Cornac: Tackling Huge Graph Visualization with Big Data Infrastructure

Alexandre Perrot, David Auber

Cornac: Tackling Huge Graph Visualization with Big Data Infrastructure

Alexandre Perrot and David Auber

Abstract—
The size of available graphs has drastically increased in recent years. The real-time visualization of graphs with millions of edges is a challenge but is necessary to grasp information hidden in huge datasets. This article presents an end-to-end technique to visualize huge graphs using an established Big Data ecosystem and a lightweight client running in a Web browser. For that purpose, levels of abstraction and graph tiles are generated by a batch layer and the interactive visualization is provided using a serving layer and client-side real-time computation of edge bundling and graph splatting. A major challenge is to create techniques that work without moving data to an ad hoc system and that take advantage of the horizontal scalability of these infrastructures.
We introduce two novel scalable algorithms that enable to generate a canopy clustering and to aggregate graph edges. These two algorithms are both used to produce levels of abstraction and graph tiles. We prove that our technique guarantee a quality of visualization by controlling both the necessary bandwidth required for data transfer and the quality of the produced visualization. Furthermore, we demonstrate the usability of our technique by providing a complete prototype. We present benchmarks on graphs with millions of elements and we compare our results to those obtained by state of the art techniques. Our results show that new Big Data technologies can be incorporated into visualization pipeline to push out the size limits of graphs one can visually analyze.

Index Terms—Computer Society, IEEE, IEEEtran, journal, LaTeX, paper, template.

1 Introduction

The beginning of the 21st century marked what is now called the "digital age". It corresponds to an explosion of the available digital storage space, mainly in the form of digital disks and computer hard drives. All this available storage allowed to save all the data previously lost and to consider generating and recording much more. This is the phenomenon called "Big Data".

With it emerged new needs for data storage and analysis and new technologies to fulfill those needs. The amount data collected is such that a centralized storage solution is not suitable, leading to the emergence of data centers, huge warehouses filled with computers ready to store and process huge amounts of data. On the software side also, specialized solutions were designed, mainly in the domain of distributing computing. The leading solution and *de facto* standard solution is *Hadoop*, which combines both a distributed file system called HDFS and a distributed computing framework called MapReduce. The collection of raw data is now mainly done through such systems and directly stored in a distributed environment. With this in mind, classical centralized analysis and visualization solutions would require to fetch all the data locally, limiting their possible application. Using the already distributed nature of the collected data would allow to more efficiently process the data without having to transfer it.

Data collected this way can be of many different types, such as raw text, URLs, images or social media posts. Furthermore, it is not known in advance what information is to be extracted. A popular way to model data without a predefined structure is to use a graph. Social networks, commercial products review, sales and recommendations or financial transactions are good examples. Thus, the arrival of "Big Data" leads to an increase in the number, variety and size of available graphs.

The huge amount of data to be displayed does not only lead to problems in terms of screen resolution, but also in terms of information that the user can acquire in a single visualization. A standard screen has only a few million pixels. This is significantly less than the amount of space needed to accurately represent a graph with several million nodes. Indeed, to be distinguished, nodes should, at the very least, not occupy adjacent pixels. With a lower bound of 9 pixels per node, only 200K nodes could be displayed on a resolution of 1920×1080. Moreover, such a small screen size does not allow all the visual variables associated with graph visualization, such as size, color or shape, to be mapped. When considering edges, this simple observation is reinforced. Although they may cross, edges will need more pixels to be displayed in an adequate fashion. An edge requires at least the screen distance between its extremities. The average distance between two random points on the screen is on the order of the screen width. Thus, without overlap, only a few thousand edges can be drawn on the screen. Considering that edges can overlap and that a good graph drawing algorithm should generate more short edges than long edges, this limit can be mitigated somewhat. With those constraints in mind, a good bound for the largest graph that can be shown on a 1920×1080 screen would be approximately 1K nodes and 10K edges. This shows that big graphs cannot be displayed as-is, even if the technology is able to provide decent framerates.

To visualize graphs with several million nodes and edges, aggregation is necessary. This step should convey information from the underlying full-size graph, while allowing real-time representation and scaling the number of elements to a size that the screen can display.

In this paper, we present a complete method for visualizing huge graphs whose drawing is already computed. We incorporate

- *Both authors are with*
  Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France
  CNRS, LaBRI, UMR 5800, F-33400 Talence, France
  E-mails: {aperrot,auber}@labri.fr

Manuscript received April 19, 2005; revised August 26, 2015.

the Big Data infrastructure into our visualization environment. Our method draws from existing multiscale visualization techniques and uses the Big Data infrastructure, as both a computing and a storage platform, to scale up. This enables us to store, compute and retrieve huge graphs with hundreds of millions of elements. Thanks to the horizontal scalability provided by the Big Data cluster, the only limit to the size of visualized graphs is the size of the cluster. We contribute a fully distributed and scalable multilevel aggregation technique, implemented on Spark. The visualization always takes place with a bounded number of elements on the screen, thus guaranteeing the client performance. The visualization client exploiting this architecture is implemented using Web technologies. It incorporates a variation of Kernel Density Estimation (KDE) edge bundling to reduce visual clutter.

The paper is organized as follows. First, we present previous publications about multiscale visualization and visualization on the Big Data infrastructure. We then explain the method used to generate the levels of detail, giving benchmarks of our implementation on Spark and GraphX. Finally, we explain how we conceived the visualization client to allow interaction with the generated hierarchy and give an example with the *Panama Papers* graph.

2 Previous Work

Multiscale visualization has been widely studied. The core idea of multiscale (also known as multilevel) visualization is to produce several abstractions of the base dataset. Generally, this is accomplished by using a clustering algorithm. This step is often recursive, with the same algorithm being used to generate the next abstraction from the current one. This visualization paradigm is tightly linked to Shneiderman's mantra [3], providing the "overview" and "details on demand" parts.

Quigley and Eades [4] coined the idea that large graphs could be represented at a higher level of abstraction to grasp their structure. They draw graphs using a force-directed algorithm and a recursive space decomposition. This decomposition can then be used as a geometrical clustering, in which each level of the decomposition is a geometrical abstraction of the starting graph. Their technique computes both the drawing and the clustering at the same time, so it does not leave the option to change the drawing algorithm depending on the type of the graph.

Auber et al. [5] proposed the use of this type of visualization for small world networks. They show that the nature of small world networks is well suited for recursive clustering and multiscale visualization. Using an edge strength metric, they produce a clustering by removing weak edges. Each remaining connected component is considered as a cluster. Recursively applying this technique gives multiple abstraction levels. Furthermore, they propose the production of *quotient graphs* from this clustering for easier visualization. This technique would unfortunately be difficult to apply to non-small-world graphs without finding an adequate edge metric.

The adequacy between small-world networks and multiscale visualization was also demonstrated by van Ham and van Wijk [6]. In this work, the graphs are drawn using a custom force-directed algorithm. In the visualization, nodes are rendered as 3D spheres with a constant screen size. When zooming out, nodes draw closer and spheres begin to intersect, creating a visual fusion of nodes and highlighting clusters. This technique is then extended with geometrical clustering based on the Euclidean distance between nodes, enabling several levels of abstraction for a graph. Finally, interaction techniques dynamically change the abstraction shown in an area of interest. Scaling this type of techniques to large graphs requires non-trivial work to produce an acceptable aggregation, since it depends on a custom drawing algorithm.

Similar ideas were extended by Zinsmaier et al. [7]. They proposed the use of *Kernel Density Estimation* (KDE) [8] to enable the multiscale visualization of large graphs. The visualization of the nodes' density highlights the clusters. Edge endpoints are then moved to the nearest local maximum of the density function. The combination of those two techniques creates a visual aggregation of nodes into dense clusters based on geometrical proximity and routes edges between those dense clusters. When zooming in, the clusters fall apart and a fine grained structure is revealed. KDE is known to be a good representation for large point sets, because it does not suffer from overplot. However, its computational cost results in trouble scaling up with the number of points.

Abello et al. [9] proposed a complete system for multiscale visualization. Clustering is precomputed on the desired graph. Users can then interactively explore the graph by expanding and contracting clusters. The system treats the amounts of screen space, main memory and disk space as finite resources, that must be carefully managed. The whole aggregation and visualization process is guided by the amount of resources available. However, this technique recomputes the layout for the visible graph each time nodes are added or removed.

The TugGraph system [10], by Archambault et al., enables the exploration of a graph using a precomputed hierarchy and computes a drawing on the fly for the current state of the visualization. The navigation in the hierarchy is controlled by an antichain in the cluster tree.

