Algorithms for the Vertex Cover Problem on Large Graphs

Eric Angel ¹, Romain Campigotto ¹ and Christian Laforest ²

¹ Laboratoire IBISC, Université d’Evry
Tour Evry 2, 1er étage
523 place des Terrasses de l’Agora
91000 Evry, France

² LIMOS, UMR 6158 CNRS
Université Blaise Pascal
Campus scientifique des Cézeaux
63173 Aubière cedex, France

— Discrete Optimization and Algorithmic —

IBISC – Université d’Evry-Val d’Essonne
Tour Evry 2, 523 place des Terrasses de l’Agora
91000 Evry Cedex, France
Les rapports de recherche d’IBISC sont disponibles aux formats PostScript® et PDF® à l’URL :

http://www.ibisc.univ-evry.fr/Vie/TR

Research reports from IBISC are available in PostScript® and PDF® formats at the URL:

http://www.ibisc.univ-evry.fr/Vie/TR

© March 2010 by Eric Angel, Romain Campigotto  and Christian Laforest
Algorithms for the Vertex Cover Problem on Large Graphs

Eric Angel, Romain Campigotto and Christian Laforest

{angel,campigotto}@ibisc.univ-evry.fr
christian.laforest@isima.fr

Abstract

In a classical treatment of an optimization graph problem, the whole graph is available in the machine. It can be modified, updated (vertices can be marked, edges deleted, etc.) and the solution can be stored. However, many applications produce now data that are too large and are impossible to store and treat in this model.

We focus in this paper on the treatment of the classical NP-complete minimization Vertex Cover problem in large graphs. The difficulty is twofold: to the intrinsic NP-completeness is added the difficulty to manipulate the input graph with severe restrictions: (1) the input graph must not be modified (integrity of the input instance), (2) the graph must be “scanned” piece by piece since it is too large to be entirely loaded in the main memory and (3) the result must be sent to an output memory once a new piece of solution is calculated.

Hence, we suppose that the size and the source of the input graph impose severe treatment constraints. After modeling this situation, we show that most of the known approximation algorithms are not compatible with our model. In a second part, we propose three algorithms (that are variations on the same “root” list algorithms) that satisfy our severe constraints. In the following, we compare them by giving exact formulas expressing the expected size of the Vertex Cover constructed by the algorithms on specific designed graphs.

General Terms: Algorithms, Measurement, Performance, Theory

Additional Key Words and Phrases: large graphs, Vertex Cover, mean analysis, approximation
## Contents

1 Introduction ............................................................... 6  
   1.1 About I/O-Efficient algorithms and Streaming .................. 7  
   1.2 A quick overview of existing algorithms for Vertex Covering .. 7  
   1.3 Organization of the paper ........................................ 8  

2 Description of our 3 algorithms ...................................... 8  
   2.1 The algorithm LISTLEFT ........................................... 9  
   2.2 The algorithm SORTED-LL ......................................... 11  
   2.3 The algorithm ANTI SORTED-LL .................................. 13  

3 Mean analysis on graph families ...................................... 14  
   3.1 Graphs where SORTED-LL  
       is the best in expectation  
       ............................................................... 17  
       3.1.1 Graphs where $SLL \prec LL \prec SLL$ ......................... 17  
       3.1.2 Graphs where $SLL \prec SLL \prec LL$ ....................... 18  
       3.1.3 Graphs where $SLL \equiv SLL \prec LL$ ....................... 19  
   3.2 Graphs where ANTI SORTED-LL is the best in expectation .... 20  
       3.2.1 Graphs where $SLL \prec LL \prec SLL$ ....................... 20  
       3.2.2 Graphs where $SLL \prec SLL \prec SLL$ .................... 21  
   3.3 Graphs where LISTLEFT is the best in expectation .......... 22  
       3.3.1 Graphs where $LL \prec SLL \prec SLL$ ...................... 22  
       3.3.2 Graphs where $LL \prec SLL \prec SLL$ ...................... 24  
   3.4 Summary of mean analysis realized on graph families ......... 24  
   3.5 Experiments on random graphs ................................... 25  

4 Properties of our algorithms on trees ............................... 25  
   4.1 General properties ................................................ 26  
   4.2 Properties of LISTLEFT ........................................... 29  
   4.3 Properties of ANTI SORTED-LL .................................. 30  
   4.4 Properties of SORTED-LL .......................................... 30  
   4.5 SORTED-LL and ANTI SORTED-LL on paths ..................... 31  

5 An equivalent edge-by-edge algorithm for SORTED-LL .......... 33  

6 Conclusion .................................................................. 34  

A A different proof for Theorem 9 ...................................... 37  

B A so-called better list algorithm: LISTRIGHT ...................... 39
1 Introduction

Nowadays, many fields such as biology, meteorology, finance, etc., produce large amount of data. These data are usually collected and stored on large distributed databases called data warehouses and then must be exploited, analyzed, etc. During this process, some optimization procedures have sometimes to be run on these data. Our goal is to propose algorithmic methods adapted for optimization problem on such large instances.

Our model of access to data. An illustration of our model is given in Figure 1. Suppose that we have one standard computer called “Processing Unit” for manipulating data and running the treatment algorithm. The input data are stored on a data warehouse called “Input data”. The output data are stored on a data warehouse called “Result”. The access of these data warehouses is subject to a set of constraints:

1. The data cannot be modified (read-only) since they often come from experimentations and can be used by different users for different goals. The integrity of input data must be preserved.

2. The data cannot be loaded totally into memory of the processing unit since the whole amount is too large for the memory of only one machine.

3. The solution must be sent in a data warehouse, called here “Result”. Moreover, in many cases, the solution is (in order of magnitude) as large as the input data, i.e. impossible to maintain in the memory of the processing unit. This means that the solution must be sent piece by piece to “Result” as soon as they are produced. In this situation, if the algorithm in the processing unit needs to use or change a past piece of the solution, it must load it from “Result”. This can be complex and take time. We adopt here a radical point of view in proposing methods that scan data and send final results as soon as they are produced, without keeping in memory trace of past computation and without modifying past part of the solution.

![Figure 1: Model’s overview](image)

We have chosen to deal with a well-known classical NP-Complete optimization graph problem, the Vertex Cover problem [11, 9], that has received a particular attention these last decades. In particular, this problem occurs in many concrete applications, such as the alignment of multiple sequences [14], the resolution of biological conflicts [16] or the network monitoring [18]; and many approximation algorithms have been proposed (see the section of [1] devoted to this problem or report to Subsection 1.2 for some examples).
Graphs $G = (V, E)$ considered throughout this paper are undirected, simple and unweighted. We denote by $n$ the number of vertices ($n = |V|$) and by $m$ the number of edges ($m = |E|$). For any vertex $u \in V$, we denote by $N(u)$ the set of neighbors of $u$ (i.e. the set of vertices sharing an edge with $u$), $d(u) = |N(u)|$ the degree of $u$ (i.e. the number of neighbors) and $\Delta$ the maximum degree of $G$.

The Vertex Cover problem. A cover $C$ of $G$ is a subset of vertices such that every edge $e = uv \in E$ contains (or is covered by) at least one vertex of $C$, that is $C \subseteq V$ and $\forall e = uv, u \in C$ or $v \in C$. The Vertex Cover problem is to find a cover of minimum size. We note $OPT(G)$ the size of an optimal cover of graph $G$.

Labeling of nodes. In real applications, the vertices have labels (depending on the applications domain) which are assumed to be pairwise distinct and that can be ordered (for example by lexicographic order). We formalize this as follows. The vertices of graph $G = (V, E)$ are labeled by a function $L$ giving, for each vertex $u$, a unique label $L(u) \in [1, n]$. We call labeled graph a graph on which we apply a labeling function $L$. We note $L(G) = (L(V), E)$ such a graph. We call right neighbors of $u$ the neighbors of $u$ having a label larger than $u$. We call left neighbors of $u$ the neighbors of $u$ having a label smaller than $u$. We suppose that graphs are stored by adjacency list. For any vertex $u \in L(V)$, the algorithm (running on the processing unit) can access to its degree and, vertex by vertex, to its neighbors (left and right) on “Input data”.

1.1 About I/O-Efficient algorithms and Streaming

The I/O-Efficient algorithms (see [17] for a survey) are subject to a special study in algorithmic: the evaluation of the number of requests to the outside (the external data warehouses “Input data” and “Result” in our model). The number of requests generated by our algorithms could also be considered, but we will focus here on the quality of the solution obtained.

Another domain which can be linked to our model is streaming (see [12] for a survey). In the streaming model, the graph is also processed piece by piece (mostly edge by edge) and the size of the memory is limited. But, as it is said in Theorem 2.3 of [12], the memory space required is $\Omega(n)$ ($n$ is the number of vertices in the graph). Moreover, the sub-models described by T. C. O’Connell [12] relax the memory constraint (for the Semi-Streaming model) or add processes not adapted to our model (this is the case for W-Stream and Stream-Sort models). Thus, a priori, streaming algorithms are not adapted to our model.

1.2 A quick overview of existing algorithms for Vertex Covering

Many solutions have been proposed for the Vertex Cover problem. As it is NP-complete, there is no polynomial algorithm constructing an optimal Vertex Cover. Most of the methods are approximation algorithms or heuristics. In this subsection, we give a rapid view of existing categories of algorithms.

A well-known heuristic is to select a vertex of maximum degree and delete this vertex and its incident edges from the graph\(^1\), until all edges have been removed. This heuristic is called Maximum Degree

\(^1\)The degrees of vertices are updated.
Greedy (see [13]). It has an approximation ratio at most \( H(\Delta) \approx \log \Delta \), where \( H(\Delta) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{\Delta} \) is the harmonic number of rank \( \Delta \).

Another popular solution is EdgeDeletion. It has been proposed by Gavril in 1979 (see [9]). It constructs a maximal matching of the input graph and returns the vertices of the matching. It has an approximation ratio at most 2. It is the best known constant approximation ratio.

The best known result (in the worst case point of view) is the solution proposed by G. Karakostas [10], which proves that an algorithm with an approximation ratio of \( 2 - \theta \left( \frac{1}{\sqrt{\log n}} \right) \) can be found.

Inspired by Maximum Degree Greedy, the algorithm Greedy Independent Cover processes as follows: choose a vertex of minimum degree, select its neighbors and delete all these vertices with their incident edges from the graph, until all edges have been removed. The algorithm Greedy Independent Cover has an approximation ratio at least \( \frac{\Delta}{2} \) and is optimal for trees (see Section 4 for more details).

