

Relja_Retrieval v2.0

Manual for Developers

Relja Arandjelović

24 September 2014

- [1. Introduction](#)
- [2. Prerequisites](#)
- [3. Example: Build an Image Graph](#)
 - [3.1. Doing it sequentially \(i.e. on a single CPU/thread\)](#)
 - [3.1.1. Loading the indexes](#)
 - [3.1.2. Creating the retriever](#)
 - [3.1.3. Computing the graph](#)
 - [3.2. Doing it in parallel](#)
 - [3.2.1. Running it on a cluster](#)
 - [3.3. Compiling and linking the program](#)
- [4. Evaluating a retrieval method](#)
- [5. Building an API and web interface](#)
- [6. Further suggestions and miscellaneous stuff](#)
- [7. References](#)

1. Introduction

This document provides a glimpse into the `relja_retrieval` library that I have developed during my DPhil/Postdoc in the Visual Geometry Group, Department of Engineering Science, University of Oxford. It would take months to fully document everything, so I won't do that. I will instead show the basics: how to load an inverted index, forward index, set up a particular retrieval method, use it to query the database, make a parallel program, perform standard evaluation on the Oxford buildings benchmark, and build a basic web interface. These basics actually cover most use cases, and when combined with the stuff in **`docs/Manual_for_Black_Box_Usage.pdf`** (how to index a database, run a backend/frontend, talk to API) one should have a solid basis for using the library. Further topics (e.g. details of the indexing procedure, described on a high level in **`docs/Conceptual_overview.pdf`**) are left to the reader who can then further explore the code to understand it in depth.

2. Prerequisites

- Solid knowledge of C++, including:
 - Core things such as loops, functions, pointers/references, new/delete, passing arguments by value or reference, basic knowledge of templates, what does it mean when a method is declared 'const', etc
 - Ability to compile/link a program
 - Object oriented programming: classes, constructor/destructor, inheritance (no need for multiple inheritance), virtual functions, abstract classes
 - Knowledge of the standard library, e.g. what are and how to use `std::vector`, `std::set`, `std::map`, iterators, `std::sort`, etc
 - Desirable: protocol buffers
- Basic knowledge of algorithms (though the more the better!), including:
 - Knowledge about algorithmic complexity (the big-O notation), e.g. what's the complexity of adding an element to `std::set`, or iterating through the values?
 - Depending on the level of involvement, larger knowledge might be required. E.g. for indexing one needs to know how to implement merge sort, what are priority queues (heaps); for image graph analysis one needs to know graph algorithms such as depth first search, connected components, etc.
- Read the:
 - `docs/Conceptual_Overview.pdf` [1]
 - `docs/Manual_for_Black_Box_Usage.pdf` [2]
- For building a web interface: need to know basic python and `python-cherrypy3`

3. Example: Build an Image Graph

It's probably best to dig right into it. This is an example of how to build an image graph, the graph consists of nodes which are images, and edges signify that images contain the same object. We construct it by querying with each image in the database and record the positive retrievals. Note that our graph will potentially be asymmetric as the burstiness regularization is asymmetric, and so is the knn relationship (as I only keep top k retrievals).

Here I will illustrate how to set up a retriever, use it to query the database with each image in turn, parse the results and save them to a file. I will also show how to do this in parallel, either on a shared memory system (i.e. using multi-threading) or on a cluster (i.e using MPI).

- The relevant files are:

```
v2/retrieval/image_graph.h
v2/retrieval/image_graph.cpp
v2/retrieval/CMakeLists.txt
v2/retrieval/tests/example_image_graph.cpp
v2/retrieval/CMakeLists.txt
```

3.1. Doing it sequentially (i.e. on a single CPU/thread)

Let's first look at a basic approach: go through images sequentially and query using them. For this, ignore all lines that contain MPI_* in them, and assume the variable rank equals 0.

The program should be executed (from the build directory) as:

v2/retrieval/tests/example_image_graph s

Look at:

```
v2/retrieval/tests/example_image_graph.cpp
and the command line argument means that choice=='s'
```

First we set up several strings for various filenames:

the output: imageGraphFn

the inputs: iidxFn, fidxFn, wgghtFn, hammFn . iidxFn, fidxFn and hammFn refer to the inverted index (iidx), forward index (fidx) and the Hamming Embedding information files, these get created automatically at indexing stage. The wgghtFn contains idf weights and L2 normalization factors for all images, and it gets created the first time a tf-idf based retriever is created. I hardcoded all these filenames here, of course it would be better to have a configuration file, but this would clutter the example too much and divert focus from the core stuff.

3.1.1. Loading the indexes

Next we load the `fidx` and `iidx`, the commands are identical so I go through only one of these:

```
protoDbFile dbFidx_file(fidxFn);  
protoDbInRam dbFidx(dbFidx_file, rank==0);  
protoIndex fidx(dbFidx, false);
```

The first line sets up the `protoDb` file (see [1] slides 5-6). `dbFidx_file.getData(ID, data)` then given an ID returns a vector of strings (the vector of strings is a slight complication due to implementation details, see [1] slide 13), i.e. basically we can now given an ID get some raw binary data.

