Lecture 12: Locality-Sensitive Hashing and MinHash

Sam McCauley

October 21, 2021

Williams College

Admin

- Mini-Midterm 1 handed back
- Assignment 5 out this afternoon
    - Leaderboard is back this week
- Mini-Midterm 2 next week

SIMD Cliffhanger

- We ended right before starting SIMD last time!
- Need to focus on Assignment 5 today
- If we don't get to SIMD we'll do it on Monday
- There is one SIMD part of the assignment (but you can do the rest without SIMD)

Mini-Midterm 1 review

Comments on Mini-Midterm 1

- Lots of good solutions; questions generally went well
- Biggest difference between solutions: how to store buckets?
- Let's look at a few options. We'll be using hash buckets (and doing compare-all-pairs within each bucket) on Assignment 5 too.

Storing Buckets: arrays

1. Store array of size n
   - Easy, effective
   - Space increases by # buckets. Could run out of memory on large input with high SHIFT.
2. Dynamic arrays: if bucket fills up, double size
   - Can use `realloc`
   - Insert-only, so can just double every time size is a power of two
   - Effective; space-efficient. But requires extra work to implement
3. Count bucket sizes beforehand; allocate array of correct size
   - Pretty easy and effective. Does require some extra coding (and an extra scan through the data)

Storing Buckets: Other Methods

- Linked List?
  - Easy (?) (need to make a struct for the node, but straightforward after that modulo some pointer issues)
  - Not cache-efficient to traverse!
  - One option: before calling Naive 3SUM on a bucket, first transfer list to an array
    - This means Naive 3SUM on buckets of size X costs $O(X)$ extra cache misses
    - But after that it's cache-efficient
    - ...but does that matter if $3X < M$?

Storing Buckets: Other Methods

- Sorting!
- Sort elements by their hash value. If two elements have the same hash value, compare by their actual value
- After one call to sort: buckets are all sorted and stored contiguously in memory
- Very very easy! And can store in-place
- Downsides:
    - need to store hashes of each element (or recalculate every comparison).
    - Have to be careful when comparing

Any other mini-midterm 1 questions?

How the semester is going

- Really well on my end.
- These are hard topics
- but I'm seeing consistently good understanding in the class
- Hopefully hitting a good balance between a challenge and causing stress
- Especially important during a Covid semester

Grades so far

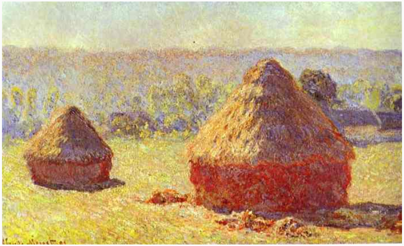
- Median assignment grade: 94
- Median mini-midterm 1 grade: 94.5
- Seems reasonable to me
- Grading is a bit tricky on take-home programming assignments
    - Only real way to get points off is to not notice a problem, or to run out of time
    - Challenging assignments can wind up being a matter of how much time each student spends rather than how much each gets correct. (Hopefully not too much!)
    - May smooth out a bit in Part 3 of class, which doesn't have any C programming

Finding Similar Items

Back to Normal Inputs

- Today: no more streaming! Have all data available to us.
- But data is still big!
  - In particular: high-dimensional
  - Table with many columns
  - For each netflix user, what movies have they seen
- Goal: solve a difficult, but important, problem

Finding Similar Pair



- Given a set of objects
- Find the most similar pair of objects in the set

Why Find Similar Objects?

- Find similar news articles for user suggestions.
- Similar music: Spotify suggests music by finding similar users, and selecting what they listen to
- Machine learning in general (training, evaluation, actual algorithms, etc.)
- Data deduplication, etc.
- "Give me a similar pair in this dataset" is a common query!

Strategies for Similarity Search

First attempt: 1-dimensional data

92
44
7
65
60
23
80
67

- Given a list of numbers
- "Similarity" is the difference between them
- How can we find the closest numbers (i.e. ones with smallest difference)?

First attempt: 1-dimensional data

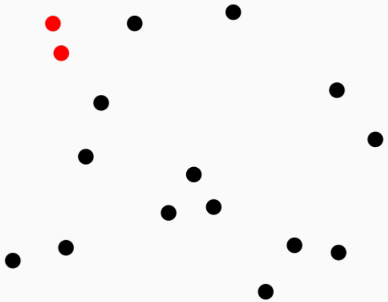
Aside: can we do better? Yes, there's a clever $O(n)$ algorithm based on sampling.

an we do

60
65
67
80
92

- Step 2: Scan through list, find most similar adjacent elements.
- $O(n \log n)$ time

Two-dimensional Data?

