Internship Report: Implementing a Vertex Separation algorithm in Sagemath from "The Vertex Separation and Search Number of a Graph"

Klaus Jaschan

19/05/2014

Abstract

This report include details about the implementation in Sage of the algorithm explained in [4] to compute the Vertex Separation of trees. It also contains an very early stage of a research to find the reason of the good behavior of the algorithm Branch and Bound [2] when is used in trees.

1. Context

The concepts learn in this internship were: Vertex Separation, Pathwidth, Path Decomposition, Search Number, Node Search Number, Layout of a Vertex Separation and how all this terms are related to each other.

The following Node Search Number and (Edge) Search Number definitions are from [5]:

In a search game the objective is to capture a fugitive who moves freely about the edges of a graph G. Initially all edges are considered *contaminated* (this means that the edge has not been checked and holds a possibility of sheltering the fugitive). A search strategy is a sequence of steps that will clear all edges of a graph. To clear an edge a searcher is placed at one end and a second searcher moves along the edge from the guarded endpoint to the other endpoint. If all other edges incident on the guarded endpoint are already clear, then the guard may be moved along the edge to the other endpoint. The (edge) search number of G, denoted by $s(G)$, is the minimum number of searchers for which a search strategy exists. In the node-searching version of the problem Searching and pebbling [6], searchers are used only to guard nodes, and an edge becomes clear when both of its endpoints are concurrently guarded by searchers. The node search number is denoted by ns(G).

A (linear) *layout* of a graph $G =< V, E >$ is a one to one mapping $L : V \rightarrow j\{1, 2, ..., |V|\}$. Thus, L is a permutation of the vertices of G. For any layout L, define $V_L(i) = \{u \in V \mid L(u) \leq i \ \wedge \exists \ v \in V : uv \in E \wedge L(v) > i\}$. $V_L(i)$ is the number of vertices of G mapped to integers less than or equal to i that are adjacent to vertices mapped to integers greater than i. [5]

The *vertex separation* number of G with respect to L, $vs_L(G)$, is the maximum number of vertices in any $V_L(i)$. The vertex separation number of G is the minimum, over all possible layouts L of G, of $vs_L(G)$. Formally, $vs_L(G) = max_{1 \leq i < |V|} \left\{ \left| V_L(i) \right| \right\}$ and $vs(G) = min\{vs_L(G) \mid L$ is a linear layout of

$G\}$. [5]

Informally, The *path decomposition* is a succession of group of nodes, that describes a search strategy, and the size of the bigger of this groups is called *pathwidth*.

In [5] defines the Path Decomposition in the following way:

Given a graph $G =< V, E >$, a sequence $X_1, ..., X_r$ of subsets of V is a *path decomposition* of G if the following conditions are satisfied:

(a) $\cup X_i = V$,

(b) for every edge e of E, some X_i contains both endpoints of e, and

(c) for $1 \le i \le j \le k \le r$, $X_i \cap X_k \subseteq X_j$.

The *pathwidth* of G, denoted by $pw(G)$, is the minimum value $h \ge 0$ such that G has a path decomposition $X_1, X_2, ..., X_r$ with $|X_i| \le h + 1$ for $i = 1, ..., r$.

Vertex Separation and its Layout are the main concept used in this Internship. Being possible to center in this is because this concepts are related with each other, i.e. Knowing the Vertex Separation of a graphs means that you also knows the pathwidth.

For any graph G:

$ns(G) - 1 = pw(G)$ [4,5]

$ns(G) = vs(G) + 1$ [4,5]

$ns(G) - 1 \le s(G) \le ns(G) + 1$ [6]

s(G) <= vs(G) + 2 [4]

The meaning of this is powerful, because this proves that problems that were thought different are the same, so they can be solved in the same way.

A *tree* is a type of graph that it is undirected and any two vertices have only one path that connects them. Is also possible to represent a tree with a directed graph (a.k.a. *DiGraph*), instead of one edge connecting two nodes, there will be two edges: one that goes from the first node to the second one, and another that goes from the second node to the first one.

Informally, a node of a tree is *k-critical* when the subtree with that node as root have vertex separation *k* and the two subtrees formed with the two childrens as a root have vertex separation *k* respectively.