Another well-known heuristic is to construct a DFS spanning tree and to select the internal nodes of the tree. We call Depth First Search [15] this heuristic. It has an approximation ratio of 2 but has bad performances on paths.

So, there exists many solutions for the Vertex Cover problem but most of these solutions do not respect our constraints, because they modify the input data or because they have to load the graph totally into the memory. Thus, with our model, we have a restricted choice of solutions.

We will study and compare in this paper list algorithms, which can be adapted to our model.

1.3 Organization of the paper

We describe in the next section 3 algorithms, which are adapted from 3 list algorithms, and we show that they are suitable to our model. To compare the quality of the Vertex Cover constructed, we propose in Section 3 general formulas giving the expected size of the algorithms and apply them on different families of graphs. This study shows that none of these 3 algorithms can be elected as the best one. In Section 4, we present properties of our 3 algorithms specifically on trees. In Section 5, we present an edge-by-edge adaptation for our algorithms. Finally, we conclude in Section 6.

2 Description of our 3 algorithms

The algorithms described in this paper are adapted from list algorithms. We adapt in this section 3 list algorithms: ListLeft, Sorted-LL and Anti Sorted-LL. These algorithms are inspired by D. Avis and T. Imamura and their article A list heuristic for Vertex Cover [2]. An important result showed in this paper is that any list algorithms cannot have an approximation ratio better than \( \frac{\sqrt{\Delta}}{2} \).

List algorithms and our model. As it has been presented in [5] and in [4], a list is a permutation of the \( n \) vertices of the graph. A list algorithm processes the graph by scanning (or reading) vertices in a deterministic order without modifying the input graph. More precisely, in our model, when the processing unit scans a vertex, it sends a request to “Input data” and gets it. It gets neighbors one by one, by sending a request for each neighbor. Also, when a list algorithm scans a vertex, it must then
take an irreversible decision on it: either the vertex is in the cover or not. So, as in an online model, if
the algorithm decides that a vertex belongs to the solution, it is put immediately and definitively into the
cover (and sent to “Result”), once it is scanned.
List algorithms are good candidates to our model. However, an important point is that the decision for a
given vertex \( u \) must be independent of the decisions taken for the previously scanned vertices (otherwise,
the processing unit must maintain in memory the trace of already processed vertices).

**Equivalence between vertex lists and labeled graphs.** The vertices are naturally labeled in real
applications. To simplify, we suppose here that labels are ordered and unique, between 1 and \( n \). Hence,
enumerating the vertices in the order of labels create an associated list (for example, the vertex labeled
by 4 is located in the list just before the vertex labeled by 5 and immediately after the vertex labeled by
3). At the opposite, if a list of vertices is given, we just can label each vertex by its rank in the list and we
obtain a corresponding labeled graph. In the following, we will used indiscriminately the two notions.

**Remark 1.** Unlike list algorithms, we will see that with our 3 algorithms adapted on labeled graphs, we
don’t need to scan vertices by increasing order of labels (i.e. from 1 to \( n \)): vertices can be scanned in
arbitrary order.

### 2.1 The algorithm \textsc{ListLeft}

In the version described by F. Delbot and C. Laforest [5], the input vertex list is scanned from left to
right (i.e. from 1 to \( n \)) and the condition for \( u \) is as follows: “\( u \) is added to the Vertex Cover if and only
if it has at least one (left or right) neighbor not in the cover”. Equivalently, we can say: “\( u \) is added to
the cover if and only if it has at least one right neighbor”. Indeed, when we are scanning \( u \), necessarily
all the vertices appearing after it (i.e. on the right) are not in the cover. Thus, if \( u \) has right neighbors,
necessarily, they are not in the cover. Also, we don’t need to verify if \( u \) has left neighbors not in the
cover, because when we are scanning \( u \), we are sure that all of its left neighbors are already in the cover,
because symmetrically, \( u \) is a right neighbor of its left neighbors.

#### Algorithm 1 \textsc{ListLeft (LL)}

**Require:** labeled graph \( L(G) = (L(V), E) \)

\[
\begin{align*}
C & \leftarrow \emptyset \\
\text{for each vertex } u \in L(V) \text{ do} & \\
& \text{if } u \text{ has at least one right neighbor then} \\
& \quad C \leftarrow C \cup \{u\} \\
\text{return } C
\end{align*}
\]

Unlike the list version presented in [5], in our version adapted on labeled graphs (see Algorithm 1),
whatever the processing order of vertices, the result is the same, because on labeled graphs, notions of
left and right neighbors are based on labels given before the execution of the algorithm and not on the
apparition order of vertices. Hence, we can scan vertices in arbitrary order. However, if we modify the
labels, the result will not be the same.
Figure 2: Execution of LISTLEFT on a labeled graph

Figure 2 shows an example of algorithm LISTLEFT executed on a labeled graph. The solution returned is the vertices represented by squares (1, 2, 3 and 4).

**Approximation ratio.** LISTLEFT has an approximation ratio of at least $\Delta$ (on stars, LISTLEFT can return all the leaves, if the center is labeled by $n$) and at most $\Delta + 1$ (see Remark 2).

**Remark 2.** For any graph $G = (V, E)$, $n \leq \text{OPT}(G)(\Delta + 1)$, where $\Delta$ is the maximum degree of $G$.

**Lemma 1.** For any labeled graph $L(G)$, LISTLEFT never returns the vertex labeled by $n$.

**Proof.** Let $u$ be any labeled graph $L(G)$ constituted by $n$ vertices, and $u$ be the vertex of label $n$ ($L_u = n$). Hence, vertex $u$ has no right neighbor and is not selected by the algorithm. \qed

**Variability of LISTLEFT.** On labeled graphs, LISTLEFT is deterministic. But there exists exactly $n!$ ways to label a graph. For example, consider 2 labeling functions $L$ and $L'$. From a graph $G = (V, E)$, we can produce 2 different labeled graphs: $L(G) = (L(V), E)$ and $L'(G) = (L'(V), E)$.

We call *variability* of an algorithm the difference between the size of the best solution it can return and the size of the worst solution it can return. Let $L_G$ be the set of all possible labeling function on $G$. For LISTLEFT, the variability is defined by the size of the best solution,

$$B\{\mathcal{L}(G)\} = \min_{L \in L_G} \mathcal{L}(L(G))$$

and by the size of the worst solution,

$$W\{\mathcal{L}(G)\} = \max_{L \in L_G} \mathcal{L}(L(G))$$

We will show that for any graph, algorithm LISTLEFT can return an optimal solution or return the worst solution, depending on the labels. According to Lemma 1, the size of the worst solution is bounded above by $n - 1$.

First, we show that on any graph, LISTLEFT can always return the worst solution.

**Theorem 1.** For any connected graph $G$, $W\{\mathcal{L}(G)\} = n - 1$. 
Proof. Let $T$ be any spanning tree of $G$. Let $r$ be any vertex of $T$. Label the vertices as follows. Vertex $r$ gets label $n$. The $d_1$ neighbors/children of $r$ in $T$ get the $d_1$ labels $(n - d_1, \ldots, n - 1)$ just before. The $d_2$ vertices at distance 2 get the $d_2$ preceding labels $(n - d_1 - d_2, \ldots, n - d_1)$, etc. until each vertex receives a label, level by level. In this order, noted $L$, each vertex $u$ has exactly one right neighbor: its parent $v$ in the tree $T$ rooted at $r$ ($u$ and $v$ are neighbors and $L_u < L_v$; moreover, the other neighbors of $u$ are children of $u$, are in the “next level” and thus receive a lower label). As $T$ is a tree spanning $G$, in the order $L$, each vertex $u$ of $G$ also has a right neighbor (at least the one in $T$). Hence, there exists a label $L$ of vertices of $G$ for which LISTLEFT returns all the vertices, except exactly one. See illustration at Figure 3. In this tree, LISTLEFT returns $n - 1$ vertices. Note that the dotted lines correspond to edges which are present in the graph but not in the spanning tree.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{tree.png}
\caption{Example of a labeled spanning tree of a graph}
\end{figure}

Remark 3. If $G$ is not connected and has $c$ connected components, then $\mathcal{W}\{\mathcal{L}\mathcal{L}(G)\} = n - c$.

Theorem 2. For any graph $G$, $B\{\mathcal{L}\mathcal{L}(G)\} = OPT(G)$, i.e. algorithm LISTLEFT can return an optimal solution for any graph.

Proof. Let $C^*$ be an optimal solution for the Vertex Cover problem. For any graph $G$, we can always obtain a labeled graph $L^*(G)$ where the vertices of $C^*$ get labels between 1 and $OPT(G)$ and where the vertices of $V \setminus C^*$ get labels between $OPT(G) + 1$ and $n$. If algorithm LISTLEFT is executed on such a labeled graph, it returns all the vertices of $C^*$ (since each vertex $u$ of $C^*$ has at least a neighbor in $V \setminus C^*$ with a higher label) and no vertex of $V \setminus C^*$ (because $V \setminus C^*$ is an Independent Set and no vertex has a right neighbor in this set).

2.2 The algorithm SORTED-LL

The algorithm SORTED-LL, which is also denoted by $SLL$, is an extension of LISTLEFT and has been described in the article of D. Avis and T. Imamura [2]. It executes LISTLEFT on a list sorted by decreasing order of degrees. Our model is not adapted for sorting vertices by degree. So, we present an alternative version of SORTED-LL, which is suitable to our model.
Equivalence between sorted vertex lists and labeled graphs. Let $L(G)$ be any labeled graph. We suppose that for each vertex $u \in L(V)$, we can access to degrees of its neighbors. \textsc{Sorted-LL} takes its decision for vertex $u$ as follows. If there is at least one $v \in N(u)$ with lower degree, select $u$ ($u$ has a right neighbor); otherwise, if $u$ has only neighbors with higher degree, $u$ is not selected ($u$ has only left neighbors). The last case is when $u$ has neighbors with the same degree (without having neighbors of lower degree). The selection of $u$ depends on the label $L$ of its neighbors. If there is at least one such neighbor with higher label, $u$ is selected otherwise, it is not selected.

This is the principle of Algorithm 2. We can see that a global sort of the vertices is useless because this sorting vertices can be done “locally” (i.e. only with informations on neighbors).

\begin{algorithm}
\caption{\textsc{Sorted-LL} ($SLL$)}
\begin{algorithmic}
\Require labeled graph $L(G) = (L(V), E)$
\State $C \leftarrow \emptyset$
\For {each vertex $u \in L(V)$}
\If {u has at least one neighbor with a lower degree or a right neighbor with the same degree}
\State $C \leftarrow C \cup \{u\}$
\EndIf
\EndFor
\Return $C$
\end{algorithmic}
\end{algorithm}

Figure 4 shows an example of algorithm \textsc{Sorted-LL} executed on a labeled graph. The solution returned is the vertices represented by squares (3 and 4).

