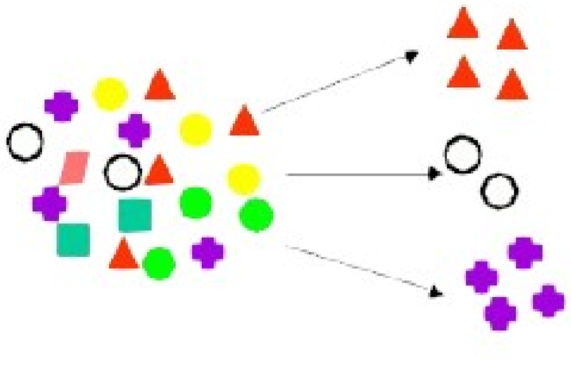

Efficient Evidence Accumulation Clustering for large datasets/big data

Diogo Alexandre Oliveira Silva
Alferes Aluno Engenheiro Electrotécnico 136787-A

Thesis to obtain the Master of Science Degree in

Aeronautical Military Sciences in the speciality of Electrical Engineering

Examination Committee

Chairperson:	Professor Full Name
Supervisor:	Dra. Ana Luísa Nobre Fred
Co-supervisor:	Dra. Helena Aidos Lopes
Member of the Committee:	Professor Full Name 3

Month 2015

Dedicated to someone special...

Acknowledgments

A few words about the university, financial support, research advisor, dissertation readers, faculty or other professors, lab mates, other friends and family...

Resumo

Inserir o resumo em Português aqui com o máximo de 250 palavras e acompanhado de 4 a 6 palavras-chave...

Palavras-chave: palavra-chave1, palavra-chave2,...

Abstract

Insert your abstract here with a maximum of 250 words, followed by 4 to 6 keywords...

Keywords: keyword1, keyword2,...

Contents

List of Tables

List of Figures

Glossary

API	Application Programming Interface.
CPU	Central Processing Unit.
EAC	Evidence Accumulation Clustering.
GPGPU	General Purpose computing in Graphics Processing Units.
GPU	Graphics Processing Unit.
HAC	Hierarchical Agglomeration Clustering.
PCA	Principal Component Analysis.
PC	Principal Component.
QK-Means	Quantum K-Means.
Qubit	Quantum bit.
SL-HAC	Single-Linkage Hierarchical Agglomeration Clustering.
SVD	Singular Value Decomposition.

Chapter 1

Methodology

The aim of this thesis is the optimization and scalability of EAC, with a focus for large datasets. EAC is divided in three steps and each has to be considered for optimization.

The first step is the accumulation of evidence, i.e. generating an ensemble of partitions. The main objective for the optimization of this step is speed. Using fast clustering methods for generating partitions is an obvious solution, as is the optimization of particular algorithms aiming for the same objective. Since each partition is independent from every other partition, parallel computing over a cluster of computing units would result in a fast ensemble generation. Using either or any combination of these strategies will guarantee a speed-up.

Initial research was under the field of quantum clustering. After this pursuit proved fruitless regarding one of the main requirements (computational speed), the focus of researched shifted to parallel computing, more specifically a K-Means parallel version within the GPGPU paradigm.

The second step is mostly bound by memory. The complete co-association matrix has a space complexity of $\mathcal{O}(n^2)$. Such complexity becomes prohibitive for big data, e.g. a dataset of 2×10^6 samples will result in a complete co-association matrix of $14901\ GB$ if values are stored in single floating-point precision. This problem is addressed by exploring the inherent sparse nature of the co-association matrix.

The last step has to take into account both memory and speed requirements. The final clustering must be able to produce good results and be fast while not exploding the already big space complexity from the co-association matrix. The work in this last step was initially directed towards parallelization and afterwards out-of-core processing using a disk stored co-association matrix.

The methodology for optimizing for speed is relevant and permeates the approach taken to every problem faced in the present work. For optimizing the speed of an algorithm, one starts by first profiling said algorithm and analyze which parts take the longest to compute. These parts are the focus of optimization as any change on them will have the greatest effect on the overall algorithm. To further illustrate this point, let us consider an algorithm that spends 75% on a section of the code to which a speed-up of 2 is possible and 25% on a part to which an infinite speed-up is possible (this translates in its execution time being negligible). Let us further assume that only one part of the algorithm may be optimized. If one optimizes the first part, the local speed-up is 2 and the overall speed-up is 1.6. On the other hand, if one optimizes the second part, the local speed-up is infinite but the overall speed-up is only $1.333(3)$.

All the algorithms were implemented in Python with high reliance on numerical and scientific modules, namely NumPy [1] and SciPy [2, 3, 4]. Important modules for visualizing and processing statistical results were MatplotLib [5] and Pandas [6], respectively. The SciKit-Learn [**?**] machine learning library has a plethora of implemented algorithms which were used for benchmarking as well as integration in the devised solutions. The iPython [7] module was used for interactive computing which allowed for accelerated prototyping and testing of algorithms. Python is a interpreted language which translates in its performance being worse than a compiled language such as C or Java. To address this problem, the open-source Numba module from Continuum Analytics was used to allow for writing code in native Python and have it converted to fast compiled code. Furthermore, this module provides a pure Python interface for the CUDA API. The proprietary NumbaPro module was used for two high level CUDA functions, namely *argsort* and *max*.

1.1 Quantum Clustering

Research under this paradigm aimed to find a candidate for the first phase of EAC. Two candidates were considered: Quantum K-Means (QK-Means) and Horn and Gottlieb's quantum clustering (QC)algorithm. These algorithms were chosen so as to have a representative from both approaches identified in chapter **??**.

The implementation QK-Means followed the description presented in section **??**, with the exception of the genetic operations *cross-over* and *mutation*. This change was made because literature [8] suggested that enough variability is produced in the quantum rotation step with the angle-distance rotation method. Furthermore, the main goal for the first phase of EAC is speed but these operations aim at improving the quality of the solution. As such, it was an implementation decision to cut overhead where possible since literature supported that no significant changes would affect the final result.

