Affinage de Partitions et applications aux graphes

Philippe ROBERT

November 20, 2010

1 Introduction

Nous nous plaçons dans le cadre d'un problème où il faut partitionner un ensemble fini E en une partition $\mathcal{P} = X_1, ..., X_n$ vérifiant des propriétés prédéfinies. Par exemple, étant donnée une relation d'équivalence sur E, quelle est la partition en classes d'équivalences de E?

Une première méthode serait de commencer par une partition avec des classes petites, composées d'un seul élément par exemple, puis de les fusionner successivement jusqu'à atteindre une partition correcte.

Dans cette synthèse, au contraire, nous nous intéressons au paradigme de *raffinement de partition*, c'est-à-dire des algorithmes qui commencent avec une partition initiale grossière, par exemple $\mathcal{P}_1 = E$: l'ensemble non partitionné, et vont itérativement subdiviser les classes de $\mathcal{P}_i$ en une autre partition $\mathcal{P}_{i+1}$ jusqu'à ce qu'à une partition correcte $\mathcal{P}_N$.

Exemple: Le tri Quick-Sort peut être vu comme un affinage de partition.

Algorithm 1 Quick-Sort

Problème : tri d'entiers
Entrée : ensemble $E = a_1, ..., a_N$ de N entiers entre 1 à n
Sortie : une partition ordonnée de E

1: $\mathcal{P} = E$. {La partition initiale est E}
2: **tant que** il existe une classe de $\mathcal{P}$ qui ait au moins deux éléments à valeurs distinctes **faire**
3: choisir un élément p de X_i {l'élément pivot}
4: avec $S = \{a_i \in X_i | a_i \leq p\}$, {l'ensemble pivot}
5: remplacer X_i par S et insérer une nouvelle classe $X_i \backslash S$ à droite de X_i (si non vide).
6: **fin tant que.**
7: **return** A

La partition courante est un ensemble ordonné de classes, qui sera initialisée à $\{E\}$, affinée, et sera correcte (triée) quand :

- Il n'y a qu'une seule valeur par classe : $\forall i, \forall (a_p, a_q) \in X_i, a_p = a_q$
- Les classes sont ordonnées l'une par rapport à l'autre par valeurs croissantes : $\forall i < j, \forall a_p \in X_i, a_q \in X_j, a_p < a_q$

A chaque étape, une classe est séparée en deux sous-classes : l'une avec les valeurs inférieures à p, l'autre avec les valeurs strictement supérieures, en maintenant l'ordre entre les classes.

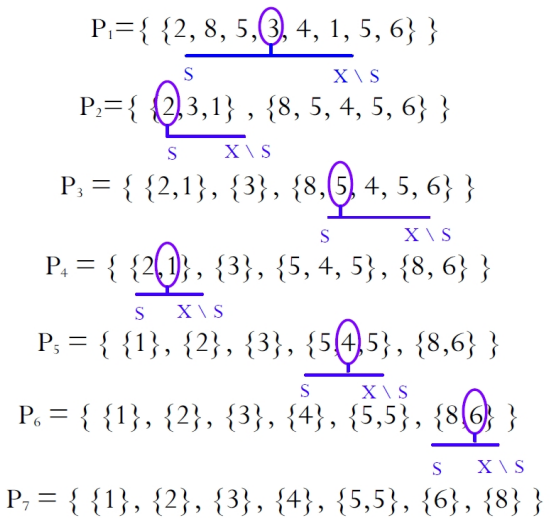


Figure 1: Exécution d'un quicksort. A chaque étape, l'élément pivot p est entouré, et l'ensemble pivot S et le reste $X \backslash S$, sont indiqués. Le choix du pivot est libre, à condition de ne pas prendre la plus grande valeur de la classe.

Dans le cas général, comme l'exemple ci-dessus, on sera amené à utiliser un élément 'pivot' p et un ensemble pivot S créé à partir de p, pour couper les classes de la partition courante.

Contrairement à l'exemple, un affinage de partition peut ne pas être de type 'diviser pour régner' car un pivot pourra modifier plusieurs classes simultanément (cf. problème de la partition fonctionnelle maximum plus loin), ce qui signifie que les sous-problèmes 'partitionner correctement une classe de P' ne sont pas forcément indépendants les uns les autres. De même, un algorithme 'diviser pour régner' ne peut pas forcément être vu comme affinage de partitions, puisque, dans ce dernier cas, une fois deux classes séparées, on ne peut plus échanger d'élément entre celles-ci, ce qui exclut le tri fusion.

Suivant la démarche de la thèse de C.Paul[1], dans une première partie, nous allons formaliser de manière générale l'affinage de partitions, puis présenter dans une seconde partie en détails comment quelques problèmes peuvent être résolus de cette méthode, et enfin, dans une troisième partie, nous montrerons des application sur les graphes.

2 Formalisation

Définition : Soient $\mathcal{P}$ et $\mathcal{Q}$ deux partitions, on dira que $\mathcal{P}$ est plus fine que $\mathcal{Q}$ (ou que $\mathcal{Q}$ est plus grossière que $\mathcal{P}$), si $\mathcal{P}$ peut s'obtenir à partir de $\mathcal{Q}$ en subdivisant les partitions de $\mathcal{Q}$

$$\mathcal{P} \prec \mathcal{Q} \Leftrightarrow \forall X \in \mathcal{P}, \exists Y \in \mathcal{Q} | X \in Y.$$

Affiner une partition $\mathcal{P}$ par un ensemble S consiste à subdiviser toutes les classes X qui s'intersectent 'strictement' avec S, i.e. telles que $\emptyset \neq (X \cap S) \neq X$. Les détails sont donnés ci-dessous.

2.1 Algorithme général

Les ingrédients principaux sont :

1: La gestion de pivots. il faut maintenir une liste d'éléments pivots. Ces 'pivots' sont des éléments qui auront la propriété de discriminer les individus d'au moins une classe. Tant que la partition n'est pas correcte, il faut trouver des pivots, sans quoi l'algorithme rendrait une partition inachevée.

Deux fonctions s'en chargeront : $Initialise$, quand la liste de pivots est vide, et $AjouteNouveauxGroupesPivots$ durant les étapes d'affinage, si on peut éventuellement identifier des pivots futurs.

