GraphDB-Tree: A structure to manage large graphs in common PCs

Lucas Fonseca Navarro
lucas.navarro@dc.ufscar.br

August 15, 2014

Abstract

The interest about complex networks is growing in the last few years, most power by the Web expansion with social networks and so on. Thus a lot of new methods used to extract information on these networks have been develop, for example counting and finding triangles is a very important task that allows extracting some characteristics related to communities in complex network. Despite of that, for process a large social network is required a lot of computational resources. Therefore, most of the algorithms assume that large complex networks fit on main memory which is not true, specially using a personal computer. There are specific architectures, such as MapReduce, but this type of architectures usually requires a expensive hardware.

This documents presents *GraphDB-tree* in technical level, a data structure in secondary memory that support graph mining algorithms such as common neighbours and allows use a personal computer to store large complex networks efficiently. A user manual is also presented to show how it can be used.

0.1 Introduction

One common problem when implementing a graph mining algorithm in these days is to make the performance (execution time) scale well as the graphs vertex and edge number grows from hundreds to millions and sometimes billions. This factor was the motivation to the creation of the graph disk structure called GraphDB-Tree.

The GraphDB-Tree is a structure created for fast storage and recovery of a graph on secondary memory. The complexity to recover the neighbor list for any node is $O(1)$, so this structure is very efficient to graph algorithms that uses just the locality of the nodes (e.g., find graph cliques – such as triangles -, calculting common neighbors and extra neighbor scores, jaccard and adamic/adar scores, etc).

To achieve the performance in the node's locality algorithms, the GraphDB-Tree stores the graph partitioned in disk pages, and the entire set of nodes being numeric, sorted and continuous from 1 to $|V|$. Mostly of the graphs, such as the NELL's graph, haven't this characteristics, so a preprocessing task is necessary before the storage on GraphDB-Tree.

This report contains an explanation of GraphDB-Tree within an example of algorithm to count triangles of a graph(see in **Section 2**), and also a "user manual for the GraphDB-Tree" (see in **Section 3**).

A paper about GraphDB-Tree were already published:

L. F. Navarro, A. P. Appel, and E. R. Hruschka Jr., *"Graphdb storing large graphs on secondary memory"*, ADBIS Special session on Big Data: New Trends and Applications (BiDaTA) in conjunction with the 17th East-European Conference on Advances in Databases and Information Systems (ADBIS), vol. 17, pp. 177–186, 2013.

0.2 The GraphDB-Tree structure

Our intention with GraphDB-Tree was to create and implement a structure to store large complex network on secondary memory, making possible to applying graph mining algorithms in a efficient way. There are a lot of ways to represent a graph on a computer, specially in primary memory, such as adjacency matrix and list, but for secondary memory there is almost none. We work mostly with power law graphs, so an adjacency matrix is not a good option, since most all of complex network are very sparse, wasting a lot of space, besides the searching algorithms will be very slow.

The *Vpages*, is the arrays of nodes, each node have: index, degree, pointer to edge list and the initial position on the edge list. The *Epages* is the array of adjacencies, just storing nodes indexes. The *Vpages* and *Epages* are of the size of one disk page, so the operations of write and read have better efficiency.

The *PageManager* is responsible for managing the file structure, providing an efficient read and write operations. We read the complex network from a edge list file, that must be **numeric**, **sorted** and the nodes has to have **continuous** index from 1 to n, where $|V| = n$ is the number of nodes. With this two conditions, we can store the nodes in order and we could easily recover any node by the ID, cause we know exactly where they are in the file, by dividing

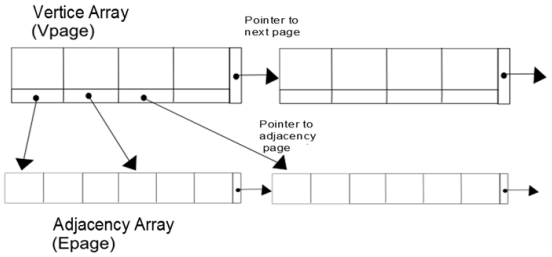


Figure 1: Graphical representation of *Vpages* and *Epages* from *GraphDB-Tree*

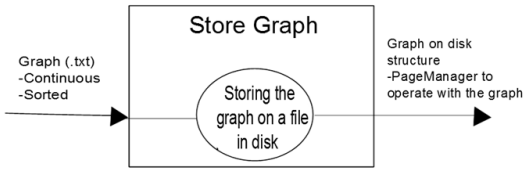


Figure 2: *GraphDB-Tree* core, which is responsible to store the network, with the input and the return parameters

the index of de *node(id)*, for the size of *VPage(—Vp—)*, to find the page, and the rest would give the position. For the triangle query we work only with undirected networks, so if the network is directed, we convert it before we store it. To attend these conditions, we have algorithms to pre-process the edge list of network.

