Superseding Traditional Indexes with Multicriteria Data Structures

GIORGIO VINCIGUERRA
PhD student in Computer Science
giorgio.vinciguerra@phd.unipi.it
Outline

1. Multicriteria data structures
2. The dictionary problem
   - External memory model
   - Multiway trees
   - Novel approaches
   - Our results
3. Bonus slides
Motivation

1. Algorithms and data structures often offer a collection of different trade-offs (e.g. time, space occupancy, energy consumption, ...)

2. Software engineers have to choose the one that best fits the needs of their application

3. These needs change with time, data, devices, and users
Multicriteria Data Structures

A *multicriteria data structure* selects the best data structure within some performance and computational constraints.

- **FAMILY** of data structures
- **CONSTRAINTS** space, time, energy...
- **OPTIMISATION** find the best structure
The dictionary problem

We are given a set of “objects”, and we are asked to store them succinctly and to support efficient retrieval.

Databases
File Systems
Search Engines
Social Networks
<table>
<thead>
<tr>
<th>id</th>
<th>comment</th>
<th>from_user</th>
<th>to_user</th>
<th>created_at</th>
<th>updated_at</th>
<th>item_id</th>
<th>attachment</th>
<th>instagram_id</th>
<th>is_disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>290</td>
<td>This jacket</td>
<td>5569</td>
<td>4634</td>
<td>2016-10-07</td>
<td>2016-10-07</td>
<td>11378</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>291</td>
<td>Love everything!</td>
<td>5569</td>
<td>6327</td>
<td>2016-12-28</td>
<td>2016-12-28</td>
<td>15686</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>292</td>
<td>Thank you 😊 emcollins</td>
<td>6620</td>
<td>6620</td>
<td>2016-11-27</td>
<td>2016-11-27</td>
<td>8234</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>293</td>
<td>Awesome!</td>
<td>5569</td>
<td>2024</td>
<td>2016-11-19</td>
<td>2016-11-19</td>
<td>15055</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>294</td>
<td>Love this top 😍</td>
<td>5569</td>
<td>4147</td>
<td>2016-09-28</td>
<td>2016-09-28</td>
<td>10590</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>295</td>
<td>Love!!!! 🌸 (%)</td>
<td>5569</td>
<td>6998</td>
<td>2016-09-07</td>
<td>2016-09-07</td>
<td>9073</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>296</td>
<td>Love this shot 🤩</td>
<td>12141</td>
<td>4384</td>
<td>2016-09-01</td>
<td>2016-09-01</td>
<td>5375</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>297</td>
<td>Love this shot xx</td>
<td>7508</td>
<td>570</td>
<td>2016-06-24</td>
<td>2016-06-24</td>
<td>7970</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>298</td>
<td>So cool 😍</td>
<td>8995</td>
<td>5510</td>
<td>2015-11-01</td>
<td>2015-11-01</td>
<td>13421</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>299</td>
<td>Love all your looks but this is the best look I've seen on the...</td>
<td>8360</td>
<td>9204</td>
<td>2015-06-28</td>
<td>2015-06-28</td>
<td>5684</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>300</td>
<td>Lovely! 😍</td>
<td>5569</td>
<td>10399</td>
<td>2015-10-05</td>
<td>2015-10-05</td>
<td>11201</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>301</td>
<td>Lovely! 😍</td>
<td>5569</td>
<td>6555</td>
<td>2016-11-21</td>
<td>2016-11-21</td>
<td>15190</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>302</td>
<td>Stunning!</td>
<td>6386</td>
<td>8768</td>
<td>2015-08-09</td>
<td>2015-08-09</td>
<td>5614</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>303</td>
<td>Can I join u skating, in that amazing outfit?! 😍</td>
<td>9691</td>
<td>11728</td>
<td>2016-06-08</td>
<td>2016-06-08</td>
<td>6437</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>304</td>
<td>Nice texture mix!</td>
<td>11863</td>
<td>5569</td>
<td>2016-09-25</td>
<td>2016-09-25</td>
<td>7871</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>305</td>
<td>Bangert! 😍</td>
<td>8572</td>
<td>880</td>
<td>2016-10-06</td>
<td>2016-10-06</td>
<td>11304</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>306</td>
<td>So pretty! 😍</td>
<td>9691</td>
<td>8852</td>
<td>2016-08-08</td>
<td>2016-08-08</td>
<td>6440</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>307</td>
<td>Love this! Saw the full shoot on your blog x</td>
<td>378</td>
<td>11302</td>
<td>2015-05-09</td>
<td>2015-05-09</td>
<td>436</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>308</td>
<td>Your hair!!! 😍 (°_°)</td>
<td>5569</td>
<td>8809</td>
<td>2016-08-26</td>
<td>2016-08-26</td>
<td>8326</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>309</td>
<td>Your eyes are gorgeous!</td>
<td>3630</td>
<td>8995</td>
<td>2015-07-30</td>
<td>2015-07-30</td>
<td>3216</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>310</td>
<td>Cool shirt!</td>
<td>8965</td>
<td>1191</td>
<td>2016-09-15</td>
<td>2016-09-15</td>
<td>9554</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
<tr>
<td>311</td>
<td>Love the top 😍</td>
<td>7663</td>
<td>12232</td>
<td>2016-10-12</td>
<td>2016-10-12</td>
<td>8021</td>
<td>NULL</td>
<td>NULL</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
Memory hierarchy
Memory hierarchy
Memory hierarchy
Memory hierarchy

