Diviser n'est pas régner ?

Laurent Lyaudet*

22 décembre 2022

Résumé

En première année d'algorithmie, on découvre des exemples de schémas « Diviser pour régner » où une bonne division de l'instance du problème algorithmique étudié fournit aussi un algorithme de résolution efficace. Dans cet article, je présente un moyen de diviser efficacement (en temps quasi-linéaire) tous les graphes de degré borné. Ce qui amène au fait intéressant que si $\mathsf{P} \neq \mathsf{NP}$, alors « Diviser n'est pas régner. ».

Version initiale : 2022/05/08 Version courante : 2022/12/22
Nombre de pages : 18 + annexe avec code source.

1 Introduction

Depuis le début de l'algorithmique, une des méthodes les plus efficaces de résolution d'un problème est de diviser ou de décomposer une instance, puis de recombiner les solutions partielles des parties issues de la division/décomposition pour obtenir une solution globale (optimale). Trouver de bonnes manières de décomposer un problème est toujours un sujet de recherche très actif, notamment en théorie algorithmique et structurelle des graphes. On peut citer les décompositions modulaires, arborescentes, de branche, de rang, booléennes, etc. Il y en a pléthore et certaines sont à peu près équivalentes entre elles dans les instances qu'elles arrivent à décomposer.

Dans une première partie, je vous propose d'en découvrir une basée sur le principe de première différence que j'ai introduite en 2019 (Lyaudet (2019)) qui permet de décomposer tous les graphes de degré borné. Dans un second temps, nous verrons que cette décomposition amène naturellement à un algorithme de complexité « modérément exponentielle » pour le problème du stable maximum, même si cet algorithme n'est pas aussi performant que ceux de l'état de l'art pour le même problème. Le terme consacré pour des algorithmes de complexité inférieure à $O(2^n)$ dans la littérature est « subexponential », ce qui est assez fâcheux puisque la complexité de ces algorithmes dans le pire cas reste exponentielle, on a juste une base plus petite. Je préfère employer les termes de « modérément exponentielle » pour qualifier la complexité de ces algorithmes. Rappelons que le problème du stable maximum consiste à trouver un ensemble de sommets du graphe de cardinalité maximum, tel qu'aucune paire de sommets de cet ensemble n'est reliée par une arête. Le terme anglophone est « Maximum Independent Set ».

*https://lyaudet.eu/laurent/, laurent.lyaudet@gmail.com

2 Le principe de première différence et la largeur arborescente questionnable bijective équilibrée

Le principe de première différence est un principe très simple que tout le monde connaît sans le savoir, il a au moins 4000 ans comme la numération de position [1]. L'exemple le plus direct est l'ordre du dictionnaire ; pour comparer et ranger deux mots on regarde la première lettre qui diffère de celle de l'autre mot à la position équivalente. Si cette lettre est plus petite que l'autre dans l'ordre des lettres, on range le mot avant, si elle est plus grande, on range le mot après. (L'ordre des lettres correspond en gros à l'ordre alphabétique ; mais, selon les pays, les lettres accentuées peuvent être placées différemment ; voire dans les langues sans alphabet comme le chinois, on doit déterminer un ordre sur des caractères symboliques plutôt que des lettres.) Par exemple, « première » vient avant « principe », car on a comparé la première différence entre le « e » et le « i ». En adhérant au second principe qu'une « lettre inexistante » vient toujours avant, on obtient l'ordre lexicographique utilisé dans le dictionnaire. On fait exactement la même chose pour comparer deux nombres dont les écritures dans une même base ont la même longueur. Par exemple, 1118 vient avant 1122, car, en troisième position des deux suites de chiffres, le chiffre 1 vient avant le chiffre 2. En mettant là aussi avant les écritures les plus courtes, on obtient cette fois pour les nombres l'ordre hiérarchique, puisque la longueur totale des écritures est considérée avant le principe de première différence. C'est cet usage avec la numération de position qui me permet de « dater » ce principe, même s'il est très probable qu'il soit resté implicite, y compris parmi les utilisateurs de la numération mésopotamienne. Il est possible que l'usage de ce principe dans l'ordre lexicographique, de manière explicite ou implicite, soit encore plus ancien ; mes connaissances historiques modérées à ce sujet ne me permettent pas de le dire.

J'ai proposé deux idées en conclusion de Lyaudet (2019) :

— on peut utiliser le principe de première différence non seulement pour définir des relations d'ordre mais aussi d'autres types de relations binaires, comme par exemple l'adjacence entre sommets dans les graphes ; la première différence entre les caractères associés à deux sommets déterminera si ceux-ci sont adjacents ou non ;

— on peut étendre le principe de première différence entre deux suites de caractères à un principe de première différence entre des caractères disposés sur un arbre et non un chemin.

Voyons ce que cela donne dans le cas des graphes.

Soit un ensemble de sommets V, une (V, k)-*suite-d'applications* est une suite d'applications (fonctions totales au sens mathématique) de V vers les sommets de graphes de cardinalité au plus k (un même graphe par application).

Definition 2.1. *Soit G un graphe. Une* (k, β)-décomposition questionnable *de G est une $(V(G), k)$-suite-d'applications de longueur β, telle que, pour toute paire de som-*

1. Consulter l'article sur Wikipedia sur la numération positionnelle `https://fr.wikipedia.org/wiki/Notation_positionnelle`. Ne sachant si cela date du début ou de la fin du troisième millénaire avant Jésus-Christ, nous avons choisi l'hypothèse la plus certaine d'au moins 4000 ans ; mais la vérité est peut-être plus proche de 5000 ans.

mets $x, y \in V(G)$, la première différence entre l'image de x et de y dans cette suite d'applications existe et qu'elle corresponde à deux sommets adjacents, resp. non-adjacents, si x et y sont adjacents, resp. non-adjacents. k est appelée la largeur de la décomposition ; β est appelée la profondeur de la décomposition.

Le terme « questionnable » vient de mon initiative, plus ou moins heureuse, d'appeler la première différence entre deux suites, la « question [2] » de ces deux suites, avant de trouver l'antériorité des termes de « principe de première différence ». En me documentant d'avantage, j'ai fini par montrer en 2020 que les graphes de largeur questionnable k sont exactement ceux de largeur modulaire k. Rappelons que la décomposition modulaire d'un graphe consiste à décomposer en des parties qui pour chaque paire de parties sont soit reliées totalement, soit nullement reliées entre elles. Ces parties sont appelées modules car aucun sommet extérieur ne peut distinguer les sommets d'un module ; il est soit adjacent à tous, soit à aucun. Sans rentrer dans les détails, la décomposition modulaire fait « les bons choix » de modules à chaque étape. Quand elle peut couper proprement le graphe en deux modules ou plus et ajouter toutes les arêtes entre modules, on parle de nœud/composition série, habituellement notée * ; quand elle peut couper proprement le graphe en deux modules ou plus sans ajouter d'arête entre modules, on parle de nœud/composition parallèle, habituellement notée // ; quand elle doit découper en quatre modules ou plus avec des adjacences diverses, on parle de nœud primitif associé au graphe primitif obtenu en fusionnant tous les sommets de chacun des modules choisis en un sommet unique. (Le lecteur qui ne connaît pas le sujet trouvera sans doute intéressant de prouver, par lui-même, ce saut de 2 modules à 4 modules, en constatant qu'il n'existe pas de graphe primitif à trois sommets dans le cas des graphes non-orientés et non-étiquetés.) Avec la convention que la largeur de la décomposition modulaire est le maximum des cardinalités des graphes primitifs utilisés (et de deux), on obtient l'équivalence citée. La décomposition modulaire d'un graphe est unique contrairement aux décompositions questionnables, mais une décomposition questionnable de largeur optimale peut « encoder » la décomposition modulaire sous une forme sérielle « multiplexée ». (Les branches de l'arbre de décomposition modulaire se retrouvent multiplexées.) La racine de la décomposition modulaire correspond à la fin de la suite d'applications.

Nous allons à présent utiliser le principe de première différence sur une structure d'arbre :

Definition 2.2. *Soit G un graphe. Une (k, α, β)-*décomposition arborescente questionnable bijective *de G est un triplet (A, ef, en) (A comme arbre, ef comme étiquetage des feuilles et en comme étiquetage des nœuds) :*

- *A est un arbre binaire enraciné ;*
- *les feuilles de A sont en bijection, par l'intermédiaire de la fonction ef, avec les sommets de G ;*
- *ainsi à chaque nœud interne $node$ est associé l'ensemble de sommets de G union des valeurs $ef(f)$ pour toutes les feuilles f sous le nœud $node$, ce qui définit $ef(node)$;*

2. Exemple d'encodage binaire : (être, être, être, être) et (être, être, être, ne pas être).

— *en est une application ayant pour domaine les nœuds internes de A, telle que $en(node)$ est une $(ef(node), k)$-suite-d'applications,*
— *en conséquence, à chaque sommet de G correspond un sous-arbre (qui est un chemin dans cette définition simplifiée) de A, et puisque l'intersection de deux arbres, resp. chemins, est un arbre, resp. chemin, on a aussi un chemin correspondant à tout couple de sommets (x, y). On peut ainsi définir la $(\{x, y\}, k)$-suite-d'applications obtenue en concaténant les $(ef(node), k)$-suites-d'applications restreintes à $\{x, y\}$, et l'on impose que la première différence entre l'image de x et de y dans cette suite d'applications existe et qu'elle corresponde à deux sommets adjacents, resp. non-adjacents, si x et y sont adjacents, resp. non-adjacents ;*
— *α est la profondeur de l'arbre A ;*
— *β est la profondeur de l'arbre étendu A' obtenu en remplaçant chaque nœud interne par un chemin de nœuds (un pour chaque application de la suite associée au nœud original).*

k est appelée la largeur de la décomposition ; α est appelée la profondeur structurelle de la décomposition ; β est appelée la profondeur logique de la décomposition.

En fait cette définition est très puissante puisque tout est décomposable avec une largeur de 2 et une profondeur linéaire. Il suffit d'adjoindre les sommets à un arbre ressemblant à un peigne un par un. Afin de l'illustrer, nous allons décomposer un chemin de longueur 3 (4 sommets), affectueusement nommé P_4 par les initiés, qui est l'unique obstruction à chercher comme sous-graphe induit, pour savoir si un graphe a une largeur modulaire/questionnable égale à deux. La notation — correspond à la composition série sous sa forme application vers les sommets d'un graphe à deux sommets adjacents. La notation | correspond à la composition parallèle sous sa forme application vers les sommets d'un graphe à deux sommets non-adjacents.

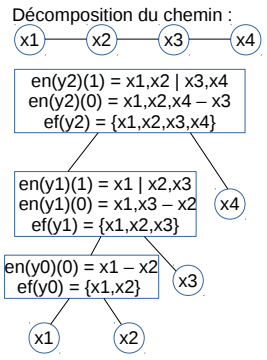


FIGURE 1 – Décomposition d'un chemin avec la première différence de bas en haut

On voit aisément que deux applications suffisent dans chaque nœud interne dans

le cas des graphes simples. On peut bien de la sorte tout décomposer avec une largeur de 2, une profondeur structurelle au plus égale au nombre de sommets du graphe, et une profondeur logique au plus égale à 2 fois le nombre de sommets du graphe. Dans le cas de graphes étiquetés (les étiquettes pouvant traduire des arêtes multiples) et/ou orientés, ce résultat se généralise avec une profondeur logique au plus égale à k fois le nombre de sommets du graphe, si l'on a k types d'adjacence. (La non-adjacence est un type d'adjacence de ce point de vue.) Pour limiter cette facilité un peu trop grande à tout décomposer sans que la largeur n'augmente, on se restreint aux décompositions équilibrées, pour lesquelles la profondeur structurelle est logarithmique en la taille du graphe décomposé. Le prochain exemple illustré omettra totalement de préciser ef, et omettra de préciser de quelle application d'une image de en il s'agit. J'espère que le lecteur admettra que c'est tout aussi lisible et moins lourd que le premier exemple.

3 La décomposition des graphes de degré borné

Quand on cherche à décomposer un graphe, dans certains cas basés sur la connexité du graphe, on cherche à trouver des ensembles minimaux de sommets ou d'arêtes, qui une fois enlevé(e)s donnent un graphe non-connexe. Ils servent alors de frontières dans la décomposition. C'est notamment le cas avec les décompositions selon un chemin et arborescentes (path-decompositions and tree-decompositions). Intuitivement, dans ce cas, on essaye de mettre ensemble les parties les plus connectées entre elles. Sauf que là ça ne marche pas du tout ; il faut faire l'inverse. Si on a un graphe où le degré maximal d'un sommet est au plus d, disons 3 par exemple, il y a au plus $1 + d + d \times (d - 1) = 1 + d^2$, par exemple 10, sommets dans une boule de rayon 2 centrée sur un sommet. Donc on peut découper notre graphe en au plus $1 + d^2$, par exemple 10, parties pour lesquelles chaque sommet est au moins à distance 3 de tous les autres sommets de la même part. Il suffit d'un algorithme glouton : tant qu'un sommet n'est pas affecté à une des parts, on l'affecte à la première part qui ne contient pas d'autre sommet de la boule de rayon 2 centrée sur ce sommet.

Du coup, dans la décomposition, on a un stable pour chaque partie, c'est-à-dire un sous-graphe induit sans arête, avec une décomposition modulaire ou questionnable triviale : on prend un arbre binaire équilibré où les feuilles sont en bijection avec les sommets de la partie et où chaque nœud interne est associé à une seule ≪ application parallèle ≫ qui envoie les sommets de gauche dans le sommet gauche et les sommets de droite dans le sommet droit d'un graphe à deux sommets non-adjacents, un à gauche, un à droite.

Pour relier les parties entre elles, on utilise à nouveau la structure de peigne qui décompose tout. Du coup, on a un sous-graphe qui grossit de plus en plus car on lui ajoute les parties une à une, en ayant initialisé le sous-graphe par une partie quelconque. Supposons que ce sous-graphe est toujours à gauche et que la partie ajoutée est à droite. Comme les sommets de la partie ajoutée sont à distance au moins 3, aucun sommet du sous-graphe à gauche n'est adjacent à plus d'un sommet de la partie droite. On a donc un graphe biparti relativement trivial, tout comme sa décomposition bimodulaire (Fouquet et al. (2004)), puisque c'est une union de sommets isolés et d'étoiles. (Une étoile en théorie des graphes est un graphe avec un sommet central qui est adjacent à

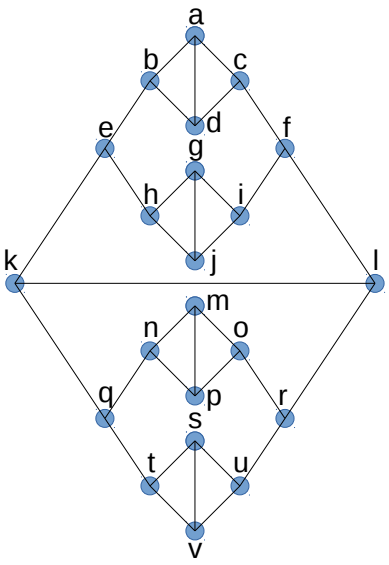
des sommets extérieurs et aucune des paires de sommets extérieurs n'est adjacente.) En termes de suite d'applications, c'est quasi aussi trivial qu'à l'intérieur des parties. On commence par faire des $\ll$ applications parallèles $\gg$ un nombre logarithmique de fois, pour décomposer le graphe biparti selon ses composantes connexes (les sommets isolés et les étoiles). Et on finit par une $\ll$ application série $\gg$ vers un graphe à deux sommets adjacents, un à gauche, un à droite, un pour le sous-graphe de gauche et un pour la partie de droite. Par application du principe de première différence, les adjacences définies correspondent aux étoiles et aux sommets isolés, les adjacences intra-parties ayant déjà été fixées auparavant. On obtient une décomposition de largeur 2 dont les profondeurs structurelles et logiques sont logarithmiques.

Il est aisé d'encoder les informations liées à un sommet avec 4 caractères : g ou d sommet gauche ou droit d'une application, G ou D branche de gauche ou de droite dans la décomposition, voire 6 avec g' et d' si on veut distinguer les applications séries et parallèles. (L'option avec 6 caractères donne une information redondante puisque l'image d'une application ne dépend pas d'un sommet mais de la position dans l'arbre de décomposition. De plus, seule la dernière application de la réunion de deux parties a pour image deux sommets adjacents.) Une fois que l'on a les 2 suites de caractères associées à deux sommets, on part de la fin des deux suites (la racine de l'arbre de décomposition), et on garde tous les caractères avant tant qu'il n'y a pas de différence entre deux majuscules (G ou D). Ainsi, on sélectionne le sous-chemin commun aux deux sommets dans l'arbre de décomposition. Ensuite, il n'y a plus qu'à regarder la première différence entre ces deux sous-suites qui sont nécessairement de même longueur. On peut aussi réduire le nombre de caractères sans augmenter la longueur, car en réalité 0 et 1 suffisent ; la donnée de l'arbre de décomposition avec le nombre d'applications dans chaque nœud suffit à savoir si un 0 correspond à un g ou un G et si un 1 correspond à un d ou un D. Mais pour cela, il faut à nouveau partir de la fin de la suite de caractères et non du début. La variante à 4 caractères nous semble le meilleur compromis, le meilleur taux de redondance d'information, pour l'esprit humain.

