Décomposition modulaire de 2-structures

Rapport de projet

étudiante : Eleni Pistiloglou

encadrant : Frédéric Peschanski

Faculté Sciences et Ingénierie,
Master 1 Informatique, parcours STL

1 Introduction

La décomposition modulaire d'un graphe est utilisée dans plusieurs problèmes d'optimisation combinatoire. Elle peut être appliquée à des structures plus génériques, comme les 2-structures, qui sont des graphes orientés complets ayant des arcs colorés. Dans ce cas, une réprésentation arborescente de l'hierarchie des modules peut être construite par des décompositions successives des modules maximaux. Plusieurs algorithmes calculant la décomposition des diverses structures discrètes en temps polynomial ont été développés les derniers années.

Le but de ce projet est d'expliquer les principes de l'algorithme de Ehrenfeucht et al. [1] qui calcule l'arbre de décomposition modulaire d'une 2-structure en temps $O(n^2)$, et de tester la performance de son implémentation en langage Python. Ensuite, l'objectif est de comparer cette implémentation avec celle proposée par le logiciel SageMath pour les graphes non-dirigés.

2 Décomposition modulaire

Dans le premier chapitre on donne les définitions nécessaires pour la compréhension de la notion de module et de la décomposition d'un graphe. Ensuite, on explique la décomposition arborescente en modules forts maximaux.

2.1 La notion du module

Une 2-structure est une coloration des arcs d'un graphe complet orienté à k couleurs numérotées de 0 à k-1. On peut transformer un graphe orienté G en une 2-structure à deux couleurs en donnant à chaque arc u, v la couleur 1, et à chaque arc du graphe complémentaire à G la couleur 0.

Un module d'un graphe $G(V, E)$ non-orienté est un sous-ensemble constitué des noeuds qui partagent le même voisinage. Dans le cas d'une 2-structure cette définition prend en compte le sens et les couleurs des arcs (image 1).

Définition 1. Soit M un module d'une 2-structure G, alors $u, v \in M$ ssi pour tout $z \in V - M$, zv et zu ont la même couleur, ainsi que vz et uz.

Si cette propriété n'est pas vérifiée, on dit que z distingue u et v. On note que pour un graphe orienté, un noeud en dehors de M soit partage des arcs de même direction avec tous les noeuds dans M, ou n'est lié à aucun noeud de M. On dit aussi que M est uniforme par rapport à tout noeud dans $V - M$. Les modules d'un graphe non-orienté G forment une famille partitive [2] qui possède le caractéristique que l'union, l'intersection, la différence et la différence symétrique de deux modules de G est aussi un module de G. Un module X est premier ou fort si pour tout autre module Y de G une des trois propositions est vérifiée : X $\subseteq Y$, $Y \subseteq X$ *ou* $X \cap Y = \emptyset$. Les singletons contenant un seul noeud et l'ensemble des noeuds V sont les modules triviaux de G.

Une rélation d'équivalence transitive et symétrique est définie par la distinction des noeuds du graphe. Pour tout noeud x, y et z, si $v \in V$ ne distingue pas x et y ni y et z, alors v ne distingue pas x et z. L'ensemble des classes d'équivalence de cette relation est l'ensemble des modules de G. Si on choisit un représentat pour chaque module on peut construire le graphe quotient $g = G/R$ (image 2). Il est clair que si M et L sont des modules, alors tous les noeuds de M sont liés aux noeuds de L avec des arcs de même sens et couleur.

Définition 2. Le graphe quotient $g = G/R$ est le graphe induit par l'ensemble des représentats des modules de G.

On note que la relation d'inclusion des modules de G est héritée par g. Si $A \cup B$ est un module dans G, alors $\{a, b\}$ est un module dans g, où a et b sont les représentants de A et B respectivement.

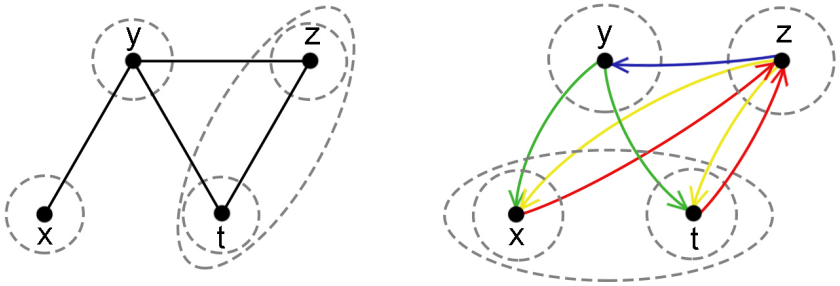


Image 1. Partition en modules d'un graphe non-orienté et d'une 2-structure.

