PP-Index: Using Permutation Prefixes for Efficient and Scalable Approximate Similarity Search

Andrea Esuli

andrea.esuli@isti.cnr.it
Istituto di Scienza e Tecnologie dell'Informazione “A. Faedo”
Consiglio Nazionale delle Ricerche
Via Giuseppe Moruzzi, 1 — 56124 Pisa, Italy

ISTI:Science seminar, May 12, 2009
Outline

1. Introduction
2. The PP-Index
3. Experiments
4. Demo
5. Conclusions
Outline

1 Introduction
   • Similarity search
     • Permutation based methods
     • Local similarity hashing methods

2 The PP-Index

3 Experiments

4 Demo

5 Conclusions
Similarity search

The similarity search model involves:
- A collection of objects $D$, belonging to a domain $O$;
- a query object $q \in O$;
- a distance function $d : O \times O \rightarrow \mathbb{R}^+$.

The goal is to sort the objects in $D$ by their distance with respect to $q$, returning the objects that are closer to $q$, which are considered to be the most similar.

Typically only the $k$-top ranked objects are returned ($k$-NN query), or those within a maximum distance value $r$ (range query).

The determination of a meaningful $r$ value is often a non-easy task.

$k$-NN queries are usually preferred, specially in end-user applications, also for the direct control on the result set size.
Similarity search

Example ($\mathbb{R}^2, L_2$):

\begin{itemize}
  \item $o_1$
  \item $o_2$
  \item $o_3$
  \item $o_4$
  \item $o_5$
  \item $o_6$
  \item $o_7$
  \item $o_8$
  \item $o_9$
  \item $o_{10}$
  \item $o_{11}$
  \item $o_{12}$
  \item $q$
  \item $r$
\end{itemize}

**Figure 1:** Range query.

\begin{itemize}
  \item $o_1$
  \item $o_2$
  \item $o_3$
  \item $o_4$
  \item $o_5$
  \item $o_6$
  \item $o_7$
  \item $o_8$
  \item $o_9$
  \item $o_{10}$
  \item $o_{11}$
  \item $o_{12}$
  \item $q$
\end{itemize}

**Figure 2:** $k$-NN query ($k = 5$).
Approximate similarity search

- **Exhaustive search**: for all \( o_i \in D \) compute the distance \( d(q, o_i) \), while keeping track of which objects satisfy the query.
  - It does not scale to large collections.

- **Exact methods**: equivalent to exhaustive search, but using data structures that leverage on the properties of the observed similarity space (e.g., vectorial spaces, metric spaces) in order to reduce the number of objects of \( D \) to be compared with the query.
  - Usually efficient but still not enough for huge collections.

- **Approximate methods**: accepting that the results could contain errors (e.g., \( d(q, o_1) < d(q, o_2) \), \( o_2 \) is in the results and \( o_1 \) is not), gaining efficiency.
  - Approximation is acceptable, e.g., when \( d \) is an approximation of a complex, human-perceived concept of similarity.
  - It (obviously) scales!
  - Typically derived from “relaxed” exact methods.
  - Natively approximated proposals, e.g.: local similarity hashing (LSH) index and permutation-based index (the PP-Index takes inspiration from both).
Approximate similarity search

Approximation quality:
What have we missed? What have we included? How much have we saved?

Figure 3: Approximate result for a $k$-NN query ($k = 5$).
Outline

1 Introduction
   - Similarity search
   - Permutation based methods
   - Local similarity hashing methods

2 The PP-Index

3 Experiments

4 Demo

5 Conclusions
Permutation based methods

Independently proposed by Amato and Savino\(^1\) and Chavez et al.\(^2\), using different data structures.

The idea: *an object is represented by its view of the surrounding world.*

Intuively, if two objects “see” the elements of a set of reference objects \(R\) in the same order of (increasing) distance, they are likely to be close one to the other.

Example

Where am I likely to live if I see the main European cities in the following order?


