Randomized and Big Data Algorithms, Approximate Nearest Neighbor Search

Anton Belyy

Roadmap

- Nearest neighbor search
    - Task definition
    - Naïve approaches
    - LSH
- Minwise hashing
    - Motivation
    - Connection to LSH
    - Efficient algorithm

NN and *c*-ANN search: recap

Given a set *P* of *n* points in $\mathbb{R}^d$ (assuming $d \gg 1$) and a metric *D*, construct a data structure that, given any point *q* in $\mathbb{R}^d$ …

Nearest Neighbor Search (NN): return the **closest** point $p \in P$, such that $p = \operatorname*{argmin}_{p \in P} D(p, q)$

***c*-Approximate Nearest Neighbor Search (*c*-ANN)**: return a **close** point $p \in P$, such that $D(p, q) \leq c \min_{p'} D(p', q)$

Naïve Approaches to NN search

- Approach #1: Brute Force

  preprocessing time: $O(1)$

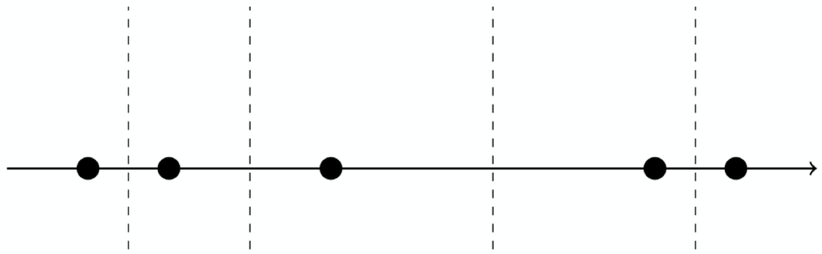
  query time: $\boldsymbol{O(nd)}$

  space: $O(nd)$
- Approach #2: Binary Search Tree ($d = 1$)

  preprocessing time: $O(n \log n)$

  query time: $O(\log n)$

  space: $O(n)$



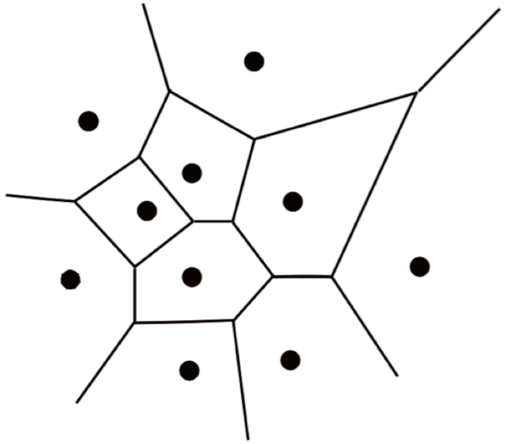
Less Naïve Approaches to NN search

- Voronoi Diagram ($d = 2$)

  preprocessing time: $O(\mathrm{n} \log n)$

  query time: $O(\log n)$

  space: $O(n)$

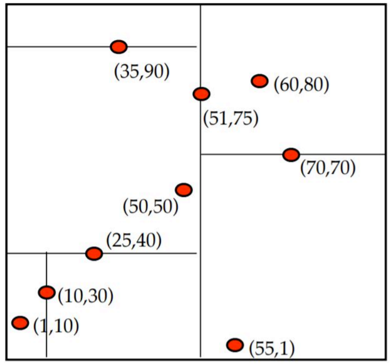


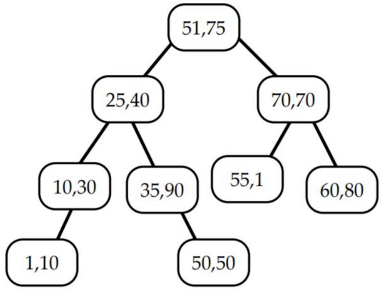
- k-d Tree (small *d*)

  preprocessing time: $O(nd \log n)$

  query time: $\boldsymbol{O(\log n + 2^d)}$

  space: $O(nd)$ “Curse of dimensionality”





https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/kdtrees.pdf
https://courses.cs.washington.edu/courses/cse373/02au/lectures/lecture22l.pdf

Solve *c*-ANN with dimensionality reduction?

