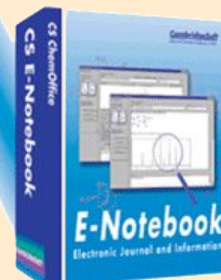
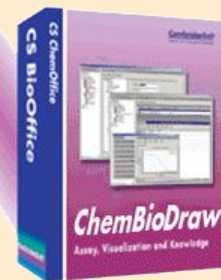




CambridgeSoft®
Life Science Enterprise Solutions
Naturwissenschaftliche Unternehmenslösungen
Solutions globales pour les sciences de la vie
ライフサイエンス・エンタープライズ・ソリューション



Its All In The Bits

Recent Advances in Information Boosting,
Database Indexing and Molecular Similarity
Searching

Harold Helson and Andrew Smellie

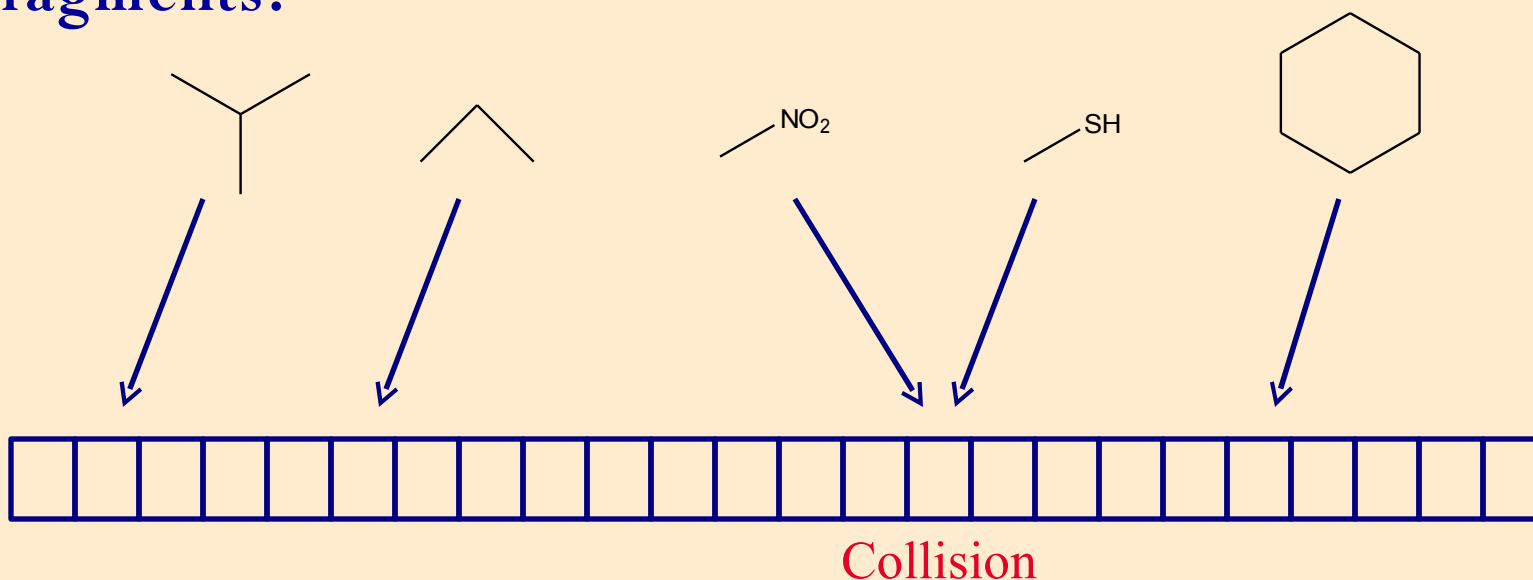
CambridgeSoft, Inc.

8th ICCS conference, Noordwijkerhout, June '08

CambridgeSoft

Introduction: Molecules in Databases

- Are often represented as fingerprints (bitstrings)
- Each bit (key) codes for the presence of functional groups or fragments
- Fingerprints are generally short (<2048 bits) for compactness
- Collisions occur because every key codes for an infinite number of fragments.



Anatomy of our Regular Fingerprint

Width=370 (ded=53 + Frag= 317; MaxFrag= 6)

===== Dedicated =====

10 <- R6_x1

43 <- ELM_TYP: Other Common

===== Fragment (2D) =====

Fragment	64-bit code	Key
C~C	-> 504	-> 125
C~C~N	-> 8188	-> 275
C~C~N~C	-> 131128	-> 82
C~C~N~C~C	-> 2098168	-> 226
C~C~N~C~C~C	-> 33555448	-> 331

...

Whole screen has 75 bits (20% used):

10,43,57,60,70,73,74,77-85,88-92,102,105,111,
116,118,122,125,126,129-131,134,139,184,189,
190,193-195,197,226,227,234,239,261,262,271-
277,280,281,285,305,307,310,311,317,322-324,
326,328-332,334,343,352,358

...

