Rapport de projet :
Partial Independant Dominating set with Obligations in graphs

Thomas Ekindy - Paul Nautré
Encadré par Christian Laforest
PREP'ISIMA II - 2020/2021

25 mai 2021

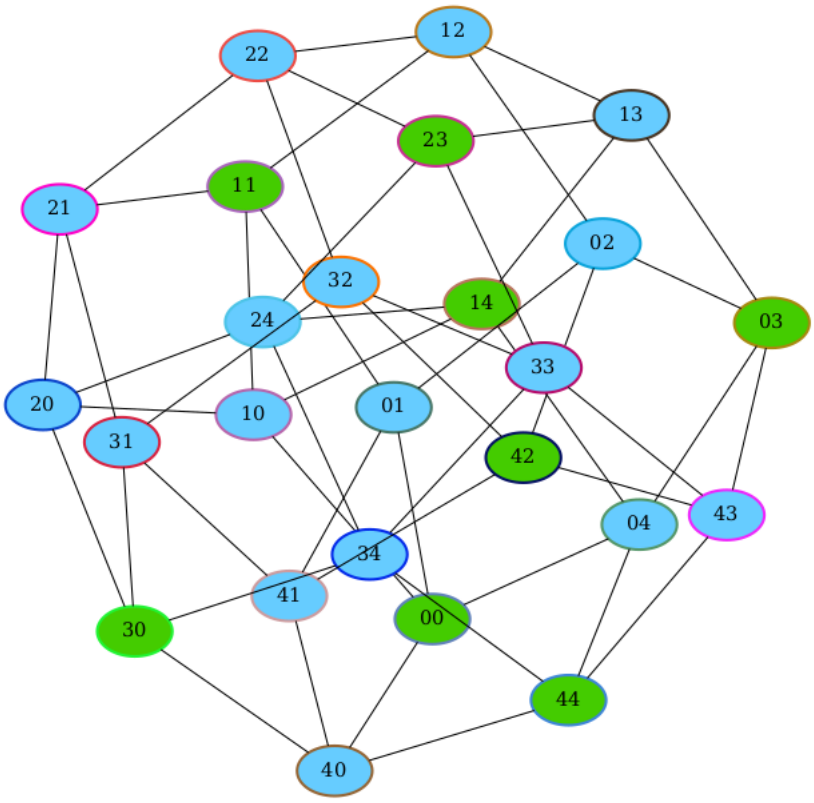

Table des matières

1 Introduction du sujet

1.1 PIDO

On rappelle qu'un **graphe** G est le couple d'un ensemble de **sommets** V et d'un ensemble d'**arêtes** E. Pour chaque couple de sommets (v_1, v_2) avec $v_1, v_2 \in V$, il existe ou non une unique arête $e \in E$ reliant v_1 et v_2 (on dit que les sommets sont **voisins**).
Cet objet mathématique, dans le principe très simple, est un puissant outils de modélisation, aussi bien utilisé dans la simulation nucléaire que dans l'analyse musicale [1].

Il existe de nombreux problèmes autour de la théorie des graphes. Une partie d'entre eux sont dit **NP complets**, c'est à dire que l'on ne connaît pas d'algorithme qui trouve à coup sûr une solution exacte dans n'importe quel cas. Pour ces problèmes, on développe alors des algorithmes de **résolution partielle**, qui donne des solutions incomplètes mais suffisamment proches des solutions exactes pour être exploitées.

Parmi ces problèmes, on a celui de la **domination de graphe** : on dispose d'un graphe et on veut colorer un sous ensemble d'au plus k de ses sommets, non voisins deux à deux, tel que chaque sommet du graphe soit **couvert**, c'est à dire :
— soit coloré (**dominant**)
— soit voisin d'un sommet coloré (**dominé**)
(k est un entier compris entre 0 et la taille du graphe).
Ce problème étant NP complet, on va développer des algorithmes qui coloreront une partie du graphe de sorte à dominer un maximum de sommets tout en minimisant tant que possible le nombre de sommets dominants.
C. Laforest a proposé une variante de ce problème : en plus du graphe, on dispose maintenant d'un **ensemble d'obligations**, un partitionnement **stable** des sommets du graphes : chaque sommet du graphe est affecté à un et un seul sous ensemble (appelé une obligation) tel qu'il n'est voisin d'aucun sommet de ce même sous ensemble. On cherchera maintenant une **configuration dominante** tel que si un sommet appartient à l'ensemble des sommets dominants, alors tous les sommets de son obligation sont dominants. Il en découle que si un sommet n'est pas dominant, c'est qu'aucun des sommets de son obligation ne l'est.
Ce nouveau problème est en fait équivalent au premier, avec une contrainte additionnelle. Il a été démontré par C. Laforest qu'il était également NP complet. On aura donc des algorithmes de résolution partielle. C. Laforest en a justement proposé un dans un article [2] publié avec T. Martinod. Ce sera notre base de travail tout au long de ce projet.

1.2 Objectifs

Dans un premier temps nous chercherons à **comprendre** et à **implémenter** l'algorithme mentionné précédemment. Cela implique de développer quelques outils pour manipuler des instances qui seront, rappelons le, des couples graphe + ensemble d'obli-

1. Tonnetz harmonic analysis : `https://www.youtube.com/watch?v=NQ7LkWCzKxI`
2. On the complexity of Independent Dominating Set with Obligations in graphs : `https://hal.archives-ouvertes.fr/hal-02946979v2`

gations.

Dans un second temps nous mettrons en œuvre cet algorithme et nous l'évaluerons. Pour cela, il faudra être capable de proposer une grande variété d'**instances** (donc de les **générer**) et de sélectionner et d'extraire des **indicateurs de qualité** des résultats obtenus.

Il est assez naturel, pour évaluer un objet ou une personne, de le mettre en concurrence avec d'autres objets ou personnes effectuant la même tâche. Nous proposerons donc à notre tour des **variantes de l'algorithme** étudié pour afin de mieux situer ses performances et peut-être de trouver une meilleure alternative (dans certains cas du moins).

L'étape finale de ce projet sera la rédaction de ce rapport et une soutenance orale.

2 Méthodes de travail

2.1 Outils

Nous travaillons avec le **langage Python**, selon la volonté de notre tuteur. Nous utilisons les bibliothèques **graphiz** et **matplotlib** pour nos outils de visualisation et les bibliothèques **math** et **random** en complément des outils natifs du langage. En dehors des types objets déjà disponibles, nous nous tenons au paradigme de **programmation impérative** de sorte à fluidifier le passage de pseudo code à Python.
Enfin, afin de faciliter le travail en binôme, nous utilisons l'outils de gestion de version et de collaboration **GitHub**[3].

3 Instances

3.1 Généralités sur nos générateurs

Dans le but d'assurer la comptabilité de nos algorithmes de résolution du problème PIDO avec nos **générateurs d'instance**, nous avons mis en vigueur des **normes de fonctionnement** pour ces derniers en ce qui concerne la façon de représenter les informations. En excluant la possibilité d'user du principe de la programmation orientée objet, il a été établi que le meilleur moyen de représenter un **graphe** était d'utiliser un **dictionnaire** dans lequel chaque **sommet** serait une **clé** qui aura pour **valeur** l'**ensemble de ses sommets voisins**. Quant aux **obligations**, même si elles sont un ensemble d'ensembles, elles ne seront pas modélisées par un `set()` d'ensembles mais plutôt par une `list()` d'ensembles, ainsi nous aurons accès à la fonction `shuffle()` pour mélanger les obligations et donc changer leur ordre de lecture, ce qui se trouve être une étape importante de la génération d'obligations.
Cette **unicité** dans le principe de fonctionnement des générateurs nous a permis de créer des fonctions facilitant la **manipulation des graphes** : `addEdge()`, `createGraph()`, `addVertex()` et `deleteVertex()`.

3. Dépôt du projet : `https://github.com/loremipsumdsa/PIDO`

Il est à noter que chaque **générateur** de graphe prendra en **paramètre** un **intervalle** auquel son **nombre d'obligation**, choisi aléatoirement avant la génération, devra appartenir ainsi que d'**autres paramètres**, variant selon les générateurs, donnant des instructions sur les **intervalles** dans lesquels doivent se trouver le nombre de **sommet** et le nombre d'**arrête** du graphe.
Enfin, en plus de leur graphe, les générateurs de graphe retourneront également une liste d'obligations qui sera, elle, produite par le générateur d'obligations.

3.2 Graphe aléatoire

Utilité Le premier générateur que nous avions pensé à faire était un générateur de **graphe aléatoire** car évaluer la dominance de nos algorithmes sur ce dernier était la manière la plus **objective** de les comparer entre eux. Nous avons réalisé deux générateurs de ce type d'instance, l'un est complètement aléatoire tandis que l'autre est obligatoirement **connexe**, c'est à dire que tous ses sommets sont liés à au moins un autre sommet. Sur ce type de problème, avoir un ou plusieurs sommets isolés peut être à la fois une contrainte et un avantage, nous avons trouvé utile de bien différencier ces types de graphe afin de **généraliser notre étude**.
Nous avons par ailleurs programmer un générateur d'instance aléatoire qui choisis aléatoirement un type d'instance à générer, cependant nous ne lui avons pas trouvé de grande utilité.