JL Lemma: Given a set of P points in $\mathbb{R}^d$, there exists a mapping f: $\mathbb{R}^d \mapsto \mathbb{R}^k$, such that for any two points p_i and p_j, their distance $\left\|p_i - p_j\right\|_2$ is preserved with $(1 \pm \varepsilon)$ multiplicative error, and k = $O\left(\frac{1}{\varepsilon^2}\log n\right)$

Corollary: k-d trees with JL dimensionality reduction has:

preprocessing time: $O\left(n\frac{\log^2 n}{\varepsilon^2}\right)$

query time: $\boldsymbol{n^{O(1/\varepsilon^2)}}$

space: $O\left(n\frac{\log n}{\varepsilon^2}\right)$

Still super-linear dependence on *n* for small $\boldsymbol{\varepsilon}$

Sublinear time algorithms for *c*-ANN

We will show an alternative way of distance-preserving dimensionality reduction from d to k, based on **locality sensitive** hash functions.

Using a simple lookup over multiple hash tables, we will solve c-ANN problem in provably **sublinear** time.

All we need is to construct a locality-sensitive **hash family** for a given metric D we care about.

Locality-Sensitive Hashing (LSH) Framework
(Indyk and Motwani, 1998)

Definition: A hash family $\mathcal{H} = \{h\colon\ U \mapsto S\}$ is called $(\boldsymbol{r_1}, \boldsymbol{r_2}, \boldsymbol{p_1}, \boldsymbol{p_2})$-**LSH** for a metric function D, if for all points $x, y \in U$,

1. If $D(x, y) \le r_1$, then $\Pr_{\mathrm{h} \in \mathcal{H}}[h(x) = h(y)] \ge p_1$
2. If $D(x, y) > r_2$, then $\Pr_{\mathrm{h} \in \mathcal{H}}[h(x) = h(y)] \le p_2$

Compare:

2-universal hashes: $\Pr[h(x) = h(y)] \le \frac{1}{|S|}$

Locality sensitive hashes: $\Pr[h(x) = h(y)] \propto \frac{1}{D(x,y)}$

Locality-Sensitive Hashing (LSH) Framework
(Indyk and Motwani, 1998)

Exercise (LSH for Hamming distance):

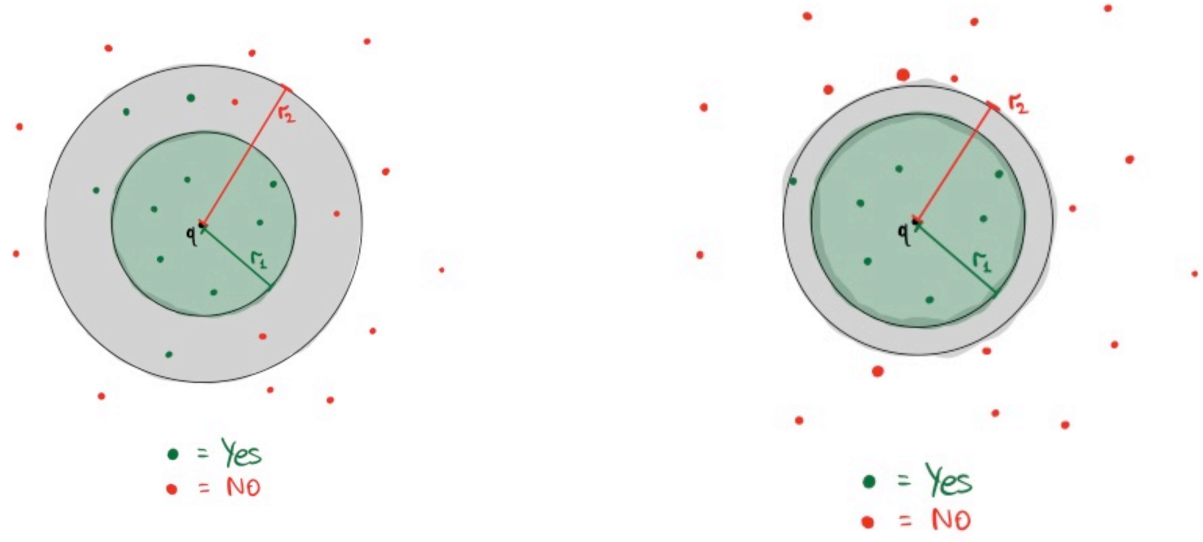
$\mathrm{U} = \{0,1\}^{\mathrm{d}}$, $\mathcal{H} = \{h_i\colon h_i(x) = x_i, i \in [d]\}$,

