RAIRO-Oper. Res. 47 (2013) 199–221
DOI: 10.1051/ro/2013034

RAIRO Operations Research
www.rairo-ro.org

SOLVING THE MINIMUM INDEPENDENT DOMINATION SET PROBLEM IN GRAPHS BY EXACT ALGORITHM AND GREEDY HEURISTIC *

Christian Laforest[1] and Raksmey Phan[1]

Abstract. In this paper we present a new approach to solve the Minimum Independent Dominating Set problem in general graphs which is one of the hardest optimization problem. We propose a method using a clique partition of the graph, partition that can be obtained greedily. We provide conditions under which our method has a better complexity than the complexity of the previously known algorithms. Based on our theoretical method, we design in the second part of this paper an efficient algorithm by including cuts in the search process. We then experiment it and show that it is able to solve almost all instances up to 50 vertices in reasonable time and some instances up to several hundreds of vertices. To go further and to treat larger graphs, we analyze a greedy heuristic. We show that it often gives good (sometimes optimal) results in large instances up to 60 000 vertices in less than 20 s. That sort of heuristic is a good approach to get an initial solution for our exact method. We also describe and analyze some of its worst cases.

Keywords. Combinatorial optimization, heuristics, exact algorithm, worst case analysis, experimentations, independent dominating set in graphs.

Mathematics Subject Classification. 05C69, 05C85.

Received June 13, 2012. Accepted March 19, 2013.

* *This work is supported by the French Agency for Research under the DEFIS program TODO, ANR-09-EMER-010.*

[1] LIMOS, CNRS UMR 6158, Université Blaise Pascal, Clermont–Ferrand Campus des Cézeaux, 24 avenue des Landais, 63173 Aubière Cedex, France.
christian.laforest@isima.fr.

Article published by EDP Sciences

1. Introduction

In discrete optimization, many problems are hard to solve exactly in polynomial time and need exponential time algorithms. A theoretical goal is to propose exact algorithms with (exponential) complexity as small as possible. However, these methods are not always implemented and there is no practical information available on them. In this paper, we propose an exact algorithm to solve a well−known problem, namely the minimum size independent dominating set. We analyse its worst-case complexity, implement and experiment it using efficient rules to avoid to explore useless solutions. For large graphs, we present a greedy heuristic to compute a solution and show that it is close to the optimal. Let us first define the problem.

In this paper we consider an undirected, unweighted finite graph $G = (V, E)$ where V is the set of its *vertices* and E is the set of its *edges*. If uv is an edged between vertices u and v then u and v are *adjacent* or *neighbors*. A *dominating set* in a graph $G = (V, E)$ is a subset $D \subseteq V$ such that every vertex $v \in V \backslash D$ is neighbor of at least one vertex in D. An *independent set* in G is a subset I of vertices such that no two of them are adjacent. The *minimum Independent Dominating Set* ($mIDS$) problem is to find an *independent dominating set* of minimum size in a given graph G. It is known to be NP−hard [3]. Magnús M. Halldórsson (1993) [4] showed that $mIDS$ problem cannot be approximated under a factor $n^{1-\epsilon}$ with $\epsilon > 0$ in polynomial time, unless $P = NP$ (where n is the number of vertices). In practice, this kind of problem is usual in communication and network optimization (see for example [8]).

Various studies have already been done on specific families of graphs. In 2012, Song *et al.* [14] proposed a criterion to classify the computational complexity of independent domination on tree convex bipartite graphs; they proved that in some of these graphs the problem is not NP-Complete. Recently Shiu *et al.* (2010) [13] proposed an upper bound on Triangle−free graphs. In 2009, Orlovich *et al.* [11] studied $2P_3$–free perfect graphs and in 2007 Julie Havland [6] studied regular graphs.

For general graphs, the trivial exhaustive $O^*(2^{|V|})$ complexity has been broken up in 1988 by Johnson *et al.* [7]: The notation $O^*(.)$ is used to express the non polynomial part of the complexity of an algorithm (ignoring polynomial factors). In 2006, Liu and Song [10] developed a smart $O^*(\sqrt{3}^{|V|})$ time algorithm to solve this problem on general graphs. They used maximum matchings to partition the graph and reduce problem complexity. Bourgeois, Escoffier, and Paschos [2] in 2010 devised a branching algorithm that finds the $mIDS$ in $O^*(2^{0.424|V|})$ running time.

All these works are theoretical since the complexity is evaluated in worst-case mode and that the algorithms were not implemented and experimented by their authors. In 2011, Anupama Potluri and Atul Negi [12] have computed and compared the algorithm of Liu and Song [10] with their *Intelligent Enumeration Algorithm*.

In this paper, we firstly (Sect. 2) present our new approach to solve the $mIDS$ problem, then we prove that our algorithm tends to have a complexity better than

any other known algorithm regarding the problem when the graph is large and dense. We then describe our algorithm (Sect. 3) to compute the $mIDS$ in practice and we give results of our experimentations. Finally (Sect. 4) we analyze and experiment polynomial time algorithms to calculate a lower bound and a greedy heuristic to compute an independent dominating solution on large graphs (up to 60 000 vertices). Note that for future comparisons, our instances and our code are available online at: `http://todo.lamsade.dauphine.fr/spip.php?article42`.

2. Description of the clique partition approach and its analysis

In [10] Liu and Song use maximum matchings to reduce the trivial complexity $O^*(2^{|V|})$ of the standard branching algorithm for the $mIDS$ problem. In this paper, we extend/adapt a part of their technique by using a partition of the vertices of the graph in cliques (a *clique* is a graph in which every pair of vertices are connected by an edge) instead of a matching. To describe our method, we need some preliminary notations and definitions. We call a *clique partition* of a graph $G = (V, E)$ a partition of the set V of vertices into disjoint subsets $C = \{C_1, \ldots, C_k\}$ ($\cup_{i=1}^{k} C_i = V$ and if $i \neq j$, $C_i \cap C_j = \emptyset$) such that each subgraph induced by C_i in G is a clique (its number of vertices is denoted by c_i ($1 \leq c_i \leq |V|$)).

A clique partition of any graph G can easily be obtained by a greedy polynomial algorithm (see Sect. 3.4). Suppose now we have a clique partition $C = \{C_1, \ldots, C_k\}$ of G.

2.1. Clique partition approach

As each C_i is a clique, any independent dominating set has *at most* one vertex in each C_i (otherwise S would not be an independent). Based on this remark, the idea of our algorithm is to construct each possible set S composed of *at most one vertex* of each clique of C and, for each such S to test if S is an independent (no edge between vertices of G) and if S is a dominating set of G (each vertex in $V \backslash S$ has at least one neighbor in S). If it is so, we update the minimum size independent dominating set found from the beginning and return the smallest one at the end. Note that each test can easily be done in polynomial time.

Doing this construction as described, we generate every possible solution (and also sets that are not solution) and hence any optimal one.

Each "explored" set S is composed of at most one vertex of each clique of C. This represents exactly c_i+1 possibilities for each clique C_i of C. Thus, the number of candidates S tested during our algorithm is exactly:

$$\prod_{i=1}^{k} (c_i + 1)$$

As each test is polynomial, our algorithm runs in $O^*\left(\prod_{i=1}^{k} (c_i + 1)\right)$.

2.2. Complexity analysis

The goal of this subsection is to show that when G is large and dense enough, our algorithm tends to have a complexity better than any exponential complexity of the type $A^{b|V|}$ where $(A, b) \in (\mathbb{N}\backslash\{0,1\}, \mathbb{R}_+^*)$. This analysis will be used in Sections 2.3 and 2.4 to find conditions in which our method has better complexity than the ones published by Bourgeois, Escoffier, and Paschos [2] ($A = 2$ and $b = 0.424$) and by Liu and Song [10] ($A = 3$ and $b = 0.5$).