The second line sets up a `protoDbInRam` (the extra argument, remember in this subsection `rank==0` so the 2nd argument is “true”, is just to do with verbosity), i.e. it provides exactly the same functionality as `protoDbFile` but it preloads the entire file into RAM so that `getData()` call is much much faster (you should really really do this unless there is insufficient RAM, but if there is insufficient RAM you shouldn't be trying to run a retrieval system anyway, unless it is some sort of an emergency...). Here there's already a glimpse of object oriented programming which one needs to know about in order to use the code: both `protoDbFile` and `protoDbInRam` implement a `protoDb` (they are derived classes from a purely abstract class `protoDb`) so they are both able to answer `getData()`.

The third line now uses this `protoDbInRam` as an index (see [1] slide 7), i.e. we now know how to interpret the binary data we read from `getData()`, the data is interpreted as `rr::indexEntry`. Now we can call `fidx.getEntries(ID, entries)`. For the forward index, the input ID is the imageID (I usually call it document ID, `docID`), and get relevant information ([1] slide 9); similarly for `iidx` the input ID is a visual word ID, and entries are the posting lists ([1] slide 8).

As an example (not needed for the image graph, but useful in general), to get a posting list for visual word #10, one would do:

```
iidx.getData(10, entries).
```

Entries is actually a `std::vector<rr::indexEntry>` as the potentially large amount of data needs to be divided into chunks (see [1] slide 13), for simplicity let's just examine the first chunk (often there will indeed only be one chunk, but be careful not to forget that more than one chunk is possible!):

```
rr::indexEntry const &entry= entries[0];
```

Now to read the first element of the posting list (assuming that it is not empty, this should be checked too!), e.g. to print the smallest `docID` (because `docID`'s are sorted in a posting list) which contains word #10, do:

```
std::cout<<entry.id(0)<<"\n";
```

Other potential data that can be stored in an index can be seen from the `rr::indexEntry` definition

- this is actually a protocol buffer (read a bit on them, it is fast to understand), so the c++ code has been automatically generated from the protocol buffer specification in

v2/indexing/index_entry.proto . So for example, if geometry is saved in the index as well (I tend to do this), then there will be x and y entries, or more likely qx and qy (rounded values to the nearest integer, for compression reasons), e.g.

```
std::cout<<entry.qx(0)<<" "<<entry.qy(0)<<"\n";
```

But before assuming that qx and qy exist, it's probably best to have a quick check:

```
ASSERT(entry.id_size()==entry.qx_size() && entry.id_size()==entry.qy_size());
```

See about ASSERT a bit later in this document, but should be self-explanatory.

The index can store various other stuff, e.g. if it is a BoW index it could potentially store word counts (*count* field), if it is soft assigned BoW which needs floating point weights (not recommended as it requires too much space), there will be a *weight* field, or unspecified binary data can be stored (field *data*) - this field I use for example to store the Hamming Embedding signatures. [1] talks a bit more about the index in slides 7-13.

3.1.2. Creating the retriever

Now that we have loaded the indexes, we can use them for retrieval. E.g. we make the tf-idf based retriever:

```
tfidfV2 tfidfObj(&iidx, &fidx, wghtFn);
```

If wghtFn doesn't exist it will compute the idf weights and save them into wghtFn, otherwise it will just load the file. tfidfV2 is derived from retrieverV2 (v2/retrieval/retriever_v2.h) which in turns is derived from retriever (retrieval/retriever.h). We can see now that we can use any retriever object, including tfidfV2, to perform queries such as:

```
tfidfObj.internalQuery(docID, queryRes, toReturn)
```

which given a docID (=imageID), queries the database (in this case we know that tf-idf uses the fidx and iidx for fast retrieval), returns up to toReturn number of results (if toReturn==0 then the entire database is ranked) into the vector queryRes. queryRes is a ranked list of (docID, score) pairs such that score is non-increasing. We will use exactly the internalQuery later when making the image graph.

However, I don't want to use tfidf as the retriever, I want Hamming Embedding (+burstiness normalization) plus spatial reranking, as these work better. So I create these now:

```
hammingEmbedderFactory embFactory(hammFn, 64);  
hamming hammingObj(tfidfObj, &iidx, embFactory, &fidx);  
spatialVerifV2 spatVerifHamm(hammingObj, &iidx, &fidx, true);
```

The first line creates a Hamming Embedder factory - sorry, but I don't have time to explain details of this as then I would need to write a 200 page document for every single function in the library. Examining the code, function names, and especially how I use it, should be sufficient to explain it. It basically enables one to make a hammingEmbedder which in turn can convert a descriptor into its Hamming signature, or decode binary data (the one in rr::indexEntry as

mentioned in 3.1.1) as Hamming signatures (this is exactly what the Hamming retriever will use it for).

The second line make the hamming retriever: it needs the tfidf retriever as an argument (as it does use the idf weights and it would be silly to not re-use the code from tfidf), the hammingEmbedderFactory and of course the fidx and iidx.

Finally, the third line creates a spatial verifier retriever. This retriever takes as an argument another retriever (in this case hamming, but we could use tfidf as well, or any other one) because when we query with spatialVerifV2, it calls the base retriever first (here hamming), and then reranks a short-list. Here we can again see object oriented programming - spatialVerifV2 doesn't care if it uses hamming, tfidf, <whatever>, it just needs an object to which it can talk to as a retriever object.