Les pivots sont stockés par groupe dans $GroupesPivots$. Par exemple, pour le quicksort, il est intéressant d'affiner selon plusieurs pivots en même temps du moment qu'ils sont dans des classes différentes. On peut dans ce cas prendre à chaque étape un groupe constitué de 1 pivot par classe non correcte.

2: L'affinage. Pour affiner la partition selon un élément pivot p, on calcule son ensemble pivot S, et toutes les classes qui s'intersectent avec S seront coupées en deux : l'intersection avec S et le reste s'il existe.

On uitilisera la fonction $InsererADroite$, pour choisir comment sont insérées les nouvelles sous-classes. L'ordre des éléments dans une classe n'a pas d'importance. En revanche, maintenir un ordre entre les classes peut être nécessaire, d'où l'intérêt d'une telle fonction.

Algorithm 2 Schéma global d'un affinage de partition

Problème : Construire une partition vérifiant une contrainte C
Entrée : une partition $\mathcal{P}$ d'un ensemble E, qui ne vérifie pas forcément C
Sortie : un affinage de la partition de départ, qui vérifie C.

$Entree = \mathcal{P}$
$GroupesPivots = \emptyset$
tant que $(GroupesPivots = Initialise(\mathcal{P})) \neq \emptyset$ **faire**
  tant que $\exists G \in GroupesPivots$ **faire**
    Pour tout $p \in G$ **faire**
      $S = EnsemblePivot(p, \mathcal{P}, G)$
      {On affine maintenant P selon S}
      Pour tout $X \in \mathcal{P}$, tel que $\emptyset \neq (X \cap S) \neq S$ **faire**
        $X_a \leftarrow X \cap S$
        $X \leftarrow X \backslash S$
        if $InsererADroite(X, X_a, p)$ **then**
          Insérer X_a à droite de X dans $\mathcal{P}$
        else
          Insérer X_a à gauche de X dans $\mathcal{P}$
        fin si.
        $GroupesPivots = AjouteNouveauxGroupesPivots(X, X_a, \mathcal{P}, GroupesPivots)$
      fin pour.
    fin pour.
  fin tant que.
fin tant que.
return P

On détaillera concrètement comment peuvent choisies les quatre fonctions $Initialise$, $EnsemblePivot$, $InsererADroite$ et $AjouteNouveauxGroupesPivots$ sur des exemples, dans la partie 3.

2.2 Structure de données

C.Paul a proposé une structure de liste doublement chaînée pour les éléments de E, où tous les éléments d'une même classe sont consécutifs, et où les classes ont un pointeur vers leur premier et dernier élément. Par ailleurs, chaque élément a un pointeur vers sa classe.

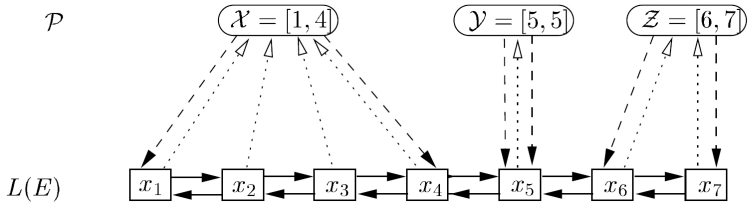


Figure 2: Structure de liste doublement chaînée pour stocker une partition en cours d'affinage.

Cette structure apporte une complexité intéressante dans les cas suivants : (avec E l'ensemble à partitionner, $|E| = \mathrm{N}$)

1. balayer les éléments d'une classe X, en temps $|X|$. (les éléments entre le pointeur début et fin de la classe)

2. balayer les classes dans l'ordre de la liste, en temps N (il suffit de balayer la liste des éléments et de lire le pointeur vers leur classe au fur et à mesure)

3. couper une classe X selon S, en temps $|X|$, et par conséquent, affiner $\mathcal{P}$ par S en temps N (en effet, voici le détail des opérations pour couper une classe :

   - On se place dans le cas où $\emptyset \neq (X \cap S) \neq S$. On veut partager X en $X_a = X \cap S$ et $X_{reste} = X \backslash S$. La case X_a est créée en temps constant dans la liste des classes. La classe X deviendra X_{reste}.
   - Si on veut placer X_a à gauche de X_{reste}, on balaye les éléments $x \in X$. Chaque fois que l'on a aussi $x \in S$, x est déplacé en temps constant juste avant X_{reste} dans la liste (i.e. exactement entre la fin de X_a et le début de X_{reste}). Soit au plus $|X_a|$ déplacements en temps constant.
   - Les pointeurs (premier élément, dernier élément) des classes X_a et X_{reste}, sont mis à jour ensuite en temps constant. (il suffit d'avoir compté combien d'éléments ont été mis dans X_a et de déplacer correctement ces pointeurs).

   D'où enfin la complexité linéaire de l'étape d'affinage pure.

2.3 Correction

On emploiera souvent le même motif afin de prouver la correction des algorithmes d'affinage de partition. Il s'agit de s'appuyer sur un invariant qui assurera que, si la partition n'est pas encore correcte, alors il existe une partition correcte, et il existe une manière de l'améliorer (un pivot).

On décomposera cet invariant en deux propriétés qui seront maintenues vraies au long de l'exécution de l'algorithme :

- une propriété A qui assure que il existe une solution au problème qui soit une partition plus fine que la partition en cours. (i.e. il est encore possible d'obtenir une solution en affinant P).

- une propriété B, qui assure que, si la partition P en cours n'est pas une solution au problème, alors il existe un pivot qui affine strictement P.

Pour le Quick-Sort,

- la propriété $\mathcal{A}$ serait "'Les classes sont bien ordonnées : si x est dans une classe à gauche de y, alors $x < y$"'. En effet, on voit que si $\mathcal{A}$ est vraie, alors on pourra encore trier $\mathcal{P}$ et réciproquement.

- la propriété $\mathcal{B}$ serait "'Les classes ne contiennent qu'une valeur"'. En effet, si une classe a deux éléments a et b, tels que $a < b$, en prenant a comme pivot, on affine strictement cette classe.