$D(x,y) = \sum_{i=1}^{d} [x_i \neq y_i]$. Show that $\mathcal{H}$ is $\left(r, cr, 1 - \frac{r}{d}, 1 - \frac{cr}{d}\right)$-LSH.

(c,r)-ANN: a bridge between LSH and c-ANN

$(\boldsymbol{c}, \boldsymbol{r})$**-ANN**: Given a set P of points in $\mathbb{R}^d$, construct a data structure for the set P, such that given any point q in $\mathbb{R}^d$,

1. If there exists $p \in P$ such that $D(p,q) \leq r$, return **YES** and *any* point $p' \in P$ with $D(p',q) \leq cr$
2. If there is no point $p \in P$ s.t. $D(p,q) \leq cr$, return **NO**



Using (c,r)-ANN to solve c-ANN

Lemma 1: Given an algorithm for $(1+\varepsilon, r)$-ANN (for all r), there exists an algorithm for $(1+\varepsilon)$-ANN

Idea: construct (c,r)-ANN instances over increasing radii r:

$$D_{min}, \quad (1+\varepsilon)D_{min}, \quad (1+\varepsilon)^2 D_{min}, \quad \dots, \quad D_{max}$$

Use **binary search** to find the smallest ball that contains points from P.

Using LSH to solve (*c*,*r*)-ANN

Theorem 1: Given an (r, cr, p_1, p_2)-LSH family for a metric *D*, there exists an algorithm for (c, r)-ANN, which succeeds with probability > 0 and uses:

preprocessing time: $O(n^{1+\rho} \log n)$*

query time: $O(n^{\rho} \log n)$*

space: $O(n^{1+\rho} \log n)$, where $\rho = \dfrac{\ln 1/p_1}{\ln 1/p_2} < 1$

We will prove this constructively, i.e. by showing an algorithm.

Corollary: Having LSH for a metric *D* allows for a sublinear (*c*,*r*)-ANN search, since $O(n^{\rho} \log n) = o(n)$.

* measured in the number of hash evaluations

Using LSH to solve (*c*,*r*)-ANN

Index: for each data point $p \in P$, compute *L* hash signatures of size *K* each, using LSH (denoted as $g_i(p), i = 1 \dots L$), and store *p* in *L* hash tables ($HT_i, i = 1 \dots L$), where each HT_i is indexed by $g_i(p)$:

```
for i ∈ {1,2,...L} do   // Initialize L empty hash tables
   HT_i = HashTable()
forall p ∈ P do         // Insert p's into L hash tables
   for i ∈ {1,2,...L} do
      HT_i[g_i(p)].add(p)
```

Using LSH to solve (c,r)-ANN

Query: given a query q, look for potential c-ANN candidates of q in each of HT_i hash tables. Stop after checking exactly $2L$ candidates.

```
num_checked = 0
for i ∈ {1,2, … L} do
    for p′ ∈ HT_i[g_i(q)]:
      num_checked += 1
      if D(p′, q) ≤ cr: return (YES, p′)
      if num_checked == 2L: return NO
return NO
```

Using LSH to solve (c,r)-ANN

Correctness proof:

We need to show that, given a query q,

1) there are at most $2L - 1$ "false positives" ($D(p', q) > cr$, but $\exists j': g_{j'}(p') = g_{j'}(q)$)
2) if there is a "true positive" c-ANN $p \in P$, then $\exists j: g_j(p) = g_j(q)$

With probability > 0

Using LSH to solve (*c*,*r*)-ANN

Claim #1: there are at most $2L - 1$ "false positives" ($D(p', q) > cr$, but $\exists j'$: $g_{j'}(p') = g_{j'}(q)$) with probability > 1/2

Recall from LSH that, if $D(p', q) > cr$, then $\Pr_{\mathrm{h} \in \mathcal{H}}[h(p') = h(q)] \leq p_2$

If we compute *K* independent hashes, then the probability is $\leq p_2^K$

Since we have *n* points and *L* hash tables, we can upper bound:

$\mathrm{E}[\text{false positives}] \leq p_2^K \cdot n \cdot L = \frac{1}{n} \cdot n \cdot L = L$ **(choose $\boldsymbol{K = \log_{1/p_2} n}$)**

Then, by Markov's inequality:

$$\Pr[\text{false positives} > 2L] \leq \frac{\mathrm{E}[\text{false positives}]}{2L} \leq \frac{L}{2L} \leq \frac{1}{2}.$$

Using LSH to solve (*c*,*r*)-ANN

Claim #2: if there is a "true positive" c-ANN $p \in P$, then $\exists j: g_j(p) = g_j(q)$

Recall from LSH that if $D(p,q) \leq r$, then $\Pr_{\mathrm{h} \in \mathcal{H}}[h(p) = h(q)] \geq p_1$

If we compute *K* independent hashes, then the probability is at least

$$p_1^K = p_1^{\log_{1/p_2} n} = n^{-\frac{\ln 1/p_1}{\ln 1/p_2}} = n^{-\rho}$$

Considering all L hash tables **(and choose $\boldsymbol{L = n^{\rho}}$)**, we get:

$$\left(1 - \frac{1}{x}\right)^x < \frac{1}{e}$$

$$1 - (1 - n^{-\rho})^L = 1 - (1 - n^{-\rho})^{n^\rho} \geq 1 - \frac{1}{e} \geq \frac{1}{2}$$

Using LSH to solve (c,r)-ANN

Since both claims are true w.p. at least 0.5, then by the union bound,

$$\Pr[(1\text{ is true})\text{ AND }(2\text{ are true})] = 1 - \Pr[(1\text{ is false})\text{ OR }(2\text{ is false})]$$
$$\geq 1 - \Pr[1\text{ is false}] - \Pr[2\text{ is false}] > 1 - \frac{1}{2} - \frac{1}{2} > 0.$$

\end{proof}

Thus, with a constant probability, we can find c-ANNs by "mapping" to to hash signatures of size $K*L$, and checking at most $2*L$ candidates.

How small is this compared to *n*? *nd*?**

Discussion

- Computing many hash signatures per data point is a bottleneck of LSH:
  - Theorem 1 requires a signature of size $K * L = O\left(n^{\rho} \log_{1/p_2} n\right)$:

n \ (p_1, p_2)	(0.6, 0.4)	(0.7, 0.3)	(0.8, 0.2)	(0.9, 0.1)
$n = 10^5$	7,702	290	35	8
$n = 10^6$	33,365	687	58	11
$n = 10^7$	140,518	1586	94	15

$$\rho = \frac{\ln 1/p_1}{\ln 1/p_2}$$

  - In practice, *K*L* is around 100..1000

⇒ Fast computation of signatures is key to practical success in LSH

Outline

- Nearest neighbor search
  - Task definition
  - Naïve approaches
  - LSH
- Minwise hashing
  - Motivation
  - Connection to LSH
  - Efficient algorithm

Fingerprints for Duplicate Web Page Detection

Parallel to LSH framework, a method to filter duplicate web pages in search results was developed by Andrei Broder