3.1.3. Computing the graph

```
imageGraph imGraph;  
imGraph.computeSingle(imageGraphFn, fidx.numIDs(), spatVerifHamm, 100, 10 );
```

Self explanatory - we make the imageGraph and compute it using a single core/thread. The computation function needs to know how where to save the output, how many image there are (so that it can query using all of them), what retriever to use, and two thresholds (how many edges should a node be allowed to have, and the score threshold which deems that two images are similar enough; here I set these two arbitrarily to 100 and 10, don't think that this is optimal).

Now let's examine the computeSingle (v2/retrieval/image_graph.cpp)

We create the file in which the graph will be stored. Actually we will again use protoIndex (and protoDb as the backbone), as what we want is to store for every docID (=imageID) the docID's of its neighbours. rr::indexEntry fits this just fine ([1] slide 7, and 3.1.1) as it can store a list of IDs as well as a list of weights (this is where we will store similarity scores). So we start by creating a protoDbFileBuilder (i.e. object which is able to add binary data into a new protoDb file), and a indexBuilder which uses the protoDbFileBuilder which stores the rr::indexEntry into the protoDb file.

```
protoDbFileBuilder dbBuilder(filename, "image graph");  
indexBuilder idxBuilder(dbBuilder, false, false, false);
```

The extra parameters to the indexBuilder constructor are irrelevant (the first 'false' should be 'true' if one wants to compress an index, see [1] slide 11, but for this docIDs need to be sorted while we are sorting by score.. could alternatively sort by docID which would enable compression).

Digression: A useful utility class is timing::progressPrint

```
timing::progressPrint graphBuildProgress(numDocs, "imageGraph");
```

We tell the `progressPrint` object that there will be `numDocs` jobs (as we will query with each document in turn) and a prefix for printing. Then in every for loop iteration we simply call `graphBuildProgress.inc()` and this will print some useful progress information:

```
imageGraph: 2014-Sep-20 23:28:07 1 / 5062
imageGraph: 2014-Sep-20 23:28:07 2 / 5062; time 00:00:00; left 00:18:54; avg 0.224288 s
imageGraph: 2014-Sep-20 23:28:07 3 / 5062; time 00:00:00; left 00:23:08; avg 0.274506 s
imageGraph: 2014-Sep-20 23:28:08 4 / 5062; time 00:00:00; left 00:25:06; avg 0.297808 s
imageGraph: 2014-Sep-20 23:28:08 8 / 5062; time 00:00:01; left 00:21:02; avg 0.249838 s
imageGraph: 2014-Sep-20 23:28:10 16 / 5062; time 00:00:03; left 00:18:13; avg 0.216658 s
imageGraph: 2014-Sep-20 23:28:14 32 / 5062; time 00:00:06; left 00:18:42; avg 0.223111 s
imageGraph: 2014-Sep-20 23:28:20 64 / 5062; time 00:00:13; left 00:18:09; avg 0.21797 s
...
```

I.e. each line will start with the specified prefix, then we see the current date and time, current status (e.g. 16/5062 means we're working on #64 out of #5062 jobs), elapsed time, estimated time left until all is done, and average time per job.

The `inc` function can actually also take an argument, an extra prefix, in order to specify extra information specific to current state, e.g. in `computeSingle()` we actually also print out the current average number of neighbours per node.

Back to image graph computation. The core code is simply:

```
for (uint32_t docID=0; docID < numDocs; ++docID){
    queryRes.clear();
    retrieverObj.internalQuery( docID, queryRes, maxNeighs );
```

Namely, go through all images, and issue a query using the retriever. Looking at `retriever/retriever.h` we can see several more options, e.g. `internalQuery()` that we use queries with the entire image. We can also query using a custom query object (`represent/query.h`) where we can specify a query ROI, or use an external query `externalQuery()` which takes an `imageFn` as input.

Slight digression: what I learnt through years + in my internship at Google - it's quite useful to have simple sanity checks as we go along. It is surprising how often one makes silly mistakes, and these can often be caught by simple checks. Here for example I check (using my macro `ASSERT` defined in `util/macros.h`):

```
ASSERT( maxNeighs==0 || queryRes.size()<=maxNeighs );
```

I.e. I just check to make sure that we did indeed get less than `maxNeighs` results back (that is if `maxNeighs` is not 0, as 0 is a special value which tells the retrieval to rank the entire database), as the retriever we use might not be implemented properly and maybe returns more than we want. Similarly, I did a check of a posting list in 3.1.1 to verify it is not corrupt somehow, and I do a few more checks later, such as ensuring that the graph loaded from a file is valid, etc.

Now we simply go through the list of `queryRes` (query results), which are in the (`docID`, `score`) form, and record the `docID`'s for which score passes the threshold.