From [4], the definition of *k-critical* and *label*: Given a tree *T*, and *x* a node of *T*, let $T[u]$ denote the tree with root *u* within the rooted tree *T*. Let $T[u, v_1, v_2, ..., v_i]$ denote the tree with root *u* from which the subtrees with roots v_1 to v_i have been removed.

A vertex *x* is *k-critical* in a rooted tree *T* iff $vs(T[x]) = k$ and there are two children *y* and *z* of *x* such that $vs(T[y]) = vs(T[z]) = k$.

For any tree $T[u]$ define the *label* of *u* to be the list of integers $(a_1, ..., a_p)$, where $a_1 > a_2 > ... > a_p \ge 0$, and for which there exist a set of vertices $\{v_1, ..., v_p\}$ such that:

(a) $vs(T[u]) = a$.

(b) For $1 \le i < p$, $vs(T[u, v_1, ..., v_i]) = a_{i+1}$.

(c) For $1 \le i < p$, v_i is an a_i-*critical* vertex in $T[u, v_1, ..., v_{i-1}]$.

(d) v_p is *u*. If a_p is marked with a prime (‘) then there is no a_p-*critical* vertex in $T[u, v_1, ..., v_{p-1}]$. If a_p is not marked with a prime, then v_p is an a_p-*critical* vertex. In both cases $T[u, v_p] = T[u, u]$ is the

empty tree.

The *Branch and Bound* (B&B from now on) algorithm that is analysed in this internship is the exposed in [2] section 3. This B&B algorithm has two main parts. The first one is the pre-processing of the input graph and the second one is the B&B algorithm itself (B&B phase). The objective of the pre-processing part is to apply reduction rules whenever is possible. These rules will reduce the number of nodes, for each time that a rule is applied, the amount of nodes decreases by one. With this, for each successfully applied rule, the worst case time-complexity is divided by n. When is not possible to apply a reduction rule, the next phase starts. In the B&B phase, the main idea is to cut the exploration of the layouts. This works by testing the *prefix*. The prefix is a linear ordering, that is made from a subset of nodes from V. By testing the prefix, the B&B algorithm applies two different rules to determinate if is possible to discard all the layouts that starts with that prefix (first rule) or if the search is restricted to only layouts that starts with that prefix (second rule). These rules are supported by the Lemma 7 and 6 (respectively) in [2] in the section 2.3. The B&B algorithm is recursive, and it start with an empty prefix, that prefix grows with the second rule, applying a greedy sub-procedure, generating a new prefix that have the previous one contained within it. Then it calls the B&B algorithm appending a node v to the prefix, for all v in V without the nodes already contained in the prefix.

2. Purpose of the internship

The purpose of the internship is to study practical algorithms for computing pathwidth and path decomposition of graphs. To become more familiar with path decomposition of graphs and with the already implemented algorithms. To implement algorithms based on linear programming or/and algorithms that are dedicated to particular graph classes (sparse graphs) and to learn to code in Sage and Python.

3. Resume of accomplished work

In this internship the realised tasks were:

- Write functions in sage applying the concepts of:
    - Measure of a Layout
      Inputs: graph (sage graph), layout (list)
      Output: width (integer)
      This function calculates and return the vs_L of the given layout L of the given connected graph.
    - Width of a Path Decomposition
      Inputs: path-decomposition (list of lists)
      Output: width (integer)
      This function return the width of the given path decomposition.
    - Elimination of redundant nodes of a Path Decomposition
      Inputs: path-decomposition (list of lists)
      Output: path-decomposition (list of lists)
      This function returns a simplified version of the given path decomposition, eliminating the redundant elements that do not add information respect the corresponding layout.
    - Compute a Path decomposition with width at most k from a layout with vertex separation

at most *k*

Inputs: graph (sage graph), layout (list)

Output: path-decomposition (list of lists)

Given a connected graph and its layout with vertex separation at most *k*, this function returns the corresponding path decomposition with width at most *k*.

- Compute a Layout with vertex separation at most *k* from a Path Decomposition with width at most *k*

  Inputs: path-decomposition (list of lists)

  Output: layout (list)

  Given a path decomposition with width at most *k*, this function returns the corresponding layout with vertex separation at most *k*

- Validate a given Path Decomposition

  Inputs: graph (sage graph), path-decomposition (list of lists)

  Output: true/false (boolean)

  Given a connected graph and a path decomposition candidate, this function return "true" when the given path decomposition is a valid one, otherwise its returns "false".