More recently, Nachmanson et al. [11] used the online map metaphor to navigate large graphs. They took two elements from online map viewing: tiling and filtering. Tiling is a spatial indexing scheme that is organized as a pyramid of squares with decreasing size and has become popular in recent years. Filtering the graph enables only the most important elements to be shown to the user. The less important elements will be visible when zoomed. This method has trouble scaling to large graphs, with a computing time of 6 h for a graph with 38K nodes and 85K edges.

A comparable tiling scheme was used by Elmqvist et al. [12] to explore adjacency matrices. This spatial decomposition was very adequate, due to the space-filling nature of adjacency matrices.

Useful techniques and guidelines for multiscale visualization have been compiled by Elmqvist and Fekete [13]. In their article, they formalize several concepts stemming from previous publications. Most notably, the concept of *entity budget* emphasizes the need to limit the number of visual entities displayed, both to keep interactive frame rates and avoid overflowing the user's perception.

The increasing size of graphs to be visualized calls for tools outside the classical desktop visualization paradigm. To this end, parallel and distributed computing can help a precomputation phase for multiscale visualization or graph drawing. For example, the commercial system Graphistry [1] uses a GPU cluster to compute graph drawings and analytics. Recently, a few frameworks emerged enabling very fast computations on graphs with a moderate size machine, such as GraphCHI [14], X-Stream [15] or Chaos [16]. While these system achieve very good performance on many graph algorithms, using them to process data stored in

1. https://www.graphistry.com/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2018.2869165, IEEE Transactions on Big Data

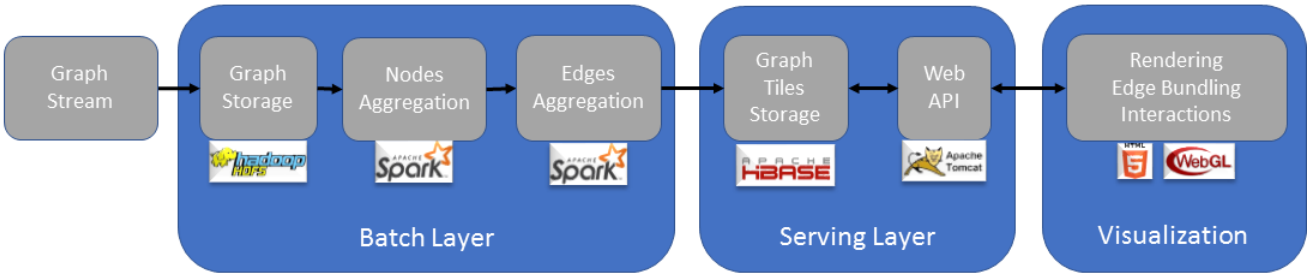


Fig. 1: The system's architecture. Data is first stored in the HDFS distributed file system. Graph aggregation of this data is implemented on top of Spark [1] and GraphX [2]. The result is stored into the HBase distributed NoSQL database. The visualization client requests visible parts of the graph on the fly through a front server. It implements several interaction techniques to reduce visual clutter.

the cloud requires to transfer the dataset on a client computer before reinjecting the result into the cloud storage. Furthermore, a user might not want to install, configure and maintain yet another specialized system for a single purpose.

The Big Data infrastructure has gained popularity in recent years as a general computing platform, especially with the emergence of the Apache Hadoop ecosystem and its implementation of MapReduce [17]. Hadoop v2 introduces YARN (Yet Another Resource Negotiator), which serves as a resource manager for the entire cluster. This opens the Hadoop platform to run many versatile applications.

Perrot et al. [18] have shown how the Hadoop ecosystem can be used to scale existing visualization techniques to handle more data. Namely, they use Spark to precompute levels of detail for a density visualization. The visualization can then be browsed interactively through a Web browser. Their system comes with guarantees toward the amount of data transferred on the network, which helps interactivity.

For graphs more specifically, the Big Data ecosystem has been used for graph drawing. Arleo et al. [19] implemented a spring-embedder [20] on top of *Giraph* and then a multi-level force-directed algorithm [21]. Their algorithm uses edges to pass information about neighbors, using Pregel. The distance that messages are allowed to travel in the graph is a parameter. In the article, results are shown with distances of 2 and 3. They emphasize the small financial cost of their algorithm on popular *PaaS* (Platform as a service) providers, such as Amazon AWS. These services give access to Big Data infrastructure in a few seconds. The total price depends on the amount of resources used and the total time.

Hinge and Auber [22] used Spark to implement a custom force-directed algorithm, with a multi-level approach [23]. Their algorithm is especially designed for the Big Data infrastructure, considering that data is no longer collocated. This entails that computing all node pair distances is very expensive in this environment. To overcome this limitation, the authors rely on an approximate calculation of repulsive forces using centroids as representatives for nodes.

Infrastructures such as Hadoop are especially interesting for visualization systems, as they easily enable the creation of a *visualization-as-a-service* platform. Such a platform would enable many concurrent users to process, store and visualize different big graphs at the same time. Often, in addition to the graph structure, metadata must also be stored for other analysis purposes. Extracting data from the cluster to process it on a single computer would significantly hinder the efficiency of single machines systems. Furthermore, this infrastructure can easily be shared with other applications depending on user needs. Sharing a cluster between several types of workloads has the advantage of providing a unified place for data collection, storage, analysis and retrieval, eliminating the need to extract data from one system to load it into another. This type of shared repository is commonly known as a *data lake*. Finally, using a Big Data cluster comes with out-of-the-box data replication, computation fault-tolerance and enables horizontal scalability (*i.e.*, being able to add new machines to the cluster to process more data).

3 Overview

Our approach aims at providing a way to interactively explore large graphs whose drawings are already computed. This separation between the drawing step and the abstraction generation step allows us to change the algorithms used in one without affecting the other. Thus, our method can be general enough to be applied to many kinds of graphs. Drawing large graphs is outside the scope of this article, so we consider the drawing (*i.e.,* node positions) to be given as an input. It exists a rich litterature on large graph drawing. For instance, Hadany and Harel [24] introduced the basis of multi-scale algorithm, [25], [26], [27] presented several distributed algorithms and [19], [21], [22] presented the first working algorithms on Big Data infrastructure. We refer the reader to the handbook of graph drawing [28] for more information on that topic.

Multiscale visualization has proved its efficiency at handling large graphs [6], [7], [9] and large data in general [18]. We want to adapt the existing techniques to the new sizes of graphs and the new Big Data ecosystem.

We mainly build on the works by Perrot et al. [18], Nachmanson et al. [11] and Zinsmaier et al. [7] to handle large graphs in the Big Data infrastructure. Our technique consists of generating several levels of detail as abstractions of the graph to visualize. To accomplish that, we aggregate both nodes and edges of the graph. Each level generated will then be stored in a distributed database. A lightweight client requests data for visible parts on the fly for interactive visualization. The client is also responsible for rendering the visualization and computing optimizations to reduce visual clutter. Edge bundling is implemented in real time, along with neighborhood highlighting. See Figure 1 for an overview of the whole architecture. Our system is built around Big Data technologies, of which *Hadoop*, whose mascot is an elephant, is the *de facto* standard. The system is thus named *Cornac*, *i.e.* "a person who drives an elephant".

To handle node aggregation, we use geometrical clustering, *i.e.*, grouping nodes considered to be too close to each other. This type of clustering has been used in several publications [4], [6], [7]. We use the same general Canopy Clustering algorithm as in [18], but present a new implementation using well-known distributed graph algorithms. Indeed, the version presented in [18] suffered from load balancing problems and could generate visual artifacts due its canopy selection being separated in two phases. We also implement it using *GraphX* [2], the graph library part of *Spark* [1], [29].

To handle edge aggregation, we combine different techniques. First, there is a natural edge aggregation when nodes are grouped. Edges whose endpoints are grouped together are no longer visible. However, they can be displayed as loops or the node can reflect the structure of the underlying subgraph. Two edges with the same source and target after node aggregation are merged. To ensure a tight bound on the number of edges displayed, long edges are further aggregated.

Each level of detail is then divided into *tiles* for indexing. Tile indexing comes from online map applications and has already been used for graph navigation in [11]. We define our tile indexing as follows:

- Each tile is a square. By convention, a single tile corresponds to a fixed screen area of $256px^2$.
- The top-most level of the aggregation hierarchy is numbered 0 and has only a single tile. Each level is then numbered with increasing numbers.
- Each tile of level $i-1$ corresponds to 4 tiles of level i. Level i thus has 4^i tiles in total.
- The numbering of tiles begins in the bottom-left corner with $(0,0)$ and increases. Level i thus has tiles up to $(2^i,2^i)$.

The lightweight client uses the *Fatum* library [30] to render the graph. *Fatum* has been designed to be able to display a hundred thousand visual elements at 60fps in a web browser, using WebGL. The aggregation process ensures that this amount of data will not be reached, so it ensures a good framerate for the visualization. This leaves time to use costly interaction techniques without compromising the framerate.