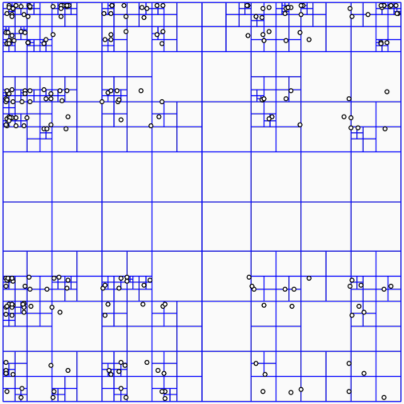


- You likely saw this in CS 256.
- Divide and conquer, $O(n \log n)$ time.
- (Again, possible in $O(n)$)

What About Higher Dimensions?

- We want VERY high dimensions (millions)
- Songs listened to, movies watched, image tags, etc.
- Words that appear in a book, k-grams that appear in a DNA sequence
- Classic options: quad trees, kd trees

How Efficient are High-dimensional Algorithms?



- $O(n \log n)$ for constant dimensions
- But: exponential in dimension!
- Worse than trying all pairs if $> \log n$ dimensions

Curse of Dimensionality

- Many problems have running time exponential in the dimension of the objects.
- Well-known phenomenon
- Applies to similarity search, machine learning, combinatorics
  - Approximate techniques, like those we learn about today, are underused to a slightly shocking extent—even in ML people sometimes keep dimensionality low to avoid this issue, affecting the quality of results

Avoiding the Curse of Dimensionality

- Today we're talking about how to get efficient algorithms for *arbitrarily* large dimensions.
- Linear cost in terms of dimension (but expensive in terms of the problem size).
- Two tools to get us there:
  - Assume that the close pair is much closer than any other (*approximate* closest pair)
  - Use hashing! ...A special kind of hashing
- For many of these problems, random inputs are worst-case inputs
  - Worst case behavior actually occurs for many common use cases; guarantees (even approximate) can be very valuable

We'll come back to this later

Locality-Sensitive Hashing

Locality-Sensitive Hashing

- Normally, hashing *spreads out* elements.
- This is key to hashing: no matter how clustered my data begins, I wind up with a nicely-distributed hash table
- Locality-sensitive hashing tries to hash similar items together

Locality-Sensitive Hashing: Formal Definition

- Needs a similarity threshold r, an approximation factor $c < 1$
- Two guarantees:
  - If two items x and y have similarity $\geq r$, $h(x) = h(y)$ with probability at least p_1.
  - If two items x and y have similarity $\leq cr$, $h(x) = h(y)$ with probability at most p_2.
- High level: close items are likely to collide. Far items are unlikely to collide.
- Generally want p_2 to be about $1/n$; then we get a normal hash table for far (i.e. distance $\geq cr$) elements.

Why Locality-Sensitive Hashing Helps

	(101, 37, 65) (103,37,64)	(91,84,3)		(100,18,79)
0	1	2	3	4

Ideally, close items hash to the same bucket.

Issue: Low probability of success!

We'll put numbers on this later

- If we have $p_2 = 1/n$, then p_1 is usually very small.

- How can we increase this probability?

- Repetitions! Maintain many hash tables, each with a different locality-sensitive hash function, and try all of them.

LSH with Repetitions

0	1	2	3	4
(101, 37, 65)	(103,37,64)	(91,84,3)	(100,18,79)	

0	1	2	3	4
	(101,37,65) (103,37,64)	(91,84,3)		(100,18,79)

0	1	2	3	4
(101, 37, 65)		(103,37,64)		(91,84,3) (100,18,79)

Similarity

What Do We Mean by "Similar"?

- How can we measure the similarity of objects?
- Images in machine learning: often Euclidean distance (the distance we're familiar with on a day-to-day basis)
- What about *sets*?
  - Songs listened to by a user
  - Movies watched by a user
  - Human-generated tags given to an image
  - Words that appear in a document
- Need a way to measure set similarity

Set Similarity

Very similar!

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Khalid
Khalid	Drake
Drake	Travis Scott
Travis Scott	

- When are two sets similar?
- Let's look at our two sets. Similar if they have a lot of *overlap*
- I.e. : lots of artists in common, compared to total artists in either list

Set Similarity

Not very similar!

