Implementation and Comparison of Heuristics for the Vertex Cover Problem on Huge Graphs

Eric Angel, Romain Campigotto, Christian Laforest

Implementation and Comparison of Heuristics for the Vertex Cover Problem on Huge Graphs*

Eric Angel[1], Romain Campigotto[2] and Christian Laforest[3]

[1] Laboratoire IBISC, EA 4526 – Université d'Évry-Val d'Essonne
IBGBI, 23 boulevard de France, 91 037 Évry Cedex, France
eric.angel@ibisc.univ-evry.fr
[2] LAMSADE, CNRS UMR 7243 – Université Paris-Dauphine
Place du Maréchal de Lattre de Tassigny, 75 775 Paris Cedex 16, France
romain.campigotto@lamsade.dauphine.fr
[3] LIMOS, CNRS UMR 6158 – Université Blaise Pascal, Clermont-Ferrand
Campus des Cézeaux, 24 avenue des Landais, 63 173 Aubière Cedex, France
christian.laforest@isima.fr

Abstract. We present in this paper an experimental study of six heuristics for a well-studied **NP**-complete graph problem: the VERTEX COVER. These algorithms are adapted to process huge graphs. Indeed, executed on a current laptop computer, they offer *reasonable* CPU running times (between twenty seconds and eight hours) on graphs for which sizes are between $200 \cdot 10^6$ and $100 \cdot 10^9$ vertices and edges.
We have run algorithms on specific graph families (we propose generators) and also on random power law graphs. Some of these heuristics can produce good solutions. We give here a comparison and an analysis of results obtained on several instances, in terms of quality of solutions and complexity, including running times.

Key words: *implementation of algorithms, experimental analysis, huge graphs, low memory, vertex cover*

1 Introduction

The VERTEX COVER problem [14] is a well-known classical **NP**-complete optimization graph problem that has received a particular attention these last decades. In particular, it occurs in many concrete situations [17], in fields such as biology, meteorology, finance, etc. where amount of data is more and more important. This leads to the problem of designing algorithms well suited to cope with such large instances.

Notations. Graphs $G = (V, E)$ considered throughout this paper are undirected, simple, unweighted and represent the *instance* to be treated here. 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 $N(u) = \{v \mid uv \in E\}$ the set of *neighbors* of u and we call *degree* the number of neighbors of vertex u.

* Work partially supported by the French Agency for Research under the DEFIS program *TODO*, ANR-09-EMER-010.

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

Related Work. Several studies focused on massive data sets these last decades [3]. In particular, for the MAX CLIQUE problem [9], experiments on graphs with $53 \cdot 10^6$ vertices and $170 \cdot 10^6$ edges have been performed [2]. But no such study has been done for the VERTEX COVER problem. However, it has been extensively studied theoretically: many exact (exponential) algorithms, approximation algorithms, online algorithms, etc. have been proposed (due to space limitations, we do not give references about these works: some of them can be found in the introductions of [8] and [13]). Several experimental studies have already been made, often to compare the quality of several algorithms [12, 15] or validate specific methods [5, 7]. Nevertheless, no one achieved the huge graph sizes we consider in this paper (in these studies, the largest graphs has 10,000 vertices).

Our General Model of Treatment. To the intrinsic **NP**-completeness is added the difficulty to manipulate graphs and run algorithms with severe constraints. Indeed, with respect to their huge sizes, the processing unit (we consider a standard computer) cannot load them entirely in its memory. Moreover, the graph, which is stored on an external disk, must not be modified, since it often comes from experimentations and can be used by different users for different goals. Specifically, the important cost of graph creation forces us to preserve its *integrity*, in order to be able to run several algorithms on it.

Organization of the Paper. We give in the next section a general description of our experiments. We present, analyze and compare in Sect. 3 results obtained by executing the six algorithms on several instances. Finally, in Sect. 4, we conclude and give some perspectives.

2 General Description

In this section, we describe elements used for experiments. Programs (executables, with source code) are available at [1].

