Efficient algorithms for fingerprint similarity search and diversity selection

Andrew Dalke
dalke@dalkescientific.com

Cambridge Cheminformatics Network Meeting
8 June 2022

Fingerprint Similarity Search is *Easy!*

- **binary fingerprints - not count fingerprints**
- **"short" - typically in the 100s or 1,000s of bits**
- **"dense" - typically 5%-40% of the bits are on**
- **exact - not approximate similarity**

Tanimoto Similarity

Tanimoto(FP_A, FP_B) = $\| FP_A \wedge FP_B \| / \| FP_A \vee FP_B \|$

= number of bits in intersection / number of bits in the union

= c / (A + B - c)

Avoid double-counting the bits in common

I follow Daylight nomenclature

A is the count of the bits on in object A.
B is the count of the bits on in object B.
c is the count of the bits on in both object A and object B.

"This nomenclature differs from that used by others, in particular the Sheffield group."

https://www.daylight.com/dayhtml/doc/theory/theory.finger.html

A Friday Afternoon...

... in 1986, shortly after publications by Willett, Winterman, Bawden

```
% rdkit2fps --maccs166 chembl_30.sdf.gz -o maccs.fps
% head maccs.fps
#FPS1
#num_bits=166
#type=RDKit-MACCS166/2
#software=RDKit/2021.09.4 chemfp/4.0
#source=/Users/dalke/databases/chembl_30.sdf.gz
#date=2022-05-28T20:56:45
0000000108440020 01b49d900002c01354a178a813   CHEMBL153534
0000000100043000 01f6bedf897afe838d3ffeff1b   CHEMBL440060
0000000100043000 81f20ecf8972fed59d7ff6ff1f   CHEMBL440245
```

Examples of chemfp's "FPS" fingerprint exchange format.

Tanimoto in Python

```
>>> fpA = 0x00000003000000001d414d91323915380f138ea1f  # CHEMBL113
>>> fpB = 0x00000003000000001d414d91323915380e178ea1f  # CHEMBL1114
>>> union = fpA | fpB
>>> intersection = fpA & fpB
>>>
>>> fpA.bit_count()
46
>>> fpB.bit_count()
46
>>> union.bit_count()
47
>>> intersection.bit_count()
45
>>> tanimoto = intersection.bit_count() / union.bit_count()
>>> tanimoto
0.9574468085106383
```

int.bit_count() added in Python 3.10

Caffeine and theobromine have very similar MACCS fingerprints.

What's similar to caffeine?

```
# MACCS fingerprint for caffeine
query = 0x000000003000000001d414d91323915380f138ea1f

hits = []
for line in open("maccs.fps"):
    # skip header
    if line[:1] == "#":
        continue

    # decode the fingerprint and compute the Tanimoto
    hex_value, id = line.split()
    fp = int(hex_value, 16)
    score = ((query & fp).bit_count() /
             (query | fp).bit_count()    )

    # Must be at least 0.95 similar
    if score >= 0.95:
        hits.append( (score, id) )

# Order from highest to lowest score
for score, id in sorted(hits, reverse=True):
    print(id, score)
```

```
CHEMBL87121 1.0
CHEMBL74063 1.0
CHEMBL113 1.0
CHEMBL89062 0.9787234042553191
CHEMBL1498670 0.9787234042553191
CHEMBL143715 0.9787234042553191
CHEMBL284855 0.9782608695652174
CHEMBL600294 0.9583333333333334
CHEMBL483663 0.9583333333333334
CHEMBL480529 0.9583333333333334
CHEMBL26897 0.9583333333333334
CHEMBL26119 0.9583333333333334
CHEMBL21053 0.9583333333333334
CHEMBL1767 0.9583333333333334
CHEMBL1595028 0.9583333333333334
CHEMBL440329 0.9574468085106383
CHEMBL226211 0.9574468085106383
CHEMBL190 0.9574468085106383
CHEMBL1158 0.9574468085106383
CHEMBL113241 0.9574468085106383
CHEMBL1114 0.9574468085106383
```

Takes 2.5 seconds to search 2.1M MACCS keys.

Fast Fingerprint Similarity Search is Not Easy!

Why go fast?

- **Faster results enable more exploration.**
- **Sub-second response helps maintain flow. (Miller 1968)**
- **Can deploy on a wider range of hardware:**
  - **Add search everywhere, including your web server.**
  - **"The laptop you have is better than the GPU you don't."**
- **Emotional:**
  - **Don't like "wasting" the computer.**
  - **Want a faster implementation than anyone else.**
  - **Annoyed by people who say their method is fast, but use a slow implementation as their baseline.**

Python is slow

Use a compiled language like C, C++, Rust, Julia, ...

As a rough estimate, Python is 50x slower than C.

Depends very much on the algorithm!

Caffeine search from Python takes chemfp 1.7 milliseconds.

Chemfp uses Python/C extensions with some inline assembly.

Avoid processing lines of text

- **Process in chunks of text, or,**
- **Convert to a binary form which is easier to process**

I pre-processed the ChEMBL fingerprints into a list, and saved it to a file as a Python pickle.

Search time dropped from 2.5 seconds to 1.6 seconds.
Of which only 0.77 seconds was spent in search.

In chemfp I developed the "FPB" format for fast load times.

Match Data to Representation

Python integers can be arbitrary length.

They store a count, and an array of that many "digits". Each digit is a 15- or (usually) 30-bit value. (Base 1,073,741,824)

More efficient to store a fingerprint as a byte string.
Easier for C code to access the fingerprint bits.

Avoid intermediate objects

The Tanimoto calculation creates two temporary integers.

```
score = ((query & fp).bit_count() /
         (query | fp).bit_count()   )
```

Instead, work more directly on the representation.

```
score = byte_tanimoto(query, fp)
```

Byte-string search

```
import pickle
from chemfp.bitops import byte_tanimoto

# Load the pickled fingerprints
with open("byte_maccs.pkl", "rb") as f:
    fp_data = pickle.load(f)

# MACCS fingerprint for caffeine
query = bytes.fromhex("000000003000000001d414d91323915380f138ea1f")

hits = []
for id, fp in fp_data:
    score = byte_tanimoto(query, fp)

    # Must be at least 0.95 similar
    if score >= 0.95:
        hits.append( (score, id) )

# Order from highest to lowest score
for score, id in sorted(hits, reverse=True):
    print(id, score)
```

Now, only 0.60 seconds in search. (Earlier was 0.77 seconds.)

(Overall performance is about the same because the load time increases.)

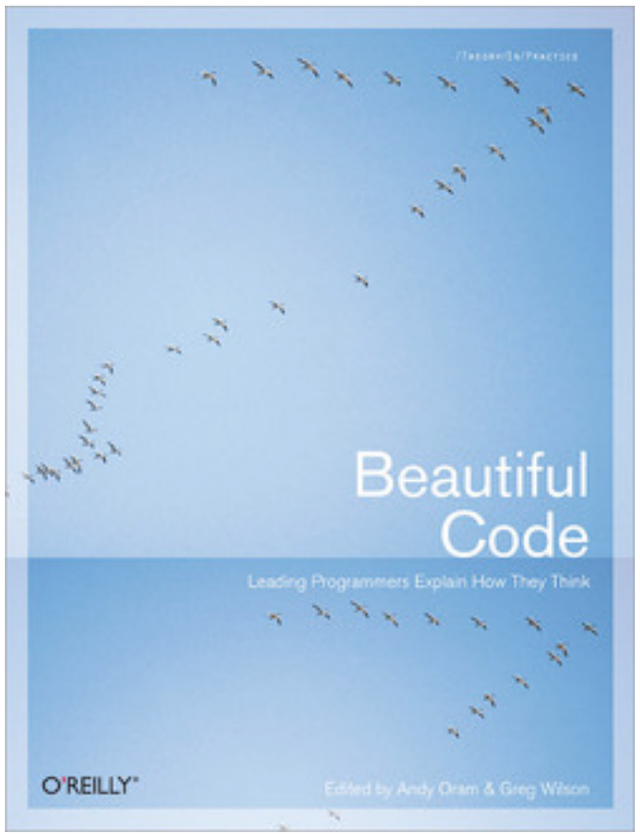
byte_tanimoto()

```
double byte_tanimoto(
    int num_words,
    const unsigned char *fpA,
    const unsigned char *fpB) {

  int num_in_intersection = 0;
  int num_in_union = 0;

  for (int i=0; i<num_words; i++) {
    num_in_intersection += popcount(fpA[i] & fpB[i]);
    num_in_union += popcount(fpA[i] | fpB[i]);
  }

  if (num_in_union == 0) {
    return 0.0;
  }
  return num_in_intersection / (double) num_in_union;
}
```

"popcount"?