4 Graph Aggregation

To generate the levels of detail, the algorithm proceeds in an iterative bottom-up fashion. Levels are numbered in a top-down order, so aggregation starts at level i_{max} and ends with level 0. Every level corresponds to a grouping distance d_i. The process of generating a level can be decomposed into two phases. First, it aggregates nodes closer than d_i. Edges are then aggregated based on the result of this node aggregation. The output is a new graph, where each node and edge is weighted by the number of elements they represent. The whole process is recursive, *i.e.*, level $i+1$ is taken as the input to compute level i. Every step of the process has been primarily designed to be executed in a distributed environment. It is implemented on top of Spark, which was designed to handle iterative computation and is thus well fitted for the aggregation step. Moreover, Spark comes with a graph computing library, called GraphX.

4.1 Geometric aggregation

To be able to explore the entire graph, a certain number of levels of detail must be computed. This number is determined by the extent to which nodes are aggregated at each level. The final objective is to keep no more than N nodes at level 0. Each level i should then have no more than 4^iN nodes. This number N is given as the input to the aggregation process. At each level of aggregation, nodes closer than d_i are grouped. As shown in [18], there is a direct relationship between the number of nodes to keep and the grouping distance. By knowing the bounding box of the graph, it is easy to determine the distance d_0 such that aggregating with d_0 will keep no more than N points. Each subsequent level i will use $d_0/2^i$ as its grouping distance. The total number of levels in the hierarchy is now determined by the minimum distance between two points in the original dataset, which we call d_m. By definition, no two points are closer than d_m, so it is not necessary to aggregate with smaller distances. The total number of required levels is $\lceil \log_2(\frac{d_0}{d_m}) \rceil$. It follows from this formula that it is unnecessary to know d_m exactly; a 2-approximation is sufficient.

When level i has been aggregated, no two points are closer than d_i. This means that the input data for level $i-1$ will also not contain any two points closer than d_i. This property is very important for the ability to maintain good load balancing and guaranteeing that operations run only on a constant number of points. It enables the assumption that several steps of the process are completed in constant time. Without loss of generality, we now describe only the process of generating a single level.

4.2 Node Aggregation

The objective of node aggregation is to group close points by implementing a variation of *canopy clustering* [31], in which the two distances used by *canopy clustering* are equal. To avoid confusion, we call this variation *unit disk clustering*. Perrot et al. [18] proposed an implementation using Map-Reduce and Spark by partitioning the data into strips, and performing local computations in each strip before a merging phase. As explained by the authors, this approach has load balancing problems when many points become clustered in a single strip. Better load balancing could be achieved by knowing the point distribution and adapting the strip partitioning accordingly. However, the authors use it only on geographical data, in which the point distribution can be assumed to be sufficiently uniform.

We propose a novel algorithm for *unit disk clustering* whose load balancing is not sensitive to the point distribution. It is thus better suited for graph visualization. Indeed, in the context of our algorithm, no assumption can be made about the point distribution, because it depends on the drawing algorithm used. This version can be decomposed into two phases: finding pairs of points that are closer than d_i and choosing representatives meeting the two following criteria. Representatives must be separated by at least d_i. In addition, non-representatives must not be farther than d_i from at least one representative.

The problem of choosing representatives in a set of points based on Euclidean distance is a spatial problem. However, it can easily be transformed into a graph problem. Finding pairs of points closer than d_i corresponds to a unit disk graph construction. A *unit disk graph* (UDG) is a graph whose edges link vertices if they are closer than a unit distance. With the unit distance set to d_i, the edges of the unit disk graph are the desired pairs of points. Once the graph is computed, the representative election step directly maps to a well-known graph problem: finding a *maximal independent set* (MIS). In a graph, an independent set is a set of vertices such that no two neighbors are in the set. It

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2018.2869165, IEEE Transactions on Big Data

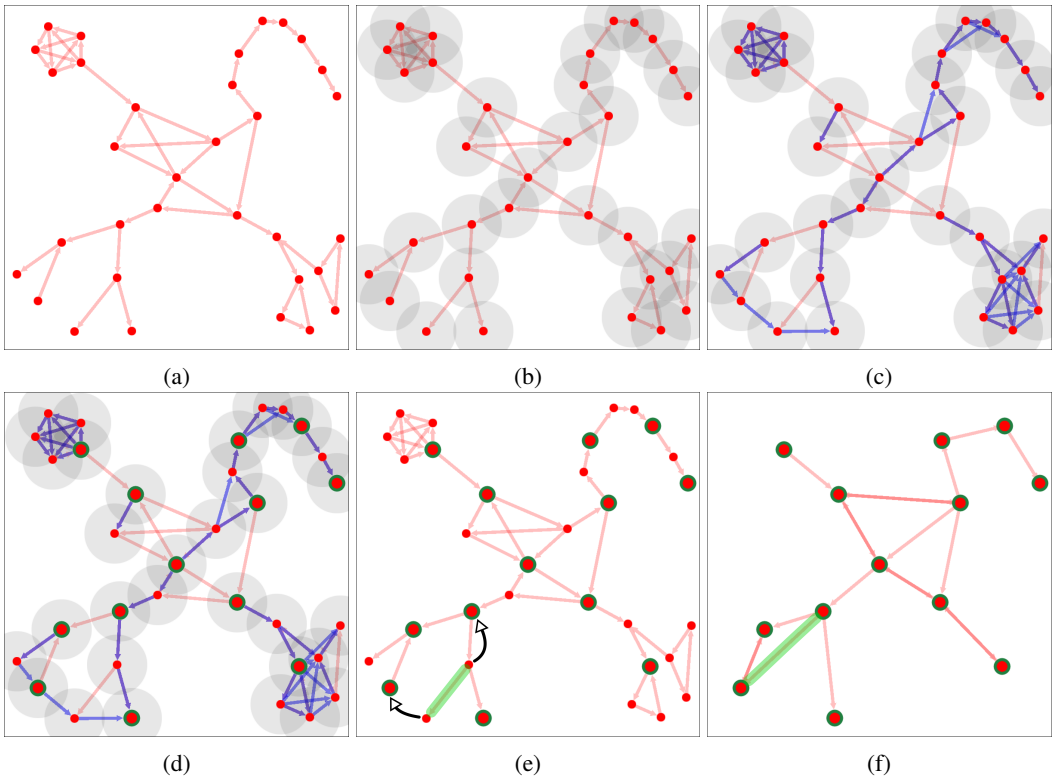


Fig. 2: The steps of the node aggregation process. The aggregation distance is exaggerated compared to a real case. (a): The original graph to aggregate. It contains some particular subgraphs : a clique, a tree and a chain. (b): Unit disks on top of the graph. The radius of the disks is half of the aggregation distance. A conflict arises when two disks intersect. (c): The unit disk graph generated on top of the original graph. Blue edges represent pairs of nodes closer than the aggregation distance. Notice that some edges did not exist in the original graph. (d): Result of the Maximal Independent Set algorithm for representative election. Bigger nodes are representatives. (e): The same MIS, shown on the original graph. Notice that both extremities of the hightlighted edge in the bottom left corner are aggregated on a different node, leading to the creation of a new edge. (f): The final aggregated graph. Only vertices in the MIS have been conserved. Edges have been merged accordingly. Notice the aggregation of the subgraphs.

is *maximal* if the addition of any vertex to the set violates this property. An MIS satisfies the criteria for *unit disk clustering*. Because it is an independent set, the representatives will be further apart than d_i, and because it is maximal every node is closer than d_i to a representative.

Transforming the original spatial problem into a graph problem enables remaining in a graph environment, and the implementation will benefit from distributed graph processing optimizations. Figure 2 shows an overview of the node aggregation process.

As in [6], the graph topology is not considered for the aggregation. It has already been considered when computing the drawing of the graph. Thus, the node aggregation is purely geometrical. Overplotting in the layout will be reflected in the aggregation by merging unconnected nodes. However, this ambiguity was introduced by the layout algorithm and would also have been present when visualizing the entire non-aggregated graph. This also means that the generated hierarchy is not path preserving [10].

4.2.1 Implementation

Unit disk graph construction: To build the unit disk graph, we detect the pairs of points closer than the unit distance d_i. The detected pairs will constitute the edges of the unit disk graph. To minimize the number of distance calculations, we partition the points using a regular grid. The resolution of the grid is d_i. Now, it is necessary only to compute distances with points that fall into the same grid cell, or one of the eight neighboring cells. Because no two points are closer than d_{i+1}, it is straightforward to show that there are no more than a constant number of points in those 9 cells. The total number of cells generated is $O(n)$, but they can be treated in parallel and independently.