Nous avons démontré le théorème suivant :

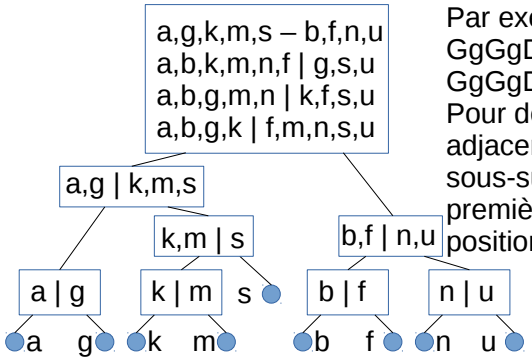
Théorème 3.1 (Décomposition des graphes de degré borné par le principe de première différence)**.** *Tout graphe de degré au plus d à n sommets admet une décomposition arborescente questionnable bijective équilibrée de largeur 2, de profondeur structurelle au plus $\lceil \lg(n) \rceil + d^2$ et de profondeur logique au plus $\lceil \lg(n) \rceil + d^2 \times (\lceil \lg(n) \rceil + 1) = \lceil \lg(n) \rceil \times (1 + d^2) + d^2$. De plus, cette décomposition est calculable en temps quasi-linéaire.*

Exemple de graphe 3-régulier :



L'algorithme glouton sur les sommets dans l'ordre des lettres nous donne les 6 parties suivantes : (a,g,k,m,s), (b,f,n,u), (c,e,o,t), (d,h,l,p,v), (i,q,r), (j).

Exemple de début de décomposition correspondant aux 2 premières parties où la première différence s'évalue en partant du bas de la décomposition et en remontant vers la racine : on représente avec la barre verticale '|', respectivement le tiret '–', quand on associe les sommets à deux sommets non adjacents, resp. adjacents.
Il est aisé d'encoder les informations liées à un sommet avec 4 caractères : g ou d sommet gauche ou droit d'une application, G ou D branche de gauche ou de droite.



Par exemple, k donne GgGgDdGgdgg, b donne GgGgDgggd.
Pour déterminer, leur adjacence, on garde les sous-suites gdgg et gggd, la première différence est en position 2 sur deux sommets non adjacents donc k et b sont non adjacents.

FIGURE 2 – Exemple de décomposition

4 Une application simple au problème de stable maximum

Le problème du stable maximum reste NP-complet sur les graphes de degré 3 (Garey et al. (1976)). La réduction depuis 3-SAT est assez simple. Déjà sans restreindre le degré, on peut associer chaque clause à un triangle (un graphe à 3 sommets tous reliés), où chaque sommet du triangle correspond à un des littéraux de la clause. De plus, chaque littéral est adjacent aux littéraux correspondant à sa négation présents dans les autres clauses. De cette manière, il est impossible d'avoir un littéral et une de ses négations dans un stable maximum. Comme on ne peut pas avoir plus d'un sommet par triangle, c'est-à-dire un littéral par clause, un stable maximum ne peut avoir plus de sommets que le nombre de clauses, et il n'atteint ce nombre que s'il correspond à une affectation de valeurs de vérité rendant vraie au moins un littéral par clause. Pour avoir un degré au plus 3, puisque le degré dans chaque triangle est de 2, au lieu de relier directement chaque littéral à ses négations, on relie les littéraux portant sur la même variable booléenne par un cycle de longueur paire. Pour chaque cycle, on fixe une origine et on associe les littéraux positifs à l'un des sommets de rang pair depuis l'origine, et les littéraux négatifs à l'un des sommets de rang impair depuis l'origine. Comme un stable dans un cycle pair ne peut couvrir qu'un sommet sur deux et que le stable maximum est atteint si l'on prend tous les sommets pairs ou bien tous les impairs, la synchronisation est assurée. Et là encore le maximum n'est atteint que si l'on s'est abstenu de prendre deux littéraux contraires.

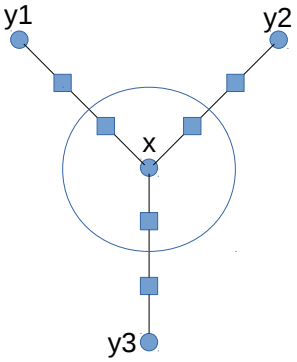
On a donc une décomposition en 10 parties pour le problème du stable maximum dans les graphes de degré 3. Trouver un stable maximum dans une partie est trivial puisque chaque partie est un stable. Dans l'union de deux parties, ce n'est pas beaucoup plus compliqué, puisque l'on a des sommets isolés et des arêtes isolées ; il suffit donc de prendre les sommets isolés et un sommet par arête isolée. Dans l'union de trois parties, chaque sommet est de degré au plus deux ; donc on a une union de cycles et de chemins, possiblement réduits à un sommet isolé ; et la résolution du sous-problème est à nouveau aisée. Mais il s'avère que dès 4 parties, c'est-à-dire la troisième fois que l'on réunit des parties, le problème devient NP-complet.

Pour montrer cela, on va d'abord voir que le problème reste NP-complet sur un problème encore moins dense que le précédent. Comment est-ce que l'on peut faire moins dense ? Avant 3, c'est 2, on ne va tout de même pas se ramener à des chemins ou des cycles ? Presque, on va remplacer chaque arête par un chemin de longueur 3, on aura donc créé deux sommets intermédiaires entre deux sommets du graphe d'origine. Tout d'abord on peut remarquer qu'en faisant cela on augmente la taille d'un stable maximum d'au plus le nombre d'arêtes. En effet, le nombre maximum de sommets choisis dans le stable maximum sur un chemin de longueur 3, c'est toujours un plus le nombre de sommets choisis aux extrémités quitte à déplacer un choix. Pour être convaincu, c'est plus simple de s'imaginer remplacer les arêtes par un chemin une à une, en obtenant autant de graphes intermédiaires que d'arêtes et de voir que la cardinalité du stable maximum n'augmente que d'un à chaque fois. Nommons $x - r - s - y$ les sommets du chemin, dans un sens :

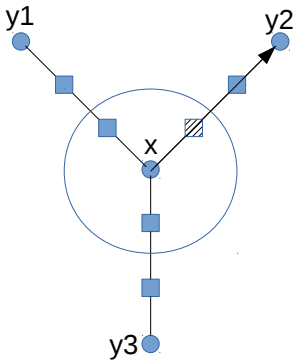
— sachant que j'ai choisi x et pas y dans un stable maximum du graphe de départ,

Allongement d'une arête :

Chaque sommet engendre 3 nouveaux sommets :



Choisir un de ces sommets, c'est comme choisir un arc sortant du sommet initial :



On ne peut pas choisir deux arcs sortants du même sommet à la même étape puisqu'ils correspondent à deux sommets à distance 2. On ne peut pas non plus avoir deux arcs opposés puisqu'ils correspondent à des sommets à distance 1.

FIGURE 3 – Rendre le graphe moins dense

je peux ajouter s et j'obtiens bien plus 1, je ne peux pas faire plus localement,
- sachant que j'ai choisi y et pas x dans un stable maximum du graphe de départ, je peux ajouter r et j'obtiens bien plus 1, je ne peux pas faire plus localement,
- sachant que je n'ai choisi ni x ni y dans un stable maximum du graphe de départ, je peux ajouter r ou s et j'obtiens bien plus 1, x et y étaient chacun bloqués par un autre sommet du graphe, je ne peux pas faire plus localement ;

dans l'autre sens :

- sachant que j'ai choisi x et y dans un stable maximum du graphe d'arrivée, je remplace arbitrairement y par s et j'obtiens bien moins 1 en revenant au graphe d'arrivée, je ne peux pas faire plus localement,
- sachant que j'ai choisi x et s, resp. y et r dans un stable maximum du graphe d'arrivée, j'obtiens bien moins 1 en revenant au graphe de départ, je ne peux pas faire plus localement,
- sachant que j'ai choisi r, resp. s, dans un stable maximum du graphe d'arrivée, j'obtiens bien moins 1 en revenant au graphe de départ, je ne peux pas faire plus localement.

Notons $StableMax(G)$ la cardinalité d'un stable maximum d'un graphe G. En notation synthétique, à la probabilité conditionnelle (c'est-à-dire, par exemple, que $StableMax(G|a, b, \neg c, \neg d)$ dénote la cardinalité d'un stable maximum du graphe G sachant que l'on se restreint aux stables contenant les sommets a, b et ne contenant pas les sommets c, d),

$$\begin{aligned} StableMax(G) = max(& \\ & StableMax(G|x, \neg y), \\ & StableMax(G|\neg x, y), \\ & StableMax(G|\neg x, \neg y) \\ ) & \\ = max(& \\ & StableMax(G'|x, \neg y) - 1, \\ & StableMax(G'|\neg x, y) - 1, \\ & StableMax(G'|\neg x, \neg y) - 1, \\ & StableMax(G'|x, y) - 1 \\ ) & \\ = StableMax(G') - 1 & \end{aligned}$$

car $StableMax(G'|x, \neg y) \geq StableMax(G'|x, y)$ par exemple.

Nous avons démontré le théorème suivant :

Théorème 4.1. *Toute instance du problème du stable maximum, sous la forme d'un graphe G à n sommets et m arêtes, peut être convertie en une instance équivalente G', selon la relation $StableMax(G) + m = StableMax(G')$, ayant $n + 2m$ sommets et $3m$ arêtes, telle que le degré des sommets initiaux reste identique et qu'ils sont à des distances multiples de 3, les nouveaux sommets sont de degré deux, donc, en particulier, tous les sommets de degré supérieur à 2 sont à distance au moins 3. Dans*

le cas des graphes 3-réguliers, G' a 4 fois plus de sommets et 3 fois plus d'arêtes. Et cette réduction s'effectue en temps linéaire.

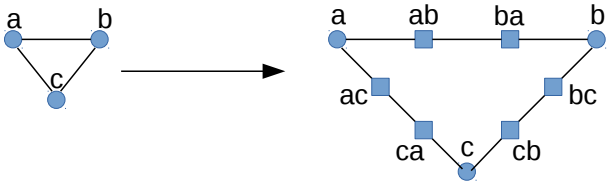
Nous allons à présent chercher à minimiser le nombre de parties de sommets à distance au moins 3. Le problème, en général, n'est pas trivial car il s'agit d'un problème de coloration propre de graphe ou plus exactement de son « carré » (puisque l'on exclut d'avoir la même couleur/partie sur deux sommets à distance 1 ou 2, on prend donc le graphe ayant pour matrice d'adjacence le carré de la matrice d'adjacence du graphe « dédensifié », modulo le fait qu'on considère avoir des boucles c.-à-d. des 1 sur la diagonale de la matrice, avant le passage au carré, et qu'on les enlève juste après). Le problème de la coloration propre des carrés est NP-complet. Le degré maximum plus un est une borne inférieure triviale sur le nombre de couleurs nécessaires, puisqu'un sommet et ses voisins forment une clique dans le carré du graphe. Et nous allons voir que cette borne inférieure est atteinte, nous donnant un résultat d'optimalité. (Le degré maximum du graphe de départ est exactement le degré maximum de son graphe dédensifié, sauf dans le cas où il vaut 1, et pas 0, c.-à-d. quand le graphe de départ est un couplage non vide et d'éventuels sommets isolés. Les résultats qui suivent ne repréciseront pas cette légère anomalie.)

Considérons donc les carrés des graphes dédensifiés. Chaque sommet initial et ses voisins dans le graphe dédensifié forment une clique dans le carré ; mais, dans le carré, on a aussi les arêtes qui relient un sommet original aux sommets intermédiaires opposés. Rappelons que :

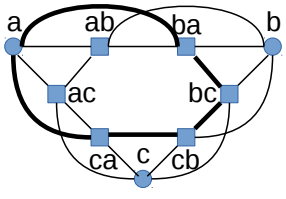
- les graphes parfaits sont ceux qui ont la propriété que, dans tout sous-graphe induit, le nombre de couleurs nécessaires dans une coloration propre (nombre chromatique) est égal à la taille d'une clique maximum ;
- un trou est un cycle sans corde (une corde étant une arête qui relie deux sommets non consécutifs du trou) ;
- un anti-trou est le complémentaire d'un trou ;
- un trou impair, respectivement un anti-trou impair, est un trou, respectivement un anti-trou, avec un nombre impair de sommets ;
- le trou à 5 sommets est auto-complémentaire, c'est un anti-trou, et c'est le plus petit graphe qui nécessite une couleur de plus (3) que la taille d'une clique maximum (2) ;
- les graphes sans trou ni anti-trou impair à au moins 5 sommets sont appelés graphes de Berge ;
- le théorème fort des graphes parfaits nous dit que les graphes parfaits sont exactement ceux de Berge.

Si l'on cherche tout d'abord à regarder du côté des graphes parfaits, on voit rapidement que les graphes dédensifiés puis carrés contiennent des trous à 5 sommets : partir d'un triangle a,b,c, le dédensifier en a, ab, ba, b, bc, cb, c, ca, ac, puis garder le trou a, ba, bc, cb, ca dans le carré (les 5 sommets sont à distance au plus deux et l'on n'a pas pris trois sommets consécutifs donc il n'y a pas de corde). Néanmoins, les sommets du graphe original forment une partie de sommets à distance au moins 3. Si l'on enlève cette partie, on obtient des graphes « dédensifiés puis carrés puis évidés » qui sont des graphes parfaits. En effet si l'on regarde les 120 sous-classes de graphes parfaits listées dans Hougardy (2006), ils appartiennent à plusieurs de ces sous-classes

Dédensification d'un triangle :



Passage au carré avec un trou de taille 5 mis en évidence (a, ba, bc, cb, ca) :



On repère bien aussi les cliques de voisins {a, ab, ac}, {b, ba, bc}, {c, ca, cb}.

Évidement, les cliques de voisins sont à présent réduites à deux sommets, on ne les distingue plus vraiment des arêtes inter-cliques :

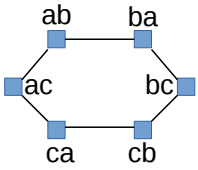


FIGURE 4 – Obtenir un graphe dédensifié puis carré puis évidé

(l'auteur n'a pas encore déterminé la liste complète), dont la première dans l'ordre alphabétique : alternately colorable. Il est facile de voir que les graphes considérés sont faits de cliques (les voisins d'un sommet du graphe de départ), chaque sommet d'une clique ayant exactement un autre voisin en plus dans une autre clique. Donc si l'on colore les arêtes des cliques avec une couleur et les arêtes inter-cliques avec une autre couleur, on obtient bien une coloration alternée pour tout trou présent dans le graphe. Les graphes dédensifiés puis carrés puis évidés sont donc bien des graphes parfaits. Et ce résultat nous permet de dire que les graphes dédensifiés puis carrés sont 1-parfaits (leur nombre chromatique égale leur taille de clique, sans que ce soit vrai pour tout sous-graphe induit).

Nous allons maintenant montrer que l'on peut obtenir une coloration optimale en un temps linéaire $O((\Delta + 1) \times n)$, avec un simple parcours en largeur modifié. (Δ est le degré maximum du graphe de départ.) L'algorithme a $\Delta + 1$ étapes ; chacune s'effectuant en temps linéaire en le nombre de sommets. La première consiste simplement à mettre les sommets du graphe de départ dans une partie. L'objectif de chaque étape suivante est de diminuer la taille de clique d'une unité. Commençons par détailler les cas de fin, si la taille de clique vaut 1, on a un stable et donc on prend tous les sommets. Si la taille de clique vaut 2, il ne reste plus qu'un ou deux sommets dans toute clique de voisins d'un sommet de départ. On a donc une union de cycles et de chemins puisqu'en plus de son éventuel voisin dans sa clique, chaque sommet a au plus un voisin dans une autre clique. Si l'on prend un sommet quelconque, sa clique va décroître d'une unité ; et si en plus on prend tous les sommets à distance paire dans sa composante connexe (réduite à un chemin ou un cycle), on a bien choisi un stable, en ne laissant qu'un autre stable. (En pratique, on peut donc créer les deux dernières parties en une étape.)

Pour les étapes intermédiaires, pour chaque composante connexe du graphe restant à l'étape courante, on choisit un sommet x :

— qui n'a pas encore été pris,
— qui appartient à une clique de voisin de taille au moins 2 dans le graphe courant,
— qui n'est pas un point d'articulation du graphe courant. (Un point d'articulation est un sommet tel que le nombre de composantes connexes augmente si on l'enlève. Déterminer les (non-)points d'articulation d'un graphe est un des problèmes courants résolus par un parcours en profondeur en temps linéaire. Comme le nombre d'étapes est borné par le degré maximum, ce calcul annexe mais nécessaire est masqué par la notation asymptotique. Une version avec code source est disponible sur le site de l'auteur.)