Idea: compute short **fingerprints** of web pages, such that similar web pages get similar fingerprints, where similarity is measured by **Jaccard Index**

This could be seen as a special case LSH where D = 1 - Jaccard Index

Andrei Broder

Minwise Hashing

Jaccard Index is a similarity function between two sets A and B, defined as

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

Lemma 2 (Minwise Hashing): Let A and B be subsets of [d] and $h_\pi(X) = \min_{x \in X} \pi(x)$, where π is a permutation of $[d]$. Then $\Pr_\pi[h_\pi(A) = h_\pi(B)] = J(A,B)$

Corollary: $\mathcal{H} = \{h_\pi : \pi \text{ is a permutation of } [d]\}$ is $(r_1, r_2, 1 - r_1, 1 - r_2)$-LSH for Jaccard distance: $D(A,B) = 1 - J(A,B)$

Minwise Hashing

Corollary: $\mathcal{H} = \{h_\pi : \pi \text{ is a permutation of } [d]\}$ is $(r_1, r_2, 1 - r_1, 1 - r_2)$-LSH for Jaccard distance: $D(A, B) = 1 - J(A, B)$

Practical Issues with Minwise Hashing

- Suppose we want to compute *m* hashes ⇒ need *m* permutations of [*d*]
- Storing *m* permutations naively requires $m \log d! = O(md \log d)$ bits
  - Typically $\mathrm{m} \sim 10^2 \ldots 10^3$ and $d = 2^{32} \ldots 2^{64}$, so this is not acceptable
- Later, Broder et al.* showed this to be sufficient for π (for any $X \subseteq [d]$ and any $x \in X$):

$$\Pr_{\pi \sim S_d}[\min\{\pi(X)\} = \pi(x)] = \frac{1}{|X|}$$

X	π_1	π_2	π_3	π_4
2	**2**	**3**	10	3
3	6	4	**4**	6
4	15	14	16	15
6	11	10	7	5
7	8	7	5	**1**

$\mathrm{d} = 2^4$

* Broder, Charikar, and Mitzenmacher. Min-Wise Independent Permutations. 1998

Practical Issues with Minwise Hashing

- In practice, $\pi(x)$ is replaced with a 2-universal hash $g(x)$ for $x \in X$
- There might be collisions, i.e. $g(x_1) = g(x_2)$ when $x_1 \neq x_2$
  - When d is large and $n \ll d$, this can be neglected
- Still, time complexity $O(nm)$, where m is the number of hashes
- In LSH, we need 1000s of hashes $\Rightarrow$ this is not acceptable

X	g_1	g_2	g_3	g_4
2	15	14	12	5
3	**1**	8	10	15
4	6	3	**4**	9
6	5	8	9	8
7	3	**1**	15	**7**

$\mathrm{d} = 2^4$

Minwise Hashing with Exponential Sampling

- Suppose we replace 2-universal hash $g(x)$ with a random variable $\widetilde{g_x} \sim Exp(1)$ ($x \in X \subset [d]$)

- **Lemma 3:** Let A and B be subsets of $[d]$ and $h_{\tilde{g}}(X) = \min_{x \in X} \tilde{g}_x$, where $\tilde{g}_x$'s are i.i.d. r.v.s drawn from $Exp(1)$. Then $\Pr_{\tilde{g}}\left[h_{\tilde{g}}(A) = h_{\tilde{g}}(B)\right] = J(A, B)$

X	$\tilde{g}_1$	$\tilde{g}_2$	$\tilde{g}_3$	$\tilde{g}_4$
2	0.80	1.26	0.92	0.78
3	0.55	1.04	0.58	2.22
4	3.31	**0.48**	1.57	0.75
6	0.84	2.60	**0.07**	**0.09**
7	**0.02**	1.79	1.51	2.04