In a distributed environment, it is inefficient and difficult to access neighboring cells. Instead, points are duplicated into the nine cells needed, such that each cell contains all necessary information for the distance calculation. For each cell, distances are computed between all pairs of points assigned to the cell. Because there is only a bounded number of such points, this operation is in $O(1)$ time complexity. The pairs of points closer than d_i form the set of edges to build the unit disk graph. The total time to build the graph is $O(\frac{n}{k})$ when distributing on k nodes.

Maximal Independent Set: Maximal Independent Set selection has received a great deal of attention, and many parallel and distributed algorithms exist to solve it. The best-known algorithms are described by Luby [32] in the PRAM (Parallel Random Access Machine) computation model. The key idea of those algorithms is to send messages about the neighbors through the edges. Each vertex can then decide if it can safely be selected. MIS has also been widely studied in the context of distributed algorithms. In particular, we use a variation of Luby's PRAM algorithm in a distributed environment by Métivier et al. [33] in which each node generates a random number $\in [0,1]$ at each iteration. A node can

then be added to the MIS if its value is a local minimum. This version is known to take $\log(n)$ iterations on average to find an MIS.

The way the algorithm works by passing messages between vertices and making decisions at nodes is very similar to one of the leading distributed graph processing paradigms, called Pregel [34], which is based on the *Bulk Synchronous Parallel* execution model [35]. Pregel is used to conduct iterative computations on a graph. The user provides functions to generate messages that are applied at each edge and to aggregate messages at each vertex. For this reason, Pregel is often designated by the phrase "Think like a vertex". An implementation of the Pregel model is available in the GraphX library [2]. The proximity between Métivier's algorithm and Pregel makes it easy to implement in this paradigm. However, it is not straightforward. Indeed, the original algorithm works with several types of messages and phases, which is impossible in Pregel. To implement this algorithm in Pregel, the phases must be fused into a single unified phase. Note that this will not changed the expected number of iterations and exchanged messages.

First, we define the selection state for a vertex. Each vertex can be in one of three states. It is *unselected*, *selected* or *has a neighbor selected*. Of course, every vertex starts in the *unselected* state. For every iteration, the messages passed from each vertex to its neighbors are composed of the current state of the vertex and the random number generated by this vertex for the current iteration. An *unselected* vertex changes its state to *selected* if its random number is less than all the random numbers from its neighbors. A *non selected* vertex also changes its state to *neighbor selected* if one of its neighbors sends *selected* as its selection state. Messages are generated only towards *unselected* vertices, because only those vertices are still active in the process. The process stops when no more messages are generated. Once the selection is complete, *unselected* vertices are filtered. Their weight is then added to the closest representative. This gives a subset of the original points, which does not contain nodes that are neighbors in the unit disk graph. This means that it does not contain nodes closer than d_i.

4.3 Edge Aggregation

First, there is a straightforward edge aggregation when nodes are grouped. Edges between grouped nodes are transformed into loops or are not shown at all. There can also be multiple edges between the same vertices, which are merged into a single weighted edge.

Unfortunately, aggregation by edge merging does not reduce the bound on the number of edges, which can still be considerably high compared with what can be transferred on the network and rendered at interactive framerates. Indeed, by keeping N points, the number of edges can be as high as N^2. Furthermore, when zooming in the visualization, the number of nodes on the screen is bounded, but the number of edges is not. A visible node can be linked to every other node in the graph. This can lead to every edge in the graph being displayed, even on detailed views. Ensuring a tighter bound on the number of displayed edges is essential to scale to bigger graphs in terms of both technical capability and visual clutter.

Angular aggregation for long edges: We distinguish two types of edges: short and long edges, as detailed on Fig 4. Short edges are not aggregated further. Long edges are edges whose endpoints will not both be visible at the same time. The number of short edges is already bounded by the number of visible nodes. Since there is a constant number of visible nodes, the number of short edges is $O(n)$, with respect to the total number of nodes. Long edges, however, do not have a tight bound, because they can exist between every vertex. The problem is that long edges introduce visual clutter and convey less information than short edges do. The only information that can be visually derived from long edges is that there is a connection between a visible node and some other invisible node in the direction of the edge. Furthermore, many long edges heading in the same direction do not add information, compared with a single one of those edges. The only important information from long edges that must be kept is their rough direction.

For this reason, it is useful to aggregate long edges by angular sector. The number of sectors can be selected as an input to the algorithm and should be a small constant. There will be only a single representative edge selected in each sector. Representative edges follow the same criteria as representative nodes, but use angular distance rather than Euclidean distance. If we consider S angular sectors, this means that two representatives will be separated by at least $2\pi/S$ radians and that every edge will be separated from its representative by less than this angle. In addition, there are no more than S long edges per node. The weight of the representatives will reflect the number of edges aggregated. This technique is similar to canopy clustering but considers angles instead of Euclidean distance. This is why we call it *angular canopy clustering*. One advantage of using a representative is that the resulting edge already exists. Thus, edges are only removed, and no new edge is being added by the angular aggregation, avoiding confusion by hinting at a non-existing connection.

For a given node, no more than one short edge per node will be visible at the same time. The number of short edges per node is thus $O(N)$. The number of long edges per node is bounded by the number of angular sectors chosen. In total, there are no more than $O(N+S)$ edges per node. Thus the total number of edges in the graph for a given level is $O(n(N+S)) = O(n)$.

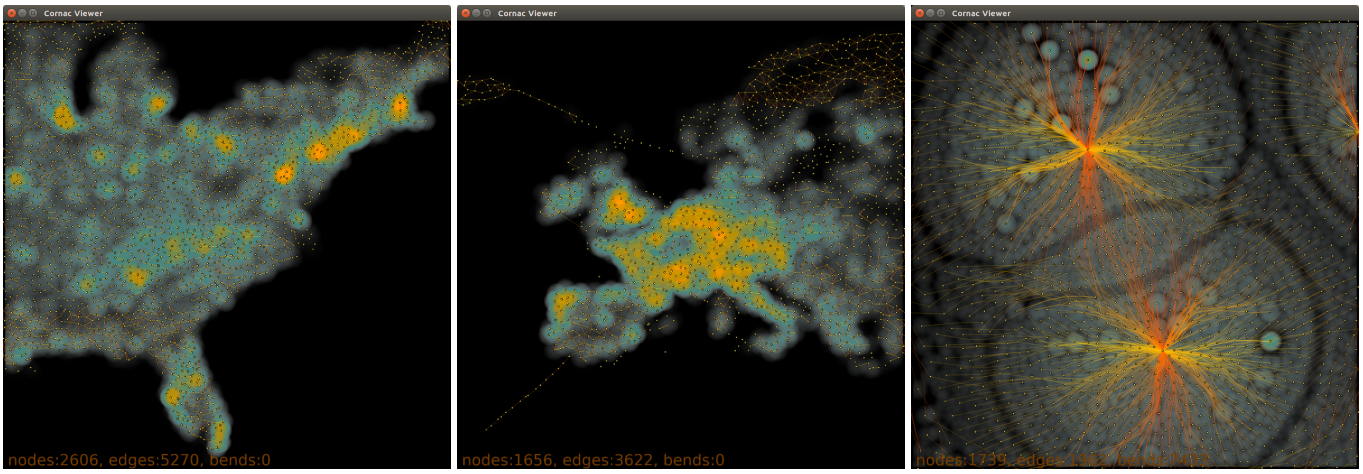
5 Benchmarks

To test the efficiency and scalability of our approach, we ran the full distributed aggregation step on several graphs. We have selected the four biggest graphs from the 9th DIMACS challenge[2]. They represent the road network of the USA. We were also able to obtain the graph of the full European road network in the same format[3], based on data from OpenStreetMap[4]. Table 1 shows the number of nodes and edges of those graphs, along with the number of levels generated with $N = 1000$. Using such geographical graphs for our benchmarks enables us to verify the validity of our approach, since the expected output is well-known.