De ce sommet, on fait un parcours en largeur modifié en n'allant que du côté de ses voisins dans la même clique, comme s'il n'y avait pas d'arête le reliant à une autre clique. Il se peut qu'il n'y ait de toute manière pas d'arête le reliant à une autre clique. Mais sinon, comme il n'est pas un point d'articulation, le parcours aboutira quand même à couvrir les sommets du côté de cet éventuel voisin. Tout au long de ce parcours, on ne garde que le sommet de départ et certains sommets à distance paire, dans la partie que l'on cherche à construire. Clairement, les sommets à distance 1 sont dans la même clique que le sommet de départ, les sommets à distance 2 sont quant à eux dans de nouvelles cliques, certains pouvant être dans une même clique auquel cas, on n'en garde qu'un. On voit par induction qu'un sommet à distance paire rentre toujours dans une clique, alors que les sommets à distance impaire en sortent (ce sera donc le

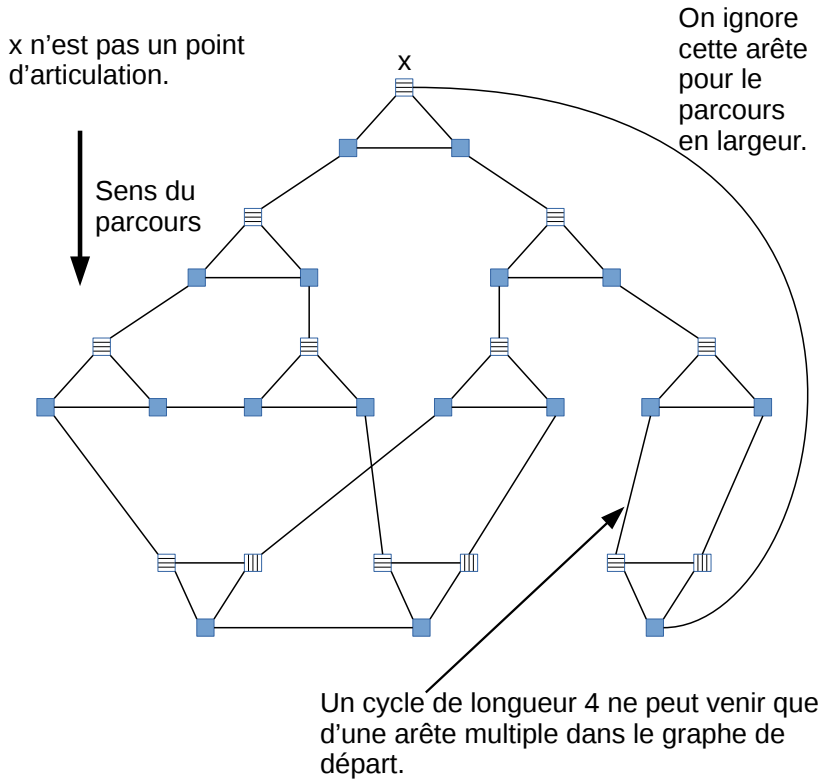


FIGURE 5 – Exemple de parcours en largeur modifié (cette fois les cliques de voisins évidées sont de taille 3 pour être distinguables)

cas de l'éventuel voisin du sommet de départ dans une autre clique). On ne peut donc avoir deux sommets adjacents à distance paire dans deux cliques distinctes. Comme le parcours passe dans toutes les cliques maximum de la composante connexe, en choisissant exactement un des sommets à distance paire par clique, on a bien la garantie de faire diminuer la taille de clique d'un.

Nous avons démontré le théorème suivant :

Théorème 4.2. *Tout graphe de degré au plus* 3 *obtenu à partir d'un graphe 3-régulier avec le théorème précédent admet une décomposition arborescente questionnable bijective équilibrée de largeur 2, de profondeur structurelle au plus* $\lceil \lg(n) \rceil + 3$ *et de profondeur logique au plus* $\lceil \lg(n) \rceil + 3 \times (\lceil \lg(4n) \rceil + 1) = \lceil \lg(n) \rceil + 3 \times (\lceil \lg(n) \rceil + 3) = \lceil \lg(n) \rceil \times 4 + 12$*. Cette décomposition peut être obtenue en temps quasi-linéaire. Le code source en annexe aidera le lecteur à voir que ce résultat se généralise aux graphes obtenus à partir de graphes de degré au plus* k *: la complexité en temps reste quasi-linéaire ; la profondeur structurelle est au plus* $\lceil \lg(n) \rceil + k$ *; et la profondeur logique est au plus* $\lceil \lg(n) \rceil + k \times (\lceil \lg((k+1) \times n) \rceil + 1) \leq \lg(n) + 1 + k \times (\lg(n) + \lg(k+1) + 2) = \lg(n) \times (k+1) + k \times \lg(k+1) + 2k + 1$.

La lectrice intéressée par le problème de la coloration des carrés pourra consulter les surveys suivants sur deux généralisations : la coloration où les sommets à distance p doivent avoir des couleurs distinctes par Kramer and Kramer (2008), et celle où les couleurs sont des entiers et l'on peut imposer une différence supérieure à 1 entre deux sommets à distance 1 ou 2 par Calamoneri (2011). (Je cite les auteurs des surveys, bien que, dans le premier cas, ils soient aussi les auteurs de la première publication sur le sujet, mais 40 ans avant.)

L'article sur le sujet le plus proche de nos résultats est sans nul doute celui de Bonamy et al. (2014) mais nos graphes sont un poil trop denses par rapport à ses résultats ; il n'y a pas de théorème pour les graphes de degré maximum 3 ; et, par exemple, la borne stricte sur le degré moyen maximum pour les graphes de degré maximum supérieur à 4 est de $\frac{7}{3}$, ce qui est déjà inférieur au degré moyen maximum de $\frac{12}{5}$ des graphes de degré maximum 4 que nous considérons comme très peu denses. Cet article liste des résultats d'un premier type dont les conditions associent degré maximum et maille d'un graphe planaire, et prouve des résultats d'un second type dont les conditions associent degré maximum et degré moyen maximum de graphes quelconques. Il rappelle aussi que les résultats du second type peuvent être convertis en résultat du premier type par la formule d'Euler. Comme les graphes non planaires mais très peu denses que nous étudions dans cet article ont une maille d'au moins 9, nous pensons qu'il serait naturel d'étudier des résultats d'un troisième type dont les conditions associent degré maximum, maille et degré moyen maximum de graphes quelconques. Ces résultats pourraient peut-être englober notre classe de graphes. De plus, l'étude de la complexité d'algorithmes effectifs pour ces résultats reste à faire.

On a donc montré que trouver un stable maximum dans la réunion de 3 parties était aisé, alors que cela devient NP-complet avec 4 parties. Pour l'algorithme de complexité « modérément exponentielle » promis, il suffit de voir que le graphe moins dense a exactement $4n$ sommets, si on avait n sommets au départ. Chacune des parties a exactement n sommets. La réunion des 3 premières parties est une union de cycles et de chemins, car tout sommet y est de degré au plus deux. Mais le point important,

c'est que l'on n'a pas d'orientation contradictoire à la quatrième étape, c'est-à-dire que l'on n'a pas pris deux sommets voisins sur le même chemin de longueur 3 obtenu à partir d'une arête. D'où pour tout sommet original, qui est nécessairement de degré 2 dans l'union des 3 premières parties, il existe un sommet original orienté vers lui dès la deuxième ou troisième étape, c'est à dire une extrémité de chemin. Ceci implique qu'il y a exactement $n/2$ chemins et donc au moins $n/2$ composantes connexes, chaque composante étant un cycle ou un chemin. Pour la troisième réunion entre les 3 premières parties et la quatrième, si l'on considère d'abord au moins $n/2-1$ sommets qui connectent tous les morceaux, on obtient un graphe n'ayant pas plus de deux chemins sommets disjoints entre deux sommets donnés. Un peu comme un arbre avec des cycles par endroits, par exemple une haltère (deux cycles reliés par un chemin). Trouver un stable maximum dans ces graphes-là est facile, l'algorithme glouton qui consiste à voir un cycle qui n'est adjacent qu'à un seul chemin que comme un gros sommet, un cycle-feuille, nous garantit de toujours avoir une vraie feuille ou un cycle-feuille à traiter. Il est facile de voir qu'il existe toujours un stable maximum qui prend une feuille donnée, donc on la prend, on enlève son sommet voisin, et on itère. De même, si l'on a un cycle-feuille, il existe forcément un stable maximum qui est optimal localement sur ce cycle et qui ne prend pas le sommet adjacent au chemin adjacent au cycle-feuille. Donc on prend ce stable maximum local, on enlève le cycle-feuille et on itère. Au final, on fait une énumération exhaustive des stables sur les $n/2+1$ sommets restants de la quatrième partie, et pour chaque choix de l'énumération, on calcule l'algorithme glouton sur ce qui reste en ayant enlevé les sommets adjacents aux sommets choisis dans le choix de l'énumération. On obtient le théorème suivant :

Théorème 4.3. *Le problème du stable maximum sur les graphes 3-réguliers peut être résolu par un algorithme de complexité $O(2^{\frac{n}{2}+1} \times n^2)$. La complexité polynomiale multiplicative de la complexité exponentielle est quadratique. La complexité polynomiale additive masquée par la notation asymptotique est quasi-linéaire.*

5 Conclusion

On voit que des principes très anciens des mathématiques, un peu remis au goût du jour, donnent des résultats sympathiques. Néanmoins, l'état de l'art sur les algorithmes modérément exponentiels pour le problème du stable maximum est plus performant, même dans le cas où le graphe n'est pas de degré borné. Pour d'autres résultats autour du principe de première différence, j'invite le lecteur intéressé à regarder les articles de Cantor (1895), Sierpiński (1932) et d'Hausdorff (1907) en théorie des ordres, la vaste littérature autour de la décomposition modulaire (dont Gallai (1967) Habib and Paul (2010) Ehrenfeucht et al. (1999) Bui-Xuan et al. (2006) Bui-Xuan (2008)) et plus modestement mes 3 articles sur le sujet (2018, 2019 et 2020), dont les deux premiers sont disponibles sur arXiv et le troisième uniquement sur ma page web.

Quant à mon interprétation théologique que sans doute Diviser n'est pas régner, je dirai juste que quand bien même il y aurait un algorithme quantique polynomial pour ce problème, avec tous ces enchevêtrements, l'algorithme quantique *relierait* les parties du tout. Et j'y verrais encore l'occasion de dire :

Merci Dieu ! Merci Père ! Merci Seigneur ! Merci Saint Esprit !

Je tiens aussi à remercier Étienne Lemaire de m'avoir suggéré d'ajouter des figures et Sylvain Perifel de m'avoir suggéré de mettre en évidence les théorèmes démontrés, afin d'améliorer l'accessibilité et la lecture de mon article. Je remercie aussi un(e) relecteur/relectrice anonyme de 1024 pour m'avoir, lui/elle aussi, suggéré d'ajouter des figures, ses corrections ortho-typographiques et ses suggestions d'améliorations rédactionnelles.

Références

M. Bonamy, B. Lévêque, and A. Pinlou. 2-distance coloring of sparse graphs. *J. Graph Theory*, 77(3) :190–218, 2014. doi : 10.1002/jgt.21782. URL https://doi.org/10.1002/jgt.21782.

B. Bui-Xuan, M. Habib, V. Limouzy, and F. de Montgolfier. Homogeneity vs. adjacency : Generalizing some graph decomposition algorithms. In F. V. Fomin, editor, *Graph-Theoretic Concepts in Computer Science, 32nd International Workshop, WG 2006, Bergen, Norway, June 22-24, 2006, Revised Papers*, volume 4271 of *Lecture Notes in Computer Science*, pages 278–288. Springer, 2006. doi : 10.1007/11917496_25. URL https://doi.org/10.1007/11917496_25.

B.-M. Bui-Xuan. *Tree-Representation of Set Families in Graph Decompositions and Efficient Algorithms*. PhD thesis, Université Montpellier II, september 2008.

T. Calamoneri. The $L(h, k)$-labelling problem : An updated survey and annotated bibliography. *Comput. J.*, 54(8) :1344–1371, 2011. doi : 10.1093/comjnl/bxr037. URL https://doi.org/10.1093/comjnl/bxr037.

G. Cantor. Beiträge zur Begründung der transfiniten Mengenlehre. *Math. Ann.*, 46 : 481–512, 1895.

A. Ehrenfeucht, T. Harju, and G. Rozenberg. *The Theory of 2-Structures - A Framework for Decomposition and Transformation of Graphs*. World Scientific Publishing Co. Pte. Ltd., 1999. ISBN 981-02-4042-2.

J. Fouquet, M. Habib, F. de Montgolfier, and J. Vanherpe. Bimodular decomposition of bipartite graphs. In J. Hromkovic, M. Nagl, and B. Westfechtel, editors, *Graph-Theoretic Concepts in Computer Science, 30th International Workshop,WG 2004, Bad Honnef, Germany, June 21-23, 2004, Revised Papers*, volume 3353 of *Lecture Notes in Computer Science*, pages 117–128. Springer, 2004. doi : 10.1007/978-3-540-30559-0_10. URL https://doi.org/10.1007/978-3-540-30559-0_10.

T. Gallai. Transitiv orientierbare graphen. *Acta Mathematica Hungarica*, 18 :25–66, 1967.

M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1(3) :237–267, 1976. ISSN 0304-3975. doi : https://doi.org/10.1016/0304-3975(76)90059-1. URL https://www.sciencedirect.com/science/article/pii/0304397576900591.

M. Habib and C. Paul. A survey of the algorithmic aspects of modular decomposition. *Comput. Sci. Rev.*, 4(1) :41–59, 2010. doi : 10.1016/j.cosrev.2010.01.001. URL https://doi.org/10.1016/j.cosrev.2010.01.001.

F. Hausdorff. Untersuchungen über Ordnungstypen V. *Ber. über die Verhandlungen der Königl. Sächs. Ges. der Wiss. zu Leipzig. Math.-phys. Klasse*, 59 :105–159, 1907.

S. Hougardy. Classes of perfect graphs. *Discrete Mathematics*, 306 :2529–2571, 2006.

F. Kramer and H. Kramer. A survey on the distance-colouring of graphs. *Discret. Math.*, 308(2-3) :422–426, 2008. doi : 10.1016/j.disc.2006.11.059. URL https://doi.org/10.1016/j.disc.2006.11.059.

L. Lyaudet. A class of orders with linear ? time sorting algorithm. *CoRR*, abs/1809.00954, 2018. URL http://arxiv.org/abs/1809.00954.

L. Lyaudet. On finite width questionable representations of orders. *CoRR*, abs/1903.02028, 2019. URL http://arxiv.org/abs/1903.02028.

L. Lyaudet. First difference principle applied to modular/questionable-width, clique-width, and rank-width of binary structures. *preprint*, 2020. URL https://lyaudet.eu/laurent/Publi/Journaux/LL2020LargeursStructuresBinaires/LargeursStructuresBinaires_v5.pdf.

W. Sierpiński. Généralisation d'un théorème de Cantor concernant les ensembles ordonnés dénombrables. *Fundamenta Mathematicae*, 18 :280–284, 1932.

Annexe

Cette annexe contient tout d'abord la classification partielle de la classe des graphes ≪ dédensifiés puis carrés puis évidés ≫, c'est-à-dire des graphes obtenus à partir d'un graphe quelconque en remplaçant chaque arête par un chemin de longueur 3, puis en reliant entre elles toutes les paires de sommets à distance au plus 2, et enfin en supprimant les sommets du graphe original et les arêtes incidentes à ces sommets. Cette classe de graphes, notée $\mathcal{DCE}$, fait partie des graphes parfaits et nous avons tenté de déterminer les relations ensemblistes entre cette classe et les 120 sous-classes de graphes parfaits listées dans Hougardy (2006). Nous reprenons les mêmes sigles dénotant certains petits graphes particuliers, et leur présence dans une cellule du tableau ci-dessus dénote leur appartenance à l'ensemble correspondant. L'article cité étant en accès libre, tout lecteur intéressé y trouvera les informations nécessaires à la compréhension de ce qui suit. Nous espérons finir cette classification en rédigeant d'avantages d'explications au

cours de l'année 2023. Ce sera très probablement la principale cause de mise à jour de cet article dans le futur.