Pour montrer qu'il est correct, il ne restera plus qu'à prouver que l'algorithme de Quick-Sort vérifie $\mathcal{A}$ par induction, et que si il n'a plus de pivots, alors $\mathcal{B}$ est vraie.

3 Premières Applications

3.1 Partition en sommets jumeaux

Définition Soit $G = (V, E)$ un graphe non orienté, soit u un sommet, on note $N[u] = N(u) \cup u$ avec $N(u)$ le voisinnage de u. On dit que deux sommets a et b sont :

- Vrais-Jumeaux ssi ils ont les mêmes voisins et ils sont eux-même voisins, i.e. $N[a] = N[b]$.
- Faux-Jumeaux ssi ils ont les mêmes voisins, mais ne sont pas voisins, i.e. $N(a) = N(b)$
- Jumeaux s'ils sont l'un ou l'autre.

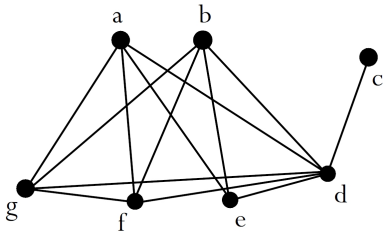


Figure 3: Exemple. Les sommets a et b sont des faux jumeaux alors que les sommets g et f sont des vrais jumeaux. f et d ne sont pas jumeaux à cause de c et e. Une décompostion en classes de jumeaux serait $\{\{a, b\}; \{g, f\}; \{c\}; \{d\}; \{e\}\}$

u	N[u]	N(u)
a	adefg	defg
b	bdefg	defg
f	abdfg	abdg
g	abdfg	abdf

Figure 4: Voisinnages (large et strict), de deux sommets respectivement faux− et vrais− jumeaux.

Proposition La relation "être jumeaux (au sens large)", est une relation d'équivalence. Une classe d'équivalence de cette relation sera composée uniquement de vrais jumeaux ou uniquement de faux jumeaux, mais jamais des deux.

Démonstration :

1. Il est évident que les trois relation différentes 'être jumeaux', sont réflexives et symétriques. La relation 'être vrais jumeaux' est transitive, 'être faux jumeaux' aussi.

2. Maintenant, que se passe-t-il si a et b sont vrais jumeaux, mais b et c sont faux jumeaux ? Si a et c sont voisins, alors ils sont vrais jumeaux, et par transitivité, c et b aussi. Contradiction. De même, si a et c ne sont pas voisins, alors ils sont faux jumeaux, et, par transitivité, a et b aussi. Contradiction.

3. Enfin, cela montre qu'un sommet ne peut pas être jumeaux à la fois de manière 'vraie' et 'fausse' avec des sommets différents, donc, si a et b, et b et c sont jumeaux au sens large, ils le sont forcément de la même manière, et on peut utiliser la transitivité des vrais ou des faux jumeaux pour conclure que a et c sont aussi jumeaux. D'où la transitivité des jumeaux au sens large.

Corollaire Dans les classes d'équivalence de la relation "'être jumeaux"', les sommets forment soit une clique, soit des sommets indépendants.

Formulation du problème : Soit un graphe $G = (V, E)$, quelle est sa décomposition en classes de sommets jumeaux ?

Pour transposer ce problème en terme d'affinage de partition, on peut remarquer que deux éléments a et b sont dans deux classes différentes (i.e. pas jumeaux du tout), si et seulement leur voisinnage extérieur (sans compter ces deux sommets), n'est pas le même. En particulier, il existera un sommet s qui sera voisin avec l'un mais pas l'autre (de a ou b). C'est un pivot !

La stratégie sera alors, pour chaque sommet s, de s'assurer qu'un voisin et un non voisin ne seront pas dans la même classe, et d'affiner la partition le cas échéant. L'ensemble pivot de s sera tout simplement $N[s]$ son voisinnage large.

Algorithm 3 Partition des sommets jumeaux

Entrée : un graphe $G = (V, E)$. $\mathcal{P} = V$
Sortie : une partition de G en classes soit de vrais, soit de faux jumeaux.

```
GroupesPivots = v1, v2...vn {tous les sommets seront pivot une fois}
tant que ∃G ∈ GroupesPivots faire
  Pour tout p ∈ G {i.e. pour tout sommet} faire
    S = EnsemblePivot(p, P, G) = N[s]
    Pour tout X ∈ P, tel que ∅ ≠ (X ∩ S) ≠ S faire
      X_a = X ∩ S ; X = X\S
      Insérer X_a à droite de X dans P
      {Pas besoin d'ajouter de pivot}
    fin pour.
  fin pour.
fin tant que.
return P
```

Correction Prenons A la propriété : 'si des sommets sont jumeaux, alors ils appartiennent à la même classe dans P.', et B : 'si il y a des non-jumeaux dans la même classe, alors il existe un pivot que l'algorithme va utiliser'.

Par récurrence, A est vérifiée puisque l'on commence par la partition $\mathcal{P} = V$ et qu'à chaque étape on ne sépare que des sommets qui ne sont pas jumeaux.

Par ailleurs, l'existence du pivot a déjà été discutée; comme l'algorithme teste tous les sommets comme pivot. Il serait inutile de réutiliser un pivot usagé puisqu'il ne changerait rien. Donc s'il n'en reste plus, il n'y en n'a plus de possible. Contradiction.

Complexité Une étape d'affinage avec le pivot x coûte $|N[x]|$ éléments à déplacer en temps constant avec la structure de données proposée. Il y a $N = |V|$ étapes d'où une complexité en $O(n + m)$.

3.2 tri lexicographique de chaînes de caractères (en bref)

On veut trier un ensemble de N mots finis d'un alphabet Σ de k lettres par ordre lexivographique.

Le principe est celui du 'bucket sort' (tri par compartiment) : on sous-partitionne les classes en fonction de la i^{eme} lettre. Les lettres les plus fortes sont triées en premier, donc le tri par une lettre plus faible ne change pas les subdivisions déjà faîtes.

Cela revient à utiliser l'algorithme général avec

- Initialise : au i-ème appel, ajoute comme pivots l'ensemble des lettres qui, sont portées à la i-ème position, $G = a \in \Sigma \cup \epsilon : \exists mot | mot[i] = a$
- EnsemblePivot(s) = $S_{i,s} = \{mots | mot[i] = s\}$

On obtient une complexité en $O(m + k)$, où m est la longueur des chaines concaténées. (on pré-calcule les $S_{i,s} = \{mots | mot[i] = s\}$ dans un tableau en une lecture).

