Extremely low bit-rate nearest neighbor search using a Set Compression Tree

Relja Arandjelović and Andrew Zisserman

Department of Engineering Science, University of Oxford
Introduction

Many computer vision / machine learning systems rely on Approximate Nearest Neighbor (ANN) search:

- Large scale image retrieval: find NNs for each local descriptor of the query image (e.g. SIFT, CONGAS)
- Large scale image retrieval: find NN for the global descriptor of the query image (e.g. GIST, VLAD)
- 3-D reconstruction: match local descriptors
- KNN classification

...
Brief ANN overview

Predominant strategy for ANN search:

- Partition the vector space
  - clustering
  - hashing
  - k-d tree
Brief ANN overview

Predominant strategy for ANN search:

- Partition the vector space
  - clustering
  - hashing
  - k-d tree

- Create an inverted index
Brief ANN overview

Given the query vector
1. Assign it to the nearest partition (typically to more than 1)
2. Do a brute force linear search within the partition
Brief ANN overview

- Positive: Fast as it skips most of the database vectors
- Negative: All database vectors need to be stored in RAM:
  - For example, 1 million images x 1k descriptors each x 128 bytes for SIFT = 128 GB of RAM
- Plausible only if descriptors are compressed
  - E.g. use Product Quantization and 8 bytes per descriptor => only 8 GB RAM required
Objective

- Improve compression quality
- For ANN search:
  - Compress posting lists
- Not specific to ANN search - we consider general vector compression
Motivating example

- 400 2-D points generated from a GMM with 16 components
- We have only 4 bits per descriptor available
- How can we best compress the data?
Motivating example

- First idea:
  - Use k-means to find 16 clusters
  - Represent each vector with the 4-bit ID of the nearest cluster

- Equivalent to state-of-the-art vector compression - product quantization (PQ):
  - Same at low bitrates
  - PQ approximates this at high bitrates
Motivating example

● First idea:
  ○ Use k-means to find 16 clusters
  ○ Represent each vector with the 4-bit ID of the nearest cluster

● Equivalent to state-of-the-art vector compression - product quantization (PQ):
  ○ Same at low bitrates
  ○ PQ approximates this at high bitrates
Motivating example

- Can we do any better?
  - 4 bits per vector is very small, large quantization errors are fully understandable and expected
  - 4 bits per vector means the vector space is divided into 16 regions - any division of the space is bound to have large quantization errors
Motivating example

- Set Compression Tree (SCT)
Motivating example

- Set Compression Tree (SCT)
  - 4 bits per vector means the vector space is divided into 16 regions - any division of the space is bound to have large quantization errors
Motivating example

- Set Compression Tree (SCT)
  - 4 bits per vector means the vector space is divided into 16 regions - any division of the space is bound to have large quantization errors
Motivating example

- Set Compression Tree (SCT)
  - 4 bits per vector means the vector space is divided into 16 regions, **only if vectors are compressed individually**
  - Much better compression is achievable if we compress the entire set **jointly**
Set Compression Tree (SCT): Overview

- **Key idea:** Compress all vectors in a set jointly
- The set of vectors is represented using a binary tree:
  - Each node corresponds to one axis-aligned box ("bounding space", "cell")
  - Each leaf node corresponds to exactly one vector from the set
  - **All** that is stored is the encoding of the tree
  - Decoding the tree reconstructs all the leaf nodes exactly
  - Vectors are reconstructed as centres of leaf cells
Constructing the SCT

1. Start from a cell which spans the entire vector space
Constructing the SCT

1. Start from a cell which spans the entire vector space
2. Split the cell into two disjunct child cells
   ○ Different from k-d tree as the split has to be independent of the data inside the cell, as otherwise one would need to store the split dimension and position (huge increase in bitrate)
   ○ For example, splitting strategy:
     i. Find the longest edge
     ii. Split it into half
Constructing the SCT

1. Start from a cell which spans the entire vector space
2. Split the cell into two disjunct child cells
3. Record the "outcome" of the split

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
<th>Number in child 1</th>
<th>Number in child 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0000</td>
<td>= 0</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>B</td>
<td>0001</td>
<td>&gt; 1</td>
<td>= 0</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>D</td>
<td>0010</td>
<td>&gt; 1</td>
<td>= 1</td>
</tr>
<tr>
<td>E</td>
<td>0011</td>
<td>= 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>= 1</td>
<td>= 1</td>
</tr>
</tbody>
</table>