An implementation of QC was already available in Matlab. Accordingly, implementing this algorithm translated into porting the code from Matlab to Python.

1.2 Speeding up Ensemble Generation with Parallel K-Means

K-Means is an obvious candidate for the generation of partitions since it is simple, fast and partitions do not need to be accurate - variability in the partitions is a desirable property. This relaxation on the accuracy of the partitions already allows for a faster generation of partitions since fewer iterations of the algorithm need to be executed for a partition to be generated - typically 3 iterations suffice. Further gains are possible with a parallel GPU version of K-Means. The overview of the algorithm used is presented in section **??**. This section will offer implementation details.

The implementation gives as much freedom as possible to the user regarding CUDA parameter choices. By default, each thread will compute the label of one pattern and the blocks are unidimensional and composed by 512 threads. The number of blocks, then, is the number of patterns divided by the the number of threads per block. If the number of blocks exceeds the maximum allowed for one dimension (65535), the grid configuration automatically uses other dimensions. The other parameter that the user can choose deals with how data is transfered back and forward between host and device. The user can choose to allow the CUDA API to handle all the memory transfers or to make the implementation optimize those. The latter will minimize the amount of data transfers and memory allocations and is the default option. Furthermore, it also allows for the algorithm to be executed multiple times without redundant transfer of the pattern set. This permits for the production of an entire ensemble with a single transfer of the pattern set. These parameters (number of patterns per thread, number of threads per block and memory transfer mode) may be configured at runtime. Still, a typical user does not need to be knowledgeable about CUDA to be able to use this implementation, since the tuning of these parameters is optional.

The CUDA implementation of the label computation part of the algorithm starts by transfering the data to the GPU (pattern set and initial centroids) and allocates space for the labels and corresponding distances. The computation of a label for one pattern is done by iteratively computing the distance from the pattern to each centroid and storing the label and the distance corresponding to the closest centroid to that pattern. Finally, the labels and distances are sent back to the host for computation of the centroids. This procedure is available as a CUDA kernel or as three different sequential versions: pure Python, based on NumPy or compiled code with Numba. These same sequential versions exist for the recomputation of the centroids. The implementation of the centroid computation starts by counting the number of patterns attributed to each centroid. Afterwards, it checks if there are any ”empty” clusters, i.e. if there are centroids that are not the closest ones to any pattern. Dealing with empty clusters is important because, although empty clusters may be desirable in some situations, the target use expects that the output number of clusters to be the same as defined in the input parameter. Centroids corresponding to empty clusters will be the patterns that are furthest away from their centroids. Any other centroid c_i will be the mean of the patterns that were labeled i.

1.3 Dealing with the space complexity of the co-association matrix

In chapter **??**, two approaches to the space complexity of the co-association matrix in the second phase of EAC are presented: one dealing with p prototypes and the other with the inherent sparsity of the matrix.

The results presented in the sparsity study of EAC [9] show that it is possible to obtain a high sparsity in the co-association matrix. In some comes the density of the matrix was as low as 1%. This motivated the focus of the work on exploiting the inherent sparsity of the matrix. A discussion on using sparse matrices and a novel solution for building the co-association matrix is presented 1.4.

As stated before, the focus of the work is on sparse matrices. Still, for the sake of completeness, the different strategies considered for the prototypes approach are discussed here. One strategy is to use the **p-Nearest Neighbors** as prototypes, as already presented in chapter **??**. Before building the co-association matrix, the p closest samples to each sample are computed and stored in the $n \times p$ neighbor matrix. During the voting mechanism of EAC only the neighbors of each pattern are considered. This strategy has a space complexity of $O(2nk)$ and requires the computation of the k-Nearest Neighbors. A second strategy is to use **p random prototypes**, which will a set of unique p patterns. This will be the same for every sample. Here the voting mechanism is altered so that if a sample is clustered with any of the prototypes, the correspondent element in the co-association matrix is incremented. This has the advantage that only a $n \times p$ matrix needs to be stored along with a p array for the chosen prototypes. Furthermore, if p if high enough to provide a representative sample of the dataset the results can be as good as the full matrix. The third, and final, possible strategy identified is similar to the random prototypes. The difference is that instead of choosing p random samples from the dataset, the prototypes will be the representatives of the dataset from another algorithm, e.g. K-Medoids, K-Means.

It would be ideal to combine both approaches (sparsity and neighbors) to further reduce space complexity, but they are not necessarily compatible. This is specially true for the p-Nearest Neighbors strategy since it is unlikely that a sample will never be clustered with its closest p neighbors. This means that the $n \times p$ co-association matrix may not have many zeros which translates in little advantage for using the sparsity augmentation approach the matrix of co-associations.

1.4 Building the sparse matrix

Building a non-sparse matrix is easy and fast since the memory for the whole matrix is allocated and indexing the matrix is direct. When using sparse matrices, neither is true. In the specific case of EAC, there is no way to know what is the number of associations the co-association matrix will have which means it is not possible to pre-allocate the memory to the correct size of the matrix. This translates in allocating the memory gradually which may result in fragmentation, depending on the implementation, and more complex data structures, which incurs overhead.

Documentation of the SciPy library recommends different sparse formats for either building a matrix or operating over it. For building a matrix, the COO (COOrdinate), DOK (Dictionary of Keys) and LIL (List of Lists) formats are recommended. All of these formats rely on extra data structures such as lists as dictionaries to efficiently build the matrices incrementally, i.e. allocating memory as it is required. For operating over a matrix, the documentation recommends converting from one of the previous formats to either CSR (Compressed Sparse Row) or CSC (Compressed Sparse Column).