We ran the benchmarks on our lab's Big Data infrastructure, composed of 16 computers. Each computer has 64GB of RAM and 2×6 hyperthreaded cores running at 2.1GHz and is linked by a 1Gb/s network. During program execution, only 15 computers are used as worker nodes, the last one being the master node. Testing the horizontal scalability of our algorithm was achieved by varying the number of executors used for each run. In *Spark*, an executor is the worker program. A Spark job can be launched with a set number of identical executors. An executor can use several processors of the host machine. This enables scalability to

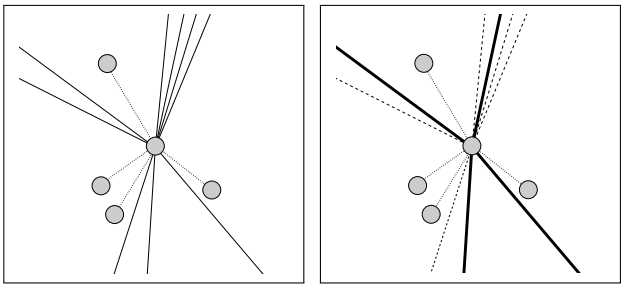
2. http://www.dis.uniroma1.it/challenge9/download.shtml
3. http://i11www.iti.uni-karlsruhe.de/resources/roadgraphs.php
4. http://www.openstreetmap.com/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2018.2869165, IEEE Transactions on Big Data



(a) (b) (c)

Fig. 3: (a): Overview the east part of the full road network of the USA (USA-FULL). Big cities, like New York and Chicago clearly appear. In this type of network, a bigger density of nodes corresponds to more road intersections, which is to be expected in big cities. (b): View of the full road network of Europe (OSM-EUROPE). While the original graph is quite big, the visualization is still interactive thanks to the aggregation step. (c): Two star structures in a web graph from [36]. The bundling and edge color help emphasize the star structure. One can also see nodes linking the two stars in the middle.



(a) (b)

Fig. 4: (a): Distinction between the short and long edges. Only edges of the central node are shown. Short edges have both endpoints visible, wheras long edges extend to a more distant part of the graph. Long edges have a lower angular resolution and induce visual clutter. (b): Result after long edge aggregation. Only long edges in bold are conserved. They convey the same information, *i.e.*, connectivity with a distant part of the graph, but visual clutter is reduced.

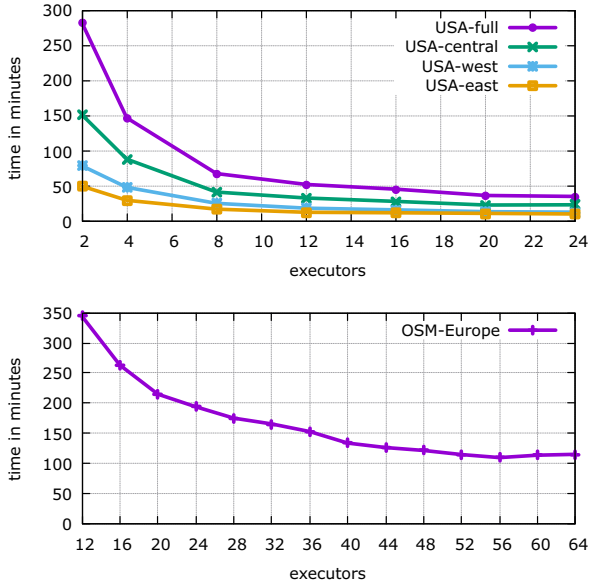


Fig. 5: Results of the benchmarks on five road graphs. This tests both horizontal and vertical scalability. Each executor is configured with 2 cores and 4GB of memory. On the five graphs, the horizontal scalability of our algorithm is clearly visible. Depending on the size of the graph, a threshold is reached. The vertical scalability is also visible, because the graph size is roughly doubled for each USA graph. The Europe graph requires more resources to compute, but exhibits the same scalability.

be tested more finely than by adding computers to the cluster. In this benchmark, each executor has 2 cores and 4GB of memory.

Figure 5 shows the results of the benchmarks for the five graphs. Time includes the full aggregation step and insertion into the HBase table. Due to its size, we used more executors for the OSM-Europe graph benchmark.

First, we can see that our algorithm has near-perfect horizontal scalability. Before reaching a threshold, doubling the number of executors enables the computing time to be halved. A very good vertical scalability can also be observed, because doubling the graph size with the same number of executors doubles the computational time. Those observations result from the great care that was taken to ensure that many steps run in $O(n/k)$.

The very good scalability exhibited does not prevent the computational time from reaching a threshold after which adding more computers does not induce any significant benefit. However, as the graph size increases, this threshold is reached with a larger number of executors. This indicates that our algorithm can scale to much larger data size, provided that the resources scale linearly.

Precomputing the whole aggregation pyramid for the entire graph affects the size of data produced, as seen on Table 1. Each tile of data is stored as text data, compressed using gzip.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2018.2869165, IEEE Transactions on Big Data

graph	nodes	edges	levels	raw	hbase
USA-EAST	3.5M	8.7M	19	258MB	5.4GB
USA-WEST	6.2M	15.2M	20	462MB	11.3GB
USA-CENT.	14M	34.2M	20	1GB	24GB
USA-FULL	23.9M	58.3M	21	1.6GB	36GB
OSM-EUR.	174M	348M	22	11.5GB	202GB

TABLE 1: Number of nodes and edges of the graphs considered in the benchmarks, as well as the number of levels generated with N=1000. Raw data is the size of the original dataset on disk. The hbase column presents the total size of the levels of aggregation, when inserted in HBase. The ratio of aggregated data to raw data is around 20 for every graph. Storing the entire tile pyramid is what enables fast interactive visualization.

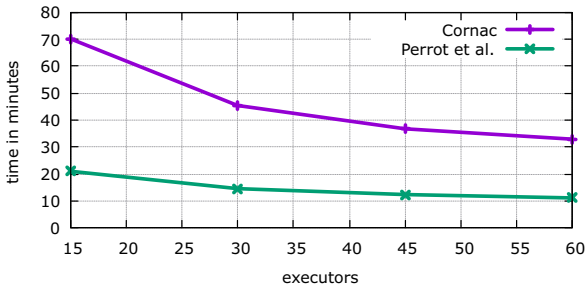


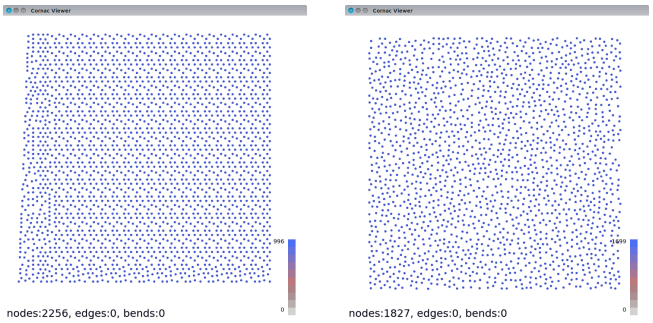
Fig. 6: Comparison of the algorithm from [18] and Cornac. In this benchmark each executor is configured with 2 cores and 8GB of RAM. Cornac is on average 3 times slower with equal resources. However, this only affects the clustering time in the Batch layer, which needs to be computed only once, and not the request time in the Serving layer. Furthermore, Cornac has a more visually appealing result, as shown on Fig. 7

Gzip compression is a good tradeoff between compression ratio and compression/decompression speed for text data. The storage overhead for the aggregated data is still significative, but does not increase with the size of the dataset. The ratio of aggregated data to raw data is around 20 for every graph, with small variations due to the shape of each graph.

Discussion: Here we compare our approach with the current state of the art for Canopy Clustering using Hadoop by Perrot et al. [18] and show what trade-offs between clustering time and the final result aspect were made.

To compare the two algorithms, we used the OSM-EUROPE dataset. We ran the algorithm from [18], and compared it to Cornac's node aggregation with the same resources. Fig. 6 shows the results of the comparison. When it comes to pure time performance, our implementation is 3 times slower than [18], using the same resources. This is easily explained by the fact that the algorithm from [18] was designed to minimize the number of data shuffles (i.e. network transfers), while Cornac's MIS computation requires several Pregel iterations and data shuffles. Furthermore, Cornac also has to compute the UDG and store it alongside the original graph, requiring more memory to process the same data.

The main motivation behind working on a new distributed algorithm for canopy clustering was that the algorithm presented in [18], while fast and producing correct output, generates artifacts in the aggregation due to the way it partitions the data using horizontal strips and selects canopies in two steps. These artifacts only appear with a high concentration of points, thus we used a complete grid of points to show the effect produced by both



(a) (b)

Fig. 7: Results of different versions of canopy clustering on a grid of $100K \times 100K$ points. (a) : Result using the algorithm from [18]. The data partition in strips is clearly visible in the resulting point distribution. (b) : Result of our algorithm. The final point distribution looks much more natural.

algorithms. Fig.7 shows the extent of these artifacts on grid of $100K \times 100K$ points and compares the results of [18] with our new algorithm. One can see that the previous canopy clustering produces a result where the data partition in strips is still visible in the result, whereas our algorithm produces a more visually pleasing and uniform point distribution. The randomness used in the MIS phase makes the result of this algorithm similar to a Poisson-disk sampling process [37], which is often used when a more "natural" looking point distribution is desired. These artifacts were not a problem for [18], since the final visualization only displayed the density function and not the points themselves. However, for a graph visualization use case, the final representative points will be displayed and such artifacts are not desirable.