3.3 Partition Fonctionnelle Maximum

Problème : étant donnée un ensemble E, une fonction de $E \to E$, et une partition $\mathcal{Q}$ de E, on recherche la partition P telle que :

1. $\mathcal{P}$ est plus fine que $\mathcal{Q}$
2. $\mathcal{P}$ est stable par f : l'image de deux éléments d'une même classe se retrouve dans une même classe.
3. $\mathcal{P}$ ait le moins de classes possibles

C'est la 'partition fonctionnelle maximum', dont on admettra ici l'existence et l'unicité.

En partant de la partition $\mathcal{Q}$, on va affiner à chaque étape en utilisant $f^{-1}(X)$ comme pivot, où X est une classe de la partition courante. En effet, pour chaque classe X_i, les éléments de $f^{-1}(X_i)$ et de ${}^C f^{-1}(X_i)$ doivent être dans des classes différentes à la fin puisqu'ils sont envoyés par f respectivement dans X_i et hors de X_i.

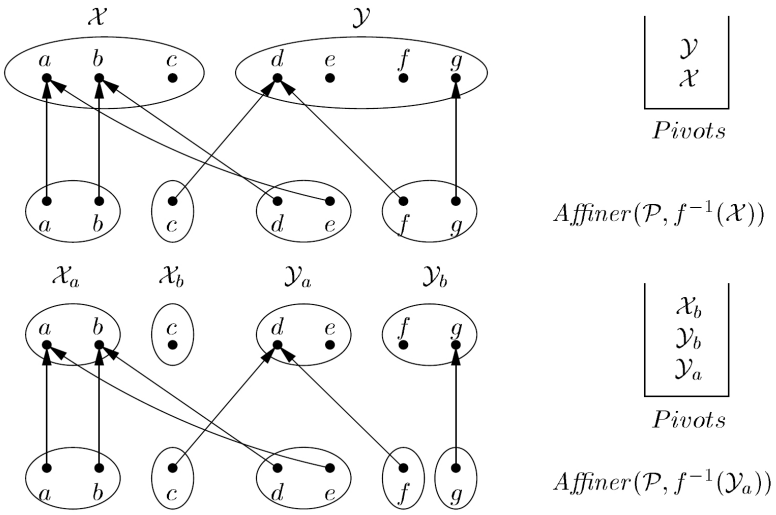


Figure 5: Deux étapes d'affinage sur un exemple (par $f^{-1}(X)$ puis $f^{-1}(\mathcal{Y}_a)$). Un exemple plus détaillé et l'algorithme seront donnés plus loin. Ici, $\{\{a,b\},\{c\},\{d,e\},\{f\},\{g\}\}$ est déjà la partition fonctionnelle maximum.

Definition: On dit qu'une partition $\mathcal{P}$ est stable par un ensemble pivot S ssi $\forall X \in \mathcal{P}, S \cap X = \emptyset ou X \backslash S = \emptyset$, c'est-à-dire si affiner $\mathcal{P}$ par S n'a aucun effet. En particulier, après que $\mathcal{P}$ a été affinée par un ensemble pivot S, $\mathcal{P}$ est désormais stable par S.

Proposition: Soit C un ensemble tel que $\mathcal{P}$ est 'stable' par C; alors, soit une classe $X \subset C$, affiner par $f^{-1}(X)$ a les mêmes conséquences que affiner par $f^{-1}(C \backslash X)$. Nous admettrons cette proposition.

Conséquence: Cette proposition nous permettra, lorsqu'on subdivise une partition X pour laquelle $\mathcal{P}$ était stable par $f^{-1}(X)$, de prendre comme nouveau pivot uniquement la plus petite des deux sous-classes de X (cf. détails ci-dessous), ce qui améliorera nettement la complexité du programme. C'est la **règle de Hopcroft** , ou **règle de la plus petite moitié**.

Principe de l'algorithme d'affinage: L'algotithme consistera à commencer avec les classes de la partition de départ pour liste de pivots ($GroupePivots$) . A chaque étape, soit X la première classe de $GroupePivots$, on affine la partition courante $\mathcal{P}$ par l'ensemble pivot $S = f^{-1}(X)$.

Cela va subdiviser plusieurs classes en deux, dont

- potentiellement des classes que nous avions en attente dans $GroupePivots$, et qui n'auront alors plus aucun sens.
- des classes $\mathcal{Y}$ qui n'étaient pas dans $GroupePivot$ (i.e. $\mathcal{P}$ était stable par $\mathcal{Y}$), mais $\mathcal{P}$ n'est plus forcément stable par les deux nouvelles sous-classes, et il faudra éventuellement les rajouter dans pivots.

En détails : plaçons nous après que $\mathcal{P}$ fût affiné selon l'ensemble pivot associé à une classe X (i.e. par l'ensemble pivot $f^{-1}(X)$). $\mathcal{P}$ est désormais stable par $f^{-1}(X)$, et on peut enlever X de la liste des pivots.

Soit $\mathcal{Y}$ une classe qui a été coupée durant cette étape d'affinage, en $\mathcal{Y}_a$ et $\mathcal{Y}_b$; deux cas se présentent :

1. $\mathcal{Y}$ n'est pas dans la liste des pivots, ce qui signifie que $\mathcal{P}$ est stable par $f^{-1}(\mathcal{Y})$, et on peut appliquer la règle de Hopcroft en n'ajoutant dans la liste des pivots uniquement la plus petite classe entre $\mathcal{Y}_a$ et $\mathcal{Y}_b$. [Remarque : ce cas concernera aussi X si elle a été subdivisée]

2. $\mathcal{Y}$ est dans la liste des pivots en attente. $\mathcal{Y}$ n'a plus de sens, il faut donc enlever $\mathcal{Y}$ de la liste des pivots et y rajouter à la fois $\mathcal{Y}_a$ et $\mathcal{Y}_b$. En pratique, avec la structure de donnée proposée dans la deuxième partie, $\mathcal{Y}$ aura été réduite à $\mathcal{Y}_b$ sur place, et il suffira d'ajouter $\mathcal{Y}_a$ dans la liste des pivots.