As observed before, the sparsity study of EAC [9] did not account for the overhead of using the data structures supporting sparse matrices. However, scaling to very large data sets must take this into account. Exploratory testing revealed that using one of the recommended matrices allowed for building the matrix took around two orders of magnitude as long as using a full allocated matrix. Furthermore, because of the extra data structures, the implementations from SciPy would saturate the main memory from data sets with less than 1 million patterns. Using the CSR format did not saturate memory but it took a completely unacceptable amount of time, many orders of magnitude above any of the formats recommended for building the matrix.

After a search in the literature for efficient and scalable ways to build a sparse matrix proved fruitless, this issues motivated the design of a new method for building the matrix specialized for the characteristics of EAC. The solution devised is based on the CSR format, which has the desired properties of having a low memory trace and allows for fast computations.

1.4.1 EAC CSR

The first step is making an initial assumption on the maximum number of associations *max_assocs* that each sample can have. A possible rule is

$$max\_assocs = 3 \times bgs$$

where bgs is the biggest cluster size in the ensemble. The biggest cluster size is a good heuristic for trying to predict the number of associations a sample will have since it is the limit of how many associations each sample will have in any partition. Furthermore, one would expect that the neighbors of each sample will not vary significantly, i.e. the same neighbors will be clustered together with a sample repeatedly in many partitions. This scheme for building the matrix uses 4 supporting arrays:

- **indices** - an array of $n \times max\_assocs$ size that stores the columns of each non-zero value of the matrix, i.e. the destination sample to which each sample associates with;
- **data** - an array of $n \times max\_assocs$ size that stores all the non-zero values;
- **indptr** - an array of n size where the *i-th* element is the pointer to the first non-zero value in the *i-th* row.
- **degree** - an array of n size that stores the number of non-zero values of each row.

Each sample (row) has a maximum of *max_assocs* pre-allocated that it can use to fill with new associations. New associations that exceed this pre-allocated maximum are discarded. The *degree* array is the one keeping track of the number of associations of each sample. Throughout this section, the interval of the *indices* array corresponding to a specific pattern (row) is referred to as the *indices* interval. If it is said that an association is added to the end of the *indices* interval, this refers to the beginning of the part of the interval that is free for new associations. Furthermore, it should be noted that this method assumes that the clusters received come sorted in a increasing order.

The first partition is inserted in a special way. Since it is the first and the clusters are sorted, it is a matter of copying the cluster to the *indices* array in the positions of each sample, with the exclusion of the sample itself. The data array (containing the co-association score) is set to 1 on the relevant positions. This process can be observed clearly in Fig. 1.1. Because it is the fastest partition to be inserted, it is picked to be the one with the least amount of clusters (more samples per cluster) so that each sample gets the most amount of associations in the beginning (on average). This is only applicable if the whole ensemble is provided, otherwise there is no way to know what is the biggest cluster of the whole ensemble. This increases the probability that any new cluster will have more samples that correspond to already established associations. Because the clusters are sorted and only a copy of each cluster was made, it is not necessary to sort each row of the matrix by column index.

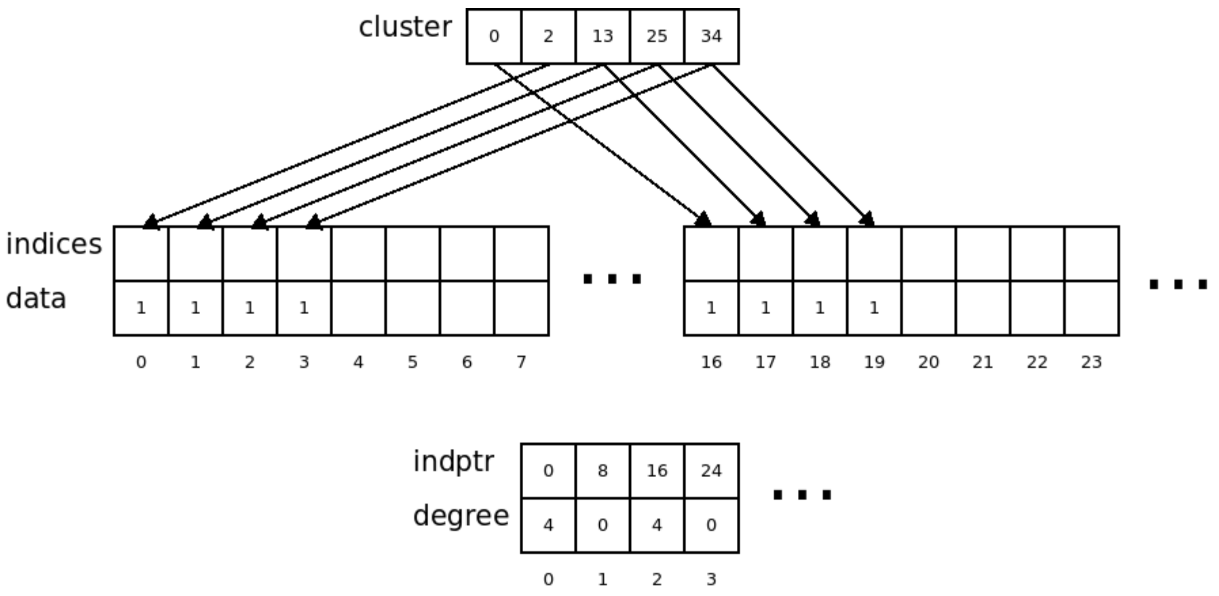

Figure 1.1: Inserting a cluster of the first partition in the co-association matrix.

For the remaining partitions, the process is different. Now, it is necessary to check if an association already exists. For each sample in a cluster it is necessary to add or increment the association to every other sample in the cluster. This is described in Algorithm 1.