Algorithme de graphe aléatoire Ce générateur complètement aléatoire initialise un graphe avec un nombre de sommet aléatoire compris dans l'intervalle fournis par l'utilisateur. Il défini un nombre d'arrête de la même façon, puis, autant de fois qu'il le faudra pour atteindre ce nombre, il ajoutera des arrêtes entre deux sommets différents et qui ne sont pas déjà voisins.
Cet algorithme nous permet donc d'obtenir un graphe aléatoire.

Algorithme de graph aléatoire connexe Il fonctionne dans sa globalité de la même manière que l'algorithme précédent sauf au moment de l'initialisation du graphe. En effet, en plus de rajouter des sommets au graphe, il lie chacun d'entre eux à un autre sommet aléatoire du graphe de manière à ce que tous les sommets de ce dernier aient une arrête qui les lie à au moins un autre et que le graphe soit donc connexe.

3.3 Graphe complet

Utilité Le **graphe complet**, contrairement aux autres, fait office de graphe de **test**. C'est un graphe dans lequel chaque sommet est lié par une arrête à tous les autres sommets. En toute logique, tous les sommets dominent l'entièreté du graphe donc l'évaluation de chacun de nos algorithmes avec ce type de graphe doit être identique. Etant donnée sa nature, évaluer les performances de nos algorithmes sur ce dernier permettait avant tout de **vérifier le bon fonctionnement** de nos fonctions de génération d'obligation et d'évaluation.

Algorithme Le principe de fonctionnement de l'algorithme de génération de **graphe complet** est très simple. Il consiste à ajouter autant de sommet qu'il le faut dans le graphe, et pour chacun d'ente eux on ajoute des arrêtes qui les lient à tous les autres sommets du graphe.

3.4 Graphes complexes : Grid, Torus et Hypercube

Utilité Un algorithme de résolution du problème PIDO peut être meilleur qu'un autre en moyenne mais il peut très bien exister un type de graphe pour lequel il sera moins bon que d'autres. Cette réflexion nous a mené à produire trois générateurs de graphes très particuliers.

- Le graphe Grid, comme son nom l'indique, est un type de graphe ressemblant à une grille.
- Le graphe Torus est également une grille mais avec une particularité bien à lui, les sommets en bordure de grille sont liés aux sommets qui se trouvent à la bordure opposée mais toujours sur la même ligne ou sur la même colonne.
- Le graphe Hypercube qui, à chaque fois que sa dimension augmente de 1, se voit ajouté un graphe similaire à celui de la dimension précédente auquel il sera lié par des arrêtes allant d'un sommet à un autre identique.

Pour un même nombre de sommet chacun de ces trois graphes gardera la même apparence contrairement aux graphes générés aléatoirement. Par exemple un graphe hypercube de dimension n sera structuré de la même façon que n'importe quel autre graphe hypercube de taille n. Cela ne veut pas pour autant dire que pour chaque nombre de sommet nous n'aurons qu'un seul cas de graphe à étudier. Il existe toujours énormément de possibilités de distribution des obligations donc il reste tout aussi important d'évaluer nos algorithmes de résolution sur un grand nombre d'instance pour ces trois types de graphe que pour des graphes aléatoires.

Algorithme graphe Grid La fonction qui génère ce type de graphe prend en paramètre des intervalles de taille de colonne et de taille de ligne afin de parcourir le graphe de la même manière qu'on parcourt un tableau.
L'algorithme va d'abord parcourir les lignes et les colonnes entrées par l'utilisateur avec deux boucles imbriquées et pour chaque combinaison de ligne et de colonne ajouter un nouveau sommet dont le nom dépendra de sa position dans la grille. On parcourt une seconde fois la grille mais cette fois en reliant par des arrêtes les sommets censés l'être, pour cela on s'aide de la nomenclature dépendante de la position du sommet. On obtient donc un graphe Grid en forme de grille.

Algorithme graphe Torus L'algorithme de génération du graphe Torus diffère de celui du graphe Grid en ajoutant les arrêtes entre les sommets en bordure et leur sommet opposé. Les sommets situés dans les coins de la grille ont deux sommets opposés d'où l'utilisation des `if` et non pas des `elif` pour vérifier leur position par rapport aux bordures.

Algorithme graphe Hypercube Il existe une méthode très simple à mettre en place permettant de générer un graph hypercube.
Considérons tout d'abord que chaque sommet de l'hypercube est nommé en binaire tel que 001101 soit un sommet du graphe hypercube de dimension 6. On peut alors dire que dans un graphe hypercube, deux sommets sont voisins si et seulement si leurs noms diffèrent d'1 seul bit, par exemple 0101 et 0100 seraient voisins dans le graphe hypercube de dimension 4.
La fonction correspondant à cette algorithme prend en paramètre une dimension (n) pour générer un graphe de 2^n sommets. Pour chaque sommet on va, avant de l'ajouter au graphe, convertir son numéro en binaire puis rajouter des 0 au début pour qu'il ait un nom respectant la nomenclature décrite précédemment. Ensuite, pour chaque bit i dans le nom du sommet, l'algorithme ajoutera une arrête allant du sommet jusqu'à un autre sommet avec qui il aura comme seule différence au niveau du nom le bit i.

3.5 Extracteur et exportateur d'instance

utilité Au cours du projet nous avons eu l'idée de développer un extracteur et un exportateur d'instance de sorte à être capable de traiter des instances générées par des personnes utilisant les mêmes normes que nous mais également pour faciliter le débogage de nos codes. Quand un résultat nous paraissait incohérent il nous suffisait de déboguer notre code et de l'exécuter à nouveau avec la même instance sauvegardée et rechargée. Si aucun chargement ni sauvegarde n'était possible nous n'aurions pas pu réaliser l'impact des changements dans nos codes de façon précise.
Pour qu'un fichier soit lu par l'extracteur d'instance il devra avoir la même structure que l'exemple ci-dessous :
a :b
b :a :e
c :b :d
e :b
d :c

a,c
e,d
b

Le premier élément de chaque ligne correspond à un sommet et les suivants sont ses voisins. Chaque ligne qui suit le saut de ligne correspond à une obligation, ses éléments sont donc séparés par des virgules. Dans le cas où la fonction d'extraction ne trouve pas d'obligations il en générera.

Algorithme extracteur d'instance L'algorithme extracteur d'instance va lire chaque ligne d'un fichier. Tant qu'il ne rencontre pas de saut de ligne, il va peu à peu remplir le graphe en y ajoutant les sommets et les arrêtes en suivant les instructions du fichier. Une fois un saut de ligne rencontré, il ne va rien modifier au graph ni aux obligations et attendre de passer à la ligne suivante pour ajouter cette fois les obligations à l'instance. Dans le cas où la liste des obligations est vide, l'extracteur va faire appel à la fonction de

génération d'obligation pour compléter l'instance.

Algorithme exportateur d'instance Cet algorithme va donc faire totalement l'inverse de ce que fait l'algorithme extracteur d'instance. Il prendra en entrée une instance et écrira dans le fichier chaque sommets de son graphe ainsi que ses voisins en respectant la relation clé/valeur dans le dictionnaire python. Lorsque toutes les clés seront lues, la fonction ferait un saut de ligne et écrira, pour chaque obligation, tous ses éléments sur une ligne.

3.6 Générateur d'obligations

Utilité La fonction de génération d'obligations permet de générer complètement aléatoirement un ensemble d'obligations stables en prenant en entrée un graphe. Cette fonction est utilisée à la fin de chacun des algorithmes de génération de graphe pour leur produire un ensemble d'obligations. Nous aurions pu donner à chaque fonction de génération de graphe sa propre fonction de génération d'obligations, cela aurait rendu les exécutions de nos tests bien plus courtes car il existe des implémentations spécifiques à chaque type de graphe pour leur générer un ensemble d'obligations de manière optimisée. Cependant elle est extrêmement utile car elle joue un grand rôle dans la polyvalence de notre programme. Elle nous permet de pouvoir développer quand nous le voulons un générateur de graphe sans avoir à nous préoccuper de refaire une fonction de génération d'obligation spécifiquement pour celui-ci.

Algorithme de génération d'obligations L'algorithme prend en paramètre un graphe et va parcourir chacun de ses sommets et pour chacun d'entre eux il va mélanger sa lise d'obligations crée au préalable pour maximiser l'aléa. Dans le cas où la liste d'obligations n'aurait pas été mélangé on aurait eu plus de chance de remplir les premières obligations que les dernières, ce qui ne convient pas à notre contrainte d'aléa. On parcourt les obligations jusqu'à en trouver une qui ne contienne aucun voisin du sommet que nous sommes en train de traiter. Une fois cela fait, nous ajoutons ce sommet dans l'obligation, et attaquons l'itération suivante de la première boucle. Dans le cas où le nombre d'obligation serait insuffisant pour stocker tous les sommets tout en restant stables on se permet d'en augmenter le nombre.
Une fois que tous les sommets sont bien réparties dans les obligations on supprime celles qui sont vides et on vérifie que le nombre d'obligations respecte bien la contrainte initiale donnée en paramètre, si ce n'est pas le cas on renvoi un message d'erreur en expliquant que la création d'obligation n'a pas pu être faite.
Dans le cas où le nombre d'obligations est égal à celui de la contrainte on renvoi la liste d'obligations générée.