Approximation ratio. D. Avis and T. Imamura have proved that \textsc{Sorted-LL} has an approximation ratio of at most $\sqrt{3} + \frac{3}{2}$ (see Theorem 3 in [2]). This bound is almost tight on Avis-Imamura graphs (see Definition 1 and Figure 5).

\textbf{Definition 1} (Avis-Imamura graphs). Let $AI_a = (X \cup Y, E)$ be a particular bipartite graph called “Avis-Imamura graph”, where:

- $X = X_1 \cup X_2$:
  - $|X_1| = a$ and $\forall u \in X_1, d(u) = a^2$,
  - $|X_2| = a$ and $\forall u \in X_2, d(u) = a$, 


\begin{itemize}
\item $|Y| = a^2$ and $\forall u \in Y$, $d(u) = a + 1$;
\item the subgraph induced by the vertices of $X_1 \cup Y$ is a complete bipartite graph,
\item each vertex $u \in X_2$ is connected to exactly $a$ vertices of $Y$ and each vertex $v \in Y$ is connected to exactly $1$ vertex of $X_2$.
\end{itemize}

Figure 5: Example of Avis-Imamura graph, with $a = 3$

On any Avis-Imamura graph, SORTED-LL returns all the $a^2 + a$ vertices of $X_1 \cup Y$, while the optimal solution is the $2a$ vertices of $X = X_1 \cup X_2$. If $\Delta = a^2$, then the ratio between a solution constructed by SORTED-LL and an optimal solution is $\frac{\sqrt{\Delta}}{2} + \frac{1}{2}$.

\section{The algorithm ANTI SORTED-LL}

The algorithm ANTI SORTED-LL, which is also denoted by $\overline{SLL}$, works like SORTED-LL on sorted lists, but it reads the lists from the end to the beginning. As with SORTED-LL, we cannot execute \textit{a priori} this algorithm in our model. So, we present an alternative version of ANTI SORTED-LL, which is suitable to our model (see Algorithm 3).

As for SORTED-LL, we suppose that for each vertex $u \in L(V)$, we can access to the degrees of its neighbors. We have showed in the previous subsection that there exists an equivalence between sorted vertex lists and labeled graphs. Thus, as for SORTED-LL, we can give an alternative adapted version of ANTI SORTED-LL.

\begin{algorithm}
\caption{ANTI SORTED-LL ($\overline{SLL}$)}
\begin{algorithmic}
\Require labeled graph $L(G) = (L(V), E)$
\State $C \leftarrow \emptyset$
\For {each vertex $u \in L(V)$}
\If {$u$ has at least one neighbor with a larger degree or a left neighbor with the same degree}
\State $C \leftarrow C \cup \{u\}$
\EndIf
\EndFor
\State \bf return $C$
\end{algorithmic}
\end{algorithm}

Figure 6 shows an example of algorithm ANTI SORTED-LL executed on a labeled graph. The solution returned is composed by the vertices represented by squares (1, 2, 5 and 6).

\textbf{Approximation ratio.} ANTI SORTED-LL has an approximation ratio of at least $\Delta$ (on stars, ANTI SORTED-LL always returns all the leaves) and at most $\Delta + 1$ (see Remark 2).
Figure 6: Execution of ANTI-SORTED-LL on a labeled graph

3 Mean analysis on graph families

Our three algorithms work deterministically on any given labeled graph. However, the labels of vertices are often totally arbitrary and only come from the application domains. Different labels give different results, i.e. Vertex Cover of different sizes. In the rest of the paper, we will compare our algorithms on the size of the returned Vertex Cover. As there are $n!$ possible labeling functions, we want to calculate the expectation of the sizes of the constructed covers. In the following, we give general formulas to calculate such an expectation on any graph for each algorithm. Then, we apply these formulas on class of graphs to compare the quality of Vertex Cover constructed by our algorithms. This shows that none of these three is always better than the other two ones.

The expected size of solution constructed by LISTLEFT. We first express the probability for any vertex $u$ to be in a solution constructed by LISTLEFT, considering that the labeling is one of the $n!$ possible ones, with the same probability. Then, we use this probability to give an exact formula for the expected size of Vertex Cover constructed (always with the same equiprobability hypothesis on labeling functions).

Lemma 2. Let $G = (V, E)$ be any graph and $C_{\text{LL}}$ a cover constructed by LISTLEFT on $G$. Consider a vertex $u$ of $G$. The probability that $u \in C_{\text{LL}}$ is

$$P[u \in C_{\text{LL}}] = 1 - \frac{1}{d(u) + 1}$$

(1)

Proof. Let $L(G) = (L(V), E)$ be any labeled graph. Consider a vertex $u$ of $L(G)$. $u$ is selected by LISTLEFT if and only if it has at least one right neighbor. Otherwise, $u$ is not chosen if and only if it has no right neighbor, which means that all of its neighbors have smaller labels than it. This event appears with a probability of $\frac{d(u)!}{(d(u) + 1)!}$. Indeed, if we sort $u$ and the $d(u)$ vertices of $N(u)$ by increasing order of labels, there are $(d(u) + 1)!$ possible permutations, and the number of permutations such that $u$ is in the last position is $d(u)!$. Thus, we have

$$P[u \in C_{\text{LL}}] = 1 - P[u \notin C_{\text{LL}}]$$

$$= 1 - \frac{d(u)!}{(d(u) + 1)!}$$

$$= 1 - \frac{1}{d(u) + 1}$$
Theorem 3 (Expectation of \textsc{ListLeft}). The expected size of solution constructed by \textsc{ListLeft} on any graph \(G\) (noted \(\mathbb{E}[\mathcal{L}\mathcal{L}(G)]\)) is

\[
\mathbb{E}[\mathcal{L}\mathcal{L}(G)] = n - \sum_{u \in V} \frac{1}{d(u) + 1}
\] (2)

Proof. We apply (1) and use the linearity of expectation by summing probabilities for each vertex of \(G\). Thus, we have

\[
\mathbb{E}[\mathcal{L}\mathcal{L}(G)] = \sum_{u \in V} \mathbb{P}[u \in C_{\mathcal{L}\mathcal{L}}]
\] = \[1 - \frac{1}{d(u) + 1}\]
= \[n - \sum_{u \in V} \frac{1}{d(u) + 1}
\]

The expected size of solution constructed by \textsc{Sorted-LL}. We obtain the expected value of \textsc{Sorted-LL} in the same way as for \textsc{ListLeft}. Let \(G = (V, E)\) be any graph and \(C_{\mathcal{S}\mathcal{L}\mathcal{L}}\) a cover constructed by \textsc{Sorted-LL} on \(G\).

Let \(S\) be the set of vertices with no neighbor of lower degree: \(S = V \setminus \{u \mid \exists v \in N(u), d(v) < d(u)\}\). Intuitively, we have

\[
\mathbb{P}[u \in C_{\mathcal{S}\mathcal{L}\mathcal{L}} \mid u \notin S] = 1
\] (3)
because \(u\) is necessarily chosen by \textsc{Sorted-LL}.

Let \(\Lambda(u) = \{v \mid v \in N(u) \land d(v) = d(u)\}\) and \(\sigma(u) = |\Lambda(u)|\) be the number of neighbors of \(u\) having the same degree as \(u\). So, we have

\[
\mathbb{P}[u \in C_{\mathcal{S}\mathcal{L}\mathcal{L}} \mid u \in S] = 1 - \frac{1}{\sigma(u) + 1}
\] (4)

(4) is obtained in the same way as (1) (for more details, see the proof of Lemma 2). Note that the isolated vertices\(^2\) belong to \(S\). So, \(\mathbb{P}[u \in C_{\mathcal{S}\mathcal{L}\mathcal{L}} \mid d(u) = 0] = 0\).

Theorem 4 (Expectation of \textsc{Sorted-LL}). The expected size of solution constructed by \textsc{Sorted-LL} on any graph \(G\) (noted \(\mathbb{E}[\mathcal{S}\mathcal{L}\mathcal{L}(G)]\)) is

\[
\mathbb{E}[\mathcal{S}\mathcal{L}\mathcal{L}(G)] = n - \sum_{u \in S} \frac{1}{\sigma(u) + 1}
\] (5)

\(^2\)An isolated vertex \(u\) is a vertex for which \(d(u) = 0\) (and therefore \(\sigma(u) = 0\)).
Proof. Let \( s = |S| \) be the number of vertices in \( S \). We apply (3) and (4) and use the linearity of expectation by summing probabilities for respectively each vertex of \( V \setminus S \) and \( S \). Thus, we have

\[
\mathbb{E}[\mathcal{SLL}(G)] = \sum_{u \notin S} \mathbb{P}[u \in C_{\mathcal{SLL}} | u \notin S] + \sum_{u \in S} \mathbb{P}[u \in C_{\mathcal{SLL}} | u \in S]
\]

\[
= \sum_{u \notin S} 1 + \sum_{u \in S} \left( 1 - \frac{1}{\sigma(u) + 1} \right)
\]

\[
= n - s + s - \sum_{u \in S} \frac{1}{\sigma(u) + 1}
\]

\[
= n - \sum_{u \in S} \frac{1}{\sigma(u) + 1}
\]

\[\square\]

**The expected size of solution constructed by Anti Sorted-LL.** We obtain the expected value of Anti Sorted-LL in the same way as for Sorted-LL. Let \( G = (V, E) \) be any graph and \( C_{\overline{\mathcal{SLL}}} \) a cover constructed by Anti Sorted-LL on \( G \).

Let \( \overline{S} \) be the set of vertices with no neighbor of larger degree: \( \overline{S} = V \setminus \{u \mid \exists v \in N(u), d(v) > d(u)\} \). As for Sorted-LL, we have

\[
\mathbb{P}[u \in C_{\overline{\mathcal{SLL}}} | u \notin \overline{S}] = 1 \tag{6}
\]

because \( u \) is necessarily chosen by Anti Sorted-LL.

Using the notation \( \sigma(u) \) (already been introduced for Sorted-LL), we have

\[
\mathbb{P}[u \in C_{\overline{\mathcal{SLL}}} | u \in \overline{S}] = 1 - \frac{1}{\sigma(u) + 1} \tag{7}
\]

(7) is obtained in the same way as (4). Note that the isolated vertices belong to \( \overline{S} \). So, as for Sorted-LL, \( \mathbb{P}[u \in C_{\overline{\mathcal{SLL}}} | d(u) = 0] = 0 \).

