Large Scale Object Retrieval

Conceptual overview

(with too much detail)

Relja Arandjelović

Overview

1. Prerequisites
2. Description of files
   - Compression
3. Retrieval procedure
   - Spatial reranking
4. Interfaces
   - API
   - Graphical user interface (GUI)
   - Connection with Visor
5. Indexing procedure, parallelization

Note: My library is named relja_retrieval (short: rr), so when I mention details of my library I will say "in rr: .."

Prerequisites

- To understand this presentation, one needs to have some basic knowledge of object retrieval:
  - Knowledge of the basic idea: local descriptors quantized into a large number of visual words yielding sparse BoW vectors
  - The sparsity of BoW vectors is exploited by an inverted index, which stores a posting list for every word, i.e. given a word ID returns a list of images which contain the same wordID and some extra information (e.g. term frequency)
  - Basic idea what is spatial reranking, query expansion, soft assignment

Overview

1. Prerequisites
2. Description of files
   - Compression
3. Retrieval procedure
   - Spatial reranking
4. Interfaces
   - API
   - Graphical user interface (GUI)
   - Connection with Visor
5. Indexing procedure, parallelization

Note: My library is named relja_retrieval (short: rr), so when I mention details of my library I will say “in rr: ..”

Description of files

- Here, I describe the basic file format I use for many things, such as the inverted index
- The backbone is the database file (db)
  - Given ID gets binary data
  - Explained in more detail on the next slide
- Many files can be cast as this simple db if interpretation of the binary data is varied
  - An index decodes “data” as list of entries, e.g. ID -> list of entries
  - An inverted index is an index where list of entries is interpreted as a posting list

Description of files (cont’d)

- Simple database (db):
  - Provides functionality for quickly getting data given an integer ID (for inverted index: wordID, for forward index: imageID)
  - In rr: proto_db, “data” is a string, i.e. a list of bytes which should be interpreted in an application-specific way. I use Google’s protocol buffers (protobuf) which naturally provide encoding/decoding (serialization/deserialization) of defined data structures to/from a string
  - Importantly: has to be thread-safe for fast retrieval (multiple threads for one query, multiple queries at the same time); no locking is used as this would make it too slow (locking/unlocking would be a bottleneck)
  - Apart from inverted/forward index, useful in many places, e.g. I use it to store ground truth data, with ID=query number, and data= {query name, query image, query ROI, positives, ignores}

Description of files (cont’d)

- Index (idx)
  - Uses a “simple database (db)” (in rr: proto_db) to provide functionality: get indexEntries (in rr: index_entry.proto) for a given ID. The “indexEntries” contains {list of IDs, list of geometry information, list of weights, extra data (list of bytes)}; explained best in the next slide

Description of files (cont’d)

- Inverted index (iidx)
    - Enables fast ranking
        - Basic functionality: retrieve posting lists a given wordID
    - iidx is an idx where ID= wordID and the “indexEntries” field contains the corresponding posting list. So, given an ID we get a string (list of bytes) from the db which gets decoded as indexEntries which is a posting list.
    - indexEntries data structure contains:
        - list of IDs: i.e list of imageIDs which contain the given wordID
        - (if present) list of geometry (ellipse specs) associated with every feature in the postinglist
        - (if present) weights/counts, i.e. in a pure tf-idf based retrieval this would be term-frequency
        - (if present) extra data: e.g. Hamming embedding signatures for all posting list features

Description of files (cont’d)

- Forward index (fidx)
  - Enables search using an internal image (therefore also needed for query expansion, and some implementations of spatial reranking)
    - Basic functionality: get a list of features/visual words for a given imageID
  - fidx is an idx where ID= imageID and the “indexEntries” field contains information about features in the image. So, given an ID we get a string (list of bytes) from the db which gets decoded as indexEntries.
  - indexEntries data structure contains:
    - list of IDs: i.e list of wordIDs which contain the given wordID
    - (if present) list of geometry (ellipse specs) associated with every feature in the image

Description of files (cont'd)