6 VISUALIZATION

The final step of the technique is to render the graph on the client side and to enable the end user to interact with the visualization. As mentioned in previous sections, the screen resolution enables only thousands of nodes to be rendered in a comprehensive way. Rendering that number of nodes may require, in the worst case, millions of edges to be displayed. In our technique, the client proceeds in two steps. First, it determines which tiles and which level of detail must be downloaded from the server, in order to only have a few thousand nodes rendered on screen. Second, the client must handle as well as possible the edge-edge/edge-node clutter because of the possibly quadratic number of edges and also show the aggregation of elements.

6.1 Retrieving tiles

To query the correct tiles, the client must select which level of detail to use. Let $m(x_{min}, y_{min})$, $M(x_{max}, y_{max})$ be the axis-aligned bounding box (AABB) of the entire layout of the graph and $m_v(x^v_{min}, y^v_{min})$, $M_v(x^v_{max}, y^v_{max})$ be the AABB of the part of the graph one wants to display on screen. Let $w = max(x_{max} - x_{min}, y_{max} - y_{min})$ and $ws = max(x^v_{max} - x^v_{min}, y^v_{max} - y^v_{min})$ be the longest side of each AABB, respectively. The level of detail to display is $l = \lfloor \log_2(w/ws) \rfloor$, and the index (i, j) of the tile that contains m_v (resp. M_v) is $i = (x^v_{min} - x_{min}) * 2^l / w$ and $j = (y^v_{min} - y_{min}) * 2^l / w$. Using that index (l, i, j), one can query the necessary tiles to render the graph.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TBDATA.2018.2869165, IEEE Transactions on Big Data

To limit the number of queries and improve interactivity of our visualization, each requested tile is cached in the client memory. Before querying the front server, the local tile cache is queried. The tile cache policy is straightforward: fill until it is full, and then clear all tiles not currently on screen. This ensures the simplicity and efficiency of the code handling for the cache to care only about filling tile information. Because the cache is filled using data transferred on the network, the rate at which it is filled is limited by the network's speed. For example, with a 1MB/s connection fully dedicated to transferring tiles, it will take more than 1 h to fill 4GB of memory. In practice, tiles are very light, being only a few kilobytes of text data, and are served faster than any human user could ever scroll to trigger tile queries. In our experiments, our cache never uses more than 1GB; that simple cache policy was optimal because we must never free the cache during the graph exploration.

In the case of very connected graphs, the number of edges to be displayed can significantly hinder the tile retrieving time and visualization rendering, despite the aggregation on the server side. Thus, we chose to limit the number of edges to be transferred to client. Similarly to what was done in [11] and [7], only the n edges of each tile that represent the most edges of the original graph are transferred. Since edges are already sorted according to this parameter in the database, it is sufficient to take the first n edges of each tile.

6.2 Rendering tiles

All geometric aggregation of nodes and edges is delegated to the Big Data cluster. After retrieving data from the front server or from the local cache, the client must produce the visual representation of the graph. For each node or metanode, we have the number of nodes it represents (noted n_{count}) in the entire dataset and its position (noted n_{pos}). For each edge or metaedge, we have the number of edges it represents e_{count} and the positions of its extremities. Furthermore, for each level of detail, we have the maximum and minimum values of n_{count} noted $n_{maxcount}$ and e_{count} noted $e_{maxcount}$.

Nodes rendering: The aggregation strategy that is used guarantees a minimal distance between two nodes in each level. This means that we have a guaranteed number of pixels separating two nodes. We thus have a guaranteed number of pixels on the screen to render a node without introducing node-node overlap. Furthermore, since we change the level of details according to the zoom level we used (i.e, part of the layout that is projected on the screen), the minimum available space in pixels is constant.

The aggregation strategy used leads to a more or less uniform distribution of nodes in areas of the graph where enough nodes are present. This can give a false impression of the density of nodes. To alleviate this problem, we render a heatmap of the node density in the background. The weight of each node corresponds to the number of nodes in the orginal graph it represents. It was shown in [38] that the density function computed with canopy clustering is a good approximation of the original density function, as demonstrated in [18]. There are many possible ways to parameterize the rendering of a heatmap. We chose to use the *Epanechnikov* kernel, since it more adequately shows the limited influence of nodes than a more classical gaussian kernel. For the color mapping scale, yellow always corresponds to the maximum node wheight shown on screen. This ensures that the color mapping can display the full range of values on screen.

Edge rendering: Even with the heavy aggregation of nodes and edges produced by the cluster and the limitation of the number of transferred edges per tile, several thousand edges may have to be rendered. This may lead to a large amount of edge/edge and node/edge clutter that complicates the visualization task. We use an edge bundling technique to limit this clutter. Edge bundling consists of aggregating parts of edges together to create bundles. By creating bundles, the number of pixels used to render edges is reduced. The visual effect is a reduction of edge/node and edge/edge clutter. Several methods for general graphs have been developed since the work on edge contraction first proposed by Dickerson et al [39] and in the fow map layout [40] paper. These techniques are now known as edge bundling techniques since the work of Holten [41]. There are now many different approaches going from force directed algorithm [42], geometric decomposition [43], edge routing [44] to kernel density estimation [45]. Our prototype does not manage directed graphs. Edge orientation can be easily computed during the aggregation step. The challenging task is to represent them efficiently. Recent advances in directional edge bundling [46] may be used to make our technique fully usable on directed graph.

For our problem, none of the existing algorithms can be precomputed on the Big Data cluster. First, these algorithms have not been designed to work on that infrastructure. Second, the use of precomputed bundling will significantly increase the amount of data to store and transfer for each tile, which is the bottleneck of our approach. Finally, one of the interests of several bundling techniques is that one can adjust the level of bundling applied to the graph. Even with a large cluster, it may be impracticable to store all tiles for each parametrization of the bundling algorithm. Therefore, we designed a bundling algorithm that could be computed on the client side. Because all tiles are loaded asynchronously, the algorithm must be called again each time the edges and nodes change on the screen. Thus, the algorithm has to be fast enough to be executed each frame.

Our algorithm is a simplification of the algorithm of Hurter et al [45]. In its original version the algorithm proceeds in four steps. It first adds bends uniformly along the edges (**step I**), then computes a discretized density function (a weighted grid) of all of these bends (**step II**) and then moves bends in the direction where the density function is the most important (**step III**). Finally, the algorithm smooths bend positions (**step IV**). Applying the algorithm several times enables to adjust the level of bundling. CPU and GPU resources on the Web browser did not enable steps II and III to be applied quickly. For **step II** we only compute for each cell $c_{i,j}$ of the grid the barycenter $b_{i,j}$ of all bends that are in $c_{i,j}$ and in adjacent cells $c_{i\pm1,j\pm1}$. We then move bends inside $c_{i,j}$ in the direction of $b_{i,j}$. To reduce edge/node clutter, we also added an obstacle constraint by modifying the barycenter of a cell if it contains an original node. In our prototype we use a grid of size 128. Due to the aggregation step, two nodes cannot be in the same grid cell. This enables us to bundle graphs of thousands of edges in less than 100 ms. We are able to re-bundle the graph at each frame during zoom and pan operations. Recomputing the bundling at each frame introduces some instability in edges' bends position. To limit the instability, we first clip all edges with the border of the screen. We then force three consecutive bends on an edge to form an angle greater than $\Pi/2$ during the bundling phase. Furthermore, the weights of edges are reflected using alpha transparency and a logarithmic scale. Exemples of the bundling result are visible on Fig.3c and 9.

Furthermore, even with a few edges displayed on screen, it can be difficult to distinguish different edges. To ease the visual separation of edges with different directions, we change slightly the color of each edge depending of its direction, as can be seen on Fig.3c and 9. This eases the visual disctinction of edges based on direction.

Implementation: The visualization client is implemented in Javascript, using the Fatum library [30]. Fatum handles the rendering with WebGL and camera management and enables thousands of nodes and edges to be rendered on a Web browser. It also includes an occlusion removal system which enable to keep the same labels visible across the different levels if possible [47]. The bundling algorithm has been implemented in C++ and then ported to Javascript using the Emscripten [48] compiler. We developped the experiments with the prototype and were able to navigate into graphs with hundreds of millions of edges as if the graphs were stored on the local computer.

7 Case Study: Panama Papers

7.1 Dataset Overview

The *panama papers* graph is built from 11.5 million documents describing offshore companies, for a total of 2.6TB of data. The graph data extracted from these documents by the International Consortium of Investigative Journalists (ICIJ) has a size of 210MB in csv format and is freely available for download at https://offshoreleaks.icij.org/pages/database.

The whole graph has 839K nodes and 1.2M edges. We only kept the biggest connected component, comprising 750K nodes and 1.1M edges, laid out using the FM^3 algorithm [49].