The "population count" is the number of 1 bits in a word.

```
int lookup_table = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, ...};
int popcount(unsigned char c) {
  return lookup_table[c];
}
```



Many algorithms for efficient popcount calculation, going back to the 1950s.

Hank Warren
"The Quest for an Accelerated Population Count" in "Beautiful Code"

POPCNT instruction

Modern computers have a popcount CPU instruction:

- **"POPCNT" on Intel and AMD**
- **"CNT" for NEON for ARM**

These instructions work on a single word, usually 64 bits.
Nearly all fingerprints are a multiple of 64 bits.
(For MACCS, use 192 = (3 * 64) bits and pad with zeros.

Several ways you might get your compiler to use it:

- **inline-assembly**
- **C compiler intrinsic:**
  - **gcc & clang: __builtin_popcountll()**
  - **MSVC: _mm_popcnt_u64()**
- **C++20's std::popcount()**
- **Rust and Julia have count_ones()**
- **WASM has i32.popcnt and i64.popcnt**

Loop Unrolling

If all fingerprints are 166-bit MACCS keys stored in 192 bits then here's a version with no loop.

```
#include <stdint.h>

int byte_tanimoto(
    int num_words,
    const uint64_t *fpA,
    const uint64_t *fpB) {

  int num_in_intersection = (
       __builtin_popcountll(fpA[0] & fpB[0]) +
       __builtin_popcountll(fpA[1] & fpB[1]) +
       __builtin_popcountll(fpA[2] & fpB[2]));

  int num_in_union = (
       __builtin_popcountll(fpA[0] | fpB[0]) +
       __builtin_popcountll(fpA[1] | fpB[1]) +
       __builtin_popcountll(fpA[2] | fpB[2]));

  if (num_in_union == 0) {
    return 0.0;
  }
  return num_in_intersection / (double) num_in_union;
}
```

Precompute each fingerprint popcount

Take a look again at the Tanimoto calculation:

$\text{Tanimoto}(FP_A, FP_B) = c / (A + B - c)$

A and B are constant and can be precomputed for each fingerprint.
Only need to calculate the *intersection popcount* for each pair.
(Some tools compute A, B and c each time!)

Replaces many popcount calculations with simple arithmetic.

Could store the count with each fingerprint.
If order doesn't matter, then group by popcount.

Group by popcount

```
from chemfp.bitops import byte_popcount, byte_intersect_popcount

# Group target fingerprints by popcount (0, 1, 2, ... 166)
bins = [[] for i in range(167)]

for line in open("maccs.fps"):
    # skip header
    if line[:1] == "#":
        continue
    # decode the fingerprint
    hex_value, id = line.split()
    fp = bytes.fromhex(hex_value)
    bins[byte_popcount(fp)].append((id, fp))

# MACCS fingerprint for caffeine
query = bytes.fromhex("000000003000000001d414d91323915380f138ea1f")
A = byte_popcount(query)

# All fingerprints in the same bin have the same popcount B
for B, fp_data in enumerate(bins):
    AB = A + B
    for id, fp in fp_data:
        c = byte_intersect_popcount(query, fp)
        score = c / (AB - c)
        if score >= 0.95:
            print(id, score)
```

Inline functions

COMPILER EXPLORER

https://gcc.godbolt.org/z/1EfKrocjE

Not needed if inlined.

```
1  #include <stdint.h>
2  __attribute__((noinline))
3  int byte_intersect_tanimoto(
4      int num_words,
5      const uint64_t *fpA,
6      const uint64_t *fpB) {
7
8    return (
9        __builtin_popcountll(fpA[0] & fpB[0]) +
10       __builtin_popcountll(fpA[1] & fpB[1]) +
11       __builtin_popcountll(fpA[2] & fpB[2])
12   );
13 }
14
15 int f(const uint64_t *fpA,
16   const uint64_t *fpB) {
17       int c = byte_intersect_tanimoto(3, fpA, fpB);
18       return c * c;
19   }
20
```

```
1  byte_intersect_tanimoto:
2          mov     rdi, rdx
3          mov     rcx, QWORD PTR [rsi]
4          mov     rax, QWORD PTR [rsi+8]
5          and     rcx, QWORD PTR [rdx]
6          and     rax, QWORD PTR [rdx+8]
7          popcnt  rcx, rcx
8          mov     rdx, QWORD PTR [rsi+16]
9          popcnt  rax, rax
10         add     eax, ecx
11         and     rdx, QWORD PTR [rdi+16]
12         popcnt  rdx, rdx
13         add     eax, edx
14         ret
15 f:
16         mov     rdx, rsi
17         mov     rsi, rdi
18         mov     edi, 3
19         call    byte_intersect_tanimoto
20         imul    eax, eax
21         ret
```

Avoid Division and Floats

The main loop from earlier:

```
for id, fp in fp_data:
    c = byte_intersect_popcount(query, fp)
    score = c / (AB - c)
    if score >= 0.95:
        print(id, score)
```

Division is expensive. But not always needed.

Floats are also expensive - prefer integers.

Use a rejection test when expecting few hits.

```
for id, fp in fp_data:
    c = byte_intersect_popcount(query, fp)
    if (c * 20 >= (AB - c) * 19):
        score = c / (AB - c)
        print(id, score)
```

Every float can be turned into a rational number.
Even better, replaced by a rational with a small denominator.

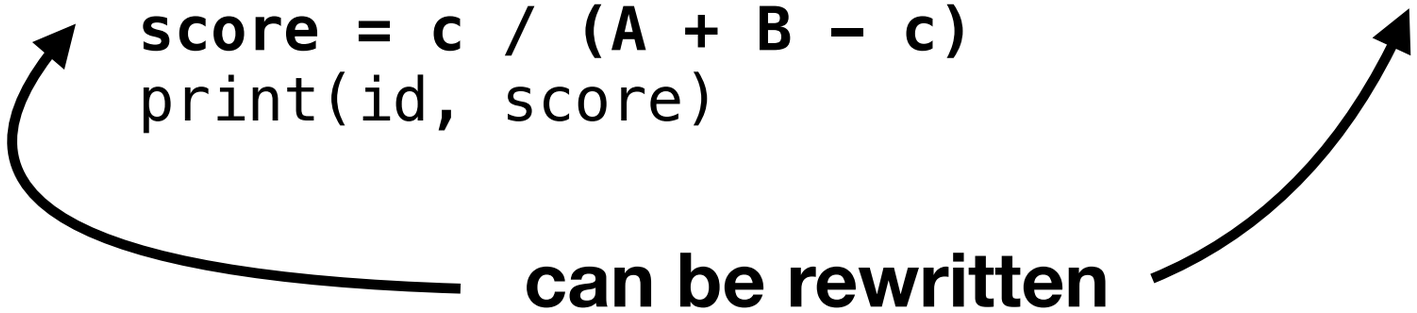
Alternatives

- **The possible values for "c" and "A+B" is small:**
  - **Store them all in a lookup table, or**
  - **Compute the lookup table only for "A+B"**

- **Store the numerator and denominator as small integers**
  - **At the end, convert (in parallel) to float32 or float64**

Avoid arithmetic

```
if (c * 20 >= (A + B - c) * 19):
    score = c / (A + B - c)
    print(id, score)
```




```
if (c >= min_c):
    score = c / (A + B - c)
    print(id, score)
```

where "min_c" is constant for a given threshold, A, and B:

$$c \overset{?}{\geq} (threshold * (A + B)) / (1 + threshold)$$

Not so simple due to IEEE 754 rounding!

```
threshold = 0.95;
AB = A + B;
min_c = (int)(threshold * AB / (1 + threshold));

/* If it is too small, round up */
if (((double) min_c) / ((double)(AB - min_c)) < threshold) {
  min_c++;
}
```

Fingerprint "arena"

Store all fingerprints sequentially in a single byte array.
Align the fingerprints to the appropriate machine word size.

index	0	1	2	3	4	5	6	...
bits	00000001	10000000	00010100	00010001	00100011	10010001	11010100	...

If ordered, also store the first position of each popcount.

popcount	0	1	2	3	4	...
start position	0	0	2	4	6	...

Fingerprints with 3 bits set start at index 4 and go up to but do not include index 6.

POPCNT isn't so fast!

Intel CPUs have only a single POPCNT execution port.
Limited to one POPCNT per clock cycle.

2048 bits / 64 bits = 32 clock cycles
@ 2 GHz = 16 nanoseconds / fingerprint.

Wojciech Muła, Nathan Kurz, Daniel Lemire
"Faster Population Counts Using AVX2 Instructions"

AVX2 operates on 256-bits at a time.
Requires more instructions, but able to use more ports.
Overall about 30% faster!

And compilers will do this for you!

Memory I/O limits

CPUs are fast. RAM latency is slow.

It takes about 100 ns to fetch something from RAM.
(That's 5-10 Tanimoto evaluations!)