Technical Characteristics. The "Processing Unit" is a laptop computer with a Dual Core processor running at 2.8 GHz, 6 Mb cache memory and 4 Gb RAM. Graphs and Covers are stored on the same external hard drive, which is a USB 2.0 hard disk of 2 Tb, running at 7200 revolutions/minute and equipped with 8 Mb cache memory. Programs are written in language C, C99 standard, in order to use specific data type `unsigned long long` and associated functions to read and write binary files.

Storage and Reading of Graphs. There exist many ways to store a graph: with an adjacency matrix, an adjacency list, etc. We use the method described in [4] (for more details, report to the Sect. 3.2 page 20). More precisely, our graphs are stored with two files:

`.list` **file** which contains $2m + 1$ values: the number of vertices in the graph (which is needed by several algorithms to create an n bits array) and the list of the neighbors of vertices in the graph;

`.deg` **file** which contains $n+1$ values, which are needed to access to the neighbors of a vertex and compute its degree.

The n vertices are labeled from 0 to $n-1$. The `.list` file contains first the value n, then vertices of set $N(0)$, then vertices of set $N(1)$, etc. (however, neighbors of each vertex can be stored in any order, not necessarily following the order of labels). The `.deg` file contains, for each vertex (and by increasing order of labels), the place of its first neighbor (in the `.list` file). It contains $n+1$ values, in order to compute the degree of the last vertex (the last value of the `.deg` file points to the end of the `.list` file). Indeed, to compute degree of vertex i, we subtract the i^{th} value from the $(i+1)^{\text{th}}$.

Figure 1 shows an example on a small graph.

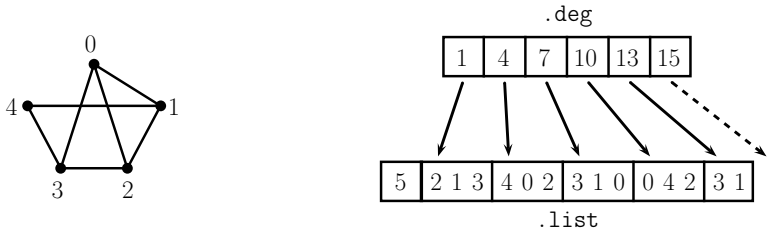

Fig. 1. Storage of a graph with 5 vertices and 7 edges

Algorithms scan graphs by reading the two files described previously. First, the `.list` file is read to know the number of vertices. Then, the `.deg` file is read to know the place of the first neighbor of the first vertex and the place of the first neighbor of the second vertex. Thereby, there are two pointers (the first one is followed by the second one) which delimit the set of neighbors of a vertex. Once these places are known, the reading process continues in the `.list` file, where neighbors of the first vertex are retrieved. If the treatment unit does not need to get all of them, it can "step over" the remaining neighbors and go immediatly to the neighbors of the second vertex. It proceeds in the same way for the following vertices.

If algorithms need to know degrees of neighbors, they read independently the `.deg` file with another playhead, which can be moved at a precise place to get the two successive values needed to compute the degree.

All the algorithms read the `.list` file in a sequential way, but they can "step over" values that are not needed. Algorithms which do not need to compute degrees of neighbors read the `.deg` file also in a sequential way.

Algorithms Implemented. We have implemented six algorithms adapted to the treatment of huge graphs: LR, ED, S-Pitt, LL, SLL and ASLL.

LR has been proposed in [11]. ED is the 2-approximative algorithm which returns vertices of a maximal matching. S-Pitt is a probabilistic algorithm inspired by the algorithm presented in [18]: it has an expected approximation ratio equal to 2. The authors have done a theoretical study of LL, SLL and ASLL in [6].

We present now a basic description of the six algorithms, by giving conditions to put vertices of the input graph into the solution.