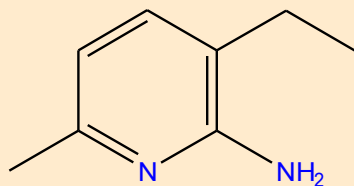
7 collisions (8.8%)

e.g. [C~C~C~N~C~N] = [C~C~C~C]

Two Sections:

- Dedicated

- Fragment



Database Operations with Fingerprints #1

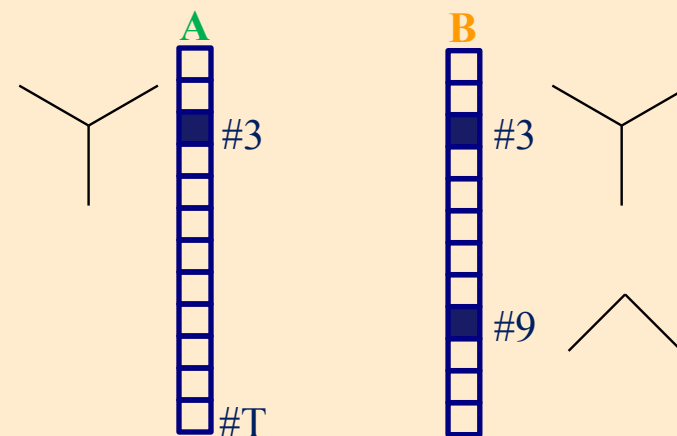
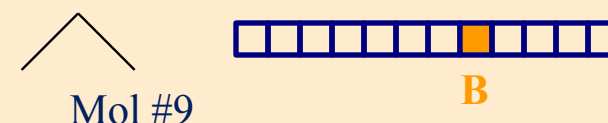
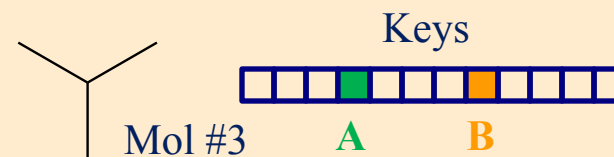
● Screening

– Regular screening:

- » Find all target molecules in the database that have the same bits on as the query
- » Naively requires all target fingerprints to be examined
=> Computation = #T x F_Width

– Inverted screening:

- » For each key there is a list of the mols containing it.
- » Screening consists of joining the lists of every key in the query.
=> Computation = #keys in Q x #T



Regular vs. Inverted Screening

$$Q = \{ 2, 5 \}$$

	K2				K5						
Mol 1	F	i	n	g	e	r	p	r	i	n	t
Mol 2	F	i	n	g	e	r	p	r	i	n	t
Mol 3	F	i	n	g	e	r	p	r	i	n	t
Mol 4	F	i	n	g	e	r	p	r	i	n	t
Mol 5	F	i	n	g	e	r	p	r	i	n	t
Mol 6	F	i	n	g	e	r	p	r	i	n	t
Mol 7	F	i	n	g	e	r	p	r	i	n	t
Mol 8	F	i	n	g	e	r	p	r	i	n	t

Regular screening. Each mol stores its fingerprint.

Fingerprint
Key 1 Key 2 Key 3 Key 4 Key 5 Key 6

M 1	M 1	M 1	M 1	M 1	M 1
M 2	M 2	M 2	M 2	M 2	M 2
M 3	M 3	M 3	M 3	M 3	M 3
M 4	M 4	M 4	M 4	M 4	M 4
M 5	M 5	M 5	M 5	M 5	M 5
M 6	M 6	M 6	M 6	M 6	M 6
M 7	M 7	M 7	M 7	M 7	M 7
M 8	M 8	M 8	M 8	M 8	M 8

Inverted screening. Each key lists the mols containing it.

Inverted Screening Requires Fewer Bit Comparisons than Regular

- The bits that must be compared for the query fingerprint { 2, 5 }



Regular screening. Each mol stores its fingerprint.

Fingerprint
Key 1 Key 2 Key 3 Key 4 Key 5 Key 6

M 1		M 1	M 1		M 1
M 2		M 2	M 2		M 2
M 3		M 3	M 3		M 3
M 4		M 4	M 4		M 4
M 5		M 5	M 5		M 5
M 6		M 6	M 6		M 6
M 7		M 7	M 7		M 7
M 8		M 8	M 8		M 8

Inverted screening. Each key lists the mols containing it.

Database Operations with Fingerprints #2

● Similarity Searching

- “Find all target molecules *M* within distance *d* of a query molecule *Q*”
- Computed in bit string space using the *Tanimoto Coefficient*

» $T(A,B)$ = *Tanimoto distance between bit strings A and B*

$$\text{» } T(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{\text{\# ON bits } A \text{ and } B \text{ have in common}}{\text{\# ON bits when } A \text{ and } B \text{ are OR'ed}}$$

- *Brute force algorithm must loop over all targets' fingerprints (as with Regular Screening)*

Database Operations with Fingerprints #2

● Similarity Searching

- “Find all target molecules M within distance d of a query molecule Q ”
- Computed in bit string space using the *Tanimoto Coefficient*

» $T(A,B)$ = *Tanimoto distance between bit strings A and B*

» $T(A,B) = I(A,B) / U(A,B)$

where:

» $I(A,B)$ = *Number of ON bits A and B have in common*

» $U(A,B)$ = *Number of ON bits when A and B are Or'ed*

- *Brute force algorithm must loop over all targets' fingerprints (as with Regular Screening)*

Part 1:

Sublime Fingerprints

Extracting the **Good Bits**

Introduction

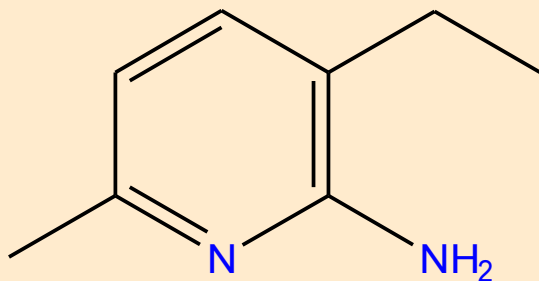
- Wider fingerprint => more discriminating
- Narrower fingerprint => faster to execute
- We will explore a way to get the best of both.
 - Generate a wide fingerprint.
 - Extract the useful bits into a narrow fingerprint.

Anatomy of our Regular Fingerprint

Width=370 (ded=53 + Frag= 317; MaxFrag= 6)

Two Sections:

- Dedicated
- Fragment



=====
Dedicated
=====

10 <- R6_x1

43 <- ELM_TYP: Other Common

=====
Fragment (2D)
=====

Fragment	64-bit code	Key
C~C	-> 504	-> 125
C~C~N	-> 8188	-> 275
C~C~N~C	-> 131128	-> 82
C~C~N~C~C	-> 2098168	-> 226
C~C~N~C~C~C	-> 33555448	-> 331

...

Whole screen has 75 bits (20% used):

10,43,57,60,70,73,74,77-85,88-92,102,105,111,
116,118,122,125,126,129-131,134,139,184,189,
190,193-195,197,226,227,234,239,261,262,271-
277,280,281,285,305,307,310,311,317,322-324,
326,328-332,334,343,352,358

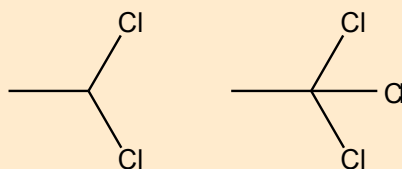
...

7 collisions (8.8%)

e.g. [C~C~C~N~C-N] = [C~C-C-C]

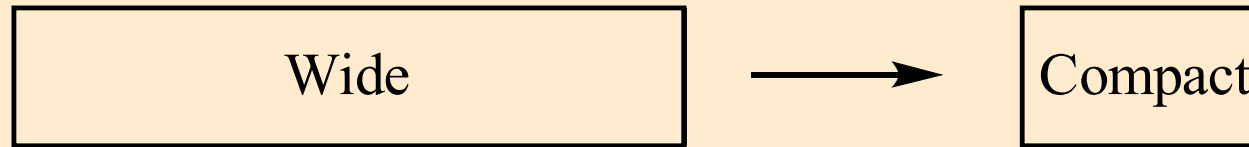
Inefficiencies in a Wide Fingerprint

- Some keys are rarely used. Some are present all the time. A key is only selective if it occurs at some intermediate frequency.
 - Can delete them with little loss in screening effectiveness.
- Some pairs of keys usually occur together.



- Can combine the keys with little loss.

Nitty Gritty



Choose a Wide Fingerprint (WF).

Wider fingerprint → fewer key collisions. Pick smallest width with acceptable rate.

2. Pick a representative set of structures, the more the better.
3. Generate fingerprints at various widths and hash functions.
4. Pick the best.

Generate Wide Fingerprints at Various Widths

28 widths and 3 hash functions tried on 38230 mols.

Average number of collisions per structure, i.e. the number of distinct FE codes that hash to the same key.

# bits	Hash#0	Hash#6	Hash#7	StatExpctd
123:	33.09	33.89	39.88	56.43
141:	31.54	38.59	45.63	44.89
143:	28.81	29.15	35.45	43.89
166:	23.82	25.37	25.53	34.96
193:	21.69	20.96	39.29	28.22
213:	23.74	52.08	51.41	24.69
224:	18.71	18.94	17.27	23.10
226:	17.00	17.61	16.92	22.83
260:	14.90	15.52	15.67	19.08
302:	16.85	12.80	12.10	15.87
351:	13.53	10.40	11.67	13.26
370:	14.12	13.01	12.66	12.46
407:	10.67	8.72	9.48	11.16
408:	11.01	14.70	10.13	11.13
474:	10.16	7.64	7.40	9.38
550:	8.45	6.72	8.53	7.95
639:	8.80	4.73	5.27	6.74
742:	5.82	3.47	4.30	5.73
862:	5.35	4.56	5.07	4.88
898:	8.97	4.73	3.49	4.68
1001:	5.95	5.08	8.99	4.17
1163:	6.57	4.20	6.27	3.56
1212:	5.16	3.30	2.85	3.41
1351:	4.21	3.46	2.21	3.04
1570:	4.92	2.06	2.36	2.60
1823:	3.63	1.61	2.03	2.23
2118:	3.98	2.65	1.33	1.91
2254:	5.30	1.27	1.28	1.79
2435:	4.29	1.02	1.50	1.66
2441:	4.53	0.92	4.48	1.65
2460:	4.52	0.97	1.68	1.64
2483:	4.85	1.53	1.30	1.63
2857:	3.65	0.54	2.19	1.41
3306:	3.17	0.75	0.51	1.21
3319:	3.40	0.75	2.61	1.21
3855:	3.87	0.89	0.83	1.04
4024:	3.25	0.61	0.88	0.99
4478:	3.47	0.41	0.49	0.89
4505:	3.39	0.34	3.08	0.89
4964:	3.24	0.35	0.60	0.80
5202:	3.51	0.73	0.63	0.77
5768:	3.30	0.36	0.38	0.69
6043:	3.13	0.82	0.31	0.66
7019:	3.14	0.48	0.45	0.57
8153:	3.01	0.20	1.04	0.49
9003:	3.06	0.27	0.40	0.44
9471:	3.03	0.40	0.11*	0.42
9509:	3.54	0.61	1.06	0.42
11001:	3.68	0.29	0.94	0.36