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	
Khalid	
Drake	
Travis Scott	

- When are two sets similar?
- Let's look at our two sets. Similar if they have a lot of *overlap*
- I.e. : lots of artists in common, compared to total artists in either list

Set Similarity

Moderately similar

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Ed Sheerhan
Khalid	Drake
Drake	Travis Scott
Travis Scott	Taylor Swift

- When are two sets similar?
- Let's look at our two sets. Similar if they have a lot of *overlap*
- I.e. : lots of artists in common, compared to total artists in either list

Jaccard Similarity

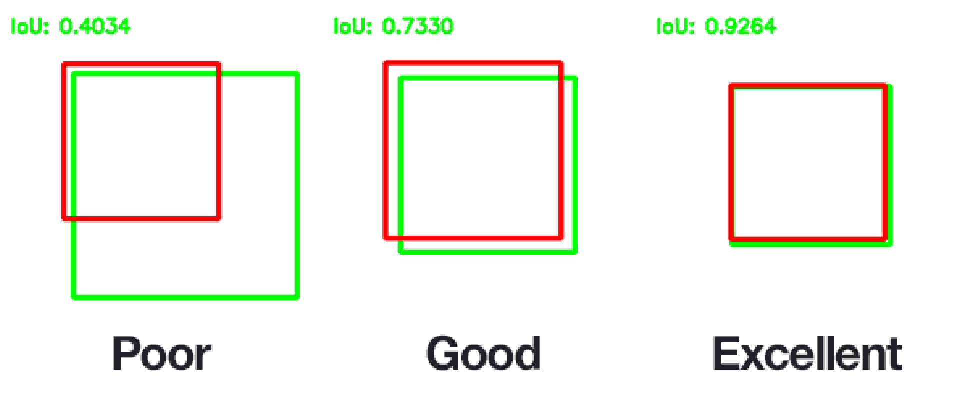
Jaccard Similarity

- Similarity measure for sets A and B
- Defined as:

$$\frac{|A \cap B|}{|A \cup B|}$$

- Intuitively: what fraction of these sets overlaps?

Jaccard Similarity Intuition 1



Jaccard Similarity Intuition 2

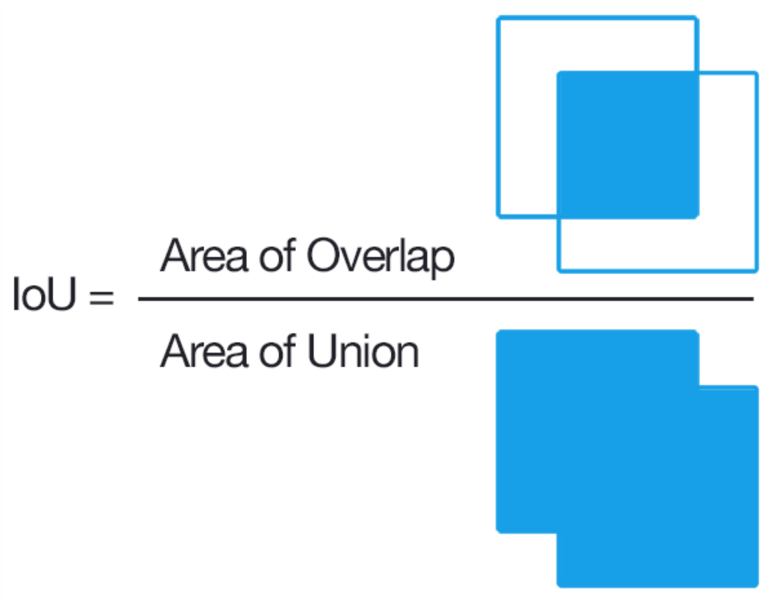
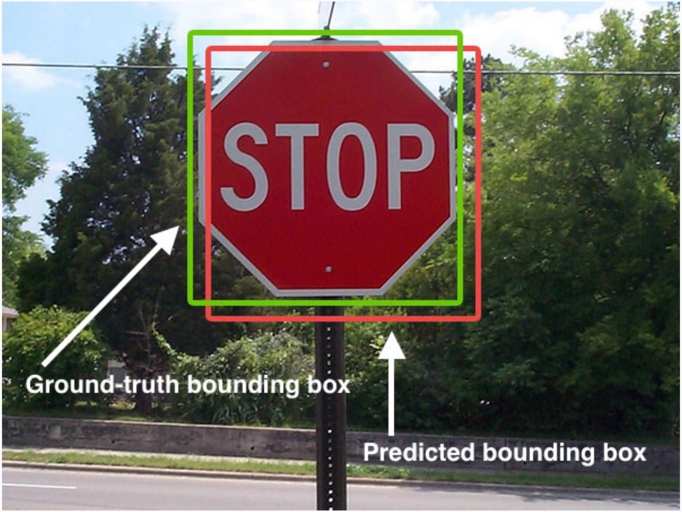


Image Search Example



Jaccard Example 1

Very similar!