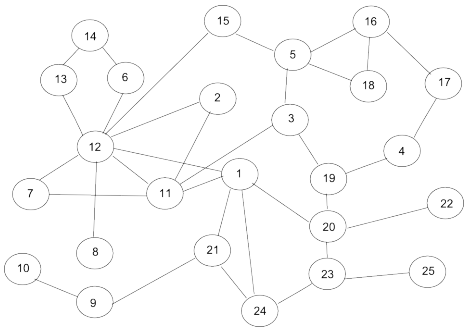
The figure 3 shows an example of an small arbitrary network (a), the respective edges file representing it, sorted and continuous (b), and after being stored (c), the first *Vpages* and *Epages* having 4 and 6 size arrays respectively (in practice we use the array sizes that makes each list of the size of a disk page, and this can vary from PC to PC).

0.3 Main Characteristics

To read a edge list and store it on GraphDB-Tree, it must be numeric, continuous and sorted. This characteristics is what made GraphDB-Tree so fast in execution time.

The continuity factor, makes searches for specific nodes and the operation to recover a adjacency list of a given node have complexity $O(1)$. Example: consider that VN_CAP is size of a VPage, if you want to find a node i VPage, you can use: $\frac{i}{VN_SIZE} + 1$. The position of this node in the page will be: $imodVN_SIZE$.

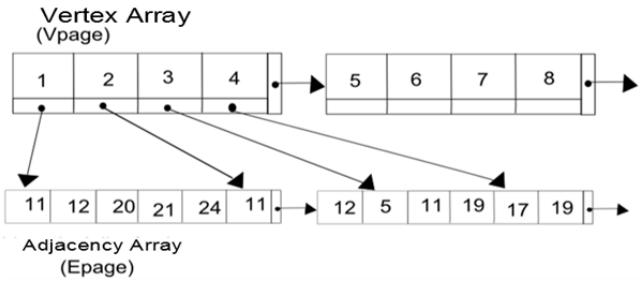
Another notable characteristic of GraphDB-Tree is that it just process **static** graphs (graphs that doesn't change in terms of nodes or edges), but this is not a problem, because it stores a graph too fast that it can process a dynamic graph statically. Is something like take snapshots of the graph in time intervals, then each of this snapshots will be stored in GraphDB-Tree as a new graph and processed.



(a) Small synthetic network

1	11	4	17	8	12	12	2	14	13	19	3	22	20
1	12	4	19	9	10	12	6	15	5	19	4	23	20
1	20	5	3	9	21	12	7	15	12	19	20	23	24
1	21	5	15	10	9	12	8	16	5	20	1	23	25
1	24	5	16	11	1	12	11	16	17	20	19	24	1
2	11	5	18	11	2	12	13	16	18	20	22	24	21
2	12	6	12	11	3	12	15	17	4	20	23	24	23
3	5	6	14	11	7	13	12	17	16	21	1	25	23
3	11	7	11	11	12	13	14	18	5	21	9		
3	19	7	12	12	1	14	6	18	16	21	24		

(b) Edges list file representation



(c) Network (a) stored in *GraphDB-Tree*

Figure 3: *GraphDB-Tree* example.

0.3.1 Triangle Counting Query

Detecting triangles could be very useful to extract some possible relations or help to find communities in a social network among other stuff. With *GraphDB-tree* present structure and the limitations of a common PC, we can't bring some large graphs (with billions of nodes or/and edges) entirely to main memory and process it, so we have to work locally with one or a limited number of Vpages at the same time. Because of this factor, we have to create a specific algorithm to process graphs on our structure.

To find a closed triangle that a node participate in, we need to read at least 2 up to 4 pages, but this can be a bigger number depending on the size and disposal of the adjacency lists of the node. In the example above, we are trying to find triangles in an arbitrary node n, so we read this *Vpage* from disk, and after that we need to find n's adjacencies, that is m_1, m_2, m_3 (to find that we had to read an *Epage*). With this list, we search in each of n's adjacency nodes, for a common adjacency with him. We start with m_1, so we read the *Vpage* to find where to recover his adjacency list, and then we read the correspondent *Epage*, recovering $< X, Y >$. After that, we have to compare if X or Y, is another node from n's list. In this example, we have to compare then to m_2 or m_3 (excluding m_1 because we do not consider graph loops).