Let $G = (V, E)$ be a graph. Let C be the cover under construction. For each vertex $u \in V$, we have

LR: if $u \notin C$, $\{v \mid uv \in E \wedge v \notin C\}$ is put in C;

ED: if $u \notin C$ and if u has a neighbor $v \notin C$, u and v are put in C;

S-Pitt: if $u \notin C$ and if u has a neighbor $v \notin C$, either u or v is put in C with equiprobability assumption;

LL: u is put in C if it has at least one neighbor v such that $v > u$ (their labels are compared);

SLL: u is put in C if it has at least one neighbor v such that $d(v) < d(u)$ or $d(v) = d(u)$ and $v > u$;

ASLL: u is put in C if it has at least one neighbor v such that $d(v) > d(u)$ or $d(v) = d(u)$ and $v < u$.

Now, we can describe how we have implemented these algorithms, in relation with the way that the graphs are stored on the external hard disk.

As described above, the algorithms scan graphs vertex by vertex and, for each current vertex u, scan its neighbors one by one. If an algorithm decides that u belongs to the solution (applying the conditions given in the descriptions of the algorithms above), u is put immediately and definitively into the cover. Then, the algorithm steps over its remaining neighbors and goes to the next vertex. Otherwise, it gets the next neighbor of u (and, at the end, requires the next vertex like in the previous case). Also, when an algorithm scans a vertex u which is already in the cover, it goes immediately to the next vertex, without scanning its neighbors.

It is worth noticing that LR, ED and S-Pitt need to allocate an n bits array to mark vertices sent to the solution (reading on the external hard drive during the execution would take too long); SLL and ASLL need to compute degrees of neighbors.

Writing the Covers on the Disk. A cover is written as a list of vertex labels into a file, which is built piece by piece: once an algorithm decides to put a vertex into the solution, it writes it into the cover file. A vertex cannot appear twice, because algorithms have been designed to produce no duplicates.

Example of Execution of Algorithm LR. We consider the graph given in Fig. 2. The execution of LR on it works as follows.

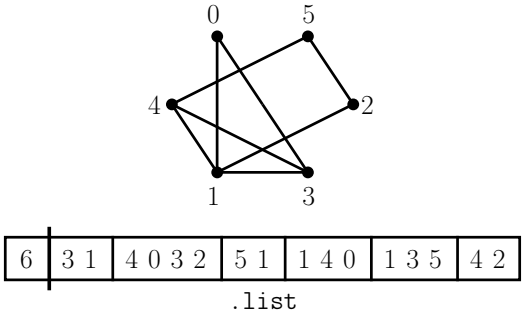

Fig. 2. Graph with 6 vertices and 8 edges

At the beginning, the cover C (which is materialized by an 6 bits array in the internal memory of the computer) is empty (i.e. all the binary flags are lowered).

1. $C = \emptyset$. We consider the vertex 0. We get its neighbor, 3: we put it in C (it is written on the disk and the corresponding flag in the memory is raised). Then, we get its neighbor, 1: we put it in C.
2. $C = \{1, 3\}$. The vertex 1 is not treated since it is already in C.
3. $C = \{1, 3\}$. We treat the vertex 2. We get its neighbor, 5: it is put in C. Then, we get its neighbor, 1: nothing is done since it is already in C.
4. $C = \{1, 3, 5\}$. The vertex 3 is not treated since it is already in C.
5. $C = \{1, 3, 5\}$. We treat the vertex 4. Its neighbors (1, 3 and 5) are retrieved but not considered, because they are already in C.
6. $C = \{1, 3, 5\}$. The vertex 5 is not treated since it is already in C.

Hence, we have just scanned seven vertices in the `.list` file (which contains sixteen labels) and the cover produced by LR contains three vertices: 1, 3 and 5.

Graph Families Used. We have executed our algorithms on different graphs:

- on sparse graphs (where $m \in \mathcal{O}(n)$): *ButterFly* graphs [21], *de Bruijn* graphs [10] and grid graphs;
- on dense graphs (where $m \in \Theta(n^2)$): hypercubes, complete bipartite graphs and *complete split graphs*[4].

[4] A *complete split graph* is a complete bipartite graph in which the vertices subset of lowest size is changed to a *clique*.