- L1: 32 KB
- L2: 256 KB
- L3: 3 MB
- 8 GB
- 256 GB
- ∞ TB

- 100 ns
- 16 µs (SSD)
- 3 ms (HDD)
- 150 ms
Latency Numbers Every Programmer Should Know

1990

- 1ns
- L1 cache reference: 181ns
- Main memory reference: 207ns
- 1,000ns ≈ 1μs
- Compress 1KB with Zippy: 362,000ns ≈ 362μs
- Send 2,000 bytes over commodity network: 1,448,000ns ≈ 1,448μs
- SSD random read: 19,000ns ≈ 19μs
- Read 1,000,000 bytes sequentially from memory: 3,038,000ns ≈ 3,038μs
- Read 1,000,000 bytes sequentially from SSD: 50,000,000ns ≈ 50ms
- Disk seek: 20,000,000ns ≈ 20ms
- Read 1,000,000 bytes sequentially from disk: 640,000,000ns ≈ 640ms
The External Memory (aka I/O) model

1. Internal memory (RAM) of capacity $M$
2. External memory (disk) of unlimited capacity
3. RAM and disk exchange blocks of size $B$
4. Count # transfers in Big O instead of # ops

$B \approx 4KiB$
The External Memory (aka I/O) model

1. Internal memory (RAM) of capacity $M$
2. External memory (disk) of unlimited capacity
3. RAM and disk exchange blocks of size $B$
4. Count # transfers in Big O instead of # ops

$B = 64B$
Back to the dictionary problem

We are given a set of “objects”, and we are asked to store them succinctly and to support efficient retrieval.

Integers or reals: e.g. point and range queries

```
61 71 12 15 18 1 24 22 88 34 3 10 5 13 55 44 60 2 5 74 90 81
```
Predecessor search & range queries

\[ \text{pred}(36) = 36, \quad \text{range}(67, 110), \quad \text{pred}(50) = 48 \]

\[ M \]
Baseline solutions for predecessor search

<table>
<thead>
<tr>
<th>Solution</th>
<th>RAM model Worst case time</th>
<th>EM model Worst case I/Os</th>
<th>EM model Best case I/Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>$O(n)$</td>
<td>$O(n/B)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

$M$ is a memory module with $B = 4$. $n$ is the number of elements in the array.
Baseline solutions for predecessor search

\[ M_B = 4 \]