Image 2. Graphes quotients correspondants aux graphes de l'image 1.

2.2 Partition modulaire maximale

Il y a un nombre exponentiel de décompositions modulaires possibles d'un graphe, c'est-à-dire des partitions de son ensemble des noeuds dont les éléments forment des modules. Cependant celle qui ne contient que les modules premiers de taille maximale au sens de l'inclusion est unique [2].

Définition 3. La partition modulaire maximale d'un graphe G est la partition constituée des modules premiers maximaux.

Les modules premiers sont deux à deux soit disjoints soit inclus l'un dans l'autre. Cette ordre partielle d'appartenance peut être répresentée par une structure arborescente, dont les feuilles sont les modules triviaux formés par les noeuds du graphe, et la racine l'ensemble V. On observe directement que le nombre des modules dans l'arbre est de l'ordre de $O(|V|)$.

Une propriété intéressante de la partition modulaire maximale est le fait que tous les noeuds d'un module M sont liés aux noeuds contenus dans un autre module-

fils de l'ancêtre directe de M [1] avec le même type (direction et couleur) de lien. Si on introduit des étiquettes sur chaque module-noeud de l'arbre, on obtient une représentation complète du graphe décomposé qui le décrit sans ambiguité. Pour les graphes non-orientés trois étiquettes sont définies : Un noeud caractérisé comme **série** désigne un module qui est constitué d'arêtes qui lient les noeuds de chaqu'un de ses fils avec les noeuds de tous les autres fils. On appele **parallèle** un module-noeud sans arêtes entre noeuds appartenent à des fils différents. Dans le cas où il n'y a pas le même type de lien entre les noeuds des fils de M on dit que M est primitif.

La décomposition d'un graphe appartenant à une famille de graphes quelconques en sa partition modulaire maximale offre la possibilité de construire des heuristiques pour plusieurs problèmes d'algorithmique difficiles. Elle nous permet de réduire le graphe à un arbre sur lequel on sait appliquer des méthodes de complexité polynomiale.

2.3 Graphes orientés et 2-structures

Pour décrire les liens d'une 2-structure, il faut définir le sens des arcs et ses couleurs. Dans ce cas les étiquettes de l'arbre de décomposition possèdent deux valeurs. Ici un autre nomage est utilisé pour le sens des arcs, qui caractérise comme complets les modules dont les arcs ont la même couleur et comme linéaires les modules pour lesquels il existe une numérotation de ces fils telle que pour tout i,j avec i<j tous les arcs i,j ont la même couleur et tous les arcs j,i ont une autre couleur. La deuxième valeur de l'étiquette est une liste de couleurs de longueur un pour les modules complets et deux pour les linéaires. Les modules primitifs sont ceux qui n'appartiennent à aucune de ces deux catégories et peuvent avoir des arêtes de couleurs différentes liant chaque paire de ses fils. Puisqu'un graphe non-orienté est un graphe orienté avec des arcs dans les deux sens, on observe que les étiquettes série et parallèle sont traduites dans le cas d'un graphe orienté à un complet de couleur 1 et 0 respectivement.

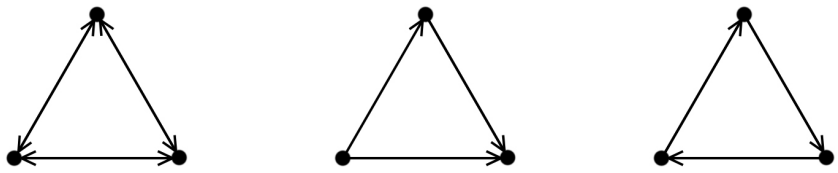


Image 3. Module complet, linéaire et primitive.

1. Cette propriété est vraie pour tous les ancêtres de M. En fait, les noeuds d'un pair de modules quelconque ont le même type de liens entre eux.

Image 4. Un sous-ensemble quelconque des noeuds d'un module complet est un module, contrairement aux modules linéaires, où seulement des noeuds consécutifs selon un certain ordre totale forment des modules. Il y a deux tels ordres : $x < y < z < t$ et $t < z < y < x$. Les deux modules du graphe linéaire sur l'image ne sont pas des modules forts maximaux, puisque ils s'intersectent. Les seuls modules forts maximaux sont les modules triviaux, et ce sont exactement ceux qui apparaîssent sur l'arbre de décomposition, qui pour ces deux graphes consisterait à une racine et quatre feuilles.