$d = 2^4$

Minwise Hashing with Exponential Sampling

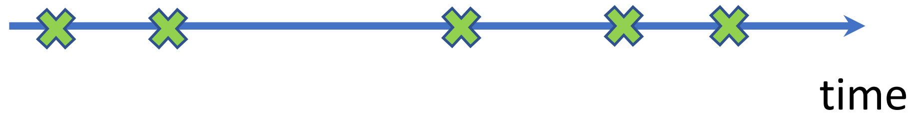
Lemma 3: Let A and B be subsets of $[d]$ and $h_{\tilde{g}}(X) = \min_{x \in X} \tilde{g}_x$, where $\tilde{g}_x$'s are i.i.d. r.v.s drawn from $Exp(1)$. Then $\Pr_{\tilde{g}}\left[h_{\tilde{g}}(A) = h_{\tilde{g}}(B)\right] = J(A, B)$

Minwise Hashing with Poisson Processes

- Why sample *n* values and throw away $(n-1)$ of them, when we can **sample the minimum** of exponentials **directly**?
- This can be done with **Poisson processes***

*Mitzenmacher and Upfal. Probability and Computing, 2nd edition, Section 8.4

Intro to Poisson Processes



- Definition:
  - **Continuous time** process with parameter λ
  - Number of arrivals in disjoint time intervals are **independent**
  - Number of arrivals in interval of duration τ, or N_τ, is $\boldsymbol{Poisson}(\lambda\tau)$-distributed and is the same for any interval $[t_0; t_0 + \tau]$
  - For very small intervals δ: $P(k,\delta) \approx \begin{cases} 1-\lambda\delta, & \text{if } k = 0 \\ \lambda\delta, & \text{if } k = 1 \\ 0, & \text{if } k > 1 \end{cases}$
- Useful property: X_1 – time of the 1st arrival – is $\boldsymbol{Exp}(\lambda)$-distributed, i.e. $P(X_1 > t) = e^{-\lambda t}$

MIT RES.6-012 Introduction to Probability, Spring 2018. Lecture 22.2

Minwise Hashing with Poisson Processes

- Why sample *n* values and throw away $(n-1)$ of them, when we can **sample the minimum** of exponentials **directly**?
- This can be done with **Poisson processes***
- **Idea**: per each $x \in X$, start a Poisson process P_x that generates hash values for all *m* signatures and stops, when cannot improve current min

$X = \{2,3\}, \qquad m = 4$

time = hash value

$P_2\colon\ (3, 0.8), \quad (1, 2.0), \quad (2, 2.4), \quad (4, 3.1)$

$P_3\colon\ (4, 0.1), \quad (2, 1.6), \quad (2, 3.5)$

3.5 > 3.1, stop early

*Mitzenmacher and Upfal. Probability and Computing, 2nd edition, Section 8.4

Minwise Hashing with Poisson Processes

- Why sample *n* values and throw away $(n-1)$ of them, when we can **sample the minimum** of exponentials **directly**?
- This can be done with **Poisson processes***
- **Idea**: per each $x \in X$, start a Poisson process P_x that generates hash values for all *m* signatures and stops, when cannot improve current min

