Ordonnancements linéaires et décompositions de graphes

Bellitto Thomas

Vendredi 7 septembre 2012

Rapport de stage de 1ère année du Magistère informatique et
télécommunications, ENS Cachan/Bretagne et Université Rennes 1
Stage effectué du 21 mai au 20 juillet 2012 à l'INRIA au sein de l'équipe de recherche Mascotte
(INRIA, I3S, CNRS, UNS)

Encadrants :
David Coudert, chargé de recherche INRIA
Nicolas Nisse, chargé de recherche INRIA

Résumé

Nous étudions dans ce rapport une décomposition de graphe, la path decomposition, que nous présenterons au travers d'une application pour les télécommunications. Ce rapport propose essentiellement une méthode dans le cas des graphes orientés pour accélérer certains algorithmes de calculs de décompositions optimales, en contractant des sommets du graphe et ainsi faire le calcul en un temps qui sera fonction du nombre de sommets d'un graphe plus petit.

Mots-clés : algorithmique ; théorie des graphes ; optimisation combinatoire ; contraction ; vertex separation ; problème de reroutage

Introduction

Un problème fréquent dans l'optimisation des télécommunications (voir [2], [6], [7], [8]) est celui du reroutage des connexions des utilisateurs. Les réseaux optiques WDM (pour Wavelength Division Multiplexing) ont en fait une capacité limitée (on ne peut faire passer qu'un nombre donné de longueurs d'onde différentes dans nos réseaux) et les chemins optiques empruntés par les requêtes de connexion qui utilisent la même longueur d'onde doivent être disjoints pour éviter les interférences.

On représente nos réseaux par des graphes orientés et les requêtes par des chemins sur ces graphes. Un routage est en fait un ensemble de chemins suivis par les requêtes. Pour simplifier, considérons dans un premier temps des fibres optiques dans lesquelles ne peuvent passer qu'une longueur d'onde. Pour qu'un routage soit valide, deux requêtes ne peuvent donc pas emprunter le même arc.

Sur nos schémas, nous représenterons le réseau en noir et les requêtes en bleu.

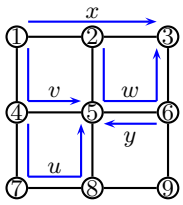


FIGURE 1 – Cette configuration est valide. Les arcs $(5,6)$ et $(6,5)$ ne sont pas les mêmes et il peut très bien y avoir une requête sur chaque.

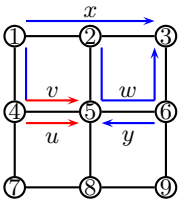


FIGURE 2 – Cette configuration n'est pas valide, les requêtes u et v empruntent toutes les deux l'arcs $(4,5)$ et interféreront.

Supposons que notre réseau soit dans la configuration R montrée en figure 3 et que l'arc (8,5) devienne indisponible (pour une panne, une maintenance, pour être utilisé par une nouvelle requête...les causes possibles sont nombreuses). La requête u ne pourrait alors plus passer par (8,5), mais à cause des autres requêtes, elle ne peut pas emprunter non plus les autres chemins de 4 à 5. Pour atteindre une nouvelle configuration valide (par exemple, la configuration R' de la figure 4), il faudra alors changer le routage de plusieurs requêtes.

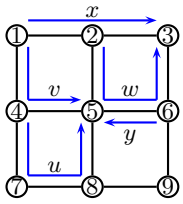


FIGURE 3 – le routage R

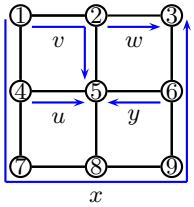


FIGURE 4 – le routage R'

Pour atteindre R' à partir de R, une solution serait d'interrompre toutes les requêtes et de les rétablir une par une dans la nouvelle configuration, mais c'est ce que nous allons essayer d'éviter. Afin de maintenir le trafic sur le réseau aussi haut que possible, notre objectif sera de minimiser le nombre de connexions coupées simultanément. On peut par exemple procéder de la façon suivante :

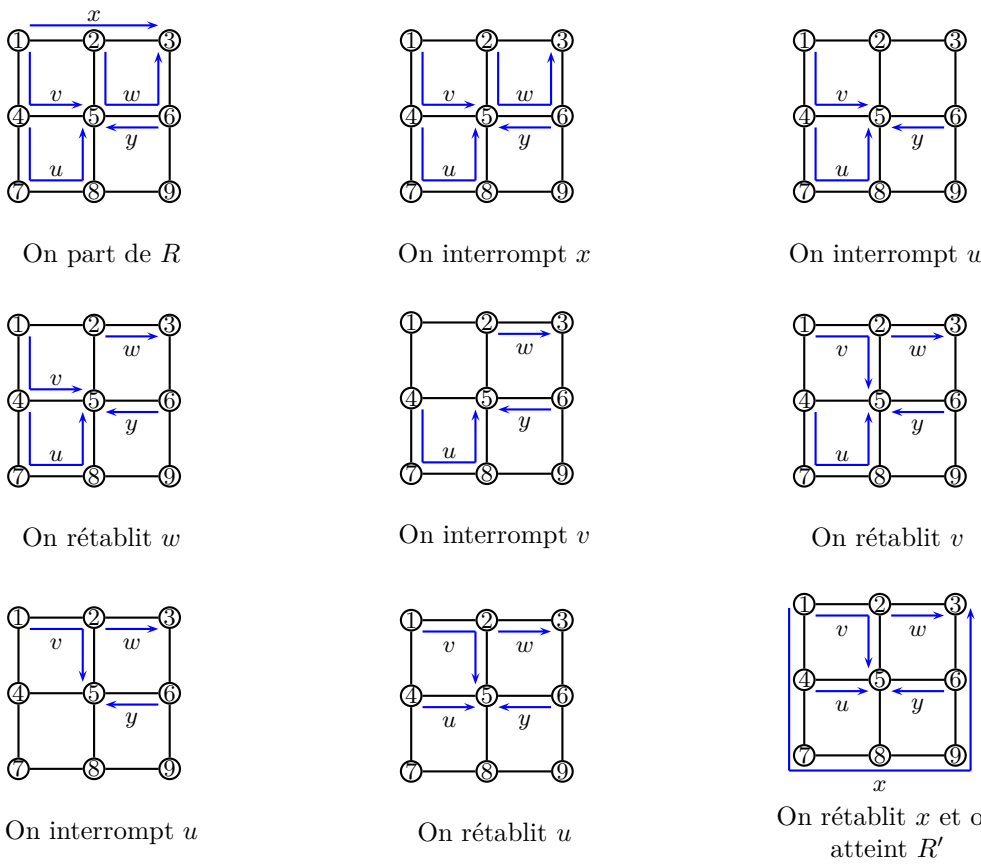


FIGURE 5 – En procédant ainsi, on ne coupe à aucun moment plus de deux requêtes à la fois.

Notre objectif dans ce rapport sera de mettre au point un algorithme efficace qui, étant donné un routage R et un routage cible R', déterminera une stratégie optimale (pour le critère donné précédemment, à savoir le nombre maximum de connexions coupées simultanément). Nous ne tiendrons pas compte dans ce rapport des solutions qui font passer une requête par des routes intermédiaires et qui nécessitent donc de couper la même connexion deux fois, une requête pourra être soit dans le routage initial, soit dans le routage cible. Nous ne nous intéresserons pas non plus à comment obtenir le routage R', bien que ce soit également un problème intéressant et que la qualité de la stratégie finale en dépendra beaucoup.

La modélisation que nous proposerons nous ramènera à un problème, classique lui aussi, d'algorithmique des graphes : le calcul de *path decomposition*. Il existe d'autres modèles du problème de reroutage, notamment dans le cas où on peut établir le nouveau chemin de la requête avant de couper l'ancien de telle sorte que le reroutage sera opéré sans ne jamais couper la connexion de l'utilisateur (cf [7], [8]). Les solutions optimales ne sont alors pas forcément les mêmes et font apparaître une autre grandeur : le *process number*. Toutefois, si le routage initial et final ont des arcs en commun (fréquent dans la pratique, les réseaux rencontrés étant beaucoup plus grands que celui de notre exemple) et la même longueur d'onde, ils ne pourront pas être établis en même temps, même si les requêtes des autres utilisateurs n'empêchent pas le nouveau routage de s'établir.

Nous étudierons donc la *path decomposition*, une notion introduite en 1983 par Robertson et Seymour (cf [12]) et généralisée aux graphes orientés par Barát (cf [3]), ainsi que la *vertex separation*, une notion assez liée. Il existe de nombreux algorithmes pour calculer rapidement la *vertex separation* de certains types de graphes, de manière exacte ou approchée ([4], [9], [13]). Toutefois, le calcul de *vertex separation* d'un graphe quelconque est NP-difficile et APX-difficile [1](cf [11]) et le meilleur algorithme exact de calcul de *path decomposition* (asymptotiquement) à nos jours est en $O^*(2^n)$ (voir [5]). La taille des instances à résoudre peut alors rapidement rendre les algorithmes exacts inutilisables et nous forcer à utiliser des algorithmes polynomiaux qui donneront des solutions dont l'écart à la solution optimale ne sera pas majorable.