**Theorem 5 (Expectation of Anti Sorted-LL).** The expected size of solution constructed by Anti Sorted-LL on any graph \( G \) (noted \( \mathbb{E}[\overline{\mathcal{SLL}}(G)] \)) is

\[
\mathbb{E}[\overline{\mathcal{SLL}}(G)] = n - \sum_{u \in S} \frac{1}{\sigma(u) + 1} \tag{8}
\]

Proof. Let \( \overline{s} = |\overline{S}| \) be the number of vertices in \( \overline{S} \). We apply (6) and (7) and use the linearity of
expectation by summing probabilities for respectively each vertex of \( V \setminus \overleftarrow{S} \) and \( \overleftarrow{S} \). We get

\[
\mathbb{E}[\overleftarrow{SLL}(G)] = \sum_{u \not\in \overleftarrow{S}} \mathbb{P}[u \in C_{\overleftarrow{SLL}} | u \not\in \overleftarrow{S}] + \sum_{u \in \overleftarrow{S}} \mathbb{P}[u \in C_{\overleftarrow{SLL}} | u \in \overleftarrow{S}]
\]

\[
= \sum_{u \not\in \overleftarrow{S}} 1 + \sum_{u \in \overleftarrow{S}} \left(1 - \frac{1}{\sigma(u) + 1}\right)
\]

\[
= n - \overleftarrow{S} + \overleftarrow{S} - \sum_{u \in \overleftarrow{S}} \frac{1}{\sigma(u) + 1}
\]

\[
= n - \sum_{u \in \overleftarrow{S}} \frac{1}{\sigma(u) + 1}
\]

\[
\]

A first result is that our 3 algorithms have the same expected values on regular graphs. A regular graph \( G_\Delta \) is a graph where all the vertices have the same degree, \( \Delta \). We can remark that, in a regular graph \( G_\Delta \), \( S = \overleftarrow{S} = V \) and \( \sigma(u) = d(u) \) for each vertex \( u \) of \( G_\Delta \). Hence, the 3 algorithms have the same behavior on regular graphs and their expected values are equal:

\[
\mathbb{E}[\overrightarrow{LL}(G_\Delta)] = \mathbb{E}[SLL(G_\Delta)] = \mathbb{E}[\overleftarrow{SLL}(G_\Delta)] = n \left( \frac{\Delta}{\Delta + 1} \right)
\]

Now, we give some results about mean analysis of the 3 algorithms on different graph families.

**Remark about notations for titles of the following subsections.** In order to simplify notations, we note \( A \prec B \) when we want to say “algorithm \( A \) is better in expectation than algorithm \( B \)”, and \( A \equiv B \) when we want to say “algorithm \( A \) is equivalent in expectation to algorithm \( B \)”.

### 3.1 Graphs where \textsc{Sorted-LL} is the best in expectation

#### 3.1.1 Graphs where \textsc{SLL} < \textsc{LL} < \textsc{SLL}

**On paths.** Let \( P_n \) be a path with \( n \) vertices. We can apply general expectation formulas on paths. A path \( P_n \) is constituted by \( n - 2 \) vertices of degree 2 and by 2 vertices of degree 1. So, for all \( n > 1 \), we have

\[
\mathbb{E}[\overrightarrow{LL}(P_n)] = n - \frac{n - 2}{2 + 1} - \frac{2}{1 + 1} = \frac{2n}{3} - \frac{1}{3}
\]

For \textsc{Sorted-LL}, the set \( S \) contains all the vertices of \( P_n \), excepted the neighbors of the leaves. So, for all \( n > 1 \), we have

\[
\mathbb{E}[SLL(P_n)] = n - \frac{n - 4}{2 + 1} - \frac{2}{0 + 1} = \frac{2n}{3} - \frac{2}{3}
\]

For \textsc{Anti Sorted-LL}, the set \( \overleftarrow{S} \) contains all the vertices of \( P_n \), excepted the 2 leaves. So, for all \( n > 1 \), we have

\[
\mathbb{E}[\overleftarrow{SLL}(P_n)] = n - \frac{n - 4}{2 + 1} - \frac{2}{1 + 1} = \frac{2n}{3} + \frac{1}{3}
\]

We can easily see that \textsc{Sorted-LL} has a better expected value than \textsc{ListLeft} and \textsc{Anti Sorted-LL} on paths.
On stars. Let $S_n$ be a star with $n$ vertices. We can apply general expectation formulas on stars. A star $S_n$ is constituted by $1$ vertex of degree $n - 1$ and by $n - 1$ vertices of degree $1$. So, for all $n > 0$, we have

$$E[LL(S_n)] = n - \frac{n - 1}{1 + 1} - \frac{1}{(n - 1) + 1} = \frac{n}{2} - \frac{1}{n + \frac{1}{2}}$$

For SORTED-LL, the set $S$ contains all the leaves of $S_n$. So, for all $n > 1$, we have

$$E[SLL(S_n)] = n - \frac{n - 1}{0 + 1} = 1$$

For ANTI SORTED-LL, the set $S$ contains only the center of the star. So, for all $n > 1$, we have

$$E[\overline{SLL}(S_n)] = n - \frac{1}{0 + 1} = n - 1$$

We can easily see that SORTED-LL has a better expected value than LISTLEFT and ANTI SORTED-LL.

3.1.2 Graphs where $SLL \prec \overline{SLL} \prec LL$

On complete bipartite graphs. Let $K_{a,b} = (X \cup Y, E)$ be a complete bipartite graph with $a + b$ vertices. We can apply general expectation formulas on complete bipartite graphs. A complete bipartite graph $K_{a,b}$ is constituted by $a$ vertices of degree $b (X)$ and by $b$ vertices of degree $a (Y)$. We suppose that $a > b$. So, we have

$$E[LL(K_{a,b})] = n - \frac{a}{b + 1} - \frac{b}{a + 1}$$

For SORTED-LL, the set $S$ contains the $b$ vertices of $Y$. So, we have

$$E[SLL(K_{a,b})] = n - \frac{a}{0 + 1} = n - a$$

For ANTI SORTED-LL, the set $S$ contains the $a$ vertices of $X$. So, we have

$$E[\overline{SLL}(K_{a,b})] = n - \frac{b}{0 + 1} = n - b$$
We can easily see that SORTEO-LL has a better expected value than ANTI SORTEO-LL on complete bipartite graphs. Now, we show that ANTI SORTEO-LL is better in expectation than LISTLEFT.

\[ \mathbb{E}\left[\text{LL}(K_{a,b})\right] - \mathbb{E}\left[\text{SLLE}(K_{a,b})\right] = a \left( \frac{b}{1 + a} - \frac{1}{1 + b} \right) \]

If \( b > -1 + \sqrt{5 + 4a} \) (we have supposed that \( a > b \)), then \( a \left( \frac{b}{1 + a} - \frac{1}{1 + b} \right) > 0 \) and thus, the expected value of ANTI SORTEO-LL is better than the LISTLEFT one.

### 3.1.3 Graphs where \( S\text{LL} \equiv \text{SLLE} < \text{LL} \)

**On Avis-Imamura graphs.** Let \( AI_a \) be an Avis-Imamura graph with \( a^2 + 2a \) vertices (report to Definition 1 for a complete description). We can apply general expectation formulas on these graphs. So, for all \( a > 1 \), we have

\[ \mathbb{E}\left[\text{LL}(AI_a)\right] = n - \frac{a^2}{(a + 1) + 1} - \frac{a}{a + 1} - \frac{a}{a^2 + 1} = n - \frac{a^2}{a + 2} - \frac{a}{a + 1} - \frac{a}{a^2 + 1} \]

For SORTEO-LL, the set \( S \) only contains the \( a \) vertices of \( X_2 \). So, for all \( a > 1 \), we have

\[ \mathbb{E}\left[\text{SLLE}(AI_a)\right] = n - \frac{a}{0 + 1} = n - a \]

For ANTI SORTEO-LL, the set \( \widetilde{S} \) only contains the \( a \) vertices of \( X_1 \). So, for all \( a > 1 \), we have

\[ \mathbb{E}\left[\text{SLLE}(AI_a)\right] = n - \frac{a}{0 + 1} = n - a \]

We can easily see that SORTEO-LL and ANTI SORTEO-LL are the same expected values. Now, we show that they are better than LISTLEFT.

\[ \mathbb{E}\left[\text{LL}(AI_a)\right] - \mathbb{E}\left[\text{SLLE}(AI_a)\right] = a - \frac{a^2}{a + 2} - \frac{a}{a + 1} - \frac{a}{a^2 + 1} \]

If \( a > 2 \), then SORTEO-LL and ANTI SORTEO-LL are better in expectation than LISTLEFT.
3.2 Graphs where ANTI SORTED-LL is the best in expectation

3.2.1 Graphs where $\overline{\text{SLL}} \prec \text{SLL} \prec \text{LL}$

Definition 2 (Avis-Imamura extended graphs (ASL version)). Let $AI_{a,b} = (X \cup Y, E)$ be a particular bipartite graph called "Avis-Imamura ASL graph", where:

- $X = X_1 \cup X_2$:
  - $|X_1| = a + b$ and $\forall u \in X_1, d(u) = a^2$,
  - $|X_2| = a$ and $\forall u \in X_2, d(u) = a$,
- $|Y| = a^2$ and $\forall u \in Y, d(u) = a + 1 + b$;
- the subgraph induced by the vertices of $X_1 \cup Y$ is a complete bipartite graph,
- each vertex $u \in X_2$ is connected to exactly $a$ vertices of $Y$ and each vertex $v \in Y$ is connected to exactly 1 vertex of $X_2$.