$X = \{2,3\}$, $m = 4$

time = hash value

P_2: $(3, 0.8)$, $(1, 2.0)$, $(2, 2.4)$, $(4, 3.1)$

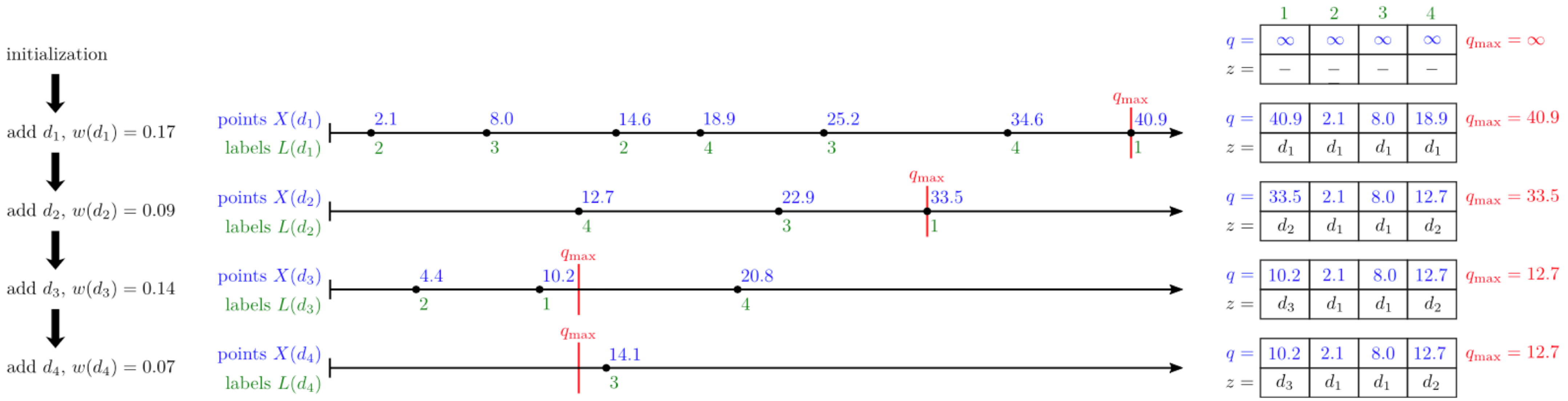
P_3: $(4, 0.1)$, $(2, 1.6)$, $(2, 3.5)$

3.5 > 3.1, stop early

	h_1	h_2	h_3	h_4
{2}	2.0	2.4	0.8	3.1
{2, 3}	**2.0**	**<u>1.6</u>**	**0.8**	**<u>0.1</u>**

*Mitzenmacher and Upfal. Probability and Computing, 2nd edition, Section 8.4

ProbMinHash algorithm (TKDE 2020)



- Each arrival is labeled with a hash signature label $L \sim Uniform(\{1,2,\dots,m\})$
- By the **split** property, a *sub-process* where all labels are $L = l_i$ is also Poisson with rate $\frac{\lambda}{m}$
- If $\lambda = m$, then the **arrival time** of the **first event** in each sub-process is $\boldsymbol{Exp(1)}$**-distributed**

Otmar Ertl. ProbMinHash A Class of Locality-Sensitive Hash Algorithms for the (Probability) Jaccard Similarity. 2020

ProbMinHash algorithm (TKDE 2020)

- Processing of the first element takes $O(m \log m)$ time (Coupon Collector argument)
- Processing of the i-th element takes $O(\frac{1}{i} m \log m)$ time (due to early stop)
- Overall, the algorithm runs in $O\left(n + \sum_{i=1}^{n} \frac{1}{i} m \log m\right) = O(n + m \log^2 m)$ time
- Substantial improvement over $O(nm)$ in Broder's method!

Input: X
Output: $h_1, h_2, \dots, h_m$

$h_1, h_2, \dots, h_m \leftarrow (\infty, \infty, \dots, \infty)$
forall $x \in X$ **do**
  R ← **new** PRNG with seed x
  $h \leftarrow R[Exp(1)]$
  while $h < h_{max}$ **do**
    $k \leftarrow R[Uniform(\{1,2,\dots,m\})]$
    if $h < h_k$ **then**
      $h_k \leftarrow h$
      $h_{max} \leftarrow \max(\{h_1, h_2, \dots, h_m\})$
      if $h \geq h_{max}$ **then break**
    $h \leftarrow h + R[Exp(1)]$

early stop

Otmar Ertl. ProbMinHash A Class of Locality-Sensitive Hash Algorithms for the (Probability) Jaccard Similarity. 2020
https://github.com/oertl/probminhash

Summary

- NN search is challenging in high-dimensional settings
- Exact algorithms scale at least linearly with either n or d
- LSH provides sublinear algorithms for c-ANN search

- Minwise hashing is a conceptually simple way to do c-ANN search with Jaccard distance as a target metric
- Poisson process-based sampling provides a much faster and statistically equivalent way to perform Minwise Hashing