<table>
<thead>
<tr>
<th>Solution</th>
<th>RAM model Worst case time</th>
<th>EM model Worst case I/Os</th>
<th>EM model Best case I/Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>(0(n))</td>
<td>(0(n/B))</td>
<td>(0(1))</td>
</tr>
<tr>
<td>Binary search</td>
<td>(0(\log n))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Baseline solutions for predecessor search

<table>
<thead>
<tr>
<th>Solution</th>
<th>RAM model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Worst case time</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Scan</td>
<td>0(n)</td>
</tr>
<tr>
<td>Binary search</td>
<td>0(log n)</td>
</tr>
</tbody>
</table>
B⁺ trees
B$^+$ trees

48?
### B+ trees

#### Solution

<table>
<thead>
<tr>
<th>Space</th>
<th>RAM model Worst case time</th>
<th>EM model Worst case I/Os</th>
<th>EM model Best case I/Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scan</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n/B)</td>
</tr>
<tr>
<td>Binary search</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log(n/B))</td>
</tr>
<tr>
<td>B+ tree</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log_B n)</td>
</tr>
</tbody>
</table>
B-trees are everywhere


2. This is still true today
B-trees are everywhere
B-trees are machine learning models

“All existing index structures can be replaced with other types of models, including deep-learning models, which we term learned indexes.”

Trained on the dataset \{(key_i, i)\}_{i=1,...,n}

position

position – \(\epsilon\), position + \(\epsilon\)
B-trees are machine learning models

“All existing index structures can be replaced with other types of models, including deep-learning models, which we term learned indexes.”
The Recursive Model Index (RMI)

```
key ∈ [pos − ε, pos + ε]?
```

Stage 1
Stage 2
Stage 3
Construction of RMI

1. Train the root model on the dataset
2. Use it to distribute keys to the next stage
3. Repeat for each model in the next stage (on smaller datasets)
Performance of RMI

<table>
<thead>
<tr>
<th>Type</th>
<th>Config</th>
<th>Map Data</th>
<th>Web Data</th>
<th>Log-Normal Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Size (MB)</td>
<td>Lookup (ns)</td>
<td>Model (ns)</td>
</tr>
<tr>
<td>Btree</td>
<td>page size: 32</td>
<td>52.45 (4.00x)</td>
<td>274 (0.97x)</td>
<td>198 (72.3%)</td>
</tr>
<tr>
<td></td>
<td>page size: 64</td>
<td>26.23 (2.00x)</td>
<td>277 (0.96x)</td>
<td>172 (62.0%)</td>
</tr>
<tr>
<td></td>
<td>page size: 128</td>
<td>13.11 (1.00x)</td>
<td>265 (1.00x)</td>
<td>134 (50.8%)</td>
</tr>
<tr>
<td></td>
<td>page size: 256</td>
<td>6.56 (0.50x)</td>
<td>267 (0.99x)</td>
<td>114 (42.7%)</td>
</tr>
<tr>
<td></td>
<td>page size: 512</td>
<td>3.28 (0.25x)</td>
<td>286 (0.93x)</td>
<td>101 (35.3%)</td>
</tr>
<tr>
<td>Learned Index</td>
<td>2nd stage models: 10k</td>
<td>0.15 (0.01x)</td>
<td>98 (2.70x)</td>
<td>31 (31.6%)</td>
</tr>
<tr>
<td></td>
<td>2nd stage models: 50k</td>
<td>0.76 (0.06x)</td>
<td>85 (3.11x)</td>
<td>39 (45.9%)</td>
</tr>
<tr>
<td></td>
<td>2nd stage models: 100k</td>
<td>1.53 (0.12x)</td>
<td>82 (3.21x)</td>
<td>41 (50.2%)</td>
</tr>
<tr>
<td></td>
<td>2nd stage models: 200k</td>
<td>3.05 (0.23x)</td>
<td>86 (3.08x)</td>
<td>50 (58.1%)</td>
</tr>
</tbody>
</table>

Figure 4: Learned Index vs B-Tree
Limitations of RMI

1. Fixed structure with many hyperparameters
   # stages, # models in each stage, kinds of regression models

2. No a priori error guarantees
   Difficult to predict latencies

3. Models are agnostic to the power of models below
   Can result in underused models (waste of space)
Our idea (submitted)