1. Il est NP-difficile de trouver une approximation de la *vertex separation* à un facteur multiplicatif k près.

Dans ce rapport, nous proposerons une méthode pour accélérer les algorithmes : on identifie en temps polynomial des sous-graphes qu'il sera possible de contracter et l'algorithme pourra ensuite traiter ces sous-graphes comme s'il ne s'agissait que d'un seul sommet. La méthode exposée fonctionne avec de nombreux algorithmes, dont l'algorithme en $O^*(2^n)$.

Dans la première partie, nous présenterons les problèmes de graphes sous-jacents à notre problème de reroutage, nous donnerons les définitions et les prérequis nécessaires pour la suite et ferons un bref état de l'art.

Les deux parties suivantes présenteront la contribution apportée lors de ce stage : nous définirons dans la partie 2 la notion de contraction de graphe et présenterons les résultats théoriques qui assurent la validité de l'algorithme, et nous décrirons l'algorithme lui-même dans la partie 3.

Enfin, la partie 4 proposera quelques pistes de recherche pour approfondir le problème, et donnera un aperçu de quelques obstacles à surmonter pour s'engager dans ces voies.

1 Prérequis et état de l'art

1.1 Quelques définitions

Notation 1

- On définit un graphe orienté par $D = (V, A)$ où V est l'ensemble de ses sommets et $A \subseteq V \times V$ celui de ses arcs.
- On note $N_D^+(v) = \{w \in V/(v, w) \in A\}$ le voisinage sortant du sommet v dans le graphe D. On note également $N_D^-(v) = \{w \in V/(w, v) \in A\}$ le voisinage entrant de v dans D. S'il n'y a aucune ambiguïté sur le graphe dans lequel on se situe, on peut noter simplement $N^+(v)$ ou $N^-(v)$.
- Un sommet est voisin entrant ou sortant d'un sous-graphe ssi il est le voisin entrant ou sortant d'un sommet de ce sous-graphe et n'appartient pas à ce sous-graphe.
- Les degrés entrants et sortants d'un sommet ou d'un sous-graphe sont respectivement les nombres de voisins entrants et sortants de ce sommet ou sous-graphe.
- On utilisera l'abréviation DAG (pour directed acyclic graph) pour parler d'un graphe qui ne contient pas de cycles orientés. On dira que les sommets sont listés dans un ordre d'effeuillage ssi chaque sommet arrive après tous ses voisins sortants (le graphe est acyclique donc il y en a un qui n'a pas de voisins sortants, le graphe privé de ce sommet reste acyclique, et on peut répéter le raisonnement...).
- On dira qu'une fonction f est en $O^*(g)$ ssi $\dfrac{f}{g}$ est majorable par une fonction polynomiale. Par exemple $n2^n = O^*(2^n)$.
- Un graphe est dit fortement connexe ssi pour tout sommet v et w, il existe un chemin menant de v à w et un chemin menant de w à v.
- Les classes de V quotienté par la relation $u \equiv v$ ssi il existe un chemin de u à v et un chemin de v à u sont appelées les composantes fortement connexes du graphe.
- On appelle transposé d'un graphe D le graphe obtenu en inversant tous les arcs de D.

Définition 1 : Path decomposition - pathwidth d'un graphe orienté[2] (cf [3]) :
Soit $D = (V, A)$. On appelle *path decomposition* de D toute suite finie $(X_i)_{1 \leqslant i \leqslant p}$ de sous-ensembles de V telle que

- $\bigcup_{1 \leqslant i \leqslant p} X_i = V$
- $\forall (i, j) \in [\![1, p]\!]^2, \forall k \in [\![i, j]\!], (X_i \cap X_j) \subset X_k$
- $\forall (v, w) \in A, \exists (i, j) \in [\![1, n]\!]^2$ tels que $i \leqslant j, w \in X_i$, et $v \in X_j$

On appelle alors largeur de la décomposition la grandeur $\max_{1 \leqslant i \leqslant p} \text{Card}(X_i) - 1$

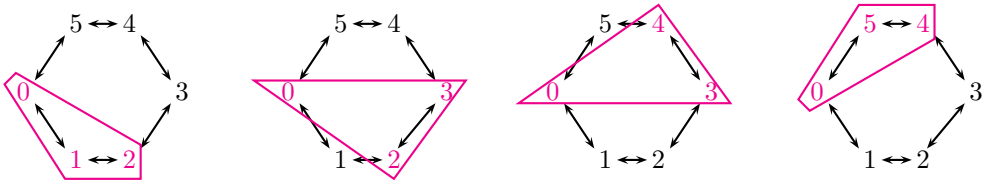
La *pathwidth* d'un graphe D, notée $\text{pw}(D)$ est la largeur de sa *path decomposition* la moins large.

2. Les notions exposées dans cette partie sont toutes transposables au cas non orienté. Il suffit de voir chaque arête (u, v) d'un graphe non orienté comme un arc (u, v) et un arc (v, u).

Exemple 1

est une décomposition optimale et la *pathwidth* du graphe vaut 1.

est une décomposition optimale et la *pathwidth* du graphe vaut 0.



est une décomposition optimale et la *pathwidth* du graphe vaut 2.

Pour tout graphe, $X = [V]$ est une décomposition valide. Pour une clique, c'est la seule et elle est donc optimale. La *pathwidth* d'une clique de taille n vaut $n - 1$.

La notion de *path decomposition* est étudiée depuis longtemps et permet de mieux comprendre la structure combinatoire des graphes, par exemple pour décomposer certains problèmes et mettre en oeuvre des algorithmes de programmation dynamique (cf [1]). Les recherches de partie stables ou de cliques maximales dans un graphe D à n sommets peuvent par exemple être résolues en $O(2^{\mathrm{pw}(D)}n)$ (en temps). La *path decomposition* est aussi utile pour effectuer des balayages de graphes et donne par exemple une stratégie dans un jeu de Cops&Robber où le fugitif est invisible, de vitesse arbitraire et se déplace dans le sens opposé des arcs du graphe.

1.2 Retour au problème de reroutage

On considère deux routages R et R' et on cherche à passer de l'un à l'autre. La première chose que nous ferons (cf [8]) sera de construire le graphe de dépendance associé au problème : nous représenterons chaque requête à rerouter par un sommet et tracerons un arc (u, v) ssi v, dans sa configuration initiale, empêche de rerouter u. Pour le routage R de la figure 6 et le routage R' de la figure 7, nous aurons donc le graphe D de la figure 8.

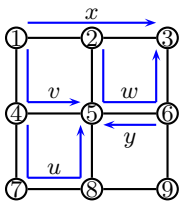


FIGURE 6 – le routage R

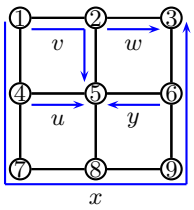


FIGURE 7 – le routage R'

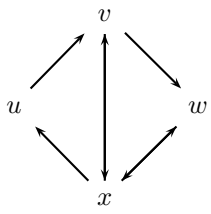


FIGURE 8 – le graphe de dépendance D

On représentera une stratégie de reroutage par la suite des ensembles de requêtes qui doivent être coupées à chaque étape. Par exemple, la stratégie qui consiste à couper tous les signaux pour les remettre dans la configuration finale sera représentée par $[\{u, v, w, x, y\}]$. La stratégie décrite en introduction sera représentée par $[\{x, w\}, \{x, v\}, \{x, u\}, \{x\}]$.

Ayant fait l'hypothèse que nous ne ferons pas passer nos requêtes par des chemins intermédiaires, on peut rajouter sans perte de généralités la contrainte suivante : un signal coupé ne peut être remis dans son routage initial. En effet, si pendant qu'un signal v était coupé, on n'a pas fait d'action qui nécessitait que v soit coupé, il était inutile de le couper, et si on en a profité pour rerouter une requête u que v empêchait de rerouter, c'est que le chemin initial de v était incompatible avec le chemin final de u, et

on ne peut donc pas remettre v dans son chemin initial. Une fois cette contrainte rajoutée, on remarque que la première apparition d'une requête dans la stratégie correspond à l'étape où on la coupe et que la première étape où elle n'apparaît plus correspond à l'étape où elle est reroutée.

THÉORÈME 1
Les stratégies valides sont les path decomposition de D.

Démonstration. Les stratégies valides sont celles qui satisfont les conditions suivantes :
- chaque requête qui change de route (celles qui sont représentées sur le graphe de dépendance) doit apparaître ($\Leftrightarrow \bigcup X_i = V$).
- si le routage final de u est incompatible avec le routage initial de v ($\Leftrightarrow (u, v)$ est un arc de D), alors u ne peut être traitée avant que v ne soit coupée. Autrement dit, v doit déjà être apparue lors de la dernière occurrence de u. ($(v, w) \in A \Rightarrow \exists (i, j) \in [\![1, n]\!]^2$ tels que $i \leqslant j, w \in X_i$, et $v \in X_j$)

La condition $i \leqslant k \leqslant j \Rightarrow X_i \cap X_j \subset X_k$ est équivalente à la contrainte précédente : une requête coupée doit être remise dans le routage final (et ne sera donc plus coupée après). □

On remarque également que nos stratégies optimales seront bien données par les décompositions optimales. [3]

Remarque 1
Pour simplifier, on avait supposé dans l'exemple qu'il n'y avait qu'une seule longueur d'onde qui circulait dans la fibre optique. Dans la pratique, deux requêtes ne peuvent interférer et ne seront donc reliés dans le graphe de dépendance que si elles ont la même longueur d'onde. Une fois le graphe de dépendance mis au point, la façon de le traiter ne change pas.

