an element, and the base is sufficiently large so that no smaller numbers can be summed to create the next power of b. This completes the reduction.

In fact, the Subset Sum problem is only **weakly NP-hard**. The number of digits in t is $O(n)$. This means that the values of the numbers used in the reduction are exponential in input, making this problem weakly NP-hard. Problems which are **strongly NP-hard** must only use number values that are polynomial in the size of the input.

This also implies that the problem can be solved by a pseudopolynomial algorithm.

Partition

Definition 5. ***Partition:*** *Given $A = \{a_1, a_2, \ldots, a_n\}$, is there a subset $S \subseteq A$ such that*

$$\sum S = \sum A \setminus S = \frac{1}{2} \sum A?$$

Partition is also weakly NP-complete. It is a special case of the Subset Sum problem, where we set $t = \frac{1}{2} \sum A$. In fact, we can reduce Partition to Subset Sum, though this is not the direction we want for the reduction.

We can reduce from Subset Sum to Partition as follows. Let $\sigma = \sum A$. Add elements $a_{n+1} = \sigma + t$ and $a_{n+2} = 2\sigma - t$ to A. Then a_{n+1} and a_{n+2} must be on different sides of the partition. In order to balance the two sides, $\sigma + t$ must be added to a_{n+1} and t must be added to a_{n+1}, so each subset has sum 2σ. Thus if we can solve Partition, we also have the subset of elements that sum to t, the target for the Subset Sum problem, completing our reduction.

Rectangle Packing

Definition 6. ***Rectangle Packing:*** *Given a set of rectangles R_i and a target rectangle T, can we pack the rectangles in T such that there is no overlap? Note that sum of the area of the rectangles R_i is equivalent to the area of the target rectangle or $\sum_i R_i = T$.*

Rectangle packing is weakly NP-hard via a reduction from Partition. For every element a_i in Partition, we create a rectangle R_i with height 1 and width $3a_i$. The target rectangle has height 2 and width $3t = \frac{3}{2} \sum A$. Because each rectangle has width at least 3, all rectangles must be packed horizontally. Thus to solve the Rectangle packing problem, we must separate the blocks into two groups with total width $3t$, which will correspond to two subsets with total sum t in the Partition problem.

Jigsaw Puzzles [Demaine & Demaine 2007]

Definition 7. *Given square tiles with no patterns, can these tiles be arranged to fit a target rectangular shape? Note that the tiles can have a side tab, pocket, or boundary, but tabs and pockets must have matching shapes.*

The most obvious reduction is from Partition. For every number, create a set of square rectangles with a unique tab and pocket, where the number of tiles is equivalent to the value of a_i, and the two end pieces of the rectangle have boundaries.

However, this reduction cannot be completed because the inputs to Partition can be exponentially large.

Instead, the reduction comes from 4-Partition.

Definition 8. ***4-Partition:*** *Given n integers $A = \{a_1, a_2, \ldots, a_n\} \in (\frac{t}{5}, \frac{t}{3})$, is there a partition into $\frac{n}{4}$ subsets of 4 elements, each with the same sum $t = \sum A / \frac{n}{4}$?*

4-Partition is a strongly NP-complete problem. We reduce from 4-Partition to Jigsaw Puzzles to show that Jigsaw Puzzles are NP-hard.

For each a_i, we create the following set of pieces. This ensures that we cannot mix pieces from different a_i.

These pieces are set into the following target board. There are $\frac{n}{4}$ rows, each of which will hold pieces corresponding to 4 a_i terms in the 4-Partition problem. The width of the target board is t, because each row must hold t pieces, so that the corresponding a_i form a group of sum t in the 4-Partition problem. Thus, the reduction is complete.

Lecture 17: Approximation Algorithms

- Definitions
- Vertex Cover
- Set Cover
- Partition

Approximation Algorithms and Schemes

Let C_{opt} be the cost of the optimal algorithm for a problem of size n. An approximation algorithm for this problem has an approximation ratio $\varrho(n)$ if, for any input, the algorithm produces a solution of cost C such that:

$$max(\frac{C}{C_{opt}}, \frac{C_{opt}}{C}) \leq \varrho(n)$$

Such an algorithm is called a $\varrho(n)$-approximation algorithm.

An approximation scheme that takes as input $\epsilon > 0$ and produces a solution such that $C = (1+\epsilon)C_{opt}$ for any fixed ϵ, is a $(1+\epsilon)$-approximation algorithm.

A Polynomial Time Approximation Scheme (PTAS) is an approximation algorithm that runs in time polynomial in the size of the input, n. A Fully Polynomial Time Approximation Scheme (FPTAS) is an approximation algorithm that runs in time polynomial in both n and ϵ. For example, a $O(n^{2/\epsilon})$ approximation algorithm is a PTAS but not a FPTAS. A $O(n/\epsilon^2)$ approximation algorithm is a FPTAS.

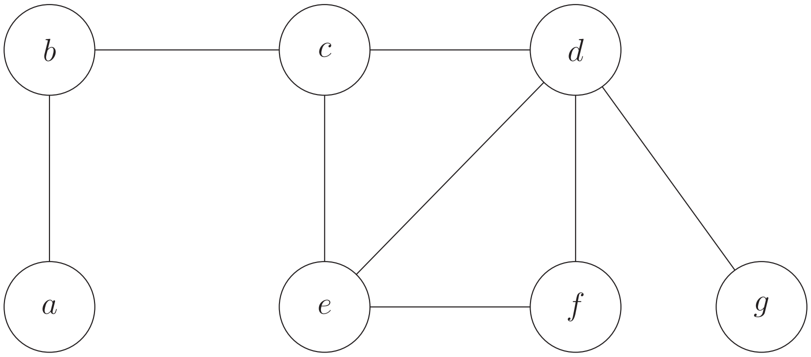
Vertex Cover

Given an undirected graph $G(V, E)$, find a subset $V' \subseteq V$ such that, for every edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$ (or both). Furthermore, find a V' such that $|V'|$ is minimum. This is an NP-Complete problem.

Approximation Algorithm For Vertex Cover

Here we define algorithm *Approx_Vertex_Cover*, an approximation algorithm for Vertex Cover. Start with an empty set V'. While there are still edges in E, pick an edge (u, v) arbitrarily. Add both u and v into V'. Remove all edges incident on u or v. Repeat until there are no more edges left in E. *Approx_Vertex_Cover* runs in polynomial time.