We have chosen these graphs because they can be easily generated (we can produce huge size graphs with a standard computer). In that sense, we have designed generators to construct these specific graphs. For example, CPU running times for the instances generation of size $100 \cdot 10^9$ are between 5 hours than 7 hours. Also, the size of their optimal solutions is known (we can give exact sizes, excepted for *de Bruijn* graphs where we can only give lower bounds). Thus, it is possible to present results on the quality of algorithms.

We have also chosen to execute algorithms on *random power law graphs*, where degree sequences follow a power law. We have used generator described in [19], which is based on the *Molloy and Reed* model [16].

This generator is able to produce random power law graphs, but it cannot create them in an online way: a memory space linear to the graph size is needed. However, on our computer, we can still create graphs with more than $10 \cdot 10^6$ vertices and edges, with CPU running times less than 3 hours.

Graph Sizes. The size of a graph is given by its number of vertices and edges. Hence, we denote by *graph size* the value $n + m$.

The *huge size* notion is relative: it depends on context considered (e.g. the size limits for algorithms with exponential complexity are lower than for algorithms with linear complexity). So, we have defined several levels for graph sizes.

1st level. The graphs size is about $200 \cdot 10^6$ (several Gb on disk). This is the largest random power law graphs size that can be generated on our computer.

2nd level. The graphs size is about $30 \cdot 10^9$ (more than 100 Gb on disk). The algorithms SLL and ASLL begin to reach their limits in terms of running times on our computer.

3rd level. The graphs size is about $100 \cdot 10^9$ (around 1.5 Tb on disk). With our computer, we cannot allocate an n bits array if the number of vertices is bigger than $30 \cdot 10^9$.

Experimentations. For the first level, we have executed algorithms S-Pitt, LL, SLL and ASLL five times on each instance (algorithms LR and ED are deterministics). From second level, we simulated a user having limited resources (time and disk space). So, for each graph, we have executed the six algorithms once. Based on results obtained and resources already spent, we have executed again several algorithms (often one time).

A total of thirty-four executions was made: thirteen in the first level, six in the second level and two in the third level. We have also executed our algorithms on thirteen instances for which sizes are between 10^6 and $4 \cdot 10^6$. But, due to space limitations, we do not give details on results obtained for these instances. We can however indicate that they are broadly similar to the results obtained in the first level.

3 Results and Observations

Evaluated Criteria. We have focused on quality of solutions produced by algorithms and complexities, expressed by the *number of requests* made to the instance, i.e. the number of neighbors read in the `.list` file. We have also considered running times. For that, we have used the UNIX command `/usr/bin/time`, which gives the time used by the processor during a program execution.

Results Presentation. We give one table per criterion. For algorithms that have been executed more than once, we give values corresponding to the best solution (in terms of quality).

For each instance, the best value (among the set of values presented for the six algorithms) is in bold font. Conversely, when values are bad (e.g. an algorithm which returns almost all the vertices or performs more than m requests), numbers are in italics font.

Table 1 (resp. 2) gives quality of solutions (resp. complexity in number of requests) obtained in the first level on graphs created by our generators and on random graphs. For random power law graphs, the last digit given in the instance name (starting by `rg`) denotes the minimum degree of the graph.

Table 1. Quality of solutions obtained in the first level, expressed in percentage of n (sizes of optimal covers for random power law graphs cannot be estimated)