If we assume that X it is m_2, above we can see one closed triangle. After that we continue the search in the rest of n's adjacency list nodes. Our algorithm operates basically like this, passing in each node of the graph. Like it was said before, the graph is stored continuously, so we do not need to do any page searches, we just have to read the pages from disk. To count all opened

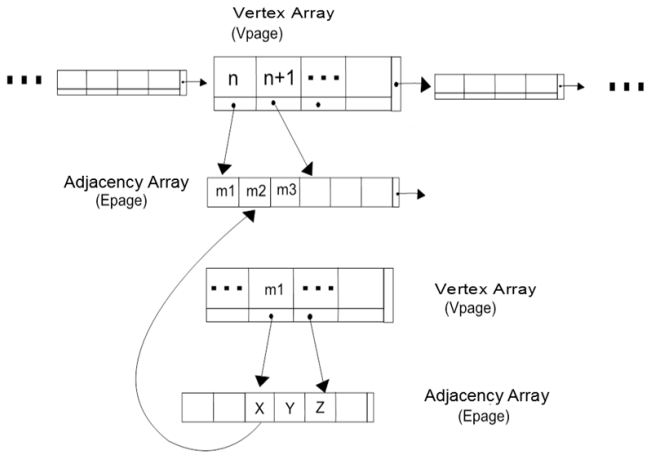


Figure 4: Example of how find a closed triangle in *GraphDB-Tree*

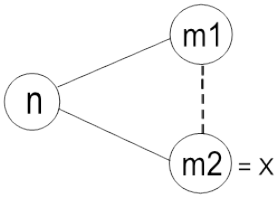


Figure 5: Example of a triangle during the search in Figure 4

triangles we test if X is equal to m_2 as shown in Figure 5, if this is not true (closed triangle), than it is an open triangle.

Our method does not count each triangles three times. As the nodes are stored continuously, we have some conditions that makes us pass through each triangle on the graph just once, while a lot of algorithms count the triangles three times and divide the number in the end.

0.4 Using the GraphDB-Tree

The current distribution of GraphDB-Tree is implemented in *C++*.

0.4.1 GraphDB-Tree Input Format

The GraphDB-Tree structure reads graph files in edges list format, by nodes being numeric, sorted and continuous. The file must contain two nodes id tab-separated in each line, representing one edge of the graph:

$$\begin{array}{cc} n_1 & n_x \\ n_2 & n_y \\ & ... \\ n_m & n_z \end{array}$$

With $n_1 \leq n_2 \leq ... \leq n_m$. The GraphDB-Tree also accepts weighted graphs, so the input edge list will have three elements per line, the third one beign the weight.

$$\begin{array}{ccc} n_1 & n_x & w_1 \\ n_2 & n_y & w_2 \\ & ... & \\ n_m & n_z & w_m \end{array}$$

Assuming that most of graph files have not the characteristics needed to be writed in GraphDB-Tree, the most common execution cycle of a graph algorithm using GraphDB-Tree will be the one presented in the figure above.

Step 1 In this step, the graph file is filtrated lefting only and edge list like in the examples above, being numeric or textual nodes

Step 2 In this step, the edge list needs to be transformed into numeric (if not), continuous and also are sorted. A mask file can be created to store the old labes of the nodes.

Step 3 Here, the edge list will finally be writed on disk via GraphDB-tree. We explain how to do this process in next subsection.

Step 4 In this step, the user algorithm can be executed, quering GraphDB-tree when necessary.

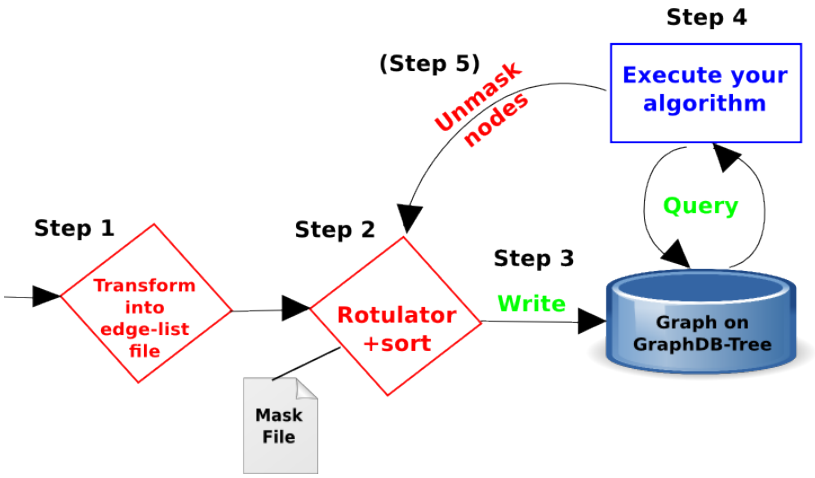


Figure 6: GraphDB-Tree execution diagram

Step 5 This step is optional, in the case of the algorithm of Step 4 output contains nodes, then this output can be unmasked by the rotulator using the mask file, recovering the old labes of the nodes.