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Khalid
Khalid	Drake
Drake	Travis Scott
Travis Scott	

- Similarity: $|A \cap B|/|A \cup B|$.
- $|A \cap B| = 4$
- $|A \cup B| = 5$
- Jaccard Similarity: $4/5 = .8$

Set Similarity

Not very similar!

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	
Khalid	
Drake	
Travis Scott	

- Similarity: $|A \cap B|/|A \cup B|$.
- $|A \cap B| = 1$
- $|A \cup B| = 5$
- Jaccard Similarity: $1/5 = .2$

Set Similarity

Moderately similar

User 1	User 2
Post Malone	Ariana Grande
Ariana Grande	Ed Sheerhan
Khalid	Drake
Drake	Travis Scott
Travis Scott	Taylor Swift

- Similarity: $|A \cap B| / |A \cup B|$.
- $|A \cap B| = 3$
- $|A \cup B| = 7$
- Jaccard Similarity: $3/7 = 0.428$

Jaccard Similarity: Properties

- Works on sets (each dimension is *binary*—an item is in the set, or not in the set)
- Always gives a number between 0 and 1
- 1 means identical, 0 means no items in common
- Jaccard similarity ignores items not in either set. So we learn nothing if neither of us like an artist. (Is this good?)
- Still works if one list is much longer than the other. (Generally, they'll have small similarity)

Locality-Sensitive Hash for Jaccard Similarity

- Want: items with high Jaccard Similarity are likely to hash together
- Items with low Jaccard Similarity are UNlikely to hash together
- Classic method: MinHash

MinHash

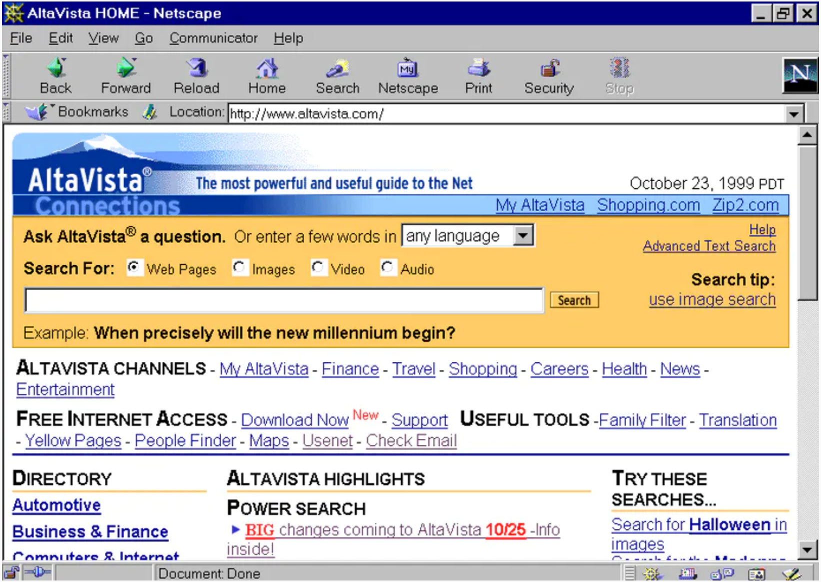
MinHash

- Developed by Andrei Broder in 1997 while working at AltaVista

- (AltaVista was probably the most popular search engine before Google, they wanted to detect similar web pages to eliminate them from search results)

- Now used for similarity search, database joins, clustering—LOTS of things.

AltaVista


AltaVista HOME - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Stop
Bookmarks Location: http://www.altavista.com/
AltaVista® The most powerful and useful guide to the Net October 23, 1999 PDT
Connections
My AltaVista Shopping.com Zip2.com
Ask AltaVista® a question. Or enter a few words in any language
Help
Advanced Text Search
Search For: Web Pages Images Video Audio
Search
Search tip: use image search
Example: When precisely will the new millennium begin?
ALTAVISTA CHANNELS - My AltaVista - Finance - Travel - Shopping - Careers - Health - News - Entertainment
FREE INTERNET ACCESS - Download Now New - Support USEFUL TOOLS -Family Filter - Translation - Yellow Pages - People Finder - Maps - Usenet - Check Email
DIRECTORY
Automotive
Business & Finance
ALTAVISTA HIGHLIGHTS
POWER SEARCH
BIG changes coming to AltaVista 10/25 -Info inside!
TRY THESE SEARCHES...
Search for Halloween in images
Document: Done