Algorithm 1 Update matrix with cluster.

```
1: procedure UPDATE_CLUSTER($indices, data, indptr, degree, cluster, max\_assocs$)
2:     for $sample$ n $in\ cluster$ do
3:         for $every\ other\ sample$ na $in\ cluster$ do
4:             $ind = binary\_search(na, indices, interval\ of\ n\ in\ indices)$
5:             if $ind \geq 0$ then
6:                 $increment\ data[ind]$
7:             else
8:                 if $maximum\ assocs\ not\ reached$ then
9:                     $add\ new\ assoc.\ with\ weight\ 1$
```

The binary search is used to search for the association in the indices array in the interval corresponding to a specific row (or pattern). This is necessary because it is not possible to index directly a specific position of a row in the CSR format. Since a binary search is performed $(ns-1)^2$ times for each cluster, where ns is the number of samples in any given cluster, the indices of each sample must be in a sorted state at the beginning of each partition insertion. Two strategies for sorting were devised. The first strategy is to just insert the new associations at the end of the *indices* array (in the correct interval for each sample) and then, at the end of processing each partition, use a sorting algorithm to sort all the samples' intervals in the *indices* array. This was easily implemented with existing code, namely using *NumPy*'s *quicksort* implementation, which has an average time complexity of $O(n \log n)$

The second strategy is more complex. It results from the observation that, if one could know in which position each new association should be inserted, it would be possible to move all old associations to their final sorted positions and insert the new ones. Furthermore, this would be done in an efficient manner with minimum number of comparisons (and thus branches in the execution of the code). For this end, the implemented binary search returns the index of the searched value (key) if it is found or $-ind - 1$, where ind is the position for a sorted insertion of the key in the array, if it is not found. This means that the position where each new association should be inserted is now available. Instead of just adding the new associations after the old ones of each sample, new associations are stored in two auxiliary arrays of size max_assocs: one (*new_assoc_ids*) for storing the patterns of the associations and the other (*new_assoc_idx*) to store the indices where the new associations should be inserted (the result of the binary search). The process is illustrated with an example in Fig. 1.2, detailed in Algorithm 2 and explained in the following paragraph.

After each pass on a cluster (adding or incrementing all associations to a sample in the cluster), the new associations have to be added to the pattern's indices interval in a sorted manner. The number of associations corresponding to the i-th pattern (*degree[i]*) is incremented by the amount of new associations to be added. An element is added to the end of the *new_assoc_idx* array with the value of *degree[i]* so that the last new association can be included in the general cycle. During the sorting process a pointer to the current index to add associations o_ptr is kept (it is initialized to the new total number of associations of a sample). The sorting mechanism looks at two consecutive elements in the *new_assoc_idx* array, starting from the end. If the i-th element of the *new_assoc_idx* array is greater or equal than the $(i-1)$-th element, then all the associations in the *indices* array between them (including the first element) are copied to the end of the *indices* interval, i.e. they are shifted to the right by i positions. Then, or in case the comparison fails, the $(i-1)$-th element of the *new_assoc_ids* is copied to the *indices* array in the position specified by *o_ptr*. The *o_ptr* pointer is decremented anytime an association is written in the *indices* array.

1.4.2 EAC CSR Condensed

The co-association matrix is symmetric, which means that only half is necessary to describe the whole matrix. That fact has consequences on both memory usage and computational effort on building the matrix. In the case of the fully allocated matrix, this translates in a reduction to 49.5% of the memory required. Furthermore, only half the operations are made since only half the matrix is accessed, which also accelerates the building of the matrix. There is, however, a small overhead for translating the two dimensional index to the single dimensional index of the upper triangle matrix. When using a sparse matrix according to the EAC CSR scheme described above, there is no direct memory usage reduction. However, the number of binary searches performed by inserting new associations is half which has a significant impact on the computation time relative to the complete matrix.

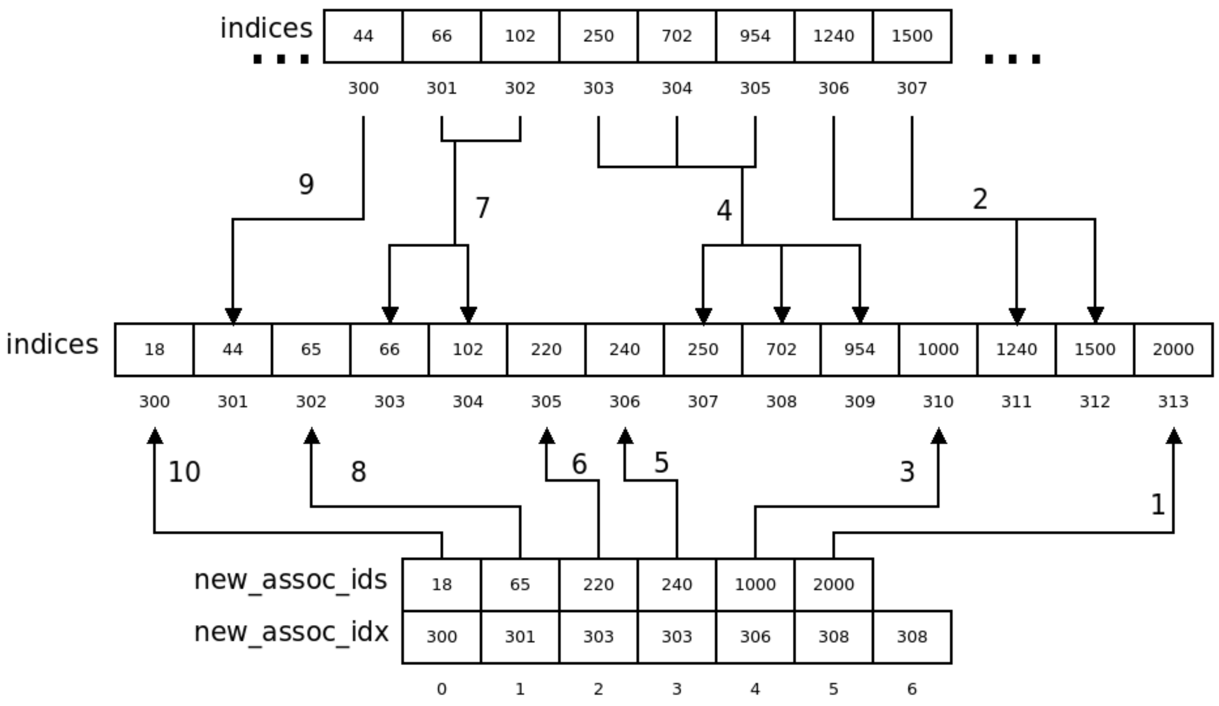

Figure 1.2: Inserting a cluster from a partition in the co-association matrix. The arrows indicate to where the indices are moved. The numbers indicate the order of the operation.