- Implementation details / Index compression:
    - It is useful to sort indexEntries by the "list of IDs" field – enables compression, and makes life easier / retrieval faster (see spatial reranking, document at the time (DAAT) traversal, ..)
    - Compression:
        - Geometry (ellipse) compression motivated by "Efficient representation of local geometry for large scale object retrieval". Perdoch et al CVPR 2009. I.e. we round ellipse centre (x,y) and store as integers* (instead floats), and compress ellipse geometry parameters (raw format: 3 floats) by quantizing scale, major axis angle and major/minor axis length ratio into 1 byte each (so 3*4 bytes -> 3 bytes). Unlike [Perdoch09], I don't learn the quantization with k-means but uniformly quantize (angle: uniform, scale and length ratios: uniform in log space)

* Actually the integers are stored even more efficiently as we know that (x,y) are generally smallish and don't use the entire dynamic range of all possible integers, see next slide which explains a similar thing for differential encoding and Protocol Buffers

Description of files (cont'd)

- Index compression (cont'd)
  - Can use differential coding to encode a sorted list of IDs, i.e. store the first ID without a change, and replace every following ID_i with ID_i – ID_(i-1), guaranteed to be non-negative as ID's are sorted. For example, [17, 1023, 2000, 5000, 11000, 16000] -> [17, 1007, 993, 3000, 6000, 5000].
  - The transformed list generally contains much smaller numbers than the original one. For an iidx example, for a 1M image database with 1k features per image and 100k visual vocabulary, posting list length is 10k on average. This means that the average ID_i-ID_(i-1) is 1M/10k=100 (i.e. need only 7 bits to encode), while one would need 20 bits if no transformation is done (as each ID is between 0 and 1M). This is obviously only a rough sketch of the idea, but should be clear enough.
  - Google's protocol buffers can be made (effortlessly) to automatically compress integers which are expected to be relatively small but could sometimes be large (i.e. exactly what we need)

Description of files (cont'd)

- Other files
  - Fairly straight forward stuff not worth going into details about, such as:
    - imageInfo: db which provides imageID -> file name, image width, image height – useful for GUI when displaying images on a web page etc. Could potentially be stored in forward index together with everything else, but to access this simple information would then require reading much more information (i.e. all feature data for an imageID), which would slow things down (e.g. imagine showing 100 images on a web page and reading all features for 100 images; would still work but would be quite wasteful)
    - Files which store visual word cluster centres, idf values, BoW L2 norms (as it is better to store unnormalized BoWs in the index, i.e. instead of storing tf/L2norm with 4 bytes per feature, store tf as integers with less than 1 byte/feature and then divide by the L2norm at ranking time), Hamming Embedding data (projection directions, median values), etc.

Description of files (cont’d)

- Tiny implementation detail:
  - Google’s protobuf’s are limited to ~60 MB (well they can be forced to be larger but are optimized for below 60 MB), so my proto_db implementation actually doesn’t return a single data string, but an array of strings. Therefore also proto_index doesn’t return a single indexEntries but an array of them, e.g. iidx instead of returning a single posting list (one indexEntries) for a given wordID, it chops up the posting list into several chunks and returns an array of those chunks.

Overview

Note: My library is named relja_retrieval (short: rr), so when I mention details of my library I will say “in rr: ..”

Retrieval procedure

- Given the provided fidx / iidx infrastructure, it is simple to do retrieval
- The query representation is again in the indexEntries format {list of wordIDs, list of geometry information, list of weights, extra data (list of bytes)}
- For BoW retrieval: query will not contain extra data, weights exist if soft assignment is performed
- For Hamming Embedding (HE): weights don’t exist, extra_data contains the list of Hamming Embedding signatures for the query features
- For every wordID, use the iidx to retrieve the relevant posting list, and follow the description of the retrieval method (e.g. for BoW just compute dot products, for HE compute hamming distances, weight by Gaussian, do burstiness.. but all are quite easy to do)
  - Note, for HE implementation details, look at Jegou’s CVPR 09 burstiness paper, as this contains the Guassian weighting while the original HE paper doesn’t

Retrieval procedure (cont'd)

- Complication with HE:
  - For internal image query (useful in a GUI, or for query expansion) we need the query HE signatures.
    - A naive solution: keep them in the fidx. However – HE signatures also need to be in the iidx for fast ranking. So if we keep them in fidx as well we would almost double RAM requirements
    - A good but a bit more complex solution: obtain them from the iidx
      - Get the wordIDs for the given internal query imageID (from fidx)
      - Query the iidx for the wordIDs and search for the entries corresponding to the given imageID. Note that entries are sorted by imageID anyway (for compression etc) so can do binary search for this
      - Copy the relevant HE signatures from the iidx into the query representation