Sous-classe de graphes parfaits $\mathcal{C}$	$\mathcal{C} \cap \mathcal{DCE}$	$\mathcal{C} \setminus \mathcal{DCE}$	$\mathcal{DCE} \setminus \mathcal{C}$
alternately colorable	P_2	$2\mathrm{P}_1, \mathrm{P}_3, \mathrm{K}_3$	$\emptyset$
alternately orientable	P_2	$2\mathrm{P}_1, \mathrm{P}_3, \mathrm{K}_3$	F_{68}
AT-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	C_6
BIP*	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	F_{68}
bipartite	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	F_{15}
brittle	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	P_4
bull-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	F_{15}
C_4-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3, \mathrm{K}_3$	$\emptyset$ si on part de graphes sans arêtes multiples, sinon C_4
chair-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	$\emptyset$
clique separable	P_2	P_3	F_{68}, voire $\overline{\mathrm{C}_6}$ si on s'autorise à partir de graphes avec des arêtes triples
cograph contraction	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	C_6, ou $\overline{\mathrm{C}_6}$ si…
comparability	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	F_{15}, ou $\overline{\mathrm{C}_6}$ si…
$\Delta \leq 6$ Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	$\emptyset$ si on part de graphes de degré maximum 6, sinon tout graphe obtenu à partir d'un graphe de degé maximum supérieur à 6
dart-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	$\emptyset$
degenerate Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	$\emptyset$
diamond-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	$\emptyset$
doc-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	$\emptyset$
elementary	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	$\emptyset$
forest	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	C_6, ou C_4 si…
gem-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	$\emptyset$
HHD-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	C_6
Hoàng	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	F_{15}, ou $\overline{\mathrm{C}_6}$ si…
i-triangulated ou Gallaï	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	C_6
I_4-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	P_8, C_8
interval	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	C_6, ou C_4 si…
K_4-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	$\emptyset$ si on part de graphes de degré maximum 3, sinon tout graphe obtenu à partir d'un graphe de degé maximum supérieur à 3
(K_5, P_5)-free Berge	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	P_6
LGBIP	P_2	$2\mathrm{P}_1, \mathrm{P}_3$	$\emptyset$

Sous-classe de graphes parfaits $\mathcal{C}$	$\mathcal{C} \cap \mathcal{DCE}$	$\mathcal{C} \setminus \mathcal{DCE}$	$\mathcal{DCE} \setminus \mathcal{C}$
line perfect	P_2	$2P_1, P_3$	F_{68}, ou $\overline{C_6}$ si…
locally perfect	P_2	$2P_1, P_3$	?
Meyniel	P_2	$2P_1, P_3$	F_{68}
murky	P_2	$2P_1, P_3$	P_6
1-overlap bipartite	?	?	?
opposition	P_2	$2P_1, P_3$	C_6
P_4-free	P_2	$2P_1, P_3$	P_4
P_4-lite	P_2	$2P_1, P_3$	P_6, C_6
P_4-reducible	P_2	$2P_1, P_3$	P_6, C_6
P_4-sparse	P_2	$2P_1, P_3$	P_6, C_6
P_4-stable Berge	P_2	$2P_1, P_3$	F_{68}, ou $\overline{C_6}$ si…
parity	P_2	$2P_1, P_3$	F_{68}
partner-graph triangle free	P_2	$2P_1, P_3$	F_{68}, ou $\overline{C_6}$ si…
paw-free Berge	P_2	$2P_1, P_3$	F_{15}, ou $\overline{C_6}$ si…
perfectly contractile	P_2	$2P_1, P_3$	F_{68}, ou $\overline{C_6}$ si…
perfectly orderable	P_2	$2P_1, P_3$	F_{68}, ou $\overline{C_6}$ si…

Nous donnons à présent le code source PHP des algorithmes décrits dans cet article avec l'exemple de graphe 3-régulier. Cela permettra au lecteur de vérifier sur des algorithmes concrets la correction et les complexités annoncées, et éventuellement d'avoir matière à corriger/déboguer. Les fonctions sont entrelacées avec la définition du graphe d'exemple, des invocations de ces fonctions et des invocations d'affichage de traces. Ainsi, le lecteur intéressé pourra découvrir les fonctions et un exemple de résultat dans la foulée. Seule la complexité asymptotique du code source a été optimisée. L'accent a plus été mis sur un nommage explicite des structures et des commentaires explicatifs. PHP est un langage de programmation interprété, mais qui a d'excellentes performances pour sa catégorie de langages. Une implémentation dans un langage connu pour produire des exécutables vraiment optimisés reste à faire ; nous espérons néanmoins que le travail didactique réalisé sur le code PHP rendra cette tâche relativement aisée même si un peu longue.

Il est possible de télécharger ce code source sur ma page web :
`https://lyaudet.eu/laurent/Publi/Journaux/LL2022DiviserNestPasRegner/decompositionArborescenteQuestionnable_v5.txt`.