Algorithm 2 Sort the *indices* array in the interval of a sample n.

```
1:  procedure SORT_INDICES($indices, data, indptr, degree, n, new\_assocs\_ptr, new\_assocs\_ids, new\_assocs\_idx$)
2:      $new\_assocs\_idx[new\_assocs\_ptr] = indptr[n] + degree[n]$
3:      $i = new\_assocs\_ptr$
4:      $o\_ptr = indptr[n] + new\_assocs\_ptr + degree[n] - 1$
5:      while $i \geq 1$ do
6:          $start\_idx = new\_assocs\_idx[i-1]$
7:          $end\_idx = new\_assocs\_idx[i] - 1$
8:          while $end\_idx >= start\_idx$ do
9:              $indices[o\_ptr] = indices[end\_idx]$
10:             $data[o\_ptr] = data[end\_idx]$
11:             $end\_idx -= 1$
12:             $o\_ptr -= 1$
13:         $indices[o\_ptr] = new\_assocs\_ids[i-1]$
14:         $data[o\_ptr] = 1$
15:         $o\_ptr -= 1$
16:         $i -= 1$
```

Even though there is no direct memory reduction for switching to the condensed matrix form, this scheme does open possibilities for it. Previously, the number of associations for each sample (number of non zero values in each row) remained constant throughout the whole set of patterns. However, using a condensed scheme means that the number of associations decreases as one steps further to the end of the set, i.e. closer to the bottom the co-association matrix. An example of this can be seen in the plot of the number of associations per sample in a complete and condensed matrix of Fig. 1.3. It is clear that, since the number of associations that is actually stored in the matrix decreases throughout the matrix, the pre-allocated memory for each sample can decrease accordingly. One possible strategy, illustrated in Fig. 1.3, is to decrease the maximum number of associations linearly. In the example, the first 5% of samples have the 100% of the maximum number of associations and it decreases linearly until 5% of the maximum number of associations for the last sample.

Table 1.1 shows the memory usage of the different co-association matrix strategies in the general case and for an example of 100000 patterns. The memory consumption for the *sparse condensed linear* type of matrix if given for the parameters presented above for Fig. 1.3.

Table 1.1: Memory used for different matrix types for the generic case and a real example of 100000 samples. The relative reduction (R.R.) refers to the memory reduction relative to the type of matrix above, the absolute reduction (A.R.) refers to the reduction relative to the full complete matrix.

Type of matrix	**Generic**	**Memory [MBytes]**	**R.R.**	**A.R.**
Full complete	N^2	9536.7431	-	-
Full condensed	$\frac{N(N-1)}{2}$	4768.3238	0.4999	0.4999
Sparse constant	$0.54 \times N \times max\_assocs$	678.8253	0.1423	0.0711
Sparse condensed linear	$N \times max\_assocs$	372.8497	0.5492	0.0390

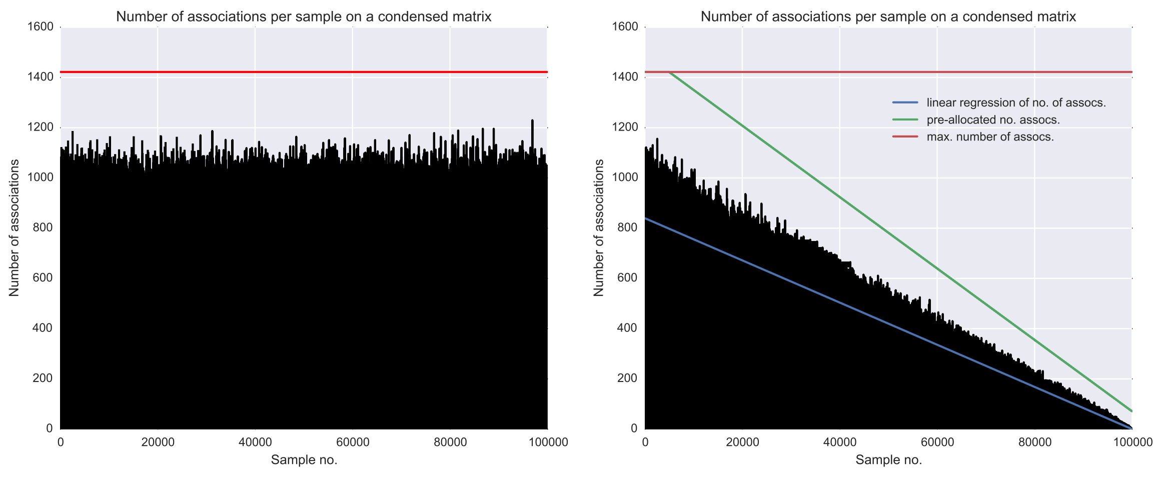

Figure 1.3: The left figure shows the number of associations per sample in a complete matrix; the right figure shows the number of associations per sample in a condensed matrix.