- Vertex Separation

  Inputs: graph (sage graph)

  Output: layout (list)

  This function calculates and returns the layout that solves the vertex separation problem of the given graph. This is done by comparing all the possible layouts, so the bigger graph size for the input is of 10 nodes.

- Implementation in sage of the Vertex Separation Algorithm of [4].
- Analyze why the Branch and Bound Algorithm behaves better on trees.

4. Details of accomplished work

In this report will included a small explanation and details about the implementation of the algorithm to compute Vertex Separation on trees of [4] and the beginning of an analysis of the Branch and Bound algorithm of [2], but will not include information about the rest of the functions made in sage, because they were made for learning purposes and do not contribute anything new.

4.1. Implementation of Vertex Separation algorithm for trees of [4]

The implementation was realize to be used in Sage (www.sagemath.org), a free open-source mathematics software that works over the computer language Python.

The algorithm uses a recursive method to compute the vertex separation with the uses of *labels* that are assigned to each node of the tree. Moreover each node also is categorized as critical or not, this information is also stored in the label.

The input of the algorithm is an unoriented tree, but is easily adaptable to make it work with previous oriented trees. The output is a list of two elements, the first one is the vertex separation number of the tree and the second one is the an array that describes the layout.

In resume, the algorithm starts assigning a root, then goes to the sons, and then to the son of the sons (if possible) until it reach a leaf node. This process is made in a recursivity way, and starts returning

from the leaf nodes to the root, computing and assigning labels to each node as it goes back. The labels contains the information of the Vertex Separation of the subtree generated by the node owner of the label and if that node is k-critical or not.

The label of a node is computed with the information of the sons' labels. The labels of each node does not only contain the vertex separation of the subtree formed by the owner node, it also says if that node is k-critical or not and saves information about about the existence of critical sons.

This means that all the information relevant of the complete tree is in the label of the initial root. Thanks to that, it is possible to determinate a way to construct a layout.

The layout is made by going forward from the initial root and checking if the label has a more than one element, because if the label have more than one element, it means that in a lower part of the tree exists a k-critical node, that means that is important to have a special ordering with the sons of that node. So, the program goes directly to that part of the tree and removes that part of the tree as well adding them in the layout. Giving priority to the k-critical vertex in the formation of the layout is the procedure that is described in the algorithm.

Something worth mentioning, is the way that the labels are stored in this implementation. The labels can have a "no critical" indicator that is represented by a prime (') that will be always in the end of the label; e.g., (4,1'). This generates a problem in the code, because when you read the the elements in the label, you will read numbers, except when the number have the prime indicator, in that case it will be considered as a string. To avoid this problem, the prime indicator is separated from the number and included as a different element in the label; e.g., (4,1,'). Also with this subtle change in the label, the way to recognize if the label's node is k-critical or not is easy. If the last element of the label is different to prime and equal to k then the node is k-critical else it is not. With this change, is avoided the work of processes the element of the label as a string, separe the prime indicator from the number and parse it to an integer.

The paper also includes an algorithm to compute the layout of the tree, it is an optional addition to the previous algorithm. It uses all the already computed labels to create a layout, this is also done by a recursive method. The implementation of this part is slightly different from the proposed one. The paper instruction "for all children y of v_i do layout(y)" was replaced by "for all children y of v_i such that y is not in S do layout(y)". "S" was declared previously as "{Let (v_1, v_2, ..., v_s) be the sequence S containing all vertices x in T[c] such that label(x) contains k}". This sequence S does not follow whatever other, it follows a logic explained in the proof of the theorem 3.1 of [4]. Also the instruction "delete v_i from T" is repositioned because it is necessary to have v_i in T for the next step. Also is worth mentioning that "if T[c] has a critical vertex" is not about if the node *c* is k-critical or not, it is about the existence of a k-critical node below the node *c*. The way to check this is if the label of the node *c* contains more than one elements or not.