The memory manager will cache pages,
and try to predict what you want.
Can also provide explicit prefetch requests.

AVX2-based similarity search is one of the rare times
I've found prefetch made a difference.

Effect of popcount implementation on Tanimoto search performance.

Popcount method	Performance relative to 8-bit lookup table			
	166 bits	881 bits	1024 bits	2048 bits
8-bit lookup table	1×	1×	1×	1×
16-bit lookup table	2.0	2.8	2.9	2.4
Gillies-Miller [18]	1.6	2.9	3.1	3.4
Lauradoux [19]		3.1	3.3	3.7
SSSE3 [15]			5.4	6.1
POPCNT (8 bytes/loop)				
Dispatch	3.6	6.0	6.3	6.4
Inline	4.9	6.6	6.9	6.6
POPCNT (fully unrolled)				
Dispatch	5.3	7.9	8.2	7.8
Inline	6.7	8.2	8.4	8.0
AVX2 [20] (fully unrolled)				
Dispatch			8.6	9.2
Dispatch, prefetch			8.7	9.3
Inline			9.8	9.9
Inline, prefetch			11.0	10.6

Andrew Dalke, "The chemfp project" (2019)

BitBound

No need to brute-force search all fingerprints.
Swamidass and Baldi pointed out a simple pruning method.

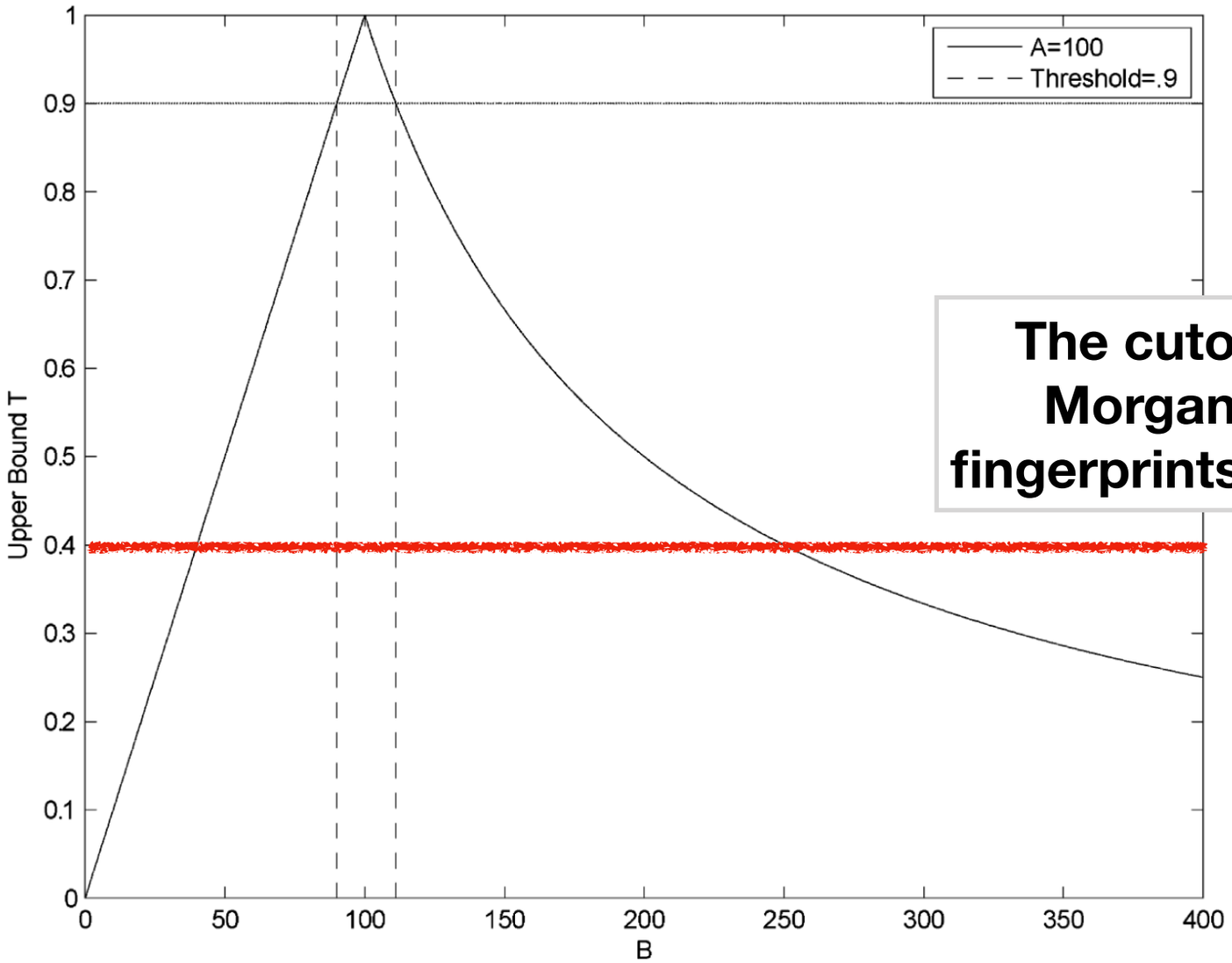
If the query threshold is T and the query has A bits set then

$$A*T \leq B \leq A/T$$

We've already sorted by popcount.
Use the index to ignore fingerprints which cannot meet the threshold.

S. Joshua Swamidass and Pierre Baldi, "Bounds and Algorithms for Fast Exact Searches of Chemical Fingerprints in Linear and Sublinear Time" (2007)

Prune search space

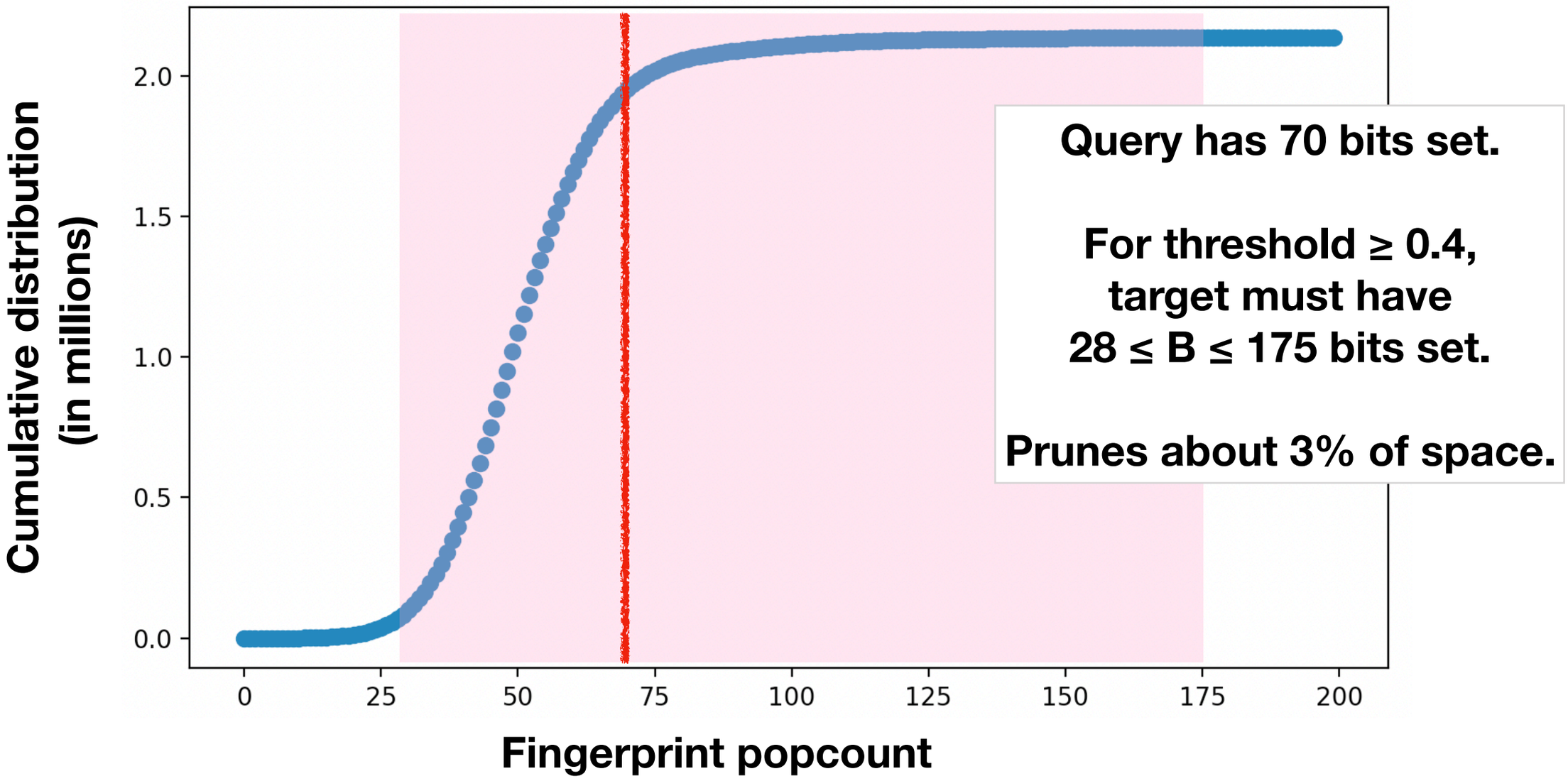


The cutoff for RDKit's Morgan (ECFP-like) fingerprints is typically 0.4.