1.3 Recherche de path decomposition optimale

Définition 2 : Ordre linéaire sur un graphe :
Un ordre linéaire sur un graphe est une permutation des sommets de ce graphe. Dans la suite, nous noterons $\sigma(E)$ l'ensemble des permutations d'un ensemble E.

Notation 2

Soit L un ordre linéaire sur un graphe. On notera $L(i)$ le $i^{\text{ème}}$ élément de l'ordre et $S_L(i) = \bigcup_{j=1}^{i} L(j)$.

Définition 3 : Mesure sur un ordre :
Soit $D = (V, A)$ un graphe de n sommets, L un ordre sur D et f une fonction de $\mathscr{P}(V)$ dans $\mathbb{N}$. La mesure de l'ordre vaudra alors $\max_{1 \leqslant i \leqslant n} f(S_L(i))$. On appellera mesure locale à l'étape i la grandeur $f(S_L(i))$.

Définition 4 : Vertex separation
Soit D un graphe orienté et L un ordre sur D. On note $\text{vs}_L(D)$ la mesure de l'ordre L donnée par la fonction $S \mapsto \text{Card}(N^+(S) \setminus S)$. Autrement dit :

$$\text{vs}_L(D) = \max_i \text{Card}(N^+(S_L(i)) \setminus S_L(i))$$

La *vertex separation* d'un graphe D, notée $\text{vs}(D)$ est le minimum de $\text{vs}_L(D)$ pour les ordres L sur D.

Proposition 1 [14]
Soit $D = (V, A)$ un graphe orienté à n sommets et L un ordre sur D. On pose pour tout i dans $[\![1, n]\!], X_i = \{L(i)\} \cup (N^+(S_L(i)) \setminus S_L(i))$. $X_1, \cdots, X_n$ est alors une *path decomposition* de D.

Démonstration.
- Comme $L(i) \in X_i$, on a bien $\bigcup_i X_i = V$.

3. Un raisonnement analogue permet de faire apparaître la notion de *path decomposition* dans beaucoup de problèmes où on trouve des graphes de dépendance, comme par exemple le processus de compilation

- Soit i, j, k tels que $1 \leqslant i < k < j \leqslant n$. Soit $v \in X_i \cap X_j$. Comme $v \in X_j$, le rang de v dans L est au moins j donc $v \notin S_L(k)$. Comme $v \in X_i$, on sait que $v \in N^+(S_L(i)) \subset N^+(S_L(k))$. On en déduit que $v \in X_k$.
- Soit $(v, w) \in A$. Soit i le rang de w dans L et j le rang de v.
  - si $i < j$, on a bien $i \leqslant j, w \in X_i$ et $v \in X_j$.
  - sinon, on a $w \in N^+(S_L(j)) \subset X_j$, d'où $v \in X_j$ et $w \in X_j$. □

La largeur de cette décomposition est $\max_i \operatorname{Card}(N^+(S_L(i)) \setminus S_L(i))$.

$\min_{L \in \sigma(V)} \max_i \operatorname{Card}(N^+(S_L(i)) \setminus S_L(i))$ nous fournit un majorant de la *pathwidth* et on peut prouver que ce majorant est atteint.

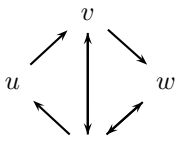
Théorème 2 : Kinnersley ([10])
Pour tout graphe orienté D,

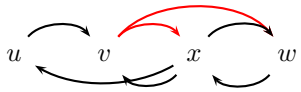
$$\operatorname{pw}(D) = \operatorname{vs}(D) = \min_{L \in \sigma(V)} \max_{1 \leqslant i \leqslant n} \operatorname{Card}(N_D^+(S_L(i)) \setminus S_L(i))$$

On cherchera alors un ordre qui minimise notre mesure, et avec la technique utilisée dans la démonstration de la propriété 1, cet ordre nous fournira une décomposition.

Exemple 2

Considérons le graphe D à droite et l'ordre $[u, v, x, w]$





On voit que $N^+\{u, v\} \setminus \{u, v\} = \{w, x\}$ et $\operatorname{vs}_{[u,v,x,w]} = 2$. On déduira la stratégie $[\{u, v\}, \{v, w, x\}, \{x, w\}, \{w\}]$ qui coupera simultanément 3 signaux.

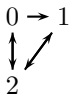
La stratégie optimale donnée $[\{w, x\}, \{x, v\}, \{x, u\}, \{x\}]$ donnée dans l'introduction découle de l'ordre $[w, v, u, x]$. On a bien $\operatorname{vs}_{[w,v,u,x]}(D) = 1$.

Il y a ainsi $n!$ ordres à comparer et on en déduit immédiatement un algorithme en $O^*(n!)$ temporel. La programmation dynamique nous permet de passer en $O^*(2^n)$: on étend facilement notre mesure aux ordres l sur des sous-ensembles $\mathcal{V}$ de V (la mesure vaut alors $\max_{1 \leqslant i \leqslant \operatorname{Card} \mathcal{V}} (N_D^+(S_l(i)) \setminus S_l(i))$).

On remarque ensuite que la mesure de $l = x_1 \cdots x_i$ est le maximum entre la mesure locale à l'insertion du dernier élément ($\operatorname{Card}(N^+(\mathcal{V}) \setminus \mathcal{V})$, qui ne dépend pas de l) et la mesure de l'ordre privé du dernier élément : $l' = x_1 \cdots x_{i-1}$. On obtient donc un ordre optimal sur $\mathcal{V}$ en cherchant un ordre optimal sur les sous-ensembles de $\mathcal{V}$ de cardinal $(\operatorname{Card} \mathcal{V} - 1)$ et en rajoutant à la fin le sommet enlevé.

Exemple 3

Recherchons un ordre optimal sur le graphe suivant :



v	Card$(N_D^+(v) \setminus v)$	coût de l	choix de l
[]	0	0	[]
[0]	2	2	[0]
[1]	1	1	[1]
[2]	2	2	[2]
$[0,1]/[1,0]$	1/1	2/1	$[1,0]$
$[0,2]/[2,0]$	1/1	2/2	[0,2] arbitrairement
$[1,2]/[2,1]$	1/1	1/2	$[1,2]$
$[1,0,2]$ $[0,2,1]$ $[1,2,0]$	0	1 2 1	$[1,0,2]$ arbitrairement

et $[1,0,2]$ est donc un ordre optimal.

Remarque 2
Le principal problème de la programmation dynamique est qu'elle est aussi en $O^*(2^n)$ en complexité spatiale, et c'est de là que viennent les premières limites de l'algorithme (on ne peut pas dépasser les graphes d'une trentaine de sommets). Il existe (cf [5]) une méthode proche qui a l'avantage d'être polynomiale en espace, mais elle est en $O^*(4^n)$ en temps et devient inexploitable encore plus tôt.

Cette méthode qui calcule en même temps une *path decomposition* optimale (où en tout cas, un ordre duquel on pourra déduire une *path decomposition* en temps linéaire) est actuellement la méthode la plus rapide (du moins asymptotiquement) de calcul de *pathwidth/vertex separation.*

Enfin, on peut encore accélérer le calcul avec les résultats suivants :

Théorème 3

- *La vertex separation d'un graphe est le maximum de la vertex separation sur ses composantes fortement connexes*[4].
- *En remarquant que les composantes fortement connexes d'un graphe forment un DAG, on obtient un ordre optimal sur le graphe en concaténant des ordres optimaux sur les composantes fortement connexes insérés dans un ordre d'effeuillage.*

Les parties suivantes auront donc pour objectif d'accélérer les algorithmes de calcul d'ordre optimal sur les graphes orientés fortement connexes. Nous chercherons notamment à accélérer l'algorithme dynamique présenté plus haut.

2 Contractions et théorèmes

2.1 Contraction de graphes et extension d'ordres

Définition 5 : Contraction :

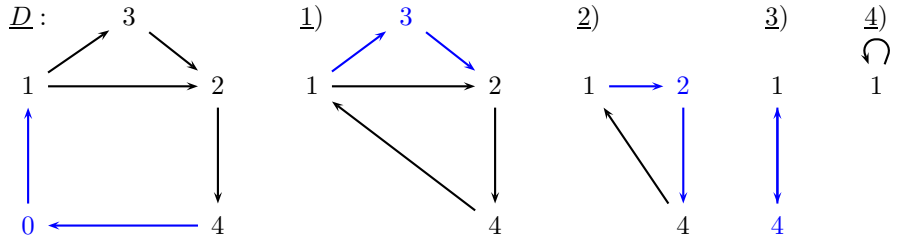
- Contracter un DAG de degré sortant 1 dans un graphe revient en fait à enlever du graphe tous les sommets de ce DAG et à relier les sommets qui pointent vers le DAG (que nous appellerons les sources du DAG) au voisin sortant (que nous appellerons cible). On dira que la contraction crée des arcs parallèles ssi une des sources du DAG était déjà reliée à la cible. On dira qu'elle crée une boucle si la cible du DAG est aussi une source.
- Soit D un graphe orienté. On appelle graphe contracté de D et on note D^* le graphe obtenu en contractant tous les DAG d de D ayant une unique cible v et dont toutes les sources u vérifient une des propriétés suivantes :
  1. il existe un chemin de u à v ne passant que par des sommets de d de degré sortant 1.