4.1.1 Testing the implementation

The implementation was test against a set of trees with known Vertex Separation provided by David Coudert. Then it was tested with random generated trees, comparing the results with the implementation of the algorithm Branch and Bound [2] and with the Sage function "sage.graphs.graph_decompositions.vertex_separation.path_decomposition". The random generated trees were of size from 2 to 200 nodes (2 to 30 against the Sage function due its own size limit) and more of 1 million of random generated trees were tested. After that, the implementation was tested against perfectly

balanced trees generated by the Sage function “graphs.BalancedTree(r,h)” where “r” is the degree of the root node and “h” is the height of the balanced tree.

The biggest instance of a tree that was tested in this implementation (and also the slowest to compute) was a perfect balanced tree of degree 7 and height 7, that is 960800 nodes, the time was 1075.95 seconds. This was done in a machine with Intel core 2 duo of 2.33 GHz and 4 GiB of RAM, running Linux, Debian distribution.

4.2. Branch and Bound analysis

The objective of this analysis is to search a reason that explains the good behavior of the Branch and Bound Implementation made in [2] when it is used to compute the Vertex Separation of trees. This algorithm can compute the Vertex Separation of any connected and loopless graph, it is not limited to trees, but for some reason unknown to the authors, it have good execution time when the analysed graph is a tree.

4.2.1. Questions

Is there a common characteristic between trees that could explain this good behavior?
If exist, is this characteristic enough to explain good behavior in all the trees? and in other kind of graphs?

4.2.2. Hypothesis

The following idea is based on a hunch. Given that trees have a restriction that they can not have cycles, it means that the amount of edges in trees is always the same $2(n - 1)$ being n the amount of nodes. While any (connected) graph does not have to follow this condition, they can have more edges than a tree.

Given that, the hunch is that while the amount of edges is lower in a graph the algorithm gets a faster resolution time, because the graph will be more similar to a tree.

For compare trees with graph that are not trees, I will use the *average degree* (amount of edges divided amount of nodes) of the graph. The trees have an average degree of $\frac{2(n-1)}{n}$ that depends on the amount of nodes (n) and has limit 2 when the amount of nodes goes to infinity. Instead, any connected graph will have always a bigger amount, because if it were the same, it will be a tree, and having less is impossible given that is a connected graph.

4.2.3. Results

The chart was made with the data shown in the appendices, 200 different graphs, 100 of them trees. Each point in the charts represent one graph.

In the following chart (Fig. 1.) is visible the *average degree* versus *time* used to compute the vertex separation. The random graph were connected, the amount of nodes and edges were generated by a uniform distribution. The amount of nodes was limited by a maximum of 100 and a minimum of 10. The random trees used the same amount of nodes restriction.

The first impression seems to be that at lower average degree, more time the algorithm takes to compute for graph that are not trees. This gives evidence that the hypothesis is false.