Le fichier a été renommé de .php à .txt et la balise ouvrante PHP retirée pour que le serveur web puisse le restituer sans l'exécuter. Le lecteur doit remettre la balise ouvrante PHP pour pouvoir l'exécuter.

```
<?php
/*
This file is source code accompanying "Diviser n'est pas régner ?"
scientific article.

This source code is free software: you can redistribute it
and/or modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation,
either version 3 of the License,
or (at your option) any later version.

This source code is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU Lesser General Public License for more details.

A copy of the GNU Lesser General Public License is not included.
Please see <http://www.gnu.org/licenses/>.

©Copyright 2022 Laurent Lyaudet
*/



$grapheDeTest = [
  "listeDeSommets" => [
```

```
    "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k",
    "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v"
  ],
  "listeDesAretes" => [
    ["a", "b"], ["a", "c"], ["a", "d"],
    ["b", "d"], ["b", "e"],
    ["c", "d"], ["c", "f"],
    ["e", "h"], ["e", "k"],
    ["f", "i"], ["f", "l"],
    ["g", "h"], ["g", "i"], ["g", "j"],
    ["h", "j"],
    ["i", "j"],
    ["k", "l"], ["k", "q"],
    ["l", "r"],
    ["m", "n"], ["m", "o"], ["m", "p"],
    ["n", "p"], ["n", "q"],
    ["o", "p"], ["o", "r"],
    ["q", "t"],
    ["r", "u"],
    ["s", "t"], ["s", "u"], ["s", "v"],
    ["t", "v"],
    ["u", "v"],
  ]
];



function chargerUnGraphe($graphe){
  $graphe["nombreDeSommets"] = count($graphe["listeDeSommets"]);
  $sommetsDedupliques = array_unique($graphe["listeDeSommets"]);
  if($graphe["nombreDeSommets"] != count($sommetsDedupliques)){
    throw new Exception(
      "Il y a des sommets dupliqués dans la liste."
    );
  }

  // La liste de sommets correspond aussi avec PHP
  // avec l'application/bijection de l'identifiant
  // interne entier d'un sommet vers son libellé.
  // Pour parser la liste d'arêtes,
  // on veut aussi la bijection réciproque.
  $graphe["etiquetteDeSommetVersIdentifiantInterne"] = array_flip(
    $graphe["listeDeSommets"]
  );
  // Copie de tableau par référence.
  // On veut juste gagner un peu sur la longueur des lignes de code.
```

```
  $etiquetteVersIdentifiantInterne
    = &$graphe["etiquetteDeSommetVersIdentifiantInterne"]
  ;

  // On initialise des listes d'adjacences vides.
  $graphe["listesDadjacences"] = [];
  $listesDadjacences = &$graphe["listesDadjacences"];
  for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
    $listesDadjacences[$i] = [];
  }
  // Puis on les remplit.
  foreach($graphe["listeDesAretes"] as $arete){
    if(!isset($etiquetteVersIdentifiantInterne[$arete[0]])){
      throw new Exception(
        "Le sommet ".$arete[0]." est référencé dans une arête"
       ." mais n'est pas listé dans les sommets."
      );
    }
    $idDuSommet1 = $etiquetteVersIdentifiantInterne[$arete[0]];
    if(!isset($etiquetteVersIdentifiantInterne[$arete[1]])){
      throw new Exception(
        "Le sommet ".$arete[1]." est référencé dans une arête"
       ." mais n'est pas listé dans les sommets."
      );
    }
    $idDuSommet2 = $etiquetteVersIdentifiantInterne[$arete[1]];
    $listesDadjacences[$idDuSommet1] []= $idDuSommet2;
    $listesDadjacences[$idDuSommet2] []= $idDuSommet1;
  }

  // On calcule le degré minimum et le degré maximum du graphe
  $graphe["degreMinimum"] = count($listesDadjacences[0]);
  $graphe["degreMaximum"] = count($listesDadjacences[0]);
  for($i = 1; $i < $graphe["nombreDeSommets"]; ++$i){
    $graphe["degreMinimum"] = min(
      $graphe["degreMinimum"],
      count($listesDadjacences[$i])
    );
    $graphe["degreMaximum"] = max(
      $graphe["degreMaximum"],
      count($listesDadjacences[$i])
    );
  }
  return $graphe;
}
```

```
$grapheCharge = chargerUnGraphe($grapheDeTest);
//var_dump($grapheCharge);
echo(
  "Chargement d'un graphe à ".$grapheCharge["nombreDeSommets"]
 ." sommets"
 ." de degré minimum ".$grapheCharge["degreMinimum"]
 ." et de degré maximum ".$grapheCharge["degreMaximum"].".\n"
);



function listeDeSommetsInternePourAffichage($graphe, $liste){
  $identifiantVersEtiquette = $graphe["listeDeSommets"];
  $listeDeSommetsEtiquetes = [];
  foreach($liste as $sommet){
    $listeDeSommetsEtiquetes []=
      $identifiantVersEtiquette[$sommet]
    ;
  }
  return $listeDeSommetsEtiquetes;
}



/**
 * Fonction qui calcule les boules de rayon 2
 * centrées sur tous les sommets d'un graphe.
 * Cette fonction a une complexité dans le pire cas
 * par rapport au nombre de sommets :
 * - linéaire en temps et en espace sur les graphes de degré borné,
 *   sur lesquels l'article se focalise,
 * - quadratique en espace et cubique en temps
 *   sur les graphes en général.
 * Les algorithmes à base de multiplication de matrice
 * permettent de faire mieux dans le cas des graphes denses.
 */
function calculerLesBoulesDeRayon2($graphe){
  // Une application qui associe à un sommet
  // la boule de rayon 2 (liste de sommets)
  // centrée sur le sommet.
  $graphe["sommetVersBouleDeRayon2"] = [];

  // Pour chaque sommet (facteur n)
  for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
```

```
    // On calcule la liste des sommets
    // dans la boule de rayon 2 centrée sur ce sommet
    // (facteur constant si le degré maximum est borné)
    $voisins = $graphe["listesDadjacences"][$i];
    $voisinsDeVoisins = [];
    foreach($voisins as $voisin){
      $voisinsDeVoisins = array_merge(
        $voisinsDeVoisins,
        $graphe["listesDadjacences"][$voisin]
      );
    }
    $boule = array_unique(array_merge($voisins, $voisinsDeVoisins));
    $graphe["sommetVersBouleDeRayon2"][$i] = $boule;
  }
  return $graphe;
}



/**
 * Fonction de décomposition d'un graphe en parties
 * de sommets à distance au moins 3 par un algorithme glouton.
 */
function partitionnerEnPartiesDeSommetsADistanceAuMoins3($graphe){
  if(!isset($graphe["sommetVersBouleDeRayon2"])){
    $graphe = calculerLesBoulesDeRayon2($graphe);
  }

  // ** = puissance donc ici degré max au carré
  $nombrePartiesMax = 1 + $graphe["degreMaximum"]**2;
  // On initialise un tableau donnant la partie
  // associée à chaque sommet
  $graphe["partiesEspaceesDe3"] = [
      "identifiantDeSommetVersPartie" => [],
      "listeDesParties" => [],
  ];
  $sommetVersPartie
    = &$graphe["partiesEspaceesDe3"]["identifiantDeSommetVersPartie"]
  ;
  // -1 correspond à "non affecté"
  for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
    $sommetVersPartie[$i] = -1;
  }

  // Pour chaque sommet (facteur n)
  for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
```

```
  $boule = $graphe["sommetVersBouleDeRayon2"][$i];

  // Pour chaque partie
  // (facteur constant si le degré maximum est borné)
  for($j = 0; $j < $nombrePartiesMax; ++$j){
    // On regarde si un sommet de la boule y est déjà affecté.
    // $partieDejaPrise = false;
    foreach($boule as $sommetDansLaBoule){
      if($sommetVersPartie[$sommetDansLaBoule] === $j){
        // $partieDejaPrise = true;
        continue 2; // On passe directement à la partie suivante
      }
    }
    $sommetVersPartie[$i] = $j;
    break;
  }
}

// On regarde de combien de parties on a réellement eu besoin
$indexMaximumDePartieUtilisee = 0;
for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
  if($sommetVersPartie[$i] < 0){
    // Cela ne devrait pas se produire,
    // sauf si quelqu'un injecte un graphe
    // avec un degré maximum faux.
    throw new Exception(
      "Le sommet ".$graphe["listeDeSommets"][$i]
     ." n'a pas de partie."
    );
  }
  $indexMaximumDePartieUtilisee = max(
    $indexMaximumDePartieUtilisee,
    $sommetVersPartie[$i]
  );
}

// On remplit les parties
$listeDesParties
  = &$graphe["partiesEspaceesDe3"]["listeDesParties"]
;
for($i = 0; $i <= $indexMaximumDePartieUtilisee; ++$i){
  $listeDesParties[$i] = [];
}
for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
  $partie = $sommetVersPartie[$i];
  $listeDesParties[$partie] []= $i;
```

```
  }

  return $graphe;
}



$grapheAvecParties = partitionnerEnPartiesDeSommetsADistanceAuMoins3(
  $grapheCharge
);
/*
var_dump(
  $grapheAvecParties["partiesEspaceesDe3"]
  ["identifiantDeSommetVersPartie"]
);
//*/
$partiesEspaceesDe3 = &$grapheAvecParties["partiesEspaceesDe3"];
foreach(
  $grapheAvecParties["sommetVersBouleDeRayon2"]
  as $i => $boule
){
  echo(
    "La boule de rayon 2 centrée sur ".(
      $grapheAvecParties["listeDeSommets"][$i]
    )
   ." contient les sommets suivants : ".join(
      ",",
      listeDeSommetsInternePourAffichage(
        $grapheAvecParties,
        $boule
      )
    ).".\n"
  );
}
$listeDesParties
  = &$partiesEspaceesDe3["listeDesParties"]
;
foreach($listeDesParties as $i => $partie){
  echo(
    "La partie ".($i+1)." contient les ".count($partie)
   ." sommets suivants : ".join(
      ",",
      listeDeSommetsInternePourAffichage(
        $grapheAvecParties,
        $partie
      )
```

```
    ).".\n"
  );
}



/**
 * Fonction de vérification qu'un ensemble forme un stable.
 */
function verifierQueDesSommetsFormentUnStable(
  $graphe,
  $listeDeSommetsDuStable
){
  $tableauUnFlagParSommet1 = []; // pour les sommets du stable
  $tableauUnFlagParSommet2 = []; // pour les voisins de ces sommets
  for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
    // temps linéaire
    $tableauUnFlagParSommet1[$i] = false;
    $tableauUnFlagParSommet2[$i] = false;
  }
  foreach($listeDeSommetsDuStable as $sommet){
    // temps linéaire
    $tableauUnFlagParSommet1[$sommet] = true;
  }

  foreach($listeDeSommetsDuStable as $sommet){
    // facteur n
    $voisins = $graphe["listesDadjacences"][$sommet];
    foreach($voisins as $voisin){
      // facteur constant si le degré maximum est borné
      $tableauUnFlagParSommet2[$voisin] = true;
    }
  }

  for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
    // temps linéaire
    if($tableauUnFlagParSommet1[$i] && $tableauUnFlagParSommet2[$i]){
      return false;
    }
  }
  return true;
}



foreach($listeDesParties as $i => $partie){
```

```
  $estStable = verifierQueDesSommetsFormentUnStable(
    $grapheAvecParties,
    $partie
  );
  if(!$estStable){
    throw new Exception(
      "La partie ".($i+1)." contenant les sommets suivants : ".join(
        ",",
        listeDeSommetsInternePourAffichage(
          $grapheAvecParties,
          $partie
        )
      )." n'est pas un stable.\n"
    );
  }
}



function creerUnNouveauNoeudDeDecomposition($sommets){
  if(count($sommets) === 1){
    return [
      "typeDeNoeud" => "feuille",
      "sommet" => $sommets[0],
    ];
  }
  return [
    "typeDeNoeud" => "interne",
    "sommets" => $sommets,
    "suiteDApplications" => [],
    "applicationFilsGaucheFilsDroit" => null,
    "filsGauche" => null,
    "filsDroit" => null,
  ];
}



/**
 * Fonction de calcul de la décomposition arborescente
 * questionnable bijective équilibrée d'un stable.
 * Temps et espace quasi-linéaire en le nombre de sommets.
 */
function decomposerUnStableParPremiereDifference(
  $listeDeSommetsDuStable
){
```

```
if(count($listeDeSommetsDuStable) < 1){
  throw new Exception("Pas de décomposition de stable vide");
}
if(count($listeDeSommetsDuStable) === 1){
  return creerUnNouveauNoeudDeDecomposition(
    $listeDeSommetsDuStable
  );
}

$racine = creerUnNouveauNoeudDeDecomposition(
  $listeDeSommetsDuStable
);
$chunks = array_chunk(
  $listeDeSommetsDuStable,
  ceil(count($listeDeSommetsDuStable) / 2)
);
$listeDeSommetsDuSousStable1 = $chunks[0];
$listeDeSommetsDuSousStable2 = $chunks[1];
// logique entre guillemet,
// voir la définition des profondeurs logique
// et structurelle de la décomposition
$applicationLogique = [];
foreach($listeDeSommetsDuSousStable1 as $sommet){
  $applicationLogique[$sommet] = 'g';
}
foreach($listeDeSommetsDuSousStable2 as $sommet){
  $applicationLogique[$sommet] = 'd';
}
$applicationStructurelle = [];
foreach($listeDeSommetsDuSousStable1 as $sommet){
  $applicationStructurelle[$sommet] = 'G';
}
foreach($listeDeSommetsDuSousStable2 as $sommet){
  $applicationStructurelle[$sommet] = 'D';
}

$racine["suiteDApplications"] []= $applicationLogique;
$racine["applicationFilsGaucheFilsDroit"] =
  $applicationStructurelle
;
$racine["filsGauche"] = decomposerUnStableParPremiereDifference(
  $listeDeSommetsDuSousStable1
);
$racine["filsDroit"] = decomposerUnStableParPremiereDifference(
  $listeDeSommetsDuSousStable2
```

```
  );

  return $racine;
}



/**
 * Fonction de calcul de la décomposition arborescente
 * questionnable bijective équilibrée d'un graphe de degré borné.
 * En fait ça marche pour tous les graphes,
 * mais la décomposition n'est pas nécessairement équilibrée
 * si le degré n'est pas borné.
 * Cette fonction et les fonctions qu'elle appelle implémentent
 * un algorithme de complexité quasi-linéaire
 * démontrant le Théorème 3.1.
 */
function decomposerUnGrapheParPremiereDifference(
  $graphe
){
  if(isset($graphe["partiesEspaceesDe3"])){
    $grapheAvecParties = &$graphe;
  }
  else{
    $grapheAvecParties
      = partitionnerEnPartiesDeSommetsADistanceAuMoins3(
        $graphe
    );
  }

  $grapheAvecParties["decompositionParPremiereDifference"] = [
    "decompositionDesParties" => [],
    "decompositionGlobale" => null,
    "sommetVersChaineBinaire" => [],
    "sommetVersChaineQuaternaire" => [],
    // Sénaire : à six chiffres/caractères
    "sommetVersChaineSenaire" => [],
  ];
  foreach(
    $graphe["partiesEspaceesDe3"]["listeDesParties"]
    as $partie
  ){
    $grapheAvecParties["decompositionParPremiereDifference"]
        ["decompositionDesParties"]
      []= decomposerUnStableParPremiereDifference(
        $partie
```

```
  );
}

$decompositionPartielle
  = $grapheAvecParties["decompositionParPremiereDifference"]
    ["decompositionDesParties"][0]
;
$sommetsGauche =
  $graphe["partiesEspaceesDe3"]["listeDesParties"][0]
;
$tableauUnFlagParSommetAGauche = [];
$tableauUnFlagParSommetAGaucheRegroupeDansUneEtoile = [];
for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
  // temps linéaire
  $tableauUnFlagParSommetAGauche[$i] = false;
  $tableauUnFlagParSommetAGaucheRegroupeDansUneEtoile[$i] = false;
}
foreach($sommetsGauche as $sommet){
  // temps linéaire (sous-ensemble de sommets)
  $tableauUnFlagParSommetAGauche[$sommet] = true;
}
for(
  $i = 1,
  $iMax = count(
    $graphe["partiesEspaceesDe3"]["listeDesParties"]
  );
  $i < $iMax;
  ++$i
){
  // Dans le cas des graphes de degré borné,
  // cette boucle est exécutée un nombre borné de fois.
  // On joint la décomposition partielle et la partie courante
  $partieCourante
    = $graphe["partiesEspaceesDe3"]["listeDesParties"][$i]
  ;
  $sommetsACetteEtape = array_merge(
    $sommetsGauche,
    $partieCourante
  );
  $racine = creerUnNouveauNoeudDeDecomposition(
    $sommetsACetteEtape
  );
  // La partie structurelle est immédiate et peu intéressante
  $applicationStructurelle = [];
  foreach($sommetsGauche as $sommet){
    // temps linéaire un nombre borné de fois
```

```
  // (nombre de parties - 1)
  // dans le cas des graphes de degré borné
  $applicationStructurelle[$sommet] = 'G';
}
foreach($partieCourante as $sommet){
  // temps linéaire (sous-ensemble de sommets disjoint)
  $applicationStructurelle[$sommet] = 'D';
}
$racine["applicationFilsGaucheFilsDroit"] =
  $applicationStructurelle
;
$racine["filsGauche"] = $decompositionPartielle;
$racine["filsDroit"]
  = $grapheAvecParties["decompositionParPremiereDifference"]
    ["decompositionDesParties"][$i]
;

// Calcul de la suite d'applications
// 1) calcul des composantes connexes du graphe biparti
// On exploite le fait que l'on sait déjà que ce sont des sommets
// isolés ou des étoiles centrées à droite
foreach($sommetsGauche as $sommet){
  // temps linéaire un nombre borné de fois si degré borné
  $tableauUnFlagParSommetAGaucheRegroupeDansUneEtoile[$sommet]
    = false
  ;
}
$etoilesOuIsole = [];
foreach($partieCourante as $sommet){
  // temps linéaire (sous-ensemble de sommets disjoint)
  // fois une constante si degré borné
  $composanteConnexe = [$sommet];
  $voisins = $graphe["listesDadjacences"][$sommet];
  foreach($voisins as $voisin){
    if($tableauUnFlagParSommetAGauche[$voisin]){
      $tableauUnFlagParSommetAGaucheRegroupeDansUneEtoile
          [$voisin]
        = true
      ;
      $composanteConnexe []= $voisin;
    }
  }
  $etoilesOuIsole []= $composanteConnexe;
}
foreach($sommetsGauche as $sommet){
  // temps linéaire un nombre borné de fois si degré borné
```

```
  if(
    !$tableauUnFlagParSommetAGaucheRegroupeDansUneEtoile[$sommet]
  ){
    $etoilesOuIsole []= [$sommet];
  }
}
// 2) On exploite les index entiers des composantes connexes
// pour avoir gratuitement les applications
$iBitSelectionne = 1;
$nombreDeComposantes = count($etoilesOuIsole);
$indexMax = $nombreDeComposantes - 1;
while($iBitSelectionne <= $indexMax){
  // nombre logarithmique de fois un traitement linéaire
  $applicationLogique = [];

  foreach($etoilesOuIsole as $index => $composante){
    if($index & $iBitSelectionne){
      foreach($composante as $sommet){
        // chaque sommet n'est vu qu'au plus une fois
        // par les 3 boucles foreach pour une exécution
        // de la boucle while
        $applicationLogique[$sommet] = 'd';
      }
    }
    else{
      foreach($composante as $sommet){
        // chaque sommet n'est vu qu'au plus une fois
        // par les 3 boucles foreach pour une exécution
        // de la boucle while
        $applicationLogique[$sommet] = 'g';
      }
    }
  }
  $racine["suiteDApplications"] []= $applicationLogique;

  // On multiplie par 2
  $iBitSelectionne = $iBitSelectionne << 1;
}

// Ajout de l'application logique finale