3 Algorithme de décomposition modulaire

Plusieurs algorithmes de décomposition existent de complexité polynomiale à linéaire selon le type de structure décomposée. L'algorithme développé par Ehrenfeucht et al. [1] construit l'arbre de la partition maximale pour les 2-structures symétriques [2] ou non-symétriques en temps $O(n^2)$ en se basant sur une partition en modules forts du graphe induit par $V - \{v\}$, où $v \in V$ est tiré au hasard. L'étape qui suit la décomposition associée à cette partition est un tri des modules qui sert comme étape intermédiaire pour trouver le module dans lequel il faut attacher v pour passer de la partition de $V - \{v\}$ à celle de V.

Dans la suite de ce paragraphe on considère le cas plus générique des 2-structures non-symétriques.

L'algorithme choisit $v \in V$ au hasard et partitionne $V - \{v\}$ en modules uniformes par rapport à v et insère les modules dans une file L. Puis il partitionne récursivement chaque module M dans L par rapport aux noeuds qui se trouvent en dehors de M et ne sont pas encore traités pendant les itérations précedentes. Ainsi chaque arc est traité au plus une fois et la compléxité de cette étape est $O(|E|)$.

Le graphe quotient g associé à cette partition contient que des modules qui possèdent la propriété suivante [1] :

Propriété 1. Un module X est un noeud de g ssi pour tout module Y du graphe initial tel que $X \subset Y$, Y contient v.

On considère que v est le seul élément de son module $\{v\}$ et on rajoute son représentant v' dans g. À partir du graphe résultant un autre graphe orienté est construit, noté $G' = G(g, v)$, dont les noeuds sont l'ensemble des noeuds de g privé de $\{v\}$. Il existe un arc de x vers y dans G' si x distingue v' et y.

Propriété 2. Chaque arc x, y de G' signifie que si on considère le plus petit ancêtre commun p de x, y et v dans l'arbre de décomposition, alors y et v appartiennent à des modules-fils de p différents.

Le graphe G' révèle alors les relations entre ces modules. Ces deux propriétés sont importantes car elles nous permettent de définir une procédure récursive pour résoudre la décomposition (image 5).

2. L'algorithme calcule la décomposition d'un graphe non-orienté, puisque c'est une 2-structure symétrique à deux couleurs, et d'un graphe orienté considéré comme une 2-structure à deux couleurs.

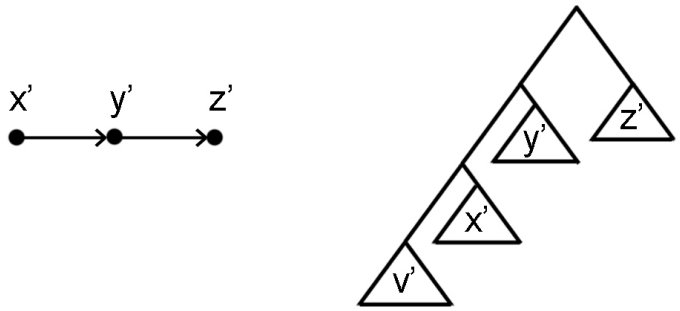


Image 5. Si le graphe G' est un chemin x'y'z', alors par les propriétés 1 et 2 l'ancêtre directe de chaqu'un des modules représentés par x', y' et z' a un fils qui contient v. De plus, v ne peut pas appartenir au module de y' ni de z'. Donc l'arbre de décomposition du module de z' est situé à une branche différente de ce de v' (de même pour y').

Propriété 3. S'il y a un composant connexe de taille supérieure à 1 dans G', l'union de ses modules est un module primitif.

En effet, si on choisit un arc x, y du composant connexe on voit que x distingue v et y. Mais il y a aussi un chemin de y vers x, et y distingue aussi x et v. De plus, x et y ne peuvent pas se trouver dans le même module différent de V, car cela contredit la propriété 1.

Pour reconnaître les modules primitifs on crée le graphe G" des composants connexes de G'. On choisit un puit y dans G" et on crée dans l'arbre de la décomposition modulaire maximale deux branches à partir de la racine u. Une des branche est réservée pour l'arbre de décomposition de y et l'autre pour l'arbre de tous les modules qui restent dans G", qui seront calculés pendant l'itération suivante. Le puit est ensuite enlevé et la procédure est appliquée récursivement jusqu'à décomposer tous les modules dans G".