Take for example the following graph G:



Approx_Vertex_Cover could pick edges (b, c), (e, f) and (d, g), such that $V' = \{b, c, e, f, d, g\}$ and $|V'| = 6$. Hence, the cost is $C = |V'| = 6$. The optimal solution for this example is $\{b, d, e\}$, hence $C_{opt} = 3$.

Claim: *Approx_Vertex_Cover* is a 2-approximation algorithm.

Proof: Let $U \subseteq V$ be the set of all the edges that are picked by *Approx_Vertex_Cover*. The optimal vertex cover must include at least one endpoint of each edge in U (and other edges). Furthermore, no two edges in U share an endpoint. Therefore, $|U|$ is a lower bound for C_{opt}. i.e. $C_{opt} \geq |U|$. The number of vertices in V' returned by *Approx_Vertex_Cover* is $2 \cdot |U|$. Therefore, $C = |V'| = 2 \cdot |U| \leq 2C_{opt}$. Hence $C \leq 2 \cdot C_{opt}$. $\square$

Set Cover

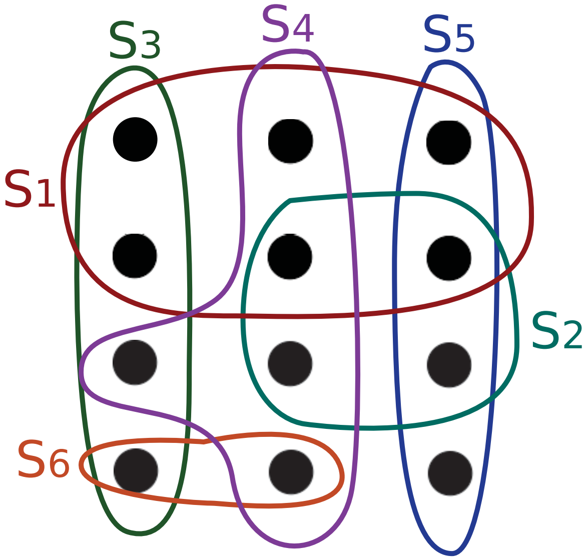
Given a set X and a family of (possibly overlapping) subsets $S_1, S_2, \cdots, S_m \subseteq X$ such that $\cup_{i=1}^{m} S_i = X$, find a set $P \subseteq \{1, 2, 3, \cdots, m\}$ such that $\cup_{i \in P} S_i = X$. Furthermore find a P such that $|P|$ is minimum.

Set Cover is an NP-Complete problem.

Approximation Algorithm for Set Cover

Here we define algorithm *Approx_Set_Cover*, an approximation algorithm for Set Cover. Start by initializing the set P to the empty set. While there are still elements in X, pick the largest set S_i and add i to P. Then remove all elements in S_i from X and all other subsets S_j. Repeat until there are no more elements in X. *Approx_Set_Cover* runs in polynomial time.

In the following example, each dot is an element in X and each S_i are subsets of X.



Approx_Set_Cover selects sets S_1, S_4, S_5, S_3 in that order. Therefore it returns $P = \{1, 4, 5, 3\}$ and its cost $C = |P| = 4$. The optimal solution is $P_{opt} = \{S_3, S_4, S_5\}$ and $C_{opt} = |P_{opt}| = 3$.

Claim: *Approx_Set_Cover* is a $(\ln(n)+1)$-approximation algorithm (where $n = |X|$).

Proof: Let the optimal cover be P_{opt} such that $C_{opt} = |P_{opt}| = t$. Let X_k be the set of elements remaining in iteration k of *Approx_Set_Cover*. Hence, $X_0 = X$. Then:

- for all k, X_k can be covered by t sets (from the optimal solution)
- one of them covers at least $\frac{|X_k|}{t}$ elements
- *Approx_Set_Cover* picks a set of (current) size $\geq \frac{|X_k|}{t}$

- for all k, $|X_{k+1}| \leq (1-\frac{1}{t})|X_k|$ (More careful analysis (see CLRS, Ch. 35) relates $\varrho(n)$ to harmonic numbers. t should shrink.)
- for all k, $|X_{k+1}| \leq (1-\frac{1}{t})^k \cdot n \leq e^{-k/t} \cdot n$ ($n = |X_0|$)

Algorithm terminates when $|X_k| < 1$, i.e., $|X_k| = 0$ and will have cost $C = k$.

$$e^{-k/t} \cdot n < 1$$

$$e^{k/t} > n$$

Hence algorithm terminates when $\frac{k}{t} > \ln(n)$. Therefore $\frac{k}{t} = \frac{C}{C_{opt}} \leq \ln(n) + 1$. Hence *Approx_Set_Cover* is a $(\ln(n)+1)$-approximation algorithm for Set Cover. □

Notice that the approximation ratio gets worse for larger problems as it changes with n.

Partition

The input is a set $S = \{1, 2, \cdots, n\}$ of n items with weights $s_1, s_2, \cdots, s_n$. Assume, without loss of generality, that the items are ordered such that $s_1 \geq s_2 \geq \cdots \geq s_n$. Partition S into sets A and B to minimize $\max(w(A), w(B))$, where $w(A) = \sum\limits_{i \in A} S_i$ and $w(B) = \sum\limits_{j \in B} S_j$.

Define $2L = \sum\limits_{i=1}^{n} s_i = w(S)$. Then optimal solution will have cost $C_{opt} \geq L$ by definition.

Partition is an NP-Complete problem. Want to find a PTAS $(1+\epsilon)$-approximation. (Note that 2-approximation in this case is trivial). Also, an FPTAS also exists for this problem.

Approximation Algorithm for Partition

Here we define *Approx_Partition*. Define $m = \lceil \frac{1}{\epsilon} \rceil - 1$. ($\epsilon \approx \frac{1}{m+1}$) The algorithm proceeds in two phases.

First Phase: Find an optimal partition A', B' of $s_1, \cdots, s_m$. This takes $O(2^m)$ time.

Second Phase: Initialize sets A and B to A' and B' respectively. Hence they already contain a partition of elements $s_1, \cdots, s_m$. Then, for each i, where i goes from $m+1$ to n, if $w(A) \leq w(B)$, add i to A, otherwise add i to B.

Claim: *Approx_Partition* is a PTAS for Partition.

Proof: Without loss of generality, assume $w(A) \geq w(B)$. Then the approximation ratio is $\frac{C}{C_{opt}} = \frac{w(A)}{L}$. Let k be the last item added to A. There are two cases, either k was added in the first phase, or in the second phase.