*Best value

49 widths and 3 hash functions tried on 443068 mols.

Table shows average # collisions per structure, i.e. the # distinct fragment codes that hash to the same key.

# bits	Hash#0	Hash#6	Hash#7	Stat.Exp.
2857:	3.65	0.54	2.19	1.41
3306:	3.17	0.75	0.51	1.21
3319:	3.40	0.75	2.61	1.21

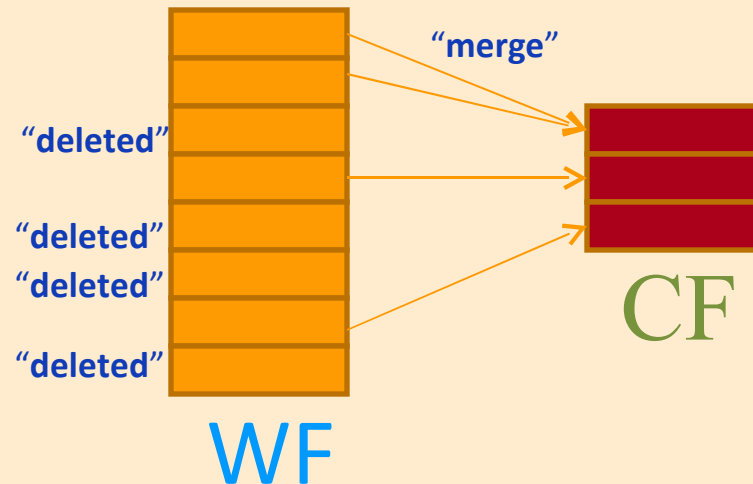
S u b l i m a t i o n :

D e s i g n t h e C o m p a c t F i n g e r p r i n t

- 256 arbitrarily used for **CF** width.
- Create a mapping of keys in the **WF** to the **CF**:
 - a. Measure key frequencies & correlations.
 - b. Omit high-frequency keys
 - c. Merge highly correlated keys
 - d. Merge low-frequency keys
 - e. Shift keys to balance frequencies.

The Actions are “Virtual”

Keep in mind that keys are not literally deleted or merged.
Rather, the map is defined to get this effect.



Wide->Condensed Mapping

0->494 1->494 2->494 3->494 4->494 5->del 6-> 2
7->493 8->493 9->493 10->493 11->454 12->del 13->493
etc.

Input and Definitions

- *WF width = 3306*
- *GF width = 256*
- *GF desired frequency: we'll try several*

- **Load** = Avg # keys per structure
= Avg key freq * # keys
- **Optimal Load** = load for desired freq.
- **Crowdedness** = current load / optimal load

Step 1. Measure Key Frequencies and Correlations

RAW KEY COUNTS

31011	3769	721
23463	1706	520931
137420	27079	<i>etc.</i>

FREQUENCIES

2	0	0
2	0	41
11	2	<i>etc.</i>

Most correlating keys (*extract*)

73 (32, 33)	50 (342 , 677)	43 (767, 2463)
41 (1465, 3247)	40 (1647, 2633)	40 (85, 165)
...		

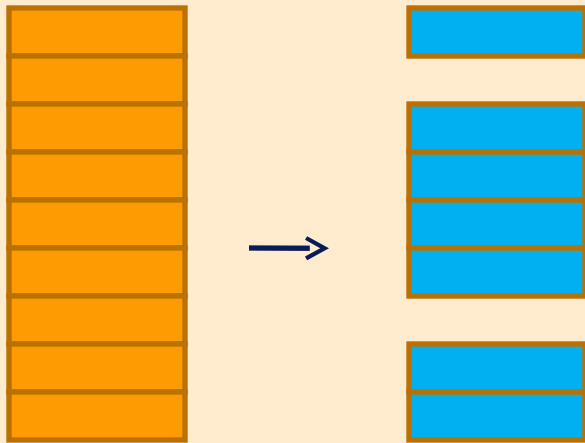
<u>Key#</u>	<u>W-Provenance</u>
-------------	---------------------