Compute the **optimal** piecewise linear approx with **guaranteed error** $\varepsilon$ in $O(n)$
Our idea (submitted)

Save the $m$ segments in a vector as triples $s_i = (\text{key, slope, intercept})$
Our idea (submitted)

Drop all the points except $s_i$. key
Our idea (submitted)

... and repeat!
Memory layout of the PGM-index

$s_4 = (key, slope, intercept)$

$pos = f_{s_4}(k)$

$[pos - \varepsilon, pos + \varepsilon]$
Some asymptotic bounds

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Space of index</th>
<th>RAM model</th>
<th>EM model</th>
<th>EM model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Worst case time</td>
<td>Worst case I/Os</td>
<td>Best case I/Os</td>
</tr>
<tr>
<td>Plain sorted array</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O\left(\log \frac{n}{B}\right)$</td>
<td>$O\left(\log \frac{n}{B}\right)$</td>
</tr>
<tr>
<td>Multiway tree</td>
<td>$\Theta(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log_B n)$</td>
<td>$O(\log_B n)$</td>
</tr>
<tr>
<td>RMI</td>
<td>Fixed</td>
<td>$O(?)$</td>
<td>$O(?)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>PGM-index</td>
<td>$\Theta(m)$</td>
<td>$O(\log m)$</td>
<td>$O\left(\log_e m\right)$</td>
<td>$\Omega(\log m) \geq 2\varepsilon = \Omega(B)$</td>
</tr>
</tbody>
</table>
PGM-index in practice

3 seconds to compute

Whole datasets

First 25M entries

Web logs = 715M points
Longitude = 166M points
IoT = 26M points
Space-time performance

![Graph showing performance comparison between different index space sizes and time measurements.](image)
How to explore this space of trade-offs?

Given a space bound $S$, find efficiently the index that minimizes the query time within space $S$ and vice versa.
A multicriteria data structure is defined by a family of data structures and an optimisation algorithm that selects the best data structure in the family within some computational constraints.
The Multicriteria PGM-index

1. We designed a cost model for the space $s(\varepsilon)$ and the time $t(\varepsilon)$
2. ... but we don’t have a closed formula for $s(\varepsilon)$, it depends on the input array
3. We fit $s(\varepsilon)$ with a power law of the form $a\varepsilon^{-b}$
Under the hood

1. A sort of interpolation search over $\varepsilon$ values
2. Each iteration improves the fitting of $a\varepsilon^{-b}$ updating $a, b$
3. Bias the $\varepsilon$-iterate towards the midpoint of a bin. search
4. In practice, given a space (time) bound, it finds the fastest (most compact) index for 715M keys in < 1 min
Future work

1. Insertion and deletions
2. Non-linear models
3. Compression
Bonus slides

Tools that you may find useful
**Elapsed Time**: 196.366s

- Clockticks: 667,690,000
- Instructions Retired: 828,920,000
- CPI Rate: 0.805
- MUX Reliability: 0.951
- Retiring: 52.8% of Pipeline Slots
- Front-End Bound: 8.4% of Pipeline Slots
- Bad Speculation: 12.5% of Pipeline Slots
- Branch Mispredict: 12.5% of Pipeline Slots
- Machine Clears: 0.0% of Pipeline Slots
- Back-End Bound: 26.3% of Pipeline Slots
- Memory Bound: 15.8% of Pipeline Slots
- L1 Bound: 23.4% of Clockticks
  - DTLB Overhead: 0.0% of Clockticks
  - Loads Blocked by Store Forwarding: 8.1% of Clockticks
  - Lock Latency: 0.0% of Clockticks
  - Split Loads: 0.0% of Clockticks
  - 4K Aliasing: 0.0% of Clockticks
  - FB Full: 1.1% of Clockticks
  - L2 Bound: 0.0% of Clockticks
- L3 Bound: 3.1% of Clockticks
- DRAM Bound: 2.3% of Clockticks
- Store Bound: 3.9% of Clockticks
- Core Bound: 10.5% of Pipeline Slots
- Divider: 0.0% of Clockticks
- Port Utilization: 21.8% of Clockticks
- Cycles of 0 Ports Utilized: 30.8% of Clockticks
- Cycles of 1 Port Utilized: 16.2% of Clockticks
- Cycles of 2 Ports Utilized: 18.6% of Clockticks
- Cycles of 3+ Ports Utilized: 34.2% of Clockticks
- Vector Capacity Upper (VPUD): 0.0%