Instance	n	OPT	LR	ED	S-Pitt	LL	SLL	ASLL
butterfly-21	46,137,344	50	**50**	*100*	78.16	90.91	85.06	90.91
debruijn-25	33,554,432	> 50	**66.67**	88.89	77.56	66.78	72.71	82.39
grid-6000.9000	54,000,000	50	**50**	*99.99*	81.41	*99.99*	*99.99*	*99.99*
hypercube-23	8,388,608	50	**50**	*99.97*	99.26	*99.99*	*99.99*	*99.99*
compbip-7000.15000	22,000	31.82	68.18	63.64	62.98	**31.82**	**31.82**	68.18
split-7500.12000	19,500	38.46	*99.99*	61.67	61.39	**38.46**	**38.46**	*99.99*
rg-20m_1	20,000,000	–	**9.94**	19.65	13.41	49.30	10.42	*99.99*
rg-20m_2	20,000,000	–	**34.12**	62.89	45.60	49.30	36.89	*99.88*
rg-25m_1	25,000,000	–	**14.19**	28.09	18.91	41	14.89	*99.95*
rg-25m_2	25,000,000	–	**38.76**	69.99	51.29	48.13	42.12	*99.65*
rg-30m_1	30,000,000	–	**43.48**	76.61	57.17	59.02	47.44	*97.57*
rg-30m_2	30,000,000	–	**15.68**	31.05	21.10	30.98	16.46	*99.93*
rg-35m_2	35,000,000	–	**43.12**	76.16	56.79	53.70	46.98	*99.12*

For the first level, expected running times of each algorithm are between twenty seconds and two minutes.

Tables 3 and 4 give respectively quality of solutions and number of requests obtained for the second and third levels. Table 5 gives CPU running times. For the third level, we only give values for algorithms that have been executed until the end.

Table 2. Number of requests performed in the first level (in percentage of m)

Instance	m	LR	ED	S-Pitt	LL	SLL	ASLL
butterfly-21	88,080,384	*100*	**60.51**	91.94	*103.30*	*104.76*	80.16
debruijn-25	67,108,861	66.67	**61.11**	87.03	*113.87*	*106.84*	*100.48*
grid-6000.9000	107,985,000	*100*	**49.93**	86.24	91.66	91.66	75.03
hypercube-23	96,468,992	*100*	**11.83**	23.57	17.54	17.58	17.51
compbip-7000.15000	105,000,000	*100*	**53.34**	54.31	*100.01*	*100.01*	*100.02*
split-7500.12000	118,121,250	**0.02**	47.47	47.82	76.20	76.20	0.17
rg-20m_1	59,624,494	49.12	47.99	55.20	**34.29**	59.22	57.73
rg-20m_2	90,808,193	40.23	36.92	48.73	38.20	56.57	**34.27**
rg-25m_1	70,911,180	45.44	44.57	53.13	**36.23**	58.19	47.08
rg-25m_2	87,837,432	45	**40.93**	55.45	51.81	65.50	41.84
rg-30m_1	82,356,722	50.09	**45.04**	62.78	57.89	75.78	52.29
rg-30m_2	81,819,916	44.86	44.02	52.84	**39.76**	58.28	45.23
rg-35m_2	96,555,269	50.14	**45.11**	62.70	62.52	75.62	53.31

Table 3. Quality of solutions obtained in the second and third levels, expressed in percentage of n

Instance	n	OPT	LR	ED	S-Pitt	LL	SLL	ASLL
butterfly-28	7,784,628,224	48.28	**48.28**	96.55	78.76	93.24	93.10	96.55
debruijn-33	8,589,934,592	> 50	**66.67**	88.89	77.56	96.55	*99.99*	*99.99*
grid-75000.90000	6,750,000,000	50	**50**	*99.99*	81.41	*99.99*	*99.99*	*99.99*
hypercube-30	1,073,741,824	50	**50**	*99.99*	99.78	*99.99*	*99.99*	*99.99*
compbip-35000.500000	535,000	6.54	93.46	13.08	13.11	**6.54**	**6.54**	93.46
split-70000.180000	250,000	28	*99.99*	48.19	48.01	**28**	**28**	*99.99*
butterfly-30	33,285,996,544	48.28	–	–	–	98.26	–	–
compbip-250000.380000	630,000	39.68	**60.32**	79.37	79.46	84.39	–	–

Table 4. Number of requests performed in the second and third levels, expressed in percentage of m