We save the neighbours into the graph in RAM (see the `graph_` variable declaration in `v2/retrieval/image_graph.h`), as well as onto disk (i.e. we add it into `idxBuilder`)

```
graph_[docID]= neighs;  
idxBuilder.addEntry(docID, entry);
```

Finally, we close the index file holding our graph:

```
idxBuilder.close();
```

Going back to `v2/retrieval/tests/example_image_graph.cpp` we can now print out some graph chunks, if we run the code with

`v2/retrieval/tests/example_image_graph e`
(i.e. `choice=='e'`)

This will load the graph - see `v2/retrieval/tests/image_graph.cpp` : `loadFromFile()`. Should be simple to understand: we read the `protoIndex` (this time there is no need for `protoDbInRam` as we will only read the file once), and fill in the `graph_` structure with the loaded data. Note that here I also had some sanity checks such as:

```
ASSERT( entry.id_size() == entry.weight_size() );
```

just to make sure all is ok (i.e. there are as many resulting `docID`'s as similarity scores).

This part of the code will now print out some graph edges, as well as create a rudimentary html file for easier visualisation (you need to make a soft link in the `.html` file folder towards the Oxford buildings dataset, e.g. **`cd path/to/html/folder; ln -s ~/path/to/OxfordBuildings/oxc1 oxc1`**). One thing I wanted to illustrate with this piece of code (apart from that the image graph is reasonable), is the `datasetV2` class. This one is basically what is described as "imageInfo" in [1] slide 12, i.e. it contains `docID` <-> filename mapping and image resolution info. Here we use it to get the file name for a specified `docID` in order to be able to visualise the results, e.g. `dset.getFn(docIDres)`.

That's it! Well, that's it if you want to spend days on computing the image graph for larger datasets, otherwise we need to parallelize things and use the cluster. Coming up next.

3.2. Doing it in parallel

Now that we know how to make an image graph sequentially, we will convert the code so that it can be executed in parallel, i.e. we will look at `computeParallel()` in `v2/retrieval/image_graph.h`. First make sure to read [1] slides 27-30. The idea is: we will have one master process which will manage scheduling of jobs, sending jobs to workers, getting results back and saving them. The workers will simply perform something that is computationally expensive. Here a natural split is: workers will do the `internalQuery` while the master will save results to RAM / `graph_`.

Recall from [1], we parallelize in two ways - though you don't need to, often only 1 of the 2 is enough, but for illustration purposes I do it in both ways now. They are:

1. The CPUs don't share memory. Common case is a cluster with separate computational nodes which can talk to each other. For this we will use Message Passing Interface (MPI), specifically boost library's functions for mpi.
2. The CPUs share memory. This is the case for systems with multiple cores on a single machine. In this case we don't have to do MPI and can use multithreading

In option (1.) because cores don't share RAM, each core needs to load the entire iidxfidx into its RAM (unless maybe there is a smart way of splitting jobs so that a portion of the indexes is sufficient, but this is unlikely). One can use MPI on CPUs which share memory, but this would (generally speaking) be wasteful as then each core would load its own version of the index. It is better to just have one index in RAM that all cores can use at the same time (**note**: this is only possible because I implemented the indexes in a thread-safe manner), and execute workers in threads (option 2). We need to take care to properly address these slight differences between approaches (1.) and (2.).

Generally:

- On a cluster: use MPI
- At runtime: use multithreading as it is most memory efficient (but make your functions thread safe!)

Approach 1, MPI:

At the very start of the program (example_image_graph.cpp) we need to initialize MPI, I provide a macro for this: `MPI_INIT_ENV`. To run a program for this approach, one needs to do `mpirun -np <number of CPUs> executable`

so to run

`v2/retrieval/tests/example_image_graph p`

we need to do, e.g.:

`mpirun -np 4 v2/retrieval/tests/example_image_graph p`

or if using a GridEngine on a cluster, do simply:

`mpirun -np $NSLOTS v2/retrieval/tests/example_image_graph p`

where \$NSLOTS is automatically populated from what you specify in the qsub command (we'll do this later).

When a program is executed with mpirun, it is actually started multiple times, so `mpirun -np 4` will start 4 instances of the program (on a cluster this will be set up such that each instance is scheduled to a different CPU). Inside a program it is possible to check "which instance am I?" through the rank variable (provided by the `MPI_GLOBAL_ALL` or `MPI_GLOBAL_RANK` macros I made). Without loss of generality, I always assume `rank==0` is the master, and `rank>0` are workers. `MPI_GLOBAL_ALL` also defines `numProc` which tells us how many processors there are. Conveniently, if the code was not run using mpirun, then `numProc==1` so we know automatically that we will use a multithreading approach instead.

So, as before, we start by loading iidx and fidx, and remember - every process is loading its own. If you just write `std::cout<<"test\n"` you will get "test" written N times where N is the number of processes! So it is much better to do `if (rank==0) std::cout<<"test\n";` This is the background story behind the line I glanced over last time (in the sequential case):

```
protoDbInRam dbFidx(dbFidx_file, rank==0);
```

I'm basically telling the protoDbInRam constructor to be verbose only if rank==0 (i.e. it's the master).

Approach 2, Multithreading: Nothing different to the single thread case has happened yet as we are still running only a single thread of execution, until we get to image_graph.cpp . To run, simply call

v2/retrieval/tests/example_image_graph p

Now let's have a look at computeParallel() in image_graph.cpp - it is quite simple. We create a manager object (i.e. the thing that master will run), and a worker object (i.e. the object that does the work on many machines), and we run the parallel queue with these two. We also specify if we want to use threads (which is automatically detected by `detectUseThreads()`), as if numProc==1 we want threads; however, I haven't found/investigated a nice way of specifying how many threads we want so it is currently hard coded with a random =4 number; clearly this depends on the number of CPUs you have).