4 Algorithmes

4.1 Algorithme initial

Algorithme L'ensemble du projet est construit autour d'un algorithme de résolution partiel proposé par C. Laforest. Voici L'algorithme, tel qu'il est décrit dans l'article :

Algorithme 1 : PIDO par C. Laforest

Soit G un graphe et OS un ensemble d'obligations sur les sommets de G;
Créer une solution vide S;
tant que *obligations restantes dans* OS **faire**
  Ajouter la plus grande obligation B dans S;
  Soit $N(B)$ l'ensemble des sommets qui sont voisins d'au moins un sommet de B;
  Soit $O(B)$ l'ensemble des sommets contenus dans les obligations contenant au moins un sommet de B ;
  Supprimer dans G tout les sommets de B et de $O(B)$;
  Supprimer dans OS l'obligation B ainsi que toutes les obligations des sommets de $N(B)$;
fin
Retourner S;

Pour mieux comprendre la notion de "plus grande obligation", voici l'algorithme qui sélectionne B :

Algorithme 2 : Plus grande obligation

Soit G un graphe et OS un ensemble d'obligations sur les sommets de G;
B est la première obligation de OS;
pour *chaque obligation* O *dans* OS **faire**
  si O *contient plus de sommets que* B **alors**
    B devient O;
  fin
fin
Retourner B;

Implémentation Pour des raisons qui seront explicitées par la suite, nous avons choisi de séparer dans l'implémentation le sélecteur de l'obligation B du corps de l'algorithme. Nous avons donc une fonction `searchIDO()` qui reçoit une instance : un graphes sous la forme d'un dictionnaire et un ensemble d'obligation représenté par un tableau d'ensembles. Pour ne pas altérer l'instance, au cas ou elle serait réutilisée plus tard, la fonction va travailler sur une copie. Plusieurs fois au cours du traitement, `searchIDO()` appelle une autre fonction passé en argument, `selector()` et lui passe l'instance pour qu'elle lui renvoie une obligation à traiter. Dans le cas présent, `selector()` est `bigestObligation()` et renvoie la plus grande obligation de l'ensemble.

Commentaire Pour toute instance (G, OS), on obtiendra S, un sous ensemble de sommets du graphe respectant la contrainte additionnelle définie dans le chapitre précédent,

à savoir si un sommet appartient à S, alors tous les sommets appartenant à la même obligation sont dans S. Puisque qu'on suppose que toutes les obligations stables, on aura également S stable. En revanche rien n'assure que S soit une configuration dominante complète sur G. Il est possible qu'un sommet de G ne soit ni dans S, ni voisin d'un sommet de S.
Un étape qui caractérise cet algorithme est la sélection de l'obligation B. Ici, on choisi la plus grande obligation, celle qui contient le plus de sommets. On remarque que qu'en réalité quelque soit l'obligation sélectionnée, les assertions précédentes restent vraies. De là, nous comprenons que c'est en jouant sur ce choix, précisément sur le critère de sélection, que nous obtiendrons des variantes plus ou moins intéressantes de l'algorithme.

4.2 Première alternative : par couverture

Algorithme A partir de maintenant nous conserverons le corps de l'algorithme précédemment exposé et nous nous intéresserons exclusivement à la sélection de B. Jusque là, B était l'obligation contenant le plus de sommets. Nous avons eu l'intuition qu'il serait plus intéressant de comptabiliser l'ensemble des sommets **couverts** par l'obligation. A savoir, tout les sommets de l'obligation plus tout les voisins de chaque sommet de l'obligation qui ne sont pas déjà couvert (rappelons qu'on a une élimination des sommets déjà traités). Nous avons donc proposé le sélecteur suivant :

Algorithme 3 : Obligation la plus couvrante

```
Soit G un graphe et OS un ensemble d'obligations sur les sommets de G;
B est la première obligation de OS;
CB est un ensemble initialement vide;
CO est un ensemble initialement vide;
pour chaque obligation O dans OS faire
    pour chaque sommet S de O faire
        Ajouter S dans CO;
        Ajouter tous les voisins de S dans CO;
    fin
    si CO contient plus de sommets que CB alors
        B devient O;
        CB devient CO;
    fin
    Vider CO;
fin
Retourner B;
```

Implémentation Puisque nous avions précédemment séparé le corps de l'algorithme du sélecteur de B, nous n'avons qu'à écrire une nouvelle fonction de sélection `mostCoveringObligation()`, qui prend également l'ensemble des obligations et le graphe, et qui retourne l'obligation la plus couvrante. Nous pouvons maintenant utiliser ce nouveau sélecteur en le passant en paramètre à notre première fonction `searchIDO()`. Notons que cette fois ci, le graphe est bien nécessaire pour déterminer l'obligation la plus couvrante (l'ensemble d'obligation

seul n'est pas suffisant).

Commentaire Cette méthode de sélection implique plus de calculs que la première, car en plus de parcourir l'ensemble des obligations, On accède à chaque sommet de chaque obligation et on regarde ses voisins.
Il est important de se rappeler qu'un ensemble ne peut pas contenir deux fois un même élément. Ainsi, si un sommet est voisin de deux sommets d'une obligation O, il apparaît une seule fois dans CO. Enfin lorsqu'une obligation est supprimée par l'algorithme principale, tous ses sommets sont retirés du graphe. Par conséquent ici le sélecteur ne compte que les sommets qui ne sont pas encore couverts.

4.3 Deuxième alternative : par voisinage

Algorithme L'algorithme précédent comptait les sommets couverts par une obligation, soit la somme des sommets de cette obligation plus tous les voisins de ses sommets. Nous proposons donc une variante dans laquelle ne sont comptés que les sommets dominés par l'obligation, soit les voisins des sommets de l'obligation en question.

Algorithme 4 : Obligation la plus couvrante

Soit G un graphe et OS un ensemble d'obligations sur les sommets de G;
B est la première obligation de OS;
CB est un ensemble initialement vide;
CO est un ensemble initialement vide;
pour *chaque obligation O dans OS* **faire**
  pour *chaque sommet S de O* **faire**
    Ajouter tous les voisins de S dans CO;
  fin
  si *CO contient plus de sommets que CB* **alors**
    B devient O;
    CB devient CO;
  fin
  Vider CO;
fin
Retourner B;

Implémentation Notre nouvelle fonction de sélection `mostDominantObligation()` est à peu près un copier coller de la précédente, on retire simplement la ligne qui ajoute les sommets de l'obligation considérée dans l'ensemble construit.

Commentaire Notre intuition est que ce sélecteur devrait donner en général des solutions plus petite, c'est à dire utiliser moins de sommets dominants pour dominer un graphe. Supposons un cas critique, on a une instance avec un graphe étoile de taille $n > 2$(un sommet central, les $n-1$ autres sommets sont voisins uniquement de ce sommet central) et deux obligations, l'une contenant le central, l'autre tout les autres sommets. Le sélecteur de plus grande obligation choisira de couvrir tous le graphe en retenant comme

solution la deuxième obligation (donc $|S| = n - 1$). Notre nouveau sélecteur choisira la première obligation pour dominer également l'ensemble du graphe ($|S| = 1$)). On aurait donc deux solutions complètes mais la deuxième serait bien plus efficace. (remarquons que notre sélecteur par couverture n'aurait pas fait de différence entre les deux obligations puisque l'ensemble couvert est le même dans les deux cas).

4.4 Troisième alternative : par rapport de domination

Algorithme Nous avons proposé des algorithmes de sélection fonction de la taille des obligations et de la taille de leur voisinage ou bien de la somme des deux. Pour aller plus loin sur le problème d'efficacité, nous proposons donc une dernière alternative qui évalue le rapport des deux, c'est à dire qui choisi l'obligation dont le rapport taille du voisinage sur taille de l'obligation est maximum.

Algorithme 5 : Obligation ayant le meilleur rapport de domination

Soit G un graphe et OS un ensemble d'obligations sur les sommets de G;
B est la première obligation de OS;
CB est un ensemble initialement vide;
CO est un ensemble initialement vide;
pour *chaque obligation* O *dans* OS **faire**
  pour *chaque sommet* S *de* O **faire**
    Ajouter tous les voisins de S dans CO;
  fin
  si $\frac{|CO|}{|O|}$ *est plus grand que* $\frac{|CB|}{|B|}$ **alors**
    B devient O;
    CB devient CO;
  fin
  Vider CO;
fin
Retourner B;

Implémentation La fonction de sélection `mostDominatingObligation()` est cette fois encore sensiblement proche dans l'implémentation de nos fonctions précédentes. Nous pouvons indifféremment calculer le rapport de la taille de l'ensemble dominé ou de l'ensemble couvert par une obligation par sa taille. En effet si o est une obligation, d l'ensemble dominé et c l'ensemble couvert par o, on a $c = d + o$ donc $\frac{|c|}{|o|} = \frac{|d|}{|o|} + 1$.

Commentaire Avec ce sélecteur, nous ne nous attendons pas à nous rapprocher d'un calculateur de solutions complètes. Il est probable que les solutions retournées ne couvre qu'une petite partie du graphe. En revanche nous espérons déterminer des solutions localement très efficace, c'est-à-dire un sommet dominant doit dominer beaucoup de sommets.

4.5 Alternative témoin : par hasard

Algorithme Nous sommes dans une démarche de comparaison et d'évaluation. Nous voulons donc situer la valeur ajoutée de nos algorithmes, les uns par rapport aux autres mais aussi vérifier leur crédibilité, c'est à dire si il est intéressant de les utiliser où si on peut faire aussi bien sans.