Current tree encoding: C
Set tree encoding: 01
Constructing the SCT

1. Start from a cell which spans the entire vector space
2. Split the cell into two disjunct child cells
3. Record the "outcome" of the split
4. Find a cell (depth first) which contains >1 vector, go to step (2)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
<th>Number in child 1</th>
<th>Number in child 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0000</td>
<td>= 0</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>B</td>
<td>0001</td>
<td>&gt; 1</td>
<td>= 0</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>D</td>
<td>0010</td>
<td>&gt; 1</td>
<td>= 1</td>
</tr>
<tr>
<td>E</td>
<td>0011</td>
<td>= 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>= 1</td>
<td>= 1</td>
</tr>
</tbody>
</table>

Current tree encoding: C
Set tree encoding: 01
Constructing the SCT

1. Start from a cell which spans the entire vector space
2. Split the cell into two disjunct child cells
3. Record the "outcome" of the split
4. Find a cell (depth first) which contains >1 vector, go to step (2)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
<th>Number in child 1</th>
<th>Number in child 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0000</td>
<td>= 0</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>B</td>
<td>0001</td>
<td>&gt; 1</td>
<td>= 0</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>D</td>
<td>0010</td>
<td>&gt; 1</td>
<td>= 1</td>
</tr>
<tr>
<td>E</td>
<td>0011</td>
<td>= 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>= 1</td>
<td>= 1</td>
</tr>
</tbody>
</table>

Current tree encoding: CC
Set tree encoding: 01 01
Constructing the SCT

1. Start from a cell which spans the entire vector space
2. Split the cell into two disjunct child cells
3. Record the "outcome" of the split
4. Find a cell (depth first) which contains >1 vector, go to step (2)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
<th>Number in child 1</th>
<th>Number in child 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0000</td>
<td>= 0</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>B</td>
<td>0001</td>
<td>&gt; 1</td>
<td>= 0</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>D</td>
<td>0010</td>
<td>&gt; 1</td>
<td>= 1</td>
</tr>
<tr>
<td>E</td>
<td>0011</td>
<td>= 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>= 1</td>
<td>= 1</td>
</tr>
</tbody>
</table>

Current tree encoding: CCF
Set tree encoding: 01 01 1
Constructing the SCT

- **All** that is recorded is the sequence of split outcomes
- No information is encoded on a per-vector basis

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
<th>Number in child 1</th>
<th>Number in child 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0000</td>
<td>= 0</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>B</td>
<td>0001</td>
<td>&gt; 1</td>
<td>= 0</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>D</td>
<td>0010</td>
<td>&gt; 1</td>
<td>= 1</td>
</tr>
<tr>
<td>E</td>
<td>0011</td>
<td>= 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>= 1</td>
<td>= 1</td>
</tr>
</tbody>
</table>

Final tree encoding: CCFAFDF
Set tree encoding: 01 01 1 0000 1 0010 1
Bitrate: 15/7 = 2.14 bits per vector
Decoding the SCT

1. Start from a cell which spans the entire vector space
2. Split the cell into two disjunct child cells
3. Read the "outcome" of the split
4. Find a cell (depth first) which contains >1 vector, go to step (2)

Final tree encoding: CCFADFDF

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
<th>Number in child 1</th>
<th>Number in child 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0000</td>
<td>= 0</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>B</td>
<td>0001</td>
<td>&gt; 1</td>
<td>= 0</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>D</td>
<td>0010</td>
<td>&gt; 1</td>
<td>= 1</td>
</tr>
<tr>
<td>E</td>
<td>0011</td>
<td>= 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>= 1</td>
<td>= 1</td>
</tr>
</tbody>
</table>
Decoding the SCT

1. Start from a cell which spans the entire vector space
2. Split the cell into two disjunct child cells
3. Read the "outcome" of the split
4. Find a cell (depth first) which contains >1 vector, go to step (2)

Final tree encoding: CCFAFDF

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
<th>Number in child 1</th>
<th>Number in child 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0000</td>
<td>= 0</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>B</td>
<td>0001</td>
<td>&gt; 1</td>
<td>= 0</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>D</td>
<td>0010</td>
<td>&gt; 1</td>
<td>= 1</td>
</tr>
<tr>
<td>E</td>
<td>0011</td>
<td>= 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>= 1</td>
<td>= 1</td>
</tr>
</tbody>
</table>
Decoding the SCT

1. Start from a cell which spans the entire vector space
2. Split the cell into two disjunct child cells
3. Read the "outcome" of the split
4. Find a cell (depth first) which contains >1 vector, go to step (2)