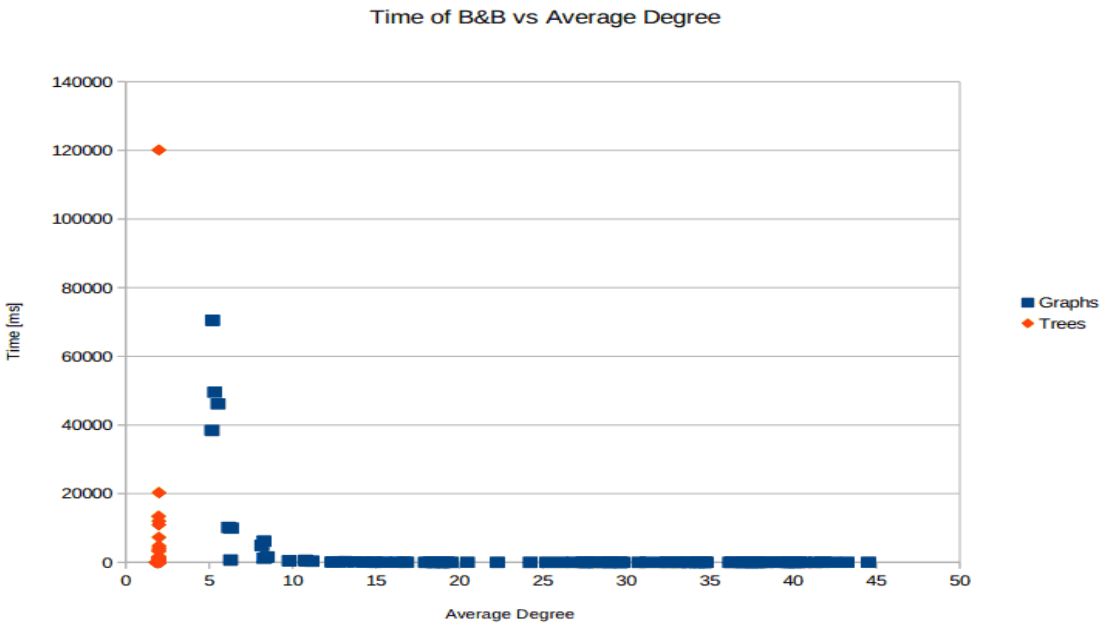

Fig. 1. Data of the the time took by B&B and average degree for random connected graphs and random trees.

5. Conclusion

Respect the B&B analysis, the data obtained does not seems to add something new to clarify the reason of the good behavior of the B&B algorithm in trees. But the data does reject the proposed hypothesis. It is possible to assume that the good behavior is not related to the little amount of edges (compared to any connected graph), but I do not discard the possibility to be related to the loopless characteristic of trees.

The implementation of the algorithm shown in [4] was a success, it can handle big trees using little time, with a normal computer. It still have room for improvements, specifically in the partial ordering of the layout.

The experience of this internship was great to learn how the academic world is, with it I made a step closer to how to do research and how to approach problems. But it still left in me a feeling of emptiness, 3 months is very little time to catch up and do something meaningful. Either way I appreciate the knowledge acquired in this internship.

6. Acknowledgments

This Internship was done in Inria Méditerranée, Sophia Antipolis. Under supervision of Nicolas Nisse, and the COATI Team.

This Internship was possible thanks to the grants from Inria Associated Team AlDyNet between COATI (Inria, I3S (CNRS/UNS)); Universidad Adolfo Ibanez (Santiago, Chile) and Action ECOS-SUD, Chile, Algorithmes distribués pour le calcul de la structure des réseaux.

References

[1] Coudert, D., F. Huc, D. Mazauric, *A Distributed Algorithm for Computing the Node Search Number in Trees*, Algorithmica **63** (2012), pp. 158-190.
URL http://hal.inria.fr/inria-00587819/

[2] Coudert, D., D. Mazauric, N. Nisse, *Experimental Evaluation of a Branch and Bound Algorithm for computing Pathwidth*, Research Report **8470**, INRIA (2014).
URL http://hal.inria.fr/hal-00943549/

[3] Coudert, D. and J.-S. Sereni, *Characterization of graphs and digraphs with small process numbers*, Discrete Applied Mathematics **159** (2011), pp. 1094-1109.
URL http://hal.inria.fr/inria-00171083/

[4] Ellis, J. A., I. H. Sudborough, J. S. Turner, *The Vertex Separation and Search Number of a Graph*, Information and Computation **113** (1994), pp. 50-79.

[5] Kinnersley, N. G., *The vertex separation number of a graph equals its path-width*, Information Processing Letters **42** (1992), pp. 345-350.

[6] Kirousis, L. M., C. H. Papadimitrou, *Searching And Pebbling*, Theoretical Computer Science **47** (1986), pp. 205-218.

Appendices

Data of the charts

Connected Graphs				Trees		
Amount of Nodes	**Average Degree**	**Time [ms]**		**Amount of Nodes**	**Average Degree**	**Time [ms]**
42	41.72	37		96	1.97	24
43	8.16	4957		58	1.97	232
62	19.48	26		40	1.96	17
66	40.32	35		17	1.90	4
18	32.48	29		57	1.98	20321
89	20.48	41		49	1.88	4
72	33.72	32		28	1.97	25
47	8.28	6164		83	1.97	22
43	28.76	27		99	1.97	31
10	5.16	38434		94	1.97	3400
31	14.84	68		80	1.98	743
61	13.68	127		51	1.95	11
15	30.76	27		71	1.96	19
64	29.04	31		70	1.92	7
57	8.28	1226		98	1.98	4139