4. Certains algorithmes, comme l'algorithme de Tarjan, permettent de calculer en temps linéaire en la taille du graphe ses composantes fortement connexes.

2. $u = v$ où u pointe vers au moins deux sommets de $d \cup \{v\}$

On peut également obtenir D^* sans faire de recherche de DAG, en contractant les sommets un par un. Si la contraction d'un sommet x de degré sortant 1 crée des arcs parallèles ou une boucle sur toutes ses sources, on la fait. Sinon, on ne contracte x que si on n'a pas encore créé d'arcs parallèles ou de boucles partant de x.

Exemple 4
On peut contracter le graphe D à gauche en suivant les étapes indiquées, jusqu'à obtenir le graphe D^* ci-dessous à droite.



À noter qu'à l'étape 1, en contractant le sommet 3, on a créé des arcs parallèles partant de 1. De ce fait, on ne peut donc pas contracter le sommet 1 sur le sommet 4 à l'étape 2, puisque cette contraction ne fait pas apparaître d'arcs parallèles.

Notre contraction revient à contracter le DAG $3 \to 2 \to 4 \to 0$. Nous en avons bien le droit puisque ce DAG a un degré sortant de 1, (il ne pointe que sur 1), et que sa source 1 pointe vers deux sommets du DAG (2 et 3).

Remarque 3
Il est important de tenir compte des éventuelles boucles créées sur les sommets contractés. Elles ne serviront plus une fois la contraction finie, mais un sommet qui pointe vers un voisin extérieur et a une boucle vers lui-même ne peut pas être contracté comme s'il s'agissait d'un sommet de degré sortant 1.

Définition 6 : extension d'un ordre
Soit D un graphe et D^* son contracté. Soit L^* un ordre sur D^*. On dit qu'un ordre L sur D étend L^* ssi on peut l'obtenir en insérant les sommets contractés dans L^*. Autrement dit, pour deux sommets a et b de D^*, si a est avant b dans L^*, il doit rester avant b dans L.

On notera $\overline{L^*}^D$ l'ensemble des ordres sur D qui étendent L^*.

Lemme 1
Soit L un ordonnancement de D et L^* l'unique ordonnancement de D^* tel que $L \in \overline{L^*}^D$. On a alors $\mathrm{vs}_L(D) \geqslant \mathrm{vs}_{L^*}(D^*)$.

Démonstration. On remarque d'abord que L^* est l'ordonnancement de D^* obtenu en enlevant les sommets contractés de l'ordonnancement L. On en déduit :

$$\begin{aligned}
\mathrm{vs}_{L^*}(D^*) &= \max_{i \in [\![1, n^*]\!]} \mathrm{Card}\left[N^+_{D^*}(S_{L^*}(i)) \setminus S_{L^*}(i)\right] \\
&= \max_{i \in [\![1, n]\!]} \mathrm{Card}\left[N^+_{D^*}(S_L(i) \cap V^*) \setminus (S_L(i) \cap V^*)\right] \\
&= \max_{i \in [\![1, n]\!]} \mathrm{Card}[\underbrace{N^+_{D^*}(S_L(i) \cap V^*)}_{\subset N^+_D(S_L(i))} \setminus (S_L(i))] \\
&\leqslant \max_{i \in [\![1, n]\!]} \mathrm{Card}\left[N^+_D(S_L(i)) \setminus S_L(i)\right] = \mathrm{vs}_L(D)
\end{aligned}$$

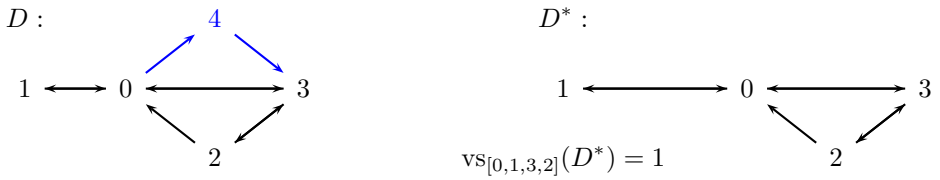
□

Corollaire 1
Si D^* est obtenu en contractant D, alors $\mathrm{vs}(D^*) \leqslant \mathrm{vs}(D)$.

Démonstration. On utilise le lemme précédent sur un ordonnancement optimal L de D. On a alors $\mathrm{vs}(D) = \mathrm{vs}_L(D) \geqslant \mathrm{vs}_{L^*}(D^*) \geqslant \mathrm{vs}(D^*)$. □

Remarque 4

- *La vertex separation d'un graphe et de son contracté peuvent être différentes :*



On voit que $\mathrm{vs}(D^*) = 1$ *alors que* $\mathrm{vs}(D) = 2$. *En effet :*
- *Les préfixes* $[0], [2]$ *et* $[3]$ *ont une mesure d'au moins 2 donc tous les ordres qui commenceront par 0,2 ou 3 auront une mesure d'au moins 2.*
- *Les préfixes* $[1, 0], [1, 2], [1, 3]$ *et* $[1, 4]$ *ont une mesure d'au moins 2.*
- *Les préfixes* $[4, 0], [4, 1], [4, 2], [4, 3]$ *aussi.*

- *On n'obtient pas forcément un ordre optimal sur* D^* *en enlevant d'un ordre optimal sur* D *les sommets contractés. En réutilisant les graphes plus haut,* $[3, 2, 0, 4, 1]$ *est optimal sur* D, $[3, 2, 0, 1]$ *n'est pas optimal sur* D^*.

2.2 Extension optimale

Définition 7

Soit D un graphe et D^* le graphe contracté qui lui est associé. Soit L^* un ordonnancement de D^*. Dans la suite du rapport, on notera $\mathrm{ext}_D(L^*)$ l'ordonnancement de D obtenu en insérant dans L^* les DAG de D contractés, dans l'ordre inverse de celui dans lequel ils ont été contractés en suivant les règles suivantes : pour chaque DAG d à insérer, on parcourt L^* jusqu'à trouver soit une source u de d, soit sa cible v, soit un voisin entrant de v dans D :

- si on trouve en premier une source u, on distingue deux cas :
  1. si $u \neq v$ et s'il existe un chemin de u à v ne passant que par des sommets de degré sortant 1 : on insère après u les sommets de ce chemin (sauf v) dans l'ordre dans lequel ils apparaissent dans le chemin. On insère ensuite les sommets restant de d dans un ordre d'effeuillage.
  2. sinon, on insère d juste avant u, dans un ordre d'effeuillage.
- sinon, si on trouve en premier v ou un voisin entrant de v dans D, on insère d juste après, dans un ordre d'effeuillage.

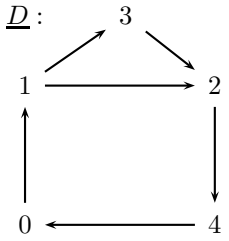
Là encore, il est plus facile de coder la fonction d'extension en raisonnant sur les sommets. On décontracte les sommets dans l'ordre inverse de celui dans lequel ils ont été contractés :

- si on trouve d'abord le voisin sortant de notre sommet, ou un de ses voisins entrants, on met notre sommet juste après.
- si on trouve d'abord un sommet source u :
  - si la contraction de notre sommet x à insérer crée une boucle ou des arcs parallèles, on met x avant u.
  - sinon, on met x après u.

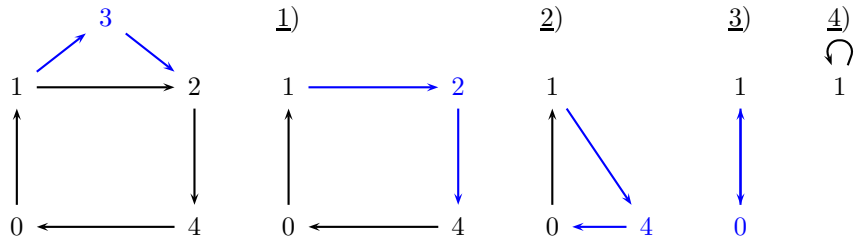
Remarque 5

La fonction d'extension renverra une image différente d'un ordre en fonction de l'ordre dans lequel les DAG/sommets ont été contractés.

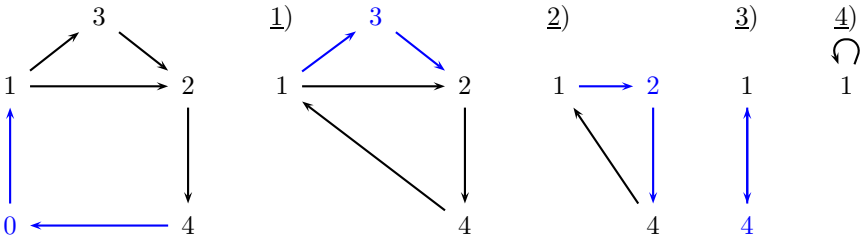
Exemple 5
Reprenons le graphe de l'exemple 1 :



En contractant le DAG $3 \to 2 \to 4 \to 0$, on trouve $\text{ext}([1]) = [0, 4, 2, 3, 1]$. C'est ce qu'on retrouve en contractant les sommets ainsi :



Dans l'exemple 1, nous avions contracté notre DAG de la façon suivante :



On trouve alors l'ordre $[4; 0; 2; 3; 1]$. Toutefois, la contraction de sommet n'étant qu'un cas particulier de contraction de DAG, cet ordre vérifiera aussi les propriétés établies dans les parties suivantes et démontrées dans le cadre des contractions de DAG.