1.5 Final partition recovery

This section will present the considered candidates for the final step of EAC. A GPU version of SL is described in section 1.5.1. External algorithms are a candidate solution and this approach is presented in section 1.5.2. This solution is based on storing the co-association matrix on disk, performing the expensive computation (memory and speed wise) of *argsort* and then processing the matrix in batches until the final MST is extracted.

1.5.1 Single-Link and GPGPU

Single-Link is an inherently sequential algorithm which means an efficient GPU version is hard to implement. However, SL can be performed by computing the MST and cutting edges until we have the desired number of clusters, as described in chapter **??**. Considerable speed-ups are reported in the literature for MST solvers in the GPGPU paradigm. The considered solution uses the efficient parallel variant of Borůvka's algorithm [10]. After the necessary pruning of the MST edges has been performed to achieve the desired number of clusters, the output MST is fed to the labeling algorithm which will provide the final clustering.

Generating the MST

The output of the Borůvka's algorithm is a list of the indices of the edges constituting the MST. These indices point to the *destination* of the original graph. The original algorithm [10] assumes connected graphs, but this is not guaranteed in EAC's co-association matrix. In graph theory, given a connected graph $G = (V, E)$, there is a path between any $s, t \in V$, where V is the set of vertices in G and E is the set of edges that connects the vertices. In an unconnected graph, this is not the case and unconnected subsets are called components, i.e. a connected component $C \subset V$ ensures that for each $u, v \in C$ there is a path between u and v. In the present implementation, the issue of unconnected graphs was solved in the first step (finding the minimum edge connected to each vertex or supervertex). If a vertex has no edges connected to it (an outdegree of 0 since all edges are duplicated to cover both directions) then it is marked as a mirrored edge. This means that the independent components will be marked as supervertices in all subsequent iterations of the algorithm. The overhead of having these redundant vertices is low since the number of independent components is typically low compared to the number of vertices in the the graph and the processing of such vertices is very fast. As a consequence, the stopping criteria becomes the lack of edges connected to the remaining vertices, which is the same as saying that all elements of the *outdegree* array are zero. This condition can be checked as a secondary result of the computation of the new *first_edge* array. This step is performed by applying an exclusive prefix sum over the *outdegree*. The prefix sum was implemented in such a way that it returns the sum of all elements, which translates in very low overhead to check this condition. The final output is the MST array and the number of edges in the array. The later is necessary because the number of edges will be less than $|V| - 1$ when independent components exist, and also because the MST array is pre-allocated in the beginning of the algorithm when the number of edges is not yet known.

Number of clusters and pruning the MST

The number of clusters can be automatically computed with the lifetime technique or be supplied. Either way, a list (*mst_weights*) with the weights of each edge of the MST is compiled and ordered. The list of edges is also ordered according to the order of the *mst_weights*. If the number of clusters k was given, the algorithm removes the $k-1$ heaviest edges. If there are independent components inside the MST those are taken into consideration before removing any edges. If the number of clusters k is higher than the number of independent components the final number of clusters will be the number of independent components.

To compute the number of clusters (which results in a truly unsupervised method) the lifetimes are computed. In the traditional EAC, lifetimes are computed on the weights of the clusters by the order they are formed. With the MST, the lifetimes are computed over the ordered *mst_weights* array, which is equivalent. If there are independent components, an extra lifetime is computed from the representative weight connecting independent components. This lifetime will serve as the threshold between choosing the number of independent components as the number of clusters or looking into the lifetimes within the MST. This is because links between independent vertices are not included in the MST. For this reason, the lifetime for going from independent edges to the heaviest link in the MST (where the first cut would be performed) is computed separately. If the maximum lifetime within the MST is bigger than the threshold, than the number of clusters is computed as in traditional EAC plus the number of independent components. Otherwise, the independent components will be the final clusters. Most of this process can be done by the GPU. Library kernels were used to sort the array and compute the $argmax$, and a simple kernel was used to compute the lifetimes.

Building the new graph

The final MST (after performing the pruning) is then converted into a new, reduced graph. The function responsible for this takes the original graph and the final selection of edges that composing the MST and, afterwards, produces a graph in CSR format. The function has to count the number of edges for each vertex, compute the origin vertex of each edge (the original *destination* array only contains the destination vertices) and, with that information, build the final graph. This process can be done by the GPU with simple mapping kernels and a modified binary search for determining the origin vertex.

Final clustering

The last step is computing the final clusters. Any incision in the MST translates in independent components in the constructed graph. This means that the problem of computing the clusters translates into a problem of finding independent connected components in a graph, a problem that usually goes by the name of Connected Component Labeling. To implement this part in the GPU, the aforementioned Borůvka algorithm was modified to output the an array *labels* of length $|V|$ such that the $i-th$ position contained the label of the $i-th$ vertex. To this effect, the flow of the algorithm was slightly altered as shown in Figure 1.4. The kernel dealing with the MST was removed and a kernel to update the labels at each iteration, shown in Algorithm 3, was implemented. In the first iteration of the algorithm the converged colors are copied to the labels array. In every iteration the kernel to update the labels takes in the propagated colors and the array with the new vertex IDs. For each vertex in the array, the kernel first takes in the the color of the current vertex and maps it to the new color (one should notice that the color is actually a vertex ID and that that vertex has had its color updated). Afterwards, the kernel maps the new color with the new vertex ID that color will take, to keep consistency with future iterations.