1. **décomposition_modulaire(V, E)** =
2. Créer un noeud t dans l'arbre de la décomposition
3. $décomposer(t, V)$
4. **retourner** t

1. **décomposer(t,V)** =
2. **si** $|V| = 1$ **alors**

3. $t = V$
4. **sinon**
5. $u := t$
6. Choisir $v \in V$ au hasard
7. $P = partitionner(V - \{v\}, v)$
8. Créer les graphes g, G' et G" à partir de P
9. **tant que** $G"! = \emptyset$ **faire**
10. Enlever les puits $p_1, ...p_n = Y$ de G"
11. $traiter(u, Y)$
12. Créer un fils vide u' de u
13. $u := u'$
14. $u = \{v\}$

1. **traiter(u,Y)** $=$
2. **si** $|Y| = 1$ et $|p_1| > 1$ **alors**
3. // s'il y a un seul puit qui contient plus qu'un module
4. $étiquette(u)$ ='PRIME'
5. **sinon si** la couleur d'un arc de v vers F est la même que l'arc
6. de F vers v **alors**
7. $étiquette(u)$ ='COMPLETE'
8. $couleur(u) = couleur(v, F)$
9. **sinon**
10. $étiquette(u)$ ='LINEAR'
11. $couleur(u) = [couleur(v, F_1), couleur(F_1, v)]$
12. // F1 est un module de F
13. **pour** tout module $M \in \bigcup p_i$
14. $t' = décomposition\_modulaire(M)$
15. // fusion
16. **si** u et t' sont complets et de même couleur **ou**
17. linéaires et de même couleur **alors**
18. ajouter les fils de t' aux fils de u
19. **sinon**
20. ajouter t' aux fils de u

Pseudocode 1. La procedure partitionner(V,v) permet de partitionner V en modules uniformes par rapport à v, comme décrit plus haut.

Si y contient plus qu'un module, u est primitif et la racine de l'arbre de décomposition de y est fusionnée avec u. Si le père de v est complet, g contient un seul module et G" un seul puit. D'autre part, si le père de v est linéaire, g contient un ou deux modules [3] et G" un ou deux puits. De plus, la couleur de l'arc vy et différente de celle de yv si

3. Si v est le premier ou le dernier noeud du module linéaire selon l'ordre défini entre les modules, qui montre les couleurs des arcs, alors g contient un seul noeud. Sinon il contient le représentat du module contennant les modules avant v et celui des modules après v.

le module qui contient v et y est linéaire, et la même si ce module est complet (image 6). Par conséquent il suffit de connaître le nombre de puits et le nombre de modules dans le puit, en particulier s'il n'y a qu'un seul module, pour décider si le module correspondant est primitive, complet ou linéaire.

À chaque itération une fusion de u avec la racine d'une branche est effectuée si les deux sont complets de même couleur ou linéaires de mêmes couleurs. Un sous-ensemble quelconque des fils d'un module M complet est aussi un module, mais la partition maximale est unique. Pour cela les sous-ensembles formés des fils de M n'apparaîssent pas dans l'arbre. Les fils de M retenus dans l'arbre sont les plus petits au sens de l'inclusion parmi tous ses fils possibles. Le même traitement est effectué si M est linéaire (image 7).

L'algorithme construit des branchements séparant à chaque itération le module qui contient v par un autre qui ne le contient pas, jusqu'à trouver la place de v, qui est positioné sur la branche qui reste vide après que l'algorithme est fini (image 5).