**μPipe**

This diagram represents inefficiencies in CPU usage. Treat it as a pipe with an output flow equal to the "pipe efficiency" ratio: (Actual Instructions Retired)/(Maximum Possible Instruction Retired). If there are pipeline stalls decreasing the pipe efficiency, the pipe shape gets more narrow.

- 8.41% - Front-End Bound
- 15.78% - Memory Bound
- 52.81% - Retiring
- 10.52% - Core Bound
- 12.48% - Bad Speculation

The metric value is high. This can indicate that the

This metric represents how much Core non-memory
assert(value >= parent->segments[i].key);
update_data();

// Segment approximation
auto pos_f = std::min(next_segment.inter[0], pos);
auto pos_u = uint32_t(pos_f);
pos = UNLIKELY(std::signbit(pos_f)) ? 0u : pos_u;

// Correction of the position
__mm_prefetch(parent->ptr_data + pos + 15,
__mm512 Val = __mm512_set1_epi32(value);
__mm512 Keys = __mm512_loadu_si512(reinter);
__m512 mask = __mm512_cmpge_epi32_mask(0);
uint32_t mask = __mm_popcnt_u32(__mm512_movzb_epi32(mask));

uint32_t count = __mm_popcnt_u32(__mm512_movzb_epi32(mask));

pos += count - 1;
upper_bound = *(parent->ptr_data + pos + 1);
assert(parent->ptr_data[pos] <= value);
assert(parent->ptr_data[pos + 1] > value);

return pos;

reference operator*() { return segment.key; }
The \code{\%timeit} built-in line magic

```
In [1]:
from random import uniform
from itertools import cycle

gen_point = \code{lambda}: (uniform(0, 100), uniform(0, 100))
points_pairs = [\((\text{\textbf{gen} } \text{\textpoint} \text{\textbf{point}()} \text{, } \text{\textbf{gen} } \text{\textpoint} ()\) \textbf{for } _ \textbf{in} \text{range(100000)}\]
iter_points_pairs = cycle(points_pairs)
```

```
In [2]:
import math

def py_distance(p1, p2):
    dx = p2[0] - p1[0]
    return math.sqrt(dx**2 + dy**2)

\%timeit pts = next(iter_points_pairs); py_distance(*pts)

891 ns ± 139 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
In [3]:
from scipy.spatial import distance

\%timeit pts = next(iter_points_pairs); distance.euclidean(*pts)

31.9 \(\mu\)s ± 9.77 \(\mu\)s per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```
Cython

In [ ]:

```
pip install cython
%load_ext Cython
```

In [5]:

```
%%cython -a

cimport libc.math

def cython_distance((double, double) p1, (double, double) p2):
    cdef double dx = p2[0] - p1[0]
    cdef double res = libc.math.sqrt(dx * dx + dy * dy)
    return res
```

Out[5]:

Generated by Cython 0.29.7

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

1:
   cimport libc.math
3:
+4: def cython_distance((double, double) p1, (double, double) p2):
+5:   cdef double dx = p2[0] - p1[0]
+7:   cdef double res = libc.math.sqrt(dx * dx + dy * dy)
+8:   return res

In [6]:

```
%timeit pts = next(iter_points_pairs); cython_distance(*pts)
```

272 ns ± 173 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)

3× faster than py_distance
117× faster than scipy.spatial.distance.euclidean
The `%lprun` magic (from the `line_profiler` module)

```
In [ ]: %pip install line_profiler
%load_ext line_profiler

In [9]: %lprun -f distance.euclidean [distance.euclidean(p1, p2) for p1, p2 in points_pairs]
```

```
def euclidean(u, v, w=None):
    """
    Computes the Euclidean distance between two 1-D arrays.