Figure 3. Pruning the search space. The binary fingerprint of the query molecule satisfies $A = 100$. As a function of B, the Tanimoto similarity measure $S(\vec{A},\vec{B})$ is upper-bounded by the curve $T(A,B)$. If the similarity threshold is set at 0.9, only molecules with B in a very small interval around 100 need to be searched. All other molecules have similarity scores that are below the threshold.

S. Joshua Swamidass and Pierre Baldi, "Bounds and Algorithms for Fast Exact Searches of Chemical Fingerprints in Linear and Sublinear Time" (2007)

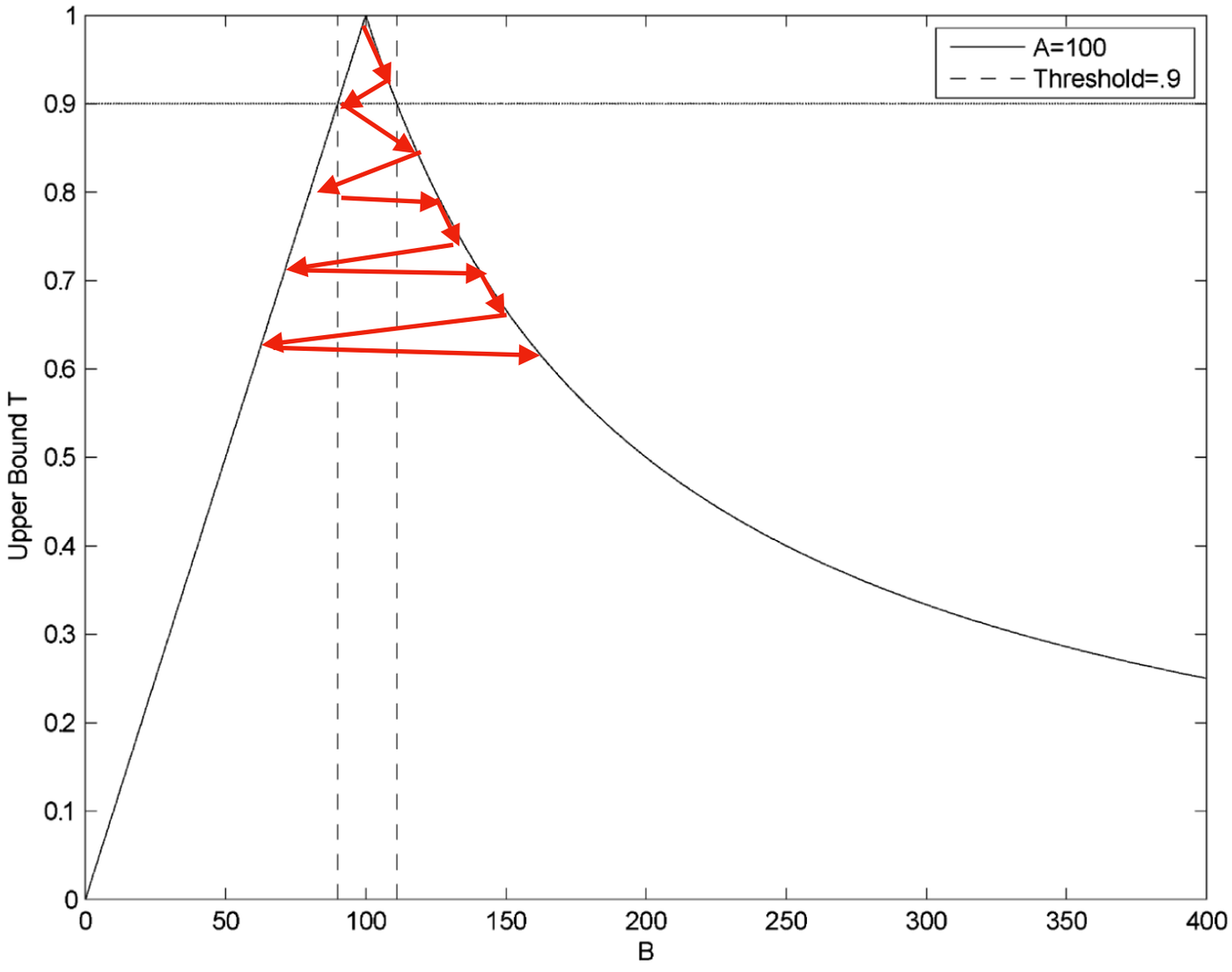
Fingerprint popcount distribution in ChEMBL 30



k-NN search ordering

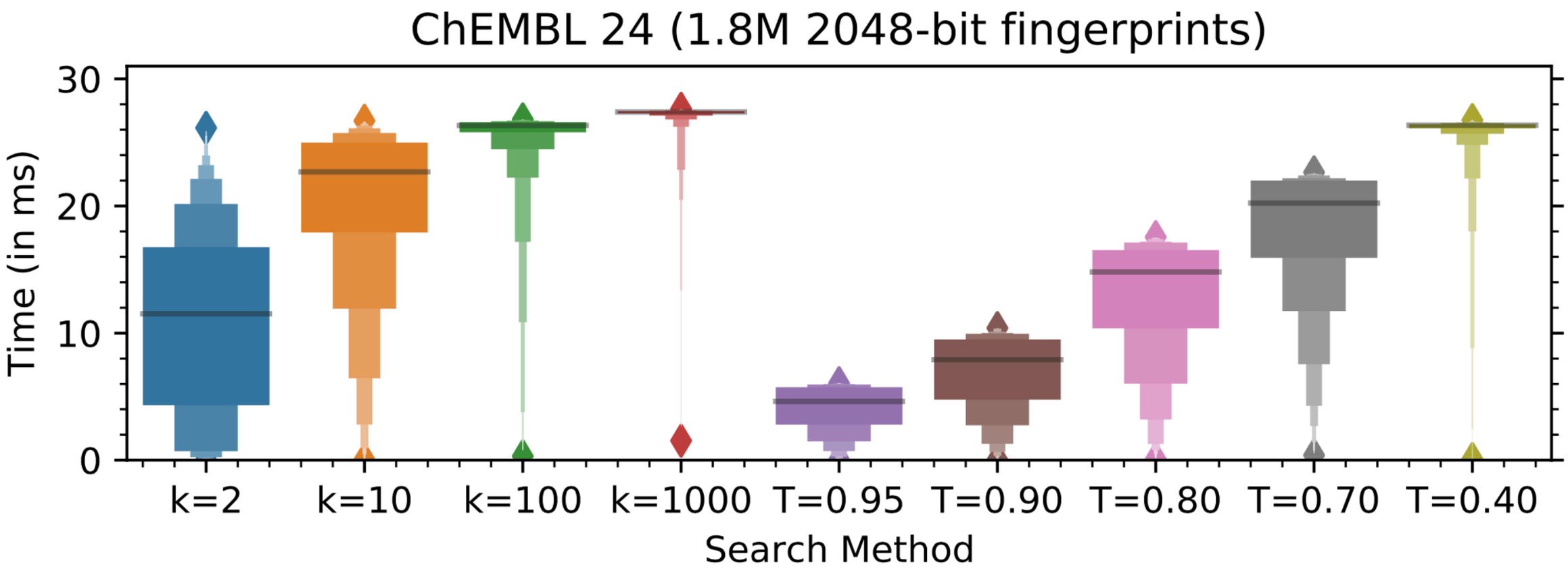
Preferentially search the popcount bins *ordered by potential best score*.

Update the bounds after each new neighbor.



Highly similar fingerprints are likely to have a similar number of bits set.
(The converse isn't true.)