Instance	m	LR	ED	S-Pitt	LL	SLL	ASLL
butterfly-28	15,032,385,540	*99.99*	**61.91**	91.59	*103.50*	*101.79*	72.62
debruijn-33	17,179,869,183	66.67	**61.11**	87.03	*108.91*	*108.33*	91.67
grid-75000.90000	13,499,835,000	*100*	**49.98**	86.24	91.67	91.67	75
hypercube-30	16,106,127,360	*100*	**9.10**	18.22	13.42	13.33	13.33
compbip-35000.500000	17,500,000,000	*100*	93	**92.97**	*100.01*	*100.01*	*100.02*
split-70000.180000	15,049,965,000	**0.002**	60.24	60.45	83.72	83.72	0.02
butterfly-30	64,424,509,440	–	–	–	*102.18*	–	–
compbip-250000.380000	95,000,000,000	*100*	34.21	34.05	**25.88**	–	–

Table 5. CPU running times obtained in the second and third levels (number of executions are given in parenthesis: we give the average time here)

Instance	LR	ED	S-Pitt	LL	SLL	ASLL
butterfly-28	**1:11:55**	1:15:53	1:18:41 *(2)*	1:20:35 *(2)*	*5:10:47*	*3:38:43*
debruijn-33	**1:20:37**	1:24:10	1:26:14 *(2)*	1:29:35 *(2)*	*7:43:53*	*5:08:37*
grid-75000.90000	**1:02:47**	1:08:57	1:09:58 *(2)*	1:11:35 *(2)*	*3:24:52*	*3:01:32*
hypercube-30	0:41:06	0:33:21	0:36:02 *(3)*	**0:33:19** *(3)*	1:07:46 *(2)*	1:07:38 *(2)*
compbip-35000.500000	0:23:15	0:22:23	0:22:28 *(4)*	**0:22:13** *(4)*	*6:11:51*	*6:12:03*
split-70000.180000	**0:00:17**	0:15:21	0:15:36 *(5)*	0:16:11 *(5)*	*4:27:39*	0:00:29 *(8)*
butterfly-30	–	–	–	5:47:43	–	–
compbip-250000.380000	2:02:16	1:01:19	1:01:21	**0:32:17**	–	–

Observations on Quality of Solutions. As we can see on Tables. 1 and 3, the algorithm LR is almost always the best. Moreover, it often returns the optimal solution. However, it can be very bad on complete bipartite and split graphs.

In general, SLL offers good performance, especially on random power law graphs (its performance is close to LR). Nevertheless, it is less efficient on regular graphs[5].

The global performance of algorithms S-Pitt and LL is intermediate but, for LL, it fluctuates more than S-Pitt. Indeed, on one instance, LL can be the best or the worst, that is not the case for S-Pitt.

Finally, ED and ASLL are overall the worst algorithms (and ED reaches often its approximation ratio of 2). For ED, these results confirm observations made by *F. Delbot et al.* [12].

Observations on the Number of Requests. As we can see on Tables. 2 and 4, the algorithm ED is almost always the best. Furthermore, it always performs less than m requests.

The algorithm LR often reaches m requests (it cannot perform worse), except on instances on which it returns a bad solution (it is better on them).

The performance of S-Pitt is close to LR: it is often better on specific graphs (except on complete split graphs) but it is worse on random graphs.

Algorithms LL, SLL and ASLL can perform more than m requests. This explains the fact that LR is generally the second algorithm in terms of complexity (even if its upper bound of m requests is often reached). ASLL is better, especially on random power law graphs and complete split graphs.

Analysis of Running Times. We focus on values presented for the second level in Tab. 5 (CPU running times obtained in first level are too similar to be exploited). To obtain an estimation of real running times (observed on our computer), one can multiply by 3.2 (resp. 1.6) CPU running times given in Tab. 5 for algorithms LR, ED, S-Pitt and LL (resp. SLL and ASLL).

[5] A *regular graph* is a graph where all the vertices have the same degree.

Algorithms SLL and ASLL are different because they have to use another playhead on `.deg` file to calculate degrees of neighbors. Therefore, their CPU running times are bigger. For this reason, we focus primarily on values observed for algorithms LR, ED, S-Pitt and LL.