Case 1: k is added to A in the first phase. This means that $A = A'$. We have an optimal partition since we can't do better than $w(A')$ when we have $n \geq m$ items, and we know that $w(A')$ is optimal for the m items.

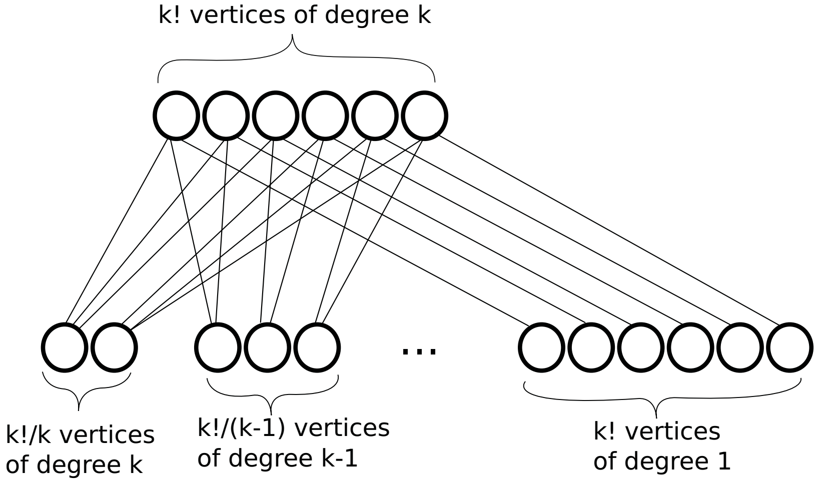
Case 2: k is added to A in the second phase. Here we know $w(A) - s_k \leq w(B)$ since this is why k was added to A and not to B. (Note that $w(B)$ may have increased after this last addition to A). Now, because $w(A)+w(B) = 2L$, $w(A) - s_k \leq w(B) = 2L - w(A)$. Therefore $w(A) \leq L + \frac{s_k}{2}$. Since $s_1 \geq s_2 \geq \cdots \geq s_n$, we can say that $s_1, s_2, \cdots, s_m \geq s_k$. Now since $k > m$, $2L \geq (m+1)s_k$.

Now, $\frac{w(A)}{L} \leq \dfrac{L + \frac{s_k}{2}}{L} = 1 + \frac{s_k}{2L} \leq 1 + \frac{s_k}{(m+1) \cdot s_k} = 1 + \frac{1}{m+1} = 1 + \epsilon$. Hence *Approx_Partition* is a $(1+\epsilon)$-approximation for Partition. □

Natural Vertex Cover Approximation

Here we describe *Approx_Vertex_Cover_Natural*, a different approximation algorithm for Vertex Cover. Start with an empty set V'. While there are still edges left in E, pick the vertex $v \in V$ that has maximum degree and add it to V'. Then remove v and all incident edges from E. Repeat until no more edges left in E. In the end, return V'.

The following example shows a bad-case example for *Approx_Vertex_Cover_Natural*. In the example, the optimal cover will pick the $k!$ vertices at the top.



Approx_Vertex_Cover_Natural could possibly pick all the bottom vertices from left to right in order. Hence the cost could be $k! \cdot (\frac{1}{k} + \frac{1}{k-1} + \cdots + 1) \approx k! \log k$. Which is a factor of $\log k$ worse than optimal.

Claim: *Approx_Vertex_Cover_Natural* is a $(\log n)$-approximation.

Proof: Let G_k be the graph after iteration k of the algorithm. And let n be the number of edges in the graph, i.e. $|G| = n = |E|$. With each iteration, the algorithm selects a vertex and deletes it along with all incident edges. Let $m = C_{opt}$ be the number of vertices in the optimal vertex cover for G. Then let's look at the first m iterations of the algorithm: $G_0 \to G_1 \to G_2 \to \cdots \to G_m$.

Let d_i be the degree of the maximum degree vertex of G_{i-1}. Then the algorithm deletes all edges incident on that vertex to get G_i. Therefore:

$$|G_m| = |G_0| - \sum_{i=1}^{m} d_i$$

Also:

$$\sum_{i=1}^{m} d_i \geq \sum_{i=1}^{m} \frac{|G_{i-1}|}{m}$$

This is true because given $|G_{i-1}|$ edges that can be covered by m vertices, we know that there is a vertex with degree at least $\frac{|G_{i-1}|}{m}$. Then:

$$\sum_{i=1}^{m} \frac{|G_{i-1}|}{m} \geq \sum_{i=1}^{m} \frac{|G_m|}{m} = |G_m|$$

This is true since $|G_i| \leq |G_{i-1}|$ for all i. Then, it follows:

$$|G_0| - |G_m| \geq |G_m|$$

Because $|G_m| \leq \sum_{i=1}^{m} d_i$. Hence after m iterations, the algorithm will have deleted half or more edges from G_0. And generally, since every m iterations it will halve the number of edges in the graph, in $m \cdot \log |G_0|$ iterations, it will have deleted all the edges. And since with each iteration it addes 1 vertex to the cover, it will end up with a vertex cover of size $m \cdot \log |G_0| = m \cdot \log n$. Since we assumed that m was the size of the optimal vertex cover, $\frac{C}{C_{opt}} = \frac{m \log n}{m} = \log n$. Hence *Approx_Vertex_Cover_Natural* is a $(\log n)$-approximation. □

Note that since $n \approx k! \log k$ in the example of Figure , the worst-case example is $\log k \approx \log \log n$ worse, but we have only shown an $O(\log n)$ approximation.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

Lecture 18: Fixed-Parameter Algorithms

- Vertex Cover
- Fixed-Parameter Tractability
- Kernelization
- Connection to Approximation

Fixed Parameter Algorithms

Fixed Parameter Algorithms are an alternative way to deal with NP-hard problems instead of approximation algorithms. There are three general desired features of an algorithm:

1. Solve (NP-)hard problems
2. Run in polynomial time (fast)
3. Get exact solutions

In general, unless P = NP, an algorithm can have two of these three features, but not all three. An algorithm that has Features 2 and 3 is an algorithm in P (poly-time exact). An approximation algorithm has Features 1 and 2. It solves hard problems, and it runs fast, but it does not give exact solutions. Fixed-parameter algorithms will have Features 1 and 3. They will solve hard problems and give exact solutions, but they will not run very fast.