Algorithm 4 Partition Fonctionnelle Maximum par affinage de partition

Entrée : une fonction f sur un ensemble E ; une partition $\mathcal{Q}$ de E, qui n'est pas forcément stable par f.
Sortie : un affinage de $\mathcal{Q}$, stable par f et maximum.

```
P = Q
Pour chaque élément x ∈ E, pré-calculer f^{-1}(x)
GroupesPivots = { classes de Q }.
tant que ∃Y ∈ GroupesPivots faire
  enlever Y de GroupesPivots
  S = f^{-1}(Y) = (∪f^{-1}(p), p ∈ Y)
  {On affine maintenant P selon S}
  Pour tout X ∈ P, tel que ∅ ≠ (X ∩ S) ≠ S faire
    X_a = X ∩ S
    X = X\S
    Insérer X_a à droite de X dans P
    {Ajoute les nouveaux pivots}
    if X est dans GroupePivots, (sauf si X ⊂ Y) then
      ajouter X et X_a dans GroupesPivots.
    else
      ajouter la plus petite des classes entre X et X_a dans GroupesPivots.
    fin si.
  fin pour.
fin tant que.
return P
```

complexité: $O(|E| \log |E|)$

Démonstration L'étape limitante en complexité est le nombre d'affinages et le calcul de S.

Tout d'abord, l'union $S = f^{-1}(Y) = (S \cup f^{-1}(p), p \in Y)$ peut se calculer en temps $|Y|$. En effet, à partir des $f^{-1}(x), x \in E$ pré-calculés, comme $f^{-1}(\mathcal{Y}_a \cup$

$\mathcal{Y}_b) = f^{-1}(\mathcal{Y}_a) \cup f^{-1}(\mathcal{Y}_b)$, et qu'ils sont disjoints, il suffit de les mettre bout à bout en $O(|Y|)$ (avec une structure de liste doublement chaînée par exemple).

On a déjà vu que l'affinage de P par S se fait en temps $|S|$.

On remarque d'autre part que, durant l'exécution,

1. on traite en premier lieu *une fois* chaque classe ou toutes ses sous classes, tant qu'elle n'a pas été sortie de la liste de pivots.

2. Puis, à chaque fois qu'on reprend une sous-classe hors de la liste de pivots (cas [1] ci-dessus), on reprend moins de la moitié de sa taille (donc de son travail).

3. Si on attribue un 'étage' aux classes dans la liste de Pivots, valant 0 au début tant qu'elles n'ont pas été sorties de la liste de pivots, qui est conservé pour les sous-classes dans le cas [1], mais qui est incrémenté de 1 quand une classe est repêchée (cas [2]). Alors, la 'taille' concaténée des classes de l'étage $i+1$ sera au plus la moitié de la 'taille' de l'étage i.

Enfin, si on regarde l'ensemble des classes $\{Z_{i,j}\}$ ayant le même 'étage' i fixé,

- elles sont disjointes (car elles doivent provenir de deux classes du même étage précédent, donc distinctes par récurrence)

$$\sum_j |Z_{i,j}| < |E|$$

- et donc leurs images réciproques aussi :

$$\sum_j |f^{-1}(Z_{i,j})| < |E|$$

Pour un étage, donc, la complexité (amortie) sur le calcul de S et l'affinage, sera majoré par $|E|$.

Et, comme il y a au plus $\log_{[2]} |E| + 1$ étages, on obtient finalement une complexité en $O(|E| \log |E|)$.

Correction Elle peut se prouver de manière habituelle présentée au début, avec les invariants :

- A : "P est plus fine que Q"
- B : "Les éléments hors de la liste de pivot, laissent stable la partition en cours". Quand la liste de pivots est vide (à la fin), la partition est stable par f (sans quoi on pourrait trouver une classe qui affine strictement).