THÉORÈME 4
Soit D un graphe, D^* son contracté et L^* un ordre sur D. $\text{ext}_D(L^*)$ est optimal parmi les ordres qui étendent L^*. Autrement dit :

$$\text{vs}_{\text{ext}_D(L^*)}(D) = \min_{L \in \sigma_{L^*}(V)} \text{vs}_L(D)$$

Démonstration. Voir annexe A □

Remarque 6
On n'obtient pas forcément un ordre optimal sur D en étendant, même de manière optimale, un ordre optimal sur D^*.

Considérons par exemple les graphes D et D^* [5] ci-dessous.

5. Ici, D^* n'est pas entièrement contracté. Il y a bien entendu aussi des contre-exemples dans le cas où D^* est entièrement contracté, mais il faut envisager des graphes qui ont plus de sommets.

$L^* = [3, 0, 2]$ *est optimal sur* D^**, on l'étend en* $L = [3, 0, 1, 2]$ *qui n'est pas optimal sur* D*. Si on avait pris* $L^* = [3, 2, 0]$*, on aurait pu trouver un ordre optimal, mais on n'a a priori aucun moyen de le savoir à l'avance.*

Proposition 2
Pour tout graphe orienté D, $\text{vs}(D^*) \leqslant \text{vs}(D) \leqslant \text{vs}(D^*) + 1$.

Démonstration. voir annexe B □

THÉORÈME 5
Si D^* *est le contracté de* D*, il existe un ordonnancement optimal sur* D *de la forme* $\text{ext}_D(L^*)$ *où* L^* *est un ordre sur* D^**.*

Démonstration. En effet, considérons un ordonnancement optimal L de D. Les $\overline{L^*}^D$ formant une partition de l'ensemble des ordres sur D, il existe L^* tel que $L \in \overline{L^*}^D$ et le théorème 4 nous indique que $\text{vs}_{\text{ext}_D(L^*)}(D) \leqslant \text{vs}_L(D)$ donc que $\text{ext}_D(L^*)$ est optimal. □

Les algorithmes de recherche d'ordre consiste à chercher parmi les $n!$ ordonnancements possibles lequel est le meilleur. Comme nous savons qu'il y a un ordre optimal dans l'image de la fonction ext_D, on peut restreindre nos recherches à $n^*!$ candidats, où n^* est le nombre de sommets de D^*. Dans la suite, nous verrons comment adapter nos algorithmes pour restreindre les recherches à l'image de ext, et ainsi, par exemple, faire passer l'algorithme dynamique d'une complexité en $O(2^n)$ à $O(2^{n^*})$.

3 Algorithme

3.1 Preprocessing

Pour pouvoir utiliser les contractions pour accélérer des algorithmes, nous aurons besoin de généraliser notre fonction d'extension pour qu'elle puisse prendre en argument des préfixes d'un ordre L^* et renvoyer un préfixe de $\text{ext}_D(L^*)$.

Définition 8
On prolonge ext_D aux sous-ordres de D^* de la façon suivante : on appelle l^* le préfixe donné en argument, pour tout DAG contracté, on parcourt l^* jusqu'à trouver soit une source u de notre DAG, soit sa cible v, soit un voisin entrant de sa cible, comme on le fait dans la fonction d'extension. Si on trouve, on insère d de la façon décrite lors de la définition de notre fonction d'extension, sinon, on ne rajoute pas d. Notre préfixe l sera ainsi le plus long préfixe commun aux images par ext_D des ordres admettant comme préfixe l^*.

Évidemment, s'il fallait appliquer cette fonction à chacun des 2^{n^*} préfixes qu'on étudie, on prendrait beaucoup de temps.

Proposition 3
Pour deux préfixes p et q de D^*, on a (en notant @ le symbole de concaténation) :

$$\text{ext}_D(p@q) = \text{ext}_D(p)@\left(\text{ext}_D(q) \setminus \text{ext}_D(p)\right)$$

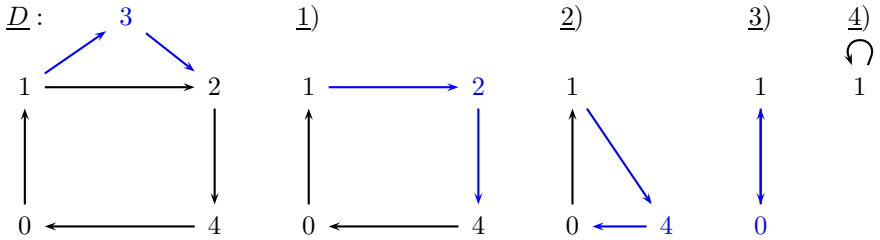
Il suffit donc de construire l'image par ext_D des ordres réduits à un sommet de D^*, on pourra en déduire l'image de n'importe quel préfixe rapidement. La construction de ces préfixes peut se faire en même temps que la contraction en procédant de la façon suivante :

Algorithme 1 : Preprocessing

Entrées : Un graphe D à n sommets
Sorties : Le graphe D^* et la liste des préfixes associés à chaque sommet de D^*

Soit Préfixe un tableau de taille n
pour *i de 0 à $n-1$ (en considérant que les sommets de D sont étiquetés de 0 à $n-1$)* **faire**
    Préfixe.(i) :=[i] (le préfixe associé à chaque sommet est, dans un premier temps, le sommet lui-même)
pour *tout sommet x de D de degré sortant 1* **faire**
    Soit l la liste des voisins entrants de x, soit v son voisin sortant
    Soit new_loop :=$(l \subset (N_D^-(v) \cup \{v\}))$(booléen traduisant que la contraction crée une boucle ou des arcs parallèles sur toutes les sources)
    Soit no_former_loop :=(x ==Préfixe(x).(0)) (x en tête de son préfixe traduit qu'on n'a pas créé d'arcs parallèles ou de boucles sur x)
    si *new_loop ou no_former_loop (on peut alors contracter)* **alors**
        pour $t \in (l \cup N_D^-(v) \cup \{v\}) \setminus x$ **faire**
            si *$t \in l$ et si $t \in N_D^-(v) \cup \{v\}$* **alors**
                Préfixe(t) :=Préfixe(x) @ (Préfixe(t)\Préfixe(x)) (il est important de ne pas mettre les éléments du préfixe de x après t, puisque c'est x qui apparaîtra en premier)
            sinon
                Préfixe(t) :=Préfixe(t) @ (Préfixe(x)\Préfixe(t)) (@ est ici le symbole de concaténation)
        on élimine x de D ainsi que tous les arcs partant de x ou allant vers x
        pour *u dans l* **faire**
            si $(u, v) \notin D$ **alors**
                on rajoute (u, v) à D
retourner *D et Préfixe*

Exemple 6
On reprend le graphe de l'exemple 1 :



0. Au début, le tableau des préfixes vaut $[0 : [0]; 1 : [1]; 2 : [2]; 3 : [3]; 4 : [4]]$.
1. On contracte ensuite le sommet 3 en créant des arcs parallèles partant de 1. Les seuls préfixes qui changeront sont ceux de 1 (la source) et 2 (la cible), la cible n'ayant pas d'autres voisins entrants. 3 ira donc en tête du préfixe de 1 et à la fin du préfixe de 2. Notre tableau devient alors $[0 : [0]; 1[3; 1]; 2 : [2; 3]; 4 : [4]]$. Le préfixe de 3 n'évoluera plus et ne servira pas. Pour plus de lisibilité, on ne le réécrira pas à chaque étape dans le tableau.
2. On contracte ensuite le sommet 2. Les préfixes qui changent sont ceux de 1 (la source) et 4 (la cible). Le préfixe de 2, $[2, 3]$ ira donc à la fin du préfixe de 4. Notre contraction n'a pas créé d'arcs parallèles ou de boucles, mais on remarque que 1 est source et qu'il n'est pas en tête de son préfixe, ce qui veut donc dire qu'on a créé des arcs parallèles partant de 1. On mettra donc le préfixe de 2 en tête du préfixe de 1. Comme 3 figure déjà dans le préfixe de 2, on ne remet pas celui qu'on voit dans le préfixe de 1. Le tableau devient : $[0 : [0]; 1 : [2; 3; 1]; 4 : [4; 2; 3]]$.
3. On contracte ensuite le sommet 4. Avec le même raisonnement que précédemment, le tableau devient $[0 : [0, 4]; 1 : [4; 2; 3; 1]]$.

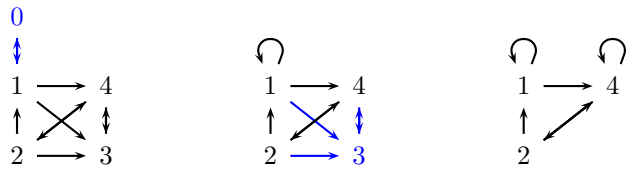
4. Après la contraction de 0, on trouve : $[1 : [0; 4; 2; 3; 1]]$.

Ici, on obtient du même coup l'image de [1] qui est le seul ordre sur D^*. L'ordre $[0; 4; 2; 3; 1]$ est donc optimal.