Idea: The idea is to aim for an exact algorithm but isolate exponential terms to a specific *parameter*. When the value of this parameter is small, the algorithm gets fast instances. Hopefully, this parameter will be small in practice.

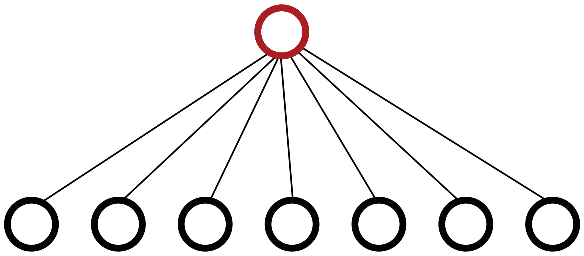
Parameter: A parameter is a nonnegative integer $k(x)$ where x is the problem input. Typically, the parameter is a natural property of the problem (some k in input). It may not necessarily be efficiently computable (e.g., OPT).

Parameterized Problem: A parameterized problem is simply the problem plus the parameter or the problem as seen with respect to the parameter. There are potentially many interesting parameterizations for any given problem.

Goal: The goal of fixed-parameter algorithms is to have an algorithm that is polynomial in the problem size n but possibly exponential in the parameter k, and still get an exact solution.

k-Vertex Cover

Given a graph $G = (V, E)$ and a nonnegative integer k, is there a set $S \subseteq V$ of vertices of size at most k, $|S| \leq k$, that covers all edges. This is a decision problem for Vertex Cover and is also NP-hard. We will use k as the parameter to develop a fixed-parameter algorithm for k-Vertex Cover. Note that we can have $k << |V|$ as the figure below shows:



Brute-force solution (bad)

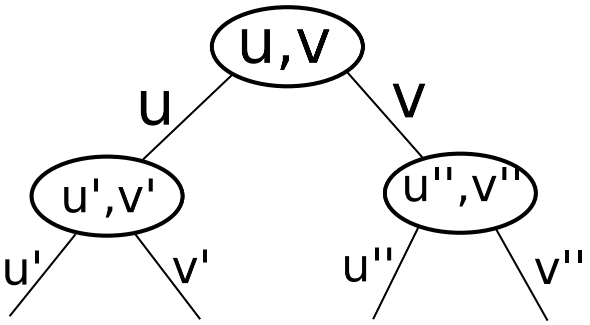
Try all $\binom{v}{k} + \binom{v}{k-1} + \cdots + \binom{v}{0}$ sets of $\leq k$ vertices. Can skip all terms smaller than $\binom{v}{k}$ because bigger sets have more coverage. Testing coverage takes $O(m)$ time where m is the number of edges. Therefore, the total runtime is $O(V^k|E|)$. It is polynomial for fixed k but not the same polynomial for all k's. It is inefficient in most cases. Hence we define $n^{f(k)}$ to be **bad**, where $n = |V| + |E|$ is the input size.

Bounded search-tree algorithm (good)

This is a general technique used to improve brute force searches. It works as follows:

- pick arbitrary edge $e = (u, v)$
- we know that either $u \in S$ or $v \in S$ (or both) but don't know which
- guess which one: try both possibilities
  1. add u to S, delete u and incident edges from G, and recurse with $k' = k - 1$.
  2. do the same but with v instead of u
  3. return the OR of the two outcomes

This is like guessing in dynamic programming but memoization doesn't help here. The recursion tree looks like the following:



At a leaf ($k = 0$), return YES if $|E| = 0$ (all edges covered). It takes $O(V)$ time to delete u or v. Therefore this has a total runtime of $(2^k|V|)$.

- $O(V)$ for fixed k
- degree of polynomial is independent of k
- also polynomial for $k = O(\lg |V|)$
- practical for e.g. $k \leq 32$
- Hence we define $f(k) \cdot n^{O(1)}$ to be **good**

Fixed Parameter Tractability

A parameterized problem is fixed-parameter tractable (FPT) if there is an algorithm with running time $\leq f(k) \cdot n^{O(1)}$, such that $f : \mathbb{N} \to \mathbb{N}$ (non negative) and k is the parameter, and the $O(1)$ degree of the polynomial is independent of k and n.

Question: Why $f(k) \cdot n^{O(1)}$ and not $f(k) + n^{O(1)}$?

Theorem: $\exists f(k) \cdot n^c$ algorithm $\iff \exists f'(k) + n^{c'}$

Proof:
($\Leftarrow$)
Trivial (assuming $f'(k)$ and $n^{c'}$ are ≥ 1)
($\Rightarrow$)
if $n \leq f(k)$, then $f(k) \cdot n^c \leq f(k)^{c+1}$

if $f(k) \le n$ then $f(k) \cdot n^c \le n^{c+1}$
Therefore $f(k) \cdot n^c \le \max(f(k)^{c+1}, n^{c+1}) \le f(k)^{c+1} + n^{c+1} = f'(k) + n^{c'}$ □

Alternatively, since $xy \le x^2 + y^2$, can just make $f'(k) = (f(k))^2$ and $c' = 2c$.
Example: $O(2^k \cdot n) \le O(4^k + n^2)$

Kernelization

Kernelization is a simplifying self-reduction. It is a polynomial time algorithm that converts an input (x, k) into a *small* and *equivalent* input (x', k'). Here, *small* means $|x'| \le f(k)$ and *equivalent* means the answer to x is the same as the answer to x'.

Theorem: a problem is FPT $\iff$ $\exists$ a kernelization

Proof:
$(\Leftarrow)$
Kernelize $\Rightarrow n' \le f(k)$
Run any finite $g(n')$ algorithm
Totals to $n^{O(1)} + g(f(k))$ time
$(\Rightarrow)$
let A be an $f(k) \cdot n^c$ algorithm, then assuming k is known:
if $n \le f(k)$, it's already kernelized.
if $f(k) \le n$, then

1. run $A \rightarrow f(k) \cdot n^c \le n^{c+1}$ time
2. output $O(1)$-sized YES/NO instance as appropriate (to kernelize)

if k is unknown: run A for n^{c+1} time and if it is still not done, we know it is already kernelized.

So we know (exponential) kernel exists. Recent work aims to find polynomial (even linear) kernels when possible.