Note that the manager object is created only for rank==0 (i.e. if we are using MPI and we are the master, or if we are using multithreading where we are still by definition in the main thread as no other threads have been started yet). Similarly at the end of the function we delete the manager only if rank==0 as otherwise manager==NULL so there is nothing to delete.

Now, we only have to define what does a manager do, what does a worker do, and what kind of a result is passed between them.

imageGraphResult : we define it to be simply the same type as queryRes, i.e. a list of pairs of (docID, score)

```
typedef std::vector<indScorePair> imageGraphResult;
```

imageGraphWorker : it derives from a queueWorker<imageGraphResult>, i.e. a queueWorker which returns a imageGraphResult result. It takes the retriever object and maxNeighs as constructor arguments and stores them. What any worker does is defined in the operator() function, whose arguments are jobID and result (where the jobID has been automatically sent to the worker by the master). So, we only need to define what a worker does for a particular jobID - we define docID=jobID, so we, as expected, simply query with the docID:

```
queryRes.clear();  
retriever_.internalQuery(docID, queryRes, maxNeighs_);
```

imageGraphManager : it derives from queueManager<imageGraphResult>, i.e. a

queueManager which expects imageGraphResult from workers. Actually this one derives from managerWithTiming<imageGraphResult> which in turn derives from queueManager<imageGraphResult>, and simply also prints progress information along the way automatically. Here we need to implement compute(jobID, result), i.e. what does a master do once a worker has finished jobID and returned result, and potentially finalize() which performs some final tasks.

For compute(): we again copy relevant parts from computeSingle(), i.e. we keep results above some threshold, and save them to RAM and onto disk. However, there is one complication - things are not sequential any more, jobIDs 5, 6 might complete before jobID 4 finishes due to some jobs taking longer due to their complexity, or because a node is slower etc. Therefore we do additional bookkeeping where we store results into an intermediate buffer_, which we write onto the disk whenever possible, i.e. if we get job's finishing in this sequence: 1 4 3 2 6 5, we will:

- 1 comes: save 1 to buffer, write 1 to disk, delete 1 from buffer
- 4 comes: save 4 to buffer
- 3 comes: save 3 to buffer
- 2 comes: save 2 to buffer, save 2 to disk, delete 2 from buffer, save 3 to disk, delete 3 from buffer, save 4 to disk, delete 4 from buffer
- 6 comes: save 6 to buffer
- 5 comes: save 5 to buffer, save 5 to disk, delete 5 from buffer, save 6 to disk, delete 6 from buffer

In finalize() we simply close the idxBuilder file. Here again I have a sanity check `ASSERT(buffer_.size()==0);` to ensure that the buffer is free i.e. my algorithm is correct.

Now we just run the parallel queue with the manager and worker:

```
parQueue<imageGraphResult>::startStatic(  
    numDocs, worker, manager, useThreads, numWorkerThreads);
```

where numDocs is the number of jobs to be executed.

3.2.1. Running it on a cluster

_image_graph.sh

```
#!/bin/bash  
#$ -cwd
```

```
for modname in apps/cmake/2.8.9/gcc-4.4.6 apps/hdf5_mpi/1.8.9/gcc-4.4.6+openmpi-1.6.3  
compilers/gcc/system libs/boost/1.51.0/gcc-4.4.6+openmpi-1.6.3 libs/gcc/system  
mpi/openmpi/1.6.3/gcc-4.4.6; do module add $modname; done
```

```
cd /users/relja/Relja/Code/relja_retrieval/build
```

```
mpirun -np $NSLOTS v2/retrieval/tests/example_image_graph p
```

Some of the modules can probably be removed (e.g. cmake) but I keep the list the same as the one I use for compiling the program (see [2]).

Run on (for example) 50 nodes:

qsub -pe mpislots 50 _image_graph.sh

the *-pe mpislots* bit depends on the cluster setup and the name of the relevant queue which supports MPI (it's like this for Titan right now, September 2014).

3.3. Compiling and linking the program

I use cmake as it makes life easier. We compile a library `image_graph`, see

v2/retrieval/CMakeLists.txt :

```
add_library( image_graph image_graph.cpp )
target_link_libraries( image_graph
    index_entry.pb
    par_queue
    proto_db
    proto_db_file
    proto_index
    retriever)
```

This basically tells the dependencies of `image_graph` library: it needs the `index_entry.pb` (which defines `rr::indexEntry`), `par_queue` which has the parallel queue stuff, `proto_*` which contain the `proto*` classes, and `retriever`.

For the actual executable `example_image_graph` we look in **v2/retrieval/tests/CMakeLists.txt** :

```
add_executable( example_image_graph example_image_graph.cpp )
target_link_libraries( example_image_graph
    dataset_v2
    hamming
    image_graph
    mpi_queue
    proto_db
    proto_db_file
    proto_index
    spatial_verif_v2
    tfidf_v2 )
```

We see that this one also depends on specific things such as `tfidf_v2`, `hamming`, etc. The reason that `image_graph` doesn't depend on `tfidf_v2` or `hamming` is that it doesn't care which retriever it uses, it depends only on `retriever`. But `example_image_graph` actually cares which retriever it uses, because it explicitly creates the `hamming` retriever and passes it to `image_graph`.