Algorithme 6 : Obligation au hasard

Soit G un graphe et OS un ensemble d'obligations sur les sommets de G;
Retourner une obligation B prise au hasard dans OS;

Implémentation Ce dernier sélecteur `randomObligation()` tient en une ligne grâce à la méthode `choice()` de la bibliothèque `random`. Par ailleurs il n'a besoin en entré que de l'ensemble des obligations, le graphe n'est pas nécessaire (comme le premier).

Commentaire Ce sélecteur est minimal, il nécessite très peu de calcul. Notons que puisque les ensembles d'obligations ne sont pas ordonnées par notre générateur, on aurait pu choisir de renvoyer toujours l'obligation en position 0 dans le tableau par exemple. Nous avons cependant préféré utiliser une position pseudo aléatoire au cas où un ensemble d'obligation serait justement ordonnée, comme ce pourrait être le cas avec une instance extraite d'un fichier. On aurait alors un biais qui fausserait la valeur de témoin. Cette réflexion nous a inspiré le mode de sélection suivant.

4.6 Dernière alternative : sélection statique

Algorithme Jusque là, au cours du traitement, B était sélectionné en fonction de l'état courant de l'instance. Puisque l'algorithme modifiait l'instance (suppression des sommets couverts et des obligations partiellement couvertes), nous pouvions dire que la sélection de B était dynamique. Il nous vient donc l'idée que celle ci pourrait être statique : On aurait un ensemble d'obligations initialement ordonné selon un critère (par exemple la taille du voisinage de l'obligation). Au traitement, l'algorithme considérerait d'abord la première obligation de l'ensemble. Il est possible que le traitement de celle ci entraîne la suppression d'autres obligations, sans pour autant altérer l'ordre. Par la suite, l'algorithme considérera toujours l'obligation suivante dans l'ensemble, jusqu'à ce que celui ci soit totalement vidé.

Dans un premier temps on veut pouvoir trier un ensemble d'obligations selon l'un des

critères précédemment définis. Pour cela on propose un algorithme générique :

Algorithme 7 : Tri de l'ensemble d'obligation

Soit G un graphe et OS un ensemble d'obligations sur les sommets de G;
Soit S une fonction de sélection parmi celles définies précédemment;
OOS est une liste initialement vide;
tant que OS *n'est pas vide* **faire**
    $B = S(G, OS)$;
    Ajouter B à la suite de OOS;
    Supprimer B de OS;
fin
Retourner OOS;

Une fois l'ensemble trié, l'instance est traité classiquement par notre algorithme principal en faisant appel à un sélecteur qui retourne toujours le premier élément de l'ensemble d'obligations (cet élément est ensuite supprimé par l'algorithme principal, donc remplacé par le suivant).

Algorithme 8 : Obligation suivante

Soit G un graphe et OS un ensemble d'obligations ordonné sur les sommets de G;
Retourner la première obligation de OS

Implémentation Puisque pour des raisons pratique nous utilisions déjà des listes pour la représentation de nos ensembles d'obligations, Nous n'avons aucune difficulté à les trier en utilisant les fonctions de sélection. Nous avons pour cela implémenter la fonction `obligationsOrder()` qui reçoit une instance et un sélecteur et qui retourne l'ensemble des obligations trié.
Si `os` est une liste triée, `os[0]` est son premier élément. Maintenant si on supprime cet élément (`os.del(0)`), on retrouve en position 0 l'élément qui était avant en seconde position (la liste est utilisée comme une pile). On a donc une fonction `nextObligation()` qui reçoit une instance dont la liste d'obligations est ordonnée. On récupère la première, on la supprime de la liste et on la renvoie. C'est cette fonction donc qui jouera le rôle de sélecteur.

Commentaire Avant même de faire des tests, nous pouvons prévoir certains comportements de cette nouvelle méthode de sélection. D'abord nous comprenons que nous ne devrions pas remarquer de différence entre la sélection aléatoire statique et dynamique. En effet dans les deux cas, l'obligation retournée à chaque itération est choisie aléatoirement. De même, nous prévoyons une équivalence entre la sélection par plus grande obligation en statique ou en dynamique : en effet la taille des obligations n'évolue pas au cours du traitement. La différence se fera donc sur les sélections par voisinage, couverture et rapport de domination. Nous n'espérons pas de meilleurs résultats en statique qu'en dynamique, en fait nous pensons plutôt souligner l'importance de la prise en compte de l'évolution de l'instance. Enfin cette méthode, couplée à l'outils d'importation d'instance depuis un fichier, ouvre à de nouvelles possibilités.

5 Évaluation

5.1 Évaluation d'une solution

Critères d'évaluation Nous avons à présent de quoi générer un graphe et des algorithmes pour traiter celui-ci. Le problème qui se pose est : comment savoir si l'algorithme à bien fait son travail, c'est à dire comment évaluer la solution retournée. Dans un premier temps, nous vérifions si la solution est recevable, c'est-à-dire si elle respecte bien la contrainte d'obligations. Rappelons que la contrainte est : si un sommet appartient à la solution, alors tous les sommets de son obligation y sont aussi. L'algorithme proposé par C. Laforest respecte cette contrainte. Si nous venions à produire une solution irrecevable, ce serait donc une erreur d'implémentation. Après plusieurs batteries de test de recevabilité sans erreur, nous décidons de contourner cette étape de vérification en supposant que l'implémentation est correcte.

Puisque la solution est recevable, nous nous intéressons maintenant à sa qualité. L'objectif est de montrer que pour une même instance, tel algorithme a mieux répondu au problème que tel autre. Le problème étant comment maximiser la couverture du graphe tout en minimisant le nombre de sommets dominants, nous allons mesurer deux caractéristiques de la solution qui nous serviront de critères de qualité : La taille de l'ensemble des sommets de la solution et la taille de l'ensemble des sommets du graphe couverts par la solution. Afin de rendre ces chiffres plus parlants, nous les transformons en pourcentage. Nous obtenons alors un **taux dominant** et un **taux couvert** du graphe. À partir de ces deux valeurs, nous pouvons calculer un troisième critère : le rapport des deux, que nous nommons **coefficient de domination**. Ce rapport est donc proportionnel à la couverture et inversement-proportionnel à la taille de la solution.Enfin on peut déterminer si une solution est complète, si 100% des sommets du graphe sont couverts.

Pour une solution sur une instance données, nous avons maintenant un état de recevabilité, trois indicateurs numériques de qualité et un état de complétude. Nous pourrions décider de combiner ces critères pour obtenir une note globale. Cependant nous ne préférons pas, de la même façon que nous ne considérons pas que le coefficient de domination se substitue au couple taux de couverture et au taux dominant. A cet étape nous pensons que la qualité d'une solution est subjective et dépend des besoins de l'utilisateur. C'est à lui de décider l'équilibre acceptable entre maximisation de la couverture et minimisation de la solution.

Implémentation Nous avons deux fonctions, pour les deux étapes de l'évaluation, chacune prenant en paramètre un graphe et une solution.

La première, `checkSolution()`, vérifie la recevabilité de la solution. Pour cela elle en prend chaque obligation une par une et vérifie qu'il n'en existe pas dont l'intersection avec la solution soit non vide et la différence vide. Elle retourne `True` ou `False` selon le cas. Cette vérification est simple mais implique beaucoup de calcul, c'est pour cette raison que nous avons décidé de l'écarter une fois la fiabilité des algorithmes éprouvée.

Pour cela, la fonction `recevability_test()` teste un grands nombre de couple instance solution, respectivement généré avec tous les générateurs possibles et calculés avec tous les sélecteurs disponibles. Une seule exécution de cette fonction nous a permis de supposer que l'implémentation était correcte.

La seconde fonction, `evaluatePIDO()`, prend donc une instance et une solution (seul le graphe est important, l'ensemble d'obligation n'est pas nécessaire). Elle extrait nos indicateurs de qualité et les retourne dans un tuple à trois champs. Pour cela elle construit l'ensemble des sommets couverts, le mesure et le rapporte en pourcentage de la taille du graphe. elle mesure également la taille proportionnelle de la solution et le rapport de la taille de l'ensemble couvert par la solution. Le dernier champs du tuple est un booléen qui indique si la solution est complète ou nous (complet si tous les sommets du graphe sont couverts).

5.2 Évaluation d'un algorithme

Valeur statistique Nous somme maintenant capable d'évaluer une solution fournie par un algorithme pour une instance donnée. Nous savons qu'un essai sur une instance avec un algorithme nous donne très peu d'informations sur l'algorithme lui-même. Pour mesurer ses performances, il nous faut faire un grand nombre de tests avec un grand nombre d'instances, et idéalement, avoir une base de comparaison. La suite logique est de réaliser une mise en concurrences statistique des différents algorithmes.

Nous sommes capable de générer différents types d'instances (à graphes connexe, grid, torus...) et nous nous attendons à ce que les algorithmes soient plus ou moins performants selon les types. Nous générons donc un nombre n d'instances d'un type donné (n paramétrable), nous passons chacune de ces instances dans chacun des algorithmes, évaluons chaque solution et calculons une valeur moyenne pour chaque critère et pour chaque algorithme. Ici l'algorithme à sélection aléatoire prend tout son sens car ses scores nous serviront de socle de comparaison (un algorithme dont les indicateurs serait proche de celui ci pourrait être considéré comme médiocre). Puisque les entrées sont très variées, il faut beaucoup de cas pour obtenir des résultats exploitables. On considère que n est assez grand quand les chiffres ne varient pas (ou très peu) entre deux sessions. Nous voulons également savoir, sur n instances, combien chaque algorithme détermine de solutions complètes. nous avons finalement fixé $n = 30000$, ce qui donne des sessions de calcul d'environs 26h.