On Avis-Imamura extended graphs (ASL version). Let $AI_{a,b}$ be an Avis-Imamura ASL graph with $a^2 + 2a + b$ vertices (see Definition 2 for a complete description). We can apply general expectation formulas on these graphs. So, for all $a > 1$ and $\forall b$ such that $1 \leq b \leq a^2 - a - 2$, we have

$$\mathbb{E}[\mathcal{LL}(AI_{a,b})] = n - \frac{a^2}{(a + 1 + b) + 1} - \frac{a}{a + 1} - \frac{a + b}{a^2 + 1}$$

$$= n - \frac{a^2}{a + b + 2} - \frac{a}{a + 1} - \frac{a + b}{a^2 + 1}$$

For SORTED-LL, the set $S$ only contains the $a$ vertices of $X_2$. So, for all $a > 1$ and $\forall b$ such that $1 \leq b \leq a^2 - a - 2$, we have

$$\mathbb{E}[\overline{\text{SLL}}(AI_{a,b})] = n - \frac{a}{0 + 1} = n - a$$
For **ANTI SORTED-LL**, the set $\overleftarrow{S}$ contains the $a + b$ vertices of $X_1$. So, for all $a > 1$ and $\forall b$ such that $1 \leq b \leq a^2 - a - 2$, we have

$$
\mathbb{E}[\overleftarrow{S}LL(AI_{a,b})] = n - \frac{a + b}{0 + 1} = n - a - b
$$

We can easily see that **ANTI SORTED-LL** is better than **SORTED-LL** on these graphs. As we have

![Diagram showing vertices of $V \setminus S$ and $V \setminus \overleftarrow{S}$ on an Avis-Imamura graph $AI_{3,2}$](image)

 showed on classical Avis-Imamura graphs, **LISTLEFT** is worse in expectation than **SORTED-LL** and **ANTI SORTED-LL**, for all $a > 2$.

### 3.2.2 Graphs where $\overleftarrow{S}LL < LL < SLLL$

**Definition 3** (Avis-Imamura extended graphs (ALS version)). Let $AI^+_a = (X \cup Y, E)$ be a particular bipartite graph called “Avis-Imamura ALS graph”, where:

- $X = X_1 \cup X_2$:
  - $|X_1| = a + b$ and $\forall u \in X_1$, $d(u) = a^2$,
  - $|X_2| = a$ and $\forall u \in X_2$, $d(u) = a$,
- $Y = Y_1 \cup Y_2$:
  - $|Y_1| = a^2$ and $\forall u \in Y_1$, $d(u) = a + 1 + b$,
  - $|Y_2| = 1$ and $d(u) = a$;
- the subgraph induced by the vertices of $X_1 \cup Y_1$ is a complete bipartite graph,
- each vertex $u \in X_2$ is connected to exactly $a$ vertices of $Y_1$ and each vertex $v \in Y_1$ is connected to exactly 1 vertex of $X_2$,
- the subgraph induced by the vertices of $X_2 \cup Y_2$ is a complete bipartite graph.
On Avis-Imamura extended graphs (ALS version). Let $AI_{a,b}^+$ be an Avis-Imamura ALS graph with $a^2 + 2a + b + 1$ vertices (see Definition 3 for a complete description). We can apply general expectation formulas on these graphs. So, for all $a > 1$ and $\forall b$ such that $1 \leq b \leq a^2 - a - 2$, we have

$$E[LL(AI_{a,b}^+)] = n - \frac{a^2}{a+1+b} - \frac{a}{a+1} - \frac{a+b}{a^2+1} - \frac{1}{a+1}$$

For Sorted-LL, the set $S$ only contains the vertex of $Y_2$. So, for all $a > 1$ and $\forall b$ such that $1 \leq b \leq a^2 - a - 2$, we have

$$E[SLL(AI_{a,b}^+)] = n - \frac{1}{a+b+2} = n - 1$$

For Anti Sorted-LL, the set $\overline{S}$ contains the $a + b$ vertices of $X_1$. So, for all $a > 1$ and $\forall b$ such that $1 \leq b \leq a^2 - a - 2$, we have

$$E[SLL(AI_{a,b}^+)] = n - \frac{a+b}{0+1} = n - a - b$$

We can easily see that Anti Sorted-LL is better than Sorted-LL on these graphs. As Sorted-LL always returns $n - 1$ vertices (the worst solution), here, the algorithm LISTLEFT is better in expectation than Sorted-LL.

3.3 Graphs where LISTLEFT is the best in expectation

3.3.1 Graphs where $LL \prec SLL \prec \overline{SLL}$

On grid graphs. Let $GR_{p \times q}$ be a grid graph with $p \times q$ vertices. We can apply general expectation formulas on grid graphs. A grid graph $GR_{p \times q}$ is constituted by $(p-2)(q-2)$ vertices of degree 4, by $2(p+q-4)$ vertices of degree 3 and by 4 vertices (the corners) of degree 2. So, $\forall p > 2$ and $\forall q > 2$, we have

$$E[LL(GR_{p \times q})] = n - \frac{(p-2)(q-2)}{4+1} - \frac{2(p+q-4)}{3+1} - \frac{4}{2+1} = \frac{4n - p + q - 2}{10} - \frac{2}{15}$$

Figure 12: Vertices of $V \setminus S$ and $V \setminus \overline{S}$ on a Avis-Imamura ALS graph $AI_{3,2}^+$
For Sorted-LL, the set \( S \) contains all the vertices of \( GR_{p \times q} \), excepted \( 2(p + q - 2) \) vertices which are neighbors to the border vertices and the corner vertices. So, \( \forall p > 2 \) and \( \forall q > 2 \), we have

\[
\mathbb{E}[SLL(GR_{p \times q})] = n - \frac{(p - 4)(q - 4)}{4 + 1} - \frac{2(p + q - 8)}{2 + 1} - \frac{4}{0 + 1} = \frac{4n}{5} + \frac{2(p + q)}{15} - \frac{28}{15}
\]

For Anti Sorted-LL, the set \( \overline{S} \) contains all the vertices of \( GR_{p \times q} \), excepted \( 2(p + q - 2) \) vertices which are on the border of the grid. So, \( \forall p > 2 \) and \( \forall q > 2 \), we have

\[
\mathbb{E}[\overline{SLL}(GR_{p \times q})] = n - \frac{(p - 4)(q - 4)}{4 + 1} - \frac{2(p + q - 8)}{3 + 1} - \frac{4}{2 + 1} = \frac{4n}{5} + \frac{3(p + q)}{10} - \frac{8}{15}
\]

We can easily see that ListLeft is better in expectation than Sorted-LL and Anti Sorted-LL on grid graphs. Also, the expected value of Sorted-LL is better than the Anti Sorted-LL one, because

\[
\mathbb{E}[SLL(GR_{p \times q})] - \mathbb{E}[\overline{SLL}(GR_{p \times q})] = \frac{n + 2l}{6} + \frac{4}{3}.
\]

**On spider graphs.** Let \( W_{l \times p} \) be a spider graph with \( l \times p \) vertices\(^3\). We can apply general expectation formulas on spider graphs. A spider graph \( W_{l \times p} \) is constituted by \( n - 2l \) vertices of degree 4 and by \( 2l \) vertices of degree 3 (the borders). So, \( \forall l > 2 \) and \( \forall p > 2 \), we have

\[
\mathbb{E}[LCL(W_{l \times p})] = n - \frac{n - 2l}{4 + 1} - \frac{2l}{3 + 1} = \frac{4n}{5} - \frac{l}{10}
\]

For Sorted-LL, the set \( S \) contains all the vertices of \( W_{l \times p} \), excepted \( 2l \) vertices which are neighbors to the border vertices. So, \( \forall l > 2 \) and \( \forall p > 2 \), we have

\[
\mathbb{E}[SLL(W_{l \times p})] = n - \frac{n - 4l}{4 + 1} - \frac{2l}{2 + 1} = \frac{4n}{5} + \frac{2l}{15}
\]

For Anti Sorted-LL, the set \( \overline{S} \) contains all the vertices of \( W_{l \times p} \), excepted \( 2l \) vertices which are on the borders of the spider graph. So, \( \forall l > 2 \) and \( \forall p > 2 \), we have

\[
\mathbb{E}[\overline{SLL}(W_{l \times p})] = n - \frac{n - 4l}{4 + 1} - \frac{2l}{3 + 1} = \frac{4n}{5} + \frac{3l}{10}
\]
We can easily see that LISTLEFT is better in expectation than SORTED-LL and ANTI SORTED-LL on spider graphs. Also, the expected value of SORTED-LL is better than the ANTI SORTED-LL one, because
\[
\mathbb{E}[SLL(W_{l\times p})] - \mathbb{E}[S\bar{L}L(W_{l\times p})] = \frac{1}{6}.
\]

### 3.3.2 Graphs where $LL \prec \bar{S}LL \prec SLL$

We have not yet found such graphs.

### 3.4 Summary of mean analysis realized on graph families

We have seen in this section that expectation performances of the 3 algorithms are close. Indeed, as we can see in Table 1, no algorithm is totally dominant over the 2 others. There exists graphs for which SORTED-LL is the best, but it also exists graphs for which ANTI SORTED-LL is the best and graphs for which LISTLEFT is the best. More precisely, on some graph families, the expected values of the

<table>
<thead>
<tr>
<th>Graph families</th>
<th>Algorithms</th>
<th>LISTLEFT</th>
<th>SORTED-LL</th>
<th>ANTI SORTED-LL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular graphs</td>
<td></td>
<td>1st ex aequo</td>
<td>1st ex aequo</td>
<td>1st ex aequo</td>
</tr>
<tr>
<td>Paths</td>
<td></td>
<td>2nd</td>
<td>1st</td>
<td>3rd</td>
</tr>
<tr>
<td>Stars</td>
<td></td>
<td>2nd</td>
<td>1st</td>
<td>3rd</td>
</tr>
<tr>
<td>Complete bipartite graphs</td>
<td></td>
<td>3rd</td>
<td>1st</td>
<td>2nd</td>
</tr>
<tr>
<td>Avis-Imamura graphs</td>
<td></td>
<td>3rd</td>
<td>1st ex aequo</td>
<td>1st ex aequo</td>
</tr>
<tr>
<td>Avis-Imamura extended graphs (ASL version)</td>
<td>3rd</td>
<td>2nd</td>
<td>1st</td>
<td></td>
</tr>
<tr>
<td>Avis-Imamura extended graphs (ALS version)</td>
<td>2nd</td>
<td>3rd</td>
<td>1st</td>
<td></td>
</tr>
<tr>
<td>Grid graphs</td>
<td></td>
<td>1st</td>
<td>2nd</td>
<td>3rd</td>
</tr>
<tr>
<td>Spider Graphs</td>
<td></td>
<td>1st</td>
<td>2nd</td>
<td>3rd</td>
</tr>
</tbody>
</table>

Table 1: Results of domination in expectation obtained for our algorithms on several graph families

---

3 A spider graph $W_{l\times p}$ is a Cartesian product of a cycle and a path: $W_{l\times p} = C_l \times P_p$. 
algorithms are close (this is the case for paths) and on some graph families, the difference between the expected values of the algorithms can be high (this is the case for Avis-Imamura extended graphs ALS and grid graphs).

In the next subsection, we will present results about experiments realized on random graphs.