Polynomial kernel for k-Vertex Cover

To create a kernel for k-Vertex Cover, the algorithm follows the following steps:

- Make graph simple by removing all self loops and multi-edges
- Any vertex of degree $> k$ must be in the cover (else would need to add $> k$ vertices to cover incident edges)

- Remove such vertices (and incident edges) one at a time, decreasing k accordingly
- Remaining graph has maximum degree $\leq k$
- Each remaining vertex covers $\leq k$ edges
- If the number of remaining edges is $> k$, answer NO and output canonical NO instance.
- Else, $|E'| \leq k^2$
- Remove all isolated vertices (degree 0 vertices)
- Now $|V'| \leq 2k^2$
- The input has been reduced to instance (V', E') of size $O(k^2)$

The runtime of the kernelization algorithm is naively $O(VE)$. ($O(V+E)$ with more work.) After this, we can apply either a brute-force algorithm on the kernel, which yields an overall runtime $O(V+E+(2k^2)^k k^2) = O(V+E+2^k k^{2k+2})$. Or we can apply a bounded search-tree solution, which yields a runtime of $O(V+E+2^k k^2)$.

The best algorithm to date: $O(kV + 1.274^k)$ by [Chen, Kanj, Xia - TCS 2010].

Connection to Approximation Algorithms

Take an optimization problem, integral OPT and consider its associated decision problem: "OPT $\leq k$?" and parameterize by k.

Theorem: optimization problem has EPTAS
(EPTAS: efficient PTAS, $f(\frac{1}{\epsilon}) \cdot n^{O(1)}$ e.g. $Approx_Partition[L17]$)
$\Rightarrow$ decision problem is FPT

Proof: (like FPTAS, pseudopolynomial algorithm)

- Say maximization problem (and $\leq k$ decision)
- run EPTAS with $\epsilon = \frac{1}{2k}$ in $f(2k) \cdot n^{O(1)}$ time.
- relative error $\leq \frac{1}{2k} < \frac{1}{k}$
- $\Rightarrow$ absolute error < 1 if OPT $\leq k$

- So if we find a solution with value $\leq k$, then OPT $\leq (1 + \frac{1}{2k}) \cdot k \leq k + \frac{1}{2}$
  Integral $\Rightarrow$ OPT $\leq k \Rightarrow$ YES
- else OPT $> k$

□

Also: $=$, $\leq$, $\geq$ decision problems are equivalent with respect to FPT.

(Can use this relation to prove that EPTASs don't exists in some cases)

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

Lecture 21: Cryptography: Hashing

In this lecture, we will be studying some basics of cryptography. Specifically, we will be covering

- Hash functions
- Random oracle model
- Desirable Properties
- Applications to security

1 Hash Functions

A hash function h maps arbitrary strings of data to fixed length output. The function is deterministic and public, but the mapping should look "random". In other words,

$$h : \{0,1\}^* \to \{0,1\}^d$$

for a fixed d. Hash functions do not have a secret key. Since there are no secrets and the function itself is public, anyone can evaluate the function. To list some examples,

Hash function	MD4	MD5	SHA-1	SHA-256	SHA-512
d	128	128	160	256	512

In practice, hash functions are used for "digesting" large data. For example, if you want to check the validity of a large file (potentially much larger than a few megabytes), you can check the hash value of that file with the expected hash. Therefore, it is desirable (especially for cryptographic hash functions covered here) that the function is collision resistant. That is, it should be "hard" to find two inputs m_1 and m_2 for hash function h such that $h(m_1) = h(m_2)$. Most modern hash functions hope to achieve security level of 2^{64} or better, which means that the attacker needs to test more than 2^{64} different inputs to find a collision. Unfortunately, MD4 and MD5 aimed to provide 2^{64} security, but has been shown to be broken using 2^6 and 2^{37} inputs respectively. SHA-1 aimed to provide 2^{80} security, but has been shown (at least theoretically) to be no more than 2^{61} security.

1.1 Random Oracle

The Random Oracle model is an ideal model of the hash function that is not achievable in practice. In this model, we assume there exists an oracle h such that on input $x \in \{0,1\}^*$, if h has not seen x before, then it outputs a random value as $h(x)$. Otherwise, it returns $h(x)$ it previously output. The random oracle gives a random value for all new inputs, and gives deterministic answers to all inputs it has seen before. Unfortunately, a random oracle does not exist since it requires infinite storage, so in practice we use pseudo-random functions.

1.2 Desirable Properties

There are many desirable properties of a hash function.

1. One-way (pre-image resistance): Given $y \in \{0,1\}^d$, it is hard to find an x such that $h(x) = y$.

2. Strong collision-resistance: It is hard to find any pair of inputs x, x' such that $h(x) = h(x')$.

3. Weak collision-resistance (target collision resistance, 2nd pre-image resistance): Given x, it is hard to find x' such that $h(x) = h(x')$.

4. Pseudo-random: The function behaves indistinguishable from a random oracle.

5. Non-malleability: Given $h(x)$, it is hard to generate $h(f(x))$ for any function f.

Some of the properties imply others, and some others do not. For example,

- $2 \Rightarrow 3$
- $1 \not\Rightarrow 2, 3$.

Furthermore, collision can be found in $O(2^{d/2})$ (using birthday paradox), and inversion can be found in $O(2^d)$.

To give more insight as to why some properties do not imply others, we provide examples here. Consider h that satisfies 1 and 2. We can construct a new h' such that h' takes in one extra bit of input, and XORs the first two bits together to generate an input for h. That is, $h'(a, b, x_2, \ldots, x_n) = h((a \oplus b), x_2, \ldots, x_n)$. h' is still one-way, but is not weak collision resistant. Now consider a different h'' that evaluates to $0||x$ if $|x| \leq n$, and $1||h(x)$ otherwise. h'' is weak collision resistant, but is not one-way.

1.3 Applications

There are many applications of hash functions.

1. Password Storage: We can store hash $h(p)$ for password p instead of p directly, and check $h(p)$ to authenticate a user. If it satisfies the property 1, adversary comprising $h(p)$ will not learn p.

2. File Authenticity: For each file F, we can store $h(F)$ in a secure location. To check authenticity of a file, we can recompute $h(F)$. This requires property 3.

3. Digital Signature: We can use hash functions to generate a signature that guarantees that the message came from a said source. For further explanation, refer to Recitation 11.

4. Commitments: In a secure bidding, Alice wants to bid value x, but does not want to reveal the bid until the auction is over. Alice then computes $h(x)$, and publicize it, which serves as her commitment. When bidding is over, then she can reveal x, and x can be verified using $h(x)$.