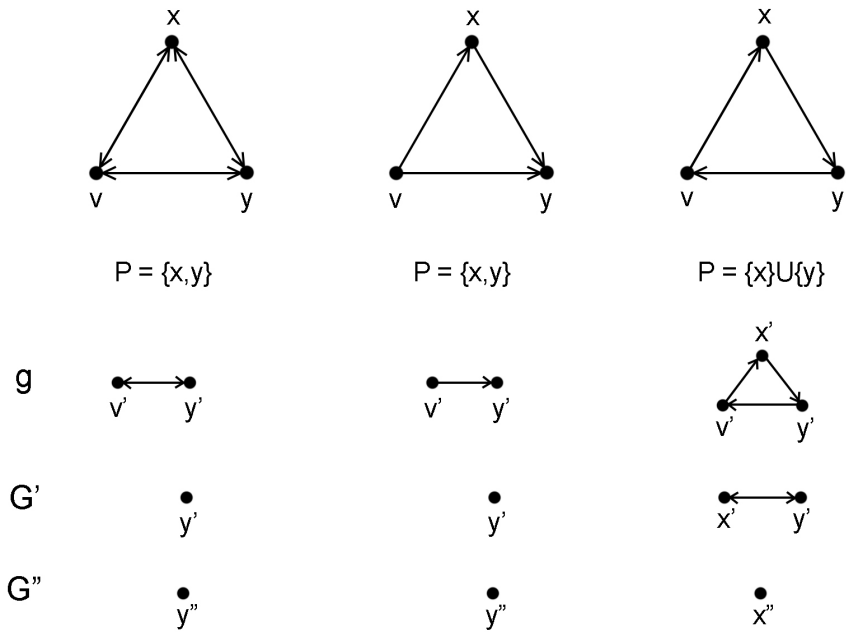


Image 6. Les étapes du premier appel à la procedure décomposer(t,V) *pour les graphes de l'image 1. Le noeud choisit au hasard est v.*

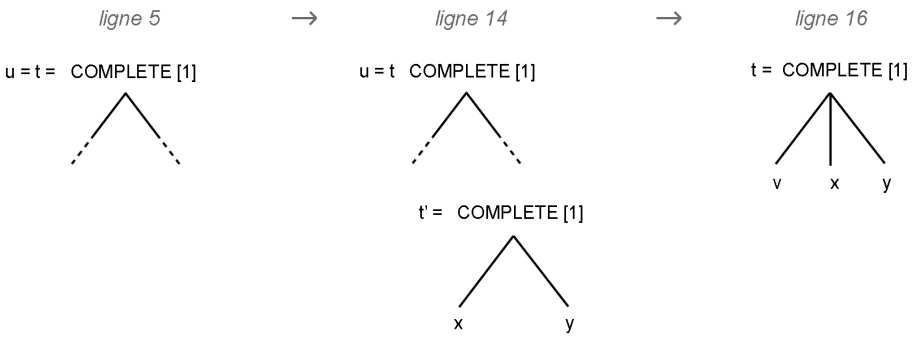


Image 7. Etapes de la procédure traiter(u,V) pour le premier graphe de l'image 6.

4 Contribution au SageMath

SageMath est utilisé pour la résolution des problèmes d'algèbre, de combinatoire, de théorie de nombres et des graphes. C'est un logiciel libre écrit principalement en Python et Cython.

La version 9.2 offre une fonction pour la décomposition modulaire des graphes non dirigés basée sur une implémentation de l'algorithme de Habib et Maurer [3] ayant une complexité théorique de $O(n^3)$ [4].

4.1 Optimization du code

L'objectif de ce projet étant d'enrichir à terme la librairie du logiciel SageMath avec une fonction pour la décomposition de 2-structures et de graphes dirigés, il est nécessaire de s'assurer que l'implémentation de l'algorithme décrit dans le premier chapitre est suffisament performante en terme de temps d'exécution. À ce titre, des testes de performance ont étaient conçus pour découvrir les eventuels points d'amélioration. Les parties du code qui consomment un grand pourcentage du temps d'exécution ont été détectés à l'aide de l'outil de profiling du language Python et du module d'extraction des statistiques dépuis le fichier contenant les résultats de cette opération. Les améliorations effectuées comportent le remplacement des structures de données qui alourdissent l'interpréteur et la réécriture des certaines routines.

```
Wed May  5 20:12:52 2021    test2_output_v1_5_5.dat

15348851 function calls (15348609 primitive calls) in 5.448 seconds

Ordered by: internal time

 ncalls  tottime  filename:lineno(function)
4326940    2.225  twostructure.py:83(color_of)
1081821    1.245  moddecomp.py:9(distinguish)
8679384    1.073  {method 'get' of 'dict' objects}
   9358    0.576  moddecomp.py:15(partition_module)
1127530    0.164  {method 'add' of 'set' objects}
     87    0.089  moddecomp.py:42(maximal_modules)
   8692    0.025  twostructure.py:24(edge)
     87    0.016  moddecomp.py:75(build_quotient)
  188/1    0.008  moddecomp.py:280(modular_decomposition)
  53673    0.007  {built-in method builtins.len}
    187    0.004  twostructure.py:13(slice)
  19629    0.003  {method 'append' of 'list' objects}
```

Tableau 1. Performances du code initial.