7.2 Visualization

When the visualization[5] is loaded, the level of detail shown is chosen so that the graph is fully visible in the visualization window. This level of detail gives a good overview of the graph layout, as can be seen on Fig. 8a. Hopefully, the heatmap allows to visually detect a dense cluster on the right of the visualization. One can see that this cluster does not share many connections with the rest of the graph, since the area surrounding it is mostly empty of edges. The interactivity of our system enable to challenge that assumption by using zooming and pan. Several other smaller clusters are also visible, with higher inter-connection. When zooming on this cluster, more detailed data is loaded, allowing to detect the double ring structure shown on Fig. 8b. Zooming further reveals this structure is caused by a node with very high degree, labeled "Portcullis". This node is linked to the inner ring, while nodes in the inner ring connect to the outer ring.

Many similar two-fold structures can be detected in the graph, where two dense clusters serve as "bridges" between two high degree nodes, as seen on Fig.9. Furthermore, the bundling applied helps reducing edge-node clutter when viewing big clusters.

7.3 Performance

Aggregation: The aggregation process for this graph generated 15 levels. On our cluster, it took 6 minutes using 4 Spark executors with 2 cores each, and inserting the generated data into HBase took an additional minute. The aggregated data stored in HBase for this dataset occupies 594MB. This aggregation operation only has to be executed once before many people are able to visualize the dataset.

Visualization: When exploring the graph, each vertex tile transferred weighed 12kB on average and edge tiles 50kB on average. The transfer time for each tile was between 10ms and 100ms. Rendering time without bundling was between 10ms and 30ms. Rendering time with bundling greatly depends on the amount of edges on the screen and the chosen number of iterations. It took between 40ms up to 70ms with many edges on screen. Thanks to the aggregation, few edges need to be bundled, allowing the bundling time to stay relatively low and to apply bundling every frame. All in all, the framerate on the client is more than 30fps without bundling and around 15fps with bundling.

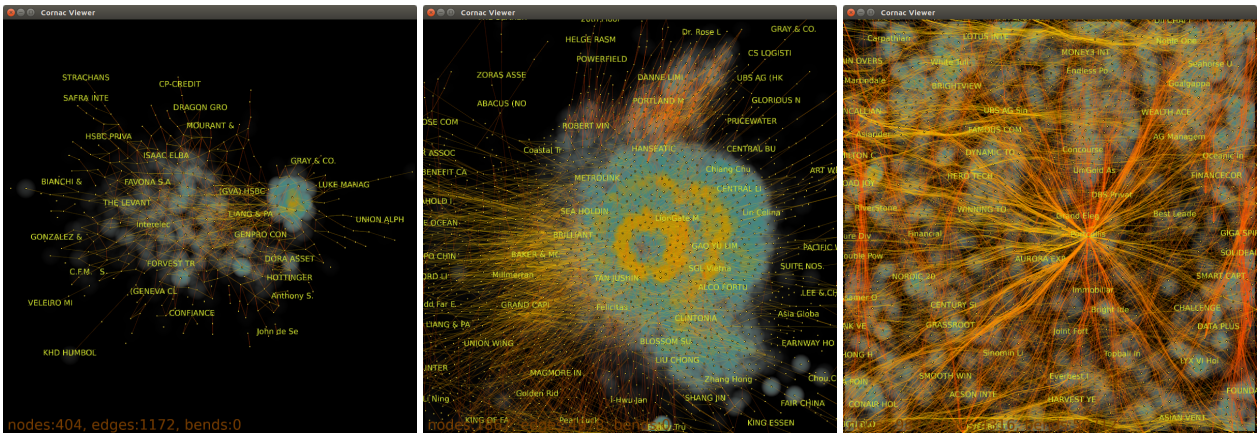
7.4 Discussion

To show the benefits of our contribution, we compare the visualization of the panama papers graph with Cornac to the visualization with another tool from the litterature called LaGo [7]. LaGo does not use distributed computing, but relies on the GPU to draw density function of nodes. It uses CPU to recompute the visualization during interactions and requires a heavy-weight application. We tested LaGo on a laptop equipped with a Core i7-4710HQ, an Nvidia GTX 970M GPU and 16GB of RAM. On this computer, LaGo is able to display the panama papers graph after 2 minutes of preprocessing (see Fig. 10, which is cached for further processing. This is faster than Cornac's preprocessing phase but since LaGo is centralized the data is only available to a single user. Afterwards, every interaction (zoom, pan, changing the size of the density function kernel) required recomputing the density function, leading to an interaction time ranging from 100ms to 3s with large kernel sizes. Cornac achieves constant and lower interaction times even though it is using a deported client which has to fetch from a remote database and is applying edge bundling and label occlusion, thanks to the bounded amount of data transferred on the network and rendered on screen. Furthermore, LaGo seems to show a slightly different structure in the graph than what is shown by Cornac. LaGo was not able to display the star shape in the double ring structure previously mentioned. Indeed, the node with high degree in the center of the structure is in a low density area, so its edges are moved to the high density area inside the ring, wrongly showing a very connected node inside the ring. This is because LaGo does not guarantee a maximum displacement of the edges whereas the Unit disk clustering used by Cornac guarantees that a node's representative is only a few pixels farther. When testing bigger graphs with LaGo, it was able to display all the USA road graphs using almost the entirety of the laptop's 16GB after up to 8 minutess of preprocessing, but was not able to display the OSM-EUROPE graph since 16GB of RAM were not sufficient.

8 Conclusion

In this article, we introduced the first Hadoop-based technique for interactive huge graph visualization. We showed that it enables Web-based visualization of graphs with hundreds of millions of elements on a standard computer. Our system is based on a novel aggregation and indexing algorithm specifically designed for graph visualization. We also proposed modification of the Hurter et al. [45] bundling algorithm to provide on the fly node/node and edge/node clutter reduction on the client. A complete prototype has been developed to benchmark and demonstrate the efficiency

5. An interactive demo of this case study can be found at http://vps245056.ovh.net:1529/Cornac.



(a) (b) (c)

Fig. 8: Visualization of the panama papers dataset. (a) shows an overview of the whole dataset. The heatmap of node density enables to spot a dense cluster on the upper right. (b) shows detail of this cluster. A double ring structure is visible. (c) shows the cause of this structure: a very connected node labeled "Portcullis".

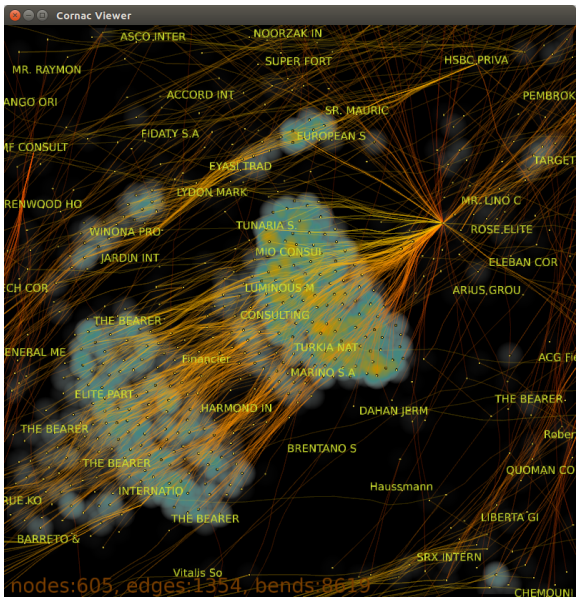


Fig. 9: Zoom on a cluster with bundling enabled.

of this approach, showing it allows huge graph exploration as if they were stored on the client computer.

The technique has been designed as a cornerstone for building domain specific big graph visualization tools. That web based visualization technique may enable to build collaborative exploration and annotations software for massive analysis of large graphs. On the cluster side, future work will concentrate on handling streamed graphs. This feature may be feasible using the so-called lambda architecture. On the client side, many costly visualization techniques may be computed on the fly on the Web browser thanks to the distributed aggregation and must be explored to simplify huge graph analysis.

ACKNOWLEDGMENTS

The authors would like to thank the *LaBRI* for the use of the LSD distributed computing platform.

REFERENCES

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 10, 2010, p. 10.

[2] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: A resilient distributed graph system on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 2.

[3] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *Visual Languages, 1996. Proceedings., IEEE Symposium on*. IEEE, 1996, pp. 336–343.

[4] A. Quigley and P. Eades, "Fade: Graph drawing, clustering, and visual abstraction," in *Graph Drawing*. Springer, 2001, pp. 197–210.

[5] D. Auber, Y. Chiricota, F. Jourdan, and G. Melançon, "Multiscale visualization of small world networks." IEEE, 2003.

[6] F. Van Ham and J. J. Van Wijk, "Interactive visualization of small world graphs," in *Information Visualization, 2004. INFOVIS 2004. IEEE Symposium on*. IEEE, 2004, pp. 199–206.