Now typing in

make example_image_graph

should do the trick (usually doing **make -j** instead of **make** works well for me, it compiles things in parallel). See [2] for how to compile on titan (not much is different).

4. Evaluating a retrieval method

Ground truth files for a few benchmarks are already included in data/gt/ : Oxford, Paris, Holidays, San Francisco landmarks, Pittsburgh. You will need to make a folder (somewhere outside of the code) for temporary files, e.g. do:

```
mkdir ~/gt
cd ~/gt
ln -s path/to/relja_retrieval/data/gt datasets
mkdir rr_format
```

Now have a look at **v2/retrieval/tests/retv2_temp.cpp**

I hardcoded the paths, put your own ones. You can see similar stuff to what we saw in the image graph example - we load the fidx, iidx, make a retriever we want (note again: evaluation doesn't care which retriever is used, it only cares that we can do retrieverObj.internalQuery()), and load the dataset information. The only two extra required things are:

```
evaluatorV2 evalObj( gtPath, "Oxford", &dset );
double mAP_hamm= evalObj.computeMAP( hammingObj, NULL, false, true );
```

The first line sets up an evaluation object, which needs to know where ground truth files are (i.e. ~/gt in our example), and which dataset we want to evaluate. Note that this creates a file **gt/rr_format/Oxford.v2bin** the first time it runs (other times it just loads this file), which makes a mapping between Oxford ground truth and our own docID numbering system (to speed up computing mAP so that we don't need to convert between docID's and filenames all the time). This is important to know, as if your docID's change, you should delete the file!! This in general won't be needed, even if you re-index the dataset docID's will remain the same given that the same imagelist.txt file is given to the indexer (see [2] Indexing). If you however change the imagelist.txt and shuffle things around, then you need to delete the created file and make the evaluator regenerate it using the new dataset information.

The second line just runs the evaluation, i.e. queries with all images in the ground truth and outputs mAP. There are a few options - if you want to get also AP for each query, create a vector of doubles:

```
std::vector<double> APs;
```

and pass &APs as the second argument instead of NULL.

The other two arguments are to do with verbosity, for full verbosity put (true, true), for none put (false, false), while the above example (false, true) does something in between (prints some of the stuff). Running the code v2/retrieval/tests/retv2_temp with provided files should, amongst other things, output:

```
mAP_hamm= 0.7070
mAP_hamm+sp= 0.7319
```

That's all!

Note: I haven't tried evaluating on Paris with the latest version of my code so this part of code is untested, but should work or should be very close to working. If you can't get it to work, email me and I'll be able to help.

To evaluate on another dataset, e.g. San Francisco, see **v2/retrieval/tests/eval_SF.cpp**, and things will be quite similar. Main differences are:

```
evaluatorV2 evalObj( gtPath, "SF_landmarks_2014_04", &dset );  
double rec_hamm= evalObj.computeAvgRecallAtN( hammingObj, 100, 10, NULL, true,  
true );
```

It is different because the evaluation procedure for San Francisco is different - we don't measure mAP but averageRecallAtN. Note that things are a bit hacky for SF_landmarks and Pittsburgh - queries are external images, so one first needs to generate features for all query images (e.g. see v2/evaluation/tests/compute_SF_query_feats.cpp) and then these filenames are hardcoded in evaluator_v2.cpp (no time to make everything neat unfortunately..).

5. Building an API and web interface

For information on how to run a web interface or talk to the API, consult [2].

Here I will just give a brief description of what is going on. For the backend (I call it api which might be confusing as there are many API's around), see `v2/api/api_v2.cpp`, it sets everything up, the retriever, the featureGetter (an object which extract features, e.g. RootSIFT), etc. It then constructs an API object (`api/spatial.h`, `api/spatial.cpp`) which does all the core things of receiving the query via TCP, parsing the query xml, executing the query, making an output xml, and sending it back over TCP. For example, here is one arbitrary query:

```
<internalQuery>
<docID>50</docID>
<x1>5</x1><xu>100</xu><y1>30</y1><yu>400</yu>
<numberToReturn>100</numberToReturn>
</internalQuery> $END$
```

I.e. we are searching with `docID=50` and roi defined by corners (5,30), (100,400), and we want to get first 100 results. The function `API::getReply` processes this query, the XML is parsed and checked for various query types, we are interested in the `if (pt.count("internalQuery"))` block as this one is the one executed. It simply reads the `docID` from the XML:

```
uint32_t docID= pt.get<uint32_t>("internalQuery.docID");
```

constructs a query object based on the specified ROI (default values of (-infinity, -infinity), (+infinity, +infinity) are used if the ROI is not specified), reads the `numberToReturn` and `startFrom`, and calls `queryExecute` with these parameters. Query execute then simply calls the `spatialRetriever` object (`spatialRetriever` and not just `retriever` as we also return the homographies for nice visualisation of query regions). Finally, the results are being put back into XML form by `API::returnResults`.

The actual communication over TCP is done in `api/abs_api.cpp`, but you don't really need to know details of how this works.