Permutation based methods

The method:

- A set of reference objects $R = \{r_0, \ldots, r_{|R|-1}\} \subset O$ is defined (e.g., by randomly selecting $|R|$ objects from $D$).
- Every object $o_i \in D$ is then represented by a permutation $\Pi_{o_i}$ of $\langle 0, \ldots, |R| - 1 \rangle$, i.e., the list of the identifiers of reference objects, so that the identifiers are sorted by the distance of their relative reference objects with respect to $o_i$.
- The search process mainly consists in computing $\Pi_q$ and estimating the true distance $d(q, o_i)$ using a permutation-based distance $d'(\Pi_q, \Pi_{o_i})$, e.g., the Spearman's footrule distance.

Amato and Savino have shown that using only the prefix $\Pi_{o_i}^l$ of the permutation $\Pi_{o_i}$ (e.g., $l = 100$ when $|R| = 500$) improves both efficiency and effectiveness.

The PP-Index adopts a permutation-based data representation model, using very short prefixes (e.g., $l = 6$ when $|R| = 1000$).

Differently from previous approaches, the permutation prefixes are used just to quickly find a small set of candidate objects from $D$ for inclusion into results, not to estimate their relative order.
Permutation based methods

Figure 4: Regions of the 2-dimensional space identified by 6 randomly selected reference points, using the Euclidean distance, and full-length permutations (left) or permutation prefixes of length 3.
Introduction

1. Introduction
   - Similarity search
   - Permutation based methods
   - Local similarity hashing methods

2. The PP-Index

3. Experiments

4. Demo

5. Conclusions
Local similarity hashing methods

A family $\mathcal{H}$ of hash functions $f : \mathcal{O} \rightarrow U$ is called $(r, \epsilon, p_1, p_2)$-sensitive, with $r, \epsilon > 0, p_1 > p_2 > 0$, if for any $p, q \in \mathcal{O}$:

- if $d(p, q) \leq r$ then $\Pr[h(p) = h(q)] \geq p_1$
- if $d(p, q) > r(1 + \epsilon)$ then $\Pr[h(p) = h(q)] \leq p_2$

for any function $h$ randomly selected from $\mathcal{H}$.

Intuitively: two objects have a (high) probability $x_1 \geq p_1$ to collide if they are closer than $r$, and a (low) probability $x_2 \leq p_2$ if they are more distant than $r(1 + \epsilon)$.

**LSH-Index**

- $j$ randomly chosen functions $h_i \in \mathcal{H}$ define a hash function $g(x) = (h_1(x)h_2(x) \ldots h_j(x))$, i.e. bad collision probability is significantly lowered to $p_2^j$.

- $t$ different hash tables are built, based on randomly generated $g_1 \ldots g_t$ functions, in order to increase good collision probability.

---

Local similarity hashing methods

It is hard to tune LSH-Index (length of hash keys) in order to obtain good efficacy, due to the dependence between data distribution and hash length.

LSH-Forest\textsuperscript{4}:

- Use of \textit{variable length hash keys}.
- Long hash key are indexed in a \textit{prefix tree} (LSH-Tree).
- At search time the key length is varied in order to retrieve a given number of candidate objects.
- Candidate objects are retrieved \textit{sequentially} from a data storage on disk.
- Multiple LSH-Tree, i.e., a forest, are used to improve effectiveness.

The PP-Index uses similar data structure.

Outline

1 Introduction
2 The PP-Index
3 Experiments
4 Demo
5 Conclusions
Outline

1 Introduction

2 The PP-Index
   • Data structures
   • Algorithms

3 Experiments

4 Demo

5 Conclusions
PP-Index: data structures

The PP-Index represents each indexed object with a permutation prefix of length $l$.

Data structures:
- a prefix tree kept in main memory, indexing the permutation prefixes,
- and a data storage kept on disk, storing the information required to compute real distances between objects in $D$ and any object in $O$.

The prefix tree is used in order to rapidly identify a set of at least $z$ candidates ($z \geq k$), leaving to the original distance function the task of determining the final $k$-NN result from such set of candidates.

Candidates are retrieved from the data storage with a few sequential disk accesses.

The PP-Index adopts a bulk data processing model, similar to the one used for text-based inverted list indexes (assumption on the static nature of data).