Enfin, pour bien comprendre les résultats, il est intéressant de disposer de données sur les instances utilisées pour le test. C'est pourquoi nous faisons suivre les relevés statistiques par un ensemble de métadonnées : Pour une session de test, il s'agit du type de graphes utilisées, du nombre moyen, minimum et maximum de sommets composants les graphes et du nombre moyen, minimum et maximum d'obligations associées.

Implémentation Nous avons voulu rendre la gestion des tests la plus flexible possible. Pour cela nous avons écrit une fonction `statisticCompare()` qui prend en paramètre n, le nombre d'échantillons à tester, un générateur d'instance et une liste de sélecteur à

comparer.

n fois, la fonction génère une instance et enregistre les métadonnées associées. Pour chaque sélecteur de la liste,elle détermine une solution, fait évaluer la solution et sauvegarde les scores. Une valeurs moyenne est déterminée pour chaque indicateur et pour chaque sélecteur. Enfin, on compte le nombre de solutions complètes.

Finalement, sont retournés :
— un dictionnaire, dont les clés sont les noms des sélecteurs et les valeurs sont des listes contenants chacun des trois scores moyens et le nombre de solutions complètes dans la session.
— un dictionnaire de métadonnées concernant les instances échantillons. Ce dernier donne le type des graphes utilisés, le nombre de sommets moyen, minimum et maximum et le nombre d'obligations moyen, minimum et maximum.

5.3 Visualisation

Nous avons une certaine quantité de données et nous comptons les exploiter. Pour nous simplifier le travail nous décidons donc de produire deux outils de visualisation.

Le premier nous permet de visualiser une solution sur une instance donnée. A l'aide de la bibliothèques `graphviz`, nous pouvons produire un rendu graphique répondant aux règles de coloration suivantes :
— Toute les sommets d'une même obligation apparaissent tracés d'une même couleur (choisie aléatoirement)
— Les sommets appartenant à la solution sont remplis en vert
— Les sommets dominés par la solution sont remplis en bleu
— Les sommets non couvert par la solution sont remplis en blanc
Cette outil nous a permis de vérifier que nos algorithmes avaient bien le comportement que nous pouvions prévoir sur de petites instances. En revanche il devient vite inutilisable sur les grands graphes pour lesquels l'affichage est très dense.

Le deuxième outil, beaucoup plus intéressant, nous assiste dans nos observations statistiques. Il s'agit d'une fonction qui prend les données produites par nos sessions de tests (le dictionnaire de statistiques et le dictionnaire de métadonnées) et les reporte sur un graphique en barre généré à l'aide de `Matplotlib`. On construit une barre pour chaque algorithme, donnant la part de sommets dominants, la part de sommet couverts et le rapport de domination ainsi que le nombre de solutions complètes trouvées dans la session. Le tout est légendé et le titre est construit automatiquement à partir des métadonnées. Les différences de performance entre les algorithmes deviennent ainsi remarquable au premier coup d'œil.
Les graphiques nous donne chaque fois un taille de solution moyenne s et un taux de couverture moyen c. Il faut bien comprendre que cela veut dire "en moyenne, on a des solution de taille s et en moyenne des couvertures moyenne de c" et non "dans le cas moyen, on a une solution de taille s qui couvre c. D'ailleurs on n'a pas d, le coefficient de domination, tel que $d = \frac{c}{s}$.

La structure du graphique est détaillée en annexe 2.

6 Résultats

6.1 sélection statique ou dynamique

Comme nous l'avions prévu, les sélecteurs aléatoire et par taille d'obligation se comportent de la même façon en statique ou en dynamique. En revanche contrairement à notre intuition, les taux de couvertures avec les autres sélecteurs sont meilleurs en statique qu'en dynamique. On observe un gonflement des chiffres, mais les coefficients de domination sont meilleurs en dynamique. En particulier le sélecteur par rapport de domination semble beaucoup moins efficace en statique (les résultats deviennent proches du sélecteur aléatoire). Enfin les tendances restent les mêmes dans les deux modes mais les écarts entre algorithmes sont amoindris en dynamique.
Le graphique détaillé 3 est en annexe.

6.2 Taille d'ensemble d'obligations

Nous avons fait deux types de session, avec des grands et des petits ensembles d'obligations. On remarque qu'on couvre beaucoup mieux les instances dotées d'obligations nombreuses (20% de différence avec le sélecteur par taille), mais on produit également des solutions plus grandes (10.4% de différence) et les coefficients de domination sont meilleurs avec de petit ensemble (à 0.6 point près). On pourrait d'ailleurs couvrir systématiquement tous les graphe en créant des obligations minimale, d'un seul sommets chacune (stabilité assurée). Cela revient à une recherche d'ensemble dominant sans minimisation. les tendances dans les deux cas sont toujours les mêmes et on conserve sensiblement les écarts de taux de couverture entre les différents algorithmes.
Le graphique détaillé 4 est en annexe.

6.3 Instances à graphe complet

Les graphes complets engendrent un comportement particulier sur nos algorithmes. Pour un graphe complet à n sommet, il existe un unique ensemble d'obligation possible : celui-ci contient n obligations, chacune de taille 1. Une seule de ces obligations (c'est à dire un seul sommet) suffit à couvrir tout le graphe. Ainsi, pour nos algorithmes, toute obligations est équivalente :

- en taille (1)
- en potentiel de couverture (n)
- en potentiel de ($n-1$)
- en rapport de domination ($n/1$)

de là, nous comprenons que nos sélecteurs deviennent impuissants face à ce type d'instance, et quelque soit l'obligation sélectionnée, le résultat est le même : 100% de couverture et un coefficient de domination maximum.
Le graphique détaillé 5 est en annexe.

6.4 Instances classiques

Pour ce type d'instance, on remarque que les taux de couvertures sont globalement bas (entre 84 et 78.2%, 80% par sélection aléatoire) et les tailles de solutions grandes, ce qui donne des coefficients de domination plutôt bas. Les meilleures couvertures sont données par le sélecteur appliqué sur la taille d'obligation et les meilleures coefficients de domination sont produits par le sélecteur par rapport de domination, sans surprise en mode dynamique (3.7%, contre 1.4% en aléatoire et 1.3% en sélection par taille).
Le graphique détaillé 6 est en annexe.

6.5 Instances à graphe connexes

En terme de taux de couverture, on a sensiblement la même configuration qu'avec les instances classiques. En revanche les solutions sont systématiquement plus petites (32.9% contre 41.7% en sélection dynamique par taille) et les coefficients de domination plus élevés (1.7 contre 1.3).
Le graphique détaillé 7 est en annexe.

6.6 Instances à graphe Grid

On a cette fois le nombre d'instances complètement solvées comme indicateur supplémentaire. Cela nous permet de remarquer que les sélecteurs par potentiel de dominance et par couverture présente des taux de couverture inférieurs au sélecteur aléatoire mais un bien plus grand nombre d'instances complètement solvées (696 et 784 contre 463). Le sélecteur par taille reste largement en tête, avec 81.9% contre 79.3%, le second meilleur (par couverture). On note que les solutions ont encore diminué en taille relative. Cf Dans le graphique détaillé 8 en annexe.

6.7 Instances à graphe Torus

Ici, les résultats sont très proches de ceux données sur les graphes Grid, avec simplement des solutions plus petites (environs 1% de différence partout) et des coefficients de domination plus légèrement supérieurs (0.1 point partout). Cela n'est pas étonnant quand on connaît la proximité structurelle qu'il existe entre les graphes en grille et les graphes de Torus. Cf Dans le graphique détaillé 9 en annexe.

6.8 Instances à graphe Hypercube

Les résultats produits sur les instances à graphes hypercubes sont assez remarquables : on a des taux de couverture et des coefficients de domination très élevés (93% et 3.8 au plus). Par ailleurs les écarts sur les différents indicateurs sont réduits par rapport à d'habitude, et pour la première fois on a deux algorithmes qui rattrapent le sélecteur par taille d'obligation en couverture : en dynamique, les sélecteurs par couverture et par domination sont aussi couvrants que ce dernier. Par ailleurs sur cette session le sélecteur par rapport de domination ne détient pas le coefficient de domination maximum (3.8 contre 3.9 en aléatoire) et en revanche son taux de couverture est inhabituellement élevé

(89.4% contre 88.2 en aléatoire).
Le graphique détaillé 10 est en annexe.