Bit Vectors

- Can represent any set as a vector of bits
- Each bit is an item. “1” means that that item is in the set, “0” means it’s not
- So if I’m keeping track of different people’s favorite colors, my universe may be $\{$red, yellow, blue, green, purple, orange$\}$
- If someone likes red and blue, we can store that information as 101000.
- Effective if universe is smallish; use a list for larger universe

Bit Vectors: Jaccard Similarity

- How can we determine $A \cap B$?
  - This is exactly A & B in C-style notation
- What about $A \cup B$?
  - This is exactly A | B in C-style notation
- We want the size of these sets—need to count the number of 1s in A & B, or A | B.

MinHash

128 in the assignment

- The hash consists of an *permutation* of all possible items in the universe
- To hash a set A: find the first item of A in the order given by the permutation. That item is the hash value!

MinHash example

- Let's stick with favorite colors, out of {red, yellow, blue, green, purple, orange}
- To hash, we randomly permute them. Let's say our current hash is given by the permutation (blue, orange, green, purple, red, yellow)
- First set is 101000 (same as {red, blue}). blue is in the set, so the hash value is blue.
- Second set is 110010 (we could also write {red, yellow, purple}). blue is not in the set; nor is orange; nor is green. purple is, so purple is the hash value

MinHash for Bit Vectors

- On the assignment, have bit vectors of length 128
- To get a hash function, we need a random permutation of the indices of these bits. That is to say, a random permutation of $\{0, 1, 2, \ldots, 127\}$
- To hash an item x, go through the random permutation. Find the first index i in the list such that the ith bit of x is 1.

For the sake of space let's do 8 bits

- Let's say $x = 10100101$, and the permutation is $(1, 5, 2, 0, 7, 6, 4, 3)$.
- Then the hash of x is 5.

MinHash

- A single MinHash: hashes each set to one of its elements (i.e. the position of one of its one bits)

- Not useful yet—output is too small

Analysis of Basic MinHash

Analysis

- What is the probability that $h(A) = h(B)$?
- Let's look at the permutation that defines h. We can ignore any item that is not in A or B.
- Look at the first index in the permutation that is in A or B (i.e. it is in $A \cup B$)
  - If this index is in *both* A and B, then $h(A) = h(B)$
  - If this index is in only one of A or B, then $h(A) \neq h(B)$
- Any index in $A \cup B$ is equally likely to be first. If the index is in $A \cap B$, they hash together; otherwise they do not
- Therefore: probability of hashing together is $|A \cap B|/|A \cup B|$.

MinHash as an LSH

- This means MinHash is an LSH!
- If two items have similarity at least r, they collide with probability at least r
- If two items have similarity at most cr, they collide with probability at most cr

Analysis: Phrased as bit vectors

- What is the probability that $h(A) = h(B)$?
- Let's look at the permutation that defines h. We can ignore any index that is 0 in both A and B.
- Look at the first index in the permutation that is 1 in A or B
  - If this index is in *both* A and B, then $h(A) = h(B)$
  - If this index is in only one of A or B, then $h(A) \neq h(B)$
- Any index that is 1 in $A|B$ is equally likely to be first. If the index is in $A\&B$, they hash together; otherwise they do not
- Therefore: probability of hashing together is (number of 1s in $A\&B$)/(number of 1s in $A|B$).

Analysis Example

- Let's say we have $A = \{\text{red, blue, green}\}$ and $B = \{\text{red, orange, purple, green}\}$.
- When do A and B hash together?
- If red or green appears before blue, orange, and purple then they hash together
- If blue or orange or purple appear before red and green, then they don't hash together
- Probability that red or green is first out of $\{\text{red, blue, green, orange, purple}\}$ is $2/5$.
- Therefore, A and B hash together with probability $2/5$.

Making Sure We Find the Close Pair

- To find the close pair, compare all pairs of items that hash to the same value
    - (We'll talk about how to do this later—it's similar to MiniMidterm 1)
- Let's say our close pair has similarity .5. How many times do we need to repeat?
- Each repetition has the close pair in the same bucket with probability .5. So need 2 repetitions in expectation.

An Aside on Expectation

Lemma 1

If a random process succeeds with probability p, then in expectation it takes $1/p$ iterations of the process before success.

Examples:

- It takes two coin flips in expectation before we see a heads
- We need to roll a 6-sided die 6 times before we see (say) a three.