In a first time we give conditions to have:

$$\prod_{i=1}^{k} (c_i + 1) \leq A^{b|V|} \tag{2.1}$$

As in (2.1) $c_i, i = 1, \ldots, k$ represent the size of cliques of the partition of V we have: $\sum_{i=1}^{k} c_i = |V|$.

The integers $c_1, \ldots, c_k$ represent what is called a partition of the integer $|V|$. More generally, let $n > 0$ be any positive integer, a *partition of* n is a list of positive (non null) integers $[n_1, \ldots, n_l]$ such that $\sum_{i=1}^{l} n_i = n$.

The *length* of the partition $[n_1, \ldots, n_l]$ is l. Two lists composed of the same elements in different orders are equal. We denote by $P(n)$ the *set* of all partitions of n and $p(n)$ its size. For example, if $n = 5$ we have:

$$P(5) = \{[1,1,1,1,1], [1,1,1,2], [1,2,2], [1,1,3], [2,3], [1,4], [5]\} \text{ and } p(5) = 7.$$

We denote now by $Q(|V|)$ the set (and by $q(|V|)$ its size) of partitions of the integer $|V|$ that *do not* verify inequality (2.1), when A and b are given. For example, if $A = 3$ and $b = \frac{1}{2}$, this is the expression of the complexity of algorithm of [10], when $|V| = 5$ we get $Q(5) = \{[1,1,1,1,1], [1,1,1,2], [1,2,2], [1,1,3]\}$.

We now show that when n tends to infinity, $\frac{q(n)}{p(n)}$ tends to 0 (see Thm. 2.4). This shows that asymptotically, the proportion of partitions of n violating (2.1) among all the partitions of n tends to 0.

To do that, we need several preliminary results. Lemma 2.1 gives a sufficient condition for a partition $c_i, i = 1, \ldots, k$ of $|V|$ to verify inequality (2.1). This proves that if we can have a partition of $|V|$ in which the average−value of c_i is greater than a certain value to determine (denoted by $M(A, b)$ in the Lemma), then the inequality (2.1) is satisfied.

Lemma 2.1. *Let* $(A, b) \in (\mathbb{N}\backslash\{0,1\}, \mathbb{R}_+^*)$ *then there exists* $M(A, b) \in \mathbb{R}$ *such that:*

$$M(A,b) \leq \frac{1}{k} \sum_{i=1}^{k} c_i \Rightarrow \prod_{i=1}^{k} (c_i + 1) \leq A^{b|V|}$$

Proof. We study $f(x) = A^x - \frac{1}{b}x - 1$. We get the derivative of f, $f'(x) = A^x \ln(A) - \frac{1}{b}$. Let $x_1 = \frac{\ln\left(\frac{1}{\ln(A)b}\right)}{\ln(A)}$; $\forall x > x_1$, $f(x)$ increases. Moreover as $\lim_{x\to\infty} f(x) = \infty$ there exists $g \geq x_1$ such that $\forall x > g$, $f(x)$ increases and is greater than 0.

We choose $M(A,b) = \frac{g}{b}$. Now suppose that $M(A,b) \leq \frac{1}{k}\sum_{i=1}^{k} c_i$ then:

$$\frac{g}{b} = M(A,b) \leq \frac{1}{k}\sum_{i=1}^{k} c_i \quad \Leftrightarrow \quad g \leq b.\frac{1}{k}\sum_{i=1}^{k} c_i$$

As we have $f(x) = A^x - \frac{1}{b}x - 1 \geq 0, \forall x \geq g$ we get:

$$A^{b\frac{\sum_{i=1}^{k} c_i}{k}} - \frac{1}{b}.b.\frac{1}{k}\sum_{i=1}^{k} c_i - 1 \geq 0 \Leftrightarrow A^{b\frac{\sum_{i=1}^{k} c_i}{k}} \geq \frac{1}{k}\sum_{i=1}^{k} c_i + 1 = \frac{\sum_{i=1}^{k}(c_i+1)}{k}$$

We remind the inequality of arithmetic and geometric means (see [15] for *e.g.*):

$$\frac{\sum_{i=1}^{k} x_i}{k} \geq \left(\prod_{i=1}^{k} x_i\right)^{\frac{1}{k}} \tag{2.2}$$

Finally using this and because $\sum_{i=1}^{k} c_i = |V|$, we obtain $\prod_{i=1}^{k}(c_i+1) \leq A^{b|V|}$. □

Lemma 2.1 proves that given A and b, there exists a value $h = M(A,b)$ such that, if $h \leq \frac{1}{k}\sum_{i=1}^{k} c_i$ then $\prod_{i=1}^{k}(c_i+1) \leq A^{b|V|}$. Given this value h, we define $M_h(n)$ as the set (and $m_h(n)$ its size) of partitions $[n_1,\ldots,n_l] \in P(n)$ of n that verify $h > \frac{1}{l}\sum_{i=1}^{l} n_i$, *i.e.* that have an *average value* strictly smaller than h and thus that do not respect the sufficient condition of Lemma 2.1 (when $h = M(A,b)$). If we denote by $n = |V|$ and $h = M(A,b)$, we have $Q(n) \subseteq M_h(n)$ because any partition $[n_1,\ldots,n_l]$ of $Q(n)$ verifies $h > \frac{1}{l}\sum_{i=1}^{l} n_i$ otherwise it would satisfy the sufficient condition of Lemma 2.1 and thus would verify (2.1), hence would not be in $Q(n)$. Based on this remark, we prove in what follows that $\frac{m_h(n)}{p(n)} \to 0$ for $n \to \infty$ that is sufficient to get $\frac{q(n)}{p(n)} \to 0$.

We remark that as a partition $[n_1,\ldots,n_l] \in M_h(n)$ verifies $h > \frac{1}{l}\sum_{i=1}^{l} n_i$, and as $\sum_{i=1}^{l} n_i = n$ we equivalently have $l > \frac{n}{h}$; this means that $M_h(n)$ is also the set

of partitions of n of length greater than $\frac{n}{h}$. This is why we introduce now the set $P(n,k)$ (of size $p(n,k)$) of partitions of n of length k (k is an integer).

Lemma 2.2. *$p(n,n-i) \leq p(i)$ for all $i \in \{1,\ldots,n\}$.*

Proof. Let $X = [x_1,\ldots,x_{n-i}] \in P(n,n-i)$. Let $Z = [z_1,\ldots,z_l]$ be the list obtained by subtracting 1 from each element of X and by keeping only values that are strictly greater than 0 ($z_k \geq 1, \forall k \leq l \leq n-i$).

Hence, $\sum_{k=1}^{l} z_k = \sum_{k=1}^{n-i} x_i - (n-i) = n-(n-i) = i$ and Z is a partition of i. Now, let $X' = [x'_1,\ldots,x'_{n-i}] \in P(n,n-i)$, be another partition. We denote by Z' the list obtained by the same process from X'. Suppose that $Z = Z'$. As X and X' have same length and since $Z = Z'$, X and X' have the same number of 1 (that "disappear" in Z and Z'). The decreasing by one unit of the other l components (l is the length of $Z = Z'$) gives Z and Z'. This means that these non 1 components are equal in X and X' and thus $X = X'$ (the order of the elements in the lists are not important); there is a contradiction with the hypothesis that $X \neq X'$.