The steps 1, 2 and 3 will need to be implemented by the user, in the next subsections we explain how to execute the step 3 and 4, that are part of the of the GDB-Tree.

0.4.2 Writing a graph in GraphDB-Tree

To write a graph on the GraphDB-Tree, as it was said above, it must be in an edge-list format file beign numeric, continuous and sorted (by the first column). After you have this edge list, to write your graph in your disk using GraphDB-Tree, you just need the following code:

//Create a file manager (PageManager) for the graph
PageManager *pm;
pm = new PageManager(< $name\_for\_graphdb - tree\_file$ >, false); //The second parameter indicates that you're creating a new file, instead of opening an existing one.
//Create the graph
Graph g1(< $graph\_file\_name$ >, pm);
g1.MakeGraph();

The GraphDB-Tree implementation that we are current distributing have two rotulators: one to make a already numeric and sorted edge list to continuous, and another that receives a graph and a categorization list and make them numeric and continuous, needing a sort process later. It can be used by the following command on linux:

./GraphDB-tree.cpp (-e) -n/-t < $graph\_file\_name$ > < $name\_for\_graphdb-tree\_file$ > (-o < cat_list_file >)

- The parenthesis represents optional commands.
- The -e optional parameter shall be used if you want to store a weighted graph.
- Put -n if the edge-list is numeric, and -t if it is textual.
- Use -o optional parameter and pass a categorization list if your input graph is ontological.

Examples:

Unweighted, numeric:
./GraphDB-tree.cpp -n myGraph.txt myGraph.dat

Weighted, textual, ontological:
./GraphDB-tree.cpp -e -t myGraph.txt myGraph.dat -o myGraphCategoryList.txt

0.4.3 Querying a graph stored on GraphDB-Tree

To query a graph file in the GraphDB-Tree format, first you have to create a pagemanager to open you graph file stored on GraphDB-Tree:

//Create a file manager (PageManager) for the graph and page pointers
PageManager *pm = new PageManager($< graphdb - tree\_file\_name >$, true);
VPage *vp;
Epage *ep;

As it was said in previous sections, the graph will be stored in diskpages, called VPages (to store nodes) abd EPages (to store adjacency list for each node). With the Pagemanager you can load this pages using the following function:

vp=pm− >ReadVPage(vpg_id);
ep=pm− >ReadEPage(epg_id);

It exists two constants (VN_CAP and EN_CAP) that are used to determine the maximum number of nodes in each page, ajusted to make any VPage or EPage have the size of a disk page. As the graph is continuous and sequentially stored, you can retrieve any node and his adjacency list with complexity $O(1)$, for example I want to print the adjacency list and weight of the edges of the node with ID 2000 :

vp=pm− >ReadVPage($2000/VN\_CAP + 1$);
ep=pm− >ReadEPage(vp− >getEpageID($2000\%VN\_CAP$));

int i=vp-− >getEpagePos($2000\%VN\_CAP$);
int j=vp-− >getGrau($2000\%VN\_CAP$);

```
for(i; i < i + j; i + +)
    cout<<ep− >getNodeID(i)<<“ ”<<ep− >getWt(i)<<endl;
```

To free the variables with loaded pages you can use:

```
pm− >freeVPage(vp);
pm− >freeEPage(ep);
```

A great example of how to query a graph stored in GraphDB-tree, is the algorithm on the file “TriCounter.cpp“ distributed within GraphDB-Tree. It's an algorithm to count all triad of a graph.

Another useful methods:

```
pm− >getNumNodes(); //To get the number of nodes in the graph
pm− >getEPageCount(); //To get the number of total EPage of the graph
pm− >getVPageCount(); //To get the number of total VPage of the graph
```

0.4.4 Important Notes

- If this manual is not suficient to you, or you are have problems with GDB-Tree or the implementation of your algorithm please send me and e-mail. I'll be pleased to help!

- Part of the code comments of Pagemanager is still in portuguese, I'll be translating it soon, but if you have any question of some specific part of the code, send me and e-mail!

- The GraphDB-Tree was implemented and tested just on Linux Ubuntu systems, we use a kernel function sort to do some sorting tasks on the scrip (GDBT.sh) within the code. If you want to execute GDB-Tree in linux you might need to replace this sort function.

- Checkout ”TriCounter.cpp“ to learn how to query on GraphDB-Tree.

0.5 Conclusion

A Paper was already published about the GraphDB-Tree, but this document has a technical characteristic. Our intention for this document is to help any one that might want to use the GraphDB-Tree, by explaining the concepts and intentions behind GraphDB-Tree and also providing an user manual to help users to pre-process it graph file, write it to GraphDB-Tree and query it later with its own algorithm.