On sparse graphs (where n and m are similar), CPU running times are close. They depend on number of requests performed by algorithms and size of covers constructed. Indeed, the size of solutions can be as huge as n, and writing on a disk is longer than reading. Moreover, there is often a trade off between the number of requests performed and the quality of solutions constructed: algorithms which produce the best solutions often perform the biggest number of requests (in any case, on one instance, an algorithm is never both the best in terms of quality of solution and complexity).

On dense graphs (where n is negligible compared to m), the analysis is less intricate because the size of covers written is tiny compared to the number of requets performed. Thus, CPU running times are mainly influenced by the number of requests done.

But these two criteria are not sufficient to explain CPU running times observed. Another technical aspects, linked to operating systems, are involved. Indeed, the access to the hard drive is indirect: the processing unit uses buffers. Also, the atomic unit of access depends on the size of disk sectors. If we read only one vertex on `.list` file, the system loads more vertices into its buffers. Hence, the number of physical access is lower than the number of requests performed. One overtakes in this regard practical considerations highlighted in the *I/O-efficient* model (see [20] for a survey).

Limits Encountered on our Machine. In the third level, we have generated two instances: a complete bipartite graph with 630,000 vertices and a *ButterFly* graph of dimension 30. On the complete bipartite graph, we were able to run algorithms LR, ED, S-Pitt and LL: their real running times do not exceed (on our computer) eight hours (executions of SLL and ASLL were stopped after twenty hours). On the *ButterFly* graph, we can only use LL (its real running time is about fifteen hours) because, on our computer, we cannot allocate an array of $33 \cdot 10^9$ bits (and executions of SLL and ASLL would take too long).

Therefore, LL is the only algorithm that can be run with our computer on all instances.

4 General Synthesis

We have implemented six algorithms for the VERTEX COVER problem on huge graphs. We were able to run these algorithms with a standard laptop computer on instances of sizes up to $30 \cdot 10^9$ vertices and edges (about 300 Gb on disk). The CPU running times we obtained do not exceed eight hours (and corresponding real running times are lower than ten hours).

We have observed that SLL and ASLL are almost always the slowest, since they have to compute degrees of neighbors. In this direction, they are "less adapted". However, SLL is still interesting, because it can give good solutions.

To test limits of algorithms, we have generated two instances of sizes about $100 \cdot 10^9$ vertices and edges (at least 1 Tb on disk): a complete bipartite graph with 630,000 vertices and $95 \cdot 10^9$ edges (a dense graph), and a *ButterFly* graph of dimension 30, with $33 \cdot 10^9$ vertices and $64 \cdot 10^9$ edges (a sparse graph).

- On the complete bipartite graph, we were able to run LR, ED, S-Pitt and LL (executions of SLL and ASLL were stopped before the end). For the slowest (LR), its CPU running time barely reaches two hours (and its corresponding real running time is about seven hours).
- On the *ButterFly* graph, we were only able to execute the algorithm LL: executions of LR, ED and S-Pitt failed because we could not allocate an array of $33 \cdot 10^9$ bits (and, as for the complete bipartite graph above, SLL and ASLL were stopped before the end).

General Observations. By summarizing the set of results presented for instances we used, among the six algorithms, LR is the one which gives the best solutions. It is closely followed by SLL, while ED and ASLL gives the worst solutions. However, ED performs the smallest number of requests. Based on that, choosing an algorithm which satisfies both quality of solutions and complexity in number of requests is difficult. On sparse graphs, where running times are often similar, we should promote the quality of solutions. Therefore, LR is a good candidate. Unfortunately, it needs to allocate an n bits array to be run, that is not always possible. On dense graphs, running times can change significantly, this makes choice trickier, since the most efficient algorithms are often the slowest.

Perspectives. We could extend our work by designing efficient algorithms on large instances for other problems. Then, we could compare our treatment method with the existing ones, e.g. with the semi-external greedy randomized adaptive search procedure presented in [2] for the MAX CLIQUE problem.