### 3.5 Experiments on random graphs

After having analyzed our algorithms on several graph families, we expose a more global study on random graphs. In our study, we use the most popular random graphs class: the Erdős-Rényi graphs [7].

**Definition 4** (Erdős-Rényi graphs). Let $G_{n,p} = (V, E)$ be a random graph, where $n = |V|$ is the number of vertices and $p$ the existence probability for each edge, such that $\forall (u, v) \in V^2, \Pr[uv \in E] = p$.

In Figure 15, graphics show, for each value of $p$, the average value of expectations calculated on several random graphs for our three algorithms.

We can see in the four graphics that values for SORTED-LL (represented by a dashed line) are lower than the 2 others, while the values for ANTI SORTED-LL (represented by a dotted line) are upper. The values for LISTLEFT (represented by a continuous line) are between the 2 others. Thus, the algorithm SORTED-LL has an average value of expectations better than the 2 others on random graphs. But we can see that difference between the 3 curves is thin, which means that values for the 3 algorithms are close.

### 4 Properties of our algorithms on trees

Trees belong to a particular family graph, which is characterized by many properties. The most important is that Vertex Cover problem is polynomial on trees. The algorithm GREEDY INDEPENDENT COVER (GIC), which have been introduced in Subsection 1.2, always returns an optimal solution on trees.

**Algorithm 4** GREEDY INDEPENDENT COVER (adapted on trees)

Require: tree $T = (V, E)$

$C \leftarrow \emptyset$

while $E \neq \emptyset$

Choose a leaf $u \in F(T)$

$C \leftarrow C \cup \{v \mid uv \in E\}$

$T \leftarrow T \setminus \{u, v\}$

return $C$

In this section, we will present firstly some general properties about the Vertex Cover problem on trees. After, we will present some particular properties for each of our 3 algorithms on trees. These properties will be reused to prove that SORTED-LL always return a better solution than ANTI SORTED-LL on labeled paths.
4.1 General properties

In this subsection, we present some properties about the Vertex Cover problem on trees. These properties will be reused in the following subsections, when we will deal with the properties on Sorted-LL and Anti Sorted-LL.

The first general result is that algorithm Greedy Independent Cover can return any optimal solutions on any tree. We call this result “completeness of Greedy Independent Cover”.

**Theorem 6 (Completeness of Greedy Independent Cover).** Let $T$ be any tree. For every optimal cover $C$ of $T$, there exists an execution of Greedy Independent Cover on $T$ returning $C$.

**Proof.** By induction on $n$.

- **Base case.**
For $n = 1$. The tree $T$ reduced to one (isolated) vertex has no leaf. Thus, GREEDY INDEPENDENT COVER returns an empty solution. This is the unique possibility.

For $n = 2$. The tree $T$ reduced to 2 vertices has exactly 2 leaves $u$ and $v$ connected by edge $uv$. There exists 2 optimal solutions: $C_a = \{u\}$ and $C_b = \{v\}$. They can be returned by GREEDY INDEPENDENT COVER. For $C_a$, it chooses in the first step $v$; for $C_b$, it chooses in the first step $u$.

For $n = 3$. The tree $T$ reduced to 3 vertices has exactly 2 leaves $u$ and $v$ and one internal node $i$ adjacent to both $u$ and $v$ ($T$ is a path $P_3$). There exists one optimal cover which contains only the internal node $i$, and this solution is well returned by GREEDY INDEPENDENT COVER.

- General case.

For $n > 3$.

**Induction hypothesis.** Let $T$ be any tree with $n$ vertices and $C$ be any optimal cover of $T$. $C$ can be constructed by GREEDY INDEPENDENT COVER.

**Induction.** Let $T$ be a tree with $n + 1$ vertices and $C$ be any optimal cover of $T$. We will show that there exists an execution of GREEDY INDEPENDENT COVER on $T$ returning $C$.

Let $u$ be a leaf of $T$ and $v$ its unique neighbor. There are 2 possibilities.

**First possibility:** $v \in C$. In this case, $u \not\in C$ (otherwise, $C$ wouldn’t be minimal for inclusion$^4$). We note $T_1, T_2, \ldots, T_k$ the $k$ subtrees of $v$ (excluding $u$). For all $i$ from 1 to $k$:

![Diagram of tree T with vertices u, v, and subtrees T1, T2, ..., Tk](image)

Figure 16: Overview of $T$, from any edge $uv$ (where $v \in C$)

- $C_i = C \cap V(T_i)$ is an optimal cover of $T_i$.
- Otherwise (we suppose the opposite), there exists an optimal cover $C'_i$ of $T_i$ such that $|C'_i| < |C_i|$. In this case, we would have $C' = (C \setminus C_i) \cup C'_i$ a cover of $T$ and $|C'| < |C|$, which contradicts the fact that $C$ is an optimal cover of $T$.

So, we give the GREEDY INDEPENDENT COVER execution to construct $C$:


---

$^4$If we remove a vertex, $C$ is no longer a cover. An optimal cover is always minimal for inclusion.
2. Now, the $T_i$ are isolated trees from each other. $C_i$ are optimal covers and $|T_i| < n$. We can have an execution of \textsc{Greedy Independent Cover} returning $C_1$, then an execution of \textsc{Greedy Independent Cover} returning $C_2$, ..., then an execution of \textsc{Greedy Independent Cover} returning $C_k$ (we apply the induction hypothesis on each subtree $T_i$).

This global execution of \textsc{Greedy Independent Cover} returns $C$ at the end.

Second possibility: $v \notin C$. In this case, all the neighbors of $v$ are in $C$.

\begin{itemize}
\item No $T_i$ is reduced to one vertex $v_i$, otherwise, $C' = C \setminus \{v, u\} \cup \{v\}$ would be a cover of $T$ and $|C'| < |C|$, which contradicts the fact that $C$ is optimal.
\end{itemize}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{overview.png}
\caption{Overview of $T$, from any edge $uv$ (where $u \in C$)}
\end{figure}

For all $i$ from 1 to $k$:

\begin{itemize}
\item $C_i = C \cap V(T_i)$ and $|C_i| \geq 1$, by property a). $C_i$ is an optimal cover of $T_i$.
\item Otherwise (we suppose the opposite), there exists an optimal cover $C'_i$ of $T_i$ such that $|C'_i| < |C_i|$. Construct a new cover of $T$. We substitute the vertices of $C_i$ by the vertices (structurally less numerous) of $C'_i$. We remove $u$ of $C''$ and we put $v$ in the new cover $C'' = (C \setminus C_i \setminus \{u\}) \cup C'_i \cup \{v\}$
\end{itemize}

1. $C''$ is well a cover of $T$ because all the edges of $T_i$ are covered, and also all the edges incident to $v$ (because $v \in C'$) and all the other edges (their cover has not changed).
2. $|C''| < |C|$ because $|C'_i| < |C_i|$ and the change “from $u$ to $v$” don’t modify the number of vertices of $C$.

These properties contradict the fact that $C$ is an optimal cover of $T$.

So, we give an execution of \textsc{Greedy Independent Cover} returning $C$.

\textbf{Step $i$.} We apply \textsc{Greedy Independent Cover} on $T_i$: it returns $C_i$. It’s possible by induction hypothesis, because $C_i$ is an optimal cover of $T_i$ and $|T_i| < n$.

\textbf{Step $k + 1$.} It remains an edge $uv$ on which we apply the last step of \textsc{Greedy Independent Cover}: it returns $u$ (the chosen leaf adjacent to the leaf $v$ on the last edge).
All the steps give an execution of Greedy Independent Cover constructing $C$.

Now, our goal is to show that on trees, it never exists an optimal cover containing all the leaves. The idea is that if Greedy Independent Cover never returns an optimal solution containing all the leaves, then it doesn’t exist such a solution.

**Lemma 3.** For any tree $T$, the algorithm Greedy Independent Cover never returns a cover $C$ such that $F(T) \subseteq C$, i.e. containing all the leaves.

**Proof.** Let $u$ be the first leaf considered by Greedy Independent Cover. The unique neighbor $v$ of $u$ is put in $C$ and $u$ and $v$ are then removed from $T$ and never considered again later. Hence, at least a leaf $u$ (the first one considered during the execution) is never in the cover returned by Greedy Independent Cover.

**Corollary 1** (Derivative from Theorem 6 and Lemma 3). For any tree $T$, it doesn’t exist an optimal cover $C$ such that $F(T) \subseteq C$, i.e. containing all the leaves.

Another result presented in Theorem 7 about leaves on trees for the Vertex Cover problem is that we can always construct a cover which contains no leaf.

**Theorem 7.** For any tree $T$ with $n > 2$ vertices, there exists an optimal cover $C$ such that $C \cap F(T) = \emptyset$, i.e. containing no leaf.

**Proof.** Let $T$ be a non-trivial tree.\(^5\) Let $C$ be an optimal cover of $T$. Suppose that $C$ contains some leaves of $T$. Let $C_F(T) \subset F(T)$ be these leaves. Consider $u \in C_F(T)$. Let $v$ be its neighbor. We know that $C$ is minimal for inclusion because $C$ is optimal. Hence, $v \notin C$ ($u$ only covers the edge $uv$). We can substitute each leaf $u \in C_F(T)$ by its neighbor $v$. This operation doesn’t change the size of $C$.

Figure 18: Construction of an optimal solution without leaf

Figure 18 shows a transformation of an optimal solution to obtain an optimal solution without leaf.

### 4.2 Properties of LISTLEFT

We know that LISTLEFT can always return an optimal solution or the worst solution (with $n - 1$ vertices) for any graph (Theorems 1 and 2). Therefore, it’s also true on trees. So, we focus on Anti Sorted-LL and Sorted-LL.

By reusing the general properties presented in the previous subsection, we will present properties about Anti Sorted-LL and Sorted-LL on trees.

---

\(^5\)A non-trivial tree is a tree with at least 3 vertices ($n > 2$).
4.3 Properties of Anti Sorted-LL

**Lemma 4.** For any non-trivial tree $T$, the algorithm Anti Sorted-LL always returns a cover $C$ such that $F(T) \subseteq C$, i.e. containing all the leaves.

**Proof.** Let $T$ be a non-trivial tree. All the leaves in $T$ have neighbors with an upper degree, because it cannot exist an edge $uv$ where $u \in F(T)$ and $v \in F(T)$ ($T$ is connected). Therefore, Anti Sorted-LL returns all the leaves.

Let $T$ be a tree with $F(T)$ its leaves. Let $f$ be the number of leaves contained on $T$ ($f = |F(T)|$). According to Lemma 4, we have $B\{\overline{SLL}(T)\} \geq f$.