This shows that for each distinct partition of n of length $n-i$ we can extract at least one distinct partition of i. Then $p(n,n-i) \leq p(i)$. □

For the next two results, we suppose that A and b are given. We denote by $h = M(A,b)$, the value of Lemma 2.1 and $q(n)$ is also defined according to A and b.

Lemma 2.3. *With the previous notations, we have:* $m_h(n) \leq 1 + \sum_{i=1}^{n-\lceil \frac{n}{h} \rceil} p(i)$.

Proof. As mentioned previously, every partition of $P(n)$ of length greater than $\frac{n}{h}$ has a mean-value strictly less than h. These partitions are the ones of $M_h(n)$ so:

$$|M_h(n)| = m_h(n) = \sum_{k=\lceil \frac{n}{h} \rceil}^{n} p(n,k).$$

Because $p(n,n) = 1$ we can write: $m_h(n) = 1 + \sum_{k=\lceil \frac{n}{h} \rceil}^{n-1} p(n,k)$.

Now let $k = n-i$ then: $m_h(n) = 1 + \sum_{i=1}^{n-\lceil \frac{n}{h} \rceil} p(n,n-i)$.

Then with Lemma 2.2: $m_h(n) \leq 1 + \sum_{i=1}^{n-\lceil \frac{n}{h} \rceil} p(i)$. □

We are now ready to show that the proportion in $P(n)$ of partitions in $M_h(n)$ (and thus of $Q(n)$) tends to 0 when n tends to infinity.

Theorem 2.4. *It holds that:* $\lim_{n\to\infty} \frac{q(n)}{p(n)} = 0$

Proof. As $q(n) \leq m_h(n)$ it is sufficient to show that $\lim_{n\to\infty} \frac{m_h(n)}{p(n)} = 0$. Lemma 2.3 proves $m_h(n) \leq 1 + \sum_{i=1}^{n-\lceil \frac{n}{h} \rceil} p(i)$ and as $p(n)$ is strictly increasing we get:

$$m_h(n) \leq 1 + \left(n - \left\lceil \frac{n}{h} \right\rceil\right) p\left(n - \left\lceil \frac{n}{h} \right\rceil\right) \Leftrightarrow \frac{m_h(n)}{p(n)} \leq \frac{1}{p(n)} + \frac{(n - \lceil \frac{n}{h} \rceil)}{p(n)} p\left(n - \left\lceil \frac{n}{h} \right\rceil\right)$$

We have $\lim_{n\to\infty} \frac{1}{p(n)} = 0$. For the second part of the equation, we use the asymptote of Ramanujan's formula [1]: $p(n) \sim \frac{1}{4n\sqrt{3}} e^{\pi\sqrt{\frac{2n}{3}}}$ as $n \to \infty$.

$$\frac{(n - \lceil \frac{n}{h} \rceil)}{p(n)} p\left(n - \left\lceil \frac{n}{h} \right\rceil\right) = \left(n - \left\lceil \frac{n}{h} \right\rceil\right) \frac{\frac{1}{4} \exp^{\pi\sqrt{\frac{2}{3}\left(n - \lceil \frac{n}{h} \rceil\right)}}}{\frac{(n - \lceil \frac{n}{h} \rceil)\sqrt{3}}{\frac{1}{4} \frac{\exp^{\pi\sqrt{\frac{2}{3}n}}}{n\sqrt{3}}}} = n \exp^{\pi\sqrt{\frac{2}{3}}\left(\sqrt{n - \lceil \frac{n}{h} \rceil} - \sqrt{n}\right)}$$

and $\lim_{n\to\infty} n \exp^{\pi\sqrt{2/3}\left(\sqrt{n - \lceil \frac{n}{h} \rceil} - \sqrt{n}\right)} = 0$. Then: $\lim_{n\to\infty} \frac{m_h(n)}{p(n)} = 0$. □

We can apply these results to compare our complexity with the ones of Liu and Song [10], $O^*(\sqrt{3}^{|V|})$ ($A = 3$, $b = 1/2$) and those of Bourgeois *et al.*, [2], $O^*(2^{0.424|V|})$ ($A = 2$ and $b = 0.424$). In both cases, our algorithm has better complexity when the partition in cliques is good; the proportion of partitions of $|V|$ inducing a complexity greater tends to 0 when the size of G tends to infinity.

Of course, given a specific graph $G = (V, E)$ with n vertices, there are not always $p(n)$ partitions of the set V. In particular, when G is not dense the number of possible partitions of V will be less than $p(n)$. Hence, if the number of partitions of V is close to $p(n)$ then the probability to obtain a good partition of V is high. This is the case when G is dense.

2.3. Comparison with branching algorithm: Bourgeois *et al.*

In the article of Bourgeois, Escoffier and Paschos [2], the authors devise a branching algorithm that computes a $mIDS$ with running time $O^*(2^{0.424|V|})$. In what follows we find a value h such that:

$$h \leq \frac{\sum_{i=1}^{k} c_i}{k} \Rightarrow \prod_{i=1}^{k} (c_i + 1) \leq 2^{0.424|V|}$$

Let $f(x) = 2^x - \frac{1000}{424}x - 1$ and $f'(x) = 2^x \ln(2) - \frac{125}{52}$. Trivially, $f'(x) > 0, \forall x \geq 2$ and $f(x) \geq 0, \forall x \geq 3.03$. Then $f(x) \geq 0, \forall x \geq 3.03$.

By using the result of proof of Lemma 2.1, we deduce $h = 7.2$ and we have:

$$2^{0.424|V|} \geq \prod_{i=1}^{k} (c_i + 1)$$

Hence if we find a clique partition of the given graph with an average−value greater than 7.2, our algorithm has a better exponential factor than $2^{0.424|V|}$. Moreover Theorem 2.4 proves that the number of partitions with an average−value less than 7.2 is insignificant in comparison to $p(n)$ when n is large.

2.4. Comparison with exact algorithm of Liu and Song

In [10] the authors develop a simple $O^*(\sqrt{3}^{|V|})$ algorithm to solve $mIDS$ in general graphs. We propose a value h such that:

$$h \leq \frac{\sum_{i=1}^{k} c_i}{k} \Rightarrow \prod_{i=1}^{k} (c_i + 1) \leq \sqrt{3}^{|V|}$$

Let $f(x) = 3^x - 2x - 1$ and $f'(x) = 3^x \ln(3) - 2$. Trivially, $f'(x) > 0, \forall x \geq 1$ and $f(x) \geq 0, \forall x \geq 1$. Then $f(x) \geq 0, \forall x \geq 1$.

By using the result of proof of Lemma 2.1, we deduce $h = 2$ and we have:

$$\sqrt{3}^{|V|} \geq \prod_{i=1}^{k} (c_i + 1)$$

This means that if we find a clique partition of the given graph with an average−value greater than 2, our algorithm has a better exponential factor than $\sqrt{3}^{|V|}$. Moreover Theorem 2.4 proves that the number of partitions with an average−value less than 2 is insignificant comparing to $p(n)$ when n is large.

3. Implementation and experimentation of our exact algorithm for the $mIDS$

In this section we describe precisely our method to compute an $mIDS$ solution. In the first part, we describe again our exhaustive algorithm to compute the $mIDS$ based on clique partition (developed in Sect. 2). In the second part, we adapt it by adding several conditions to reduce the number of solutions that are checked. We also describe the (already known) greedy heuristic that we use to compute a first IDS solution before starting our main algorithm. Finally, we implement and test these methods on several families of graphs.