La version initiale du code définit une class TwoStructure pour la représantation d'une 2-structure qui utilise un dictionnaire pour stocker les arcs, dont la clef est l'indice de la source et la valeur est un dictionnaire contenant les destinations associés à la couleur. Ainsi l'appel à la fonction qui décide si un noeud distingue deux autres nécessite d'accèder aux valeurs des deux dictionnaires. De même pour la fonction qui retourne la couleur l'un arc. La complexité théorique d'une recherche dans un dictionnaire est $O(1)$ grâce au hachage, mais l'implémentation des opérations liées aux dictionnaire en Python est très chronophage (tableau 1). La première tentative alors est de déterminer les structures de données les plus adaptés pour représenter les arcs et les couleurs d'une 2-structure et de concevoir une nouvelle implémentation pour les parties qui étaient basées sur les dictionnaires.

La deuxième version propose une implémentation améliorée de ces fonctions qui est possible en n'utilisant que des ensembles (set) ou des listes. Les sets sont préférés par rapport aux listes puisque leurs éléments peuvent être retrouvés en temps O(1) grace au hachage, contrairement à une recherche dans une liste, qui est effectuée en temps O(n). Puisque le nombre des couleurs d'une 2-structure est beaucoup plus petit que le nombre de ses arcs, un stockage qui associe des couleurs à des couples source-destination favoriserait la vitesse de calcul, qui s'effectuera en temps $O(K) \cdot O(1)$. Une solution bucket-sort stocke chaque arc dans un ensemble d'arcs qui contient que les arcs de même couleur. Les ensembles sont stockés dans une liste dont l'indice correspond à la couleur des arcs contenus dans cet ensemble. Une partie du code source des deux versions est donnée ci-dessous.

```
class TwoStructure:
    def __init__(self):
        # the graph of a two structure
        self.nodes = set()
        self.edges = dict()
        self.colors = dict()


    def color_of(self, v1, v2):
        edge = self.edges.get(v1)
        if edge is None:
            return 0
        col = edge.get(v2)
        if col is None:
            return 0
        else:
            return col


    def distinguish(g, w, v1, v2):
        """Does w distinguish v1 and v2 in (two structure or graph) g?
        """
        return (g.color_of(w, v1) != g.color_of(w, v2)) \
            or (g.color_of(v1, w) != g.color_of(v2, w))
```

```
    def partition_module(g, w, module):
        todo = [module]
        refined = []
        while len(todo) > 0:
            module = todo.pop()
            for v1 in module:
                v1_yes = set()
                v1_no = {v1}
                for v2 in module:
                    if v2 != v1:
                        if distinguish(g, w, v1, v2):
                            v1_yes.add(v2)
                        else:
                            v1_no.add(v2)
                if len(v1_yes) > 0:
                    todo.append(v1_yes)
                    refined.append(v1_no)
                    break
                else:
                    refined.append(v1_no)
        return refined
```

Code source 1. Extrait du code initial. La méthode color_of *contient deux appels à* dist.get*, ce qui augmente significativement le temps d'exécution total.*

```
class TwoStructure:
    def __init__(self, graph=None):
        # the graph of a two structure
        self.nodes = set()
        self.colors = [set()] # list of sets,
        # self.colors[i] contains the set of edges of color i
        self.modules = [] # only for quotient graph,
        # contains the nodes of the graph if they are modules


    def color_of(self, v1, v2):
        for color in range(len(self.colors)):
            if (v1, v2) in self.colors[color]:
                return color
        return 0


    def distinguish(g, w, v1, v2):
        """
        returns true if w distinguishes v1 and v2 in two-structure g
        """
        w_v1_col = g.color_of(w, v1)
        if (w,v2) in g.colors[w_v1_col]
        or (w_v1_col==0 and g.color_of(w,v2)==0):
            v1_w_col = g.color_of(v1, w)

            if (v2, w) in g.colors[v1_w_col]
```

```
        or (v1_w_col==0 and g.color_of(v2,w)==0):
            return False
    return True


def partition_module(g, w, module):
"""
partitions module in strong maximal modules
:param g: a two-structure
:param w: the first node used for partitioning module
:param module: a set of nodes of g to partition
:return: a list of sets of nodes
"""
modules = [ [set() for c in g.colors] for c_ in g.colors ]
is_empty = [ [True for c in g.colors] for c_ in g.colors ]
for n in module :
    c_w_n = g.color_of(w, n)
    c_n_w = g.color_of(n, w)
    modules[c_w_n][c_n_w].add(n)
    if is_empty[c_w_n][c_n_w] : is_empty[c_w_n][c_n_w] = False
flattened = [module for l in modules for module in l]
for v in is_empty:
    for var in v:
        if var:
            flattened.remove(set())
return flattened
```

Code Source 2. Extrait du code de la deuxième version.