BitBound Effectiveness

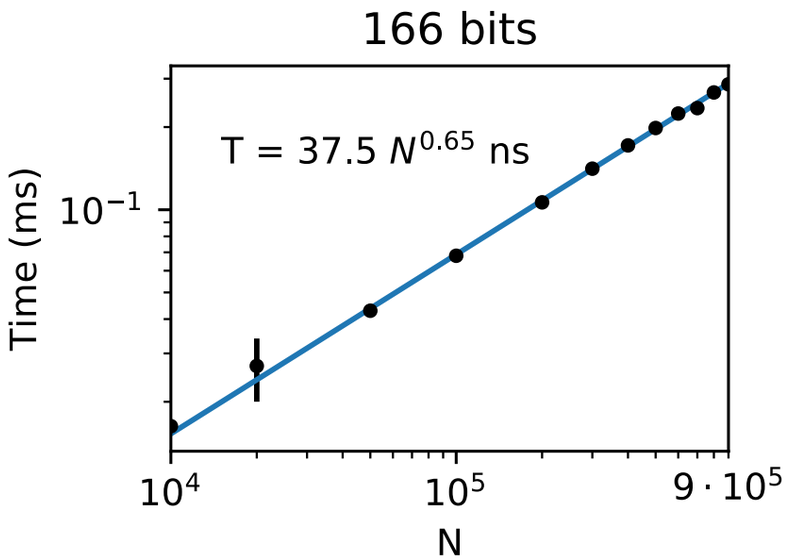


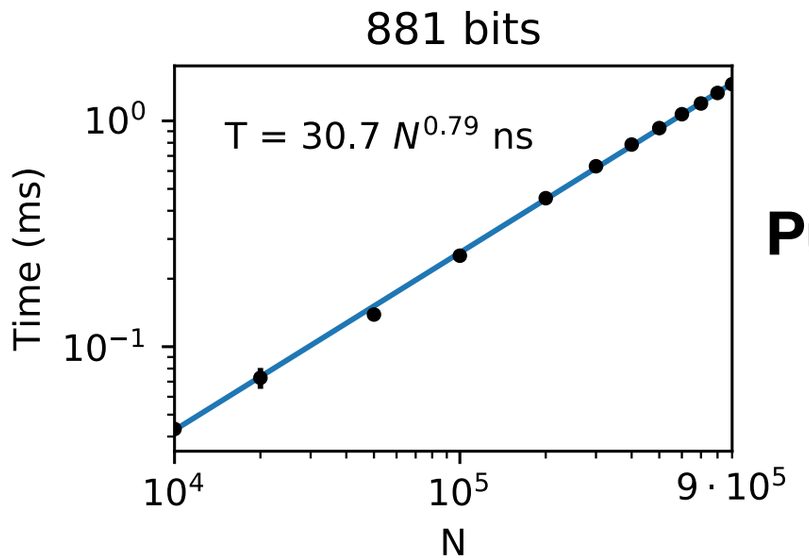
Single query search times for chemfp 3.3. Boxen plots for k=2, 10, 100, and 1000 nearest-neighbor and threshold=0.95, 0.80, 0.70, and 0.40 searches of ChEMBL 24. Each search samples 1000 fingerprints to use as queries so each query is always found in the result. Python's garbage collector was disabled for each timing as it adds a roughly 25 ms delay about every 1000 timings.

Andrew Dalke, "The chemfp project" (2019)

NN search is sublinear(!)

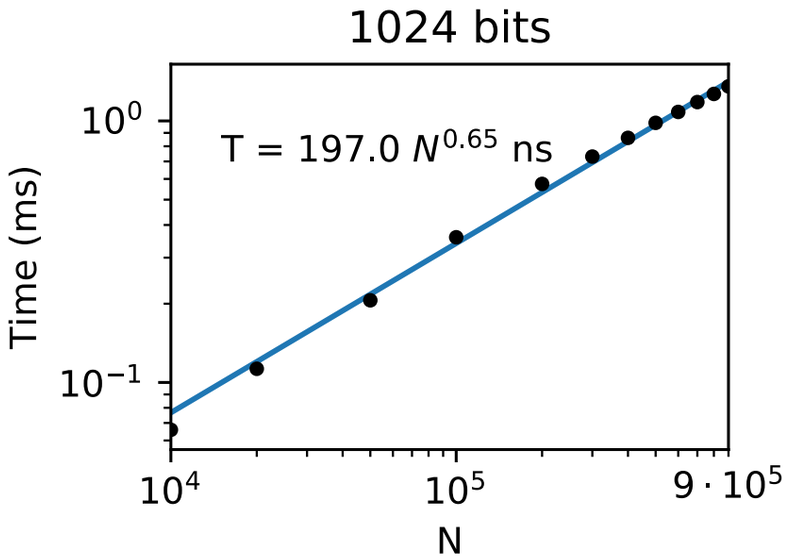
MACSS

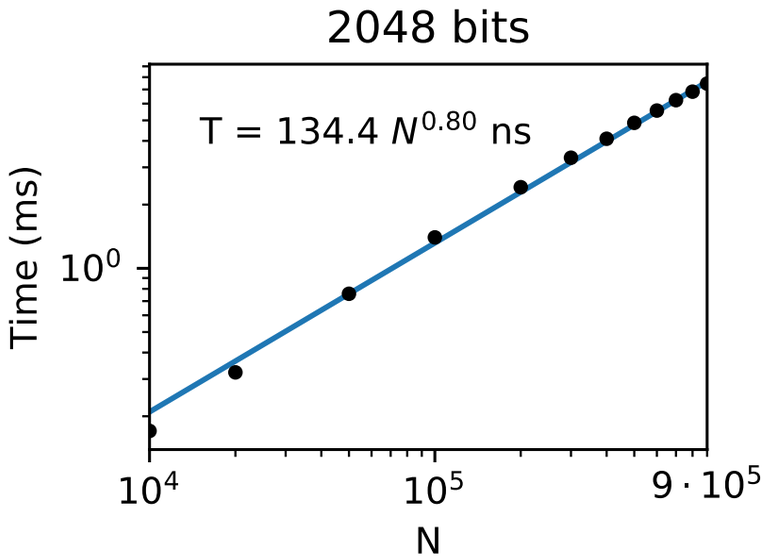




PubChem

Open Babel FP2





RDKit Morgan

Time for k=1 nearest-neighbor search, scaling in the number of targets.

Andrew Dalke, "The chemfp project" (2019)

Multicore

I use OpenMP to parallelize NxN and NxM searches.
One query, one OpenMP thread.

Memory I/O issues are weird!

With 4 threads, if the queries are in popcount order than there's 3x performance; in random order, only 1.6x.
Likely due to improved cache locality.

Using two threads for a single query was slower.
My best guess is the memory bandwidth was too low.
(Need to revisit on more modern hardware.)

Memory bandwidth

One DDR4-3200 memory channel is ~25 GB/s.
That's 105M 2048-bit fingerprints / sec.
About 10 ns / fingerprint.

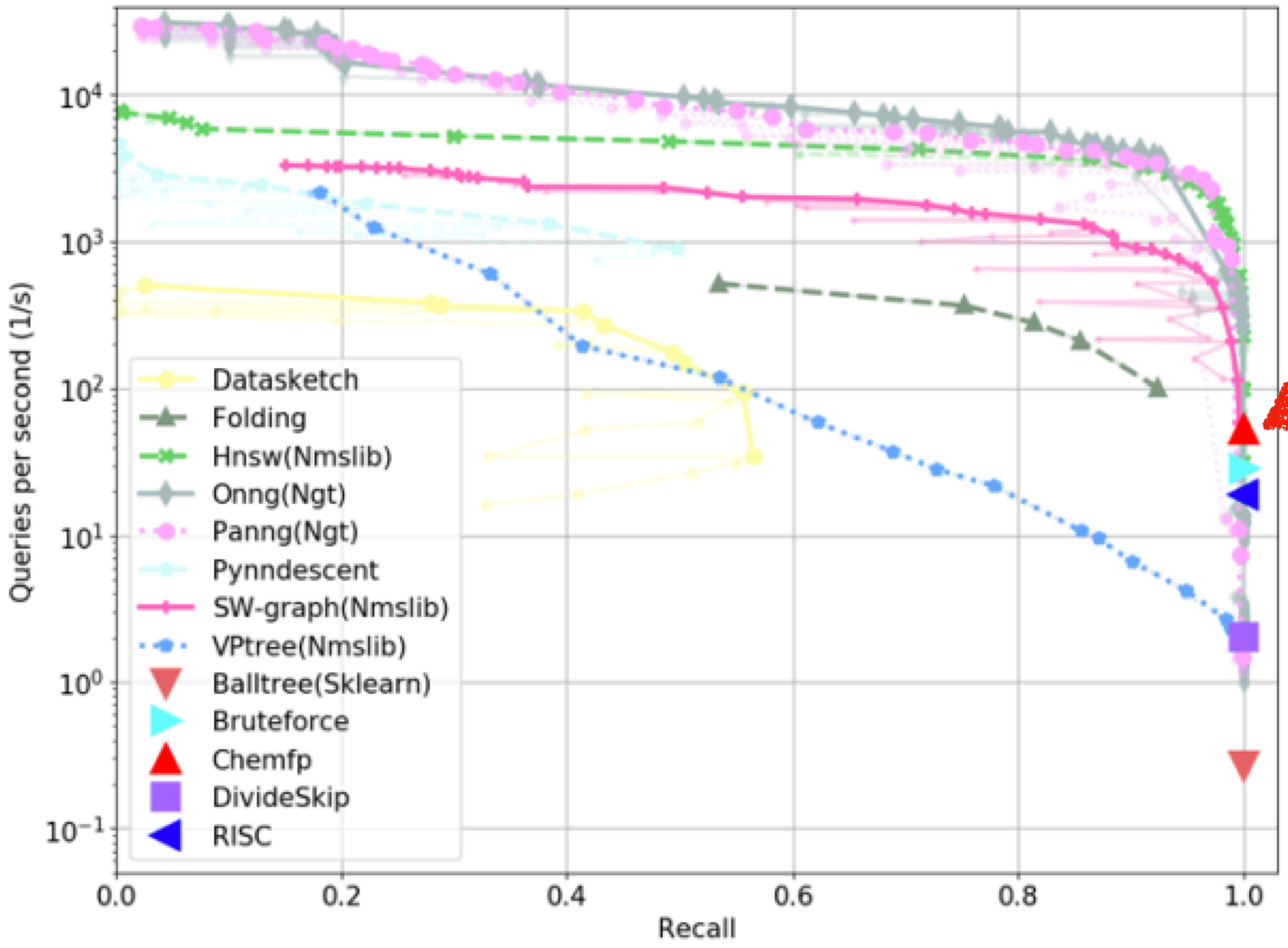
Which is about the CPU time needed to evaluate a Tanimoto.