Algorithm 3 Update component labels kernel

```
1: procedure UPDATE_LABELS($vertex\_id, labels, colors, new\_vertex$)
2:     $curr\_color \leftarrow labels[vertex\_id]$
3:     $new\_color \leftarrow colors[curr\_color]$
4:     $new\_color\_id \leftarrow new\_vertex[new\_color]$
5:     $labels[vertex\_id] \leftarrow new\_color\_id$
```

Memory transfer with the GPU

The whole algorithm of computing the SL clustering has been implemented in the GPU with minimizing the memory utilization in mind. Transferring the initial graph is the most relevant memory transfer. It has to be transfered twice: first for computing the MST and then to build the processed MST graph. This happens because the initial device arrays used for the graph are deallocated to give space for the arrays of subsequent iterations of the MST algorithm. This implementation design had in mind memory consumption in mind and could easily be avoided with the cost of having to store the bigger initial graph for the entire duration of the MST computation, which might be worthwhile if the GPU memory is abundant. The final labels array is transfered back to the host in the end of computation, but its size is small relative the to the size of the original graph. Furthermore, because the control logic is processed by the host and it is dependent on some values computed by the device, extra memory transfers of single values (e.g. number of vertices and number of edges on each iteration) are necessary. These, however, may be safely dismissed since they are of little consequence in the overall computation time.

1.5.2 Single-Link and external memory algorithms

The co-association matrix can become very large for large data sets. Even using the EAC CSR method to reduce memory usage, the memory used occupy a significant portion of main memory. Furthermore, the algorithm to compute a MST brings additional space complexity that make it infeasible to perform the final partition recovery on a large co-association matrix in main memory. External memory algorithms address this issue. External memory algorithms refers to algorithms that are based on disk (or other large capacity but typically low latency storage technology), usually by loading small batches to main memory, processing them, saving the results and repeat until the whole computation is done. This section describes the process of building the MST with low memory usage. The details of converting the MST in a Single-Link clustering are described in chapter **??** and omitted here.

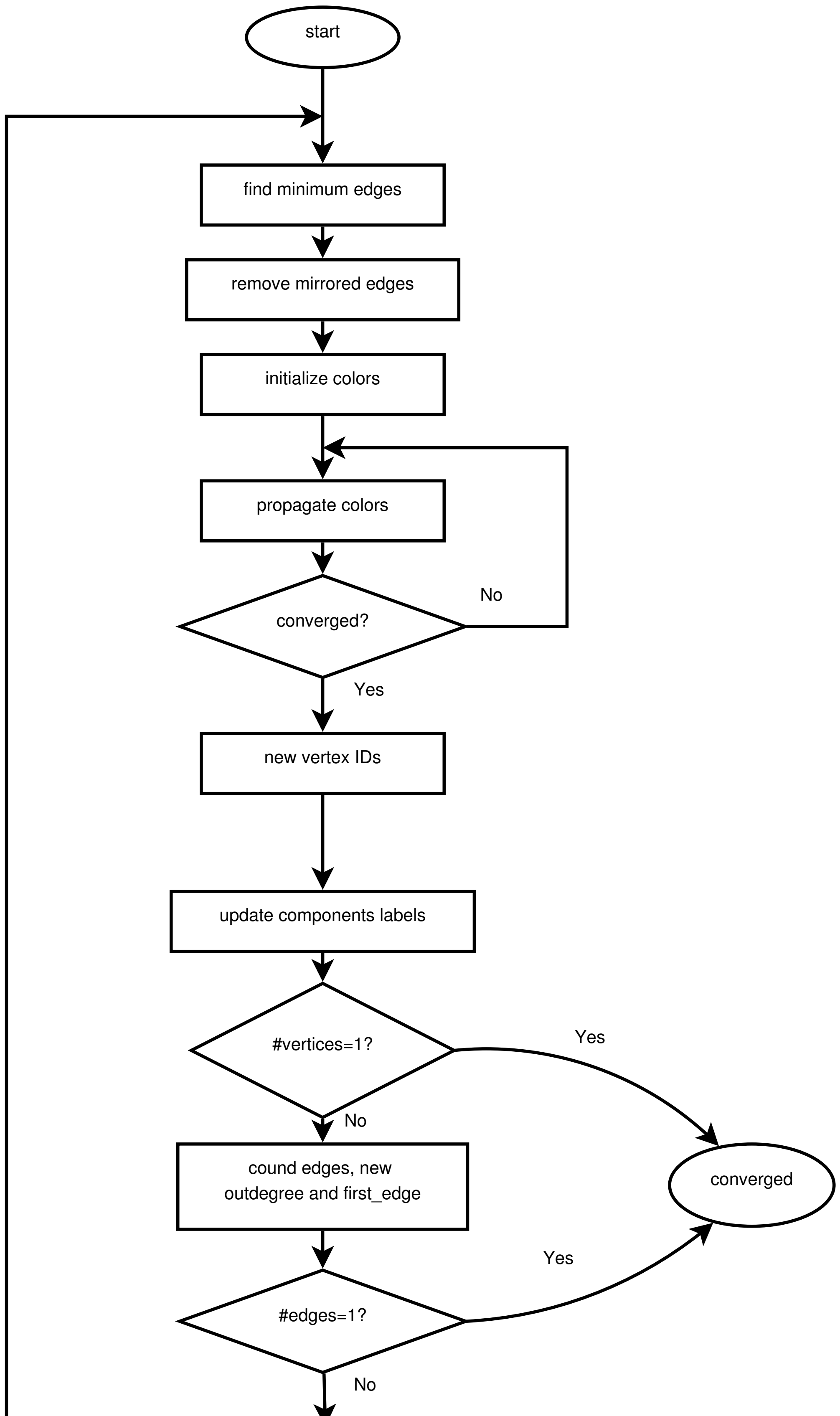

Figure 1.4: Diagram of the connected components labeling algorithm used.

Kruskal's algorithm and implementation

The sequential MST algorithm used was Kruskal. The original paper of Kruskal's [11] algorithm describes three approaches for finding a MST. The SciPy library offers an efficient implementation for the following construct (taken directly from the original paper of Kruskal's algorithm):

> CONSTRUCTION A. Perform the following step as many times as possible: Among the edges of G not yet chosen, choose the shortest edge which does not form any loops with those edges already chosen. Clearly the set of edges eventually chosen must form a spanning tree of G, and in fact it forms a shortest spanning tree.