Now, to make a web user interface, we build a python API object, which defines the same functions as the backend api, and executes them by writing the XML request, sending it to the backend, getting results back and parsing them. This is all done in `api/api.py`, e.g:

```
def internalQuery( self, docID= None, x1= None, xu= None, y1= None, yu= None,
startFrom= 0, numberToReturn= 20 ):
    request= "<internalQuery>";
    request+= ...
    ...
    request+= "</internalQuery>";
```



```
reply= self.customRequest( request );
```

```
results= self.getResults(reply);
```

```
return results;
```

So the first bits make the XML string, then customRequest() sends it over TCP, and getResults parses the resulting XML and returns a list of results.

Now for a web interface, we use python-cherrypy3. In short, this is a web server where an HTTP request is simply converted into a python function. So what we do is:

In **ui/web/webserver.py** we set up various stuff (read the config file, construct the python API object which will talk to the backend via the APIport, etc) and run cherrypy server with the webserver object. The webserver in turn states `self.dosearch= self.doSearch_obj.index`; where the `self.doSearch_obj.index` is actually a function, so calling `self.dosearch()` will call the `doSearch_obj.index()` function.

Now, one can execute the `self.doSearch_obj.index()` function by simply going to the web server address using a browser, and going to /dosearch (as dosearch redirects to doSearch_obj.index) e.g. <http://localhost:8080/dosearch>

Now have a look at: **src/ui/web/do_search.py** which defines the index() function:

```
def index(self, docID= None, uploadID= None, xl= None, xu= None, yl= None, yu= None, startFrom= "1", numberToReturn= "20", tile= None, noText= None, dsetname= None):
```

We can call this function by going to a web browser, e.g. do:

<http://localhost:8080/dosearch?docID=50&xl=5&xu=100&yl=30&yu=400&numberToReturn=100>

Cherrypy now automatically turns this into a function call of `index(docID= "50", xl= "5", xu= "100", yl= "30", yu= "400", numberToReturn= "100")`;

We have everything now. A user can go to a browser, click on a specific link, which in turn executes a python function in `do_search.py`: `index()`. In this function we call the python API object, which communicates with the c++ backend and retrieves the results. Finally, the `index()` function returns a web page.

Of course, there is much more to this code as well but I can't write a 200 page document. There are also a few subtleties, e.g. we don't show entire images on a web page as this could take quite some time to load, but actually generate thumbnails on the fly (see `dynamic_image.py`; the code can also draw ROIs, crop the ROIs etc).

Python rant

My recommendation: run away from python as much as you can. It is a very useful language for quickly doing things, such as making a web server, or parsing textual files etc, so it is certainly something you should know as it can simplify your life. However, many things are actually "broken" (they are supposed to be like that, i.e. it is a feature, not a bug), e.g.:

- multithreading doesn't actually exist (it's just an illusion)
- destructors are not guaranteed to be called (e.g. I was writing some meta information into a file in a destructor but because the destructor was not called, as the automatic garbage

collection was confused because I had some circular references, the file was never closed and was therefore corrupt; wasted a day of cluster computations + debugging time)

- storing a serious amount of data can be very expensive, as often variables take lots more space (there's often a lot of extra information stored)
- it is generally much slower than C++, e.g. for loops are expensive (somewhat like matlab).. I recommend using python-numpy if you really want to do some intensive computations, which kind of gives you an ability to do matlab-like vectorization

6. Further suggestions and miscellaneous stuff

Try to make methods **const** if possible (of course it often isn't), this makes life easier, e.g. for ensuring thread-safety and easy parallelization. It should be noted that if a method is **const** it is **NOT** necessarily automatically thread-safe (hint: check out the *mutable* qualifier, also a method could be reading from a file which could be changing across runs, etc), but it usually is.

As mentioned before, do insert **ASSERT's** here and there for sanity checks. Btw, the difference between my macro **ASSERT** and the default **assert** is that **assert's** are ignored at runtime on Release builds.

Another useful thing to keep in mind - I believe it is a good philosophy I have and Google does too: **don't write useless code**. People often get preoccupied by writing copy constructors, assignment operators and a bunch of other things, inflating their code by a huge amount, and then these never get used. It 1) wastes time, 2) increases errors as the code is harder to maintain (test, update..). So, only implement things when you need them. For this, I have a useful macro in util/macros.h : **DISALLOW_COPY_AND_ASSIGN** . Namely, when you make a class (e.g. image_graph.h) at the end add:

```
private:  
    DISALLOW_COPY_AND_ASSIGN(imageGraph)
```

If any code tries to use a copy or assignment constructor of imageGraph there will be a compilation error. This is very useful as sometimes one by accident copies an object (e.g. it is enough to forget a **&** in a function argument), this macro prevents it from happening.

Learn about **protocol buffers**. They are very useful for storing data in a consistent format - I first used boost's serialization but this was actually terrible (I love boost, but serialization was a nightmare - it wasn't necessarily portable across computers!), while protocol buffers are great. They work well, Google actively uses it, and it also has some automatic compression (if you know how to properly use it). I use it mainly for rr::indexEntry (v2/indexing/index_entry.proto), e.g. there diffid is repeated uint32, so there is automatic compression if diffid is on average not large (e.g. if it is often under 16k it will get represented with 2 bytes, but it is capable of representing also numbers up to 4 billion; if one used fixed32 it would take 4 bytes instead of 2!). I also use it for storing idf's (v2/retrieval/tfidf_data.proto) - it should be enough for one to check my code (e.g. see v2/retrieval/tfidf_v2.cpp : save()), read a bit about protocol buffers, and become very proficient quickly. One good use would also be to simply always make queueManager / queueWorker operate with std::string, and actually have these to be protocol buffers and encode / decode them in manager / worker.