Après cette première amélioration, les mêmes tests ont montré une diminution significative du temps d'exécution des fonctions *color_of* et *distinguish*. En même temps on observe que l'ajout des listes en comprehension dans la deuxième version a augmenté le temps d'exécution total. Une deuxième amélioration consiste alors au changement de l'implémenation de la fonction *slice* qui utilise ces listes. Cette intervention a diminué à la moitié le temps d'exécution de cette fonction.

```
Wed May  5 20:12:55 2021    test2_output_v2_5_5.dat

103581 function calls (103382 primitive calls) in 0.152 seconds

Ordered by: internal time

   ncalls  tottime  filename:lineno(function)
    20107    0.100  src_twostructure_v2.py:78(color_of)
      321    0.014  src_moddecomp_v2.py:24(partition_module)
      187    0.008  src_twostructure_v2.py:24(slice)
    27715    0.004  {method 'add' of 'set' objects}
       91    0.004  src_moddecomp_v2.py:44(maximal_modules)
      321    0.003  src_moddecomp_v2.py:32(<listcomp>)
    188/1    0.003  src_moddecomp_v2.py:326(modular_decomposition)
    13671    0.003  {method 'remove' of 'list' objects}
    23258    0.003  {built-in method builtins.len}
      192    0.001  src_twostructure_v2.py:35(edge)
     9885    0.001  {method 'append' of 'list' objects}
  187/175    0.001  src_scc.py:33(visit)
       91    0.001  src_moddecomp_v2.py:86(build_quotient)
       91    0.001  src_moddecomp_v2.py:175(build_distinction_graph)
      321    0.001  src_moddecomp_v2.py:33(<listcomp>)
```

Tableau 2. Performances de la version finale.

Ensuite tous les fonctions ont été examinées pour retrouver des eventuels problèmes d'algorithmique, et leur code a été réécrit de façon plus lisible en évitant des boucles ou des appels non nécessaires.

L'implémentation de la fonction *partition_module(g,w,module)*, qui construit des sous-modules de *module* uniformes par rapport au noeud w, a été remplacée par une implémentation en $O(|E|)$ basé sur l'idée que pour partitionner un module en sous-ensembles des noeuds distingués par w il suffit de parcourir une seul fois les noeuds du module et de les séparer selon la couleur de l'arc qui les lie à w. La version initiale utilise une implémentation qui examine plusieurs fois le même arc pour arriver à la décomposition finale (code source 1). Au début un noeud v_1 de l'ensemble à décomposer est choisi et les arcs entre w et tout noeud v_2 de l'ensemble sont examinés pour décider si w distingue v_1 et v_2. Ensuite tous les noeuds distingués de v_1 par w doivent être reexaminés. Cela signifie que les arcs de w vers ces noeuds doivent être parcourus encore une fois, et cela est répété jusqu'à trouver un module qui ne peut plus être raffiné. Cela résulte à une complexité de $O(K^2 \cdot n)$, où n est la taille de *module*, contrairement à la description de la décomposition en modules uniformes décrit en [1], qui propose une solution qui utilise une seule fois chaque arc. La version améliorée implémente cette solution pour la redéfinition de la fonction *partition_module(g,w,module)* (code source 2), où la distinction est calculée avec une seule boucle en temps $O(2 \cdot n)$. Chaque noeud v est placé directement dans son module en regardant la couleur de wv et de vw une seule fois. Les noeuds sont séparés dans des ensembles (set) selon la couleur des arcs les liant à w. Des ensembles vides stockés dans une liste de 2 dimensions sont crées pour chaque couple des couleurs possible.

La case i,j contient les noeuds v pour lesquels wv est de couleur i et vw de couleur j. Ensuite chaque noeud est placé dans l'ensemble correspondant et les ensembles vides sont supprimés. La différence entre les deux complexités théoriques de $O(K^2 \cdot n)$ et $O(2 \cdot n)$ n'est pas grande, mais Python est lente avec le traitement des boucles, et le remplacement de deux boucles embriqués par une seule a amélioré la performance de la deuxième version.