60	10.8	412		97	1.82	3
33	40.72	35		76	1.88	3
56	18.4	33		54	1.96	27
89	41.84	39		30	1.97	13433
18	5.2	70496		42	1.95	11
30	36.96	32		58	1.98	55
34	32.04	32		60	1.98	1685
80	16.68	63		50	1.97	30
46	26.84	26		13	1.98	1562
15	26.08	27		88	1.83	3
14	28.16	29		48	1.88	4
50	38.44	36		92	1.97	1084
70	43.24	37		22	1.96	20
52	22.28	32		90	1.98	3356
53	27.6	27		80	1.93	6
86	16.84	42		97	1.98	128
18	44.52	40		59	1.94	9
37	19.24	42		97	1.83	3
47	38.72	37		47	1.83	2
10	19.08	36		28	1.94	9
50	18.44	40		42	1.97	26
64	15.52	53		11	1.94	8
64	5.52	46143		38	1.97	316
33	34.48	30		39	1.98	40
18	39	37		55	1.92	5
71	12.72	153		92	1.97	257
29	29.6	27		80	1.98	583
18	5.32	49567		87	1.96	13
46	37.2	31		30	1.88	4
48	34.44	33		17	1.94	8
72	42.48	39		87	1.98	120129
66	32.92	29		57	1.91	6
87	39.88	37		39	1.96	19
47	10.76	586		69	1.98	572
22	6.28	644		74	1.98	7288
99	39.52	36		72	1.97	34
77	12.44	133		49	1.97	37
29	18.92	31		36	1.90	5
70	27.96	28		17	1.96	14
44	11.16	364		66	1.98	3353
12	16.28	71		36	1.89	3
87	18.72	46		82	1.97	416
34	17.96	34		59	1.93	7
35	37	32		80	1.95	10
16	34.2	30		10	1.97	159

52	33.72	31		61	1.97	10924
54	40.36	37		20	1.87	4
21	28.96	27		10	1.98	735
18	37.48	34		52	1.98	39
59	18.92	57		85	1.95	14
96	39.68	33		47	1.92	7
32	37.2	33		56	1.97	21
43	31.2	28		16	1.92	5
99	6.32	9991		55	1.97	24
54	39.88	34		44	1.97	26
47	27.56	28		64	1.88	4
33	36.32	36		18	1.97	226
81	29.6	27		79	1.98	40
11	28.96	28		11	1.87	3
23	33.24	31		22	1.91	6
32	29.8	29		79	1.83	3
51	34.72	30		86	1.91	6
71	18.2	48		73	1.96	93
69	18.92	35		38	1.96	14
93	41.32	35		10	1.98	12050
45	38	36		50	1.98	766
84	34.32	30		60	1.95	13
13	39.76	37		35	1.98	4797
74	6.16	10148		86	1.82	3
44	24.24	30		22	1.97	33
17	40.08	36		70	1.93	18
41	37.72	35		18	1.97	132
65	18.52	46		20	1.98	947
34	12.36	105		35	1.98	38
49	27.4	27		31	1.98	1196
84	14.92	73		63	1.97	84
27	9.8	429		22	1.87	3
64	37.68	32		61	1.97	203
83	34.8	33		62	1.96	17
59	36.2	34		67	1.97	33
53	13.16	175		13	1.96	19
49	37.88	32		72	1.97	207
30	14.32	92		87	1.97	46
96	25.24	25		58	1.95	12
96	8.48	1546		76	1.97	418