Proof:

$$\sum_{i=1}^{\infty} ip(1-p)^{i-1} = \frac{p}{(1-(1-p))^2} = \frac{1}{p}$$

Concatenations and Repetitions

Problems with this Approach

- Buckets are really big!! (After all, lots of items are pretty likely to have a given bit set.)
- How can we decrease the probability that items hash together?
- Answer: concatenate multiple hashes together.

Concatenating Hashes

- Rather than one hash h, concatenate k independent hashes $h_1, h_2, \dots h_k$, each with its own permutation $P_1, P_2, \dots P_k$.
- To hash an item: repeat the process of searching through the permutation for each hash. Then concatenate the results together (can just use string concatenation)
- How does this affect the probability for sets A and B?
    - For each of the k independent hashes, A and B collide with probability $|A \cap B|/|A \cup B|$.
    - We only obtain the same concatenated hashes if *all* of the hashes are the same.
    - They are independent, so we can multiply to obtain probability $(|A \cap B|/|A \cup B|)^k$ of A and B colliding.

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash A.
    - First hash: red is in A.
    - Second hash: orange not in A, nor is green. Blue is in A.
    - Third hash: red is in A.
- Concatenating, we have $h(A) = \text{redbluered}$

Concatenation Example

- Let's say we have $A = \{\text{red, blue}\}$ and $B = \{\text{red, orange}\}$, and $k = 3$.
- $P_1 = \{\text{red, green, blue, orange}\}$,$P_2 = \{\text{orange, green, blue, red}\}$,$P_3 = \{\text{red, green, blue, orange}\}$
- Let's hash B.
  - First hash: red is in B.
  - Second hash: orange is in B.
  - Third hash: red is in B.
- Concatenating, we have $h(B) = \text{redorangered}$

Putting it all Together

- For each hash table, we concatenate k hashes.
- Hash all items into buckets. Check every pair of items in each bucket and see if it's the closest
- Quite often we'll get unlucky and the close pair won't be in the same bucket. What can we do?
- Need to repeat all of that multiple times until we find the close pair (let's say we repeat R times)
- So: overall need kR permutations
- What kind of values work for k and R?

Putting it Together: Analysis

- Let's say we have a set of n items $x_1, \ldots, x_n$
- The close pair of items has Jaccard similarity $3/4$
- Every other pair of items has similarity $1/3$
- How should we set k? How many repetitions R is it likely to take?

Putting it Together: Analysis (Finding k)

- Non-similar pairs have similarity $1/3$
- We want buckets to be small (have $O(1)$ size)
- Look at an element x_i. What is the expected size of its bucket?
- $\sum_{j \neq i} (1/3)^k$ (since x_i and any x_j with $j \neq i$ share a hash value with probability $1/3^k$)
- We can then solve $(n-1)(1/3)^k = 1$ to get $k = \log_3(n-1)$.

Putting it Together: Analysis (Predicting R)

- The similar pair has Jaccard similarity .75
- So they are in the same bucket with probability $(.75)^k$
- We have $k = (\log_3 n - 1)$. So....we need to do some algebra. (Let's assume that k is already an integer)
- $(.75)^{\log_3(n-1)} = 2^{\log_2(n-1)\log_2(3/4)/\log(3)} = (n-1)^{\log(3/4)/\log(3)} \approx 1/n^{.26}$
- So we expect about $R = n^{.26}$ repetitions. That's a lot!
- But it's essentially the best we know how to do.

Finding R and k in general

Let's say we have n points where the close pairs have similarity j_1, and all other pairs have similarity at most j_2

- First, set k so that each bucket has size $O(1)$: $k = \log_{1/j_2} n$.
  - Doable at home: show that this is the optimal value for k.
- Then, number of R we need in expectation is:

$$\left(\frac{1}{j_1}\right)^k = \left(\frac{1}{j_1}\right)^{\log_{1/j_2} n} = n^{\log_{(1/j_2)}(1/j_1)}.$$

Practical MinHash Considerations

So many Permutations!

- OK, so kR repetitions is a LOT of preprocessing, and a lot of random number generation
- And most of this won't ever be used! Most of the time, when we hash, we don't make it more than a few indices into the permutation.
- Idea: Instead of taking just the first hash item that appears in the permutation, take the first (say) 3. Concatenate them together. Then we just need $k/3$ permutations per hash table to get similar bounds.
- So let's say we have $A = \{$black, red, green, blue, orange$\}$, and we're looking at a permutation $P = \{$purple, red, white, orange, yellow, blue, green, black$\}$.
- Then A hashes to redorangeblue

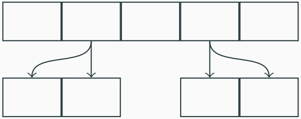
Reducing Permutations

- If you take the $\hat{k}$ first items when hashing, rather than just taking the first one, we only need $kR/\hat{k}$ total permutations.
- Does this affect the analysis?
  - Yes; the k we're concatenating for each hash table are no longer independent!
  - But this works fine in practice (and is used all the time)

Problems with Expectation

- We chose parameters so that buckets are small in expectation (i.e. on average)
- But: time to process a bucket is *quadratic*.
- So getting unlucky is super costly!
- What can we do if we happen to get a big bucket?