3.1. Exhaustive search based on clique partition

In this subsection we give the main notations that will be used throughout the rest of this section and we remind our exhaustive search that will be improved in next subsections. We consider that we have a graph $G = (V, E)$ and a *clique partition* of G, $C = \{C_1, \ldots, C_k\}$ (see Sect. 2.1).

Let S be any *independent dominating set* (IDS) of G. As S is an independent set and as each C_i induces a clique, we have: $|C_i \cap S| \leq 1$. The idea of our exhaustive method is to construct each subset S of V having at most one vertex in each C_i (thus satisfying $|C_i \cap S| \leq 1$, $\forall i \in \{1, \ldots, k\}$), to check if it is an IDS and, at the end, to return the one with minimum size. We are sure to get the optimal one since this constructs each IDS. The number of candidate sets produced and tested is then $\prod_{i=1}^{k} (|C_i| + 1)$. Note that our method can be implemented using a recursive branching algorithm, exploring cliques of C one by one in a given order, that we fix here as: C_1, then C_2, etc. until C_k. At each recursive level i, there are $|C_i|+1$ choices: one vertex of C_i or no vertex to be included in the current solution under construction (the exhaustive search examines all these choices). When an IDS is found, the best solution found from the beginning is updated (replaced by the current solution if it is better).

To initialize the best solution, we just need any IDS of G. For some efficiency considerations, we choose the solution constructed by a greedy heuristic (explained later in Sect. 3.3).

3.2. Adding cuts to our exhaustive search based on clique partition

In Section 3.1 we described our general search method. We now improve it by adding cuts in this recursive exhaustive scheme. This technique is inspired by the well known Branch and Bound heuristic (see for *e.g.* [9]). This helps us to avoid many candidate sets that are useless to consider and to cut into this recursive branching algorithm. To illustrate our approach and cuts, we will use $G_{\text{Example}} = (V, E)$, an undirected graph with 13 vertices and 21 edges (Fig. 1). In the same figure bold edges form one possible clique partition (note that 8 is the only vertex in clique C_3).

Here are our cuts that we add to our general method.

"Domination" Cut: If at step i all vertices are already dominated we do not need to explore further. We cut the exploring branch at this point. Suppose, at the beginning, we have a clique partition $C = \{\{1, 2, 3, 4\}, \{5, 6, 7\}, \{8\}, \{9, 10, 11\}, \{12, 13\}\}$ of the graph G_{Example} of Figure 1. Now suppose that the execution is at recursive level 4 and that the current solution is $S = \{1, 5, 8, 11\}$. Figure 2 gives a graphical representation of the situation (edges into the cliques are not drawn). At this current step, the solution S already dominates all the graph (and is independent) then no need to proceed further in subsequent recursive searching steps.

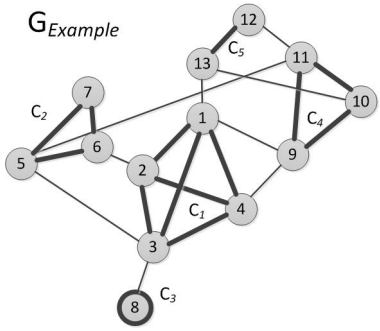

FIGURE 1. G_{Example}: An Example of Graph and Clique Partition.

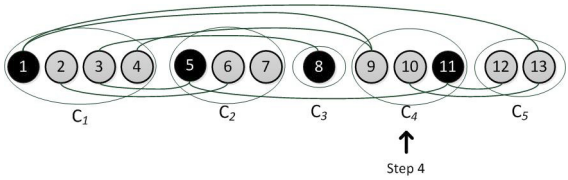

FIGURE 2. Clique Partition: Domination Cut.

"Only Dominated by Its Own Clique" Cut: We consider one vertex v in clique C_i and its neighbors. Now suppose it has *all* its neighbors in clique C_i. We deduce that every solution S that does not have at least one vertex in C_i, cannot dominate v. That means we can ignore the branch of the searching process with no vertex of C_i.

"Last Chance to Dominate" Cut: Each vertex can only be dominated by a limited number of vertices (its neighbors). From a given clique partition, each vertex can only be dominated by a limited number of cliques (containing its neighbors). Then from a given ordered clique partition $(C_1, C_2, \ldots, C_k)$, for each vertex we compute (in a pre−processing phase) an index containing the number of the last clique that contains at least one of its neighbors. We use this index to detect non dominated vertices that cannot longer be dominated. More precisely, suppose we have a current solution S at step i, and that there is one non dominated vertex v in clique C_k with $k < i$ and v does not have neighbors in cliques C_m $\forall m \geq i$. Then v cannot longer be dominated and it is useless to continue with current solution. For example, from the previous clique partition C, suppose that at step 3 (clique 3) we have the current solution $S = \{5\}$. In this cases whatever we do at Step 3, Steps 4 and 5, solution S will be unfeasible because vertex 2 can never be dominated (Fig. 3).

"Minimum Size" Cut: This is a classical and simple cut in a minimum optimization problem. In each step we compare the size of the current candidate solution with the best known solution found in previous steps. We also cut branches if

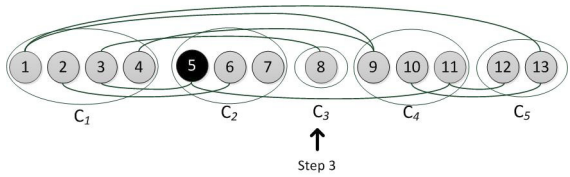

FIGURE 3. Clique Partition: Last Chance to Dominate.

the current solution is already larger than the last one. It avoids all solutions that will not be better than the best known.

The three first cuts previously presented only reduce the number of unfeasible solutions. If we only use them, we have an enumerating algorithm for IDS and $mIDS$. The "Minimum size" cut is the only cut reducing the number of feasible solutions.

3.3. First feasible solution: greedy heuristic (GH)

As described in previous subsections, our method needs a first feasible solution to start and to serve as the current best solution (for the last cut rule). We have implemented a *Greedy Heuristic* (GH) to construct it. At each step of the GH algorithm, we add into the current solution S a non dominated vertex u having the maximum number of non dominated neighbors, then we mark these neighbors and u itself as dominated. We repeat that step until all vertices are dominated. It is easy to see that this ensures to have an IDS of G.

We will show in the experimental Section 3.5 that GH constructs pretty good solutions in general. But we also show in Sections 3.3.1 and 3.3.2 that there are particular graphs for which the solutions it returns are far from the optimal.

3.3.1. Special star graph

Let k be an integer. Let $G_{SpecialStar} = (V, E)$ be the graph containing $|V| = k^2 - k + 1$ vertices. It is constructed as follows. A vertex a has k neighbors: $u_1, \ldots, u_k$. Each u_i has $k-1$ neighbors: a and $v_{i,1}, \ldots, v_{i,k-2}$ (see Fig. 4).

It is easy to see that GH constructs a feasible solution with $1 + k(k-2)$ vertices (selecting first vertex a of higher degree then all the remaining non dominated vertices $v_{i,j}$) while the optimal solution has only k vertices ($u_1, \ldots, u_k$ are sufficient). Figure 5 gives an example of these solutions when $k = 4$ (the worst solution has 9 vertices while the best has only 4 vertices).

3.3.2. Special two subsets graph

Let k be an integer. $G_{STS} = (V, E)$ is a graph with $2k+1$ vertices. Vertex a has $k+1$ neighbors: b, c and $v_1, \ldots, v_{k-1}$. Vertex b has k neighbors: a and $v_1, \ldots, v_{k-1}$. Vertex c has k neighbors: a and $u_1, \ldots, u_{k-1}$ (see Fig. 6).