   It should be "binding" so that Alice cannot generate a new x that has the same commitment, and it should "hide" x so no ones learns anything before x is revealed. Furthermore, it should be non-malleable. To guarantee secrecy, we need more than the properties from the previous section, as $h'(x) = h(x)||MSB(x)$ actually satisfies 1, 2, and 5. In practice, this problem is bypassed by adding randomness in the commitment, $C(x) = h(r||x)$ for $r \in_R \{0,1\}^{256}$, and reveal both randomness and x at the end.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

Lecture 22: Cryptography: Encryption

- Symmetric key encryption
- Key exchange
- Asymmetric key encryption
- RSA
- NP-complete problems and cryptography
  - graph coloring
  - knapsack

Symmetric key encryption

$$c = e_k(m)$$
$$m = d_k(c)$$

Here c is the *ciphertext*, m is the *plaintext*, e is the encryption function, d is the decryption function and k is the secret key. e, d permute and reverse-permute the space of all messages.

Reversible operations: $\oplus$, $+/-$, shift left/right.

Symmetric algorithms: AES, RC5, DES

Key Management Question

How does secret key k get exchanged/shared?

Alice wants to send a message to Bob. There are pirates between Alice and Bob, that will take any keys or messages in unlocked box(es), but won't touch locked boxes. How can Alice send a message or a key to Bob (without pirates knowing what was sent)?

Solution:

- Alice puts m in box, locks it with k_A
- Box sent to Bob

- Bob locks box with k_B
- Box sent to Alice
- Alice unlocks k_A
- Box sent to Bob
- Bob unlocks k_B, reads m

Notice that this method relied on the commutativity of the locks. This means that the order of the lock and unlock operations doesn't matter.

Diffie-Hellman Key Exchange

$$G = F_p^*$$

Here F_p^* is a finite field (mod p, a prime). $*$ means invertible elements only ($\{1, 2, ..., p-1\}$)

Alice	g public p public	Bob	$2 \leq g \leq p-2$
Select a Compute g^a	$\longrightarrow$	g^a Select b Compute g^b	$1 \leq a, b \leq p-2$
g^b	$\longleftarrow$		

Alice can compute $\left(g^b\right)^a \mod p = k$.

Bob can compute $(g^a)^b \mod p = k$.

Assumes the Discrete Log Problem is hard (given g^a, compute a) and Diffie Hellman Problem is hard (given g^a, g^b compute g^{ab}).

Can we attack this? Man-in-the-middle:

- Alice doesn't know she is communicating with Bob.
- Alice agrees to a key exchange with Eve (thinking she is Bob).
- Bob agrees to a key exchagne with Eve (thinking she is Alice).
- Eve can see all communications.

Public Key Encryption

$$\text{message} + \text{public key} = \text{ciphertext}$$
$$\text{ciphertext} + \text{private key} = \text{message}$$

The two keys need to be linked in a mathematical way. Knowing the public key should tell you nothing about the private key.

RSA

- Alice picks two large secret primes p and q.
- Alice computes $N = p \cdot q$.
- Chooses an encryption exponent e which satisfies $gcd(e, (p-1)(q-1)) = 1$, $e = 3,\ 17,\ 65537$.
- Alice's public key= (N, e).
- Decryption exponent obtained using Extended Euclidean Algorithm by Alice such that $e \cdot d \equiv 1 \mod (p-1)(q-1)$.
- Alice private key=(d, p, q) (storing p and q is not absolutely necessary, but we do it for efficiency).

Encryption and Decryption with RSA

$$\begin{aligned} c &= m^e \mod N && \text{encryption} \\ m &= c^d \mod N && \text{decryption} \end{aligned}$$

Why it works

$$\phi = (p-1)(q-1)$$

Since $ed \equiv 1 \mod \phi$ there exists an integer k such that $ed = 1 + k\phi$.
Two cases:
Case 1 $gcd(m, p) = 1$. By Fermat's theorem,

$$\begin{aligned} m^{p-1} &\equiv 1 \mod p \\ \left(m^{p-1}\right)^{k(q-1)} \cdot m &\equiv m \mod p \\ m^{1+k(p-1)(q-1)} = m^{ed} &\equiv m \mod p \end{aligned}$$

Case 2 $gcd(m, p) = p$. This means that $m \mod p = 0$ and so $m^{ed} \equiv m$

Thus, in both cases, $m^{ed} \equiv m \mod p$. Similarly, $m^{ed} \equiv m \mod q$. Since p, q are distinct primes, $m^{ed} \equiv m \mod N$. So $c^d = (m^e)^d \equiv m \mod N$

Hardness of RSA

- Factoring: given N, hard to factor into p, q.
- RSA Problem: given e such that $gcd(e, (p-1)(q-1)) = 1$ and c, find m such that $m^e \equiv c \mod N$.

NP-completeness

Is N composite with a factor within a range? unknown if NP-complete

Is a graph k-colorable? In other words: can you assign one of k colors to each vertex such that no two vertices connected by an edge share the same color? NP-complete

Given a pile of n items, each with different weights w_i, is it possible to put items in a knapsack such that we get a specific total weight S? NP-complete

NP-completeness and Cryptography

- NP-completeness: about worst-case complexity
- Cryptography: want a problem instance, with suitably chosen parameters that is hard on average

Most knapsack cryptosystems have failed.

Determining if a graph is 3-colorable is NP-complete, but very easy on average. This is because an average graph, beyond a certain size, is not 3-colorable!

Consider a standard backtracking search to determine 3-colorability.

- Order vertices $v_1, ..., v_t$. Colors $= \{1, 2, 3\}$
- Traverse graph in order of vertices.
- On visiting a vertex, choose smallest possible color that "works".
- If you get stuck, backtrack to previous choice, and try next choice.

- Run out of colors for 1^{st} vertex → output 'NO'
- Successfully color last vertex → output 'YES'

On a random graph of t vertices, average number of vertices traveled < 197, regardless of t!

Knapsack Cryptography

General knapsack problem: NP-complete

Super-increasing knapsack: linear time solvable. In this problem, the weights are constrained as follows:

$$w_j \geq \sum_{i=1}^{j-1} w_i$$

Merkle Hellman Cryptosystem

Private key → super-increasing knapsack problem $\xrightarrow{\text{Private transform}}$ "hard" general knapsack problem → public key.

Transform: two private integers N, M s.t. $\gcd(N, M) = 1$.

Multiply all values in the sequence by N and then take mod M.