[7] M. Zinsmaier, U. Brandes, O. Deussen, and H. Strobelt, "Interactive level-of-detail rendering of large graphs," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 12, pp. 2486–2495, 2012.

[8] B. W. Silverman, *Density estimation for statistics and data analysis*. CRC press, 1986, vol. 26.

[9] J. Abello, F. Van Ham, and N. Krishnan, "Ask-graphview: A large scale graph visualization system," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 669–676, 2006.

[10] D. Archambault, T. Munzner, and D. Auber, "TugGraph: Path-preserving hierarchies for browsing proximity and paths in graphs," in *PacificVis' 09*. IEEE, 2009, pp. 113–120.

[11] L. Nachmanson, R. Prutkin, B. Lee, N. Henry Riche, A. E. Holroyd, and X. Chen, "Graphmaps: Browsing large graphs as interactive maps," in *Graph Drawing*. Springer, 2015.

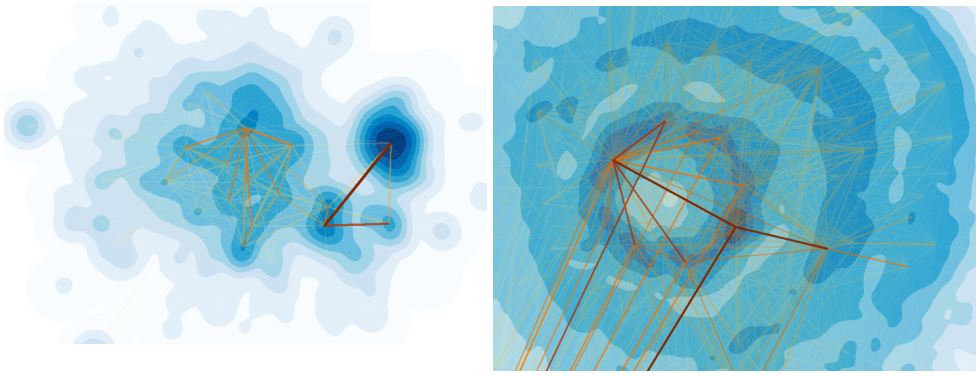
[12] N. Elmqvist, T.-N. Do, H. Goodell, N. Henry, and J.-D. Fekete, "Zame: Interactive large-scale graph visualization," in *PacificVIS'08*. IEEE, 2008, pp. 215–222.

[13] N. Elmqvist and J.-D. Fekete, "Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 16, no. 3, pp. 439–454, 2010.

[14] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: large-scale graph computation on just a pc," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 31–46.

[15] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 472–488.

[16] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proceedings of*



(a) (b)

Fig. 10: Visualization of the panama papers dataset using LaGo [7]. While the node density function accurately shows the double ring structure, the edge displacement does not allow to see the star structure inside the double ring, as Cornac does (see Fig. 8b and c).

the 25th Symposium on Operating Systems Principles. ACM, 2015, pp. 410–424.

[17] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[18] A. Perrot, R. Bourqui, N. Hanusse, F. Lalanne, and D. Auber, "Large interactive visualization of density functions on big data infrastructure," in *5th IEEE Symposium on Large Data Analysis and Visualization*, 2015.

[19] A. Arleo, W. Didimo, G. Liotta, and F. Montecchiani, "Large graph visualizations using a distributed computing platform," *Information Sciences*, vol. 381, pp. 124–141, 2017.

[20] T. M. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Softw., Pract. Exper.*, vol. 21, no. 11, pp. 1129–1164, 1991.

[21] A. Arleo, W. Didimo, G. Liotta, and F. Montecchiani, "A distributed multilevel force-directed algorithm," in *Graph Drawing*. Springer, 2016.

[22] A. Hinge and D. Auber, "Distributed graph layout with Spark," in *Information Visualisation (iV), 2015 19th International Conference on*. IEEE, 2015, pp. 271–276.

[23] P. Gajer and S. G. Kobourov, "Grip: Graph drawing with intelligent placement," in *Graph Drawing*. Springer, 2001, pp. 222–228.

[24] R. Hadany and D. Harel, "A multi-scale algorithm for drawing graphs nicely," *Discrete Applied Mathematics*, vol. 113, no. 1, pp. 3–21, 2001.

[25] C. Mueller, D. P. Gregor, and A. Lumsdaine, "Distributed force-directed graph layout and visualization." *EGPGV*, vol. 6, pp. 83–90, 2006.

[26] A. Tikhonova and K.-L. Ma, "A scalable parallel force-directed graph layout algorithm," in *Proceedings of the 8th Eurographics conference on Parallel Graphics and Visualization*. Eurographics Association, 2008, pp. 25–32.

[27] S. Chae, A. Majumder, and M. Gopi, "Hd-graphviz: highly distributed graph visualization on tiled displays," in *Proceedings of the Eighth Indian Conference on Computer Vision, Graphics and Image Processing*. ACM, 2012, p. 43.

[28] R. Tamassia, *Handbook of graph drawing and visualization*. CRC press, 2013.

[29] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[30] A. Perrot and D. Auber, "Fatum-fast animated transitions using multi-buffers," in *Information Visualisation (iV), 2015 19th International Conference on*. IEEE, 2015, pp. 75–82.

[31] A. McCallum, K. Nigam, and L. H. Ungar, "Efficient clustering of high-dimensional data sets with application to reference matching," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2000, pp. 169–178.

[32] M. Luby, "A simple parallel algorithm for the maximal independent set problem," in *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. ACM, 1985, pp. 1–10.

[33] Y. Métivier, J. M. Robson, N. Saheb-Djahromi, and A. Zemmari, "An optimal bit complexity randomized distributed MIS algorithm," *Distributed Computing*, vol. 23, no. 5-6, pp. 331–340, 2011.

[34] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[35] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[36] B. Cheswick, H. Burch, and S. Branigan, "Mapping and visualizing the internet." in *USENIX Annual Technical Conference, General Track*. Citeseer, 2000, pp. 1–12.

[37] R. Bridson, "Fast poisson disk sampling in arbitrary dimensions." in *SIGGRAPH sketches*, 2007, p. 22.

[38] Y. Zheng, J. Jestes, J. M. Phillips, and F. Li, "Quality and efficiency for kernel density estimates in large data," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 433–444.

[39] M. Dickerson, D. Eppstein, M. T. Goodrich, and J. Y. Meng, "Confluent drawings: Visualizing non-planar diagrams in a planar way." *J. Graph Algorithms Appl.*, vol. 9, no. 1, pp. 31–52, 2005.

[40] D. Phan, L. Xiao, R. Yeh, and P. Hanrahan, "Flow map layout," in *Information Visualization, 2005. INFOVIS 2005. IEEE Symposium on*. IEEE, 2005, pp. 219–224.

[41] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE Trans. Vis. Comput. Graph.*, vol. 12, no. 5, pp. 741–748, 2006.

[42] D. Holten and J. J. van Wijk, "Force-directed edge bundling for graph visualization," in *11th Eurographics/IEEE-VGTC Symposium on Visualization (Computer Graphics Forum; Proceedings of EuroVis 2009)*, vol. 31, 2009, pp. 983–990.

[43] W. Cui, H. Zhou, H. Qu, P. C. Wong, and X. Li, "Geometry-based edge clustering for graph visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1277–1284, 2008.

[44] A. Lambert, R. Bourqui, and D. Auber, "Winding roads: Routing edges into bundles," *Computer Graphics Forum*, vol. 29, no. 3, pp. 853–862, 2010.

[45] C. Hurter, O. Ersoy, and A. Telea, "Graph bundling by kernel density estimation," in *Computer Graphics Forum*, vol. 31, no. 3pt1. Wiley Online Library, 2012, pp. 865–874.

[46] M. van der Zwan, V. Codreanu, and A. Telea, "Cubu: Universal real-time bundling for large graphs," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 12, pp. 2550–2563, 2016.

[47] A. Perrot, "Information Visualization in the Big Data era : tackling scalability issues using multiscale abstractions," Theses, Université de Bordeaux, Nov. 2017. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01664983

[48] A. Zakai, "Emscripten: an llvm-to-javascript compiler," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2011, pp. 301–312.

[49] S. Hachul and M. Jünger, "Drawing large graphs with a potential-field-based multilevel algorithm," in *International Symposium on Graph Drawing*. Springer, 2004, pp. 285–295.

David Auber received his PhD degree from the University of Bordeaux I in 2003. He has been an assistant professor in the University of Bordeaux Department of Computer Science since 2004. His current research interests include information visualization, graph drawing, bioinformatics, databases, and software engineering.

Alexandre Perrot received his PhD degree from the University of Bordeaux in 2017. He is currently a research engineer at the University of Bordeaux. His research interest include Big Data, distributed computing and algorithms and interactive visualization of graphs and massive data.