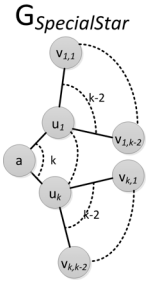

FIGURE 4. Special Star Graph.

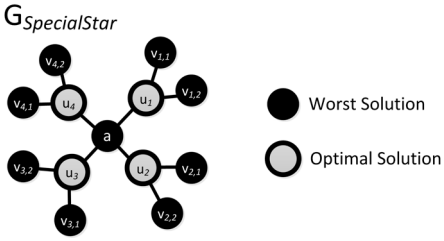

FIGURE 5. Special Star Graph: Example for $k = 4$.

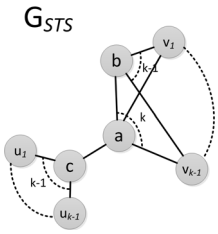

FIGURE 6. Special Two Subsets Graph

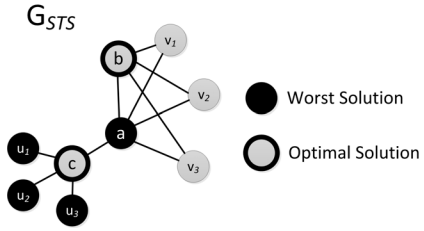

FIGURE 7. Special Two Subsets Graph: Example for $k = 4$

Applying GH gives a feasible solution with k vertices (a and $u_1, \ldots, u_{k-1}$) while the optimal solution has only 2 vertices (b and c). Figure 7 provides an example with $k = 4$ (the worst solution has 4 vertices while the best has 2 vertices).

3.4. Creation of a clique partition of G

Our method needs a clique partition of G. We use a simple greedy algorithm, *Stochastic Greedy Approach* (SGA), to construct it.

Let us call *free vertex* a vertex that is not in a clique C_i. At the beginning of SGA, all vertices are *free* and SGA stops when no vertex is free. SGA creates a clique C_i with a free random vertex v. SGA adds iteratively into the current clique C_i one of the common free neighbors of all vertices in C_i until there are no more. Then it creates a new clique C_{i+1} with a new free vertex randomly selected (if there is one; otherwise stop) and continues the process. Figure 1 illustrates one of possible clique partitions for graph G_{Example}.

3.5. Computational evaluation

In this subsection we present computational results of our method described in Section 3.2 on various graphs. Our software is implemented in C language. We

TABLE 1. Results: Random Graphs With Probability 0.1.

	$n = 80$	90	100	110
$NbExec$	10	10	0	5
$Greedy\_time$	0.00	0.00	0.00	0.00
$\|Greedy\|$	16	14	13	15
Opt_time	740.20	8049.50	38460.00	126985.00
Opt_size	10	11	12	12
$NbUpdate$	5.00	3.00	1.00	3.00

have evaluated it on hundreds of different graphs that we describe in the following subsection.

Our graphs have been created with Maple software and then imported as input by our C program. Each instance is executed 10 times on an AMD Opteron Processor 2352 clocked at 2.1 GHz, with different seeds to have average values.

The tables in the following subsections present our experimental results. The number of vertices of the graphs is denoted by n. Each line contains: The number of executions "$NbExec$" (this number can be less than 10 because some executions have exceed), the running time of Greedy Heuristic (GH) "$Greedy\_time$" in seconds (average value), the size of the solution constructed by the GH "$|Greedy|$", the running time of our algorithm "Opt_time" in seconds (average value), the size of the $mIDS$ solution "Opt_size" and the number of improvements done by our algorithm "$NbUpdate$" (average value) between the greedy initial solution and the final one.

3.5.1. Random graphs

We use the classical random graphs model: For an integer n and a probability p, we have a graph with n vertices and each possible edge is generated with probability p.

Table 1 reports statistics for a probability $p = 0.1$. The running time rises very fast from 740 s for 80 vertices to more than 126 000 s (almost one and a half days) for 110 vertices.

Figure 8 summarizes a part of the experimentations: The running time is on the vertical-axis and the number of vertices on the horizontal−axis. Only probabilities 0.2, 0.3 and 0.4 are shown. We can see that our software can solve large graphs in reasonable running time when p is larger than 0.3.

3.5.2. Random tree graphs

According to previous results, we note that our algorithm is slower when p is small, *i.e.* when G has low density. This is why we now test it on Random Tree Graphs. A tree is an undirected connected graph with n vertices and $n - 1$ edges (so very sparse).

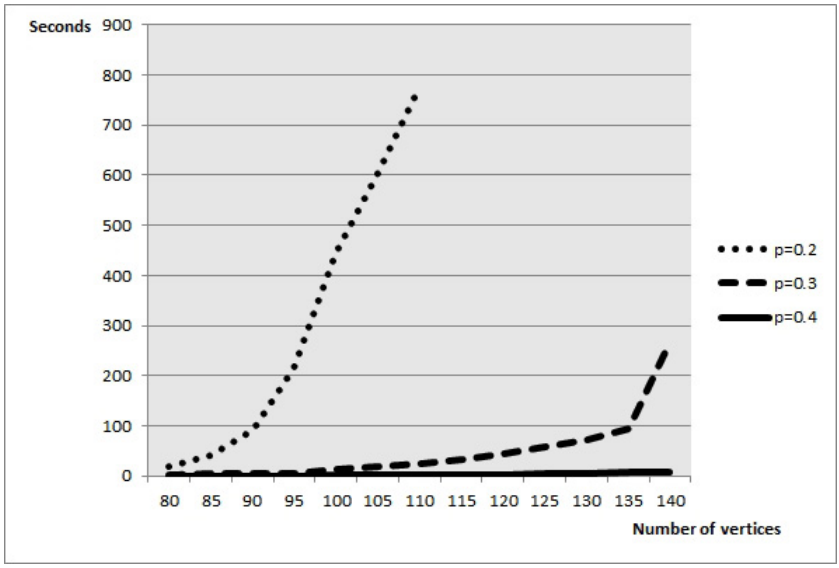

FIGURE 8. Evolution of The Running Time Depending on Number of Vertices.

TABLE 2. Results: Random Tree Graphs.

	$n = 50$	55	60	65	70	75
$NbExec$	10	10	10	10	10	10
$Greedy\_time$	0.00	0.00	0.00	0.00	0.00	0.00
$\|Greedy\|$	18	20	23	26	27	30
Opt_time	2.90	5.30	41.90	37.20	210.70	881.30
Opt_size	17	20	23	24	26	26
$NbUpdate$	1.00	0.00	0.00	2.00	1.00	4.00

Table 2 reports results for different number of vertices. For 50 vertices, the running time is less than 3 s. We are able to solve instances with up to 75 vertices in less than 15 minutes. We also can note that GH is very efficient here: Its size is very close of the optimal, for a quasi null time.

We extend our tests on Path Graphs which are particular trees: Results are similar even if our method seems to take more computational time to find $mIDS$ solution.

3.5.3. Hypercube graphs

We have seen that our method is efficient on dense graphs. We now test our software on Hypercube Graphs. For any integer d, a hypercube is a graph $G = (V, E)$ with $|V| = 2^d$ and $|E| = 2^{d-1}d$. Each vertex is labeled by a vector of d components of $\{0, 1\}$ and two vertices are connected if and only if their vectors

TABLE 3. Results: Hypercube Graphs.

	$n = 4$	8	6	32	64
NbExec	10	10	10	10	10
Greedy_time	0.00	0.00	0.00	0.00	0.00
$\|Greedy\|$	2	2	4	8	16
Opt_time	0.00	0.00	0.00	0.00	423.30
Opt_size	2	2	4	8	12
NbUpdate	0.00	0.00	0.00	0.00	2.60

TABLE 4. Grid Graphs: Results Comparison.