    The Euclidean distance between 1-D arrays \( \mathbf{u} \) and \( \mathbf{v} \), is defined as

    .. math::

        \| \|\mathbf{u} - \mathbf{v}\|\|_2^2 = \left( \sum_{i} (u_i - v_i)^2 \right)^{1/2}

    Parameters

    ----------
    u : (N,) array_like
        Input array.
    v : (N,) array_like
        Input array.
    w : (N,) array_like, optional
        The weights for each value in \( \mathbf{u} \) and \( \mathbf{v} \). Default is None, which gives each value a weight of 1.0

    Returns

    -------
    euclidean : double
        The Euclidean distance between vectors \( \mathbf{u} \) and \( \mathbf{v} \).

    Examples

    -------
    >>> from scipy.spatial import distance
    >>> distance.euclidean([1, 0, 0], [0, 1, 0])
    1.4142135623730951
    >>> distance.euclidean([1, 1, 0], [0, 1, 0])
    1.0
    >>> return minkowski(u, v, p=2, w=w)
```

---

Total time: 8.01555 s
Function: euclidean at line 566

<table>
<thead>
<tr>
<th>Line #</th>
<th>Hits</th>
<th>Time</th>
<th>Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>566</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>def euclidean(u, v, w=None):</td>
</tr>
</tbody>
</table>
| 567    |      |      |         |        | """
| 568    |      |      |         |        | Computes the Euclidean distance between two 1-D arrays. |
| 569    |      |      |         |        | The Euclidean distance between 1-D arrays `\u` and `\v`, is defined as |
| 570    |      |      |         |        | .. math:: |
| 571    |      |      |         |        | \|\|\mathbf{u} - \mathbf{v}\|\|_2^2 = \left( \sum_{i} (u_i - v_i)^2 \right)^{1/2} |
| 572    |      |      |         |        | Parameters |
| 573    |      |      |         |        | ---------- |
| 574    |      |      |         |        | u : (N,) array_like |
| 575    |      |      |         |        | Input array. |
| 576    |      |      |         |        | v : (N,) array_like |
| 577    |      |      |         |        | Input array. |
| 578    |      |      |         |        | w : (N,) array_like, optional |
| 579    |      |      |         |        | The weights for each value in `\u` and `\v`. Default is None, which gives each value a weight of 1.0 |
| 580    |      |      |         |        | Returns |
| 581    |      |      |         |        | ------- |
| 582    |      |      |         |        | euclidean : double |
| 583    |      |      |         |        | The Euclidean distance between vectors `\u` and `\v`. |
| 584    |      |      |         |        | Examples |
| 585    |      |      |         |        | ------- |
| 586    |      |      |         |        | >>> from scipy.spatial import distance |
| 587    |      |      |         |        | >>> distance.euclidean([1, 0, 0], [0, 1, 0]) |
| 588    |      |      |         |        | 1.4142135623730951 |
| 589    |      |      |         |        | >>> distance.euclidean([1, 1, 0], [0, 1, 0]) |
| 590    |      |      |         |        | 1.0 |
| 591    |      |      |         |        | """
| 592    |      |      |         |        | return minkowski(u, v, p=2, w=w) |
The %lprun magic (from the line_profiler module)

In [ ]:
```
pip install line_profiler
%load_ext line_profiler
```

In [9]:
```
%lprun -f distance.euclidean [distance.euclidean(p1, p2) for p1, p2 in points_pairs]
```

In [10]:
```
%lprun -f distance.minkowski [distance.euclidean(p1, p2) for p1, p2 in points_pairs]
```

```
477 minkowski : double
  The Minkowski distance between vectors `u` and `v`.

Examples

```
GIORGIO VINCIGUERRA
PhD student in Computer Science
http://pages.di.unipi.it/vinciguerra/
giorgio.vinciguerra@phd.unipi.it