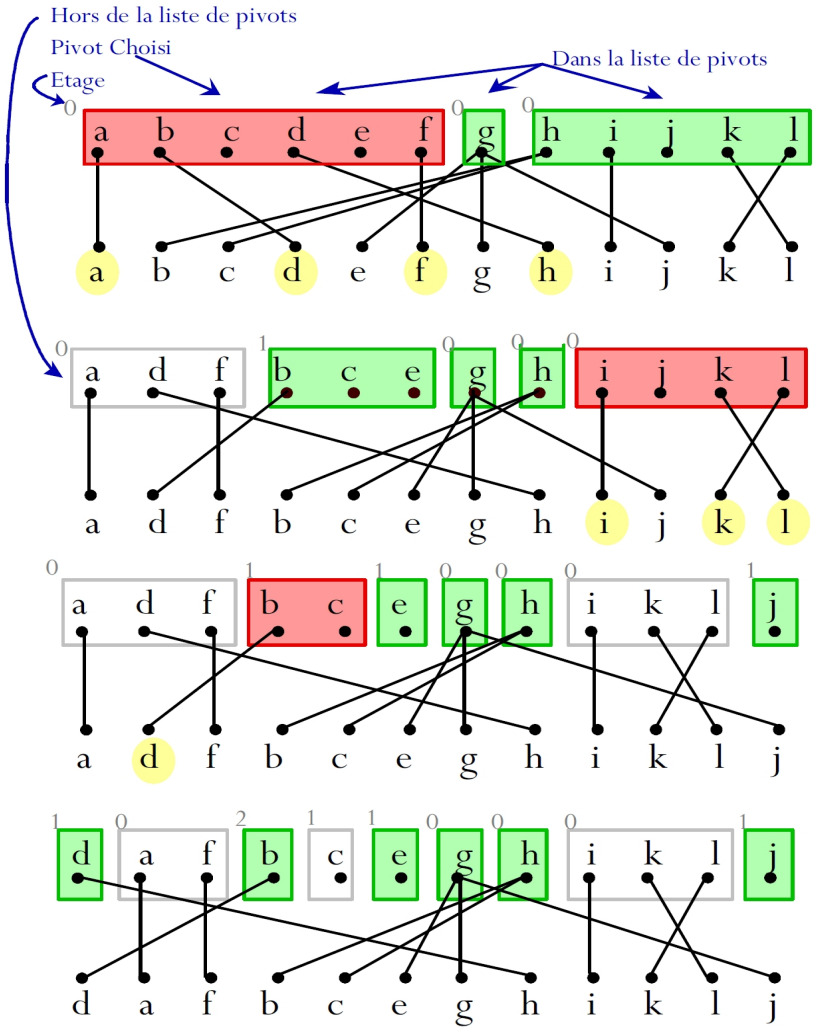


Figure 6: Exécution détaillée sur un example.Les classes dans la liste de pivots en attente sont coloriées (en rouge pour le pivot choisi et en vert pour les autres). L'"étage" est noté au-dessus de chaque classe. Au premier pas, à partir de $X = (abcdef)$, on calcule $S = (adfg)$, qui va séparer d'une part $(abcdef)$ en (adf) et (bce). Le lieu d'insertion n'a pas d'importance ici mais il a été fait à droite par conversion. Comme (bce) est reprise dans la liste de pivots, son étage est incrémenté. D'autre part, $(hijkl)$ est séparé en (h) et $(ijkl)$. Comme $(hijkl)$ était dans la liste de pivot (et n'était pas le pivot choisi), les deux sous-classes restent dans la liste de pivots, et leur étage n'est pas incrémenté. Etc. A la dernière itération, les dernières classes dans la liste ne changent plus la partition. On a bien atteint la partition fonctionnelle maximum.

Dans sa thèse, C.Paul transforme ce problème pour résoudre un problème de minimisation d'automates, par regroupement d'états qui reconnaissent le même langage. Il présente aussi le problème de partition relationnelle maximum, qui est une généralisation du présent problème où f est une relation d'équivalence.

Il doit être possible de ramener quelques problèmes de graphes à une partition fonctionnelle maximum. Par exemple, si on considère un terrain militaire où différentes bases sont reliées par des routes, le tout représenté par un graphe. On applique une fonction à chaque sommet (base), qui envoie les militaires dans une autre base au coup suivant. Si on veut étudier le trajet global ou des sommets de regroupement, on peut envisager de faire une partition fonctionnelle maximum et de relier chaque classe à la classe vers laquelle elle se dirige.

4 Application à certaines classes de graphes

4.1 Décomposition Modulaire

Définition Soit $G = (V, E)$ un graphe non orienté, un module M de G est un sous-ensemble de sommets tel que tous les sommets de M aient les mêmes voisins à l'extérieur de M.

Formellement,

$$M \text{ module de } G \Leftrightarrow M \subset V; \forall x, y \in M, N(x) \backslash M = N(y) \backslash M$$

Définition Un module non trivial est fort s'il est maximal pour l'inclusion.

Proposition Si M et M' sont deux modules d'intersection non vide, alors

1. $M \backslash M'$ et $M' \backslash M$ sont des modules
2. $M \cap M'$ est un module
3. $M \cup M'$ est un module

Corollaire Comme les sommets forment des modules triviaux, il y aura existence et unicité d'une partition en modules forts d'un graphe.

Contrairement à la partition d'un graphe en sommets jumeaux, il y a cette fois une entière liberté sur les arêtes à l'intérieur d'un module.

On va s'intéresser à un problème un tantinet plus général :

Problème : Soit $\mathcal{Q}$ une partition de V (dans $G = (V, E)$). Une partition modulaire pour $\mathcal{Q}$ est une partition P de V, qui soit plus fine que $\mathcal{Q}$, et ne contienne que des modules de G.

Vers un Affinage de Partition Pour résoudre ce problème, comme dans les exemples précédents, l'idée consiste à, tant qu'une partition $\mathcal{P}$ n'est pas correcte, trouver un élément qui va discriminer une ou plusieurs classes en deux sous-classes strictes. Un élément discriminera une classe si il est à la fois voisin et non voisin avec des membres de cette classe.

Plus formellement écrit,

$$\begin{aligned}
& \mathcal{P}\text{n'est pas une décomposition modulaire de}G \\
\Leftrightarrow\ & \exists X \in \mathcal{P}, \text{X n'est pas un module} \\
\Leftrightarrow\ & \exists X \in \mathcal{P}, \exists v \in V\backslash X, \exists x, y; \in X, v \in N(x)\text{mais } v \notin N(y) \\
\Leftrightarrow\ & \exists X \in \mathcal{P}, \exists v \in V\backslash X, \emptyset \neq (N(v) \cap X) \neq X \\
\Leftrightarrow\ & \exists X \in \mathcal{P}, \exists v \in V\backslash X, X\text{n'est pas stable par le pivot}N(v)
\end{aligned}$$

Un élément discriminateur sera donc un sommet et non une classe. Son ensemble pivot est son voisinnage (éventuellement privé de sa propre classe). On pourrait donc déjà écrire un algorithme basique d'affinage de partition, qui commence avec la liste des classes de $\mathcal{P}$. Pour toute classe dans la liste, la retirer,pour tout élément de cette classe, affiner P par l'ensemble pivot de cet élément, et remettre dans la liste les sous-classes crées :

Algorithm 5 Algorithme naïf de décomposition modulaire

Entrée : Un graphe $G = (V, E)$; une partition $\mathcal{Q}$ de V
Problème : Trouver la partition modulaire maximale de G, adaptée à Q.

$\mathcal{P} = \mathcal{Q}$
$GroupesPivots = \{X_1, ..., X_n\}$ {les classes de Q}
tant que $\exists Y \in GroupesPivots$ **faire**
  Retirer Y de $Groupespivots$
  Pour tout $p \in Y$ **faire**
    $S = N(s)\backslash Y$
    {On affine maintenant P selon S}
    Pour tout $X \in \mathcal{P}$, tel que $\emptyset \neq (X \cap S) \neq S$ **faire**
      $X_a = X \cap S$
      $X = X\backslash S$
      Insérer X_a à droite de X dans $\mathcal{P}$
      Placer X_a et X dans GroupesPivots
    fin pour.
  fin pour.
fin tant que.
return P

Il y a bien terminaison puisque l'on ne rajoute X_a et X dans GroupesPivots que s'il y a eu affinnage strict ($\emptyset \neq (X \cap S) \neq S$), ce qui ne peut se produire que n^2 fois au maximum. (en effet, on ne peut pas ajouter plus de n^2 contraintes différentes du type x et y ne sont pas dans la même classe.) On pourrait aussi dire qu'une classe n'est remplacée que par des classes de taille strictement inférieure.