It is easy to provide update capabilities (i.e., insert, delete, modify).
Outline

1 Introduction

2 The PP-Index
   - Data structures
   - Algorithms

3 Experiments

4 Demo

5 Conclusions
Figure 5: The BuildIndex function.
PP-Index: building the index

Input: dataset $D$, distance function $d$, reference objects $R$, prefix length $l$.

Indexing process:
- **Main loop**: permutation prefixes are inserted into the prefix tree, data blocks are appended to data storage.
- **Data storage reordering**: data blocks are sorted to reflect the order of prefixes.
PP-Index: building the index

Index characteristics
|D|=10, |R|=6, l=3

Permutation prefixes
w₀₀ =<1, 3, 2> w₀₁ =<2, 3, 0>
w₀₂ =<5, 2, 3> w₀₃ =<4, 1, 3>
w₀₄ =<1, 3, 2> w₀₅ =<4, 1, 3>
w₀₆ =<1, 3, 4> w₀₇ =<5, 2, 3>
w₀₈ =<1, 3, 2> w₀₉ =<4, 3, 5>

**Figure 6:** Sample data.

Main loop: permutation prefixes are inserted into the prefix tree, data blocks are appended to data storage.

**Figure 7:** Index data structure after the first phase of object insertion.
**PP-Index: building the index**

![Prefix tree](image1)

**Figure 8:** Index data structure after the first phase of object insertion.

Data storage reordering: data blocks are sorted to reflect the order of prefixes.

The leaves of the final prefix tree point to *intervals* of the data storage.

Efficiency alert: performed using a $m$-way merge sort algorithm.

![Prefix tree](image2)

**Figure 9:** Index data structure after the first phase of object insertion.
### PP-Index: search function

**Algorithm FindCandidates**

1. $w_q \leftarrow \text{ComputePrefix}(q, R, d, l)$
2. for $i \leftarrow l$ to 1 do
3.   $w_q^i \leftarrow \text{SubPrefix}(w_q, i)$
4.   node $\leftarrow \text{SearchPath}(w_q^i, \text{prefixTree})$
5.   if node $\neq \text{NIL}$ then
6.     $\text{minLeaf} \leftarrow \text{GetMin}(\text{node}, \text{prefixTree})$
7.     $\text{maxLeaf} \leftarrow \text{GetMax}(\text{node}, \text{prefixTree})$
8.     if $(\text{maxLeaf}.h_{\text{end}} - \text{minLeaf}.h_{\text{start}} + 1) \geq z \lor i = 1$ then
9.       return $(\text{minLeaf}.p_{\text{start}}, \text{maxLeaf}.h_{\text{end}})$
10.  return $(0, 0)$

**Figure 10:** The **FindCandidates** function.

Given the prefix representing the query, **FindCandidates** searches for the smallest subtree of the prefix tree pointing to at least $z$ data blocks ($z'$).
PP-Index: search function

```plaintext
SEARCH(q, k, z, index)
1  \( (p^{\text{start}}, p^{\text{end}}) \leftarrow \text{FindCandidates}(q, index.\text{prefixTree}, index.R, index.d, index.l, z) \)
2  \( \text{resultsHeap} \leftarrow \text{EmptyHeap()} \)
3  cursor \leftarrow p^{\text{start}}
4  \textbf{while} cursor \leq p^{\text{end}}
5  \textbf{do} \text{dataBlock} \leftarrow \text{Read}(cursor, index.\text{dataStorage})
6     \text{AdvanceCursor}(cursor)
7  \text{distance} \leftarrow index.d(q, dataBlock.data)
8     \textbf{if} \text{resultsHeap.size} < k
9     \textbf{then} \text{Insert}(\text{resultsHeap}, distance, dataBlock.id)
10    \textbf{else if} distance < \text{resultsHeap.top.distance}
11    \textbf{then} \text{ReplaceTop}(\text{resultsHeap}, distance, dataBlock.id)
12  \text{Sort(\text{resultsHeap})}
13  \textbf{return} \text{resultsHeap}
```

![Figure 11: The SEARCH function.](image)

The \( z' \) candidate data blocks are sequentially read from the data storage. A heap is used to keep track of the best \( k \) results.
PP-Index: improving the search effectiveness

The basic search strategy is designed for efficiency. Effectiveness can be boosted using various search strategies:

- **Multiple index**: building $n$ PP-Index using different $R$ sets, $R_1 \ldots R_n$ (LSH-Forest style).
  - Projecting different $R$-induced “grids” on the objects, helps to approximate a better (less skewed) partitioning of the space.
  - Can be implemented using *data replication* (faster/more storage) or using *data referencing* (slower/less storage).
  - $k$-NN results from the various indexes are merged together in the final one.