Handling Big Buckets



- One option: recurse!
- Take all items in any really large bucket, rehash them into subbuckets
- Might need to repeat
- This option can shave off small but significant running time
- (Not required; just one optimization suggestion.)

Assignment Parameters

- 128 bit integers (stored as two unsigned 64 bit ints "Item")
- Universe: $\{0, \ldots, 127\}$. (You can pretend that these are images, each of which is labelled with a subset of 128 possible tags.)
- Each bit is a 0 or 1 at random
- (Not realistic case, but hard case!)

What About Hashing?

- MinHash: go through each index in the permutation
- See if the corresponding bit is a 1 in the element we're hashing.
- How can we do this?
- Most efficient way I know is not clever. Just go through each index, and check to see if that bit is set (say by calculating `x & (1 << index)` —but remember that these are 128 bits)

Concatenating Indices

- Each time you hash you'll get k indices
- Each is a number from 0 to 127
- How can these get concatenated together?
- Option 1: convert to strings, call `strcat`
- Note: need to make sure to convert to *three-digit* strings! Otherwise hashing to 12 and then 1 will look the same as hashing to 1 and then 21. (012 and 001 instead)
- Option 2: Treat as bits. 0 to 127 can be stored in 7 bits. Store the hash as a sequence of k 8-bit chunks.

Getting a Good k

- In theory we want buckets of size 1.
- In practice, we want *slightly* bigger.
- Why? Lots of buckets and lots of repetitions have bad constants.
- Smaller k means fewer buckets, fewer repetitions (but bigger buckets and more comparisons)
- Start with $k \approx \log_3 n$, but experiment with slightly smaller values.

Repetitions?

- You're guaranteed that there exists a close pair in the dataset
- My implementation just keeps repeating until the pair is found (no maximum number of repetitions)
- The discussion of repetitions in the lecture is for two reasons: 1. analysis, 2. give intuition for the tradeoff by varying k

How to Deal with Buckets?

- Each time we hash, (i.e. build a new "hash table") need to figure out what hashes where so that we can compare elements with the same hash
- Unfortunately, we're not hashing to a number from (say) 0 to $n-1$. We're instead concatenating indices
- How to keep track of buckets?
- Similar to mini-midterm 1: may want to create buckets. Can also do it in-place using sorting.

Storing a Hash

- Just need a permutation on $\{0, \ldots, 127\}$
- How can we store that?
- First key observation: we (basically) *never* make it through the whole permutation (we'll always see at least one 1 first)
- Taking that a bit further: we only really need the first few indices. If we're using $\hat{k}$ indices from one ordering, something like $8\hat{k}$ or $16\hat{k}$ will almost certainly suffice.
- What about elements that hash further? Answer: just give them the value of the last index in the ordering.

Truncating Hash Example

- Let's say our permutation is $\{47, 11, 85, 64, 13, 74, 70, 107, 112, 103, 7, 95, 3, \ldots\}$ and $\hat{k} = 2$.

- I only store $\{47, 11, 85, 64, 13, 74, 107, 112\}$. If we go past 112 for some x, and we have not seen $\hat{k}$ indices that are a 1 in x, I just write 112 until I get $\hat{k}$ numbers.

Takeaway from Truncating Hashes

- This means we can store fewer bits, fewer random numbers
- Might be easier to handle. (Arrays of size 16-20 are nicer than arrays of size 128.)

Back to SIMD

SIMD on lab computers

```
(gdb) print $ymm0
$1 = {
 v8_float={0,0,0,0,0,0,0,0},
 v4_double={0,0,0,0},
 v32_int8={0 <repeats 32
     times>},
 v16_int16={0 <repeats 16
     times>},
 v8_int32={0,0,0,0,0,0,0,0},
 v4_int64={0,0,0,0},
 v2_int128={0,0}
}
```

- We have SSE, AVX, AVX2 instruction sets (don't have AVX-512)
- 16 "YMM" registers; each 256 bits
- (Older processors may only have 128 bit "XMM" registers.)
- Need to include `#include <immintrin.h>` and compile with `-mavx2`

SIMD Examples

What is SIMD good for?