We should expect 1 CPU to consume one memory channel.

More sophisticated use of caching in:

Imran S. Haque, Vijay S. Pande, and W. Patrick Walters
"Anatomy of High-Performance 2D Similarity Calculations" (2011)

Approximate Search



(a) QPS and RECALL for the ChEMBL dataset

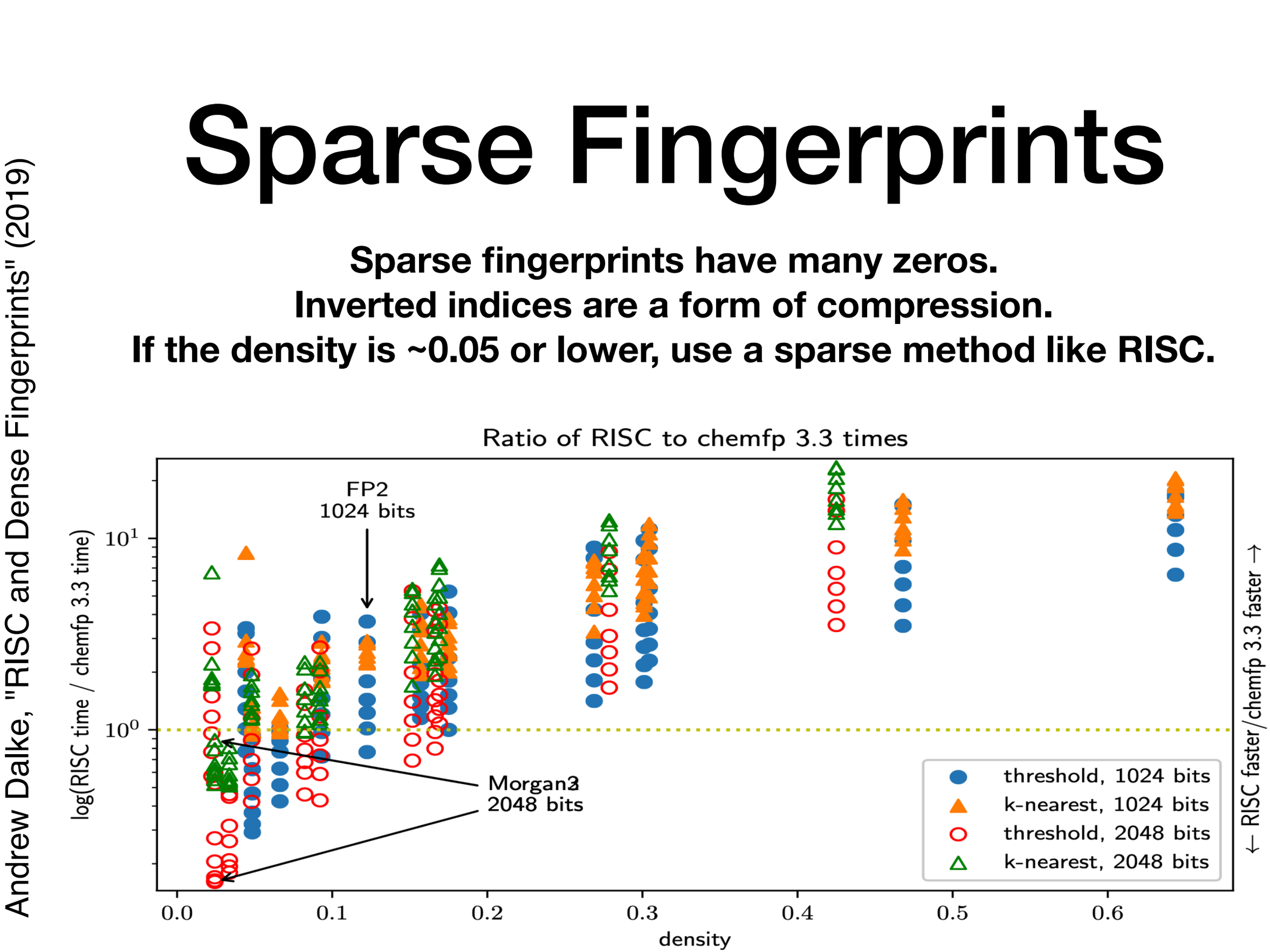
chemfp 1.6 was the fastest for perfect recall

chemfp 1.6 does not use AVX2.

Chun Jiang Zhu, Minghu Song, Qinqing Liu, Chloé Becquey, and Jinbo Bi
"A benchmark on indexing algorithms for accelerating molecular similarity search" (2020)

Sparse Fingerprints

Sparse fingerprints have many zeros.
Inverted indices are a form of compression.
If the density is ~0.05 or lower, use a sparse method like RISC.



Comparisons made using a range of fingerprint types and query types.

Diversity Selection

Diversity means different things to different people.

MaxMin

Which compounds are most dissimilar from any other compounds?
- **and dissimilar from compounds in a reference data set**
- **while still being reasonable**

Sphere Exclusion

Pick compounds at random:
- **but exclude compounds similar to previous picks**
- **or from a reference set**
- **while still being reasonable**

Many other approaches (cluster w/ NxN similarity matrix, simulated annealing, ...)