Final tree encoding: CCFAFDF

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
<th>Number in child 1</th>
<th>Number in child 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0000</td>
<td>= 0</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>B</td>
<td>0001</td>
<td>&gt; 1</td>
<td>= 0</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>D</td>
<td>0010</td>
<td>&gt; 1</td>
<td>= 1</td>
</tr>
<tr>
<td>E</td>
<td>0011</td>
<td>= 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>= 1</td>
<td>= 1</td>
</tr>
</tbody>
</table>
Decoding the SCT

1. Start from a cell which spans the entire vector space
2. Split the cell into two disjunct child cells
3. Read the "outcome" of the split
4. Find a cell (depth first) which contains >1 vector, go to step (2)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
<th>Number in child 1</th>
<th>Number in child 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0000</td>
<td>= 0</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>B</td>
<td>0001</td>
<td>&gt; 1</td>
<td>= 0</td>
</tr>
<tr>
<td>C</td>
<td>01</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>D</td>
<td>0010</td>
<td>&gt; 1</td>
<td>= 1</td>
</tr>
<tr>
<td>E</td>
<td>0011</td>
<td>= 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>= 1</td>
<td>= 1</td>
</tr>
</tbody>
</table>
Remarks

- Bitrate: 2.14 bits per vector
- First split, encoded with 2 bits, halves the positional uncertainty for all 7 vectors
  - If vectors were encoded individually this would cost 1 bit per vector (half of our bitrate!)
  - We use only 2 bits for 7 vector, so 0.29 bits per vector
Brute force NN search

- Simply decompress the tree and compare the reconstructed vectors with the query vector
- Memory efficient (negligible overhead for decompression):
  - The tree is traversed (while being decoded) depth-first so only information about a single cell (+ some small bookkeeping) is maintained at any point in time.
  - Vectors are decoded one at the time
- 1 million 32-D vectors on a single core 2.66 GHz laptop (not fully optimized!):
  - Compression $O(N \log N)$: 14 seconds
  - Search $O(N)$: 0.5 seconds
Implementation: Split choice

- Has to be independent of the data in the cell
- Splitting dimension:
  - Pick the longest edge of the cell
  - Minimizes the expected approximation error
- Split position:
  - Could pick the midpoint
  - Aim for balancing the tree: We pick the median value of the training data in the splitting dimension, clipped by the cell
Implementation: Optimal binary encoding

- The 6 split outcomes are encoded using variable length binary codes.
- Tree is encoded in two stages:
  a. Use a default encoding when constructing the tree, while keeping occurrence counts for each of the 6 outcomes.
  b. Use Huffman coding to obtain the optimal codes.
  c. Re-encode the tree.
     - Simple to do - just translating the codes from the default to optimal ones.
- Storing the Huffman tree requires only 18 bits - this is usually worth the savings.
Implementation: Finer representation

- It is simple to obtain a finer representation of the vectors by increasing the bitrate:
  - Split the leaf cell with a rate of 1 bit per split, encoding which side of the split the vector is
  - We bias the additional splitting towards large cells (i.e. vectors which have been represented poorly)
- There is scope for improving this, e.g. use product quantization for the residual
Implementation: Dimensionality reduction

- SCT is not appropriate to use (without refinement) when the vector dimensionality is large compared to \( \log_2 N \), where \( N \) is the number of vectors
- For example:
  - \( N=1\text{M} \Rightarrow \text{Expected tree depth } \sim \log_2 (1\text{M})=20 \)
  - At most \( \sim 20 \) dimensions will be split to obtain the final cell
  - Trying to compress 128-D SIFT descriptors would mean that \( \sim 100 \) dimensions would not be split at all
- Important to do PCA before compression, or enough refinement
- Also perform random rotation to balance the variance, like [1]

Obtaining the vector ID

- Decompressing the SCT permutes input vectors according to the depth-first traversal of the tree
- Do we need to store the permutation? Huge storage cost!
Obtaining the vector ID

No!