6.9 Conclusion sur les résultats

Appréciation générale des algorithmes Nos tests ont montré que le sélecteur par taille d'obligation était, en général, celui qui répondait le mieux au problème, à savoir couvrir au plus le graphe en respectant les obligations. En revanche nous avons montré que les autres sélecteurs pouvaient produire des solutions tel qu'un sommet dominant couvre localement plus de sommets qu'avec notre premier sélecteur. Cela nous mène à penser que sélectionner les obligations sur leur taille ne produit pas de solution optimale. Sans étonnement, on remarque que plus un graphes comporte d'arêtes, plus les solutions déterminés sont petites (minimal pour les graphes complets puisque les solutions sont constituées d'un seul sommet).

retour d'intuition Nous avons globalement été plutôt déçu par les résultats que nous avons obtenu, pour deux raisons. D'abord les différences de performances entre les différents algorithmes étaient moins flagrantes que nous ne le pensions. Nous avons dû faire de très longues sessions de calculs pour obtenir des statistiques stables (minimiser les variations des indicateurs entre deux sessions) et nous avons craint un lissage de ces indicateurs qui ne nous aurait pas permis de les comparer. Nous avons cependant fini par obtenir des résultats stables et des différences de comportements entre les algorithmes exploitables. Ensuite nous avons remarqué que nos différents algorithmes ne produisaient en général pas de solution biens meilleures que l'algorithme témoin, avec un facteur aléatoire. Il semble en fait que la contrainte d'obligation guide beaucoup la déterminisation de la solution et que le poids du sélecteur est moindre.

limites de nos tests Il faut bien comprendre que les instances que nous générons, même avec une spécialisation et dans un intervalle de tailles donnés, sont extrêmement variées, ne serait-ce que dans la distribution des obligations. Par conséquent il peut être risqué d'essayer de mesurer un comportement moyen. Nous avons d'ailleurs vu qu'il était compliqué de stabiliser nos résultats, même avec des sessions très longues. Nous savons qu'il existe une méthodologie très complète pour l'étude de qualité d'algorithme que nous ne maîtrisons pas. Cela nous pousse à remettre en doute les conclusions que nous tirons de nos graphiques. En particulier, il aurait peut être été plus judicieux d'aller plus loin dans la spécialisation d'instance, par exemple étudier exclusivement les performances de nos algorithmes sur des instances à graphes hypercubes plus précisément paramétrées. Également si nous avions su que nos premiers résultats seraient si peu évidents, nous aurions essayé d'utiliser d'autres outils statistiques comme les médianes, les écarts types etc. Enfin avec plus de temps nous aurions pu nous baser sur nos premiers résultats pour chercher à les détailler ou à les améliorer en retouchant notre code. Par exemple, nous avons remarqué après nos premières sessions que le nombre de solutions complètes déterminé était un indicateur intéressant, mais nous n'avons pas eu le temps de refaire des sessions pour l'intégrer.

7 Conclusion

Intérêts Ce projet nous a plu pour plusieurs raisons. En premier lieu, comme nous le voulions initialement, il nous a permis de nous mettre dans le rôle d'un chercheur en étudiant un problème plutôt abstrait, en cherchant des solutions et en essayant de les démontrer. Ensuite il nous a permis de nous initier aux graphes, qui, nous le savons, sont des outils mathématiques très puissants. Enfin nous avons été mené à une utilisation de python nouvelle pour nous, en particulier nous avons dû réfléchir à la représentation et à la distribution de l'information dans notre programme.

Perspectives Si ce projet était à refaire ou à continuer, nous essaierions d'appliquer les idées précédemment définies, notamment spécialiser plus nos tests pour répondre à la problématique sur un ensemble d'instances plus précis. Il serait en particulier intéressant de nous pencher plus sur le générateur d'obligation, par exemple produire un générateur d'obligations équilibrées (toutes de même taille).