	$n = 25$	36	49	56
Opt_time	0.00	0.00	9.90	630.00
Opt_size	7	10	12	16
NbUpdate	1.00	2.70	4.10	3.60
IEA	1.00	254.00	141242.00	–
Liu and Song Algorithm	11.00	39225.00	–	–

differ on exactly one component. Hypercube Graphs are slightly dense and all vertices have exactly the same degree d (no side effect). In spite of its density, the maximum size of cliques is 2. Thus, our method does not benefit of large cliques.

Table 3 reports results for different number of vertices. We show that we can solve a large Hypercube Graphs (of 64 vertices) in reasonable running time (approximately 7 min). For smaller Hypercube Graphs, the solution is found almost instantaneously.

3.5.4. Grid graphs

In the goal of comparing our method with the experimentations of Potluri and Negi [12], we evaluated our method on several Grid Graphs from 5×5 to 8×8. Table 4 summarizes the comparison with the algorithm of Liu and Song [10] and the *Intelligent Enumeration Algorithm* (IEA) of Anupama Potluri and Atul Negi [12]. We give their running time in seconds. Note that Anupama Potluri and Atul Negi execute their programs on a server having Intel(R) Xeon(TM) CPU 3.00 GHz dual processor quad core with 8GB RAM which is more powerful than our equipment.

We remind that our results are average values of 10 executions. Clearly IEA is faster than graph matching algorithm of Liu and Song because it does not go through all the enumerating solutions. It is so clear that our method is better to find $mIDS$ than the two others. While the IEA tests each solution to determine if it is or if it is not an IDS, we ignore a lot of non IDS solutions, thanks to our cuts. Then for 49 vertices, our algorithm takes 9.90 s running time while IEA takes more than one day.

TABLE 5. Results: Special Star Graph.

	$n = 21$	91	211	381	601
NbExec	10	10	10	10	10
Greedy_time	0.00	0.00	0.00	0.00	0.00
$\|Greedy\|$	16	81	196	361	576
Opt_time	0.00	0.00	3.70	227.00	8485.10
Opt_size	5	10	15	20	25
NbUpdate	4.80	9.90	14.30	19.60	24.70

3.5.5. Special star graphs

The Special Star Graph is a family of graphs we have found to trick the Greedy Heuristic (see Sect. 3.3.1). Table 5 reports results of experiments on these particular graphs. GH gives naturally very bad results. But the rest of our method is able to correct quickly this bad initial solution up to 381 vertices. The last case ($n = 601$) can be solved in less than 2h30.

3.5.6. Special two subsets graphs

Special Two Subsets Graph is another family of graphs on which the Greedy Heuristic gives very bad results (see Sect. 3.3.2). However our clique partition method is very efficient to improve this initial solution. It solves $mIDS$ problem quasi instantaneously (in less than one second in our experiments) on such graphs up to 901 vertices. This can be explained as follows: As the best solution is of size 2 and as it is quickly found, the Minimum Size cut (see Sect. 3.2) avoids all the branches of the searching process containing solutions larger than 2 (which means a lot of branches).

3.5.7. Analysis

To conclude on this, we can note that Greedy Heuristic is very fast (less than 1 s) and very efficient (often close to the optimal) except on some specific families of graphs specially designed in that matter. In the following section, we will analyze its effectiveness on large graphs. A second point to note is that our cuts are very efficient; They allow us to construct optimal solutions in reasonable time for graphs up to 75 vertices and in at most a few hours for graphs with 150 vertices. But we also note that for some large sparse graphs our method could not be applied (too long). Moreover on Grid Graphs which is a (sparse) classical topology in communication and network optimization, we obtained much better results than two other recent methods experimented in the literature.

4. GREEDY HEURISTIC APPROACH AND LOWER BOUND FOR THE $mIDS$

We have previously shown that our exact algorithm was able to find optimal solutions but when the size of the graph increases the processing time becomes

too high. To try to treat graphs of large size we deeper analyze in this section the Greedy Heuristic (see Sect. 3.3; we remind that GH was used to find an initial solution in our exact method) to quickly construct an IDS and try to evaluate its quality. For that, it would be useful to compare this IDS to the optimal one. But as this is not possible, we compare it here to a lower bound of the optimal.

In Section 4.1 we present our (polynomial) algorithm LWB that computes a lower bound for the $mIDS$ problem. In Section 4.1.1 we *prove* that LWB calculates a lower bound of $mIDS$ problem. In Section 4.1.2 we describe the worst behavior of LWB.

In the following section our goal is to evaluate solutions produced by GH using the lower bound from LWB algorithm. We have proved in Sections 3.3.1 and 3.3.2 that there exist families of graphs for which the greedy solution can be very large compared to optimal. However we experimentally show in Section 4.2 that in practice/average they are close, even for very large graphs, meaning that GH is an interesting algorithm to construct IDS of pretty good quality in graphs of large size.

4.1. Lower bound

In this subsection we give a polynomial algorithm that computes a lower bound for the $mIDS$ problem. Before that, we need to define some preliminary notations. We remind that $G = (V, E)$ is any graph and $C = \{C_1, \ldots, C_k\}$ is any clique partition of G. We denote by $d(C_i)$ the maximum degree of vertices in C_i: $d(C_i) = max\{d(u), u \in C_i\}$. We call *degenerative clique partition*, the particular clique partition of G in which each clique is reduced to only one vertex.

We describe now our algorithm called *LWB*. Without loss of generality we suppose $C = \{C_1, \ldots, C_k\}$ is sorted in decreasing order of maximal degree: $d(C_1) \geq \ldots \geq d(C_k)$. LWB builds its solution by adding successively one vertex of maximal degree u_i of C_i ($d(u_i) = d(C_i)$) in the natural order $(1, 2, \ldots)$ in a set S initially empty. LWB stops when $\sum_{u \in S} d(u) \geq |V| - |S|$. We denote by $S_p = \{u_1, \ldots, u_p\}$ (with $p \leq k$) the solution returned by the algorithm (we do *not* assert that S_p is a $mIDS$). LWB is clearly polynomial.

4.1.1. LWB *Gives a Lower Bound*

Theorem 4.4 proves that $|S_p|$ is a lower bound of $mIDS$ problem in G.

Property 4.1. *A set $S \subseteq V$ verifies property 4.1, if:*

1. $|S \cap C_i| \leq 1, \forall i \in \{1, \ldots, k\}$.
2. $\sum_{u \in S} d(u) \geq |V| - |S|$.

Lemma 4.2. *Let S be any dominating set of G. We have* $\sum_{u \in S} d(u) \geq |V| - |S|$.

Proof. Let $E_{out}(S)$ be the number of edges between S and $V\backslash S$, then we have $E_{out}(S) \leq \sum_{u \in S} d(u)$.

As S is a dominating set of G, there are at least $|V| - |S|$ edges between S and $V\backslash S$ and $|V| - |S| \leq E_{out}(S)$. Combining both inequalities the lemma follows. □

Lemma 4.3. *S_p is the smallest subset of V verifying Property 4.1.*

Proof. By construction of S_p with our algorithm LWB, there is at most one vertex in each C_i: $|S_p \cap C_i| \leq 1$. Moreover LWB stops when $\sum_{u \in S_p} d(u) \geq |V| - |S_p|$, thus S_p verifies Property 4.1.