3.2 Processing

On utilise ensuite l'algorithme de calcul d'ordre optimal pour trouver l'ordre sur D^* dont la mesure de l'image par ext est la plus petite possible. L'image par ext_D de l'ordre sur D^* sélectionné est alors un ordre optimal sur D.

Exemple 7
On considère le graphe D contracté comme indiqué ci-dessous. Nous allons chercher un ordre optimal sur D en utilisant l'algorithme dynamique décrit en partie 1.



On trouve comme tableau de préfixe $[1 : [3; 0; 1], 2 : [3; 2; 0], 4 : [3; 4]]$.

On va ensuite comparer les préfixes éventuels de L. Comme on cherche L dans l'image d'ext, on n'étudiera que les images des préfixes sur L^*.

préfixe de D^*	préfixe de D correspondant	coût (dans D)
[]	[]	0
[1]	$[3, 0, 1]$	2
[2]	$[3, 2, 0]$	3
[4]	$[3, 4]$	1
$[1, 2]/[2, 1]$	$[3, 0, 1, 2]/[3, 2, 0, 1]$	2/3 on choisit donc $[1, 2]$
$[1, 4]/[4, 1]$	$[3, 0, 1, 4]/[3, 4, 0, 1]$	2/2 on choisit arbitrairement $[1, 4]$
$[2, 4]/[4, 2]$	$[3, 2, 0, 4]/[3, 4, 2, 0]$	3/1 on choisit $[4, 2]$
$[1, 2, 4]$	$[3, 0, 1, 2, 4]$	2
$[1, 4, 2]$	$[3, 0, 1, 4, 2]$	2
$[4, 2, 1]$	$[3, 4, 2, 0, 1]$	1

Et on en déduit donc que [3,4,2,0,1] est un ordre optimal sur D.

Remarque 7
On remarque que la taille d'un préfixe de L ne dépend pas seulement de la taille de son antécédent dans L^*, mais aussi des sommets qui en font partie (l'image de [2] contient ici plus de sommet que l'image de [4]). Par contre, l'ordre dans lequel apparaissent les sommets du préfixe de L^* n'a aucune importance ni sur la taille de son image dans L, ni sur les sommets qui la composent (les images de $[1, 4]$ et $[4, 1]$ contiennent exactement les mêmes sommets, seuls l'ordre dans lequel ils apparaissent diffère). On compare donc à chaque étape des préfixes de L de même taille et contenant les mêmes sommets.

Notons enfin qu'il faut être vigilant quand on évalue la mesure d'un préfixe. Sur l'exemple, on dit que le préfixe $[3, 2, 0]$ a une mesure de 3. On atteint en fait ce maximum sur $[3, 2]$, la mesure locale au moment de l'ajout du 0 valant en fait 2. Ceci dit, il n'est pas obligatoire de calculer la mesure locale à chaque étape.

Proposition 4
Le mesure de $\text{ext}(a@b)$ est en fait le maximum entre la mesure de a (qu'on connaît déjà), la mesure locale juste avant l'ajout de b et la mesure locale juste après l'ajout de b.

Démonstration. Conséquence des lemmes 2, 3 et 4 énoncés et démontrés en annexe A. Dans le seul cas où la mesure peut dépasser la valeur atteinte en b, le maximum est atteint juste avant b. □

4 Aller plus loin ?

Une façon d'augmenter encore l'efficacité de l'algorithme serait de contracter plus de sommets. Cette partie propose quelques pistes de recherche.

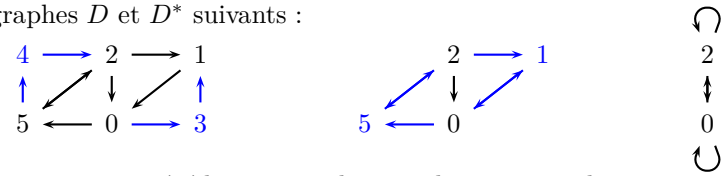
4.1 Contracter tous les sommets de degré sortant 1 ?

Dans le cas d'un DAG $0 \to 1 \to 2$, notre algorithme interdit de contracter le 1 si on a déjà créé des arcs parallèles dessus. Voyons plus en détail pourquoi nous avons interdit cette contraction.

Si on ne tient pas compte des arcs parallèles créés sur le 1, c'est à dire si on place 1 après le premier rencontré entre 0 et 2, l'algorithme ne sera pas valide, puisque dans le graphe décontracté, 1 a plusieurs voisins sortant.

Une idée serait de le contracter en tenant compte du fait qu'il va créer des arcs parallèles sur le 0, c'est à dire en le mettant devant le 0 ou derrière le 2, mais là encore, ça ne marche pas. En effet, nous ne sommes pas exactement dans le cas couvert par le lemme 4 (voir annexe A), car si notre DAG était placé après u, l'augmentation de la mesure n'aurait pas forcément lieu à l'étape où on rencontre u, et elle peut parfois avoir lieu à un moment où la mesure locale est plus faible.

Exemple 8
Considérons les graphes D et D^* suivants :

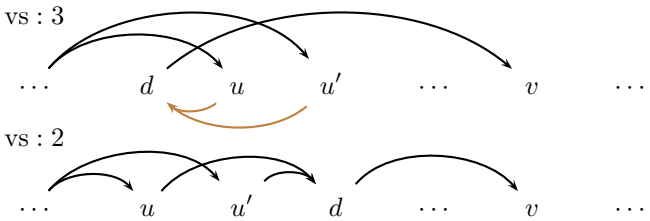


Nos règles de contraction précédentes interdisaient de contracter le sommet 5. En effet, sa contraction ne crée pas d'arcs parallèles sur le 0 et la contraction du 4 avait créé des arcs parallèles partant du 5. Voyons les conséquences de cette contraction.

L'ordre renvoyé par l'algorithme sera : $[4, 5, 1, 3, 0, 2]$.

On constate alors que $\text{vs}_{[4,5,1,3,0,2]}(D) = 2$ alors que $\text{vs}_{[1;3;0;2;4;5]} = 1$ et la solution renvoyée par l'algorithme n'est pas optimale. On voit que dans la solution optimale proposée, 4 et 5 font également augmenter localement la mesure, mais à un moment où elle est plus faible.

Pour pouvoir faire cette contraction, il faudra donc une fonction d'extension plus sophistiquée. Un autre risque serait alors de ne plus pouvoir faire de programmation dynamique. En effet, considérons un DAG d de voisins entrant u et u' et de voisin sortant v. Si on trouve u en premier dans l'ordre, on serait tenté de placer d juste avant. La solution est en effet optimale si on trouve v avant u', mais pas dans le cas contraire



Ainsi, pour pouvoir faire cette contraction, on devrait peut-être mettre au point une fonction d'extension qui, si on voit u en premier, attend de savoir si le suivant sera u' ou v avant de placer d. Le problème sera alors que les extensions d'un ordre qui a pour préfixe u n'auront pas forcément de préfixe commun, et l'algorithme de programmation dynamique s'effondrerait.

Toutefois, cet exemple montre seulement qu'une fonction d'extension qu'on peut prolonger aux sous-ordres ne respectera pas le théorème 4 ($\text{ext}(L^*)$ optimal parmi les ordres qui étendent L^*). En fait, le théorème dont vient la validité de l'algorithme est le théorème 5 (existence d'un ordre optimal dans l'image de ext), et le théorème 4, bien que suffisant, n'est en aucun cas nécessaire.

4.2 Et pour le degré entrant 1 ?

On peut en effet remarquer qu'un graphe D et son graphe transposé ont même *vertex separation*. En effet, si $X_1, \cdots, X_n$ est une *path decomposition* sur D, $X_n, \cdots, X_1$ sera une *path decomposition* sur le transposé de D, et la largeur sera évidemment la même. Si on remarque que notre graphe a plus de DAG de degré entrant 1 que de DAG de degré sortant 1, on peut donc appliquer notre algorithme sur le graphe transposé pour un résultat plus rapide. Toutefois, il serait intéressant de pouvoir dans le même algorithme contracter les DAG de degré entrant et ceux de degré sortant 1. Il faudrait donc imaginer une fonction d'extension adaptée, et qui comme avant, permet la programmation dynamique.

Conclusion

Nous avons construit un algorithme efficace pour accélérer le calcul de *vertex separation* et de *path decomposition*, grandeurs qui ont de nombreuses applications. L'algorithme procède en contractant des sommets pour pouvoir faire le calcul en un temps qui dépendra de la taille d'un graphe plus petit que le graphe de départ. De plus, l'algorithme ne se contente pas de construire un graphe plus petit dont la *vertex separation* est la même, il reste valable même si les contractions effectuées modifient la *vertex separation* du graphe.

De plus, la méthode qui consiste à chercher un ordre sur D^* qui minimise la mesure dans D de son image par la fonction d'extension a l'avantage de pouvoir être utilisée pour accélérer d'autres algorithmes de recherche d'ordre optimal que celui présenté dans la partie 1.