Vertex separation for a Tree algorithm of [4] implementation in Sagemath

```
# Vertex Separation for a Tree.
# Paper: The Vertex Separation and Search Number of a Graph
# Authors: Ellis, Sudborough and Turner.
# Year: 1994.
def vs(tree, compute_layout = False, verbose = False):
	#TODO: check if the given tree is a tree, raise ValueError if is not.
	GLOBAL_LABELS = {};
	#checking if the tree is oriented or not:
	if not tree.is_directed() :
		T = tree.strong_orientation();
		u = tree.vertices()[0];
		if len(T.neighbors_in(u)) != 0 :
			raise ValueError("This was not suppose to happen, You were trying
to compute something different than a Tree Graph?");
	else : raise ValueError("The tree must be undirected.");
	asd = compute_label(T, u, verbose, compute_layout, GLOBAL_LABELS);
	asd = copy(asd);
	if compute_layout :
		tree_copy = tree.strong_orientation().copy();
		layout = [];
		if verbose : print "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~";
		layout_vs(tree_copy,u,GLOBAL_LABELS,layout,verbose);
	if verbose : print "Final Label",asd;
	if not compute_layout : return asd[0];
	else : return (asd[0],layout);
def compute_label(tree, root, verbose = False, compute_layout = False, GLOBAL_LABELS =
None):
	if len(tree.neighbors_out(root)) == 0 :
		out = [0,"'"];
		if verbose : print "Label of the node:",root,"is",out,"(leaf)";
		if compute_layout : GLOBAL_LABELS.update({root:out});
		return out;
	else:
		labels = list();
		for v in tree.neighbors_out(root) :
			labels.append(compute_label(tree, v, verbose, compute_layout,
GLOBAL_LABELS));
		out = combine_labels(labels, verbose);
		if verbose : print "Label of the node:",root,"is",out,"(no-leaf)";
		if compute_layout : GLOBAL_LABELS.update({root:out});
		return out;
def combine_labels(labels, verbose = False):
	output = [0, "'"];
	prevm = [];
	for i in labels :
		prevm.append(i[0]);
		if 0 in i :
			output = [1,"'"];
	m = max(prevm);
	for k in range(1, m + 1) :
		subLabels = [];
		#n = number of labels containing an element k
		n = 0;
		for i in labels :
			if k in i:
				n = n + 1;
				subLabels.append(i);
```

```
                if n < 1 : continue; #This is to avoid the check for "is_critical", no
case checks for n < 1.

                if n < 3 :
                        #Check if "the element k is critical in the label"
                        is_critical = True;
                        counter = 0;
                        for i in subLabels :
                                if i[-1] == "'" and i[-2] == k :
                                        counter = counter + 1;
                        if counter == len(subLabels) : is_critical = False;

                #Cases:
                if n >= 3 :
                        output = [k + 1, "'"];
                        if verbose : print "Case 1", output;
                elif n == 2 and is_critical :#at least one element k is critical (no ')
                        output = [k + 1, "'"];
                        if verbose : print "Case 2",
output,str(k)+"-Critical:",is_critical;
                elif n == 2 and not is_critical :#neither element k is critical
                        output = [k];
                        if verbose : print "Case 3",
output,str(k)+"-Critical:",is_critical;
                elif n == 1 and (k in output) and is_critical :#element k is critical (no
')
                        output = [k + 1, "'"];
                        if verbose : print "Case 4", output;
                elif n == 1 and (not k in output) and is_critical :#element k is critical
(no ')
                        aux = list();
                        aux.append(k);
                        output = aux+output;
                        if verbose : print "Case 5", output;
                elif n == 1 and not is_critical :#element k is not critical
                        output = [k, "'"];
                        if verbose : print "Case 6", output;
                else :
                        raise ValueError("No Case was applied, code must be checked!");
        return output;
def layout_vs(tree, root, GLOBAL_LABELS,layout,verbose=False):
        # Declaring variables:
        x = copy(root);
        c = copy(x);
        k = GLOBAL_LABELS[x][0];

        if verbose : print "Start of the function -
root:",x,"label["+str(x)+"]:",GLOBAL_LABELS[x];

        if k == "'" : raise ValueError("k had the value ', this should not be
happening."); #This could not be necessary. TEST

        #(START) Paper: "if T[c] has a critical vertex". What I did: "if the label of
the node c contains more than one label".
        if len(GLOBAL_LABELS[c]) > 2 and  GLOBAL_LABELS[c][-1] == "'" or
GLOBAL_LABELS[c][-1] != "'" and len(GLOBAL_LABELS[c]) > 1:

                if verbose : print "-Critical vertex found - node:",c,"
label["+str(c)+"]:",GLOBAL_LABELS[c];
```

```
                # While c is not a k-critical