A Annexes

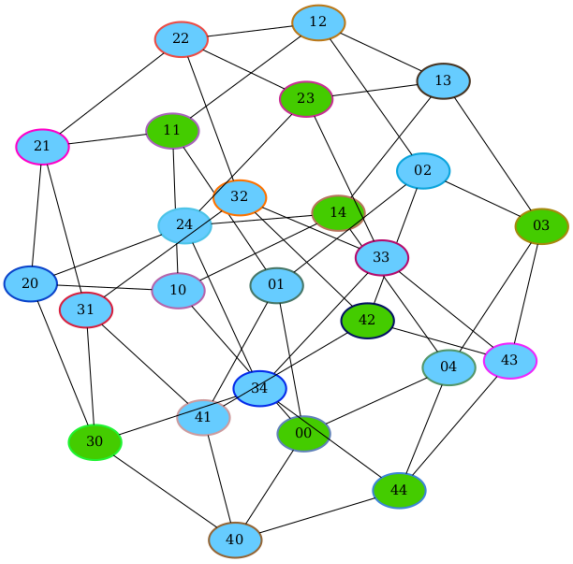

FIGURE 1 – Visualisation d'une instance à graphe Torus complètement solvée

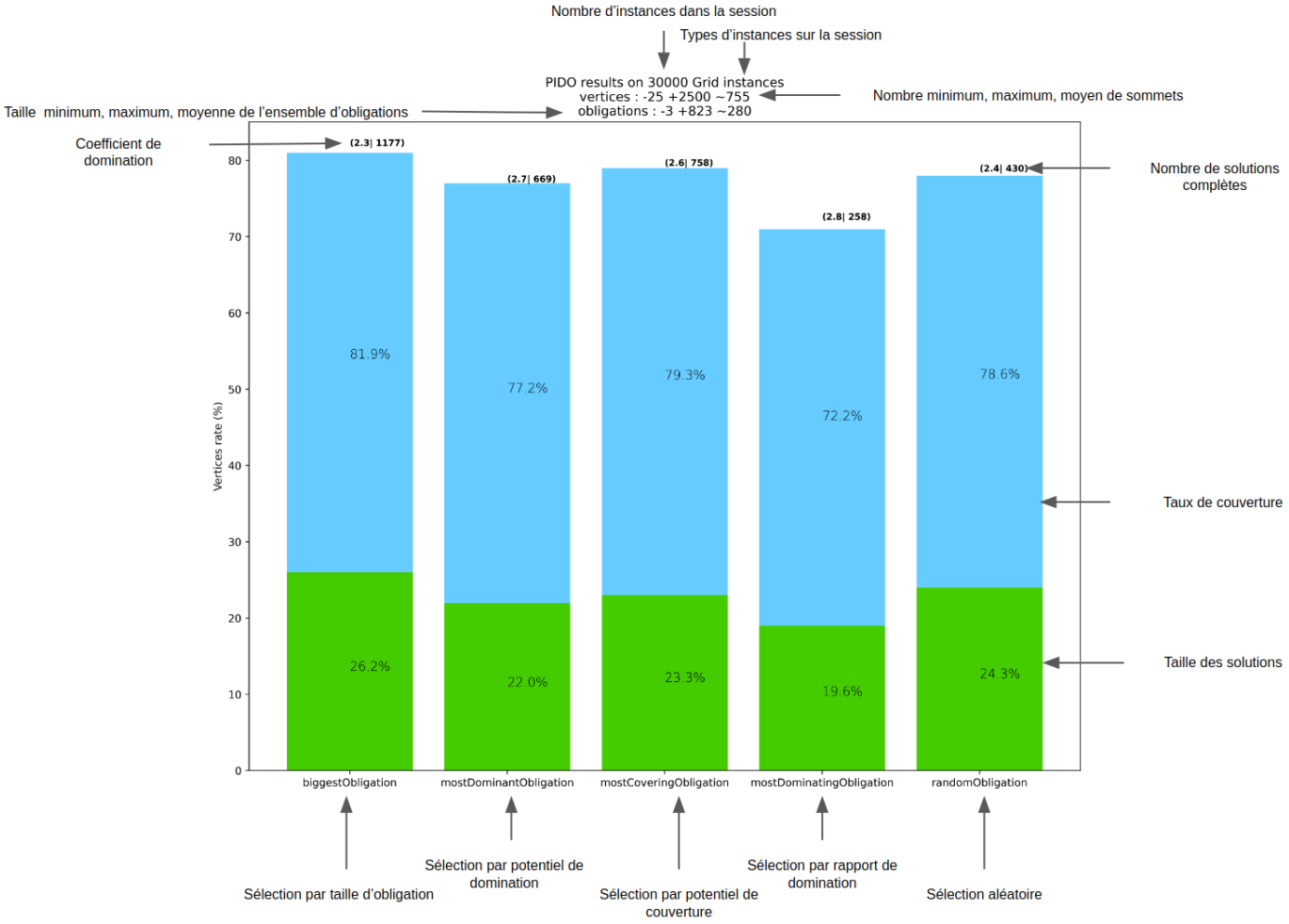

FIGURE 2 – Description détaillée des graphiques générés

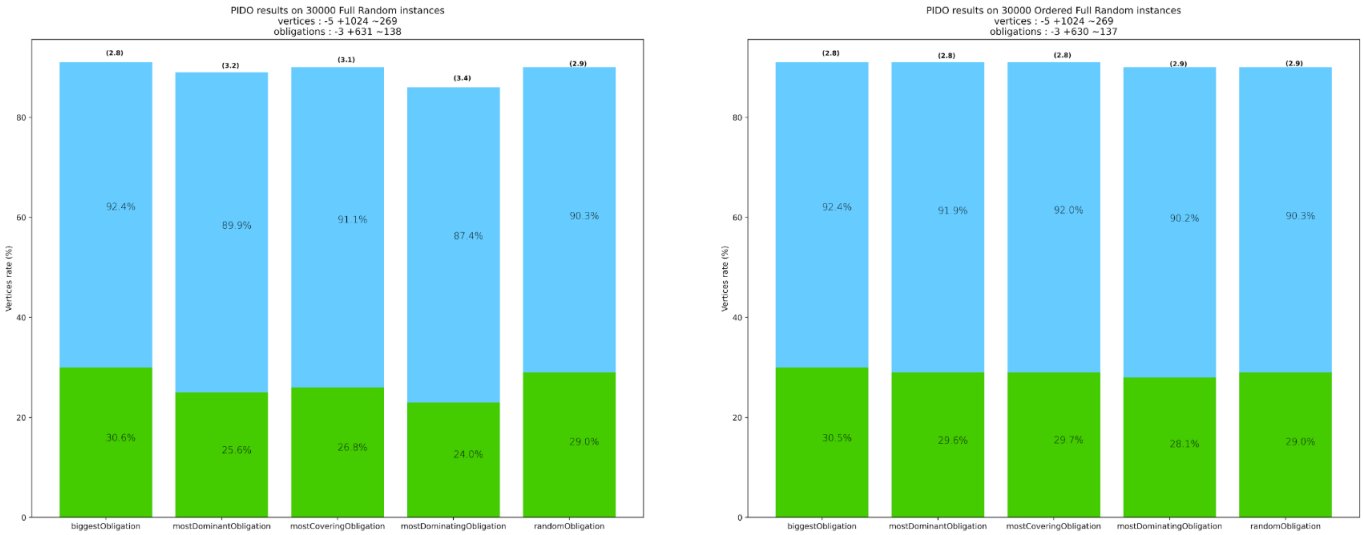

FIGURE 3 – Comparaison des modes de sélection statique et dynamique

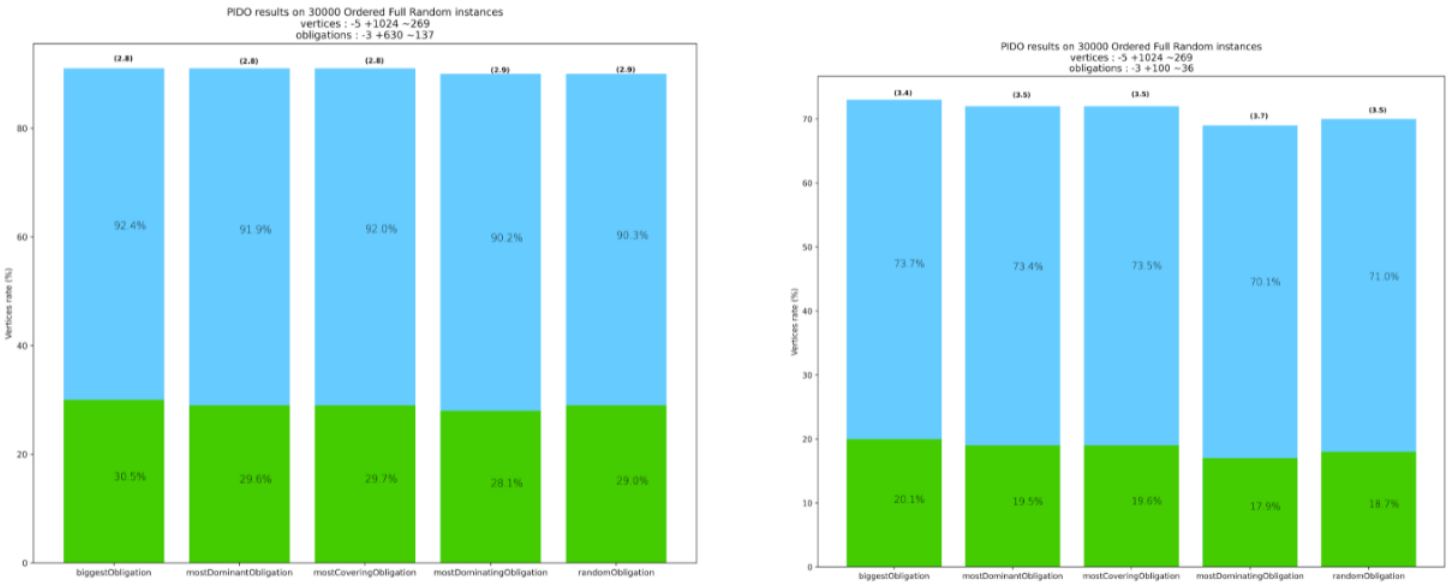

FIGURE 4 – Comparaison avec le facteur taille de l'ensemble d'obligations

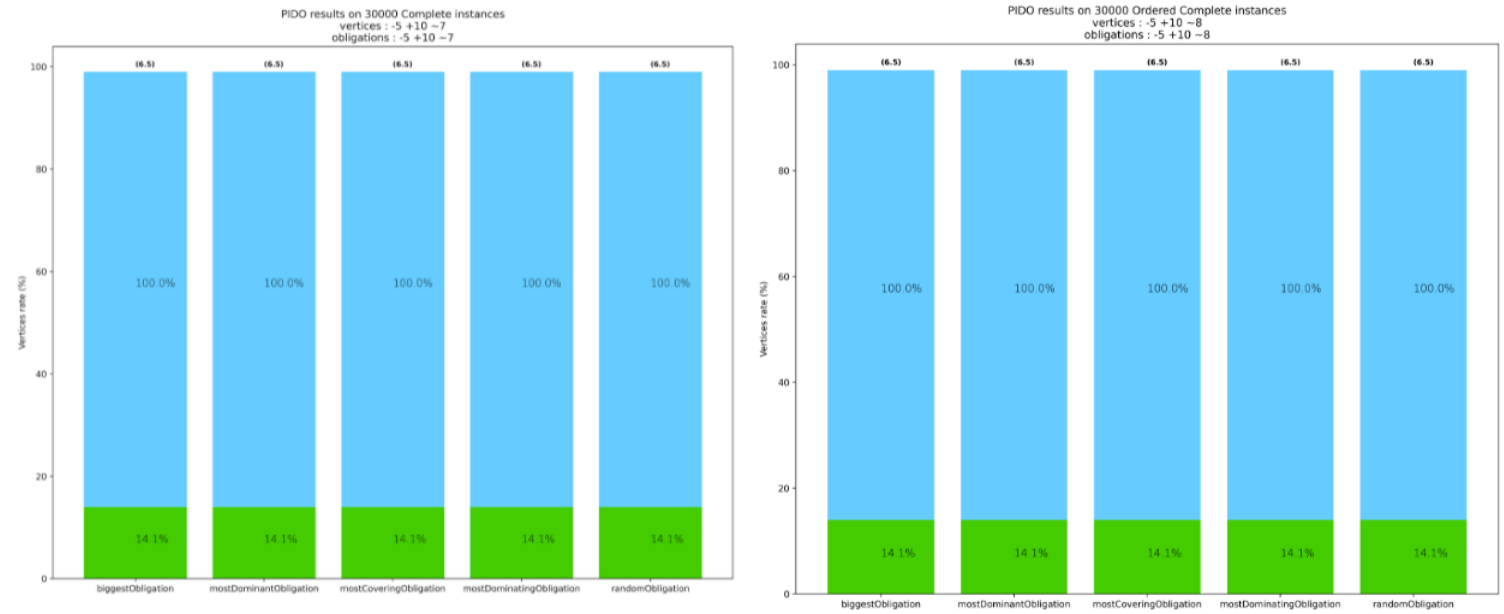