We have $\sum_{i=1}^{p} d(C_i) \geq |V| - p$ and $\sum_{i=1}^{m} d(C_i) < |V| - m, \forall m < p$.

Suppose there exists a set $S = \{v_1, \ldots, v_m\}$, smaller than S_p ($|S| \leq |S_p|$) verifying Property 4.1. We denote by C_{v_i} the clique in C that contains v_i.

Then $\sum_{i=1}^{m} d(C_{v_i}) \geq |V| - m$ and thus: $\sum_{i=1}^{m} d(C_{v_i}) > \sum_{i=1}^{m} d(C_i)$.

This is a contradiction with the fact that C was sorted in decreasing order of maximal degree. Thus S_p is the smallest subset of V that verifies Property 4.1. □

Theorem 4.4. *Let S_{mIDS} be a solution of mIDS then $|S_{mIDS}| \geq |S_p|$.*

Proof. As S_{mIDS} is a dominating set, by Lemma 4.2 it verifies $\sum_{u \in S_{mIDS}} d(u) \geq |V| - |S_{mIDS}|$. Moreover as S_{mIDS} is independent $|S_{mIDS} \cap C_i| \leq 1, \forall i \in \{1, \ldots, k\}$. Hence S_{mIDS} verifies Property 4.1. However Lemma 4.3 proves that S_p is the smallest subset of V that respects Property 4.1, and hence $|S_{mIDS}| \geq |S_p|$. □

4.1.2. Degenerative clique partition Is always the worst

Theorem 4.5 proves that the degenerative clique partition (cliques are reduced to only one vertex) gives the worst lower bound with LWB.

Theorem 4.5. *The lower bound found by LWB with the degenerative clique partition is always smaller than or equal to the lower bound found with any other clique partition.*

Proof. Let C_g be the degenerative clique partition of G and $C_{g_i} = \{x_i\}, i = 1, \ldots, |V|$. W.l.o.g. we suppose that $d(x_1) \geq \ldots \geq d(x_{|V|})$. Now let $C = C_1, \ldots, C_m$ be any other sorted clique partition (non degenerative) of G with $m < |V|$ and $y_i, i = 1, \ldots, m$ a vertex in clique C_i such that $d(y_i) = d(C_i)$.

From the construction of the two previous clique partitions we have $d(y_i) \leq d(x_i), i = 1, \ldots, m$. Then the theorem is verified. □

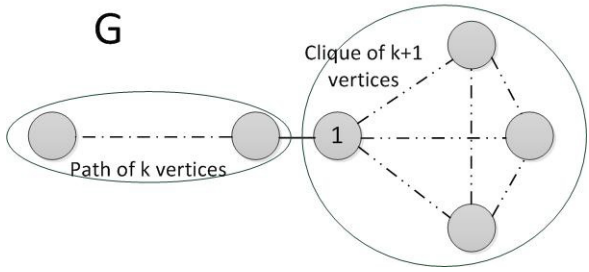

FIGURE 9. Lower Bound: Clique Partition.

In what follows we will use the graph G of Figure 9 to show that in some particular cases the difference between the lower bounds can be very large. Let k be an even not null integer. The graph has $2k+1$ vertices and is composed, at the left of vertex 1, of a path of k vertices and at its right, including 1 itself, of a clique of $k+1$ vertices.

Now consider a first clique partition $C1 = \{C1_1, \ldots, C1_{k/2+1}\}$ sorted in decreasing order of maximal degree, with $|C1_1| = k+1$ ($C1_1$ is the whole clique of $k+1$ vertices) and $|C1_i| = 2, \forall i \in \{2, \ldots, k/2+1\}$ ($k/2$ cliques of size 2 in the path of k vertices). We have $d(C1_1) = k+1$ and $d(C1_i) = 2, \forall i \in \{2, \ldots, k/2+1\}$. Algorithm LWB returns only one vertex of the big clique of $k+1$ vertices and $\lceil (k-1)/3 \rceil$ vertices of the other cliques of size 2. This leads to a solution of size $p = 1 + \lceil (k-1)/3 \rceil$ that gives a lower bound of $mIDS$ with this clique partition $C1$.

Now consider a degenerative clique partition $C2$ in which each vertex is a clique (there are $2k+1$ such cliques). $C2$ is sorted in decreasing order of maximal degree. This time, algorithm LWB returns a solution containing only two vertices of the big clique: $S_2 = \{u_1, u_2\}$ with $d(u_1) = k+1$ and $d(u_2) = k$ then the lower bound of $mIDS$ with the clique partition $C2$ being only 2.

The lower bound of the first clique partition $C1$ depends on k while the lower bound of the second clique partition $C2$ is a constant. This example shows how the choice of the clique partition can impact the quality of our lower bound.

4.2. Computational evaluation

In this section we present a summary of the key aspect of our different computational results with Greedy Heuristic (GH) and Lower Bound algorithm (LWB). We have implemented our algorithms in C language. We have evaluated several large graphs from 200 vertices to 60 000 vertices.

As in Section 3 each instance is executed 10 times on an AMD Opteron Processor 2352 clocked at 2.1 GHz.

In the following tables we present our experimental results. The number of vertices of the graphs is denoted by n. Each line contains: The size of the solution constructed by the GH "$|Greedy|$"; The lower bound with degenerative clique

TABLE 6. Results: Random Graphs with 10 000 vertices.

Prob	0.001	0.005	0.01	0.05	0.1	0.5
$\|Greedy\|$	1774	559	338	93	47	10
$\|LWBV\|$	558	144	77	18	10	2
$\|LWBC\|$	558.7	144	77	18	10	2
Std. Dev.	0.46	0	0	0	0	0

TABLE 7. Results: Random Graphs With Probability 0.001.

	$n = 5 \times 10^2$	10^3	2×10^3	4×10^3	6×10^3	8×10^3
$\|Greedy\|$	383	626	942	1239	1499	1636
$\|LWBV\|$	238	307	385	459	502	541
$\|LWBC\|$	326.8	338.2	391.5	460.6	503	542.3
Std. Dev.	0.6	2.89	1.02	1.02	0	0.46

TABLE 8. Results: Random Graphs With Probability 0.001.

	$n = 10 \times 10^3$	20×10^3	30×10^3	40×10^3	50×10^3	60×10^3
$\|Greedy\|$	1774	2214	2443	2659	2811	2955
$\|LWBV\|$	558	631	670	698	716	732
$\|LWBC\|$	558.7	631	671	698	716	732
Std. Dev.	0.46	0	0	0	0	0

partition "$|LWBV|$"; The average size of the lower bounds with random clique partitions "$|LWBC|$" and its standard deviation "*Std. Dev*". Note that *average size* and *standard deviation* are calculated from 10 executions of the same instance: Thus we obtain 10 different solutions with 10 different clique partitions.

4.2.1. Random graphs

In Section 3 we have defined the classical random graphs model that we use here. As it was said previously GH is very fast for all of our experimentations. On several 10 000–vertices graphs, the running times do not exceed 1 s.

Table 6 reports statistics for graphs with 10 000 vertices. As expected size of GH rises when probability is small. The $|LWBV|$ and $|LWBC|$ are often equal (more than 90% of tested graphs) and standard deviations are tiny: reaching 0.46 for probability 0.001. Tables 7 and 8 report statistics for random graphs with probability 0.001 with 500 vertices to 60 000 vertices. GH remains efficient when compared to the lower bound. We can see also that for small and sparse graphs LWB with clique partition is clearly better than LWB used with degenerative clique partition. However the difference is marginal when n is large.

TABLE 9. Results: Random Tree Graphs.