342	O-Cr=O
------------	--------

677	O-Cr
------------	------

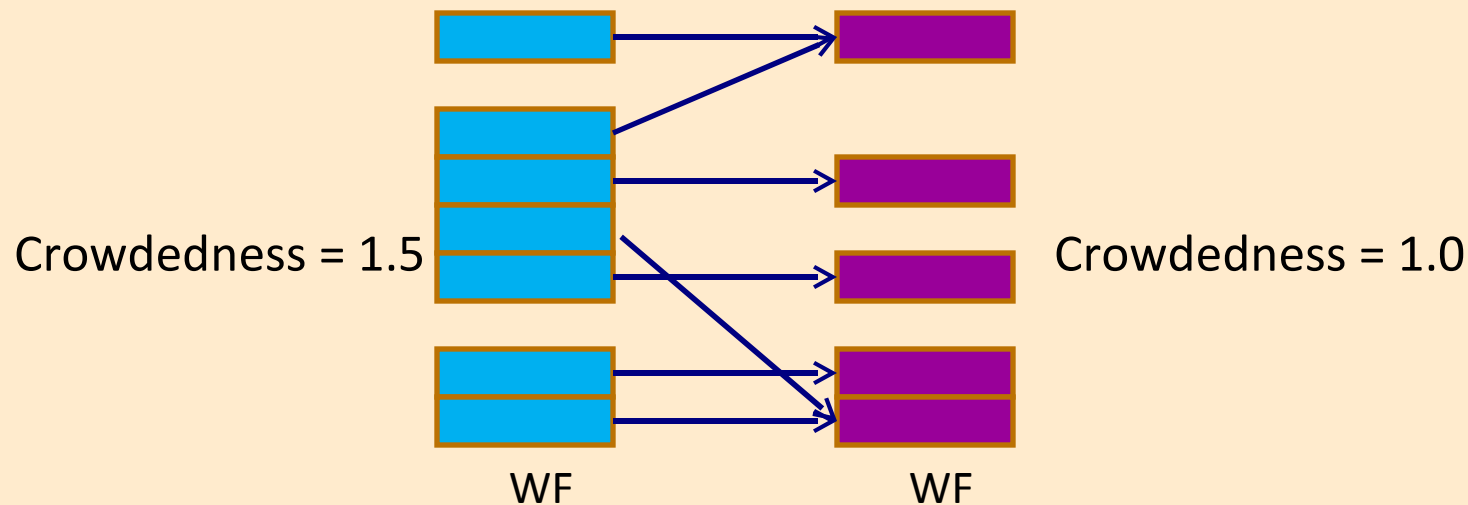
Step 2. Omit High-Frequency Keys

- Choose arbitrary cutoff:
max useful freq = optimal freq + 0.2
- “Delete” all keys more common than this frequency.



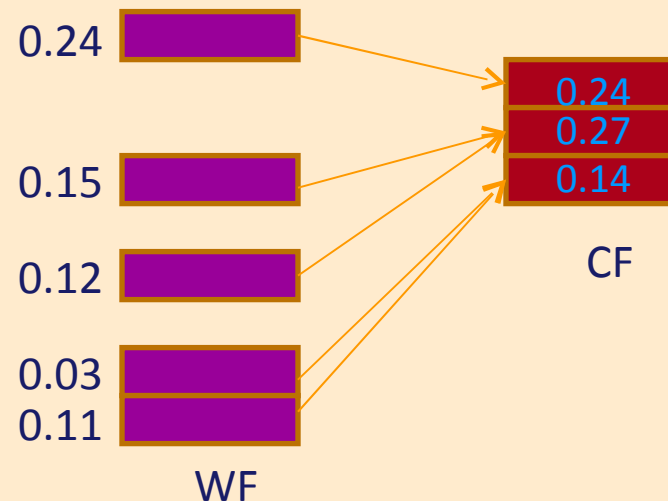
Step 3. Merge Correlating Keys

- Correlation is pairwise for simplicity: the fraction of times they occur together.
- The correlation sum for a key is the sum of its correlations to every other key.
- Delete the key with the highest correlation sum, repeating until the crowdedness falls to 1.0.



Step 4. Merge Low-Frequency Keys

- Merge the lowest frequency keys until the desired number of keys is obtained.
- Balance frequencies in the **CF** by reassigning wide keys from one compact key to another.



Making a CF with the Map

- Use the map WF \rightarrow CF to create real CF's.

Wide- \rightarrow Condensed Mapping

0- \rightarrow 494 1- \rightarrow 494 2- \rightarrow 494 3- \rightarrow 494 4- \rightarrow 494 5- \rightarrow del 6- \rightarrow 2
7- \rightarrow 493 8- \rightarrow 493 9- \rightarrow 493 10- \rightarrow 493 11- \rightarrow 454 12- \rightarrow del 13- \rightarrow 493
etc.