**Theorem 8.** For any non-trivial tree $T$, the algorithm Anti Sorted-LL never returns an optimal cover on $T$.

**Proof.** According to Lemma 4, Anti Sorted-LL always returns all the leaves of trees with size $n > 2$. But, according to Corollary 1, it doesn’t exist an optimal cover on trees which contains all the leaves. Therefore, Anti Sorted-LL never returns an optimal cover on trees.

![Figure 19: A tree for which Anti Sorted-LL never returns $n-1$ vertices](image)

Even if we can easily show that there exists trees $T$ for which $W\{\overline{SLL}(T)\} \neq n-1$ (report to Figure 19 for example), a priori, Anti Sorted-LL is not a good candidate for the Vertex Cover problem on trees.

4.4 Properties of Sorted-LL

**Lemma 5.** For any non-trivial tree $T$, the algorithm Sorted-LL always returns a cover $C$ such that $C \cap F(T) = \emptyset$, i.e. containing no leaf.

**Proof.** Let $T$ be a non-trivial tree. All the leaves in $T$ have exactly one neighbor with an upper degree (it doesn’t exist an edge $uv$ where $u \in F(T)$ and $v \in F(T)$). Hence, no leaf is selected by Sorted-LL.

According to Lemma 5, we have $W\{\overline{SLL}(T)\} \leq n-f$.

![Figure 20: A tree for which Sorted-LL never returns an optimal solution](image)
According to Theorem 7, for any non-trivial tree $T$, it always exists an optimal cover $C$ on $T$ which contains no leaf. But, there exists trees on which SORTED-LL never returns an optimal solution (see Figure 20).

### 4.5 SORTED-LL and ANTI SORTED-LL on paths

In this subsection, we prove that SORTED-LL is always better than ANTI SORTED-LL on non-trivial labeled paths (a path is a tree with exactly 2 leaves). We need to introduce 3 classes (or subsets) of vertices in a labeled tree, related to SORTED-LL and ANTI SORTED-LL. In order to simplify the definition of these classes, we suppose throughout this subsection (and more particularly in the proof of Theorem 9) that vertices are labeled by following a decreasing order of degrees; but, of course, results presented are also valid for unsorted labeled paths.

**Definition 5** (Classes of vertices in a SLT). Let $L_s(T)$ be a labeled tree where vertices are labeled by following a decreasing order of degrees. We call SLT (Sorted Labeled Tree) such a labeled tree. So, in $L_s(T)$, we have:

- $M_{L_s(T)}$ represents vertices returned by SORTED-LL and by ANTI SORTED-LL. More precisely, these vertices have right neighbors and left neighbors. We denote by $M_T$ its size.

- $R_{L_s(T)}$ represents vertices returned by SORTED-LL but not returned by ANTI SORTED-LL. More precisely, these vertices have right neighbors but have not left neighbors. We denote by $R_T$ its size.

- $L_{L_s(T)}$ represents vertices returned by ANTI SORTED-LL but not returned by SORTED-LL. More precisely, these vertices have left neighbors but have not right neighbors. We denote by $L_T$ its size.

We can divide $L_{L_s(T)}$ in 2 subclasses: (1) $F(T)$, because $F(T) \subseteq L_{L_s(T)}$; and (2) $L'_{L_s(T)}$, the other vertices ($L'_L \cap F(T) = \emptyset$).

Figure 21 shows an example of decomposition in vertex classes of a SLT.

![Figure 21: Example of a SLT where vertices with the different classes of vertices](image)

**Remark 4.** We could define a fourth class of vertices, $I_{L_s(T)}$. It would correspond to the vertices which are returned by none of the 2 algorithms. In fact, $I_{L_s(T)}$ represents the isolated vertices. Hence, for any labeled tree, $I_{L_s(T)} = \emptyset$. 
Now, we can show that Sorted-LL is better than Anti Sorted-LL on non-trivial paths.

**Theorem 9.** For all non-trivial sorted labeled path \( L_s(P_n) \), we have \( L_{P_n} > R_{P_n} \).

**Proof.** Let \( \Sigma = \{L, M, R\} \) be an alphabet (i.e. a set of symbols), where:

- \( L \) corresponds to the vertices of \( L_{L_s}(P_n) \),
- \( M \) corresponds to the vertices of \( M_{L_s}(P_n) \),
- \( R \) corresponds to the vertices of \( R_{L_s}(P_n) \).

Let \( \mathcal{E}_P \) be the regular expression defined on \( \Sigma \) for a labeled path \( L(P_n) \), by simulating its route from left to right. For any labeled path, we obtain

\[
\mathcal{E}_P = L(M^* R M^* L)^+
\]

We denote by \( x_{\mathcal{E}_P} \) the apparition number of the symbol \( x \in \Sigma \) in the expression \( \mathcal{E}_P \). In the parenthesis of \( \mathcal{E}_P \), there is the same number of \( L \) and \( R \). Therefore, \( L_{\mathcal{E}_P} = R_{\mathcal{E}_P} + 1 \). The regular expression \( \mathcal{E}_P \) well simulate the route of a labeled path from left to right. We can easily show it, by listing the different properties of \( \mathcal{E}_P \).

1. We are sure that expression starts with \( L \) and ends with \( L \), because there are exactly 2 leaves in a labeled path.
2. We are sure that existence of sequences \( LL \) and \( RR \) is impossible in our route, because in \( uv \), \( u \) is left neighbor of \( v \) and \( v \) is right neighbor of \( u \).
3. We are sure that existence of sequences \( RM^* R \) or \( LM^* L \) is impossible, else it would mean that \( \exists u \in M_{L_s}(P_n) \) such that \( d(u) > 2 \), that is not possible in paths.

Consequently, for any sorted labeled path \( L_s(P_n) \), we have \( L_{P_n} = R_{P_n} + 1 \). \( \square \)

So, we have proved that Sorted-LL always returns a better solution than Anti Sorted-LL on any sorted labeled path. As we have said at the beginning of this subsection, this result is also verified on unsorted labeled paths. Indeed, as an illustration for \( R \) (the principle is the same for \( M \) and \( L \)), in any labeled tree \( L(T) \), according to Definition 5, \( R_{L(T)} \) represents vertices only returned by Sorted-LL (and not by Anti Sorted-LL): they have neighbors with lower degree or right neighbors with the same degree. In fact, in the proof of Theorem 9, we have just supposed that neighbors with lower degree have labels larger, that is equivalent (we have showed in Section 2 an equivalence between sorted vertex lists and labeled graphs).
5 An equivalent edge-by-edge algorithm for SORTED-LL

In the previous sections, we have studied vertex-by-vertex algorithms. But several applications manipulate graphs edge by edge. This is the case in streaming graph applications [8, 12]. So, we will show that our algorithms can be easily adapted to equivalent edge-by-edge algorithms, by giving an edge-by-edge algorithm similar to SORTED-LL.

The principle of our algorithm is very simple: for each edge, we choose one of the 2 endpoints. If we choose the endpoints randomly, for any graph such that $m \geq n$, we can return a solution which contains $n$ vertices. This is the case in the example of Figure 23 (vertices of solution are represented by squares).

![Figure 23: Execution of an edge-by-edge random algorithm on a cycle $C_5$](image)

Our edge-by-edge algorithm for the Vertex Cover problem is called EDGE DEGREE IDENTIFIER (or EDI for the short name). As for LISTLEFT, SORTED-LL and ANTI SORTED-LL, it takes a labeled graph in input. We can see in Figure 24 an execution of Algorithm 5 on a 4 vertices labeled graph, where the cover returned contains 2 vertices (2 and 4).

![Figure 24: Execution of EDGE DEGREE IDENTIFIER on a 4 vertices labeled graph](image)

**Algorithm 5 EDGE DEGREE IDENTIFIER (EDI)**

```plaintext
Require: labeled graph $L(G) = (L(V), E)$

$C \leftarrow \emptyset$

for each edge $e = uv \in E$ do
    if $d(u) > d(v) \lor (d(u) = d(v) \land L_u < L_v)$ then
        $C \leftarrow C \cup \{u\}$
    else
        $C \leftarrow C \cup \{v\}$

return $C$
```


Lemma 6 (Equivalence between Edge Degree Identifier and Sorted-LL). For any labeled graph $L(G)$, we have

$$C_{\text{EDI}}(L(G)) = C_{\text{SLLL}}(L(G))$$

Proof. Let $L(G) = (L(V), E)$ be any labeled graph. If we execute Edge Degree Identifier on $L(G)$, for each vertex $u \in L(V)$, we obtain:

- $u \in C_{\text{EDI}}(L(G))$, if $\exists v \in N(u)$ such that $d(v) < d(u)$ or $\exists v \in N(u)$ such that $d(v) = d(u)$ and $L_v > L_u$;
- $u \not\in C_{\text{EDI}}$, if $\forall v \in N(u)$, $d(v) \geq d(u)$ and if $\exists v \in N(u)$ such that $d(v) = d(u)$ and $L_v > L_u$.

These properties are similar to the Sorted-LL properties, because

- “$u$ has at least one right neighbor” means there exists $v \in N(u)$ such that $d(v) < d(u)$ or there exists $v \in N(u)$, such that $d(v) = d(u)$ and such that $L_v > L_u$;
- “$u$ has no right neighbor” means that for all $v \in N(u)$, $d(v) \geq d(u)$ and it doesn’t exist $v \in N(u)$ such that $d(v) = d(u)$ and such that $L_v > L_u$.

Thus, for any labeled graph $L(G)$, we have $C_{\text{EDI}}(L(G)) = C_{\text{SLLL}}(L(G))$. $\square$

So, we can say that algorithm Edge Degree Identifier has the same properties as Sorted-LL. Hence, it has the same approximation ratio, $\sqrt{\frac{\Delta}{2}} + \frac{3}{2}$.

From the algorithm Edge Degree Identifier and the proof of Lemma 6, by following the same principle, we could define equivalent edge-by-edge algorithms for ListLeft and Anti Sorted-LL.

6 Conclusion

We have presented, described and analyzed three algorithms adapted from list algorithms for the Vertex Cover problem: ListLeft, Sorted-LL and Anti Sorted-LL. These 3 algorithms are suitable to our model: they don’t need to modify the input graph, they don’t need to read the solution during the execution and they don’t need to use memory of the processing unit.

In Section 3, we have given exact expectation formulas for our algorithms. Results of our analysis in expectation on several graph families have showed that no algorithm can be elected as the best one. Indeed, we have seen on several graph families that each algorithm can be the best in expectation and the difference between the expected values of the 3 algorithms can be large.