MaxMin

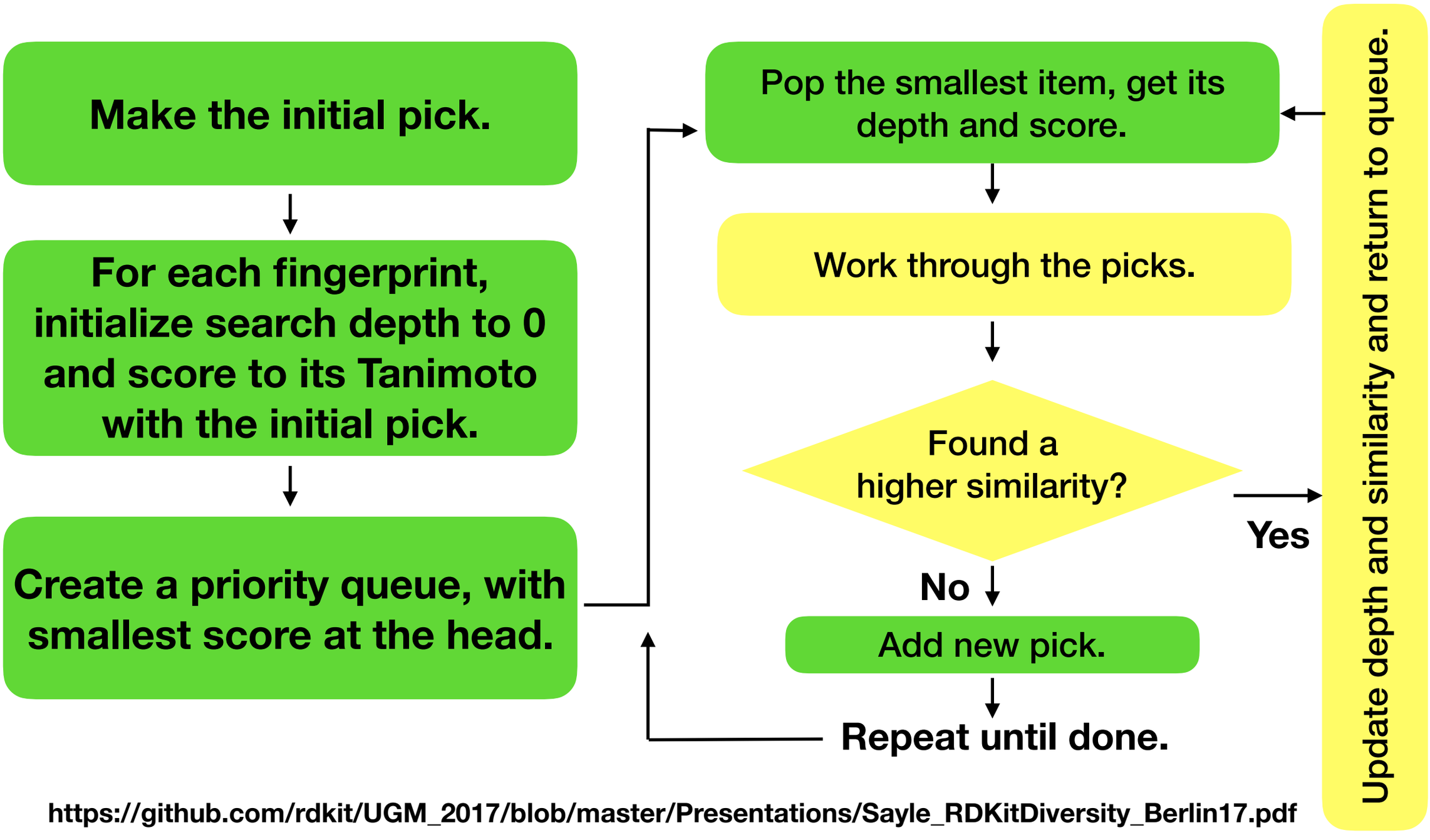
This is an approximate, iterative solution.

1. **Pick one or more initial fingerprints.**
2. **From the remaining fingerprints,**
   find the most dissimilar.
3. **Add it to the picks.**
4. **Repeat until done.**

I'll assume that "dissimilar" means "1-Tanimoto".

Sayle/RDKit algorithm

Simplified and not accurate.
(Uses a round-robin table instead of a priority queue.)



https://github.com/rdkit/UGM_2017/blob/master/Presentations/Sayle_RDKitDiversity_Berlin17.pdf

"Obvious" Improvements

- **Hard-code the choice of Tanimoto similarity.**
- **Store the fingerprint's popcount with the fingerprint item.**
- **Use a faster intersection popcount method.**
- **Store 16-bit ratios of c/d rather than 64-bit doubles.**

Result is about 2-3x faster.

Need to be fair!
RDKit's MaxMin is also faster if the fingerprint are in popcount order.

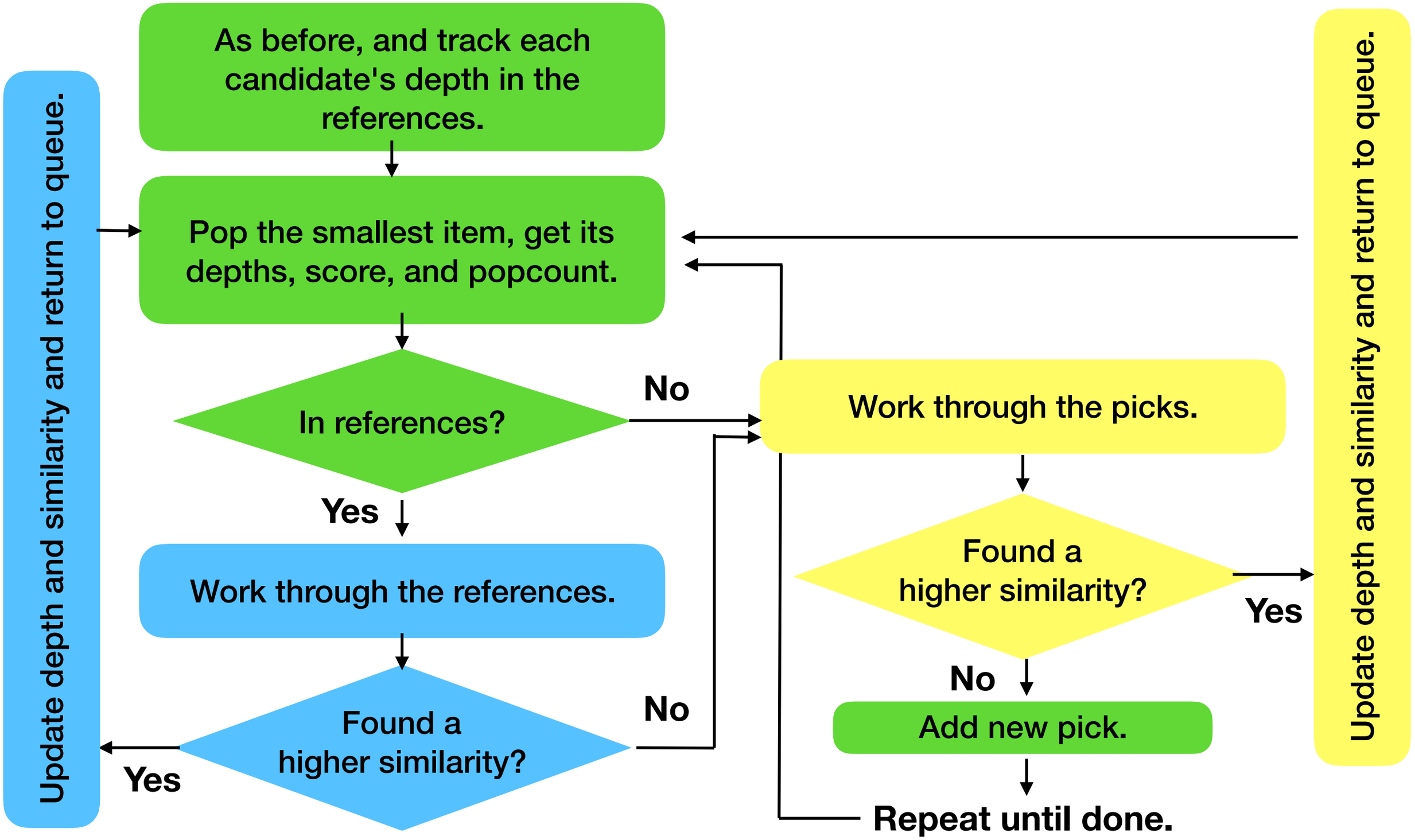
Candidates vs. References

"Pick compounds from eMolecules to improve ChEMBL diversity"

Can treat this as one large fingerprint set combining the references (ChEMBL) and the candidates (eMolecules), where the references are the initial picks.

Alternatively, interpret this as a nearest-neighbor problem.

Two-component MaxMin



BitBound MaxMin

**When searching the reference arena,
don't simply go from start to end.**

Unlike the picks, the reference arena is constant.

Use the BitBound NN search ordering:

- **Enriches the chance of finding a near neighbor.**
- **Improves pruning.**
- **More complicated to implement.**

**"Pick 100 compounds from eMolecules
to improve ChEMBL diversity"**

chemfp - 86 seconds
RDKit - 2774 seconds

~30x

Initial MaxMin pick

Choose at random? In the middle? First? Last?

Pick the one with the lowest maximum similarity.