Example: $N = 31$, $M = 105$, private key=$\{2, 3, 6, 14, 27, 52\}$,
public key=$\{62, 93, 81, 88, 102, 37\}$

Merkle Hellman Example

$$\begin{aligned}
\text{Message} &= 011000 \quad 110101 \quad 101110 \\
\text{Ciphertext:}011000 &\qquad 93 + 81 = 174 \\
110101 &\qquad 62 + 93 + 88 + 37 = 280 \\
101110 &\qquad 62 + 81 + 88 + 102 = 333 \\
&= 174, 280, 333
\end{aligned}$$

Recipient knows $N = 31, M = 105$, $\{2, 3, 6, 14, 27, 52\}$. Multiplies each ciphertext block by $N^{-1} \mod M$. In this case, $N^{-1} = 61 \mod 105$.

$$\begin{aligned}
174 \cdot 61 &= 9 = 3 + 6 = 011000 \\
280 \cdot 61 &= 70 = 2 + 3 + 13 + 52 = 110101 \\
333 \cdot 61 &= 48 = 2 + 6 + 13 + 27 = 101110
\end{aligned}$$

Beautiful but broken

Lattice based techniques break this scheme.

Density of knapsack $d = \frac{n}{\max\{\log_2 w_i : 1 \leq i \leq n\}}$

Lattice basis reduction can solve knapsacks of low density. Unfortunately, M-H scheme always produces knapsacks of low density.

Lecture 23: Cache-oblivious Algorithms I

This lecture introduces cache-oblivious algorithms. Topics include

- memory hierarchy
- external memory vs. cache-oblivious model
- cache-oblivious scanning
- cache-oblivious divide & conquer algorithms: median finding & matrix multiplication

1 Modern Memory Hierarchy

So far in this class, we have viewed all operations and memory accesses as equal cost. However, modern computers have memory hierarchy.

$$\boxed{\text{CPU}} \longrightarrow \boxed{\text{L1}} \longrightarrow \boxed{\text{L2}} \longrightarrow \boxed{\text{L3}} \longrightarrow \boxed{\text{memory}} \longrightarrow \boxed{\text{disk}}$$

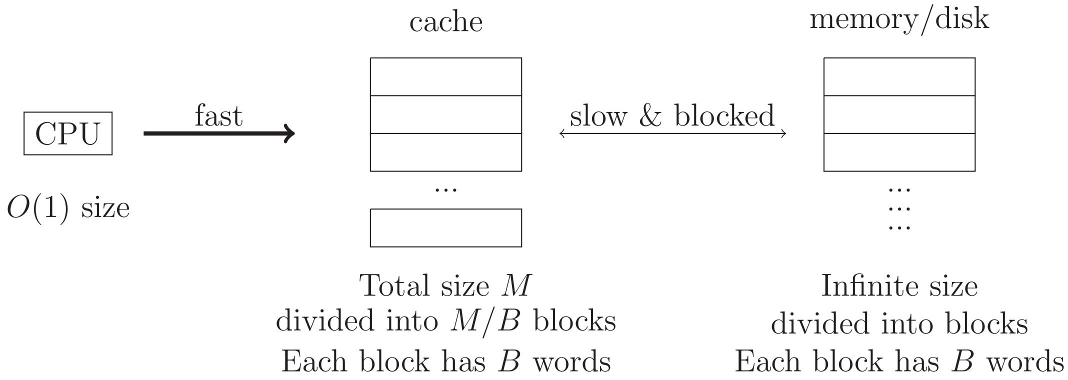
size	~	10KB	100KB	MB	GB	TB
latency	~	1ns	10ns	10ns	100ns	10ms

Each hierarchy on the right is bigger, but has longer latency due to the longer distance data has to travel. Yet, bandwidth between different hierarchies is usually matched.

A common technique to mitigate latency is *blocking*: when fetching a word of data, get the entire block containing it. Using algorithmic terminology, the idea is to amortize latency over a whole block. For this idea to work, we additional require the algorithm to use all words in a block (spatial locality) and reuse blocks in cache (temporal locality).

2 Memory Model of Algorithms

2.1 External Memory Model



In this model, cache acecsses are free. The algorithm explicitly reads and writes memory in blocks. We will count the number of memory transfers between the cache and the memory, as an important metric of the algorithm's performance.

2.2 Cache-oblivious Model

The only change from the external memory model is that the algorithm no longer knows B and M. Accessing a word in the memory automatically fetches an entire block into the cache, and evicts the least recently used (LRU) block from the cache if the cache is full.

Every algorithm is a cache-oblivious algorithm, but we would like to find the algorithm that minimizes our new metirc — the number of memory transfers.

Why do we like cache-olivious algorithms (as opposed to letting the algorithm know B and M)? Because this way, an algorithm can auto-tune itself and run efficiently on different computers (possibly with different B and M). Besides, it is cool research!

3 Scanning

Example program:

```
    for $i$ in range($N$): sum += $A[i]$
```

Assume A is stored contiguously in memory. External memory model can align A with a block boundary, so it needs $\lceil N/B \rceil$ memory transfers.

Cache-oblivious algorithms cannot control alignment (because it does not know B), so it needs $\lceil N/B \rceil + 1 = N/B + O(1)$ memory transfers. $O(1)$ parallel scans still need $O(N/B + 1)$ memory transfers.

4 Divide & Conquer

Divide & Conquer algorithms divide problems down to $O(1)$ size. The base case of the recursion is either when

- problem fits in cache i.e., $\leq M$, or
- problem fits in $O(1)$ blocks, i.e., $O(B)$.

Below we will see one example for each.

4.1 Median Finding / Order Statistics

Recall the steps of the algorithm

1. view array as partitioned into columns of 5 (each column is $O(1)$ size).
2. sort each column
3. recursively find the median of the column medians
4. partition array by x
5. recurse on one side

We will now analyze the number of memory transfers in each step. Let $MT(N)$ be the total number of memory transfers.

1. free
2. a scan, $O(N/B + 1)$
3. $MT(N/5)$, this involves a pre-processing step that coallesces the $N/5$ elements in a consecutive array
4. 3 parallel scans, $O(N/B + 1)$
5. $MT(7N/10)$

Therefore, we get the recursion

$$MT(N) = MT(N/5) + MT(7N/10) + O(N/B + 1).$$

Solving the recursion requires setting a base case. An obvious base case is $MT(O(1)) = O(1)$.

But we can get a stronger base case: $MT(O(B)) = O(1)$. Uing this base case, the recursion solves to $MT(N) = O(N/B + 1)$. (Intuition: cost at level of the recursion decreases geometrically, so the cost at root dominates.)