- **Multiple query**: generating $m$ perturbed versions of $w_q$ in order to explore the neighborhood of $w_q$.
  - The perturbed $w_q^i$ prefixes are generated by swapping pairs of elements of $w_q$, first selecting those with the smaller distance difference with respect to $q$.
  - All the $w_q^i$ prefixes are used to find candidates on the same index.
PP-Index: prefix tree optimizations

**Scalability alert**: reducing to a single leaf any subtree pointing to less than \( z \) data blocks.

- Applicable when \( z \) is hardcoded into the search function.
- Does not affect search results quality.
- Lossy with respect to index update operations.

![Prefix tree optimization diagrams](image-url)

**Figure 12**: Pruning of only-child paths to leaves.

**Figure 13**: Only-child paths compression.
PP-Index: merging (and updating) the index

**Scalability alert:** the index (prefix tree) reaches its maximum memory requirement at the end of the main loop of the indexing process.

Could not fit into memory, when the final index will (after optimizations).

**Strategy:** building many smaller indexes, using the same $R$ set, then merging them together.

The merge process is efficient:

- The source prefix trees are merged into the final prefix tree by performing a parallel **ordered visit** on them.
- Data storages are merged into the final data storage while building the final prefix tree.
- Can be done from-disk-to-disk, minimum memory occupation.
- Linear cost with respect to index size (if not done in an $m$-way style).
- Uses only sequential reads/writes.

Update operations can be supported by keeping track of such operations by using a small all-in-memory index and performing periodic merge operations.
Outline

1. Introduction
2. The PP-Index
3. Experiments
4. Demo
5. Conclusions
Outline

1. Introduction
2. The PP-Index
3. Experiments
   - The CoPhIR collection
   - Evaluation measures
   - Results
4. Demo
5. Conclusions
The CoPhIR consists of a crawl of 106 millions images from the Flickr photo sharing website.

Textual data + five MPEG-7 visual descriptors (240 GB of XML description data). Visual similarity measure: linear combination of distance functions defined on the MPEG-7 descriptors.

<table>
<thead>
<tr>
<th>MPEG-7 Visual Descriptor</th>
<th>Distance type</th>
<th>Dimension</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalable Color</td>
<td>$L_1$</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>Color Structure</td>
<td>$L_1$</td>
<td>64</td>
<td>3</td>
</tr>
<tr>
<td>Color Layout</td>
<td>sum of $L_2$</td>
<td>80</td>
<td>2</td>
</tr>
<tr>
<td>Edge Histogram</td>
<td>$L_1$</td>
<td>62</td>
<td>4</td>
</tr>
<tr>
<td>Homogeneous Texture</td>
<td>$L_1$</td>
<td>12</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Table 1: Details on the five MPEG-7 visual descriptors used in CoPhIR, and the weights used in the linear combination. The “Dim.” column refer to the specific dimension for visual descriptors adopted by the CoPhIR data set.

Experiments made on 1, 10, and 100 millions images, using 100 randomly selected images (excluded from indexes).

Outline

1 Introduction

2 The PP-Index

3 Experiments
   - The CoPhIR collection
   - Evaluation measures
   - Results

4 Demo

5 Conclusions
Evaluation measures

Effectiveness measures:

- **Recall** (*ranking*-based):  
  \[
  \text{Recall}(k) = \frac{|D_q^k \cap P_q^k|}{k} \quad (1)
  \]

- **Relative Distance Error** (*distance*-based):  
  \[
  \text{RDE}(k) = 1 - \frac{1}{k} \sum_{i=1}^{k} \frac{d(q, P_q^k(i))}{d(q, D_q^k(i))} \quad (2)
  \]
  where $D_q^k$ is the list of the $k$ closest elements of $D$ to $q$, sorted by their distance with respect to $q$, and $P_q^k$ is the list returned by the algorithm.