Le gain de temps dû aux contractions est difficile à évaluer puisqu'il dépend beaucoup du graphe donné en entrée, mais l'algorithme accéléré est forcément au moins aussi rapide que l'algorithme basique. En effet, dans les cas où il n'y a pas de contractions, le preprocessing est instantané et la partie principale est la même qu'avec l'algorithme classique. Sinon, pour un algorithme en $O(2^n)$, chaque sommet contracté divise par deux le temps de calcul de la partie principale. À l'origine, cet algorithme a été conçu pour être appliqué sur des graphes de dépendance. Ces graphes peuvent être grands, mais leur densité n'augmente pas très rapidement en fonction du nombre de sommets, et la probabilité de trouver des DAG qu'on peut contracter va donc être forte. De plus, le fait de pouvoir faire des contractions qui vont créer des arcs parallèles permet de contracter sans augmenter la densité du graphe obtenu et permet donc souvent des contractions en chaîne.

Il reste toutefois de nombreuses pistes à explorer pour pouvoir généraliser ou adapter ce qui a été fait ici. Le fait de pouvoir faire des contractions même si elles modifient la *vertex separation* des graphes ouvre énormément de portes et le simple fait de rajouter quelques contractions peut occasionner de grands gains de temps.

Remerciements

Sincères remerciements à David Coudert, Nicolas Nisse et à toute l'équipe Mascotte pour le temps qu'ils m'ont consacré et pour le stage très agréable que j'ai effectué.

Références

[1] Stefan Arnborg. Efficient algorithms for combinatorial problems with bounded decomposability - a survey. *BIT*, 25(1) :2–23, 1985.

[2] Dhritiman Banerjee and Biswanath Mukherjee. Wavelength-routed optical networks : linear formulation, resource budgeting tradeoffs, and a reconfiguration study. *IEEE/ACM Trans. Netw.*, 8(5) :598–607, 2000.

[3] János Barát. Directed path-width and monotonicity in digraph searching. *GRAPHS AND COMBINATORICS*, 22 :161–172, 2006.

[4] H. L. Bodlaender and F. V. Fomin. Approximation of pathwidth of outerplanar graphs. Technical Report UU-CS-2000-23, Department of Information and Computing Sciences, Utrecht University, 2000.

[5] Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. A note on exact algorithms for vertex ordering problems on graphs. *Theory Comput. Syst.*, 50(3) :420–432, 2012.

[6] Xiaowen Chu, Tianming Bu, and Xiang-Yang Li. A study of lightpath rerouting schemes in wavelength-routed wdm networks. In *ICC*, pages 2400–2405, 2007.

[7] David Coudert, Florian Huc, Dorian Mazauric, Nicolas Nisse, and Jean-Sébastien Sereni. Reconfiguration of the Routing in WDM Networks with Two Classes of Services. In *Conference on Optical Network Design and Modeling (ONDM)*, Braunschweig, Germany, 2009.

[8] David Coudert and Jean-Sébastien Sereni. Characterization of graphs and digraphs with small process numbers. *Discrete Applied Mathematics*, 159(11) :1094–1109, 2011.

[9] Jonathan A. Ellis, Ivan Hal Sudborough, and Jonathan S. Turner. The vertex separation and search number of a graph. *Inf. Comput.*, 113(1) :50–79, 1994.

[10] Nancy G. Kinnersley. The vertex separation number of a graph equals its path-width. *Inf. Process. Lett.*, 42(6) :345–350, 1992.

[11] Nimrod Megiddo, S. Louis Hakimi, M. R. Garey, David S. Johnson, and Christos H. Papadimitriou. The complexity of searching a graph. *J. ACM*, 35(1) :18–44, 1988.

[12] Neil Robertson and Paul D. Seymour. Graph minors. i. excluding a forest. *J. Comb. Theory, Ser. B*, 35(1) :39–61, 1983.

[13] Konstantin Skodinis. Computing optimal linear layouts of trees in linear time. In *ESA*, pages 403–414, 2000.

[14] Boting Yang and Yi Cao. Digraph searching, directed vertex separation and directed pathwidth. *Discrete Applied Mathematics*, 156(10) :1822–1837, 2008.

A Théorème 4

THÉORÈME 4
Soit D un graphe, D^* son contracté et L^* un ordre sur D. $\text{ext}_D(L^*)$ est optimal parmi les ordres qui étendent L^*. Autrement dit :

$$\text{vs}_{\text{ext}_D(L^*)}(D) = \min_{L \in \sigma_{L^*}(V)} \text{vs}_L(D)$$

A.1 Cas où la contraction porte sur un seul DAG

Dans toute cette sous-partie, D sera un graphe à n sommets, d un dag de D à n_d sommets et de degré sortant 1, D' le graphe obtenu en contractant d dans D, n' le nombre de sommets de D' et L' un ordre sur D'. On appellera v le voisin sortant de d dans D.

Posons $x = \min \left[\{L'^{-1}(v)\} \cup \bigcup_{w \in N_D^-(v)} \{L'^{-1}(w)\} \right]$ et $y = \min \bigcup_{u \in N_D^-(d)} \{L'^{-1}(u)\}$. Soit l un ordre d'effeuillage sur d.

Lemme 2
Si $x < y$, alors en posant $L = L'[1:x].l.L'[x+1, n']$, on a $vs_L(D) = vs_{L'}(D')$. [6]

Démonstration. Montrons que pour tout $j \in [\![1, n]\!]$, il existe $k \in [\![1, n']\!]$ tel que la mesure locale dans L à l'étape j soit égale à la mesure locale dans L^* à l'étape k.

- Pour $j \leqslant x$, $L[1:j] = L'[1:j]$. De plus, comme $j < y$, il n'y a pas de voisins entrants de d dans $L[1:j]$ et $\forall u \in S_L(j), N_D^+(u) = N_{D'}^+(u)$. Ainsi, $\forall j \leqslant x, N_D^+(S_L(j)) = N_{D'}^+(S_{L'}(j))$.
- Pour $j > x + n_d$:
  $S_L(j) = S_{L'}(j - n_d) \cup d$ et comme v est le seul voisin sortant de d, $N_{D'}^+(S_{L'}(j-n_d)) \subset N_D^+(S_L(j)) \subset N_{D'}^+(S_{L'}(j-n_d)) \cup d$. Comme $d \in S_L(j)$, $N_D^+(S_L(j)) \setminus S_L(j) = N_{D'}^+(S_{L'}(j-n_d)) \setminus S_L(j-n_d)$.
- Pour $j \in [\![x+1, x+n_d]\!]$:
  un sommet de d ne peut avoir comme voisin sortant que v ou d'autres sommets de d. Comme les sommets de d sont listés dans un ordre d'effeuillage et qu'à l'étape x, v est soit inséré, soit dans le voisinage sortant des sommets insérés,
  $\forall j \in [\![x+1, x+n_d]\!], N_D^+(S_L(j)) \setminus S_L(j) = N_{D'}^+(S_{L'}(x)) \setminus S_{L'}(x)$

et finalement,

$$\begin{aligned} \text{vs}_{L'}(D') &= \max_{1 \leqslant j \leqslant n'} \text{Card}(N_{D'}^+(S_{L'}(j)) \setminus S_{L'}(j) \\ &= \max_{1 \leqslant j \leqslant n} \text{Card}(N_D^+(S_L(j)) \setminus S_L(j)) = \text{vs}_L(D) \end{aligned}$$

Les schémas suivants résument la situation :

FIGURE 9 – Si on trouve d'abord v, on ne rajoute que des arcs droite-gauche (les arcs internes à d ne sont pas représentés, mais d étant inséré dans un ordre d'effeuillage, ce ne sont que des arcs droite-gauche)

6. on est dans le cas où on rencontre en premier v ou un de ses voisins sortants dans L'. On insère d juste après, et on veut montrer que la *vertex separation* se conserve.

FIGURE 10 – Si on trouve d'abord un voisin entrant w de v, le seul arc gauche-droite rajouté, n'a aucune incidence

Dans tous les cas, la *vertex separation* se conserve. □

Lemme 3
Si $y < x$ et s'il existe un chemin $c_1, \cdots, c_l$ de $L'[y]$ à v ne passant que par des sommets de d de degré sortant 1, alors en posant $L = L'[1:y].c_1.c_2.\cdots.c_l.(l \setminus c).L'[y+1:n']$, on a $\mathrm{vs}_L(D) = \mathrm{vs}_{L'}(D')$. [7]

Démonstration. Montrons que pour tout $j \in [\![1, n]\!]$, il existe $k \in [\![1, n^*]\!]$ tel que la mesure locale dans L à l'étape j soit égale à la mesure locale dans L^* à l'étape k.