C.Paul propose, comme c'était le cas dans le problème de partition fonctionnelle maximum, une manière de gérer intelligemment la liste des groupes pivots pour améliorer la complexité. L'algorithme et sa correction sont un peu obscures donc je n'en parle pas ici, mais le principe de base pour gérer la liste de pivots ressemble fortement à la règle de Hopcroft. Sa complexité est en $O(|V| + |E| \log |V|)$.

4.2 Reconnaissance des graphes triangulés et Lex-BFS

Dans cette partie, le but est de reconnaître si un graphe est triangulé. Après quelques rappels sur cette catégorie de graphes, et sur un algorithme qui résout ce problème (Lex-BFS), nous montrerons qu'il peut être vu comme un affinage de partition.

Définition Un graphe est dit triangulé ssi tout cycle de plus de 4 sommets possède une corde (i.e. une arête supplémentaire entre deux sommets du cycle).

Définition Un sommet d'un graphe est dit simplicial si son voisinnage est une clique.

Définition Soient a et b deux sommets d'un graphe $G = (V, E)$, un $a - b$ séparateur dans G est un ensemble S de sommets tels que $G \backslash S$ sépare a et b dans deux composantes connexes différentes.

Définition Un ordre d'élimination simplicial est une suite de sommets $\{x_1, ..., x_n\}$ tels que, $\forall i \in [|1..n|], x_i$ est simplicial dans le graphe induit par $\{x_{i+1}, ..., x_n\}$.

Dans de nombreuses situations ou domaines, on est amené à travailler sur des graphes qui sont intrinsèquement triangulés. Par ailleurs, ils offrent des propriétés puissantes, on peut citer le fait que le nombre chromatique d'un graphe triangulé est la taille de sa clique maximale, qui peut se trouver polynomialement dans leur cas. Un théorème important de caractérisation des graphes triangulés est :

Théorème : Soit G un graphe connexe non orienté, les propositions suivantes sont équivalentes :

1. G est triangulé
2. Il existe un ordre d'élimination simplicial pour G (qui se termine par le sommet de son choix).
3. tout séparateur minimal (pour l'inclusion), est une clique.

Démonstration Nous rappelons la preuve car elle servira aussi à démontrer la correction de l'algorithme Lex-BFS :

$1 \rightarrow 3$ Soient a, b deux sommets, S un $a - b$ séparateur minimal, et A et B les composantes connexes respectives de a et b. Soient deux sommets s_1 et s_2 de S. Soit C le cycle formé par le plus court chemin entre s_1 et s_2, dans A et B respectivement. Si s_1 et s_2 ne sont pas connectés, ce cycle n'a pas de corde (puisque A et B sont déconnectées et que le cycle est formé dans A et B uniquement). Contradiction. Donc S est une clique.

$3 \rightarrow 1$ Soit un cycle de longueur supérieure à 4, soient a et b deux sommets non voisins sur ce cycle. Soit un $a - b$ séparateur S. On décompose le cycle en deux chemins disjoints entre a et b. S contient forcément un sommet sur ces deux chemins. Par hypothèse, S est une clique, donc le cycle admet une corde.

Lemme intermédiaire : Soit G un graphe triangulé, soit c'est une clique, soit il a deux sommets simpliciaux non adjacents.

Démonstration : Par récurrence sur la taille des graphes. Le lemme est vrai pour le cas de base à 3 sommets. Supposons le lemme vrai pour tout graphe à moins de n sommets. Soit $G = (V, E)$ triangulé à $n+1$ sommets, Si G n'est pas complet, prenons a et b non voisins et S un séparateur minimal. G est triangulé donc S est une clique. Prenons A et B les composantes connexes de $G\backslash S$, contenant respectivement a et b. On peut trouver un sommet simplicial dans A. En effet, par hypothèse de récurrence, appliquée à $A \cup S$ (que l'on suppose non complet sinon a est simplicial dans G), comme un sous-graphe d'un graphe triangulé ést lui-même triangulé, il y a deux sommets simpliciaux non voisins dans $A \cup S$, dont un qui est forcément dans A puisque S est une clique. Ce sommet sera encore simplicial dans G puisque son voisinnage est dans $A \cup S$. De même, il existe un sommet simplicial dans B et il n'est pas voisin avec celui que l'on a trouvé dans A.

Suite de la démonstration

$1 \rightarrow 2$ Par récurrence, soit G un graphe triangulé, soit $x \in V$. Si $|V| = 3$, on peut construire à la main un ordre d'élimination se terminant par x (3 cas). Si $|V| = n + 1$, d'après le lemme intermédiaire, on peut trouver un sommet simplicial s_1 distinct de x. $G\backslash\{x\}$ est encore triangulé, donc, par hupothèse de récurrence, il existe un ordre d'élimination simplicial qui se termine par x dans $G\backslash\{x\}$. Prolongé par s_1 en première position, c'est un ordre d'élimination simplicial sur G.

$2 \rightarrow 1$ Soit un graphe possédant un ordre d'élimination simplicial. Soit C un cycle d'au moins 4 sommets. Soit s le sommet du cycle de rang minimum dans l'ordre d'élimination simplicial. Alors, ses deux voisins dans le cycle ont un rang supérieur, sont donc dans une clique, et sont donc voisins, ce qui forme une corde.

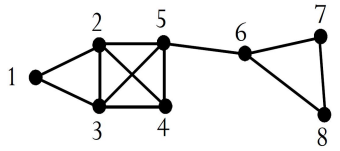


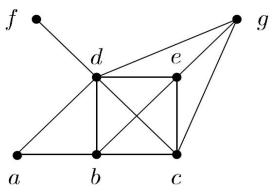
Figure 7: Exemple d'ordre simplicial sur un graphe triangulé. On peut vérifier que les voisinnages successifs sont bien des cliques.

Algorithme de Reconnaissance L'algorithme suivant, le 'Lexical Breadth First Search' (Lex-BFS), est capable, si il existe, de trouver un ordre de décomposition simplitial sur un graphe. Il construit cet ordre en sens inverse, en commençant par un sommet donné.