Efficiency measures:

- index time.
- index size (RAM, disk).
- number of candidates retrieved from disk ($z'$).
- average search time.

---

Outline

1 Introduction

2 The PP-Index

3 Experiments
   - The CoPhIR collection
   - Evaluation measures
   - Results

4 Demo

5 Conclusions
Experiments

Results

Figure 14: Indexing time w.r.t. to the size of $R$ and the data set size.

| $|D|$ | indexing time (sec) | prefix tree size | data storage | $l'$ |
|-----|-------------------|-----------------|--------------|------|
|     |                   | full            | comp.        |      |
| 1M  | 419               | 7.7 MB          | 91 kB        | 349 MB | 2.1 |
| 10M | 4385              | 53.8 MB         | 848 kB       | 3.4 GB | 2.7 |
| 100M| 45664             | 354.5 MB        | 6.5 MB       | 34 GB  | 3.5 |

Table 2: Indexing times (with $|R| = 100$), resulting index sizes, and average prefix tree depth $l'$ (after prefix tree compression with $z = 1,000$), for the various data set sizes.
Results

Figure 15: Search time w.r.t. to the size of $R$ and the data set size. Search performed with $z = 1,000$ and $k = 100$ (single index, single query).

Table 3: Average $z'$ value ($z = 1,000$), i.e., average number of retrieved candidate objects for a query, with respect to the size of the reference objects set and data set size.
Results

Figure 16: Effectiveness with respect of the size of $R$ set, on various index sizes, using $k = 100$, and $z = 1,000$ (single index, single query).

Figure 17: Effectiveness with respect of the size of $R$ set, on the 100M index, using $z = 1,000$ (single index, single query).
Figure 18: Effectiveness of the multiple index search strategy on the 100M index, using $|R| = 1,000$ and $z = 1,000$.

Figure 19: Effectiveness of the multiple query search strategy on the 100M index, using $|R| = 1,000$ and $z = 1,000$. 
Figure 20: Effectiveness of the combined multiple query and multiple index search strategies, using eight queries and eight indexes, on various data set sizes, using $|R| = 100$, and $z = 1,000$.

Figure 21: Effectiveness of the combined multiple query and multiple index search strategies, using eight queries and eight indexes, on various data set sizes, using $|R| = 1,000$, and $z = 1,000$. 
Outline

1 Introduction
2 The PP-Index
3 Experiments
4 Demo
5 Conclusions
Demo

MiPai

similarity search engine

http://mipai.esuli.it
Outline

1 Introduction
2 The PP-Index
3 Experiments
4 Demo
5 Conclusions
Outline

1. Introduction
2. The PP-Index
3. Experiments
4. Demo
5. Conclusions
   - Summary
   - Questions
The PP-Index:

- is a simple but effective data structure for approximate similarity search.
- scales well, both at indexing time and at search time.
- can be kept updated with minor additional effort.
- has good parallelization properties.
- relates well with other data structures (i.e., inverted lists).

There is a lot still to investigate:

- policies for reference points selection.
- studying the relations between $l$, $|R|$, $z$, $k$, $l'$, and $z'$.
- giving a theoretical foundation to the permutation based methods.
- applicability to other domains and similarity space types.
- policies for data partitioning.
Outline

1. Introduction
2. The PP-Index
3. Experiments
4. Demo
5. Conclusions
   - Summary
   - Questions
Questions?
Q: How does the PP-Index differ from the “orthodox” metric approach $X$?

A: Please help yourself:

- No “explicit” requirement of metric properties.
- Use of a predetermined (i.e., fixed) set of reference points.
- Any reference point has a “global” influence.
- Different data access model.
- Different data update model.
- ...
Q: What are the key differences between the permutation-based methods and the LSH-based methods?

A: The permutation-based methods are mostly based on geometrical considerations, while LSH-based methods are mostly based on probabilistic considerations.

The permutation-based methods are able to take into account how data is distributed in the similarity space (by means of $R$), while the LSH hash functions are derived only from the distance function.

Each element of the hash key generated by an LSH hash function is independent from the others, while the order relation between the elements of a permutation is the crucial information for a permutation-based method.