1. Linear traversal over the entire vector database:
   ○ Example: searching global image descriptors (GIST, VLAD..)
   ○ Returning the 3rd descriptor as the NN means we should return the 3rd image to the user
   ○ What does "3rd image" mean? Usually the order is arbitrary:
     i. 3rd image usually means: look up the 3rd row in a table with meta-data, which contains image title, url, etc
     ii. Nothing stops us from permuting the rows of the table
Obtaining the vector ID

2. ANN search:
   - The order of items in the posting list doesn't matter at all
   - We can safely permute it as long as we don't break the (vector, imageID) pairs
Properties of SCT: Unique description

- Each cell contains exactly one vector
  - Even in areas of large density, one can discriminate between vectors
  - No competing method can do this
- Methods which compress vectors individually can't possibly perform well at low bitrates:
  - 1M vectors, 10 bits per vector
  - On average: 1k vectors are indistinguishable from each other
  - NN search is bound to fail
  - SCT provides a unique description with less than 5 bits per vector
Properties of SCT: Asymmetric search

- [1] Better not to quantize query vectors as this obviously discards information
- SCT is asymmetric in nature as query vectors are compared directly to the reconstructed database vectors

Evaluation datasets

1. 1M SIFT descriptors (SIFT1M) [2]
   - 1M SIFT descriptors: 128-D vectors
   - Dataset division:
     - 10k query descriptors
     - 100k training descriptors
     - 1M database descriptors
   - Evaluation metric:
     - Average recall of the first NN at R retrievals (usually R=100)
     - i.e. the proportion of query vectors for which the NN is ranked within the first R retrievals

Evaluation datasets

2. 580k Tiny Images (Tiny580k) [3], a subset of 80M Tiny Images
   ○ 580k GIST descriptors: 384-D vectors
   ○ 5 random splits into:
     ■ 1k query descriptors
     ■ 579k database (same as training) descriptors
   ○ Evaluation metrics:
     ■ mAP-50NN: mean average precision where positives are 50 nearest neighbors for each query descriptor
     ■ mAP-thres: mean average precision where positives are all descriptors within a distance D from the query (where D = average distance to the 50th NN)

Baselines

- **Product Quantization (PQ)**: Each vector is split into $m$ parts, each part is vector quantized independently using $k$ clusters, $m \log_2(k)$ bits per vector.
- **Locality Sensitive Hashing (LSH)**: Code = signs of random projections with random offsets.
- **Shift Invariant Kernels (SIK)**: Code = signs of random Fourier features with random offsets.
- **PCA with random rotation (RR)**: Vector is projected onto $D$ principal components, followed by a random rotation. Code = sign of final values.
- **Iterative quantization (ITQ)**: Start from RR method and then iteratively find the rotation which minimizes quantization errors.
- **Spectral Hashing (SH)**: Coding scheme obtained deterministically by trying to ensure similar training descriptors get hashed to similar binary codes.
Results

- SIFT1M
- Tiny580k
Results: Discussion

- SCT outperforms all competing methods using all evaluation metrics
- SIFT 1M, recall@100:
  - SCT with 4.97 bits achieves 0.344
  - Second best (PQ) at 6 bits achieves 0.005, i.e. 69 times worse
  - Even at 16 bits (3.2 times larger), PQ only reaches 55% of SCT at 4.97 bits
- Poor performance of competing methods at very low bitrates is expected, see "Unique description"
Dimensionality reduction

- At extremely low bitrates performance drops with increasing number of principal components (PCs) due to increasing approximation error.
- Increasing the bitrate makes it possible to use more PCs and thus represent the underlying data more accurately.
- For a given #PCs, the upper bound is reached quickly:
  - For 16 PCs it is reached at 32 bits, so 2 bits per dimension for a value that would commonly be represented with 32 or 64 bits.
"Small" dimensionality requirement revisited

- As mentioned earlier, SCT is not appropriate to use without refinement when the dimensionality is larger than \( \sim \log_2 N \)
- However:
  - We demonstrated state-of-the-art performance with dimensionality reduction for 128-D and 384-D descriptors
  - "Small" descriptors are often used: PCA'd VLAD and SIFT are commonly used, CONGAS are 40-D, Simonyan et al. achieve good performance with 60-D learnt local descriptors
- All baselines (apart from PQ) perform dimensionality reduction:
  - RR and ITQ start from PCA (bitrate = # PCs)
  - LSH and SIK use random projections (bitrate = # projections)
  - SH learns projections from training data (bitrate = # projections)
Qualitative results (Tiny580k)
Conclusions

- Large improvement in compression rates by compressing a set of vectors jointly, instead of each vector individually
- Set Compression Tree (SCT) sets the state-of-the-art
  - Hugely outperforms all competing methods at extremely low bitrates
  - Dominates at high bitrates too; there is scope for improvement
- Only tool of choice for extremely low bitrates
- All vectors are uniquely represented, even in high-density areas