In Section 4, we have proved that on labeled paths, Sorted-LL always returns a better solution than Anti Sorted-LL. In Section 5, we have showed that our algorithms can be easily transformed into edge-by-edge versions.

An ongoing work is to prove that on labeled trees, Sorted-LL always returns a better solution than Anti Sorted-LL. Another ongoing work is to analyze the number of requests made by our algorithms during their executions, in order to evaluate more precisely their performances.
References


Appendix

A A different proof for Theorem 9

In this section, we give a more classical and exhaustive proof for Theorem 9.

Proof. By induction on $n$.

• Base case.

For $n = 3$. Let $L_s(P_3)$ be a sorted labeled path constituted by 2 leaves $f_1$ and $f_2$ and by one vertex $u$ of degree 2, which is neighbor of $f_1$ and $f_2$. We have necessarily $u \in R_{L_s(P_3)}$ and $L_{L_s(P_3)} = F(P_3)$.

Hence, $L_{P_3} = R_{P_3} + 1$.

For $n = 4$. Let $L_s(P_4)$ be a sorted labeled path constituted by 2 leaves $f_1$ and $f_2$ and by 2 vertices $u$ and $v$ of degree 2. $u$ and $v$ are connected. Suppose that $f_1$ is connected with $u$ and $f_2$ is connected with $v$. We have necessarily $L_{L_s(P_4)} = F(P_4)$, $u \in R_{L_s(P_4)}$ and $v \in M_{L_s(P_4)}$ or otherwise. Hence, $L_{P_4} = R_{P_4} + 1$.

• General case.

For $n > 4$.

Induction hypothesis. For all non-trivial sorted labeled path $L_s(P_n)$, we have $L_{P_n} = R_{P_n} + 1$.

Induction. For all non-trivial sorted labeled path $L_s(P_{n+1})$, we will show that $L_{P_{n+1}} = R_{P_{n+1}} + 1$.

We can always represent $L_s(P_{n+1})$ as follows

\[
\begin{array}{c}
\mathcal{R} \\
\mathcal{M} \\
\mathcal{L} \\
\mathcal{F}
\end{array}
\]

\(L_s(P_{n+1})\)

i.e. having first the vertices of $\mathcal{R}_{L_s(P_{n+1})}$, then those of $\mathcal{M}_{L_s(P_{n+1})}$, then those of $\mathcal{L}_{L_s(P_{n+1})}$ and finally the 2 leaves.

Let $u$ be the unique neighbor of the last leaf $f$ in $L_s(P_{n+1})$. We know that $u \notin \mathcal{L}_{L_s(P_{n+1})}$.

First possibility: $u \in \mathcal{M}_{L_s(P_{n+1})}$. Let $v$ the other neighbor of $u$.

$v$ can be in $\mathcal{R}_{L_s(P_{n+1})}$.

\[
\begin{array}{c}
\mathcal{R} \\
\mathcal{M} \\
\mathcal{L} \\
\mathcal{F}
\end{array}
\]

Or

$v$ can be in $\mathcal{M}_{L_s(P_{n+1})}$.
Let $L_s(P_n) = L_s(P_{n+1}) \setminus \{f\}$ be the labeled path where $u$ becomes a leaf and takes the position of $f$. For all the cases, we can see that $L_s(P_n)$ is well a sorted labeled path. Indeed, if $v$ was previously in $R_{L_s(P_{n+1})}$, then $v$ is in $M_{L_s(P_n)}$; if $v$ was previously in $M_{L_s(P_{n+1})}$, then $v$ is in $R_{L_s(P_n)}$. Therefore, we can apply the induction hypothesis on $L_s(P_n)$ in order to compute the sizes of the different vertices subsets. As $R_{P_{n+1}} = R_{P_n}$ and $L_{P_{n+1}} = L_{P_n}$, then we have $L_\square_{P_{n+1}} = R_\square_{P_{n+1}} + 1$.

Second possibility: $u \in R_{L_s(P_{n+1})}$. Let $v$ be the other neighbor of $u$.

1. $v$ can be in $M_{L_s(P_{n+1})}$.

2. $v$ can be in $L'_s(L_s(P_{n+1}))$. Let $w$ be the other neighbor of $v$. Let $L_s(P_n) = L_s(P_{n+1}) \setminus \{f\}$ be the labeled path where $u$ becomes a leaf and takes the position of $f$. Whatever the position of $w$ in $L_s(P_{n+1})$, we can see that $L_s(P_n)$ is well a sorted labeled path, because $v$ moves to $R_{L_s(P_n)}$ and $w$ rests in the same class. So, $R_{P_{n+1}} = R_{P_n}$ and $L_{P_{n+1}} = L_{P_n}$, thus, we have $L_\square_{P_{n+1}} = R_\square_{P_{n+1}} + 1$.

If $w \in M_{L_s(P_{n+1})}$, then we have 2 possibilities:

- $v$ moves to $M_{L_s(P_n)}$ and thus, $w$ rests in $M_{L_s(P_n)}$. Then, we have $R_{P_{n+1}} = R_{P_n} + 1$ and $L_{P_{n+1}} = L_{P_n} + 1$. Hence, $L_{P_{n+1}} = R_{P_{n+1}} + 1$.

- $v$ moves to $M_{L_s(P_n)}$ and thus, $w$ moves to $L'_s(L_s(P_n))$. Then, we have $R_{P_{n+1}} = R_{P_n}$ and $L_{P_{n+1}} = L_{P_n}$. Hence, $L_{P_{n+1}} = R_{P_{n+1}} + 1$.
If \( w \in R_{L_s(P_n+1)} \), then \( v \) moves to \( R_{L_s(P_n)} \) and \( w \) rests in \( R_{L_s(P_n)} \). Then, we have \( R_{P_n+1} = R_{P_n} + 1 \) and \( L_{P_n+1} = L_{P_n} + 1 \). Hence, \( L_{P_n+1} = R_{P_n+1} + 1 \).

Therefore, for any sorted labeled path \( L_s(P_n) \), we have \( L_{L_s(P_n)} = R_{L_s(P_n)} + 1 \).

### B A so-called better list algorithm: LISTRIGHT

It’s an other list algorithm which has been described and analyzed by F. Delbot et al. in their articles *A better list heuristic for vertex cover* [5] and *Mean analysis of an online algorithm for the vertex cover problem* [3]. LISTRIGHT is based on solution proposed by M. Demange and V. Paschos in *On-line vertex covering* [6]. It has some interesting properties. We resume it:

**Property 1** It always returns a cover minimal for inclusion.

**Property 2** If we execute LISTLEFT and LISTRIGHT on the same vertex list (assuming that LISTLEFT scans the vertices from left to right and LISTRIGHT from right to left), LISTRIGHT always returns a better solution than LISTLEFT.

**Property 3** For every graph \( G \), the expected size of \( LR(G) \) is bounded by \( (2 - \frac{2}{n}) \cdot OPT(G) \), and this bound is tight.

Properties 1 and 2 are given in [5] (with respectively Theorems 1 and 2) and Property 3 is given in [3] (with Theorem 1).

**Approximation ratio.** M. Demange and V. Paschos have showed that for any graph, the approximation ratio of any cover minimal for inclusion is bounded above by \( \Delta \) (see Lemma 1 and Corollary 1 in [6]). Therefore, LISTRIGHT has an approximation ratio at most \( \Delta \), and this ratio is tight on stars.

**LISTRIGHT and our model.** As it has been described in [5], LISTRIGHT has the following behavior: for each vertex \( u \), if \( u \) has a right neighbor not in the cover, then \( u \) is added to it. With our model, this condition means that during the execution, we have to store the vertices of solution into the memory of the processing unit. The size of the cover can be potentially large. Thus, LISTRIGHT is a list algorithm but unlike LISTLEFT, SORTED-LL and ANTI SORTED-LL, it requires memory space during its execution.

In the article of J. Feigenbaum et al. [8], a fundamental result have been highlighted: testing many properties in a graph with \( n \) vertices requires \( \Omega(n) \) bits of space (see Lemma 1 in [8]). So, according to this important fact, we could relax the memory constraint and then adapt LISTRIGHT to our model. But another difficulty appears. Consider any labeled graphs \( L(G) = (L(V), E) \). Contrary to our three algorithms, the order apparition of vertices is important for LISTRIGHT, even if vertices are labeled. Indeed, for 2 different processing order of vertices, we can have 2 different solutions, that is not the case for our algorithms. For example, on the labeled graph of Figure 25, if we process vertices in the following order: 5, 2, 4, 1, 3; we obtain the following solution: \{2, 4, 1\}; while if we process vertices in the following order: 3, 2, 5, 4, 1; we obtain the following solution: \{3, 4, 1\}. 
Thus, if we want to adapt LISTRIGHT to our model, we have to suppose that vertices arrive in a predetermined order and that the processing unit has a memory of size $n$. These 2 assumptions are not compatible with our model: a memory of size $n$ doesn’t respect the memory constraint; also, it is important in a system where a large graph is stored on a large data warehouse that vertices can arrive and be scanned in any order. Hence, the algorithm LISTRIGHT cannot be adapted to our model.

Figure 25: A regular labeled graph with 5 vertices
Algorithms for the Vertex Cover Problem on Large Graphs

Eric Angel, Romain Campigotto and Christian Laforest

Abstract

In a classical treatment of an optimization graph problem, the whole graph is available in the machine. It can be modified, updated (vertices can be marked, edges deleted, etc.) and the solution can be stored. However, many applications produce now data that are too large and are impossible to store and treat in this model.

We focus in this paper on the treatment of the classical NP-complete minimization Vertex Cover problem in large graphs. The difficulty is twofold: to the intrinsic NP-completeness is added the difficulty to manipulate the input graph with severe restrictions: (1) the input graph must not be modified (integrity of the input instance), (2) the graph must be “scanned” piece by piece since it is too large to be entirely loaded in the main memory and (3) the result must be sent to an output memory once a new piece of solution is calculated.

Hence, we suppose that the size and the source of the input graph impose severe treatment constraints. After modeling this situation, we show that most of the known approximation algorithms are not compatible with our model. In a second part, we propose three algorithms (that are variations on the same “root” list algorithms) that satisfy our severe constraints. In the following, we compare them by giving exact formulas expressing the expected size of the Vertex Cover constructed by the algorithms on specific designed graphs.

General Terms: Algorithms, Measurement, Performance, Theory

Additional Key Words and Phrases: large graphs, Vertex Cover, mean analysis, approximation