1. Generate the structure's WF.

WF = { 3, 8, 23, 52-55, 57, ... }

3. Apply the map to arrive at the CF.

3 maps to 7 23 maps to 1
8 maps to 504 *etc.*
CF = { 1, 7, 9-11, ... }

Measure the CF Effectiveness

- Use substructure searching.
- Any collection of representative queries.
- **Screen-out Rate (S OR)** = fraction of targets screened out for a given query.
- **Average Screen-out Rate (AS OR)** = avg over all queries.
- **Relative Screen Effectiveness (RSE)**:

$$\text{RSE} = \frac{\text{AS OR (Compact Screen)}}{\text{AS OR (Wide Screen)}} \quad 0..1$$

Results

The screen layout configurations are represented by letters, in order of decreasing effectiveness:

Code	RSE	Avg_Rej	Avg_False+	Description
A	1.0000	0.95535	0.01726	3306-7
B	0.9930	0.94867	0.02394	3306-7.256.40-50
C	0.9930	0.94867	0.02394	3306-7.256.50-60
D	0.9930	0.94867	0.02394	3306-7.256.60-70
E	0.9930	0.94867	0.02394	3306-7.256.70-80
F	0.9930	0.94867	0.02394	3306-7.256.90-100
G	0.9913	0.94700	0.02561	3306-7.256.30-40
H	0.9567	0.91395	0.05866	3306-7.256.20-30
I	0.8614	0.82290	0.14971	3306-7.256.10-20
J	0.4234	0.40446	0.56814	3306-7.256.00-10

Where: Avg_Rej line is the screen-out rate averaged over all queries.

RSE (relative screening effectiveness) is the ratio of a config's 'Avg_Rej' to that of the best.

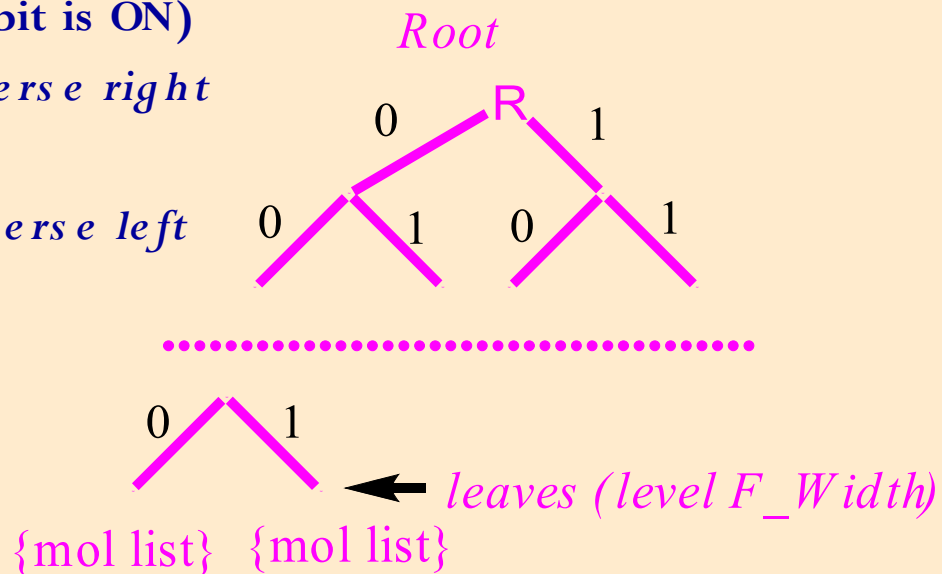
Avg_False+ is the average rate of false positives.

Part 2:

The Bit Binary Tree

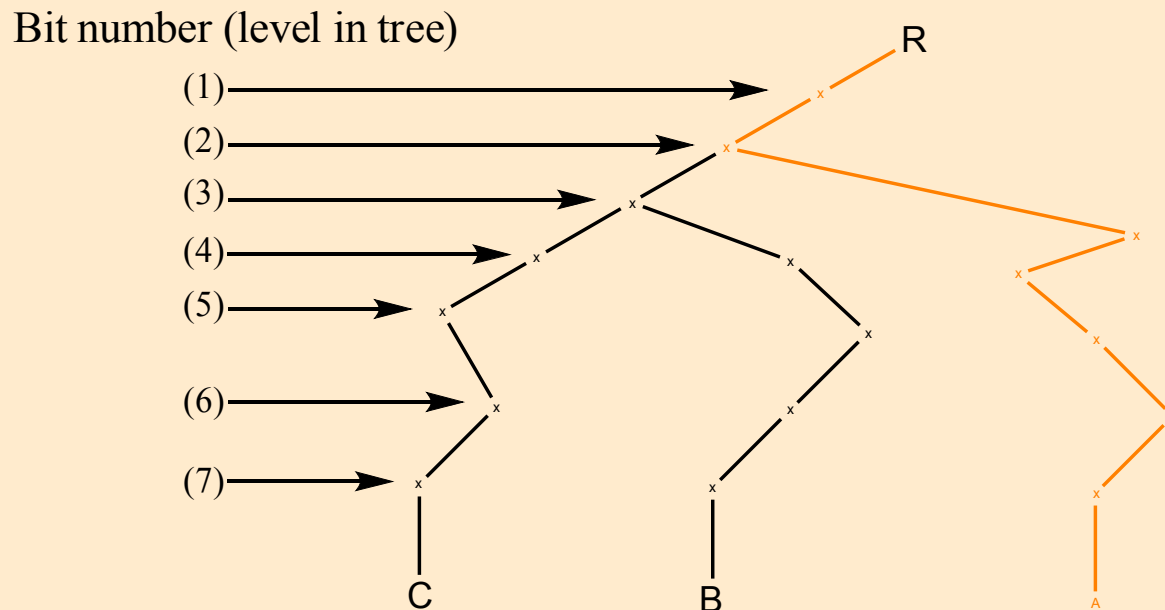
The Bit Binary Tree

- Data structure for rapid indexing and similarity calculation
- To insert a new molecule M into the tree:
 - Compute M's fingerprint (F)
 - Let $F_Width = \# \text{ bits in fingerprint}$
 - Set $node$ to the root of the tree
 - For (level = 1 to F_Width)
 - if F[level] is TRUE (i.e. the levelth bit is ON)
 - $node = node \rightarrow RightChild$ //traverse right
 - else
 - $node = node \rightarrow LeftChild$ //traverse left
- 5. Store M with the leaf-node "node".



Bit Binary Tree - Example

	1	2	3	4	5	6	7
A	◊	◊	1	◊	1	1	◊
B	◊	◊	◊	1	1	◊	◊
C	◊	◊	◊	◊	◊	1	◊

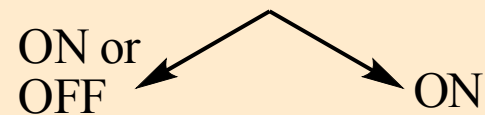


Bit Binary Tree - Properties

- The i^{th} bit in a fingerprint presents at the i^{th} level in the tree.
 - Molecules with the first N bits in common in their fingerprints share the same path through the first N levels in the tree.
 - » Some compaction is possible!
- Molecules are stored at the leaves of the tree
 - Molecules with the same fingerprint are stored at the same leaf.

Bit Binary Tree – Substructure Screening

- The Goal: substructure screening
 - Find all targets whose fingerprints contain that of the query.
- Using the bit binary tree, with a given query fingerprint:
 - Walk the tree
 - At depth d in the tree, examine the d^{th} bit in the query, $Q[d]$
 - » If $Q[d]$ is ON, ONLY walk right in the tree
 - » If $Q[d]$ is OFF, walk left and right in the tree
 - Every leaf reached contains a target molecule that has ALL bits turned ON that are also on in the query.



Bit Binary Tree – Molecular Similarity

- Recall: similarity between target and query molecules is computed with the Tanimoto coefficient:

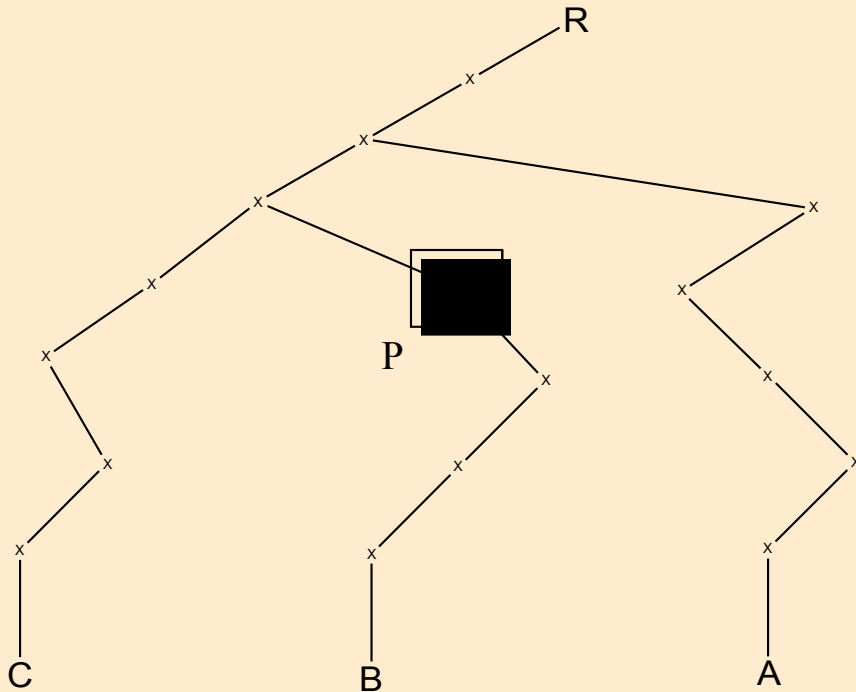
$$\gg T(A,B) = \frac{|A \cap B|}{|A \cup B|} \frac{\text{\# ON bits } A \text{ and } B \text{ have in common}}{\text{\# ON bits when } A \text{ and } B \text{ are OR'ed}}$$

- The bit tree can be used to devise a “Tanimoto lookahead condition” such that the whole tree does not need to be traversed in a search.