- Pour $j < y$, $L[1:j] = L'[1:j]$. De plus, comme $j < y$, il n'y a pas de voisins entrants de d dans $L[1:j]$ et $\forall w \in S_L(j), N_D^+(w) = N_{D'}^+(w)$. Ainsi, $\forall j < y, N_D^+(S_L(j)) = N_{D'}^+(S_{L'}(j))$.
- À l'étape y, on insère une source u de d dans L et dans L'. La seule différence est que dans L, au lieu de pointer vers v, u pointe vers c_1. Comme ni v ni c_1 n'ont déjà été rencontrés dans nos ordres, on a $\mathrm{Card}(N_D^+(S_L(y)) \setminus S_L(y)) = \mathrm{Card}(N_{D'}^+(S_{L'}(y)) \setminus S_{L'}(y))$.
- Lors de l'insertion du chemin, c'est à dire pour $j \in [\![y+1, y+l]\!]$, en posant $x_0 = u$ et $x_{l+1} = v$, on a $\forall j \in [\![y+1, y+l]\!], (N_D^+(S_L(j)) \setminus S_L(j)) = (N_D^+(S_L(j-1)) \setminus S_L(j-1)) \cup \{x_{j-i+1}\} \setminus \{x_{j-i}\}$
  On montre alors par récurrence que $\mathrm{Card}(N_D^+(S_L(j)) \setminus S_L(j)) = \mathrm{Card}(N_D^+(S_L(y)) \setminus S_L(y)) = \mathrm{Card}(N_{D'}^+(S_{L'}(y)) \setminus S_{L'}(y))$.
- Pour $j \in [\![y+l+1, y+n_d]\!]$: une fois le chemin inséré, on insère les autres sommets de d dans un ordre d'effeuillage, après x_l qui pointe vers v. Le lemme 2 nous indique donc que la *vertex separation* n'augmentera pas lors de l'insertion de l.
- Pour $j > y + n_d$: le voisinage sortant de $S_L(y+n_d)$ est égal au voisinage sortant de $S_{L'}(y) \cup d$. On insère ensuite les mêmes sommets dans le même ordre. On a alors $\forall j > y, N_{D'}^+(S_{L'}(j)) \setminus (S_{L'}(j)) = N_D^+(S_L(j+n_d)) \setminus (S_L(j+n_d))$.

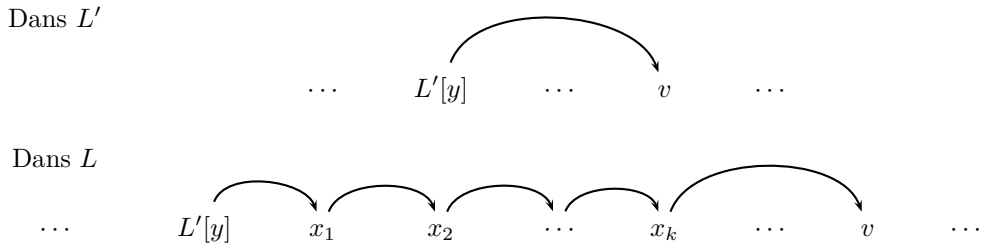


FIGURE 11 – On voit que la *vertex separation* se conserve.

□

Lemme 4
Si $y \leqslant x$ et si $L'[y] = v$ ou $L'[y]$ pointe vers au moins deux sommets de $d \cup \{v\}$, alors en posant $L = L'[1:y-1].l.L'[y:n']$, on a :

1. pour tout sommet w de D', la mesure locale en w dans L restera la même que dans L'.
2. si la vs augmente, elle augmentera où qu'on insère d.

Démonstration. On pose $u = L'[y]$

1. Si w est avant u : comme aucun sommet avant w ne pointe vers d, la mesure locale sera la même que dans L'.

7. On est dans le cas où on trouve en premier un voisin entrant de d différent de v. On insère d juste après et on veut prouver que la *vertex separation* se conservera.

Pour w à partir de u : d est inséré donc le fait que certains sommets pointent vers d dans D et pas dans D' ne fera pas augmenter la mesure. Le seul voisin sortant de d est v mais u pointe déjà vers v dans L'. La mesure locale en w ne changera donc pas.

2. Comme d a un degré sortant de 1, qui est donc inférieur ou égal au degré sortant de u dans D', la mesure en d ne peut dépasser celle en u que si u a un degré sortant de 1 dans D' et si u a un voisin entrant placé avant lui dans L'. En effet, on ajoutera v au voisinage sortant avant d'en enlever u.

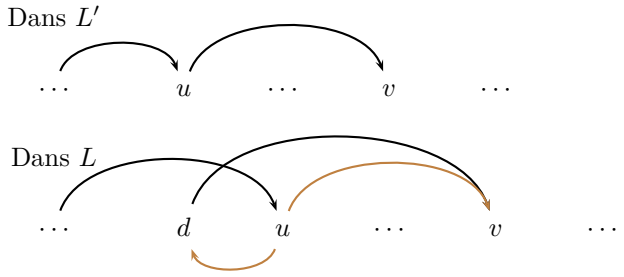


FIGURE 12 – Le seul cas où la mesure augmente est le cas où on a un voisin entrant de u avant u dans L'.

Dans ce cas, notre mesure atteint donc (1+la mesure dans L' en u). Il nous faut alors montrer que notre mesure atteindra cette valeur où qu'on place d.

- Si on place un voisin entrant de v avant u : on atteindra encore v avant d'enlever u et le voisinage sortant des sommets avant u sera le même dans L et dans L'. La mesure augmentera donc toujours.
- Si on met tous les voisins sortant de u après u :
  - Si $u = v$, la boucle sur d ne fait pas augmenté la mesure en v dans L', alors que l'arc de v vers d comptera dans L. La mesure dans L sera donc supérieur à la mesure dans L'.

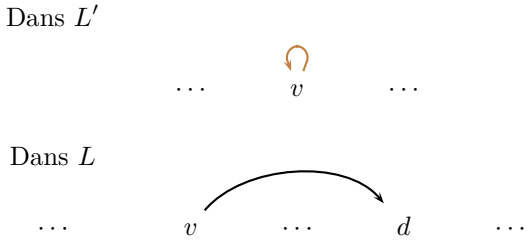


FIGURE 13 – La situation si on insère d après u et si $u = v$

  - sinon, par hypothèse, u pointe vers deux sommets de $d \cup \{v\}$ et qu'on n'a rencontré aucun de ces sommets au moment où on rencontre u, la mesure en u dans L sera supérieure à la mesure en u dans L' où u n'a qu'un degré sortant de 1.

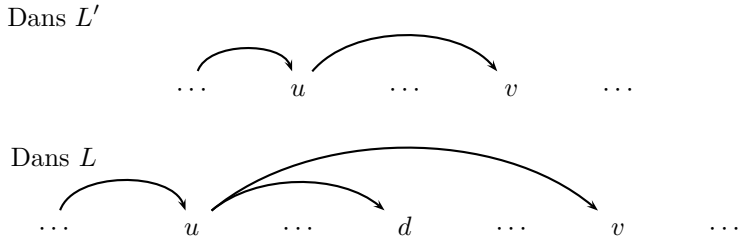


FIGURE 14 – La situation si on insère d après u et si $u \neq v$

et on voit qu'au moment d'insérer u, notre mesure vaudra toujours 1 de plus que sur L', l'augmentation ayant toujours lieu au même endroit ce qui garantit que le maximum sera le même.

- Si on met tous les voisins entrants de v après u et un voisin sortant x de u avant u : comme D est fortement connexe, il existe un chemin de x à v. Comme v est le seul voisin sortant de d, ce chemin ne passe que par des sommets de d. Comme tous les voisins entrants de v ont été placés après u, on rajoute bien un arc d'un sommet à gauche de d vers un sommet à droite de d et on fait ainsi augmenter la mesure, toujours au même endroit.

La *vertex separation* augmente donc dans tous les cas.

□

A.2 Cas général

On note $L = \text{ext}_D(L^*)$

- si $\text{vs}_L(D) = \text{vs}_{L^*}(D^*)$, alors par le lemme 1, L est optimal sur D
- sinon : il y a une étape d'insertion à laquelle on fait augmenter la mesure. Les lemmes 2, 3 et 4 nous indique que cette augmentation est due à l'insertion d'un DAG d juste avant sa source u. Le lemme 4 nous indique qu'à cette étape de décontraction, la mesure aurait atteint (1+la mesure en u au moment de l'insertion de d) où qu'on insère d. On remarque alors qu'aucune des insertions déjà effectuées n'a pu faire augmenter la mesure en u et que la mesure en u au moment d'insérer d est égale à la mesure en u dans L^*. On est donc dans un cas où la mesure augmente quoiqu'on fasse.

Dans tous les cas, $\text{ext}_D(L^*)$ est optimal parmi les ordres qui étendent L^*.

On remarque ici que l'ordre dans lequel les contractions ont été faites n'intervient pas dans la démonstration.

B Proposition 2

Proposition 2
Pour tout graphe orienté D, $\text{vs}(D^*) \leqslant \text{vs}(D) \leqslant \text{vs}(D^*) + 1$.

Démonstration. La première inégalité est fournie par le lemme 1.

Soit D un graphe et D^* son contracté.
Nous savons par les lemmes 2, 3 et 4 que le seul cas où la mesure augmente est celui où on insère un sommet x juste avant un voisin entrant u et où il y a un voisin entrant t de u placé avant u dans l'ordre provisoire au moment de l'insertion de x. La mesure en x atteint alors 1+la mesure en u.

Supposons par l'absurde que la mesure en x atteigne $\text{vs}(D^*) + 2$. On sait alors que la mesure en u vaut $\text{vs}(D^*) + 1$. Les lemmes 2, 3 et 4 nous indiquent que les insertions ne changent pas la mesure locale en les sommets de D^* donc u est un sommet qui a été inséré.

Comme il y a un voisin entrant t de u avant u, on est dans le cas où on insère u après son voisin entrant, donc la contraction de u ne crée ni boucles ni arcs parallèles. Or x a été inséré avant u donc sa contraction a créé des arcs parallèles sur u. Nos règles de contraction aurait alors interdit la contraction de u, d'où la contradiction. □