Spatial reranking

- Implementation depends on the choice of data organization:
    1. Geometry information is stored in the fidx
    2. Geometry information is stored in the iidx

- Both options have the same backbone – given putative matches (i.e. features which have the same wordID and, in the case of HE, their binary signatures are similar enough) compute the number of inliers using RANSAC
- Easily parallelized (see parallelization description under “indexing procedure” later) – master tells workers which images to verify, workers verify and send inlier counts back to the master which performs the final re-sorting

Spatial reranking (cont'd)

- Implementation depends on the choice of data organization:
  1. If geometry information is stored in the fidx, then for every image to verify – read data from fidx, perform spatial verification
     - Made faster by storing entries in fidx(imageID) sorted by wordID (we already do it to aid compression) as putative matches can be found fast (look up "intersection of two sorted arrays")

Spatial reranking (cont'd)

- Implementation depends on the choice of data organization:

  2. If geometry information is stored in the iidx (e.g. for "weak geometric consistency ranking" geometry is needed in the iidx) then we do DAAT traversal of the iidx (see "Size Matters: Exhaustive Geometric Verification for Image Retrieval" by Stewenius et al.) and perform spatial verification on the pre-selected images
     - Putative matches automatically fall out of the DAAT traversal
     - Less data is visited than for option (1.), i.e. it is faster:
       - Exactly the same iidx entries are visited anyway at the ranking stage (so don't need to read/decompress them twice)
       - (1.) needs the same iidx entires at ranking, and on top of that it needs all fidx entries for each image to verify. Many entries are irrelevant as putative matches make a small proportion of all image features, so a lot of data reads are useless
     - In a way it is simpler for HE retrieval as otherwise in (1.) one would need to do hamming distance computation twice (once when ranking images before spatial reranking, and once during spatial reranking). This way I compute it once at ranking, and store the match/no-match decisions for the spatial reranking stage

Spatial reranking (cont'd)

- Implementation depends on the choice of data organization:
  2. If geometry information is stored in the iidx (cont'd)
     - Like for HE retrieval, for internal image queries we need to fetch geometry information from the iidx for the query features
       - This is also needed even without spatial reranking if ROI queries are allowed (i.e. to know which query image feature is in the query ROI)

- Summary: though it is more complex to implement, storing geometry in the iidx is my method of choice as it is faster than option (1.) and compatible with HE, WGC, but also still compatible with BoW
  - I do have both methods implemented in my rr library, but I didn't implement option (1.) in combination with HE

Overview

1. Prerequisites
2. Description of files
   - Compression
3. Retrieval procedure
   - Spatial reranking
4. Interfaces
   - API
   - Graphical user interface (GUI)
   - Connection with Visor
5. Indexing procedure, parallelization

Note: My library is named relja_retrieval (short: rr), so when I mention details of my library I will say “in rr: ..”

Interfaces

- API:
  - Fairly simple to implement – have the program listen to a port and answer to requests such as “compute features for this image” or “return a ranked list given these features”, etc. Care needs to be taken that everything works correctly with multiple requests at the same time (e.g. inverted index access needs to be thread-safe)
  - The calls are blocking (a.k.a. synchronous), i.e. something requests an action and stays while response is given. Note, this doesn’t mean that the backend is blocked and cannot answer multiple queries at the same time, it is completely unrelated.

Interfaces

- Graphical User Interface (GUI):
  - Fairly simple to implement given access to the API
  - The web server (in rr: python-cherrypy) gets urls such as dosearch?docID=774&xl=541&yl=73&xu=696&yu=229
    and it converts these into a function call to dosearch(docID, xl, yl, xu, yu) with the argument values passed to it
  - Each function (implemented in python) then communicates with the API to request an action, obtains the results and displays them in the html
  - Note: I consistently use relative URL’s as this simplifies the setup for public demos (i.e. the ReverseProxy configuration on Zeus is then completely straightforward)

Interfaces

- VISOR / AXES:
  - These assume a somewhat different backend API functionality
  - The requests are asynchronous (i.e. non-blocking), so a request is made (e.g. query the database), the requestor goes away and does other things, and later asks – is the request done?
  - The backend needs somewhat more bookkeeping than for my previously descriebed API, i.e. to keep queryIDs, intermediate results, ensure locking is done properly (e.g. need to handle the case when the query just finished and results are to be saved in a variable at the same time as a request is made which wants to read the same variable)
  - On the other hand, the API provides more functionality (multi-image queries, editing the training data while not redoing everything, etc)
  - Implementation: I made a (python) wrapper around my API which acts as VISOR-API, therefore not duplicating backend code, and keeping my API simple (good enough for many demos, for research/debug, and much easier to use)