FIGURE 5 – Résultats sur les instances à graphe complet

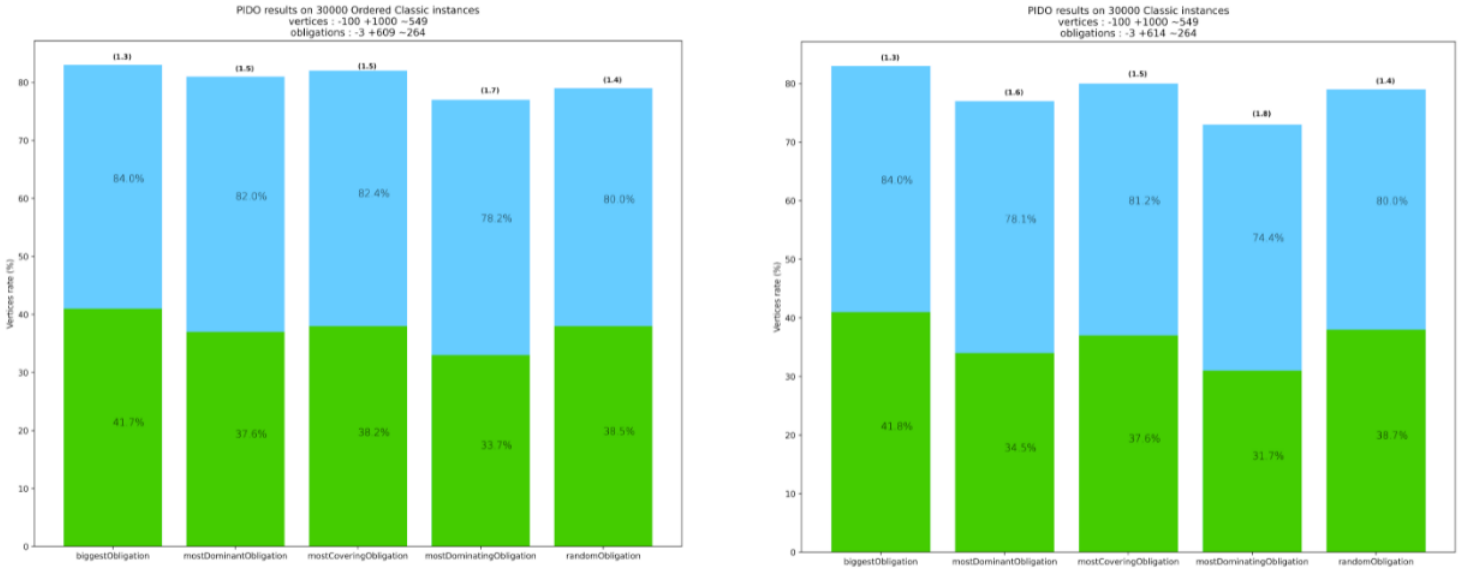

FIGURE 6 – Résultats sur les instances à graphe classique

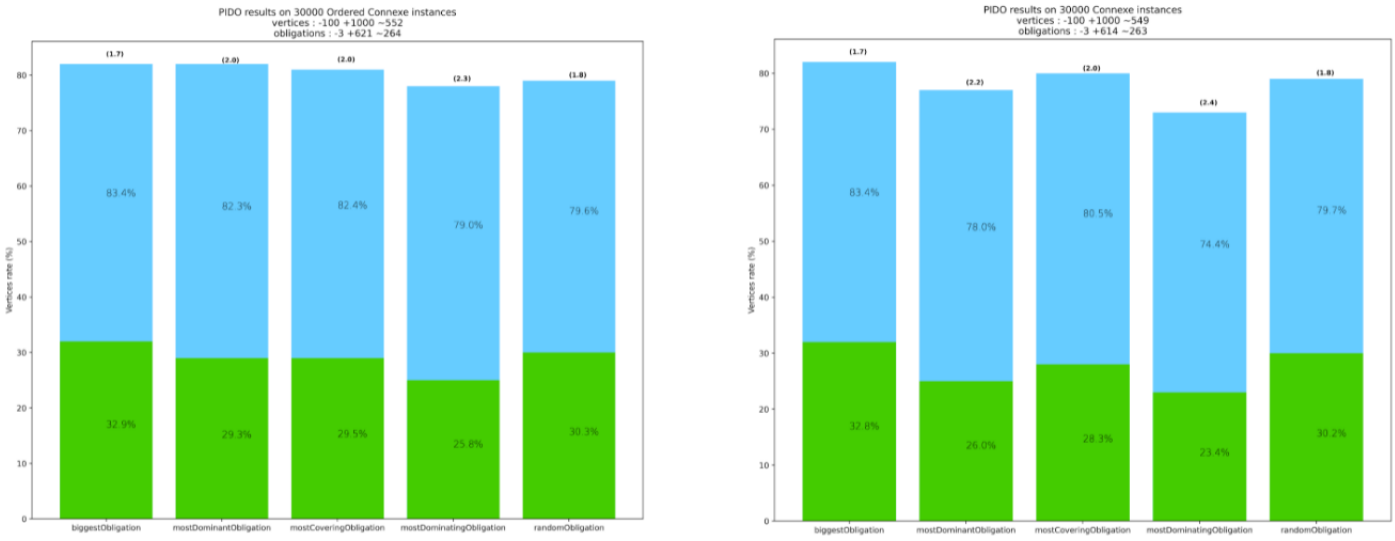

FIGURE 7 – Résultats sur les instances à graphe connexe

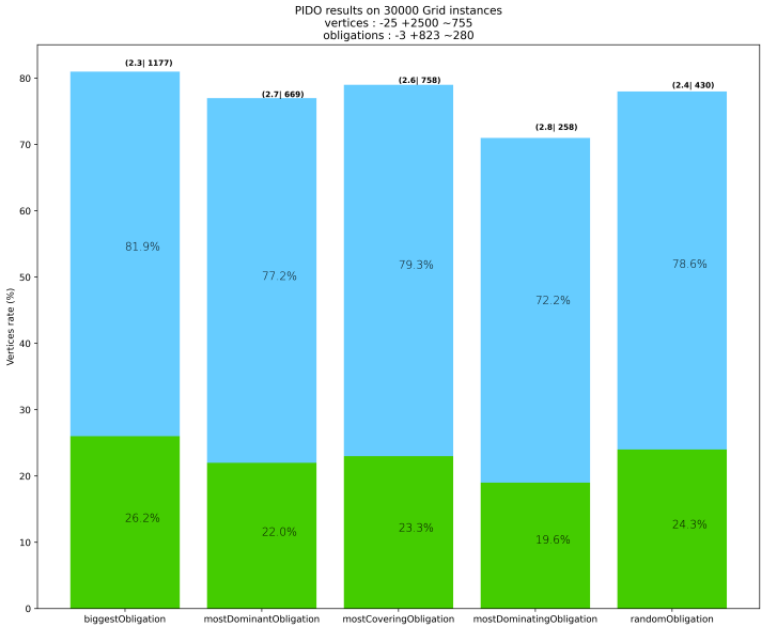

FIGURE 8 – Résultats sur les instances à graphe en grille

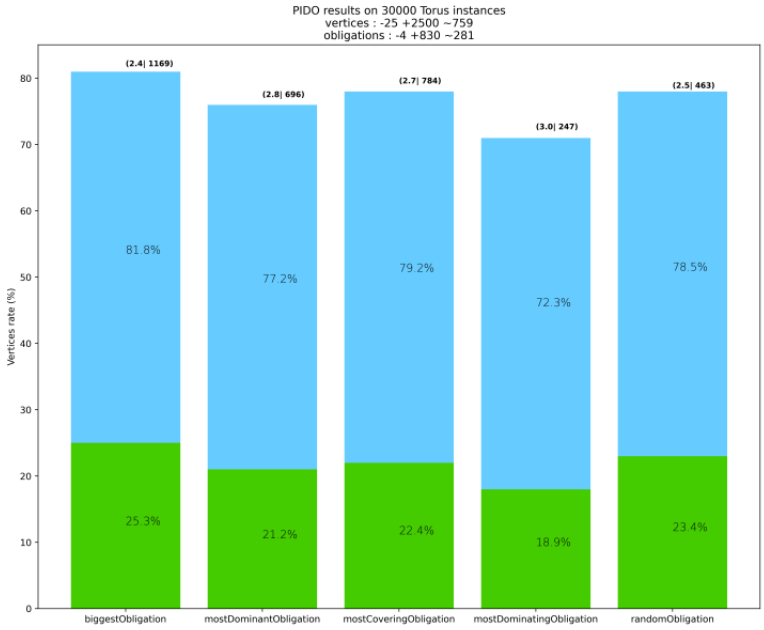

FIGURE 9 – Résultats sur les instances à graphe Torus

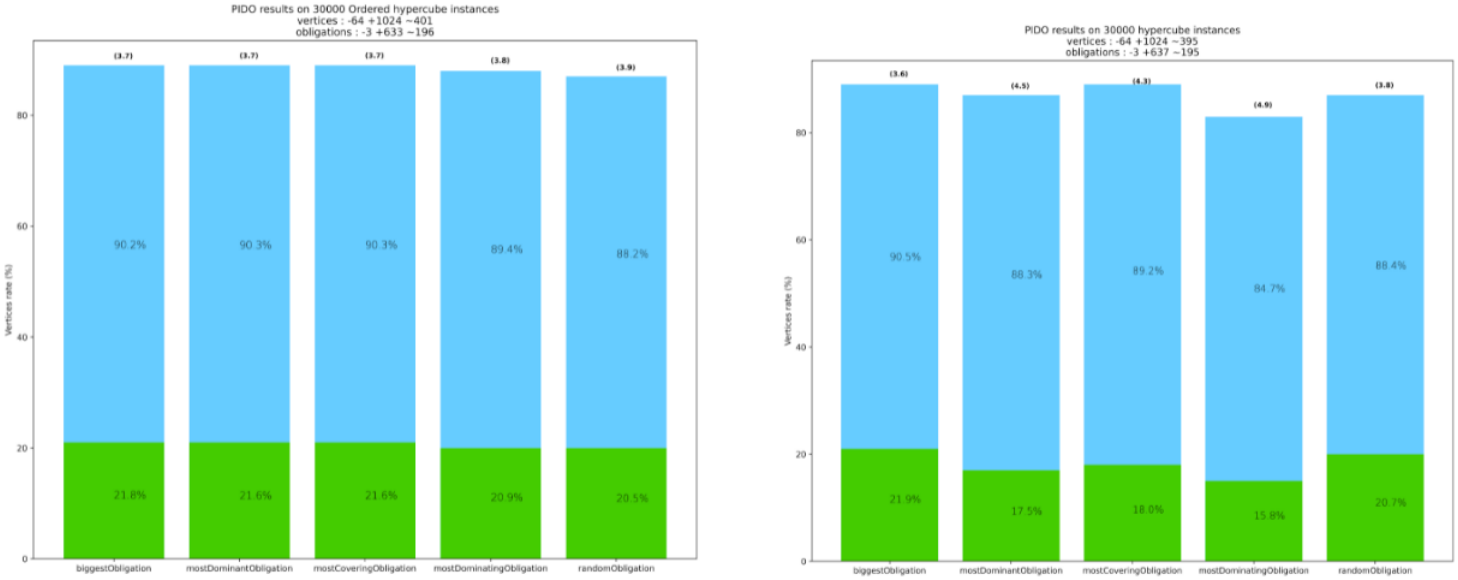

FIGURE 10 – Résultats sur les instances à graphe hypercube