Tanimoto Lookahead Condition

- Given a query with bits:
0 0 1 1 1 0 0

↑
d=4



- Traverse the target tree. At position p in the tree at depth d , keep track of:
 - #bits shared between the targets' fingerprint up to (d) and the query's:
 $\text{RIC}(p) = \text{Running intersection count}$
 - # ON bits from targets' fingerprint up to (d) and the query combined:
 $\text{RUC}(p) = \text{Running union count}$
- At the marked position:
 - From the query: 0 0 1 1
 - From the target: 0 0 0 1
 - $\text{RIC}(p) = 1; \text{RUC}(p) = 2$

Tanimoto Lookahead Condition

- We can compute:
 - RIC(p,d) – Running intersection count at position p (depth d in the tree)
 - RUC(p,d) – Running union count at position p (depth d in the tree)
- We know how many bits are ON AFTER the dth position in the query:
 - Example: Bits in query: 0 0 1 1 1 0 0
 - Then FQC[d] = # bits ON after dth position in query: 3 3 2 1 0 0 0
- Thus at any position in the tree we can compute the maximum value of the Tanimoto similarity to ANY molecule in a lower branch.
 - $$T_{\max} = \frac{\max(\# \text{ keys in common})}{\min(\text{total } \# \text{ keys})} = \frac{\text{RIC}(p,d) + \text{FQC}(d)}{\text{RUC}(p,d) + \text{FQC}(d)}$$
 - » Numerator is the maximum # bits in common between any molecule lower in the tree and the query.
 - » Denominator is the minimum # bits that can be ON after OR'ing the bits between any molecule lower in the tree and the query.
- We can prune the tree if the **Tanimoto Lookahead Condition** holds:
 - $T_{\max} < \text{similarity threshold}$

Molecular Similarity Performance

# Mols	Simil Thresh	Mean # hits	Mean brute time (ms)	Mean tree time (ms)	# Tree nodes	Mean visited tree nodes
100,000	0.6	236	83	293	23,499,449	3,195,786
	0.7	88	83	156	23,499,449	1,668,634
	0.8	35	83	65	23,499,449	684,639
	0.9	11	83	11	23,499,449	114,796
450,477	0.6	685	353	1189	96,020,200	13,054,848
	0.7	168	353	625	96,020,200	6,755,879
	0.8	51	353	253	96,020,200	2,668,936
	0.9	13	353	39	96,020,200	391,582

Bit Binary Tree Compression

- **Observation:**

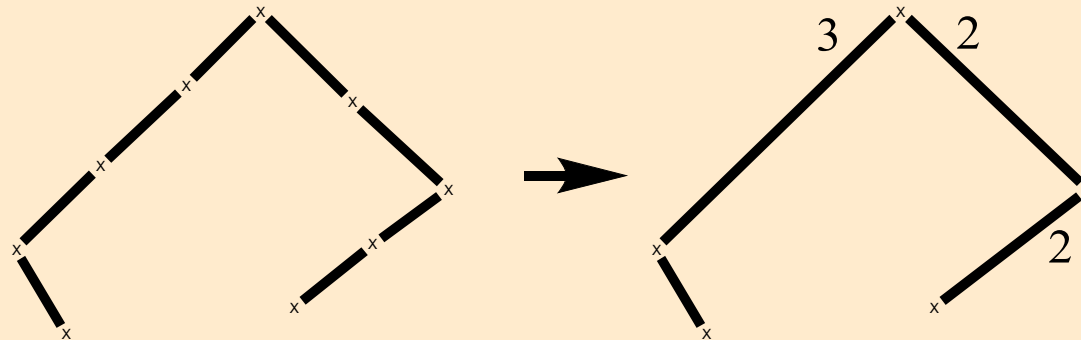
- Fingerprints are often sparse, with long series of consecutive 0's.

- **Thus**

- Compress the tree by storing the “run length” at each tree node
 - » Run length encode the number of consecutive left or right children at any given node

- **Results in fewer tree nodes**

- Faster computation
- More compact representation



Molecular Similarity Performance - Compressed

# mols	Simil Thresh	# Mean hits	Mean brute time (ms)	Mean tree time (ms)	# Tree nodes	Mean visited tree nodes
100,000	0.6	236	76	37	4,989,252	939,971
	0.7	88	76	23	4,989,252	550,362
	0.8	35	76	12	4,989,252	248,803
	0.9	11	76	4	4,989,252	48,166
450,477	0.6	685	328	169	25,551,291	4,239,553
	0.7	168	328	104	25,551,291	2,372,574
	0.8	51	328	52	25,551,291	1,006,140
	0.9	13	328	11	25,551,291	168,718
967,749	0.6	2021	665	278	45,730,909	7,257,712
	0.7	602	665	175	45,730,909	4,130,738
	0.8	187	665	90	45,730,909	1,811,328
	0.9	52	665	21	45,730,909	325,567

Substructure Screening Performance - Compressed

# mols	Mean # hits	Mean brute time (ms)	Mean tree time (s)	# Tree nodes	Mean visited tree nodes
100,000	829	58	5	4,989,252	197,168
420,889	2694	251	18	25,551,291	748,251
852,661	12016	504	50	45,730,909	2,218,782

A c k n o w l e d g e m e n t s

- Thanks to CambridgeSoft for their support

<http://www.CambridgeSoft.com>

- This research was conducted by Andrew Smellie (bit binary trees) and myself (sublime fingerprints)