Overview

1. Prerequisites
2. Description of files
   - Compression
3. Retrieval procedure
   - Spatial reranking
4. Interfaces
   - API
   - Graphical user interface (GUI)
   - Connection with Visor
5. Indexing procedure, parallelization

Note: My library is named relja_retrieval (short: rr), so when I mention details of my library I will say “in rr: ..”

Indexing procedure

- Given a list of images to index, we want to automatically create all files needed for retrieval (fidx, iidx) as well as other files if needed (visual vocabulary, Hamming Embedding, etc)
- Needs to be very parallelized in order to index millions of images using a cluster

Parallelization (on cluster)

- One option is to split data (e.g. the entire image database into smaller sets) and let each computational node process one chunk. Two problems:
  - Intermediate data needs to be stored (potentially wasteful), and the chunks still need to be combined in some way
  - The splits are predetermined and therefore it often happens that a node finishes much earlier than another one. It often happens because certain nodes can be slower (or under heavy load from other jobs), or parts of the data can take longer to process (e.g. a chunk could have relatively high-resolution images compared to the mean, therefore more features get extracted and everything takes longer)

Parallelization (on cluster) (cont'd)

- Solution – slightly more complex system (but not too much) where there is communication across nodes:
  - There are many worker nodes, and a single master node
  - The worker nodes receive work from the master, do some processing and send a result back to the master
  - The master node takes care of a few things:
    - Schedules work to workers, e.g. given a list of images to index, it would go through the list, give one image to every worker, wait until a worker is free, send it a new job. Thus if one node is faster than the other it will receive more jobs and the entire processing task will finish faster than the static splits from the previous slide.
    - Takes care of storing the final results, e.g. when making a fidx the master will be in charged of creating the proto_index file and writing workers' results to it. This removes common problems that occur with parallel programs, such as workers simultaneously writing to the same file thus overwriting each other's work

Parallelization (on cluster) (cont’d)

- Comments on the master-workers solution:
  - Downsides:
    - All nodes need to be available simultaneously, a simple split-into-chunks strategy doesn’t need this
    - If one node fails, everything fails. Unfortunately this is the case with the parallelization system I use (OpenMPI) and cannot be fixed. The only alternative seems to be MapReduce but I’m not sure how suited this is for our needs (i.e. can it run on grid engine, would probably require quite a bit of setting up, needs its own way of storing files, takes some time to learn, etc). Not actually a big problem as the only time something fails on our cluster is when the code is buggy, so it needs to be fixed anyway, or when disk space runs out, and there is no fixing to this whichever parallization one uses
  - With a small number of nodes, using 1 node for master could be wasteful (e.g. 3 workers and 1 idle-ish master node is significantly slower than 4 workers) – my code supports a hybrid solution, where the master is also a worker (i.e. 4 workers with one of them performing master duties after each job). Only recommended if the number of nodes is quite small.

Parallelization (on cluster) (cont'd)

- Comments on the master-workers solution:
  - Can handle both types of parallelization:
    - Working on a cluster with many nodes which don't share memory – implemented using OpenMPI (message passing interface), nodes send messages to each other (master <-> worker)
    - Working on a single machine with many CPUs which share memory – workers are then implemented as threads, there is no need for explicitly sending messages as memory is shared so a thread only needs to store data at a predefined location in RAM (pointer)
    - The two setups require a slightly different configuration, e.g. if workers need to access a large amount or memory (e.g. one is computing an image graph, so every worker needs the iidx which is potentially large) – in a shared memory system only one instance of iidx is needed (given it is implemented to be thread-safe, as it is in rr), while on a cluster each worker needs to load the iidx.

Indexing - Overview

- Indexing stages:
  1. Compute a subsample of descriptors for training
  2. Compute the visual vocabulary
  3. Compute Hamming Embedding parameters
  4. Compute iidx, fidx, imageInfo

Indexing - Overview

- Indexing stages:
  1. Compute a subsample of descriptors for training
  2. Compute the visual vocabulary
  3. Compute Hamming Embedding parameters
  4. Compute iidx, fidx, imageInfo

Indexing – 1. Descriptor subsample

- Master sends each worker a randomly sampled image, waits for workers to extract features and send them back, saves the features. It stops sending work once the requested number of training features is obtained.