	$n = 10 \times 10^3$	20×10^3	30×10^3	40×10^3
$\|Greedy\|$	4065	8078	12 153	16 156
$\|LWBV\|$	2164	4314	6467	8614
$\|LWBC\|$	2175	4337.6	6500.8	8657
Std. Dev.	1.18	2.42	2.6	2.39

TABLE 10. Results: Grid Graphs.

	$n = 10 \times 10^3$	20×10^3	30×10^3	40×10^3
Dim.	100×100	100×200	100×300	100×400
$\|Greedy\|$	3303	6604	9954	13 303
$\|LWBV\|$	2000	4000	6000	8000
$\|LWBC\|$	2000	4000	6000	8000
Std. Dev.	0	0	0	0

TABLE 11. Results: Grid Graphs.

	$n = 10 \times 10^3$	20×10^3	30×10^3	40×10^3
Dim.	10×1000	10×2000	10×3000	10×4000
$\|Greedy\|$	3333	6666	10 000	13 333
$\|LWBV\|$	2000	4000	6000	8000
$\|LWBC\|$	2000	4000	6000	8000
Std. Dev.	0	0	0	0

4.2.2. Random Tree Graphs

According to the previous results, the standard deviation is small when G has low density. Thus we now test our software on Random Tree Graph.

Table 9 reports results for trees with 10 000 vertices to 40 000 vertices. For each execution the running time is less than 20 s. Standard deviation is tiny and we notice that GH gives solutions close (factor at most 2) from lower bound.

4.2.3. Grid Graphs

Tables 10 and 11 report results for different kinds of grid graphs. The running time is always less than 20 s. As the previous graphs we can see that GH gives good solutions when compared to the lower bound. Moreover LWB with any type of clique partition give exactly the same results because grid graphs are almost regular graphs.

4.2.4. Hypercube Graphs

As seen before, GH seems to be efficient on sparse and regular graphs. Then it is interesting to apply GH and LWB algorithm on hypercube graphs (see Sect. 3.5.3

TABLE 12. Results: Hypercube Graphs.

	$n = 8$	16	32	64	128	256	32768	65536
$\|Greedy\|$	2	4	8	16	16	32	2048	4096
$\|Opt.\|$	2	4	8	12	16	32	2048	4096
$\|LWBV\|$	2	4	6	10	16	29	2048	3856
$\|LWBC\|$	2	4	6	10	16	29	2048	3856
Std. Dev.	0	0	0	0	0	0	0	0

for definition). In 1993, Harary and Livingston [5] proved that for $d = 2^k - 1$ or $d = 2^k$ (k integer) the independent dominating number is 2^{d-k}. Thus we can now compare our methods with the optimal solution.

Table 12 reports results for different sizes of hypercube graphs for which we can compute or calculate the size of the optimal. We show that GH and also LWB algorithm are efficient: GH finds 7 of 8 optimal solutions and LWB gives 5 of 8 optimal solutions. The running times do not exceed 11 s for graphs with more than 65000 vertices.

4.2.5. Analysis

With these tests, we have confirmed that Greedy Heuristic is very fast: All our executions take less than 20 s of computational time. Experiences show that solutions produced by Greedy Heuristic do not exceed 5 times the lower bound which means they are within a factor at most 5 of an optimal solution. We have also seen that the lower bounds do not change very much for the same graph: The standard deviation of 10 executions on the same graph with 10 different clique partitions does not exceed 1%.

5. CONCLUSION AND PERSPECTIVES

The Minimum Independent Dominating Set problem is one of the hardest optimization problems since it is NP−Hard and cannot even be approximated with a factor better than $n^{1-\epsilon}$ with $\epsilon > 0$ in polynomial time, unless $P = NP$ (see [4]). In this paper, we proposed an exact algorithm to solve it in any graph G. We did a worst case analysis and gave conditions by which our method has lower complexity than other ones (in particular [2, 10]).

In the perspective to have a practical tool running on any graph, we implemented our generic method by adding cuts that avoid to explore useless branches of the searching tree. We experimented our software on several families of graphs. The conclusion is that we are able to solve the problem on graphs with up to 50 vertices (and even more, up to several hundreds vertices for some of them) in reasonable time. Moreover, we have compared our approach with experiments done on two other algorithms on Grid Graphs: Results show that our method is much better on the studied cases. We have also shown that the Greedy Heuristic

can handle large graphs and constructs solutions close from the optimal (in experiments, the ratio between the greedy solution and the lower bound was less than 5). The fact that GH is a good algorithm was already conjectured; it is confirmed in most of our experiments but we however proved with analytical arguments that for some particular graphs it can be very bad.

The non polynomial part of the complexity of our exact algorithm depends on the size of the cliques in a clique partition of G (that can be constructed greedily in a first phase). We made the observation that the order of the cliques and the order of the vertices in each clique can have an impact on the duration of the execution. Our goal is now to find these best orders in a pre−processing phase, before using them in our method.

A final perspective is to adapt our general method to other independent or dominating problems.

References

[1] L. Baez-Duarte, Hardy–ramanujan's asymptotic formula for partitions and the central limit theorem. *Adv. Math.* **125** (1997) 114–120.

[2] N. Bourgeois, B. Escoffier and V.T. Paschos, Fast algorithm for min independent dominating set. *SIROCCO, Lect. Notes Comput. Sci.* **6058** (2010) 247–261.

[3] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP–Completeness (Series of Books in the Mathematical Sciences)*. W.H. Freeman and Co Ltd, first edition edition (1979).

[4] M.M. Halldórsson. Approximating the minimum maximal independence number. *Inf. Process. Lett.* **46** (1993) 169–172.

[5] F. Harary and M. Livingston, Independent domination in hypercubes. *Appl. Math. Lett.* **6** (1993) 27–28.

[6] J. Haviland, Independent domination in triangle–free graphs. *Discrete Math.* **308** (2008) 3545–3550.

[7] D.S. Johnson, C.H. Papadimitriou and M. Yannakakis, On generating all maximal independent sets. *Inf. Process. Lett.* **27** (1988) 119–123.

[8] F. Kuhn, T. Nieberg, T. Moscibroda and R. Wattenhofer, Local approximation schemes for ad hoc and sensor networks. *DIALM-POMC* (2005) 97–103.

[9] J. Little, *Branch and Bound Methods for Combinatorial Problems*. Ulan Press (2012).

[10] C. Liu and Y. Song, Exact algorithms for finding the minimum independent dominating set in graphs. *ISAAC, Lect. Notes Comput. Sci.* **4288** (2006) 439–448.

[11] Y.L. Orlovich, V.S. Gordon and D. de Werra, On the inapproximability of independent domination in $2p_3$–free perfect graphs. *Theor. Comput. Sci.* **410** (2009) 977–982.

[12] A. Potluri and A. Negi, Some observations on algorithms for computing minimum independent dominating set. in Springer *IC3, Commun. Comput. Inf. Sci.* **168** (2011) 57–68.

[13] W.C. Shiu, X.-G. Chen and W.H. Chan, Triangle–free graphs with large independent domination number. *Discrete Optim.* **7** (2010) 86–92.

[14] Y. Song, T. Liu and K. Xu, Independent domination on tree convex bipartite graphs, in *Frontiers in Algorithmics and Algorithmic Aspects in Information and Management*, edited by J. Snoeyink, P. Lu, K. Su and L. Wang. Springer Berlin Heidelberg, *Lect. Notes Comput. Sci.* **7285** (2012) 129–138.

[15] J. Steele, *The Cauchy–Schwarz master class: an introduction to the art of mathematical inequalities*. MAA problem books series. Cambridge University Press (2004).