If the graph if connected, the algorithm will stop before processing all edges when $|V| - 1$ edges are added to the MST. Within the EAC context, it is very common for the co-association graph to have independent components which translates in every edge being processed. It should be noted that this implementation works on a sparse matrix in the CSR format.

One of the main steps of the implementation is computing the order of the edges of the graph without sorting the edges themselves, an operation called *argsort*. To illustrate this, the argsort operation on the array $[4, 5, 2, 1, 3]$ would yield $[3, 2, 4, 0, 1]$ since the the smallest element is at position 3 (starting from 0), the second smalled at position 2, etc. This operation is useful to avoid computing the edge with minimum (*min*) weight in every iteration. Doing so would increase time complexity since a sorting algorithm may have an average time complexity to $O(nlogn)$ (QuickSort algorithm) while a single minimum has a time complexity of $O(n)$. Computing the minimum at every iteration when every edge must be processed means that there will be $|E|$ iterations and so the total time complexity for the minimum operation alone will be $O(E^E)$ if processed edges are only given the maximum value possible or $O(E!)$ if they are removed. These time complexities are much higher than a $O(ElogE)$. The result of this operation is an array of length $|E|$ (i.e. the number of associations in EAC), which means it will occupy at least the same size as the array containing the weights of the edges. However, the total size is typically 8 times larger for EAC since the data type of the weights uses only one byte and the number of associations is very large, forcing a the use o 8 bytes for the argsort array. This is the real motivation to store the co-association matrix in disk and use an external sorting algorithm.

Kruskal's implementation with an external sorting algorithm

The *PyTables* library [12], which is built on top of the *HDF5* format [13], was used for storing the graph, performing the external sorting for the argsort operation and loading the graph in batches for processing. This implementation starts by storing the CSR graph to disk. However, instead of saving the *indptr* array directly, an expanded version is stored instead. The expanded version will be of the same length as the *indices* array and the i-th element contains the origin vertex (or row) of the i-th edge. This way, a binary search for discovering the origin vertex becomes unnecessary.

Afterwards, the argsort operation is performed by building a completely sorted index (CSI) of the *data* array of the CSR matrix. The details of CSI implementation fall outside the scope of this dissertation but are explained in further detail in [14]. It should be noted that the arrays themselves are not sorted.

Instead, the CSI allows for a fast indexing of the arrays in a sorted manner (according to the order of the edges). The process of building the CSI has a very low main memory usage that can be disregarded.

The SciPy implementation of Kruskal's algorithm was modified to work with batches of the graph. This was easily implemented just by making the additional data structures used in the building of MST persistent between calls of the function. The new implementation loads the graph in batches and in a sorted manner, e.g. first load a batch of the 1000 smallest edges, then a batch of the next 1000 smalled edges, etc. Each batch must be processed sequentially since the edges must be processed in a sorted manner, which means that there is no possibility for parallelism in this process. Typically, the batch size is a very small fraction of the size of the edges, which means that the total memory usage for building the MST is overshadowed by the size of the co-association matrix. The time complexity for building the CSI is higher that of computing the argsort operation, but the formal time complexity is not reported in the source [14]. As an example, for a 500000 pattern set the SL-MST approach took 54.9 seconds while the external memory approach took 2613.5 seconds - an order of magnitude higher.

Bibliography

[1] Stéfan Van Der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science and Engineering*, 13(2):22–30, 2011. ISSN 15219615. doi: 10.1109/MCSE.2011.37.

[2] Eric Jones, Travis Oliphant, Pearu Peterson, and Others. {SciPy}: Open source scientific tools for {Python}. URL `http://www.scipy.org/`.

[3] Travis E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9(3): 10–20, 2007. ISSN 15219615. doi: 10.1109/MCSE.2007.58.

[4] K. Jarrod Millman and Michael Aivazis. Python for scientists and engineers. *Computing in Science and Engineering*, 13(2):9–12, 2011. ISSN 15219615. doi: 10.1109/MCSE.2011.36.

[5] John D Hunter. Matplotlib: A 2D graphics environment. *Computing in science and engineering*, 9 (3):90–95, 2007.

[6] Wes McKinney. Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python in Science Conference*, 1697900(Scipy):51–56, 2010. URL `http://conference.scipy.org/proceedings/scipy2010/mckinney.html`.

[7] Fernando Pérez and Brian E. Granger. IPython: A system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, 2007. ISSN 15219615. doi: 10.1109/MCSE.2007.53.

[8] Wenjie Liu, Hanwu Chen, Qiaoqiao Yan, Zhihao Liu, Juan Xu, and Yu Zheng. A novel quantum-inspired evolutionary algorithm based on variable angle-distance rotation. *2010 IEEE World Congress on Computational Intelligence, WCCI 2010 - 2010 IEEE Congress on Evolutionary Computation, CEC 2010*, 2010. doi: 10.1109/CEC.2010.5586281.

[9] André Lourenço, Ana L N Fred, and Anil K. Jain. On the scalability of evidence accumulation clustering. *Proceedings - International Conference on Pattern Recognition*, 0:782–785, 2010. ISSN 10514651. doi: 10.1109/ICPR.2010.197.

[10] Cristiano da Silva Sousa, Artur Mariano, and Alberto Proença. A Generic and Highly Efficient Parallel Variant of Boruvka 's Algorithm. 2015. URL `https://github.com/Beatgodes/BoruvkaUMinho`.

[11] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

[12] Francesc Alted, Ivan Vilata, and Others. {PyTables}: Hierarchical Datasets in {Python}. URL http://www.pytables.org/.

[13] The HDF Group. Hierarchical Data Format, version 5.

[14] Francesc Altet i Abad and Ivan Vilata Balaguer. OPSI : The indexing system of PyTables 2 Professional Edition. pages 1–27, 2007. URL http://www.pytables.org/docs/OPSI-indexes.pdf.