- Lots of identical operations on a set of elements; these operations are costly

- Elements are in nicely-sized chunks
    - Can always used specialized code to handle other cases

Example 1: Adding two arrays

- Let's add two arrays of 8 32-bit integers with one SIMD operation

- `simdtests.c`

Assembly examples

```
.LBE24:
# simdtests.c:23:   __m256i b = _mm256_set_epi32(B[7], B[6], B[5], B[4], B[3], B[2], B[1], B[0]);
    .loc 1 23 14
    vmovdqa %ymm0, 160(%rsp)    # D.25654, b
    vmovdqa 128(%rsp), %ymm0    # a, tmp178
    vmovdqa %ymm0, 256(%rsp)    # tmp178, __A
    vmovdqa 160(%rsp), %ymm0    # b, tmp179
    vmovdqa %ymm0, 288(%rsp)    # tmp179, __B
```

```
    .loc 3 121 33
    vpaddd  %ymm0, %ymm1, %ymm0 # _76, _75, _77
.LBE27:
```

Example 2: Adding single value to array

- Let's add one value (10) to an array.
- Do we need to declare a new array to do this? Or can we make a vector of 10s manually?

Speed comparison

- How much time does SIMD add (in total in our implementation) take compared to normal add?

- It's a bit faster

Example 3: Searching for Particular Value in Array

- Can do vector comparisons, but get a 256 bit vector out
- Need a way to make that vector into something useful for us. Let's look at the code.
- `int _mm256_movemask_epi8(_mm256 arg)`: returns a 32 bit `int` where the *i*th bit of the `int` is the first bit in the *i*th byte of the argument `arg`

Optimization comparison?

- What happens when we change to O3?
- Everything gets faster!
- In previous tests: for adding, normally suddenly outpaces SIMD; finding the 0 element doesn't
- Guesses as to why? ...Let's take a look at the assembly
    - gcc is vectorizing the operations by itself and doing it very slightly better

SIMD Discussion

Tradeoffs

What are some downsides of using an SIMD instruction?

- SIMD instructions may be a little slower on a per-operation basis (folklore is a factor of ≈ 2 even for the operation itself, but it seems modern implementations are much better)
- Cost to gather items in new location
- SIMD is *not* always faster

How much can we save using SIMD? Let's say we're using 256 bit registers, and operating on 32 bit data.

- Factor of $256/32 = 8$ at absolute best
- Realistically is going to be quite a bit lower in practice

Tradeoffs

- Bear in mind Amdahl's law when considering SIMD
- Only worth using on the most costly operations, and only when they work very well with SIMD

One Question

- What's a problem we've seen this semester that is particularly suited for SIMD speedup?
    - Hint: I'm not referring to any of the assignment problems

- Matrix multiplication: lots of time doing multiplications on successive matrix elements

- (SIMD works for some other problems too; I just wanted to highlight this as one of the classic examples.)

Compiler?

- A lot of the examples we saw were super simple
- Can the compiler use these operations automatically?
- As we just saw: yes it can
  - `--ftree-vectorize`
  - `--ftree-loop-vectorize` (turned on with `O3`)
  - Lots of extra option to tune `gcc` parameters for how it vectorizes
- But, as always, only is going to work in "obvious" situations.

Automatic Vectorization Example

```c
void addArrays(int* A, int* B, int size){
    for(int i = 0; i < size; i++) {
        A[i] += B[i];
    }
}


int main() {

    int* A = malloc(800*sizeof(*A));
    int* B = malloc(800*sizeof(*B));

    for(int i = 0; i < 800; i++) {
        A[i] = i;
        B[i] = 800 -i;
    }

    addArrays(A, B, 8);
```

```
# autosimd.c:10:        A[i] += B[i];
    .loc 1 10 8 is_stmt 0 discriminator 3 view .LVU7
    movdqu  (%rdi,%rax), %xmm0  # MEM[base: A_12(D), in
    movdqu  (%rsi,%rax), %xmm1  # MEM[base: B_13(D), in
    paddd   %xmm1, %xmm0    # tmp154, vect__7.16
    movups  %xmm0, (%rdi,%rax)  # vect__7.16, MEM[base:
    .loc 1 9 27 is_stmt 1 discriminator 3 view .LVU8
    .loc 1 9 17 discriminator 3 view .LVU9
    addq    $16, %rax   #, ivtmp.30
```

We can see the `paddd` SIMD instruction (on `xmm1` and `xmm0`) when compiling with `-O3`.