Set up a scoreboard for each fingerprint:

- **"maximum seen score" and "at max"**

Do 50 random sweeps:

- **Randomly pick a fingerprint (these will be "at max")**
- **Find its similarity to the rest & update scoreboard**

Set up priority queue:

- **If smallest is "at max", it's the first pick.**
- **Otherwise, sweep using this fingerprint**

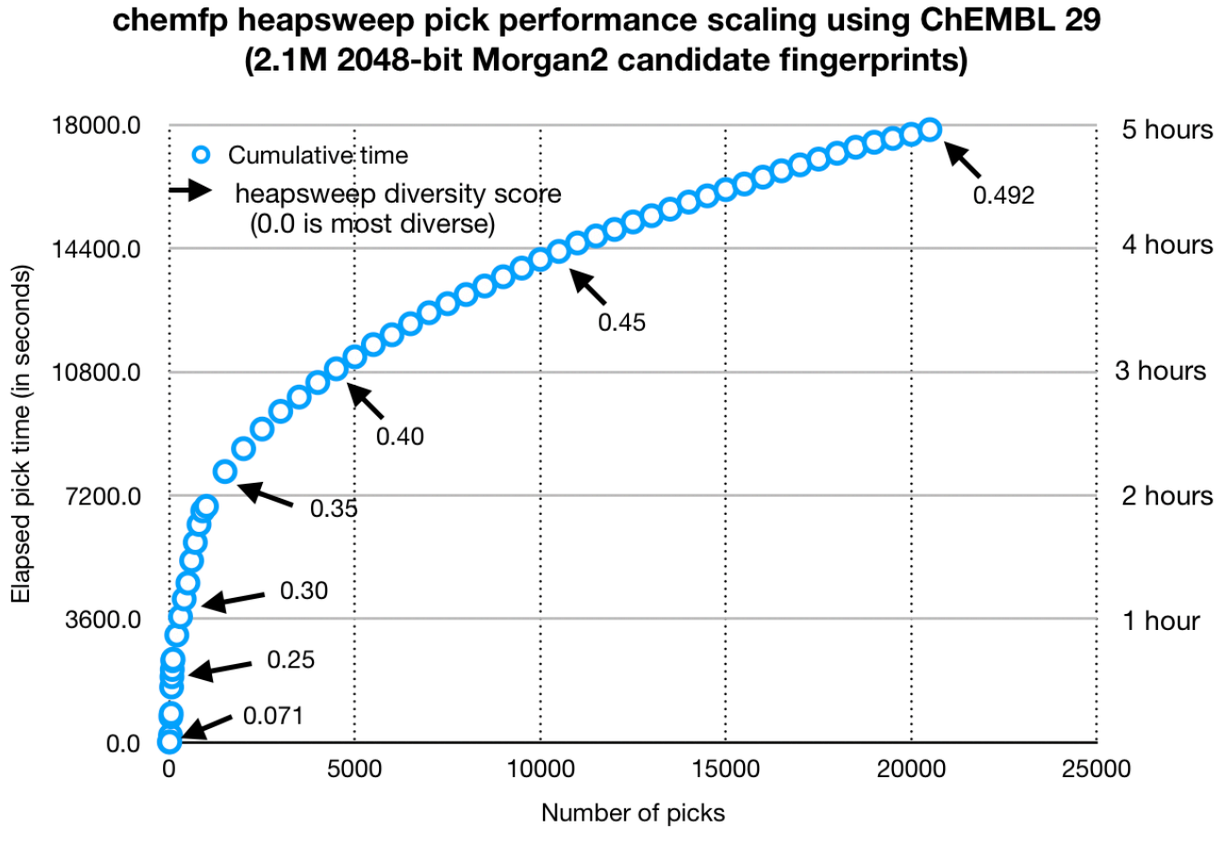
Observation: The first fingerprint (when ordered by popcount) is often also the most dissimilar to the rest.

HeapSweep

Can continue the process.

I leave the fingerprint in the heap (no compaction).
Full HeapSweep does NxN comparisons.

Takes about 50x longer (!) than MaxMin but gives an exact solution.



Sphere Exclusion

1. Pick a fingerprint.
2. Remove all fingerprints which are sufficiently similar.
   - radius = 1 - similarity threshold
3. Repeat until done.

Use BitBound and work on the popcount bin level.

1. Assume Tanimoto similarity
2. Bin based on popcount
3. Pick a fingerprint at random
4. Use BitBound to limit the bin search space
5. Remove similar fingerprints and compact each bin
6. If not done, go to 3.

Result is about 3-4x the performance of RDKit.

DISE

"Directed Sphere Exclusion", Gobbi and Lee (2002)

Choose the fingerprint with the lowest associated value.

The DISE paper assigns a rank based on"descending similarities with three reference molecules."

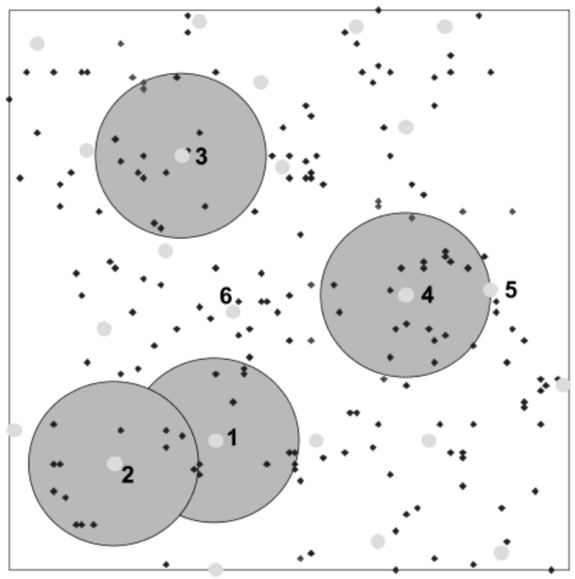


Figure 2. Sphere Exclusion – Simulated Selection. Selected points are given as (●). The first four spheres are shown as well as the selection order of the first six points (numbers).

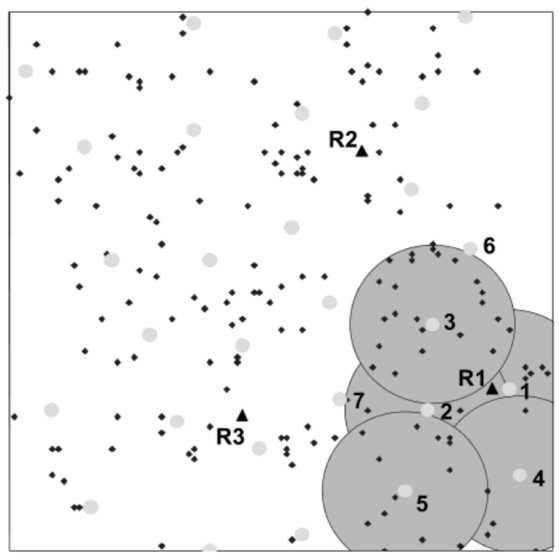


Figure 4. DISE – Simulated Selection. The three reference points (▲: R1, R2, R3) were added for display, but not as part of the candidate set. Selected points are given as (●). The first five spheres are shown as well as the selection order of the first seven points.

Biases selection towards the primary reference.

Roger Sayle points out the ranking can be anything. Cost might be an interesting one.

BitBound DISE

- **Primary sort by popcount**
- **Secondary sort by the rank value**
  - **Assume there are few duplicate ranks**
  - **Then don't need an index or secondary bins**

- **During the fingerprint pick step:**
  - **only count initial fingerprints with the lowest rank**
  - **sample from them**
  - **O(number of fingerprint bits)**

If there are many duplicates, will need a secondary index.

Remove Junk

Make sure you filter out inappropriate structures *first*!

[82Rb] CHEMBL4297424
CN1CCC(N(C)C(=O)/C(C#N)=C/c2ccccc2)CC1 CHEMBL1740398
[K+].[OH-] CHEMBL2103983
BrCCCCCBr CHEMBL3182198
FC(F)(F)C(F)(F)C(F)(F)C(F)(F)C(F)(F)F CHEMBL1899801
[Na+].[O-]Cl CHEMBL1334078
[Li+] CHEMBL1234004
Cl.Cl.NN CHEMBL542171
[123IH] CHEMBL1909276
[S] CHEMBL2105487
[C-]#[O+] CHEMBL1231840
I CHEMBL1233550
[Rb] CHEMBL1201326
[C] CHEMBL2106049
[Cs] CHEMBL4302507
[131Cs] CHEMBL4297365
[Kr] CHEMBL1233877
[85SrH2] CHEMBL2106034
[Se-2] CHEMBL4597517
[223Ra+2].[Cl-].[Cl-] CHEMBL2107816
[Xe] CHEMBL1236802

First 21 structures from a MaxMin search of ChEMBL 30

Suggests the need for new functionality.
"Has at least 1 neighbor with similarity T."

Use to select fingerprints similar enough to your desired chemical space.

Should be faster than "choose/count all neighbors with similarity T"

Scaling to large data sets?

MaxMin uses a lot of effectively random-access lookups.
~10x slower than the Tanimoto calculation.

At some point computing the full similarity matrix is faster.
(Estimating >~100K picks from the 2M in ChEMBL).

What about using multiple passes?

- **Start with 1B fingerprints.**
- **Split into 100 sets of 10M fingerprints.**
- **In parallel, pick 10K fingerprints from each set.**
- **Merge the pick results into a set of 1M fps.**
- **Pick 100K fingerprints from those.**

Thanks!

- **Roger Sayle**
- **The RDKit developers**
- **Jakub Gunera**

chemfp

- **Command-line tools and Python API**
- **Interfaces to RDKit, OpenEye, Open Babel, and CDK**
- **Designed to be embedded in existing tools/servers**
- **Source code license**
  - **macOS and Linux**
  - **no time limit on use (doesn't require ongoing support)**
- **Binary license for eval and academics**
  - **Linux only**
- **No cost to academics and independent researchers**

http://chemfp.com/