Algorithm 6 Lex-BFS

Entrée : Un graphe $G = (V, E)$;
Problème : Une suite σ de sommets, simpliciale si elle existe

Pour tout Sommet $x \in V$ **faire**
  $Marque(x) \leftarrow \emptyset$
fin pour.
for $i = n$ à 1 **faire**
  Choisir un sommet non numéroté de marque maximum pour l'ordre lexicographique
  $\sigma(i) \leftarrow x$
  Pour tout voisin non numéroté y de x **faire**
    marque(y) $\leftarrow$ marque(y) $\cup\{i\}$
  fin pour.
fin pour.
return σ



σ	7	6	5	4	3	2	1
a	$\emptyset$	7	7,6	7,6	• 7,6		
b	$\emptyset$	• 7					
c	$\emptyset$	7	• 7,6				
d	• $\emptyset$						
e	$\emptyset$	7	7,6	• 7,6,5			
f	$\emptyset$	7	7	7	7	7	• 7
g	$\emptyset$	7	7	7,5	7,5,4	• 7,5,4	

Figure 8: Exemple d'exécution de Lex-BFS. Le sommet choisi entre l'indice n et 1 est marqué par un rond noir. Il faut lire le résultat de Lex-BFS en sens inverse (i.e. de 1 à n), pour obtenir un ordre simplicial.

On peut aussi interpréter Lex-BFS comme un algorithme d'affinage de partition.

en partant de la partition grossière $P = E$, l'ordre lexicographique est maintenu en subdivisant les classes par des étiquettes de plus en plus faibles. A chaque pas, le sommet x est pris dans la dernière classe qui ait un sommet non visité, c'est-à-dire la classe d'étiquette la plus grande (au sens lexicographique). Ensuite, la partition sera affinée avec pour ensemble pivot le voisinnage de x avec une étiquette plus faible, donc qui ne dérangera pas l'ordre lexicographique entre les classes déjà existantes, mais qui créera des sous-classes qui vérifieront encore l'ordre lexicographique. La figure 8 illustre le propos.

Algorithm 7 Lex-BFS vu comme affinage de partition

Entrée : Un graphe $G = (V, E)$;
Problème : Une suite σ de sommets, simpliciale si elle existe

$\mathcal{P} \leftarrow V$
$\forall x \in V, \pi(x) \leftarrow 0$ {Sommet non visité}
$i \leftarrow n$
tant que $\mathcal{P}$ contient une classe non-singleton, **faire**
  Prendre X_k la dernière classe qui ait encore des éléments non visités
  enlever un sommet x de X
  $\pi(x) \leftarrow i$
  $i \leftarrow i - 1$
  $S = N(x)$
  Pour tout $X_i, i \leq k$ **faire**
    if $\emptyset \neq (X_i \cap S) \neq X_i$ **then**
      $X_i \leftarrow (X_i \cap S)$
      Insérer $(X_i \backslash S)$ à gauche de X_i
    fin si.
  fin pour.
fin tant que.
return σ

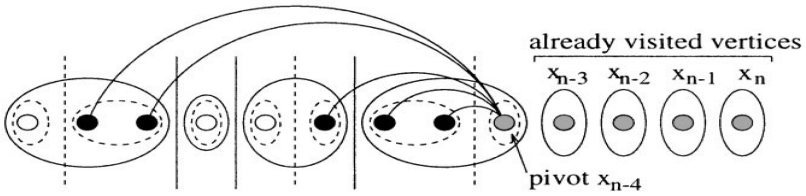


Figure 9: Organisation des classes : un sommet tiré subdivisera les classes au niveau de ses voisins, et selon une étiquette plus petite que celles qui distinguent déjà les classes, de sorte que l'affinage maintient l'ordre lexicographique entre les classes.

4.3 Prolongements

Avec l'algorithme Lex-BFS, on peut déjà détecter si un graphe est triangulé, en récupérant l'ordre donné par Lex-BFS, et en vérifiant s'il est simplicial ou non. Si oui, on peut conclure que le graphe est triangulé.

Dans un article de M.Habib, quelques autres utilisations de Lex-BFS sont proposé, en particulier l'orientation transitive d'un graphe, et la reconnaissance des graphes d'intervalles, ce qui n'est pas un problème aisé. Pour information, voici quelques définitions associées.

Définition Un graphe d'intervalles est un graphe non orienté pour lequel il existe une attribution d'un intervalle de $\mathcal{R}$ à chaque sommet tel que deux sommets sont connectés ssi leurs intervalles s'intersectent (sur un ouvert).

Définition Un graphe simple orienté $G = (V, E)$ est dit transitif ssi $\forall a, b, c \in V$, $(ab) \in E$ et $(bc) \in E \Rightarrow (ac) \in E$.

Définition Un graphe est dit de comparibilité ssi il est transitif et acyclique.

Définition Un graphe est dit de co-comparabilité ssi son complémentaire est un graphe de comparabilité.

propriété Un graphe est un graphe d'intervalles si et seulement si il est triangulé et son complémentaire est de comparabilité.

5 Conclusion et Bibliographie

Cette synthèse est basée sur 3 principaux documents :

1. Thèse de Christophe Paul : Parcours en largeur lexicographique, un algorithme de partitionnement, application aux graphes et généralisation. 1998. En particulier, chapitres 1, 4 à 8.

2. M.Habib et al, *Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing*, Theoretical Computeer Science 234 (2000) 59-84

3. M.C.Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Annals of discrete Mathematics, Elsevier (2004)

Suivant la trame de la thèse de C.Paul, le but était d'introduire la notion d'affinage de partition, et de montrer qu'elle peut être utilisée dans des problèmes variés. Les applications aux graphes et les preuves associées deviennent très vite compliquées et je n'ai pas trop insisté sur ces exemples. De plus, la thèse et l'article sur ces thèmes sont difficiles à aborder.

Un avantage que j'ai trouvé à l'affinage de partition, est de raisonner par contraintes et donc, de ne faire que des choix nécessaires, ce qui maintient en général une propriété de maximalité, chose qui est plus dûre à prouver quand on suis un algorithme de type fusion.