$applicationLogique = [];
foreach($sommetsGauche as $sommet){
  // lettre après g plutôt que g' pour rester à taille fixe
  $applicationLogique[$sommet] = 'h';
}
foreach($partieCourante as $sommet){
```

```
    // lettre après d plutôt que d' pour rester à taille fixe
    $applicationLogique[$sommet] = 'e';
  }
  $racine["suiteDApplications"] []= $applicationLogique;

  // Préparation des données pour l'étape suivante
  $decompositionPartielle = $racine;
  $sommetsGauche = $sommetsACetteEtape;
  foreach($partieCourante as $sommet){
    // temps linéaire (sous-ensemble de sommets disjoint)
    $tableauUnFlagParSommetAGauche[$sommet] = true;
  }
}
$grapheAvecParties["decompositionParPremiereDifference"]
  ["decompositionGlobale"] = &$racine;

for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
  // Pour chaque sommet, on concatène les caractères liés.
  // Le temps par sommet est logarithmique,
  // si la profondeur logique de la décomposition
  // est logarithmique, par exemple si le degré est borné.
  $motSenaire = "";
  $noeudDecompositionCourant = $racine;
  while($noeudDecompositionCourant["typeDeNoeud"] !== "feuille"){
    // On parcourt la suite d'applications à l'envers
    // puisque l'on part de la racine.
    for(
      $j = count(
        $noeudDecompositionCourant["suiteDApplications"]
      ) - 1;
      $j >= 0;
      --$j
    ){
      $application
        = $noeudDecompositionCourant["suiteDApplications"][$j]
      ;
      if(!isset($application[$i])){
        var_dump($i, $noeudDecompositionCourant, $application);
        throw new Exception("Un sommet isolé a dû se perdre.");
      }
      $motSenaire .= $application[$i];
    }
    $bifurcation
      = $noeudDecompositionCourant
        ["applicationFilsGaucheFilsDroit"][$i]
    ;
```

```
      $motSenaire .= $bifurcation;
      if($bifurcation === 'G'){
        $noeudDecompositionCourant
          = $noeudDecompositionCourant["filsGauche"]
        ;
      }
      else{
        $noeudDecompositionCourant
          = $noeudDecompositionCourant["filsDroit"]
        ;
      }
    }

    // On renverse la chaîne (de longueur logarithmique)
    $motSenaire = strrev($motSenaire);
    $motBinaire = strtr($motSenaire, "ghGdeD", "000111");
    $motQuaternaire = strtr($motSenaire, "he", "gd");
    $grapheAvecParties["decompositionParPremiereDifference"]
        ["sommetVersChaineBinaire"][$i]
      = $motBinaire
    ;
    $grapheAvecParties["decompositionParPremiereDifference"]
        ["sommetVersChaineQuaternaire"][$i]
      = $motQuaternaire
    ;
    $grapheAvecParties["decompositionParPremiereDifference"]
        ["sommetVersChaineSenaire"][$i]
      = $motSenaire
    ;
  }
  return $grapheAvecParties;
}



$grapheDecompose = decomposerUnGrapheParPremiereDifference(
  $grapheAvecParties
);
/*
var_dump(
  $grapheDecompose["decompositionParPremiereDifference"]
);
//*/
foreach($grapheDecompose["listeDeSommets"] as $i => $sommet){
  echo(
    "Le sommet $sommet est representé par la chaîne "
```

```
    .$grapheDecompose["decompositionParPremiereDifference"]
        ["sommetVersChaineSenaire"][$i]
    .".\n"
  );
}



/**
 * Dédensifier un graphe en remplaçant chaque arête
 * par un chemin de longueur 3.
 * Il suffit de savoir que les tableaux associatifs de PHP
 * sont des tables de hashage et de constater qu'il n'y a pas
 * de boucles imbriquées sur les données pour savoir
 * que cette fonction prouve la complexité linéaire annoncée
 * au Théorème 4.1.
 */
function dedensifierUnGraphe($graphe){
  // On commence par vérifier qu'aucun sommet n'a une étiquette
  // contenant la sous-chaîne "_Vers_",
  // car on va l'utiliser pour l'étiquetage des nouveaux sommets.
  foreach($graphe["listeDeSommets"] as $sommet){
    if(strpos($sommet, "_Vers_") !== false){
      throw new Exception(
        "L'étiquette $sommet contient la sous-chaîne _Vers_."
      );
    }
  }

  $graphe["nombreDeSommetsAvantDedensification"]
    = $graphe["nombreDeSommets"]
  ;
  $graphe["nombreDeSommets"]
    = $graphe["nombreDeSommets"]
    + 2 * count($graphe["listeDesAretes"])
  ;
  $graphe["listeDesAretesAvantDedensification"]
    = $graphe["listeDesAretes"]
  ;
  $graphe["listesDadjacencesAvantDedensification"]
    = $graphe["listesDadjacences"]
  ;
  // On calcule le degré minimum et le degré maximum du graphe
  $graphe["degreMinimumAvantDedensification"]
    = $graphe["degreMinimum"]
  ;
```

```
$graphe["degreMaximumAvantDedensification"]
  = $graphe["degreMaximum"]
;
if(count($graphe["listeDesAretesAvantDedensification"]) > 0){
  $graphe["degreMinimum"] = min(2, $graphe["degreMinimum"]);
  $graphe["degreMaximum"] = max(2, $graphe["degreMaximum"]);
}

$etiquetteVersIdentifiantInterne
  = &$graphe["etiquetteDeSommetVersIdentifiantInterne"]
;

// On convient que si x-y devient x-x_Vers_y-y_Vers_x-y,
// alors cette fonction sera appelée la projection
// d'une arête initiale vers un chemin de longueur 3.
$graphe["projectionAreteVersChemin"] = [];
// On inventorie aussi les chemins associés à un sommet
$graphe["sommetVersChemins"] = [];
// Ces chemins sont stockés sous la forme d'une liste de 4 sommets,
// ainsi que des involutions suivantes :
// visAVis : x est y sont leurs vis-à-vis respectifs,
// idem pour x_Vers_y et y_Vers_x.
// proche : x et x_Vers_y sont proches, idem pour y et y_Vers_x.
// lointain : x et y_Vers_x sont lointains,
// idem pour y et x_Vers_y.
$graphe["cheminsDeLongueur3"] = [];

$graphe["sommetVersIdDeClique"] = [];


//Armé de ces notions, on remplit tout.
$graphe["listeDesAretes"] = [];
foreach(
  $graphe["listeDesAretesAvantDedensification"] as $i => $arete
){
  $x = $arete[0];
  $y = $arete[1];
  // On crée les deux nouveaux sommets et on les enregistre.
  $x_Vers_y__etiquette = $x."_Vers_".$y;
  $y_Vers_x__etiquette = $y."_Vers_".$x;
  $graphe["listeDeSommets"] []= $x_Vers_y__etiquette;
  $etiquetteVersIdentifiantInterne[$x_Vers_y__etiquette]
    = count($graphe["listeDeSommets"]) - 1
  ;
  $graphe["listeDeSommets"] []= $y_Vers_x__etiquette;
  $etiquetteVersIdentifiantInterne[$y_Vers_x__etiquette]
```

```
  = count($graphe["listeDeSommets"]) - 1
;
$x__id = $etiquetteVersIdentifiantInterne[$x];
$y__id = $etiquetteVersIdentifiantInterne[$y];
$x_Vers_y__id
  = $etiquetteVersIdentifiantInterne[$x_Vers_y__etiquette]
;
$y_Vers_x__id
  = $etiquetteVersIdentifiantInterne[$y_Vers_x__etiquette]
;
$chemin = [
  "areteDeDepart" => $i,
  "sommets" => [
    $x__id,
    $x_Vers_y__id,
    $y_Vers_x__id,
    $y__id,
  ],
  "visAVis" => [
    $x__id => $y__id,
    $y__id => $x__id,
    $x_Vers_y__id => $y_Vers_x__id,
    $y_Vers_x__id => $x_Vers_y__id,
  ],
  "proche" => [
    $x__id => $x_Vers_y__id,
    $x_Vers_y__id => $x__id,
    $y__id => $y_Vers_x__id,
    $y_Vers_x__id => $y__id,
  ],
  "lointain" => [
    $x__id => $y_Vers_x__id,
    $y_Vers_x__id => $x__id,
    $y__id => $x_Vers_y__id,
    $x_Vers_y__id => $y__id,
  ],
];
$graphe["cheminsDeLongueur3"] []= $chemin;
$graphe["projectionAreteVersChemin"][$i]
  // stocker l'id plutôt que le pointeur ne sert à rien,
  // c'est l'identité
  // = count($graphe["cheminsDeLongueur3"]) - 1
  = $chemin
;
// On ajoute les liens des sommets vers les chemins.
if(!isset($graphe["sommetVersChemins"][$x__id])){
```

```
    $graphe["sommetVersChemins"][$x__id] = [];
  }
  $graphe["sommetVersChemins"][$x__id] []= $chemin;
  if(!isset($graphe["sommetVersChemins"][$y__id])){
    $graphe["sommetVersChemins"][$y__id] = [];
  }
  $graphe["sommetVersChemins"][$y__id] []= $chemin;
  // Les sommets internes ne sont concernés que par un chemin.
  $graphe["sommetVersChemins"][$x_Vers_y__id] = [$chemin];
  $graphe["sommetVersChemins"][$y_Vers_x__id] = [$chemin];

  // On crée les nouvelles arêtes
  $graphe["listeDesAretes"] []= [$x, $x_Vers_y__etiquette];
  $graphe["listeDesAretes"]
    []= [$x_Vers_y__etiquette, $y_Vers_x__etiquette]
  ;
  $graphe["listeDesAretes"] []= [$y_Vers_x__etiquette, $y];
  $indiceMaxArete = count($graphe["listeDesAretes"]) - 1;
  $chemin["aretesDArrivee"] = [
    $indiceMaxArete - 2,
    $indiceMaxArete - 1,
    $indiceMaxArete
  ];
  $graphe["sommetVersIdDeClique"][$x__id] = $x__id;
  $graphe["sommetVersIdDeClique"][$x_Vers_y__id] = $x__id;
  $graphe["sommetVersIdDeClique"][$y_Vers_x__id] = $y__id;
  $graphe["sommetVersIdDeClique"][$y__id] = $y__id;
}


// On initialise des listes d'adjacences vides.
$graphe["listesDadjacences"] = [];
$listesDadjacences = &$graphe["listesDadjacences"];
for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
  $listesDadjacences[$i] = [];
}
// Puis on les remplit.
foreach($graphe["listeDesAretes"] as $arete){
  if(!isset($etiquetteVersIdentifiantInterne[$arete[0]])){
    throw new Exception(
      "Le sommet ".$arete[0]." est référencé dans une arête"
     ." mais n'est pas listé dans les sommets."
    );
  }
  $idDuSommet1 = $etiquetteVersIdentifiantInterne[$arete[0]];
  if(!isset($etiquetteVersIdentifiantInterne[$arete[1]])){
```

```
      throw new Exception(
        "Le sommet ".$arete[1]." est référencé dans une arête"
       ." mais n'est pas listé dans les sommets."
      );
    }
    $idDuSommet2 = $etiquetteVersIdentifiantInterne[$arete[1]];
    $listesDadjacences[$idDuSommet1] []= $idDuSommet2;
    $listesDadjacences[$idDuSommet2] []= $idDuSommet1;
  }

  return $graphe;
}



$grapheDedensifie = dedensifierUnGraphe($grapheCharge);
// var_dump($grapheCharge, $grapheDedensifie);
echo(
  "Dédensification d'un graphe à ".$grapheCharge["nombreDeSommets"]
 ." sommets"
 ." de degré minimum ".$grapheCharge["degreMinimum"]
 ." et de degré maximum ".$grapheCharge["degreMaximum"]
 ." en un graphe à ".$grapheDedensifie["nombreDeSommets"]
 ." sommets"
 ." de degré minimum ".$grapheDedensifie["degreMinimum"]
 ." et de degré maximum ".$grapheDedensifie["degreMaximum"]
 .".\n"
);



/**
 * partitionnerUnGrapheDedensifie()
 * EnPartiesDeSommetsADistanceAuMoins3 nom complet trop long :)
 * Fonction de partitionnement, en k + 1 parties de sommets
 * à distance au moins 3, des sommets d'un graphe
 * obtenu par dédensification d'un graphe de degré maximum k.
 * Cet algorithme s'exécute en temps quadratique
 * en le nombre de sommets dans le pire cas,
 * il fonctionne à base d'orientation d'arêtes,
 * d'un point de vu conceptuel.
 * Plus bas, je donne un algorithme de complexité linéaire
 * sur les graphes de degré borné.
 */
function partitionnerUnGrapheDedensifie($grapheDedensifie){
  if(
```

```
  !isset($grapheDedensifie["nombreDeSommetsAvantDedensification"])
){
  throw new Exception("Ce n'est pas un graphe dédensifié.");
}
$nombrePartiesMax
  = $grapheDedensifie["degreMaximumAvantDedensification"] + 1
;
// On initialise un tableau donnant la partie
// associée à chaque sommet
$grapheDedensifie["partiesEspaceesDe3"] = [
    "identifiantDeSommetVersPartie" => [],
    "listeDesParties" => [],
];
$sommetVersPartie
  = &$grapheDedensifie["partiesEspaceesDe3"]
    ["identifiantDeSommetVersPartie"]
;
// -1 correspond à "non affecté".
for($i = 0; $i < $grapheDedensifie["nombreDeSommets"]; ++$i){
  $sommetVersPartie[$i] = -1;
}
$nombreDeSommetsOriginaux
  = $grapheDedensifie["nombreDeSommetsAvantDedensification"]
;
// Les sommets originaux vont dans la première partie.
for($i = 0; $i < $nombreDeSommetsOriginaux; ++$i){
  $sommetVersPartie[$i] = 0;
}

// Les k-1 parties intermédiaires se construisent
// en prenant un voisin pour chaque sommet original
// et en renversant un nombre linéaire de choix si nécessaire.
for($k = 1; $k < $nombrePartiesMax - 1; ++$k){
  for($i = 0; $i < $nombreDeSommetsOriginaux; ++$i){
    // On regarde les au plus k chemins de longueur 3
    // partant du sommet.
    $chemins = $grapheDedensifie["sommetVersChemins"][$i];
    $dernierCheminCandidat = null;
    for($j = 0; $j < count($chemins); ++$j){
      $chemin = $chemins[$j];
      $proche = $chemin["proche"][$i];
      $lointain = $chemin["lointain"][$i];
      // $lointain est aussi le vis-à-vis de $proche.
      if($sommetVersPartie[$proche] !== -1){
        // On ne prend pas un sommet déjà pris
        // à une étape précédente.
```

```
    continue;
  }
  $dernierCheminCandidat = $chemin;
  if($sommetVersPartie[$lointain] === $k){
    // On ne prend pas en première intention
    // un sommet dont le voisin est dans la partie courante.
    continue;
  }
  $sommetVersPartie[$proche] = $k;
  // Si on a trouvé, on passe à l'itération suivante
  // de la boucle sur les sommets.
  continue 2;
}
if($dernierCheminCandidat === null){
  // S'il n'y a plus de chemin candidat,
  // c'est que le sommet avait un degré inférieur à k
  // et qu'il est déjà traité complètement.
  continue;
}

// Si on arrive là, c'est qu'il faut renverser un chemin.
$proche = $dernierCheminCandidat["proche"][$i];
$lointain = $dernierCheminCandidat["lointain"][$i];
$sommetVersPartie[$proche] = $k;
$sommetVersPartie[$lointain] = -1;
$sommetPrecedent = $i;
$sommetCourant = $dernierCheminCandidat["visAVis"][$i];
while(true){
  // On regarde les k chemins de longueur 3 partant du sommet.
  $chemins
    = $grapheDedensifie["sommetVersChemins"][$sommetCourant]
  ;
  $dernierCheminCandidat = null;
  for($j = 0; $j < count($chemins); ++$j){
    $chemin = $chemins[$j];
    $sommetSuivant = $chemin["visAVis"][$sommetCourant];
    if($sommetSuivant === $sommetPrecedent){
      // Pas le droit de retoucher son père.
      continue;
    }
    $proche = $chemin["proche"][$i];
    $lointain = $chemin["lointain"][$i];
    if($sommetVersPartie[$proche] !== -1){
      // On ne prend pas un sommet déjà pris
      // à une étape précédente.
      continue;
```

```
        }
        $dernierCheminCandidat = $chemin;
        if($sommetVersPartie[$lointain] === $k){
          // On ne prend pas en première intention
          // un sommet dont le voisin est dans la partie courante.
          continue;
        }
        $sommetVersPartie[$proche] = $k;
        // Si on a trouvé, on passe à l'itération suivante
        // de la boucle for sur les sommets.
        continue 3;
      }

      if($dernierCheminCandidat === null){
        // S'il n'y a plus de chemin candidat,
        // c'est que le sommet avait un degré inférieur à k
        // et qu'il est déjà traité complètement,
        // sauf pour le fait de "retoucher son père"
        // qu'il pourra faire lors de la prochaine partie.
        break; // et pas continue...
        // boucle infinie potentielle pour des graphes
        // non k-réguliers
      }

      // Sinon on continue à renverser le chemin.
      $proche = $dernierCheminCandidat["proche"][$sommetCourant];
      $lointain
        = $dernierCheminCandidat["lointain"][$sommetCourant]
      ;
      $sommetVersPartie[$proche] = $k;
      $sommetVersPartie[$lointain] = -1;
      $sommetPrecedent = $sommetCourant;
      $sommetCourant
        = $dernierCheminCandidat["visAVis"][$sommetCourant]
      ;
    }
  }
}
// Très important : un sommet de degré strictement inférieur à k
// ne peut pas accumuler plus d'une partie de retard.
// Car il n'accumule ce retard que si son "père" dans un
// retournement de chemin était l'unique choix restant
// pour ce sommet.
// Et nécessairement, il résorbe ce retard à l'étape d'après.
// En conséquence, à l'issue du calcul des k premières parties,
// tous les sommets de degré strictement inférieur à k ont été
```

```
// traités
// ou bien ils ont été bloqués par leur "père" à l'étape k.

// La (k + 1)-ème partie se construit
// en prenant un voisin pour chaque sommet original
// et en échangeant un nombre linéaire de choix si nécessaire
// entre la k-ème et la (k + 1)-ème partie.
$k = $nombrePartiesMax - 2; // La première partie correspond à 0
for($i = 0; $i < $nombreDeSommetsOriginaux; ++$i){
  // On regarde les k chemins de longueur 3 partant du sommet.
  $chemins = $grapheDedensifie["sommetVersChemins"][$i];
  $dernierCheminCandidat = null;
  for($j = 0; $j < count($chemins); ++$j){
    $chemin = $chemins[$j];
    $proche = $chemin["proche"][$i];
    $lointain = $chemin["lointain"][$i];
    // $lointain est aussi le vis-à-vis de $proche.
    if($sommetVersPartie[$proche] !== -1){
      // On ne prend pas un sommet déjà pris
      // à une étape précédente.
      continue;
    }
    $dernierCheminCandidat = $chemin;
    if($sommetVersPartie[$lointain] === $k + 1){
      // On ne prend pas en première intention
      // un sommet dont le voisin est dans la partie courante.
      break; // C'est la dernière partie, donc un seul candidat.
    }
    $sommetVersPartie[$proche] = $k + 1;
    // Si on a trouvé, on passe à l'itération suivante
    // de la boucle sur les sommets.
    continue 2;
  }

  if($dernierCheminCandidat === null){
    // S'il n'y a plus de chemin candidat,
    // c'est que le sommet avait un degré inférieur à k
    // et qu'il est déjà traité complètement.
    continue;
  }

  // Si on arrive là, c'est qu'il faut renverser un chemin.
  // On trouve le chemin sortant de $i à l'étape k
  $cheminSuivant = null;
  for($j = 0; $j < count($chemins); ++$j){
    $chemin = $chemins[$j];
```

```
  $proche = $chemin["proche"][$i];
  if($sommetVersPartie[$proche] === $k){
    $cheminSuivant = $chemin;
    break;
  }
}
$proche = $dernierCheminCandidat["proche"][$i];
// Avant de poser un autre choix pour l'étape k
$sommetVersPartie[$proche] = $k;
if($cheminSuivant === null){
  // Pas de chemin suivant, on a réussi à compléter en k étapes
  // un sommet de degré inférieur à k qui n'était complété
  // qu'en k+1 étapes :)
  break;
}
$sommetCourant = $i;

while(true){
  $sommetPrecedent = $sommetCourant;
  $cheminCourant = $cheminSuivant;
  $sommetCourant = $cheminCourant["visAVis"][$sommetPrecedent];

  // On trouve le chemin sortant de $sommetCourant à l'étape k
  $chemins