                while not is_k_critical(k, GLOBAL_LABELS[c]):

                        if verbose : print "--The actual node isn't k-critical -
node:",c,"removing label:",k,"from:",GLOBAL_LABELS[c],;

                        GLOBAL_LABELS[c].remove(k);

                        if verbose : print "resulting:",GLOBAL_LABELS[c];

                        if len(GLOBAL_LABELS[c]) == 1 and GLOBAL_LABELS[c][0] == "'" :
GLOBAL_LABELS[c].remove("'"); #This could not be necessary. TEST

                        #c = the child of c with k in its label
                        c_aux = [c_candidate for c_candidate in tree.neighbors_out(c) if k
== GLOBAL_LABELS[c_candidate][0]];

                        if len(c_aux) != 1 : raise ValueError("Next value for C is not 1
value."); #This could not be necessary. TEST

                        c = c_aux[0];

                        if verbose : print "--Moving to another node - node:",c,"
label["+str(c)+"]:",GLOBAL_LABELS[c];

                if verbose : print "--The actual node is k-critical - node:",c,"
label["+str(c)+"]:",GLOBAL_LABELS[c];
                # I changed the original indentation of the paper, because it doesn't
make sense to have the rest of the code "inside" of the previous "if"

        # Creating the order S:
        # Instruction of the paper: "{Let (v_1, v_2, ..., v_s) be the sequence S
containing all vertexes x in T[c] such that label(x) contains k}" (it says nothing
about the order)
        # Maybe this could be optimized:

        #getting the sons of the node that have k in the label
        sons = tree.neighbors_out(c);
        aux = [];
        for son in sons :
                if GLOBAL_LABELS[son][0] == k : aux.append(son);
        len_aux = len(aux);
        #if the node doesn't have sons with k in the label:
        if len_aux == 0 : S = [c];
        #if the node have 1 son with k in the label:
        elif len_aux == 1 :
                nodes_iter_s1 = tree.breadth_first_search(aux[0]);
                S1 = [y for y in nodes_iter_s1 if k == GLOBAL_LABELS[y][0]];
                S = [c]+S1; #decidir que lado [c]+S1 o S1+[c] TEST
        #if the node have 2 sons with k in the label:
        elif len_aux == 2 :
                nodes_iter_s1 = tree.breadth_first_search(aux[0]);
                nodes_iter_s2 = tree.breadth_first_search(aux[1]);
                S1 = [y1 for y1 in nodes_iter_s1 if k == GLOBAL_LABELS[y1][0]];
                S2 = [y2 for y2 in nodes_iter_s2 if k == GLOBAL_LABELS[y2][0]];
                S1.reverse();
                S = S1+[c]+S2;

        elif len_aux > 2 : raise ValueError("more than 2 sons of size k="+str(k)+". This
```

```
should not happen."); #This could not be necessary. TEST

        if verbose : print "Making the sequence S - S:",S;
        # Continuing with the paper:
        # For i:=1 to s do (this is python, so the syntaxes change a lot because python
have another way to see the "for cycle")
        for v in S :
                if not v in tree : raise ValueError("node "+str(v)+" was not found in the
tree"); #This could not be necessary. TEST

                if verbose : print "-Adding node of the sequence to the layout -
node:",v,"label["+str(v)+"]:",GLOBAL_LABELS[v],"layout:",layout;

                layout.append(v);
                # In this line is suppose to be the line "delete v_i from T" but i had to
move it because I need v_i in the tree for the next part.

                # I added a rule that was not in the paper:
                # paper: for all children y of v_i do layout(y);
                # what I did: for all children y of v_i such that y is not in S do
layout(y)
                childens = [];
                for son in tree.neighbors_out(v) :
                        if not son in S :
                                childens.append(son);
                tree.delete_vertex(v); #this is the line that was supposed to be before.

                if verbose and len(childens) > 0 : print "-Checking the children of the
node of the sequence - Amount of children:",len(childens);

                for y in childens :

                        if verbose : print "--Starting recursive call with a child -
layout("+str(y)+")";

                        layout_vs(tree, y, GLOBAL_LABELS, layout, verbose);

                if v == c and x != c :

                        if verbose : print "-Starting recursive call with the initial node
of the call - layout("+str(x)+")";

                        layout_vs(tree, x, GLOBAL_LABELS, layout,verbose);

def is_critical(label):
        if len(label) == 0 : raise ValueError("Empty Label");
        if label[-1] != "'" : return True;
        return False;
def is_k_critical(k, label):
        if len(label) == 0 : raise ValueError("Empty Label");
        if label[-1] != "'" and label[-1] == k : return True;
        return False;
```