Indexing – 1. Descriptor subsample (cont’d)

- Implementation details:
  - We want this step to be repeatable, would be bad if we can’t replicate our own results if some files are lost/deleted, but also useful for debugging (e.g. did my recent code change break something?)
    - Clearly need to use the same random seed for the random sampling of images, however – that’s not all
    - Workers can finish processing jobs at different times across different runs e.g. the first image could be finished after a longer time if it is executed on a slower node, and therefore some other node might finish processing another image earlier. This would cause features to be saved in different order, this then changes the outcome of visual word clustering (as initialization is different). Solution – ensure the master remembers the order in which it should save results (i.e. don’t just save features as they come back from any worker, but do some bookkeeping to keep the order)

Indexing – 1. Descriptor subsample (cont’d)

- Implementation details:
  - RootSIFT: SIFT is commonly stored as 128 bytes, as each element of the 128-D vector is stored as an integer 0-255 by all the standard SIFT implementations. RootSIFT however has floating point values for all 128 dimensions (though I’m sure some sort of simple uniform scalar quantization would work well), and therefore occupies 4x the space of SIFT on disk. So, to avoid wasting space and RAM (and this matters given I usually sample 15-20M descriptors in this stage, so we are taking about a difference between 2.5 GB and 10 GB), if RootSIFT is used I still only store original SIFT descriptors, and convert them on the fly in the following indexing stages.

Indexing - Overview

- Indexing stages:
  1. Compute a subsample of descriptors for training
  2. Compute the visual vocabulary
  3. Compute Hamming Embedding parameters
  4. Compute iidx, fidx, imageInfo

Indexing – 2. Visual vocabulary

- The only remaining piece of code from James Philbin
- Use the Approximate k-means from Philbin et al. CVPR07, I found it very reliable (and repeatable for the same random seed).
    - Unlike my code which is all in C++, this code uses python, so it adds a bit of complexity in setting everything up (but not too much due to 0 dependencies apart from OpenMPI which I already use and a python library, pypar, for using OpenMPI)
- I made some modifications:
    - There were some bugs with dependencies (python-tables) which erroneously requested 100’s of GBs of RAM on our improved cluster. I removed these and re-implemented the relevant parts
    - Made a change needed for RootSIFT – as discussed in (1.) SIFT->RootSIFT conversion is performed on-the-fly in order to save storage/RAM

Indexing - Overview

- Indexing stages:
  1. Compute a subsample of descriptors for training
  2. Compute the visual vocabulary
  3. Compute Hamming Embedding parameters
  4. Compute iidx, fidx, imageInfo

Indexing – 3. Hamming Embedding

- Need to compute:
  - B projections of descriptors (B is the bitrate), i.e. a BxD matrix (D = descriptor dimensionality)
  - B medians for each of the K visual words.
- First step:
  - Assign all training descriptors to their visual words, easily done in parallel by master sending chunks of training descriptors to workers who assign them and send assignments back. As before, master needs to make sure ordering is preserved when saving assignments to the file.

Indexing – 3. Hamming Embedding (cont’d)

- Computing projections:
    - Jegou’s Hamming Embedding paper suggests using B random projections of the D-dimensional space. However I found it slightly better to compute B top principal components of the residuals (descriptor – its cluster centre), keep those and then do a random rotation of the B-dimensional projections.

Indexing – 3. Hamming Embedding (cont’d)

- Computing medians for each :
  - Simple to parallelize across visual words where each worker computes medians for some chunks of visual words.
  - Actually I implemented a slightly more elaborate method to keep the memory usage down, though in retrospect it is probably unnecessary. Instead of storing all values of training descriptors assigned for a particular word (this is the straight forward median computation approach – sort these, get the middle one), I switch to a histogram-based estimation of the median if the number of training descriptors is too large.