Acknowledgements

We would like to thank the anonymous referees for their insightful comments and suggestions, which have helped to improve the presentation of this paper.

References

1. `http://todo.lamsade.dauphine.fr/spip.php?article39`
2. Abello, J., Pardalos, P.M., Resende, M.G.C.: External Memory Algorithms, DIMACS, vol. 50, chap. On Maximum Clique Problems in Very Large Graphs, pp. 119–130. American Mathematical Society (1999)

3. Abello, J., Pardalos, P.M., Resende, M.G.C. (eds.): Handbook of Massive Data Sets, Massive Computing, vol. 4. Springer-Verlag (2002)
4. Ajwani, D.: Design, Implementation and Experimental Study of External Memory BFS Algorithms. Master's thesis, Max-Planck-Institut für Informatik, Saarbrücken, Germany (2005)
5. Alber, J., Dorn, F., Niedermeier, R.: Experimental Evaluation of a Tree Decomposition-Based Algorithm for Vertex Cover on Planar Graphs. Discrete Applied Mathematics 145, 219–231 (2004)
6. Angel, E., Campigotto, R., Laforest, C.: Analysis and Comparison of Three Algorithms for the Vertex Cover Problem on Large Graphs with Low Memory Capacities. Algorithmic Operations Research 6(1), 56–67 (2011)
7. Asgeirsson, E., Stein, C.: Vertex Cover Approximation on Random Graphs. In: 6th Workshop on Experimental Algorithms. vol. LNCS 4525, pp. 285–296 (2007)
8. Bar-Yehuda, R., Hermelin, D., Rawitz, D.: Minimum Vertex Cover in Rectangle Graphs. In: 18th Annual European Conference on Algorithms. pp. 255–266 (2010)
9. Bomze, I.M., Budinich, M., Pardalos, P.M., Pedillo, M.: Handbook of Combinatorial Optimization, chap. The Maximum Clique Problem, pp. 1–74. Kluwer Academic Publishers (1999)
10. de Bruijn, N.G.: A Combinatorial Problem. Koninklijke Nederlandse Akademie v. Wetenschappen 49, 758–764 (1946)
11. Delbot, F., Laforest, C.: A Better List Heuristic for Vertex Cover. Information Processing Letters 107, 125–127 (2008)
12. Delbot, F., Laforest, C.: Analytical and Experimental Comparison of Six Algorithms for the Vertex Cover. ACM Journal of Experimental Algorithmics 15 (2010)
13. Escoffier, B., Gourvès, L., Monnot, J.: Complexity and Approximation Results for the Connected Vertex Cover Problem in Graphs and Hypergraphs. Journal of Discrete Algorithms 8, 36–49 (2010)
14. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
15. Gilmour, S., Dras, M.: Kernelization as Heuristic Structure for the Vertex Cover Problem. In: 5th International Workshop on Ant Colony Optimization and Swarm Intelligence. vol. LNCS 4150, pp. 452–459 (2006)
16. Molloy, M., Reed, B.: A Critical Point for Random Graphs With a Given Degree Sequence. Random Structures and Algorithms pp. 161–179 (1995)
17. Pirzada, S., Dharwadker, A.: Applications of Graph Theory. Journal of The Korean Society for Industrial and Applied Mathematics (KSIAM) 11(4), 19–38 (2007)
18. Pitt, L.: A Simple Probabilistic Approximation Algorithm for Vertex Cover. Tech. Rep. 404, Yale University, Department of Computer Science (1985)
19. Vigier, F., Latapy, M.: Random Generation of Large Connected Simple Graphs with Prescribed Degree Distribution. In: 11th International Conference on Computing and Combinatorics. Kunming, Yunnan, Chine (2005)
20. Vitter, J.S.: Algorithms and Data Structures for External Memory, vol. 2. Foundations and Trends in Theoretical Computer Science, Boston – Delft (2009)
21. Weisstein, E.W.: Butterfly graph, from MathWorld – A Wolfram Web Ressource: `http://mathworld.wolfram.com/ButterflyGraph.html`