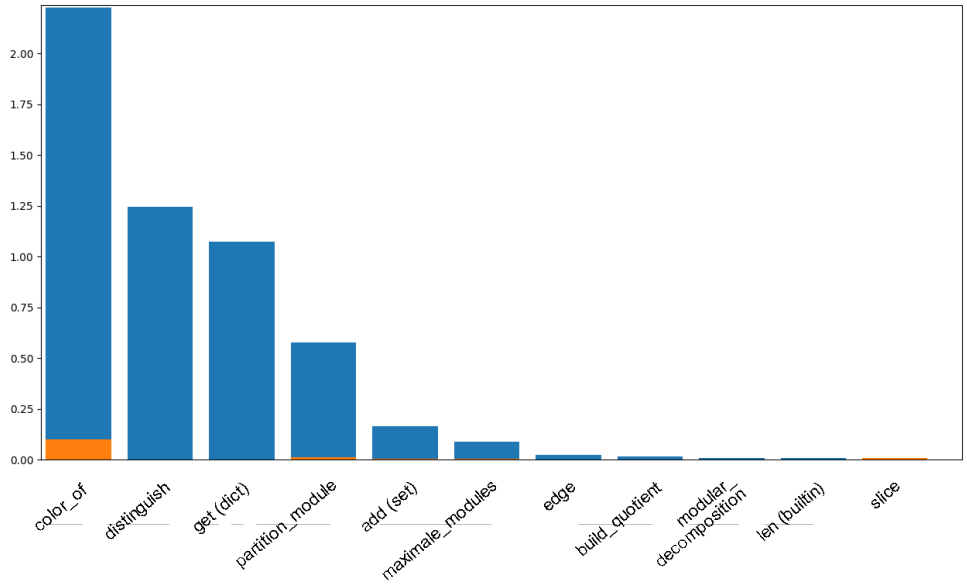
La définition de la classe des 2-structures a été enrichie par une variable d'instance qui aide a alléger les fonctions qui manipulent les graphes quotient. Elle stocke les modules d'une 2-structure représentés par les noeuds du graphe quotient dans une liste de telle façon que chaque noeud du graphe quotient soit égal à l'indice du module dans la liste.

4.2 Résultats et perspectives

Après les modifications effectuées sur le 50% des fonctions initiales pour les adaptés au changement des structures de données et pour améliorer leur compléxité, des tests de performances sur des graphes de structure aléatoire [4] de 1000 noeuds et 100 arcs on été effectués pour comparer les deux versions du code. Le temps d'exécution total de la décomposition d'un graphe dirigé a été diminué à 0.152 s en deuxième version de 5.448 s avant les améliorations (graphe 1).

La performance a aussi été comparée avec celle de SageMath sur des graphes non-dirigés. Pour ces tests la version ligne de commande installée du logiciel a été utilisée. L'implémentation de l'algorithme développée dans le cadre du projet est deux fois plus lente que SageMath testée sur des graphes de 100 arcs avec l'interpréteur Python. Concernant les graphes plus grandes, voire 1000 arcs, notre programme est exécuté en 5 minutes environ et SageMath en 0.7 secondes. Cette grande différence impose d'effectuer des tests sur des graphes de types différentes pour déterminer les causes du retard. De plus, il serait intéressant d'écrire une partie du code concernant les 2-structures en Cython, puisque c'est le language utilisé par SageMath pour le module des graphes et est connu pour sa capacité d'optimisation de code.

4. Un générateur de graphes Erdös-Renyi du module networkx a été utilisé pour tous les tests. Le générateur prend en paramètre le nombre de noeuds et une probabilité selon laquelle un arc est choisi parmi les arcs du graphe complet correspondant au même ensemble de noeuds.



Graphe 1. Temps d'exécution en secondes des fonctions avant (bleu) et après (orange) les améliorations de performance pour la décomposition d'un graphe de 100 arêtes.

Références

[1] A. Ehrenfeucht, H. N. Gabow, R. M. MacConnell, and S. J. Sullivan. 1994. An $O(n^2)$ Divide-and-Conquer Algorithm for the Prime Tree Decomposition of Two-Structures and Modular Decomposition of Graphs. Journal of algorithms 16, 2 (1994), 283–294. DOI :https ://doi.org/10.1006/jagm.1994.1013

[2] M. Habib and C. Paul. 2010. A survey of the algorithmic aspects of modular decomposition. Computer science review 4, 1 (2010), 41–59. DOI :https ://-doi.org/10.1016/j.cosrev.2010.01.001

[3] Lokesh Jain. 2017. Modular Decomposition. https ://github.com/sagemath/sage/blob/develop/src/sage/graphs/graph_decompositions/modular_decomposition.py (2021)

[4] M. Habib and M.C. Maurer. On the X-Join decomposition for undirected graphs. 1979. Discrete Applied Mathematics, 3 (1979), 201-207. DOI :http ://dx.doi.org/10.1016/0166-218X(79)90043-X.