4.2 Matrix Multiplication

Problem: compute $Z = X \cdot Y$ where X, Y, Z are all $N \times N$ matrices. Also suppose X is stored in row-major order, and Y is stored in column-major order to improve locality.

If we use the basic algorithm, computing one element in Z requires one or two parallel scans, because it either requires scanning either a new row from X, or a new column from Y. Computing each element takes $O(N/B + 1)$ memory transfers, so computing the entire Z costs $O(N^3/B + N^2)$ memory transfers.

Instead we will use the blocked matrix multiplcation algorithm (not Strassen). Note that key block should be stored consecutively. We get recursion

$$MT(N) = 8MT(N/2) + O(N^2/B + 1)$$

The first term is recursive sub-matrix multiplication, and the second term is matrix addition which requires scanning the matrices.

Again, we can have different base cases

- weak: $MT(O(1)) = O(1)$
- better: $MT(O(B)) = O(1)$
- even better: $MT(\sqrt{M/3}) = O(M/B)$

The third case represents the case that the three involved matrices can fit in the cache together. Therefore, to multiply them, we only need one scan to load all of them into cache.

If we draw the recursion tree, the cost at each level is geometrically increasing this time, $N^2/B, 8(\frac{N}{2})^2/B, 8^2(\frac{N}{4})^2/B, \ldots$. Therefore, the cost at the leaves dominate, and the total cost = cost per leave $\cdot$ number of leaves,

$$MT(N) = O(M/B) \cdot 8^{O(\lg N/\sqrt{M})} = O(M/B) \cdot O((N/\sqrt{M})^3) = O(N^3/B\sqrt{M}).$$

Lecture 24: Cache-oblivious algorithms II

- Search
  - binary
  - B-ary
  - cache-oblivious
- Sorting
  - mergesorts
  - cache-oblivious

Why LRU block replacement strategy?

$LRU_M \leq 2 \cdot OPT_{M/2}$ [Sleater and Tarjan 1985]

Proof.

- partition block access sequence into maximal phases of M/B distinct blocks
- LRU spends $\leq M/B$ memory transfers/phase
- OPT must spend $\geq \frac{M}{2}/B$ memory transfers per phase: at best, starts phase with entire $M/2$ cache with needed items. But there are M/B blocks during phase. So $\leq$ half free

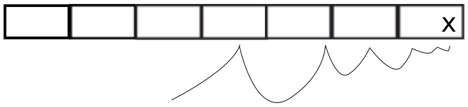
Search

Preprocess n elements in comparison model to support predecessor search for x.

B-trees

They support predecessor (and insert and delete) in $O(\log_{B+1} N)$ memory transfers.

- each node occupies $\Theta(1)$ blocks
- height$= \Theta(\log_B N)$
- need to know B

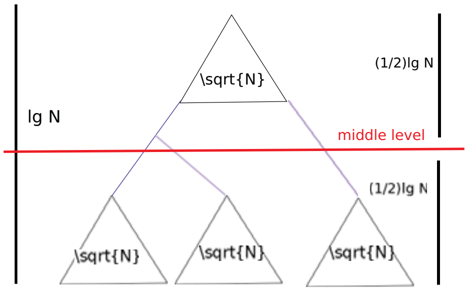


Binary search

Approximately, every iteration visits a different block until we are in x's block. Thus, $MT(N) = \Theta(\log N - \log B) = \Theta(\log(N/B))$. SLOW

van Emde Boas layout

[Prokop 1999]



- store N elements in complete BST
- carve BST at middle level of edges
- recursively layout the pieces and concatenate
- like block matrix multiplication, order of pieces doesn't matter; just need each piece to be stored consecutively

Analysis of BST search in vEB layout:

- consider recursive level of refinement at which structure has $\leq B$ nodes
- the height of the vEB tree is between $\frac{1}{2}\lg B$ and $\lg B \implies$ size is between $\sqrt{B}$ and B
  $\implies$ any root-to-node path (search path) visits $\leq \frac{\lg N}{\frac{1}{2}\lg B} = 2\log_B N$ trees that have size $\leq B$
- each tree of size $\leq B$ occupies ≤ 2 memory blocks

$$\implies \leq 4\log_B N = O(\log_B N) \text{ memory transfers}$$

- this generalizes to heights that are not powers of 2, B-trees of constant branching factor and dynamic B-trees: $O(\log_B N)$ insert/delete. [Bender, Demaine, Farach-Colton 2000]

Sorting

B-trees

N inserts into (cache-oblivious) B-tree $\implies$ $MT(N) = \Theta(N \log_B N)$ NOT OPTIMAL. By contrast, BST sort is optimal $O(N \lg N)$

Binary mergesort

- binary mergesort is cache-oblivious.
- the merge is 3 parallel scans
  $\implies MT(N) = 2MT(N/2) + O(N/B + 1)$
  $MT(M) = O(M/B)$
- the recursion tree has $\lg(N/M)$ levels, and each level contributes $O(N/B)$
  $\implies MT(N) = \frac{N}{B} \lg \frac{N}{M}$. $\leftarrow$ $\frac{B}{\lg B}$ faster than the B-tree version discussed earlier!

M/B-way mergesort

- split array into M/B equal subarrays
- recursively sort each
- merge via M/B parallel scans (keeping one "current" block per list)

$$\implies MT(N) = \frac{M}{B} MT\left(\frac{N}{M/B}\right) + O(N/B + 1)$$
$$MT(M) = O(M/B)$$

$$\begin{aligned}\implies \text{height becomes } \log_{M/B} \frac{N}{M} + 1 \\ &= \log_{M/B} \frac{N}{B} \cdot \frac{B}{M} + 1 \\ &= \log_{M/B} \frac{N}{B} - \log_{M/B} \frac{M}{B} + 1 \\ &= \log_{M/B} \frac{N}{B}\end{aligned}$$

$$\implies MT(N) = O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$$

This is asymptotically optimal, in the comparison model.

Cache-oblivious Sorting

This requires the tall-cache assumption: $M = \Omega(B^{1+\epsilon})$ for some fixed $\epsilon > 0$, e.g., $M = \Omega(B^2)$ or $M/B = \Omega(B)$.

Then, $\approx N^{\epsilon}$-way mergesort with recursive ("funnel") merge works.

Priority Queues

- $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ per insert or delete-min
- generalizes sorting
- external memory and cache-oblivious
- see 6.851