    = $grapheDedensifie["sommetVersChemins"][$sommetCourant]
  ;
  $cheminSuivant = null;
  for($j = 0; $j < count($chemins); ++$j){
    $chemin = $chemins[$j];
    $proche = $chemin["proche"][$sommetCourant];
    if($sommetVersPartie[$proche] === $k){
      $cheminSuivant = $chemin;
      break;
    }
  }
  // On inverse entre sommet précédent et sommet courant.
  $proche = $cheminCourant["proche"][$sommetPrecedent];
  $sommetVersPartie[$proche] = $k + 1;
  $lointain = $cheminCourant["lointain"][$sommetPrecedent];
  if($sommetVersPartie[$lointain] !== $k + 1){
    // Pas de conflit, on peut sortir de la boucle while.
    break;
  }
  // Sinon, comme on sait déjà où se poursuit le chemin
  // de l'étape k, on peut corriger le sommet du chemin courant.
  $sommetVersPartie[$lointain] = $k;
```

```
      if($cheminSuivant === null){
        // Pas de chemin suivant, on a réussi à compléter en k étapes
        // un sommet de degré inférieur à k qui n'était complété
        // qu'en k+1 étapes :)
        break;
      }

      // Et on boucle
    }
  }

  // On remplit les parties
  $listeDesParties
    = &$grapheDedensifie["partiesEspaceesDe3"]["listeDesParties"]
  ;
  for($i = 0; $i < $nombrePartiesMax; ++$i){
    $listeDesParties[$i] = [];
  }
  for($i = 0; $i < $grapheDedensifie["nombreDeSommets"]; ++$i){
    $partie = $sommetVersPartie[$i];
    $listeDesParties[$partie] []= $i;
  }

  return $grapheDedensifie;
}



$grapheDedensifieAvecParties = partitionnerUnGrapheDedensifie(
  $grapheDedensifie
);
/*
var_dump(
  $grapheDedensifieAvecParties["partiesEspaceesDe3"]
  ["identifiantDeSommetVersPartie"]
);
//*/
$partiesEspaceesDe3
  = &$grapheDedensifieAvecParties["partiesEspaceesDe3"]
;
$listeDesParties
  = &$partiesEspaceesDe3["listeDesParties"]
;
echo "Résultat du partitionnement en temps quadratique.\n";
foreach($listeDesParties as $i => $partie){
  echo(
```

```
    "La partie ".($i+1)." contient les ".count($partie)
   ." sommets suivants : ".join(
      ",",
      listeDeSommetsInternePourAffichage(
        $grapheDedensifieAvecParties,
        $partie
      )
    ).".\n"
  );
}



/**
 * partitionnerUnGrapheDedensifieParAlgoGlouton()
 * Fonction de partitionnement, en k + 2 parties de sommets
 * à distance au moins 3, des sommets d'un graphe
 * obtenu par dédensification d'un graphe de degré maximum k.
 * Cet algorithme s'exécute sur les graphes de degré borné
 * en temps linéaire en le nombre de sommets dans le pire cas.
 * Il est donc plus rapide que la fonction précédente,
 * mais malheureusement il utilise une partie de plus.
 */
function partitionnerUnGrapheDedensifieParAlgoGlouton($graphe){
  if(!isset($graphe["sommetVersBouleDeRayon2"])){
    $graphe = calculerLesBoulesDeRayon2($graphe);
  }

  $nombrePartiesMax = 2 + $graphe["degreMaximum"];
  // On initialise un tableau donnant la partie
  // associée à chaque sommet
  $graphe["partiesEspaceesDe3"] = [
      "identifiantDeSommetVersPartie" => [],
      "listeDesParties" => [],
  ];
  $sommetVersPartie
    = &$graphe["partiesEspaceesDe3"]["identifiantDeSommetVersPartie"]
  ;
  // -1 correspond à "non affecté"
  for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
    $sommetVersPartie[$i] = -1;
  }
  $nombreDeSommetsOriginaux
    = $grapheDedensifie["nombreDeSommetsAvantDedensification"]
  ;
  // Les sommets originaux vont dans la première partie.
```

```
for($i = 0; $i < $nombreDeSommetsOriginaux; ++$i){
  $sommetVersPartie[$i] = 0;
}

// Pour chaque sommet (facteur n)
for(
  $i = $nombreDeSommetsOriginaux;
  $i < $graphe["nombreDeSommets"];
  ++$i
){
  $boule = $graphe["sommetVersBouleDeRayon2"][$i];

  // Pour chaque partie
  // (facteur constant si le degré maximum est borné)
  for($j = 1; $j < $nombrePartiesMax; ++$j){
    // On regarde si un sommet de la boule y est déjà affecté.
    // $partieDejaPrise = false;
    foreach($boule as $sommetDansLaBoule){
      if($sommetVersPartie[$sommetDansLaBoule] === $j){
        // $partieDejaPrise = true;
        continue 2; // On passe directement à la partie suivante
      }
    }
    $sommetVersPartie[$i] = $j;
    break;
  }
}

// On regarde de combien de parties on a réellement eu besoin
$indexMaximumDePartieUtilisee = 0;
for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
  if($sommetVersPartie[$i] < 0){
    // Cela ne devrait pas se produire,
    // sauf si quelqu'un injecte un graphe
    // avec un degré maximum faux.
    throw new Exception(
      "Le sommet ".$graphe["listeDeSommets"][$i]
     ." n'a pas de partie."
    );
  }
  $indexMaximumDePartieUtilisee = max(
    $indexMaximumDePartieUtilisee,
    $sommetVersPartie[$i]
  );
}
```

```
  // On remplit les parties
  $listeDesParties
    = &$graphe["partiesEspaceesDe3"]["listeDesParties"]
  ;
  for($i = 0; $i <= $indexMaximumDePartieUtilisee; ++$i){
    $listeDesParties[$i] = [];
  }
  for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
    $partie = $sommetVersPartie[$i];
    $listeDesParties[$partie] []= $i;
  }

  return $graphe;
}



/**
 * Voir la fonction juste après.
 */
function sousCalculPointDarticulation(
  &$graphe,
  &$listeDeSommets,
  &$listesDadjacence,
  &$horaireDeVisite,
  &$horaireDesAncetres,
  &$sommetVersStatut,
  $sommetPrecedent,
  $sommetCourant,
  $horaireCourant
){
  ++$horaireCourant;
  $horaireDeVisite[$sommetCourant] = $horaireCourant;
  $horaireDesAncetres[$sommetCourant] = $horaireCourant;
  foreach(
    $graphe["listesDadjacences"][$sommetCourant] as $voisin
  ){
    if(
      // Si le voisin n'est pas dans le sous-graphe induit,
      $sommetVersStatut[$voisin] === -1
      // ou que c'est notre père,
      || $voisin === $sommetPrecedent
    ){
      // on l'ignore.
      continue;
    }
```

```
    if($horaireDeVisite[$voisin] !== -1){
      // On a un arc retour.
      $horaireDesAncetres[$sommetCourant] = (
        $horaireDeVisite[$voisin]
      );
      continue;
    }
    // Sinon, on parcours
    sousCalculPointDarticulation(
      $graphe,
      $listeDeSommets,
      $listesDadjacence,
      $horaireDeVisite,
      $horaireDesAncetres,
      $sommetVersStatut,
      $sommetCourant,
      $voisin,
      $horaireCourant
    );
    if(
      $horaireDesAncetres[$voisin]
      >= $horaireDeVisite[$sommetCourant]
    ){
      $sommetVersStatut[$sommetCourant] = true;
    }
    $horaireDesAncetres[$sommetCourant] = min(
      $horaireDesAncetres[$sommetCourant],
      $horaireDesAncetres[$voisin]
    );
  }
}



/**
 * Calcul en temps linéaire (n+m) des points d'articulation
 * d'un sous-graphe induit (algorithme standard).
 * Cette fonction renvoie un tableau associant chaque sommet
 * à son statut oui/non/-1 (-1 pour non concerné).
 */
function calculerLesPointsDArticulationDunSousGrapheInduit(
  &$graphe,
  &$listeDeSommets,
  $cleListesDadjacence
){
  $horaireDeVisite = [];
```

```
  $horaireDesAncetres = [];
  $sommetVersStatut = [];
  $listesDadjacence = $graphe[$cleListesDadjacence];
  for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
    $horaireDeVisite[$i] = -1;
    $horaireDesAncetres[$i] = -1;
    $sommetVersStatut[$i] = -1;
  }
  foreach($listeDeSommets as $i){
    // Pas un point d'articulation par défaut.
    $sommetVersStatut[$i] = false;
  }

  foreach($listeDeSommets as $i){
    if($horaireDeVisite[$i] !== -1){
      continue;
    }
    sousCalculPointDarticulation(
      $graphe,
      $listeDeSommets,
      $listesDadjacence,
      $horaireDeVisite,
      $horaireDesAncetres,
      $sommetVersStatut,
      null, // Pas de sommet précédent dans la composante connexe
      $i,
      0
    );
  }
  return $sommetVersStatut;
}



/**
 * partitionnerUnGrapheDedensifie2()
 * EnPartiesDeSommetsADistanceAuMoins3 nom complet trop long :)
 * Fonction de partitionnement, en k + 1 parties de sommets
 * à distance au moins 3, des sommets d'un graphe
 * obtenu par dédensification d'un graphe de degré maximum k.
 * Cet algorithme s'exécute en temps linéaire
 * en le nombre de sommets dans le pire cas,
 * sur les graphes de degré borné,
 * il fonctionne en colorant le carré du graphe
 * avec des parcours en largeur.
 * Il permet de démontrer la complexité annoncée
```

```
 * au Théorème 4.2, avec l'aide de l'algorithme implémenté
 * dans la fonction decomposerUnGrapheParPremiereDifference().
 */
function partitionnerUnGrapheDedensifie2($grapheDedensifie){
  if(
    !isset($grapheDedensifie["nombreDeSommetsAvantDedensification"])
  ){
    throw new Exception("Ce n'est pas un graphe dédensifié.");
  }
  // Passage au carré
  if(!isset($grapheDedensifie["sommetVersBouleDeRayon2"])){
    $grapheDedensifie = calculerLesBoulesDeRayon2($grapheDedensifie);
  }
  $nombrePartiesMax
    = $grapheDedensifie["degreMaximumAvantDedensification"] + 1
  ;
  // On initialise un tableau donnant la partie
  // associée à chaque sommet
  $grapheDedensifie["partiesEspaceesDe3"] = [
      "identifiantDeSommetVersPartie" => [],
      "listeDesParties" => [],
  ];
  $sommetVersPartie
    = &$grapheDedensifie["partiesEspaceesDe3"]
      ["identifiantDeSommetVersPartie"]
  ;
  // -1 correspond à "non affecté".
  for($i = 0; $i < $grapheDedensifie["nombreDeSommets"]; ++$i){
    $sommetVersPartie[$i] = -1;
  }
  $nombreDeSommetsOriginaux
    = $grapheDedensifie["nombreDeSommetsAvantDedensification"]
  ;
  // Les sommets originaux vont dans la première partie.
  for($i = 0; $i < $nombreDeSommetsOriginaux; ++$i){
    $sommetVersPartie[$i] = 0;
  }

  // Les k-2 parties intermédiaires se construisent
  // avec un parcours en largeur modifié.
  for($k = 1; $k < $nombrePartiesMax - 1; ++$k){
    $listeDeSommetsRestants = [];
    $sommetsDansLaFile = [];
    for($i = 0; $i < $grapheDedensifie["nombreDeSommets"]; ++$i){
      if($sommetVersPartie[$i] === -1){
        $listeDeSommetsRestants []= $i;
```

```
    $sommetsDansLaFile[$i] = false;
  }
}
$sommetVersStatutPointDarticulation = (
  calculerLesPointsDArticulationDunSousGrapheInduit(
    $grapheDedensifie,
    $listeDeSommetsRestants,
    "sommetVersBouleDeRayon2"
  )
);
foreach($listeDeSommetsRestants as $sommet){
  if(
    // On a déjà vu le sommet dans un parcours en largeur
    // de cette étape.
    $sommetsDansLaFile[$sommet]
    // On ne veut pas un point d'articulation.
    || $sommetVersStatutPointDarticulation[$sommet]

  ){
    continue;
  }
  // Parcours en largeur modifié
  $sommetVersPartie[$sommet] = $k;
  $file = [];
  foreach(
    $grapheDedensifie["sommetVersBouleDeRayon2"][$sommet]
    as $voisin
  ){
    if(
      $grapheDedensifie["sommetVersIdDeClique"][$sommet]
      === $grapheDedensifie["sommetVersIdDeClique"][$voisin]
      && $sommetVersPartie[$voisin] === -1
    ){
      // On ajoute le sommet avec sa distance dans la file.
      // C'est purement pédagogique.
      // Une des conditions ci-dessous
      // rend la gestion de la distance facultative.
      $file []= [$voisin, 1];
      $sommetsDansLaFile[$voisin] = true;
    }
  }
  while(!empty($file)){
    $courant = array_shift($file);
    $sommet = $courant[0];
    $distance = $courant[1];
    $voisinDansLaPartieCourante = false;
```

```
      foreach(
        $grapheDedensifie["sommetVersBouleDeRayon2"][$sommet]
        as $voisin
      ){
        if(
          $sommetVersPartie[$voisin] === -1
          && $sommetsDansLaFile[$voisin] === false
        ){
          $file []= [$voisin, $distance + 1];
          $sommetsDansLaFile[$voisin] = true;
          continue;
        }
        if($sommetVersPartie[$voisin] === $k){
          $voisinDansLaPartieCourante = true;
        }
      }
      if(
        !$voisinDansLaPartieCourante
        // La ligne ci-dessous peut-être enlevée
        // et le reste du code simplifié.
        && ($distance % 2) === 0
      ){
        $sommetVersPartie[$sommet] = $k;
      }
    }
  }
}

// On crée les deux dernières parties en une passe.
$listeDeSommetsRestants = [];
$sommetsDansLaFile = [];
for($i = 0; $i < $grapheDedensifie["nombreDeSommets"]; ++$i){
  if($sommetVersPartie[$i] === -1){
    $listeDeSommetsRestants []= $i;
    $sommetsDansLaFile[$i] = false;
  }
}
foreach($listeDeSommetsRestants as $sommet){
  if($sommetVersPartie[$sommet] !== -1){
    // Un des parcours en largeur modifié de cette étape
    // s'en est déjà occupé.
    continue;
  }
  // Parcours en largeur
  $sommetVersPartie[$sommet] = $k;
  $file = [];
```

```
  foreach(
    $grapheDedensifie["sommetVersBouleDeRayon2"][$sommet]
    as $voisin
  ){
    if($sommetVersPartie[$voisin] === -1){
      $file []= $voisin;
      $sommetsDansLaFile[$voisin] = true;
    }
  }
  while(!empty($file)){
    $sommet = array_shift($file);
    $voisinDansLaPartieCourante = false;
    foreach(
      $grapheDedensifie["sommetVersBouleDeRayon2"][$sommet]
      as $voisin
    ){
      if(
        $sommetVersPartie[$voisin] === -1
        && $sommetsDansLaFile[$voisin] === false
      ){
        $file []= $voisin;
        $sommetsDansLaFile[$voisin] = true;
        continue;
      }
      if($sommetVersPartie[$voisin] === $k){
        $voisinDansLaPartieCourante = true;
      }
    }
    if($voisinDansLaPartieCourante){
      $sommetVersPartie[$sommet] = $k + 1;
    }
    else{
      $sommetVersPartie[$sommet] = $k;
    }
  }
}

// On remplit les parties
$listeDesParties
  = &$grapheDedensifie["partiesEspaceesDe3"]["listeDesParties"]
;
for($i = 0; $i < $nombrePartiesMax; ++$i){
  $listeDesParties[$i] = [];
}
for($i = 0; $i < $grapheDedensifie["nombreDeSommets"]; ++$i){
  $partie = $sommetVersPartie[$i];
```

```
    $listeDesParties[$partie] []= $i;
  }

  return $grapheDedensifie;
}



$grapheDedensifieAvecParties = partitionnerUnGrapheDedensifie2(
  $grapheDedensifie
);
/*
var_dump(
  $grapheDedensifieAvecParties["partiesEspaceesDe3"]
  ["identifiantDeSommetVersPartie"]
);
//*/
$partiesEspaceesDe3
  = &$grapheDedensifieAvecParties["partiesEspaceesDe3"]
;
$listeDesParties
  = &$partiesEspaceesDe3["listeDesParties"]
;
echo "Résultat du partitionnement en temps linéaire.\n";
foreach($listeDesParties as $i => $partie){
  echo(
    "La partie ".($i+1)." contient les ".count($partie)
   ." sommets suivants : ".join(
      ",",
      listeDeSommetsInternePourAffichage(
        $grapheDedensifieAvecParties,
        $partie
      )
    ).".\n"
  );
}


$grapheDedensifieDecompose = decomposerUnGrapheParPremiereDifference(
  $grapheDedensifieAvecParties
);
/*
var_dump(
  $grapheDedensifieDecompose["decompositionParPremiereDifference"]
);
//*/
```

```
foreach(
  $grapheDedensifieDecompose["listeDeSommets"] as $i => $sommet
){
  echo(
    "Le sommet $sommet est representé par la chaîne "
    .$grapheDedensifieDecompose["decompositionParPremiereDifference"]
        ["sommetVersChaineSenaire"][$i]
    .".\n"
  );
}



/**
 * Calcul en temps linéaire (n+m) des composantes connexes
 * d'un sous-graphe induit.
 * Cette fonction renvoie un tableau associant chaque sommet
 * à son numéro de composante connexe.
 */
function calculerLesComposantesDunSousGrapheInduit(
  $graphe,
  $listeDeSommets
){
  $sommetVersComposante = [];
  // L'absence du tableau associatif (table de hashage)
  // nous permet de tester la (non-)appartenance aux sommets du
  // sous-graphe induit en temps constant.
  // Donc on utilise la valeur -1 pour avoir l'appartenance
  // d'un sommet qui n'a pas encore reçu sa composante.
  foreach($listeDeSommets as $i){
    $sommetVersComposante[$i] = -1;
  }
  $numeroComposanteCourante = 0;
  foreach($listeDeSommets as $i){
    if($sommetVersComposante[$i] !== -1){
      continue;
    }

    $sommetsADepiler = [$i];
    // Parcours en profondeur classique
    while(!empty($sommetsADepiler)){
      $sommetCourant = array_pop($sommetsADepiler);
      $sommetVersComposante[$sommetCourant]
        = $numeroComposanteCourante
      ;
      foreach(
```

```
        $graphe["listesDadjacences"][$sommetCourant] as $voisin
      ){
        if(
          // Si le voisin n'est pas dans le sous-graphe induit,
          !isset($sommetVersComposante[$voisin])
          // ou s'il a déjà une composante
          || $sommetVersComposante[$voisin] !== -1
        ){
          // on l'ignore.
          continue;
        }
        // Sinon, on l'ajoute.
        $sommetVersComposante[$voisin] = $numeroComposanteCourante;
        $sommetsADepiler []= $voisin;
      }
    }
    ++$numeroComposanteCourante;
  }
  return $sommetVersComposante;
}



/*
$sommetVersComposante = calculerLesComposantesDunSousGrapheInduit(
  $grapheCharge,
  [0, 1, 2, 3, 6, 10, 11, 17] // a,b,c,d,g,k,l,r
);
var_dump($sommetVersComposante);
//*/



/**
 * Étant donné un graphe connexe $graphe,
 * et un sous-graphe induit $listeDeSommets
 * dont les c composantes connexes sont à distance
 * exactement 2 dans $graphe,
 * calcul en temps quadratique (complexité non optimisée)
 * d'un sous-graphe induit connexe
 * $listeDeSommetsEtendue qui contient $listeDeSommets.
 * Dans le cas qui nous intéresse où les sommets restants
 * sont de degré 2, on ajoute exactement c-1 sommets de plus.
 */
function calculerUneExtensionConnexeDunSousGrapheInduit(
  $graphe,
```

```
  $listeDeSommets
){
  $sommetVersComposante = calculerLesComposantesDunSousGrapheInduit(
    $graphe,
    $listeDeSommets
  );
  $listeDeSommetsEtendue = $listeDeSommets;
  for($i = 0; $i < $graphe["nombreDeSommets"]; ++$i){
    if(isset($sommetVersComposante[$i])){
      // On cherche les sommets à ajouter, pas ceux qu'on a déjà.
      continue;
    }
    $composantesDesVoisins = [];
    foreach(
      $graphe["listesDadjacences"][$i] as $voisin
    ){
      // Temps linéaire n+m
      if(
        // Si le voisin n'est pas dans le sous-graphe induit,
        !isset($sommetVersComposante[$voisin])
      ){
        // on l'ignore.
        continue;
      }
      // Sinon, on ajoute sa composante.