Definitely use a version control system, **git** is my favourite. Makes it easy to revert to previous versions of the code, work on things in parallel via branches, etc.

For checking memory leaks or debugging segmentation faults: **valgrind** is a very useful tool (sudo apt-get install valgrind). It makes the code run slower (but that's ok as it is only used for debugging and not in real life), and can't really handle huge amount of data (e.g. if you need more than 1 GB of RAM then probably best to forget it), but it is very good in pointing out if there are memory leaks, and what has been leaked. I run it every time I do something new (actually I just didn't run it on the image_graph example, but it should be fine). It also has a profiling tool.

Under **obsolete/** I've kept some old code, which could potentially be useful (e.g. it has libsvm and how to talk to it).

Some feature detectors / descriptors are given in **preprocessing/**, e.g. see feat_standard.h . One can construct a Hessian-Affine detector by Krystian Mikolajczyk, etc. Note that his code is integrated into relja_retrieval under **external/KMCode_relja/** . The code is very messy though - I fixed all memory leaks in functions I was using (for the Hessian-Affine detector and SIFT descriptors) and didn't touch other functions. However, you hopefully will never need to deal with his code, as my **preprocessing/** is sufficient. There is also colour sift (OpponentSIFT) by van de Sande, but this relies on an external binary you can download from his website (and need to put it into your system path).

Note that you should really really prefer to use code which is compiled with your program, and not having to, like for OpponentSIFT, call an external binary. I had this problem once - if your code occupies a lot of RAM (e.g. a large database is searched and the index is 20 GB), calling the feature computation using an external binary will take **ages**! This is because it can only be executed as a subprocess, and starting subprocesses is actually expensive if the main process is using a lot of RAM (used to be that all the RAM is copied into the subprocess, now only some things are supposed to be copied such as paging information, but with recent Linux I still had problems where feature computation would take 20+s instead 1s due to this).

There's a bunch of **other useful code** in relja_retrieval, e.g:

- v2/api/api_v2.cpp shows how to parse a configuration file, e.g. `pt.get<std::string>(dsetname+".idxFn")`
- v2/retrieval/mq_* : implements various multiple query methods (see my BMVC 2012 work, or more recent: the IJMIR paper if it gets in), i.e. this is the backbone of the on-the-fly retrieval where one downloads images from google image search and then uses the set of images for querying (instead of a single image). On how to set it all up, consult v2/api/api_v2.cpp
- compression/product_quant.h: Implements product quantization (but not learning vocabularies!) and supports asymmetric search (getDistsSq()). The code for the coarse to fine approach by Jegou (i.e. ANN search by quantizing to the nearest cluster and then checking PQ signatures) is basically there but commented out as I haven't had the time to upgrade it when I made some major rewrites to my code (to basically switch from an old indexing format to the new one). It should be very easy to adapt obsolete/tests/coarseann_test.cpp and related stuff to use the new protoIndex, as rr::indexEntry is already very suited for this (with it's data field which right now I use in a

similar way for Hamming Embedding). If needed, I can probably do it quite quickly.

- v2/indexing/daat.h: performs document-at-the-time traversal of the index, see “Size Matters” by Stewenius et. al. ECCV 2012 (though the method is slightly different).
- util/timing.h: Amongst others matlab-like time measuring: `double t0= timing::tic(); <do stuff>; std::cout<<timing::toc(t0);`
- util/image_util.h: Can convert images, e.g. png->jpeg
- util/arg_sort.h: Instead of sorting an array it gives the list of indices needed to sort it (i.e. it is the same as `[~, inds]= sort(a)` in matlab)
- util/slow_construction.h: Say you are running a backend which uses a 20 GB iidx and it crashes. Your code (hopefully) automatically restarts it (see [2]), but it takes a while to bring it back up just due to the sheer time it takes to load 20 GB back from disk into RAM. `slowConstruction` helps exactly with this: one can use a `protoDbFile` while `protoDbInRam` is being constructed (it will be slow but at least the backend will work), and when the construction of `protoDbInRam` finishes it is automatically used. See `protoDbInRamStartDisk` in `proto_db.h`, it is used in `v2/api/api_v2.cpp`
- v2/retrieval/unique_retriever.h: Filters a ranked list for equivalence classes. E.g. if one searches a geotagged database, it is useless to return results which are very close together (if you see 1 false result, it's bad to see another 10 which are all 5 meters away from each other). So a simple way to diversify the list is to divide earth into tiles (e.g. 25x25 meters), say that all images in one tile have the same “metalD” and let `unique_retriever` filter it out.

7. References

[1] Relja Arandjelović, “*Conceptual Overview*”, Internal technical report (docs/Conceptual_Overview.pdf), September 2014

[2] Relja Arandjelović, “*Manual for Black Box Usage*”, Internal technical report (docs/Manual_for_Black_Box_Usage.pdf), September 2014