Indexing - Overview

- Indexing stages:
  1. Compute a subsample of descriptors for training
  2. Compute the visual vocabulary
  3. Compute Hamming Embedding parameters
  4. Compute iidx, fidx, imageInfo

Indexing – 4. iidx, fidx

- Main indexing bit: iidx, fidx, imageInfo
- Forward index (fidx) :
  - Fairly simple to compute and parallelize, master sends image names to workers, they extract descriptors, assign them to visual words, send results back, master saves results to the fidx.
  - Similar holds for imageInfo (i.e. worker sends back the width x height)
- Inverted index (iidx):
  - More complicated because of the “inversion”, i.e. we need to group same wordID’s together
  - It essentially boils down to sorting – given (imageID, wordID, extra_data) for every database feature which arrives from workers sorted by imageID, we want to sort the huge list by wordID.

Indexing – 4. iidx (cont’d)

- Constructing the iidx:
  - We don’t want to do it all in RAM, e.g. for 4B features with 32-bit imageID and wordID, and 64-bit HE signature, and ~7 bytes for ellipse geometry+location = 100 GB. It is possible to allocate that much RAM on the cluster, but:
    - It is wastefull
    - What if we don’t have large enough nodes (we didn’t use to) , we want something more flexible and therefore more portable
    - Makes it harder to schedule the job if we need the entire node – could need to wait for a long time, especially when deadlines are near
    - Parallelization would only be used to extract features, and not for sorting
    - Not even possible to store all of the data in a single data structure in C++ (maximal number of elements storable by std::vector is 1.07B)

Indexing – 4. iidx (cont'd)

- Constructing the iidx:
  - Adopt the "Blocked sort-based indexing" approach from the book "Introduction to Information Retrieval" by Manning, and adapt it to our case, parallelize etc.
  - Brief overview:
    1. Process all images and save all (imageID, wordID, extra_data) on the disk
    2. Divide the file into chunks which can be kept in RAM, sort every chunk by wordID and save to disk
    3. Create the final list of sorted items by merging all chunks at the same time. This can be done without storing all chunks in RAM (see later)
  - These are explained in more details next

Indexing – 4.1 iidx

1. Process all images and save all (imageID, wordID, extra_data) on the disk:
   - Simple to parallelize, see fidx construction
   - In real life, this stage is merged with fidx construction so that feature extraction (large amount of computation) is not repeated for fidx and iidx separately for no reason
   - Each worker saves its own items to its own chunk, i.e. items are not sent back to the master to save as the master would then be the bottleneck due to doing all the disk writes
     - Workers write separate files, so no problem
     - They send the names of the chunk files to the master
     - They keep the chunks smallish (~2GB) so they fit in RAM later
     - Careful: this part is not repeatable across runs as workers can get images in different order depending on node-speed etc. However, the final iidx is identical across runs

Indexing – 4.2 iidx

2. Divide the file into chunks which can be kept in RAM, sort every chunk by wordID and save to disk
    - Division was done by (1.)
    - Master goes through the list of chunks and assigns them to each worker
    - The workers sort the files
        - Note, for repeatability, that here (and in the next step) we need to define a total ordering between all (imageID, wordID, extra_data)
        - Namely, if a wordID appears twice in one image, we need to know how to order those two features in the posting list, just in order to be able to repeatably obtain the same end-product (iidx) and therefore simplify our lives
        - E.g. After wordID and imageID I sort by the feature x position, then y position, then scale etc..

Indexing – 4.3 iidx

3. Create the final list of sorted items by merging all chunks at the same time
   - Because all chunks are sorted, we don't need to load them all in RAM, but instead just sequentially read them out and merge them into one sorted list
   - See MergeSort – to merge two sorted lists, e.g. [1,5,8] and [2,4,9] we keep pointers to items in each list – at start (1,2) and then add to the merged list the minimal one, and advance the pointer
     - Start: pointers (1,2), merged list []
     - 1st step: pointers (5,2), merged list [1]
     - 2nd step: pointers (5,4), merged list [1,2]
     - 3rd step: pointers (5,9), merged list [1,2,4]
     - Etc.

Indexing – 4.3 iidx (cont'd)

3. Create the final list of sorted items by merging all chunks at the same time
   - Here however we don't merge 2 lists but k of them (k=number of chunks) – look up k-way merge
   - The idea is the same as for 2 lists – have a list of k pointers, find the minimal one, add that one to the sorted list and advance the relevant pointer.
   - However, with k lists the naïve way is O(k), we want something faster
   - Easily done with a priority queue (i.e. heap data structure) which provides O(1) search for the minimum and O(log k) addition/deletion from the list. An implementation of priority queue is available in C++ standard library (std::priority_queue)