      $composantesDesVoisins []= $sommetVersComposante[$voisin];
    }
    $composantesDesVoisins = array_unique($composantesDesVoisins);
    if(count($composantesDesVoisins) > 1){
      $listeDeSommetsEtendue []= $i;
      $composanteMin = min($composantesDesVoisins);
      $sommetVersComposante[$i] = $composanteMin;
      foreach($composantesDesVoisins as $composante){
        // Bien que l'on soit dans une boucle imbriquée,
        // pour l'ensemble des sommets de la première boucle for,
        // on ne peut rentrer dans cette boucle qu'au plus n fois
        if($composante === $composanteMin){
          continue;
        }
        foreach($sommetVersComposante as $sommet => $saComposante){
          // Chaque sommet est parcouru,
          // d'où la complexité quadratique.
          if($saComposante === $composante){
            // On réunit tout
            $sommetVersComposante[$sommet] = $composanteMin;
          }
```

```
        }
      }
    }
  }
  return $listeDeSommetsEtendue;
}



$partiesEspaceesDe3
  = &$grapheDedensifieDecompose["partiesEspaceesDe3"]
;
$listeDesParties
  = &$partiesEspaceesDe3["listeDesParties"]
;
$sommetsPeuConnectes
  = calculerUneExtensionConnexeDunSousGrapheInduit(
      $grapheDedensifieDecompose,
      array_merge(
        $listeDesParties[0],
        $listeDesParties[1],
        $listeDesParties[2]
      )
);
echo(
  "Les ".count($sommetsPeuConnectes)
 ." sommets suivants sont connectés mais peu : ".join(
    ",",
    listeDeSommetsInternePourAffichage(
      $grapheDedensifieDecompose,
      $sommetsPeuConnectes
    )
  ).".\n"
);
echo(
  "L'important pour la complexité modérément exponentielle"
 ." c'est que le nombre de sommets soit au moins égal à"
 ." 3 * 22 + ((22 / 2) - 1) = 76.\n"
 ." Ce qui implique qu'il reste au plus 12 =  22/2 + 1 sommets,"
 ." pour l'énumération exhaustive.\n"
);



/**
 * Temps constant si le degré est borné.
```

```
 */
function calculerLeDegreDansUnSousGrapheInduit(
  &$graphe,
  &$sommet,
  // Sous forme id => true plutôt que liste
  &$sommetsCandidats
){
  $degre = 0;
  foreach(
    $graphe["listesDadjacences"][$sommet] as $voisin
  ){
    if(isset($sommetsCandidats[$voisin])){
      ++$degre;
    }
  }
  return $degre;
}



/**
 * Cette fonction implémente l'algorithme glouton
 * qui détecte une feuille
 * pour étendre un stable en un stable maximum
 * de la sous-forêt considérée.
 * Elle fonctionne en temps linéaire n+m,
 * où n est le nombre de sommets du sous-graphe,
 * et m est la somme des degrés de ces sommets,
 * mais dans le graphe de départ et non le sous-graphe.
 * Comme je considère des graphes de degré borné,
 * cela n'importe que peu.
 * Son fonctionnement est légèrement étendu pour accepter
 * un sous-graphe induit quelconque,
 * donc elle va "grignoter" les feuilles jusqu'à obtenir
 * un graphe sans feuille.
 * Et elle admet comme optimisation une pile avec test d'appartenance
 * en temps constant (je fais ça avec deux tableaux associatifs PHP),
 * qui permet de restreindre l'ensemble de sommets initiaux
 * à considérer pour trouver une feuille.
 */
function trouverUnStableMaximumDUneSousForetInduite(
  $graphe,
  // passages de tableaux par référence
  &$listeDeSommetsDuSousGraphe, // une liste
  &$sommetsCandidats = null, // idSommet => true
  &$sommetsADepilerPourTrouverUneFeuille = null, // Une pile
```

```
  &$sommetsAEvaluerPourTrouverUneFeuille = null // idSommet => true
){
  $listeDeSommetsDuStable = [];
  // Sous forme id => true dans la table de hashage.
  // Cela permet d'ajouter ou d'enlever un sommet,
  // de les compter, ou de tester l'appartenance en temps constant.
  if($sommetsCandidats === null){
    $sommetsCandidats = [];
    foreach($listeDeSommetsDuSousGraphe as $sommet){
      $sommetsCandidats[$sommet] = true;
    }
  }
  if($sommetsADepilerPourTrouverUneFeuille === null){
    // On va utiliser ce tableau comme une pile.
    // Quand le degré est borné, chaque sommet est ajouté
    // à cette pile au plus un nombre constant de fois.
    $sommetsADepilerPourTrouverUneFeuille
      = $listeDeSommetsDuSousGraphe
    ;
    // Et on utilise cette copie pour évaluer
    $sommetsAEvaluerPourTrouverUneFeuille = $sommetsCandidats;
    // Donc dans le cas qui nous intéresse la recherche de feuilles
    // se fait en temps linéaire sur tout l'algorithme.
  }

  while(count($sommetsADepilerPourTrouverUneFeuille) > 0){
    $sommet = array_pop($sommetsADepilerPourTrouverUneFeuille);
    unset($sommetsAEvaluerPourTrouverUneFeuille[$sommet]);
    // Si entre temps, le sommet n'est plus candidat,
    // à cause d'un ajout de feuille au stable,
    // on passe ce sommet.
    if(!isset($sommetsCandidats[$sommet])){
      continue;
    }
    $degre = calculerLeDegreDansUnSousGrapheInduit(
      $graphe,
      $sommet,
      $sommetsCandidats
    );
    if($degre > 1){
      continue;
    }
    // On a trouvé une feuille ou un sommet isolé ;
    // on la prend.
    $listeDeSommetsDuStable []= $sommet;
    unset($sommetsCandidats[$sommet]);
```

```
    if($degre === 1){
      foreach(
        $graphe["listesDadjacences"][$sommet] as $voisin
      ){
        if(isset($sommetsCandidats[$voisin])){
          // On a trouvé le voisin.
          break;
        }
      }
      // Le voisin n'est plus candidat.
      unset($sommetsCandidats[$voisin]);
      // On regarde s'il a lui même des voisins candidats
      // qui pourraient être des feuilles.
      foreach(
        $graphe["listesDadjacences"][$voisin] as $voisinDeVoisin
      ){
        if(
          // Si le voisin de voisin n'est pas candidat,
          !isset($sommetsCandidats[$voisinDeVoisin])
          // ou qu'il est déjà dans la pile
          || isset(
            $sommetsAEvaluerPourTrouverUneFeuille[$voisinDeVoisin]
          )
        ){
          // alors il ne nous intéresse pas
          continue;
        }
        // Sinon, on l'ajoute à la pile
        $sommetsADepilerPourTrouverUneFeuille []= $voisinDeVoisin;
        $sommetsAEvaluerPourTrouverUneFeuille[$voisinDeVoisin]
          = true
        ;
        // Si après ça, vous n'êtes pas convaincu que les tableaux
        // associatifs de PHP sont de vrais couteaux suisses,
        // je ne sais pas ce qu'il vous faut :)
      }
    }
  }
  return $listeDeSommetsDuStable;
}



$sousForet = [0,1,2,4,7,10];
$stableMaximumDUneSousForet
  = trouverUnStableMaximumDUneSousForetInduite(
```

```
      $grapheCharge,
      $sousForet
);
echo(
  "Les ".count($stableMaximumDUneSousForet)
 ." sommets suivants : ".join(
    ",",
    listeDeSommetsInternePourAffichage(
      $grapheCharge,
      $stableMaximumDUneSousForet
    )
  )
  ." forment un stable maximum de la sous-forêt : ".join(
    ",",
    listeDeSommetsInternePourAffichage(
      $grapheCharge,
      $sousForet
    )
  )
  .".\n"
);



/**
 * Cette fonction renvoie en temps linéaire
 * (cf. fonction précédente) un stable maximum d'un cycle induit.
 */
function trouverUnStableMaximumDUnCycleInduit(
  $graphe,
  $listeDeSommetsDuCycle,
  $sommetANePasPrendre = null
){
  if($sommetANePasPrendre === null){
    // On peut casser le cycle de manière arbitraire
    // avec un sommet que l'on ne prend pas.
    $sommetANePasPrendre = $listeDeSommetsDuCycle[0];
  }
  $sommetsCandidats = [];
  foreach($listeDeSommetsDuCycle as $sommet){
    $sommetsCandidats[$sommet] = true;
  }
  unset($sommetsCandidats[$sommetANePasPrendre]);
  $listeDeSommets = array_keys($sommetsCandidats);
  // On ne part qu'avec un seul sommet à dépiler :
  // l'un des deux voisins du sommet à ne pas prendre.
```

```
  // De cette manière la pile n'aura jamais plus d'un élément.
  $sommetsADepilerPourTrouverUneFeuille = [];
  $sommetsAEvaluerPourTrouverUneFeuille = [];
  foreach(
    $graphe["listesDadjacences"][$sommetANePasPrendre] as $voisin
  ){
    if(isset($sommetsCandidats[$voisin])){
      break;
    }
  }
  $sommetsADepilerPourTrouverUneFeuille []= $voisin;
  $sommetsAEvaluerPourTrouverUneFeuille[$voisin] = true;

  return trouverUnStableMaximumDUneSousForetInduite(
    $graphe,
    $listeDeSommets,
    $sommetsCandidats,
    $sommetsADepilerPourTrouverUneFeuille,
    $sommetsAEvaluerPourTrouverUneFeuille
  );
}



$cycleInduit = [0,1,2,4,5,6,7,8];
$stableMaximumDUnCycleInduit
  = trouverUnStableMaximumDUnCycleInduit(
      $grapheCharge,
      $cycleInduit
);
echo(
  "Les ".count($stableMaximumDUneSousForet)
 ." sommets suivants : ".join(
    ",",
    listeDeSommetsInternePourAffichage(
      $grapheCharge,
      $stableMaximumDUnCycleInduit
    )
  )
  ." forment un stable maximum du cycle induit : ".join(
    ",",
    listeDeSommetsInternePourAffichage(
      $grapheCharge,
      $cycleInduit
    )
  )
```

```
  ."."\n"
);



/**
 * Cette fonction implémente l'algorithme glouton
 * qui détecte une feuille ou un cycle-feuille
 * pour étendre un stable en un stable maximum
 * du sous-graphe peu dense considéré.
 * Peu d'effort d'optimisation a été fait.
 * Elle a une complexité quadratique en temps.
 */
function trouverUnStableMaximumDUnSousGrapheInduitPeuDense(
  $graphe,
  $listeDeSommets
){
  $listeDeSommetsDuStable = [];
  $sommetsCandidats = [];
  foreach($listeDeSommets as $sommet){
    $sommetsCandidats[$sommet] = true;
  }
  $sommetsADepilerPourTrouverUneFeuille = $listeDeSommets;
  $sommetsAEvaluerPourTrouverUneFeuille = $sommetsCandidats;

  while(count($sommetsCandidats) > 0){
    $listeDeSommetsDuStable = array_merge(
      $listeDeSommetsDuStable,
      // Rappel : cette fonction accepte aussi un sous-graphe induit
      // qui n'est pas une sous-forêt, auquel cas elle se contente
      // de "grignoter" les feuilles éventuelles.
      trouverUnStableMaximumDUneSousForetInduite(
        $graphe,
        $listeDeSommets,
        $sommetsCandidats,
        $sommetsADepilerPourTrouverUneFeuille,
        $sommetsAEvaluerPourTrouverUneFeuille
      )
    );
    // Si on arrive là, on regarde s'il reste des candidats
    if(count($sommetsCandidats) <= 0){
      break;
    }
    // On cherche un cycle-feuille.
    // Le moyen le plus simple en temps linéaire,
    // c'est de faire un parcours en profondeur.
```

```
// En effet, tout cycle boucle nécessairement sur un ancêtre
// du sommet courant dans l'arbre de parcours.
// Il suffit alors de vérifier qu'au plus un sommet a un degré
// supérieur à deux.
reset($sommetsCandidats);
// reset() ne vide pas le tableau.
// Cela rembobine juste l'itérateur intégré.
// On part du premier candidat
$premierCandidat = key($sommetsCandidats);
$sommetsADepiler = [$premierCandidat];
$predecesseurs = [];
$predecesseurs[$premierCandidat] = false;
// Parcours en profondeur classique
while(!empty($sommetsADepiler)){
  $sommetCourant = array_pop($sommetsADepiler);
  foreach(
    $graphe["listesDadjacences"][$sommetCourant] as $voisin
  ){
    if(
      // Si le voisin n'est pas dans le sous-graphe induit,
      !isset($sommetsCandidats[$voisin])
    ){
      // on l'ignore.
      continue;
    }
    if(isset($predecesseurs[$voisin])){
      // Si on a détecté un cycle,
      // on regarde si c'est un cycle-feuille.
      $sommetDeDegreSuperieurA2 = null;
      $sommetDuCycle = $sommetCourant;
      $listeDeSommetsDuCycle = [$voisin];
      while($sommetDuCycle !== $voisin){
        $listeDeSommetsDuCycle []= $sommetDuCycle;
        $degre = calculerLeDegreDansUnSousGrapheInduit(
          $graphe,
          $sommetDuCycle,
          $sommetsCandidats
        );
        if($degre > 2){
          if($sommetDeDegreSuperieurA2 === null){
            $sommetDeDegreSuperieurA2 = $sommetDuCycle;
          }
          else{
            $sommetDeDegreSuperieurA2 = "PlusDun";
            break;
          }
```

```
    }
    $sommetDuCycle = $predecesseurs[$sommetDuCycle];
  }
  if($sommetDeDegreSuperieurA2 !== "PlusDun"){
    // On a bien trouvé un cycle feuille.
    $listeDeSommetsDuStable = array_merge(
      $listeDeSommetsDuStable,
      trouverUnStableMaximumDUnCycleInduit(
        $graphe,
        $listeDeSommetsDuCycle,
        // Le sommet à ne pas prendre
        $sommetDeDegreSuperieurA2
      )
    );
    foreach($listeDeSommetsDuCycle as $sommetDuCycle){
      unset($sommetsCandidats[$sommetDuCycle]);
    }
    if($sommetDeDegreSuperieurA2 !== null){
      // Si on avait un sommet de degré supérieur à 2
      // dans le cycle, son éventuel voisin est peut être
      // devenu une feuille.
      foreach(
        $graphe["listesDadjacences"]
          [$sommetDeDegreSuperieurA2]
        as $voisin
      ){
        if(
          isset($sommetsCandidats[$voisin])
          && !isset(
            $sommetsAEvaluerPourTrouverUneFeuille[$voisin]
          )
        ){
          $sommetsADepilerPourTrouverUneFeuille []= $voisin;
          $sommetsAEvaluerPourTrouverUneFeuille[$voisin]
            = true
          ;
        }
      }
    }
    // Il faut 3 niveaux pour repartir
    // dans la boucle while principale
    // while(count($sommetsCandidats) > 0)
    continue 3;
  }
}
$sommetsADepiler []= $voisin;
```

```
      }
    }
  }
  return $listeDeSommetsDuStable;
}



/**
 * Maintenant que le lecteur est familiarisé
 * avec les tableaux associatifs de PHP,
 * on peut factoriser un peu :)
 */
function listeVersTableauIdToBool(&$liste){
  $idToBool = [];
  foreach($liste as $element){
    $idToBool[$element] = true;
  }
  return $idToBool;
}



/**
 * C'est ici que se passe l'énumération exhaustive,
 * plus une partie de l'assemblage des morceaux.
 */
function trouverUnStableMaximumDUnGrapheDedensifie(
  $grapheDedensifie
){
  $grapheDedensifieAvecParties = partitionnerUnGrapheDedensifie(
    $grapheDedensifie
  );
  $partiesEspaceesDe3
    = &$grapheDedensifieAvecParties["partiesEspaceesDe3"]
  ;
  $listeDesParties
    = &$partiesEspaceesDe3["listeDesParties"]
  ;
  $sommetsPeuConnectes
    = calculerUneExtensionConnexeDunSousGrapheInduit(
        $grapheDedensifieAvecParties,
        array_merge(
          $listeDesParties[0],
          $listeDesParties[1],
          $listeDesParties[2]
```

```
      )
);
$sommetsPeuConnectesIdToBool = listeVersTableauIdToBool(
  $sommetsPeuConnectes
);
$sommetsRestants = [];
$sommetsRestantsIdToBool = [];
foreach(
  $grapheDedensifie["etiquetteDeSommetVersIdentifiantInterne"]
  as $sommet
){
  if(!isset($sommetsPeuConnectesIdToBool[$sommet])){
    $sommetsRestants []= $sommet;
    $sommetsRestantsIdToBool[$sommet] = true;
  }
}
$stableMaximum = [];
// énumération exhaustive sur les sommets restants
// chaque bit correspond au choix d'un sommet.
// ATTENTION : je ne gère pas plus de 63 sommets restants
// avec cette énumération exhaustive à la "va comme je te pousse".
// Le lecteur qui a lu jusqu'ici et aura compris,
// ne devrait pas avoir de mal à utiliser php-gmp
// pour palier à cette limitation.
for($i = 0; $i < 2**count($sommetsRestants); ++$i){
  $sousStable = [];
  foreach($sommetsRestants as $j => $sommet){
    if(($i & (1 << $j)) > 0){
      $sousStable []= $sommet;
    }
  }
  $estStable = verifierQueDesSommetsFormentUnStable(
    $grapheDedensifie,
    $sousStable
  );
  if(!$estStable){
    continue;
  }
  $sousStableIdToBool = listeVersTableauIdToBool(
    $sousStable
  );
  // On ampute les sommets peu connectés
  // des voisins des sommets du sous-stable.
  $listeDeSommets = [];
  foreach($sommetsPeuConnectes as $sommet){
    foreach(
```

```
        $grapheDedensifie["listesDadjacences"][$sommet] as $voisin
      ){
        if(isset($sousStableIdToBool[$voisin])){
          continue 2;
        }
      }
      $listeDeSommets []= $sommet;
    }
    $stableCandidat = array_merge(
      $sousStable,
      trouverUnStableMaximumDUnSousGrapheInduitPeuDense(
        $grapheDedensifie,
        $listeDeSommets
      )
    );
    if(count($stableCandidat) > count($stableMaximum)){
      $stableMaximum = $stableCandidat;
    }
  }

  return $stableMaximum;
}



/**
 * Voilà la fonction de calcul d'un stable maximum
 * d'un graphe 3-régulier où on finit l'assemblage
 * des morceaux.
 */
function trouverUnStableMaximumDUnGraphe3Regulier($graphe){
  $grapheDedensifie = dedensifierUnGraphe($graphe);
  $stableMaximumDedensifie
    = trouverUnStableMaximumDUnGrapheDedensifie(
      $grapheDedensifie
  );
  $stableMaximumDedensifieIdToBool = listeVersTableauIdToBool(
    $stableMaximumDedensifie
  );
  $etiquetteVersIdentifiantInterne
    = &$graphe["etiquetteDeSommetVersIdentifiantInterne"]
  ;
  // On "redensifie" le stable
  foreach($graphe["listeDesAretes"] as $arete){
    $sommet1 = $etiquetteVersIdentifiantInterne[$arete[0]];
    $sommet2 = $etiquetteVersIdentifiantInterne[$arete[1]];
```

```
    if(
      // Si deux sommets adjacents sont dans le stable
      isset($stableMaximumDedensifieIdToBool[$sommet1])
      && isset($stableMaximumDedensifieIdToBool[$sommet2])
    ){
      // On en enlève un de manière arbitraire.
      unset($stableMaximumDedensifieIdToBool[$sommet2]);
    }
  }
  $stableMaximum = [];
  foreach(
    $graphe["etiquetteDeSommetVersIdentifiantInterne"]
    as $sommet
  ){
    if(isset($stableMaximumDedensifieIdToBool[$sommet])){
      $stableMaximum []= $sommet;
    }
  }
  return $stableMaximum;
}



$stableMaximum = trouverUnStableMaximumDUnGraphe3Regulier(
  $grapheCharge
);
echo(
  "Les ".count($stableMaximum)
 ." sommets suivants : ".join(
    ",",
    listeDeSommetsInternePourAffichage(
      $grapheCharge,
      $stableMaximum
    )
  )
  ." forment un stable maximum du graphe en exemple.\n"
);
$estStable = verifierQueDesSommetsFormentUnStable(
  $grapheCharge,
  $stableMaximum
);
if(!$estStable){
  throw new Exception(
    "Le résultat de la fonction"
   ." trouverUnStableMaximumDUnGraphe3Regulier()"
   ." n'est pas un stable.\n"
```

```
  );
}

?>
```